IDT79RC36100
Highly Integrated RISController
Hardware User's Manual

Version 2.1
August 1998

2975 Stender Way, Santa Clara, California 95054
Telephone: (800) 345-7015 • TWX: 910-338-2070 • FAX: (408) 492-8674
Printed in U.S.A.

About This Manual

Notes

This hardware user's manual provides updated functional overviews and operational details on the IDT79RC36100 Integrated RISController™. As noted below, a MIPS-I architectural overview is included. However, for more detailed software information or descriptions on individual CPU/FPU instructions, refer to the *IDT MIPS Microprocessor Family Software Reference Manual*.

Summary of Contents

Chapter 1, "RC36100 Device Overview," presents an introduction to the operation of the RC36100 device and provides a revised block diagram that illustrates system components. This chapter includes an updated features list, pin description table, logic diagram and performance overview.

Chapter 2, "Instruction Set Architecture," contains an overview of the MIPS-1 architecture set and discusses the programmers' model for this device. This chapter also provides a summary of the CPU registers.

Chapter 3, "Cache Architecture," reviews the fundamentals of general cache operation and provides a discussion on the organization of the RC36100's on-chip caches.

Chapter 4, "Virtual-to-Physical Address Translation and Address Map," explains the devices two operating states (kernal and user) as well as the virtual-to-physical address translation mechanisms provided in the RC36100.

Chapter 5, "Coprocessor 0 Register Set," describes the implementation of the RC36100's system control coprocessor (CPO). This chapter provides complete descriptions of registers and includes field actions and values.

Chapter 6, "Interruption and Exception Handling," discusses the pipeline stages implemented in the RC36100 and provides a structure for understanding the device's exception handling processes. This chapter also contains a section on "Basic Software Techniques For Handling Interrupts."

Chapter 7, "System Bus Interface Unit Overview," provides an operational overview on the RC36100's execution core, as well as operation of the various memory controllers during both internal and external peripheral transactions.

Chapter 8, "Memory Controller," addresses the RC36100's on-chip memory controller interface and includes pin descriptions and timing diagrams. This chapter includes an explanation on how this interface relates to typical external hardware ROMs and RAMs.

Chapter 9, "I/O Controller," provides an overview of the I/O controller interface, pin descriptions and timing diagrams. A discussion on the relationship between this interface and typical external hardware I/O devices is also included.

Chapter 10, "DRAM Controller," presents an overview on the DRAM controller interface of the RC36100 and includes pin descriptions, timing diagrams and a discussion on the relationship between the interface and typical external hardware DRAM systems.

Chapter 11, "Direct Memory Access (DMA) Controller," explains the DMA Controller interface and its relationship to typical internal and external hardware DMA systems. Complete pin definitions, signal information, timing diagrams, and register drawings are also provided.

Chapter 12, "Parallel Input/Output (PIO)," provides an operational overview on the PIO programming interface and includes register descriptions and a table indicating the alternate functions that are mapped to the PIO pins.

Chapter 13, "Peripheral Expansion Interrupt Controller," overviews the features and operation of this controller. A block diagram as well as pin and register descriptions are also provided.

Chapter 14, "Timers," overviews the features and operation of the RC36100's timer programming interface. This chapter also provides pin descriptions and a discussion on how these timers relate to typical internal and external systems.

Chapter 15, "Serial Ports," provides an operational overview on the RC36100's two independent serial port channels and includes a block diagram, register descriptions and pin definitions.

Notes

Chapter 16, "Bidirectional Parallel Port," explains the interfacing functions of the bidirectional parallel port and provides information on the various modes and options available. Register definitions and a system connection example are also included.

Chapter 17, "Reset Initialization and Input Clocking," details the RC36100's initialization selectable features and discusses the processor's required reset sequence.

Chapter 18, "Debug Mode Features," describes features that have been included to facilitate the debugging of RC36100-based systems.

Where To Find More Product Information

Details on this device's electrical interface can be found in the product's data sheet, which also includes packaging, pin-out, and ordering information.

For information on development tools and how to use this product in various applications, refer to IDT's on-line library of data sheets, application notes, evaluation board manuals, software reference manuals, and the IDT Advantage Program Guides.

Your local IDT sales representative can help you identify and use any of these resources.

Table of Contents

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

List of Tables

Notes

Table No.	Table Title	Page
Table 1.1	RC36100 Pin Descriptions	1-8
Table 2.1	Instruction Set Mnemonics	2-5
Table 2.2	RC36100 CP0 Registers	2-8
Table 2.3	Big-Endian (32-bit memory system)	2-12
Table 2.4	Byte Addressing in Load/Store Operations (32-bit memory)	2-12
Table 2.5	Big-Endian (16-bit memory system)	2-13
Table 2.6	Byte Addressing in Load/Store Operations (16-bit memory)	2-13
Table 2.7	Load and Store Instructions	2-14
Table 2.8	ALU Immediate Operations	2-15
Table 2.9	Three Operand Register-Type Operations	2-16
Table 2.10	Shift Operations	2-17
Table 2.11	Multiply and Divide Operations	2-17
Table 2.12	Jump Instructions	2-18
Table 2.13	Branch Instructions	2-19
Table 2.14	Special Instructions	2-19
Table 2.15	Co-Processor Operations	2-20
Table 2.16	System Control Co-Processor (CP0) Operations	2-21
Table 2.17	Opcode Encoding	2-23
Table 3.1	Instruction Cache to Address Mapping under Various Cache Locking Conditions	3-11
Table 3.2	Data Cache to Address Mapping under Various Cache Locking Conditions	3-12
Table 4.1	Virtual and Physical Address Relationships in Base Versions	4-4
Table 4.2	RC36100 On-Chip Resources and Address Map	4-5
Table 4.3	Example: FIFO load code using FCM memory space	4-6
Table 5.1	RC36100 CPO Register Addresses	5-2
Table 5.2	RC36100 Cache Configuration Register Lock Field	5-3
Table 5.3	RC36100 DBlockRefill Field	5-3
Table 5.4	RC36100 D-Cache Index Control Field	5-3
Table 5.5	RC36100 Halt Field	5-4
Table 5.6	RC36100 I-Cache Index Control Field	5-4
Table 5.7	RC36100 Reduced Frequency Mode Field	5-4
Table 5.8	RC36100 ForceDCacheMiss Field	5-5
Table 5.9	RC36100 ForceICacheMiss Field	5-5
Table 5.10	RC36100 Data Cache Write Disable Field	5-5
Table 5.11	RC36100 Instruction Cache Write Disable Field	5-6
Table 5.12	Cause Register Exception Codes	5-7
Table 6.1	RISCore3000 Family Architecture Exceptions	6-2
Table 6.2	Cause Register Exception Codes	6-4
Table 6.3	Exception Vectors When BEV = 0	6-6
Table 6.4	Exception Vectors When BEV = 1	6-6
Table 6.5	RC36100 Exception Priority	6-7
Table 8.1	List of the Memory and I/O Controller Registers (1 of 2)	8-6
Table 8.2	List of the memory and I/O Controller Registers (2 of 2)	8-7
Table 8.3	Memory and I/O Controller Base Addresses	8-8
Table 8.4	Memory Mask Field Definitions and Values	8-9
Table 8.5	Memory and I/O Control Register Bit Assignments	8-10

Notes

Notes

Notes

List of Figures

Notes

Notes

Notes

Notes

Notes

Notes

RC36100 Device Overview

Notes

Introduction

The IDT79RC36100 is a highly integrated member of the IDT RISC MIPS processor family. The RC36100 processor incorporates the "system on a chip" philosophy and is well-suited for a wide variety of low-cost embedded applications.

The RC36100 uses a RISCore32 series CPU core, implements the MIPS instruction set architecture, and contains substantial amounts of on-chip Instruction (I-Cache) and Data Cache (D-Cache) memory. In addition, the RC36100 integrates four on-chip Memory Controllers, including ROM, DRAM, I/O, and DMA; data communication peripherals, including an IEEE 1284 Parallel Port, and two Serial Communications Ports; and standard embedded peripherals, including an Interrupt Controller, Timers, and Parallel Inputs and Outputs. Such extensive integration simplifies the overall system design, reduces external component requirements, system cost, and development time.

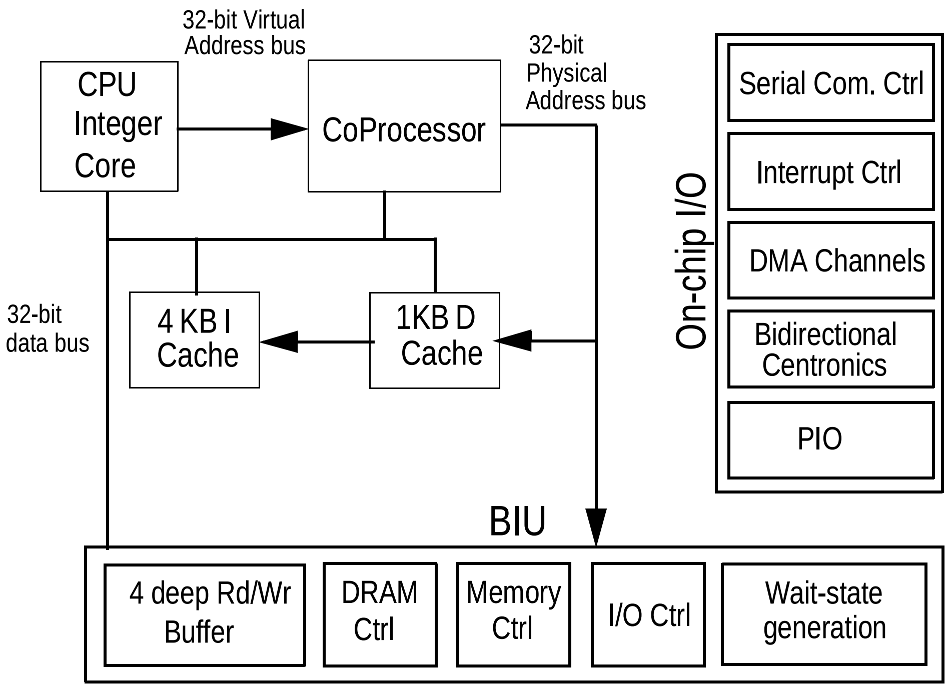


Figure 1.1 RC36100 Block Diagram

The RC36100 is software compatible with all of the IDT RISC processor family, including the low-cost 32-bit RISCore3000 family RISControllers and the RC4000/RC5000 family of high-performance 64-bit CPUs. Common instruction set architecture (ISA) enables the same applications software to be used across a wide variety of price/performance points.

The RC36100 integrates four on-chip Bus Controllers, allowing seamless interfacing with a wide variety of standard memories and peripherals that include:

- *Standard page-mode DRAMs*
- *EPROMs, FLASH, SRAM, Dual-Port SRAM*
- *FIFOs, SCSI, A/D, and other I/O peripherals*
- *Ethernet, Data Compression, and other coprocessors*

Notes

The RC36100 integrates asynchronous and synchronous Serial Ports, an IEEE Parallel Port, and multiple timers, to serve data communications applications that include:

- *Local Area Network (LAN) interface cards*
- *CSU/DSU SDLC/HDLC line driver cards*
- *Routers, switches, and data compressor cards*

RC36100 Features List

- *Instruction set compatible with the RISCore3000 family and the IDT RC4000/RC5000 64-bit family of RISC CPUs*
- *System cost minimized through a high level of integration*
  - *RISC CPU*
  - *Instruction Cache*
  - *Data Cache*
  - *Flexible Bus Interface*
  - *Controllers*
  - *Peripheral*
- *31 MIPS/ 55K Dhrystones-2.1 at 33 MHz*
- *5V operation*
- *Low-cost MQUAD packaging*
- *On-chip instruction and data caches*
  - *4KB of Instruction Cache*
  - *1KB of Data Cache*
  - *Improved Cache Control for fast data movement and cache locking*
- *Flexible bus interface allows simple, low cost designs*
  - *Separate de-multiplexed Address Bus and Data Bus*
  - *Synchronized Bus Interface Timing*
  - *On-chip 4-deep write buffer eliminates memory write stalls*
  - *On-chip 4-word read buffer supports burst or simple block reads*
  - *Programmable port width interface (8-,16-, and 32-bit memory sub-regions)*
- *On-chip DRAM Controller with Address Multiplexer*
  - *Supports non-interleaved or Interleaved DRAM memory*
- *On-chip Memory and I/O Controller*
  - *Chip Selects*
  - *Wait-State Generator*
  - *Supports non-interleaved or interleaved ROMs*
  - *Boot from 8-bit, 16-bit, 32-bit or interleaved ROMs*
  - *Supports CS/Rd/Wr I/O protocol*
  - *Supports CS/Wr/Strobe I/O protocol*
- *On-chip DMA Controller for autonomous burst data movement*
  - *4 internal channels*
  - *2 external channels*
  - *On-chip Parallel I/O pins*
  - *On-chip Interrupt Expansion controller*
- *On-chip Timers*
- *On-chip Serial Port(s)*
- *On-chip IEEE 1284 Bidirectional Centronics Target Interface Controller*
- *“Reduced Frequency Mode” assists in power-managed applications*
- *Complete software support*
  - *Optimizing compilers*
  - *Real-time operating systems*
  - *Monitors/debuggers*
  - *Floating Point emulation software*
  - *Printer Page Description Languages*
  - *Built-in Debug/Emulator Support*

Notes

Device Overview

The RC36100 can be viewed as a “system on a chip,” the embodiment of a discrete system built around the RISCore32 series CPU. Integrating system functions onto a single chip reduced the system’s cost, complexity, size, power requirements and minimized system development time.

Figure 1.1 on page 1 provides a block-level representation of the RC36100’s functional units. This section includes the RC36100’s logic symbol diagram (Figure 1.2 on page 7) and provides a complete pin description table (Table 1.1 on page 8). An overview of the RC36100’s CPU Core, System Control Co-processor (CPO), Clock Generator Unit, Instruction and Data Caches, Bus Interface Unit (BIU), Memory Controller, DRAM Controller, I/O Controller, DMA Control and Interface, Counter/Timers, PIO Interface, Serial Communications Controller, Interrupt Controller, and Bidirectional Centronics devices is presented below. More detailed information on each topic is available in subsequent chapters.

CPU Core

The RC36100 is based on the RISCore32 series CPU core. Through the use of its five-stage pipeline, the RISCore32 series is a full 32-bit RISC integer execution engine that is capable of sustaining a peak single cycle execution rate. The CPU core contains an integer ALU unit and bit shifter with a separate integer multiplier/divider unit, address adder and program counter generator, and 32 orthogonal 32-bit registers. The RC36100 execution core implements the MIPS-I Instruction Set Architecture (ISA); therefore, the RC36100 is binary compatible with all other MIPS CPU engines, including the low-cost RISCore3000 family and the high-speed RC4000 64-bit family.

System Control Co-Processor

The RC36100 integrates an on-chip System Control Co-processor (CP0). CP0 manages the RC36100’s exception handling operations, its virtual-to-physical address memory mapping, and its various programmable bus-to-cache interface capabilities. Each topic is discussed in detail throughout the manual.

The RC36100 does not include the optional TLB found in other members of the IDT processor families. Instead, the RC36100 uses a mechanism of virtual-to-physical address mapping that is identical to that of the RISCore3000 family’s Base Versions. These Base Version devices still support distinct kernel and user mode operation but do not require page management software or an on-chip TLB, leading to a simplified operating system software model and a lower cost processor.

Clock Generator Unit

The RC36100 is driven from a single, 2x-frequency input clock. An on-chip clock generator unit is responsible for managing the interaction of the CPU core, caches, and bus interface. The clock generator unit replaces the external delay line that was required in discrete RISCore32 series based systems.

For power sensitive or “Green” applications, the RC36100 supports a reduced frequency mode, allowing the system to reduce power consumption in idle periods.

Instruction Cache

The RC36100 integrates 4KB of on-chip Instruction Cache, organized with a line size of 16 bytes (four 32-bit entries). This relatively large cache contributes substantially to the high performance inherent in the RC36100, which allows systems based on the RC36100 to achieve high performance even from low-cost memory systems. The cache is implemented as a direct mapped cache and is capable of caching instructions from anywhere within the larger physical address space. The cache is implemented using physical addresses and physical tags (rather than virtual addresses or tags), which does not require flushing on context switches.

The RC36100 implements special features that allow the instruction cache to be split into halves or quarters; each section then services a different area of the large address space. This feature enables the system software to “lock” time-critical code—such as router address hash-table look-up algorithms and interrupt service routines—into one of the halves or quarters while allowing other tasks to use unused portions without disrupting the time-critical code. This technique permits software to perform instruction cache “locking” without requiring memory management support.

Notes

Data Cache

The RC36100 incorporates an on-chip data cache of 1KB organized as a 4 bytes (one word) line size. This relatively large data cache substantially contributes to the RC36100's high performance. As with the instruction cache, the data cache is implemented as a direct mapped physical address cache and is capable of mapping any word within the larger physical address space.

The data cache is implemented as a write-through cache, to ensure that main memory is always consistent and coherent with the internal cache. To minimize processor stalls due to data write operations, the bus interface unit incorporates a 4-deep write buffer that captures address and data at the processor execution rate, allowing the data to be retired to main memory at a much slower rate without impacting the performance of the CPU core.

The RC36100 contains special features that allow the data cache to be split into halves or quarters; each section services a different area of the larger address space. This feature enables the system software to "lock" time-critical data—such as routing address information tables and the interrupt stack—into one of the halves or quarters while allowing other tasks to use unused portions without disrupting the critical data. This technique permits software to perform data cache "locking" without requiring memory management support.

Bus Interface Unit

The RC36100 uses its large internal caches to provide the execution engine with most of its memory bandwidth requirements. The execution engine pipeline can then perform both 1 instruction fetch and 1 data load/store per clock cycle. Only on the rare occasion of a cache miss or on writes does the RC36100 require its external bus interface; therefore, the RC36100 is able to use a simple bus interface that connects to slower, inexpensive memory devices.

The RC36100 bus interface uses a de-multiplexed address and data bus. The bus interface readily connects to memory subsystems that are 8-, 16-, 32-bits wide, and/or interleaved 32-bit.

The RC36100 incorporates a 4-deep write buffer to decouple the speed of the execution engine from the speed of the memory system. The write buffers capture and FIFO the processor's address and data information during internal store operations at the CPU pipeline rate. The write buffer then presents the bus interface write transactions at the rate the memory system can accommodate.

During main memory writes, the RC36100 can break large data—such as a 32-bit word—into a series of smaller transactions—such as bytes—according to the width of the memory port being written. This operation is transparent to the software that initiated the store, ensuring that the same software is able to run in a variety of memory systems.

The RC36100 read interface performs both single data reads and quad word reads. To accommodate slower reads, the RC36100 incorporates a 4-deep read buffer FIFO, allowing the external interface to queue up data within the processor before releasing it to perform a burst fill of the internal caches.

In addition, the RC36100 can perform on-chip data packing when performing large data reads—such as quad words—from narrower memory systems—such as 16-bits. Once again, this operation is transparent to the software, simplifying migration of software to different memory systems and simplifying field upgrades to wider memory. Because this capability works for either instruction or data reads, the RC36100 easily supports 8-, 16-, 32-bit, or interleaved boot PROMs.

As described throughout this manual, it is actually one of the on-chip memory bus controllers that services bus transactions. The bus interface unit merely provides a common translation between these memory bus controllers and the CPU core.

Memory Controller

The RC36100 uses the on-chip memory controller to gluelessly attach external ROM—including FLASH—and/or SRAM in a number of system configurations. For example, the memory controller supports interleaved ROM and/or SRAM, 8-bit boot ROM, 32-bit burst ROMs, as well as an array of simple 32-bit wide EPROMs. Under the control of boot software, the memory controller integrates all control signals and manages the access timing and wait-state generation for multiple banks.

Notes

DRAM Controller

The RC36100 integrates an on-chip DRAM controller. The DRAM controller directly manages up to four banks of standard page mode DRAMs in a number of configurations, including systems with varying densities of DRAM; 32-bit wide, interleaved DRAM; and 16-bit wide DRAM subsystems.

I/O Controller

To perform all necessary address decoding and wait-state generation for external I/O devices, the RC36100 has an on-chip I/O controller. The I/O controller interfaces to both M- and I-style standard peripherals.

DMA Control and Interface

The RC36100 features on-chip DMA control for internal peripherals, external peripherals, and external memory. Multiple internal channels are provided, allowing block moves of data between any combination of memory and I/O device. Each channel can also be interrupt controlled so that an I/O peripheral—like the serial port—can regulate the individual transactions of a block move.

The RC36100 also supports external DMA masters that take over the external system bus via a bus request and grant handshake. Once in control, the external DMA master can read and write to memory, I/O, and internal peripherals via the RC36100's bus controllers.

Counter/Timers

The RC36100 contains three general purpose timers. Each timer consists of a 16-bit count register as well as a 16-bit compare register. The count register resets to zero and then counts upward until it equals the compare register. When the count register equals the compare register, the TCN output is asserted and the count is reset back to zero.

To increase the amount of time each timer can handle, the timers use a common 16-bit prescaler counter. Each timer is programmable to select a power-of-2 divisor of the prescaler. When the default mode is used, each timer can also be used as a general purpose real-time clock. Some special effects include:

- *Bus time-out timer*
- *Watch-dog timer*
- *PWM/square wave/baud rate generator*
- *Gated clock external event counter*

PIO Interface

For controlling multi-purpose utility pins, the RC36100 has a Parallel Input/Output (PIO) interface. The PIO pins can be programmed to act as general purpose inputs or outputs.

Each PIO pin is multiplexed with other controller's inputs or outputs. This flexible arrangement allows the system designers to customize RC36100's resources according to their needs. Therefore, designs needing a special purpose controller—such as the IEEE Parallel Port—can allocate the IEEE Parallel Port pins for that purpose; other applications can use those pins for general purpose inputs or outputs.

Serial Communications Controller

The RC36100 integrates a dual channel serial port. This peripheral controller performs a variety of synchronous and asynchronous protocols, including RS-232C, LocalTalk, SDLC, and HDLC. To maximize throughput, the on-chip Serial Port is optionally serviced by the auto-initiated on-chip DMA controller, which can automatically move data blocks to and from the port.

Interrupt Controller

The RC36100 integrates an on-chip interrupt controller to manage both external interrupts and interrupts signaled from the on-chip peripherals. The interrupt controller speeds interrupt service of the internal interrupts and assists in interrupt prioritizing and nesting as well as interfacing with the auto-initiated DMA.

Notes

IEEE 1284 Bidirectional Centronics

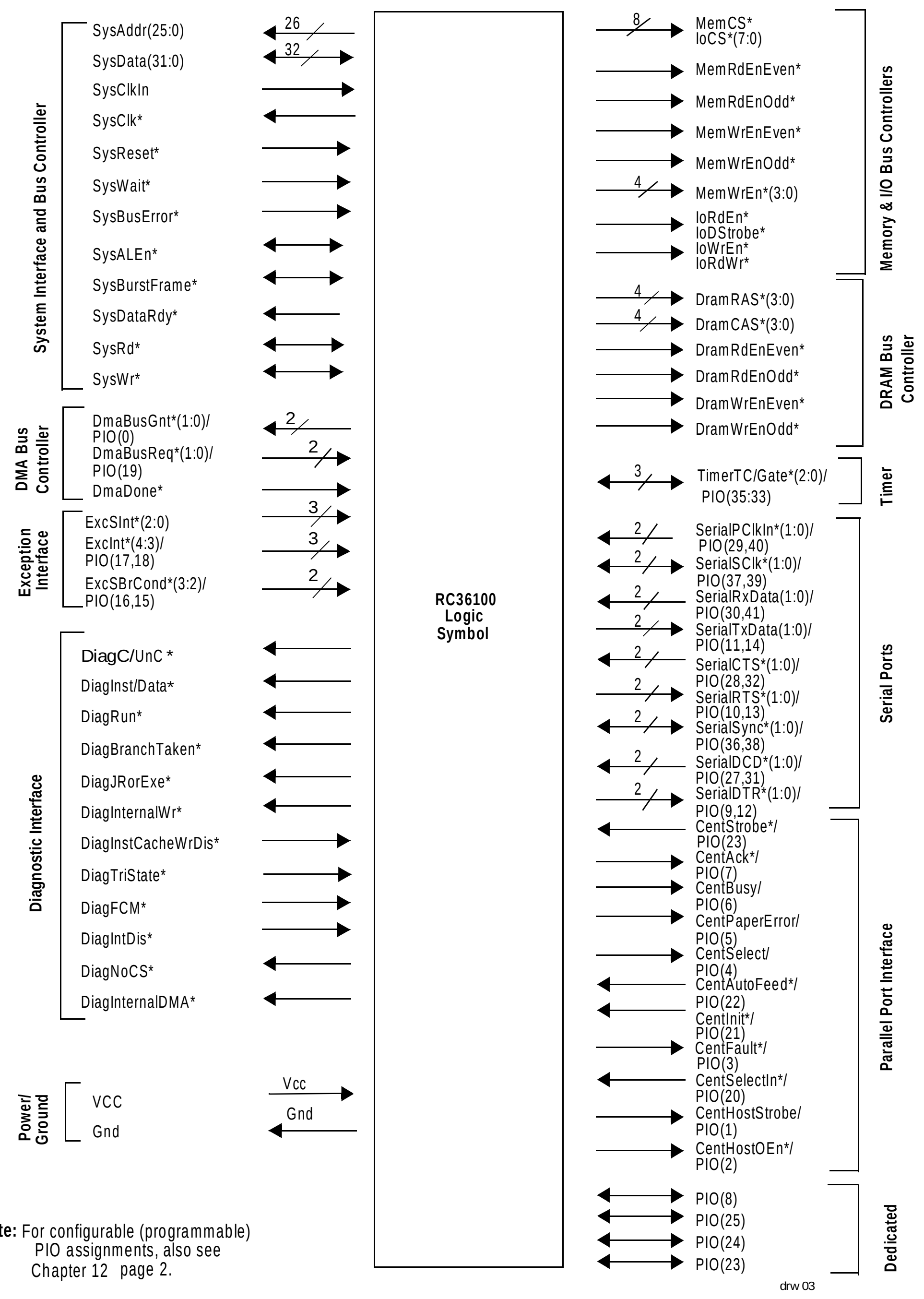
The RC36100 includes an internal IEEE1284 parallel port peripheral, which implements a true bidirectional Centronics port. Features include:

- *8-bit input Target Compatible protocol (for backward compatibility with Centronics)*
- *Nibble and byte mode output protocol (for backward compatibility with PCs)*
- *EPP protocol (for communications applications)*
- *External transceiver interface control pins*
- *Auto-initiated DMA via internal interrupts*

Notes

Pin Information

Logic Symbol



Note: For configurable (programmable) PIO assignments, also see Chapter 12 page 2.

Figure 1.2 RC36100 Logic Symbol

Notes

Pin Descriptions

Table 1.1 contains the RC36100's various bus interface, controller, timer, serial port, diagnostic, and exception handling pin names and signal descriptions. Those signals marked with an asterisk are active when low.

Pin Name	Type	Description
System Bus Interface Pins		
SysAddr(25:0)	O	**System Address Bus.** Also serves as the DramAddr(13:2) Bus.
SysData(31:0)	I/O	System Data Bus.
SysClkIn	I	**System Clock Input.** Twice (2x) the internal CPU frequency.
SysClk*	O	**System Clock Output.** All other outputs are referenced to this system clock.
SysReset*	I	**System Reset.** Initializes entire chip, except for JTAG circuitry.
SysWait*	I	**System Wait.** Extends current bus transaction.
SysBusError*	I	**System Bus Error.** Signals a bus error exception on the current read transaction.
SysALEn*	O/I(DMA)	**System Address Latch Enable.** Indicates valid address at the beginning of a bus transaction.
SysBurstFrame*	O/I(DMA)	**System Burst Frame.** First indicates the beginning of a bus transaction. Then indicates if the bus transaction is a burst and if the next data is the last data.
SysDataRdy*	O	**System Data Ready.** Indicates valid data during each data of a bus transaction (except when SysWait is asserted).
SysRd*	O/I(DMA)	**System Read.** Indicates current bus transaction is a read.
SysWr*	O/I(DMA)	**System Write.** Indicates current bus transaction is a write.
DRAM Controller Pins		
DramRAS*(3:0)	O	DRAM Row Address Strobe.
DramCAS*(3:0)	O	DRAM Column Address Strobe.
DramRDEnEven*	O	DRAM Read Enable for Even FCT245/543 Type Banks. On FCT260 type banks, it is the read enable for both.
DramRdEnOdd*	O	DRAM Read Enable for Odd FCT245/543 Type Banks. On FCT260 Type Banks, it is the path select.
DramWrEnEven*	O	DRAM Write Enable for Even Banks.
DramWrEnOdd*	O	DRAM Write Enable for Odd Banks.

Table 1.1 RC36100 Pin Descriptions (Page 1 of 4)

Notes

Pin Name	Type	Description
Memory Controller Pins		
MemCS*/IoCS*(7:0)	O	Memory or I/O Chip Selects. MemCS*(0) and optionally MemCS*(1) are reserved for the Boot PROM. IoCS*(6) and/or IoCS*(7) are optionally reserved for the Centronics Port if used.
MemRdEnEven*	O	Memory Read Enable for Even FCT245/543 Type Banks. On FCT260 Type banks, it is the read enable for both even and odd banks.
MemRdEnOdd*	O	Memory Read Enable for Even FCT245/543 Type Banks. On FCT260 Type Banks, it is the path select.
MemWrEn*(3:0)	O	Memory Write Enable for each byte lane. Memories can directly connect their byte write enables to the RC36100 MemWrEn*(3:0) signals. During 16-bit accesses, either MemWrEn*(3:2) or MemWrEn*(1:0) are used, both pairs are equivalent.
IoRdEn*/DStrobe*	O	I/O Read Enable or I/O Data Strobe.
IoWrEn*/RdWr*	O	I/O Write Enable or I/O Read/Write.
DMA Controller Pins		
DmaBusGnt*(1:0)	O	DMA Bus Grant Indicates that the CPU has tri-stated the bus and other DMA related signals.
DmaBusReq*(1:0)	I	DMA Bus Request. Indicates that external DMA agent would like control of the bus and other DMA related signals.
DmaDone*	I/O	DMA transaction done
Serial Port Pins		
SerialPClkIn*(1:0)	I	Optional Primary Serial Clock Input.
SerialSClk*(1:0)	I/O	Optional Secondary Serial Clock Input or Output.
SerialRxData(1:0)	I	Serial Receiver Data Stream.
SerialTxData(1:0)	O	Serial Transmitter Data Stream.
SerialCTS*(1:0)	I	Serial Clear To Send.
SerialRTS*(1:0)	O	Serial Request To Send.
SerialSync*(1:0)	I/O	Serial Frame Sync.
SerialDCD*(1:0)	I	Serial Data Carrier Detect.
SerialDTR*(1:0)	O	Serial Data Terminal Ready.
Timer Pins		
TimerTC*(2:0) /TimerGate*(2:0)	I/O	Timer Terminal count output or Timer Count Gate Enable input. Terminal count asserts when Timer Count equals 0. Timer Gate enables counter to count upward or to stop.

Table 1.1 RC36100 Pin Descriptions (Page 2 of 4)

Notes

Pin Name	Type	Description
PI0 Pins		
PI0(41:0)	I/O	Parallel Inputs or Parallel Outputs. Parallel inputs and parallel outputs are multiplexed with various peripheral inputs and peripheral outputs. If the peripheral is unused, the input or output pin can be reconfigured to be a general purpose input or output.
Bi-Directional Centronics Interface Pins		
CentStrobe*	I	Centronics Strobe. In compatible mode, strobes data into the printer. Has other uses for other modes.
CentAck*	O	Centronics Acknowledge. In compatible mode, acknowledges a strobe. Has other uses for other modes.
CentBusy	O	Centronics Busy. In compatible mode, delays the host from sending more data. Has other uses for other modes.
CentPaperError	O	Centronics Paper Out/Jam Error. In Compatible mode, indicates that the printer has a paper error when asserted with CentFault*. Has other uses for other modes.
CentSelect	O	Centronics Select. In Compatible mode, used to indicate that this printer is on-line. Has other uses for other modes.
CentAutoFeed*	I	Centronics Auto Page Feed. In compatible mode, sends a paper feed to the printer. Has other uses for other modes.
CentInit*	I	Centronics Initialization/Reset. In Compatible mode, resets the printer. Has other uses for other modes.
CentFault*	O	Centronics Fault. In Compatible mode, indicates that the printer has a problem. Has other uses for other modes.
CentSelectIn*	I	Centronics Select In. In Compatible mode, indicates that the Host wants to select this printer on a shared cable. Has other uses for other modes
CentHostStrobe	O	Centronics Host Strobe. Used to clock/latch Host data on the external FCT952/374 data transceiver during a Host write.
CentHostOEn*	O	Centronics Host Output Enable. Used to enable the external FCT952/374 data transceiver during a Host read.
Diagnostic Pins		
DiagC/UnC*	O	Diagnostic Cached versus Uncached. On read bus transactions indicates whether the read is cached or uncached.
DiagInst/Data*	O	Diagnostic Instruction versus Data. On read bus transactions indicates whether the read is for instructions or data.

Table 1.1 RC36100 Pin Descriptions (Page 3 of 4)

Notes

Pin Name	Type	Description
DiagRun*	O	Diagnostic Run. Indicates an internal pipeline run cycle. This pin has iso-synchronous timing.
DiagBranchTaken*	O	DiagBranchTaken Indicates that a branch, jump, or exception has been taken. This pin has asynchronous timing.
DiagJRorExe*	O	Diagnostic Jump Register or Exception occurring. Indicates that a jump register or exception is executing. This pin has asynchronous timing.
DiagInternalWr*	O	Diagnostic Internal Write. Indicates that a MTCO to CP0 register $3 is occurring.
DiagInstCacheWrDis*	O	Diagnostic Cache Write Disable. Disables writes to the instruction cache. This pin has iso-synchronous timing and is not recommended for functional use.
DiagTriState*	I	Diagnostic Tri-State all outputs. All outputs are tri-stated including SysClk. This pin is asynchronous such that tri-stating asserts or de-asserts output enables immediately.
DiagFCM*	I	Diagnostic Force Cache Miss. This pin has iso-synchronous timing. If used for functional board tests, it is recommended that it be (de-)asserted statically at reset time and left (de-)asserted.
DiagIntDis*	I	Diagnostic Interrupt Disable.
DiagNoCS*	O	Diagnostic No Chip Select. On clock 0, indicates no internal or external chip select has occurred for the current bus transaction. On clock 1, indicates an internal peripheral register read or write has occurred for the current bus transaction.
DiagInternalDMA*	O	Diagnostic Internal DMA. Asserts whenever any of the Internal DMA channels is generating the current bus transaction.
Exception Handling Pins		
ExcSInt*(2:0)	I	Exception Synchronized Interrupts. Also used as the reset initialization vector for 2:Boot16, 1:Boot8, and 0:BigEndian modes.
ExcInt*(4:3)	I	Exception Interrupts.
ExcSBrCond(3:2)	I	Exception Synchronized Branch Condition inputs.
Power/Ground Pins		
Vcc	I	Power pin. All power pins must be connected, 5V.
Gnd	I	Ground pin (VSS). All ground pins must be connected, 0V.

Table 1.1 RC36100 Pin Descriptions (Page 4 of 4)

Notes

System Usage

The IDT79RC36100 is specifically designed to easily implement low-cost memory based systems. Typical low-cost memory based systems use EPROMs and DRAM as well as application specific peripherals. Some embedded systems also optionally contain or substitute DRAM with static RAMs.

Figure 1.3 illustrates the low-system cost inherent in using the RC36100. In this example, the system uses a low cost 8-bit EPROM for boot and start-up operation. A peripheral connected to the bus uses 16 bits and for maximum memory bandwidth the DRAM is 32-bits wide. This programmable bus width allows the designer maximum flexibility in price/performance trade-off.

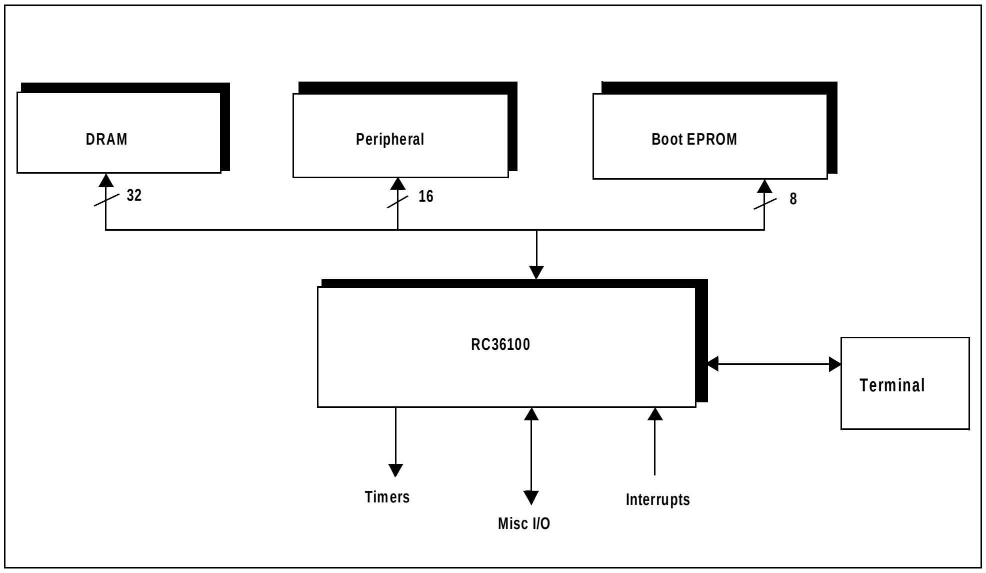


Figure 1.3 Low-cost RC36100 Based System

Development Support

The IDT RC36100 is supported by a rich set of development tools through the AdvantageIDT development tools program.

Figure 1.4 shows an overview of the system development process that is typically used with the RC36100. Tools that allow timely, parallel development of hardware and software for RC36100 family-based applications support all phases of RC36100 project development.

Some of the available support tools are:

- *Optimizing compilers from a number of leading compiler vendors.*
- *The IDT/c compiler, based on the GCC/GNU tool chain.*
- *The high-performance IDT floating-point library software.*
- *The IDT Evaluation Board, which includes RAM, EPROM, I/O, and the IDT PROM Monitor.*
- *The IDT/sim PROM Monitor, which implements a full PROM monitor (diagnostics, remote debug support, downloading utilities).*
- *IDT/kit (Kernel Integration Tool Kit), provides library support and a frame work for the system run-time environment.*

Notes

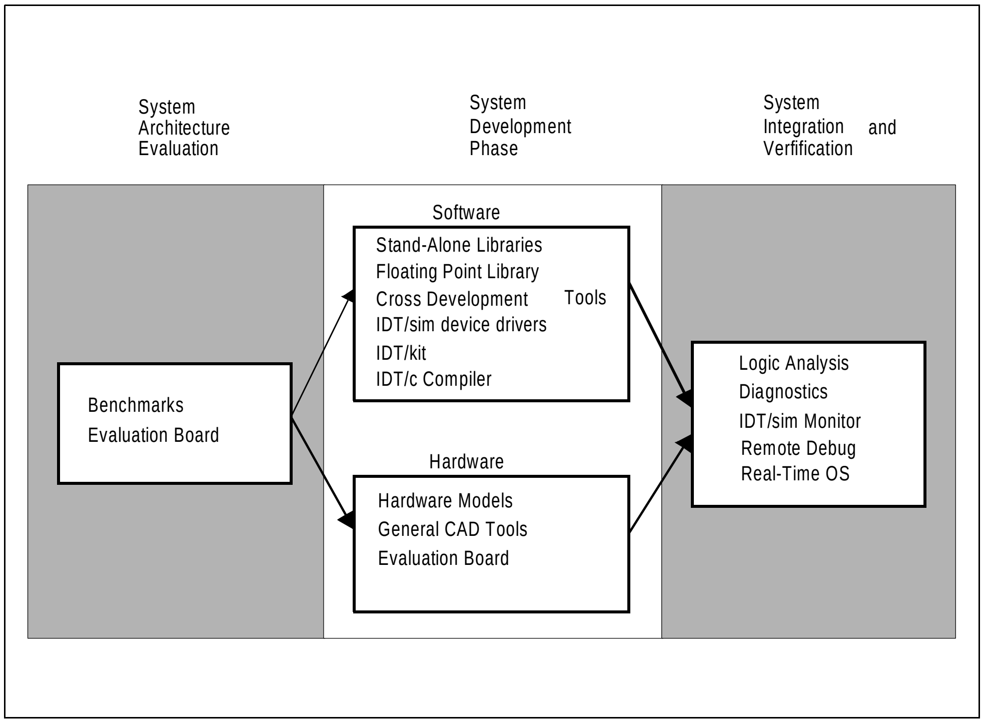


Figure 1.4 Development Support

Performance Overview

The RC36100 achieves a high level of performance based on the following features:

- ***An efficient execution engine.***
  The CPU executes almost all instructions at a single-cycle rate. Thus, the RC36100 achieves over 31 Dhrystones MIPS performance at 33MHz. By using a traditional 5-stage pipeline, the performance of the RC36100 does not degrade in applications with a high-degree of data dependency.
- ***Large on-chip caches.***
  The RC36100 contains caches which are larger than those on the majority of low-cost embedded microprocessors. These large caches minimize the required number of bus transactions and allow the RC36100 to achieve actual sustained performance that is very close to its peak execution rate, even with low-cost memory systems.
- ***Autonomous multiply and divide operations.***
  The RC36100 features an on-chip integer multiplier/divide unit that is separate from the other ALU. This allows the RC36100 to perform multiply or divide operations in parallel with other integer operations, using a single multiply or divide instruction rather than with "step" operations.
- ***Integrated write buffer.***
  The RC36100 features a four deep write buffer, which captures store target addresses and data at the processor execution rate and retires it to main memory at the slower main memory access rate. Use of on-chip write buffers eliminates the need for the processor to stall when performing store operations.
- ***Burst read support.***
  The RC36100 enables the system designer to utilize page, static or nibble mode RAMs when performing read operations, to minimize the main memory read penalty and increase the effective cache hit rates.
- ***Tightly coupled memory system.***
  Integration of on-chip memory controllers allows system resources to be accessed and managed efficiently for the needs of the execution core.

Notes

Instruction Set Architecture

Notes

Introduction

The IDT79RC36100 contains the RISCore3000, which allows achievement of dramatic performance levels through execution engine efficiency.

The RC36100 is software compatible throughout the RC3000 family as well as address map compatible with the base versions of the RISCore3000 family. However, to reduce system cost, the TLB functions that are present in the "E" versions are not available in the RC36100.

This chapter presents an overview of the MIPS-I architecture implemented in the RC36100 and discusses the programmers' model for this device. Further details on the processor software model can be found in the *IDT MIPS Microprocessor Family Software Reference Manual*.

Processor Features Overview

The RC36100 has many of the same attributes of the IDT RISCore3000 family, at a higher level of integration geared to lower system cost. These features include:

- ***Full 32-bit Operation.***
  The RC36100 contains thirty-two 32-bit general-purpose registers, and all instructions and addresses are 32- bits.
- ***Efficient Pipeline.***
  To achieve an execution rate approaching one instruction per cycle, the CPU uses a 5-stage pipeline design. Pipeline stalls, hazards, and exceptional events are handled precisely and efficiently.
- ***Large On-Chip Instruction and Data Caches.***
  The RC36100 uses large on-chip caches to provide the execution engine with a high-bandwidth. The large size of the caches insures high hit rates, minimizing stalls due to cache miss processing, and dramatically contributing to overall performance. Both the instruction and data cache can be accessed during a single CPU cycle.
- ***On-chip Memory Management.***
  The RC36100 is compatible with the base versions of the IDT RISCore3000 family, which do not use a TLB, but perform fixed segment-based mapping of the virtual space to physical addresses. In addition, through programming of the on-chip memory controllers and peripherals, the RC36100 allows kernel software to manage the system interface.

CPU Registers Overview

The RC36100 provides thirty-two general purpose 32-bit registers, an internal 32-bit Program Counter, and two dedicated 32-bit registers that hold the result of an integer multiply or divide operation. The CPU registers, illustrated in Figure 2.1, are discussed later in this chapter.

Note that the MIPS architecture does not use a traditional Program Status Word (PSW) register. The functions normally provided by such a register are instead provided through the use of “Set” instructions and conditional branches. By avoiding the use of traditional condition codes, the architecture can be more finely pipelined. This, coupled with the fine granularity of the instruction set, allows the compilers to achieve dramatically higher levels of optimizations.

Overflow and exception conditions are then handled through the use of the on-chip *Status* and *Cause* registers, which reside on-chip as part of the System Control Coprocessor (Coprocessor 0). These registers contain information about the run-time state of the machine and any exception conditions it has encountered.

Notes

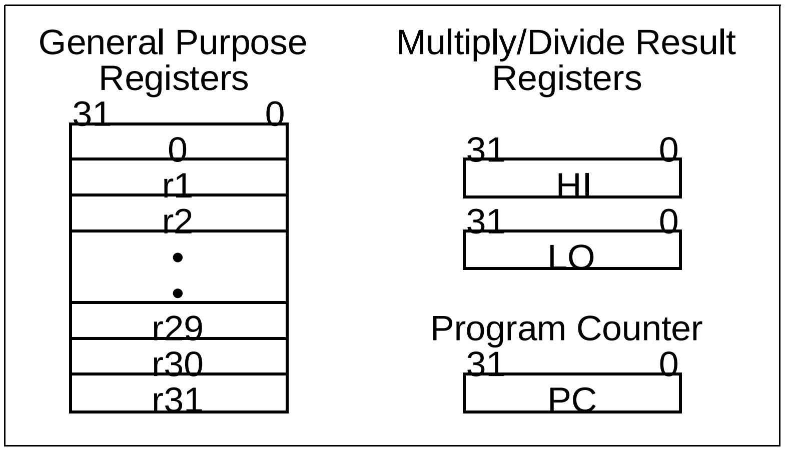


Figure 2.1 CPU Registers

Instruction Set Overview

All RC36100 instructions are 32-bits long and include three basic instruction formats. This approach dramatically simplifies instruction decoding, permitting higher frequency operation. More complicated (but less frequently used) operations and addressing modes are synthesized by the compiler/assembler, using sequences of the basic instruction set. This approach enables object code optimizations at a finer level of resolution than achievable in micro-coded CPU architectures.

Figure 2.2 shows the instruction set encoding used by the MIPS architecture. This approach simplifies instruction decoding in the CPU.

The RISCore32 series instruction set (implemented in the RC36100) can be divided into the following three basic groups:

- ***Load/Store*** *instructions move data between memory and the general registers. They are all encoded as "I-Type" instructions, and the only addressing mode implemented is base register plus signed, immediate offset. This directly enables the use of three distinct addressing modes: register plus offset; register direct; and immediate.*
- ***Computational*** *instructions perform arithmetic, logical, and shift operations on values in registers. They are encoded as either "R-Type" instructions, when both source operands as well as the result are general registers, and "I-Type," when one of the source operands is a 16-bit immediate value. Computational instructions use a three address format so that operations will not needlessly interfere with the contents of source registers.*
- ***Jump and Branch*** *instructions change the control flow of a program. A Jump instruction can be encoded as a "J-Type" instruction, in which case the Jump target address is a paged absolute address formed by combining the 26-bit immediate value with the upper four bits of the Program Counter. This form is used for subroutine calls.*

Alternately, Jumps can be encoded using the "R-Type" format, in which case the target address is a 32-bit value contained in one of the general registers. This form is typically used for returns and dispatches.

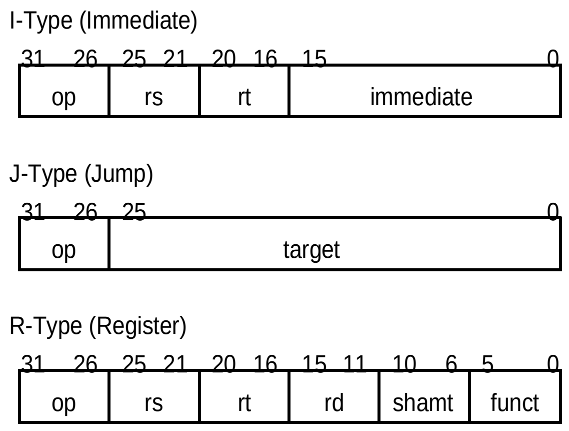
Branch operations are encoded as "I-Type" instructions. The target address is formed from a 16-bit displacement, relative to the Program Counter. The Jump and Link instructions save a return address in General Register r31. These are typically used as subroutine calls, where the subroutine return address is stored into r31 during the call operation.

- ***Coprocessor*** *instructions perform operations on the co-processor set. Coprocessor Loads and Stores are always encoded as "I-Type" instructions; in the MIPS architecture, co-processor operational instructions have co-processor dependent formats.*

In the RC36100, the System Control Coprocessor (CP0) contains registers that are used in system interface control, cache control, and exception handling.

- ***Special*** *instructions perform a variety of tasks, including movement of data between special and general registers, system calls, and breakpoint operations. Special instructions are always encoded as "R-Type" instructions.*

Notes



where:

op	is a 6-bit operation code
rs	is a 5-bit source register specifier
rt	is a 5-bit target register or branch condition
immediate	is a 16-bit immediate, or branch or address displacement
target	is a 26-bit jump target address
rd	is a 5-bit destination register specifier
shamt	is a 5-bit shift amount
funct	is a 6-bit function field

Figure 2.2 Instruction Encoding

Table 2.1 lists the instruction set mnemonics of the RC36100. These operations are presented in more detail later in this chapter and in the *IDT MIPS Microprocessor Family Software Reference Manual*.

Notes

OP	Description	OP	Description
	Load/Store Instructions		**Multiply/Divide Instructions**
LB	Load Byte	MULT	Multiply
LBU	Load Byte Unsigned	MULTU	Multiply Unsigned
LH	Load Halfword	DIV	Divide
LHU	Load Halfword Unsigned	DIVU	Divide Unsigned
LW	Load Word		
LWL	Load Word Left	MFHI	Move From HI
LWR	Load Word Right	MTHI	Move To HI
SB	Store Byte	MFLO	Move From LO
SH	Store Halfword	MTLO	Move To LO
SW	Store Word		
SWL	Store Word Left		**Jump and Branch Instructions**
SWR	Store Word Right	J	Jump
		JAL	Jump and Link
	Arithmetic Instructions (ALU Immediate)	JR	Jump to Register
		JALR	Jump and Link Register
ADDI	Add Immediate	BEQ	Branch on Equal
ADDIU	Add Immediate Unsigned	BNE	Branch on Not Equal
SLTI	Set on Less Than Immediate	BLEZ	Branch on Less than or Equal
SLTIU	Set on Less Than Immediate		to Zero
	Unsigned	BGTZ	Branch on Greater Than Zero
ANDI	AND Immediate	BLTZ	Branch on Less Than Zero
ORI	OR Immediate	BGEZ	Branch on Greater Than or
XORI	Exclusive OR Immediate		Equal to Zero
LUI	Load Upper Immediate	BLTZAL	Branch on Less Than Zero and
			Link
		BGEZAL	Branch on Greater Than or Equal to Zero and Link
	Arithmetic Instructions (3-operand, register-type)		
ADD	Add		**Special Instructions**
ADDU	Add Unsigned	SYSCALL	System Call
SUB	Subtract	BREAK	Break

Notes

OP	Description	OP	Description
SUBU	Subtract Unsigned		
SLT	Set on Less Than		**Coprocessor Instructions**
SLTU	Set on Less Than Unsigned	LWCz	Load Word from Coprocessor
AND	AND	SWCz	Store Word to Coprocessor
OR	OR	MTCz	Move To Coprocessor
XOR	Exclusive OR	MFCz	Move From Coprocessor
NOR	NOR	CTCz	Move Control To Coprocessor
		CFCz	Move Control From Coprocessor
	Shift Instructions	COPz	Coprocessor Operation
SLL	Shift Left Logical	BCzT	Branch on Coprocessor z True
SRL	Shift Right Logical	BCzF	Branch on Coprocessor z False
SRA	Shift Right Arithmetic		
SLLV	Shift Left Logical Variable		**System Control Coprocessor**
SRLV	Shift Right Logical Variable		**(CP0) Instructions**
SRAV	Shift Right Arithmetic Variable	MTC0	Move To CP0
		MFC0	Move From CP0
		TLBR†	Read indexed TLB entry
		TLBWI†	Write indexed TLB entry
		TLBWR†	Write Random TLB entry
		TLBP†	Probe TLB for matching entry
		RFE	Restore From Exception
†These instructions are not valid with the RC36100, which does not include a TLB.			

Table 2.1 Instruction Set Mnemonics

Programming Model

This section describes the organization of data in the general registers and in memory and discusses the set of available general registers. A summary of all of the CPU registers is presented.

Data Formats and Addressing

The MIPS-I architecture defines a word as 32-bits, a half-word as 16-bits, and a byte as 8-bits. The byte ordering convention is configurable during hardware reset into either a *big-endian* or *little-endian* convention.

When configured as a big-endian system, byte 0 is always the most significant (leftmost) byte in a word. But when configured as a little-endian system, byte 0 is always the least significant (rightmost) byte in a word.

Notes

Figure 2.3 shows the ordering of bytes within words and the ordering of words within multiple word structures for the big-endian and little-endian conventions.

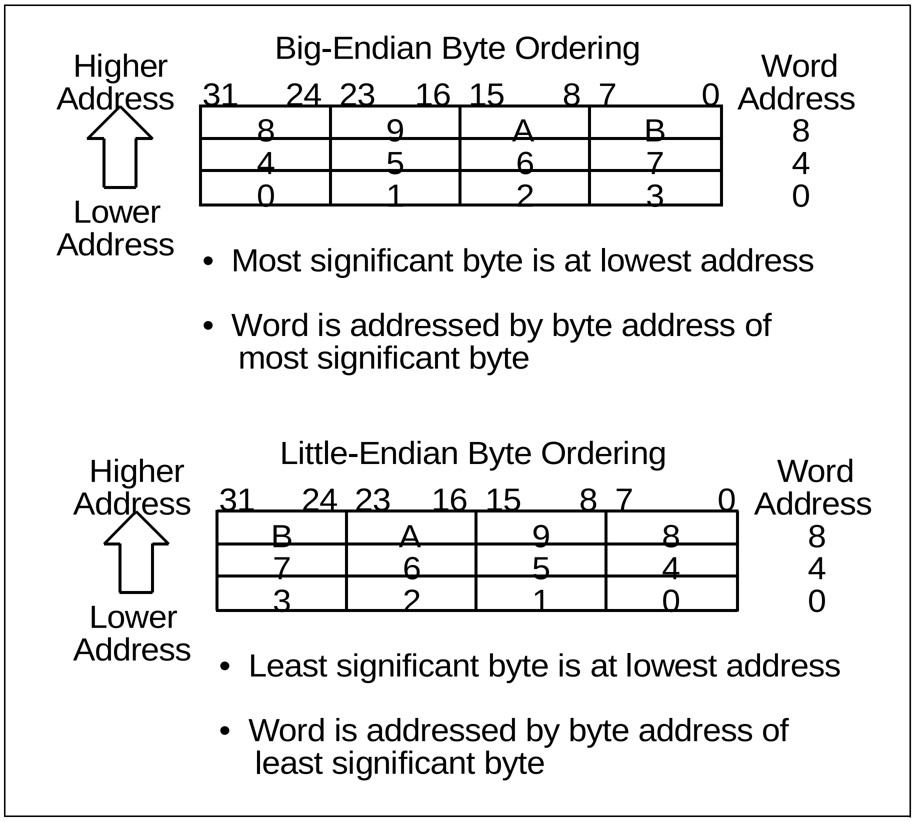

Big-Endian Byte Ordering
Higher Address
Lower Address
Word Address
31 24 23 16 15 8 7 0
8 9 A B 8
4 5 6 7 4
0 1 2 3 0
• Most significant byte is at lowest address
• Word is addressed by byte address of most significant byte
Little-Endian Byte Ordering
Higher Address
Lower Address
Word Address
31 24 23 16 15 8 7 0
B A 9 8 8
7 6 5 4 4
3 2 1 0 0
• Least significant byte is at lowest address
• Word is addressed by byte address of least significant byte

Figure 2.3 Byte Ordering Conventions

The RC36100 uses byte addressing for all accesses, including half-word and word. The MIPS architecture has alignment constraints that require half-word accesses to be aligned on an even byte boundary and word accesses to be aligned on a modulo-4 byte boundary. Thus, in big-endian systems, the address of a multiple-byte data item is the address of the most-significant byte, while in little-endian systems it is the address of the least-significant byte of the structure.

The MIPS instruction set provides special instructions for addressing 32-bit words that are not aligned on 4-byte boundaries. These instructions—Load/Store Left/Right—are used in pairs to provide addressing of misaligned words. This means that these types of data movements require only one-additional instruction cycle over that required for properly aligned words (note that unaligned data is read by the CPU in the same number of cycles as would be required for a full hardware solution and provides a much more efficient way of dealing with this case than is possible using sequences of loads/stores and shift operations or by using traps).

Various tool chains, such as the IDT/c compiler, can automatically generate these instructions for "packed" data. Figure 2.4 shows the bytes accessed when addressing a mis-aligned word with a byte address of 3, for each of the two byte ordering conventions.

Notes

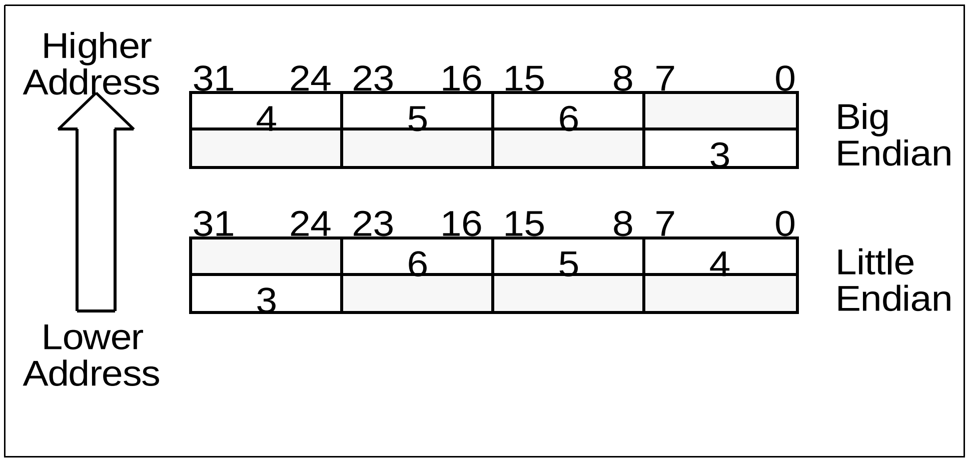


Figure 2.4 Unaligned Words

CPU General Registers

The RC36100 contains 32 general registers, each containing a single 32-bit word. The 32 general registers are treated symmetrically (orthogonally), with two notable exceptions: general register r0 is hardwired to a zero value, and r31 is used as the link register in Jump and Link instructions

When used as a source register, register r0 maintains the value zero under all conditions and discards data written to it. Thus, instructions that attempt to write to r0 may be used as No-Op Instructions. The use of a register wired to the zero value allows the simple synthesis of different addressing modes, no-ops, register or memory clear operations, etc., without requiring expansion of the basic instruction set.

Register r31 is used as the link register in jump and link instructions. These instructions are used in subroutine calls, and the subroutine return address is placed in register r31. This register can be written to or read as a normal register in other operations.

In addition to the general registers, the CPU contains two registers (HI and LO) which store the double-word, 64-bit result of integer multiply operations, and the quotient and remainder of integer divide operations.

CP0 Special Registers

In addition to the general CPU registers, the RC36100 contains a number of special on-chip registers. These registers logically reside in the on-chip System Control Co-processor, CP0, and are used in memory management and exception handling.

Table 2.2 on page 8 shows the logical CP0 address of each of the registers. The format of these registers, and their use, is discussed in later chapters. Note that the MIPS architecture allows CP0 to vary by implementation. The RC36100 contains some new CP0 registers; however, their definition is such that it still remains possible to use a single binary program across all family members, in that these registers are typically managed only at reset.

Notes

Number	Mnemonic	Description
0	Reserved(1)	
1	Reserved(1)	
2	Reserved(1)	
3	Config(3)	Cache Usage Configuration
4	Reserved(1)	
5:7	Reserved	
8	BadVAddr	Bad Virtual Address
9	Reserved(3)	
10	Reserved(3)	
11	Reserved(3)	
12	SR	Status Register
13	Cause	Cause of Last Exception
14	EPC	Exception Program Counter
15	PRId	Processor Revision Identifier
16:31	Reserved	
NOTES: This register is used in Extended Architecture CPUs to control the TLB and virtual memory system. In the “E” versions, register $2 is "TLB EntryLo", and register $10 is "TLB EntryHi". This register is reserved in other family members and has a different meaning in them.		

Table 2.2 RC36100 CP0 Registers

Operating Modes

The RC36100 supports two operating modes: User and Kernel. The RC36100 operates in User mode until an exception is detected, forcing it into kernel mode. It remains in Kernel mode until a Return From Exception (RFE) instruction is executed, returning it to its previous operation mode.

The processor supports these levels of protection by segmenting the 4GB virtual address space into 4 distinct segments. One segment is accessible from either the User state or the Kernel mode, and the other three segments are only accessible from kernel mode.

In addition to providing memory address protection, the kernel can protect the co-processors (in the case of the RC36100, CP0) from access or modification by the user task. Chapter 4 discusses the memory management facilities of the processor.

Pipeline Architecture

The RC36100 uses the same basic pipeline structure as that implemented in the RISCore32 series. Thus, the execution of a single instruction is performed in the following five distinct stages:

- *Instruction Fetch (IF). In this stage, the instruction virtual address is translated to a physical address and the instruction is read from the internal Instruction Cache.*
- *Read (RD). During this stage, the instruction is decoded and required operands are read from the on-chip register file.*
- *ALU. The required operation is performed on the instruction operands.*
- *Memory Access (MEM). If the instruction was a load or store, the Data Cache is accessed. Note that there is a skew between the instruction cycle which fetches the instruction and the one in which the required data transfer occurs. This skew is a result of the intervening pipeline stages.*
- *Write Back (WB). During the write back pipestage, the results of the ALU stage operation are updated into the on-chip register file.*

Notes

Each pipestage requires approximately one CPU cycle, as shown in Figure 2.5. Parts of some operations lap into the next cycle, while other operations require only 1/2 cycle.

The net effect of the pipeline structure is that a new instruction can be initiated every clock cycle. Thus, the execution of five instructions at a time is overlapped, as shown in Figure 2.6.

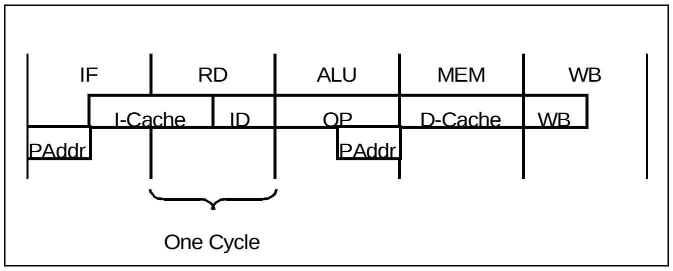


Figure 2.5 5-Stage Pipeline

The pipeline operates efficiently, because different CPU resources such as address and data bus access, ALU operations, and the register file, are used on a non-interfering basis.

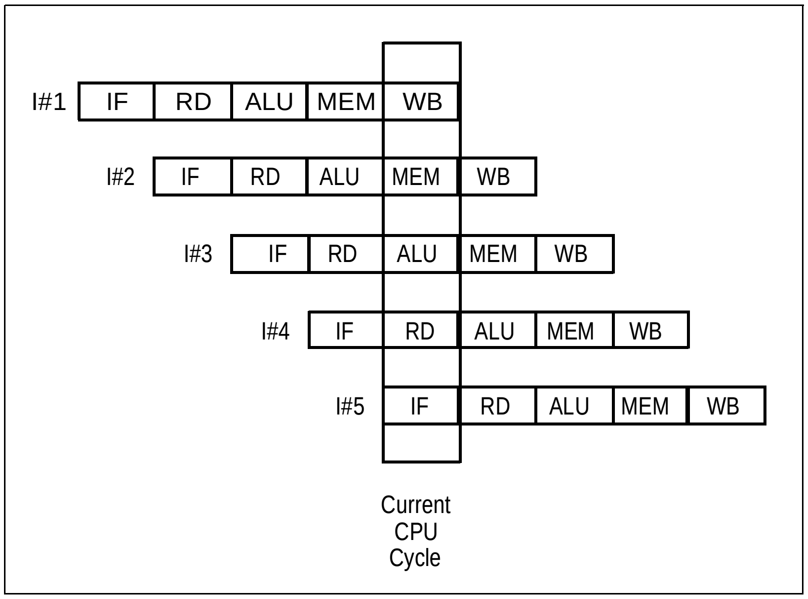


Figure 2.6 5-Instructions per Clock Cycle

Pipeline Hazards

In a pipelined machine such as the RC36100, there are certain instructions which, based on the pipeline structure, can potentially disrupt the smooth operation of the pipeline. The basic problem is that the current pipestage of an instruction may require the result of a previous instruction, still in the pipeline, whose result is not yet available. This class of problems is referred to as pipeline hazards.

An example of a potential pipeline hazard occurs when a computational instruction n+1) requires the result of the immediately prior instruction (instruction n). Instruction n+1 wants to access the register file during the RF pipestage. However, instruction n has not yet completed its register write-back operation, and thus the current value is not available directly from the register file. In this case, special logic within the execution engine forwards the result of instruction n's ALU operation to instruction n+1, prior to the true writeback operation. The pipeline is undisturbed, and no pipeline *stalls* need to occur.

Notes

Another example of a pipeline hazard handled in hardware is the integer multiply and divide operations. If an instruction attempts to access the HI or LO registers prior to the completion of the multiply or divide, that instruction will be *interlocked* (held off) until the multiply or divide operation completes. Thus, the programmer is isolated from the actual execution time of this operation. The optimizing compilers attempt to schedule as many instructions as possible between the start of the multiply/divide and the access of its result, to minimize stalls.

However, not all pipeline hazards are handled in hardware. There are two notable categories of instructions which require software intervention to insure logical operation. The optimizing compilers (and peephole scheduler of the assembler) are capable of insuring proper execution. These two instruction classes are:

- *Load instructions have a delay, or latency, of one cycle before the data loaded from memory is available another instruction. This is because the ALU stage of the immediately subsequent instruction is processed simultaneously with the Data Cache access of the load operation. Figure 2.7 illustrates the cause of this delay slot.*

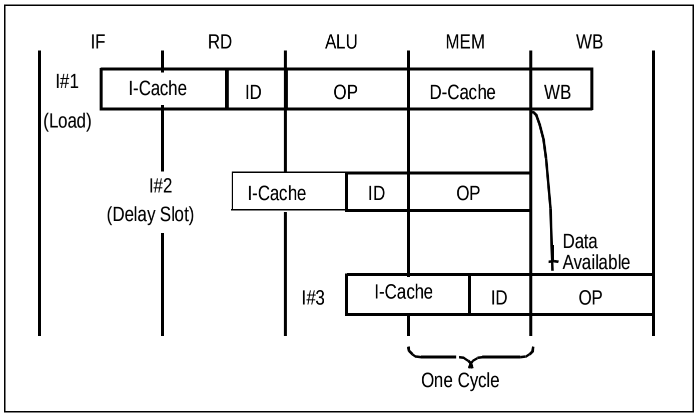


Figure 2.7 Load Delay

- *Jump and Branch instructions have a delay of one cycle before the program flow change can occur. This is due to the fact that the next instruction is fetched prior to the decode and ALU stage of the jump/branch operation. Figure 2.8 illustrates the cause of this delay slot.*

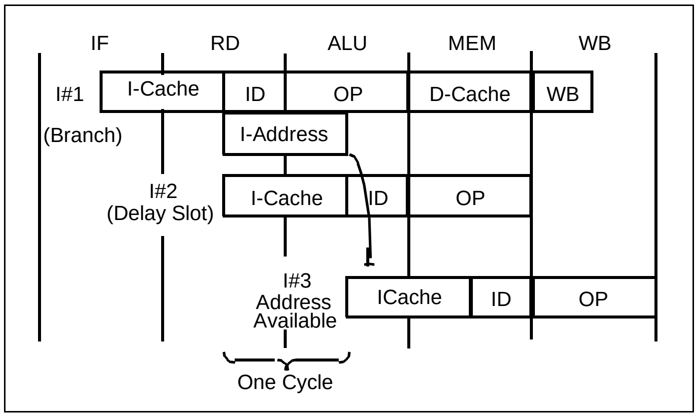


Figure 2.8 Branch Delay

The RC36100 continues execution, despite the delay in the operation. Thus, loads, jumps and branches do not disrupt the pipeline flow of instructions, and the processor always executes the instruction immediately following one of these “delayed” instructions.

Note: Note that there may also be latencies associated with changes to various of the CP0 registers; for example, changing the bus interface control register may require multiple cycles before the change is actually reflected in the chip interface.

Notes

Rather than include extensive pipeline control logic, the MIPS-I instruction set gives responsibility for dealing with “delay slots” to software. Thus, peephole optimizations (which can be performed as part of compilation or assembly) can re-order the code to insure that the instruction in the delay slot does not require the logical result of the “delayed” instruction. In the worst case, a NOP can be inserted to guarantee proper software execution.

Chapter 6 discusses the impact of pipelining on exception handling. In general, when an instruction causes an exception, it is desirable for all instructions initiated prior to that instruction to complete, and all subsequent instructions to abort. This insures that the machine state presented to the exception handler reflects the logical state that existed at the time the exception was detected. In addition, it is desirable to avoid requiring software to explicitly manage the pipeline when handling or returning from exceptions. The IDT RC36100 pipeline is designed to properly manage exceptional events.

Instruction Set Summary

This section provides an overview of the RC36100 instruction set by presenting each category of instructions in a tabular summary form. Refer to the "IDT RISCore3000 Family Software Reference Manual", for a detailed description of each instruction.

Instruction Formats

Every instruction consists of a single word (32 bits) aligned on a word boundary. There are only three instruction formats, as shown in Figure 2.2 on page 3. This approach simplifies instruction decoding. More complicated or less frequently used operations and addressing modes are synthesized by the compilers.

Instruction Notational Conventions

In this manual, all variable sub-fields in an instruction format (such as rs, rt, immediate, and so on) are shown in lower-case names.

For the sake of clarity, an alias is sometimes used for a variable sub-field in the formats of specific instructions. For example, “base” rather than “rs” is used in the format for Load and Store instructions. Such an alias is always lower case, since it refers to a variable sub-field.

Instruction opcodes are shown in upper case.

The actual bit encoding for all the mnemonics is specified at the end of this chapter.

Load and Store Instructions

Load/Store instructions move data between memory and general registers. They are all I-type instructions. The only addressing mode directly supported is base register plus 16-bit signed immediate offset. This can be used to directly implement immediate addressing (using the r0 register) or register direct (using an immediate offset value of zero).

All load operations have a latency effect of one instruction. That is, the data being loaded from memory into a register is not available to the instruction that immediately follows the load instruction: the data is available to the second instruction after the load instruction. An exception to this rule is that for the target register for the “load word left” and “load word right” instructions may be specified as the same register used as the destination of the related unaligned load instruction that immediately precedes it.

The Load/Store instruction opcode determines the size of the data item to be loaded or stored, as shown in Table 2.1 on page 5. Regardless of access type or byte numbering-order (endianness), the address specifies the byte that has the smallest byte address of all bytes in the addressed field. For a big-endian access, this is the most significant byte; for a little-endian access, this is the least significant byte.

Note: In an RC36100 based system, the endianness of a given access is dynamic, in that the RE (Reverse Endianness) bit of the Status Register can be used to force user space accesses of the opposite byte convention of the kernel.

Notes

Big-Endian (32-bit memory system)

Size	CPU Core VAdrLo (1)	CPU Core VAdrLo (0)	BE(3) Data (31:24)	BE(2) Data (23:16)	BE(1) Data (15:8)	BE(0) Data(7:0)
Word	0	0	Yes	Yes	Yes	Yes
Tri-Byte	0	0	Yes	Yes	Yes	No
Tri-Byte	0	1	No	Yes	Yes	Yes
16-Bit	0	0	Yes	Yes	No	No
16-Bit	1	0	No	No	Yes	Yes
Byte	0	0	Yes	No	No	No
Byte	0	1	No	Yes	No	No
Byte	1	0	No	No	Yes	No
Byte	1	1	No	No	No	Yes

Table 2.3 Big-Endian (32-bit memory system)

Little-Endian (32-bit memory system)

Size	CPU Core VAdrLo (1)	CPU Core VAdrLo (0)	BE(3) Data (31:24)	BE(2) Data (23:16)	BE(1) Data (15:8)	BE(0) Data (7:0)
Word	0	0	Yes	Yes	Yes	Yes
Tri-Byte	0	0	No	Yes	Yes	Yes
Tri-Byte	0	1	Yes	Yes	Yes	No
16-Bit	0	0	No	No	Yes	Yes
16-Bit	1	0	Yes	Yes	No	No
Byte	0	0	No	No	No	Yes
Byte	0	1	No	No	Yes	No
Byte	1	0	No	Yes	No	No
Byte	1	1	Yes	No	No	No

Table 2.4 Byte Addressing in Load/Store Operations (32-bit memory)

Notes

Big-Endian (16-bit memory system)

Size	CPU Core VAdrLo(1)	CPU Core VAdrLo(0)	First Transfer BE16(1) Data(31:24)	First Transfer BE16(0) Data(23:16)	Second Transfer BE16(1) Data(31:24)	Second Transfer BE16(0) Data(23:16)
Word	0	0	Yes	Yes	Yes	Yes
Tri-Byte	0	0	Yes	Yes	Yes	No
Tri-Byte	0	1	No	Yes	Yes	Yes
16-Bit	0	0	Yes	Yes	N/A	N/A
16-Bit	1	0	Yes	Yes	N/A	N/A
Byte	0	0	Yes	No	N/A	N/A
Byte	0	1	No	Yes	N/A	N/A
Byte	1	0	Yes	No	N/A	N/A
Byte	1	1	No	Yes	N/A	N/A

Table 2.5 Big-Endian (16-bit memory system)

Little-Endian (16-bit memory system)

Size	CPU Core VAdrLo (1)	CPU Core VAdrLo (0)	First Transfer BE16(1) Data (15:8)	First Transfer BE16(0) Data (7:0)	Second Transfer BE16(1) Data (15:8)	Second Transfer BE16(0) Data (7:0)
Word	0	0	Yes	Yes	Yes	Yes
Tri-Byte	0	0	Yes	Yes	No	Yes
Tri-Byte	0	1	Yes	No	Yes	Yes
16-Bit	0	0	Yes	Yes	N/A	N/A
16-Bit	1	0	Yes	Yes	N/A	N/A
Byte	0	0	No	Yes	N/A	N/A
Byte	0	1	Yes	No	N/A	N/A
Byte	1	0	No	Yes	N/A	N/A
Byte	1	1	Yes	No	N/A	N/A

Table 2.6 Byte Addressing in Load/Store Operations
(16-bit memory)

Note that the size of the operand requested by the load instruction is independent of the memory width of the addressed memory. Thus, if the actual size of the data is 32-bits, software can safely use a load or store word instruction, even if the addressed memory is actually only 8- or 16-bits wide. The bus interface unit will interact with CP0 to determine the width of the addressed memory, and will, if necessary, perform multiple data transfers to satisfy a single load or store instruction.

The bytes within the addressed word that are used can be determined directly from the access size and the two low-order bits of the address, as shown in Table 2.3, Table 2.4, Table 2.5, and Table 2.6. Note that certain combinations of access types and low-order address bits can never occur: only the combinations shown in these tables are permissible.

Table 2.7 shows the load/store instructions supported by the MIPS-I ISA.

Notes

Instruction	Format and Description
Load Byte	LB rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Sign-extend contents of addressed byte and load into rt.
Load Byte Unsigned	LBU rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Zero-extend contents of addressed byte and load into rt.
Load Halfword	LH rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Sign-extend contents of addressed half-word and load into rt.
Load Halfword Unsigned	LHU rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Zero-extend contents of addressed half-word and load into rt.
Load Word	LW rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Load contents of addressed word into register rt.
Load Word Left	LWL rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Shift addressed word left so that addressed byte is leftmost byte of a word. Merge bytes from memory with contents of register rt and load result into register rt.
Load Word Right	LWR rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Shift addressed word right so that addressed byte is rightmost byte of a word. Merge bytes from memory with contents of register rt and load result into register rt.
Store Byte	SB rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Store least significant byte of register rt at addressed location.
Store Halfword	SH rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Store least significant halfword of register rt at addressed location.
Store Word	SW rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Store least significant word of register rt at addressed location.
Store Word Left	SWL rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Shift contents of register rt right so that leftmost byte of the word is in position of addressed byte. Store bytes containing original data into corresponding bytes at addressed byte.
Store Word Right	SWR rt, offset (base) Sign-extend 16-bit offset and add to contents of register base to form address. Shift contents of register rt left so that rightmost byte of the word is in position of addressed byte. Store bytes containing original data into corresponding bytes at addressed byte.

Table 2.7 Load and Store Instructions

Computational Instructions

Computational instructions perform arithmetic, logical, and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats. There are four categories of computational instructions:

Notes

- ***ALU Immediate*** *instructions are summarized in Table 2.8.*
- ***3-Operand Register Type*** *instructions are summarized in Table 2.9 on page 16.*
- ***Shift*** *instructions are summarized in Table 2.10 on page 17.*
- ***Multiply/Divide*** *instructions are summarized in Table 2.11 on page 17.*

Instruction	Format and Description
ADD Immediate	ADDI rt, rs, immediate Add 16-bit sign-extended immediate to register rs and place 32-bit result in register rt. Trap on two's complement overflow.
ADD Immediate Unsigned	ADDIU rt, rs, immediate Add 16-bit sign-extended immediate to register rs and place 32-bit result in register rt. Do not trap on overflow.
Set on Less Than Immediate	SLTI rt, rs, immediate Compare 16-bit sign-extended immediate with register rs as signed 32-bit integers. Result = 1 if rs is less than immediate; otherwise result = 0. Place result in register rt.
Set on Less Than Unsigned Immediate	SLTIU rt, rs, immediate Compare 16-bit sign-extended immediate with register rs as unsigned 32-bit integers. Result = 1 if rs is less than immediate; otherwise result = 0. Place result in register rt. Do not trap on overflow.
AND Immediate	ANDI rt, rs, immediate Zero-extend 16-bit immediate, AND with contents of register rs and place result in register rt.
OR Immediate	ORI rt, rs, immediate Zero-extend 16-bit immediate, OR with contents of register rs and place result in register rt.
Exclusive OR Immediate	XORI rt, rs, immediate Zero-extend 16-bit immediate, exclusive OR with contents of register rs and place result in register rt.
Load Upper Immediate	LUI rt, immediate Shift 16-bit immediate left 16 bits. Set least significant 16 bits of word to zeroes. Store result in register rt.

Table 2.8 ALU Immediate Operations

Notes

Instruction	Format and Description
Add	ADD rd, rs, rt Add contents of registers rs and rt and place 32-bit result in register rd. Trap on two’s complement overflow.
ADD Unsigned	ADDU rd, rs, rt Add contents of registers rs and rt and place 32-bit result in register rd. Do not trap on overflow.
Subtract	SUB rd, rs, rt Subtract contents of registers rt and rs and place 32-bit result in register rd. Trap on two’s complement overflow.
Subtract Unsigned	SUBU rd, rs, rt Subtract contents of registers rt and rs and place 32-bit result in register rd. Do not trap on overflow.
Set on Less Than	SLT rd, rs, rt Compare contents of register rt to register rs (as signed 32-bit integers). If register rs is less than rt, result = 1; otherwise, result = 0.
Set on Less Than Unsigned	SLTU rd, rs, rt Compare contents of register rt to register rs (as unsigned 32-bit integers). If register rs is less than rt, result = 1; otherwise, result = 0.
AND	AND rd, rs, rt Bit-wise AND contents of registers rs and rt and place result in register rd.
OR	OR rd, rs, rt Bit-wise OR contents of registers rs and rt and place result in register rd.
Exclusive OR	XOR rd, rs, rt Bit-wise Exclusive OR contents of registers rs and rt and place result in register rd.
NOR	NOR rd, rs, rt Bit-wise NOR contents of registers rs and rt and place result in register rd.

Table 2.9 Three Operand Register-Type Operations

Notes

Instruction	Format and Description
Shift Left Logical	SLL rd, rt, shamt Shift contents of register rt left by shamt bits, inserting zeroes into low order bits. Place 32-bit result in register rd.
Shift Right Logical	SRL rd, rt, shamt Shift contents of register rt right by shamt bits, inserting zeroes into high order bits. Place 32-bit result in register rd.
Shift Right Arithmetic	SRA rd, rt, shamt Shift contents of register rt right by shamt bits, sign-extending the high order bits. Place 32-bit result in register rd.
Shift Left Logical Variable	SLLV rd, rt, rs Shift contents of register rt left. Low-order 5 bits of register rs specify number of bits to shift. Insert zeroes into low order bits of rt and place 32-bit result in register rd.
Shift Right Logical Variable	SRLV rd, rt, rs Shift contents of register rt right. Low-order 5 bits of register rs specify number of bits to shift. Insert zeroes into high order bits of rt and place 32-bit result in register rd.
Shift Right Arithmetic Variable	SRAV rd, rt, rs Shift contents of register rt right. Low-order 5 bits of register rs specify number of bits to shift. Sign-extend the high order bits of rt and place 32-bit result in register rd.

Table 2.10 Shift Operations

Instruction	Format and Description
Multiply	MULT rs, rt Multiply contents of registers rs and rt as twos complement values. Place 64-bit result in special registers HI/LO
Multiply Unsigned	MULTU rs, rt Multiply contents of registers rs and rt as unsigned values. Place 64-bit result in special registers HI/LO
Divide	DIV rs, rt Divide contents of register rs by rt treating operands as twos complements values. Place 32-bit quotient in special register LO, and 32-bit remainder in HI.
Divide Unsigned	DIVU rs, rt Divide contents of register rs by rt treating operands as unsigned values. Place 32-bit quotient in special register LO, and 32-bit remainder in HI.
Move From HI	MFHI rd Move contents of special register HI to register rd.
Move From LO	MFLO rd Move contents of special register LO to register rd.
Move To HI	MTHI rd Move contents of special register rd to special register HI.
Move To LO	MTLO rd Move contents of register rd to special register LO.

Table 2.11 Multiply and Divide Operations

Notes

Jump and Branch instructions

Jump and Branch instructions change the control flow of a program. All Jump and Branch instructions occur with a one instruction delay: that is, the instruction immediately following the jump or branch is always executed while the target instruction is being fetched, regardless of whether the branch is to be taken.

An assembler has several possibilities for utilizing the branch delay slot productively:

- *It can insert an instruction that logically precedes the branch instruction in the delay slot since the instruction immediately following the jump/branch effectively belongs to the block preceding the transfer instruction.*
- *It can replicate the instruction that is the target of the branch/jump into the delay slot provided that no side-effects occur if the branch falls through.*
- *It can move an instruction up from below the branch into the delay slot, provided that no side-effects occur if the branch is taken.*
- *If no other instruction is available, it can insert a NOP instruction in the delay slot.*

The J-type instruction format is used for both jumps-and-links for subroutine calls. In this format, the 26-bit target address is shifted left two bits, and combined with high-order 4 bits of the current program counter to form a 32-bit absolute address.

The R-type instruction format, which takes a 32-bit byte address, contained in a register is used for returns, dispatches, and cross-page jumps.

Branches have 16-bit offsets relative to the program counter (I-type). Jump-and-Link and Branch-and-Link instructions save a return address in register r31.

Table 2.12 summarizes the RC36100's Jump instructions and Table 2.13 on page 19 summarizes the Branch instructions.

Instruction	Format and Description
Jump	J target Shift 26-bit target address left two bits, combine with high-order 4 bits of PC and jump to address with a one instruction delay.
Jump and Link	JAL target Shift 26-bit target address left two bits, combine with high-order 4 bits of PC and jump to address with a one instruction delay. Place address of instruction following delay slot in r31 (link register).
Jump Register	JR rs Jump to address contained in register rs with a one instruction delay.
Jump and Link Register	JALR rs, rd Jump to address contained in register rs with a one instruction delay. Place address of instruction following delay slot in rd.

Table 2.12 Jump Instructions

Notes

Instruction	Format and Description
Branch Target	All Branch instruction target addresses are computed as follows: Add address of instruction in delay slot and the 16-bit offset (shifted left two bits and sign-extended to 32 bits). All branches occur with a one instruction delay.
Branch on Equal	BEQ rs, rt, offset Branch to target address if register rs equal to rt
Branch on Not Equal	BNE rs, rt, offset Branch to target address if register rs not equal to rt.
Branch on Less than or Equal Zero	BLEZ rs,offset Branch to target address if register rs less than or equal to 0.
Branch on Greater Than Zero	BGTZ rs,offset Branch to target address if register rs greater than 0.
Branch on Less Than Zero	BLTZ rs,offset Branch to target address if register rs less than 0.
Branch on Greater than or Equal Zero	BGEZ rs,offset Branch to target address if register rs greater than or equal to 0.
Branch on Less Than Zero And Link	BLTZAL rs, offset Place address of instruction following delay slot in register r31 (link register). Branch to target address if register rs less than 0.
Branch on greater than or Equal Zero And Link	BGEZAL rs, offset Place address of instruction following delay slot in register r31 (link register). Branch to target address if register rs is greater than or equal to 0.

Table 2.13 Branch Instructions

Special Instructions

The two Special instructions let software initiate traps. They are always R-type. Table 2.14 summarizes Special Instructions.

Instruction	Format and Description
System Call	SYSCALL Initiates system call trap, immediately transferring control to exception handler.
Breakpoint	BREAK Initiates breakpoint trap, immediately transferring control to exception handler.

Table 2.14 Special Instructions

Co-processor Instructions

Co-processor instructions perform operations in the co-processors. Co-processor Loads and Stores are I-type. Co-processor computational instructions have co-processor-dependent formats. The only co-processor operations of relevance for the RC36100 are those targeted at the on-chip CP0. Table 2.15 summarizes the Co-processor Instruction Set of the MIPS ISA.

Notes

Instruction	Format and Description
Load Word to Co-processor	LWCz rt, offset (base) Sign-extend 16-bit offset and add to base to form address. Load contents of addressed word into co-processor register rt of co-processor unit z.
Store Word from Co-processor	SWCz rt, offset (base) Sign-extend 16-bit offset and add to base to form address. Store contents of co-processor register rt from co-processor unit z at addressed memory word.
Move To Co-processor	MTCz rt, rd Move contents of CPU register rt into co-processor register rd of co-processor unit z.
Move from Co-processor	MFCz rt,rd Move contents of co-processor register rd from co-processor unit z to CPU register rt.
Move Control To Co-processor	CTCz rt,rd Move contents of CPU register rt into co-processor control register rd of co-processor unit z.
Move Control From Co-processor	CFCz rt,rd Move contents of control register rd of co-processor unit z into CPU register rt.
Co-processor Operation	COPz cofun Co-processor z performs an operation. The state of the RC36100 is not modified by a co-processor operation.
Branch on Co-processor z True	BCzT offset Compute a branch target address by adding address of instruction in the 16-bit offset (shifted left two bits and sign-extended to 32-bits). Branch to the target address (with a delay of one instruction) if co-processor z's condition line is true.
Branch on Co-processor z False	BCzF offset Compute a branch target address by adding address of instruction in the 16-bit offset (shifted left two bits and sign-extended to 32-bits). Branch to the target address (with a delay of one instruction) if co-processor z's condition line is false.

Table 2.15 Co-Processor Operations

System Control Co-processor (CP0) Instructions

Co-processor 0 instructions perform operations on the System Control Co-processor (CP0) registers to manipulate the memory management, bus programmability, timer, and exception handling facilities of the processor. Memory management, bus programmability, and exception handling are described in later chapters.

Notes

Table 2.16 summarizes the instructions available to work with CP0.

Instruction	Format and Description
Move To CP0	MTC0 rt, rd Store contents of CPU register rt into register rd of CP0. This follows the convention of store operations.
Move From CP0	MFC0 rt, rd Load CPU register rt with contents of CP0 register rd.
Read Indexed TLB Entry	TLBR† Load EntryHi and EntryLo registers with TLB entry pointed at by Index register.
Write Indexed TLB Entry	TLBWI† Load TLB entry pointed at by Index register with contents of EntryHi and EntryLo registers.
Write Random TLB Entry	TLBWR† Load TLB entry pointed at by Random register with contents of EntryHi and EntryLo registers.
Probe TLB for Matching Entry	TLBP† Load Indexed register with address of TLB entry whose contents match EntryHi and EntryLo. If no TLB entry matches, set high-order bit of Index register.
Restore From Exception	RFE Restore previous interrupt mask and mode bits of status register into current status bits. Restore old status bits into previous status bits.
†These operations are undefined/reserved in the RC36100, which does not include an on-chip TLB.	

Table 2.16 System Control Co-Processor (CP0) Operations

RC36100 Opcode Encoding

◆ *Table 2.17 shows the opcode encoding for the MIPS architecture.*

Notes

28..26 **OPCODE**

31..29	0	1	2	3	4	5	6	7
0	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	COP3	†	†	†	†
3	†	†	†	†	†	†	†	†
4	LB	LH	LWL	LW	LBU	LHU	LWR	†
5	SB	SH	SWL	SW	†	†	SWR	†
6	LWC0	LWC1	LWC2	LWC3	†	†	†	†
7	SWC0	SWC1	SWC2	SWC3	†	†	†	†

2..0 **SPECIAL**

5..3	0	1	2	3	4	5	6	7
0	SLL	†	SRL	SRA	SLLV	†	SRLV	SRAV
1	JR	JALR	†	†	SYSC ALL	BREA K	†	†
2	MFHI	MTHI	MFLO	MTLO	†	†	†	†
3	MULT	MULTU	DIV	DIVU	†	†	†	†
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	†	†	SLT	SLTU	†	†	†	†
6	†	†	†	†	†	†	†	†
7	†	†	†	†	†	†	†	†

18..16 **BCOND**

20..19	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ						
1								
2	BLTZAL	BGEZAL						
3								
4								

23..21 **COPz**

25..24	0	1	2	3	4	5	6	7
0	MF		CF		MT		CT	
1	BC	†	†	†	†	†	†	†
2								
3								

18..16 **Co-Processor Specific Operations**

Notes

20..19	0	1	2	3	4	5	6	7
0	BCzF	BCzT						
1								
2								
3								

CP0

	2..0							
4..3	0	1	2	3	4	5	6	7
0		TLBR	TLBWI				TLBWR	
1	TLBP							
2	RFE							
3								

Table 2.17 Opcode Encoding

Notes

Cache Architecture

Notes

Introduction

The RC36100 achieves its high standard of performance by combining a fast, efficient execution engine (that of the RISCore32 series) with high-memory bandwidth, supplied from its large internal instruction and data caches. These caches insure that the majority of processor execution occurs at the rate of one instruction per clock cycle, and serve to decouple the high-speed execution engine from slower, external memory resources.

Portions of this chapter review the fundamentals of general cache operation and may be skipped by readers already familiar with these concepts. This chapter also discusses the particular organization of the on-chip caches of the RC36100. However, as these caches are managed by the RC36100 itself, the system designer does not typically need to be explicitly aware of this structure.

Fundamentals of Cache Operation

High-performance microprocessor-based systems frequently borrow from computer architecture principles long used in mini-computers and mainframes. These principles include instruction execution pipelining (discussed in Chapter 2) and instruction and data caching.

A cache is a high-speed memory store that contains the instructions and data most likely to be needed by the processor. That is, rather than implement the entire memory system with zero wait-state memory devices, a small zero wait-state memory is implemented. This memory, called a cache, contains the instructions/data most likely to be referenced by the processor. If indeed the processor issues a reference to an item contained in the cache, then a zero wait-state access is made; if the reference is not contained in the cache, then the longer latency associated with the true processor memory is incurred. The processor will achieve its maximum performance as long as its references “hit” (are resident) in the cache.

Caches rely on the principles of software locality. These principles state that when a data/instruction element is used by a processor it, and its close neighbors, are likely to be used again soon. The cache is then constructed to keep a copy of instructions and data referenced by the processor so that subsequent references occur with zero wait-states.

Since the cache is typically many orders of magnitude smaller than main memory or the virtual address space, each cache element must contain both the data (or instruction) required by the processor and the information that can be used to determine whether a cache “hit” occurs. This information, called the cache “TAG”, is typically some or all of the address in main memory of the data item contained in that cache element as well as a “Valid” flag for that cache element. Thus, when the processor issues an address for a reference, the cache controller compares the TAG with the processor address to determine whether a hit occurs.

To minimize cost while maintaining high-performance, the RC36100 integrates a reasonable amount of cache internal to the chip, eliminating the cost and complexity of external caches.

RC36100 Cache Organization

There are a number of algorithms possible for managing a processor cache. This section describes the cache organization and operation of the RC36100.

Basic Cache Operation

When the processor makes a reference, its 32-bit internal physical address bus contains the address it desires. The processor address bus is split into two parts: (1) the low-order address bits that specify a location in the cache to access and (2) the remaining high-order address bits that contain the value expected from the cache TAG.

Notes

Thus, both the instruction/data element and the cache TAG are simultaneously fetched from the cache memory. If the value read from the TAG memories is the same as the high-order address bits, a cache hit occurs and the processor is allowed to operate on the instruction/data element retrieved. Otherwise, a cache miss is processed. This operation is illustrated in Figure 3.1.

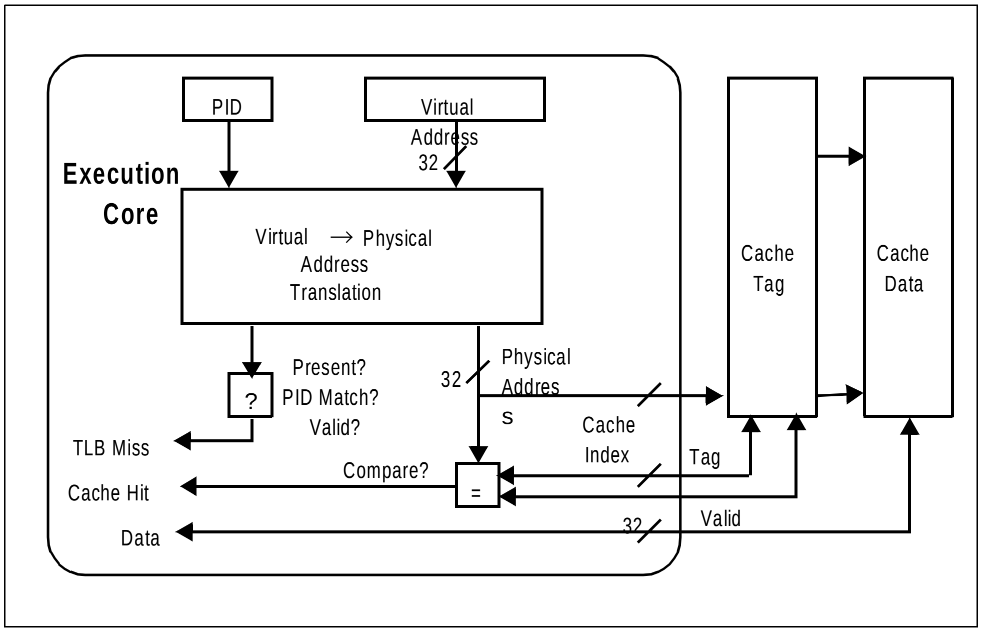


Figure 3.1 Cache Line Selection

To maximize performance, the RC36100 implements a Harvard Architecture caching strategy. That is, there are two separate caches: one contains instructions (operations), and the other contains data (operands). By separating the caches, higher overall bandwidth to the execution core is achieved, and thus higher performance is realized.

Memory Address to Cache Location Mapping

The RC36100's caches are direct-mapped. That is, each main memory address can be mapped to (contained in) only one particular cache location. This is different from set-associative mappings, where each main memory location has multiple candidate cache locations for address mapping.

This organization, coupled with the relatively large cache sizes resident on the RC36100, achieve extremely high hit rates while maximizing speed and minimizing complexity and power consumption.

Cache Addressing

The address presented to the cache and cache controller is that of the physical (main) memory element to be accessed. That is, the virtual address to physical address translation is performed by the memory management unit prior to the processor issuing its reference address.

Some microprocessors utilize virtual indexing and virtual tagging in the cache, where the processor virtual address is used to specify the cache element to be retrieved. This type of cache structure complicates software and slows embedded applications:

- *When the processor performs a context switch, a virtually tagged cache must be flushed. This is because two different tasks can use the same virtual address but mean totally different physical addresses. This cache flushing for a large cache dramatically slows context switch performance.*
- *Software must be aware of and specifically manage against "alias" problems. An alias occurs when two different virtual addresses correspond to the same physical address. If that occurs in a virtually indexed cache, then the same data element may be present in two different cache locations. If one virtual address is used to change the value of that memory location, and a different address used to read it later, then the second reference will not get the current value of that data item.*

By providing for the virtual-to-physical address translation in the processor pipeline, physical cache addressing is used with no inherent performance penalty.

To support cache locking, the RC36100 allows the kernel software to select certain high-order physical address bits to replace normal high-order cache index lines. This separates the cache into two portions: a lower portion, which services physical addresses below the high-order address; and a higher portion, which services physical addresses above the high-order address. Even when this mode is enabled, the RC36100 implements direct-mapped, physically indexed, physically tagged caches.

Write Policy

The RC36100 utilizes a write-through cache. That is, whenever the processor performs a write operation to memory, then both the cache (data and TAG fields) and main memory are written. If the reference is uncacheable, then only main memory is written.

To minimize the delays associated with updating main memory, the RC36100 contains a 4 element write buffer. The write buffer captures the target address and data value in a single processor clock cycle, and subsequently performs the main memory write at its own, slower rate. The write buffer can FIFO up to 4 pending writes, as described in a later chapter.

Partial Word Writes

In the case of partial word writes (store operations of less than 32-bits), the RC36100 operates by performing a read-modify-write sequence in the cache: the store target address is used to perform a cache fetch; if the cache “hits”, then the partial word data is merged with the cache and the cache is updated. If the cache read results in a hit, the memory interface will see the full word write, rather than the partial word. This allows the designer to observe the actual activity in the on-chip caches.

If the cache lookup of a partial word write “misses” in the cache, then only main memory is updated.

Instruction Cache Line Size

The “line size” of a cache refers to the number of cache elements mapped by a single TAG element. In the RC36100, the instruction cache line size is 16 bytes, or four words.

This means that each cache line contains four adjacent words from main memory. In order to accommodate this, an instruction cache miss is processed by performing a quad word (block) read from the main memory, as discussed in a later chapter. This insures that a cache line contains four adjacent memory locations. Note that since the instruction cache is typically never written into directly by user software, the larger line size is permissible. If software does explicitly store into the instruction cache (perform store operations with the caches “swapped”), the programmer must insure that either the written lines are left invalidated, or that they contain four adjacent instructions.

Block refill uses the principle of locality of reference. Since instructions typically execute sequentially, there is a high probability that the instruction address immediately after the current instruction will be the next instruction. Block refill then brings into the cache those instructions immediately near the current instruction, resulting in a higher instruction cache hit rate.

Block refill also takes advantage of the difference between memory latency and memory bandwidth. Memory latency refers to the amount of time required to perform a processor request, while bandwidth refers to the rate at which subsequent transfers can occur. Factors that affect memory latency include address decoding, bus arbitration, and memory pre-charge requirements; factors which maximize bandwidth include the use of page mode or nibble mode accesses, memory interleaving, and burst memory devices.

The processing of a quad word read is discussed in a later chapter; however, it is worth noting that the RC36100 can support either true “burst” accesses or can utilize a simpler, slower memory protocol for quad word reads. Also note that the variable bus sizing capability of the RC36100 means that block reads can occur from 8- or 16-bit memory systems. This includes the case of instruction fetches; the bus interface unit will automatically translate the block read protocol into a larger number of sub-word reads, depending on the memory width programmed for the target memory location.

Finally, note that the RC36100 performs “streaming” during instruction cache refill. That is, the processor will simultaneously refill the instruction cache and execute the incoming instructions. Streaming contributes an average of 5% of performance.

Notes

Data Cache Line Size

The data cache line size is different from that of the instruction cache, based on differences in their use. The data cache is organized as a line size of one word (four bytes).

This is optimal for the write policy of the data cache: since an individual cache word may be written by a software store instruction, the cache controller cannot guarantee that four adjacent words in the cache are from adjacent memory locations. Thus each word is individually tagged. The partial word writes (less than 4 bytes) are handled as a read-modify-write sequence, as described above.

Although the data cache line size is one word, the system may elect to perform data cache updates using quad word reads (block refill). The performance of the data cache update options can be measured in an actual system, by turning on the two different options under software control and using the DBlock Refill (‘DBR’) option in the Coprocessor 0 Cache Configuration Register (for more information on this register, see Chapter 5, “Coprocessor 0 Register Set”).

Summary

The on-chip caches of the RC36100 family can be thought of as constructed from discrete devices around the RISCore32 series. Figure 3.2 shows the block diagram of the cache interface for the RC36100.

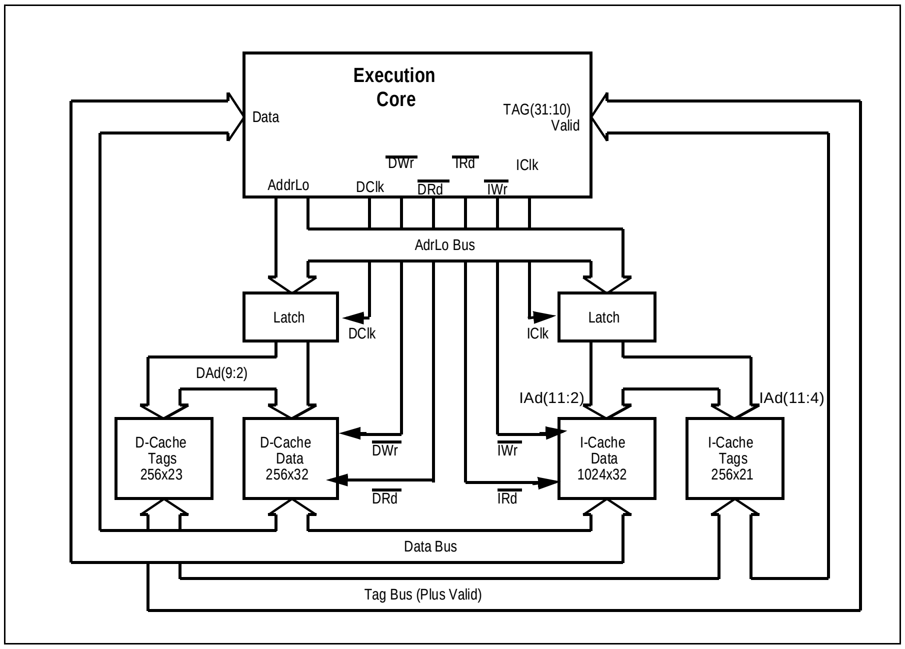


Figure 3.2 RC36100 Execution Core and Cache Interface

Cache Operation

The operation of the on-chip caches is very straightforward, and is automatically handled by the processor.

Basic Cache Fetch Operation

As with the RISCore32 series, the RC36100 can access both the instruction and data caches in a single clock cycle, resulting in high bandwidth to the execution core. It does this by time multiplexing the cycle in the cache interface:

- *During the first phase, a data cache address is presented, and a previous instruction cache read is completed.*
- *During the second phase, the data cache is read into the processor (or written by the processor). Also, the instruction cache is addressed with the next desired instruction.*
- *During the first phase of the next cycle, the instruction fetch begun in the previous phase is completed and a new data transaction is initiated.*

Notes

This operation is illustrated in Figure 3.3 on page 5. As long as the processor hits in the cache, and no internal stall conditions are encountered, it will continue to execute *run* cycles. A run cycle is defined to be a clock cycle in which forward progress in the processor pipeline occurs. Note that data in the cache is organized into 32-bit words, regardless of the width associated with main-memory from which the datum was taken. Thus, cache hits can retrieve a full 32-bits in a single cycle, minimizing the performance impact of the narrower memory system.

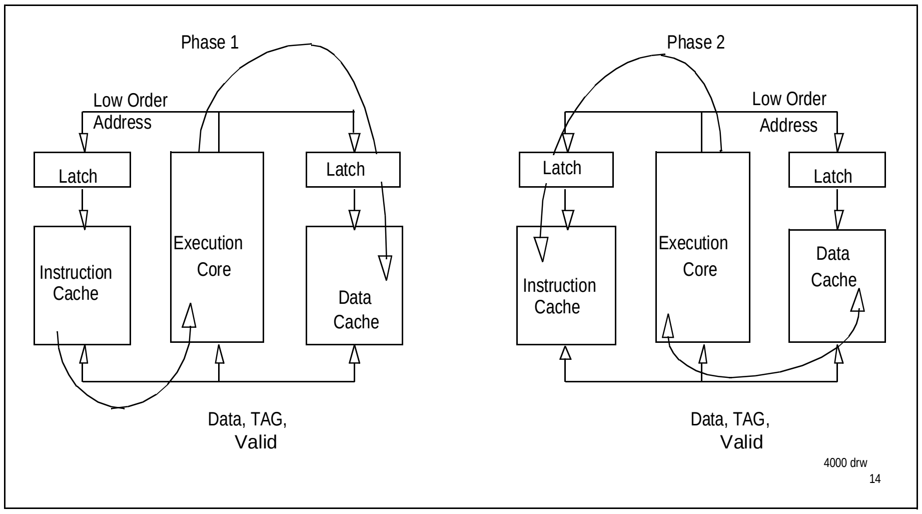


Figure 3.3 Phased Access of Instruction and Data Caches

Cache Miss Processing

In the case of a cache miss (due to either a failed tag comparison or because the processor issued an uncacheable reference), the main memory interface (discussed in a later chapter) is invoked. If, during a given clock cycle, both the instruction and data cache miss, the data reference will be resolved before the instruction cache miss is processed.

While the processor is waiting for a cache miss to be processed, it will enter *stall* cycles until the bus interface unit indicates that it has obtained the necessary data.

When the bus interface unit returns the data from main memory, it is simultaneously brought to the execution unit and written into the on-chip caches. This is performed in a processor *fixup* cycle.

During a fixup cycle, the processor re-issues the cache access that failed; this occurs by having the processor re-address the instruction and data caches, so that the data may be written into the caches and brought into the execution core. If the cache miss was due to an uncacheable reference, the write is not performed, although a fixup cycle does occur to allow the data to be brought into the execution core.

Instruction Streaming

A special feature of the RC36100 is utilized when performing block reads for instruction cache misses. This process is called *instruction streaming.* Instruction streaming is simultaneous instruction execution and cache refill.

As the block is brought in, the processor refills the instruction cache. Execution of the instructions within the block begins when the instruction corresponding to the cache miss is returned by the bus interface unit to the execution core. Execution continues until the end of the block is reached (in which case normal execution is resumed), or until some event forces the processor core to discontinue execution of that stream. These events include:

- *Taken branches*
- *Data cache miss*
- *Internal stalls (TLB miss, multiply/divide interlock)*
- *Exceptions*

When one of these events occur, the processor re-enters simple cache refill until the rest of the block has been written into the cache, to insure that one TAG describes all four adjacent words.

Notes

Cacheable References

Chapter 4 explains how the processor determines whether a particular reference (either instruction or data) is to a memory location that may reside in the cache. The fundamental mechanism is that certain virtual addresses are considered to be “cacheable”. If the processor attempts to make a reference to a cacheable address, then it will employ its cache management protocol through that reference. Otherwise, the cache will be bypassed, and the execution engine core will directly communicate with the bus interface unit to process the reference.

Whether a given reference should be cacheable or not depends on the application and on the target of the reference. Generally, I/O devices should be referenced as uncacheable data; for example, if software was polling a status register, and that register was cached, then it would never see the device update the status (note that most compiler suites support the “volatile” data type to insure that the I/O device status register data in this case never gets allocated into an internal register). In the RC36100, the cacheability of the on-chip registers in the I/O and peripheral devices is automatically selected to be “non-cacheable”.

There may be other instances where the uncacheable attribute is appropriate. For example, software which directly manipulates or flushes the caches can not be cached; similarly, boot software can not rely on the state of the caches, and thus must operate uncached at least until the caches are initialized.

Software Directed Cache Operations

In order to support certain system requirements, the RC36100 provides mechanisms for software to explicitly manipulate the caches. These mechanisms support diagnostics, cache and memory sizing, and cache flushing. In general, these mechanisms are enabled/disabled through the use of the Status Register in CP0.

The primary mechanisms for supporting these operations are cache swapping and cache isolation. Cache swapping forces the processor to use the data cache as an instruction cache, and vice versa. It is useful for allowing the processor to issue store instructions which cause the instruction cache to be written. Cache isolation causes the current data cache to be “isolated” from main memory; store operations do not cause main memory to be written, and all load operations “hit” in the data cache.

These mechanisms are enabled through the use of the “IsC” (Isolate Cache) and “SwC” (Swap Cache) bits of the status register, which resides in the on-chip System Control Co-Processor (CP0). *The 5 instructions which immediately precede and succeed these operations must not be cacheable, so that the actual swapping/isolation of the cache does not disrupt operation.*

Cache Sizing

It is possible for software to determine the amount of cache resident on any given RISCore3000 family-based chip (note that the RC3041, RC3051, RC3052, and RC3081 each feature differing amounts of cache on chip). Having software determine the size of the cache at boot time, rather than building static values into the software, allows for maximum flexibility in using various members of the RISCore3000 family, including future devices.

Cache sizing in an RC36100 is performed much like traditional memory sizing algorithms, but with the cache isolated. This avoids side-effects in memory from the sizing algorithm, and allows the software to use the “Cache Miss” bit of the status register in the sizing algorithm.

Notes

To determine the size of the instruction cache, software should:

1. Swap Caches (not needed for D-Cache sizing)
2. Isolate Caches
3. Write a value at location 8000_0000
4. Write a value at location 8000_0200 (8000_0000 + 512B)

Read location 8000_0000.
Examine the CM (Cache_Miss) bit of the status register; if it indicates a cache miss, then the cache is 512B; otherwise, the cache is 1kB or larger.

5. Write a value at location 8000_0400 (8000_0000 + 1kB)

Read location 8000_0000.
Examine the CM (Cache_Miss) bit of the status register; if it indicates a cache miss, then the cache is 1KB; otherwise, the cache is 2KB or larger.

6. etc...

Of course a more generalized algorithm could be developed to determine the cache size; this may be desirable for compatibility with other RISCore3000 family members. However, any algorithm will probably include the Swap and Isolate of the Instruction Cache, and the use of the Cache Miss bit. Sizing the data cache is done with a similar algorithm, although the caches need not be swapped, and smaller cache sizes need to be considered.

Note that this software should operate as uncached. Once this algorithm is done, software should return the caches to their normal state by performing either a complete cache flush or an invalidate of those cache lines modified by the sizing algorithm.

Cache Flushing

Cache flushing refers to the act of invalidating (indicating a line does not have valid contents) lines within either the instruction or data caches. Flushing must be performed before the caches are first used as real caches, and might also be performed during main memory page swapping or at certain context switches (note that the RC3051 family implements physically addressed caches, so that cache flushing at context switch time is not generally required).

The basic concept behind cache flushing is to have the “Valid” bit of each cache line set to indicate invalid. This is done in the RC36100 by having the cache isolated, and then writing a partial word quantity into the current data cache. Under these conditions, the CPU will negate the “Valid” bit of the target cache line.

Again, this software should operate as uncached. To flush the data cache:

1. Isolate Caches
2. Perform a byte write every 4 bytes, starting at location 0, until 256 such writes have been performed (128 in the RC3041, more for other RISCore3000 family members).
3. Return the data cache to its normal state by clearing the IsC bit.

To flush the instruction cache:

1. Swap Caches
2. Isolate Caches
3. Perform a byte write every 16 bytes (based on the instruction cache line size of 16 bytes). This should be done until each line (256 lines in the RC36100, more or less for other RISCore3000 family devices) have been invalidated. Note that treating the RC36100 as if it had larger on-chip caches, and flushing/invalidating more than 256 lines is acceptable though less efficient.
4. Return the caches to their normal state (unswapped and not isolated).

To minimize the execution time of the cache flush, this software should probably use an “unrolled” loop. That is, rather than have one iteration of the loop invalidate only one cache line, each iteration should invalidate multiple lines. This spreads the overhead of the loop flow control over more cache line invalidates, thus reducing execution time.

Also, of course it is preferable to use the cache sizing algorithm described earlier to determine the number of lines to be flushed.

Forcing Data Into the Caches

Using these basic tools, it is possible to have software directly place values into the caches. When combined with appropriate software techniques, this could be used to “lock” values into the on-chip caches, by insuring that software does not issue other cacheable address references which may displace these locked values.

Notes

In order to force values into a cache, the cache should be Isolated. If software is trying to write instructions into the instruction cache, then the caches should also be swapped.

When forcing values into the instruction cache, software must take care with regards to the line size of the instruction cache. Specifically, a single TAG and Valid field describe four words in the instruction cache; software must then insure that any instruction cache line tagged as Valid actually contains valid data from all four words of the block.

Cache-Locking Operation

The RC36100 implements the ability to segregate the caches into 2 or 4 portions, or to allow it to operate as a normal single contiguous entity. Either or both the instruction and data cache can be run in any split or non-split mode independently.

As an example, splitting the cache into halves or quarters allows interrupt service routines and data to be locked into part of the cache, while the remainder of the cache is used for the user program and data.

If run in the normal mode (as a single contiguous entity), the cache index (used internally to address the Cache Data and Tag RAMs) is derived solely from the low-order physical address bits. For example, the cache index for the data cache is PhysAddr(9:2); and for the instruction cache, the cache index is PhysAddr(11:2).

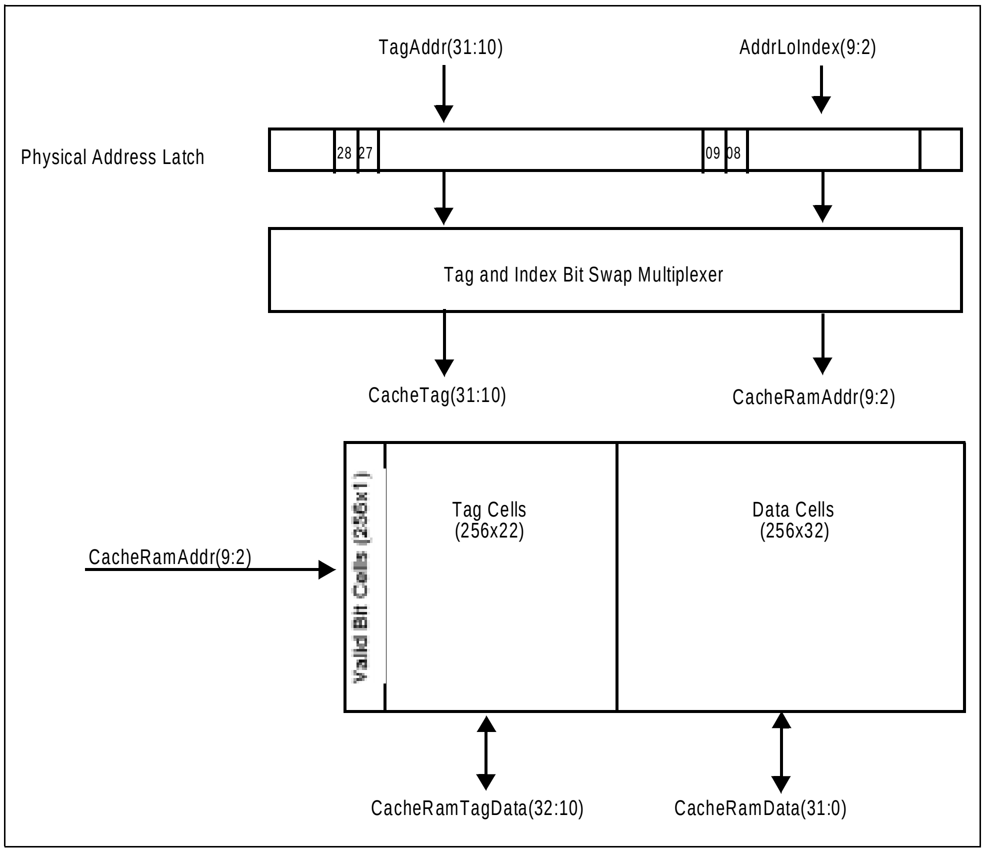


Figure 3.4 RC36100 Instruction Cache Index Address Path

In the normal mode case, a reference with the same low-order PhysAddr bits but different high-order PhysAddr tags will cause the current cache contents to be replaced. For example, location 0x0000_1008 will be entered into the line at cache index 0x0000; if that line previously was cached with main memory location 0x0000_0008, it would be replaced with new data and tag. Any address which is modulo 4kB (for instance 0x1004_0008) could cause replacement of that cache line.

Notes

On the other hand, in the split modes, the system software can instruct the cache controller to use either or both of PhysAddr(28:27) as the uppermost two index bits (2 or 4 portions). In this case, the cache simultaneously direct-maps multiple distinctly different memory spaces. The 10 bits for the instruction cache index can be constructed as PhysAddr(28, 10:2), for example.

When the caches are operated in split mode, typically the MSBs going to main memory (bits 31-29) are masked out from the physical address decode. On the RC36100, the physical address decode is part of either the DRAM Controller's or the Memory/IO Controller's Page Register and Page Mask Register (which ever one is used to control main memory). Masking out the MSBs allows physical RAM space to be contiguous (for instance contained within a 1MB block), while the virtual program space can vary the MSBs.

For instance the virtual program space can consist of a 512KB block beginning at 0x0000_0000 and a second virtual program space with a 512KB block beginning at 0x1004_0000. The two virtual addresses will translate (see the next chapter for more details) to physical addresses 0x4000_0000 and 0x5004_0000, respectively, as far as cache memory is concerned. Since bits 31-29 are ignored by the main memory controller, the two physical addresses are effectively 0x0000_0000 and 0x0004_0000 as far as main memory RAM is concerned. Thus by using the Page Mask Register, the caches can see 2 or 4 blocks of address spaces, while main memory sees a single large block of address space.

To continue the instruction cache example, the upper 2kB portion of the I-cache services physical addresses in the range of 0x1000_0000 and above; physical addresses in the range 0x0fff_ffff and below are serviced by the lower 2kB I-cache portion.

In this example, the instruction at physical location 0x1004_0008 will *not* replace the contents of the line which holds memory location 0x0000_0008. These two portions of software will not interfere with each other in the caches. The software developer typically specifies the address region for code in either the kernel, or with the linker. This mechanism allows the programmer to separate code into portions and independently lock them, without requiring page management software or complex operating system software.

Physical address 0x0000_0008 is accessed via Kuseg (explained in the next chapter), and is typically in the area of the exception vector. Physical address 0x1000_0008 is spaced 256MB higher in memory. In this system, system tasks operating higher in kuseg or in kseg0 do not “knock out” the exception service code, effectively locking this time critical code into one half of the on-chip cache.

Table 3.1 on page 11 shows the correlation between physical address lines, cache index lines, and cache sub-segments supported by the RC36100. Figure 3.5, Figure 3.6, and Figure 3.7 on page 11 shows the mapping of physical addresses to cache when the cache is 1, 2, or 4 portions.

Note that these tables and drawings assume that the code operates out of kuseg (explained in the next chapter). Since the RC36100 implements 32-bit virtual and physical addressing, the patterns shown repeat every time a very high-order (PhysAddr(29) and above) is changed; thus, there are 8 such copies of each cache region, separated by 512MB each. The tables and example assume that PhysAddr(31:29) are all '0' throughout system software. However, memory spaces larger than 512MB are rarely used with embedded systems, the example in these tables will suffice for almost all systems.

Notes

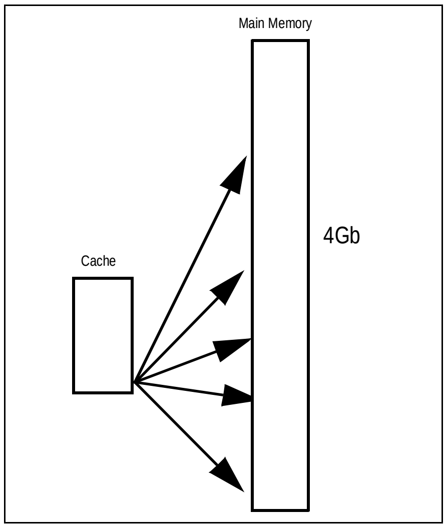


Figure 3.5 RC36100 Cache in One Portion

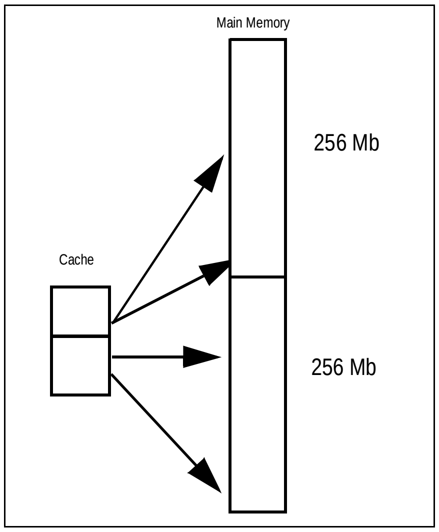


Figure 3.6 RC36100 Cache in Two Portions

Notes

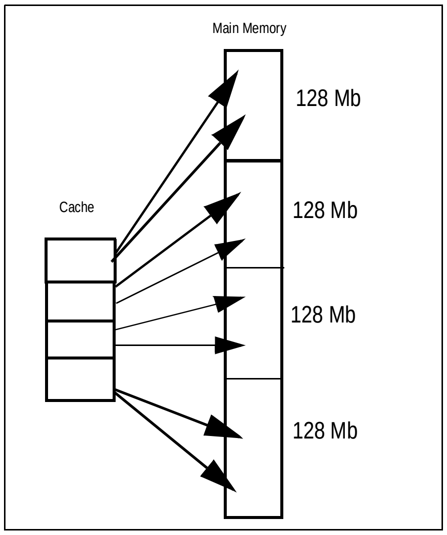


Figure 3.7 RC36100 Cache in Four Portions

Cache IndexAddr(11)	Cache IndexAddr(10)	Physical Address Range	Cache Size
PhysAddr(11)	PhysAddr(10)	0x0000_0000 0x1FFF_FFFF	4kB
PhysAddr(28)	PhysAddr(10)	0x0000_0000 - 0x0FFF_FFFF	2kB
		0x1000_0000 - 0x1FFF_FFFF	2kB
PhysAddr(28)	PhysAddr(27)	0x0000_0000 - 0x07FF_FFFF	1kB
		0x0800_0000 - 0x0FFF_FFFF	1kB
		0x1000_0000 - 0x17FF_FFFF	1kB
		0x1800_0000 - 0x1FFF_FFFF	1kB
This table describes byte-addressable caches, with the lsb of the cache index == 2.			

Table 3.1 Instruction Cache to Address Mapping under Various Cache Locking Conditions

Notes

Cache Index Addr(9)	Cache Index Addr(8)	Physical Address Range	Cache Size
PhysAddr(9)	PhysAddr(8)	0x0000_0000 - 0x1FFF_FFFF	1kB
PhysAddr(28)	PhysAddr(8)	0x0000_0000 - 0x0FFF_FFFF	512B
		0x1000_0000 - 0x1FFF_FFFF	512B
PhysAddr(28)	PhysAddr(27)	0x0000_0000 - 0x07FF_FFFF	256B
		0x0800_0000 - 0x0FFF_FFFF	256B
		0x1000_0000 - 0x17FF_FFFF	256B
		0x1800_0000 - 0x1FFF_FFFF	256B
This table describes byte-addressable caches, with the lsb of the cache index == 2.			

Table 3.2 Data Cache to Address Mapping under Various Cache Locking Conditions

Summary

The on-chip caches of the RC36100 are key to the inherent performance of the processor. The RC36100 design, however, does not require the system designer (either software or hardware) to explicitly manage this important resource, except to correctly choose virtual addresses which may or may not be cached and to flush the caches at system boot. This contributes to both the simplicity and performance of an RC36100 system.

Virtual-to-Physical Address Translation and Address Map

Notes

The RC36100 provides the same basic virtual-to-physical address translation as the rest of the RISCore3000 family base versions (the RC3041, RC3051, RC3052, and RC3081). These devices provide segment-based virtual-to-physical address translation, and support the segregation of kernel and user tasks without requiring extensive virtual page management.

The extended versions of the RISCore3000 family (the RC3051 RC3052, and RC3081) provide a full featured memory management unit (MMU). The extended MMU uses an on-chip translation lookaside buffer (TLB) and dedicated registers in CP0 to provide for software management of page tables. There is no Extended Architecture version of the RC36100.

This chapter describes the operating states of the processor (kernel and user), and describes the virtual-to-physical address translation mechanisms provided in the RC36100.

Virtual Memory in the RISCore32 series Architecture

There are two primary purposes of the memory management capabilities of the RISCore32 series Architecture:

- *Various areas of main memory can have individual sets of attributes associated with them. For example, some segments may be indicated as requiring kernel status to be accessed; others may have cacheable or uncacheable attributes. The virtual-to-physical address translation establishes the rules appropriate for a given virtual address. The RC36100 memory manager provides for these mechanisms, without requiring the use of a TLB.*
- *The virtual memory system can be used to logically expand the physical memory space of the processor, by translating addresses composed in a large virtual address space into the physical address space of the system. This is particularly important in applications where software may not be explicitly aware of the hardware resources of the processor system, and includes applications such as X-Window display systems. These types of applications may be better served by the “E” (extended architecture) versions of the RISCore3000 family. On the other hand, certain real-time operating systems offer similar functionality without requiring an MMU; for example, the IDT/c tool chain supports position-independent code without requiring a page fault manager in the operating system.*

Figure 4.1 shows the virtual address format. The most significant 20 bits of the 32-bit virtual address are called the virtual page number, or VPN. In the extended architecture versions, the VPN allows mapping of virtual addresses based on 4kB pages; in the base versions (and thus in the RC36100), only the three highest bits (segment number) are involved in the virtual-to-physical address translation.

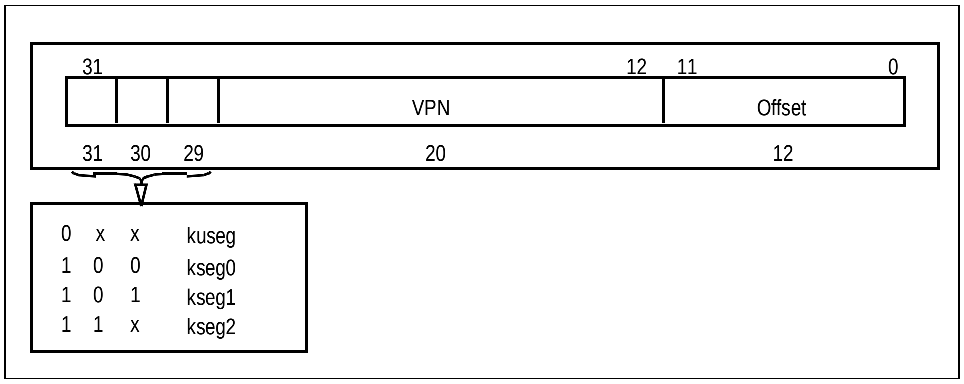


Figure 4.1 Virtual Address Format

Notes

The three most significant bits of the virtual address identify which virtual address segment the processor is currently referencing; these segments have associated with them the mapping algorithm to be employed, and whether virtual addresses in that segment may reside in the cache. The translation of the virtual address to an equivalent privilege level/segment is the same for the base and extended versions of the architecture.

Privilege States

The RC36100 provides for two unique privilege states: the "Kernel" mode, which is analogous to the "supervisory" mode provided in many systems, and the "User" mode, where non-supervisory programs are executed. Kernel mode is entered whenever the processor detects an exception; when a Restore From Exception (RFE) instruction is executed, the processor will return either to its previous privilege mode or to User mode, depending on the state of the machine and when the exception was detected.

User Mode Virtual Addressing

While the processor is operating in User mode, a single, uniform virtual address space **(kuseg)** of 2GB is available for Users. All valid user-mode virtual addresses have the most significant bit of the virtual address cleared to 0. An attempt to reference a Kernel address (most significant bit of the virtual address set to 1) while in User mode will cause an Address Error Exception. Kuseg begins at virtual address 0 and extends linearly for 2GB. This segment is typically used to hold user code and data, and the current user processes.

Also note that the physical address space corresponding to kuseg is independent of the physical address spaces of the various kernel only segments. Thus, systems can be constructed which preclude user tasks from affecting kernel memory. On the other hand, simple systems can, by virtue of the address decode, compress the mapping into a single address region.

Kernel Mode Virtual Addressing

When the processor is operating in Kernel mode, four distinct virtual address segments are simultaneously available. The segments are:

- ***kuseg.*** *The kernel may assert the same virtual address as a user process, and have the same virtual-to-physical address translation performed for it as the translation for the user task. This facilitates the kernel having direct access to user memory regions. The virtual-to-physical address translation, including the Port Size attributes, is identical with User mode addressing to this segment.*
- ***kseg0.*** *Kseg0 is a 512MB segment, beginning at virtual address 0x8000_0000. This segment is always translated to a linear 512MB region of the physical address space starting at physical address 0. All references through this segment are cacheable.*
  When the most significant three bits of the virtual address are "100", the virtual address resides in kseg0. The physical address is constructed by replacing these three bits of the virtual address with the value "000". As these references are cacheable, kseg0 is typically used for kernel executable code and some kernel data.
- ***kseg1.*** *Kseg1 is also a 512MB segment, beginning at virtual address 0xa000_0000. This segment is also translated directly to the 512MB physical address space starting at address 0. All references through this segment are uncacheable.*
  When the most significant three bits of the virtual address are "101", the virtual address resides in kseg1. The physical address is constructed by replacing these three bits of the virtual address with the value "000". Unlike kseg0, references through kseg1 are not cacheable. This segment is typically used for I/O registers, boot ROM code, and operating system data areas such as disk buffers.
- ***kseg2.*** *This segment is analogous to kuseg, but is accessible only from kernel mode. This segment contains 1GB of linear addresses, beginning at virtual address 0xc000_0000. As with kuseg, the virtual-to-physical address translation depends on whether the processor is a base or extended architecture version.*

Notes

When the two most significant bits of the virtual address are "11," the virtual address resides in the 1024MB segment kseg2. The virtual-to-physical translation is done either through the TLB (extended versions of the processor) or through a direct segment mapping (base versions). An operating system would typically use this segment for stacks, per-process data that must be re-mapped at context switch, user page tables, and for some dynamically allocated data areas.

Base versions of the RISCore3000 family (including the RC36100) are distinguishable from extended versions in software by examining the TS (TLB Shutdown) bit of the Status Register after reset, before the TLB is used. If the TS bit is set (1) immediately after reset, indicating that the TLB is non-functional, then the current processor is a base version of the architecture. If the TS bit is cleared after reset, then the software is executing on an extended architecture version of the processor.

The PRId register—described in a later chapter—can be used to distinguish the RC36100 from other members of the RISCore3000 family.

RC36100 address translation

Processors which only implement the base versions of memory management perform direct segment mapping of virtual-to-physical addresses, as illustrated in Figure 4.2. Thus, the mapping of kuseg and kseg2 is performed as follows:

- *Kuseg is always translated to a contiguous 2GB region of the physical address space, beginning at location 0x4000_0000. That is, the value "00" in the two highest order bits of the virtual address space are translated to the value "01", and "01" is translated to "10", with the remaining 30 bits of the virtual address unchanged.*
- *Virtual addresses in kseg2 are directly output as physical addresses; that is, references to kseg2 occur with the physical address unchanged from the virtual address.*
- *Virtual addresses in kseg0 and kseg1 are both translated identically to the same physical address region.*

The base versions of the architecture allow kernel software to be protected from user mode accesses, without requiring virtual page management software. User references to kernel virtual address will result in an address error exception.

Note that the special areas of the virtual address space shown in Figure 4.2 are translated to physical addresses identically with the remainder of their virtual address segment. In the RISCore3000 family, these address areas were indicated as "reserved" for compatibility with future devices.

Notes

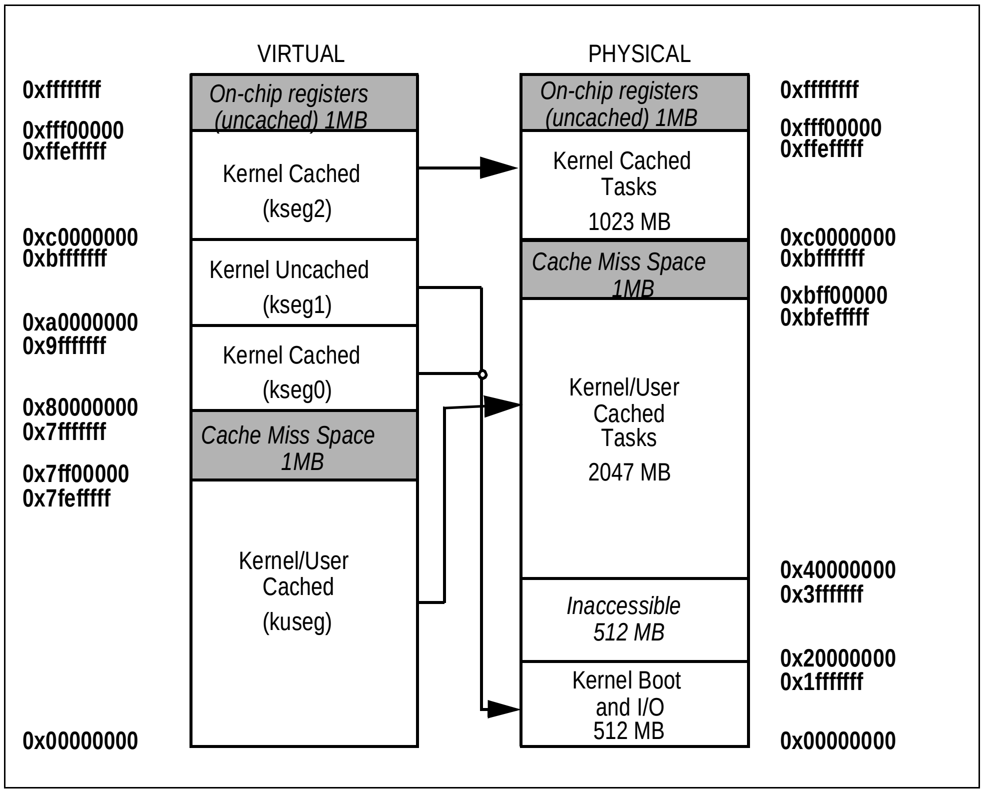


Figure 4.2 Virtual-to-Physical Address Translation in RC36100

Some systems may elect to protect external physical memory as well. That is, the system may include distinct memory devices which can only be accessed from kernel mode. The physical address output determines whether the reference occurred from kernel or user mode, according to Table 4.1. Some systems may wish to limit accesses to some memory or I/O devices to those physical address bits which correspond to kernel mode virtual addresses.

Alternately, some systems may wish to have the kernel and user tasks share common areas of memory. Those systems could choose to have their address decoder ignore the high-order physical address bits, and compress all of memory into the lower region of physical memory. The high-order physical address bits may be useful as privilege mode status outputs in these systems.

Physical Address (31:29)	Virtual Address Segment
‘000’	Kseg0 or Kseg1
‘001’	Inaccessible
’01x’	Kuseg
’10x’	Kuseg
’11x’	Kseg2

Table 4.1 Virtual and Physical Address Relationships in Base Versions

On-Chip Registers

The top 1MB of virtual memory—which resides in the protected kernel space, kseg2—is treated as “non-cacheable” by the cache controller. The rest of kseg2 is treated as cacheable. The on-chip memory controllers and peripherals have their register sets mapped into this address space; these registers need to be uncached to insure proper operation. Table 4.2 shows the address map for the on-chip resources.

Notes

Note that writes to addresses above 0xFFFF_E000 are propagated out to the external bus. However, none of the memory controllers are activated. This feature is provided to facilitate debug and in-circuit emulation equipment. Reads in this address range are propagated to the external bus.

Base Virtual Address	On-chip Resource
0xFFFF 8000	External Debug/Emulator Controller
0xFFFF 9000 0xFFFF A000 0xFFFF B000 0xFFFF C000 0xFFFF D000 0xFFFF E000	Reserved
0xFFFF E100	DRAM Controller
0xFFFF E200	Memory and IO Controller
0xFFFF E300	Internal DMA Controller
0xFFFF E400	External DMA Controller
0xFFFF E500	Internal Debug/Emulator Controller
0xFFFF E600	Reserved
0xFFFF E700	Reserved
0xFFFF E800	Serial Port Interface
0xFFFF E900	Timer Interface
0xFFFF EA00	PIO Interface
0xFFFF EB00	Interrupt Peripheral Interface
0xFFFF EC00	Centronics Interface (P1284 interface)
0xFFFF ED00	Reserved
0xFFFF EE00	
0xFFFF EF00	
0xFFFF F000	

Table 4.2 RC36100 On-Chip Resources and Address Map

As a general rule, the registers residing above 0xFFFF_E000 are 16-bits and, in some cases, 8-bits wide. Thus, technically, these registers should use either halfword or byte unsigned load and store instructions for proper access.

Because of this less-than-a-word access, if the system is big endian, the registers will either need a halfword offset of 0x2 or a byte offset of 0x3. Little endian systems do not need an offset; however, if the software system can ignore or mask the unused bits, then regular word load-and-store instructions may be used. In this case, neither Endianness needs an offset.

Cache Miss Area

The top 1MB of kuseg is also special. In the RC36100, this area is the “Cache Miss” area.

If software attempts to “load” data with a modulo 16 address (lowest 4 address bits == 0), the cache controller will consider the access to have “missed” in the cache, regardless of the current tag contents.

Notes

This operation can speed certain types of data movement operations, especially when the contents of the corresponding main memory area may be updated externally to the processor. For example (See Table 4.3) if the main memory is a FIFO type memory, the code may perform a load to the FIFO address; the memory controller would burst four words into the cache (presuming a data block refill setting of four words) and load word “0” into the target register. The remaining words of the quad word read would be accessed from the cache. Once all four words are consumed, the code would issue another load with an offset of “0”, causing another cache miss process to the FIFO. Burst data movement is faster, since the software does not need to explicitly flush the cache line between bursts, nor does it need to use slower “uncached” single datum transfers.

```
#define FIFO_BASE 0x7FF00000          /* phys addr is 0xBFF00000 */


get_fifo:
li              t0, FIFO_BASE
lw              t1, 0x00(t0)
lw              t2, 0x04(t0)
lw              t3, 0x08(t0)
lw              t4, 0x0C(t0)          /* 13 cached clocks per 4 words */
```

Table 4.3 Example: FIFO load code using FCM memory space.

Summary

The RISCore3000 family architecture provides two models of memory management: a very simple, segment based mapping, found in the base versions of the architecture, and a more sophisticated, TLB-based page mapping scheme, present in the extended versions of the architecture. Each scheme has advantages to different applications. The RC36100 only implements the base version address translation in order to support low-cost systems.

Coprocessor 0 Register Set

Notes

Introduction

The MIPS architecture separates a processor into two (in the case of a device with an on-chip FPA, there are three) functional units: (1) the general purpose CPU, which executes the actual code and remains device compatible and (2) the system control coprocessor (CP0), which manages the machine state, virtual-to-physical address translation, exception handling, and any other device-specific attributes.

Through modifications to CPO, each device can be tailored to the needs of specific applications, and yet, through the compatible CPU unit, software compatibility for the actual application is retained.

Implementation of the RC36100's CP0 is discussed in this chapter. In general, the exception handling methods of the RC36100 are identical to those remaining members of the RISCore3000 family, and the memory management resources are identical to those of the base versions of the RISCore3000 family. In fact, the only significant difference between the RC36100 and the RISCore3000 family is in the implementation of the Cache Control register.

Coprocessor 0 Bus Interface Control

Figure 5.1 illustrates the coprocessor 0 registers found in the RC36100. Note that the MIPS architecture allows the register set of CP0 to vary by implementation; software can easily identify the RC36100 (and its CP0 registers) from other devices by reading the PRId from CP0.

The fields of these registers are described below. Table 5.1 lists the register numbers for the various RC36100 CP0 registers.

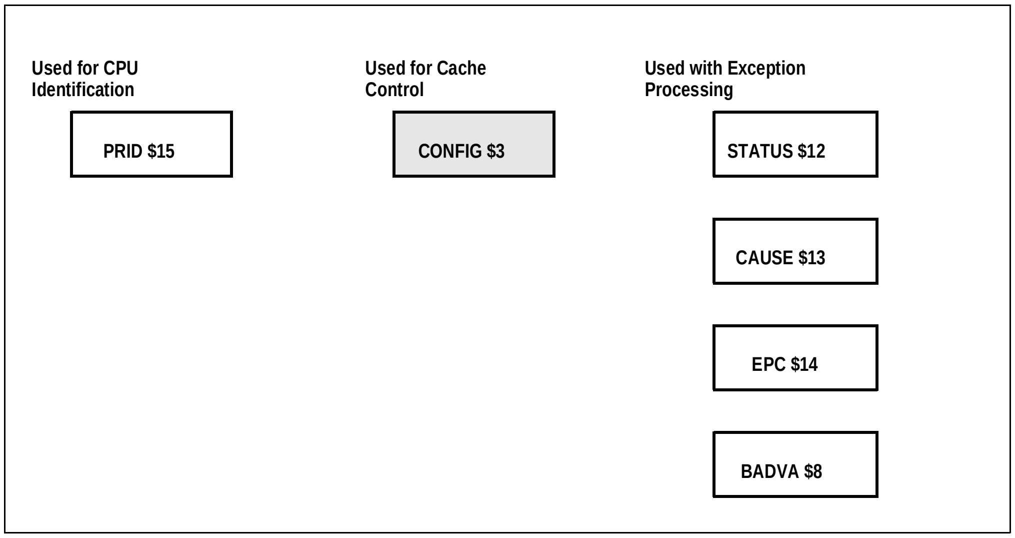


Figure 5.1 RC36100 CPO Registers

Notes

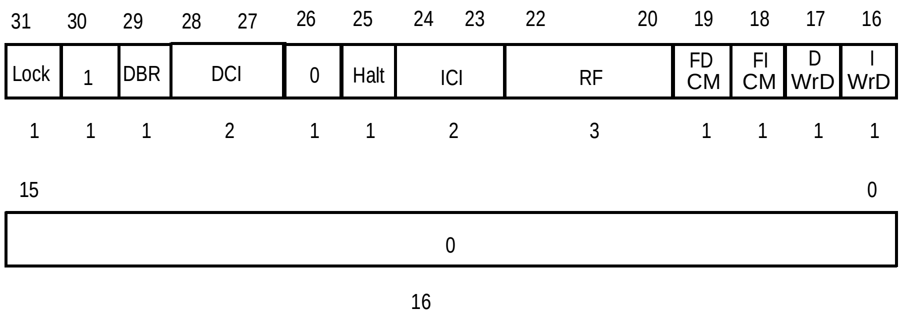
Mnemonic	CP0 Register #	Description
Config	$3	Cache and CPU configuration control
BadVA	$8	Bad Virtual Address for last Addressing Exception
Status	$12	Processor status, control, and diagnostic information
Cause	$13	Cause of current exception/exception state
EPC	$14	Exception Program Counter; return address for exception handler
PrID	$15	Identification information for current processor

Table 5.1 RC36100 CPO Register Addresses

Cache Configuration Register

The cache configuration register allows the kernel to control various operational aspects of the on-chip caches of the RC36100. These features can be used to improve performance and/or implement debug capability for the RC36100. The Config register is both readable and writable.

Figure 5.2 illustrates the various fields of the cache configuration register. The reset defaults for this register insure RISCore3000 compatible operation.



Lock:	Register Write Lock
'1':	Reserved: Must be written as '1'
DBR:	Data Burst Refill Mode
DCI:	Data Cache Index
'0':	Reserved: Must be written as '0'
Halt:	Halt Mode
ICI:	Instruction Cache Index
RF:	Reduced Frequency Mode
FDCM:	Force Data Cache Miss
FICM:	Force Instruction Cache Miss
DWrD:	Data Cache Write Disable
IWrD:	Instruction Cache Write Disable

Figure 5.2 RC36100 Cache Control Register

Lock ('Lock')

The lock bit can be used by the kernel to inhibit subsequent write operations to this register. It is useful in ensuring that operating systems written for other RISCore3000 family-based applications do not inadvertently change the fields of the Cache Configuration register.

Notes

Table 5.2 illustrates the Cache Configuration Register Lock Field. At reset, the register is unlocked (Lock bit is '0'). Thus, the Config register can be written and re-written as the operating system chooses. Once the Lock bit is written with a '1', subsequent writes to the Config register will be ignored.

Value	Action
'0'	Leave Unlocked (Default)
'1'	Lock register from future writes

Table 5.2 RC36100 Cache Configuration Register Lock Field

Reserved-High ('1')

This bit is reserved for testing of the RC36100. At reset, the bit will be set high ('1'). Writes to the Config register must maintain this bit as high ('1').

Reserved-Low ('0')

These fields are reserved for testing and for future RISCore3000 family-based devices. At reset, these bit fields are reset ('0'). Writes to the Config register must maintain these bit fields as low ('0').

DBlockRefill ('DBR')

Table 5.3 indicates the value and action of the DBR. If this bit is set high ('1'), data cache misses will be processed as a quad (four-word) read. If this bit is reset low ('0'), data cache misses will be processed as a single word read. At reset, this bit is reset low ('0').

Value	Action
'0'	Data cache misses use single word refill (default).
'1'	Data cache misses use quad word refill.

Table 5.3 RC36100 DBlockRefill Field

D-CacheIndexControl ('DCI')

This two bit field controls which bits of the physical address provide the high-order data cache index, as described in Chapter 3. Table 5.4 shows the actions of the various bit combinations. At reset, this field is cleared to '00', resulting in normal operation.

Value	DCache Index(9)	DCache Index(8)	Cache Portions
'00'	PhyAddr(9)	PhyAddr(8)	1 (default)
'01'	reserved	reserved	Not useful. Reserved
'10'	PhyAddr(28)	PhyAddr(8)	2
'11'	PhyAddr(28)	PhyAddr(27)	4

Table 5.4 RC36100 D-Cache Index Control Field

Halt Mode ('Halt')

If this bit is set high ('1'), the CPU pipeline will be stalled until either an interrupt is asserted (regardless of current masking) or a reset exception is signalled. If this bit is set low ('0'), the pipeline will continue operation.

If the halt mode is exited, for example by an interrupt, the RF mode (described below) will also be exited. Table 5.5 shows the actions and values of the RC36100 Halt Mode ('Halt').

Notes

Value	Action
'0'	Normal pipeline operation (default).
'1'	Halt until interrupt or Reset

Table 5.5 RC36100 Halt Field

I-CacheIndexControl ('ICI')

This two bit field controls which bits of the physical address provide the high-order instruction cache index, as described in chapter 3. Table 5.6 shows the actions of the various bit combinations. At reset, this field is cleared to '00', resulting in normal operation.

Value	ICache Index(11)	ICache Index(10)	Cache Portions
'00'	PhyAddr(11)	PhyAddr(10)	1 (default)
'01'	reserved	reserved	Not useful. Reserved
'10'	PhyAddr(28)	PhyAddr(10)	2
'11'	PhyAddr(28)	PhyAddr(27)	4

Table 5.6 RC36100 I-Cache Index Control Field

ReduceFrequency ('RF')

This 3 bit field can be used to divide the normal pipeline frequency down to a lower frequency, thus lowering device power consumption. Table 5.7 shows the actions of the various bit settings. At reset, this field is cleared to '000', resulting in normal operation. Similarly, whenever the halt mode is exited, this field will be cleared to '000'.

Value	Action
'000'	Normal Pipeline frequency (default)
'001'	Divide by 2
'010'	Divide by 4
'011'	Divide by 8
'100'	Divide by 16
'101'	Divide by 32
'110'	Divide by 64
'111'	Reserved

Table 5.7 RC36100 Reduced Frequency Mode Field

When a reduced frequency mode is enabled, both the pipeline frequency and the system interface frequency will be reduced by the programmed amount. The minimum allowed frequency is a CPU pipeline frequency of 0.5MHz. To prevent internal synchronization problems, software should always switch from the Normal frequency to a particular divide by frequency or vice-versa. Thus if a switch between 64 and 32 is desired, first switch from 64 to Normal and then to 32.

Note that the "RF" mode also impacts the frequency of the bus interface, including the on-chip devices. System software may need to adjust timer values, baud rates, DRAM refresh, and other frequency sensitive system variables when entering and exiting "RF" mode.

Notes

ForceDCacheMiss ('FDCM')

Table 5.8 shows the values and actions for the RC36100 ForceDCacheMiss field, Bit 19. If Bit 19 is set high ('1'), all cacheable data load references will be forced to miss in the data cache. The data references will then be supplied using the Data Cache miss protocol (including DBlockRefill). Store operations will continue to update the cache, and the cache miss processing will update the cache. Thus, this bit provides a quick method of initializing or reloading the cache from an external device.

To maintain used line coherency, partial word stores in the ‘FDCM’ mode will continue to read data and tags directly from the data cache. At reset, Bit 19 is reset low ('0'), allowing normal operation of the data cache. Note also that this bit is logically "OR'ed" with the emulator interface "FCM" pin.

Value	Action
'0'	Normal data cache operation (default).
'1'	Force data cache operations to miss.

Table 5.8 RC36100 ForceDCacheMiss Field

ForceICacheMiss ('FICM')

Table 5.9 shows the values and actions for the ForceICacheMiss field. If this bit is set high ('1'), all cacheable instruction references will be forced to miss in the instruction cache. The instruction references will then be supplied using the Instruction Cache miss protocol (a quad word read). Cache miss processing will update the cache. Thus, this bit provides a quick method of initializing the cache or reloading the cache from an external device.

At reset, this bit is reset low ('0'), allowing normal operation of the instruction cache. Note also that this bit is logically "OR'ed" with the emulator interface "FCM" pin.

Value	Action
'0'	Normal instruction cache operation (default).
'1'	Force instruction cache operations to miss.

Table 5.9 RC36100 ForceICacheMiss Field

DCacheWriteDisable('DWrD')

Table 5.10 shows the values and actions for the Data Cache Write Disable field. According to this table, when set high ('1'), this field causes data cache writes to be ignored. The data cache will thus contain the older value, regardless of the reason for the cache miss processing. Similarly, store instructions will not cause the D-cache to be updated. When cleared low ('0'), normal cache operation results.

Value	Action
'0'	Normal data cache operation (default).
'1'	Data cache writes inhibited.

Table 5.10 RC36100 Data Cache Write Disable Field

I-CacheWriteDisable ('IWrD')

Table 5.11 shows the values and actions for the Instruction Cache Write Disable field. According to this table, when set high ('1'), this field causes instruction cache writes to be ignored. The instruction cache will thus contain the older value, regardless of the reason for the cache miss processing. When cleared low ('0'), normal cache operation results.

Notes

Value	Action
'0'	Normal data cache operation (default).
'1'	Instruction cache writes inhibited.

Table 5.11 RC36100 Instruction Cache Write Disable Field

The Cause Register

The contents of the Cause register describe the last exception. A 5-bit exception code indicates the cause of the current exception; the remaining fields contain detailed information specific to certain exceptions.

All bits in this register, with the exception of the SW bits, are read-only. The SW bits can be written to set or reset software interrupts. Figure 5.3 illustrates the format of the Cause register. Table 5.12 details the meaning of the various exception codes.

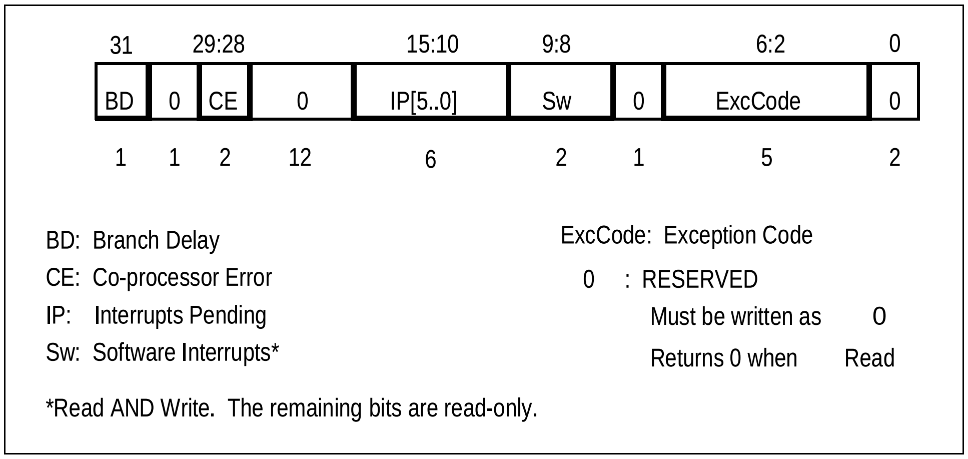


Figure 5.3 RC36100 Cause Register

Notes

Number	Mnemonic	Description
0	Int	External Interrupt
1	MOD†	TLB Modification Exception
2	TLBL†	TLB miss Exception (Load or instruction fetch)
3	TLBS†	TLB miss exception (Store)
4	AdEL	Address Error Exception (Load or instruction fetch)
5	AdES	Address Error Exception (Store)
6	IBE	Bus Error Exception (for Instruction Fetch)
7	DBE	Bus Error Exception (for data Load or Store)
8	Sys	SYSCALL Exception
9	Bp	Breakpoint Exception
10	RI	Reserved Instruction Exception
11	CpU	Co-Processor Unusable Exception
12	Ovf	Arithmetic Overflow Exception
13:31	-	Reserved
†These exceptions will not occur in RC36100.		

Table 5.12 Cause Register Exception Codes

Definitions of the other cause register bits are as follows:

BD The Branch Delay bit is set (1) if the last exception was taken while the processor was executing in the branch delay slot. If so, then the EPC will be rolled back to point to the branch instruction, so that it can be re-executed and the branch direction re-determined.

CE The Coprocessor Error field captures the coprocessor unit number referenced when a Coprocessor Unusable exception is detected.

IP The Interrupt Pending field indicates which interrupts are pending. Regardless of which interrupts are masked, the IP field can be used to determine which interrupts are pending.

SW The Software interrupt bits can be thought of as the logical extension of the IP field. The SW interrupts can be written to force an interrupt to be pending to the processor, and are useful in the prioritization of exceptions. To set a software interrupt, a "1" is written to the appropriate SW bit, and a "0" will clear the pending interrupt. There are corresponding interrupt mask bits in the status register for these interrupts.

ExcCode The exception code field indicates the reason for the last exception. Its values are listed in Table 5.12 on page 7.

The EPC (Exception Program Counter) Register

The 32-bit EPC register contains the virtual address of the instruction which took the exception, from which point processing resumes after the exception has been serviced. When the virtual address of the instruction resides in a branch delay slot, the EPC contains the virtual address of the instruction immediately preceding the exception (that is, the EPC points to the Branch or Jump instruction).

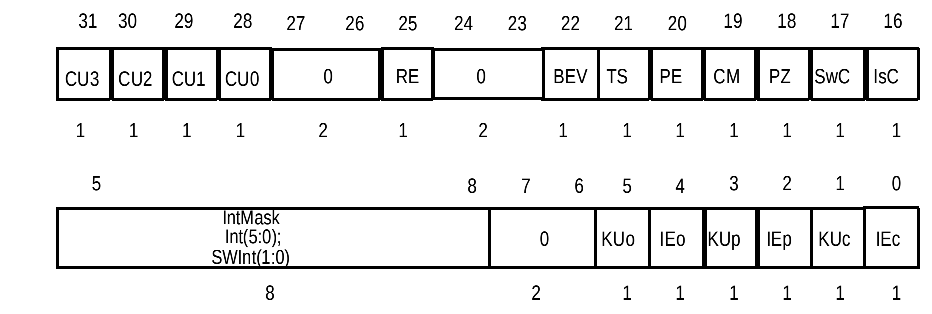
Notes

Bad VAddr Register

The Bad VAddr register saves the entire bad virtual address for any addressing exception.

The Status Register

The Status register contains all the major status bits; any exception puts the system in Kernel mode. All bits in the status register, with the exception of the TS (TLB Shutdown) bit, are readable and writable; the TS bit is read-only. Figure 5.4 on page 8 shows the functions of the various bits in the status register.



CU(n):	Co-processor 'n' Usable	IntMask:	Interrupt Mask field
'0':	Reserved: must be written as '0'	KUo:	Kernel/User mode (old)
RE:	Reverse Endian enable	IEo:	Interrupt Enable (old)
BEV:	Boot-time Exception vector	KUp:	Kernel/User mode (previous)
TS:	TLB Shutdown	IEp:	Interrupt Enable (previous)
PE:	Parity Error	KUc:	Kernel/User mode (current)
CM:	Cache Miss	IEc:	Interrupt Enable (current)
PZ:	Parity Zero		
SwC:	Swap Caches		
IsC:	Isolate Cache		

Figure 5.4 RC36100 Status Register

The status register contains a three-level stack (current, previous, and old) of the kernel/user mode bit (KU) and the interrupt enable (IE) bit. The stack is pushed when each exception is taken and popped by the Restore From Exception instruction. These bits may also be read or written to directly.

At reset, the SWc, KUc, and IEc bits are set to zero; BEV is set to one; and the value of the TS bit is set to "1". The rest of the bit fields are undefined after reset. The various bits of the status register are defined as follows:

CU Coprocessor Usability. These bits individually control user level access to coprocessor operations, including the polling of the BrCond input pins and the manipulation of the System Control Coprocessor (CP0).

RE Reverse Endianness. The RISCore3000 family architecture allows the system to determine the byte ordering convention for the Kernel mode, and the default setting for user mode, at reset time. If this bit is cleared, the endianness defined at reset is used for the current user task. If this bit is set, then the user task will operate with the opposite byte ordering convention from that determined at reset. This bit has no effect on kernel mode. Also note that the setting of this bit does not affect the byte lanes used in 16- and 8-bit memory ports; thus, external byte lane shift logic is not required.

Note: When RE = 1, set to Big Endian
When RE = 0, set to Little Endian.

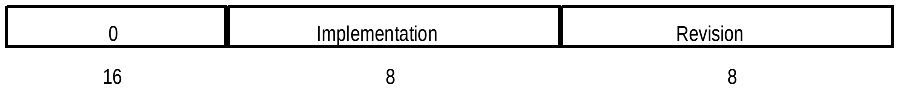
Notes

BEV	Bootstrap Exception Vector. The value of this bit determines the locations of the exception vectors of the processor. If BEV = 1, then the processor is in “Bootstrap” mode, and the exception vectors reside in uncacheable space. If BEV = 0, then the processor is in normal mode, and the exception vectors reside in cacheable space.
TS	TLB Shutdown. This bit reflects whether the TLB is functioning. At reset, this bit can be used to determine whether the current processor is a base or extended architecture version. For the RC36100, this bit is frozen at "1".
PE	Parity Error. This field should be written with a "1" at boot time. Once initialized, this field will always be read as "0'.
CM	Cache Miss. This bit is set if a cache miss occurred while the cache was isolated. It is useful in determining the size and operation of the internal cache subsystem.
PZ	Parity Zero. This field should always be written with a "0".
SwC	Swap Caches. Setting this bit causes the execution core to use the on-chip instruction cache as a data cache and vice-versa. Resetting the bit to zero un-swaps the caches. This is useful for certain operations such as instruction cache flushing. This feature is not intended for normal operation with the caches swapped.
IsC	Isolate Cache. If this bit is set, the data cache is “isolated” from main memory; that is, store operations modify the data cache but do not cause a main memory write to occur, and load operations return the data value from the cache whether or not a cache hit occurred. This bit is also useful in various operations such as flushing, as described in Chapter 3.
IntMask	Interrupt Mask. This 8-bit field can be used to mask the hardware and software interrupts to the execution engine (that is, not allow them to cause an exception). IM(1:0) are used to mask the software interrupts, and IM (7:2) mask the 6 external interrupts. A value of ‘0’ disables a particular interrupt, and a ‘1’ enables it. Note that the IE bit is a global interrupt enable; that is, if the IE is used to disable interrupts, the value of particular mask bits is irrelevant; if IE enables interrupts, then a particular interrupt is selectively masked by this field.
KUo	Kernel/User old. This is the privilege state two exceptions previously. A ‘0’ indicates kernel mode.
IEo	Interrupt Enable old. This is the global interrupt enable state two exceptions previously. A ‘1’ indicates that interrupts were enabled, subject to the IM mask.
KUp	Kernel/User previous. This is the privilege state prior to the current exception A ‘0’ indicates kernel mode.
IEp	Interrupt Enable previous. This is the global interrupt enable state prior to the current exception. A ‘1’ indicates that interrupts were enabled, subject to the IM mask.
KUc	Kernel/User current. This is the current privilege state. A ‘0’ indicates kernel mode.
IEc	Interrupt Enable current. This is the current global interrupt enable state. A ‘1’ indicates that interrupts are enabled, subject to the IM mask.
‘0’	Fields indicated as ‘0’ are reserved; they must be written as ‘0’, and will return ‘0’ when read.

Notes

PRId Register

This register is useful to software in determining which revision of the processor is executing the code. The format of this register is illustrated in Figure 5.5. For the RC36100, the value returned is 0x0000_071x. On the RC36100, the most significant 4 bits of the Revision field form an extension to the Implementation field. The least significant 4 bits of the Revision field are reserved for manufacturing. This value is different from other members of the IDT RISController family, so that software can easily determine the CPU type. This facilitates the development of one binary working with all family members.



0:	Returns '0' when Read
Implementation:	CPU Implementation number ('07' for IDT embedded)
Revision:	Revision ('10' for RC36100)

Figure 5.5 RC36100 PrID Register

Interrupt and Exception Handling

Notes

Introduction

Processors execute code in a highly-directed fashion: the instruction immediately subsequent to the current instruction is fetched and then executed. If the fetched instruction is a branch instruction, then the program's execution is diverted to the specified location. As such, program execution is relatively straightforward and predictable.

Exceptions are a mechanism used to break into the execution stream, forcing the processor into handling another task (typically related to the system state or in response to an undesirable execution of the program stream). Thus, programmers typically view exceptions as asynchronous interruptions of their program.[1]

The RISCore3000 family architecture provides for extremely fast, flexible interrupt and exception handling. The processor makes no assumptions regarding interrupt causes or handling techniques and allows the system designer to build his own model of the best response-to-exception conditions. However, the processor does provide enough information and resources to minimize the response-to-exception time and the amount of software required to preserve processor state information so that the normal instruction stream may be quickly resumed.

This chapter addresses exception handling issues that occur in RC36100-based systems. The topics examined are: the RC36100 exception model, the machine state to be saved on an exception, and nested exceptions. Representative software examples of exception handlers are also provided, as are techniques and issues appropriate to specific exception classes.

RC36100 Exception Model

The exception processing capability of the RC36100 assures an orderly transfer of control from an executing program to the kernel. Exceptions may be broadly divided into the following two categories:

- *exceptions that are caused by an instruction or instruction sequence, including an unusual condition arising during its execution.*
- *exceptions caused by external events such as interrupts.*

When an RC36100 detects an exception, the normal instruction flow sequence is suspended and the processor is forced into kernel mode where it can respond to the abnormal or asynchronous event. Table 6.1 on page 2 lists the exceptions recognized by the RISCore3000 family architecture.

1. Note that exceptions are not necessarily unpredictable or asynchronous: the events that cause the exception may be exactly repeatable by the same software executing on the same data; however, the programmer does not typically "expect" an exception to occur when and where it does and thus will view exceptions as asynchronous events.

Notes

Exception	Mnemonic	Cause
Reset	Reset	Assertion of the Reset signal causes an exception that transfers control to the special vector at virtual address 0xbfc0_0000.
UTLB Miss†	UTLB	User TLB Miss. A reference is made (in either kernel or user mode) to a page in kuseg that has no matching TLB entry. This can occur only in extended architecture versions of the processor.
TLB Miss†	TLBL (Load) TLBS (Store)	A referenced TLB entry's Valid bit isn't set, or there is a reference to a kseg2 page that has no matching TLB entry. This can occur only in extended architecture versions of the processor.
TLB Modified†	Mod	During a store instruction, the Valid bit is set but the dirty bit is not set in a matching TLB entry. This can occur only in extended architecture versions of the processor.
Bus Error	IBE (Instruction) DBE (Data)	Assertion of the Bus Error input during a read operation, due to such external events as bus time-out, backplane memory errors, invalid physical address, or invalid access types.
Address Error	AdEL (Load) AdES (Store)	Attempt to load, fetch, or store an unaligned word; that is, a word or halfword at an address not evenly divisible by four or two, respectively. Also caused by reference to a virtual address with most significant bit set while in User Mode.
Overflow	Ovf	Twos complement overflow during add or subtract.
System Call	Sys	Execution of the SYSCALL Trap Instruction
Breakpoint	Bp	Execution of the break instruction
Reserved Instruction	RI	Execution of an instruction with an undefined or reserved major operation code (bits 31:26), or a special instruction whose minor opcode (bits 5:0) is undefined.
Co-processor Unusable	CpU	Execution of a co-processor instruction when the CU (Co-processor Usable) bit is not set for the target co-processor.
Interrupt	Int	Assertion of one of the six hardware interrupt inputs or setting of one of the two software interrupt bits in the Cause register.
†These exceptions will not occur in an RC36100, or in any base member of the RISCore3000 family.		

Table 6.1 RISCore3000 Family Architecture Exceptions

Precise vs. Imprecise Exceptions

One classification of exceptions refers to the precision with which the exception cause and processor context can be determined. That is, some exceptions are precise in their nature, while others are "imprecise."

In a **precise exception**, much is known about the system state at the exact instance the exception is caused. Specifically, the exact processor context and the exact cause of the exception are known. The processor thus maintains its exact state before the exception was generated, and can accurately handle the exception, allowing the instruction stream to resume when the situation is corrected. Additionally, in a precise exception model, the processor can not advance state; that is, subsequent instructions, which may already be in the processor pipeline, are not allowed to change the state of the machine.

Many real-time applications benefit from a processor model that guarantees precise exception context and cause information. The MIPS architecture, including the RC36100, implements a precise exception model for all exceptional events.

Notes

Exception Processing

The RC36100 exception handling system efficiently handles machine exceptions, including arithmetic overflows, I/O interrupts, system calls, breakpoints, reset, and co-processor unusable conditions. Any of these events interrupt the normal execution flow; the RC36100 aborts the instruction causing the exception and also aborts all those following in the exception pipeline which have already begun, thus not modifying processor context. The CPU then performs a direct jump into a designated exception handler routine. This insures that the RC36100 is always consistent with the precise exception model.

Exception Handling Registers

The system co-processor (CP0) registers contain information pertinent to exception processing. Software can examine these registers during exception processing to determine the cause of the exception and the state of the processor when it occurred There are four registers used in exception processing, shown in Chapter 5. These are the *Cause* register, the *EPC* register, the *Status* register, and the *BadVAddr* register. A brief description of each follows.

The Cause Register

The contents of the Cause register describe the last exception. A 5-bit exception code indicates the cause of the current exception; the remaining fields contain detailed information specific to certain exceptions. All bits in this register, with the exception of the SW bits, are read-only. The SW bits can be written to set or reset software interrupts. Figure 6.1 illustrates the cause register.

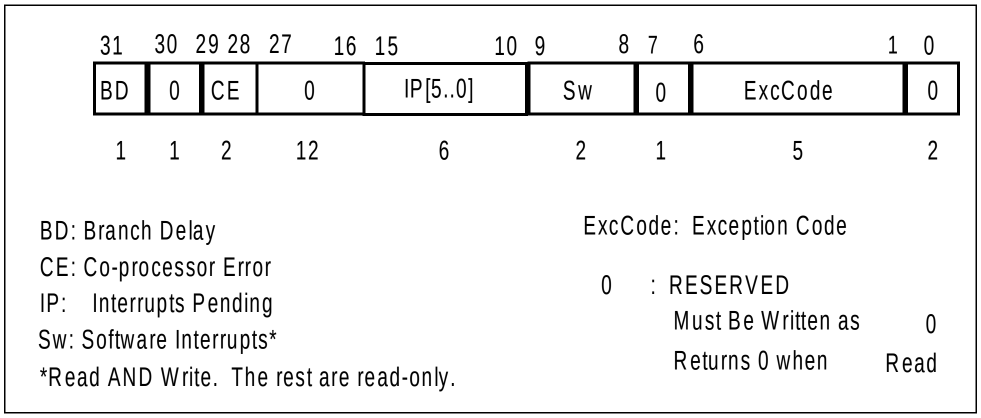


Figure 6.1 RC36100 Cause Register

The remaining bits of the cause register are defined as follows:

BD The Branch Delay bit is set (1) if the last exception was taken while the processor was executing in the branch delay slot. If so, then the EPC will be rolled back to point to the branch instruction, so that it can be re-executed and the branch direction re-determined.

CE The Co-processor Error field captures the co-processor unit number referenced when a Co-processor Unusable exception is detected.

IP The Interrupt Pending field indicates which interrupts are pending. Regardless of which interrupts are masked, the IP field can be used to determine which interrupts are pending.

SW The Software interrupt bits can be thought of as the logical extension of the IP field. The SW interrupts can be written to force an interrupt to be pending to the processor and are useful in exception prioritizing. To set a software interrupt, a “1” is written to the appropriate SW bit, and a “0” will clear the pending interrupt. There are corresponding interrupt mask bits in the status register for these interrupts.

Notes

ExcCode The exception code field indicates the reason for the last exception. Its values are listed in Table 6.2.

Number	Mnemonic	Description
0	Int	External Interrupt
1	MOD†	TLB Modification Exception
2	TLBL†	TLB miss Exception (Load or instruction fetch)
3	TLBS†	TLB miss exception (Store)
4	AdEL	Address Error Exception (Load or instruction fetch)
5	AdES	Address Error Exception (Store)
6	IBE	Bus Error Exception (for Instruction Fetch)
7	DBE	Bus Error Exception (for data Load or Store)
8	Sys	SYSCALL Exception
9	Bp	Breakpoint Exception
10	RI	Reserved Instruction Exception
11	CpU	Co-Processor Unusable Exception
12	Ovf	Arithmetic Overflow Exception
13:31	-	Reserved

†These exceptions will not occur the RC36100

Table 6.2 Cause Register Exception Codes

The EPC (Exception Program Counter) Register

The 32-bit EPC register contains the virtual address of the instruction that took the exception, from which point processing resumes after the exception has been serviced. When the virtual address of the instruction resides in a branch delay slot, the EPC contains the virtual address of the instruction immediately preceding the exception (that is, the EPC points to the Branch or Jump instruction).

Bad VAddr Register

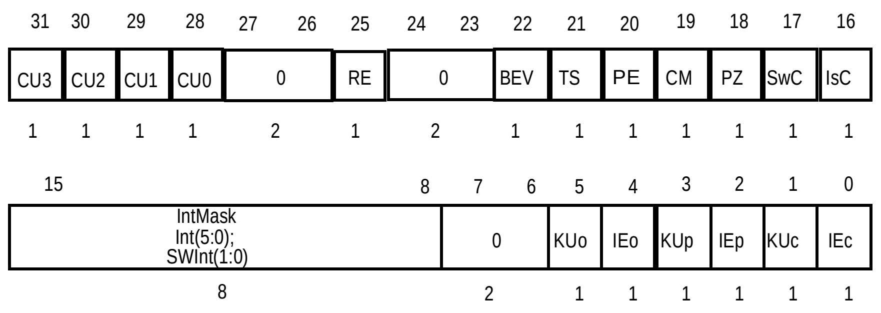
The Bad VAddr register saves the entire bad virtual address for any addressing exception.

The Status Register

The Status register contains all of the major status bits; any exception puts the system in Kernel mode. All bits in the status register, with the exception of the TS (TLB Shutdown) bit, are readable and writable; the TS bit is read-only, and frozen to '1' in the RC36100. Figure 6.2 shows the definition and position of the various bits in the status register.

The status register contains a three level stack (current, previous, and old) of the kernel/user mode bit (KU) and the interrupt enable (IE) bit. The stack is pushed when each exception is taken, and popped by the Restore From Exception instruction. These bits may also be directly read or written. At reset, the SWc, KUc, and IEc bits are set to zero; BEV is set to one; and the value of the TS bit is set to "1". The rest of the bit fields are undefined after reset.

Notes



CU(n):	Co-processor 'n' Usable	IntMask:	Interrupt Mask field
'0':	Reserved: must be written as '0'	KUo:	Kernel/User mode (old)
RE:	Reverse Endian enable	IEo:	Interrupt Enable (old)
BEV:	Boot-time Exception vector	KUp:	Kernel/User mode (previous)
TS:	TLB Shutdown	IEp:	Interrupt Enable (previous)
PE:	Parity Error	KUc:	Kernel/User mode (current)
CM:	Cache Miss	IEc:	Interrupt Enable (current)
PZ:	Parity Zero		
SwC:	Swap Caches		
IsC:	Isolate Cache		

Figure 6.2 The Status Register

The various bits of the status register are defined in chapter 5. The bits of most relevance in exception processing are repeated below.

BEV Bootstrap Exception Vector. The value of this bit determines the locations of the exception vectors of the processor. If BEV = 1, then the processor is in “Bootstrap” mode, and the exception vectors reside in uncacheable space. If BEV = 0, then the processor is in normal mode, and the exception vectors reside in cacheable space.

IM Interrupt Mask. This 8-bit field can be used to mask the hardware and software interrupts to the execution engine (that is, not allow them to cause an exception). IM(1:0) are used to mask the software interrupts, and IM (7:2) mask the 6 external interrupts. A value of ‘0’ disables a particular interrupt, and a ‘1’ enables it. Note that the IE bit is a global interrupt enable; that is, if the IE is used to disable interrupts, the value of particular mask bits is irrelevant; if IE enables interrupts, then a particular interrupt is selectively masked by this field.

KUo Kernel/User old. This is the privilege state two exceptions previously. A ‘0’ indicates kernel mode.

IEo Interrupt Enable old. This is the global interrupt enable state two exceptions previously. A ‘1’ indicates that interrupts were enabled, subject to the IM mask.

KUp Kernel/User previous. This is the privilege state prior to the current exception A ‘0’ indicates kernel mode.

IEp Interrupt Enable previous. This is the global interrupt enable state prior to the current exception. A ‘1’ indicates that interrupts were enabled, subject to the IM mask.

KUc Kernel/User current. This is the current privilege state. A ‘0’ indicates kernel mode.

IEc Interrupt Enable current. This is the current global interrupt enable state. A ‘1’ indicates that interrupts are enabled, subject to the IM mask.

Notes

Exception Vector Locations

The RISCore3000 family architecture separates exceptions into three vector spaces. The value of each vector depends on the status register's BEV (Boot Exception Vector) bit, which allows two alternate sets of vectors (and thus two different pieces of code) to be used.

Typically, this is used to allow diagnostic tests to occur before the functionality of the cache is validated; processor reset forces the value of the BEV bit to a '1'. Table 6.3 and Table 6.4 list the exception vectors of the two different modes.

Exception	Virtual Address	Physical Address
Reset	0xbfc0_0000	0x1fc0_0000
UTLB Miss	0x8000_0000	0x0000_0000
General	0x8000_0080	0x0000_0080

Table 6.3 Exception Vectors When BEV = 0

Exception	Virtual Address	Physical Address
Reset	0xbfc0_0000	0x1fc0_0000
UTLB Miss	0xbfc0_0100	0x1fc0_0100
General	0xbfc0_0180	0x1fc0_0180

Table 6.4 Exception Vectors When BEV = 1

Exception Prioritization

To understand the processor's exception priority model, it is important to know the structure of the RC36100's instruction execution unit. The RC36100 runs instructions through a five-stage pipeline, shown in Figure 6.3.

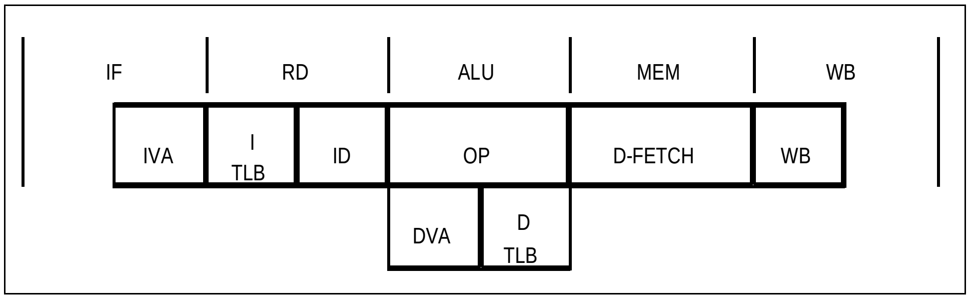


Figure 6.3 Pipelining in the RC3051 family

The pipeline stages are as follows:

- *IF (Instruction Fetch). This cycle contains two parts: the IVA (Instruction Virtual Address) phase, which generates the virtual instruction address of the next instruction to be fetched, and the ITLB phase, which performs the virtual to physical translation of the address.*
- *RD (Read and Decode). This phase obtains the required data from the internal registers and also decodes the instruction.*
- *ALU (Arithmetic Logic Unit). This phase either performs the desired arithmetic or logical operation, or generates the address for the upcoming data operation. For data operations, this phase contains both the data virtual address stage, which generates the desired virtual address, and the data TLB stage, which performs the virtual to physical translation.*
- *MEM (Memory). This phase performs the data load or store transaction.*
- *WB (Write Back). This stage updates the registers with the result data.*

Notes

High performance is achieved because five instructions are operating concurrently, each in a different stage of the pipeline. However, since multiple instructions are operating concurrently, it is possible to have multiple exceptions generated concurrently. If so, the processor must decide which exception to process, basing this decision on the stage of the pipeline that detected the exception. The processor will then flush all preceding pipeline stages to avoid altering processor context, thus implementing precise exceptions. This determines the relative priority of the exceptions.

For example, an illegal instruction exception can only be detected in the instruction decode stage of the RC36100; an Instruction Bus Error can only be determined in the I-Fetch pipe stage. Since the illegal instruction was fetched before the instruction that generated the bus error was fetched, and since it is conceivable that handling this exception might have avoided the second exception, it is important that the processor handle the illegal instruction before the bus error. Therefore, the exception detected in the latest pipeline stage has priority over exceptions detected in earlier pipeline stages. All instructions that are fetched subsequent to this (all preceding pipeline stages) are flushed to avoid altering state information, maintaining the precise exception model.

Table 6.5 lists the priority of exceptions from highest first to lowest.

Mnemonic	**Pipestage**
Reset	Any
AdEL	Memory (Load instruction)
AdES	Memory (Store instruction)
DBE	Memory (Load or store)
MOD†	ALU (Data TLB)
TLBL†	ALU (DTLB Miss)
TLBS†	ALU (DTLB Miss)
Ovf	ALU
Int	ALU
Sys	RD (Instruction Decode)
Bp	RD (Instruction Decode)
RI	RD (Instruction Decode)
CpU	RD (Instruction Decode)
TLBL†	I-Fetch (ITLB Miss)
AdEL	IVA (Instruction Virtual Address)
IBE	RD (end of I-Fetch)
†These exceptions will not occur in an RC36100, which does not include a TLB.	

Table 6.5 RC36100 Exception Priority

Exception Latency

In interrupt driven systems, a critical measurement of a processor's throughput is the interrupt "latency" of the system. Interrupt latency is a measurement of the time between the assertion of an interrupt and the initiation of software handling. Often included when discussing latency, is the amount of overhead associated with restoring context once the exception is handled, although this is typically less critical than the initial latency.

In systems where the processor is responsible for managing a number of time-critical operations in real time, it is important that the processor minimize interrupt latency. That is, it is more important that *every* interrupt be handled at a rate above some given value, rather than *occasionally* handle an interrupt at very high speed.

Notes

Factors that affect the interrupt latency of a system include the types of operations it performs (for example, systems that have long operating sequences during which interrupts can not be accepted will have long latency), how much information must be stored and restored to preserve and restore processor context, and the priority scheme of the system.

Table 6.5 illustrates what pipestage recognizes which exceptions. As mentioned above, to avoid altering state execution, all instructions less advanced are flushed from the pipeline and will be restarted when the exception handler completes.

Once the exception is recognized, the address of the appropriate exception vector will be the next instruction to be fetched. In general, the latency to the exception handler is one instruction cycle and is, at worst, the longest stall cycle in that system.

The RC36100 implements mechanisms that can help improve exception response time. Primary among these is the cache locking mechanism described in earlier chapters. System software can be easily arranged such that the exception service routines and/or critical exception data are locked into the on-chip cache. The result will be both high-speed and fully deterministic.

Interrupts Inputs in the RC36100

The organization of interrupts in an RC36100-based system is up to the system architect. Specifically, the RC36100 multiplexes various interrupt pins with PIO pins; depending on the programming of the PIO unit, the system may have six external interrupts and two BrCond input pins available for interrupt software. This section describes operation assuming all such inputs are available to system software. Later chapters describe the on-chip PIO and interrupt control units.

Interrupt Operation in the RC36100

The RC36100 family features two types of interrupt inputs: synchronized internally and non-synchronized or direct.

The $\overline{\text{ExcSInt}}(2:0)$ bus (Synchronized Interrupts) allow the system designer to connect unsynchronized interrupt sources to the processor. The processor includes special logic on these inputs to avoid meta-stable states associated with switching inputs right at the processor sampling point. Because of this logic, the synchronized interrupt sources have slightly longer latency from the ExcSInt(n) pin to the exception vector than the non-synchronized inputs. The synchronized interrupt operation is illustrated in Figure 6.4.

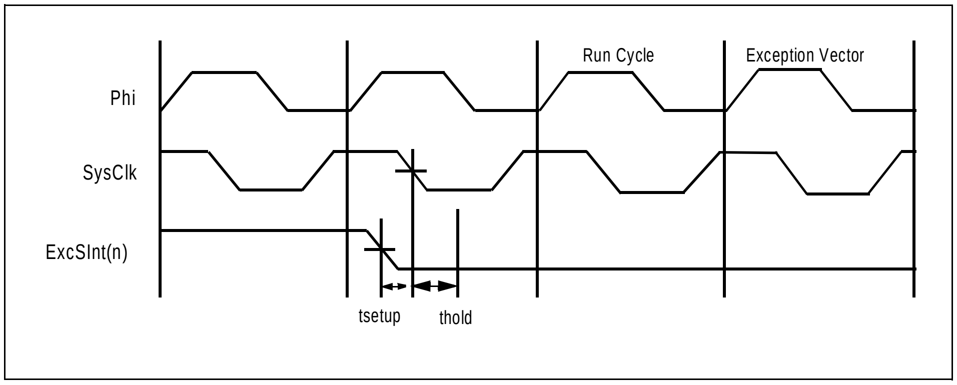


Figure 6.4 Synchronized Interrupt Operation Wave Forms

The other interrupts, $\overline{\text{Int}}(5:3)$, do not contain this synchronization logic and thus have slightly better latency to the exception vector. These inputs are useful for interrupting agents that operate asynchronously with the RC36100. However, the interrupting agent must guarantee that it will always meet the processor's interrupt input set-up and hold time requirements. The operation of these interrupts is illustrated in Figure 6.5.

Notes

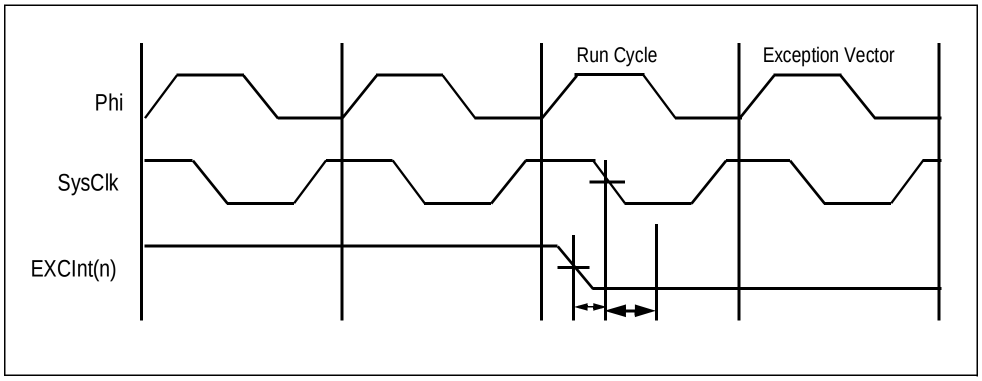


Figure 6.5 Direct Interrupt Operation Wave Forms

Because the interrupt exception is detected during the ALU stage of the current instruction, at least one run cycle must occur between (or at) the assertion of the external interrupt input and the fetch of the exception vector. Thus, if the processor is in a stall cycle when an external agent sends an interrupt, it will execute at least one run cycle before beginning exception processing. In this instance, there would be no difference in the latency of synchronized and direct interrupt inputs.

All of the interrupts are level-sensitive and active low. They continue to be sampled after an interrupt exception has occurred and are not latched within the processor when an interrupt exception occurs. Until software acknowledges the interrupt, it is important that the external interrupting agent maintain the interrupt line.

Note that the RC3081 incorporates a hardware floating point accelerator on-chip. The MIPS architecture recommends that $\overline{\text{Int}}$(3) be used to handle the floating point interrupt; thus, the RC3081 defaults to this interrupt assignment. However, the RC3081 Config register (which differs from the RC36100 Config register) can be used to change the assignment. Also, the on-chip interrupt controller of the RC36100 will signal its interrupt to the CPU using one of the available CPU interrupts. The interrupt controller uses $\overline{\text{Int}}$(5) for this operation.

Each of the eight interrupts (6 hardware and 2 software) can be individually masked by clearing the corresponding bit in the Interrupt Mask field of the Status Register. All eight interrupts can be masked at once by clearing the IEc bit in the Status Register.

On the synchronized interrupts, care should be taken to allow at least two clock cycles between the negation of the interrupt input and the re-enabling of the interrupt mask for that bit. In general, it is recommended that software continue polling the IP field of the Cause register once it has instructed the peripheral to negate its interrupt, prior to re-enabling its mask, to avoid a spurious interrupt.

The value shown in the interrupt pending bits of the Cause register reflects the current state of the interrupt pins of the processor. These bits are not latched (except for sampling from the data bus to guarantee that they are stable when examined), and the masking of specific interrupt inputs does not mask the bits from being read.

Using the BrCond Inputs

In addition to the interrupt pins themselves, many systems can use the BrCond(3:2) input port pins in their exception model. These pins can be directly tested by software, and can be used for polling or fast interrupt decoding. The kernel must enable the use of the corresponding co-processor unit before testing the state of the BrCond input pin.

The RC36100 provides two synchronized BrCond inputs: SBrCond(3:2). Note that BrCond(0), corresponding to the on-chip CP0, and BrCond(1), corresponding to Co-Processor 1 (the FPA, present on the RC3081), are not available on the RC36100 as user inputs. Instructions that use BrCond(1:0) will always see a '1' on the RC36100. Also note that the SBrCond(3:2) on the RC36100 may or may not be enabled in the PIO unit; if not, then the SBrCond(3:2) input values are undefined. When programmed to be SBrCond(3:2) inputs, the timing requirements of the SBrCond inputs are illustrated in Figure 6.6. Since these inputs are synchronized by the RC36100, they do not need to be driven synchronously to the processor.

Notes

Similar to the interrupt inputs, at least one instruction must be executed (in the ALU stage) of the instruction pipeline prior to software being able to detect a change in one of these inputs. This is because the processor actually captures the value of these flags one instruction prior to the branch on co-processor instruction. Before executing a Branch Condition instruction (i.e. BCzT, BCzF) the corresponding co-processor usable bit in the CP0 status register must be set; otherwise, a co-processor unusable exception will be signalled.

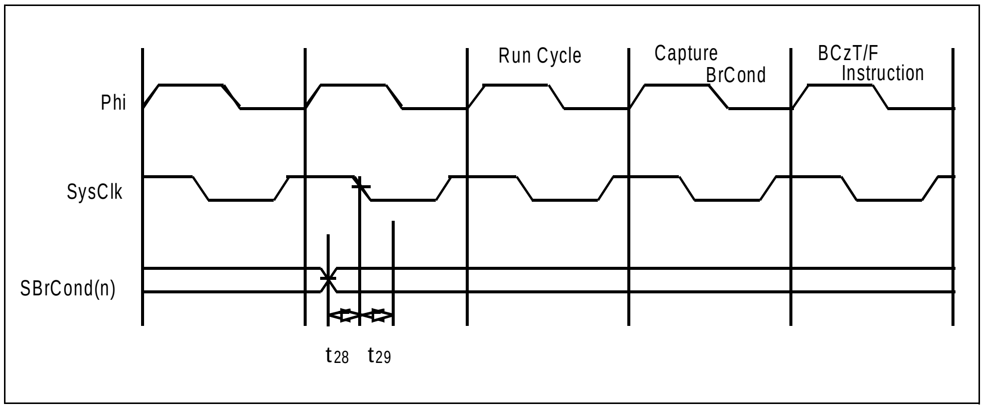


Figure 6.6 Synchronized BrCond Inputs

Interrupt Handling

The assertion of an unmasked interrupt input causes the RC36100 to branch to the general exception vector at virtual address 0x8000_0080, and write the 'Int' code in the Cause register. The IP field of the Cause register shows which of the six hardware interrupts are pending and the SW field in the Cause register show which of the two software interrupts are pending. Multiple interrupts can be pending at the same time, with no priority assumed by the processor.

If the interrupt asserted is due to the on-chip interrupt controller, the interrupt controller must be accessed to determine which of its interrupt sources caused the assertion. This operation is described in a later chapter.

When an interrupt occurs, the KUp, IEp, KUc and IEc bits of the Status register are saved in the KUo, IEo, KUp, IEp bit fields in the Status register, respectively, as illustrated in Figure 6.7. The current kernel status bit KUc and the interrupt bit IEc are cleared. This will mask all of the interrupts and then place the processor in kernel mode. This sequence will be reversed by the execution of an *rfe* (restore from exception) instruction, typically in the branch delay slot of the branch which resumes normal execution.

Notes

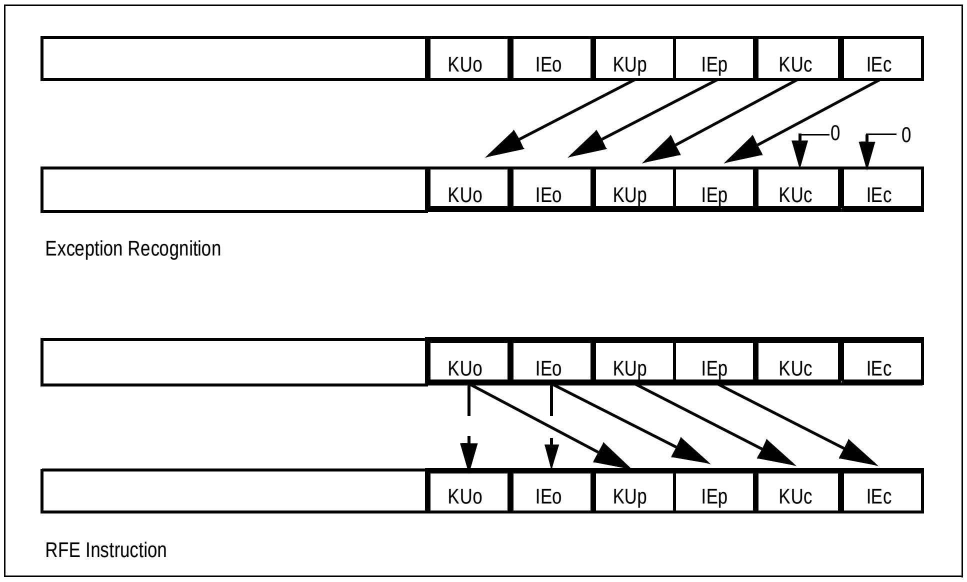


Figure 6.7 Kernel and Interrupt Status Being Saved on Interrupts

Interrupt Servicing

In case of a hardware interrupt, the interrupt must be cleared by de-asserting the interrupt line, which has to be done by alleviating the external conditions that caused the interrupt. Software interrupts have to be cleared by clearing the corresponding bits, SW(1:0), in the Cause register to zero. It is recommended that software continue polling the IP field of the Cause register once it has instruted the peripheral to negate its interrupt, prior to re-enabling its mask, to avoid a spurious interrupt.

Basic Software Techniques For Handling Interrupts

Once an exception is detected the processor suspends the current task, enters kernel mode, disables interrupts, and begins processing at the exception vector location. The EPC is loaded with the address the processor will return to once the exception event is handled.

The specific actions of the processor depend on the cause of the exception being handled. The MIPS architecture classifies exceptions into three distinct classes: RESET, UTLB Miss , and General.

Coming out of reset, the processor initializes the state of the machine. In addition to initializing system peripherals, page tables, the TLB, and the caches, software clears both STATUS and CAUSE registers, and initializes the exception vectors.

The code located at the exception vector may be just a branch to the actual exception code; however, in more time critical systems the instructions located at the exception vector may perform the actual exception processing. In order to cause the exception vector location to branch to the appropriate exception handler (presuming that such a jump is appropriate), a short code sequence such as that illustrated in Figure 6.8 may be used.

It should be noted the contents of register k0 are not preserved. This is not a problem for software, since MIPS compiler and assembler conventions reserve k0 (and often k1) for kernel processes, and do not use it for user programs. For the system developer it is advised that the use of k0 be reserved for use by the exception handling code exclusively. This will make debugging and development much easier.

The "IDT RISCore3000 Family Software Reference Manual" provides a great deal of information on the software requirements of exception management, including interrupt service.

Notes

```
        .set    noreorder              # tells the assembler not to reorder the code
/*
**      code sequence copied to UTLB exception vector
*/
        la      k0,excep_utlb          #address of utlb excp. handler
        j       k0                     # jump via reg k0
        nop
/*
**      code sequence copied to general exception vector
*/
        la      k0,excep_general       #address of general excp. handler
        j       k0                     # jump via reg k0
        nop
```

Figure 6.8 Code Sequence to Initialize Exception Vectors

Preserving Context

The RC36100 has the following four registers related to exception processing:

- *The Cause register*
- *The EPC (exception program counter) register*
- *The Status register*
- *The BadVAddr (bad virtual address) register*

Typical exception handlers preserve the status, cause, and EPC registers in general registers (or on the system stack). If the exception cause is due to an address error, software may also preserve the bad virtual address register for later processing.

Note that not all systems need to preserve this information. Since the RC36100 disables subsequent interrupts, it is possible for software to directly process the exception while leaving the processor context in the CP0 registers. Care must be taken to insure that the execution of the exception handler does not generate subsequent exceptions.

Preserving the context in general registers (and on the stack) does have the advantage that interrupts can be re-enabled while the original exception is handled, thus allowing a priority interrupt model to be built.

A typical code sequence to preserve processor context is shown in Figure 6.9. This code sequence preserves the context into an area of memory pointed to by the k0 kernel register. This register points to a block of memory capable of storing processor context. Constants identified by name (such as *R_EPC*) are used to indicate the offset of a particular register from the start of that memory area.

It should be noted that this sequence for fetching the co-processor zero registers is required because there is a one clock delay in the register value actually being loaded into the general registers after the execution of the mfc0 instruction.

Notes

```
        la      k0,except_regs      # fetch address of reg save array
        sw      AT,R_AT*4(k0)       # save register AT
        sw      v0,R_V0*4(k0)       # save register v0
        sw      v1,R_V1*4(k0)       # save register v1
        mfc0    v0,C0_EPC           # fetch the epc register
        mfc0    v1,C0_SR            # fetch the status register
        sw      v0,R_EPC*4(k0)      # save the epc
        mfc0    v0,C0_CAUSE         # fetch the cause register
        sw      v1,R_SR*4(k0)       # save status register

/*      The above code is about the minimum required
**      The user specific code would follow
*/
```

Figure 6.9 Preserving Processor Context

Examining Exceptions

The cause register indicates the reason the exception handler was invoked. Thus, to invoke the appropriate exception service routine, software merely needs to examine the cause register, and use its contents to direct a branch to the appropriate handler.

One method of decoding the jump to an appropriate software routine to handle the exception and cause is shown in Figure 6.10. Register v0 contains the cause register, and register k0 still points to the register save array.

```
.set    noreorder
sw      a0,R_A0*4(k0)       # save register a0
and     v1,v0,EXCMASK       # isolate exception code
lw      a0,cause_table(v1)  # get address of interrupt routine.
sw      a1,R_A1*4(k0)       # use delay slot to save register a1
j       a0
sw      k1,R_K1*4(sp)       # save k1 register
.set    reorder             # re-enable pipeline scheduling
```

Figure 6.10 Exception Cause Decoding

The above sequence of instructions extracts the exception code from the cause register and uses that code to index into the table of pointers to functions (the cause_table). The *cause_table* data structure is shown in Figure 6.11.

Each of the entries in this table point to a function for processing the particular type of interrupt detected. The specifics of the code contained in each function is unique to a given application; all registers used in these functions must be saved and restored.

Notes

```
int (*cause_table[16])()  = {
                int_extern,          /* External interrupts              */
                int_tlbmod,          /* TLB modification error           */
                int_tlbmiss,         /* load or instruction fetch        */
                int_tlbmiss,         /* write miss                       */
                int_addrerr,         /* load or instruction fetch        */
                int_addrerr,         /* write address error              */
                int_ibe,             /* Bus error - Instruction fetch    */
                int_dbe,             /* Bus error - load or store data   */
                int_syscall,         /* SYSCALL exception                */
                int_breakpoint,      /* breakpoint instruction           */
                int_trap,            /* Reserved instruction             */
                int_cpunuse,         /* coprocessor unusable             */
                int_trap,            /* Arithmetic overflow              */
                int_unexp,           /* Reserved                         */
                int_unexp,           /* Reserved                         */
                int_unexp            /* Reserved                         */
                };
```

Figure 6.11 Exception Service Branch Table

Returning From Exceptions

Returning from the exception routine is made through the *rfe* instruction. When the exception first occurs the RC36100 automatically saves some of the processor context, the current value of the interrupt enable bit is saved into the field for the previous interrupt enable bit, and the kernel/user mode context is preserved.

The *IE* interrupt enable bit must be asserted (a one) for external interrupts to be recognized. The *KU* kernel mode bit must be a zero in kernel mode. When an exception occurs, external interrupts are disabled and the processor is forced into kernel mode. When the *rfe* instruction is executed at completion of exception handling, the state of the mode bits is restored to what it was when the exception was recognized (presuming the programmer restored the status register to its value when the exception occurred). This is done by “popping” the old/previous/current KU and IE bits of the status register.

The code sequence in Figure 6.12 is an example of exiting an interrupt handler. The assumption is that registers and context were saved as outlined above. To properly exit from exception handling, this code sequence must either be replicated in each of the cause handling functions or each of them must branch to this code sequence.

Note that this code sequence must be executed with interrupts disabled. If the exception handler routine re-enables interrupts, they must be disabled when the CP0 registers are being restored.

Notes

```
gen_excp_exit:
        .set    noreorder
                                    # by the time we have gotten here
                                    # all general registers have been
                                    # restored (except of k0 and v0)
                                    # reg. AT points to the reg save array
        lw      k0,C0_SR*4(AT)      # fetch status reg. contents
        lw      v0,R_V0*4(AT)       # restore reg. v0
        mtc0    k0,C0_SR            # restore the status reg. contents
        lw      k0,R_EPC*4(AT)      # Get the return address
        lw      AT,R_AT*4(AT)       # restore AT in load delay
        j       k0                  # return from int. via jump reg.
        rfe                         # the rfe instr. is executed in the
                                    # branch delay slot
        .set    reorder
```

Figure 6.12 Returning from Exception

Special Techniques

There are a number of techniques which take advantage of the RC36100 architecture to minimize exception latency and maximize throughput in interrupt driven systems. This section discusses a number of those techniques.

Interrupt Masking

Only the six external and two software interrupts are maskable exceptions. The mask for these interrupts are in the status register.

To enable a given external interrupt, the corresponding bit in the status register must be set. The IEc bit in the status register must also be set. It follows that by setting and clearing these bits within the interrupt handler that interrupt priorities can be established. The general mechanism for doing this is performed within the external interrupt-handler portion of the exception handler.

The interrupt handler preserves the current mask value when the status register is preserved. The interrupt handler then calculates which (if any) external interrupts have priority, and sets the interrupt mask bit field of the status register accordingly. Once this is done, the IEc bit is changed to allow higher priority interrupts. Note that all interrupts must again be disabled when the return from exception is processed.

Using BrCond For Fast Response

The RC36100 instruction set contains mechanisms to allow external or internal co-processors to operate as an extension of the main CPU. Some of these features may also be used in an interrupt-driven system to provide the highest levels of response.

Specifically, the RC36100 allows external input port signals, the SBrCond(3:2) signals. These signals are used by external agents to report status back to the processor. The instruction set contains instructions which allow the external bits to be tested, and branches to be executed depending on the value of the SBrCond input.

An interrupt-driven system can use these SBrCond signals, and the corresponding instructions, to implement an input port for time-critical interrupts. Rather than mapping an input port in memory (which requires external logic), the SBrCond signals can be examined by software to control interrupt handling.

Notes

There are actually two techniques to use this advantageously. One method uses these signals to perform interrupt polling; in this method, the processor continually examines these signals, waiting for an appropriate value before handling the interrupt. A sample code sequence is shown in Figure 6.13.

The software in this system is very compact, and easily resides in the on-chip cache of the processor. Thus, the latency to the interrupt service routine in this system is minimized, allowing the fastest interrupt service capabilities.

A second method utilizes external interrupts combined with the SBrCond signals. In this method, both the SBrCond signal and one of the external interrupt lines are asserted when an external event occurs. This configuration allows the CPU to perform normal tasks while waiting for the external event.

For example, assume that a valve must be closed and then normal processing continued when SBrCond(2) is asserted TRUE. The valve is controlled by a register that is memory-mapped to address 0xaffe_0020 and writing a one to this location closes the valve. The software in Figure 6.14 accomplishes this, using SBrCond(2) to aid in cause decoding.

The number of cycles for a deterministic system is five cycles between the time the interrupt occurred and it was serviced. Interrupts were re-enabled in four additional cycles. Note that none of the processor context needs to be preserved and restored for this routine.

```
            .set    noreorder           # prevents the assembler from
                                        # reordering the code below

polling_loop:                           # branch to yourself until
            bc2f    polling_loop        # BrCond(2) is asserted
            nop
                                        # Once BrCond(2) is asserted, fall through
                                        # and begin processing the external event
fast_response_cp2:
                                        # code sequence that would do the
                                        # event processing

            b       polling_loop        # return to polling
```

Figure 6.13 Polling System Using BrCond

Notes

```
        .set    noreorder               # prevents the assembler from reordering
                                        # the code sequences below
/* This section of code is placed at the general exception
** vector location 0x8000_0080. When an external interrupt is
** asserted execution begins here.


*/


        bc2t    close_valve             # test for emergency condition and
        li      k0,1                    # jump to close valve if TRUE
        la      k0,gen_exp_hand         # otherwise,
        j       k0                      # jump to general exc. handler
        nop                             # and process less critical excepts.


/* This is the close valve routine - its sole purpose is to close the
** valve as quickly as possible. The registers'k0' and'k1' are reserved
** for kernel use and therefore need not be saved when a client or
** user program is interrupted. It should be noted that the value to
** write to the valve close register was put in reg'k0' in the
** branch delay slot above - so by the time we get here it is
** ready to output to the close register.


*/
close_valve:
        la      k1,0xaffe0020           # the address of the close register
        sw      k0,0(k1)                # write the value to the close register
        mfc0    k0,C0_EPC               # get the return address to cont processing
        nop
        j       k0                      # return to normal processing
        rfe                             # restore previous interrupt mask
                                        # and kernel/user mode bits of the
                                        # status register.
        .set    reorder
```

Figure 6.14 Using BrCond for Fast Interrupt Decoding

Cache Locking

The RC36100 allows the cache to be split into multiple sections, each servicing a different section of the processor address space. Using this technique, the system can "dedicate" one such section to exception service. Since the portion of the cache which deals with exception service will not be disturbed by normal system operation, the exception service code can be effectively locked into the on-chip cache. This has two positive benefits.

First, this insures that the exception service routine will operate directly out of the on-chip cache, and avoid main memory I-cache miss fetches. This speeds overall execution.

Secondly, this insures that the exception service routine performance will not be dependent on the tasks run since it was last invoked. Since those tasks will not displace the exception software from the on-chip cache, the exception software performance will be deterministic.

Notes

Nested Interrupts

Note that the processor does not automatically stack processor context when an exception occurs; thus, to allow nested exceptions it is important that software perform this stacking.

Most of the software illustrated above also applies to a nested exception system. However, rather than using just one register (pointed to by k0) as a save area, a stacking area must be implemented and managed by software. Also, since interrupts are automatically disabled once an exception is detected, the interrupt handling routine must mask the interrupt it is currently servicing, and re-enable other interrupts (once context is preserved) through the IEc bit.

The use of Interrupt Mask bits of the status register to implement an interrupt prioritization scheme was discussed earlier. An analogous technique can be performed by using an external interrupt encoder to allow more interrupt sources to be presented to the processor.

Software interrupts can also be used as part of the prioritization of interrupts. If the interrupt service routine desires to service the interrupting agent, but not completely perform the interrupt service, it can cause the external agent to negate the interrupt input but leave interrupt service pending through the use of the SW bits of the Cause register.

Catastrophic Exceptions

There are certain types of exceptions that indicate fundamental problems with the system. Although there is little the software can do to handle such events, they are worth discussing. Exceptions such as these are typically associated with faulty systems, such as in the initial debugging or development of the system.

Potential problems can arise because the processor does not automatically stack context information when an exception is detected. If the processor context has not been preserved when another exception is recognized, the value of the status, cause, and EPC registers are lost and thus the original task can not be resumed.

An example of this occurring is an exception handler performing a memory reference that results in a bus error (for example, when attempting to preserve context). The bus error forces execution to the exception vector location, overwriting the status, cause, and context registers. Proper operation cannot be resumed.

Handling Specific Exceptions

This section documents some specific issues and techniques for handling particular RC36100 exceptions.

Address Error Exception

Cause

This exception occurs when an attempt is made to load, fetch, or store a word that is not aligned on a word boundary. Attempting to load or store a half-word that is not aligned on a half-word boundary will also cause this exception. The exception also occurs in User mode if a reference is made to a virtual address whose most significant bit is set (a kernel address). This exception is not maskable.

Handling

The RC36100 branches to the General Exception vector for this exception. When the exception occurs, the CPU sets the ADEL or ADES code in the Cause register ExcCode field to indicate whether the address error occurred during an instruction fetch or a load operation (ADEL) or a store operation (ADES).

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register.

The RC36100 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively and clears the KUc and IEc bits.

Notes

When this exception occurs, the BadVAddr register contains the virtual address that was not properly aligned or that improperly addressed kernel data while in User mode. The contents of the VPN field of the Context and EntryHi registers are undefined.

Servicing

A kernel should hand the executing process a segmentation violation signal. Such an error is usually fatal; although, an alignment error might be handled by simulating the instruction that caused the error.

Breakpoint Exception

Cause

This exception occurs when the RC36100 executes the BREAK instruction and is not maskable.

Handling

The RC36100 branches to the General Exception vector for the exception and sets the BP code in the CAUSE register ExcCode field.

The RC36100 saves the *KUp, IEp, KUc,* and *IEc* bits of the Status register in the *KUo, KUp,* and *IEp* bits, respectively, and clears the *KUc* and *IEc* bits.

The *EPC* register points at the BREAK instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the *EPC* register points at the BRANCH instruction that preceded the BREAK instruction and sets the *BD* bit of the Cause register.

Service

The breakpoint exception is typically handled by a dedicated system routine. Unused bits of the BREAK instruction (bits 25..6) can be used pass additional information. To examine these bits, load the contents of the instruction pointed at by the *EPC* register.

Note: If the instruction resides in the branch delay slot, add four to the contents of the *EPC* register to find the instruction.

To resume execution, change the *EPC* register so that the RC36100 does not execute the BREAK instruction again. To do this, add four to the *EPC* register before returning.

Note: If a BREAK instruction is in the branch delay slot, the BRANCH instruction must be interpreted in order to resume execution.

Bus Error Exception

Cause

This exception occurs when the Bus Error input to the CPU is asserted by external logic during a read operation. For example, events like bus time-outs, backplane bus parity errors, and invalid physical memory addresses or access types can signal exception. This exception is not maskable.

This exception is used for synchronously occurring events such as cache miss refills. The general interrupt mechanism must be used to report a bus error that results from asynchronous events such as a buffered write transaction.

Handling

The RC36100 branches to the General Exception vector for this exception. When exception occurs, the RC36100 sets the *IBE* or *DBE* code in the CAUSE register ExcCode field to indicate whether the error occurred during an instruction fetch reference (IBE) or during a data load or store reference (DBE).

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the BRANCH instruction that preceded the exception-causing instruction and sets the BD bit of the cause register.

The RC36100 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Notes

Servicing

The physical address where the fault occurred can be computed from the information in the CP0 registers:

- *If the Cause register's IBE code is set (showing an instruction fetch reference), the virtual address resides in the EPC register.*
- *If the Cause register's DBE exception code is set (specifying a load or store reference), the instruction that caused the exception is at the virtual address contained in the EPC register (if the BD bit of the cause register is set, add four to the contents of the EPC register). Interpret the instruction to get the virtual address of the load or store reference and then use the TLBProbe (tlbp) instruction and read EntryLo to compute the physical page number.*

A kernel should hand the executing process a bus error when this exception occurs. Such an error is usually fatal.

Co-processor Unusable Exception

Cause

This exception occurs due to an attempt to execute a co-processor instruction when the corresponding co-processor unit has not been marked usable (the appropriate CU bit in the status register has not been set). For CP0 instructions, this exception occurs when the unit has not been marked usable and the process is executing in User mode: CP0 is always usable from Kernel mode regardless of the setting of the CP0 bit in the status register. This exception is not maskable.

Handling

The RC36100 branches to the General Exception vector for this exception. It sets the CPU code in the CAUSE register ExcCode field. Only one co-processor can fail at a time.

The contents of the cause register's CE (Co-processor Error) field show which of the four co-processors (3,2,1, or 0) the RC36100 referenced when the exception occurred.

The EPC register points at the co-processor instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the co-processor instruction and sets the BD bit of the Cause register.

The RC36100 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Servicing

To identify the co-processor unit that was referenced, examine the contents of the Cause register's CE field. If the process is entitled to access, mark the co-processor usable and restore the corresponding user state to the co-processor.

If the process is entitled to access to the co-processor, but the co-processor is known not to exist or to have failed, the system could interpret the co-processor instruction. If the BD bit is set in the Cause register, the BRANCH instruction must be interpreted; then, the co-processor instruction could be emulated with the EPC register advanced past the co-processor instruction.

If the process is not entitled to access to the co-processor, the process executing at the time should be handed an illegal instruction/privileged instruction fault signal. Such an error is usually fatal.

Interrupt Exception

Cause

This exception occurs when one of eight interrupt conditions (software generates two, hardware generates six) occurs.

Each of the eight external interrupts can be individually masked by clearing the corresponding bit in the IntMask field of the status register. All eight of the interrupts can be masked at once by clearing the IEc bit in the status register.

Notes

Handling

The RC36100 branches to the General Exception vector for this exception. The RC36100 sets the INT code in the Cause register's ExcCode field.

The IP field in the Cause register show which of six external interrupts are pending, and the SW field in the cause register shows which two software interrupts are pending. More than one interrupt can be pending at a time.

The RC36100 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Servicing

If software generates the interrupt, clear the interrupt condition by setting the corresponding Cause register bit (SW1:0) to zero.

If external hardware generated the interrupt, clear the interrupt condition by alleviating the conditions that assert the interrupt signal.

Overflow Exception

Cause

This exception occurs when an **ADD ADDI, SUB,** or **SUBI** instruction results in two's complement overflow. This exception is not maskable.

Handling

The RC36100 branches to the General Exception vector for this exception. The RC36100 sets the OV code in the CAUSE register.

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the Branch instruction that preceded the exception-causing instruction and sets the BD bit of the CAUSE register.

The RC36100 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Servicing

A kernel should hand the executing process a floating point exception or integer overflow error when this exception occurs. Such an error is usually fatal.

Reserved Instruction Exception

Cause

This exception occurs when the RC36100 executes an instruction whose major opcode (bits 31..26) is undefined or a Special instruction whose minor opcode (bits 5..0) is undefined.

This exception provides a way to interpret instructions that might be added to or removed from the MIPS processor architecture.

Handling

The RC36100 branches to the General Exception vector for this exception. It sets the RI code of the Cause register's ExcCode field.

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the Branch instruction that preceded the reserved instruction and sets the BD bit of the CAUSE register.

The RC36100 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Servicing

If instruction interpretation is not implemented, the kernel should hand the executing process an illegal instruction/reserved operand fault signal. Such an error is usually fatal.

Notes

An operating system can interpret the undefined instruction and pass control to a routine that implements the instruction in software. If the undefined instruction is in the branch delay slot, the routine that implements the instruction is responsible for simulating the branch instruction after the undefined instruction has been "executed". Simulation of the branch instruction includes determining if the conditions of the branch were met and transferring control to the branch target address (if required) or to the instruction following the delay slot if the branch is not taken. If the branch is not taken, the next instruction's address is [EPC] + 8. If the branch is taken, the branch target address is calculated as [EPC] + 4 + (Branch Offset * 4).

Note that the target address is relative to the address of the instruction in the delay slot, not the address of the branch instruction. For details on how branch target addresses are calculated, refer to the description of branch instruction.

Reset Exception

Cause

This exception occurs when the RC36100 RESET signal is asserted and then de-asserted.

Handling

The RC36100 provides a special exception vector for this exception. The Reset vector resides in the RC36100's un-mapped and un-cached address space; Therefore the hardware need not initialize the Translation Lookaside Buffer (TLB) or the cache to handle this exception. The processor can fetch and execute instructions while the caches and virtual memory are in an undefined state.

The contents of all registers in the RC36100 are undefined when this exception occurs except for the following:

- *The SWc, KUc, and IEc bits of the Status register are cleared to zero.*
- *The BEV bit of the Status register is set to one.*
- *The TS bit of the Status register is frozen at one.*
- *The Config register is unlocked and initialized as described in Chapter 5.*

Servicing

The reset exception is serviced by initializing all processor registers, co-processor registers, caches, and the memory system. Typically, diagnostics would then be executed and the operating system bootstrapped, including setting of the PortSize, Config, and BusCtrl registers. The reset exception vector is selected to appear in the uncached, un-mapped memory space of the machine so that instructions can be fetched and executed while the cache and virtual memory system are still in an undefined state.

System Call Exception

Cause

This exception occurs when the RC36100 executes a SYSCALL instruction.

Handling

The RC36100 branches to the General Exception vector for this exception and sets the SYS code in the CAUSE register's ExcCode field.

The EPC register points at the SYSCALL instruction that caused the exception, unless the SYSCALL instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the SYSCALL instruction and the BD bit of the CAUSE register is set.

The RC36100 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

Notes

Servicing

The operating system transfers control to the applicable system routine. To resume execution, alter the EPC register so that the SYSCALL instruction does not execute again. To do this, add four to the EPC register before returning.

Note: If aSYSCALL instruction is in a branch delay slot, the branch instruction must be interpreted in order to resume execution.

Notes

System Bus Interface Unit Overview

Notes

Introduction

The IDT79RC36100 RISController is an integration of system memory controllers and peripherals around a RISCore3000 family core. Thus, the system interface can be described at many levels:

- *Operation of the execution core, including caches and write buffers.*
- *Operation of the various memory controllers during external transactions.*
- *Operation during internal peripheral transactions.*

This chapter includes an overview on these interfaces, a complete description of the rules for the internal core, and an operational overview and detailed timing diagrams of the write interface. Details on each of the internal memory controllers and peripherals are provided in subsequent chapters.

Bus Interface Overview

The RC36100 RISController bus interface uses separate de-multiplexed address and data buses, along with control signals which select the targeted memory resource and perform the necessary data path steering. Figure 7.1 is a conceptual representation of the RC36100 bus interface.

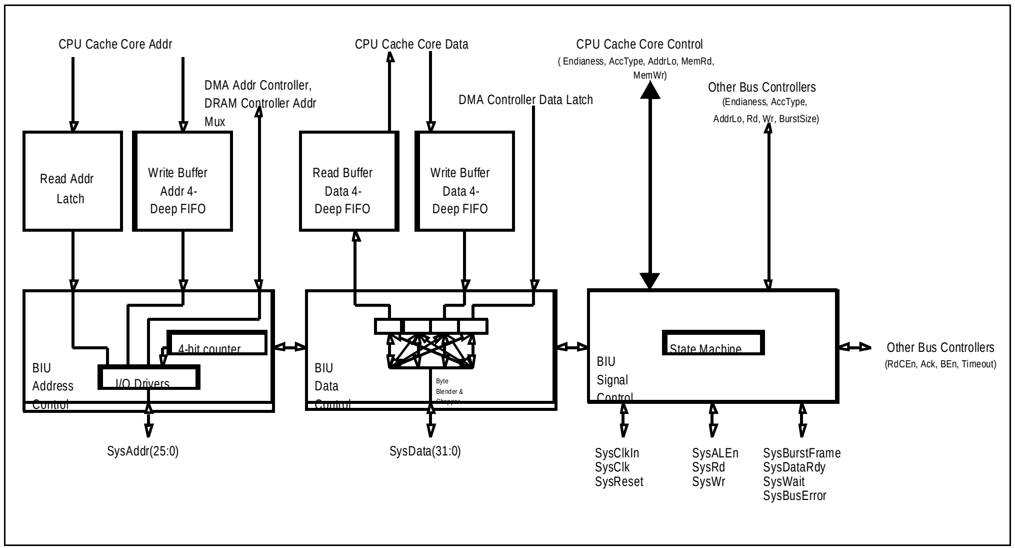

CPU Cache Core Addr
DMA Addr Controller, DRAM Controller Addr Mux
CPU Cache Core Data
DMA Controller Data Latch
CPU Cache Core Control (Endianess, AccType, AddrLo, MemRd, MemWr)
Other Bus Controllers (Endianess, AccType, AddrLo, Rd, Wr, BurstSize)
Read Addr Latch
Write Buffer Addr 4-Deep FIFO
Read Buffer Data 4-Deep FIFO
Write Buffer Data 4-Deep FIFO
4-bit counter
I/O Drivers
BIU Address Control
BIU Data Control
Byte Blender & Chopper
State Machine
BIU Signal Control
Other Bus Controllers (RdCEn, Ack, BEn, Timeout)
SysAddr(25:0)
SysData(31:0)
SysClkIn SysClk SysReset
SysALEn SysRd SysWr
SysBurstFrame SysDataRdy SysWait SysBusError

Figure 7.1 RC36100 Bus Interface Unit Block Diagram

The address bus of the RC36100 is a 26-bit address bus. Although the address bus is only 26 bits wide, the address space of the RC36100 is actually 32 bits wide; the internal address decoder provides "Chip Selects" to target particular memory subsystems; thus, the width of the address bus only limits the size of any one memory sub-system, not the overall addressable memory.

The data bus is a 32-bit wide bus, with the capability to gather or parcel out data into smaller pieces when working with 8- or 16-bit data ports. Thus, the RC36100 can mate directly with 8-, 16- and/or 32-bit memory subsystems; in fact, the widths of the various sub-systems are independently programmable through the various control registers. In addition, the RC36100 can be used to implement either Big- or Little-endian memory systems, as selected at reset.

The control signals provided by the RC36100 enable the processor to directly connect with a wide variety of external memory devices and peripherals. Wait-state generation and address decode is performed internally by the processor; once the proper external device for a transfer is determined by the RC36100, the output control signals implement the protocol and timing selected for that memory sub-space.

Notes

During accesses to DRAM, the address bus will first carry the row address, then the column address, in a fashion suitable for direct connection with external page mode DRAM devices. Additional control signals provide RAS, CAS, and transceiver control. Thus, the RC36100 does not require external address multiplexors or complicated DRAM control state machines.

These techniques enable the RC36100 to simply implement a wide variety of low cost systems, minimizing both system cost and development time, while maintaining high system throughput.

Pin Description

This section describes the signals used in the basic bus interface. Detailed information on the behavior of these signals, and of other signals used in other peripheral or memory control subsystems, are described in later chapters.

Note: Many RC36100 signals have multiple functions, the exact behavior of a given pin is typically selected at device reset, and that signals indicated with an overbar (for example, overbar) are active low.

System Bus Interface Signals

These signals are used by the bus interface to generate and provide global read and write signals.

SysAddr(25:0) **Output/(Input during external DMA)**

System Address Bus: The RC36100 uses a dedicated 26-bit physical address bus always driven by the RC36100, except during the Address Strobe portion of external DMA cycles.

The address first becomes valid on the same (first) clock cycle that the address latch enable indicator, SysALEn, asserts. Coincidently either SysRd or SysWr will also assert on the first valid cycle. The address is valid until either SysRd or SysWr de-asserts at the end of a transaction.

During Memory or I/O transactions, SysAddr(25:0) contain the internally latched 26-bit physical address. The SysAddr(3:0) bits represent the doubleword, word, halfword, and byte addresses that count with each datum during quad-word-burst and mini-burst reads and writes.

During DRAM transactions, SysAddr(13:2) are driven with the multiplexed DRAM row and column address. The 3 LSB's of the SysAddr bits, during the column address period, will represent the doubleword, word, or halfword addresses that count with each datum during quad-word-burst and mini-burst reads and writes.

During idle cycles between valid transactions, the behavior of Addr(25:0) is undefined.

SysData(31:0) **Input/Output**

System Data Bus: The RC36100 uses a dedicated 32-bit data bus. For reads, data is sampled on the rising edge of the SysClk reference clock. Although the data bus is 32-bits wide, the RC36100 directly supports the use of narrower memory subsystems. In these cases, the RC36100 bus unit interface will gather smaller data into the requested transfer size on reads, and break write data up into a series of smaller pieces on writes, depending on whether the port is 32, 16 or 8-bits wide. The bus interface will shift and adjust the LSB SysAddr bits accordingly for big or little endian data. Collectively, when these types of transactions are a result of word size or smaller accesses these are referred to as "mini-bursts" and quad-word accesses are referred to as "bursts". However, on the RC36100, in contrast to the RC3051 family, both mini-bursts and bursts assert the system burst signal, SysBurstFrame.

Note: During internal peripheral register reads, SysData(15:0) is driven with the register contents by the RC36100. Also note that the SysData Bus may tri-state when the Bus Interface Unit is not in use.

Clock and Reset Signals

SysClkIn **Input**

System Clock Input: This is a double frequency input clock used to generate the timing of the processor.

Notes

SysClk **Output**

System Output Clock: This output clock provides the master timing reference for all bus interface signals. Most input signals are sampled on the rising edge of SysClk, and all outputs except for the external strobes DramCAS(3:0) and SysWrEn(3:0) are generated from the rising edge of SysClk. Thus, external logic can use the rising edge of the SysClk output reference to generate control signals back to the RC36100, and to sample RC36100 outputs.

The number of loads on SysClk should be kept below or equal to a maximum of 5 CMOS type loads due to internal pin skew feedback monitoring on the RC36100. Systems requiring additional loads should either buffer or invert SysClk.

There is no guaranteed AC timing delay relationship between the SysClkIn input clock and the SysClk output clock. However, the phase relationship can be guaranteed via the method described in Chapter 19, "Debug Mode Features."

Reset **Input**

Reset: This active low input signal initializes the processor and is required after power up before correct operation can begin. Optional features of the processor including the Endianess and the Port Width of the Boot ROM are established during the last cycle of reset using the reset configuration mode inputs (also known as the reset initialization vector) which are multiplexed with the interrupt pins. See Chapter 18 for more information on the reset initialization vector.

Bus Interface Control Signals

SysALEn **Output (Input during external DMA)**

System Address Latch Enable: As an output signal, this active low signal indicates when a new address is first valid. SysALEn de-asserts high one clock after being asserted by the RC36100.

As an input signal, it is used by an external DMA agent to indicate that it has provided a valid address on the processor data bus. The RC36100 will then use this address to select which memory subsystem or peripheral is the target of the DMA, and perform the necessary access. As an input, its timing is similar to the output case; however, its assertion by the external DMA agent may be 1 or more clocks long. The address is sampled by the CPU on the first rising SysClk edge where SysALEn is asserted. Any additional SysALEn asserted clocks are ignored as far as the address is concerned, however, its assertion delays the second phase of the DMA transaction where the external DMA agent tri-states most of the bus control signals and lets the RC36100 memory controllers drive the bus control signals.

SysRd **Output (Input during external DMA)**

System Read: SysRd is always driven by the RC36100, except during external DMA cycles.

As an output, SysRd is an active low read control signal. During external read transfer cycles, this signal will be asserted. This signal can be used for external control and diagnostics as needed.

Note: During internal peripheral register reads, the SysData(15:0) bus is driven with the register contents by the RC36100.

As an input, this signal can be driven by an external DMA engine. However, the RC36100 does not use this signal during DMA. If SysWr is high when SysALEn is asserted, then a DMA read is implied.

SysWr **Output (Input during external DMA)**

System Write: SysWr is always driven by the RC36100, except during external DMA cycles.

As an output, SysWr is an active low control signal to indicate that the current transaction is a write. This signal can be used for external control and diagnostics as needed.

As an input, this signal is used internally during DMA as a read/write input signal. If SysWr is high when SysALEn is de-asserted, then a DMA read is implied.

SysBurstFrame **Output (Input during external DMA)**

System Burst: SysBurstFrame is always driven by the RC36100, except during external DMA transactions.

Notes

This active low control signal specifies that at least one more datum will be written or read. This signal is valid for both reads and writes. SysBurstFrame always asserts on the first clock of a transaction. Thereafter it de-asserts high on the first clock of the last datum. Thus, if there is only a single datum, SysBurstFrame will assert for one clock only. If there are multiple datum, SysBurstFrame remains asserted until the last datum begins.

However, an external DMA has the ability to continue a burst for an indefinite length. Under this condition, the RC36100 is unable to determine when the last datum will occur, and it is assumed that the external DMA agent is able to determine how many datum it needs to read or write.

So, on external DMA transactions, SysBurstFrame is an input signal for the first clock cycle of an RC36100 DRAM, memory, or I/O controller access. During the bus transaction phase of external DMA, SysBurstFrame reverts back to an output, but is undefined after the SysALE phase.

SysDataRdy **Output**

System Data Ready: This active low output signal indicates that the CPU is ready to receive data on a read or that it is driving data on a write.

If wait-states are inserted by the use of SysWait (as opposed to the internally wait-state generator), then SysDataRdy is kept asserted for the same number of clock cycles. Thus external diagnostic tools such as logic analyzers may want to gate SysDataRdy with SysWait.

During external DMA transactions, SysDataRdy is still driven by the CPU the entire time. Thus during the second part of the DMA transaction where the RC36100 memory controller is being used to access data, the CPU drives SysDataRdy whenever it has or expects valid data.

SysWait **Input**

System Wait: Active low is used to extend the length of the memory cycle by stalling the Memory, or I/O Controller for as many cycles as SysWait is sampled low. SysWait can be asserted anytime during single word or smaller read and write transactions. However, the effect of SysWait occurs 1 clock later due to internal pipelining of the signal. During burst reads, SysWait is valid up until the internal Acknowledge is generated to release the execution core for refill cycles. Thus systems that are quad-word bursting and using SysWait must either not use the internal Acknowledge, or guarantee that SysWait is only asserted during the early part of a cycle with the Memory or I/O Controller Wait option programmed.

The specific behavior of SysWait is dependent on the type of memory accessed. In general, its effect is to delay the de-assertion of the pertinent data strobe, such as MemRdEnOdd.

SysBusError **Input**

System Bus Error: This is an active low input signal that must be asserted at least 1 clock cycle before the internal acknowledge. Thus, systems using this feature should assert SysBusError when SysAlen asserts, which implies that SysBusError be generated asynchronously (for instance, by the equation!(! SysAlen &! DiagNoCs)). This signal causes a DBE exception to be signalled on reads, no exception is signalled on writes. Neither reads nor writes are terminated by SysBusError. Termination relies on the internal wait-state generator of the memory controllers and also on DmaLast if an external DMA burst is occurring. Users may wish the assertion of SysBusError to cause a CPU interrupt to be asserted for general handling.

CPU Core Transaction Types

The RC36100 RISController execution core is capable of requesting the following types of transactions:

Read Operation

The processor executes an instruction fetch or a data load operation as the result of either a cache miss or an uncacheable reference.

Notes

Quad word reads occur when the processor requests a contiguous block of four words from memory. Quad word reads occur in response to instruction cache misses, and will occur in response to a data cache miss if the DBlockRefill option in the CP0 Cache Configuration register is enabled. The RC36100 incorporates an on-chip 4-word deep read buffer which may be used to “queue up” the read response before passing it through to the high-bandwidth cache and execution core. Read buffering is appropriate in systems which require wait states between adjacent datums of a block read or in interfacing to memory systems narrower than 32-bits wide.

On the other hand, systems that use high-bandwidth memory techniques—such as page mode, static column, nibble mode, or memory interleaving—can effectively bypass the read buffer by providing words of the block at the processor clock rate. Note that the choice of burst vs. read buffering is independent of the initial latency of the memory; that is, burst mode can be used even if multiple wait states are required to access the first datum of the block.

Single data reads (single word, tri-byte, halfword, or byte) are used for uncacheable references (such as for I/O or boot code) and will be used in response to a data cache miss if the DBlockRefill option in the CP0 Cache Configuration register is disabled. A single data read returns one unit of data per read transaction.

Write Operations

The RC36100 utilizes an on-chip write buffer to isolate the execution core from the speed of external memory during write operations. There is a single primary type of write:

Single data writes (word, tri-byte, halfword, or byte writes corresponding to 32-bit, 16-bit, and 8-bit interfaces, respectively) are used in response to a store operation, either cached or uncached (the RC36100 uses a write-through cache).

Although the CPU execution core is capable of producing only single data writes, the DMA Controller can produce 4 word burst writes.

Quad word writes occur when the DMA Controller is instructed to transfer 4 words at a time. The DMA Controller will first read 4 words into the read buffer, then latch out the data for 4 consecutive writes.

Although the data bus is 32-bits wide, the RC36100 directly supports the use of narrower memory subsystems. In these cases, the RC36100 will gather smaller data into the requested transfer size on reads, and break write data up into a series of smaller pieces on writes. Collectively, these types of transactions are referred to as "mini-bursts".

Multiple Operations

It is possible for the RC36100 execution core to have multiple activities pending. Specifically, there may be data in the write buffer, a read request—such as due to a cache miss—or two read requests (both the I-cache and D-cache misses in a single clock cycle), even as the bus interface is servicing some external DMA activity.

In establishing the order in which the requests are processed, the RC36100 is sensitive to possible conflicts and data coherency issues as well as to performance issues. For example, if the on-chip write buffer contains data which has not yet been written to memory, and the processor issues a read request to the target address of one of the write buffer entries, then the processor strategy must insure that the read request is satisfied by the new, current value of the data.

There are two levels of priority: that performed by the CPU engine, and that performed by the bus interface unit. The internal execution engine can be viewed as making requests to the bus interface unit. In the case of multiple requests in the same clock cycle, the CPU core will:

Perform the data request first. That is, if both the data cache and instruction cache miss in the same clock cycle, the processor core will request a read to satisfy the data cache first. Similarly, a write buffer full stall will be processed before an instruction cache miss.

Perform a read due to an instruction cache miss.

This prioritization is important in maintaining the precise exception model of the MIPS architecture. Since data references are the result of instructions which entered the pipeline earlier, they must be processed (and any exceptions serviced) before subsequent instructions (and their exceptions) are serviced.

Once the processor core internally decides which type of request to make to the bus interface unit, it then presents that request to the bus interface unit.

Notes

In the RC36100 Bus Interface Unit, multiple operations are serviced in the following order:

1. DRAM refreshes may delay the start of a read or write DRAM data access.
2. Ongoing transactions are completed without interruption.
3. DMA requests are serviced according to the DMA priorities established in the RC36100 DMA Controller (For more information on this operation, see Chapter 11).
4. Instruction cache misses are processed.
5. Pending writes are processed.
6. Data cache misses or uncacheable reads/uncacheable instruction fetches are processed.

This service order has been designed to achieve maximum performance, minimize complexity, and solve the data coherency problem possible in write buffer systems.

This order assumes that the write buffer does not contain instructions which the processor may wish to execute. The processor does not write directly into the instruction cache: store instructions generate data writes which may change only the data cache and main memory. The only way in which an instruction reference may reside in the write buffer is in the case of self modifying code, generated with the caches swapped. However, in order to unswap the caches, an uncacheable instruction which modifies CP0 must be executed; the fetch of this instruction would cause the write buffer to be flushed to memory. Thus, this ordering enforces strong ordering of operations in hardware, even for self modifying code. Of course, software could perform an uncacheable reference to flush the write buffer at any time, thus achieving explicit memory synchronization with software.

Execution Engine Fundamentals

This section describes the fundamentals of the processor interface and its interaction with the execution core. These fundamentals will help to explain the relationship between design trade-offs in the system interface and the performance achieved in RC36100 systems.

Execution Core Cycles

The RC36100 execution core utilizes many of the same operation fundamentals as does the RISCore32 series processor. Thus, much of the terminology used to describe the activity of the RC36100 is derived from the terminology used to describe the RISCore32 series. In many instances, the activity of the execution core is independent of that of the bus interface unit.

Cycles

A cycle is the basic timing reference of the RC36100 execution core. Cycles in which forward progress is made (the processor pipeline advances) are called Run cycles. Cycles in which no forward progress occurs are called Stall cycles. Stall cycles are used for resolving exigencies such as cache misses, write stalls, and other types of events. All cycles can be classified as either run or stall cycles.

Run Cycles

Run cycles are characterized by the transfer of an instruction into the processor execution core, and the optional transfer of data into or out of the execution core. Thus, each run cycle can be thought of as having an instruction and data, or ID, pair.

There are two types of run cycles: cache-run cycles and refill-run cycles. Cache-run cycles, typically referred to as just-run cycles, occur while the execution core is executing out of its on-chip cache; these are the principal execution mechanism.

Refill-run cycles, referred to as streaming cycles, occur when the execution core is executing instructions as they are brought into the on-chip cache. For the RC36100, streaming cycles are defined as cycles in which data is brought out of the on-chip read buffer into the execution core, rather than defining them as cycles in which data is brought from the memory interface to the read buffer.

Stall Cycles

There are three types of stall cycles:

Wait Stall Cycles. These are commonly referred to simply as stall cycles. During wait stall cycles, the execution core maintains a state consistent with resolving a stall causing event. No cache activity will occur during wait stalls.

Notes

Refill Stall Cycles. These occur only during memory reads, and are used to transfer data from the on-chip read buffer into the caches.

Fixup Stall Cycles. Fixup cycles occur during the final cycle of a stall; that is, one cycle before entering a run cycle or entering another stall. During the final fixup cycle (the one which occurs before finally re-entering run operation), the Instruction/Data (ID) pair which should have been processed during the last run cycle is handled by the processor. The fixup cycle is used to restart the processor and co-processor pipelines, and in general to fixup conditions which caused the stall.

There are five basic stalls that are caused by the following conditions:

Read Busy Stalls: If the processor core requires read data, either to process a cache miss or an uncacheable reference, then it will be stalled until the read data is brought back to the execution core.

Write Busy Stalls: If the processor attempts to perform a store operation while the on-chip write buffer is already full, then the processor will stall until a write transaction is begun on the interface to free up room in the write buffer for the new address and data.

Multiply/Divide Busy Stalls: If software attempts to read the result registers of the integer multiply/divide unit (the HI and LO registers) while a multiply or divide operation is underway, the processor execution core will stall until the results are available.

Micro-TLB[1] Fill Stalls: These stalls can occur when an instruction translation misses in the instruction TLB cache (the micro-TLB, which is a two-entry cache of the main TLB used to translate instruction references). When such an event occurs, the execution core will stall for one cycle, in order to refill the micro-TLB from the main TLB. Since this is a single-cycle stall, it is of necessity a fixup cycle.

Multiple Stalls: Multiple stalls are possible whenever more than one stall initiating event occurs within a single run cycle. An example of this condition is when a single cycle results in both an instruction cache miss and a data cache miss. The most important characteristic of any multiple stall cycle is the validity of the Instruction/Data (ID) pair processed in the final fixup cycle. The RC36100 execution core keeps track of nested stalls to insure that orderly operation is resumed once all of the stall causing events are processed.

For the general case of multiple stalls, the service order is:

1. Micro-TLB Miss and Partial Word Store
2. Data Cache Miss or Write Busy Stall
3. Instruction Cache Miss
4. Multiply/Divide Unit Busy

Internal Acknowledgment

To speed performance, the RC36100 CPU core allows the CPU to exit wait stalls and begin refill and/or streaming, even while the bus interface continues to provide additional data to the CPU.

To do this, the RC36100 incorporates an on-chip 4-entry read buffer. In response to a quad word read, data begins to be returned to the CPU. As each datum is returned, it is entered into the read buffer. At some point, the internal core is "Acknowledged" (the "AckN" internal signal) to indicate that the read buffer contents may begin being transferred to the internal caches and execution core.

Transfer from the read buffer to the core/caches occurs at the pipeline rate. Thus, the ideal time to provide such an acknowledgment is 3 cycles before the last datum is returned to the RC36100. In this case, the last datum will be entered into the read buffer, and in the very next clock cycle be placed into the cache/core. Note that in the case of single word reads, acknowledge is provided with the last byte of the requested transfer; in the very next clock cycle, the datum is transferred into the core.

To facilitate this operation, the RC36100 requires that the various memory controllers be programmed for the optimal placement of "Ack", the internal control signal which is used to begin refill/streaming. As a rule, Ack should be placed 3 cycles before the last response datum in a quad word read.

1. Micro-TLB stalls will not occur in the RC36100, which does not include an on-chip TLB.

Notes

Read Interface Timing Overview

The read interface is designed to allow a variety of memory strategies. An overview of how data is transmitted from memory and I/O devices to the processor is discussed below.

Initiation of a Read Request

A read transaction occurs when the processor internally performs a run cycle which is not satisfied by the internal caches. Immediately after the run cycle, the processor enters a stall cycle and asserts the internal control signal $\overline{\text{MemRd}}$. This signals to the internal bus interface unit arbiter that a read transaction is pending.

Assuming that the read transaction can be immediately processed (that is, there are no ongoing bus operations and no higher priority operations pending), the processor will initiate a bus read transaction on the rising edge of SysClk which occurs during phase two of the processor stall cycle. Higher priority operations would have the effect of delaying the start of the read by inserting additional processor stall cycles.

Figure 7.2 illustrates the initiation of a read transaction, based on the internal assertion of the $\overline{\text{MemRd}}$ control signal. This figure is useful in determining the overall latency of cache misses on processor operation.

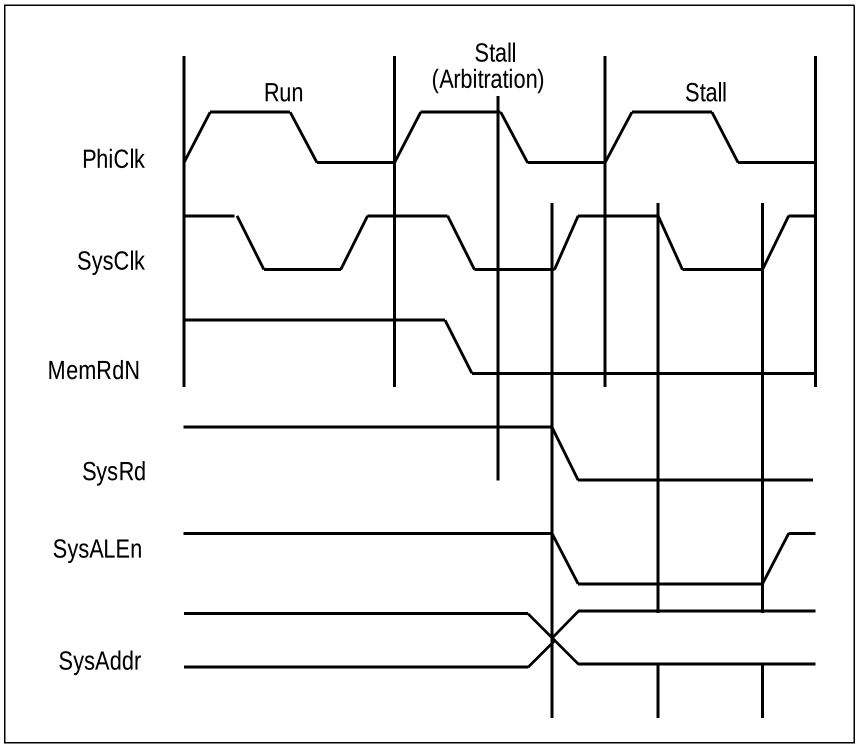


Figure 7.2 CPU Latency to Start of Read

Memory Addressing

A read transaction begins when the processor asserts its $\overline{\text{SysRd}}$ control output, and also drives the address and other control information onto the SysAddr and memory interface buses. Figure 7.3 illustrates the start of a processor read transaction.

The addressing occurs throughout the read transaction. At the rising edge of $\overline{\text{SysClk}}$, the processor will drive the read target address onto the SysAddr bus. At this time, SysALEn will also be asserted, to allow an external ASIC or peripheral to capture the address. During the initial part of the read phase, all-memory control read enables will be held high indicating that memory drivers should not be enabled onto the SysData bus.

Notes

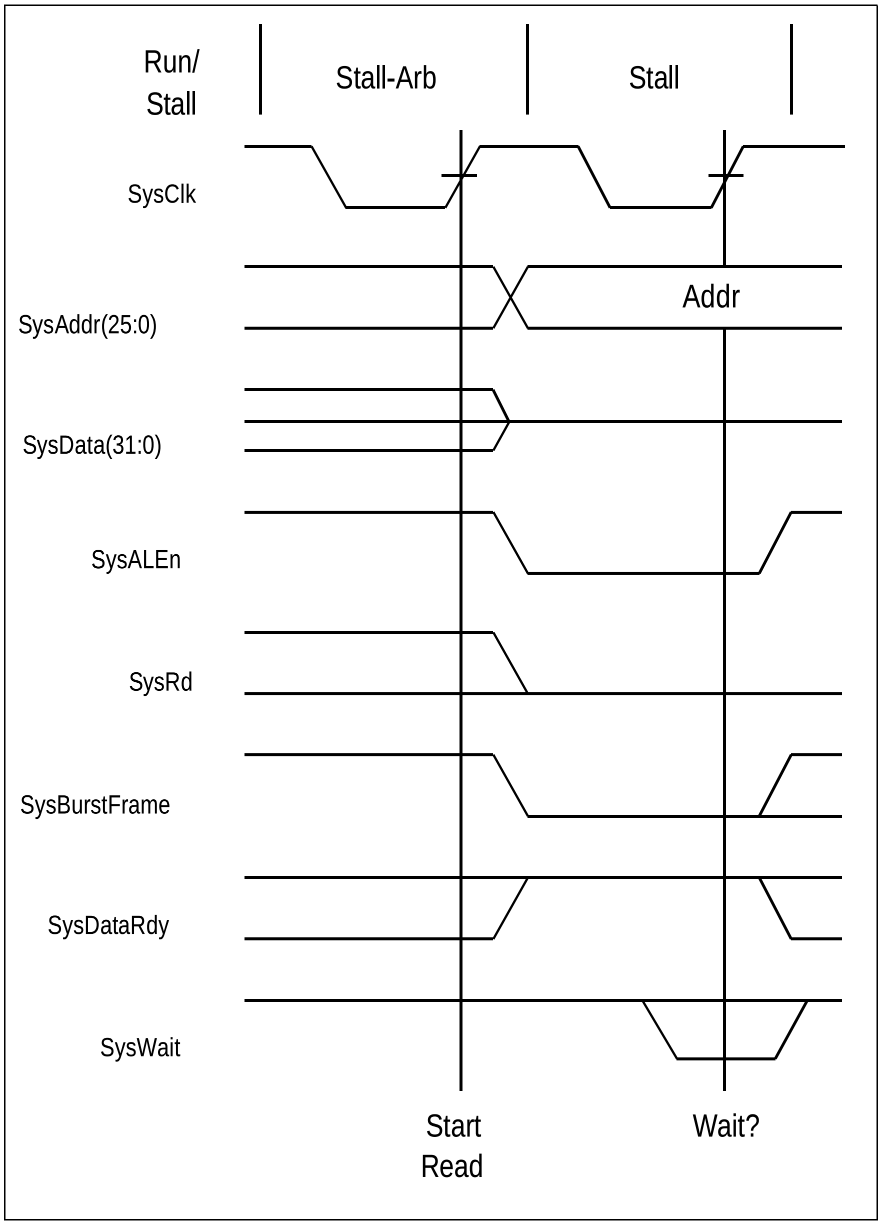


Figure 7.3 Start of Bus Read Operation

Concurrent with driving addresses on the SysAddr bus, the processor will redundantly indicate the beginning of the read transaction with SysBurstFrame asserting. A multi-datum transaction and bursts will be indicated by SysBurstFrame remaining asserted as the current datum is sampled. The functioning of the SysAddr(3:0) counter during mini-burst and burst reads is also described later.

Initiation of the Data Phase

Once the SysAddr bus has presented the address for the transfer, the various memory controller read enables assert and data is ready to be sampled.

Bringing Data into the Processor

Regardless of whether the transfer is a burst read or a single datum transfer, the basic mechanism for transferring data presented on the A/D bus into the processor is the same.

Although there are two internal control signals involved in terminating read operations, only the internal RdCEnN signal is used to cause data to be captured from the bus.

Notes

The memory system asserts internal RdCEnN to indicate to the processor that it has (or will have) data on the data bus to be sampled. The earliest that internal RdCEnN can be detected by the processor is the rising edge of SysClk after it has asserted SysALEn (start of phase 1 of the second clock cycle of the read).

If internal RdCEnN is detected as asserted by the internal wait-state generator, the processor will capture (with proper setup and hold time) the contents of the SysData bus on the immediately subsequent rising edge of SysClk. This captures the data in the internal read buffer for later processing by the execution core/cache subsystem.

The RC36100 integrates on-chip a 4-word read buffer, capable of acting as a speed-matching FIFO between the system interface and the execution core. This bus interface then performs byte or half-word gathering, and assembles them into 32-bit words for the read buffer. Thus, the bus interface supports 8-, 16-, and 32-bit memory subsystems, even for quad word reads, with no real system impact. Figure 7.4 illustrates the sampling of data by the RC36100.

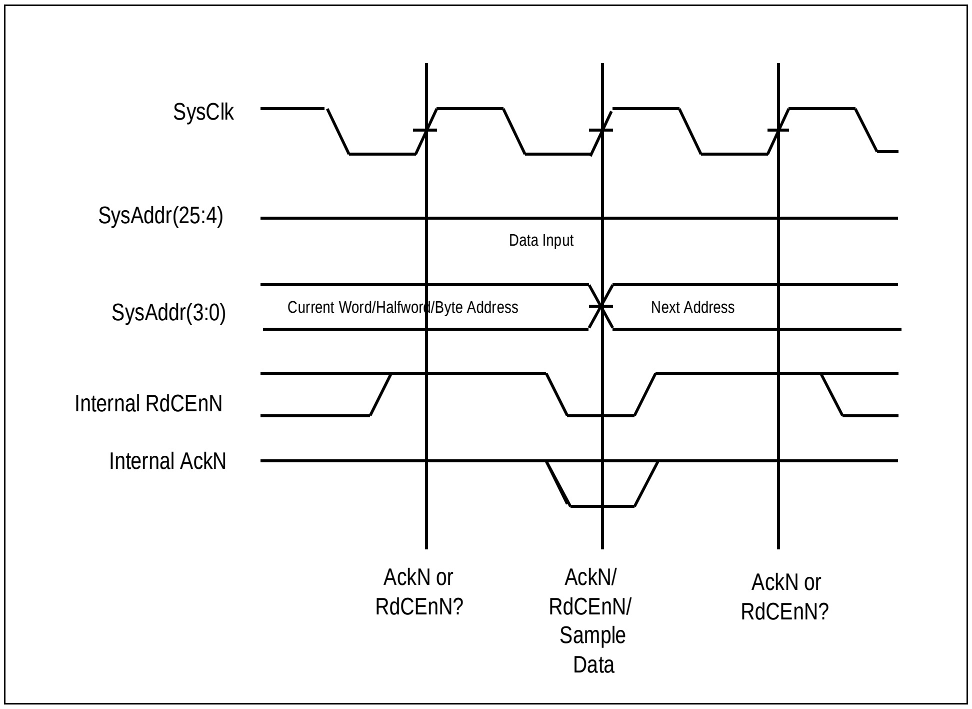


Figure 7.4 Data Sampling.

Terminating the Read

To terminate an ongoing read operation, the external memory system will use the following three methods:

- *It can supply an internal AckN (acknowledge) to the processor, to indicate that it has sufficiently processed the read request and has or will supply the requested data in a timely fashion. Note that internal AckN may be signalled to the processor "early", to enable it to begin processing the read data even while additional data is brought from the SysData bus. This is applicable only in quad-word read operations.*
- *The external memory system can supply the requested data, using internal RdCEnN to enable the processor to capture data from the bus. The processor will "count" the number of times internal RdCEnN is sampled as asserted; once the processor counts that the memory system has returned the desired amount of data (one byte to four words), it will implicitly "acknowledge" the read after it samples the last required internal RdCEnN. This technique may be important in memory systems where the latency can vary--e.g. dual ported memory.*
- *On External DMA bus transactions, in the burst read mode, the DMADone input is used to signal the end of the transaction.*

Notes

Throughout this chapter, method one will be illustrated. The other cases can be extrapolated easily from these diagrams (for example, the system designer can assume that internal AckN is asserted simultaneous with the last internal RdCEnN of a single word read transfer and 3 clocks before the last internal RdCEnN of a quad word burst read transfer).

There are actually two phases of terminating the read: there is the phase where the memory system indicates to the processor that it has sufficiently processed the read request, and the internal read buffer can be released to begin refilling the internal caches; and there is the phase in which the read control signals are negated by the processor bus interface unit.

The difference between these phases is due to block refill: it is possible for the memory system to “release” the execution core even though additional words of the block are still required; in that case, the processor will continue to assert the external read control signals until all four words are brought into the read buffer, while simultaneously refilling/executing based on the data already brought on board.

To determine the end of the read transaction, one of the following methods may be used:

- *Systems that only use 32-bit memory sub-region ports as with the RC3051 family only have single datum reads or burst reads and can either count the number of wait-cycles or use the de-asserting edge of SysRd to end the transaction.*
- *Systems that use 16 or 8-bit ports must in general support mini-burst (multi-datum) reads. Memory controllers for such systems can use the de-asserting edge of SysRd to reset the controller. The memory controller can also look for SysBurstFrame to de-assert. When SysBurstFrame de-asserts, the controller knows that it is handling the final datum of the transaction.*

Figure 7.5 shows the timing of the control signals when the read cycle is being terminated.

Latency Between Processor Operations

In general, the processor may begin a new bus activity as soon as the phase immediately after the termination of the read cycle. Although this operation may logically be either a read, write, or bus grant, there are no cases where a read operation can be signalled by the internal execution core at this time.

Since a new operation may begin one-half clock cycle after the data is sampled from the bus, it is important that the external memory system cease to drive the bus prior to this clock edge. To simplify design, the processor provides various read enable outputs for each memory controller, which can be used to control either the Output Enable of the memory device (presuming its tri-state time is fast enough), or to control the Output Enable of a buffer or transceiver between the memory device data bus and the processor SysData bus.

Notes

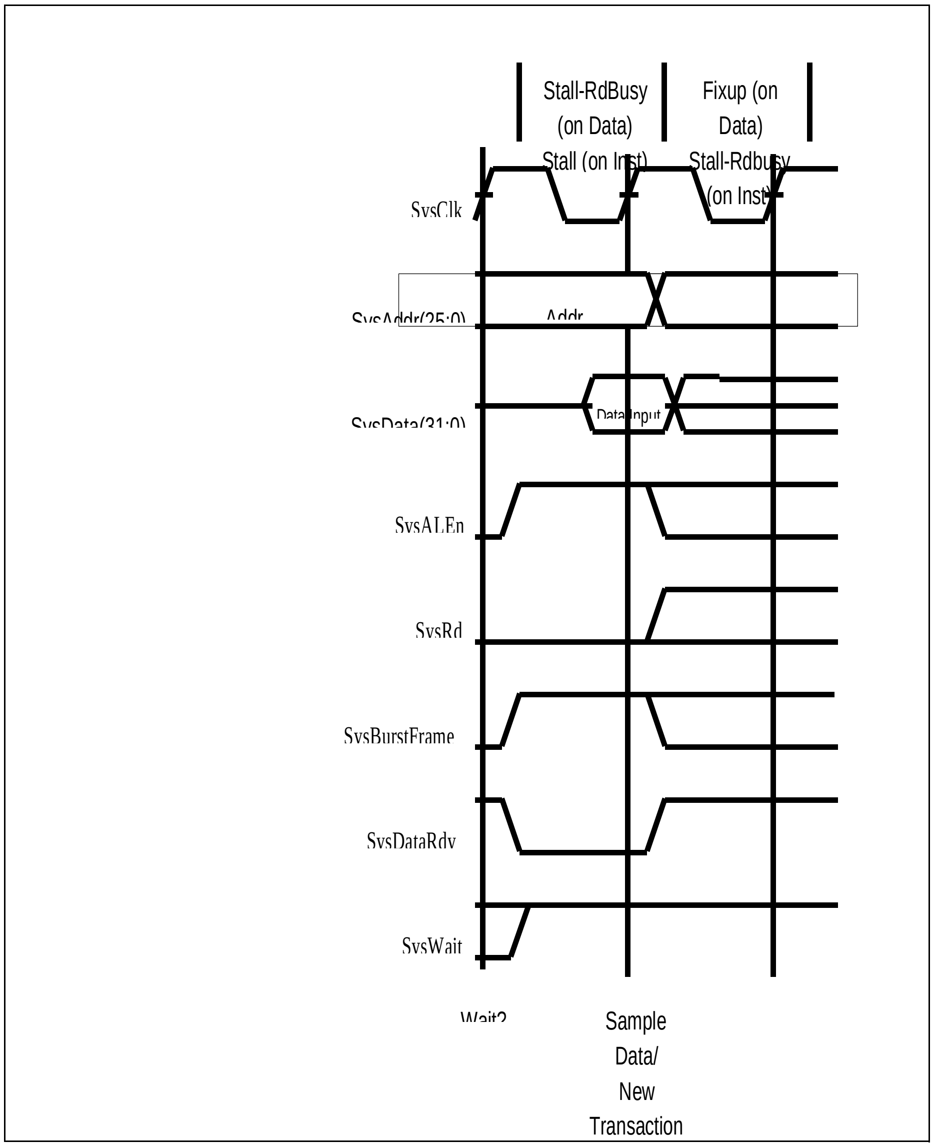


Figure 7.5 Read Cycle Termination.

The RC36100 also adds a feature to the RC3051 family to enable the system designer to lengthen the amount of time available for bus turn-around. The Bus Turn Around control field of the various memory controller Control Registers enables the system designer to extend the minimum guaranteed amount of time available for bus turn-around. This enables the system designer to eliminate some transceiver devices and/or use slower system components, without worrying about bus conflicts.

Processor Internal Activity

In general, the processor will execute stall cycles until an internal AckN is detected. It will then begin the process of refilling the internal caches from the read buffer.

The system designer should consider the difference between the time when the memory interface has completed the read, and when the processor core has completed the read. The bus interface may have successfully returned all of the required data, but the processor core may still require additional clock cycles to bring the data out of the read buffer and into the caches. Figure 7.6 illustrates the relationship between $\overline{\text{Ack}}$ and the internal activity for a block read.

Notes

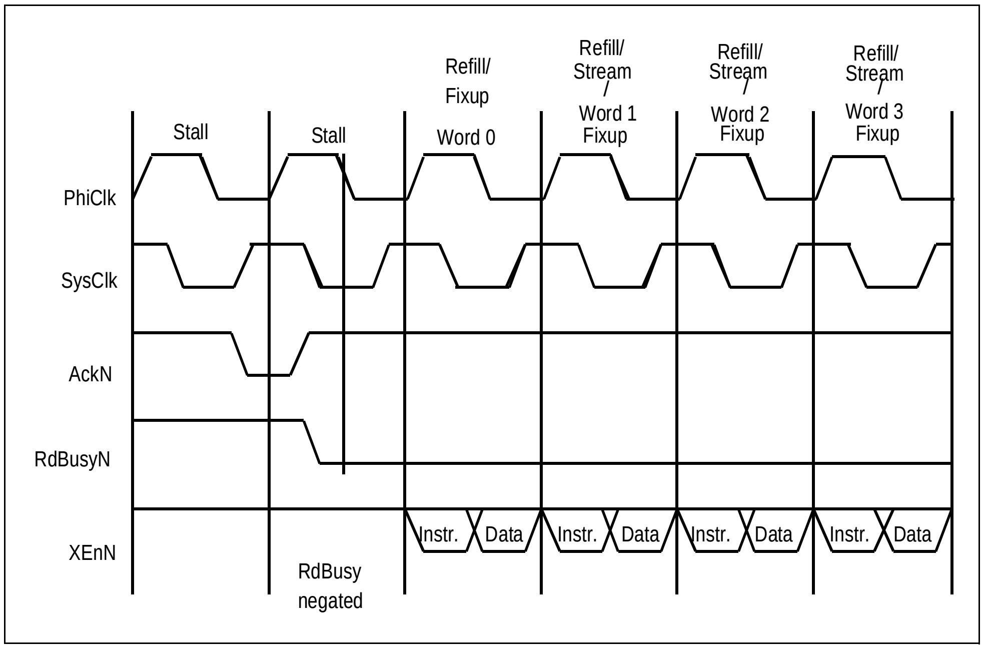

Stall
Stall
Refill/ Fixup Word 0
Refill/ Stream / Word 1 Fixup
Refill/ Stream / Word 2 Fixup
Refill/ Stream / Word 3 Fixup
PhiClk
SysClk
AckN
RdBusyN
XEnN
Instr.
Data
RdBusy negated

Figure 7.6 Internal Processor States on 4-word Burst Read.

This figure illustrates that the processor may perform either a stream, fixup, or refill cycle in cycles in which data is brought from the read buffer. The difference between these cycles is defined as follows:

- ***Refill.*** *A refill cycle is a clock cycle in which data is brought out of the read buffer and placed into the internal processor cache. The processor does not execute on this data.*
- ***Fixup.*** *A fixup cycle is a cycle in which the processor transitions into executing the incoming data. It can be thought of as a "retry" of the cache cycle which resulted in a miss.*
- ***Stream.*** *A stream cycle is a cycle in which the processor simultaneously refills the internal instruction cache and executes the instruction brought out of the read buffer.*

When reading the block from the read buffer, the processor will use the following rules:

- *For uncacheable references, the processor will bring the single word out of the read buffer using a fixup cycle.*
- *For data cache refill, it will execute either one or four refill cycles, followed by a fixup cycle.*
- *For instruction cache refill, it will execute refill cycles starting at word zero until it encounters the miss address, and then transition to a fixup cycle. It will then execute stream cycles until either the entire block is processed, or an event stops execution. If something causes execution to stop, the processor will process the remainder of the block using simple refill cycles. For example, Figure 7.7 illustrates the refill/fixup/stream sequence appropriate for a miss which occurs on the second word of the block (word address 1).*

Although this operation is transparent to the external memory system, it is important to understand this operation in order to gauge the impact of design trade-offs on performance.

Notes

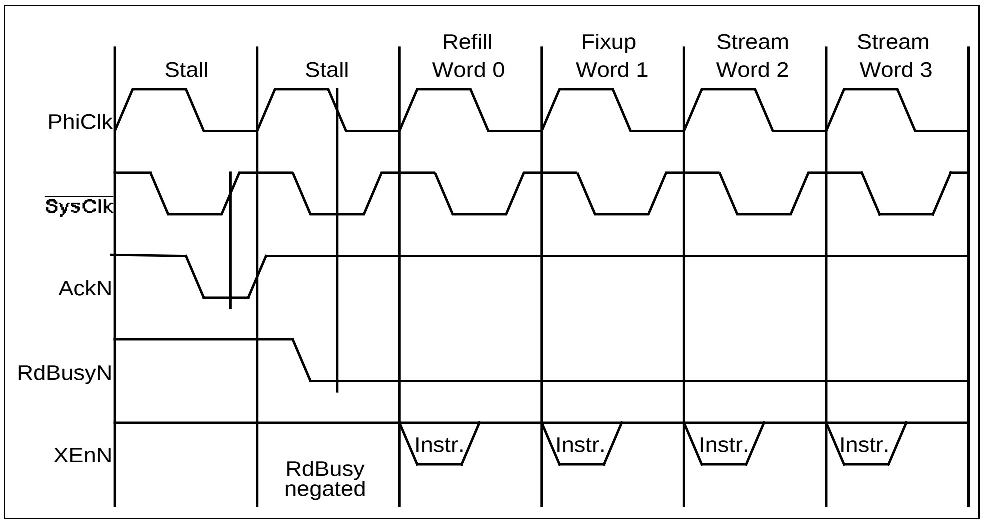


Figure 7.7 Instruction Streaming Internal Operation Example.

The Write Interface

The write protocol of the RC36100 has been designed to complement the read interface of the processor. Many of the same signals are used for both reads and writes, simplifying the design of the memory system control logic.

Importance of Writes in RC36100 Systems

The design goal of the write interface is to insure that a relatively slow write cycle does not degrade the performance of the processor. To this end, a four deep write buffer has been incorporated on-chip. The role of the write buffer is to decouple the speed of the memory interfaces from the speed of the execution engine.

The write buffer captures store information (data, address, and transaction size) from the processor at its clock rate, and later presents it to the memory interface at the rate it can perform the writes. Four such buffer entries are incorporated, thus allowing the processor to continue execution even when performing a quick succession of writes. Only when the write buffer is already filled will the processor stall; simulations have shown that significantly less than 1% of processor clock cycles are lost to write buffer full stalls.

Although it may be counter-intuitive, a significant percentage of the bus traffic will in fact be processor writes to memory. This can be demonstrated if one assumes the following:

Instruction Mix:

ALU Operations	55%
Branch Operations	15%
Load Operations	20%
Store Operations	10%

Cache Performance:

Instruction Hit Rate	95%
Data Hit Rate	90%

Notes

For these assumptions, in 100 instructions, the bus would see:

- *5 Reads to process instruction cache misses on instruction fetches*
- *10% x 20 = 2 reads to process data cache misses on loads*
- *10 store operations to the write through cache*
- *Total: 7 reads and 10 writes*

In this example, about 60% of the bus transactions are write operations, even though only 10 instructions were store operations versus 100 instruction fetches and 20 data fetches.

Types of Write Transactions

The RC36100 has two basic types of write transactions, depending on the port size selected in the CP0 Port Size Configuration register for each memory sub-region. When writes are generated from the CPU core, the 32-bit ports use only the single-word write type. DMA channels are also able to generate burst writes. The 16-bit ports can use the single halfword write or the mini-burst (double halfword) write type. And 8-bit ports are able to use the single byte write or the mini-burst (double, tri, or quad byte) or DMA burst write types.

32-Bit Write Transactions

Unlike instruction fetches and data loads, which are usually satisfied by the on-chip caches and thus are not seen at the bus interface, all 32-bit write activity is seen at the bus interface as single write transactions from the CPU core. There is no such thing as a “four word block burst write” from the CPU core; the processor performs a word or sub-word write as a single autonomous bus transaction. However, the on-chip DMA channels are capable of generating multi-word burst writes if so programmed. The SysBurstFrame output is used to decode burst writes. Successive write transactions can be processed using page mode writes by DRAM Memory Controller. This is particularly important when “flushing” the write buffer before performing a data read.

Uncached writes—which contain only 1, 2, or 3 bytes of data—assert the appropriate byte enables, MemWrEn(3:0) or DramCAS(3:0). Thus, there really is only one type of non-burst 32-bit write transaction. However, in some cases such as with the DRAM Controller, the memory system may elect to take advantage of the assertion of a page comparator internal write near signal during a write to perform quicker write operations than would otherwise be performed.

In processing 32-bit writes, there is only one parameter of interest: the latency of the write. This latency is influenced by the overall system architecture as well as the type of memory system being addressed: time required to perform address decoding and bus arbitration, memory pre-charge requirements, and memory control requirements, as well as memory access time.

The RC36100 has been designed to accommodate a wide variety of memory system designs, including no-wait cycle operations (write completed in two cycles) through simpler, slower systems incorporating many bus wait cycles.

16-Bit Transactions

When the RC36100 uses a 16-bit port, it does its writes in halfword size or smaller increments. Thus if the data contains 8 or 16 bits (1 or 2 bytes), it will be handled with a single halfword write with the appropriate byte enables, MemWrEn(1:0) or DramCAS(1:0) asserted. Note that during 16-bit accesses, MemWrEn and DramCAS bit3 is equal to bit 0 and bit 2 is equal to bit 1.

If the data contains 24 or 32 bits (3 or 4 bytes), it will be handled with a double halfword write mini-burst with the appropriate byte enables, MemWrEn(1:0) or DramCAS for each halfword asserted. A mini-burst puts both halfwords out as separate data phases of the same write transaction. The memory system simply returns an internal AckN for each halfword datum which will automatically increment SysAddr(3:1) and change the write enables if appropriate.

Similar to a read mini-burst, a write mini-burst can be detected using the SysBurstFrame signal to determine when the final halfword data is being returned or by using the de-assertion of the SysWr line. The RC36100 is designed to accommodate a wide variety of different memory bandwidths, including DRAM systems that need pre-charge wait cycles for the first halfword and then use a fast page mode access for bursting the second halfword.

Notes

The data lines used in 16-bit ports are always SysData(31:16) for big endian systems and SysData(15:0) for little endian systems. This is regardless of the Reverse Endianess bit in the CP0 Status register. For big endian systems, MemWrEn(3) and DramCAS(3) correspond to the byte lane in SysData(31:24) and MemWrEn(2) and DramCAS(2) correspond to SysData(23:16). For little endian systems, MemWrEn(1) and DramCAS(1) correspond to the byte lane in SysData(15:8) and MemWrEn(0) and DramCAS(0) correspond to SysData(7:0).

8-Bit Transactions

When the RC36100 uses an 8-bit port, it performs writes in byte size increments. Thus if the data contains 1 byte, it will be handled with a single byte write. If the data contains 2, 3, or 4 bytes, it will handled with a double, tri, or quad byte write mini-burst, respectively. A mini-burst puts 2, 3, or 4 bytes out as separate data phases of the same write transaction.

The memory system simply returns an internal AckN for each byte datum which will automatically increment SysAddr(3:0). Similar to a read mini-burst, a write mini-burst can be detected using the SysBurstFrame signal to determine when the final byte datum is being returned or by using the de-assertion of the SysWr line. The RC36100 is designed to accommodate a wide variety of different memory bandwidths, including DRAM systems that need pre-charge wait cycles for the first byte and then use a fast page mode access for bursting subsequent bytes.

The data lines used in 8-bit ports are always SysData(31:24) for big endian systems and SysData(7:0) for little endian systems. This is regardless of the Reverse Endianess bit in the CP0 Status register.

Write Interface Timing Overview

The protocol for transmitting data from the processor to memory and I/O devices are discussed below.

Initiating the Write

A write transaction occurs when the processor has placed data into the write buffer, and the bus interface is either free, or write has the highest priority. Internally, the processor bus arbiter uses the NotEmpty indicator from the write buffer to indicate that a write is being requested.

Assuming that the write transaction can be processed (that is, there are no ongoing bus operations, and no higher priority operations pending), the processor will initiate a bus write transaction on the next rising edge of SysClk. Higher priority operations would have the effect of delaying the start of the write. Figure 7.8 on page 17 illustrates the initiation of a write transaction, based on the internal negation of the internal WbEmptyN control signal. This figure applies when the processor is performing a write, and the write buffer is otherwise empty. If the write buffer already had data in it, the buffer would continually request the use of the bus until it was emptied; it would be up to the bus interface unit arbiter to decide the priority of the request relative to other pending requests. Additional stores would be captured by other write buffer entries, until the write buffer was filled.

Memory Addressing

A write transaction begins when the processor asserts its $\overline{\text{SysWr}}$ control output, and also drives the address and other control information onto the SysAddr and memory interface buses. The data is driven with SysALEn asserting. Figure 7.8 also illustrates the start of this type of processor write transaction, including the addressing of memory and presenting the store data on the SysData bus.

At the rising edge of SysClk, the processor will drive the write target address onto the SysAddr bus. At this time, SysALEn and SysBurstFrame will also be asserted, to indicate to external ASICs and peripherals that a memory transaction is beginning.

The Data Phase

Simultaneous with driving the address out, the data phase begins.

The processor enters the data phase by performing the following sequence of events:

- *It negates $\overline{\text{SysALEn}}$.*
- *It internally captures the data in a register in the bus interface unit, and enables this register onto its output drivers on the SysData bus. At this time, it begins to look for the end of the write cycle.*

Notes

Terminating the Write

There are only two methods for the external memory system to terminate a write operation:

- *It can supply the appropriate number of internal AckNs (acknowledges) to the processor by using an internal memory controller wait-state generator to indicate that it has sufficiently processed the write request, and that the processor may terminate the write.*
- *On External DMA bus transactions, in the burst write mode, the $\overline{\text{DMADone}}$ input pin is used to signal the end of the transaction.*

Figure 7.9 shows the timing of the control signals when the write cycle is being terminated.

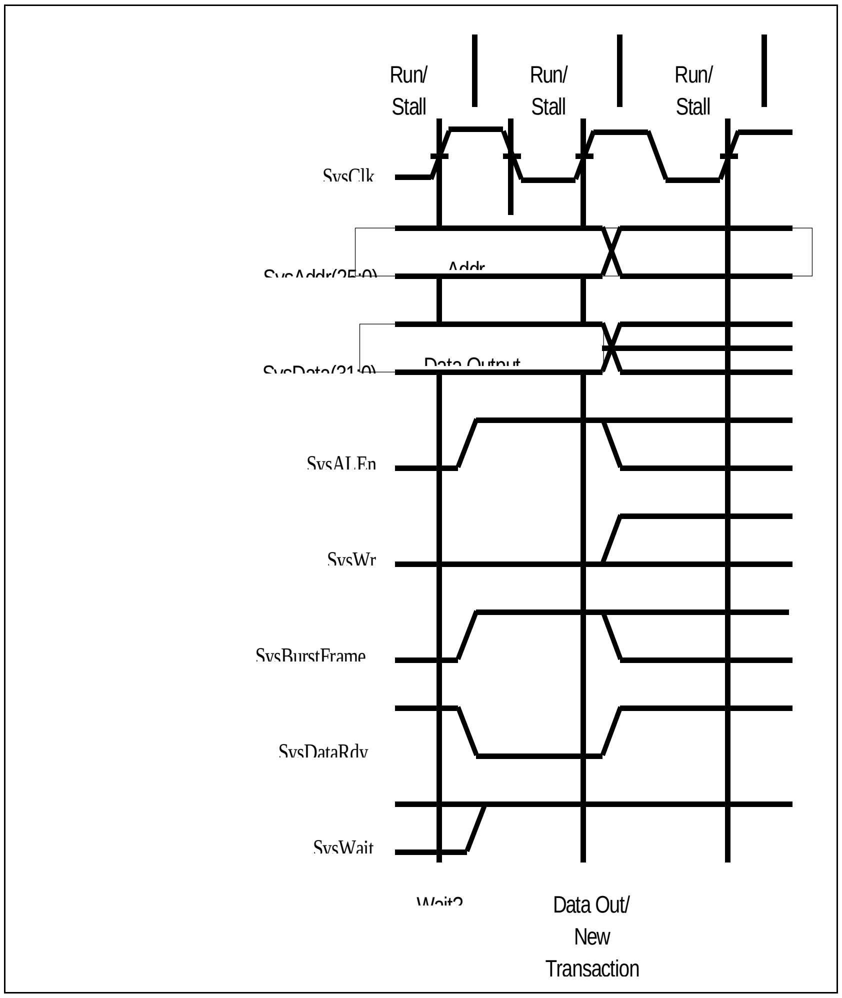


Figure 7.8 Write Cycle Termination

To determine the end of the write cycle, one of the following methods may be used:

- *Systems that only use 32-bit memory sub-region ports, such as the RC3051 family, only have single datum writes as generated from the CPU and either count the number of wait-cycles or use the de-asserting edge of SysWr to end the transaction. However, since the on-chip DMA Controller can generate burst writes, memory systems in general must be able to handle bursts.*
- *Systems that use 16 or 8-bit ports must in general support mini-burst writes. Memory controllers for such systems can use the de-asserting edge of SysWr to reset the controller. An external memory controller or logic analyzer can also look for SysBurstFrame to de-assert. When SysBurstFrame de-asserts, the controller knows that it is handling the final datum of the transaction.*

Notes

Latency Between Processor Operations

In general, the processor may begin a new bus activity in the phase immediately after the termination of the write cycle. This operation may be either a read, write, or bus grant. A new operation may begin as soon as one clock cycle after the final internal AckN is sampled from the interface.

Also note that bus turn around after a write transaction does not occur. That is, the processor continues to drive the SysAddr and SysData buses throughout the write transaction (both address and data phases), and it will also drive the SysAddr bus during the start of either a subsequent read or write transaction. Since no change in bus ownership occurs, the Bus Turn Around field of the CP0 Bus Control register does not apply after write transactions.

Write Buffer Full Operation

It is possible that the execution core on occasion may be able to fill the on-chip write buffer. If the processor core attempts to perform a store to the write buffer while the buffer is full, the execution core will be stalled by the write buffer until a space is available. Once space is made available, the execution core will use an internal fixup cycle to “retry” the store, allowing the data to be captured by the write buffer. It will then resume execution.

The write buffer can actually be thought of as “four and one-half” entries: it contains a special data buffer which captures the data being presented by an ongoing bus write transaction. Thus, when the bus interface unit begins a write transaction, the write buffer slot containing the data for that write is freed up in the second phase of the write transaction. If the processor was in a write busy stall, it will be released to write into the now available slot at this time, regardless of how long it takes the memory system to retire the ongoing write.

Note that each write buffer entry is one internal 32-bit word wide, but each entry can only hold the result of one store operation. Thus a 32-bit port can store 4 words while a 16-bit port can store up to 8 halfwords when using store word operands. However, if for example, four store byte operations are done, each byte takes a full entry.

Figure 7.9 illustrates the write-buffer-full operation.

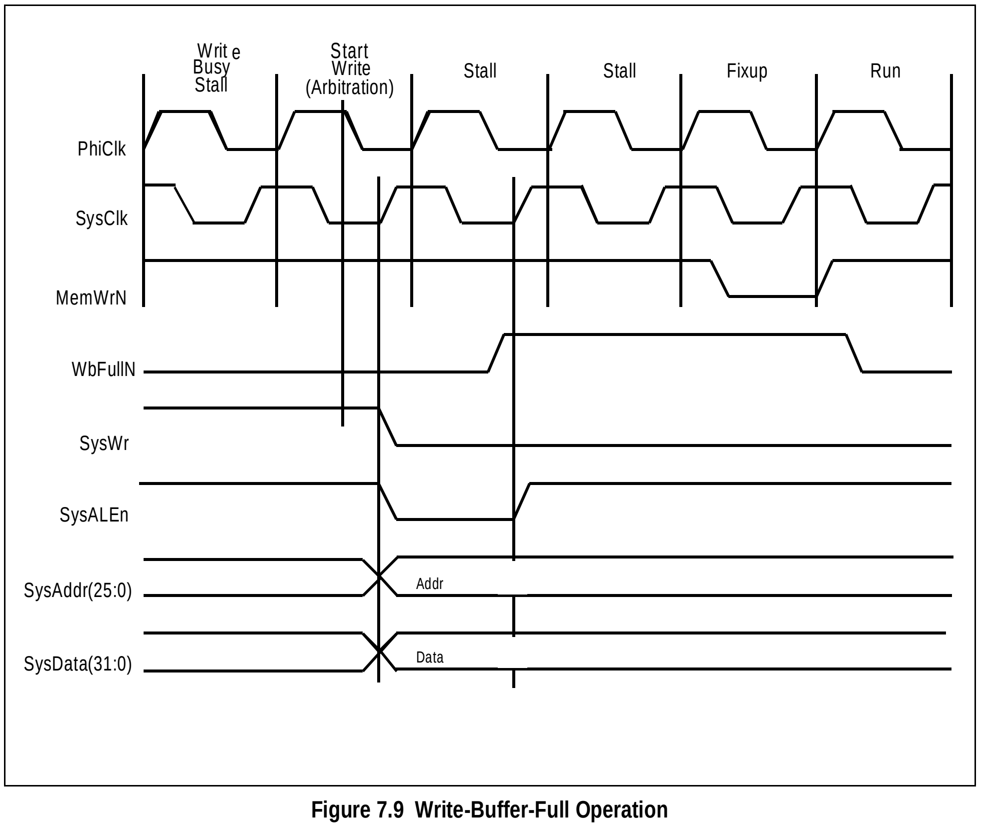


Figure 7.9 Write-Buffer-Full Operation

Memory Controller

Notes

Introduction

The IDT79RC36100 RISController integrates bus controllers and peripherals around the RISCore3000 family CPU core. One of the on-chip bus controllers is the “**Memory Controller**” and is described in this chapter.

This chapter will provide an overview of the Memory Controller interface, a complete description of the signal pins and their timing, and an explanation on how the interface relates to typical external hardware ROMs and RAMs.

Features

◆ *Controls ROM, Flash, EEPROM, SRAM and PCMCIA style memories*

◆ *Controls up to 8 banks of memory*
(Note: chip selects are shared with the I/O Controller)

◆ *Interleaved and Non-Interleaved support*

◆ *Each $\overline{\text{MemCS}}$ can be programmed to:*
- *Individual chip selects*
- *Combined interleaved pair-wise chip selects*
- *Combined PCMCIA/MEM-style pair-wise chip selects*

◆ *Each Bank has Programmable Base Address*

◆ *Each Bank Size programmable from 32KB - 64MB*

◆ *8, 16, 32-bit, and interleaved 32-bit boot prom support*

◆ *Wait-State Generator features:*
- *Programmable time from start to end of each data access for each area*
- *Programmable time options for reads and writes*
- *Programmable time options for single word accesses and for burst block accesses*
- *Internally generates the RdCEnN and AckN timing for all CPU accesses*
- *A programmed value may be overridden by the $\overline{\text{SysWait}}$ input signal*
- *Direct control of data path transceivers supports various options:*
- *Direct Bus Connection*
- *FCT260 Bidirectional Latched Multiplexor*
- *FCT245 Bidirectional Transceiver*
- *FCT543 Bidirectional Registered Transceiver*

Block Diagram

The functional block diagram of the Memory Bus Controller is shown in Figure 8.1. Starting at the bottom, the main Memory Controller Control Signal State Machine is responsible for generating the basic Memory Control signals used to connect to external PROMs, SRAMs, and other similar types of memory. These signals include chip selects, read enables/strobes, and write enables/strobes. The Memory Controller as a whole works in cooperation with the Bus Interface Unit described in Chapter 7.

Thus the Control Signal State Machine sends and receives information from the BIU Controller for assistance with controlling the port width and controlling partial word reads and writes. The Control Signal State Machine also uses information stored in the software programmable Memory Controller Register Bank for example, to control interleaved versus non-interleaved memory cycles.

Notes

The Memory Controller Wait-State Generator is located in the center of Figure 8.1. The Wait-State Generator takes care of sending and receiving information from the BIU Controller in order to control the sequencing and timing of reading and writing each individual datum. The number of wait-states is derived from the settings programmed into the Register Bank. Once the correct number of wait-states has been counted out, then the Wait-State Generator sets the appropriate internal BIU Acknowledge signals. With the programmable Wait-State Generator, it is possible to eliminate the external state machines that are traditionally used for this function.

At the top of Figure 8.1 is the Memory Controller Decoder. The decoder constantly monitors the Bus Interface Unit's address and data bus to see if either:

1. The access is to the Memory Controller's Register Bank.
2. The access is in one of the Memory Controller's Chip Select Areas that is responsible for controlling the bus transaction.

And, at the left of Figure 8.1, is the Memory Controller Register Bank. The Register Bank allows the software programmer access to the many different options of the Memory Controller.

The chip select address ranges, the number of wait-states, the port-width of the chip select, and other similar options are programmed into the Register Bank as part of the software initialization sequence of the boot operating system.

Because the Memory Controller is typically used by the boot PROM, the essential default values of the boot PROM chip select, MemCS(0) are set by the Reset Initialization Vector as described in Chapter 18, “Reset and Clocking.”

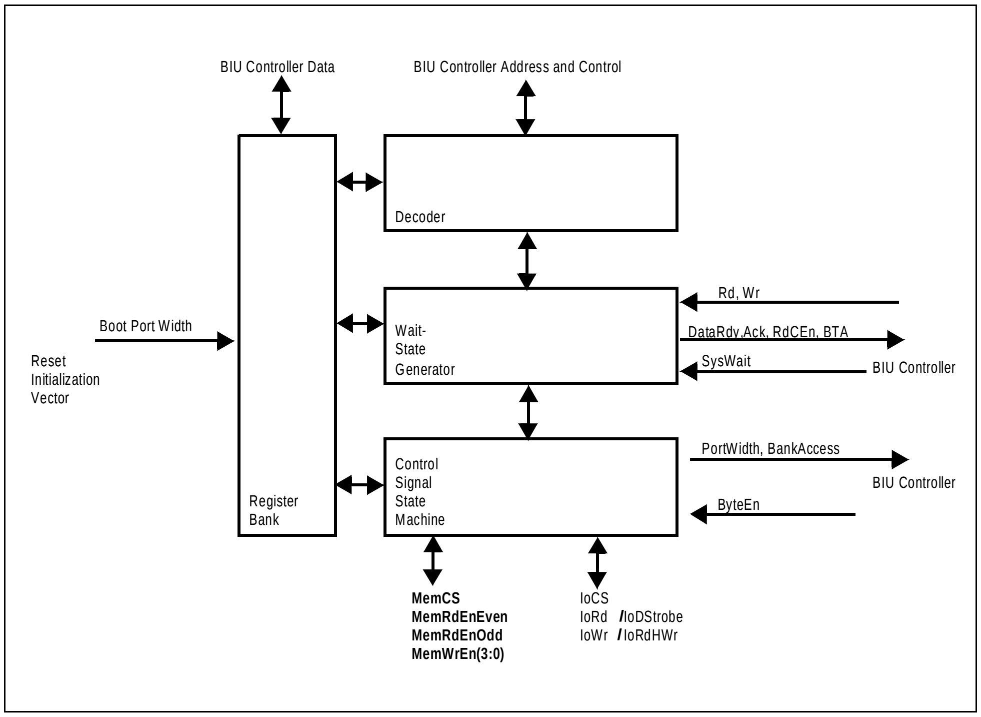


Figure 8.1 RC36100 Memory Bus Controller Block Diagram

Memory Controller Signals

These external pins are typically attached directly from the RC36100 RISController to external ROM and RAM chips and their transceivers.

Notes

MemCS(7:0)/ Output
IoCS(7:0)

Memory Chip Select: The MemCS signals are active low outputs used to select one of the programmable memory controller areas. Typically a bank of external memory chips each attach to a MemCS signal such that the memory bank can be selected and turned on during a memory transaction. When the address from the CPU or DMA Controller matches the memory block corresponding to a particular MemCS signal, that MemCS will assert at the beginning of the next memory transaction and de-asserts at the end of that transaction.

MemCS signals are used individually for non-interleaved systems or in pairs (i.e., MemCS 0 & 1, 2 & 3, 4 & 5, or 6 & 7) for interleaved systems. When using interleaved memory, the pair of chip selects are both asserted for a read but only one at a time is asserted for a write.

The boot PROM is assigned to MemCS(0) and if interleaved, MemCS(1). The port width option is determined using the Reset Initialization Vector on the ExcInt(2:0) pins. Other options are set to universal settings which the boot software can reprogram.

The MemCS chip selects are selectable and shared between Memory and I/O type-types (see Chapter 9, “I/O Bus Controller”).

MemRdEnEven Output

Memory Read Enable Even Bank: This active-low output signal is used as a read enable strobe used in conjunction with the even chip selects, MemCS(6:4:2:0). Typically, MemRdEnEven is attached to all even bank's memory chips and transceivers (if present). If the banks are interleaved, this signal is the output enable for the even bank of the selected memory pair. Whether the banks are interleaved or non-interleaved, all banks that share the same transceiver must use either all even or all odd chip selects (MemCSs) rather than an odd/even pair unless external gating circuitry is provided. MemRdEnEven controls when the memory chip and its transceiver (if present) can drive the data signals back on to the main system data bus, SysData(31:0).

Transceivers (FCT260, FCT245, or FCT543) can be used to isolate memory systems and allow for the different turnoff times for various memory devices. If transceivers are used, then during Multiplexer FCT260-Type interleaved accesses, MemRdEnEven OR's with MemRdnOdd internally such that MemRdEnEven remains asserted for the majority of the bus transaction for both even and odd accesses. Typically in the Multiplexer Mode, MemRdEnEven is attached to the common output enable input pin of the multiplexer while MemRdEnOdd is attached to the path input pin of the multiplexer. During FCT245-Type interleaved type accesses, MemRdEnEven OR's with an internally generated MemWrEnEven.

MemRdEnOdd Output

Memory Read Enable Odd Bank: This active-low output signal is used as a read enable strobe used in conjunction with the odd chip selects MemCS(7:5:3:1). Typically, MemRdEnOdd is attached to all odd bank's memory chips and transceivers (if present). During FCT245-Type interleaved type accesses, MemRdEnOdd OR's with MemWrEnOdd internally. Please see the signal description for MemRdEnEven for more information.

MemWrEn(3:0) / Output/(Input during DMA)
MemByteEn(3:0)/
MemAddr(29:26)/

System Write Enable: These dedicated byte enable strobes are always driven by the RC36100, except during the Address Strobe of external DMA cycles. The pins always act as write enables except during PCMCIA-type accesses or PCI-style accesses.

Output - During 32-bit accesses, these strobes are always de-asserted, except for the appropriate byte lane(s) for partial-word writes and mini-bursts. SRAM and Flash EPROM memories can directly connect their byte write enables to the RC36100 MemWrEn(3:0) signals. During 16-bit accesses, either MemWrEn(3:2) or MemWrEn(1:0) are used as both pairs will be equivalent. During 8-bit accesses, either MemWrEn(3) or MemWrEn(0) are used as they will be equivalent.

Notes

Input - During External DMA accesses, the external DMA Controller must assert the appropriate byte lanes during Phase 1 of a write when SysALEn and SysWr are asserted. The byte enables are ignored by the BIU Controller during external DMA reads.

MemWrEn(3:0) assert and de-assert on the falling edge of SysClk, whereas most control signals use the opposite edge of the clock. Thus it is not generally recommended that they be used for any external state machine inputs. During idle cycles, MemWrEn(3:0) will return to inactive (high).

Memory Byte Enable Bus: Similar to PCI style accesses, the MemWrEn(3:0) bus can be programmed to assert the appropriate byte lanes on both reads and writes, rather than just on writes. In this mode as with the other modes, MemByteEn(3:0) are required to return de-asserted high at the end of every bus transaction.

Memory Address Bus: During a PCMCIA Memory or I/O type access, the MemWrEn(3:0) bus is instead driven with inverted Physical Address bits. On the RC36100 and R3051-base family memory map, virtual and physical addresses (29:0) are the same. An application using PCMCIA can for example use MemAddr(27:26) to externally decode PCMCIA style chip selects into as many as four (256M/64M = 4) slots. In this mode as in the other modes, the signals all return inactive high at the end of the bus transaction.

BIU Controller Signals

Many of the BIU Controller Signals are necessary to complete the memory interface. These signals are listed here as a reminder. Information specific to the Memory Controller is given here and general information about the signal is given in Chapter 7, “BIU Controller.”

SysAddr(25:0) Output/Input

System Address Bus: SysAddr is an output bus when used with the Memory Controller. Typical 32-bit memory banks connect the word offset of the Least Significant Bits (LSB) of SysAddr to each memory chip. Thus the typical 32-bit memory bank skips SysAddr(1:0) and connects starting with SysAddr(2) on up. Typical 16-bit memory banks connect the halfword offset of the LSBs of SysAddr to each memory chip starting with SysAddr(1) on up. Typical 8-bit memory banks connect the LSBs of SysAddr to each memory chip starting with SysAddr(0) on up.

SysData(31:0) Output/Input

System Data Bus: Typical 32-bit memory banks connect the entire 32-bit SysData bus to the memory chips' data pins or to their transceivers. *16-bit and 8-bit memory banks connect to particular data pins depending on whether the Endianess of the system is Big Endian or Little Endian.* Thus 16-bit memory banks use SysData(31:16) if they are Big Endian and SysData(15:0) if they are Little Endian. 8-bit memory banks use SysData(31:24) if they are Big Endian and SysData(7:0) if they are Little Endian. The User Mode Reverse Endianess Bit in the CP0 Status Register has no effect on the connections to SysData, however, it strongly recommended that the Reverse Endianess Bit not be used to “correct” an Endianess connection as it does not function in the kernel mode, nor does it correct the boot PROM Endianess.

SysWait Input

System Wait Negated: SysWait can be used by an external source to add wait-states to the Memory Controller. Since the Memory Controller itself has a Wait-State Generator, SysWait typically is not needed and can be pulled-up with a resistor. The most likely application of SysWait is for an asynchronous memory event such as a Dual-Port Memory Busy signal which can be used to attach to SysWait to delay the beginning of a Memory Transaction in the Wait Mode option of the Memory Control Register 2. Please see Chapter 7. “BIU Controller” for a general description of SysWait.

Memory Controller Overview

The Memory Controller (MemCntrl) provides control for all memory spaces except for the DRAM space. These memory areas are intended for use by items such as boot ROM, Flash memory cards, additional EPROM and EEPROM space, SRAMs, and Dual Port RAM. Such memories typically have address inputs, data I/O, chip select, read output enable, and if writable, a write enable strobe.

Notes

Chip Selects

The Memory Controller (MemCntrl) contains up to 8 separate memory spaces, each having its own Memory Controller Chip Select (MemCS) output pin. Each MemCS space occupies from 8K to 256MB of address space of which 64MB is externally addressable (due to the 26 address lines). The address space that each MemCS decodes is programmable. The MemCntrl will use the programmed information in the MSB and LSB Base Address Registers along with the size (8K to 256MB) of the given area as programmed in the MSB and LSB Page Mask Registers. This information is used to compare with the address asserted by the CPU-BIU or DMA Controller to determine if that particular MemCS area is being accessed for the current read or write. Each area supports single reads, burst reads, single writes, and burst writes. The port size of the data path (8, 16, 32-bit, or interleaved) of each area is also programmable with each area's Control Register.

The MemCS signals can be used in pairs for interleaving. The pairs are MemCS(1:0) for one interleaved area and for the others, MemCS(3:2), MemCS(5:4), and MemCS(7:6). When interleaved, both chip selects within a pair must be programmed to the same values by the user. Note that the memory controller does not support seamless jumperless upgrades from single bank to interleaved systems because of the system dependent address multiplexing involved. This can be done externally for RAM types, however, for ROM types, the PROM chip programmer would require “switching of the binary object file” which is rarely worth the trouble.

Transceiver Control Interface

The Memory Controller provides transceiver output enables and write enables that are suitable for direct bus connection, FCT260 multiplexors, FCT245 transceivers, or for FCT543 bidirectional registers. The selection of the type of memory is software programmable. 8/16/32-bit wide Boot PROMs can use either the direct bus connection, FCT244s, or FCT543s. FCT245s can be used if the Boot PROM initializes the Memory Controller before doing any writes. Interleaved Boot PROMs are assumed to use the FCT260 type. FCT245 or FCT543 Boot PROM types must initialize the memory controller, before attempting to switch banks, perform writes, or perform burst reads.

Wait-State Generator

The Wait-State Generator (WSG) controls the speed of the memory accesses to and from the Bus Interface Unit Controller. This includes the time from the start of a memory transaction until the first datum is sent or received and the time between consecutive datum on burst transactions. The WSG also is programmed to generate the internal RdCEnN and AckN signals for CPU read and write requests.

The internal Acknowledge signal, “AckN” (described in Chapter 7) is the same as the external signal pin that the R3051 RISController family uses. On single word reads and on both single word and burst writes, AckN is automatically placed at the end of the transaction by the WSG. However, on burst reads, the user is required to program the correct value into each corresponding MemCS area's Control Register “Burst Ack” field, to optimize the CPU pipeline restart after a burst read. The most optimal value is 3 pipeline clocks previous to when the last datum is sampled. Less optimal values can be used if, for instance, SysWait is used. For more information, refer to the “Start2BurstAck” chart at the end of this chapter.

The signal called SysWait can be used to override the Wait-State Generator’s programmed settings. The actual action performed by the WSG when SysWait is asserted will depend on when it is asserted, relative to the transaction. SysWait has a pipeline delay, such that it must be asserted two clocks before the desired effect is noticeable. By asserting it immediately after a datum is received or transmitted, the next datum can be delayed. However, use for this purpose is generally not recommended since the WSG has the same functionality.

The SysWait signal is useful for accessing memories such as Dual-Port-type memory, off-card “Ack”-type memory, and PCI-style memory.

Register Option Programmability

The Memory Controller contains 8 sets of registers, one set for each chip select. These registers allow the Memory Controller to be configured for different types and speeds of memory. Thus, almost any system speed/cost/manufacturing trade-off can be accommodated.

Notes

Register Descriptions

The Memory Controller Registers are divided into 8 sets of registers, one set for each chip select memory area. Table 8.1 and Table 8.2 provide the address map and descriptions of the Memory and I/O Controller registers. These registers are shared with the I/O Controller as described in Chapter 9.

Phys. Address	Register Description
0xFFFF_E200	Memory and I/O LSB Base Address Register for Bank 0
0xFFFF_E204	Memory and I/O MSB Base Address Register for Bank 0
0xFFFF_E208	Memory and I/O LSB Bank Mask Register for Bank 0
0xFFFF_E20C	Memory and I/O MSB Bank Mask Register for Bank 0
0xFFFF_E210	Memory and I/O Control Register for Bank 0
0xFFFF_E214	Memory and I/O LSB Wait-State Generator Register for Bank 0
0xFFFF_E218	Memory and I/O MSB Wait-State Generator Register for Bank 0
0xFFFF_E220	Memory and I/O LSB Base Address Register for Bank 1
0xFFFF_E224	Memory and I/O MSB Base Address Register for Bank 1
0xFFFF_E228	Memory and I/O LSB Bank Mask Register for Bank 1
0xFFFF_E22C	Memory and I/O MSB Bank Mask Register for Bank 1
0xFFFF_E230	Memory and I/O Control Register for Bank 1
0xFFFF_E234	Memory and I/O LSB Wait-State Generator Register for Bank 1
0xFFFF_E238	Memory and I/O MSB Wait-State Generator Register for Bank 1
0xFFFF_E240	Memory and I/O LSB Base Address Register for Bank 2
0xFFFF_E244	Memory and I/O MSB Base Address Register for Bank 2
0xFFFF_E248	Memory and I/O LSB Bank Mask Register for Bank 2
0xFFFF_E24C	Memory and I/O MSB Bank Mask Register for Bank 2
0xFFFF_E250	Memory and I/O Control Register for Bank 2
0xFFFF_E254	Memory and I/O LSB Wait-State Generator Register for Bank 2
0xFFFF_E258	Memory and I/O MSB Wait-State Generator Register for Bank 2
0xFFFF_E260	Memory and I/O LSB Base Address Register for Bank 3
0xFFFF_E264	Memory and I/O MSB Base Address Register for Bank 3
0xFFFF_E268	Memory and I/O LSB Bank Mask Register for Bank 3
0xFFFF_E26C	Memory and I/O MSB Bank Mask Register for Bank 3
0xFFFF_E270	Memory and I/O Control Register for Bank 3
0xFFFF_E274	Memory and I/O LSB Wait-State Generator Register for Bank 3
0xFFFF_E278	Memory and I/O MSB Wait-State Generator Register for Bank 3
0xFFFF_E280	Memory and I/O LSB Base Address Register for Bank 4
0xFFFF_E284	Memory and I/O MSB Base Address Register for Bank 4
0xFFFF_E288	Memory and I/O LSB Bank Mask Register for Bank 4
0xFFFF_E28C	Memory and I/O MSB Bank Mask Register for Bank 4
0xFFFF_E290	Memory and I/O Control Register for Bank 4
0xFFFF_E294	Memory and I/O LSB Wait-State Generator Register for Bank 4
0xFFFF_E298	Memory and I/O MSB Wait-State Generator Register for Bank 4

NOTES to table:
1. Big Endian software must offset these addresses by b'10 (0x2), if halfword operations are used.

Table 8.1 List of the Memory and I/O Controller Registers (1 of 2).

Notes

Phys. Address	Register Description
0xFFFF_E2A0	Memory and I/O LSB Base Address Register for Bank 5
0xFFFF_E2A4	Memory and I/O MSB Base Address Register for Bank 5
0xFFFF_E2A8	Memory and I/O LSB Bank Mask Register for Bank 5
0xFFFF_E2AC	Memory and I/O MSB Bank Mask Register for Bank 5
0xFFFF_E2B0	Memory and I/O Control Register for Bank 5
0xFFFF_E2B4	Memory and I/O LSB Wait-State Generator Register for Bank 5
0xFFFF_E2B8	Memory and I/O MSB Wait-State Generator Register for Bank 5
0xFFFF_E2C0	Memory and I/O LSB Base Address Register for Bank 6
0xFFFF_E2C4	Memory and I/O MSB Base Address Register for Bank 6
0xFFFF_E2C8	Memory and I/O LSB Bank Mask Register for Bank 6
0xFFFF_E2CC	Memory and I/O MSB Bank Mask Register for Bank 6
0xFFFF_E2D0	Memory and I/O Control Register for Bank 6
0xFFFF_E2D4	Memory and I/O LSB Wait-State Generator Register for Bank 6
0xFFFF_E2D8	Memory and I/O MSB Wait-State Generator Register for Bank 6
0xFFFF_E2E0	Memory and I/O LSB Base Address Register for Bank 7
0xFFFF_E2E4	Memory and I/O MSB Base Address Register for Bank 7
0xFFFF_E2E8	Memory and I/O LSB Bank Mask Register for Bank 7
0xFFFF_E2EC	Memory and I/O MSB Bank Mask Register for Bank 7
0xFFFF_E2F0	Memory and I/O Control Register for Bank 7
0xFFFF_E2F4	Memory and I/O LSB Wait-State Generator Register for Bank 7
0xFFFF_E2F8	Memory and I/O MSB Wait-State Generator Register for Bank 7
NOTES to table: 1. Big Endian software must offset these addresses by b'10 (0x2), if halfword operations are used.	

Table 8.2 List of the memory and I/O Controller Registers (2 of 2)

Memory MSB Base Address Register for Bank 7..0 ('MemMSBBaseAddrReg(7..0)'), and Memory LSB Base Address Register for Bank 7..0 ('MemLSBBaseAddrReg(7..0)')

There are 8 pairs of Base Address MSB & LSB Registers, a pair for each Memory Chip Select (MemCS). Each pair of memory base address registers is concatenated into an internal 32-bit register and refers to the most significant 16 address bits and the least significant 16 address bits.

The formats of the MemMSBBaseAddrReg and MemLSBBaseAddrReg are displayed in Figure 8.2 and Figure 8.3. These registers are both readable and writable. The Base Address Registers are used to determine the starting location of a particular Memory Chip Select.

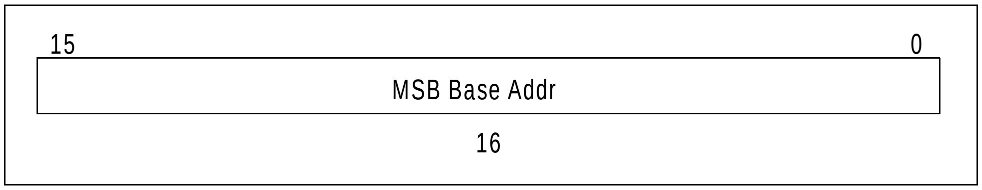


Figure 8.2 Memory and I/O MSB Base Address Register ('MemMSBBaseAddrReg').

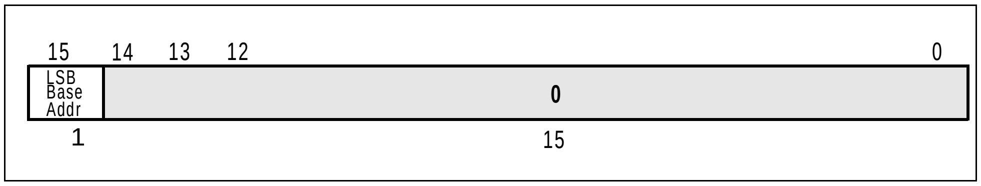


Figure 8.3 Memory and I/O LSB Base Address Register ('MemLSBBaseAddrReg').

Notes

Because of the possibility of interleaving, there are four groups of two chip selects, as follows:

Group 0:	MemCS(1:0)
Group 1:	MemCS(3:2)
Group 2:	MemCS(5:4)
Group 3:	MemCS(7:6)

Bits 31:28 of each group must be identically programmed since the internal hardware uses bits 31:28 from the even register, $\overline{\text{MemCS(0,2,4,6)}}$, for each group. This corresponds to setting each group of four chip selects into 1 of 16 possible 256MB address spaces.

Bits 27:15 must be programmed to the desired base address. This corresponds to separate address spaces for each chip select of 32KB to 256MB.

Internally, bits 14:0 are reserved to ‘0’ and must be programmed as ‘0’. This corresponds to having minimum contiguous chip select banks of 32KB. The default base addresses at reset are shown in Table 8.3.

In summary:

1. 1. Bits 31:28 of each group of four $\overline{\text{MemCS}}$ base addresses is set by the first $\overline{\text{MemCS}}$ of the group.
2. 2. Bits 27:15 of each $\overline{\text{MemCS}}$ base address is used to distinguish the starting address of each memory space within a group.
3. 3. Bits 14:0 are always ignored.

Chip Select	Default Value
MemCS(0)	0x 1FC0_0000
MemCS(1)	0x 1FF0_0000 non-interleaved (if interleaved, then 0x 1FC00000)
MemCS(2)	0x 2FC0_0000
MemCS(3)	0x 2FF0_0000 non-interleaved (if interleaved, then 0x 2FC0_0000)
MemCS(4)	0x 3FC0_0000
MemCS(5)	0x 3FF0_0000 non-interleaved (if interleaved, then 0x 3FC0_0000)
MemCS(6)	0x 4FC0_0000
MemCS(7)	0x 4FF0_0000 non-interleaved (if interleaved, then 0x 4FC0_0000)

Table 8.3 Memory and I/O Controller Base Addresses.

Memory MSB Bank Mask Register for Bank 7..0 ('MemMSBBankMaskReg(7..0)’), and Memory LSB Bank Mask Register for Bank 7..0 ('MemLSBBankMaskReg(7..0)’)

There are 8 pairs of Bank Mask Registers, one pair for each chip select ($\overline{\text{MemCS}}$). The two Bank Mask Registers are concatenated into an internal “32-bit” register and refer to the most significant 16 address bits and the least significant 16 address bits.

The formats of the MemMSBBankMaskReg and MemLSBBankMaskReg are displayed in Figure 8.4 and Figure 8.5. These registers are both readable and writable and bits 31:15 are set and bits 14:0 are cleared by default on reset.

Notes

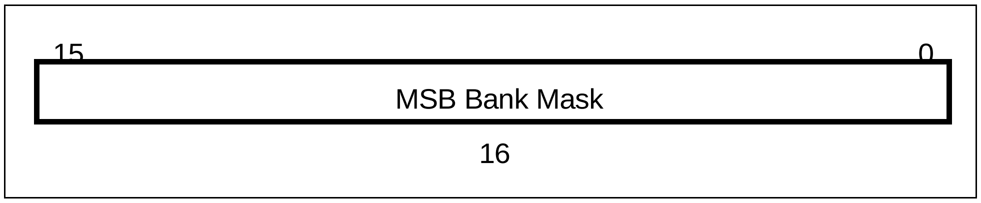


Figure 8.4 Memory and I/O MSB Bank Mask Register ('MemMSBBankMaskReg').

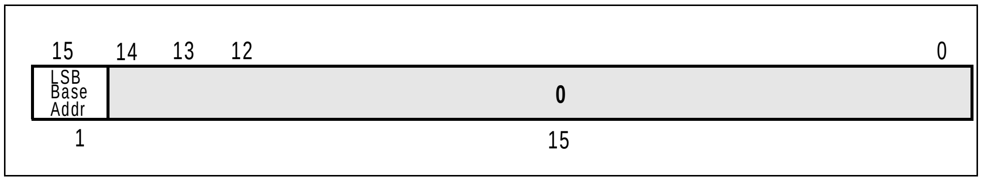


Figure 8.5 Memory and I/O LSB Bank Mask Address Register ('MemLSBBankMaskReg').

The Bank Mask Registers are used to decide which address bits in the base address are to be used for comparing whether a chip select ($\overline{\text{MemCS}}$) is to be activated. The internal grouping of chip selects is as follows:

Group 0:	MemCS(1:0)
Group 1:	MemCS(3:2)
Group 2:	MemCS(5:4)
Group 3:	MemCS(7:6)

Bits 31:28 of each group must be programmed identically since the internal hardware uses bits 31:28 from the even register, $\overline{\text{MemCS}}$(0,2,4,6) for each group. This corresponds to setting each group of four chip selects into 1 of 16 possible 256M address spaces.

Internally, bits 27:13 must be programmed to the desired Bank mask. This corresponds to separate address spaces for each chip select of 8K to 256M. Bits 12:0 are reserved to ‘0’ and must be programmed as ‘0’. This corresponds to having minimum contiguous chip select banks of 8K.

In summary:

1. Bits 31:28 of each group of four $\overline{\text{MemCS}}$ Bank masks is set by the first $\overline{\text{MemCS}}$ of the group.
2. Bits 27:15 of each MemCS Bank mask are used to distinguish the size of each memory space.
3. Bits 14:0 are always ignored.

Table 8.4 lists the values and actions for the Memory Mask Field Encoding, Figure 8.6 shows the Memory and I/O Control Register, and Table 8.5 provides the register’s bit assignments.

Value	Action
‘1’	Bit is used in Address comparison
‘0’	Bit is masked out

Table 8.4 Memory Mask Field Definitions and Values

Notes

Memory and I/O Control Register for Bank 7..0 ('MemControlReg(7..0)'),

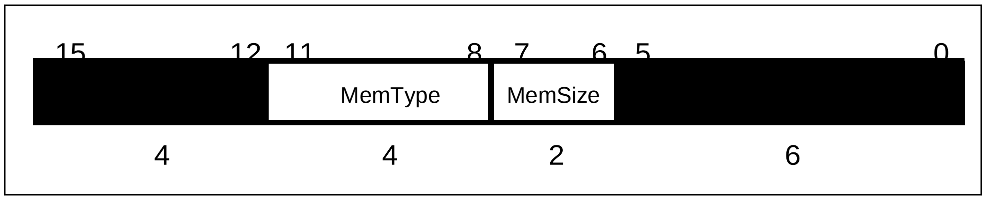


Figure 8.6 Memory and I/O Control Register Bit Assignments.

Bit	Assignment
11:8	Memory Type ('MemType')
7:6	Port Size Width ('MemSize')

Table 8.5 Memory and I/O Control Register Bit Assignments.

Memory Type ('Type') Field

The Type field determines the type of timing the Bus Interface will use. Values and actions for this field are listed in Table 8.6.

Value	Action
'1011'	PCMCIA-Memory-Style
'1010'	PCMCIA-I/O-Style
'1001'	M-Type I/O
'1000'	I-Type I/O
'0010'	Memory-Type for FCT260 (default for interleaved boot reset option)
'0001'	Memory-Type for FCT245
'0000'	Memory-Type for FCT543
All others	Reserved, undefined

1. Use FCT543 mode for FCT260 or FCT543 non-interleaved even banks.
2. FCT245 banks can be booted in the default FCT543 mode but must be put into the FCT245 mode before any writes occur.
3. PCMCIA-Style supports a PCMCIA host mode subset that is likely to be used with PCMCIA peripherals. PCMCIA-Memory and -IO Styles are intended for dynamic swapping by the software onto the same pair of chip selects. Typically, the Memory-Style is left on, and the I/O-Style is swapped in whenever it is needed, then swapped back to Memory-Style.

Table 8.6 Memory Type Field ('MemType') Encoding.

Port Size Width ('MemSize') Field

The PortSize field determines the width of the memory or I/O port. The value is inverted relative to the reset initialization vector value. Encoding information for this field is listed in Table 8.7

Notes

Value	Action
‘11’	64-bit (32-bit 2-way interleaved) accesses (Valid for Memory Type only)
‘10’	16-bit accesses
‘01’	8-bit accesses
‘00’	32-bit accesses

Table 8.7 PortSize ('MemSize') Encoding.

Memory LSB Wait-State Register for Bank 7..0 ('MemLSBWaitStateReg(7..0)’)

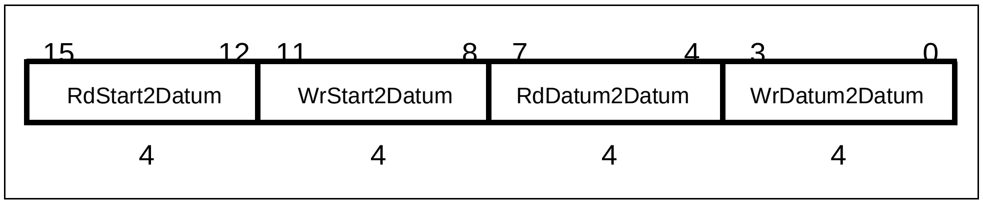


Figure 8.7 Memory LSB Wait-State Register ('MemLSBWaitStateReg').

The Wait-State Generator registers provide fields to control the access timings to/from the CPU and the memory control and I/O control areas or these control areas for DMA accesses. The parameters controlled are:

1. Time from CS asserted to the first RdCEnN for read burst or time to RdCEnN and AckN for single word access or time to the first AckN for a burst DMA write.
2. Time between the RdCEnN's for burst reads or AckN's for burst DMA writes.
3. Time from the first RdCEnN to the AckN for burst reads or time from first AckN to last AckN for burst DMA writes.
4. The functionality of the SysWaitN signal for the corresponding control area.

There is a MemLSBWaitStateReg for each of the control areas or a total of 8 MemLSBWaitStateReg's.

The various fields and bit assignments of the Memory LSB Wait-State Register are shown in Figure 8.7 and Table 8.8.

Bit	Assignment
15:12	RdStart2Datum
11:8	WrStart2Datum
7:4	RdDatum2Datum
3:0	WrDatum2Datum

Table 8.8 Memory LSB Wait-State Register ('MemLSBWaitStateReg') Bit Assignments.

Notes

Read Start Cycle to the First Datum (‘RdStart2Datum’) Field: and Write Start Cycle to the First Datum (‘WrStart2Datum’) Field:

This field sets the number of cycles from the last (if repeated) 'S0 Start' bus cycle to the first RdCEnN for a burst read or to the RdCEnN and AckN for a single word read or to the first AckN for a burst DMA write or to the AckN for a single word write. The time can be from 1 cycle (Start2Datum = b’0000) to 16 cycles (Start2Datum = b’1111). The wait-states (Start2Rd value - 1) are injected onto the “2nd” clock edge after the bus cycle begins, such that the Second,'S1' state is repeated. Encoding information for this field is contained in Table 8.9.

Value	Action
‘15’	16 clock cycles to 1st datum (default).
...	
‘0’	1 clock cycle from Start to 1st datum.
Note: At least 1 clock cycle is always implied by 'S1' state.	

Table 8.9 Start to the first Datum (‘RdStart2Datum’ and 'WrStart2Datum') Field Encoding.

Read Datum to Datum (‘RdDatum2Datum') Field: and Write Datum to Datum (‘WrDatum2Datum') Field:

This field sets the number of cycles between RdCEnN Datum for burst reads or AckN Datum for burst DMA writes. The time can be from 1 (Datum2Datum = b’0000) to 16 cycles (Datum2Datum = b’1111) such that the 'S2' state is repeated. Interleaved delay is between pairs of datum. Encoding information for this field is listed in Table 8.10.

Value	Action
‘15’	16 clock cycles to next burst datum (default
‘2’	3 clock cycles to next burst datum
‘1’	2 clock cycles to next burst datum
‘0’	1 clock cycle to next burst datum

Table 8.10 Datum-to-Datum (RdDatum2Datum, WrDatum2Datum) Field Encoding

Memory MSB Wait-State Register for Bank 7..0 ('MemMSBWaitStateReg(7..0)’)

These are read/write registers. The programming information for this field is located in Figure 8.8 Memory MSB Wait-State Register) and Table 8.11 (Memory MSB Wait-State Register Bit Assignments).

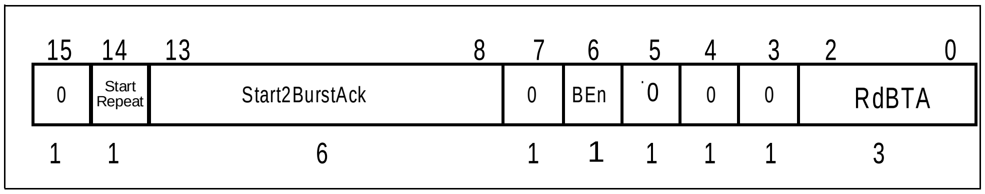


Figure 8.8 Memory MSB Wait-State Register (‘MemMSBWaitStateReg).

Notes

Bit	Assignment
15	Reserved to ‘0’
14	StartRepeat (default is 1)
13:8	Start2BurstAck (default is 0)
7	Reserved to '0'
6	BEn (default is 0)
5:3	Reserved to '0'
2:0	RdBTA (default is 1)

Table 8.11 Memory MSB Wait-State Register ('MemMSBWaitStateReg') Bit Assignments.

Repeat Start Bus Cycle State 0 (‘StartRepeat’) Field

This field controls the number of times the S0 first bus cycle state is repeated until the second 'S1' state is entered. An application is to allow more time for address setup to the chip select on the slowest 600ns PCMCIA cards. Field encoding information for this field is located in Table 8.12.

Value	Action
‘1’	Repeat Start Cycle 1 time (default).
‘0’	Repeat Start Cycle 0 times (typical).

Table 8.12 Repeat Start Bus Cycle State 0 (‘StartRepeat’) Field Encoding.

Start of Read to AckN on Burst Reads (‘Start2BurstAck’) Field

This field sets the number of cycles from the first bus cycle to the AckN for a burst read or to the last AckN for a burst DMA write. The time can be from 0 cycles (Start2BurstAck = b’000000) to 62 cycles (Start2BurstAck = b’111110). This field is only valid for Memory type and not valid for I/O type accesses. Feld encoding information for this field is listed in Table 8.13. Note that if this field is not programmed, burst reads will incur a three clock internal pipeline penalty, thus decreasing system throughput. For assistance in setting this field to its optimal value, refer to the Start2BurstAck chart at the end of this section.

Value	Action
‘63’	No Ack (default).
‘62’...’0’	62 to 0 cycles until Ack (typical is 0).

Table 8.13 First Read to AckN on Burst Reads (‘Start2BurstAck’) Field Encoding.

Byte Enables on Reads (‘BEn’) Field

This field selects whether the $\overline{\text{MemWrEn(3:0)}}$ pins are asserted on reads. $\overline{\text{MemWrEn(3:0)}}$ always returns high at the end of a bus transaction. Field encoding information for this field is listed in Table 8.14.

Notes

Value	Action
‘1’	Allow Byte Enables on reads.
‘0’	Allow Byte Enables on writes only with $\overline{\text{MemWrEn}}$ (default).

Table 8.14 Byte Enables on Reads (‘BEn’) Field Encoding.

Read Cycle Bus Turn-Around (‘RdBTA’) Field

This field sets the minimum number of idle data bus cycles after a read. This field must be set to at least '1' because of the possibility of a read followed by a write. On a read followed by a read, the number of idle cycles is from the pertinent read enable de-asserting to the next pertinent read enable (Memory, I/O, or DRAM) asserting. Field encoding information is listed in Table 8.15

Value	Action
‘111’	Minimum of 7 clocks (may increase in future products)
‘110’	6 clocks
‘101’	5 clocks
'100'	4 clocks
‘011’	3 clocks
‘010’	2 clocks
‘001’	1 clock (default)
‘000’	Reserved

Table 8.15 Bus Turn-Around (‘BTA’) Field Encoding.

Formulas for Calculating Memory Controller Start2BurstAck Field Valuei

Non-interleave read:
# of read cycles
= Sysale + repeat+(Rdstart2Data +1) +(RdData2Data +1) * (# of Data -1)

Interleave read:
# of read cycles
= 1 + repeat + (RdStart2Data +2) + (RdData2Data +2) * $\frac{\text{\# of Data}}{2}$ - 1

Start2BurstAck should be programmed no less than # of read clock cycles on sysbus - 5

Non-interleave read:
32-bit port:
Start2burstAck Š [repeat +2 + RdStart2Data +(RdData2Data +1) * 3] - 5

16-bit port:
Start2burstAck Š [repeat +2 + RdStart2Data +(RdData2Data +1) * 7] - 5

8-bit port:
Start2burstAck Š[repeat +2 +RdStart2Data +(RdData2Data +1) * 15] - 5

Interleave read:
always 32-bit port:

Notes

Start2burstAck Š [repeat +3 + RdStart2Data + (RdData2Data +2)] - 5
Non-interleave read:

Port Size	Start Repeat	Read Start2Data	Read Data2Data	Start2BurstAck
32 bit # of data = 4	0	0	0	0
	0	1	0	1
	0	1	1	4
	0	1	2	7
	1	2	2	9
	1	2	3	12
	1	4	5	20
16 bit # of data = 8	0	0	0	4
	0	1	0	5
	0	1	1	12
	0	1	2	19
	1	2	2	21
	1	2	3	28
	1	4	5	44
8 bit # of data = 16	0	0	0	12
	0	1	0	13
	0	1	1	28
	0	1	2	43
	1	2	2	45
	1	2	3	60
	1	16	5	97

Interleave read:

Port Size	Start Repeat	Read Start2Data	Read Data2Data	Start2BurstAck
64 bit # of data = 4	0	0	0	0
	0	0	1	1
	0	0	2	2
	0	1	0	1
	0	1	1	2
	1	1	2	4
	1	2	0	3
	1	2	1	4
	1	2	2	5
	1	3	3	7

Notes

Memory Controller Timing Diagrams

This section includes a number of timing diagrams that are applicable to RC36100 RISController memory transactions. AC parameter values are contained in the "RC36100 RISController Data Sheet."

Read Transactions

The bus interface timing for read transactions is described in the following section. The internal bus interface to CPU core timing for reads is described in Chapter 7, "Bus Interface Unit Controller".

Basic 1-Datum Read with 0 Wait-States

Figure 8.9 illustrates a basic Memory-Type Memory Controller read transaction. Each transaction begins with both SysALEn and SysBurstFrame asserting. At this time, SysRd and the appropriate MemCS() assert (if they are not already in this state, as the result of a previous transaction) to indicate the read transaction and which memory bank is being used. After the completion of this initial 'Start' cycle stage, the Data Sampling stage is begun. In this second stage, the appropriate Memory Read Enable signal, either MemRdEnEven or MemRdEnOdd, will assert to allow the external memory bank to turn on and begin driving data back to the RISController.

Since Figure 8.10 is for a single data read, this Data Sampling Stage is the last Data for this transaction and thus SysBurstFrame is de-asserted. To end a Data Sampling Stage, SysDataRdy asserts on clock cycles where data is expected. After the last Data is sampled, note that the signals, SysRd and MemCS may not necessarily de-assert as the next bus transaction may already be starting. The internal signal, AckN, that is associated with R3051-family read cycles is generated automatically for single word reads (and writes).

Notes

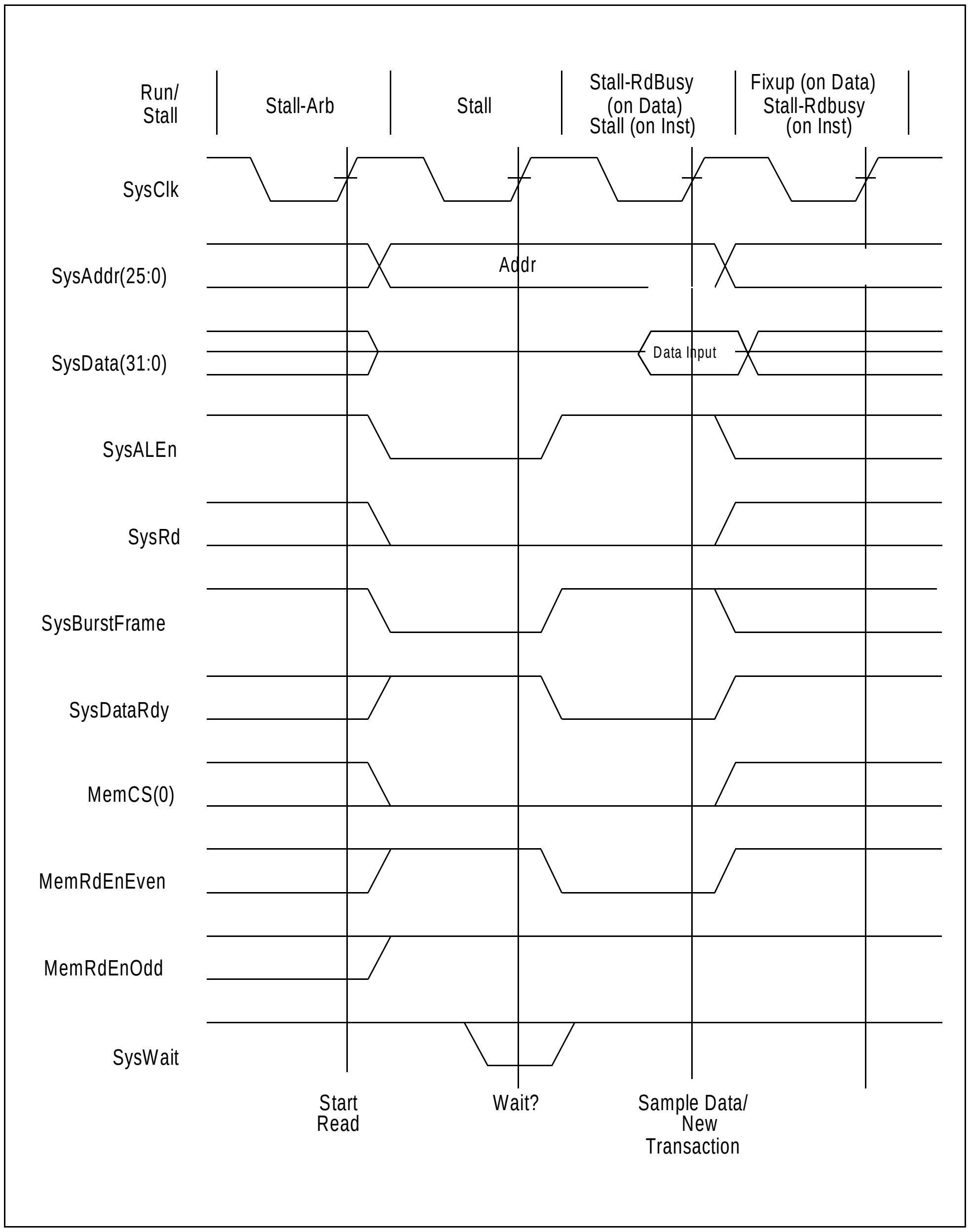


Figure 8.9 1-Datum Read with 0 Wait-States.

1-Datum Read with 0 Wait-States using Odd Chip Select

Figure 8.10 illustrates a basic Memory-Type Memory Controller read transaction, except that the access is to an Odd Memory Chip Select, MemCS(7,5,3,1) instead of an even one. Because an odd MemCS is asserted, MemRdEnOdd is asserted during the Data Sampling Stage instead of its even counterpart.

Notes

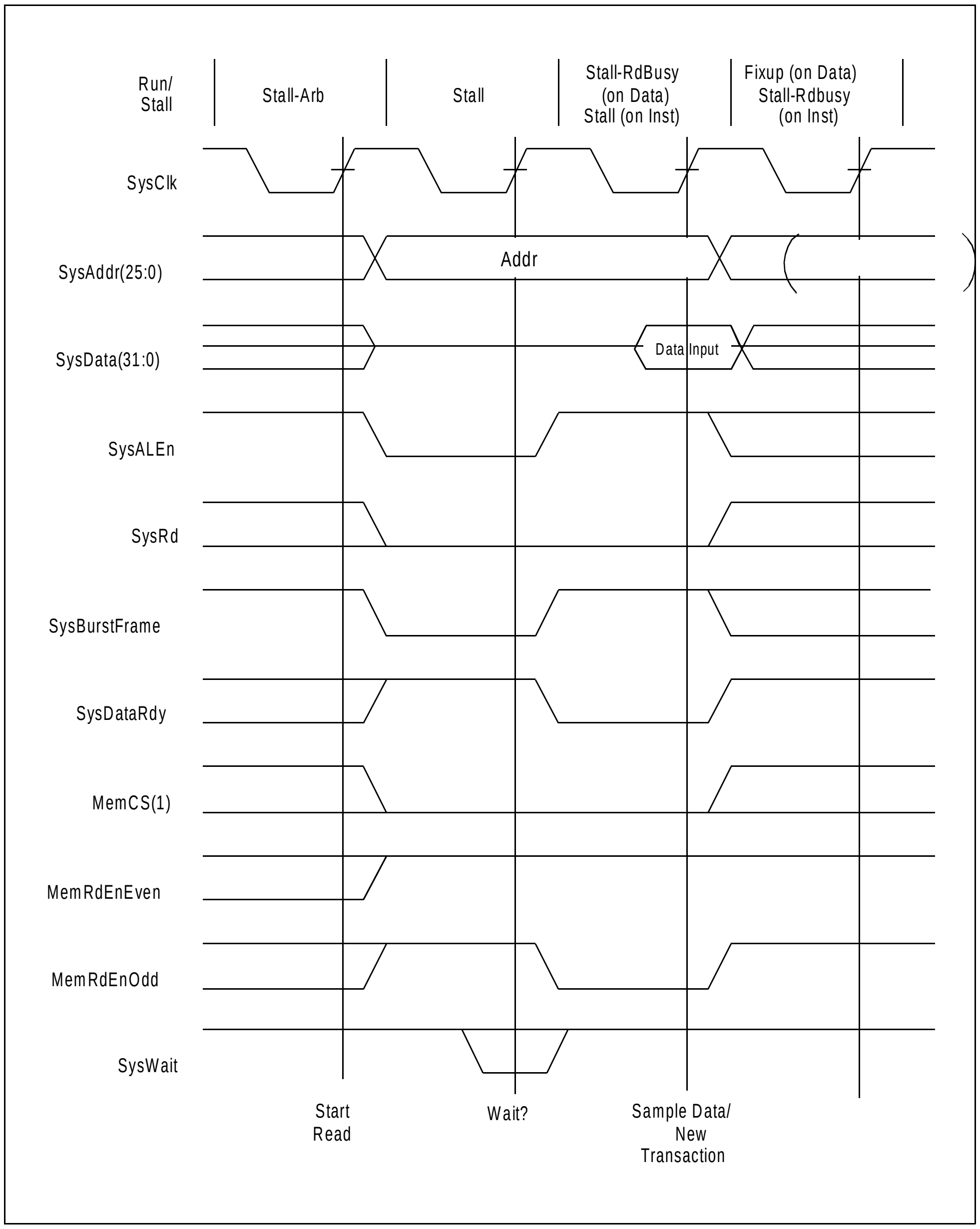


Figure 8.10 1-Datum Read with 0 Wait-States Using an Odd Chip Select.

Read with Wait-State using Start Repeat Field

Figure 8.11 illustrates a basic Memory-Type Memory Controller read where 1 wait-state has been added by repeating the Start Cycle. This special effect is programmed into the Wait-State Generator using the Start Repeat Field in the MemLSBWaitStateReg() Register. When the Start Cycle repeats, the Data Sampling Stage is delayed and the assertion of the Memory Read Enable strobe, MemRdEnEven or MemRdEnOdd is delayed. This effect is useful for very slow memories or memories that require significant address setup before the chip is selected. An example is the 600ns access time mode of the PCMCIA memory protocol. The Start repeat Field affects both reads and writes.

Notes

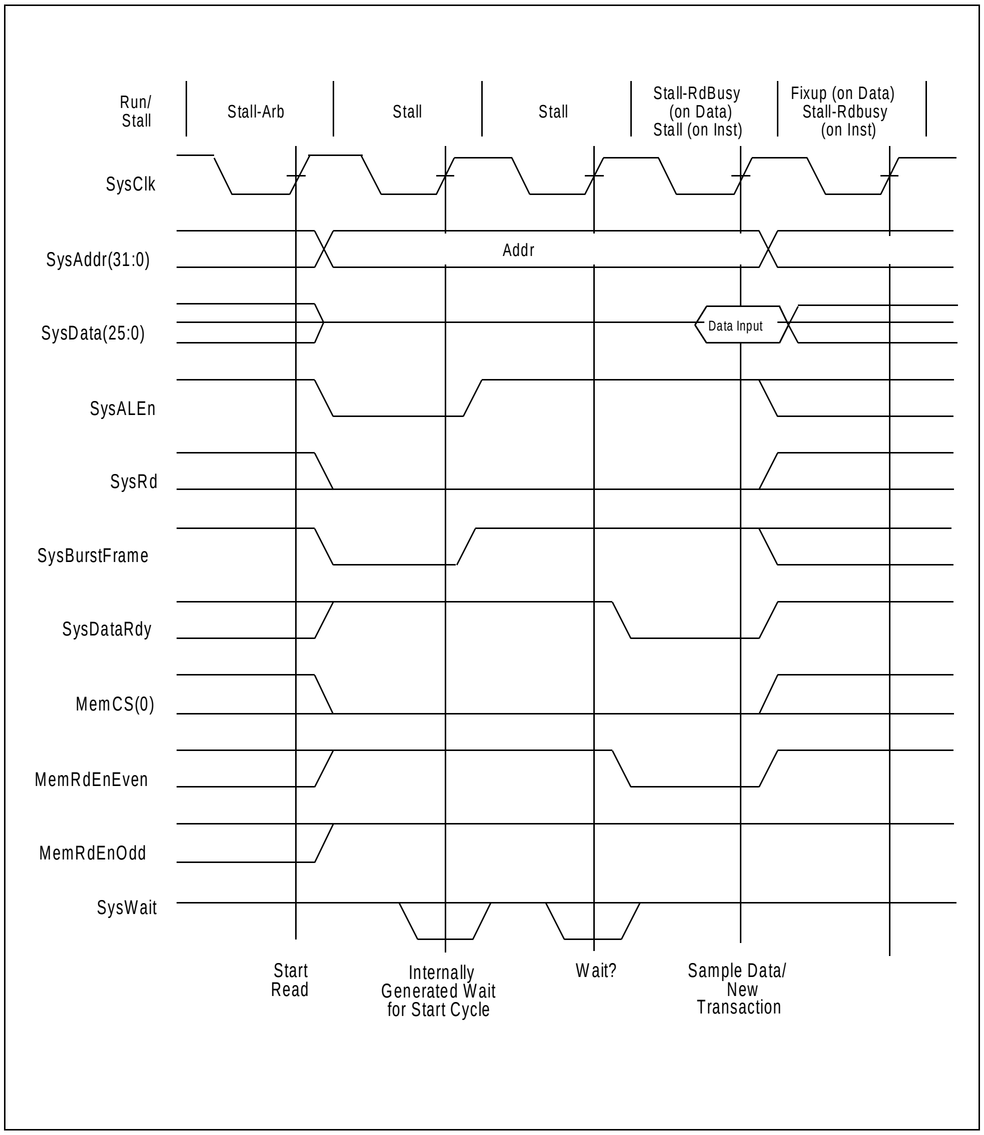


Figure 8.11 1-Datum Read with 1 Wait-State using StartRepeat Field.

Read with Wait-State using RdStart2Datum Field

Figure 8.12 illustrates a basic Memory-Type Memory Controller read where 1 wait-state is added using the RdStart2Datum Field of the MemMSBWaitStateReg() Register. Any number from 0 to 15 internal wait-states may be added using the RdStart2Datum Field. With this field, the Memory Read Enable strobe, either MemRdEnEven or MemRdEnOdd is asserted as normal, but then wait-states are added where SysDataRdy is not asserted until the RdStart2Datum Field has finished counting. When SysDataRdy is asserted, then the Data from the external Memory Bank is sampled into the RISController.

Notes

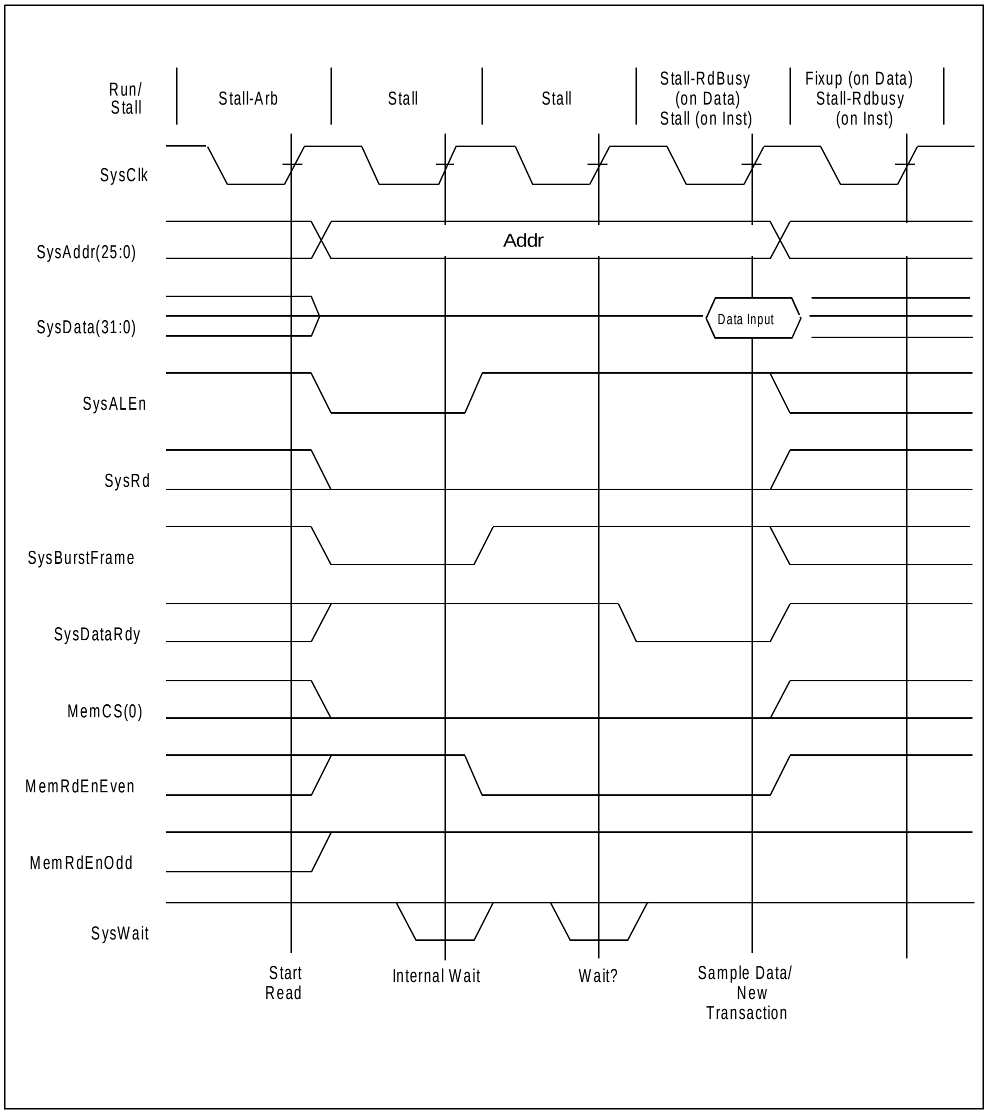


Figure 8.12 Read with Wait-State using RdStart2Datum Field.

Read with Wait-State using SysWait

Figure 8.13 illustrates a basic Memory-Type Memory Controller read where 1 wait-state is added using the external signal pin, SysWait. SysWait is not expected to be used for conventional memories, since it is easier to program the Wait-State Generator to produce wait-states. However, SysWait can be useful for Dual-Port memory and off-card memories where there may be an indeterminate amount of time before the access can begin. Since SysWait is sampled a clock ahead of when it is used, its effect is seen two clocks later than when it is asserted. If SysWait is asserted when SysDataRdy is asserted then an additional Data Sampling clock cycle is repeated with SysDataRdy remaining low. Thus external logic analyzers or other debug equipment may want to gate SysDataRdy with SysWait in order to decode valid Data samples.

Notes

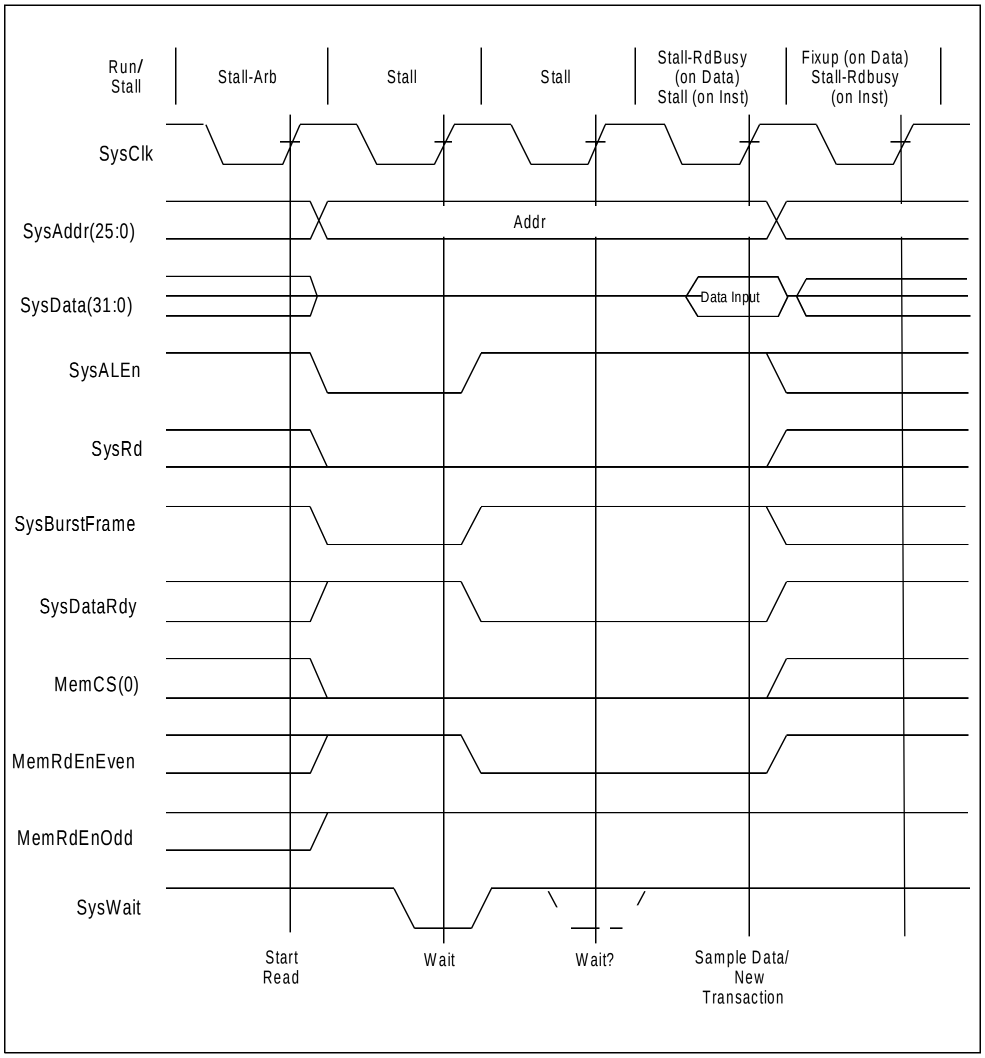


Figure 8.13 Read with Wait-State using SysWait.

4-Word Burst Read with 0 Wait-States

Figure 8.14 illustrates a 4-word Burst Read when using the Memory-Type Memory Controller. This example can also be generalized to demonstrate any multi-datum read generated from the CPU core which includes double, triple, quad, and 16-byte reads using an 8-bit port width, and double, and octi-halfword reads using a 16-bit port width. Note that the DMA Controller can potentially do any length from 1 to 16 datum.

As shown in Figure 8.14, $\overline{\text{SysFrameBurst}}$ can be used to determine which is the last datum to be sampled. After the first datum is sampled, the Wait-State Generator uses the RdDatum2Datum field in the MemMSBWaitStateReg() Register in order to determine the number of wait-states to generate. Figure 8.15 gives an example of inserting wait-states between later data elements. By programming different values into the RdStart2Datum and the RdDatum2Datum Fields, a burst read can be throttled to, for example, give a longer access time to the first datum and shorter access time for any subsequent datum.

To obtain the maximum optimization of a 4-word burst read (4 32-bit datum, 8 16-bit datum or 16 8-bit datum) the Start2BurstAck Field of the MemMSBWaitStateReg() Register must be programmed. Programming this field to 3 clock cycles before the last datum is sampled places the internal R3051-family like signal, AckN, such that the CPU pipeline can be restarted early. Systems that have an indeterminate number of external SysWait wait-states must program this field to give an internal AckN. In such cases, like single word reads, AckN is generated automatically on the last datum sample.

Notes

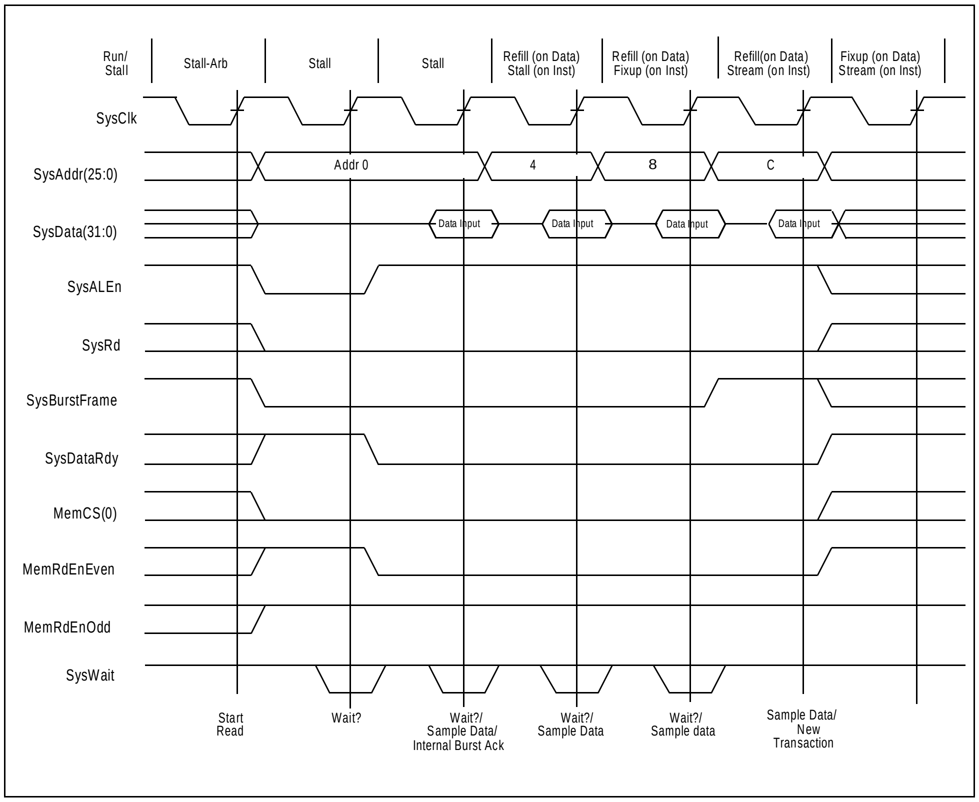


Figure 8.14 4-Word Burst Read with 0 Wait-States.

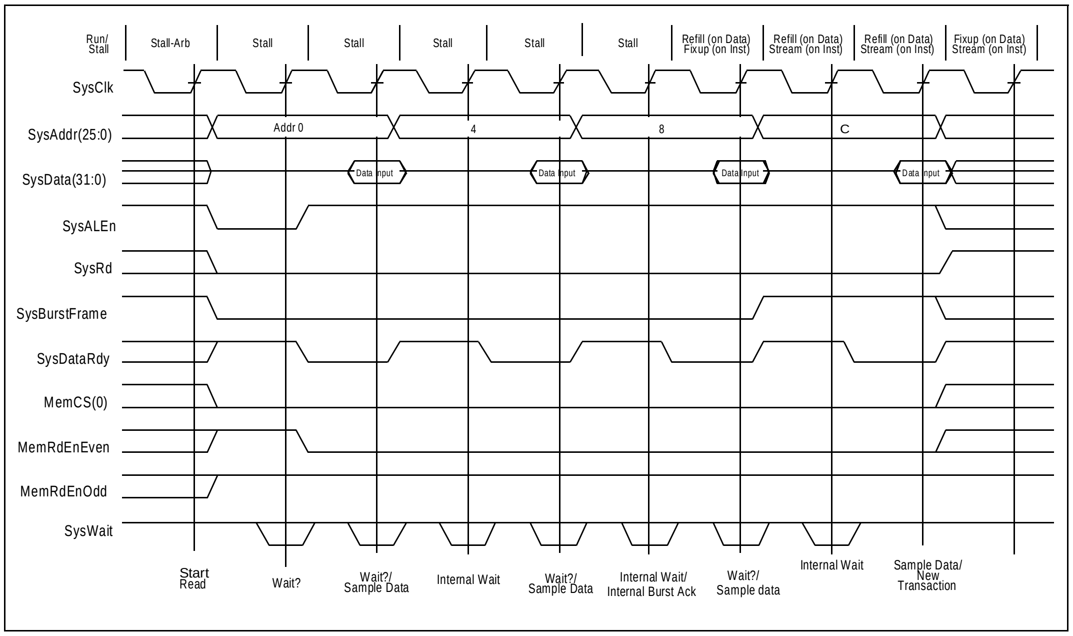


Figure 8.15 4-Word Burst Read with Wait-States using RdDatum2Datum Field.

Notes

Basic 16-bit PCMCIA-style Memory Read with Zero Wait-States

Figure 8.16 illustrates a basic 16-bit PCMCIA-Style Memory Controller read transaction. Each transaction begins with both SysALEn and SysBurstFrame asserting. At this time, SysRd asserts (if it is not in this state, as the result of the previous transaction). Assuming there are no internally programmed StartRepeat wait-states, on the next clock cycle, SysBurstFrame de-asserts and the MemCS() pair asserts. Note that on PCMCIA transactions the MemCS() pair is asserted, according to which bytes are enabled and valid. Thus if the even byte is valid, then the even MemCS() will assert. If the odd byte is valid, then the odd MemCS() will assert. If both bytes are valid, then both MemCS()s in the pair assert. Assuming there are no internally programmed RdStart2Datum wait-states, SysDataRdy asserts to indicate that the data from the memory device is being sampled into the RISController. On the next clock--the final clock of the transaction--the MemCS() pair de-asserts and, simultaneously, the next transaction may begin.

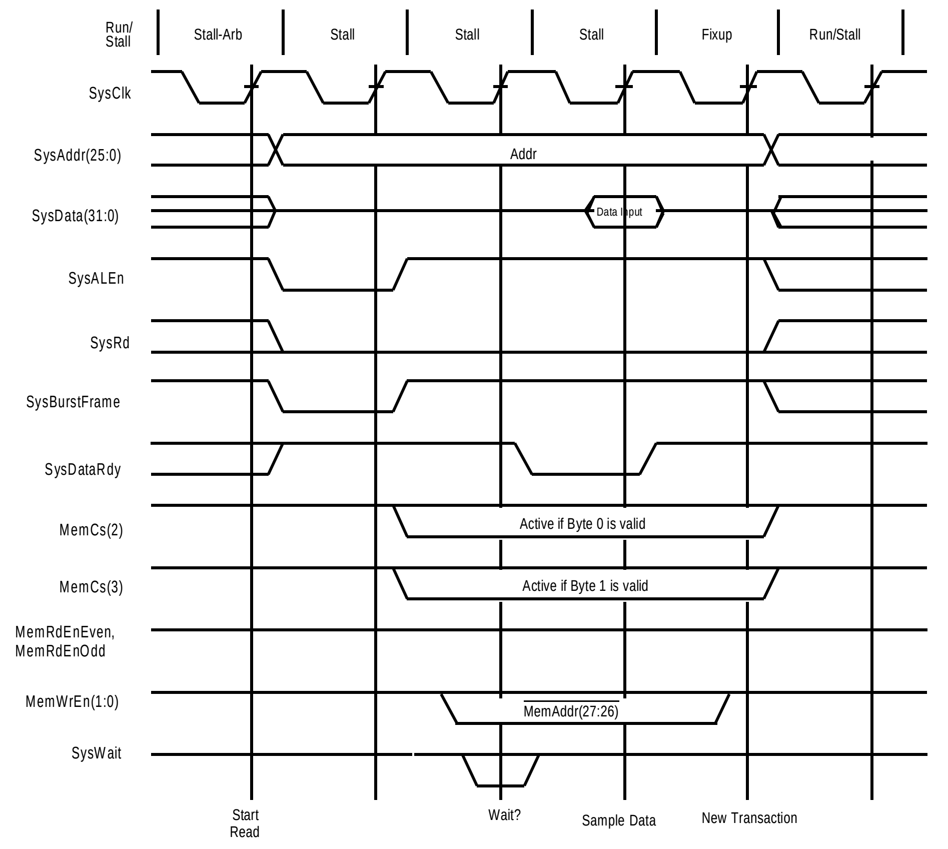


Figure 8.16 PCMCIA-Style Memory Read with 0 Wait-States

Write Transactions

The bus interface aspect of write transactions is described in the following section. The internal bus interface to CPU core aspect of writes is described in Chapter 7, “Bus Interface Unit Controller”.

Notes

Single Datum Write

Figure 8.17 illustrates a basic Memory-Type Memory Controller write transaction. Each transaction begins with both SysALEn and SysBurstFrame asserting. At this time, SysWr the appropriate MemCS assert (if they are not already in this state) to indicate the write transaction and the memory bank being used. After completing this Start cycle stage, the Data Driving stage begins. In this second stage, MemWrEn(3:0) (which acts as write byte enable strobes) assert. In general, from 1 to 4 of the MemWrEn(3:0) signals will be asserted.

Since Figure 8.17 is for a single datum write, this Data Driving Stage is the last Datum for this transaction and thus SysBurstFrame is de-asserted. To end a Data Driving Stage, SysDataRdy asserts on clock cycles where data is expected to be latched with the trailing de-asserting edges of MemWrEn(3:0). After the last Data is driven, note that signals such as SysWr and MemCS may not necessarily de-assert, as the next bus transaction may already be starting. The internal signal, AckN, associated with R3051-family write cycles, is generated automatically for single word writes (and reads).

Note: This manual does not illustrate a Memory-type write transaction to an Odd Memory Chip Select. However, the only difference is that during the Data Driving Stage an odd MemCS is asserted instead of its even counterpart.

Notes

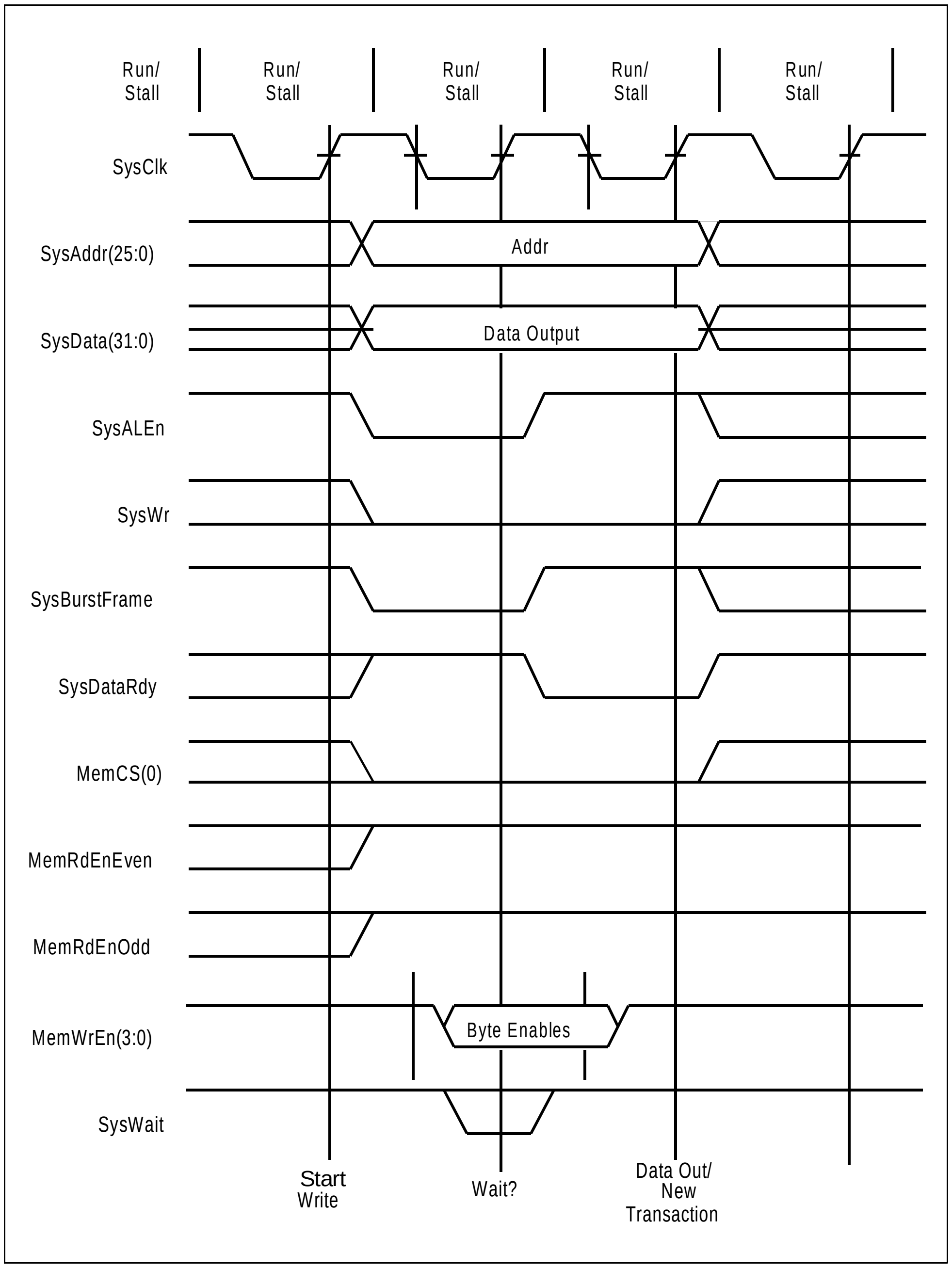


Figure 8.17 1-Datum Write with 0 Wait-States

1-Datum Write with 0 Wait-States using FCT245-Type Field

Figure 8.18 illustrates a basic write using the FCT245-Type Field. Either the even or the odd read enable, $\overline{\text{MemRdEnEven}}$ or $\overline{\text{MemRdEnOdd}}$ will assert on even or odd writes such that a FCT245 transceiver output enable can be connected.

Notes

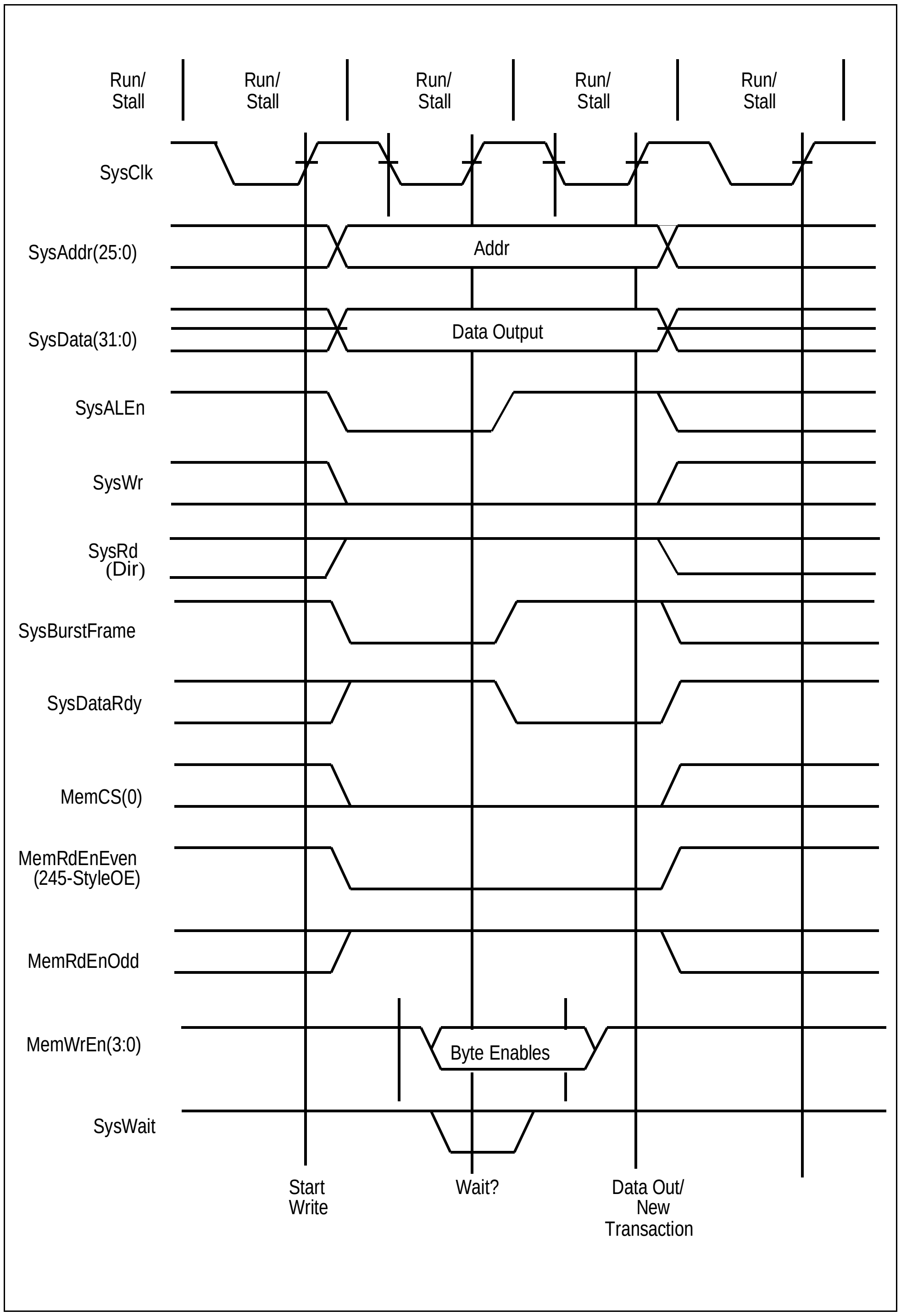


Figure 8.18 1-Datum Write with 0 Wait-States using FCT245-Type Field.

1-Datum Write with Wait-State using StartRepeat Field

Figure 8.19 illustrates a basic Memory-Type Memory Controller write where 1 wait-state has been added by repeating the Start Cycle. This special effect is programmed into the Wait-State Generator using the Start Repeat Field in the MemLSBWaitStateReg() Register. When the Start Cycle repeats, the Data Driving Stage is delayed and the assertion of the Memory Write Enable strobes, MemWrEn(3:0), are delayed. This effect is useful for very slow memories or memories that require significant address setup before the chip is selected. For example, with the 600ns access time mode of the PCMCIA memory protocol, the Start repeat Field affects both reads and writes.

Notes

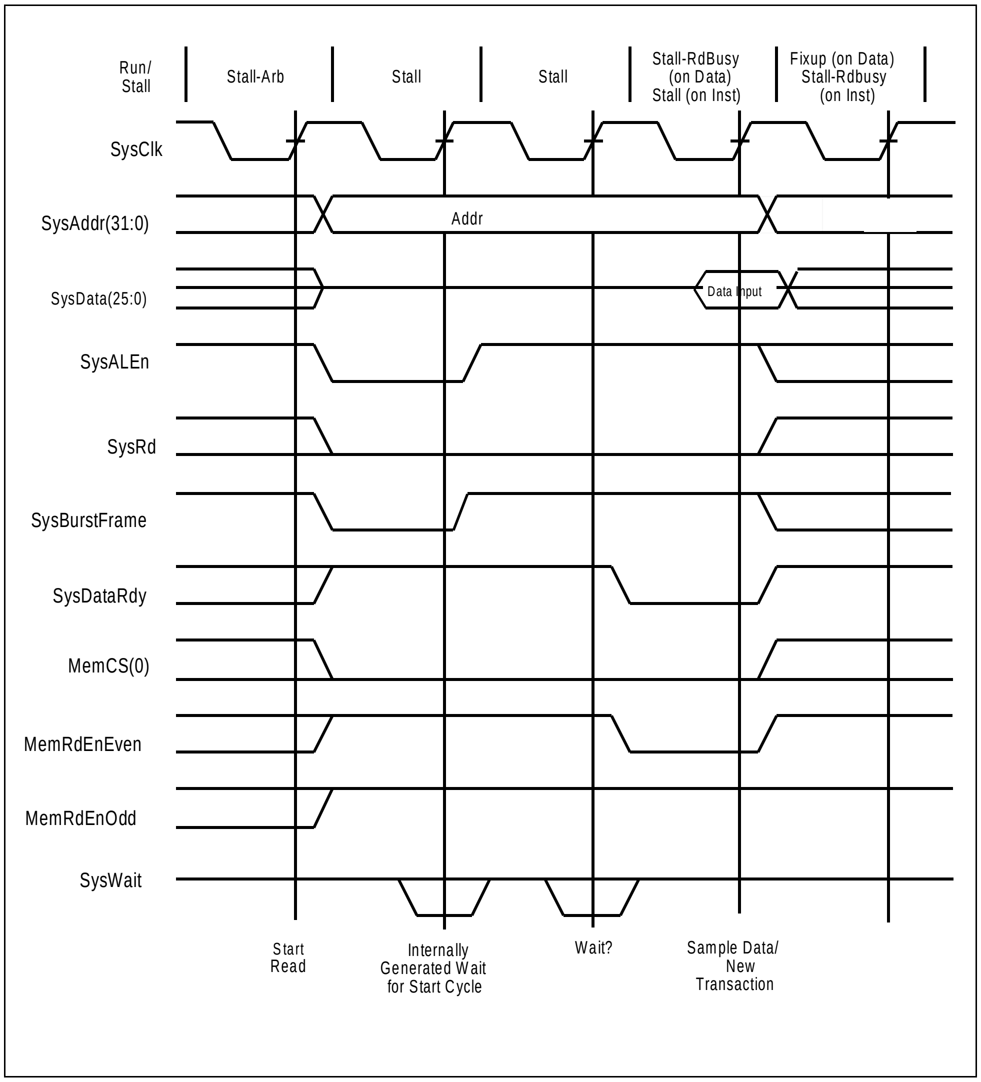


Figure 8.19 1-Datum Write with Wait-State using StartRepeat Field.

1-Datum Write with Wait-State using WrStart2Datum Field

Figure 8.20 illustrates a basic Memory-Type Memory Controller write where 1 wait-state is added using the WrStart2Datum Field of the MemMSBWaitStateReg() Register. Any number from 0 to 15 internal wait-states may be added using the WrStart2Datum Field. With this field, the Memory Write Enable strobe (either MemWrEnEven or MemWrEnOdd) as well as the write byte enables (MemWrEn(3:0)) are asserted as normal. However, wait-states are added (where SysDataRdy is not asserted) until the WrStart2Datum Field has finished counting. When SysDataRdy is asserted, then the Data from the external Memory Bank is sampled by external memory.

Notes

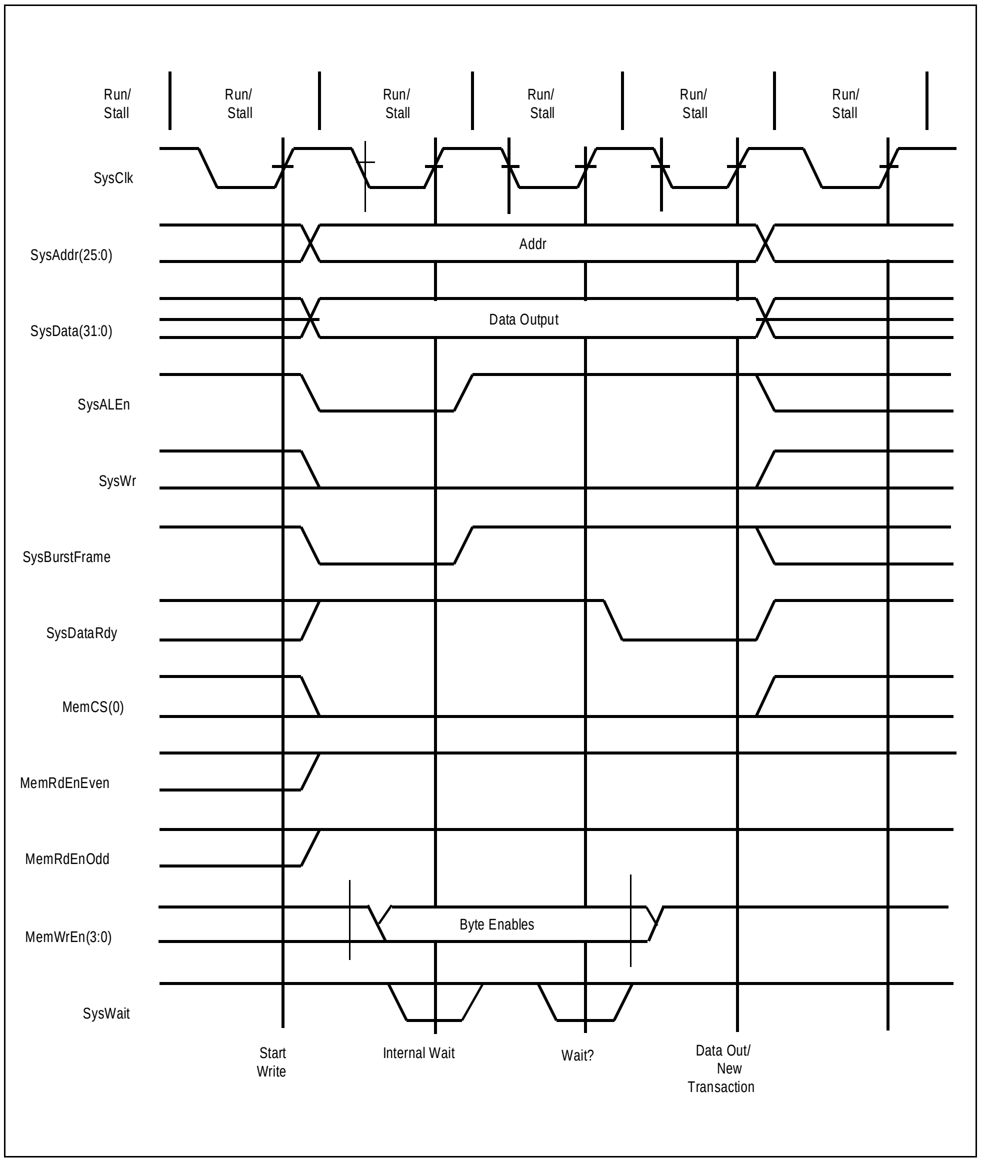


Figure 8.20 1-Datum Write with Wait-State using WrStart2Datum Field.

1-Datum Write with Wait-State using SysWait

Figure 8.21 illustrates a basic Memory-Type Memory Controller write where 1 wait-state is added using the external signal pin, SysWait. SysWait is not expected to be used for conventional memories, since it is easier to program the Wait-State Generator to produce internal wait-states. However, SysWait can be useful for Dual-Port memory and off-card memories where there may be an indeterminate amount of time before the access can begin. Since SysWait is sampled a clock ahead of when it is used, its effect is seen two clocks later than when it is asserted. If SysWait is asserted when SysDataRdy is asserted then an additional Data Sampling clock cycle is repeated with SysDataRdy remaining low. Thus external logic analyzers or other debug equipment may want to gate SysDataRdy with SysWait in order to decode valid Data samples.

Notes

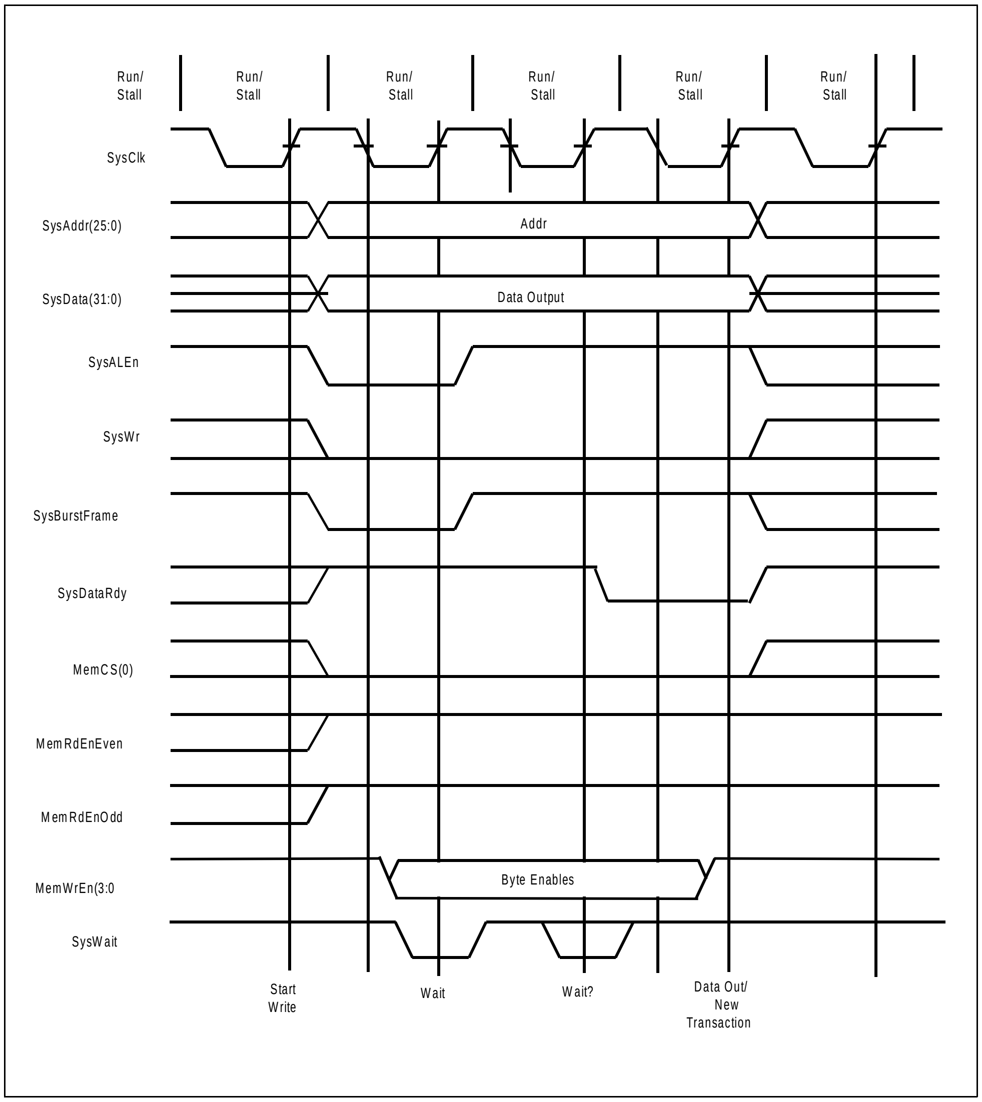


Figure 8.21 1-Datum Write with Wait-State using SysWait.

Multi-Datum Burst Write

Figure 8.22 illustrates a 2-datum Burst Write when using the Memory-Type Memory Controller. This example can also be generalized to demonstrate any multi-datum write generated from the CPU core which includes double, triple, and quad byte writes using an 8-bit port width, and double half-word writes using a 16-bit port width. Note that although the CPU core will not generate bursts in the 32-bit width, the internal and external DMA Controller channels can do so, and can potentially do any length.

In Figure 8.21, SysBurstFrame can be used to determine which is the last datum to be sampled. At the end of each datum, MemWrEn(3:0) de-asserts for 1 clock cycle. They re-assert if more datum needs to be processed. After the first datum is sampled, then the Wait-State Generator uses the WrDatum2Datum Field in the MemMSBWaitStateReg() Register to determine the number of wait-states to generate.

Most conventional SRAMs need to use WrDatum2Datum wait-states to de-assert MemWrEn(3:0) at the end of each word written. The zero wait-state case is useful for certain types of FIFOs.

Notes

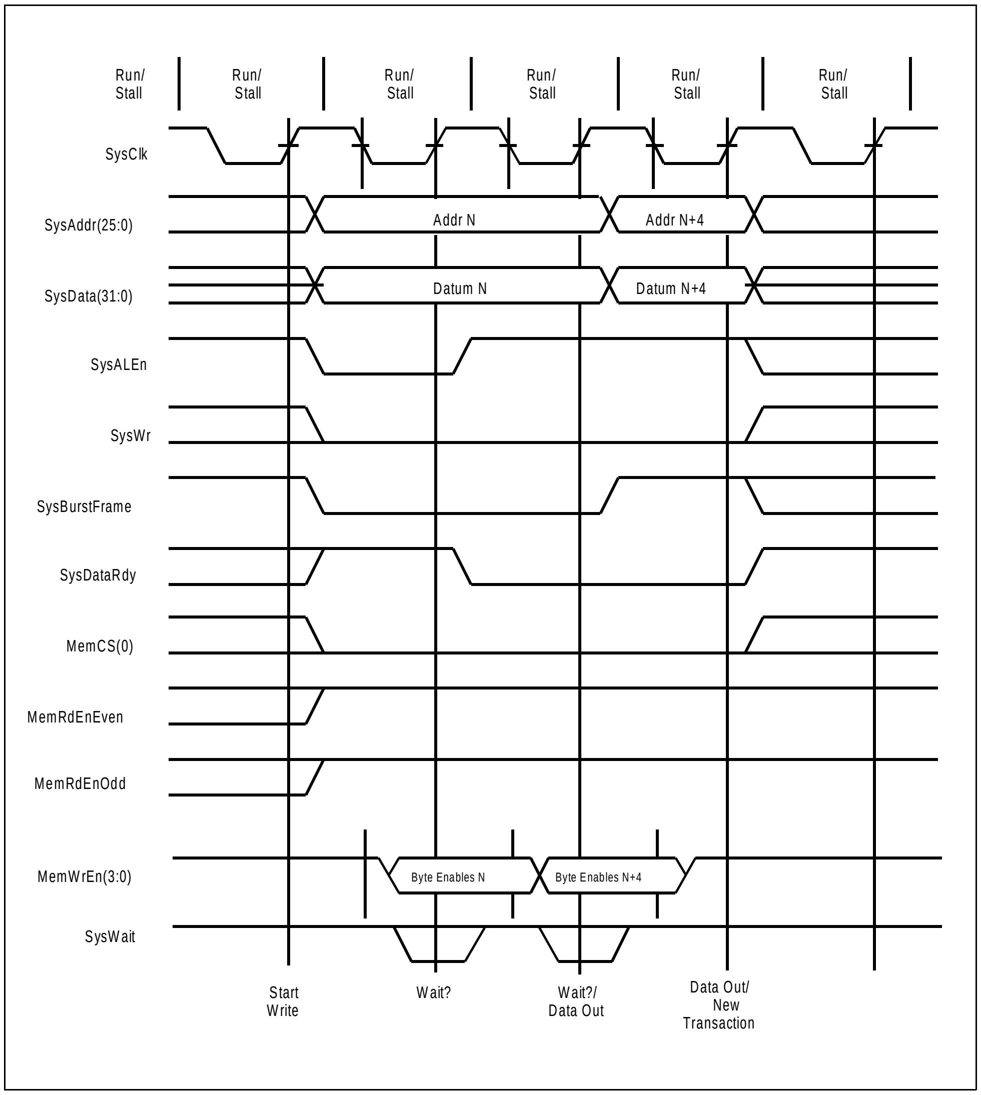


Figure 8.22 Multi-Datum Burst Write.

Multi-Datum Burst Write using Wait-State with WrDatum2Datum

Figure 8.23 gives an example of inserting wait-states between later data elements. By programming different values into the WrStart2Datum and the WrDatum2Datum Fields, a burst write can be throttled to for instance give a longer access time to the first datum and shorter access time for any subsequent datum.

Like single word writes, AckN is generated automatically on the last datum sample.

Notes

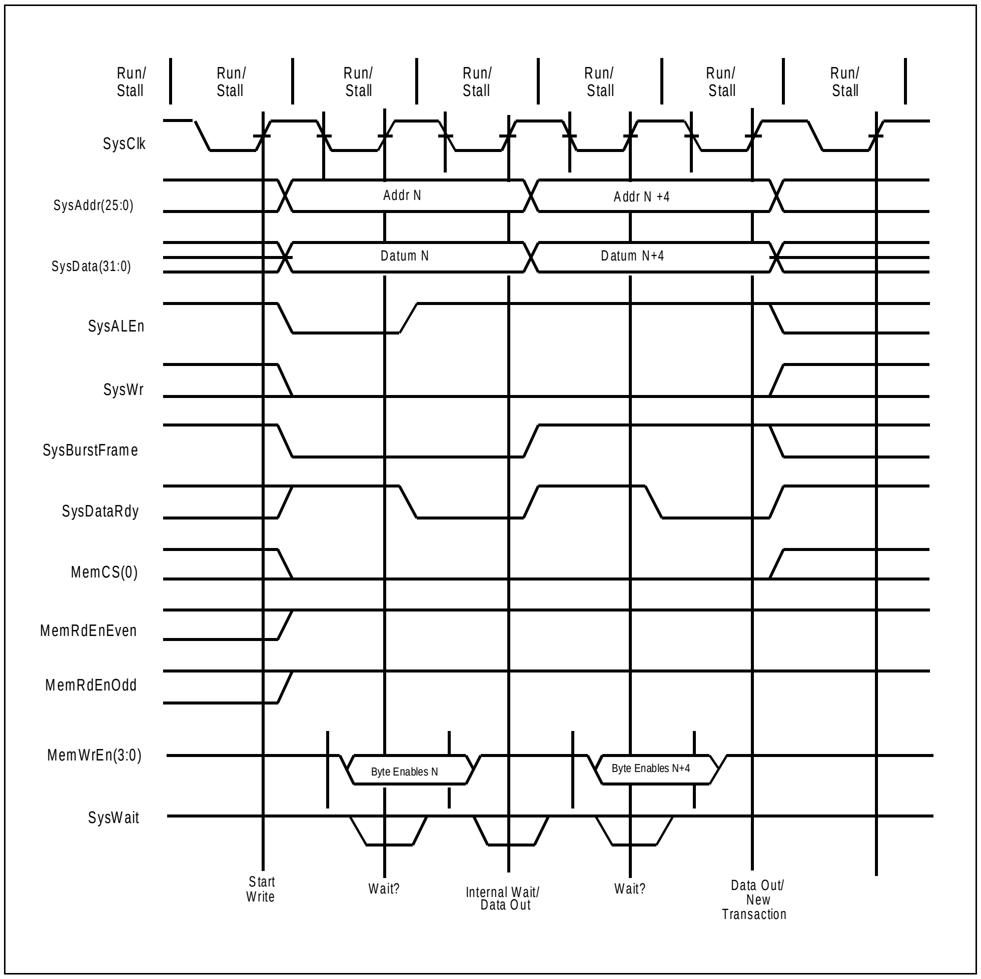


Figure 8.23 Multi-Datum Burst Write using Wait-State with WrDatum2Datum.

Basic PCMCIA-Type Memory Write with 0 Wait-States

Figure 8.24 illustrates a basic PCMCIA-Type Memory Controller write transaction. Each transaction begins with both SysALEn and SysBurstFrame asserting. At this time, SysWr asserts if hasn't done so already (from the previous transaction), and it is guaranteed that MemCS() will be in its de-asserted state. Assuming that there are no internally programmed StartRepeat wait-states, on the next clock cycle, SysBurstFrame de-asserts.

On the third cycle MemCS() asserts. On PCMCIA transactions the MemCS() pair is asserted according to which bytes are enabled and valid. If the even byte is valid, the even MemCS() will assert, while if the odd byte is valid, then the odd MemCS() will assert. If both bytes are valid, then both MemCS() signals in the pair will assert.

Assuming there are no internally programmed RdStart2Datum wait-states, SysDataRdy asserts to indicate that the data from the RISController is ready to be latched into the memory device. On the 4th clock cycle, MemCS() de-asserts, providing a means for the write data from the RISController to be latched into the memory device. On the next clock, the next transaction may begin.

During a PCMCIA-Type transaction, there is 1 clock of address setup time before MemCS() asserts. All signals are setup before MemCS(), which is being used as the write strobe, de-asserts.

Notes

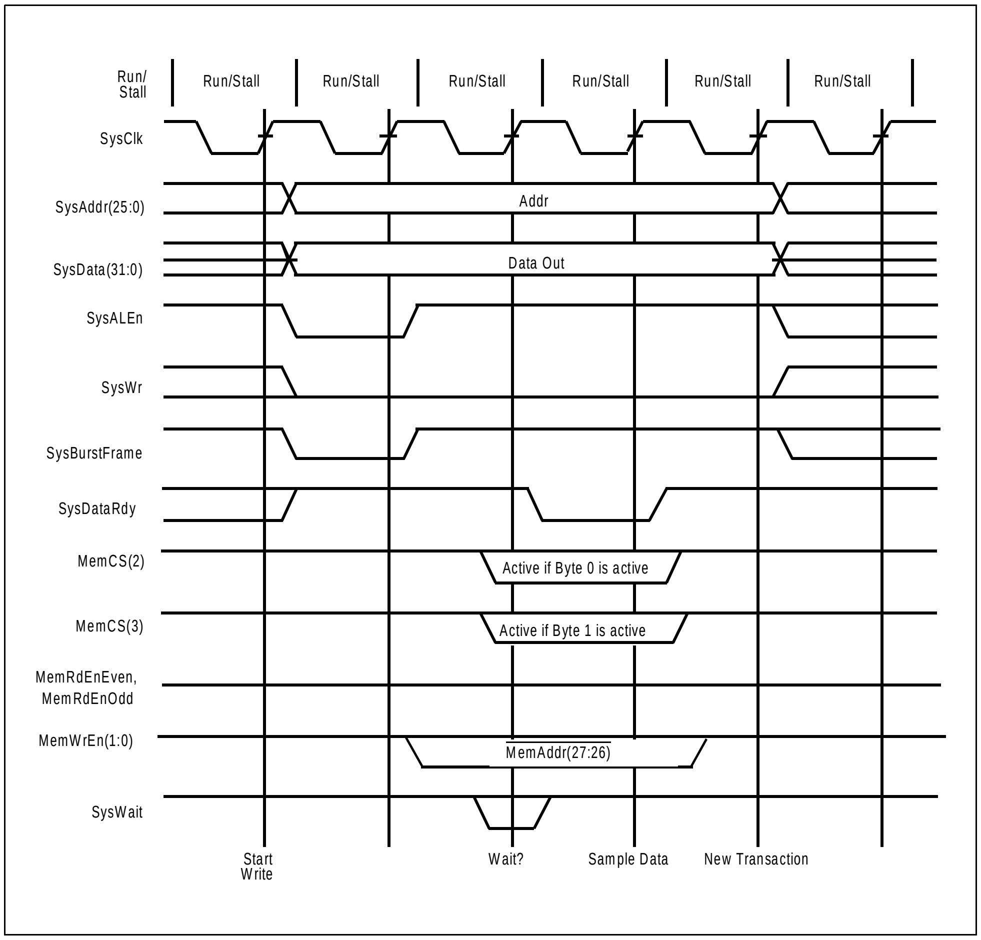


Figure 8.24 PCMCIA-Style Memory Write with 0 Wait-States

Interleaved-Type Transactions

The RC36100 RISController's Memory Controller has the capability to interleave memory transactions such that 64-bits of memory are accessed at a time and then funneled into or out of the CPU in two 32-bit chunks. Thus on a burst read, the CPU begins a pair of 32-bit accesses at the same time. The first word is read in as normal, while the second word is externally latched. Then while the second word is read, the third and forth words begin their accesses. When large memory arrays are used, interleaved systems speed up the overall transaction time of burst reads and do not add cost to the system, since transceivers are typically needed to isolate the multiple memory banks anyway. The RC36100 supports a variety of different data transceiver options, as shown here.

Interleaved Read using FCT260-Type Field

Figure 8.25 illustrates an interleaved 4-word burst read using the FCT260-Type. The odd Read Enable, MemRdEnOdd is used as the Path Select and the Odd Latch Enable. MemRdEnEven changes its functionality in that it asserts for both the even and odd Data Sampling periods. As with the non-interleaved types, various throttled wait-state options are available via the internal Wait-State Generator including optimal Burst Ack placement. It is implied that the odd words are always returned 1 clock after the even words.

Figure 8.26 shows a single datum access to an “even” bank of an interleaved memory system using 'FCT260 transceivers. Note that the timing of this access is identical with the timing of the first word of a 4-word access.

Figure 8.27 shows the analogous access to the “odd” bank of an interleaved memory system, using 'FCT260 transceivers. In this figure, the timing is identical with the timing of the access of the *second* word of a 4-word access; however, the first word is not actually returned to the CPU.

Notes

Due to a limitation on the number of transceiver control signals, there is a performance difference between even and odd single word accesses. However, note that this should not affect system performance in the following areas:

- *single word accesses will occur for uncached loads, and uncached instructions. These are typically not time critical programs or data.*
- *for cached data, these accesses will only occur if the data block refill option is selected to "one word" rather than "four words." However, this selection is unlikely for an interleaved memory, which dramatically mitigates the time required for 4-word accesses, and thus is expected to use 4-word accesses on data cache misses. For more information, refer to the DBlockRefill ('DBR') option in the Coprocessor0 Cache Configuration documentation in Chapter 5.*
- *cached instruction misses are always satisfied using 4-word read accesses.*

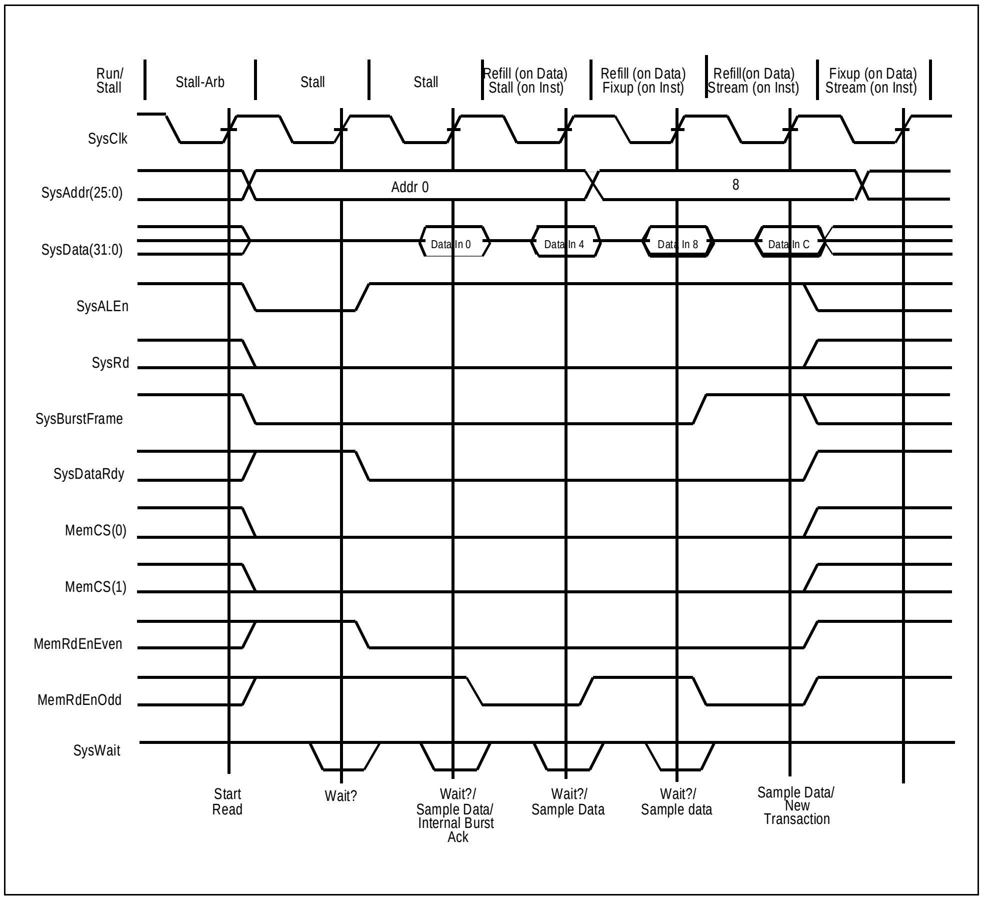


Figure 8.25 Interleaved Read using FCT260-Type Field.

Notes

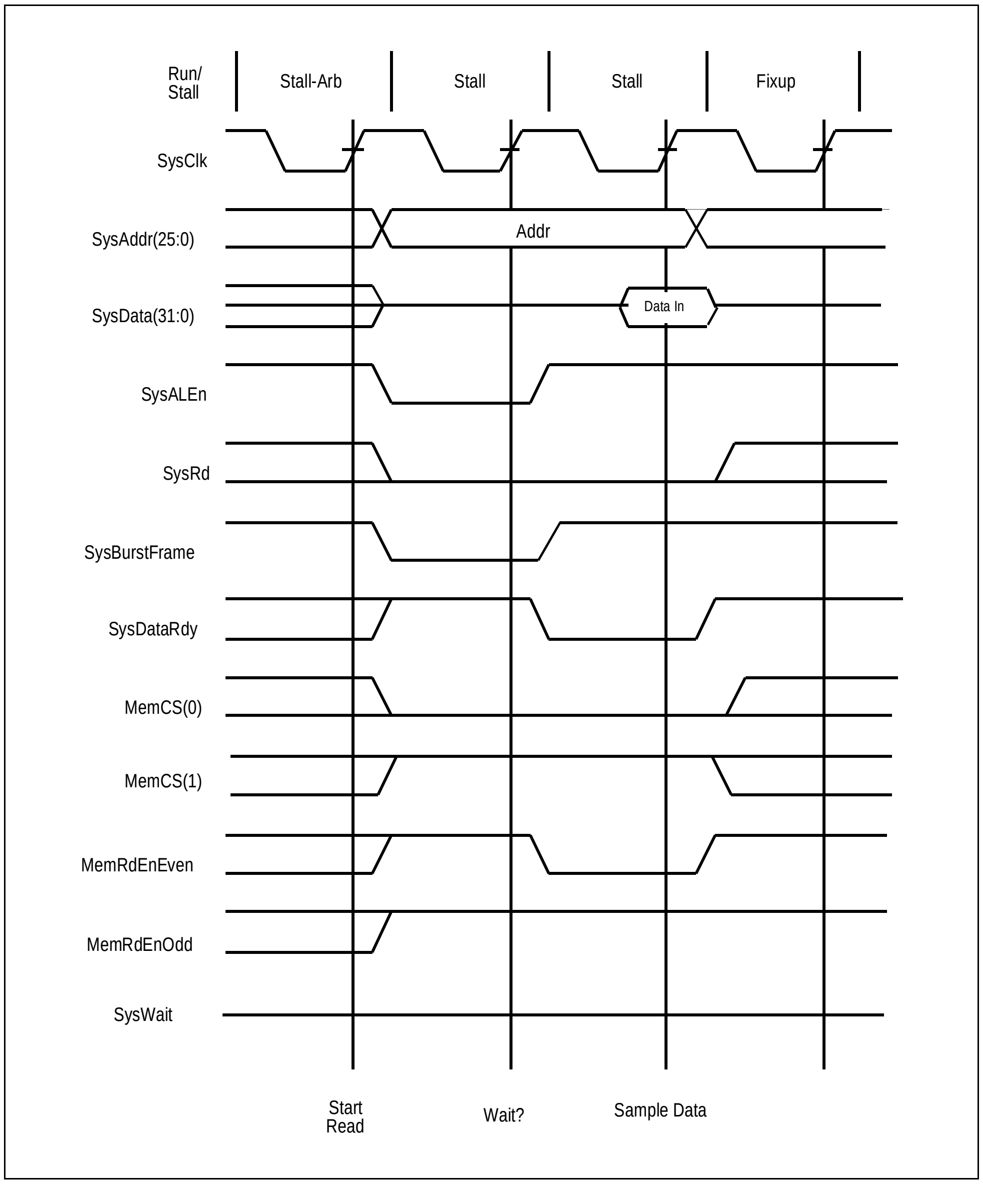


Figure 8.26 Interleaved "Even" Read of FCT260-Type Memory

Notes

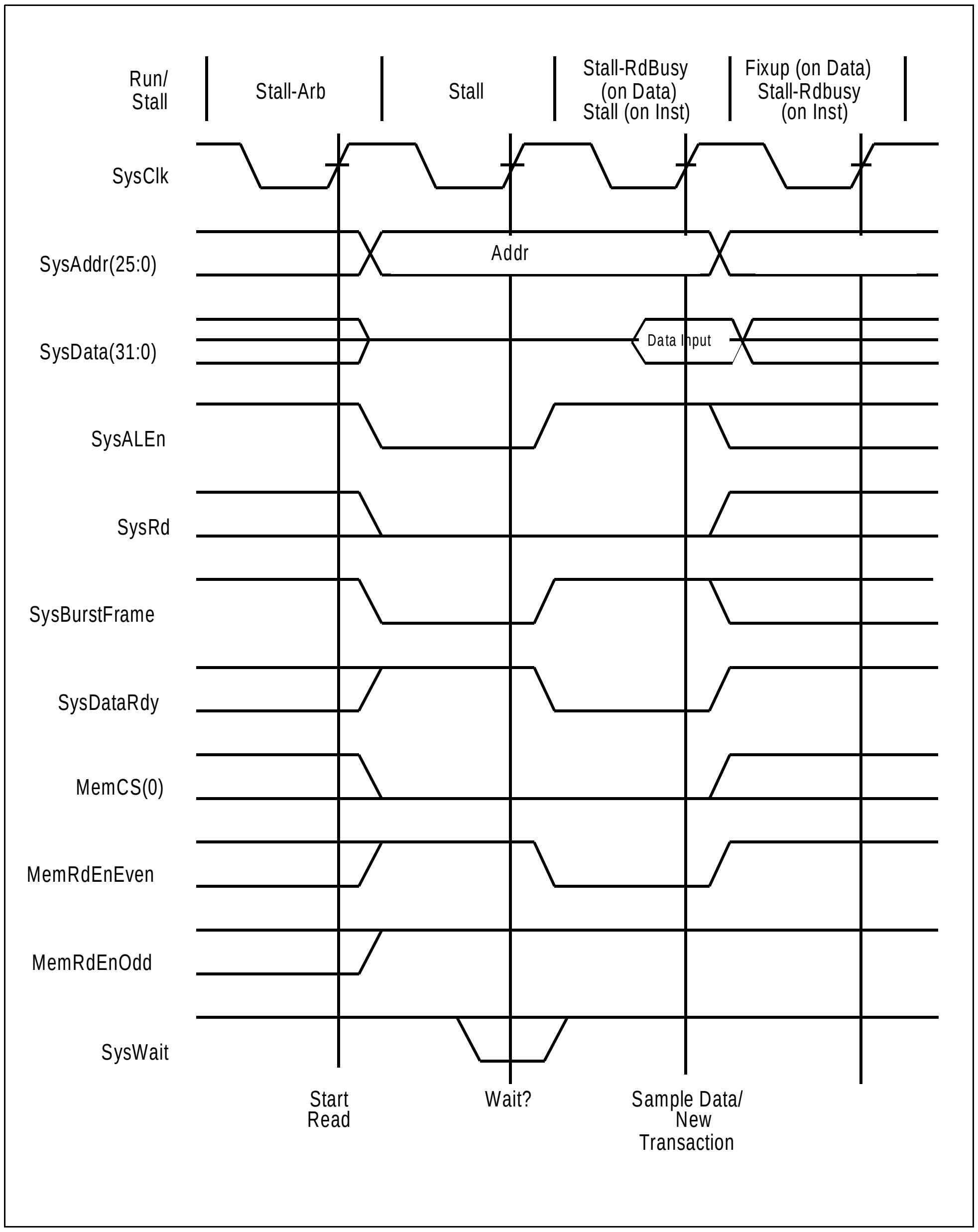


Figure 8.27 Interleaved “Odd” Read of FCT260-Type Memory

Notes

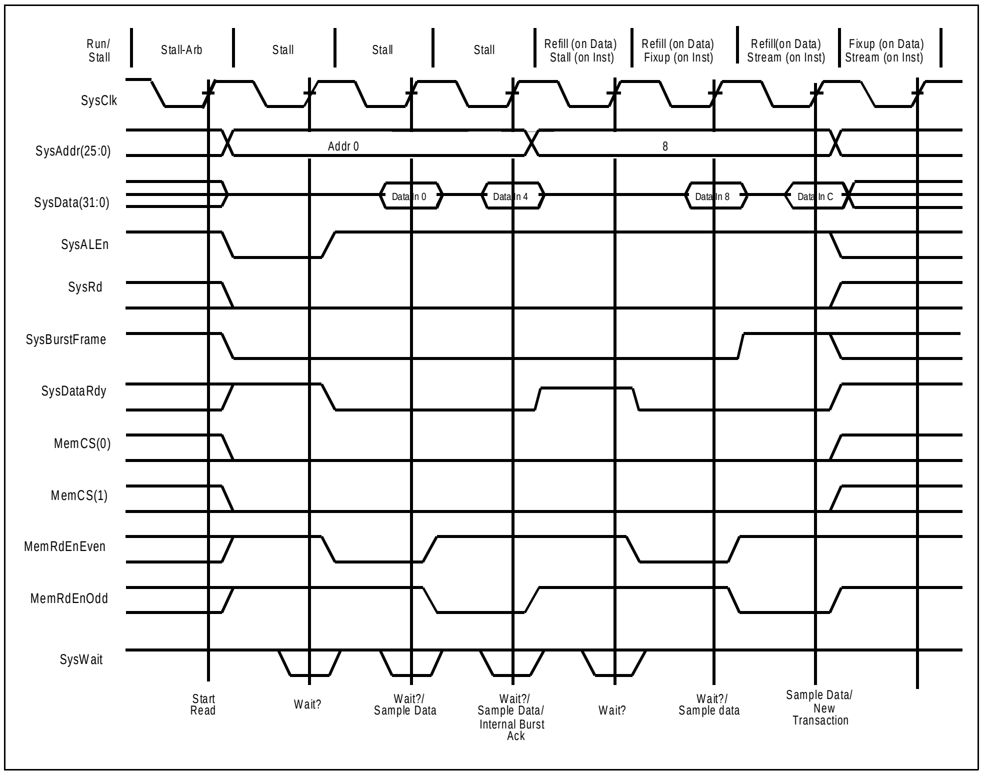


Figure 8.28 Interleaved Read using FCT245-Type Field.

Interleaved Read using FCT543-Type Field

Figure 8.29 illustrates an interleaved 4-word burst read using the FCT543-Type. The read and write output enables match the functionality of memory chip read and write output enables.

Notes

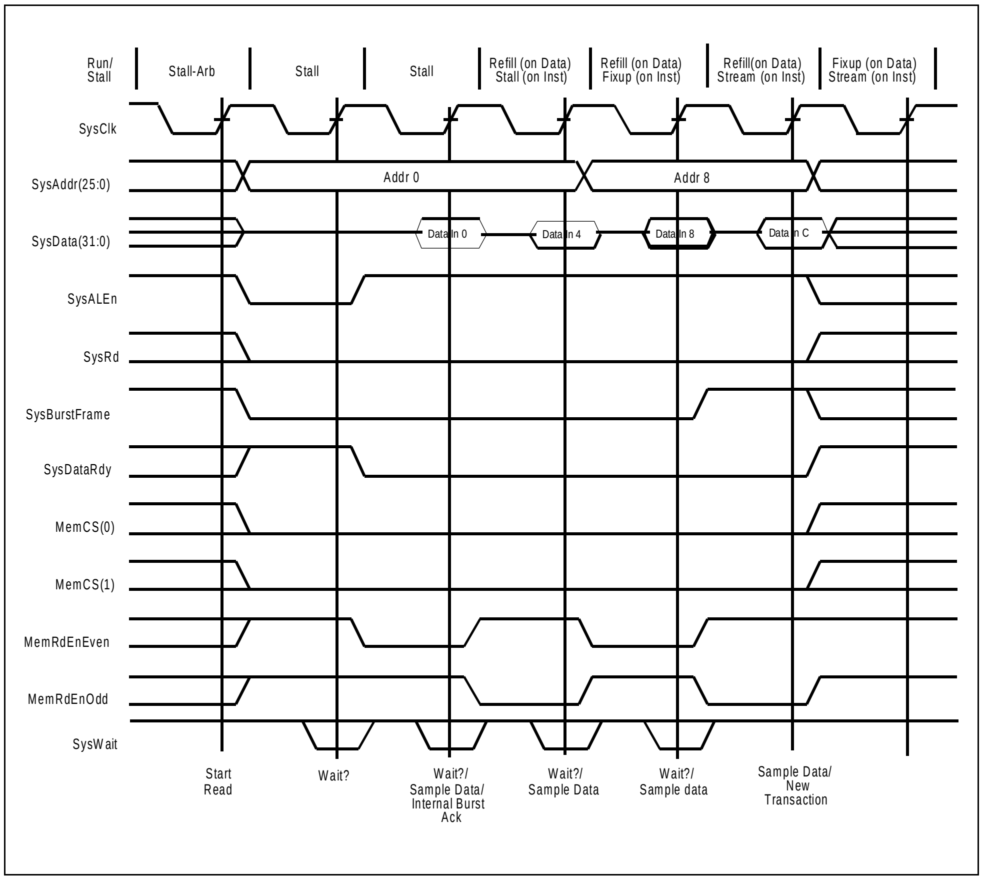


Figure 8.29 Interleaved Read using FCT543-Type Field.

The following Figure 8.30 shows a single datum access to an “even” bank of an interleaved memory system using 'FCT543 transceivers. Note that the timing of this access is identical with the timing of the first word of a 4-word access.

Figure 8.31 shows the analogous access to the “odd” bank of an interleaved memory system, using 'FCT543 transceivers. In this figure, the timing is identical with the timing of the access of the *second* word of a 4-word access; however, the first word is not actually returned to the CPU.

Notes

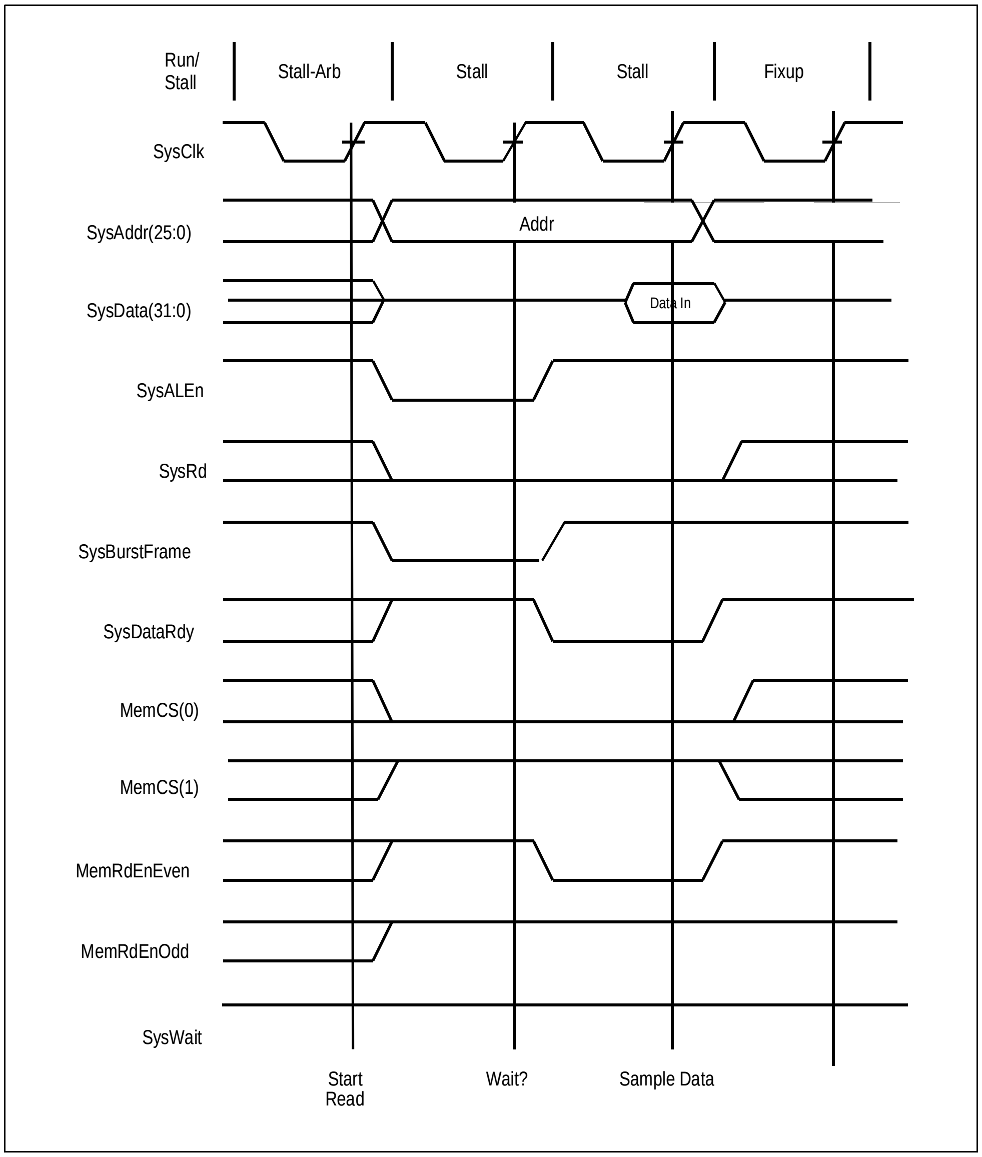


Figure 8.30 "Even" Read of FCT543-Type Memory

Due to a limitation on the number of transceiver control signals, there is a performance difference between even and odd single word accesses. Note, however, that this should not affect system performance of the following operations:

- *single word accesses will occur for uncached loads, and uncached instructions. These are typically not time critical programs or data.*
- *cached data accesses will only occur if the data block refill option is selected to "one word" rather than "four words." However, this selection is unlikely for an interleaved memory, which dramatically mitigates the time required for 4-word accesses, and thus is expected to use 4-word accesses on data cache misses. For more information, refer to the DBlockRefill ('DBR') option in the Coprocessor0 Cache Configuration documentation in Chapter 5.*
- *cached instruction misses are always satisfied using 4-word read accesses.*

Notes

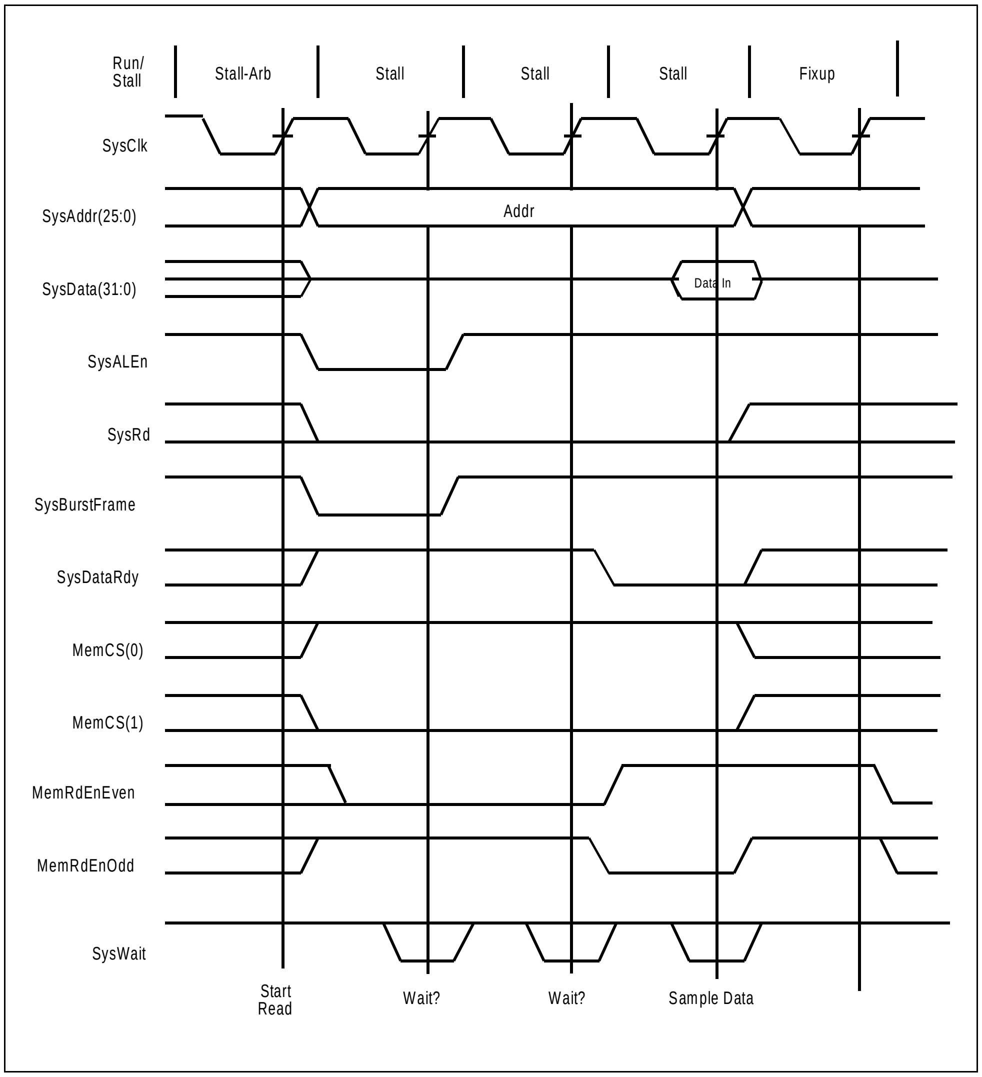


Figure 8.31 "Odd" Read of FCT543-Type Memory

Interleaved Writes

Figure 8.32 and Figure 8.33 illustrate interleaved writes for the 260-Type, the 543-Type, and the 245-Type, respectively. Because the byte enables, MemWrEn(3:0), are not duplicated for even and odd cases, burst writes do not occur any quicker than they do for non-interleaved cases. Any subsequent words are delayed by the MemWrEn(3:0) de-asserting for 1 clock (or 2 clocks if the Start Repeat Field is set). The 245-Type is different from the others in that MemRdEnEven or MemRdEnOdd is asserted for the 245's Output Enable pin.

Notes

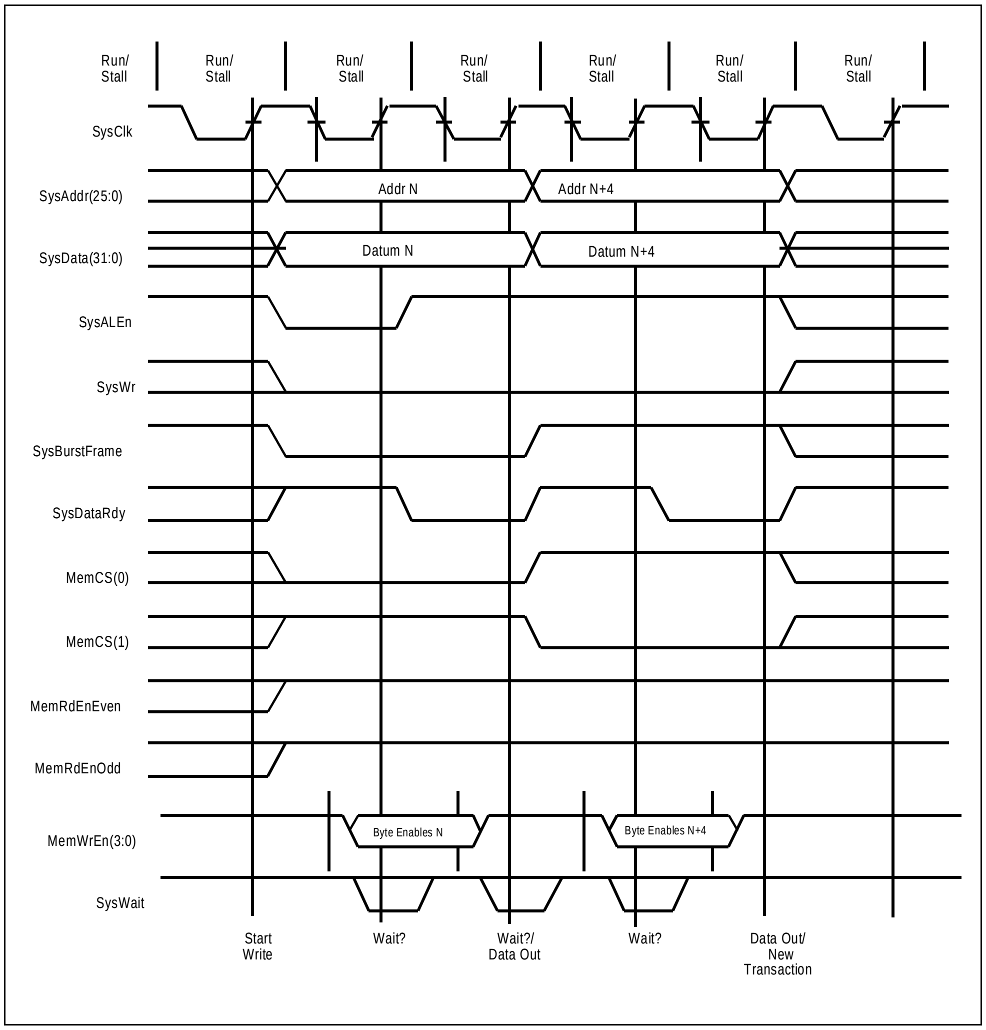


Figure 8.32 Interleaved Write using FCT260-Type and FCT543-Type Fields.

Notes

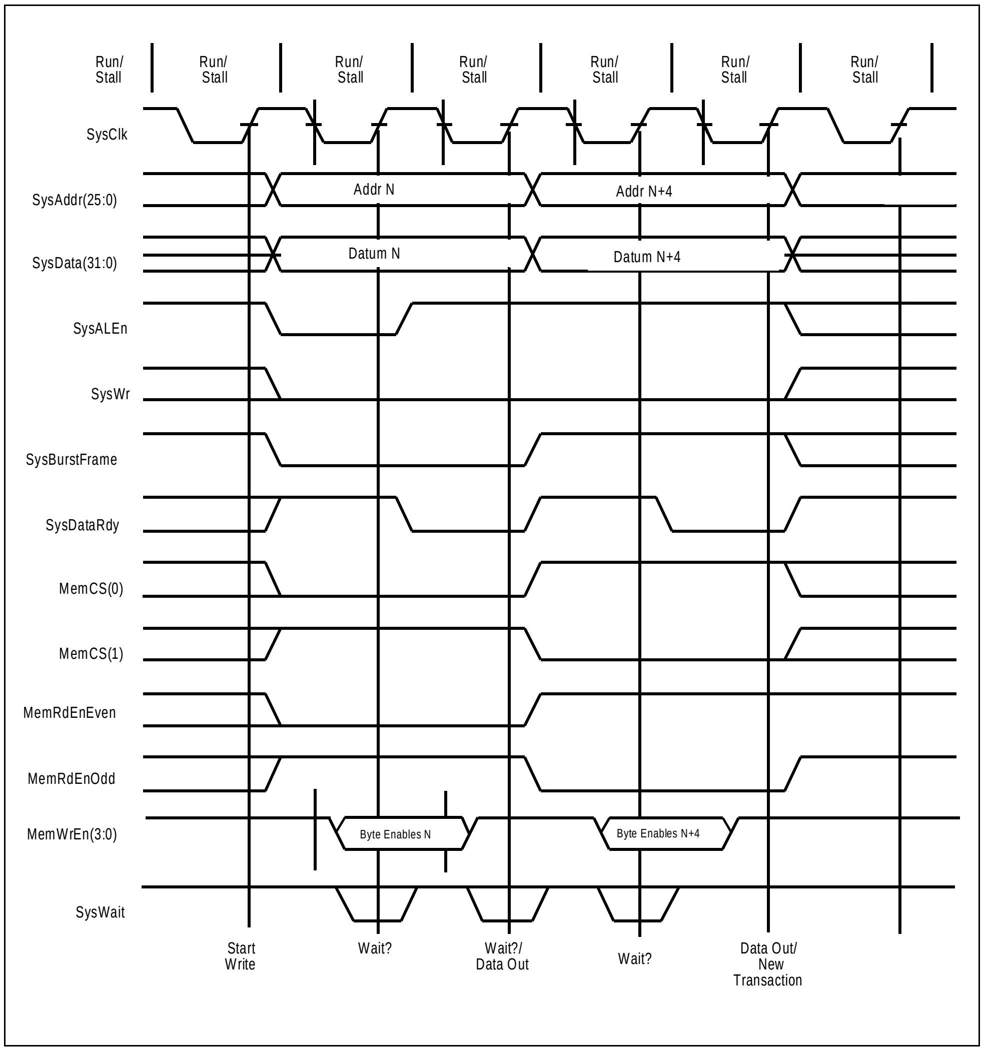


Figure 8.33 Interleaved Write using FCT245-Type Field.

System Examples

32-bit SRAM/ROM Directly Connected or using 543 Transceivers

Figure 8.34 shows a typical 32-bit SRAM memory system. The example is also applicable to Flash Memory systems and with the elimination of the write-related signals, to ROM systems. In small systems, the SRAM/ROM can be attached directly to the SysAddr and SysData buses. In larger systems, FCT543 transceivers can be added between the memory bank and the SysData bus. Also in large systems, the SysAddr bus can also be buffered using FCT244 buffers. Note that all even MemCS()s that use transceivers must be placed behind the same set of transceivers unless external decoding is done for each MemRdEnEven OR MemCS().

Notes

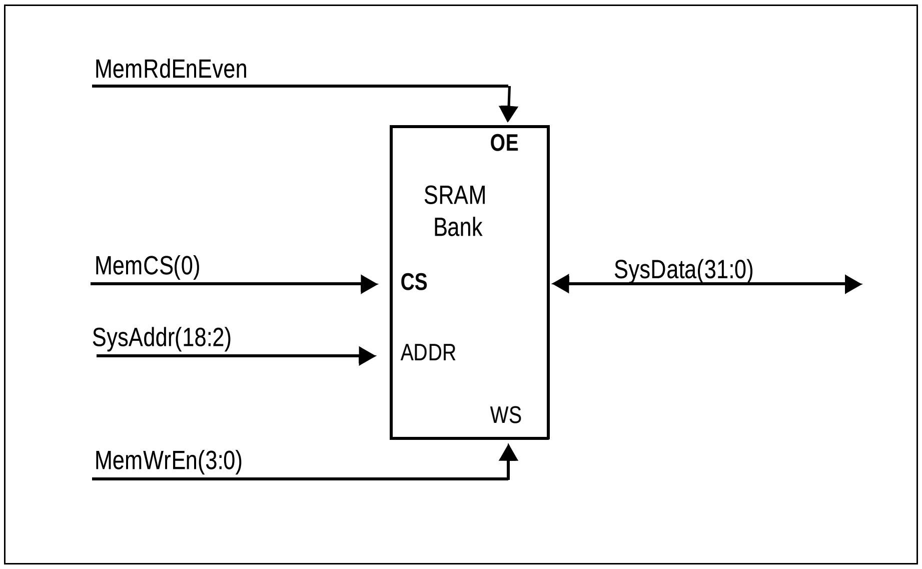


Figure 8.34 32-bit SRAM System.

Figure 8.35 shows a typical 32-bit SRAM memory system using an odd MemCS() line. Note that all odd MemCS()s that use transceivers must be behind the same set of transceivers unless external decoding is done for each MemRdEnOdd OR MemCS().

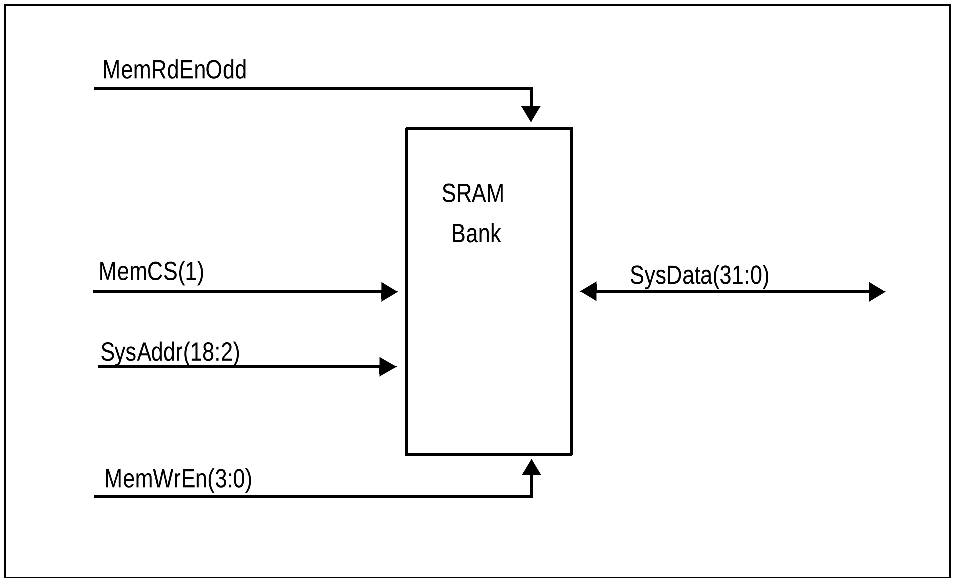


Figure 8.35 32-bit SRAM System using an Odd Chip Select.

32-bit SRAM using 245 Transceivers

Figure 8.36 shows a typical 32-bit SRAM memory system using FCT245 transceivers. This example is also applicable to Flash Memory systems and with the elimination of the write-related signals and the substitution of FCT244 buffers to ROM systems. The functionality of MemRdEnEven or MemRdEnOdd is modified so that the read enables will also assert on writes. The use of a 245-Type chip select requires that all the odd or all the even memory chip selects also be of the same 245-Type type.

Notes

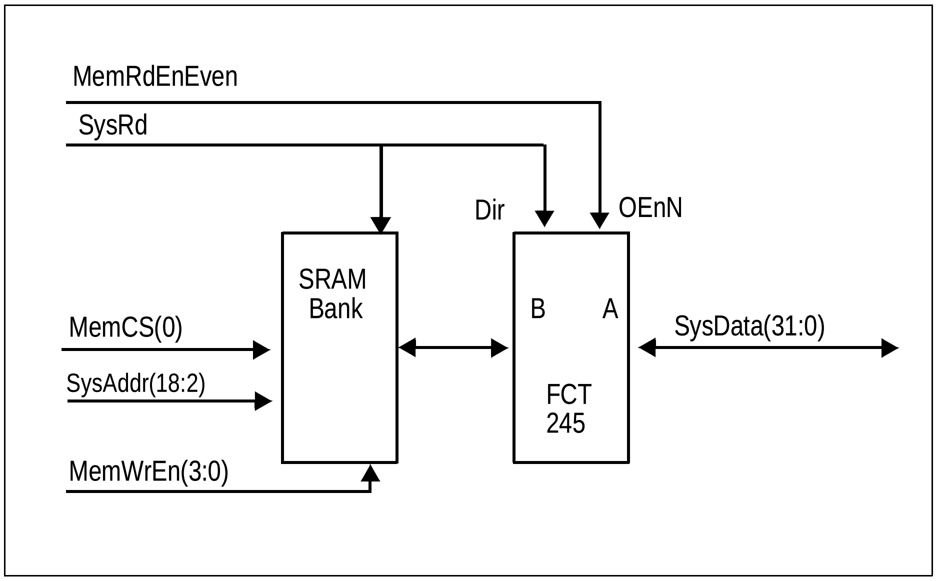


Figure 8.36 32-bit SRAM System using FCT245-Type.

Interleaved SRAM/ROM using 245 Transceivers

In Figure 8.37 an interleaved system using the Interleaved-FCT245-Type is shown. FCT245s are 16-bit (or 8-bit) bidirectional transceivers. Four (or eight) are required per bank. Because FCT245s do not latch the data of the odd words, the generation of the next double word address (0x8) for words three and four is delayed. MemRdEnEven and MemRdEnOdd change their functionality in that if their bank is selected, as they assert on *both reads and writes* such that they can be connected to their bank's 245 Output Enable pin.

Note: The RC36100 cannot have a stand-alone 32-bit odd bank, because it must be interleaved with an even bank partner. For the FCT245, interleaved ROMs must be booted with 32-bit port size and load the initial first few instructions from every even word, up until the boot code initializes the memory controller registers.

Notes

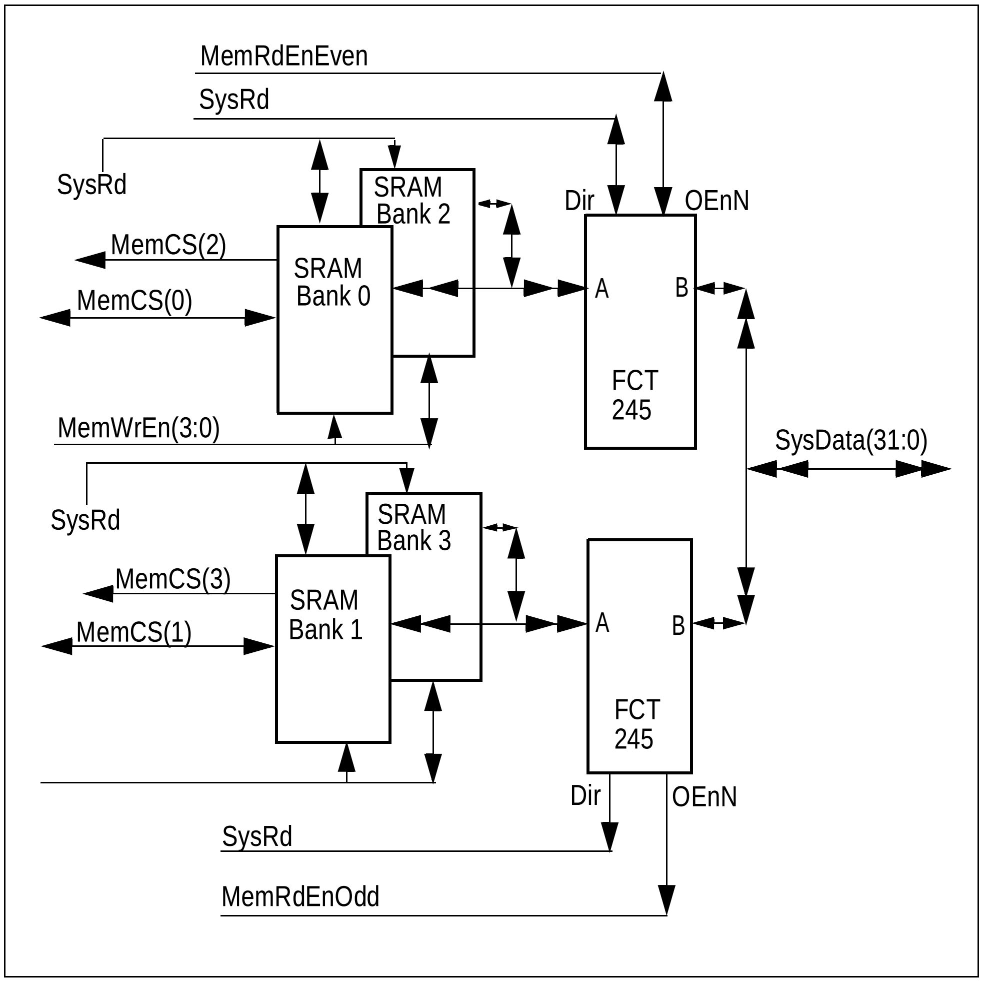


Figure 8.37 Interleaved FCT245-Type System.

Interleaved SRAM/ROM using 260 Multiplexers

In Figure 8.38 an interleaved system using the Interleaved-FCT260-Type is shown. FC260s are 12-bit bidirectional multiplexors. Thus three FCT260s are required per bank pair. MemRdEnOdd is used for both the Path Selection and the Odd Bank Latch Enable. MemRdEnEven changes its functionality in that it asserts for both the even and odd Data Sampling periods. The LSB address lines change on a doubleword boundary from 0x0 to 0x8 at the same time word 2 is being read.

Note that during an even (odd) bank read access, the chip select (for example, CS0 or CS2) is used to keep the two banks from contending on the data bus. In this example, SysWr is used to control the data path outputs on writes.

Notes

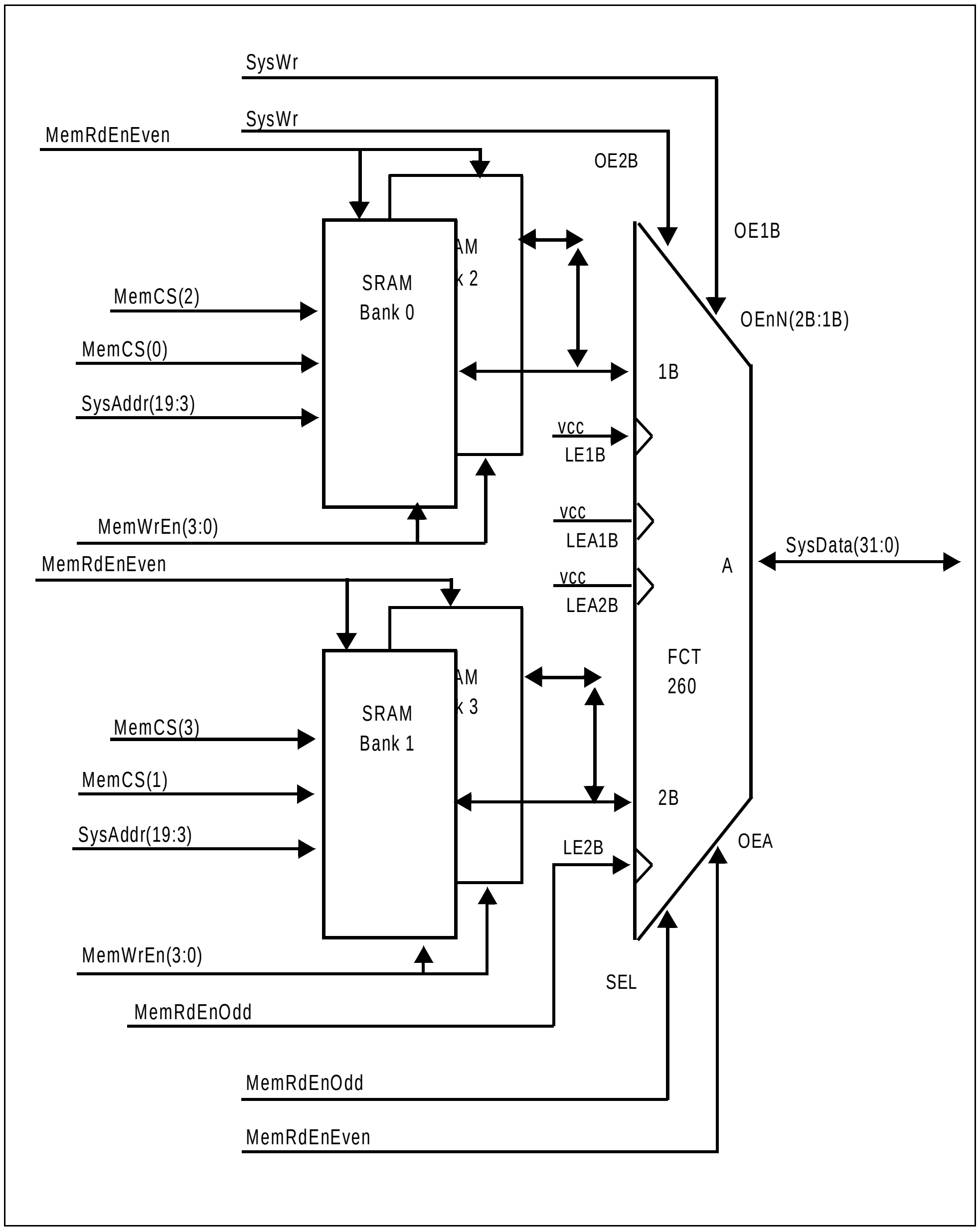


Figure 8.38 Interleaved FCT260-Type System.

Interleaved SRAM/ROM using FCT543 Transceivers

Figure 8.39 shows an interleaved system using the Interleaved-FCT543-Type. FCT543s are 16-bit bidirectional registers. Thus, four FCT543s are required per bank pair. MemRdEnOdd is used for both the Path Selection and the Odd Bank Latch Enable. MemRdEnEven changes its functionality in that it asserts for both the even and odd Data Sampling periods. The LSB address lines change on a doubleword boundary from 0x0 to 0x8 at the same time that word 2 is being read.

In systems that contain memories with fast output disable times, only two FCT543s may be required by letting MemRdEnEven and MemRdEnOdd control the outputs of the two memory banks behind a common transceiver. Such an approach usually implies that the two memory banks be of the same type and that the type specify output disable times less than output enable times.

Note: The RC36100 cannot have a stand-alone 32-bit odd bank, because it must be interleaved with an even bank partner. For the FCT543, interleaved ROMs must be booted with 32-bit port size and load the initial first few instructions from every even word, up until the boot code initializes the memory controller registers.

Notes

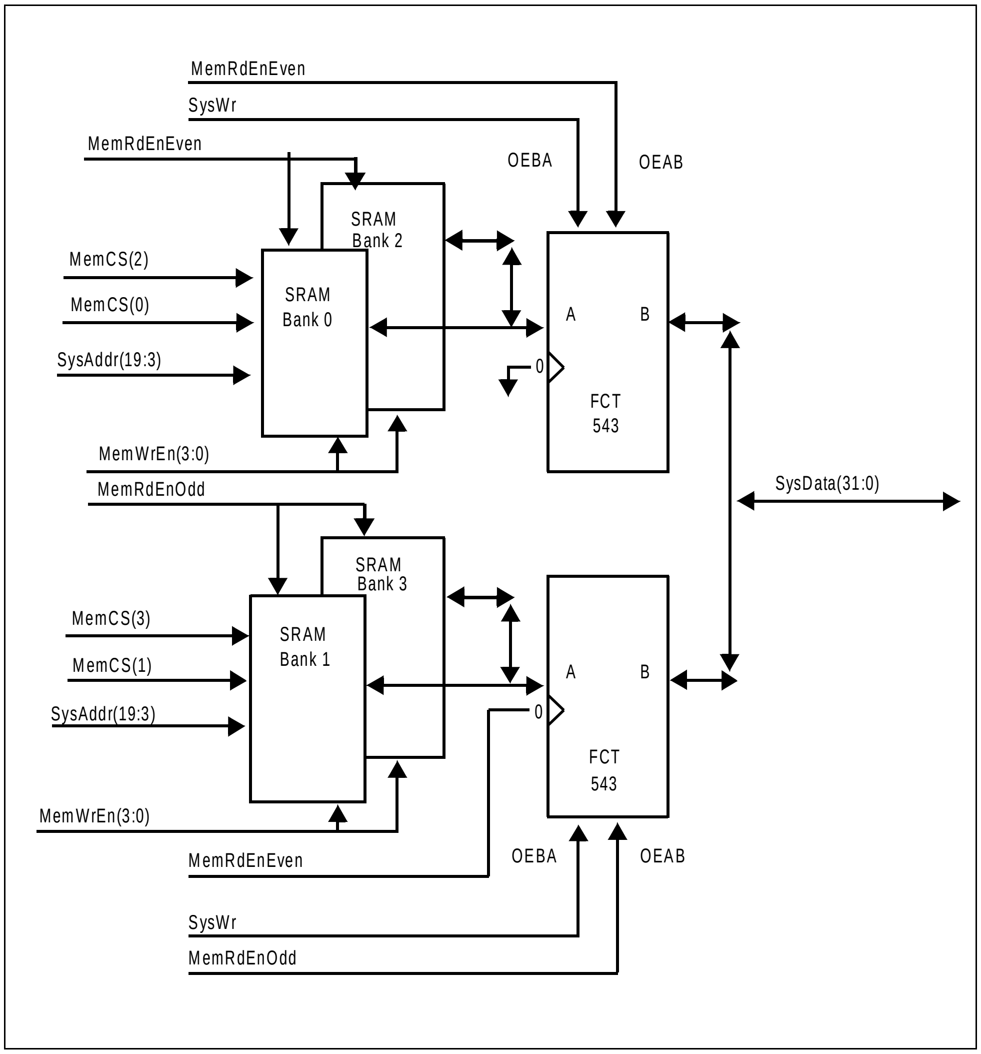


Figure 8.39 Interleaved FCT543-Type System

16-bit SRAM/ROM

Figure 8.40 and Figure 8.41 show a typical 16-bit SRAM memory system. These examples are also applicable to Flash Memory systems and, with the elimination of the write-related signals, to ROM systems. Since a 16-bit system is a smaller system, the SRAM/ROM can be attached directly to the SysAddr and SysData buses. In larger systems, FCT543 transceivers can be added between the memory bank and the SysData bus. Also in large systems, the SysAddr bus can be buffered using FCT244 buffers.

Note: With 16-bit systems it is imperative that the correct data line connections are made. Big Endian systems must attach SysData(31:16) and Little Endian systems must attach SysData(15:0).

Hooking up the correct data lines insures that byte gathering can occur on word accesses and is required for boot-proms to execute instructions. SRAM systems that require later expansion to 32-bits must externally multiplex a MSB address line with SysAddr(1).

Notes

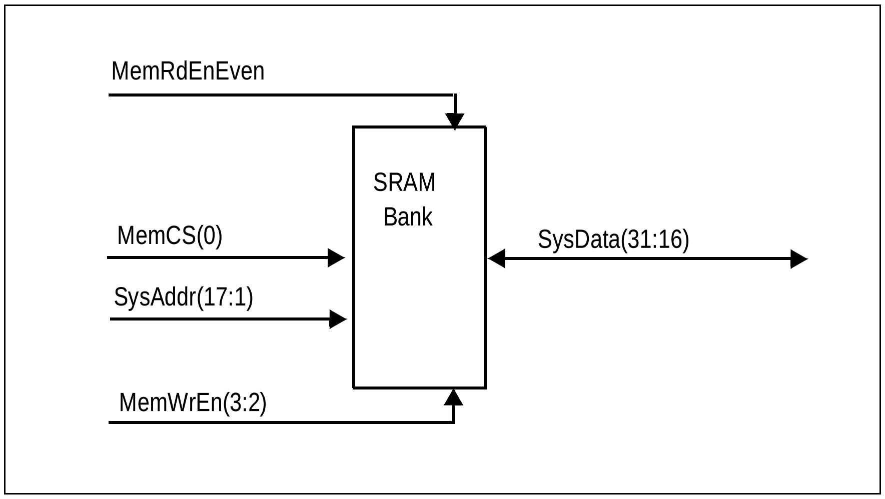


Figure 8.40 16-bit Big Endian SRAM System.

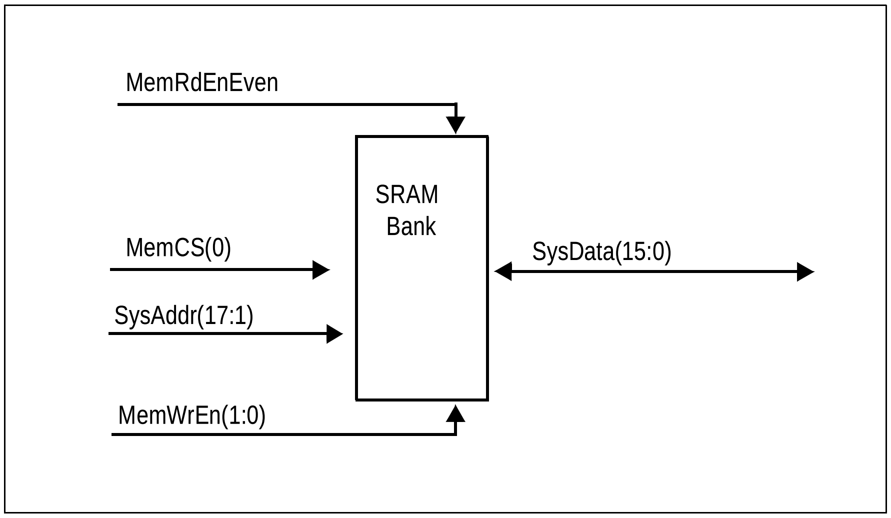


Figure 8.41 16-bit Little Endian SRAM System.

8-bit SRAM/ROM

Figure 8.42 and Figure 8.43 show two typical 8-bit SRAM memory systems. These examples are also applicable to Flash Memory systems and, with the elimination of the write-related signals, to ROM systems. Since an 8-bit system is a smaller system, the SRAM/ROM can be attached directly to the SysAddr and SysData buses. In larger systems, FCT543 transceivers can be added between the memory bank and the SysData bus. Also in large systems, the SysAddr bus can be buffered using FCT244 buffers.

Note: With 8-bit systems it is imperative that the correct data line connections are made. Big Endian systems must attach SysData(31:24) and Little Endian systems must attach SysData(7:0).

Hooking up the correct data lines insures that byte gathering can occur on word accesses, and it is required for boot-proms to execute instructions. SRAM systems that require later expansion to 32-bits must externally multiplex a MSB address line with SysAddr(0).

Notes

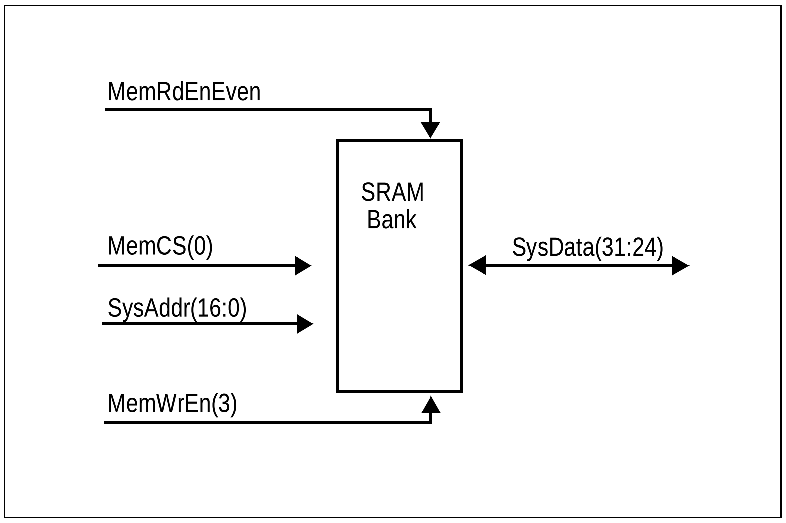


Figure 8.42 8-bit Big Endian SRAM System.

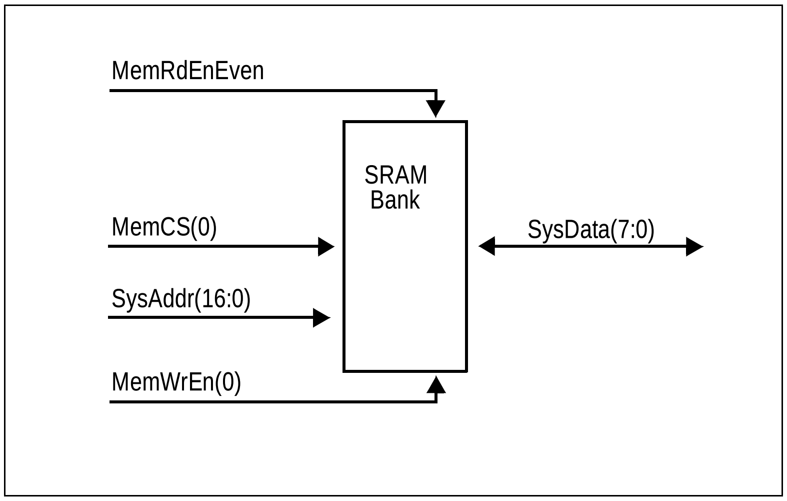


Figure 8.43 8-bit Little Endian SRAM System.

Dual-Port-Type

Dual-Port memory systems usually have a BusyN pin that indicates both ports of a memory location are being accessed simultaneously. The port that is accessed second receives the BusyN signal, indicating it must wait until the port that was accessed first is finished. Such Dual-Port systems can use the $\overline{\text{SysWait}}$ signal.

This allows a full Dual-Port memory access cycle when BusyN de-asserts. Using this type, if the Dual-Port memory glitches BusyN, (for instance, if the addresses match before $\overline{\text{MemCS}}$ is asserted, but don't afterwards) then the Dual-Port access will not be optimal in the sense that additional wait-states will be injected; however, operation will be correct. If increased optimization is required, the system designer can add external circuitry to gate BusyN with the beginning of $\overline{\text{MemCS}}$ so that it is ignored until BusyN is valid.

PCMCIA-Style Application

Multiple cards can be externally decoded using $\overline{\text{SysWrEn(1:0)}}$ programmed as inverted MemAddr(27:26). Table 8.16 lists the PCMCIA and RC36100 functional equivalents.

Notes

PCMCIA	RC36100 Function
A	SysAddr(25:0).
REG	Discrete PIO output pin.
~CE(2:1)	$\overline{\text{MemIoCS}}$() pair program defaulted to PCMCIA-Memory Style and then to PCMCIA-IO-Style when necessary.
~IORD	$\overline{\text{IoRd}}$.
~IOWR	$\overline{\text{IoWr}}$.
~OE	~(~$\overline{\text{MemIoCS}}$() & ~ $\overline{\text{SysRd}}$ & ~ $\overline{\text{SysDataRdy}}$)
~WE	$\overline{\text{MemWrEnEven}}$.
D	SysData(15:0).
~WAIT	SysWait
~INPACK	Ignored.
~IOIS16	Ignored.
RESET	System dependent or PIO output pin.
~IREQ	$\overline{\text{ExcInt}}$() pin.
CD(2:1)	Two PIO input pins.
BVD(2:1)	Two PIO input pins or ignore.
WP	System dependent.

Table 8.16 PCMCIA and RC36100 Functional Equivalents.

Notes

I/O Controller

Notes

Introduction

The IDT79RC36100 RISController integrates bus controllers and peripherals around the RISCore3000 family CPU core. One of the on-chip bus controllers is the "**I/O Controller**."

This chapter includes an overview on the I/O Controller interface, a complete description of the signal pins and their timing, and a discussion on how the interface relates to typical external hardware I/O devices and peripherals.

Because the I/O Bus Controller shares the Memory Controller (see Chapter 8) Registers and Chip Selects, the register description is not repeated here, except for the I/O specific "types" as described in the Type Field of each chip select's Control Register.

Features

- *The I/O controls support industry standard peripherals:*
- *I-Type I/O support*
- *M-Type I/O support*
- *PCMCIA-Style I/O support*
- *Controls up to 8 banks of I/O*

Note: Chip selects are shared with the Memory Controller.

- *Each $\overline{\text{IOCS}}$ can be programmed as:*
  - *Individual chip selects*
  - *Combined PCMCIA-I/O-style pair-wise chip selects*
- *Each Bank has Programmable Base Address*
- *Each Bank Size programmable from 32KB - 64MB*
- *8, 16, and 32-bit support*
- *Wait-State Generator features:*
  - *Programmable time from start to end of each data access for eacharea*
  - *Programmable time options for reads and writes*
  - *Programmable time options for single word accesses*
  - *Internally generates the RdCEnN and AckN timing for all CPU accesses*
  - *A programmed value may be overridden by the SysWait input signal*
- *Direct control of data path transceivers supports various options:*
  - *Direct Bus Connection*
  - *FCT245 Bidirectional Transceiver*
  - *FCT543 Bidirectional Registered Transceiver*

Block Diagram

Figure 9.1 is a functional block diagram of the Memory and I/O Controller. The main Memory and I/O Controller Control Signal State Machine is responsible for generating the basic I/O Control signals used to connect to external peripherals. These signals include chip selects, read enables/strobes, and write enables/strobes.

The I/O Controller works in cooperation with the Bus Interface Unit as described in Chapter 7. Thus the Control Signal State Machine sends and receives information from the BIU Controller for assistance with controlling the port width and controlling partial word reads and writes. The Control Signal State Machine also uses information stored in the software programmable I/O Controller Register Bank for example, to control I-Type versus M-Type accesses.

Notes

The Wait-State Generator takes care of sending and receiving information from the BIU Controller in order to control the sequencing and timing of reading and writing each individual datum. The number of wait-states is derived from the settings programmed into the Register Bank. Once the correct number of wait-states has been counted out, then the Wait-State Generator sets the appropriate internal BIU 'Acknowledge' signals. With the programmable Wait-State Generator it is possible to eliminate the external state machines that are traditionally used for this function.

The I/O Controller Decoder constantly monitors the Bus Interface Unit's address and data bus to see if either:

- *The access is to the I/O Controller's Register Bank.*
- *The access is in one of the I/O Controller's Chip Select Areas that is responsible for controlling the bus transaction.*

The Register Bank allows the software programmer access to the many different options of the I/O Controller. The chip select address ranges, the number of wait-states, the port-width of the chip select, and other similar options are programmed into the Register Bank as part of the software initialization sequence of the boot operating system.

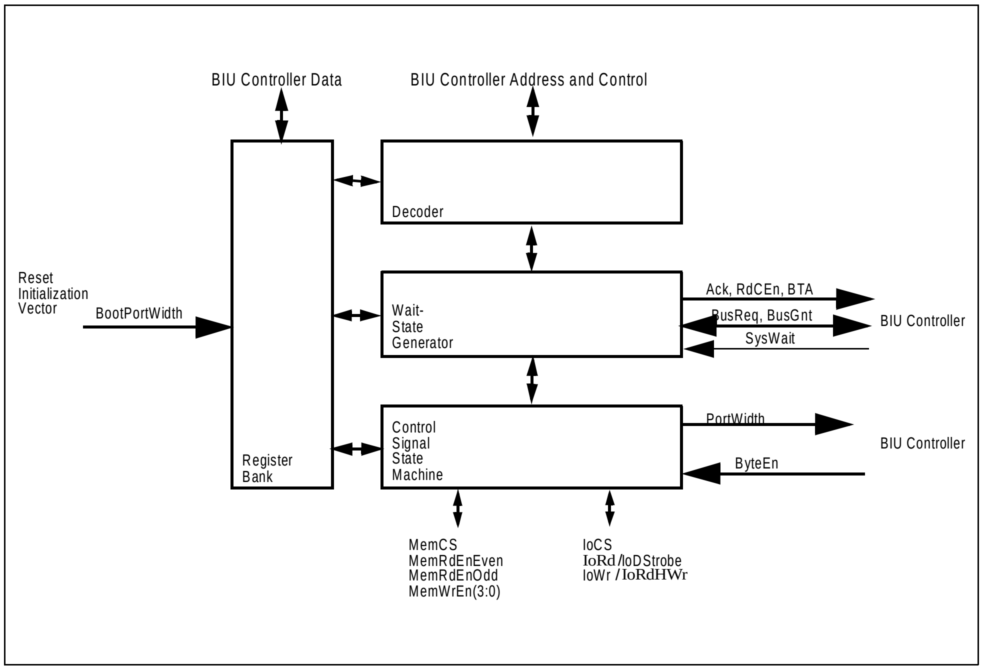


Figure 9.1 RC36100 I/O Bus Controller Block Diagram

I/O Bus Controller Interface Signals

These external interface pins are typically attached directly from the RC36100 RISController to external peripheral chips and their transceivers. Their descriptions are as follows:

IoCS(7:0)/ Output
MemCS(7:0)

I/O Chip Select: The IoCS signals are active low outputs used to select one of the programmable I/O controller areas. Typically each external peripheral is attached to a MemCS signal such that the peripheral can be selected and turned on during an I/O transaction. When the address from the CPU or DMA Controller matches the I/O memory block corresponding to a particular IoCS signal, that IoCS asserts at the beginning of the next bus transaction and de-asserts at the end of that transaction.

IoCS signals are used individually for non-PCMCIA systems or in pairs (for example IoCS 0 & 1, 2 & 3, 4 & 5, or 6 & 7) for PCMCIA systems.

The boot PROM memory is assigned to Mem/IoCS(0) and if interleaved, Mem/IoCS(1).

The IoCS chip selects are selectable and shared between Memory and I/O-types (see Chapter 8).

Notes

IoRd/ IoDStrobe **Output**

This output is multiplexed depending on the I/O-Type selected in the Control Register of each I/O Space. This output signal asserts only during an I/O Controller bus transaction.

Input/Output Read: This active low output signal is used as a read enable strobe in conjunction with the write enable strobe, IoWr. IoRd controls when the peripheral chip can drive the data signals back on to the main system data bus, SysData(). The timing of IoRd is such that SysAddr is stable before and after IoRd asserts.

Input/Output Data Strobe: This active low output signal is used as a data strobe for both reads and writes. IoDStrobe works in conjunction with the IoRdWr write status line. IoDStrobe controls when the data bus is valid. The de-asserting edge of IoDStrobe can be used to strobe the data into the peripheral on writes and indicates that the data has just been clocked into the CPU on reads. The timing of IoDStrobe is such that SysAddr is stable before and after IoDStrobe asserts. In some cases, IoCS() can be used instead of IoDStrobe.

IoWr/ IoRdHWr **Output**

This output is multiplexed depending on the I/O-Type selected in the Control Register of each I/O Space. This output only asserts during an I/O Controller bus transaction.

Input/Output Write: This active low output signal is used as an I/O write strobe in conjunction with the I/O read enable strobe, IoRd. IoWr controls when data is valid on the system data bus, SysData. When IoWr de-asserts, the peripheral can strobe the data into the chip. The timing of IoWr is such that SysAddr is stable before and after IoWr asserts.

Input/Output Read High and Write: This output is active high during I/O reads and active low during I/O writes. IoRdHWr is used as a read versus write status line. IoRdHWr works in conjunction with the IoDstrobe. The timing of IoDStrobe is such that both IoRdHWr and SysAddr is stable before and after IoDStrobe asserts. Note that since this signal asserts on the same clock edge as IoCS(), if setup is required to IoCs() as well, then SysWr (which asserts a clock earlier) can be substituted for IoRdHWr.

MemAddr(29:26) / MemWrEn(3:0) / MemByteEn(3:0) **Output/(Input during DMA)**

Memory Address Bus: During a PCMCIA Memory or I/O type access, the MemWrEn(3:0) bus is instead driven with Physical Address bits. On the RC36100 and RC3051-base family memory map, virtual and physical addresses (29:0) are the same. An application using PCMCIA can, for example, use MemAddr(27:26) to externally decode PCMCIA style chip selects into as many as four (256M/ 64M = 4) slots. In this mode (as in the other modes), the signals all return inactive high at the end of the bus transaction.

BIU Controller Signals

Many of the BIU Controller Signals are necessary to complete the I/O interface. These signals are listed here as a reminder. Information specific to the I/O Controller is given here and general information about the signal is given in Chapter 7, “System BIU Controller.”

SysAddr **Output/Input**

System Address Bus: SysAddr is an output bus when used with the I/O Controller. The MIPS architecture does not provide distinct memory and I/O spaces; thus MIPS I/O is considered to be “memory mapped I/O.” A 32-bit peripheral connects to the word offset of the Least Significant Bits (LSB) of SysAddr. Thus such a 32-bit peripheral skips SysAddr(1:0) and connects starting with SysAddr(2) on up. A 16-bit peripheral may connect to the halfword offset of the LSBs of SysAddr starting with SysAddr(1) on up. An 8-bit peripheral may connect the LSBs of SysAddr starting with SysAddr(0) on up. 16-bit and 8-bit peripherals traditionally use the word offset on MIPS systems so that they can be addressed from either Big or Little endian data paths.

Notes

SysData Output/Input

System Data Bus: A 32-bit peripheral connects the entire 32-bit SysData bus to its data pins or to its transceivers. A system implementation choice must be made on 16-bit and 8-bit peripherals.

In case 1, traditionally, MIPS systems have connected 16-bit and 8-bit peripherals with a word offset address. In such a case, a 16-bit device can be attached to SysData(31:16) or SysData(15:0) and accessed with either endianess by using a half-word offset of 0x2 for the opposite endianess. In such a case, an 8-bit device can be attached to SysData(31:24) or SysData(7:0) and accessed with either endianess by using a byte offset of 0x3 for the opposite endianess.

In case 2, 16-bit devices can use the halfword offset address, SysAddr(x:1) and 8-bit devices can use the byte address, SysAddr(x:0).

Note: In this case, 16-bit and 8-bit peripherals connect to particular data pins depending on whether the Endianess of the system is Big Endian or Little Endian.

Thus 16-bit peripherals use SysData(31:16) if they are Big Endian and SysData(15:0) if they are Little Endian. 8-bit peripherals use SysData(31:24) if they are Big Endian and SysData(7:0) if they are Little Endian. The User Mode Reverse Endianess Bit in the CP0 Status Register has no effect on the connections to SysData, however, it is strongly recommended that the Reverse Endianess Bit not be used to “correct” an endianess connection as it does not function in the kernel mode.

$\overline{\text{SysWait}}$ Input

System Wait Negated: $\overline{\text{SysWait}}$ can be used by an external source to add wait-states to the I/O Controller. Since the I/O Controller itself has a Wait-State Generator, $\overline{\text{SysWait}}$ typically is not needed and can be pulled-up with a resistor. The most likely application of $\overline{\text{SysWait}}$ is for an asynchronous memory event such as a Dual-Port Memory Busy signal which can be used to attach to $\overline{\text{SysWait}}$ to delay the beginning of a I/O Transaction in the Wait Mode option of the Memory I/O Control Register 2. Please see Chapter 7 for a general description of $\overline{\text{SysWait}}$.

Overview of the I/O Controller

The I/O Controller provides control for all I/O spaces. These I/O spaces are memory mapped and are intended for use by items such as LAN controllers, SCSI controllers, I/O signal conditioners, A/D, and D/A chips. Such peripheral chips typically have address inputs, data I/O, chip select, read output enable, and if writable, a write enable strobe.

Chip Selects

The I/O Controller contains a maximum of eight separate memory spaces, each having its own I/O Controller Chip Select ($\overline{\text{IOCS}}$) output pin. Each $\overline{\text{IOCS}}$ space occupies from 32KB to 256MB of address space of which 64MB is externally addressable (due to the 26 address lines), and the address space that each $\overline{\text{IOCS}}$ decodes is programmable. The I/O Controller will use the programmed information in the MSB and LSB Base Address Registers along with the size (32KB to 256MB) of the given area as programmed in the MSB and LSB Bank Mask Registers.

This information is used to compare the address asserted by the CPU-BIU or DMA Controller to determine if that particular $\overline{\text{IOCS}}$ area is being accessed for the current read or write. Each area supports single datum reads and writes. Burst reads and writes are not supported in the IOCS area. The port size of the data path (8, 16, or 32-bit) of each area is also programmable with each area's Control Register.

The $\overline{\text{IOCS}}$ signal can be used in pairs for PCMCIA. The pairs are $\overline{\text{IOCS(3:2)}}$ for one interleaved area and for the others, $\overline{\text{MemCS(5:4)}}$ and $\overline{\text{MemCS(7:6)}}$. When in the PCMCIA mode, both chip selects within a pair must be programmed to the same values.

Signal Control Interface

The I/O Controller provides read enables and write enables that are suitable for direct chip connection. I-Type read and write enables can in general also be attached to FCT543 type transceivers. M-Type write line and data strobes can in general also be attached to FCT245 type transceivers.

Notes

Wait-State Generator

The Wait-State Generator (WSG) controls the speed of the I/O accesses to and from the Bus Interface Unit Controller. This includes the time from the start of an I/O transaction until the first datum is sent or received. The WSG also is programmed to generate the internal RdCEnN and AckN signals for CPU read and write requests.

The internal Acknowledge signal, "AckN" (as described in Chapter 7) is the same as the external signal pin that the RC3051 RISController family uses. On single word reads and on both single word and burst writes, AckN is automatically placed at the end of the transaction by the WSG. Since burst I/O reads and writes are not supported, the Control Register 'Burst Ack' field is not needed.

Since burst reads and writes to I/O are not supported, the software is required to access I/O spaces with single datum loads which are in general non-cached and the same data size as the port width.

The signal called SysWait can be used to override the programmed settings of the Wait-State Generator. When SysWait is asserted, actions performed by the WSG will depend on when it is asserted relative to the transaction. SysWait has a pipeline delay, such that it must be asserted two clocks before the desired effect is noticeable. By asserting it immediately after a datum is received or transmitted, the next datum can be delayed. However, using SysWait for this purpose is generally not recommended because the WSG has the same functionality. SysWait is useful for accessing off-card "Ack"-type peripherals.

Register Option Programmability

The Memory and I/O Controller contains 8 sets of registers, one set for each chip select. These registers allow the I/O Controller to be configured for different types and speeds of peripherals. Thus a wide variety of system speed/cost/manufacturing trade-offs can be accommodated.

Register Descriptions

The Memory and I/O Controller Registers are divided into 8 sets of registers, one set for each chip select memory area. Physical addresses and register descriptions are shown in Table 9.1.

Notes

Phys. Address	Register Description
0xFFFF_E200	Memory and I/O LSB Base Address Register for Bank 0
0xFFFF_E204	Memory and I/O MSB Base Address Register for Bank 0
0xFFFF_E208	Memory and I/O LSB Bank Mask Register for Bank 0
0xFFFF_E20C	Memory and I/O MSB Bank Mask Register for Bank 0
0xFFFF_E210	Memory and I/O Control Register for Bank 0
0xFFFF_E214	Memory and I/O LSB Wait-State Generator Register for Bank 0
0xFFFF_E218	Memory and I/O MSB Wait-State Generator Register for Bank 0
0xFFFF_E220	Memory and I/O LSB Base Address Register for Bank 1
0xFFFF_E224	Memory and I/O MSB Base Address Register for Bank 1
0xFFFF_E228	Memory and I/O MSB LSB Bank Mask Register for Bank 1
0xFFFF_E22C	Memory and I/O MSB MSB Bank Mask Register for Bank 1
0xFFFF_E230	Memory and I/O MSB Control Register for Bank 1
0xFFFF_E234	Memory and I/O MSB LSB Wait-State Generator Register for Bank 1
0xFFFF_E238	Memory and I/O MSB MSB Wait-State Generator Register for Bank 1
0xFFFF_E240	Memory and I/O MSB LSB Base Address Register for Bank 2
0xFFFF_E244	Memory and I/O MSB MSB Base Address Register for Bank 2
0xFFFF_E248	Memory and I/O MSB LSB Bank Mask Register for Bank 2
0xFFFF_E24C	Memory and I/O MSB MSB Bank Mask Register for Bank 2
0xFFFF_E250	Memory and I/O MSB Control Register for Bank 2
0xFFFF_E254	Memory and I/O MSB LSB Wait-State Generator Register for Bank 2
0xFFFF_E258	Memory and I/O MSB MSB Wait-State Generator Register for Bank 2
0xFFFF_E260	Memory and I/O MSB LSB Base Address Register for Bank 3
0xFFFF_E264	Memory and I/O MSB MSB Base Address Register for Bank 3
0xFFFF_E268	Memory and I/O MSB LSB Bank Mask Register for Bank 3
0xFFFF_E26C	Memory and I/O MSB MSB Bank Mask Register for Bank 3
0xFFFF_E270	Memory and I/O MSB Control Register for Bank 3
0xFFFF_E274	Memory and I/O MSB LSB Wait-State Generator Register for Bank 3
0xFFFF_E278	Memory and I/O MSB MSB Wait-State Generator Register for Bank 3
0xFFFF_E280	Memory and I/O MSB LSB Base Address Register for Bank 4
0xFFFF_E284	Memory and I/O MSB MSB Base Address Register for Bank 4
0xFFFF_E288	Memory and I/O MSB LSB Bank Mask Register for Bank 4
0xFFFF_E28C	Memory and I/O MSB MSB Bank Mask Register for Bank 4
0xFFFF_E290	Memory and I/O MSB Control Register for Bank 4
0xFFFF_E294	Memory and I/O MSB LSB Wait-State Generator Register for Bank 4
0xFFFF_E298	Memory and I/O MSB MSB Wait-State Generator Register for Bank 4
0xFFFF_E2A0	Memory and I/O MSB LSB Base Address Register for Bank 5
0xFFFF_E2A4	Memory and I/O MSB MSB Base Address Register for Bank 5
0xFFFF_E2A8	Memory and I/O MSB LSB Bank Mask Register for Bank 5
0xFFFF_E2AC	Memory and I/O MSB MSB Bank Mask Register for Bank 5
0xFFFF_E2B0	Memory and I/O MSB Control Register for Bank 5
0xFFFF_E2B4	Memory and I/O MSB LSB Wait-State Generator Register for Bank 5
0xFFFF_E2B8	Memory and I/O MSB MSB Wait-State Generator Register for Bank 5
0xFFFF_E2C0	Memory and I/O MSB LSB Base Address Register for Bank 6
0xFFFF_E2C4	Memory and I/O MSB MSB Base Address Register for Bank 6
0xFFFF_E2C8	Memory and I/O MSB LSB Bank Mask Register for Bank 6
0xFFFF_E2CC	Memory and I/O MSB MSB Bank Mask Register for Bank 6
0xFFFF_E2D0	Memory and I/O MSB Control Register for Bank 6
0xFFFF_E2D4	Memory and I/O MSB LSB Wait-State Generator Register for Bank 6
0xFFFF_E2D8	Memory and I/O MSB MSB Wait-State Generator Register for Bank 6
0xFFFF_E2E0	Memory and I/O MSB LSB Base Address Register for Bank 7
0xFFFF_E2E4	Memory and I/O MSB MSB Base Address Register for Bank 7
0xFFFF_E2E8	Memory and I/O MSB LSB Bank Mask Register for Bank 7
0xFFFF_E2EC	Memory and I/O MSB MSB Bank Mask Register for Bank 7
0xFFFF_E2F0	Memory and I/O MSB Control Register for Bank 7
0xFFFF_E2F4	Memory and I/O MSB LSB Wait-State Generator Register for Bank 7
0xFFFF_E2F8	Memory and I/O MSB MSB Wait-State Generator Register for Bank 7

Note: Big Endian software must offset these addresses by b'10 (0x2), if halfword operations are used.

Table 9.1 Memory and I/O Controller Register Addresses and Descriptions

Notes

Memory and I/O Control Register7..0 ('MemIoCntrlReg(7..0)')

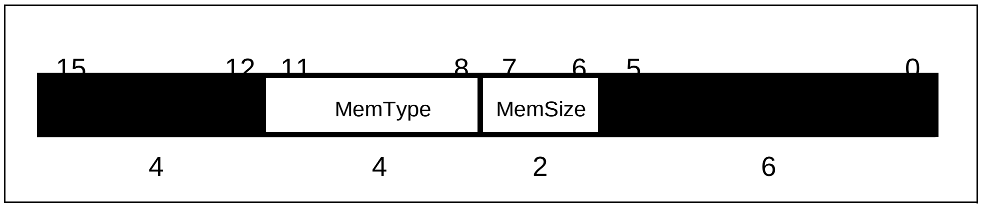


Figure 9.2 Memory and I/O Control Register (‘MemIoCntrlReg(7..0)’)

As shown in Figure 9.2, each MemCntrlReg contains a Type Field that has options specific to I/O types of signals.

Memory Type (‘MemType’) Field

The Type field determines the type of timing the Bus Interface will use. Table 9.2 lists the possible Type field values and their actions.

Value	Action
‘1011’	PCMCIA-Memory-Style
‘1010’	PCMCIA-I/O-Style
‘1001’	M-Type I/O
‘1000’	I-Type I/O
'0010'	Memory-Type for FCT260
‘0001’	Memory-Type for FCT245 (non-interleaved)
‘0000’	Memory-Type (for FCT543)
All other values	Reserved

Note:
PCMCIA-Style supports a PCMCIA host mode subset that is likely to be used with PCMCIA peripherals. PCMCIA-Memory and -IO Styles are intended for dynamic swapping by the software onto the same pair of chip selects. Typically, the Memory-Style is left on, and the I/O-Style is swapped in whenever it is needed, then swapped back to Memory-Style.

Table 9.2 Memory Type (MemType) Field Values and Descriptions

Portsize Width (‘MemSize’) Field

The PortSize field, shown in Table 9.3, determines the width of the memory or I/O port. The value is inverted relative to the reset initialization vector value.

Value	Action
‘11’	64-bit (32-bit 2-way interleaved) accesses (Valid for Memory Type only)
‘10’	16-bit accesses
‘01’	8-bit accesses
‘00’	32-bit accesses

Table 9.3 Portsize Width Field Values and Definitions

Notes

I-Type I/O Type:

The I-Type (Intel Type) puts the bus interface into a mode such that the I/O signals IoCS(), IoRd, and IoWr support I-type devices. Such devices have an address decoded chip select, IoCS(), and separate read (IoRd) and write (IoWr) data strobes. Older I-Type devices may also have an active high Reset input; an artifact that may require an external inverter.

M-Type I/O Type:

The M-Type (Motorola Type) puts the bus interface into a mode such that the I/O signals IoRd and IoWr are used as IoDStrobe and IoRdHWr, respectively. The device is assumed to have an address decoded chip select, IoCS() while using the IoRdHWr line as a status line indicating a read or write, and using IoDStrobe as a data strobe.

PCMCIA-I/O Style:

The PCMCIA-Style puts the bus interface into a mode such that the I/O signals IoCS(odd,even), IoRd, and IoWr support 16-bit slave PCMCIA devices. The odd and even chip select pair is used to indicate whether one or both of the byte lanes are valid. IoRd and IoWr are used as separate read and write data strobes.

I/O Controller Timing Diagrams

This section illustrates a number of timing diagrams applicable to RC36100 I/O transactions. The values for the AC parameters are contained in the separate document, "RC36100 RISController Data Sheet."

I/O Datum Size

All I/O accesses must be single datum accesses. For example, a 32-bit port must not use a cached burst access; a 16-bit port must use an uncached store or load halfword or byte operation (such as one of these: sh, lhu, sb, lbu). An 8-bit port must only use uncached byte operations (such as one of these: sb, lbu).

Read Transactions

The bus interface timing for read transactions is described in the following section. The internal bus interface to CPU core for read loads is described in Chapter 7.

Basic I-Type I/O Read with 0 Wait-States

Figure 9.3 illustrates a basic I-Type I/O Controller read transaction. Each transaction begins with both SysALEn and SysBurstFrame asserting. At this time, SysRd asserts (if it is not already in this state, as the result of a previous transaction). Now it is guaranteed that IoCS(), IoRd, and IoWr will be in their de-asserted states. And assuming there are no internally programmed StartRepeat wait-states, on the next clock cycle, SysBurstFrame de-asserts and IoCS() asserts.

On the third cycle, IoRd asserts. Then assuming there are no internally programmed RdStart2Datum wait-states, SysDataRdy asserts to indicate that the data from the I/O device is being sampled into the RISController. On the 4th clock cycle, IoRd de-asserts, indicating that the read data from the I/O device has just been latched into the RISController. On the next clock--the final clock of the transaction--IoCS() de-asserts, and the next transaction may begin.

During an I-Type transaction, there is 1 clock of address setup time before IoCS() asserts. All signals are setup before the read strobe, IoRd, asserts. After data has been sampled by the CPU, IoRd de-asserts, with all other signals having hold time. On the next clock, IoCS() de-asserts.

Notes

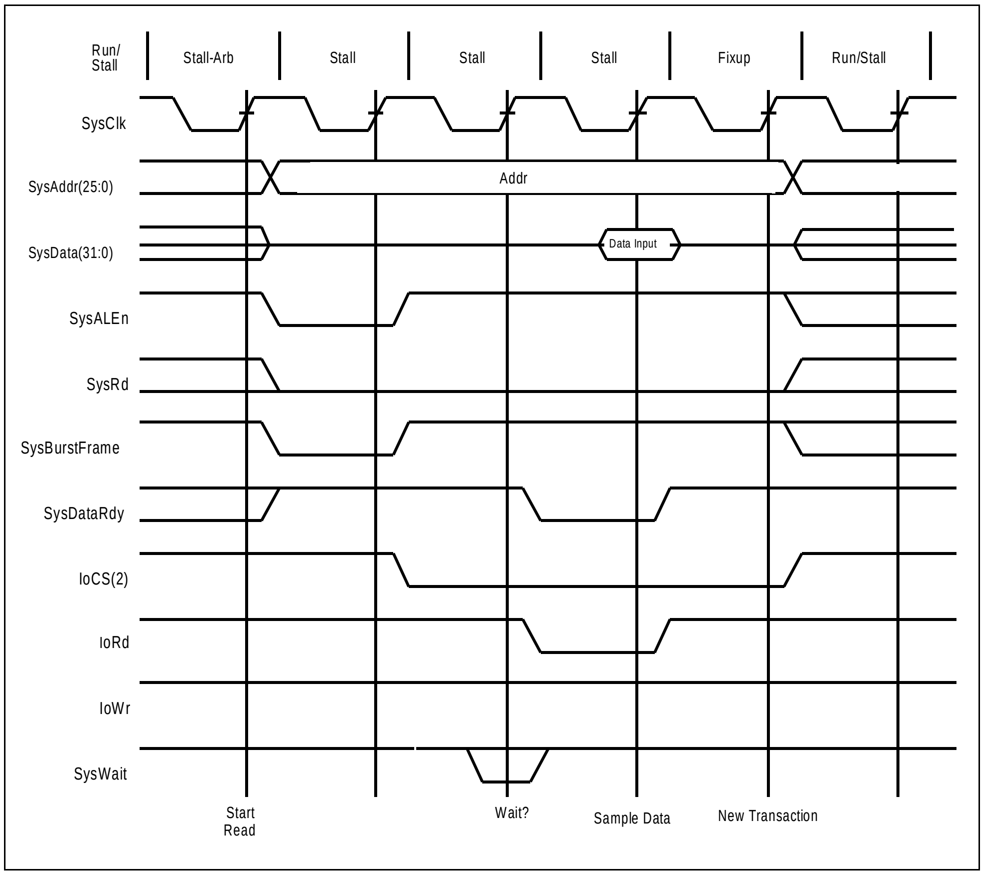


Figure 9.3 I-Type I/O Read with 0 Wait-States

Basic M-Type I/O Read with 0 Wait-States

Figure 9.4 illustrates a basic M-Type I/O Controller read transaction. Each transaction begins with both SysALEn and SysBurstFrame asserting. At this time, SysRd asserts (if it is not already in this state, as the result of a previous transaction). Now it is guaranteed that IoCS(), IoDStrobe (also known as IoRd), and IoRdHWr (also known as IoWr) will be in their de-asserted states. Then assuming there are no internally programmed StartRepeat wait-states, on the next clock cycle, SysBurstFrame de-asserts; and IoCS() asserts. If this is a write transaction, IoRdHWr (IoWr) will assert.

On the third cycle, IoDStrobe (IoRd) asserts. And assuming there are no internally programmed RdStart2Datum wait-states, SysDataRdy asserts, to indicate that the data from the I/O device is being sampled into the RISController. On the 4th clock cycle, IoDStrobe (IoRd) de-asserts, indicating that the read data from the I/O device has just been latched into the RISController. On the next clock--the final clock of the transaction--IoCS() de-asserts and, at the same time, the next transaction may begin.

During an M-Type transaction, there is 1 clock of address setup time before IoCS() asserts. All signals are setup before the data strobe, IoDStrobe, asserts. After data has been sampled by the CPU, IoDStrobe de-asserts, with all other signals having hold time. On the next clock, IoCS() de-asserts.

Note that because IoRdHWr asserts on the same clock edge as IoCS(), systems that require setup time from IoRdHWr to IoCs() can substitute SysWr for IoRdHWr. Many systems can also substitute IoCS() for IoDStrobe which, if the peripheral timing allows, may require fewer wait-states.

Notes

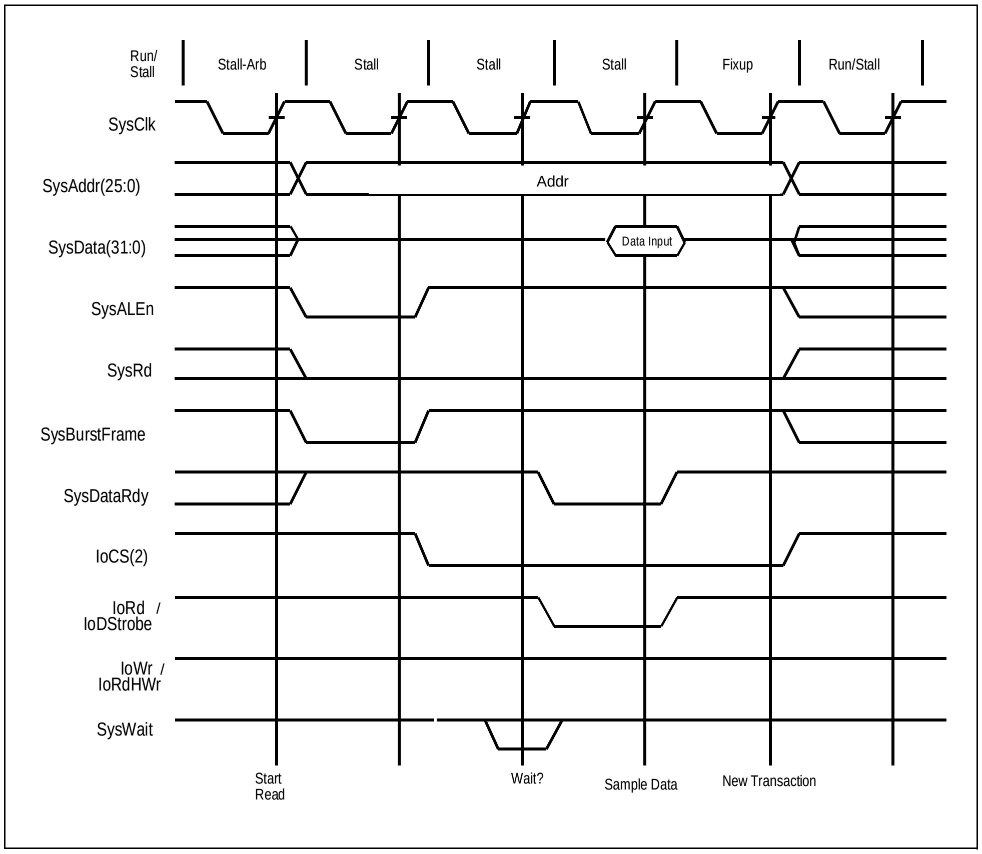


Figure 9.4 I/M-Type I/O Read, 0 Wait-States

Basic 16-bit PCMCIA-style I/O Read with 0 Wait-States

Figure 9.5 illustrates a basic 16-bit PCMCIA-Style I/O Controller read transaction. Each transaction begins with both SysALEn and SysBurstFrame asserting. At this time, SysRd asserts (if it is not already in this state, as the result of a previous transaction). Now it is guaranteed that IoCS(), IoRd, and IoWr will be in their de-asserted states. Then assuming that there are no internally programmed StartRepeat wait-states, on the next clock cycle, SysBurstFrame de-asserts and the IoCS() pair asserts. Note that on PCMCIA transactions, the IoCS() pair is asserted according to which bytes are enabled and valid. Thus if the even byte is valid, then the even IoCS() will assert. If the odd byte is valid, then the odd IoCS() will assert. If both bytes are valid, then both IoCS() signals in the pair assert.

On the third cycle IoRd asserts. And assuming there are no internally programmed RdStart2Datum wait-states, SysDataRdy asserts to indicate that the data from the I/O device is being sampled into the RISController. On the 4th clock cycle, IoRd de-asserts, indicating that the read data from the I/O device has just been latched into the RISController. On the next clock--the final clock of the transaction--the IoCS() pair de-asserts and, at the same time, the next transaction may begin.

During an I-Type transaction, there is 1 clock of address setup time before the IoCS() pair asserts. All signals are setup before the read strobe, IoRd, asserts. After data has been sampled by the CPU, IoRd de-asserts, with all other signals having hold time. On the next clock the IoCS() pair de-asserts.

Notes

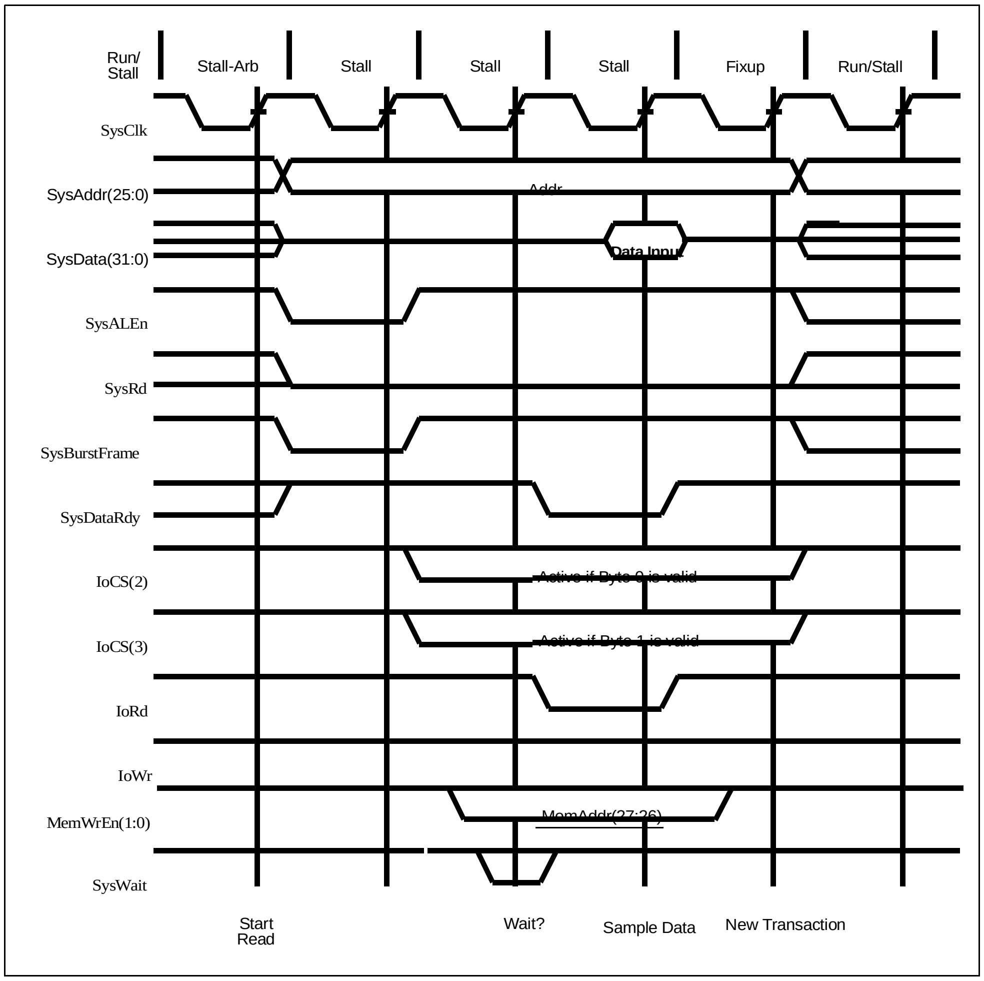


Figure 9.5 PCMCIA-Style I/O Read with 0 Wait-States

Basic I-Type I/O Write with 0 Wait-States

Figure 9.6 illustrates a basic I-Type I/O Controller write transaction. Each transaction begins with both SysALEn and SysBurstFrame asserting. At this time, SysWr asserts (if it is not already in this state, as the result of a previous transaction). Now it is guaranteed that IoCS(), IoRd, and IoWr will be in their de-asserted states. Assuming that there are no internally programmed StartRepeat wait-states, on the next clock cycle, SysBurstFrame de-asserts and IoCS() asserts.

On the third cycle, IoWr asserts. And assuming there are no internally programmed RdStart2Datum wait-states, SysDataRdy asserts to indicate that the data from the RISController is ready to be latched into the I/O device. On the 4th clock cycle, IoWr de-asserts, providing a means for the write data from the RISController to be latched into the I/O device. On the next clock--which is the final clock of the transaction--IoCS() de-asserts and, at the same time, the next transaction may begin.

During an I-Type transaction, there is 1 clock of address setup time before IoCS() asserts. All signals are setup before the write strobe, IoWr, asserts. After data has been sampled by the CPU, IoWr de-asserts, with all other signals having hold time. On the next clock, IoCS() de-asserts.

Notes

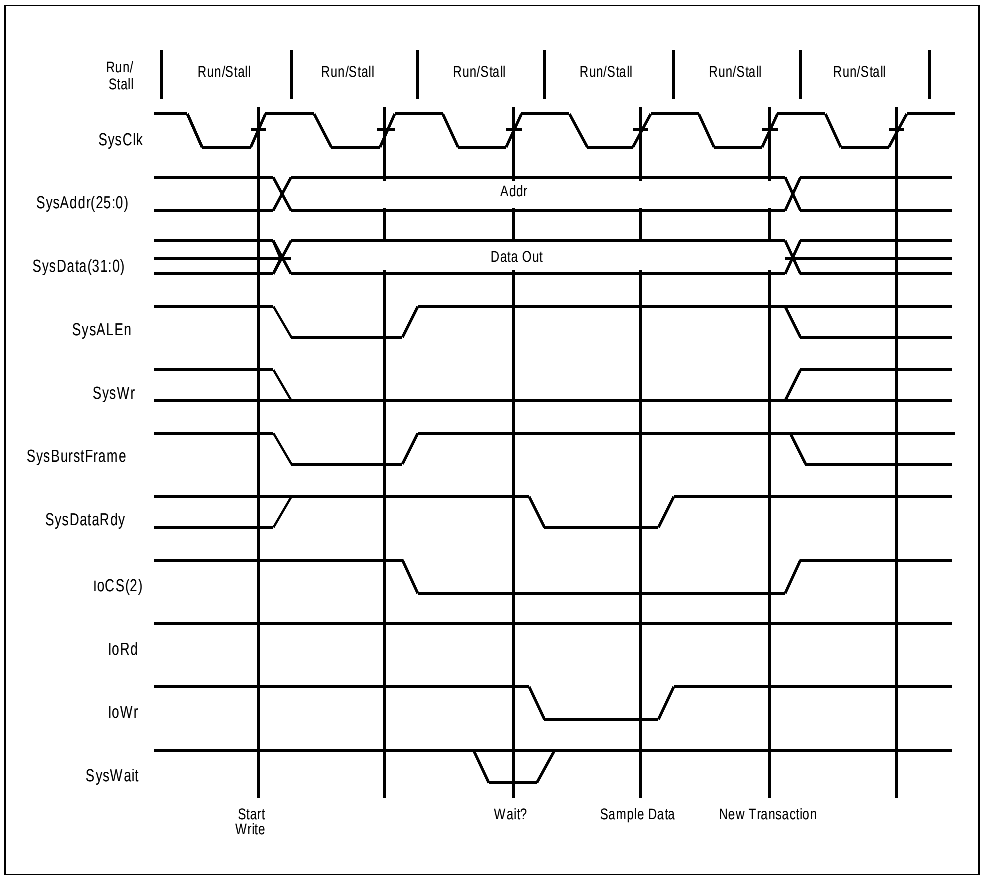


Figure 9.6 I-Type I/O Write with 0 Wait-States

Basic M-Type I/O Write with 0 Wait-States

Figure 9.7 illustrates a basic M-Type I/O Controller write transaction. Each transaction begins with both SysALEn and SysBurstFrame asserting. At this time, SysWr asserts (if it is not already in this state, as the result of a previous transaction). Now it is guaranteed that IoCS(), IoDStrobe (also known as IoRd), and IoRdHWr (also known as IoWr) will be in their de-asserted states. Assuming that there are no internally programmed StartRepeat wait-states, on the next clock cycle, SysBurst-Frame de-asserts; and IoCS() asserts. IoRdHWr (IoWr) also asserts here to indicate the I/O write transaction.

On the third cycle IoDStrobe (IoRd) asserts. And assuming that there are no internally programmed RdStart2Datum wait-states, SysDataRdy asserts to indicate that the data from the I/O device is being sampled into the RISController. On the 4th clock cycle, IoDStrobe (IoRd) de-asserts indicating that the read data from the I/O device has just been latched into the RISController. On the next clock--the final clock of the transaction--IoCS() and IoRdHWr de-assert and, at the same time, the next transaction may begin.

During an M-Type transaction, there is 1 clock of address setup time before IoCS() asserts. All signals are setup before the data strobe, IoDStrobe, asserts. After data has been sampled by the CPU, IoDStrobe de-asserts with all other signals having hold time. On the next clock, IoCS() de-asserts.

Note that because IoRdHWr asserts on the same clock edge as IoCS(), systems that require setup time from IoRdHWr to IoCs() can substitute SysWr for IoRdHWr. Many systems can also substitute IoCS() for IoDStrobe, which if the peripheral timing allows, may require fewer wait-states.

Notes

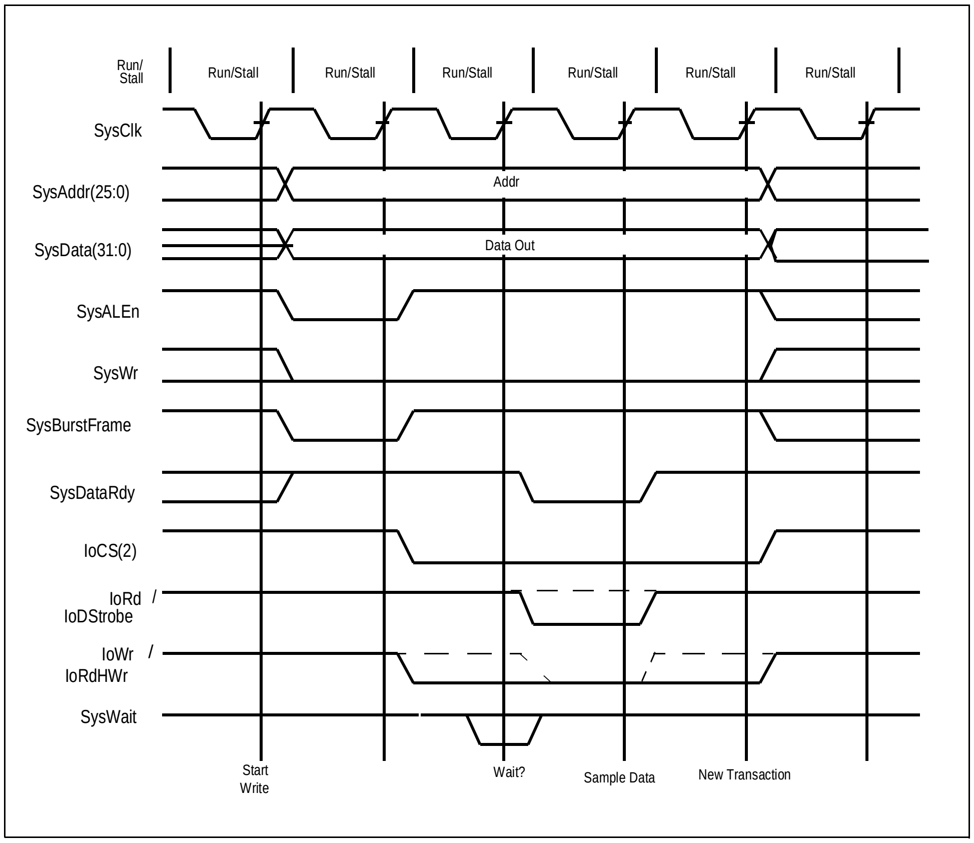


Figure 9.7 M-Type I/O Write with 0 Wait-States

Basic 16-bit PCMCIA-Style I/O Write with 0 Wait-States

Figure 9.8 illustrates a basic 16-bit PCMCIA-Style I/O Controller write transaction. Each transaction begins with both SysALEn and SysBurstFrame asserting. At this time, SysWr asserts (if it is not already in this state, as the result of a previous transaction). Now it is guaranteed that IoCS(), IoRd, and IoWr will be in their de-asserted states. And assuming there are no internally programmed StartRepeat wait-states, on the next clock cycle, SysBurstFrame de-asserts, and the IoCS() pair asserts. Note that on PCMCIA transactions the IoCS() pair is asserted according to which bytes are enabled and valid. Thus if the even byte is valid, then the even IoCS() will assert. If the odd byte is valid, then the odd IoCS() will assert. If both bytes are valid, then both IoCS() signals in the pair assert.

On the third cycle IoWr asserts. Assuming there are no internally programmed RdStart2Datum wait-states, SysDataRdy asserts to indicate that the data from the RISController is ready to be latched into the I/O device. On the fourth clock cycle, IoWr de-asserts, providing a means for the write data from the RISController to be latched into the I/O device. On the next clock cycle--the final clock of the transaction--the IoCS() pair de-asserts and, at the same time, the next transaction may begin.

Before the IoCS() pair asserts, during an I-Type transaction, there is 1 clock of address set-up time. All signals are set-up before the write strobe signal, IoWr, asserts. After data has been sampled by the CPU, IoWr de-asserts, with all other signals having hold time. On the next clock, the IoCS() pair de-asserts.

Note: The RC36100 inverts the value of Virtual Address(26:27) to generate the values of MemWrEn(1:0) during an I-Type transfer.

Notes

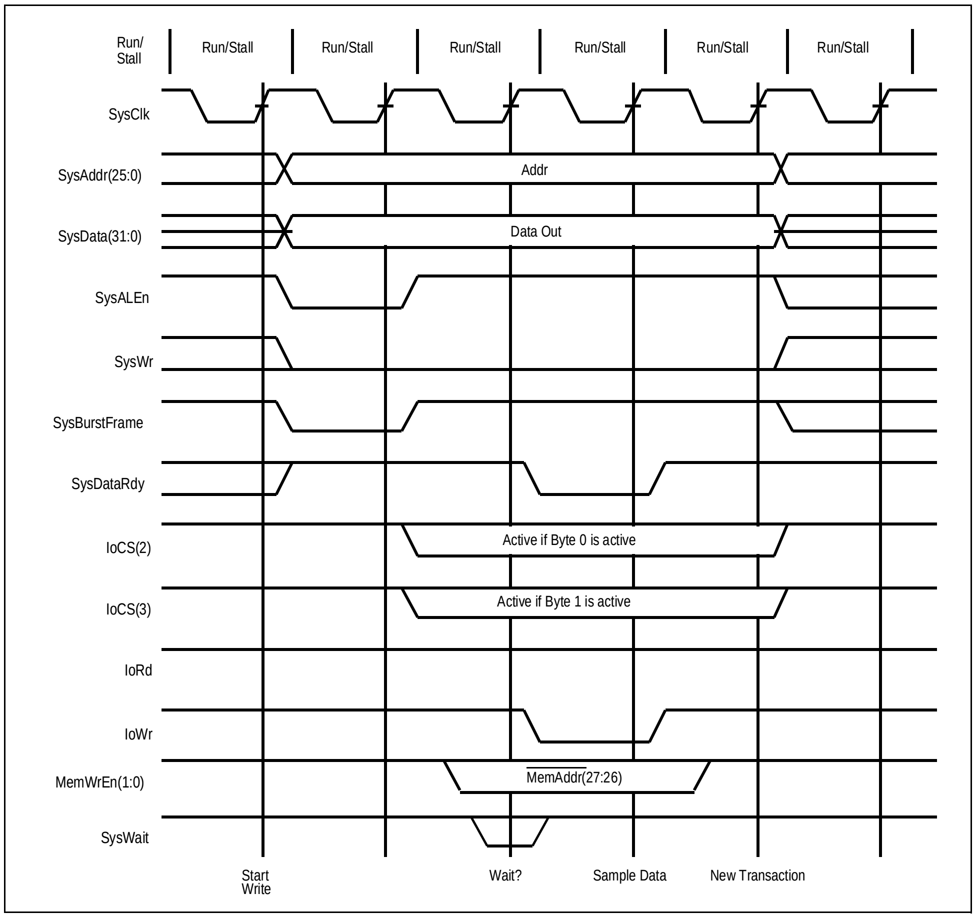


Figure 9.8 PCMCIA-Style I/O Write with 0 Wait-States

Read with Wait-State using Start Repeat Field

The left half of Figure 9.9 illustrates a basic I/O Controller read where 1 wait-state has been added by repeating the Start Cycle. Although Figure 9.9 shows an I-Type transaction, the same general timing concept applies to M-Type, and PCMCIA-style accesses as well. This special effect is programmed into the Wait-State Generator using the Start Repeat Field in the MemIoLSBWait-StateReg() Register. When the Start Cycle repeats, the IoCS() assertion is delayed. This effect is useful for very slow peripherals or peripherals that require significant address setup before the chip is selected. An example is the 600ns access time mode of the PCMCIA I/O protocol. The Start repeat Field affects both reads and writes.

Read with Wait-State using RdStart2Datum Field

The right half of Figure 9.9 illustrates a basic I/O Controller read where 1 wait-state is added using the RdStart2Datum Field of the MemIoMSBWaitStateReg() Register. Any number from 0 to 15 internal wait-states may be added using the RdStart2Datum Field. With this field, IoCS() and the read, write, or data strobe is asserted as normal, but then wait-states are added such that SysDataRdy is not asserted until the RdStart2Datum Field has finished counting. When SysDataRdy is asserted, then the Data from the external Memory Bank is sampled into the RISController.

Notes

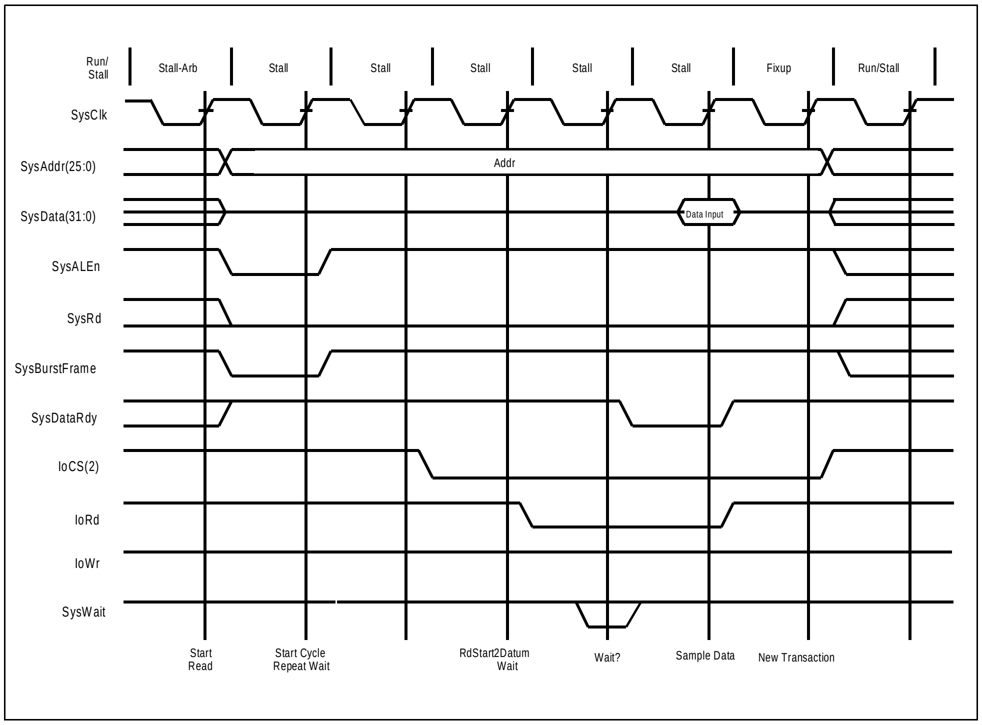


Figure 9.9 I/O Read with Internal Wait-States

Read with Wait-State using SysWait

Figure 9.10 illustrates a basic I/O Controller read where 1 wait-state is added using the external signal pin, SysWait. SysWait is not expected to be used for conventional I/O systems, since it is easier to program the Wait-State Generator to produce internal wait-states. However, SysWait can be useful for off-card I/O where there may be an indeterminate amount of time before the access can begin. Since SysWait is sampled a clock ahead of when it is used, its effect is seen two clocks later than when it is asserted. If SysWait is asserted when SysDataRdy is asserted, then an additional Data Sampling clock cycle is repeated with SysDataRdy remaining low. Thus external logic analyzers or other debug equipment may want to gate SysDataRdy with SysWait in order to decode valid Data samples.

Notes

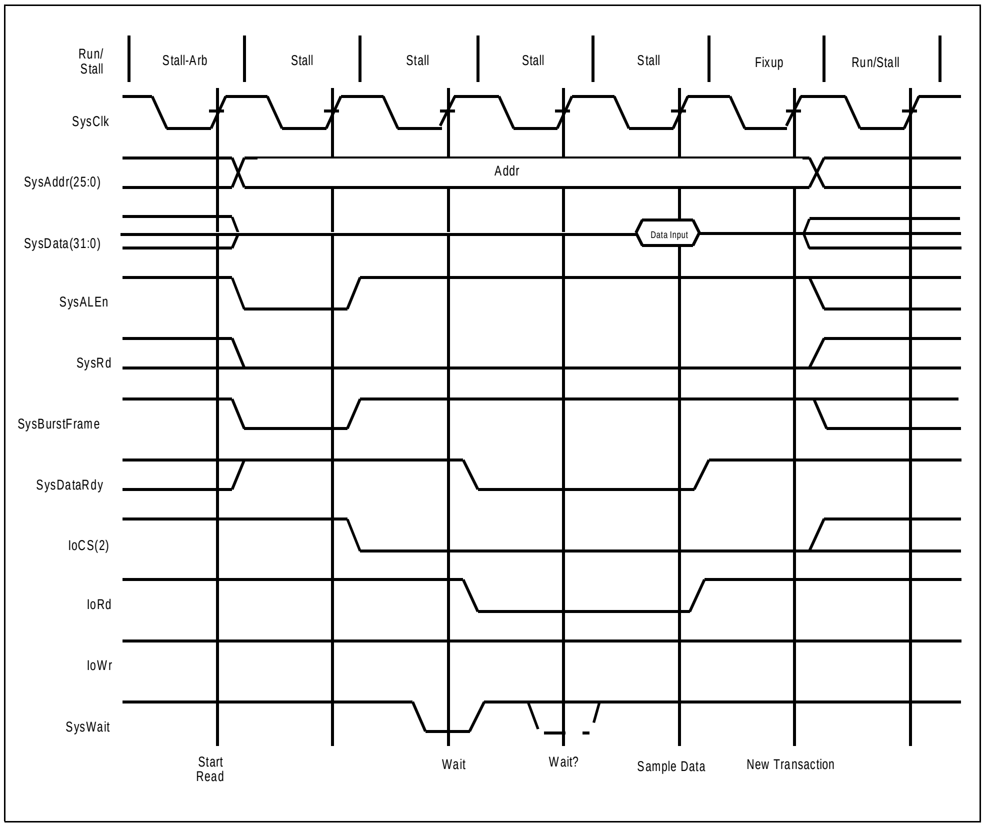


Figure 9.10 I/O Read with external SysWait Wait-State

1-Datum Write with Wait-State using StartRepeat Field

The right half of Figure 9.11 illustrates a basic I/O Controller write where 1 wait-state has been added by repeating the Start Cycle. This special effect is programmed into the Wait-State Generator using the Start Repeat Field in the MemIoLSBWaitStateReg() Register. When the Start Cycle repeats, the assertion of IoCS() is delayed. This effect is useful for very slow peripherals or peripherals that require significant address setup before the chip is selected. An example is the 600ns access time mode of the PCMCIA I/O protocol. The Start repeat Field affects both reads and writes.

1-Datum Write with Wait-State using WrStart2Datum Field

The left half of Figure 9.11 illustrates a basic I/O Controller write where 1 wait-state is added using the WrStart2Datum Field of the MemIoMSBWaitStateReg() Register. Any number from 0 to 15 internal wait-states may be added using the WrStart2Datum Field. With this field, the I/O Write Enable strobe, either IoWr or IoDStrobe is asserted as normal, but then wait-states are added where SysDataRdy is not asserted until the WrStart2Datum Field has finished counting. When SysDataRdy is asserted, then the Data from the external Memory Bank is sampled by external memory.

Notes

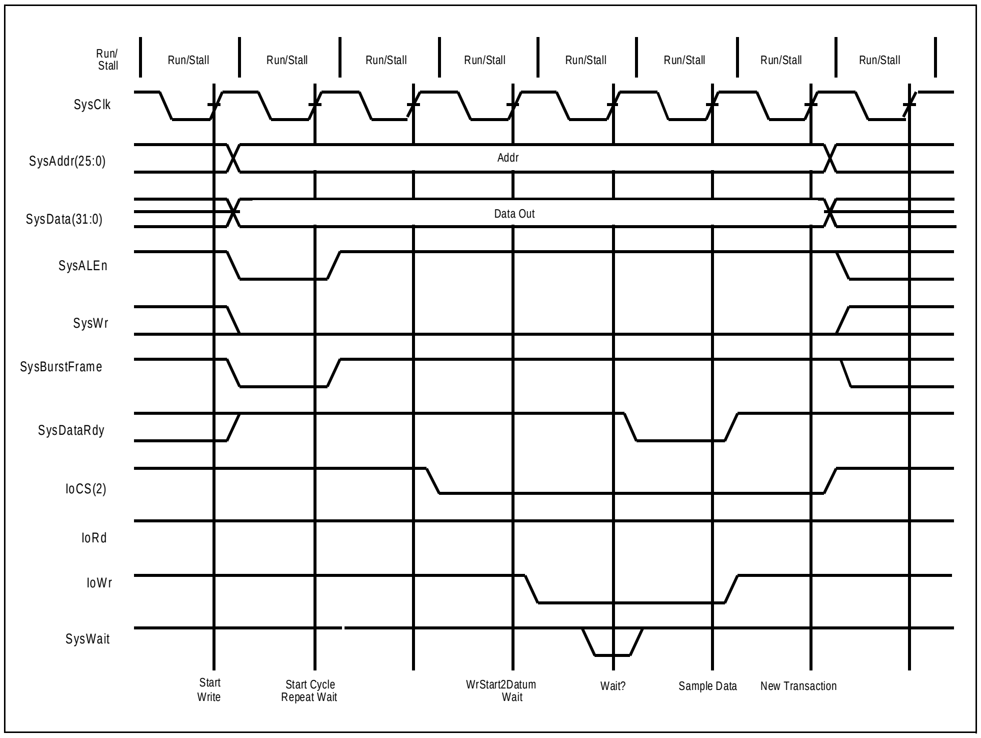


Figure 9.11 I/O Write with Internal Wait-States

1-Datum Write with Wait-State using SysWait

Figure 9.12 illustrates a basic I/O Controller write where 1 wait-state is added using the external signal pin, SysWait. SysWait is not Wait-State Generator to produce internal wait-states. However, SysWait can be useful for off-card peripherals where there may be an indeterminate amount of time before the access can begin. Since SysWait is sampled a clock ahead of when it is used, its effect is seen two clocks later than when it is asserted. If SysWait is asserted when SysDataRdy is asserted then an additional Data Sampling clock cycle is repeated with SysDataRdy remaining low. Thus external logic analyzers or other debug equipment may want to gate SysDataRdy with SysWait in order to decode valid Data samples.

Notes

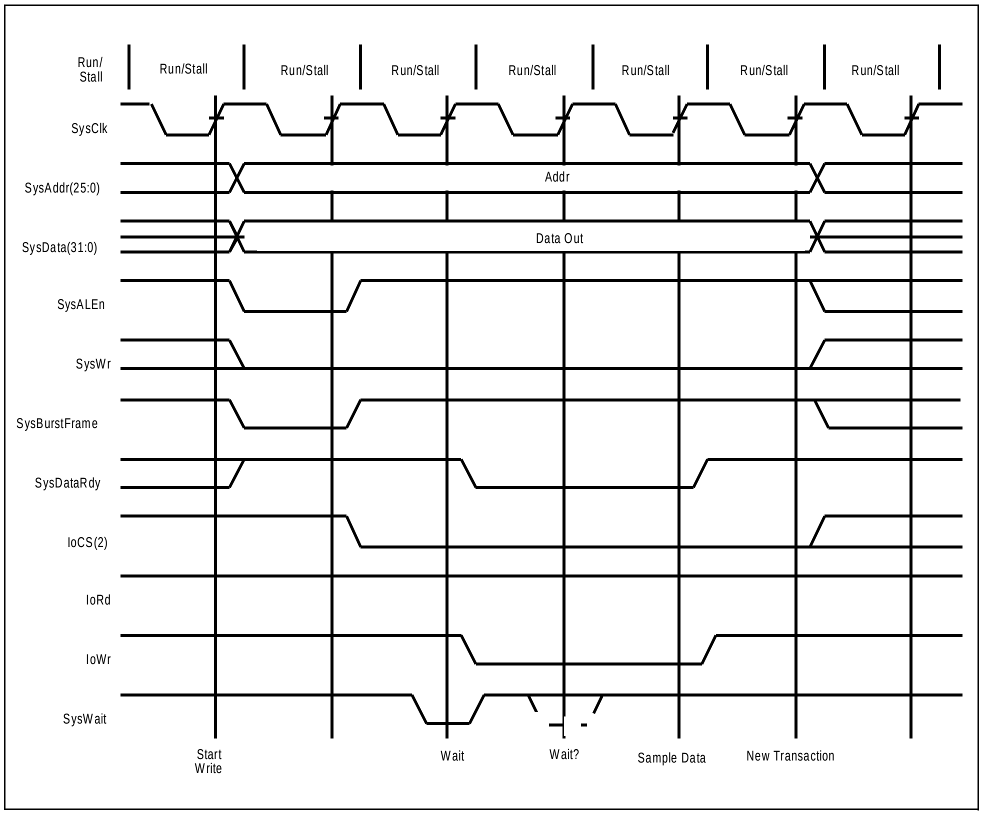


Figure 9.12 I/O Write with external $\overline{\text{SysWait}}$ Wait-State

System Examples

32-bit I/O Device Directly Connected to Bus

Figure 9.13 shows a typical 32-bit I/O device using the I-Type. And Figure 9.14 shows a typical 32-bit I/O device using the M-Type. In small systems, the I/O device can be attached directly to the SysAddr and SysData buses. If the current load is relatively large or the device turn-off time after a read is relatively long, the I/O device should be isolated with a transceiver.

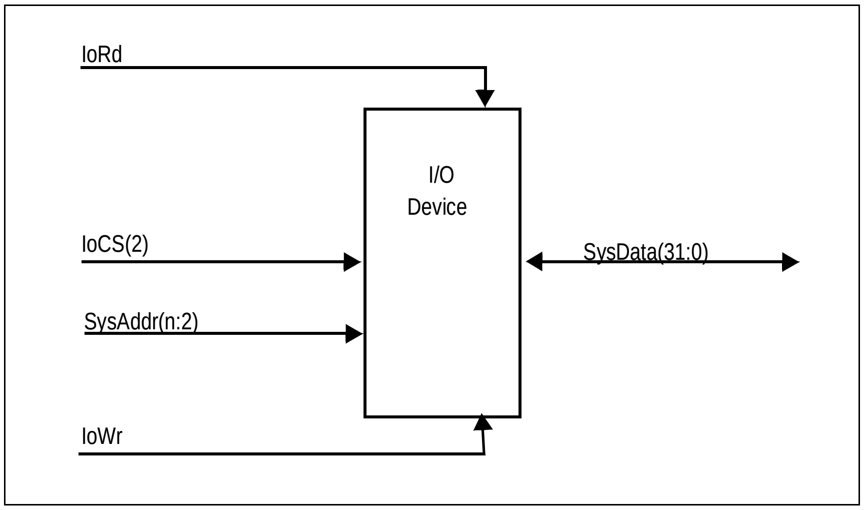


Figure 9.13 I-Type I/O System with Direct Bus Connection

Notes

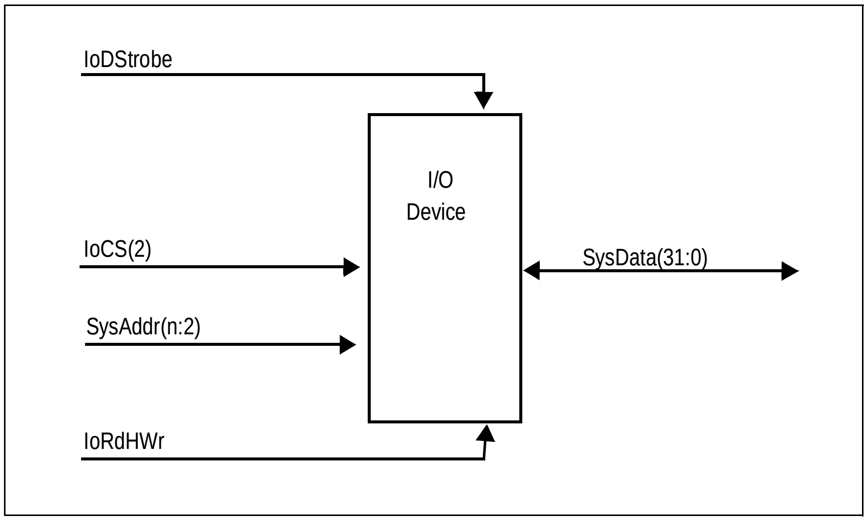


Figure 9.14 M-Type I/O System with Direct Bus Connection

I/O Reset Application

For generating a Reset, certain I/O device types require both Read and Write to be asserted. An M-state type can be used to do so, by dynamically switching from I-style to M-style and back again; however, this requires that no other devices be of M-style. If such devices must be used with M-style devices, then the Read and Write Reset device must externally gate its chip select with either the read or write line.

Other types of I/O devices require an active high reset legacy. This can either be accomplished with an external inverter of ResetN or with a spare PIO line.

32-bit I/O Device using 245 Transceivers

Figure 9.15 shows a typical I-Type I/O device using FCT245 transceivers. Figure 9.16 shows a typical M-Type I/O device using FCT245 transceivers.

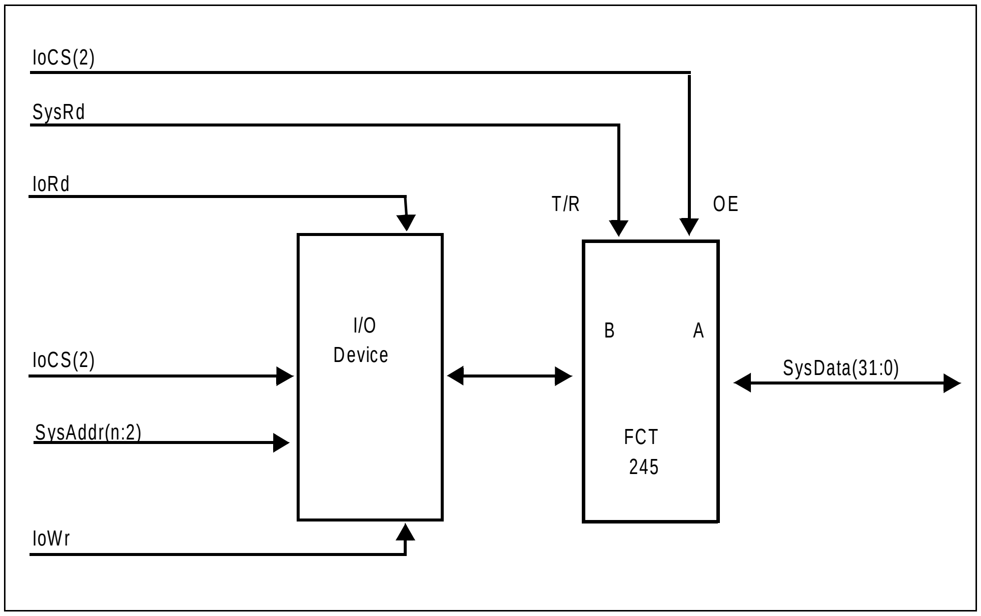


Figure 9.15 I-Type I/O System using FCT245 Transceivers

Notes

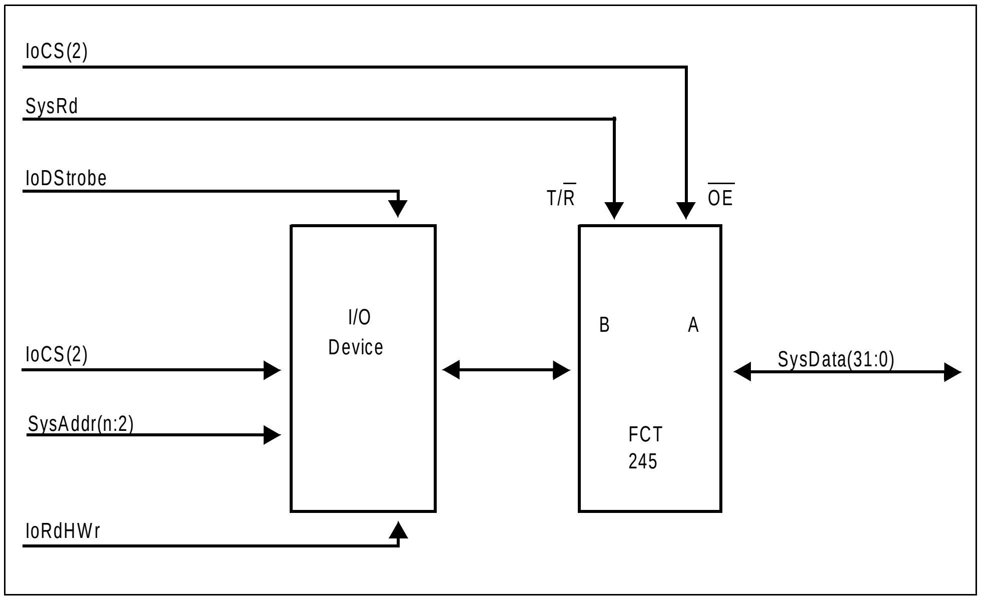


Figure 9.16 M-Type I/O System using FCT245 Transceivers

32-bit I/O Device using 543 Transceivers

Figure 9.17 shows a typical I-Type I/O device using FCT543 transceivers. This example system takes advantage of the dual output enable and chip select gating of the FCT543 where both the output enable and the chip select need to be asserted for the transceiver to drive its outputs. Figure 9.18 shows a typical M-Type I/O device using FCT543 transceivers.

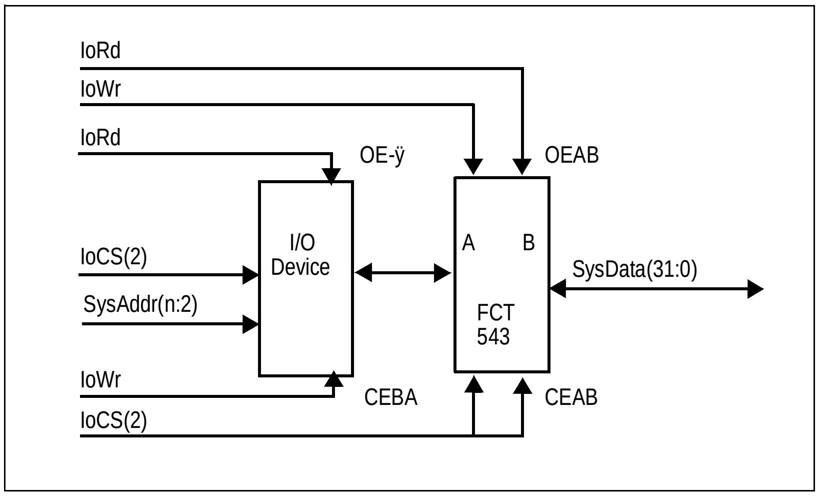


Figure 9.17 I-Type I/O System using FCT543 Transceivers

Notes

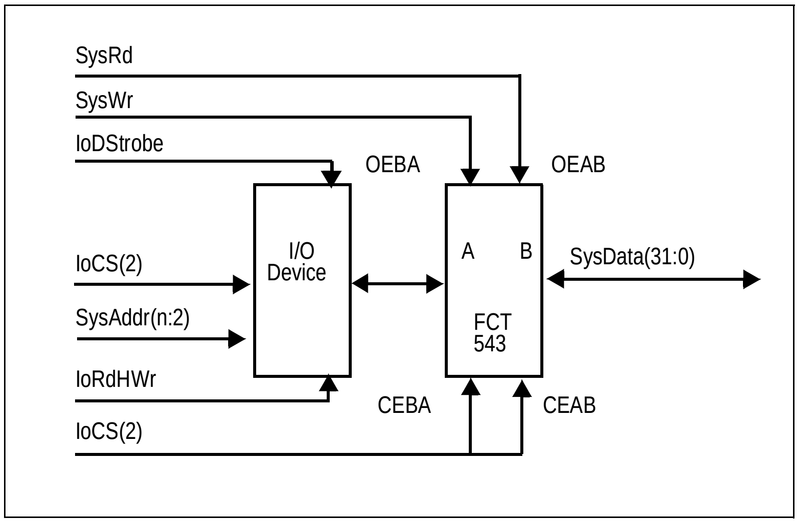


Figure 9.18 M-Type I/O System using FCT543 Transceivers

Using more than one device behind each transceiver

Multiple I/O devices can be put behind the same set of transceivers. The most obvious method is to add an external decoder to divide the chip select up into individual chips selects for each device using the MSB SysAddr line as the select. A second method is to use a spare RC36100 $\overline{\text{IoCS}}$() pin and assign it the same address spaces as the devices behind the transceiver. Thus the common $\overline{\text{IoCS}}$() combines all of the address spaces of the devices behind the transceiver, such that the transceiver is turned on for any of those devices.

16-bit I/O Devices

Figure 9.19 and Figure 9.20 show typical 16-bit I/O devices. There are two choices for hooking up 16-bit I/O devices.

For the first option, refer back to the 32-bit case in Figure 9.15 on page 19 and Figure 9.16 on page 20. The 16-bit device is word-aligned (using SysAddr bits n:2) even though it is a 16-bit device. This is the traditional MIPS connection and allows the device to be accessed from either endianess. For example, if the device is connected to SysData(15:0), then little endian software accesses the registers like 0x00, 0x04, 0x08, 0x0C,... and big endian software accesses the registers with a 0x02 offset like 0x02, 0x06, 0x0A, 0x0E,... For example, if the device is connected to SysData(31:16), then little endian software accesses the registers with a 0x02 offset with addresses like 0x02, 0x06, 0x0A, 0x0E,... and big endian software accesses the registers with addresses like 0x00, 0x04, 0x08, 0x0C,...

In the second option, the 16-bit device is halfword-aligned (using SysAddr bits n:1). With 16-bit systems it is imperative that the correct data line connections are made. Big Endian systems must attach SysData(31:16) and Little Endian systems must attach SysData(15:0). For example with half-word aligned connections, the software accesses the registers with addresses like 0x00, 0x02, 0x04, 0x06.

Notes

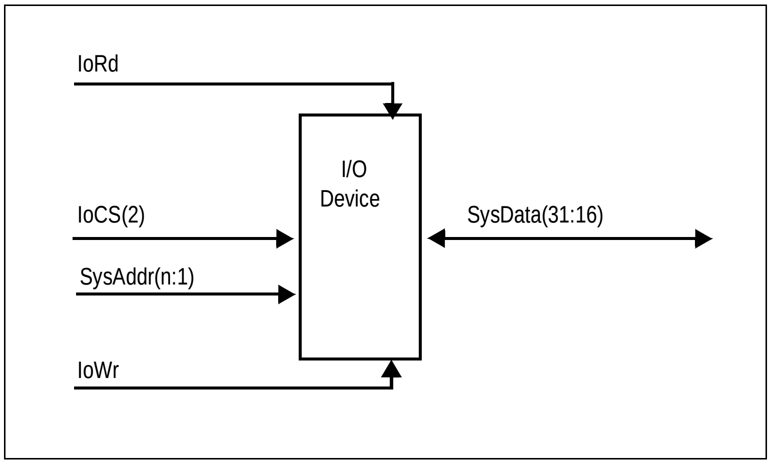


Figure 9.19 16-bit I/O System with Big Endian Connection

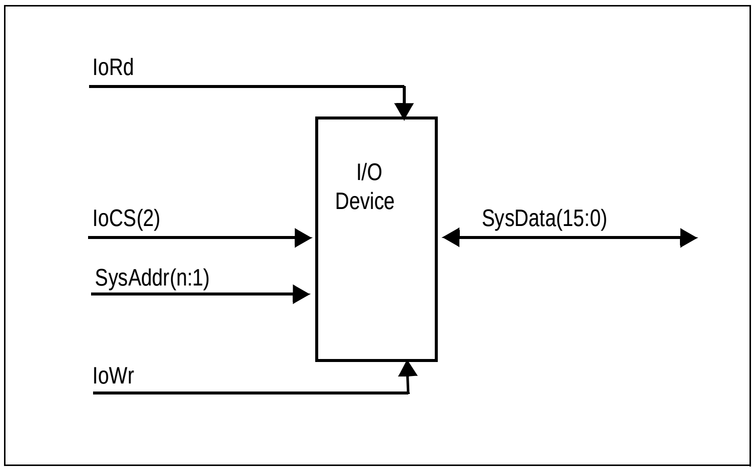


Figure 9.20 16-bit I/O System with Little Endian Connection

8-bit I/O Devices

Figure 9.21 and Figure 9.22 show two typical 8-bit I/O device systems. There are two choices for hooking up 8-bit I/O devices.

In choice 1, refer back to the 32-bit case in Figure 9.15 on page 19 and Figure 9.16 on page 20. The 8-bit device is word-aligned (using SysAddr bits n:2) even though it is a 8-bit device. This is the traditional MIPS connection and allows the device to be accessed from either endianess. For example, if the device is connected to SysData(7:0), then little endian software accesses the registers like 0x00, 0x04, 0x08, 0x0C,... and big endian software accesses the registers with a 0x03 offset like 0x03, 0x07, 0x0B, 0x0F,... For example, if the device is connected to SysData(31:24), then little endian software accesses the registers with a 0x03 offset with addresses like 0x03, 0x07, 0x0B, 0x0F,... and big endian software accesses the registers with addresses like 0x00, 0x04, 0x08, 0x0C,...

In choice 2, the 8-bit device is byte-aligned (using SysAddr bits n:0). With 8-bit systems it is imperative that the correct data line connections are made. Big Endian systems must attach SysData(31:24) and Little Endian systems must attach SysData(7:0). For example with half-word aligned connections, the software accesses the registers with addresses like 0x00, 0x01, 0x02, 0x03.

Since an 8-bit system is probably a smaller system, the SRAM/ROM can be attached directly to the SysAddr and SysData buses. In larger systems, FCT245 transceivers can be added between the memory bank and the SysData bus. Also in large systems, the SysAddr bus can also be buffered using FCT244 buffers.

Notes

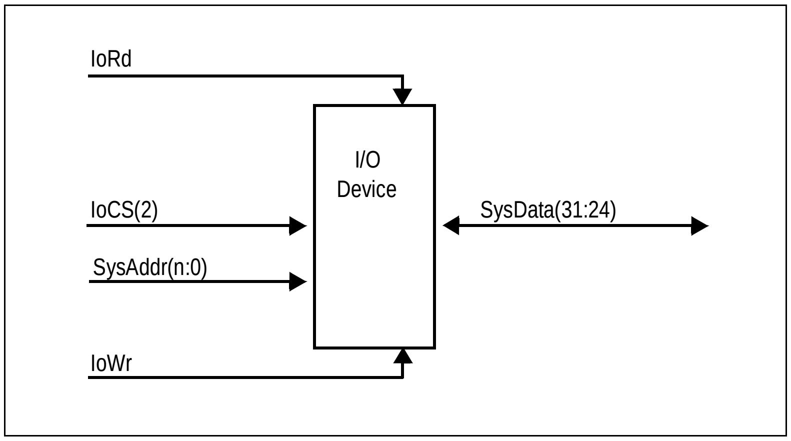


Figure 9.21 8-bit I/O System with Big Endian Connection

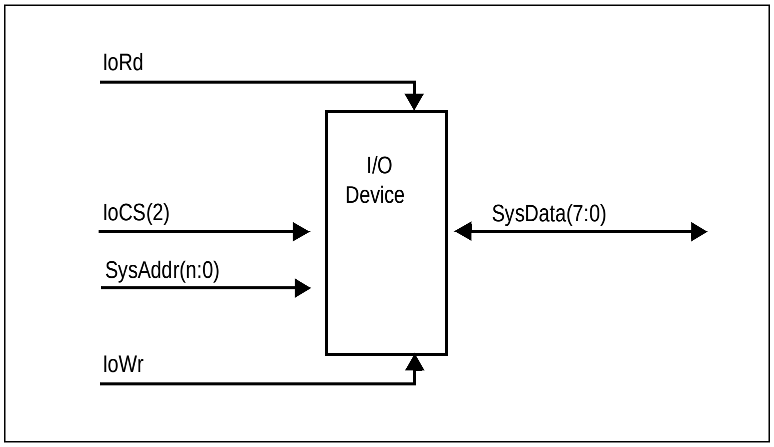


Figure 9.22 8-bit I/O System with Little Endian Connection

Notes

DRAM Controller

Notes

Introduction

The IDT79RC36100 RISController integrates bus controllers and peripherals around the RISCore3000 family CPU core. One of the four on-chip bus controllers in the RC36100 is the **DRAM Controller**.

This chapter provides an overview of the DRAM Controller interface, complete pin and signal timing descriptions as well as an explanation on how the DRAM Controller interface relates to typical external hardware DRAM systems.

Features

- *Controls up to 4 banks of Page-Mode DRAMs*
- *Each bank pair programmable to Interleaved or non-Interleaved mode*
- *Each bank programmable to use 1M, 4M, or 16M DRAM chips*
- *Each bank programmable to 32-bit or 16-bit Mode*
- *Provides jumper-less 16-bit to 32-bit or Interleaved upgrade*
- *Built-in $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ Refresh Timer*
- *Wait-State Generator features:*
  - *Programmable time from start to finish of each data access for each area*
  - *Programmable time options for Reads and Writes*
  - *Programmable time options for Single and Burst Accesses*
  - *Internally generates the RdCEnN and AckN timing for all CPU accesses*
- *Direct Control of Data Path Transceivers include:*
  - *Direct Bus Connection*
  - *FCT260 Bidirectional Bus Exchanger Multiplexer*
  - *FCT245 Bidirectional Transceiver*
  - *FCT543 Bidirectional Registered Transceiver*

Block Diagram

The functional block diagram of the DRAM Controller is shown in Figure 10.1. Located at the bottom of Figure 10.1, the DRAM Control Signal State Machine is responsible for generating the basic control signals used to connect external DRAM chips and their transceivers. These signals include row and column address strobes, read enables, and write enables. The DRAM Controller as a whole works in cooperation with the Bus Interface Unit described in Chapter 7. Thus, the Control Signal State Machine sends and receives information from the BIU Controller for assistance with controlling the port width and controlling partial word reads and writes.

The Control Signal State Machine also uses information stored in the software programmable DRAM Controller Register Bank, for example, to control FCT260-Type versus FCT245-Type (transceiver interface) accesses. In addition to the Control Signal State Machine, there is a Refresh Timer and State Machine. The refresh circuitry implements $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh timing as required by conventional page-mode DRAMs.

The DRAM Controller Wait-State Generator is shown in the center of Figure 10.1. The Wait-State Generator takes care of sending and receiving information from the BIU Controller in order to control the sequencing and timing of reading and writing each individual data. The number of wait-states is derived from the settings programmed into the Register Bank. Once the correct number of wait-states has been counted out, the Wait-State Generator sets the appropriate internal BIU Acknowledge signals. The programmable Wait-State Generator eliminates the need for external state machines that are traditionally used for this function.

Notes

The DRAM Controller Decoder is shown at the top of Figure 10.1. The decoder constantly monitors the Bus Interface Unit's address and data bus to see if (1) the access is to the DRAM Controller's Register Bank, or, (2) the access is in one of the DRAM Controller's Chip Select areas that are responsible for controlling the bus transaction.

The DRAM Address Multiplexer is also shown at the top of Figure 10.1. The DRAM multiplexer switches the address lines between the MSB row address and LSB column address as required by conventional page mode DRAM chips. The multiplexer also includes address options to allow seamless upgrades from 16-bit to 32-bit or to interleaved 32-bit systems. The row address is also stored and compared using the Page Comparator circuitry. The Page Comparator allows the page mode DRAMs to enter into their faster page access mode whenever consecutive locations are accessed in the same block of memory.

The DRAM Controller Register Bank is shown at the left in Figure 10.1. The Register Bank allows the software programmer access to the many different options of the DRAM Controller. The chip select address ranges, the number of wait-states, the port-width of the chip select, and other similar options are programmed into the Register Bank as part of the software initialization sequence of the boot operating system.

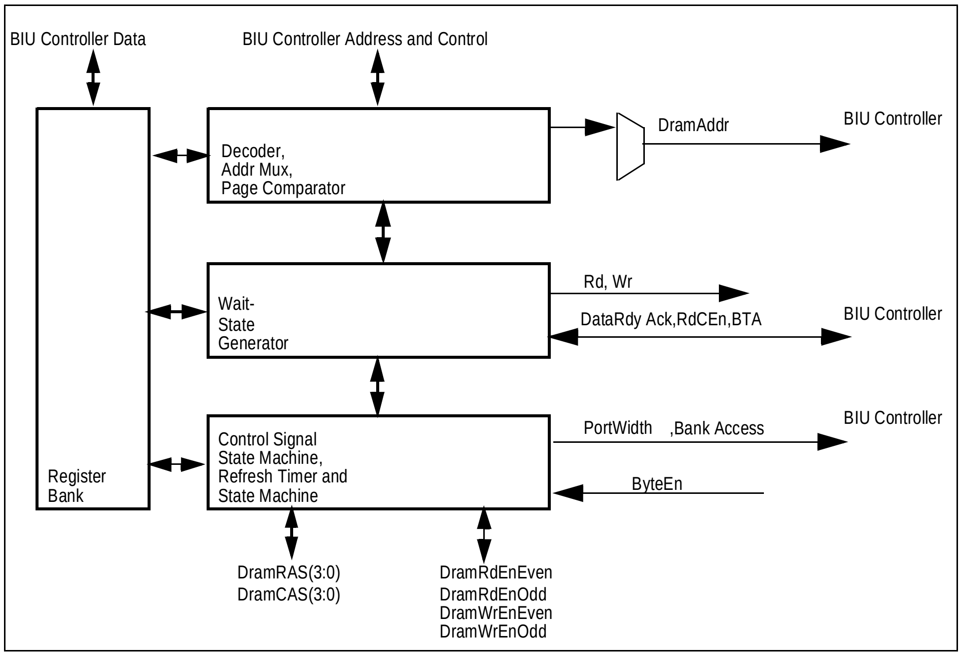


Figure 10.1 RC36100 DRAM Bus Controller Block Diagram

DRAM Bus Controller Interface Signals

The following external pins are typically attached directly from the RC36100 RISController to external DRAM devices and their transceivers:

SysAddr(13:2) Output

The System Address provides the byte multiplexed address for DRAMs. This allows maximum of 16M words of unique locations to be accessed, thus providing a maximum of 64MBytes of memory for each bank. These signals share 12 of the lower 26 system address pins, SysAddr(13:2). Whenever a bus transaction is decoded to be a DRAM access, the behavior of these pins change and they act as DRAM-style multiplexed Row- and Column- address lines. Address assignments within the address multiplexer are such that a 16-bit system can be upgraded to a 32-bit system without external jumpers. In addition address assignments within the address multiplexer are such that single bank non-interleaved systems can be upgraded to pair-wise interleaved systems without external jumpers.

Notes

DramRAS(3:0) Output

DRAM Row Address Strobes are active low outputs used to strobe the row address into the DRAM. Each DramRAS() signal drives one bank of DRAM.

DramCAS(3:0) Output

DRAM Column Address Strobes are active low outputs used to strobe the column address into the DRAM. If the system uses a 16-bit wide bus instead of a 32-bit wide bus, then DramCAS(3:2) are used for a big endian system, while DramCAS(1:0) are used for a little endian system.

DramWrEnEven Output

DRAM Write Enable for Even Bank is an active low output signal used to write the selected DRAM bank 0 or 2. "Early write" cycles are used, so the byte selection is done by activating the leading edge of appropriate DramCAS() signals. Note that the DRAM specific write enables must be used instead of SysWr or MemWrEn(3:0) because refreshes may occur simultaneously with Memory Controller writes which could potentially cause 4M-16Mbit DRAMs to enter a test mode.

DramWrEnOdd Output

DRAM Write Enable for Odd Bank is an active low output signal used to write the selected DRAM bank 1 or 3. "Early write" cycles are used, so the byte selection is done by activating the leading edge of appropriate DramCAS() signals.

Note that the DRAM specific write enables must be used instead of SysWr or MemWrEn(3:0) because refreshes may occur simultaneously with Memory Controller writes, which could cause 4M-16Mbit DRAMs to enter a test mode.

DramRdEnEven Output

DRAM Read Enable for Even Bank is an active low output signal that is used to control the enabling of DRAM bank 0 or 2. Typically, DramRdEnEven is attached to the DRAM bank data transceiver output enable of banks 0 and 2, while DramCAS() controls the output enabling between the DRAM chips on the corresponding byte lane of bank 0 and 2.

In FCT260 type systems, DramRdEnEven is used as the overall DramRdEn path enable.

In FCT245 type systems, DramRdEnEven asserts on both reads and writes as a DramEnEven even bank transceiver enable.

DramRdEnOdd Output

DRAM Read Enable for Odd Bank is an active low output signal used to control the enabling of DRAM bank 1 or 3. Typically DramRdEnOdd is attached to the DRAM bank data transceiver output enable of banks 1 and 3, while DramCAS() controls the output enabling between the two banks on each of the corresponding byte lanes. In FCT260-type systems, DramRdEnOdd is used as the overall DramRdPathSel path select. In FCT245-type systems, DramRdEnOdd asserts on both reads and writes as a DramEnOdd odd bank transceiver enable.

BIU Controller Signals

The BIU Controller Signals are used to complete the DRAM interface. These signals are also listed here for reference. Information specific to the DRAM Controller is given here and general information about the signal is given in Chapter 7, "System BIU Controller."

SysData Output/Input

System Data Bus: A 32-bit peripheral connects the entire 32-bit SysData bus to its data pins or to its transceivers. 16-bit systems use the halfword offset address, A(x:1). (Note that the corresponding SysAddr() line is not SysAddr(x:1), since the DRAM address mux starts with SysAddr(2)). 8-bit systems can use the byte address, A(x:0). *In this case, 16-bit DRAMs connect to particular data pins depending on whether the Endianness of the system is Big Endian or Little Endian.* Thus 16-bit DRAMs use SysData(31:16) if they are Big Endian and SysData(15:0) if they are Little Endian. The User Mode Reverse Endianess Bit in the CP0 Status Register has no effect on the connections to SysData, however, it strongly recommended that the Reverse Endianess Bit not be used to "correct" an endianess connection as it does not function in the kernel mode address space.

Notes

SysALEn **Input/Output**

SysALEn on a DRAM Controller initiated access indicates the beginning of the transaction. However, because of various precharge, RAS-left asserted, and refresh conditions, the actual DRAM access may or may not begin immediately.

SysWait **Input**

System Wait: The SysWait signal can be used in a debug mode, during DRAM Controller accesses. It requires that the BurstAck field in the DRAM MSB Control Register be turned off. Whenever DramCAS() is asserted, if SysWait is asserted before the next falling edge of SysClk, wait-states will occur. SysWait must then be held with hold time relative to SysClk rising.

Overview of the DRAM Controller

The RC36100 RISController's DRAM Bus Controller supports a maximum of four individual banks of standard RAS/CAS controlled page-mode DRAM chips. Each bank can have a minimum of 2x256KBytes (16-bit) and a maximum of 4x16MBytes (32-bit) of memory. Each bank can individually be programmed for 32-bits or 16-bits. Each pair of banks can be programmed to be non-interleaved or pair-wise interleaved. Thus the system as a whole can support up to four banks of DRAM and anywhere from 512KBytes up to 256MBytes.

The RC36100 DRAM Controller, in addition to the RAS and CAS control lines, provides transceiver enable pins, and an address multiplexer (mux). The DRAM Controller also provides software configured options for wait-states as well as for CAS-before-RAS refresh timing.

Address mapping

All four banks must be contained within a single 256MByte address space. Then, by programming the Base and Page Mask Registers of each bank, they can be individually mapped anywhere within the selected DRAM-designated 256MByte address space. This allows systems with mixed sized banks to have a contiguous memory space.

Although mixed sized banks can be contained within a pair of DRAM banks, banks with different port widths (32- or 64-bit) must be contained in different pairs of banks (16-bit banks may not be mixed with other port width sized banks). Cacheability or non-cacheability of the references to these memory banks depends as to which virtual segment they are mapped to, as per the RC36100's memory map and CP0 Cache Control Register. This choice designates which references will be serviced as burst or non-burst references.

32-bit and16-bit mode support

Each bank of DRAM memory can be programmed individually as either an interleaved, 32-bit, or a 16-bit bank. A two bit'PortWidth' field is provided in the DRAM configuration register, one for each bank, which will select between the port width configuration options.

In 32-bit mode, the DRAM interface behaves similar to the 32-bit interface of the RC3051 family in that address bits (1:0) are ignored as per word-aligned addressing. Thus address lines A(1) & A(0) are mapped out of the address multiplexing generation. Single word reads/writes are treated as one 32-bit datum. Partial word reads or writes are treated as single partial word read or write cycles by activating appropriate CAS signals, depending on the endianess of the processor. For block reads, the controller does four DRAM reads, back-to-back, to bring four words into the processor, using the page mode feature of DRAM.

The 16-bit mode is treated slightly different from the 32-bit mode. Most importantly, address line A(1) is included in the address multiplexer in order to support halfword-alignment. In case of 16-bit read/write instructions and data, the controller treats them as halfword mini-burst/burst read or write cycles, activating appropriate CAS signals based on the endianness of the device, DramCAS(3:2) for big endian and DramCAS(1:0) for little endian.

The mini-burst is continued until all halfword datum are either read or written. Data will be driven on SysData(31:16) lines if it is a big-endian system or on SysData(15:0) if it is little-endian system. In cases of byte read or write instructions, the controller will activate appropriate CAS signals depending on the endianess of the processor.

Notes

With 16-bit mode block read accesses, the controller will perform 8 back to back reads in burst mode. This will be treated as a single burst transaction, bringing 16 bits of data every datum. The number of wait-states added between each datum can be programmed to differ from the number of wait-states added to the first datum, such that page mode DRAMs have optimal timing.

When the processor wants to read or write 32-bit data from a 16-bit memory, two 16-bit datum transfers occur back-to-back within the same address transaction, which is called a mini-burst write. The number of wait-states added between burst datum can be programmed to differ from the number of wait-states added to the first datum, such that page-mode DRAMs have optimal timing. Note, though, that 16-bit mode banks may not be mixed with other 32-bit or interleaved banks.

The interleaved mode is treated slightly different from the 32-bit mode. Most importantly, address line A(2) is ignored, along with A(1:0), per doubleword-aligned addressing. In essence, two words are simultaneously read from two separate banks; however, the second word must wait 1 extra clock before it can be latched into the CPU. Depending on the access time of the DRAM, this saves 1-2 clocks per every pair of words read. In addition, if every other word is latched, the CPU can pipeline the address for the second pair of words, a clock early, for additional savings.

Types of memory supported

The following three types of memory cycles are supported by the controller:

- ***Page Mode support**. The page mode feature of the DRAM Controller is always enabled. In the case of a mini-burst or burst refill, the page mode is used to obtain data by use of an octi (16-bit) or quad (32-bit) word read. In the cases specified in a later section, "Page Comparator Algorithm," $\overline{\text{RAS}}$ will be left active, expecting a subsequent page mode access to the same block of memory. The controller has an on-chip page register and comparator which uses the programmed DRAM density to determine whether or not a given access can take advantage of page mode; as well as whether or not to leave $\overline{\text{RAS}}$ asserted at the end of the transaction.*
- ***Non-Interleaved support.** At any given time in the non-interleaved mode, only one bank will have an active $\overline{\text{RAS}}$. In the case of an access to a different bank, first the $\overline{\text{RAS}}$ of the active bank will be deactivated and then the $\overline{\text{RAS}}$ of the accessed bank will be activated. In case of a page miss within the same bank, $\overline{\text{RAS}}$ will deactivate for a precharge period, a new page address will be strobed in by driving the new row address, and then re-activating $\overline{\text{RAS}}$.*
- ***Interleaved support.** A programmable option field is provided in the configuration register which will enable or disable two-way interleaving. Various interleaving sub-options allow various types of transceivers to be used by changing the functionality of the controller's output enables.*

Programmable wait state generation

A programmable wait state feature is supported by:

- *$\overline{\text{RAS}}$ Precharge, Row Addr Setup, and Row Addr Hold settings*
- *$\overline{\text{CAS}}$ Addr Setup/Precharge setting*
- *$\overline{\text{CAS}}$ Pulse Width and $\overline{\text{CAS}}$ Addr Hold settings*

Page Comparator Algorithm

An internal page comparator compares the page address of consecutive DRAM bus cycles. After completing the current DRAM bus access, $\overline{\text{RAS}}$ is held asserted after:

- *Writes*
- *Single Word Reads*

When asserted, $\overline{\text{RAS}}$ is de-asserted if:

- *refresh occurs*
- *non-page write occurs*
- *non-page read occurs*

Notes

The page comparator is not affected by non-DRAM accesses. It is assumed that uncached reads and writes are unlikely to be done to DRAM, thus the distinction between instruction and data is not statistically important to throughput. Also, the maximum assertion time for $\overline{\text{RAS}}$ is assumed to be covered by the occurrence of refreshes.

Unaligned page accesses

Since long bursts are always aligned to the burst length, bursts across non-page boundaries are not possible from the CPU or internal DMA channels. Therefore, unaligned page accesses will not occur. There is one possible exception: if external DMA with long burst access does a non-aligned burst, then a page boundary crossing is possible. Most DMA agents capable of long bursts (for example, 16 words) also pre-align or are capable of pre-aligning the burst on a boundary (for example, align to a 16 word boundary).

Refresh Timing

The $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh mode is supported. The refresh rate is programmable in order to take into account the speed of the processor.

Initialization

The system boot software is responsible for initializing the DRAMs after reset. The DRAM Controller is guaranteed to hold all DRAM control signals de-asserted until a proper DRAM cycle is initiated by the user. Usually initialization involves the software OS to program a wait of 200us after power up (reset), initializing all of theDRAM control registers, and then doing 8 refresh cycles. Because the refresh period is programmable, the boot code can temporarily set the refresh period to a very small value, to complete the 8 refresh cycles quickly.

Programmable features

The DRAM Controller has the following programmable features:

- *Page Size*
- *$\overline{\text{RAS}}$ assertion selection*
- *$\overline{\text{RAS}}$ precharge time*
- *$\overline{\text{RAS}}$ Addr Setup and Addr Hold time*
- *$\overline{\text{CAS}}$ precharge/Addr Setup time*
- *$\overline{\text{CAS}}$ Addr Hold Time on Writes (WrBTA)*
- *$\overline{\text{CAS}}$ Pulse Width*
- *Internal Burst Ack generation*

Signal Control Interface

The DRAM Controller provides read enables and write enables that are suitable for direct chip connection. The read and write enables can in general also be attached to FCT260, FCT245, and FCT543 type transceivers.

Wait State Generator

The Wait-State Generator controls the speed of the DRAM accesses to and from the Bus Interface Unit Controller. This includes the time from the start of a DRAM transaction until the first datum is sent or received. The Wait-State Generator also is programmed to generate the internal RdCEnN and AckN signals for CPU read and write requests.

The internal Acknowledge signal, “AckN” (as described in Chapter 7), is the same as the external signal pin that the RC3051 RISController family uses. On single word reads and on both single word and burst writes, AckN is automatically placed at the end of the transaction by the Wait-State Generator. Burst DRAM read operations return AckN earlier than the end of the transaction (because of the read buffer); thus, a Control Register 'Burst Ack' field is provided.

Notes

Register Option Field Programmability

The DRAM Controller contains 4 sets of registers, one set for each chip select, $\overline{\text{DramRAS(3:0)}}$. There is also a global set of registers for the address multiplexer options and refresh timing options. These registers allow the DRAM Controller to be configured for different speeds and types of DRAM chips; therefore, almost any system speed/cost/manufacturing trade-off can be accommodated.

Register Descriptions

Table 10.1 provides the address map for the DRAM Controller registers. Note that Big Endian software must offset these addresses by b'10 (0x2), if halfword operations are used. For correct operation, note that all registers must be programmed before use.

Phys. Address	Register
0xFFFF_E100 0xFFFF_E104	DRAM Refresh Count Register DRAM Refresh Compare Register
0xFFFF_E110 0xFFFF_E114	DRAM RAS Multiplexer Select Register for Pair 1:0 DRAM RAS Multiplexer Select Register for Pair 3:2
0xFFFF_E120 0xFFFF_E124	DRAM CAS Multiplexer Select Register Pair1:0 DRAM CAS Multiplexer Select Register Pair 3:2
0xFFFF_E180 0xFFFF_E184 0xFFFF_E188 0xFFFF_E18C	DRAM MSB Base Address Register for Bank 0 DRAM MSB Bank Mask Register for Bank 0 DRAM LSB Control Register for Bank 0 DRAM MSB Control Register for Bank 0
0xFFFF_E190 0xFFFF_E194 0xFFFF_E198 0xFFFF_E19C	DRAM MSB Base Address Register for Bank 1 DRAM MSB Bank Mask Register for Bank 1 DRAM LSB Control Register for Bank 1 DRAM MSB Control Register for Bank 1
0xFFFF_E1A0 0xFFFF_E1A4 0xFFFF_E1A8 0xFFFF_E1A	DRAM MSB Base Address Register for Bank 2 DRAM MSB Bank Mask Register for Bank 2 DRAM LSB Control Register for Bank 2 DRAM LSB Control Register for Bank 2
0xFFFF_E1B0 0xFFFF_E1B4 0xFFFF_E1B8 0xFFFF_E1BC	DRAM MSB Base Address Register for Bank 3 DRAM MSB Bank Mask Register for Bank 3 DRAM LSB Control Register for Bank 3 DRAM MSB Control Register for Bank 3

Table 10.1 DRAM Controller Registers.

DRAM Refresh Count Register ('DramRefreshCountReg')

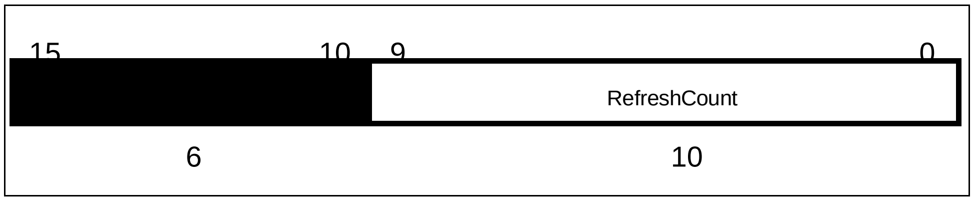


Figure 10.2 DRAM Refresh Count Register ('DramRefreshCountReg')

Notes

BIT	Function
9:0	RefreshCount

Table 10.2 DRAM Refresh Count Register (DramRefreshCountReg’) Bit Assignments.

The lower 10 bits form a 10-bit binary up-counter. The Count register, shown in Figure 10.2, ticks upward on each system clock. When Count equals Compare, the DRAM Controller will initiate a refresh sequence and the Count register will be reset back to 0. The upper 6 bits are reserved to be “0”. The default value of the DRAM Refresh Count Register, shown in Table 10.2, is 0x0000 at reset. The register is both readable and writable.

Staggered Refresh

In order to reduce the amount of peak instantaneous current and intra-bus transaction average current used by refreshing DRAMs, refresh is done by refreshing (RAS’ing) Banks 0 & 2 together, then afterwards refreshing (RAS’ing) Banks 1 & 3 together.

Refresh Arbitration

Refreshes on the RC36100 must obtain the DRAM Controller before doing a refresh. DRAM systems in general must not use $\overline{\text{MemWrEn}}$ or $\overline{\text{SysWr}}$, because some other peripheral driving them low during a refresh would accidently put some types of DRAM chips in their internal test mode. If the CPU or DMA channel tries to access DRAM at the same time as a Refresh, they will wait for the Refresh to finish.

Panic Mode Refresh Application

Ordinarily it is only possible to have a 4-word burst DMA after which the refresh controller can regain the bus and do a pending refresh. However, external DMA can burst up to a system defined length. In such a situation, multiple (depending on the application) refresh ticks may be missed. If this is of concern to the system designer, they can either (1) divide the external DMA burst into smaller units, or, (2) initiate N refreshes before and N refreshes after the burst, where N is the potential number of refreshes missed.

Reduced Frequency Mode Application

When using DRAMs in the reduced frequency mode, the primary objective is assumed to be to save power. To obtain this, suggestion 1 is to use self-refreshing DRAMs, and suggestion 2 is to use low power extended refresh DRAMs.

Since the indirect objective is to minimize the $\overline{\text{RAS}}$ low time, a $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ Refresh should complete as soon as possible. Therefore, one way of accomplishing this is to reprogram the Refresh count/compare registers to suitable values so that once every refresh period (64ms) the CPU is internally interrupted. The interrupt will exit the halt and RF modes. A short interrupt handler loop can then strobe through all 512 row addresses and return the CPU into halt and RF modes.

Notes

DRAM Refresh Compare Register

('DramRefreshCompReg')

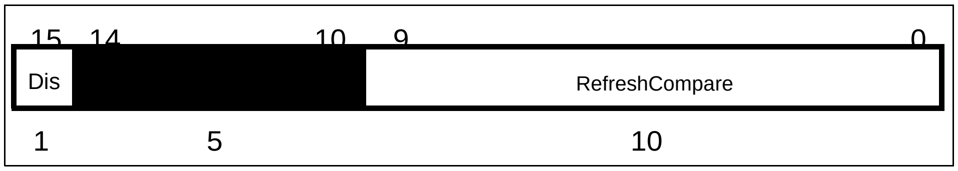


Figure 10.3 DRAM Refresh Compare Register.

This register forms a 10-bit Compare Register, shown in Figure 10.3. Bit 15 is a Disable Field, and bits 14:10 are reserved and should be written to with the same value as that of bit 15.

Note: Bit 15 disables the DRAM Refresh Function from occurring as specified; however, it does not disable the actual count register. As such, in the case of software diagnostics, the user should expect the refresh count register to continue incrementing. The following are two intended applications:

- *To not have external RAS/CAS toggle for self-refreshing DRAMs.*
- *If the DRAM Controller is not needed, then the refresh counter can be used as a general purpose counter.*

When Compare equals Count, the DRAM Controller will reset the count back to 0. If the Refresh Disable Field is set to 'enable', then a refresh sequence is initiated. The default value of the DRAM Refresh Count Register is 0x0000 at reset. As an example: for 25 MHz CPU with 8ms/512 refresh period, Compare should be programmed to Floor(8m/(512+1) / (1/25M))-1 = 0x0185. The register is both readable and writable. Table 10.3 lists the bits assignments for the DRAM Refresh Compare register.

Note: Technical worst case accounts for maximum burst length, where it is sufficient to add 1 to the DRAM page size.

The Refresh Compare Register is provided in a binary count manner (as opposed to a frequency select manner) to allow easier access for diagnostic and test purposes. The default value at reset is 0xFFFF (refer to Table 10.4).Common refresh settings are given in Table 10.5.

Bit	Function
15	Refresh Disable
14:10	Reserved (write the same value as bit 15)
9:0	Refresh Count

Table 10.3 DRAM Refresh Compare Register ('DramRefreshCompareReg') Bit Assignments.

Value	Action
'1'	Disable Refresh Counter (0xFFFF default)
'0'	Enable Refresh Counter

Table 10.4 Refresh Disable ('RefreshDis') Field Encodings.

Notes

Refresh Compare	CPU Frequency
0x00F8	16
0x0136	20
0x0184	25
0x0201	33
0xFFFF	default: disabled

Table 10.5 Common Refresh Settings for 8ms/512 or 16ms/1024 DRAMs.

DRAM $\overline{\text{RAS}}$ Multiplexer Select Register for Pair(1:0, 3:2) ('DramRasMuxSelReg'1_0, 3_2)

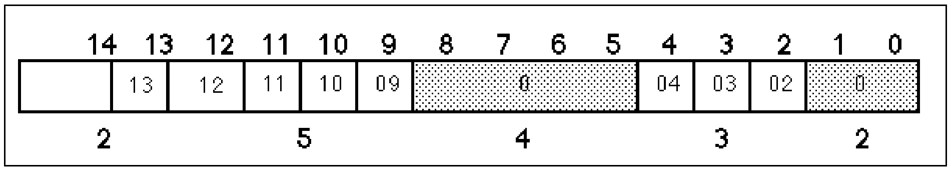


Figure 10.4 DRAM $\overline{\text{RAS}}$ Mux Select Register ('DramRasMuxSelReg').

The DRAM $\overline{\text{RAS}}$ Address Multiplexer Select Register, shown in Figure 10.4, programs which address bits go out to a DRAM Pair system during the row address period. The different selections allow software to upgrade the size of the DRAM chips and the memory port width without the use of external hardware jumpers. The register is both readable and writable with no default value at reset. This register must be programmed before the DRAM Controller is first used.

Bit	Function	High	Low
13	SysAddr $\overline{\text{RAS}}$ 13	A26	A24
12	SysAddr $\overline{\text{RAS}}$ 12	A23	A10
11	SysAddr $\overline{\text{RAS}}$ 11	A22	A20
10	SysAddr $\overline{\text{RAS}}$ 10	A19	A10
09	SysAddr $\overline{\text{RAS}}$ 09	A18	A9
04	SysAddr $\overline{\text{RAS}}$ 04	A25	A13
03	SysAddr $\overline{\text{RAS}}$ 03	A22	A12
02	SysAddr $\overline{\text{RAS}}$ 02	A21	A11

Table 10.6 DRAM $\overline{\text{RAS}}$ Mux Select Register Bit Assignments.

Note: Bits 15, 14, 8:5, 1, and 0 are reserved for future use.

DRAM $\overline{\text{CAS}}$ Multiplexer Select Register for Pair (1:0, 3:2) ('DramCasMuxSelReg'1_0, 3_2)

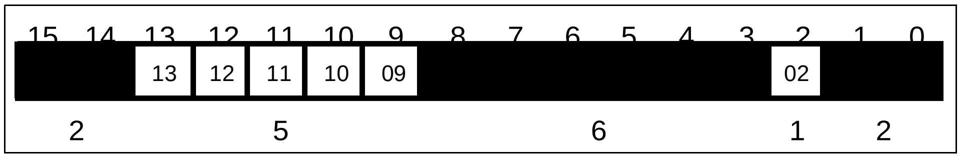


Figure 10.5 DRAM $\overline{\text{CAS}}$ Mux Select Register

Notes

The DRAM Pair $\overline{\text{CAS}}$ Multiplexer Select register, shown in Figure 10.5, programs which address bits go out to the DRAM system during the column address period. The different selections allow software to upgrade the size of the DRAM chips and the memory port width without the use of external hardware jumpers.

The register is both readable and writable with no default value at reset. This register must be programmed before the DRAM Controller is first used. The DRAM $\overline{\text{CAS}}$ Mux Select Register Bit Assignments are listed in Table 10.7. Refer to Table 10.8 for an example of DRAM $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ Mux select register settings.

Note that for debugging purposes, it is helpful to program the unused most significant CAS select bits to ‘1’. For example, in a 256KB DRAM system, programming these bits to ‘1’ will prevent CAS 13 from aliasing each time A1 flips on.

Bit	Function	High	Low
13	SysAddr $\overline{\text{CAS}}$ 13	A13	A1
12	SysAddr $\overline{\text{CAS}}$ 12	A24	A12
11	SysAddr $\overline{\text{CAS}}$ 11	A11	A1
10	SysAddr $\overline{\text{CAS}}$ 10	A10	A1
09	SysAddr $\overline{\text{CAS}}$ 09	A9	A0
02	SysAddr $\overline{\text{CAS}}$ 02	A20	A2

Table 10.7 DRAM $\overline{\text{CAS}}$ Mux Select Register (‘DramCasMuxSelReg’) Bit Assignments.

Note: Bits 15, 14, 8:3, 1, and 0 are reserved for future use.

Notes

SysAddr	Row 32-bit	Col 256K	Row 32-bit	Col 1M	Row 32-bit	Col 4M	Row 16-bit	Col 256K	Row 16-bit	Col 1M	Row 16-bit	Col 4M
SysAddr(13)												
SysAddr(12)					A23	A12					A10	A12
SysAddr(11)			A20	A11	A20	A11			A20	A1	A20	A11
SysAddr(10)	A19	A10	A19	A10	A19	A10	A10	A1	A19	A10	A19	A1
SysAddr(9)	A18	A9	A18	A9	A18	A9	A18	A9	A18	A9	A18	A9
SysAddr(8)	A17	A8	A17	A8	A17	A8	A17	A8	A17	A8	A17	A8
SysAddr(7)	A16	A7	A16	A7	A16	A7	A16	A7	A16	A7	A16	A7
SysAddr(6)	A15	A6	A15	A6	A15	A6	A15	A6	A15	A6	A15	A6
SysAddr(5)	A14	A5	A14	A5	A14	A5	A14	A5	A14	A5	A14	A5
SysAddr(4)	A13	A4	A13	A4	A13	A4	A13	A4	A13	A4	A13	A4
SysAddr(3)	A12	A3	A12	A3	A22	A3	A12	A3	A12	A3	A22	A3
SysAddr(2)	A11	A2	A21	A2	A21	A2	A11	A2	A11	A2	A21	A2
SysAddr(1)												
SysAddr(0)												
Total	3e00	3e00	3604	3e00	3600	2600	3a00	3a00	3600	3600	2600	2a00

SysAddr	Row 64-bit	Col 256K	Row 64-bit	Col 1M	Row 64-bit	Col 4M
SysAddr(13)						
SysAddr(12)					A23	A24
SysAddr(11)			A22	A11	A22	A11
SysAddr(10)	A19	A10	A19	A10	A19	A10
SysAddr(9)	A18	A9	A18	A9	A18	A9
SysAddr(8)	A17	A8	A17	A8	A17	A8
SysAddr(7)	A16	A7	A16	A7	A16	A7
SysAddr(6)	A15	A6	A15	A6	A15	A6
SysAddr(5)	A14	A5	A14	A5	A14	A5
SysAddr(4)	A13	A4	A13	A4	A13	A4
SysAddr(3)	A12	A3	A12	A3	A12	A3
SysAddr(2)	A11	A20	A21	A20	A21	A20
SysAddr(1)						
SysAddr(0)						
Total	3e00	3e04	3e04	3e04	3e04	3e04

Notes

SysAddr	Row 16-bit	Col 16M	Row 32-bit	Col 16M	Row 64-bit	Col 16M
SysAddr(13)	A24	A1	A24	A13	A26	A13
SysAddr(12)	A23	A12	A23	A12	A23	A24
SysAddr(11)	A20	A11	A20	A11	A22	A11
SysAddr(10)	A19	A10	A19	A10	A19	A10
SysAddr(9)	A18	A9	A18	A9	A18	A9
SysAddr(8)	A17	A8	A17	A8	A17	A8
SysAddr(7)	A16	A7	A16	A7	A16	A7
SysAddr(6)	A15	A6	A15	A6	A15	A6
SysAddr(5)	A14	A5	A14	A5	A14	A5
SysAddr(4)	A13	A4	A25	A4	A25	A4
SysAddr(3)	A22	A3	A22	A3	A12	A3
SysAddr(2)	A21	A2	A21	A2	A21	A20
SysAddr(1)						
SysAddr(0)						
Total	160c	0e00	161c	2e00	3e14	3e04

Table 10.8 Example 'DramRasMuxSelReg' and 'DramCasMuxSelReg' Settings.

DRAM MSB Base Address Register for Bank 0..3
('DramMSBBaseAddrReg(0..3)')

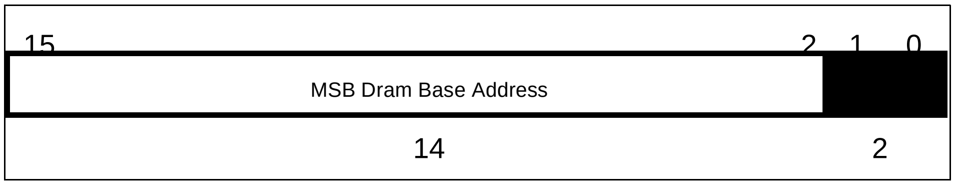


Figure 10.6 DRAM MSB Base Address Register ("DramMSBBaseAddrReg')

This field contains Bits 31-18 of the starting base physical address of the DRAM Bank. The programmer must write the same value for bits 31-28 to all DRAM MSB Base Address Registers. In addition, the programmer must write "0" for bits 17-16 of all DRAM MSB Base Address Registers, which restricts the smallest multiple DRAM bank size to 256K. The default value on reset is 0x77ff. Thus, the upper 3 DRAM Bank MSB Base Address registers must be programmed before any DRAM accesses can be initiated. The register is both readable and writable. Figure 10.6 illustrates the DRAM MSB Base Address Register.

Internally to the RC36100, bits 31-28 of Bank 0 MSB Base Address are used for the starting address of all banks. Also internally, bits 17-16 are reserved and hardwired to 0.

An example for 4 banks (2 pairs) of 1MByte interleaved DRAM starting at physical address 0 is in Table 10.9.

Notes

Bank	Phys. Address
Bank0	0x0000_0000
Bank1	0x0000_0000
Bank2	0x0020_0000
Bank3	0x0020_0000

Table 10.9 Example Bank Base Address Register (‘DramMSBBaseAddrReg’) Assignment.

An example for 1M DRAM + 2 banks of 4MByte interleaved DRAM at physical address 0 is in Table 10.10. Note that Bank1 must be assigned to an unused memory space.

Bank	Phys. Address
Bank0	0x0000_0000
Bank1	0x0F00_0000
Bank2	0x0010_0000
Bank3	0x0010_0000

Table 10.10 Example Bank Base Address Register (‘DramMSBBaseAddrReg’) Assignment.

DRAM MSB Bank Mask Register for Bank 0..3 (‘DramMSBBankMaskReg(0..3)’)

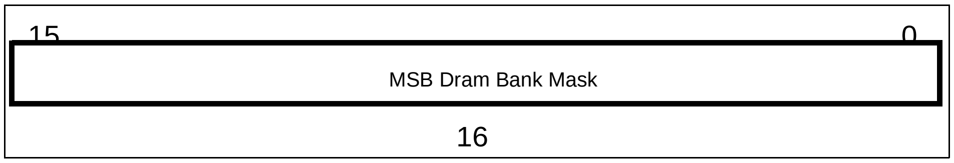


Figure 10.7 DRAM MSB Bank Mask (‘DramMSBBankMask(3:0)’) Registers.

There are 4 bank mask registers, one for each DRAM bank. The bank mask address register, shown in Figure 10.7, represents the most significant 16 address bits (bits 31:16). Bit settings for this register are listed in Table 10.11.

The bank mask registers are used to decide which address bits in the base address are to be used for comparing whether a DRAM bank select is to be activated. This DRAM Bank Mask is independent of the DRAM $\overline{\text{RAS}}$ Page Size Mask.

Internally, bits 31-16 must be programmed to the desired bank mask. This corresponds to separate address spaces for each chip select of 64K to 256M. Internally, bits 15:0 are ignored for bank mask comparisons.

To summarize, Bits 31:16 of each DRAM bank page mask are used to distinguish the size of each memory space. The format of the DramMSBBankMask is displayed in the above figure. The register is both readable and writable and is set to 0xFFFF by default on reset. This register should be programmed before the DRAM Controller is first used.

Value	Action
'1'	Bit is used in comparison
'0'	Bit is masked out of comparison

Table 10.11 DRAM MSB Bank Mask Bit Settings.

Notes

DRAM LSB Control Register for Bank 0..3 (‘DramLSBControlReg(0..3)’)

The DRAM LSB Control Register, shown in Figure 10.8, is used to control various DRAM controller options. This register is both readable and writable. The default value at reset is 0xFC03.

15 14	13 9	8	7 5	4 3	2 0
1	LSB RAS Page Mask	0	DRAM Type	Size	Reserved ‘011’
2	5	1	3	2	3

Figure 10.8 DRAM LSB Bank Control Register (‘DramLSBControlReg’)

This register should be programmed before the DRAM Controller is first used. Bit assignments for this register are listed in Table 10.12.

Note: While in the interleave mode, both banks must be programmed to have exactly the same value.

Bit	Description
15:14	Reserved to 1
13:9	RASPageMask
8	Reserved to 0
7:5	DramType
4:3	Size
2:0	Reserved to 011

Table 10.12 DRAM LSB Control Register (‘DramLSBControlReg’) Bit Assignments.

$\overline{\text{RAS}}$PageMask (‘RASPageMask’) Field (bits 15:8)

The $\overline{\text{RAS}}$ Page Compare Mask is used to determine how many of the upper physical address bits will be compared to determine if subsequent DRAM accesses are on the same $\overline{\text{RAS}}$ page of memory and thus do not need to initiate a $\overline{\text{RAS}}$ precharge.

Note that the $\overline{\text{RAS}}$ Bank Mask is independent and does not have the same function as the DRAM Bank Mask. To determine the bank mask, Physical Address Bits 31:16 are always compared, and Physical Address Bits 15:8 are compared if their corresponding $\overline{\text{RAS}}$ Bank Mask bit is clear. Page Mask bits are listed in Table 10.13 and DRAM LSB Page Mask bit settings are listed in Table 10.14.

Notes

DRAMType	PortSize	Interleaved(1)	PageMask to LSB(2)
16MxN	64-bit	yes	11 (0x 3c << 10)
16MxN	32-bit	non-	13(2) (0x 30 << 10)
16MxN	16-bit	non-	12 (0x 38 << 10)
4MxN	64-bit	yes	11 (0x 3c << 10)
4MxN	32-bit	non-	12 (0x 38 << 10)
4MxN	16-bit	non-	9 (0x 3f << 10)
1MxN	64-bit	yes	11 (0x 3c << 10)
1MxN	32-bit	non-	11 (0x 3c << 10)
1MxN	16-bit	non-	10 (0x 3e << 10)
256KxN	64-bit	yes	10 (0x 3e << 10)
256KxN	32-bit	non-	10 (0x 3e <<10)
256KxN	16-bit	non-	9 (0x 3f << 10)

NOTES:
1. Interleaved systems compare 1 less bit than theoretically possible, due to Column Address selection limitations.
2. Most configurations compare less bits than theoretically possible, as a trade-off for the jumper-less expansion.
3. 16Mx32 systems compare less bits than theoretically possible, due to Column Address selection limitations.
4. For very small memory systems, don't set any of the Page Type ('PType') control bits. In this case, the page comparator is ignored.

Table 10.13 PageMask ('PMask') Bits.

Value	Action
'1'	Bit is used in comparison
'0'	Bit is masked out of comparison

Table 10.14 DRAM LSB Page Mask Bit Settings.

DRAM Type ('DramType') Field (bits 7:5)

The DRAM Type selections are used to chose between different types of DRAM configurations. DRAM Type settings are listed in Table 10.15.

Value	Action
'7:3'	Reserved
'2'	FCT260
'1'	FCT245
'0'	FCT543 (default)

Table 10.15 DRAM Type ('DramType') Settings.

Notes

FCT543-Type (Latched Non-Multiplexer Type)

FCT543 Latched Mode assumes that the transceiver hardware between DRAM and the RC36100 consists of latched or registered transceivers such as the FCT543.

Setting this type with an interleaved port width causes the DRAM controller to use the interleave bus protocol. Interleaved systems use a different bus protocol which in essence accesses two banks at the same time, but only output enables 1 bank at a time, and thus can be burst read very quickly.

Non-interleaved systems use a bus protocol which in essence accesses the two banks in separate distinct address spaces. Latching words 1 and 3 on burst reads allows words 1 and 3 to be returned exactly 1 clock after words 0 and 2, respectively. This in turn allows the LS double word address to be bumped up (from 0x00 to 0x08) one clock earlier.

FCT245 Type (Non-latched Transceiver Type)

Non-Latched Mode assumes that the transceiver hardware between DRAM and the RC36100 consists of non-latched or non-registered transceivers such as the FCT245. In addition, in order to support the direction select and the single output enables of FCT245s, the even read output enable, DramRdOEnEven, logically OR's DramRdEnEven with DramWrEnEven. Likewise, the odd read output enable, DramRdOEnOdd, logically OR's DramRdEnOdd with DramWrEnOdd. DramWrEn-Even and DramWrEnOdd are unchanged as they are needed to indicate writes. If the FCT245 Type is used, all four banks must be of FCT245 Type.

FCT260-Type (Latched Multiplexer Type)

FCT260 Latched Multiplexer Type assumes that the transceiver hardware between DRAM and the RC36100 consists of latched or registered multiplexers such as the FCT260. In order to support the path select and single output enables of multiplexers, the even read output enable, DramRdOEnEven, logically OR's DramRdOEnEven with DramRdOEnOdd and stays low for most of the bus transaction instead of toggling just for even banks, so that it can be used for the FCT260 output enable. DramRdOEnOdd is unchanged, so that it can be used for a path select.

Setting this type with an interleaved port width causes the DRAM controller to use the interleave bus protocol. Interleaved systems use a different bus protocol which in essence accesses two banks at the same time, but only enables 1 bank at a time, and thus can be burst read very quickly. Non-interleaved systems use a bus protocol which in essence accesses the two banks in separate distinct address spaces. Latching words 1 and 3 on burst reads allows words 1 and 3 to be returned exactly 1 clock after words 0 and 2, respectively. This in turn allows the LS double word address to be bumped up (from 0x00 to 0x08) one clock earlier.

Note: At present, the FCT260 hardware approach represents one of the better price/performance ratios for interleaved systems, since only 3 chips instead of 4 chips are required.

Port Size ('Size') Field (bits 4:3)

The Port size of a bank determines its memory width. Table 10.16 lists the DRAM Port Width Encoding field. Note that in interleaved mode, both banks of the pair must have their MSB and LSB Bank Controller Registers programmed identically.

Value	Port Size
'10'	16-bit
'00'	32-bit (default)
'11'	64-bit (2x32-bit interleaved)
'01'	Reserved

Table 10.16 DRAM Port Width ('Size') Encoding Field.

Notes

DRAM MSB Control Register for Bank 0..3 (‘DramMSBControlReg'0..3)

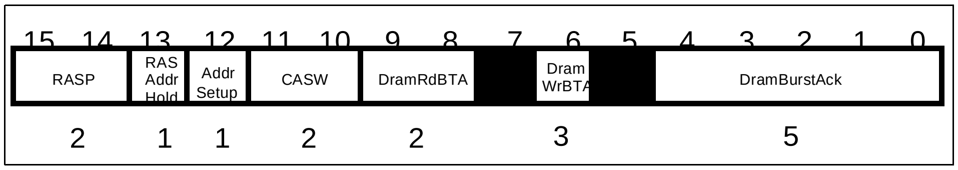


Figure 10.9 DRAM MSB Bank Control Register (‘DramMSBControlReg’).

The DRAM MSB Control Register, shown in Figure 10.9, is used to control various DRAM controller options. This register is both readable and writable and the default value is 0x051f. Bit assignments for this register are listed in Table 10.17.

Note: While in interleaved mode, both banks must be programmed to have exactly the same value.

Bit	Description
15:14	RASP
13	RASAddrHold
12	AddrSetup
11:10	CASW
9:8	DramRdBTA
7	‘0’
6	DramWrBTA
5	‘0’
4:0	DramBurstAck

Table 10.17 DRAM MSB Control Register Bit Assignments.

$\overline{\text{RAS}}$ Precharge Period (‘RASP’) Field (bit 15:14)

Before initiating a DRAM access to a new page, $\overline{\text{RAS}}$ must be held de-asserted in order to precharge the DRAM chip row decoders and sense amps. The RASP setting defines the length of this precharge period. The default value at reset is ‘0’ which encodes to 1 clock. RAS Precharge Field Encodings are listed in Table 10.18.

Value	Action
‘3’	reserved
‘2’	reserved
‘1’	2 clocks
‘0’	1 clock (default)

Table 10.18 $\overline{\text{RAS}}$ Precharge (‘RASP’) Field Encodings.

$\overline{\text{RAS}}$ Address Hold Time (‘RASAddrHold’) Field (bit 13)

DRAM Address Hold Time is required in the following three places:

- *after $\overline{\text{RAS}}$ asserts*
- *after $\overline{\text{CAS}}$ asserts*
- *after $\overline{\text{CAS}}$ re-asserts*

Notes

The RASAddrHold field handles the Address Hold Time from $\overline{\text{RAS}}$ asserting operation. Address Hold Time from $\overline{\text{CAS}}$ asserting and re-asserting on reads is handled by the CASW field. Address Hold Time from $\overline{\text{CAS}}$ asserting and re-asserting (on writes) is handled by the DRAMWrBTA field.

RASAddrHold (see Table 10.19 for field encodings) defines the length of the DRAM row address hold time. Normally, 0.5 clocks is enough hold time since most DRAMs require that the row address be held for about 10ns after $\overline{\text{RAS}}$ asserts. However, in very fast systems where the clock period is short, or in very noisy, heavily delayed systems, additional address hold time may be needed. Thus RASAddrHold can be extended from 0.5 to 1.5 clocks if necessary.

Value	Action
‘1’	1.5 clocks
‘0’	0.5 clocks (default)

Table 10.19 $\overline{\text{RAS}}$ Address Hold Time (‘RASAddrHold’) Field Encoding.

Address Setup Time to $\overline{\text{RAS}}$ and to $\overline{\text{CAS}}$ (‘AddrSetup’) Field (bit 12)

DRAM Address Setup Time is required in the following three places:

- *Row Address Setup Time to $\overline{\text{RAS}}$ asserting*
- *Column Address Setup Time to $\overline{\text{CAS}}$ asserting (also Early Write Signal Setup Time to $\overline{\text{CAS}}$ asserting)*
- *Column Address Setup Time to $\overline{\text{CAS}}$ re-asserting on mini-bursts or bursts (also Early Write Signal Setup Time to $\overline{\text{CAS}}$ re-asserting on mini-bursts or bursts)*

AddrSetup (field encodings are listed in Table 10.20) defines the length of the DRAM address set-up time. Because most DRAMs require that the row or column address be set-up 0ns before $\overline{\text{RAS}}$ or $\overline{\text{CAS}}$ asserts, a minimum of 0.5 clocks is, normally, sufficient setup time.

However, in very fast systems where the clock period is short, or in very noisy, heavily delayed systems, additional address (and also early write) set-up time may be needed. Therefore, AddrSetup can be extended from 0.5 to 2.5 clocks if necessary. Thus, for a new page-DRAM access, AddrSetup may add 1 extra address set-up clock cycle before $\overline{\text{RAS}}$ asserts, before $\overline{\text{CAS}}$ asserts, or before every $\overline{\text{CAS}}$ re-assertion.

Note: In the particular case of extra set-up time to RAS, choosing the longer set-up time causes ‘AddrSetup’ to be one cycle longer than the precharge time.

Value	Action
‘1’	2.5 clocks (This value reflects an address set-up time of 1.5 clocks plus the RAS precharge time of 1 clock)
‘0’	0.5 clocks (default)

Table 10.20 Address Setup Time to $\overline{\text{RAS}}$ or to $\overline{\text{CAS}}$ (‘AddrSetup’) Field Encoding.

$\overline{\text{CAS}}$ Active Pulse Width (‘CASW’) Field (bit 11:10)

The 2-bit encoding lengthens the $\overline{\text{CAS}}$ active pulse width from a minimum of 1.5 clocks to a minimum of 2.5 clocks. The default value is ‘1’ at reset which decodes to 1.5 clocks (see Table 10.21 for field encodings). This field can also be thought of as the CASAddrHold field on read accesses; however, on DRAM writes, DRAMWrBTA must be used to extend the address hold and early write signal hold time.

Notes

Value	Action
‘3’	reserved
‘2’	2.5 clocks
‘1’	1.5 clocks (default)
‘0’	reserved

Table 10.21 $\overline{\text{CAS}}$ Width (‘CASW’) Field Encoding.

DRAM Read Cycle Bus Turn-Around (‘DramRdBTA’) Field (bit 9:8)

The Bus Turn-Around field determines the minimum number of clocks between the end of a read and the beginning of the next non-DRAM bus transaction. Sometimes a slow interface is needed because of the amount of time it takes the DRAM or its transceivers to tri-state off of their respective busses. Thus, a slow bus turnaround option is incorporated into the DRAM Controller. A two-bit value stored in a control register will stall the bus interface unit between the end of a read cycle and from starting the subsequent transfer by up to two system clock cycles. The default at reset is the value ‘1’, which decodes to 1 clock of BTA. Field encodings are listed in Table 10.22.

Value	Action
‘3’	3 clocks
‘2’	2 clocks
‘1’	1 clock (default)
‘0’	0 clocks

Table 10.22 DRAM Read Cycle Bus Turn-Around (‘DramRdBTA’) Field Encoding.

DRAM Write Cycle Bus Turn Around (‘DramWrBTA’) Field (bit 6)

The DRAM Write Cycle Bus Turn Around Field determines whether other transactions can begin either one clock cycle before the final $\overline{\text{CAS}}$ de-assertion or after the final $\overline{\text{CAS}}$ de-assertion.

DramWrBTA defines the minimum number of clocks between the end of a write and the beginning of the next non-DRAM bus transaction. The DRAM Controller uses the early write protocol and thus can typically give up the data and address buses one clock before the end of the write (1 clock before $\overline{\text{CAS}}$ de-asserts for the last time).

However, this leaves the column address and the early write signal 0.5 clocks of Hold Time (assuming CASW == 1). Thus very fast systems or very noisy systems may want to extend the Address Hold Time after $\overline{\text{CAS}}$ writes to 1.5 clocks. This can be done indirectly by changing the Dram-WrBTA field. (Address Hold Time on DRAM reads is always at least 1.5 clocks). The default at reset is the value ‘0’ which decodes to 0 clocks of Write Cycle BTA. Field encodings are listed in Table 10.23.

Value	Action
‘1’	1 clock
‘0’	0 clocks (default)

Table 10.23 DRAM Write Cycle Bus Turn-Around (‘DramWrBTA’) Field Encoding.

Notes

Burst Acknowledge Placement (‘DramBurstAck’) Field (bit 4:0)

On 4-word burst reads, the acknowledge back to the CPU core needs to be placed so that the CPU pipeline can restart optimally. Acknowledge should be placed 3 clock cycles before the last datum arrives. Wait states, via SysWait, delay the next de-assertion of CAS; however, BurstAck may have already been asserted. Thus, on burst memory cycles where SysWait may potentially be asserted, BurstAck must be programmed to ‘31’ (for field encodings, see Table 10.24) such that it asserts with the last Datum.

As a reference point, it is from the clock cycle that CAS first asserts and the DramBurstAck internal counter begins counting. The DRAM controller uses the page hit case with no extra CAS Precharge
(Addr setup = 0.5) as its minimum value and in the case of page misses and/or in the case of extra CAS Precharge (Addr setup = 1.5) settings, automatically delays the DramBurstAck internal counter.

Note: In the Debug mode of the RC36100, if the DebugFCMN pin is asserted, BAck is always automatically asserted with the last Datum.

Value	Action
‘31’	Acknowledge with last Datum (Default).
‘30’...‘0’	Acknowledge from 30 to 0 clocks (referenced to CAS first asserting).

Table 10.24 DRAM Burst Read Acknowledge (‘DramBurstAck’) Encoding.

Option	Configuration				
	16-Bit	32-Bit	64-Bit 543	64-Bit 260	64-Bit 245
Default	11	3	0	0	1
Addrhold = 0.5 or 1.5	11	3	0	0	1
Addr setup = 1.5 CASW = 1.5	18	6	1	1	2
Addrsetup = 0.5 CASW = 2.5	19	7	2	2	3
Addrsetup = 1.5 CASW = 2.5	26	10	3	3	4

Table 10.25 Typical DRAM Burst Read Acknowledge Settings

Timing Diagrams

The timing diagrams for the RC36100 DRAM Controller are divided into the following five sections:

- *basic reads*
- *basic writes*
- *interleaved reads*
- *interleaved writes*
- *refreshes*

In the Basic Reads and Basic Writes sections, ordinary 8/16/32-bit DRAM accesses are discussed. Concepts including single versus multiple datum accesses, and many of the option fields, including RASP, RASAddrHold, AddrSetup, and CASW are shown.

The Interleaved Reads and Interleaved Writes sections discuss the timing for connecting two banks of 32-bit DRAM such that for burst read accesses, both even and odd words are accessed simultaneously and thus improve the performance of the DRAM system.

Notes

In the Refresh timing section, the RC36100 is shown to implement a staggered refresh cycle using the CAS-before-RAS protocol of standard DRAMs.

Standard DRAM Chip Summary

The RC36100 DRAM Controller uses standard page mode DRAM chips. These DRAM chips multiplex their address pins, support page-mode, and use CAS-before-RAS refresh. Thus on the first part of a DRAM access, upper half of the address is strobed in with a Row Address Strobe (RAS) and on the last part of the access, the lower half of the address is strobed in with a Column Address Strobe (CAS).

Because page mode DRAMs have an internal array such that an entire row (page) of memory data cells are selected, once that row (page) is turned on, subsequent accesses using CAS can be done much quicker than the first access of any given row.

Thus on burst accesses, the RC36100 DRAM Controller keeps RAS asserted and toggles CAS to get multiple datum. However, if a new row (page) is accessed, then the DRAM row array must be re-precharged, typically for two clock cycles. Similarly, CAS is toggled in order to precharge the CAS array before accessing a new memory data location.

On writes, DRAM chips have two modes: early writes and regular writes. Because of the timing advantage of early writes, the RC36100 DRAM Controller uses early writes where data is strobed into the DRAM chip with the assertion of CAS instead of with the assertion of the write strobe.

Finally, DRAM devices require that their contents be periodically refreshed. One method is simply to make sure each DRAM row is accessed periodically; however, DRAM chips also have a special CAS-before-RAS refresh protocol: if CAS is asserted before RAS, the chip internally executes a refresh access and bumps up an internal row address counter. The RC36100 DRAM Controller uses the CAS-before-RAS refresh protocol.

Basic New Page DRAM Read

In Figure 10.10 on page 23, a basic new page DRAM read transaction is shown. The transaction is initiated like other transactions with the assertion of SysALE and SysBurstFrame. Along with the assertion of SysALE, the SysAddr() bus drives the row address (the upper half of the addresses that the DRAM chips are expecting).

Unlike the Memory Controller, the DRAM Controller has many signals that assert and/or de-assert using the falling edge of SysClk in order to fully optimize the timing for DRAM systems. Thus, 1/2 clock cycle after SysALE asserts, one of the four DramRAS(3:0) strobes will assert depending on which of the four banks is selected. This gives the DRAM chips minimal address setup time to the RAS strobe.

One-half clock cycle after DramRAS asserts, the SysAddr() bus switches, giving 1/2 clock of address hold time, and begins driving the column address (the lower half of the addresses that the DRAM chips are expecting). One-half cycle after SysAddr() changes to the column address, from one to four of the DramCAS(3:0) strobes will assert depending on if a particular byte is required on the read.

Note that the RC36100 may assert all four CAS lines even though it only requires some of the bytes (in such a case, the unneeded bytes are ignored by the RC36100 internally). The default CAS assertion gives the column address setup time to the CAS strobe. In a typical read, DramCAS() remains active for 1.5 clocks. On the final clock of the assertion of DramCAS(), SysDataRdy is asserted and the data from the DRAM is latched into the CPU on the final SysClk rising edge.

During the time that DramRAS() is active, one of the read enable strobes will also be asserted. These read enable strobes, either DramRdEnEven (see Figure 10.10) or DramRdEnOdd (asserted analogously to the ‘Even' signal) can be used to select even (DramRAS(2) or DramRAS(0)) or odd (DramRAS(3) or DramRAS(1)) memory banks, respectively when multiple banks or transceivers are used. The use of DramRdEn(Even/Odd) varies slightly depending on the type of transceivers and interleaving factor and will be explained in later sections of this chapter. This includes "Interleaved Reads," “Interleaved Writes,” and “System Examples.”

Notes

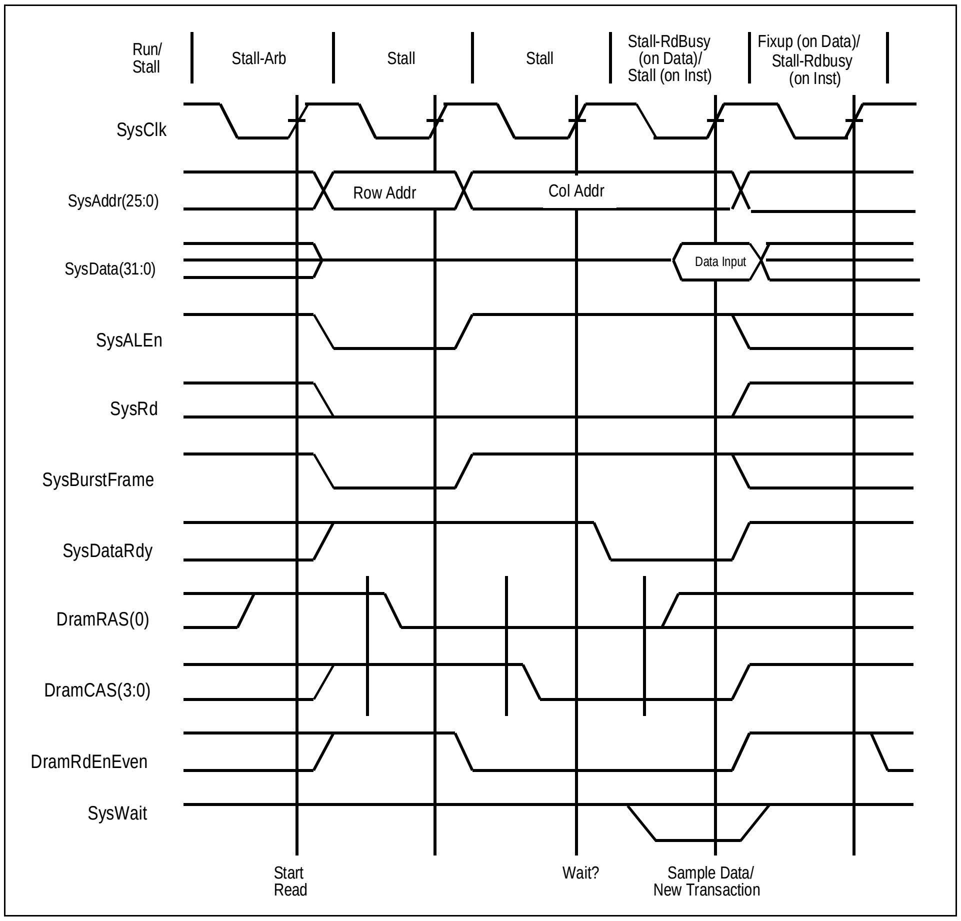


Figure 10.10 Basic DRAM Read

Note: DRAM styles FCT245 and FCT260 have slightly different $\overline{\text{DRAMRdEnEven}}$ behavior than the case shown in Figure 1.10.

$\overline{\text{RAS}}$ Asserted at End of Transfer

The RC36100 DRAM Controller leaves the $\overline{\text{DRAMRAS()}}$ asserted after single reads. Leaving $\overline{\text{RAS}}$ asserted allows a subsequent DRAM transaction to go directly into the $\overline{\text{CAS}}$ stage if the next transaction is to the same row (page) as the previous transaction. Thus by using the Control LSB Register Page Type ('PType') field, the DRAM Controller can keep $\overline{\text{RAS}}$ asserted after burst reads, single word reads, and/or writes. The DRAM Controller accomplishes this by using its internal Page Comparator as described in the Page Comparator Algorithm section earlier in this chapter. Figure 1.11 illustrates this operation.

Notes

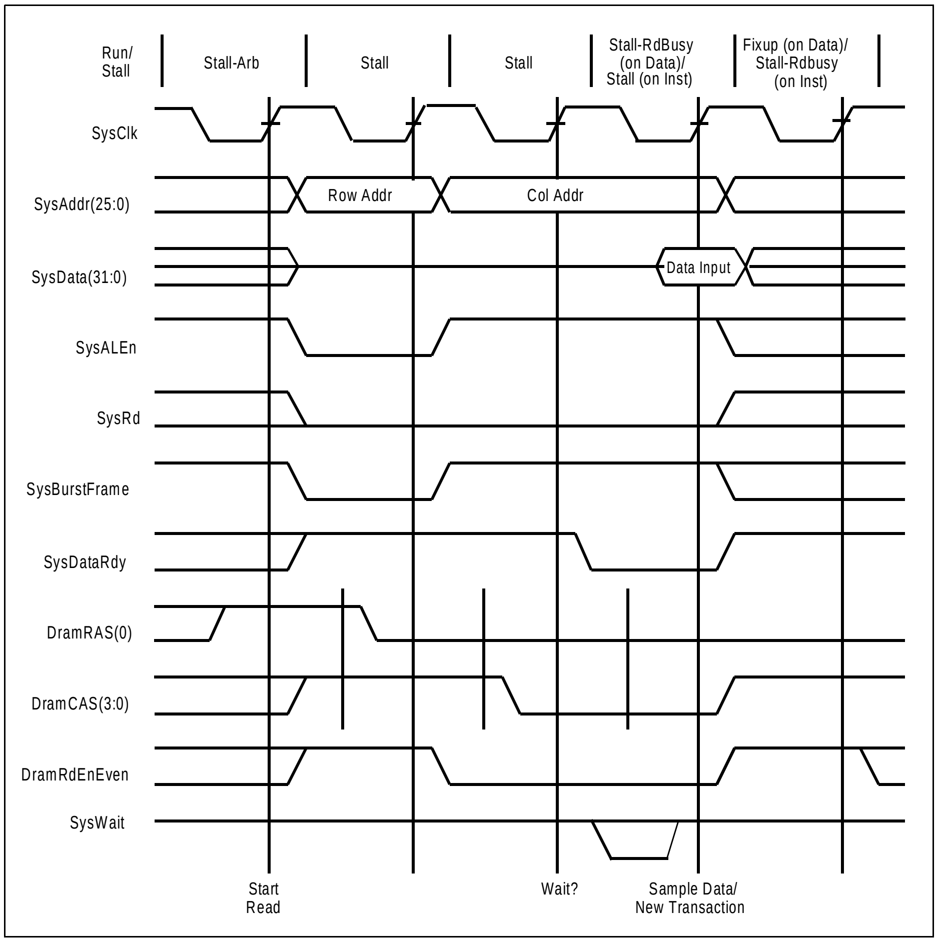


Figure 10.11 $\overline{\text{RAS}}$ asserted at End of Transfer

$\overline{\text{RAS}}$ Asserted at Start of Transfer

If the RC36100 DRAM Controller leaves DramRAS() asserted at the end of a previous DRAM transaction and the current DRAM transaction is on the same row (page), then DramRAS() does not go through a precharge/address strobe stage and is skipped. In this case as shown in Figure 10.12, the CAS address strobe and data access stages happen immediately at the start of the transaction. Note that intervening non-DRAM accesses do not affect the page comparator.

$\overline{\text{RAS}}$ Asserted Throughout Transfer

If the RC36100 DRAM Controller leaves DramRAS() asserted at the end of a previous transaction and the current transaction is on the same row (page), then DramRAS() does not go through a precharge/address strobe stage and is skipped. In this case, as shown in Figure 1.12, the CAS address strobe and data access stages happen immediately at the start of the transaction.

Notes

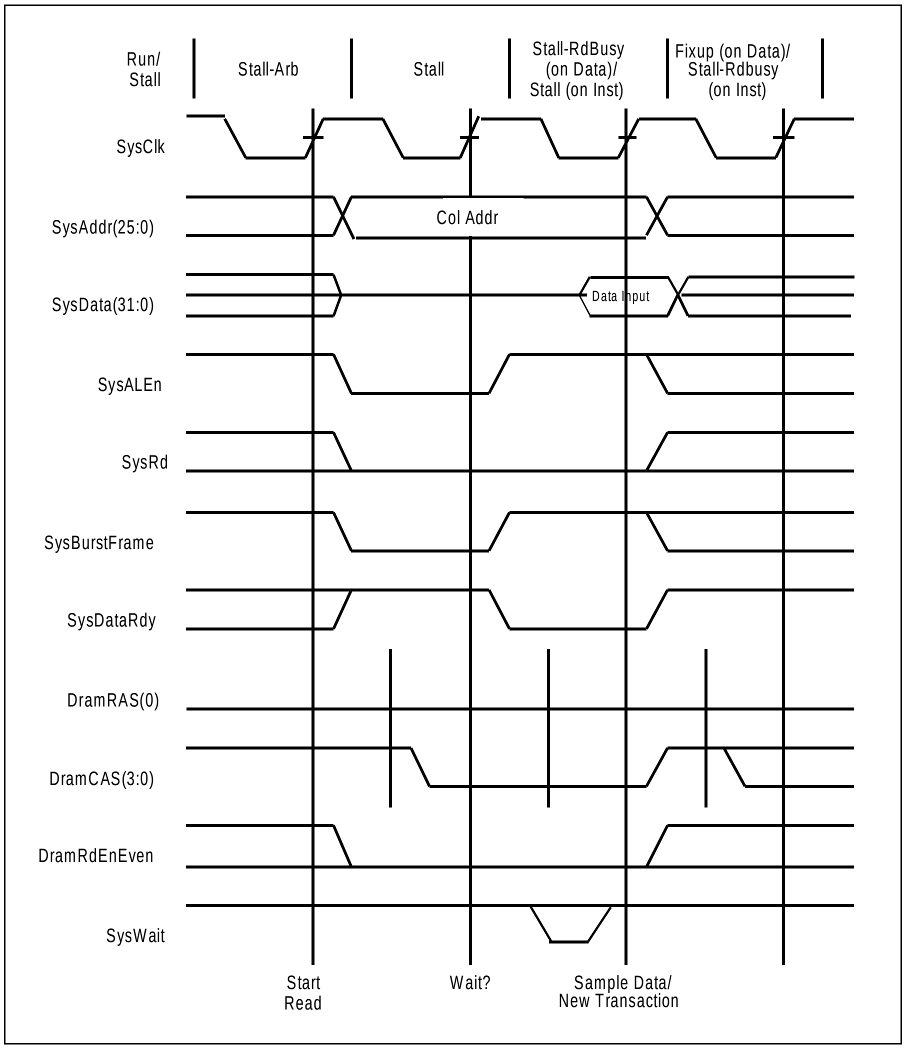


Figure 10.12 $\overline{\text{RAS}}$ asserted at Start of Transfer

$\overline{\text{RAS}}$ Precharge Field

The RC36100 DRAM Controller generates an access that either has $\overline{\text{RAS}}$ asserted on a row (page) or has $\overline{\text{RAS}}$ de-asserted. On a subsequent access to a different row (page), the DRAM Controller then verifies that $\overline{\text{RAS}}$ either is de-asserted or has been de-asserted for at least an amount of clocks equal to the Control Register 0 $\overline{\text{RAS}}$ Precharge ('RASP') Field.

Figure 1.13 shows a RASP of 2 clocks where $\overline{\text{RAS}}$ was left asserted on the previous DRAM transaction. To precharge the DRAM chips, $\overline{\text{DramRAS()}}$ must first de-assert for 2 clocks. Then DRAMRAS() asserts after the 2 clocks, and the transaction continues.

Notes

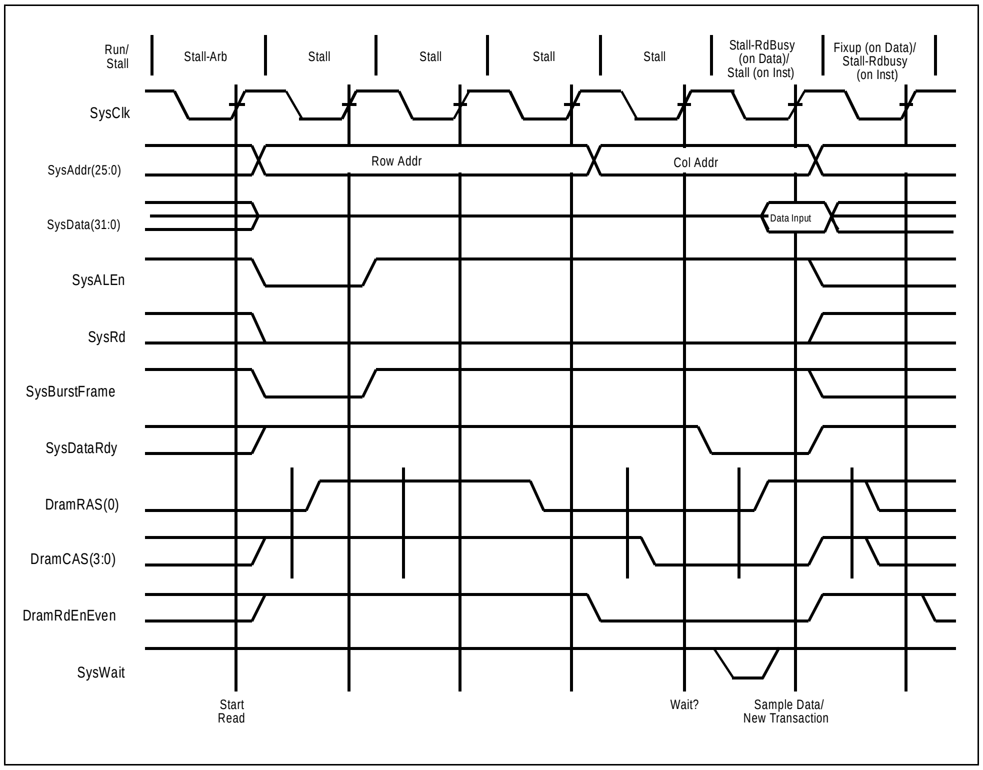


Figure 10.13 $\overline{\text{RAS}}$ Precharge at start of Transfer

$\overline{\text{RAS}}$ Address Hold Field

The RASAddrHold setting can provide extra row address hold time by extending the number of clocks that the DRAM Address Multiplexer delays before switching between row and column addresses. Figure 10.14 shows a DRAM read where RASAddrHold has been set for 1.5 clocks instead of the default of 0.5 clocks in fast or noisy systems, as shown in Figure 1.14.

Notes

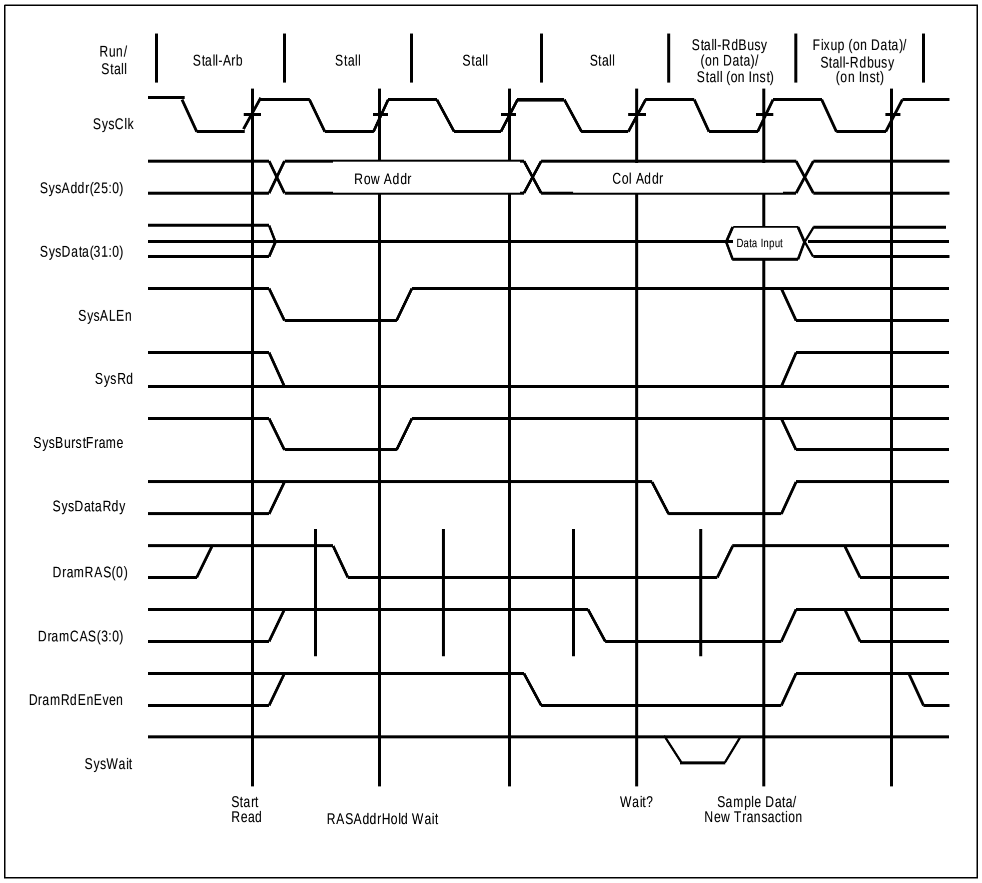


Figure 10.14 Extended Row Address Hold

Address Setup Field

Whenever the RC36100 DRAM Controller generates an access where $\overline{\text{RAS}}$ is de-asserted and a new $\overline{\text{RAS}}$ is generated (new page case) or a new $\overline{\text{CAS}}$ is generated, the DRAM Controller is responsible for making sure that $\overline{\text{RAS}}$ or $\overline{\text{CAS}}$ is de-asserted for at least the DRAM Control Register Addr Setup Field amount of clocks after the row or column address is valid.

Figure 10.15 shows a case with AddrSetup of 1.5 clocks between multiple datum on a read. Although this field primarily controls the address setup time of $\overline{\text{CAS}}$ relative to the column address being valid, this field also allows control over the precharge time before $\overline{\text{DramCAS()}}$ asserts. To match the column address setup time characteristic, the RC36100 DRAM Controller also applies this field to the $\overline{\text{DramRAS()}}$ signal relative to the row address on cases where $\overline{\text{DramRAS()}}$ was left de-asserted from a previous transaction.

Although not pictured, the AddrSetup field also applies to $\overline{\text{CAS}}$ in the case where $\overline{\text{RAS}}$ is left asserted and then a subsequent same page access occurs. The RASPrecharge field takes care of the case where $\overline{\text{RAS}}$ is asserted and then a subsequent different page access occurs.

Notes

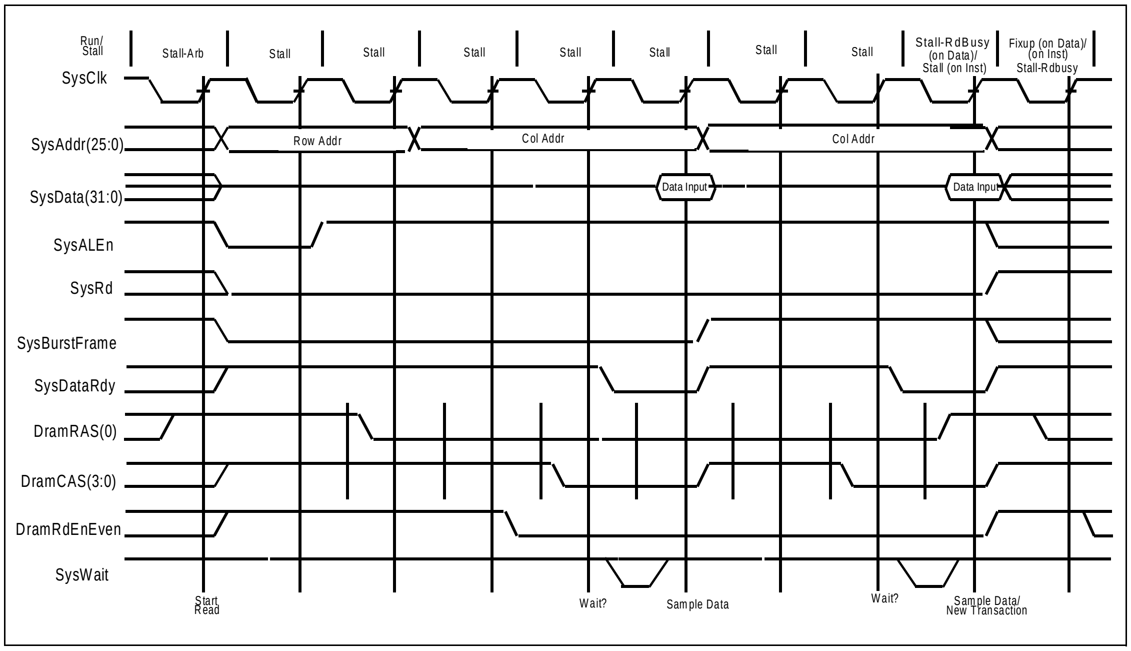


Figure 10.15 Extended Address Set-up

$\overline{\text{CAS}}$ Width Field

The RC36100 DRAM Controller can support slower DRAM speeds by increasing the $\overline{\text{CAS}}$ pulse width. This option is programmable using the DRAM Control Register $\overline{\text{CAS}}$ Width ('CASW') field. Figure 10.16 shows the case where CASW has been set to 2.5 clocks instead of the default 1.5 clocks.

Notes

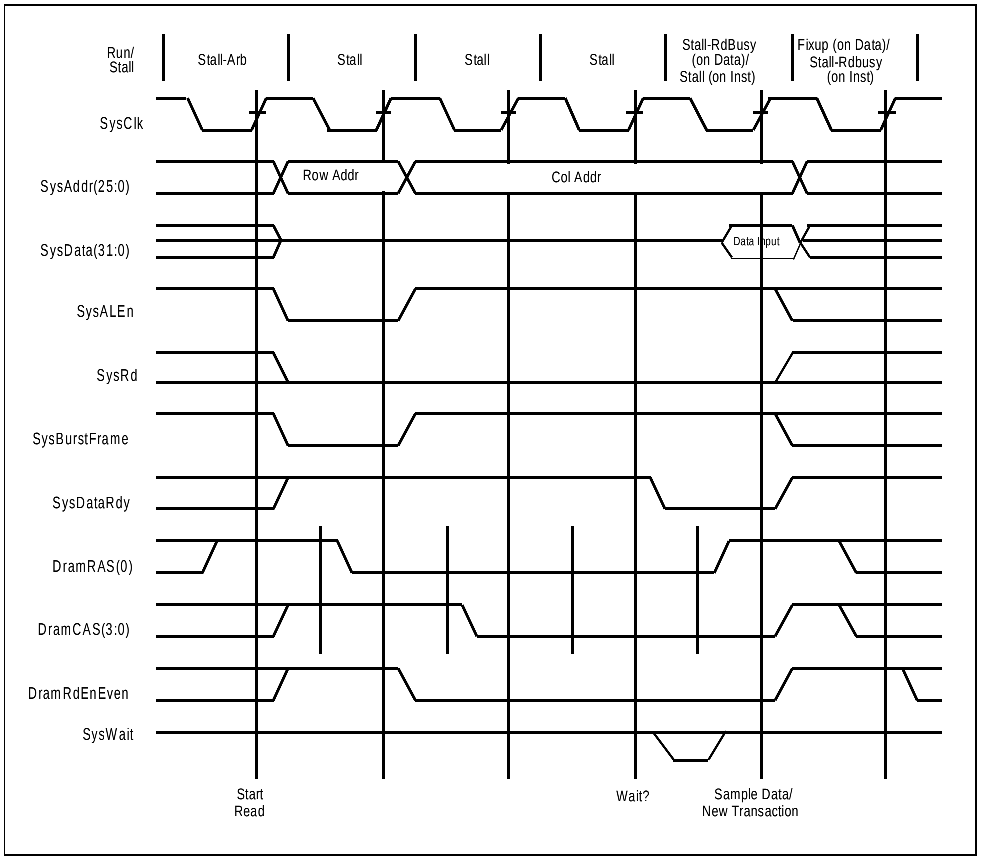


Figure 10.16 Extended $\overline{\text{CAS}}$ Width

Multiple Data Reads

The RC36100 DRAM Controller groups mini-bursts (word and tri-byte accesses on a 16-bit wide port) and non-interleaved bursts (4-word cache refill) the same way. Mini-bursts and burst reads require multiple data. As shown in Figure 10.17, second and subsequent data are first preceded with DramCAS() de-asserting for 1/2 clock (default) and the DRAM mux'ed SysAddr() counting up towards the next column address. (On 16-bit ports A1 is the LSB; on 32-bit ports A2 is the LSB). In each case, the final data is denoted by SysBurstFrame de-asserting. Although not shown, AddrSetup and CASW fields apply to each $\overline{\text{CAS}}$ Data.

Note: In the 16-bit mode, $\overline{\text{DRAMRdEnEven}}$ de-asserts between every word of a 4-word cached-burst refill for one clock cycle.

Notes

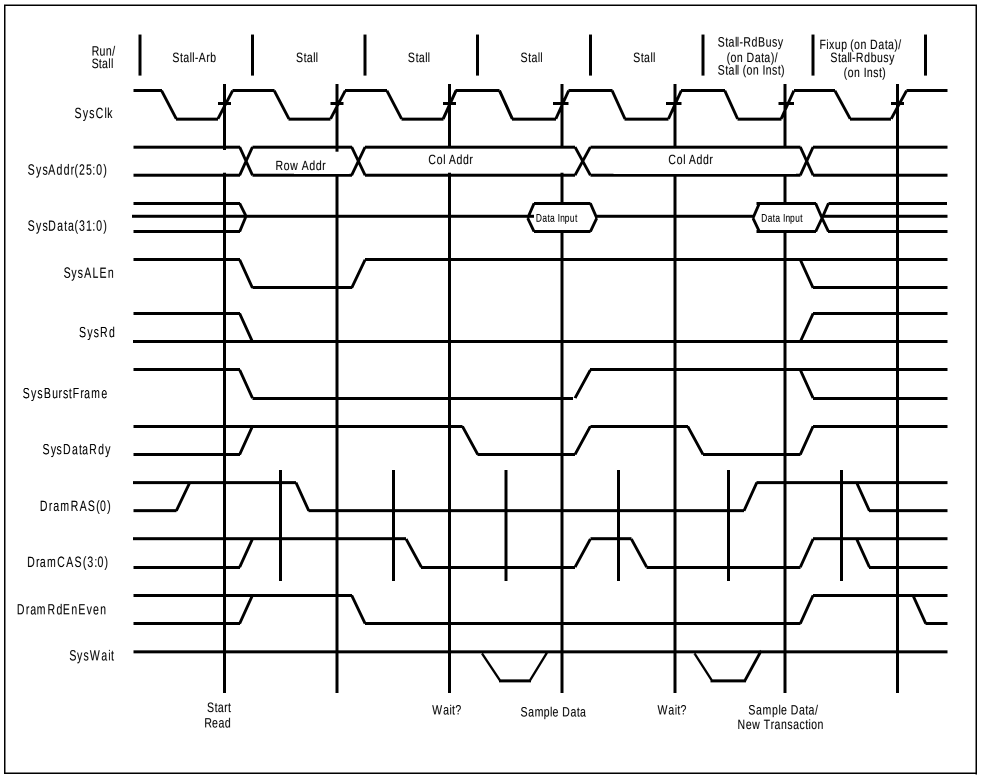


Figure 10.17 Multiple Data read

Basic DRAM Write

Figure 10.18 shows a basic DRAM write transaction on a precharged (e.g., after reset or a refresh) new page (row). The transaction is initiated like other transactions with the assertion of SysALE and SysBurstFrame. Along with the assertion of SysALE, the SysAddr() bus drives the row address (the upper half of the addresses that the DRAM chips are expecting).

Unlike the Memory Controller, the DRAM Controller has many signals that asserted and/or de-assert using the falling edge of SysClk in order to fully optimize the timing for DRAM systems. Thus 1/2 clock cycle after SysALE asserts, one of the four DramRAS(3:0) strobes will assert (depending on which one of the four banks is selected). This gives the DRAMs address setup time to the RAS strobe. 1/2 clock cycle after DramRAS asserts, the SysAddr() bus switches and begins driving the column address (the lower half of the addresses that the DRAM chips are expecting).

In addition, the SysData() bus begins driving the appropriate data. One-half cycle after SysAddr() changes, from one to four of the DramCAS(3:0) strobes will assert, depending on if a particular byte is required on the write. The default CAS assertion gives the column address setup time and data setup time to the CAS strobe. The DRAM Controller uses the early write mode of page mode DRAMs where the data is latched by the DRAM chips on the asserting edge of CAS instead of the de-asserting edge. Thus SysDataRdy also asserts a clock early, to indicate to external resources, such as a logic analyzer, that data is valid.

The early write mode allows address pipelining if another non-DRAM access is waiting to use the system bus. Because of the early write mode and address pipelining, the data for the write may disappear on the final clock of the write, if the Write Bus Turn Around is programmed to be '0', because another non-DRAM transaction may have already started. To prevent address pipelining on systems that require additional data hold time (either very high frequency systems or very noisy systems), the Write Bus Turn Around can be programmed to be '1'.

Notes

During the time that DramCAS() is active, one of the write enable strobes will also be asserted. These write enable strobes, either DramWrEnEven or DramWrEnOdd can be used to select even (DramRAS(2) or DramRAS(0)) or odd (DramRAS(3) or DramRAS(1)) memory banks, respectively when multiple banks or transceivers are used. The use of DramWrEn(Even:Odd) and DramWrEn(Even:Odd) varies slightly depending on the type of transceivers and interleaving factor and will be further explained in a later section of this chapter, "System Examples."

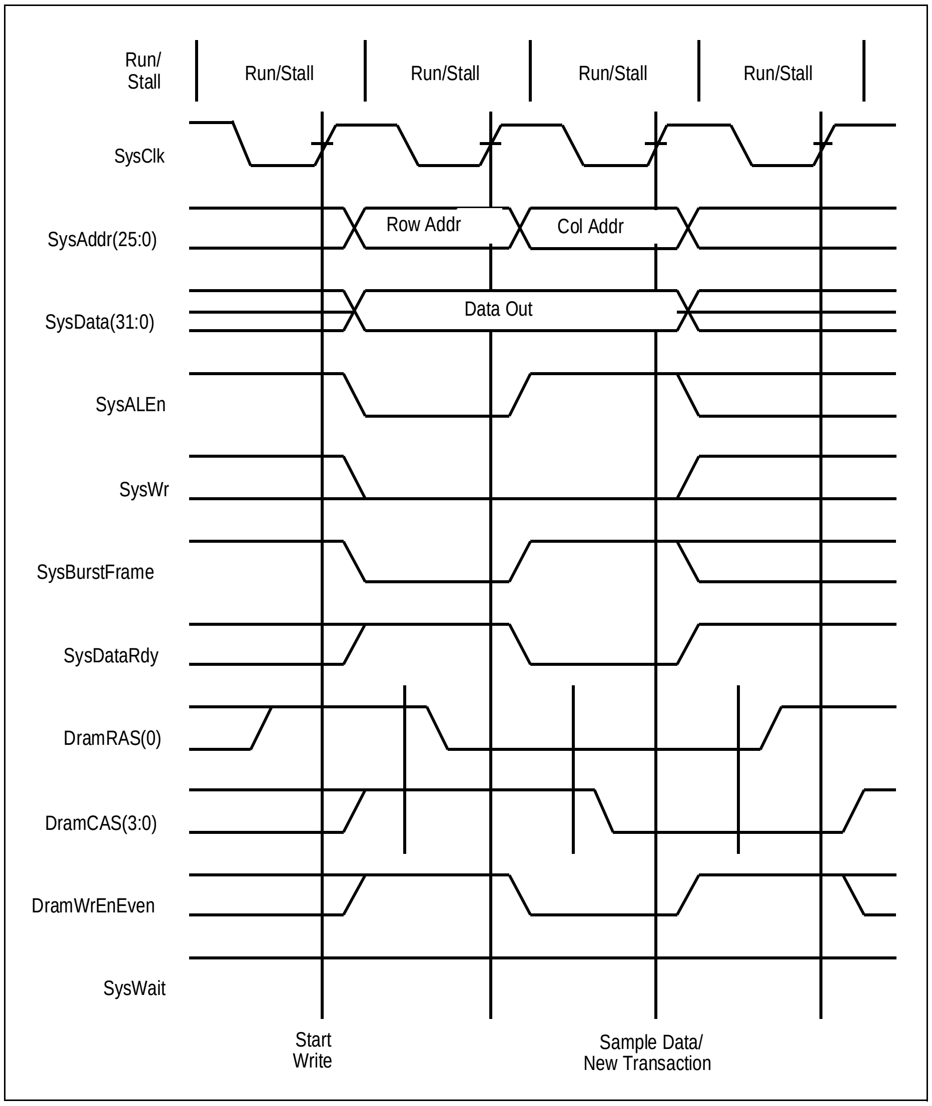


Figure 10.18 Basic DRAM Write

Note: DRAMRdEnEven or Odd (not shown) have slightly different behaviors depending on the DRAM style (FCT245, 260, or 543).

RAS Asserted at Start of Write

If the RC36100 DRAM Controller leaves DramRAS() asserted at the end of a previous transaction and the current transaction is on the same row (page), then DramRAS() does not go through a precharge/address strobe stage and is skipped. In this case, as shown in Figure 10.19, the CAS address strobe and data access stages happen immediately at the start of the transaction.

Notes

$\overline{\text{RAS}}$ Asserted at End of Write

The RC36100 DRAM Controller leaves the $\overline{\text{DRAMRAS()}}$ asserted after write transactions. Leaving $\overline{\text{RAS}}$ asserted allows a subsequent DRAM transaction to go directly into the $\overline{\text{CAS}}$ stage if the next transaction is to the same row (page) as the previous transaction. Thus by using the Control 0 MSB Register Page Type ('PType') field, the DRAM Controller can keep $\overline{\text{RAS}}$ asserted after burst reads, single word reads, and/or writes. The DRAM Controller accomplishes this by using its internal Page Comparator as described in the Page Comparator Algorithm section earlier in this chapter.

$\overline{\text{RAS}}$ Asserted Throughout Write

If the RC36100 DRAM Controller leaves $\overline{\text{DramRAS()}}$ asserted at the end of a previous transaction and the current transaction is on the same row (page), then $\overline{\text{DramRAS()}}$ does not go through a precharge/address strobe stage and is skipped. In this case, as shown in Figure 1.19, the $\overline{\text{CAS}}$ address strobe and data access stages happen immediately at the start of the transaction.

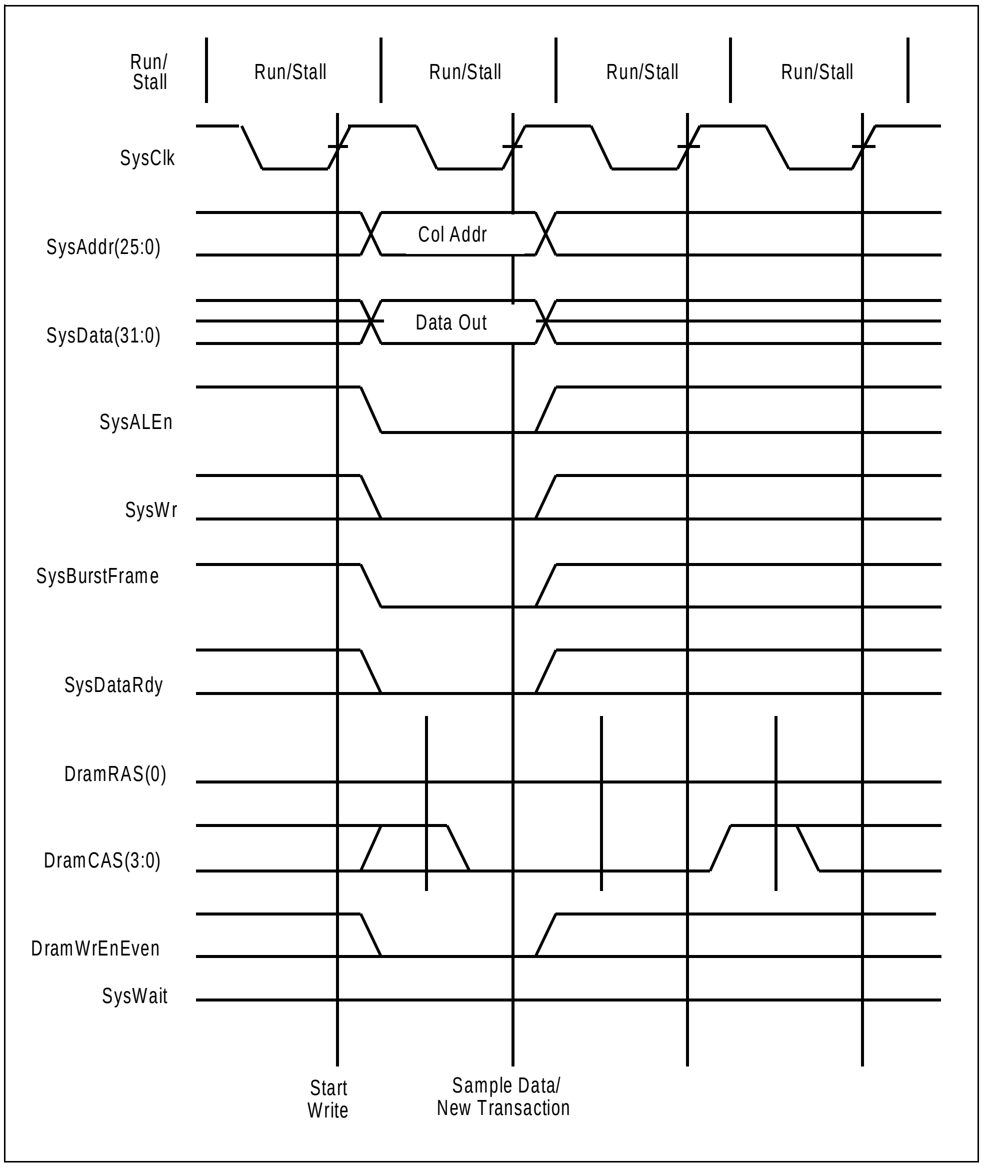


Figure 10.19 $\overline{\text{RAS}}$ Asserted Throughout DRAM Write

Other DRAM timing Controls

Most of the DRAM Control Fields work identically for reads and writes, this includes:

- *$\overline{\text{RAS}}$ Precharge Field*

Notes

- *$\overline{\text{RAS}}$ Address Hold Field*
- *Address Setup Field*
- *$\overline{\text{RAS}}$ Address Hold Field*
- *CASW Field*

For more details, see the DRAM Read Timing section.

Write Bus Turn-Around

Normally, a subsequent non-DRAM transaction can potentially begin 1 clock before the DRAM has actually completed the write. For example, this DRAM Write Pipelining can occur when a DRAM write is followed by an instruction read from PROM. In some cases where either the system clock frequency is very high or the column address is very noisy, the column address needs additional hold time. By using the Write Bus Turn-Around Field in a DRAM bank's MSB Control Register, the column address is held for an extra clock by delaying any non-DRAM transactions for 1 clock, as shown in Figure 10.20.

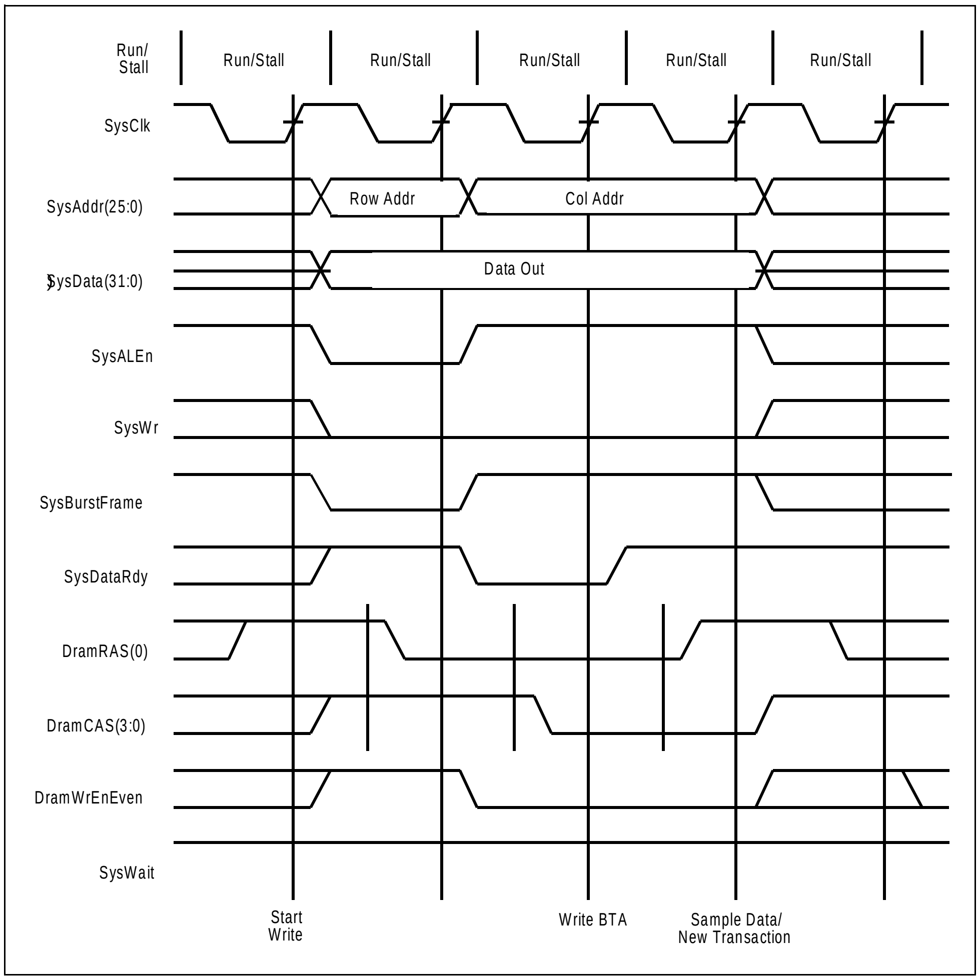


Figure 10.20 Write Bus Turn-around

Notes

Two Datum Write Transaction

In cases with a 16-bit bus port width that access more than a halfword (tri-byte, word, or DMA Burst Write) or in cases with a non-interleaved 32-bit bus port width that is a DMA Burst Write, the RC36100 DRAM Controller does a two datum or multi-datum write transaction, as shown in Figure 10.21. The second or subsequent data have finished using the DRAM page mode, such that new SysData() is put on the data bus and DramCAS() is re-asserted. The control lines, DramRdEn(Odd/Even) and DramWrEn(Odd/Even) for the FCT245-Type transceivers operate slightly differently than for FCT260- or FCT543-Type transceivers as will be explained later this chapter in the sections on “Interleaved Reads,” “Interleaved Writes,” and “System Examples”.

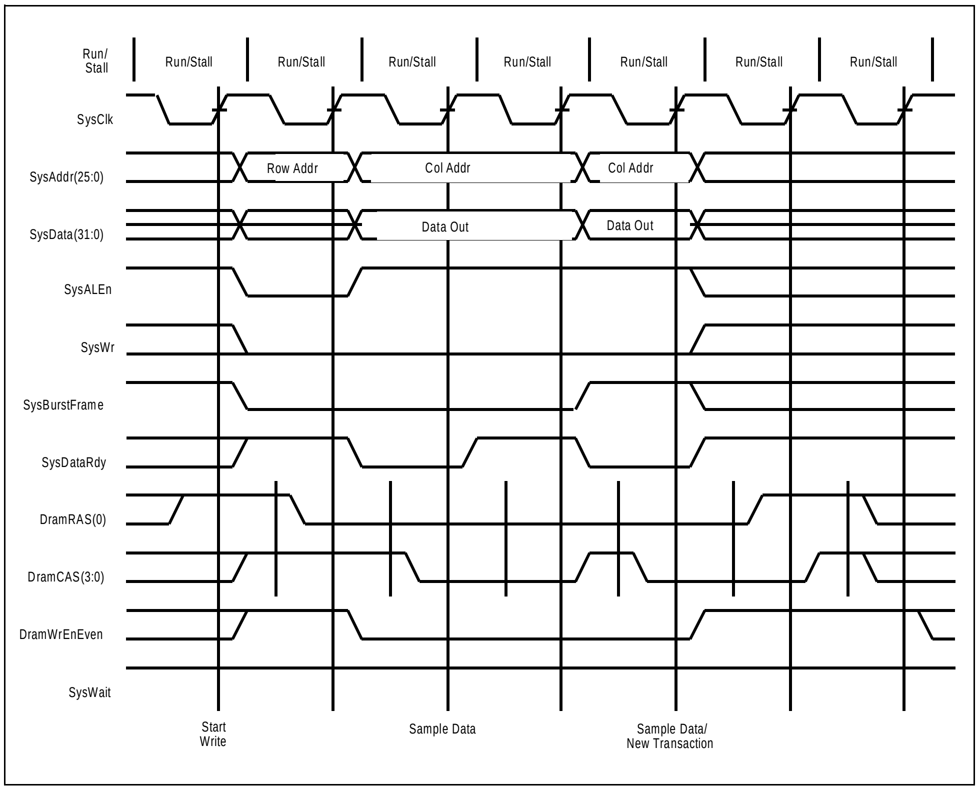


Figure 10.21 Two Datum Write

Interleaved Read Timing Diagrams

Interleaved FCT245 Reads

If the DRAM LSB Control Register Type Field of a pair of chip select channels is programmed to the 'Interleaved FCT245' setting, then the DRAM Controller assumes the timing shown in Figure 10.22 for reads. Because data is not latched by the transceiver, the 2nd, 4th, 6th,... datum must be read with a constant address and CAS assertion. Thus the interleaved FCT245 case saves 1 clock per odd datum over the non-interleaved case. In the Interleaved FCT245 case, the read enables DramRdEn(Odd,Even) are used as transceiver enables on both reads and writes. The Dram-WrEn(Odd,Even) signals can be used for the direction.

Notes

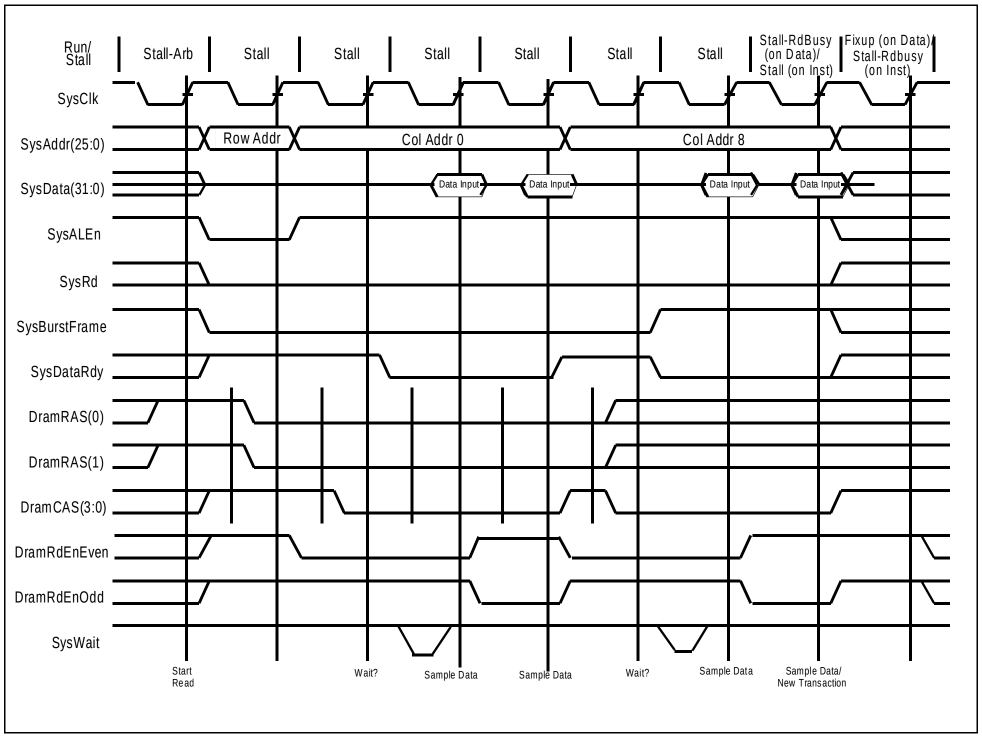


Figure 10.22 Interleaved ‘FCT245 type read

Interleaved FCT260 Reads

If the DRAM LSB Control Register Type Field of a pair of chip select channels is programmed to the 'Interleaved FCT260' setting, then the DRAM Controller assumes the timing shown in Figure 10.23 for reads. It is assumed odd datum are latched by the multiplexer, such that the next address and $\overline{\text{CAS}}$ lines (for the even datum) can be pipelined to change 1 clock early. Thus, the interleaved FCT260 case saves at least 3 clocks for each 4-word burst read. In the Interleaved FCT260 case, the even read enable $\overline{\text{DramRdEnEven}}$ is used to latch the odd datum while the odd read enable $\overline{\text{Dram-RdEnOdd}}$ is used as the overall read enable for the multiplexer.

Notes

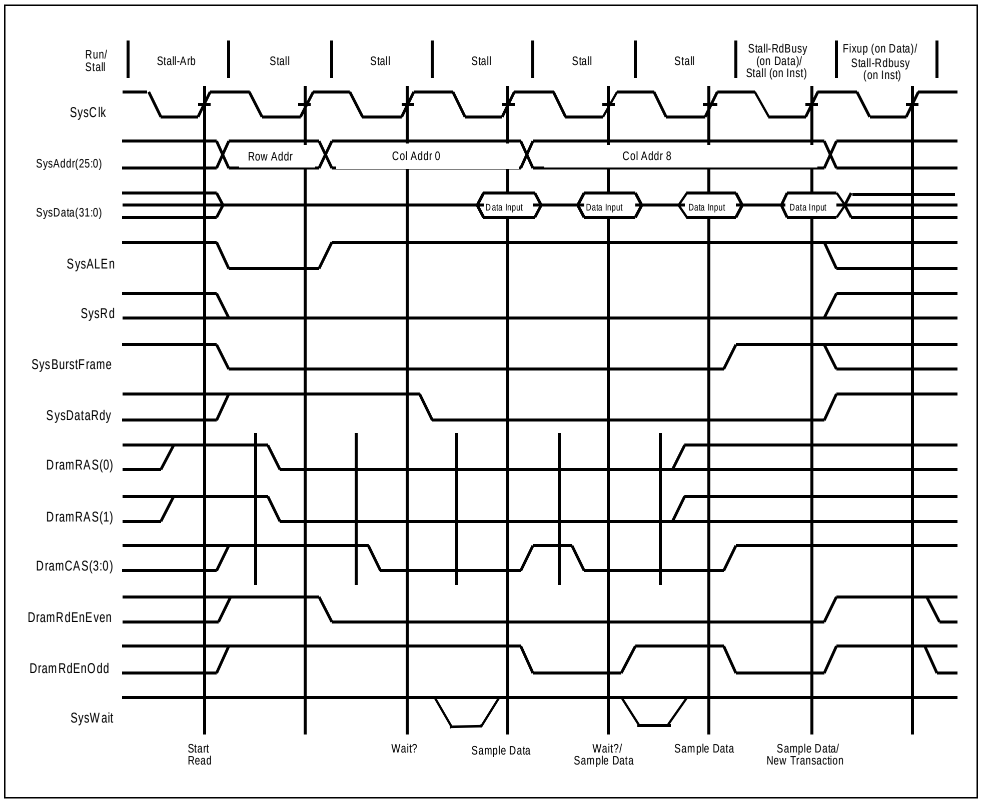


Figure 10.23 Interleaved FCT260 Read

Figure 10.24 shows a single datum access to an interleaved memory system using FCT260-type multiplexers in the data path. Note that the timing of this access is identical to the timing of the first word access of a quad word read.

Notes

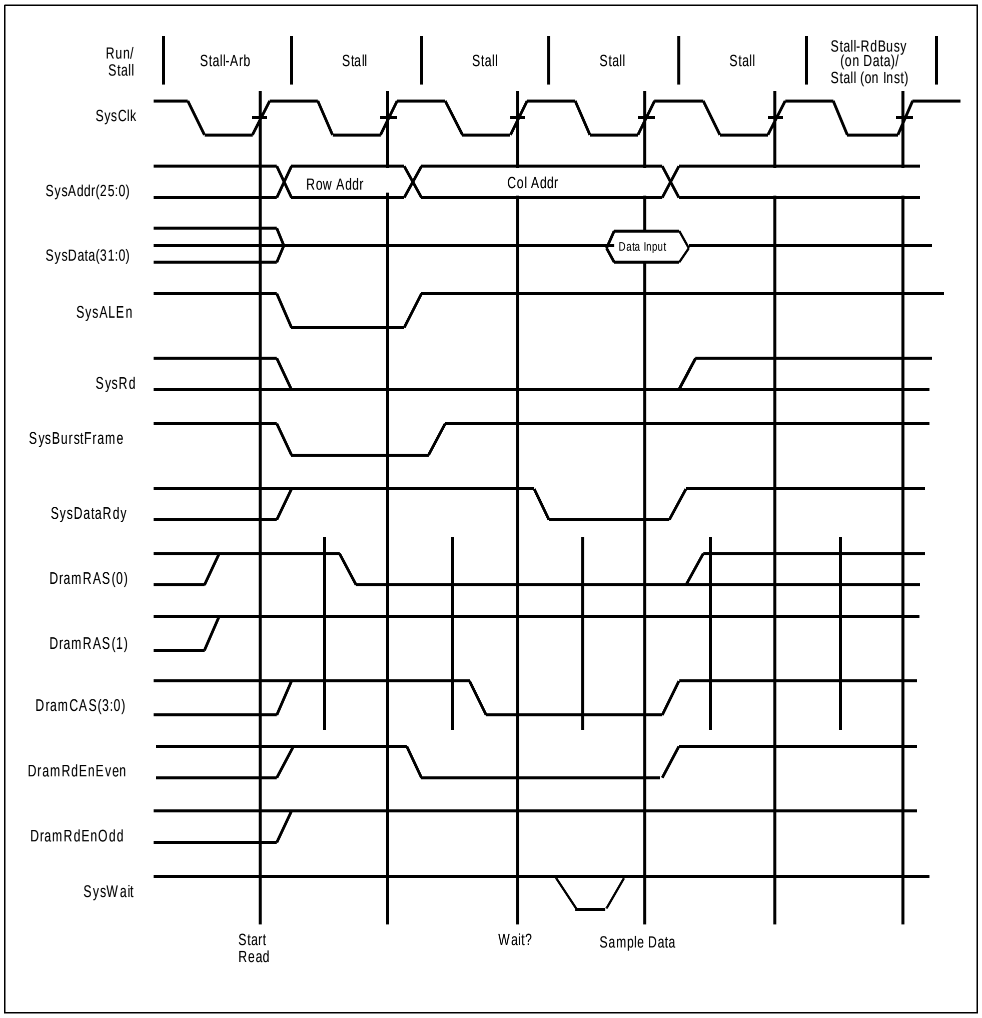


Figure 10.24 Single word access to even bank of FTC260-type system

Figure 10.25 shows the analogous access to the “odd” bank of an interleaved FCT260-type memory system. In this figure, the timing is identical with the timing of the access of the *second* word of a 4-word access; however, the first word is not actually returned to the CPU.

Therefore, there is a performance difference between even and odd single-word accesses, due to a limitation on the number of transceiver control pins available. However, for the following reasons, this should not adversely affect system performance:

- *Single word accesses occur for uncached instruction or data fetches. These are typically not used in performance critical parts of the system software.*
- *Cached instruction misses are always satisfied using 4-word refills, and utilize instruction streaming to resume execution once the critical missing instruction is returned from memory.*
- *Single word accesses may be used for cached data refills, if the data block refill parameter is set accordingly. However, the use of an interleaved memory in the first place indicates that the burst performance of the memory system is very high, leading to an extremely high probability that 4-word D-cache refill is used. For more information, refer to the DBlockRefill (‘DBR’) explanation located in the Coprocessor 0 Configuration section in Chapter 5.*

Notes

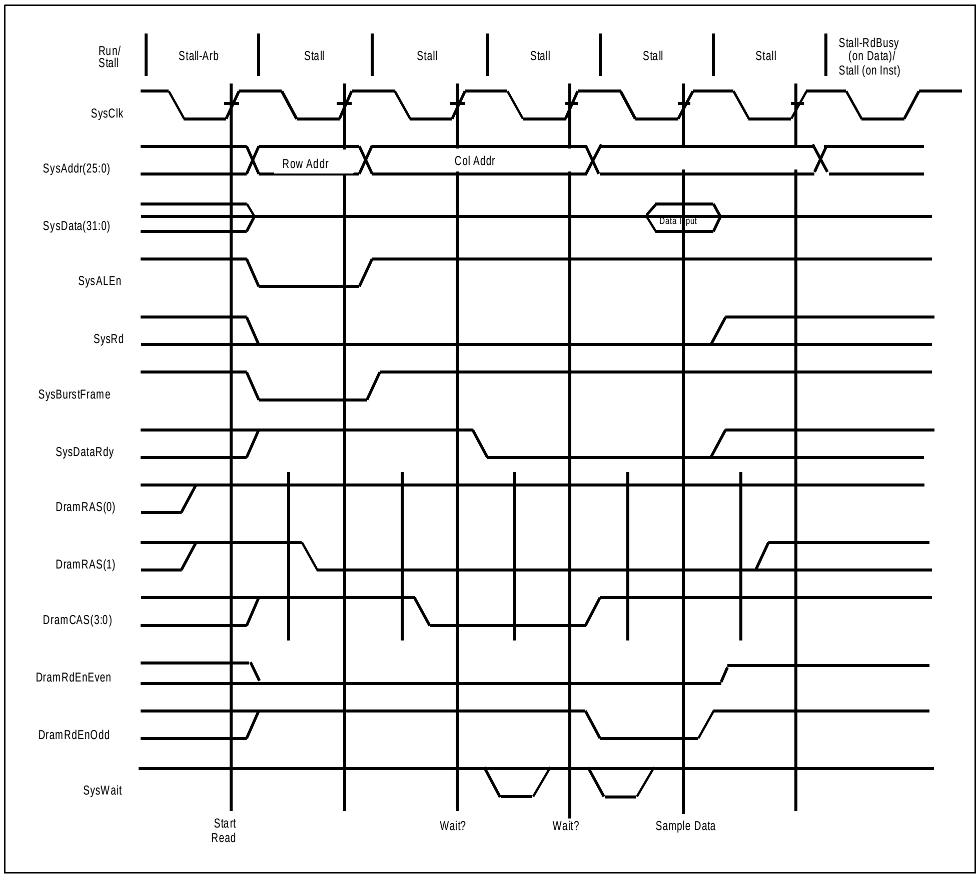


Figure 10.25 Single word access to odd bank of FCT260-type system

Interleaved FCT543 Reads

If the DRAM LSB Control Register Type Field of a pair of chip select channels is programmed to the 'Interleaved FCT543' setting, then the DRAM Controller assumes the timing shown in Figure 10.26 for reads. It is assumed odd datum are latched by the registered transceiver, such that the next address and $\overline{\text{CAS}}$ lines (for the even datum) can be pipelined to change 1 clock early. Thus the interleaved FCT543 case saves at least 3 clocks for each 4-word burst read. In the Interleaved FCT543 case, the two read enables and two write enables match up with the FCT543 part directly.

Notes

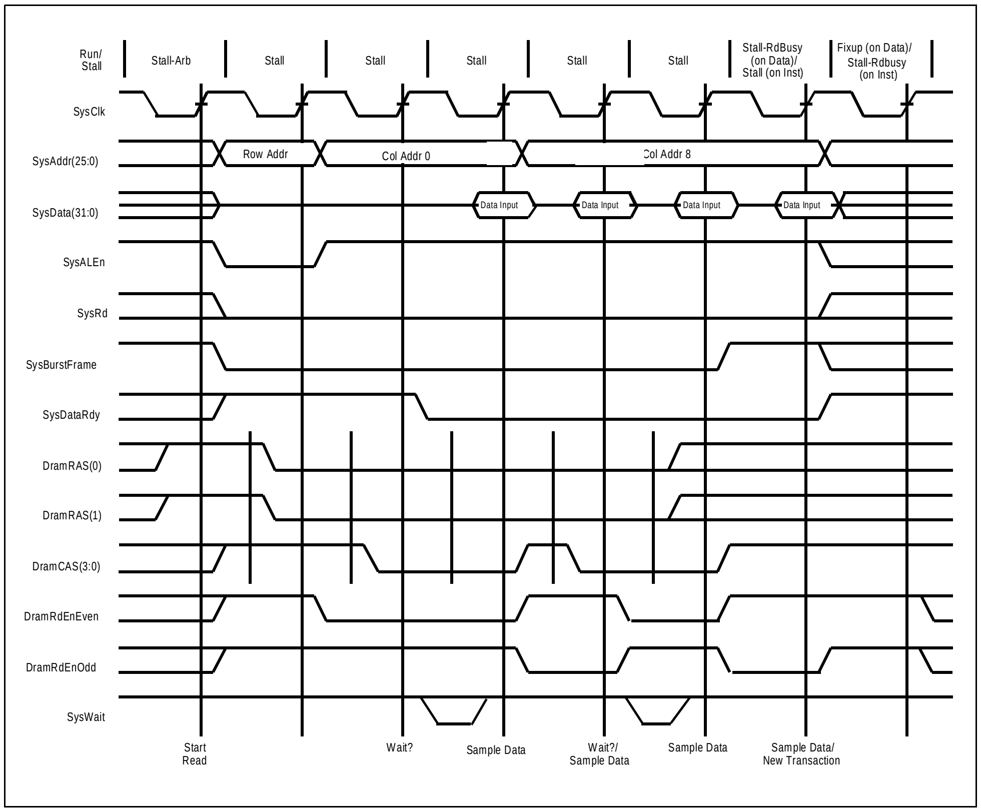


Figure 10.26 Interleaved FCT543 Read

Figure 10.27 shows a single datum access to an interleaved memory system using FCT543-type multiplexers in the data path. Note that the timing of this access is identical with the timing of the first word access of a quad word read.

Notes

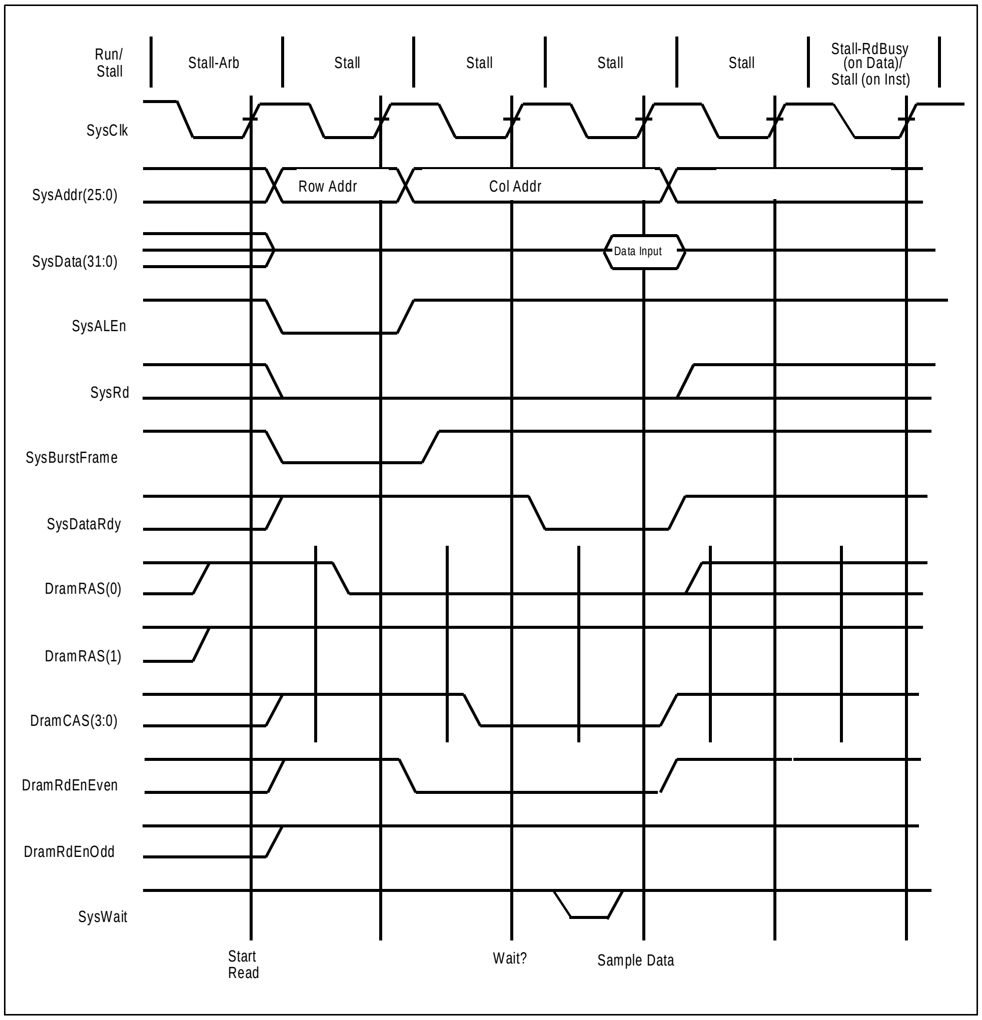


Figure 10.27 Single word access to even bank of FCT543-type system

Figure 10.28 shows the analogous access to the “odd” bank of an interleaved FCT543-type memory system. In this figure, the timing is identical with the timing of the access of the *second* word of a 4-word access; however, the first word is not actually returned to the CPU.

Thus, there is a performance difference between even and odd single word accesses, due to a limitation on the number of transceiver control pins available. However, for the following reasons, this should not adversely affect system performance:

- *Single word accesses occur for uncached instruction or data fetches. These are typically not used in performance critical parts of the system software.*
- *Cached instruction misses are always satisfied using 4-word refills, and utilize instruction streaming to resume execution once the critical missing instruction is returned from memory.*
- *Single word accesses may be used for cached data refills, if the data block refill parameter is set accordingly. However, the use of an interleaved memory in the first place indicates that the burst performance of the memory system is very high, leading to an extremely high probability that 4-word D-cache refill is used. For more information, refer to the DBlockRefil (‘DBR’) option in the Coprocessor 0 Cache Configuration section of Chapter 5.*

Notes

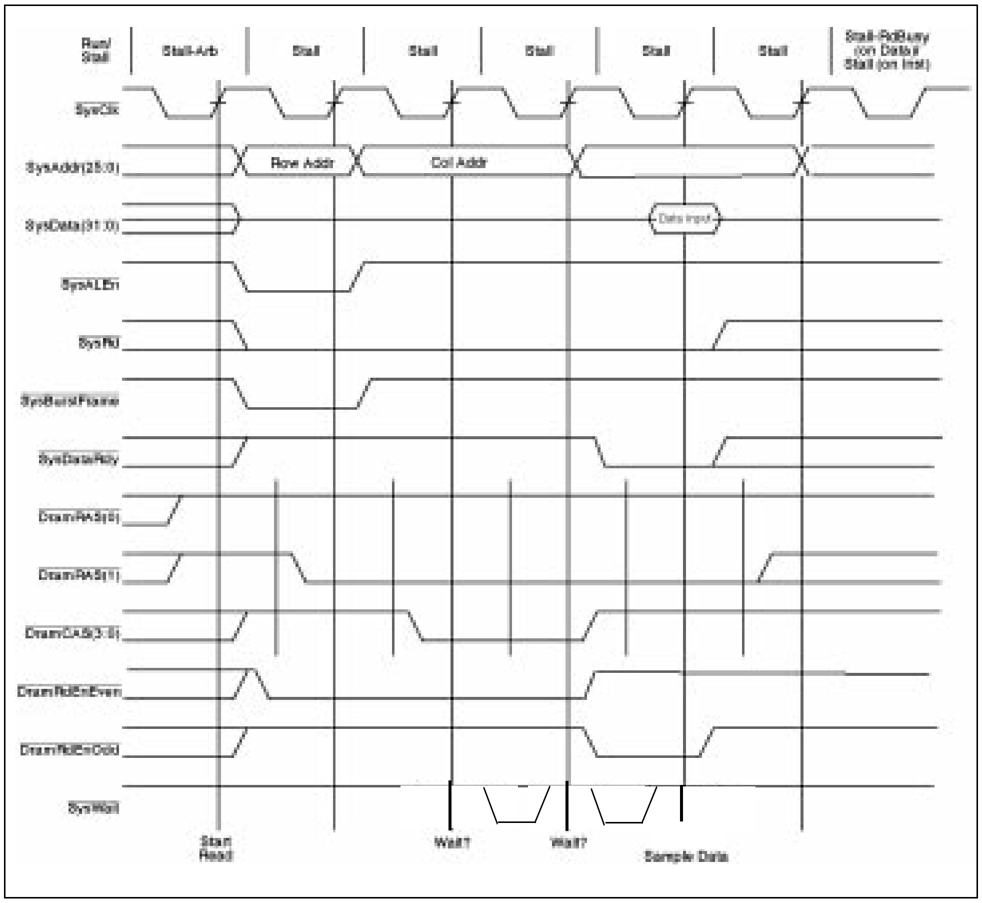


Figure 10.28 Single word access to odd bank of FCT543-type system

Interleaved Writes

Interleaved writes on the RC36100 occur one word at a time on the respective bank. The RC36100 CPU core is only capable of issuing one write at a time. However, the DMA engines are capable of issuing burst writes. At present, such burst writes are not highly optimized on the RC36100 and issue sequentially one after another with separate RAS (as well as CAS) strobes, switching between banks. This choice is due to the leading edge of CAS needing to be delayed for early writes on fully optimized bursts, which would cause needless complications for more typical systems.

Single Word Interleaved FCT245 Write

If the DRAM Control MSB Register Type Field of a pair of chip select channels is programmed to the 'Interleaved FCT245' setting, then the DRAM Controller assumes the timing similar to that shown in the first half of Figure 10.29 on page 42 on writes.

In the Interleaved FCT245 case, the read enables DramRdEn(Odd/Even) are used as transceiver enables on both reads and writes. The DramWrEn(Odd/Even) signals can be used for the direction. The Single Word case is similar to the multi-word case, except that the second assertion of CAS does not occur.

Interleaved FCT245 Writes

If the DRAM Control MSB Register Type Field of a pair of chip select channels is programmed to the 'Interleaved FCT245' setting, then the DRAM Controller assumes the timing shown in Figure 1.29 on writes.

On interleaved writes, the RC36100 DRAM Controller does the writes with 'early writes' and thus if a burst write occurs, separate CAS strobes occur for each datum.

Notes

In the Interleaved FCT245 case, the read enables DramRdEn(Odd/Even) are used as transceiver enables on both reads and writes. The DramWrEn(Odd/Even) signals can be used for the direction.

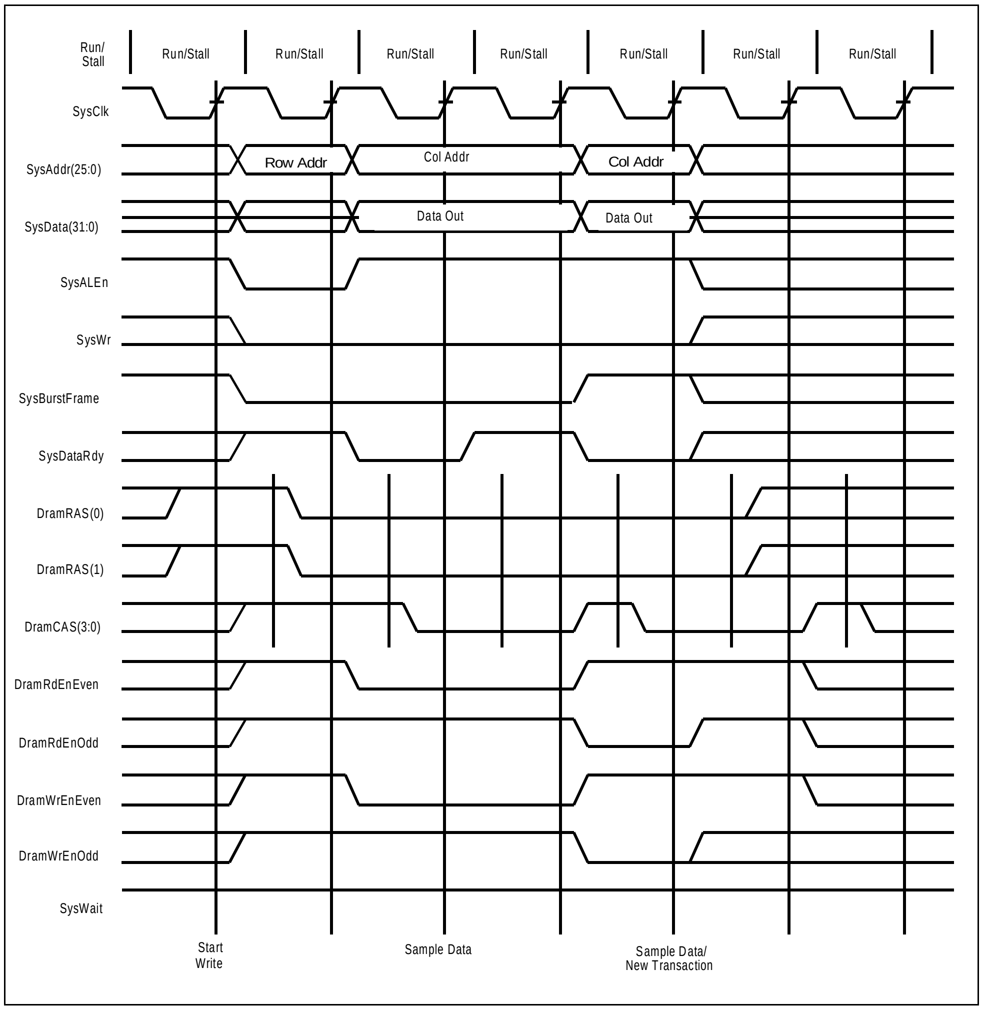


Figure 10.29 Interleaved FCT245-type Writes

Single Word Interleaved FCT260 Write

If the DRAM Control MSB Register Type Field of a pair of chip select channels is programmed to the 'Interleaved FCT260' setting, then the DRAM Controller assumes the timing shown in on writes.

On interleaved writes, the RC36100 DRAM Controller does the writes with 'early writes' and thus if a burst write occurs, separate CAS strobes occur for each datum.

In the Interleaved FCT260 case, the even read enable DramRdEnEven is used to latch the odd datum while the odd read enable DramRdEnOdd is used as the overall read enable for the multiplexer.

The Single Word case is similar to the multi-word case, except that the second assertion of CAS does not occur.

Interleaved FCT260 Writes

If the DRAM Control MSB Register Type Field of a pair of chip select channels is programmed to the 'Interleaved FCT260' setting, then the DRAM Controller assumes the timing shown in Figure 10.30 on writes.

On interleaved writes, the RC36100 DRAM Controller does the writes with 'early writes' and thus if a burst write occurs, separate CAS strobes occur for each datum.

Notes

In the Interleaved FCT260 case, the even read enable $\overline{\text{DramRdEnEven}}$ is used to latch the odd datum while the odd read enable $\overline{\text{DramRdEnOdd}}$ is used as the overall read enable for the multiplexer.

Interleaved FCT543 Writes

If the DRAM Control MSB Register Type Field of a pair of chip select channels is programmed to the 'Interleaved FCT543' setting, then the DRAM Controller assumes the timing shown in Figure 10.30 on writes.

On interleaved writes, the RC36100 DRAM Controller does the writes with'early writes' and thus if a burst write occurs, separate $\overline{\text{CAS}}$ strobes occur for each datum.

In the Interleaved FCT543 case, the two read enables and two write enables match up with the FCT543 part directly.

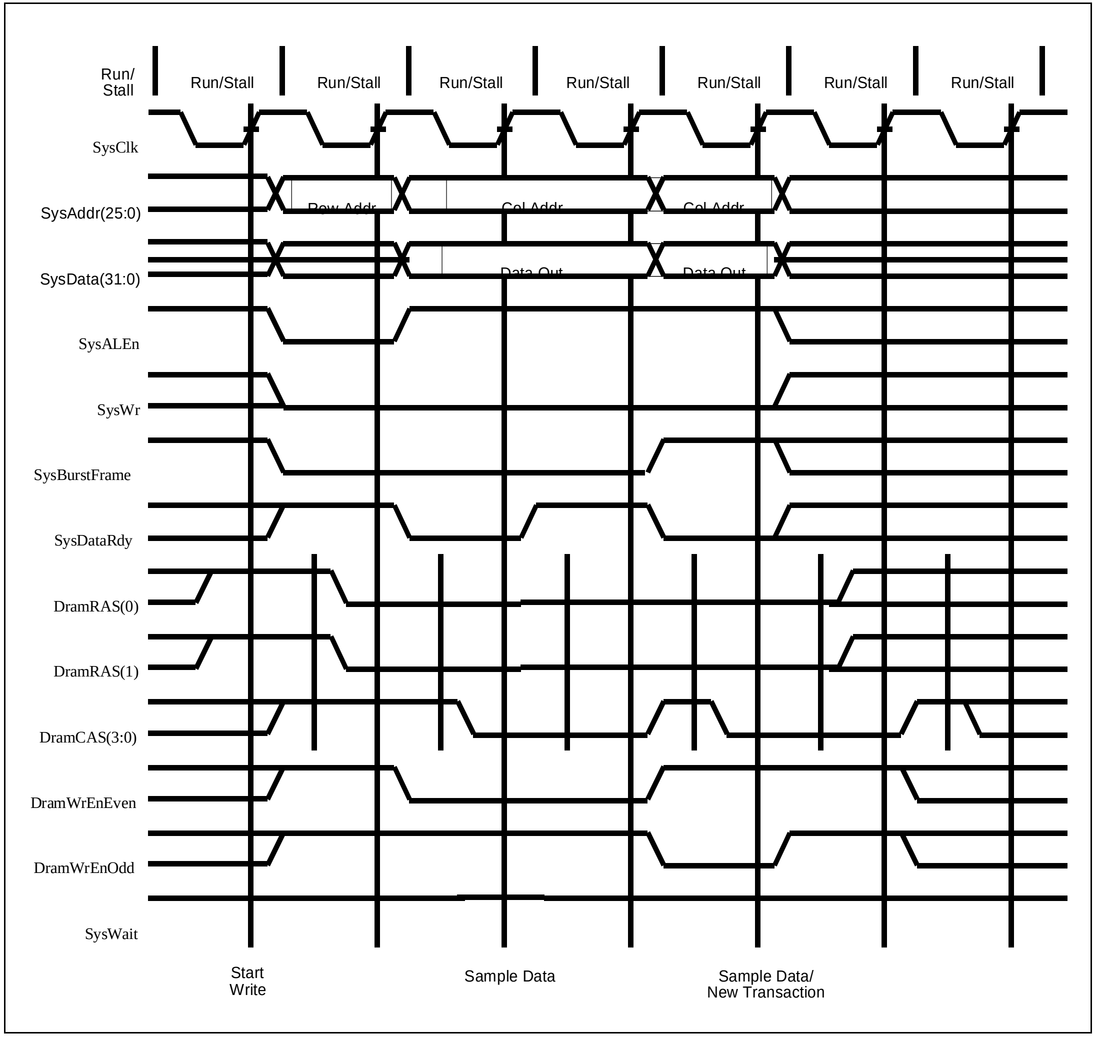


Figure 10.30 Interleaved FCT260, FCT543-type Writes

Notes

Refresh

The refresh cycle of DRAM chips is supported by the DRAM Controller by using the CAS-before-RAS refresh protocol. All four DramCAS() lines are asserted for CASW time followed after 1 clock by asserting the even DramRAS() lines for CASW+0.5 time followed (staggered) by the odd DramRAS() lines asserting for CASW+0.5 clocks time. All four DramCAS() lines, as shown in Figure 10.31, de-assert 1.5 clocks after the odd DramRAS() lines assert. Staggering the RAS lines allows the peak power consumption of turning DRAM chips on to be minimized. The DRAM Controller guarantees that the write enables, DramWrEn(Odd/Even) are de-asserted during refreshes to avoid entry into an internal test mode of higher density (4-16Mbit) DRAM chips.

Refresh cycles can occur in parallel with non-DRAM accesses. If a CPU or DMA transfer requiring DRAM occurs concurrently or after a refresh, the refresh has priority and will complete first.

Because refreshes can happen in parallel with non-DRAM accesses, the RC36100 Debug Interface provides a DiagNoCS() signal to decode precisely when a load or store occurs when no chip select (or RAS line) is asserted for a load/store transaction.

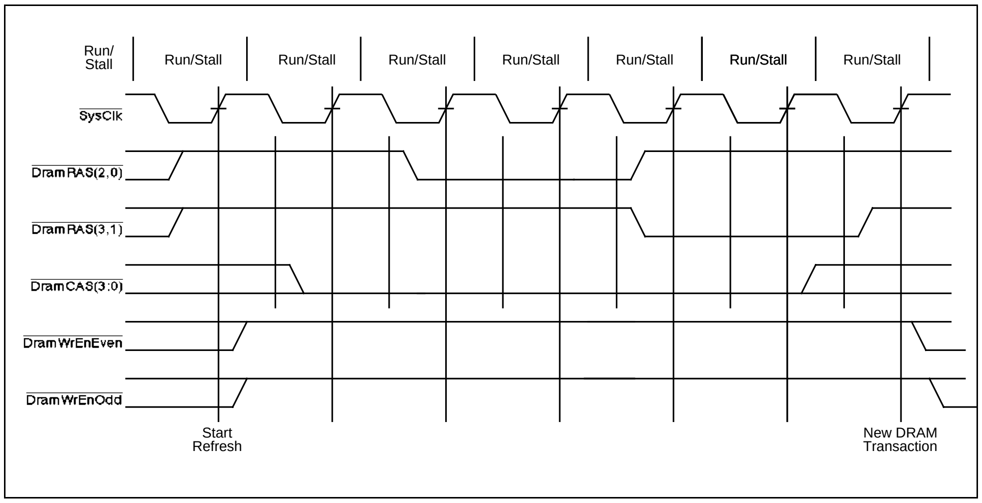


Figure 10.31 DRAM Staggered Refresh

System Examples

The following DRAM systems concentrate on distinguishing the data path connections between the three different DRAM types:

- *FCT245 Transceiver Type*
- *FCT260 Latched Multiplexer (Bus Exchanger) Type*
- *FCT543 Registered Transceiver Type*

The address path of a particular system will depend on the total number of loads the address bus needs to drive. Typically there are 8 DRAM chips per bank, each of them with an address connection. Assuming that a ROM bank is also connected, that is already 12 loads. Assuming that the DRAM and the EPROM are CMOS type input loads (micro-amps), typically the drive current from the RC36100 is rarely an issue.

However, as the number of loads gets larger, the output propagation delay will also have a capacitive delay factor as well as a noise factor from the trace length. If more than about 8 loads are connected to the SysAddr bus, then allowances in the programmable timing settings of the DRAM Controller should be made for ringing and settling time as well as capacitive load delay derating. If optimal timing is still desired, then address buffers such as the FCT244, FCT344, or FCT827 can be used.

Notes

DRAM System using FCT245 Transceivers

DRAM Systems using FCT245 transceivers can be expanded from 1 bank to 4 banks. The first bank, even bank DramRAS(0), uses one set of transceivers and shares the transceiver set with the other optional even bank DramRAS(2). If present, the second bank, odd bank DramRAS(1), uses a separate set of transceivers and shares the transceiver set with the other optional odd bank DramRAS(3). The use of a second set of transceivers allows even and odd banks to be used in the interleaved mode.

In an FCT245 type system, DramRdEn(Odd,Even) are used as the common output enable. Thus DramRdEn(Odd,Even) for the FCT245 type, assert for both reads and writes and could be called, "DramEn(Odd,Even)." Because the FCT245 does not contain a latch, address pipelining optimization cannot occur. The use of SysRd or perhaps SysWr (depending on whether the data path from the CPU is A to B or perhaps B to A) avoids leading edge bus contention from direction to output enable skew. Note that the use of SysRd or SysWr on DRAM accesses may in the future, limit the use of the future use of DMA fly by accesses. (The RC36100 does not presently support DMA fly by accesses. The other DRAM types described below do not use SysRd or SysWr).

Note that the RC36100 DRAM Controller depends on CAS without RAS having no effect (always true of standard DRAMs, since some chips do not have dedicated output enable pins) in order to share transceivers between DRAM chip banks.

With present day pricing, the FCT245 type system is the least expensive interleaved option, however, it is not as fast on burst reads as the other two types.

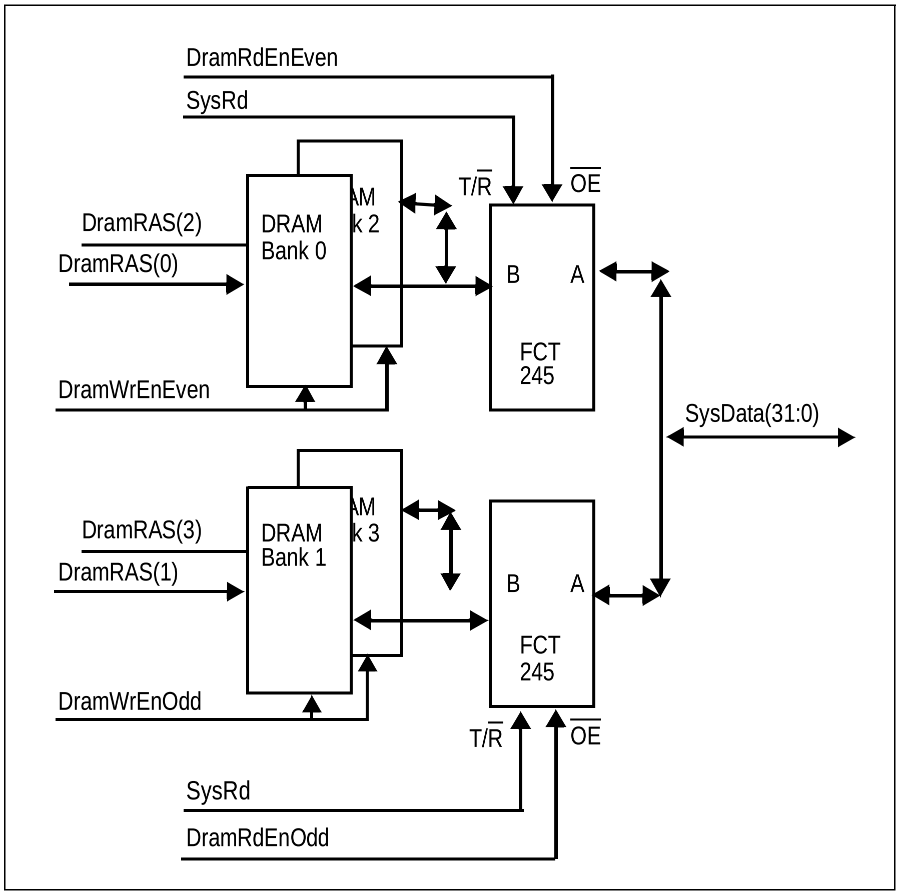


Figure 10.32 Interleaved FCT245 Interface

Low Cost DRAM System using FCT245 Transceivers

In very low cost systems that do not need the extra throughput of interleaving, a single set of transceivers can be used for all 4 banks. However, this requires that the banks not be put into their software programmed interleaved mode and that the read enables, DramRdEn(Odd,Even) be externally OR'ed.

Notes

Very Low Cost DRAM System without Transceivers

In simpler systems, it is also possible to remove the transceivers completely, such that the DRAM bank is attached directly to the SysData bus. The Bus Turn Around setting can be adjusted to prevent bus contention between DRAM chips and the CPU on a DRAM read followed by a CPU write. For more information, refer to “Dram Read Cycle Bus Turn-Around (‘DramRdBTA’) Field Encoding” (Table 10.22 on page 20).

DRAM System using FCT260 Multiplexers

DRAM Systems using a set of FCT260 latched multiplexers can be expanded from 1 bank to 4 banks. The even banks share one data path while the odd banks share the other data path.

In an FCT260 system, DramRdEnEven is used as the common read data path enable to the CPU. DramRdEnEven for the FCT260 type, asserts for both even and odd reads and could be called, "DramRdEn." DramRdEnOdd is used to latch the odd read data temporarily so that address pipelining can occur. DramRdEnOdd is also used for the FCT260 path select. The DRAM write enables, DramWrEn(Odd/Even) are hooked up in a straightforward manner, to the odd and even write data path enables of the FCT260, respectively.

The FCT260 system (see Figure 10.33 for system diagram) is one of the least expensive interleaved options, since just 3 chips are required instead of 4 chips. In addition, burst reads are fully optimized with address pipelining, and thus save an additional clock on each burst read relative to a FCT245 system. Thus for many multi-bank systems, the FCT260 system is the best cost/performance alternative.

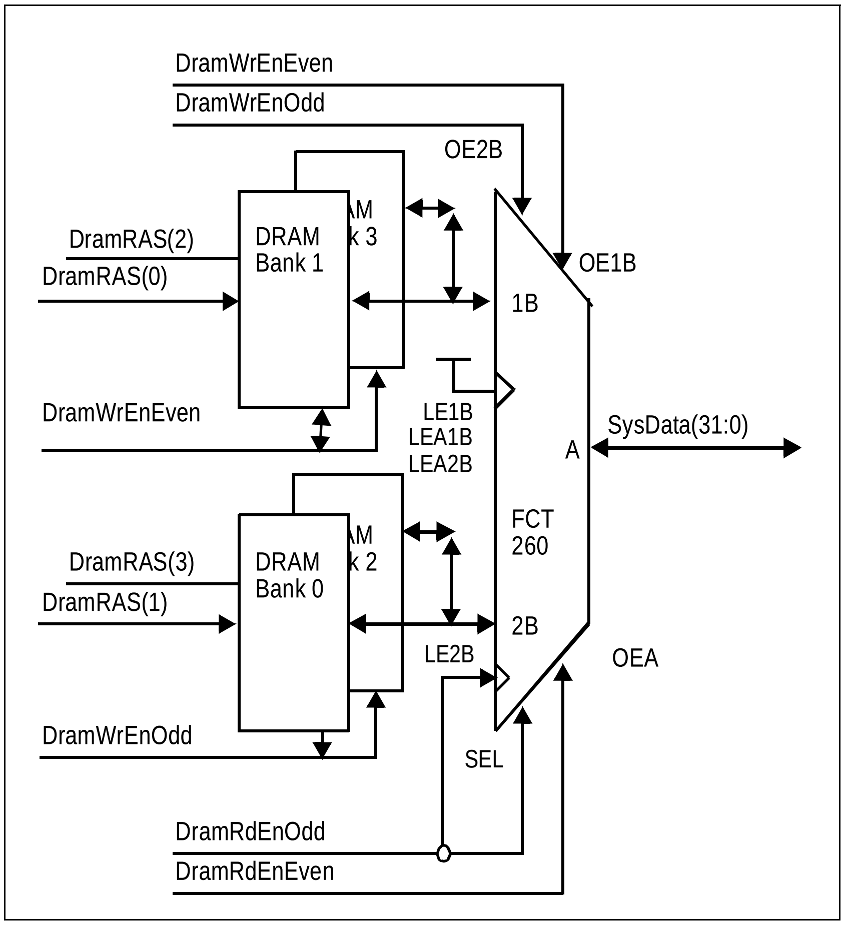


Figure 10.33 Interleaved FCT260 Interface

Notes

DRAM System using FCT543 Registered Transceivers

DRAM Systems using a set of FCT543 registered transceivers can be expanded from 1 bank to 4 banks. The even banks share one set of transceivers while the odd banks share another set of transceivers.

The FCT543 system may be more expensive than other options and has the same performance as the FCT260 option. Because the connections are more straightforward, and therefore easier to understand, the FCT543 option is mentioned here as an example system.

In the FCT543 system (see Figure 10.34 for system diagram), the read enables, $\overline{\text{DramRdEn(Odd/Even)}}$, are hooked up to the odd and even transceiver banks' read data path enables respectively. Likewise, the write enables, $\overline{\text{DramWrEn(Odd/Even)}}$, are hooked up to the odd and even transceiver banks' write data path enables respectively.

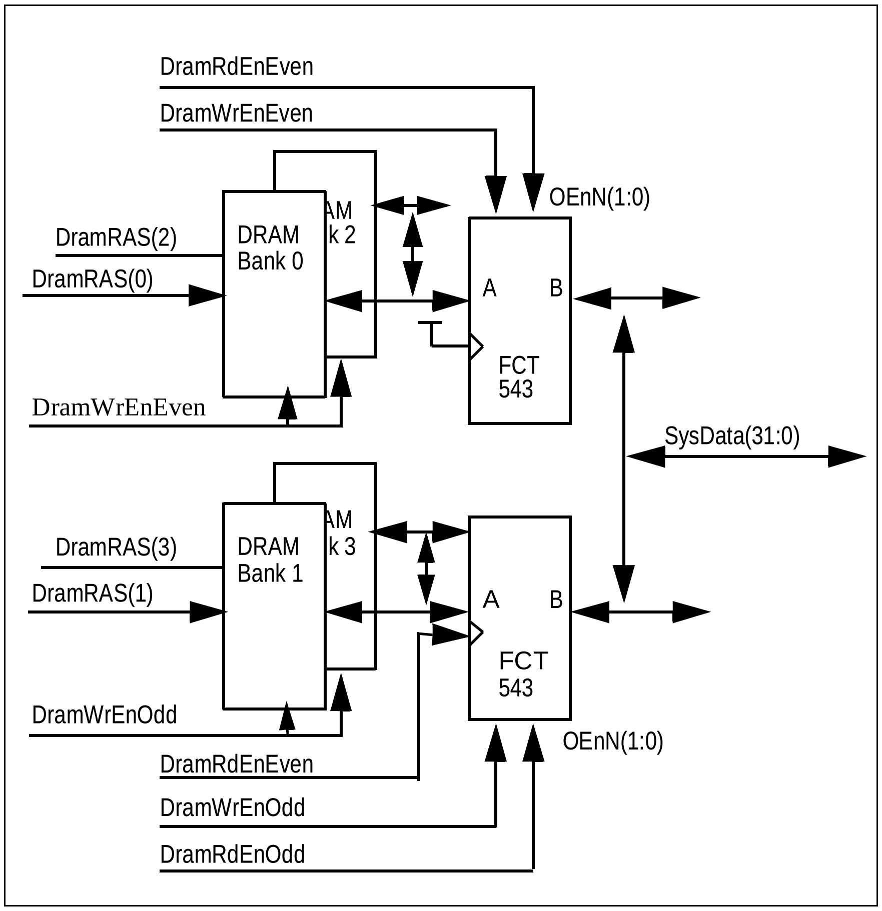


Figure 10.34 Interleaved FCT543 Interface

Notes

Direct Memory Access (DMA) Controller

Notes

Introduction

The IDT79RC36100 RISController integrates bus controllers and peripherals around the RISCore3000 family CPU core. One of the on-chip memory controllers is the **Direct Memory Access (DMA) Controller**.

This chapter includes a functional overview, complete pin descriptions, signal information, timing diagrams, register drawings and an explanation on how the DMA Controller interface relates to typical internal and external hardware DMA systems.

Features

- *4 internal channels*
  - *Slave-mode device support for using RC36100 controlled memory*
  - *Physical memory to physical memory transfers*
  - *Link chaining protocol, for consecutive transfers*
- *2 external channels*
  - *Master mode device support, for using RC36100 controlled memory*
  - *Physical memory to physical memory transfers*
- *Fixed priority arbitration*
- *Coordinates BIU port width, Endianess, Byte Enable, and Read Buffer logic*
- *Single word read/write mode*
- *4-word burst read/write mode*

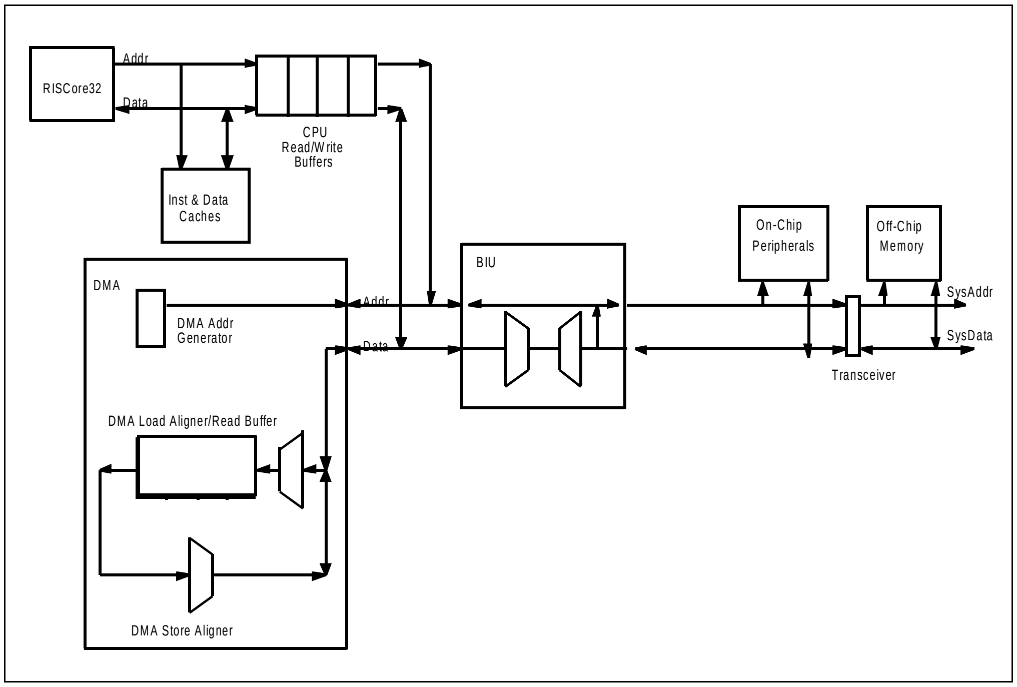


Figure 11.1 DMA Controller Address and Data Flow Diagram.

Notes

Block Diagram Overview

The functional block diagram for the address and data paths of the DMA Controller is shown in Figure 11.1. The DMA Controller—as one of the bus controllers—coordinates and shares the Bus Interface Unit (BIU) resources with the CPU; however, when the DMA Controller channels can make use of the BIU, an arbitration unit (not shown) coordinates resource sharing.

When an internal DMA Controller channel is granted the BIU, it first uses the DMA Addr Generator to put a source address out to the BIU. The BIU then generates a read to the System Interface. The various BIU control signals, including Endianess and AccTyp (Byte Enables and Burst Length), are also coordinated by the DMA Controller.

The System Interface executes the read from the source (for instance, to DRAM or to one of the on-chip peripherals). If the source read data is a different port width than 32-bits, then the BIU takes care of the byte gathering and the Read Buffer takes care of burst gathering. The source data is FIFO'ed into the Read Buffer, similar to a CPU read.

At this point, the DMA Controller takes the read data from the Read Buffer and generates the target address. The read data is then sent to the BIU in 32-bit quantities—with the proper Endianess and AccTyp (Byte Enables and Burst Length) and target address—until the Read Buffer is emptied. On each BIU write, the System Interface is invoked and the write is completed out to the target (for instance, to one of the on-chip peripherals or to the off-chip DRAM).

Thus, on each DMA transaction, the DMA Controller first generates a read from the source address into its own 4-word FIFO and then generates a write to the target address, using data from the 4-word FIFO.

Functional Overview

The Direct Memory Access (DMA) controller has two basic functions:

- *An internal DMA function is capable of internally generating addresses and controlling a read/write pair for each data transfer.*
- *An external DMA function loads an address from an external DMA controller and then generates a fly-by read or fly-by write for each data transfer.*

Internal DMA Channels

In the RC36100, there are four independent DMA Channels. Each of the DMA channels is functionally identical—with the exception of their priority encoding—and initialized with a set of chaining registers to determine:

- *the DMA source start base address*
- *the DMA target start base address*
- *the data transfer number*
- *the protocol style selection*

Thus, the programmable link chaining registers act as a set of instructions that the DMA channel must execute and complete. After completing the link chaining register instructions, the DMA channel may be instructed to stop and interrupt the CPU core, or it may be instructed to load a new set of link chaining register instructions.

At the beginning of a DMA transaction, the channel will first arbitrate for the system bus. With multiple DMA requests pending, after a DMA access, the bus is granted to the CPU instead of to the next highest requester. Thus, there are two priority tiers: (1) high/ ExtDMA and IntDMA and (2) low/ CPU.

Within these two tiers, the priority algorithm is either fixed or rotating. However, outside these tiers, after each arbitration, the bus is given to the highest requesting tier. Once within a tier, that tier may choose to keep the bus within the tier (for example, IntDMA keep bus mode). Specific to the ExtDMA/IntDMA, once a channel is done, the tier arbitration token is given to the other tiers, At the next arbitration point, to grant a specific DMA channel, if the ExtDMA/IntDMA receives the bus, the ExtDMA/IntDMA will use either the fixed or rotating priority scheme. Figure 11.2 illustrates the rotating priority scheme, and the fixed priority encodings are listed in Table 11.1.

Notes

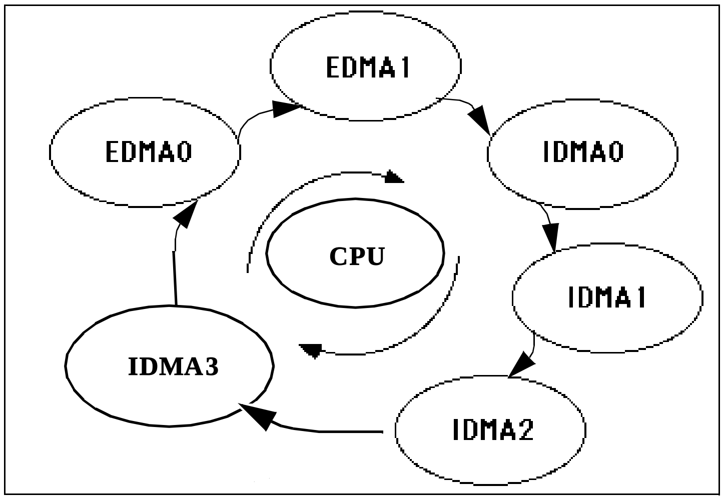


Figure 11.2 Rotating Priority Scheme

Fixed Priority	Agent
Highest	EDMA0
•	EDMA1
•	IDMA0
•	IDMA1
•	IDMA2
Lowest	IDMA3

Table 11.1 Fixed Priority Encoding

Once arbitration is granted, the DMA channel generates a read cycle with the source base address. The control register determines whether or not it is a burst. Typically, the source address will be through an internal memory controller on the RC36100 (for example, the DRAM Controller). Thus the internal memory controller will take the address and generate data, acknowledges, etc., back to the DMA controller channel. The DMA controller uses the DMA 4-word deep buffer FIFO to absorb the potential burst read data.

After the read is completed, the DMA channel initiates a write to the target address, by emptying out the read buffer FIFO. As with the read, the write is typically through an internal memory controller on the RC36100 (for example, the I/O Controller). The internal memory controller will take the address and data from the FIFO and generate a write transaction.

At the end of the transaction, the DMA channel's count register is decremented by 1. If the count register has not reached 0, the source and target addresses are incremented to their next value (which could be by +0, +1, +2, +4, or +16 depending on whether incrementing is enabled and whether a mini-burst or burst occurred).

If the count register has reached 0, then the DMA channel is finished with its current link chaining register assignment. If the control register so instructs, the channel may set an interrupt and/or stop, and/or it may reload a new link/set of chaining registers. If a new link/set is loaded, then the DMA channel will repeat the basic DMA channel transaction by copying the new link instructions into the current instructions and executing them.

Internal DMA Algorithm

Figure 11.3 shows the internal DMA algorithm.

Notes

```
while (stop_field == false) /* note that if at anytime stop_field == true then break */
{
if (count_field != 0)
/* Start up */
if (wait_for_interrupt_field == true)
    {while (DMAInterruptN == false)
    { /* wait for DMAInterruptN */; } }

assert BusReqN = active_low;
while (BusGntN == non_active_low)
    {/* wait for BusGntN */;}

/* Do the source read */
BusInterfaceUnit(source_addr_field, BurstN,
    BEnN(3:0), BigEndianFlag, RdN);
while (Bus_Interface_Unit(FIFO_Data_WrN)) {
    FIFO_Data[] = Data;
}
/* Do the target write */
BusInterfaceUnit(target_addr_field, Burst,
BEnN(3:0), BigEndianFlag, WrN);
while (Bus_Interface_Unit(FIFO_Data_RdN))
{Data = *FIFO_Data++;}

/* Finish up */
if (keep_bus_field == false)
assert BusReqN = non_active_low;
count_field = count_field - 1;
if (source_inc_field == true)
{source_addr_field = source_addr_field + burst_length_field;}
if (target_inc_field == true)
{target_addr_field = target_addr_field + burst_length_field;}
if (wait_for_interrupt_field == false)
    {break}
} /* if count != 0 */
else { /* count == 0 */
assert ExcInt(DMA_Done_Int()); /* pulse DMADoneN for 1 clock */
if (break_field == true) {
assert Stop_Field == true;
break;
}
else { /* break == false */
case link_field {
0:  {DMA_Registers = LinkA_Registers;
break /* from case */; }
1:  {DMA_Registers = LinkB_Registers;
 break /* from case */; }
2:  {DMA_Registers = LinkC_Registers;
        break /* from case */; }
3:  {DMA_Registers = LinkD_Registers;
        break /* from case */; }
}
}
} /* end count == 0 */
```

Figure 11.3 Internal DMA Algorithm.

External DMA Channels

External DMA channels are conceptually simpler than their internal counterparts. Much of the control logic is implemented by a user supplied external off-chip DMA controller agent; so essentially, all the RC36100 is required to do is get off the system bus and react to reads and writes to internal memory controllers.

Notes

Thus, the RC36100 first gives the bus to the external DMA Agent which issues either a read or write command to the RC36100. The external DMA agent then gets off the address and control bus. The RC36100 then executes the command on one of the memory controllers and does fly-by data accesses where the external DMA agent either reads or writes the data at the same time the memory controller writes or reads the data.

To allow an external DMA agent to take control of the bus, the external DMA Controller uses the customary bus request/grant, DMABusReq() and DMABusGnt(), handshake signals. When the DMA agent takes control of the bus, it drives address and control information onto the SysData pins of the RC36100. SysWr is asserted if the transaction is to be a write, de-asserted if the transaction is to be a read. Single word writes must set the byte enables via MemWrEn(3:0).

All write enable signals become byte enable inputs, when performing external DMAs. Also, during ExtDMA commands, the user must assert all four byte enables for word access, the DRAM or Memory controller will not perform data packing, and the lower order address lines (A1:A0) will be ignored.

Note that internal peripherals must use single word reads, not burst reads. SysALEn may remain asserted for more than one clock; however, the address is latched in on the first rising clock edge where SysALEn is asserted. This allows SysALEn to follow the PCI FRAME# convention and to delay an access. In PCI mode, MemWrEn(3:0) are sampled on the clock after SysALEn is first asserted. The SysData bus is used to get the external DMA 32-bit physical address.

If the physical address corresponds to an on-chip controller and the transfer is a read, the RC36100 generates a read from the proper device and drives the necessary data lines with the data. If the transaction is a burst read or burst write, to indicate the end of the transaction, the accesses must be full word reads or writes and use a properly programmed DMADone input. DMADone must be asserted on the falling edge of SysClk, to end ExtDMA burst accesses.

Because the RC36100 must reuse the SysAddr and Sys Control lines, during the second half of a DMA access, the DMA agent must tri-state its address (and data transceivers) after driving SysAddr, SysALEn, SysBurstFrame, SysWr, and MemWrEn(3:0) into the RC36100. The tri-stating must occur by the second clock after de-asserting SysALEn. Before taking the bus back, the CPU drives all control lines de-asserted; therefore, the DMA agent doesn't necessarily have to do so. With the exception of write SysData(), the RC36100 will take over the bus to do a memory cycle. When the RC36100 has completed its internal memory cycle, it will assert SysDataRdy and de-assert SysAddr and all System Control lines.

Burst transfers may be from 1 to 64 words and must be aligned with a 64-word block. Burst transfer mode also requires the use of the DMADone pin. DMABusReq() must be kept asserted at least one clock after DMABusGnt() asserts. DMABusReq() must be deasserted before the last SysDataRdy occurs, unless another external DMA transaction is to occur.

Pin Descriptions

Direct Memory Access (DMA) Controller Signals

$\overline{\text{DmaBusReq(1:0)}}$ Input

DMA Bus Request: Active low. Input signal to the RC36100 that the external DMA controller would like to gain mastership of the system bus. DmaBusReq can be software programmed to be active high by using the ReqH field in the External DMA Control Register.

$\overline{\text{DmaBusGnt(1:0)}}$ Output

DMA Bus Grant: Active low. Output signal to the external DMA controller that it is now master of the system bus.

$\overline{\text{DmaDone}}$ Input

DMA Done: Active low. Signals the RC36100 that the current DMA transaction is the last transaction by the current DMA agent.

Notes

For internal DMA, if DMADone is asserted, the present link will abort. For external DMA, DMADone will indicate that the next data is the last data on a burst read or write. During a burst transfer, DMADone is required.

Note: For a read cycle, DMADone should be activated at the beginning clock of the last word (datum). For a write cycle, DMADone should be asserted before the beginning clock of the last word (datum).

System Control Signals used during DMA Controller Accesses

SysALEn	**Output/(Input during External DMA)**
SysBurstFrame	**Output/(Input during External DMA)**
SysRd	**Output/(Input during External DMA)**
SysWr	**Output/(Input during External DMA)**
MemWrEn(3:0)	**Output/(Input during External DMA)**

During the initial part of an external DMA access, these signals are used to give the RC36100 a read, write, burst read, or burst write command. During this period, they are inputs.

SysRd:

SysRd can be driven optionally by the external DMA agent; however, it is ignored by the RC36100, which uses an unasserted SysWr to indicate a read command.

MemWrEn():

External DMA, in addition to the regular cases, can support the '1111' case which is equivalent to the '0000' all asserted case and the '1001' case which is a case the RISCore32 series core does not generate.

SysDataRdy Output

SysDataRdy asserts low whenever the CPU expects data to be read or written. It is always a CPU output. The datum is read or written by the CPU bus controller simultaneous with being written/read by the DMA agent.

Register Descriptions

The RC36100 DMA Controller has two sets of registers: one internal (four channels) and one external (two channels).

Internal DMA Controller Register Descriptions

Table 11.2 is an address map of the Internal DMA Controller registers. Big Endian software must offset these addresses by b’10 (0x2), if halfword accesses are used. All Internal DMA Registers are uninitialized, except for the on/off control bit15 in the LSB Control registers of channels 0-3.

Notes

Physical Address	Description
0xFFFF_E300	DMA LSB Source Address Register for Channel 0
0xFFFF_E304	DMA MSB Source Address Register for Channel 0
0xFFFF_E308	DMA LSB Target Address Register for Channel 0
0xFFFF_E30C	DMA MSB Target Address Register for Channel 0
0xFFFF_E310	DMA LSB Count Register for Channel 0
0xFFFF_E314	DMA MSB Count Register for Channel 0
0xFFFF_E318	DMA LSB Control Register for Channel 0
0xFFFF_E31C	DMA MSB Control Register for Channel 0
0xFFFF_E320	DMA LSB Source Address Register for Channel 1
0xFFFF_E324	DMA MSB Source Address Register for Channel 1
0xFFFF_E328	DMA LSB Target Address Register for Channel 1
0xFFFF_E32C	DMA MSB Target Address Register for Channel 1
0xFFFF_E330	DMA LSB Count Register for Channel 1
0xFFFF_E334	DMA MSB Count Register for Channel 1
0xFFFF_E338	DMA LSB Control Register for Channel 1
0xFFFF_E33C	DMA MSB Control Register for Channel 1
0xFFFF_E340	DMA LSB Source Address Register for Channel 2
0xFFFF_E344	DMA MSB Source Address Register for Channel 2
0xFFFF_E348	DMA LSB Target Address Register for Channel 2
0xFFFF_E34C	DMA MSB Target Address Register for Channel 2
0xFFFF_E350	DMA LSB Count Register for Channel 2
0xFFFF_E354	DMA MSB Count Register for Channel 2
0xFFFF_E358	DMA LSB Control Register for Channel 2
0xFFFF_E35C	DMA MSB Control Register for Channel 2
0xFFFF_E360	DMA LSB Source Address Register for Channel 3
0xFFFF_E364	DMA MSB Source Address Register for Channel 3
0xFFFF_E368	DMA LSB Target Address Register for Channel 3
0xFFFF_E36C	DMA MSB Target Address Register for Channel 3
0xFFFF_E370	DMA LSB Count Register for Channel 3
0xFFFF_E374	DMA MSB Count Register for Channel 3
0xFFFF_E378	DMA LSB Control Register for Channel 3
0xFFFF_E37C	DMA MSB Control Register for Channel 3
0xFFFF_E380	DMA LSB Source Address Register for Link A
0xFFFF_E384	DMA MSB Source Address Register for Link A
0xFFFF_E388	DMA LSB Target Address Register for Link A
0xFFFF_E38C	DMA MSB Target Address Register for Link A
0xFFFF_E390	DMA LSB Count Register for Link A
0xFFFF_E394	DMA MSB Count Register for Link A
0xFFFF_E398	DMA LSB Control Register for Link A
0xFFFF_E39C	DMA MSB Control Register for Link A
0xFFFF_E3A0	DMA LSB Source Address Register for Link B
0xFFFF_E3A4	DMA MSB Source Address Register for Link B
0xFFFF_E3A8	DMA LSB Target Address Register for Link B
0xFFFF_E3AC	DMA MSB Target Address Register for Link B
0xFFFF_E3B0	DMA LSB Count Register for Link B
0xFFFF_E3B4	DMA MSB Count Register for Link B
0xFFFF_E3B8	DMA LSB Control Register for Link B
0xFFFF_E3BC	DMA MSB Control Register for Link B

Table 11.2 Internal Channel DMA Controller Register Address Map

Notes

Physical Address	Description
0xFFFF_E3C0	DMA LSB Source Address Register for Link C
0xFFFF_E3C4	DMA MSB Source Address Register for Link C
0xFFFF_E3C8	DMA LSB Target Address Register for Link C
0xFFFF_E3CC	DMA MSB Target Address Register for Link C
0xFFFF_E3D0	DMA LSB Count Register for Link C
0xFFFF_E3D4	DMA MSB Count Register for Link C
0xFFFF_E3D8	DMA LSB Control Register for Link C
0xFFFF_E3DC	DMA MSB Control Register for Link C
0xFFFF_E3E0	DMA LSB Source Address Register for Link D
0xFFFF_E3E4	DMA MSB Source Address Register for Link D
0xFFFF_E3E8	DMA LSB Target Address Register for Link D
0xFFFF_E3EC	DMA MSB Target Address Register for Link D
0xFFFF_E3F0	DMA LSB Count Register for Link D
0xFFFF_E3F4	DMA MSB Count Register for Link D
0xFFFF_E3F8	DMA LSB Control Register for Link D
0xFFFF_E3FC	DMA MSB Control Register for Link D

Table 11.2 Internal Channel DMA Controller Register Address Map

DMA LSB Source Address Register for Channel 0..3 (‘DmaLSBSourceAddrReg(0..3)’)

DMA LSB Source Address Register for Link A..D (‘DmaLSBSourceAddrReg(A..D)’)

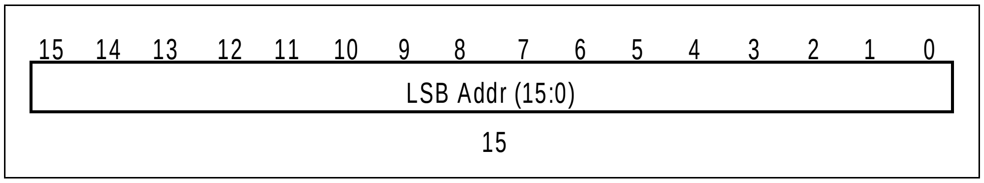


Figure 11.4 Internal DMA LSB Source Address Register (‘DmaLSBSourceAddrReg’).

DMA MSB Source Address Register for Channel 0..3 (‘DmaMSBSourceAddrReg(0..3)’)

DMA MSB Source Address Register for Link A..D (‘DmaMSBSourceAddrReg(A..D)’)

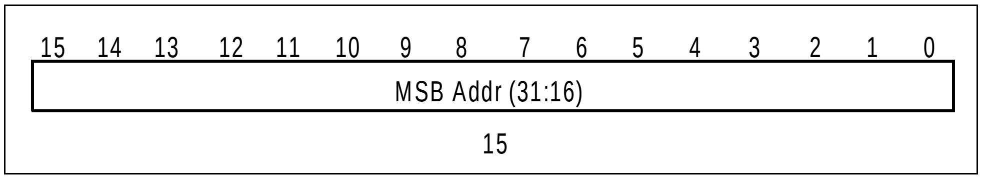


Figure 11.5 Internal DMA MSB Source Address Register (‘DmaMSBSourceAddrReg’).

The Source Address Register, shown in Figure 11.5, must be programmed with the initial address of the peripheral or memory that data is to be read from. The channel Source Address Register will be incremented by the Source Burst Size amount after each DMA transaction. Normally the Source Address Register is only written; however, it may also be read for diagnostic reasons.

Notes

DMA LSB Target Address Register for Channel 0..3 (‘DmaLSBTargetAddrReg(0..3)’)

DMA LSB Target Address Register for Link A..D (‘DmaLSBTargetAddrReg(A..D)’)

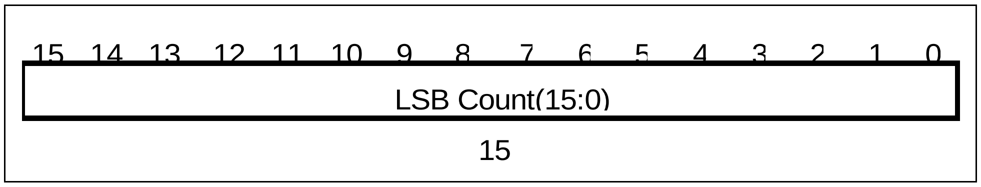


Figure 11.6 Internal DMA LSB Target Address Register (‘DmaLSBTargetAddrReg’).

DMA MSB Target Address Register for Channel 0..3 (‘DmaMSBTargetAddrReg(0..3)’)

DMA MSB Target Address Register for Link A..D (‘DmaMSBTargetAddrReg(A..D)’)

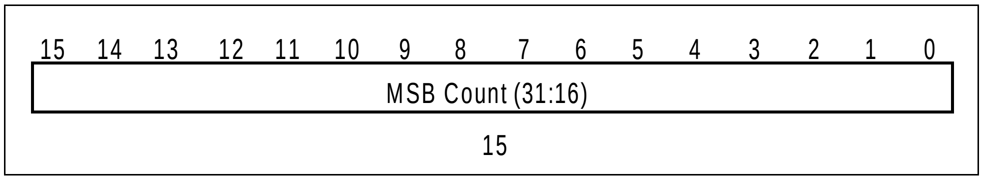


Figure 11.7 Internal DMA MSB Target Address Register (‘DmaMSBTargetAddrReg’).

The Target Address Register (shown in Figure 11.6 and Figure 11.7) must be programmed with the initial address of the peripheral or memory that data is to be written to. The channel Target Address Register will be incremented by the Target Burst Size amount after each DMA transaction. Normally the Target Address Register is only written, however, it may also be read for diagnostic reasons.

DMA LSB Count Register for Channel 0..3 (‘DmaLSBCountReg(0..3)’)

DMA LSB Count Register for Link A..D (‘DmaLSBCountReg(A..D)’)

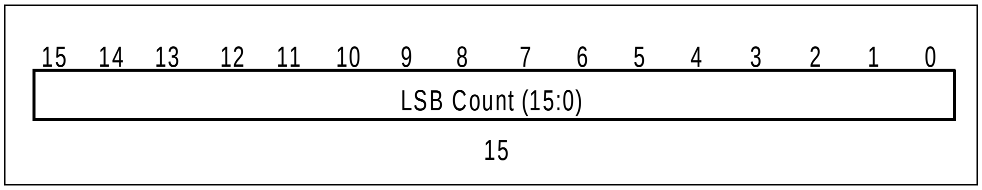


Figure 11.8 DMA LSB Count Register (‘DmaLSBCountReg’).

Notes

DMA MSB Count Register for Channel 0..3 (‘DmaMSBCountReg(0..3)’)

DMA MSB Count Register for Link A..D (‘DmaMSBCountReg(A..D)’)

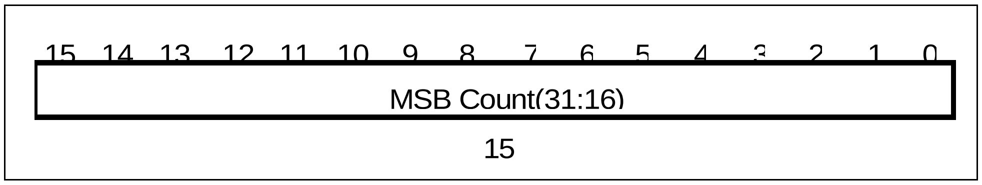


Figure 11.9 Internal DMA MSB Count Register (‘DmaMSBCountReg’).

Note: The count is the number of read/write transactions to be done. A burst or mini-burst read/write only counts once. The channel count register is decremented by 1 after each DMA transaction. If the count is 0, then the DMA Channel will immediately proceed to the next link. The LSB and MSB registers are writable. They are also readable for diagnostic purposes.

The InternalDMA LSB Control register is shown in Figure 11.10, with bit assignments listed in Table 11.3.

DMA LSB Control Register for Channel 0..3 (‘DmaLSBControlReg(0..3)’)

DMA LSB Control Register for Link A..D (‘DmaLSBControlReg(A..D)’)

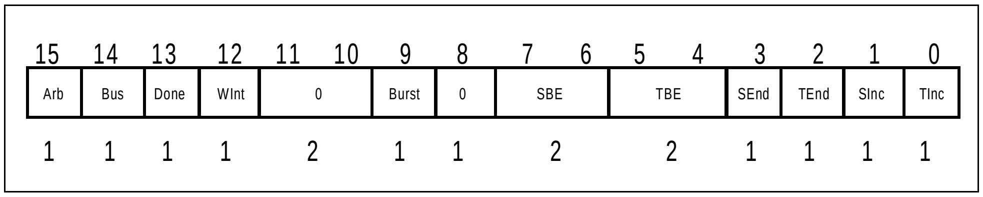


Figure 11.10 Internal DMA LSB Control Register (‘DmaLSBControlReg’).

Bit	Assignment
15	Arbitration Type
14	Keep Bus
13	Allow DMADone

Table 11.3 Internal DMA LSB Control Register (‘DmaLSBControlReg’) Bit Assignments.

Notes

Bit	Assignment
12	Wait for Interrupt
11:10	‘0’
9	Burst Type
8	‘0’
7:6	Source Byte Enable Type, Access Type(1:0)
5:4	Target Byte Enable Type, Access Type(1:0)
3	Source Endianess
2	Target Endianess
1	Increment Source
0	Increment Target

Table 11.3 Internal DMA LSB Control Register (‘DmaLSBControlReg’) Bit Assignments.

Arbitration Type (‘Arb’) Field (Bit 15):

The Arbitration Type is only applicable to Channel 0; however, all channels must be programmed to the same value. Table 11.4 lists the programming values for implementing the two arbitration types available in the RC36100.

Value	Action
‘1’	Fixed Priority Arbitration (default)
‘0’	Rotating Priority Arbitration

Table 11.4 Arbitration Type (‘Arb’) Field Encoding

Keep Bus (‘Bus’) Field (Bit 14):

Value	Action
‘1’	Keep Bus until done with current link.
‘0’	Release Bus at the end of each read/write.

Table 11.5 Keep Bus (‘Bus’) Field Encoding

Allow DMADone (‘Done’) Field Bit (13):

Note: $\overline{\text{DMADone}}$ behaves as a “stop-link” indicator; if $\overline{\text{DMADone}}$ gets asserted during a channel’s DMA $\overline{\text{BusGnt}}$ asserted period, then it will abort the DMA channel and its link after this current transaction completes. Field values and descriptions are listed in Table 11.6.

Value	Action
‘1’	“Stop-link”
‘0’	Normal.

Table 11.6 Allow DMADone (‘Done’) Field Encoding.

Wait for Interrupt (‘WInt’) Field (Bit 12):

The DMA Controller cannot autonomously acknowledge the source of the interrupt. Thus, it is expected that interrupts either pulse low for 1 clock or self reset when the pertinent data port is read or written.

Notes

When programmed in the wait mode, the DMA Controller does not sample for the interrupt until the clock after the internal bus grant is released. This gives the external I/O device being read or written to adequate time to reset its internal interrupt generator. Refer to Table 11.7 for the programming values of bit 12.

Value	Action
'1'	Wait after each transfer until the next interrupt
'0'	Continuous Style.

Table 11.7 Wait for Interrupt ('Int') Field Encoding

Burst Type ('Burst') Field (Bit 9):

If the DMA transaction is to be a burst, the Burst Type Field must be set, and the Burst Size Field in the DMA MSB Control register must also be programmed. Programming values for bit 9 are listed in Table 11.8.

Value	Action
'1'	Transaction is a burst of more than 1 word.
'0'	Transaction is of 1 word or less.

Table 11.8 Burst Type ('Burst') Field Encoding.

Source Byte Enable Type ('SBE') Field (Bit7:6):

The selected type is combined with Addr(1:0) and Endianness to form the byte-enables. Refer to Table 11.9 for Source Byte Enable field values and descriptions.

Value	Selected Type
'11'	Word.
'10'	Reserved.
'01'	Halfword.
'00'	Byte.

Table 11.9 Source Byte Enable Type ('SBE') Field Encoding.

Target Byte Enable Type ('TBE') Field (Bit 5:4):

The selected type is combined with Addr(1:0) and Endianness (see Table 11.11 for Source BigEndianness type field values and descriptions or Table 11.12 for Target Endianness values and descriptions) to form the byte-enables. Refer to Table 11.10 for Target Byte field values and descriptions.

Value	Selected Type
'11'	Word.
'10'	Reserved.
'01'	Halfword.
'00'	Byte.

Table 11.10 Target Byte Enable Type ('TBE') Field Encoding

Notes

Source Endianness Type (‘SEndian’) Field (Bit 3):

Value	Endianness
‘1’	Big Endian.
‘0’	Little Endian.

Table 11.11 Source Big Endianess Type (‘SEndian’) Field Encoding

Target Endianness Type (‘TEndian’) Field (Bit 2):

Value	Endianness
‘1’	Big Endian.
‘0’	Little Endian.

Table 11.12 Target Big Endianess Type (‘TEndian’) Field Encoding

Increment Source Address (‘SInc’) Field (Bit 1):

Value	Action
‘1’	Increment source address
‘0’	Constant source address

Table 11.13 Increment Source Address (‘HInc’) Field Encoding

Increment Target Address (‘TInc’) Field (Bit 0):

Programming information for the Increment Source and Increment Target fields is listed in Table 11.13 and Table 11.14.

Value	Action
‘1’	Increment target address
‘0’	Constant target address

Table 11.14 Increment Target Address (‘TInc’) Field Encoding

DMA MSB Control Register for Channel 0..3 (‘DmaMSBControlReg(0..3)’)

DMA MSB Control Register for Link A..D (‘DmaMSBControlReg(A..D)’)

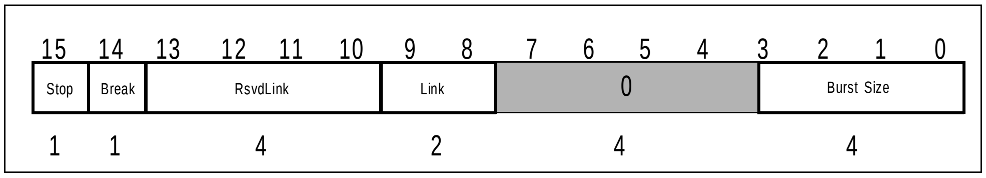


Figure 11.11 Internal DMA MSB Control Register (‘DmaMSBControlReg’).

The Internal DMA MSB Control register fields are shown in Figure 11.11, with bit assignments listed in Table 11.15. Field values and descriptions are listed in Table 11.16, Table 11.17, Table 11.18, and Table 11.19 and Table 11.20.

Notes

Bit #	Field Name
15	Stop
14	Break
13:10	Reserved Link
9:8	Link
7:4	Reserved
3:0	Burst Size

Table 11.15 Internal DMA MSB Control Register ('DmaMSBControlReg') Bit Assignments

Stop ('Stop') Field (Bit 15):

Value	Action
'1'	Stop Immediately and abort any DMA link in progress (default). Any DMA bus transaction may complete.
'0'	Enable DMA.

Table 11.16 Stop ('Stop') Field Encodin

Break ('Break') Field (Bit 14):

Value	Action
'1'	Break at the end of this DMA chain (default). All Reserved Link and all Link Bits must also be set to 1.
'0'	Execute next link at the end of this DMA chain.

Table 11.17 Break ('Break') Field Encoding

ReservedLink Field ('RsvdLink') (Bit 13:10):

Value	Action
'1111'	Must be written with the same value as the Break field. Undefined during reads.
'0000'	Must be written with the same value as the Break field. Undefined during reads.

Table 11.18 Reserved Link ('RsvdLink') Field Encoding

Link ('Link') Field (Bit9:8):

Value	Action
'3'	Load LinkD at the end of this DMA chain and execute (default).
'2'	Load LinkC at the end of this DMA chain and execute.
'1'	Load LinkB at the end of this DMA chain and execute.
'0'	Load LinkA at the end of this DMA chain and execute.

Table 11.19 Link ('Link') Field Encoding

Notes

Burst Size ('BurstSize') Field (Bit 3:0)

This field is only used if the Burst type 'Burst' Field is set.

Value	Action
'12'	Reserved
'8'	Reserved
'4'	Reserved
'0'	4 word burst

Table 11.20 Burst Size ('BurstSize') Field Encoding

External DMA Controller Registers

Table 11.21 is an address map of the External DMA Controller registers. Note that Big Endian software must offset these addresses by b'10 (0x2), if a halfword access is used. The External DMA Control register 0...1 fields are shown in Figure 11.12, bit assignments are listed in Table 11.22. Field values and descriptions are given in Table 11.23, Table 11.24, and Table 11.25.

Timing Diagrams for External DMA Single Datum Read using the Memory Controller (Figure 11.13 on page 17), External DMA Single Datum Write using Memory Controller (Figure 11.14 on page 18), External DMA Two-Datum Burst Read using the Memory Controller (Figure 11.15 on page 18), and External DMA Two-Datum Burst Write using the Memory Controller (Figure 11.16 on page 19) are also included in this section.

Phys. Addr	Description
0xFFFF_E400	ExtDMA Control Register 0
0xFFFF_E410	ExtDMA Control Register 1

Table 11.21 External DMA Controller Register Address Assignments

External DMA Control Register 0..1 ('ExtDmaControlReg(0..1)')

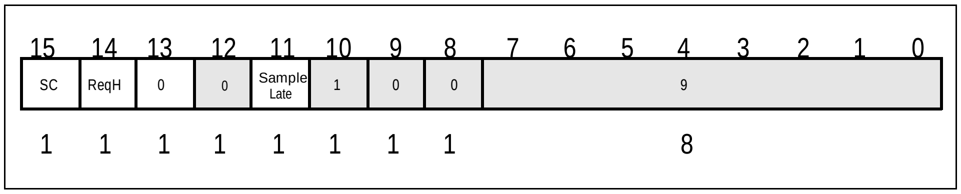


Figure 11.12 External DMA Control Register ('ExtDmaControlReg')

Notes

Bit	Assignment
15	Stop Channel
14	DMABusReq() active High
13	Reserved
12	Reserved
11	Sample $\overline{\text{MemWrEn(3:0)}}$ and $\overline{\text{SysBurst-}}$ Frame one clock later
10	Reserved to ‘1’
9:0	Reserved to ‘0’

Table 11.22 External DMA Control Register (‘ExtDmaControlReg’) Bit Assignments

Stop Channel (‘EC’) Field (Bit 15):

Value	Action
‘1’	Stop/Disable External DMA Channel (default for Channel 1)
‘0’	Enable External DMA Channel (default for Channel 0).

Table 11.23 Enable Channel (‘EC’) Field Encoding

Bus Request Protocol High (‘ReqH’) Field (Bit 14):

.

Value	Action
‘1’	Active High Protocol; DMABusReq is active high for this channel (SCSI controller convention).
‘0’	Active Low Protocol; $\overline{\text{DMABusReq}}$ is active low for this channel (default).

Table 11.24 Bus Request Protocol High (‘ReqH’) Field Encoding

Sample $\overline{\text{MemWrEn}}$ and $\overline{\text{SysBurstFrame}}$ 1 Clock Later (‘SampleLate’) Field (Bit 11):

This field selects whether to sample $\overline{\text{MemWrEn(3:0)}}$ and $\overline{\text{SysBurstFrame}}$ on writes with $\overline{\text{SysALEn}}$ or one clock later as would be done for PCI accesses.

Value	Action
‘1’	Sample $\overline{\text{MemWrEn}}$ and $\overline{\text{SysBurstFrame}}$ one clock after $\overline{\text{SysALEn}}$ first asserts.
‘0’	Sample $\overline{\text{MemWrEn}}$ and $\overline{\text{SysBurstFrame}}$ with $\overline{\text{SysALEn}}$.

Table 11.25 Sample $\overline{\text{MemWrEn}}$ and $\overline{\text{SysBurstFrame}}$ 1 clock later (‘SampleLate’) Field Encoding

External DMA Transactions

External DMA transactions have two major phases:

- ◆ *The external agent command phase*
  - *External DMA agent obtains the bus and drives a read or write command into the RC36100.*
- ◆ *RC36100 execution phase*
  - *The RC36100 takes the received command and actually executes it on the Sysbus.*

Notes

On bursts, External DMA uses the DMADone input to terminate the burst. Bursts may have a block aligned length maximum of 64 words. DMADone must be asserted such that it is sampled by the RC36100 up to the clock that the last datum begins.

In general, where a zero wait-state is the border case, for a word access type during a read operation, DMADone should be asserted one clock after the 2nd to the last SysDataRdy is asserted

Note: For a half word access type DMADone should be asserted one clock after the third to the last SysDataRdy is asserted. For a byte access type, one clock after the fifth to the last SysDataRdy is asserted.

External DMA Operation Timing Diagrams

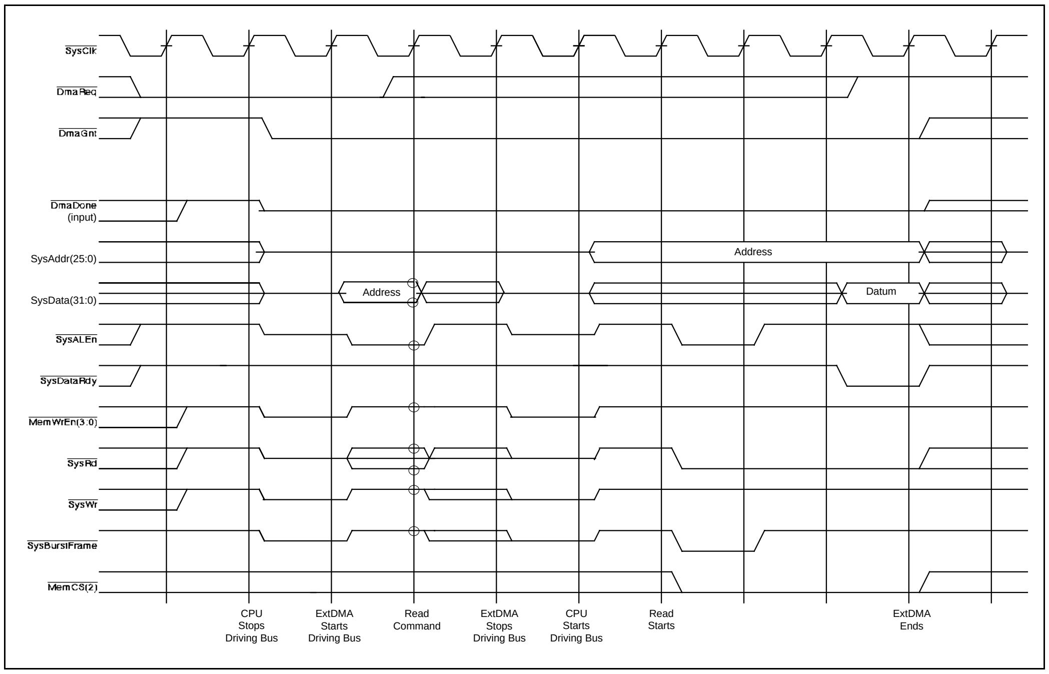


Figure 11.13 External DMA Single Data Read using the Memory Controller (Data Transfer from Memory to Device)

Notes

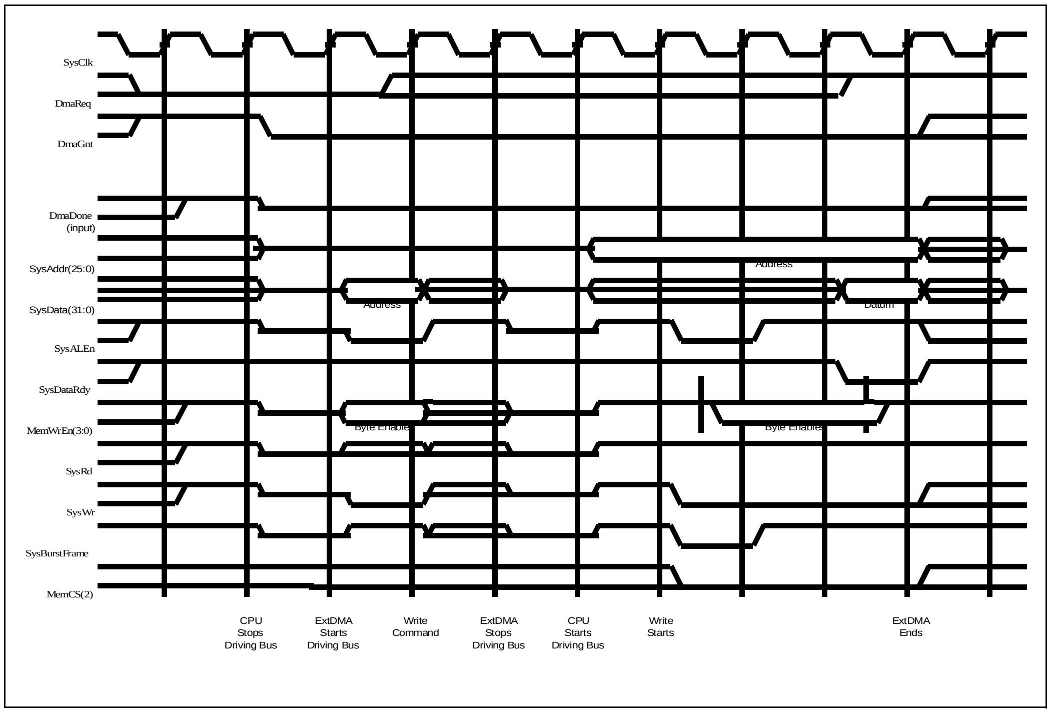
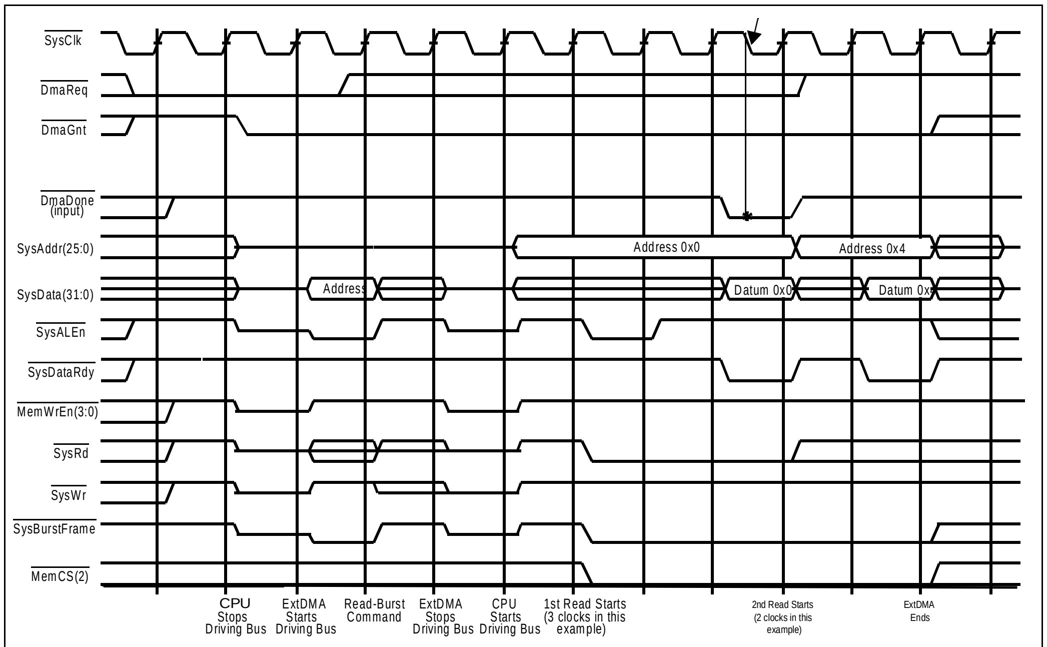


Figure 11.14 External DMA Single Data Write using the Memory Controller (Data Transfer from Device to Memory



***Note that DMADone is sampled on the falling edge of Sysclk.**

Figure 11.15 External DMA Two-Data Burst Read using the Memory Controller (Data Transfer from Memory to Device)

Notes

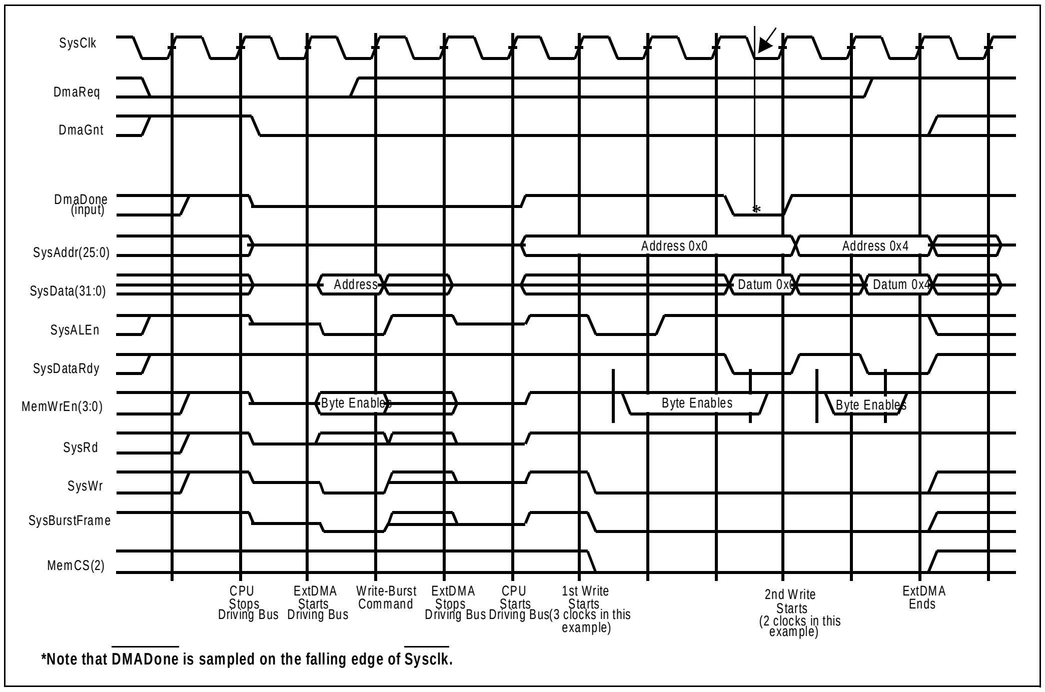


Figure 11.16 External DMA Two-Data Burst Write using the Memory Controller (Data Transfer from Device to Memory)

System Examples

Memory-to-Memory Copying

One common system operation may be a memory-to-memory transfer. For example, the system may transfer from one block of DRAM starting at virtual address 0x80030000 to another block of DRAM starting at virtual address 0x80038000. The block is 1520 bytes long. Only one DMA Channel is needed.

One of the DMA Channels can be programmed so that the source address register is 0x00030000 (the equivalent physical address for virtual address 0x80030000) and target address register 0x00038000. The count register is 1520 / 16 = 95 since the maximum DMA burst transaction is 16 bytes (the size of the DMA read buffer). In the LSB Control Register, the DMA Channel is set up to Burst, Increment Source address, and Increment Target address. The MSB Control Register is set up to Break after all bytes have been transferred, to have a Burst length for 16 bytes, and to Enable the DMA Channel to begin. The CPU can then continue with another process the read while the memory-to-memory transfer takes place.

Transfers between I/O and Memory

Transfer operations between I/O and memory might be desirable when copying data between the UART channels and memory. For example, the serial port may be used in full-duplex mode, and, as such, the serial port would then be simultaneously receiving and transmitting data. In this scenario, two DMA Channels are used: one for receiving and one for transmitting.

Using the Interrupt Controller (for details, see the section titled “Select Interrupt ‘SelInt()’ Field” in Chapter 13), certain interrupts can be steered to certain DMA Channels. For instance, the Serial Receive Interrupt can be steered to DMA Channel 1 and the Transmit Interrupt can be steered to DMA Channel 0.

Notes

In the receiving case, the DMA Channel is programmed to transfer data from the serial port to a DRAM buffer block. When the buffer becomes full, the CPU can act on a DMA Channel 1 Done Interrupt and/or set up another DRAM buffer block via the link registers. The DMA Channel is programmed to not Increase the Source Address for the serial port and to Increase the Target Address after each byte transfer. The DMA Channel is set to transfer 1 byte per count (for the burst length) and to Wait for Interrupt between bytes. After each byte is transferred, the DMA Channel will wait for another receive interrupt before transferring another byte from the serial port to DRAM.

In the transmit case, the DMA Channel is programmed to transfer the message length number of bytes from DRAM which Increases the Source Address after each byte to the Serial Port which does not Increase the Target Address after each byte. The DMA Channel is set to transfer 1 byte per count (as the burst length) and to Wait for Interrupt between bytes. (Alternatively, since the serial port's transmit FIFO is two bytes, DMA could transfer two bytes at a time). After each byte is transferred, the DMA Channel will wait for another transmit FIFO empty interrupt before transferring another byte from DRAM to the serial port.

By using the two DMA Channels, the CPU is freed from either having to constantly poll for serial port status or from constantly handling interrupts and interrupt service routines.

Distinguishing Between CPU and Internal DMA Accesses

If the external system needs to distinguish between a bus transaction generated by the CPU core versus one of the internal DMA channels, there are a couple of options:

- *The system software can code an internal MSB address for the internal DMA channel, and have that address bit be ignored by the address map. The assertion of this address would then signal a DMA transfer.*
- *Use the $\overline{\text{DiagInternalDma}}$ pin.*

Internal DMA Channel Chaining

If a continual chain between a DMA channel and a Link register descriptor is to be used, as soon as the DMA channel is enabled, a new set of instructions can be loaded into the Link register descriptor. However, because a DMA Done interrupt only occurs when the channel is completely stopped, the interrupt service routine should, in its critical section, restart the DMA channel, and in its non-critical section, it should reload the next link.

Note that because DMA is no longer occurring—up until the critical section—the serviced peripheral must either be stopped or contain a sufficiently sized FIFO, as is the case with the Serial Ports.

Parallel Input/ Output (PIO)

Notes

Introduction

The IDT79RC36100 RISController integrates bus controllers and peripherals around the RISCore3000 family CPU core. The on-chip peripherals include ***Parallel Input/Output (PIO) Pins*** (see Figure 12.1 for block diagram) as described in this chapter.

This chapter provides an overview of the PIO programming interface, a complete description of the signal pins, and discusses how PIOs relate to typical internal and external systems.

Features

- *42 general purpose PIO parallel input/output pins*
  - *PIO pins multiplexed with controller functions*

Block Diagram

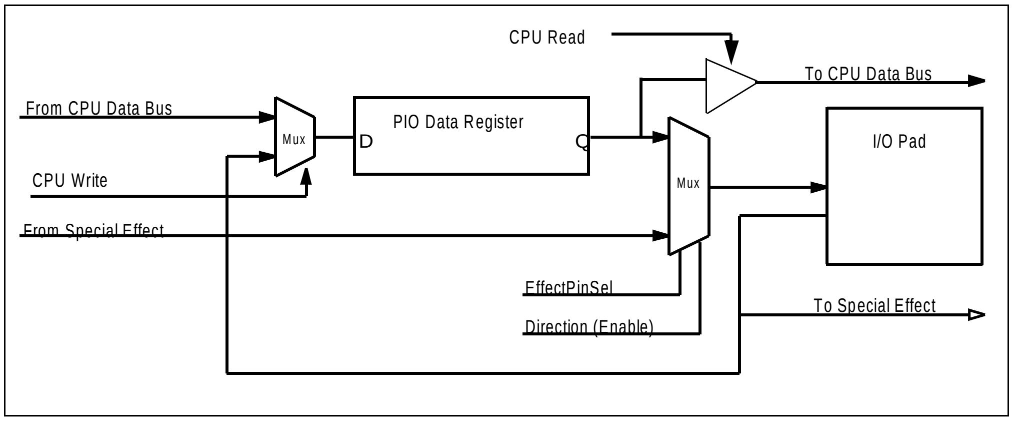


Figure 12.1 PIO Block Diagram.

Overview

The Parallel Input/Output (PIO) pins are programmable multi-purpose pins that act as inputs or outputs. Each PIO pin is also multiplexed with other controller's inputs or outputs. This flexible arrangement allows system designers to customize the RC36100's resources according to their needs. However, applications that do not need the alternate function can use those pins for general purpose inputs or outputs. Inputs are not synchronized beyond the requirements of the destination, unless otherwise noted. Outputs are non-synchronized (typically they are synchronized by the originating peripheral) and are multiplexed.

Pin Descriptions

Most of the PIO pins are multiplexed with other pin functions. Whether or not the internal peripherals are input or output, only the PIO pins can each be programmed as either inputs or outputs.

PIO(n) Input/Output

Parallel Input/Output: These bi-directional signals can be used as generic input/output pins. They are set individually through control registers in the PIO interface and can be read by software reads to the appropriate registers. Table 12.1 shows the relationship between the PIO pins and the other RC36100 function pins that are multiplexed onto the same device pin.

Notes

PIO Number	Alternate Function	Register Number	Bit Position
PIO(32)	SerialCTS(0)	1	14
PIO(31)	SerialDCD(0)	1	13
PIO(30)	SerialRxData(1)	1	12
PIO(29)	SerialPClkIn(1)	1	11
PIO(28)	SerialCTS(1)	1	10
PIO(27)	SerialDCD(1)	1	9
PIO(26)	No alternate function	1	8
PIO(25)	No alternate function	1	7
PIO(24)	No alternate function	1	6
PIO(23)	CentStrobe	1	5
PIO(22)	CentAutoFeed	1	4
PIO(21)	CentInit	1	3
PIO(20)	CentSelectIn	1	2
PIO(19)	DMABusReq(1)	1	1
PIO(18)	ExcInt(4)	1	0
PIO(17)	ExcInt(3)	2	14
PIO(16)	BrCond(3)	2	13
PIO(15)	BrCond(2)	2	12
PIO(41)	SerialRxData(0)	2	11
PIO(40)	SerialPClkIn(0)	2	10
PIO(39)	SerialSClk(0) (Note 1)	2	9
PIO(38)	SerialSync(0) (Note 1)	2	8
PIO(37)	SerialSClk(1) (Note 1)	2	7
PIO(36)	SerialSync(1) (Note 1)	2	6
PIO(35)	TimerTc/Gate(2)	2	5
PIO(34)	TimerTc/Gate(1)	2	4
PIO(33)	TimerTc/Gate(0)	2	3
PIO(14)	SerialTxData(0)	0	14
PIO(13)	SerialRTS(0)	0	13
PIO(12)	SerialDTR(0)	0	12
PIO(11)	SerialTxData(1)	0	11
PIO(10)	SerialRTS(1)	0	10
PIO(9)	SerialDTR(1)	0	9
PIO(8)	No alternate function	0	8
PIO(7)	CentAck	0	7
PIO(6)	CentBusy	0	6
PIO(5)	CentPError	0	5
PIO(4)	CentSelect	0	4
PIO(3)	CentFault	0	3
PIO(2)	CentHostOEn	0	2
PIO(1)	CentHostStrobe	0	1
PIO(0)	DMABusGnt(1)	0	0

[1] This PIO pin must be programmed to be an output before the internal peripheral may be programmed to be an output or else internal signal contention may occur.
[2] The Register-Number and Bit-Position fields describe which of the PIO Data, Direction, and Effect-Select register/bit combinations control that PIO signal.

Table 12.1 Alternate RC36100 functions mapped to PIO pins

Register Definitions

Note: Big Endian software must offset these addresses by b'10 (0x2), if a halfword access is used.

Table 12.2 provides an address map and descriptions of the PIO Registers. Figure 12.2 shows the PIO Data Registers. Additional programming information is located in Table 12.3.

Notes

Phys. Address	Description
0xFFFF_EA00	PIO Data Register 0
0xFFFF_EA04	PIO Direction Control Register 0
0xFFFF_EA08	PIO Effect Select Control Register 0
0xFFFF_EA10	PIO Data Register 1
0xFFFF_EA14	PIO Direction Control Register 1
0xFFFF_EA18	PIO Effect Select Control Register 1
0xFFFF_EA20	PIO Data Register 2
0xFFFF_EA24	PIO Direction Control Register 2
0xFFFF_EA28	PIO Effect Select Control Register 2

Table 12.2 PIO Register Address Assignments.

PIO Data Register 0..2 (‘PioDataReg'0..2)

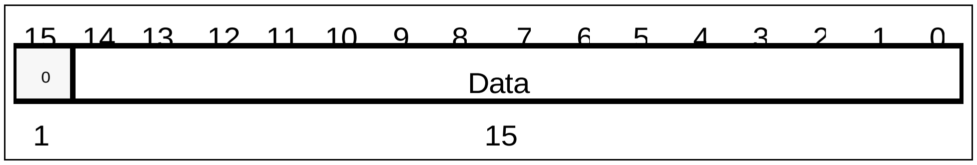


Figure 12.2 PIO Data Register (‘PioDataReg’).

PIO Data (‘PIOData’) Field:

Value	Action
‘1’	PIO pin is high (default).
‘0’	PIO pin is low.

Table 12.3 PIO Data (‘PIOData’) Field Encoding.

PIO Direction Register 0..2 (‘PioDirReg'0..2)

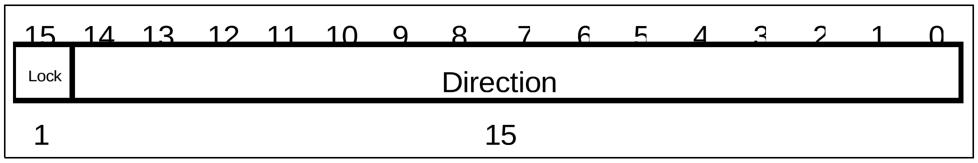


Figure 12.3 PIO Direction Register (‘PioDirReg’).

The PIO Direction Control Registers (see Figure 12.3) contain 16 bits each:

- *The MSB is a lock bit (see Table 12.4).*
- *The bits in PIO Direction Register control whether the PIOs are inputs or outputs (see Table 12.5).*

Notes

Lock ('Lock') Field:

Value	Action
'1'	Locks the Register from being altered by future writes.
'0'	No action (default).

Table 12.4 Lock ('Lock') Field Encoding.

Direction ('Dir') Field:

Value	Action
'1'	PIO pin is an output.
'0'	PIO pin is an input (default).
Note: To avoid internal device damage, this field must be programmed carefully. When used in the input direction, any internal output or I/O driver—for example, a serial port—must first be programmed to be tri-state or an input (default). Only then can the PIO 'Dir' field be safely changed from output to input.	

Table 12.5 Direction ('Dir') Field Encoding.

PIO Effect Select Register 0..2 ('PioEffectSelReg'0..2)

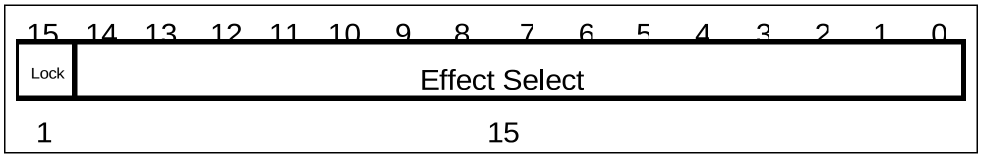


Figure 12.4 PIO Effect Select Register ('PioEffectSelReg').

The PIO Effect Select Registers (see Figure 12.4) contain 16 bits each and are defined as follows:

- *The MSB is a lock bit (see Table 12.6).*
- *The bits in the PIO Effect Select Control Register 1 control whether the PIOs function as a special effect pin (for example, Serial Port) or as a general purpose PIO pin (see Table 12.7).*

Lock ('Lock') Field:

Value	Action
'1'	Lock the Register from future writes.
'0'	No action (default).

Table 12.6 Lock ('Lock') Field Encoding.

Notes

Effect Select (‘EffectSel’) Field:

Value	Action
‘1’	PIO pin is a special effect pin (default).
‘0’	PIO pin is a general purpose pin.

Table 12.7 Effect Select (‘EffectSel’) Field Encoding.

Notes

Peripheral Expansion Interrupt Controller

Notes

Introduction

The IDT79RC36100 RISController integrates bus controllers and peripherals around the RISCore3000 family CPU core. Many of the on-chip peripherals use and produce interrupts. Thus, the RC36100 includes an *Expansion Interrupt Controller,* which is described in this chapter.

The Expansion Interrupt Controller, see Figure 13.1, works in conjunction with the CP0 Status and Cause Registers. The Expansion Interrupt Controller steers the 20+ peripheral interrupts into the last of the six CP0 interrupt lines, Interrupt #5.

A second function of the Expansion Interrupt Controller is to provide interrupt steering and selection to each channel of the DMA controller. By providing access to the interrupts, Datacom peripherals can use the DMA channels to transfer data autonomously.

An overview of the Expansion Interrupt programming interface and a complete description of the signal pins follow. Also included in this chapter is an explanation on how expansion interrupts relate to typical internal and external systems.

Features

- *Allows masking and status checking of all peripherally generated interrupts.*
- *Allows each DMA Controller Channel to receive an interrupt.*

Block Diagrams

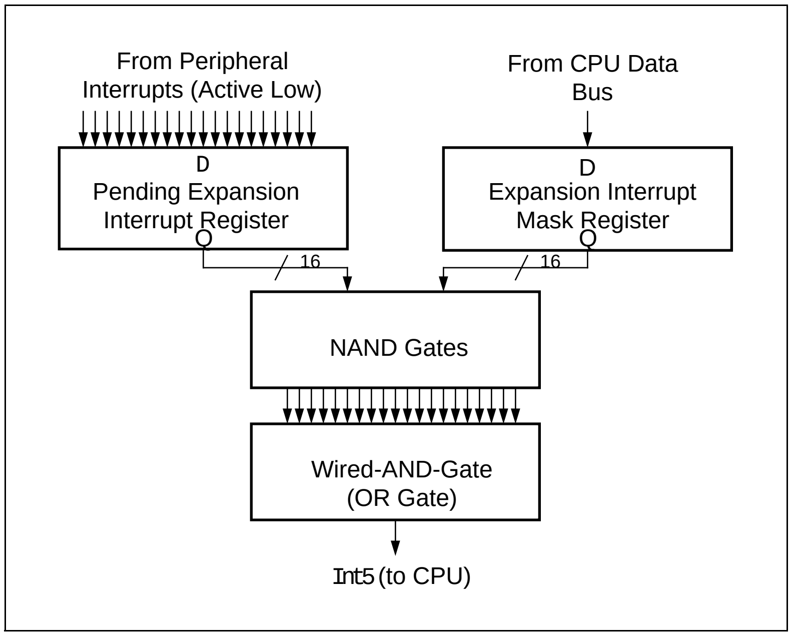

From Peripheral Interrupts (Active Low)
From CPU Data Bus
D
Pending Expansion Interrupt Register
Q
D
Expansion Interrupt Mask Register
Q
16
16
NAND Gates
Wired-AND-Gate (OR Gate)
Int5 (to CPU)

Figure 13.1 Expansion Interrupt Controller (to CPU Interrupt).

Notes

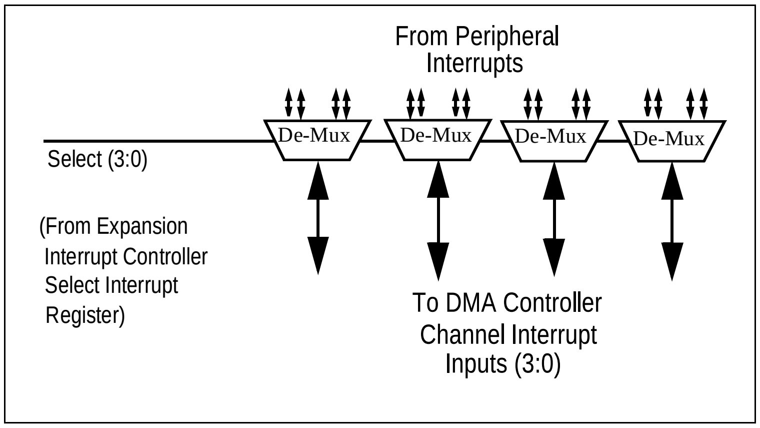

From Peripheral Interrupts
De-Mux
De-Mux
De-Mux
De-Mux
Select (3:0)
(From Expansion Interrupt Controller Select Interrupt Register)
To DMA Controller Channel Interrupt Inputs (3:0)

Figure 13.2 Expansion Interrupt Controller: Steering Interrupts to DMA Requests.

Overview

The Peripheral Expansion Interrupt Controller provides a means of steering the various 20+ peripheral generated interrupts (Figure 13.2). These peripheral interrupts are combined into a single CPU interrupt, Int(5). Each of the peripheral interrupts are stored (active high) in the Pending Expansion Interrupt Register on every system clock. If the corresponding mask bit in the Expansion Interrupt Mask Register is also set (active high), then the overall interrupt line Int(5) is set. At that point it is up to the CPU and ISR software to enable and handle Int(5).

The Peripheral Expansion Interrupt Controller also provides a means of steering a number of the peripheral interrupts to the DMA Channels. Each channel can select from a list of 4 peripheral interrupts. Table 13.1 provides an address map and description of the Expansion Interrupt Controller Register.

Pin Descriptions

Exception Signals

ExcSInt(2:0),
ExcInt(4:3) **Input**

Processor Exception Synchronized Interrupt: These signals are functionally the same as the Int(4:0) signals of the RISCore32 series. The synchronized interrupt inputs are internally synchronized by the RC36100, and therefore may be generated by an asynchronous interrupt agent; the direct interrupts must be externally synchronized by the interrupt agent.

ExcSBrCond(3:2) **Input**

Exception Synchronized Branch Condition Input: These input ports to the processor can use the Branch on Co-Processor Condition instructions to test their polarity. The branch condition inputs are synchronized by the RC36100; therefore, they may be driven by an asynchronous source.

Phys. Address	Description
0xFFFF_EB00	Pend 0
0xFFFF_EB04	Mask 0
0xFFFF_EB10	Pend 1
0xFFFF_EB14	Mask 1
0xFFFF_EB80	DMA Select

Table 13.1 Expansion Interrupt Controller Register Address Assignments

Notes

Expansion Interrupt Mask Register 0..1 ('ExpIntMaskReg0..1)'),

and

Expansion Interrupt Pending Register 0..1 ('ExpIntPendReg0..1)'),

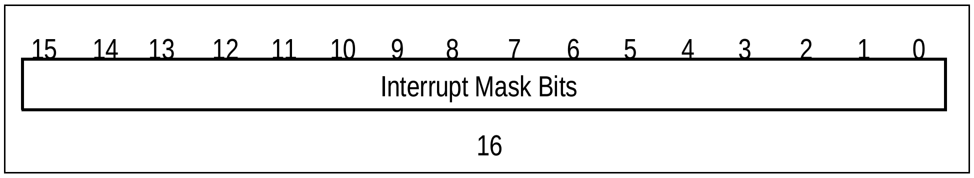


Figure 13.3 Expansion Interrupt Mask Register ('ExpIntMaskReg').

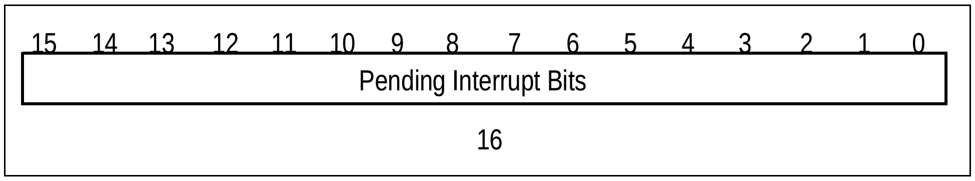


Figure 13.4 Expansion Interrupt Pending Register ('ExpIntPendReg').

A write to the Interrupt Pending Register (Figure 13.4) resets the register bit de-asserted low if the write data bit is a 1, or leaves the register bit with its present value if the write data bit is a 0. The pending interrupt register samples on every clock and holds an interrupt assertion until it is acknowledged by a write to the pending interrupt register. Note that the Expansion Interrupt Pending Register is different than the CPU CP0 Pending Interrupt Field in that the Expansion Interrupt Pending Register has the added feature holding a pulsed (edge-driven) interrupt until acknowledged. The Expansion Interrupt Mask Register is shown in Figure 13.3.

Additional programming instructions for the Expansion Interrupt Mask and Expansion Interrupts Pending registers are located in Table 13.2, Table 13.3, Table 13.4, and Table 13.5.

Reserved Low ('0') Field:

Must be written to '0' for future compatibility. Value when read is undefined.

Mask Bits ('Mask') Field:

Bit	Description
'4'	SerialInt
'3'	SerialRx_Req(1)
'2'	SerialRx_Req(0)
'1'	SerialTx_Req(1)
'0'	SerialTx_Req(0)
Note:	Default values for the Pending and Mask Registers are all '0'.

Table 13.2 Expansion Interrupt Mask Register 1 and Expansion Interrupt Pending Register 1 Bit Assignments.

Notes

Pending Bits ('Pend') Field:

Bit	Description
15	Reserved
14	CentReadInt
13	CentWriteInt
12	CentResetInt
11:8	Reserved
7	TimerTC(2)
6	TimerTC(1)
5	TimerTC(0)
4	Reserved
3	DMADoneInterrupt3
2	DMADoneInterrupt2
1	DMADoneInterrupt1
0	DMADoneInterrupt0
Note:	Default values for the Mask and Pending Registers are all '0'.

Table 13.3 Expansion Interrupt Mask Register 0 and Expansion Interrupt Pending Register 0 Bit Assignments.

Value	Description
'1'	Interrupt pending.
'0'	Interrupt not pending.

Table 13.4 Pending Interrupt Field Encoding.

Value	Description
'1'	Interrupt enabled/allowed.
'0'	Interrupt disabled/disallowed.

Table 13.5 Interrupt Mask Field Encoding.

Expansion Interrupt DMA Select Register ('ExpIntDMASelReg')

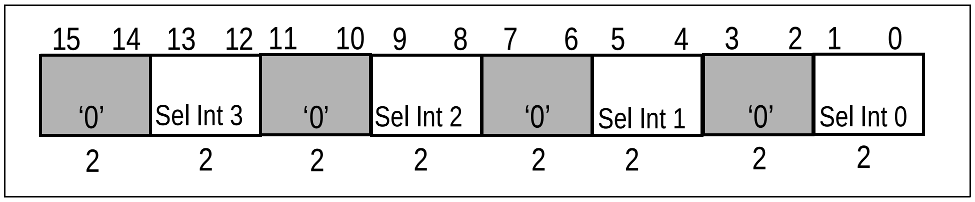


Figure 13.5 Expansion Interrupt DMA Select Register ('ExpIntDMASelReg').

Notes

Select Interrupt ‘SelInt()’ Field:

Figure 13.5 gives the fields for the Expansion Interrupt DMA Select Register. The Select Interrupt field does a 1-in-4 select on the inputs to particular Interrupt Pending Register fields. The resulting input is passed to the Internal DMA Controllers. The input to the Interrupt Pending Register is used so that the DMA Controller can bypass and does not need to acknowledge the Interrupt Pending Register by resetting it (such as ignoring it).

It is implied that the peripheral device receiving or transmitting the data will automatically de-assert its interrupt line either by using a (one clock minimum) pulse/edge or by de-asserting, when it receives a data strobe. Additional programming instructions for this register are located in Table 13.6 and Table 13.7.

Note: Some interrupts are intentionally duplicated in multiple channels so that the system programmer can choose the relative priority of the interrupts.

Sel	IDMA(3)	IDMA(2)	IDMA(1)	IDMA(0)
3	CentWriteInt	CentReadInt	SerialTxReq(0)	Reserved
2	SerialRx_Req(1)	SerialTx_Req(1)	SerialRx_Req(0)	SerialTx_Req(0)
1	TimerTc(2)	CentWriteInt	TimerTC(1)	TimerTC(0)
0	ExcInt(3)	ExcInt(2)	ExcInt(1)	ExcInt(0)

Table 13.6 DMA Channel versus Interrupt De-Multiplexer.

Value	Action
‘15’-’4'	Reserved.
‘3’	Select Interrupt 3.
‘2’	Select Interrupt 2.
‘1’	Select Interrupt 1.
‘0’	Select Interrupt 0.

Table 13.7 Select Interrupt (‘SelInt()’) Field Encoding.

Notes

Timers

Notes

Introduction

The IDT79RC36100 RISController integrates bus controllers and peripherals around the RISCore3000 family CPU core. The on-chip peripherals include three ***Timers***.

This chapter will provide an overview of the Timer programming interface, a complete description of the signal pins, and a discussion on how the timers relate to typical internal and external systems. A block diagram of the RC36100's Timers is located in Figure 14.1.

Features

- *3 16-bit Timer Channels, with global 16-bit prescaler*
- *3 $\overline{\text{TC}}/\overline{\text{Gate}}$ pins*
- *Each Timer has*
  - *16-bit count register with selectable 16-bit frequency divider/prescaler of pipeline clock input*
  - *16-bit compare register*
  - *TC control bit allowing auto reset vs. compare register write ack for interrupts*
  - *Gate option control bit (gate option allows PWM counting or time stamping)*
- *Default Timer use is for a Real Time Clock/Timer.*
- *Timers have bus time-out control bit*
  - *Reset on bus start, gate on bus cycles*
- *Timer0 has 16-bit PWM low time compare register*
  - *Square Wave Generator*

Block Diagram

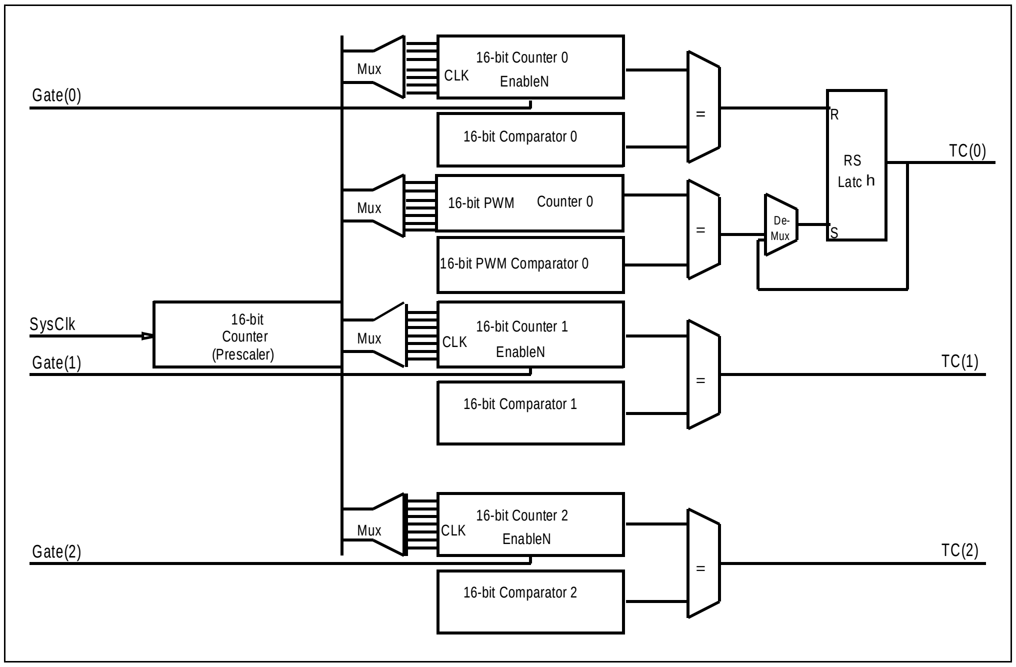


Figure 14.1 Block Diagram of the RC36100 Timers.

Notes

Overview

The RC36100 contains 3 timers. Each timer consists of a 16-bit count register as well as a 16-bit compare register. The count register resets to 0 and counts upward until it equals the compare register. When the count register equals the compare register, the $\overline{TC}$ output is asserted and the count register is reset back to 0.

To expand the amount of time the timers can handle, each timer uses a common 16-bit prescaler counter and can be programmed to select a power-of-2 divisor of the prescaler as its fundamental base frequency for clock ticks. The prescaler counter itself is based off of the System Clock, $\overline{SysClk}$.

Using the default mode, each timer can be used as a real-time counter. Special effects include:

- *Counter*
- *Real-time interrupt-based timer*
- *Bus time-out timer*
- *Gated clock external event counter*

In addition to the above effects, timer channel 0 also has a PWM (Pulse Width Modulation, i.e., controllable duty cycle) compare register that controls the number of counter ticks that the timer output, $\overline{TC}$, remains asserted low, allowing for use as a PWM generator.

Timer0 has a programmable PWM feature that determines the number of low prescaled clocks per timer period. For Timer 0, the Count Compare Register determines the overall number of prescaled clocks for the total rollover period as Compare +1 prescaled clocks. For the condition where CountCompare is greater than PWMCompare, Timer0's count algorithm is as listed in Table 14.1.

TCn Value	Prescaled Clocks
high + low	CountCompare + 1
low	PWMCompare +1
Note: CountCompare must be greater than PWMCompare.	

Table 14.1 Timer0 Count Algorithm

The border case for Timer 0—where CountCompare is less than or equal to PWMCompare—does not produce useful behavior. Therefore, to produce an all high or all low Timer period, the Timer TCn output must be overridden by placing its corresponding PIO pin into its general purpose data mode and programming its data to either a "1" or "0".

Timer1 and Timer2 do not have a programmable PWM feature, so their PWM value should be considered as fixed at "1". The count algorithm for these timers is listed in Table 14.2.

TCn Value	Prescaled Clocks
high + low	Compare + 1
low	2

Table 14.2 Timer1 and Timer2 Count Algorithm

Thus, for Timers 1 and 2, the output TCn will be high for Compare-1 prescaled clocks and low for 2 prescaled clocks. The total rollover period is Compare+1 prescaled clocks.

Note: The Timer functions are based off of the prescaler clock; in particular, the Timer Interrupt Acknowledge mechanism where the Compare Register is written and should generally be used instead of using the regular timer mode.
If the prescaled clock is 2**10, in the regular mode, TCn could be low for 1 or 2 prescaled clocks, which could be about 1024 clocks before it occurs, after which the Interrupt Pending Register must be cleared. Thus, most real-time applications will want to either keep the prescaled clock at a very low value, such as 2**0 or 2**1, to get a better response resolution or use the Timer Interrupt Acknowledge mechanism.

Notes

Pin Descriptions

Timer Peripheral Signals

The Timer $\overline{\text{TC}}$ output signal and the Gate input signal are driven and sampled on the falling edge of SysClk. Under this condition, the user must expect or meet setup and hold times relative to the falling edge of the clock. Furthermore, the $\overline{\text{TC}}$ output signal is relatively slow, especially with a non divide-by-1 prescale clock divisor, transitioning very late in the clock cycle.

TC(2:0), Input/Output

TimerGate(2:0)

Gate: Active low input mode where the corresponding Timer() can increment its count whenever Gate() is asserted low. The Timer() stops counting whenever Gate() is high. Typically, the Timer() prescaler is set to the divide-by-1 mode so that each Gated clock corresponds to a Timer tick. Applications include Time Stamping, Pulse Width Measurement, and Timer expansion.

Terminal Count: Active low output mode is where the corresponding Timer() asserts $\overline{\text{TC}}$ low whenever its Count Register equals its Compare Register. There is a one clock delay from Count equalling Compare until $\overline{\text{TC}}$ is seen on the external pin, due to internal synchronization. Normally $\overline{\text{TC}}$ asserts low then immediately de-asserts back high after 2 timer clock cycles. Thus, the rising edge of $\overline{\text{TC}}$ will be seen 1 clock after the counter value programmed.

If the 'AckOnWrCompare' bit option in the Timer Control Register is asserted, then $\overline{\text{TC}}$ de-asserts back to high only when the Timer Compare Register is written. In this mode, the Timer can be used to generate an interrupt, and then the interrupt handler can acknowledge the interrupt.

Timer 0 has a Pulse Width Modulation (PWM) Compare Register that can be programmed to de-assert $\overline{\text{TC}}$ back high after a user-programmed number of clock cycles. Applications include using the Timer to strobe external events, generating Real-time interrupts, Timer Expansion, and PWM control.

Note: Timers 1 and 2 have the low count hard coded to 1 instead of 0. Timer 0 features a programmable PWM Compare Register that programs the number of prescaled clocks TC is held asserted.

For timers 1 and 2, the assertion of $\overline{\text{TC}}$ is held for two prescaled clocks, and the user should program the counter to 1 less than specified. If $\overline{\text{TC}}$ is synchronized for gating into an external counter/function, then $\overline{\text{TC}}$ must take the 2.0 prescaled clocks into account.

Register Descriptions

Table 14.3 provides a Timer Register Physical Address Map. Note that Big Endian software must offset these addresses by b'10 (0x2), if halfword accesses are used.

Address	Description
0xFFFF_E900	Timer Prescaler Count Register
0xFFFF_E904	Timer PWM Count Register 0
0xFFFF_E910	Timer Count Register 0
0xFFFF_E914	Timer Compare Register 0
0xFFFF_E918	Timer PWM Compare Register 0
0xFFFF_E91C	Timer Control Register 0
0xFFFF_E920	Timer Count Register 1
0xFFFF_E924	Timer Compare Register 1
0xFFFF_E92C	Timer Control Register 1
0xFFFF_E930	Timer Count Register 2
0xFFFF_E934	Timer Compare Register 2
0xFFFF_E93C	Timer Control Register 2

Table 14.3 Timer Register Physical Address Map

Notes

Timer Prescaler Count Register ('TimerPrescalerCountReg')

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Count															
16															

Figure 14.2 Timer Prescaler Count Register ('TimerPrescalerCountReg').

The prescaler counter starts at reset, and continuously counts in an upward direction, and wraps around on overflow. The reset default value is 0x0000. The prescaler counter uses the System Clock, SysClk, as its fundamental base clock frequency. The Timer Prescaler Count Register is illustrated in Figure 14.2. This is a write-only register.

Timer Count Register 0..2, Timer PWM Count Register

('TimerCountReg'0..2, TimerPWMCount Reg'0)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Count															
16															

Figure 14.3 Timer Count Register ('TimerCountReg').

The Count Register does not count if the PIOIsInputGate is enabled via the Timer Control Register and the input gate signal, $\overline{\text{Gate}}$, is high. The Count Register is 16-bit Readable and Writable, if LockCountAndCompare control bit is not active. The Count Register ignores the $\overline{\text{Gate}}$ input, if the $\overline{\text{Gate}}$ input control bit is not turned on.

The PWM Count Register is similar to the other count register, except that its function is to determine the number of clock cycles $\overline{\text{TC}}$ is to be held asserted (low).

Timer Compare Register 0..2 ('TimerCompareReg'0..2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Compare															
16															

Figure 14.4 Timer Compare Register ('TimerCompareReg').

The Compare Register, illustrated in Figure 14.4, is a 16-bit register containing the value that will reset the counter when the Count Register is equal to it. The default value at reset for the Compare Register is 0xFFFF. This register is 16-bit readable and writable, if the 'LockCountAndCompare' control bit is not active. If written and 'WriteCompareAck' Timer Control Register bit is active, then it brings $\overline{\text{TC}}$ back to inactive high.

Notes

Timer Pulse Width Modulation Register 0 (‘TimerPWMReg0’)

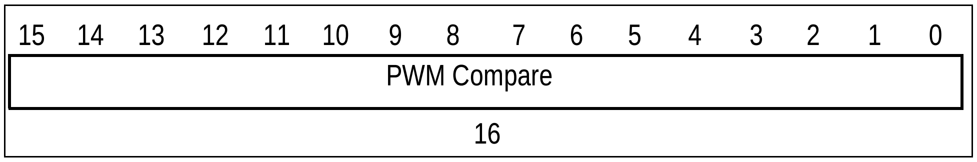


Figure 14.5 Timer Pulse Width Modulation Register (‘TimerPWMReg’).

The Timer Pulse Width Modulation Register, illustrated in Figure 14.5, has a 16-bit value which brings PWM $\overline{TC}$ output back high ‘N’ prescaled clocks after $\overline{TC}$ goes low. Thus, this register is analogous to the Timer Compare Register, but instead controls the length of $\overline{TC}$ assertion. The reset default value is 0x0000.

By programming various values for the compare and the PWM compare registers, the duty cycle of the $\overline{TC}$ output can be varied. For instance, by using a compare value equal to a PWM-1 value, the duty cycle will be 50/50. For the $\overline{TC}$ output to remain low, the $\overline{TC}$ PIO pin must be programmed as a general purpose output with a value of 1. Additional programming information for this register is located in Table 14.4.

Value	Action
0	high after 1 clock
1	high after 2 clocks
etc.	etc.

Table 14.4 Timer Pulse Width Modulation Register (‘TimerPWMReg’) Bit Fields.

Timer Control Register 0..2 (‘TimerControlReg'0..2)

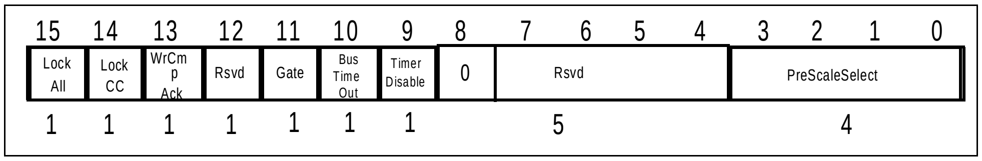


Figure 14.6 Timer Control Register (‘TimerControlReg’).

Figure 14.6 illustrates the field names and bit assignment breakdown of the Timer Control Register. Additional programming information is located in Table 14.5, Table 14.6, Table 14.7, Table 14.8, Table 14.9, Table 14.10, Table 14.11, and Table 14.12.

Notes

Bit	Description
15	Lock.
14	LockCountAndCompare.
13	AutoAck vs WriteCompareAck.
12	Reserved, Must be written as ‘1’. **Note:** Default is ‘0’
11	PIO is Input $\overline{\text{Gate}}$.
10	BusTimeout.
9	TimerDisable.
8:4	Reserved. Must be written as '0'. Because an undefined value will occur for these bits, reads should mask them.
3:0	Prescale 1 of 16 Select.

Table 14.5 Timer Control Register (‘TimerControlReg’) Bit Assignments.

Lock (‘Lock’) Field (Bit 15):

Value	Action
‘1’	Control locked from future writes.
‘0’	No action (default).

Table 14.6 Lock (‘Lock’) Field Encoding.

Lock Count and Compare (‘LockCC’) Field (Bit 14):

Value	Action
‘1’	Count and Compare Registers locked from future writes.
‘0’	No action (default).

Table 14.7 Lock Count and Compare (‘LockCC’) Field Encoding.

Write Compare Ack (‘Ack’) Field (Bit13):

The Ack field is intended for timer driven interrupts. When the interrupt occurs, the service routine writes to the Timer Compare Register, which brings $\overline{\text{TC}}$ back high. The corresponding Pending Interrupt Register bit can then be cleared. Note that in the ‘Ack’ mode, the timer continues to count when $\overline{\text{TC}}$ is asserted low.

Value	Action
‘1’	WriteCompareAck requires a write to the compare register to set $\overline{\text{TC}}$ back high.
‘0’	$\overline{\text{TC}}$ back high 1 clock after $\overline{\text{TC}}$ asserts low. (default)

Table 14.8 Write Compare Ack (‘Ack’) Field Encoding.

PIO is Input Gate (‘Gate’) Field (Bit 11):

The PIO is Input Gate field must be programmed identically to the actual PIO pin.

Notes

Value	Action
‘1’	If PIO pin is programmed elsewhere as an input, then $\overline{TC}$ is driven in from that PIO input.
‘0’	$\overline{TC}$ output mode.

Table 14.9 PIO is Input Gate (‘Gate’) Field Encoding.

BusTimeout (‘BTO’) Field (Bit 10):

Value	Action
‘1’	Enable Bus Time-out feature. Reset counter to 0 on the beginning of each external bus cycle.
‘0’	Disable use Bus Time-out feature. (default)

Table 14.10 BusTimeout (‘BTO’) Field Encoding.

Timer Disable (‘TimerDis’) Field (Bit 9):

This field enables the timer to count. **Note:** Also write a ‘1’ to reserved bit 12, to enable the timer.

Value	Action
'1'	Stop
'0'	Timer Enabled (default)

Table 14.11 Timer Enable (‘TimerEn’) Field Encoding

Prescaler Select (‘PSel’) Field (Bit 3:0):

Note that changing the Prescaler Select can take up to 2*16 clocks to take effect, due to internal synchronization. To force the new prescaler select to be updated, write a Ox ffff to the Prescaler Count Register, which will cause it to rollover/update.

Value	Prescaler Divide-By Frequency
‘F’	div 2**15 (divide by 32768)
...	...
‘2’	div 2**2 (divide by 4)
‘1’	div 2**1 (divide by 2)
‘0’	div 2**0 (divide by 1) (default)

Table 14.12 Prescaler Select (‘PSel’) Field Encoding.

Notes

Serial Ports

Notes

Introduction

The IDT RC36100 RISController integrates bus controllers and peripherals around the RISCore3000 series family CPU core. One such peripheral is the Serial Communications Controller, (SCC) described in this chapter.

The SCC is largely compatible with the Advanced Micro Devices (AMD) Am85C30 © or ™ Serial Communications Controller. While this chapter describes the SCC's programming interface for users with serial communications programming experience, readers needing introductory or more detailed information should consult the AMD publication *Am8530H/Am85C30 Serial Communications Controller Technical Manual.*

Differences between the RC36100 SCC and the AMD product are described *(usually in italics)* at various places in the text. Figure 15.1 is a block diagram of the SCC.

Features

- *Two full duplex channels, independent except for shared interrupt structure. Each channel has its own Baud rate generator, Crystal Oscillator, DPLL for clock recovery.*
- *Byte Synchronous and SDLC/HDLC data rate of up to 1/8th of the IDT36100 clock. (ie 6.25 Mhz with a 50 Mhz CPU clock). Note, the PCLK clock referred to in the AMD literature is ½ of the RC36100 CPU clock.*
- *Asynchronous mode support for: 8 bits per character; 1, 1½, and 2 stop bits; Odd or Even Parity; x1 (with external clock), x16, x32, or x64 clock modes; Break generate/ detect; parity error, overrun error, and framing error detect.*
- *Byte Synchronous mode support for: Arbitrary 8, 16, 6, or 12 bit sync patterns; Software transparent CRC generate/detect; Choice of two CRC methods.*
- *SDLC/HDLC mode support for: Software transparent zero bit insert/delete; Software controlled rejection of frames without matching Address field; Software transparent CRC generate/ detect; SDLC Loop Mode.*
- *Buffered data registers: Quadruple for receiver; Double for transmitter.*
- *Special operating modes: NRZ, NRZI, FM0, FM1 encoding and decoding; Manchester decoding. Local Loopback; Auto Echo modes.*
- *Features to ease SDLC/HDLC programming effort: 10 X 19 bit Frame Status FIFO for high speed SDLC/HDLC reception without CPU intervention. Each FIFO entry includes received frame size and status; Automatic SDLC/HDLC Tx Underrun/EOM flag reset; Automatic SDLC/ HDLC CRC generator preset.*

Overview

The SCC contains two full duplex channels. The channels are completely independent, except for the interrupts generated by them (which are prioritized so that they can be combined into a single SCC interrupt).

Each channel can operate in a Bit Synchronous (SDLC, HDLC), or Byte Synchronous (Monosync, Bisync), or Asynchronous mode. In each mode there are options (as listed under *Features* above).

Character I/O from the SCC may be polled, interrupt driven, or use DMA. With DMA, DMA links can be used for uninterrupted reception into non-contiguous memory buffers, avoiding the need for software intervention to handle the buffer switching.

In SDLC, a hardware FIFO records the receive status and frame size for a maximum of ten frames, each of up to 16KB. While the use of DMA frees software from per-character processing, the FIFO frees software from per-frame processing for up to ten consecutive frames.

Notes

In each of the three modes, the SCC provides a choice of external clocking (using either of two clock signal inputs), or internal clocking (using a baud rate generator), a choice of encoding methods (NRZ, NRZI, or FM), and (for NRZI and FM encoding) the option of using a digital phase locked loop (DPLL) for clock recovery.

For byte synchronous (eg monosync, bisync) modes, the SCC has two ways of synchronizing to the input stream. In internal synchronization, the SCC scans the input stream for a synchronizing pattern (a process known as hunting). In external synchronizing, a synchronizing signal is supplied externally. SDLC mode always uses internal synchronization and the pattern is the SDLC “flag”, 0x7E.

A token passing variation of the SDLC protocol, called SDLC loop mode, is also supported. This variation runs on multiple stations connected in a unidirectional loop. One of the stations serves as a controller and passes a token (which is a specific bit pattern called an EOP) around the ring. Each of the other stations transmits on receiving the token, following which it forwards the token to the next station.

The SCC provides for two modem control outputs and three modem control inputs. In “auto enables” operation, two of the inputs act to enable the transmitter and receiver. This implements modem side flow control without software monitoring of modem status.

To aid in diagnostic applications, the SCC supports internal loopback (in which the transmitter is internally connected to the receiver) and autoecho (in which incoming characters are echoed on the transmitter).

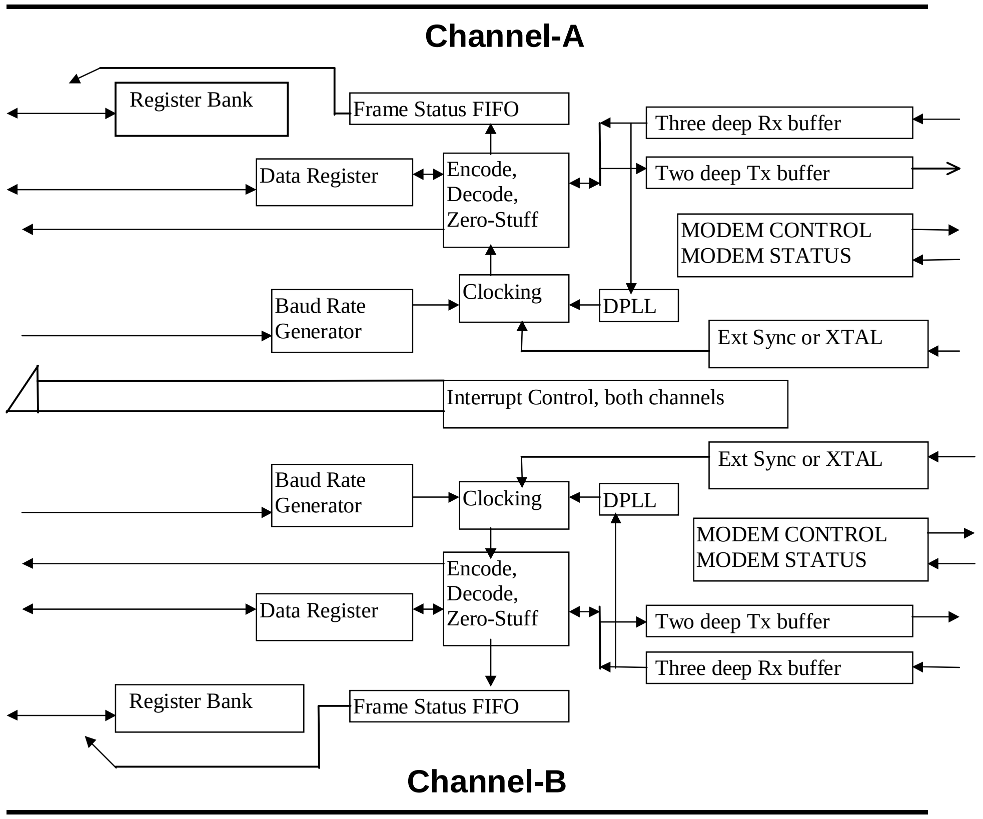


Figure 15.1 Block Diagram of Serial Communication Controller

Figure 15.2 illustrates programmer access to the components of the RC36100 Serial Communications Controller.

Notes

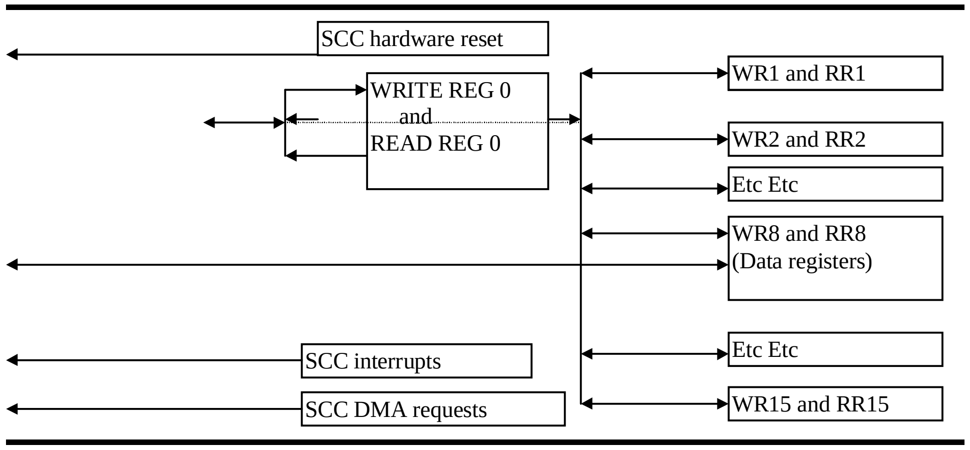


Figure 15.2 Programmer access to the Serial Communications Controller

Registers

The SCC has sixteen software accessible read registers and sixteen software accessible write registers. These are termed RR0 through RR15 and WR0 through WR15. In some cases, the same register acts as a read and a write register so that what is written can be read back.

Two of the read registers (RR6 and RR7) read from the top entry in the SDLC received Frame Status FIFO.

WR0 and RR0 can be accessed directly by writing to and reading from location:

0xFFFF_E800 for Channel 0 (also known as Channel A).

0xFFFF_E804 for Channel 1 (also known as Channel B).

The other 15 registers are accessed using a two step process:

1. Write the register number (such as 11 for RR1 or WR11) to WR0.
2. Read or write RR0 or WR0. This will actually access register 11.

The first write to WR0 sets an internal POINTER to access one of the other registers in place of RR0/WR0. The next access (which is to the OTHER register) resets this pointer so that another access will require another two step process.

The Data Registers (for transmitting and receiving data bytes) can be accessed in two ways: Either through an access to WR8 (transmit) or RR8 (receive), or by a direct read or write from location:

Channel	Location
A	0xFFFF_E80C
B	0xFFFF_E804

Note that the registers are byte sized, but the addresses are word addresses. These addresses should be used for 32 bit reads or writes, with the actual register values in the lowest byte.

Interrupts

The SCC has five interrupt outputs which are available as bits in one of the “interrupt pending” registers of the RC36100 Peripheral Expansion Interrupt Controller. This register is at location:

Address	Description
0xFFFF_EB10	Interrupt outputs

Notes

Bit assignments are shown in Figure 15.1.

Bit	Assignment
0	Set while the Channel A “Transmitter Empty” interrupt is active.
1	Set while the Channel B “Transmitter Empty” interrupt is active.
2	Set while the Channel A “Receiver Ready” interrupt is active.
3	Set while the Channel B “Receiver Ready” interrupt is active.
4	Set while the master SCC interrupt is active.

Table 15.1 Bit Assignment for the interrupt outputs

DMA

For DMA operation, the SCC has one “ready to transmit” and one “ready to receive” DMA enable for each channel. These enables are tied to specific DMA channels as shown below. In addition, each DMA channel has three alternative sources of enables, making it necessary to set a DMA “steering register” to select the SCC enables. This is done as follows:

Address	Description
0xFFFF_EB80	Steering Register

Bits	Setting	Description
1:0	2	DMA-0 is tied to Channel-A, Transmit Ready
5:4	2	DMA-1 is tied to Channel-A, Receive Ready
9:8	2	DMA-2 is tied to Channel-B, Transmit Ready
13:12	2	DMA-3 is tied to Channel-B, Receive Ready

Table 15.2 Steering Register

External connections of the SCC

Externally visible Serial Communications Controller signals are:

- *SerialTxData(1:0) are the transmitter outputs for the two channels. In AMD literature, and elsewhere in this note, this signal is named TxD.*
- *SerialRxData(1:0) are the receiver inputs for the two channels. In AMD literature, and elsewhere in this note, this signal is named RxD.*
- *SerialPrimaryClkIn(1:0) are potential clock source inputs for the two channels. In AMD literature, and elsewhere in this note, this signal is named RTxC.*
- *SerialSecondaryClk(1:0) are present in the two channels and can be programmed as an input or as an output. As an output it can make available any of three internal clocks. As an input it is a potential clock source. In AMD literature and elsewhere in this note, this signal is named TRxC.*
- *SerialSync(1:0) are present in the two channels function as inputs or as outputs depending on the operating mode. In Asynchronous mode, it serves as a general purpose input. In Synchronous mode with External Synchronization, it serves as an input and must be driven low two clocks after the Synchronizing pattern is detected. In SDLC mode, and in Synchronous Mode with Internal Synchronization, it functions as an output, going low when a synchronizing pattern is detected.*

Notes

- *SerialCTS(1:0) are the Clear To Send modem status inputs for the two channels.*
- *Serial RTS(1:0) are the Request To Send modem control outputs for the two channels.*
- *SerialDCD(1:0) are the Data Carrier Detect modem status inputs for the two channels.*
- *SerialDTR(1:0) are the Data Terminal Ready modem control outputs for the two channels.*

SCC Operations

Polled, Interrupt, and DMA operation are the three basic methods of data input and output from the Serial Communications Controller.

Polled Operation

In POLLED operation, software repeatedly checks status bits in Read Register 0 (RR0) to determine when to perform I/O.

Bit 0 is the “Rx Character Available” bit. It is high when a character has been received but not read from the data register. When it is detected high, the received character should be read from the Data Register. An “Overrun Error” results if the data is not read quickly enough, and if the arrival of more data causes the receive data FIFO to overflow.

Bit 2 is the “Tx Buffer Empty” bit. It is high to indicate that the SCC is ready to accommodate the next outgoing character, which should be written into the Data Register. For the actual transmission, this character is then loaded into another register, called the Transmitter Shift Register. As soon as this load occurs, the Data Register can receive another byte, and this bit goes high again. The load into the shift register occurs after the current shift register contents get transmitted. At that time, if the Data Register is still empty, a “Transmit Underrun” occurs. This has no effect in Asynchronous Mode. But in synchronous modes, gaps in outgoing data cannot be allowed so the SCC starts to transmit something else (depending on how it is programmed), either a synchronizing pattern or the CRC of the outgoing frame.

Occurrences of receive events are indicated by certain bits in RR1 being high: Bit 4 (Parity Error), Bit 5 (Overrun Error), Bit 6 (CRC or Framing Error), and Bit 7 (SDLC End of Frame).

Interrupt operation

INTERRUPT operations make use of the SCC’s interrupt structure. Each SCC channel generates three categories of interrupts, Transmit interrupts, Receive interrupts, and External/Status interrupts. A TRANSMIT interrupt occurs when a character is loaded from the outgoing Data Register to the Transmitter Shift Register (as described above), causing RR0 bit 2 to go high. In response, software can write the next character to the Data Register. If there are no more outgoing characters, software must instead issue a “Reset TxINT Pending” by writing 0x28 to WR0. This removes the interrupt until the next time a character moves from the Data to the Shift register.

In interrupt based transmission, as long as there is data to be sent, one byte is written to the Data Register for each interrupt. When there is no more data, the interrupt is reset. When data becomes available again, the first byte is directly written into the Data Register. It is immediately loaded into the Shift Register, causing the next interrupt.

There are two types of RECEIVE interrupts which may occur. An “Rx Character” interrupt occurs each time a character is received into the Data Register. A “Special Rx Condition” interrupt occurs when any of the following conditions arise: a “Receiver Overrun” error; a “Framing Error” in Asynchronous mode; a “Parity Error” (while parity is enabled); an SDLC “EOF” (an End of Frame is detected due to receipt of a flag).

Note: In synchronous modes, a “CRC error” does not, by itself constitute a Special Rx condition. The reason is that a non-zero CRC bit is considered an error only at one point in time - after a frame’s last CRC byte has been received. At this time it must be zero. At all other times, it is expected to be 1. In SDLC, the CRC bit is checked when the EOF - which IS a Special condition - is received].

When an Rx interrupt occurs, the SCC does not explicitly indicate whether it is due to a Character, a Special Condition, or both. The interrupt handler must first read RR1 to pick up the special status, if any. It must also read RR0 (before or after reading RR1), and if RR0:Bit 0 is set, it must read the received byte from the Data Register. *Note that reading the Data Register discards the contents of RR1, which must therefore be read first.*

Notes

The Character interrupt is automatically reset by the Data Register read. But if examination of the RR1 value shows a special condition, then it must be reset by issuing the “Error Reset” command by writing 0x30 to WR0.

An EXTERNAL/STATUS interrupt occurs (when enabled) due to transitions of certain signals. Each signal may be individually enabled to cause an E/S interrupt. The signals (described next) are all available as individual bits in RR0. They are Sync/Hunt (bit 4), Break/Abort (bit 7), Zero Count (bit 1), Tx Underrrun/EOM (bit 6), Clear To Send (bit 5), and Data Carrier Detect (bit 3).

A feature of E/S interrupts is that—for the signals for which interrupts are enabled—latches are present between the signal sources and their values in RR0 bits. These latches close on any transition of any of these signals, and are re-opened when the E/S interrupt gets serviced. For these signals, the RR0 bits reflect the latched values. For the other signals, RR0 has the real-time values. *[A detailed description, including the effects of multiple transitions while the latches are closed, is in the AMD 85C30 manual].*

An E/S interrupt is serviced by reading RR0 (to open the latches) and issuing the command “Reset External/Status interrupts” by writing 0x10 to WR0.

Of the E/S sources:

Sync/Hunt, Data Carrier Detect, and Clear To Send have been described above, in the section on “External Connections to the SCC”. Either transition on these generate an interrupt if enabled. In internal synchronous modes, Sync/Hunt goes high on power on, or when an “Enter Hunt” command is issued (by writing a 1 to bit 4 of WR3). This command forces the SCC receiver to resynchronize on the input data stream. Once synchronization is achieved, Sync/Hunt goes low.

The Zero-Count signal is high while the baud-rate generator (a counter used to generate SCC transmit and receive clocks) passes through its zero value. Only its low to high transition generates an interrupt if enabled.

The Break/Abort signal is high if a Break (a continuous zero value received in Asynchronous mode) or an Abort (a string of seven or more contiguous 1s received in SDLC mode) is detected. Either transition generates an interrupt if enabled.

The Transmit Underrun/EOM *(“EOM” stands for End of Message)* is used to control CRC transmission in Synchronous and SDLC modes. It works as follows:

Primarily, it goes high when the transmitter “underruns”, that is, it has no more data to transmit because none has been written into the Data Register by software. This happens when, at the end of a frame, software responds to the Transmit Empty condition by resetting the transmitter (by issuing a “Reset TxINT Pending” command). *(Note, despite this command’s name, it is issued to reset the transmitter even in non-interrupt operation).* Thereafter, once the last character has been sent, the SCC raises the Underrun / EOM signal. But before it does, that, it checks whether the signal is already high. If it is, it takes no further action.

But it is low (how this happens is described next), then the SCC starts to send the frame’s CRC bytes (and also raises the Underrun / EOM signal). Once the CRC bytes have been sent, another Transmit Empty condition occurs and the software responds by once again resetting the transmitter interrupt.

(The AMD manual mentions that instead of resetting the transmitter the first word of the next frame can be written at this point; however, on the RC36100 this is not recommended. There should be a delay before starting the next frame to allow at least one flag or synchronizing pattern to be sent between frames).

Once the signal is high, it needs to be made low again, if the next frame’s CRC is to be sent. This is done by issuing the “Reset Tx Underrun/EOM latch” command by writing 0xC0 to WR0. In SDLC mode, it can also be done automatically by the SCC. To do this, set high bit 1 in WR7’ (this is a special “SDLC enhancement” register. It exists along with the normal WR7 and is written to by first setting WR15 bit 0 to 1, to select it).

The resetting of this signal must be done after writing the first character of a frame, and before the final “empty” condition arises. (This is because it cannot be reset while the shift register is empty). The most convenient way is to issue it after writing the frame’s first character.

Only the low to high transition of this signal will cause an interrupt if enabled.

Notes

The SCC's Interrupt Structure

In the RC36100, the Tx and Rx interrupts are directly connected to the Peripheral Expansion Interrupt Controller.

Also connected to the Controller is a single common SCC interrupt line, on which the SCC is capable of internally prioritizing multiple pending interrupts as follows:

1. Rx Chl-A
2. Tx Chl-A
3. E/S Chl-A
4. Rx Chl-B
5. Tx Chl-B
6. E/S Chl-B

The SCC offers two approaches to interrupt handling:

- *ACKNOWLEDGED: the SCC handles prioritizing*
- *NON-ACKNOWLEDGED: the SCC does not handle prioritizing*

In ACKNOWLEDGED interrupt handling, the SCC internally prioritizes multiple pending interrupts, presenting them one at a time in RR2 in response to an ACKNOWLEDGE signal. The interrupt selected is the one with the highest priority of those pending. It is termed the IUS (INTERRUPT UNDER SERVICE), and when it has been processed, a "Reset IUS" command must be issued to the SCC (by writing 0x38 to WR0), after which the process is repeated for the next pending interrupt.

The interrupt is presented in RR2 encoded in bits 3:1 as follows:

000: Channel B Tx
001: Channel B E/S
010: Channel B Rx Data
011: Channel B Rx Special Condition
100: Channel A Tx
101: Channel A E/S
110: Channel A Rx Data
111: Channel A Rx Special Condition

In NON-ACKNOWLEDGED interrupt handling, prioritizing is not done, and the driver looks at certain "interrupt pending" bits in RR3 to identify simultaneously pending interrupts, which it handles in a suitable manner and sequence.

The RR3 bits are:

Bit 0: Set if Channel-B E/S interrupt is pending.
Bit 1: Set if Channel-B Tx interrupt is pending.
Bit 2: Set if Channel-B Rx interrupt is pending.
Bit 3: Set if Channel-A E/S interrupt is pending.
Bit 4: Set if Channel-A Tx interrupt is pending.
Bit 5: Set if Channel-A Rx interrupt is pending.

Note: In this mode, Rx Data interrupts and Rx Special Condition interrupts are not separately indicated as they are in Acknowledged mode. Interrupt handler software will have to check RR1 for special conditions and RR0 for data).

It is **important** to note that these bits can be read only in RR3 of Channel A. They do not appear in RR3 of channel B. *(Because SCC interrupt handling must combine both channels, it is an area in which both channels can behave differently).*

Selecting between Acknowledged and Non-Acknowledged modes is done by reading or not reading RR2 on entering the interrupt service routine. If RR2 is read then acknowledged interrupt handling starts. Otherwise it does not.

Just as RR3 is read in channel A, RR2 must be read in channel B. (A read of RR2 in channel will return the value that was last written into WR2).

In Non-Acknowledged mode, RR3 can be read once on entering the interrupt service routine, and all the bits in it can be processed in a single pass. Then it can even be read and processed again before leaving the ISR. This can allow multiple interrupt events to get processed in a single ISR invocation. In this mode, unlike in Acknowledged mode, it is not necessary to explicitly acknowledge the interrupt with a "Reset IUS". Another advantage of using this mode is that software can give a higher priority to any type of event than the SCC would have done.

Notes

(**Alert:** As with other SCC registers, reading RR2 or RR3 is a two step process. Between the two steps, that is while the value in WR0 is either 2 or 3, preparatory to accessing RR2 or RR3, the SCC freezes its entire interrupt structure so that the RR2 and RR3 values are not altered. The sequence should not be prolonged for any reason, as doing may lead to missed interrupts and unstable operation).

Unlike the AMD 85C30, the RC36100 requires certain bits in WR9, the interrupt configuration register, to have fixed values: Bits 0, 1, 2 high, bit 4 low, bit 5 high.

The AMD 85C30 is a product intended for use in diverse environments, and has a plethora of interrupt configuration options. The RC36100 reduces much of this complexity for programmers. Specifically, the RC36100 does not have interrupt acknowledge cycles, and hence no vectored interrupts. It also does not participate in interrupt daisy chaining with external devices. Status bits are always present in bits 3-2-1 of RR3 in channel A.

DMA operation

DMA operation utilizes the fact that, in the RC36100, the "Rx Character Available" and the "Tx Buffer Empty" signals are internally connected to DMA request lines.

(In the AM85C30, certain signals may be used for DMA or for other purposes. In the RC36100, these are always used for DMA. To ensure this, WR1 bit 5, bit 6 and bit 7 and WR14 bit 2 are all set to 1. The effect is to have the DMA connections active. They are also permanently enabled, so that control of DMA must be done through the DMA controller).

Therefore, instead of using polling or interrupts to transfer characters, the DMA channels assigned to the SCC Tx and Rx are enabled to transfer characters.

An issue in using DMA is the handling of receive errors. DMA will not read RR1 to check for receive errors. To cater to this, the SCC provides three modes of receive interrupt operation. These are "interrupt on all characters and on special conditions", "interrupt on first character - of a frame - or on special conditions", and "interrupt on special conditions only".

Of these, the first is unsuitable for DMA, since DMA picks up received characters. The second is relevant when it is necessary for software to reset the Tx Underrun/EOM latch after writing the frame's first character. It is not necessary if done automatically by the SCC in SDLC mode. It is also not needed in Asynchronous mode. In these cases the third option can be used. Note that, while responding to Special Condition interrupts, do not pick up received characters, even if they are ready. That is the DMA's job.

In SDLC, the SCC offers an on-board Frame Status FIFO for the high speed reception of SDLC frames. Up to ten consecutive received frames have their sizes recorded in entries in this FIFO. Also recorded is the occurrence of CRC errors in these frames. *(Unlike in the AMD product, overrun errors are not recorded. These can still be detected since they will lead to CRC errors).*

The FIFO is enabled by setting bit 2 of WR15. To check FIFO status, the registers RR6, RR7, and RR1 must be read, in that order. *(For the AMD 85C30, the sequence is RR7, RR6, RR1).* RR7 Bit 6 is set if the FIFO has one or more entries. RR7 Bit 7 is set if the FIFO has overflowed. If there is an entry, then its frame size is recorded in RR6 (bits 7-0) and RR7 bits 5-0 (bits 13 - 8 of the size. The size is 14 bits, allowing 16K frames) and RR1 will have the CRC and overrun error bits from the FIFO entry, and real-time values for its remaining bits.

When using the FIFO, DMA should be used, as it is pointless to use polled or interrupt I/O with the FIFO doing its job. Rx should be on interrupt, in the "Special Only" mode. (This is because, even though CRC errors are recorded in the FIFO, the CPU still has to issue "Error Reset" commands to clear special conditions.

Each "Error Reset" will clear all the special conditions till that point). If an overrun error occurs, the SCC locks the DMA. Issuing the "Error Reset" will also unlock the DMA. (Software can also discover the overrun condition by reading RR1 in the special condition interrupt, although it is not recorded in the FIFO).

A FIFO overflow or receiving an Abort means the FIFO has to be flushed of all its entries and all received data till that point discarded (as it is no longer known what data belongs to which frame). For this reason, during FIFO operation, the E/S interrupt for BREAK/ABORT should remain enabled, and if it occurs the handler should arrange for the FIFO to be flushed as soon as possible.

Notes

Other SCC Subsystems include data encoding methods and digital phase locked loop commands.

Data Encoding

ENCODING methods supported by the SCC includes Non Return to Zero (NRZ, transitions at bit cell boundaries, levels same as values - zero or one), Non Return to Zero Inverted (NRZI, zero means a transition at bit cell boundary, 1 means no transition), FM0 (transition at each boundary, and if zero then a further transition in mid cell), and FM1 (like FM0, but with mid cell transition if 1). The receiver can also decode Manchester encoding (data level in first half of each bit cell, its complement in second half). Encoding is specified by writing WR10 bits 6-5, with 00 = NRZ, 01 = NRZI, 10 = FM1, 11 = FM0. (*The AMD manual deals with issues related to encoding. Also, encodings other than NRZ may create problems in Asynchronous mode, if the line is left at level zero after sending a byte. In this mode level zero is interpreted as a BREAK).*

Digital Phase Locked Loop (DPLL)

In FM and NRZI modes, a Digital Phase Locked Loop (DPLL) can be used to extract the clock from the data, for more reliable reception. DPLL operation is initiated by issuing the “Enter Search Mode” command (by writing 001 to WR14 bits 7-5). At any time if the DPLL misses one or more expected transitions, it sets the “One Clock Missing” bit (RR10 bit 7) or the “Two Clocks Missing bit” (RR10, bit 6).

Recovery from this condition is by issuing a “Reset Missing Clocks” command (by writing 010 to WR14 bits 7-6). The command “Disable DPLL” (011 to WR14 bits 7-6) disables the DPLL, and another “Enter Search Mode” command must be given to restart it. For other DPLL commands see the description of WR14 bits 7-6 in the SCC Registers section below.

External Connections

AUTOENABLES, selected by setting bit 5 of the WR3, makes the CTS input the transmitter enable and the DCD input the receiver enable. Both are active high.

LOCAL LOOPBACK mode (selected by setting WR14 bit 4) internally connects TxD to RxD so that all transmitted data also reaches the receiver. AUTO ECHO mode (selected by setting WR14 bit 3) TxD is only connected to RxD internally (and not to the outside). RxD also remains connected to the outside.

SDLC Loop Mode

(SDLC LOOP MODE is described in detail in the AMD manual).

Clocking Options

The BAUDRATE GENERATOR (BRG) is a 16 bit counter, generating a clock transition each time it reaches zero. The countdown value is written into WR13 (high byte) and WR12 (low byte). *(For applications where the BRG may be started and stopped, the AMD manual explains how to do this glitchlessly).*

The BRG is enabled by setting WR14 bit 0. The BRG’s clock source is chosen by WR14 bit 1. This bit is set to drive the BRG from the system clock, PCLK. On the RC36100, PCLK is half the value of the externally attached crystal. (eg 25 Mhz if the crystal is 50 Mhz). If this bit is reset, the BRG clock is taken from the RTxC input, ie the SerialPrimaryClkIn pin.

The DPLL’s CLOCK may be the BRG output or the RTxC input, selected by setting WR14 bits 7-5 to 100 for BRG and 101 for RTxC.

The TRANSMIT CLOCK source and RECEIVE CLOCK source are selected by WR11 bits 4-3 (transmit) and bits 6-5 (receive). Values for these bits are 00 to select RTxC, 01 for TRxC (ie the SerialSecondaryClk pin), 10 for the BRG output, and 11 to be driven by the DPLL’s output.

The TRxC pin becomes an input if WR11 bit 2 is 0. Otherwise it is an output and WR11 bits 1-0 determine whether it carries the Transmit Clock (01), BRG output (10), or the DPLL output (11). *(The remaining value, 00, is used in the AMD product to make it carry the output of an externally attached crystal. The RC36100 does not support an external crsytal).*

Notes

In addition to these, there are many lesser features and enables, which are covered in the AMD manual. To know whether any such feature is supported in the RC36100, look at the corresponding register in the SCC registers section below. This will indicate what bit values are accepted for each register.

SCC Operating Sequence

Reset: A hardware reset is accomplished by a read from location 0xFFFF_E810. This resets both SCC channels. Writing 11 to WR9 bits 7-6 also does a hardware reset. Individual channels are reset by writing to the same bits, 10 to reset channel A, and 01 to reset channel B. (Unlike most other registers, for which each channel has a copy, only one copy of WR9 exists, shared by both channels. Therefore a WR9 write can be done in either channel). Following power on, it is recommended that four hardware resets be done for proper SCC initialization. A reset on either channel does not affect ongoing operations on the other channel.

Parameter setting: Following a reset, the transmitter and receiver are both disabled. Before enabling them, all operating parameters must be specified by writing values into registers.

The steps in starting up the SCC are listed next. (Exact register values can be obtained from the "SCC registers" section).

- *At the outset, the Tx (Transmit), Rx (Receive), Digital Phase Locked Loop (DPLL), and Baud Rate Generator (BRG) must all be disabled to prevent output glitches while parameters are being set. (Since all the registers cannot be written to simultaneously, there will be times when different register hold mutually incompatible values). Of these, all are disabled by a reset, except the BRG, which is disabled only by a hardware reset, and not by a channel reset.*
- *Interrupts and DMA should also be disabled through WR1.*
- *Next a "Null" command should be issued by writing 0 to WR0.*
- *WR4 is used to specify the basic mode (SDLC, Synch, Asynch). Therefore it must be the first to be set, as values written into other registers are often interpreted according to the mode.*
- *WR10, which sets encoding (as well as miscellaneous SDLC settings) is set next. One SDLC setting, called "Mark or Flag on idel" should be set to "Mark" (bit 3 is 1).*
- *WR6 and WR7 carry the synchronizing pattern in SDLC and Synchronous modes. These are to be set next. (Note, WR7 is distinct from the Enhanced SDLC register, WR7'. If WR15 bit 0 is 0 - which it is after a reset - then WR7 is accessed. Otherwise WR7' is.*
- *Receive parameters are now written to WR3, followed by transmit parameters to WR5, but neither should be enabled as yet.*
- *A dummy "interrupt vector" value is written into WR2. (It can be any value. The RC36100 SCC does not use vectored interrupts).*
- *Clocking parameters are now written into WR11.*
- *The baud rate divisor is written into WR12 and WR13.*
- *Issue two DPLL commands to WR14 (using bits 7-5) setting the DPLL clock source and the DPLL mode (FM or NRZI). While issuing these commands, keep bit 0 low to keep the BRG disabled.*
- *Write to WR15 keeping bit 0 high to make WR7' accessible. (The other bits are as per the External/Status interrupt enables desired, and to enable the SDLC Frame Status FIFO if needed. Enabling E/S interrupts is harmless since the master interrupt enable, WR9 bit 3, is low after a reset, and till it is set interrupts cannot occur).*
- *Write desired value to WR7'.*
- *Write to WR14 to enable DPLL and BRG. (Only enable if they are to be used).*
- *Enable Rx through WR3, then Tx through WR5.*
- *If SDLC mode, then change the "SDLC Mark on idle" setting in WR10 to "Flag on idle" (bit 3 set to 0).*
- *Issue the "Reset Tx CRC command" to WR0.*
- *Issue the "Reset E/S int" command to WR0. Repeat this once.*

Notes

- *Enable interrupts and DMA through WR1.*
- *The last step is to enable the Master Interrupt Enable (Set WR9 bit 3 to 1).*

I/O

In Asynchronous Mode, there is no special I/O sequence (other than the transmit and receive logic described earlier).

In Synchronous and SDLC modes, it is also necessary to reset the transmit underrun latch after each frame's first byte, and to reset Tx and Rx CRC logic at frame starts.

In SDLC, the SCC can be programmed to do all of this automatically, by setting various bits in WR7'. (See the "SCC Registers" section). In Synchronous mode, or if WR7' bits are not used, it is necessary to do all of this in software. Consult the AMD manual.

SCC Registers

The following drawings illustrate in detail the SCC write registers and the associated bit configuration for each.

Note: These abbreviations are used throughout the following register drawings: Sn=SYNC bit n; An=Address bit n; MS=Monosync; BS=Bisync; SD=SDLC

Write Register 0

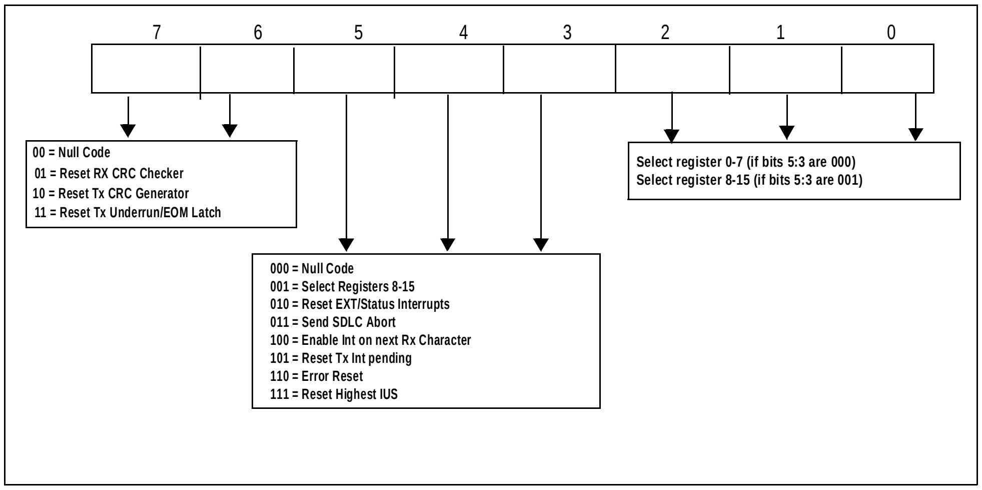

7
6
5
4
3
2
1
0
00 = Null Code
01 = Reset RX CRC Checker
10 = Reset Tx CRC Generator
11 = Reset Tx Underrun/EOM Latch
000 = Null Code
001 = Select Registers 8-15
010 = Reset EXT/Status Interrupts
011 = Send SDLC Abort
100 = Enable Int on next Rx Character
101 = Reset Tx Int pending
110 = Error Reset
111 = Reset Highest IUS
Select register 0-7 (if bits 5:3 are 000)
Select register 8-15 (if bits 5:3 are 001)

Figure 15.3 Write Register 0 (WR0) Bit Values and Field Descriptions

Write Register 1

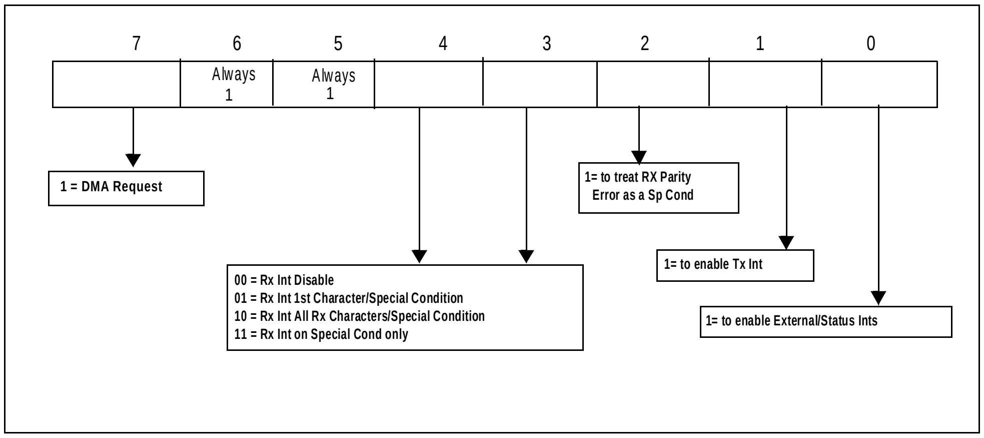

7
6
5
4
3
2
1
0
Always 1
Always 1
1 = DMA Request
1= to treat RX Parity Error as a Sp Cond
1= to enable Tx Int
1= to enable External/Status Ints
00 = Rx Int Disable
01 = Rx Int 1st Character/Special Condition
10 = Rx Int All Rx Characters/Special Condition
11 = Rx Int on Special Cond only

Figure 15.4 Write Register 1 (WR1) Bit Values and Configurations

Notes

Write Register 2

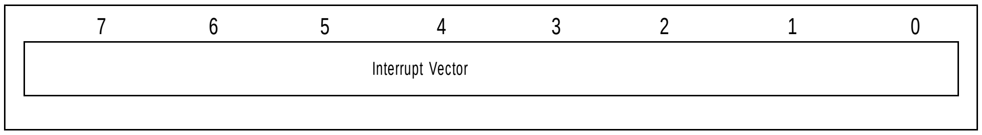


Figure 15.5 Write Register 2 (WR2)

Write Register 3

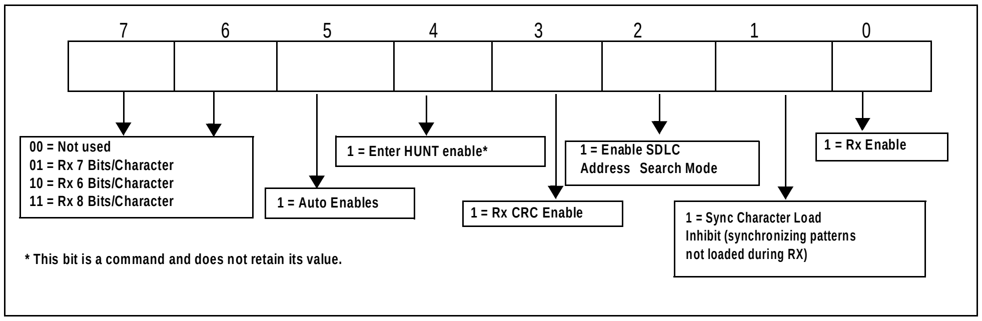

7
6
5
4
3
2
1
0
00 = Not used
01 = Rx 7 Bits/Character
10 = Rx 6 Bits/Character
11 = Rx 8 Bits/Character
1 = Auto Enables
1 = Enter HUNT enable*
1 = Rx CRC Enable
1 = Enable SDLC Address Search Mode
1 = Sync Character Load Inhibit (synchronizing patterns not loaded during RX)
1 = Rx Enable
* This bit is a command and does not retain its value.

Figure 15.6 Write Register 3 (WR3) Bit Values and Configurations

Write Register 4

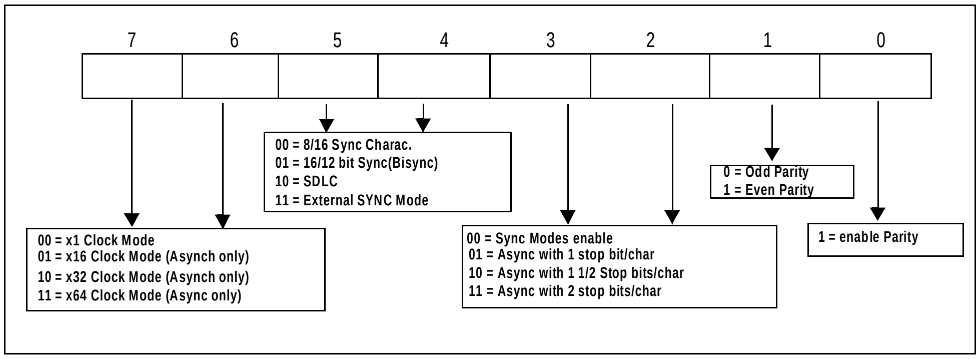

7
6
5
4
3
2
1
0
00 = x1 Clock Mode
01 = x16 Clock Mode (Asynch only)
10 = x32 Clock Mode (Asynch only)
11 = x64 Clock Mode (Async only)
00 = 8/16 Sync Charac.
01 = 16/12 bit Sync(Bisync)
10 = SDLC
11 = External SYNC Mode
00 = Sync Modes enable
01 = Async with 1 stop bit/char
10 = Async with 1 1/2 Stop bits/char
11 = Async with 2 stop bits/char
0 = Odd Parity
1 = Even Parity
1 = enable Parity

Figure 15.7 Write Register 4 (WR4) Bit Values and Configurations

Write Register 5

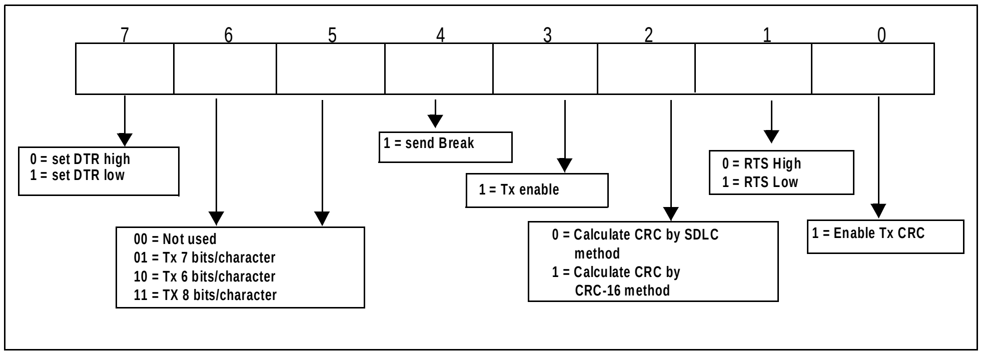


Figure 15.8 Write Register 5 (WR5) Bit Values and Configurations

Notes

Write Register 6

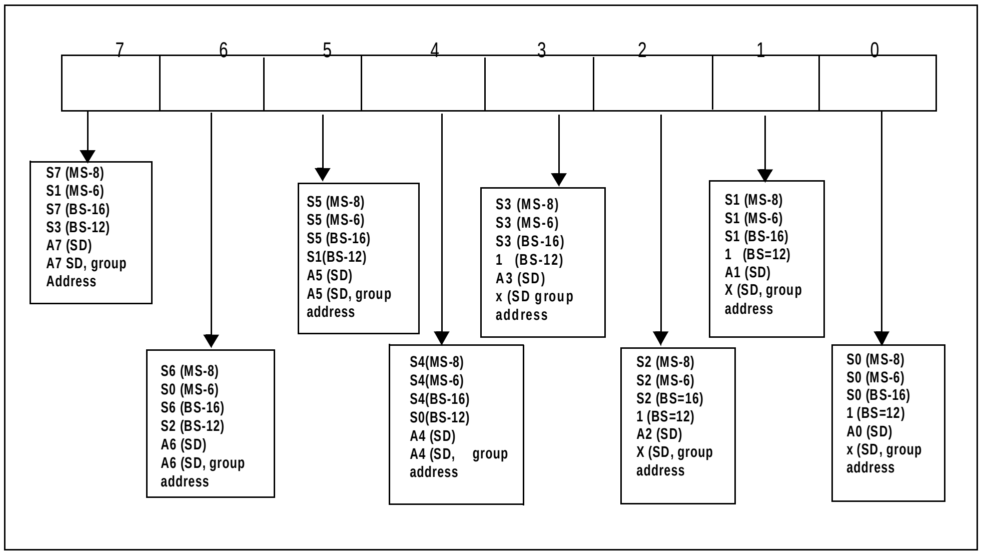


Figure 15.9 Write Register 6 (WR6) Bit Values and Configurations

Write Register 7

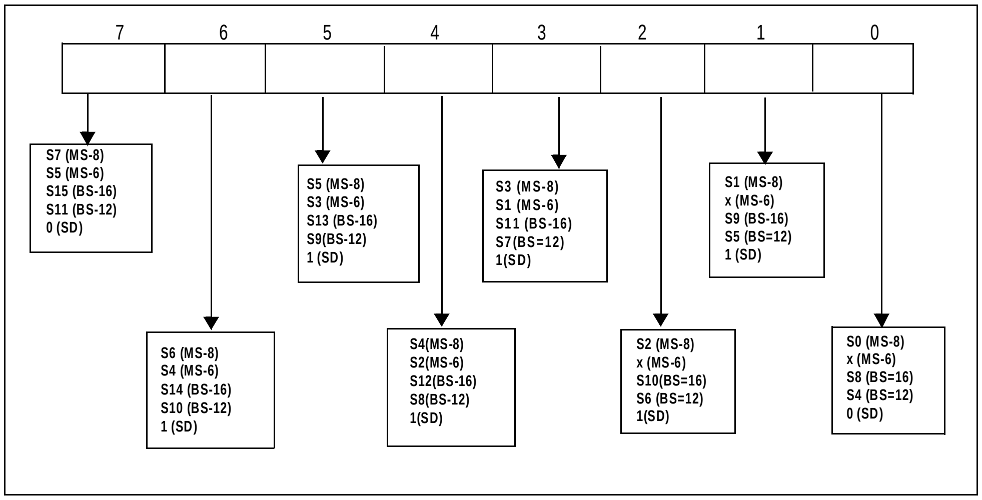


Figure 15.10 Write Register 7 (WR7) Bit Values and Configuration

Notes

Write Register 7’

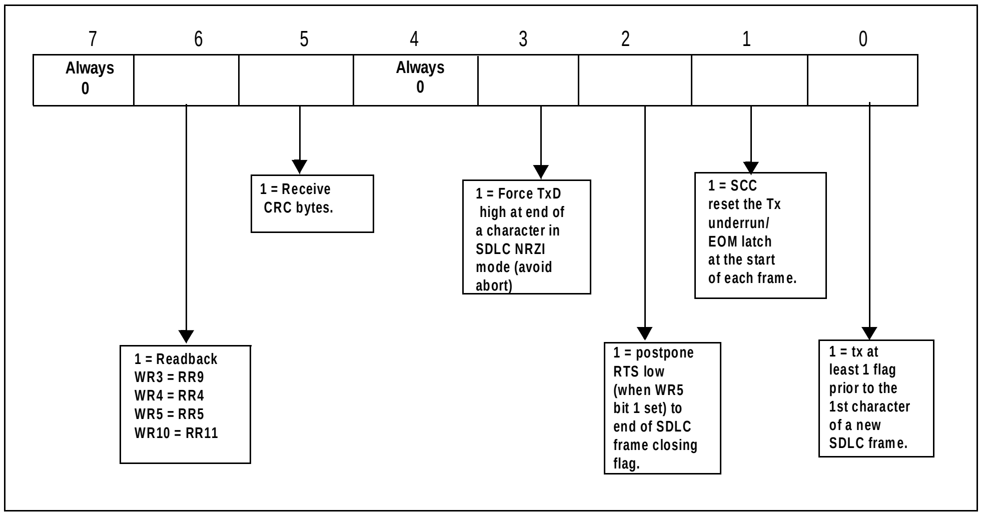

7
6
5
4
3
2
1
0
Always 0
Always 0
1 = Receive CRC bytes.
1 = Force TxD high at end of a character in SDLC NRZI mode (avoid abort)
1 = SCC reset the Tx underrun/ EOM latch at the start of each frame.
1 = Readback WR3 = RR9 WR4 = RR4 WR5 = RR5 WR10 = RR11
1 = postpone RTS low (when WR5 bit 1 set) to end of SDLC frame closing flag.
1 = tx at least 1 flag prior to the 1st character of a new SDLC frame.

Figure 15.11 Write Register 7’ (WR7’) Bit Values and Configuration

Write Register 8 is the Data Register.

Write Register 9

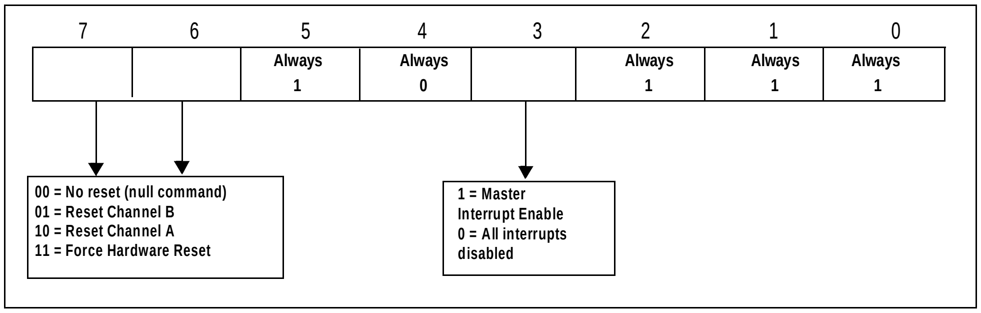

7
6
5
4
3
2
1
0
Always 1
Always 0
Always 1
Always 1
Always 1
00 = No reset (null command)
01 = Reset Channel B
10 = Reset Channel A
11 = Force Hardware Reset
1 = Master Interrupt Enable
0 = All interrupts disabled

Figure 15.12 Write Register 9 (WR9) Bit Values and Configuration

Write Register 10

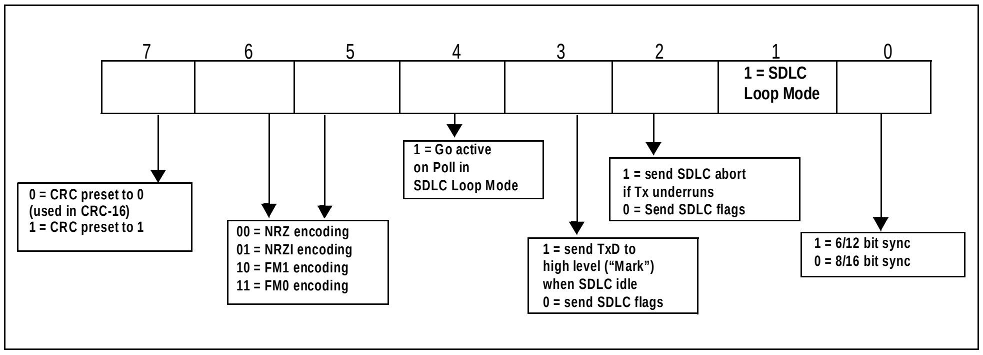


Figure 15.13 Write Register 10 (WR10) Bit Values and Configuration

Notes

Write Register 11

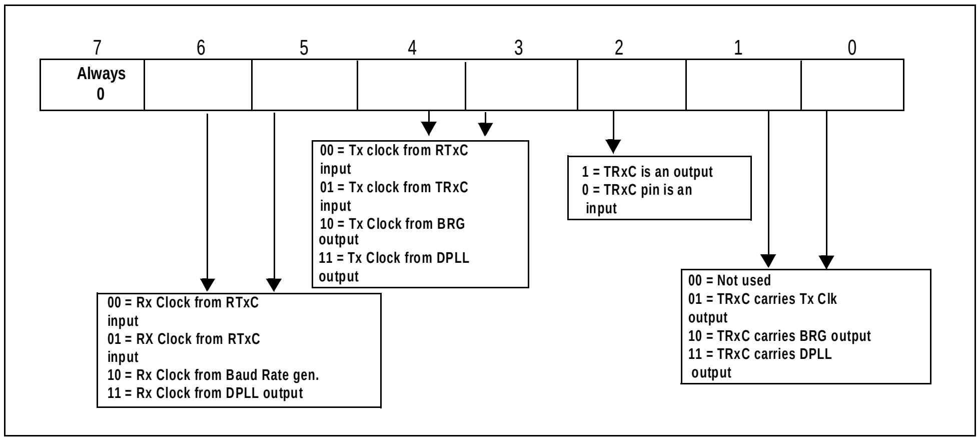

7
6
5
4
3
2
1
0
Always 0
00 = Tx clock from RTxC input
01 = Tx clock from TRxC input
10 = Tx Clock from BRG output
11 = Tx Clock from DPLL output
1 = TRxC is an output
0 = TRxC pin is an input
00 = Not used
01 = TRxC carries Tx Clk output
10 = TRxC carries BRG output
11 = TRxC carries DPLL output
00 = Rx Clock from RTxC input
01 = RX Clock from RTxC input
10 = Rx Clock from Baud Rate gen.
11 = Rx Clock from DPLL output

Figure 15.14 Write Register 11 (WR11) Bit Values and Configuration

Write Register 12 carries the low byte of the Baud Rate Generator Time Constant.

Write Register 13 carries the high byte of the Baud Rate Generator Time Constant.

Write Register 14

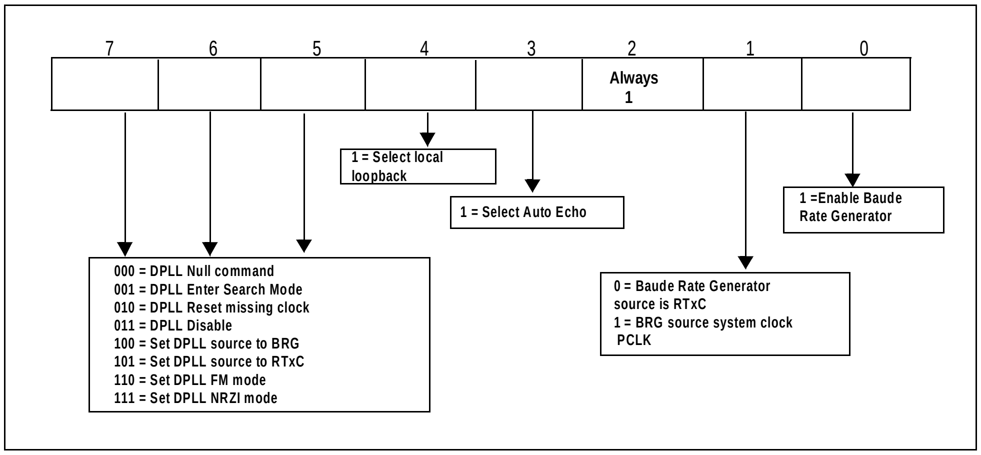

7
6
5
4
3
2
1
0
Always 1
1 = Select local loopback
1 = Select Auto Echo
1 =Enable Baude Rate Generator
000 = DPLL Null command
001 = DPLL Enter Search Mode
010 = DPLL Reset missing clock
011 = DPLL Disable
100 = Set DPLL source to BRG
101 = Set DPLL source to RTxC
110 = Set DPLL FM mode
111 = Set DPLL NRZI mode
0 = Baude Rate Generator source is RTxC
1 = BRG source system clock PCLK

Figure 15.15 Write Register 14 (WR14) Bit Values and Configuration

Notes

Write Register 15

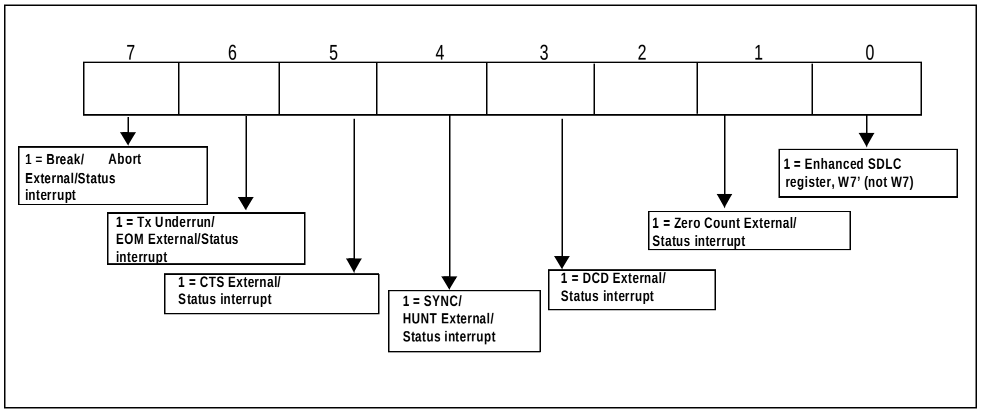

7 6 5 4 3 2 1 0
1 = Break/ Abort External/Status interrupt
1 = Tx Underrun/ EOM External/Status interrupt
1 = CTS External/ Status interrupt
1 = SYNC/ HUNT External/ Status interrupt
1 = DCD External/ Status interrupt
1 = Zero Count External/ Status interrupt
1 = Enhanced SDLC register, W7' (not W7)

Figure 15.16 Write Register 15 (WR15) Bit Values and Configuration

Read Register 0

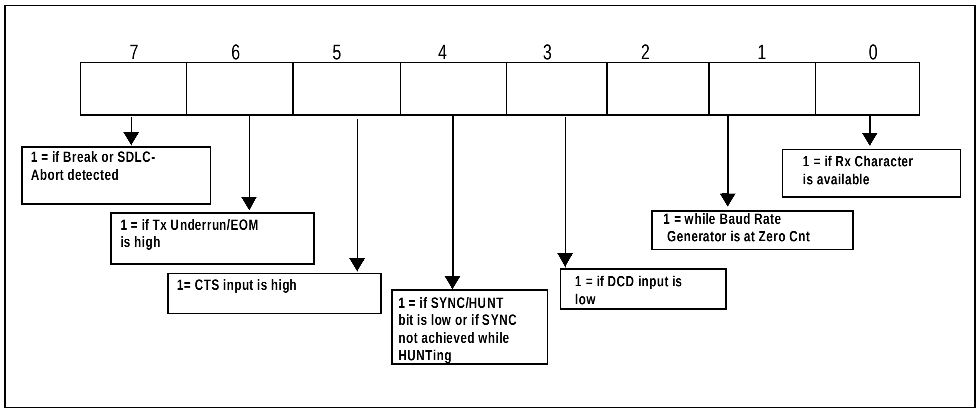

7 6 5 4 3 2 1 0
1 = if Break or SDLC-Abort detected
1 = if Tx Underrun/EOM is high
1= CTS input is high
1 = if SYNC/HUNT bit is low or if SYNC not achieved while HUNTing
1 = if DCD input is low
1 = while Baud Rate Generator is at Zero Cnt
1 = if Rx Character is available

Figure 15.17 Read Register 0 (RR0) Bit Values and Configuration

Read Register 1

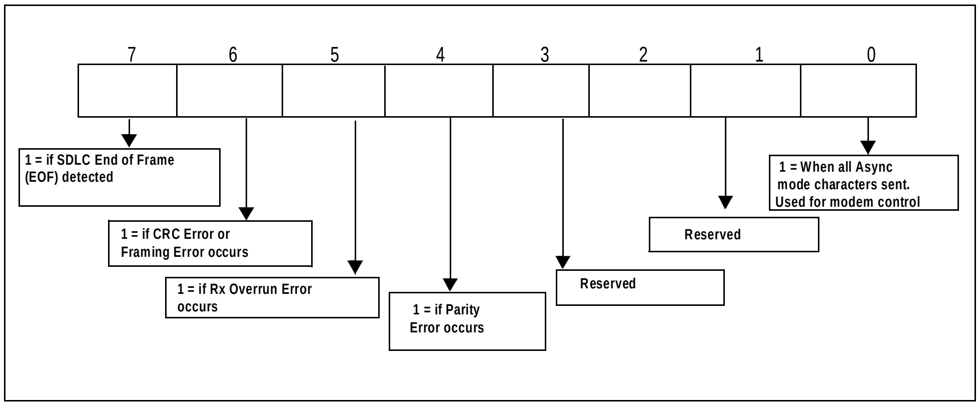

7 6 5 4 3 2 1 0
1 = if SDLC End of Frame (EOF) detected
1 = if CRC Error or Framing Error occurs
1 = if Rx Overrun Error occurs
1 = if Parity Error occurs
Reserved
Reserved
1 = When all Async mode characters sent. Used for modem control

Figure 15.18 Read Register 1 (RR1) Bit Values and Configuration

Read Register 2: Reads back WR2 in channel A, includes interrupt status (in bits 3:1) in Channel B.

Vector values are as follows:
000: Channel B Tx
001: Channel B E/S
010: Channel B Rx Data

Notes

011: Channel B Rx Special Condition
100: Channel A Tx
101: Channel A E/S
110: Channel A Rx Data
111: Channel A Rx Special Condition

Read Register 3 (Channel A only)

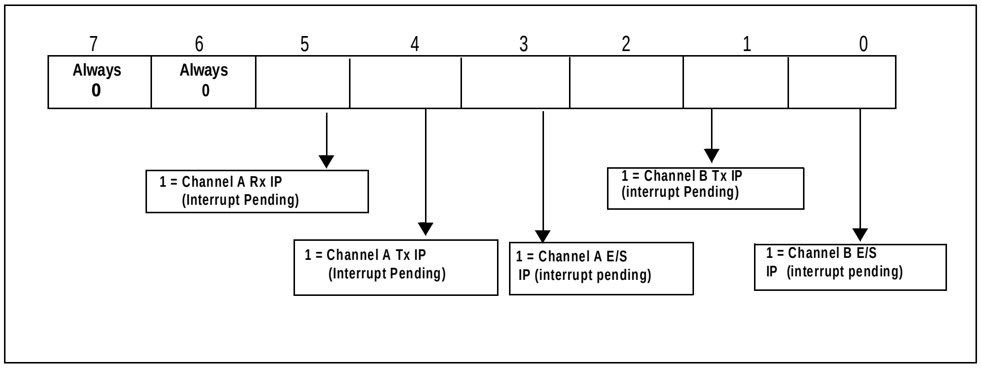

7
6
5
4
3
2
1
0
Always 0
Always 0
1 = Channel A Rx IP (Interrupt Pending)
1 = Channel B Tx IP (interrupt Pending)
1 = Channel A Tx IP (Interrupt Pending)
1 = Channel A E/S IP (interrupt pending)
1 = Channel B E/S IP (interrupt pending)

Figure 15.19 Read Register 3 (RR3) (Channel A only) Bit Values and Configuration

Read Register 4 is the readback register for WR4 if WR7' bit 6 is set.

Read Register 5 is the readback register for WR5 if WR7' bit 6 is set.

Read Register 6 carries the lowest byte of the Frame Size value within the topmost entry (if any) of the SDLC receive Frame Status FIFO.

Read Register 7

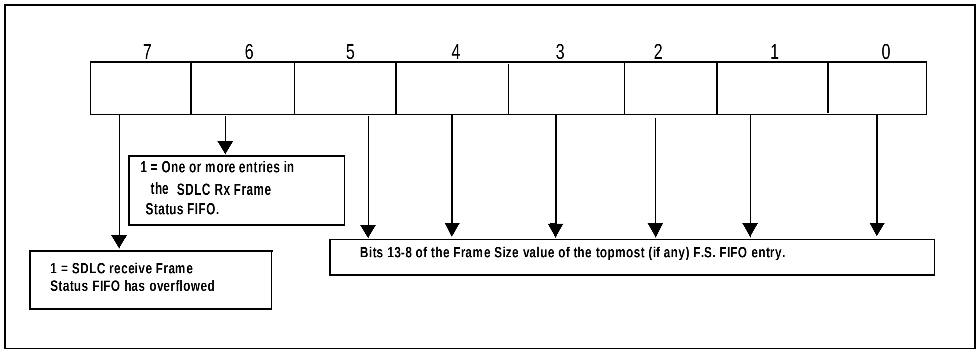

7
6
5
4
3
2
1
0
1 = One or more entries in the SDLC Rx Frame Status FIFO.
1 = SDLC receive Frame Status FIFO has overflowed
Bits 13-8 of the Frame Size value of the topmost (if any) F.S. FIFO entry.

Figure 15.20 Read Register 7 (RR7) Bit Values and Configuration

Notes

Bidirectional Parallel Port

Notes

Introduction

The RC36100 RISController integrates bus controllers and peripherals around the RISCore3000 family CPU core. One of the many on-chip peripherals is the **Bidirectional Centronics Parallel Port**, which is described in this chapter through a functional overview complete with signal pin descriptions and various aspects of their timing.

Features

- *Bidirectional ParallelPortTarget/Peripheral/PrinterController provided on-chip*
- *Provides 9 pin interface to Bidirectional Centronics IEEE 1284 Standard Parallel Port*
- *Provides 2 pins for host transceiver control*
- *Reuses 3 I/O Controller pins for peripheral transceiver control*
- *Uses external transceiver or bidirectional FIFO for data storage*
- *DMA auto-initiate via internal interrupt*
- *Compatible 8-bit input host to peripheral protocol (backward compatibility with Centronics standard)*
- *Nibble mode peripheral to host output protocol (Microsoft/PC standard)*
- *Byte mode peripheral to host output protocol (IBM PS2 applications)*
- *ECP bidirectional protocol (Windows PC/Laser standard)*
- *EPP bidirectional protocol (Datacom applications)*
- *200KBytes/sec to 1 MByte/sec data transfer rate*

Block Diagram

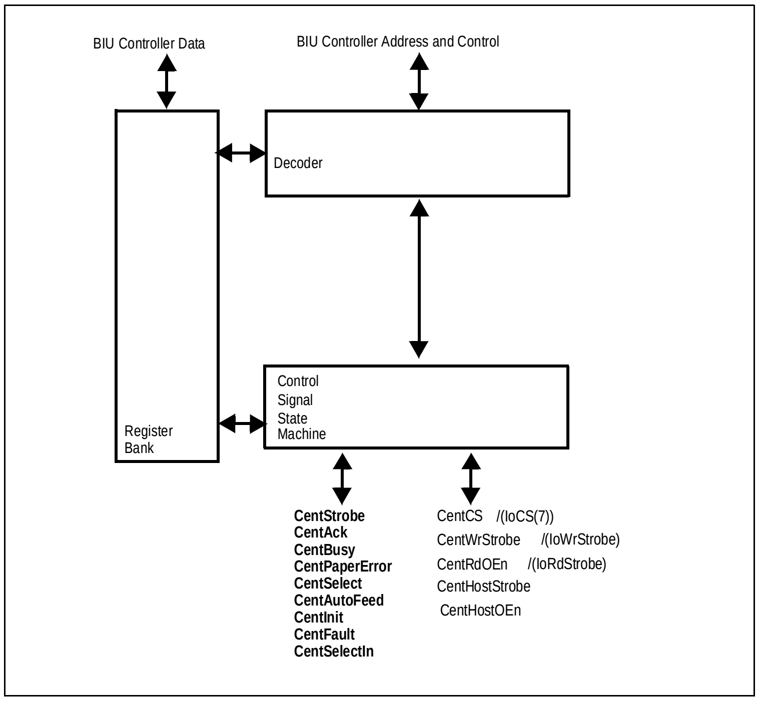


Figure 16.1 Block Diagram of the Bidirectional Parallel Port.

Notes

Overview

The Bidirectional Parallel Port Target/Peripheral/Printer Interface is an implementation of the interface described in “Standard Signaling Method for a Bidirectional Parallel Peripheral Interface for Personal Computers,” IEEE Standard 1284.

The purpose of this interfacing function is to allow Laser Printers or add-in communication cards (such as external SCSI drives or external Ethernet ports) to communicate with a PC host in both directions, by use of receive and transmit channels.

In the IBM PC compatible environment, the original Centronics compatible mode is only unidirectional and cannot report general purpose status information back to the PC host. However, with the addition of a reverse transfer 4-bit IEEE 1284 nibble mode, the Centronics port on the printer peripheral can now communicate bidirectionally with the majority of legacy PC hosts by using the present printer status lines to pass 4-bits at a time back to the PC host. In newer PC’s, such as the PS2 series, the PC hosts can use a truer bidirectional mode such as the IEEE 1284 reverse transfer byte mode.

The RC36100 also supports the newer IEEE 1284 Extended Capabilities Port mode (ECP) and IEEE 1284 Enhanced Parallel Port (EPP) mode, which provide more efficient interlocked handshaking as well as symmetric byte and host controlled read/write byte channel protocols, respectively. Both ECP and EPP are commonly found on Enhanced IDE I/O PC cards.

The Bidirectional Parallel Port Interface uses 14 pins (see block diagram in Figure 16.1). The pins include 9 control signals multiplexed in/out with PIO. The data lines are supported by either an external 8-bit data register transceiver chip or bidirectional FIFO that is controlled by the I/O Controller chip select pair, IoCS(7:6); the I/O read strobe; and the I/O write strobe. These pins also include 2 external register control lines: One is used to clock the data from the PC to the printer port, while the other enables the external register to the PC.

When used with two 8-bit external buffers/transceivers and a compliant physical connector, the RC36100 Bidirectional Parallel Port Interface implementation meets the IEEE 1284 definition of a compliant device. The RC36100 supports the following peripheral modes:

- *Compatible (standard forward transfer)*
- *nibble (4-bit reverse transfer)*
- *byte (8-bit reverse transfer)*
- *ECP (Extended Capabilities Port) (forward and reverse interlocked handshake transfers with arbitration for host/port control)*
- *EPP (Enhanced Parallel Port) (host controlled forward and reverse read/write-liketransfers)*

The RC36100 also contains support for the negotiation phase necessary for transition between the different modes. As described below, each mode has different phases associated with them:

Compatible Mode Phases are:

1. Forward Data Transfer
2. Forward Idle
3. (Negotiation)

Nibble and Byte Mode Phases are:

1. Forward Data Transfer
2. Forward Idle
3. Negotiation
4. Host Busy Data Available
5. Reverse Data Transfer
6. Host Busy Data Not Available
7. Reverse Idle
8. Interrupt Host
9. Terminate

Note: In nibble and byte modes, the RC36100 Centronics port always goes from state 3 --> 7 and then to state 8 --> 4, never from state 3 to 4. For data ready status, this requires the host to poll using the 7/4 states, not in the 3/4 states.

Notes

ECP Mode Phases are:

1. Forward Data Transfer
2. Forward Idle
3. Negotiation
4. Setup
5. Forward Idle
6. Forward
7. Forward to Reverse
8. Reverse Idle
9. Reverse
10. Reverse to Forward
11. Terminate

EPP Mode Phases are:

1. Forward Data Transfer
2. Forward Idle
3. Negotiation
4. Initial EPP Idle
5. Address Read
6. Data Read
7. Address Write
8. Data Write
9. EPP Idle
10. Terminate

Support for the compatible mode includes the three variations listed in Table 16.1.

Variation	Strobe/Busy	Busy/Ack
Centronics Classic	Busy-after-Strobe	Ack(2500 ns)-after-Busy
IBM/Epson	Busy-after-Strobe	Ack(2500 ns)-while-Busy
Standard 1284	Busy-while-Strobe	Ack(500 ns)-in-Busy

Table 16.1 Compatible Forward Data Transfer Variations.

Negotiation Phase

The Parallel Port Interface is initially put into “compatible mode” after reset. While in compatible mode, the host can send data out to the peripheral in a forward data transfer phase. To get into any of the other modes that support reverse data transfers, the port must undergo a negotiation phase in order to see if the port can support the requested mode. The Bidirectional Parallel Port Interface software driver must also configure the compatible mode to one of the three supported modes (IBM, classic, or standard) and to a data transfer option (DMA or interrupt per byte). Setting any of these modes and options is done by writing to the mode register.

In the interrupt per byte mode, the RC36100 will read data from the external Centronics Data Register each time it responds to a CentRdInt interrupt. In the DMA mode, the RC36100 will initialize one of the Internal DMA Channel Controllers register to the start of the DMA operation. The Bidirectional Parallel Port Interface software driver can be notified by interrupt when the DMA counter reaches the terminal count.

The negotiation is indicated by:

1. Host asserts 1284Active (nSelectIn) and de-asserts HostBusy (nAutoFd).
2. The peripheral responds by bringing AckDataReq (PError), nDataAvail (nFault), Xflag (Select) high and PtrClk (nAck) low.
3. Host nStrobes 8-bit extensibility request value on the data lines and also brings HostBusy (nAutoFd) high.
4. Peripheral sets Xflag (Select) to a particular value, and in the nibble and byte modes, nDataAvail (nFault) and AckDataReq (PError). Busy and Ptr (nAck) are set appropriately.

Notes

After step 1, the peripheral will generate a CentNegInt interrupt. After step 3, the RC36100 is interrupted by the Parallel Port Interface CentWrInt signal. The interrupt service routine must then read the extensibility request value from the external Centronics Data Register and write the appropriate mode and response bits back to the Parallel Port Interface so that it can finish the negotiation. If the extensibility link bit is asserted, then a second CentWrInt will occur during the negotiation.

A host request to return to compatibility mode, from any of the other modes, is indicated to the RC36100 by the assertion of the CentRstInt interrupt.

Nibble Mode Phase

When the host requests a two-nibble (8-bits total) transfer, the Parallel Port Interface interrupts the RC36100 by asserting CentWrInt. The RC36100 responds by writing data to the Parallel Port's Nibble Data Register. The Parallel Port Interface then sends the two nibbles out to the host over the appropriate Centronics control lines in two consecutive nibble transactions.

Byte Mode Phase

The Parallel Port Interface will interrupt the RC36100 by asserting CentWrInt when the host requests a byte transfer. The RC36100 will respond by writing data to the external Centronics Data Register.

Note: In Nibble and Byte Mode, the peripheral can arbitrate for the port, but only if it is left in reverse idle phase.

Extended Capabilities Port (ECP) Mode Phase

The ECP Mode allows both the host and the printer port to arbitrate for the bus and send commands/data to each other. A maximum of 128 different channels (communication streams) are supported by the protocol.

DMA and interrupt-per-byte options are supported for the ECP mode.

In the interrupt-per-byte option, the Parallel Port Interface will first assert CentRdInt for host read or write requests, and then it will assert CentWrInt for host write requests or CentRdInt for host read requests. The RC36100 will read or write from the external Centronics Data Register in response to the interrupts.

In reverse transfer, in response to CentWrInt, the RC36100 needs to write to the Parallel Port's Status Register (to the Busy bit) to indicate if it is sending a command or data byte, and also write the data/command to an external Centronics Data Register.

In forward transfer, in response to CentRdInt, the RC36100 needs to read from the Parallel Port's Command Register (nAutoFd bit) to see if the Centronics Data Register has a data or command byte. Run Length Encoding (RLE) Compression/decompression, if implemented, must be done by the software driver.

In the DMA transfer option, data will be transferred by an internal DMA channel as long as the direction of the host requests matches the direction of the DMA. Software must handle Centronics interrupts until the address and control is set up. Afterwards, a data stream can be handled by DMA. CentWrInt will be asserted when the host requests data. CentRdInt will be asserted when the host sends data or when the host sends a command byte.

Enhanced Parallel Port (EPP) Mode Phases

The EPP mode allows the host to address the printer port much like a read and writable memory interface. However, as per the IEEE1284 specification, the peripheral can not initiate transfers in this mode.

DMA and interrupt-per-byte options are supported for the EPP mode. In the interrupt-per-byte option, the Parallel Port Interface will assert CentRdInt for host read requests, and will assert CentWrInt for host write requests. The RC36100 will read or write from the external Centronics Data Register in response to the interrupts.

CentWrInt will be asserted when the host requests data (from the IEEE1284 port to the host). CentRdInt will be asserted when the host sends data or when the host sends an address byte. Software must handle Centronics interrupts until the address and control is set up. Afterwards, a data stream can be handled by DMA.

Notes

CPU Control Mode Phases

The CPU control mode allows direct control of the peripheral signals by writing values to the Bidirectional Parallel Port's status register.

Programmable Timing

To allow for higher data rates than those specified by the IEEE1284 Standard, the minimum delay—such as on Strobe/Busy and Busy/Ack— can be programmed to lower values than the minimum required by the standard.

Centronics, Interrupts & DMA Requests

$\overline{\text{CentRtcInt}}$

The Centronics Return to Compatibility Idle interrupt can be generated from the following conditions:

- *at the beginning of the negotiation mode phase. If generated due to the start of the negotiation phase, then the Negotiation Interrupt Pending field (bit 11) of the control register will be set.*
- *when the Initialize Negated field (nInit) is asserted and the Select In Negated field (nSelectIn, 1284Active) is not asserted. If generated due to the assertion of the nInit field bit, the iprime Interrupt Pending field (bit 14) of the control register will be set.*

Note: When the reset Interrupt pending field is set, the iprime Interrupt field may or may not be set, depending on the state of the nInit field bit.

- *when the peripheral returns to the compatibility mode idle. If generated because a return to the compatibility mode phase has occurred, then the reset interrupt pending field (bit 9) of the control register will be set.*

Note: A return to the compatibility idle phase may be caused when the nInit field (bit 2 of the Centronics Host Status Register) is asserted while the nSelectIn (bit 1 of the Host Status Register, 1284 Active) field is not asserted and the Bidirectional Parallel Port has not been set to the CPU control mode. Termination (either valid or immediate) of one of the IEEE 1284 modes will also cause a return to the compatibility mode.

To clear a $\overline{\text{CentRtcInt}}$ interrupt generated by the peripheral, write 1 to the appropriate pending bit (bit 9, bit 11, or bit 14) of the control register in the parallel port interface, write 1 to bit 12 of the interrupt pending register 0 in the interrupt controller.

$\overline{\text{CentRdInt}}$

The Centronics Read Interrupt will be generated during a forward transfer[1] phase to read information from an external Centronics data register into the peripheral. To clear a CentRdInt interrupt generated by the peripheral, write 1 to bit 14 of the interrupt pending register 0 in the interrupt controller.

$\overline{\text{CentrWrInt}}$

The Centronics Write Interrupt will be generated during a negotiation to read an extensibility request value from an external Centronics data register. This interrupt will also be generated during a reverse transfer[1] phase to write information to an external Centronics data register from the peripheral. To clear a CentWrInt interrupt generated by the peripheral, write 1 to bit 13 of the interrupt pending register 0 in the interrupt controller.

DMARdReq

A DMA Read Request is generated during a forward transfer[1] phase to read information from an external Centronics data register and write it to the DMA target location.

---

1. Depending on the parallel port mode, a transfer can be data, commands, or addresses.

Notes

DMAWrReq

A DMA Write Request is generated during a reverse transfer[1] phase to write information read from a DMA source location to an external Centronics data register.

Pin Descriptions

Note that the following pin descriptions are given in terms of the Centronics Modes. Actually, each mode has various phases that may further define the functionality of the signal. Please refer to IEEE 1284 Standard for additional detail.

Bidirectional Parallel Port Centronics Interface Signals

$\overline{\text{CentStrobe}}$ Input

(Aliases: nStrobe, HostClk, nWrite)
Centronics Strobe:
Compatibility: Data strobe.
Nibble: Acknowledges reverse data transfer.
Byte: Acknowledges reverse data transfer.
ECP: Handshakes with Busy.
EPP: Indicates Address write or Data write.

$\overline{\text{CentAck}}$ Output

(Aliases: nAck PtrClk, PeriphClk, Int)
Centronics Acknowledge:
Compatibility: Data Acknowledge.
Nibble: Data Acknowledge.
Byte: Data Acknowledge.
ECP: Handshakes with HostAck.
EPP: Active High Interrupt.

CentBusy Output

(Aliases: Busy PtrBusy, PeriphAck, nWait)
Centronics Busy:
Compatibility: Active high indication that the peripheral is busy.
Nibble: In later phases of nibble mode, Data bit 3 and 7.
Byte: Active high indication that the peripheral is busy.
ECP: Flow control in the forward direction, Command/Data bit in the reverse direction.
EPP: Active low wait signal delaying an address or data.

CentPaperError Output

(Aliases: PError, AckDataReq, nAckReverse)
Centronics PaperError:
Compatibility: When asserted with nFault, indicates a Paper Error. Additional uses during other phases including 1284Support.
Nibble: Data bits 2 and 6.
Byte: Same as nFault.
ECP: Request nReverseRequest.
EPP: User Defined (unused).

CentSelect Output

(Aliases: Select Xflag, Xflag)

Centronics Select:

Compatibility: Peripheral is on line. In other phases, the Extensibility Flag (XFlag).

1. Depending on the parallel port mode, a transfer can be data, commands, or addresses.

Notes

Nibble: Data bits 1 and 5.
Byte: Peripheral is on line. In other phases, the Extensibility Flag (XFlag).
ECP: In some phases, the Extensibility Flag (XFlag).
EPP: User Defined (unused).

$\overline{\text{CentAutoFeed}}$ Input

(Aliases: nAutoFd HostBusy, HostAck, nDStrb)

Centronics Auto Feed:

Compatibility: Typically indicates auto linefeed mode, but often unused or redefined. Also used during Negotiation Phase as HostBusy.
Nibble: Typically indicates auto linefeed mode. In other phases, used for several purposes.
Byte: Typically indicates auto linefeed mode. In other phases, used for several purposes.
ECP: Handshakes with PeriphClk.
EPP: Denotes data cycle.

$\overline{\text{CentInit}}$ Input

(Aliases: nInit, nReverseRequest)

Centronics Initialize:

Compatibility: When pulsed with 1284Active de-asserted, resets to idle phase.
Nibble: When pulsed with 1284Active de-asserted, resets to idle phase.
Byte: When pulsed with 1284Active de-asserted, resets to idle phase.
ECP: Host allows the peripheral to drive the bi-directional data signals.
EPP: When asserted, resets to compatibility mode.

$\overline{\text{CentFault}}$ Output

(Aliases: nFault nDataAvail, nPeriphRequest)

Centronics Fault:

Compatibility: Set low indicating an error. In other phases, set high to ack 1284, data available.
Nibble: Data bits 0 and 4.
Byte: Set low indicating an error. In other phases, set high to ack 1284 and data available.
ECP: Peripheral requests communication with host which host may chose to ignore.
EPP: User defined (unused).

$\overline{\text{CentSelectIn}}$ Input

(Aliases: nSelectIn 1284Active, 1284Active, nAStrb)

Centronics Select Input:

Compatibility: Selects this peripheral (if Centronics is shared). In some phases, indicates 1284Active.
Nibble: Selects this peripheral (if Centronics is shared). In some phases, indicates 1284Active.
Byte: Selects this peripheral (if Centronics is shared). In some phases, indicates 1284Active.
ECP: Active high, if de-asserted, return to compatibility mode.
EPP: Indicates an address cycle.

Bidirectional Parallel Port Centronics Peripheral and Host Interface Signals

$\overline{\text{CentCS(7:6)}}$ Output

Centronics I/O Chip Select: Use either 1 or both of the $\overline{\text{IoCS(7:6)}}$ pins to create this signal. The RC36100 uses this active low signal to select the externally provided 8-bit Centronics Data Register/ Transceiver. With some types of transceivers, $\overline{\text{CentCS(7:6)}}$ must both be used and/or they must be externally gated with $\overline{\text{CentRdStrobe}}$ and or $\overline{\text{CentWrStrobe}}$.

Notes

CentWrStrobe Output

Centronics Write Strobe: Use IoWrStrobe to create this signal. The RC36100 uses this active low signal to write data into a registered transceiver so that the host may later retrieve the data. The transceiver must also be gated with an appropriately programmed IoCS().

CentRdOEn Output

Centronics Read Output Enable: Uses IoRdStrobe to create this signal. The RC36100 uses this active low signal to read data into the peripheral from the registered transceiver which the host had previously stored. The transceiver must also be gated with an appropriately programmed IoCS().

CentHostStrobe Output

Centronics Host Strobe: This signal is similar to CentStrobe, but it is active high and gated for actual host data writes, since CentStrobe is also used by various IEEE 1284 modes to acknowledge actions other than writes. Active high output is attached to an external registered transceiver in order to clock/latch-enable the data from the host into the registered transceiver. The CentHostStrobe pin is multiplexed with a PIO pin, and thus the PIO pin must be programmed to the CentHostStrobe special effect and to be an output.

CentHostOEn Output

Centronics Host Output Enable: Active low output attached to an external registered data transceiver in order to allow the host to read data from the registered transceiver. The CentrHostOEn pin is multiplexed with a PIO pin, and thus the PIO pin must be programmed to use the CentHostOEn special effect and to be an output.

Register Definitions

Table 16.2 lists the Bidirectional Parallel Port Interface addresses and descriptions. Note that Big Endian software must offset these addresses by b'10 (0x2).

Phys. Address	Description
0xFFFF_EC00	Centronics Sub Mode Register
0xFFFF_EC04	Centronics Status Register
0xFFFF_EC08	Centronics Control Register
0xFFFF_EC0C	Centronics Nibble Data Register
0xFFFF_EC10	Centronics Host Status Register
0xFFFF_EC14	Centronics Minimum Delay Register
0xFFFF_EC18	Centronics LSB Host time-out Register
0xFFFF_EC1C	Centronics MSB Host time-out Register
0xFFFF_EC20	Centronics LSB Host time-out Counter
0xFFFF_EC24	Centronics MSB Host time-out Counter

Table 16.2 Bidirectional Parallel Port Interface Centronics Controller Registers.

Notes

Centronics Sub Mode Register ('CentSubModeReg')

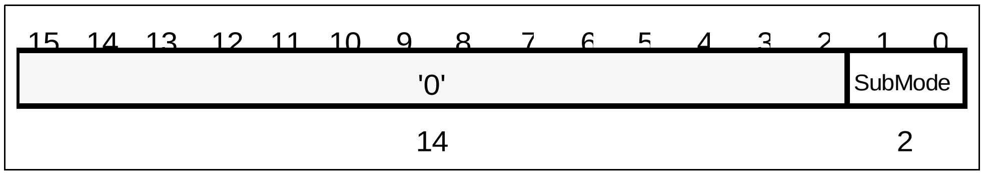


Figure 16.2 Centronics Sub ModeRegister ('CentSubModeReg').

Additional programming information and instructions for the Centronics Sub Mode Register are located in Figure 16.2, Table 16.3, and Table 16.4.

Bit	Description
1:0	CMode

Table 16.3 Centronics Sub ModeRegister ('CentSubModeReg') Bit Assignments.

Centronics Compatible Sub Modes ('SubMode') Field

Value	Action
'11'	Classic Centronics.
'10'	Reserved.
'01'	IBM/Epson.
'00'	Standard 1284 (default).
Note: See Table 16.1 for more information.	

Table 16.4 Centronics Compatible Sub Mode ('SubMode') Field Encoding.

Centronics Status Register ('CentStatusReg')

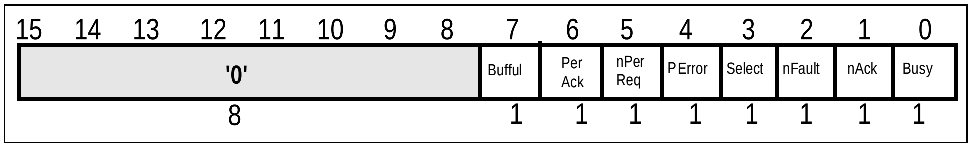


Figure 16.3 Centronics Status Register ('CentStatusReg').

The Centronics Status Register is shown in Figure 16.3. Additional programming information and instructions for this register are located in Table 16.5, Table 16.16, Table 16.17, Table 16.18, Table 16.19, Table 16.20, Table 16.9, Table 16.10, Table 16.11, Table 16.12, and Table 16.13.

Notes

Bit	Description
7	ECP Peripheral Buffer Full (BufFul)
6	ECP Peripheral Acknowledge (PerAck)
5	ECP Peripheral Request Negated (NPerReq)
4	Printer Error (PError)
3	Printer On Line Select (Select)
2	Printer Fault Negated (nFault)
1	Printer is Acknowledging Negated (nAck)
0	Printer is Busy (Busy)

Table 16.5 Centronics Status Register ('CentStatusReg') Bit Assignments.

Extended Capabilities Port Buffer Full (BufFul)

Through software, while in the ECP mode, the BufFul bit in the status register can be used to indicate whether the peripheral buffer is full. If the BufFul bit is set, then the Centronics Parallel Port Interface will hold off forward transfers, and the host side will then be unable to continue transferring data, preventing peripheral buffer overrun.

Value	Action
'1'	Peripheral buffer full during forward transfer in ECP mode.
'0'	Peripheral buffer not full during forward transfer in ECP mode.

Table 16.6 ECP Buffer Full Field

Extended Capabilities Peripheral Acknowledge Field (nPerAck)

Note that in the ECP mode status register, the nPerReq and nPerAck fields are used to replace nFault and Busy fields, respectively. This eases software driver development, because nFault and Busy fields do not need to be saved when the parallel port changes from compatibility mode to ECP mode.

Value	Action
'1'	Reverse transfer Data in ECP mode.
'0'	Reverse transfer Commands in ECP mode.

Table 16.7 ECP Peripheral Acknowledge Field

Extended Capabilities Peripheral Request Field (nPerReq)

Value	Action
'1'	Peripheral request communication with the host in ECP mode.
'0'	Peripheral not request communication with the host in ECP mode.

Table 16.8 ECP Peripheral Request Field

Notes

Printer Error ('PError') Field:

Value	Meaning
'1'	Printer Error
'0'	Normal (default).

Table 16.9 Printer Error Field ('PError') Field Encoding.

Printer On Line Select ('Select') Field:

Value	Meaning
'1'	Printer on line
'0'	Printer off line (default).

Table 16.10 Select ('Select') Field Encoding.

Printer Fault ('Fault') Field:

Value	Meaning
'1'	Printer Fault
'0'	Normal (default).

Table 16.11 Printer Fault ('Fault') Field Encoding.

Printer Acknowledge Negated ('AckN') Field:

Value	Meaning
'1'	Normal
'0'	Printer Acknowledge (default).

Table 16.12 Printer Acknowledge Negated ('AckN') Field Encoding.

Printer Busy ('Busy') Field:

Value	Meaning
'1'	Printer Busy
'0'	Printer not Busy (default).

Table 16.13 Printer Busy Field ('Busy') Encoding.

Centronics Control Register ('CentControlReg')

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
'0'	IPrime Pending	CS(7) Mask	CS(6) Mask	Neg Pend	Neg En	Reset Pend	Reset En	0	DMA Int	Nib ID	HBDA En	NegRep	NegMode		
1	1	1	1	1	1	1	1	1	1	1	1	1	3		

Figure 16.4 Centronics Control Register ('CentControlReg').

Notes

Figure 16.4 illustrates the Centronics Control Register and its fields. Field encoding values are given in Table 16.14, Table 16.24, and Table 16.25.

Bit	Description
14	Iprime Interrupt Pending Field (Iprime Pending)
13	IoCS(7) Mask Enable
12	IoCS(6) Mask Enable
11	Negotiation Interrupt Pending
10	Negotiation Interrupt Enable
9	Reset Interrupt Pending
8	Reset Interrupt Enable
6	DMA or Interrupt in ECP/EPP mode
5	Nibble ID mode supported
4	Host busy data available (HBDA) enable
3	Negotiation XFlag Reply (NegRep)
2:0	Negotiation Mode (NegMode)

Table 16.14 Centronics Control Register (‘CentControl’) Bit Assignments.

Iprime Interrupt Pending Field

Value	Action
1	On read, Iprime interrupt pending
	On write, clear Iprime interrupt
0	On read, no Iprime interrupt pending
	On write, do not change current interrupt state

Table 16.15 Iprime Interrupt Pending Field Encoding

The Iprime interrupt is generated when the host asserts the nInit signal and the nSelectIn (1284Active) is high. When this occurs, all of the parallel port state machines will be reset. The Iprime Interrupt Pending field fit is set whenever the host has asserted nInit and the nSelectIn signal is not asserted.

Thus, bit 14—together with bit 9 and bit 11 of the control register—can be used to determine the cause of a CentRtcInt interrupt. If bits 14 and 9 are both set, then the CentRtcInt interrupt is caused by the assertion of the nInit signal while nSelectIn is not asserted and the peripheral is not in the CPU control mode. If only bit 9 is set to 1, then the CentRtcInt interrupt is caused by the termination of one of the IEEE 1284 modes.

IoCS(7), IoCS(6) Mask Enable

Typically, a system using the RC36100 bidirectional centronics interface will use an external buffer to buffer data between the CPU bus and the Centronics port. These bits control whether one or both of IoCS(7:6) signals are used to control that buffer. The most common strategy is to use one IoCS for reads and one for writes; although, other systems may simply use one IoCS and decode reads or writes from the control bus.

In addition, when writing to the external Centronics data register, SysAddr(15) must be low.

Notes

Value	Action
'1'	Use IoCS for Centronics (SysAddr(15) must be low on writes)
'0'	Don't use IoCS for Centronics

Table 16.16 IoCS(7:6) Mask Enable Field Encoding

Centronics Negotiation Interrupt Pending

Value	Action
'1'	On reads, means negotiation interrupt is pending.
	On writes, clears the pending interrupt.
'0'	On reads, means no negotiation interrupt is pending.
	On writes, means do not change current interrupt state.

Table 16.17 Centronics Negotiation Interrupt Pending Field.

Centronics Negotiation Interrupt Enable

Value	Action
'1'	Signal Pending Negotiation Interrupts to the interrupt controller.
'0'	Do not signal the interrupt controller (default).

Table 16.18 Centronics Negotiation Interrupt Enable Field.

Centronics Reset Interrupt Pending

Value	Action
'1'	On reads, means reset interrupt is pending.
	On writes, clears the pending interrupt.
'0'	On reads, means no reset interrupt is pending.
	On writes, means do not change current interrupt state.

Table 16.19 Centronics Reset Interrupt Pending Field.

Centronics Reset Interrupt Enable

Value	Action
'1'	Signal Pending Reset and Pending iprime Interrupts to the interrupt controller.
'0'	Do not signal the interrupt controller (default).

Table 16.20 Centronics Negotiation Interrupt Enable Field.

Notes

DMA or Interrupt Field in ECP/EPP Mode

Value	Action
1	In ECP mode forward transfer, use DMA for data and Interrupt for commands.
	In EPP mode, use DMA for data and interrupt for addresses
0	In ECP mode forward transfer, use interrupt for data or commands.
	In EPP mode forward transfer, use interrupt for data or addresses

Table 16.21 DMA or Interrupt in ECP/EPP Mode Field Encoding

When this field is set to 1, in forward ECP mode or in EPP mode, all data transfers are DMA based and command transfers (in forward ECP mode) or address (in EPP mode) transfers are interrupt based.

Note: The reverse ECP mode is not affected by setting this field.

In forward ECP mode or in EPP mode, if DMA is desired, then this field must be set to 1. Otherwise, if this field is set to 0, then interrupt-per-byte is used for all transfers, including data, commands, or addresses. For other modes, this field is don't care. Either interrupt based or DMA based can be used to perform transfers depending on the setup in the interrupt or DMA controllers.

Nibble ID Mode Field

Value	Action
1	Nibble ID mode is supported
0	Nibble ID mode is not supported

Table 16.22 Nibble ID Mode Field Encoding

During Negotiation, if the Centronics Parallel Port is to support the Nibble ID mode, then set the Nibble ID Field in the control register to 1, set NegRep Field to 0, and set the negotiation mode field to the nibble mode (001).

Host Busy Data Available Enable Field (HBDA)

Value	Action
1	Host Busy data available is enabled
0	Host Busy data available is not enabled

Table 16.23 Host Busy Data Available Enable Field Encoding

When the host busy data available is true in nibble, byte, nibble ID, or byte ID mode, setting this field to 1 enables more efficient reverse transfers. If this field is not enabled when host busy data available is true, then the reverse transfer will still occur, however, more delay will be incurred than when enabled. Note that this field must be set to 0 when the host busy data available is not true.

Notes

Negotiation XFlag Reply ('NegRep') Field:

Value	Action
'1'	Nibble Mode: mode requested not supported.
	Other Modes: mode requested supported (default).
'0'	Nibble Mode: mode requested supported.
	Other Modes: mode requested not supported.

Table 16.24 Negotiation XFlag Reply ('NegRep') Field Encoding.

Negotiation Mode ('NegMode') Field:

Value	Action
b'111'	Termination
b'110'	Extensibility Link
b'101'	CPU Control
b'100'	EPP
b'011'	ECP
b'010'	Byte
b'001'	Nibble
b'000'	Compatible (default)

Table 16.25 Negotiation Mode ('NegMode') Field Encoding.

During negotiation, if nibble mode is not supported, write 1 to the NegRep field and 111 (termination) to the NegMode field in the control register. For any other unsupported IEEE 1284 modes (including nibble ID), write 0 to the NegRep field and 111 (termination) to the NegMode field in the control register.

If nibble mode is supported, write 0 to the NegRep field and 001 (nibble) to the NegMode field. If nibble ID is supported, write 0 to NegRep field, 001 (nibble) to NegMode field, and 1 to the nibble ID field in the control register. For all other supported IEEE 1284 modes, write 1 to the NegRep field and then write to the appropriate Negotiation Mode field in the control register.

Centronics Nibble Data Register ('CentNibbleDataReg')

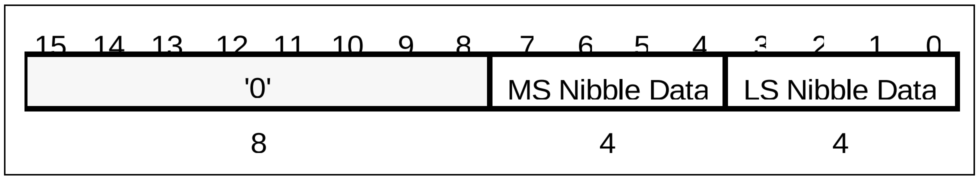


Figure 16.5 Centronics Nibble Data Register ('CentNibbleDataReg').

The Centronics nibble data register cannot be written to until the peripheral recognizes the $\overline{\text{CentWrInt}}$ signal generated to request nibble data for a reverse transfer in nibble mode. During the nibble mode reverse transfer, two consecutive nibbles of data are taken from this register and transferred using four parallel interface control lines.

Notes

Bit	Description
7:4	Most Significant Nibble Data
3:0	Least Significant Nibble Data

Table 16.26 Centronics Nibble Data Register ('CentNibbleDataReg') Bit Assignments.

Figure 16.5 and Figure 16.6 are illustrations of the Centronics Nibble Data and Host Status Registers. Additional programming instructions for these registers are located in Table 16.26, Table 16.27, Table 16.28, Table 16.29, Table 16.30, Table 16.31.

Centronics Host Status Register ('CentHostStatusReg')

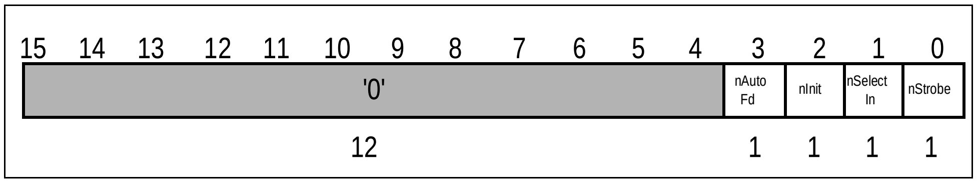


Figure 16.6 Centronics Host Status Register ('CentHostStatusReg').

Bit	Description
3	AutoFeed Negated (nAutoFd)
2	Initialize Negated (nInit)
1	Select In Negated (nSelectIn)
0	Host Strobe Negated (nStrobe)

Table 16.27 Centronics Host Status Register ('CentHostStatusReg') Bit Assignments.

AutoFeed Negated ('nAutoFeed') Field:

Value	Meaning
'1'	Normal (default).
'0'	AutoFeed mode.

Table 16.28 AutoFeed Negated ('nAutoFeed') Field Encoding.

Initialize Negated ('nInit') Field:

Value	Meaning
'1'	Normal (default).
'0'	Initialize the printer.

Table 16.29 Initialize Negated ('nInit') Field Encoding.

Notes

Select In Negated ('nSelectIn') Field:

Value	Meaning
'1'	Don't select this printer (default).
'0'	Select this printer.

Table 16.30 Select In Negated ('nSelectIn') Field Encoding.

Host Strobe Negated ('nStrobe') Field:

Value	Meaning
'1'	Host Strobe pulse de-asserted (default).
'0'	Host Strobe pulse asserted.

Table 16.31 Host Strobe Negated ('nStrobe') Field Encoding.

Centronics Minimum Delay Register ('CentDelayReg')

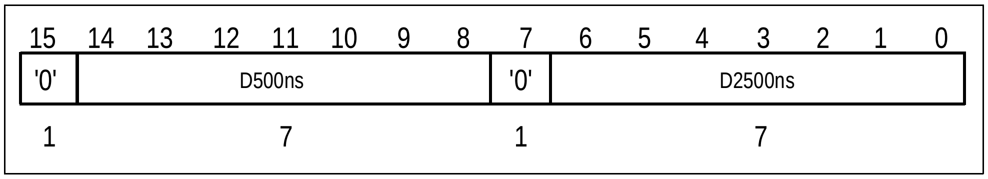


Figure 16.7 Centronics Minimum Delay Register ('CentDelayReg').

The Centronics Minimum Delay Register allows programmable parallel port interface delay timing. The D500ns field is used for minimum delays in IEEE 1284 modes, while the D2500ns field is used for minimum delays in the compatibility mode.

Figure 16.7 is an illustration of the Centronics Minimum Delay Register. Refer to Table 16.32, Table 16.33, Table 16.34, and Table 16.35 for additional programming information.

Bit	Description
15	Reserved. Must be written as '0'.
14:8	200ns Delay Type Field (D500ns)
7	Reserved. Must be written as '0'.
6:0	2500ns Delay Type Field (D2500ns)

Table 16.32 Centronics Minimum Delay Register ('CentDelayReg') Bit Assignments.

Notes

2500ns Delay Type Field ('D2500ns') Field:

Value	Action
'0x7f'	0x7f clock delay.
...	
'0x01'	0x01 clock delay.
'0x00'	undefined. (default).

Table 16.33 2500ns Delay Type Field ('D2500ns') Field Encoding.

500ns Delay Type Field ('D500ns') Field:

Value	Action
'0x7f'	0x7f clock delay.
...	
0x01'	0x01 clock delay.
'0x00'	undefined. (default).

Table 16.34 500ns Delay Type Field ('D500ns') Field Encoding.

CPU MHz	2500ns	500ns
33	0x53	0x11
25	0x3f	0xOd
20	0x32	0xOa
16	0x28	0x08

Table 16.35 Example Settings for Delay Type Fields.

LSB/MSB Host Time-Out Register

The 16-bit LSB Host Time-Out Counter register and the 10-bit MSB Host Time-Out Counter register form a 26-bit compare register. When the value set in these two registers equals the host time-out counter value, a host time-out occurs. Both registers must be set to an appropriate value so that the peripheral has a host time-out delay of one second. This 26-bit host time-out counter allows maximum parallel port input clock frequencies of 67.11 MHz with a one second host time-out delay.

LSB/MSB Host Time-Out Counter Register

The 16-bit LSB Host Time-Out Counter and the 10-bit Host Time-Out Counter form a 26-bit host time-out counter, which is readable/writable when the host time-out counter is enabled during host response time period for interface testing purposes. This counter is for diagnostic purposes only and is not intended to be read from or written to by users.

Timing Diagram

Figure 16.8 illustrates a typically classic compatible mode transaction.

Note: All Centronics signals are generated or sampled on the falling edge of $\overline{\text{SysClk}}$.

Notes

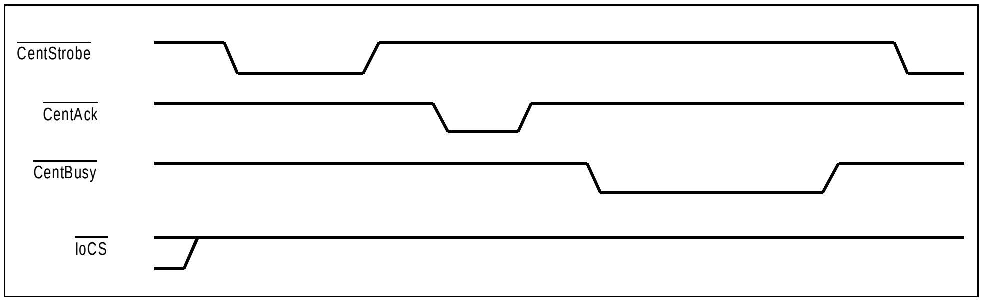


Figure 16.8 Typically Classic Compatible Mode Transaction.

System Example

Figure 16.9 illustrates typical parallel port system connections.

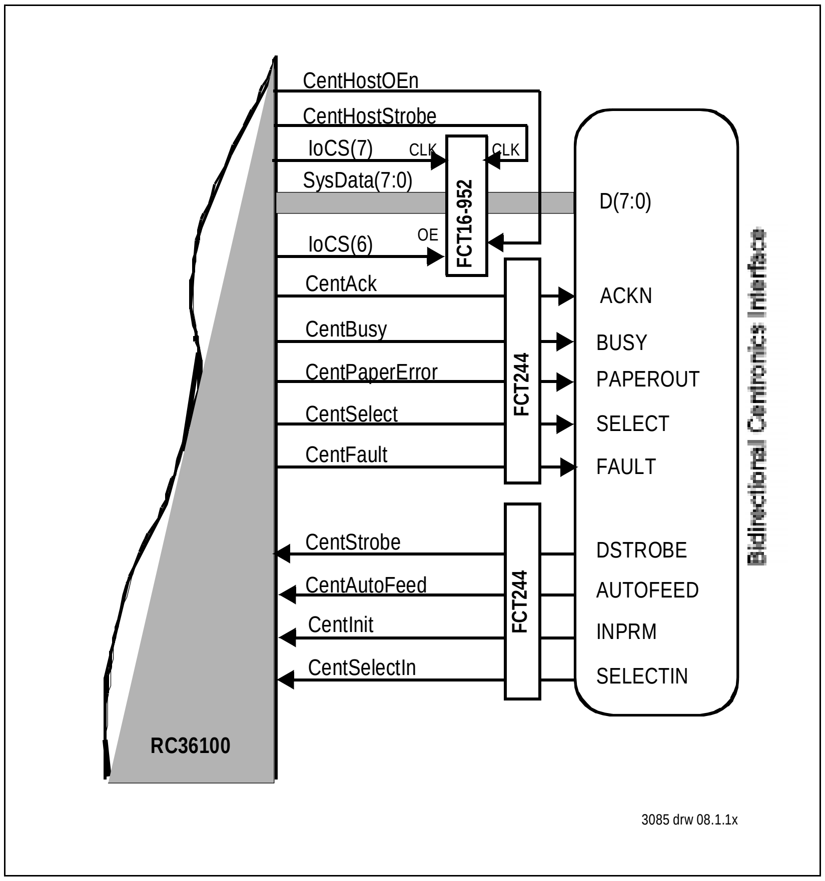


Figure 16.9 Typical Parallel Port System Connections.

Note: The virtual address for the external Centronics data register write clock, IoCS(7), (in this example, FCT16-952) must have address bit 15 low.

Notes

Reset Initialization and Input Clocking

Notes

Introduction

There are a number of initialization selectable features in the RC36100. These mode selectable features are determined by the polarity of the appropriate reset configuration mode inputs, when the rising edge of SysReset occurs.

This chapter discusses the reset initialization sequence that is required by the RC36100 and includes information on the processor's configuration mode selectable features and the boot program software requirements.

Reset Timing

The RC36100 requires a very simple reset sequence. There are only two concerns for the system designer:

- *That the set-up and hold time requirements of the reset configuration mode feature inputs with respect to the rising edge of SysReset are met.*
- *That the minimum SysReset pulse width is satisfied.*

Reset Configuration Mode Features

The RC36100 has features that are determined at reset time. This is achieved by using a latch internal to the CPU: this latch samples the contents of the reset mode feature bus at the negating edge of SysReset. The encoding of the mode selectable features on the reset mode feature bus is described in Table 17.1.

Pin	Mode Feature
ExcSInt(0)	LittleEndian/BigEndian
ExcSInt(1)	BootProm16
ExcSInt(2)	BootProm8

Table 17.1 RC36100 Reset Configuration Mode Features

Reset Configuration Mode Pin Descriptions

Exception Signals

SysReset Input

System Reset is an active-low-input master processor reset signal that initializes the processor. The processor's optional features are established during the last cycle of reset, using the reset configuration mode inputs from ExcSInt(2:0).

ExcSInt(2:0) Input

Exception Synchronized Interrupt: These signals are the same as the RC3051 SInt(2:0) signals except for the Reset Configuration Modes.

Notes

Value	Description
‘11’	32-bit wide (non-interleaved) Boot Prom
‘10’	16-bit wide Boot Prom
‘01’	8-bit wide Boot Prom
‘00’	64-bit wide (interleaved 32-bit wide) Boot Prom

Table 17.2 Boot Prom Reset Configuration Modes for ExcSIntN(2:1) pins.

Note: The values of the Reset Initialization Vector for the Boot PROM are inverted relative to the internal size field in the configuration register of the Memory Controller.

LittleEndian

Use Little Endian Addressing: if asserted (active high), the processor will operate as a little-endian machine, and the RE bit of the status register would then allow user-mode big-endian tasks to operate in a little-endian system. If negated (inactive high), the processor will operate as a big-endian machine, and the RE bit will allow little-endian user-mode tasks to operate on a big-endian machine.

BootProm8

8-bit Boot PROM Mode: If asserted (active low), this mode will cause the Memory Controller to initialize memory chip selects, MemIoCs(0), sub-regions to 8-bit ports instead of 32-bit ports. Thus, an 8-bit boot PROM can be used to initialize the RC36100. If both BootProm8 and BootProm16 are asserted low, then the Memory Controller will initialize the Memory Chip select pair, MemIoCs(1:0), to use interleaved 32-bit ports. Table 17.2 shows the encoding of this bit at reset.

BootProm16

16-bit Boot PROM Mode: If asserted (active low), this mode will cause the Memory Controller to initialize memory chip select MemIoCs(0) to 16-bit ports instead of 32-bit ports. A 16-bit boot PROM can be used to initialize the RC36100. If both BootProm8 and BootProm16 are asserted low, then the Memory Controller will initialize the Memory Chip select pair, MemIoCs(1:0), to use interleaved 32-bit ports.

RISCore32 series Equivalent Modes

The RISCore32 series features a number of modes, which are selected at Reset time. Although most of those modes are irrelevant, a number of equivalences can be made:

- *IBlkSize = 4 word refill.*
- *DBlkSize = 1 or 4 word refill, depending on the DBlockRefill mode as selected in the CP0 Cache Configuration register.*
- *Reverse Endianness capability enabled.*
- *Instruction Streaming enabled.*
- *Partial Word Stores enabled.*

Other modes of the RISCore32 series pertain primarily to its cache interface, which is incorporated within the RC36100 and transparent to users of this processor.

Reset Behavior

While Reset is asserted, the processor maintains its interface in a state that allows the rest of the system to also be reset. Specifically:

- *SysClk operates at one-half the ClkIn frequency.*
- *SysData() is tri-stated*
- *SysALEn is driven de-asserted (high).*
- *Control signals are driven de-asserted (high).*
- *SysAddr() and SysDiag functions are driven (value undefined).*

Notes

The RC36100 samples for the negation of $\overline{\text{SysReset}}$ relative to a falling edge of $\overline{\text{SysClk}}$. On a rising edge of $\overline{\text{SysClk}}$, 6 cycles after the negation of $\overline{\text{SysReset}}$ is detected, the processor initiates a read request for the instruction located at the Reset Exception Address Vector. These cycles are a result of:

- *$\overline{\text{SysReset}}$ input synchronization performed by the CPU. The $\overline{\text{SysReset}}$ input uses special synchronization logic, thus allowing $\overline{\text{SysReset}}$ to be negated asynchronously to the processor. This synchronization logic introduces a two cycle delay between the external negation of $\overline{\text{SysReset}}$ and the negation of $\overline{\text{SysReset}}$ to the execution core.*
- *Internal clock cycles in which the execution core flushes its pipeline, before it attempts to read the exception vector.*
- *One additional cycle for the read request to propagate from the internal execution core to the read interface, as described in Chapter 8.*

Boot Software Requirements

Basic mode selection is performed using hardware during the reset sequence, as discussed in the mode initialization section. However, there are certain aspects of the boot sequence that must be performed by software.

The assertion and subsequent negation of reset forces the CPU to begin execution at the reset vector, which is physical address 0x1FC0_0000. This address resides in uncached, un-mapped memory, and thus does not require that the caches be initialized for the processor to execute boot code.

The processor must perform the following activities during boot:

- *Initialize the CP0 Status Register. The processor must be assured of having the kernel enabled to perform the boot sequence. Typically, a'mtc0 rx, CO_SR' instruction is one of the first few instructions in the boot sequence. Specifically, co-processor usable bits, and cache control bits, must be set to the desired value before any data references (cached or uncached), diagnostics or initialization occur.*
- *Initialize the CP0 Configuration Registers. The software should decide on the Cache Configuration, Port Sizes, and Bus Control during initialization.*
- Initialize the caches. *The processor must determine the sizes of the on-chip caches, and flush each entry, as discussed in Chapter 3. This must be done before the processor attempts to execute cacheable code.*
- Re-initialize CP0 Registers. *The processor should establish appropriate values in various CP0 registers, including:*
  - *The IM bits of the status register.*
  - *The BEV bit.*
  - *Initialize KUp/IEp so that user state can be entered using a RFE instruction.*
- Initialize on-chip memory and I/O controllers. *The boot software should establish the appropriate timing parameters, control options, timer values, PIO uses, etc., as appropriate to the particular system.*
  - *Note that the Serial Ports have a special reset initialization sequence. For more details, refer to the "Serial Port Initialization and General User's Notes" section in Chapter 15.*
- Enter User State.

Branch to the first user task and perform an RFE to enter the user mode.

Detailed Reset Timing Diagrams

The timing requirements of the processor reset sequence are illustrated below. The timing diagrams reference AC parameters whose values are contained in the RC36100 data sheet.

Reset Pulse Width

There are two parameters to be concerned with: the power on reset pulse width and the warm reset pulse width.

Notes

Mode Initialization Timing Requirements

The mode initialization vectors are sampled by an internal transparent latch, whose output enable is directly controlled by the SysReset input of the processor. The internal structure of the processor is illustrated in Figure 17.3. As illustrated in Figure 17.4, the mode vectors have a set-up and hold time with respect to the rising edge of SysReset.

Figure 17.1 illustrates the power on reset requirements of the RC36100 family. Figure 17.2 illustrates the warm reset requirements of the processor.

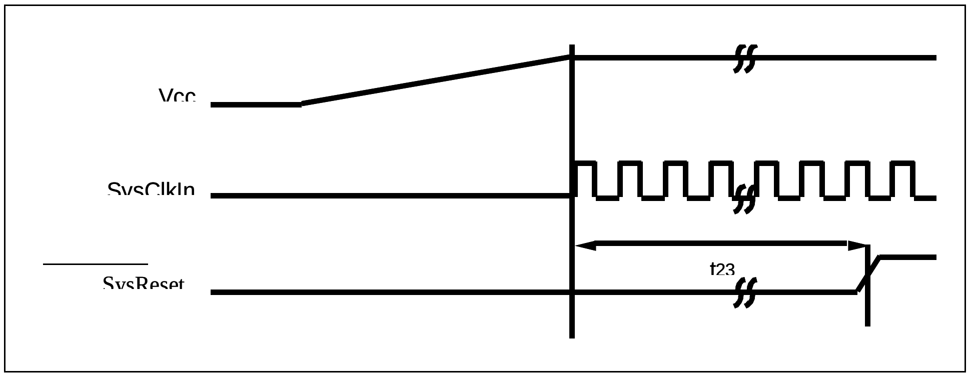


Figure 17.1 Cold Start

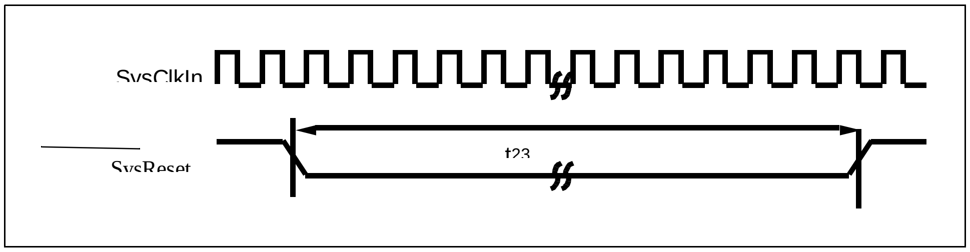


Figure 17.2 Warm Reset

Notes

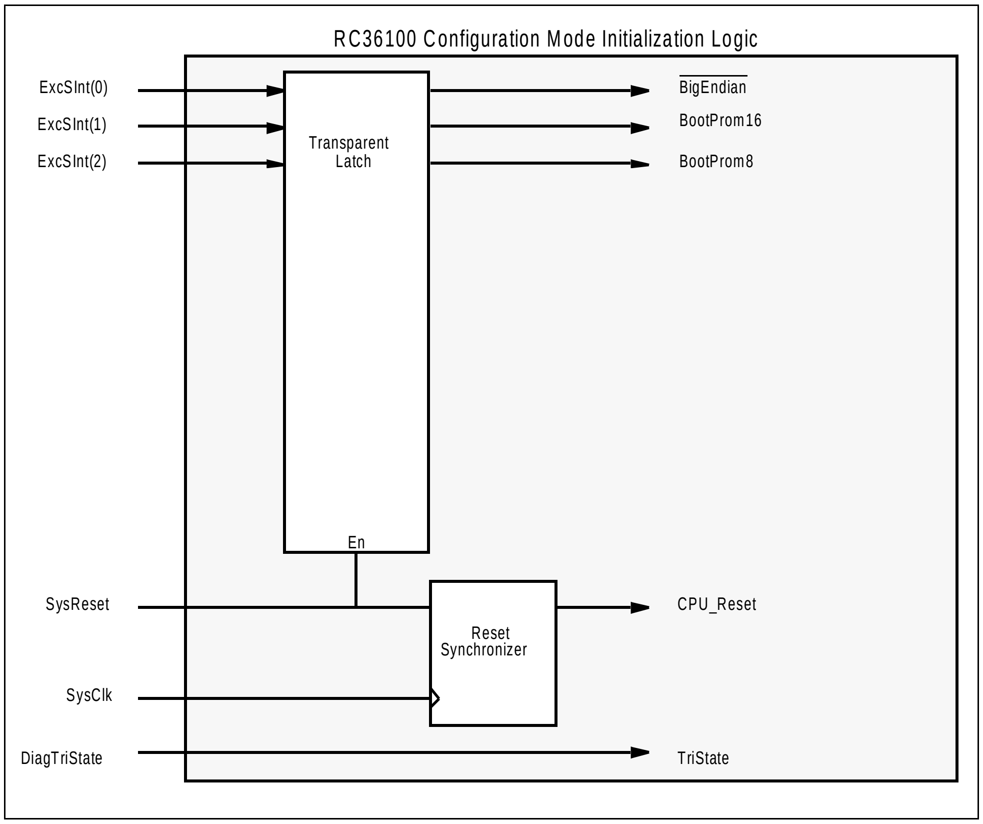


Figure 17.3 Configuration Mode Initialization Logic

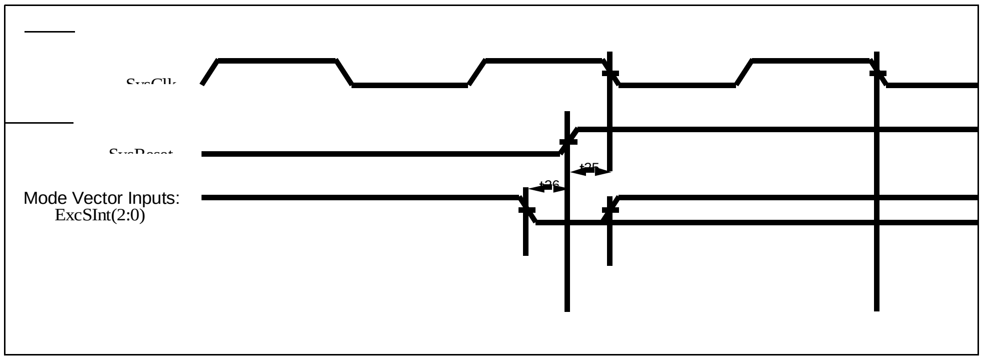


Figure 17.4 Mode Vector Timing

Reset Setup Time Requirements

The reset signal incorporates special synchronization logic that allows it to be driven from an asynchronous source. This allows the processor SysReset signal to be derived from a simple circuit, such as an RC network, with a time constant long enough to guarantee that the reset pulse width requirement is met.

Such a system should buffer the RC circuit such that a sufficiently fast monotonic rise time is generated which is capable of synchronously resetting any external state machines and logic at the same time as resetting the CPU.

The SysReset set-up time parameter can be thought of as the amount of time SysReset must be negated before the rising edge of SysClk, for guaranteed recognition. Failure to meet this requirement will delay the internal recognition of the end of reset by one clock cycle. This does not affect the timing of the sampling of the mode initialization vectors. Figure 17.5 illustrates the set-up time parameter of the RC36100.

Notes

ClkIn Requirements

The input clock timing requirements are illustrated in Figure 17.6. The system designer does not need to be explicitly aware of the timing relationship between ClkIn and $\overline{\text{SysClk}}$. Note that $\overline{\text{SysClk}}$ is driven even during the $\overline{\text{SysReset}}$ period as long as ClkIn is provided.

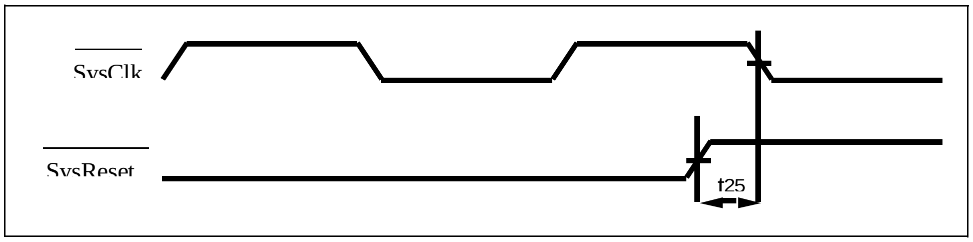


Figure 17.5 Reset Timing

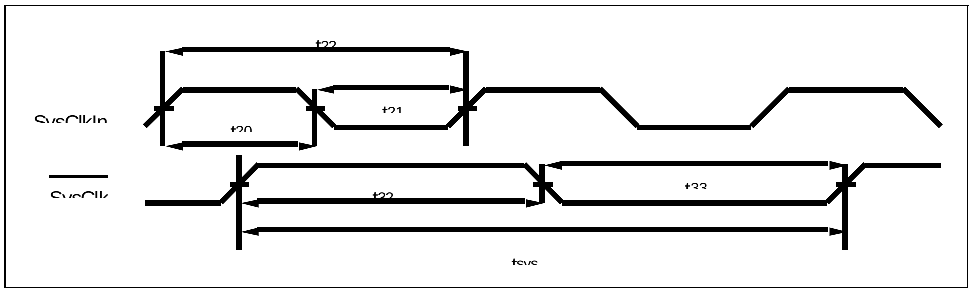


Figure 17.6 RC36100 Clocking

Debug Mode Features

Notes

Introduction

This chapter discusses features that have been included to facilitate the debugging of RC36100-based systems. These features are intended to be used by an in-circuit emulator, in-circuit tester, board-level tester, logic analyzer, a hardware modeler (or similar tool).

Features

- *Hardware trace/halt support, cache write suppression, and K0 preservation*
- *Cause register write option of the exception code bits (CP0)*
- *Instruction stepping support via virtual address debug trace watch register*
- *The ability of the processor to have instruction and data cache misses forced, thus allowing all internal cache accesses to be displayed on the bus interface.*
- *The ability to tri-state all output pins including SysClk, thus allowing an in-circuit emulator or tester to drive and control the output pins directly.*
- *The ability to deterministically set the phase relationship of the SysClk output relative to the SysClkIn input. This feature allows board level testers and hardware modelers to control the SysClk output.*
- *The ability to distinguish data and instruction accesses, allowing logic analyzers to do instruction disassembly.*
- *A software breakpoint instruction.*

Note: The features described in this chapter are intended for initial debug or production testing rather than for use in an end-user system.

The following are several debug/emulator hooks included in the normal functioning chip:

- *tri-stateable outputs*
- *tracepoint register*
- *extended CP0 cache configuration register*

Tri-Stateable Outputs

The tri-stateable outputs feature uses a dedicated input pin that if asserted causes all outputs (including SysClk) on the chip to tri-state. This feature is used in in-circuit manufacturing tests and by in-circuit emulators with non-socketed target CPUs.

Tracepoint Registers

The tracepoint registers consist of two memory mapped virtual address registers and a control register. When enabled through the control register, the tracepoint registers cause an exception when the virtual address register has the same value as the internal ALU stage Program Counter (PC).

When the exception occurs, a status/cause bit in the control register is set so that software can locate the exception's cause. Tracepoints in a delay slot will work, but they are not recommended. Tracepoints will set the BD CP0 Cause Register bit as expected; however, it would be up to the software to jump past the delay slot correctly (by subtracting 4 and re-executing the branch).

Extended CP0 Cache Configuration Register

The CP0 Cache Configuration Register, as described in an earlier section, contains several software controllable Force Cache Miss features that allow logic analyzers to interface to the RC36100.

Cause and EPC Register Writes

The RC36100 adds a control bit, which if asserted, enables writes to the CP0 Cause register cause field and the CP0 EPC.

Notes

Features specific to debug/emulators

It is envisioned that operating system debug kernels always echo MTC0 writes. Addresses are reserved in a scheme that frames the following registers:

- *32 General Purpose Registers.*
- *32 CP0 Registers (only 16 presently used for RISCore3000 family systems).*
- *64 CP1 Floating Point Registers (presently, only 32 are used for RISCore3000 family systems).*

Physical Address	Register
FFFF_8F68	K0 $26
FFFF_8F6C	K1 $27
FFFF_8F8C	CP0 $3 Config

Table 18.1 Reserved Emulator Addresses.

Pin Descriptions

Debug/Emulator and Diagnostic Signals

$\overline{\text{DiagCache/UnCache}}$ Output

Diagnostic Cached versus Uncached and Burst Miss Address 3: An output signal specifying cacheability type attribute of external system bus transactions. Signal is low during Uncached references, high during Cached ones. During the second clock of burst reads, outputs the miss address 3. The first and remainder clocks output cached versus uncached.

DiagInst/$\overline{\text{Data}}$ Output

Diagnostic Instruction versus Data Status and Burst Miss Address 2: An output signal specifying data type attribute of external system bus transactions. Signal is high during Instruction references, low during Data references.

During the second clock of burst reads, outputs the miss address 2. The first and remainder clocks output instruction versus data. Internal DMA transactions are always data transactions.

$\overline{\text{DiagRun}}$ Output

Diagnostic Run: A pseudo-synchronized active low output version of the internal CPU core RunN signal.

$\overline{\text{DiagBranchTaken}}$ Output

Diagnostic Branch Taken: A pseudo-synchronized active low output signal indicating when a branch is taken (same as the RC3041A).

$\overline{\text{DiagJRorExe}}$ Output

Diagnostic Jump Register or Exception: A pseudo-synchronized active low output signal indicating when a jump register instruction is executed or an exception is taken. $\overline{\text{DiagJorExe}}$ must either be externally gated with $\overline{\text{DiagRun}}$ and $\overline{\text{DiagBranchTaken}}$ or have pre-initialized all instruction cache data fields.

$\overline{\text{DiagInternalWr}}$ Output

Diagnostic Internal Resource Write: An active low output signal indicating that an MTC0 instruction to register 3 was executed. This signal is used to indicate to the debug/emulator that it may want to interrogate the RC36100 to find out if a control register that may have an effect on debug/emulator interpretation was altered.

Notes

DiagTriState **Input**

Diagnostic Tri-State: An input signal that when asserted low causes all outputs to tri-state. Can be used to:

1. Disable target board CPU during emulation.
2. Disable CPU during in-circuit manufacturing testing.

DiagInstCacheWrDis **Input**

Diagnostic Instruction Cache Write Disable: An active low input signal that disables instruction cache misses from updating the instruction cache. Meant to be asserted after DiagFICM and an instruction miss.

DiagFCM **Input**

Diagnostic Force Instruction and Data Cache Miss: An active low input signal causing all instructions and data loads (except internal partial word store reads) to miss the cache and do an external system bus read. In this mode no newly initiated read cache misses are written into the cache. During the assertion of DiagFCM, internal generation of 'AckN' is delayed on burst reads until after the bus transaction completes.

Note: Although emulators typically assert this pin during functional operation, the non-emulator user should either assert or not assert this pin during power-up and continuously leave it asserted or not asserted.

DiagIntDis **Input**

Diagnostic Interrupt Disable: An active low input signal when asserted, causes all external and internal interrupts to be disabled.

DiagNoCS **Output**

Diagnostic No Chip Select: An active low input signal concurrently asserted with SysALEn indicating that no external chip select was activated for this read or write. On subsequent bus clocks, after SysALEn asserts, active low indicates that an internal chip select has been activated.

DiagInternalDmaBusGnt **Output**

Diagnostic Internal DMA Channel Bus Grant: An active low output signal asserted whenever one of the four internal DMA channels receives a bus grant. This signal can be gated with a peripheral chip select to distinguish between a peripheral control register access versus a DMA access.

Register Descriptions

Note that Big Endian software must offset these addresses by b'10 (0x2), if halfword accesses are used.

Phys. Address	Description
0xFFFF_E500	Tracepoint Control Register
0xFFFF_E504	Debug Control Register
0xFFFF_E510	TraceLSB(0) Address Register
0xFFFF_E514	TraceMSB(0) Address Register
0xFFFF_E520	TraceLSB(1) Address Register
0xFFFF_E524	TraceMSB(1) Address Register

Table 18.2 Debug Interface Register Address Assignments

Notes

MSB Debug Tracepoint Address Register LSB Debug Tracepoint Address Register ('DebugTraceAddrReg')

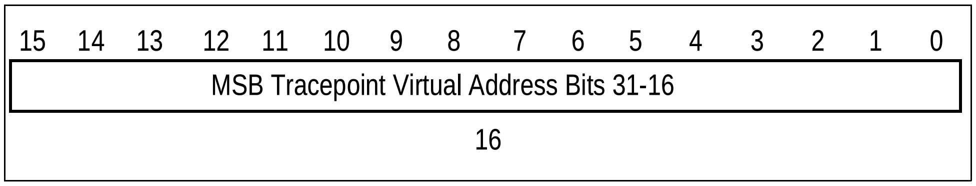


Figure 18.1 Debug Tracepoint Address Register ('MSB DebugTraceAddrReg').

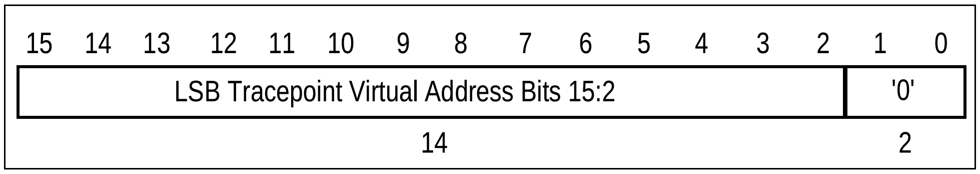


Figure 18.2 Debug Tracepoint Address Register ('LSB DebugTraceAddrReg').

When the Debug Tracepoint Virtual Address Register MSB and LSB matches the internal program counter (ALU pipeline stage), and the feature is enabled via the debug tracepoint control register, an exception is taken and a debug tracepoint control register cause bit is set.

Bit	Description
15:2	Tracepoint Virtual Address
1:0	Reserved '0'

Table 18.3 Debug Tracepoint Address Register ('DebugTraceAddrReg') Bit Assignments.

Debug Tracepoint Control Register ('DebugTraceControlReg')

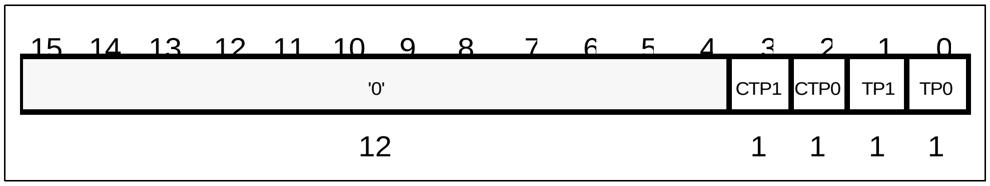


Figure 18.3 Debug Tracepoint Control Register ('DebugTraceControlReg').

The Debug Tracepoint Control Register is used to access and control tracepoint and single step functions.

Notes

Bit	Description
15:4	Reserved ‘0’
3	CTP1
2	CTP0
1	TP1
0	TP0

Table 18.4 Table Debug Tracepoint Control Register (‘DebugTraceControlReg’) Bit Assignments.

Reserved Low (‘0’) Field:

Must be written to ‘0’ for future compatibility. Value when read is undefined.

Cause is Tracepoint (‘CTP’) Field:

After getting an exception, if the CTP field is found to be an active ‘1’, the exception handler should acknowledge the exception by writing a ‘0’ to the CTP bit. There are two fields: branch taken/ branch not taken‘.

Value	Action
‘1’	Cause of exception is Tracepoint.
‘0’	Cause of exception is not Tracepoint (default).

Table 18.5 Cause is a Tracepoint (‘CTP’) Field Encoding.

Tracepoint (‘TP’) Field:

Value	Action
‘1’	Tracepoint On. Forces CPU to allow tracepoint register to activate if the Program Counter matches the Tracepoint Address Register.
‘0’	Tracepoint Off (default).

Table 18.6 Tracepoint Enable (‘TP’) Field Encoding.

Debug Control Register (‘DebugControlReg’)

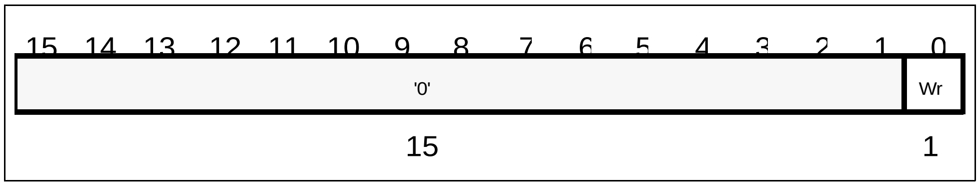


Figure 18.4 Debug Control Register (‘DebugControlReg’).

Notes

Bit	Description
15:1	Reserved ‘0’
0	Wr

Table 18.7 Debug Control Register (‘DebugControlReg’) Bit Assignments.

Reserved Low (‘0’) Field:

Must be written to ‘0’ for future compatibility. Value when read is undefined.

Writability (‘Wr’) Field:

Value	Action
‘1’	Allow CP0 Cause Bits and EPC Register to be written.
‘0’	CP0 Cause Bits and EPC Register are read only.

Table 18.8 Writability (‘Wr’) Field Encoding.

Initializing SysClk for Test

Another feature for board level testing is the ability to initialize the phase of SysClk to its high phase. A low to high transition on Reset will cause the internally synchronized (delay of less than or equal to 2 clocks) version of Reset to always set SysClk high during its next phase. Thus the state of SysClk can be deterministically controlled within a known number of ClkIn transitions. The two cases are shown in Figure 18.5 and in Figure 18.6.

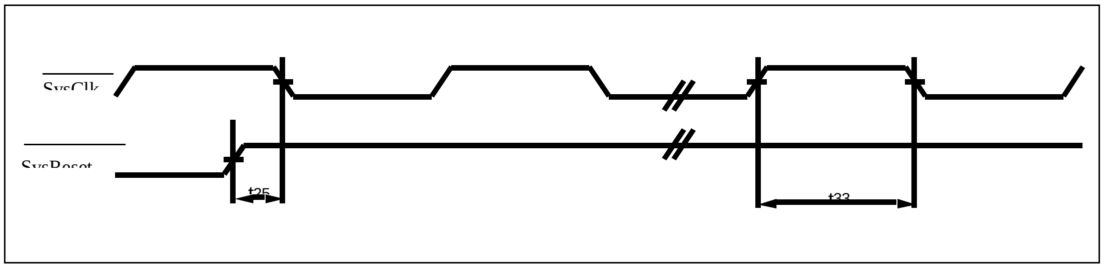


Figure 18.5 RC36100 SysClk Phase Initialization Case A

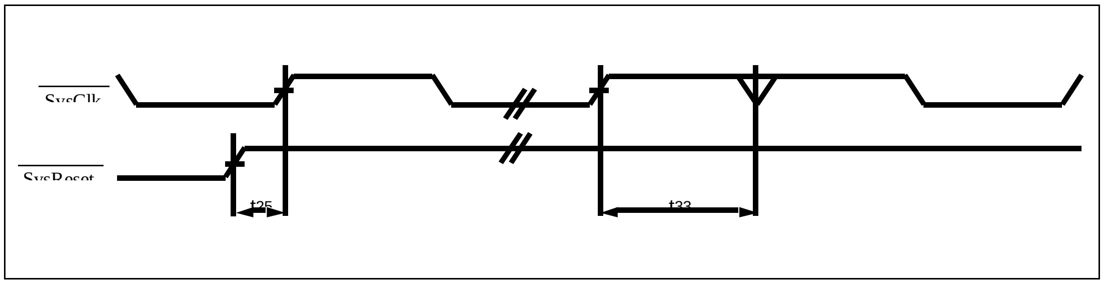


Figure 18.6 RC36100 SysClk Phase Initialization Case B

Using Diag for Instruction Disassembly

The RC36100 provides a Diagnostic pin which during its data phase outputs whether a read transaction is the result of an instruction fetch or the result of a data fetch. This information is independent of the information given during the address phase of whether or not the read was a result of a cached or uncached read. Note that this pin is undefined on writes; however, by necessity all writes must be data writes.

Notes

Other Considerations

- *SysAddr Changes in the last clock of DRAM Burst Writes.*

  During DRAM Burst writes, the SysAddr may change in the last clock of the burst. Because the DRAM address is actually latched on the asserting edge of $\overline{\text{CAS}}$, this does not present a functional failure, and users can ignore this condition. However, emulators and Logic Analyzers as well as test vector samplers for board testing may need this information as far as when to latch the display address.
- *SysAddr is late on unused bus cycles.*

  The SysAddr bus switches very late in the clock during unused bus cycles, specifically 1 clock before $\overline{\text{SysALE}}$ asserts. Most users can ignore the switches. However, during board testing, test vector samplers should avoid sampling the address during this clock.

Notes

Index

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes