Digital Logic & Computer Design
CS 4341

Professor Dan Moldovan

Spring 2010

Copyright © 2007 Elsevier

Chapter 7 :: Microarchitecture

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

Copyright © 2007 Elsevier

Chapter 7 :: Topics

- **Introduction**
- **Performance Analysis**
- **Single-Cycle Processor**
- **Multicycle Processor**
- **Pipelined Processor**
- **Exceptions**
- **Advanced Microarchitecture**

Copyright © 2007 Elsevier

Introduction

- Microarchitecture: how to implement an architecture in hardware
- Processor:
  - Datapath: functional blocks
  - Control: control signals

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

Copyright © 2007 Elsevier

Microarchitecture

- Multiple implementations for a single architecture:
  - Single-cycle
    - Each instruction executes in a single cycle
  - Multicycle
    - Each instruction is broken up into a series of shorter steps
  - Pipelined
    - Each instruction is broken up into a series of steps
    - Multiple instructions execute at once.

Copyright © 2007 Elsevier

Processor Performance

- Program execution time

Execution Time = (# instructions)(cycles/instruction)(seconds/cycle)

- Definitions:
  - Cycles/instruction = CPI
  - Seconds/cycle = clock period
  - 1/CPI = Instructions/cycle = IPC
- Challenge is to satisfy constraints of:
  - Cost
  - Power
  - Performance

Copyright © 2007 Elsevier

MIPS Processor

- We consider a subset of MIPS instructions:
  - R-type instructions: `and`, `or`, `add`, `sub`, `slt`
  - Memory instructions: `lw`, `sw`
  - Branch instructions: `beq`
- Later consider adding `addi` and `j`

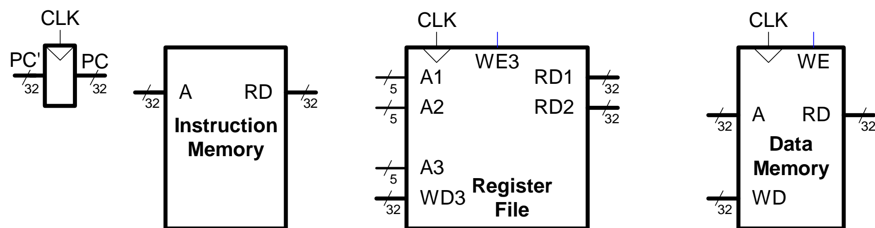
Copyright © 2007 Elsevier

Architectural State

- Determines everything about a processor:
  - `PC`
  - 32 registers
  - Memory

Copyright © 2007 Elsevier

MIPS State Elements



Copyright © 2007 Elsevier

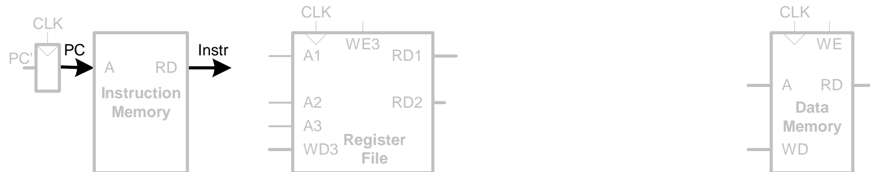
Single-Cycle MIPS Processor

- Datapath
- Control

Copyright © 2007 Elsevier

Single-Cycle Datapath: `lw` fetch

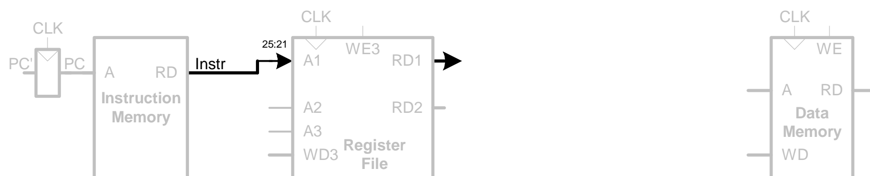
- First consider executing `lw`
- **STEP 1:** Fetch instruction



Copyright © 2007 Elsevier

Single-Cycle Datapath: `lw` register read

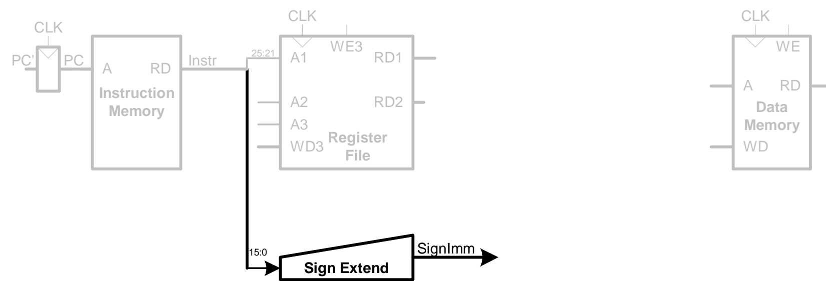
- **STEP 2:** Read source operands from register file



Copyright © 2007 Elsevier

Single-Cycle Datapath: `lw` immediate

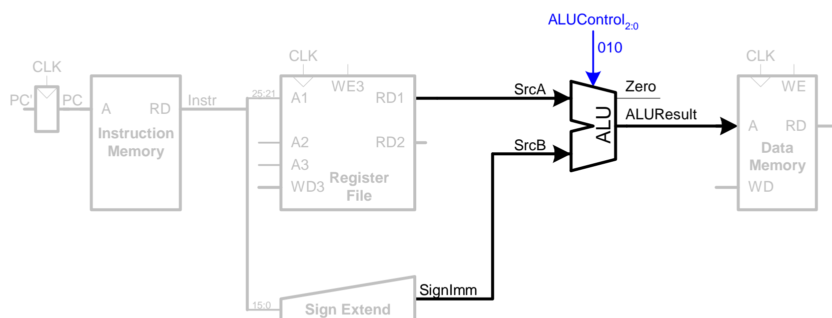
- **STEP 3:** Sign-extend the immediate



Copyright © 2007 Elsevier

Single-Cycle Datapath: `lw` address

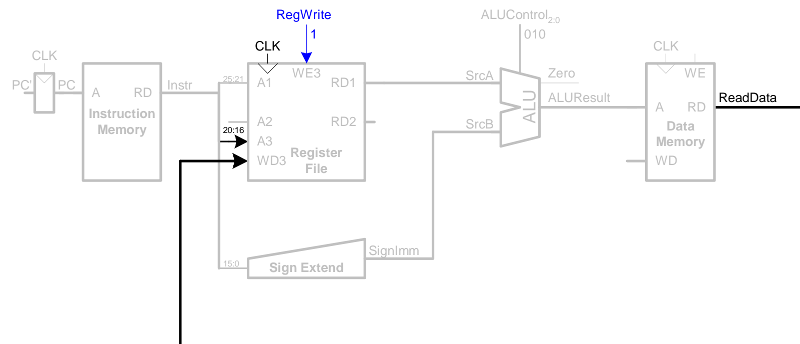
- **STEP 4:** Compute the memory address



Copyright © 2007 Elsevier

Single-Cycle Datapath: `lw` memory read

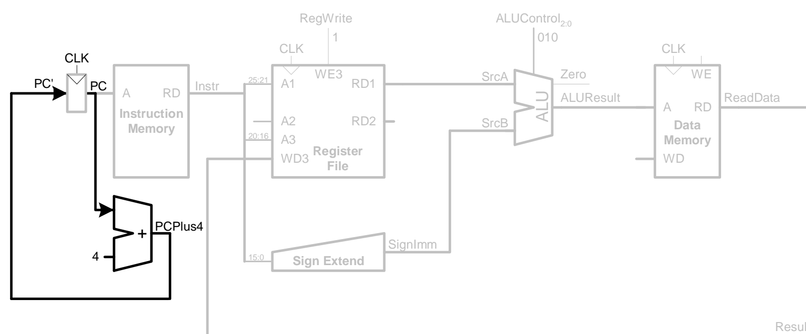
- **STEP 5:** Read data from memory and write it back to register file



Copyright © 2007 Elsevier 7-<15>

Single-Cycle Datapath: `lw` PC increment

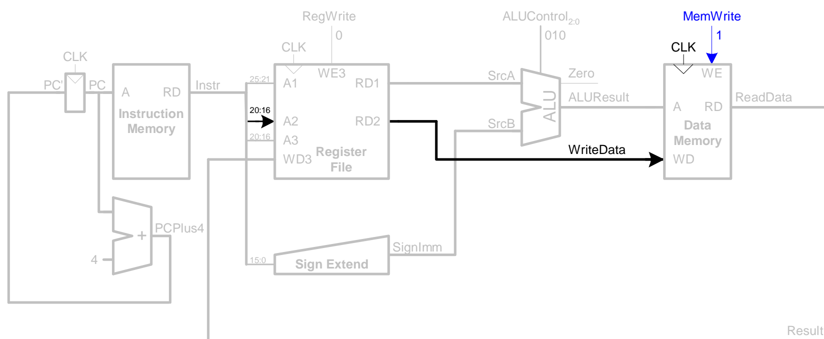
- **STEP 6:** Determine the address of the next instruction



Copyright © 2007 Elsevier 7-<16>

Single-Cycle Datapath: sw

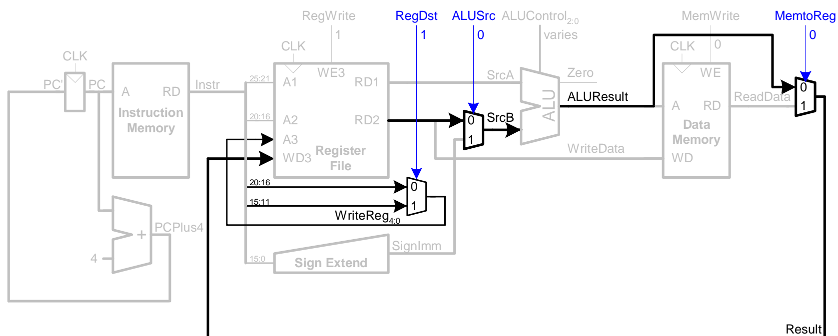
- Write data in rt to memory



Copyright © 2007 Elsevier

Single-Cycle Datapath: R-type instructions

- Read from rs and rt
- Write *ALUResult* to register file
- Write to rd (instead of rt)

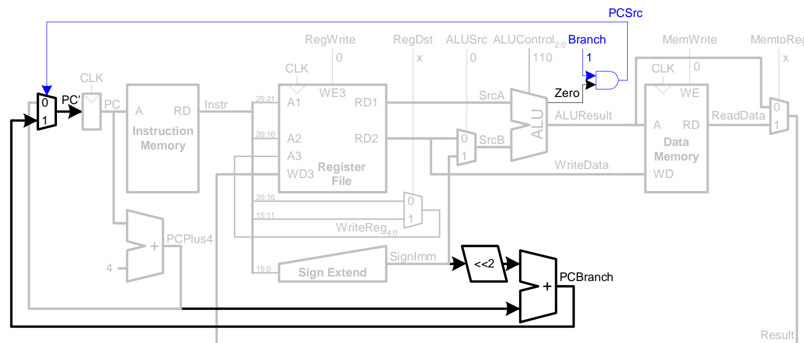


Copyright © 2007 Elsevier

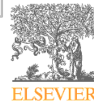
Single-Cycle Datapath: `beq`

- Determine whether values in `rs` and `rt` are equal
- Calculate branch target address:

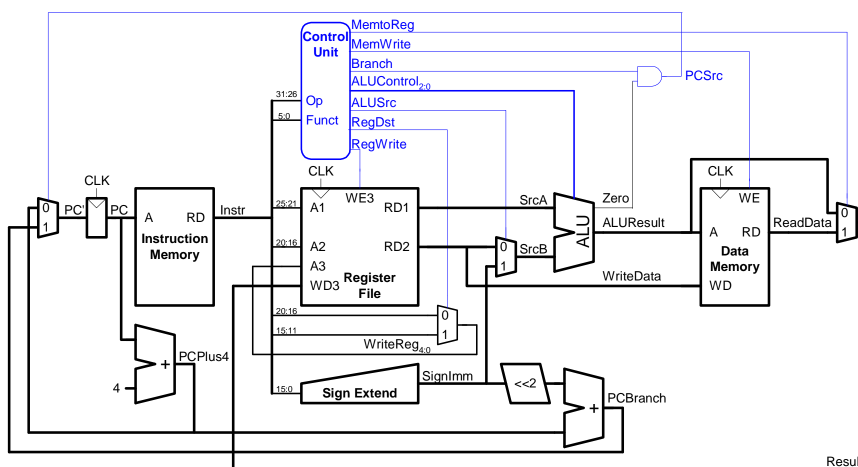
  BTA = (sign-extended immediate << 2) + (PC+4)



Copyright © 2007 Elsevier

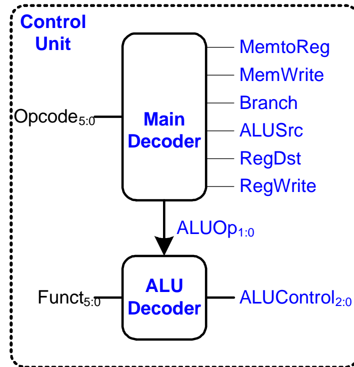


Complete Single-Cycle Processor



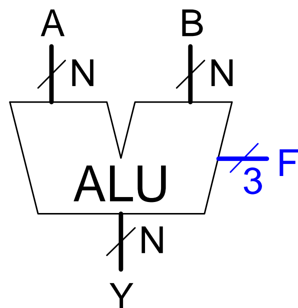
Copyright © 2007 Elsevier

Control Unit



Copyright © 2007 Elsevier

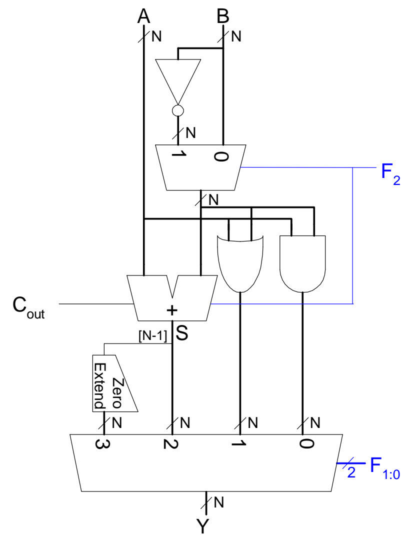
Review: ALU



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Copyright © 2007 Elsevier

Review: ALU



Copyright © 2007 Elsevier

Control Unit: ALU Decoder

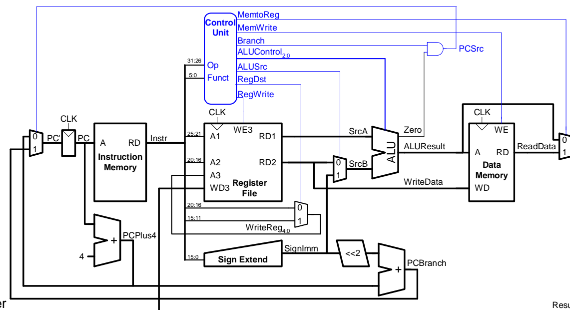
$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

Copyright © 2007 Elsevier

Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000							
lw	100011							
sw	101011							
beq	000100							



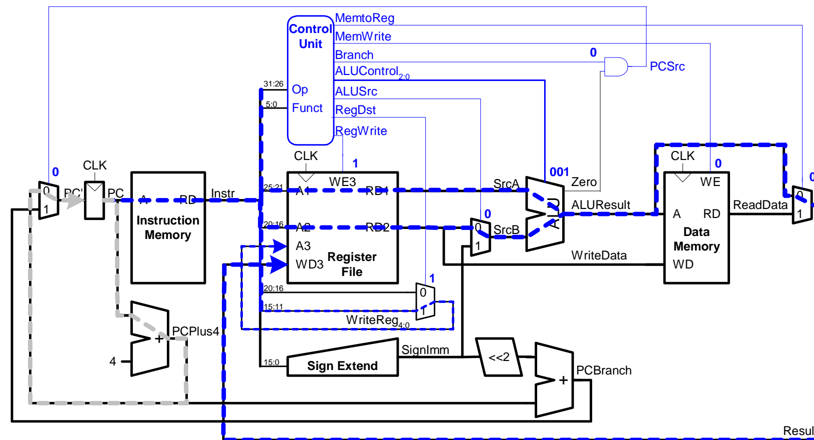
Copyright © 2007 Elsevier

Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	0	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01

Copyright © 2007 Elsevier

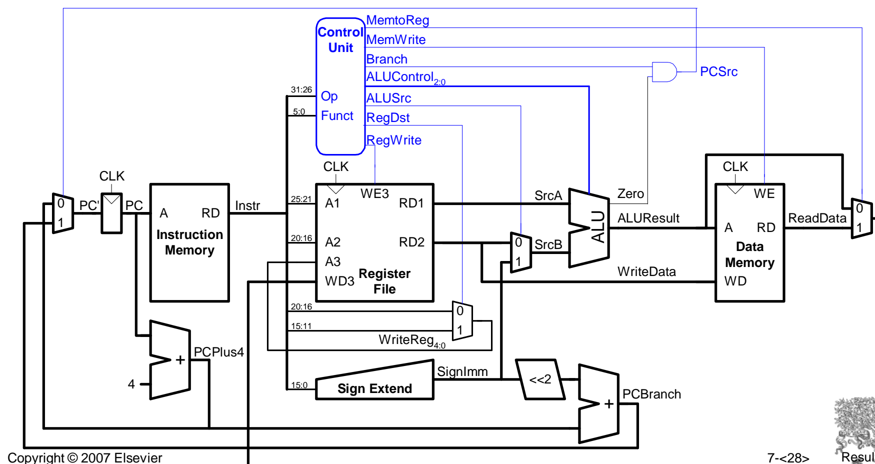
Single-Cycle Datapath Example: `or`



Copyright © 2007 Elsevier

Extended Functionality: `addi`

- No change to datapath



Copyright © 2007 Elsevier

Control Unit: addi

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000							

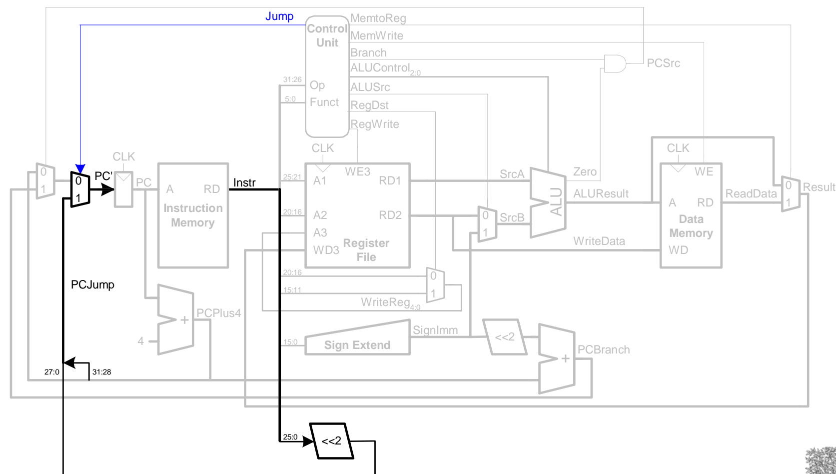
Copyright © 2007 Elsevier

Control Unit: addi

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00

Copyright © 2007 Elsevier

Extended Functionality: j



Copyright © 2007 Elsevier

Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000100								

Copyright © 2007 Elsevier

Control Unit: Main Decoder

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	00	0
`sw`	101011	0	X	1	0	1	X	00	0
`beq`	000100	0	X	0	1	0	X	01	0
`j`	**000100**	**0**	**X**	**X**	**X**	**0**	**X**	**XX**	**1**

Copyright © 2007 Elsevier

Review: Processor Performance

Program Execution Time

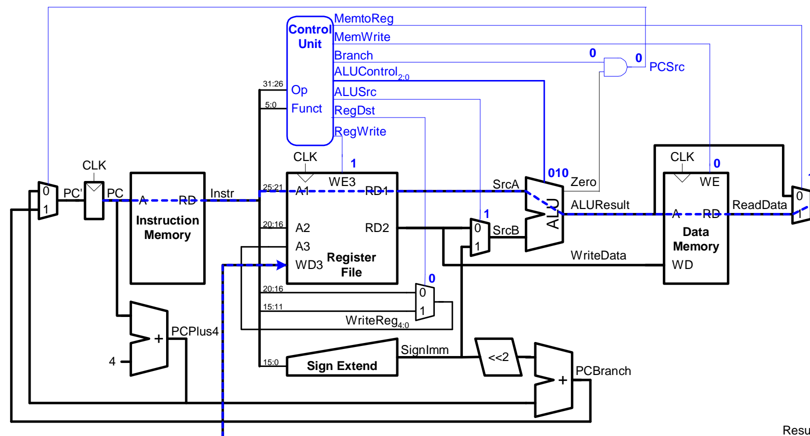
= (# instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

Copyright © 2007 Elsevier

Single-Cycle Performance

- T_C is limited by the critical path (`lw`)



Copyright © 2007 Elsevier

Single-Cycle Performance

- Single-cycle critical path:

  $T_c = t_{pcq_PC} + t_{\text{mem}} + \max(t_{RF\text{read}}, t_{sext} + t_{\text{mux}}) + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$

- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$

Copyright © 2007 Elsevier

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$T_c =$

Copyright © 2007 Elsevier

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$T_c = t_{pcq_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$
$= [30 + 2(250) + 150 + 25 + 200 + 20]$ ps
$= 925$ ps

Copyright © 2007 Elsevier

Single-Cycle Performance Example

- For a program with 100 billion instructions executing on a single-cycle MIPS processor,

Execution Time =

Copyright © 2007 Elsevier

Single-Cycle Performance Example

- For a program with 100 billion instructions executing on a single-cycle MIPS processor,

Execution Time = # instructions x CPI x T_C
= $(100 \times 10^{9})(1)(925 \times 10^{-12}\text{ s})$
= 92.5 seconds

Copyright © 2007 Elsevier

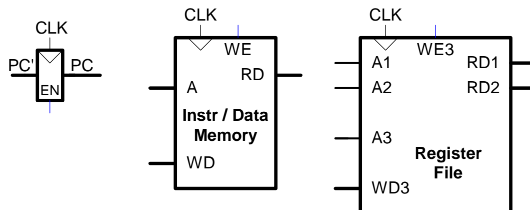
Multicycle MIPS Processor

- Single-cycle microarchitecture:
  - \+ simple
  - \- cycle time limited by longest instruction (`lw`)
  - \- two adders/ALUs and two memories
- Multicycle microarchitecture:
  - \+ higher clock speed
  - \+ simpler instructions run faster
  - \+ reuse expensive hardware on multiple cycles
  - \- sequencing overhead paid many times
- Same design steps: datapath & control

Copyright © 2007 Elsevier

Multicycle State Elements

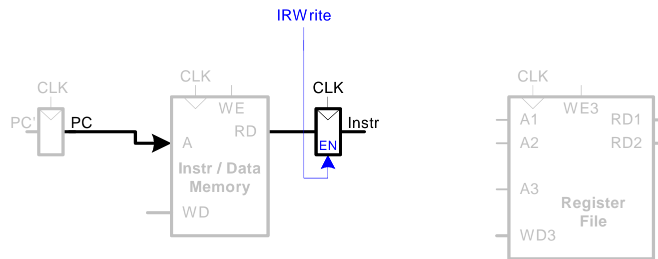
- Replace Instruction and Data memories with a single unified memory
  - – More realistic



Copyright © 2007 Elsevier

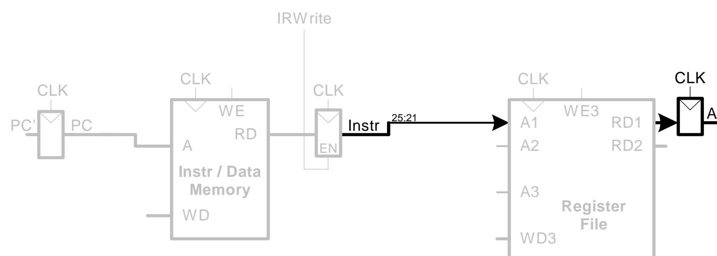
Multicycle Datapath: instruction fetch

- First consider executing `lw`
- **STEP 1:** Fetch instruction



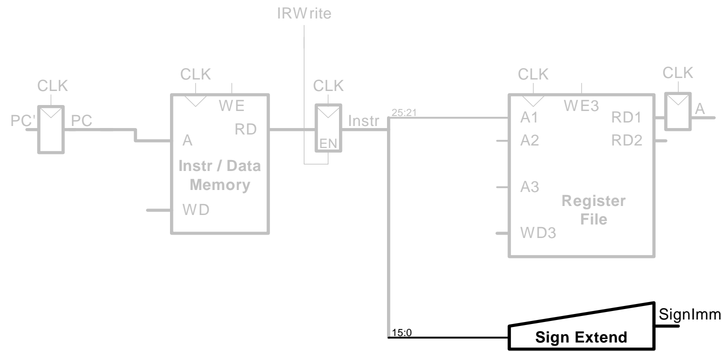
Copyright © 2007 Elsevier

Multicycle Datapath: `lw` register read



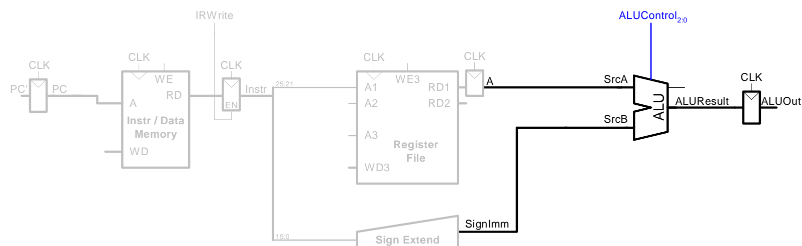
Copyright © 2007 Elsevier

Multicycle Datapath: `lw` immediate



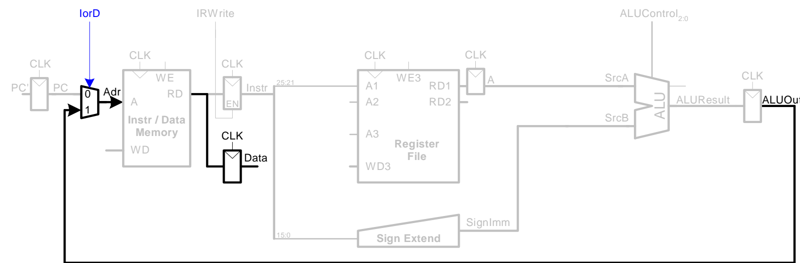
Copyright © 2007 Elsevier

Multicycle Datapath: `lw` address



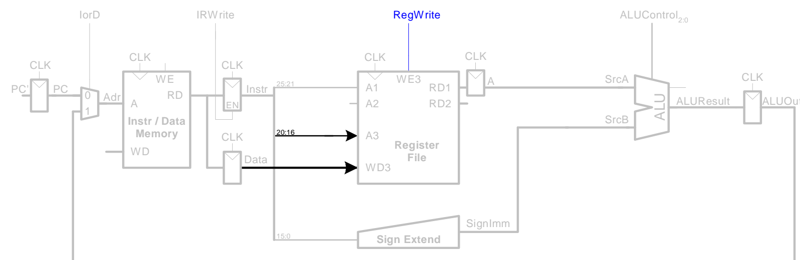
Copyright © 2007 Elsevier

Multicycle Datapath: `lw` memory read



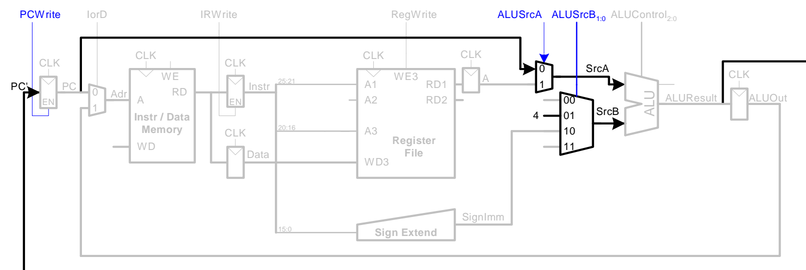
Copyright © 2007 Elsevier

Multicycle Datapath: `lw` write register



Copyright © 2007 Elsevier

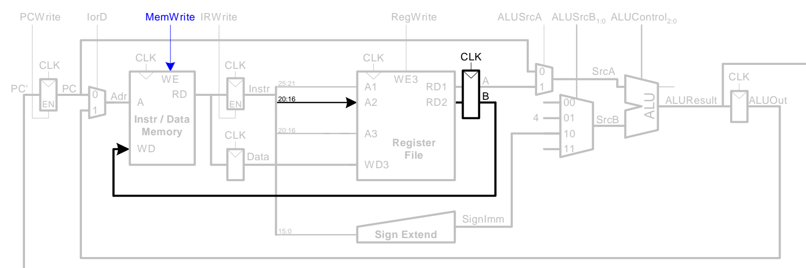
Multicycle Datapath: increment PC



Copyright © 2007 Elsevier

Multicycle Datapath: `sw`

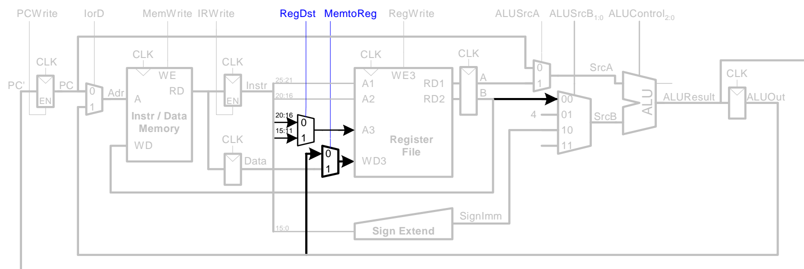
- Write data in `rt` to memory



Copyright © 2007 Elsevier

Multicycle Datapath: R-type Instructions

- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)

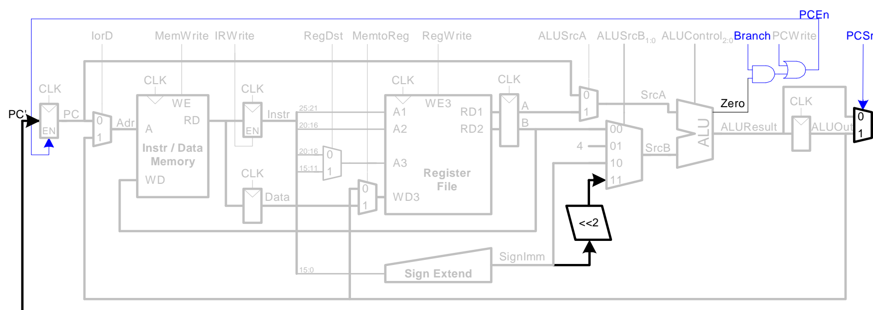


Copyright © 2007 Elsevier

Multicycle Datapath: `beq`

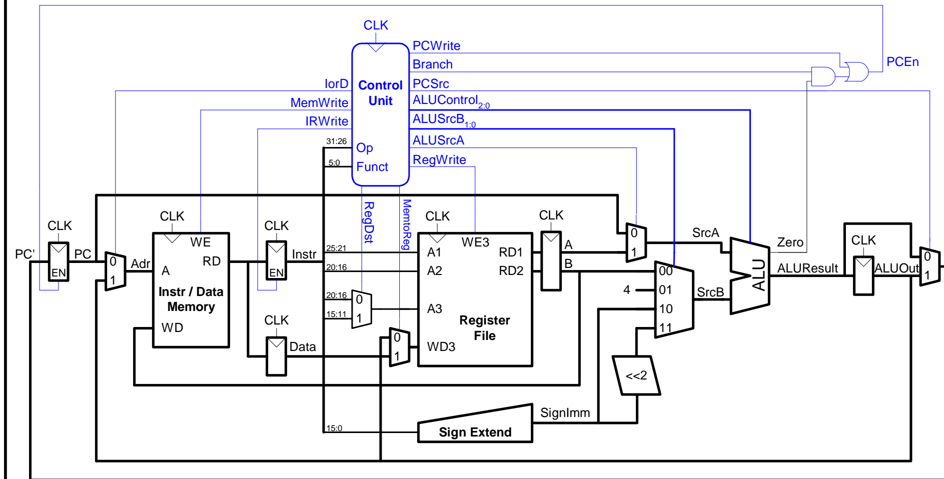
- Determine whether values in `rs` and `rt` are equal
- Calculate branch target address:

  BTA = (sign-extended immediate << 2) + (PC+4)



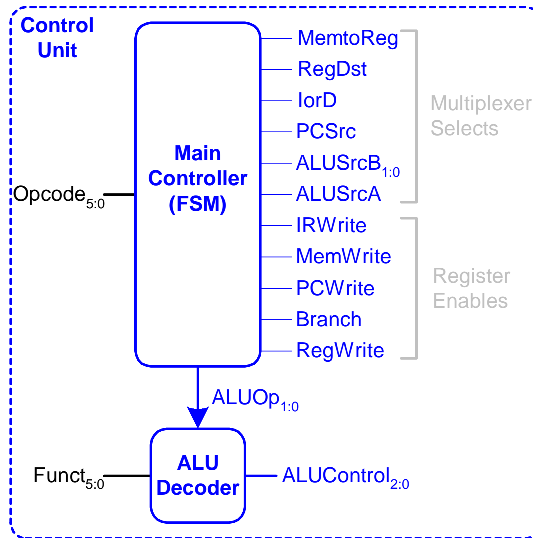
Copyright © 2007 Elsevier

Complete Multicycle Processor



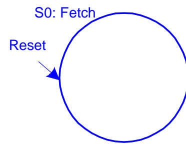
Copyright © 2007 Elsevier

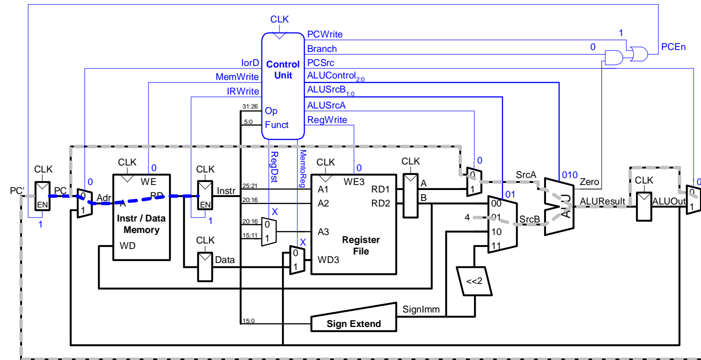
Control Unit



Copyright © 2007 Elsevier

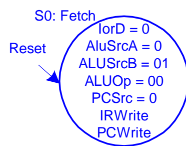
Main Controller FSM: Fetch

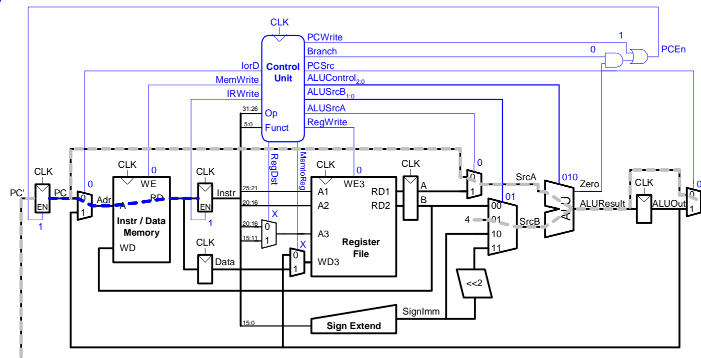




Copyright © 2007 Elsevier

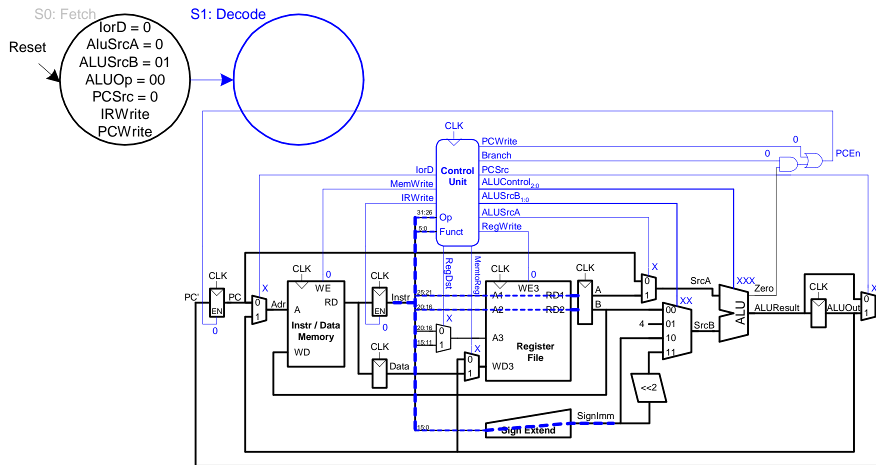
Main Controller FSM: Fetch





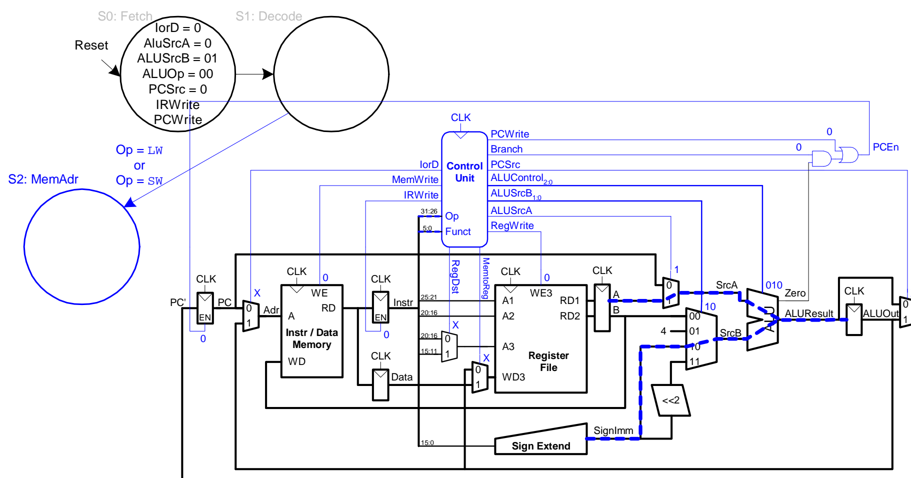
Copyright © 2007 Elsevier

Main Controller FSM: Decode



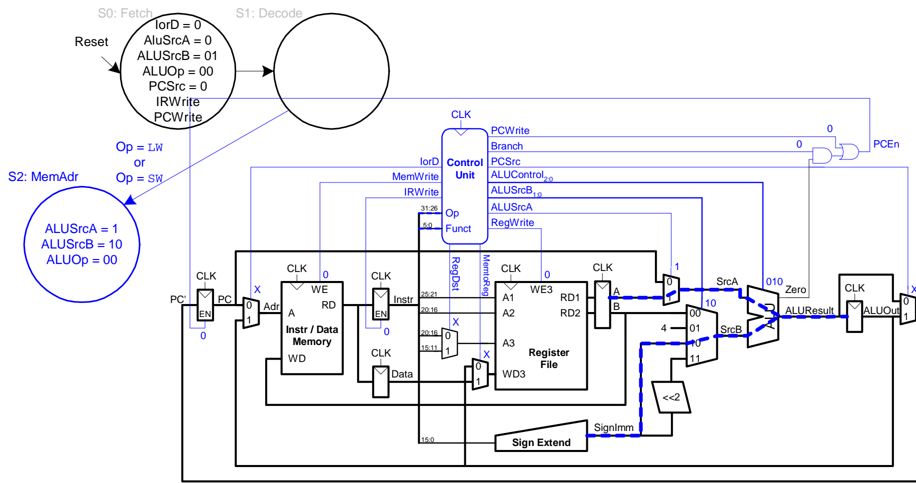
Copyright © 2007 Elsevier

Main Controller FSM: Address Calculation



Copyright © 2007 Elsevier

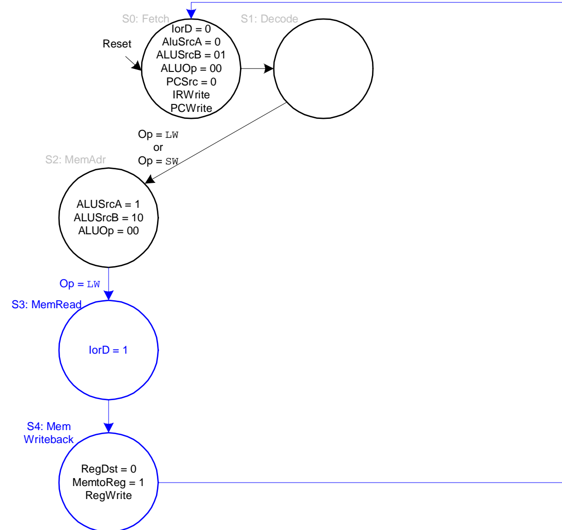
Main Controller FSM: Address Calculation



Copyright © 2007 Elsevier

7-<59>

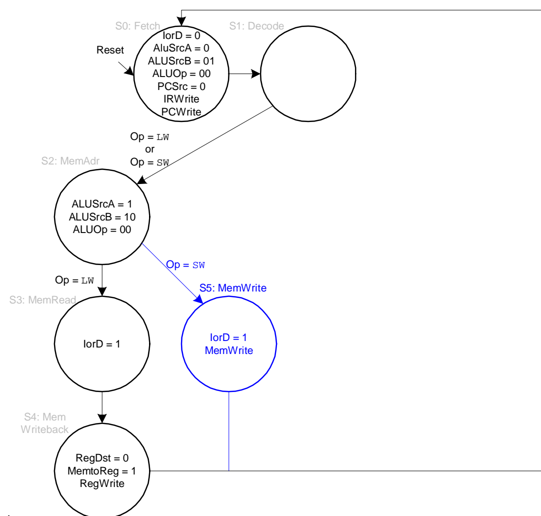
Main Controller FSM: lw



Copyright © 2007 Elsevier

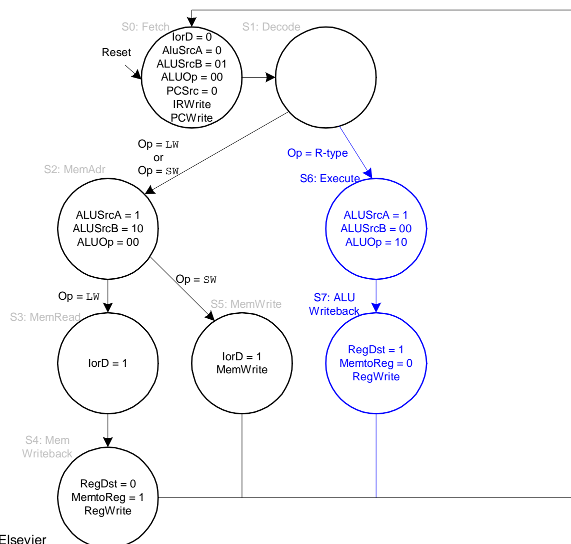
7-<60>

Main Controller FSM: `sw`



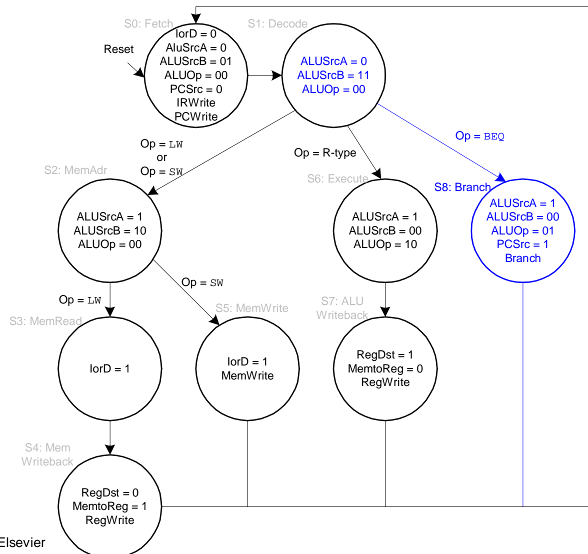
Copyright © 2007 Elsevier

Main Controller FSM: `R-Type`



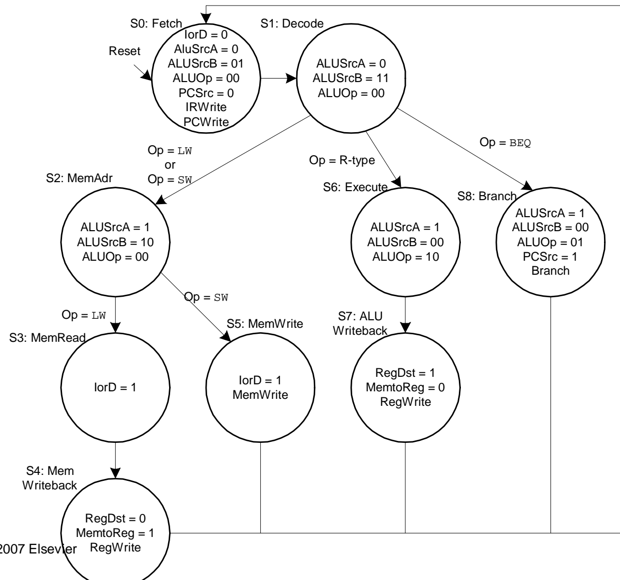
Copyright © 2007 Elsevier

Main Controller FSM: beq



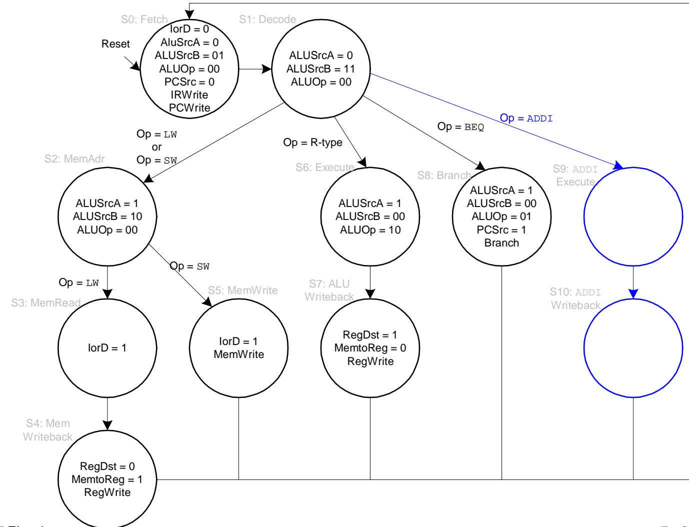
Copyright © 2007 Elsevier

Complete Multicycle Controller FSM



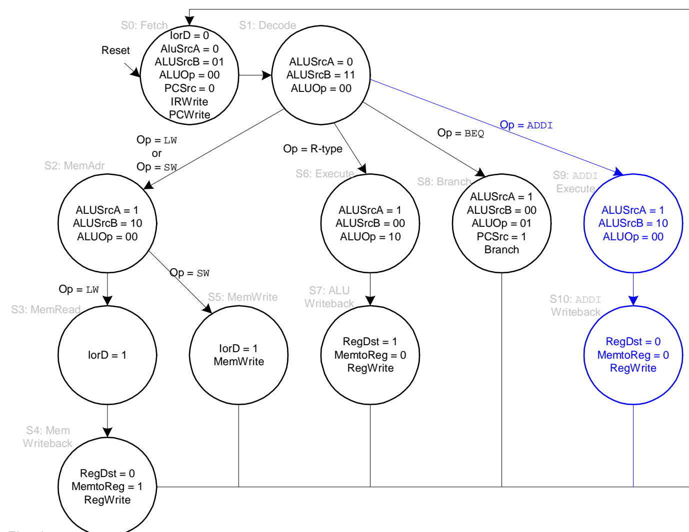
Copyright © 2007 Elsevier

Main Controller FSM: `addi`



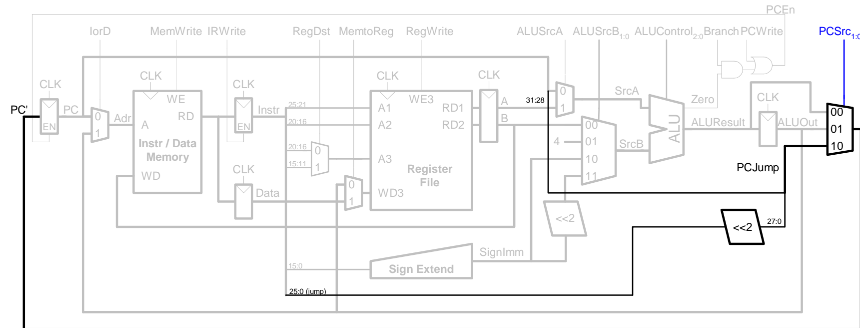
Copyright © 2007 Elsevier

Main Controller FSM: `addi`



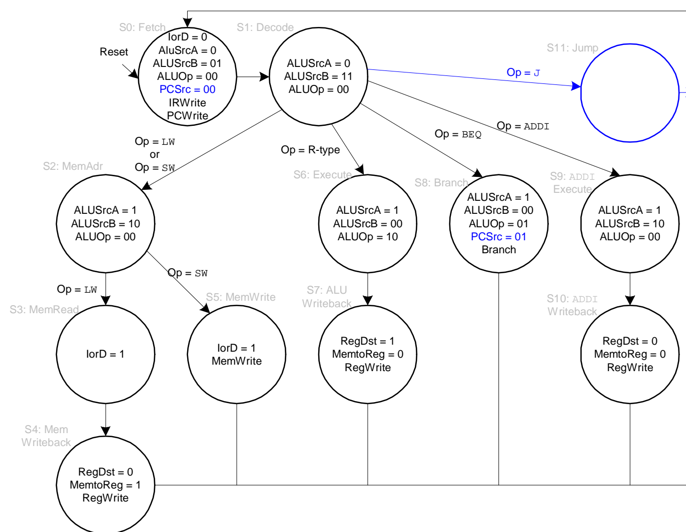
Copyright © 2007 Elsevier

Extended Functionality: j



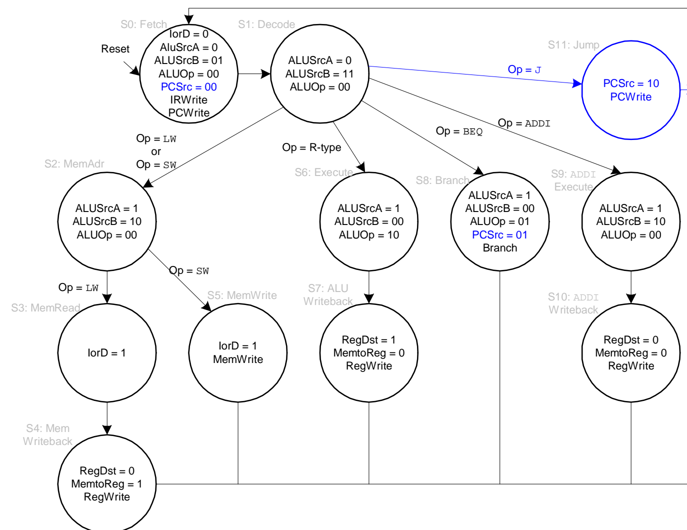
Copyright © 2007 Elsevier

Control FSM: j



Copyright © 2007 Elsevier

Control FSM: `j`



Copyright © 2007 Elsevier

Multicycle Performance

- Instructions take different number of cycles:
  - 3 cycles: `beq`, `j`
  - 4 cycles: R-Type, `sw`, `addi`
  - 5 cycles: `lw`
- CPI is weighted average
- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type

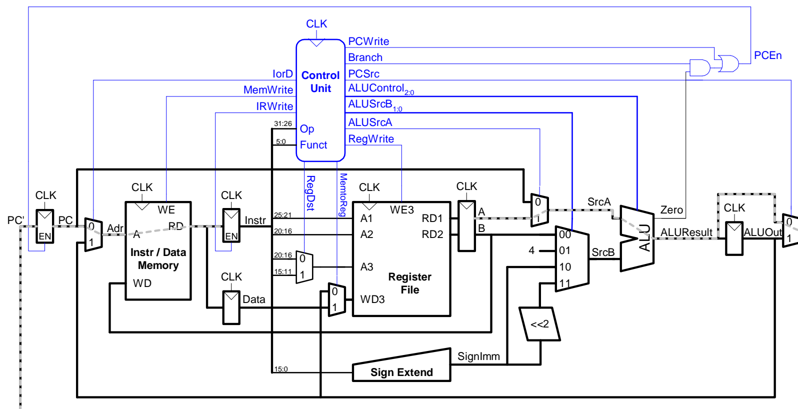
Average CPI = (0.11 + 0.2)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12

Copyright © 2007 Elsevier

Multicycle Performance

- Multicycle critical path:

  $T_c =$

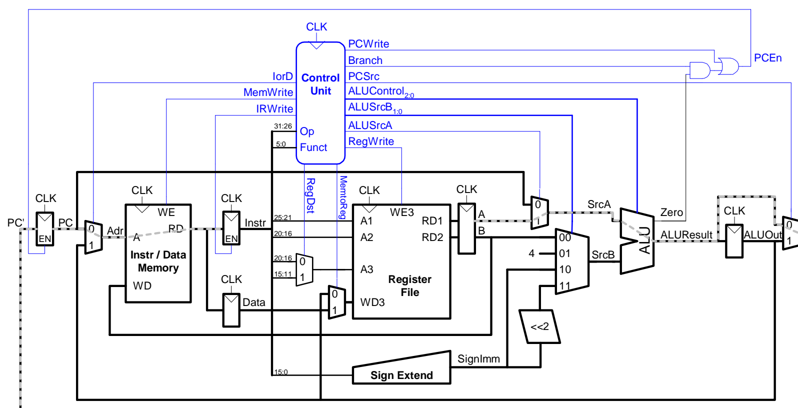


Copyright © 2007 Elsevier

Multicycle Performance

- Multicycle critical path:

  $T_c = t_{pcq} + t_{\text{mux}} + \max(t_{\text{ALU}} + t_{\text{mux}}, t_{\text{mem}}) + t_{\text{setup}}$



Copyright © 2007 Elsevier

Multicycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\mathrm{read}}$	150
Register file setup	$t_{RF\mathrm{setup}}$	20

$T_c =$

Copyright © 2007 Elsevier

Multicycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\mathrm{read}}$	150
Register file setup	$t_{RF\mathrm{setup}}$	20

$$\begin{aligned} T_c &= t_{pcq_PC} + t_{\mathrm{mux}} + \max(t_{\mathrm{ALU}} + t_{\mathrm{mux}}, t_{\mathrm{mem}}) + t_{\mathrm{setup}} \\ &= t_{pcq_PC} + t_{\mathrm{mux}} + t_{\mathrm{mem}} + t_{\mathrm{setup}} \\ &= [30 + 25 + 250 + 20] \text{ ps} \\ &= 325 \text{ ps} \end{aligned}$$

Copyright © 2007 Elsevier

Multicycle Performance Example

- For a program with 100 billion instructions executing on a multicycle MIPS processor
  - CPI = 4.12
  - T_c = 325 ps

Execution Time =

Copyright © 2007 Elsevier

Multicycle Performance Example

- For a program with 100 billion instructions executing on a multicycle MIPS processor
  - CPI = 4.12
  - T_c = 325 ps

Execution Time = (# instructions) × CPI × T_c
= $(100 \times 10^9)(4.12)(325 \times 10^{-12})$
= 133.9 seconds

- This is slower than the single-cycle processor (92.5 seconds). Why?

Copyright © 2007 Elsevier

Multicycle Performance Example

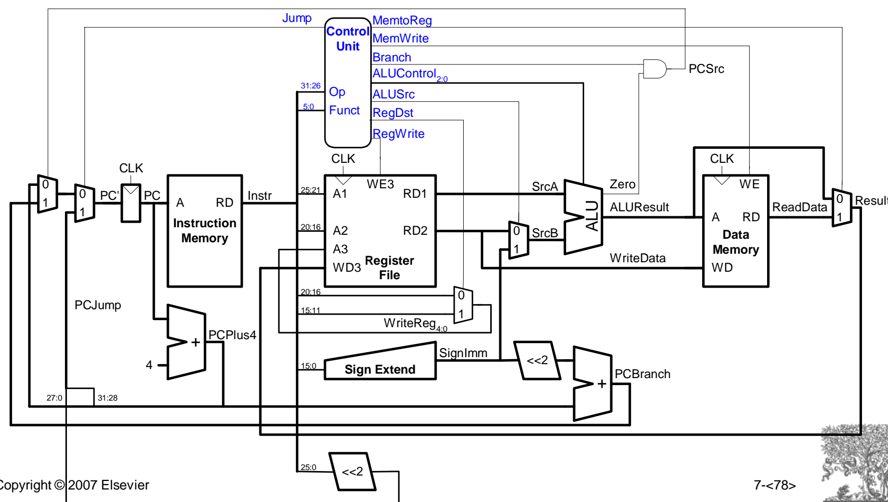
- For a program with 100 billion instructions executing on a multicycle MIPS processor
  - CPI = 4.12
  - T_c = 325 ps

Execution Time = (# instructions) × CPI × T_c
= $(100 \times 10^9)(4.12)(325 \times 10^{-12})$
= 133.9 seconds

- This is slower than the single-cycle processor (92.5 seconds). Why?
  - Not all steps the same length
  - Sequencing overhead for each step ($t_{pcq} + t_{setup}$= 50 ps)

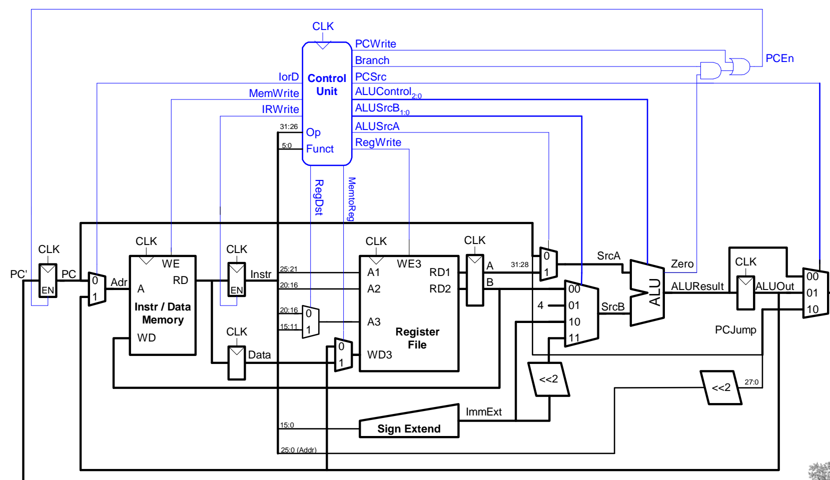
Copyright © 2007 Elsevier

Review: Single-Cycle MIPS Processor



Copyright © 2007 Elsevier

Review: Multicycle MIPS Processor



Copyright © 2007 Elsevier

7-<79>

Pipelined MIPS Processor

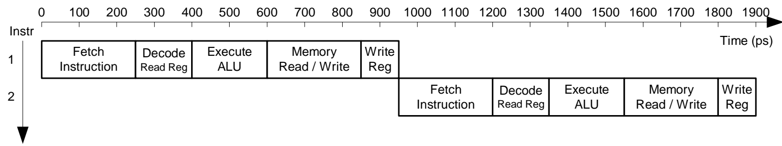
- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

Copyright © 2007 Elsevier

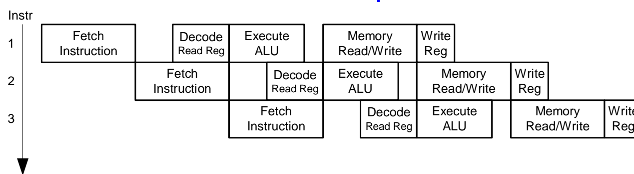
7-<80>

Single-Cycle vs. Pipelined Performance

Single-Cycle

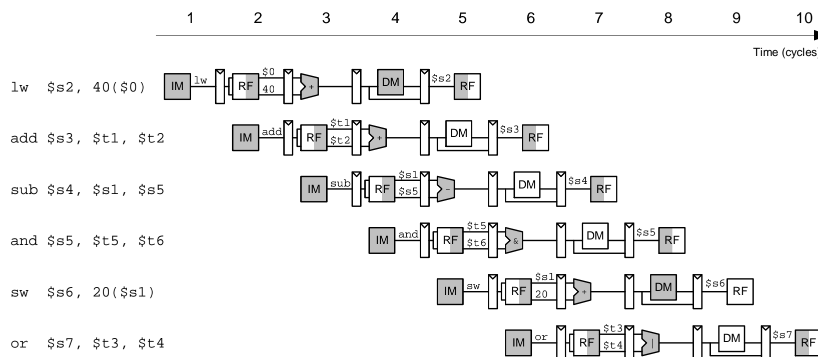


Pipelined



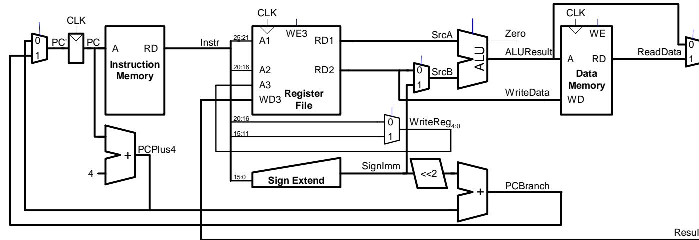
Copyright © 2007 Elsevier

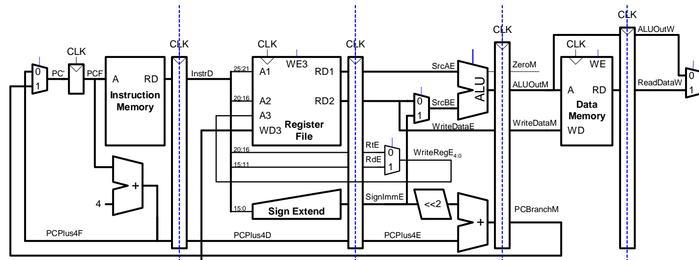
Pipelining Abstraction



Copyright © 2007 Elsevier

Single-Cycle and Pipelined Datapath

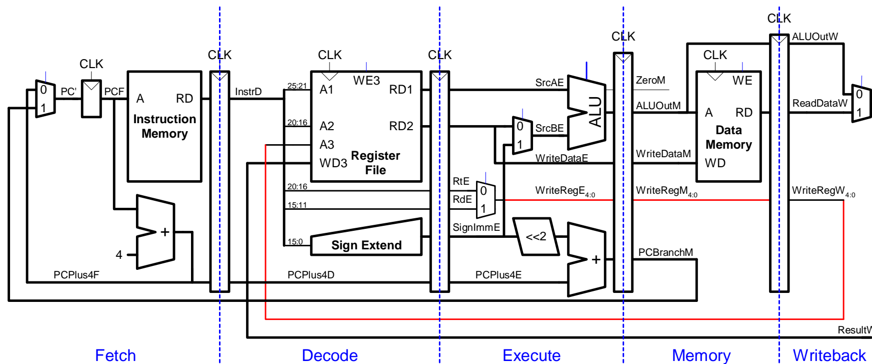




Copyright © 2007 Elsevier

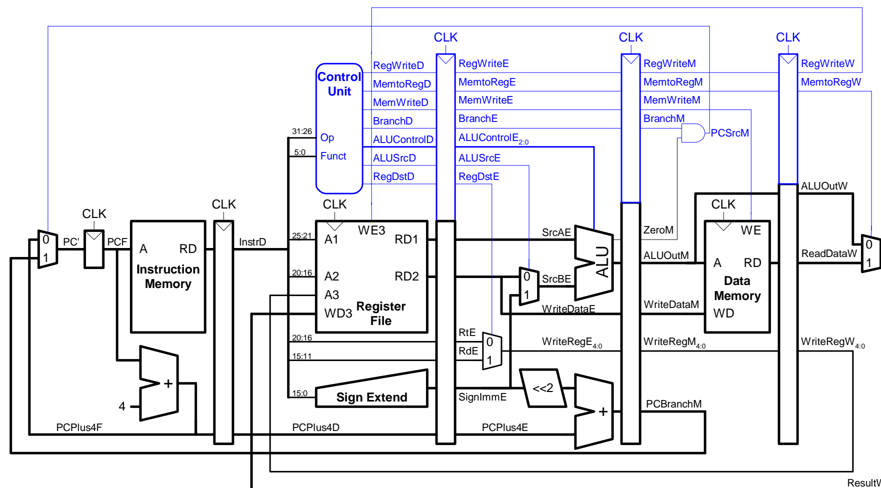
Corrected Pipelined Datapath

- WriteReg must arrive at the same time as Result



Copyright © 2007 Elsevier

Pipelined Control



Same control unit as single-cycle processor

Control delayed to proper pipeline stage

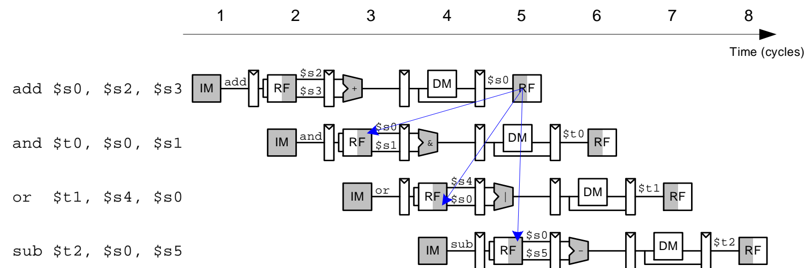
Copyright © 2007 Elsevier

Pipeline Hazard

- Occurs when an instruction depends on results from previous instruction that hasn’t completed.
- Types of hazards:
  - **Data hazard:** register value not written back to register file yet
  - **Control hazard:** next instruction not decided yet (caused by branches)

Copyright © 2007 Elsevier

Data Hazard



Copyright © 2007 Elsevier

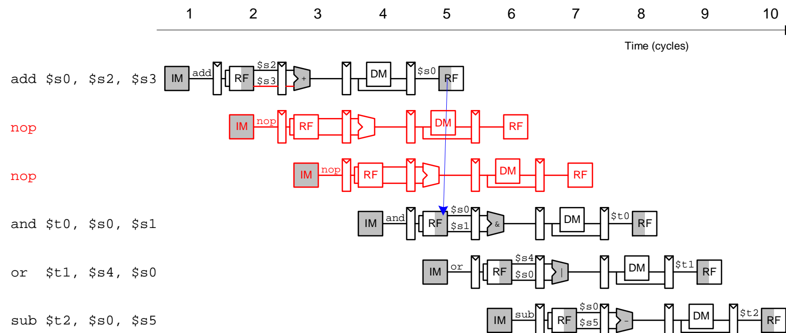
Handling Data Hazards

- Insert nops in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Copyright © 2007 Elsevier

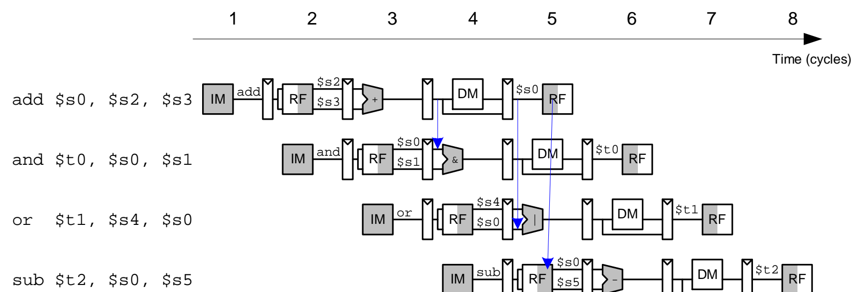
Compile-Time Hazard Elimination

- Insert enough nops for result to be ready
- Or move independent useful instructions forward



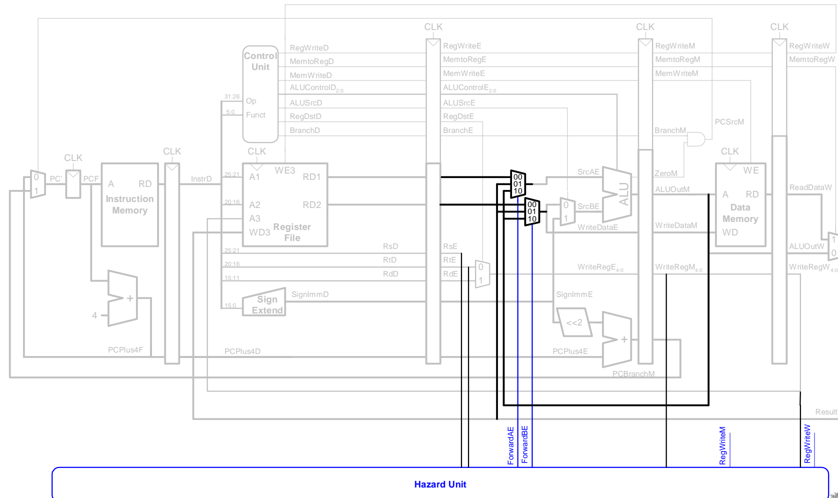
Copyright © 2007 Elsevier

Data Forwarding



Copyright © 2007 Elsevier

Data Forwarding



Copyright © 2007 Elsevier

Data Forwarding

- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage

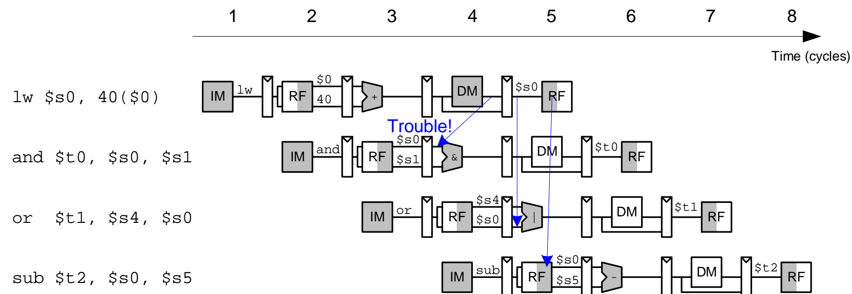
- Forwarding logic for *ForwardAE*:

```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
then    ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
then      ForwardAE = 01
else      ForwardAE = 00
```

- Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*

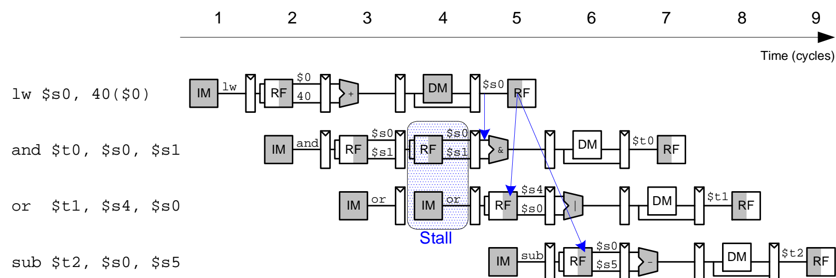
Copyright © 2007 Elsevier

Stalling



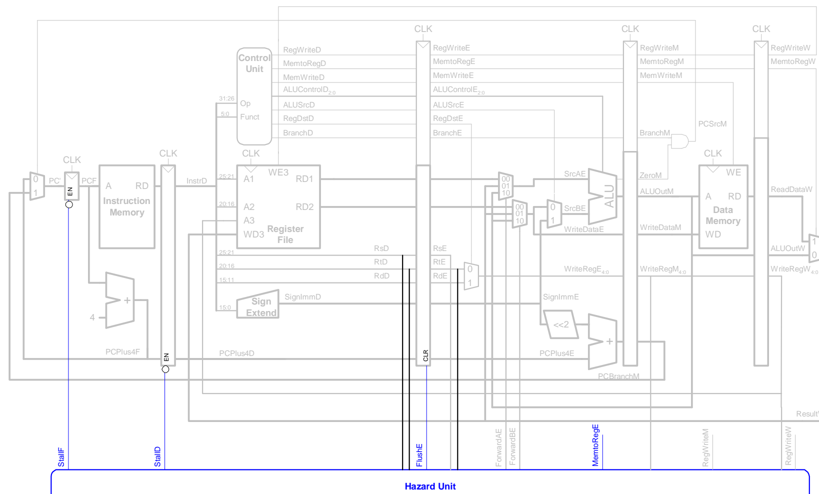
Copyright © 2007 Elsevier

Stalling



Copyright © 2007 Elsevier

Stalling Hardware



Copyright © 2007 Elsevier

Stalling Hardware

- Stalling logic:

```
lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE

StallF = StallD = FlushE = lwstall
```

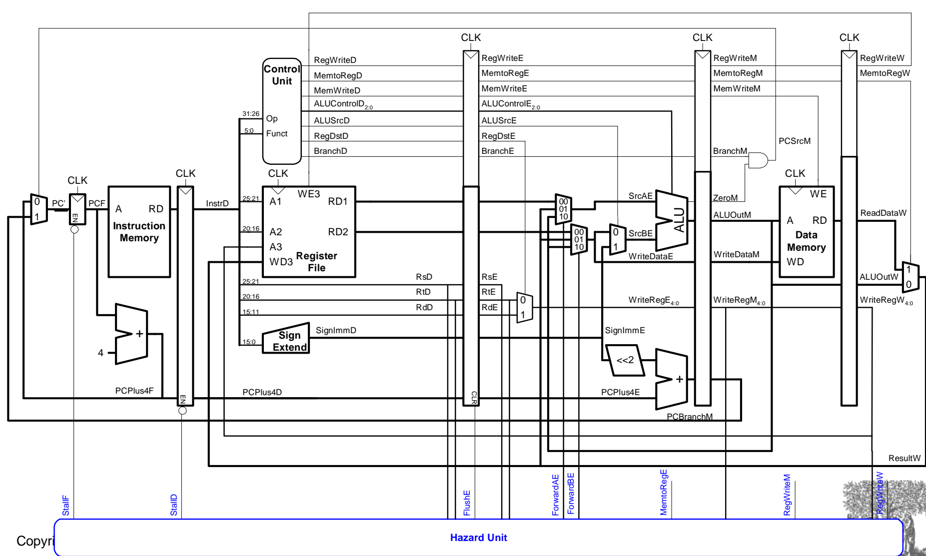
Copyright © 2007 Elsevier

Control Hazards

- beq:
  - branch is not determined until the fourth stage of the pipeline
  - Instructions after the branch are fetched before branch occurs
  - These instructions must be flushed if the branch happens
- Branch misprediction penalty
  - number of instruction flushed when branch is taken
  - May be reduced by determining branch earlier

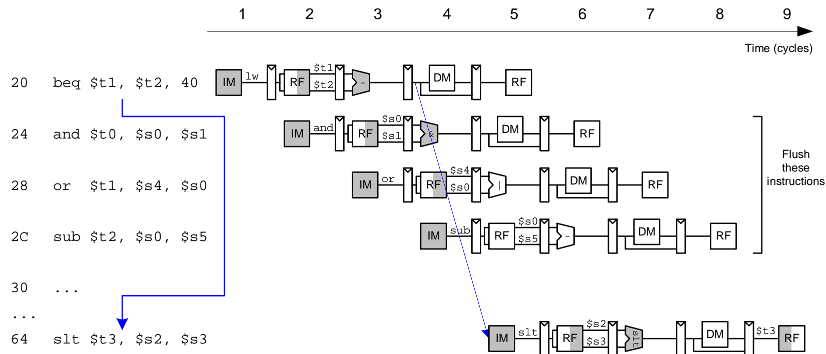
Copyright © 2007 Elsevier

Control Hazards: Original Pipeline



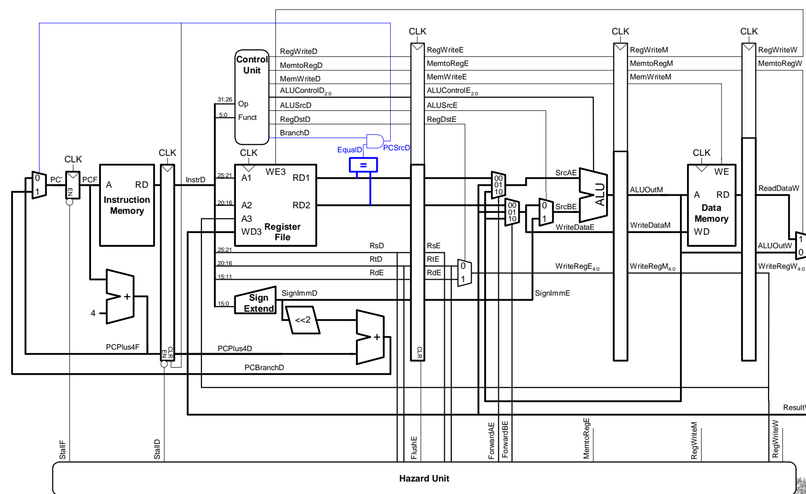
Copyri

Control Hazards



Copyright © 2007 Elsevier

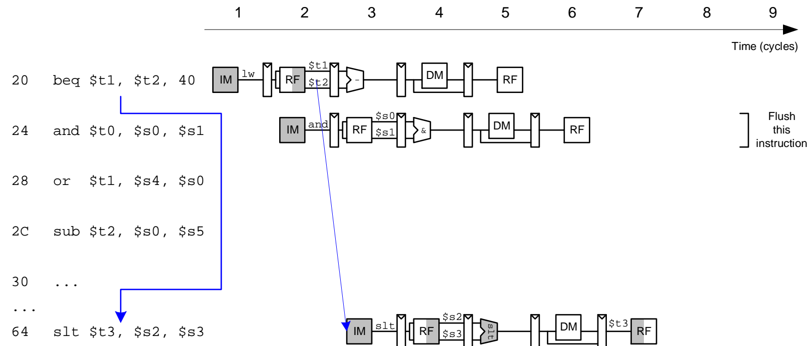
Control Hazards: Early Branch Resolution



Copyright © 2007 Elsevier

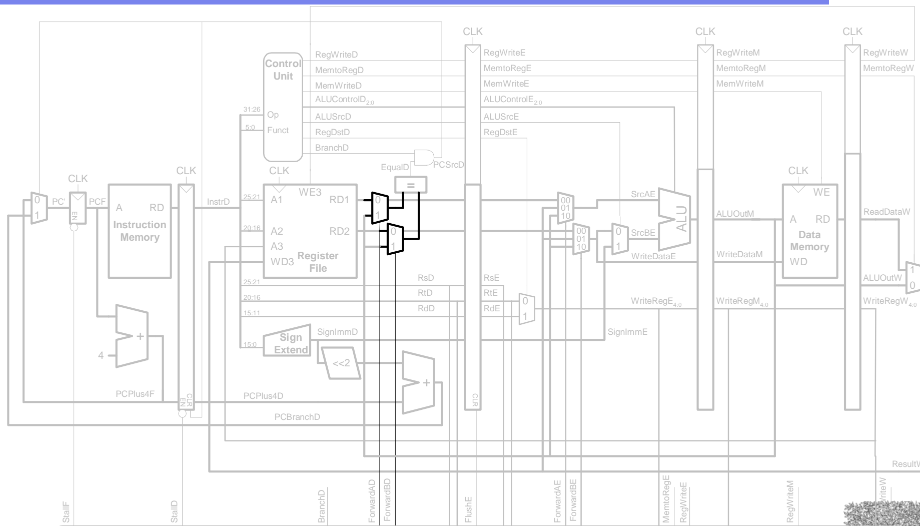
Introduced another data hazard in Decode stage

Control Hazards with Early Branch Resolution



Copyright © 2007 Elsevier

Handling Data and Control Hazards



Control Forwarding and Stalling Hardware

- Forwarding logic:

```
ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND RegWriteM
```

- Stalling logic:

```
branchstall = BranchD AND RegWriteE AND
                (WriteRegE == rsD OR WriteRegE == rtD)
              OR
              BranchD AND MemtoRegM AND
                (WriteRegM == rsD OR WriteRegM == rtD)

StallF = StallD = FlushE = lwstall OR branchstall
```

Copyright © 2007 Elsevier

Branch Prediction

- Guess whether branch will be taken
  - Backward branches are usually taken (loops)
  - Perhaps consider history of whether branch was previously taken to improve the guess
- Good prediction reduces the fraction of branches requiring a flush

Copyright © 2007 Elsevier

Pipelined Performance Example

- Ideally CPI = 1
- But need to handle stalling (caused by loads and branches)
- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
- **What is the average CPI?**

Copyright © 2007 Elsevier

Pipelined Performance Example

- SPECINT2000 benchmark:
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- Suppose:
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
  - All jumps flush next instruction
- **What is the average CPI?**
  - Load/Branch CPI = 1 when no stalling, 2 when stalling. Thus,
  - $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
  - $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$
  - Thus,

Average CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1)

= 1.15

Copyright © 2007 Elsevier

Pipelined Performance

- Pipelined processor critical path:

$$
\begin{aligned}
T_c = \max\{ & \\
& t_{pcq} + t_{\text{mem}} + t_{\text{setup}} \\
& 2(t_{RFread} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + t_{\text{mux}} + t_{\text{setup}}) \\
& t_{pcq} + t_{\text{mux}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{setup}} \\
& t_{pcq} + t_{\text{memwrite}} + t_{\text{setup}} \\
& 2(t_{pcq} + t_{\text{mux}} + t_{\text{RFwrite}}) \}
\end{aligned}
$$

Copyright © 2007 Elsevier

Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	$t_{RF\text{read}}$	150
Register file setup	$t_{RF\text{setup}}$	20
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	T_{memwrite}	220
Register file write	$t_{RF\text{write}}$	100 ps

$$
\begin{aligned}
T_c &= 2(t_{RFread} + t_{\text{mux}} + t_{\text{eq}} + t_{\text{AND}} + t_{\text{mux}} + t_{\text{setup}}) \\
&= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps} = 550 \text{ ps}
\end{aligned}
$$

Copyright © 2007 Elsevier

Pipelined Performance Example

- For a program with 100 billion instructions executing on a pipelined MIPS processor,
- CPI = 1.15
- T_c = 550 ps

Execution Time = (# instructions) × CPI × T_c
= $(100 \times 10^9)(1.15)(550 \times 10^{-12})$
= 63 seconds

Processor	Execution Time (seconds)	Speedup (single-cycle is baseline)
Single-cycle	95	1
Multicycle	133	0.71
Pipelined	63	1.51

Copyright © 2007 Elsevier

Review: Exceptions

- Unscheduled procedure call to the *exception handler*
- Casued by:
  - Hardware, also called an *interrupt*, e.g. keyboard
  - Software, also called *traps*, e.g. undefined instruction
- When exception occurs, the processor:
  - Records the cause of the exception (`Cause` register)
  - Jumps to the exception handler at instruction address 0x80000180
  - Returns to program (`EPC` register)

Copyright © 2007 Elsevier

Example Exception

sequential circuits.¶

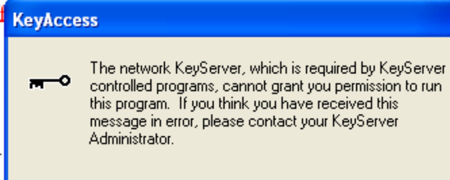
Can we design a spiff

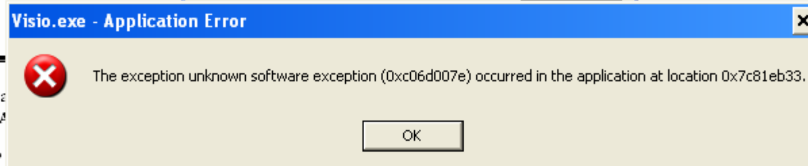
Figure 2.1 shows a

inputs, A and B, and on

box indicates that it is

this case, the function is





Harris a

—26 A

CHAP

words, we say the output Y is a function of the two inputs A and B where the function performed is A OR B.¶

The *implementation* of the combinational circuit is independent of its functionality. Figure 2.1 and Figure 2.2 show two possible implementa-

Copyright © 2007 Elsevier

Exception Registers

- Not part of the register file.
  - `Cause`
    - Records the cause of the exception
    - Coprocessor 0 register 13
  - `EPC` (Exception PC)
    - Records the PC where the exception occurred
    - Coprocessor 0 register 14
- Move from Coprocessor 0
  - `mfc0 $t0, Cause`
  - Moves the contents of `Cause` into `$t0`

mfc0

010000	00000	`$t0` (8)	`Cause` (13)	00000000000
31:26	25:21	20:16	15:11	10:0

Copyright © 2007 Elsevier

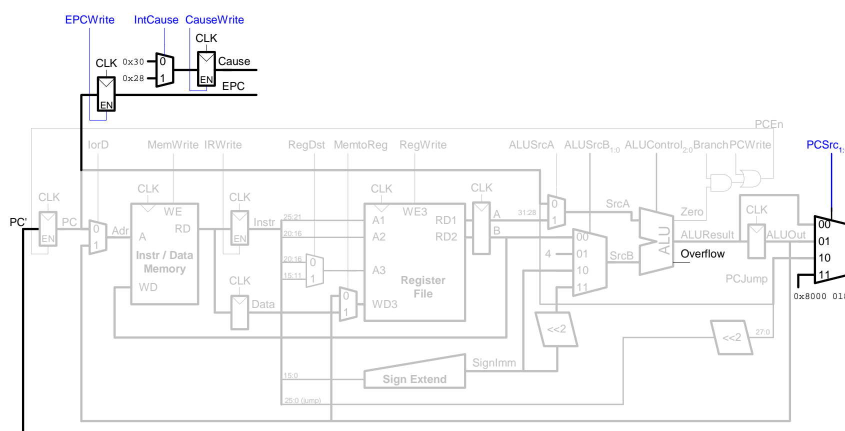
Exception Causes

Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

We extend the multicycle MIPS processor to handle the last two types of exceptions.

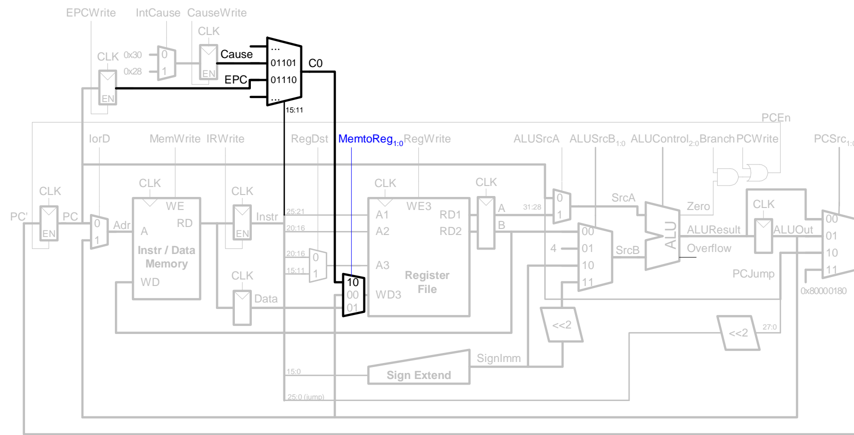
Copyright © 2007 Elsevier

Exception Hardware: EPC & Cause



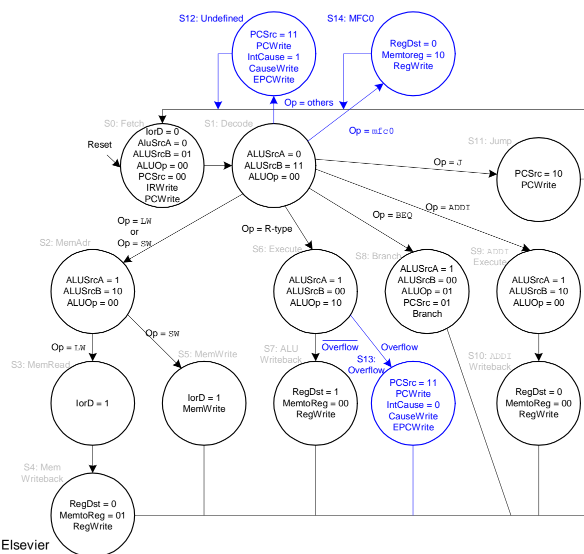
Copyright © 2007 Elsevier

Exception Hardware: `mfc0`



Copyright © 2007 Elsevier

Control FSM with Exceptions



Copyright © 2007 Elsevier

Advanced MicroArchitecture

- Deep Pipelining
- Branch Prediction
- Superscalar Processors
- Out of Order Processors
- Register Renaming
- SIMD
- Multithreading
- Multiprocessors

Copyright © 2007 Elsevier

Deep Pipelining

- 10-20 stages typical
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost

Copyright © 2007 Elsevier

Branch Prediction

- Ideal pipelined processor: CPI = 1
- Branch misprediction increases CPI
- Static branch prediction:
  - Check direction of branch (forward or backward)
  - If backward, predict taken
  - Otherwise, predict not taken
- Dynamic branch prediction:
  - Keep history of last (several hundred) branches in a *branch target buffer* which holds:
    - Branch destination
    - Whether branch was taken

Copyright © 2007 Elsevier

Branch Prediction Example

```
  add  $s1, $0, $0      # sum = 0
  add  $s0, $0, $0      # i   = 0
  addi $t0, $0, 10      # $t0 = 10
for:
  beq  $t0, $t0, done   # if i == 10, branch
  add  $s1, $s1, $s0    # sum = sum + i
  addi $s0, $s0, 1      # increment i
  j    for
done:
```

Copyright © 2007 Elsevier

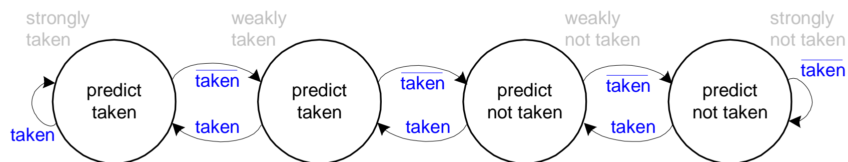
1-Bit Branch Predictor

- Remembers whether branch was taken the last time and does the same thing
- Mispredicts first and last branch of loop

Copyright © 2007 Elsevier

2-Bit Branch Predictor

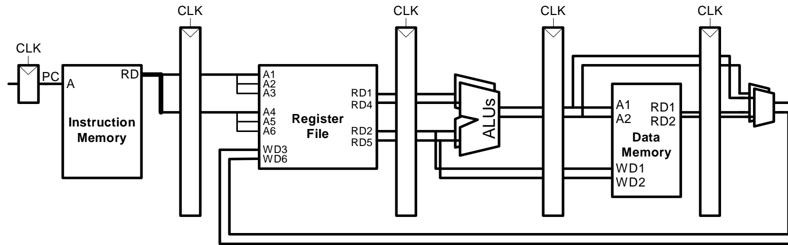
- Only mispredicts last branch of loop



Copyright © 2007 Elsevier

Superscalar

- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once



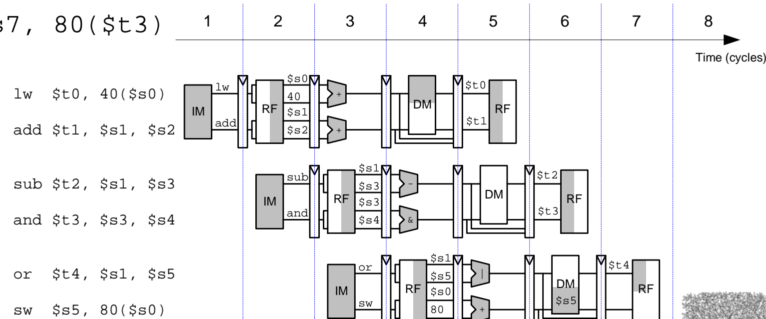
Copyright © 2007 Elsevier

Superscalar Example

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

Ideal IPC: 2

Actual IPC: 2



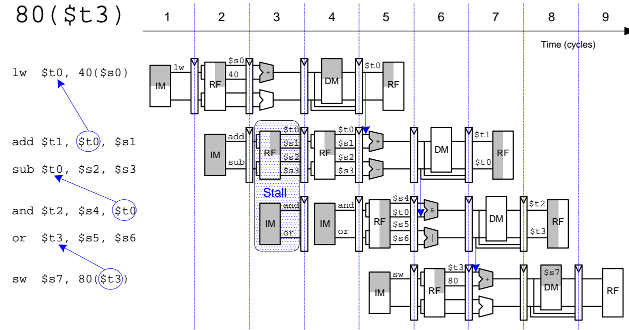
Copyright © 2007 Elsevier

Superscalar Example with Dependencies

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Ideal IPC: 2

Actual IPC: 6/5 = 1.17



Copyright © 2007 Elsevier

Out of Order Processor

- Looks ahead across multiple instructions to issue as many as possible at once
- Issues instructions out of order as long as no dependencies
- Dependencies:
  - **RAW** (read after write): one instruction writes, and later instruction reads a register
  - **WAR** (write after read): one instruction reads, and a later instruction writes a register (also called an *antidependence*)
  - **WAW** (write after write): one instruction writes, and a later instruction writes a register (also called an *output dependence*)

Copyright © 2007 Elsevier

Out of Order Processor

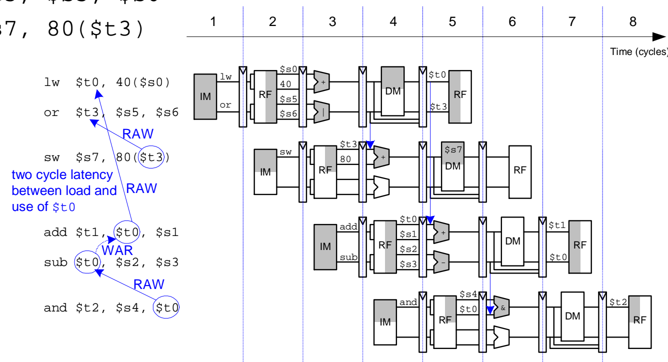
- **Instruction level parallelism:** the number of instruction that can be issued simultaneously (in practice average < 3)
- **Scoreboard:** table that keeps track of:
  - Instructions waiting to issue
  - Available functional units
  - Dependencies

Copyright © 2007 Elsevier

Out of Order Processor Example

```
lw   $t0, 40($s0)
add  $t1, $t0, $s1
sub  $t0, $s2, $s3
and  $t2, $s4, $t0
or   $t3, $s5, $s6
sw   $s7, 80($t3)
```

Ideal IPC: 2
Actual IPC: 6/4 = 1.5



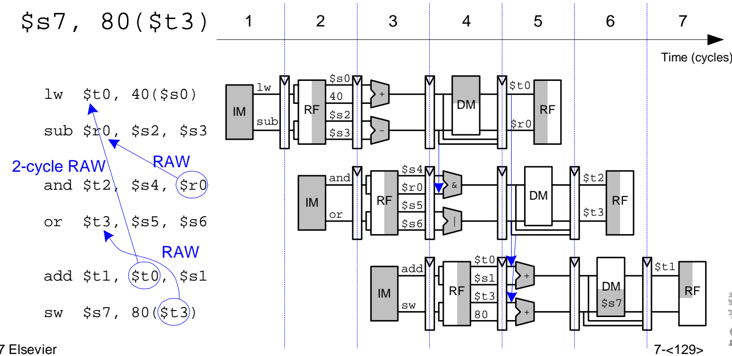
Copyright © 2007 Elsevier

Register Renaming

```
lw  $t0, 40($s0)
add $t1, $t0, $s1
sub $t0, $s2, $s3
and $t2, $s4, $t0
or  $t3, $s5, $s6
sw  $s7, 80($t3)
```

Ideal IPC: **2**

Actual IPC: **6/3 = 2**

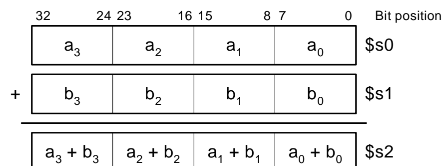


Copyright © 2007 Elsevier

SIMD

- Single Instruction Multiple Data (SIMD)
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called *packed arithmetic*)
- For example, add four 8-bit elements
- Must modify ALU to eliminate carries between 8-bit values

```
padd8 $s2, $s0, $s1
```



Copyright © 2007 Elsevier

Advanced Architecture Techniques

- Multithreading
  - Wordprocessor: thread for typing, spell checking, printing
- Multiprocessors
  - Multiple processors (cores) on a single chip

Copyright © 2007 Elsevier

Multithreading: First Some Definitions

- **Process:** program running on a computer
- Multiple processes can run at once: e.g., surfing Web, playing music, writing a paper
- **Thread:** part of a program
- Each process has multiple threads: e.g., a word processor may have threads for typing, spell checking, printing
- In a conventional processor:
  - One thread runs at once
  - When one thread stalls (for example, waiting for memory):
    - Architectural state of that thread is stored
    - Architectural state of waiting thread is loaded into processor and it runs
    - Called **context switching**
  - But appears to user like all threads running simultaneously

Copyright © 2007 Elsevier

Threads in Conventional Processor

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
  - Architectural state of that thread is stored
  - Architectural state of waiting thread is loaded into processor and it runs
  - Called **context switching**
- Appears to user like all threads running simultaneously

Copyright © 2007 Elsevier

Multithreading

- Multiple copies of architectural state
- Multiple threads **active** at once:
  - When one thread stalls, another runs immediately (no need to store or restore architectural state)
  - If one thread can't keep all execution units busy, another thread can use them
- Does not increase instruction-level parallelism (ILP) of single thread, but does increase throughput

Copyright © 2007 Elsevier

Multiprocessors

- Multiple processors (cores) with a method of communication between them
- Types of multiprocessing:
  - **Symmetric multiprocessing (SMT)**: multiple cores with a shared memory
  - **Asymmetric multiprocessing:** separate cores for different tasks (for example, DSP and CPU in cell phone)
  - **Clusters:** each core has its own memory system

Copyright © 2007 Elsevier

Other Resources

- Patterson & Hennessy's: *Computer Architecture: A Quantitative Approach*
- Conferences:
  - www.cs.wisc.edu/~arch/www/
  - ISCA (International Symposium on Computer Architecture)
  - HPCA (International Symposium on High Performance Computer Architecture)

Copyright © 2007 Elsevier