# CSCI-2500: Computer Organization

Processor Design

# Datapath

- The *datapath* is the interconnection of the components that make up the processor.
- The datapath must provide connections for moving bits between memory, registers and the ALU.

# Control

- The control is a collection of signals that enable/disable the inputs/outputs of the various components.
- You can think of the control as the brain, and the datapath as the body.
  - the datapath does only what the brain tells it to do.

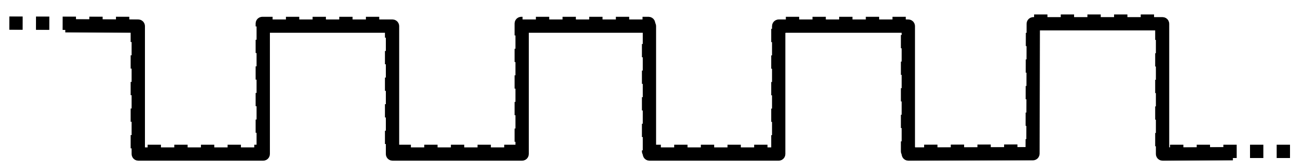
# Processor Design

The *sequencing* and *execution* of instructions

- We already know about many of the individual components that are necessary:
  - ALU, Multiplexors, Decoders, Flip-Flops
- We need to discuss how to use a *clock*
- We need to think about registers and memory.

# The Clock

*The clock* generates a never-ending sequence of alternating 1s and 0s.



All operations are synchronized to the clock.

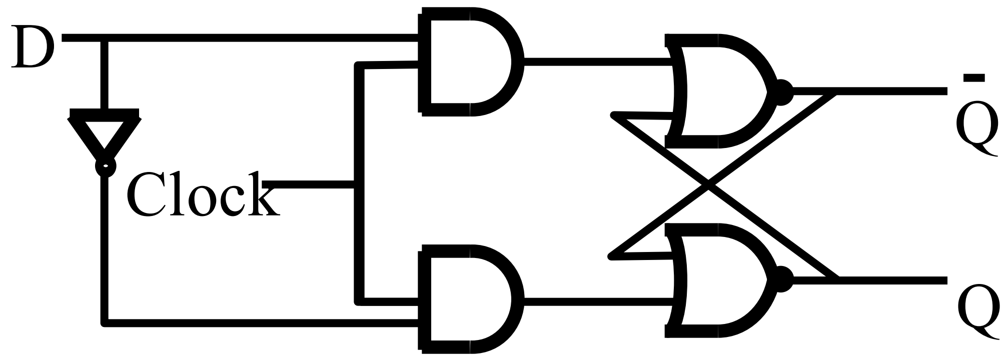
# Clocking Methodology

- Determines when (relative to the clock) a *signal* can be read and written.
  - Read: signal value is used by some component.
  - Written: a signal value is generated by some component.

# Simple Example: Enabled AND

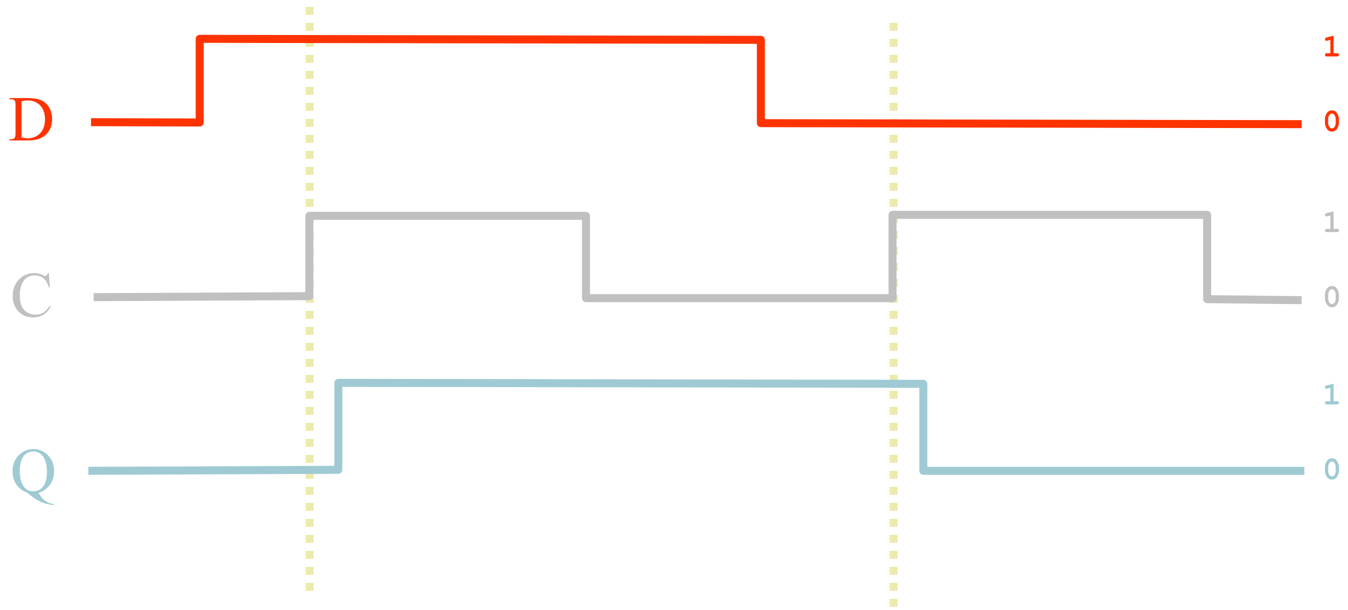
- We want an AND gate that holds it's output value constant until the clock switches from 0 (lo) to 1 (hi).
- We can use a flip-flop to hold the inputs to the AND gate constant during the time we want the output constant.
- We use a clocked flip-flop to make things happen when the clock changes.

# D Flip-Flop Reminder

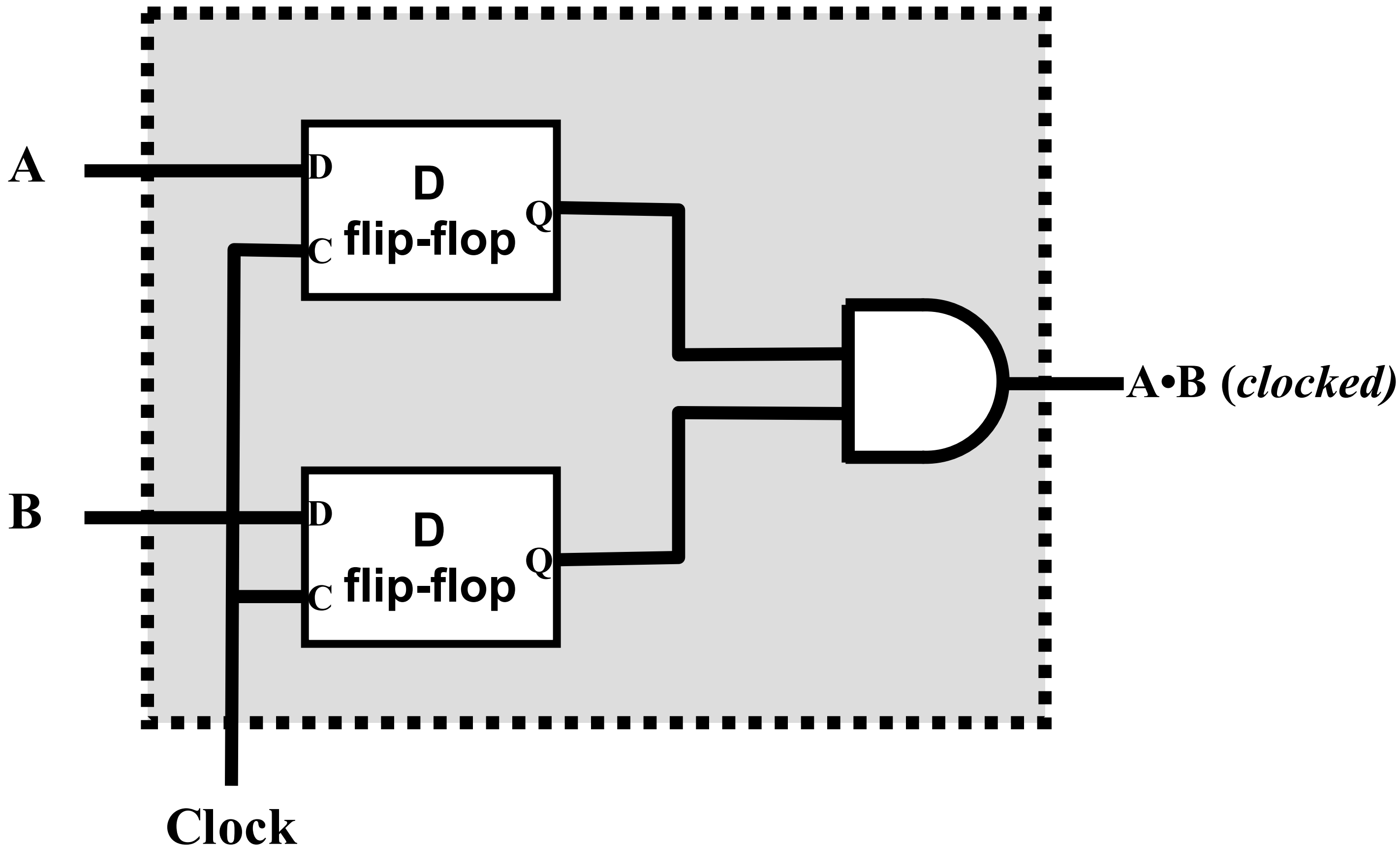



The output (Q) changes to reflect D only when the Clock is a 1.

# D Flip-Flop Timing



# Clocked AND gate



# Edge-triggered Clocking

- Values stored are updated (can change) only on a clock edge.
  - When the clock switches from 0 to 1 everybody allows signals in.
    - *everybody* means *state elements*
    - combinational elements always do the same thing, they don't care about the clock (that's why we added the flip-flops to our AND gate).

# State Elements

- Any component that *stores* one or more values is a *state element.*
  - The entire processor can be viewed as a circuit that moves from one state (collection of all the state elements) to another state.
  - At time *i* a component uses values generated at time *i-1*.

# Register File

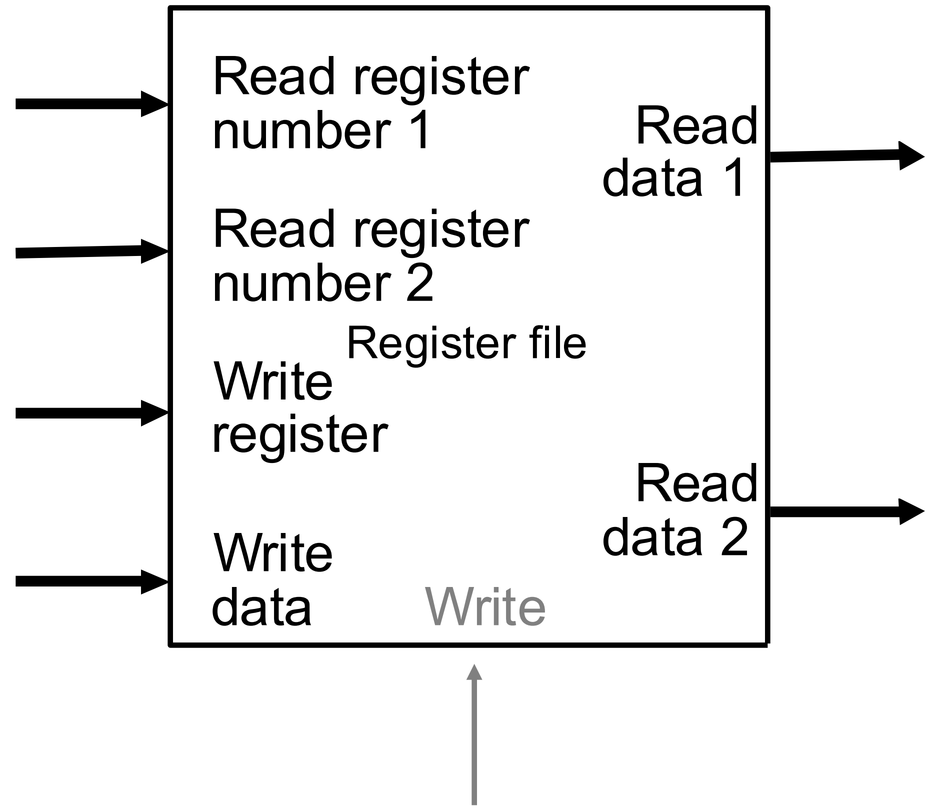
Contains multiple registers

- each holds 32 bits

- Two output ports (read ports)

- One input port (write port)

- To change the value of a register:
  - supply register number
  - supply data
  - clock (the Write control signal)



# Implementation of *Read Ports*

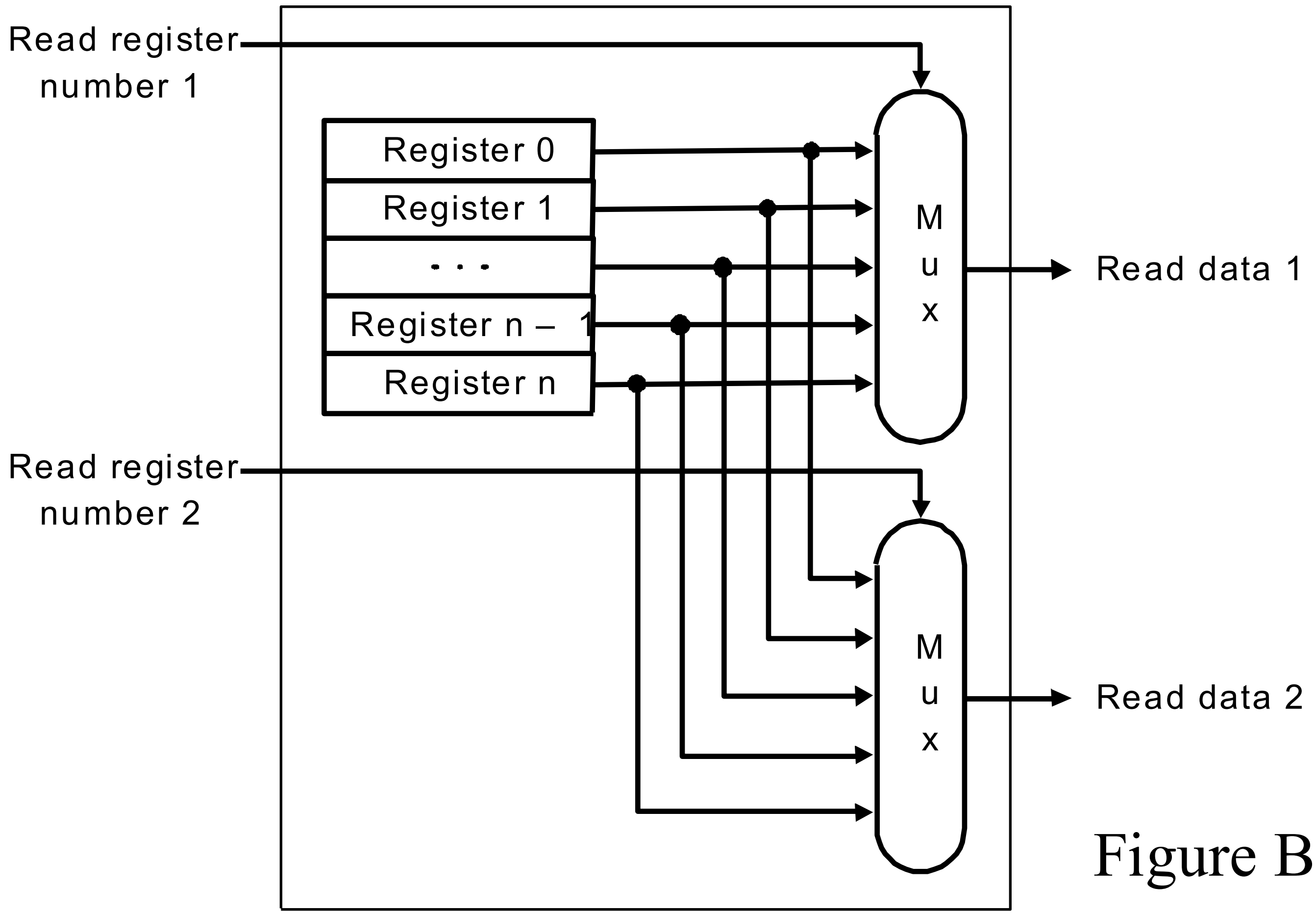
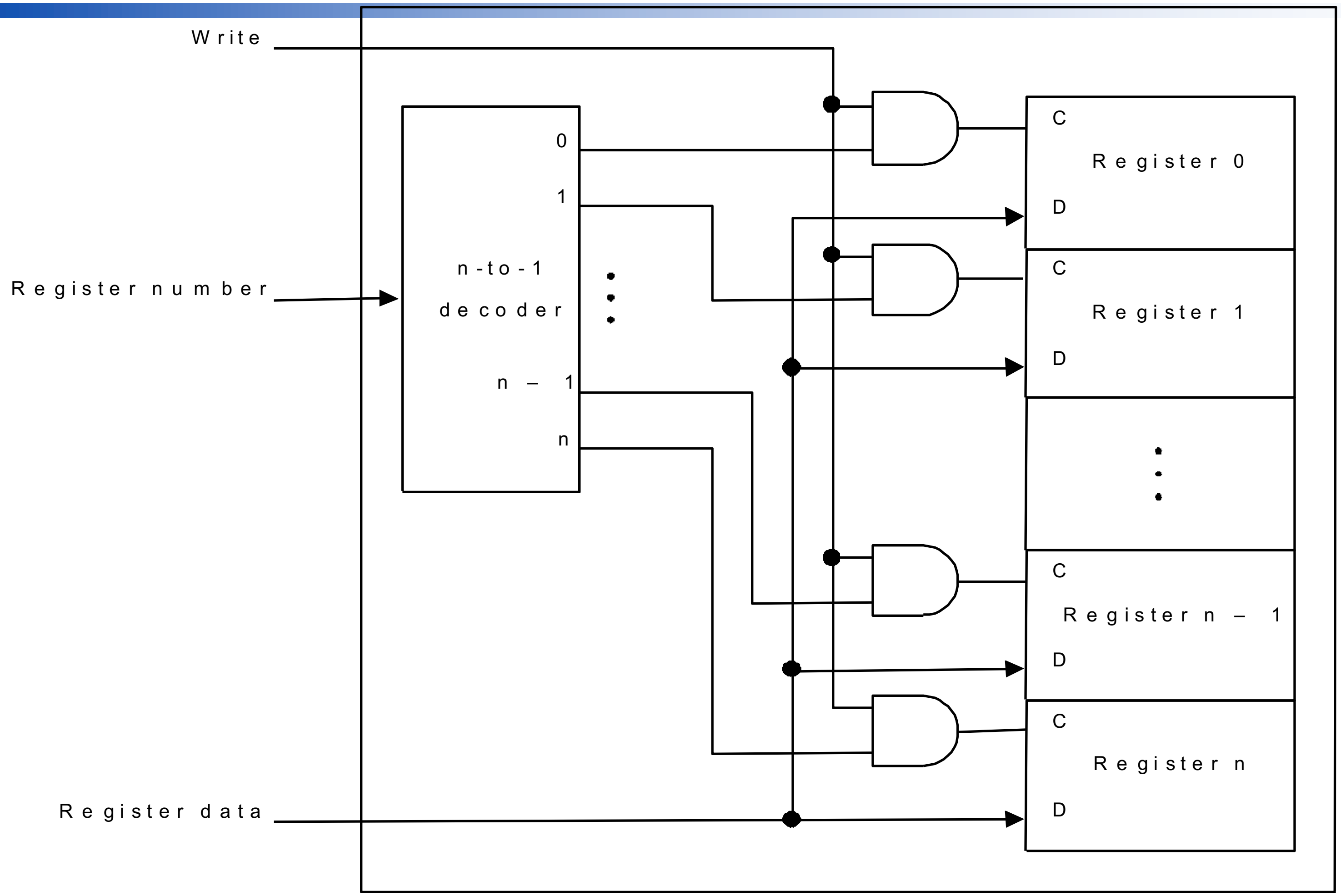



Figure B.19

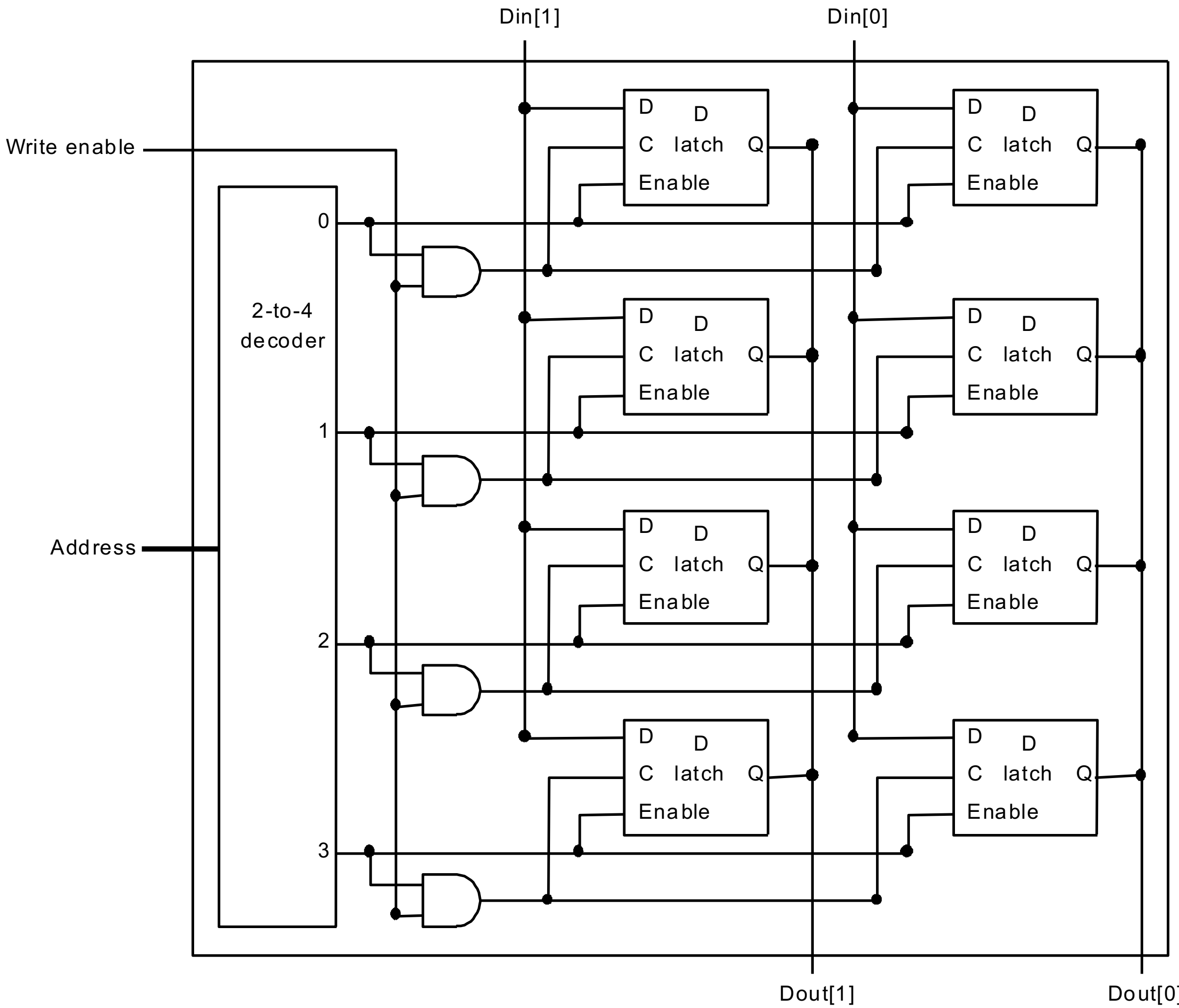
# Implementation of *Write*



# Memory

- Memory is similar to a very large register file:
  - single read port (output)
  - *chip select* input signal
  - *output enable* input signal
  - *write enable* input signal
  - address lines (determine which memory element)
  - data input lines (used to write a memory element)

# 4 x 2 Memory (SRAM)



# Memory Usage

- For now, we treat memory as a single component that supports 2 operations:
  - write (we change the value stored in a memory location)
  - read (we get the value currently stored in a memory location).
- We can only do one operation at a time!

# Instruction & Data Memory

- It is useful to treat the memory that holds instructions as a separate component.
  - instruction memory is *read-only*
- Typically there is really one memory that holds both instructions and data.
  - as we will see when we talk more about memory, the processor often has two interfaces to the memory, one for instructions and one for data!

# Designing a Datapath for MIPS

- We start by looking at the datapaths needed to support a simple subset of MIPS instructions:
  - a few arithmetic and logical instructions
  - load and store word
  - `beq` and `j` instructions

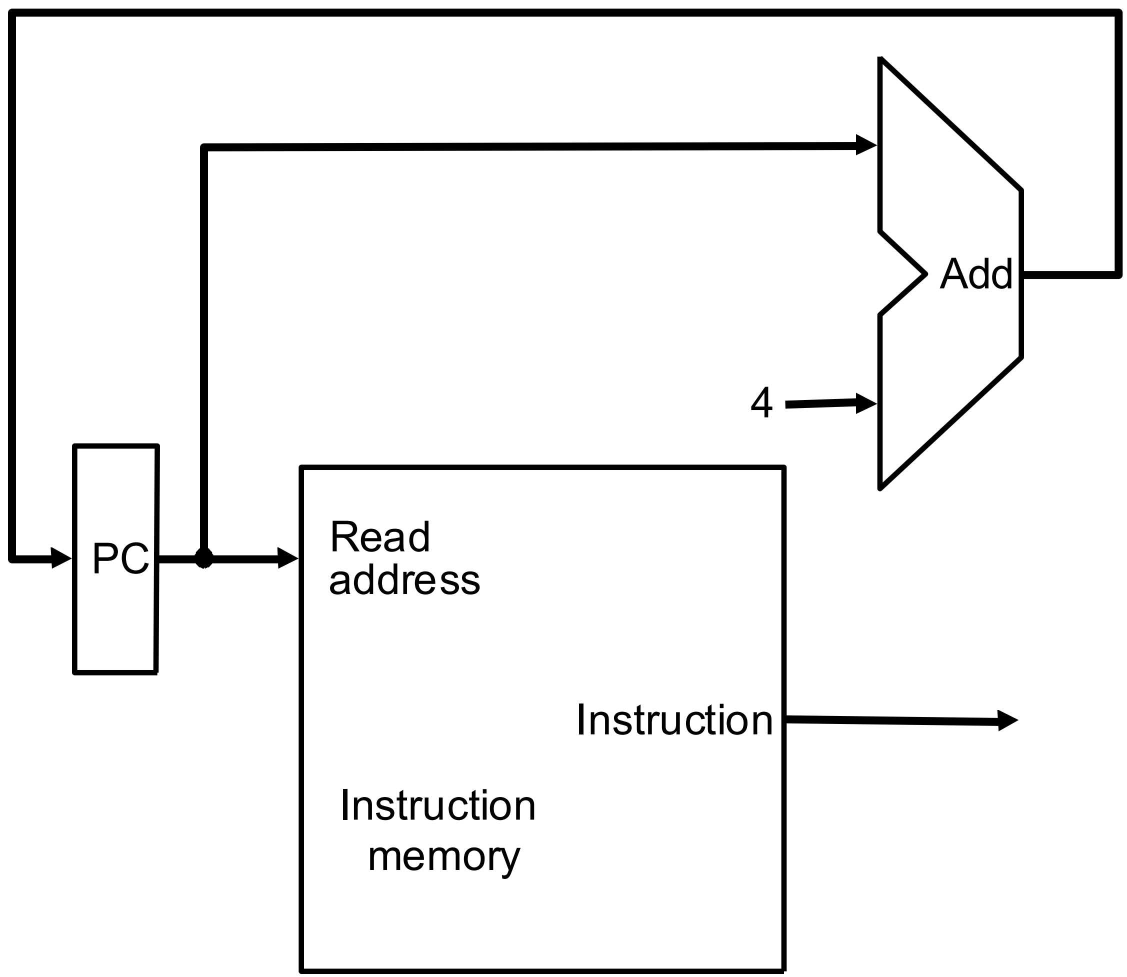
# Functions for MIPS Instructions

- We can generalize the functions we need to:
  - using the PC register as the address, read a value from the memory (read the instruction)
  - Read one or two register values (depends on the specific instruction).
  - ALU Operation , Memory read or write, …
  - Possibly change the value of a register.

# Fetching the next instruction

- PC Register holds the address
- Memory holds the instruction
  - we need to *read* from memory.
- Need to update the PC
  - add 4 to current value

# Instruction Fetch DataPath



# Supporting R-format instructions

| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
|---|---|---|---|---|---|
| `op` | `rs` | `rt` | `rd` | `shamt` | `funct` |

Includes **`add`**, **`sub`**, **`slt`**, **`and`** & **`or`** instructions.

Generalization:

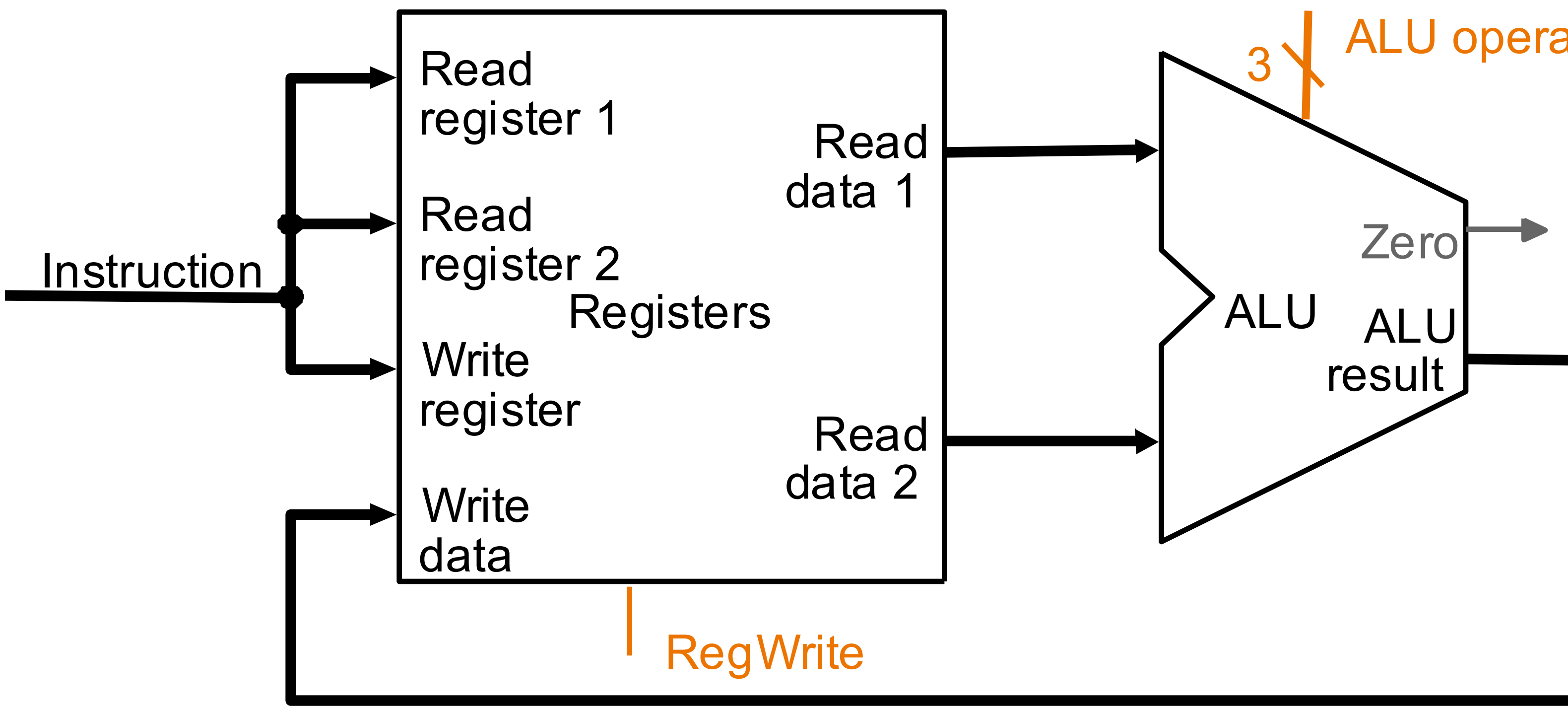
- read 2 registers and send to ALU.
- perform ALU operation
- store result in a register

# MIPS Registers

- MIPS has 32 general purpose registers.
- Register File holds all 32 registers
  - need 5 bits to select a register
  - `rs`, `rt` & `rd` fields in R-format instructions.

- MIPS Register File has 2 *read ports*.
  - can get at both *source* registers at the same time.

# Datapath for R-format Instructions



# Load and Store Instructions

Need to compute the address

- offset (part of the instruction)
- base (stored in a register).

For Load:

- read from memory
- store in a register

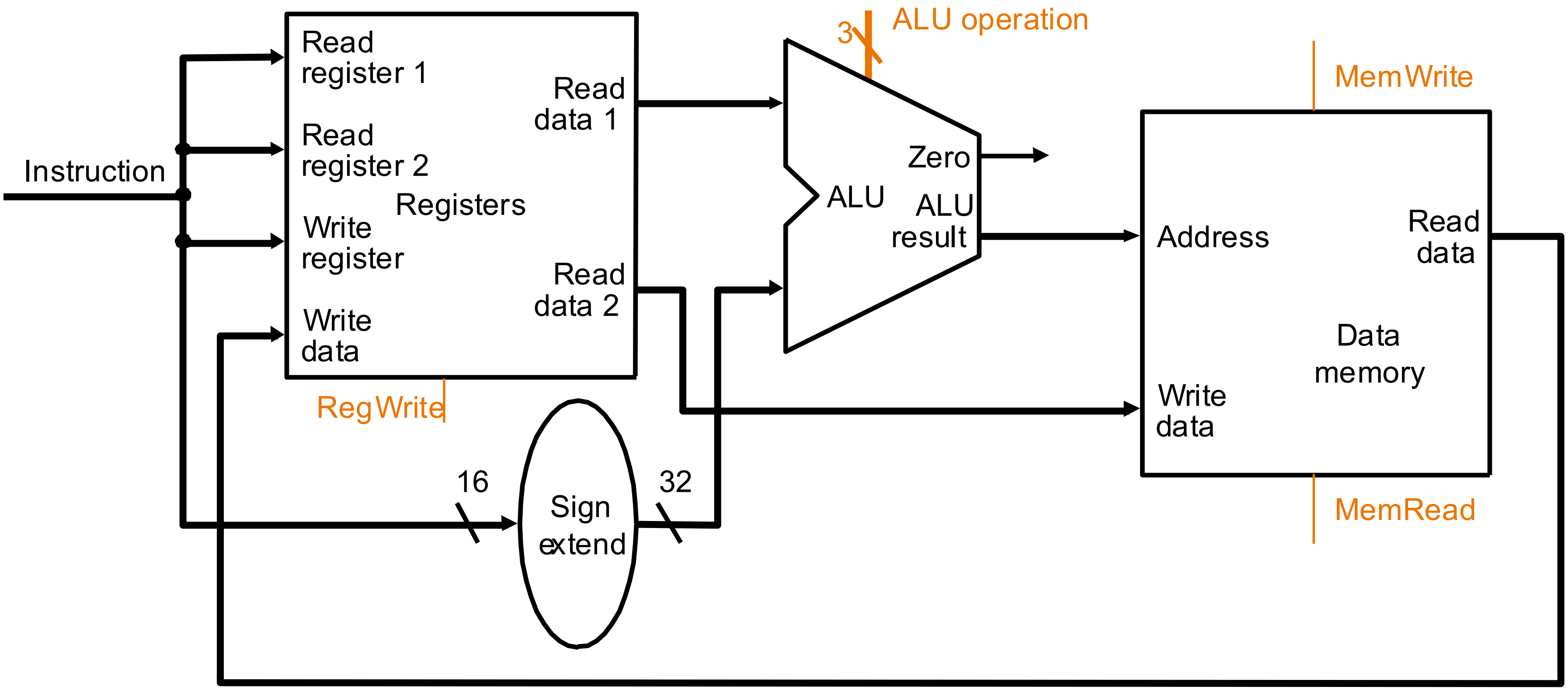
For Store:

- read from register
- write to memory

# Computing the address

- 16 bit signed offset is part of the instruction.
- We have a 32 bit ALU.
  - need to sign extend the offset (to 32 bits).
- Feed the 32 bit offset and the contents of a register to the ALU
- Tell the ALU to "add".

# Load/Store Datapath



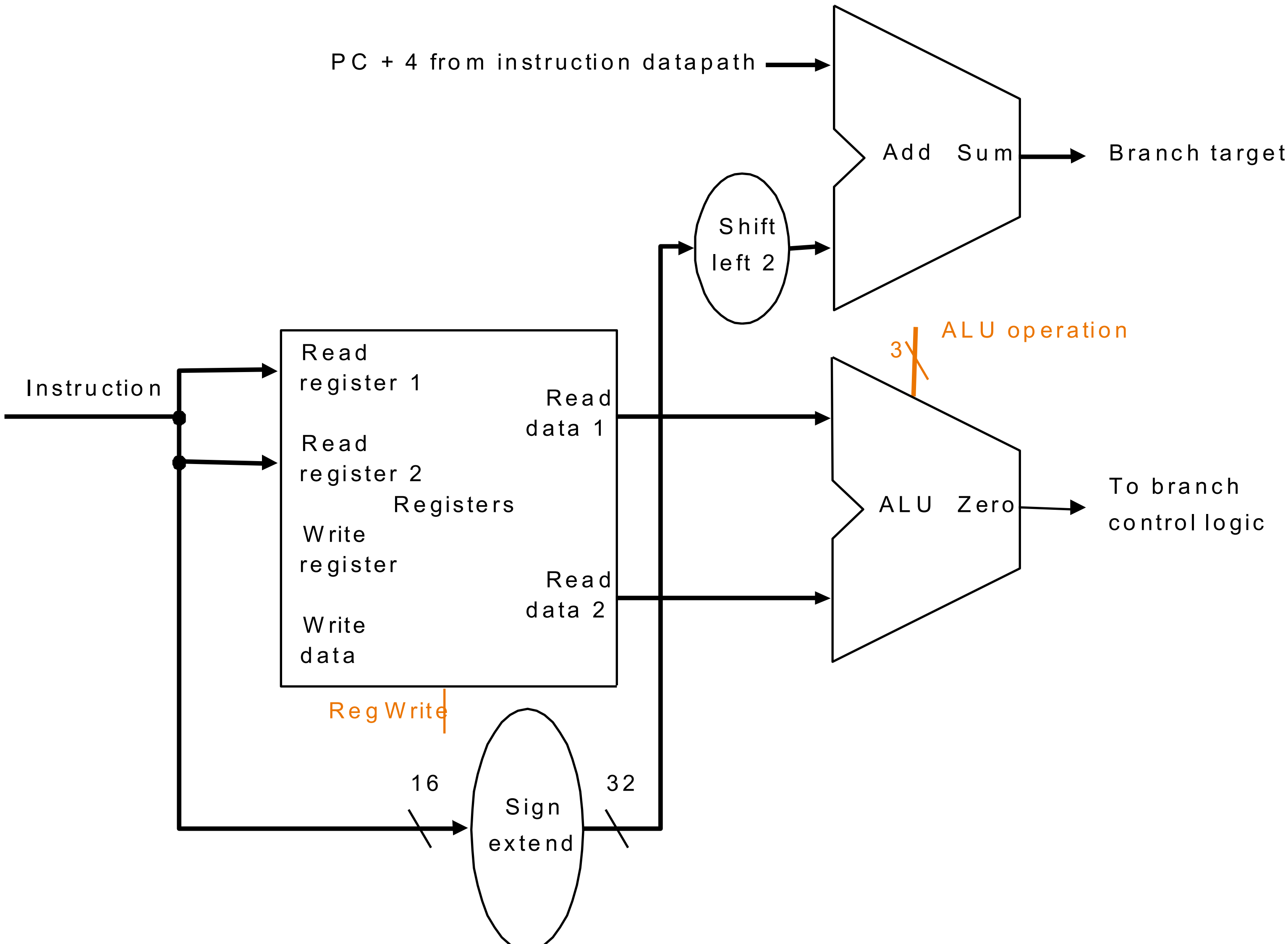
# Supporting `beq`

- 2 registers compared for equality
- 16 bit offset used to compute target address.
  - signed offset is relative to the PC
  - offset is in *words* not in bytes!
- Might branch, might not (need to decide).

# Computing target address

- Recall that the offset is actually relative to the address of the next instruction.
  - we *always* add 4 to the PC, we must make sure we use this value as the base.
- Word vs. Byte offset
  - we just need to shift the 16 bit offset 2 bits to the right (fill with 2 zeros).

# Branch Datapath



# Control & DataPath

Ref: Chapter 4

# Datapath

- The *datapath* is the interconnection of the components that make up the processor.

- The datapath must provide connections for moving bits between memory, registers and the ALU.

# Control

- The control is a collection of signals that enable/disable the inputs/outputs of the various components.
- You can think of the control as the brain, and the datapath as the body.
  - the datapath does only what the brain tells it to do.

# Datapaths

We looked at individual datapaths that support:

1. Fetching Instructions
2. Arithmetic/Logical Instructions
3. Load & Store Instructions
4. Conditional branch

We need to combine these in to a single datapath.

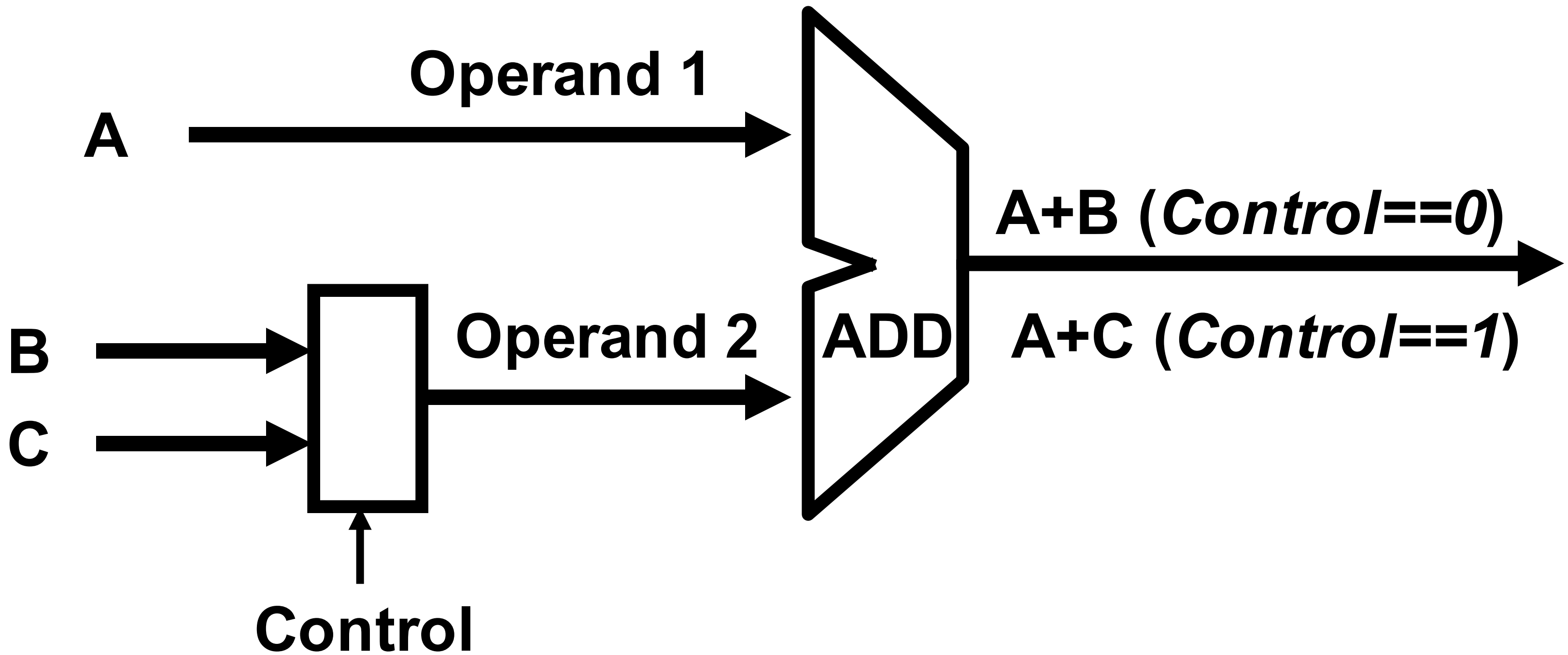
# Issues

- When designing one datapath that can be used for any operation:
  - the goal is to be able to handle one instruction per cycle.
  - must make sure no datapath *resource* needs to be used more than once at the same time.
    - if so - we need to provide more than one!

# Sharing Resources

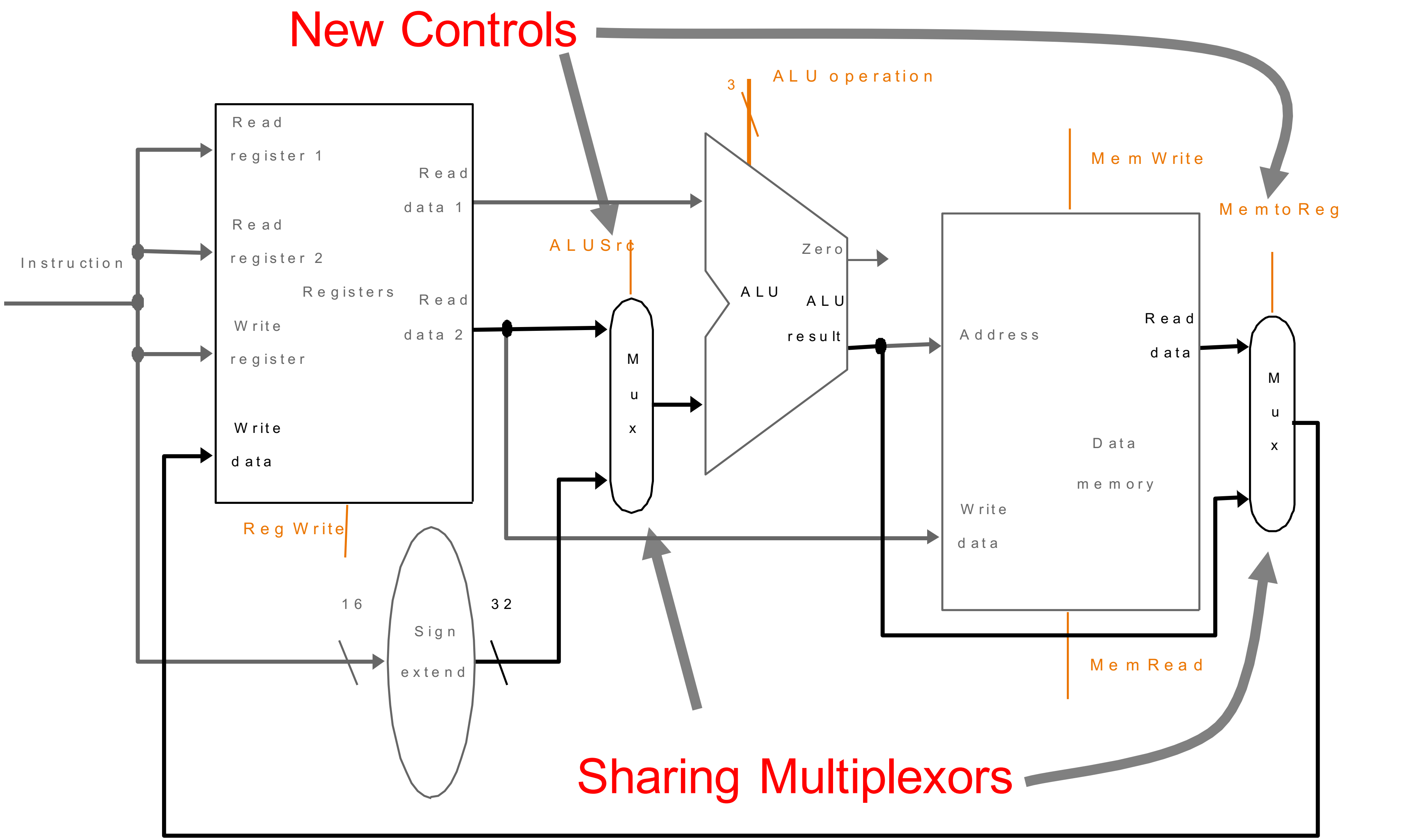
- We can share datapath resources by adding a multiplexor (and a control line).
  - for example, the second input to the ALU could come from either:
    - a register (as in an arithmetic instruction)
    - from the instruction (as in a load/store – when computing the memory address).

# Sharing with a Multiplexor Example



# Combining Datapaths for memory instructions and arithmetic instructions

- Need to share the ALU
  - For memory instructions used to compute the address in memory.
  - For Arithmetic/Logical instructions used to perform arithmetic/logical operation.

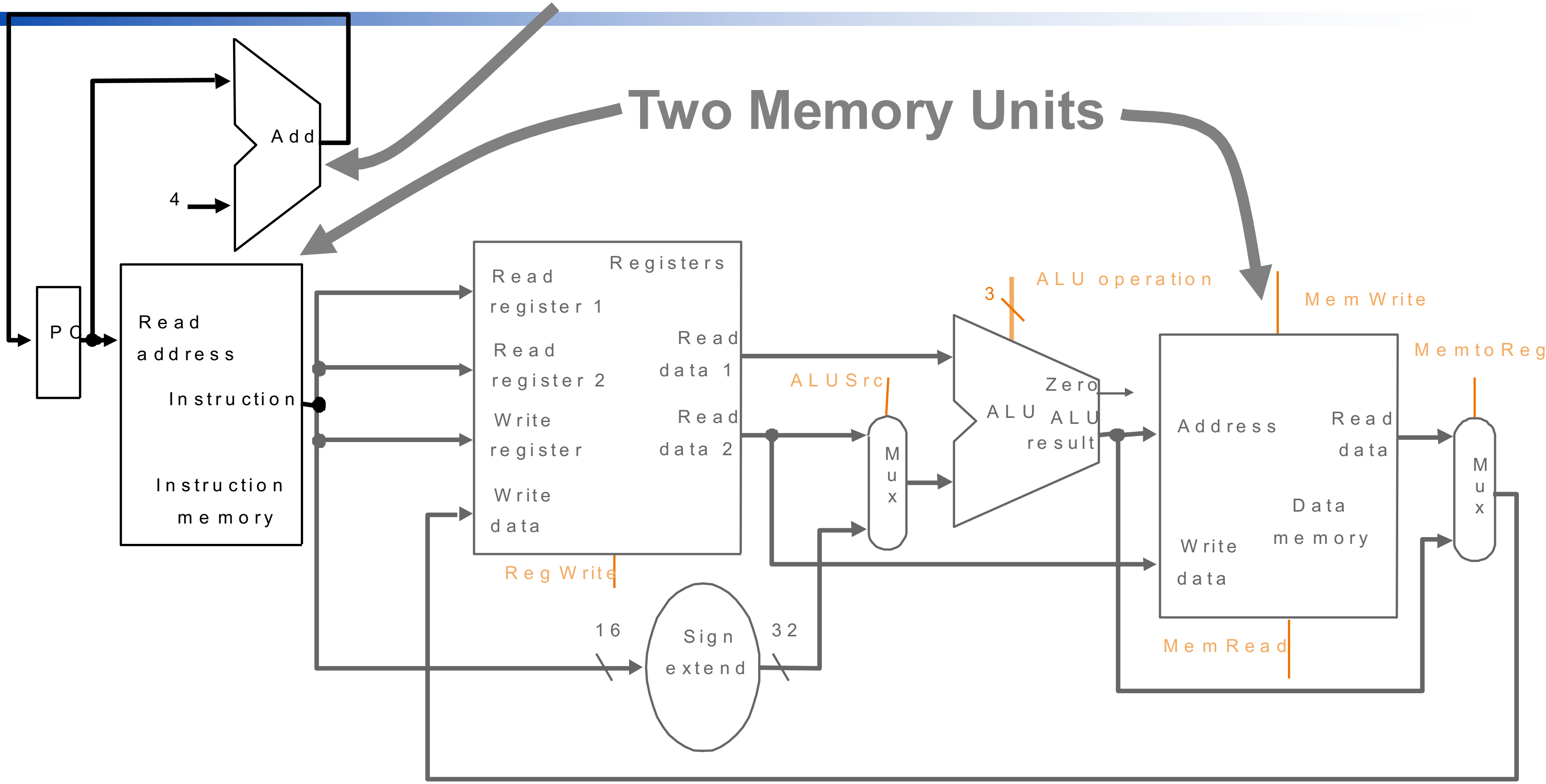

New Controls
ALU operation
3
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Instruction
RegWrite
ALUSrc
M u x
Zero
ALU
ALU result
Address
Read data
Data memory
Write data
MemWrite
MemRead
MemtoReg
M u x
16
Sign extend
32
Sharing Multiplexors

# Adding the Instruction Fetch

- One memory for instructions, separate memory for data.
  - otherwise we might need to use the memory twice in the same instruction.
- Dedicated Adder for updating the PC
  - otherwise we might need to use the ALU twice in the same instruction.

# Dedicated Adder

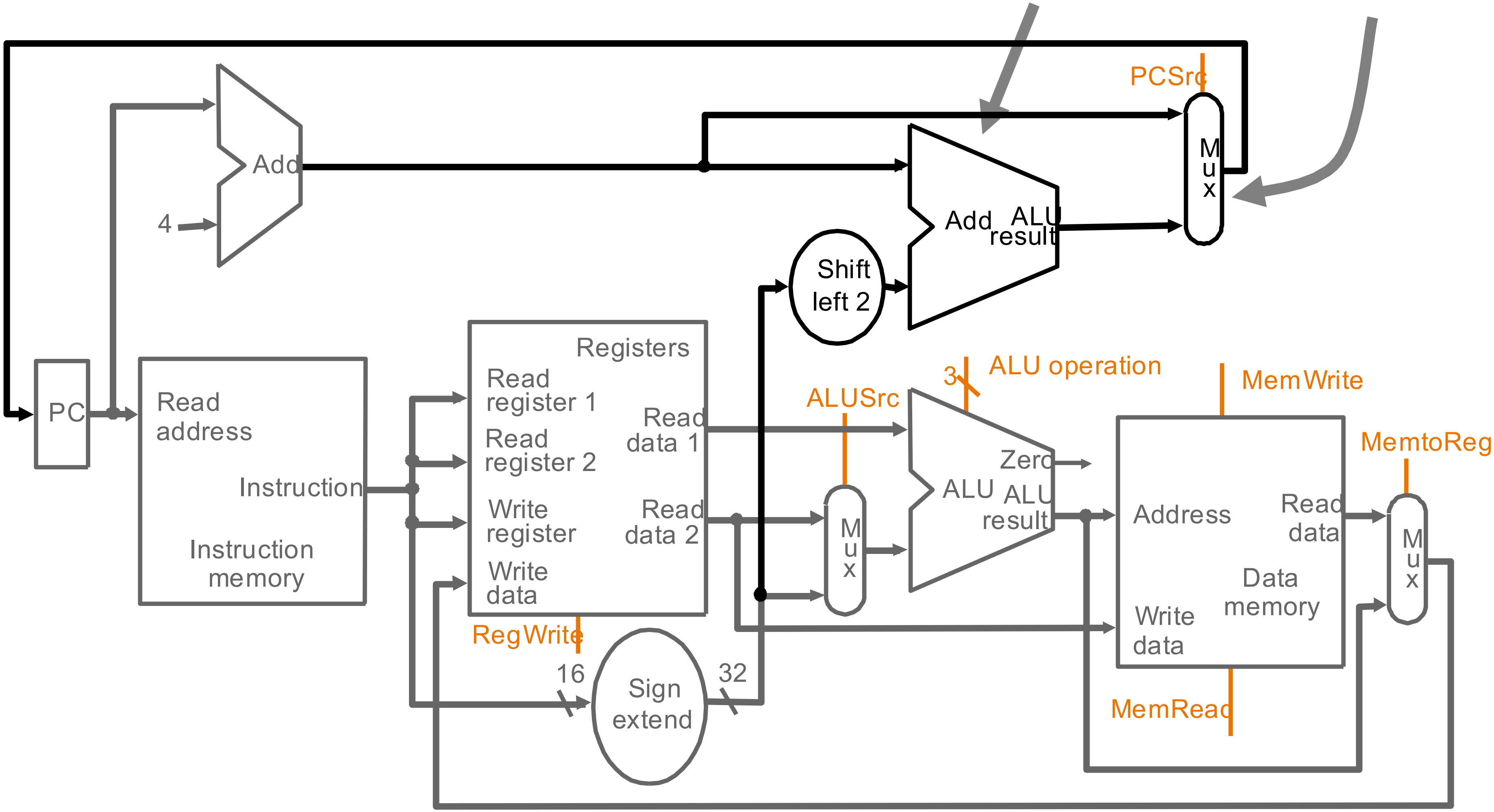
# Two Memory Units



# Need to add datapath for `beq`

- Register comparison (requires ALU).
- Another adder to compute target address.
  - One input to adder is sign extended offset, shifted by 2 bits.
  - Other input to adder is PC+4

## New adder and mux



# Whew!

- Keep in mind that the datapath we now have supports just a few MIPS instructions!

- Things get worse (more complex) as we support other instructions:

  `j` `jal` `jr` `addi`

- We won't worry about them now…

# Control Unit

- We need something that can generate the *controls* in the datapath.
- Depending on what kind of instruction we are executing, different controls should be turned on (*asserted*) and off (*deasserted*).
- We need to treat each control individually (as a separate boolean function).

# Controls

- Our datapath includes a bunch of *controls*:
  - ***ALU operation*** (3 bits)
  - ***RegWrite***
  - ***ALUSrc***
  - ***MemWrite***
  - ***MemtoReg***
  - ***MemRead***
  - ***PCSrc***

# ALU Operation Control

- A 3 bit control (assumes the ALU designed in chapter 4):

| ALU Control Input | Operation |
|---|---|
| 000 | AND |
| 001 | OR |
| 010 | add |
| 110 | subtract |
| 111 | slt |

# ALU Functions for other instructions

**`lw`** , **`sw`** (load/store): addition

**`beq`**: subtraction

**`add`**, **`sub`**, **`and`**, **`or`**, **`slt`** (arithmetic/logical): All R-format instructions

# R-Format Instructions

| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
|---|---|---|---|---|---|
| `op` | `rs` | `rt` | `rd` | `shamt` | `funct` |

Operation is specified by some bits in the **`funct`** field in the instruction.

# MIPS Instruction OPCODEs

| ← 6 bits → | |
|---|---|
| **op** | *varies depending on instruction* |

- The MS 6 bits are an OPCODE that identifies the instruction.
- R-Format: always 000000
  - (funct identifies the operation)

| lw | sw | beq |
|---|---|---|
| 100011 | 101011 | 000100 |

# Generating ALU Controls

We can view the 3 bit ALU control as 3 boolean functions. Inputs are:

- the **op** field (OPCODE)
- **funct** field (for R-format instructions only)

# Simplifying The Opcode

- For building the ALU Operation Controls, we are interested in only 4 different opcodes.
- We can simplify things by first reducing the 6 bit op field to a 2 bit value we will call **`ALUOp`**

| Instruction | ALUOp | funct | ALU action | ALU controls |
|---|---|---|---|---|
| lw | 00 | ?????? | add | 010 |
| sw | 00 | ?????? | add | 010 |
| beq | 01 | ?????? | subtract | 110 |
| add | 10 | 100000 | add | 010 |
| sub | 10 | 100010 | subtract | 110 |
| and | 10 | 100100 | and | 000 |
| or | 10 | 100101 | or | 001 |
| slt | 10 | 101010 | slt | 111 |

# Build a Truth Table

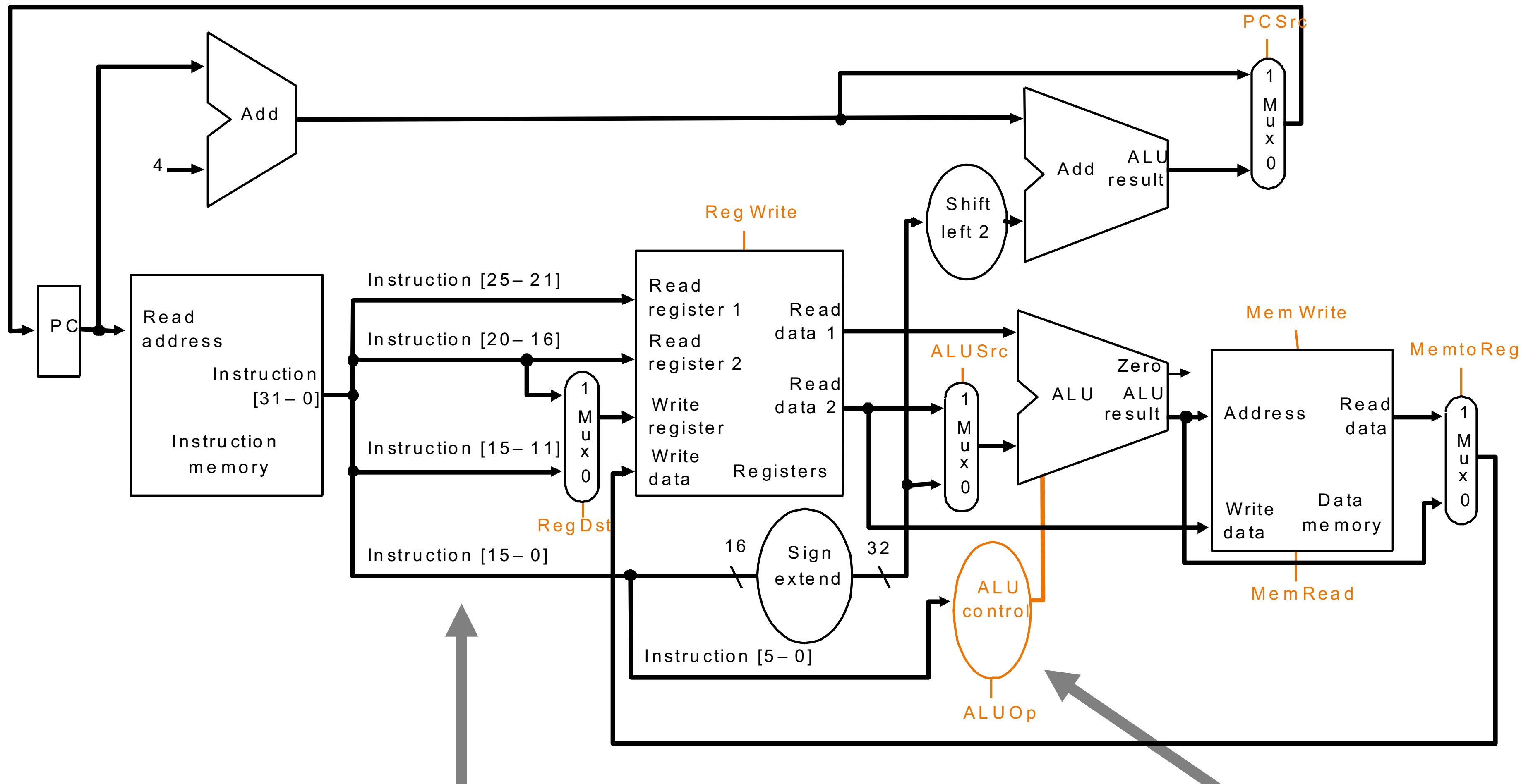
- We can now build a truth table for the 3 bit ALU control.
- Inputs are:
  - 2 bit `ALUOp`
  - 6 bit `funct` field
- Abbreviated Truth Table: only show the rows we care about!

| ALUOp | | funct | | | | | | ALU Control |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | x | x | x | x | x | x | 010 |
| x | 1 | x | x | x | x | x | x | 110 |
| 1 | x | x | x | 0 | 0 | 0 | 0 | 010 |
| 1 | x | x | x | 0 | 0 | 1 | 0 | 110 |
| 1 | x | x | x | 0 | 1 | 0 | 0 | 000 |
| 1 | x | x | x | 0 | 1 | 0 | 1 | 001 |
| 1 | x | x | x | 1 | 0 | 1 | 0 | 111 |

**x** means "don't care"

# Adding the ALU Control

- We can now add the ALU control to the datapath:
  - inputs to this control come from the instruction and from ALUOp
- If we try to show all the details the picture becomes too complex:
  - just plop in an "ALU Control" box.




Shows which bits from the instruction are fed to register file inputs

ALU Control

# Implementing Other Controls

- The other controls in out datapath must also be specified as functions.
- We need to determine the inputs to all the functions.
  - primarily the inputs are part of the instructions, but there are exceptions.
- Need  to define precisely what conditions should turn on each control.

# `RegDst` Control Line

- Controls a multiplexor that selects on of the fields **`rt`** or **`rd`** from an R-format or I-format instruction.
- I-Format is used for load and store.
- **`sw`** needs to *write* to the register **`rt`**.

| | | | | | | |
|---|---|---|---|---|---|---|
| *I-format* | op | rs | rt | address | | |
| *R-format* | op | rs | rt | rd | shamt | funct |

# RegDst usage

- **RegDst** should be
  - 0 to send **rt** to the write register # input.
  - 1 to send **rd** to the write register # input.

- **RegDst** is a function of the opcode field:
  - If instruction is **sw**, **RegDst** should be 0
  - For all other instructions **RegDst** should be **1**

# RegWrite **Control**

- a 1 tells the register file to write a register.
  - whatever register is specified by the write register # input is written with the data on the write register data inputs.
- Should be a 1 for arithmetic/logical instructions and for a store.
- Should be a 0 for load or **beq**.

# `ALUSrc` Control

- MUX that selects the source for the second ALU operand.
  - 1 means select the second register file output (read data 2).
  - 0 means select the sign-extended 16 bit offset (part of the instruction).
- Should be a 1 for load and store.
- Should be a 0 for everything else.

# `MemRead` Control

- A 1 tells the memory to put the contents of the memory location (specified by the address lines) on the Read data output.

- Should be a **`1`** for load.
- Should be a **`0`** for everything else.

# `MemWrite` Control

- 1 means that memory location (specified by memory address lines) should get the value specified on the memory Write Data input.

- Should be a **`1`** for store.
- Should be a **`0`** for everything else.

# `MemToReg` Control

- MUX that selects the value to be stored in a register (that goes to register write data input).
  - 1 means select the value coming from the memory data output.
  - 0 means select value coming from the ALU output.
- Should be a **`1`** for load and any arithmetic/logical instructions.
- Should be a **`0`** for everything else (**`sw`**, **`beq`**).

# PCSrc Control

- MUX that selects the source for the value written in to the PC register.
  - 1 means select the output of the Adder used to compute the relative address for a branch.
  - 0 means select the output of the PC+4 adder.

- Should be a `1` for beq if registers are equal!
- Should be a `0` for other instructions or if registers are different.

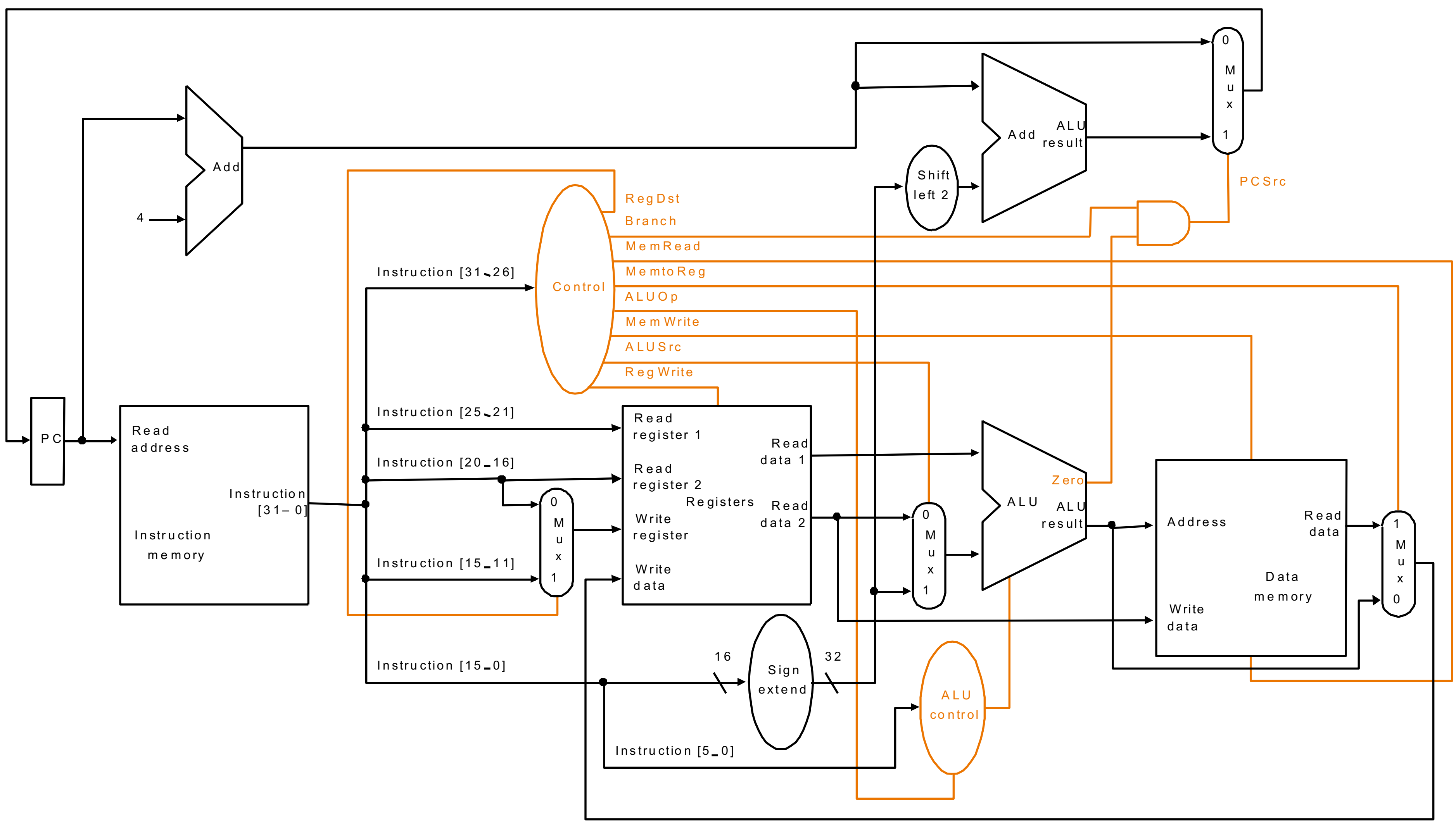
# PCSrc depends on result of ALU operation!

- This control line can't be simply a function of the instruction (all the others can).

- **PCSrc** should be a 1 only when:
  - **beq** AND **ALU zero** output is a 1
- We will generate a signal called "branch" that we can AND with the ALU zero output.

# Truth Table for Control

| Instruction | RegDst | ALUSrc | Memto-Reg | Reg Write | Mem Read | Mem Write | Branch | ALUOp |
|---|---|---|---|---|---|---|---|---|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 10 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 00 |
| sw | x | 1 | x | 0 | 0 | 1 | 0 | 00 |
| beq | x | 0 | x | 0 | 0 | 0 | 1 | 01 |



Add
4
Add
ALU
result
Shift
left 2
PCSrc
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
PC
Read
address
Instruction
[31–0]
Instruction
memory
Read
register 1
Read
register 2
Write
register
Write
data
Registers
Read
data 1
Read
data 2
ALU
Zero
ALU
result
ALU
control
16
Sign
extend
32
Address
Read
data
Write
data
Data
memory
0
M
u
x
1

# Single Cycle Instructions

- View the entire datapath as a combinational circuit.
- We can follow the *flow* of an instruction through the datapath.
  - single cycle instruction means that there are not really any *steps* – everything just happens and becomes finalized when the clock cycle is over.

# add $t1,$t2,$t3

- Control Lines:
  - ALU Controls specify an ALU add operation.
  - RegWrite will be a 1 so that when the clock cycle ends the value on the Register Write Input lines will be written to a register.
  - all other control lines are 0.

# `lw $t1,offset($t2)`

- Control Lines:
  - ALU Control set for an add operation.
  - ALUSrc is set to 1 to indicate the second operand is sign extended offset.
  - MemRead would be a 1.
  - RegDst would select the correct bits from the instruction to specify the dest. register.
  - RegWrite would be a 1.

# Disadvantage of single cycle operation

If we have instructions execute in a single cycle, then the cycle time must be long enough for the slowest instruction.

- all instructions take the same time as the slowest.

# Multicycle Implementation

- Chop up the processing of instructions in to discrete stages.
- Each stage takes one clock cycle.
  - we can implement each stage as a big combinational circuit (like we just did for the whole thing).
  - provide some way to sequence through the stages.

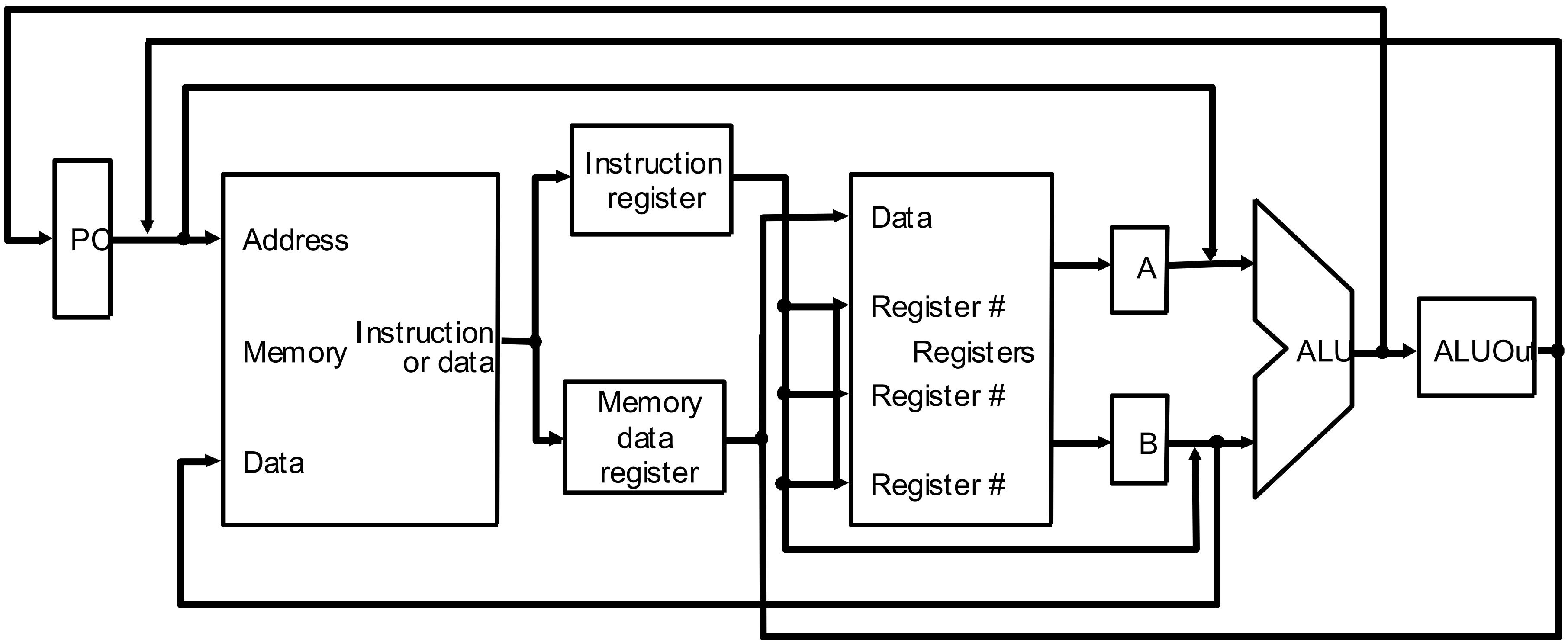
# Advantages of Multicycle

- Only need those stages required by an instruction.
  - the control unit is more complex, but instructions only take as long as necessary.
- We can share components
  - perhaps 2 different stages can use the same ALU.
  - We don't need to duplicate resources.

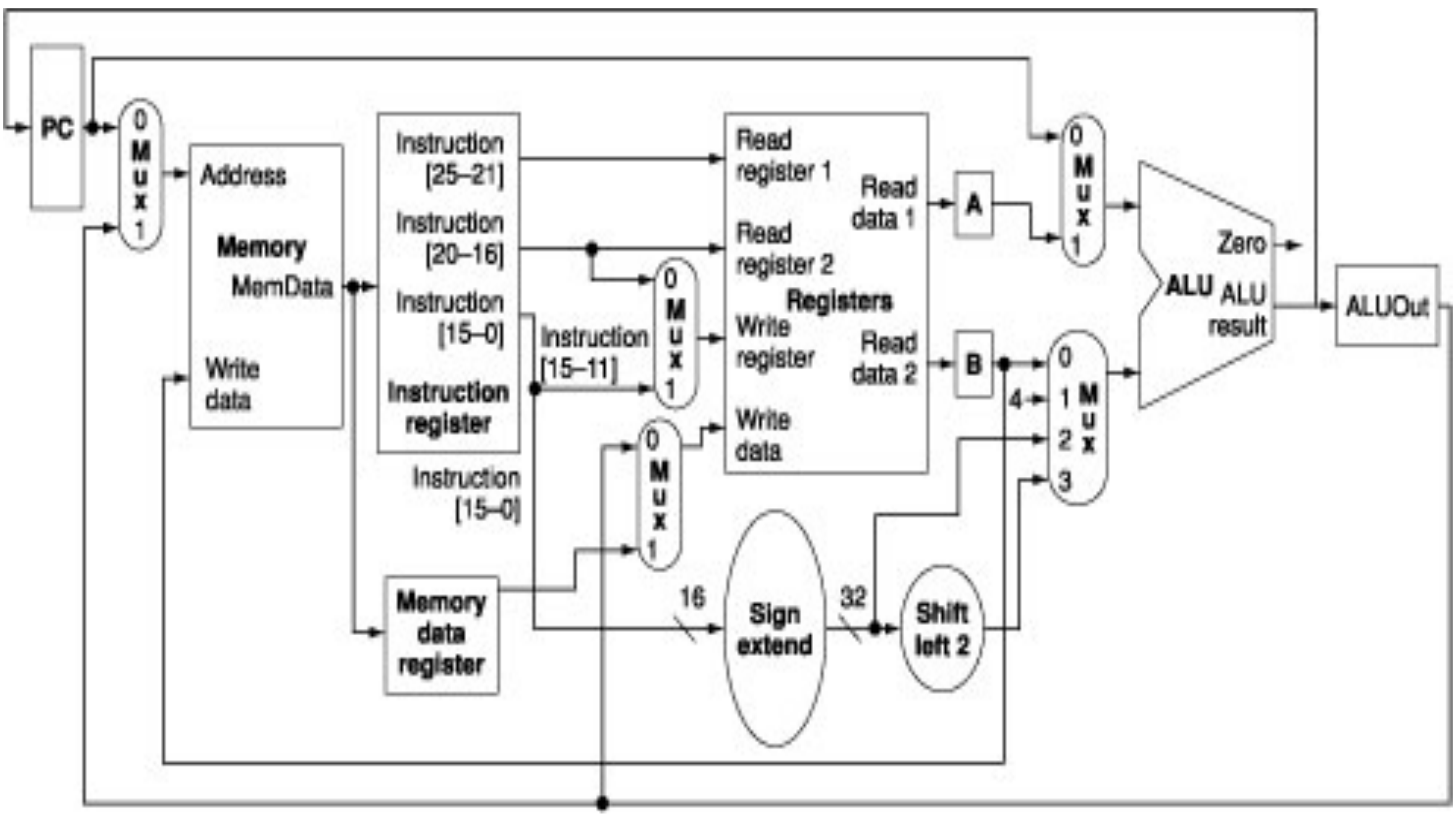
# Additional Resources for Multicycle

- To implement a multicycle implementation we need some additional registers that can be used to hold intermediate values.
  - instruction
  - computed address
  - result of ALU operation
  - ...

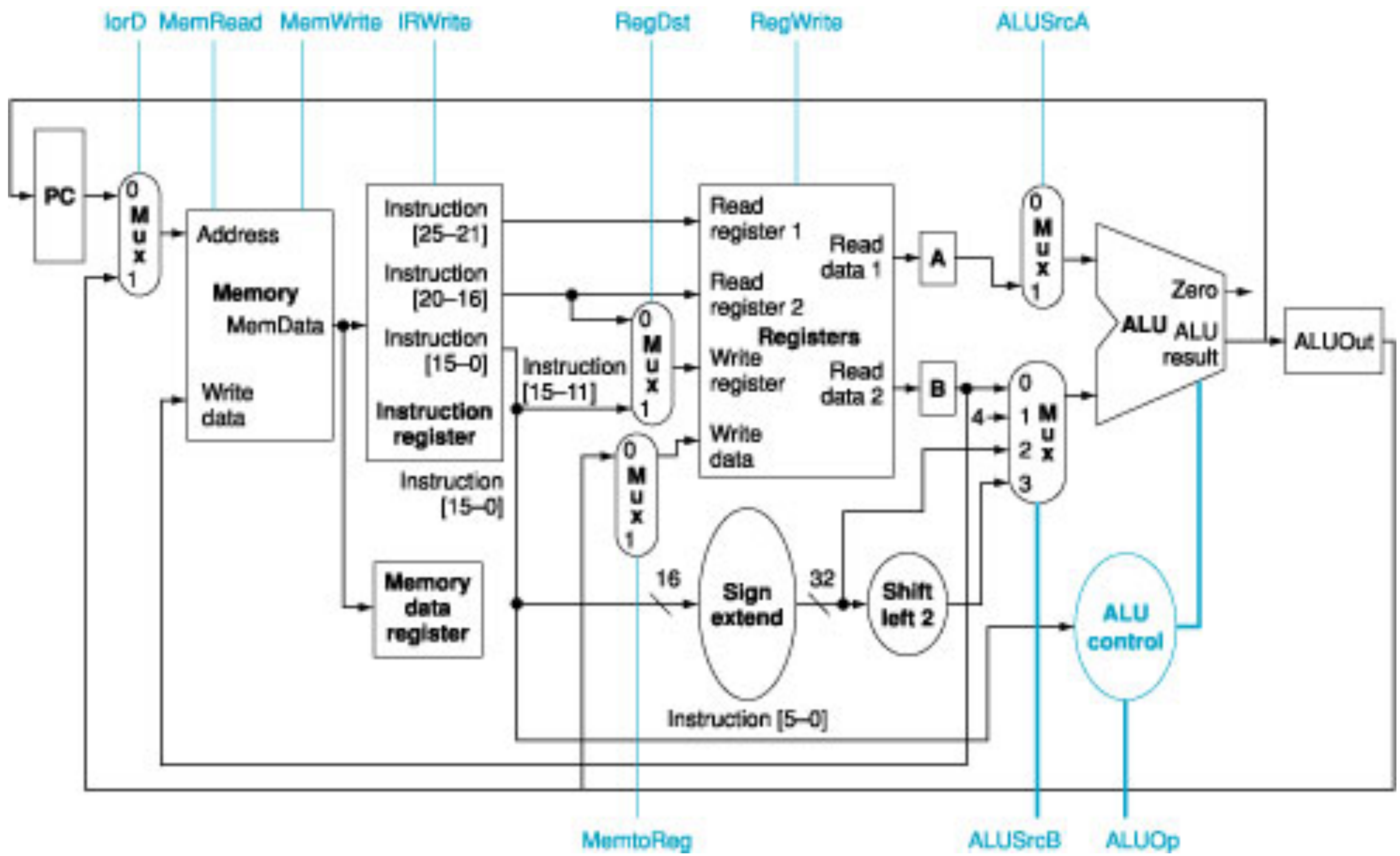
# Multicycle Datapath



# Multicycle Datapath for MIPS



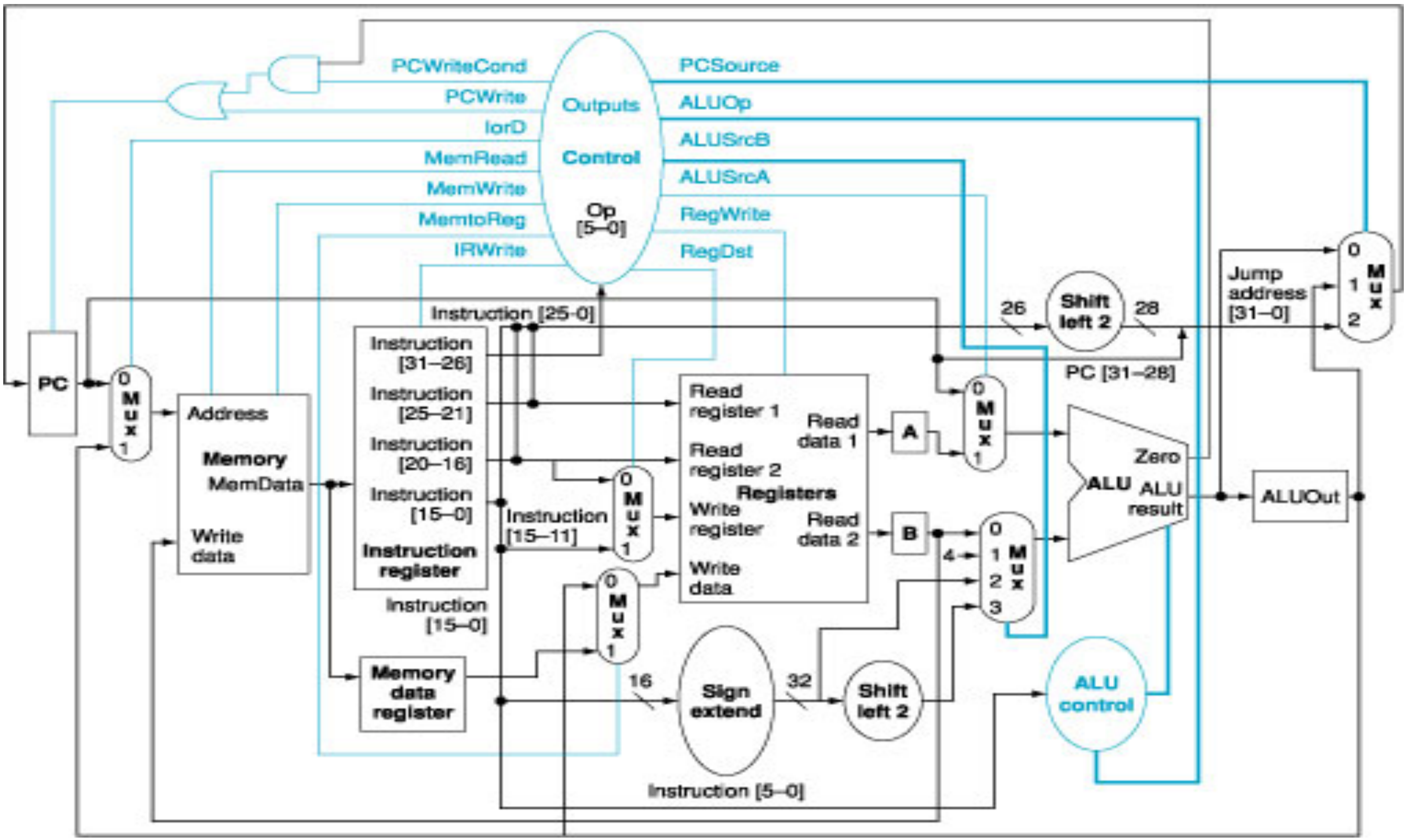
# MC DP with Control



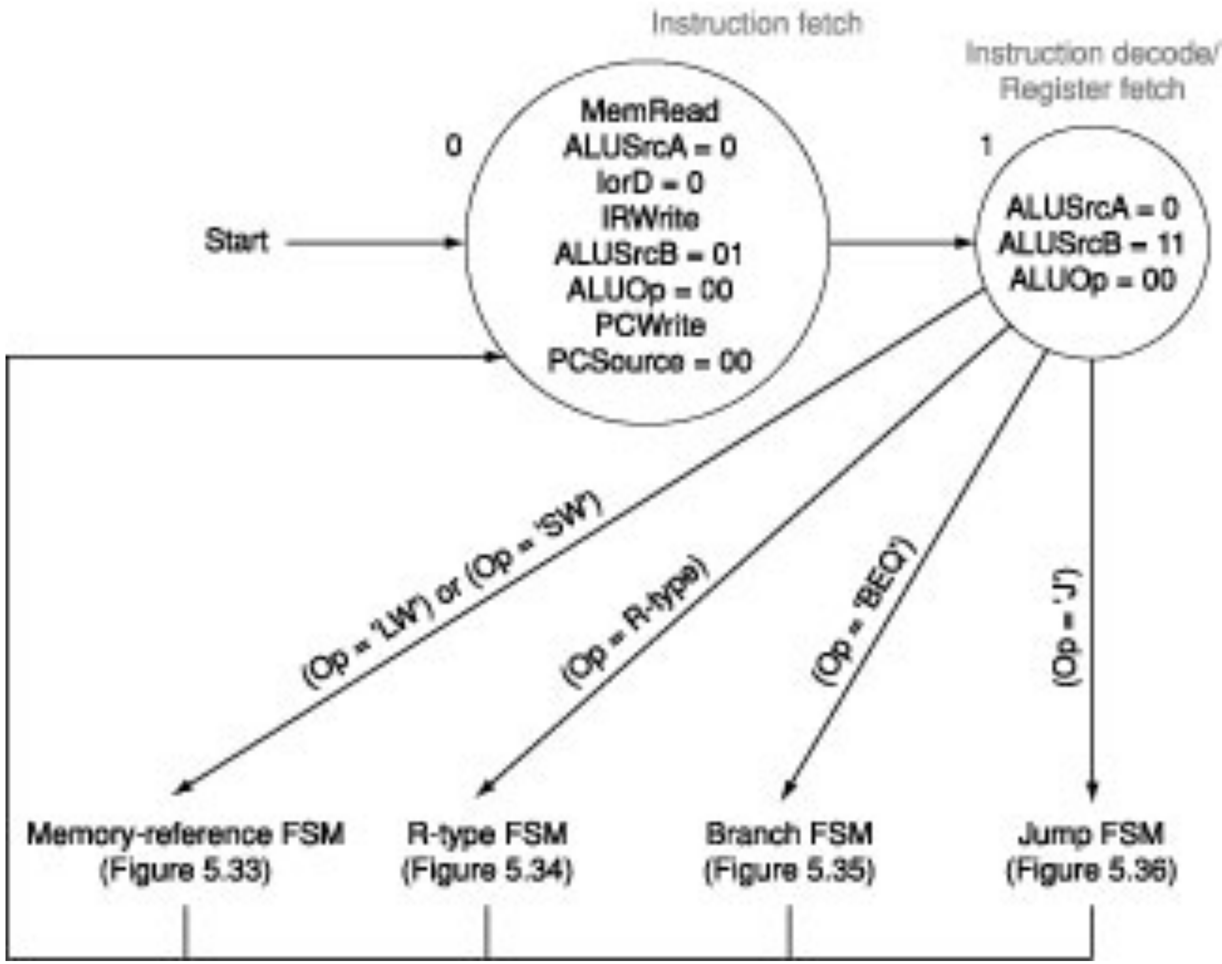
# Instruction Stages

- Instruction Fetch
- Instruction decode/register fetch
- ALU operation/address computation
- Memory Access
- Register Write

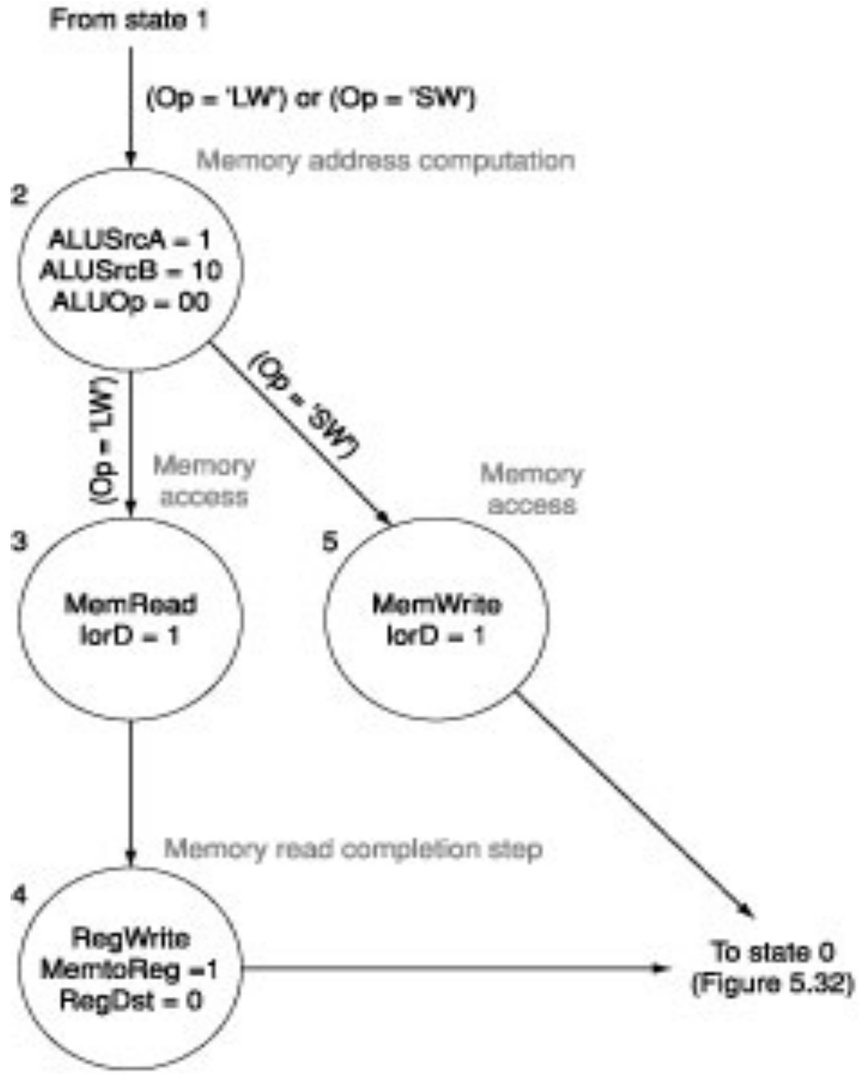
# Complete Multicyle Datapath & Control



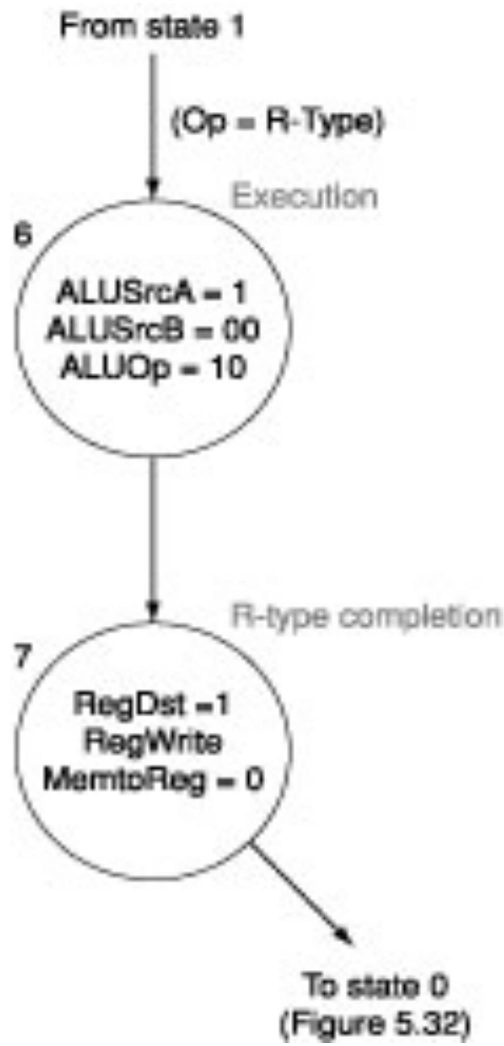
# Instruction Fetch/Decode (IF/ID) State Machine



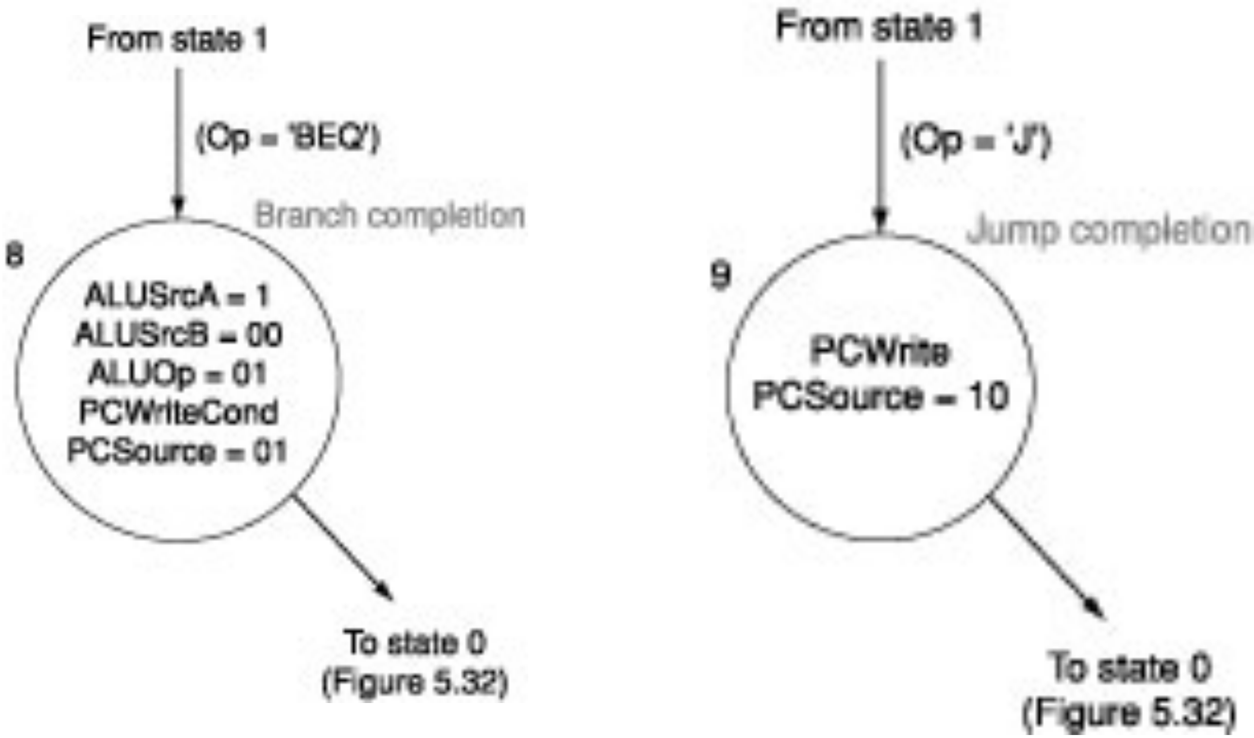
# Memory Reference State Machine



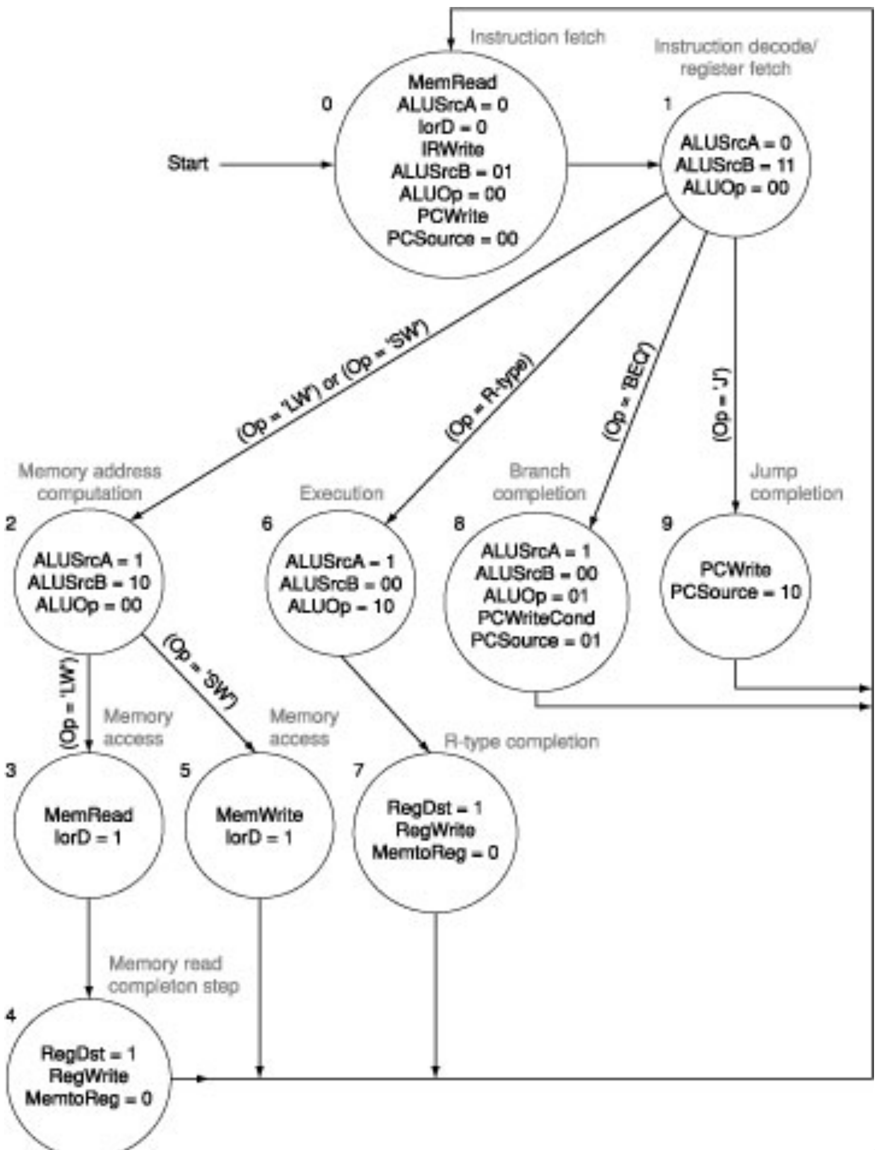
# R-type Instruction State Machine



# Branch/Jump State Machine



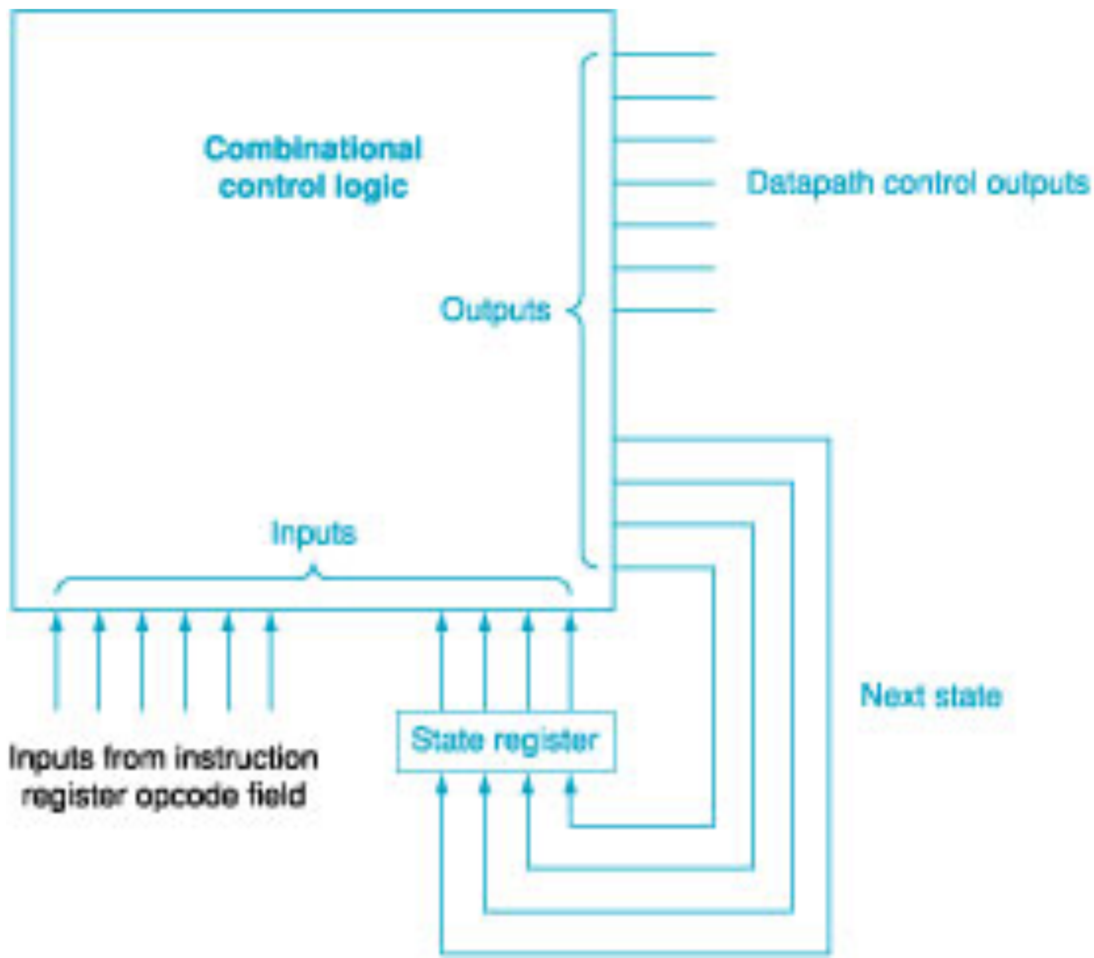
# Put it all together!



# Control for Multicycle

- Need to define the controls
- Need to come up with some way to sequence the controls
  - Two techniques
    - finite state machine
    - microprogramming

# Finite State Machine



# MicroProgramming (sec. 5.7)

- The idea is to build a (very small) processor to generate the controls signals at the right time.
- At each stage (cycle) one *microinstruction* is executed – the result changes the value of the control signals.
- Somebody writes the *microinstructions* that make up each MIPS instruction.

# Example microinstructions

Fetch next instruction: ← microinstruction

- turn on instruction memory read
- feed PC to memory address input
- write memory data output in to a holding register.

Control Signals

Compute Address:

- route contents of base register to ALU
- route sign-extended offset to ALU
- perform ALU add
- write ALU output in to a holding register.

# Sequencing

- In addition to setting some control signals, each microinstruction must specify the next microinstruction that should be executed.
- 3 Options:
  - execute next microinstruction (default)
  - start next MIPS instruction (Fetch)
  - Dispatch (depends on control unit inputs).

# Microinstruction Format

- A bunch of bits – one for each control line needed by the control unit.
  - bits specify the values of the control lines directly.
- Some bits that are used to determine the next microinstruction executed.

# Dispatch Sequencing

- Can be implemented as a table lookup.
  - bits in the microinstruction tell what row in the table.
  - inputs to the control unit tell what column.
  - value stored in table determines the microaddress of the next microinstruction.
- This is a simplified description (called a *microdescription*)

# Exceptions & Interrupts

- Hardest part of control is implementing exceptions and interrupts – i.e., events that change the normal flow of instruction execution.
- MIPS convention
  - Exception refers to any unexpected change in control flow w/o knowing if the cause is internal or external.
  - Interrupts refer to only events who are externally caused.
  - Ex. Interrupts: I/O device request (ignore for now)
  - Ex. Exceptions: undefined instruction, arithmetic overflow

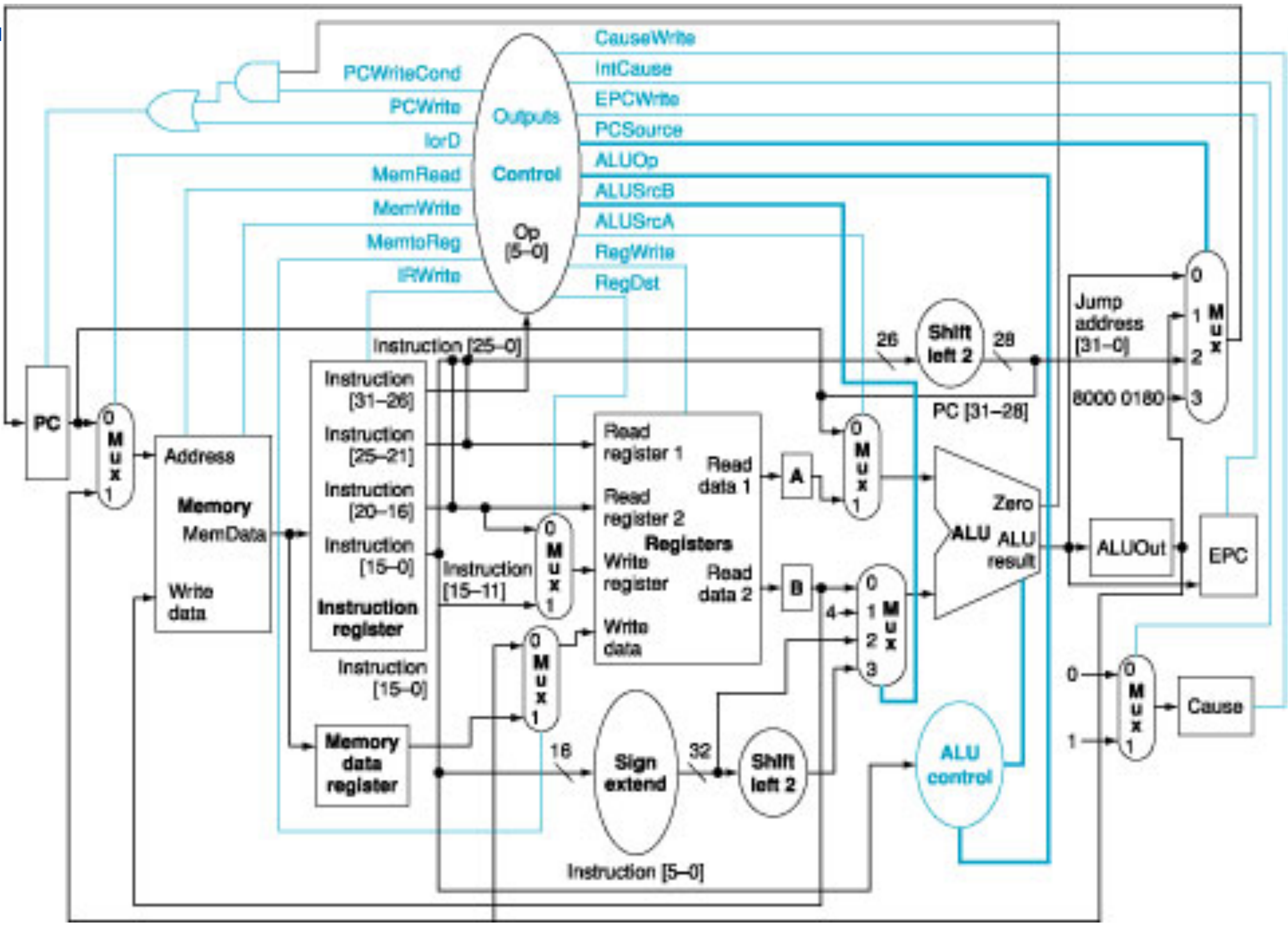
# Handling Exceptions

- Let's implemented exceptions for handling
  - Undefined instruction
  - Overflow
- Basic actions
  - Save the offending instruction address in the Exception Program Counter (EPC).
  - Transfer control to the OS at some specified address
  - Once exception is handled by OS, then either terminate the program or continue on using the EPC to determine where to restart.
- OS actions are determined based on what caused the exception.
  - So, OS needs a Cause register which determines which path w/i the exception
  - Alternative implementation – Vectored Interrupts – where each cause of an exception or interrupt is given a specific OS address to jump to.
  - We'll use the first method.

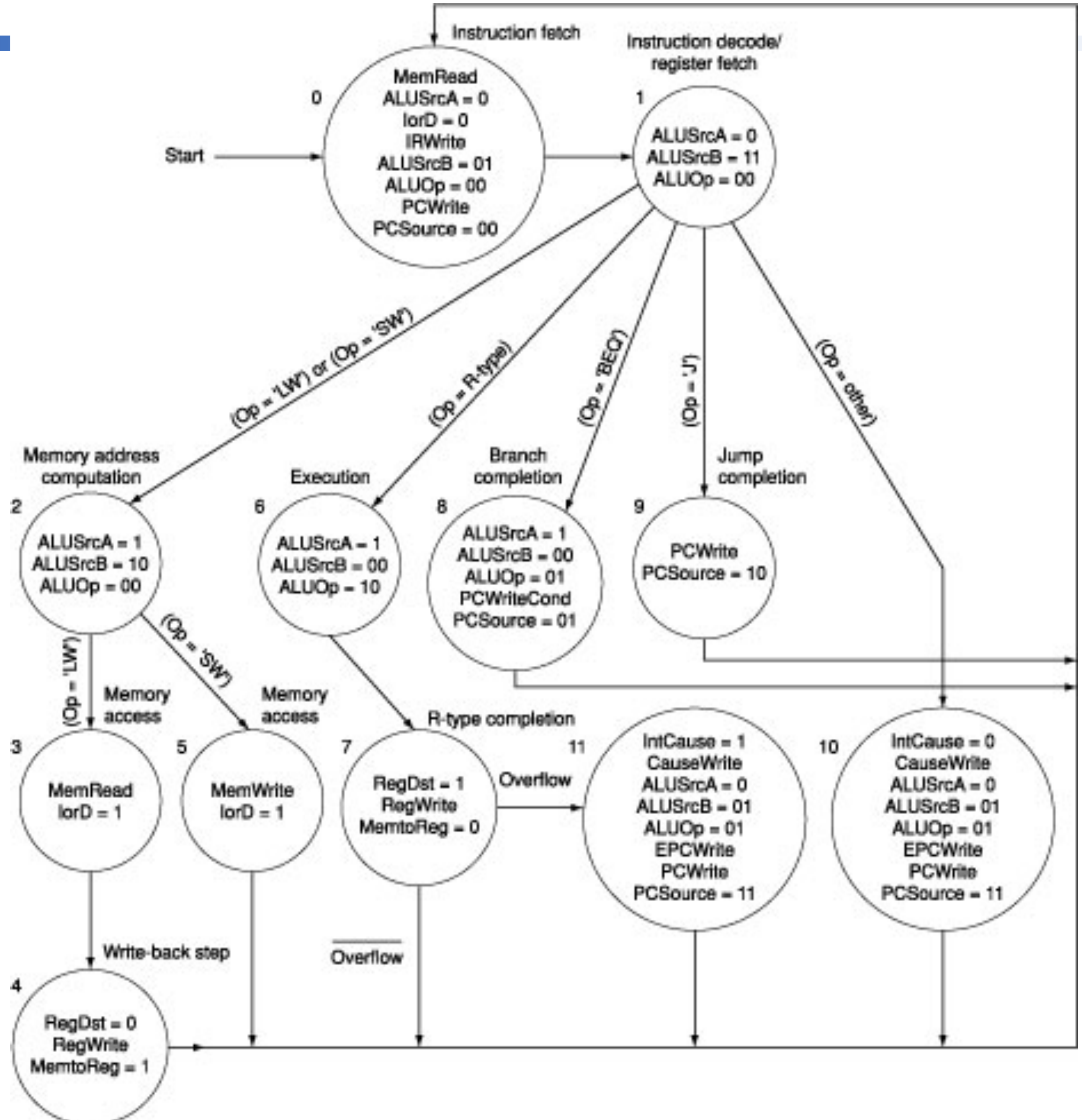
# Extending the Multicycle D&C

- What datapath elements to add?
  - **EPC:** a 32-bit register used to hold the address of the affected instruction.
  - *Cause:* A 32-bit register used to record the cause of the exception. (undef instruction = 0 and overflow = 1).
- What control lines to add?
  - ***EPCWrite*** and ***Cause write*** control signals to allow regs to be written.
  - ***IntCause*** (1-bit) control signal to set the low-order bit of the cause register to the appropriate value.

# Revised Datapath & Control



# Final FSM w/ exception handling



# Pipelining

# Multicycle Instructions

- Chop each instruction in to stages.
- Each stage takes one cycle.
- We need to provide some way to sequence through the stages:
  - microinstructions
- Stages can *share* resources (ALU, Memory).

# Pipelining

- We can overlap the execution of multiple instructions.
- At any time, there are multiple instructions being executed – each in a different stage.
- So much for sharing resources ?!?

# The Laundry Analogy

Non-pipelined approach:

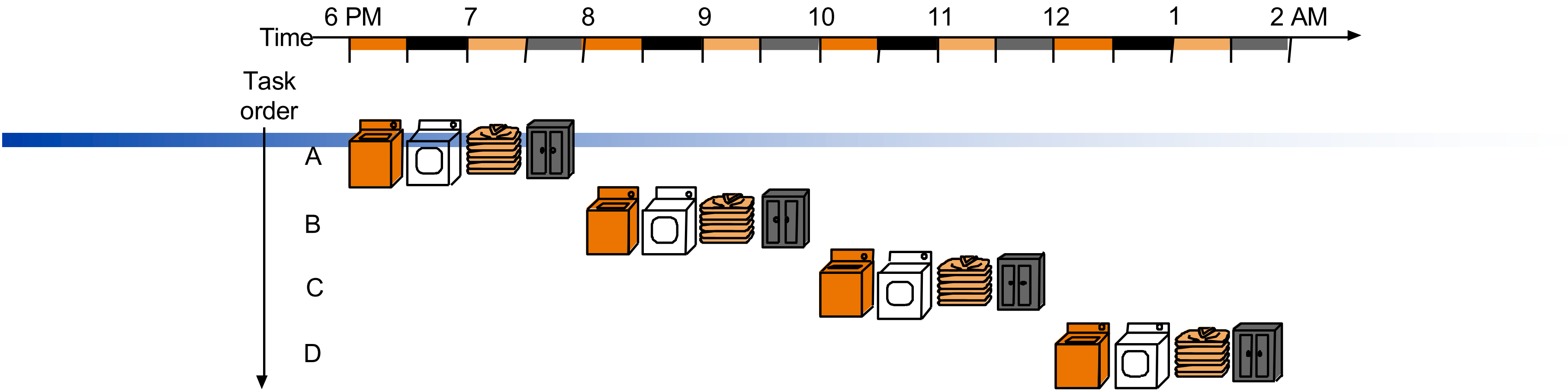
1. run 1 load of clothes through washer
2. run load through dryer
3. fold the clothes (optional step for students)
4. put the clothes away (also optional).

Two loads? Start all over.

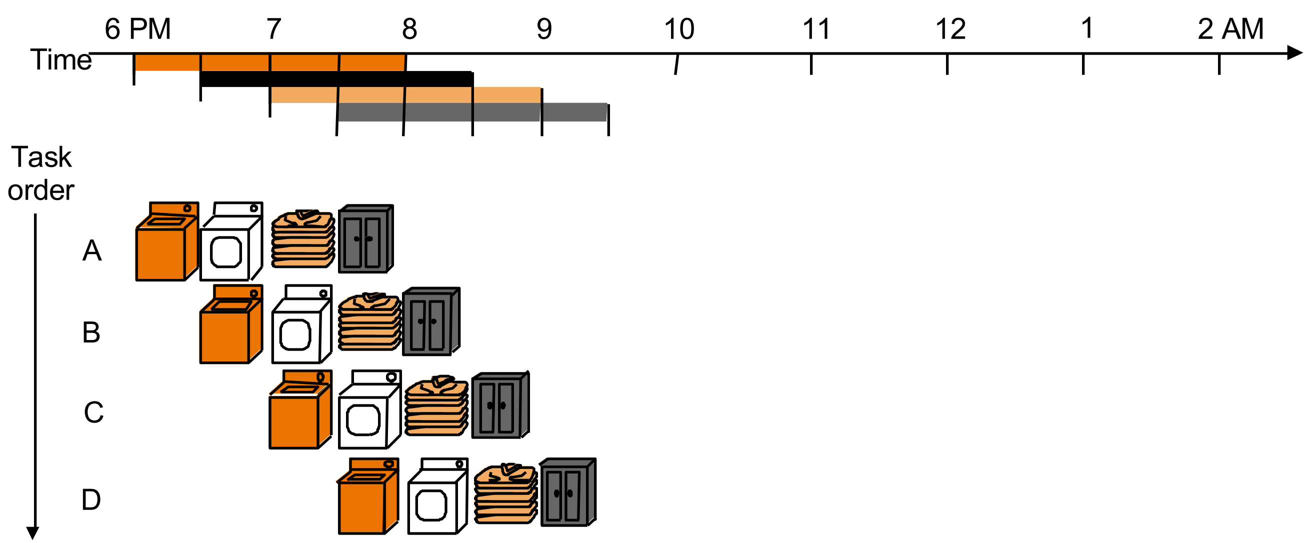
# Pipelined Laundry

- While the first load is drying, put the second load in the washing machine.
- When the first load is being folded and the second load is in the dryer, put the third load in the washing machine.

- Admittedly unrealistic scenario for CS students, as most only own 1 load of clothes…



6 PM
7
8
9
10
11
12
1
2 AM
Time
Task order
A
B
C
D

6 PM
7
8
9
10
11
12
1
2 AM
Time
Task order
A
B
C
D

# Laundry Performance

- For 4 loads:
  - non-pipelined approach takes 16 units of time.
  - pipelined approach takes 7 units of time.

- For 816 loads:
  - non-pipelined approach takes 3264 units of time.
  - pipelined approach takes 819 units of time.

# Execution Time vs. Throughput

- It still takes the same amount of time to get your favorite pair of socks clean, pipelining won't help.
- However, the total time spent away from CompOrg homework is reduced.

It's the classic "Socks vs. CompOrg" issue.

# Instruction Pipelining

First we need to break instruction execution into discrete stages:

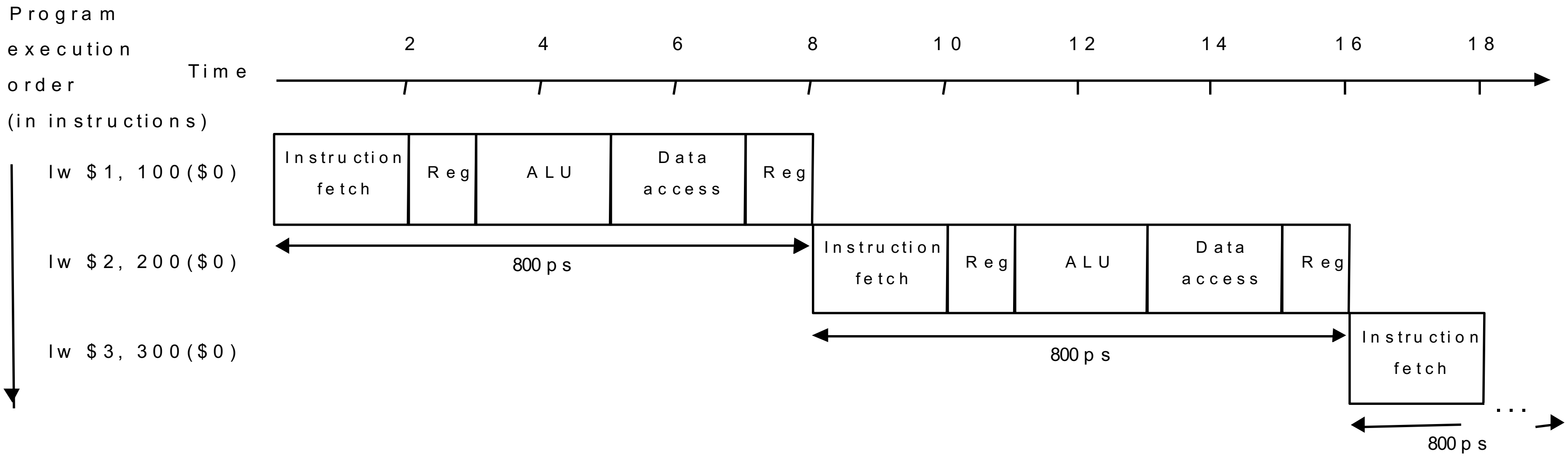
1. Instruction Fetch
2. Instruction Decode/ Register Fetch
3. ALU Operation
4. Data Memory access
5. Write result into register

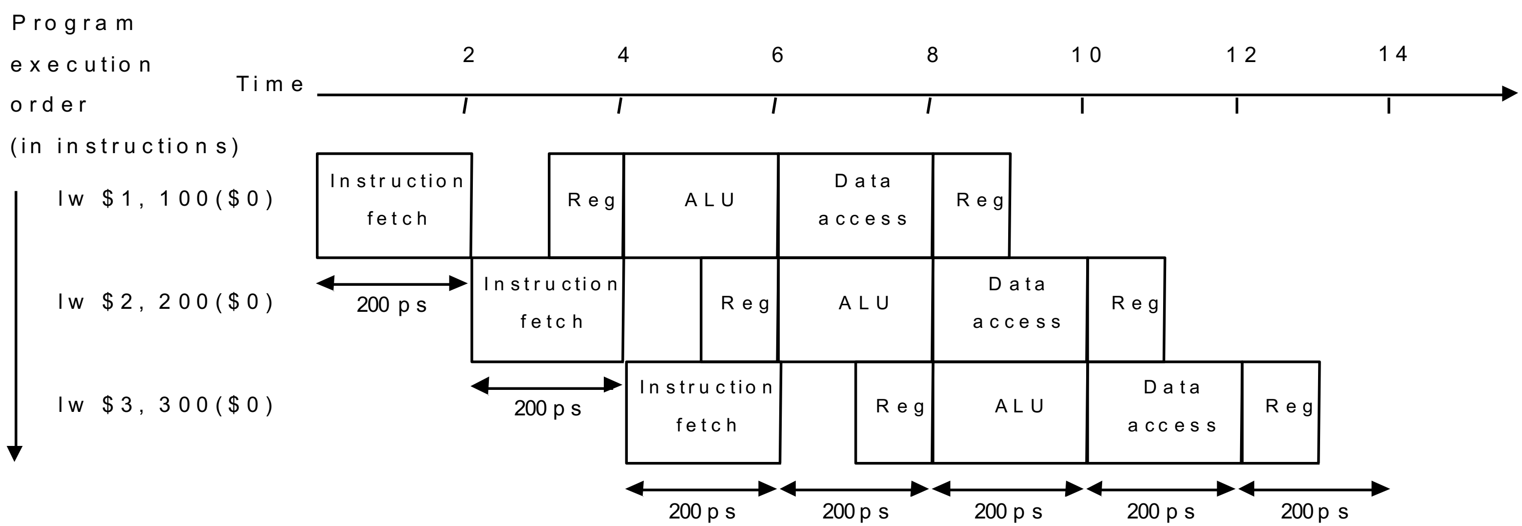
# Operation Timings

- Some estimated timings for each of the stages:

| | |
|---|---|
| Instruction Fetch | 200 ps |
| Register Read | 100 ps |
| ALU Operation | 200 ps |
| Data Memory | 200 ps |
| Register Write | 100 ps |

# Comparison


Program execution order (in instructions)
Time
2 4 6 8 10 12 14 16 18
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
Instruction fetch
Reg
ALU
Data access
Reg
800 ps
800 ps
800 ps
...


Program execution order (in instructions)
Time
2 4 6 8 10 12 14
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
Instruction fetch
Reg
ALU
Data access
Reg
200 ps
200 ps
200 ps 200 ps 200 ps 200 ps 200 ps

# RISC and Pipelining

- One of the major advantages of RISC instruction sets is the relative simplicity of a pipeline implementation.
  - It's much more complex in a CISC processor!!
- RISC (MIPS) design features that make pipelining easy include:
  - single length instruction (always 1 word)
  - relatively few instruction formats
  - load/store instruction set
  - operands must be aligned in memory (a single data transfer instruction requires a single memory operation).

# Pipeline Hazard

- Something happens that means the next instruction cannot execute in the following clock cycle.
- Three kinds of hazards:
  - structural hazard
  - control hazard
  - data hazard

# Structural Hazards

- Two stages require the same resource.
  - What if we only had enough electricity to run either the washer or the dryer at any given time?
  - What if MIPS datapath had only one memory unit instead of separate instruction and data memory?

# Avoiding Structural Hazards

- Design the pipeline carefully.
- Might need to duplicate resources
  - an Adder to update PC, and ALU to perform other operations.
- Detecting structural hazards at execution time (and delaying execution) is not something we want to do (structural hazards are minimized in the design phase).

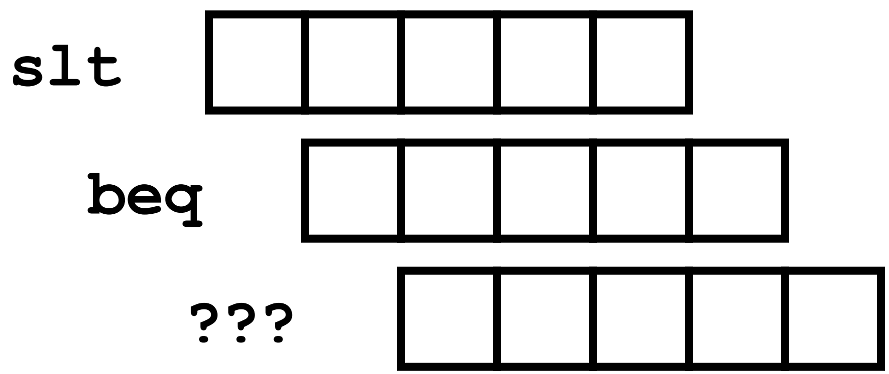
# Control Hazards

- When one instruction needs to make a decision based on the results of another instruction that has not yet finished.
- Example: conditional branch
  - The instruction that is fed to the pipeline right after a **beq** depends on whether or not the branch is taken.

# beq Control Hazard

```
a = b+c;
if (x!=0)
        y++;
...
```

```
        slt $t0,$s0,$s1
        beq $t0,$zero,skip
        addi $s0,$s0,1
skip:
        lw $s3,0($t3)
```

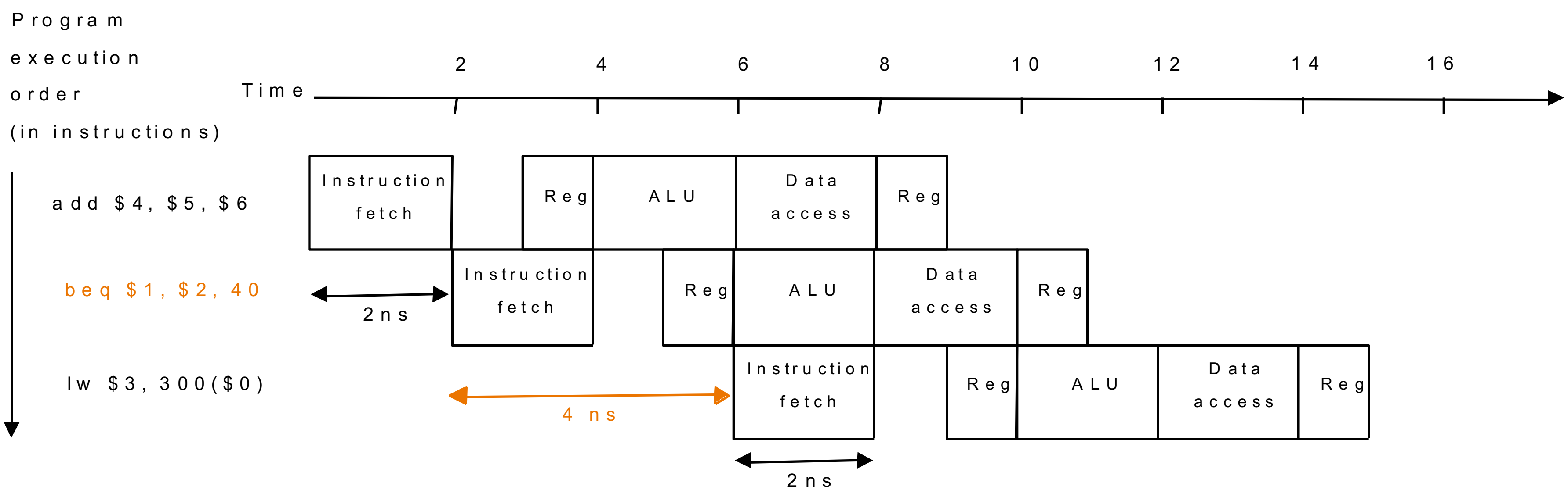



The instruction to follow the **beq** could be either the **addi** or the **lw**, it depends on the result of the **beq** instruction.

# One possible solution - stall

- We can include in the control unit the ability to *stall* (to keep new instructions from entering the pipeline until we know which one).
- Unfortunately conditional branches are very common operations, and this would slow things down considerably.

# A Stall

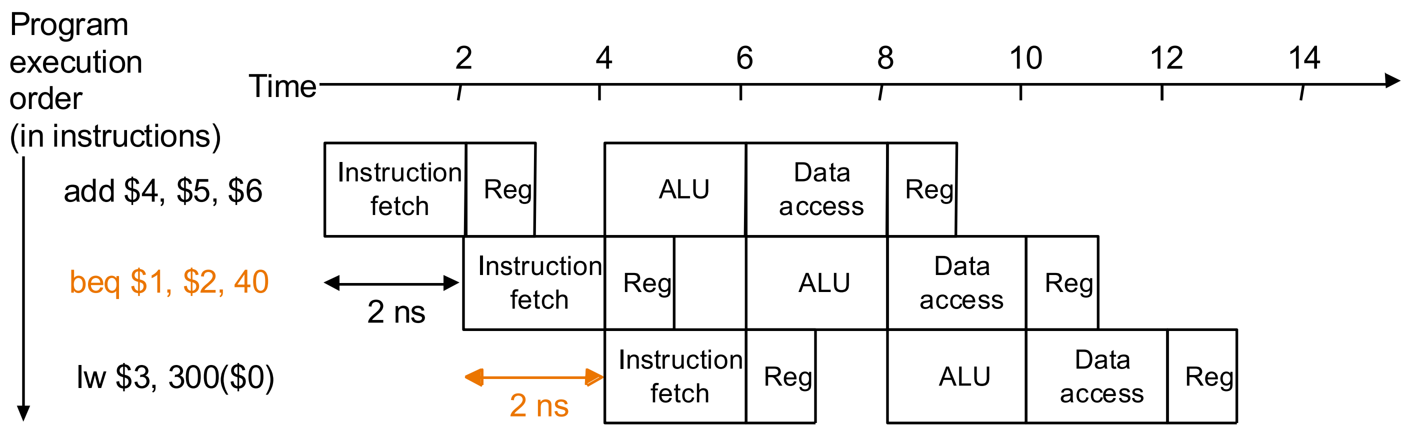


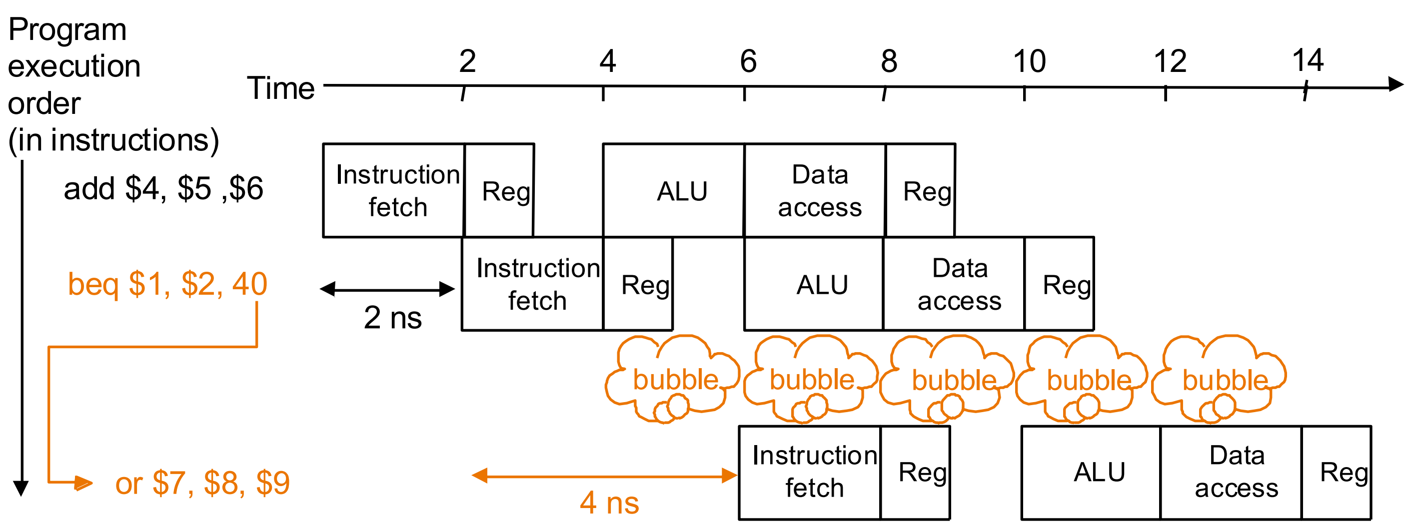

To achieve a 1 cycle stall (as shown above), we need to modify the implementation of the **beq** instruction so that the decision is made by the end of the second stage.

# Another strategy

- Predict whether or not the branch will be taken.
- Go ahead with the *predicted* instruction (feed it into the pipeline next).
- If your prediction is right, you don't lose any time.
- If your prediction is wrong, you need to undo some things and start the correct instruction

# Predicting branch not taken



# Dynamic Branch Prediction

- The idea is to build hardware that will come up with a prediction based on the past history of the specific branch instruction.
- Predict the branch will be taken if it has been taken more often than not in the recent past.
  - This works great for loops! (90% + correct).
  - We'll talk more about this …

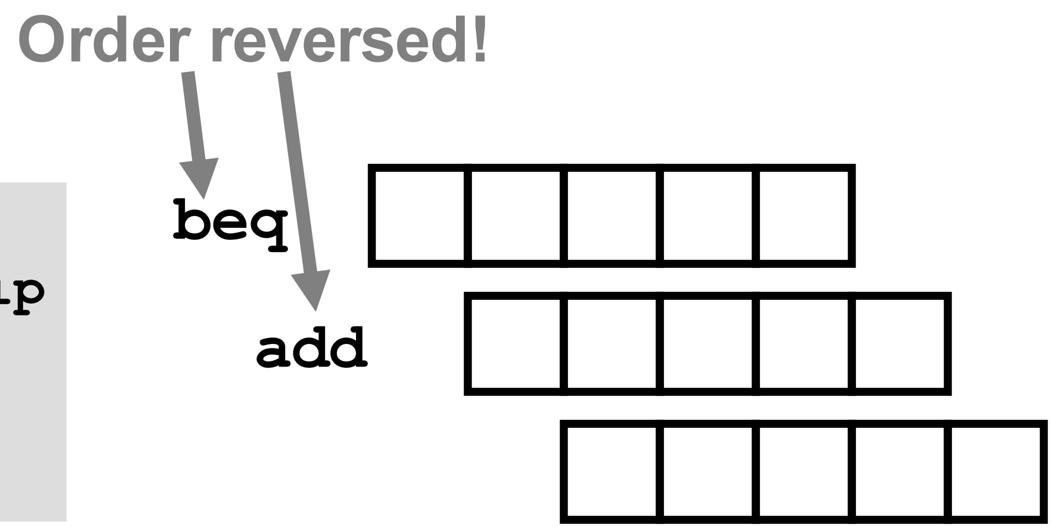
# Yet another strategy: delayed branch

- The compiler rearranges instructions so that the branch actually occurs delayed by one instruction from where its execution starts
- This gives the hardware time to compute the address of the next instruction.
- The new instruction is hopefully useful whether or not the branch is taken (this is tricky - compilers must be careful!).

# Delayed Branch

```
a = b+c;
if (x!=0)
      y++;
...
```

```
      add $s2,$s3,$s4
      beq $t0,$zero,skip
      addi $s0,$s0,1
skip:
      lw $s3,0($t3)
```

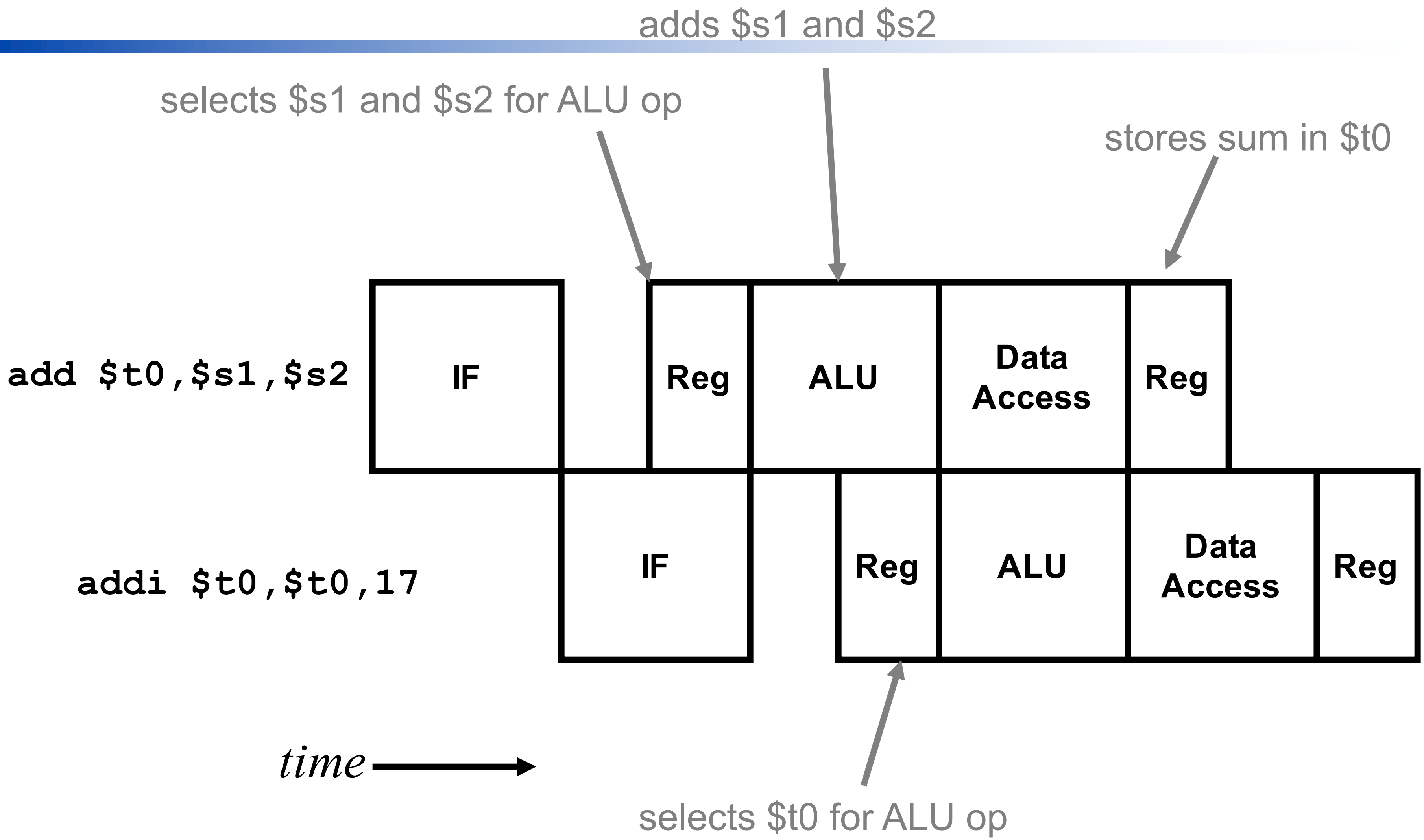



The compiler must generate code that differs from what you would expect.

# Data Hazard

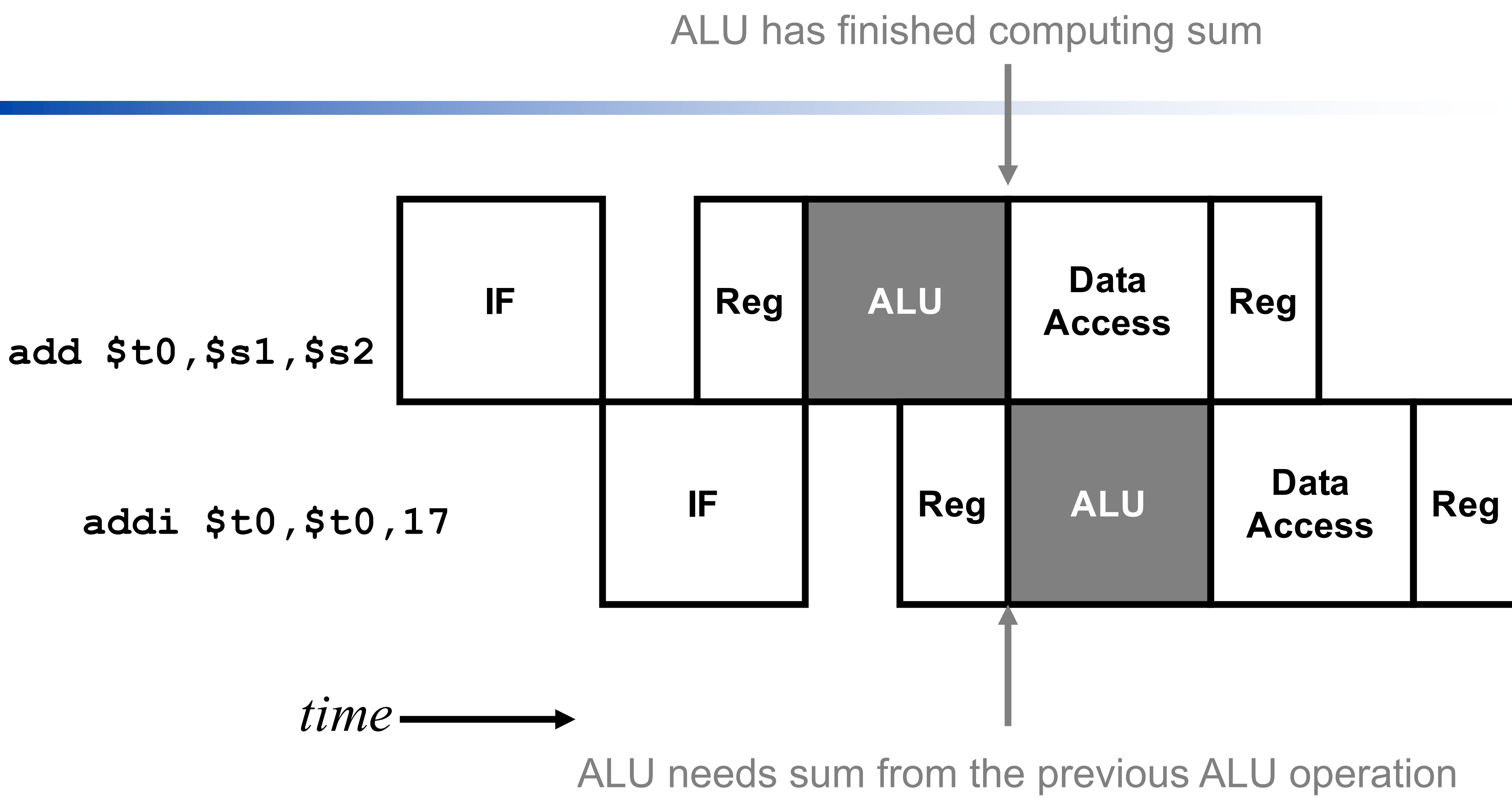
- One of the values needed by an instruction is not yet available (the instruction that computes it isn't done yet).
- This will cause a data hazard:

```
add $t0,$s1,$s2
addi $t0,$t0,17
```


adds $s1 and $s2
selects $s1 and $s2 for ALU op
stores sum in $t0
add $t0,$s1,$s2
IF
Reg
ALU
Data Access
Reg
addi $t0,$t0,17
IF
Reg
ALU
Data Access
Reg
time
selects $t0 for ALU op

# Handling Data Hazards

- We can hope that the compiler can arrange instructions so that data hazards never appear.
  - this doesn't work, as programs generally need to use previously computed values for everything!
- Some data hazards aren't real - the value needed is available, just not in the right place.

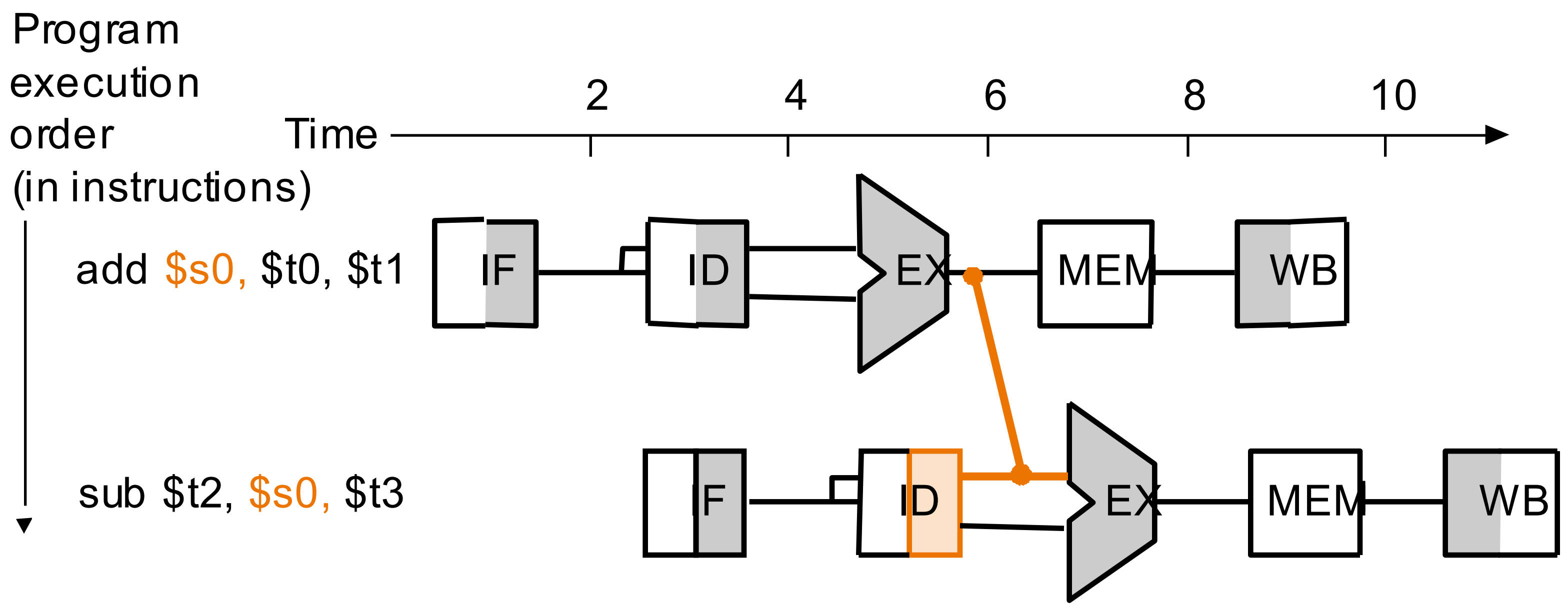



The sum is available when needed!

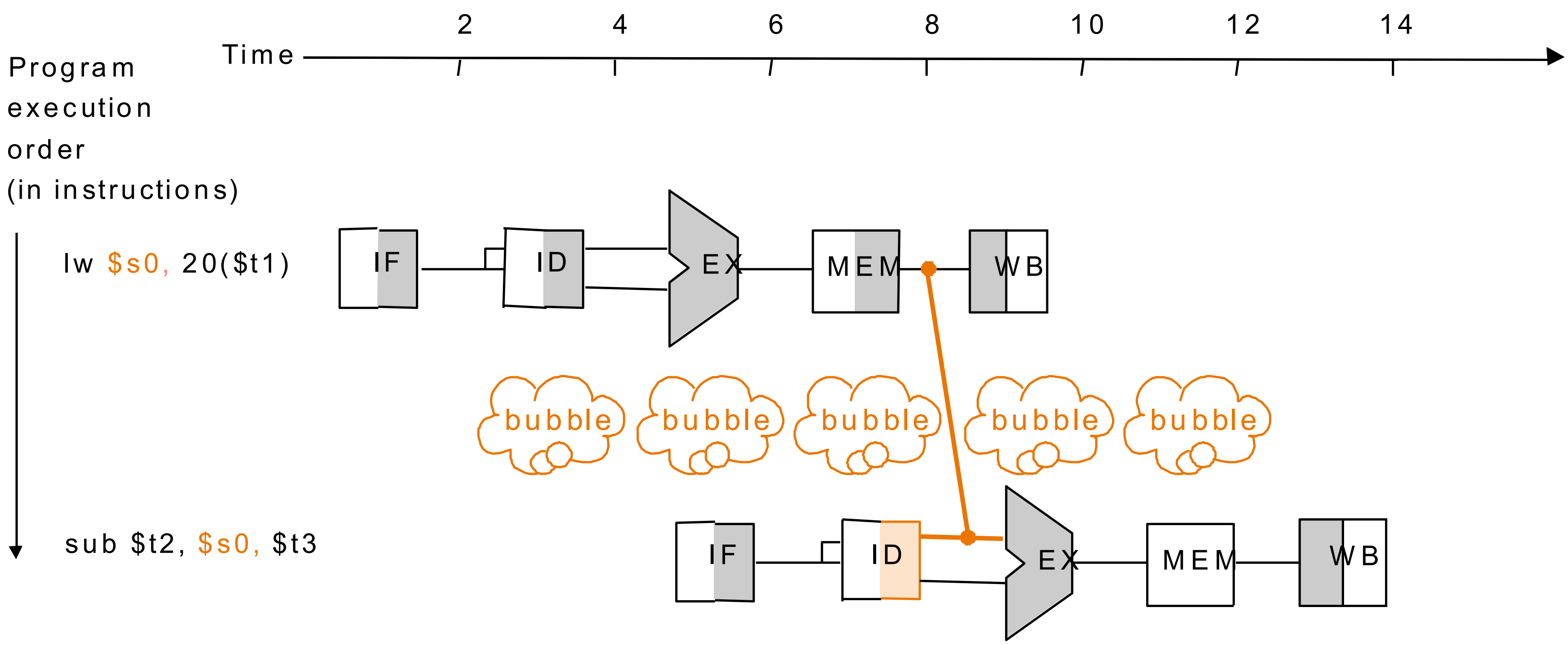
# Forwarding

- It's possible to *forward* the value directly from one resource to another (in time).
- Hardware needs to detect (and handle) these situations automatically!
  - This is difficult, but necessary.

# Picture of Forwarding



# Another Example



# Pipelining and CPI

- If we keep the pipeline full, one instruction completes every cycle.
- Another way of saying this: the average time per instruction is 1 cycle.
  - even though each instruction actually takes 5 cycles (5 stage pipeline).

CPI=1

# Correctness

Pipeline and compiler designers must be careful to ensure that the various schemes to avoid stalling do not change what the program does!

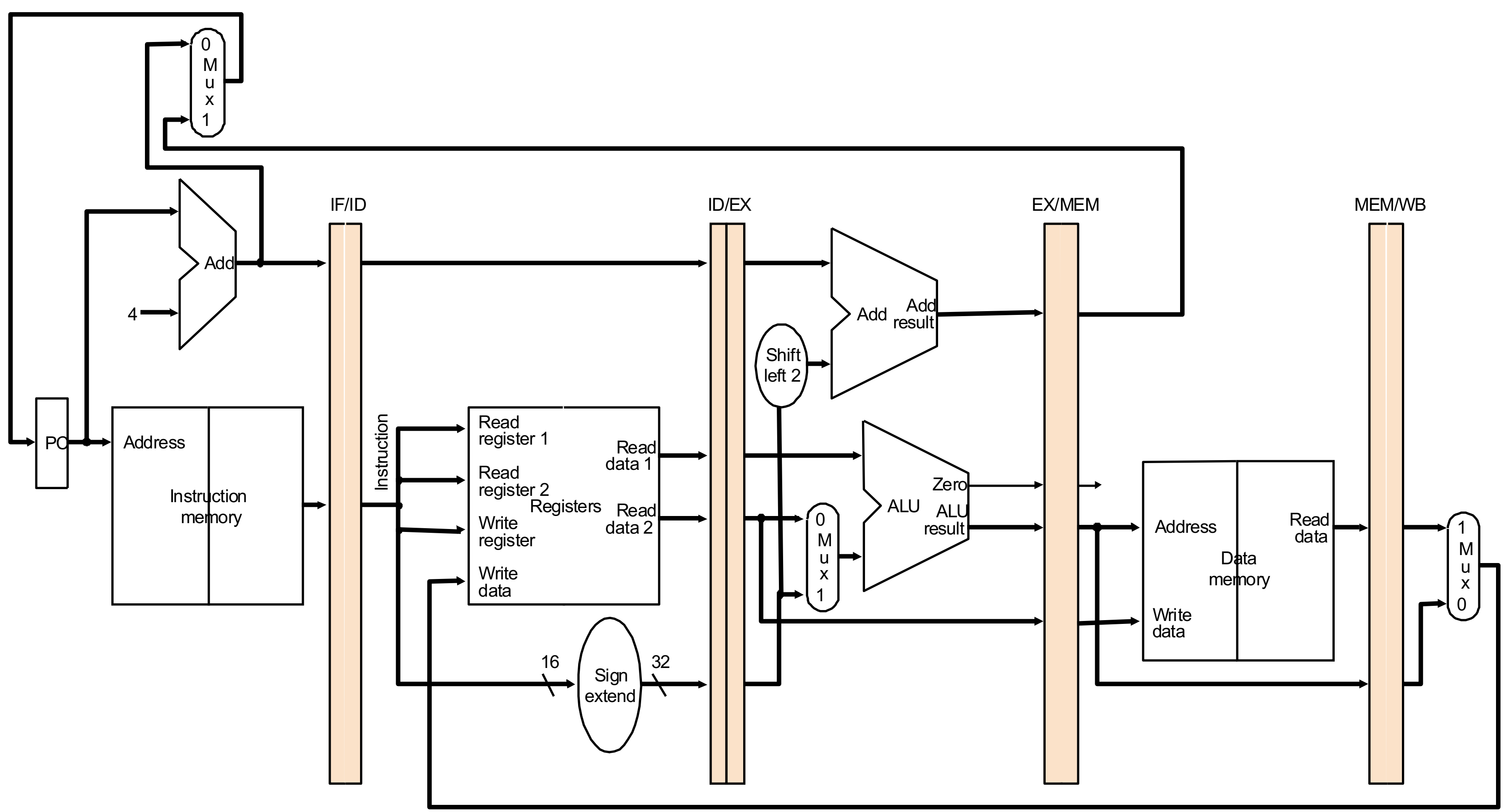
- only when and how it does it.
- It's impossible to test all possible combinations of instructions (to make sure the hardware does what is expected).
- It's impossible to test all combinations even without pipelining!

# Pipelined Datapath

We need to use a multicycle datapath.

- includes registers that store the result of each stage (to pass on to the next stage).
- can't have a single resource used by more than one stage at time.

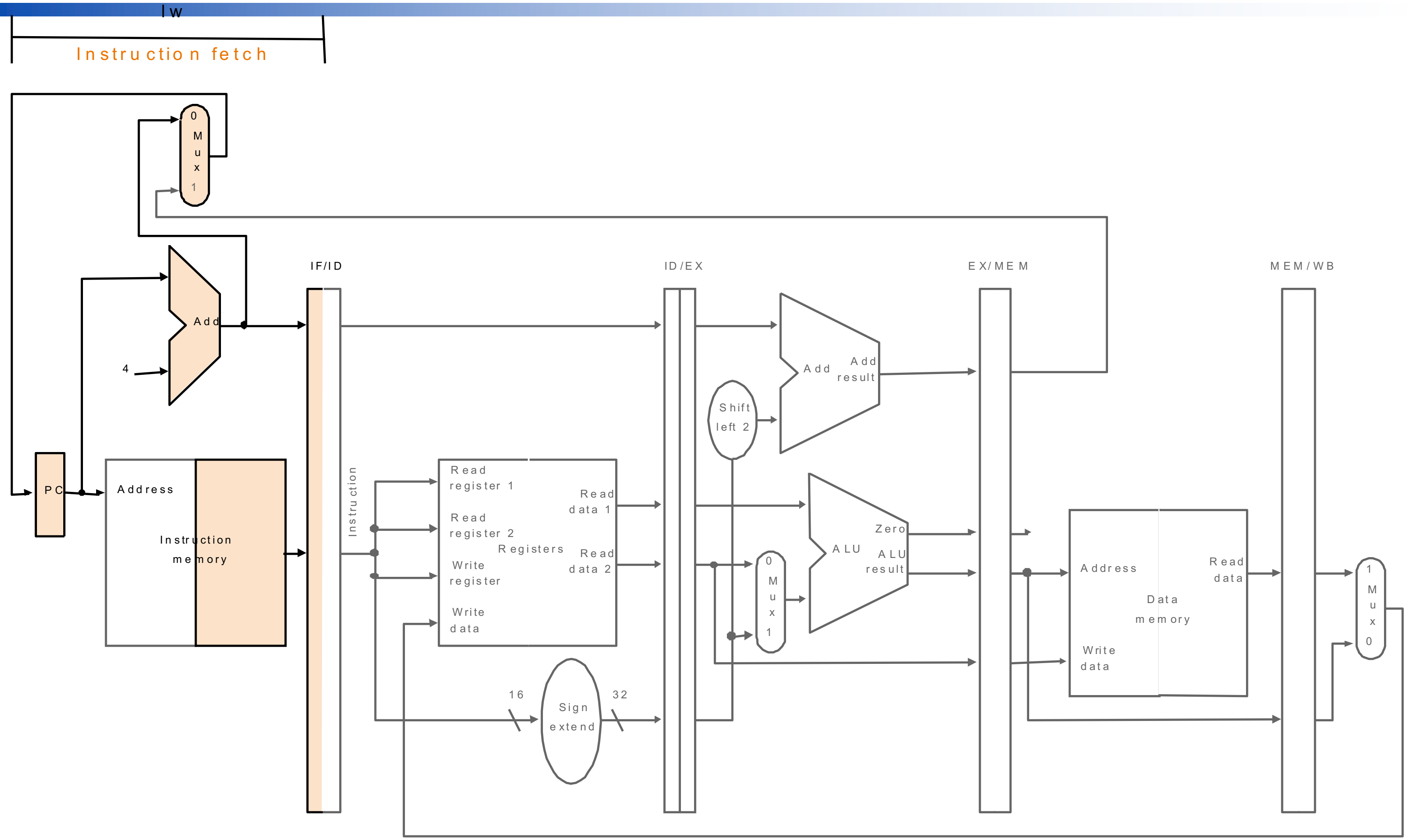
# Pipelined Datapath – 5 stages



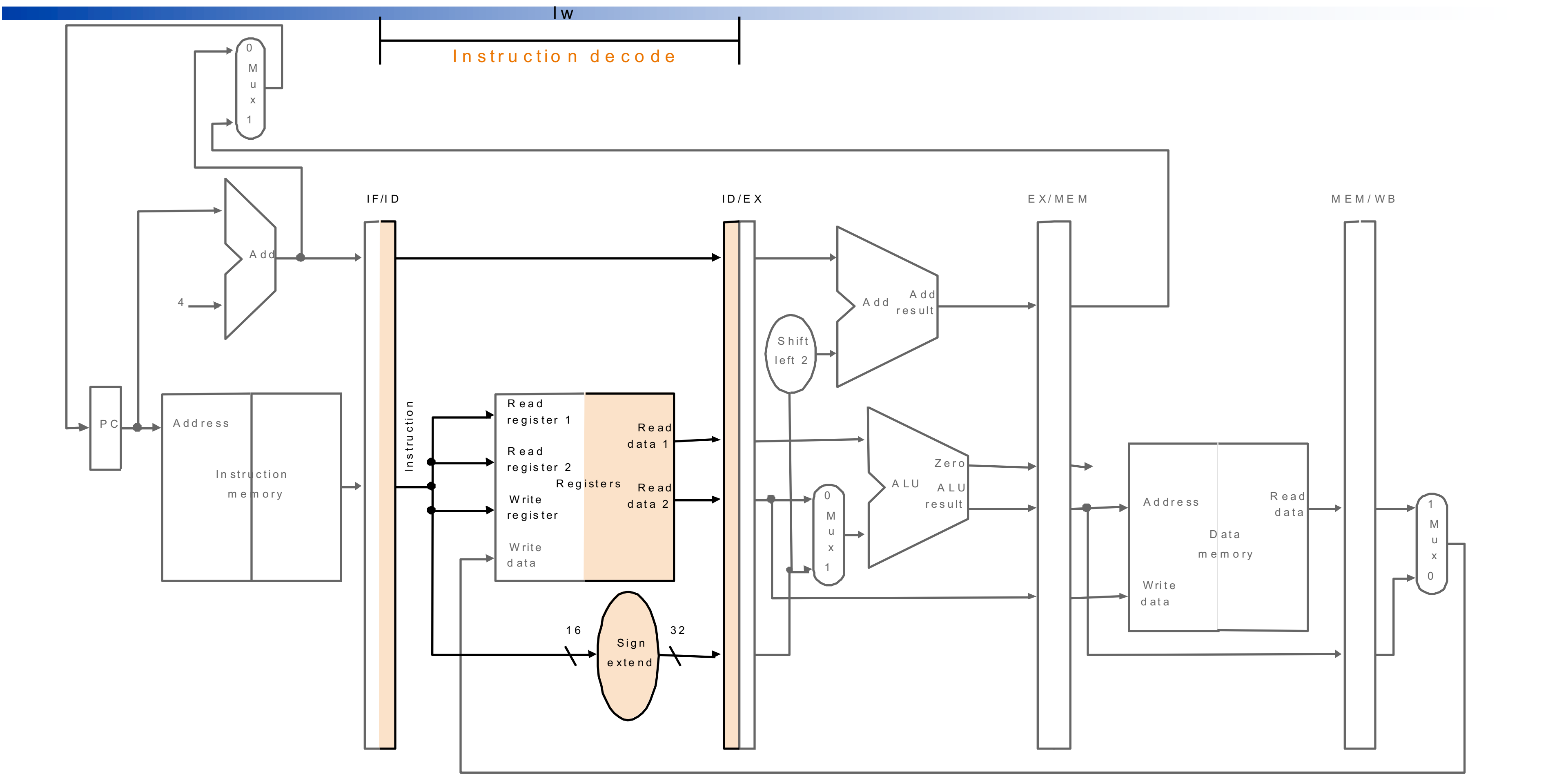
# `lw` and pipelined datapath

- We can trace the execution of a load word instruction through the datapath.

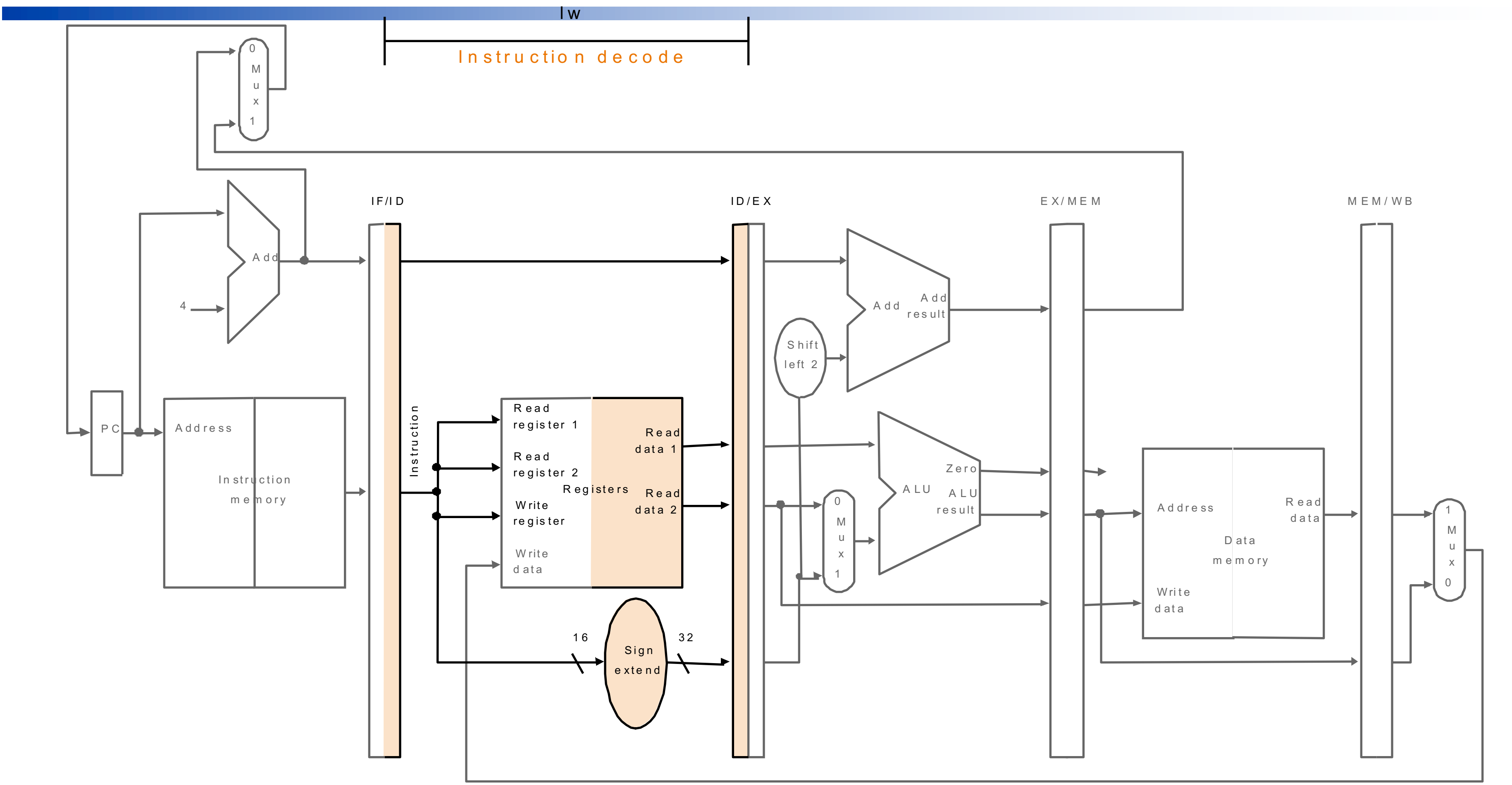
- We need to keep in mind that other instructions are using the stages not in use by our **`lw`** instruction!



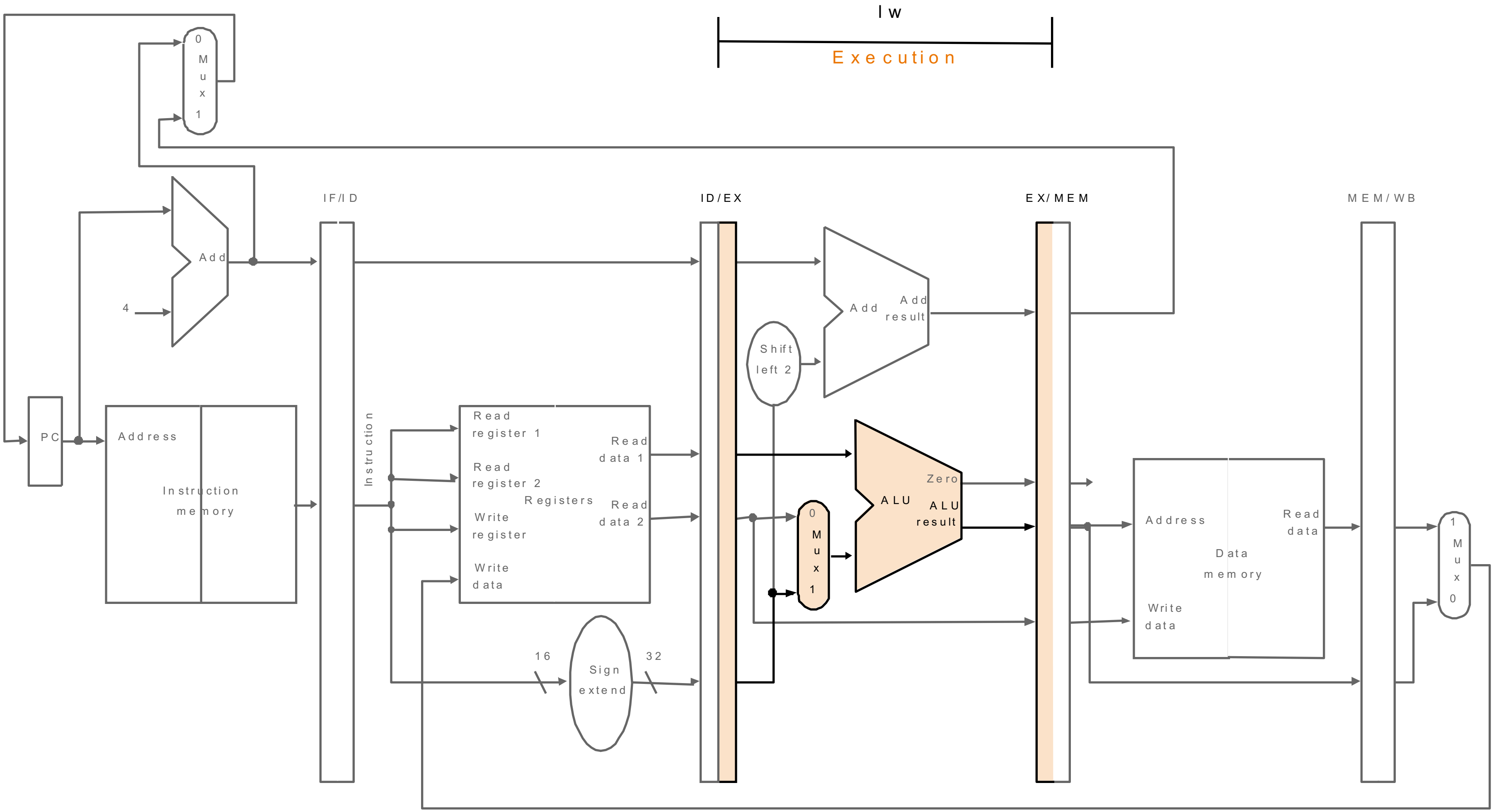

# Stage 1: Instruction Fetch (IF)



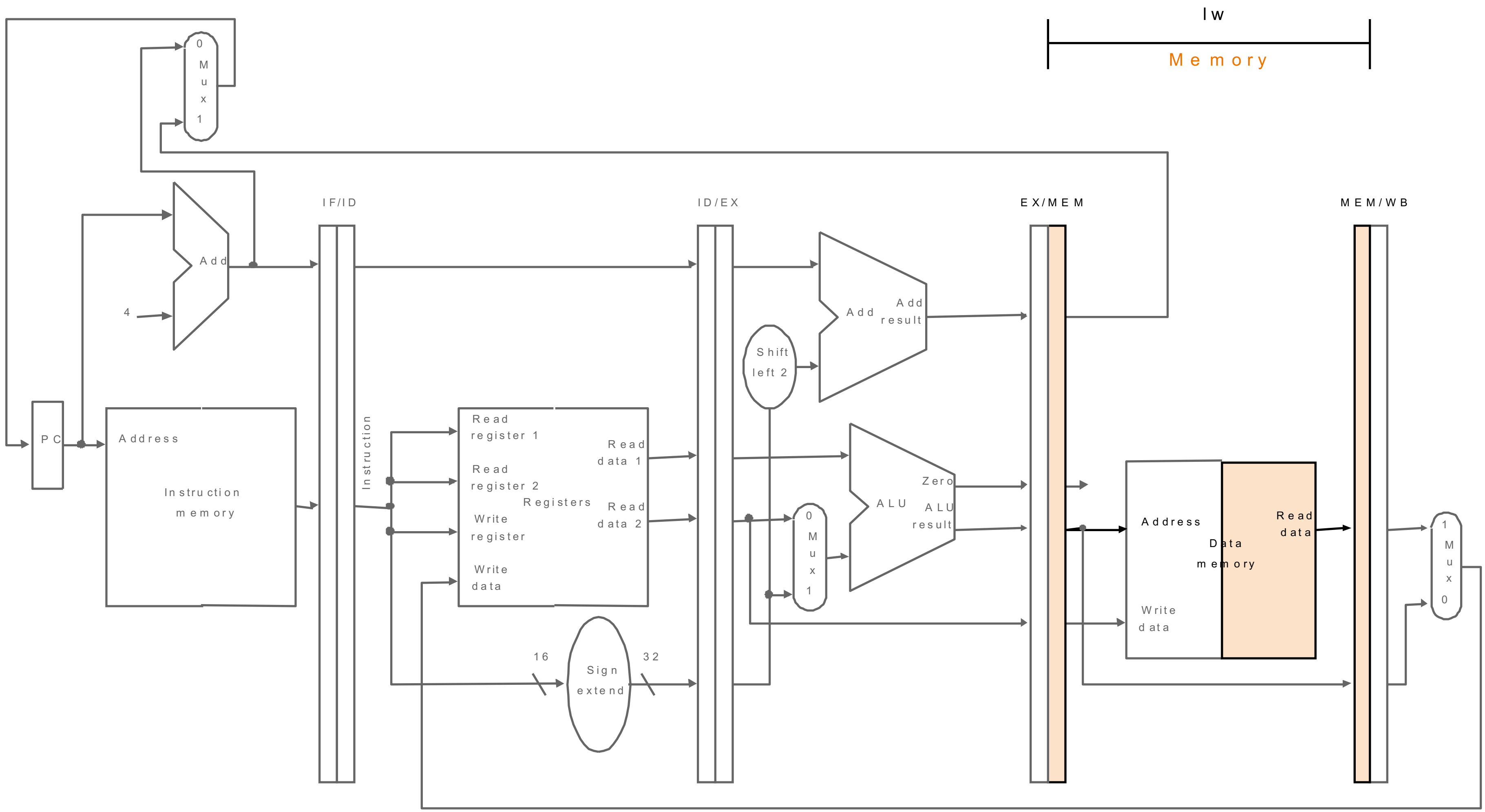

# Stage 2: Instruction Decode (ID)



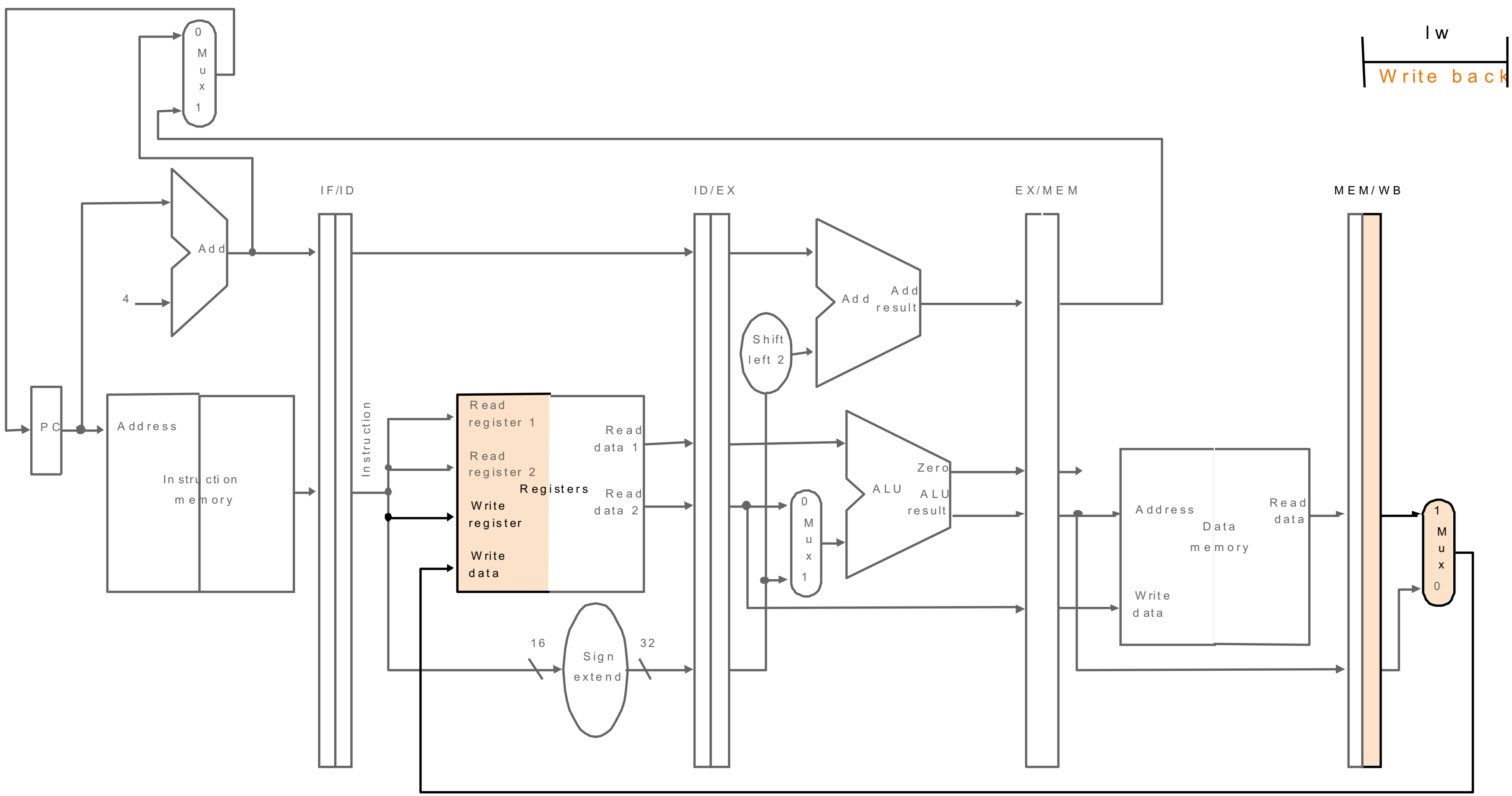

# Stage 2: Instruction Decode (ID)




# Stage 3: Execute (EX)




# Stage 4: Memory Access (MEM)




# Stage 5: WriteBack (WB)

# A Bug!

- When the value read from memory is written back to the register file, the inputs to the register file (write register #) are from a different instruction!
- To fix the bug we need to save the part of the `lw` instruction (5 bits of it specify which register should get the value from memory).

# New Datapath

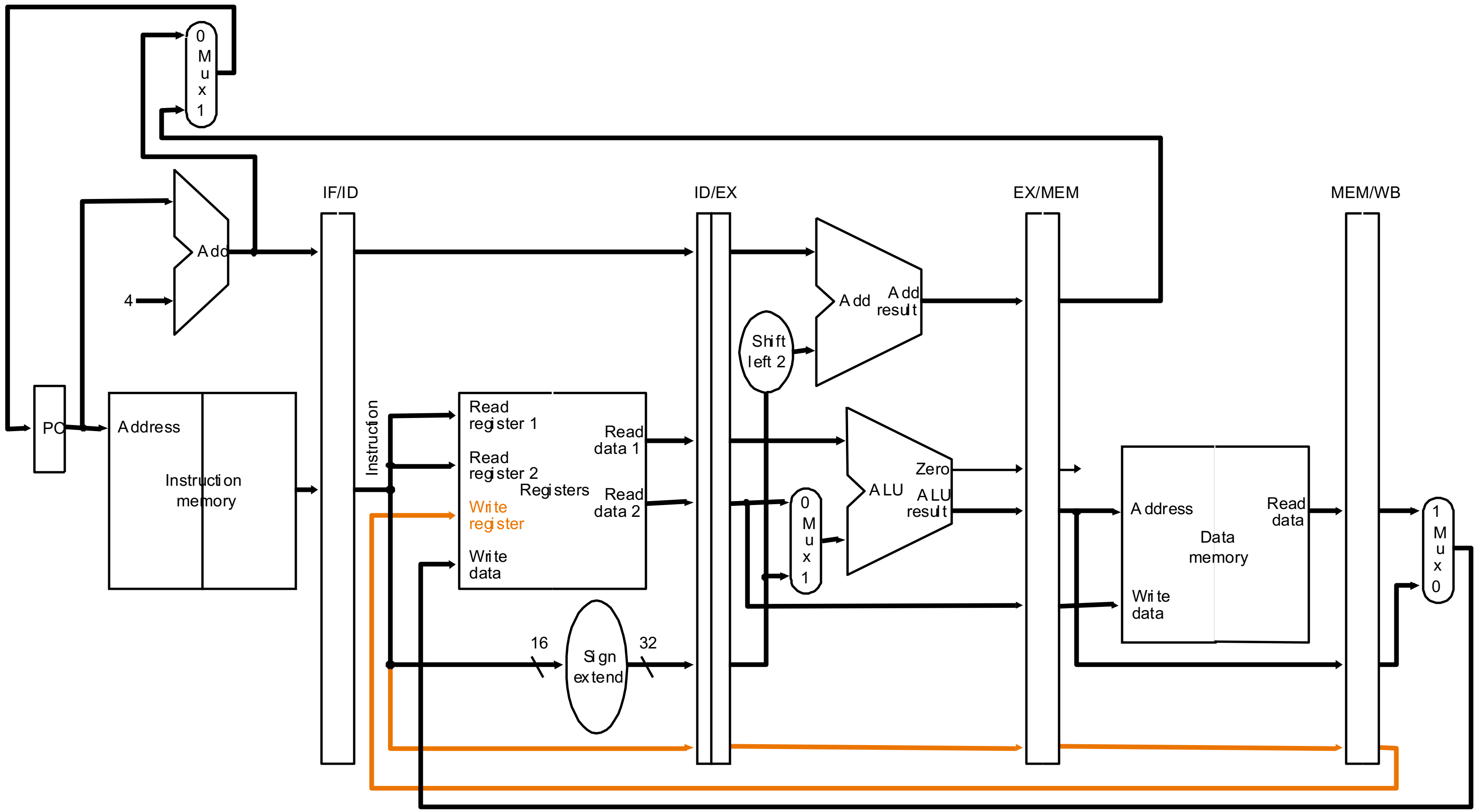
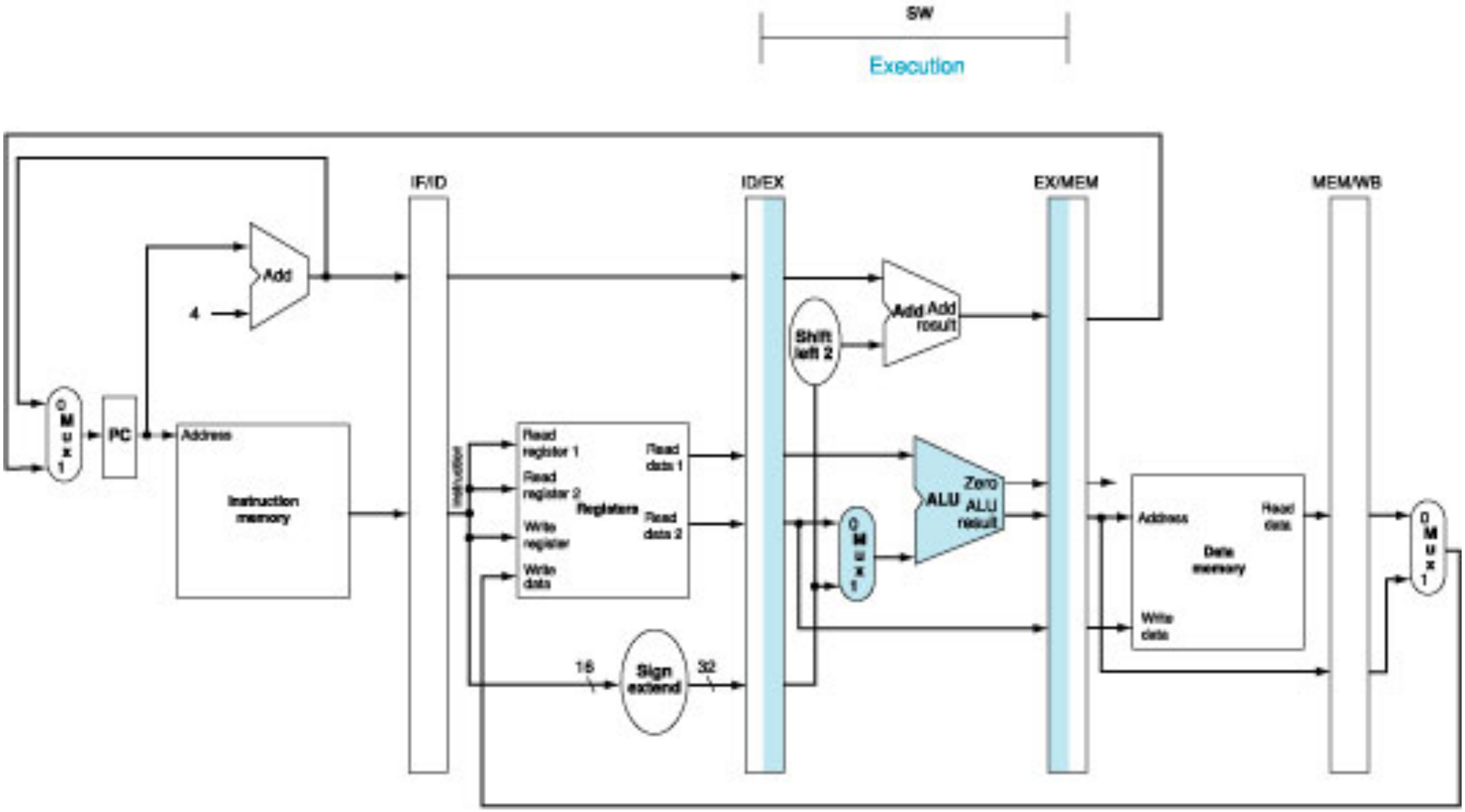
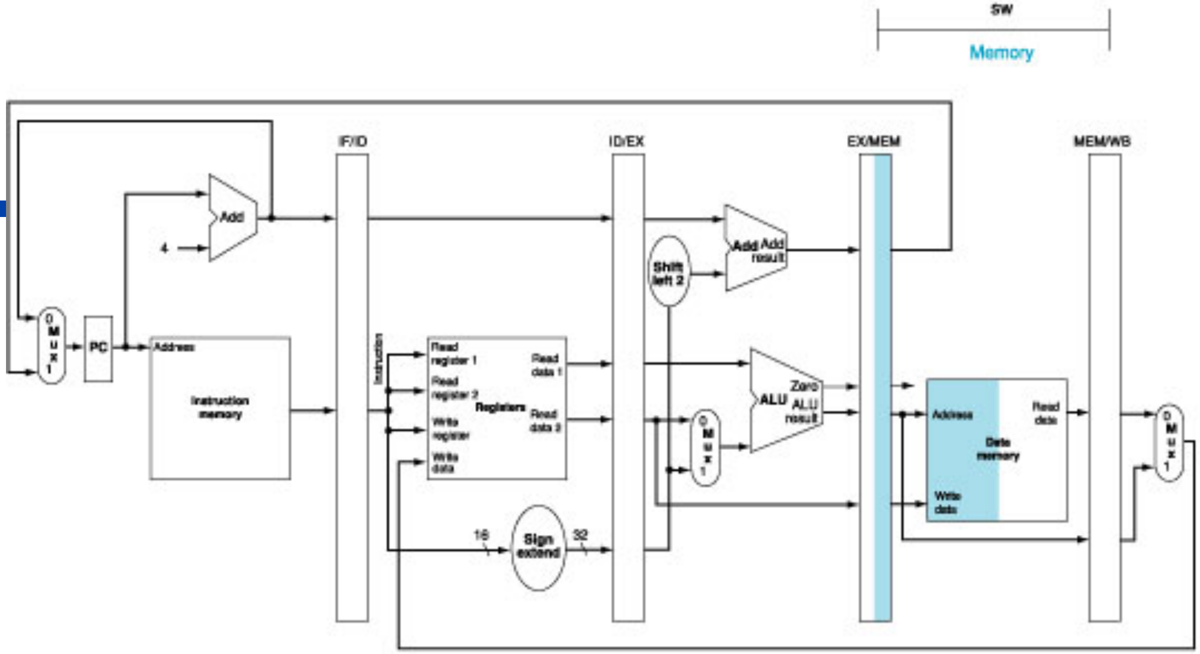


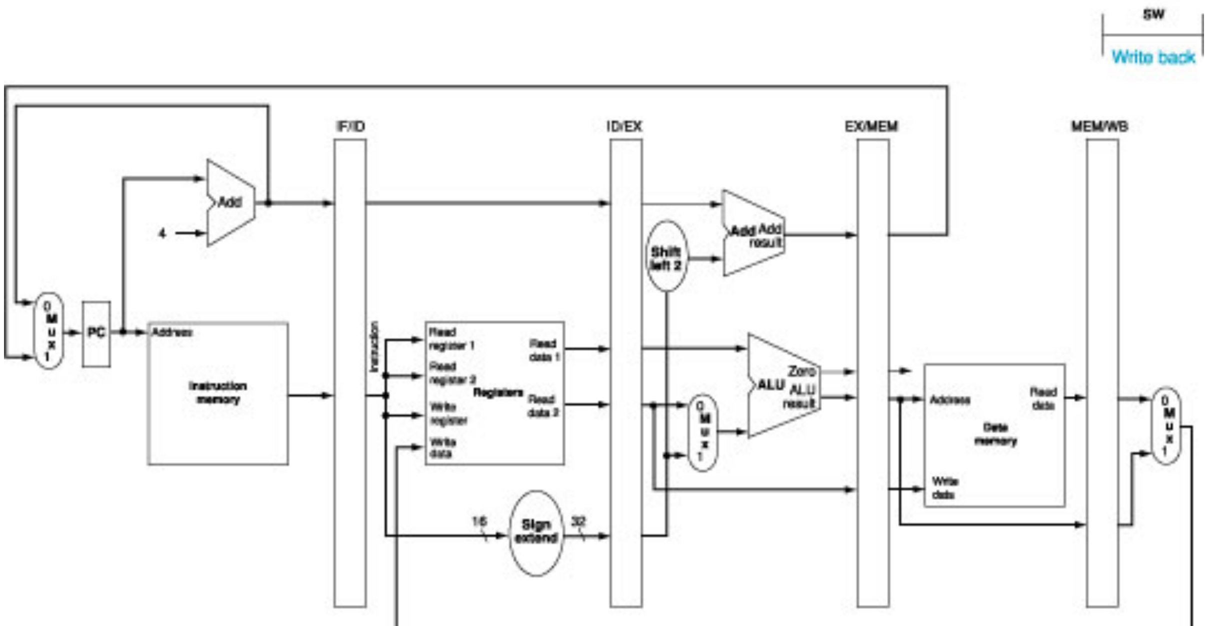

Figure 4.41

# Store Word (sw) Data Path Flow (EX)

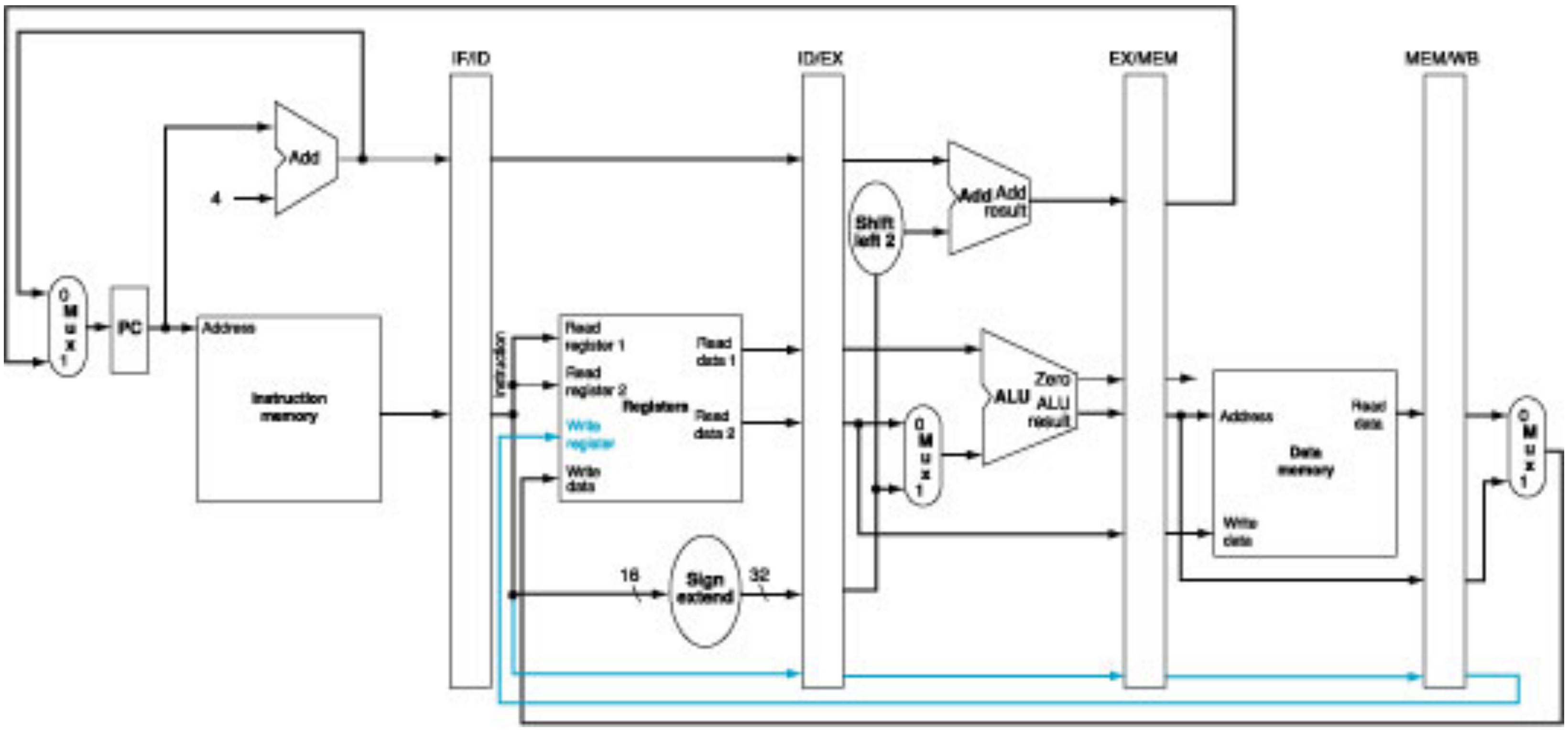


# SW Data Path (cont.)

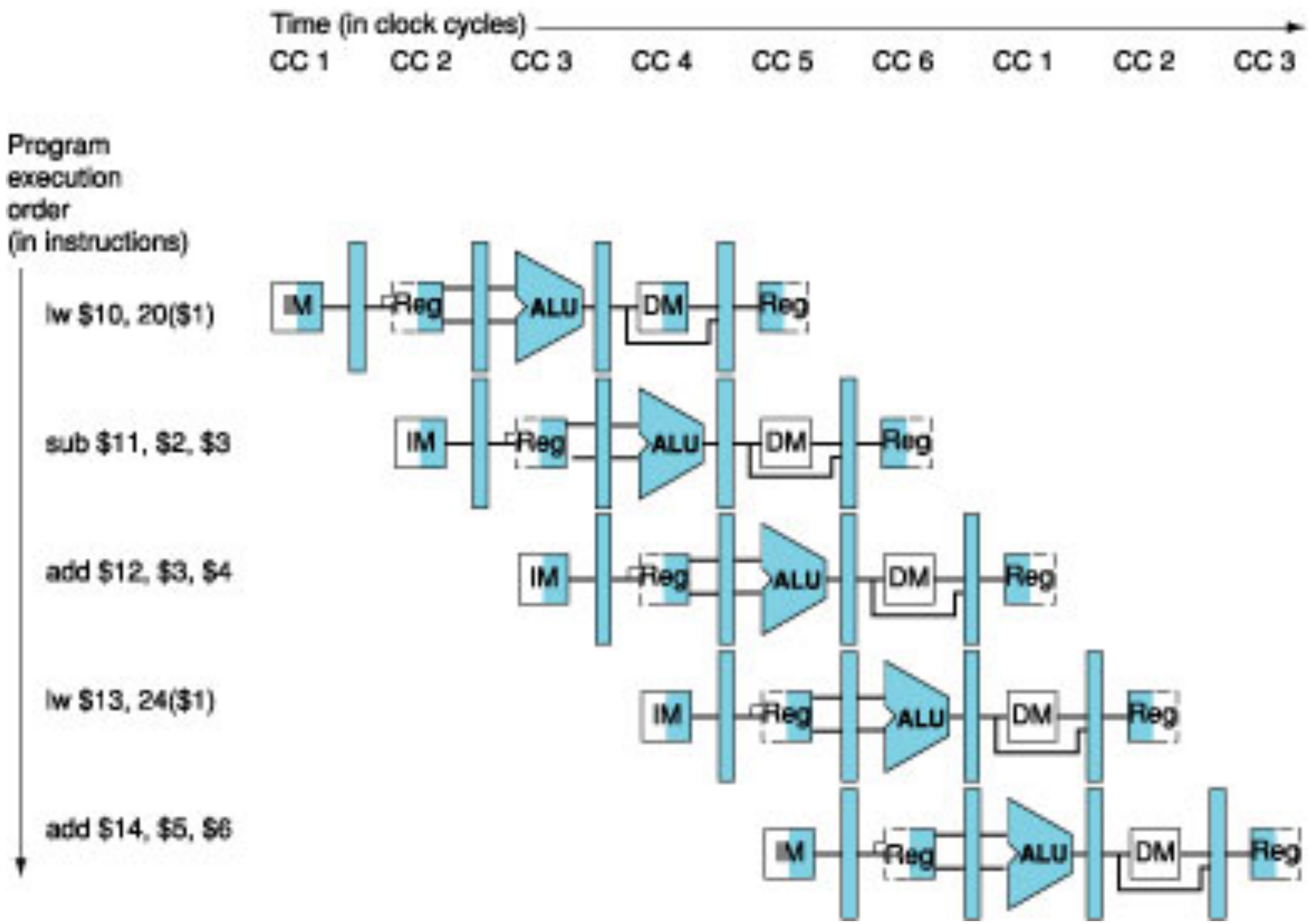


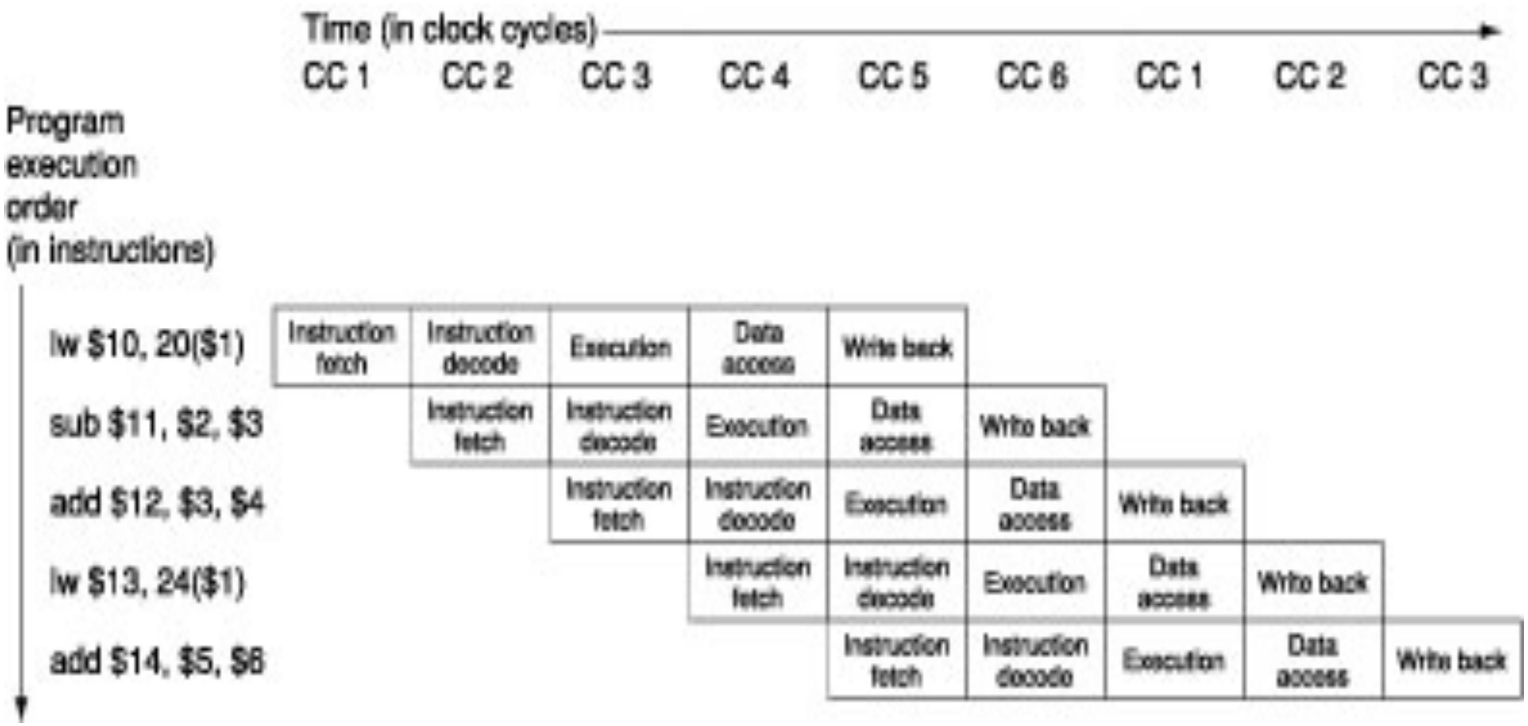
# Final Corrected Datapath



# Ex. With 5 instructions



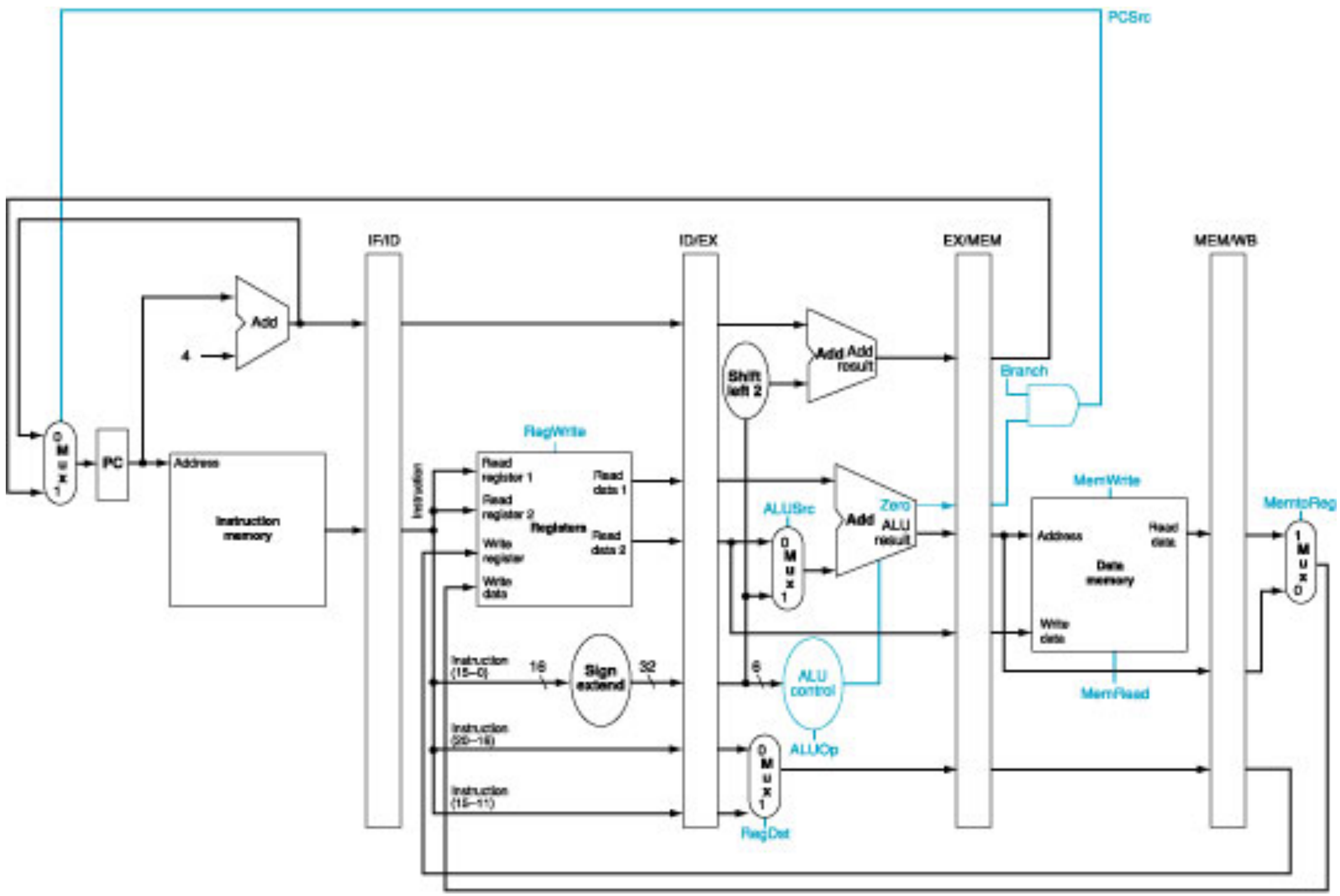
# Ex: Alt View



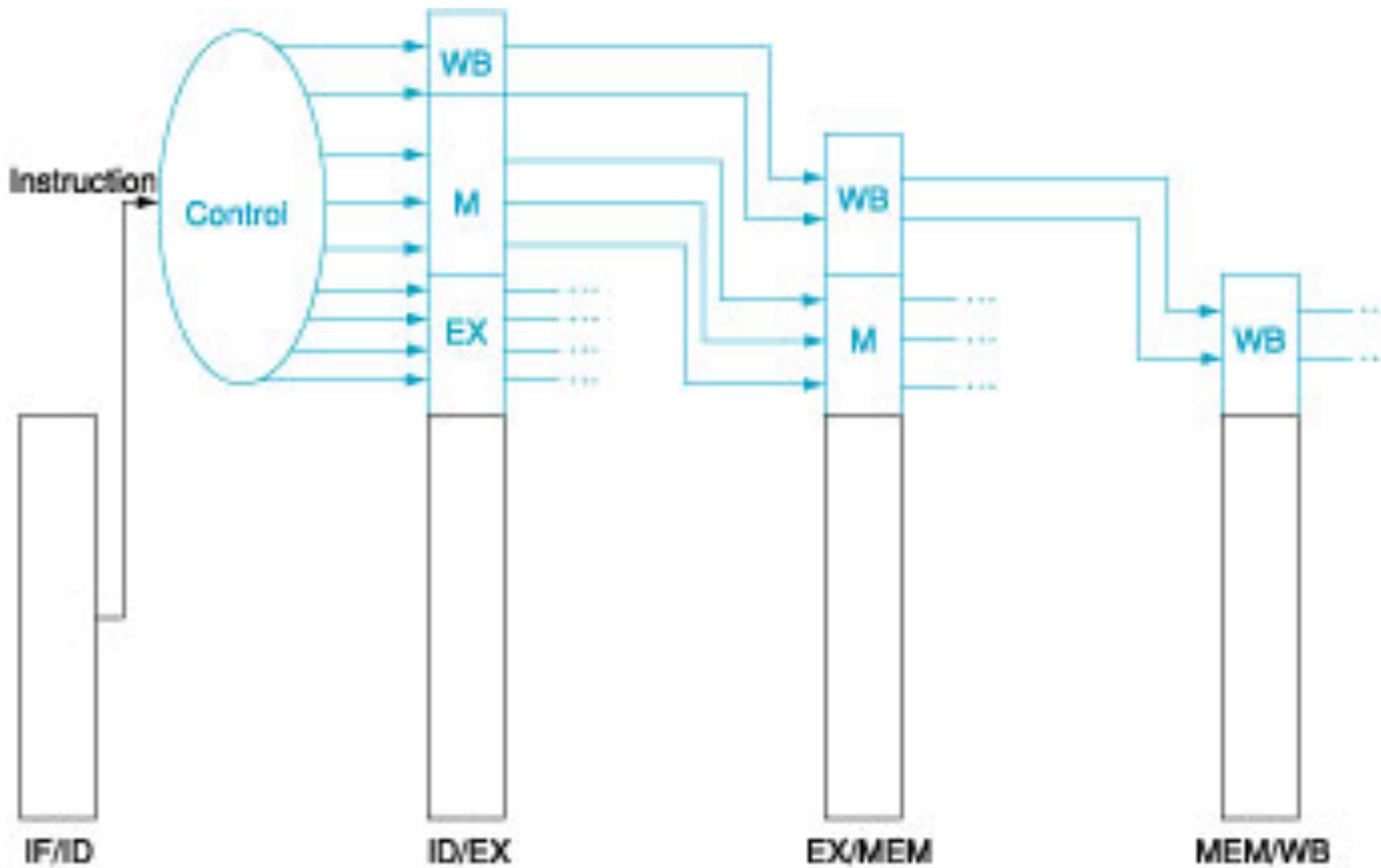
Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 1
CC 2
CC 3
Program execution order (in instructions)
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
Instruction fetch
Instruction decode
Execution
Data access
Write back

# Pipeline Control

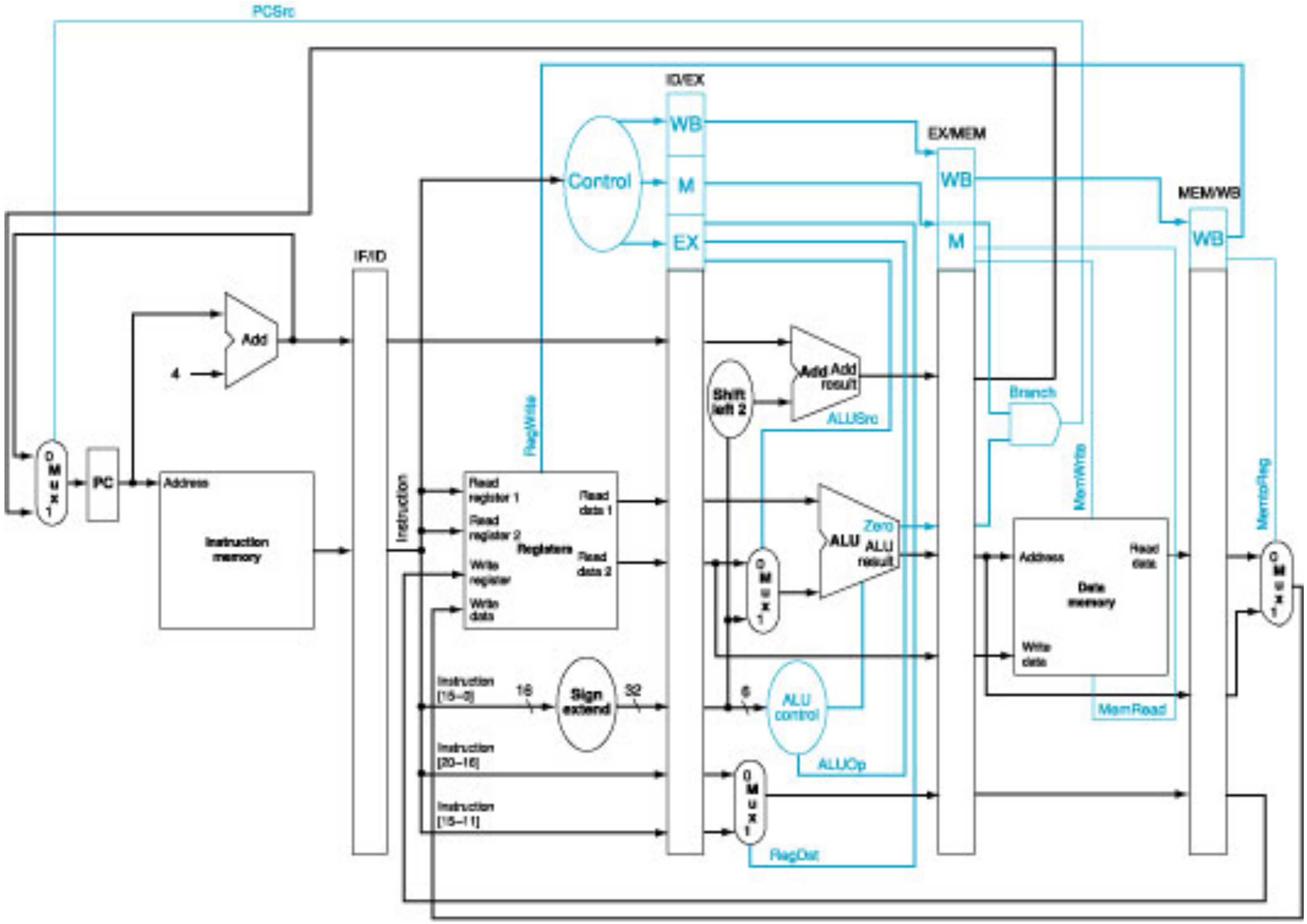
# Pipelined DP w/ signals



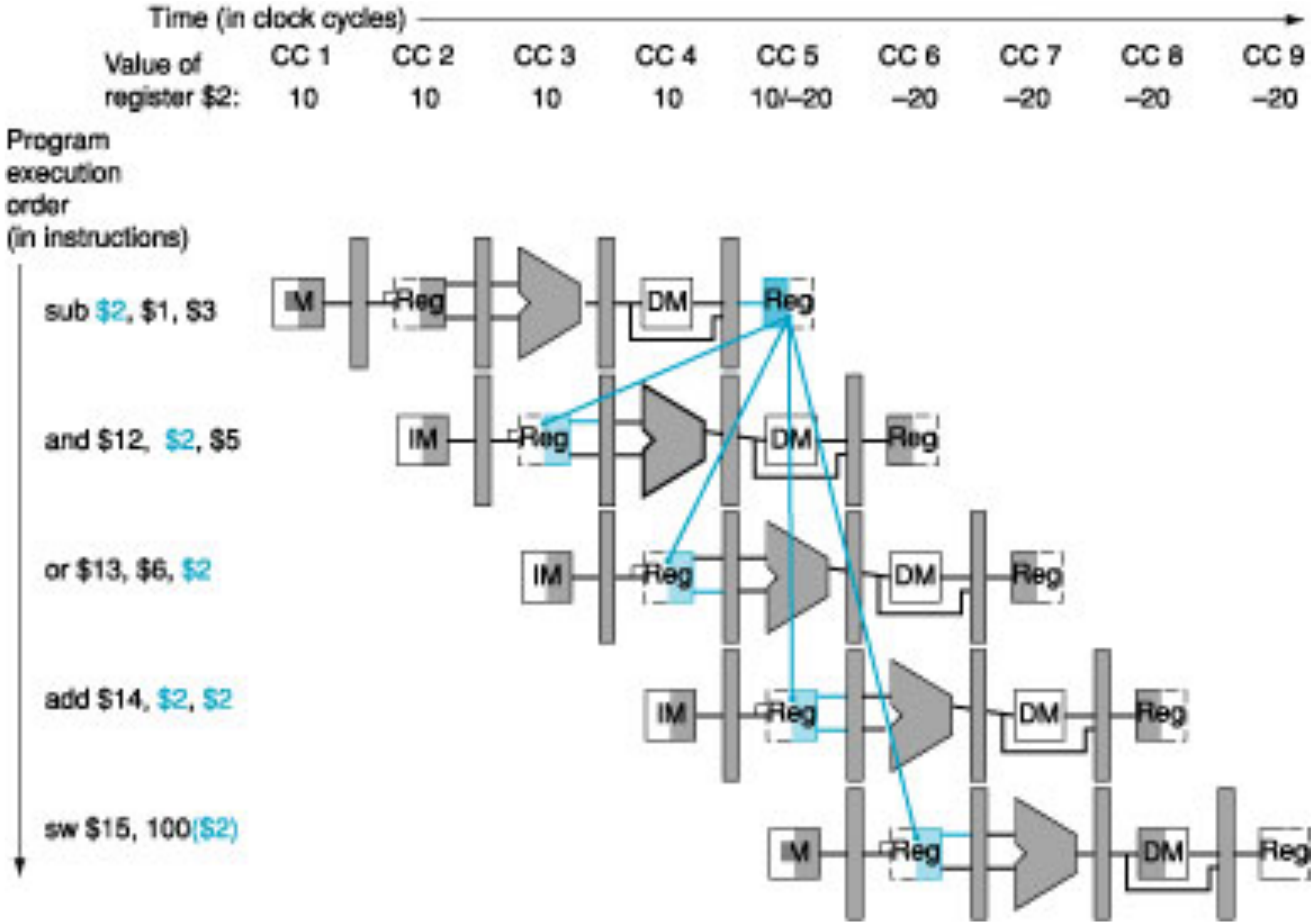
# Control lines for pipeline stages



# Pipelined DP w/ Control



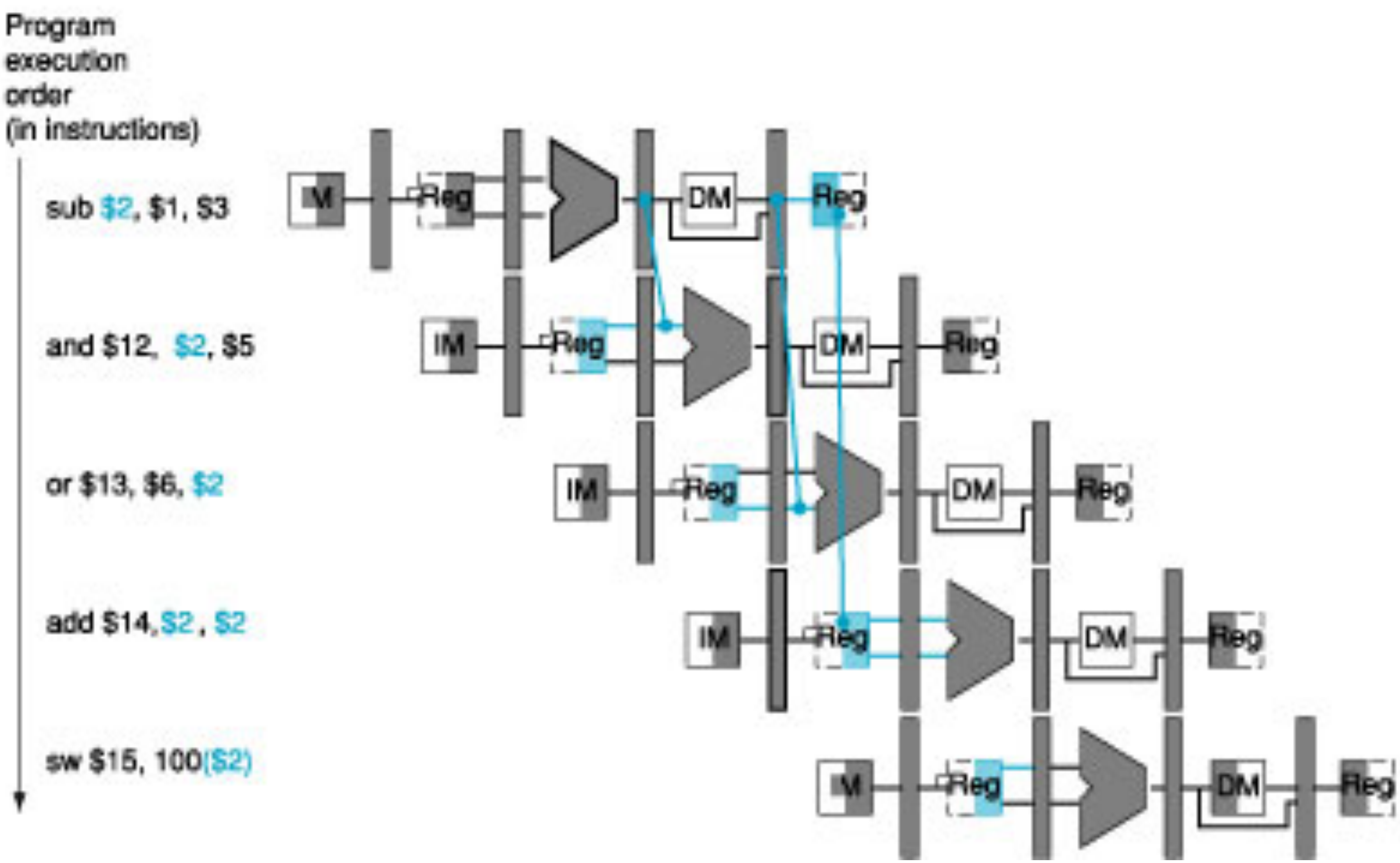
# Pipelined Dependencies



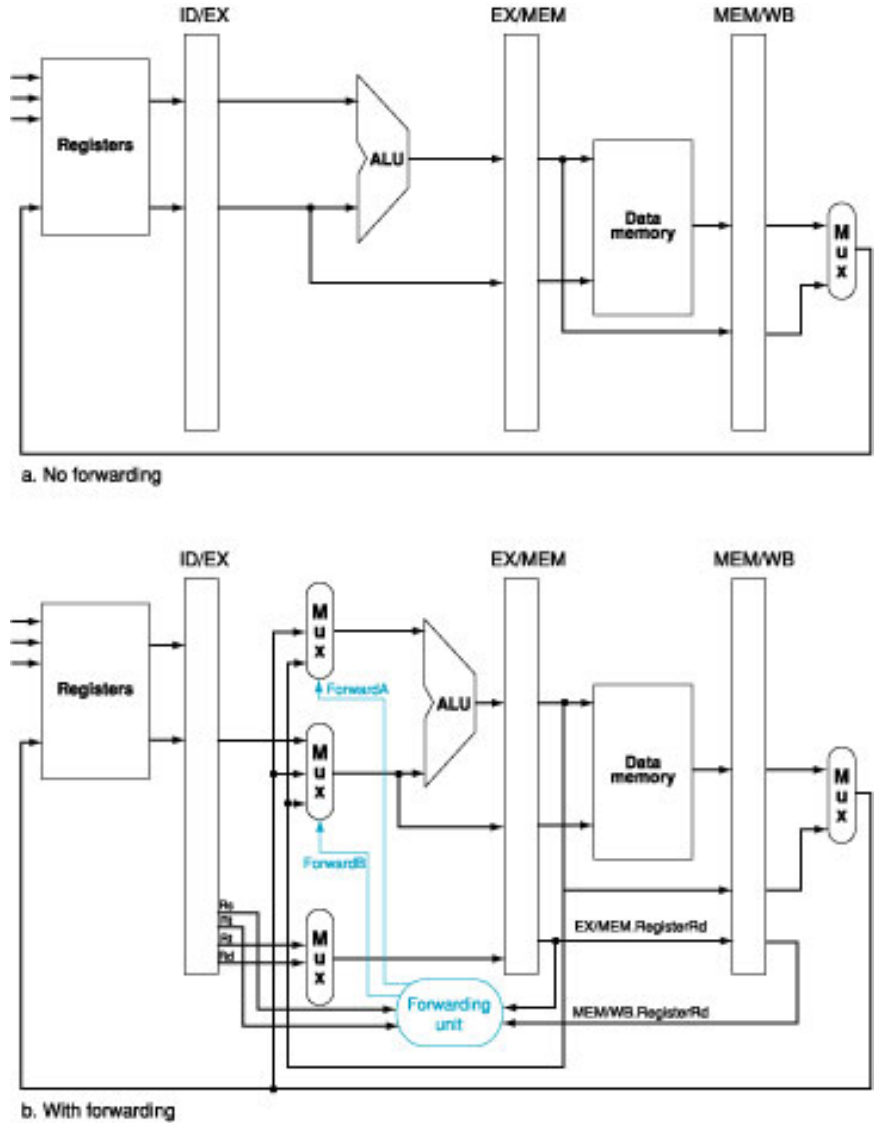
# Pipeline w/ Forwarding Values

Time (in clock cycles) →

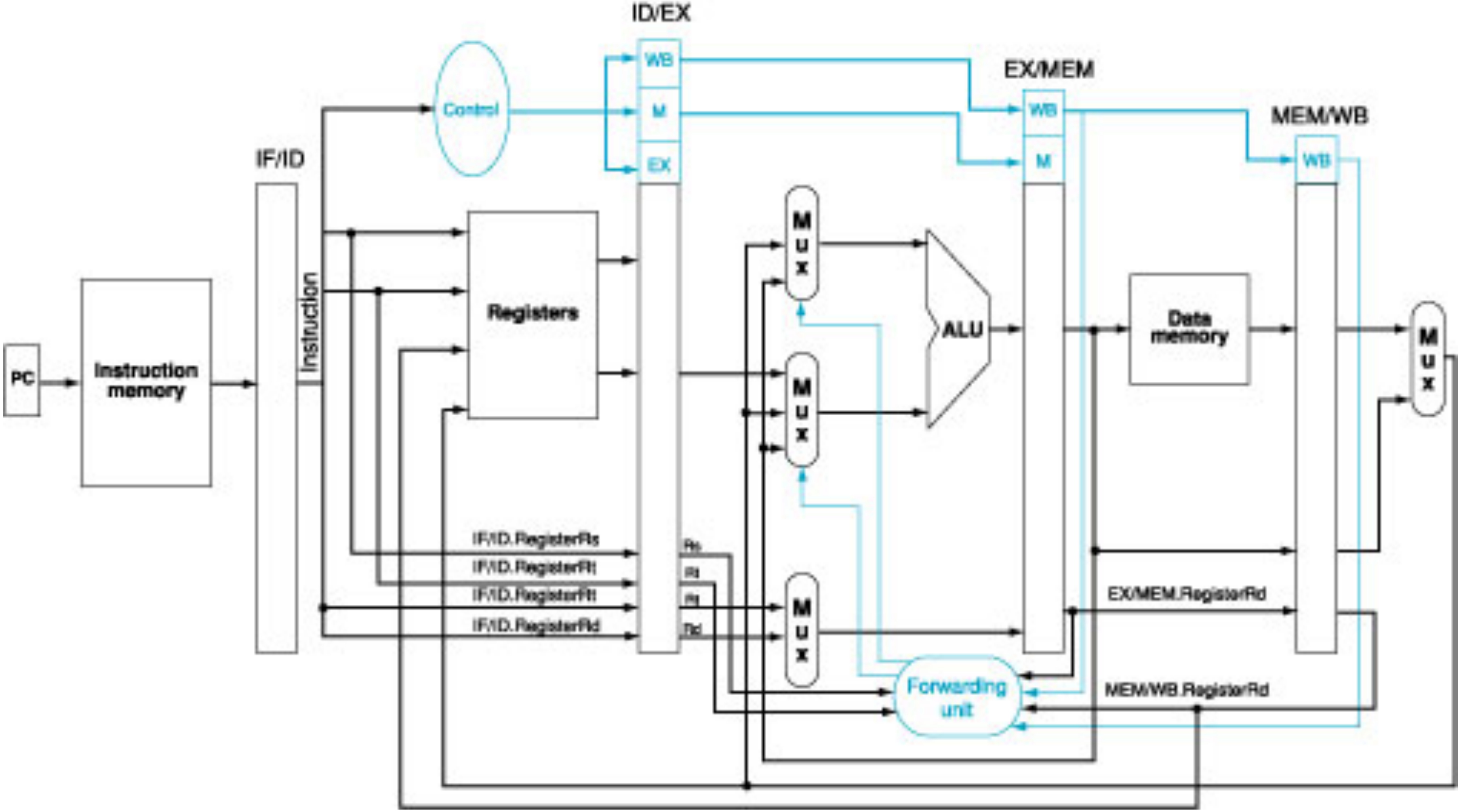
| | CC 1 | CC 2 | CC 3 | CC 4 | CC 5 | CC 6 | CC 7 | CC 8 | CC 9 |
|---|---|---|---|---|---|---|---|---|---|
| Value of register $2: | 10 | 10 | 10 | 10 | 10/–20 | –20 | –20 | –20 | –20 |
| Value of EX/MEM: | X | X | X | –20 | X | X | X | X | X |
| Value of MEM/WB: | X | X | X | X | –20 | X | X | X | X |



# ALU & Regs: B4, After Fwding



# Datapath w/ forwarding



# Forwarding Control Table

| MUX Control | Source | Reason |
|---|---|---|
| ForwardA = 00 | ID/EX | 1st ALU op from reg file |
| ForwardA= 10 | EX/MEM | 1st ALU op fwd from prior ALU result |
| ForwardA = 01 | MEM/WB | 1st ALU op fwd from data mem or earlier result |

# Forwarding Control Table (cont.)

| MUX Control | Source | Reason |
|---|---|---|
| ForwardB = 00 | ID/EX | 2nd ALU op from reg file |
| ForwardB= 10 | EX/MEM | 2nd ALU op fwd from prior ALU result |
| ForwardB = 01 | MEM/WB | 2nd ALU op fwd from data mem or earlier result |

# Resolution

- if( EX/MEM.RegWrite &&
  EX/MEM.RegisterRd != 0 &&
  EX/MEM.RegisterRd ==
  ID/EX.RegisterRs )

  ForwardA = 10
- if( EX/MEM.RegWrite &&
  EX/MEM.RegisterRd != 0 &&
  EX/MEM.RegisterRd ==
  ID/EX.RegisterRt )

  ForwardB = 10

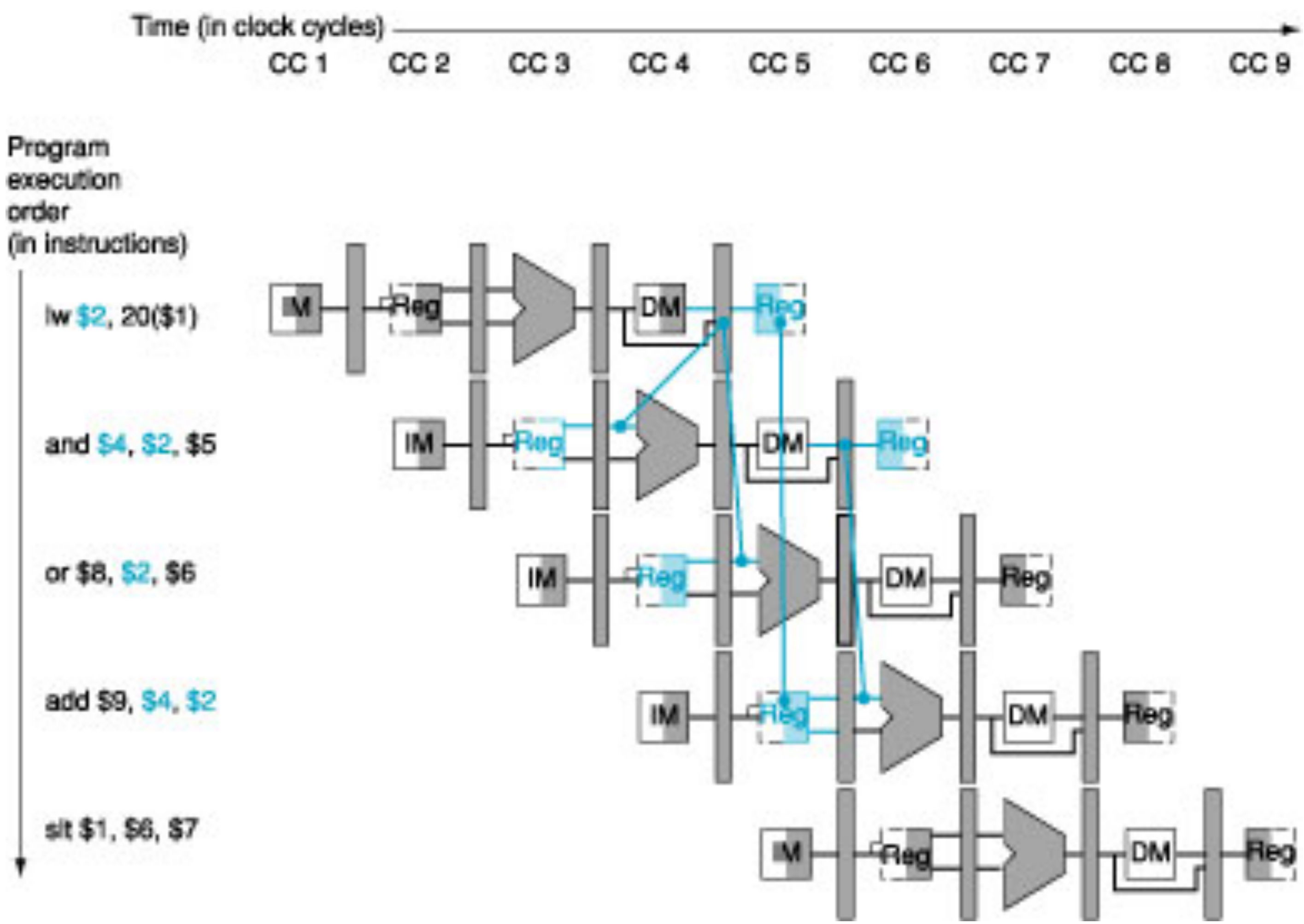
# Mem Stage Hazard Detection & Resolution

- if( MEM/WB.RegWrite &&
  MEM/WB.RegisterRd != 0 &&
  **EX/MEM.RegisterRd != ID/EX.RegisterRs** &&
  MEM/WB.RegisterRd = ID/EX.RegisterRs)
  ForwardA = 01
- if( MEM/WB.RegWrite &&
  MEM/WB.RegisterRd != 0 &&
  **EX/MEM.RegisterRd != ID/EX.RegisterRt** &&
  MEM/WB.RegisterRd = ID/EX.RegisterRt)
  ForwardB = 01

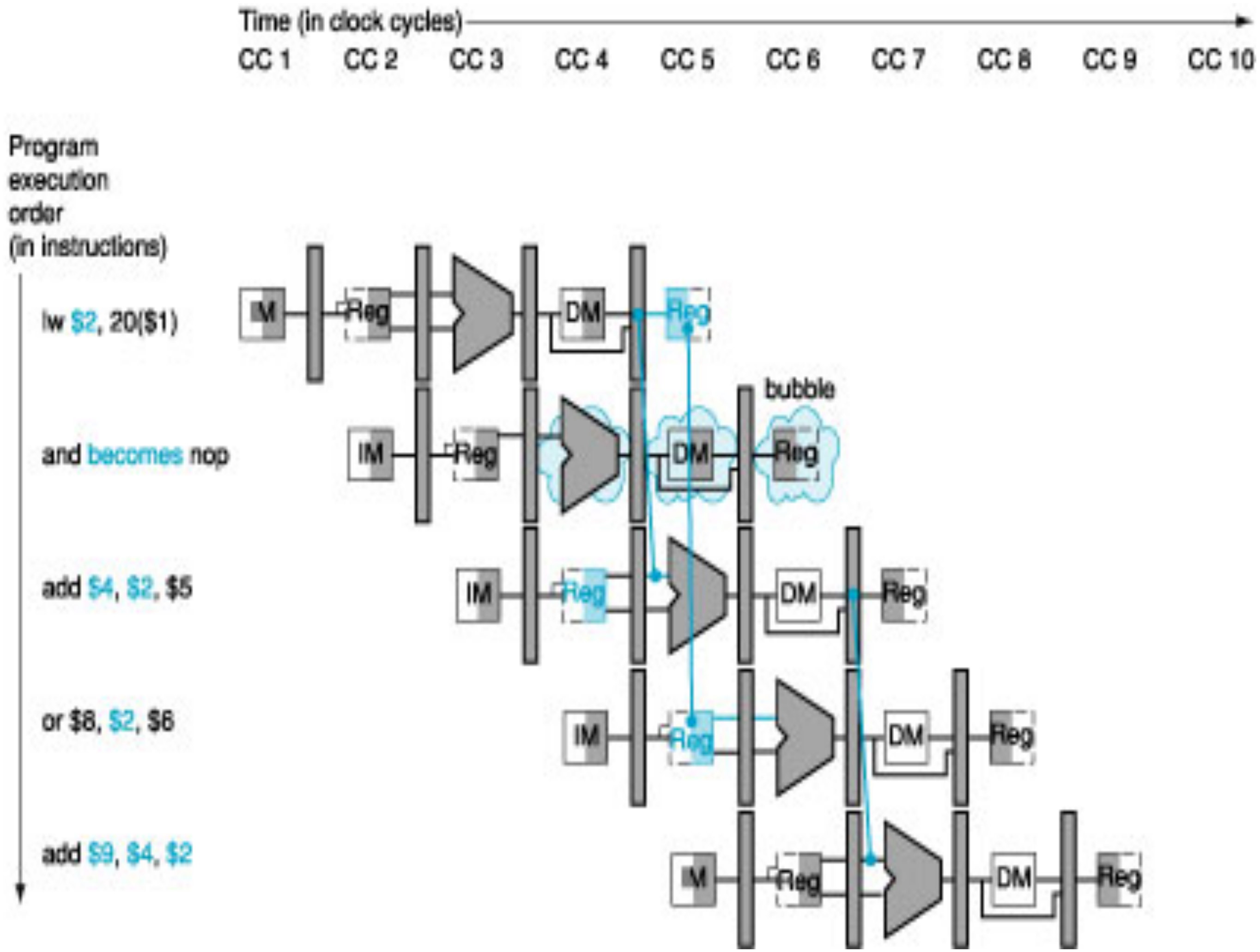
# Data Hazards & Stalls

- Need Hazard detection unit in addition to forwarding unit.
- Check for Load Instructions based on…
  - if( ID/EX.MemRead && (ID/EX.RegisterRt==IF/ID.RegisterRs || ID/EX.RegisterRt==IF/ID.RegisterRt)) StallThePipeline

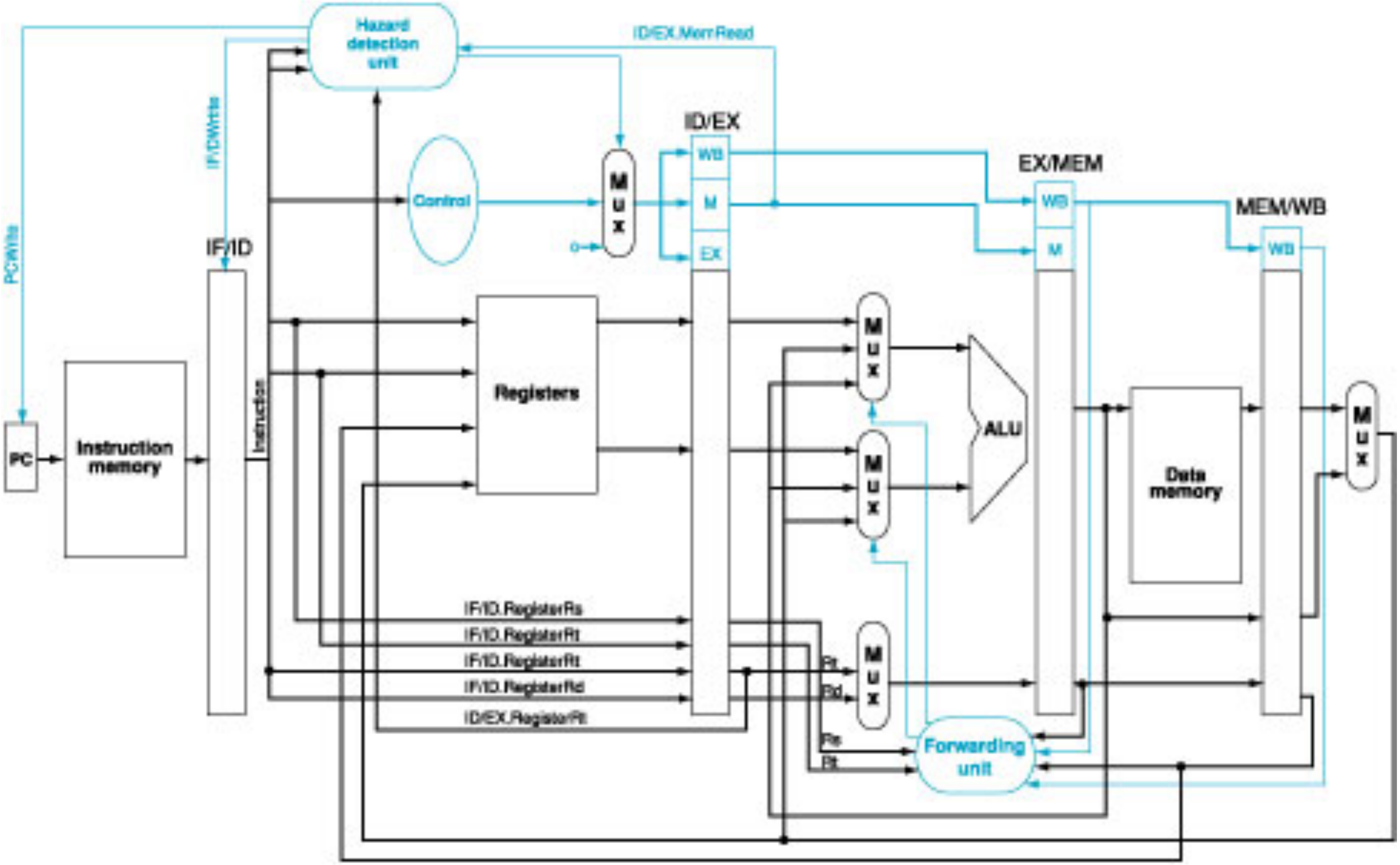
# Where Forwarding Fails…must stall



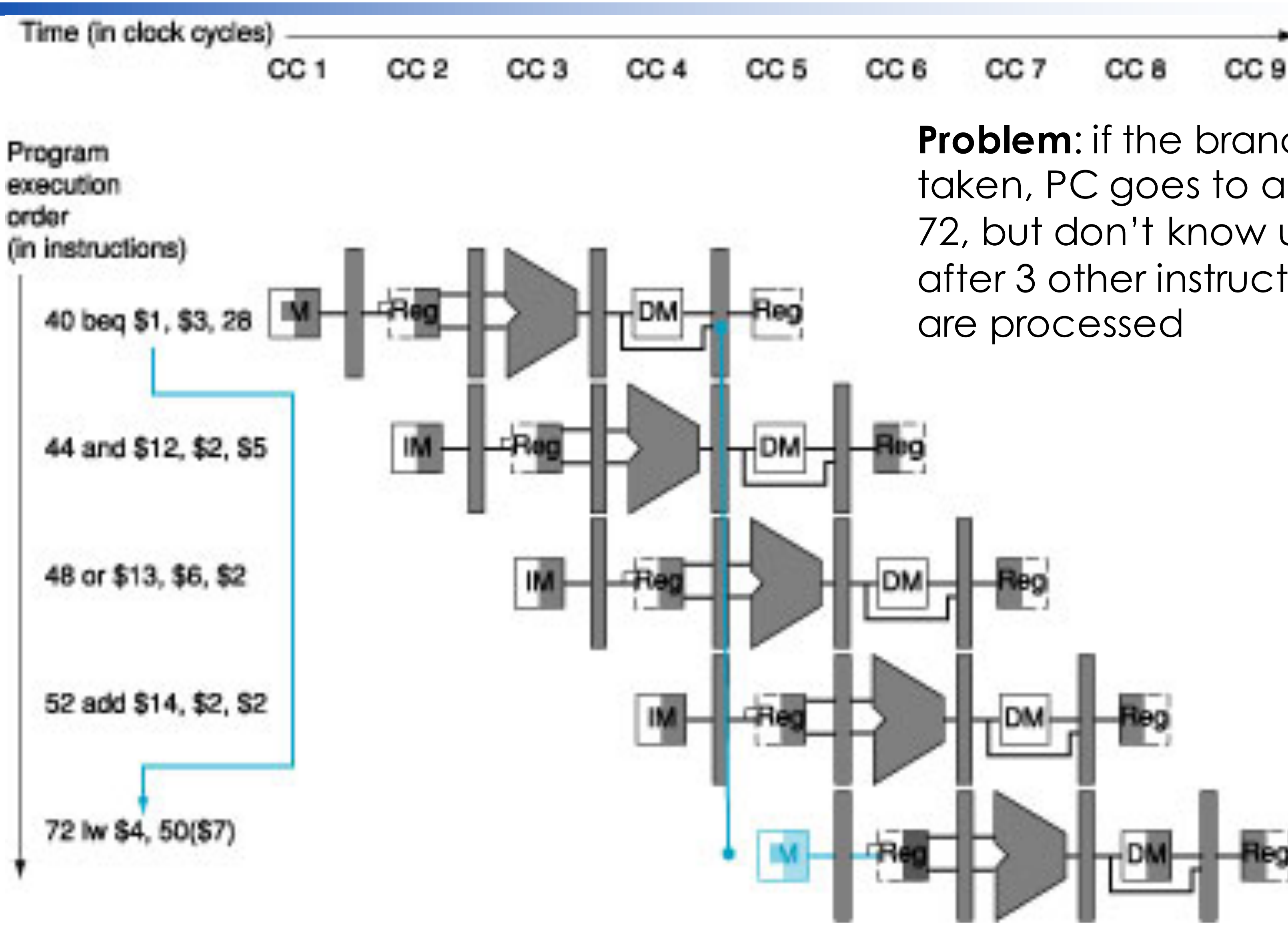
# How Stalls Are Inserted



# Pipelined control w/ fwding & hazard detection



# What about those crazy branches?




**Problem**: if the branch is taken, PC goes to addr 72, but don't know until after 3 other instructions are processed

# Branch Hazards: Assume Branch Not Taken

- Recall stalling until branch is complete is too sssssslllllooooowwww!!
- So, assume the branch is not taken…
- If taken, instructions fetched/decoded must be discarded or "squashed"
- discard instructions, just change the original control values to 0's (similar to load-use hazard),
- BIG DIFFERENCE: must flush the pipeline in the IF, ID and EX stages
- How can we reduce the "flush" costs when a branch is taken?

# Reducing the Delay of Branches

- Let's move the branch execution earlier in the pipeline.
- EFFECT: fewer instructions need to be flushed.
- NEED two actions:
  - Compute branch target address (EASY - can do on IF/ID stage).
  - Eval of branch decision (HARD)

# Faster Branch Decision

- Recall, for BEQ instruction, we would compare two regs during the ID stage and test for equality.
- Equality can be tested by XORing the two regs. (a.k.a. equality unit)
- Need additional ID stage forwarding and hazard detection hardware
- This has 2 complicating factors…

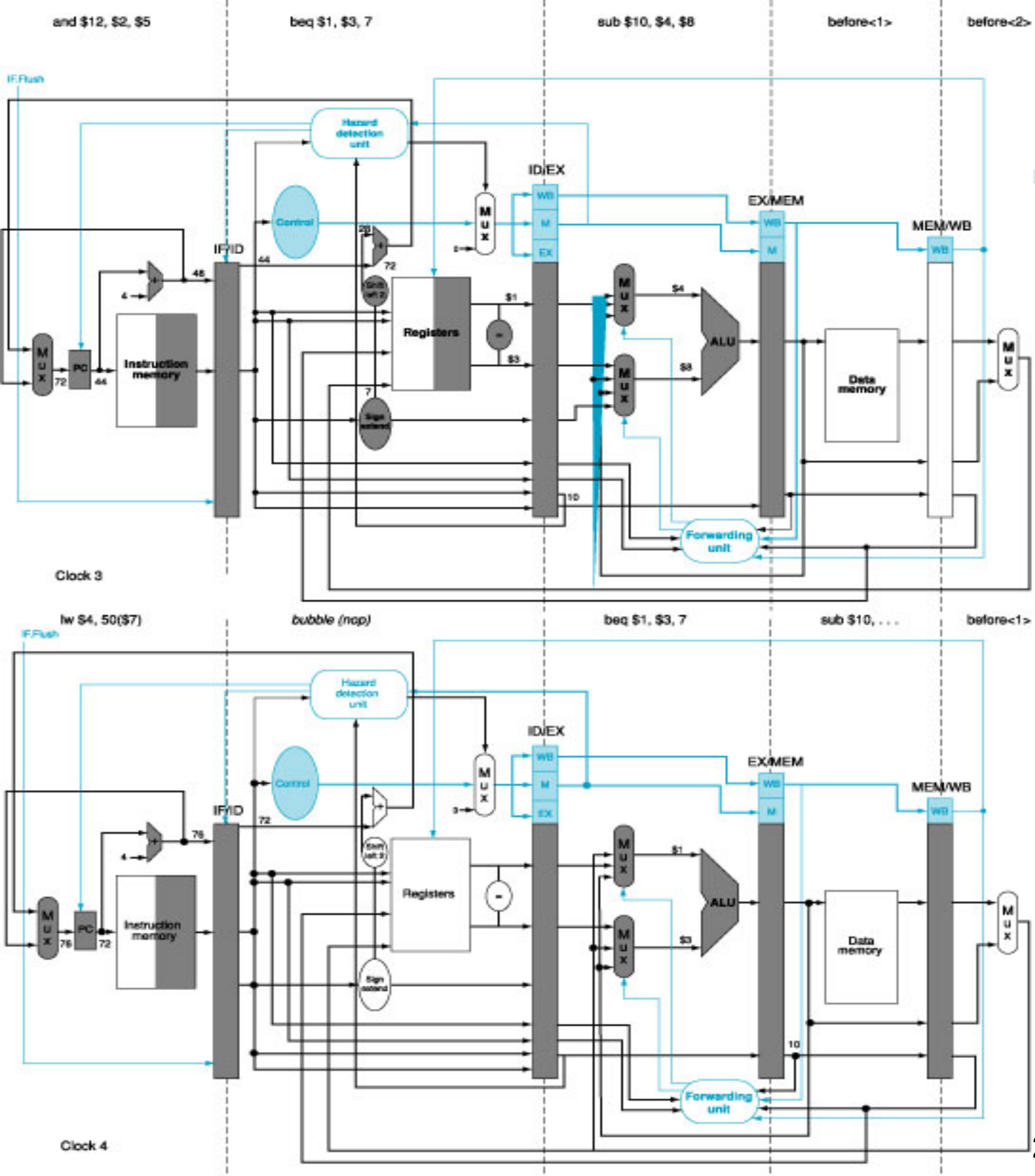
# Faster Branch Decison: Complex Factors

1. In ID stage, now we need to decide whether a “bypass” path to the “equality” unit is needed.
    - ALU forwarding logic is not sufficient, and so we need new forwarding logic for the equality unit.
2. Can stall due to a data hazard.
    - if an r-type instruction comes before the branch who operands are used in the comparision in the branch, a stall is needed

# Example Pipelined Branch

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

# Branch Processing Example



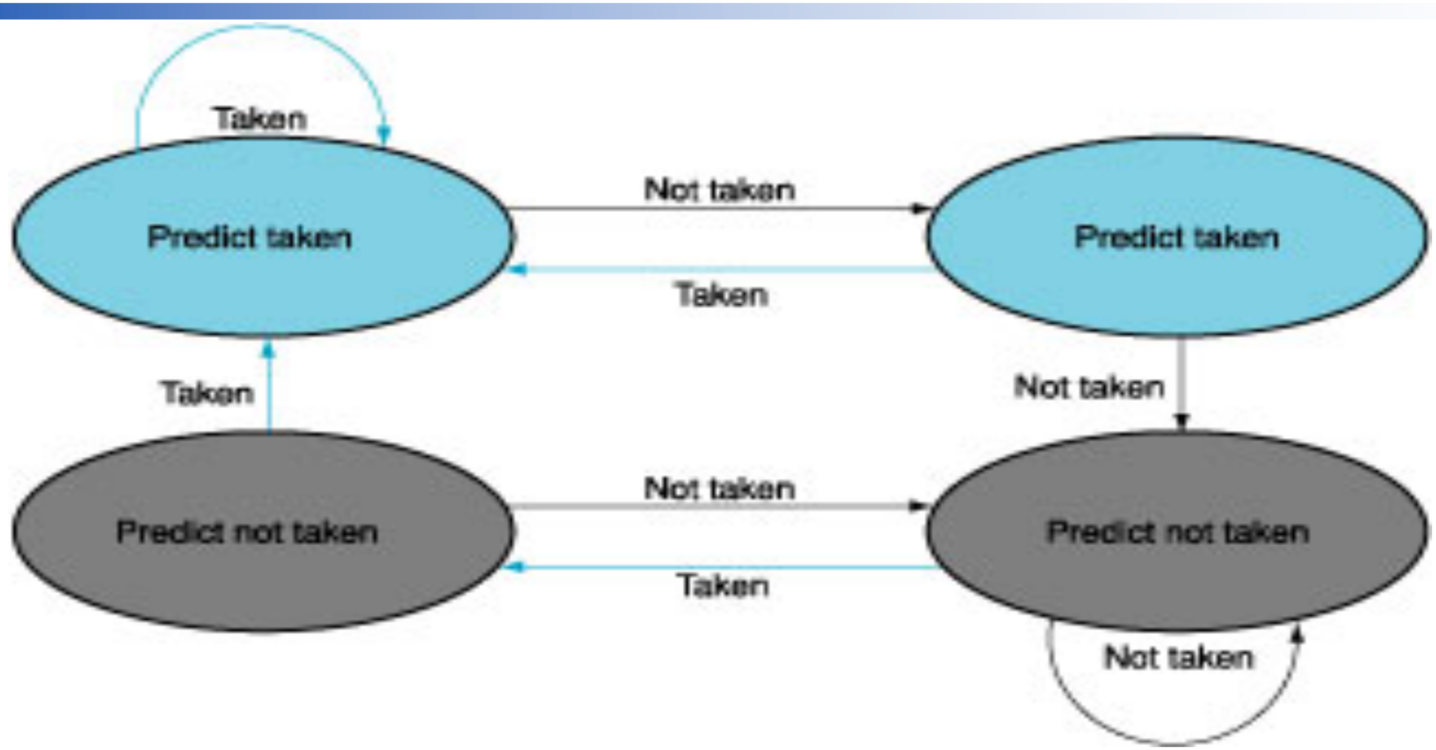
# Dynamic Branch Prediction

- From the phase "There is no such thing as a typical program", this implies that programs will branch is different ways and so there is no "one size fits all" branch algorithm.
- **Alt approach**: keep a history (1 bit) on each branch instruction and see if it was last taken or not.
- **Implementation:** branch prediction buffer or branch history table.
  - Index based on lower part of branch address
  - Single bit indicates if branch at address was last taken or not. (1 or 0)

# Problem with 1-bit Branch Predictors

- Consider a loop branch
  - Suppose it occurs 9 times in a row, then is not taken.
  - What's the branch prediction accuracy?
  - ANSWER: 1-bit predictor will mispredict the entry and exit points of the loop.
  - Yields only an 80% accuracy when there is potential for 90% (i.e., you have to guess wrong on the exit of the loop).

# Solution: 2-bit Branch Predictor




Must be wrong twice before changing prediction
Learns if the branch is more biased towards "taken" or "not taken"

# Performance: Single vs Multicycle vs. PL

- Assume: 200 ps for memory access, 100 ps for ALU ops, 50 ps for register access
- Single-cycle clock cycle:
  - **600 ps**: 200 + 50 + 100 + 200 + 50
- Futher assume instruction mix
  - 25% loads, 10% stores, 11% branches, 2% jumps, 52% ALU instructions
  - Assume CPI for multi-cycle is 3.50
  - Multicycle clock cycle: must be longest unit which is 200 ps
  - Total time for an "avg" instruction is 3.5 * 200 ps = **700ps**

# Pipeline performance (cont)

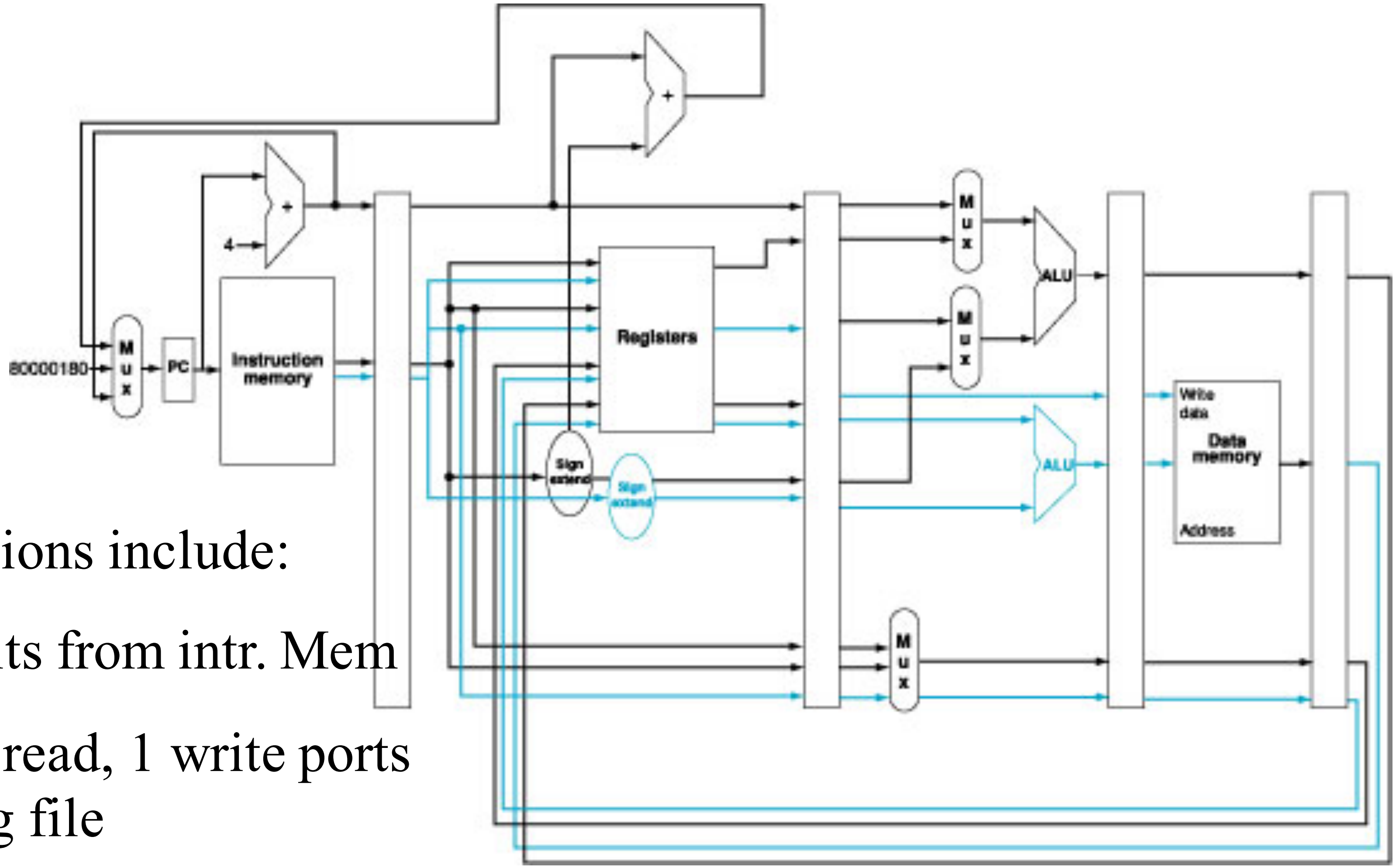
- For pipelined design…
  - Loads take 1 cycle when no load-use dependence and 2 cycles when there is yielding an average of 1.5 cycles per load.
  - Stores and ALU instructions take 1 cycle.
  - Branches take 1 cycle when predicted correctly and 2 cycles when not. Assume 75% accuracy, average branch cycles is 1.25.
  - Jumps are 2 cycles.
  - Avg CPI then is:

    1.5 x 25% + 1 x 10% + 1 x 52% + 1.25 x 11% + 2 x 2% = **1.17**
  - **Longest stage is 200 ps, so 200 x 1.17 = 234 ps**

# Even more performance…

- Ultimately we want greater and greater Instruction Level Parallelism (ILP)
- How?
- Multiple instruction issue.
  - Results in CPI's less than one.
  - Here, instructions are grouped into "issue slots".
  - So, we usually talk about IPC (instructions per cycle)
  - Static: uses the compiler to assist with grouping instructions and hazard resolution. Compiler MUST remove ALL hazards.
  - Dynamic: (i.e., superscalar) hardware creates the instruction schedule based on dynamically detected hazards

# Example Static 2-issue Datapath




Additions include:

•32 bits from intr. Mem

•Two read, 1 write ports on reg file

•1 more ALU (top handles address calc)

# Ex. 2-Issue Code Schedule

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

| | ALU/Branch | Data Xfer Inst. | Cycles |
|---|---|---|---|
| Loop: | | lw $t0, 0($s1) | 1 |
| | addi $s1, $s1, -4 | | 2 |
| | addu $t0, $t0, $s2 | | 3 |
| | bne $s1, $zero, Loop | sw $t0, 4($s1) | 4 |

***It take 4 clock cycles for 5 instructions or IPC of 1.25***

# More Performance: Loop Unrolling

- Technique where multiple copies of the loop body are made.
- Make more ILP available by removing dependencies.
- How? Complier introduces additional registers via "register renaming".
- This removes "name" or "anti" dependence
  - where an instruction order is purely a consequence of the reuse of a register and not a real data dependence.
  - Ex. lw $t0, 0($s1), addu $t0, $t0, $s2 and sw $t0, 4($s1)
  - No data values flow between one pair and the next pair
  - Let's assume we unroll a block of 4 interations of the loop..

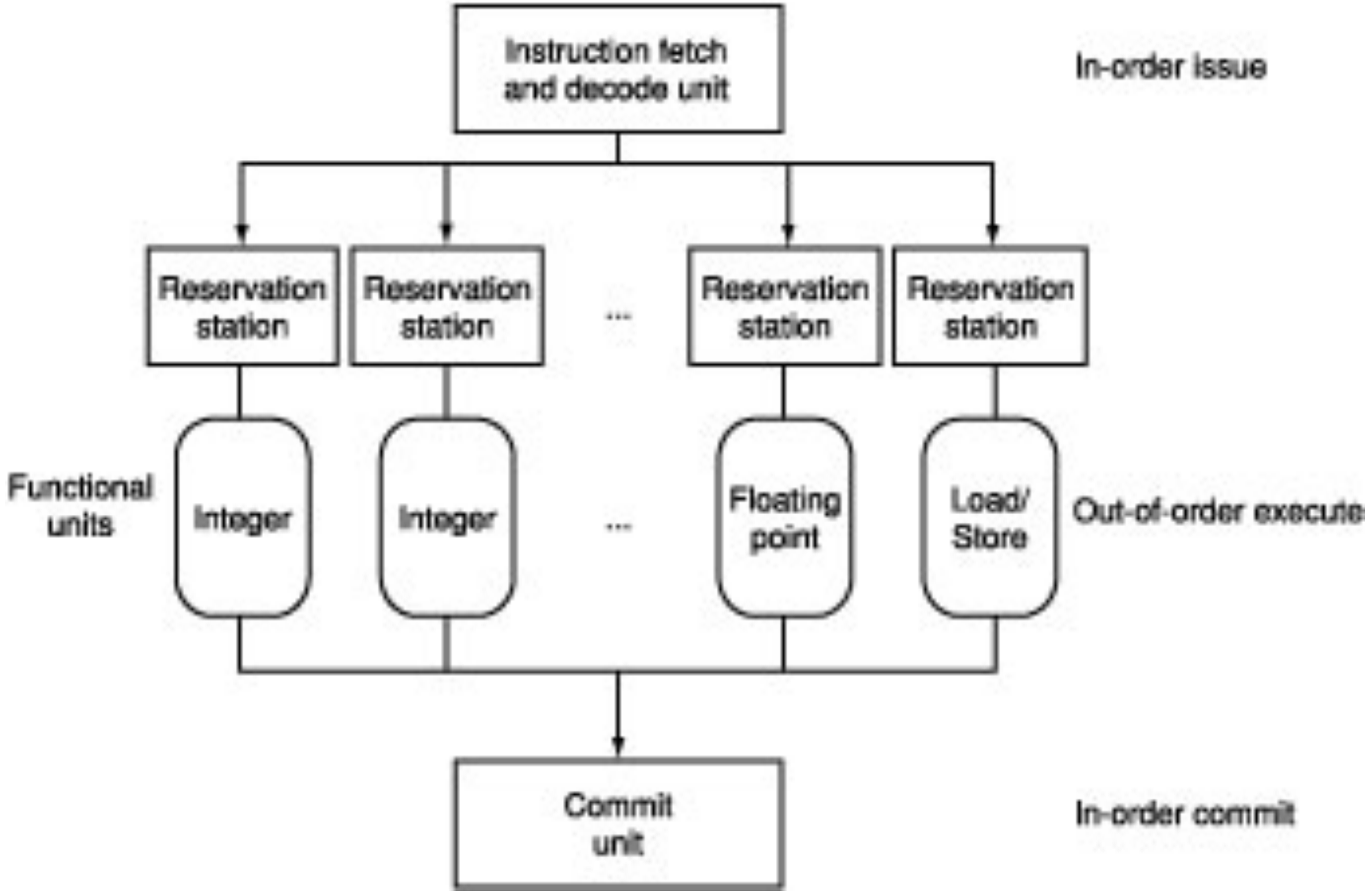
# Loop Unrolling Schedule

| | ALU/Branch Instructions | Data Xfer | Cycles |
|---|---|---|---|
| Loop | addi $s1, $s1, -16 | lw $t0, 0($s1) | 1 |
| | | lw $t1, 12($s1) | 2 |
| | addu $t0, $t0, $s2 | lw $t2, 8($s1) | 3 |
| | addu $t1, $t1, $s2 | lw $t3, 4($s1) | 4 |
| | addu $t2, $t2, $s2 | sw $t0, 16($s1) | 5 |
| | addu $t3, $t3, $s2 | sw $t1, 12($s1) | 6 |
| | | sw $t2, 8($s1) | 7 |
| | bne $s1, $zero, loop | sw $t3, 4($s1) | 8 |

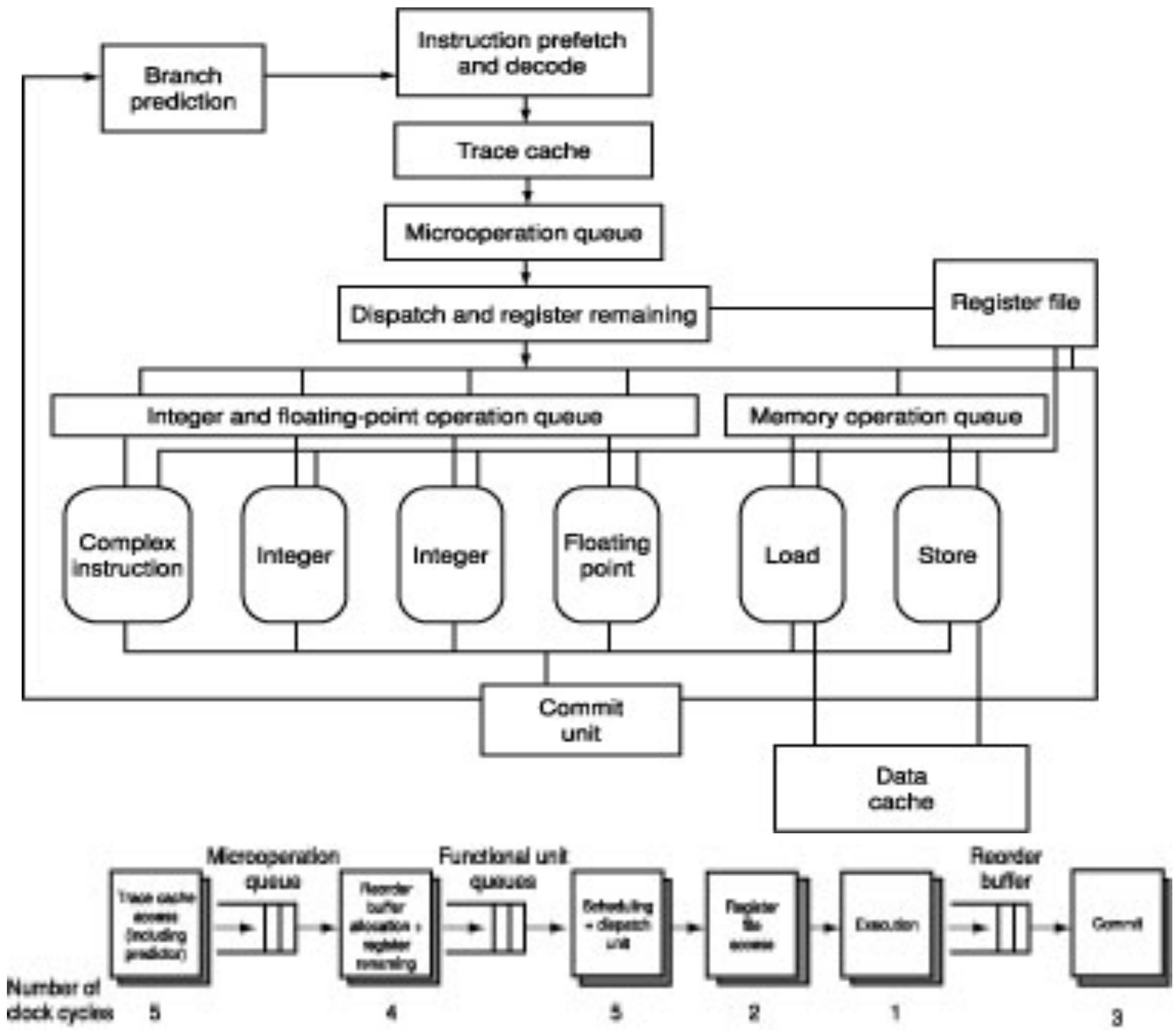
# Performance of Instruction Schedule

- 12 of 14 instructions execute in a pair.
- Takes 8 clock cycles for 4 loop iterations
- Yields 2 clock cycles per iteration
- CPI = 8/14 → 0.57
- Cost of improvement: 4 temp regs + lots of additional code

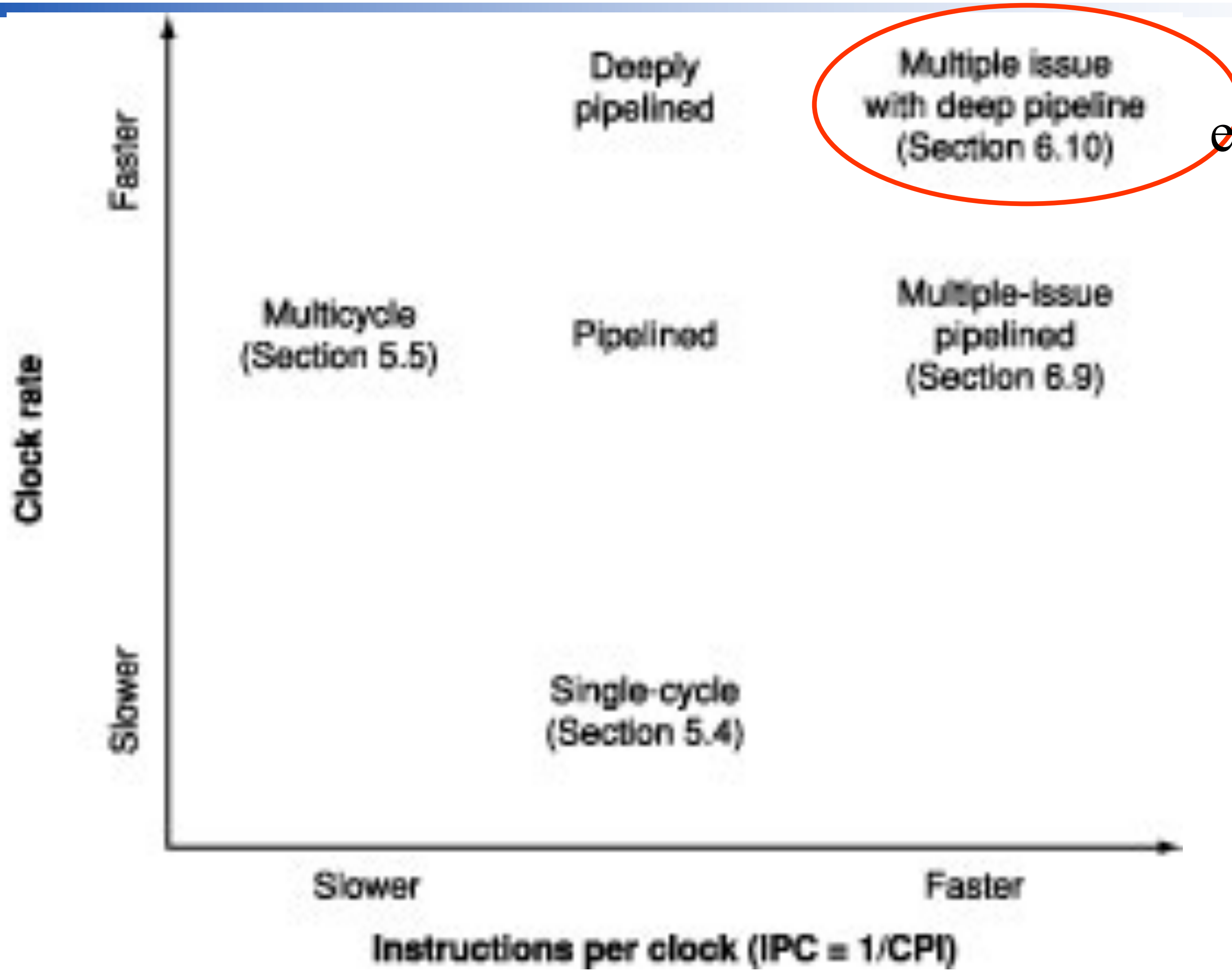
# Dynamic Scheduled Pipeline



# Intel P4 Dynamic Pipeline



# Summary of Pipeline Technology




We've exhausted this!!