Lecture 3: EITF20 Computer Architecture

Anders Ardö

EIT – Electrical and Information Technology, Lund University

November 12, 2014

Outline

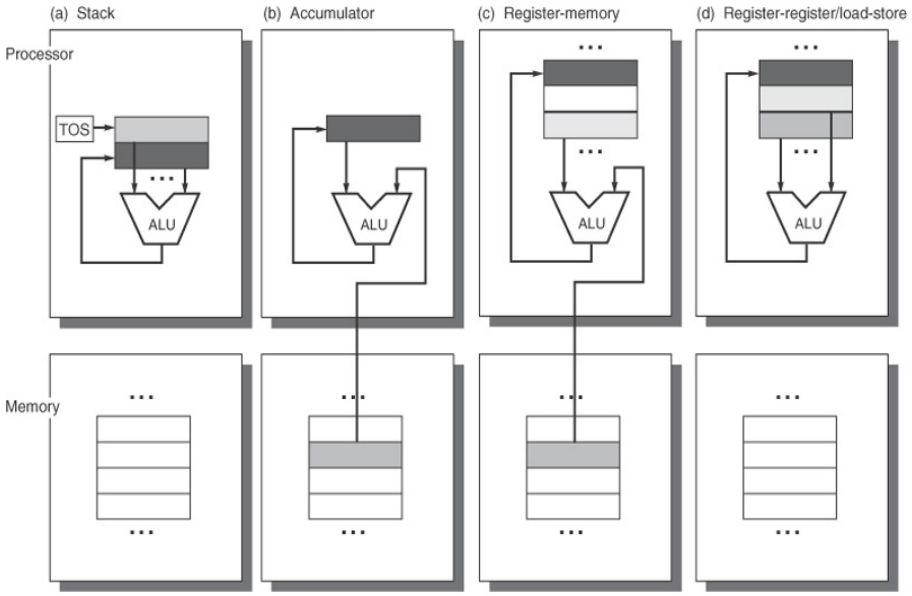
Previous lecture

Instruction Set Architectures

Lecture 2 - Instruction set principles

- Classification of instruction sets
- What's needed in an instruction set?
  - Addressing
  - Operands
  - Operations
  - Control Flow
- Encoding
- The impact of the compiler
- The MIPS instruction set architecture

Instruction set architecture classes



How can you aid the compiler?

- Rules of thumb when designing an instruction set:
  - Regularity (operations, data types and addressing modes should be orthogonal)
  - Provide primitives, not high-level constructs or solutions. Complex instructions are often too specialized.
  - Simplify trade-offs among alternatives
  - Provide instructions that bind quantities known at compile time as constants

Summary - ISA

- The instruction set architecture have importance for the performance
- The important aspects of an ISA are:
  - register model
  - addressing modes
  - types of operations
  - data types
  - encoding
- Benchmark measurements can reveal the most common case
- Interaction compiler - ISA important

Questions!

QUESTIONS?

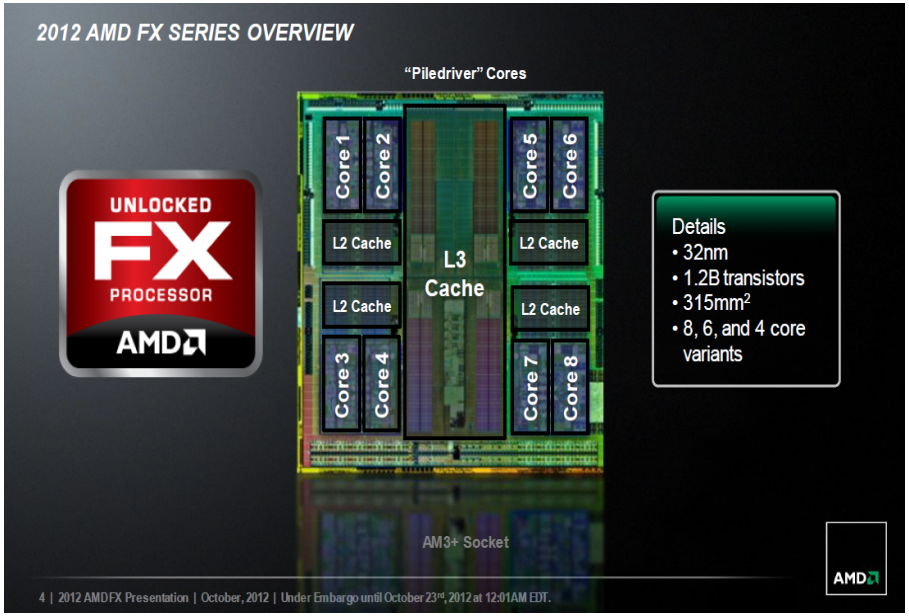
COMMENTS?

Lecture 3 agenda

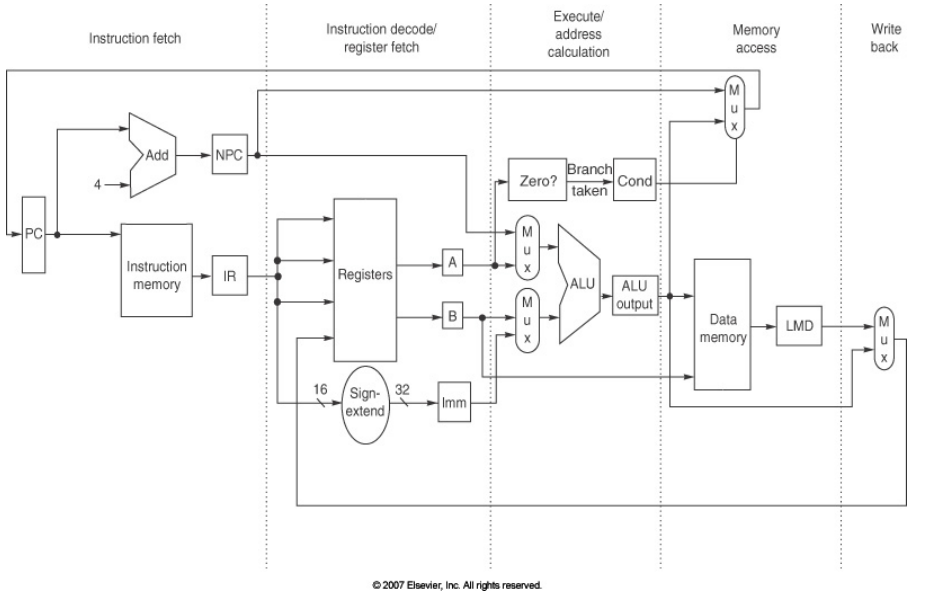
Appendix A.1 - A.5 in "Computer Architecture"

Outline

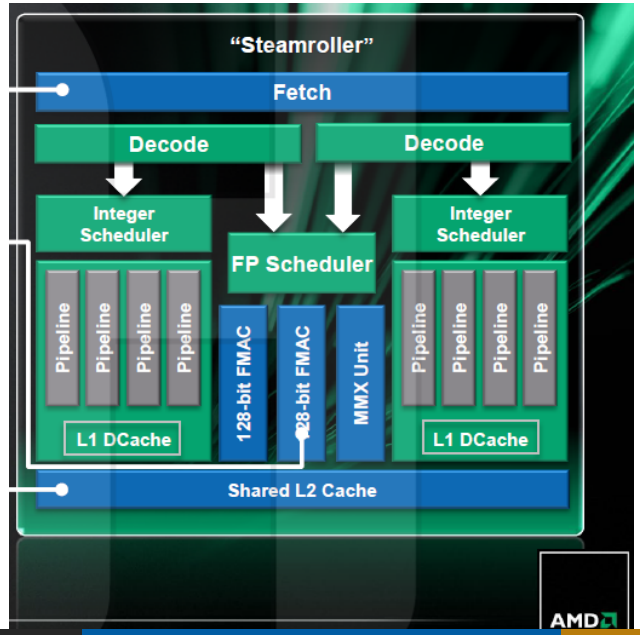
CPU chip



One core - The MIPS datapath



Modern CPU core architecture

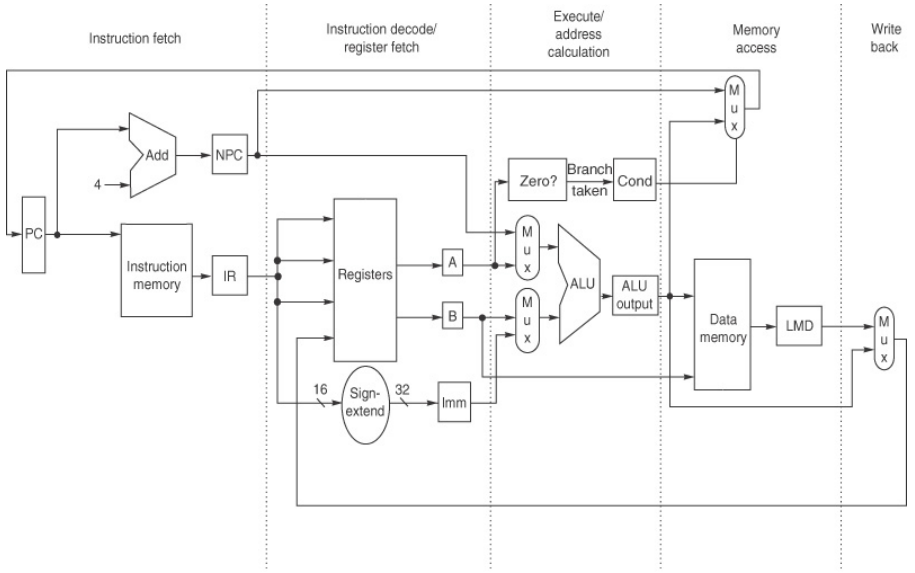


Lecture 3

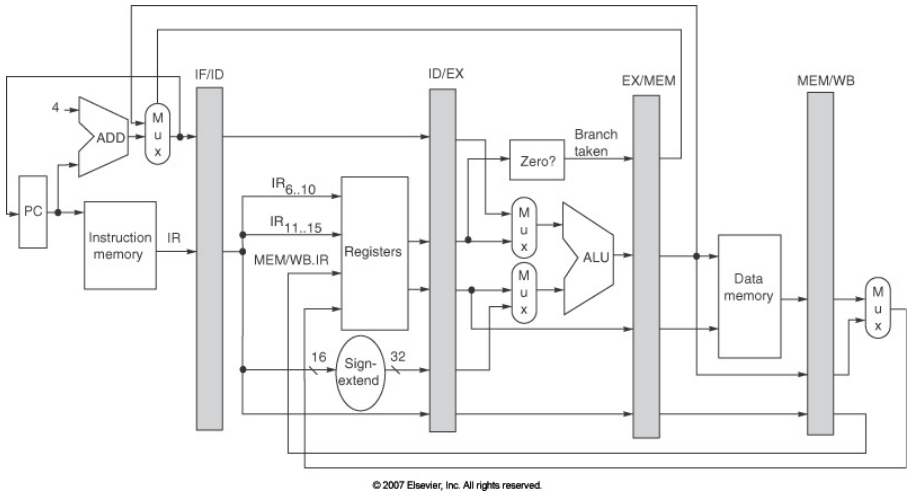
Appendix A.1 - A.5 in "Computer Architecture"

- General principles of pipelining
  (review from EIT070 Computer Organization)
- Techniques to avoid pipeline stalls due to hazards
- What makes pipelining hard to implement?
- Support of multi-cycle instructions in a pipeline

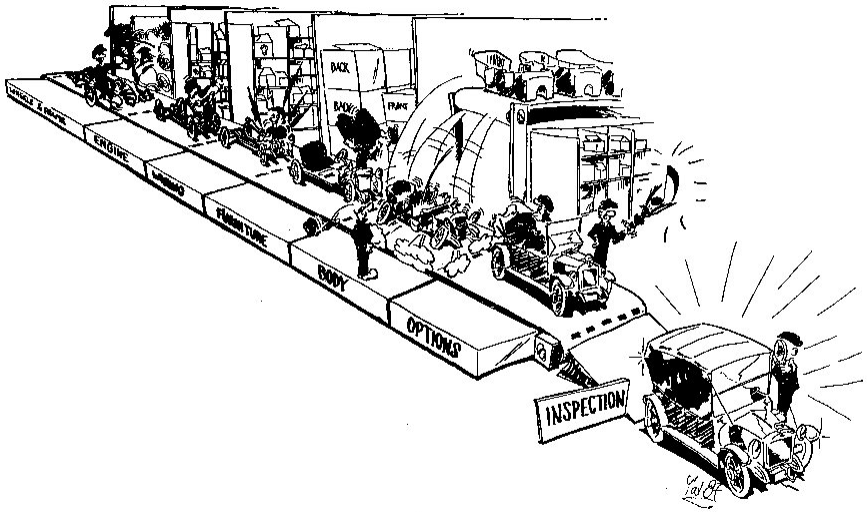
The MIPS datapath



A Pipelined MIPS Datapath



The Assembly Line ...



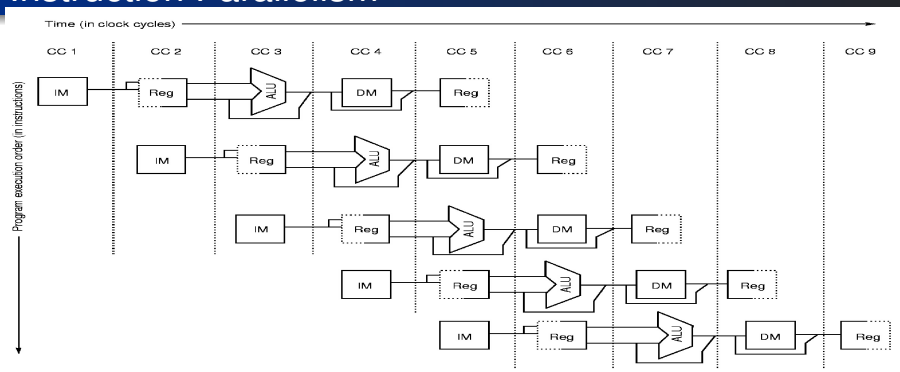
Classic RISC five stage pipeline

1. **IF** Instruction fetch
2. **ID** Instruction decode/register fetch
3. **EX** Execution/effective address
4. **MEM** Memory access
5. **WB** Write back

Pipelining Lessons

- Pipelining doesn't help latency of a single instruction, it helps throughput of the entire workload
- Pipeline rate is limited by the slowest pipeline stage
- Multiple instructions are executing simultaneously
- Potential speedup = Number of pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to fill pipeline and time to drain reduces speedup

Instruction Parallelism



	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Fundamental Limitations

- **Hazards** can prevent next instruction from executing during its designated clock cycle:
  - **Structural hazards**: Simultaneous use of a HW resource
  - **Data hazards**: Data dependencies between instructions
  - **Control hazards**: Change in program flow

A way of solving hazards is to serialize the execution by inserting **"bubbles"** that effectively **stall** the pipeline.

Pipeline speedup

$$\text{CPI}_{\text{pipelined}} = \text{IdealCPI} + \text{\# stall-cycles/instruction}$$

$$\textit{Speedup} = \frac{\text{AverageInstructionTime}_{\text{unpipelined}}}{\text{AverageInstructionTime}_{\text{pipelined}}}$$

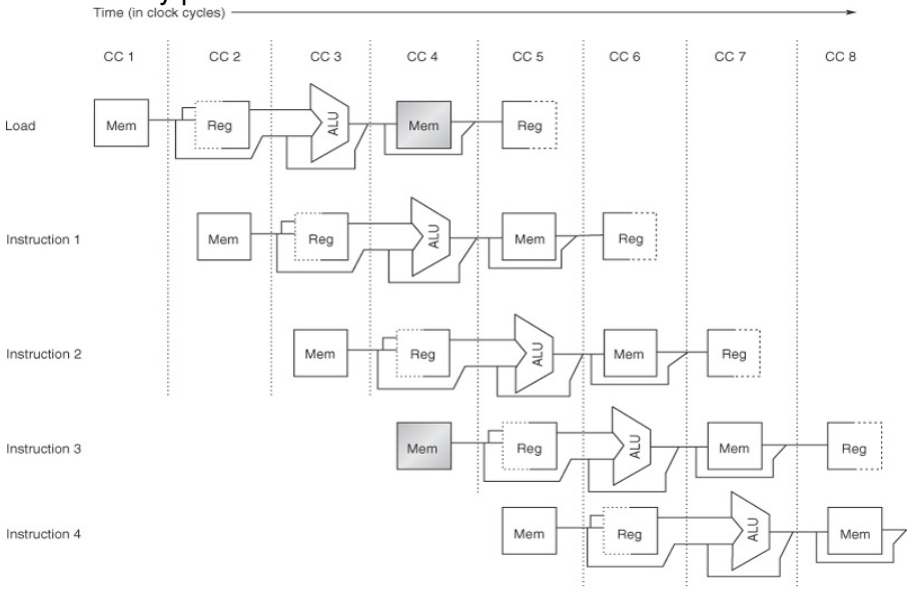
$$\textit{Speedup} = \frac{\text{CPI}_{\text{unpipelined}}}{\text{Ideal CPI} + \text{\# stall-cycles/instr}} * \frac{\text{T}C_{\text{unpipelined}}}{\text{T}C_{\text{pipelined}}}$$

$$\approx \frac{\#\textit{stages}}{1 + \text{\# stall-cycles/instruction}}$$

Outline

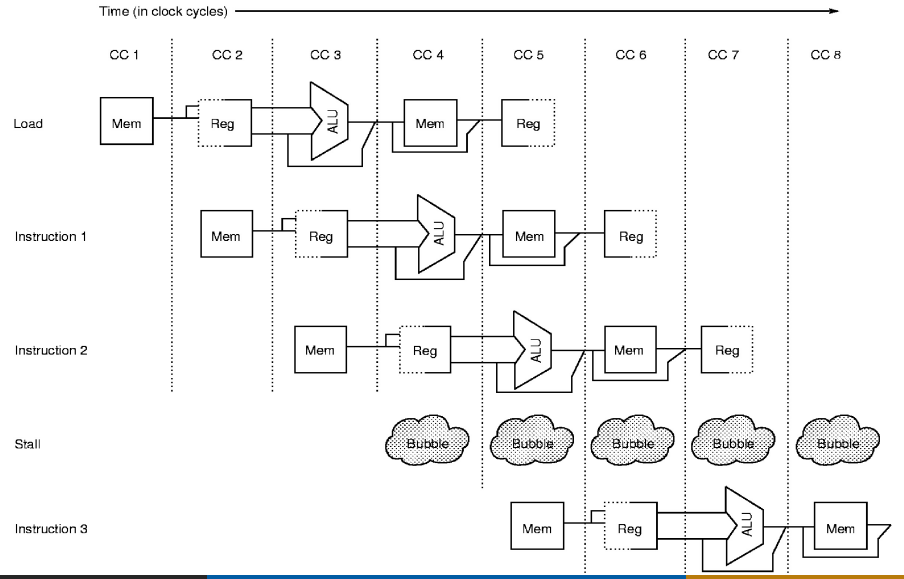
Example of Structural Hazard

One memory port



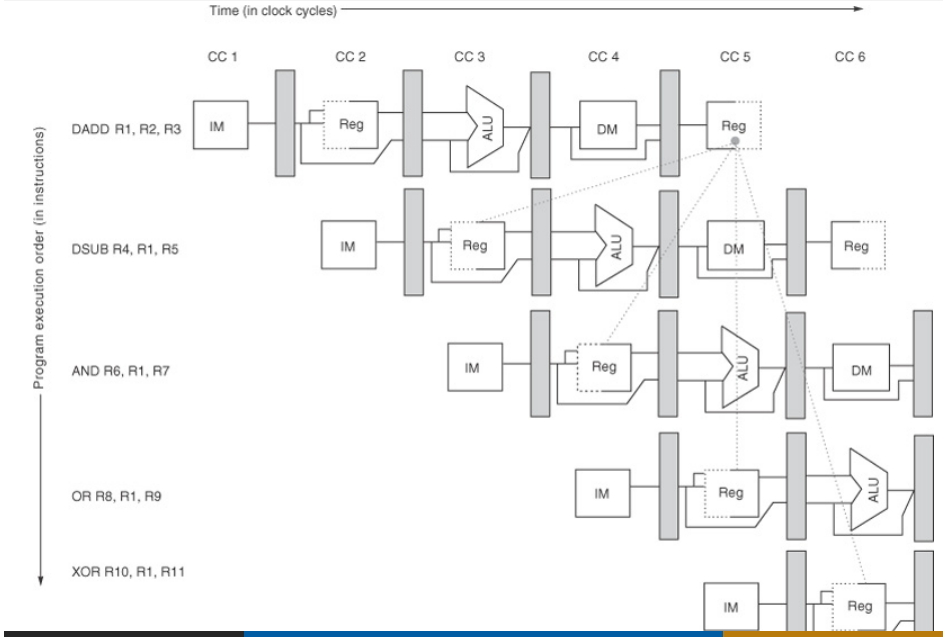
One Memory Port/Structural Hazard

A Simple Solution



Outline

Data Hazard on R1



Fundamental Types of Data Hazards

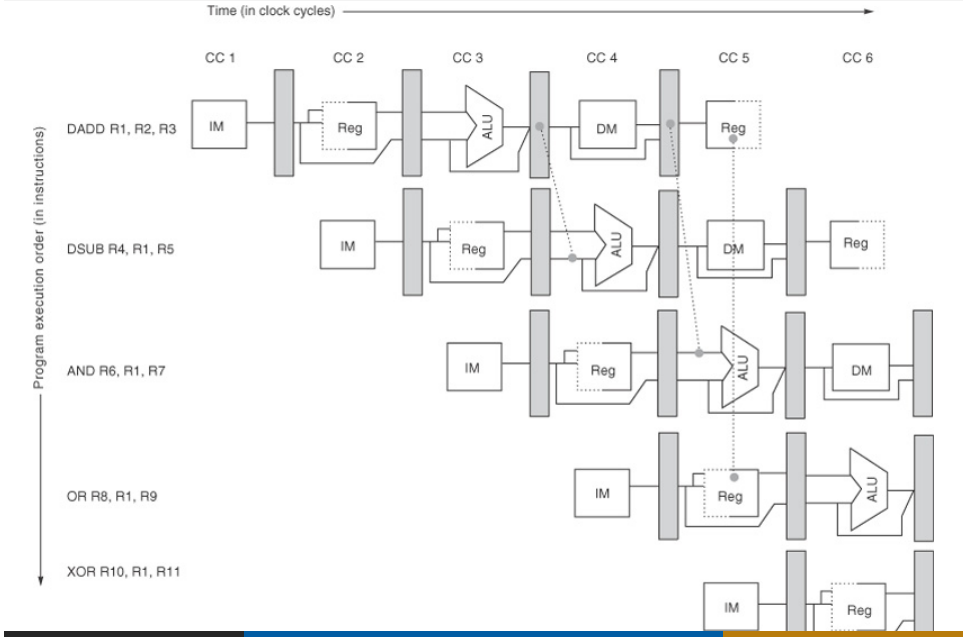
Code sequence: Op_i A
Op_{i+1} A

- **RAW** (Read-After-Write) Instruction $i+1$ reads A and i modifies A. Instruction i+1 reads old A!
- **WAR** (Write-After-Read) Instruction $i+1$ modifies A and instruction i reads new A
- **WAW** (Write-After-Write) Instructions i and $i+1$ both modifies A. The value in A is the one written by instruction i
- (*RAR*?)

Software Scheduling of Loads

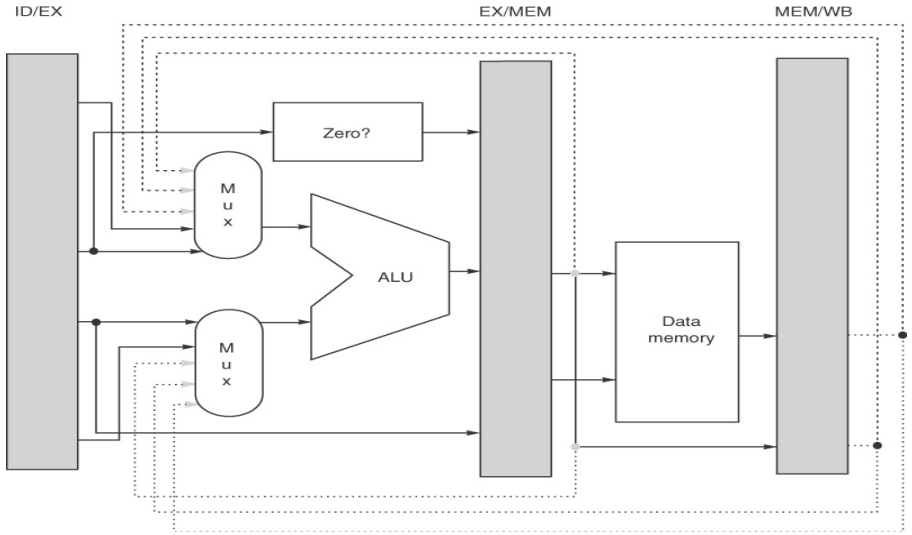
Try producing fast code for
a = b + c;
d = e - f;
assuming a, b, c, d, e, and f are in memory.

Forwarding to Avoid Data Hazards

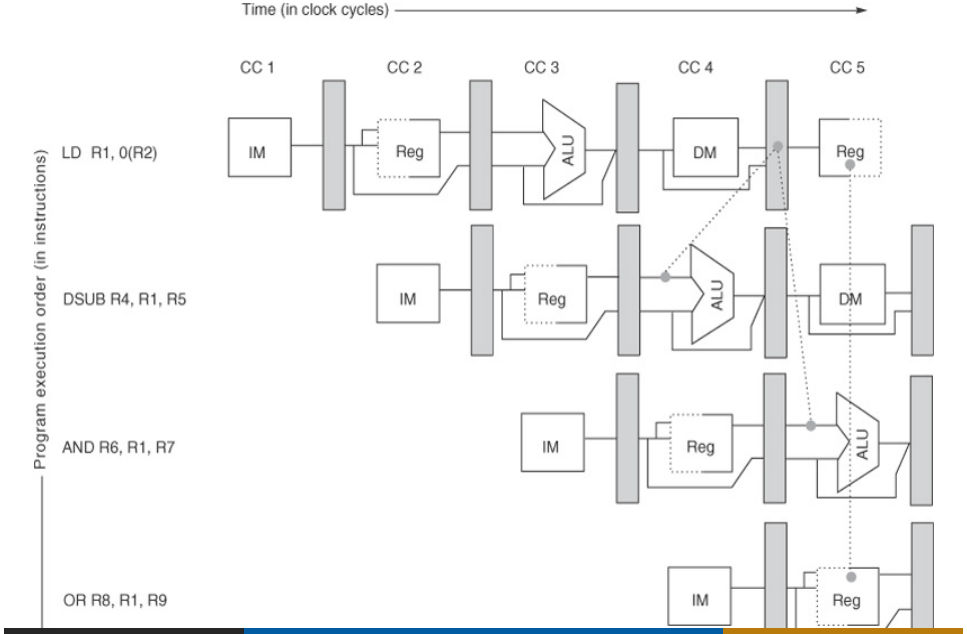


HW Support for Forwarding

- Insert bypass paths from EX/MEM and MEM/WB
- Wider MUXes at the ALU inputs

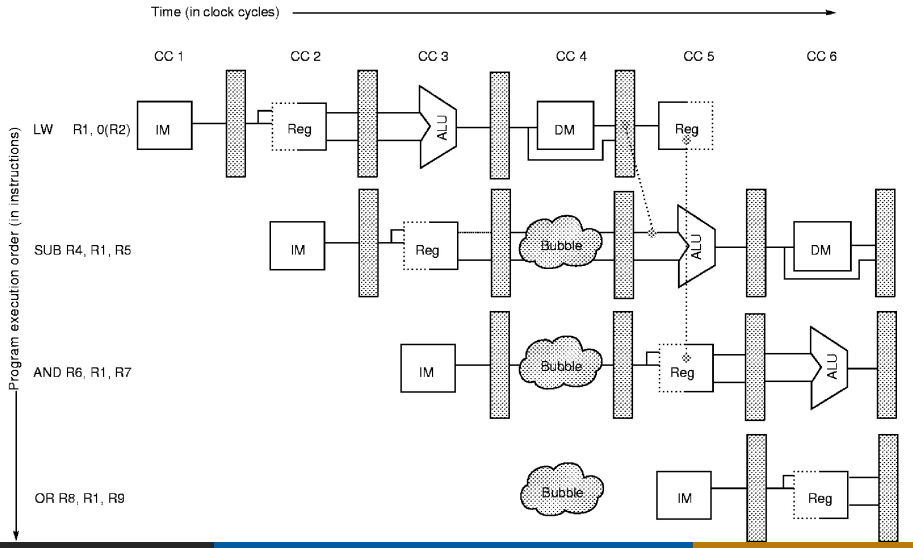


Data Hazard with Forwarding



Data Hazard with Forwarding

- Stall the pipeline!



Software Scheduling of Loads

Try producing fast code for
a = b + c;
d = e - f;
assuming a, b, c, d, e, and f are in memory.

```
LD    R1, B
LD    R2, C
DADD  R3,R1,R2
SD    R3, A
LD    R5, F
LD    R4, E
DSUB  R6,R4,R5
SD    R6, D
```

How many stalls?
How many stalls with hardware forwarding?

Software Scheduling of Loads

Try producing fast code for
a = b + c;
d = e - f;
assuming a, b, c, d, e, and f are in memory.

		Code re-order	
LD	R1, B	LD	R1, B
LD	R2, C	LD	R2, C
DADD	R3,R1,R2	LD	R4, E
SD	R3, A	LD	R5, F
LD	R5, F	DADD	R3,R1,R2
LD	R4, E	DSUB	R6,R4,R5
DSUB	R6,R4,R5	SD	R3, A
SD	R6, D	SD	R6, D

How many stalls?
How many stalls with hardware forwarding?

Outline

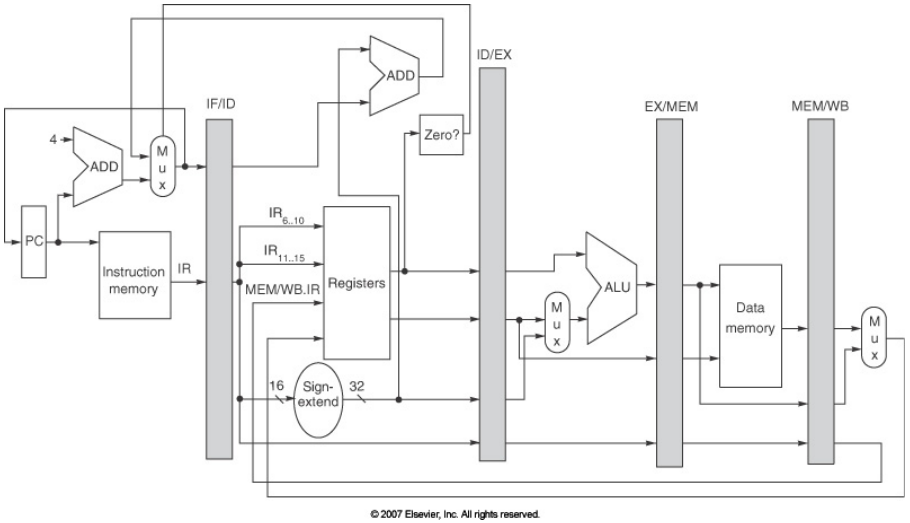
Control Hazards

Branch instruction	IF	ID	EX	MEM	WB				
Branch successor		IF	*stall*	*stall*	IF	ID	EX	MEM	WB
Branch successor + 1						IF	ID	EX	MEM	WB
Branch successor + 2							IF	ID	EX	MEM
Branch successor + 3								IF	ID	EX
Branch successor + 4									IF	ID
Branch successor + 5										IF

- Branches are not resolved until the MEM stage
- Three wasted clock cycles:
  - two stalls
  - one extra instruction fetch (IF)
- If branch is not taken, the extra IF not needed

HW Support to Reduce Control Hazard

- Calculate target address and test condition in ID
- 1 clock cycle branch penalty instead of 3!



Four Branch Hazard Alternatives

1. Stall until branch condition and target is known
2. Predict Branch not taken
   - Execute successor instructions in sequence
   - "Squash" instructions in pipeline if the branch is actually taken
   - Works well if state is updated late in the pipeline (as in MIPS)
   - 33 % MIPS conditional branches not taken on average

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure A.12 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and

tinue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following

Four Branch Hazard Alternatives

3. Predict Branch taken
   - 67 % MIPS conditional branches taken on average
   - MIPS calculates target address in ID stage! Still one cycle penalty
4. Delayed branch
   - Define branch to take place after a following instruction
   - 1 slot delay allows proper decision and branch target address calculation in a 5 stage pipeline such as MIPS

Untaken branch instruction	IF	ID	EX	MEM	WB			
Branch *delay instruction (i + 1)*		*IF*	*ID*	*EX*	*MEM*	*WB*		
Instruction $i+2$			IF	ID	EX	MEM	WB	
Instruction $i+3$				IF	ID	EX	MEM	WB
Instruction $i+4$					IF	ID	EX	MEM	WB

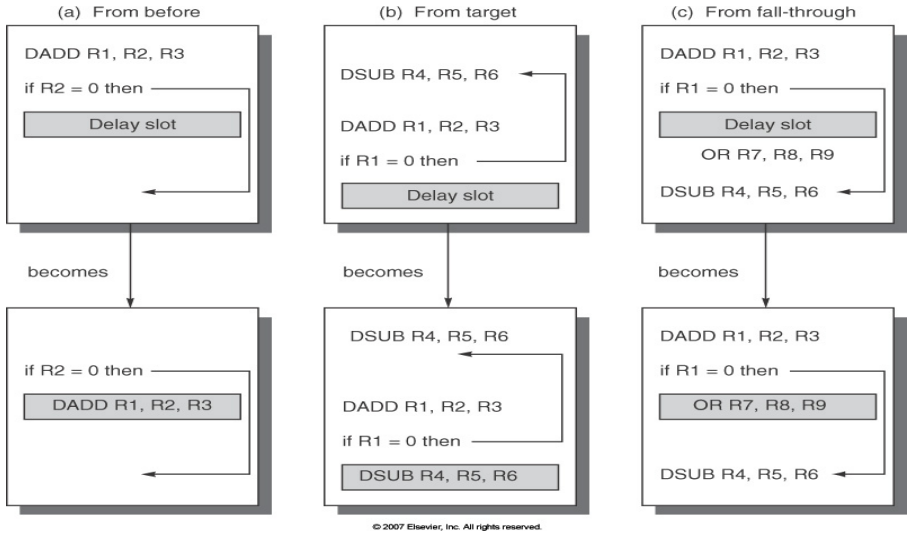
Taken branch instruction	IF	ID	EX	MEM	WB			
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB		
Branch target			IF	ID	EX	MEM	WB	
Branch target + 1				IF	ID	EX	MEM	WB
Branch target + 2					IF	ID	EX	MEM	WB

Figure A.13 The behavior of a delayed branch is the same whether or not the branch is taken. The instructions in

the instruction after the branch delay instruction; if the branch is taken, execution continues at the branch target. When the instruction in the branch delay slot is also a branch, the meaning is unclear: If the branch is not taken, what

Compiler Support for Delayed Branch

- Scheduling “from before” is always safe
- Scheduling “from target” or “fall through” is not always safe



Outline

What’s hard to implement?

- Exceptions
- Complicated instruction set architectures

Precise Exceptions

A pipeline implements precise exceptions if:

- All instructions before the faulting instruction can complete
- All instructions after (and including) the faulting instruction can safely be restarted

Some exceptions must be precise: e.g. page faults

Exceptions are Difficult in a Pipeline

We need to be able to restart an instruction that causes an exception (interrupt, trap, fault):

- Force a trap instruction into the pipeline
- Turn off all writes for the faulting instruction
- Save the PC for the faulting instruction
  - to be used in return from exception handling
  - if delayed branch is used we may need to save several PC's

More Problems with Exceptions

Exceptions may be generated out-of (program) order

IF	ID	EX	MEM	WB
page fault on instruction fetch, misaligned memory access, protection violation	undefined or illegal opcode	arithmetic exception	page fault on data fetch, misaligned memory access, memory protection	none

LD (faults in MEM)
DADD (faults in IF)
$\Longrightarrow$ The DADD faults before the LD

Solution for simple MIPS

- add a hardware status vector containing exceptions
- pass along with instruction in the pipeline
- turn of writes when an exception entered in the status vector
- handle exceptions from status vector in WB (in program order)

Instruction Set Complications

Implicit condition codes:

- In architectures like VAX, M68000 and IBM 360, almost all instructions affect the condition codes
- Difficult to reorder instructions if needed!

Example: a = b+d;
if (b == 0) ...

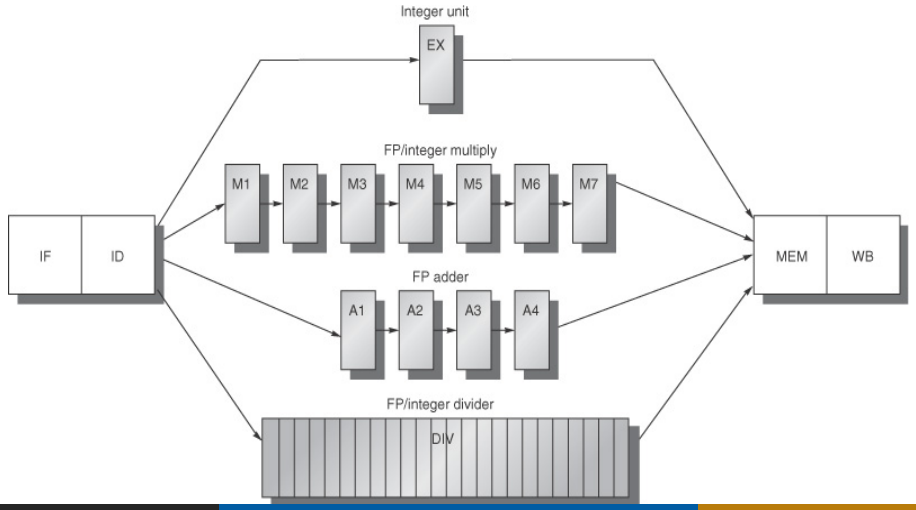
VAX code	**MIPS code**
ADDL A,R2,R3	DADD R1,R2,R3
CL R2,0	SW A,R1
BEQL lable	BEQZ R2,lable

Multicycle operations like move string

Outline

Multi-cycle Instructions in the Pipeline (FP)

- Integer unit: Handles integer instructions, branches and loads/stores
- Floating point units: May take several cycles each



Parallelism Between Integer and FP

MUL.D	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
ADD.D		IF	ID	A1	A2	A3	A4	MEM	WB		
DSUB			IF	ID	EX	MEM	WB				
LD				IF	ID	EX	MEM	WB			

- Instructions are issued in order
- Instructions may be completed out of order

Structural and RAW hazards

- To resolve these hazards:
  - Structural hazards. Stall in ID stage if:
    - the functional unit is occupied (applicable to DIV only)
    - any instruction already executing will reach the MEM/WB stage at the same time as this one
  - RAW hazards:
    - Normal bypassing from MEM and WB stages
    - Stall in ID stage if any of the source operands is destination operand in any of the FP functional units

Resolving WAR and WAW Hazards

- There are no WAR-hazards since the operands are read (in ID) before the EX-stages in the pipeline
- example of a WAW hazard
  DIV.D F0,F2,F3 FP divide 24 cycles
  ...
  SUB.D F0,F8,F10 FP subtract 3 cycles
  SUB finishes before DIV which will overwrite the result from SUB!
- WAW hazards are eliminated by:
  Stalling SUB until DIV reaches MEM stage

To Interrupt or not to Interrupt

	DIV.D	F0,F2,F4	(24 cycles)
Example:	ADD.D	F10,F10,F8	(3 cycles)
	SUB.D	F12,F12,F14	(3 cycles)

Suppose

- the SUB instruction generates an arithmetic trap
- DIV instruction hasn't completed
- ADD instruction have completed

- **Imprecise interrupt signaling**

Another problem with out-of-order completion

Handling imprecise interrupts

- ***Ignore***
  - Pro: Simple and high performance
  - Con: It is not possible to tie the interrupt to a certain instruction
- ***Buffer the results*** of earlier operations
  - Con: may require large buffers and high complexity hardware
- ***Somewhat imprecise interrupts*** but with state information so that trap-handling routines can create a precise instruction sequence
- ***Determine if interrupts are possible early*** in the pipeline (so that order is maintained at interrupt detection)

Outline

Summary

Pipelining (ILP):

- Speeds up throughput, not latency
- Speedup $\leq$ #stages
- Hazards limit performance, generate stalls:
  - Structural: need more HW
  - Data (RAW,WAR,WAW): need forwarding and compiler scheduling
  - Control: delayed branch, branch prediction

$$\begin{aligned} \textit{Speedup} &= \frac{\text{CPI}_{\text{unpipelined}}}{\text{Ideal CPI} + \text{\# stall-cycles/instr}} * \frac{\text{T}_{\text{C}_{\text{unpipelined}}}}{\text{T}_{\text{C}_{\text{pipelined}}}} \\ &\approx \frac{\#\textit{stages}}{1 + \text{\# stall-cycles/instruction}} \end{aligned}$$

Complications:

- Precise exceptions may be difficult to implement
- The instruction set can be more or less suited for pipelining