EECS 151/251A
Fall 2017
Digital Design and Integrated Circuits

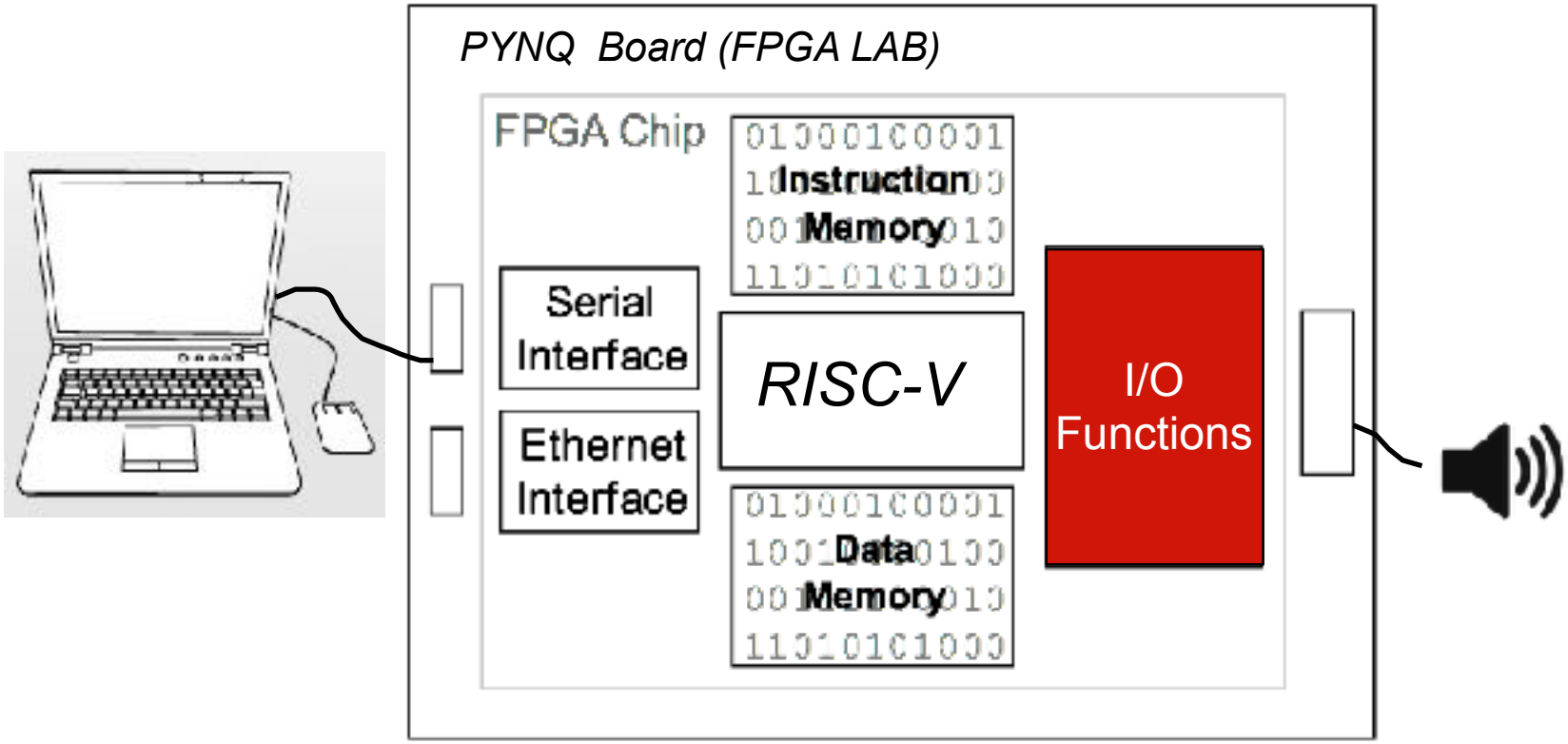
Instructor:
John Wawrzynek and Nicholas Weaver

Lecture 13

Project Introduction

- You will design and optimize a RISC-V processor
- Phase 1: Design a processor that is functional and demonstrate
- Phase 2:
  - ASIC Lab – Improve performance using accelerator
  - FPGA Lab – Add streaming input-output accerelation

EECS151/251A Project



- Executes most commonly used RISC-V instructions (http://riscv.org).
- Lightly Pipelined (high performance) implementation.
- FPGA Focus: add I/O functions and optimize.
- ASIC Focus: Adding accelerators.

Outline

- Processor Introduction
- MIPS CPU implementation
- Pipelining for Performance
- 3-Stage Pipeline

Check Harris & Harris – Chapter 6

Processor Introduction

The Purpose of This Lecture...

- To Speed Run 1.5 weeks of 61C in preparation for the project
  - Since the project is you need to build a RISC-V processor
- To include the differences between RISC-V and MIPS

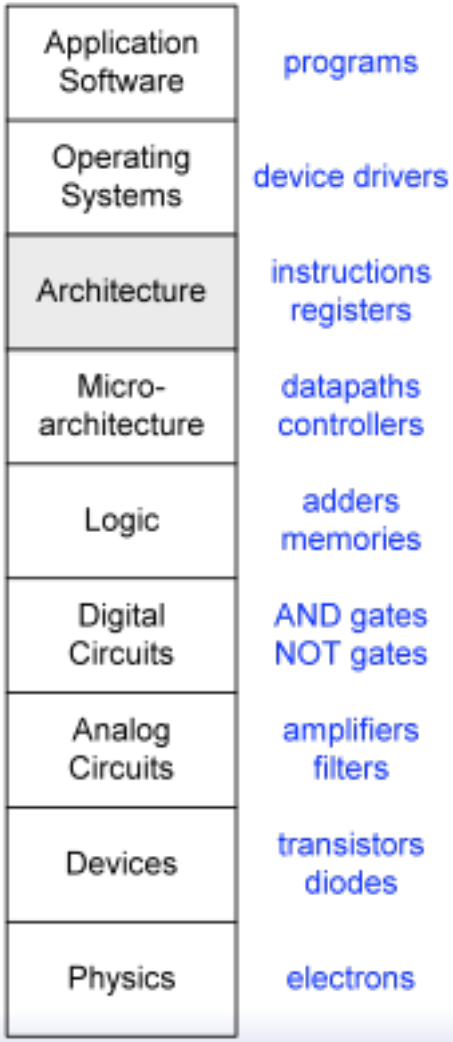
RISC-V vs MIPS

- All RISC processors are effectively the same except for one or two design decisions that "seemed like a good idea at the time"
- MIPS 'seems like a good idea':
  - The branch delay slot: Always execute the instruction after a branch or jump whether or not the branch or jump is taken
- RISC-V...
  - Nothing yet, but the immediate encoding can be hard to explain to people
- Lecture are MIPS (match book), project is RISC-V

Real Differences

- Different register naming conventions & calling conventions
  - $4 vs x4, $s0 vs s0, etc...
- Instruction encodings and encoding formats
  - 3 encodings vs 6
  - all-0 is a noop in MIPS but invalid in RISC-V
  - all RISC-V immediates are sign extended
  - RISC-V doesn't support "trap on overflow" signed math
- Instruction alignment
  - RISC-V only requires 2-byte alignment for instructions when including an optional 16b instruction encoding
    - RISC-V also supports some 48b and 64b instructions in extensions
- Instructions
  - RISC-V has dedicated "compare 2 registers & branch" operation
  - RISC-V doesn't have j or jr, just jal and jalr:
    Write to x0 to eliminate the side effect

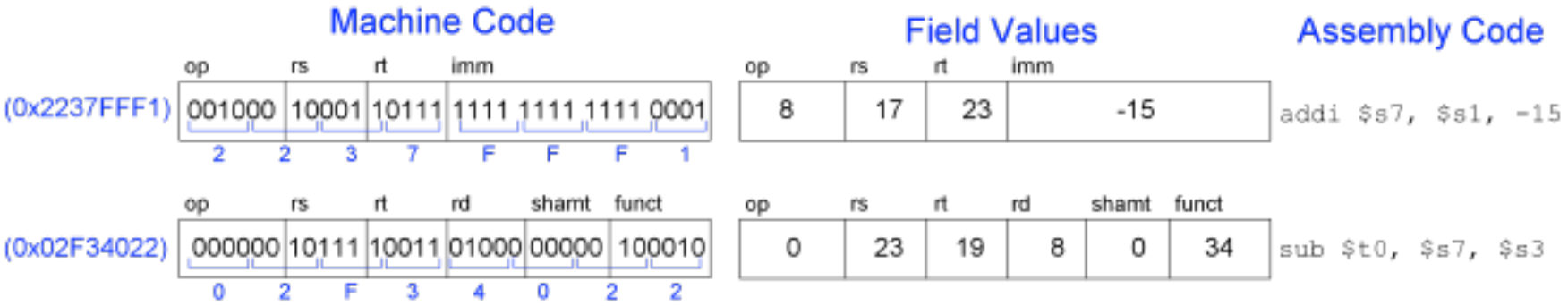
Abstraction Layers



- **Architecture**: the programmer's view of the computer
  - Defined by instructions (operations) and operand locations
- **Microarchitecture**: how to implement an architecture in hardware (covered in great detail later)
- The microarchitecture is built out of "logic" circuits and memory elements (this semester).
- All logic circuits and memory elements are implemented in the physical world with transistors.
- This semester we will implement our projects using circuits on FPGAs (field programmable gate arrays) or standard-cell ASIC design.

Interpreting Machine Code

- Start with opcode
- Opcode tells how to parse the remaining bits
- If opcode is all 0's
  - R-type instruction
  - Function bits tell what instruction it is
- Otherwise
  - opcode tells what instruction it is



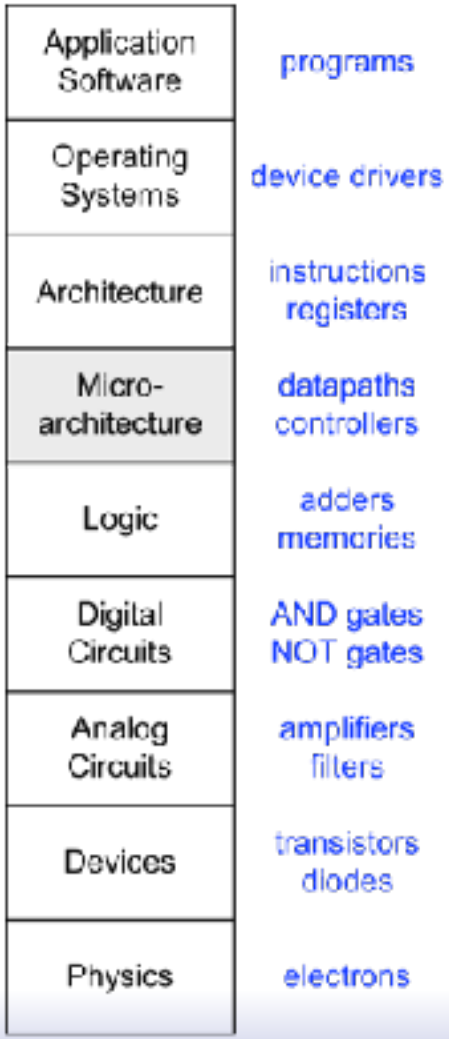
A processor is a machine code interpreter build in hardware!

Processor Microarchitecture Introduction

Microarchitecture: how to implement an architecture in hardware

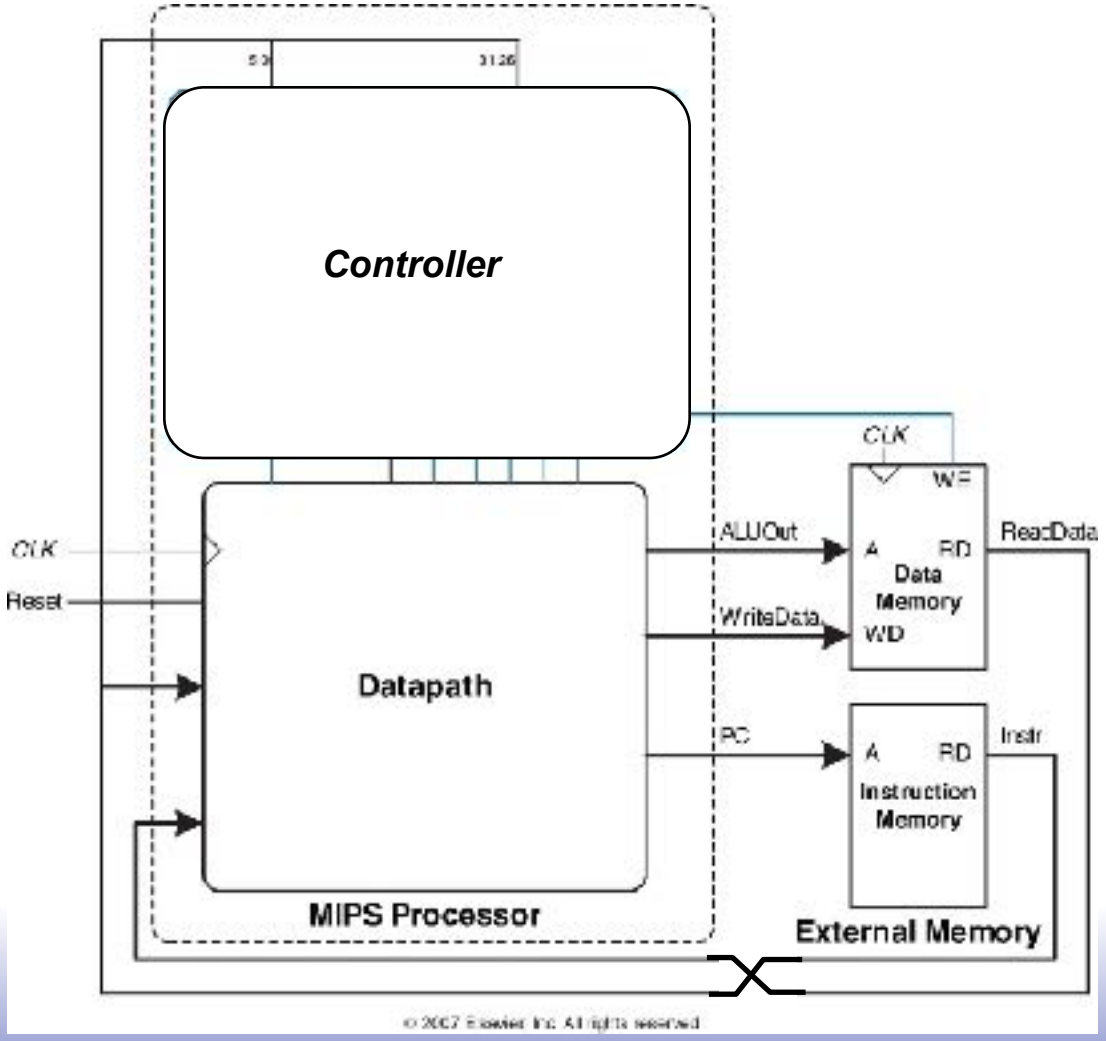
Good examples of how to put principles of digital design to practice.

Introduction to eecs151/251a final project.



MIPS Microarchitecture Oganization

Datapath + Controller + External Memory

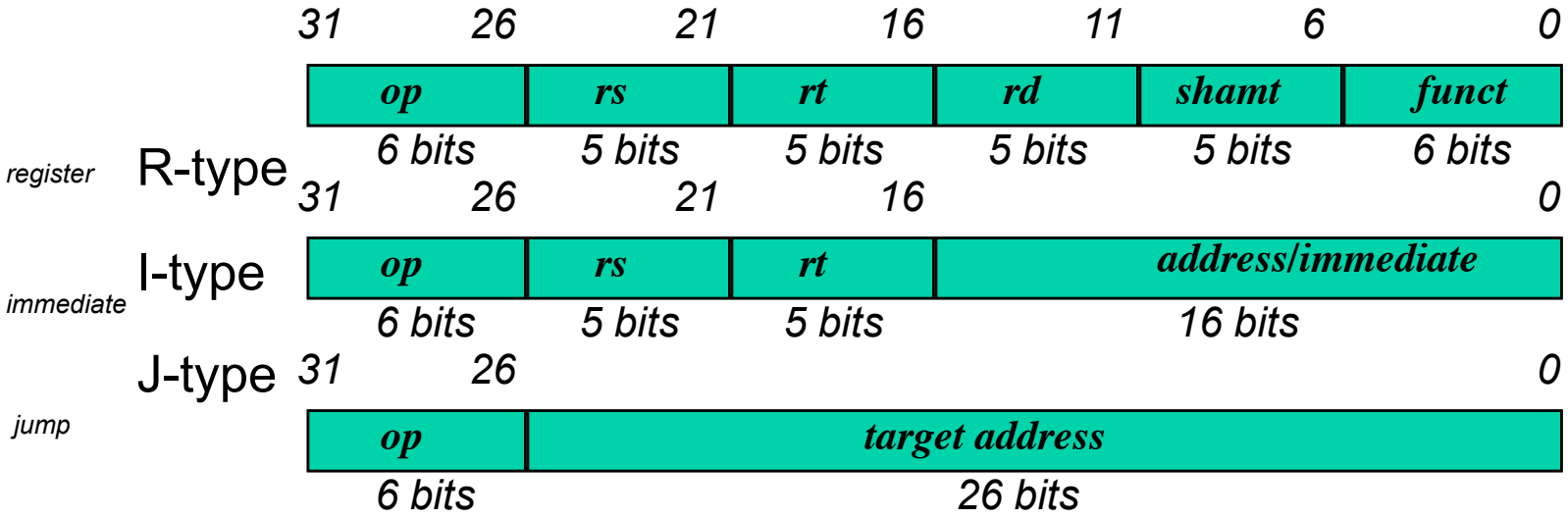


How to Design a Processor: step-by-step

1. Analyze instruction set architecture (ISA) ⟹ 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.

MIPS CPU Implementation - Datapath

Review: The MIPS Instruction Formats



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

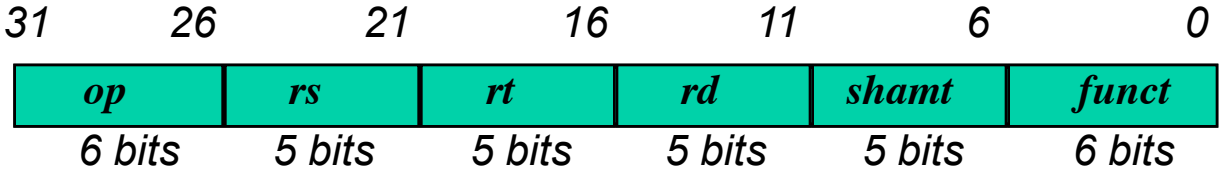
address / immediate: address offset or immediate value

target address: target address of jump instruction

Subset for Lecture

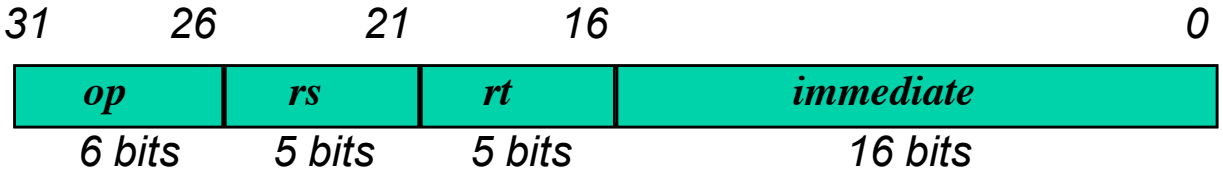
add, sub, or, slt

- addu rd,rs,rt
- subu rd,rs,rt



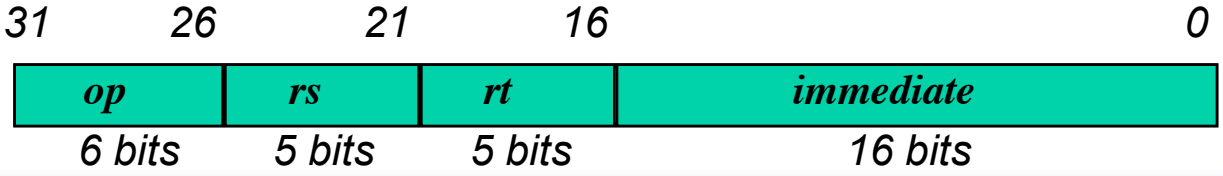
lw, sw

- lw rt,rs,imm16
- sw rt,rs,imm16



beq

- beq rs,rt,imm16



Register Transfer Descriptions

All start with instruction fetch:

```
{op , rs , rt , rd , shamt , funct} ← IMEM[ PC ]  OR
{op , rs , rt ,   Imm16} ← IMEM[ PC ]            THEN
```

inst	Register Transfers
add	`R[rd] ← R[rs] + R[rt],  PC ← PC + 4;`
sub	`R[rd] ← R[rs] - R[rt],  PC ← PC + 4;`
or	`R[rd] ← R[rs] | R[rt],  PC ← PC + 4;`
slt	`R[rd] ← (R[rs] < R[rt]) ? 1 : 0,  PC ← PC + 4;`
lw	`R[rt] ← DMEM[ R[rs] + sign_ext(Imm16)],` `PC ← PC + 4;`
sw	*DMEM[R[rs] + sign_ext(Imm16)] ← R[rt], PC ← PC + 4;*

beq	`if ( R[rs] == R[rt] ) then`
	`PC ← PC + 4 + {sign_ext(Imm16), 00}`
	`else PC ← PC + 4;`

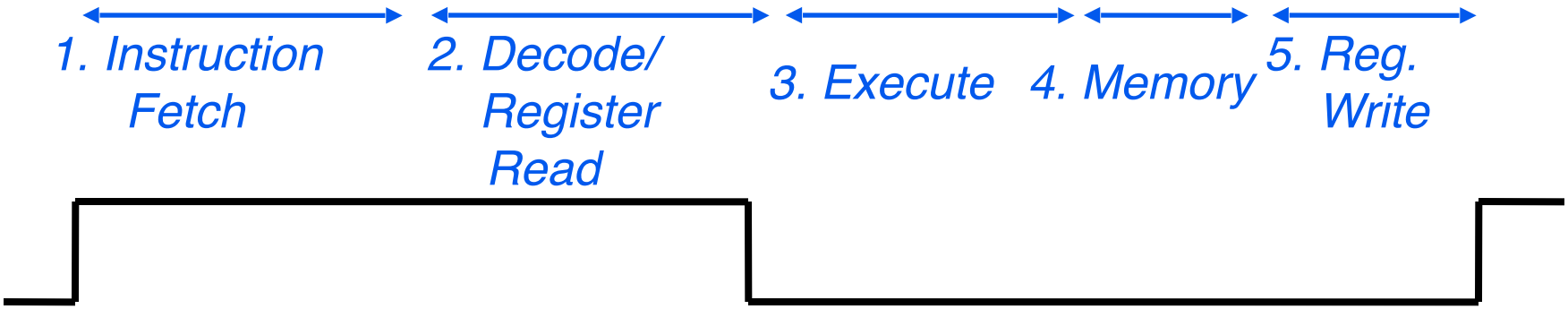
Microarchitecture

Multiple implementations for a single architecture:

- *Single-cycle*
  - Each instruction executes in a single clock cycle.
- *Multicycle*
  - Each instruction is broken up into a series of shorter steps with one step per clock cycle.
- *Pipelined (variant on "multicycle")*
  - Each instruction is broken up into a series of steps with one step per clock cycle
  - Multiple instructions execute at once by overlapping in time.
- *Superscalar*
  - Multiple functional units to execute multiple instructions at the same time
- *Out of order...*
  - Hey, who says we have to follow the program exactly....

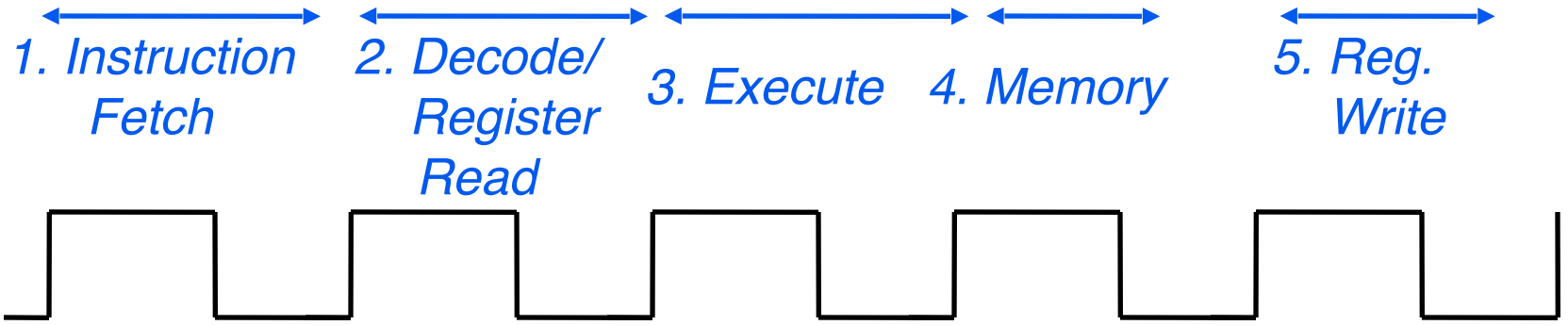
CPU clocking (1/2)

- 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.



CPU clocking (2/2)

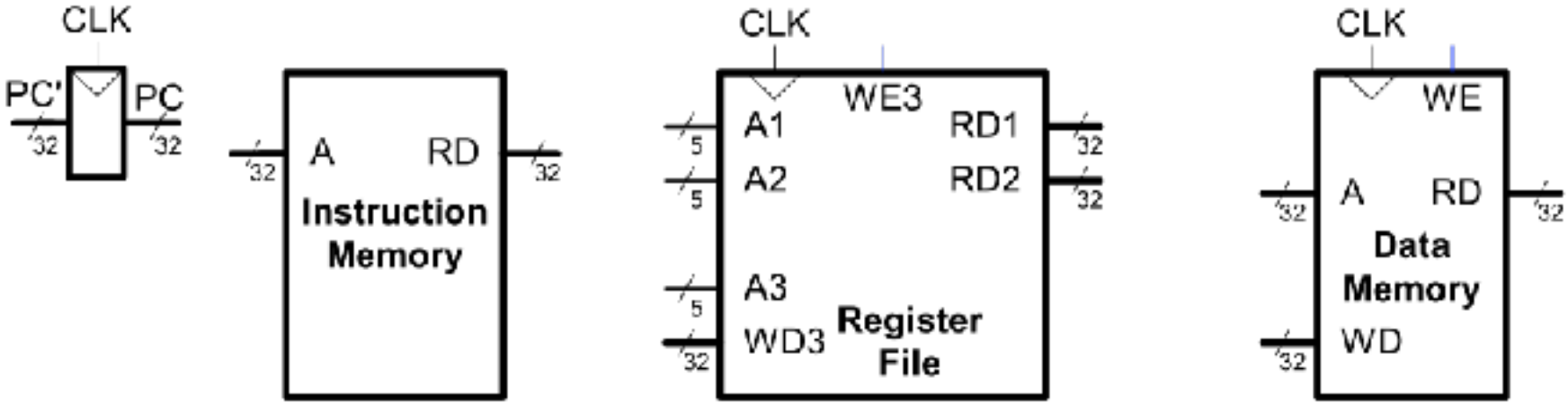
- Multiple-cycle CPU: Only one stage of instruction per clock cycle.
  - The clock is made as long as the slowest stage.



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

MIPS State Elements

- State encodes everything about the execution status of a processor:
  - **`PC`** register
  - 32 registers
  - Memory



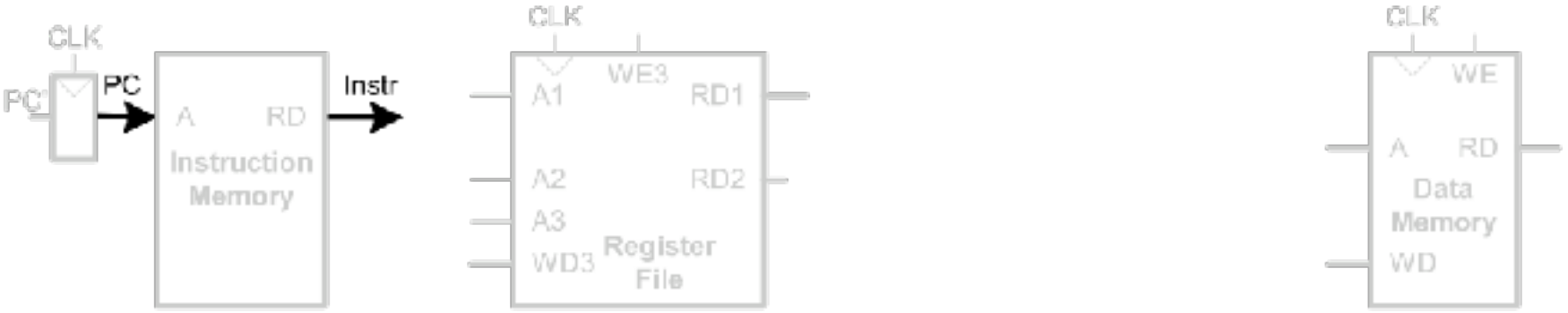
Note: for these state elements, clock is used for write but not for read (asynchronous read, synchronous write).

Single-Cycle Datapath: `lw` fetch

- First consider executing **`lw`**

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

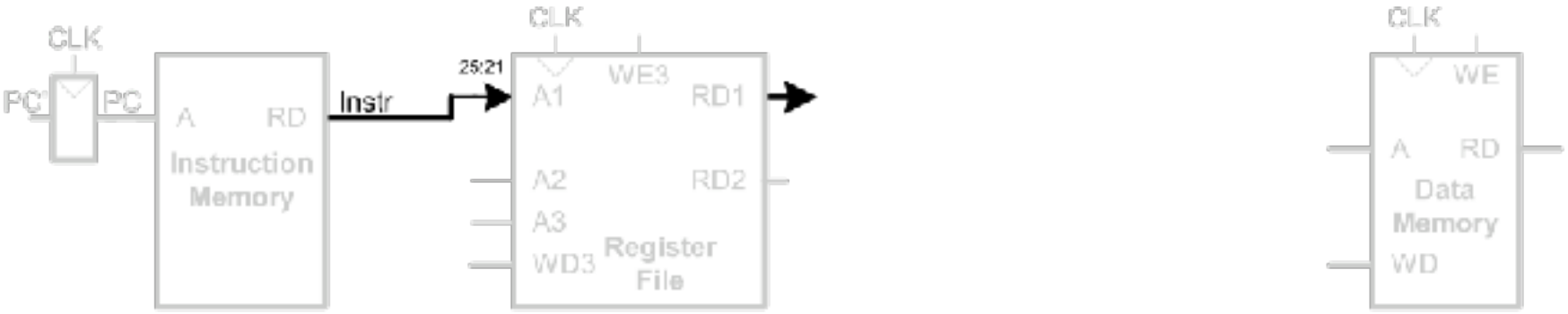
STEP 1: Fetch instruction



Single-Cycle Datapath: lw register read

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

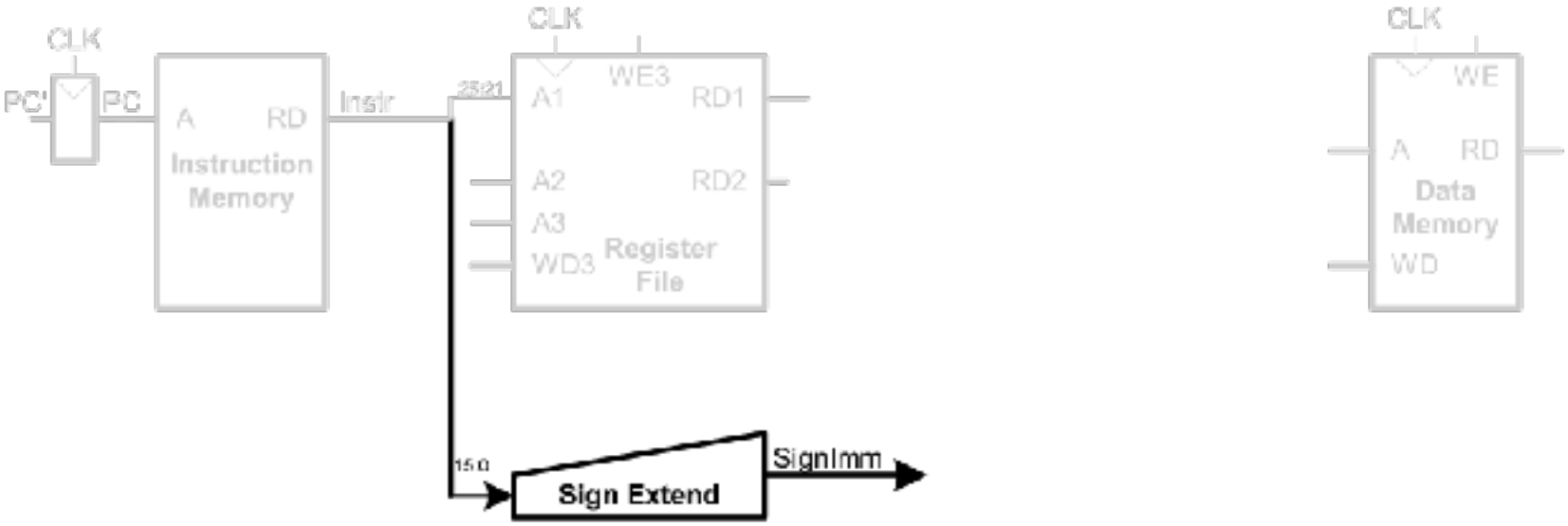
STEP 2: Read source operands from register file



Single-Cycle Datapath: `lw` immediate

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

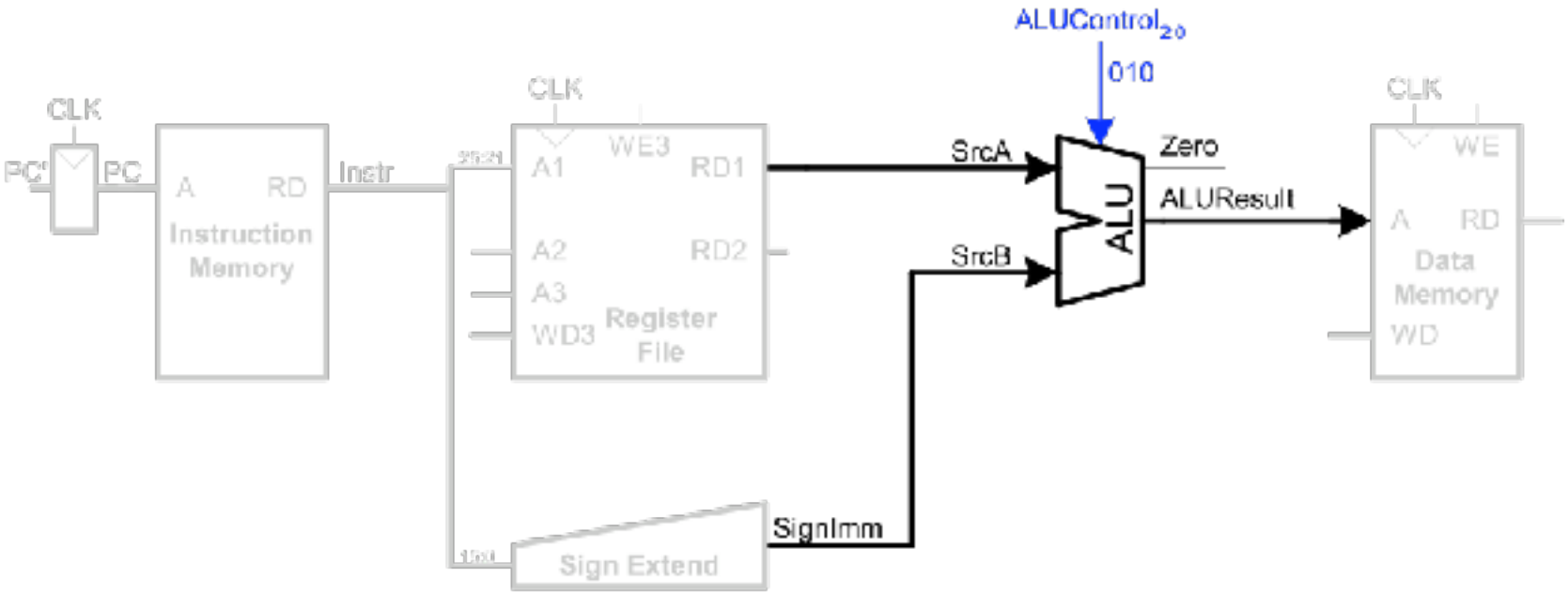
STEP 3: Sign-extend the immediate



Single-Cycle Datapath: lw address

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

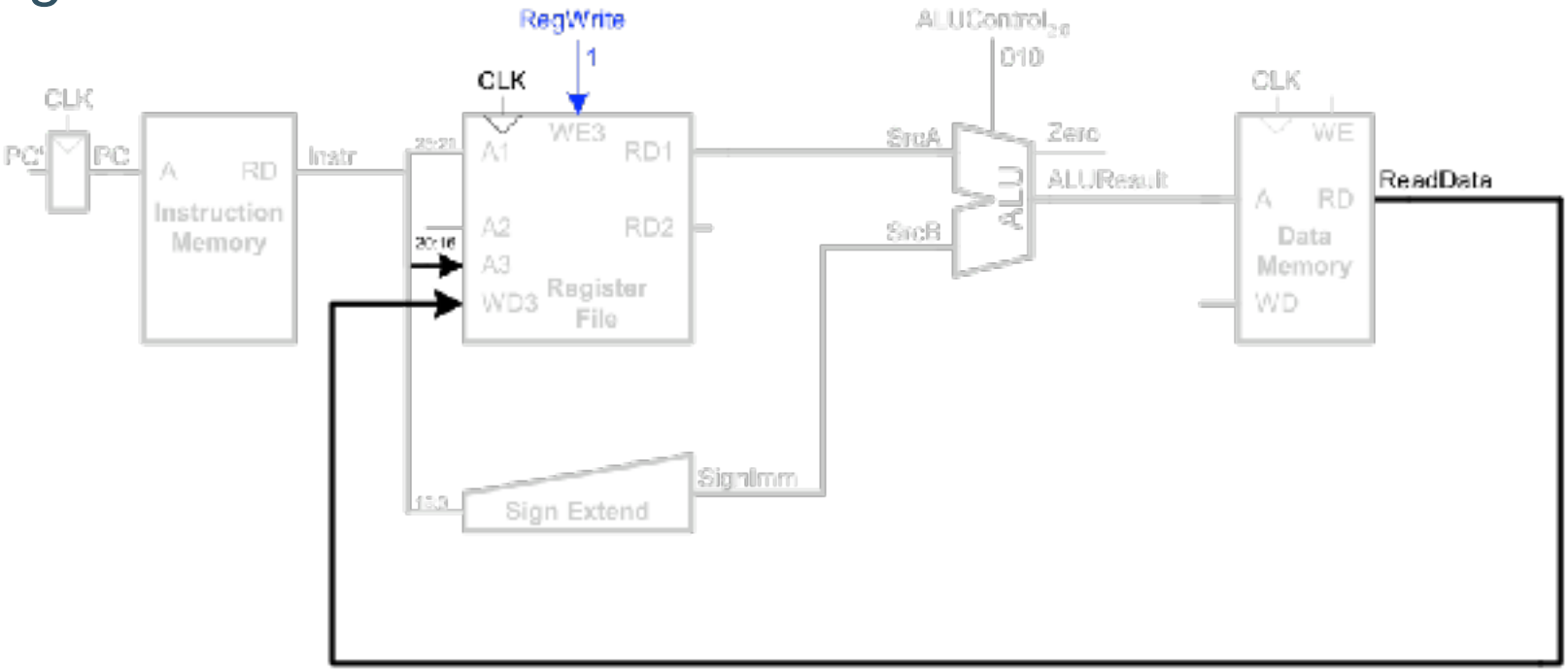
STEP 4: Compute the memory address



Single-Cycle Datapath: lw memory read

R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

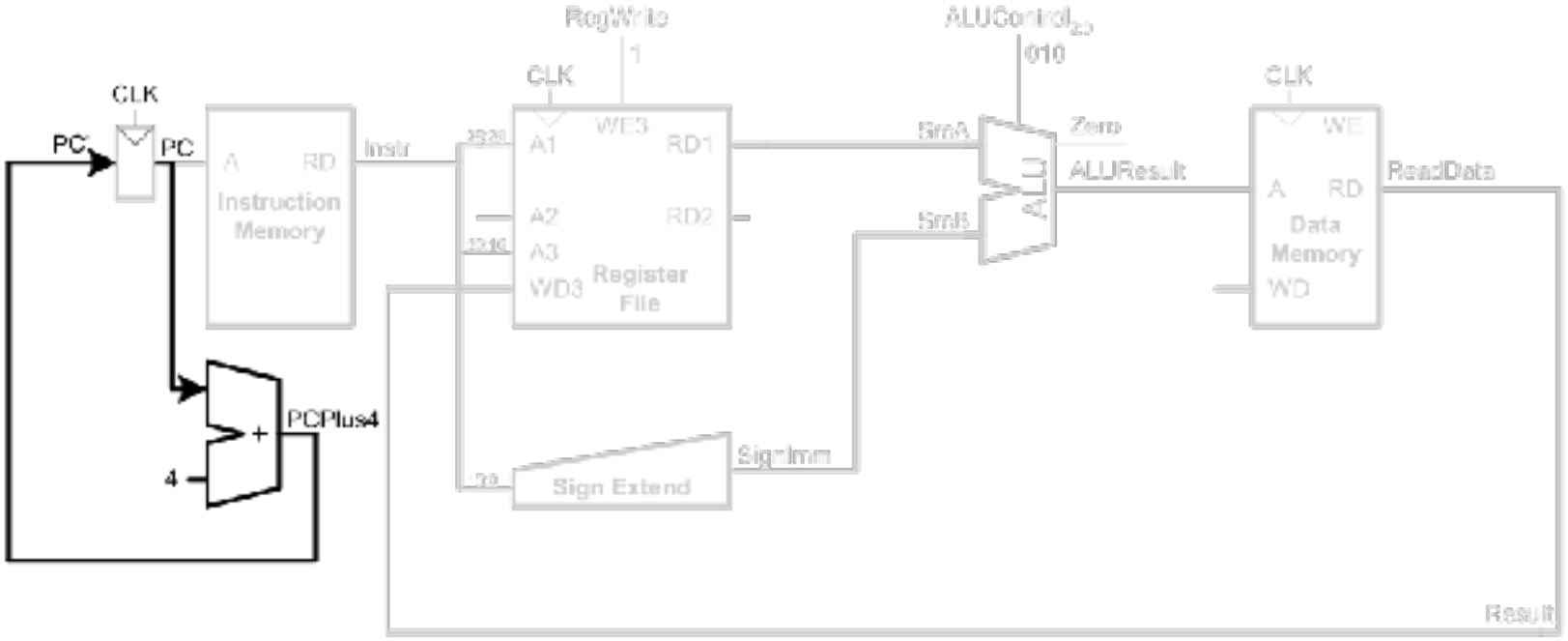
STEP 5: Read data from memory and write it back to register file



Single-Cycle Datapath: `lw` PC increment

STEP 6: Determine the address of the next instruction

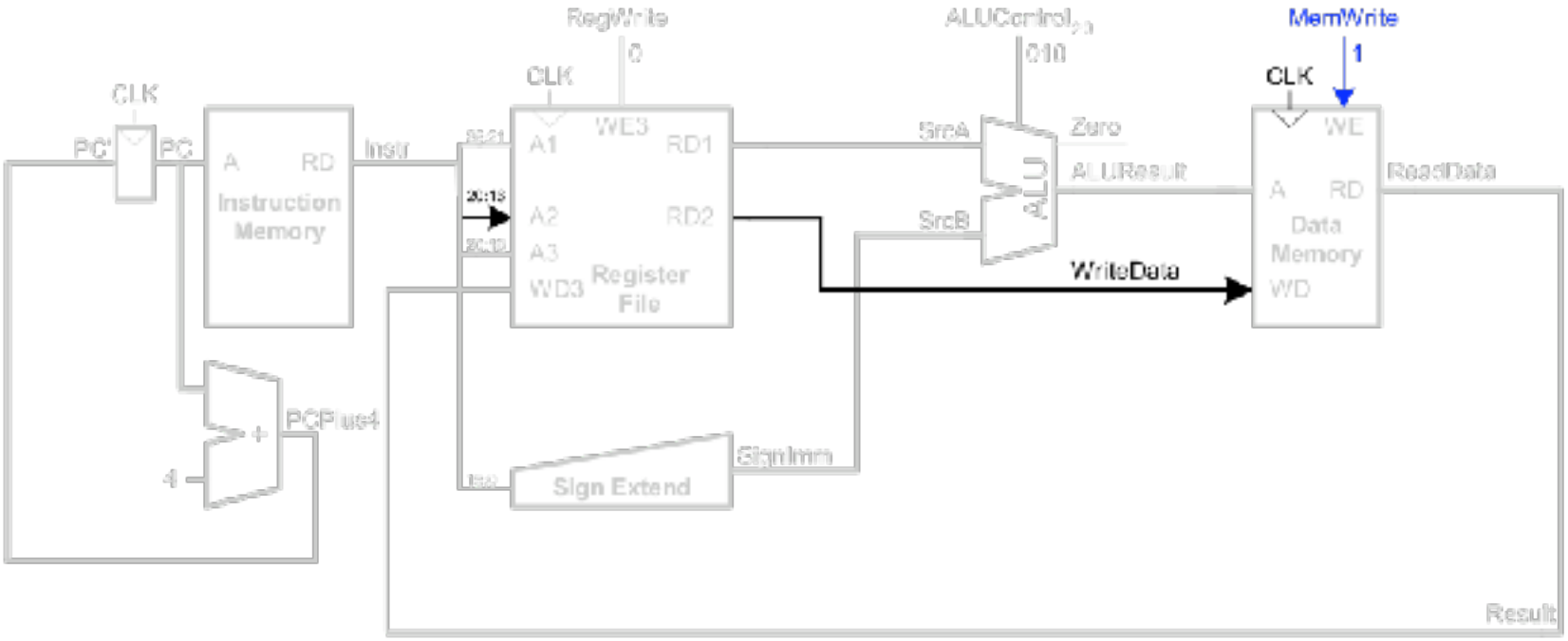
$$PC \leftarrow PC + 4$$



Single-Cycle Datapath: `sw`

DMEM[R[rs] + sign_ext(Imm16)] ← R[rt]

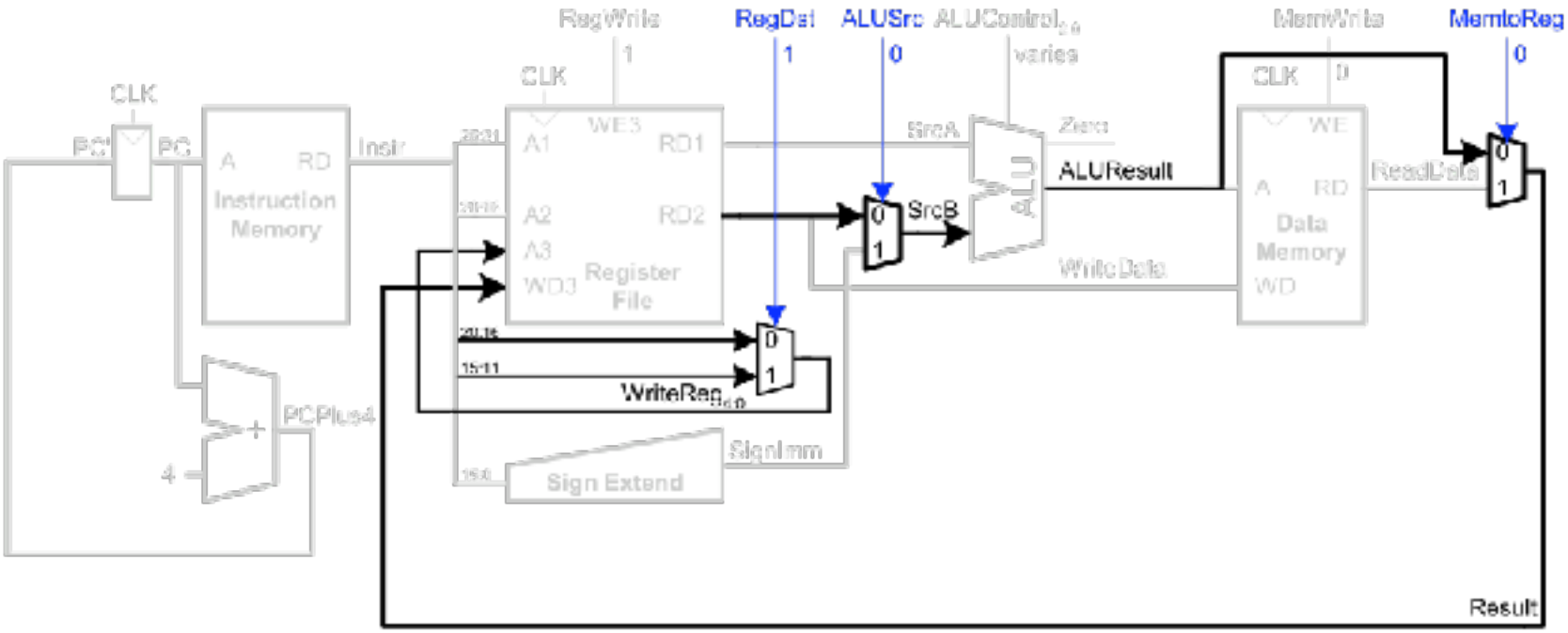
- Write data in `rt` to memory



Single-Cycle Datapath: R-type instructions

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

R[rd] ← R[rs] op R[rt]

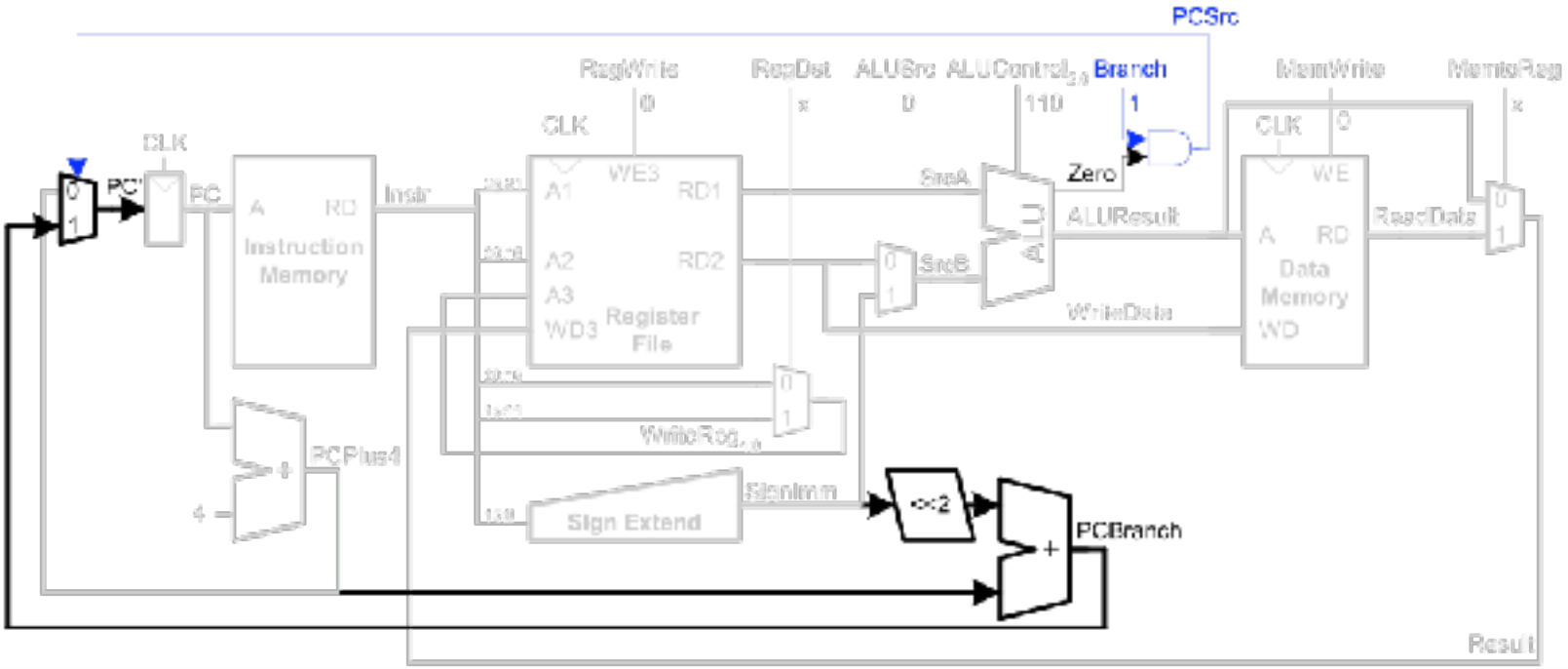


Single-Cycle Datapath: beq

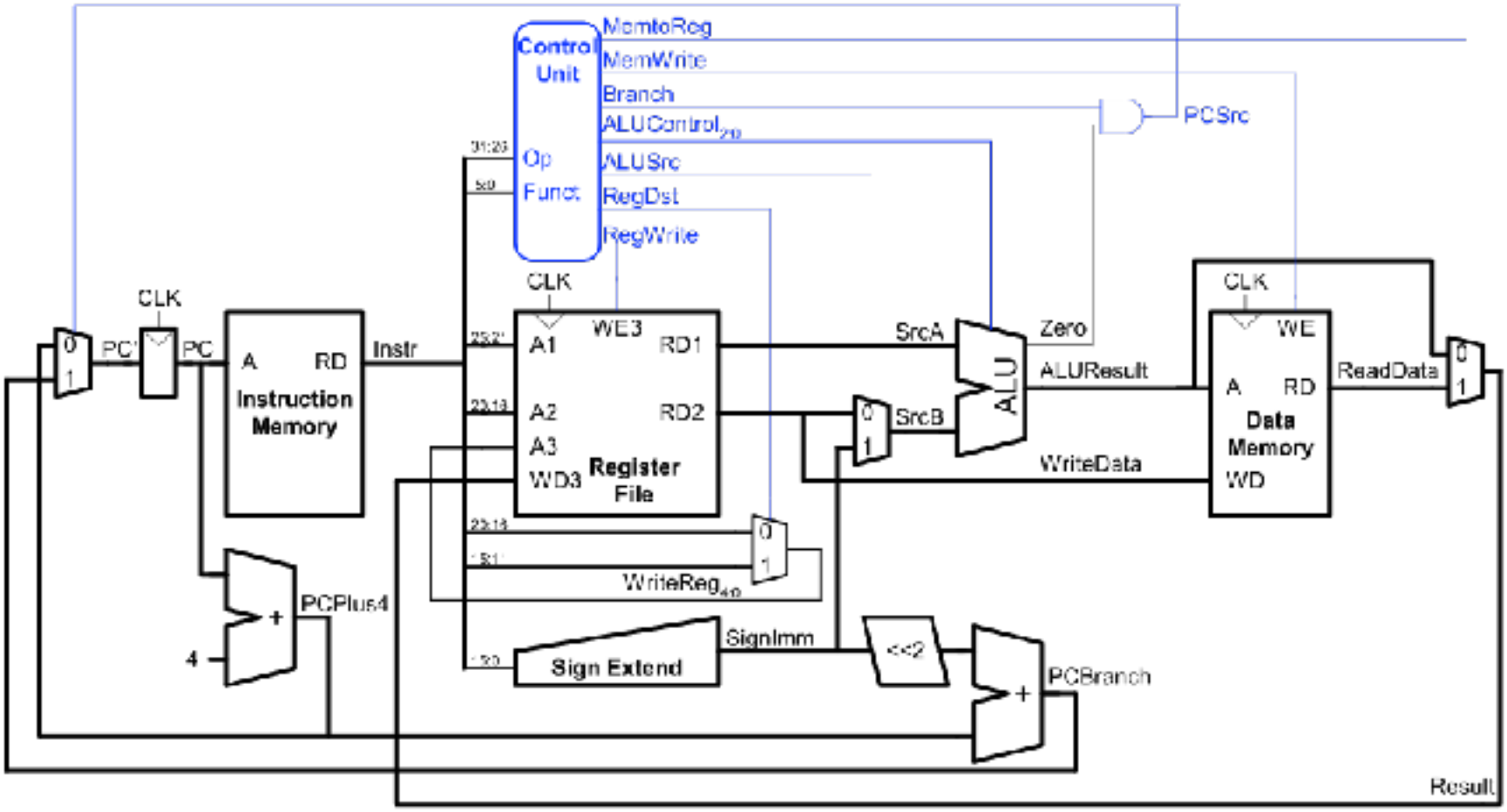
if (R[rs] == R[rt]) then PC ← PC + 4 + {sign_ext(Imm16), 00}

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

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

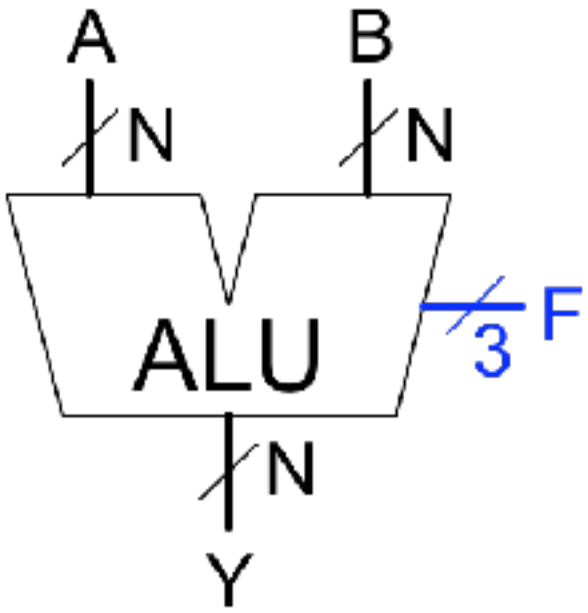


Complete Single-Cycle Processor



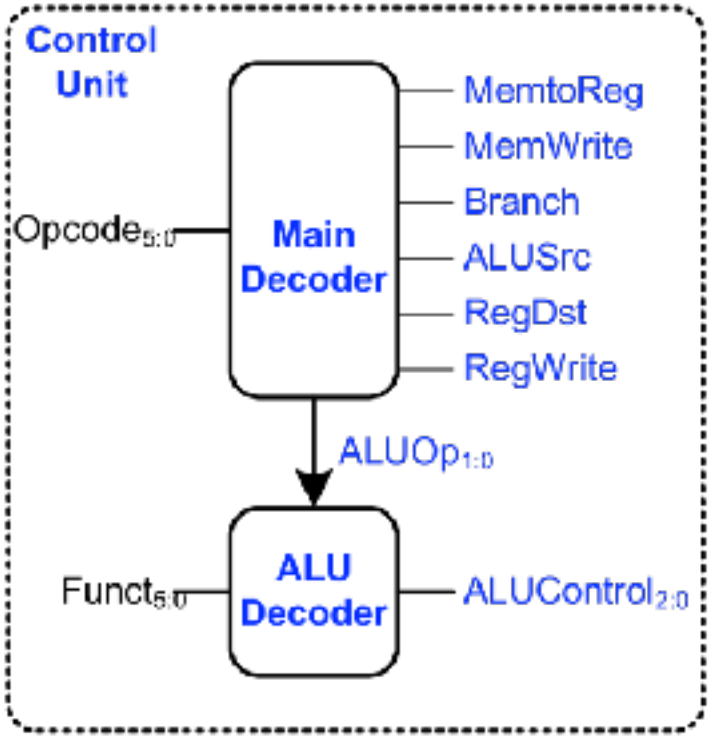
MIPs Processor Implementation - Control

ALU Control



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

Control Unit



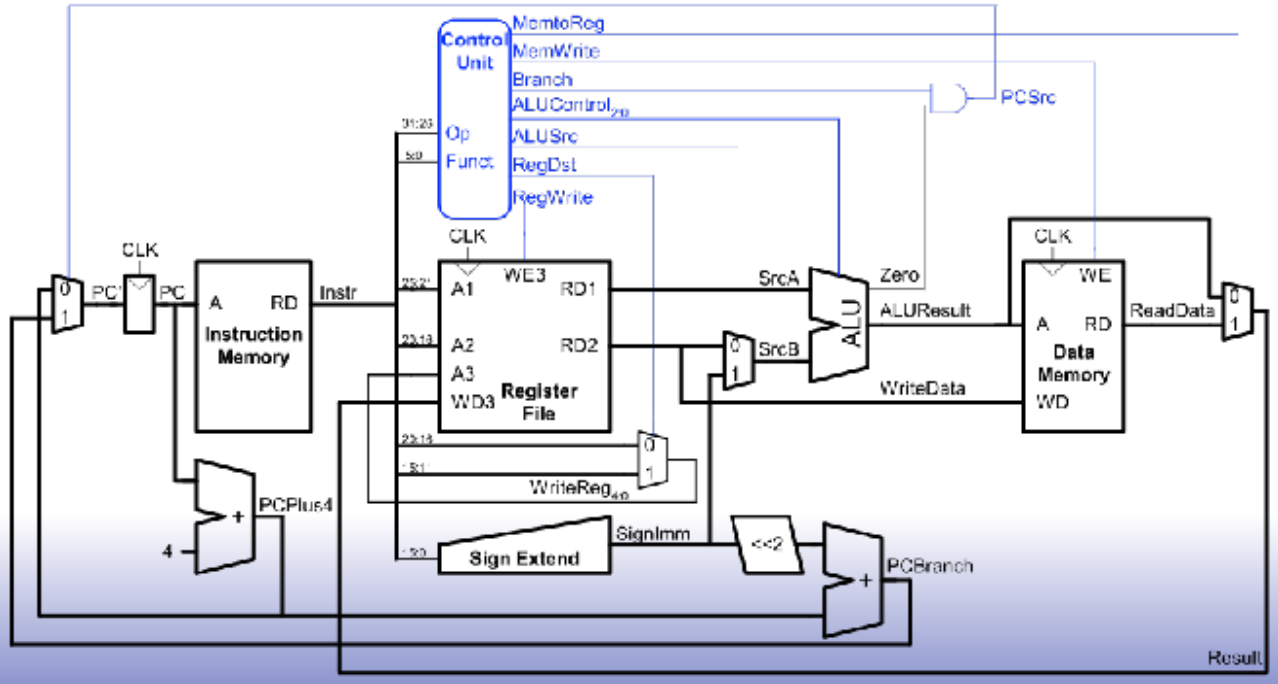
Control Unit: ALU Decoder

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

$ALUOp_{1:0}$	Funct	$ALUControl_{2:0}$
00	XXXXXX	010 (Add)
X1	XXXXXX	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)

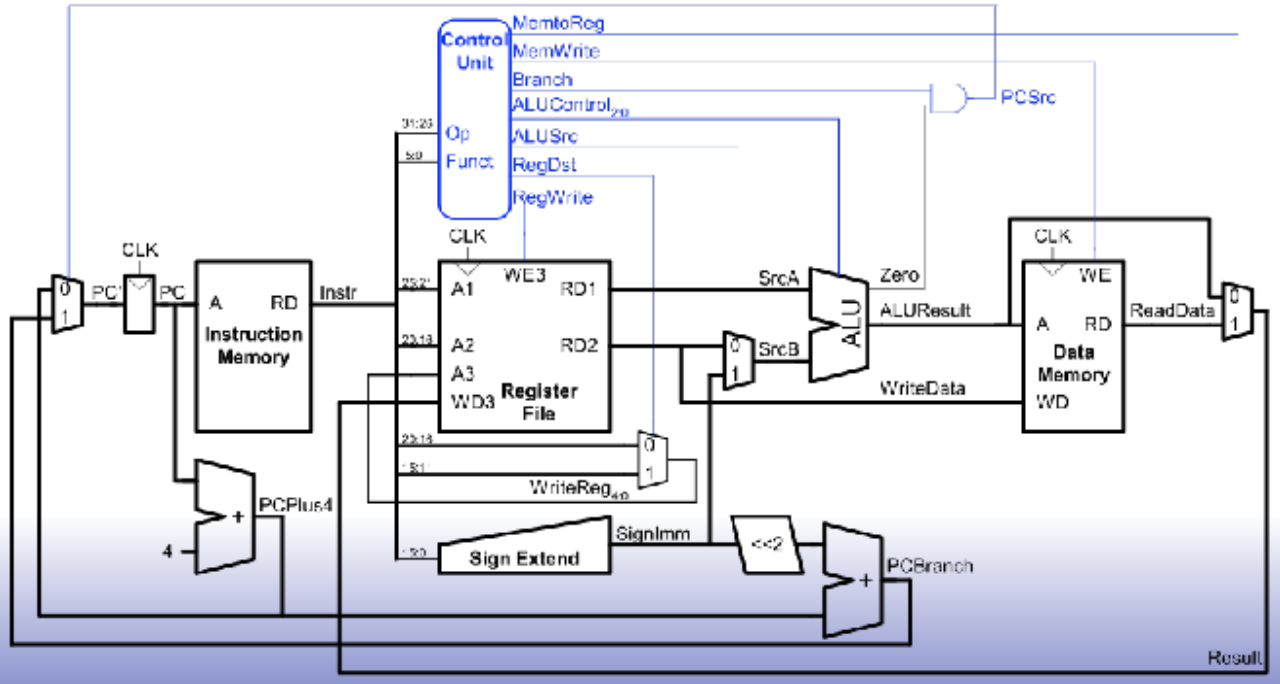
Control Unit: Main Decoder

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

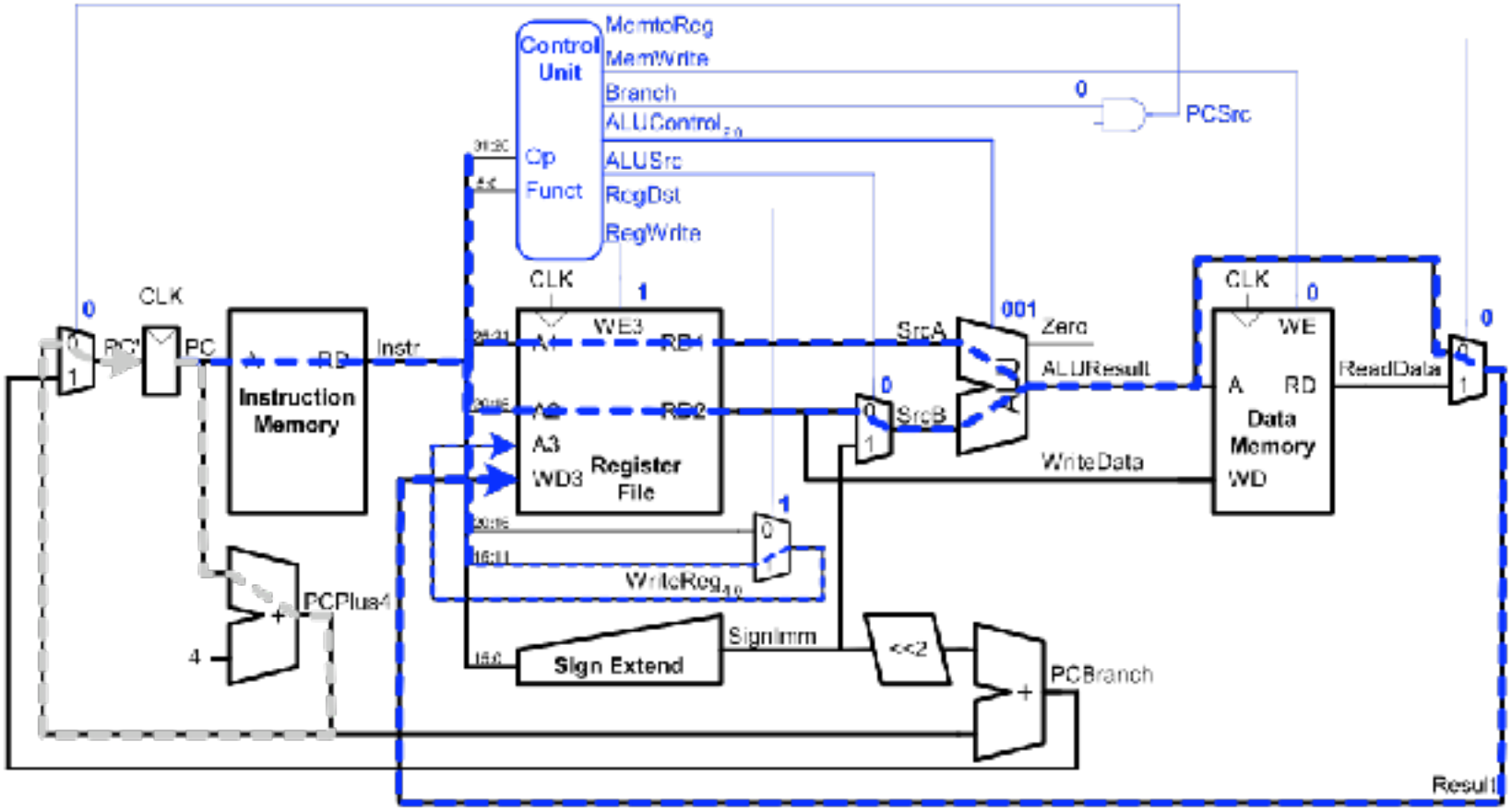


Control Unit: Main Decoder

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	1	00
`sw`	101011	0	X	1	0	1	X	00
`beq`	000100	0	X	0	1	0	X	01

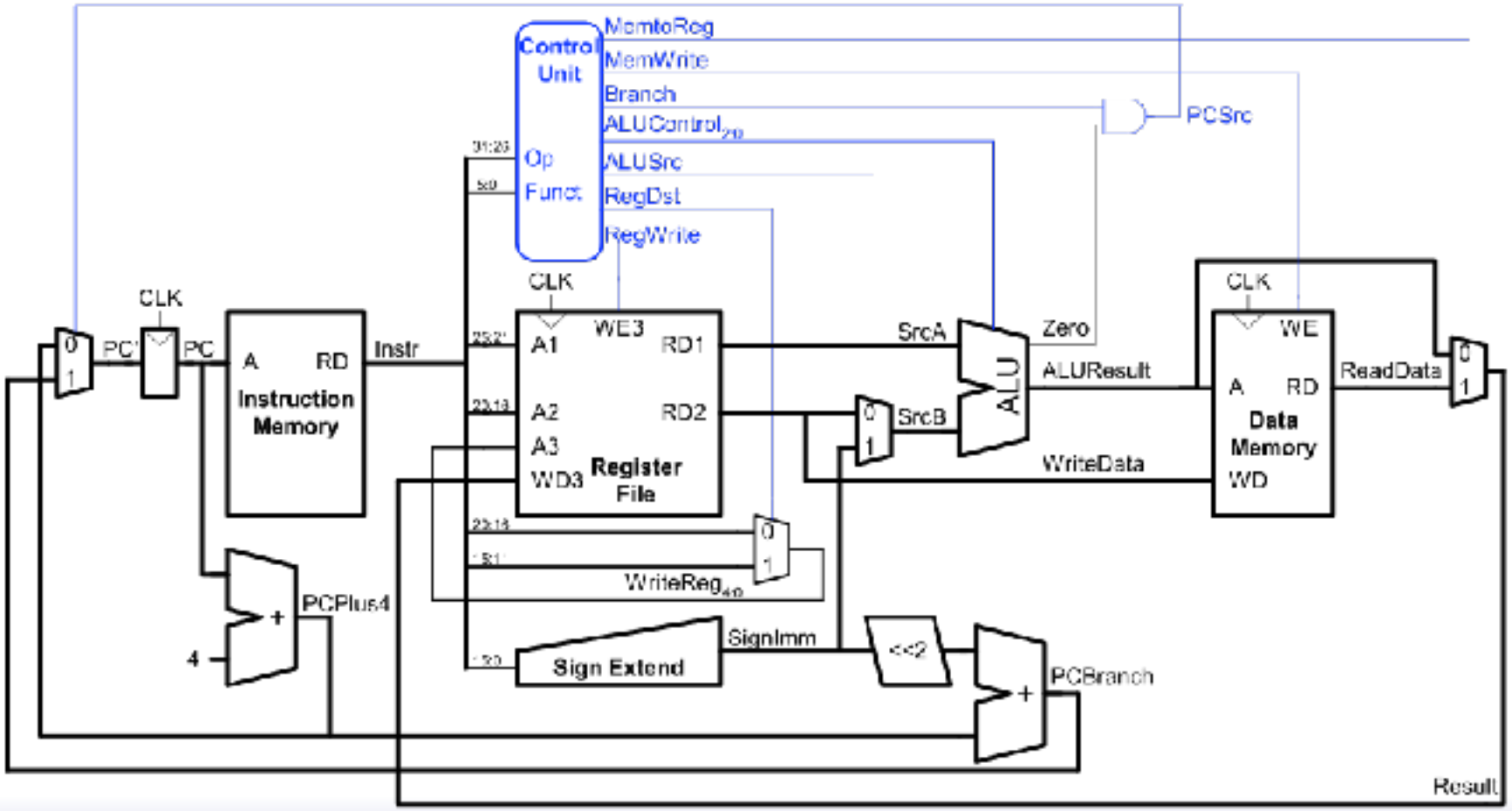


Single-Cycle Datapath Example: `or`



Extended Functionality: addi

- *No change to datapath*



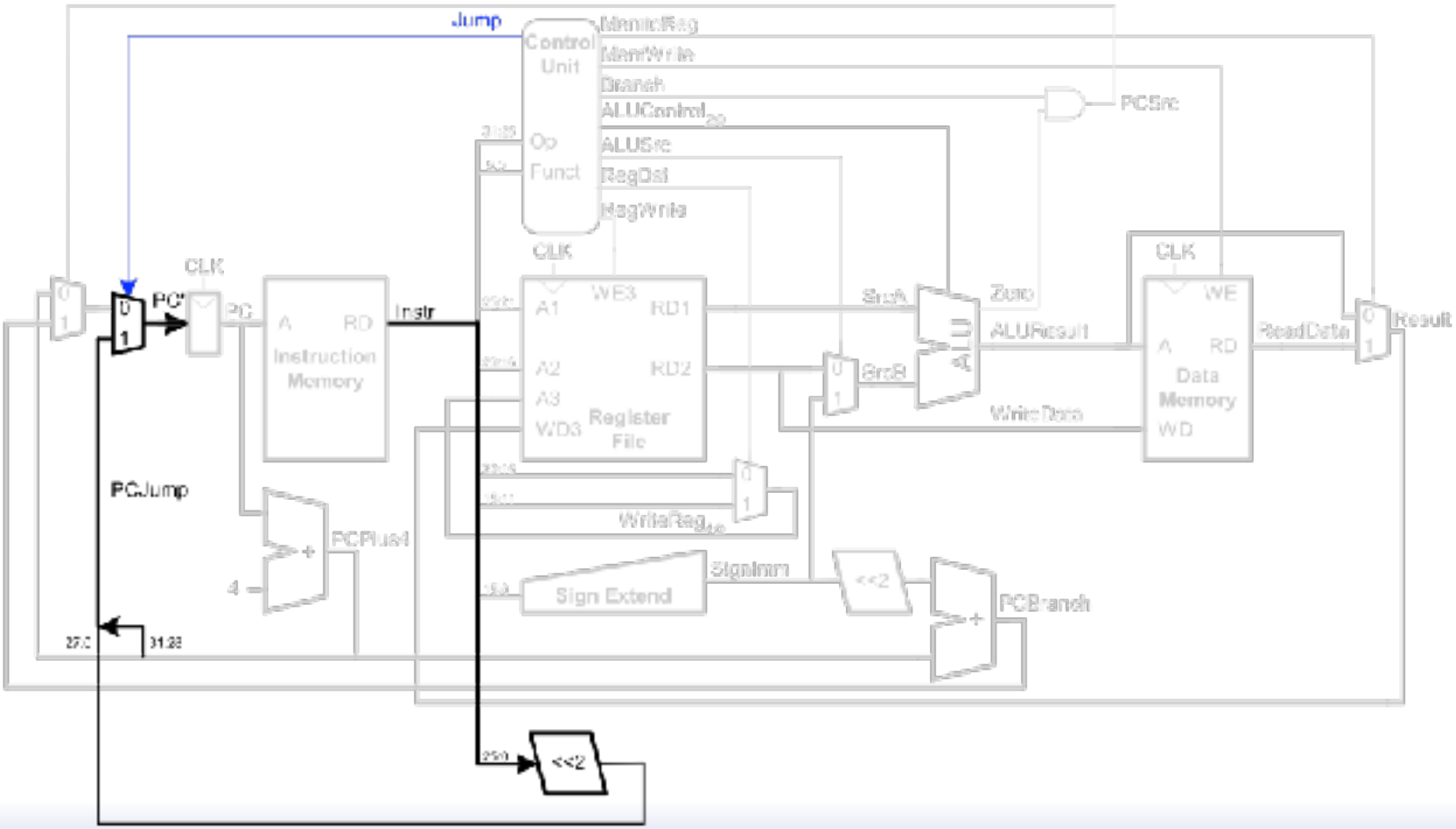
Control Unit: `addi`

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	0	1	1	0	0	0	0	10	0
`lw`	100011	1	0	1	0	0	1	0	0
`sw`	101011	0	X	1	0	1	X	0	0
`beq`	100	0	X	0	1	0	X	1	0
`j`	**100**	**0**	**X**	**X**	**X**	**0**	**X**	**XX**	**1**

Reminder on Don't Cares (X)

- You can use don't cares on any signal that doesn't change state
  - Gives the synthesis tool freedom, at the cost of potential bugs if you mess up
- You ***must never*** specify don't care on signals which cause side effects
  - Otherwise the tool ***will*** cause unintended writes

A Verilog Convention

- Control logic works really well as a case statement...

```
always @* begin
    op = instr[26:31];
    imm = instr[15:0]; ...

    reg_dst = 1'bx;   // Don't care
    reg_write = 1'b0; // Do care, side effecting
    ...
    case (op)
     6'b000000: begin reg_write = 1; ... end
     ...
```

Processor Pipelining

Review: Processor Performance
(The Iron Law)

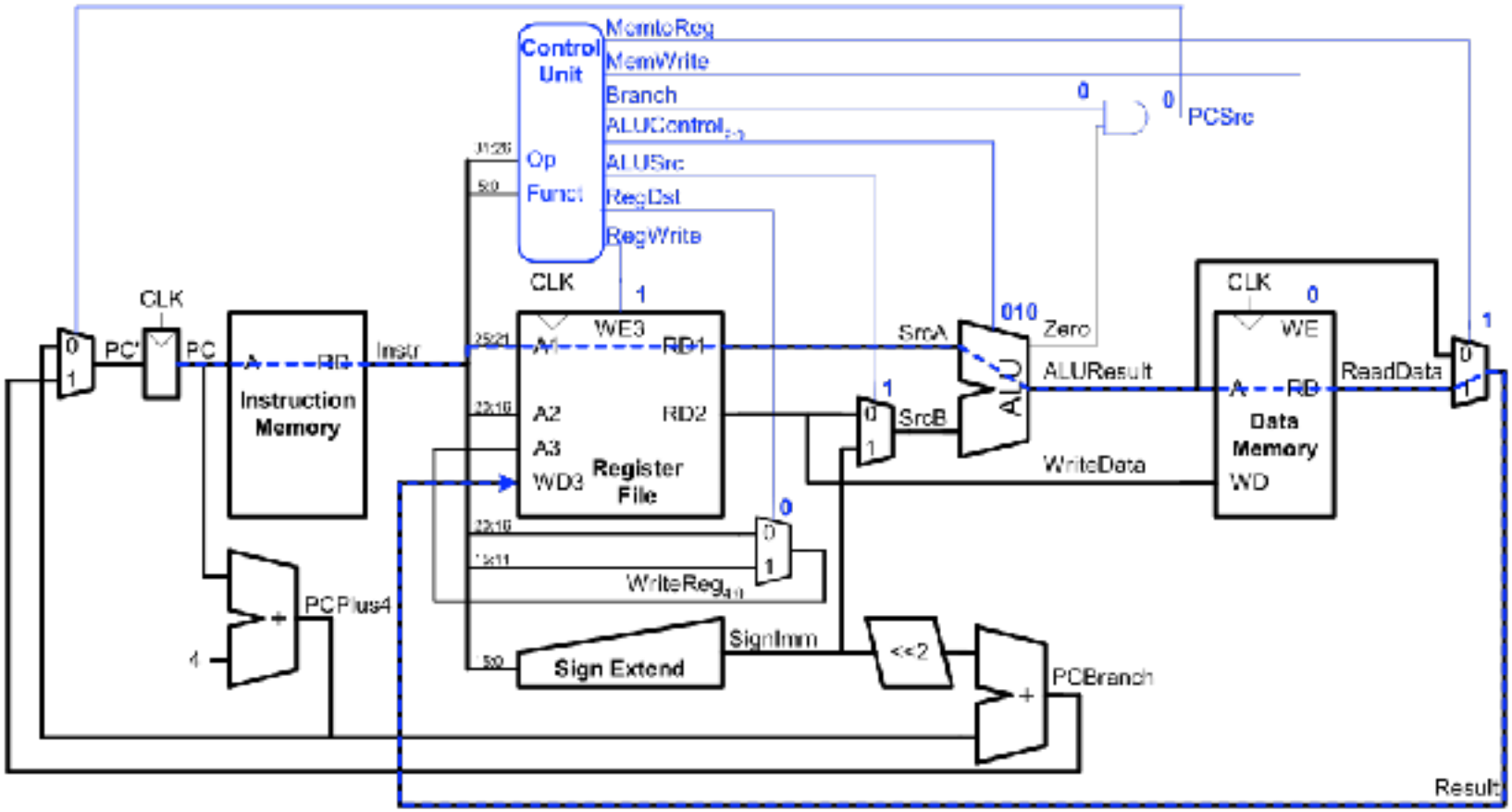
Program Execution Time

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

= # instructions x CPI x T_C

Single-Cycle Performance

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



Single-Cycle Performance

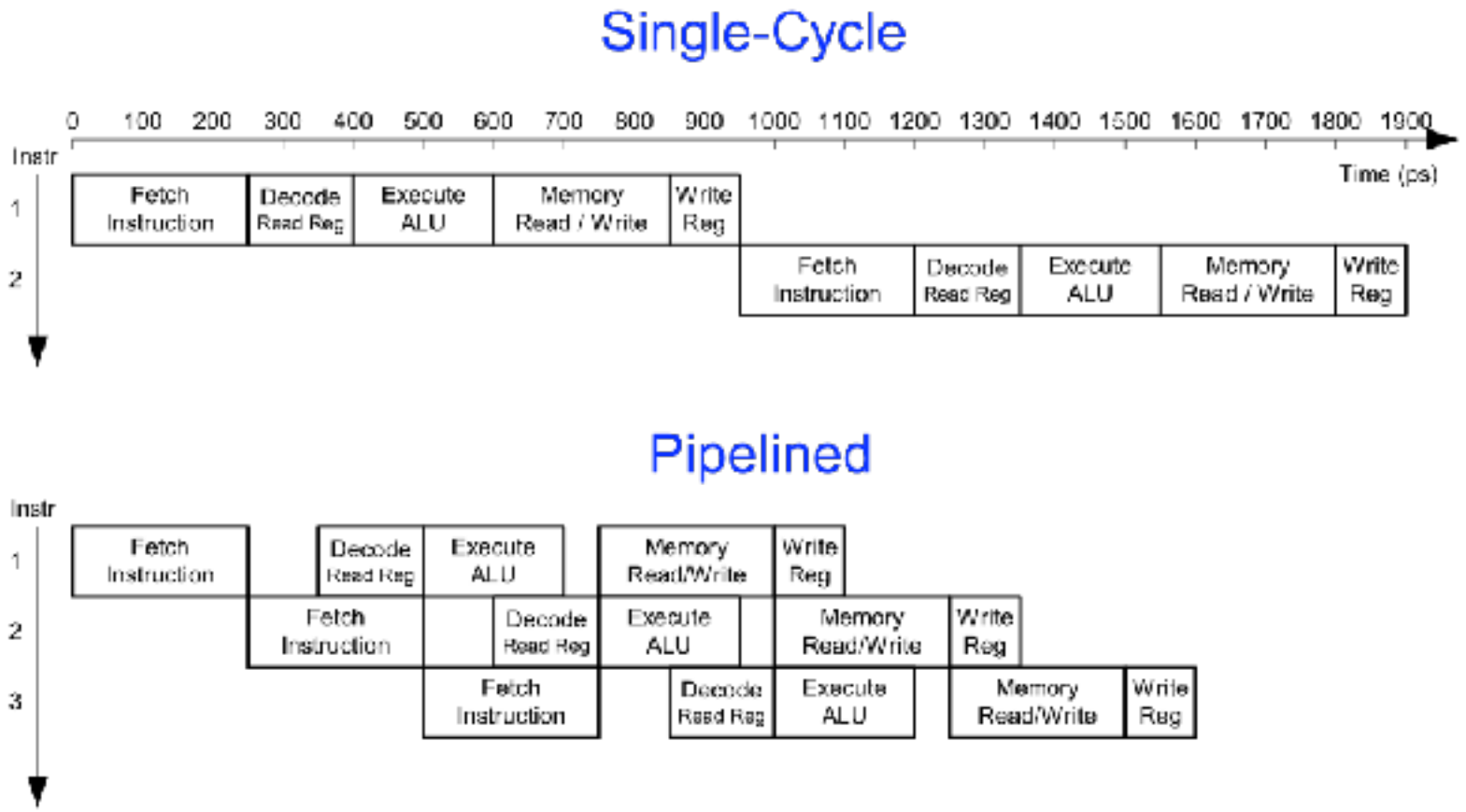
- *Single-cycle critical path:*
  $T_c = t_{q\_PC} + t_{mem} + max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$

- *In most implementations, limiting paths are:*
  - *memory, ALU, register file.*
  - $T_c = t_{q\_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$

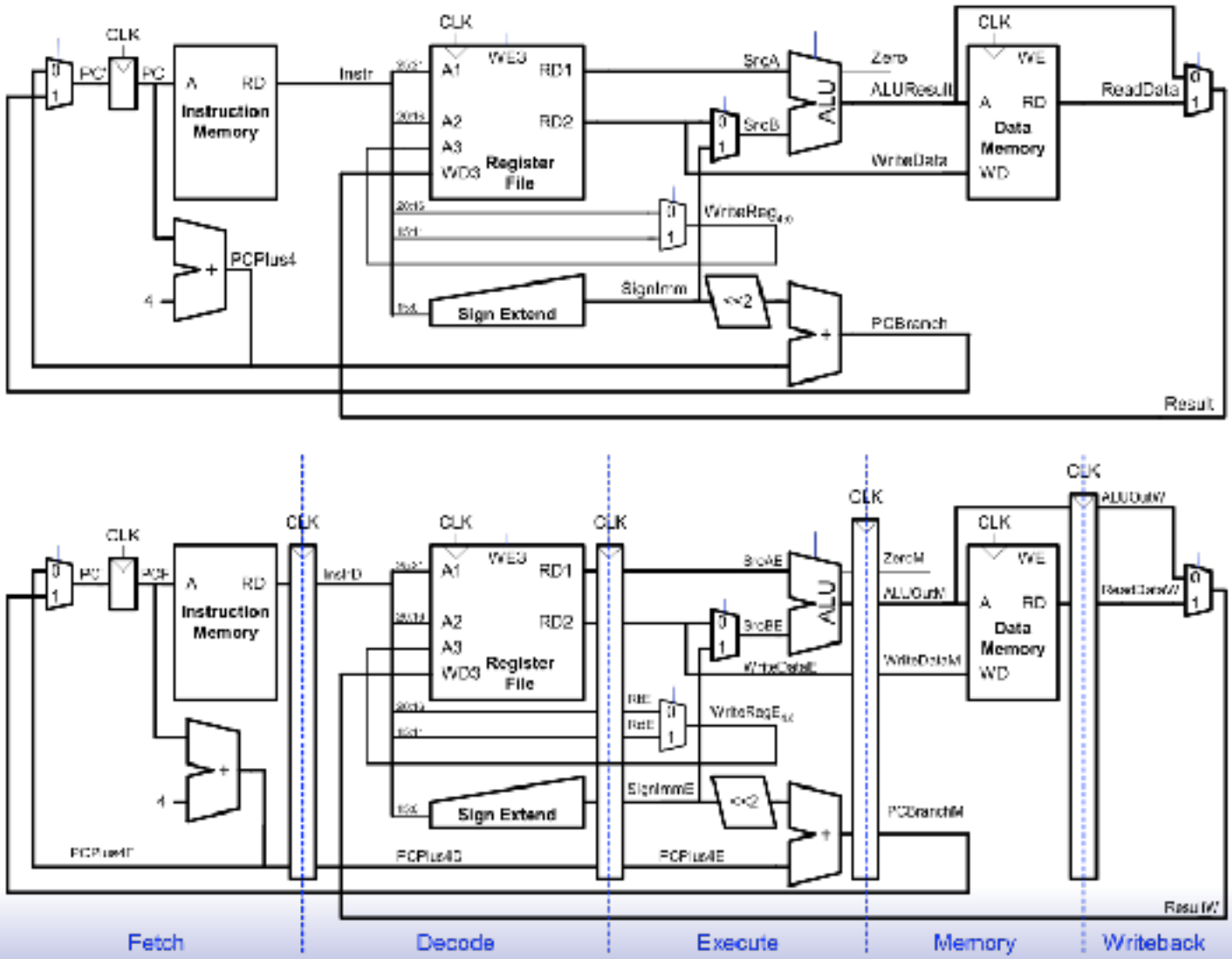
Pipelined MIPS Processor

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

Single-Cycle vs. Pipelined Performance

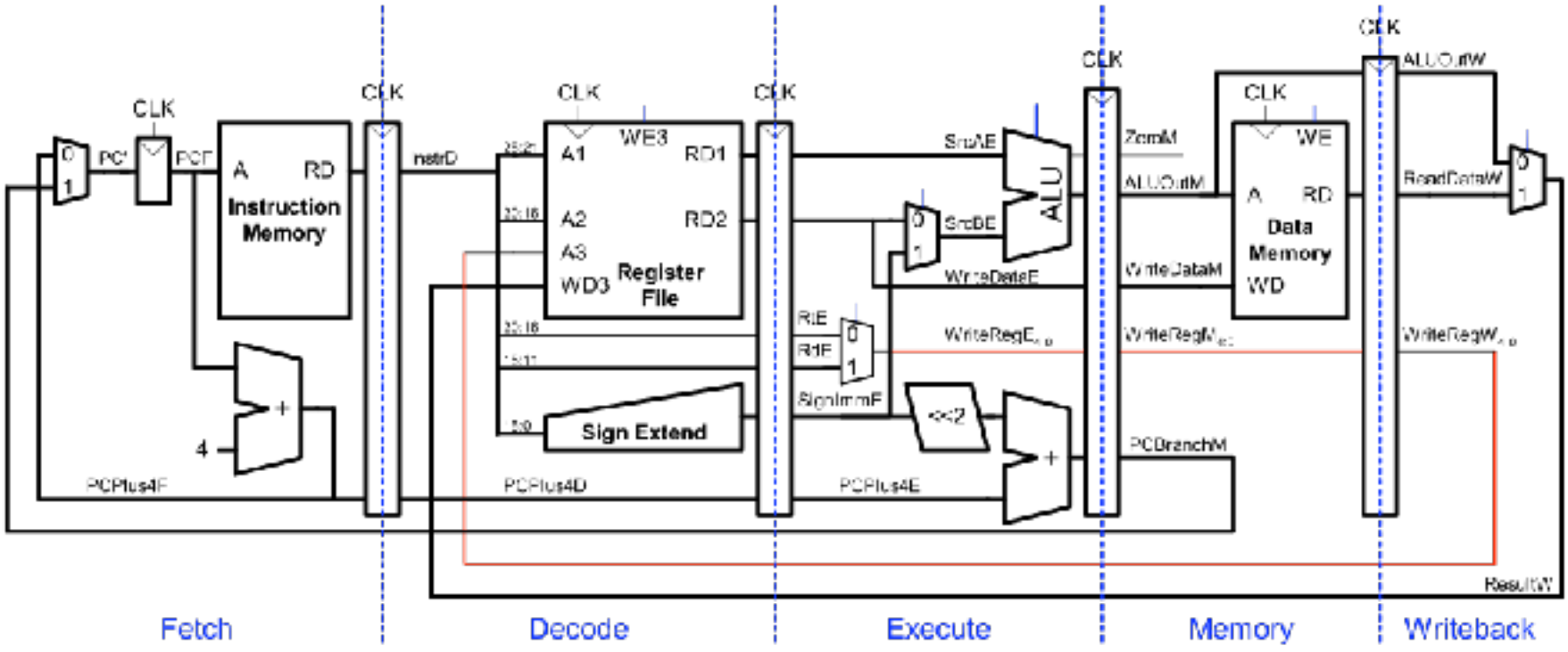


Single-Cycle and Pipelined Datapath

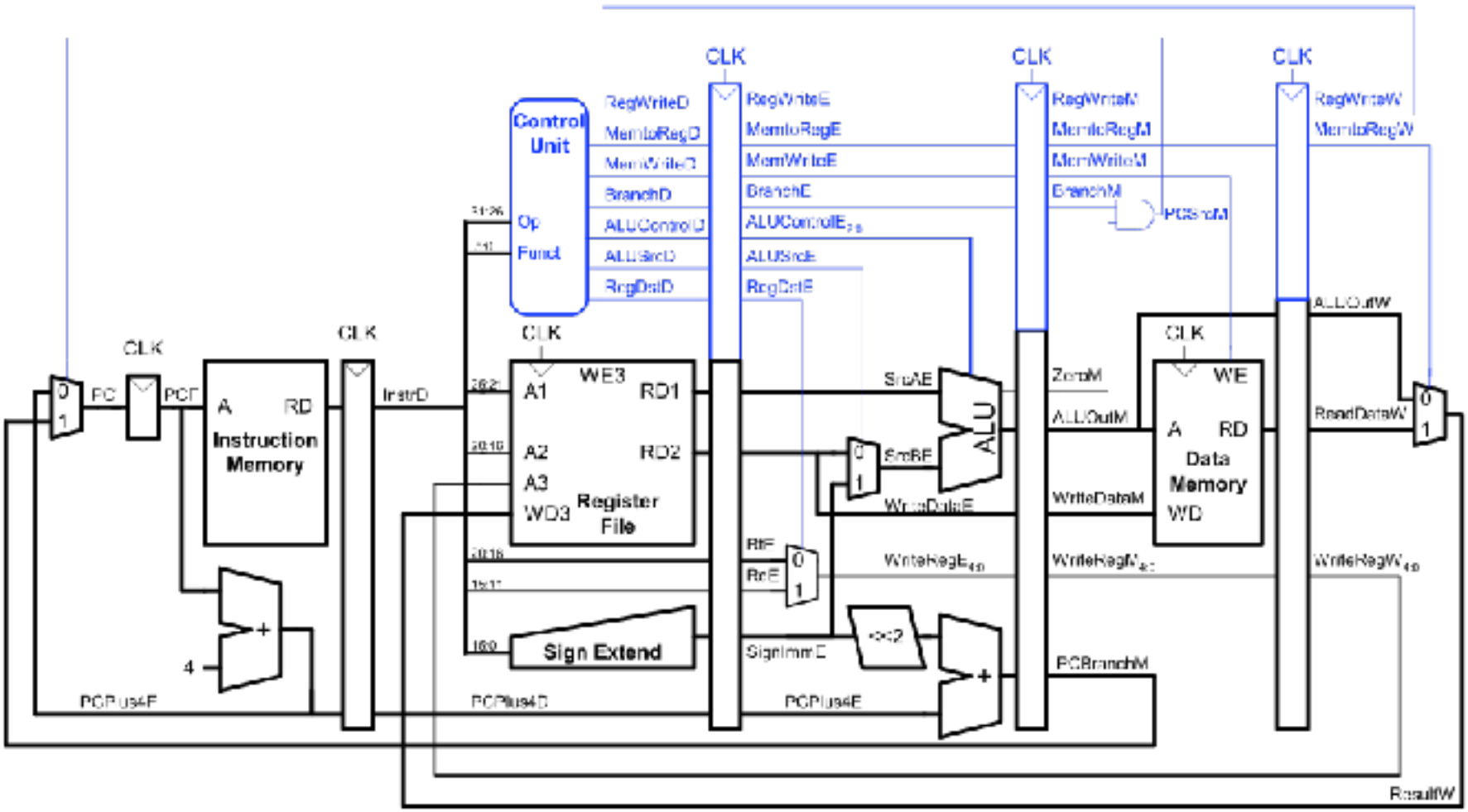


Corrected Pipelined Datapath

- *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 Hazards

- 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)

Processor Pipelining

Deeper pipeline example.

IF1	**IF2**	**ID**	**X1**	**X2**	**M1**	**M2**	**WB**	
	IF1	**IF2**	**ID**	**X1**	**X2**	**M1**	**M2**	**WB**

Deeper pipelines => less logic per stage => high clock rate.

But

Deeper pipelines => more hazards => more cost and/or higher CPI.*

Cycles per instruction might go up because of unresolvable hazards.

Remember, Performance = # instructions X Frequency$_{clk}$ / CPI

**Many designs included pipelines as long as 7, 10 and even 20 stages (like in the Intel Pentium 4). The later "Prescott" and "Cedar Mill" Pentium 4 cores (and their Pentium D derivatives) had a 31-stage pipeline.*

How about shorter pipelines ... Less cost, less performance

3-Stage Pipeline

3-Stage Pipeline (used for project)

The blocks in the datapath with the greatest delay are: IMEM, ALU, and DMEM. Allocate one pipeline stage to each:

Use PC register as address to IMEM and retrieve next instruction. Instruction gets stored in a pipeline register, also called "instruction register", in this case.

Use ALU to compute result, memory address, or compare registers for branch.

Access data memory or I/O device for load or store. Allow for setup time for register file write.

Most details you will need to work out for yourself. Some details to follow ... In particular, let's look at hazards.

3-stage Pipeline

Data Hazard

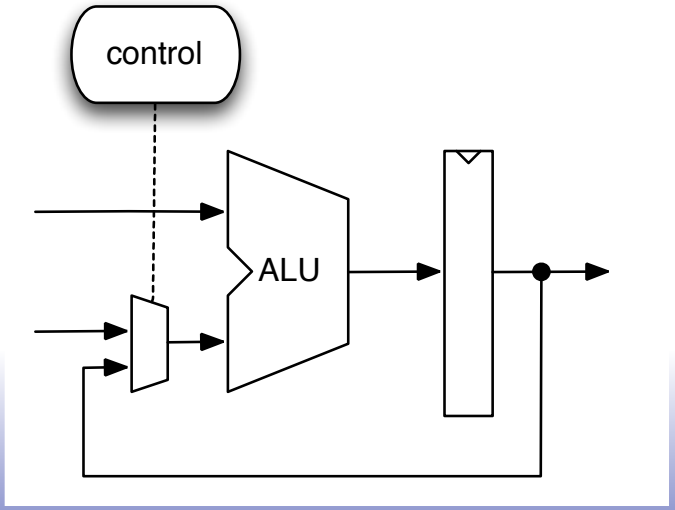
```
add  $5, $3, $4   I    X    M
add  $7, $6, $5        I    X    M
```

reg 5 value needed here!

reg 5 value updated here

The fix:

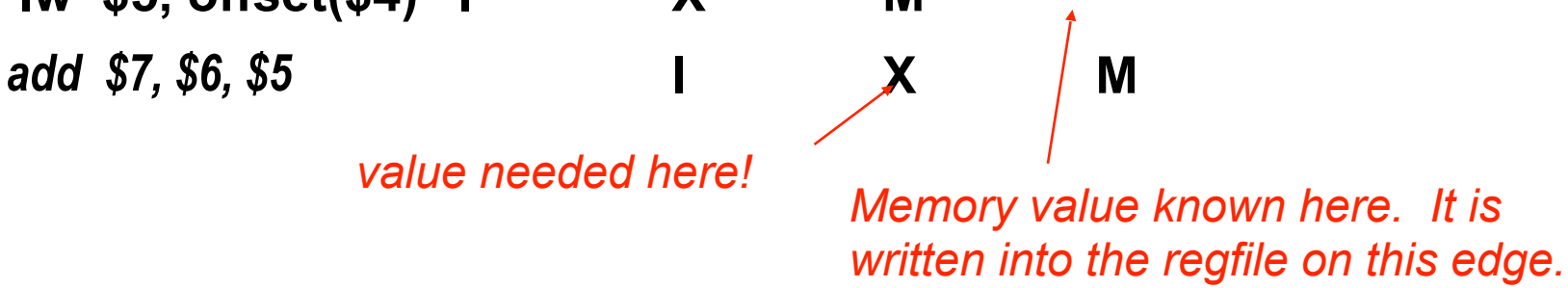
Selectively forward ALU result back to input of ALU.



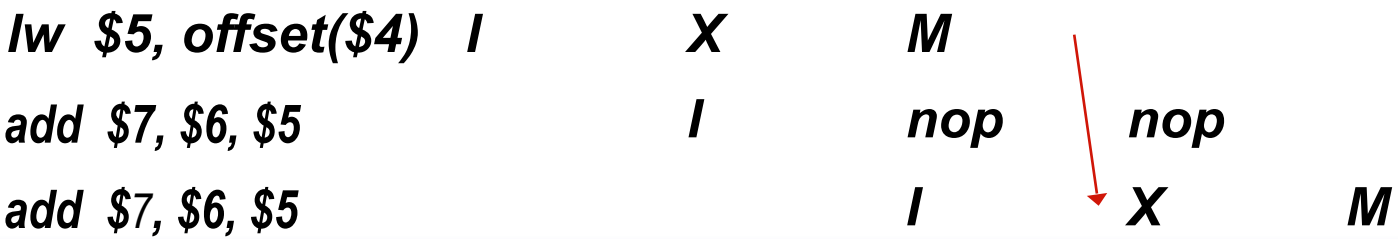
- *Need to add mux at input to ALU, add control logic to sense when to activate. Check book for details.*

3-stage Pipeline

Load Hazard

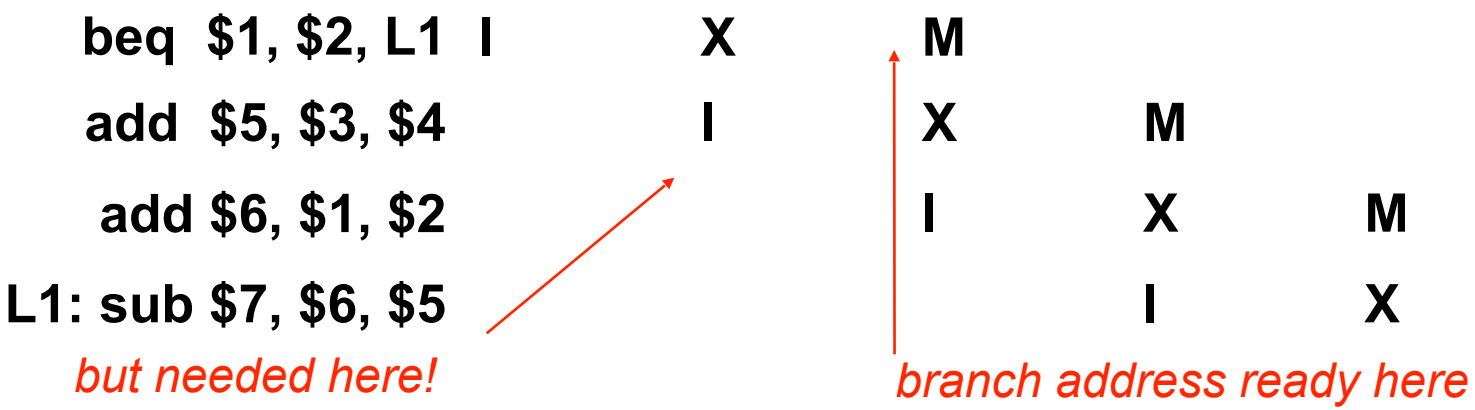


The fix: *Delay the dependent instruction by one cycle to allow the load to complete, send the result of load directly to the ALU.*



3-stage Pipeline

Control Hazard



The fix:

*Several Possibilities:**

1. *Always delay fetch of instruction after branch*
2. *Assume branch "not taken", continue with instruction at PC+4, and correct later if wrong.*
3. *Predict branch taken or not based on history (state) and correct later if wrong.*

1. *Simple, but all branches now take 2 cycles (lowers performance)*
2. *Simple, only some branches take 2 cycles (better performance)*
3. *Complex, very few branches take 2 cycles (best performance)*

** MIPS defines "branch delay slot", RISC-V doesn't*

Control Hazard

Predict "not taken"

Branch address ready at end of X stage:

- *If branch "not taken", do nothing.*
- *If branch "taken", then kill instruction in I stage (about to enter X stage) and fetch at new target address (PC)*

bneq $1, $1, L1	**I**	**X**	**M**		
add $5, $3, $4		**I**	**X**	**M**	
add $6, $1, $2			**I**	**X**	**M**
L1: sub $7, $6, $5				**I**	**X**

beq $1, $1, L1	**I**	**X**	**M**		
add $5, $3, $4		**I**	**nop**	**nop**	
L1: sub $7, $6, $5			**I**	**X**	**M**

EECS151 Project CPU Pipelining Summary

3-stage pipeline	**I** *instruction fetch*	**X** *execute*	**M** *access data memory*

- Pipeline rules:
  - Writes/reads to/from DMem are clocked on the leading edge of the clock in the "M" stage
  - Writes to RegFile use trailing edge of the clock of "M" stage
    - reg-file writes are 180 degrees out of phase
  - Instruction Decode and Register File access is up to you.
- Branch: predict "not-taken"
- Load: 1 cycle delay/stall
- Bypass ALU for data hazards
- More details in upcoming spec