ECE 486/586
Computer Architecture

Prof. Mark G. Faust

Maseeh College of Engineering
and Computer Science

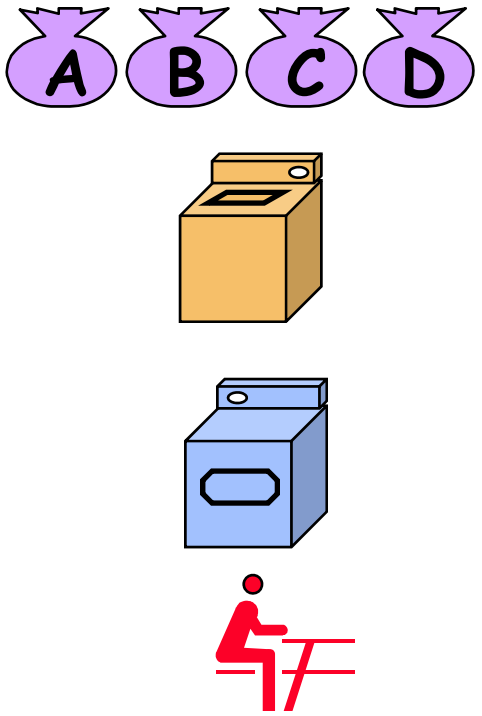
PORTLAND STATE
UNIVERSITY

Pipelining (Basics/Review)

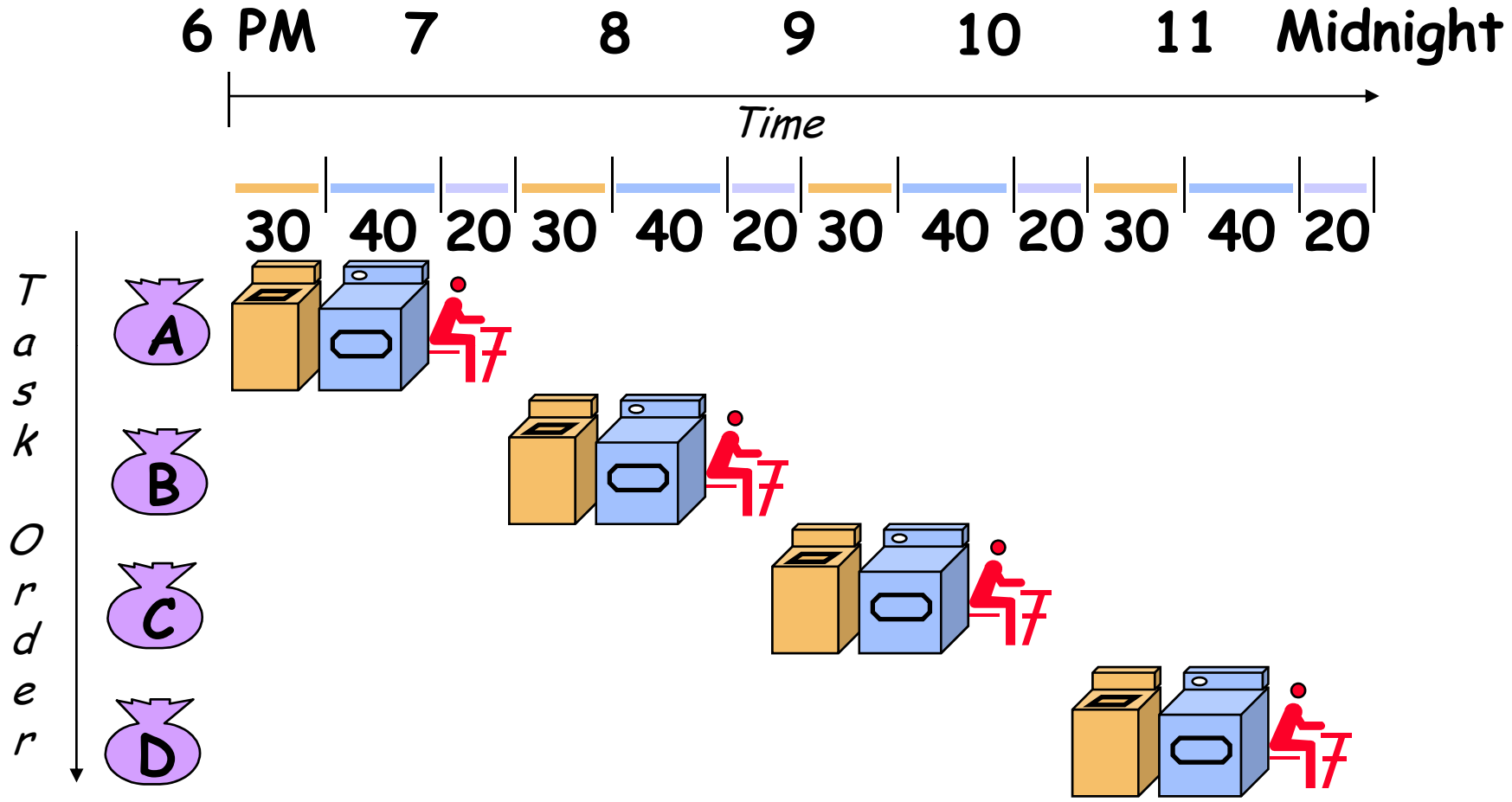
- Reading:
  - Hennessy & Patterson: Appendix A
- Homework:

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

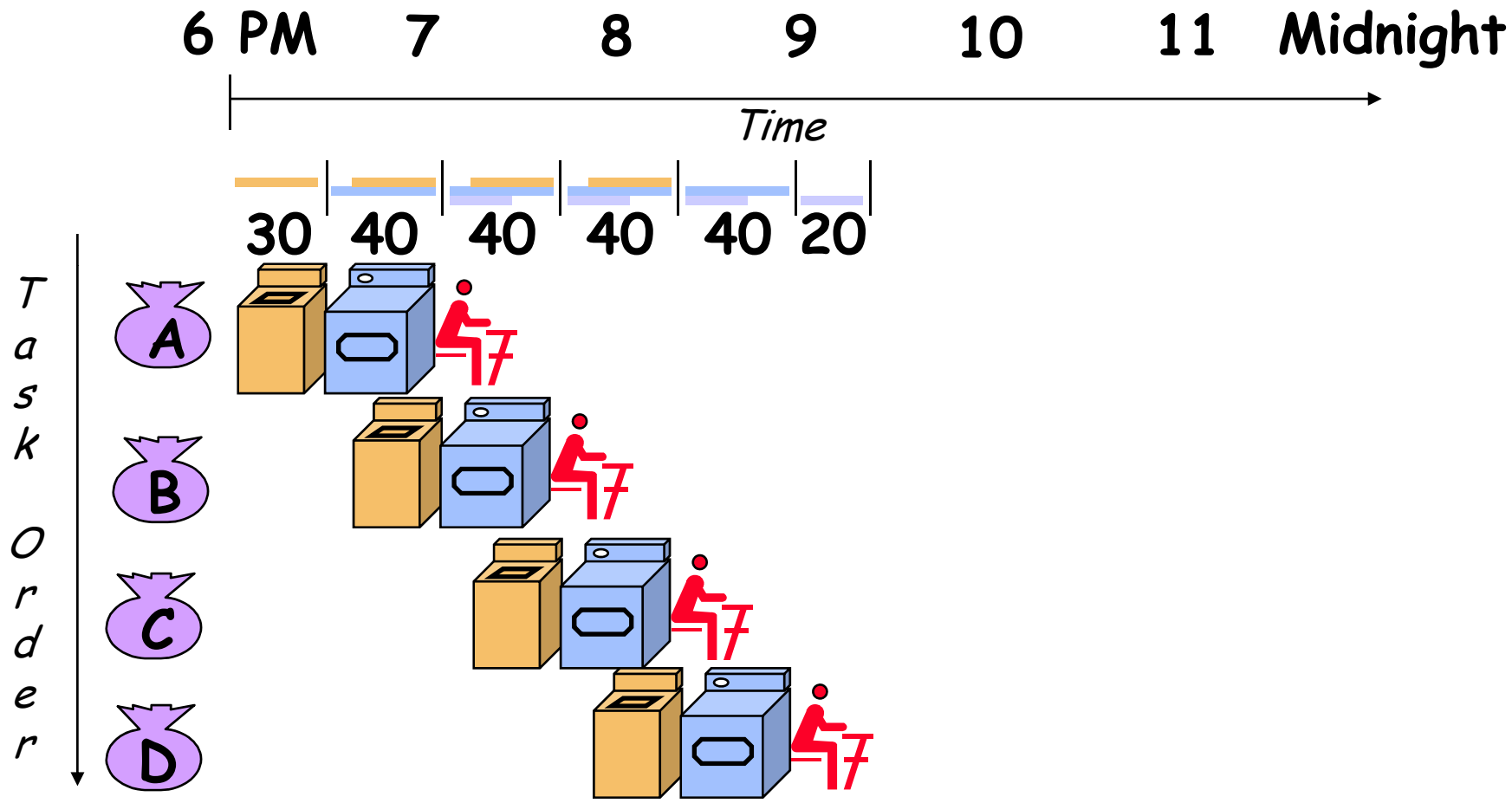


Sequential Laundry



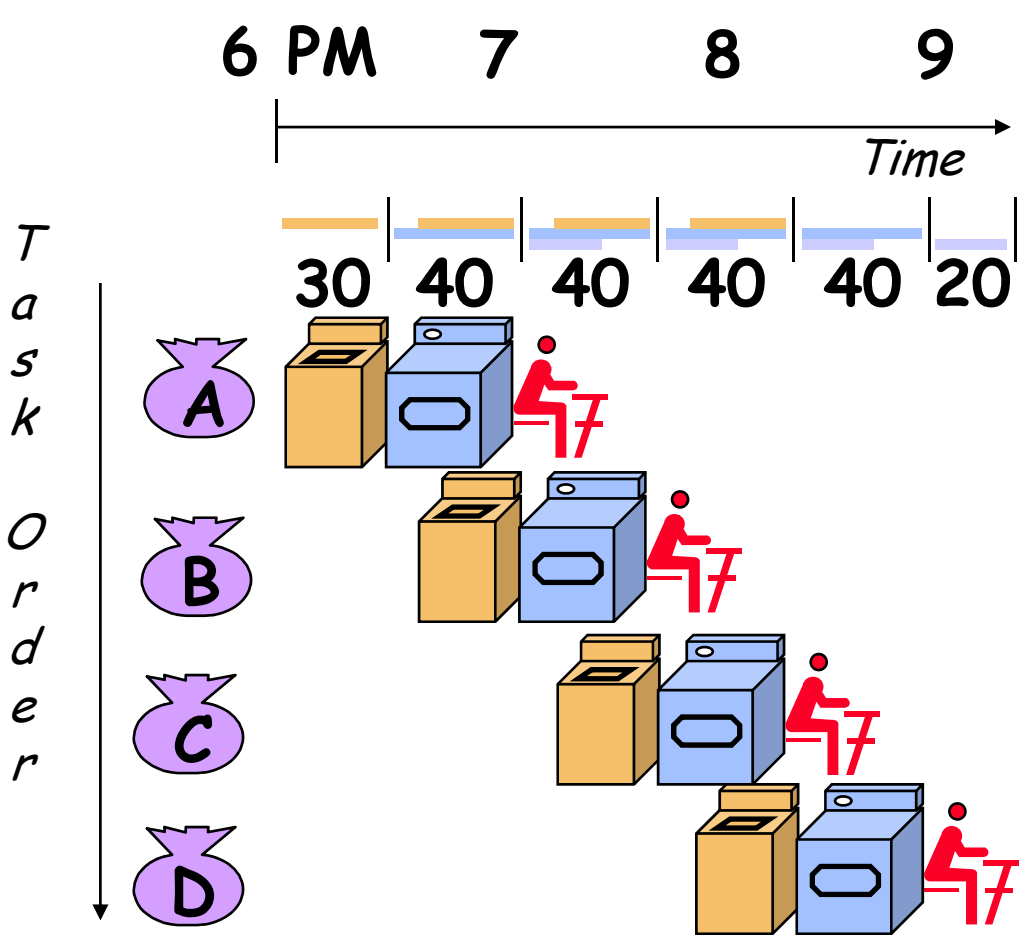
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry
Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

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
- In this pipeline the laundry operates concurrently on multiple peoples' laundry but a person's laundry is completed sequentially

Pipelining

- Exploits parallelism in sequential instruction stream
  - Resources (e.g. ALU, memory, register file) can be used concurrently by different instructions during their execution

Computer Pipelines

- Execute billions of instructions, so *throughput* is what matters
- What is desirable in instruction sets for pipelining?
  - Variable length instructions vs. all instructions same length?
  - Memory operands part of any operation vs. memory operands only in loads or stores?
  - Register operand many places in instruction format vs. registers located in same place?

A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- Memory access only via load/store instructions
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction; registers in same place
- Single address mode for load/store:
  base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Basics of a RISC Instruction Set

- All operations on data apply to data in registers
  - Typically change entire register (32- or 64-bits)
  - Contrast with IA-32 (e.g. AL, AX, EAX registers)
- Only load and store operations affect memory
  - Load/stores may operate on less than full register
  - Contrast with VAX (memory operands)
- Few instruction formats
  - Contrast with IA-32, VAX
- All instructions same size
  - Contrast with IA-32, VAX

Basics of a RISC Instruction Set

- Three classes of instructions
  - ALU
    - Operations: ADD, SUB, AND, OR
    - Sources: Register/Register or Register/Immediate
  - Load/Store
    - Sources: Register (base register) and Immediate (offset)/Register
      - Effective Address = Base Register + Offset (sign extended)
    - Load: second register is destination for data
    - Store: second register is source of data
  - Branches and Jumps
    - Branches: conditional control transfers
      - Condition codes, register comparisons
      - MIPS: Register comparisons
      - Source: Immediate (offset)
      - Target = PC + Offset (sign extended)
    - Jumps: unconditional control transfers
      - in MIPS: includes procedure calls

A Simple Implementation of a MIPS Instruction Set

- Every MIPS instruction is implemented in at most 5 clock cycles
  - Instruction Fetch (IF)
  - Instruction Decode/Register Fetch (ID)
  - Execution/Effective Address (EX)
  - Memory Access (MEM)
  - Write Back (WB)

5 Cycles

- Instruction Fetch (IF)
  - Send PC to memory
  - Fetch current instruction
  - Place in instruction register for decode
    - IR ← MEM[PC]
  - Update PC to next instruction
    - NPC ← PC + 4
- Instruction Decode (ID)
  - Decode instruction
  - Read registers (using register source specifiers) from register file
    - A ← Regs[IR[Rs]]
    - B ← Regs[IR[Rt]]
  - Sign extend immediate field (offset) of the instruction (in case needed)
    - Imm ← sign-extend(IR[Immediate Field])

Implementation: not visible ISA

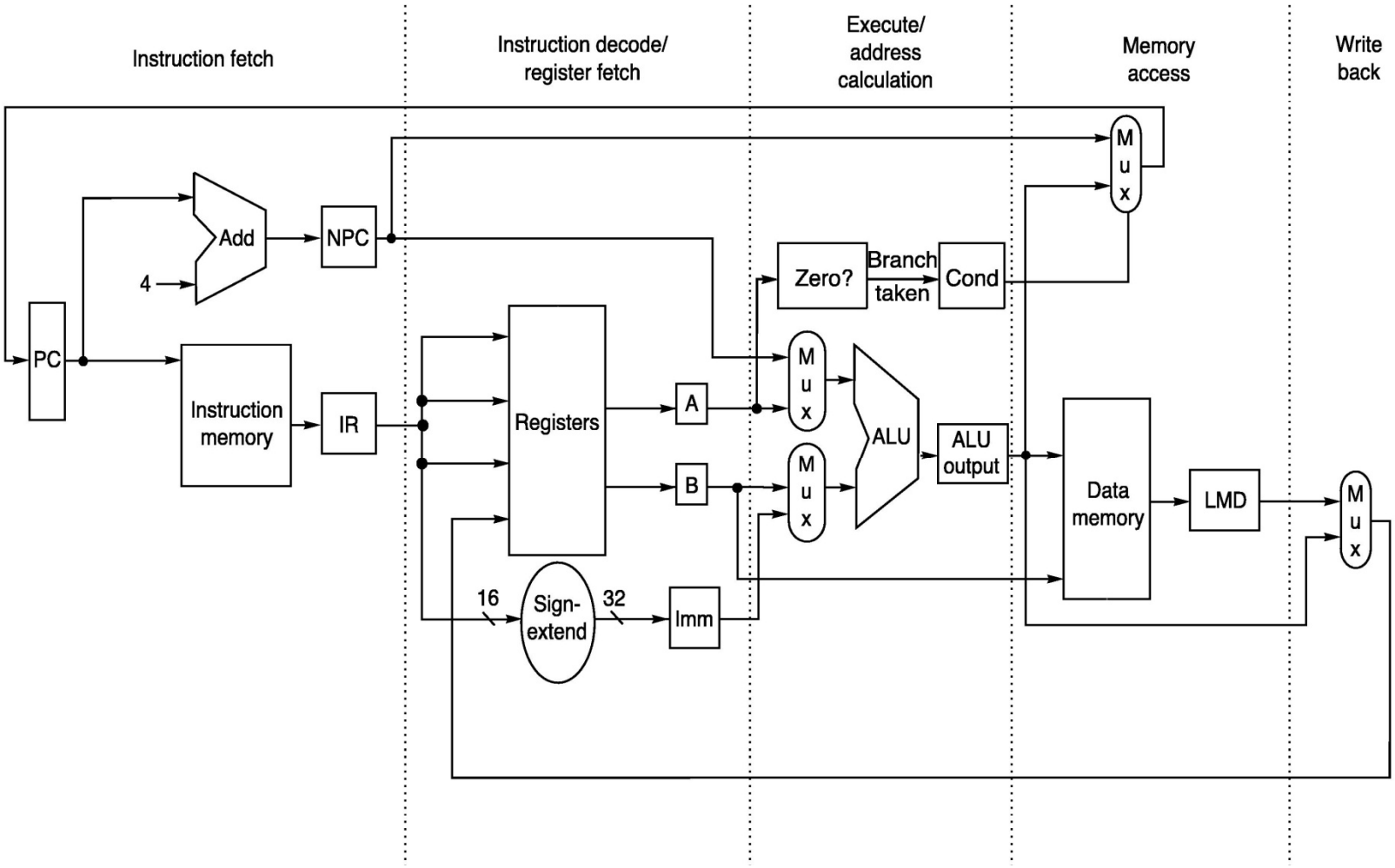
5 Cycles

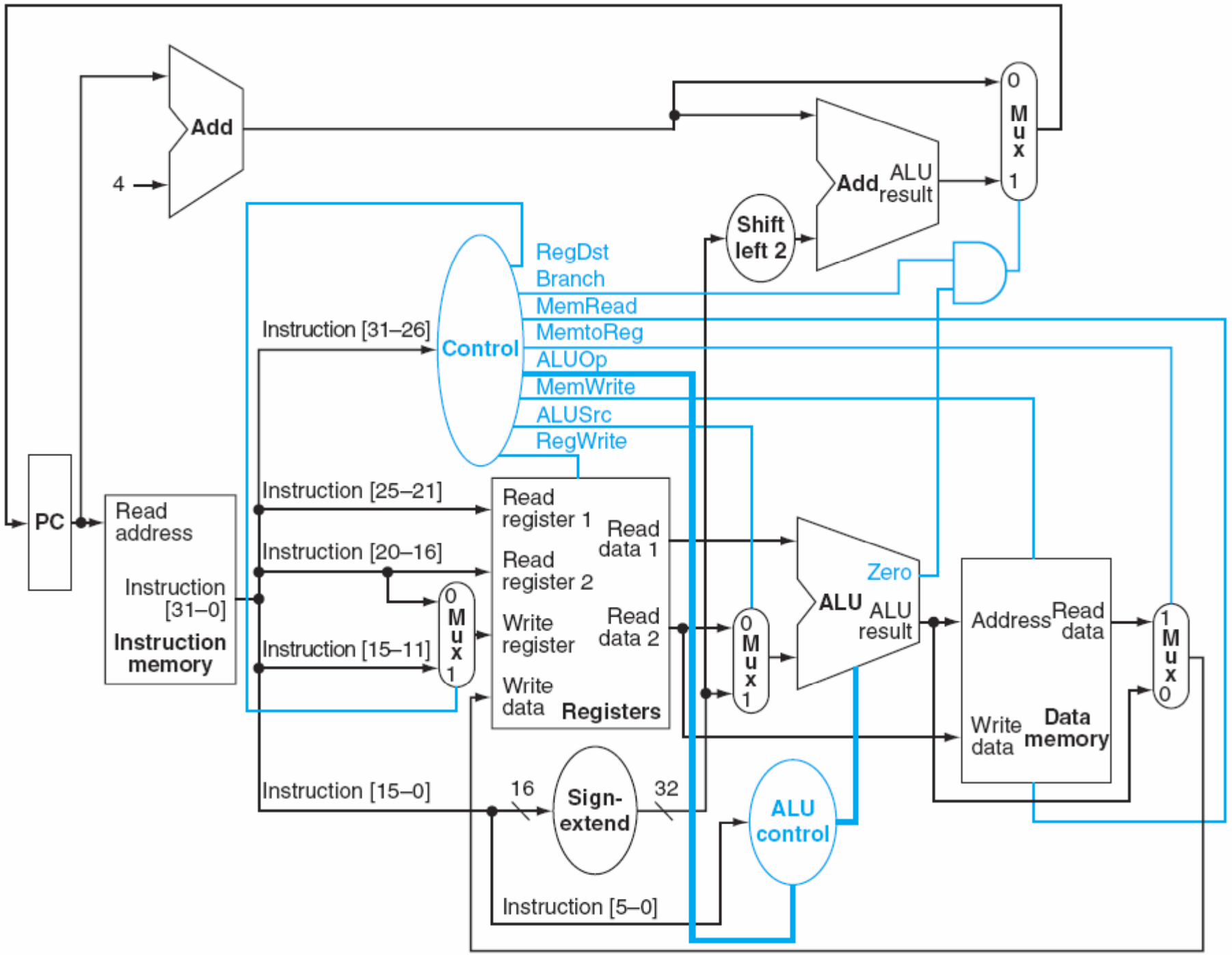
- Execution/Effective Address (EX)
  - ALU operates on operands prepared in previous cycle
  - One of four functions depending upon opcode
    - Memory Reference
      - Form effective address from base register and immediate offset
      - ALUOutput ← A + Imm
    - Register-Register ALU instruction
      - ALUOutput ← A *func* B
    - Register-Immediate ALU instruction
      - ALUOutput ← A *func* Imm
    - Branch
      - Compute branch target by creating byte offset from signed extended immediate field and adding to PC
      - ALUOutput ← NPC + (Imm << 2)
      - Check to see if branch taken (BEQZ uses BEQ R0)
      - Cond ← (A == 0)

5 Cycles

- Memory Access/Branch Completion (MEM)
  - Update PC for all instructions
    - PC ← NPC
  - Memory Reference
    - Load: data from memory placed in LMD register
      - LMD ← Mem[ALUOutput]
    - Store: data written from B register to memory
      - Mem[ALUOutput] ← B
  - Branch
    - If condition satisfied, PC is replaced with branch target address
      - If (Cond) PC ← ALUOutput
- Write Back (WB)
  - Register-Register ALU instruction
    - Regs[rd] ← ALUOutput
  - Register-Immediate ALU instruction
    - Regs[rt] ← ALUOutput
  - Load instruction
    - Regs[rt] ← LMD

MIPS Data Path Implementation




Add
4
0
M
u
x
1
Add
ALU result
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign-extend
32
ALU control
Zero
ALU
ALU result
Address
Read data
Write data
Data memory

MIPS Control Logic

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

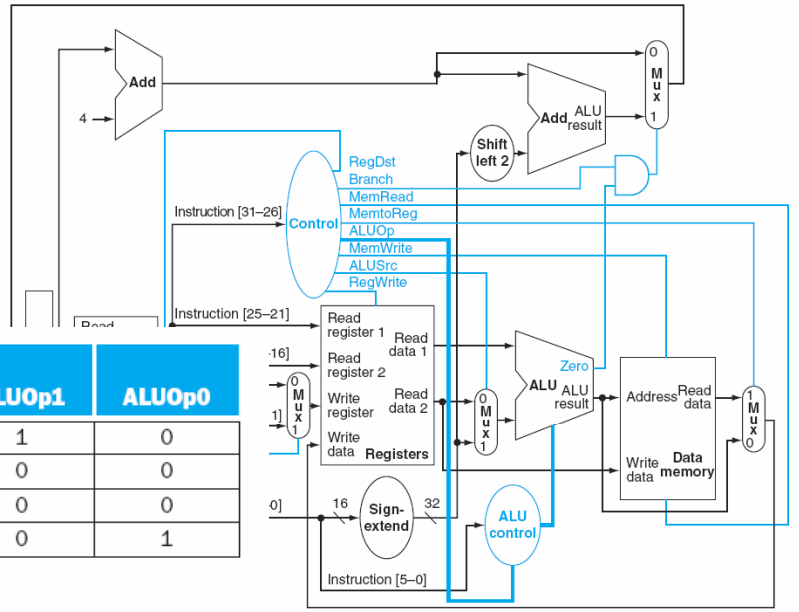
b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

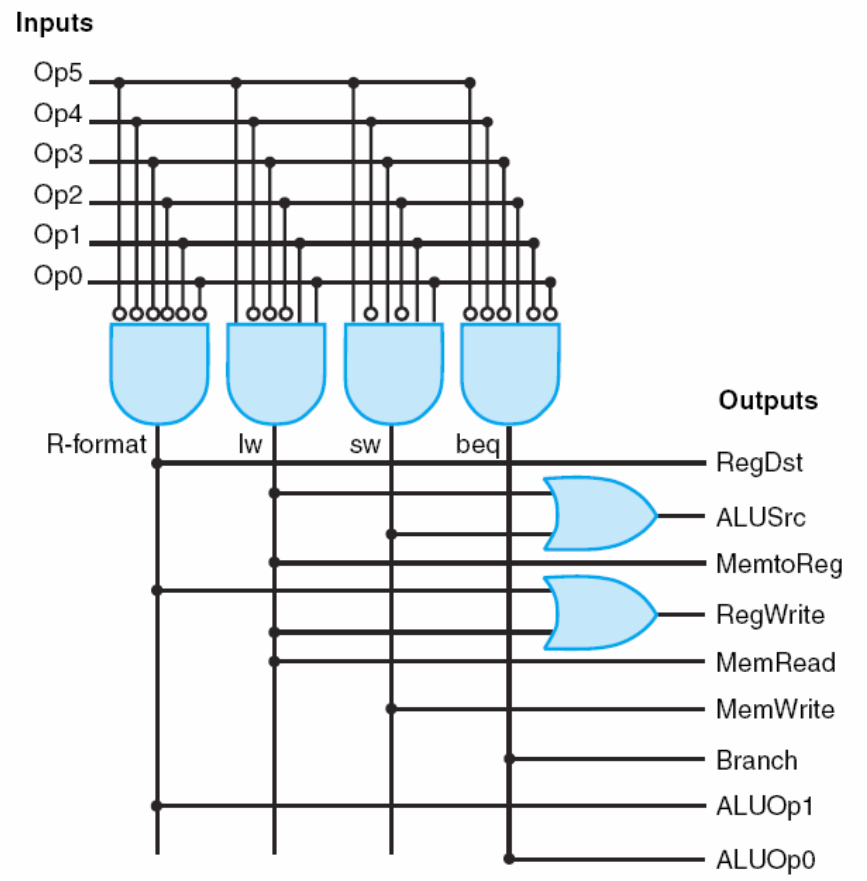
Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

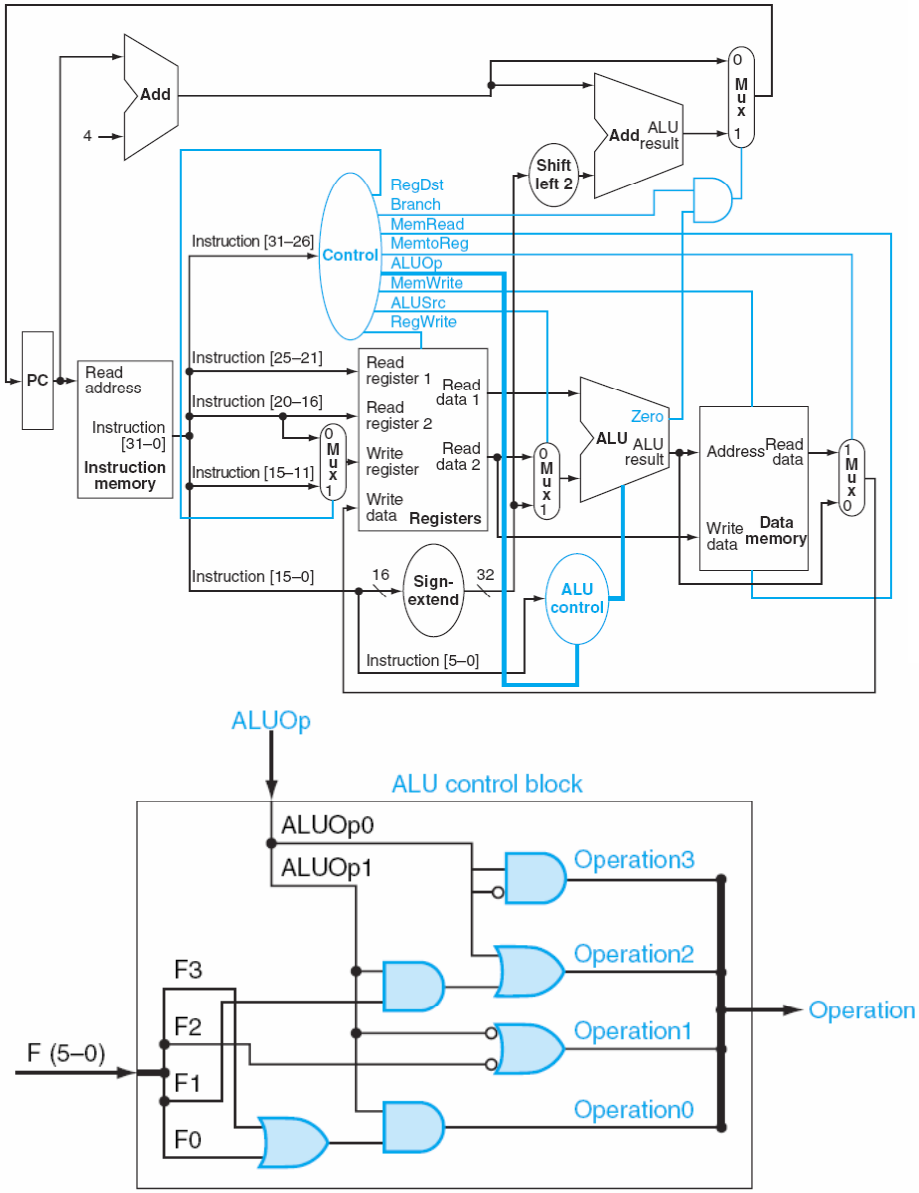
Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

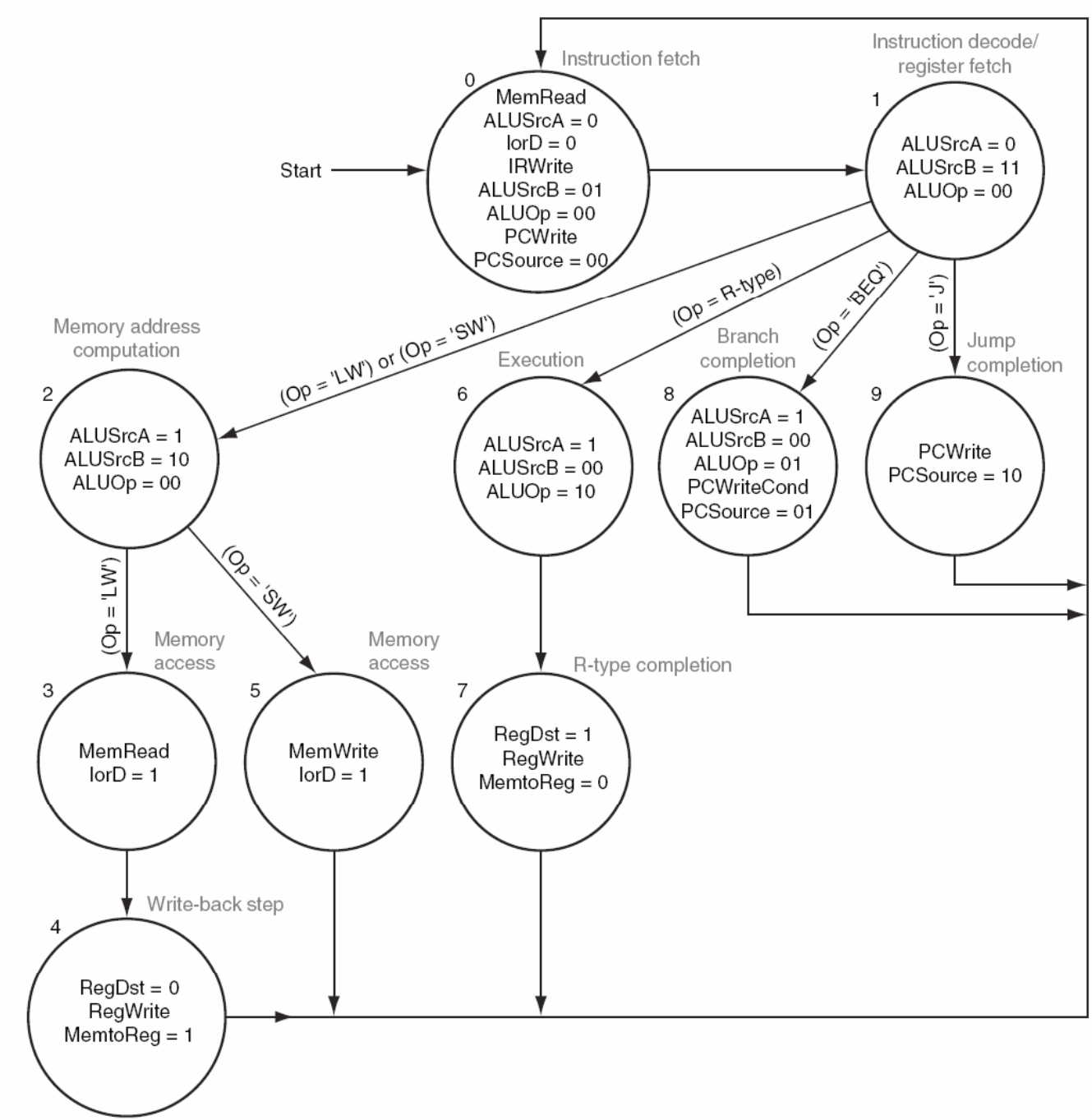


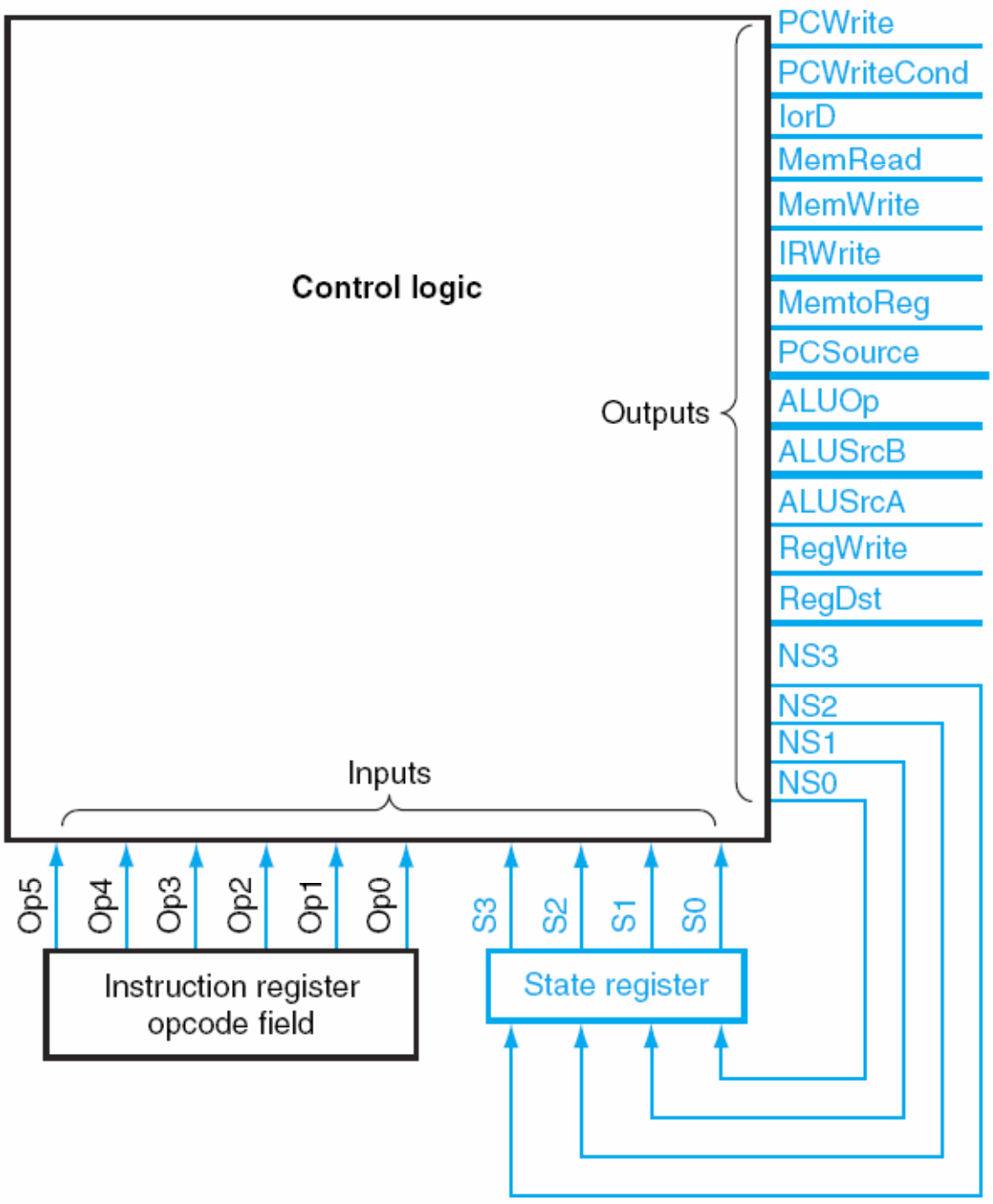
Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

MIPS Control Logic



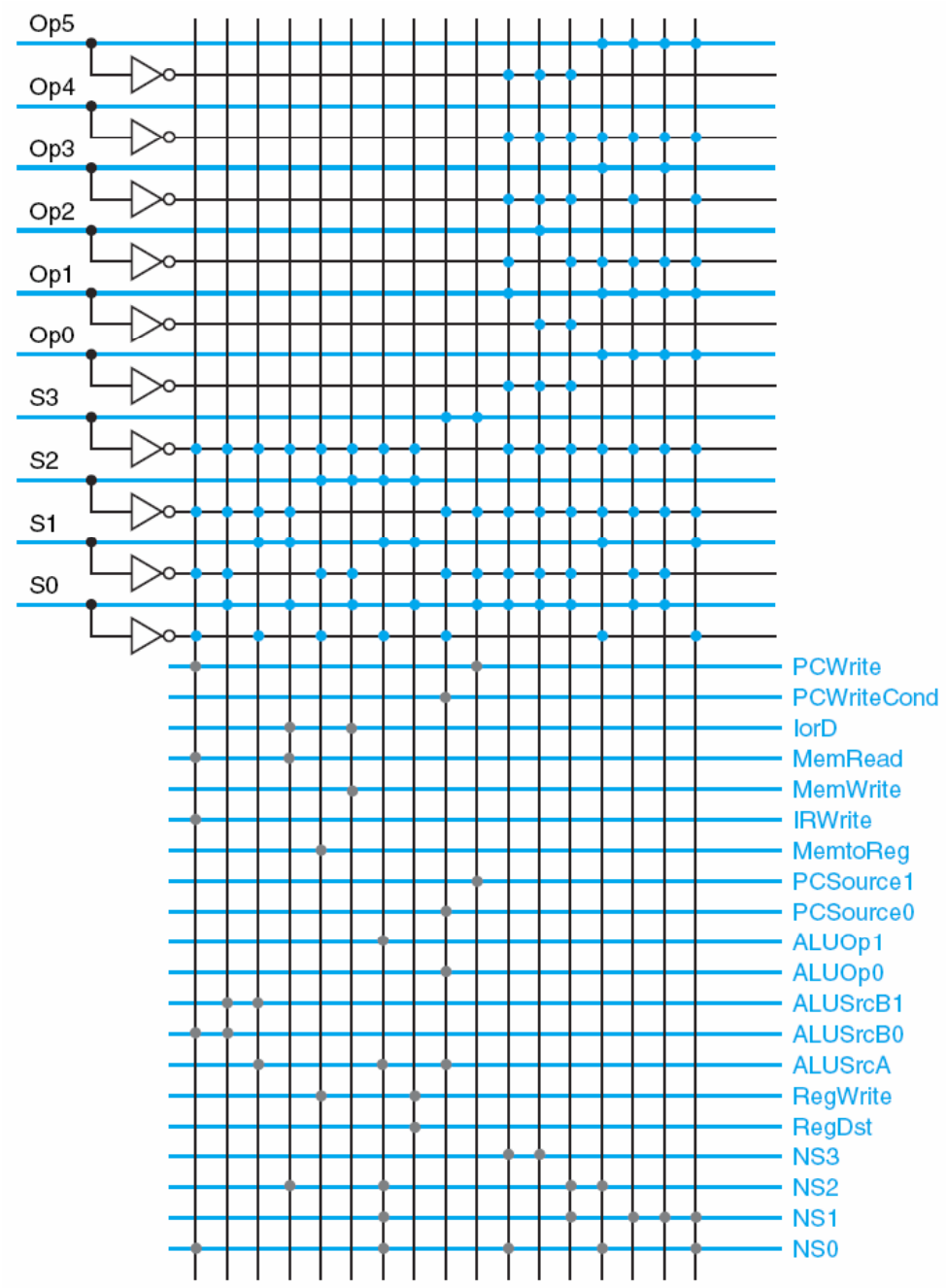



Start
Instruction fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Instruction decode/
register fetch
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
3
MemRead
IorD = 1
Memory
access
5
MemWrite
IorD = 1
R-type completion
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst = 0
RegWrite
MemtoReg = 1

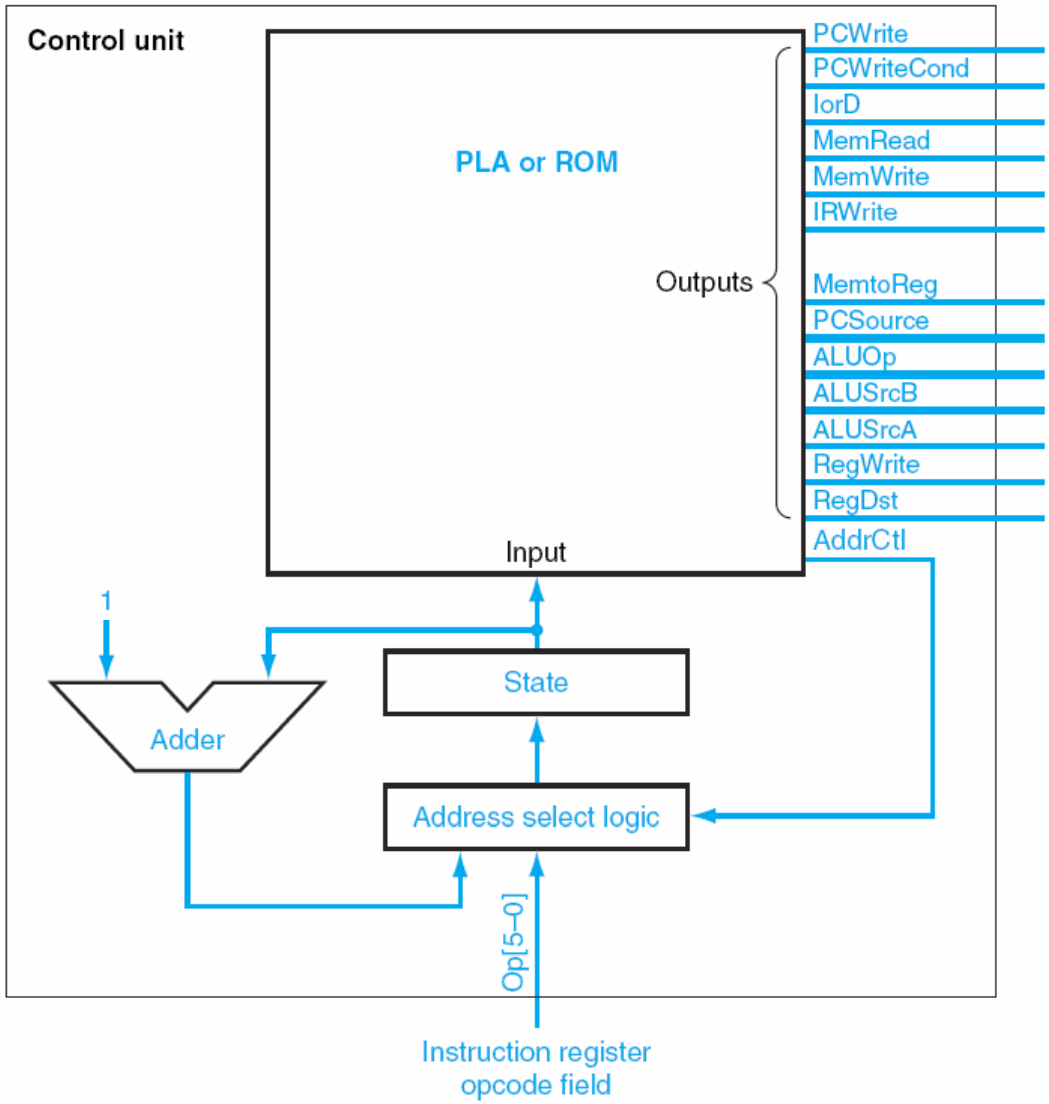

Control logic
Outputs
PCWrite
PCWriteCond
IorD
MemRead
MemWrite
IRWrite
MemtoReg
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
NS3
NS2
NS1
NS0
Inputs
Op5
Op4
Op3
Op2
Op1
Op0
S3
S2
S1
S0
Instruction register
opcode field
State register

FSM Outputs (and NextState) Table

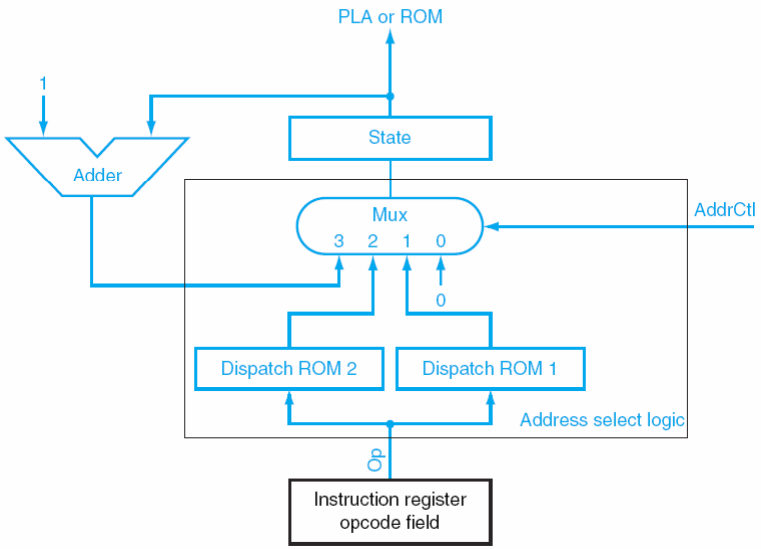
Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')


Op5
Op4
Op3
Op2
Op1
Op0
S3
S2
S1
S0
PCWrite
PCWriteCond
IorD
MemRead
MemWrite
IRWrite
MemtoReg
PCSource1
PCSource0
ALUOp1
ALUOp0
ALUSrcB1
ALUSrcB0
ALUSrcA
RegWrite
RegDst
NS3
NS2
NS1
NS0

Using Counter for Next State



AddrCtl value	Action
0	Set state to 0
1	Dispatch with ROM 1
2	Dispatch with ROM 2
3	Use the incremented state



Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

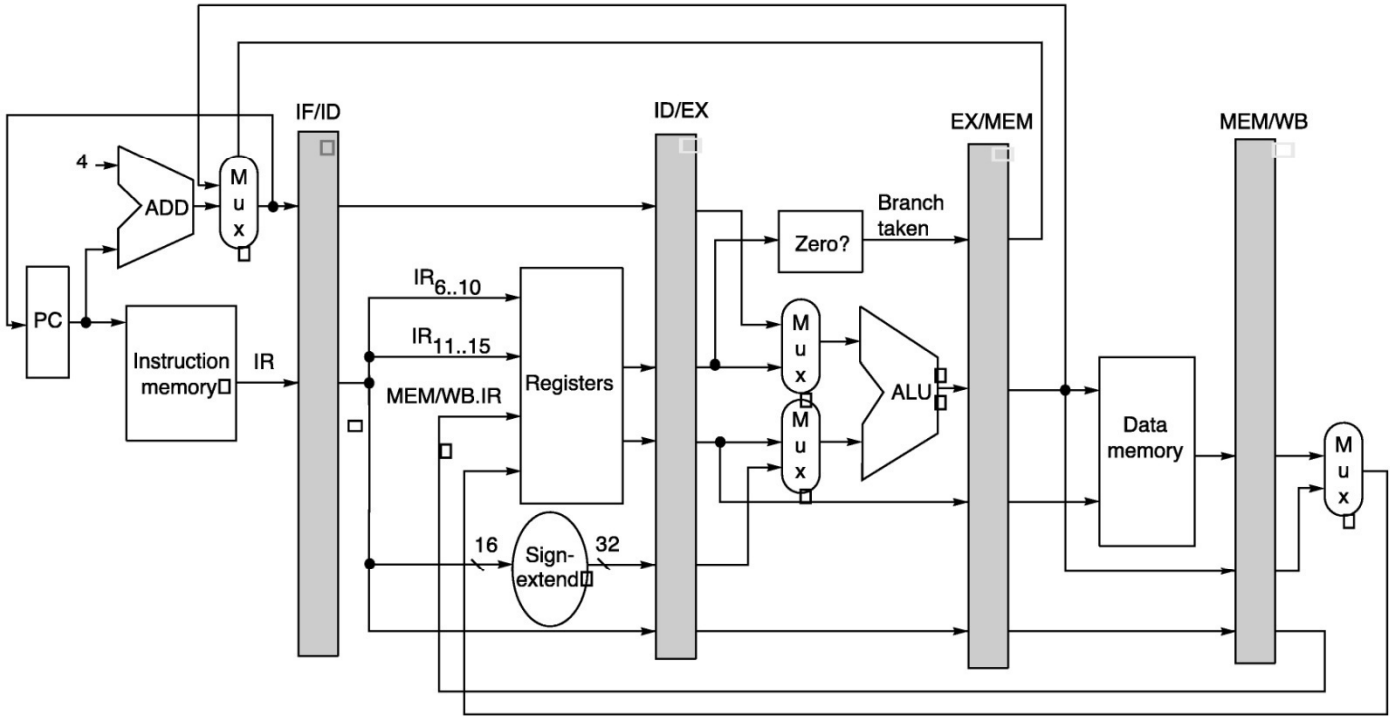
Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	0011
101011	sw	0101

FIGURE D.4.3 The dispatch ROMs each have 2^6 = 64 entries that are 4 bits wide, since that is the number of bits in the state encoding. This figure only shows the entries in the ROM that are of interest for this subset. The first column in each table indicates the value of Op, which is the address used to access the dispatch ROM. The second column shows the symbolic name of the opcode. The third column indicates the value at that address in the ROM.

State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

AddrCtl value	Action
0	Set state to 0
1	Dispatch with ROM 1
2	Dispatch with ROM 2
3	Use the incremented state

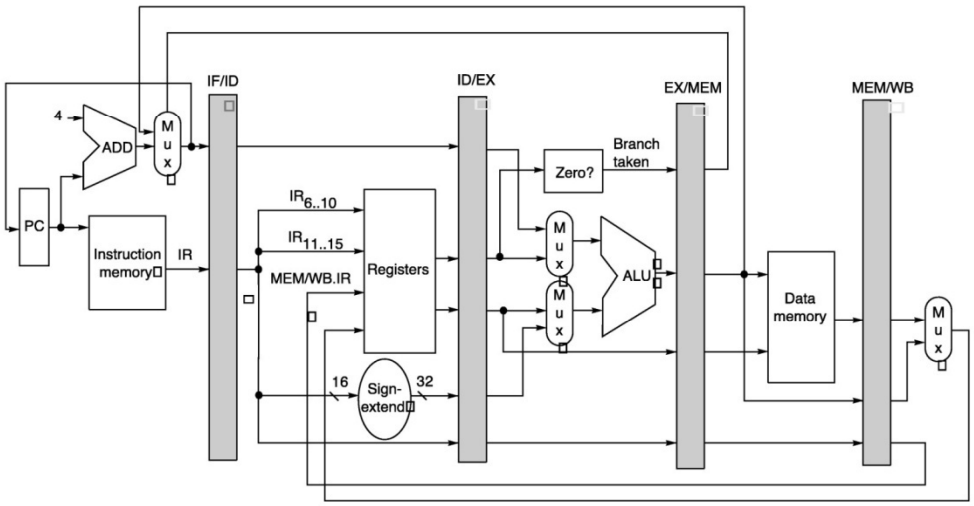
Classic 5 Stage MIPS Pipeline

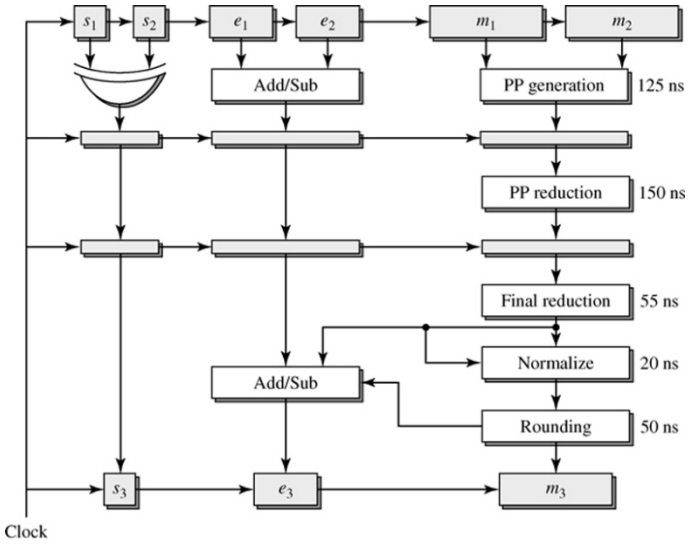


- Addition of set of registers between each pipeline stage
  - Store intermediate results
  - Store control information: decoded instruction fields
    - Data path must be controlled by instruction in that phase (not the instruction currently in ID phase!)

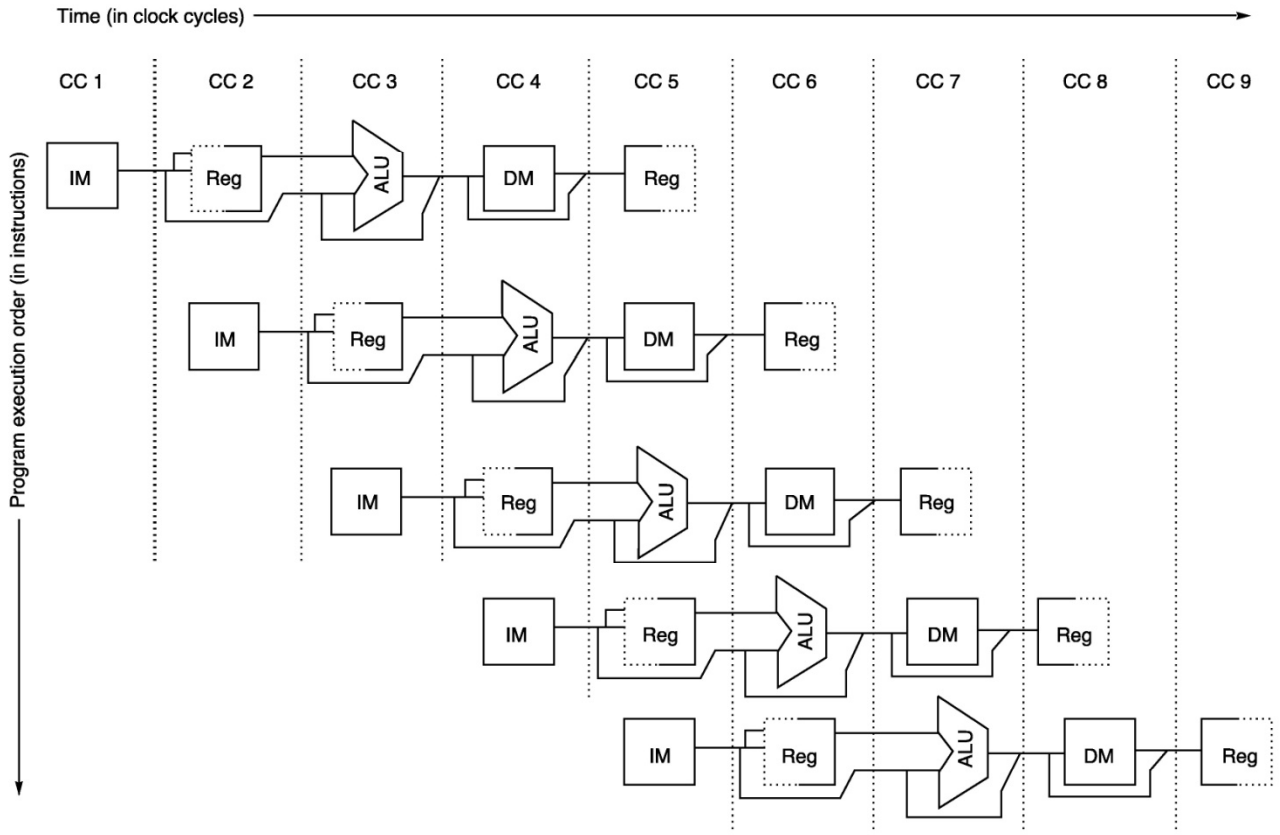
Ideal Pipeline

- Uniform sub-computations
  - Stages are of equal combinational delay
  - No additional delay introduced by pipelining
    - Setup time for pipeline registers must be added to combinational delay
- Identical Computations
  - Many repetitions of same computation to be performed
    - Ignore pipeline fill time
  - Each repetition requires same sequence of sub-computations
    - All stages always used
- Independent Computations
  - All sub-computations present in the pipeline are independent
    - Later sub-computation not awaiting completion of earlier sub-computations



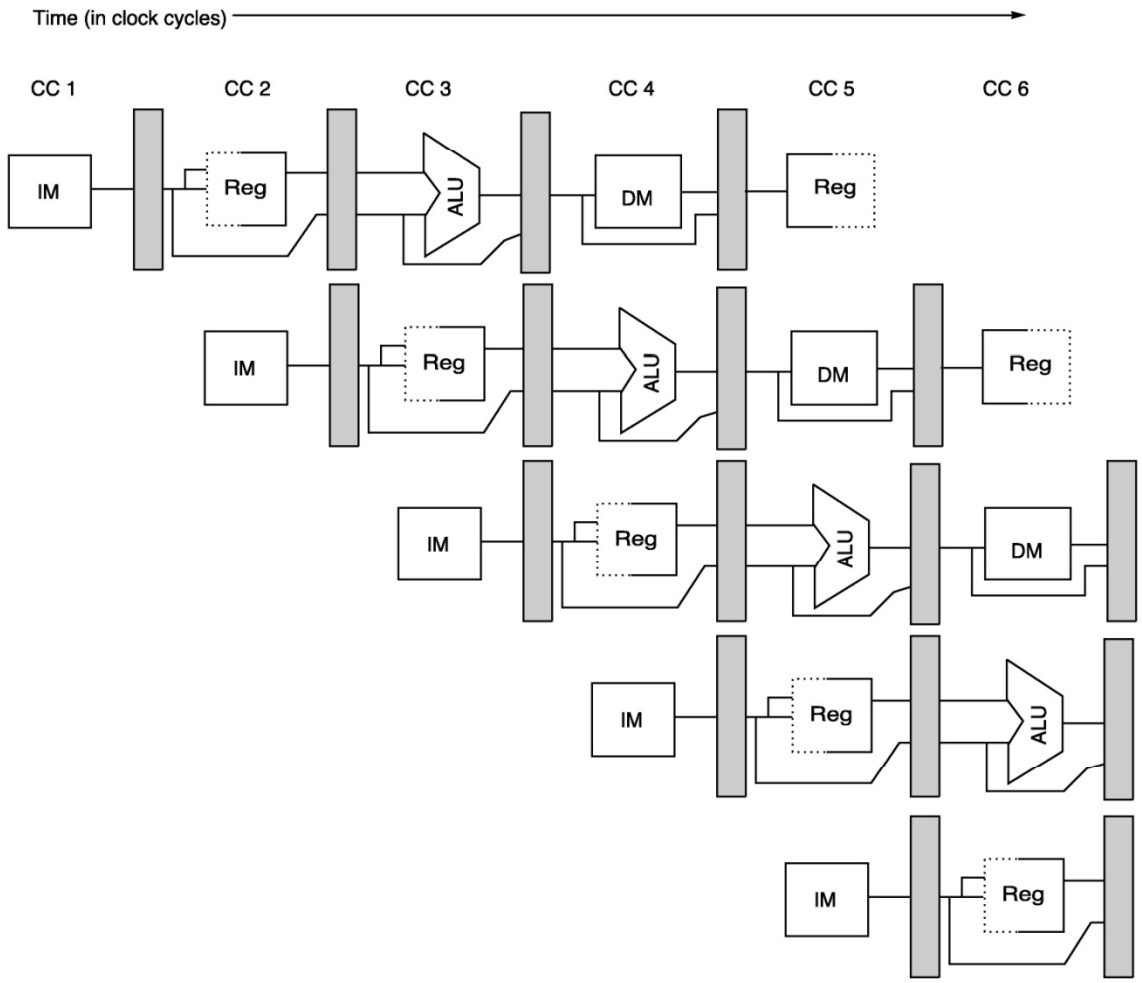


Representing Pipelined Execution



- Thought of as a series of data paths shifted in time
- Note:
  - Register Read occurs at end of ID phase
  - Register Write occurs at beginning of WB phase

Representing Pipelined Execution



- Pipeline registers made explicit

Representing Pipelined Execution

	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

- Filling the pipeline
- Steady state operation
- Emptying the pipeline
- Concurrent execution

Pipeline Performance

- Pipelining
  - At best no impact on latency
    - Still need to wait n stages (cycles) for completion of instruction
  - Improves throughput
    - No single instruction executes faster but overall throughput is better
    - Average instruction execution time improves
    - Successive instructions complete in each successive cycle (no 5 cycle wait between instructions)
  - Reality
    - Clock determined by slowest stage
    - Pipeline overhead
      - Clock skew
      - Register delay (setup time)

	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

Pipeline Performance

- Best case performance
  - Balanced pipeline
  - Zero overhead
  - Ignore pipeline fill, drain

$$\text{Time per Instruction}_{\text{pipelined}} = \frac{\text{Time per Instruction}_{\text{unpipelined}}}{\text{Number of Pipe Stages}}$$

	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

Pipeline Performance

- Example: Assume unpipelined processor with 1ns clock cycle and CPI and frequency as shown. Assume that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock. How much speedup in instruction execution rate can we expect to gain?

Instruction	Cycles	Frequency
ALU	4	40%
Branch	4	20%
Memory	5	40%

$$\begin{aligned}\text{Average Instruction Execution Time}_{\text{unpipelined}} &= \text{Clock Cycle} \times \text{Average CPI} \\ &= 1\text{ ns} \times ((40\% + 20\%) \times 4 + 40\% \times 5) \\ &= 1\text{ ns} \times 4.4 \\ &= 4.4\text{ ns}\end{aligned}$$

$$\begin{aligned}\text{Speedup from Pipelining} &= \frac{\text{Average Instruction Time}_{\text{unpipelined}}}{\text{Average Instruction Time}_{\text{pipelined}}} \\ &= \frac{4.4\text{ ns}}{\text{CPI}_{\text{pipelined}} \times (1\text{ ns} + 0.2\text{ ns})} \\ &= \frac{4.4\text{ ns}}{1 \times 1.2\text{ ns}} = \frac{4.4\text{ ns}}{1.2\text{ ns}} = 3.7\end{aligned}$$

The Major Hurdle of Pipelining: Hazards

- Hazards prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce performance from ideal speedup.
- Three classes of hazards
  - Structural hazards
    - Arise from resource conflicts when H/W cannot support all possible combinations of instructions simultaneously in the pipeline
  - Data hazards
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - Control hazards
    - Arise from pipelining branches and other instructions which change control flow of instruction sequence (i.e. change the PC)

Pipeline Performance

- Hazards may require "stalling" the pipeline: allowing earlier instructions to proceed to completion while others are delayed (and no additional instructions fetched). This is called "clearing" the hazard.

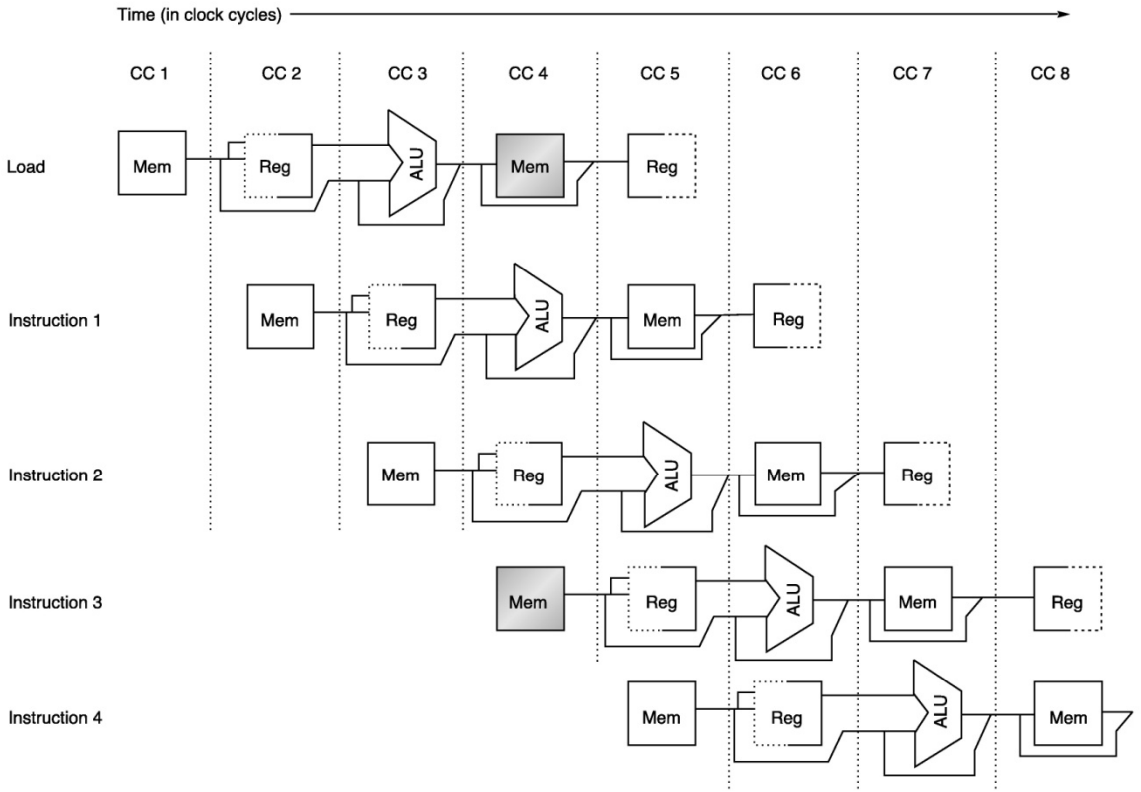
$$\text{Speedup from Pipelining} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}$$

Hazards

- Three classes of hazards
  - **Structural hazards**
    - Arise from resource conflicts when H/W cannot support all possible combinations of instructions simultaneously in the pipeline
  - Data hazards
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - Control hazards
    - Arise from pipelining branches and other instructions which change control flow of instruction sequence (i.e. change the PC)

Structural Hazards: An Example

- Two instructions both require memory access
  - Instruction Fetch
  - Load/Store Operation
- Single port memory

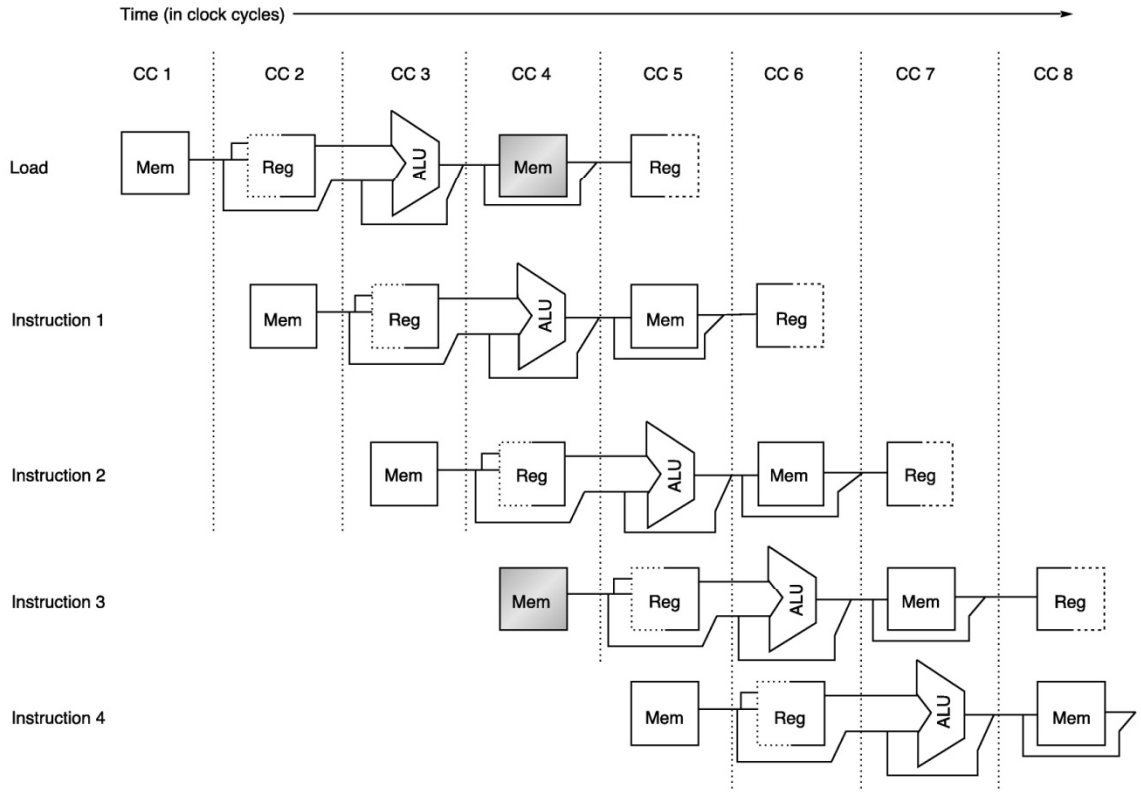


Stalls

- Instructions ahead of the stall proceed to completion
- Stall delays instruction and those following it

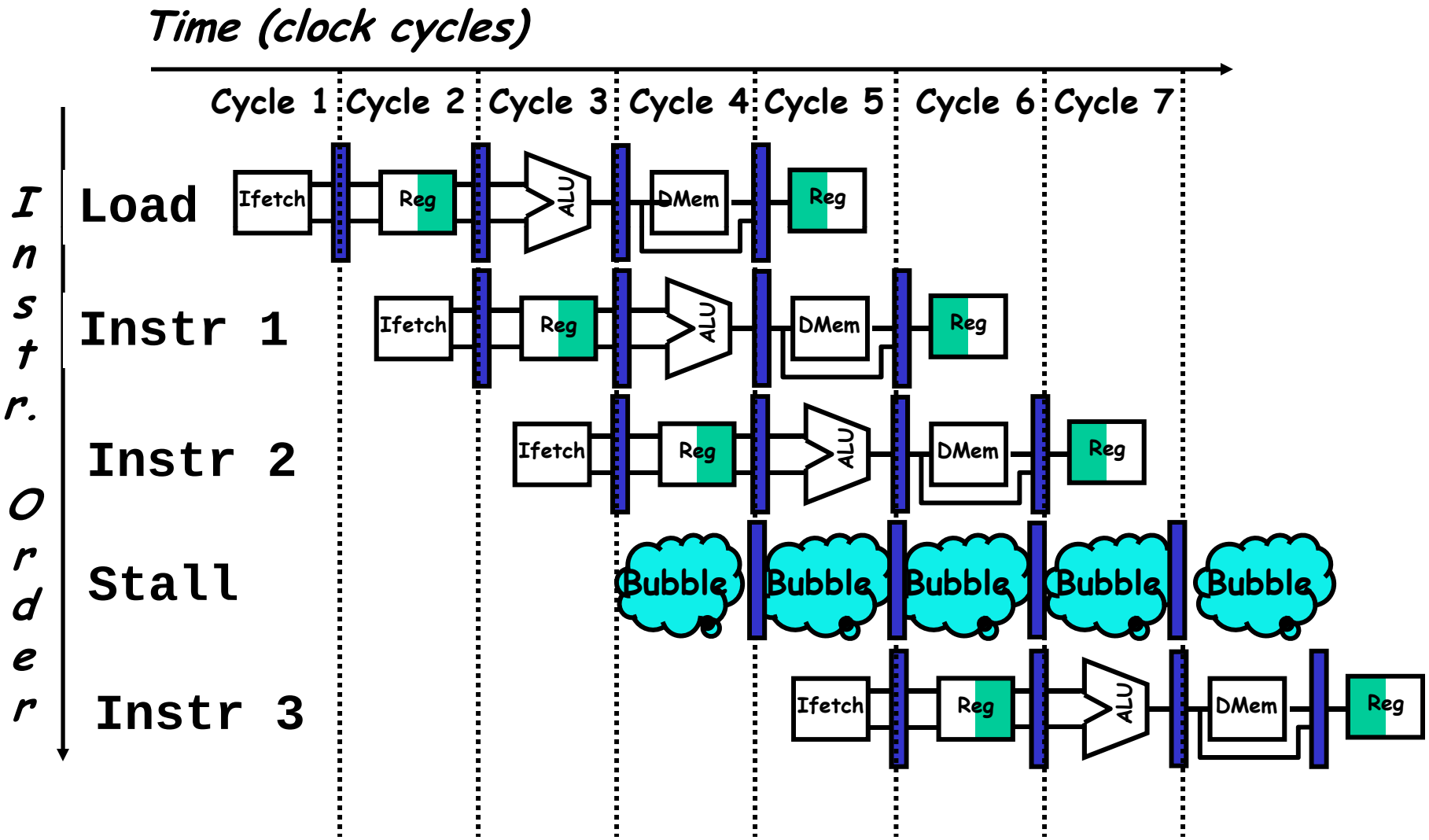
	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

Stalls



	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

One Memory Port/Structural Hazards



Cost of Stall

- Example: Consider two pipelined processors. Suppose data references represent 40% of the instructions executed and that the ideal CPI of the pipelined processor, ignoring the structural hazard, is 1. Assume the processor with the hazard has a clock rate 1.05 times higher than the hazard-free processor and incurs a one cycle stall as in our example. Which processor is faster and by how much?

$$\text{Average instruction time} = \text{CPI} \times \text{Clock Cycle Time}$$

$$\text{Average instruction time}_{\text{ideal}} = 1 \text{ x Clock Cycle Time}_{\text{ideal}}$$

$$\text{Average instruction time} = \text{CPI} \times \text{Clock Cycle Time}$$

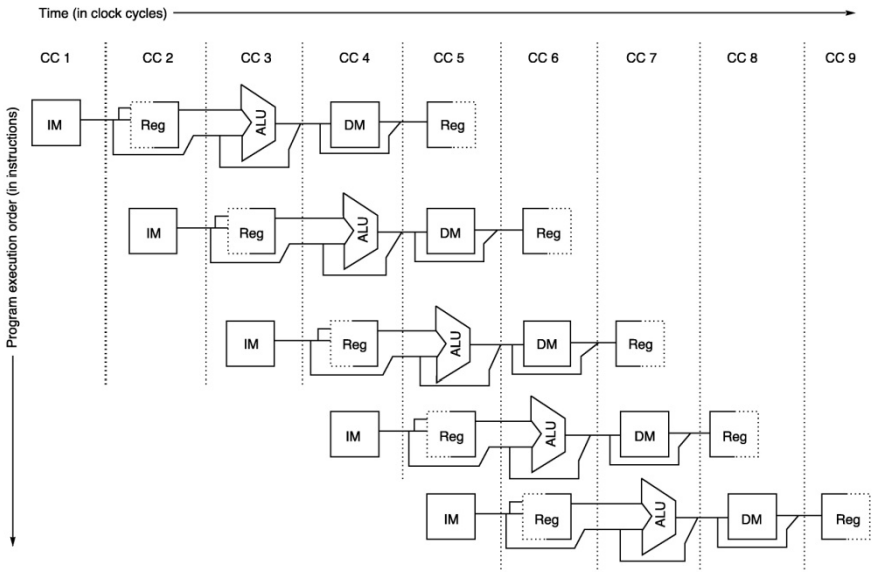
$$\text{Average instruction time}_{\text{hazard}} = (1 + 0.4 \times 1) \times \frac{\text{Clock Cycle Time}_{\text{ideal}}}{1.05}$$

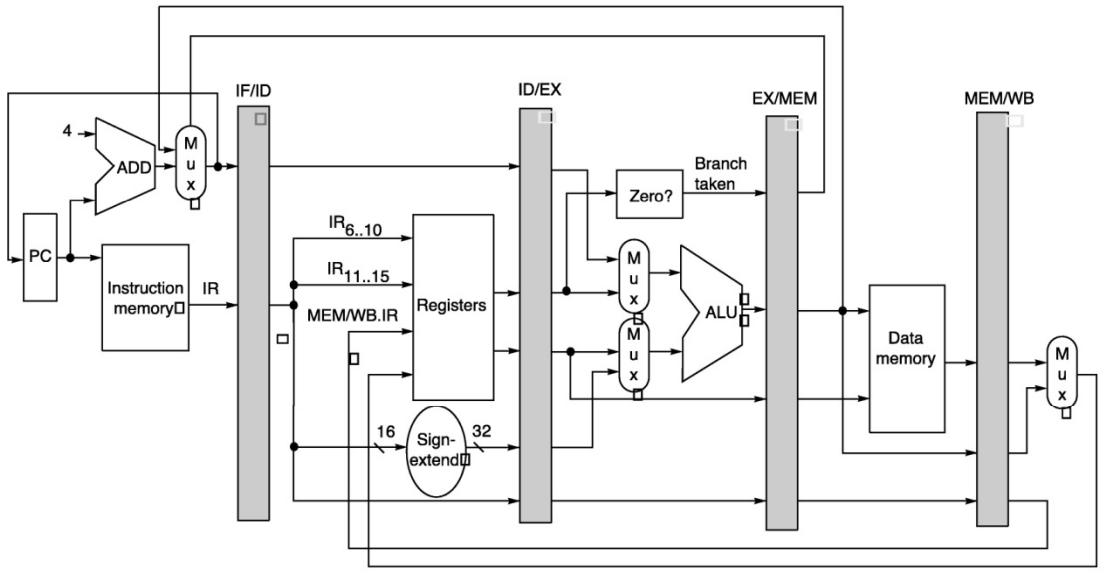
$$= 1.3 \times \text{Clock Cycle Time}_{\text{ideal}}$$

Structural Hazards

- Why no structural hazard on register access during ID phase and WB phase?
  - Dual ported register file
  - Read late in ID phase
  - Write early in WB phase
- Why no structural hazard on memory access during IF phase and MEM phase?
  - Separate instruction and data caches
- Why no structural hazard on ALU during IF phase (PC ← PC + 4) and EX phase?
  - PC + 4 in IF phase uses dedicated adder not ALU
- Eliminating structural hazards
  - Replicate hardware

Structural Hazards Avoided





Hazards

- Three classes of hazards
  - Structural hazards
    - Arise from resource conflicts when H/W cannot support all possible combinations of instructions simultaneously in the pipeline
  - **Data hazards**
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - Control hazards
    - Arise from pipelining branches and other instructions which change control flow of instruction sequence (i.e. change the PC)

Data Hazards

- Three general types of data hazards
  - Read After Write
    - Caused by “dependence” (compiler jargon). Subsequent instruction has actual need for data produced by earlier instruction.
  - Write After Read
    - Called “anti-dependence”. Can’t occur in our example MIPS pipeline. We’ll see it later in more advanced pipelines.
  - Write After Write
    - Called “output dependence”. Can’t occur in our example MIPS pipeline. We’ll see it later in more advanced pipelines.

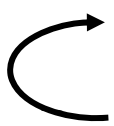
Three Generic Data Hazards

- Read After Write (RAW)
  $Instr_J$ tries to read operand before $Instr_I$ writes it

- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards

- Write After Read (WAR)
  $Instr_J$ writes operand *before* $Instr_I$ reads it

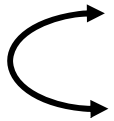


```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1"
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages
  - Reads are always in stage 2
  - Writes are always in stage 5

Three Generic Data Hazards

- Write After Write (WAW)
  $Instr_J$ writes operand *before* $Instr_I$ writes it.

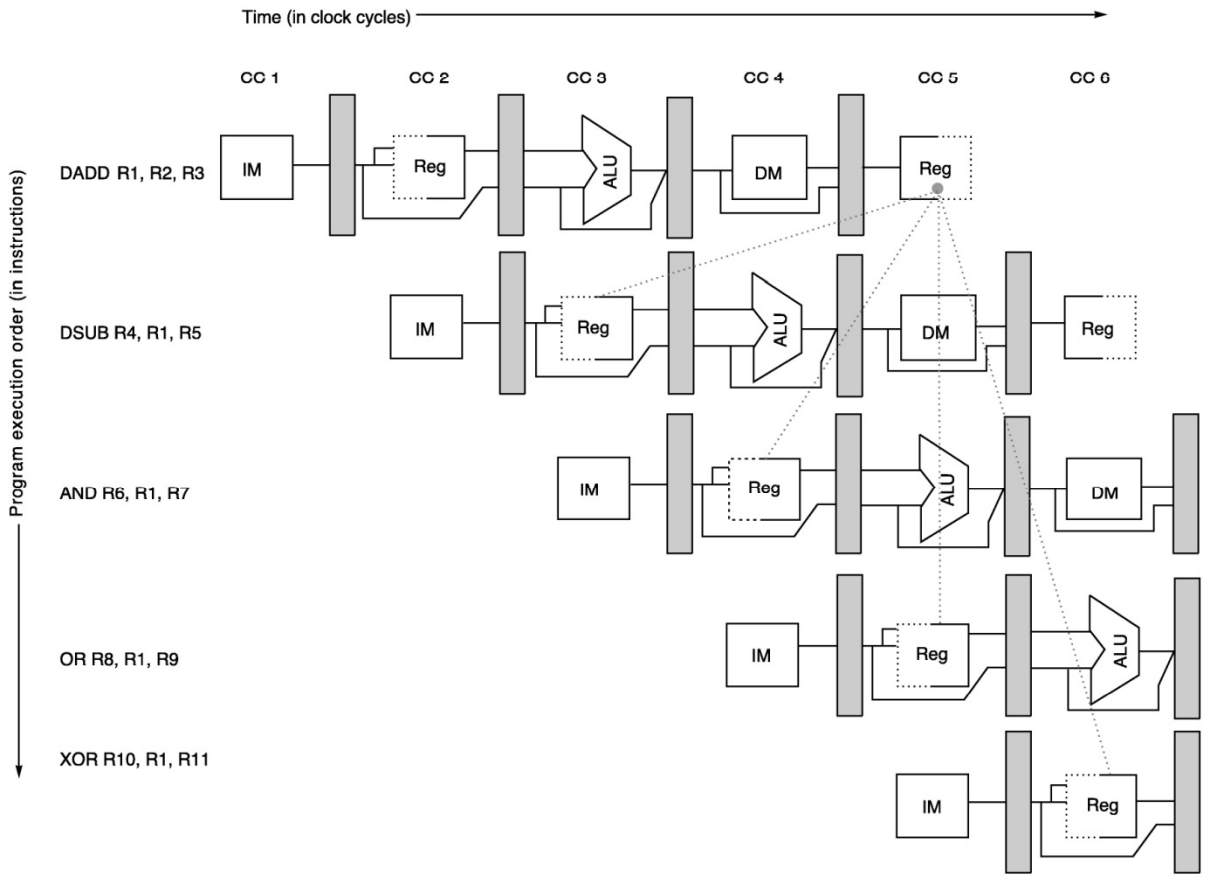


```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages
  - Writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

Read After Write (RAW) Data Hazard

```
DADD   R1,R2,R3
DSUB   R4,R1,R5
AND    R6,R1,R7
OR     R8,R1,R9
XOR    R10,R1,R11
```

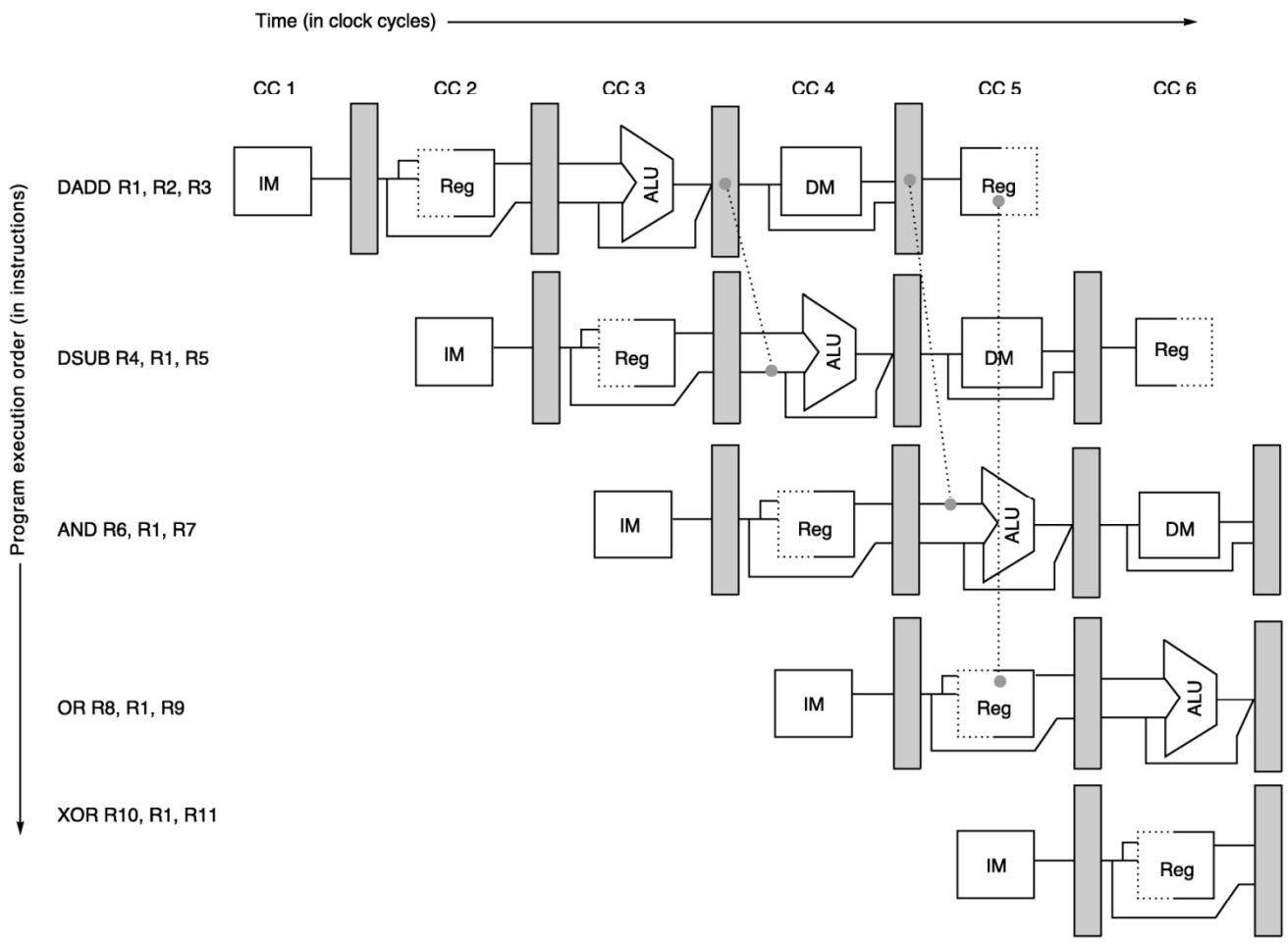


RAW Data Hazard

- Would like to avoid stalls caused by RAW data hazards
  - Significant hit to performance
  - Technique: “forwarding”
    - Also called “bypassing” or “short-circuiting”
  - Forward results from later in the pipeline to earlier stages

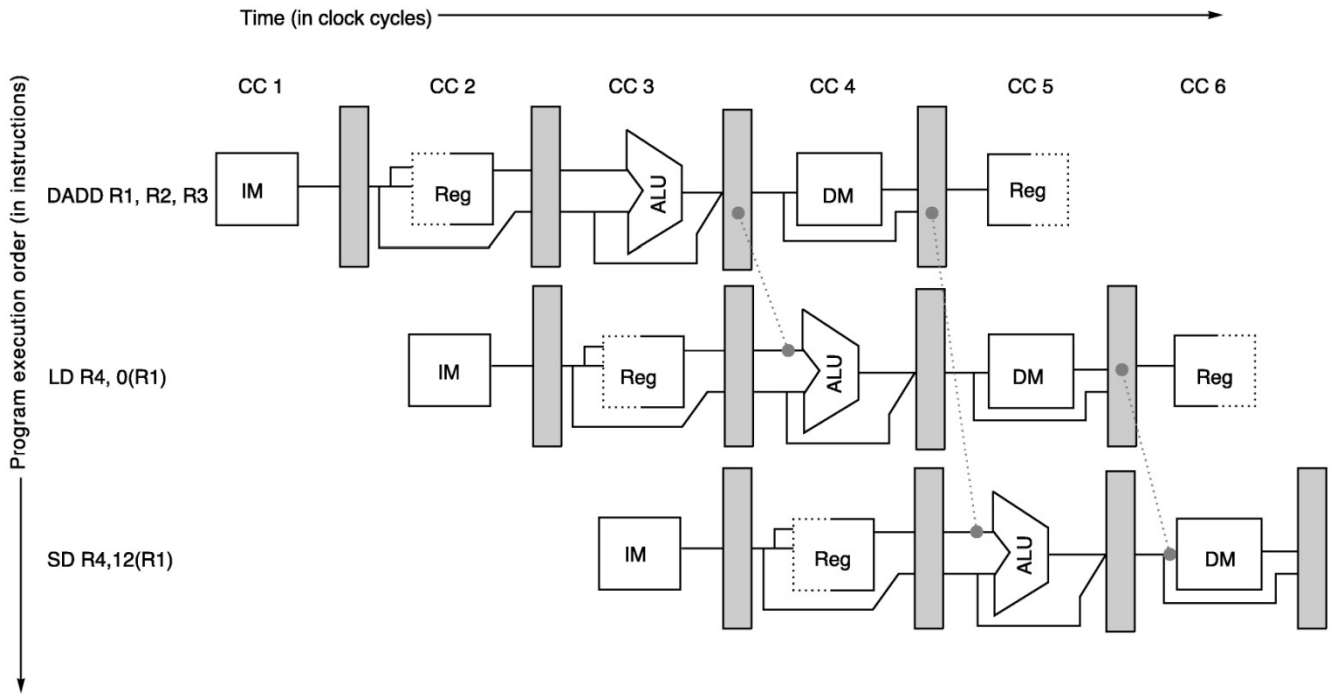
Forwarding

```
DADD   R1,R2,R3
DSUB   R4,R1,R5
AND    R6,R1,R7
OR     R8,R1,R9
XOR    R10,R1,R11
```

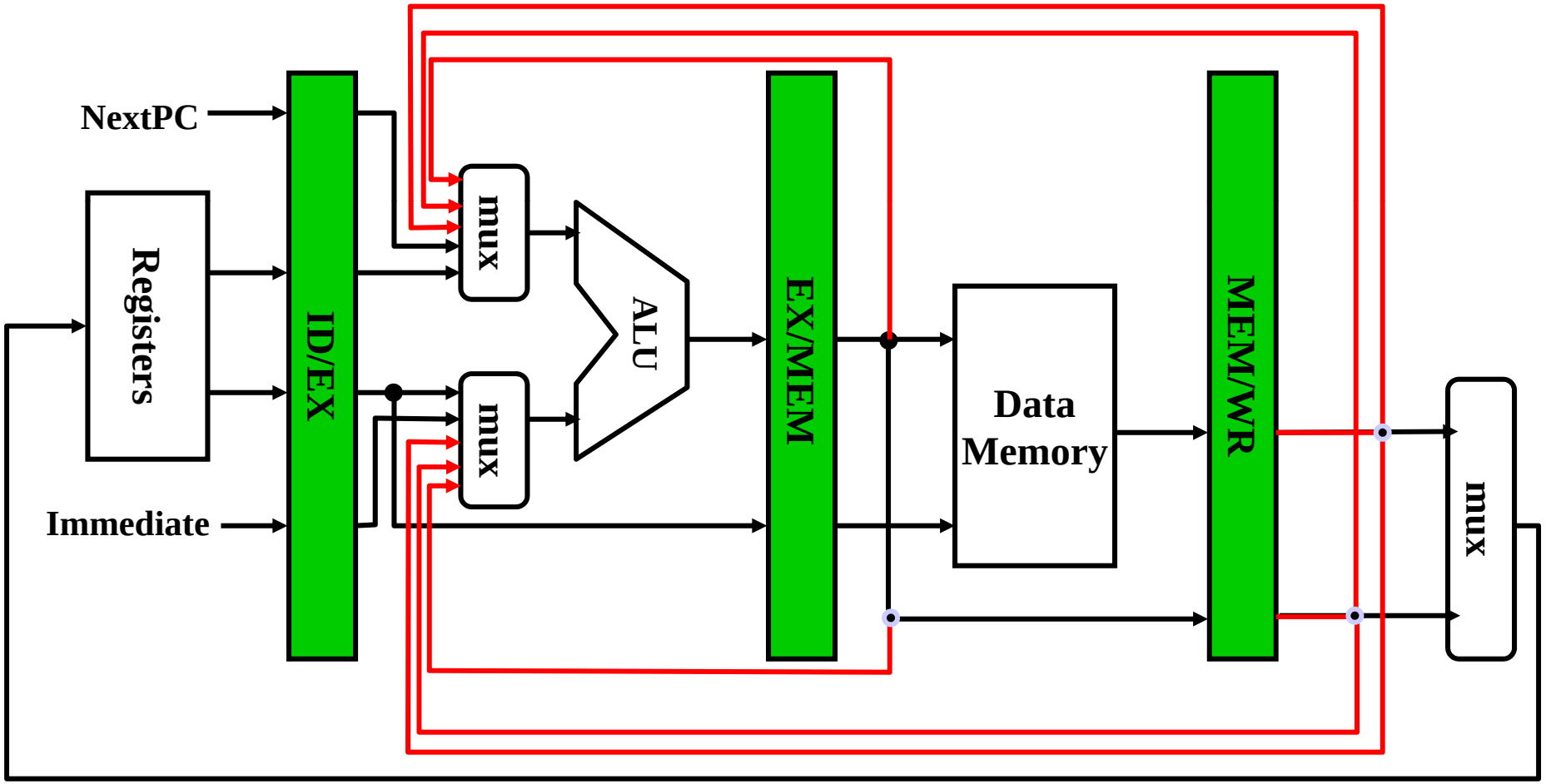


Another Example

```
DADD   R1,R2,R3
LD     R4, 0(R1)
SD     R4, 12(R1)
```

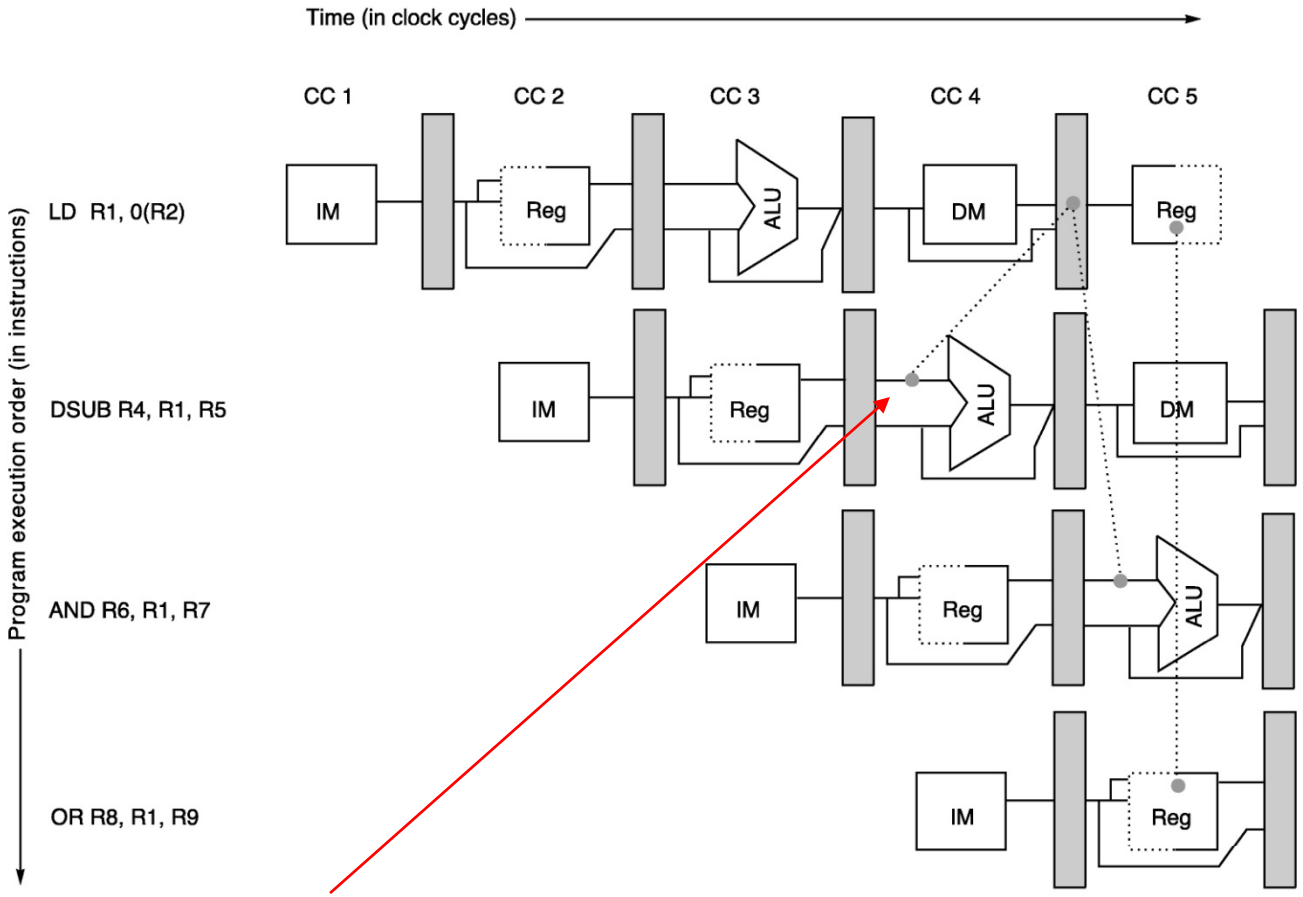


HW Change for Forwarding



Data Hazards Requiring Stalls Even With Forwarding

```
LD      R1,0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```



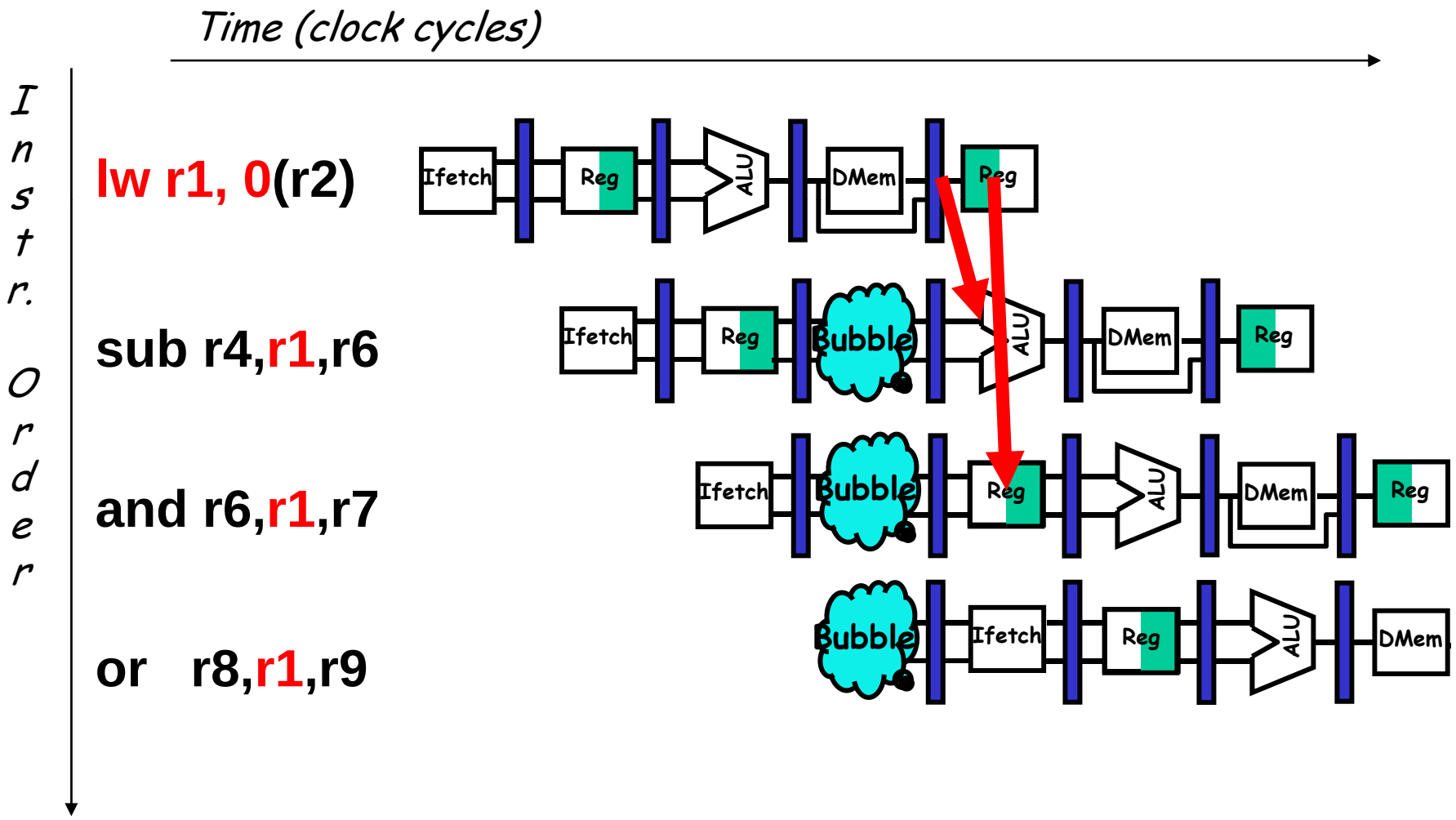
Would require forwarding backward in time!

Data Hazards Requiring Stalls Even With Forwarding

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB			
AND	R6,R1,R7			IF	ID	EX	MEM	WB		
OR	R8,R1,R9				IF	ID	EX	MEM	WB	

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

Data Hazard Even with Forwarding



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    Rc,c
ADD   Ra,Rb,Rc
SW    Ra,a
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    Rd,d
```

Data hazard forces stall (LW Rc,c / ADD Ra,Rb,Rc)

Data hazard forces stall (LW Rf,f / SUB Rd,Re,Rf)

Fast code:

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    Ra,a
SUB   Rd,Re,Rf
SW    Rd,d
```

Hazards

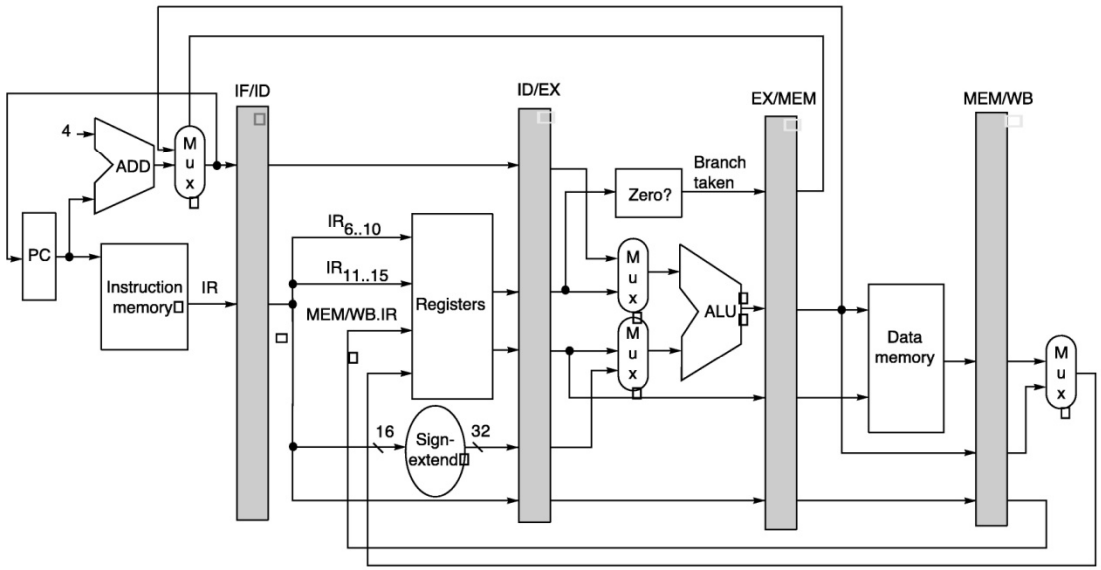
- Three classes of hazards
  - Structural hazards
    - Arise from resource conflicts when H/W cannot support all possible combinations of instructions simultaneously in the pipeline
  - Data hazards
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - **Control hazards**
    - Arise from pipelining branches and other instructions which change control flow of instruction sequence (i.e. change the PC)

Control Instructions

- Branches
  - Conditional
  - Condition codes or register compare
  - “Taken” or “Not taken”
    - Taken if PC modified (not taken if next instruction is sequential one)
  - MIPS:
    - Register compares
    - Target is immediate offset to PC
- Jumps
  - Unconditional
- Procedure Call
  - MIPS: Implemented with Jump (JAL – jump and link)

Branch Hazards

```
        BEQZ  R1,L1
        successor?
        successor? + 1
        successor? + 2
        .
        .
        .
L1:     successor?
        successor? + 1
        successor? + 2
```



Branch instruction	IF	ID	EX	MEM	WB
Branch successor		IF	stall	stall	IF
Branch successor + 1					
Branch successor + 2					

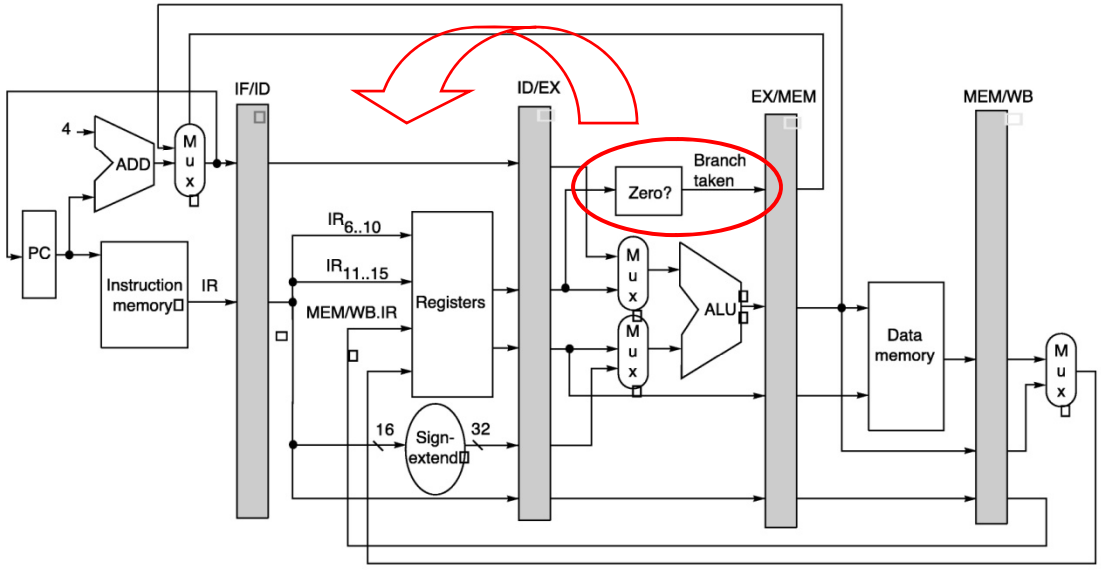
Need to know the result of the condition test and the target branch address. In the pipeline above, target branch address is computed and condition is tested in EX. However, PC isn't updated until MEM, resulting in a 3-cycle penalty. In fact, we don't even know we have a branch instruction until ID stage, so next instruction must be fetched sequentially. Then the pipeline stalls for two cycles before fetching correct successor instruction.

Implementing the Pipeline

Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM. ALUOutput} 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.ALUOutput ← ID/EX.A *func* ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR← ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

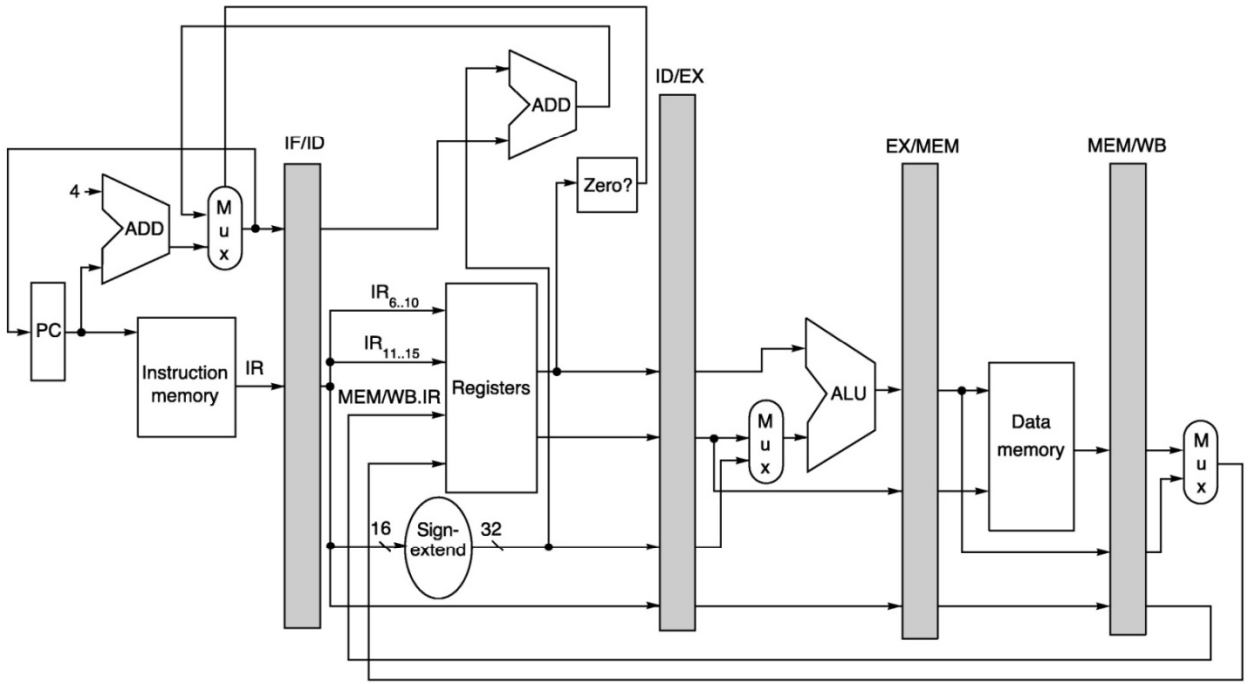
Branch Hazards

```
        BEQZ  R1,L1
        successor?
        successor? + 1
        successor? + 2
        .
        .
        .
L1:     successor?
        successor? + 1
        successor? + 2
```



Can modify the data path to move conditional test earlier in pipeline. But still won't have target address any sooner. Need to further modify data path with addition of an adder to compute target address in ID and update PC earlier, reducing 3 cycle stall to single cycle stall.

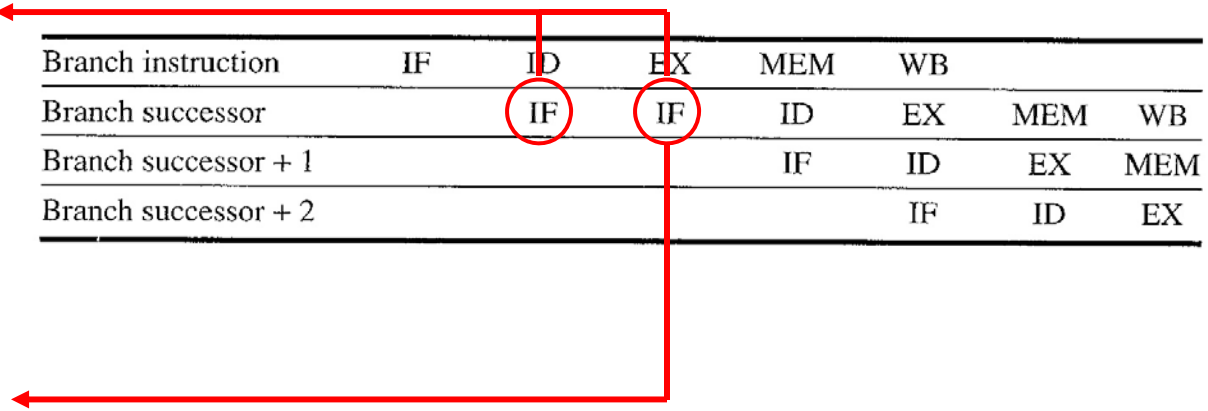
Reducing Stall from Branch Hazards



Pipe stage	Branch instruction
IF	IF/ID.IR ← Mem[PC]; **IF/ID.NPC,PC ← (if ((IF/ID.opcode == branch) & (Regs[IF/ID.IR$_{6..10}$]** **op 0)) {IF/ID.NPC + sign-extended (IF/ID.IR[immediate field] <<2) else {PC+4});**
ID	ID/EX.A ← Regs[IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs[IF/ID.IR$_{11..15}$]; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$
EX	
MEM	
WB	

Branch Hazards

```
        BEQZ  R1,L1
        successor?
        successor? + 1
        successor? + 2
        .
        .
        .
L1:     successor?
        successor? + 1
        successor? + 2
```



Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

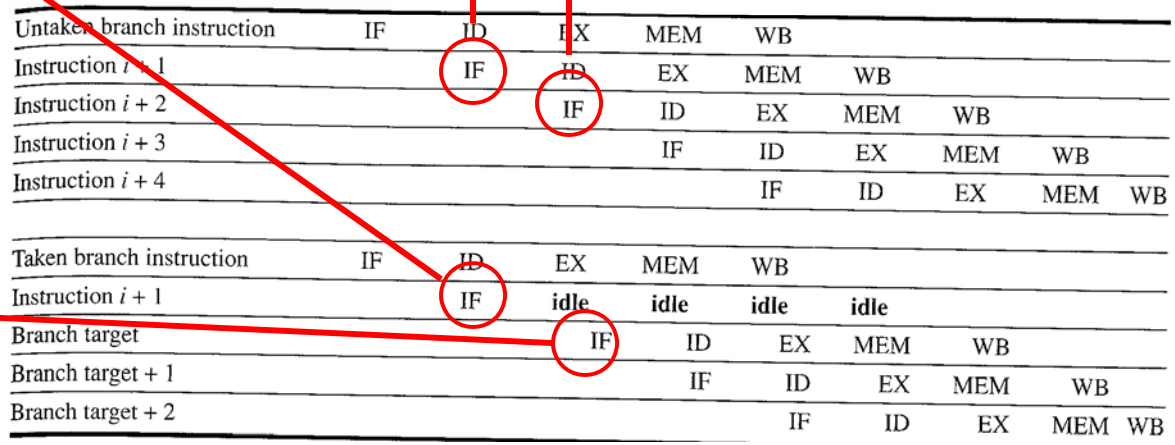
At cycle 2 we haven't even decoded the branch yet, so no choice but to fetch next sequential instruction. At cycle 3, we've detected branch, know the result of the condition test, and have the computed target available. So IF in 3rd cycle can fetch the branch target or next sequential instruction….But why re-fetch next sequential instruction if branch not taken?

Handling Branch Hazards

- Flush the pipeline
  - Stall pipeline until branch destination is known
  - Simple
  - Low performance
- Treat every branch as not taken
  - “Predicted not taken” scheme
  - Slightly more complex
  - If branch untaken, no change – keep executing
  - If branch taken, turn fetched instruction into no-op and fetch at target address

Predicted-not-taken scheme

```
        BEQZ   R1,L1
        successor?
        successor? + 1
        successor? + 2
        .
        .
        .
L1:     successor?
        successor? + 1
        successor? + 2
```



If branch untaken, no stall and therefore no performance penalty. If the branch is taken, we must turn the fetched sequential successor instruction into a NOP in the pipeline and fetch the correct successor instruction, incurring a one cycle stall.

Alternative Branch Strategies

- Treat every branch as taken ("predict taken")
  - In our 5-stage pipeline we don't know target any earlier than we know branch outcome
    - No advantage
  - Some processors this isn't true (may know target before outcome)
    - Implicitly set condition codes
    - More powerful (slower) branch instructions
  - Compilers can improve performance by using knowledge of the architecture to organize code to take advantage of processor's choice
- Delayed branch
  - Frequently used (in RISC and other processors)

Delayed Branch

- Assume branch delay of one
- If branch taken, execution is:

```
Branch instruction
Branch delay instruction
Branch target
```

- If branch untaken, execution is:

```
Branch instruction
Branch delay instruction
Instruction + 2
```

```
        BEQZ    R1,L1
        branch delay instruction
        instruction + 2
        instruction + 3
        …
L1:     branch target
        branch target + 1
        branch target + 2
```

- Instruction immediately following branch is executed whether the branch is taken or not
- Rely on compiler to make successor instructions valid and useful

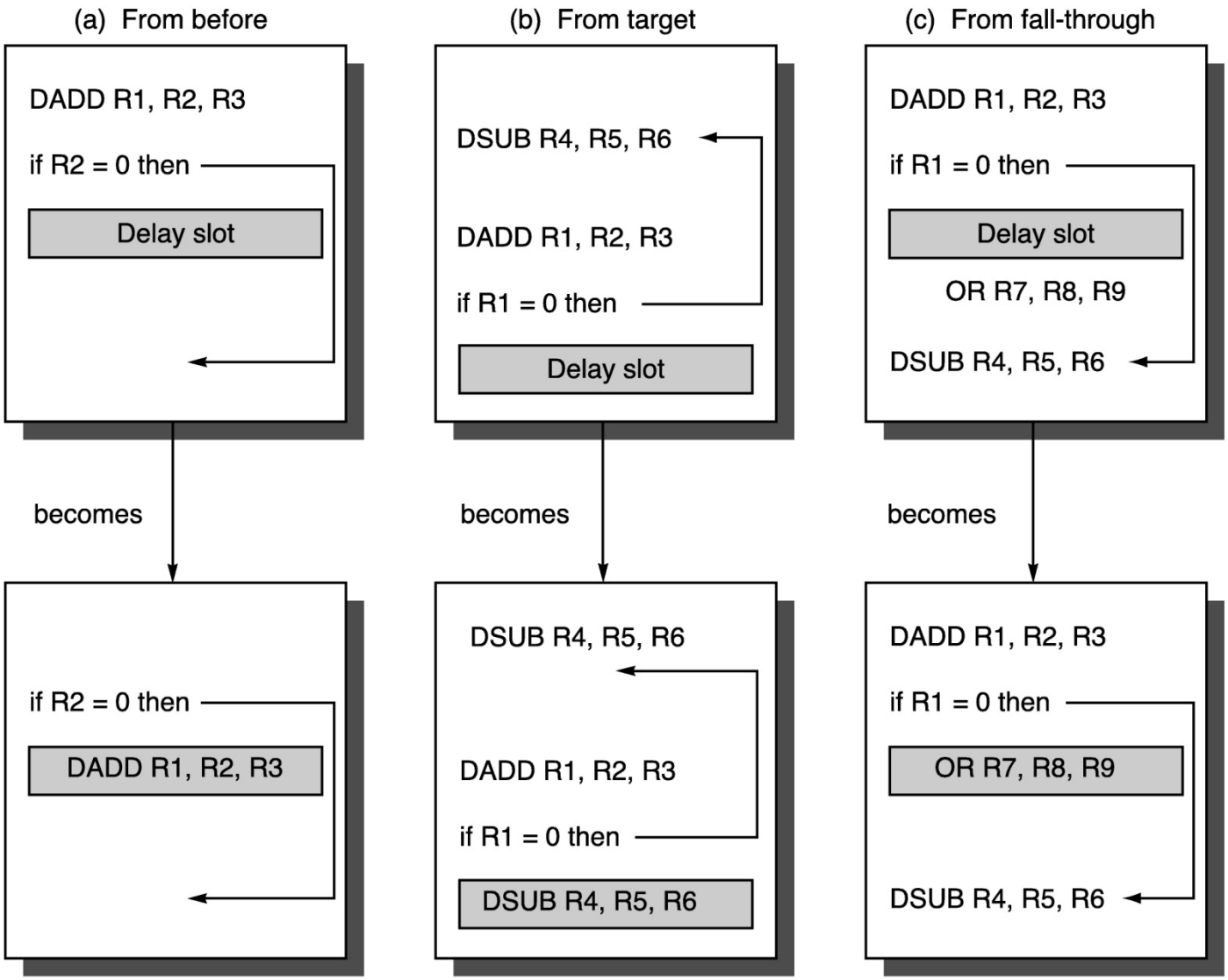
Behavior of Delayed Branch

```
        BEQZ    R1,L1
        branch delay instruction
        instruction + 2
        instruction + 3
        …
L1:     branch target
        branch target + 1
        branch target + 2
```

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

Scheduling the Branch Delay Slot



Control Hazard Impact

- Frequency of branch instructions
- Potential impact
  - 1 cycle in simple 5 stage MIPS
  - More complicated branches & longer pipelines
    - → 3 cycles
- Compare approaches

Type	Frequency
Unconditional	4%
Conditional (Not Taken)	6%
Conditional (Taken)	10%

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

Figure A.15 Branch penalties for the three simplest prediction schemes for a deeper pipeline.

	Additions to the CPI from branch costs			
Branch scheme	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

$$\frac{\text{Speedup Predicted-not-taken}}{\text{Stall pipeline}} = \frac{1.56}{1.38} = 1.13$$

Example: Branch Stall Impact

- If 30% branch, stall 3 cycles significant
- Two part solution:
  - Determine branch taken or not sooner
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
  - How do they get away with this?
  - What about >, <, ≥ , ≤ ?
    - Remember benchmarks and instruction frequency counts?
    - Make the common case fast
    - SLT instruction and subsequent test
- MIPS Solution:
  - Move zero test to ID stage
  - Adder to calculate new PC in ID stage; update PC earlier
  - 1 clock cycle penalty for branch versus 3

Cost of Stall

- Example: Assume CPI of 1 for a pipelined processor and that branches comprise 30% of all instructions and incur a 3 cycle stall. How much slower is the pipeline than its theoretical throughput because of the branch stalls?

$$\begin{aligned} CPI_{hazard} &= (1 + 30\% \times 3) \times CPI_{ideal} \\ &= 1.9 \times CPI_{ideal} \end{aligned}$$

Nearly 2x slower!

Cost of Stall

- Example: Assume CPI of 1 for a pipelined processor and that branches comprise 30% of all instructions and incur a 3 cycle stall. How much slower is the pipeline than its theoretical throughput because of the branch stalls?

$$\text{Speedup from Pipelining} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}$$

$$= \frac{1}{1 + 3 \text{ x } 30\%} \times \text{Pipeline depth}$$

$$= \frac{1}{1.9}$$

Nearly 2x slower!

Speed Up Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

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

For simple RISC pipeline, CPI = 1:

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

Implementing the Pipeline: Detecting Hazards

Situation	Example code sequence	Action

No dependence	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R6,R7`
	`OR R9,R6,R7`
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

Dependence requiring stall	`LD R1,45(R2)`
	`DADD R5,R1,R7`
	`DSUB R8,R6,R7`
	`OR R9,R6,R7`
		Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.

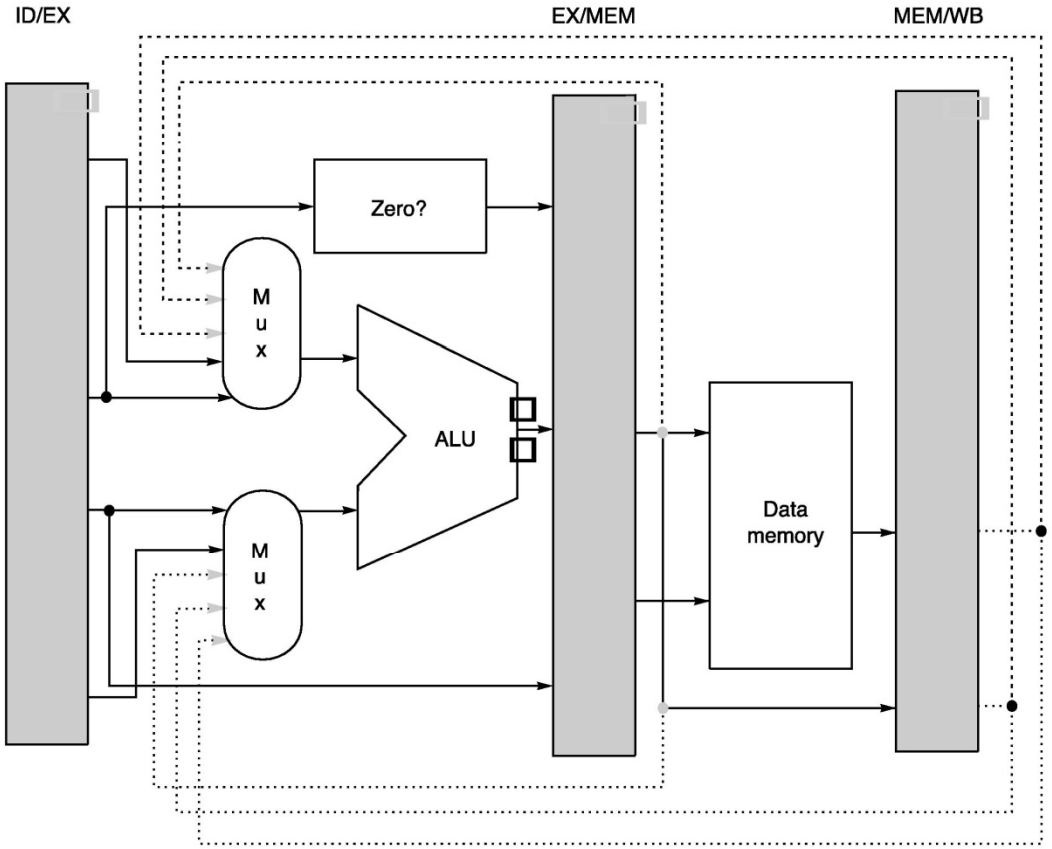
Dependence overcome by forwarding	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R1,R7`
	`OR R9,R6,R7`
		Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.

Dependence with accesses in order	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R6,R7`
	`OR R9,R1,R7`
		No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Detecting need for load interlocks during ID

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..5}$)	Matching operand fields
Load	Register-register ALU	ID/EX.IR[rt] == IF/ID.IR[rs]
Load	Register-register ALU	ID/EX.IR[rt] == IF/ID.IR[rt]
Load	Load, store, ALU immediate, or branch	ID/EX.IR[rt] == IF/ID.IR[rs]

Forwarding Results to ALU



- Three additional inputs to each ALU multiplexer
  - ALU Output from end of EX phase
  - ALU Output from end of MEM phase
  - MEM Output from end of MEM phase

Implementing the Pipeline: 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.IR[rd] == ID/EX.IR[rs]
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rd] == ID/EX.IR[rt]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rd] == ID/EX.IR[rs]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rd] == ID/EX.IR[rt]
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rt] == ID/EX.IR[rs]
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rt] == ID/EX.IR[rt]
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rt] == ID/EX.IR[rt]
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rt] == ID/EX.IR[rt]

What Makes Pipelining Difficult to Implement

- Exceptions
  - Interruptions to “normal” program flow
  - Usually outside of program’s control
  - Common “mechanism”
    - Save processor state (PC, flags, etc)
    - Handle the exception
    - Resume execution from point of interruption

Exceptions

- Common mechanism for variety of reasons
  - I/O device request
  - User program invoking OS call
  - Instruction execution trace
  - Breakpoints
  - Integer arithmetic overflow
  - Floating point arithmetic anomaly
  - Page fault
  - Misaligned memory address reference
  - Memory protection violation
  - Undefined/unimplemented instruction
  - Hardware malfunction
  - Power failure

Exceptions

- Variety of names on same and different systems
  - Exception
  - Trap
  - Fault
  - Interrupt
  - Machine check

Exceptions

- Can be characterized along 5 independent axes
  1. Synchronous vs. Asynchronous
     - Synchronous: occurs at same place every time program is executed with same data and memory allocation
     - Asynchronous: Hardware malfunctions and I/O devices (can be handled after completion of current instruction)
  2. User requested vs. Coerced
     - User requested: Can be handled after instruction completes
  3. User maskable vs. user nonmaskable
  4. Within vs. Between instructions
     - Within usually synchronous (e.g. divide by zero, page fault) since it's the instruction that triggers the exception. May require (e.g. page fault) instruction be stopped and later restarted.
     - Asynchronous within instructions usually H/W malfunction and program terminates
  5. Resume vs. Terminate
     - Terminate: program will always stop after exception
     - Resume: program will continue execution after exception

Exceptions

- Most difficulty arises with exceptions that occur within instructions and which must permit resumption of the program
- Requires “restartable” pipeline
  - Instruction is “in execution”
    - Instruction in pipeline, partly executed
  - Succeeding instructions may also be “in execution”
- Precise and imprecise exceptions
  - Precise: pipeline can be stopped so that instructions just before the faulting instruction are completed but the faulting instruction and those following are able to be restarted correctly (required by IEEE 754 floating point, demand paging)
    - No change of processor state!

Exceptions in MIPS

Pipeline Stage	Exceptions
IF	Page fault on instruction fetch
IF	Misaligned memory access
IF	Memory protection violation
ID	Undefined/illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch
MEM	Misaligned memory reference
MEM	Memory protection violation
WB	none

- Multiple exceptions can occur in same clock cycle

```
LD          ; page fault in MEM
DADD        ; arithmetic exception in EX
```

- Exceptions may occur out of order

```
LD          ; page fault in MEM
DADD        ; page fault in IF  (occurs earlier!)
```

MIPS Exceptions

- Must handle exceptions in instruction sequence order, not in order exception occurs (which is an artifact of the pipeline)
- Exception status vector carried along pipeline along with instruction. Any exceptions caused by that instruction in a given stage cause an entry to be made in the status vector
- Control logic in each stage prevents data writes by the instruction if an exception has been posted to the exception status vector (includes register writes and memory writes)
- When instruction leaves MEM or enters WB the exception status vector is checked and the exception handled. This ensures correct ordering.

Further Complications

- ISA can further complicate things…
  - "commit": when an instruction is guaranteed to complete and writes can be permitted to occur or become permanent (e.g. if "posted" temporarily to a write buffer)
  - MIPS instructions have single result (register or memory)
  - MIPS instructions commit on MEM/WB
- Some ISAs have instructions which change processor state during execution (e.g. IA-32, VAX)
  - Auto-increment addressing modes
  - String copy instructions
- Need ability to "back out" state changes
- Need ability to resume execution of instruction
  - Registers with intermediate results (e.g. index/count) saved on exception

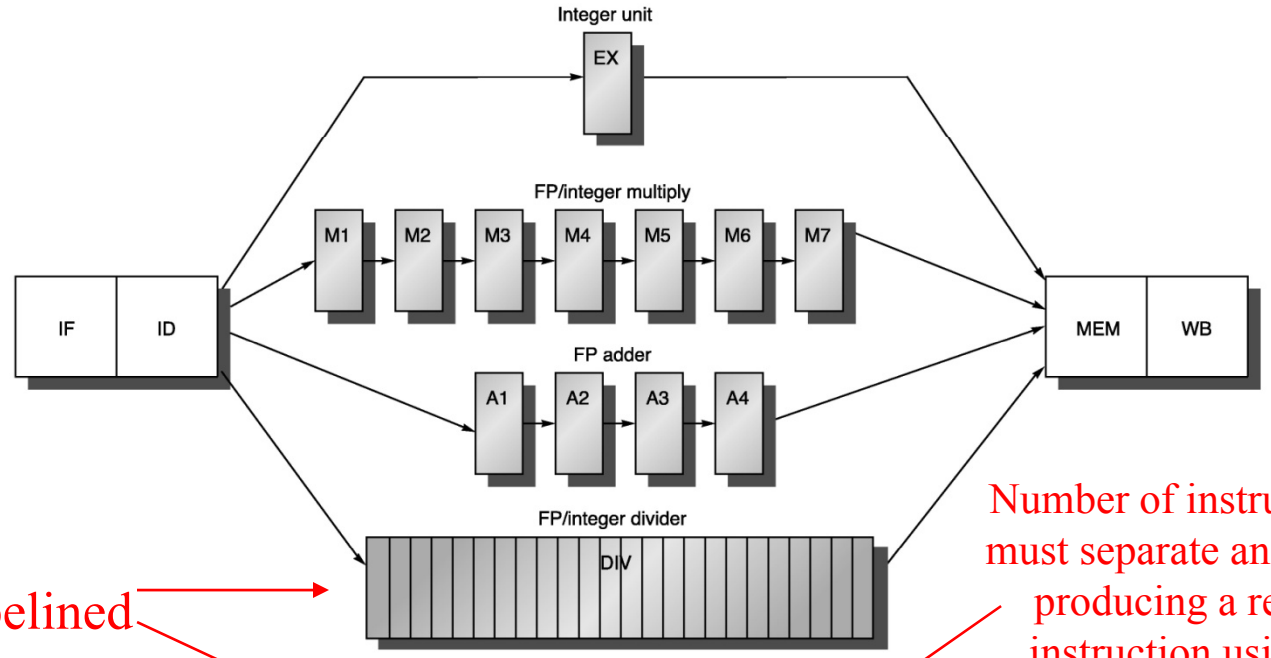
Further Complications

- "Odd" bits of state…
  - Condition codes implicitly set by preceding instructions
  - May prevent effective scheduling of instruction in delay slot
  - When is branch condition "fixed"?
    - Depends on which instructions are capable of setting condition codes
    - Delay branch condition evaluation until all prior instructions have had chance to set condition codes
    - ISAs with explicitly set condition codes can schedule the delay
    - Treat condition codes as operands in RAW hazard detection
- Multicycle operations
  - VAX instructions can require widely different number of cycles to complete
  - VAX instructions can perform none to hundreds of memory references

```
MOVL   R1,R2                  ; move between registers
ADDL3  42(R1),56(R1)+,@(R1);  add memory locations
SUBL2  R2,R3                  ; subtract registers
MOVC3  @(R1)[R2],74(R2),R3 ;  string copy
```

- A solution: IA-32 from 1995, VAX 8800
  - Microcoded implementations: pipeline the microcode

MIPS Pipelines with Multicycle Operations

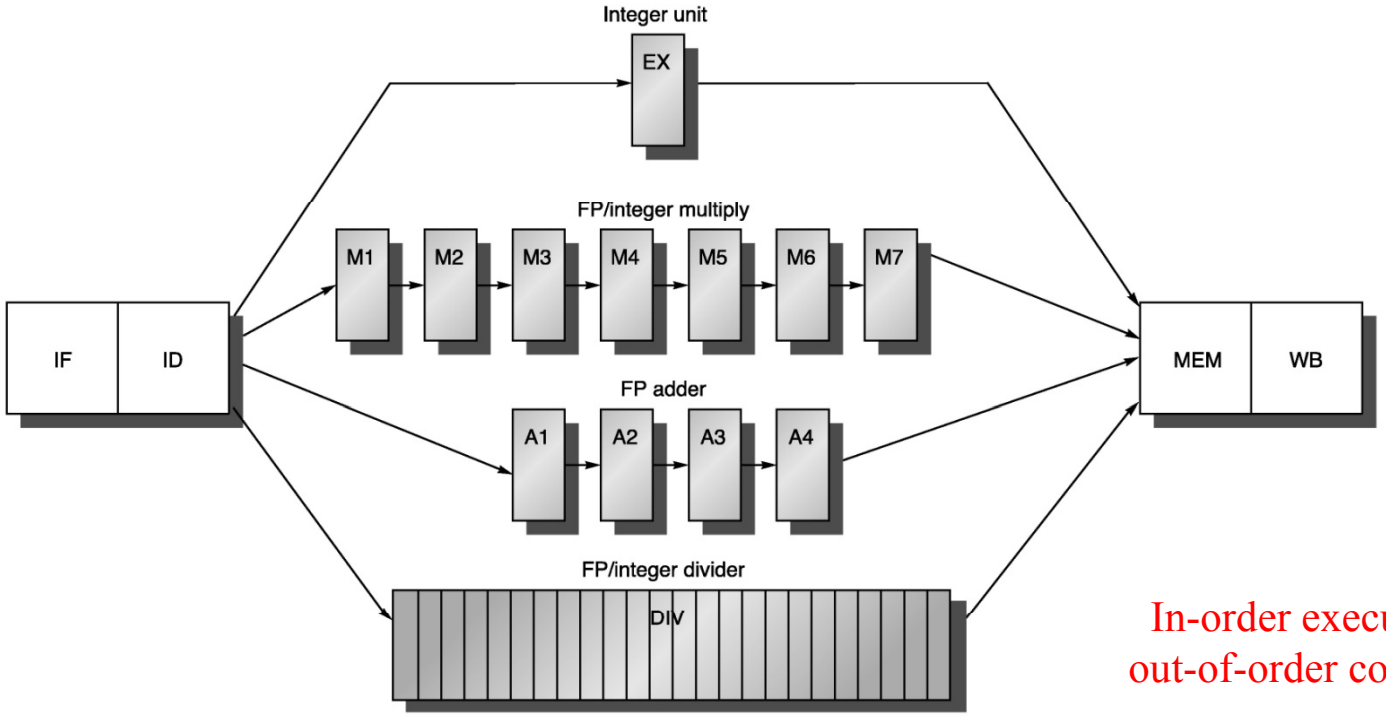


Not pipelined

Number of instructions that must separate an instruction producing a result and instruction using result

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Multi-Cycle Operations

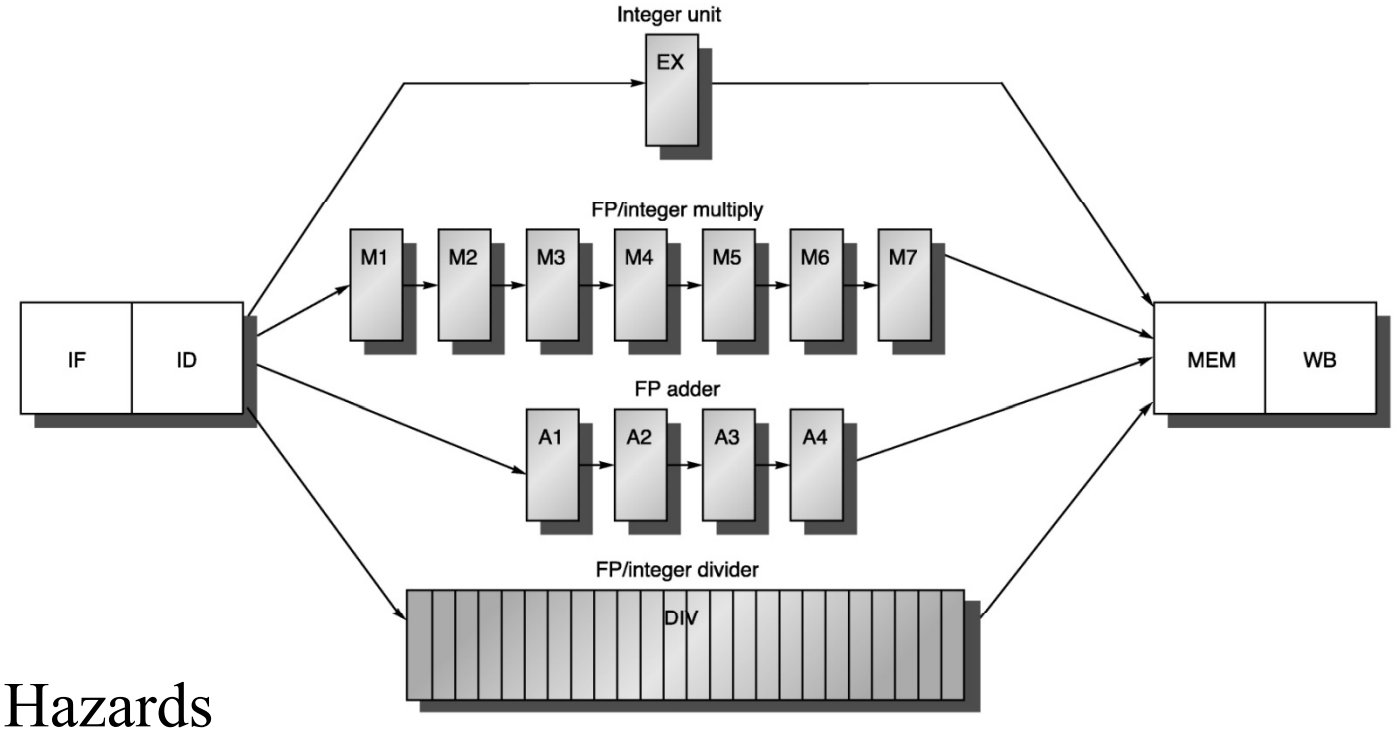


In-order execution but
out-of-order completion!

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

Figure A.32 The pipeline timing of a set of independent FP operations. The stages in italics show where data are needed, while the stages in bold show where a result is available. The ".D" extension on the instruction mnemonic indicates double-precision (64-bit) floating-point operations. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

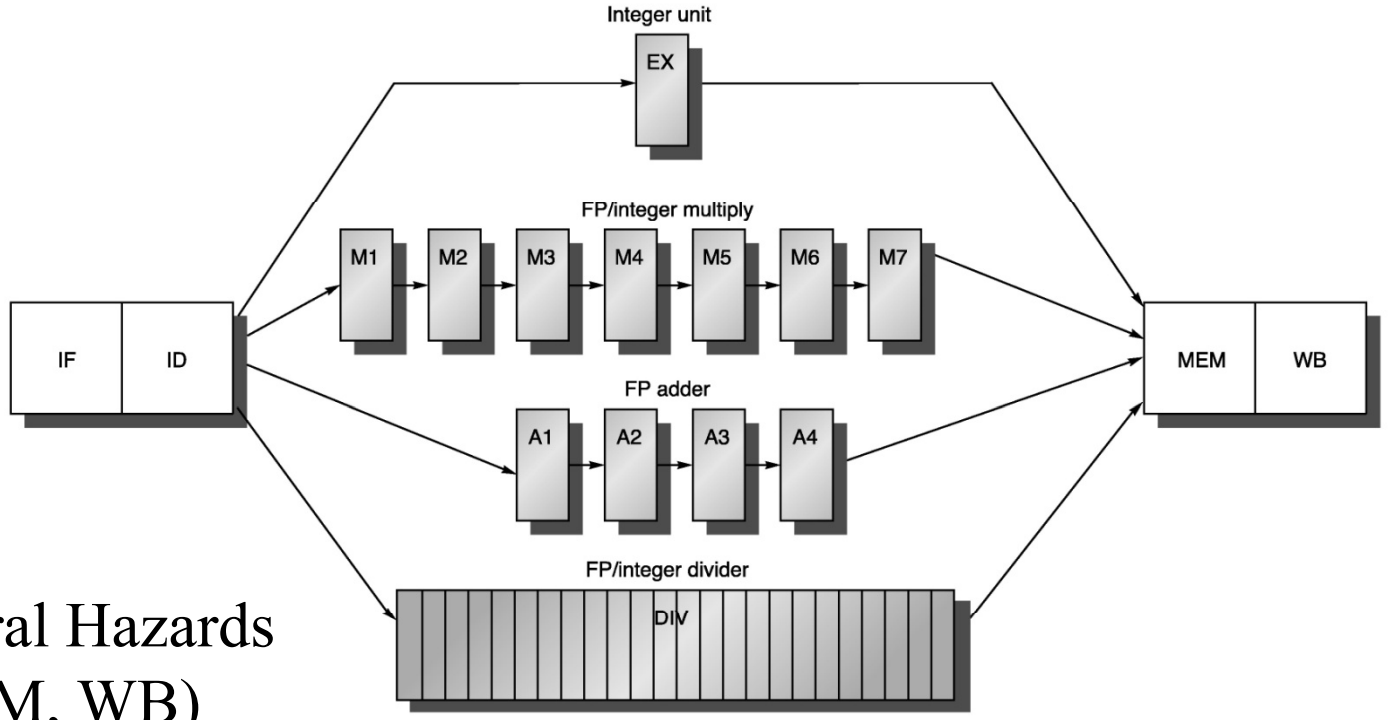
Multi-Cycle Operations



RAW Hazards

Instruction	Clock cycle number																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D F0,F4,F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2,F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	
S.D F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

Multi-Cycle Operations

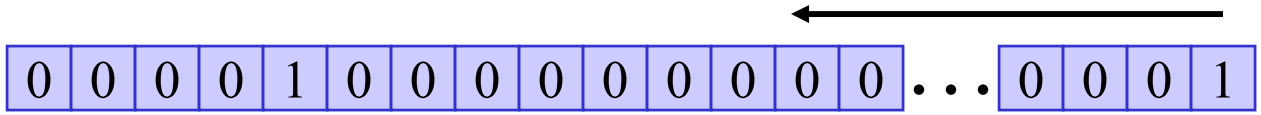


Structural Hazards (MEM, WB)

	Clock cycle number										
Instruction	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		
...						IF	ID	EX	MEM	WB	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

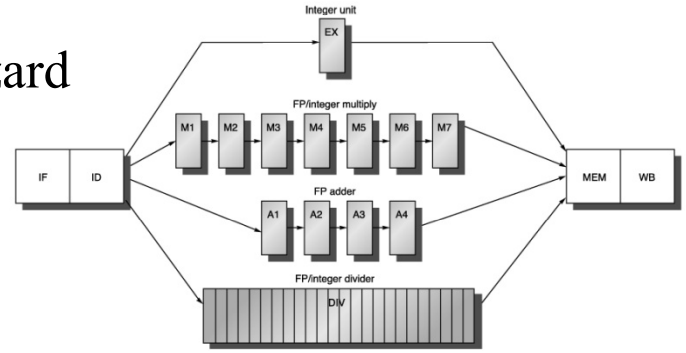
Two Solutions

- Continue MIPS implementation of detecting hazards and stalling at the ID stage
  - Use shift register as long as the number of stages with each bit representing need for an instruction to write to memory during that stage
  - During ID, instruction examines bit corresponding to time when its write will occur
    - If zero, set to 1 and proceed with execution
    - If one, stall one cycle (repeat)
    - Shift register left at each clock cycle

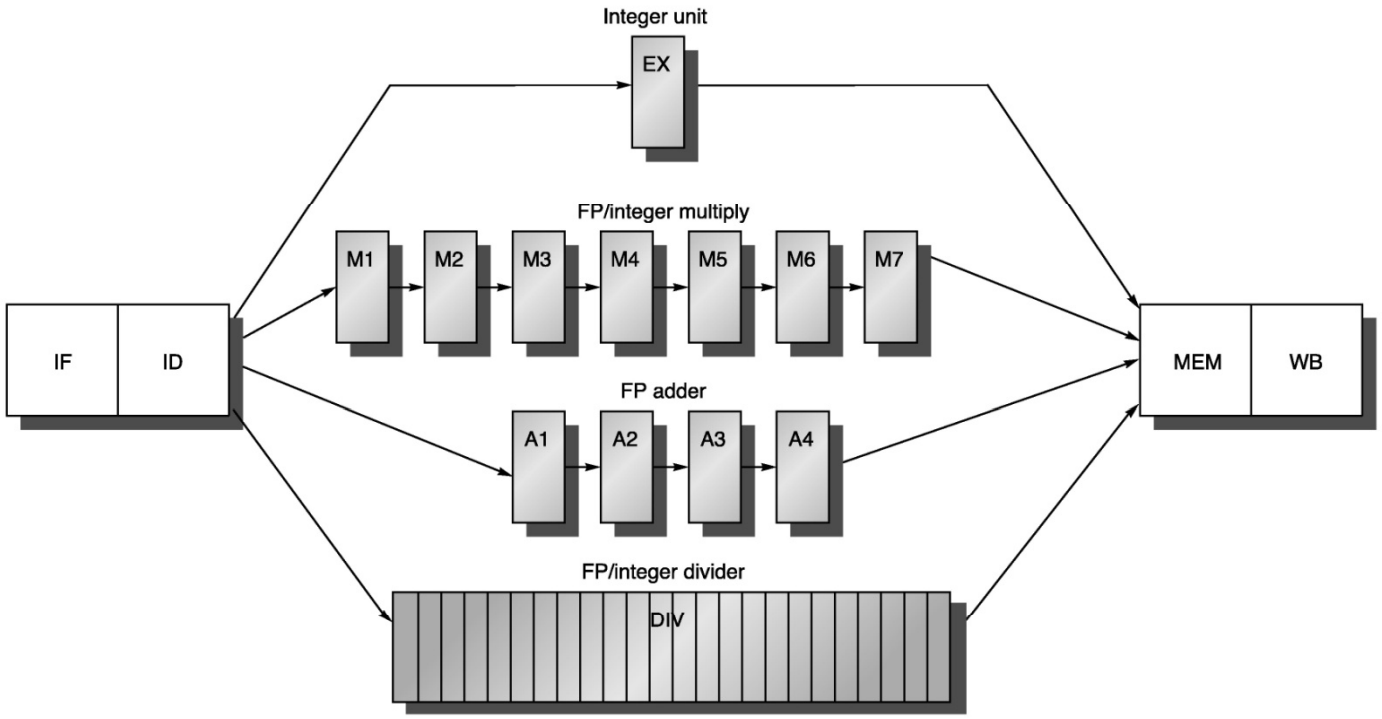


Cycle: n+1 n + pipeline depth

- Detect the hazard at beginning of stage with hazard
  - Complicates logic
  - Stall may need to be propagated backward



WAW Hazard



	Clock cycle number										
Instruction	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	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

WAW Hazard: Two Solutions

- A: Delay issue of **`L.D`** until **`ADD.D`** enters MEM stage
- B: Stamp out result of **`ADD.D`** by detecting hazard and preventing the write (which would just be overwritten anyway)
- Either case hazard can be detected in ID stage
  - A: Stall **`L.D`**
  - B: Turn **`ADD`** into No-Op
    - Difficulty in non-RISC: side-effects of auto-increment, etc
- Difficulty is in detecting that **`L.D`** may finish before **`ADD.D`**
  - Requires knowing details of pipeline (length, position of **`ADD.D`**)
- Simplest:
  - If instruction in ID wants to write same register as instruction already in pipeline, stall that instruction in ID (do not “issue”).

Hazards and Forwarding Longer Latency Pipelines

- Refer to Figure A.32 (page A-50)
  - Because divide unit is not fully pipelined, structural hazards can occur. Need to detect and stall issuing instructions
  - Because instructions have varying execution times, the number of register writes required in a cycle may be >1
  - WAW hazards are possible since instructions no longer reach WB in order (as they did in simple integer pipeline)
  - WAR hazards are not possible since register reads always occur in ID stage
  - Instructions can complete in a different order than they were issued, causing problems with exceptions
  - Because of longer latency, stalls for RAW hazards will be more frequent

Issue, Execution, Completion

- So far all instructions have had “in order issue”
- We also have “in order completion” except where we may have parallel stages (e.g. floating point and integer units with multicycle operations). Then we may have “out of order execution” and therefore “out of order completion”
- Split ID stage into two stages
  - Issue (decode instructions, check for hazards)
  - Read operands (wait until no hazards, read operands)

Static Scheduling

- Instructions are fetched and "issued" unless there's a structural hazard or a data dependence between the instruction and an instruction already in the pipeline which cannot be dealt with by forwarding
- If the hazard is unavoidable the pipeline is stalled and no new instructions are fetched or issued until the hazard is cleared
- Compilers can attempt to avoid the hazard by scheduling (re-arranging) instructions (while preserving correctness!) to avoid the hazard. This is called "static scheduling". Simple case is branch slot management. Another example we've seen is re-arranging **LOAD** instructions.

Dynamic Scheduling

- Hardware may re-arrange instructions to reduce stalls
- All instructions pass through issue stage in order (in order issue)
- They can be stalled or bypass each other in "read operands" stage and thus enter execution out of order
- New issues may arise…

```
DIV.D  F0,F2,F4
ADD.D  F10,F0,F8
SUB.D  F8,F8,F14
```

If out of order execution is permitted, **`ADD.D`** may be delayed waiting for **`DIV.D`** to complete the write to its operand F0. **`SUB.D`**, if permitted to execute, may write F8 before **`ADD.D`** is able to read it, causing a WAR hazard (not possible with in-order execution).

Q: Why do this in hardware (extra complexity, gates, die area)?
A: Software independence, binary compatibility even if microarchitecture changes

Dynamic Scheduling

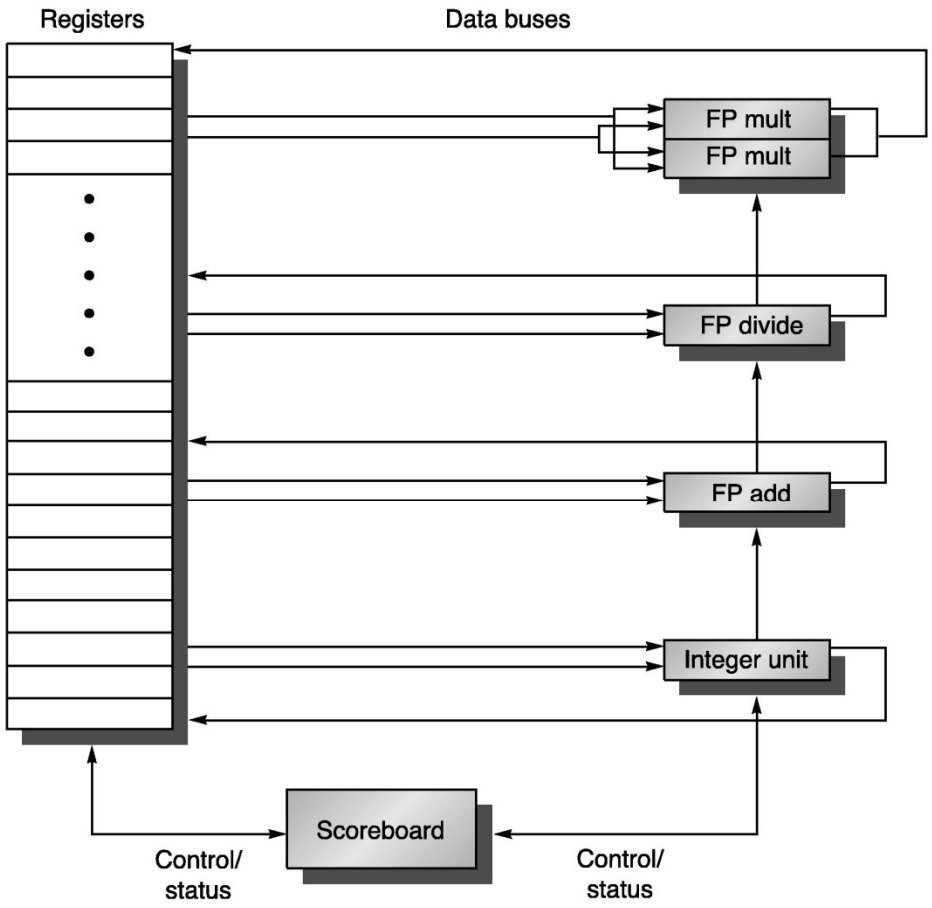
- Similarly, if destination of **SUB.D** were changed to F10, we could have WAW hazard:

```
DIV.D   F0,F2,F4
ADD.D  F10,F0,F8
SUB.D  F10,F8,F14
```

Dynamic Scheduling with a Scoreboard

- Technique first used on CDC 6600 (1964!)
- Each instruction goes through scorecard where record of dependences is constructed ("issue" stage)
- Scoreboard determines when instruction can read its operands and begin execution
- If instruction is unable to begin execution, scoreboard monitors hardware and determines when the instruction *can* execute
- Scoreboard also controls when instructions can write their result into the destination register
- Centralizes hazard detection and resolution in scoreboard

Scoreboard


Registers
Data buses
FP mult
FP mult
FP divide
FP add
Integer unit
Scoreboard
Control/
status
Control/
status

Dynamic Scheduling with a Scoreboard

- Issue
  - If FU (functional unit) is free and no other active instruction has the same destination register (WAW hazard), scoreboard "issues" instruction to the FU and updates internal data structures.
  - If structural hazard (FU not free) or WAW hazard, instruction stalls
  - Buffer between IF and Issue so that instructions may still be fetched
- Read operands
  - Scoreboard monitors availability of source operands
  - Source operand is available if no earlier issued active instruction will write the operand (RAW hazard)
  - Instructions may be sent to execution units out of order
- Execution
  - FU begins operation upon receipt of operands (notification by scoreboard)
  - When result is complete, FU notifies scoreboard
- Write result
  - Scoreboard checks for WAR hazards and stalls the completing instruction if necessary

Dynamic Scheduling

- WAR Hazard

```
DIV.D  F0,F2,F4
ADD.D  F10,F0,F8
SUB.D  F8,F8,F14
```

- **ADD** has source operand same as **DIV** destination (RAW hazard) so **SUB** may "pass" **ADD** in the pipeline
- **SUB** will stall in the Write Result stage until **ADD** reads its operands

- In general, scoreboard will prevent completion of an instruction in Write Result stage when
  - An instruction preceding it in issue hasn't read its operands and
  - One of those operands is result of completing instruction
- Because operands are read only when both are available, there is no forwarding in this scheme (but no severe penalty because registers are written when result is available – assuming no WAR hazard – rather than awaiting a future available write slot)

Scoreboard Data Structures

- Instruction Status
  - Indicates which of 4 steps the instruction is in
- Functional Unit Status
  - Indicates State of Each Functional Unit
    - Busy
    - Op (operation: e.g. ADD, SUB)
    - Fi (destination register)
    - Fj, Fk (source registers)
    - Qj, Qk (functional units producing Fj, Fk)
    - Rj, Rk (flags to indicate ready not not yet read)
      - Yes: “ready but not yet read”
      - No : “not ready (another instruction will be producing it)”
      - No & No: “both operands have been read” or “neither operand ready” – more on this later!
- Register Result Status
  - Indicates which FU will write the register (if an active instruction has it as its destination)

Scoreboard Data

		Instruction status			
Instruction		Issue	Read operands	Execution complete	Write result
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	
MUL.D	F0,F2,F4	√			
SUB.D	F8,F6,F2	√			
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2				

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Integer			Add	Divide			

Scoreboard Data

		Instruction status			
Instruction		Issue	Read operands	Execution complete	Write result
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	√
MUL.D	F0,F2,F4	√	√	√	
SUB.D	F8,F6,F2	√	√	√	√
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2	√	√	√	

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult 1			Add		Divide			

Figure A.53 Scoreboard tables just before the MUL.D goes to write result. The DIV.D has not yet read either of its operands, since it has a dependence on the result of the multiply. The ADD.D has read its operands and is in execution, although it was forced to wait until the SUB.D finished to get the functional unit. ADD.D cannot proceed to write result because of the WAR hazard on F6, which is used by the DIV.D. The Q fields are only relevant when a functional unit is waiting for another unit.

Scoreboard Data

Instruction		Instruction status			
		Issue	Read operands	Execution complete	Write result
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	√
MUL.D	F0,F2,F4	√	√	√	√
SUB.D	F8,F6,F2	√	√	√	√
DIV.D	F10,F0,F6	√	√	√	
ADD.D	F6,F8,F2	√	√	√	√

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Divide	Yes	Div	F10	F0	F6			No	No

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU						Divide			

Figure A.54 Scoreboard tables just before the `DIV.D` goes to write result. `ADD.D` was able to complete as soon as `DIV.D` passed through read operands and got a copy of F6. Only the `DIV.D` remains to finish.

Scoreboard Logic and Bookkeeping

Instruction status	Wait until	Bookkeeping
Issue	Not Busy [FU] and not Result [D]	Busy[FU]←yes; Op[FU]←op; Fi[FU]←D; Fj[FU]←S1; Fk[FU]←S2; Qj←Result[S1]; Qk← Result[S2]; Rj← not Qj; Rk← not Qk; Result[D]←FU;
Read operands	Rj and Rk	Rj← No; Rk← No; Qj←0; Qk←0
Execution complete	Functional unit done	
Write result	$\forall f$((Fj[f] ≠ Fi[FU] or Rj[f] = No) & (Fk[f] ≠ Fi[FU] or Rk[f] = No))	$\forall f$(if Qj[f]=FU then Rj[f]←Yes); $\forall f$(if Qk[f]=FU then Rk[f]←Yes); Result[Fi[FU]]← 0; Busy[FU]← No

Figure A.55 Required checks and bookkeeping actions for each step in instruction execution. FU stands for the functional unit used by the instruction, D is the destination register name, S1 and S2 are the source register names, and op is the operation to be done. To access the scoreboard entry named Fj for functional unit FU we use the notation Fj[FU]. Result[D] is the name of the functional unit that will write register D. The test on the write result case prevents the write when there is a WAR hazard, which exists if another instruction has this instruction's destination (Fi[FU]) as a source (Fj[f] or Fk[f]) and if some other instruction has written the register (Rj = Yes or Rk = Yes). The variable f is used for any functional unit.

Write Back

Want to inhibit write back if the completing instruction would write a register which a prior instruction has not yet read

But permit write back if no instruction is using that register as a source, or if a following instruction is using the register as a source

```
DIV.D  F0,F2,F4
ADD.D  F10,F0,F8
SUB.D  F8,F8,F14
```

When **SUB** completes, we want to inhibit its write of F8 because the **ADD** has not yet read F8

```
DIV.D  F0,F2,F4
ADD.D  F10,F0,F8
SUB.D  F8,F8,F14
```

When **DIV** completes, we want to permit its write of F0 because the **ADD** is waiting on F0

Write Back

Permit write back if no instruction is using that register as a source, or if a following instruction is using the register as a source

Write result	$\forall f$((Fj[f] ≠ Fi[FU] or Rj[f] = No) & (Fk[f] ≠ Fi[FU] or Rk[f] = No))	$\forall f$(if Qj[f]=FU then Rj[f]←Yes); $\forall f$(if Qk[f]=FU then Rk[f]←Yes); Result[Fi[FU]]← 0; Busy[FU]← No

Rj[f] = No where Fj[f] = Fi[FU], all others Fj[f] ≠ Fi[FU]

```
OP1   R1,R2,R3
OP2   R4,R1,R5
```

That is OP2's source (R1) is OP1's destination but R1 hasn't been read for OP2 yet. What would Rj[f], Rk[f] = {No,No} mean? Can't mean both OP2's source operands have been read, because scoreboard would have prevented them from being read if an earlier instruction (OP1) indicated intention to write one of them (Results[] vector). So {No,No} would mean both are unread (some prior instruction is generating R5).

```
OP1   R1,R2,R3
.
.
.
OP2   R4,R1,R5
```

No Fj[f] = Fi[FU]

That is OP2's hasn't yet entered issue, so it's not in execution and it's OK for OP1 to write R1.

Write Back

Want to inhibit write back if the completing instruction would write a register which a prior instruction has not yet read

Write result	$\forall f((Fj[f] \neq Fi[FU]$ or $Rj[f] = No)$ & $(Fk[f] \neq Fi[FU]$ or $Rk[f] = No))$	$\forall f$(if Qj[f]=FU then Rj[f]←Yes); $\forall f$(if Qk[f]=FU then Rk[f]←Yes); Result[Fi[FU]]← 0; Busy[FU]← No

Rj[f] = Yes where Fj[f] = Fi[FU], all others Fj[f] ≠ Fi[FU]

Want to inhibit OP2's write to R2 if OP1 hasn't yet read its R2 (because it's waiting on R3 to be produced by another earlier instruction)

```
OP1   R1,R2,R3
OP2   R2,R4,R5
```

Why would Rj[f] = Yes if OP1's R2 hasn't been read? If it were No, then Rk[f] (that is R3) is either No or Yes. If Rk[f] is Yes, then that would mean that R3 was ready but R2 wasn't. That could only be because a still earlier instruction was going to write R2. But that's impossible because OP2 is going to write R2 and WAW hazard detection would have prevented OP2 from issuing. If Rk[f] = No then it would have to mean that they've both been read and OP2 can proceed to write R2. (That is whatever prior instruction was writing R3 has completed and both R2 and R3 can be read for OP1

```
OP0   R2,R6,R7
OP1   R1,R2,R3
→ OP2   R2,R4,R5
```

WAW detection prevents OP2 from issuing

Pitfall

- Unexpected execution sequences may cause unexpected hazards
  - Why expect to encounter WAW hazard? No sane compiler would generate code to write to a register twice without an intervening read.
  - What if first write is in delay slot of a taken branch when the scheduler thought it would be untaken. Example:

```
        BNEZ   R1,Label
        DIV.D  F0,F2,F4  ;moved into delay slot
                         ;from fall through
            …
            …
Label:  L.D  F0, 0(R5)
```

If branch is taken, **L.D** could reach WB before **DIV.D** completes, causing a WAW hazard.

Pitfall

- Extensive pipelining can impact other aspects of a design, leading to overall worse performance
  - VAX 8600
    - 80 ns cycle time at introduction
  - VAX 8650
    - 55 ns cycle time
  - VAX 8700
    - Simpler pipeline, smaller CPU
    - 45 ns cycle time
  - VAX 8650 CPI advantage of ~ 20%
  - VAX 8700 clock rate ~ 20% faster
  - VAX 8700 achieved same performance with much less hardware

Pitfall

- Evaluating a compile time scheduler on the basis of un-optimized code
  - Un-optimized code is much easier to schedule than optimized ("tight") code: more likely to contain redundant load, stores, other operations that might be eliminated by optimizer