

Review

Instruction Set Architecture

Instruction Set

§2.1 Introduction

- The repertoire of instructions of a computer
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32×32 -bit register file
 - Use for frequently accessed data
 - Numbered 0 to 31
 - 32-bit data called a “word”
- Assembler names
 - \$t0, \$t1, ..., \$t9 for temporary values
 - \$s0, \$s1, ..., \$s7 for saved variables
- *Design Principle 2: Smaller is faster*
 - c.f. main memory: millions of locations



Register Operand Example

- C code:
 $f = (g + h) - (i + j);$
 - f, ..., j in \$s0, ..., \$s4
- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```



Memory Operand Example 1

- C code:

g = h + A[8];

- g in \$s1, h in \$s2, base address of A in \$s3

- Compiled MIPS code:

- Index 8 requires offset of 32

- 4 bytes per word

```
lw    $t0, 32($s3)      # load word  
add $s1, $s2, $t0
```

offset

base register



Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!



Immediate Operands

- Constant data specified in an instruction
`addi $s3, $s3, 4`
- No subtract immediate instruction
 - Just use a negative constant
`addi $s2, $s1, -1`
- Design Principle 3: Make the common case fast*
 - Small constants are common
 - Immediate operand avoids a load instruction



MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
 - op: operation code (opcode)
 - rs: first source register number
 - rt: second source register number
 - rd: destination register number
 - shamt: shift amount (00000 for now)
 - funct: function code (extends opcode)



R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add \$t0, \$s1, \$s2

special	\$s1	\$s2	\$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

0000001000110010010000000100000₂ = 02324020₁₆



MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
 - rt: destination or source register number
 - Constant: -2^{15} to $+2^{15} - 1$
 - Address: offset added to base address in rs
- *Design Principle 4: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible



Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- **beq rs, rt, L1**
 - if ($rs == rt$) branch to instruction labeled L1;
- **bne rs, rt, L1**
 - if ($rs != rt$) branch to instruction labeled L1;
- **j L1**
 - unconditional jump to instruction labeled L1



Compiling If Statements

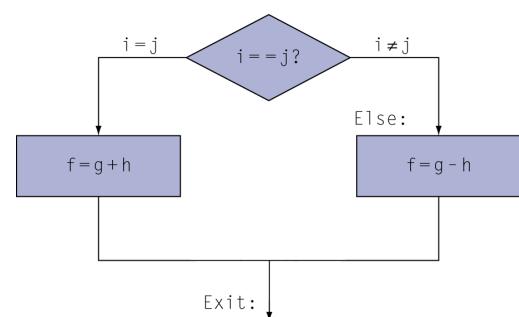
- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

 - f, g, ... in \$s0, \$s1, ...

- Compiled MIPS code:

```
bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit
Else: sub $s0, $s1, $s2
Exit:
```



Assembler calculates addresses



Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

- i in \$s3, k in \$s5, address of save in \$s6

- Compiled MIPS code:

```
Loop: sll    $t1, $s3, 2  
       add   $t1, $t1, $s6  
       lw    $t0, 0($t1)  
       bne  $t0, $s5, Exit  
       addi $s3, $s3, 1  
       j     Loop
```

```
Exit: ...
```



Procedure Calling

- Steps required

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call



Register Usage

- \$a0 – \$a3: arguments (reg's 4 – 7)
- \$v0, \$v1: result values (reg's 2 and 3)
- \$t0 – \$t9: temporaries
 - Can be overwritten by callee
- \$s0 – \$s7: saved
 - Must be saved/restored by callee
- \$gp: global pointer for static data (reg 28)
- \$sp: stack pointer (reg 29)
- \$fp: frame pointer (reg 30)
- \$ra: return address (reg 31)



Procedure Call Instructions

- Procedure call: jump and link

```
jal ProcedureLabel
```

 - Address of following instruction put in \$ra
 - Jumps to target address
- Procedure return: jump register

```
jr $ra
```

 - Copies \$ra to program counter
 - Can also be used for computed jumps
 - e.g., for case/switch statements



Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
    f = (g + h) - (i + j);
    return f;
}
```

- Arguments g, ..., j in \$a0, ..., \$a3
- f in \$s0 (hence, need to save \$s0 on stack)
- Result in \$v0



Leaf Procedure Example

- MIPS code:

leaf_example:	
addi \$sp, \$sp, -4	Save \$s0 on stack
sw \$s0, 0(\$sp)	
add \$t0, \$a0, \$a1	Procedure body
add \$t1, \$a2, \$a3	
sub \$s0, \$t0, \$t1	
add \$v0, \$s0, \$zero	Result
lw \$s0, 0(\$sp)	
addi \$sp, \$sp, 4	Restore \$s0
jr \$ra	Return



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
 - Its return address
 - Any arguments and temporaries needed after the call
- Restore from the stack after the call



Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
    if (n < 1) return f;
    else return n * fact(n - 1);
}
```

- Argument n in \$a0
- Result in \$v0



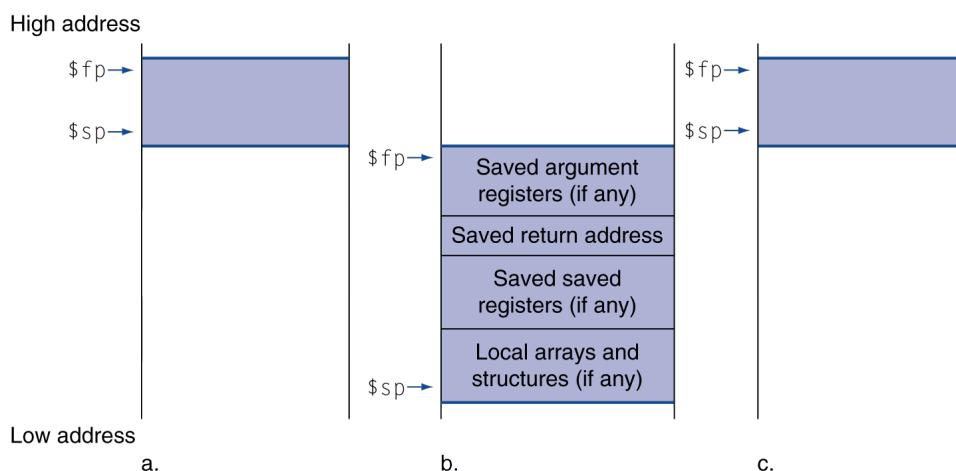
Non-Leaf Procedure Example

MIPS code:

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1        # test for n < 1
    beq $t0, $zero, L1
    addi $v0, $zero, 1      # if so, result is 1
    addi $sp, $sp, 8        # pop 2 items from stack
    jr   $ra                # and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       # and return address
    addi $sp, $sp, 8        # pop 2 items from stack
    mul $v0, $a0, $v0       # multiply to get result
    jr   $ra                # and return
```



Local Data on the Stack



- Local data allocated by callee
 - e.g., C automatic variables
- Procedure frame (activation record)
 - Used by some compilers to manage stack storage



Branch Addressing

- Branch instructions specify
 - Opcode, two registers, target address
- Most branch targets are near branch
 - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC-relative addressing
 - Target address = $PC + \text{offset} \times 4$
 - PC already incremented by 4 by this time



Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
 - Encode full address in instruction

op	address
6 bits	26 bits

- (Pseudo)Direct jump addressing
 - Target address = $PC_{31\dots28} : (\text{address} \times 4)$



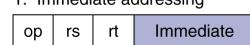
Target Addressing Example

- Loop code from earlier example
 - Assume Loop at location 80000

Loop:	sll	\$t1, \$s3, 2	80000	0 0 19 9 4 0
	add	\$t1, \$t1, \$s6	80004	0 9 22 9 0 32
	lw	\$t0, 0(\$t1)	80008	35 9 8 0 0 0
	bne	\$t0, \$s5, Exit	80012	5 8 21 2 0 0
	addi	\$s3, \$s3, 1	80016	8 19 19 1 0 0
	j	Loop	80020	2 0 0 0 0 20000
Exit:	...		80024	

Addressing Mode Summary

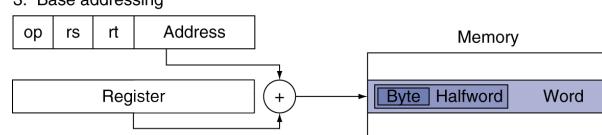
1. Immediate addressing



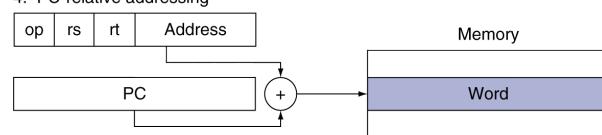
2. Register addressing



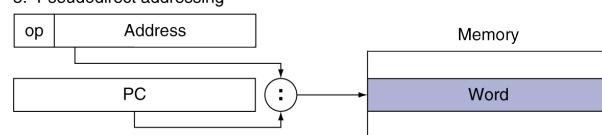
3. Base addressing



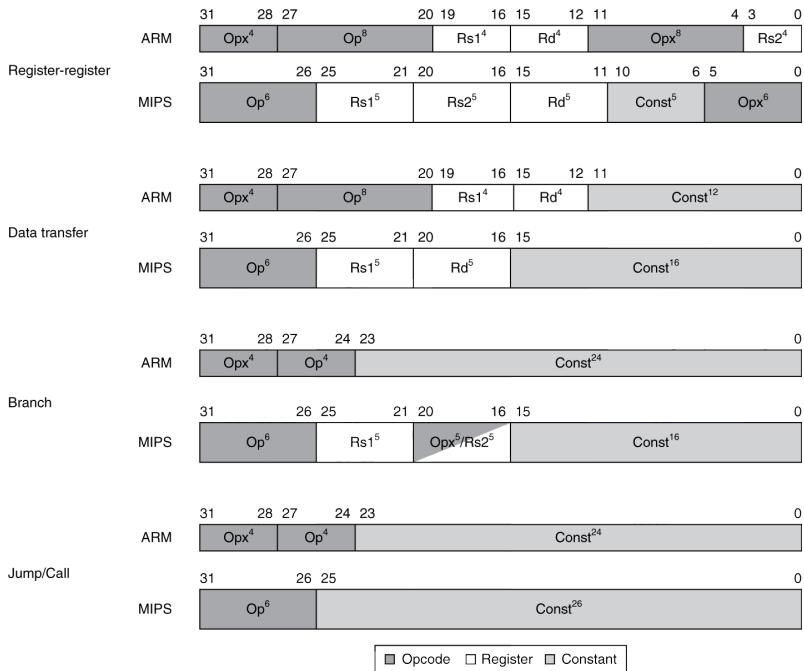
4. PC-relative addressing



5. Pseudodirect addressing



Instruction Encoding



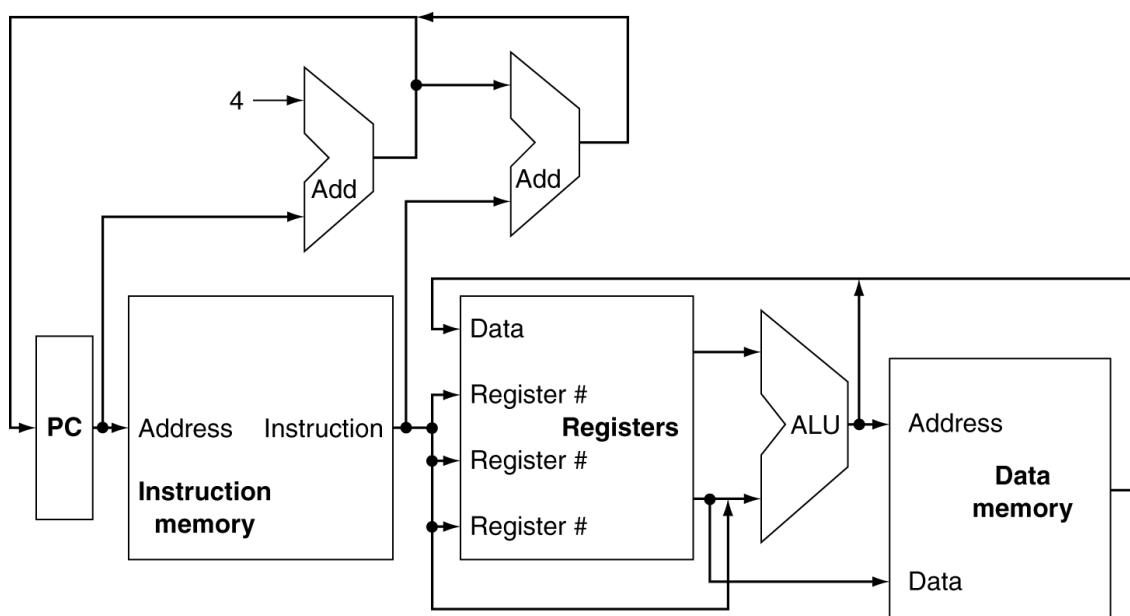
Review

The Processor

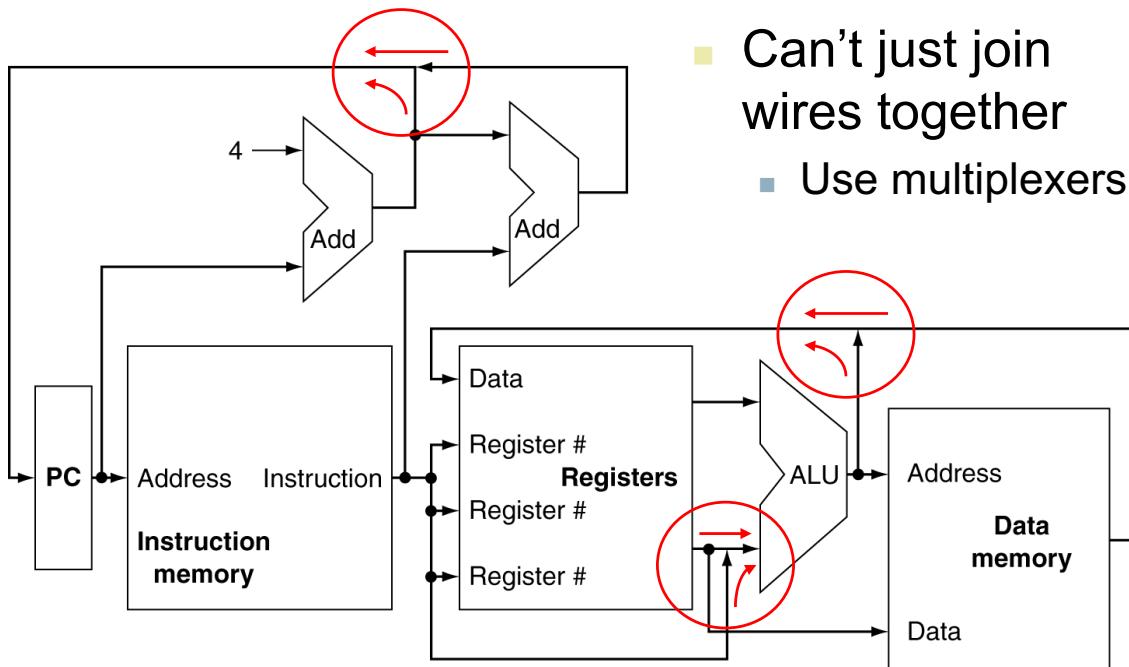
Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
 - Use ALU to calculate
 - Arithmetic result
 - Memory address for load/store
 - Branch target address
 - Access data memory for load/store
 - PC ← target address or PC + 4

CPU Overview

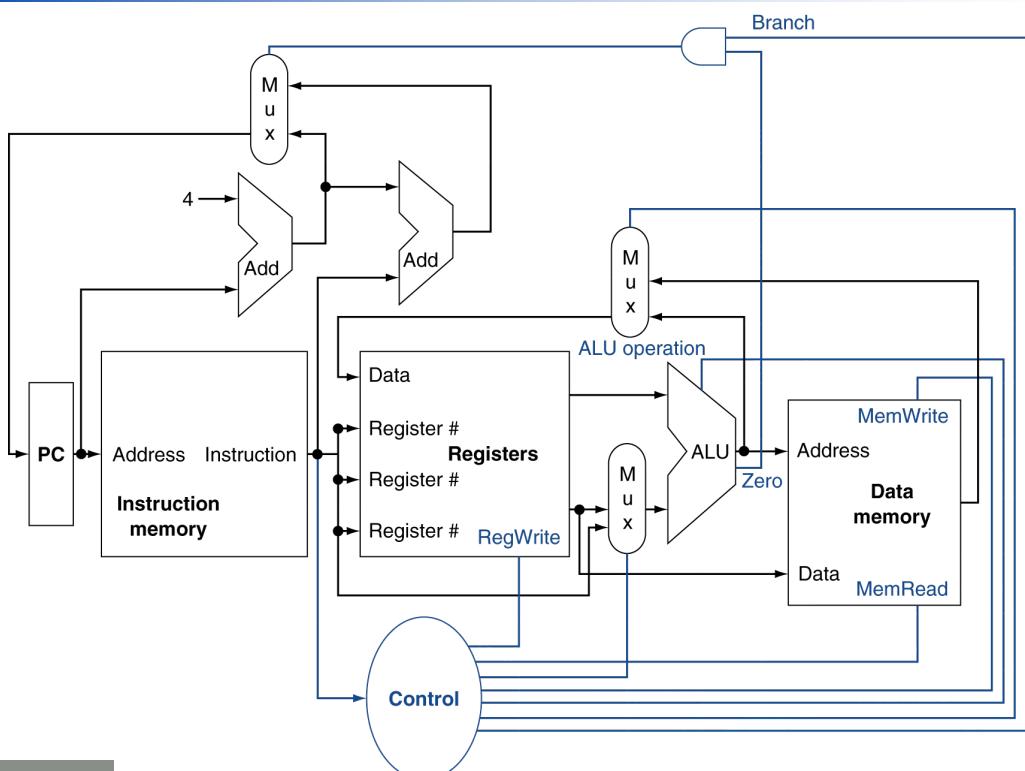


Multiplexers



- Can't just join wires together
 - Use multiplexers

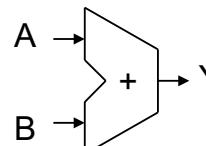
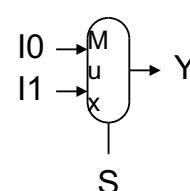
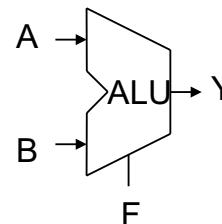
Control



Logic Design Basics

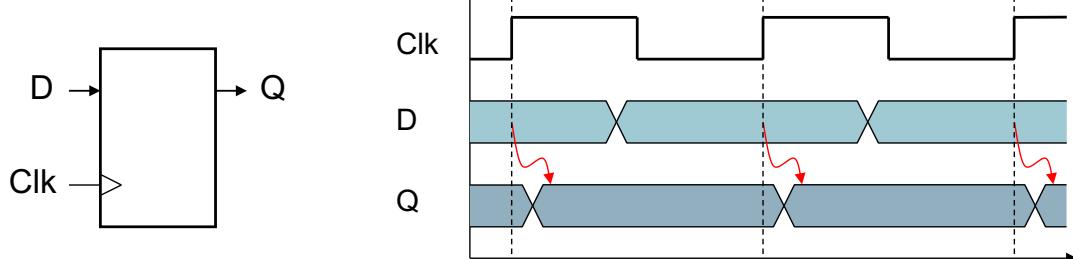
- Information encoded in binary
 - Low voltage = 0, High voltage = 1
 - One wire per bit
 - Multi-bit data encoded on multi-wire buses
- Combinational element
 - Operate on data
 - Output is a function of input
- State (sequential) elements
 - Store information

Combinational Elements

- AND-gate
 - $Y = A \& B$
- Adder
 - $Y = A + B$
- Multiplexer
 - $Y = S ? I_1 : I_0$
- Arithmetic/Logic Unit
 - $Y = F(A, B)$

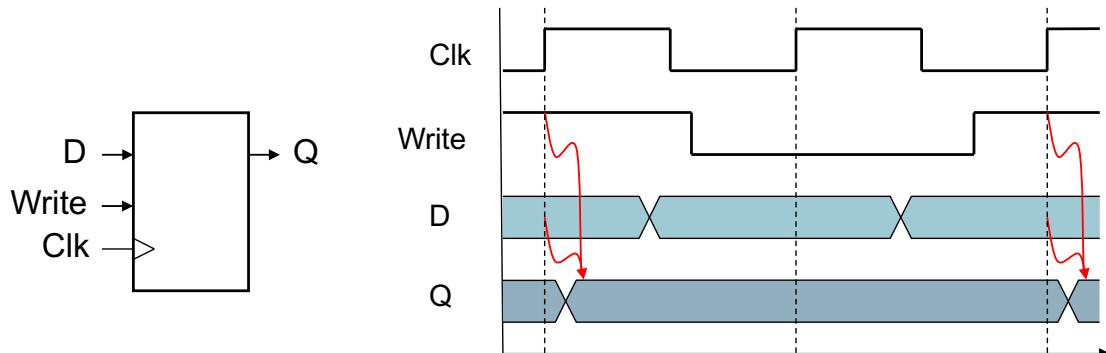
Sequential Elements

- Register: stores data in a circuit
 - Uses a clock signal to determine when to update the stored value
 - Edge-triggered: update when Clk changes from 0 to 1



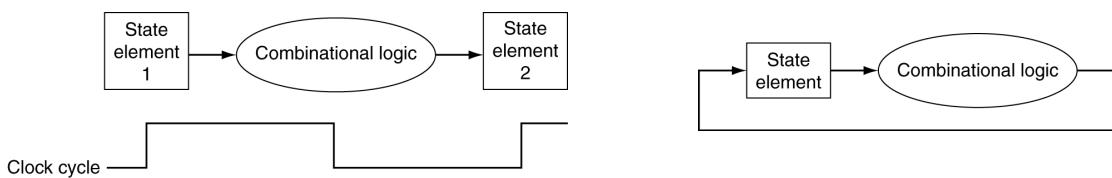
Sequential Elements

- Register with write control
 - Only updates on clock edge when write control input is 1
 - Used when stored value is required later



Clocking Methodology

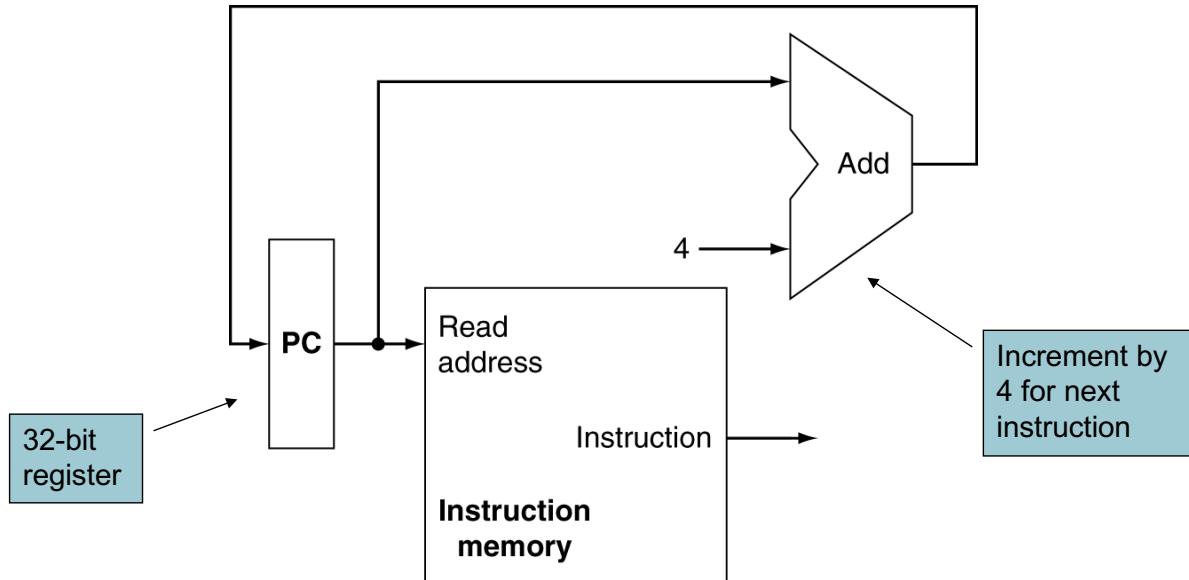
- Combinational logic transforms data during clock cycles
 - Between clock edges
 - Input from state elements, output to state element
 - Longest delay determines clock period



Building a Datapath

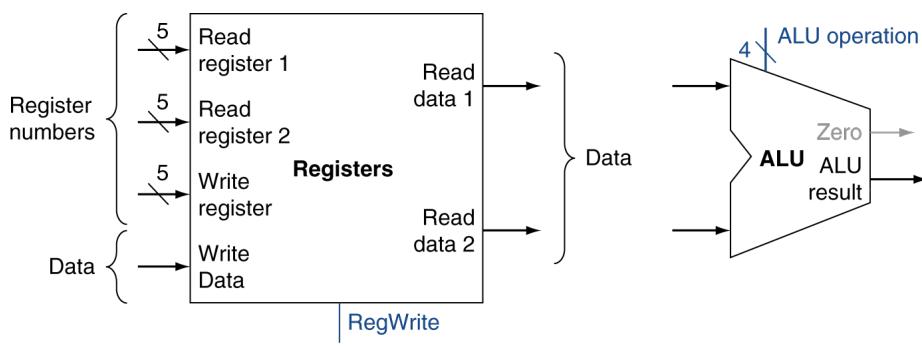
- Datapath
 - Elements that process data and addresses in the CPU
 - Registers, ALUs, mux's, memories, ...
- We will build a MIPS datapath incrementally
 - Refining the overview design

Instruction Fetch



R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



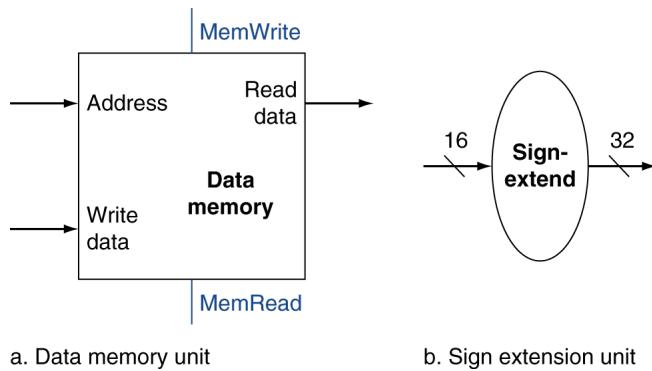
a. Registers

b. ALU



Load/Store Instructions

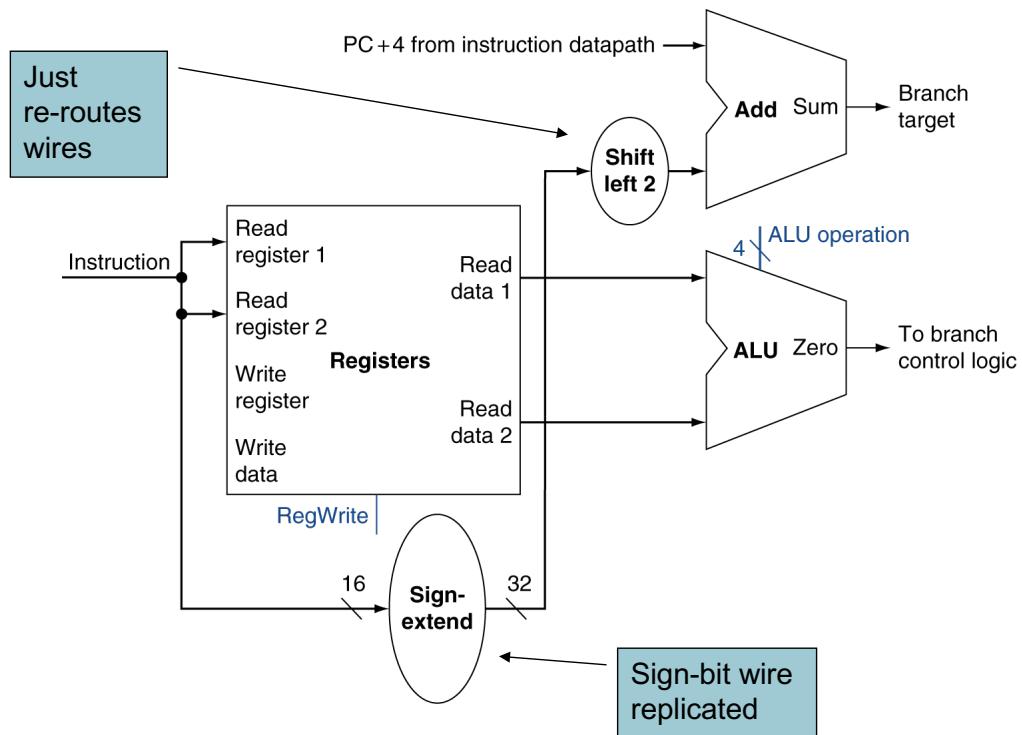
- Read register operands
- Calculate address using 16-bit offset
 - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



Branch Instructions

- Read register operands
- Compare operands
 - Use ALU, subtract and check Zero output
- Calculate target address
 - Sign-extend displacement
 - Shift left 2 places (word displacement)
 - Add to PC + 4
 - Already calculated by instruction fetch

Branch Instructions

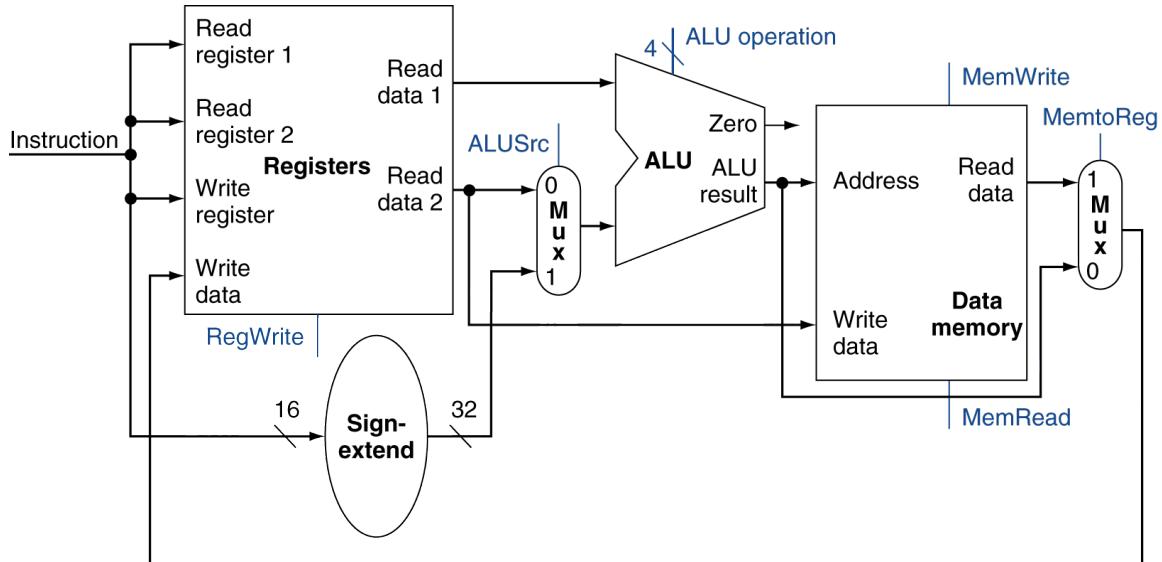


Chapter 4 — The Processor — 43

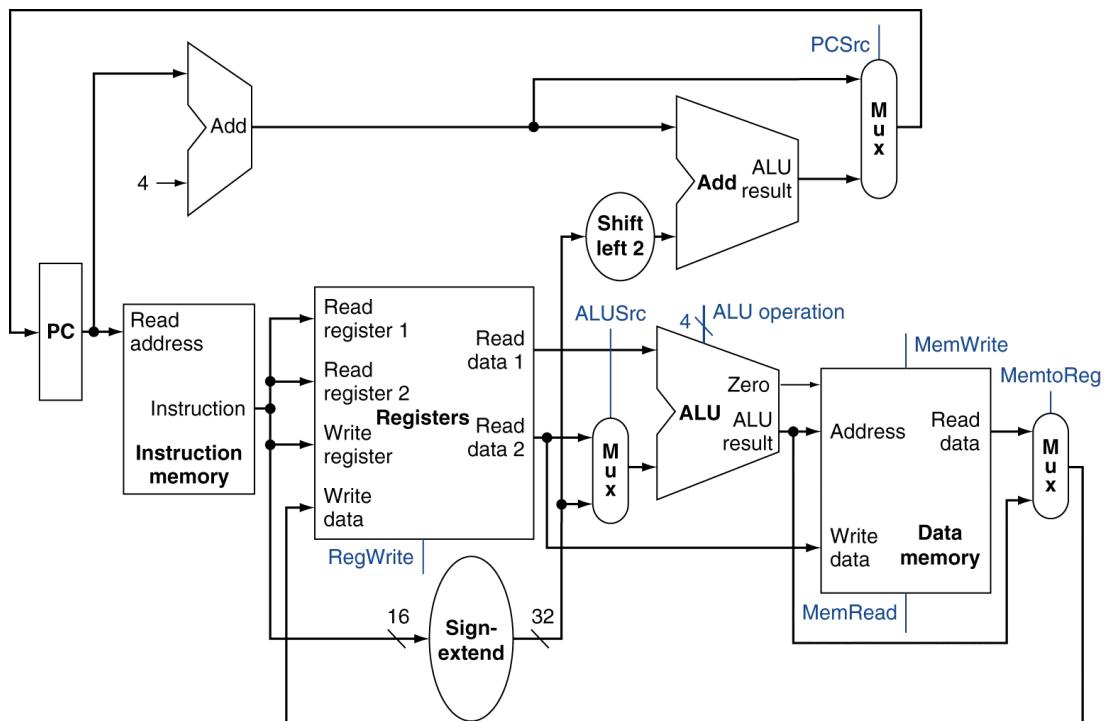
Composing the Elements

- First-cut data path does an instruction in one clock cycle
 - Each datapath element can only do one function at a time
 - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



ALU Control

- ALU used for
 - Load/Store: F = add
 - Branch: F = subtract
 - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

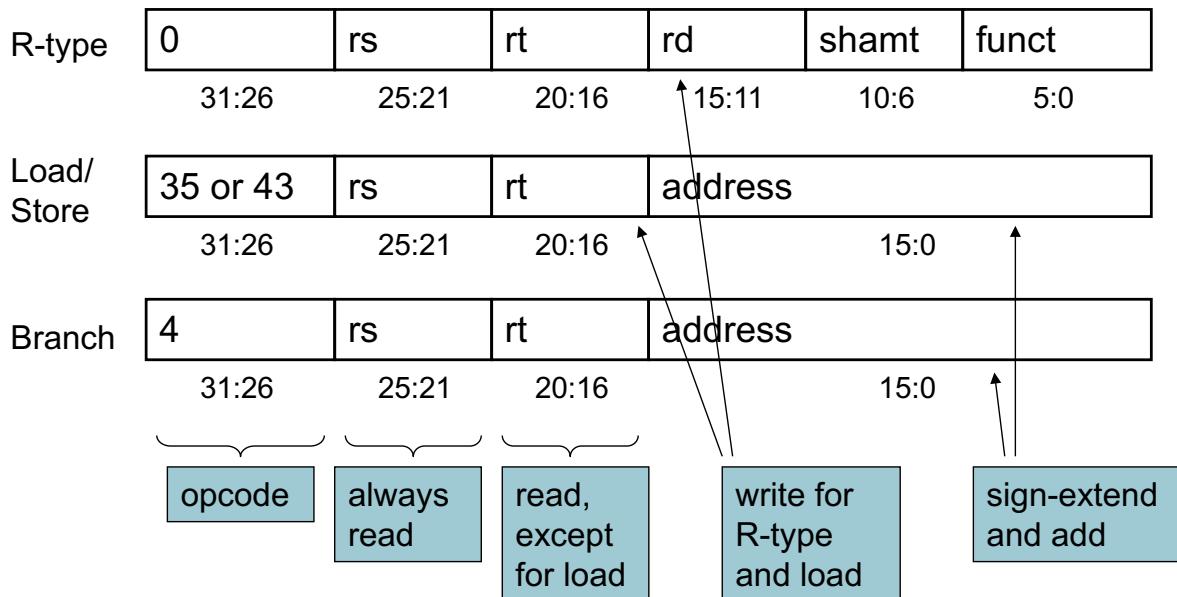
ALU Control

- Assume 2-bit ALUOp derived from opcode
 - Combinational logic derives ALU control

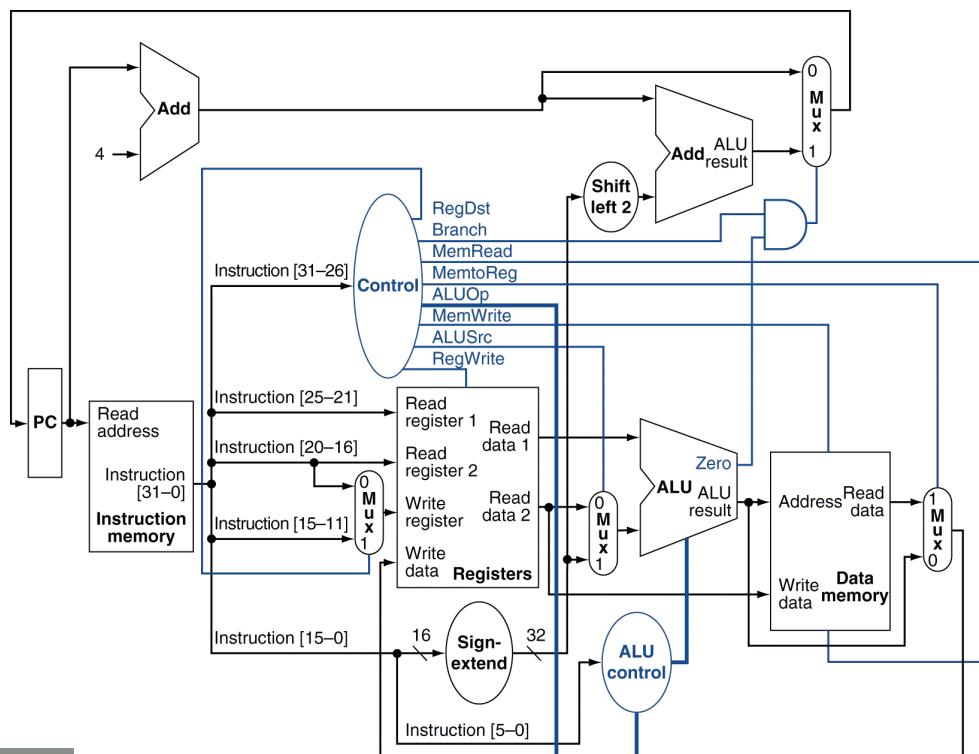
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

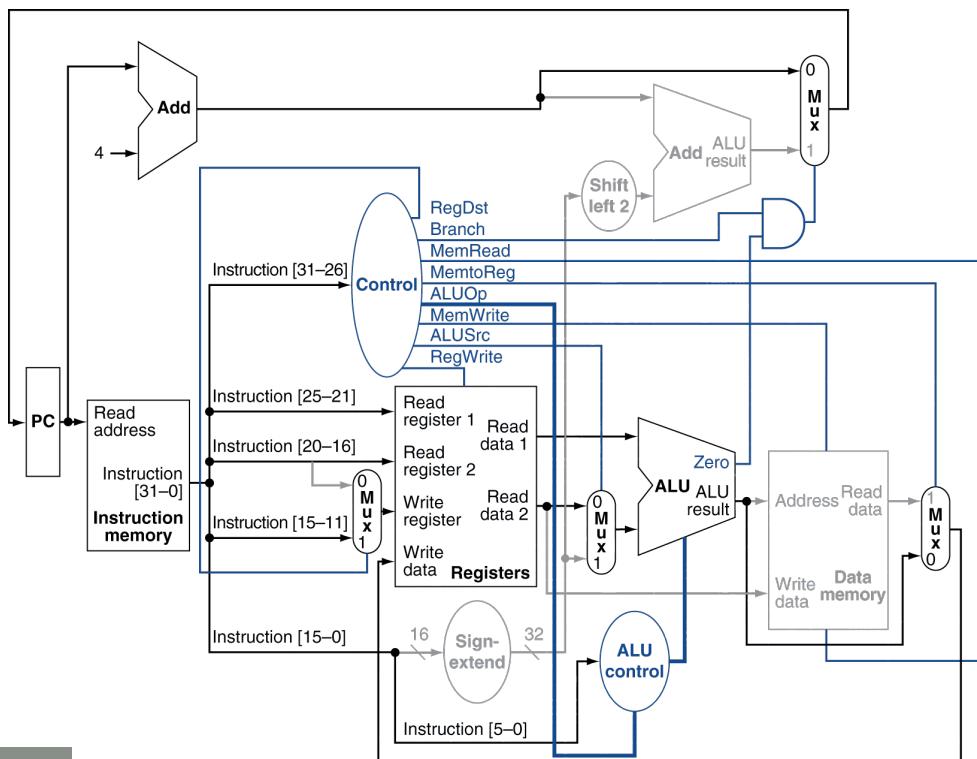
- Control signals derived from instruction



Datapath With Control



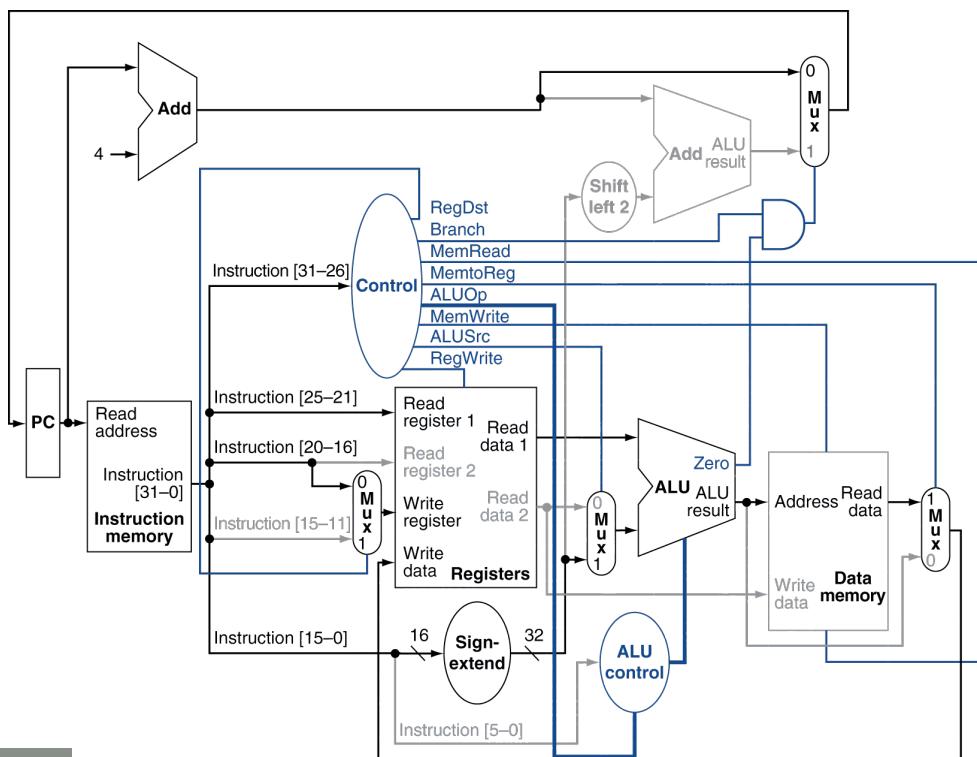
R-Type Instruction



Chapter 4 — The Processor — 51



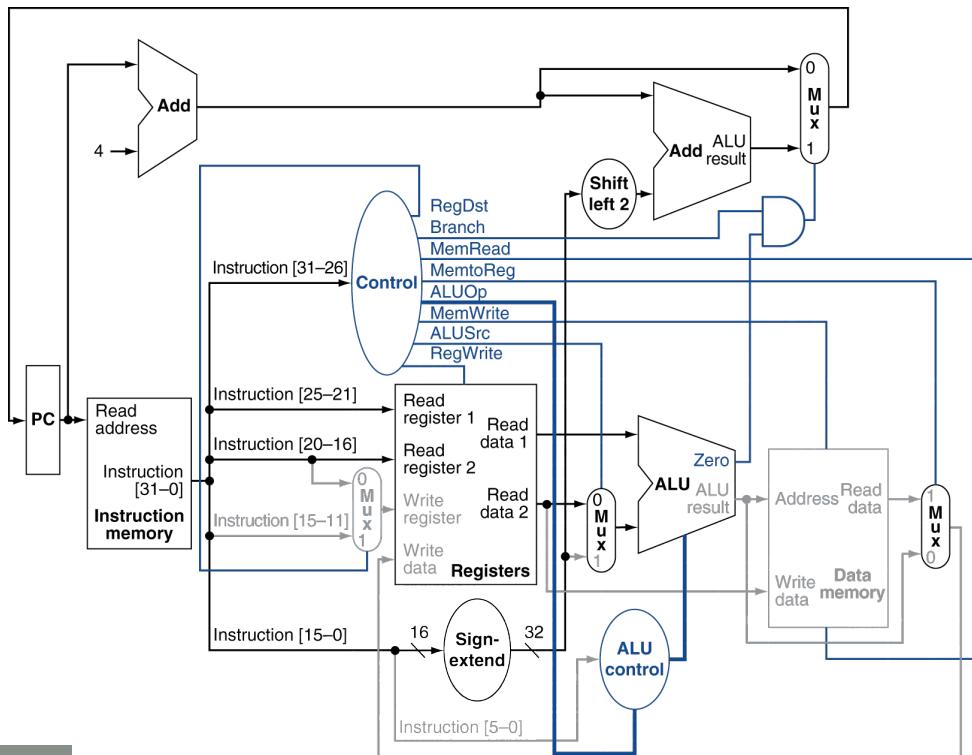
Load Instruction



Chapter 4 — The Processor — 52



Branch-on-Equal Instruction



Chapter 4 — The Processor — 53



Implementing Jumps

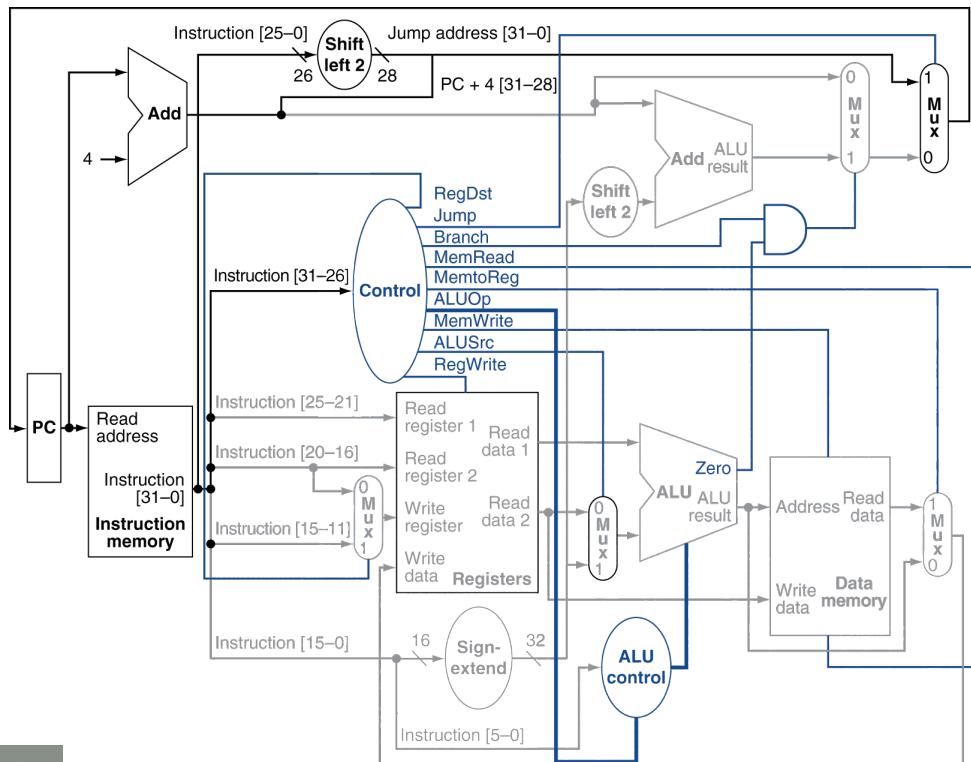


- Jump uses word address
- Update PC with concatenation of
 - Top 4 bits of old PC
 - 26-bit jump address
 - 00
- Need an extra control signal decoded from opcode



Chapter 4 — The Processor — 54

Datapath With Jumps Added



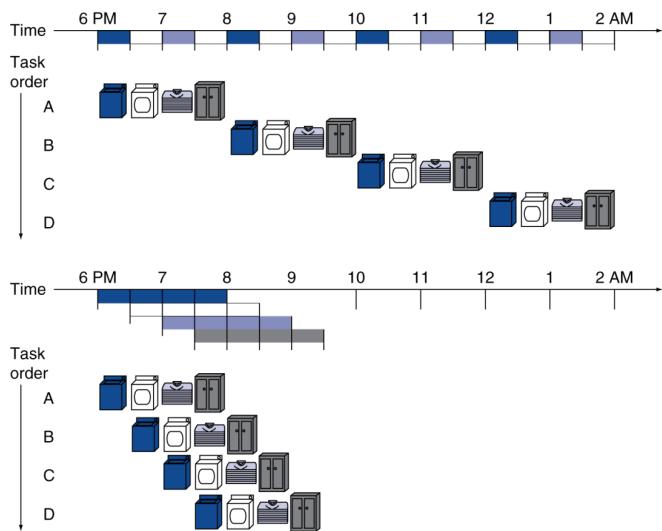
Chapter 4 — The Processor — 55

Performance Issues

- Longest delay determines clock period
 - Critical path: load instruction
 - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
 - Making the common case fast
- We will improve performance by pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
 - Parallelism improves performance



- Four loads:
 - Speedup = $8/3.5 = 2.3$
- Non-stop:
 - Speedup = $2n/0.5n + 1.5 \approx 4$
 - number of stages

MIPS Pipeline

- Five stages, one step per stage
 1. IF: Instruction fetch from memory
 2. ID: Instruction decode & register read
 3. EX: Execute operation or calculate address
 4. MEM: Access memory operand
 5. WB: Write result back to register

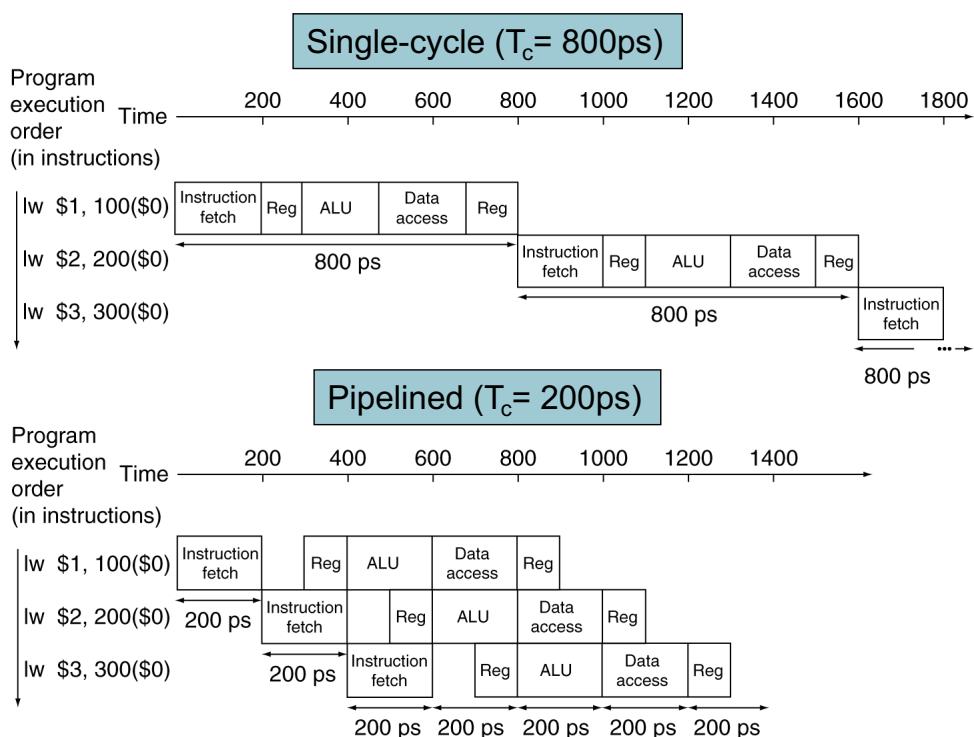
Pipeline Performance

- Assume time for stages is
 - 100ps for register read or write
 - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps



Pipeline Performance



Pipeline Speedup

- If all stages are balanced
 - i.e., all take the same time
 - Time between instructions_{pipelined}
= Time between instructions_{nonpipelined}
Number of stages
- If not balanced, speedup is less
- Speedup due to increased throughput
 - Latency (time for each instruction) does not decrease



Pipelining and ISA Design

- MIPS ISA designed for pipelining
 - All instructions are 32-bits
 - Easier to fetch and decode in one cycle
 - c.f. x86: 1- to 17-byte instructions
 - Few and regular instruction formats
 - Can decode and read registers in one step
 - Load/store addressing
 - Can calculate address in 3rd stage, access memory in 4th stage
 - Alignment of memory operands
 - Memory access takes only one cycle



Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
 - A required resource is busy
- Data hazard
 - Need to wait for previous instruction to complete its data read/write
- Control hazard
 - Deciding on control action depends on previous instruction



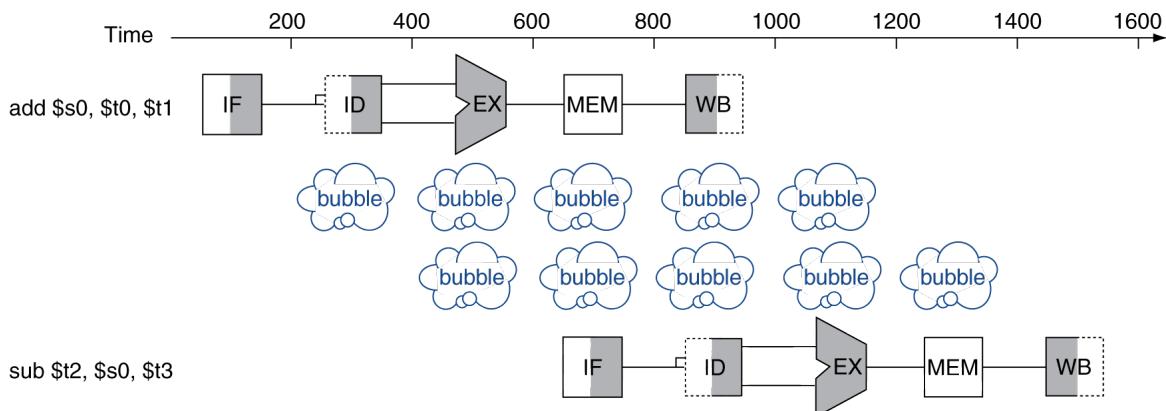
Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
 - Load/store requires data access
 - Instruction fetch would have to *stall* for that cycle
 - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
 - Or separate instruction/data caches



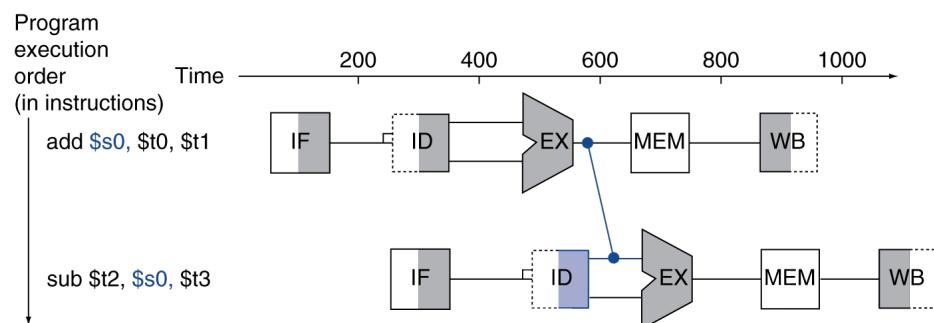
Data Hazards

- An instruction depends on completion of data access by a previous instruction
 - add \$s0, \$t0, \$t1
 - sub \$t2, \$s0, \$t3



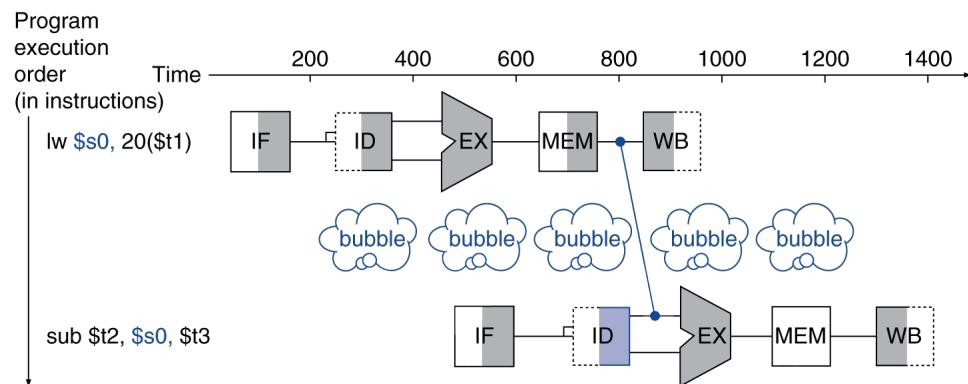
Forwarding (aka Bypassing)

- Use result when it is computed
 - Don't wait for it to be stored in a register
 - Requires extra connections in the datapath



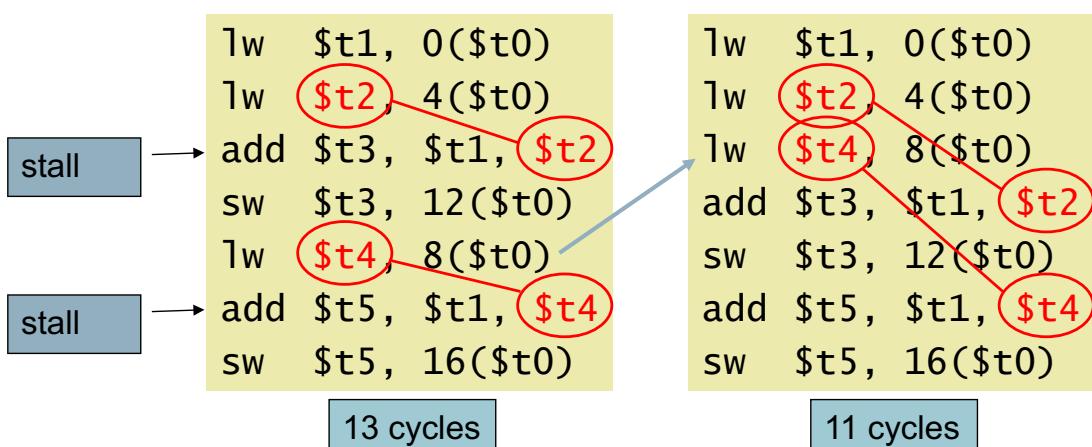
Load-Use Data Hazard

- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for $A = B + E; C = B + F;$



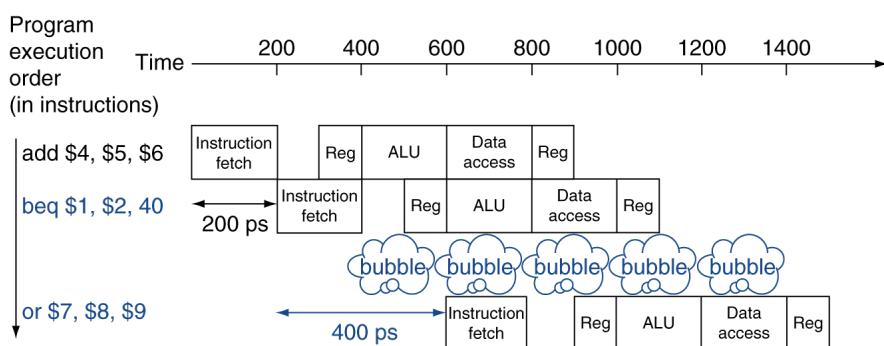
Control Hazards

- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction
 - Still working on ID stage of branch
- In MIPS pipeline
 - Need to compare registers and compute target early in the pipeline
 - Add hardware to do it in ID stage



Stall on Branch

- Wait until branch outcome determined before fetching next instruction

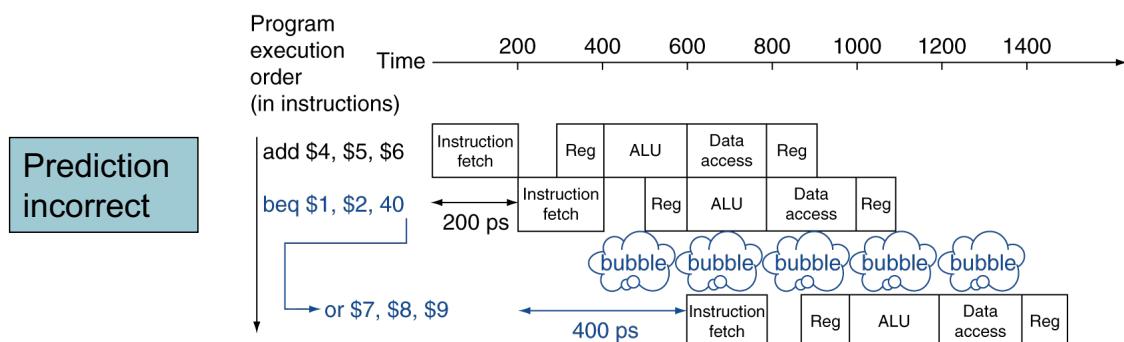
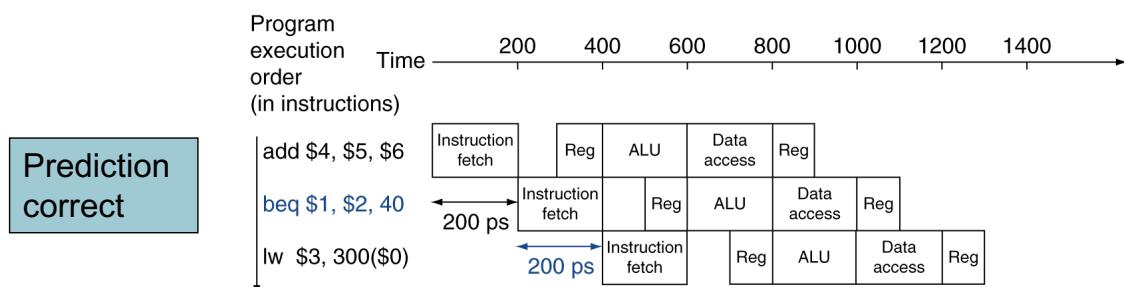


Branch Prediction

- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch
 - Only stall if prediction is wrong
- In MIPS pipeline
 - Can predict branches not taken
 - Fetch instruction after branch, with no delay



MIPS with Predict Not Taken



More-Realistic Branch Prediction

- Static branch prediction
 - Based on typical branch behavior
 - Example: loop and if-statement branches
 - Predict backward branches taken
 - Predict forward branches not taken
- Dynamic branch prediction
 - Hardware measures actual branch behavior
 - e.g., record recent history of each branch
 - Assume future behavior will continue the trend
 - When wrong, stall while re-fetching, and update history



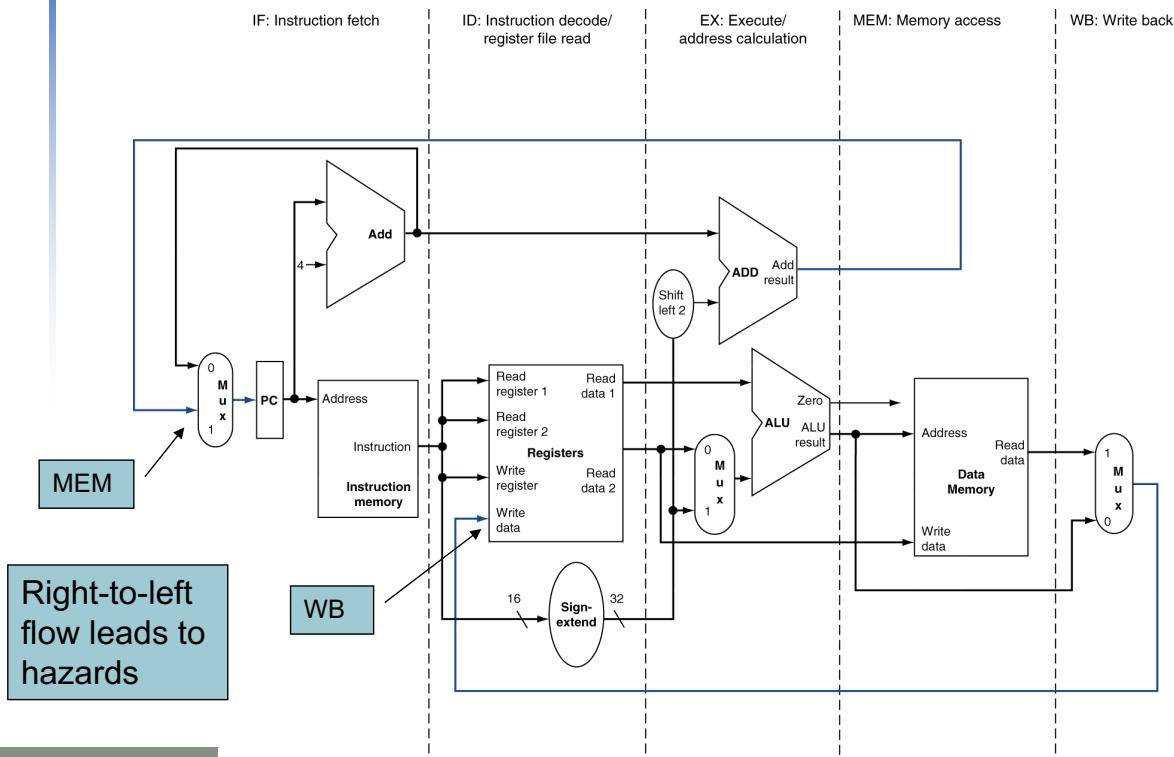
Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
 - Executes multiple instructions in parallel
 - Each instruction has the same latency
- Subject to hazards
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

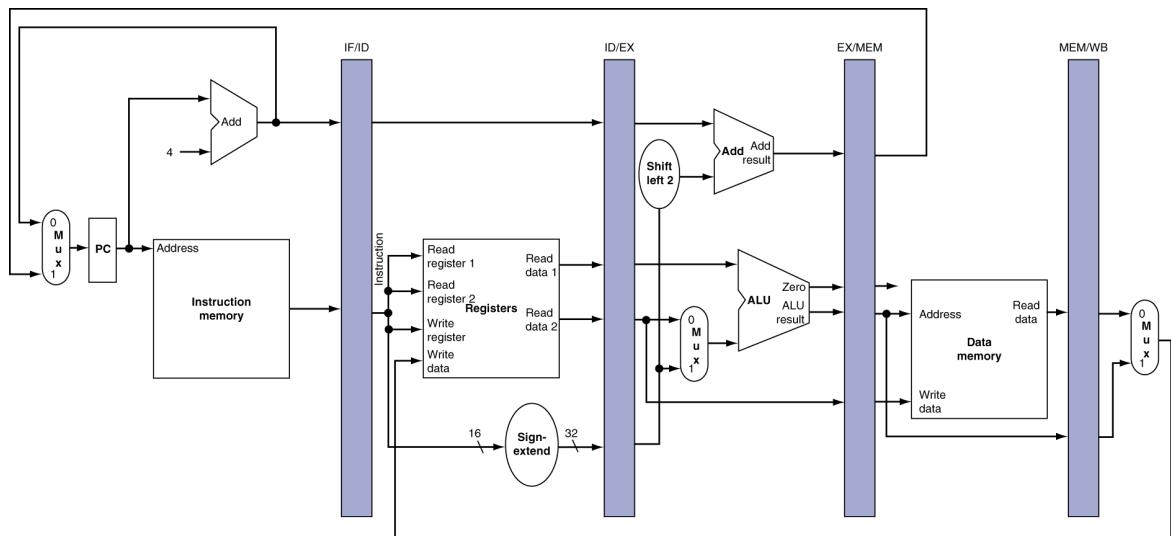


MIPS Pipelined Datapath



Pipeline registers

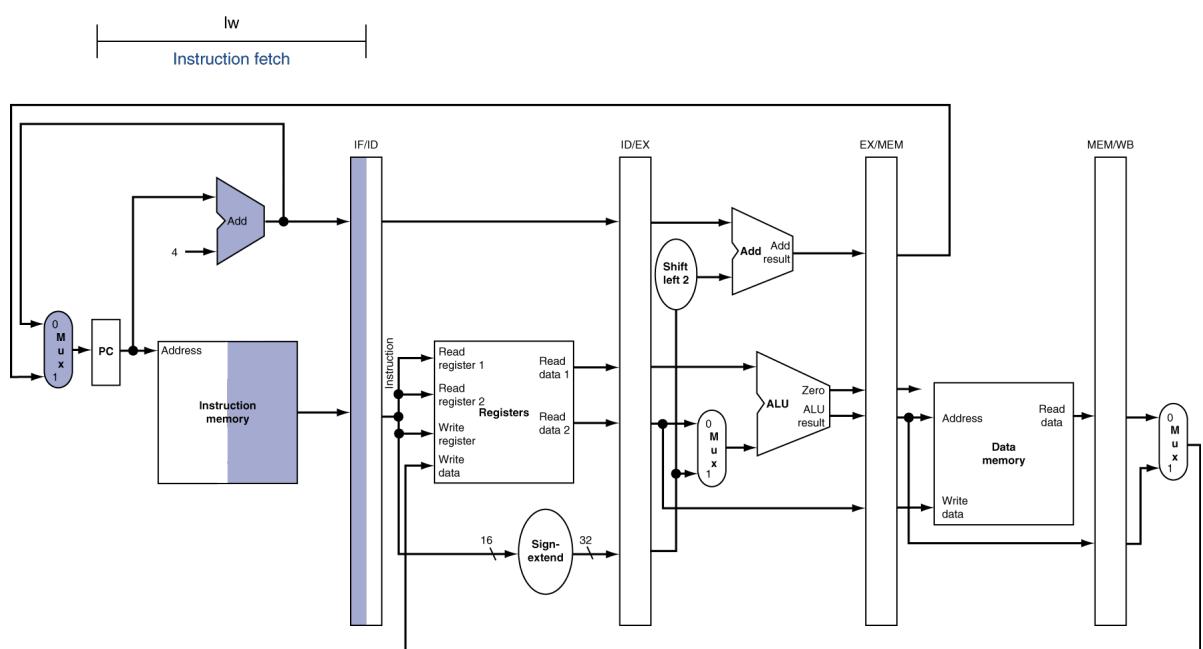
- Need registers between stages
 - To hold information produced in previous cycle



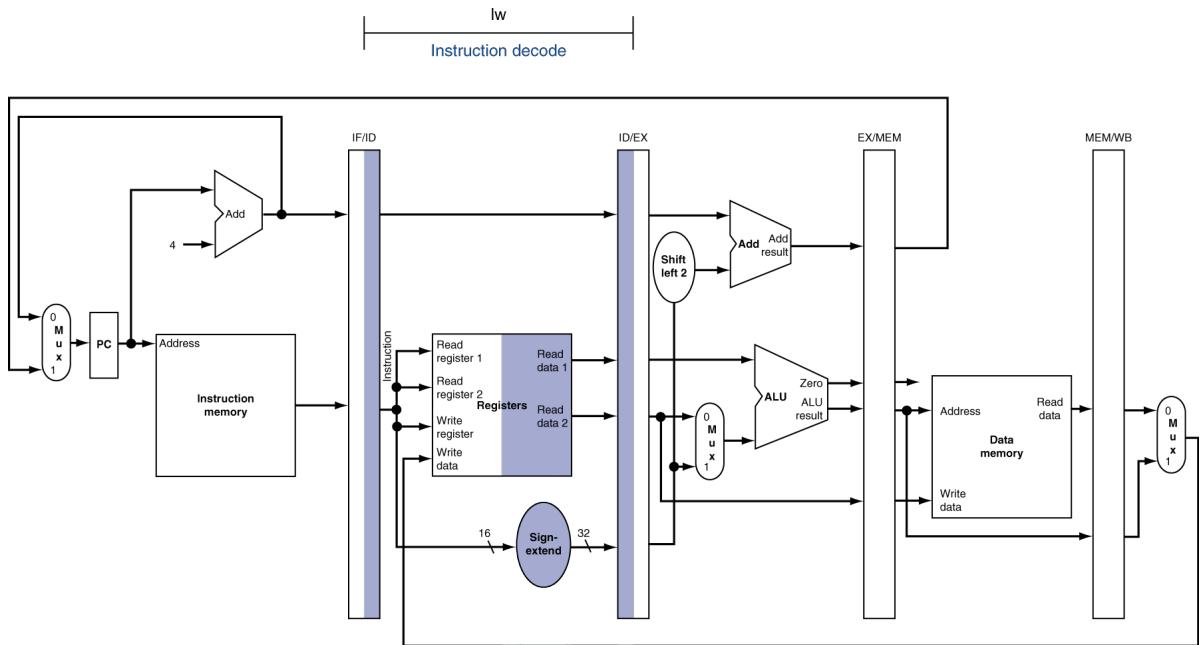
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
 - “Single-clock-cycle” pipeline diagram
 - Shows pipeline usage in a single cycle
 - Highlight resources used
 - c.f. “multi-clock-cycle” diagram
 - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load & store

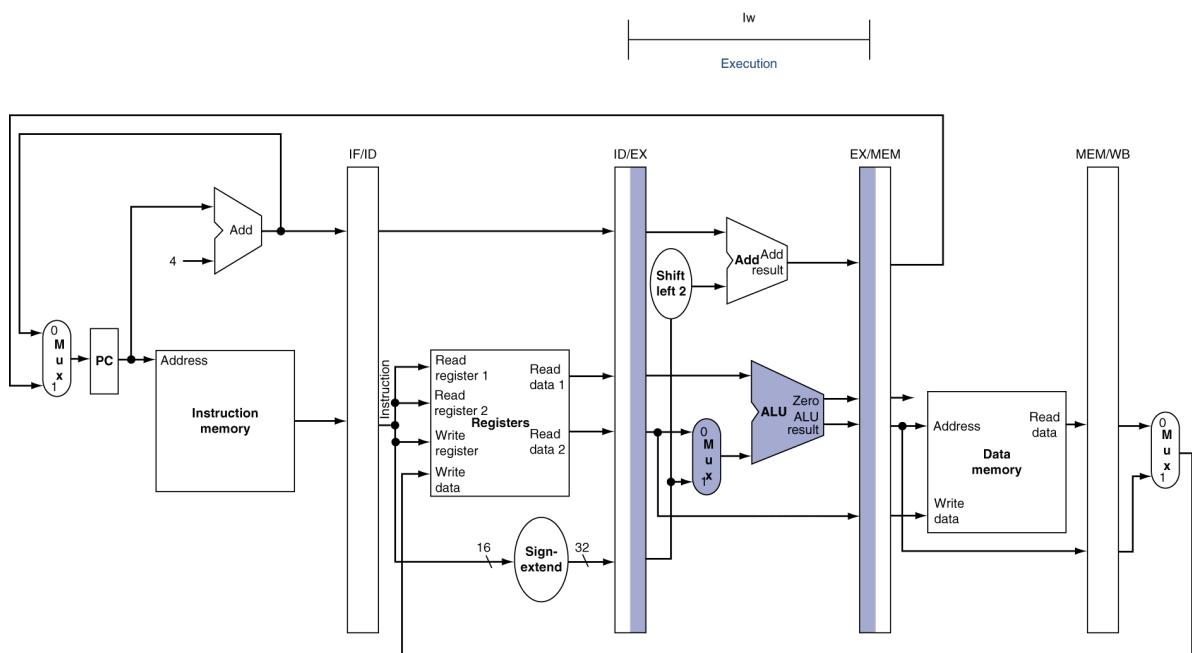
IF for Load, Store, ...



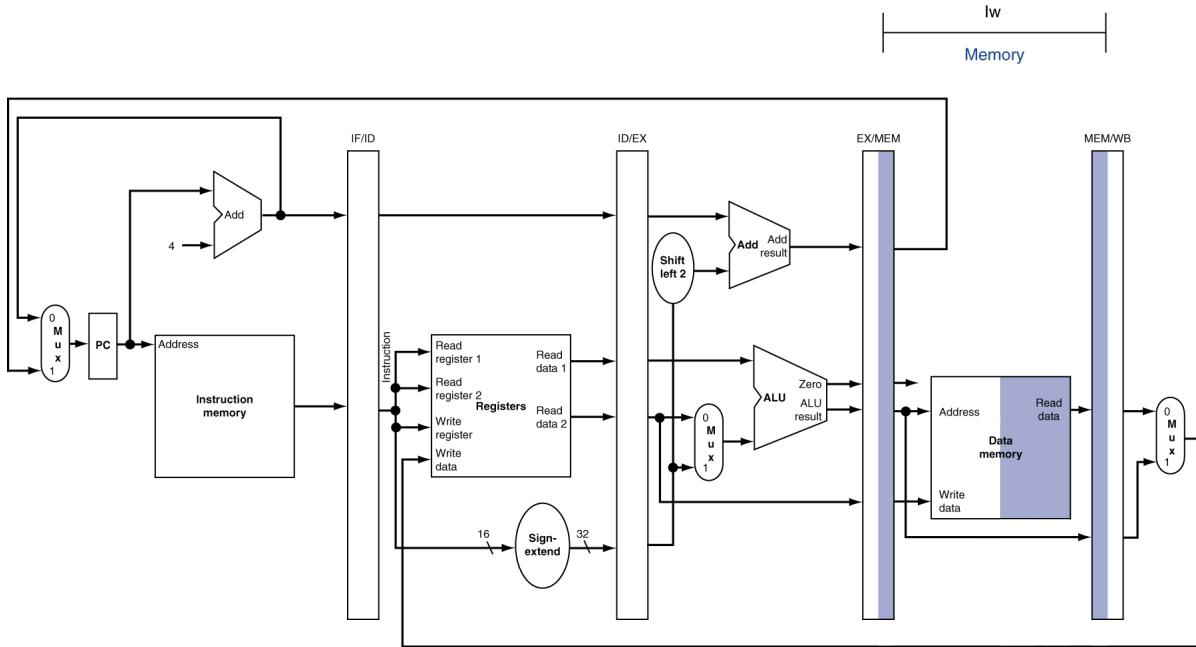
ID for Load, Store, ...



EX for Load

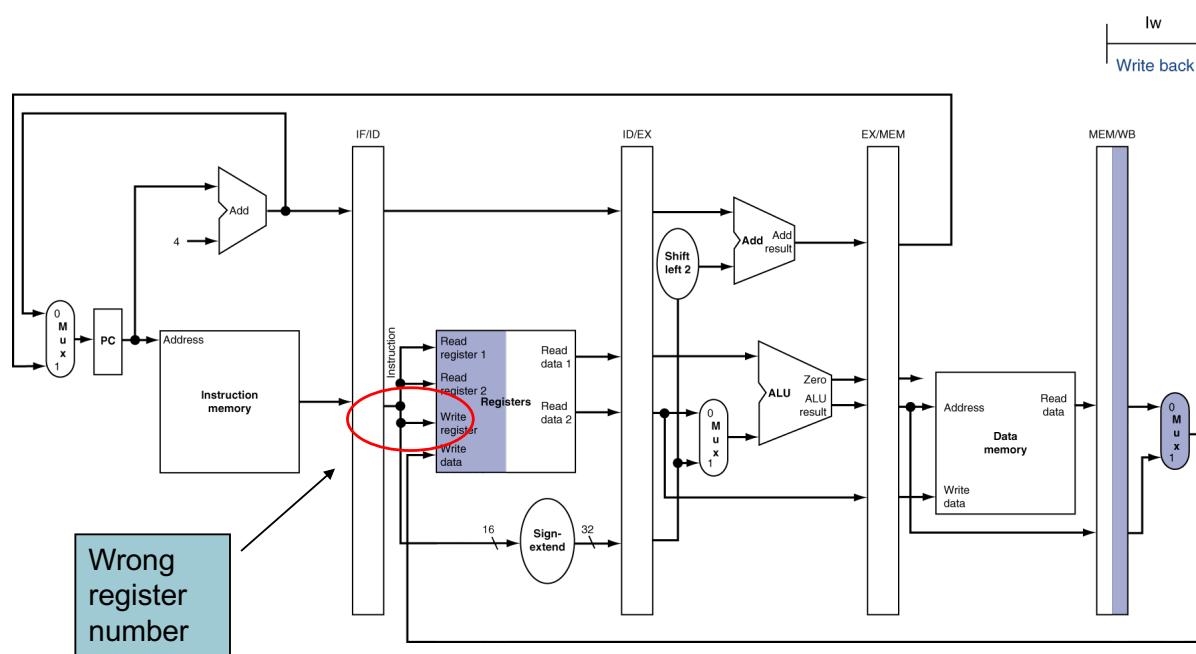


MEM for Load



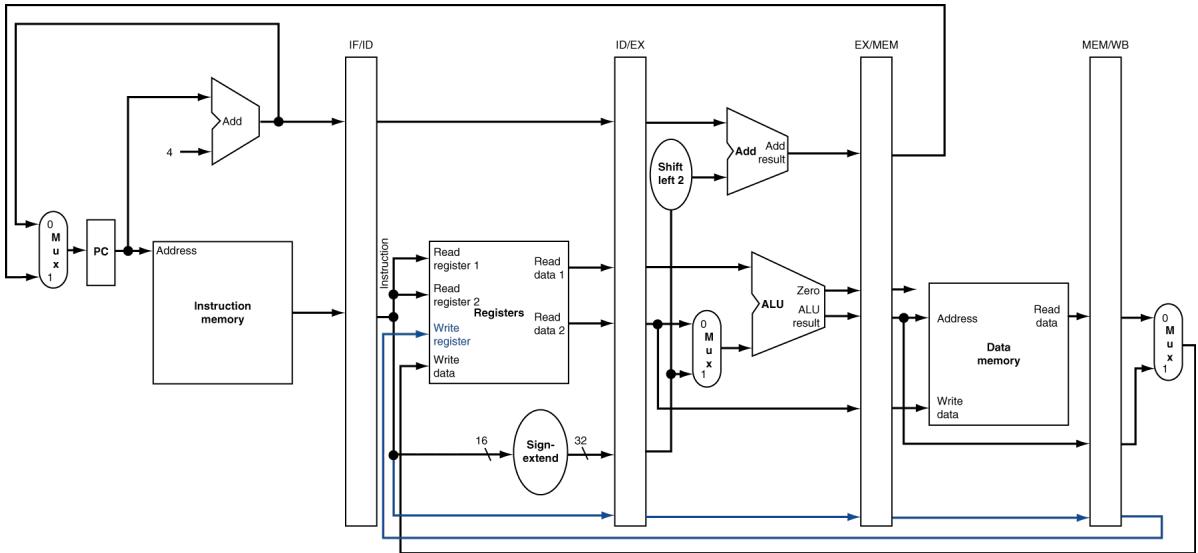
Chapter 4 — The Processor — 81

WB for Load

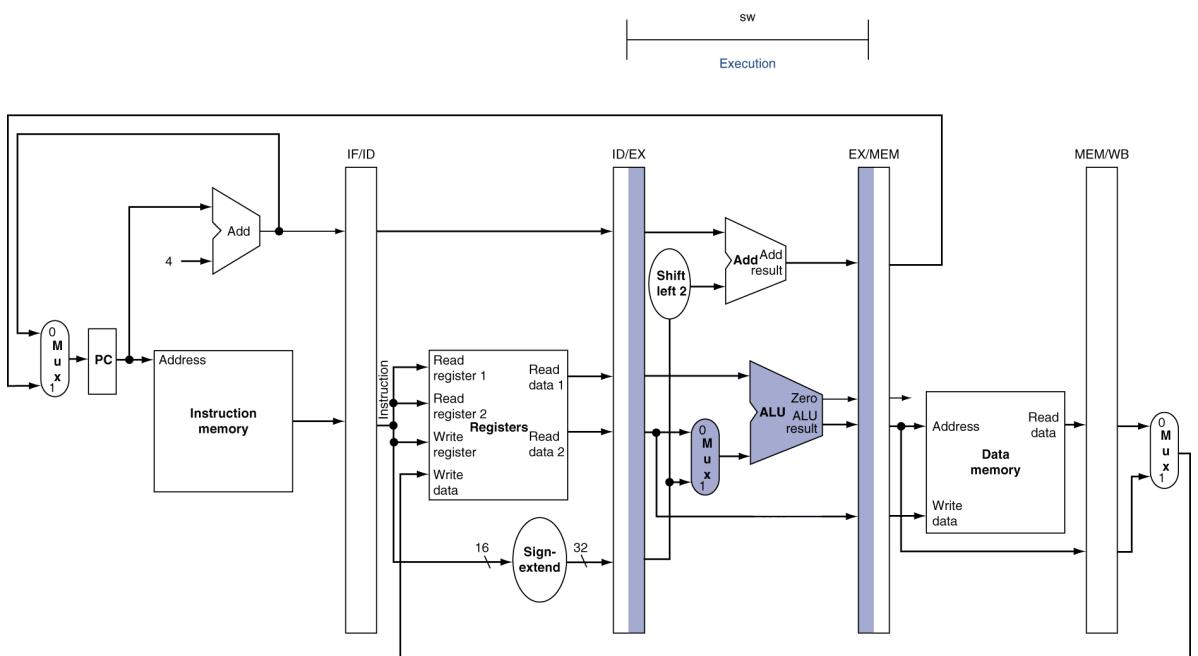


Chapter 4 — The Processor — 82

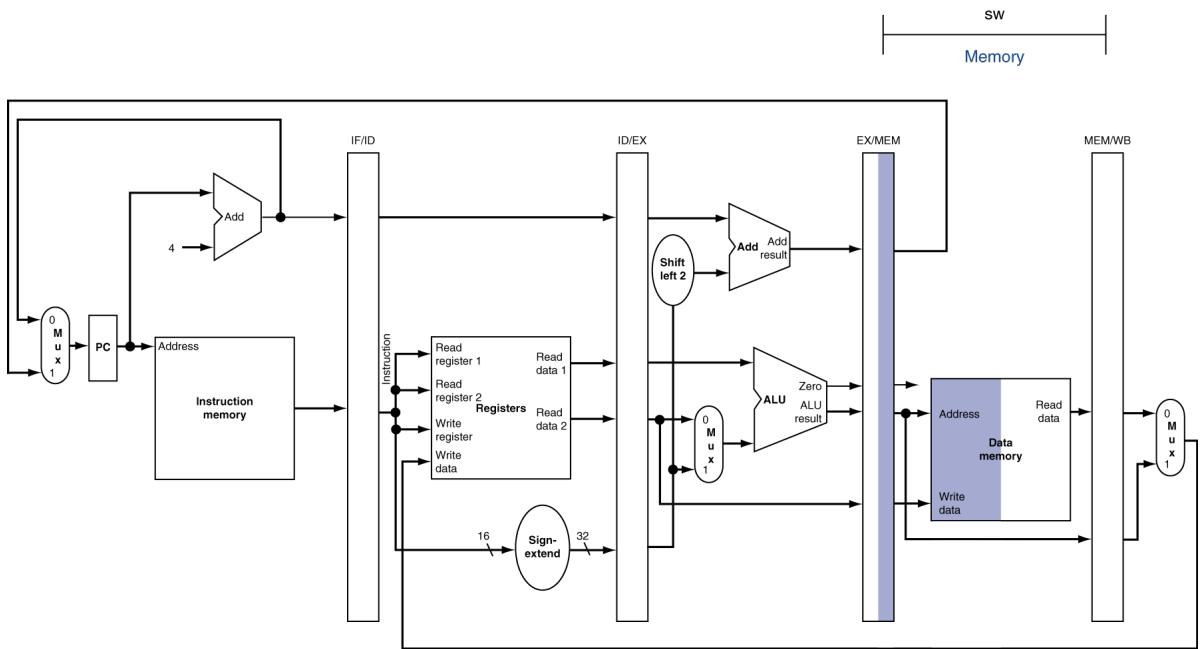
Corrected Datapath for Load



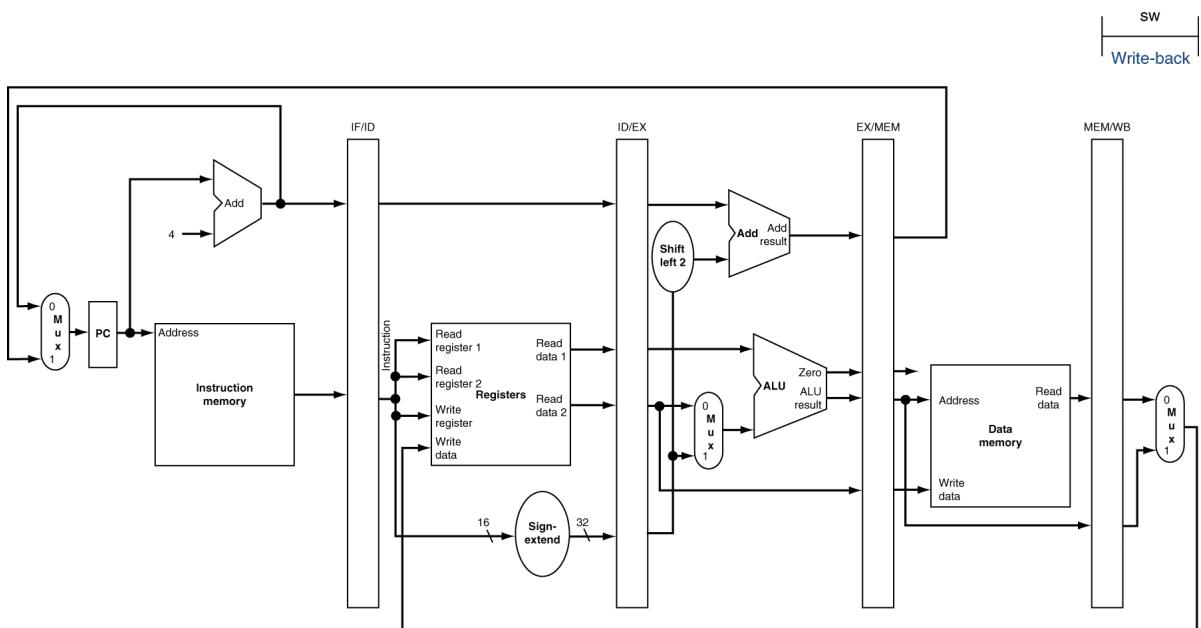
EX for Store



MEM for Store

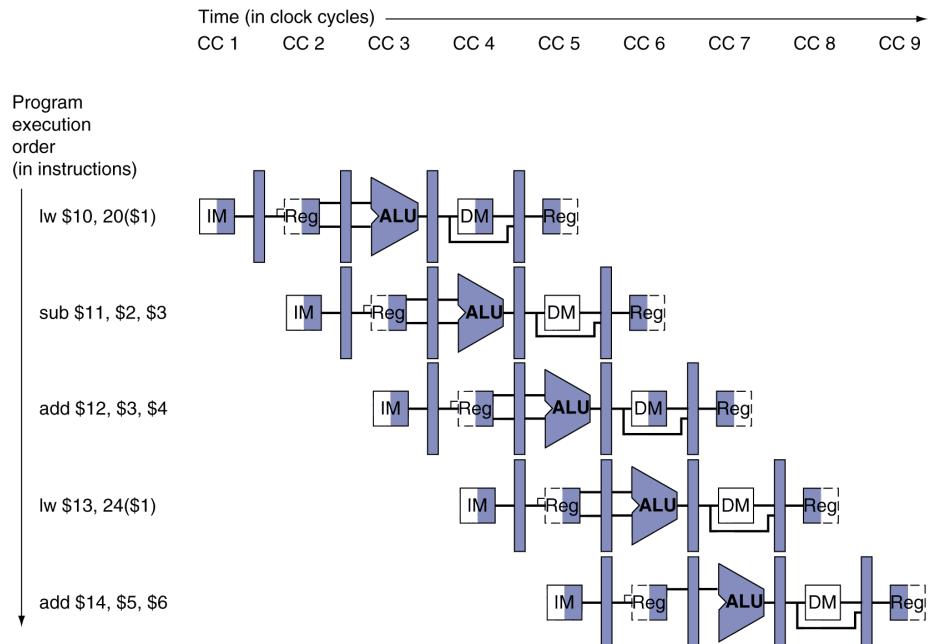


WB for Store



Multi-Cycle Pipeline Diagram

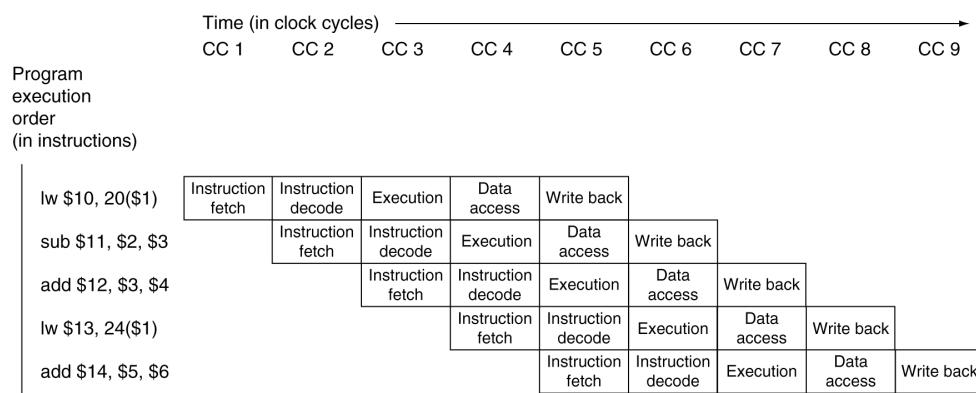
- Form showing resource usage



Chapter 4 — The Processor — 87

Multi-Cycle Pipeline Diagram

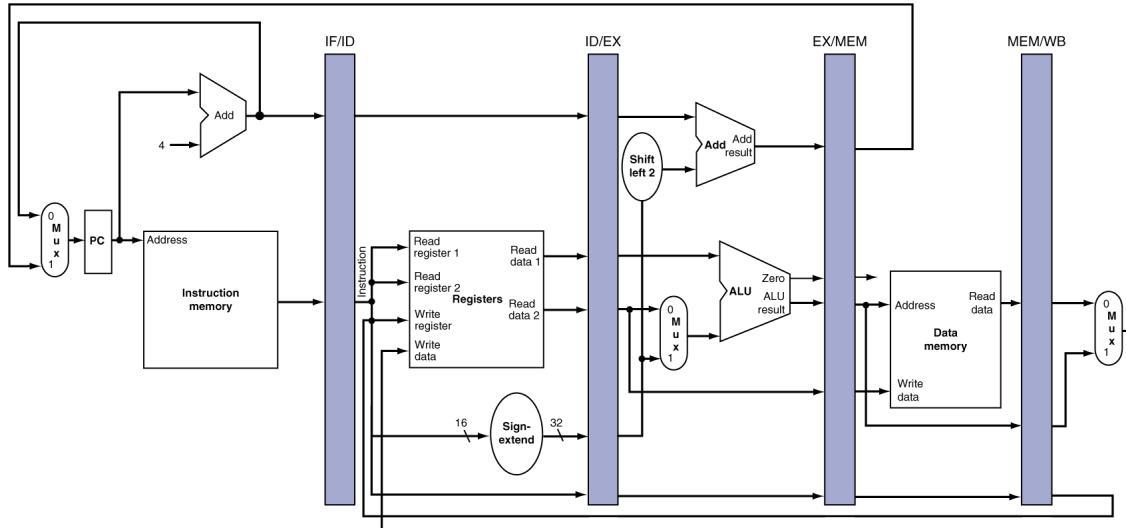
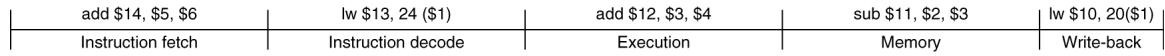
- Traditional form



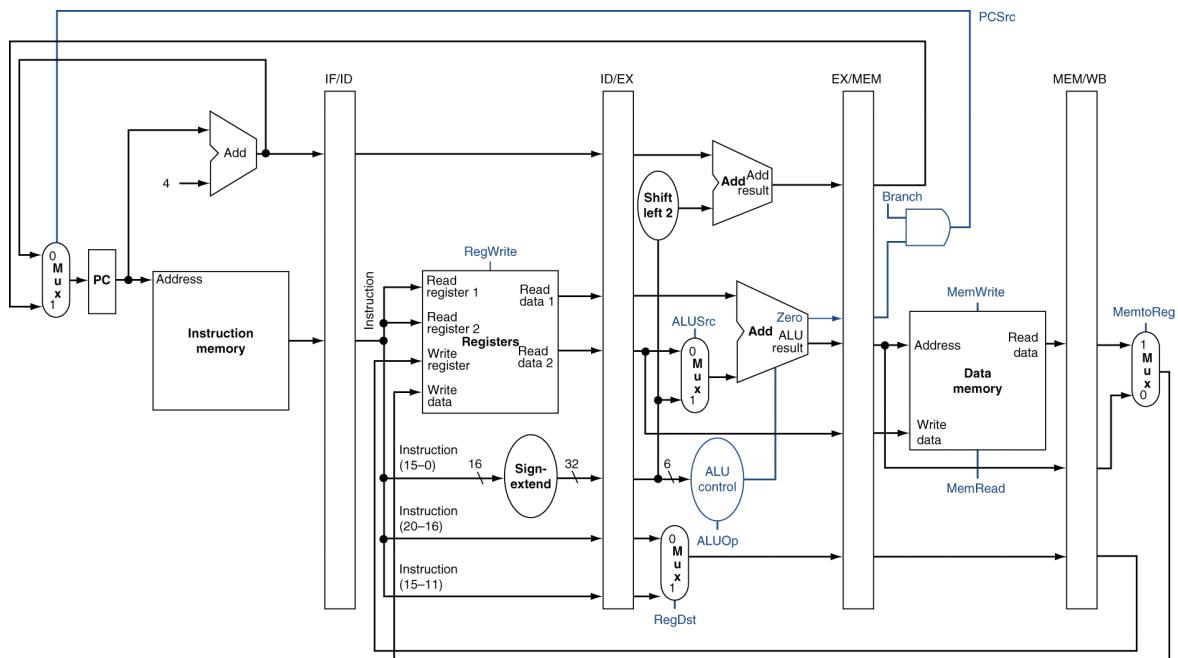
Chapter 4 — The Processor — 88

Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

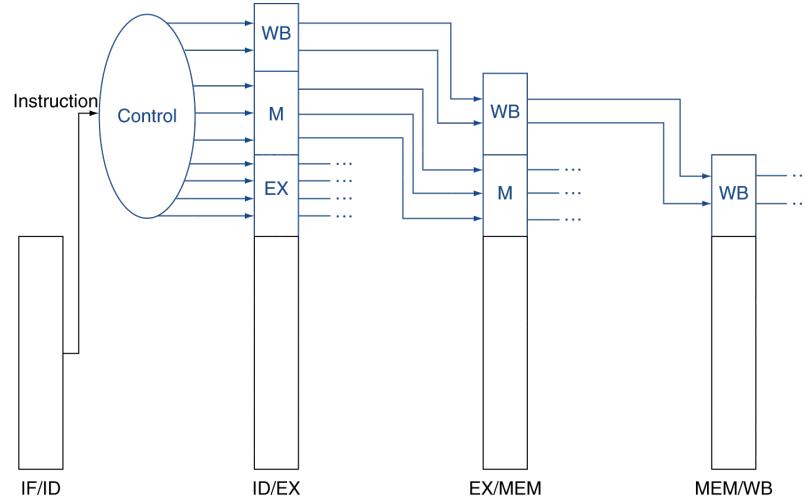


Pipelined Control (Simplified)

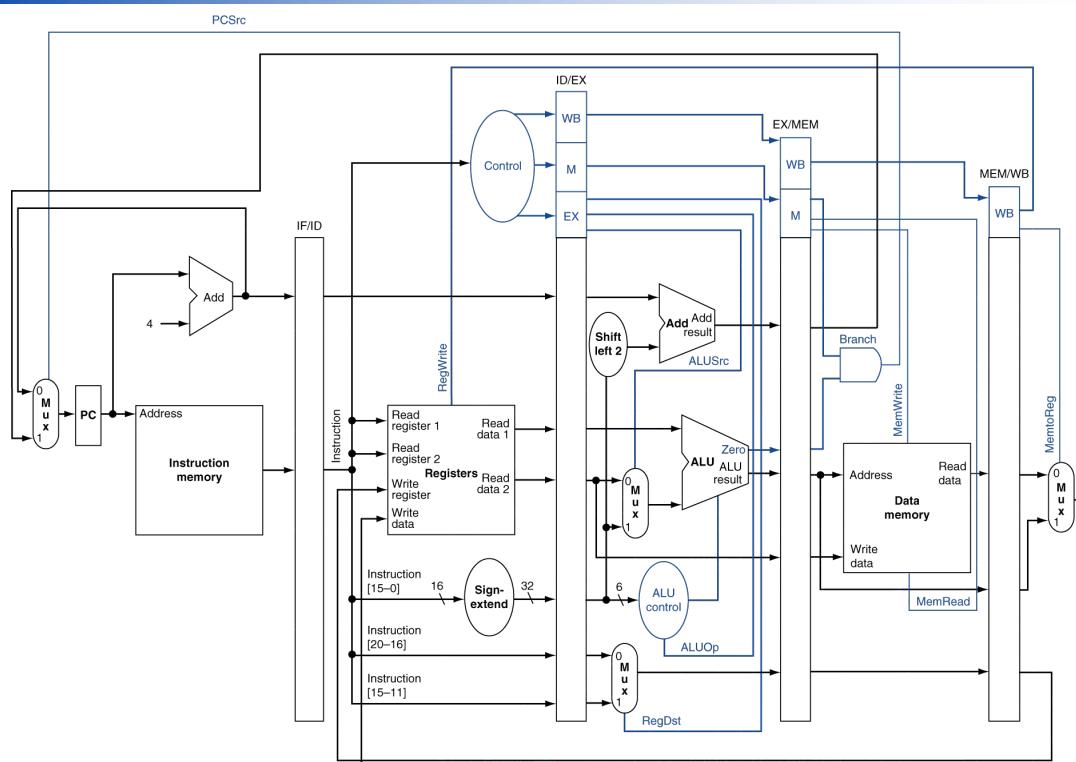


Pipelined Control

- Control signals derived from instruction
 - As in single-cycle implementation



Pipelined Control



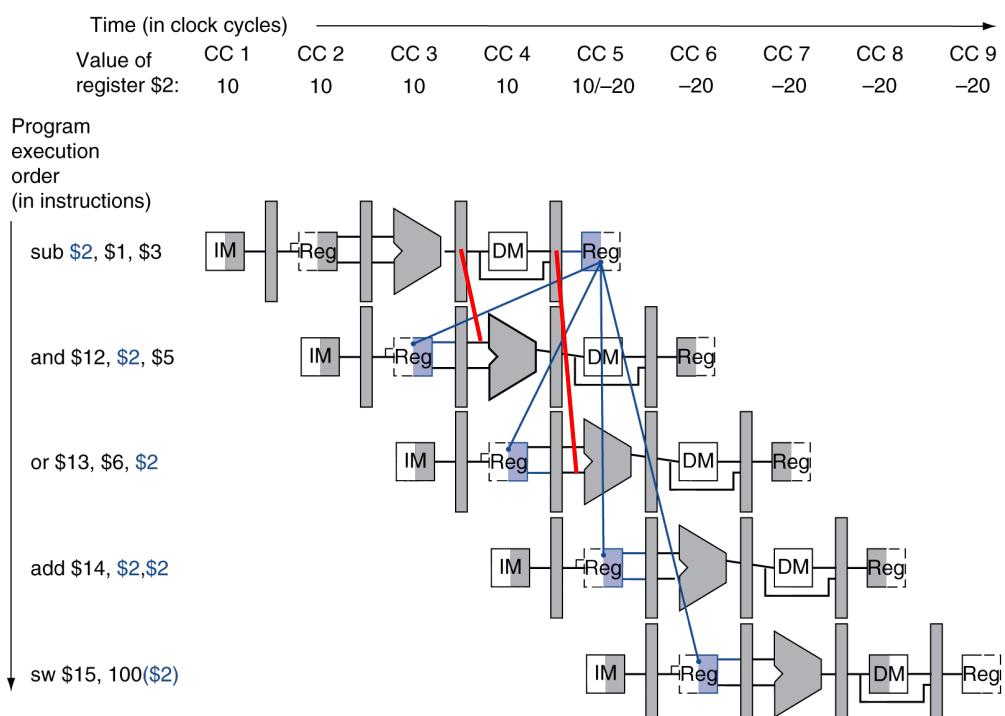
Data Hazards in ALU Instructions

- Consider this sequence:

```
sub $2, $1,$3
and $12,$2,$5
or $13,$6,$2
add $14,$2,$2
sw $15,100($2)
```

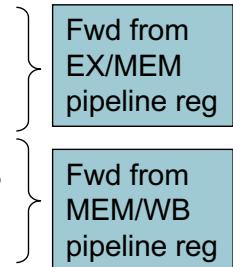
- We can resolve hazards with forwarding
 - How do we detect when to forward?

Dependencies & Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
 - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
 - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
 - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
 - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
 - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
 - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

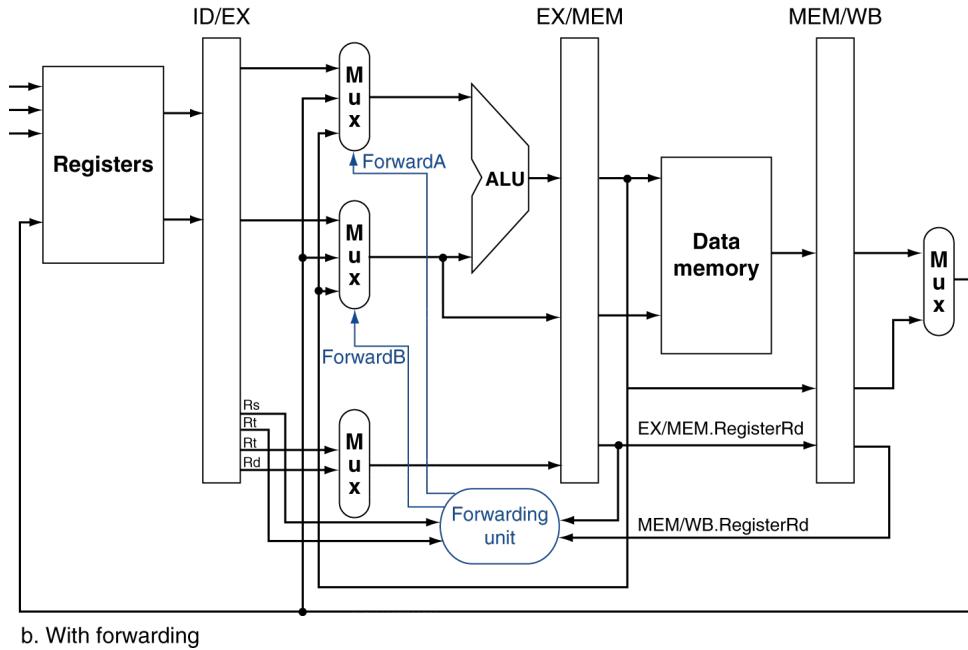


Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
 - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not \$zero
 - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0



Forwarding Paths



b. With forwarding



Forwarding Conditions

- EX hazard
 - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
ForwardA = 10
 - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
ForwardB = 10
- MEM hazard
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01



Double Data Hazard

- Consider the sequence:

add \$1,\$1,\$2

add \$1,\$1,\$3

add \$1,\$1,\$4

- Both hazards occur
 - Want to use the most recent
- Revise MEM hazard condition
 - Only fwd if EX hazard condition isn't true

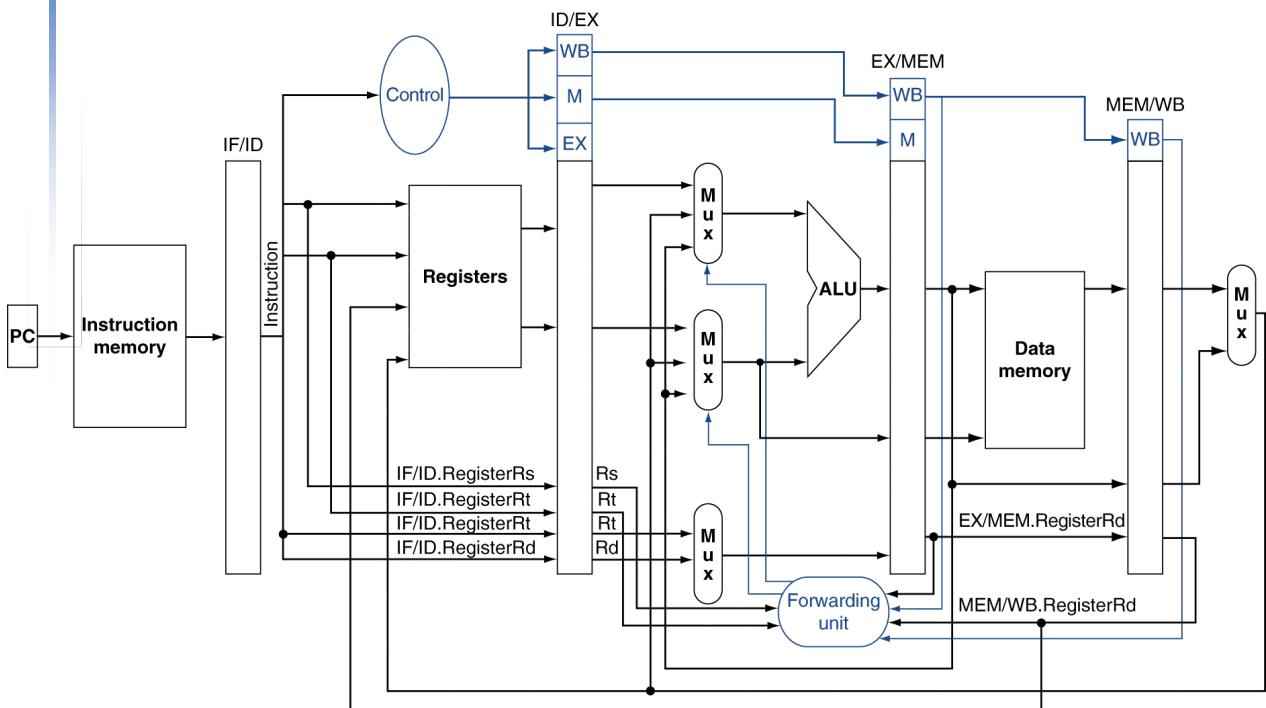


Revised Forwarding Condition

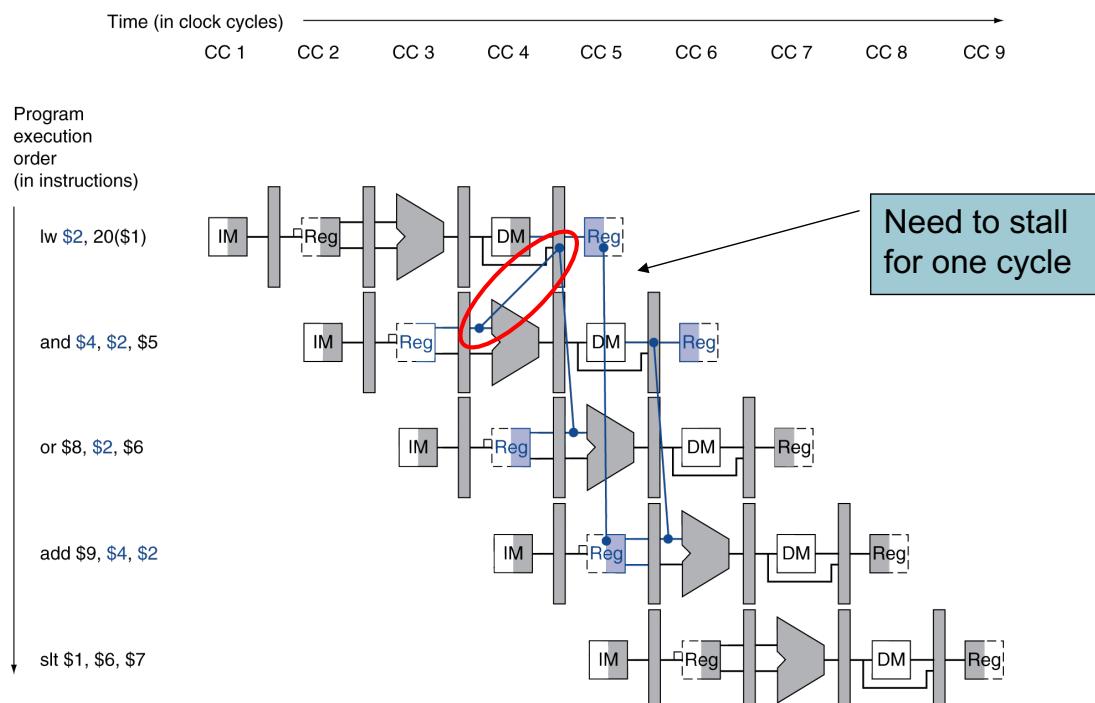
- MEM hazard
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01



Datapath with Forwarding



Load-Use Data Hazard



Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
 - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
 - ID/EX.MemRead and $((ID/EX.RegisterRt = IF/ID.RegisterRs) \text{ or } (ID/EX.RegisterRt = IF/ID.RegisterRt))$
- If detected, stall and insert bubble

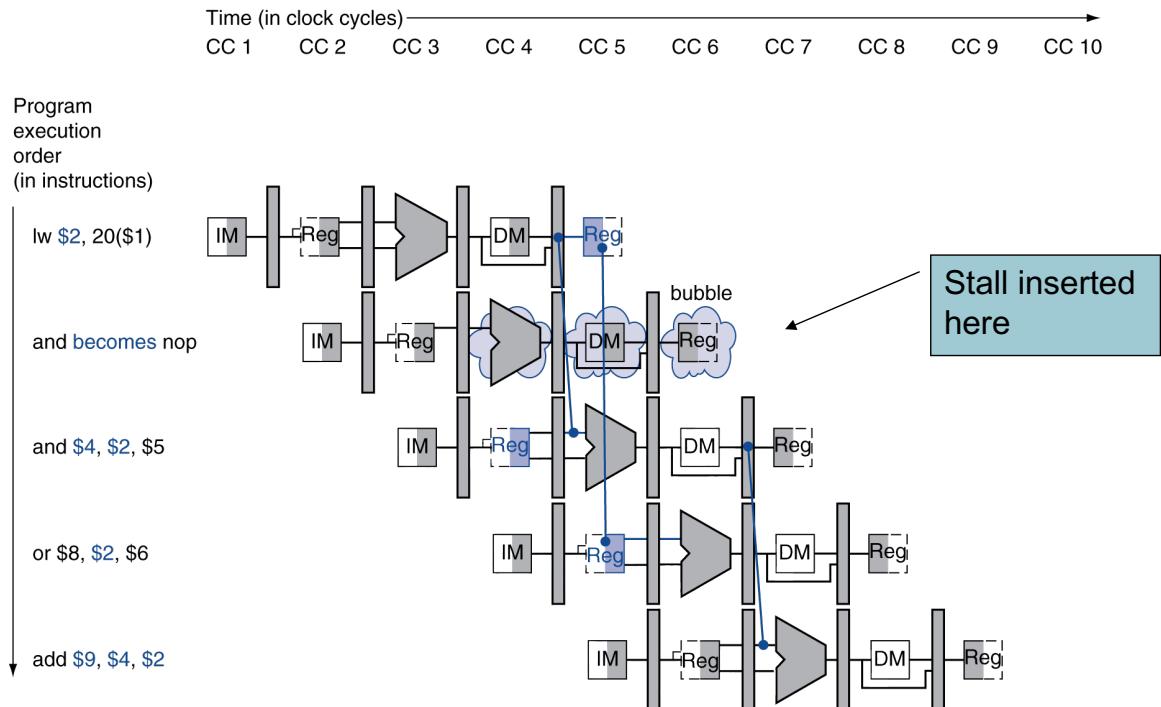


How to Stall the Pipeline

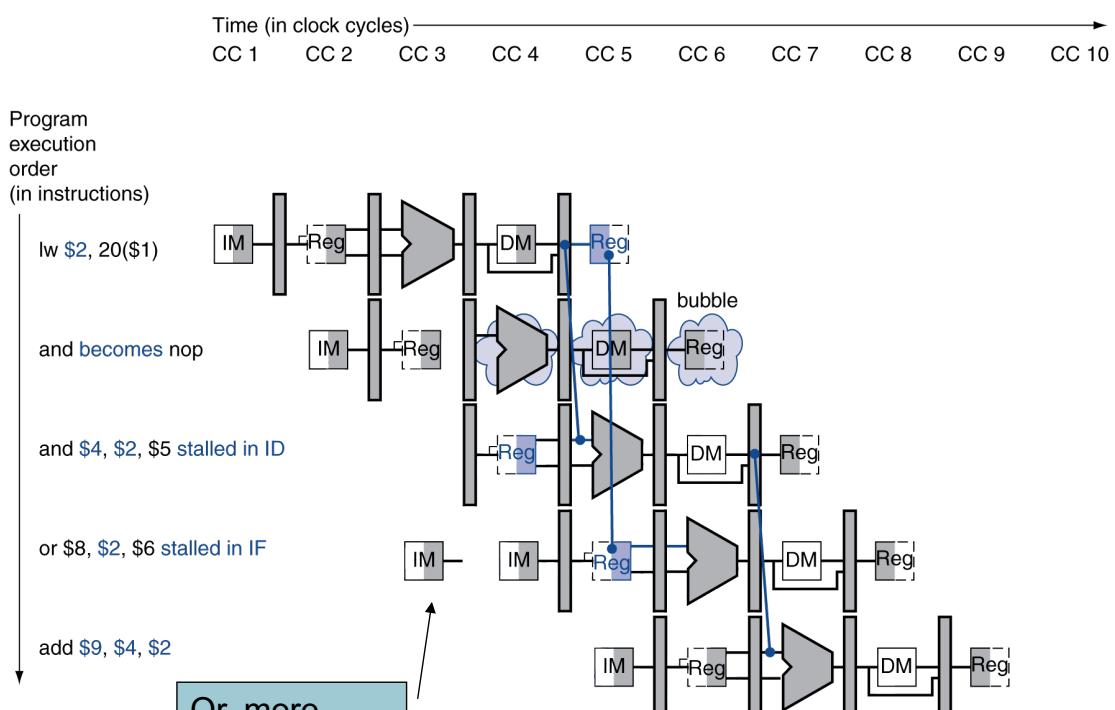
- Force control values in ID/EX register to 0
 - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
 - Using instruction is decoded again
 - Following instruction is fetched again
 - 1-cycle stall allows MEM to read data for 1w
 - Can subsequently forward to EX stage



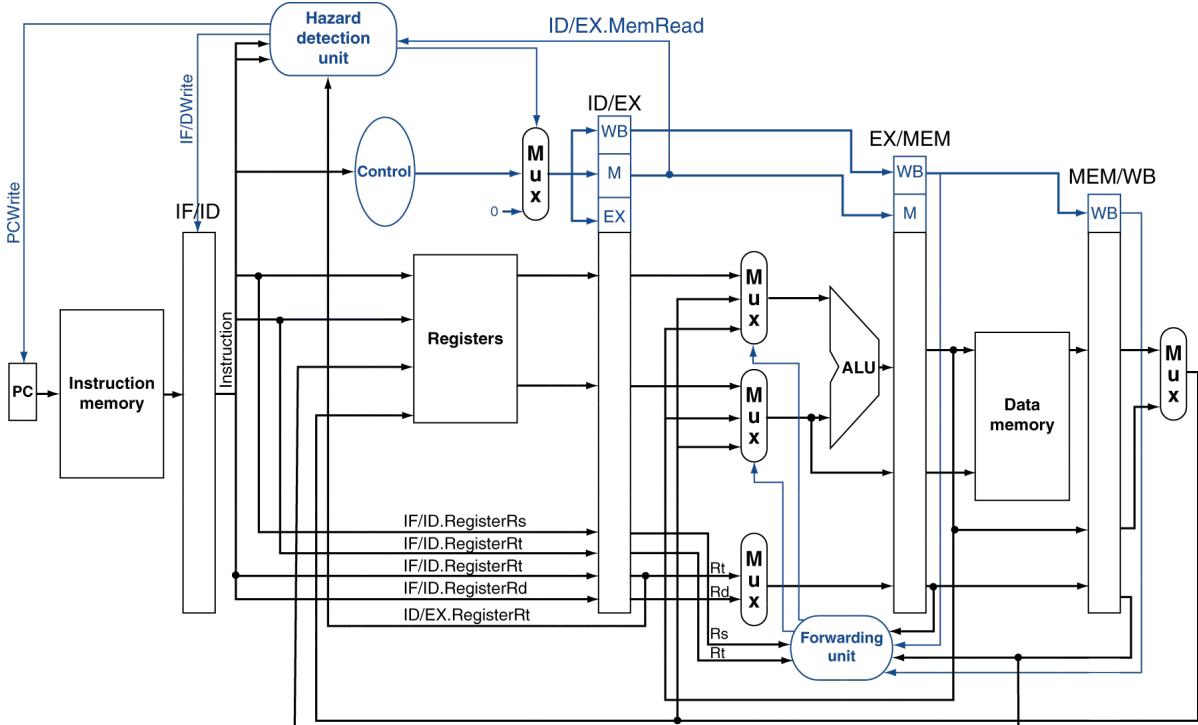
Stall/Bubble in the Pipeline



Stall/Bubble in the Pipeline



Datapath with Hazard Detection



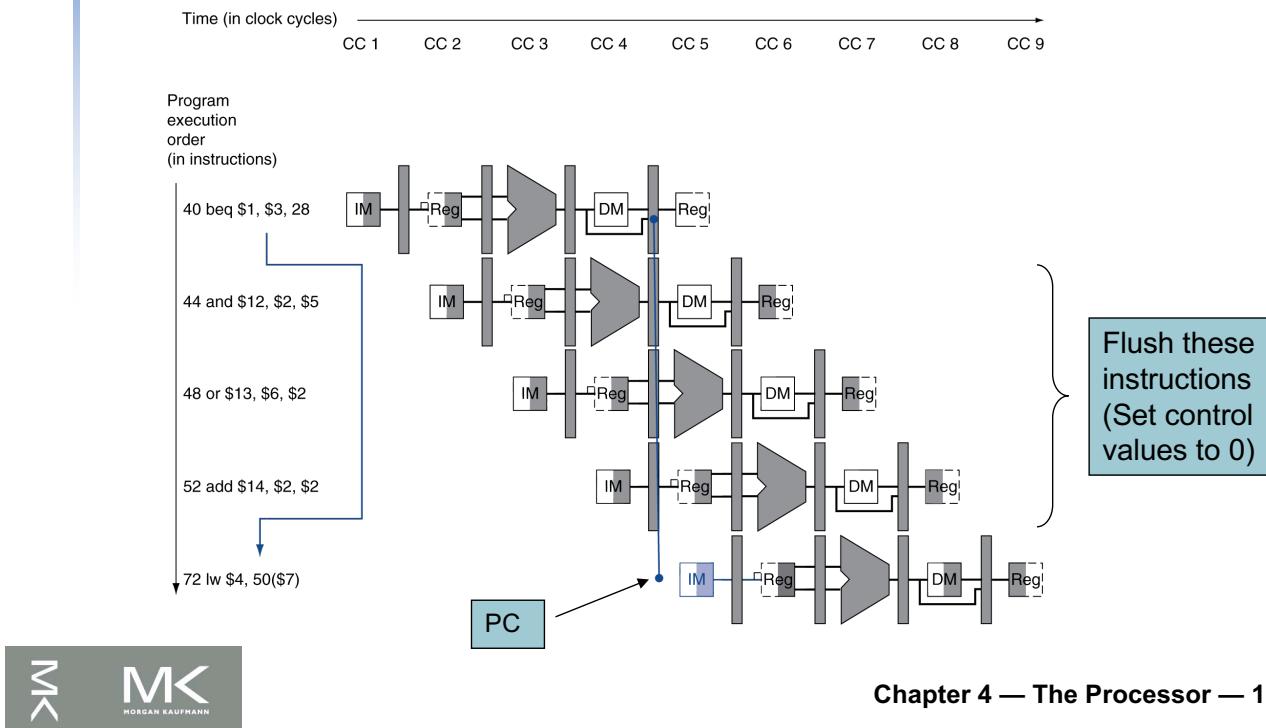
Stalls and Performance

The BIG Picture

- Stalls reduce performance
 - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
 - Requires knowledge of the pipeline structure

Branch Hazards

- If branch outcome determined in MEM



Chapter 4 — The Processor — 109

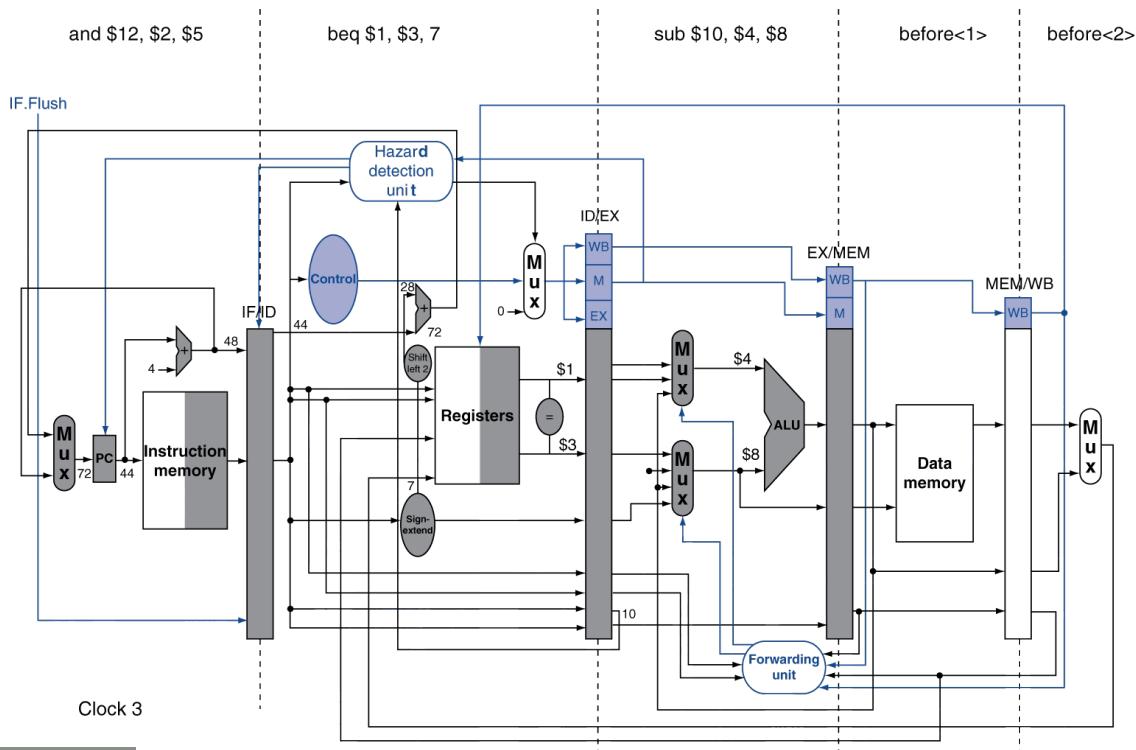
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
 - Target address adder
 - Register comparator
- Example: branch taken

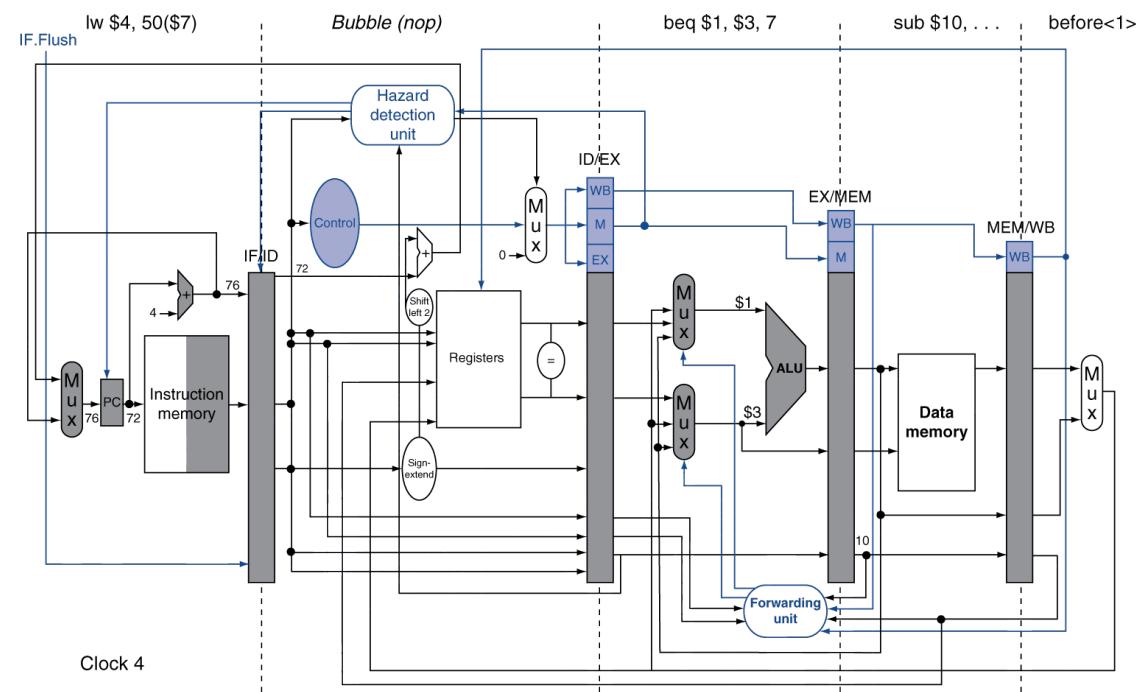

```

36: sub $10, $4, $8
40: beq $1, $3, 7
44: and $12, $2, $5
48: or $13, $2, $6
52: add $14, $4, $2
56: slt $15, $6, $7
...
72: lw $4, 50($7)
      
```

Example: Branch Taken

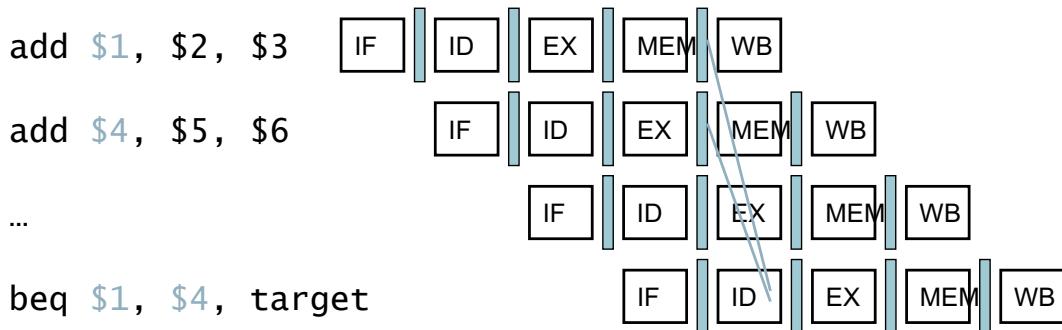


Example: Branch Taken



Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

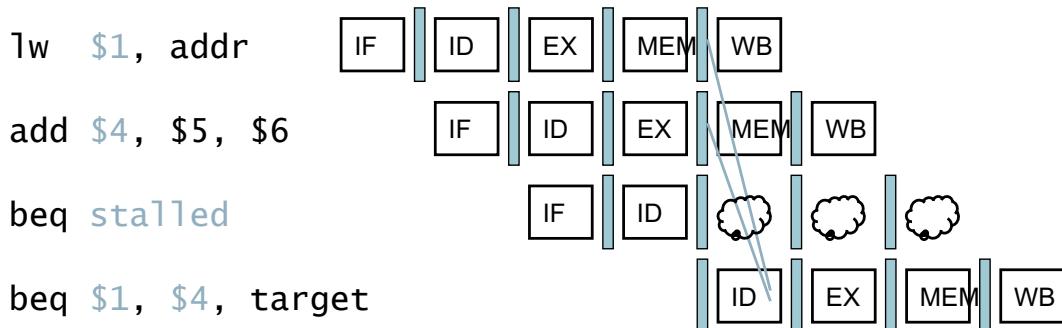


- Can resolve using forwarding



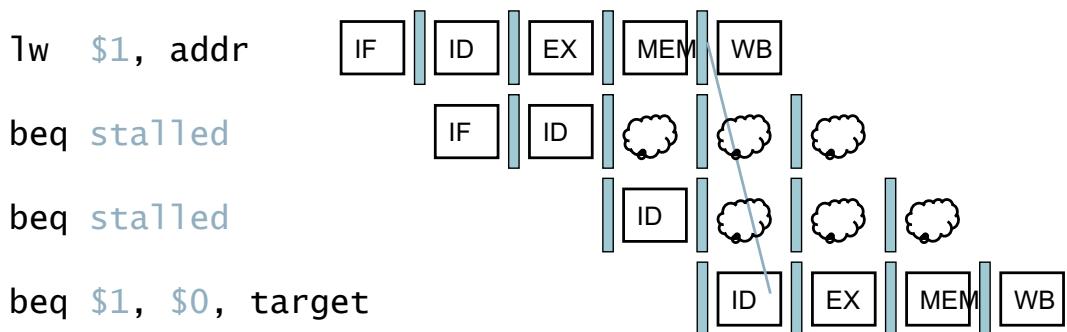
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
 - Need 1 stall cycle



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
 - Need 2 stall cycles



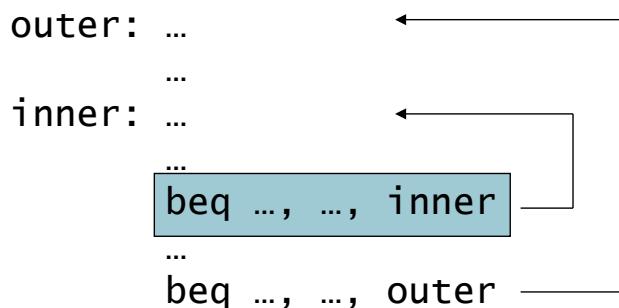
Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
 - Branch prediction buffer (aka branch history table)
 - Indexed by recent branch instruction addresses
 - Stores outcome (taken/not taken)
 - To execute a branch
 - Check table, expect the same outcome
 - Start fetching from fall-through or target
 - If wrong, flush pipeline and flip prediction



1-Bit Predictor: Shortcoming

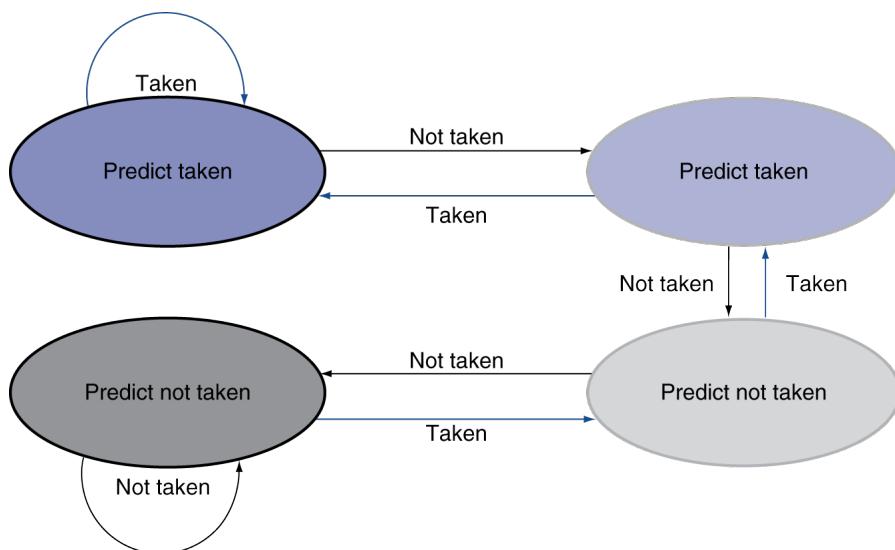
- Inner loop branches mispredicted twice!



- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
 - 1-cycle penalty for a taken branch
- Branch target buffer
 - Cache of target addresses
 - Indexed by PC when instruction fetched
 - If hit and instruction is branch predicted taken, can fetch target immediately



Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
 - Different ISAs use the terms differently
- Exception
 - Arises within the CPU
 - e.g., undefined opcode, overflow, syscall, ...
- Interrupt
 - From an external I/O controller
- Dealing with them without sacrificing performance is hard



Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
 - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
 - In MIPS: Cause register
 - We'll assume 1-bit
 - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180



An Alternate Mechanism

- Vectored Interrupts
 - Handler address determined by the cause
- Example:
 - Undefined opcode: C000 0000
 - Overflow: C000 0020
 - ...: C000 0040
- Instructions either
 - Deal with the interrupt, or
 - Jump to real handler



Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
 - Take corrective action
 - use EPC to return to program
- Otherwise
 - Terminate program
 - Report error using EPC, cause, ...

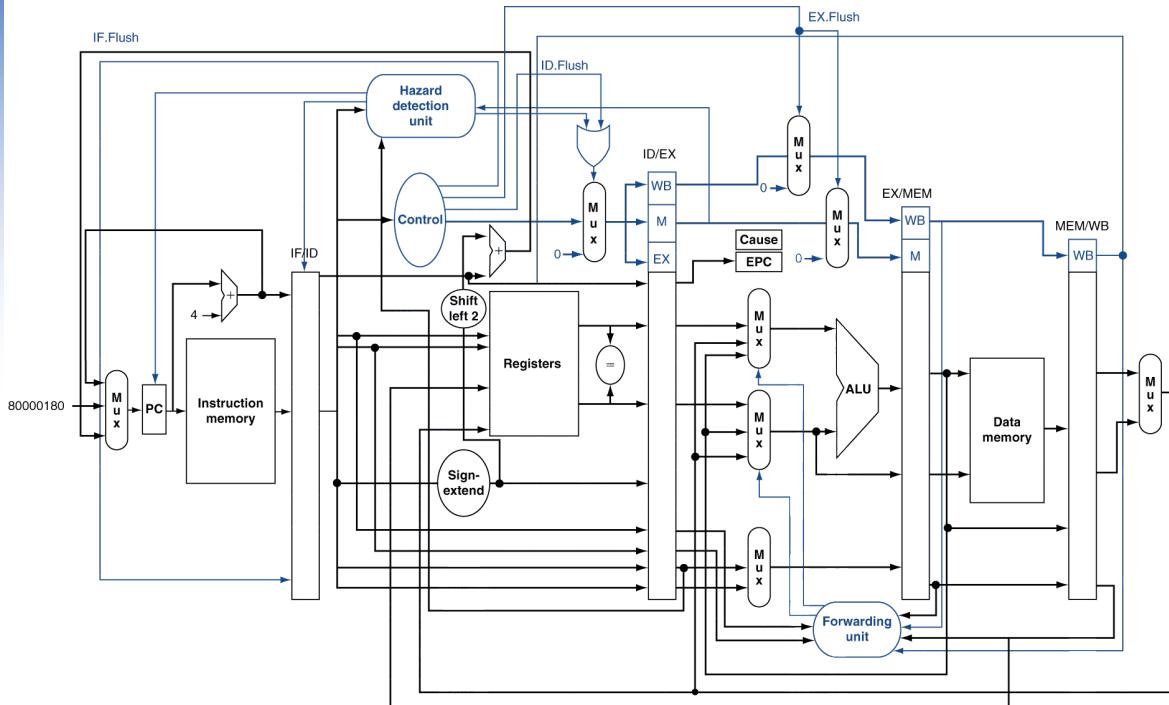


Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
 $\text{add } \$1, \$2, \$1$
 - Prevent \$1 from being clobbered
 - Complete previous instructions
 - Flush add and subsequent instructions
 - Set Cause and EPC register values
 - Transfer control to handler
- Similar to mispredicted branch
 - Use much of the same hardware



Pipeline with Exceptions



Exception Properties

- Restartable exceptions
 - Pipeline can flush the instruction
 - Handler executes, then returns to the instruction
 - Refetched and executed from scratch
- PC saved in EPC register
 - Identifies causing instruction
 - Actually PC + 4 is saved
 - Handler must adjust



Exception Example

Exception on add in

```
40    sub    $11, $2, $4
44    and    $12, $2, $5
48    or     $13, $2, $6
4C    add    $1, $2, $1
50    slt    $15, $6, $7
54    lw     $16, 50($7)
```

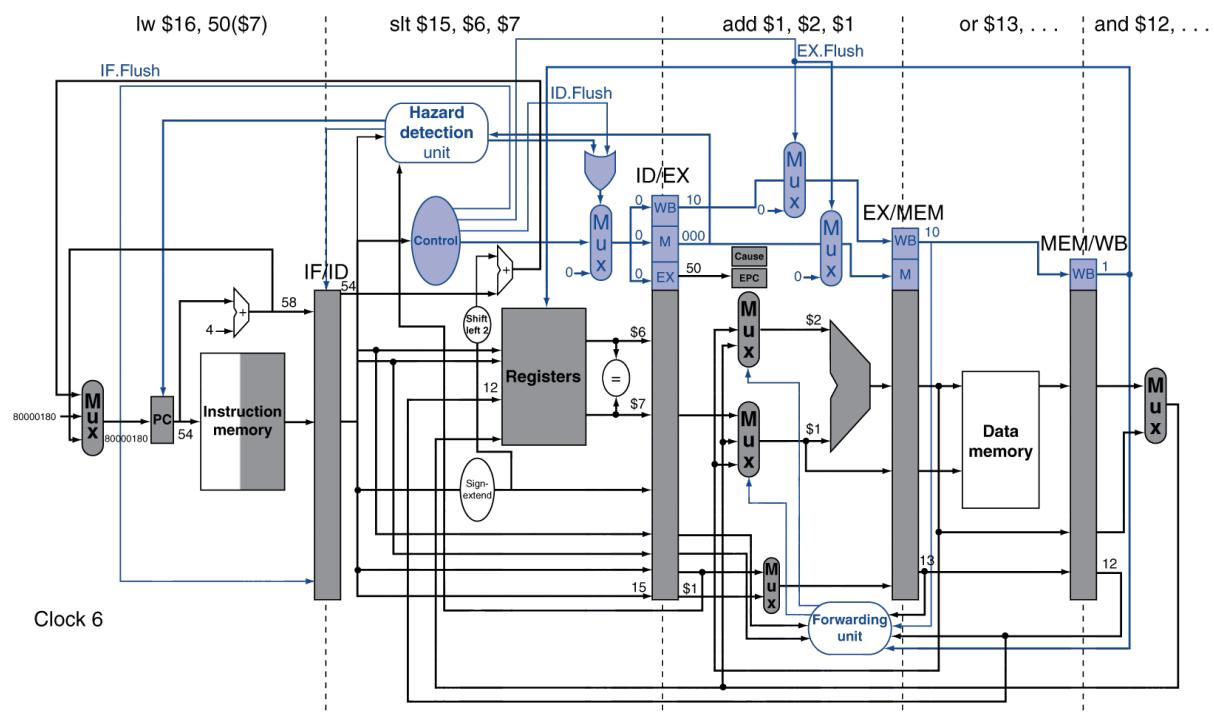
...

Handler

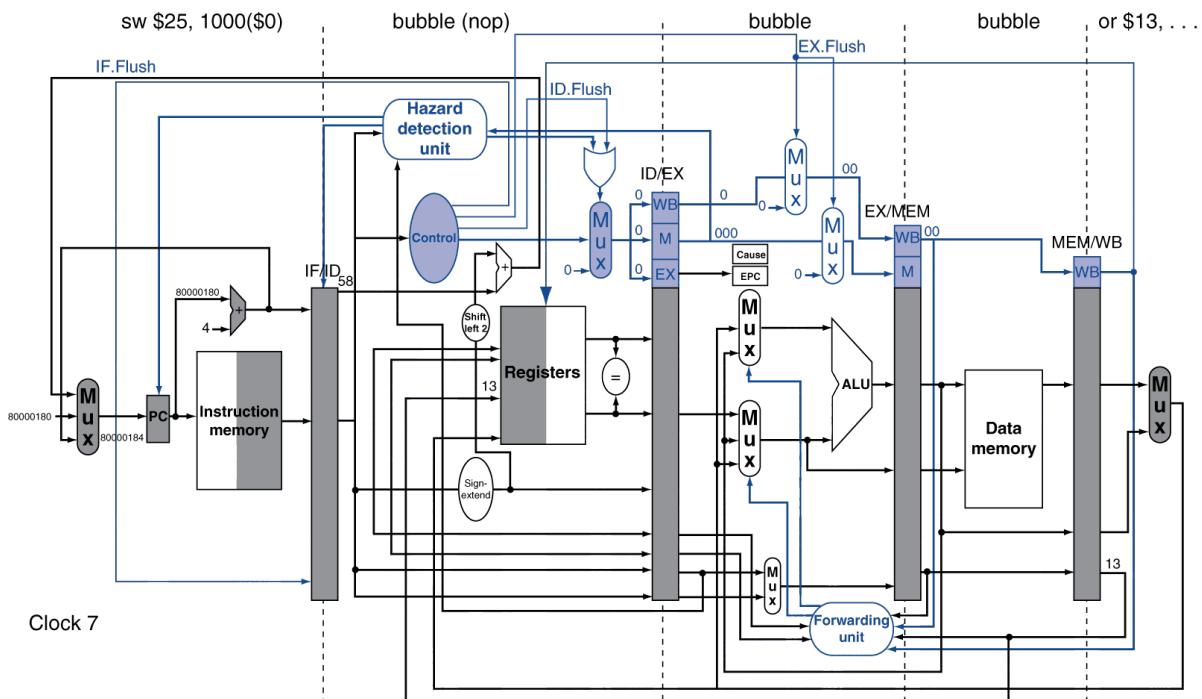
```
80000180    sw     $25, 1000($0)
80000184    sw     $26, 1004($0)
```

...

Exception Example



Exception Example



Multiple Exceptions

- Pipelining overlaps multiple instructions
 - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
 - Flush subsequent instructions
 - “Precise” exceptions
- In complex pipelines
 - Multiple instructions issued per cycle
 - Out-of-order completion
 - Maintaining precise exceptions is difficult!



Imprecise Exceptions

- Just stop pipeline and save state
 - Including exception cause(s)
- Let the handler work out
 - Which instruction(s) had exceptions
 - Which to complete or flush
 - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines



Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
 - Deeper pipeline
 - Less work per stage \Rightarrow shorter clock cycle
 - Multiple issue
 - Replicate pipeline stages \Rightarrow multiple pipelines
 - Start multiple instructions per clock cycle
 - CPI < 1, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak CPI = 0.25, peak IPC = 4
 - But dependencies reduce this in practice



Multiple Issue

- Static multiple issue
 - Compiler groups instructions to be issued together
 - Packages them into “issue slots”
 - Compiler detects and avoids hazards
- Dynamic multiple issue
 - CPU examines instruction stream and chooses instructions to issue each cycle
 - Compiler can help by reordering instructions
 - CPU resolves hazards using advanced techniques at runtime



Speculation

- “Guess” what to do with an instruction
 - Start operation as soon as possible
 - Check whether guess was right
 - If so, complete the operation
 - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
 - Speculate on branch outcome
 - Roll back if path taken is different
 - Speculate on load
 - Roll back if location is updated



Compiler/Hardware Speculation

- Compiler can reorder instructions
 - e.g., move load before branch
 - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
 - Buffer results until it determines they are actually needed
 - Flush buffers on incorrect speculation



Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
 - e.g., speculative load before null-pointer check
- Static speculation
 - Can add ISA support for deferring exceptions
- Dynamic speculation
 - Can buffer exceptions until instruction completion (which may not occur)



Static Multiple Issue

- Compiler groups instructions into “issue packets”
 - Group of instructions that can be issued on a single cycle
 - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
 - Specifies multiple concurrent operations
 - ⇒ Very Long Instruction Word (VLIW)



Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
 - Reorder instructions into issue packets
 - No dependencies with a packet
 - Possibly some dependencies between packets
 - Varies between ISAs; compiler must know!
 - Pad with nop if necessary



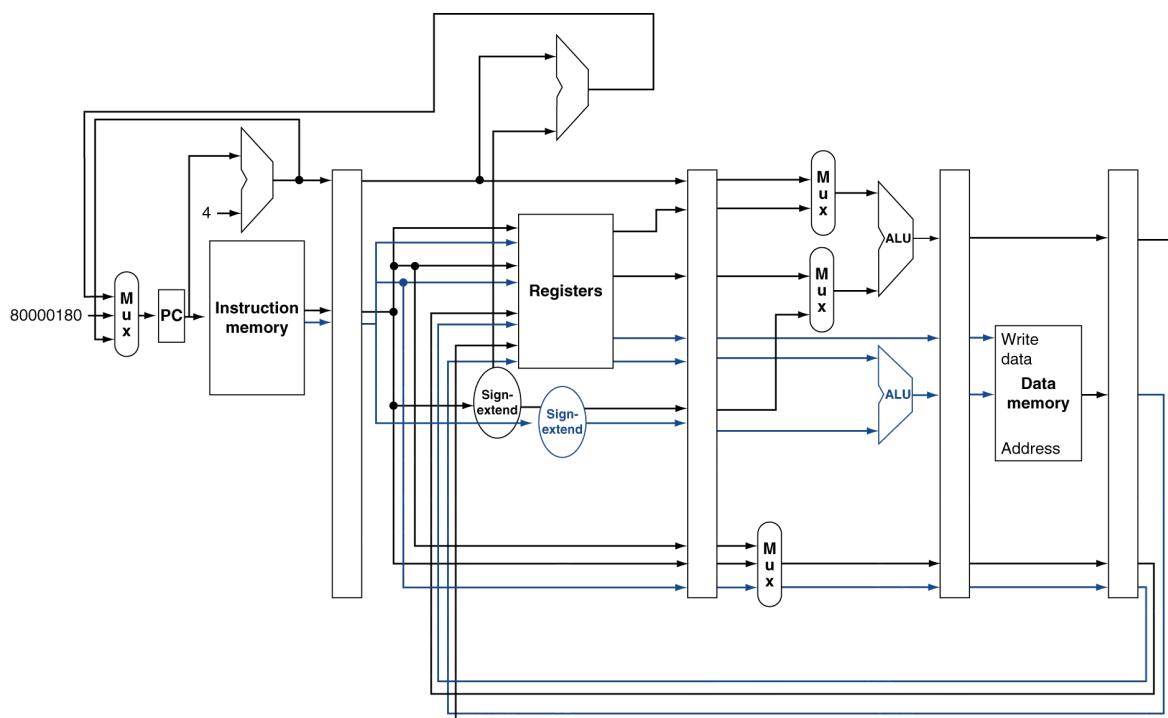
MIPS with Static Dual Issue

- Two-issue packets
 - One ALU/branch instruction
 - One load/store instruction
 - 64-bit aligned
 - ALU/branch, then load/store
 - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB



MIPS with Static Dual Issue



Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
 - Forwarding avoided stalls with single-issue
 - Now can't use ALU result in load/store in same packet
 - add \$t0, \$s0, \$s1
Load \$s2, 0(\$t0)
 - Split into two packets, effectively a stall
- Load-use hazard
 - Still one cycle use latency, but now two instructions
- More aggressive scheduling required



Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw    $t0, 0($s1)      # $t0=array element
      addu $t0, $t0, $s2      # add scalar in $s2
      sw    $t0, 0($s1)      # store result
      addi $s1, $s1,-4        # decrement pointer
      bne  $s1, $zero, Loop   # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1,-4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4

- $IPC = 5/4 = 1.25$ (c.f. peak IPC = 2)



Loop Unrolling

- Replicate loop body to expose more parallelism
 - Reduces loop-control overhead
- Use different registers per replication
 - Called “register renaming”
 - Avoid loop-carried “anti-dependencies”
 - Store followed by a load of the same register
 - Aka “name dependence”
 - Reuse of a register name



Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi \$s1, \$s1, -16	lw \$t0, 0(\$s1)	1
	nop	lw \$t1, 12(\$s1)	2
	addu \$t0, \$t0, \$s2	lw \$t2, 8(\$s1)	3
	addu \$t1, \$t1, \$s2	lw \$t3, 4(\$s1)	4
	addu \$t2, \$t2, \$s2	sw \$t0, 16(\$s1)	5
	addu \$t3, \$t4, \$s2	sw \$t1, 12(\$s1)	6
	nop	sw \$t2, 8(\$s1)	7
	bne \$s1, \$zero, Loop	sw \$t3, 4(\$s1)	8

- IPC = $14/8 = 1.75$
 - Closer to 2, but at cost of registers and code size



Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, ... each cycle
 - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
 - Though it may still help
 - Code semantics ensured by the CPU

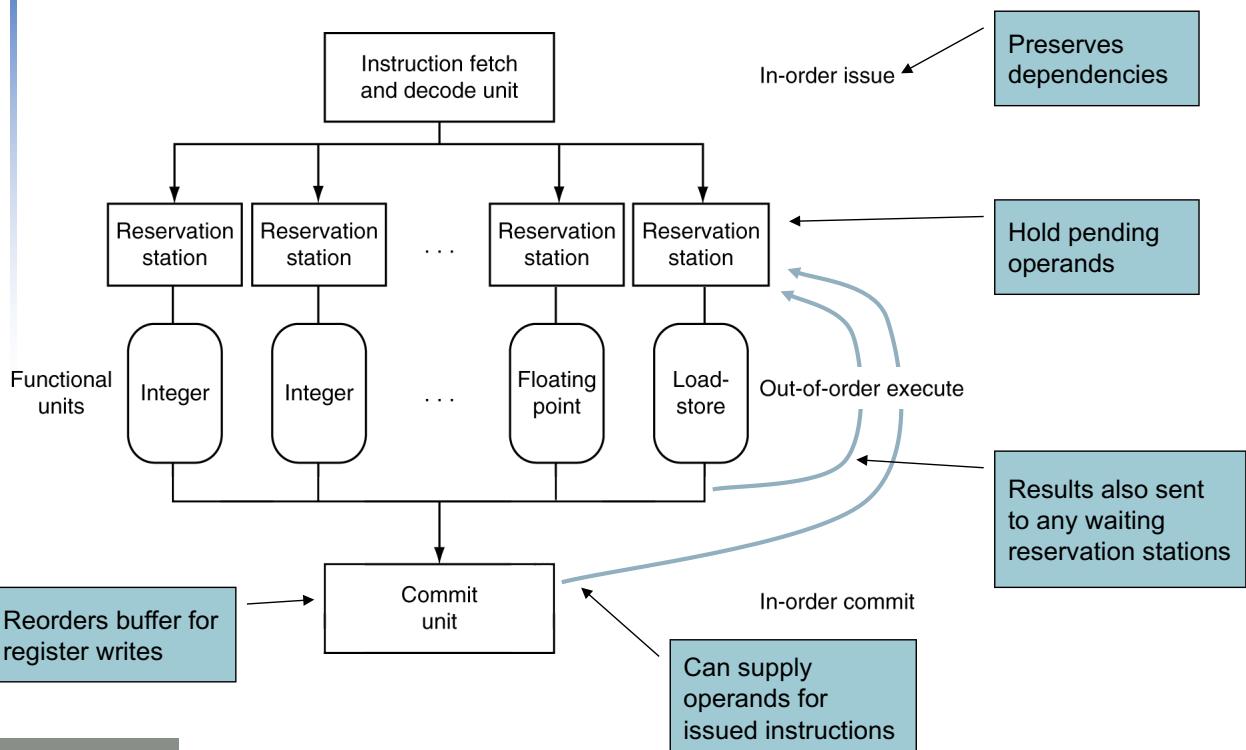


Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
 - But commit result to registers in order
- Example
 - 1w \$t0, 20(\$s2)
 - addu \$t1, \$t0, \$t2
 - sub \$s4, \$s4, \$t3
 - slti \$t5, \$s4, 20
 - Can start sub while addu is waiting for lw



Dynamically Scheduled CPU



Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
 - If operand is available in register file or reorder buffer
 - Copied to reservation station
 - No longer required in the register; can be overwritten
 - If operand is not yet available
 - It will be provided to the reservation station by a function unit
 - Register update may not be required

Speculation

- Predict branch and continue issuing
 - Don't commit until branch outcome determined
- Load speculation
 - Avoid load and cache miss delay
 - Predict the effective address
 - Predict loaded value
 - Load before completing outstanding stores
 - Bypass stored values to load unit
 - Don't commit load until speculation cleared



Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predictable
 - e.g., cache misses
- Can't always schedule around branches
 - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards



Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
 - e.g., pointer aliasing
- Some parallelism is hard to expose
 - Limited window size during instruction issue
- Memory delays and limited bandwidth
 - Hard to keep pipelines full
- Speculation can help if done well



MK
MORGAN KAUFMANN

Chapter 4 — The Processor — 151

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W



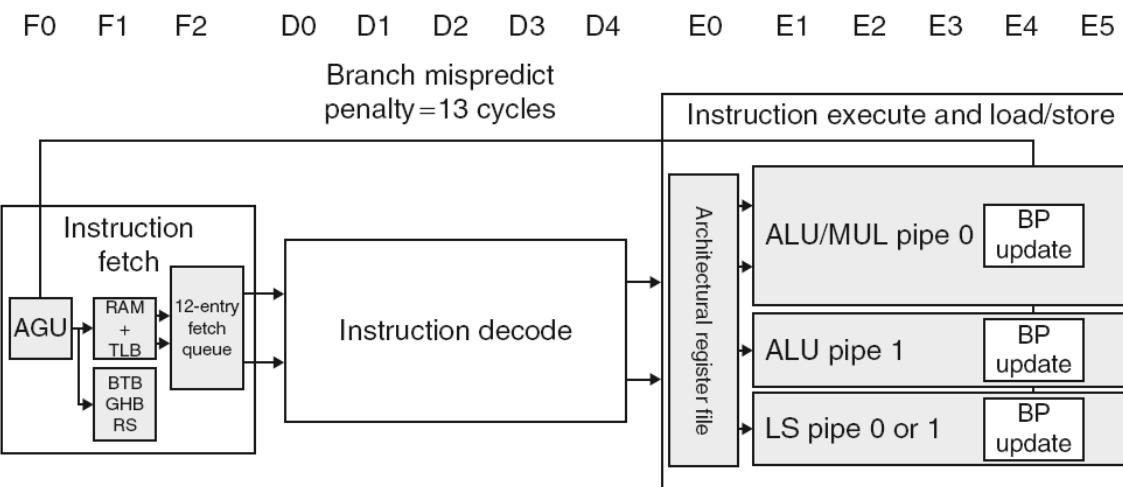
MK
MORGAN KAUFMANN

Chapter 4 — The Processor — 152

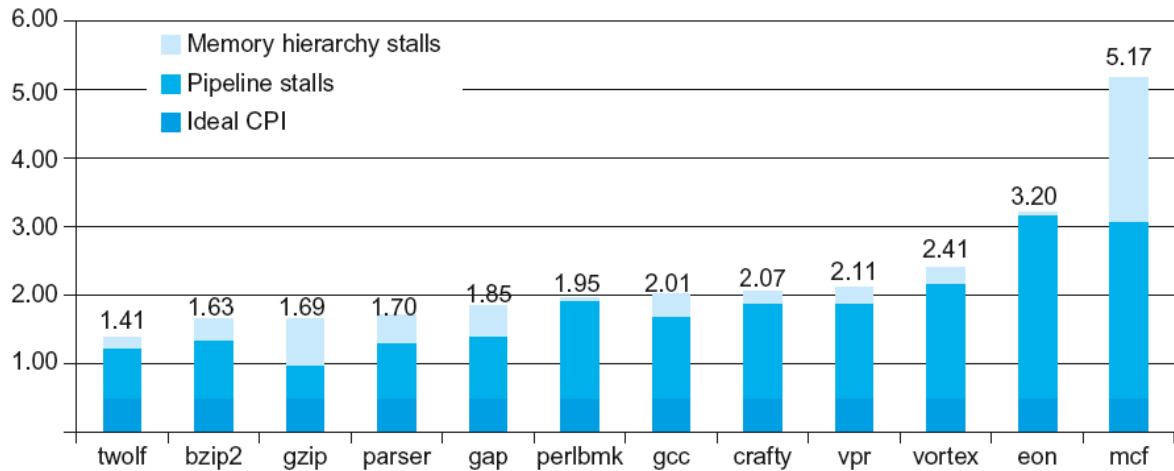
Cortex A8 and Intel i7

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	2-level	2-level
1 st level caches/core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2 nd level caches/core	128-1024 KiB	256 KiB
3 rd level caches (shared)	-	2- 8 MB

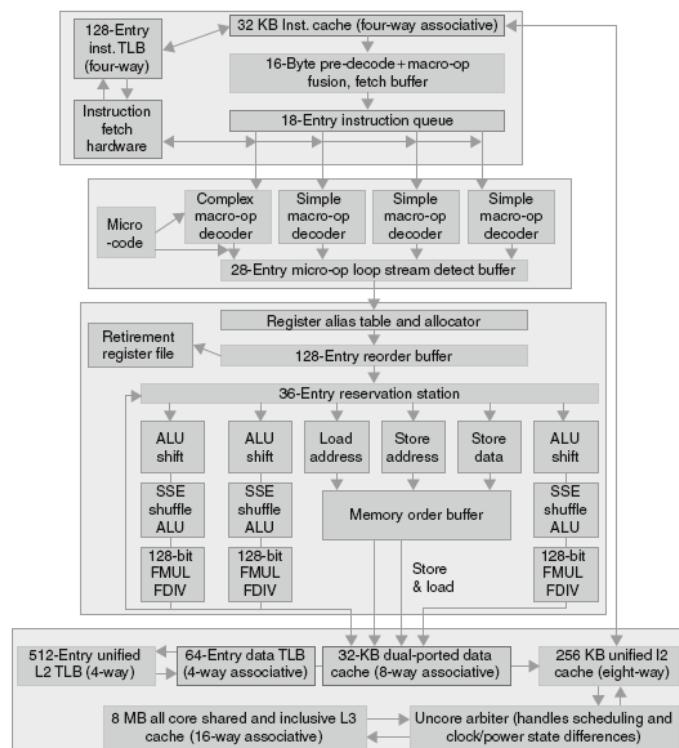
ARM Cortex-A8 Pipeline



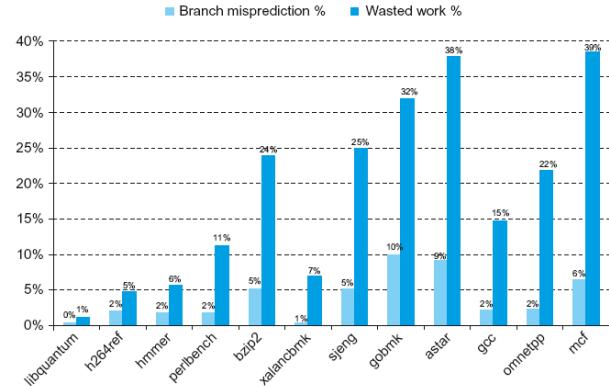
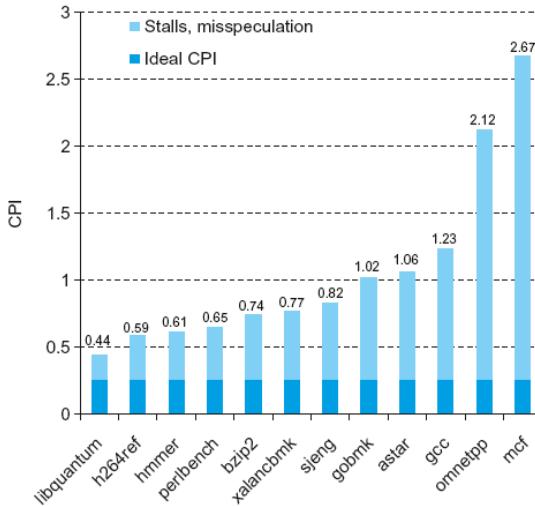
ARM Cortex-A8 Performance



Core i7 Pipeline



Core i7 Performance



Matrix Multiply

■ Unrolled C code

```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6     for ( int i = 0; i < n; i+=UNROLL*4 )
7         for ( int j = 0; j < n; j++ ) {
8             __m256d c[4];
9             for ( int x = 0; x < UNROLL; x++ )
10                 c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12             for( int k = 0; k < n; k++ )
13             {
14                 __m256d b = _mm256_broadcast_sd(B+k+j*n);
15                 for (int x = 0; x < UNROLL; x++)
16                     c[x] = _mm256_add_pd(c[x],
17                                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18             }
19
20             for ( int x = 0; x < UNROLL; x++ )
21                 _mm256_store_pd(C+i+x*4+j*n, c[x]);
22         }
23 }
```

Matrix Multiply

Assembly code:

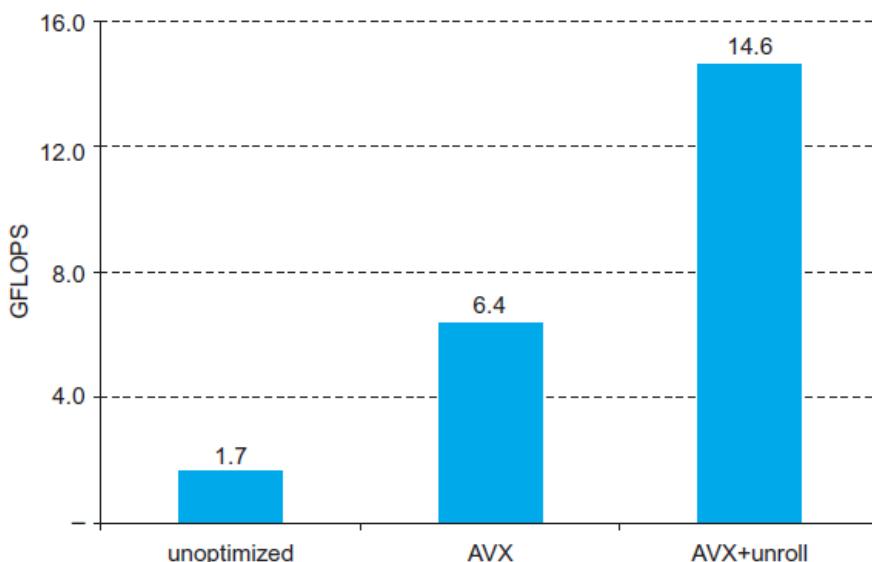
```

1 vmovapd (%r11),%ymm4          # Load 4 elements of C into %ymm4
2 mov %rbx,%rax                # register %rax = %rbx
3 xor %ecx,%ecx                # register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3      # Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2      # Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1      # Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0 # Make 4 copies of B element
8 add $0x8,%rcx # register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5    # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4     # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3     # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5 # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0 # Parallel mul %ymm1,4 A elements
15 add %r8,%rax                # register %rax = %rax + %r8
16 cmp %r10,%rcx                # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2     # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1     # Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>         # jump if not %r8 != %rax
20 add $0x1,%esi                # register %esi = %esi + 1
21 vmovapd %ymm4,(%r11)          # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)      # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)      # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)      # Store %ymm1 into 4 C elements

```



Performance Impact



Fallacies

- Pipelining is easy (!)
 - The basic idea is easy
 - The devil is in the details
 - e.g., detecting data hazards
- Pipelining is independent of technology
 - So why haven't we always done pipelining?
 - More transistors make more advanced techniques feasible
 - Pipeline-related ISA design needs to take account of technology trends
 - e.g., predicated instructions



Pitfalls

- Poor ISA design can make pipelining harder
 - e.g., complex instruction sets (VAX, IA-32)
 - Significant overhead to make pipelining work
 - IA-32 micro-op approach
 - e.g., complex addressing modes
 - Register update side effects, memory indirection
 - e.g., delayed branches
 - Advanced pipelines have long delay slots

