Design of Coarse-Grained Reconfigurable Architecture for Digital Signal Processing

Implementation Aspects

Chenxin Zhang

Master of Science Thesis
February 2009
Department of Electrical and Information Technology, Lund University
Supervisors: Thomas Lenart and Henrik Svensson

Abstract

Reconfigurable computing is an emerging trend for embedded system design. With the use of platform containing a reconfigurable architecture, it is possible to accelerate arbitrary algorithms that are executing on an embedded system. To achieve high performance to a feasible hardware cost, the reconfigurable architecture should be a trade-off between efficiency and flexibility.

This thesis discusses design and implementation of the coarse-grained reconfigurable architecture targeting for digital signal processing applications. The proposed reconfigurable architecture is constructed from a mesh of resource cells, divided into processing and memory cells, which communicate using a combination of local interconnections and a global hierarchical routing network. The processing cell can further be distinguished from a generic RISC processor and a CORDIC cell. High performance local interconnections generate a high communication bandwidth between neighboring cells, while the global network provides flexibility and access to external modules.

All the cell modules developed in the reconfigurable architecture are design-time configurable, where different hardware structure can be generated depending on the user requests. Besides, the processing and memory cells are run-time reconfigurable to enable flexible application mapping.

A 4-by-2 reconfigurable cell array containing four 16/32-bit RISC processor cells, three smart memory cells and one configurable CORDIC cell has been designed and implemented in HDL, and has been eventually integrated as a coprocessor into an embedded system. Applications of a time-multiplexed FIR filter and a 32~1,024-point time-multiplexed radix-2^2 FFT have been manually mapped onto the constructed cell array and have been verified on an FPGA platform, the Virtex-II Pro-30-7ff896 from Xilinx. It is shown that the reconfiguration code size for the mapped FFT implementation on the cell array outperforms ordinary DSP processors by a factor of 8, and the number of used clock cycles is reduced with ~20%.

Contents

Acknowledgements

First of all, I am grateful to Associated Professor Viktor Öwall and the EIT department at LTH, for providing me the great opportunity to do my master thesis. It has indeed been a pleasant and challenging time from which I will have many good memories.

Most importantly, I would like to express my sincere gratitude to my supervisor PhD Thomas Lenart. He has not only guided me stepping into this fantastic research area, but also provided me a strong basis and all possible supports ever since the start of the project. I appreciate his altruistic sharing of resources and his attitude on devoting himself into research field. For many of the project meetings, he has to drive after a whole days' work from Copenhagen all the way back to Lund to make discussions with me at late evening. I would like to extend my gratitude to Viktor Öwall, for devoting his treasure time in having discussions and providing valuable comments to me on a day-to-day basis. I would also like to show my appreciation to Viktor Öwall and Thomas Lenart both for their help and advice during the writing process of this thesis. Further, I am thankful to assistant supervisor Henrik Svensson, for having fruitful discussions with me during his very busy period documenting his own PhD thesis.

The friendly and pleasant working atmosphere in the EIT department has played an important part in making my master study an enjoyable experience. I am grateful to Peter Nilsson, Joachim Neves Rodrigues, Johan Löfgren, Deepak Dasalukunte, Isael Diaz, Stefan Molund, Erik Jonsson and all other members in the department for their friendliness and helpfulness. The project would not have been possible without their helps. Moreover, the generous electrical and mechanical tool support had facilitated my work a lot. I feel especially thankful for the extraordinary disc space allotment.

Lund, February, 2009

Chenxin Zhang

List of Acronyms

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Set Processor
BIST	Built-In Self-Test
CGRA	Coarse-Grained Reconfigurable Architecture
CORDIC	Coordinate Rotation Digital Computer
DFT	Discrete Fourier Transform
DIF	Decimation-In-Frequency
DIT	Decimation-In-Time
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FGRA	Fine-Grained Reconfigurable Architecture
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HPC	High-Performance Computing
I/O	Input-Output
ILC	Inner Loop Controller
IP	Intellectual Property
JTAG	Joint Test Action Group
LSB	Least Significant Bit
LUT	Lookup Table
MAC	Multiply-Accumulate
MC	Memory Cell
MIPS	Microprocessor without Interlocked Pipeline Stages
MSB	Most Significant Bit
PC	Processor Cell
PGM	Program Memory
PLB	Processor Local Bus
PPC	Power PC
RAM	Random Access Memory
RC	Resource Cell
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTL	Register Transfer Level
RTR	Run-time reconfiguration
SCENIC	SystemC Environment with Interactive Control
SDF	Single-path Delay Feedback

SQNR	Signal-to-Quantization-Noise Ratio
SRAM	Static Random-Access Memory
TMFIR	Time Multiplexed Finite Impulse Response
UART	Universal Asynchronous Receiver/Transmitter
VHDL	Very High Speed Integrated Circuit (VHSIC) HDL

1 Introduction

The use of application specific hardware accelerators (ASICs) is a well known approach for achieving real-time performance within the budget for physical size and energy dissipation. However, these circuits require a rather long system development time and exhaustive testing procedures both before and after the chip fabrication. Besides, ASICs are by nature tailored for specific applications, they are less flexible and can hardly be reused by other designs. The fine-grained reconfigurable architectures (FGRAs), i.e. FPGA, are field reconfigurable. With existing design IP cores, designers can sometimes gain system development time. However, the use of FGPAs requires bit level manipulations during system design time, which might not be the needs by initial requirements. Moreover, FGRAs expose a huge routing area overhead and poor routing ability. Due to these reasons, coarse-grained reconfigurable architectures (CGRAs) become a popular choice in many real applications in both industry and academia.

Emphases in this thesis are placed on discussions of hardware design and performance evaluations for the proposed CGRA in application of digital signal processing, whereas the topic on systematic analysis of the coarse-grained reconfigurable architecture are not a subject of this project. The proposed CGRA is constructed from an array of resources cells which communicate using local interconnections and a global hierarchical network. Resource cell is the common name for all types of functional units, which are divided into processing and memory cells. All the resource cells are dynamically reconfigurable to support run-time mapping of arbitrary applications. Each of the individual cells has been implemented in a hardware description language (HDL) and synthesized based on an FPGA platform to obtain performance and area metrics. By integrating different resource cells together, a 4-by-2 reconfigurable cell array containing four 16/32-bit RISC processor cells, three smart memory cells and one configurable CORDIC cell has been constructed and verified on an FPGA evaluation board, where applications of a time-multiplexed FIR filter and a 32~1,024-point time-multiplexed radix-2^2 FFT have been experimented.

Chapter 2 briefly describes the basic concepts of the CGRA and network communications inside the system. Basic structure of the proposed reconfigurable architecture is introduced, which is constructed from a tile of resource cells and a communication network. Detailed descriptions on each of the individual resource cell implementations, including the processor, CORDIC, memory and router cells, are presented in Chapter 3, 4, 5 and 6, respectively. In Chapter 7, mapping of a time-multiplexed FIR filter on the CGRA is described, where the basic functionalities in each resource cell are verified and network communications are tested. In Chapter 8, design of a 4-by-2 cell array and an FPGA based embedded system with the experimented time-multiplexed radix-2^2 FFT application is comprehensively described, and different algorithm mapping alternatives are discussed and evaluated.

2 Coarse-grained reconfigurable architecture

Reconfigurable hardware architectures are emerging as a suitable and feasible approach to achieve high performance combined with flexibility and programmability. Compared to conventional fine-grained architectures, coarse-grained architectures trade mapping flexibility to reduce reconfiguration time and achieve higher performance using word-level data processing. In this chapter, a coarse-grained dynamically reconfigurable architecture (CGRA) is briefly introduced. All the concepts and basic structure of the CGRA are presented based on the exhaustive pre-studies that have been carried out in previous work [1] and [2], and descriptions in this chapter are collections of the essential points from [1] and [3].

2.1 Reconfigurable architecture

In contrast to programmable architectures, the reconfigurable architectures enable hardware programmability. It means that not only the software that runs on a platform is modified, but also how the architecture operates and communicates. Hence, an application is accelerated by allocating a set of required processing, memory and routing resource to adapt to the current operational and processing conditions.

Reconfigurable architectures provide numerous advantages over traditional application-specific hardware accelerators, such as resource sharing to provide more functionality than there is physical hardware. Hence, currently inactivated functional units do not occupy any physical resources, which are instead dynamically configured during run-time. Another advantage is that a reconfigurable architecture may enable mapping of future functionality without additional hardware or manufacturing costs, which could also extend the lifetime of the platform.

The size of the reconfigurable elements is referred to as the *granularity* of the device. Fine-grained devices are usually based on small look-up tables (LUT) to enable bit-level manipulations. These devices are extremely versatile and can be used to map virtually any algorithm. However, fine-grained architectures are inefficient in terms of hardware utilization of logic and routing resources. In contrast, coarse-grained architectures use building blocks in a size ranging from arithmetic logic units (ALU) to full-scale processors. This yields a higher performance when constructing standard data-paths, since the arithmetic units are constructed more efficiently, but the device becomes less versatile. The properties of fine-grained and coarse-grained architectures are summarized in Table 2-1.

Coarse-grained reconfigurable architectures are arrays constructed from larger computational elements, usually in the size of ALUs or smaller programmable kernels and state-machines. The computational elements communicate using a routing network, as illustrated in Figure 2-1 for an example of such a structure. In this way, the coarse-grained architecture requires less configuration data, which improves the reconfiguration time, while the routing resources generate a lower hardware overhead.

Table 2-1. A comparison between fine-grained and coarse-grained architectures. Development time and design specification refer to applications running on the platform.

Properties	Fine-grained	Coarse-grained
Granularity	Bit-level (LUT)	Word-level (ALU)
Versatility/Flexibility	High	Medium/High
Performance	Medium	High
Interconnection overhead	Large	Small
Reconfiguration time	Long (ms)	Short (μs)
Development time	Long	Medium
Design specification	Hardware	Software
Application domain	Prototyping, HPC	RTR, HPC

Note: HPC – High-performance Computing; RTR – Run-time reconfiguration.

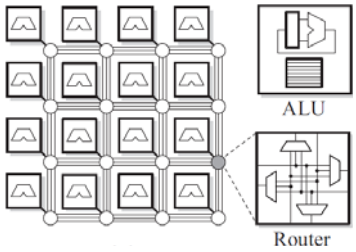


Figure 2-1. An example of a coarse-grained reconfigurable architecture, with an array of processing elements (ALU) and a routing network.

In contrast to a fine-grained FPGA, course-grained architectures are designed for partial and run-time reconfiguration. This is an important aspect due to situations when hardware acceleration is required for short time durations or only during the device initialization phase. Instead of developing an application-specific hardware accelerator for each individual situation, a reconfigurable architecture may be reused to accelerate arbitrary algorithms. Once the execution of one algorithm completes, the architecture is reconfigured for other tasks.

The possibility to support algorithmic scaling is also an important aspect. Algorithmic scaling means that an algorithm can be mapped to the reconfigurable array in multiple ways, which could be a trade-off between processing performance and area requirements. A library containing different algorithm mappings would enable the programmer or the mapping tool to select a suitable architecture for each situation. A low complexity algorithm mapping may be suitable for non-critical processing, while a parallel mapping may be required for high performance computing.

From an algorithm development perspective, the coarse-grained architectures differ considerably from the design methodology used for FPGA development. While FPGAs use a hardware-centric methodology to map functionality into gates, the coarse-grained architectures enable a more software-centric and high-level approach. Hence, it allows hardware accelerators to be developed on-demand, and potentially in

the same language used for software development. Unified programming environments enhance productivity by simplifying system integration and verification.

2.2 Coarse-grained reconfigurable architecture

The proposed coarse-grained reconfigurable architecture is constructed from a tile of $W \times H$ resource cells and a communication network, as shown in Figure 2-2. *Resource cell* (RC) is the common name for all types of functional units, which are divided into *processing cells* (PC) and *memory cells* (MC). Processing cells implement the processing functionality to map applications to the CGRA, while memory cells are used to store data tables and intermediate results during processing. Depending on the different computational demands, three types of processing cells are provided in the CGRA: a 32-bit DSP processor with radix-2 butterfly support, a 16-bit MAC processor with multiplication support, and a configurable CORDIC processor for advanced function evaluation. The DSP and MAC processors are based on a similar architecture, with a customized instruction set. Memory cells contain a number of memory banks that each can be configured to emulate FIFO functionality as well as supporting random memory access. All the resource cells are dynamically reconfigurable to support run-time mapping of arbitrary applications. Detailed architecture descriptions of the processor, memory and CORDIC cells are presented in Chapter 3, 4 and 5, respectively.

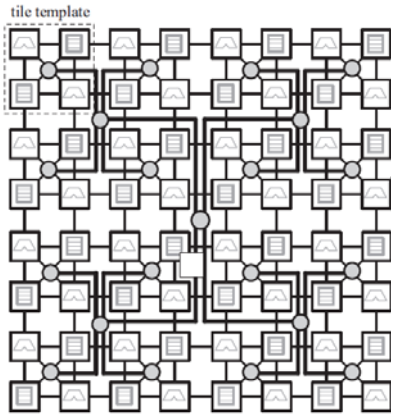


Figure 2-2. Proposed architecture with an array of processing and memory cells, connected using a local and a global hierarchical routing network. $W = H = 8$.

An array of resource cells is constructed from a *tile template*. A tile template is user-defined and contains the pattern in which processing and memory cells are distributed over the array. For example, the architecture presented in Figure 2-2 is based on a tile template of size 2×2, with two processing cells and two memory cells. The template is replicated to construct an array of arbitrary size.

The resource cells communicate over local interconnections and a global hierarchical network, as illustrated in Figure 2-3 (a) and (b), respectively. The local network with dedicated wires provides high communication bandwidth between

neighboring resource cells. Hence, it is assumed that the main part of the total communication is between neighboring cells and through local interconnections. The global network provides communication flexibility to allow any two resource cells in the array to communicate. However, transmitting data packets over the global network suffers from long initial communication latency compared with local data transmissions. This issue is further discussed in Chapter 6.

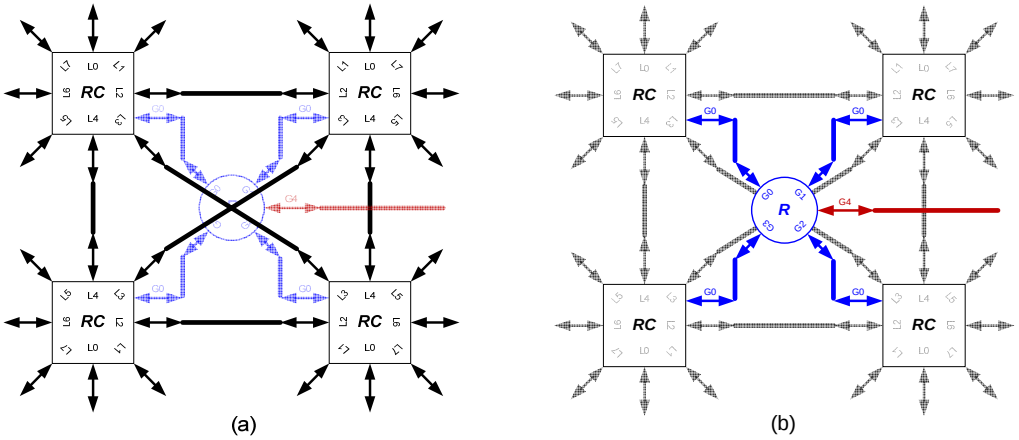


Figure 2-3. (a) Local interconnections. (b) Router cell with global routing network.

2.3 Local and global communication

2.3.1 Communication protocol

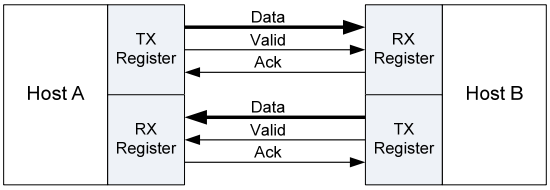


Figure 2-4. Data communication with flow control in the CGRA.

Basic hardware connections for data communications in the CGRA are shown in Figure 2-4. Communication ports are bi-directional, where each port consists of one TX line and one RX line, and flow control is used in all data transactions to avoid overflow [4] and underrun [5].

Flow control is implemented with a valid bit toggled by the data sender, indicating there is one data package available on the line, and an acknowledge bit fed back from data receiver, indicating data package has been accepted. Obviously, two intrinsic steps are involved in each data transmission, referred to as two-phase protocol, and hence requires at least two clock cycles. This is considered as inefficient in the CGRA, as the flow control overhead degrades data communication throughput and will certainly be a bottleneck in applications like data streaming.

By exploring the hardware setup, I/O port registers in both transmission terminals are not utilized efficiently in the original two-phase communication protocol, where two registers are utilized as transparent data path during each transaction, resulting in transmission delays. To address this problem, the two-phase communication protocol is revised with inspiration got from the FIFO operation scheme. The idea is to use I/O port registers in both terminals as transmission buffers, where registers are always writable as long as there is free space available. The transmission line is only suspended if both buffers are full, and transmitter has more data to send and no responding from the receiver side.

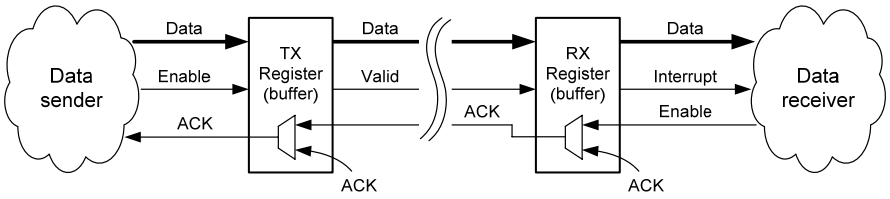


Figure 2-5. Hardware setup for FIFO like two-phase communication protocol.

Hardware setup for the FIFO like two-phase communication protocol is shown in Figure 2-5. The acknowledgement (ACK) signal towards data sending side has two responders. When data is written into an empty transmission buffer, packet is automatically acknowledged by the control logics inside that buffer; otherwise the ACK signal is switched to listen to the succeeding data receiving side when transmission buffer is full. Hence, the ACK signal can be used to reflect the status of the buffers in a transmission channel. This provides that the data transmitter can send at least two packets before getting suspended by a "silent" data receiver, if both transmission buffers are initially empty. In case when both communication terminals are synchronized in packet sending and receiving, data transmissions can be carried out in every clock cycle, thereby overcomes the communication overhead problem explored in the original two-phase protocol. Figure 2-6 illustrates the timing diagram of this revised communication protocol.

(1) Assume there is no initial data transmission since system started, the ACK signal is initialized to state high. Data sender starts transmitting data by pushing a packet into the transmission buffer at the rising edge of the clock.

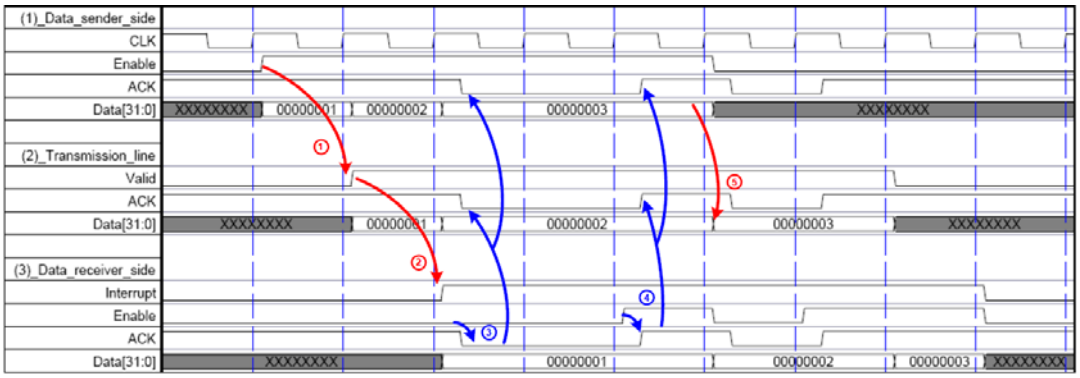


Figure 2-6. Timing diagram of the FIFO like two-phase protocol.

(2) In consecutive clock cycle, the first data packet is accepted by TX buffer, and an acknowledgement signal is automatically sent back to the data sender. When TX buffer is full, data receiving for the following packets will be determined by the remote RX side.
(3) In the third clock cycle, the first data packet is taken by RX buffer, and the automatically responded ACK signal allows TX buffer to keep receiving data from its data sender. Thereafter, both TX and RX buffers are full. So if data sender has more data to send, responses from the data receiver will determine the states for the following transactions. In this example, data receiver does not want to take any data packet at the moment, so all three ACK signals are pulled down, indicating that transmission buffers are full and a communication TX stall is asserted to the data sender.
(4) Transmission line is unfrozen along with the first data packet being accepted by the data receiver.
(5) The following data packets stored in transmission buffers are shifted towards the RX side in consecutive clock cycles. Again, depending on the response of the data receiver, the whole transmission line will either be activated or refrozen. In case the data sender has no more packets to send, communication TX stall will not be asserted anymore, and both transmission buffers will be gradually freed when stored data packets are consumed by the receiving terminal.

2.3.2 Network packets

The routers forward *network packets* over the global connections. A network packet is a carrier of data and control information from a source to a destination cell, or between resource cells and an external host. A data stream is a set of network packets, and each individual packet is send as a network *flow control digit* (flit). A flit is an atomic element that is transferred as a single word on the network, as illustrated in Figure 2-7.

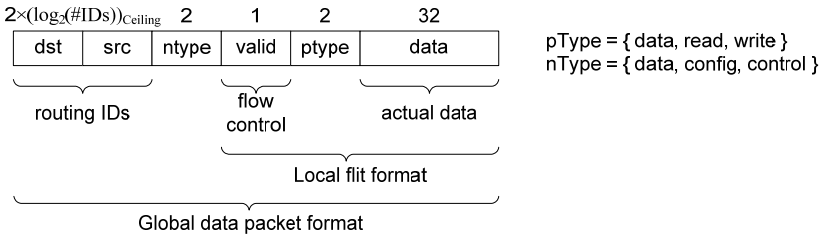


Figure 2-7. Network packet format of local and global data transmission.

A flit consists of a 32-bit payload and a 2-bit payload type identifier to indicate if the flit contains data, a read request, or a write request. For global routing, unique identification numbers are required and an additional 2-bit network type identifier indicates if the packet carries data, configuration, or control information. *Configuration* packets contain a functional description on how the resource cells

should be configured, as will be further described in each of the resource cell chapters. *Control* packets are used to notify the host processor of the current processing status, and are reserved to exchange flow control information between resource cells.

2.3.3 Network routing

Each resource cell allocates one or more network identifiers (ID), which are integer numbers to uniquely identify a resource cell in the CGRA, as shown in Figure 2-8 (a).

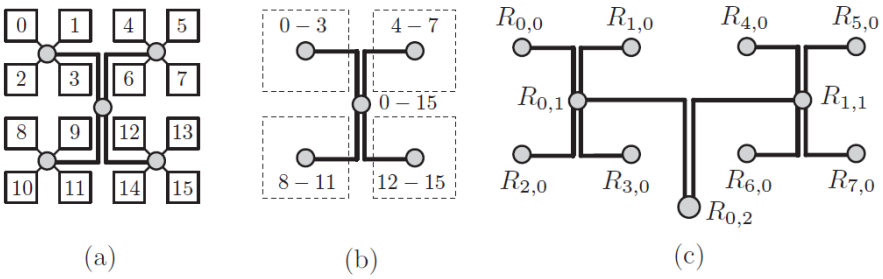


Figure 2-8. (a) Recursively assignment of network IDs. (b) A range of consecutive network IDs are assigned to each router table. (c) Hierarchical router naming as $R_{index,level}$.

A static routing table is stored inside the router cell and used to direct traffic over the network. At design-time, network IDs and routing tables are recursively assigned for resource cells by traversing the global network from the top router. Recursive assignment results in that each entry in the routing table for a router $R_{i,l}$, where i is the router index number and l is the router hierarchical level as defined in Figure 2-8 (c), is a continuous range of network IDs as illustrated in Figure 2-8 (b). Hence, network ID ranges are represented with a base address and a high address. How the routing table is utilized in data transactions is further discussed in Chapter 6.

A link from a router $R_{i,l}$ to a router $R_{j,l+1}$ is referred to as an *uplink*. Any packet received by router R is forwarded to the uplink router if the packets network ID is not found in the router table. A router may only have a single uplink port, else the communication path could become non-deterministic. A hierarchical view of the router interconnects is illustrated in Figure 2-9.

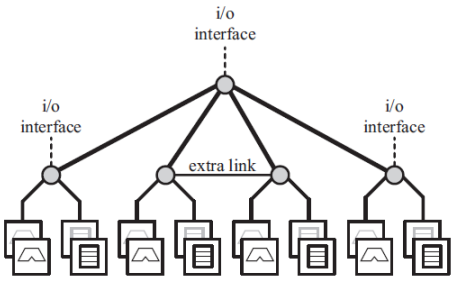


Figure 2-9. Hierarchical view of the router interconnects and external interfaces.

2.3.4 Network capacity

When the size of a CGRA increases, the global communication network is likely to handle more traffic, which requires network enhancements. A solution to improve the communication bandwidth is to increase the network capacity in the communication links, as shown in Figure 2-10 (a). Since routers on a higher hierarchical level could become potential bottlenecks to the system, these routers and router links are candidates for network link capacity scaling. Thus, this means that a *single link* between two routers is replaced by *parallel links* to improve the network capacity. A drawback of this approach is the increased complexity, since a more advanced router decision unit is required to avoid packet reordering. Otherwise, if packets from the same stream are divided onto different parallel links, this might result in that each of the individual packets arrive out-of-order at the destination.

Another way to improve the communication bandwidth is to insert additional network paths to avoid routing congestion in higher level routers, referred to as *network balancing*. Figure 2-10 (b) shows an example where all $R_{i,1}$ routers are connected to lower the network traffic through the top router. Additional links may be inserted between routers as long as the routing table in each network router is deterministic. When a network link is created between two routers, the destination router's reachable IDs are inserted in the routing table for the source router.

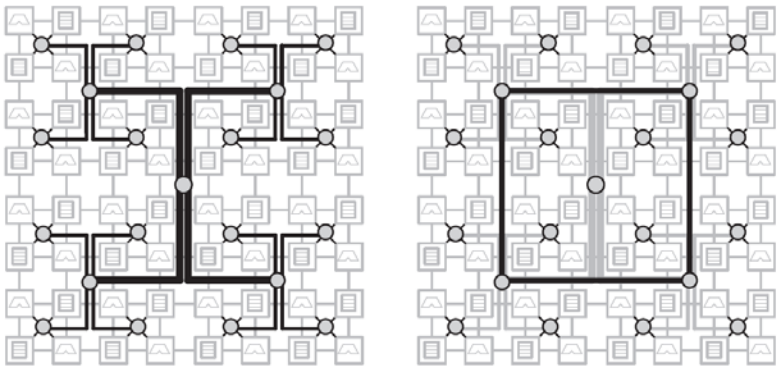


Figure 2-10. (a) Enhancing the router capacity when the hierarchical level increases. (b) Enhancing network capacity by connecting routers at the same hierarchical level.

3 Processor cell architecture

The processor cell is one of the main building blocks in the CGRA, which is used to handle computational operations for the mapped applications. In addition, it can potentially be used to control the operations of surrounding cells. Some of the hardware resources in the processor cell are configurable during system design-time and a few run-time control possibilities are provided for dynamic reconfigurations.

Hardware implementation of the processor cell was initiated and constructed in a previous work [3], where the basic cell architecture and functionalities have been comprehensively described. In this project, the processor instruction set has been extended based on the needs for intended DSP applications, and a few architectural limitations have been explored and fixed during system developments.

3.1 Cell architecture

The processor cell is similar to conventional RISC core, which contains a program memory, general purpose registers (GPR), and organized into pipeline stages. In contrast, data memory is not located inside the processor cell, but can be reached by connecting one or more external memory cells, either through the local or global routing network. Bidirectional communication I/O ports are provided for exchanging data with surrounding cells, and the I/O port registers can be accessed in the same way as the GPRs. Figure 3-1 shows the block diagram of a processor cell. Due to the absence of internal data memory, processor pipeline consists of four stages: instruction fetch (IF), instruction decode (ID), execution (EXE) and write back (WB). A more detailed architectural schematic of the processor cell is presented in Figure-Appendix 1.

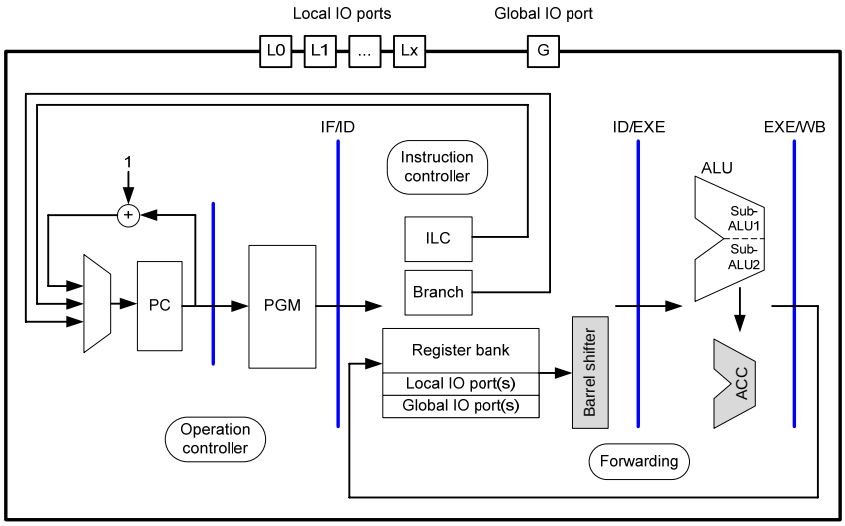


Figure 3-1. Internal building blocks in a processor cell. Optional function modules are shaded in gray.

3.1.1 Design-time architectural configuration

Configurability is one of the features in the processor cell design. The basic architecture can be configured with more advanced features during system compilation time, such as the processor data bus width, the number of I/O ports, barrel shifter and so on. This flexibility allows the user to balance the required performance of the target application against the logic area cost of the processor cell. All possible configuration parameters are summarized in Table 3-1.

Table 3-1. Possible design-time configuration parameters in the processor cell.

Item	Value range	Default value	Configurability
Data bus width [bits]	16 (MAC), 32 (DSP)	32	Design-time
General purpose registers	1 ~ 19	8	
Program memory depth (wordlength = 32 Byte)	Integer multiple of 2, e.g. …, 64, 128, …	64	
Local I/O ports	1 ~ 8	8	
Global I/O ports	1	1	Currently not configurable
Accumulator	Enable, Disable	Disable	No, automatically enabled in the MAC processor.
Barrel shifter	Enable, Disable	Disable	Design-time

By choosing different data bus widths, the processor cell can be configured to operate in two modes: the 16-bit MAC processor with multiplication support, and the 32-bit DSP processor with radix-2 butterfly support. The MAC processor uses a parallel move instruction to split and join 16-bit internal registers and 32-bit fixed length I/O port registers. In addition, a 48-bit accumulation (ACC) unit is automatically enabled when the MAC processor is selected. From a hardware cost perspective, the 32-bit hardware multiplication function is not supported in DSP processor. Instead, the real and complex valued multiplications can be processed through the more advanced arithmetical co-processing unit, the CORDIC cell, as presented in the following chapter. The number of global I/O ports is currently not configurable, where the only port available is shared between normal data transferring and system configuration package handling. Based on different data bus widths, either a 48-bit or a 64-bit barrel shifter can be selected. The number of bit shifts can be specified by using instruction flags (refer to section 3.2), where the maximum bit shifts supported is 16 bits per clock cycle. Due to the large hardware requirements, the barrel shifter is disabled by default.

3.1.2 Run-time operational configuration

A few control possibilities are provided to the user to configure the processor cell during run-time, as listed in Table 3-2.

Table 3-2. Possible run-time configuration parameters in the processor cell.

Item	Option	Configurability
Instruction download	Full, partial	Run-time
Program counter value update	Instruction address	
Operation control and debug	Start, stop, reset, single step	
Running status tracing	---	

Run-time configuration is handled by a dedicated controller inside the processor cell, referred to as an *operation controller* and shown in the block diagram in Figure 3-1. The 32-bit configuration data packets are sent over the global network, where the network type identifier is specified as "config". During system run-time, the operation controller keeps track of data packages received from the global I/O port. When a configuration package is detected, the corresponding control actions are sequentially executed. The operation controller unit is controlled by a finite state machine (FSM), and a detailed control flow graph is shown in Figure-Appendix 5.

Each configuration packet contains a *header* and a *payload*, where the header specifies the target address and size of the payload data. A processor cell has two run-time reconfigurable parts, a *program memory* and a *control register*. The program memory is indexed from address 1, where location 0 is reserved for the control register. Several different program sections can be stored in the program memory, where the user can reconfigure the program counter value on the fly in order to select which program section to execute. The control register contains configuration bits to start, stop, reset and single-step the processor. The complete configuration package format is presented in Figure-Appendix 6, where an example of run-time reconfiguration for the processor cell is shown in Figure 3-2.

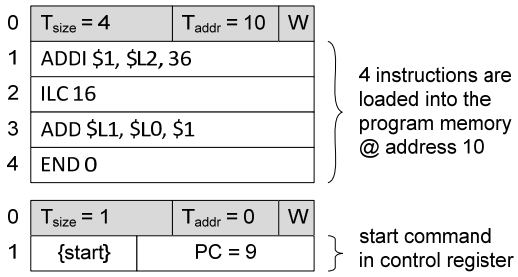


Figure 3-2. 32-bit configuration packets. Configuration of a processing cell, including program memory ($T_{addr} > 0$) and control register ($T_{addr} = 0$). Configuration headers are shaded in gray.

3.1.3 Hierarchical system resets

Three levels of system resets are used in the processor cell design, as illustrated in Figure 3-3. The top level *processor cell reset* is hard wired to the cell I/O port, which initializes the entire processor pipeline and program memory. The lower two levels of reset signals are derived from the processor cell reset and are controlled by

the operation controller. The *processor pipeline reset* initializes all internal registers including the GPRs, I/O port registers and the pipeline stage resisters, while the *pipeline register reset* only flushes out data contents stored in the pipeline stage registers.

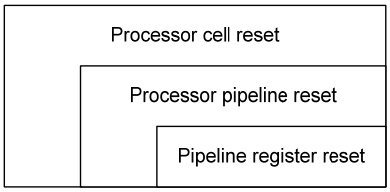


Figure 3-3. Hierarchical system resets.

3.1.4 Register bank

The 32 allocable address spaces in the *register bank* are sequentially ordered and partitioned into four divisions: general purpose, local I/O, global I/O and special purpose register (SPR) bank. All register banks are transparent and accessible for the user except for some of the SPRs. Table 3-3 summarizes four different partitions in the register bank and their corresponding accessibilities. Notice that, three of the register addresses are shared between the GPRs and the SPRs.

Table 3-3. Address space partitions and accessibility summary for the register bank.

Register bank partition	Allocated/shared address space		Accessibility
GPR	0 ~ 18		Yes
Local I/O registers	19 ~ 26		
Global I/O registers	27		
SPR	8	GIO destination ID register	No
	9	Inner loop counter	
	10	Inner loop program counter address register	
	28	Program counter	
	29	ALU status register	
	30	Low 16-bit ACC register	Yes
	31	High 32-bit ACC register	

The I/O port registers are directly accessible in the same way as the GPRs. Hence, no addition operations are required to move data between registers and ports, which significantly increase the processing rate [1]. As an example, the following instruction adds an input operand which is loaded from a GPR ($1) to an immediate value (10), and the sum is sent through a local I/O port.

```
ADDI $L2, $1, 10          // $1 + 10 => $L2
```

A program instruction that accesses I/O port register for data receiving is automatically stalled until data becomes available. Similarly, a data sending instruction cannot proceed until the corresponding port register is writable.

3.1.5 Dual and separable ALU

A conventional ALU takes two input operands and produces a single result value. In contrast, the DSP and MAC processors include two separate ALUs to produce two result values in a single instruction. This is useful when computing a radix-2 butterfly or when moving two data values in parallel [1].

Each 32-bit ALU data path can be separated into two independent 16-bit fields, where arithmetic operations are applied to both fields in parallel. This can be used when operating on complex valued data, represented as a 2×16-bit value. Hence, complex values can be added and subtracted in a single instruction [1]. Block diagram of the ALU in the processor cell is illustrated in Figure 3-4.

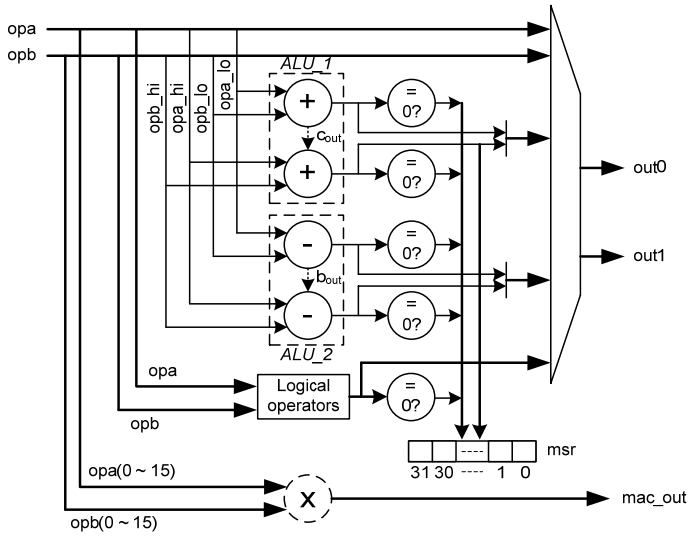


Figure 3-4. Block diagram of the ALU in the processor cell. Wordlength of the data path *opa_lo*, *opa_hi*, *opb_lo* and *opb_hi* are one-half of the data bus width *opa* and *opb*, respectively. Data bus *opa* and *out0* have the fixed wordlength of 32 bits, whereas *opb* and *out1* are dependent on the processor data bus width, e.g. 16 bits in the MAC processor and 32 bits in the DSP processor. *ALU_1* consists of adder_low and adder_high, and *ALU_2* is made up of sub_low and sub_high. Multiplier is enabled only in the 16-bit MAC processor.

Each time when a computation is performed inside the ALU, result status from all computational operators are collected and stored in a 32-bit status register (MSR), as summarized in Table 3-4. This can be used for instance when checking the computation overflows, detecting negative results and conditional program branching, etc. Currently, the accessibility of the ALU status register is not yet supported, which requires further development.

Table 3-4. Bit map of the ALU status register (MSR).

Bit position	Flag	Description
0	z0	Zero flag of *adder_low* and *logical operators*
1	z1	Zero flag of *adder_high*
2	z2	Zero flag of *sub_low*
3	z3	Zero flag of *sub_low*
4	n0	Negative flag of *adder_low* and *logical operators*
5	n1	Negative flag of *adder_high*
6	n2	Negative flag of *sub_low*
7	n3	Negative flag of *sub_high*
8	c0	Carry flag of *adder_low* and *logical operators*
9	c1	Carry flag of *adder_high*
10	b0	Borrow flag of *sub_low*
11	b1	Borrow flag of *sub_high*
12 ~ 31	---	Reserved

3.1.6 Inner loop counter

A special set of registers are used to reduce control overhead in compute-intensive inner loops. The inner loop counter (ILC) register is loaded using a special instruction that stores the next program counter address. Each instruction contains a flag that indicates end-of-loop, which updates the ILC register and reloads the previously stored program counter [1].

3.2 Processor instruction set

All processor instructions are defined in 32-bit format. Depending on different types of input operands, instructions are grouped into two categories: *type A* uses only registers as operands and *type B* uses immediate value. The different instruction types are identified by the MSB value of the instruction operation code (OPCODE). Two basic instruction templates are shown in Table 3-5. One additional option in type A instructions is a 6-bit function flag. The flag directs the processor cell to carry out additional function while executing the current instruction. Examples are the flag that ends an inner loop, or the flag that accumulates resulting values from data multiplications.

Table 3-5. Two basic instruction templates for the processor cells.

Type	31~26	25~21	20~16	15~11	10~6	5~0
A	OPCODE	D0	D1	S0	S1	Flags
B	OPCODE	D0	S0	Immediate		

A complete list of instruction set with all possible instruction flags is presented in Figure-Appendix 3 and Figure-Appendix 4. Worth mentioning is that, a few specialized instructions have been designed for the processor cell based on the frequently used operations in DSP applications. The radix-2 butterfly, 32-bit data swap and double data move instructions are examples of such.

The *branch* type instructions are handled in the *ID* pipeline stage. Thereby, two execution clocks are required to perform a branch-taken operation: one normal instruction clock and an additional clock cycle to flush out the pipeline stage registers.

3.3 Pipeline hazards

There are situations, called hazards, which prevent the next instruction in the instruction stream from executing during its designated clock cycle [8]. Pipeline hazards in a processor can be divided into three types: structural, data and control hazards. Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution [8]. Data hazards are caused by the data dependences between two adjacent instructions. Control hazards arise from instructions that change the value of program counter, such as program branching. For a single processor cell in the CGRA, there are at least six non-maskable pipeline hazards, where structural hazards generated from program execution are not included. All possible pipeline hazards are ranked according to their significance, as listed in Table 3-6.

Table 3-6. Priority of all possible pipeline hazards in a processor cell

Priority	Hazard	Hazard type
1 (Highest)	System reset	Control hazard
2	Program execution done	Control hazard
3	User control	Control hazard
4	Data receiving (RX) stall	Data hazard
5	Data transmission (TX) stall	Data hazard
6 (Lowest)	Program branch	Control hazard

System reset has the highest priority among all pipeline hazards, and the *program execution done* event comes after. These are used to ensure that all processor pipeline stages can be flushed and suspended in time. The user control events are ranked at the third level, since the user should be able to determine the current running status of the processor cell, e.g. starting or stopping data communications. Data RX stall event has a slightly higher priority than the TX stalling, which is considered to prevent any data package lost.

Hazards in pipelines can make it necessary to stall the pipeline. Basic design criterions to handle the pipeline hazards are presented in [8], and are repeated here:

a) When an instruction is stalled, all instructions issued later than the stalled instruction – and hence not as far along in the pipeline – are also stalled.

b) Instructions issued earlier than the stalled instruction – and hence farther along in the pipeline – must continue, since otherwise the hazard will never be cleared. As a result, no new instructions are fetched during the stall.

3.3.1 Hazards-handling in IF/ID stage

In IF/ID stage, "bubbles" (*NOP* instructions) are inserted into the following pipeline stages during the system reset and the program execution done hazards. This is because no instruction should be issued after resetting the entire system and before receiving any commands from the user, or there is no more instruction to be issued after executing a program section. Hazards-handling in ID/EXE stage are summarized in Table 3-7.

Table 3-7. Handling of pipeline hazards in IF/ID stage.

Priority	Hazard	Handling
1 (Highest)	System reset	a) Reset program counter value; b) Insert "NOP" instruction.
2	Program execution done	a) Preserve program counter value; b) Insert "NOP" instruction.
3	User control	Preserve pipeline registers.
4	Data receiving (RX) stall	
5	Data transmission (TX) stall	
6 (Lowest)	Program branch	a) Update program counter value; b) Insert "NOP" instruction.

3.3.2 Hazards-handling in ID/EXE stage

For the program execution done hazard in ID/EXE stage, all pipeline registers are updated, which ensures that remaining instructions in the pipeline stages can still be executed. Since bubbles have already been inserted in the IF/ID stage during this hazard, no special handling should be made here.

During the user control hazard, all current program executions are hanged up. Additionally, register control signals in the EXE stage are cleared, because the duplicated arithmetical operations in the EXE stage should be prohibited, such as the value accumulations in ACC. Furthermore, all the RX units are suspended during this hazard, which ensures that no duplicated acknowledgment signals are transmitted to the data senders. Handling of the TX units is not carried out in the ID/EXE stage during the user control hazard. This is because relevant controlling should be kept close to the place where the target objects are located – that is the WB stage for the TX units in this case.

In addition to the handling in user control hazards, two special control operations are carried out during the data RX stall hazard. Firstly, not all of the RX units are

suspended during this event, whereas units that generated RX stalls will continue receiving data from their data sender until the expected data package arrives. Another special handling is that, all register control signals in the WB stage are cleared during this hazard. Considering a case where a program instruction forwards a data packet from a RX port to a TX port, if the RX unit is stalled due to the lack of incoming data packet, a RX stall hazard is asserted. In this case, data sending actions in the TX unit should be suspended until the required data packet is available. Hazards-handling in ID/EXE stage are summarized in Table 3-8.

Table 3-8. Handling of pipeline hazards in ID/EXE stage.

Priority	Hazard	Handling
1 (Highest)	System reset	Flush pipeline registers.
2	Program execution done	Update pipeline registers.
3	User control	a) Flush register control signals in the EXE stage; b) Suspend RX units; c) Preserve other pipeline registers.
4	Data receiving (RX) stall	a) Flush register control signals in the EXE and WB stages; b) Suspend RX units that did not generate RX stall; c) Preserve other pipeline registers.
5	Data transmission (TX) stall	a) Flush register control signals in the EXE stage; b) Preserve other pipeline registers.
6 (Lowest)	Program branch	Update pipeline registers.

3.3.3 Hazards-handling in EXE/WB stage

Table 3-9. Handling of pipeline hazards in EXE/WB stage.

Priority	Hazard	Handling
1 (Highest)	System reset	Flush pipeline registers.
2	Program execution done	Update pipeline registers.
3	User control	a) Flush register control signals in the WB stages; b) Preserve other pipeline registers.
4	Data receiving (RX) stall	Update pipeline registers.
5	Data transmission (TX) stall	a) Suspend TX units who did not generate TX stall; b) Preserve pipeline registers.
6 (Lowest)	Program branch	Update pipeline registers.

Similar design criterion as described in the preceding section can be applied in EXE/WB stage to handle all possible pipeline hazards, as summarized in Table 3-9.

3.4 Performance evaluation in FPGA

To explore the performance metrics, processor cell with different configuration parameters have been synthesized for the target FPGA platform, the Virtex-II Pro-30-7ff896 from Xilinx. All results are obtained after the design synthesis. Two kinds of RTL synthesis processes have been carried out, one for extracting the maximum operation speed and the other for exploring the minimum hardware usage. Default compilation constraints are used during the RTL synthesis, and the actual synthesis work is carried out in Xilinx XST version 10.1.03. All possible configuration parameters for the processor cell have been listed in Table 1-1. Here, the size of the program memory and the number of GPR remain unchanged during the entire evaluation process, where the values are set to 32B×64 and 16, respectively. All the synthesis results are listed in Table 3-10.

Table 3-10. Performance evaluation of the processor cell based on an FPGA device.

Configuration			**Maximum speed**		**Minimum area**		**Differences**	
			Slices	**f_{max} [MHz]**	**Slices**	**f_{max} [MHz]**	**Slices**	**f_{max} [MHz]**
Data bus width = 16-bit, ACC=Yes, GPR=16	BS=No	LIO=2	1,149	50.97	1,161	41.51	-12	9.46
		LIO=4	1,240	51.06	1,243	41.51	-3	9.55
		LIO=8	1,471	56.68	1,415	41.67	56	15.01
	BS=Yes	LIO=2	1,320	40.73	1,303	30.65	17	10.08
		LIO=4	1,435	40.12	1,405	30.68	30	9.43
		LIO=8	1,607	40.47	1,574	31.82	33	8.65
Data bus width = 32-bit, ACC=No, GPR=16	BS=No	LIO=2	1,579	56.42	1,538	42.33	41	14.09
		LIO=4	1,669	55.38	1,636	42.27	33	13.12
		LIO=8	1,833	57.00	1,835	42.33	-2	14.67
	BS=Yes	LIO=2	1,684	40.63	1,697	32.82	-13	7.81
		LIO=4	1,795	40.08	1,803	32.56	-8	7.52
		LIO=8	1,992	40.14	2,018	32.86	-26	7.28

Similar results are reported from two synthesis processes, both for hardware slice usage and maximum operation speed. However, this is only true on an FPGA platform, as the utilized arithmetic multiplier is one of the hardware macros provided by the FPGA device. Thereby slice usage of the hardware multiplier is not counted in the synthesis report. From a previous ASIC implementation attempt based on a 0.13μm CMOS cell library [3], about 5% area and over 250 MHz speed differences can be found from these two different synthesis approaches.

Referring to the plots shown in Figure 3-5, hardware usage changes linearly with the variation of system configurations. The 32-bit DSP processor in general uses about 400 more FPGA slices than the 16-bit MAC processor on the target platform, and approximately 150 slices can be saved if the barrel shifter is excluded.

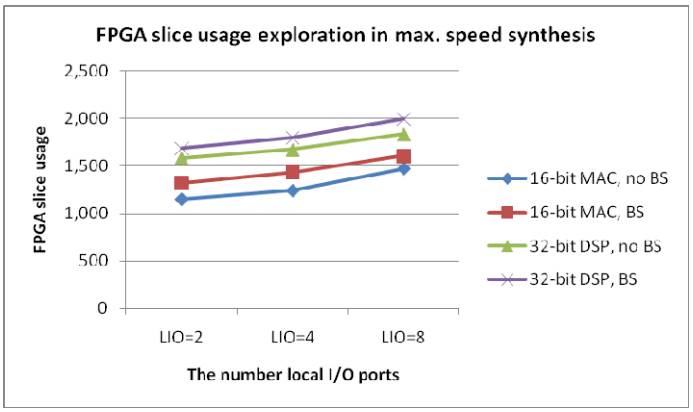


Figure 3-5. FPGA slice usage exploration based on an FPGA platform. BS is short for Barrel Shifter.

Analysis on the critical timing path for the processor cell design is performed in two phases. Firstly, all pipeline stages inside the processor cell are synthesized as a stand-alone building block. This is considered to get an impression on the capable running speed of the self-made logics, as the program memory is realized by using block RAMs inside the FPGA device, which has fixed hardware properties. In the first evaluation process, the MAC unit appears in the longest timing path, where over 60% of the total signal delays are reported from its internal data path, as illustrated in Figure 3-6.

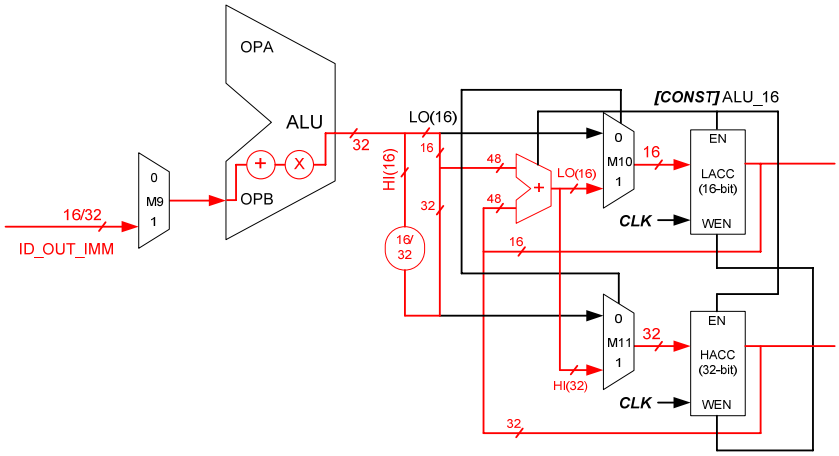


Figure 3-6. Critical timing path in the pipeline stages.

One solution to address this problem is to use a pipelined hardware multiplier and better adder/subtractor [1] implementations for the MAC unit, for instance the

[1] Based on the Xilinx FPGA platform, the "ripple carry adders" are reported from the automatic synthesis process.

coarse-grained multiplier and look-ahead adder, respectively. However, this solution results in additional hardware requirements to allow increased operation speed, which is a matter of design trade-off between hardware area and system performance.

In the second evaluation phase, the processor cell is synthesized and analyzed. By tracing the synthesis report, maximum operation speed of the entire processor cell suffers from an architectural design issue. Operations on the program memory are triggered on the falling edge of the system clock, which is considered to ensure the setup and hold time for control signals from pipeline registers. As a consequence, demanded timing constraint is pushed to the processor pipelines, as all combinatorial operations have to be accomplished within half of the execution clock. Obviously, a solution to overcome this problem is to use the same clock edge when triggering the program memory as well. But due to the signal propagation delays, instructions issued in the ID stage will be delayed for two clock cycles compared to the program counter value in the IF stage. This might require additional control logic for pipeline hazards, which needs to be further considered during future design optimization.

3.5 Conclusion

A single-issue RISC processor cell developed for the CGRA has been presented in this chapter. From a viewpoint of hardware architecture, processor cell in the CGRA is constructed from four pipeline stages, whereas data memory is located outside the cell. This is one of the main differences between the processor cell and the conventional RISC core, as data memories are global system resources that are distributed over the whole platform and are shared by all surrounding cells in the CGRA.

Processor cell is characterized from its static *configurability* and dynamic *reconfigurability*. Depending on different user requirements, the flexible hardware structure makes the processor cell possible to be configured during system design-time. Two of the processor cell configurations have been given as examples in this chapter, namely the 16-bit MAC processor with hardware multiplication support and the 32-bit DSP processor with radix-2 butterfly support. Run-time reconfiguration on the processor cell is achieved through an internal operation controller and a nested control register. By sending configuration packages over the global routing network, the user is able to download new program segments, update the program counter address and control running status of the processor cell, etc.

Due to the targeted application field of the reconfigurable cell array platform, i.e. applications in digital signal processing, the processor cell has been enhanced with a few DSP operations, such as the radix-2 butterfly, data swap and double data move instructions.

Six non-schedulable pipeline hazards and their corresponding handling have been described in this chapter. Based on the different significance inside the processor cell, all possible pipeline hazards are assigned with the priority, and are handled internally without the user interactions.

Different performance metrics for the processor cell have finally been evaluated.

By analyzing the critical timing path, an architectural design issue has been explored, which has dramatically influenced the system performance. Therefore, further system optimizations are needed in future work, where a speedup factor of about 2 is speculated.

On software level, processor cells are currently programmed in assembly language and manually translated into binary codes. Any of the design automation tools, such as the assembler or C compiler, program optimizer and function emulator, etc., is not covered in this project. A system exploration framework, the SCENIC, developed in previous work [1][2][6][7] can be a good start point for further study and developments.

3.6 Future work

3.6.1 Application specific instruction set processor (ASIP)

Generally, design of the system architect is dominated by the requirements from the target application field, where system performance can be characterized from many different aspects, such as the operating speed, hardware area, power consumption and design flexibility, etc. Hence, instead of integrating a generic processor cell on silicon which can cover a broad range of applications, it is often sufficient to use an application specific instruction set processor (ASIP) that is optimized for a narrow range of applications. Optimizations may be performed at the micro- architecture level, so that functional units and memory system are tuned to the specific application. Furthermore, it may include exploring instruction- or data-level parallel architectures. However, the most characterizing for an ASIP is the instruction set customization [7]. For instance, the MAC operation is essential in a processing intensive application such as the audio streaming, but this is not so critical in a system that has the main constraints on the hardware design area, whereas the timing requirements are relaxed. Therefore, it is desirable to have a flexible control on forming the processor instruction set. This can be achieved by packing instructions into different categories, and only enable the ones required by the specific application field in system design-time.

3.6.2 In-system reconfiguration

The run-time system reconfiguration is one of the key points in the CGRA. Currently, this is achieved through interactions between the host of the reconfigurable system and the resource cell, where the host could either be the user or an external function block. Either way, this approach demands data communications over the global routing network, sometimes might even require off-chip data transmissions. This is known to be inefficient due to the data communication latency, and hence causing the increased reconfiguration time. One solution to address this problem is to let the

processor cell support more global packet sending types, i.e. to support the “config” type, so that processor cells will have the possibilities to reconfigure their surrounding cells. By doing so, the scheduled system reconfigurations can be managed by using one of the processor cells inside the CGRA, or by using the processor cells nearby the target objects, where the configuration package sending can be kept internally over the high speed local network, thereby further shortens the reconfiguration time.

3.6.3 Debugging approaches

Although a few run-time control possibilities have been provided in the processor cell to be able to for instance trace the operation status and step through program sections, it is in general far too simple to be used for diagnosing problems, especially for complex program executions, as the ability of controlling and observing the internal registers is lacking. Several advanced debugging approaches can be used to improve the *controllability* and *observability* on a testing platform, two of them are proposed here. Firstly, the industry standard joint test action group (JTAG) chain is the most straightforward approach to use, because all the internal data contents of interests can be serially clocked out, if the corresponding internal registers are replaced with JTAG scan registers. Secondly, the built-in self-test (BIST) can be used as an additional method to verify basic functionalities of the system. If the in-system reconfiguration feature mentioned previously can be realized, the BIST is supported automatically by the CGRA without any additional hardware costs. Since one of the processor cells can be programmed as a system master that can send test patterns serially to the other resource cells, compare the processed outputs with golden references, and finally send the test report to the user.

4 CORDIC cell architecture

The CORDIC cell implements a generalized coordinate rotation digital computer (CORDIC) algorithm, which can be considered as an arithmetical co-processing unit to the processor cell in the CGRA. CORDIC is an iterative arithmetic algorithm that provides an efficient way of computing many elementary functions such as trigonometric, hyperbolic, logarithmic, and some linear functions including complex valued multiplication, etc. The idea of the CORDIC algorithm is to rotate the vector through a sequence of elementary angles using a linear, circular or hyperbolic coordinate system, where the algebraic sum of these angles approximates the desired rotation value [9]. All elementary angles are selected such that they can be implemented using only shift and add/subtract operation, hence no actual multipliers are needed.

Hardware implementation of the CORDIC cell is based on a pre-developed CORDIC kernel implemented in previous work [14]. In this project, a few functional improvements and a list of surrounding modules have been implemented to embed the CORDIC cell into the CGRA.

4.1 Theoretical background

4.1.1 CORDIC operations in circular coordinate system

The CORDIC algorithm is initially designed to perform a vector rotation, where the vector (x, y) is rotated through the angle yielding a new vector (x', y') [1], as illustrated in Figure 4-1. A general vector rotation by an angle θ can be expressed as,

$$\begin{aligned} x' &= x\cdot\cos(\theta) - y\cdot\sin(\theta) \\ &= \cos(\theta)\cdot\left[x - y\cdot\tan(\theta)\right] \\ y' &= y\cdot\cos(\theta) + x\cdot\sin(\theta) \\ &= \cos(\theta)\cdot\left[y + x\cdot\tan(\theta)\right] \end{aligned}$$

eq. 4-1 Real vector rotation

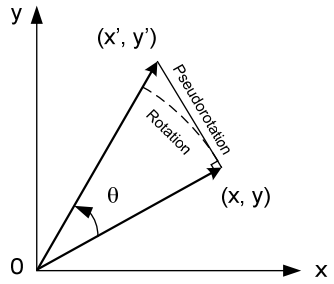


Figure 4-1. A vector rotation by an angle θ. The real vector rotation is drawn in dashed line, where the pseudo-rotation is drawn in solid line.

Rotation expressed in eq. 4-1 is called real vector rotation, where the magnitude of the vector being rotated is preserved. To simplify the CORDIC computation, the real rotation angle is replaced by a pseudo-rotation as depicted in solid line in Figure 4-1.

$$x' = x - y \cdot \tan(\theta)$$
$$y' = y + x \cdot \tan(\theta)$$

eq. 4-2 Pseudo vector rotation

This removes the term *cos(θ)* from the initial expression and hence results in a known magnitude expansion.

To further reduce the computation complexity, vector rotation in the CORDIC algorithm is realized in an iterative process that contains a sequence of successively smaller rotations, each of angle $\tan^{-1}(2^{-i})$, known as *micro-rotations* [11]. This reduces the multiplication of the tangent term to a single shift operation, since $\tan(\theta)=2^{-i}$. The direction of each micro-rotation is specified by the parameter d_i, which is chosen such that the remaining angles tend to go towards zero. The general vector rotation shown in eq. 4-1 can be expressed by using a series of micro-rotations,

$$\begin{aligned} x_{i+1} &= x_i - y_i \cdot d_i \cdot 2^{-i} \\ y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot \tan^{-1}\left(2^{-i}\right) \end{aligned} \quad \text{Where} \quad \begin{aligned} & d_i = sign(z_i) \,\&\, d_i \in \{-1,1\}, \\ & i = 0,1,...,n-1 \end{aligned}$$

eq. 4-3

The variable z is initialized by the desired rotation angle. It keeps track of the total elementary angles over micro-rotations and determines the sign of d_i. A CORDIC rotation is accomplished when z reaches 0, and the final results can be expressed as,

$$\begin{aligned} x_n &= K_n \cdot \left[x_0 \cdot \cos(z_0) - y_0 \cdot \sin(z_0)\right] \\ y_n &= K_n \cdot \left[y_0 \cdot \cos(z_0) + x_0 \cdot \sin(z_0)\right] \\ z_n &= 0 \end{aligned} \quad \text{Where} \quad K_n = \prod_{i=0}^{n-1} \cdot \sqrt{1 + 2^{-2 \cdot i}}$$

eq. 4-4

The scaling factor K_n represents the increased magnitude of the vector during the rotation process [12], which is related to the number of computation iterations. When n tends to infinite, K_n approaches the value 1.64676. Referring to eq. 4-5, the trigonometric sine and cosine functions can be obtained when CORDIC is operated in *rotation mode*. For example, by initializing $y=0$, output x and y converge to $K \times sin(z)$ and $K \times cos(z)$, respectively. For m bits of precision in the resulting trigonometric functions, at least m CORDIC iterations are needed [13].

In addition to the rotation mode, CORDIC can also be operated in *vector mode* to compute for instance the square root function. This is achieved by choosing d_i in a way so that y converges towards zero. The resulting outputs in CORDIC vector mode can be expressed as,

$$\begin{aligned} x_n &= K_n \cdot \sqrt{x_0^{\,2} + y_0^{\,2}} \\ y_n &= 0 \\ z_n &= z_0 + \tan^{-1}\left(y_0 \big/ x_0\right) \end{aligned} \quad \text{Where} \quad \begin{aligned} & d_i = -sign(x_i \cdot y_i) \,\&\, d_i \in \{-1,1\}, \\ & i = 0,1,...,n-1, \\ & K_n = \prod_{i=0}^{n-1} \cdot \sqrt{1 + 2^{-2 \cdot i}} \end{aligned}$$

eq. 4-5

4.1.2 Generalized CORDIC

By introducing a system parameter μ, the CORDIC algorithm can be generalized to operate in three different coordinate systems: circular, linear and the hyperbolic system. The generalized CORDIC is defined as,

$$\begin{aligned} x_{i+1} &= x_i - \mu \cdot y_i \cdot d_i \cdot 2^{-i} \\ y_{i+1} &= y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} &= z_i - d_i \cdot e_i \end{aligned} \qquad \text{eq. 4-6}$$

Where the partial angle values $tan^{-1}(2^{-i})$ used for calculating z_{i+1} in eq. 4-3 is redefined by using a variable e_i for each of the CORDIC iterations. Relations between μ, e_i and the CORDIC scaling factor are summarized below.

Table 4-1. Relations between system parameter μ, partial angle e_i and the CORDIC scaling factor K_n.

$\mu = 1$	Circular rotations (basic CORDIC)	$e_i = \tan^{-1}\left(2^{-i}\right)$	$K_n = \prod_{i=0}^{n-1} \cdot \sqrt{1+2^{-2 \cdot i}}$
$\mu = 0$	Linear rotations	$e_i = 2^{-i}$	1
$\mu = -1$	Hyperbolic rotations	$e_i = \tanh^{-1}\left(2^{-i}\right)$	$K_n = \prod_{i=0}^{n-1} \cdot \sqrt{1-2^{-2 \cdot i}}$

Table 4-2 [12][13]. The CORDIC functions for different operation modes.
$K_n = 1.646\ 760\ 258\ 121\ \ldots$ & $K_n' = 0.828\ 159\ 360\ 960\ \ldots$

	Rotation mode: $d_i = sign(z_i)$ & $z_n \to 0$	Vectoring mode: $d_i = -sign(x_i \cdot y_i)$ & $y_n \to 0$
Circular ($\mu=1$)	$x_n = K_n \cdot \left[x_0 \cdot \cos(z_0) - y_0 \cdot \sin(z_0)\right]$ $y_n = K_n \cdot \left[y_0 \cdot \cos(z_0) + x_0 \cdot \sin(z_0)\right]$ For *cos* & *sin*, set $x = 1/K_n$, $y = 0$	$x_n = K_n \cdot \sqrt{x_0^2 + y_0^2}$ $z_n = z_0 + \tan^{-1}\left(y_0 / x_0\right)$ For tan^{-1}, set $x = 1$, $z = 0$
Linear ($\mu=0$)	$x_n = x_0$ $y_n = y_0 + x_0 \cdot z_0$ For multiplication, set $y = 0$	$x_n = x_0$ $y_n = z_0 + y_0 / x_0$ For division, set $z = 0$
Hyperbolic ($\mu=0$)	$x_n = K_n' \cdot \left[x_0 \cdot \cosh(z_0) - y_0 \cdot \sinh(z_0)\right]$ $y_n = K_n' \cdot \left[y_0 \cdot \cosh(z_0) + x_0 \cdot \sinh(z_0)\right]$ For *cosh* & *sinh*, set $x = 1/K_n'$, $y = 0$	$x_n = K_n' \cdot \sqrt{x_0^2 - y_0^2}$ $z_n = z_0 + \tanh^{-1}\left(y_0 / x_0\right)$ For $tanh^{-1}$, set $x = 1$, $z = 0$

As can be seen from the generalized CORDIC expression (eq. 4-6), each of the computation iterations requires only three add/subtract operations and three bit shifts. All the partial rotation angles can be pre-computed and stored in a lookup table (LUT). In addition, as long as the scaling factor is known, output results with the increased magnitude can always be compensated by post-processing. A list of CORDIC functions for different operation modes has been summarized in [12][13] and is repeated here as shown in Table 4-2.

4.2 Cell architecture

An overview of the CORDIC cell architecture is shown in Figure 4-2, which is constructed from a CORDIC kernel, an I/O register bank and an operation controller. The CORDIC kernel is based on a pipeline structure, with 3 data input values and 2 data output values, capable of producing one set of CORDIC results each clock cycle. The I/O register bank handles data communications between the CORDIC and surrounding cells, monitors data traffic status, and generates flow control signals that are used in the CORDIC kernel. The internal computation wordlength, the number of iteration stages and the number of local I/O ports are configurable at system compilation time, while the configuration controller provides run-time reconfigurability, such as the computation function selection, data I/O port definitions and the option of result concatenating, etc. Possible configuration parameters for the CORDIC cell are summarized in Table 4-1.

Table 4-3. Configuration parameters in the CORDIC cell.

Item	Value range	Default value	Configuration type
Configuration register	64-bit configuration packet format	No action	Run-time
Internal wordlength	2~ 24*	16	Design-time
Iteration stages	2~ 24*	16	
Local I/O ports	1 ~ 8	8	
Global I/O ports	1 ~ 8, GIO(0) is shared with the configuration port	1	

Note: * Maximum wordlength supported is limited by the number of partial rotation angles stored in the coefficient LUT.

4.2.1 CORDIC kernel

A pipeline implementation of the CORDIC algorithm is chosen since it can produce one output value each clock cycle, which balances the processing capacity and the local network capacity in the CGRA. Due to the use of the pre- and post-processing stages, the total latency of this structure is equal to the number of processing stages plus 2.

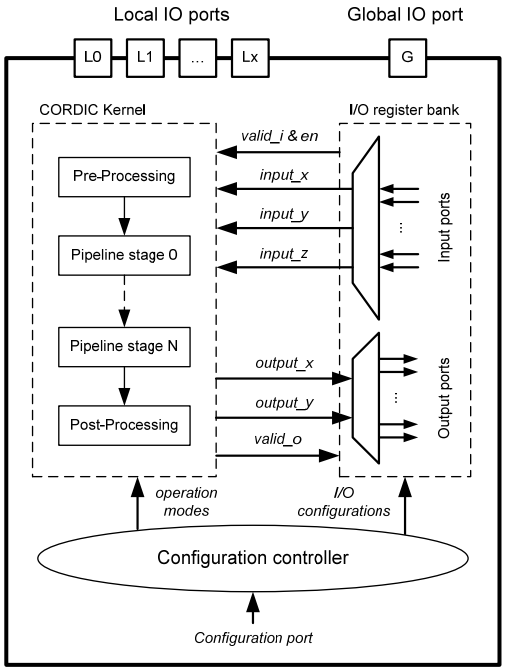


Figure 4-2. Block diagram of a CORDIC cell

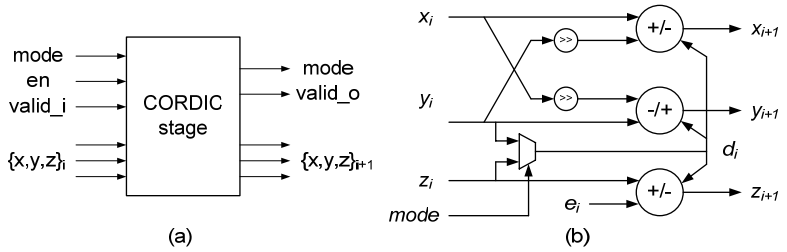


Figure 4-3 [14]. (a) A basic CORDIC building block. (b) The internal hardware structure of a single CORDIC computation stage. The constant and shift factor is unique for each block.

Figure 4-3 (a) shows the block diagram of a single CORDIC stage. In addition to three data inputs, an operation mode and two control signals are used in each pipeline stage. A global control signal (*en*) enables/disables the stage operation, while the signal *valid_i* controls the data processing in the current pipeline stage. A high *valid_i* signal triggers the current stage to take a new set of data from its input ports, compute and propagate the output values, operation mode and the flow control signal to the following stage. When the *valid_o* reaches the end of the pipeline, a complete CORDIC operation is accomplished.

In the pre-processing stage, all the input data values are checked and the ones who are initially outside the data quadrant I and IV are sited to these two areas, as the domain of computation convergence is [-99.7º, 99.7º] [13]. In the post-processing stage, the corresponding data correction for the output results is hence needed and carried out. In addition, results *output_y* and *output_z* are multiplexed to one data output in the post-processing stage, as either one converges to 0 depending on the different operation modes. Figure 4-3 (b) shows the hardware structure for a single

CORDIC computation stage, which requires three adder/subtractors, two bit shifters and control logics for operation mode selection and d_i condition. The partial rotation angle e_i is loaded from a coefficient LUT inside the CORDIC kernel. Currently, the hyperbolic operation mode is not supported in the hardware implementation, which will be added in future work.

4.2.2 I/O register bank

Based on the configured input and output ports, the I/O register bank handles data communications with surrounding cells. In addition, it monitors data traffic status and provides two flow control signals used in the CORDIC kernel, i.e. the enable signal and data input valid. The CORDIC kernel can be enabled as long as no data sending stalls are detected, since the current data set in the pipeline stage cannot be forwarded when the preceding data process is not completed. Conditions for issuing an input valid signal are resolved when all data inputs are available at the receiving ports, and when no user configurations are in progress.

At the data input, value *input_x* and *input_y* are concatenated together and transmitted through one single I/O port, while *input_z* is transmitted individually through the other port. The maximum input wordlength supported by the I/O register bank is 16-bit each. Computation results *output_x* and *output_y* can be transmitted in two ways, either sent through two I/O ports, or to be concatenated together before sending through one single data port. In the latter approach, if the concatenated output has a longer wordlength than the I/O payload's, each of the outputs will be clamped to 16 bits (high-nibble).

4.2.3 Configuration controller

Similar to the operation controller in the processor cell, the basic operations of the configuration controller unit in the CORDIC cell are controlled by an FSM to download and upload configuration packages. A detailed control flow graph of the FSM is shown in Figure-Appendix 8.

Input configurations for the CORDIC cell are defined in a 64-bit format, which are transmitted through two consecutive data packages, each of 32-bit length, over the global network. All cell configurations are stored in a 64-bit control register inside the configuration controller. This register is transparent to the user, which is both readable and writable during run-time. In the configuration table, there are control bits to enable the CORDIC kernel, define the computation function, specify the output result format, reset the CORDIC cell, and define the data I/O ports with the relevant port properties. More detailed descriptions for the configuration packages are presented in Figure-Appendix 9.

The operation mode for the CORDIC kernel is defined by using 2 configuration parameters, the function code F and the pure function selection P, as described in Table 4-4. With the pure function P enabled, the corresponding data input values are

tied to 0 in the CORDIC kernel, which prevents incorrect data inputs from the I/O port. For instance, *input_y* is discarded when calculating the trigonometric sine and cosine functions, as described in Table 4-2. The function code *F* is defined by using 3 bits, wherein 1 bit is used to specify the running mode of the CORDIC kernel such that to operate either in rotation (value 0) or vectoring (value 1) modes, and the other 2 bits are used to define the coordinate system which represents the value set {1, 0, -1}, corresponding to the circular, linear and the hyperbolic system, respectively.

Table 4-4. Configuration parameters for the CORDIC functions.

Pure function selection (*P*)	**Function code (*F*)**			**Function description**
	Mode	**Coordinate**		
0	0	0	0	Multiply and accumulate
	0	0	1	Complex number rotation
	0	1	1	Hyperbolic complex number rotation
	1	0	0	Divide and accumulate
	1	0	1	Tangent and accumulate
	1	1	1	Hyperbolic tangent and accumulate
1	0	0	0	Multiplication
	0	0	1	Sine & cosine
	0	1	1	Hyperbolic sine & cosine
	1	0	0	Division
	1	0	1	Absolute and phase
	1	1	1	Tangent

4.3 Computation accuracy analysis in MATLAB

A bit-accurate simulation model for the CORDIC kernel has been developed in MATLAB. This is used to analyze the computation accuracy in a fixed-point hardware platform. All the user configuration parameters for the CORDIC computation kernel are supported in this simulation, such as the internal calculation precision, operation modes and the pure function selection, etc. The VHDL simulation results are compared with the reference outputs generated in MATLAB.

The CORDIC calculation precision is evaluated by using a metric called *effective digits* developed by Hu [9]. The number of effective digits at the output is calculated as following,

$$d_{eff} = d_{in} - \log_2\left(E_q\right) - 1 \qquad \text{eq. 4-7}$$

where d_{in} is the input data width, E_q is the maximum computation error and *subtract 1* removes the effect from input sign bit. The number of effective digits is dependent on the wordlength used in each computation stage and the number of iterations. If *m* bits is the desired output precision, the "rule of thumb" [10] suggests that all the internal calculation stages should have $log_2(m)$ additional bits inserted to increase the

computation precision, referred to as the guard bits. This has been simulated by comparing the output effective digits when varying the internal wordlength. Three different input data vectors have been used in the simulations and the results are summarized in Table 4-5. Notice that, magnitudes for the input data vectors are scaled in half (*×0.5*) in order to avoid computation overflows[2]. The pseudo-rotation scaling factors in the CORDIC outputs are compensated in MATLAB. Both the internal wordlength and the number of iterations for the CORDIC kernel are bound together in this test.

Table 4-5. Simulation summary for the *effective digits* (real parts of the results only), where d_{input} is the input data width, and $d_{internal}$ is the internal wordlength in all computation stages. The CORDIC kernel operates in the complex number rotation mode.

$d_{internal}$ \ d_{input}	**8-bit**	**12-bit**	**16-bit**
8-bit	4.67-bit		
9-bit	5.48-bit		
10-bit	6.46-bit		
11-bit	7.31-bit		
12-bit	8.68-bit	7.56-bit	
13-bit		9.33-bit	
14-bit		9.98-bit	
15-bit		11.53-bit	
16-bit		12.03-bit	11.87-bit
17-bit			13.11-bit
18-bit			14.14-bit
19-bit			14.86-bit
20-bit			15.75-bit
21-bit			17.09-bit

As shown from the result summary, output data precisions deviate from their corresponding data input by the amount of app. $log_2(m)$ when $d_{internal}$ equals to d_{input}. Along with the increased wordlength for internal computations, the number of effective digits increases. Notice that, at the point when $d_{internal}$ equals to $log_2(m)+1$ bits, the number of effective digits at the output is higher than their input data widths. This indicates that the calculation error under this circumstance is relatively smaller than the internal computation capacity.

To summarize, decision on choosing the internal wordlength is dependent on the target application. More bits to use, more computation precisions can be gained, but at the same time more hardware resources are needed.

[2] Due to the add/subtract operations in each of the pipeline stages, internal computation overflow will occur when input data values are too large.

4.4 Performance evaluation in FPGA

Similar to the performance evaluation procedures that have been done in the processor cell, two kinds of RTL synthesis processes are carried out for the CORDIC design. During the evaluation process, the internal wordlength and the number of computation iterations are kept the same, and the number of global I/O ports is fixed to 1. All the synthesis results are summarized in Table 4-6.

Table 4-6. Performance evaluation of the CORDIC cell based on an FPGA device.

Configuration		Maximum speed		Minimum area		Differences	
		Slices	f_{max} [MHz]	Slices	f_{max} [MHz]	Slices	f_{max} [MHz]
GIO = 1; LIO = 8;	Internal wordlength = 2	276	152.27	247	97.57	29	54.70
	Internal wordlength = 8	737	162.08	606	94.35	131	67.74
	Internal wordlength = 12	1,145	132.76	986	91.52	159	41.24
	Internal wordlength = 16	1,621	133.34	1,444	91.14	177	42.19
	Internal wordlength = 20	2,077	128.92	1,859	89.94	218	38.98
	Internal wordlength = 24	2,636	125.46	2,420	88.60	216	36.86
GIO = 1; Internal wordlength = 16;	LIO = 2	1,258	133.44	1,154	104.07	104	29.37
	LIO = 4	1,332	125.15	1,257	98.78	75	26.37
	LIO = 6	1,497	124.82	1,343	88.91	154	35.92
	LIO = 8	1,621	133.34	1,444	91.14	177	42.19

Thanks to the pipeline structure, maximum operation speed (f_{max}) in each of the synthesis processes does not vary too much with the configuration parameters. Generally, the resulting FPGA slice usage from these two synthesis processes have no big difference, since the amount of arithmetical units used in the hardware implementation are limited. Furthermore, as seen from Figure 4-4, hardware slice usage changes linearly with the internal wordlength, so a trade-off between these two factors has to be decided upon the actual usage.

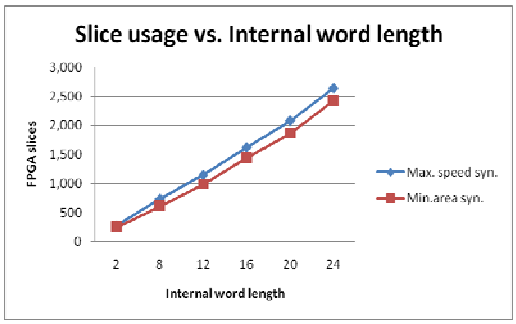


Figure 4-4. FPGA slice usage vs. internal computation wordlength.

4.5 Conclusion

The CORDIC algorithm provides an elegant convergence method for evaluating trigonometric and many other useful functions [13]. Emphasis in this chapter was placed on the CORDIC cell implementation aspects, whereas the theoretical analysis on the algorithm itself has been kept basic.

The CORDIC cell consists of three building blocks, a computation kernel, an I/O register bank and an operation controller. A pipeline implementation has been chosen for the computation kernel, to match the high communication bandwidth of the local network in the CGRA. The number of effective digits is used as a metric in analysis of the computation accuracy, and the consequence of varying the internal computation wordlength has been studied. Calculation precision versus hardware area usage always exists at the same time, and appropriate parameters have to be selected based on the target application field.

4.6 Future work

4.6.1 Coefficient generator

Often, certain properties can be explored from one or more CORDIC inputs in a real application. For example, when CORDIC operates in circular rotation mode, trigonometric sine and cosine functions can be calculated, and operand *input_z* is used as input rotation angle. If the required computations iteratively rotate through the angle space with regular steps, it is sufficient to use a sequential counter for generating the *input_z* during run-time. This is the case for instance in the FFT computation, where the phases of the complex numbers being processed are regularly rotated, i.e. complex multiplications with twiddle factors. In the conventional solution, all twiddle factors are stored as coefficients inside the data memory, or computed during run-time by using a processing unit [15]. But in general this is a waste of system resources, since the operation of generating the twiddle factors is simple. To improve the fact, a dedicated hardware unit can be embedded into the CORDIC cell, where simple operations like the one mentioned above can be handled internally.

5 Memory cell architecture

Most algorithm implementations require some kind of memory space for intermediate data storage, data reordering, or delay operations. Traditionally, data memory is located inside the processor, which results in difficulties when sharing data contents between surrounding cells. Therefore, external memory units are expected to be distributed and shared between all system components. This distributed approach not only supports information sharing between different processing elements, but also enables direct data transfers between memory cells without additional control logic.

To meet the basic requirements in most algorithm implementations, the distributed memory cells have to be intelligent and flexible enough to support different operations at run-time, such as operating in FIFO, RAM and ROM mode, etc. The run-time configurability also brings in another highlight for having this distributed architecture. Since conventionally many different type of memory modules are needed in an embedded system, but only a few of them have simultaneous operations during system run-time. Because of the control logic requirements in each memory module, this conventional system setup fashion might not be area and power efficient. By introducing a smart memory cell with relatively large data storage and a necessary operation controller, the hardware overheads mentioned above could be eliminated hence saving power.

A RTL-level memory cell implementation is initiated and constructed by a group of students in the *IC project and verification* course in 2008, according to the specification [17] made by Thomas Lenart. In this project, a few changes have been made in order to adapt the memory cell into the whole system structure.

5.1 Cell architecture

As illustrated in Figure 5-1, a memory cell consists of three main building blocks: a descriptor (DSC) table, an operation controller and a memory array. The descriptor table contains an array of configuration registers to store user-defined transactions. Each descriptor *reserves* a storage space from the memory array, called a memory bank; *defines* an operation behavior for the allocated memory bank during the designated execution cycle; *records* memory operation status; and s*pecifies* cell I/O ports for each stream transfer. The operation controller *manages* and *schedules* the descriptor processing; *monitors* data transfers; and *controls* the corresponding memory operations. The memory array is a shared memory space, which is able to handle one data read and one data write operation simultaneously.

Each memory descriptor is 64-bit wide, and the memory array has an operation wordlength of 32 bits. The length of a descriptor table, size of the memory array, and the number of cell local I/O ports are configurable at system design-time, while the contents of memory descriptors are dynamically reconfigurable during run-time. Possible memory cell configuration parameters are summarized in Table 3-1.

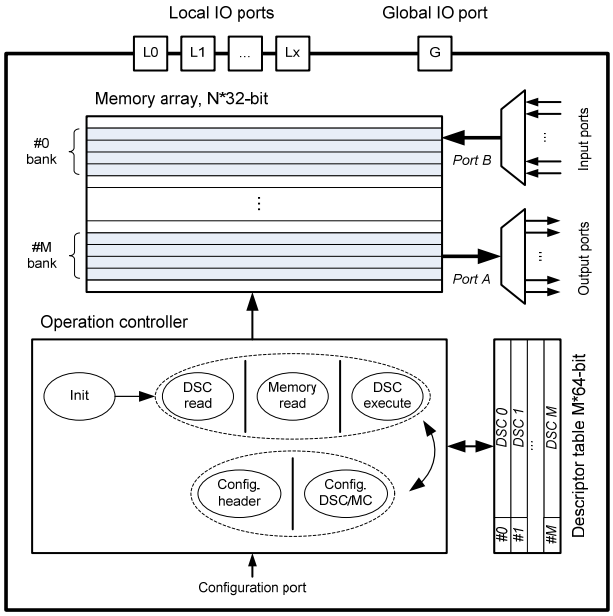


Figure 5-1. Block diagram of a memory cell.

Table 5-1. Configuration parameters in the memory cell.

Item	Value range	Default value	Configuration type
Memory descriptor	64-bit configuration file format	No action	Run-time
Descriptor table length	Integer multiple of 2, e.g. …, 2, 4, …	4	Design-time
Memory array wordlength	Integer multiple of 2, e.g. …, 16, 32, …	32	
Memory array depth	Integer multiple of 2, e.g. …, 128, 256, …	256	
Local I/O ports	1 ~ 8	8	
Global I/O ports	1 ~ 8, GIO(0) is shared with the configuration port	1	

5.1.1 Operation controller

Basic operations in a memory cell are controlled by an FSM that contains three main states: initialization, execution and configuration, as illustrated in Figure 5-2.

After cell reset, control registers and memory descriptor table are initialized in state “initialization”. Thereafter, the “execution” state is entered unconditionally and the operation controller stays here to process descriptors in sequential order. The “Configuration” state is activated if a configuration data package is received from a global input port. Memory cell configurations are managed in two sub-sequences. First of all, a list of configuration parameters are needed to be specified by receiving a 32-bit wide configuration header as defined in Table 5-2. Secondly, actual configuration data is sent to the memory array or to the descriptor table.

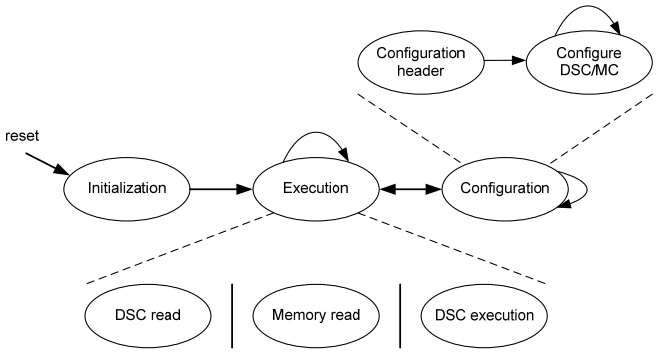


Figure 5-2. Memory cell control FSM.

Table 5-2. Configuration header file format.

Bit	31 ~ 16	15	14 ~ 1	0
Item	Transfer size	Memory/DSC	Starting address/ DSC sequential number	Read/Write

Because reading data from a memory often requires two execution cycles – control signal issues and the actual memory reading, descriptor handlings in FSM "execution" state are further pipelined in 3 stages to improve processing throughput. In the "DSC read" stage, function descriptors are sequentially read out, the corresponding memory bank reading controls are prepared, and a memory data write operation from the previous "DSC execution" stage (if any) is performed. The memory array receives reading command in "memory read" stage, and data content is pushed out in the same clock cycle. In the last stage "DSC execution", data reading and/or writing status in the allocated memory bank and data sending and/or receiving status in specified cell I/O ports are checked. Transactions are executed only when all required conditions are valid; otherwise the current descriptor operation is discarded.

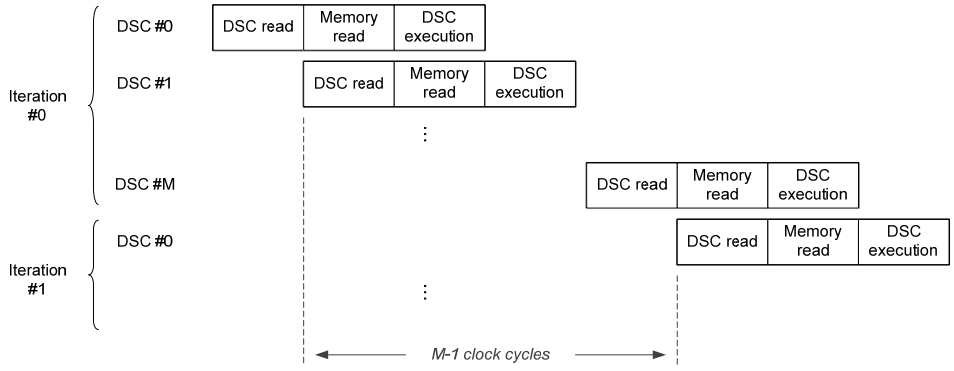


Figure 5-3. 3-stage pipeline processing in the memory descriptor handlings.

Figure 5-3 shows the memory descriptor handlings in a 3-stage pipeline process. Descriptors are executed in consecutive clock cycles, and each descriptor execution is iterated in M-1 clocks, where M is the length of a descriptor table. Because a descriptor defines an actual memory operation, empty descriptors will result in wasted execution cycles and hence processing throughput degradation.

5.1.2 Memory array

Since the memory array should be able to handle simultaneous data read and write operations, and it is better to have the memory cell operate at the same clock speed as the other building blocks, using a dual-port SRAM is hence the most straightforward approach. Because there is only one data bus inside the memory cell, one of the data ports (port A) from the memory array is dedicated for memory reading, and the other port (port B) only handles data write operations. By doing so, memory control signals are possibly simplified. Simultaneous data read and write operations at one memory location should in principle be avoided. However, the memory array in this project is configured to a read before write operation if that is desired.

5.1.3 Memory array considerations in an ASIC implementation

The memory array is an important unit that occupies most of the hardware area in a memory cell, especially when the data storage space increases. In a standard ASIC implementation, this can be realized either as a register file or as a data memory. For a small amount of memory, serially connected flip-flops are sufficient due to the small hardware area, namely the register file. As memory size increases, using the static random access memory (SRAM) often result in better performance in terms of hardware area and power metrics. Selecting the best solution between register file and SRAM is technology depended. For example, the dividing line is located at approximately 250 bits in a 0.35µm CMOS process [14], whereas the use of register file is still optimal for the Faraday UMC 0.13µm memories when storage size increases up to 64K bits, as shown in Figure 5-4.

From a previous ASIC implementation attempt based on a 0.13µm CMOS cell library ([1] pp. 137, Table 3), SRAM occupied about 65% of the total hardware area for a 8 Kb memory cell, and the area increased significantly to 83% when data storage went up to 32 Kb. Obviously, selecting a proper SRAM core has significant impact on design area and power consumption.

In the current memory cell implementation, a dual-port data memory is used as the memory array. This is sufficient for a Xilinx FPGA implementation, since a batch of dual-port block memories are provided as hardware resources inside the FPGA chip. However, in standard ASIC implementations, dual port memories are often more area and power consuming than single port memories. Taking the Faraday UMC 0.13µm memories [16] as an example, Figure 5-4 shows the different property comparisons between different SRAMs. Obviously, single-port memories are in general superior to dual-port memories. Therefore, a different approach than using the dual-port memory is highly recommended. This can be solved by using one single-port memory with double wordlength to hold two consecutive values in a single location, alternating between reading two values in one cycle and writing two values in the next cycle. This scheme requires temporary storage to synchronize the dataflow [14] and a structured access pattern.

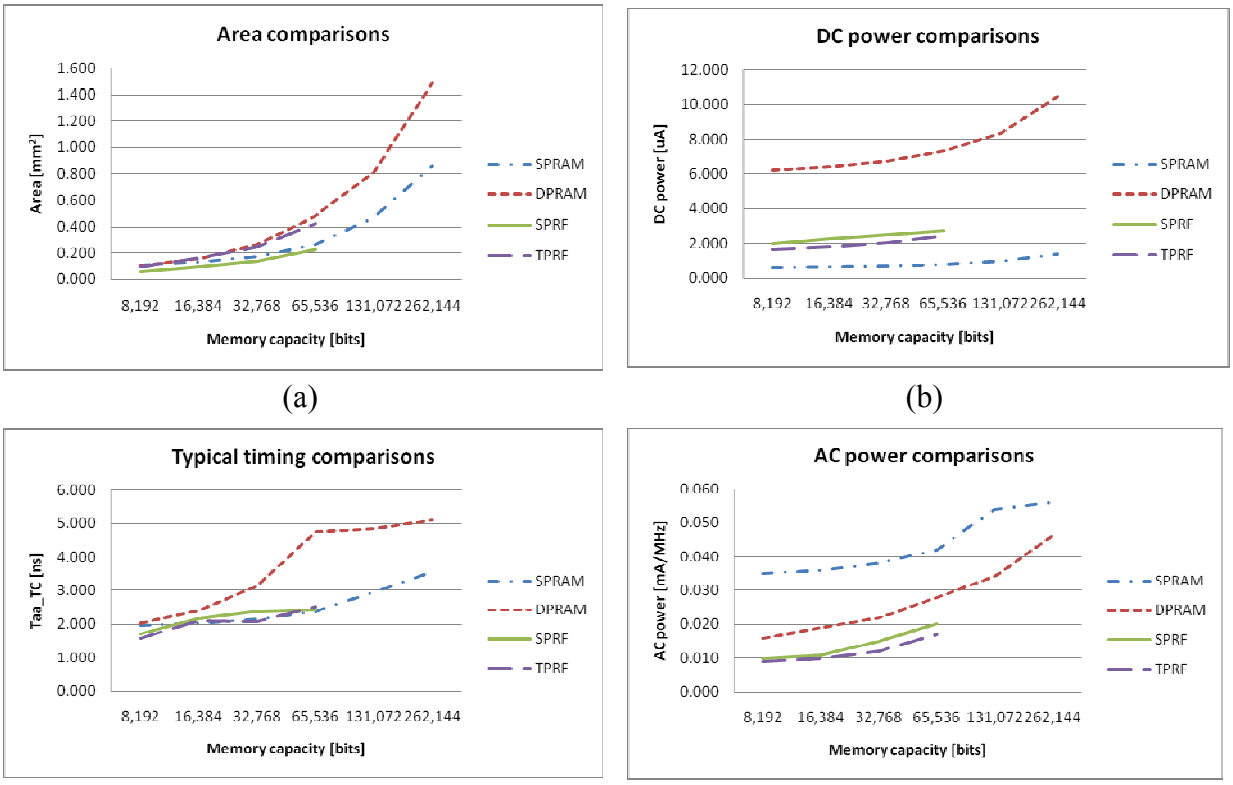


Figure 5-4. Faraday UMC 0.13µm memory core comparisons for storage sizes of 8Kb, 16Kb, 32Kb, 64Kb, 128Kb, and 256Kb. SPRAM, Single-Port RAM; DPRAM, Dual-Port RAM; SPRF, Single-Port Register File; TPRF, Two Port Register File. The SPRF and TPRF cores only support memory capacity up to 64Kb.

5.2 Memory descriptors and cell operations

5.2.1 FIFO mode

A memory operation in FIFO mode is identified by a descriptor type of '0' from the descriptor fields presented in Table 5-3. Using a *reading possible* and a *writing possible* flag, the FIFO status can be determined. A read possible and write impossible results in a full FIFO indication; a write possible and read impossible represents an empty FIFO. This running status is tracked by the operation controller each time the descriptor is executed. Memory areas can be reserved from the shared memory array by using a *base* and a *high* address. In FIFO mode, the allocated memory area operates as a circular buffer. Address pointers managed by the operation controller are used to indicate the current read and write positions, and are incremented respectively after each process. Conditions for being able to execute a FIFO descriptor are resolved from incoming and outgoing data and the current FIFO status. For example, writing data to a full FIFO cannot be executed until at least one data is read. The field *io_bank_rst* is used to flush out data packages stored in the cell I/O registers.

Worth mentioning is that the memory cells operating in FIFO mode can be

cascaded to form a larger capacity memory cell. This is a useful feature in a practical embedded system design, especially for the reconfigurable cell array architecture. Because the area and shape constraints for the physical placement are often made by system architect for each building block, data storage in memory cell is therefore restricted to meet this criterion. Cascading several memory cells together can then solve larger memory space requirements.

Table 5-3. 64-bit FIFO/Sequential ROM descriptor.

Field	Bits	Length	Description
dtype	63-62	2	Operation mode selection: FIFO mode = 0; Sequential ROM mode = 2.
rd_ok	61	1	FIFO/Sequential ROM reading status, 1: read possible.
wr_ok/ ---	60	1	FIFO writing status, 1: write possible. This field is reserved in sequential ROM mode operation.
src/ ---	59-56	4	Data source port ID, 0~7: LIO; 15: GIO. This field is reserved in sequential ROM mode operation.
dst	55-52	4	Data destination port ID, 0~7: LIO; 15: GIO.
id	51-42	10	GIO TX destination ID, only used when output through GIO.
base	41-32	10	Starting address in memory array.
high	31-22	10	Ending address in memory array.
rptr	21-12	10	Current FIFO reading pointer.
wptr/ ---	11-2	10	Current FIFO/Sequential ROM writing pointer. This field is reserved in sequential ROM mode operation.
io_bank_rst	1	1	Active high reset for memory IO bank, hardware releases reset automatically.
---	0	1	Reserved.

5.2.2 Sequential ROM mode

The term “ROM” in this case does not mean that memory region selected is not reconfigurable. This name is only used to distinguish the actual behavior of the memory bank from other operation modes, which could be used for storing constants, coefficients, etc.

Similarly, memory operation in the sequential ROM mode has been assigned with a descriptor type of ‘2’. The memory cell in this mode behaves like a write protected FIFO operation, where memory bank writing status, input data port definition, and the current writing pointer are discarded from the descriptor fields as shown in Table 5-3.

In contrast to the normal ROM operation (see section 5.2.3), address pointer in this mode is managed by the operation controller, and a consecutive data content is read out in each data transfer. Because the sequential ROM reading pointer is incremented after each valid execution, the memory host is in this mode exempt from sending data READ request.

5.2.3 RAM/ROM mode

Memory RAM/ROM mode operation handles data transfers in two phases. Firstly, the operation controller is triggered when the address port receives either a READ or a WRITE request (Table 5-4), which specifies a memory address and a transfer length. Secondly, data will start streaming from the memory array to the configured data port if the address port receives a READ request. Consequently, if the address port receives a WRITE request, data is fetched from the configured data port and stored in the memory array [1]. The descriptor keeps track of the transmission direction using "rnw", the transfer size using "tsize", the current memory address position using "ptr", and the current transfer status using a flag "active".

Table 5-4. Service request format.

Bit	**31 ~ 16**	**15 ~ 1**	**0**
Item	Transfer size	Starting address	Read/Write

Table 5-5. 64-bit RAM/ROM descriptor.

Field	**Bits**	**Length**	**Description**
dtype	63-62	2	Operation mode selection: RAM mode = 1; ROM mode = 3.
active	61	1	Active transfer flag.
rnw/ ---	60	1	Operation selection, 0: write; 1: read. This field is reserved in ROM mode operation.
paddr	59-56	4	Address port ID, 0~7: LIO; 15: GIO(0).
pdata	55-52	4	Data port ID, 0~7: LIO; 15: GIO(0).
id	51-42	10	GIO TX destination ID, only used when output through GIO.
base	41-32	10	Starting address in memory array.
high	31-22	10	Ending address in memory array.
tsize	21-12	10	Current data transfer size.
ptr	11-2	10	Current data pointer.
io_bank_rst	1	1	Active high reset for memory IO bank, hardware releases reset automatically.
---	0	1	Reserved.

5.3 Performance evaluation in FPGA

All possible configuration parameters for the memory cell have been listed in Table 3-1. However, due to the practical usage of the reconfigurable system, wordlength of the memory array and the number of global I/O ports are kept unchanged from defaults during the evaluation process. Notice that, internal memory array in the memory cell is realized by using dual-port block memories inside the FPGA device. Results from the maximum speed and the minimum area synthesis processes are compared in Table 4-6.

Table 5-6. Performance evaluation of the memory cell based on an FPGA device.

Configuration		**Maximum speed**		**Minimum area**		**Differences**	
		Slices	**f_{max} [MHz]**	**Slices**	**f_{max} [MHz]**	**Slices**	**f_{max} [MHz]**
GIO = 1; LIO = 8; Memory size = 32b×1024	DSC table length = 1	960	168.92	820	117.95	140	50.97
	DSC table length = 2	1,125	138.54	961	112.97	164	25.58
	DSC table length = 4	1,271	123.73	1,102	110.80	169	12.94
	DSC table length = 8	1,744	109.42	1,532	100.75	212	8.67
GIO = 1; DSC table length = 4; Memory size = 32b×1024	LIO = 2	772	128.60	799	117.21	-27	11.39
	LIO = 4	1,007	126.04	899	115.66	108	10.38
	LIO = 6	1,118	125.16	1,022	109.69	96	15.47
	LIO = 8	1,271	123.73	1,102	110.80	169	12.94
GIO = 1; LIO = 8; DSC table length = 4	Memory size = 32b×256	1,297	127.92	1,130	105.07	167	22.85
	Memory size = 32b×512	1,265	122.40	1,054	110.94	211	11.46
	Memory size = 32b×1024	1,271	123.73	1,102	110.80	169	12.94

Although value fluctuations can be observed from the gathered synthesis results, it is still possible to conclude that the two synthesis processes provide similar performance metrics. This should not be surprising, since not many arithmetical units are used in the memory cell implementation and there is only little room left for the design synthesizer to do area optimizations. This fact can be specifically emphasized in the first synthesis attempt. Because of the involved arithmetical units in each memory descriptor, such as the comparators, counters, etc., changing descriptor table length results in evident operating speed degradations and relative larger area requirements.

The critical timing path in the memory cell is found in the operation controller, more specifically in the pipeline state 3 of the FSM “execution” stage. In that state, the memory descriptor is updated under a batch of condition validations, therefore causes processing delays. Further design optimizations should be able to help upgrading system performance.

5.4 Conclusion

No embedded systems can be exempt from having at least one memory, either for data processing, program storage, or special usage. Using distributed smart memory cells in a system is an efficient way of handling data transfers between surrounding resource cells.

The memory cell architecture and four different operation modes have been presented in this chapter. With the use of a memory descriptor table, different operation behaviors can be emulated. Each descriptor reserves a memory space from the memory array and specifies stream transfers to be processed. The length of a descriptor table, memory array size, and the number of cell local I/O ports are compile-time configurable, while memory descriptors are run-time reconfigurable.

Based on the initial memory cell structure, a few design improvements have been done in this project, such as: the communication I/O modules have been modified to adapt memory cell into the cell array architecture; a few design optimizations on the operation controller have been made; and a new field *io_bank_rst* has been added into the descriptor table to enable software level cell reset ability.

5.5 Future work

5.5.1 Memory cell RAM and ROM mode operations

Due to the lack of time and the actual needs in the target algorithm implementations, such as the TMFIR described in Chapter 7 and the FFT presented in Chapter 8, the RAM and ROM mode operations in this project were abandoned from the initial memory cell specification. Hence, the first task in future work would be to have these modes operating in the memory cell. The RAM mode operation could be for instance used in FFT output data shuffling.

5.5.2 Memory cell processing throughput improvements

Although each memory host is allowed to use many function descriptors within a single memory cell, but in current implementation one memory operation cannot be sliced into pieces where each one is executed by a descriptor. For example, in reading coefficients from a memory cell running in ROM mode, only one memory descriptor is allowed to be used. Because memory descriptors can only be executed if all required conditions are validated, otherwise descriptor in turn will be discarded. Hence, if consecutive descriptors are configured for one stream transfer, receiving data samples will not be guaranteed to retain an expected sequence.

This usage restriction results in poor memory cell utilization if only one descriptor is configured in a memory cell, since all other descriptor cycles will be wasted. Two solutions are proposed here to address this problem. First, all empty

descriptors should be skipped to improve the memory usage. As a consequence, using this method will require additional checks on the content of memory descriptors, which would potentially increase the design's critical path, especially when the descriptor table length increases. The second approach is to add a blocking/non-blocking flag in each memory descriptor. If multi descriptors are used in the example above with blocking function activated, memory controller will be suspended if any stalling event occurs, hence maintaining the stream transfer sequences.

5.5.3 Debugging approaches

Function debugging in a memory cell could be realized in two ways. First of all, it should be able to retrieve memory descriptors during run-time. This could be used to check memory behavior configurations, to check the allocated memory bank, to track on the address pointers etc. Secondly, it might be a handy approach if data contents in the memory array can be partially or fully dumped during execution time. This could be used for instance in system logging, processor cell computing verification etc.

6 Router cell architecture

The resource cells communicate over local and global networks in the CGRA. The local interconnections provide high data throughput between neighboring cells, while the global network connects any two cells using hierarchical routing and provides an interface to external modules, such as the external host or data memories. Global communication is supported by router cells that forward data packets over a global network, as the peer-to-peer connections between any nodes result in difficulties on network scaling.

6.1 Cell architecture

The router cell is constructed from three main building blocks: *a decision unit* with a static routing table, *a routing structure* and *an output packet queue*, as illustrated in Figure 6-1.

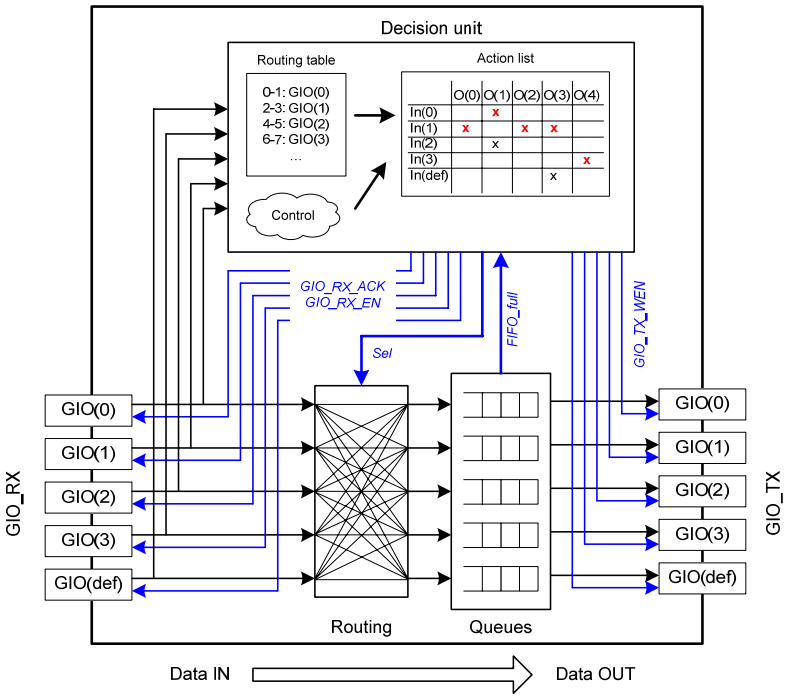


Figure 6-1. Block diagram of a router cell. Five global I/O ports and a parallel routing structure are used in this illustration.

The decision unit monitors incoming data packets from global input ports, looks up the routing path, handles data transactions and configures the routing structure to transfer data packets accordingly. In current architecture, data routing path is defined in a table, which is statically generated at system design-time based on the hardware connections in the CGRA. The static routing network is scheduled in a way that there is only one valid path from each source to each destination. This consequently simplifies the hardware implementation and enables each router instance to be

optimized individually during logic synthesis. However, a drawback with static routing is the network congestion, but mainly concerns networks with a high degree of routing freedom, for example a traditional mesh topology [1].

The routing structure is made up of passive connections between inputs and output ports, which can be implemented either by a parallel structure or a simple MUX-DEMUX switch, as illustrated in Figure 6-2. The complexity of the routing structure determines the routing capacity. A parallel routing structure has the ability of handling multiple data requests in a single clock cycle, while the single switch is limited to one transaction at a time [17]. However, the former approach is associated with a higher hardware area cost and requires a more complex decision unit [1].

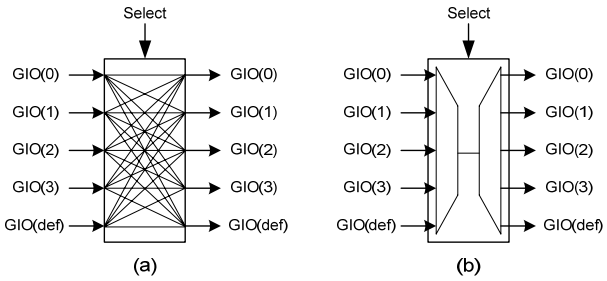


Figure 6-2. Architectural options for switching network implementation. Five global I/O ports are used in this example. (a) A parallel routing structure with full switching flexibility using five 4-1 multiplexes. (b) A simple network using one 4-1 multiplexer and one 1-4 demultiplexer.

The output packet queue is used to temporarily store packages traveling through the network [6]. When the incoming packet is accepted and handled by the decision unit, data sender is acknowledged and the packet is placed in the corresponding output packet queue if output buffer has free spaces. Data stored in the output queue are sequentially transmitted through global output ports on a first-in-first-out (FIFO) basis. The output buffer continuously tries to send its packets out, and removes them accordingly from the queue if transmission succeeded. When the output buffer is full, no new data can be accepted until the buffer has forwarded at least one of the buffered packets. The use of output packet queue increases global transmission throughput at the data sending side, because there is no need to wait for acknowledgement signals from receivers as long as the output buffer is not full. Therefore larger output buffer size is desirable under this circumstance. However, the long output packet queue requires more hardware resources and the use of output buffer causes additional transmission delay for the data receivers, as one clock cycle is required to just pass through an empty output queue. Therefore smaller output buffer size is wanted in this case, or even without any output buffer. To summarize, varying the length of output packet queue has impacts on global data communications, and the appropriate buffer size is one of the network parameters that has to be evaluated and adjusted based on the requirements.

The number of global I/O ports, network routing structure and the length of output packet queue are examples of design-time configuration parameters in the

router cell. Table 3-1 summarizes all possible parameters, where some of the concepts are further described in the following sections.

Table 6-1. Possible configuration parameters in the router cell.

Item	Value range	Default value	Configurability
Global I/O ports	≥ 1	4	Design-time
Default port selection	Enable, Disable	Enable	
Default port I/O	≥ 1	4	
Routing structure	Parallel, MUX-DEMUX	Parallel	
Arbitration policy*	Fixed, Round-robin	Fixed	
Output buffer (FIFO) size	≥ 1	4	
Routing table	Depending on the number of GIOs	Invalid routing	

Note: * Only effective in the MUX-DEMUX routing structure.

6.1.1 Routing table

Based on the global network connections in the CGRA, the routing table is statically generated for each of the router cells. The routing information is represented in the form of an array of integer pairs, where each pair defines a global ID range for input packet acceptances. A range pair contains a base and a high values, such as the notation "port 0 -> ID 0 - 3" means that I/O port 0 in the router cell accepts data packets that have destination ID specified in the range from 0 to 3. As an example, port 0 is the data forwarding port for a packet with destination ID equals to 2. If there is no range pair that accepts incoming packets, and the router is configured with a default port, this default port will be used as the packet destination. The special value “-1” (all ones) corresponds to an unspecified invalid port, which can be used if a port never accepts packets [17]. In reference to the global routing network shown in Figure 6-3 (a), examples of the routing table are shown in Figure 6-3 (b).

6.1.2 Decision unit and arbitration policy

The decision unit consists of two combinatorial processes that are operated in parallel: the transmission management and transaction handling processes. Gained from this design structure, all transmission events can be easily managed and traced, and the handling of data transactions can be flexibly controlled by applying different arbitration policies, such as the fixed and Round-robin schemes.

In the transmission management process, data packets arrived at input ports are continuously checked. Validity of the packet is confirmed by identifying the flow control signal “valid” specified in the global packet format as defined in 2.3.2 and is repeated here as shown in Figure 2-7.

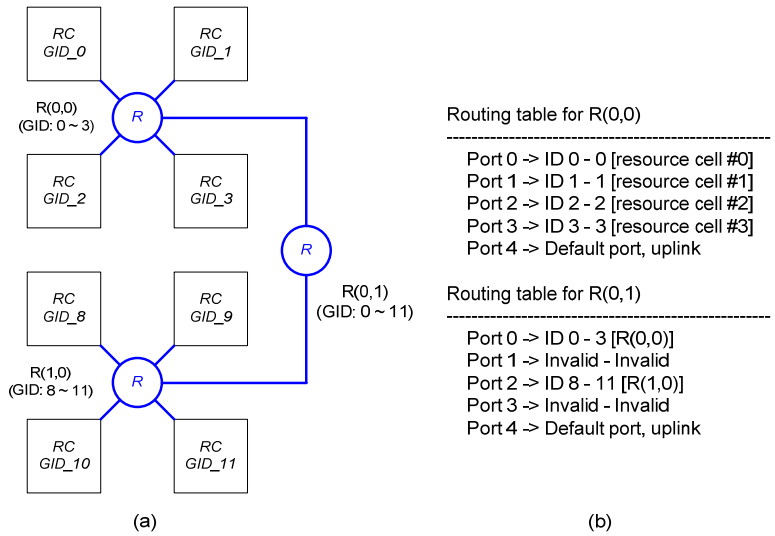


Figure 6-3. (a) Global routing network in a 4-by-2 cell array. Each resource cell (*RC*) has been assigned with a network ID (*GID*). (b) Static routing table defined for the router cell R(0,0) and R(0,1).

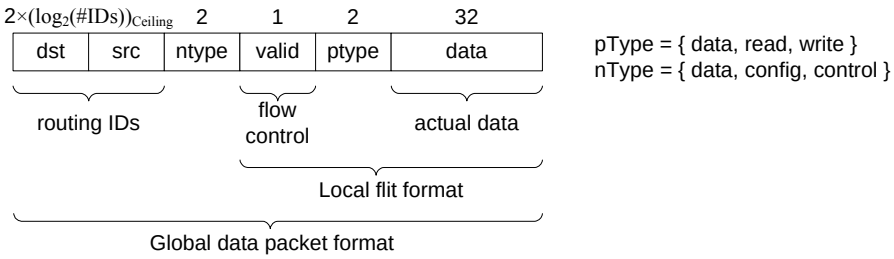


Figure 6-4. Network packet format of local and global data transmission.

The destination ID specified in each valid packet is thereafter evaluated and looked up in the routing table. Data routing path is determined if the destination ID of incoming packet falls into one of the value pairs defined in the routing table, otherwise data packet is sent to the default port which is upwards in the hierarchical routing network. The default port option is configurable during design-time, which can be used to disable the uplink in a router cell if data communications are desired to be kept within certain network hierarchal levels.

Before registering new transactions into the action list, additional historical check on the corresponding log entry is performed. Transaction can only be registered into the action list if no preceding data transmission is ongoing. This is used to prevent data sending duplications, as the communication acknowledgement for the incoming packet is updated in consecutive clock cycle, which consequently keeps the incoming "valid" signal high during two clocks and therefore triggers an extra data transaction process. Timing diagram of a transaction handling is illustrated in Figure 6-5, where the active data transaction is masked with a write protection flag that is used in the historical check in order to avoid the duplicated data sending. An example of how the action list would look like after the transmission management process is shown in Figure 6-6 (a).

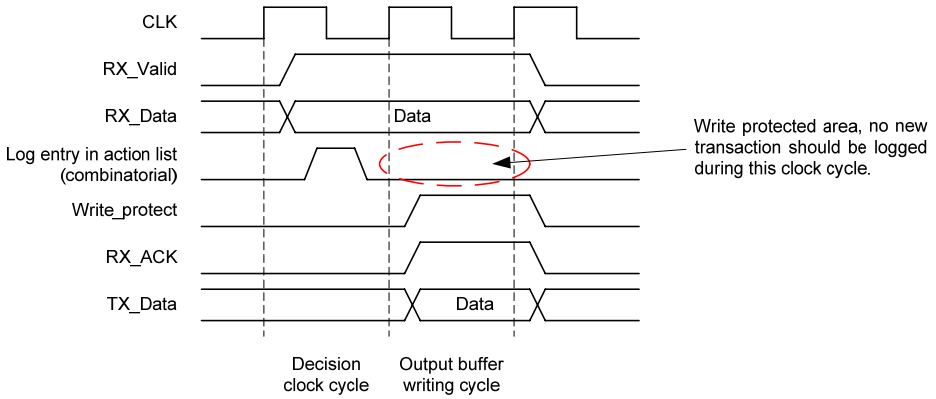
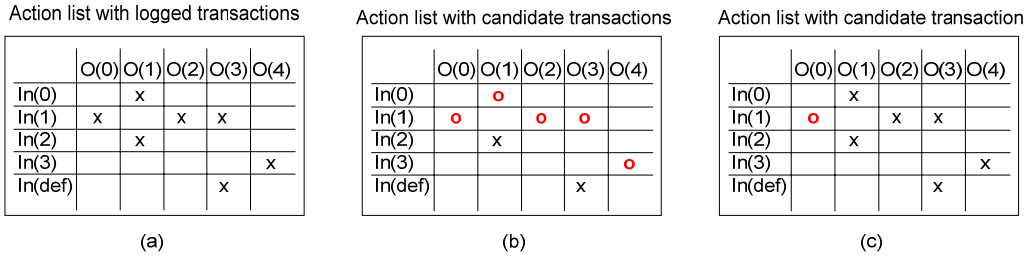


Figure 6-5. Timing diagram of the transaction handling in the router cell.



	O(0)	O(1)	O(2)	O(3)	O(4)
In(0)		x			
In(1)	x		x	x	
In(2)		x			
In(3)					x
In(def)				x	

	O(0)	O(1)	O(2)	O(3)	O(4)
In(0)		o			
In(1)	o		o	o	
In(2)		x			
In(3)					o
In(def)				x	

	O(0)	O(1)	O(2)	O(3)	O(4)
In(0)		x			
In(1)	o		x	x	
In(2)		x			
In(3)					x
In(def)				x	

Figure 6-6. Action list in the decision unit. The row and column items in the table are input ports (*In*) and output ports (*O*), respectively. (a) All valid data transactions are checked in during the transmission management process. (b) All candidate transaction handling are marked in ‘o’ during the second process. Fixed arbitration scheme in a full parallel routing structure. (c) Fixed/Round-robin arbitration scheme in a MUX-DEMUX routing structure.

In the transaction handling process, log entries in the action list are sequentially checked, and the recorded transactions are prioritized and handled based on different arbitration policy and the condition of output buffers. The fixed arbitration approach is used when a parallel routing structure is selected, while two different arbitration options are provided for the simple MUX-DEMUX routing structure, namely the fixed and Round-robin arbitration.

In fixed mode, the arbiter always starts from the first log entry (row 0, column 0) and scans column-wise through the action list until a candidate transaction is found. A transaction is considered to be a candidate when it is logged in the action list and the corresponding output buffer is not full. In this approach, all transactions are assigned with priorities according to their log position in the action list. Consequently, this might result in a case where the shared transmission channel is perpetually occupied by a data flow that has higher priority than the others. Hence, network traffic has to be well schedule when mapping applications on the CGRA if the fixed arbitration policy is applied.

In contrast, the Round-robin algorithm assigns time slices to each process in equal portions and in order, handling all transactions without priority [4]. This simple arbitration approach is work conserving, meaning that an empty transmission cycle

will be resulted if one flow is out of packets. Examples of candidate transactions in the action list are shown in Figure 6-6 (b) and (c).

The candidate transactions selected out from arbitration process are thereafter checked out from the action list, and the routing structure is configured and the data senders are acknowledged accordingly. This way of handling the communication acknowledgement signals prevents data loss from the network congestion, as data senders cannot proceed without receiving ACK. signals.

A data broadcast function is provided in the simple MUX-DEMUX routing structure. By addressing "the largest resource cell ID number – 1" as the packet destination, data received from one input port is sequentially forwarded to all the output ports available. This is achieved by sending acknowledgment signal to the data sender only when no more logged transactions are left in the row of the action list, where the row items specifying data input ports, and the output ports are specified column-wise.

6.2 Performance evaluation in FPGA

An analysis of global data transmission latency is presented in Figure 6-7, where a data packet is assumed to be forwarded by a router cell connected between host A and B in a silent global network. The data transmission latency is measured from the time when host A places a packet to its global output register to the time when the global input register in host B receives the packet from network. The resulting time usage is 5 clock cycles, which uses 3 clocks more as compared to local data communication.

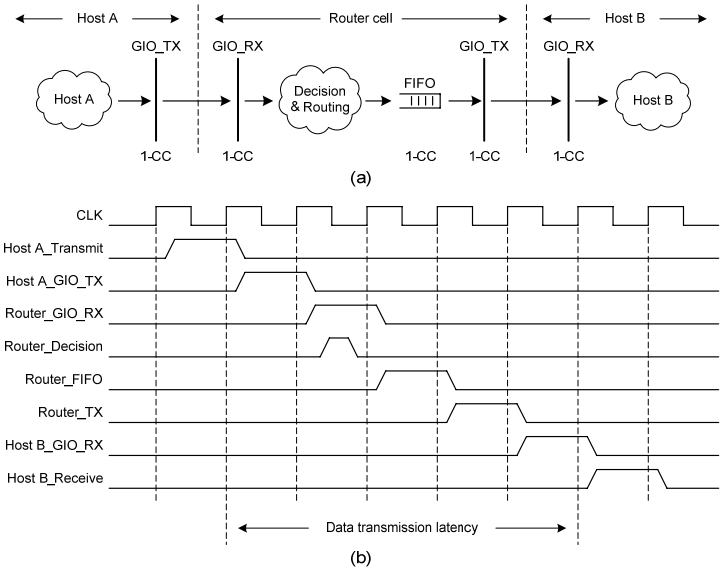


Figure 6-7. Data transmission latency over the global routing network. No initial data traffic is assumed in the routing path. (a) Hardware setup for transmitting a data packet from host A to B. Global I/O registers are drawn in vertical lines, and the clock usage for the corresponding unit are stated underneath, as x-CC. (b) Detailed timing diagram for a global data transmission.

Router cell with different configurations have been synthesized with both the maximum speed and minimum area constraints. Considering the realistic system setup, two parameters are kept as constants during the synthesis processes, i.e. the default port (uplink) is always enabled and assigned with the maximum global I/O port number. All results are listed and compared in Table 3-10.

Table 6-2. Performance evaluation of the router cell based on an FPGA device.

Configuration			Maximum speed		Minimum area		Differences	
			Slices	f_{max} [MHz]	Slices	f_{max} [MHz]	Slices	f_{max} [MHz]
Def. port = Enable; Def. GIO = Max. GIO; Routing = Parallel.	#GIO=5	#FIFO=4	1,340	174.51	1,151	160.99	189	13.52
		#FIFO=8	2,297	168.15	2,066	128.12	231	40.03
	#FIFO=4	#GIO=5	1,340	174.51	1,151	160.99	189	13.52
		#GIO=10	4,139	161.95	3,761	141.39	378	20.57
Def. port = Enable; Def. GIO = Max. GIO; Routing = MUX-DEMUX; Arbitration = Fixed.	#GIO=5	#FIFO=4	1,282	121.99	1,133	53.31	149	68.68
		#FIFO=8	2,245	113.20	2,048	47.03	197	66.17
	#FIFO=4	#GIO=5	1,282	121.99	1,133	53.31	149	68.68
		#GIO=10	2,941	61.97	2,396	15.85	545	46.13
Def. port = Enable; Def. GIO = Max. GIO; Routing = MUX-DEMUX; Arbitration = Round-robin.	#GIO=5	#FIFO=4	1,243	173.02	1,139	149.43	104	23.59
		#FIFO=8	2,189	142.03	2,053	120.31	136	21.72
	#FIFO=4	#GIO=5	1,243	173.02	1,139	149.43	104	23.59
		#GIO=10	2,561	151.13	2,208	137.68	353	13.45

Several points can be concluded from these synthesis results. First of all, the highest cell operation speed in each set of the configurations is achieved in the parallel routing structure. As mentioned previously, this is a result from concurrent processing on multiple data transactions, where all input packets heading to different output ports can be handled in a single clock cycle. Therefore, the critical timing path in this structure is limited to the arbitration process for multiple transactions on the same output port, i.e. columns in the action list as shown in Figure 6-6 (b). The drawback of the parallel routing structure is the accompanied large hardware resource usages.

Secondly, in the MUX-DEMUX routing structure, with the use of round-robin arbitration approach results in less hardware usages and higher operation speeds than using the fixed mode. This is because transaction handling in each clock cycle is deterministic in the round-robin scheme, which simplifies the operation controlling and shortens the critical timing path. In contrast, the fixed arbitration scheme scans through the entire action list in each arbitration process until a candidate transaction is found. This results in a long packet handling procedure and hence the slower operation speed. Moreover, due to the combinatorial processing in the fixed arbitration scheme, large speed differences are reported from these two synthesis processes.

6.3 Conclusion

The router cell is used to forward data packets over the global routing network in the CGRA. The routing information is specified in a table that is statically generated based on the physical network connections. The static routing is deterministic, which means that there is only one single valid path to route network traffic. This approach reduces hardware complexity as compared to adaptive routing algorithms [2].

The router cell consists of three main building blocks: the decision unit, routing structure and output packet queue. Basic cell construction parameters, such as the number of global I/O port, the routing structure and so on, can be configured during system design-time in order to meet different application requirements.

The separation between transaction handling process and transmission event managements in the decision unit results in a flexible design structure, where different arbitration policies can be easily implemented and extended in future developments. In current router cell implementation, the routing structure can be selected between a parallel network and a MUX-DEMUX switch. A parallel routing structure has the ability of handling multiple data requests in a single clock cycle, while the latter one is limited to one transaction at a time. For the MUX-DEMUX structure, two simple arbitration policies are supported and a data broadcast function is provided, while the parallel structure is configured to only use the fixed arbitration policy. Benefit from the way of handling acknowledgement signals in the decision unit, data loss due to the network congestion is completely prohibited.

6.4 Future work

6.4.1 Multicast

In addition to packet broadcast function, the multicast would also be beneficial to have in the CGRA, which is useful for instance in initializing a batch of memory cells or to issue global operation commands to the processor cells and so on. This is can be realized by applying a unique global ID address to each group of the resource cells, so data packets with group ID specified will be forwarded to all the cells included in the group.

7 Case study I: Time-multiplexed FIR filter

After creating each of the individual modules in the CGRA, a system integration test should be carried out. This is used to verify the basic functionalities in each resource cell, to test the interconnections between neighboring modules and to verify the connectivity of the hierarchical routing network. This test has been accomplished by mapping a small algorithm onto the cell array platform. A time-multiplexed FIR filter has been selected in this study due to the simple algorithm structure and relatively low system resource requirements. In addition, the author has done similar work in a related project based on a single processor and a single memory cell, hence it is also time efficient to reuse the same experiment and only expand the work to a more complex system platform.

Figure 7-1 shows a generalized design flow for mapping an algorithm onto the CGRA. This flow consists of four design phases: algorithm selection, reference model design, cell array architecture design and the actual system implementation. As illustrated in Figure 7-1, the third design phase involves additional sub-steps, where the selected system architecture is designed and evaluated. This is the most time-consuming procedure, since different mapping possibilities should be analyzed and compared, in order to efficiently map an algorithm onto the platform. In the last step, the system is implemented and tested physically on a hardware platform, such as an FPGA device.

In this case study, the actual system implementation process is replaced by a cycle-accurate HDL simulation running on a computer, since the intention of the experiment is to verify the basic operations in the CGRA. Besides, some of the design procedures are simplified due to the chosen simple algorithm structure, such as only one hardware-mapping approach is used in this study.

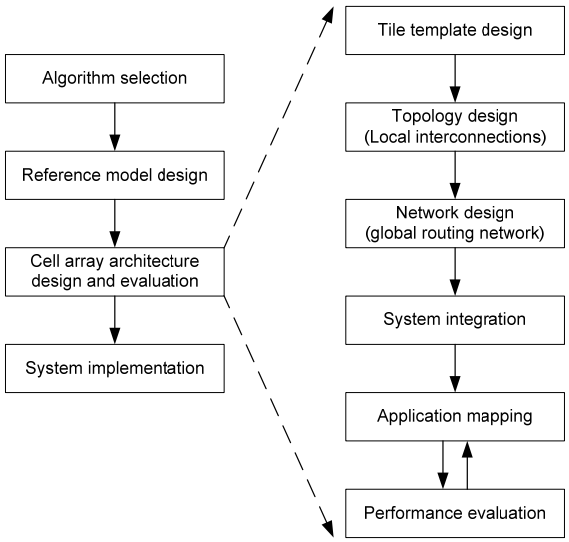


Figure 7-1. Generalized design flow for mapping an algorithm onto the CGRA.

7.1 Theoretical background

A finite impulse response (FIR) filter is a discrete linear time-invariant (LTI) system. The current filter output only depends on the current data input and its delayed input samples. This can be expressed as

$$y(n) = \sum_{i=0}^{m} h_i \cdot x(n-i), \qquad \text{eq. 7-1}$$

where the filter is non-recursive with an order m. Its impulse response $h = \{h_0, \ldots, h_m\}$ is limited to $m + 1$ taps [19].

An FIR filter can be implemented in several ways of which two are illustrated in Figure 7-2. First, a direct form hardware-mapped structure processes input data samples concurrently. It requires one multiply-accumulate (MAC) unit for each filter coefficient, which are serially connected to form a pipeline. Each unit multiplies the input value with the coefficient, adds the partial sum from the preceding stage, and forwards the data value and result to the next stage [1]. Due to the way of the concurrent processing, this structure provides high data throughput but requires large amount of hardware resources. In contrast, a time multiplexing approach requires only one MAC unit and two memory cells, one for coefficients and the other for input data buffering. All data samples are iteratively processed using the same MAC unit. Obviously this structure reduces the total hardware requirements especially for higher order filters, but with the penalty of increased system computation time. Furthermore, a time-multiplexed structure also exhibits a good property on the system flexibility, since no hardware structure changes are needed when varying the filter order.

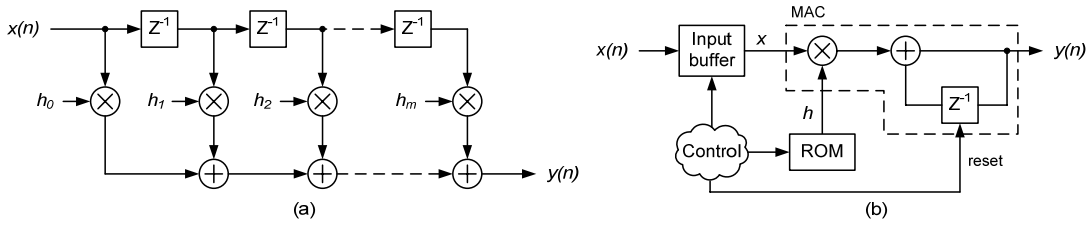
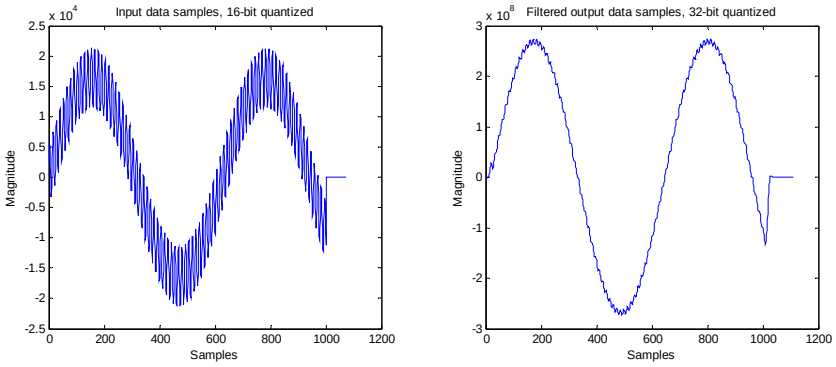


Figure 7-2. Hardware implementation structure of a FIR filter. (a) A direct form. (b) A time multiplexing approach.

7.2 MATLAB reference model simulation

The system test is based on a time-multiplexed FIR filter. Input data samples, impulse response and the filter output golden vector are all reused from the previous experiment. As previously described the filter coefficients and input data samples are quantized to 12-bit and 16-bit, respectively. The filter output is represented by 32-bit and overflow is prevented by proper scaling of the input data magnitude. A simulation using the MATLAB reference model is shown in Figure 7-3.



(a) | (b) Figure 7-3. Reference model simulation in MATLAB. (a) 16-bit input data samples. (b) 32-bit filter outputs.

7.3 System architecture

A time-multiplexed FIR filter consists of one MAC unit, one input data buffer and one coefficient ROM, as illustrated in Figure 7-2 (b). This structure can be directly mapped onto the CGRA by using a MAC processor and two memory cells. If the system resource usage is a main constraint, the input data buffer and the coefficient ROM can be realized by using two memory descriptors from the same memory cell. The latter approach has been used in a previous experiment. A direct mapping structure is selected in this study, in order to verify the interaction between different resource cells, as shown in Figure 7-4.

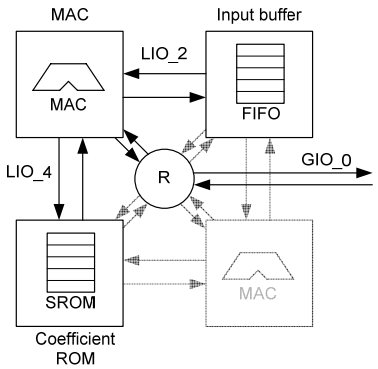


Figure 7-4. A tile template for direct-mapped time-multiplexed FIR filter on the CGRA. For illustration purpose, unused building block and network connections are drawn in lightened shade.

Input and output data samples are streamed through the global network. All intermediate data transfers are kept locally through the local interconnections. One of the memory cells operating in FIFO mode is used as a circular buffer for the input data samples. Filter coefficients are stored in the other memory cell which is configured in sequential ROM mode. The MAC processor iterates over the buffered data samples and coefficients that are multiplied pair-wise and accumulated. Every time a new iteration starts, the oldest value in the circular buffer is discarded and

replaced by an incoming data sample from the global input port. The lower 32 bits from the 48-bit accumulator inside the MAC processor are written to the output when an iteration completes, since input data samples are pre-scaled in the data source. In case output scaling is desired in a process, data can be shifted before sending it to the output port. With the use of the local and global communications, all data samples required by the MAC processor are synchronized automatically, since no new data transfer can be started before the preceding package is completely handled. This scheme provides an efficient manner of controlling the data flow in a processing intensive application.

7.4 System performance evaluation

Before HDL simulation, system platform is extended with the interconnected resource cells apart from the tile template designed previously. This is considered firstly to explore the large-scale system performance in terms of area and speed. Secondly, to have a suitable system architecture prepared for more complex algorithm mappings, such as an FFT application evaluated in case study II. More detailed system descriptions are presented in Chapter 8, as the main focus here is a basic functionality test. System architecture of a 4-by-2 cell array is depicted in Figure 7-5. A view on the synthesis floorplan and the final routed layout is shown in Figure-Appendix 10, based on the target FPGA device, the Xilinx Virtex-II Pro-30-7ff896.

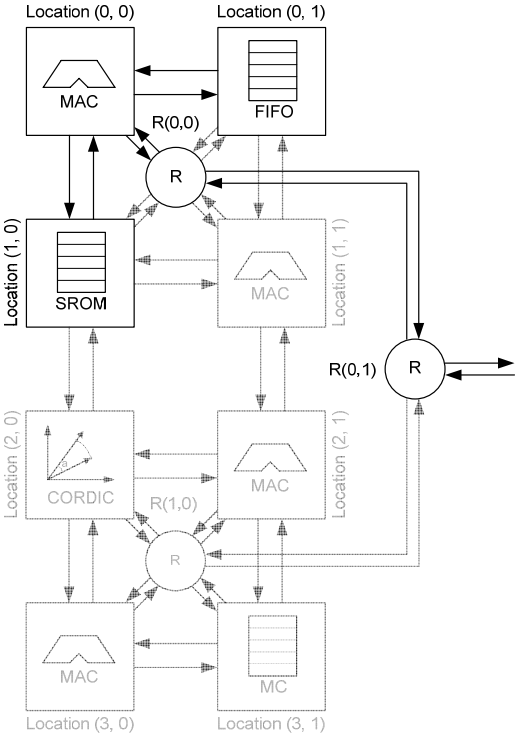


Figure 7-5. System architecture of an interconnected 4-by-2 cell array. Three resource cells and two hierarchical network routers are used in the time-multiplexed FIR filter.

To verify the filter operations, a program routine is developed for the MAC processor, as shown in Table 7-1. Notice that, only one program instruction is required to be re-downloaded when changing the filter order. By counting the execution time for each of the instructions in this program routine, a theoretical iteration period for this algorithm implementation can be obtained, as

$$cc_{iteration} = 1 \cdot 4 + 3 \cdot 36 + 2 + 4 = 118[cc]. \qquad \text{eq. 7-2}$$

The first term "1×4" sums up initial program execution cycles from instruction #3 to #6, where the first two instructions are not counted, as they are only used to initialize internal registers and are not iterated in each of the filter computations. The term "3×36" counts up the iterative executions of the instruction #7 and #8, where an additional clock cycle is used to flush the processor pipeline registers. Number "2" in the expression is the clock usage for the last loop execution, where no flush operation is needed. The last part "4" includes execution cycles for the rest of instructions, where the branch instruction requires one additional clock cycle for the pipeline register flushing.

Table 7-1. Program instructions in the MAC processor for a time-multiplexed FIR filter. Table columns are: binary code, assembly program, comment and reconfigurability.

84400024	// 01: ADDI $2, $0, 36	; Load filter order	**For reconfig.**
B000000A	// 02: GID 10	; Load GIO TX DST ID	
10000000	// 03: MUL $0, $0	; Clear acc. registers	
84350000	// 04: ADDI $1, $L2, 0	; Remove one data from FIFO	
86BB0000	// 05: ADDI $L2, $G0, 0	; Load a new data into FIFO	
40001000	// 06: ILC $2	; Load iteration counter	
1EA1AD40	// 07: DMOV $L2, $R1, $L2, $L2	; Load data from FIFO	
1000B843	// 08: MUL $L4, $1, {al}	; MAC and loops back	
00000000	// 09: NOP		
1B60FF80	// 10: JMOV $G0, $HACC, $LACC	; Send result out	
A400FFF7	// 11: BRI -9	; Loop back	

Table 7-2. Hardware configurations for resource cells in a 4-by-2 reconfigurable cell array platform.

Location	Global ID	Cell type	Hardware configurations
(0, 0)	0	Processor	16-bit MAC processor; Barrel shifter disabled; Program memory size = 64 × 32-bit; GPR = 8; LIO = 8; GIO = 1.
(1, 1)	3		
(2, 1)	5		
(3, 0)	6		
(0, 1)	1	Memory	Memory capacity = 256 × 32-bit; DSC table length = 4.
(1, 0)	2		
(3, 1)	7		
(2, 0)	4	CORDIC	16-bit pipelined core

Based on the suggested configurations [1], each memory cell is configured to have 4 function descriptors and an 8K bits memory capacity. Notice that, empty execution cycles are generated in each of the memory cells during run-time, since only one function descriptor is actually required in the FIR filter implementation. Table 7-2 summarizes the hardware configurations for all resource cells in a 4-by-2 cell array.

System performance numbers from the HDL simulation are shown in Table 7-3. The reported maximum frequency for the integrated system reveals a slower running speed than the critical building block in a 4-by-2 cell array. By tracking the synthesis report, signals from all possible processing cells are shown in the longest delay path. This implies that there are combinatorial signals connected in between those cells, which have to be fixed in future work.

Two execution time measurements are reported in the performance summary, as highlighted in bold fronts. When the descriptor table length (DSC) in each memory cell is configured to 4, system performance is limited by the clock cycle overhead in the memory cells. This can be observed from the data receiving stalls in the MAC processor. By adjusting the descriptor table length until the RX stalls are completely eliminated in the MAC processor, an optimal execution clock usage can be measured. This happens when the descriptor table length is decreased to 2. Comparing to the theoretical iteration period calculated previously, both local and global network communications are proven to be efficient.

Table 7-3. System performance explorations for the CGRA based on the different TMFIR computations.

FIR filter order	36	
Memory [bits]	16K (8K × 2)	
Memory utilization **(Memory usage [bits])**	6.32% (592 + 444)	
Maximum frequency [MHz]	30.710	
FPGA usage on Xilinx Virtex-II Pro-30-7ff896	Number of slices: 10,460 out of 13,696 – 76% usage; Number of MULT18X18s: 4 out of 136 – 2% usage; Number of BRAMs: 7 out of 136 – 5% usage	
Configuration time [clock cycles]	581	
Reconfiguration time [clock cycles]	28	
Latency [clock cycles]	173 @ DSC = 4	134 @ DSC = 2
Execution time [clock cycles]	***152 @ DSC = 4***	***118 @ DSC = 2***
Throughput [samples per second]	200,720	

7.5 Conclusion

A time-multiplexed FIR filter has been mapped onto a 4-by-2 reconfigurable cell array. Basic operations of the resource cells have been verified based on cycle-accurate HDL simulations. The outcome from the performance evaluation has proven the effectiveness of the network communications in the CGRA. The large-scale cell array architecture has been prepared for further system development, where the FIR filter experiment utilizes only one 16-bit MAC processor, two memory cells and two hierarchical network routers. The integrated 4-by-2 cell array occupies 76% of the FPGA resources, and is capable of running at approximately 30 MHz.

8 Case study II: Radix-2^2 FFT

In the preceding Chapter 7, a few basic operations in resource cells, local data communications, and the hierarchical global routing network have been studied and evaluated by mapping an algorithm onto a small region of the reconfigurable cell array. In this second case study, it is desirable to scale up the system resource usage to further explore interaction between different resource cells, and to analyze the effectiveness of the data communications on a large-scale system platform. Hence, the radix-2^2 FFT algorithm has been selected and mapped onto a 4-by-2 cell array based on a time-multiplexed structure. Finally, the design has been implemented and verified on a Xilinx XUP Virtex-II Pro development board.

8.1 Theoretical background

The discrete Fourier transform (DFT) is a commonly used operation in digital signal processing. Typical applications are linear filtering, correlation, spectrum analysis, and orthogonal frequency division multiplexing (OFDM) in modern communication systems [14].

The DFT is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot \omega_N^{k \cdot n} \qquad 0 \le k \le N \qquad \text{eq. 8-1}$$

where N is the transform length and

$$\omega_N = e^{-j \cdot 2 \cdot \pi / N} \qquad \text{eq. 8-2}$$

Evaluating eq. 8-1 requires N MAC operations for each transformed value in X, or N^2 operations for the complete DFT [14]. This time-complexity of $O(N^2)$ is known as an inefficient way to carry out computations.

The fast Fourier transform (FFT) exploits the symmetry and periodicity properties of the phase factor ω_N [21], which decomposes an N-point DFT into successively smaller DFT transforms based on a divide-and-conquer approach. This results in a $O(N \times \log_2(N))$ complexity. According to the different partition approach, the FFT decomposition can be classified into decimation-in-time (DIT) and decimation-in-frequency (DIF) algorithms. No matter which kind of algorithm to use, the basic FFT operation is adding and subtracting the same two values, which is referred to as a butterfly operation due to its butterfly-like shape in the flow graph [14].

When the transform length is a power of 2, $N = 2^q$, the processing data sequence can be decomposed into two series using a radix-2 butterfly. This radix-2 algorithm requires q decomposition steps, each computing $N/2$ butterfly operations. When the transform length is a power of 4, $N = 4^q$, a more hardware efficient radix-4 decomposition algorithm can be used. This approach reduces the number of complex

multiplications with the penalty of increased complex additions. The complex radix-4 butterfly can be further simplified by using four radix-2 butterflies as shown in Figure 8-1, which reduces the number of complex additions needed. This reduction is evident when folding on a flow graph is applied. A comparison between three decomposition algorithms mentioned above has been originally compared in [14] and is repeated here as shown in Table 8-1 in section 8.3.

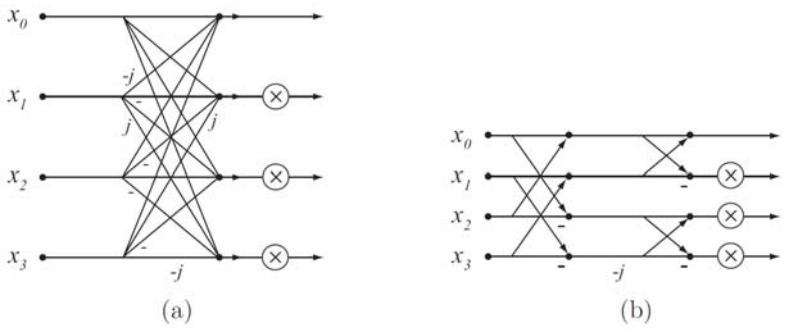


Figure 8-1 [14]. (a) Radix-4 butterfly. (b) Radix-2^2 butterfly.

Because data rounding or truncation is unavoidable operation in a fixed-point hardware implementation, the signal to quantization noise ratio (SQNR) is often used as one of the parameters to evaluate the system performance. It is defined as,

$$SQNR_{dB} = 10 \cdot \log_{10}\left(\frac{P_x}{P_q}\right) \qquad \text{eq. 8-3}$$

where P_x is the signal energy and P_q is the quantization energy.

8.2 MATLAB reference model simulation

To comprehensively understand the algorithm structure and intermediate data flows, a full precision reference model for a radix-2^2 pipeline FFT is initially designed in MATLAB. Thereafter, a fixed-point model for a time-multiplexed structure is developed to further study the quantization and overflow effects in a hardware platform. Because the complex multiplication function is realized by using a 16-bit CORDIC processor in an actual system implementation (refer to section 8.3.2), a bit-accurate simulation model for the CORDIC cell is developed in MATLAB and used in the fixed-point modeling.

In the simulation environment, the user can specify the input data width, intermediate calculation precision, overflow scheme and CORDIC configurations, etc. Furthermore, the user selects FFT size and generates input data samples. Figure 8-2 shows a 1,024-point FFT simulation comparison between the MATLAB built-in function and a fixed-point reference model. In this example, the input data width is 10 bits and all internal arithmetic operations in the fixed-point model are carried out with 16 bits precision.

Notice that the problem of internal computation overflow is not considered a topic for hardware implementations in this experiment. To prevent that, system data

inputs are restricted to a maximum of 10-bit complex numbers with scaled (×*0.5*) signal magnitudes. Complex valued data stimuli for the hardware platform are currently generated from MATLAB, where the reference model simulations can be carried out in advance to check the internal computation status.

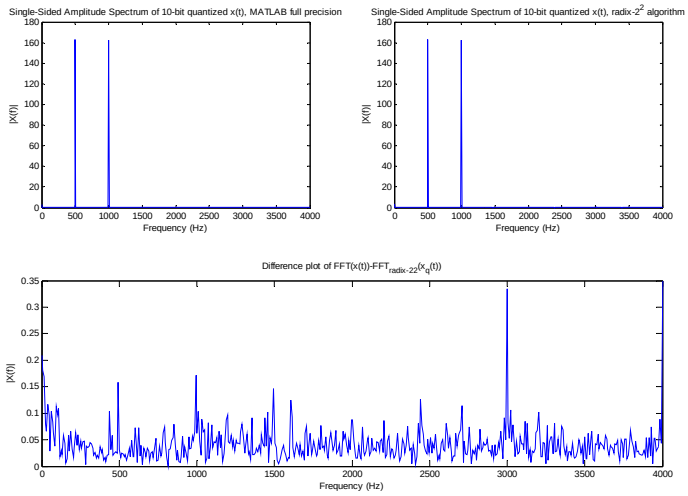


Figure 8-2. A 1,024-point FFT output comparison between the MATLAB built-in function (upper left) and a fixed point reference model (upper right). Result differences are shown in the bottom plot. The SQNR for the MATLAB built-in function full precision simulation is 40.122 dB; and the SQNR for the fixed-point reference model is 39.928 dB.

8.3 System architecture

The hardware implementation of the FFT algorithm can be mapped in several ways. In general, mapping schemes can be sorted into three categories: a direct algorithm mapping, a pipeline structure and a time-multiplexed approach. Direct algorithm mapping basically implements every processing unit in a flow graph using a unique arithmetic unit. Normally, using this approach for a large and complex algorithm is not desirable due to the huge amount of hardware resources required [1]. The alternative is to fold operations onto the same block, which reduces hardware complexity at the cost of increased computation time.

Taking a 4-point radix-2 FFT (Figure 8-3) as an example, folding the FFT algorithm vertically is referred to as a pipeline FFT, which reuses computational butterfly units at each stage. Because of the hardware resource sharing, a part of the input data samples at each stage need to be temporally stored before any computation can be performed. The required data storage is referred to as a single path delay feedback (SDF) buffer. As a penalty, computation time required in a pipeline FFT is dependent on the transform length, where a direct-mapped structure streams data in each clock cycle.

Folding the pipeline structure further horizontally reduces the arithmetic unit requirements to only one butterfly, results in a time-multiplexed architecture. Because

this approach folds the algorithm in two dimensions, the FFT computation stage quantity also becomes a factor that influences the required execution time. This architecture has the same memory requirements as a pipeline FFT. Property comparisons between these FFT architectures are shown in Table 8-1. Due to the great computation complexity reduction and the smallest hardware resource requirements, the radix-2^2 FFT architecture is selected in this study to be mapped on the CGRA.

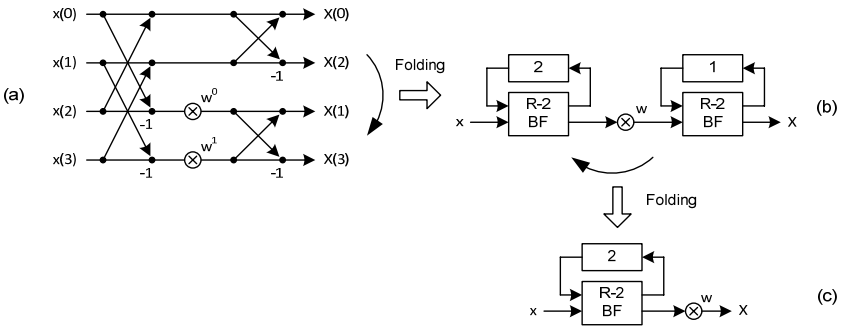


Figure 8-3. "Folding" a 4-point radix-2 FFT. (a) A flow graph. (b) A pipeline structure is built up by using two radix-2 butterfly units (R-2 BF) with SDF buffer attached, and a complex multiplier. The number stated inside SDF block represents the FIFO depth. (c) A time-multiplexed structure uses a single radix-2 butterfly unit and a complex multiplier.

Table 8-1 [14]. Properties for different FFT architectures. Multipliers and adders are complex valued. The number of clock cycles depends on the transform length N.

Hardware architecture	Adders	Multipliers	Memory	Cycles
Direct-mapped radix-2	$N\log_2 N$	$(N/2)(\log_2(N) - 1)$	0	-
Direct-mapped radix-4	$2N\log_4 N$	$(3N/4)(\log_4(N) - 1)$	0	-
Direct-mapped radix-2^2	$2N\log_4 N$	$(3N/4)(\log_4(N) - 1)$	0	-
Pipeline radix-2	$2\log_2 N$	$\log_2(N) - 1$	N - 1	$N - 1$
Pipeline radix-4	$8\log_4 N$	$\log_4(N) - 1$	N - 1	$N - 1$
Pipeline radix-2^2	$4\log_4 N$	$\log_4(N) - 1$	N - 1	$N - 1$
Time-multiplexed radix-2	2	1	N	$N\log_2 N$
Time-multiplexed radix-4	8	1	N	$N\log_4 N$
Time-multiplexed radix-2^2	4	1	N	$N\log_4 N$

8.3.1 Radix-2^2 pipeline FFT

A basic radix-2^2 FFT building block consists of two radix-2 butterfly units separated by a trivial multiplication and a complex multiplier, as shown in Figure 8-4 (b). This can be directly mapped onto the CGRA by using two tile templates, as illustrated in Figure 8-4 (c). Each tile template contains two processing cells and two memory cells. 32-bit DSP processors are used for the butterfly operations, while CORDIC processor emulates complex multiplication using vector rotation [1].

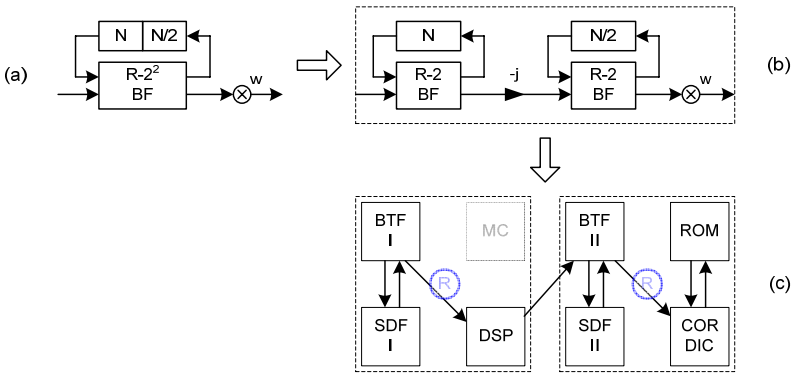


Figure 8-4. (a) A basic radix-2^2 FFT building block consists of a radix-2^2 butterfly unit (R-2^2 BF) and a complex multiplier. (b) A radix-2^2 butterfly unit is constructed from two radix-2 butterflies (R-2 BF), separated by a trivial multiplication. (c) A direct mapping on the CGRA. A basic radix-2^2 FFT building block requires two tile templates.

The BTF I and BTF II blocks in Figure 8-4 (c) stand for butterfly stage 1 and butterfly stage 2, respectively. They are functional blocks corresponding to the first and second radix-2 butterfly units drawn in Figure 8-4 (b). SDF I and SDF II blocks represent SDF buffers needed by each butterfly unit, which are implemented as FIFO operations in the memory cells. The DSP cell in the first tile template is responsible for the trivial multiplications in between two butterfly stages. The CORDIC processor takes care of the complex multiplications, and the ROM cell attached is used for feeding FFT twiddle factors required in each multiplication.

Simple mapping

A basic radix-2^2 FFT building block can be replicated in both horizontal and vertical directions to construct a larger size cell array, used for larger size FFT computations. As an example, Figure 8-6 illustrates an algorithm mapping for a 2,048-point radix-2^2 pipeline FFT (Figure 8-5) on an 8-by-8 reconfigurable cell array.

It is worth to mention that there is a list of advantages provided by this kind of hardware structure.

- Each stage in a radix-2^2 pipeline FFT occupies two tile templates, so in principle it is easy to replicate this up to achieve any transform length.
- It has a regular tile structure, where each processor has a memory cell attached and every other tile template contains a CORDIC processor. This is a versatile template structure that should be suitable for a broad range of algorithm implementations.
- All internal data transfers are realized by using local connections. The mapped pipeline FFT implementation therefore has a low demand on global data communications. Due to the high throughput data transmissions provided by the local network, this structure has a good property on data throughput and system latency.

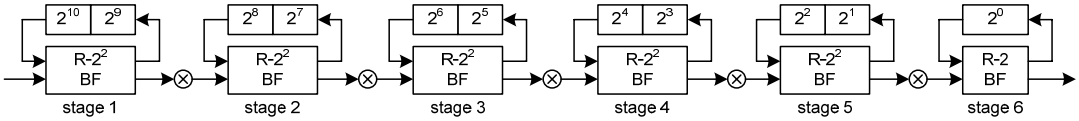


Figure 8-5. A 2,048-point pipeline FFT constructed from five radix-2^2 butterflies and one radix-2 butterfly.

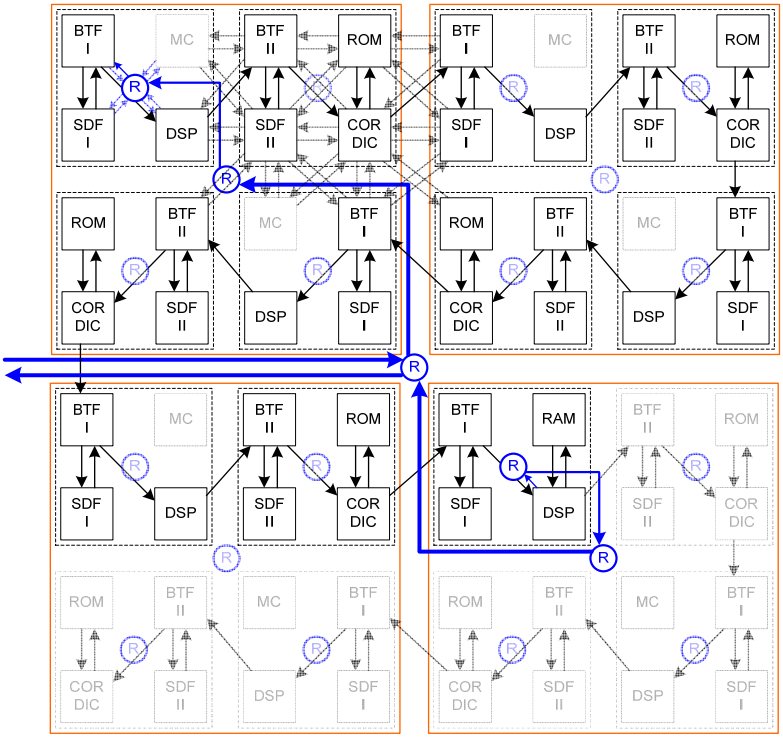


Figure 8-6. Structure of an algorithm mapping for a 2,048-point radix-2^2 pipeline FFT on an 8-by-8 reconfigurable cell array. Interconnections and building blocks drawn in lightened shade are unused system resources in this application. Complete local interconnects are shown in the second tile template of the first line, and a complete set of global interconnections is depicted in the first tile template.

- It is possible to do an output data shuffling by using the remaining DSP processor and memory cells, as shown at the last stage.

However, there are also limitations involved in this hardware mapping.

- This structure has high demand on data storage capacity for each single memory cell. For instance, the SDF buffer at the first stage in the preceding example requires a storage size of *1,024×input wordlength*. For processing the 16-bit complex data inputs, this is a 32K bits memory requirement. Hence, the actual realizable FFT implementation using this mapping scheme is limited by the provided memory capacity.
- If large data memories are possible to be integrated in each memory cell where the hardware size is a main concern here, this structure reveals low utilization

in the memory cells. In the example above, a 32 Kb memory space is required at first computation stage. However, there is only a 32-bit data storage required at the last stage, which is equivalent to a 0.1% memory usage.

- The reconfigurable cell array has an asymmetrical structure. Every other row of the tile templates have to be horizontally mirrored for proper local interconnections.
- Internal computation accuracy is restricted at a high data throughput, because current 32-bit DSP processor only supports 16-bit complex operations. For higher computation accuracy demands, the real and imaginary parts of complex inputs have to be treated separately in a processing cell, which requires separated I/O data transfers, results in a lower data throughput.

To summarize, the use of the simple mapping structure is mainly limited by the memory capacity available. This approach could be used for mapping small size FFT applications. For instance, with the reasonable memory capacity 8 Kb provided (as suggested in [1] pp. 136), the radix-2^2 pipeline FFT computation size can be supported up to 256-point.

Simple mapping with split complex number data path

To improve internal computation accuracy while maintaining a high data throughput, simple mapping structure can be combined with the use of split data path for complex number processing. This is shown in Figure 8-7.

The real and imaginary parts of the complex inputs are streamed into two different tile templates, where butterfly operations can be handled concurrently. Butterfly processed data outputs are gathered by a DSP or a CORDIC processor to perform the trivial or complex multiplications, respectively. Results are thereafter split again for further processing.

Another advantage in this structure is the relative low demand on memory storages. Because of the split data path, demand on the required SDF buffer for each processing is reduced by half. In addition, memory cells in the first butterfly stage can be concatenated to behave like a larger size SDF buffer.

Drawbacks with this structure are:

- Mapping has a high demand on system resources. Because each radix-2^2 pipeline FFT stage occupies 4 tile templates, the realizable transform length is limited by the number of resource cells available.
- This approach involves lots of hierarchical global communications for intermediate data processing. Therefore it has a high demand on the parallel switching ability in router cells. Moreover, routing data through global network results in relatively longer system latency. But due to the pipeline structure, data throughput will remain the same as previous mapping once the system runs in steady state.

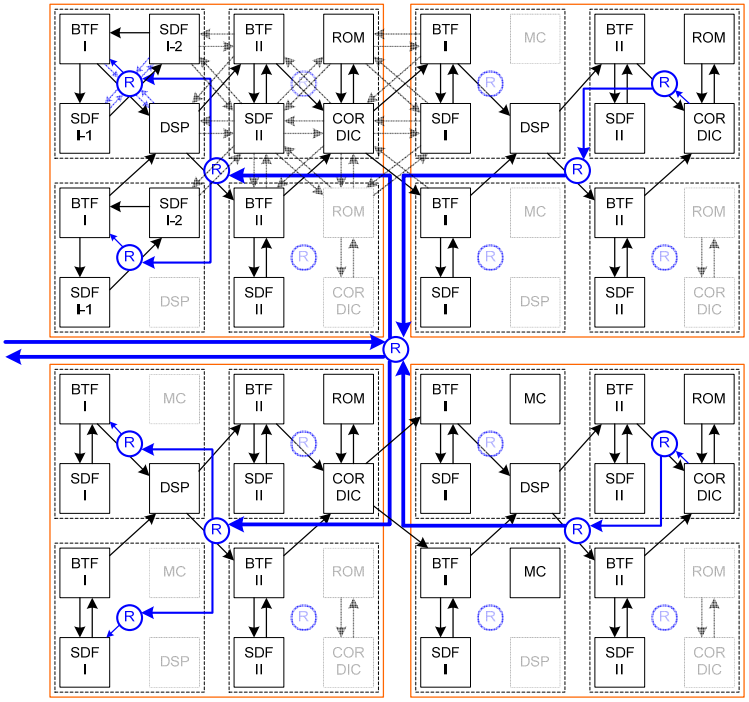


Figure 8-7. Structure of an algorithm mapping with split complex number data path for a 2,048-point radix-2^2 pipeline FFT on an 8-by-8 reconfigurable cell array.

Overall, the structure of a simple mapping with split complex number data path is mainly system resource limited, and to some extent also has impacts related to the provided memory capacity. This approach is suitable for mapping small size FFT applications with desired high computation accuracy. For instance, an 8-by-8 reconfigurable cell array supports up to 256-point radix-2^2 pipeline FFT with up to 32-bit internal calculation precision.

Simple mapping with concatenated memory cells

To reduce the high storage capacity requirement in each memory cell as mentioned in the previous two mapping approaches, memory cells can be concatenated together to provide larger data storage, which makes it feasible to be practically implemented. As shown in a previous study ([1] pp. 136), a good memory capacity selection for a memory cell is 8 Kb. So in a 2,048-point radix-2^2 pipeline FFT, SDF buffer in the first computation stage needs to have four memory cells concatenated together, as illustrated in Figure 8-8.

Comparing with the other two hardware mappings, this structure has much lower memory capacity requirements, and therefore higher memory cell utilization. Although global communications are also needed here for intermediate data processing, this has been well controlled to a regional level where no hierarchical routing is required. This approach could be used for mapping larger size FFT applications. For example, using memory cells with the suggested storage capacity, an 8-by-8 reconfigurable cell array supports up to 2,048-point radix-2^2 pipeline FFT computations.

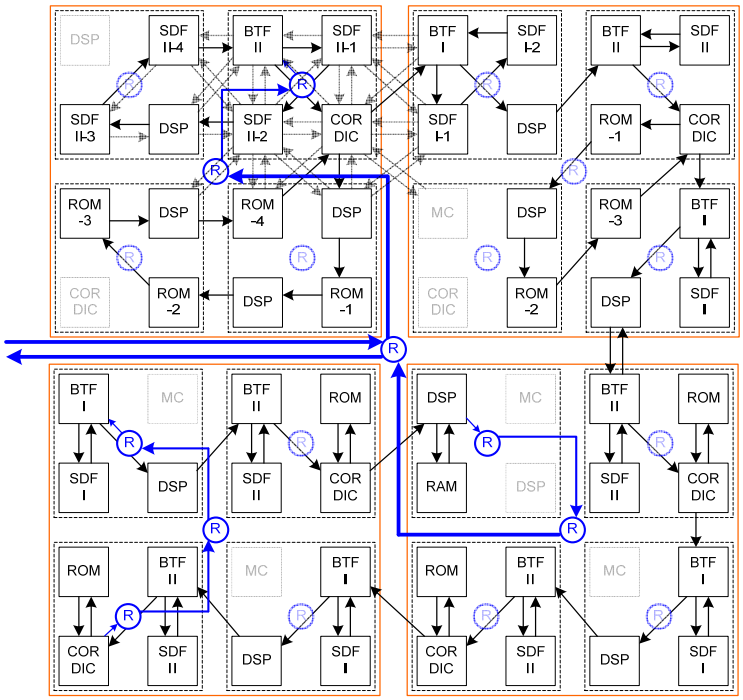


Figure 8-8. Structure of an algorithm mapping with concatenated memory cells for a 2,048-point radix-2^2 pipeline FFT on an 8-by-8 reconfigurable cell array. In contrast to the previous two hardware structures, required single radix-2 stage has been moved to the first computation stage in this example.

8.3.2 Time-multiplexed radix-2^2 FFT

A time-multiplexed radix-2^2 FFT structure is constructed by using one basic radix-2^2 FFT building block, where all the required computations are executed iteratively on the same hardware. Because of the relatively smaller system resource requirement, this structure has been implemented and realized on an FPGA platform. The detailed architecture diagram is shown in Figure 8-9 (a).

Resource cells at location (0, 0) and (2, 1) are 32-bit DSP processor cells, corresponding to the butterfly stage 1 and 2, respectively. Memory cells at (0, 1) and (3, 1) are SDF buffers required by each butterfly unit. The DSP processor at (1, 1) is responsible for the trivial multiplications, and the CORDIC processor at (2, 0) is used for the complex multiplications. In order to gain system flexibility, the FFT twiddle factor ROM has been substituted by a DSP processor placed at location (3, 0), and the coefficients are generated on the fly. The memory cell at (1, 0) runs in FIFO mode to buffer intermediate data between adjacent computation stages. Because data memories are provided as hard macros inside the FPGA, all memory cells have been equipped with a 32Kb storage space. As a result, this time-multiplexed radix-2^2 FFT structure supports a flexible transform length, from 32 to 1,024-point. Since the RAM mode operation in the memory cell is currently not available, data outputs in the hardware implementation are kept in a bit-reversed order, which can be sent back to MATLAB for post-processing. Hardware configurations for all the resource cells in

the radix-2^2 FFT application are summarized in Table 7-2. Diagram of the functional behaviors and the internal data flow graph are illustrated in Figure 8-9 (b).

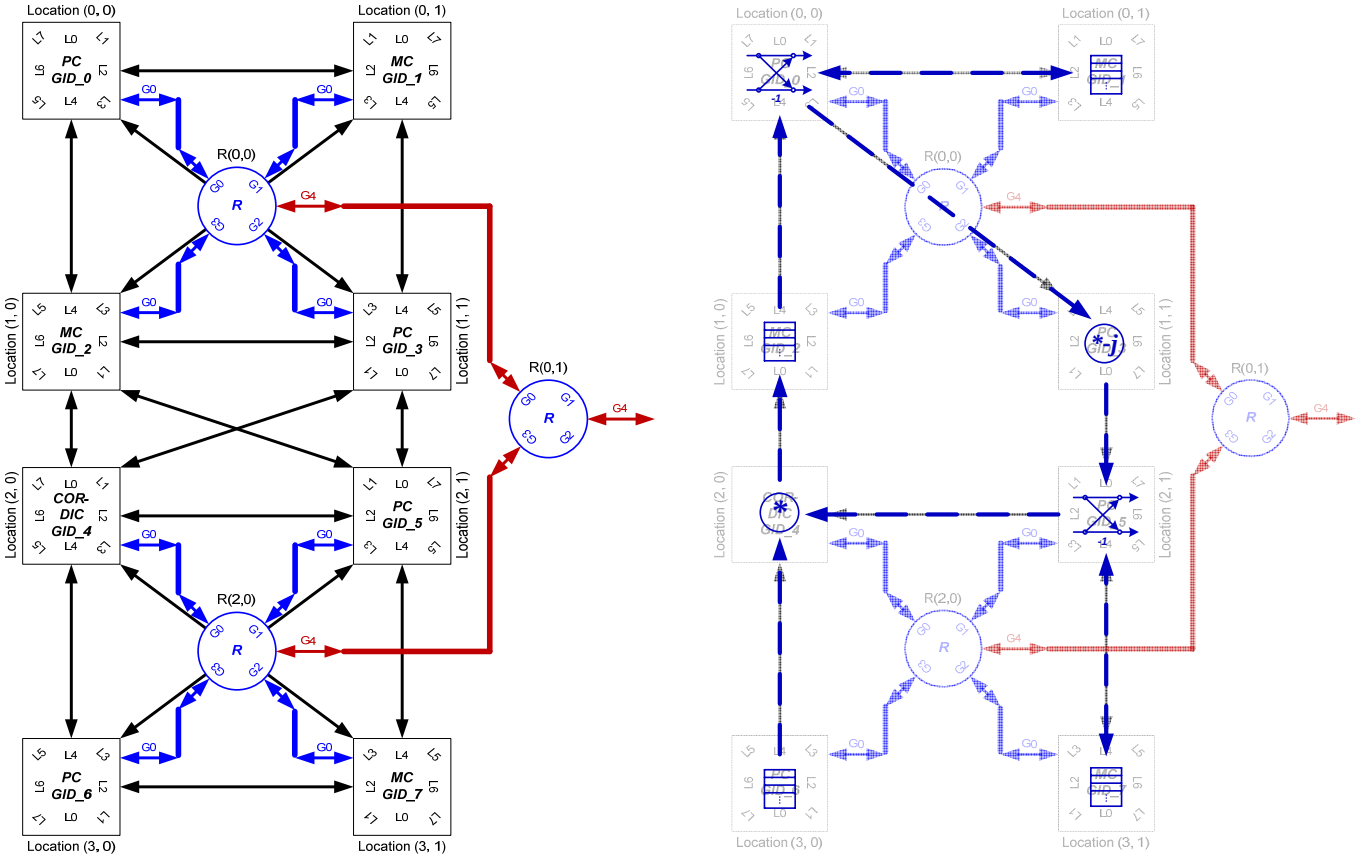


Figure 8-9. (a) Time-multiplexed radix-2^2 FFT structure in the CGRA. (b) Functional behavior of each recourse cell and internal data flow graph in the time-multiplexed radix-2^2 FFT structure.

Table 8-2. Hardware configurations for resource cells in a 4-by-2 cell array.

Location	Global ID	Cell type	Hardware configurations
(0, 0)	0	Processor	32-bit DSP processor; Program memory size = 64 × 32-bit; GPR = 11; LIO = 8; GIO = 1; Barrel shifter disabled.
(1, 1)	3		
(2, 1)	5		
(3, 0)	6		
(0, 1)	1	Memory	Memory capacity = 1024 × 32-bit; DSC table length = 4.
(1, 0)	2		
(3, 1)	7		
(2, 0)	4	CORDIC	16-bit pipelined core

The global routing network in this structure has two hierarchical levels, where the level number is indicated by the second digit in each router cell location. Global network in a lower layer is responsible for distributing data packages received from a

higher routing hierarchy. In this structure, layer 1 is the top level network hierarchy and is used for system level data transfers, such as configuration package downloading, resource cell status tracing, input/output data streaming, etc.

8.4 System performance evaluation

8.4.1 Radix-2^2 pipeline FFT

As discussed previously, mapping 2,048-point radix-2^2 pipeline FFT structure with concatenated memory cells requires an 8-by-8 cell array, as illustrated in Figure 8-8. Since the target FPGA chip can hold a maximum of 8 resource cells, the pipeline structure is far too large to be practically implemented on a FPGA platform. Besides, it is also a time-consuming task to simulate such a complex design as a computer based cycle-accurate HDL model, this design has only been analyzed theoretically in this project.

Because data samples in a pipeline structure is non-recursively streamed through the system, program routines in all processor cells are kept as simple as possible to maximize the processing throughput. By analyzing program instructions in each processor cell, execution time for processing each data sample can be extracted, as listed in Table 8-3.

Table 8-3. Instruction size and execution time evaluation for the processor cells.

Processor cell	Instruction amount	Instruction code size [bytes]	Execution time per data sample [clock cycles]		
			First sample	Typical	Last sample
BTF I	5	20	2	2	4
Trivial mul.	5	20	2	2	4
BTF II	5	20	2	2	4
Coeff. gen.	15	60	1	3	4

As an example, the program section for a function block “BTF I” in a processor cell looks like:

Table 8-4. Assembly program in the processor cell “BTF I”, designed for the radix-2^2 pipeline FFT.

```
.restart                                                  // Loop back label
ilc $C_FFT_SIZE/2                                         // 01: Set inner loop counter
dmov {l} $P_O_DATA, $P_O_SDF, $P_I_SDF, $P_I_DATA         // 02: Filling SDF buffer
ilc $C_FFT_SIZE/2                                         // 03: Set inner loop counter
btf {cl} $P_O_DATA, $P_O_SDF, $P_I_SDF, $P_I_DATA         // 04: Butterfly operation
bri .restart                                              // 05: Loop back
```

The typical clock usage for processing data samples in this cell requires no more than an instruction execution time (for operation “dmov” or “btf”), and an inner-loop control time. This is also valid for computing the first data sample. A worst case happens in the final loop back instruction. Since a branching operation requires one execution clock and one register flush clock, two additional clock cycles are needed there. Hence, there is a relatively large control overhead in this implementation when computing small size FFTs.

Overall, execution time for the radix-2^2 pipeline FFT in this design has a worst case clock usage of 4, and the average processing throughput is 3 clock cycles per data sample.

8.4.2 Time-multiplexed radix-2^2 FFT

The time-multiplexed radix-2^2 FFT structure contains three 32-bit processor cells: one 16-bit pipelined CORDIC cell and three 32 Kb memory cells, as shown in Figure 8-9 (a). Considering a realistic system configuration that is suitable for a wide range of algorithm implementations, the descriptor table length in each memory cell is configured to 4 during system design-time. In this FFT experiment, only one memory descriptor is actually needed, the other three locations are therefore left empty which result in waste of execution cycles. Although this causes system performance degradations, it is a trade-off between the system flexibility and the processing efficiency.

Data throughput on system I/O ports is not evaluated in this project, where instead input data samples are assumed to be ready for streaming through the hierarchical global routing network. Because intermediate FFT results have been kept locally, local I/O registers are used for data flow control between resource cells. System performances are evaluated based on a cycle-accurate HDL simulation model.

Program routines for all processing cells in time-multiplexed radix-2^2 FFT experiment are designed to emphasize the functional flexibilities rather than showing the processing throughput. This functional flexibility is reflected from the easy way of reconfiguring the FFT size during run-time. Changing the transform length requires at most 4 instructions to be downloaded in each processor cell. Referring to hardware configuration, the current FFT size supported by the platform is between 32 and 1024-point. The developed program routines prevent internal arithmetic overflows by conservatively scaling the results by 2 before each radix-2^2 stage. A small program segment in the processor cell (0, 0) is shown in Table 8-5, and the total instruction size for each processor cell is summarized in Table 8-6.

Table 7-3 shows a few performance metrics extracted from the simulations. Compared to a DSP solution as shown in Table 8-8, the CGRA exhibits great reconfigurability on the code size, which further implies the required reconfiguration time. These results also show that better transform throughput can be achieved by reducing the descriptor table length in the memory cells. However, this might limit the use of CGRA in other algorithm mappings due to the loss of configurability in the memory cells. To summarize, the choice of hardware configuration is closely linked

to the field of application, a trade-off between the system flexibility and the processing efficiency has to be decided before the actual implementation work begins.

Table 8-5. Program instructions in processor cell at location (0, 0) for a time-multiplexed radix-2^2 FFT. Table columns are: binary code, assembly program, comment and reconfigurability.

B000000A	// 01: GID 10	; Load GIO TX DST ID	**For reconfiguration**
8480007F	// 02: ADDI $4, $0, 127	; N_FFT/2-1	**For reconfiguration**
84400004	// 03: ADDI $2, $0, 4	; Load stage counter	**For reconfiguration**
85000000	// 04: ADDI $8, $0, 0	; Last stage flag	**For reconfiguration**
84600001	// 05: ADDI $3, $0, 1	; Load iteration counter	
40002000	// 06: ILC $4	; Stage 1, special case	
06A0D801	// 07: ADD $L2, $G0, $0 {l}		
		…	

Table 8-6. Instruction size summary for the processor cells.

Processor cell	PC (0, 0)	PC (1, 1)	PC (2, 1)	PC(3, 0)
Usage	BTF-I	Trivial mul.	BTF-II	Coeff. gen.
Instruction amount	53	34	44	20
Instruction code size [bytes]	212	136	176	80
Reconfiguration inst. amount	4	3	4	3
Reconfiguration code size [bytes]	16	12	16	12

Table 8-7. System performance explorations for the CGRA based on the different FFT computations.

FFT size [points]	32	256	1,024
Input wordlength [bits]	10		
Scaling scheme	Conservative		
SQNR [dB]	39.337	39.274	39.928
Memory [bits]	96K (32K × 3)		
Memory utilization (Memory usage [bits])	2.083% (512+512+1K)	16.67% (4K+4K+8K)	66.67% (16K+16K+32K)
Maximum frequency [MHz]	27.398		
FPGA usage on Xilinx Virtex-II Pro-30-7ff896	Number of slices: 11,022 out of 13,696 – 80% usage; Number of BRAMs: 10 out of 136 – 7% usage		
Configuration time [clock cycles]	1,552	4,912	16,432
Reconfiguration time [clock cycles]	184		
Latency [clock cycles]	433	3,965	19,837
Execution time [clock cycles]	806	7,026	32,114
1D transform per second	33,993	3,899	853

Table 8-8. FFT benchmark comparison between the CGRA and a DSP processor, the TMS320VC55x from Texas Instruments.

Architecture	**FFT size [points]**	**Execution time [clock cycles]**		**Code size [bytes]**	**Reconfiguration code size [bytes]**
		4 mem. DSC	**2 mem. DSC**		
CGRA	32	806	***423***	604	56
	256	7,026	***4,290***		
	1,024	32,114	***20,212***		
Texas TMS320VC55x [22]	32	591		462	
	256	5,389			
	1,024	25,921			

8.5 Embedded system development in FPGA

8.5.1 System overview

A FPGA based embedded system has been designed using the Xilinx Platform Studio (XPS) software environment and verified on a Xilinx XUP Virtex-II Pro development board. A 32-bit MicroBlaze soft core is selected as a primary system processor, and a 4-by-2 reconfigurable cell array is embedded as a co-processor connected on a shared processor local bus (PLB), as shown in Figure 8-10. Bidirectional data transmissions in the cell array are handled by a global I/O port. To adapt the different data transmission protocols, the reconfigurable cell array uses a wrapper to interface to a PLB bus. The UART Lite module manages system level data transfers with the external host, and the interrupt controller is responsible for informing the primary system processor to receive data from the cell array. These two units are soft IP cores from the XPS tool and are both connected to the PLB bus. The primary system processor acts as a PLB bus master, and all the other connected hardware modules operate in slave mode. The block diagram for the embedded system is illustrated in Figure 8-10, and the total FPGA device utilization is summarized in Table 8-9.

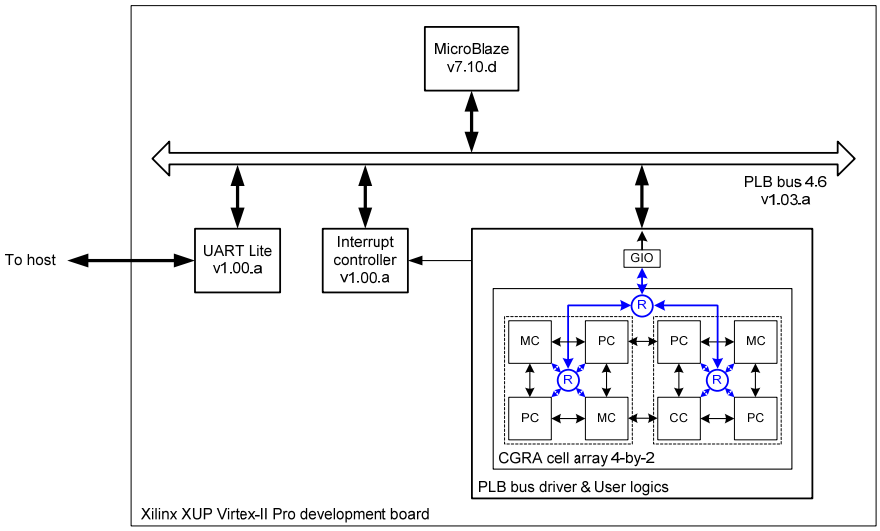


Figure 8-10. Block diagram of the embedded system in FPGA development

Table 8-9. FPGA device utilization summary for the embedded system development

Number of DCMs	1	out of	8	12%	System clock management.
Number of MULT18X18s	3	out of	136	2%	Used by the MicroBlaze.
Number of RAMB16s	42	out of	136	30%	Required by the software developments in MicroBlaze.
Number of SLICEs	12,919	out of	13,696	94%	Total slice usage, where the 4-by-2 cell array occupies 85% of the total slice usage.

From section 8.4.2, the maximum clock frequency for a 4-by-2 standalone cell array is about 27 MHz, therefore 25 MHz is used as the PLB bus clock in a FPGA embedded system. Because of the selected MicroBlaze architecture in XPS, the primary processor is forced to operate at the same clock speed as the PLB bus clock.

8.5.2 Communication with the system processor

Bidirectional communication between the cell array and the primary processor (MicroBlaze in this case) is realized through the use of four interface registers, two for each direction. An interface register is a 32-bit wide transparent software addressable register communicated through the PLB bus. Interface register 0 and 2 are used for bidirectional communication handshakes and proper controlling in each transmission direction. Register 1 and 3 are dedicated to unidirectional data transfers.

Although the cell array and the primary processor both operate at the same clock speed, communication handshaking still needs to be performed. This is because it is desirable to be implement system management software in the primary processor on a high-level programming language such as C, in order to gain the design flexibility. Hence, signal assignments from the primary processor might not be cycle controllable, which would potentially cause action duplications. For instance, by issuing an enable signal from the primary processor to the global input port in a cell array, a data transmission between two blocks is initiated. An improper release of the enable signal will trigger multiple data transfers and hence result in data package duplications.

To maintain integrity of each data package, data transmissions in the embedded system are treated as blocking read/write operations, and a two-phase communication protocol is engaged to accomplish each transaction. A data transfer can be initiated by issuing an action command from the primary processor and completed by detecting an acknowledgement action from the cell array block. The acknowledgement action is handled by the cell array wrapper design, i.e. the PLB bus driver as shown in Figure 8-10. When a communication action from the primary processor is discovered, additional control logics is activated to monitor feedback signals from the cell array. If no stalling signal is detected in the following clock, control signals to the cell array is released and the corresponding control commands in the interface register is erased. The primary processor should keep track of the corresponding interface register after sending a communication command, and this can be released by detecting an erased control bit position from that register. Communication control commands and bit map arrangement for the interface registers are summarized in Table 8-10 and Table 8-11, respectively.

A global data RX interrupt is used in data transmissions from the cell array to the primary processor, which releases the host from data polling. Currently, this is the only interrupt enabled in the system implementation. However, if more interrupt events exist in an embedded system, the global data RX event should still be kept with the highest priority, since any data miss is prohibited.

Table 8-10. Interface Communication control commands

Command	Description
tx_en	Data sending enable signal in global I/O TX port.
dst_id_en	GIO destination write enable signal in global I/O TX port.
cap_ttype	Global data transfer package type.
rx_en	Data receiving enable signal in global I/O RX port.

Table 8-11. Interface register bit map arrangement

Interface register	Bit map				
	31 ~ 18	**17 ~ 16**	**15 ~ 2**	**1**	**0**
0	Reserved	CAP_TTYPE	Reserved	DST_ID_EN	TX_EN
	Unidirectional			*Bidirectional*	
1	TX data package to the cell array				
	Unidirectional				
2	Reserved				RX_EN
	Unidirectional				*Bidirectional*
3	RX data package from the cell array				
	Unidirectional				

8.5.3 Software development

Software development for the primary processor is performed in C and compiled using the GCC compiler. Software flow graphs are shown in Figure 8-11. The main program starts by initializing the interrupt controller and a batch of message printouts. Thereafter, the system runs in a loop that monitors data inputs from the UART interface and handles different user actions accordingly. Unrecognized user inputs are discarded and a valid user command activates the corresponding operations in the cell array. A complete user command set is listed in Table-Appendix 1, and detailed descriptions are discussed in section 8.5.5. A data receiving notification from the global I/O port interrupts the sequential process in the main program. Global data receiving in the primary processor is handled by an interrupt service routine (ISR) as depicted in Figure 8-11 (b).

In order to run a real application on the cell array, certain configuration procedures have to be followed. As an example, a configuration flow graph for the FFT implementation is shown in Figure 8-12. Notice that, processor cells in the cell array are first configured. This is because reset action in the processor cell will flush out all stored values from internal registers, including the communication port registers. If processor cells are initialized after memory cell configurations, pre-loaded data transfers in communication port registers will be lost.

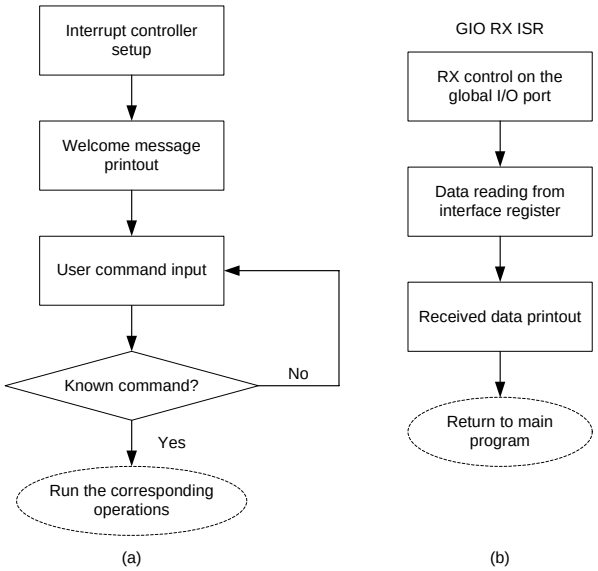


Figure 8-11. Software development flow graph in the primary processor, the MicroBlaze in current implementation. (a) Flow graph of the main program implementation. (b) Global I/O RX port data receiving interrupt service routine.

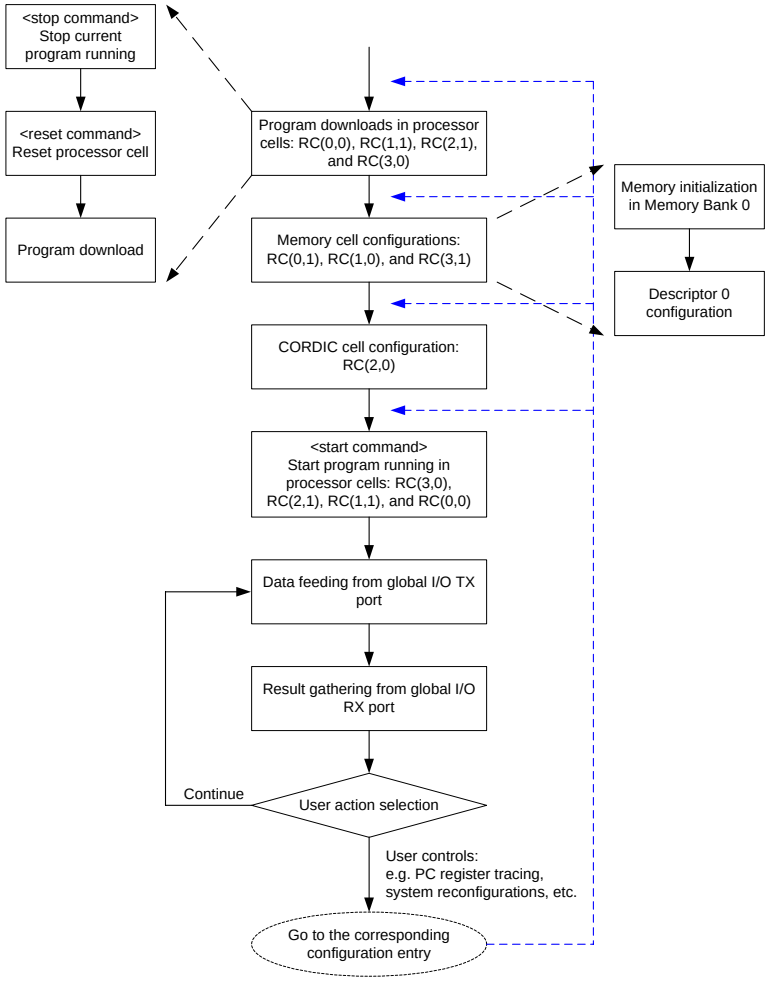


Figure 8-12. Implementation flow graph for a time-multiplexed radix-2^2 FFT application.

8.5.4 UART bit rate setup

Considering the maximum system running speed of 25MHz on the target FPGA platform, a list of UART bit rates are evaluated to find the maximum achievable communication rate.

As an example, with a bit rate of 9600 bps (bits per second), the required UART sampling clock by the Xilinx XPS UART Lite v1.00a core [20] is 16 times higher, 153.6 kHz *(16×9600 bps)*. With the system running clock 25MHz, the dividable integer clock ratio for driving the UART sampling clock is 162 *(floor(25 MHz/153.6 kHz))*. Therefore the actual realizable UART sampling clock in the Xilinx UART Lite core can be found by dividing the system clock by 162 which results in 154.321 kHz. As a consequence, the bit rate error is 0.4694% *((154.321 KHz-153.6 KHz)/153.6 KHz×100%)*. According to the specification [20], the bit rate error is considered to be acceptable if it is within 5% of the requested rate. Hence, a bit rate of 9600 bps with rate error of 0.4694% is an acceptable configuration for the system.

Using the Microsoft HyperTerminal program as a reference, the supported high speed UART bit rates (with the use of a USB to RS232 converter) are evaluated and listed in Table 8-12. According to the bit rate error criterion, all acceptable bit rate setups are marked in Bold-Italic font, and the maximum usable bit rate in the current system implementation is 115,200 bps.

Table 8-12. UART bit rate error evaluation table

Bit rate [bps]	UART SCLK required [Hz]	System CLK [MHz]	Integer CLK ratio	UART SCLK actual [Hz]	Bit rate error [%]
4,800	***76,800***	25	***325***	***76,923***	***0.1603***
9,600	***153,600***		***162***	***154,321***	***0.4694***
19,200	***307,200***		***81***	***308,642***	***0.4694***
38,400	***614,400***		***40***	***625,000***	***1.7253***
57,600	***921,600***		***27***	***925,926***	***0.4694***
115,200	***1,843,200***		***13***	***1,923,077***	***4.3336***
230,400	3,686,400		6	4,166,667	13.0281
460,800	7,372,800		3	8,333,333	13.0281
921,600	14,745,600		1	25,000,000	69.5421

8.5.5 User interface in serial line

A transparent user interface in UART line has been designed to provide the user with an easy way of controlling the embedded system. Processor cell instructions, memory or CORDIC cell configurations, control commands and global data inputs can be transmitted through the top level global port communication. In addition, a few pre-stored FFT configuration scripts are provided as fast system demonstrations. Different user actions can be executed by calling different user commands, as listed in

Table-Appendix 1. Bulk data transfers can be accomplished by sending a user defined script file.

To be able to distinguish between different user actions, serial line inputs have been divided into three categories: command input, number input and string input. Command input is wrapped around by a leading '@' and a trailing '#' sign; string input starts with a '$' sign and ends with a '#' sign; number input only accepts pure digit inputs 0 ~ 9 and ends with a user input other than digits. For example, the following command inputs download an instruction into a processor cell that is labeled with GIO ID 0.

>> @g#	// (command input) command 'g', destination cell selection.
>> 0	// (number input) resource cell GIO port ID.
>> @i#	// (command input) command 'i', instruction downloading.
>> 2	// (number input) the number of instructions to be sent.
>> $00010002#	// (string input) instruction loading header.
>> $A8000001#	// (string input) instruction "A8000001".

8.5.6 User interface in MATLAB

Based on a transparent serial line interface, a higher level user control platform is designed in MATLAB. This interface gives the user a more advanced and flexible approach to control the data streams running in and out of the embedded system. For example, input data sequences can be generated in MATLAB during run-time and result data can be collected and plotted graphically.

The MATLAB interface works as a front-end user platform, where the serial line interface runs in the background. By issuing different user commands, as listed in Table-Appendix 2, different function calls will be executed. It is worth to mention that using command 'cmd' will provide the user a transparent function control in the serial line interface for the follow-up command input.

8.6 Conclusion

A complete radix-2^2 FFT implementation based on the CGRA has been presented. Several FFT mapping alternatives have been discussed and compared. System performance evaluations have been carried out based on a pipeline FFT structure as well as on a time-multiplexed mapping approach. A 4-by-2 reconfigurable cell array has been integrated as a co-processor into an embedded system and eventually verified on a Xilinx FPGA board. The outcome from this case study is a fully functional 4-by-2 reconfigurable cell array with a manually mapped flexible radix-2^2 FFT implementation, where the transform length is run-time reconfigurable between 32 and 1,024-point.

From the initial algorithm selection to the final system implementation, all design procedures introduced in the generalized system design flow as depicted in Chapter 7

have been tightly followed in this experiment.

In comparison to an ordinary DSP solution, the time-multiplexed radix-2^2 FFT implementation on the CGRA exhibits great reconfigurability on the code size and system reconfiguration time. Results also show that up to 20% of the total clock usage can be saved when using the CGRA with configurations in pursuit of processing efficiency, namely with the proper memory descriptor length setup.

8.7 Future work

8.7.1 Serial line input speed up

Due to the current UART RX handling in the embedded system, certain character input delays have to be inserted when streaming data into the platform, for example 1-millisecond character delay is needed when sending data from Microsoft HyperTerminal program. This is used to ensure the data integrity of each character transmission, since other control actions or UART TX events could interrupt the serial data receiving and result in data misses. As a main drawback of this scheme, the user suffers from a long waiting time for data sending completions. In order to improve that, the UART RX interrupt with a suitable length of receiving FIFO should be used in the embedded system. Moreover, current primary system processor – the MicroBlaze could be replaced by an embedded Power PC core inside the FPGA, where a much higher processor clock can be used for system management, such as control handlings and information printings, etc.

Alternatively, data streaming throughput can be improved by using a high speed communication interface. One possible solution is to use a TCP/IP socket, which provides the user a data transmission rate in the scale of mega bits per second.

8.7.2 Dead lock handling

According to the communication protocol defined in Chapter 2.3.1, relevant data transactions in all resource cells should be suspended if any node in the chain gets data stalled. This might cause a data dead lock if something goes wrong in the system configuration. For instance, dead lock will occur if running a 64-point FFT computation with the use of memory cells configured for 32-point only. In that case, entire system will be suspended due to the lack of data inputs in the processing cells. Therefore, a system level user action is needed to interrupt the stalled data transmissions and to restart the entire system subsequently.

9 Conclusion and Outlook

Coarse-grained reconfigurable architecture (CGRA) coexists with fine-grained reconfigurable architecture (FGRA), aimed for higher computing performance and reduced system development time. The proposed CGRA is constructed from an array of run-time reconfigurable processing and memory cells that are interconnected over a hybrid communication network. Based on a series of pre-studies, each of the individual cell modules inside the CGRA has been designed, implemented, verified and evaluated. The outcome of this work is a system-level exploration on the usage of the CGRA targeted for DSP applications, where a time-multiplexed FIR filter and a 32~1,024-point flexible time-multiplexed radix2^2 FFT algorithm have been manually mapped onto a 4-by-2 reconfigurable cell array, and finally verified on an FPGA platform. A list of system performance metrics has been measured and mainly presented based on the FFT implementation, which showed that the reconfiguration code size in the CGRA outperforms the ordinary DSP processor by a factor of 8, and up to 20% of the total execution clock usages can be saved.

Function testing, debugging or running diagnostics on large-scale system architecture is always problematic, ability to observe the inner working status of each resource cell is therefore essential. The processing cells implemented in CGRA support run-time status tracing, where the configuration parameters and operation states can be reported upon the user requests. Besides, a few hardware assisted approaches for system-level debugging have been proposed throughout chapters, such as the industry standard JTAG chain, system BIST and memory content dumping, etc.

Looking forward, applying the CGRA into a wide application domain is a trend. However, this requires a series of system-level exploration tools to model, simulate and evaluate the use of the CGRA in different application environments, in order to extract appropriate design parameters to achieve high performance to a feasible hardware cost. Although there are still lots of system-level investigations left under considerations, the future of the CGRA is certainly bright.

Bibliography

[1] Thomas Lenart, *Design of Reconfigurable Hardware Architectures for Real-time Applications*, Ph.D dissertation, Lund University, Department of Electrical and Information Technology, 2008.

[2] Henrik Svensson, *Reconfigurable Architectures for Embedded Systems*, Ph.D dissertation, Lund University, Department of Electrical and Information Technology, 2008.

[3] Chenxin Zhang and Chao Wang, *Implementation of Processor Cell in Reconfigurable Computing*, project report in course IC Project and Verification (ETI210), Lund University, Department of Electrical and Information Technology, 2008.

[4] Wikipedia, "Buffer overflow," 28 January 2009, http://en.wikipedia.org/wiki/Buffer_overflow.

[5] Wikipedia, "Buffer underrun," 2 January 2009, http://en.wikipedia.org/wiki/Buffer_underrun.

[6] Thomas. Lenart, Henirk. Svensson and V. Öwall, "A Hybrid Interconnect Network-on-Chip and a Transaction Level Modeling approach for Reconfigurable Computing," in *Proceedings of IEEE International Symposium on Electronic Design, Test and Applications*, Hong Kong, China, January 2008, pp. 398–404.

[7] Henrik Svensson, Thomas Lenart, and Viktor Öwall, *Modeling and Exploration of a Reconfigurable Array using SystemC TLM*, in Proceedings of Reconfigurable Architectures Workshop, Miami, Florida, USA, April, 2008.

[8] J. L. Hennessy and D. A. Patterson, *Computer architecture: A Quantitative Approach*, fourth edition, Morgan Kaufmann Publishers, 2003.

[9] Yu Hen Hu, "The Quantization Effects of the CORDIC Algorithm," in *IEEE transactions on signal processing*, vol. 40, no. 4, pp. 834-844, 1992.

[10] A. Meyer-Bäse, *et al.* "A parallel CORDIC architecture dedicated to compute the Gaussian potential function in neural networks," in *Engineering Applications of Artificial Intelligence*, vol. 16, pp. 595-605, 2003.

[11] Xilinx, *CORDIC v3.0 Product Specification*, May21, 2004.

[12] Fredrik Edman, *Digital Hardware Aspects of Multiantenna Algorithms*, Ph.D dissertation, Lund University, Department of Electroscience, 2006.

[13] B. Parhami, *Computer Arithmetic: Algorithm and Hardware Designs*, Oxford University Press, 2000.

[14] Thomas Lenart, *A Hardware Accelerator for Digital Holographic Imaging*, Licentiate Thesis, Lund University, Department of Electrical and Information Technology, 2005.

[15] Chih-Pang Hsu, *Design of Fast Fourier Transform Processor in DVB-T Inner Receiver*, Master thesis, Institute of Communication Engineering, National Central University, 1995.

[16] Faraday, *UMC 0.13µm MEMAKER*, 2007.

[17] Thomas Lenart, Henrik Svensson and Viktor Öwall, *Implementation of Application Specific Stream Processors*, project specification in the IC project and verification course, Lund University, Department of Electrical and Information Technology, 2008.

[18] Wikipedia, "Round-robin scheduling," 9 December 2008, http://en.wikipedia.org/wiki/Round-robin_scheduling.

[19] Matthias Kamuf, "DSP-Design, Seminars and Labs," DSP design (ETI180) course manual, Lund University, Department of Electrical and Information Technology, 2008.

[20] Xilinx, "XPS UART Lite (v1.00a) Product Specification", July 18, 2008.

[21] J. G. Prpalos and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, third edition, Prentice-Hall, 1995.

[22] Texas Instruments, "TMS320C55x DSP Library Programmer's Reference," http://www.cucat.org/general_accessibility/accessibility/Braille%20scanner/Misc%20code/datasheets/5402programmersref.pdf.

Appendix

Appendix A – Processor cell architecture

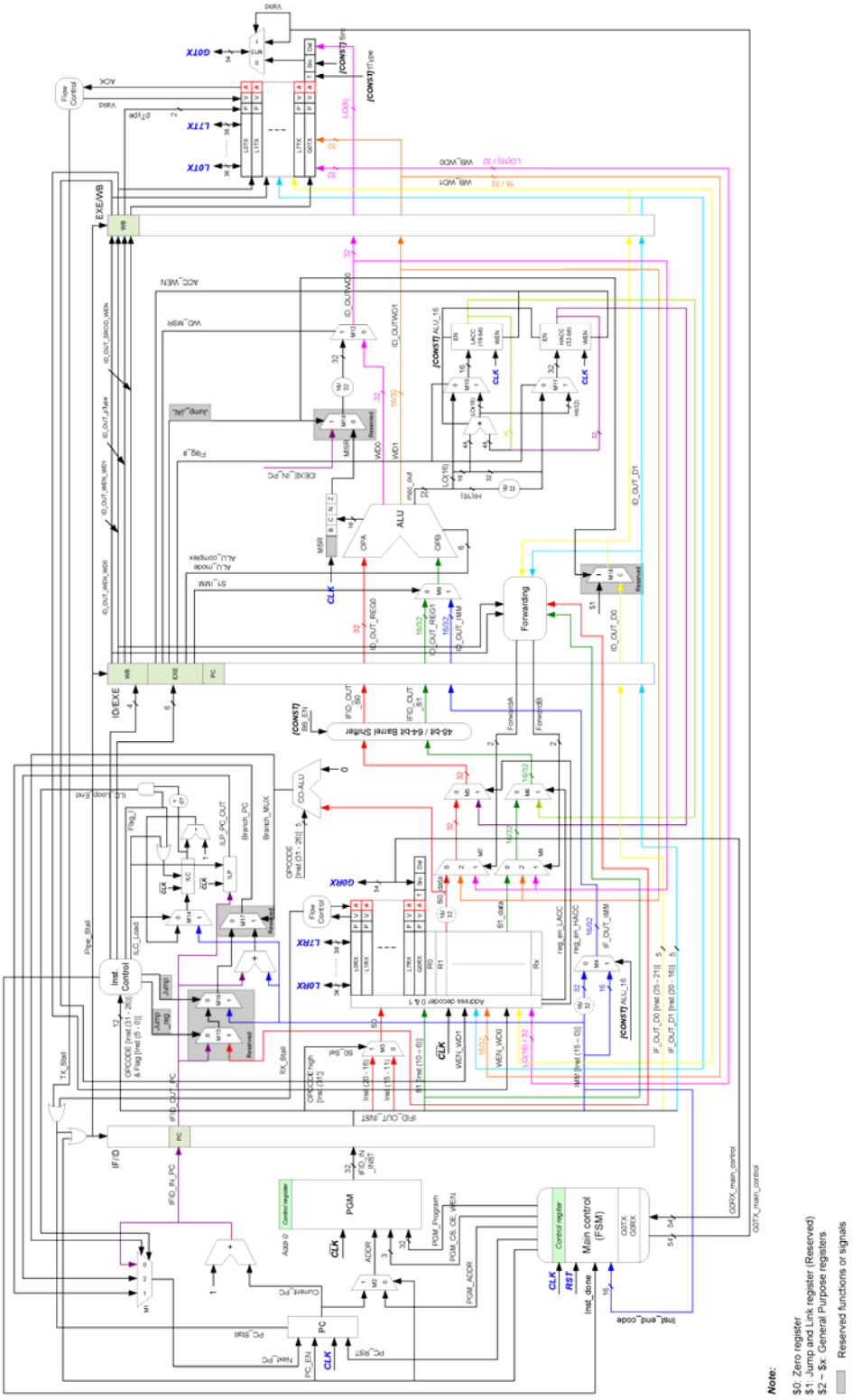


Figure-Appendix 1. Processor cell architecture

Appendix B – Processor cell register bank

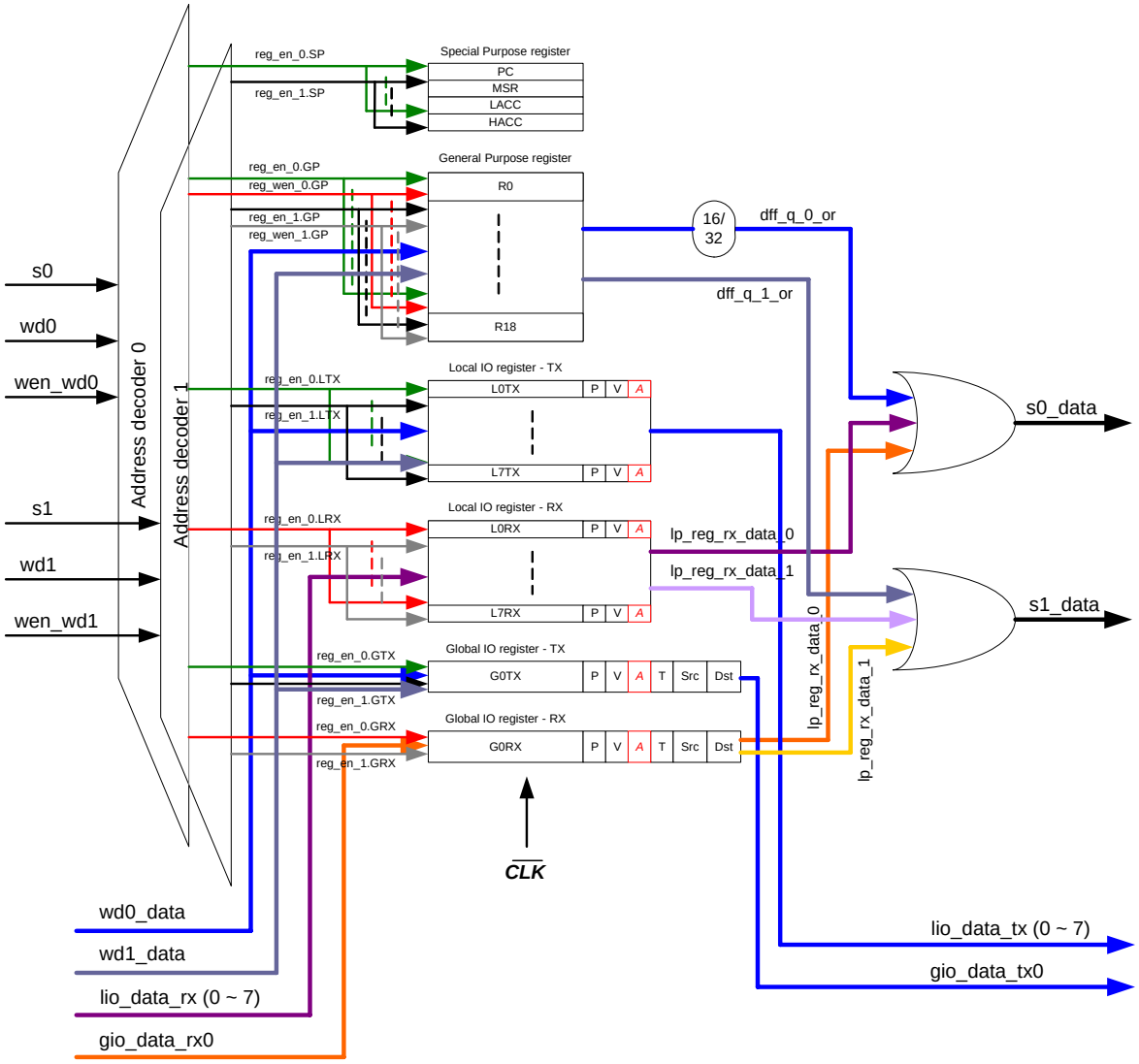


Figure-Appendix 2. Processor cell register bank

Appendix C – Processor cell instruction set

CGRA Processor Cell Instruction Set

Mnem-onics	Operands	Type	Description	Operation	Flags	Clock Cycles	Instruction Coding 31-26	25-21	20-16	15-11	10-6	5-0
Basic Instruction Set												
ADD	D0, S0, S1	A	Data add between registers	D0 <- S0 + S1	z,n,c	1	000001	D0		S0	S1	{srwcl}
SUB	D0, S0, S1	A	Data subtract between registers	D0 <- S0 - S1	z,n,b	1	000010	D0		S0	S1	{srwcl}
DMOV	D0, D1, S0, S1	A	Double data move between registers	D0 <- S0; D1 <- S1		1	000111	D0	D1	S0	S1	{srwl}
ADDI	D0, S0, Imm	B	Add register data and immediate	D0 <- S0 + SXT(Imm)	z,n,c	1	100001	D0	S0		Imm	
SUBI	D0, S0, Imm	B	Subtract register data and immeiate	D0 <- S0 - SXT(Imm)	z,n,b	1	100010	D0	S0		Imm	
BEQI	S0, Imm	B	Branch on equal to 0	PC <- PC + SXT(Imm) if S0 = 0		1~2*	100011		S0		Imm	
BNEI	S0, Imm	B	Branch on not equal to 0	PC <- PC + SXT(Imm) if S0 != 0		1~2*	100100		S0		Imm	
BLTI	S0, Imm	B	Branch on less than 0	PC <- PC + SXT(Imm) if S0 < 0		1~2*	100101		S0		Imm	
BLEI	S0, Imm	B	Branch on less than or equal to 0	PC <- PC + SXT(Imm) if S0 <= 0		1~2*	100110		S0		Imm	
BGTI	S0, Imm	B	Branch on greater than 0	PC <- PC + SXT(Imm) if S0 > 0		1~2*	100111		S0		Imm	
BGEI	S0, Imm	B	Branch on greater than or eugal to 0	PC <- PC + SXT(Imm) if S0 >= 0		1~2*	101000		S0		Imm	
SLL	D0, S0	A	Left shift 1-bit	D0 <- S0 << 1	z,n,c	1	001001	D0		S0		{srwcl}
SRL	D0, S0	A	Right shift 1-bit	D0 <- S0 >> 1	z,n,c	1	001010	D0		S0		{srwcl}
ROL	D0, S0	A	Left rotate 1-bit with carry	D0 <- (Carry + S0) rol 1	z,n,c	1	001011	D0		S0		{srwcl}
ROR	D0, S0	A	Right rotate 1-bit with carry	D0 <- (Carry + S0) ror 1	z,n,c	1	001100	D0		S0		{srwcl}
AND	D0, S0, S1	A	Logical AND two register data	D0 <- S0 & S1	z,n	1	001101	D0		S0	S1	{srwcl}
OR	D0, S0, S1	A	Logical OR two register data	D0 <- S0 | S1	z,n	1	001110	D0		S0	S1	{srwcl}
XOR	D0, S0, S1	A	Logical XOR two register data	D0 <- S0 ⊕ S1	z,n	1	001111	D0		S0	S1	{srwcl}
ANDI	D0, S0, Imm	B	Logical AND register data with immediate	D0 <- S0 & SXT(Imm)	z,n	1	101101	D0	S0		Imm	
ORI	D0, S0, Imm	B	Logical OR register data with immediate	D0 <- S0 | SXT(Imm)	z,n	1	101110	D0	S0		Imm	
XORI	D0, S0, Imm	B	Logical XOR register data with immediate	D0 <- S0 ⊕ SXT(Imm)	z,n	1	101111	D0	S0		Imm	
Special Instruction Set												
NOP	---	A	No operation	---		1	000000					
BRI	Imm	B	Unconditional branch (Relative Jump)	PC <- PC + SXT(Imm)		1~2*	101001		$0		Imm	
END	Imm	B	Stop execution and return ending code	END_CODE <- Imm		1	101010				Imm	
ILC	S0	A	Inner loop control with register	ILP <- PC + 1; ILC <- S0		1	010000			S0		
ILCI	Imm	B	Inner loop control with immediate	ILP <- PC + 1; ILC <- Imm		1	101011				Imm	
BTF	D0, D1, S0, S1	A	Butterfly operation	D0 <- S0 + S1; D1 <- S0 - S1	z,n,c,b	1	000011	D0	D1	S0	S1	{ci}

Page 1 of 2

Figure-Appendix 3. Processor cell instruction set

CGRA Processor Cell Instruction Set

SWAP	D0, S1	A	Complex number swap operation	HI(D0) <- LO(S1); LO(D0) <- (-) HI(S1)	z,n,b	1	001000	D0		$0	S1	{nl}
GID	Imm	B	Global IO TX port destination ID write	GIO.dst <- Imm		1	101100		$0	Imm		

16-ALU Specific Instruction Set

SMOV	D0, D1, S0	A	Split data move between one 32-bit register and two 16-bit registers	D0 <- HI(S0); D1 <- LO(S0)		1	000101	D0	D1	S0		{srwal}
JMOV	D0, S0, S1	A	Joint data move between two 16-bit registers and one 32-bit register	HI(D0) <- S0; LO(D0) <- S1		1	000110	D0		S0	S1	{srwal}
MUL	S0, S1	A	Multiply and accumulate	HACC <- HI(S0 * S1) + HACC; LACC <- LO(S0 * S1) + LACC;		1~2**	000100			S0	S1	{srwal}

Note:
* Branch-untaken operation requires 1 execution clock, and 2 clock cycles for a branch-taken action.
** Accumulation instruction requires 1 clock cycle for doing corresponding arithmetic operations, but requires 2 clock cycles when reading values out from accumulation registers right after the accumulation instruction "MUL", since accumulation results are updated on the rising edge of the following clock.

(1) $0: Zero register (Read Only), General Purpose register 0.
(2) Bit layout of the instruction flags:
  0: l - End of inner loop
  1: c - Complex data 2*W/2-bit
    a - Accumulate value to xACC
    n - "SWAP" specific, negate low nibble, e.g. (a, b) => (-b, a)
  2: r - Local IO port write is read request
  3: w - Local IO port write is write request
  4: Reserved.
  5: s - Barrel shift enable. When this bit is set, the lower 5 bits are used as right shifting count; else, they are used as primitive flags.
(3) *Flags* are the values stored in the ALU status register, which are updated upon each of the ALU arithmetic operations.
  z: Zero flag
  n: Negitive flag
  c: Carry out flag in addition
  b: Borrow in flag in subtraction

Figure-Appendix 4. Processor cell instruction set continued

Appendix D – Processor cell control flow

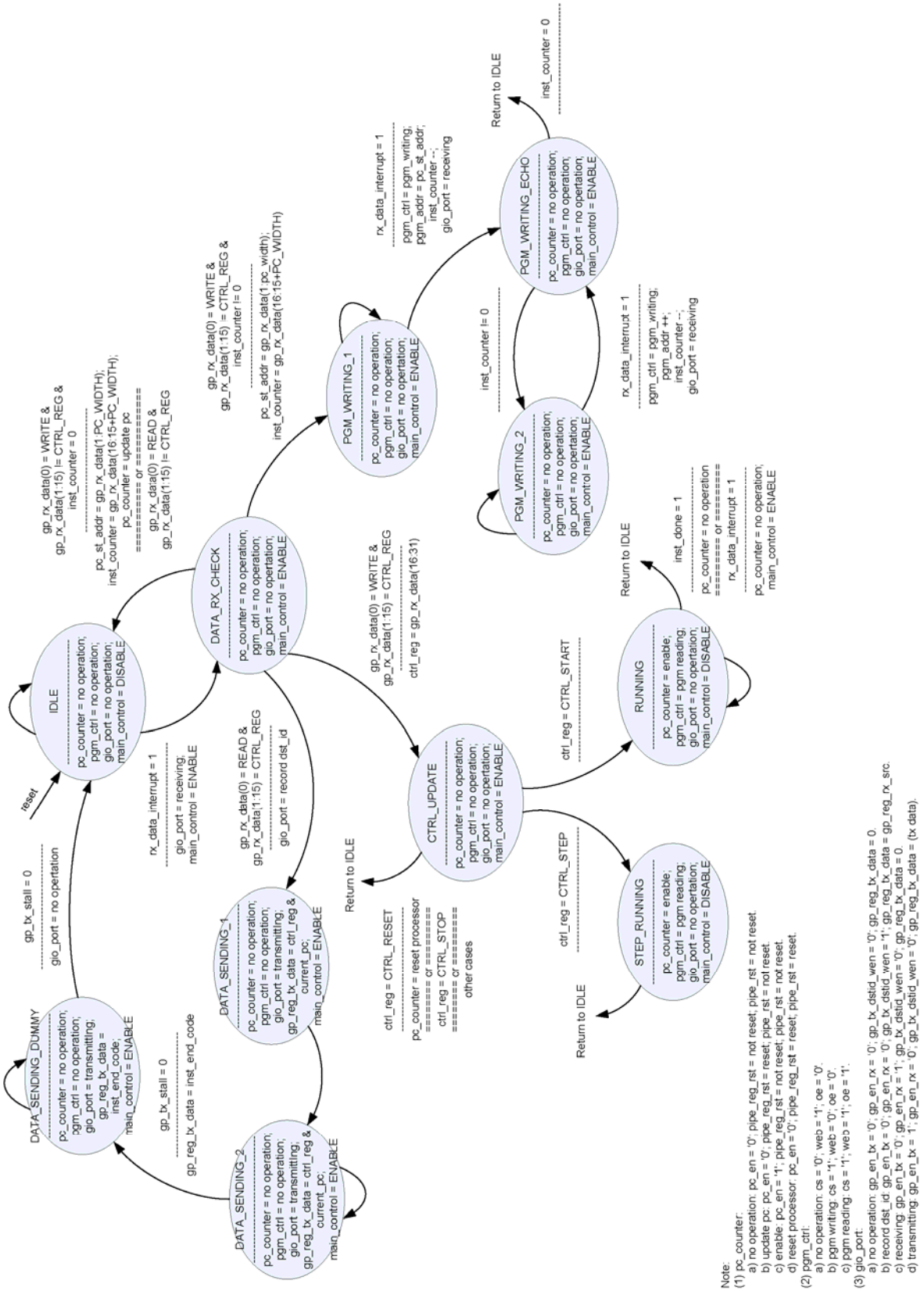


Figure-Appendix 5. Processor cell control flow

Appendix E – Processor cell control instruction set

CGRA Processor Cell Control Instruction Set

Operation	Description	Direction	Data package 31-21	20	19	18	17	16	15-1	0
Program download Header	Package header for downloading program into ProGram Memory (PGM). Address 0 is reserved for the Control register, therefore instructions start from address 1.	Host -> Processor	Count						Starting address	Write
Program download	Instruction to be downloaded into ProGram Memory. This is a consecutive operation of the "Program download header", where the instruction address is specified and controlled by the "Program download header" operation.	Host -> Processor	Instruction							
PC counter update	Update PC counter starting address, to select program section inside ProGram Memory (PGM).	Host -> Processor	X"0000"						Starting adress	Write
Control register update	Update Control register to control the operations of the Processor Cell.	Host -> Processor			Step	Reset	Stop	Start	X"0000"	Write
Control register read request	Control register status reading request.	Host -> Processor							X"0000"	Read
Control register read data 1	Sending back Control register status, data package 1. This is a consecutive operation of "Control register read request".	Processor -> Host			Step	Reset	Stop	Start	Current PC counter	
Control register read data 2	Sending back Control register status, data package 2. This is a consecutive operation of "Control register read data 1".	Processor -> Host							Instruction END code	

Note:
(1) Instruction END code has default value of "END_INVALIDE"; END code is assigned when processor finished executing one program section.
(2) Data package "Write": '0'; "Read": '1'.

Figure-Appendix 6. Processor cell control instruction set

Appendix F – Processing cell register addresses

CGRA Processor Cell Register Address Summary

Register name	*Address*
General Purpose Registers	
$0	00000
$1	00001
$2	00010
$3	00011
$4	00100
$5	00101
$6	00110
$7	00111
$8	01000
$9	01001
$10	01010
$11	01011
$12	01100
$13	01101
$14	01110
$15	01111
$16	10000
$17	10001
$18	10010

Register name	*Address*
Special Purpose Registers	
PC	11100
MSR	11101
LACC	11110
HACC	11111
GID	10000
ILC	10001
ILP	10010
Local IO Registers	
L0	10011
L1	10100
L2	10101
L3	10110
L4	10111
L5	11000
L6	11001
L7	11010
Global IO Register	
G0	11011

CGRA CORDIC Cell Register Address Summary

Register name	*Address*
Local IO Registers	
L0	0000
L1	0001
L2	0010
L3	0011
L4	0100
L5	0101
L6	0110
L7	0111
Global IO Register	
G0	1000
INVALID	1111

Figure-Appendix 7. Processor cell register address summary

Appendix G – CORDIC cell control flow

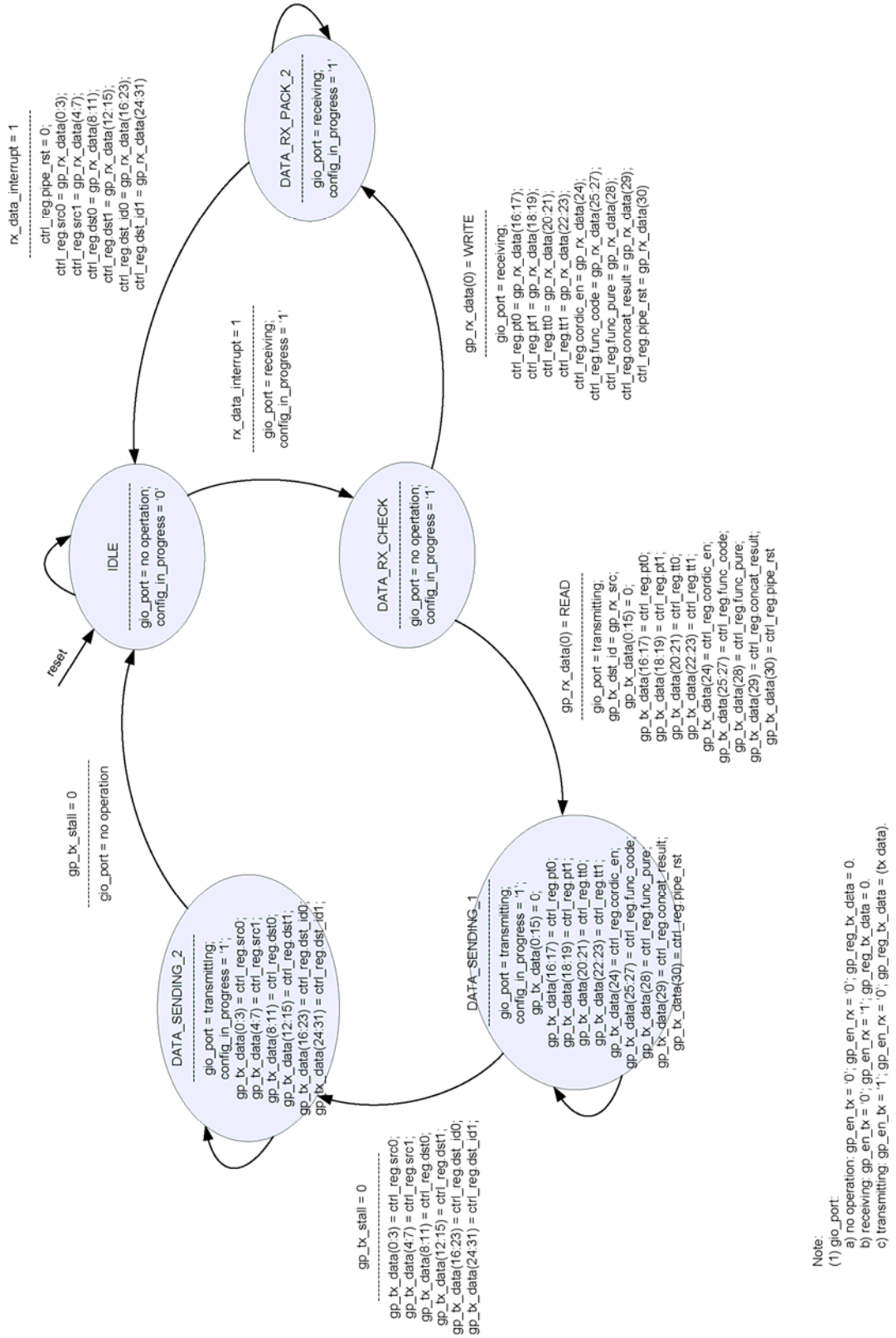


Figure-Appendix 8. Control flow of CORDIC cell configuration controller

Appendix H – CORDIC cell control instruction set

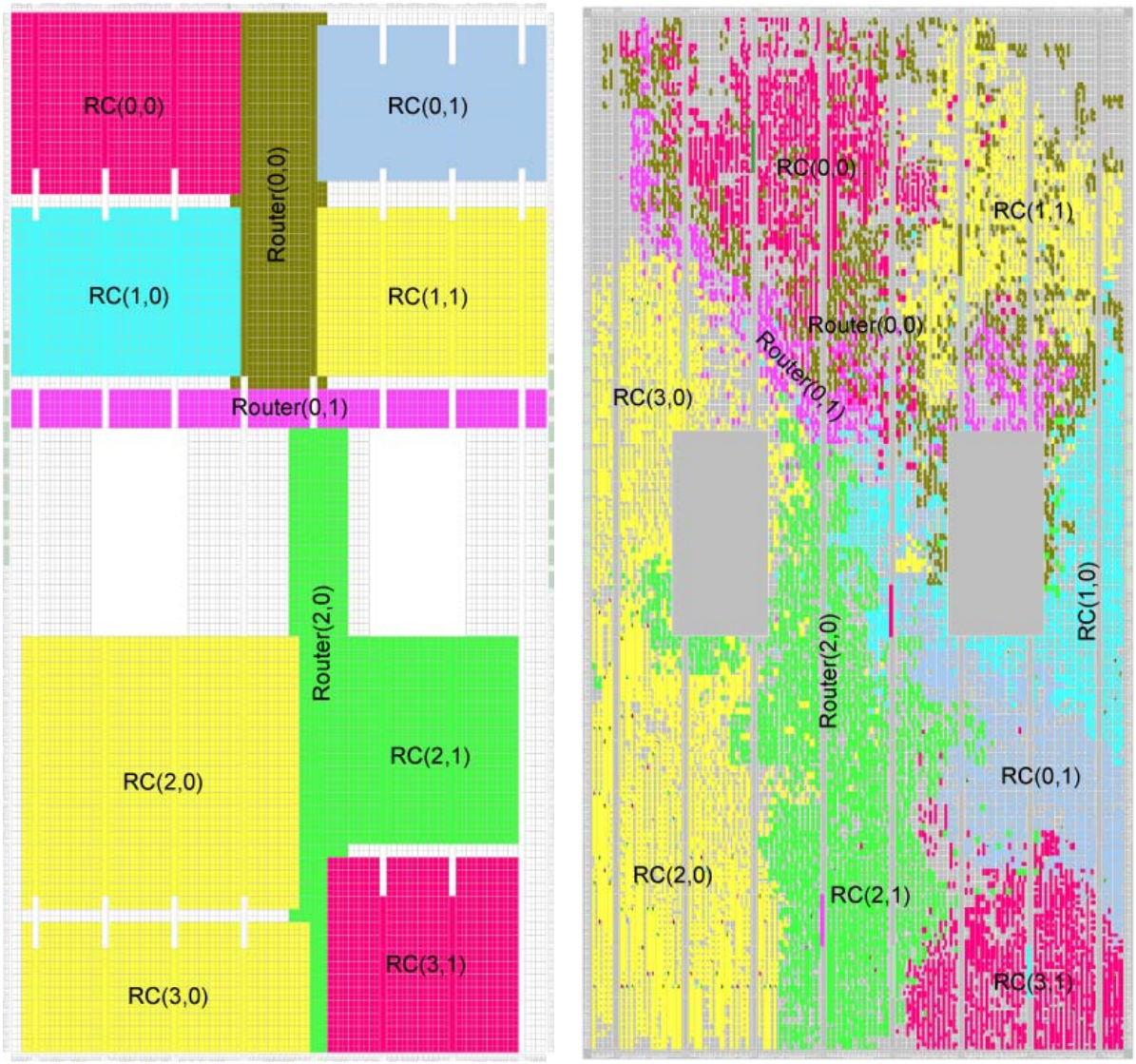
CGRA CORDIC Cell Control Instruction Set

Operation	Description	Direction	Data package 31	30	29	28	27-25	24	23-22	21-20	19-18	17-16	15-12	11-8	7-4	3-1	0
Control register update 1	Control register update package 1. PT0_PT1: Local IO TX data package type; TT0_TT1: Global IO TX data package type; E: CORDIC unit enable signal; F: CORDIC function code; P: CORDIC pure function option; C: Option of combining CCRDIC x & y results. R: Active high cell reset, hardware releases reset automatically.	Host -> Processor		R	C	P	F	E	TT1	TT0	PT1	PT0					Write
Control register update 2	Control register update package 2. This is a consecutive operation of "Control register update 1". SRC0_SRC1: Input data port number; DST0_DST1: Output data port number; DST_ID0_DST_ID1: Global output data destination routing ID	Host -> Processor	DST_ID1						DST_ID0				DST1	DST0	SRC1		SRC0
Control register read request	Control register status reading request.	Host -> Processor															Read
Control register read data 1	Sending back Control register status, data package 1. This is a consecutive operation of "Control register read request".	Processor -> Host		R	C	P	F	E	TT1	TT0	PT1	PT0					
Control register read data 2	Sending back Control register status, data package 2. This is a consecutive operation of "Control register read data 1".	Processor -> Host	DST_ID1						DST_ID0				DST1	DST0	SRC1		SRC0

Note:
(1) Data package "Write": '0'; "Read": '1'.
(2) Local IO TX data package type:
rpDATA : 00;
rpRDREQ : 01;
rpWRREQ : 10;
rpINVALID : 11;
(3) Local IO TX data package type:
rtDATA : 00;
rtCONFIG : 01;
rtCONTROL : 10;
rpINVALID : 11;
(4) CORDIC function code:
CORDIC_FUNC_MULT = 000;
CORDIC_FUNC_SINCOS = 001;
CORDIC_FUNC_SINHCOSH = 011;
CORDIC_FUNC_DIV = 100;
CORDIC_FUNC_ABSPHASE = 101;
CORDIC_FUNC_ABSHPHASEH = 111;

Figure-Appendix 9. CORDIC cell control instruction set

Appendix I – Layout of a 4-by-2 CGRA cell array



(a) Floorplan (b) Final routed layout

Figure-Appendix 10. Floorplan and final layout of a 4-by-2 CGRA cell array platform. The architecture contains four 16-bit MAC processor cells [RC(0,0), RC(1,1), RC(2,1), RC(3,0)], three 8K bits memory cells [RC(0,1), RC(1,0), RC(3,1)] and one 16-bit CORDIC cell [RC(2,0)]. The floorplan is designed in Xilinx Floorplanner, which is used as one of the user constraints for the automatic place & route process. Design is synthesized, placed and routed for exploring the maximum speed, results in a 76% of the FPGA slice usage and is capable of operating up to 30.71MHz. Notice that, there are minor deviations on the actual cell placement between the final routed layout and the floorplan. But the integrity of each resource cell has been preserved.

Appendix J – User commands in serial line interface

Table-Appendix 1. User commands in serial line interface.

Command	Description	Parameter
g	Select destination cell ID for global I/O communications.	Resource cell destination ID (number input): 0 ~ 7.
d	Send data inputs to the selected destination cell.	a) Data amount (number input); b) Data inputs (string input).
i	Send instructions or configuration packages to the selected destination cell.	a) Instruction amount (number input); b) Inst./Config. Inputs (string input).
s	Send “start” command to the selected processor cell.	None.
e	Send “step” command to the selected processor cell.	None.
p	Send “stop” command to the selected processor cell.	None.
r	Send “reset” command to the selected processor cell.	None.
t	Trace control register status from the selected destination cell. Two consecutive data packages will be sent back through global port.	None.
c	Send special user command to the selected destination cell.	User command (string input).
f	Memory cell data storage initialization (zero filling).	Memory cell destination ID (number input): 1, 2, 7.
1	Run 32-point radix-2^2 FFT *partial* configuration script.	None.
2	Run 64- point radix-2^2 FFT *partial* configuration script.	None.
3	Run 128- point radix-2^2 FFT *partial* configuration script.	None.
4	Run 256- point radix-2^2 FFT *partial* configuration script.	None.
5	Run 512- point radix-2^2 FFT *partial* configuration script.	None.
6	Run 1024- point radix-2^2 FFT *partial* configuration script.	None.
0	Run 1024- point radix-2^2 FFT *full* configuration script.	None.
h	Command help printout.	None.

Appendix K – User commands in MATLAB interface

Table-Appendix 2. User commands in MATLAB interface.

Command	Description	Parameter
cmd	User command input in serial line interface. The follow-up input prompt is a transparent window in serial line interface.	User command in serial line interface.
config	Send instructions or configuration packages from a script file. The script file name is defined in "send_config.m".	None.
data	Send global data inputs from a script file. The script file name is defined in "send_data.m".	None.
demo	Run a script demo: 32 ~ 1024-point radix-2^2 FFT computation.	None.
rxbuf	Read MATLAB UART RX buffer. This could be used to flush out the remaining data packages in RX buffer.	None.
help	Command help printout.	None.
exit	Exit user interface in MATLAB.	None.