Chapter 7 :: Microarchitecture

Digital Design and Computer Architecture

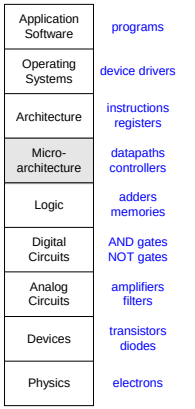
David Money Harris and Sarah L. Harris

Chapter 7 :: Topics

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

Introduction

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



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.

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

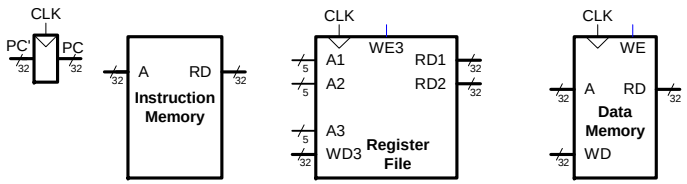
MIPS Processor

- 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`

Architectural State

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

MIPS State Elements

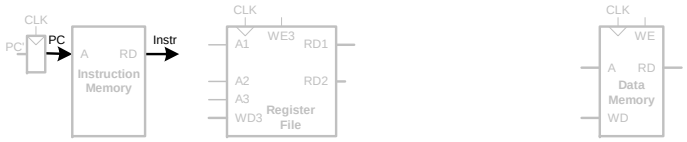


Single-Cycle MIPS Processor

- Datapath
- Control

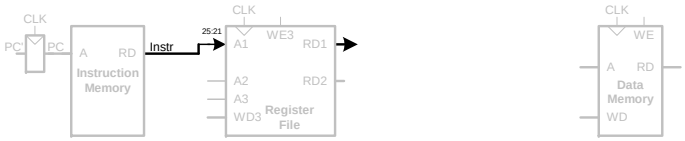
Single-Cycle Datapath: `lw` fetch

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



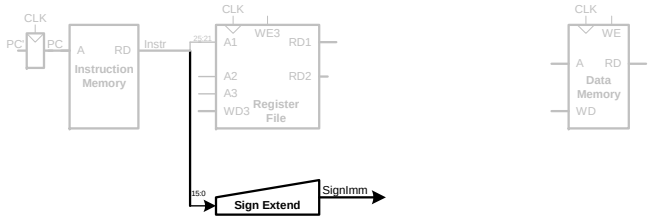
Single-Cycle Datapath: `lw` register read

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



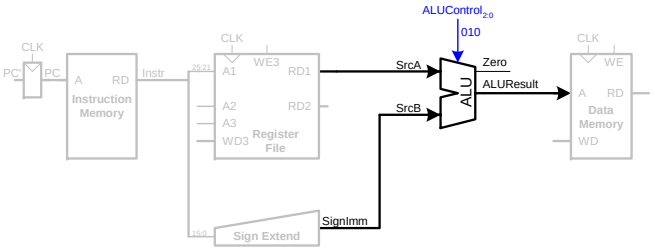
Single-Cycle Datapath: `lw` immediate

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



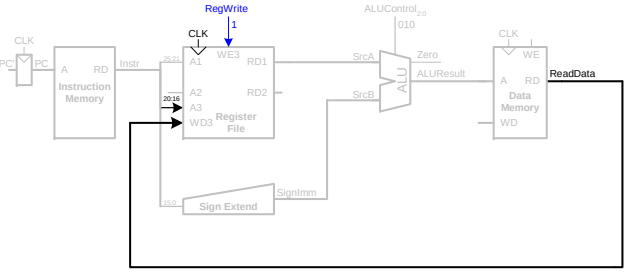
Single-Cycle Datapath: lw address

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



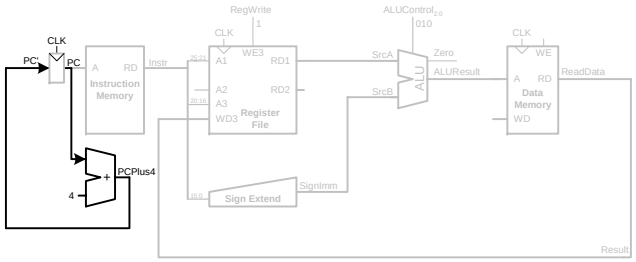
Single-Cycle Datapath: lw memory read

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



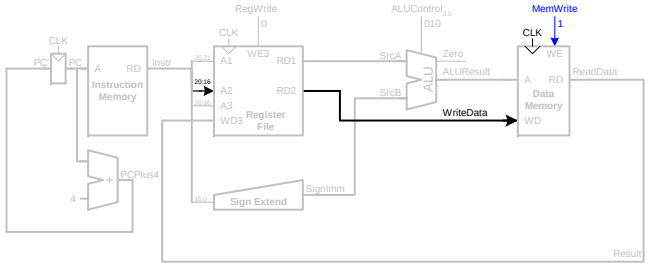
Single-Cycle Datapath: lw PC increment

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



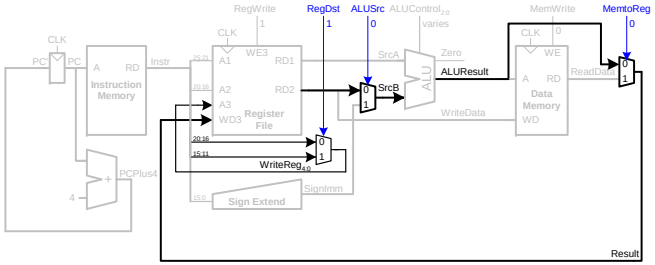
Single-Cycle Datapath: sw

- Consider `sw`
- Need to write data to memory



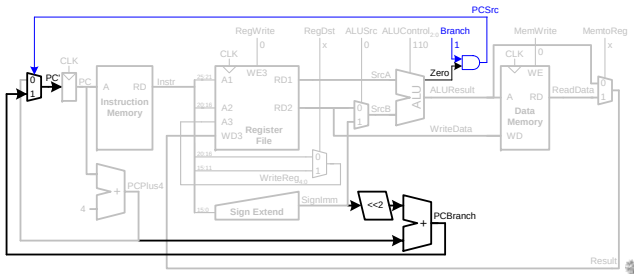
Single-Cycle Datapath: R-type instructions

- Consider R-type instructions: `add`, `sub`, `and`, `or`
- Write *ALUResult* to register file
- Writing to `rd` field of instruction (instead of `rt`)

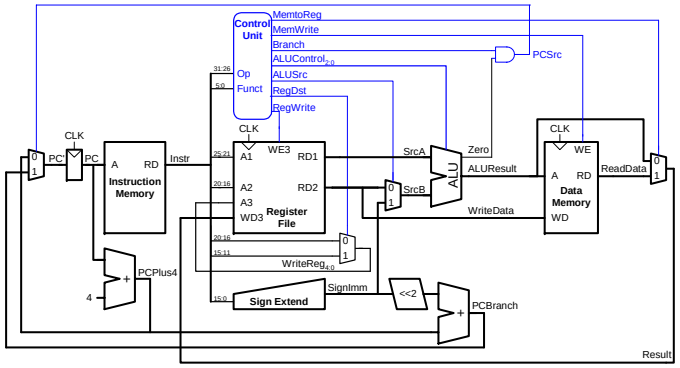


Single-Cycle Datapath: beq

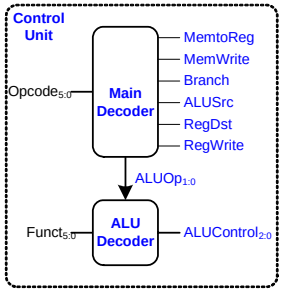
- Consider branch instruction: `beq`
- Determine whether register values are equal
- Calculate branch target address (BTA) from sign-extended immediate and PC+4



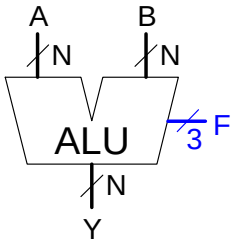
Complete Single-Cycle Processor



Control Unit

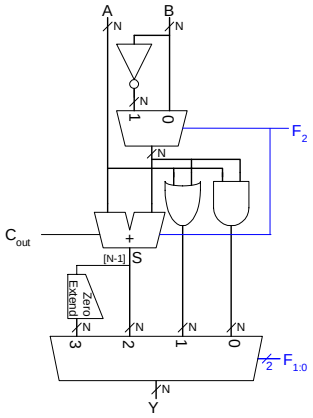


Review: ALU



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

Review: ALU



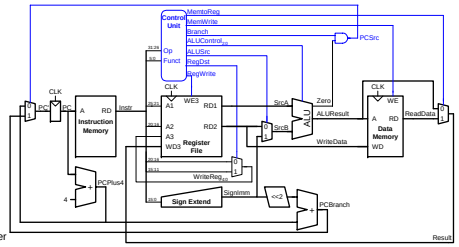
Control Unit: ALU Decoder

$ALUOp_{1:0}$	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)

Control Unit: Main Decoder

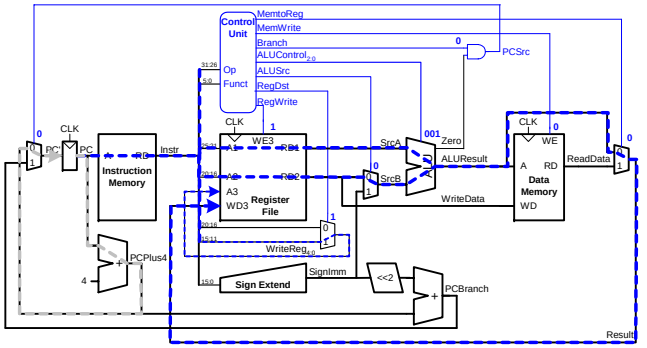
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000							
`lw`	100011							
`sw`	101011							
`beq`	000100							



Control Unit: Main Decoder

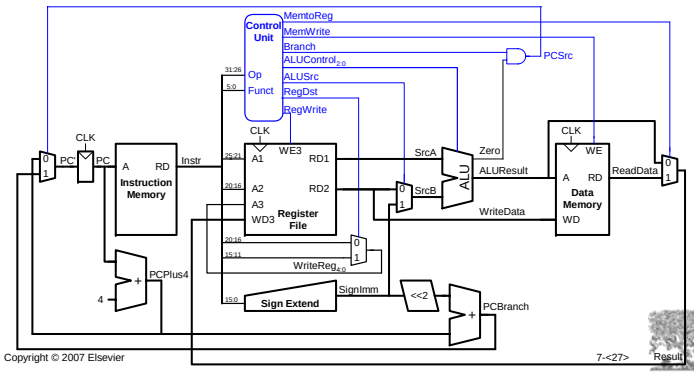
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
`lw`	100011	1	0	1	0	0	0	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

Single-Cycle Datapath Example: **or**



Extended Functionality: **addi**

- No change to datapath



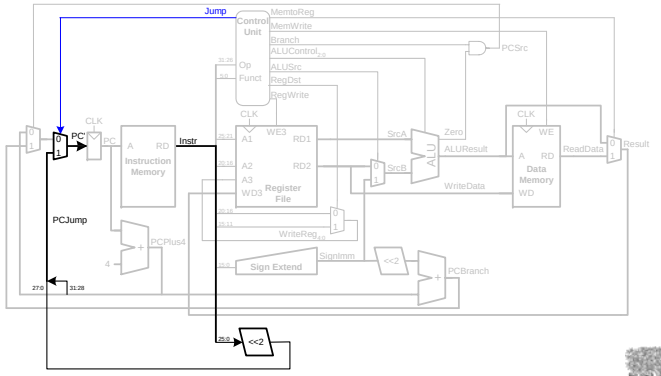
Control Unit: addi

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**							

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**

Extended Functionality: j



Control Unit: Main Decoder

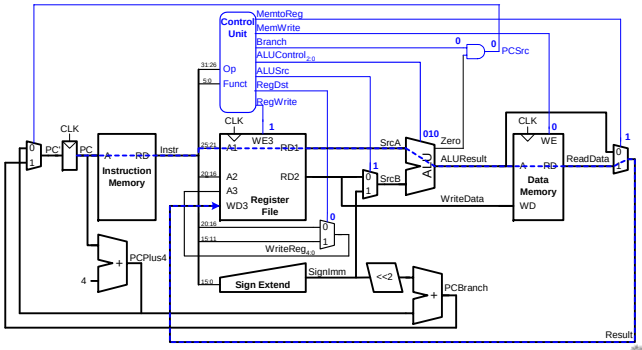
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	**000100**								

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**

Single-Cycle Performance

- Clock cycle time is limited by the critical path (`lw`)



Single-Cycle Performance

- Single-cycle critical path:
  $T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext}) + t_{mux} + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$
- In most implementations, limiting paths are:
  - memory, ALU, register file.
  - $T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + 2t_{mux} + t_{ALU} + t_{RFsetup}$

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

$T_c =$

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

$T_c = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + 2t_{mux} + t_{ALU} + t_{RFsetup}$
$= [30 + 2(250) + 150 + 2(25) + 200 + 20]$ ps
$= 950$ ps

Single-Cycle Performance Example

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

Execution Time =

Single-Cycle Performance Example

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

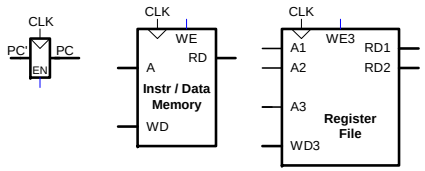
Execution Time = (# instructions)(cycles/instruction)(seconds/cycle)
= $(100 \times 10^9)(1)(950 \times 10^{-12}\ \text{s})$
= 95 seconds

Multicycle MIPS Processor

- The single-cycle microarchitecture performs all steps in one cycle
  - + simple
  - - cycle time limited by longest instruction (`lw`)
  - - two adders and two memories
- Multicycle microarchitecture uses a variable number of shorter steps (ALU or memory)
  - + higher clock speed
  - + simpler instructions run faster
  - + reuse expensive hardware on multiple cycles
  - - sequencing overhead paid many times
- Same design steps: datapath & control

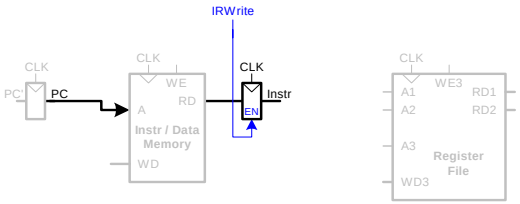
Multicycle State Elements

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

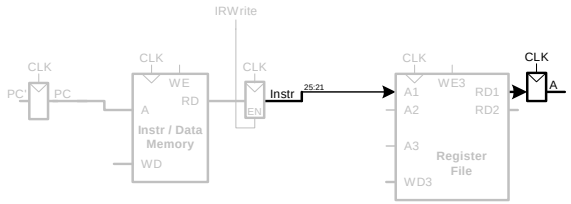


Multicycle Datapath: instruction fetch

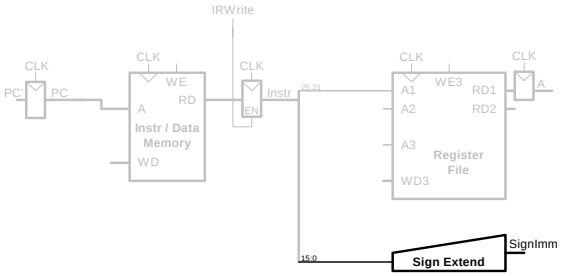
- First consider executing `lw`
- **STEP 1:** Fetch instruction



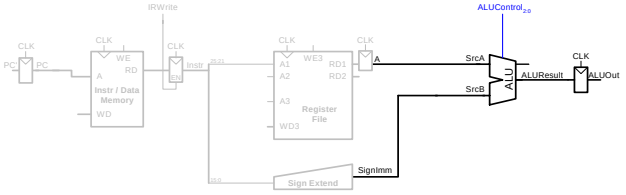
Multicycle Datapath: lw register read



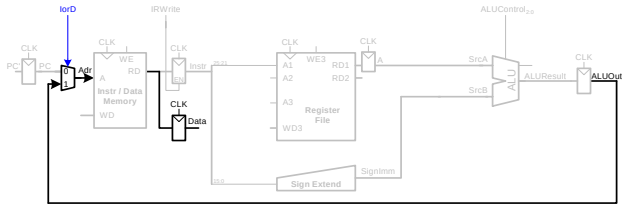
Multicycle Datapath: lw immediate



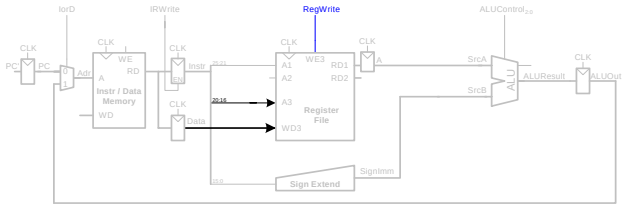
Multicycle Datapath: lw address



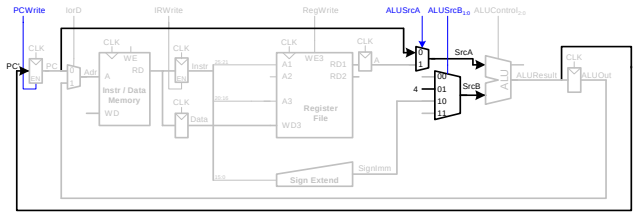
Multicycle Datapath: lw memory read



Multicycle Datapath: lw write register

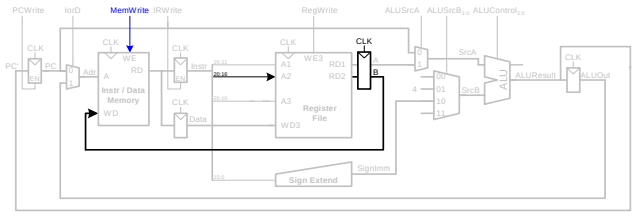


Multicycle Datapath: increment PC



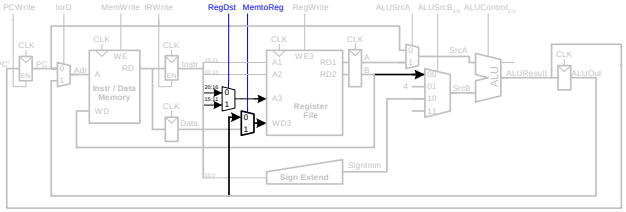
Multicycle Datapath: sw

- Consider `sw`
- Need to write data to memory



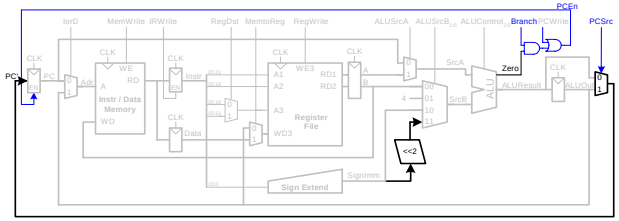
Multicycle Datapath: R-type Instructions

- Consider R-type instructions: `add`, `sub`, `and`, `or`
- Write *ALUResult* to register file
- Writing to `rd` field of instruction (instead of `rt`)

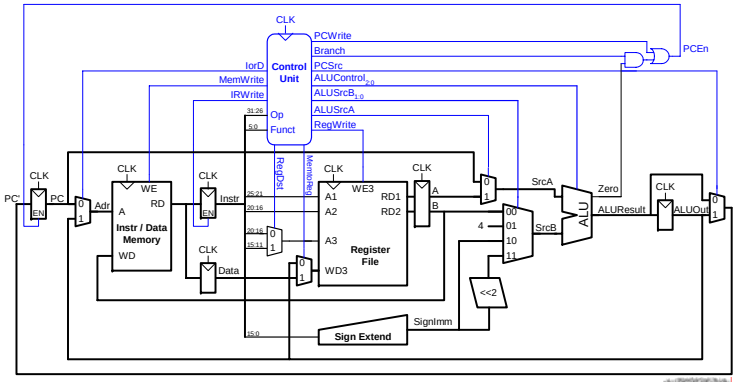


Multicycle Datapath: beq

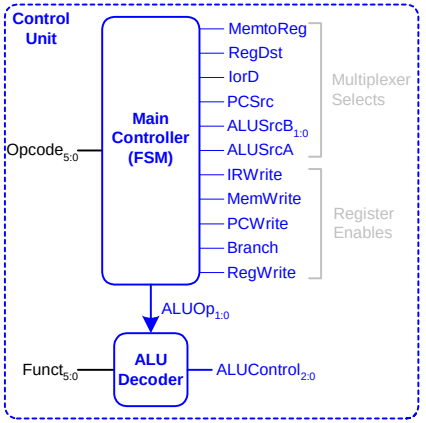
- Consider branch instruction: `beq`
- Determine whether register values are equal
- Calculate branch target address (BTA) from sign-extended immediate and PC+4



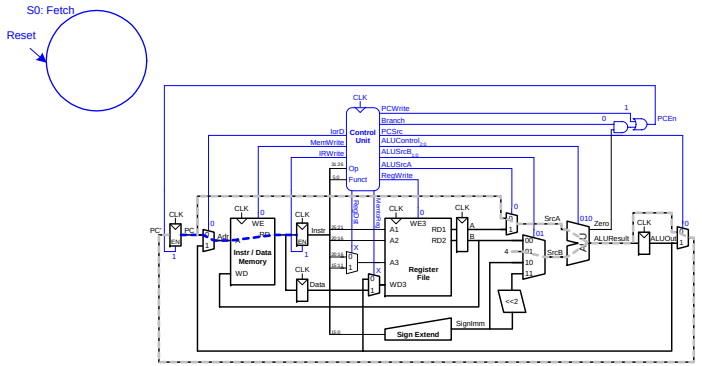
Complete Multicycle Processor



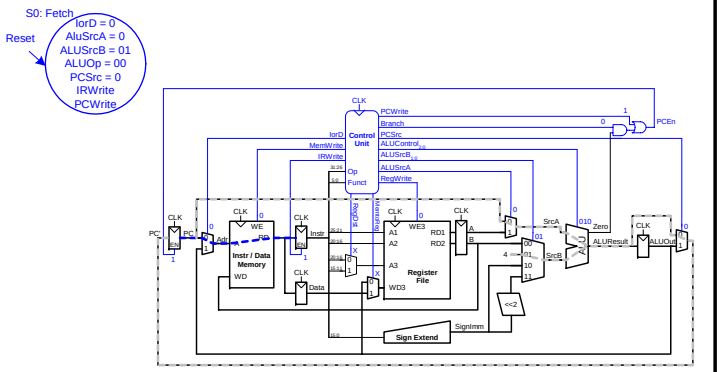
Control Unit



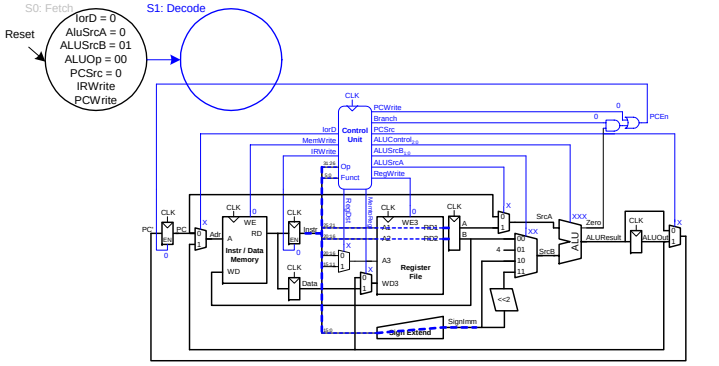
Main Controller FSM: Fetch



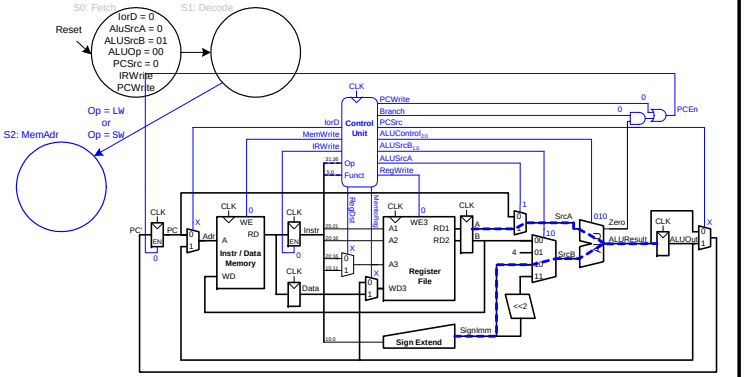
Main Controller FSM: Fetch



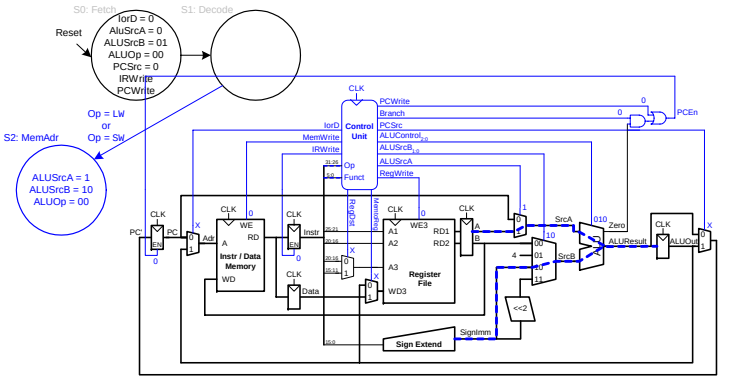
Main Controller FSM: Decode



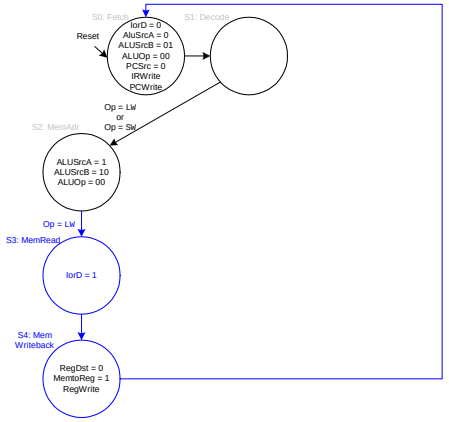
Main Controller FSM: Address Calculation



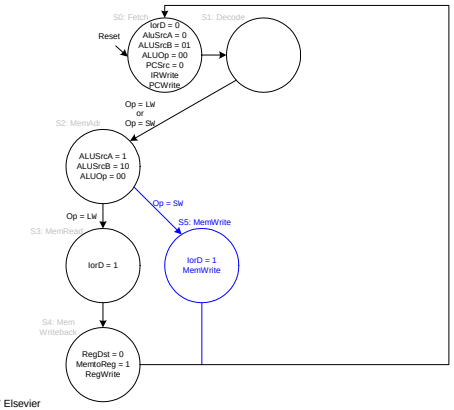
Main Controller FSM: Address Calculation



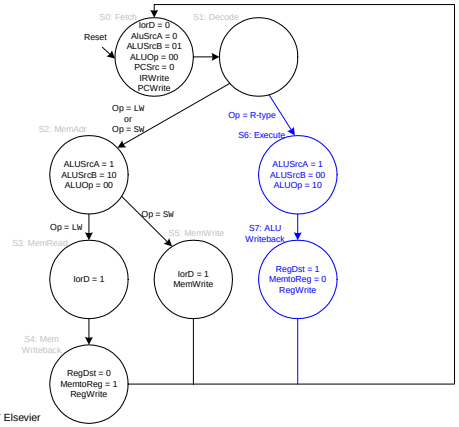
Main Controller FSM: lw



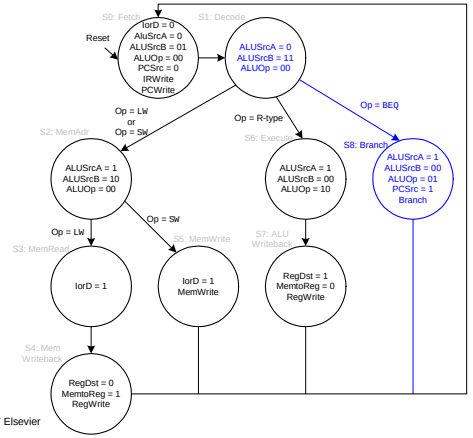
Main Controller FSM: sw



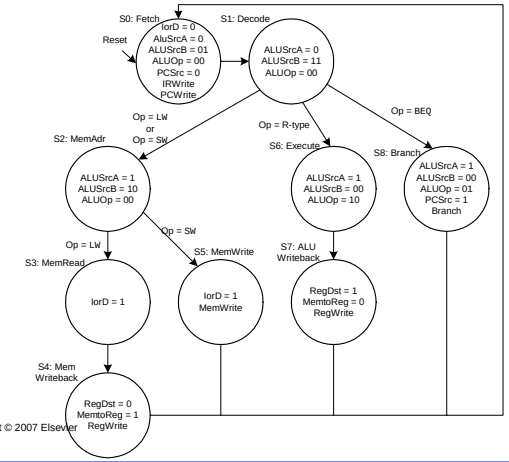
Main Controller FSM: R-Type



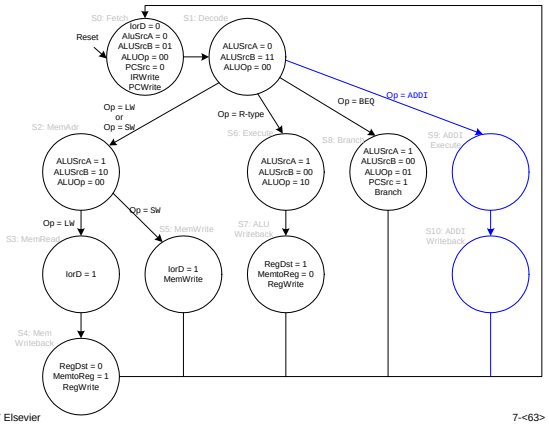
Main Controller FSM: beq



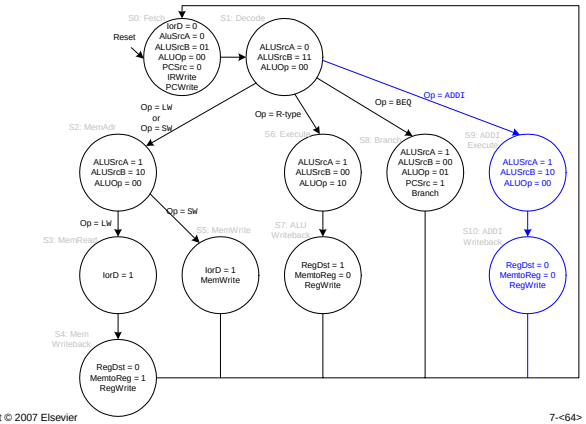
Complete Multicycle Controller FSM



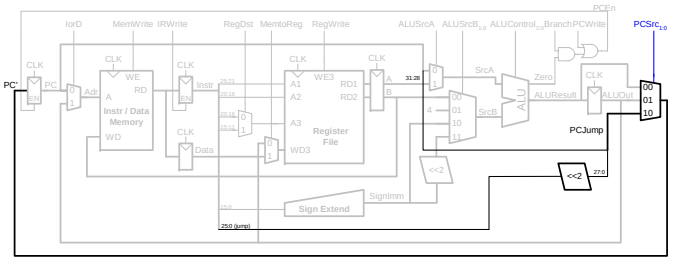
Main Controller FSM: addi



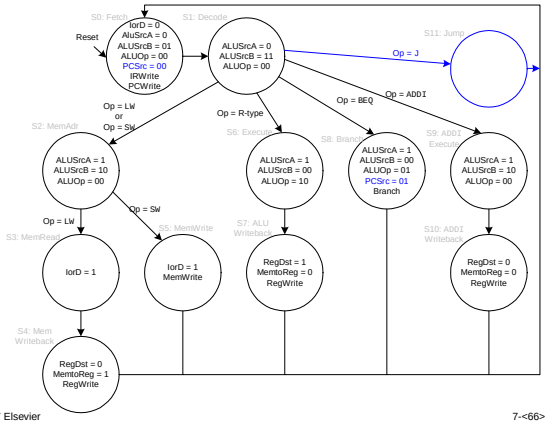
Main Controller FSM: addi



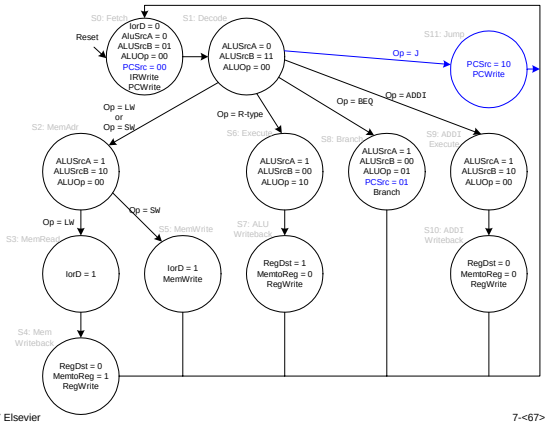
Extended Functionality: `j`



Control FSM: `j`



Control FSM: `j`



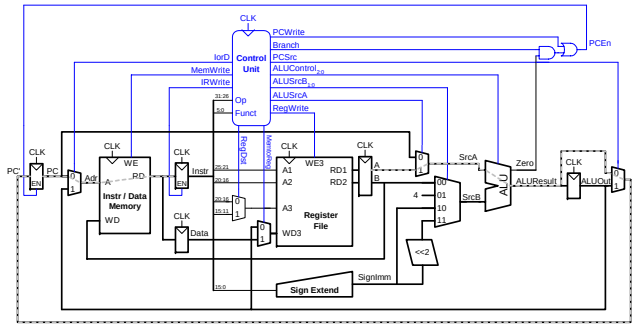
Multicycle Performance

- 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

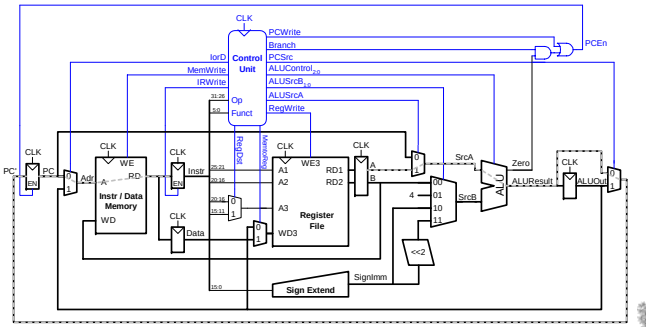
Multicycle Performance

- Multicycle critical path:
  T_c =



Multicycle Performance

- Multicycle critical path:
  $T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$



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_{RFread}	150
Register file setup	$t_{RFsetup}$	20

T_c =

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_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$T_c = t_{pcq\_PC} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$
$= t_{pcq\_PC} + t_{mux} + t_{mem} + t_{setup}$
$= [30 + 25 + 250 + 20]$ ps
$= 325$ ps

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 =

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 (95 seconds). Why?

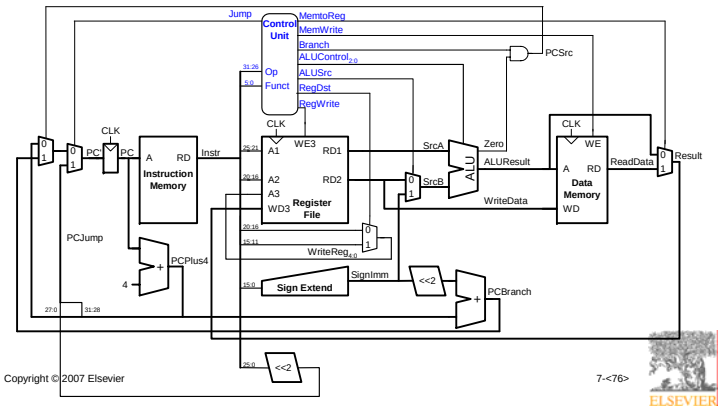
Multicycle Performance Example

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

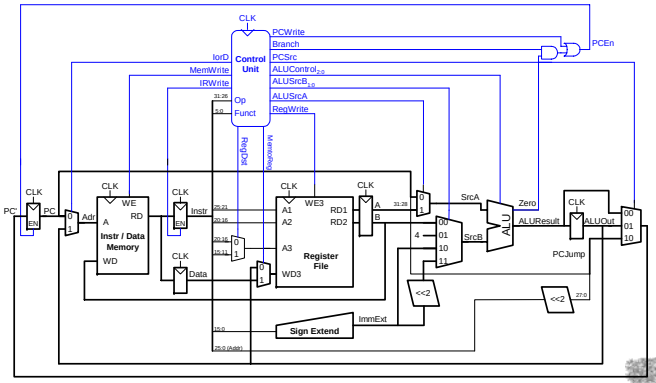
Execution Time = (# instructions) × CPI × T_c
= $(100 \times 10^9)(4.12)(325 \times 10^{-12})$
= 133.9 seconds

- This is slower than the single-cycle processor (95 seconds). Why?
  - Not all steps the same length
  - Sequencing overhead for each step ($t_{pcq} + t_{setup}$ = 50 ps)

Review: Single-Cycle MIPS Processor



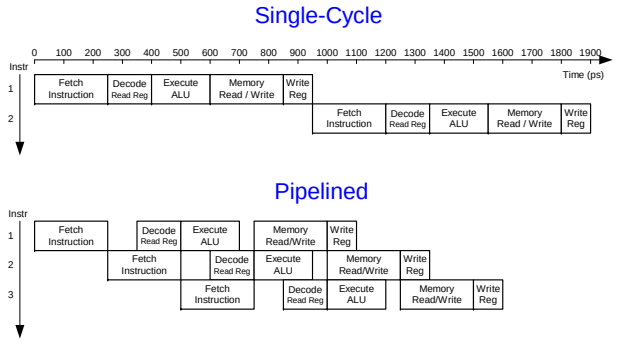
Review: Multicycle MIPS Processor



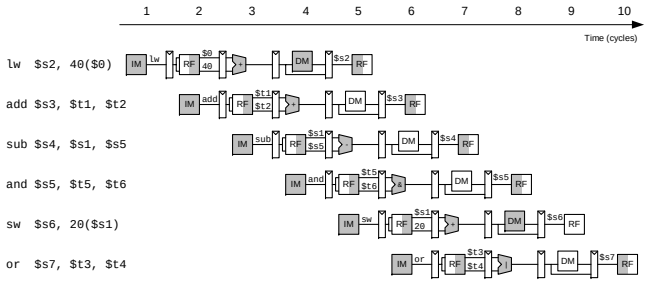
Pipelined MIPS Processor

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

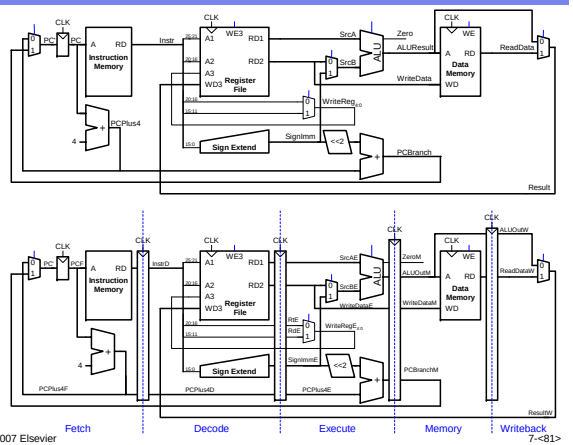
Single-Cycle vs. Pipelined Performance



Pipelining Abstraction

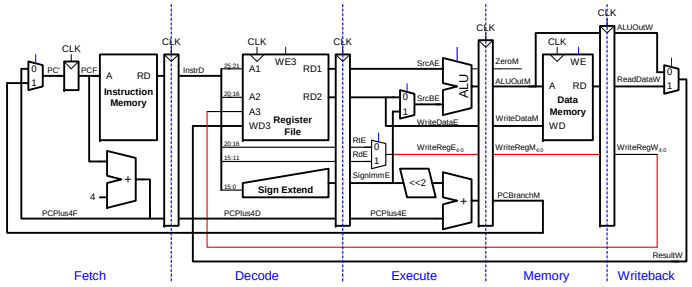


Single-Cycle and Pipelined Datapath

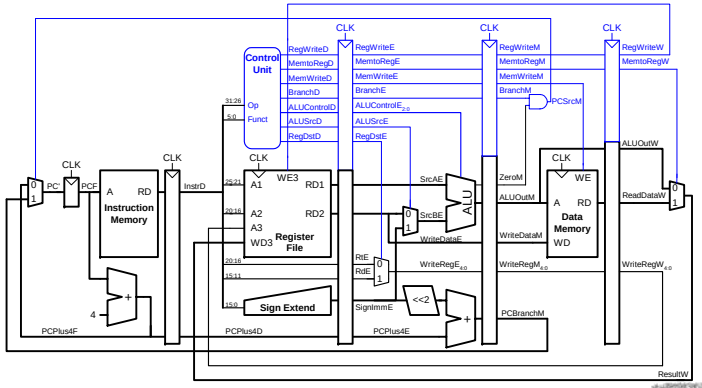


Corrected Pipelined Datapath

- WriteReg must arrive at the same time as Result



Pipelined Control



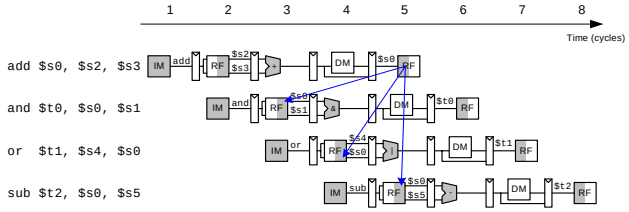
Same control unit as single-cycle processor

Control delayed to proper pipeline stage

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)

Data Hazard

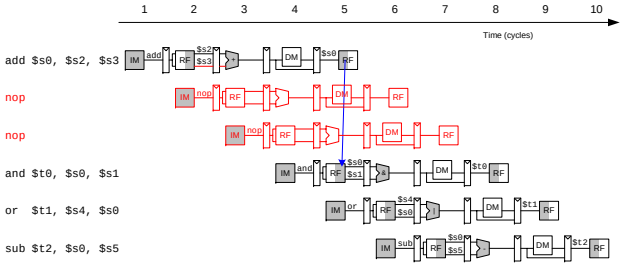


Handling Data Hazards

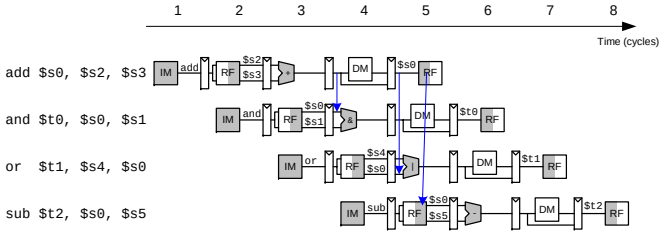
- Insert nops in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Compile-Time Hazard Elimination

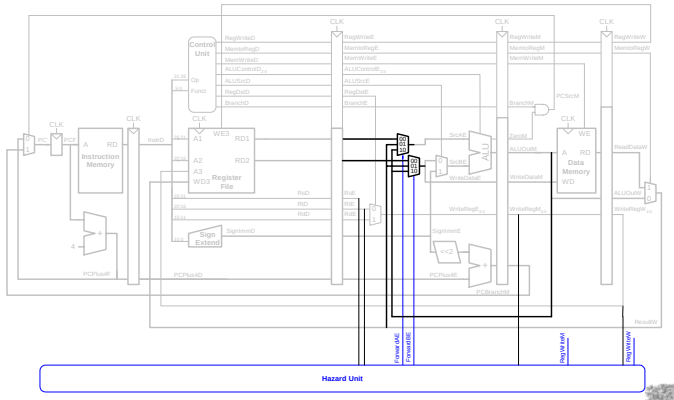
- Insert enough nops for result to be ready
- Or move independent useful instructions forward



Data Forwarding



Data Forwarding



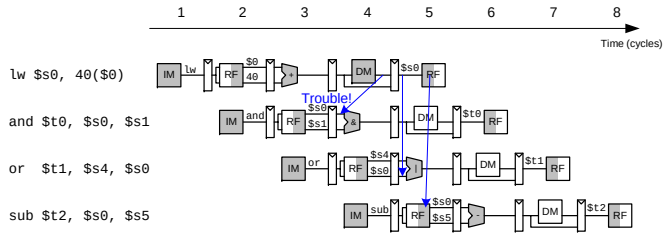
Data Forwarding

- Forward to Execute stage from either:
  - Memory stage or
  - Writeback stage
- Forwarding logic for *ForwardAE*:

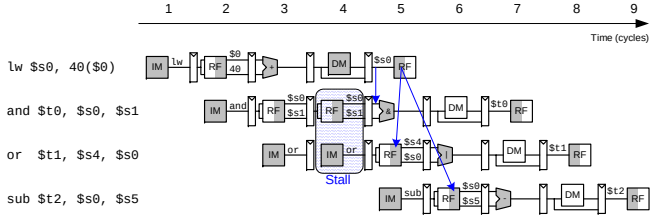
```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
then    ForwardAE = 10
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
then     ForwardAE = 01
else     ForwardAE = 00
```

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

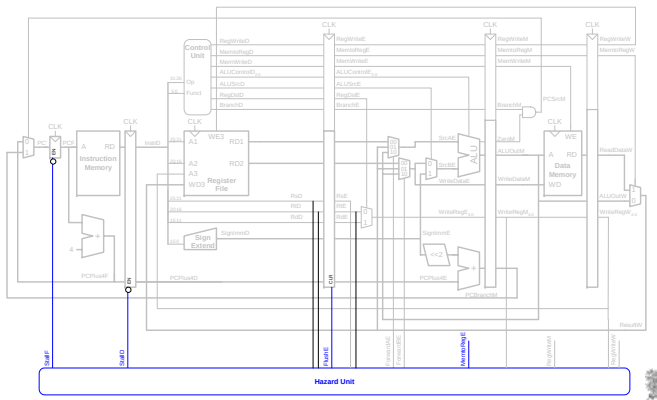
Stalling



Stalling



Stalling Hardware



Stalling Hardware

- Stalling logic:

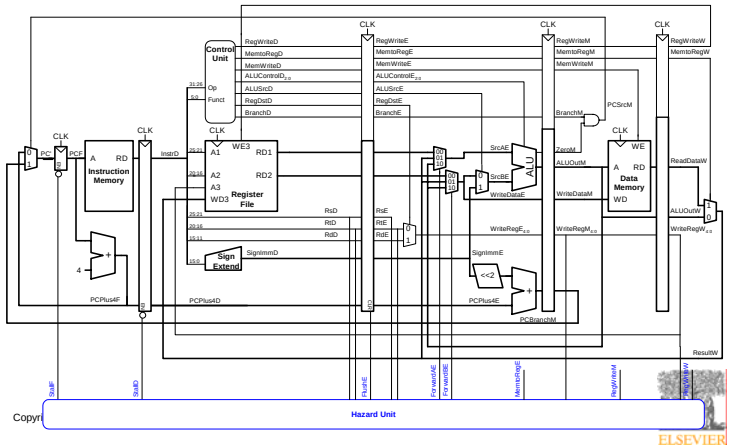
```
lwstall = ((rsD == rtE) OR (rtD == rtE)) AND MemtoRegE

StallF = StallD = FlushE = lwstall
```

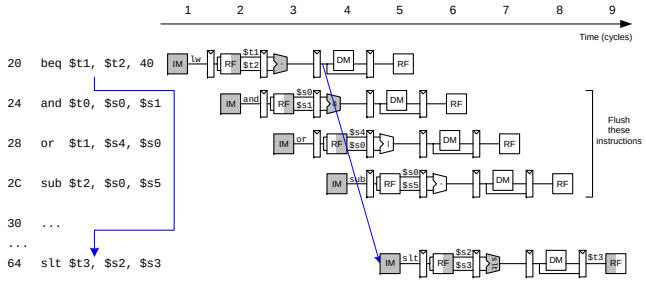

Control Hazards

- 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

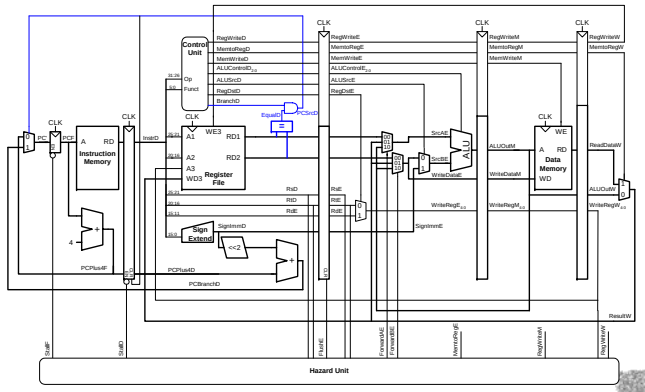
Control Hazards: Original Pipeline



Control Hazards

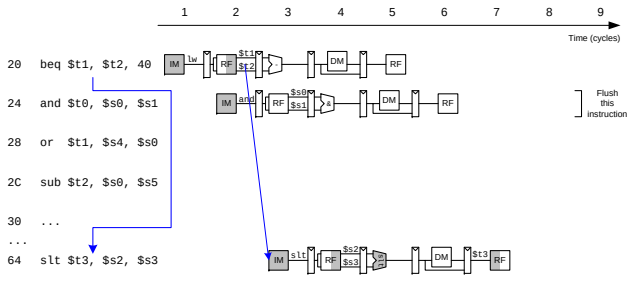


Control Hazards: Early Branch Resolution

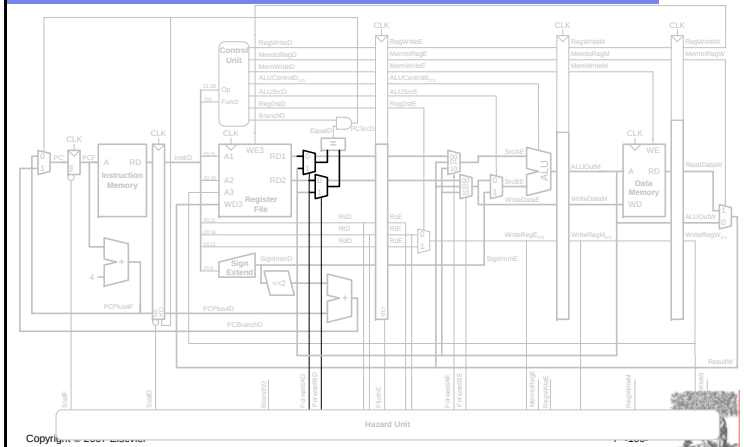


Introduced another data hazard in Decode stage

Control Hazards with Early Branch Resolution



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
```


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

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?**

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

Pipelined Performance

- Pipelined processor critical path:

T_c = max {

$t_{pcq} + t_{mem} + t_{setup}$

$2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$

$t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup}$

$t_{pcq} + t_{memwrite} + t_{setup}$

$2(t_{pcq} + t_{mux} + t_{RFwrite})$ }

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_{RFread}	150
Register file setup	$t_{RFsetup}$	20
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	$T_{memwrite}$	220
Register file write	$t_{RFwrite}$	100 ps

$T_c = 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$

$= 2[150 + 25 + 40 + 15 + 25 + 20]$ ps $= 550$ ps

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

$= (100 \times 10^9)(1.15)(550 \times 10^{-12})$

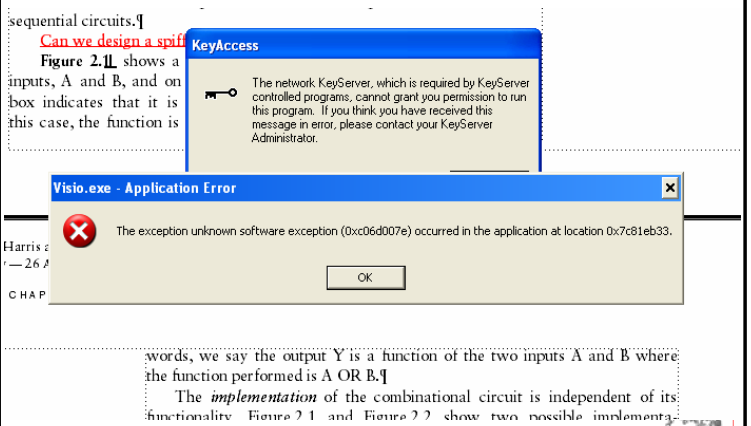
= 63 seconds

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

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)

Example Exception



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

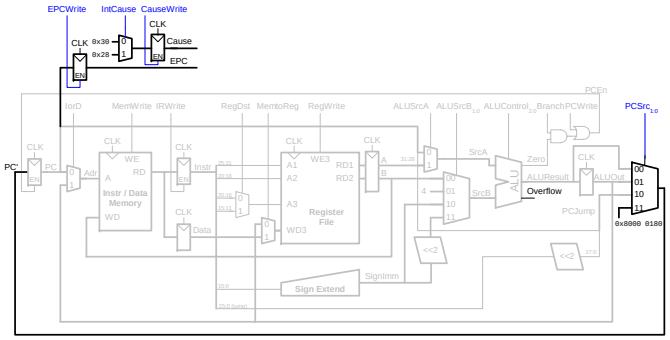
010000	00000	$t0 (8)	Cause (13)	00000000000
31:26	25:21	20:16	15:11	10:0

Exception Causes

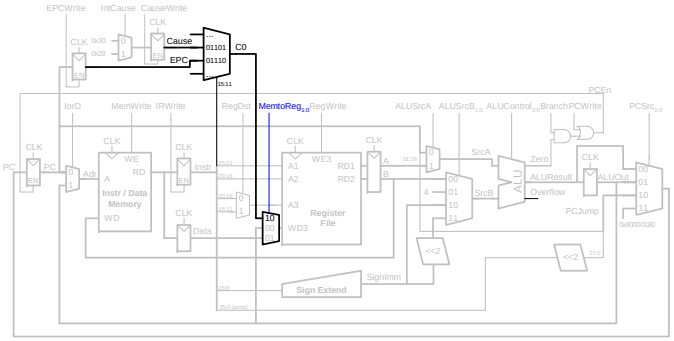
Exception	Cause
Hardware Interrupt	0x00000000
System Call	0x00000020
Breakpoint / Divide by 0	0x00000024
Undefined Instruction	0x00000028
Arithmetic Overflow	0x00000030

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

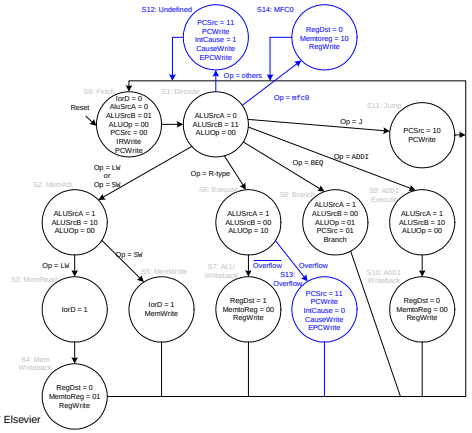
Exception Hardware: EPC & Cause



Exception Hardware: mfc0



Control FSM with Exceptions



Advanced MicroArchitecture

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

Deep Pipelining

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

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

Branch Prediction Example

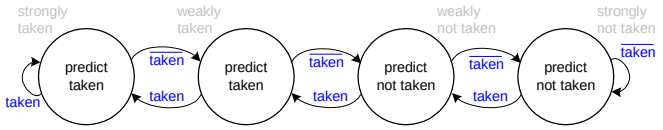
```
      add  $s1, $0, $0      # sum = 0
      add  $s0, $0, $0      # i   = 0
      addi $t0, $0, 10      # $t0 = 10
   for:
      beq  $t0, $t0, done   # if i == 10, branch
      add  $s1, $s1, $s0    # sum = sum + i
      addi $s0, $s0, 1      # increment i
      j    for
   done:
```


1-Bit Branch Predictor

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

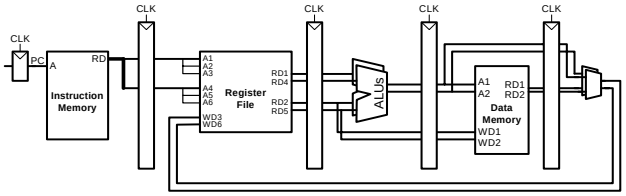
2-Bit Branch Predictor

- Only mispredicts last branch of loop



Superscalar

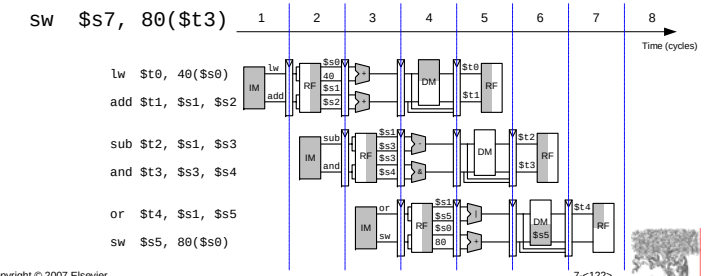
- Multiple copies of datapath execute multiple instructions at once
- Dependencies make it tricky to issue multiple instructions at once



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)
```

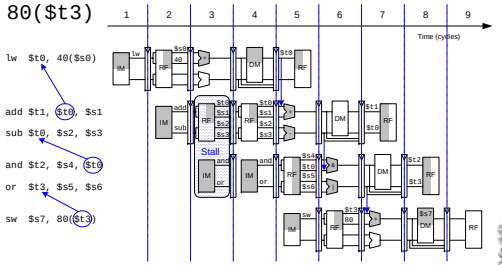
Ideal IPC: 2
Actual IPC: 2



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



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*)

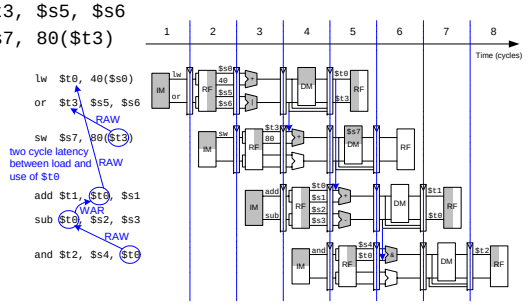
Out of Order Processor

- **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

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)
```

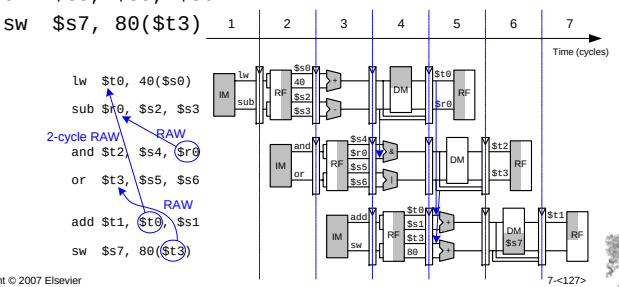
Ideal IPC: 2
Actual IPC: 6/4 = 1.5



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)
```

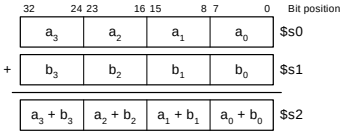
Ideal IPC: 2
Actual IPC: 6/3 = 2



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
```



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

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 ILP of single thread, but does increase throughput

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

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)