RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

ECE-332:437 DIGITAL SYSTEMS DESIGN (DSD)

Fall 2016 – Lecture 9
Micro Architecture (MIPS)

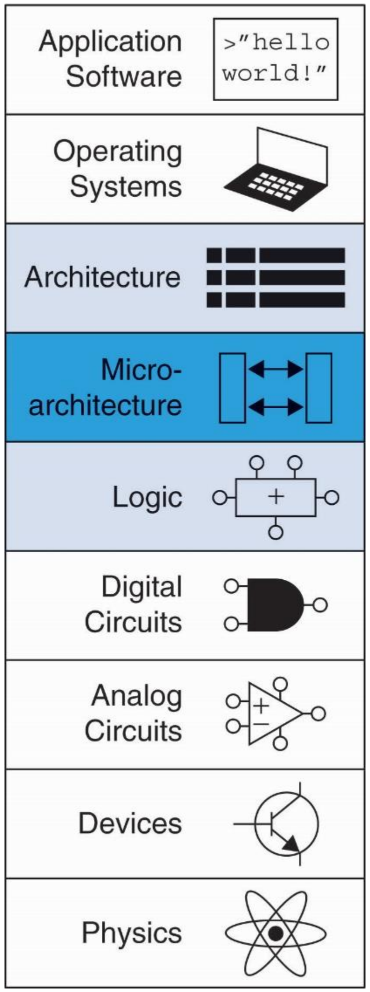
Nagi Naganathan
November 3, 2016

Topics to cover today – November 3, 2016

- Lecture 8 – Recap
- Lecture 9 – Chapter 7- Microarchitecture (MIPS)
- Slides from Text Book, Additional Books and adapted from miscellaneous material for educational purpose

Chapter 7 :: Topics

- **Introduction**
- **Single-Cycle Processor**
- **Multi-cycle Processor**
- **Pipelined Processor**
- **Exceptions**
- **Advanced Microarchitecture**

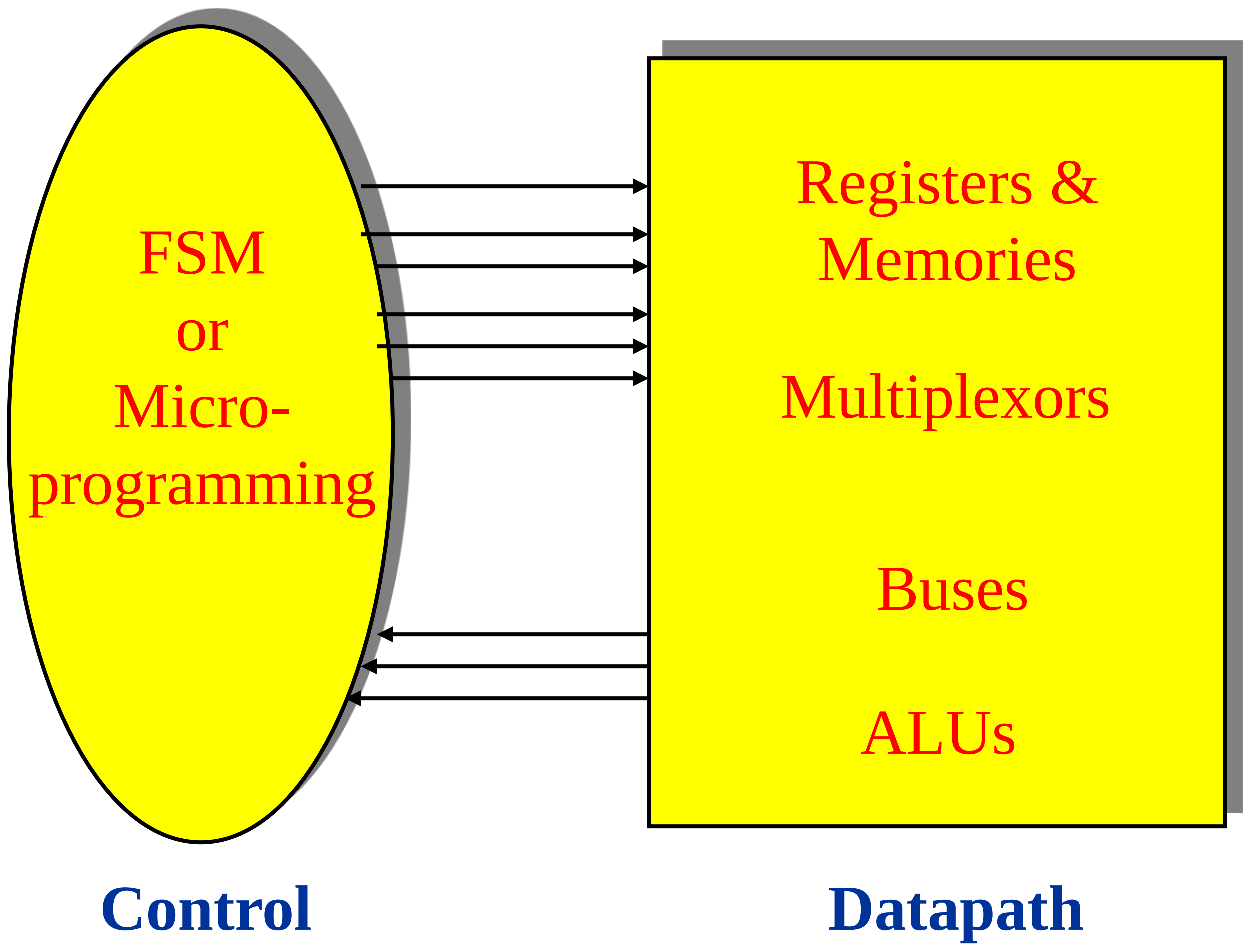


Introduction

- **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

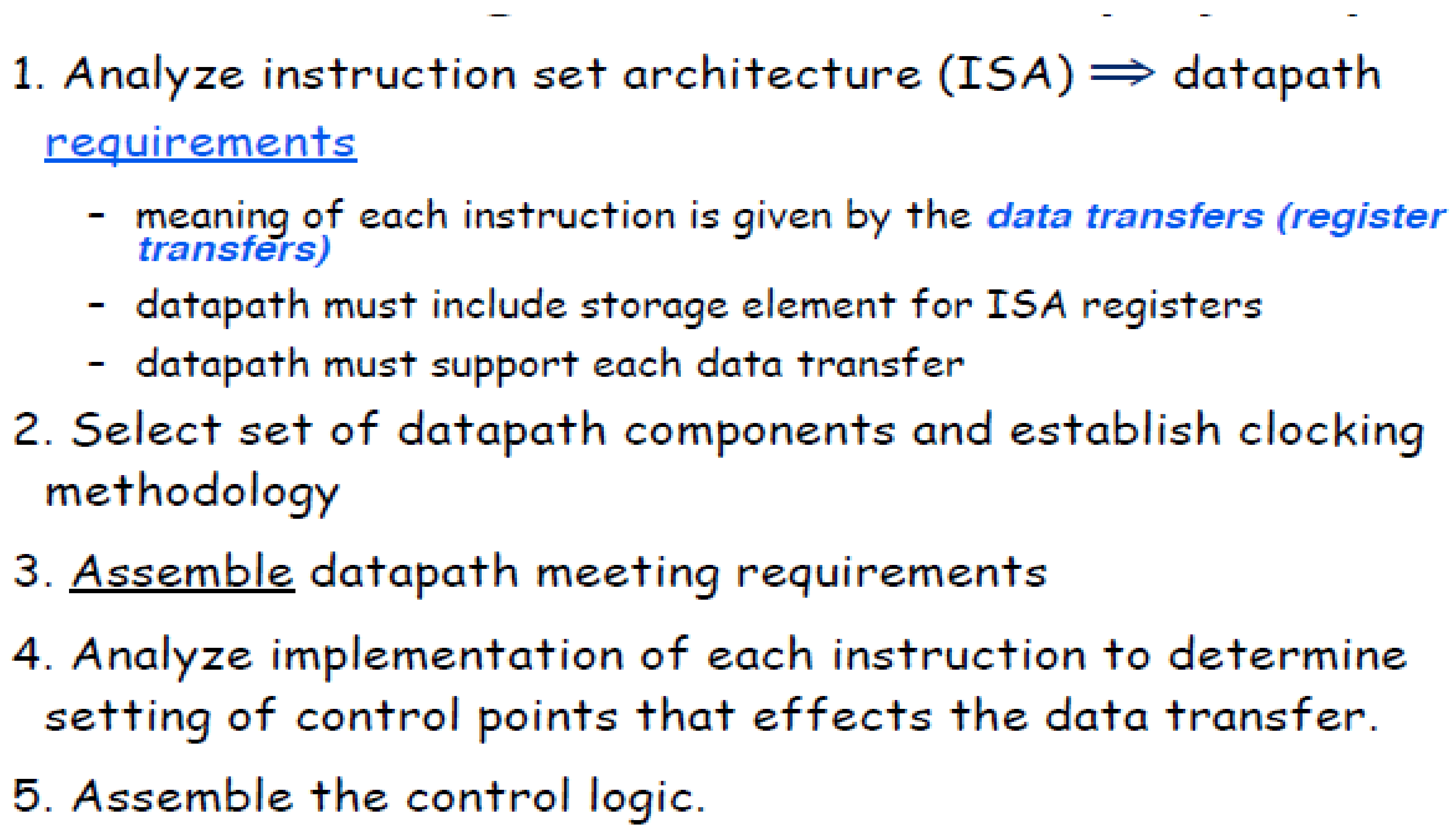
Datapath and Control



Microarchitecture

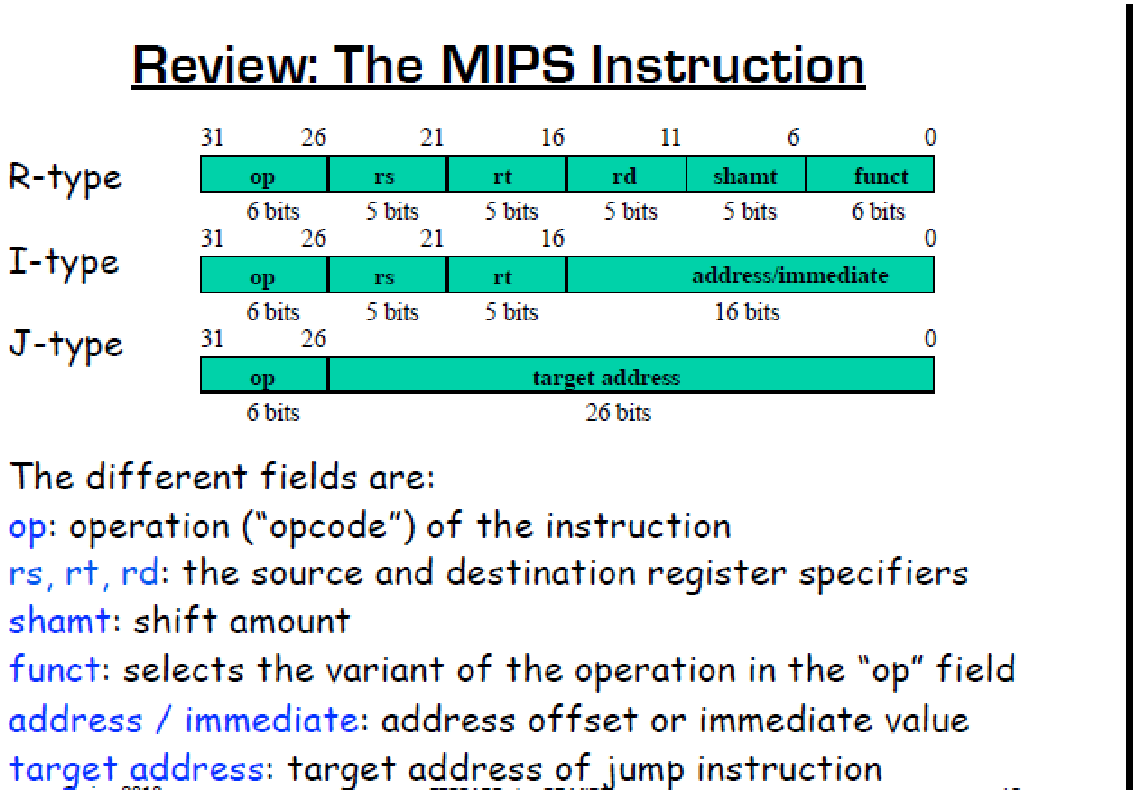
- Multiple implementations for a single architecture:
  - **Single-cycle:** Each instruction executes in a single cycle
  - **Multi-cycle:** Each instruction is broken into series of shorter steps
  - **Pipelined:** Each instruction broken up into series of steps & multiple instructions execute at once

Microarchitecture



1. Analyze instruction set architecture (ISA) $\Rightarrow$ datapath requirements
   - meaning of each instruction is given by the ***data transfers (register transfers)***
   - datapath must include storage element for ISA registers
   - datapath must support each data transfer
2. Select set of datapath components and establish clocking methodology
3. Assemble datapath meeting requirements
4. Analyze implementation of each instruction to determine setting of control points that effects the data transfer.
5. Assemble the control logic.

Microarchitecture



Review: The MIPS Instruction

	31 26	21	16	11	6	0
R-type	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
	31 26	21	16			0
I-type	op	rs	rt	address/immediate		
	6 bits	5 bits	5 bits	16 bits		
	31 26					0
J-type	op	target address				
	6 bits	26 bits				

The different fields are:

op: operation ("opcode") of the instruction

rs, rt, rd: the source and destination register specifiers

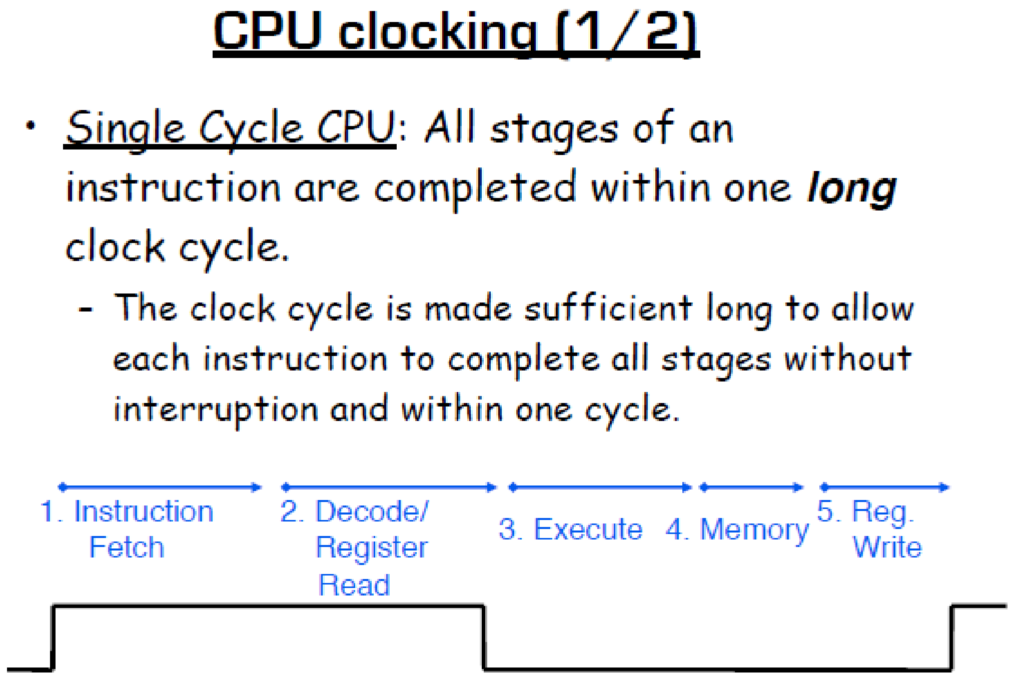
shamt: shift amount

funct: selects the variant of the operation in the "op" field

address / immediate: address offset or immediate value

target address: target address of jump instruction

Microarchitecture



CPU clocking (1/2)

- Single Cycle CPU: All stages of an instruction are completed within one ***long*** clock cycle.
  - The clock cycle is made sufficient long to allow each instruction to complete all stages without interruption and within one cycle.

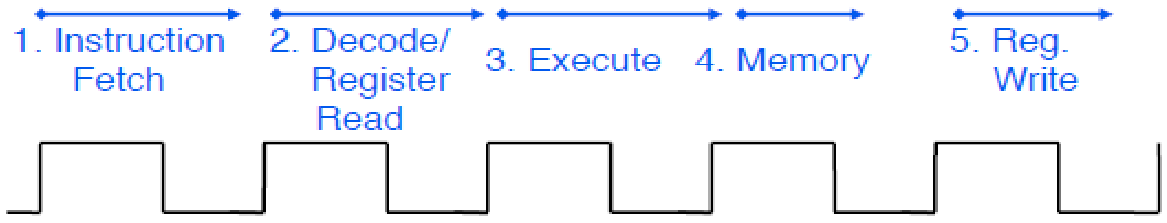
1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Reg. Write

RUTGERS

Microarchitecture

<u>CPU clocking (2/2)</u>

- <u>Multiple-cycle CPU</u>: Only one stage of instruction per clock cycle.
  - The clock is made as long as the slowest stage.



Several significant advantages over single cycle execution: Unused stages in a particular instruction can be skipped OR instructions can be pipelined (overlapped).

Processor Performance

- Program execution time

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

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

The Processor: Datapath & Control

- Simplified MIPS implementation to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do

- All instructions use the ALU after reading the registers

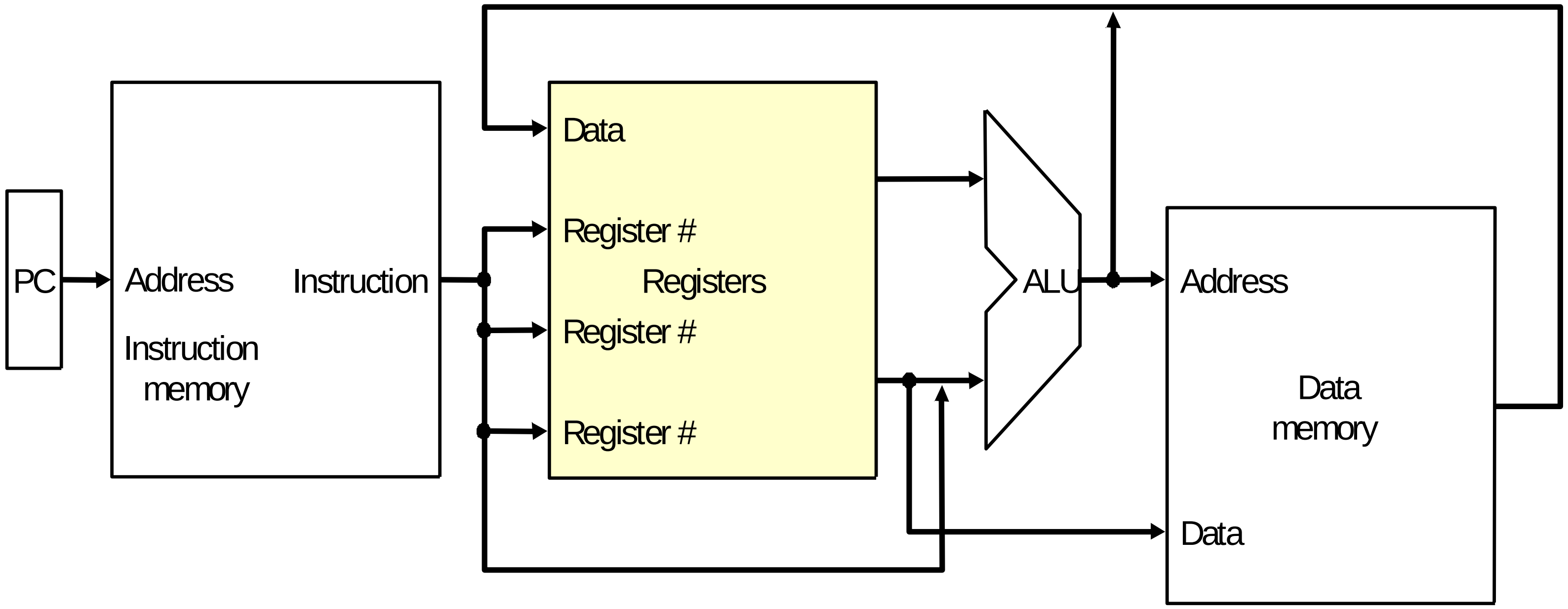
  Why?
  - memory-reference?
  - arithmetic?
  - control flow?

Architectural State

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

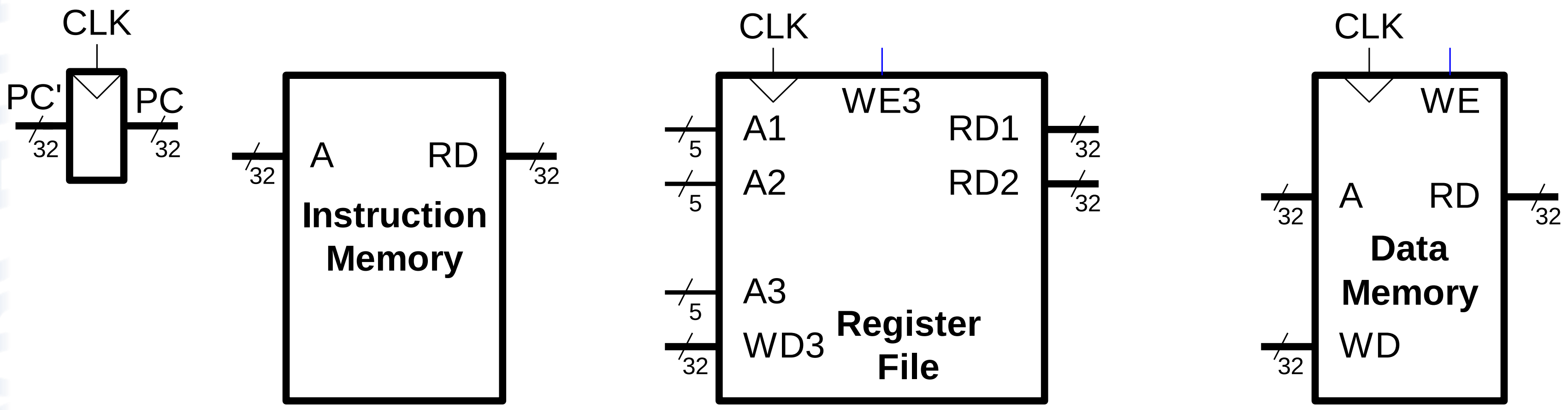
More Implementation Details

- Abstract / Simplified View:



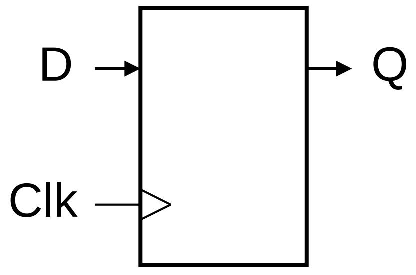
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

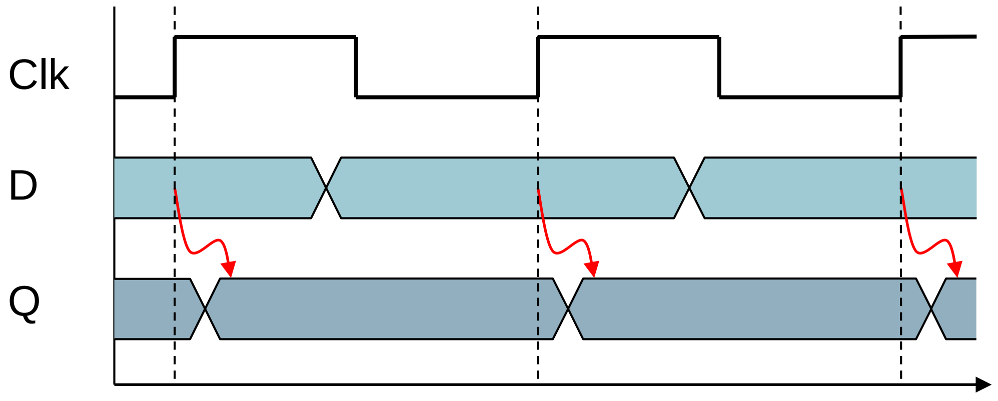
MIPS State Elements



Sequential Elements

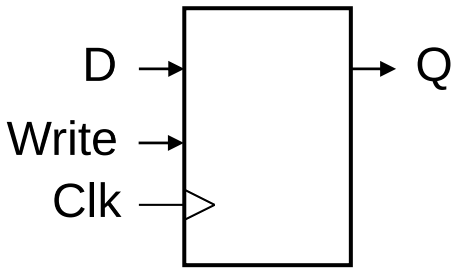
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

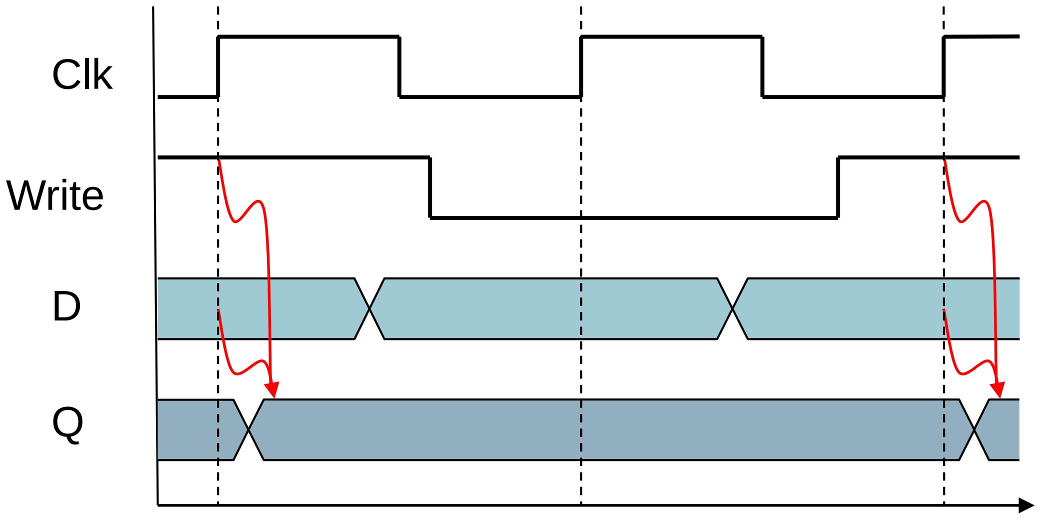




Sequential Elements

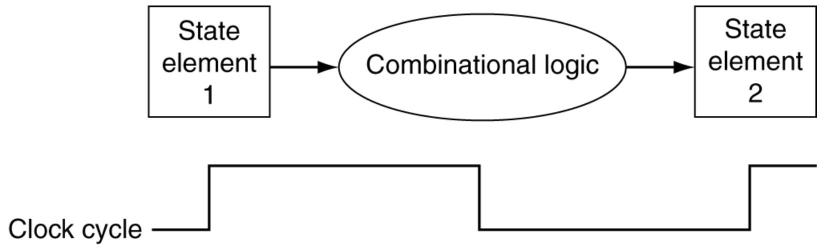
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

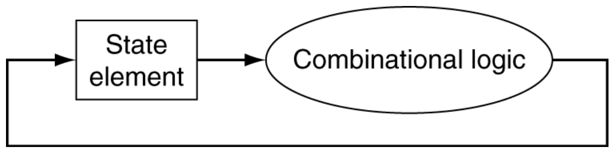




Clocking Methodology

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

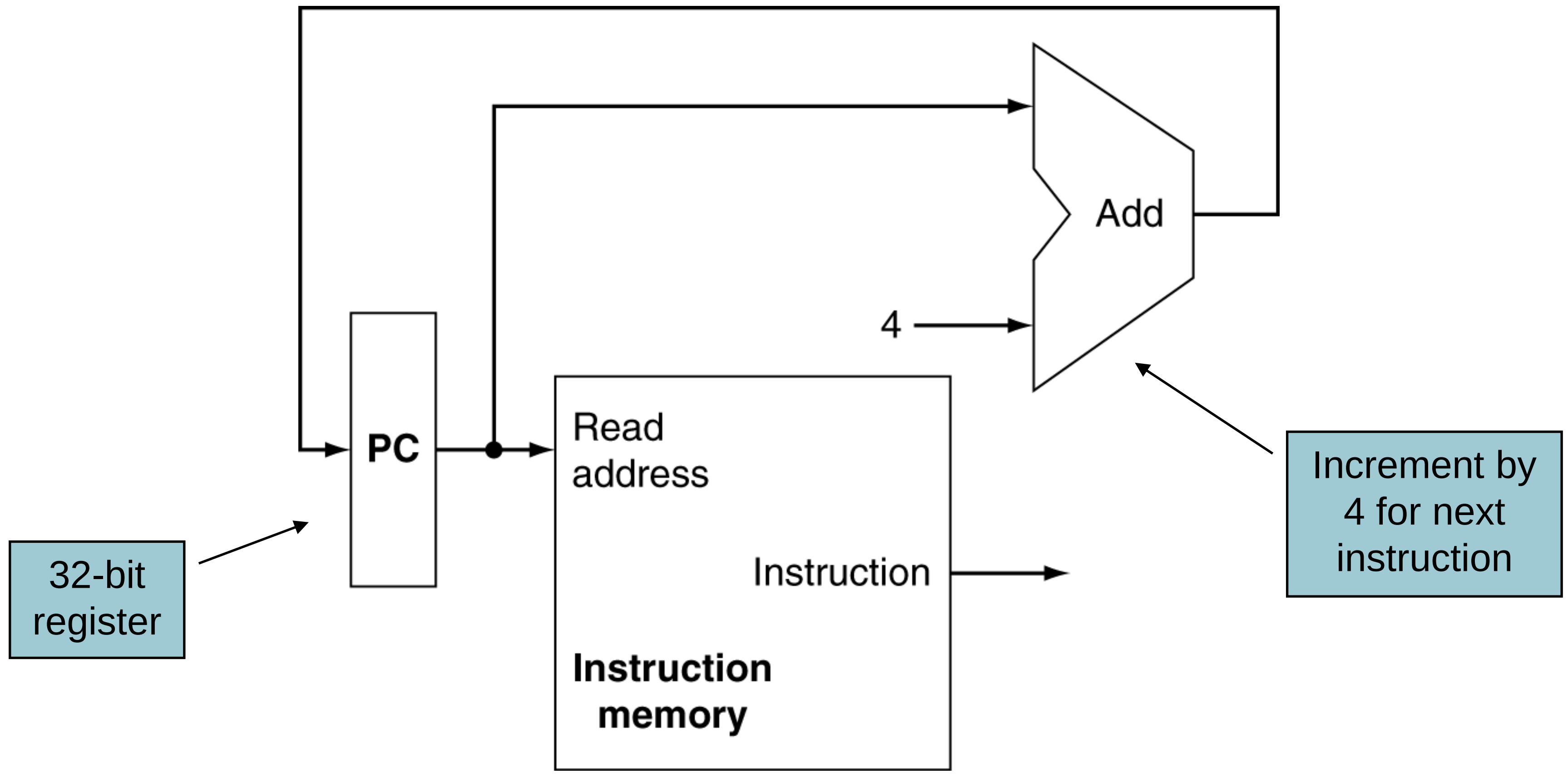




Building a Datapath

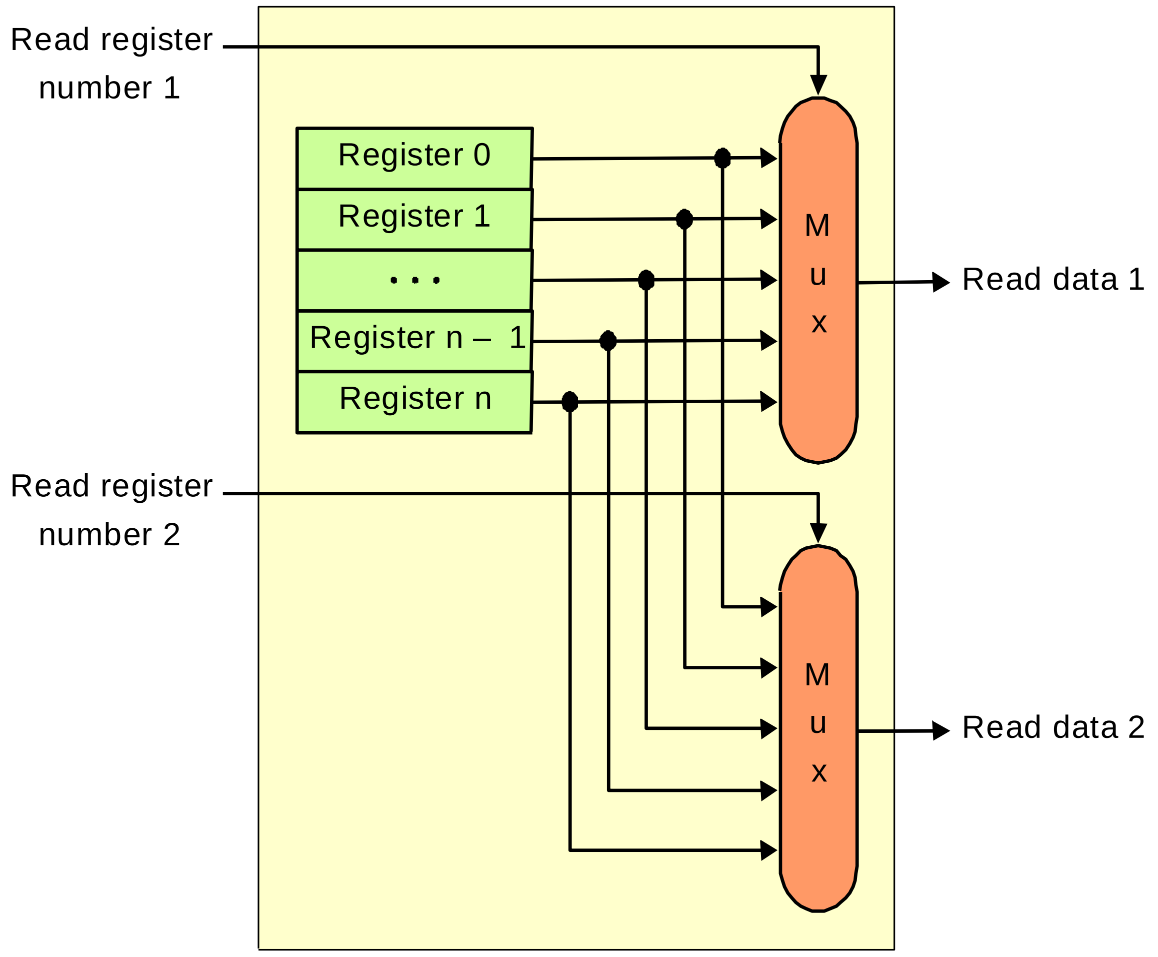
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux’s, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

Instruction Fetch



Register file: **read ports**

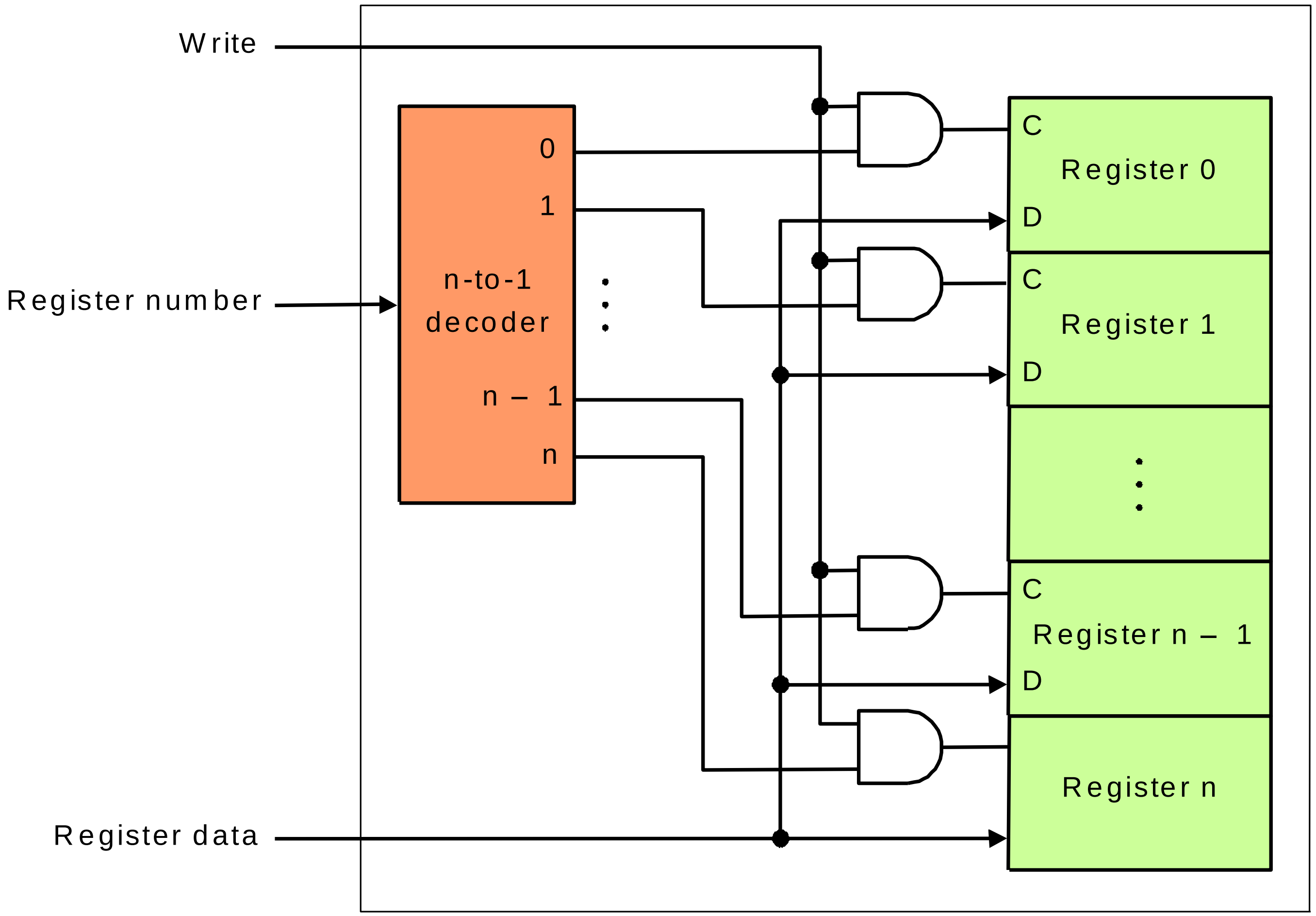
- Register file built using D flip-flops



Implementation of the read ports

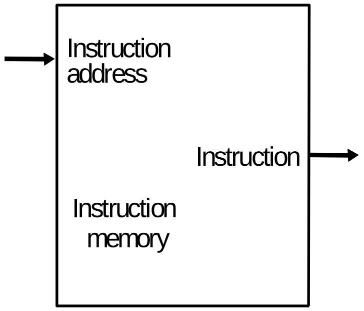
Register file: **write port**

- Note: we still use the real clock to determine when to write

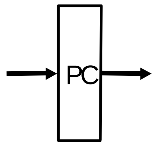


Simple Implementation

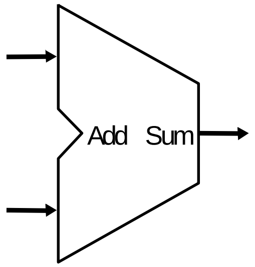
- Include the functional units we need for each instruction



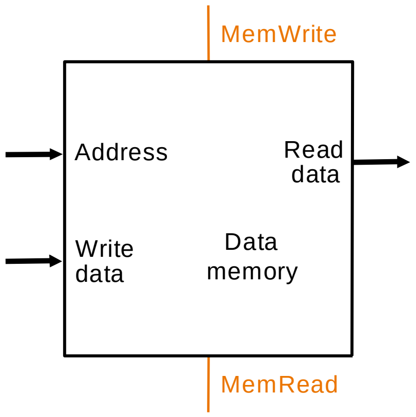
a. Instruction memory



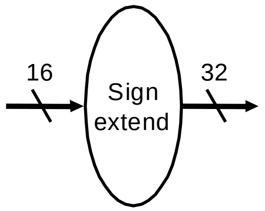
b. Program counter



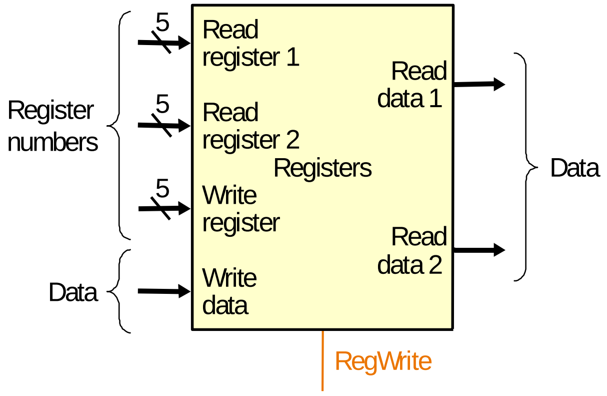
c. Adder



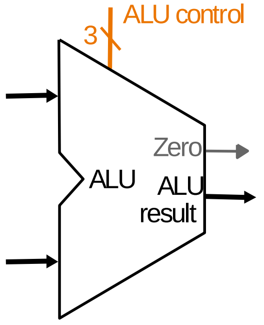
a. Data memory unit



b. Sign-extension unit



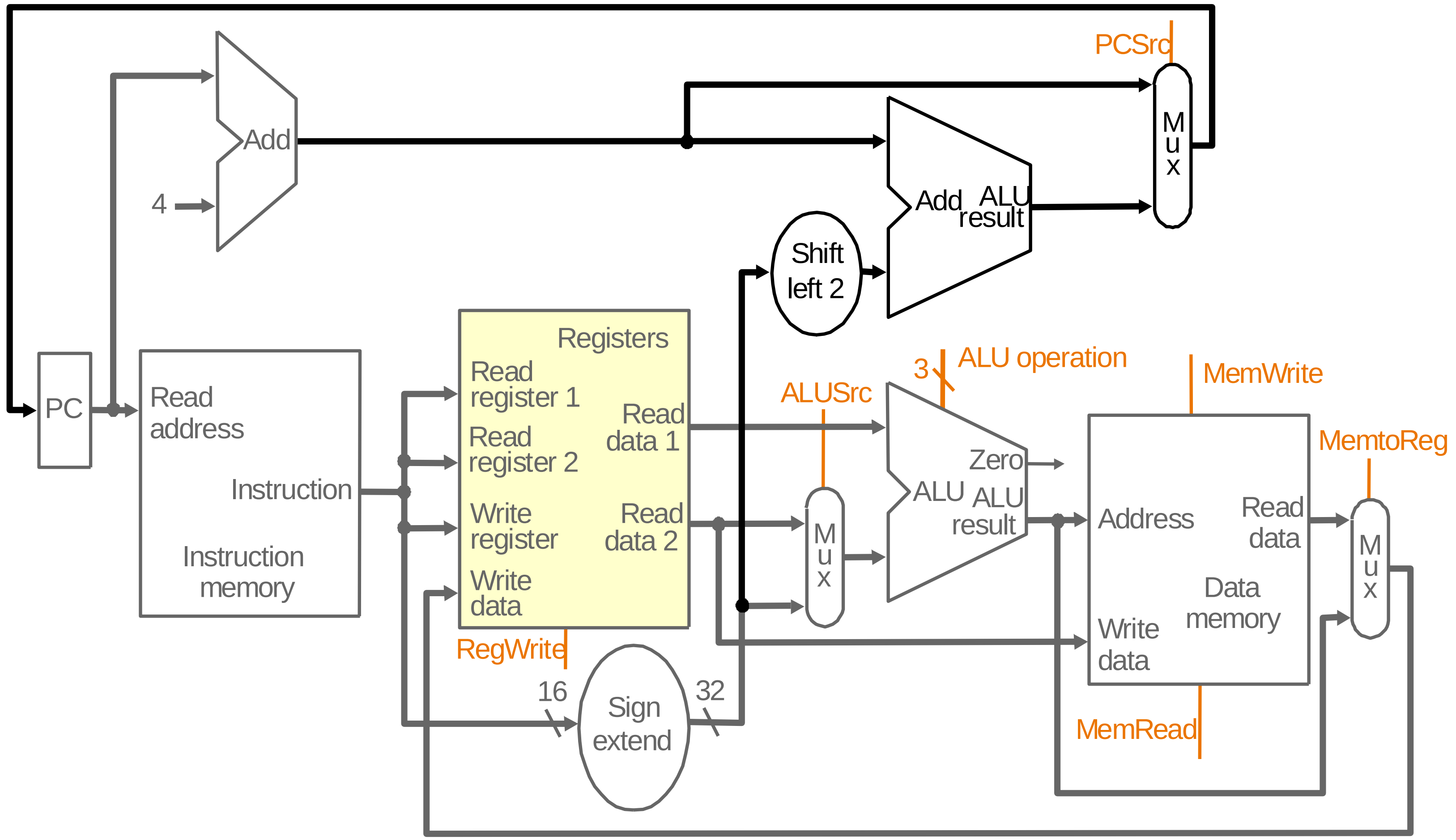
a. Registers



b. ALU

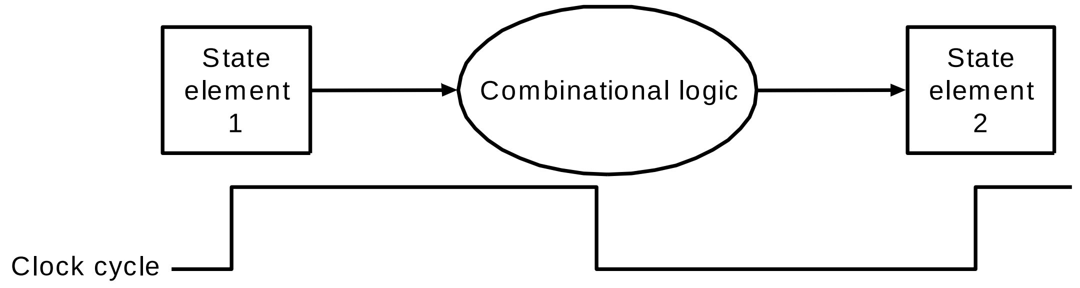
Building the Datapath

- Use multiplexors to stitch them together



Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce "right answer" right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



Control

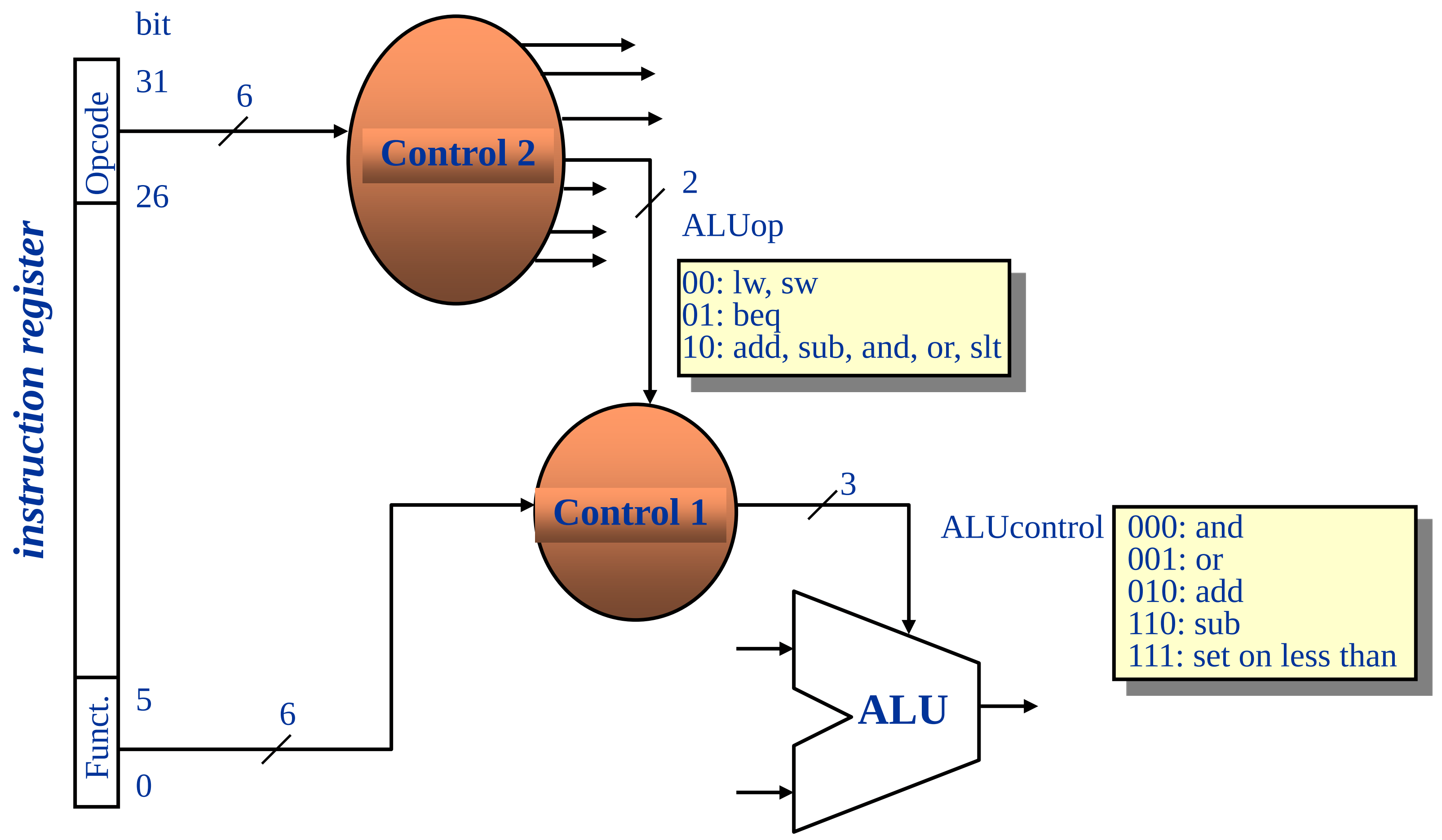
- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:

  add $8, $17, $18   Instruction Format:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- ALU's operation based on instruction type and function code

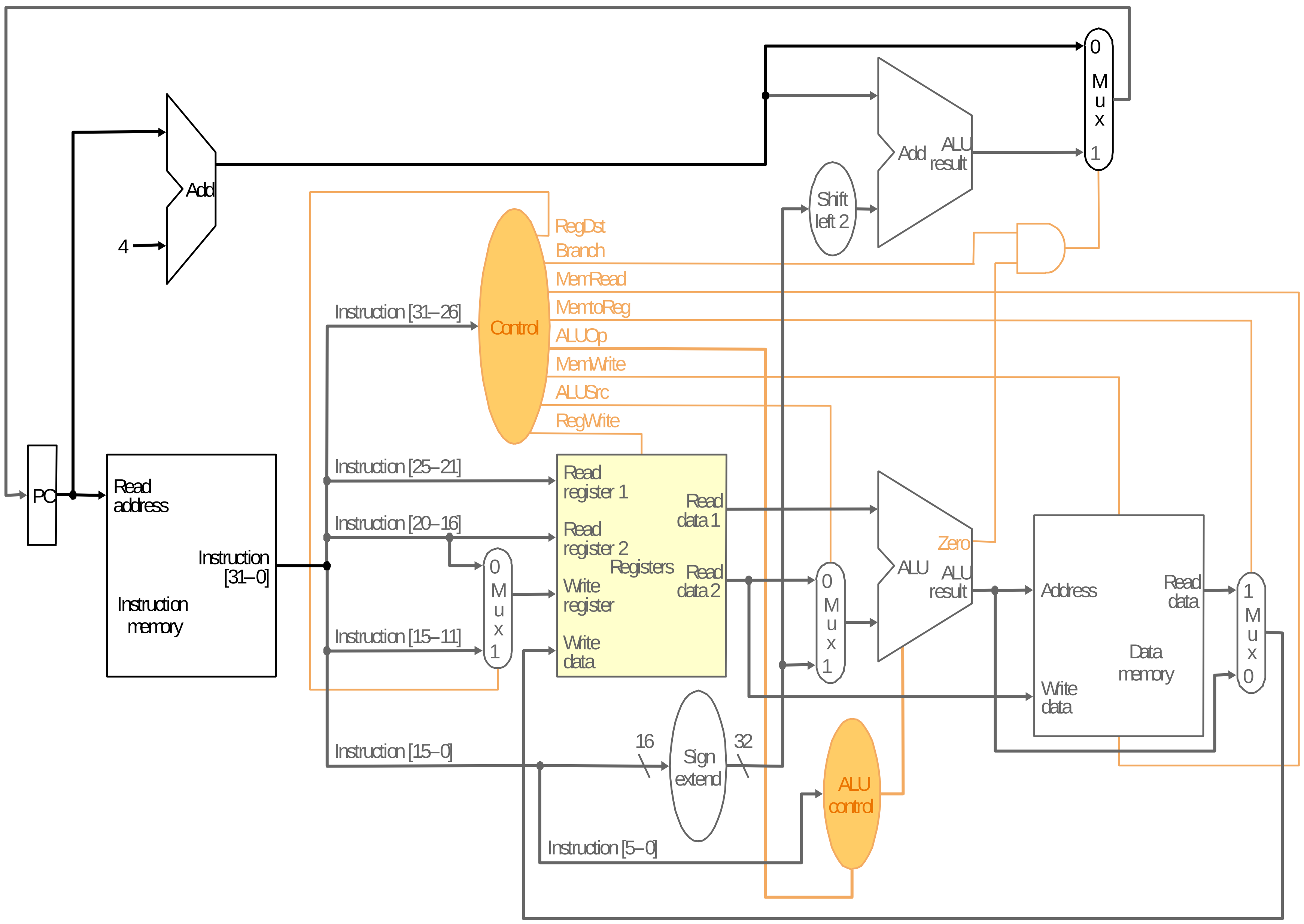
Control: 2 level implementation



00: lw, sw
01: beq
10: add, sub, and, or, slt

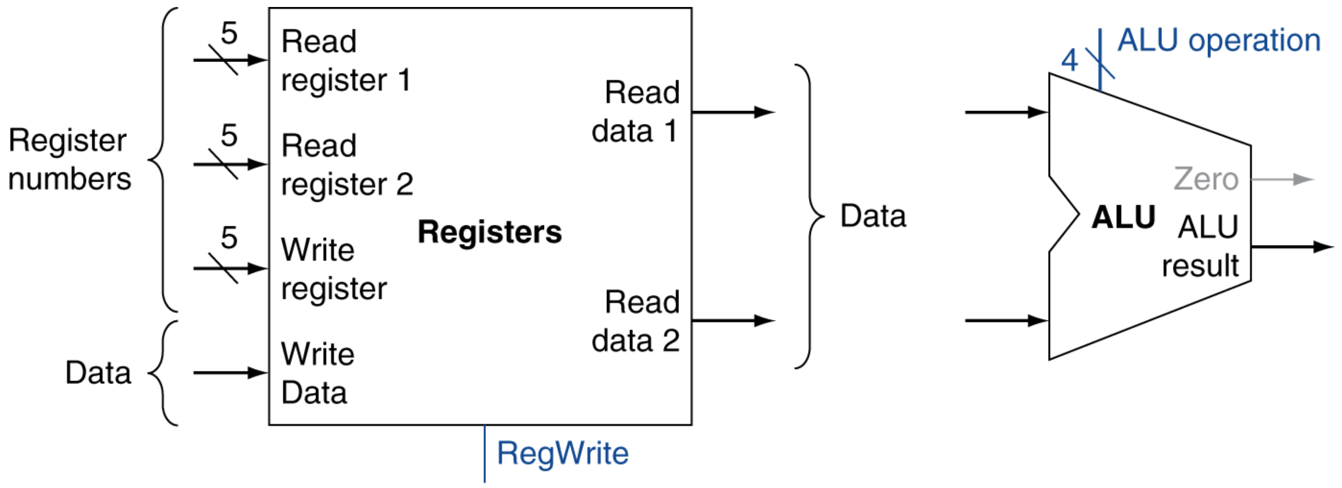
000: and
001: or
010: add
110: sub
111: set on less than

Datapath with Control



R-Format Instructions

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

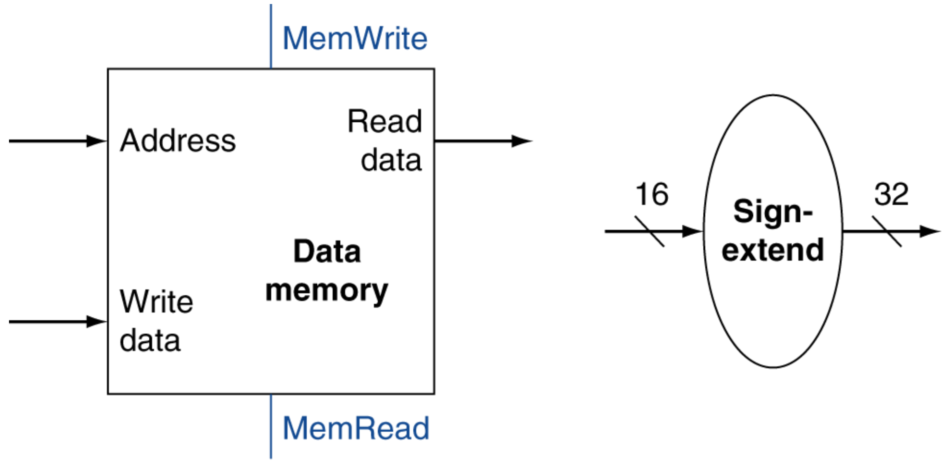


a. Registers

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

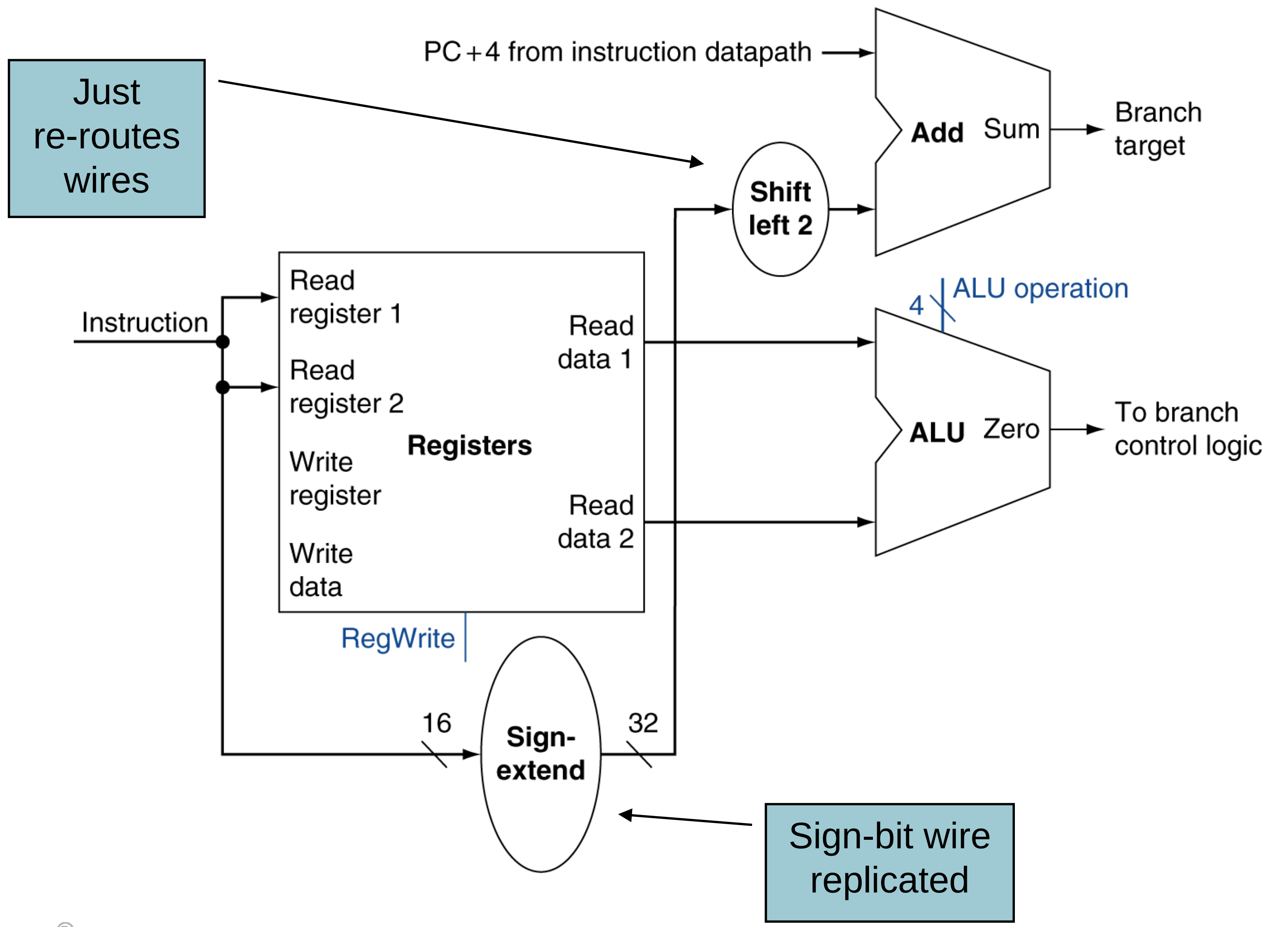


a. Data memory unit b. Sign extension unit

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



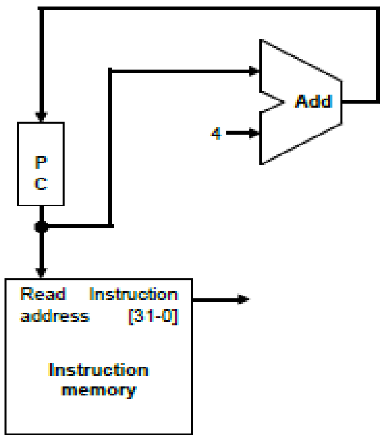
Composing the Elements

- 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

Single Cycle Processing

Instruction fetching

- The CPU is always in an infinite loop, fetching instructions from memory and executing them.
- The program counter or PC register holds the address of the current instruction.
- MIPS instructions are each four bytes long, so the PC should be incremented by four to read the next instruction in sequence.



Reading Word-Addressable Memory

- Memory read called ***load***
- **Mnemonic:** *load word* (`lw`)
- **Format:**

```
lw $s0, 5($t1)
```

- **Address calculation:**
  - add *base address* (`$t1`) to the *offset* (5)
  - address = (`$t1` + 5)
- **Result:**
  - `$s0` holds the value at address (`$t1` + 5)

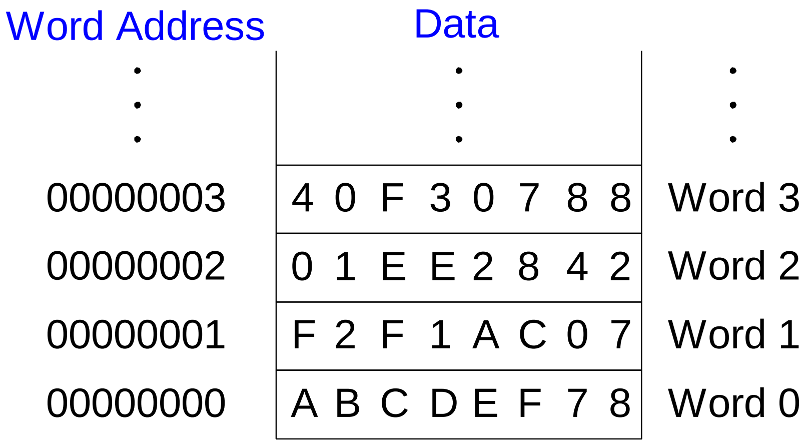
Any register may be used as base address

Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into `$s3`
  - address = (`$0` + 1) = 1
  - `$s3` = 0xF2F1AC07 after load

Assembly code

```
lw $s3, 1($0)   # read memory word 1 into $s3
```



Writing Word-Addressable Memory

- Memory write are called ***store***
- **Mnemonic:** *store word* (`sw`)

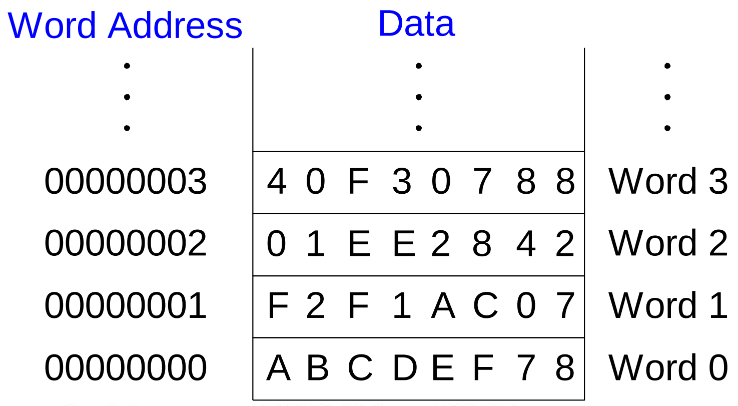
Writing Word-Addressable Memory

- **Example:** Write (store) the value in `$t4` into memory address 7
  - add the base address (`$0`) to the offset (0x7)
  - address: ($0 + 0x7) = 7

Offset can be written in decimal (default) or hexadecimal

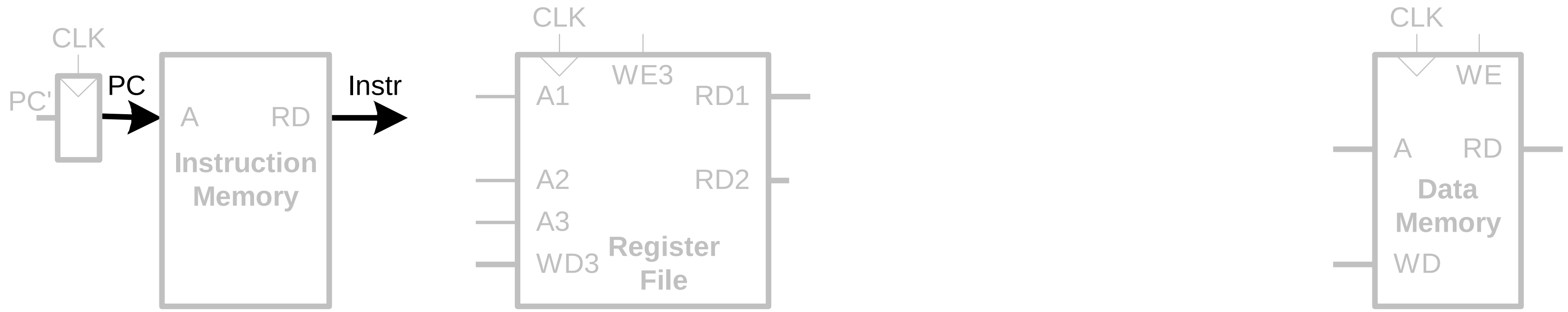
Assembly code

```
sw $t4, 0x7($0)   # write the value in $t4
                  # to memory word 7
```



Single-Cycle Datapath: `lw` fetch

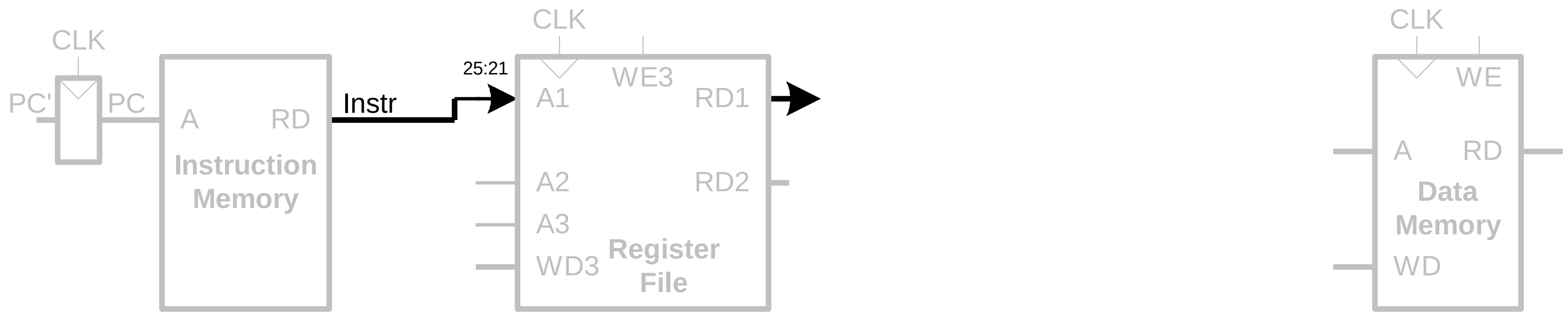
STEP 1: Fetch instruction



Single-Cycle Datapath: `lw` Register Read

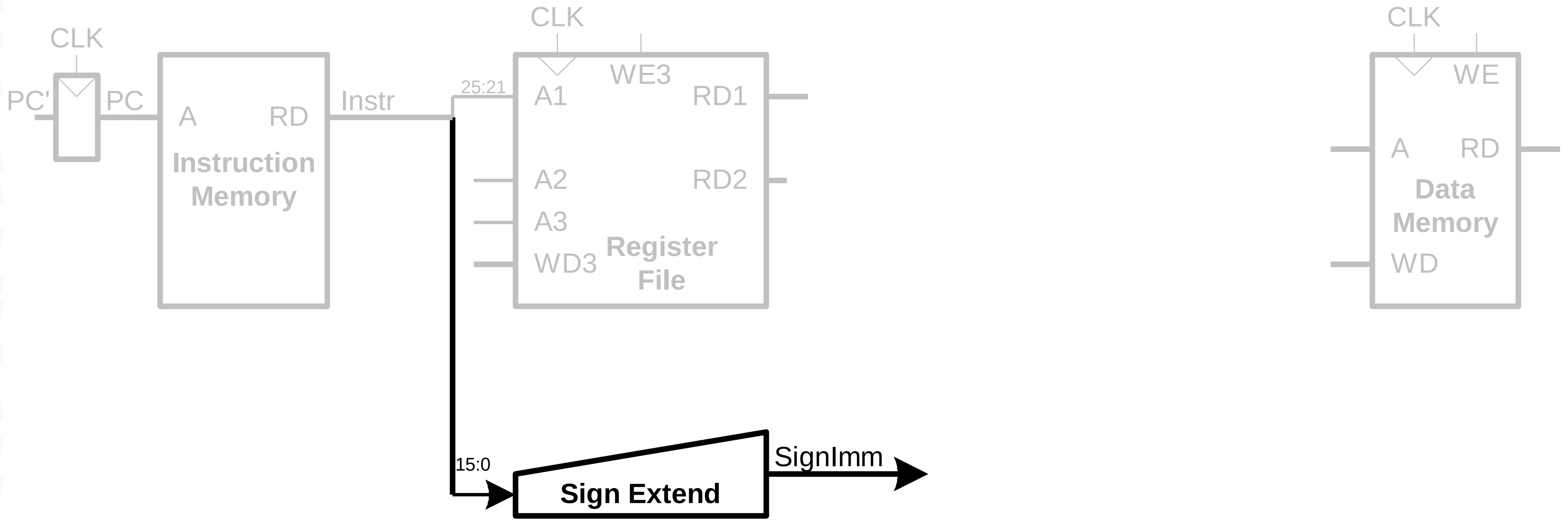
STEP 2: Read source operands from RF

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]



Single-Cycle Datapath: `lw` Immediate

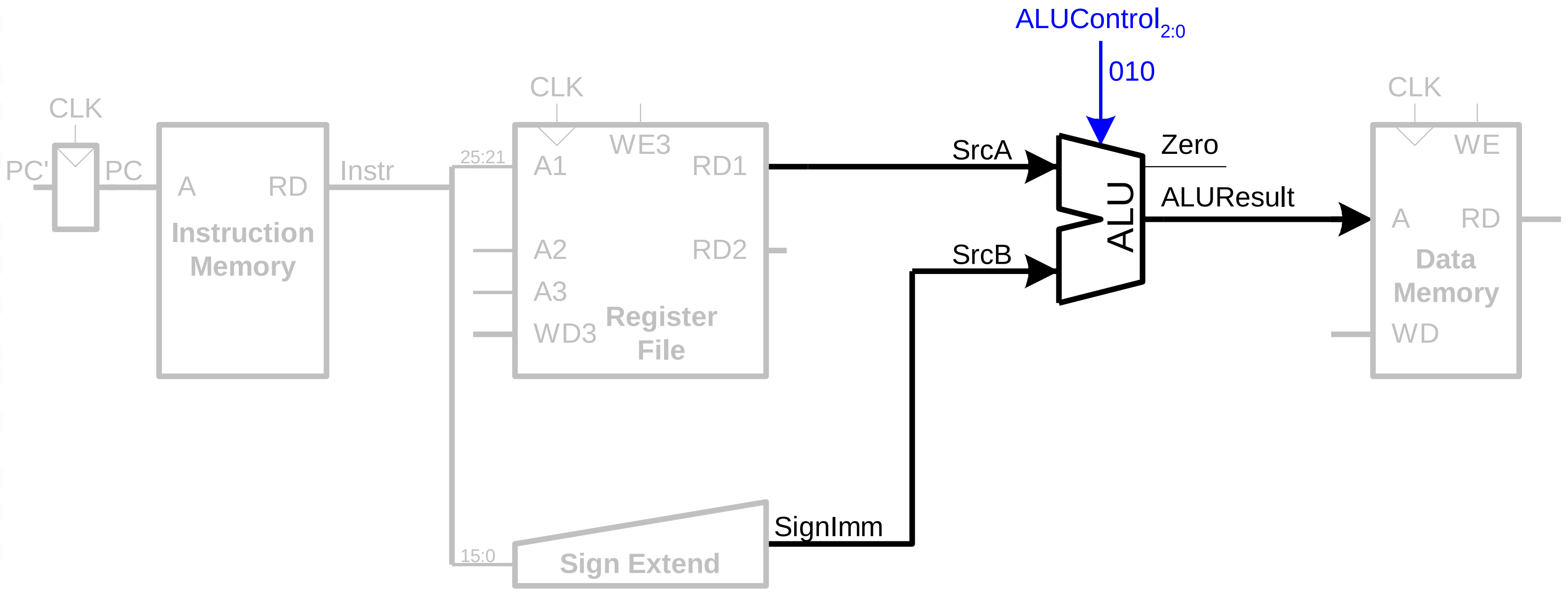
STEP 3: Sign-extend the immediate



ELSEVIER

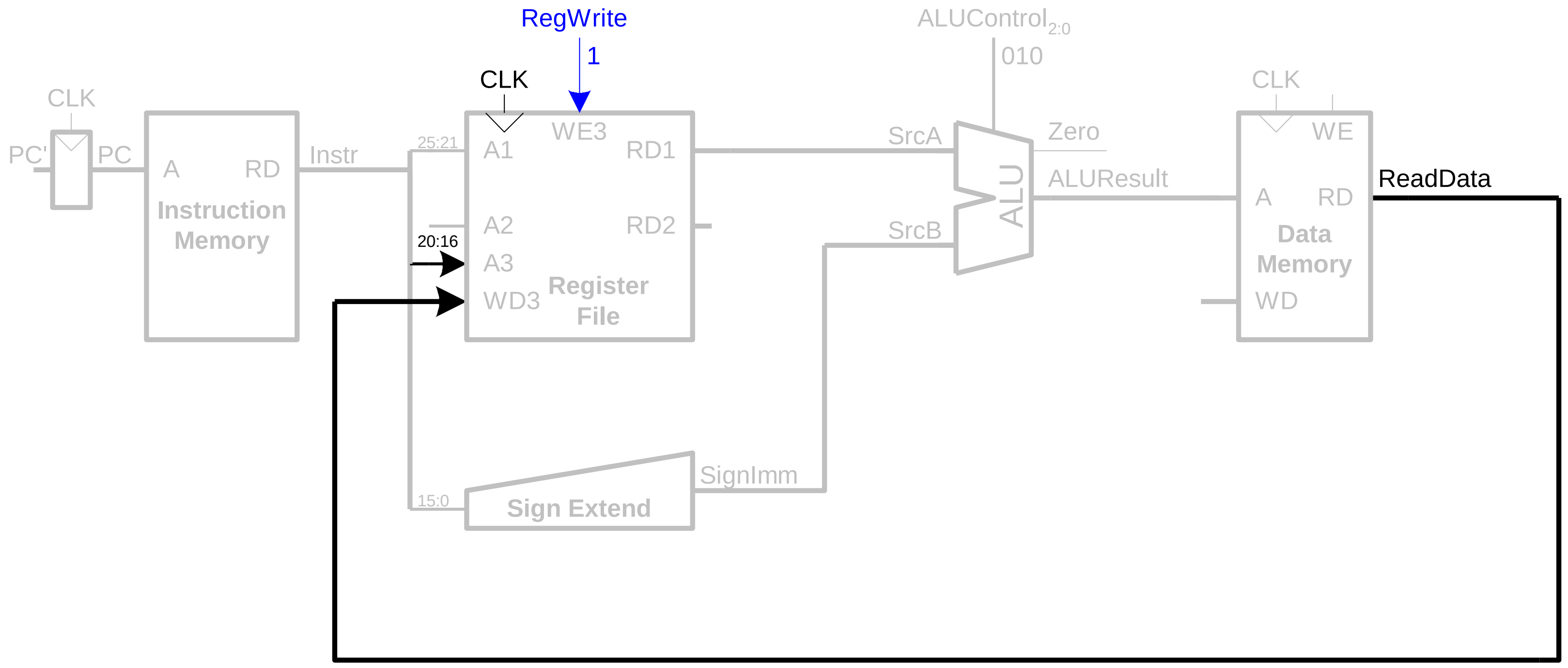
Single-Cycle Datapath: `lw` address

STEP 4: Compute the memory address



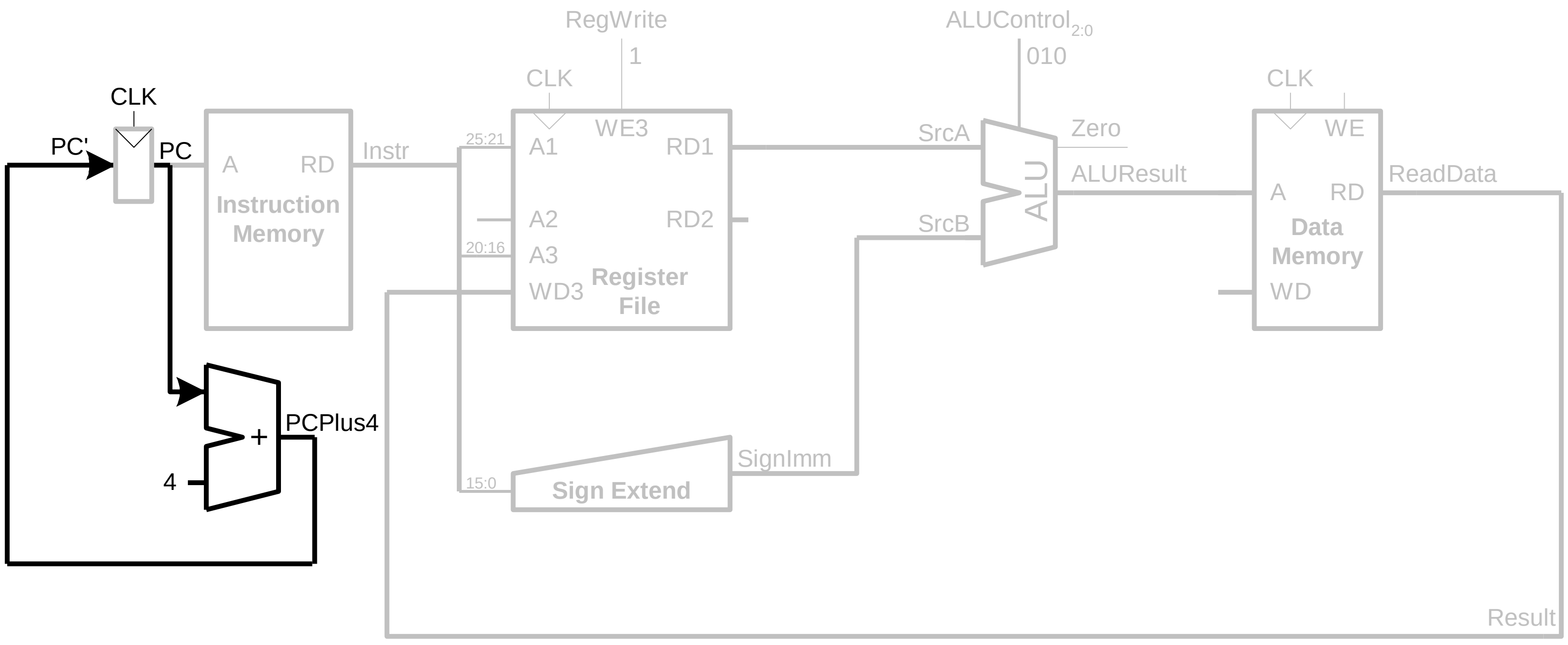
Single-Cycle Datapath: `lw` Memory Read

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



Single-Cycle Datapath: `lw` PC Increment

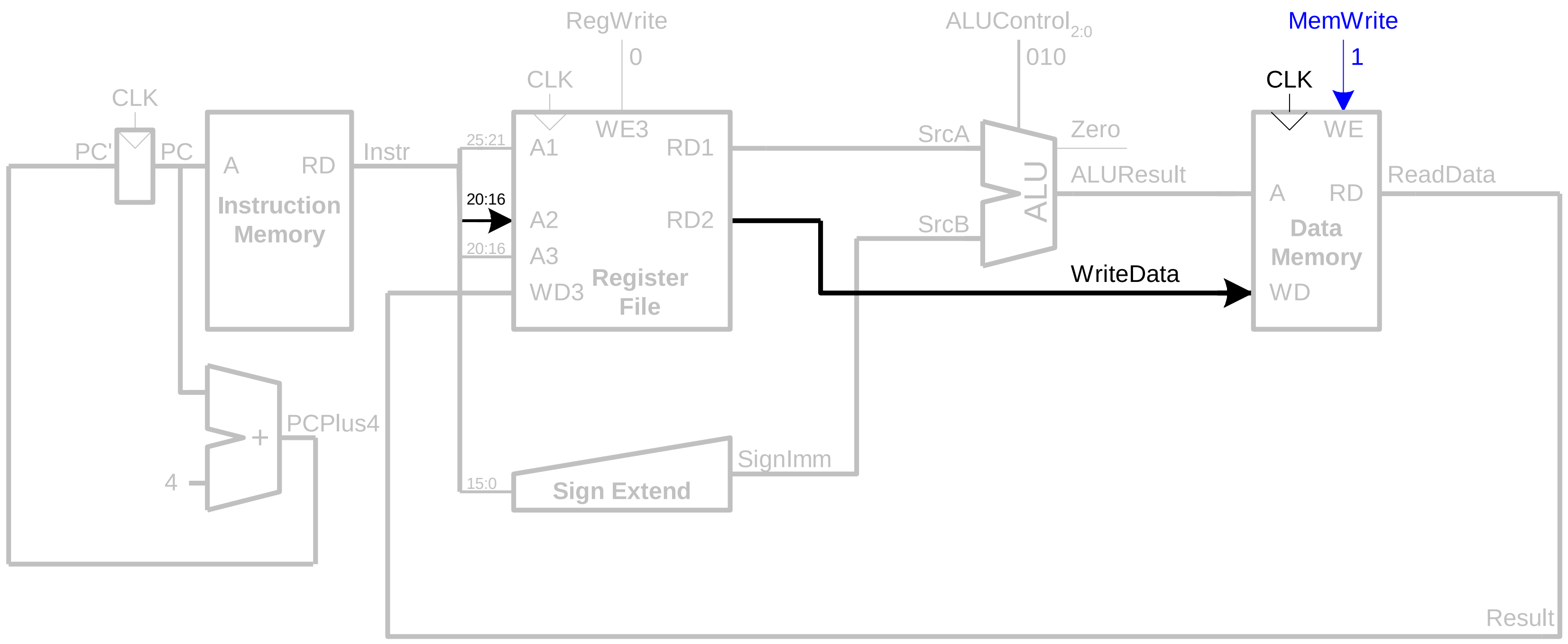
STEP 6: Determine address of next instruction



Single-Cycle Datapath: `sw`

Write data in `rt` to memory

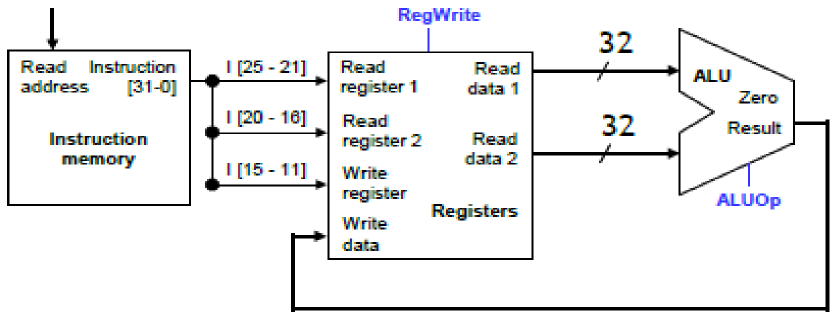
DMEM[R[rs] + sign_ext(Imm16)] ← R[rt]



Single Cycle Processing

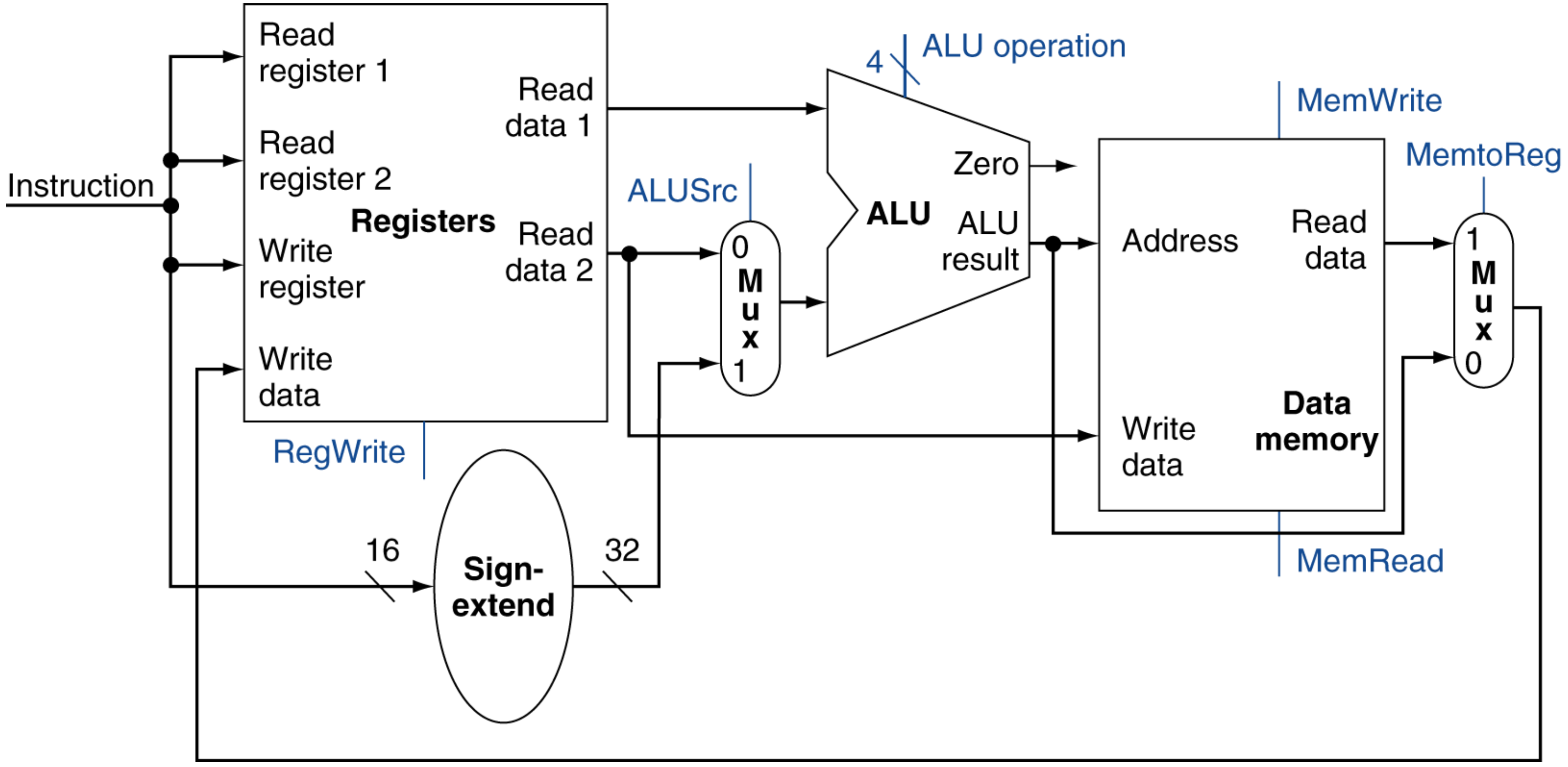
Executing an R-type instruction

1. Read an instruction from the instruction memory.
2. The source registers, specified by instruction fields rs and rt, should be read from the register file.
3. The ALU performs the desired operation.
4. Its result is stored in the destination register, which is specified by field rd of the instruction word.

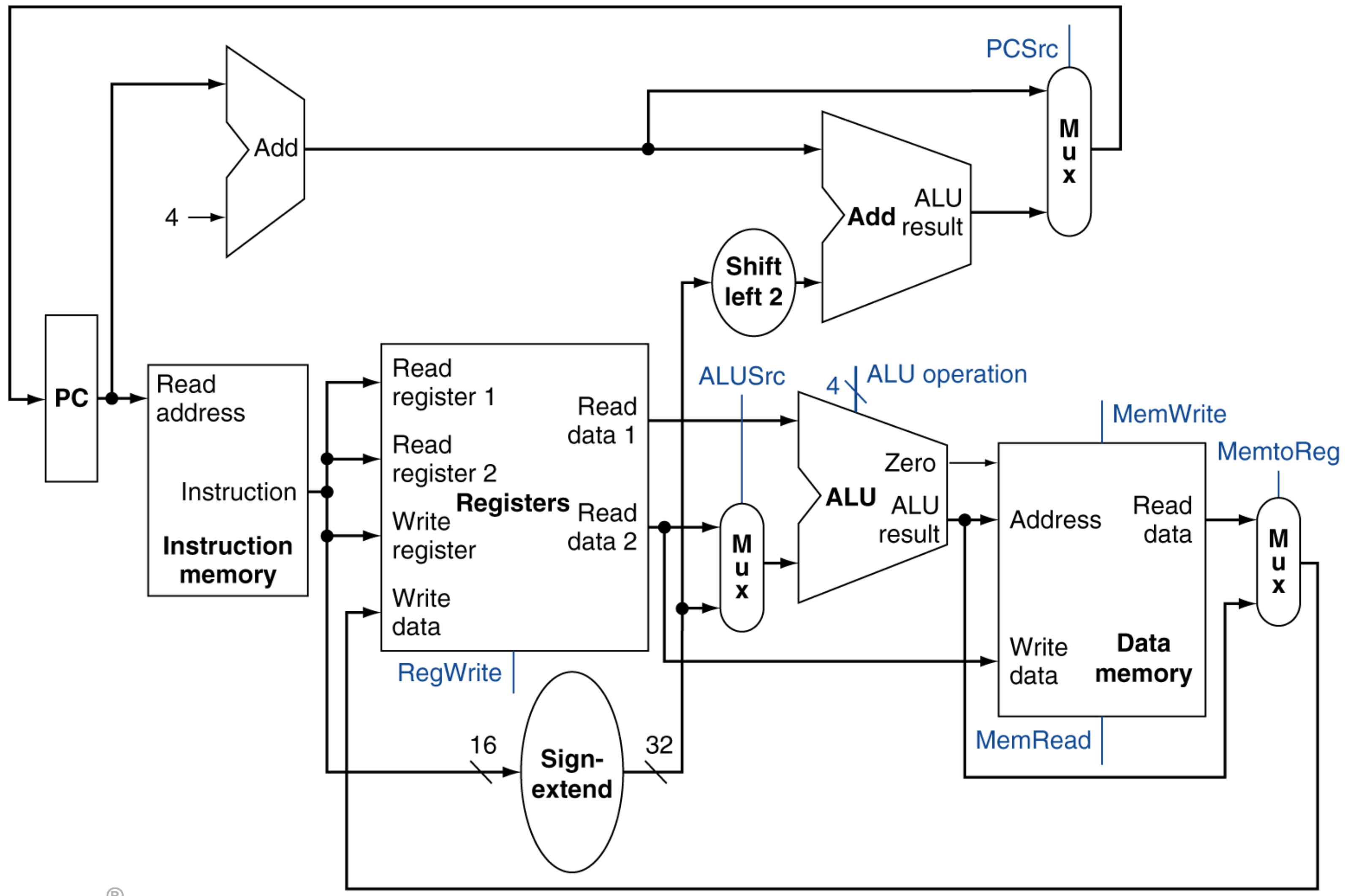


op	rs	rt	rd	shamt	func
31 26	25 21	20 16	15 11	10 6	5 0

R-Type/Load/Store Datapath



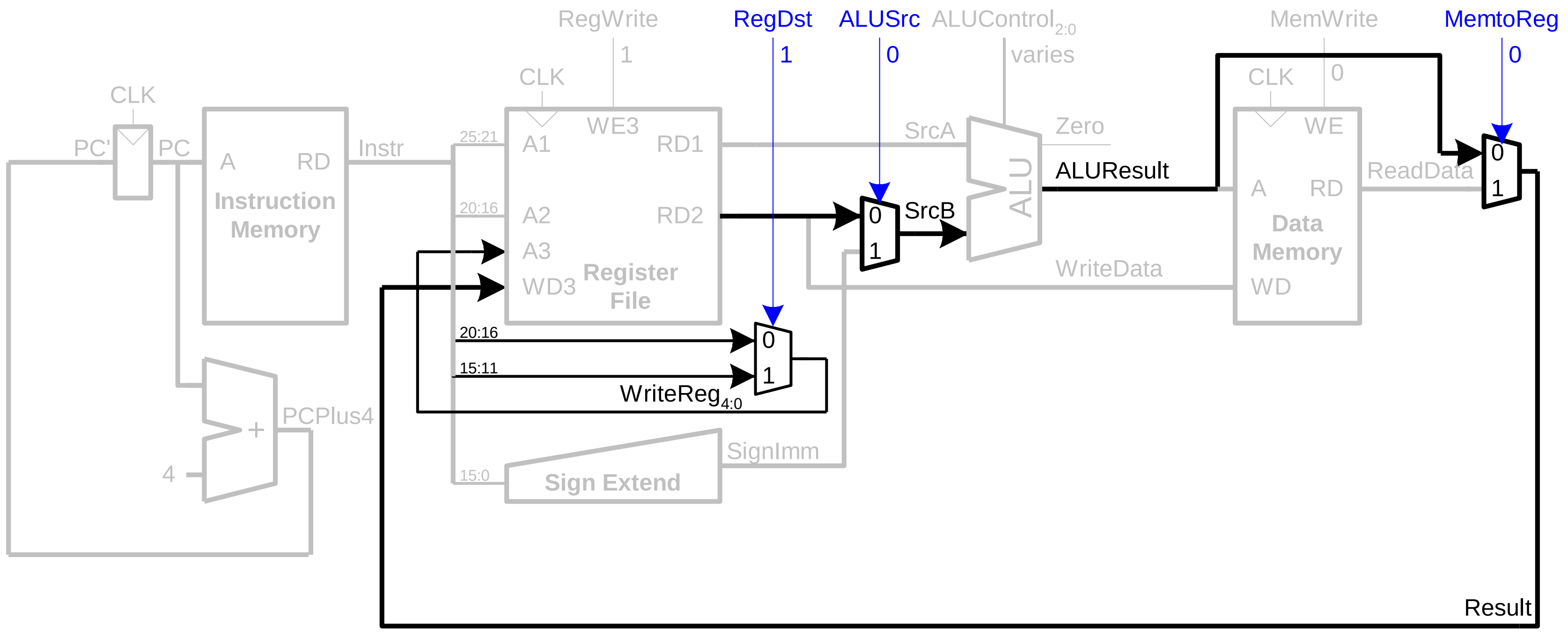
Full Datapath



Single-Cycle Datapath: R-Type

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

R[rd] ← R[rs] op R[rt]

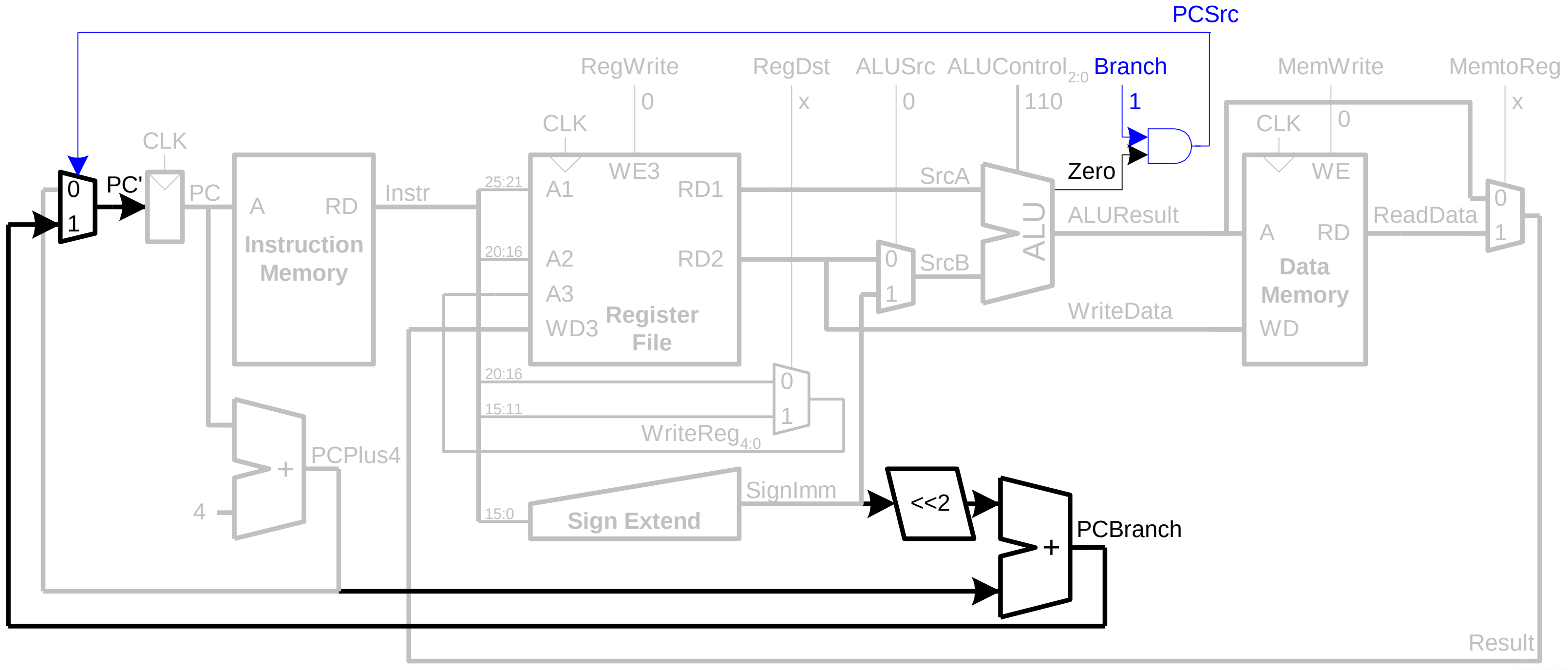


ELSEVIER

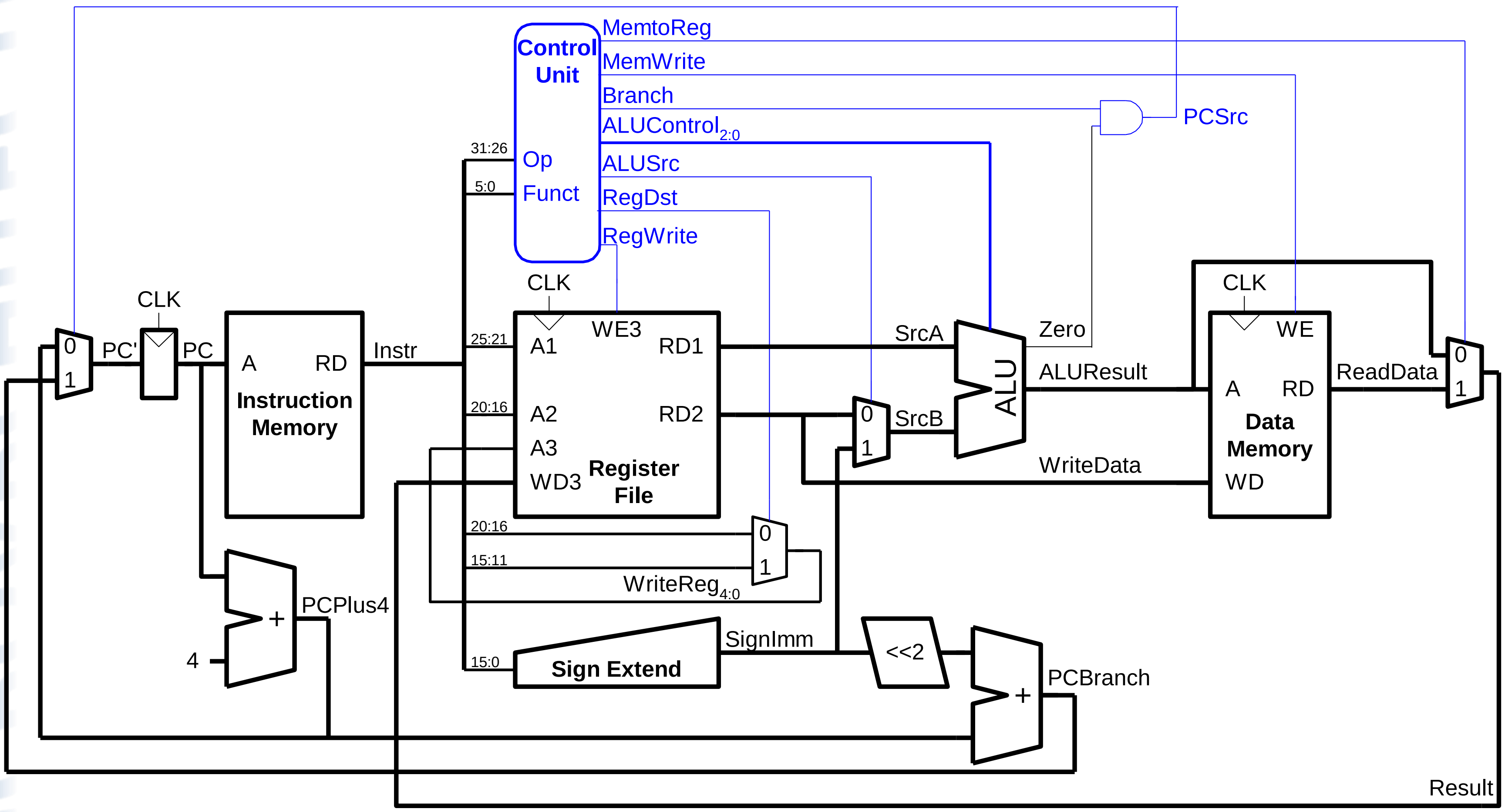
Single-Cycle Datapath: `beq`

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

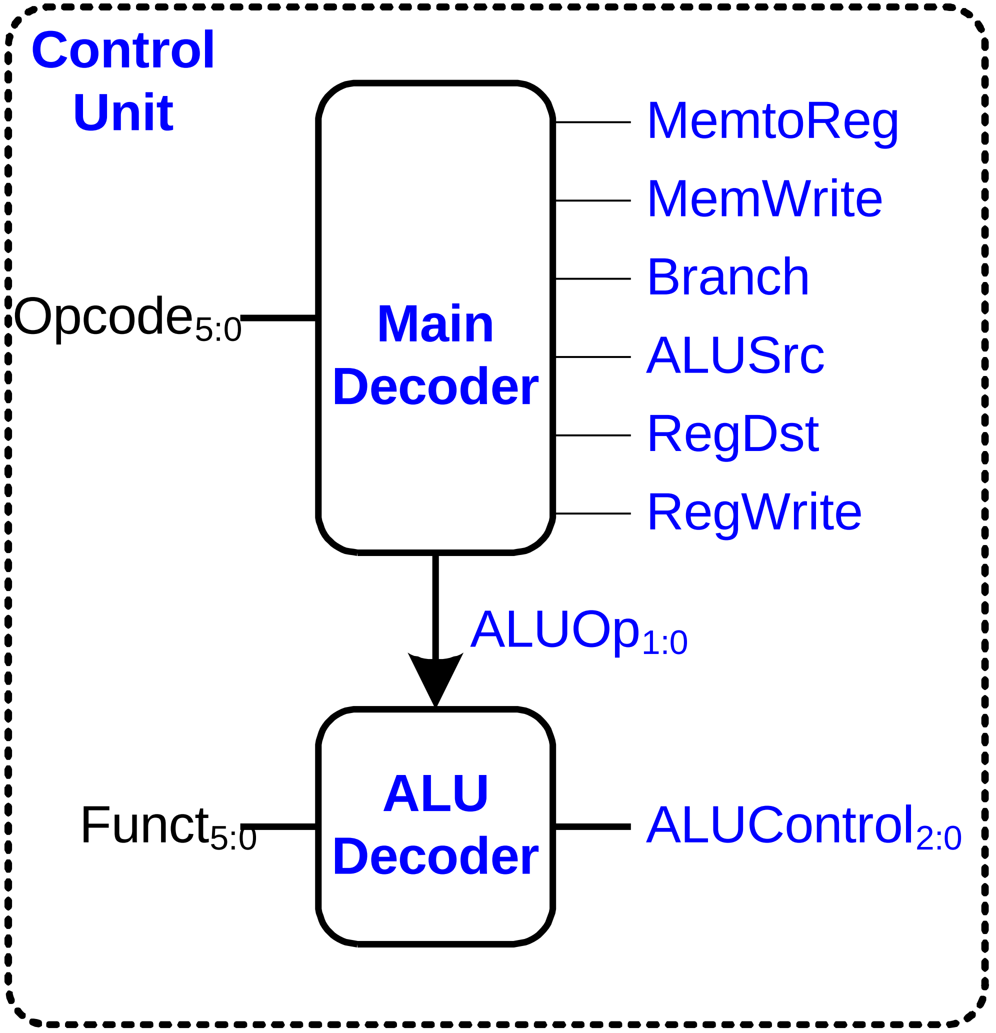
  BTA = (sign-extended immediate << 2) + (PC+4)



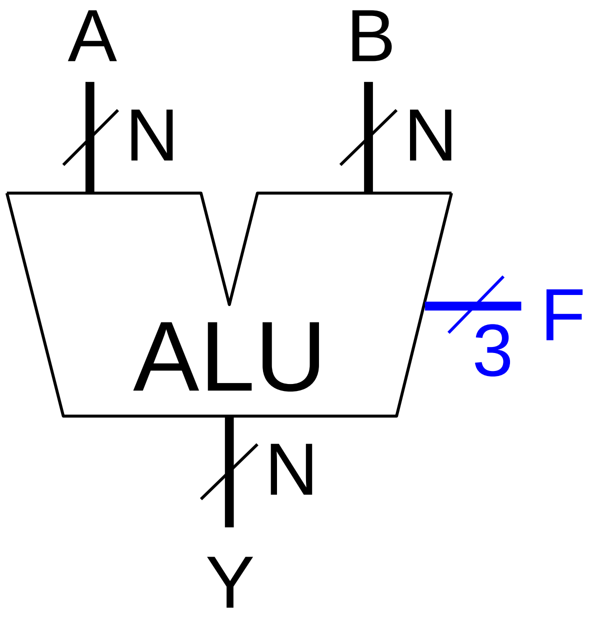
Single-Cycle Processor



Single-Cycle Control

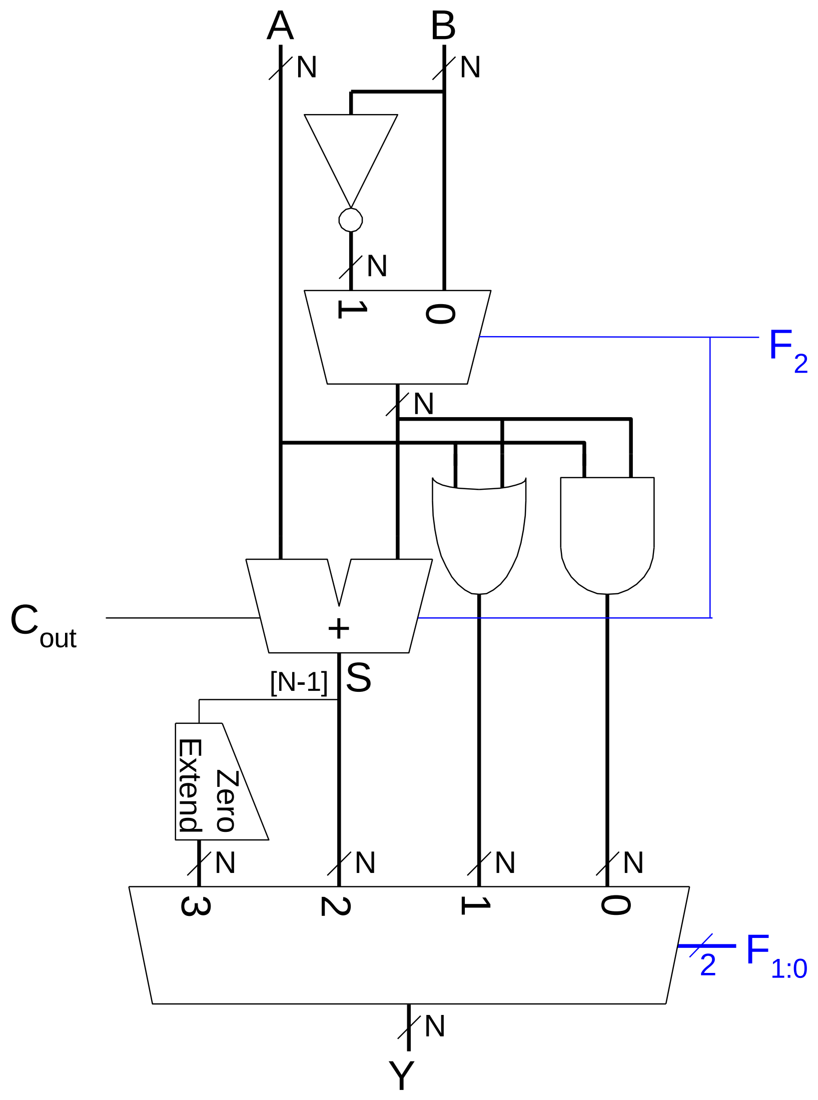


Review: ALU



$F_{2:0}$	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)

Multicycle MIPS Processor

- **Single-cycle:**
  - + simple
  - - cycle time limited by longest instruction (`lw`)
  - - 2 adders/ALUs & 2 memories
- **Multicycle:**
  - + higher clock speed
  - + simpler instructions run faster
  - + reuse expensive hardware on multiple cycles
  - - sequencing overhead paid many times
- **Same design steps: datapath & control**

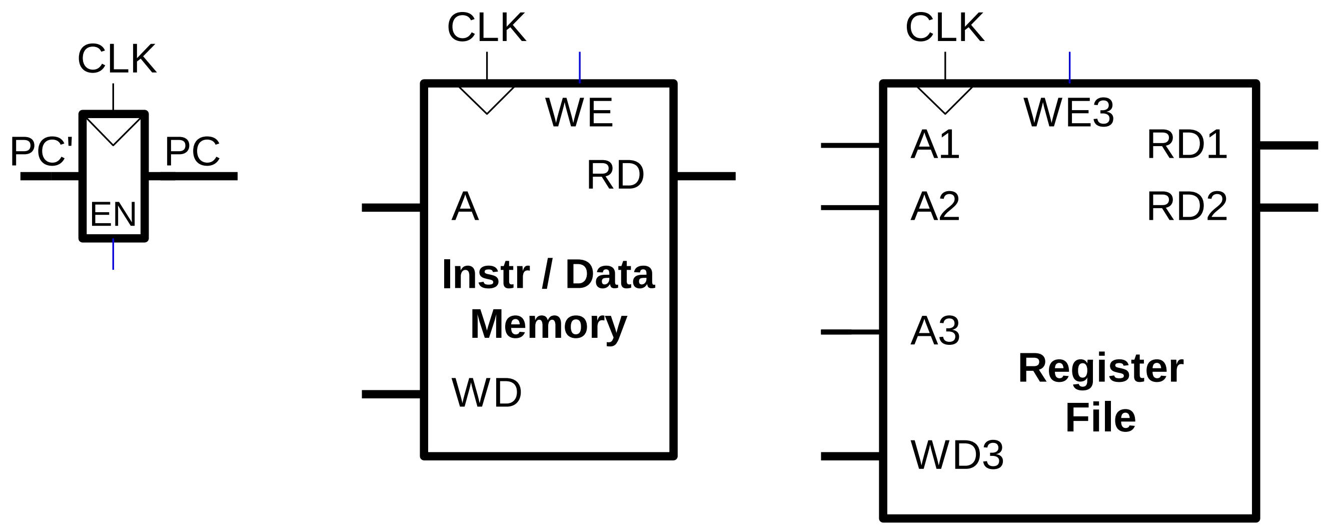
Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers

- Notice: we distinguish
  - processor state: programmer visible registers
  - internal state: programmer invisible registers (like IR, MDR, A, B, and ALUout)

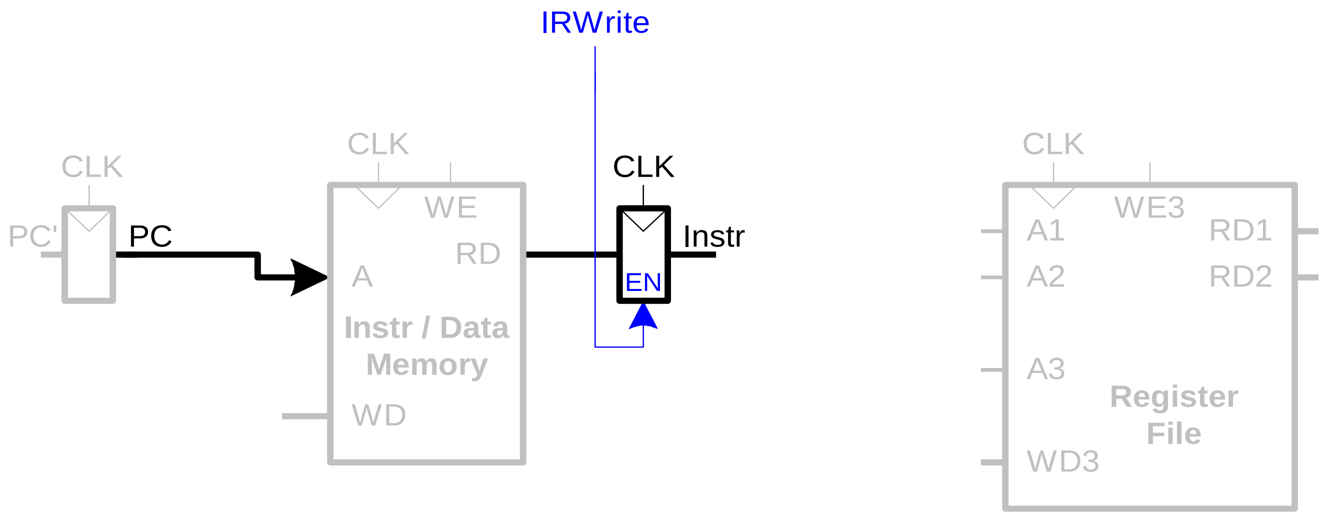
Multicycle State Elements

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



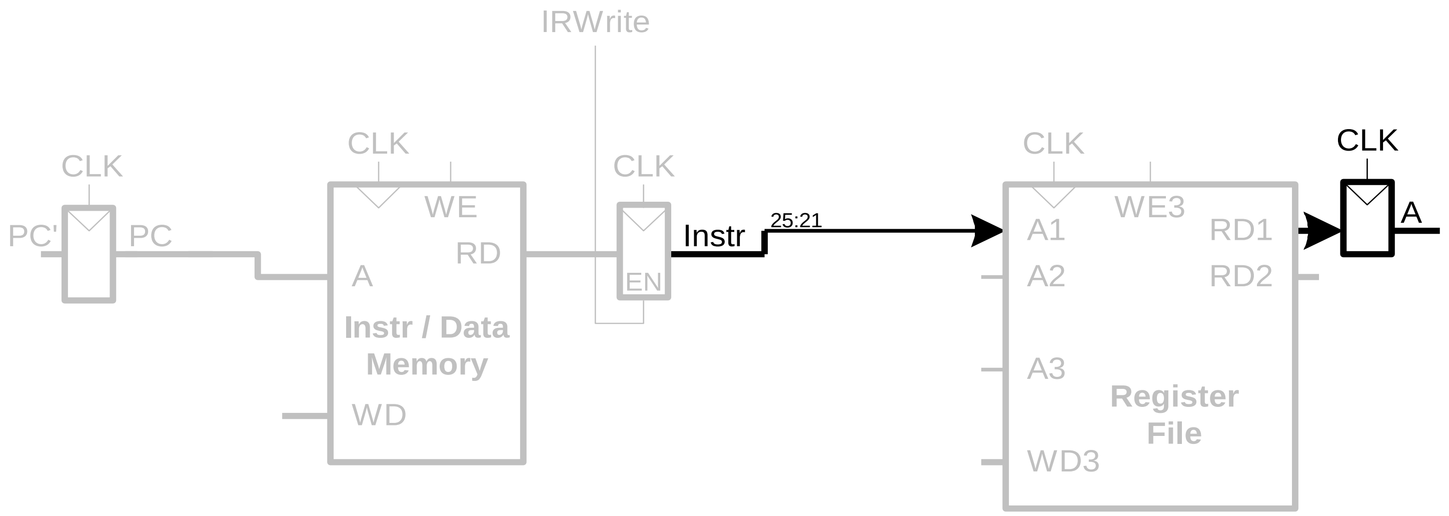
Multicycle Datapath: Instruction Fetch

STEP 1: Fetch instruction



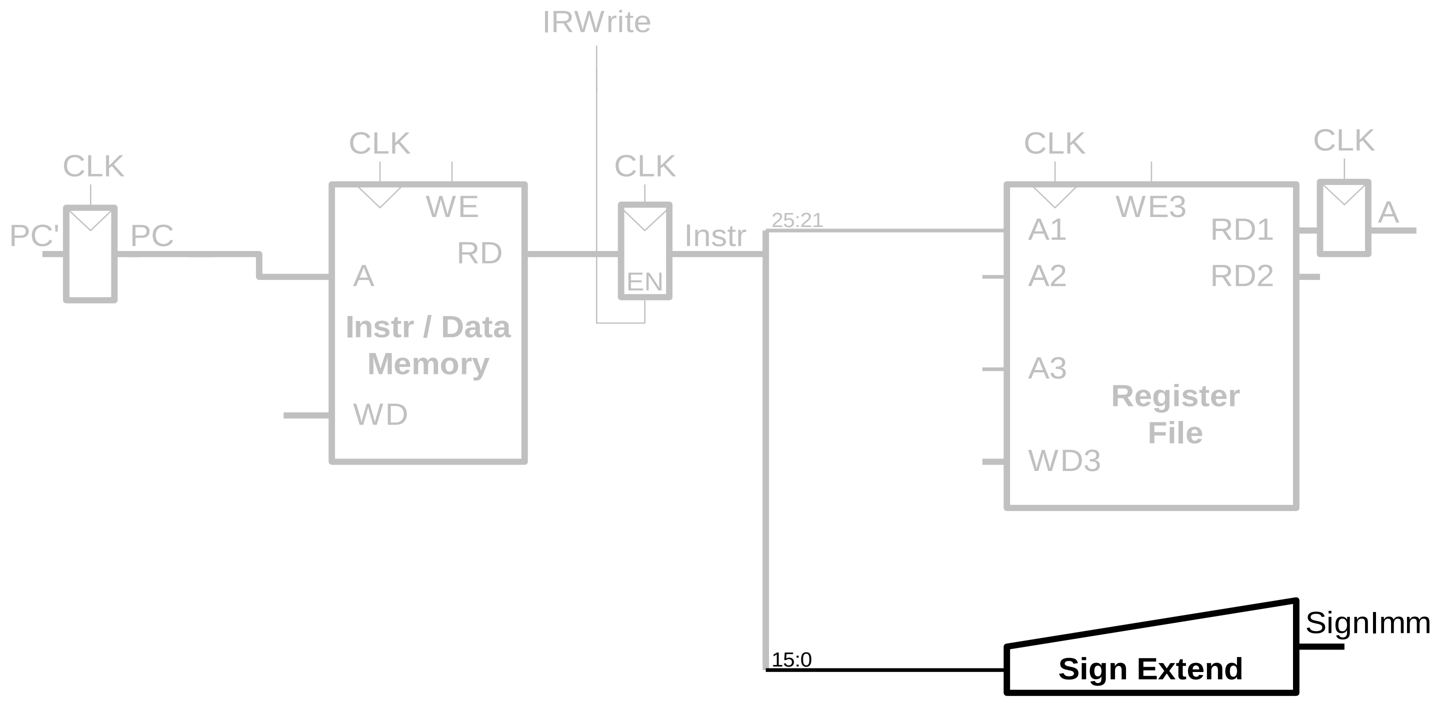
Multicycle Datapath: `lw` Register Read

STEP 2a: Read source operands from RF



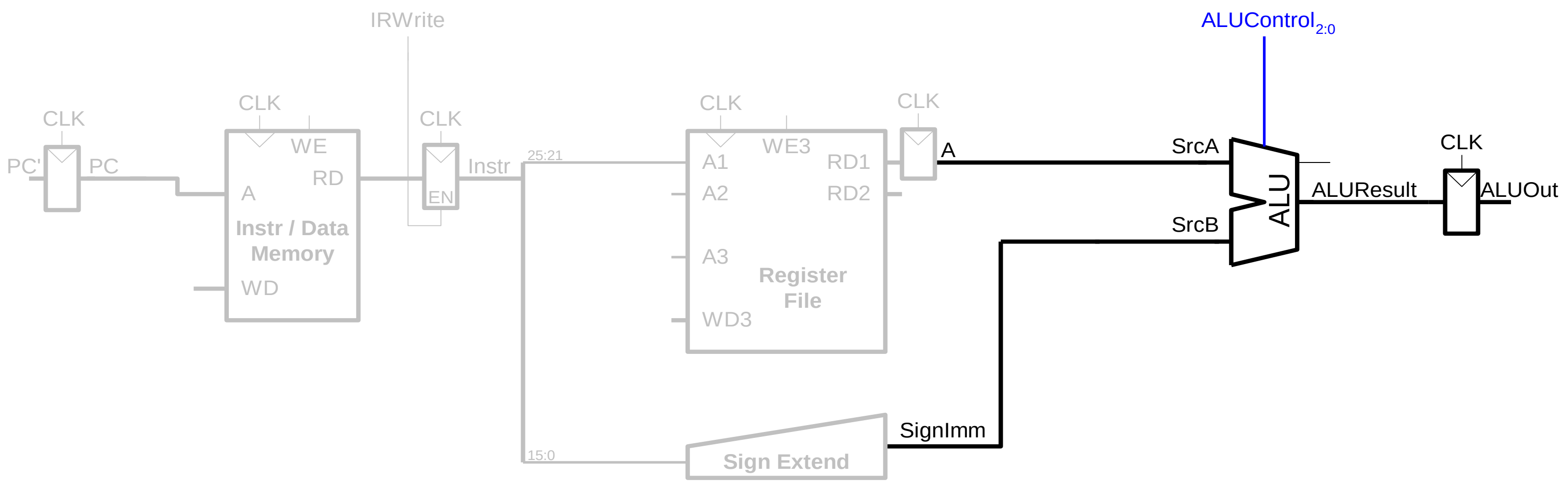
Multicycle Datapath: `lw` Immediate

STEP 2b: Sign-extend the immediate



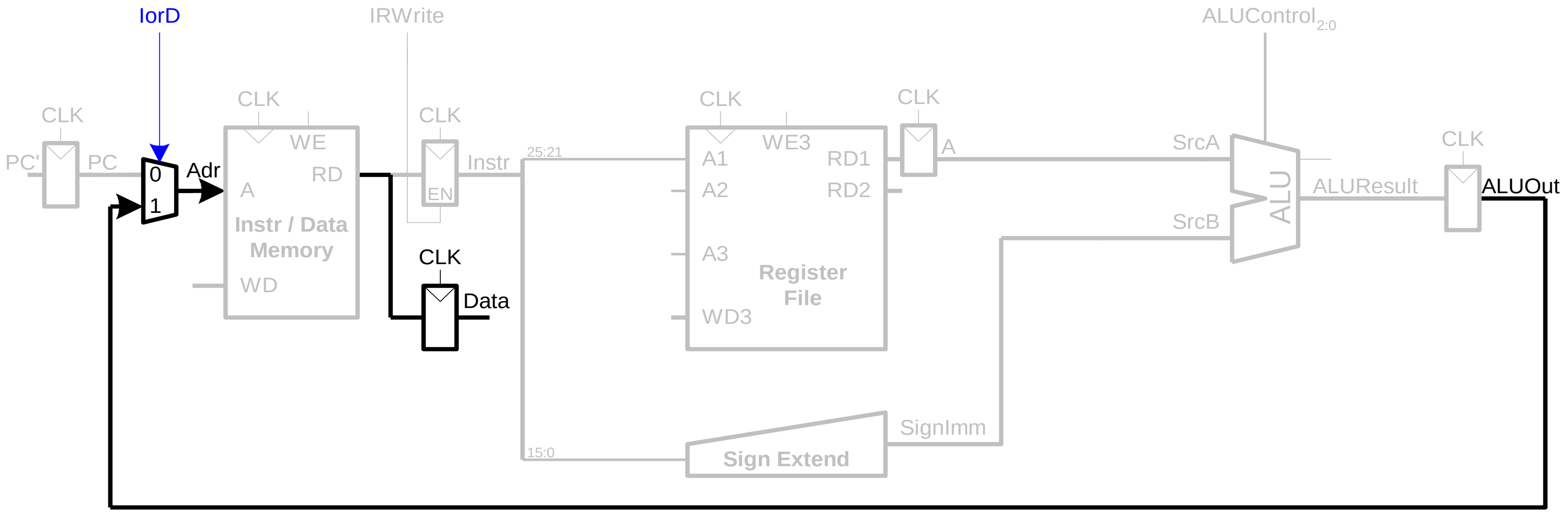
Multicycle Datapath: `lw` Address

STEP 3: Compute the memory address



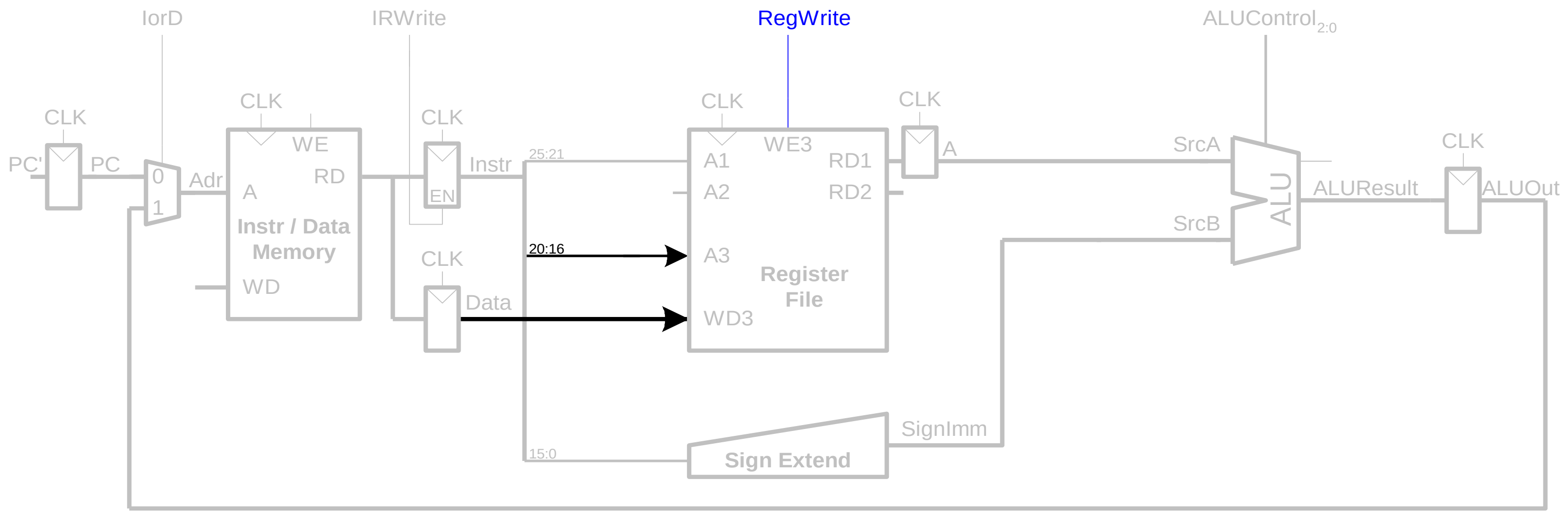
Multicycle Datapath: `lw` Memory Read

STEP 4: Read data from memory



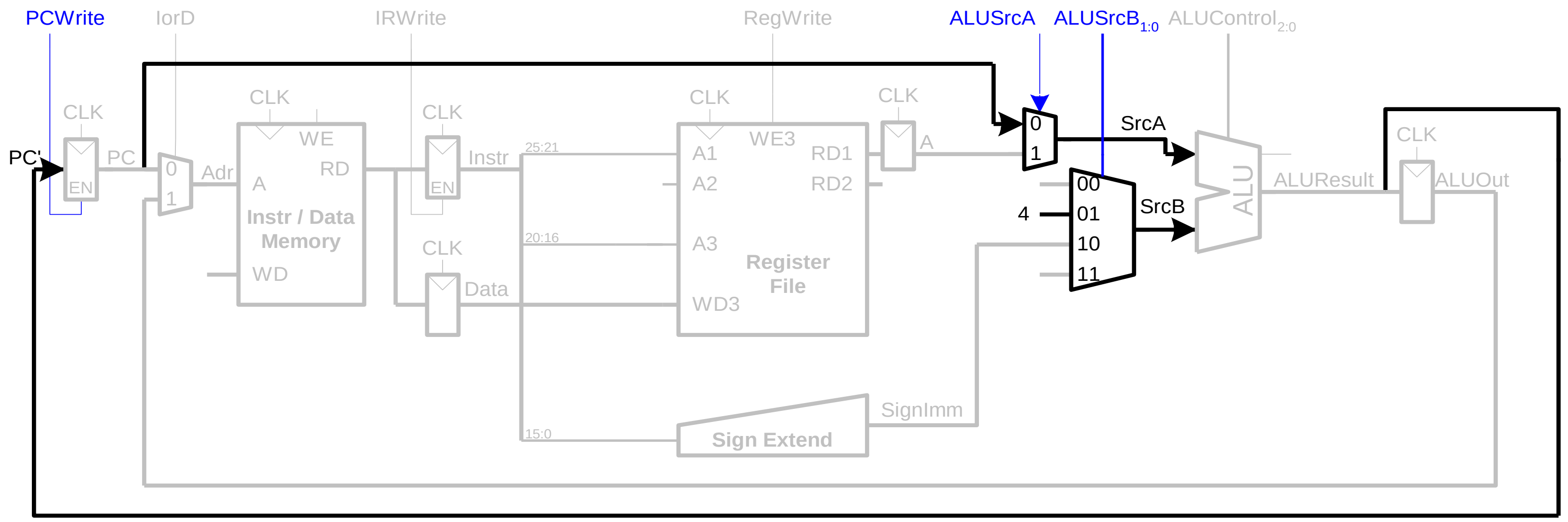
Multicycle Datapath: `lw` Write Register

STEP 5: Write data back to register file



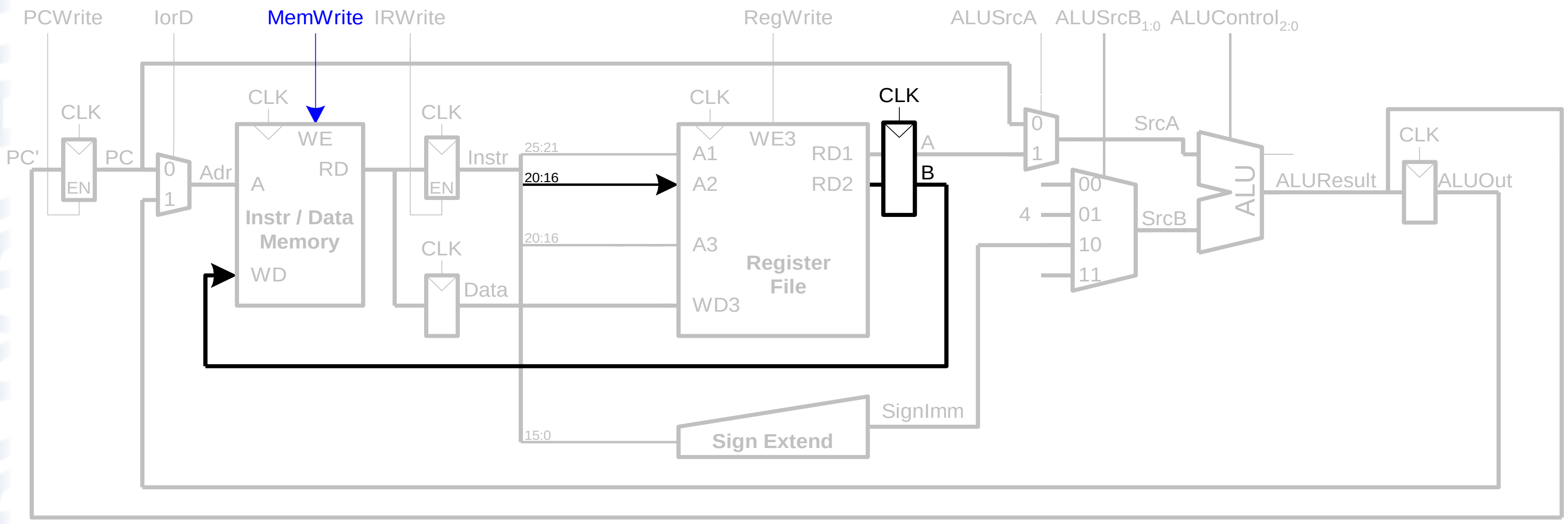
Multicycle Datapath: Increment PC

STEP 6: Increment PC



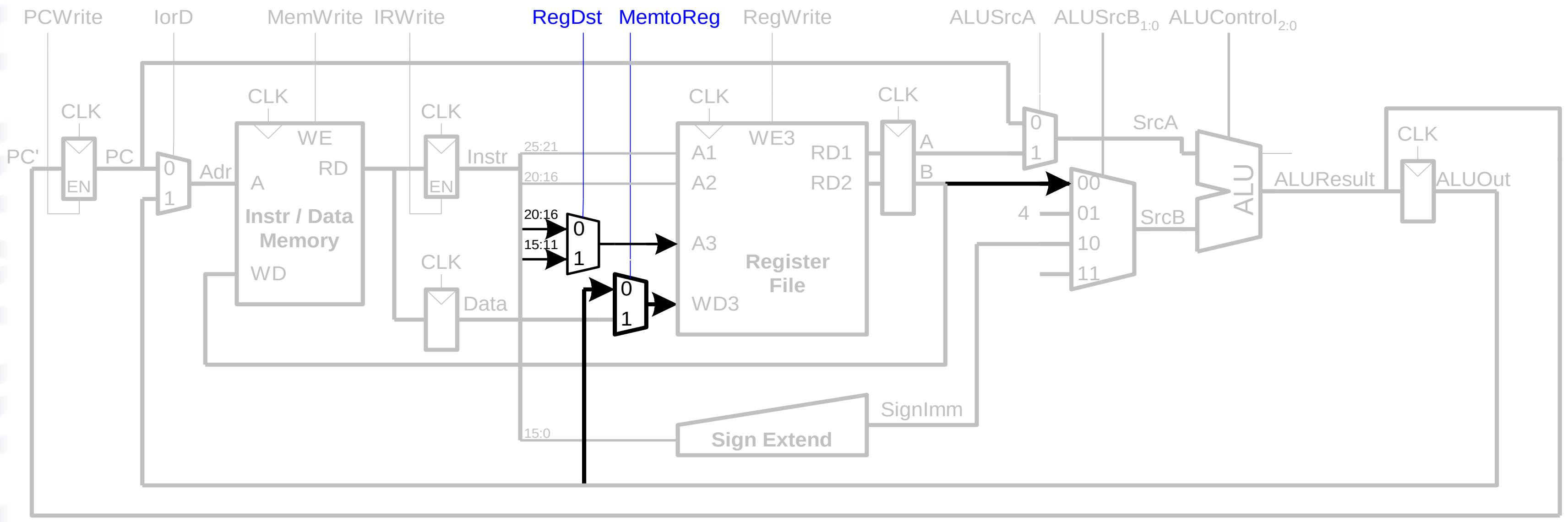
Multicycle Datapath: `sw`

Write data in `rt` to memory

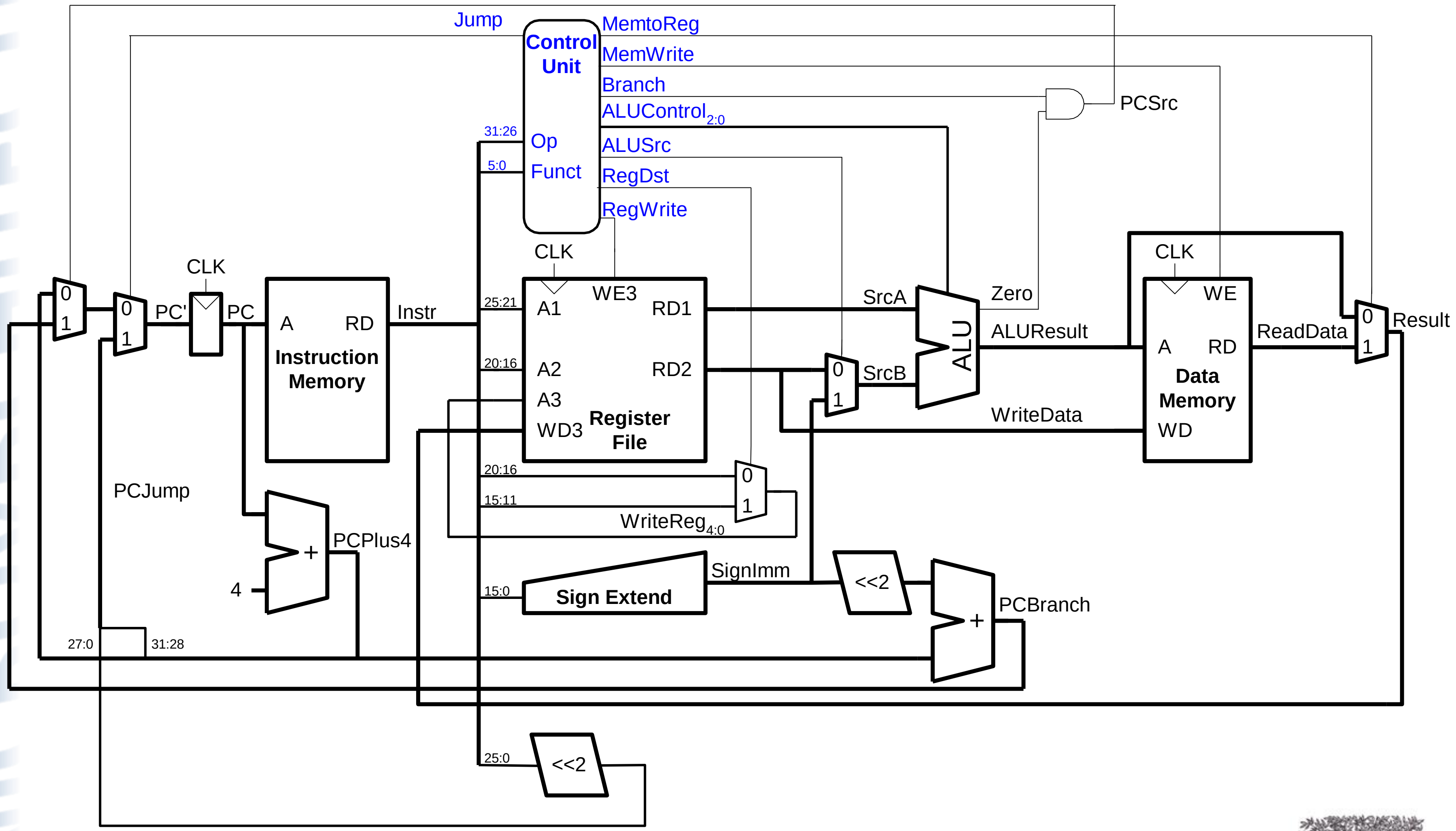


Multicycle Datapath: R-Type

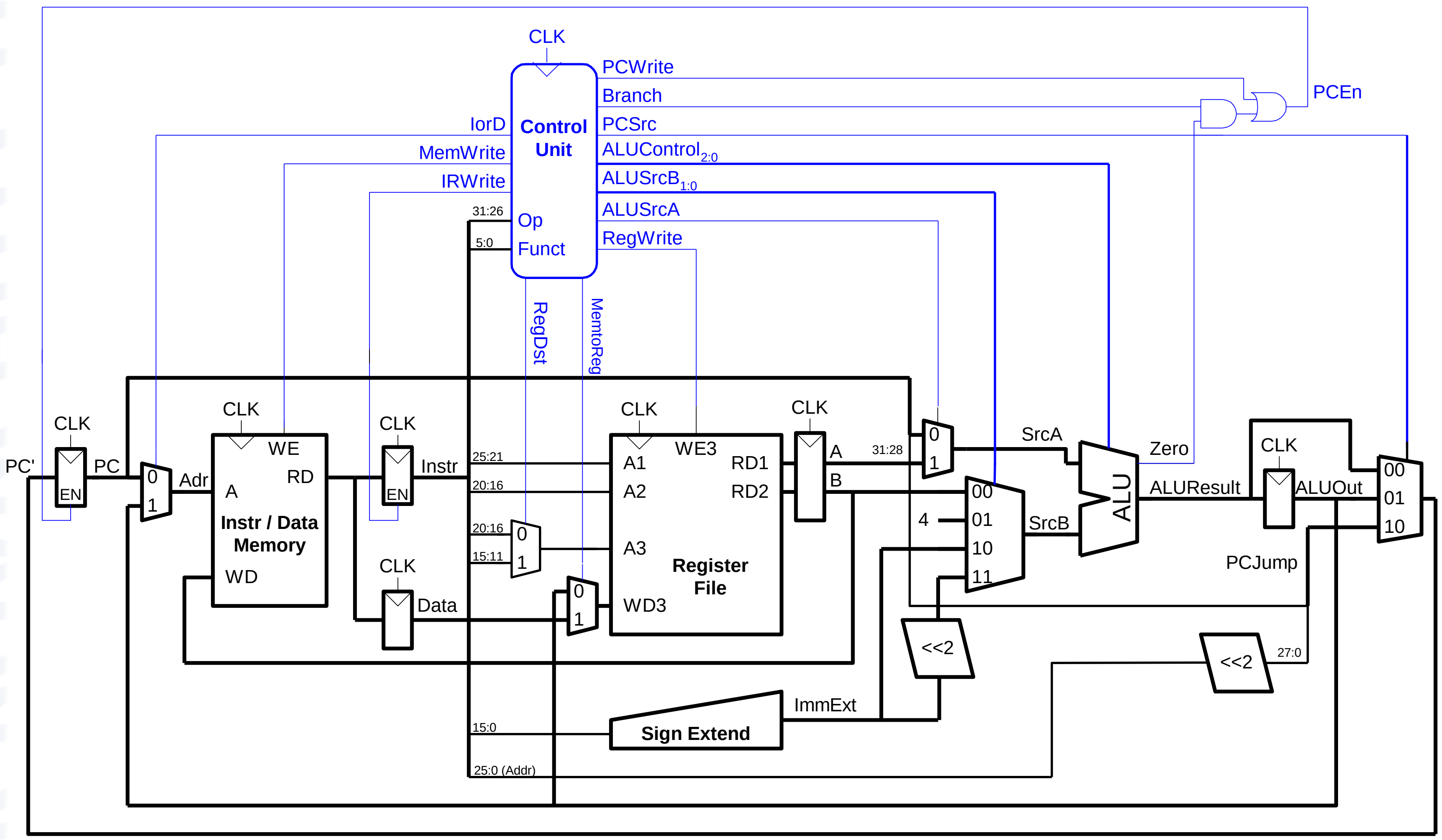
- Read from `rs` and `rt`
- Write *ALUResult* to register file
- Write to `rd` (instead of `rt`)



Review: Single-Cycle Processor

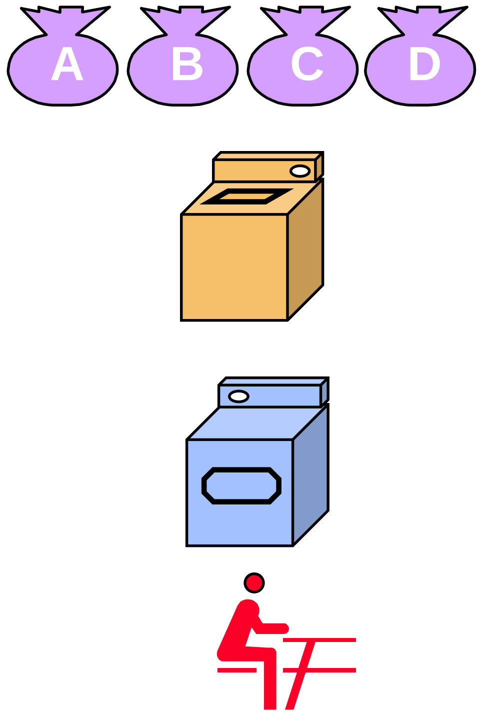


Review: Multicycle Processor

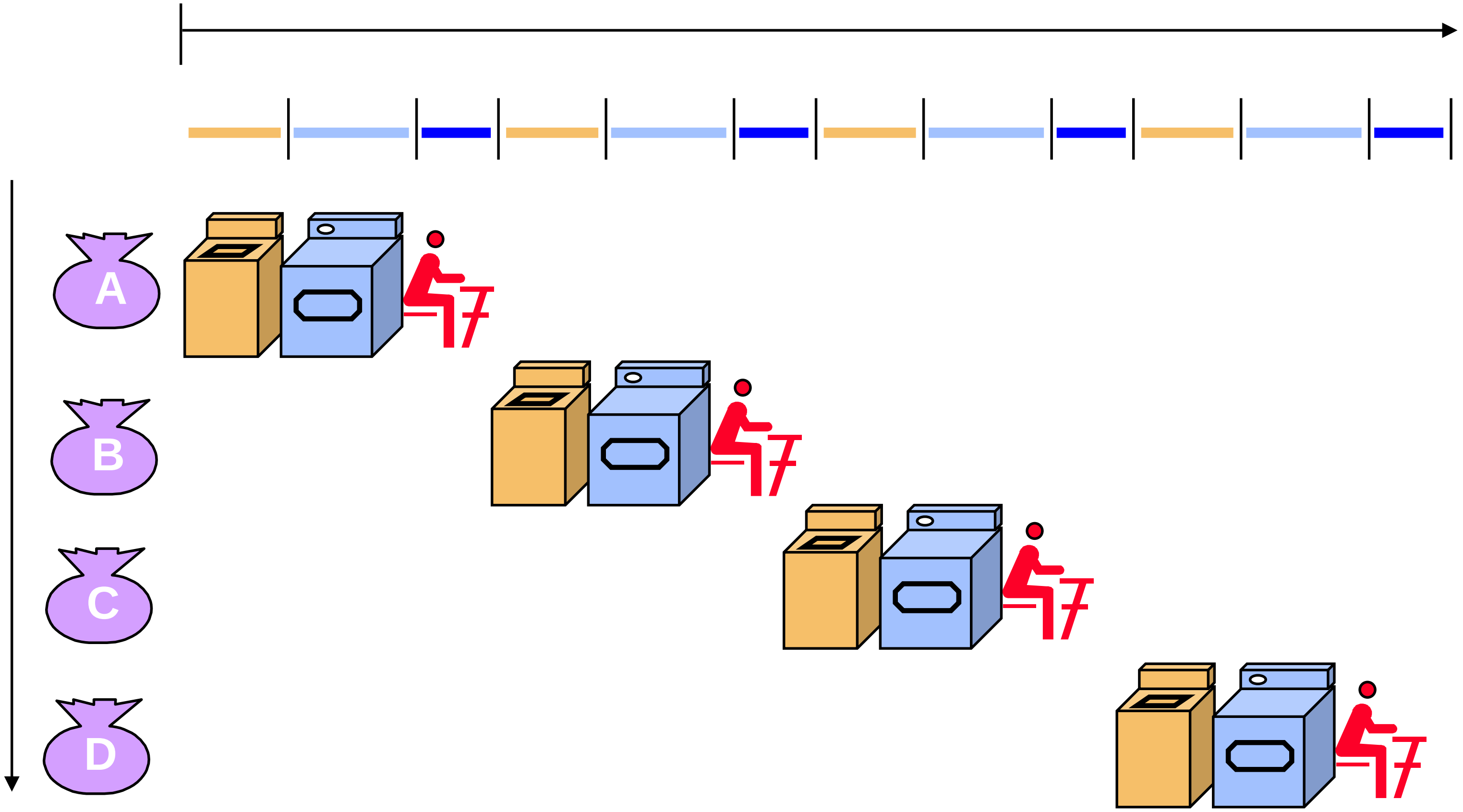


What Is Pipelining

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



What Is Pipelining

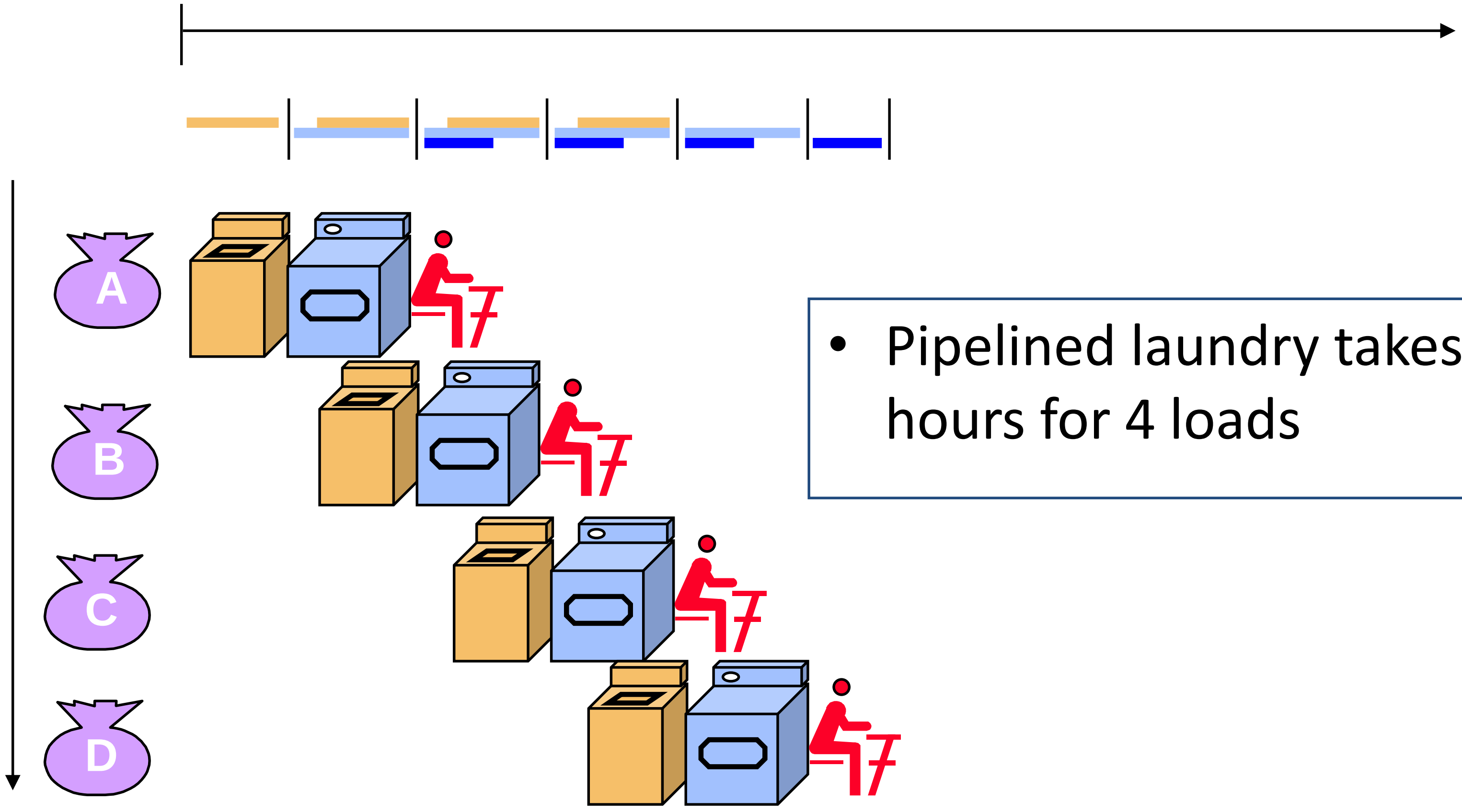


Sequential laundry takes 6 hours for 4 loads

If they learned pipelining, how long would laundry take?

What Is Pipelining

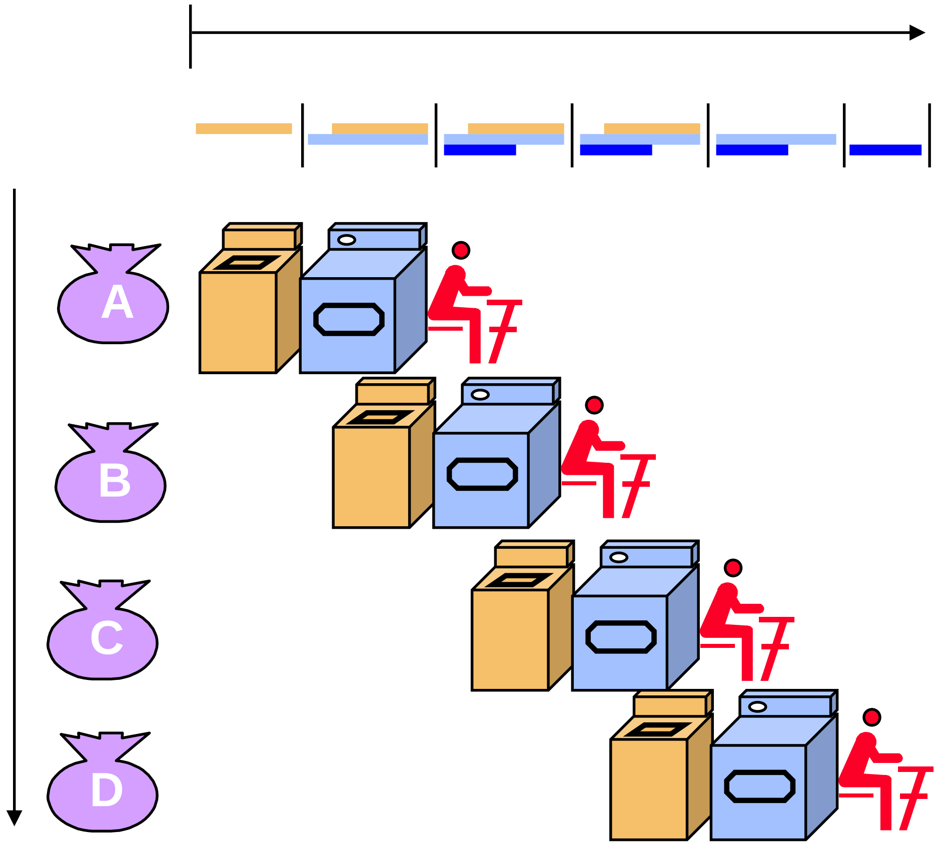
Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

What Is Pipelining

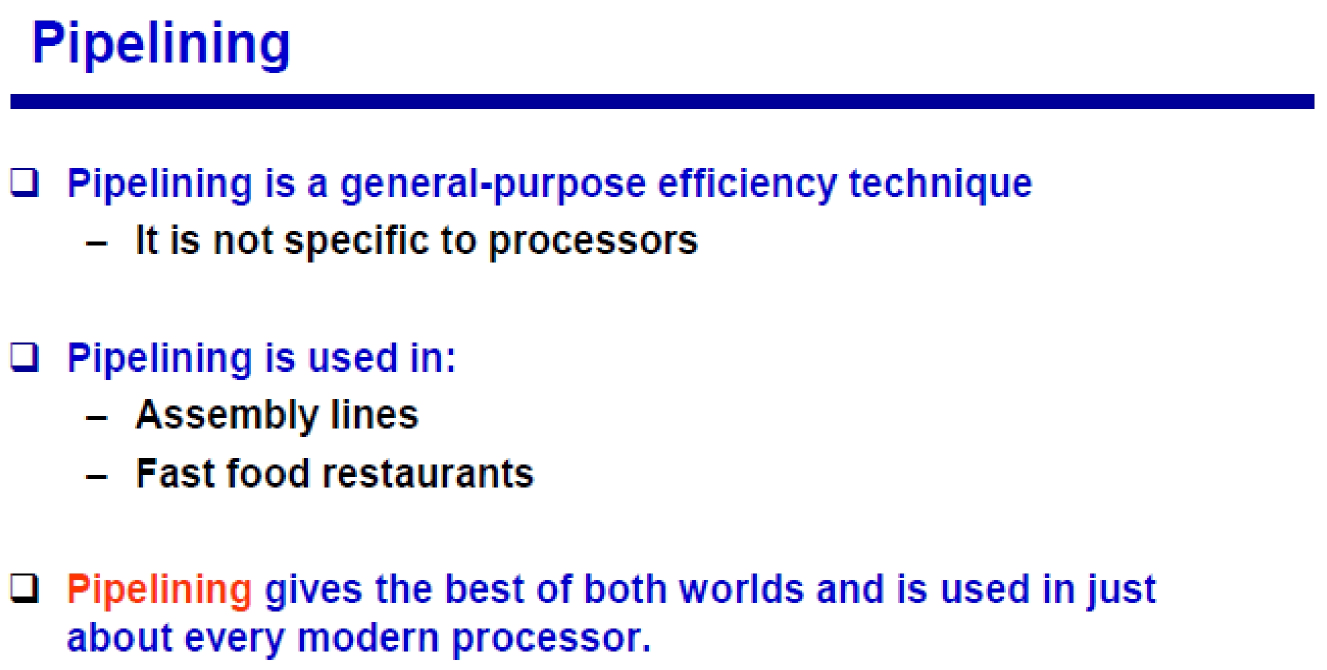
Pipelining Lessons



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

RUTGERS

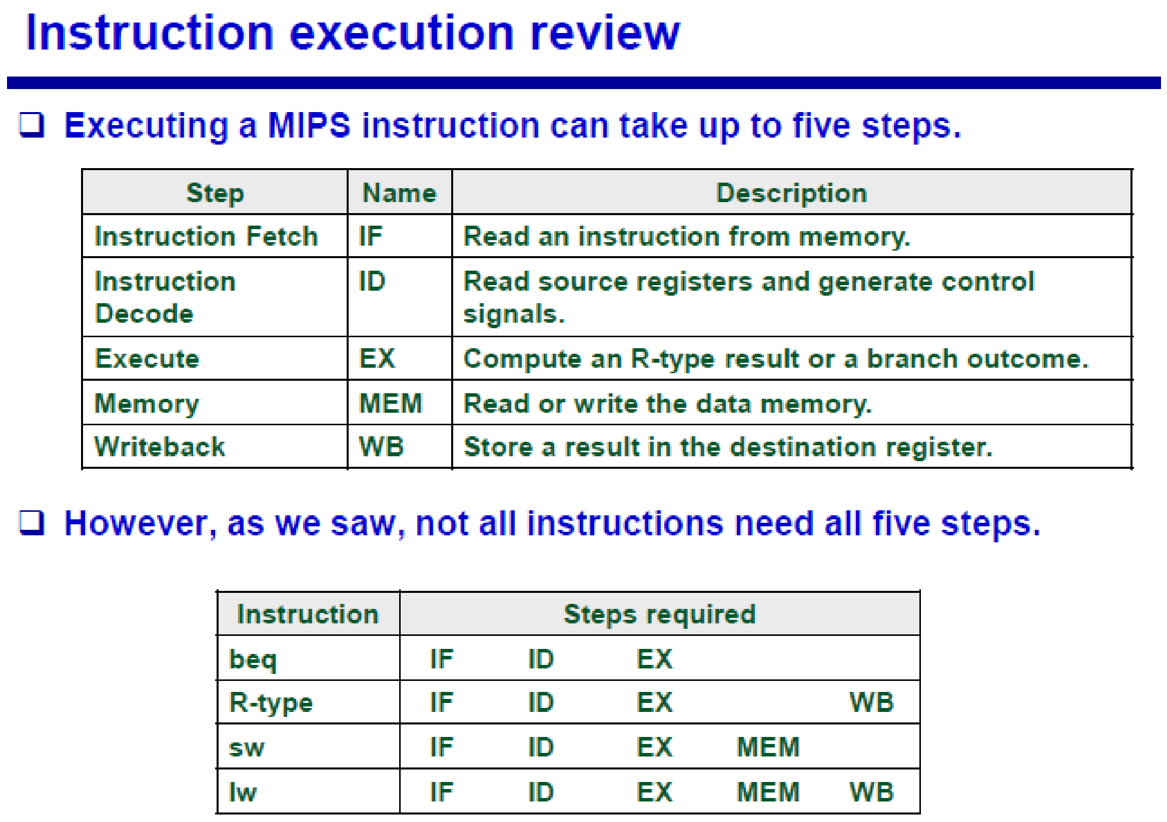
Pipelining



Pipelining

- Pipelining is a general-purpose efficiency technique
  - It is not specific to processors
- Pipelining is used in:
  - Assembly lines
  - Fast food restaurants
- Pipelining gives the best of both worlds and is used in just about every modern processor.

Pipelining



Instruction execution review

- Executing a MIPS instruction can take up to five steps.

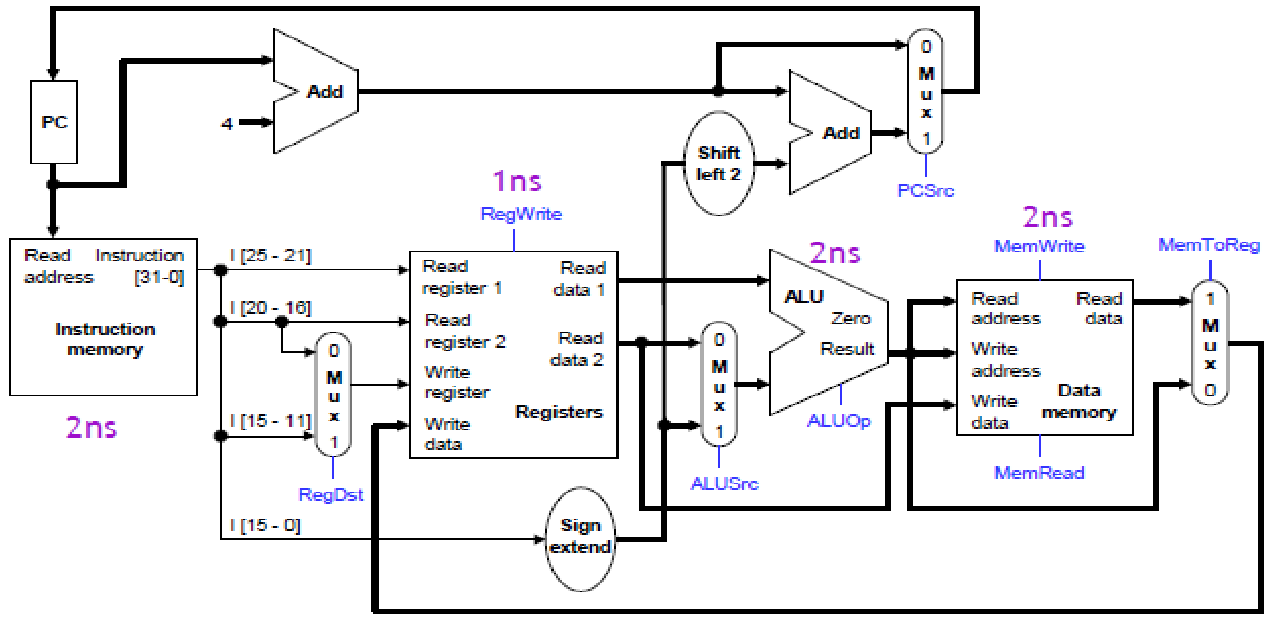
Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

- However, as we saw, not all instructions need all five steps.

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

Pipelining

Single-cycle datapath diagram



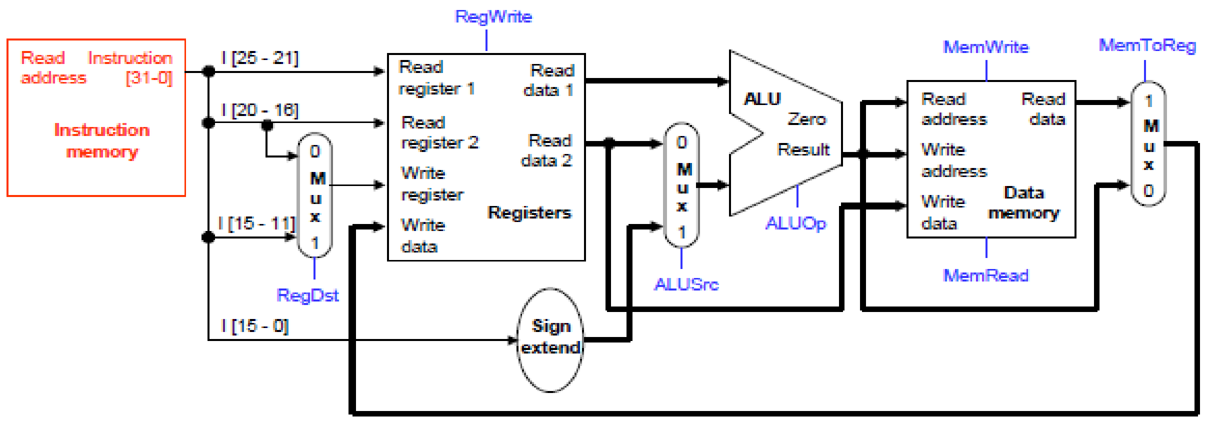
❑ How long does it take to execute each instruction?

RUTGERS

Pipelining

Review: Instruction Fetch (IF)

- Let's quickly review how lw is executed in the single-cycle datapath.
- We'll ignore PC incrementing and branching for now.
- In the Instruction Fetch (IF) step, we read the instruction memory.

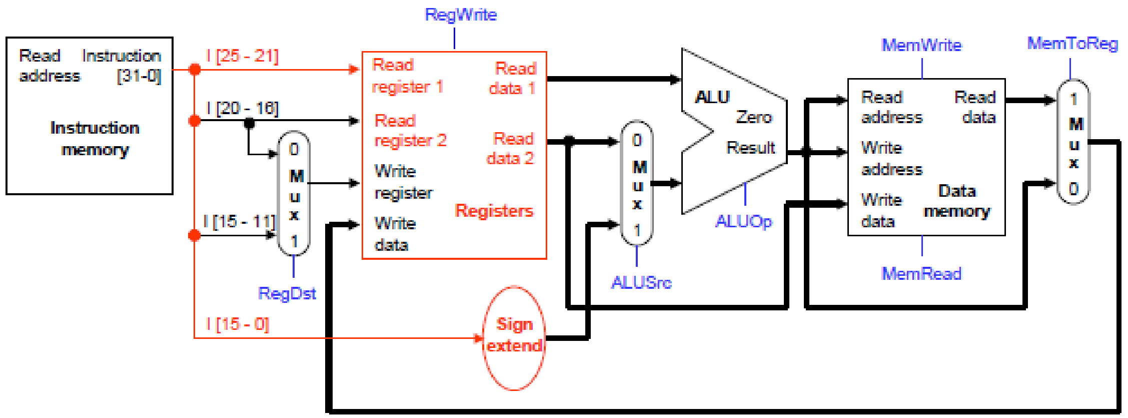


RUTGERS

Pipelining

Instruction Decode (ID)

- The Instruction Decode (ID) step reads the source register from the register file.

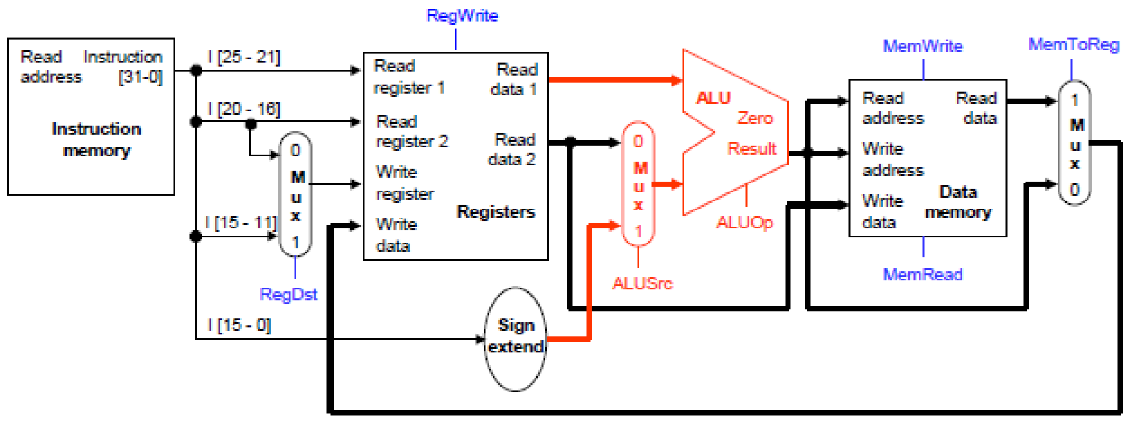


RUTGERS

Pipelining

Execute (EX)

- The third step, Execute (EX), computes the effective memory address from the source register and the instruction's constant field.

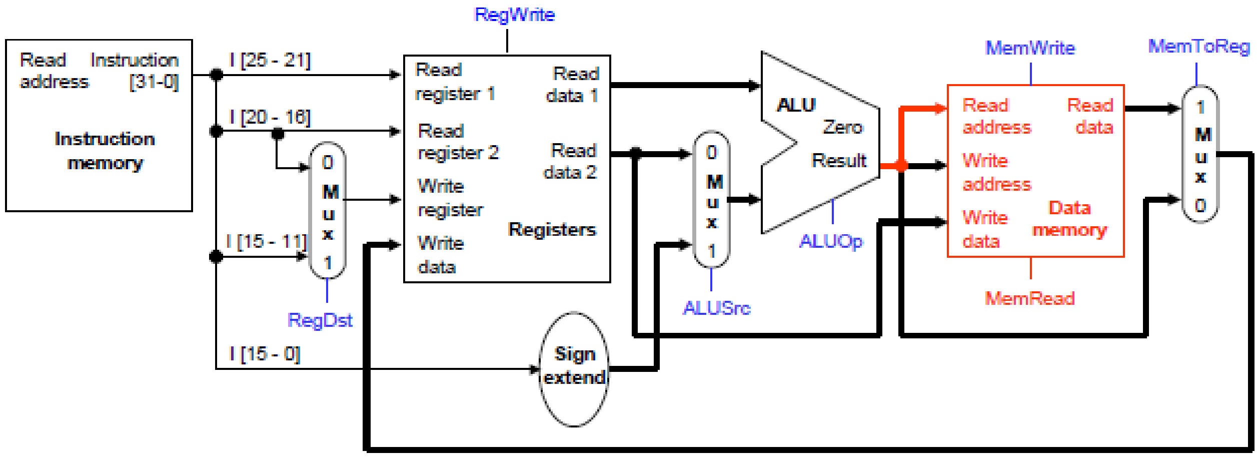


RUTGERS

Pipelining

Memory (MEM)

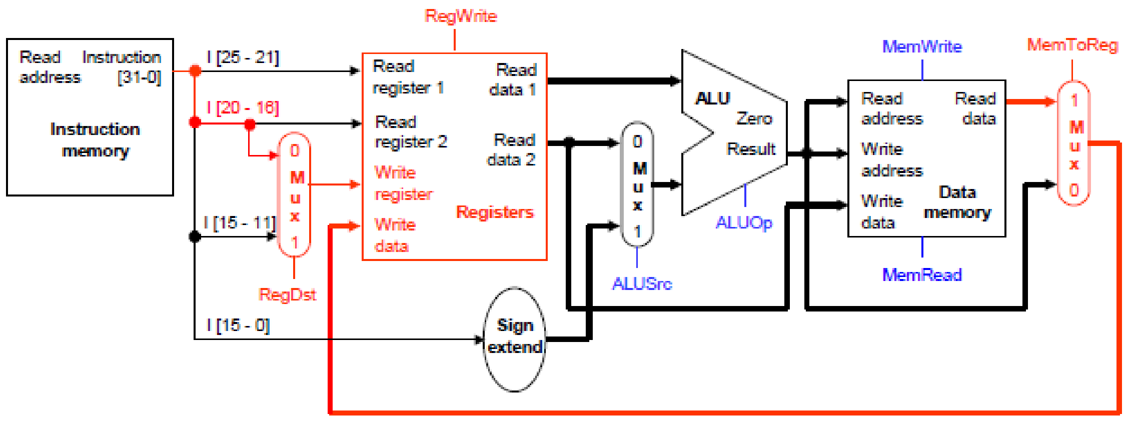
- The Memory (MEM) step involves reading the data memory, from the address computed by the ALU.



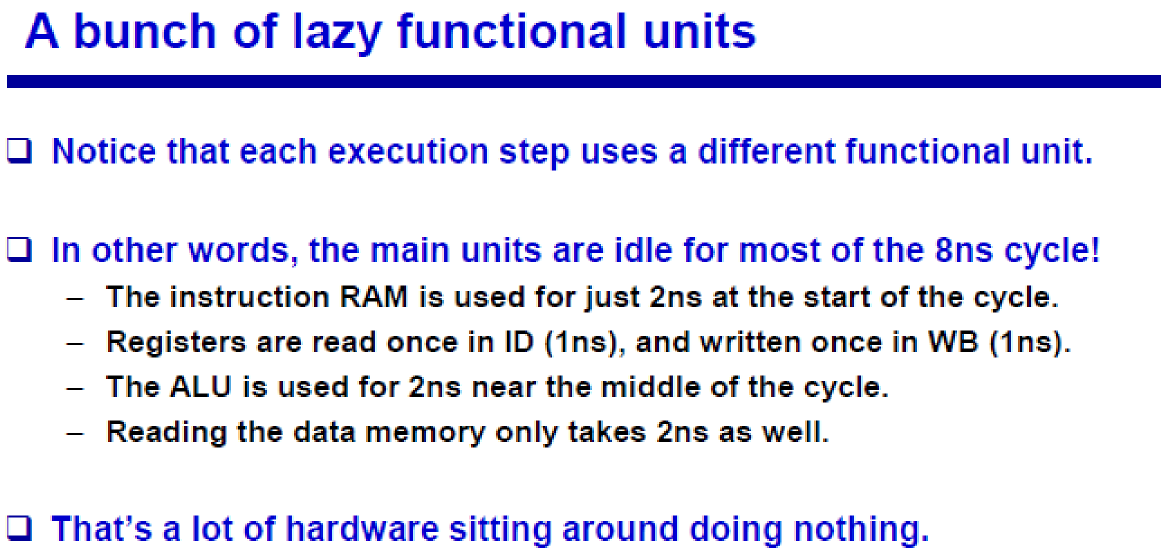
Pipelining

Writeback (WB)

- Finally, in the Writeback (WB) step, the memory value is stored into the destination register.



Pipelining



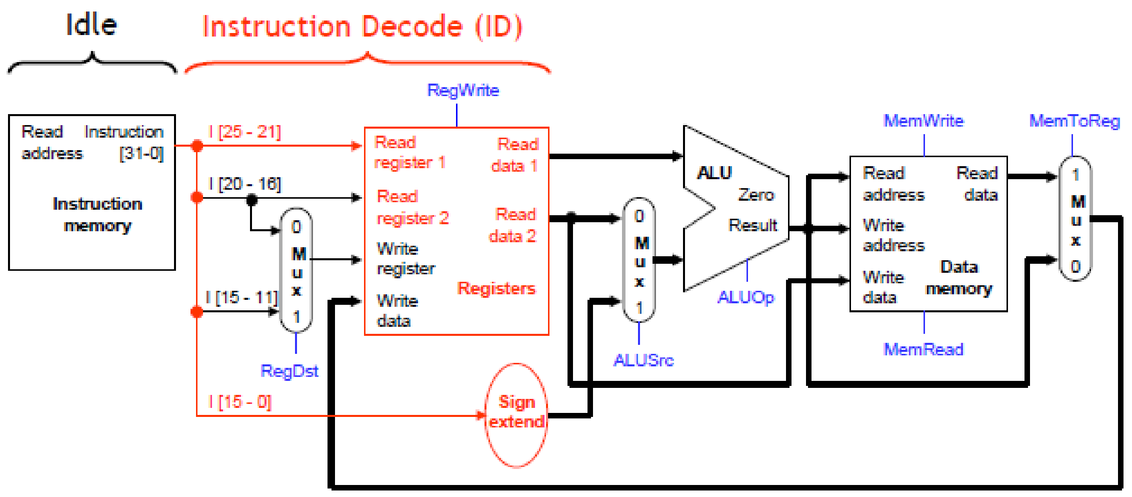
A bunch of lazy functional units

- ❑ Notice that each execution step uses a different functional unit.
- ❑ In other words, the main units are idle for most of the 8ns cycle!
  - – The instruction RAM is used for just 2ns at the start of the cycle.
  - – Registers are read once in ID (1ns), and written once in WB (1ns).
  - – The ALU is used for 2ns near the middle of the cycle.
  - – Reading the data memory only takes 2ns as well.
- ❑ That's a lot of hardware sitting around doing nothing.

Pipelining

Putting those slackers to work

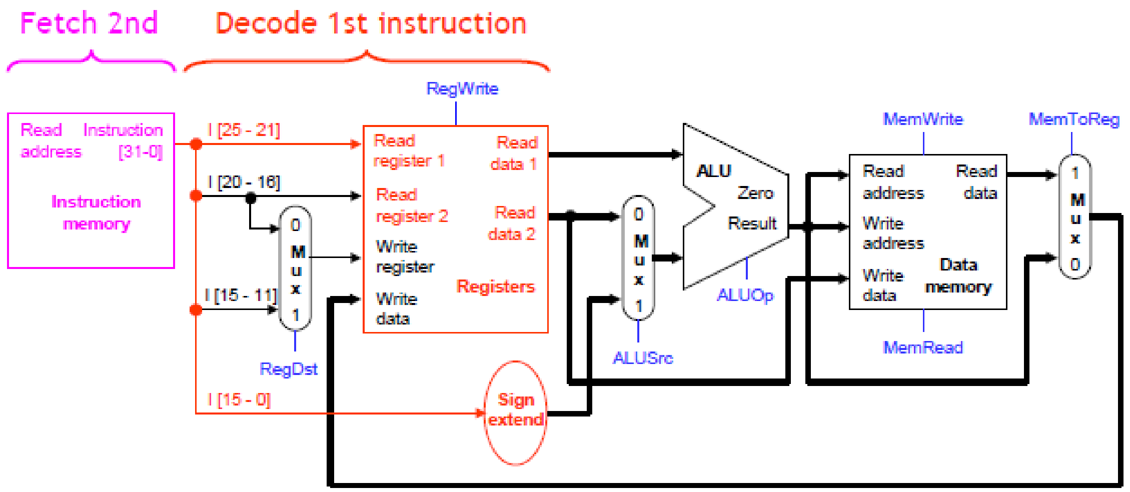
- We shouldn't have to wait for the entire instruction to complete before we can re-use the functional units.
- For example, the instruction memory is free in the Instruction Decode step as shown below, so...



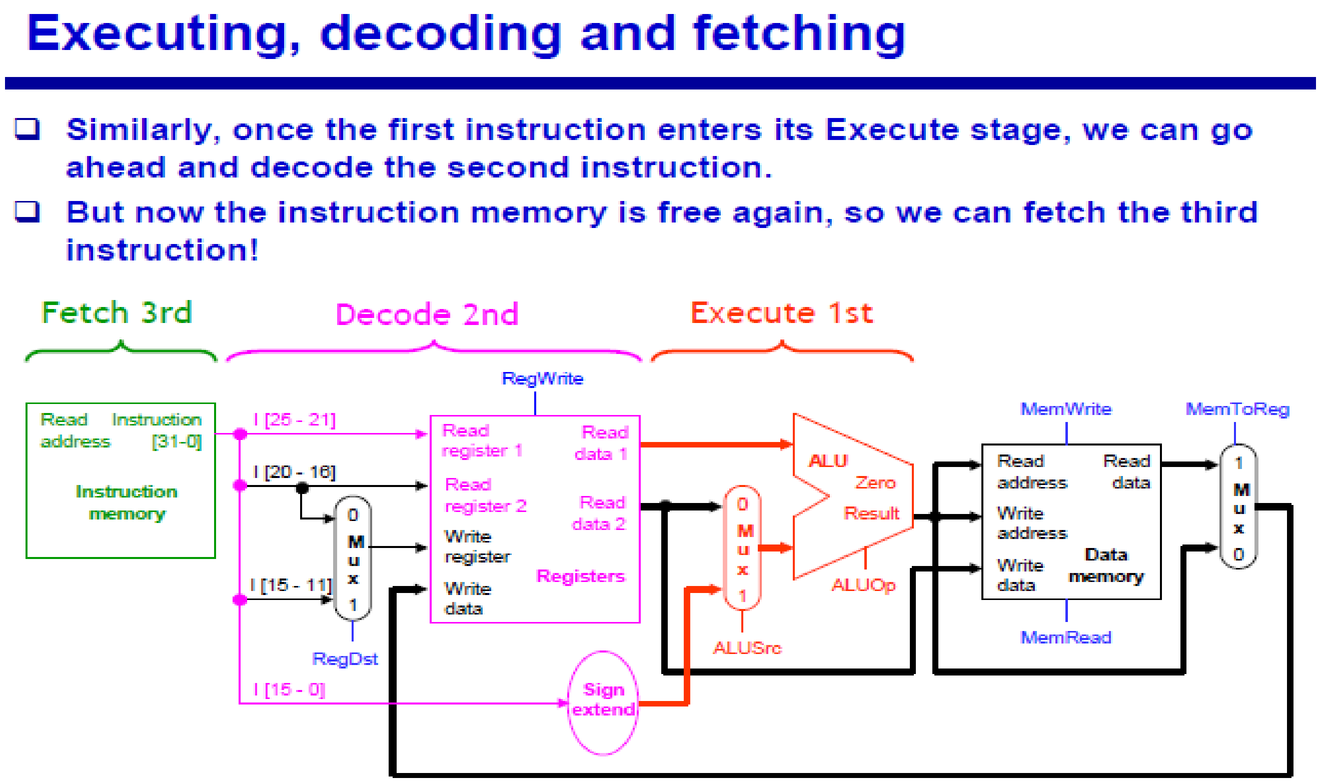
Pipelining

Decoding and fetching together

- Why don't we go ahead and fetch the *next* instruction while we're decoding the first one?



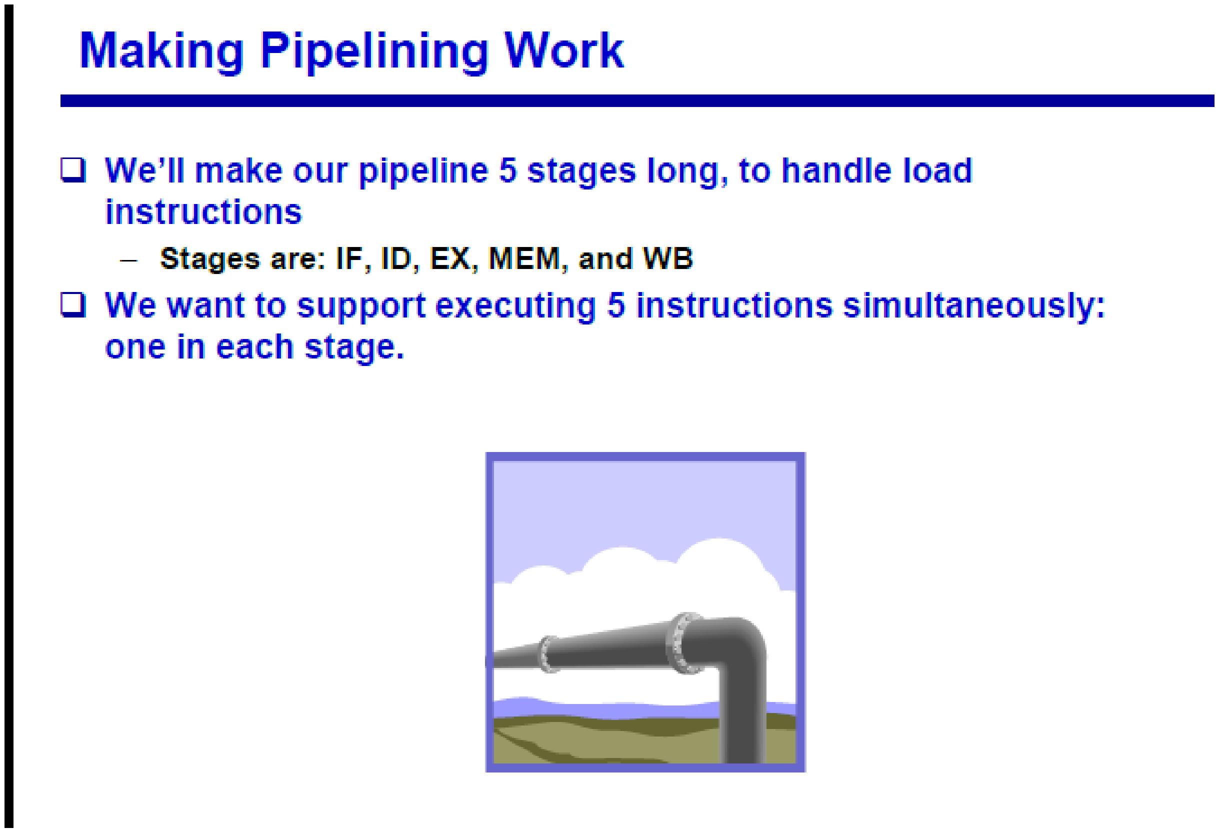
Pipelining



Executing, decoding and fetching

- ❑ Similarly, once the first instruction enters its Execute stage, we can go ahead and decode the second instruction.
- ❑ But now the instruction memory is free again, so we can fetch the third instruction!

Pipelining



Making Pipelining Work

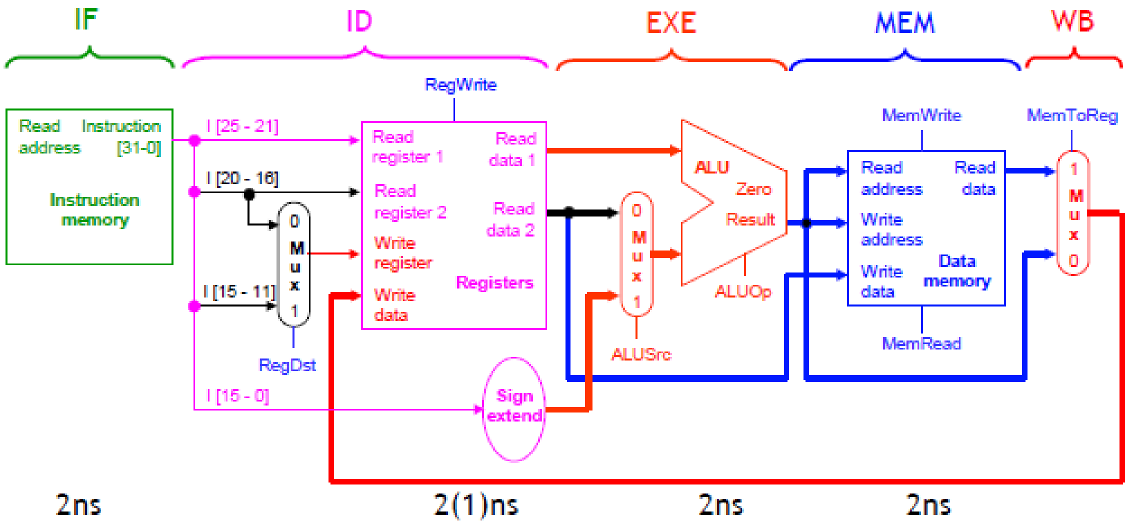
- ❑ We'll make our pipeline 5 stages long, to handle load instructions
  - Stages are: IF, ID, EX, MEM, and WB
- ❑ We want to support executing 5 instructions simultaneously: one in each stage.

RUTGERS

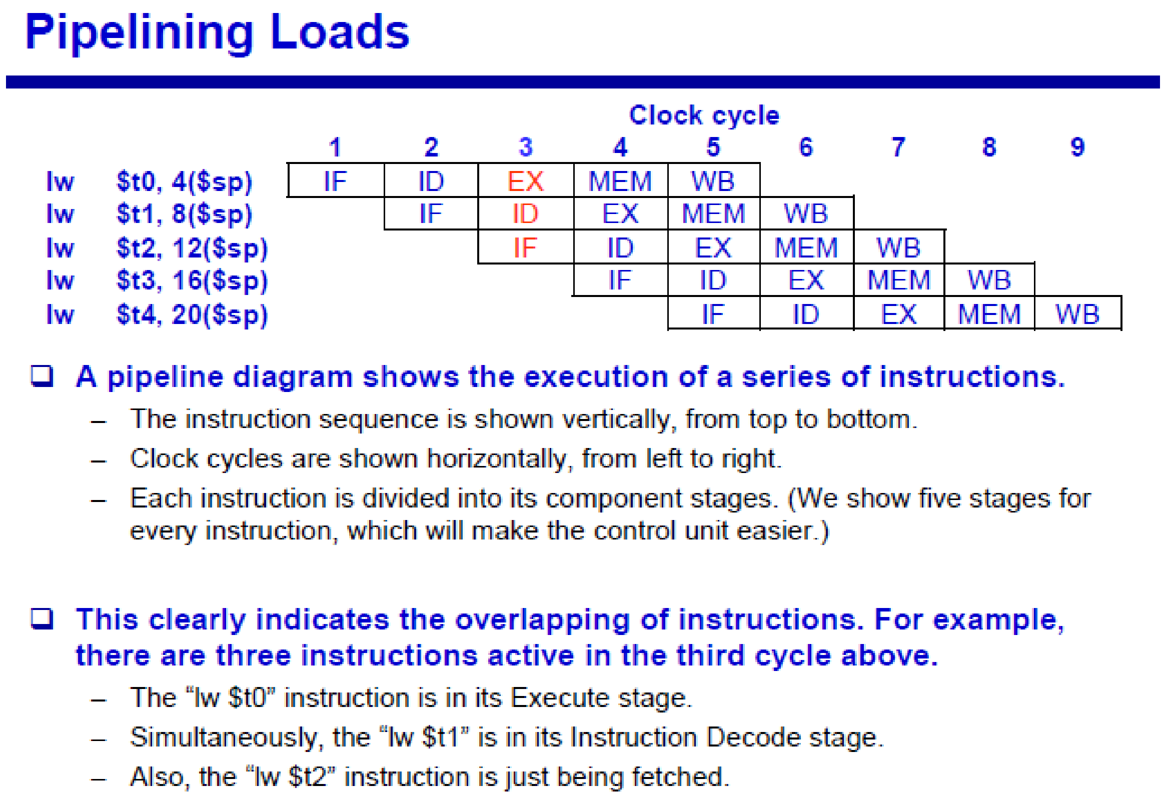
Pipelining

Break datapath into 5 stages

- Each stage has its own functional units.
- Each stage can execute in 2ns



Pipelining



Pipelining Loads

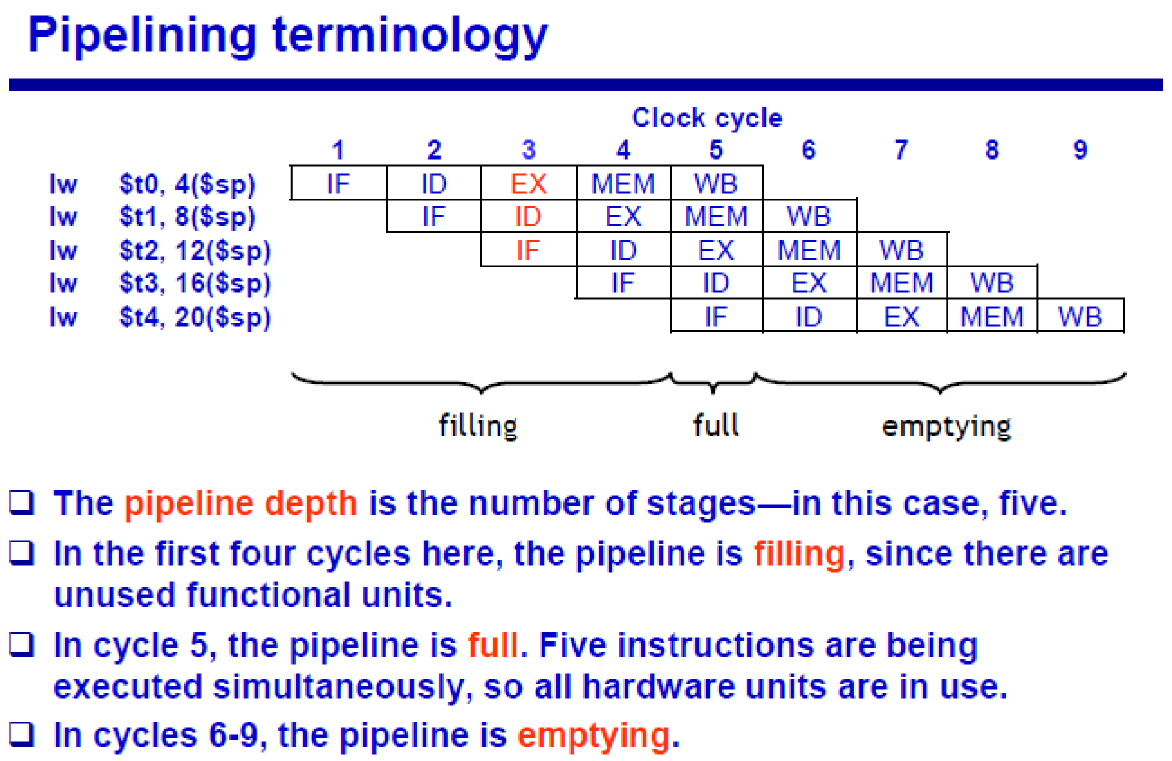
		Clock cycle								
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
lw	$t1, 8($sp)		IF	ID	EX	MEM	WB			
lw	$t2, 12($sp)			IF	ID	EX	MEM	WB		
lw	$t3, 16($sp)				IF	ID	EX	MEM	WB	
lw	$t4, 20($sp)					IF	ID	EX	MEM	WB

- **A pipeline diagram shows the execution of a series of instructions.**
  - The instruction sequence is shown vertically, from top to bottom.
  - Clock cycles are shown horizontally, from left to right.
  - Each instruction is divided into its component stages. (We show five stages for every instruction, which will make the control unit easier.)

- **This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above.**
  - The “lw $t0” instruction is in its Execute stage.
  - Simultaneously, the “lw $t1” is in its Instruction Decode stage.
  - Also, the “lw $t2” instruction is just being fetched.

RUTGERS

Pipelining



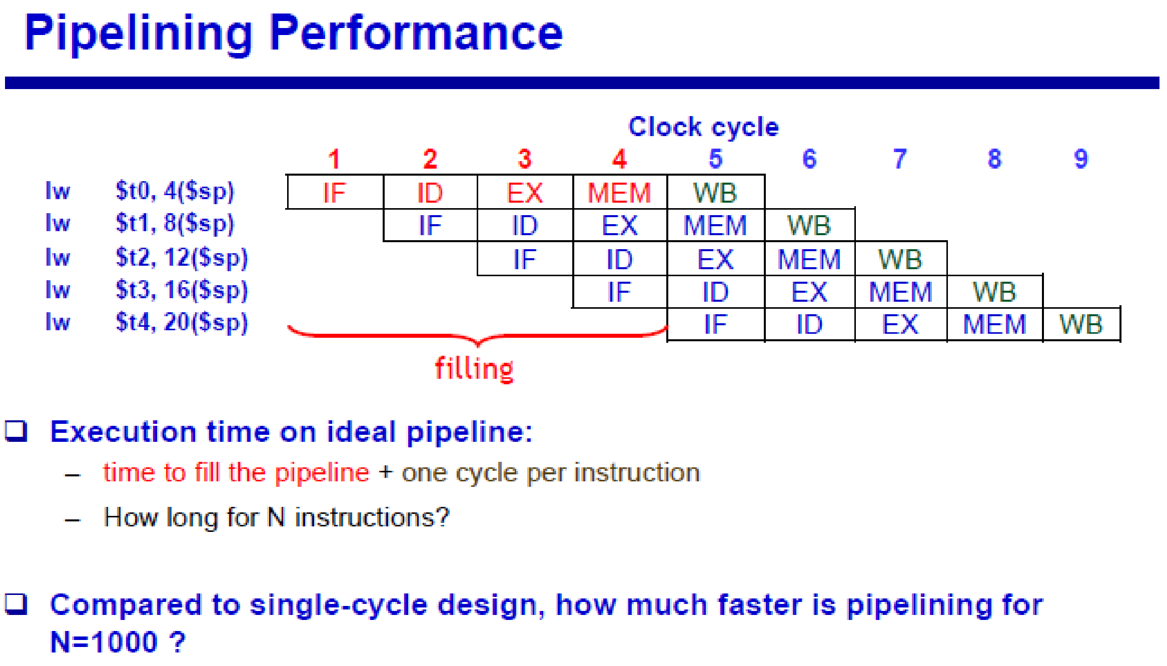
Pipelining terminology

		Clock cycle								
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
lw	$t1, 8($sp)		IF	ID	EX	MEM	WB			
lw	$t2, 12($sp)			IF	ID	EX	MEM	WB		
lw	$t3, 16($sp)				IF	ID	EX	MEM	WB	
lw	$t4, 20($sp)					IF	ID	EX	MEM	WB

filling full emptying

- The pipeline depth is the number of stages—in this case, five.
- In the first four cycles here, the pipeline is filling, since there are unused functional units.
- In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9, the pipeline is emptying.

Pipelining



Pipelining Performance

Clock cycle

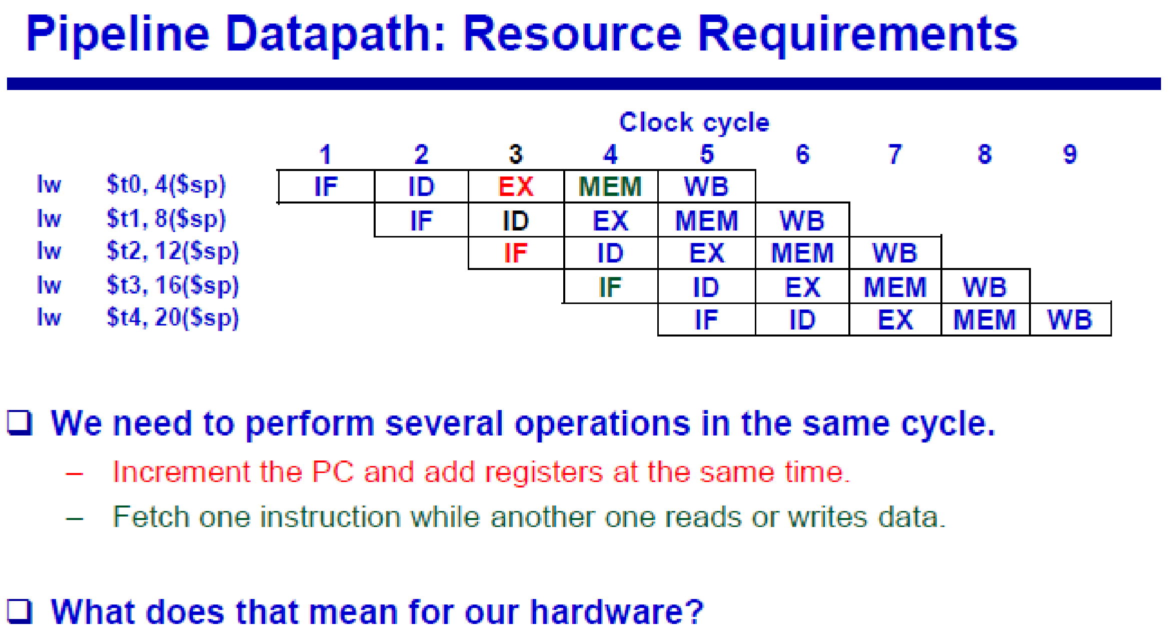
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
lw	$t1, 8($sp)		IF	ID	EX	MEM	WB			
lw	$t2, 12($sp)			IF	ID	EX	MEM	WB		
lw	$t3, 16($sp)				IF	ID	EX	MEM	WB	
lw	$t4, 20($sp)					IF	ID	EX	MEM	WB

filling

- ❑ **Execution time on ideal pipeline:**
  - time to fill the pipeline + one cycle per instruction
  - How long for N instructions?
- ❑ **Compared to single-cycle design, how much faster is pipelining for N=1000 ?**

RUTGERS

Pipelining



Pipeline Datapath: Resource Requirements

Clock cycle

		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
lw	$t1, 8($sp)		IF	ID	EX	MEM	WB			
lw	$t2, 12($sp)			IF	ID	EX	MEM	WB		
lw	$t3, 16($sp)				IF	ID	EX	MEM	WB	
lw	$t4, 20($sp)					IF	ID	EX	MEM	WB

- We need to perform several operations in the same cycle.
  - Increment the PC and add registers at the same time.
  - Fetch one instruction while another one reads or writes data.

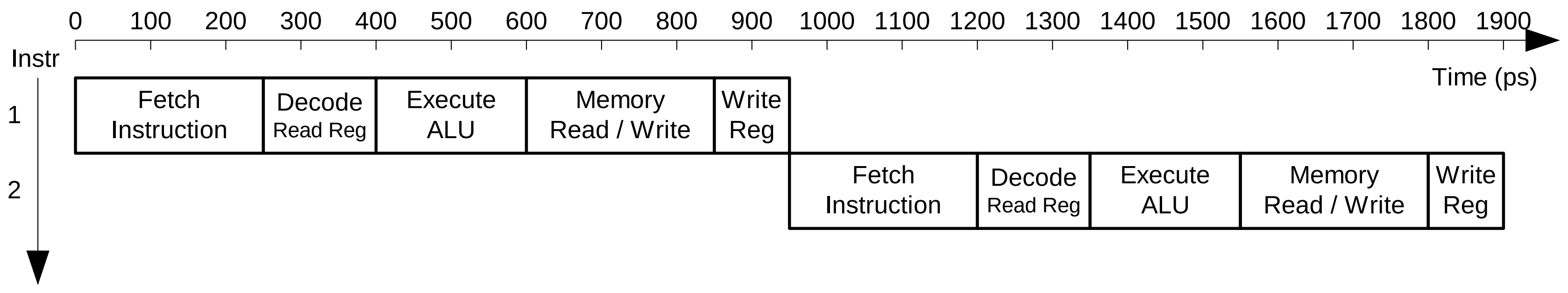
- What does that mean for our hardware?

Pipelined MIPS Processor

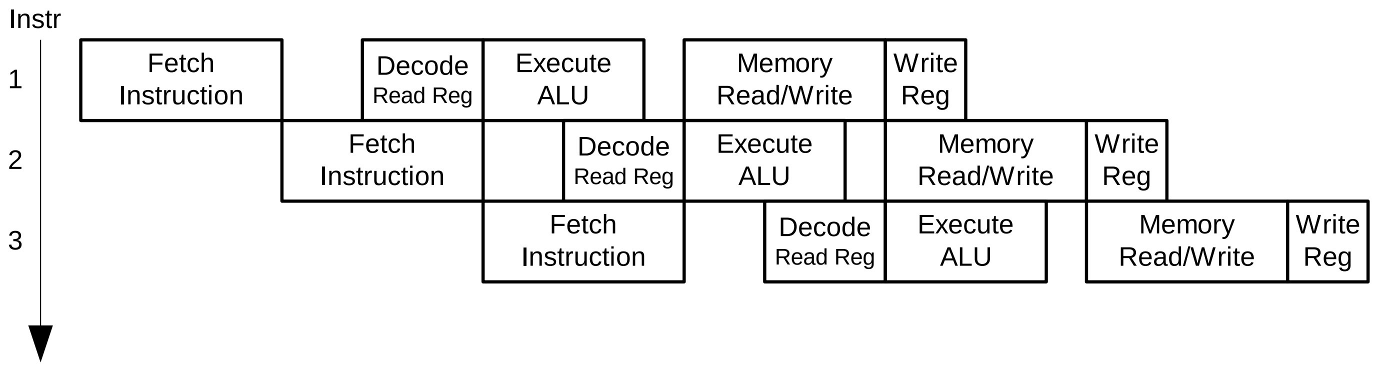
- Temporal parallelism
- Divide single-cycle processor into 5 stages:
  - Fetch
  - Decode
  - Execute
  - Memory
  - Writeback
- Add pipeline registers between stages

Single-Cycle vs. Pipelined

Single-Cycle



Pipelined



Pipeline Hurdles

Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle

- **Structural hazards:** HW cannot support this combination of instructions (single person to fold and put clothes away) – Required resource is busy
- **Data hazards:** Instruction depends on result of prior instruction still in the pipeline (missing sock)
- **Control hazards:** Pipelining of branches & other instructions that change the PC
- Common solution is to **stall** the pipeline until the hazard is resolved, inserting one or more "**bubbles**" in the pipeline

Recap: Pipeline Hazards

- Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: attempt to use the same resource two different ways at the same time
    - One memory
  - Data hazards: attempt to use data before it is ready
    - Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: attempt to make a decision before condition is evaluated
    - Branch instructions
- Pipeline implementation need to detect and resolve hazards

Pipeline Hurdles

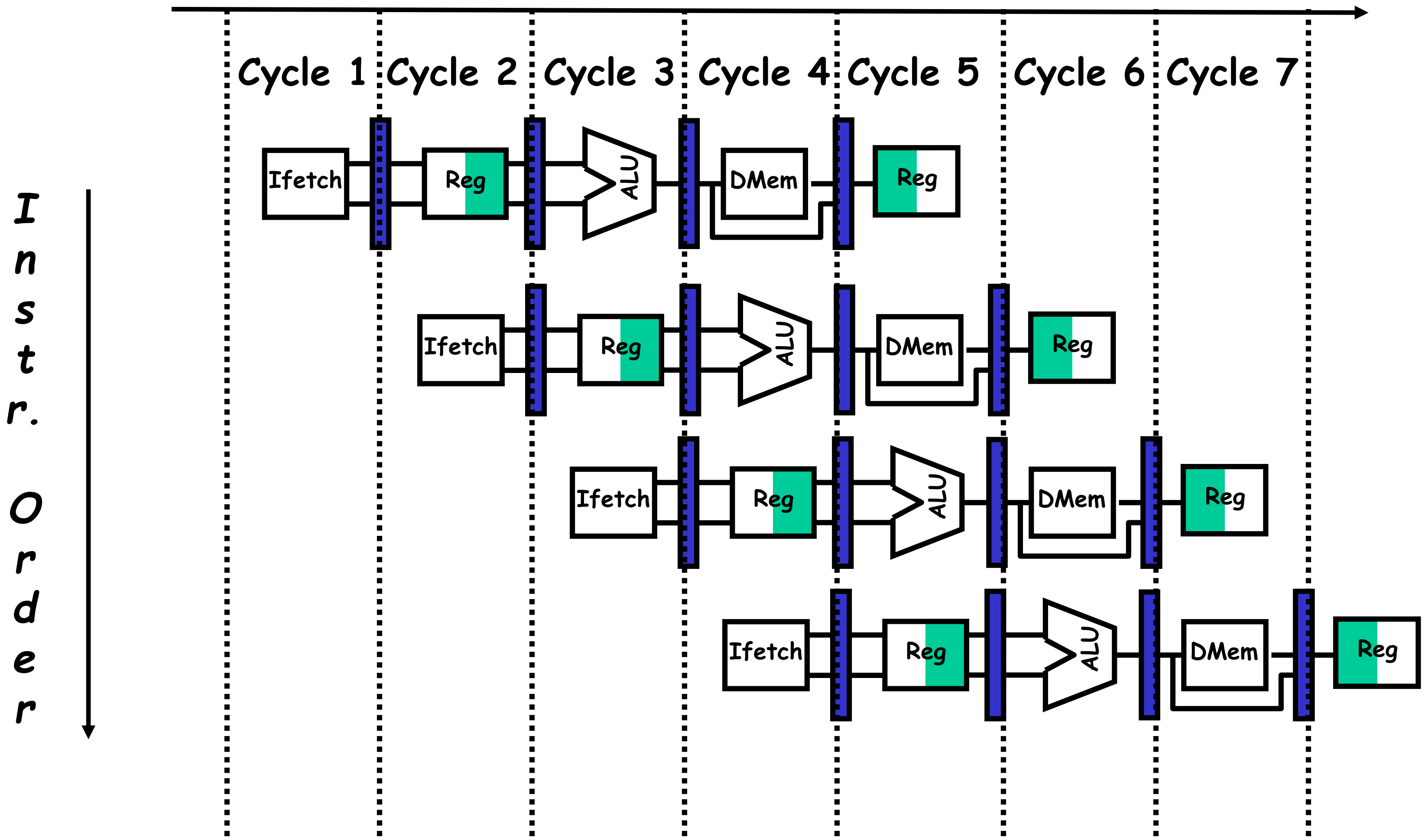
Definition

- conditions that lead to incorrect behavior if not fixed
- Structural hazard
  - **two different instructions use same h/w in same cycle**
- Data hazard
  - **two different instructions use same storage**
  - **must appear as if the instructions execute in correct order**
- Control hazard
  - **one instruction affects which instruction is next**

Resolution

- Pipeline interlock logic detects hazards and fixes them
- simple solution: stall
- increases CPI, decreases performance
- better solution: partial stall
- some instruction stall, others proceed better to stall early than late

The Basic Pipeline For MIPS



Data Hazards

These occur when at any time, there are instructions active that need to access the same data (memory or register) locations.

Where there's real trouble is when we have:

instruction A
instruction B

and B manipulates (reads or writes) data before A does. This violates the order of the instructions, since the architecture implies that A completes entirely before B is executed.

Data Hazards

Execution Order is:
Instr$_I$
Instr$_J$

Read After Write (RAW)

Instr$_J$ tries to read operand before Instr$_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

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

Data Hazards

Execution Order is:
Instr_I
Instr_J

Write After Read (WAR)

Instr_J tries to write operand *before* Instr_I reads i

– Gets wrong operand

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

– Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.

- **Can’t happen in MIPS 5 stage pipeline because:**
  – All instructions take 5 stages, and
  – Reads are always in stage 2, and
  – Writes are always in stage 5

Data Hazards

Execution Order is:
Instr_I
Instr_J

Write After Write (WAW)

Instr_J tries to write operand *before* Instr_I writes it

– Leaves wrong result (Instr_I not Instr_J)

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

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

Data Hazards

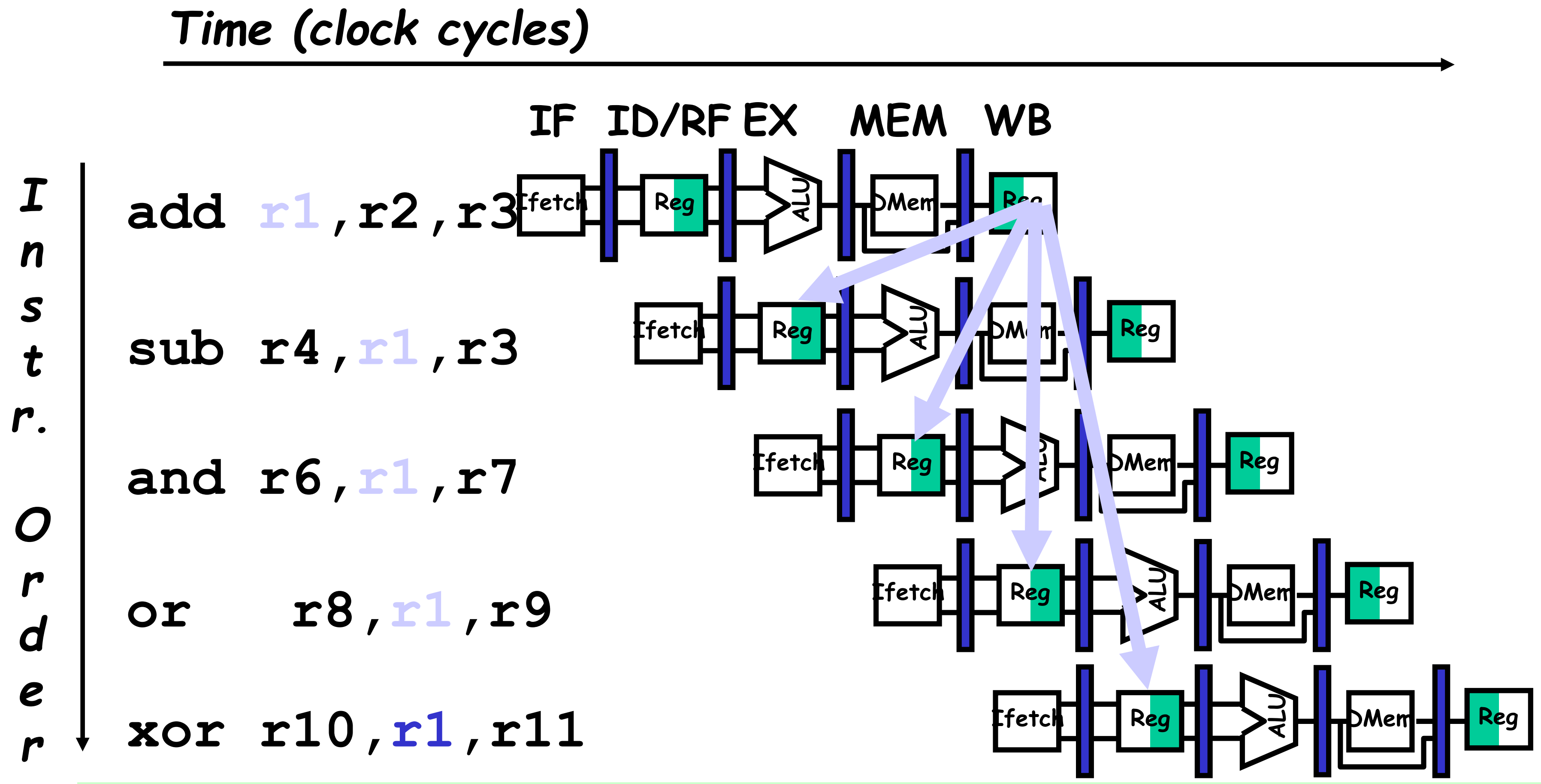
Simple Solution to RAW

- Hardware detects RAW and stalls
- Assumes register written then read each cycle
  - + low cost to implement, simple
  - -- reduces IPC
- Try to minimize stalls

Minimizing RAW stalls

- Bypass/forward/shortcircuit (We will use the word “forward”)
- Use data before it is in the register
  - + reduces/avoids stalls
  - -- complex
- Crucial for common RAW hazards

Data Hazards



The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

Data Hazards

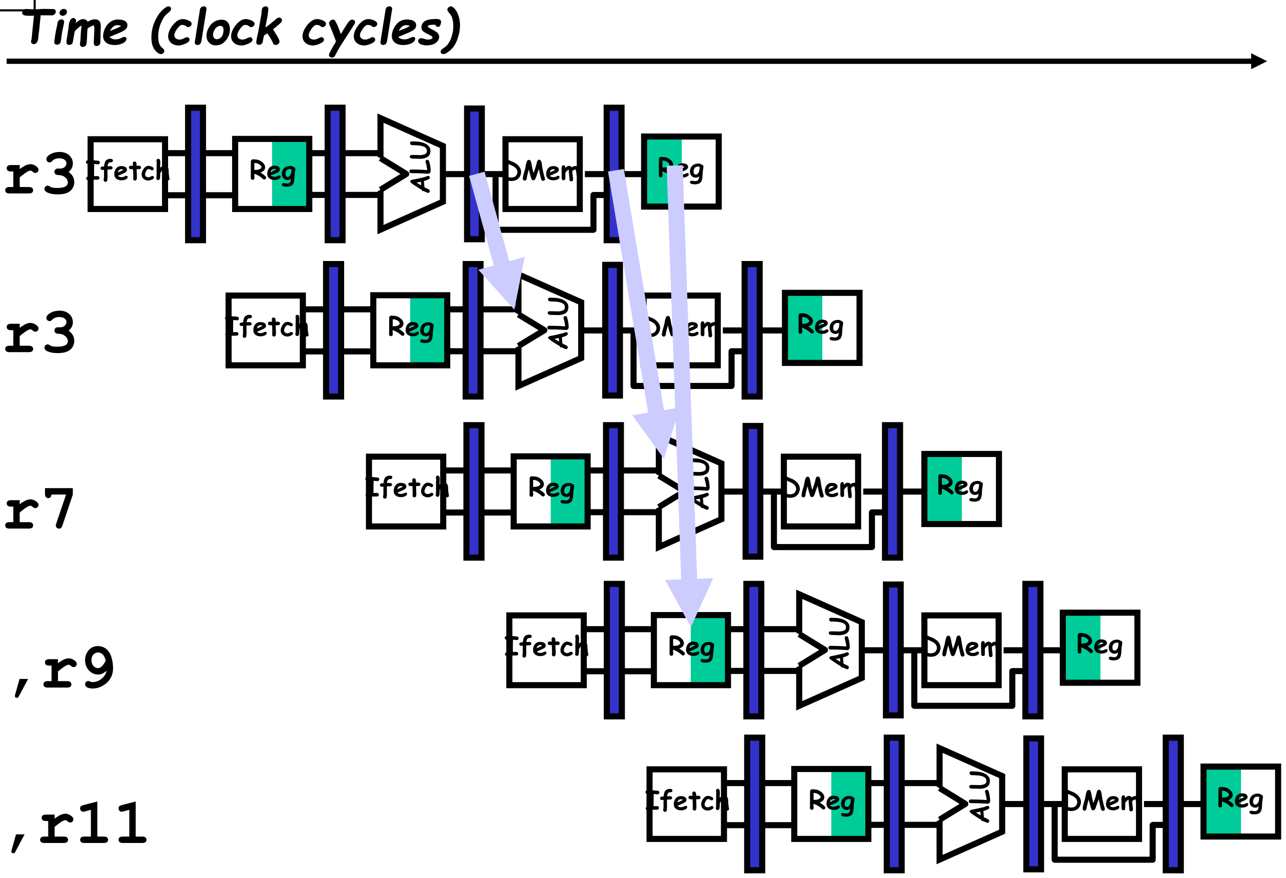
Forwarding To Avoid Data Hazard

Forwarding is the concept of making data available to the input of the ALU for subsequent instructions, even though the generating instruction hasn't gotten to WB in order to write the memory or registers.

Time (clock cycles)

Instr. Order

```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or   r8,r1,r9
xor r10,r1,r11
```



Resolving Data Hazard

- Register file design: allow a register to be read and written in the same clock cycle:
  - Always write a register in the first half of CC and read it in the second half of that CC
  - Resolve the hazard between sub and add in previous example
- Insert NOP instructions, or independent instructions by compiler
  - NOP: pipeline bubble
- Detect the hazard, then forward the proper value
  - The good way

Forwarding

- From the example,

```
sub $2, $1, $3   IF  ID  EX  MEM  WB
and $12, $2, $5      IF   ID   EX     MEM  WB
or    $13, $6, $2          IF    ID      EX     MEM  WB
```

  - And and or needs the value of $2 at EX stage
  - Valid value of $2 generated by sub at EX stage
  - We can execute and and or without stalls if the result can be forwarded to them directly

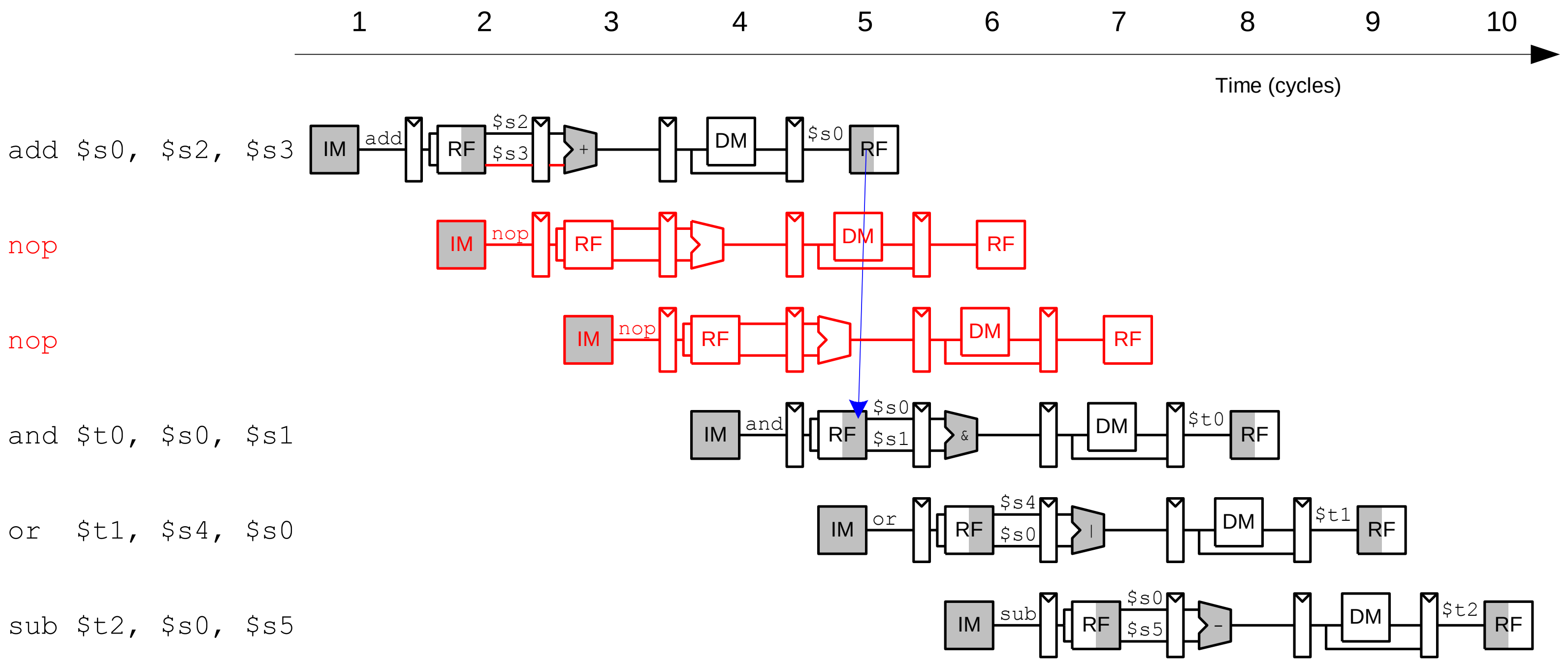
- Forwarding
  - Need to detect the hazards and determine when/to which instruciton data need to be passed

Handling Data Hazards

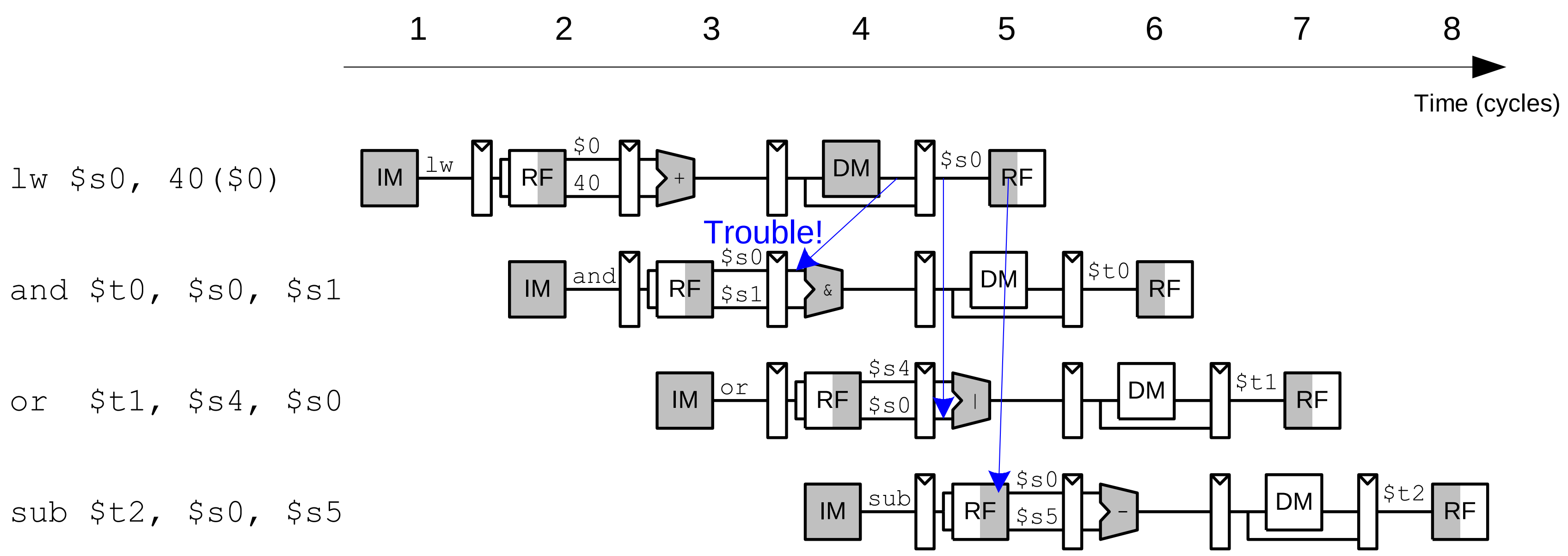
- Insert `nops` in code at compile time
- Rearrange code at compile time
- Forward data at run time
- Stall the processor at run time

Compile-Time Hazard Elimination

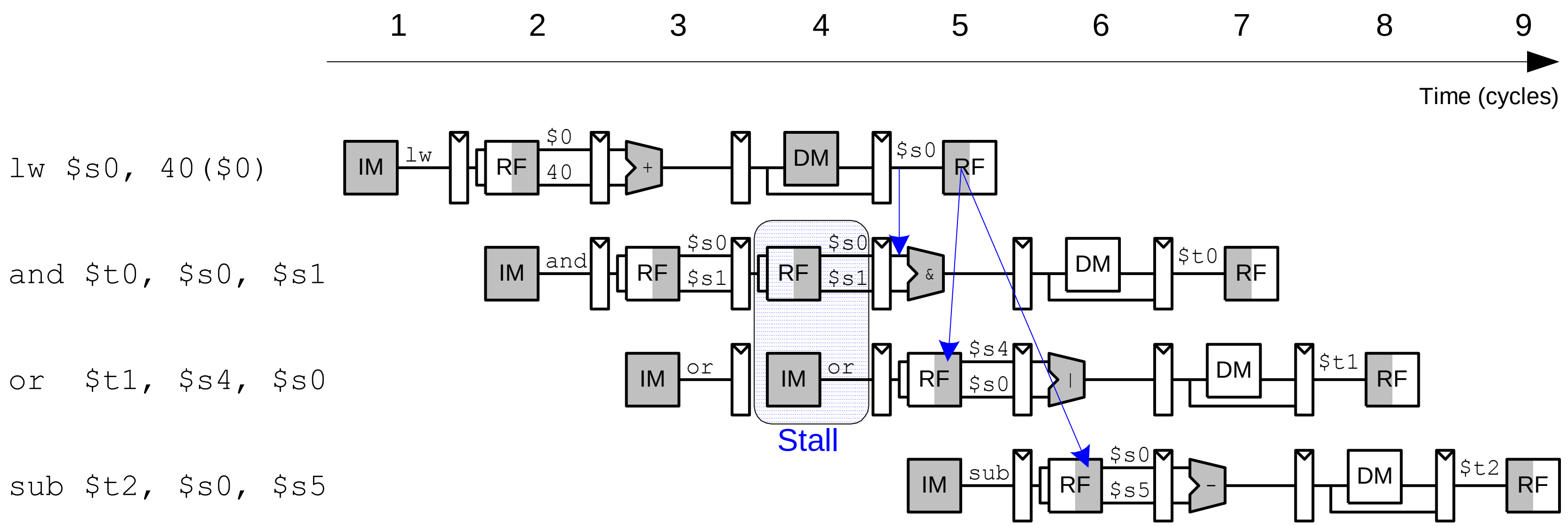
- Insert enough `nop`s for result to be ready
- Or move independent useful instructions forward



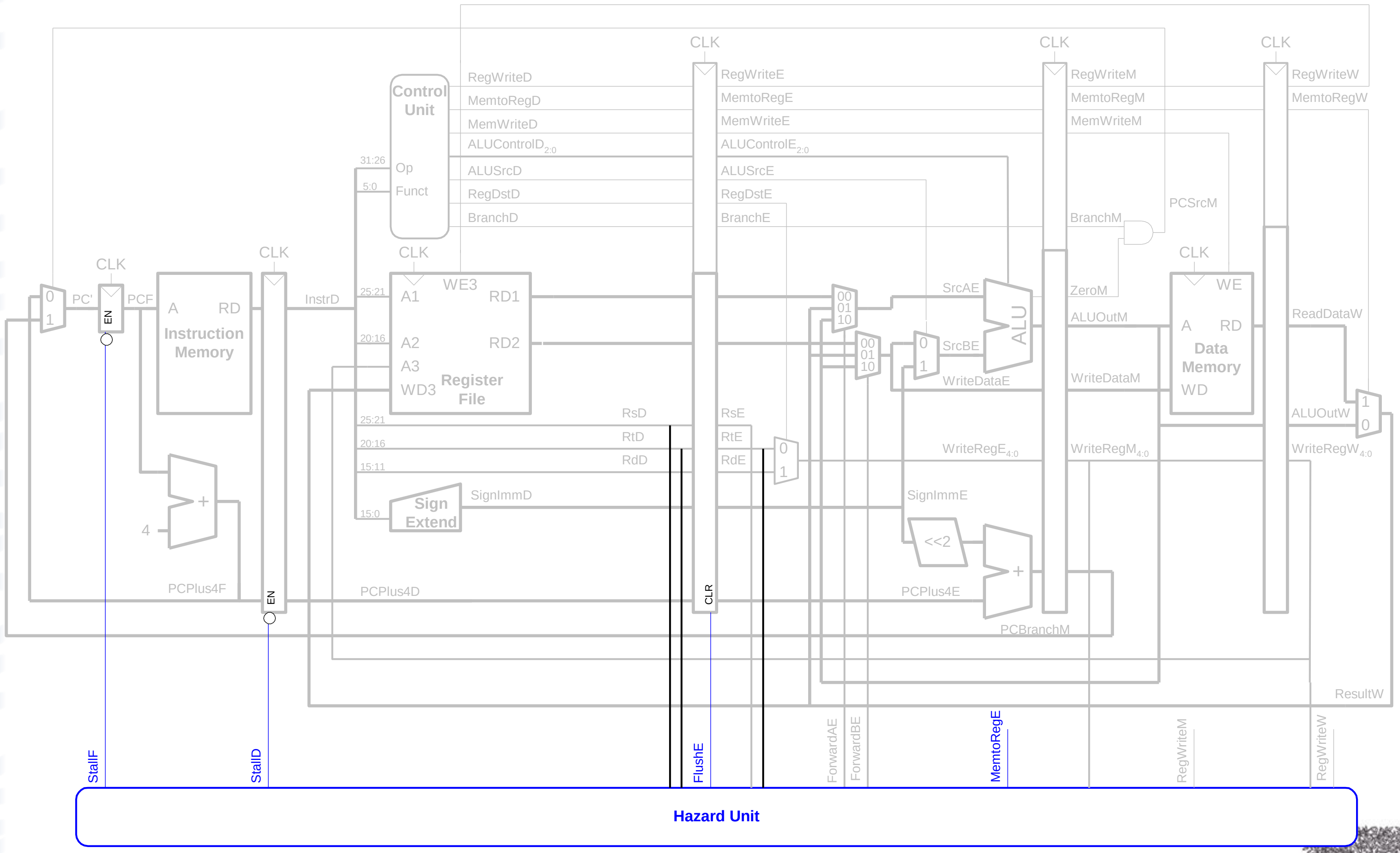
Stalling


1
2
3
4
5
6
7
8
Time (cycles)
lw $s0, 40($0)
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
IM
lw
RF
$0
40
+
DM
$s0
RF
Trouble!
and
$s0
$s1
&
$t0
or
$s4
$s0
|
$t1
sub
$s0
$s5
-
$t2

Stalling


1
2
3
4
5
6
7
8
9
Time (cycles)
lw $s0, 40($0)
and $t0, $s0, $s1
or $t1, $s4, $s0
sub $t2, $s0, $s5
IM
lw
RF
$0
40
+
DM
$s0
RF
IM
and
RF
$s0
$s1
RF
$s0
$s1
&
DM
$t0
RF
IM
or
IM
or
RF
$s4
$s0
|
DM
$t1
RF
Stall
IM
sub
RF
$s0
$s5
-
DM
$t2
RF

Stalling Hardware



Control Hazards

- **`beq`:**
  - branch not determined until 4th stage of pipeline
  - Instructions after branch fetched before branch occurs
  - These instructions must be flushed if branch happens
- **Branch misprediction penalty**
  - number of instruction flushed when branch is taken
  - May be reduced by determining branch earlier

Exceptions

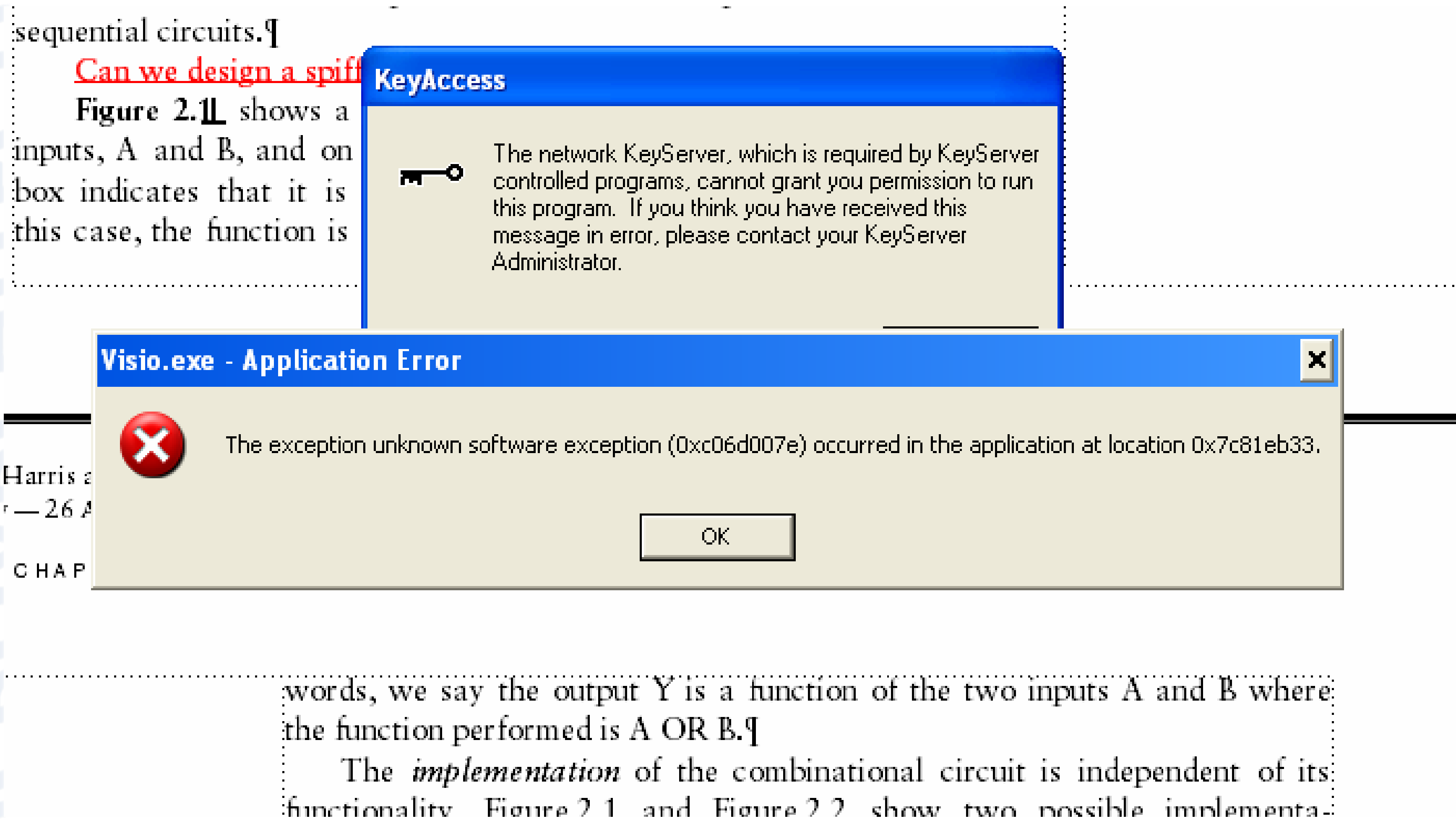
- Unexpected events
- External: interrupt
    - e.g. I/O request
- Internal: exception
    - e.g. Overflow, Undefined instruction opcode, Software trap, Page fault

- How to handle exception?
    - Jump to general entry point (record exception type in status register)
    - Jump to vectored entry point
    - Address of faulting instruction has to be recorded !

Review: Exceptions

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

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

KeyAccess

The network KeyServer, which is required by KeyServer controlled programs, cannot grant you permission to run this program. If you think you have received this message in error, please contact your KeyServer Administrator.

Visio.exe - Application Error

The exception unknown software exception (0xc06d007e) occurred in the application at location 0x7c81eb33.

OK

Harris

— 26

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-

Exception Registers

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

mfc0

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

MICROARCHITECTURE

Exception Causes

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

Extend multicycle MIPS processor to handle last two types of exceptions

ELSEVIER

Advanced Microarchitecture

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

Deep Pipelining

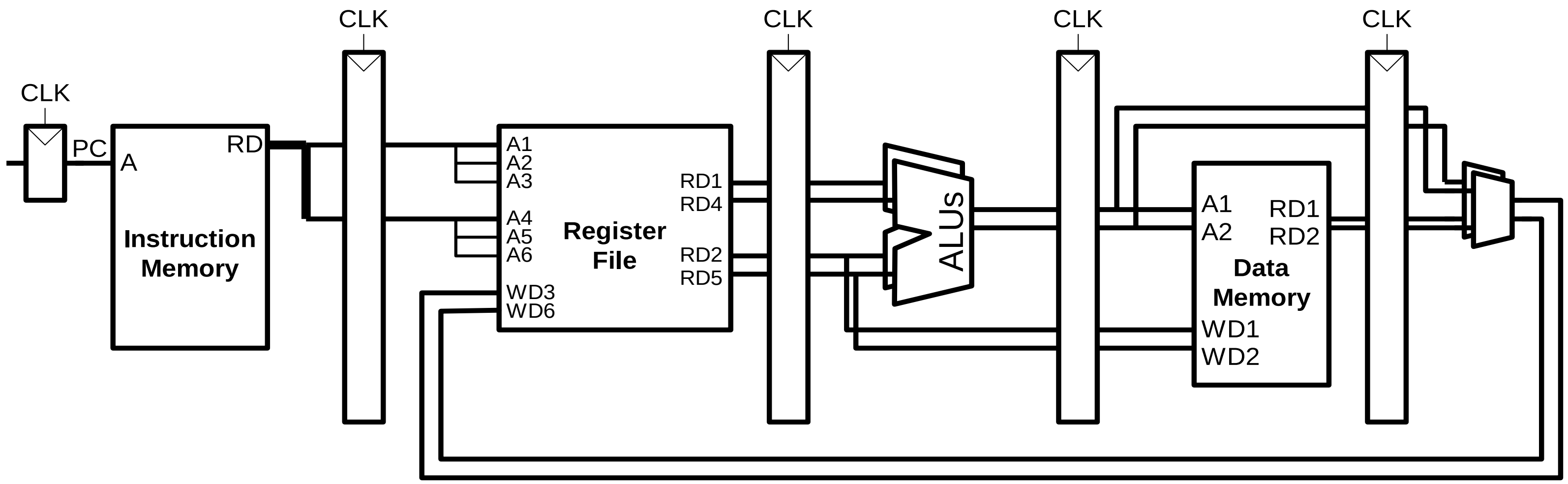
- 10-20 stages typical
- Number of stages limited by:
  - Pipeline hazards
  - Sequencing overhead
  - Power
  - Cost

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
  - Else, predict not taken
- **Dynamic branch prediction:**
  - Keep history of last (several hundred) branches in *branch target buffer*, record:
    - Branch destination
    - Whether branch was taken

Superscalar

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



Out of Order Processor

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

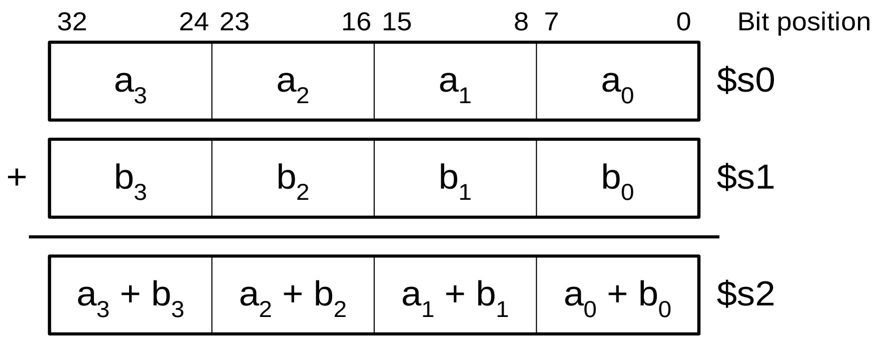
Out of Order Processor

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

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

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



Advanced Architecture Techniques

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

Threading: 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

Threads in Conventional Processor

- One thread runs at once
- When one thread stalls (for example, waiting for memory):
  - Architectural state of that thread stored
  - Architectural state of waiting thread loaded into processor and it runs
  - Called **context switching**
- 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
  - 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 increases throughput

Intel calls this "hyperthreading"

Multiprocessors

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

Backup –
Additional Material in separate slides