

Appendix A

Pipelining: Basic and Intermediate Concepts

1

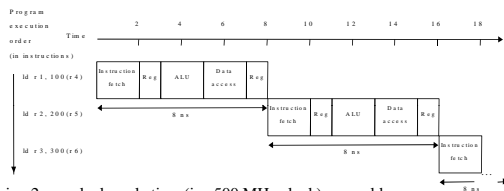
Overview

- Basics of Pipelining
- Pipeline Hazards
- Pipeline Implementation
- Pipelining + Exceptions
- Pipeline to handle Multicycle Operations

2

Unpipelined Execution of 3 LD Instructions

- Assumed are the following delays: Memory access = 2 nsec, ALU operation = 2 nsec, Register file access = 1 nsec;

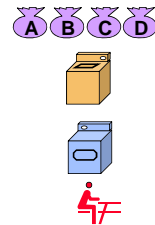


- Assuming 2nsec clock cycle time (i.e. 500 MHz clock), every ld instruction needs 4 clock cycles (i.e. 8 nsec) to execute.
- The total time to execute this sequence is 12 clock cycles (i.e. 24 nsec). CPI = 12 cycles/3 instructions = 4 cycles / instruction.

3

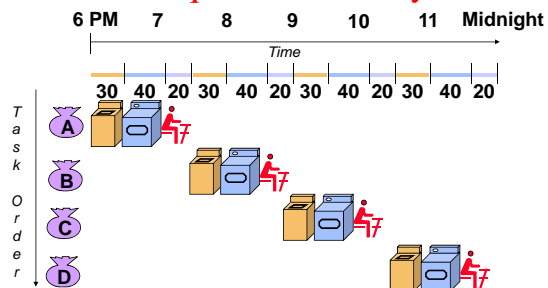
Pipelining: Its Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes



4

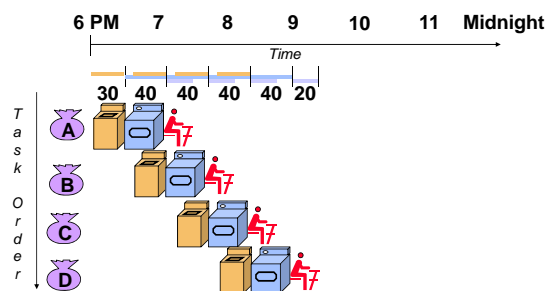
Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

5

Pipelined Laundry: Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

6

Key Definitions

Pipelining is a key implementation technique used to build fast processors. It allows the execution of multiple instructions to overlap in time.

A pipeline within a processor is similar to a car assembly line. Each assembly station is called a *pipe stage* or a *pipe segment*.

The **throughput** of an instruction pipeline is the measure of how often an instruction exits the pipeline.

7

Pipeline Stages

We can divide the execution of an instruction into the following 5 "classic" stages:

- IF:** Instruction Fetch
- ID:** Instruction Decode, register fetch
- EX:** Execution
- MEM:** Memory Access
- WB:** Register write Back

8

Pipeline Throughput and Latency



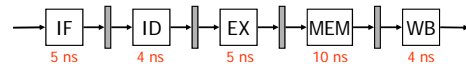
Consider the pipeline above with the indicated delays. We want to know what is the *pipeline throughput* and the *pipeline latency*.

Pipeline throughput: instructions completed per second.

Pipeline latency: how long does it take to execute a single instruction in the pipeline.

9

Pipeline Throughput and Latency



Pipeline throughput: how often an instruction is completed.

$$= \text{instr} / \max[\text{lat}(\text{IF}), \text{lat}(\text{ID}), \text{lat}(\text{EX}), \text{lat}(\text{MEM}), \text{lat}(\text{WB})]$$

$$= \text{instr} / \max[5\text{ns}, 4\text{ns}, 5\text{ns}, 10\text{ns}, 4\text{ns}]$$

$$= \text{instr} / 10\text{ns} \quad (\text{ignoring pipeline register overhead})$$

Pipeline latency: how long does it take to execute an instruction in the pipeline.

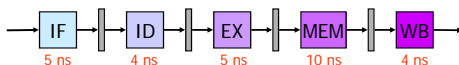
$$L = \text{lat}(\text{IF}) + \text{lat}(\text{ID}) + \text{lat}(\text{EX}) + \text{lat}(\text{MEM}) + \text{lat}(\text{WB})$$

$$= 5\text{ns} + 4\text{ns} + 5\text{ns} + 10\text{ns} + 4\text{ns} = 28\text{ns}$$

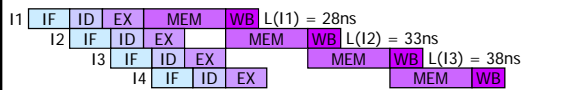
Is this right?

10

Pipeline Throughput and Latency



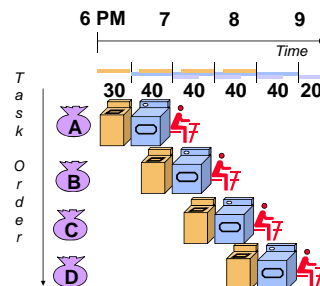
Simply adding the latencies to compute the pipeline latency, only would work for an isolated instruction



We are in trouble! The latency is not constant. This happens because this is an unbalanced pipeline. The solution is to make every state the same length as the longest one.

11

Pipelining Lessons



- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest** pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

12

Other Definitions

- Pipe stage or pipe segment
 - A decomposable unit of the fetch-decode-execute paradigm
- Pipeline depth
 - Number of stages in a pipeline
- Machine cycle
 - Clock cycle time
- Latch
 - Per phase/stage local information storage unit

13

Design Issues

- Balance the length of each pipeline stage

$$\text{Throughput} = \frac{\text{Depth of the pipeline}}{\text{Time per instruction on unpipelined machine}}$$

- Problems
 - Usually, stages are not balanced
 - Pipelining overhead
 - Hazards (conflicts)
- Performance (throughput → CPU performance equation)
 - Decrease of the CPI
 - Decrease of cycle time

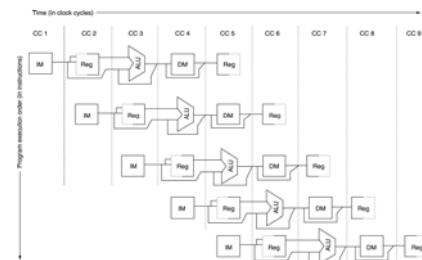
14

Basic Pipeline

	Clock number								
Instr #	1	2	3	4	5	6	7	8	9
<i>i</i>	IF	ID	EX	MEM	WB				
<i>i</i> + 1		IF	ID	EX	MEM	WB			
<i>i</i> + 2			IF	ID	EX	MEM	WB		
<i>i</i> + 3				IF	ID	EX	MEM	WB	
<i>i</i> + 4					IF	ID	EX	MEM	WB

15

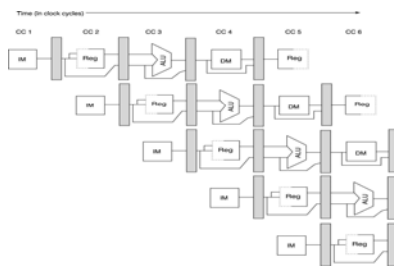
Pipelined Datapath with Resources



© 2003 Elsevier Science (USA). All rights reserved.

16

Pipeline Registers



© 2003 Elsevier Science (USA). All rights reserved.

17

Physics of Clock Skew

- Basically caused because the clock edge reaches different parts of the chip at different times
 - Capacitance-charge-discharge rates
 - All wires, leads, transistors, etc. have capacitance
 - Longer wire, larger capacitance
 - Repeaters used to drive current, handle fan-out problems
 - C is inversely proportional to rate-of-change of V
 - Time to charge/discharge adds to delay
 - Dominant problem in old integration densities.
 - For a fixed C, rate-of-change of V is proportional to I
 - Problem with this approach is power requirements go up
 - Power dissipation becomes a problem.
 - Speed-of-light propagation delays
 - Dominates current integration densities as nowadays capacitances are much lower.
 - But nowadays clock rates are much faster (even small delays will consume a large part of the clock cycle)
- Current day research → asynchronous chip designs

18

Performance Issues

- Unpipelined processor
 - 1.0 nsec clock cycle
 - 4 cycles for ALU and branches
 - 5 cycles for memory
 - Frequencies
 - ALU (40%), Branch (20%), and Memory (40%)
- Clock skew and setup adds 0.2ns overhead
- Speedup with pipelining?

19

Computing Pipeline Speedup

Speedup = $\frac{\text{average instruction time unpipelined}}{\text{average instruction time pipelined}}$

$$CPI_{\text{pipelined}} = \frac{\text{Ideal CPI}}{1 + \text{Pipeline stall clock cycles per instr}}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall per instr}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}$$

Remember that average instruction time = CPI*Clock Cycle
And ideal CPI for pipelined machine is 1.

20

Pipeline Hazards

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards**: HW cannot support this combination of instructions (single person to fold and put clothes away)
 - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)
 - **Control hazards**: Pipelining of branches & other instructions that change the PC
- Common solution is to **stall** the pipeline until the hazard is resolved, inserting one or more “**bubbles**” in the pipeline

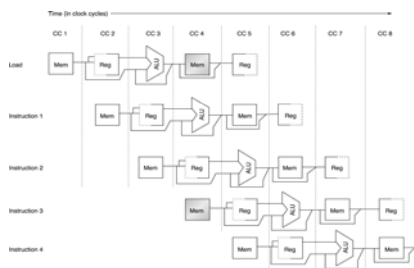
21

Structural Hazards

- Overlapped execution of instructions:
 - Pipelining of functional units
 - Duplication of resources
- Structural Hazard
 - When the pipeline can not accommodate some combination of instructions
- Consequences
 - Stall
 - Increase of CPI from its ideal value (1)

22

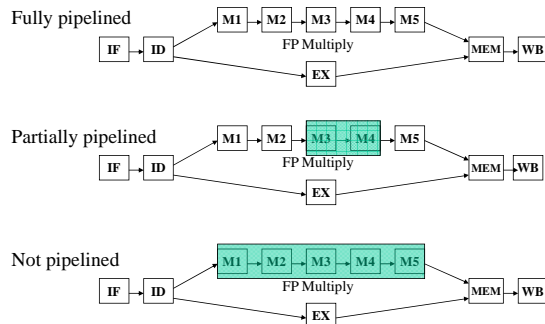
Structural Hazard with 1 port per Memory



© 2003 Elsevier Science (USA). All rights reserved.

23

Pipelining of Functional Units



24

To pipeline or Not to pipeline

- Elements to consider
 - Effects of pipelining and duplicating units
 - Increased costs
 - Higher latency (pipeline register overhead)
 - Frequency of structural hazard
- Example: unpipelined FP multiply unit in MIPS
 - Latency: 5 cycles
 - Impact on *mdjdp2* program?
 - Frequency of FP instructions: 14%
 - Depends on the distribution of FP multiplies
 - Best case: uniform distribution
 - Worst case: clustered, back-to-back multiplies

25

Example: Dual-port vs. Single-port

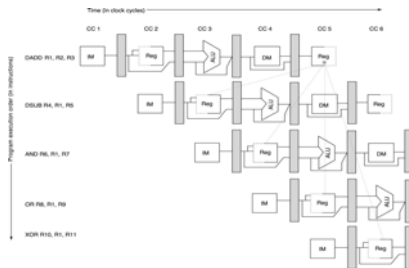
- Machine A: Dual ported memory
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

$$\begin{aligned} \text{SpeedUp}_A &= \text{Pipeline Depth} / (1 + 0) \times (\text{clock}_{\text{unpipe}} / \text{clock}_{\text{pipe}}) \\ &= \text{Pipeline Depth} \\ \text{SpeedUp}_B &= \text{Pipeline Depth} / (1 + 0.4 \times 1) \\ &\quad \times (\text{clock}_{\text{unpipe}} / (\text{clock}_{\text{unpipe}} / 1.05)) \\ &= (\text{Pipeline Depth} / 1.4) \times 1.05 \\ &= 0.75 \times \text{Pipeline Depth} \end{aligned}$$

- $\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth} / (0.75 \times \text{Pipeline Depth}) = 1.33$
- Machine A is 1.33 times faster

25

Data Hazards



27

Three Generic Data Hazards

Instr_i followed by Instr_j

- Read After Write (RAW)
 - Instr_j tries to read operand before Instr_i writes it

28

Three Generic Data Hazards (Cont'd)

Instr_i followed by Instr_j

- Write After Read (WAR)
 - Instr_j tries to write operand *before* Instr_i reads it
 - Gets wrong operand
- Can't happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Reads are always in stage 2, and
 - Writes are always in stage 5

29

Three Generic Data Hazards (Cont'd)

Instr_i followed by Instr_j

- Write After Write (WAW)
 - Instr_j tries to write operand *before* Instr_i writes it
 - Leaves wrong result (Instr_i not Instr_j)
- Can't happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

30

Examples in more complicated pipelines

- WAW - write after write

```
LW R1, 0(R2)    IF ID EX M1 M2 WB
ADD R1, R2, R3  IF ID EX M1 M2 WB
```

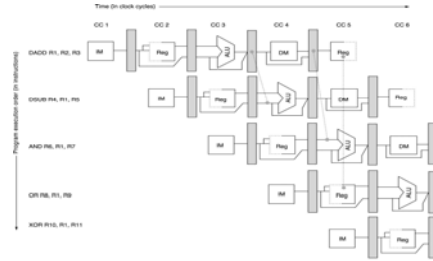
- WAR - write after read

```
SW 0(R1), R2   IF ID EX M1 M2 WB
ADD R2, R3, R4 IF ID EX M1 WB
```

This is a problem if Register writes are during The first half of the cycle And reads during the Second half

31

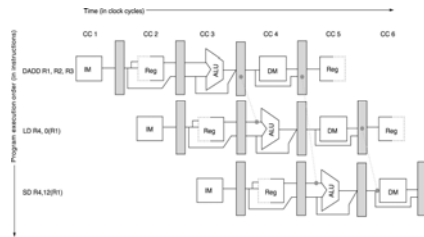
Avoiding Data Hazard with Forwarding



© 2003 Elsevier Science (USA). All rights reserved.

32

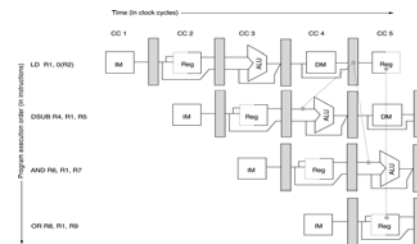
Forwarding of Operands by Stores



© 2003 Elsevier Science (USA). All rights reserved.

33

Stalls in spite of Forwarding



© 2003 Elsevier Science (USA). All rights reserved.

34

Software Scheduling to Avoid Load Hazards

Try producing fast code for

$a = b + c;$

$d = e - f;$

assuming $a, b, c, d, e,$ and f in memory.

Slow code:

LW Rb,b LW Rb,b

LW Rc,c LW Rc,c

ADD Ra,Rb,Rc LW Re,e

SW a,Ra ADD Ra,Rb,Rc

LW Re,e LW Rf,f

LW Rf,f SW a,Ra

SUB Rd,Re,Rf SUB Rd,Re,Rf

SW d,Rd SW d,Rd

Fast code:

35

Effect of Software Scheduling

LW Rb,b	IF	ID	EX	MEM	WB
LW Rc,c	IF	ID	EX	MEM	WB
ADD Ra,Rb,Rc	IF	ID	EX	MEM	WB
SW a,Ra	IF	ID	EX	MEM	WB
LW Re,e	IF	ID	EX	MEM	WB
LW Rf,f	IF	ID	EX	MEM	WB
SUB Rd,Re,Rf	IF	ID	EX	MEM	WB
SW d,Rd	IF	ID	EX	MEM	WB

LW Rb,b	IF	ID	EX	MEM	WB
LW Rc,c	IF	ID	EX	MEM	WB
LW Re,e	IF	ID	EX	MEM	WB
ADD Ra,Rb,Rc	IF	ID	EX	MEM	WB
LW Rf,f	IF	ID	EX	MEM	WB
SW a,Ra	IF	ID	EX	MEM	WB
SUB Rd,Re,Rf	IF	ID	EX	MEM	WB
SW d,Rd	IF	ID	EX	MEM	WB

36

Compiler Scheduling

- Eliminates load interlocks
- Demands more registers
- Simple scheduling
 - Basic block (sequential segment of code)
 - Good for simple pipelines
 - Percentage of loads that result in a stall
 - FP: 13%
 - Int: 25%

37

Control (Branch) Hazards

Branch	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
Branch successor+3								IF	ID
Branch successor+4									IF

- Stall the pipeline until we reach MEM
 - Easy, but expensive
 - Three cycles for every branch
- To reduce the branch delay
 - Find out branch is taken or not taken ASAP
 - Compute the branch target ASAP

38

Impact of Branch Stall on Pipeline Speedup

- If CPI = 1, 30% branch,

39

Reduction of Branch Penalties

Static, compile-time, branch prediction schemes

- 1 Stall the pipeline
 - Simple in hardware and software
- 2 Treat every branch as not taken
 - Continue execution as if branch were normal instruction
 - If branch is taken, turn the fetched instruction into a no-op
- 3 Treat every branch as taken
 - Useless in MIPS Why?
- 4 Delayed branch
 - Sequential successors (in delay slots) are executed anyway
 - No branches in the delay slots

40

Delayed Branch

#4: Delayed Branch

- Define branch to take place **AFTER** a following instruction

```

branch instruction
sequential successor1
sequential successor2
.....
sequential successorn
branch target if taken
    
```

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

41

Predict-not-taken Scheme

Untaken Branch	IF	ID	EX	MEM	WB				
Instruction i+1	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	

Taken Branch	IF	ID	EX	MEM	WB				
Instruction i+1	IF	stall	stall	stall	stall				(clear the IF/ID register)
Branch target		IF	ID	EX	MEM	WB			
Branch target+1			IF	ID	EX	MEM	WB		
Branch target+2				IF	ID	EX	MEM	WB	

Compiler organizes code so that the most frequent path is the not-taken one

42

Canceling Branch Instructions

Canceling branch includes the predicted direction

- Incorrect prediction => delay-slot instruction becomes no-op
- Helps the compiler to fill branch delay slots (no requirements for b and c)
- Behavior of a predicted-taken canceling branch

Untaken Branch	IF	ID	EX	MEM	WB
Instruction i+1	IF	stall	stall	stall	(clear the IF/ID register)
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	IF	ID	EX	MEM	WB
Instruction i+1	IF	ID	EX	MEM	WB
Branch target	IF	ID	EX	MEM	WB
Branch target i+1	IF	ID	EX	MEM	WB
Branch target i+2	IF	ID	EX	MEM	WB

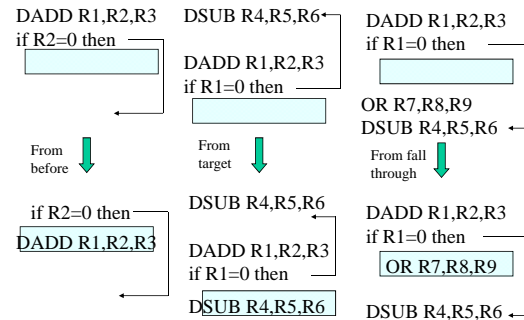
43

Delayed Branch

- Where to get instructions to fill branch delay slot?
 - Before branch instruction
 - From the target address: only valuable when branch taken
 - From fall through: only valuable when branch not taken
- Compiler effectiveness for single branch delay slot:
 - Fills about 60% of branch delay slots
 - About 80% of instructions executed in branch delay slots useful in computation
 - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)

44

Optimizations of the Branch Slot



45

Branch Slot Requirements

Strategy	Requirements	Improves performance
a) From before	Branch must not depend on delayed instruction	Always
b) From target	Must be OK to execute delayed instruction if branch is not taken	When branch is taken
c) From fall through	Must be OK to execute delayed instruction if branch is taken	When branch is not taken

Limitations in delayed-branch scheduling
 Restrictions on instructions that are scheduled
 Ability to predict branches at compile time

46

Some Working Examples

47

Branch Behavior in Programs

	Integer	FP
Forward conditional branches	13%	7%
Backward conditional branches	3%	2%
Unconditional branches	4%	1%
Branches taken	62%	70%

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Branch Penalty for predict taken = 1

Branch Penalty for predict not taken = probability of branches taken
 Slot is usefully filled (**not cancelled**) always guaranteed to be as Good or better than the other approaches.

48

Static Branch Prediction for scheduling to avoid data hazards

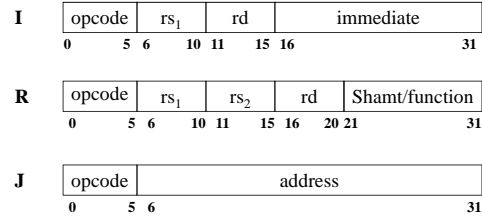
- Correct predictions
 - Reduce branch hazard penalty
 - Help the scheduling of data hazards:
- If branch is almost never taken

LW	R1, 0(R2)
SUB	R1, R1, R3
BEQZ	R1, L
OR	R4, R5, R6
ADD	R10, R4, R3
L:	ADD R7, R8, R9

If branch is almost always taken
- Prediction methods
 - Examination of program behavior (benchmarks)
 - Use of profile information from previous runs

49

How is Pipelining Implemented? MIPS Instruction Formats



Fixed-field decoding

50

1st and 2nd Instruction cycles

- Instruction fetch (IF)
 - IR ← Mem[PC];
 - NPC ← PC + 4
- Instruction decode & register fetch (ID)
 - A ← Regs[IR_{6..10}];
 - B ← Regs[IR_{11..15}];
 - Imm ← ((IR₁₆)¹⁶ # # IR_{16..31})

51

3rd Instruction cycle

- Execution & effective address (EX)
 - Memory reference
 - ALUOutput ← A + Imm
 - Register - Register ALU instruction
 - ALUOutput ← A func B
 - Register - Immediate ALU instruction
 - ALUOutput ← A op Imm
 - Branch
 - ALUOutput ← NPC + Imm; Cond ← (A op 0)

52

4th Instruction cycle

- Memory access & branch completion (MEM)
 - Memory reference
 - PC ← NPC
 - LMD ← Mem[ALUOutput] (load)
 - Mem[ALUOutput] ← B (store)
 - Branch
 - if (cond) PC ← ALUOutput; else PC ← NPC

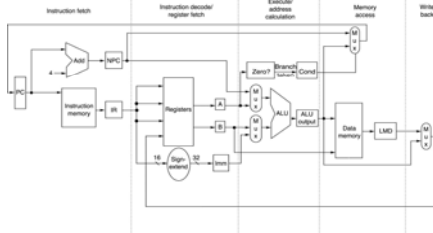
53

5th Instruction cycle

- Write-back (WB)
 - Register - register ALU instruction
 - Regs[IR_{16..20}] ← ALUOutput
 - Register - immediate ALU instruction
 - Regs[IR_{11..15}] ← ALUOutput
 - Load instruction
 - Regs[IR_{11..15}] ← LMD

54

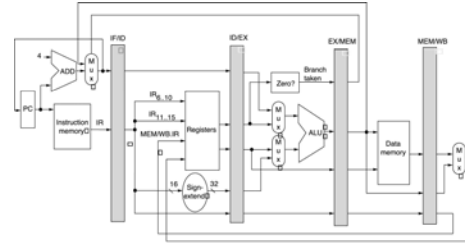
5 Stages of MIPS Datapath



© 2003 Elsevier Science (USA). All rights reserved.

55

5 Stages of MIPS Datapath with Registers



© 2003 Elsevier Science (USA). All rights reserved.

56

Events on Every Pipe Stage

Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC ← (IF/EX.MEM.opcode == branch) & EX/MEM.cond) [EX/MEM.ALLOutput] else [PC+4];		
ID	ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend[IF/ID.IR[immediate field]];		
ALU instruction	Load or store instruction	Branch instruction	
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALLOutput ← ID/EX.A Junc ID/EX.B; or EX/MEM.ALLOutput ← ID/EX.A op ID/EX.Imm;	EX/MEM.IR ← ID/EX.IR EX/MEM.ALLOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B;	EX/MEM.ALLOutput ← ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALLOutput ← EX/MEM.ALLOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALLOutput]; or MEM[EX/MEM.ALLOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rt]] ← MEM/WB.ALLOutput; or Regs[MEM/WB.IR[rt]] ← MEM/WB.ALLOutput;	For load op: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

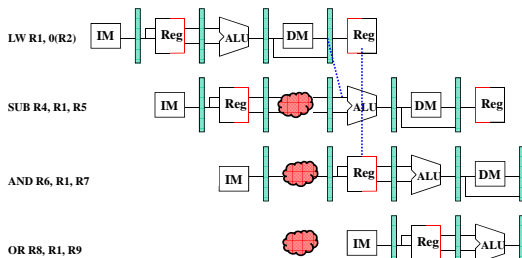
7

Implementing the Control for the MIPS Pipeline

- LD R1, 45 (R2)
DADD R5, R6, R7
DSUB R8, R6, R7
OR R9, R6, R7
- LD R1, 45 (R2)
DADD R5, R1, R7
DSUB R8, R6, R7
OR R9, R6, R7
- LD R1, 45 (R2)
DADD R5, R6, R7
DSUB R8, R1, R7
OR R9, R6, R7
- LD R1, 45 (R2)
DADD R5, R6, R7
DSUB R8, R6, R7
OR R9, R1, R7

58

Pipeline Interlocks



	IF	ID	EX	MEM	WB
LW R1, 0(R2)					
SUB R4, R1, R5	IF	ID	stall	EX	MEM
AND R6, R1, R7	IF	stall	ID	EX	MEM
OR R8, R1, R9	IF	ID	EX	MEM	WB

59

Load Interlock Implementation

- RAW load interlock detection during ID
 - Load instruction in EX
 - Instruction that needs the load data in ID
- Logic to detect load interlock

ID/EX.IR _{0..5}	IF/ID.IR _{0..5}	Comparison
Load	r-r ALU	ID/EX.IR _[RT] == IF/ID.IR _[RS]
Load	r-r ALU	ID/EX.IR _[RT] == IF/ID.IR _[RT]
Load	Load, Store, r-i ALU, branch	ID/EX.IR _[RT] == IF/ID.IR _[RS]

- Action (insert the pipeline stall)
 - ID/EX.IR_{0..5} = 0 (no-op)
 - Re-circulate contents of IF/ID

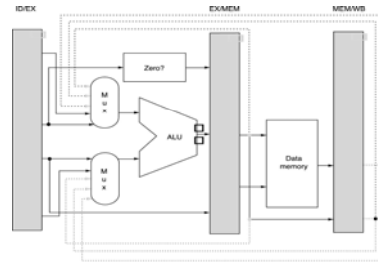
60

Forwarding Implementation

- Source: ALU or MEM output
- Destination: ALU, MEM or Zero? input(s)
- Compare (forwarding to ALU input):
- Important
 - Please refer to Fig. A.22 in slide #63

61

Forwarding Implementation (Cont'd)



© 2003 Elsevier Science (USA). All rights reserved.

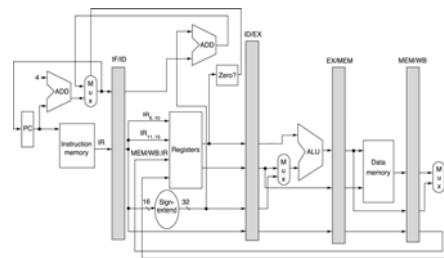
62

Forwarding Implementation - All Possible Forwarding

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.BR[rs] == ID/EX.BR[rs]
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.BR[rt] == ID/EX.BR[rt]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.BR[rs] == ID/EX.BR[rs]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.BR[rt] == ID/EX.BR[rt]
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.BR[rs] == ID/EX.BR[rs]
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.BR[rt] == ID/EX.BR[rt]
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.BR[rs] == ID/EX.BR[rs]
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.BR[rt] == ID/EX.BR[rt]
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load	Top ALU input	MEM/WB.BR[rs] == ID/EX.BR[rs]

63

Handling Branch Hazards



© 2003 Elsevier Science (USA). All rights reserved.

64

Revised Pipeline Structure

Pipe stage	Branch instruction
IF	$IF/ID.IR \leftarrow Mem[PC];$ $IF/ID.NPC.PC \leftarrow (IF/ID.opcode == branch) \& (Regs[IF/ID.IR_{6..11}] op 0) ? (IF/ID.NPC + (IF/ID.IR_{21..31} \# IF/ID.IR_{16..19} \# 000) \& 000) : else (PC+4);$
ID	$ID/EX.A \leftarrow Regs[IF/ID.IR_{6..10}]; ID/EX.B \leftarrow Regs[IF/ID.IR_{11..15}];$ $ID/EX.IR \leftarrow IF/ID.IR;$ $ID/EX.Imm \leftarrow (IF/ID.IR_{21..31} \# IF/ID.IR_{16..19})$
EX	
MEM	
WB	
--	

65

Exceptions

- I/O device request
- Operating system call
- Tracing instruction execution
- Breakpoint
- Integer overflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory access
- Memory protection violation
- Undefined instruction
- Hardware malfunctions
- Power failure

66

Exception Categories

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. nonmaskable
- Within vs. between instructions
- Resume vs. terminate
- Most difficult
 - Occur in the middle of the instruction
 - Must be able to restart
 - Requires intervention of another program (OS)

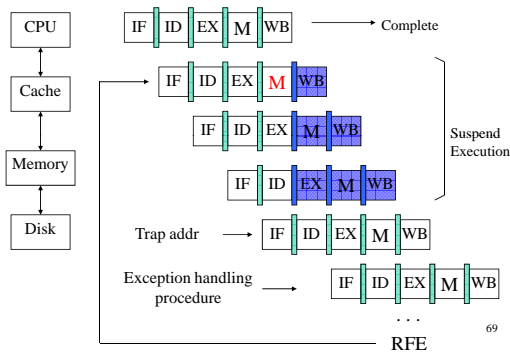
67

Overview of Exceptions

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
IO device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

68

Exception Handling



69

Stopping and Restarting Execution

- TRAP, RFE(return-from-exception) instructions
- IAR register saves the PC of faulting instruction
- Safely save the **state** of the pipeline
 - Force a TRAP on the next IF
 - Until the TRAP is taken, turn off all writes for the faulting instruction and the following ones.
 - Exception-handling routine saves the PC of the faulting instruction
- For delayed branches we need to save more PCs
- *Precise Exceptions*

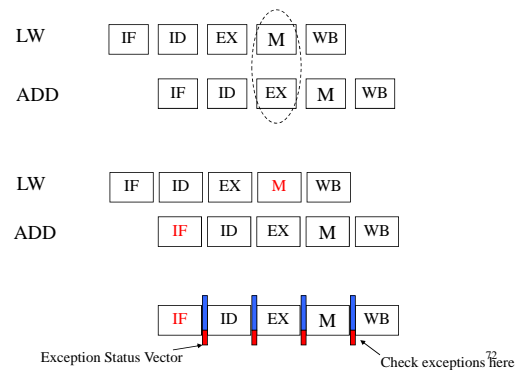
70

Exceptions in MIPS

Pipeline Stage	Exceptions
IF	Page fault, misaligned memory access, memory-protection violation
ID	Undefined opcode
EX	Arithmetic exception
MEM	Page fault, misaligned memory access, memory-protection violation
WB	None

71

Exception Handling in MIPS



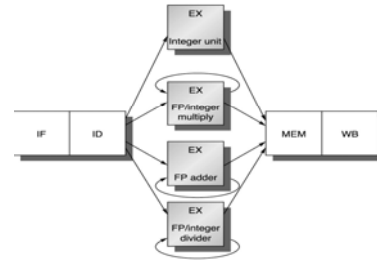
72

ISA and Exceptions

- Instructions before complete, instructions after do not, exceptions handled in order → **Precise Exceptions**
- **Precise exceptions** are simple in MIPS
 - Only one result per instruction
 - Result is written at the end of execution
- **Problems**
 - Instructions change machine state in the middle of the execution
 - Autoincrement addressing modes
 - Multicycle operations
- Many machines have two modes
 - Imprecise (efficient)
 - Precise (relatively inefficient)

73

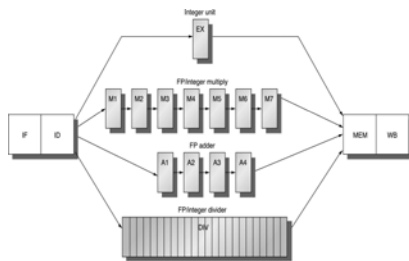
Handling Multicycle Operations



© 2003 Elsevier Science (USA). All rights reserved.

74

Handling Multicycle Operations (Cont'd)



© 2003 Elsevier Science (USA). All rights reserved.

75

Latencies and Initiation Intervals

Functional Unit	Latency	Initiation Interval
Integer ALU	0	1
Data Memory	1	1
FP adder	3	1
FP/int multiply	6	1
FP/int divider	24	25

MULTD	IF	ID	M1	M2	M3	M4	M5	M6	M7	Mem	WB
ADD	IF	ID	A1	A2	A3	A4	Mem	WB			
LD	IF	ID	EX	Mem	WB						
SD	IF	ID	EX	Mem	WB						

76

Hazards in FP pipelines

- Structural hazards in DIV unit
- Structural hazards in WB
- WAW hazards are possible (WAR not possible)
- Out-of-order completion
 - → Exception handling issues
- More frequent RAW hazards
 - ← Longer pipelines

LD F4, 0(R2) IF ID EX Mem WB

MULTD F0, F4, F6 IF ID stall M1 M2 M3 M4 M5 M6 M7 Mem WB

ADD F2, F0, F8 IF stall ID stall stall stall stall stall stall A1 A2 A3 A4 Mem WB

77

Hazards in FP pipelines (Cont'd)

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADD.D F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
L.D F2, 0(R2)							IF	ID	EX	MEM	WB

78

Hazard Detection Logic at ID

- Check for Structural Hazards
 - Divide unit/make sure register write port is available when needed
- Check for RAW hazard
 - Check source registers against destination registers in pipeline latches of instructions that are ahead in the pipeline. Similar to I-pipeline
- Check for WAW hazard
 - Determine if any instruction in A1-A4, M1-M7 has same register destination as this instruction.

79

Handling Hazards – Working Examples

80