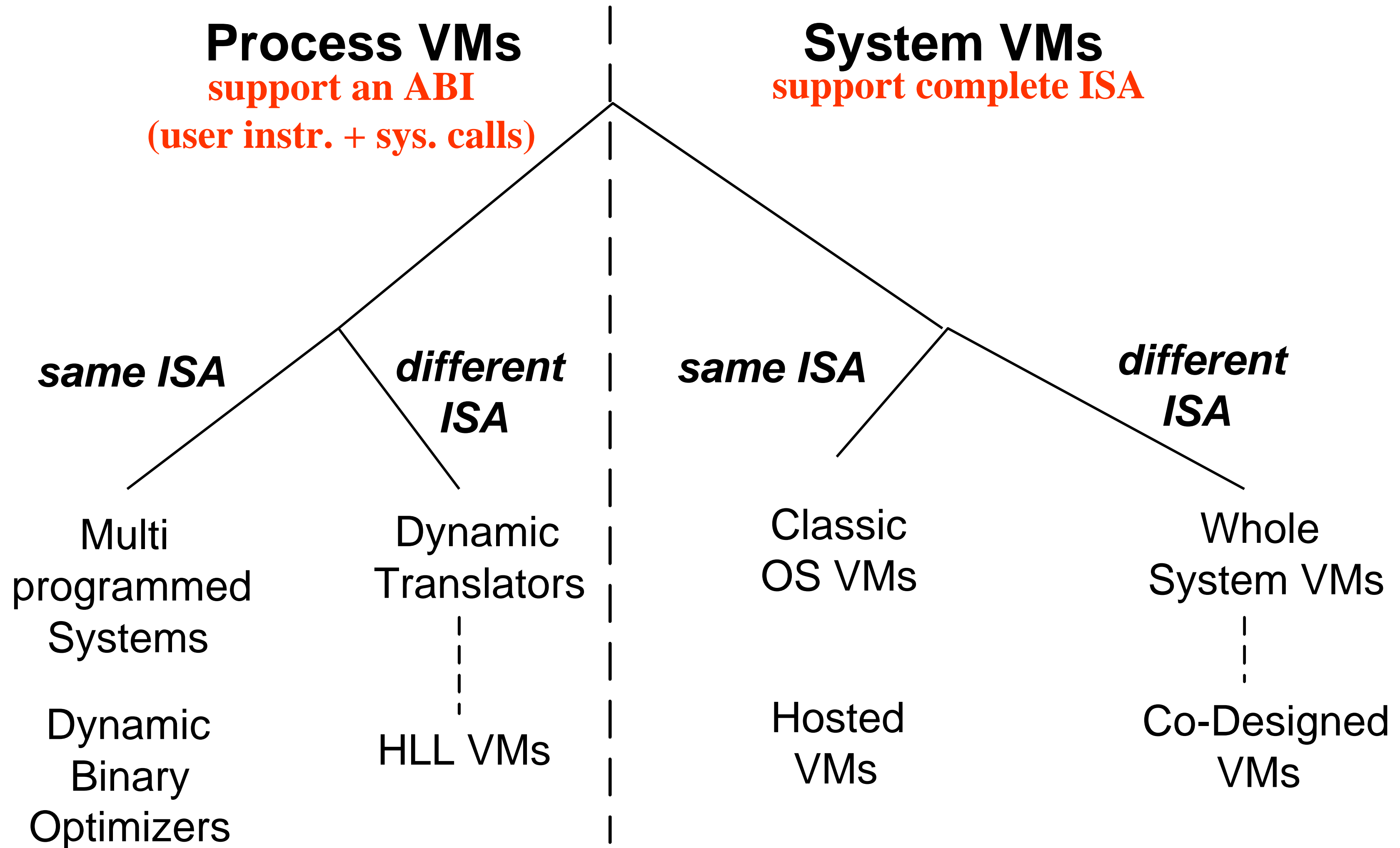
CS6270: Virtual Machines

Lecture 2: Background

Review of Basic Computer Architecture Concepts

Samarjit Chakraborty

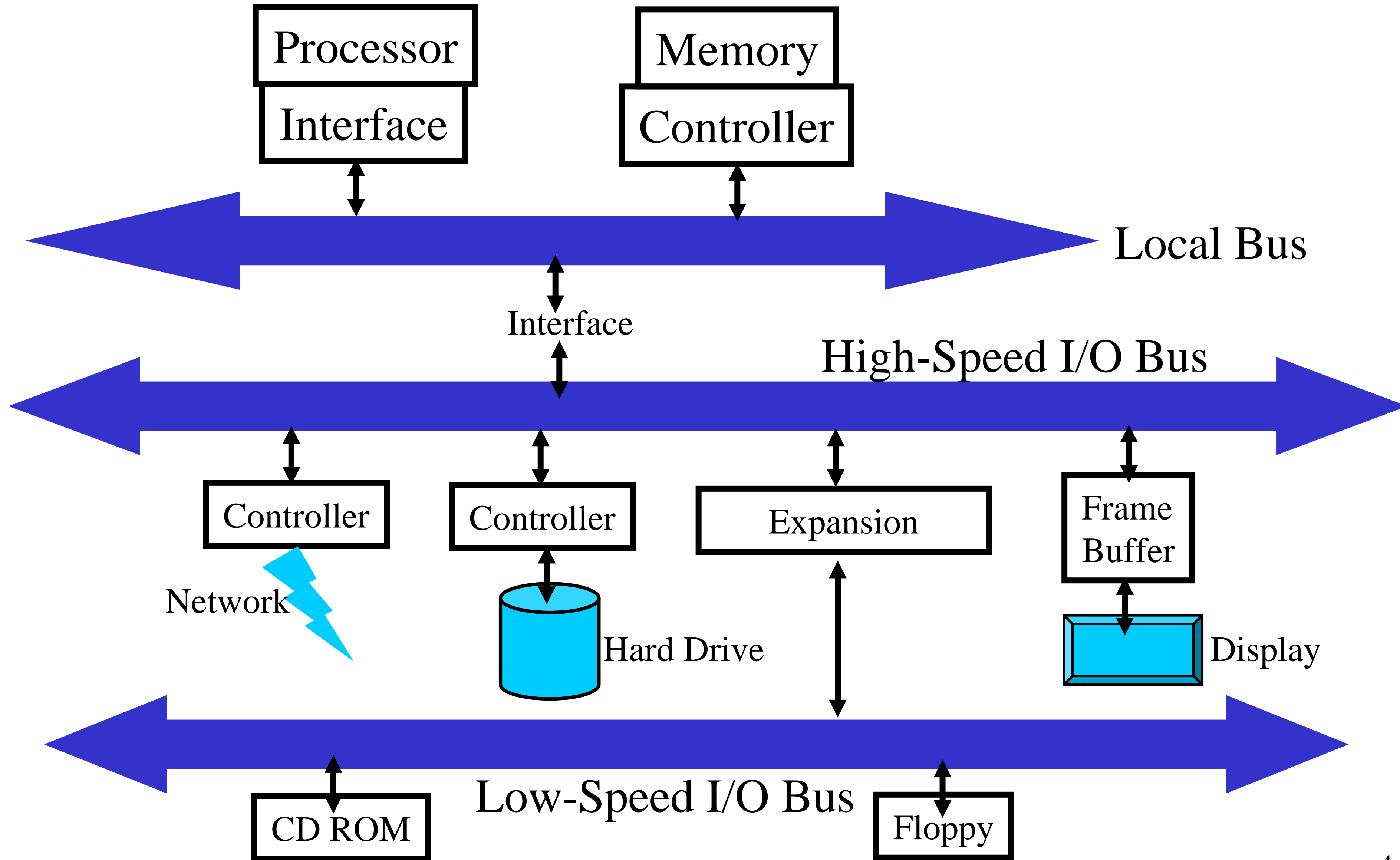
Last Week's Class: VM Taxonomy



Today: Review of Background Material

- Virtual machines essentially present an *interface* that is identical to some desired real machine
- Hence, it is important to understand the interfaces that real machines provide and how such interfaces are supported/implemented
- In particular, we will review concepts from
  - Computer architecture (today's class)
  - Operating systems

Computer System Hardware – Major Components



Basics of Processors

- We will use the MIPS instruction set to illustrate the basic concepts
  - This instruction set is used by NEC, Nintendo, Silicon Graphics, Sony, …
- MIPS fields

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *op*: Operation of the instruction (opcode)
- *rs*: First register source operand
- *rt*: Second register source operand
- *rd*: Register destination operand
- *shamt*: Shift amount
- *funct*: Function field (selects specific variant of opcode)

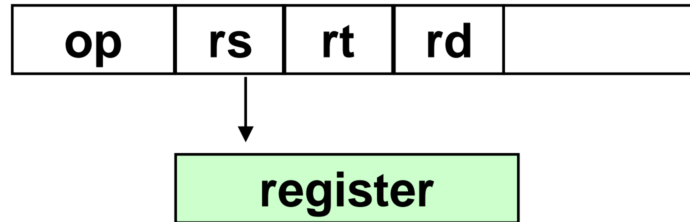
MIPS Operands: Registers and Memory

MIPS operands

Name	Examples	Comments
32 registers	\$s0-\$s7, \$t0-\$t9, \$zero, \$a0-\$a3, \$v0-\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Mem[0], Mem[4], …, Mem[4294967292].	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

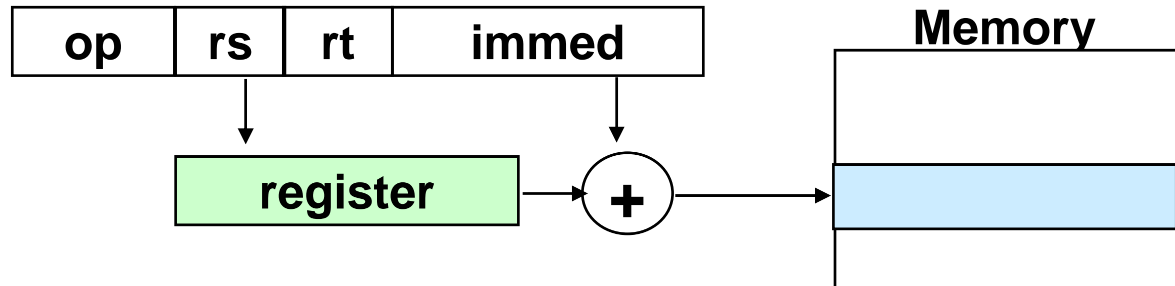
MIPS: Addressing Modes

Register (direct)

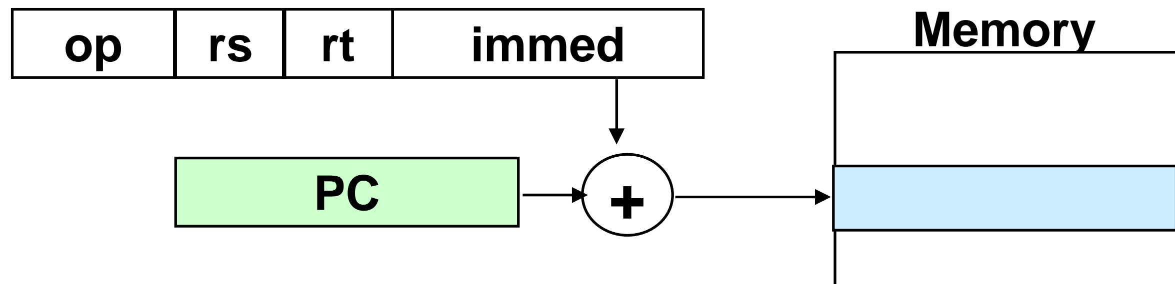


Immediate

Displacement

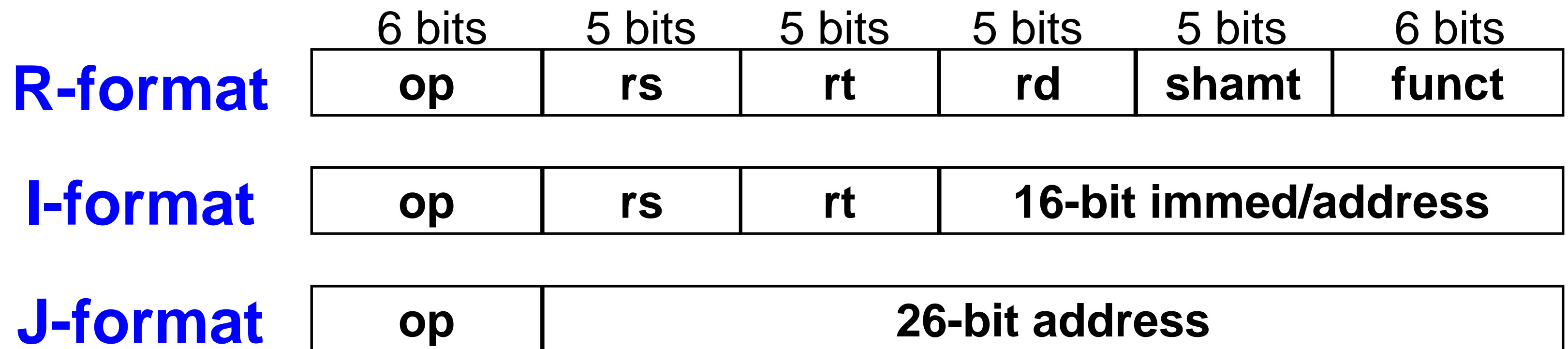


PC-relative



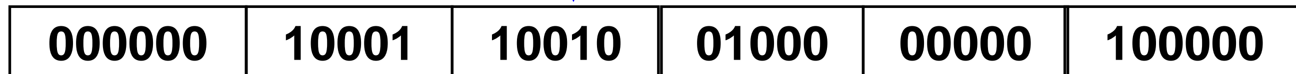
MIPS: Instruction Format

- Fixed-length instruction format
- All instructions are 32-bit long
- Very structured
- Only three instruction formats: R, I, J

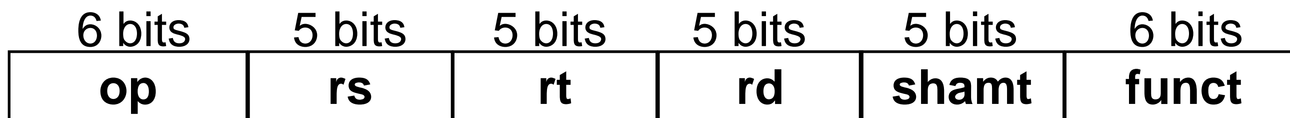


MIPS: Instruction Format (Contd.)

- **R-format**: Used for instructions with 3 register operands
- Arithmetic instructions:
  - **add $t0, $s1, $s2**   # $t0 ← $s1 + $s2
  - Note that $t0 is register 8, $s1 is register 17 and $s2 is register 18.

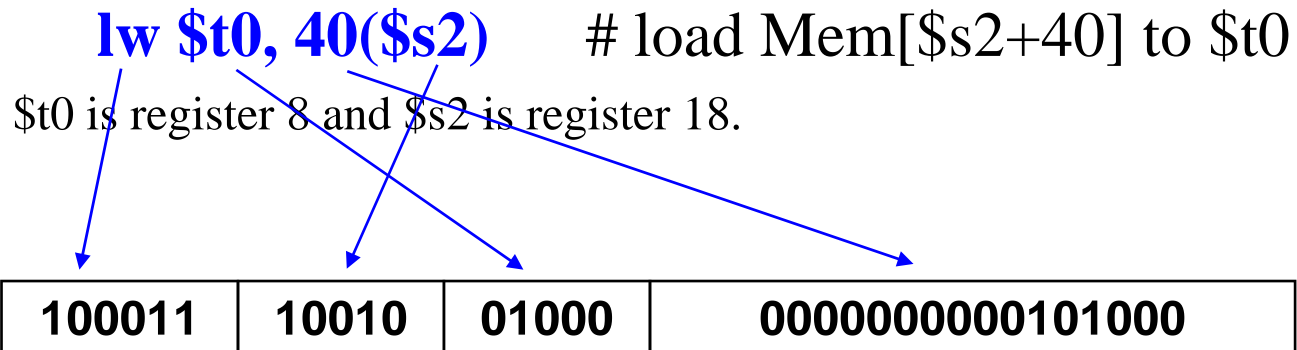


R-format



MIPS: Instruction Format (Contd.)

- **I-format**: For data transfer instructions
- Examples: load word (lw) and store word (sw)
  - One register operand and one memory address operand (specified by a constant and a register)

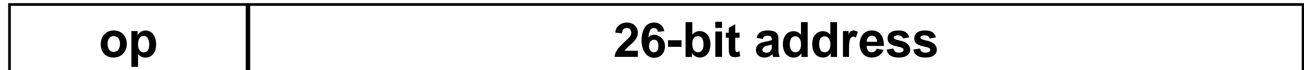


I-format

MIPS: Instruction Format (Contd.)

- **J-format**: For jump instructions
  - **j** **Label** # next instr. at Label
- Formats:

J-format



- Jump instructions just use high-order bits of PC
  - Address = bits 31-28 of PC + shift_left_2_bits(26-bit address)
  - Address boundaries of 256 MB.

Execution Time of a Program - Factors

- Instruction Count
  - Determined by compiler and ISA
- Clock cycle time
  - Determined by the architecture/implementation of the ISA
- Number of Clock Cycles per Instruction (CPI)
  - Determined by the architecture/implementation of the ISA
- We will now look at different possible implementation possibilities

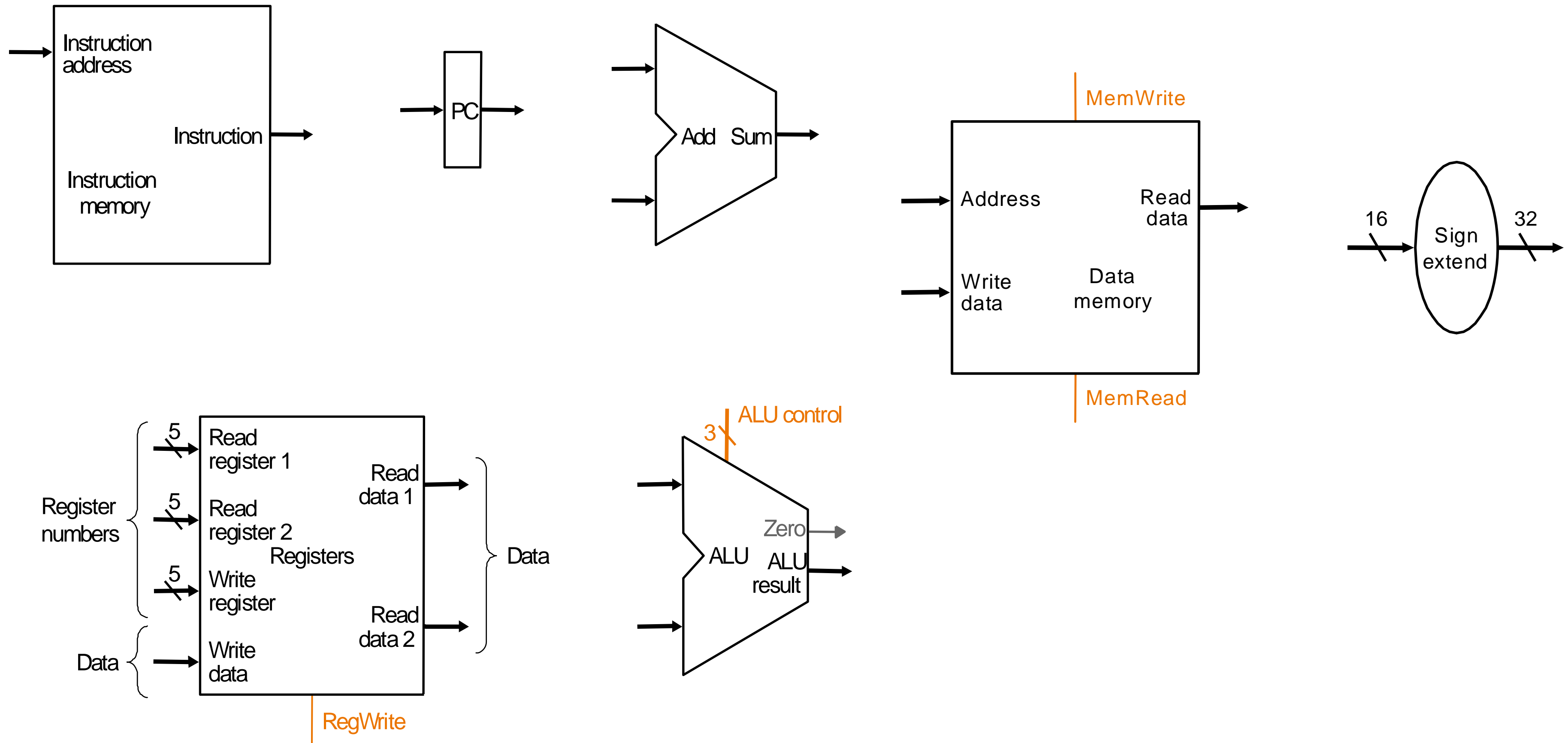
The Processor: Datapath & Control

- Implementation of the MIPS ISA
- Simplified 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

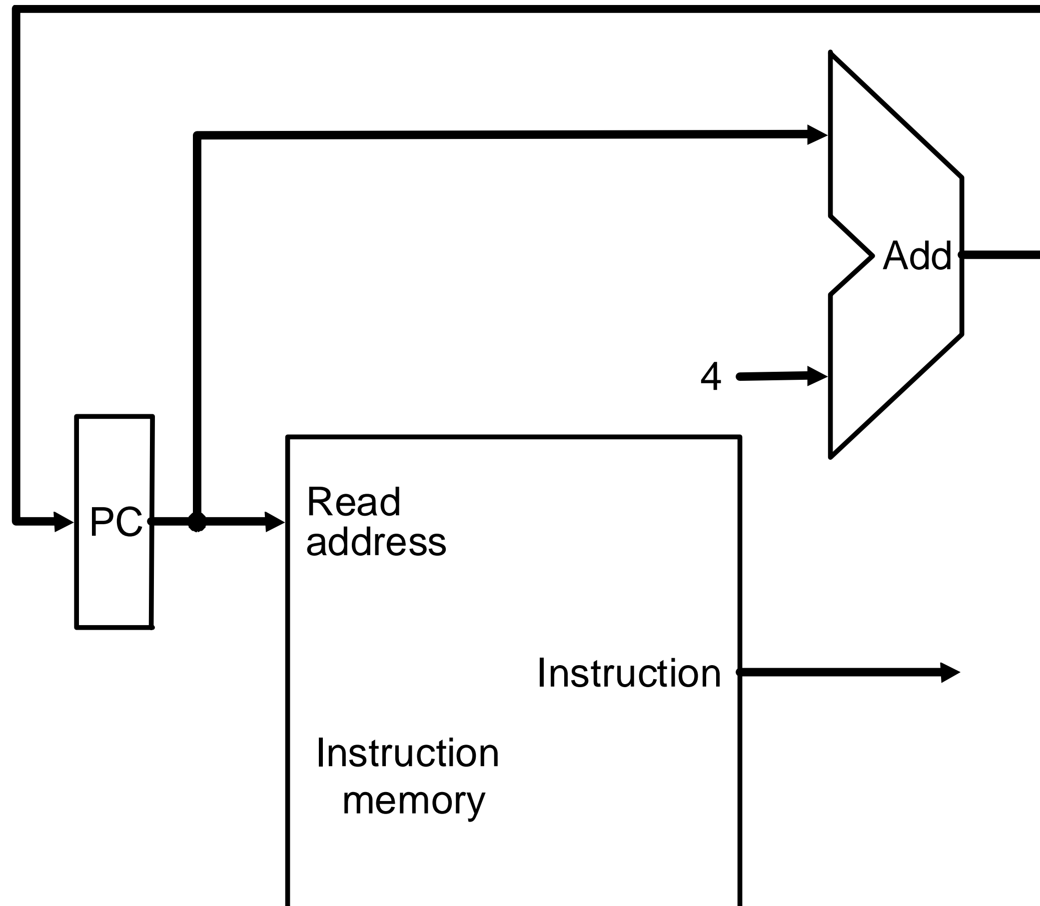
  Why?
    - memory-reference?
    - arithmetic?
    - control flow?

Building Blocks

- Different functional units we need for each instruction



Incrementing the Program Counter (PC)



- Fetching instructions and incrementing the PC

Datapath for R-type Instructions

Register numbers

5

5

5

Read register 1

Read register 2

Write register

Write data

Registers

Read data 1

Read data 2

Data

Data

RegWrite

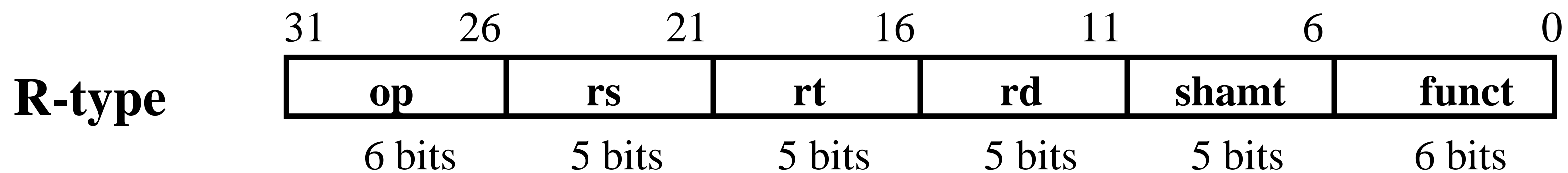
ALU control

3

ALU

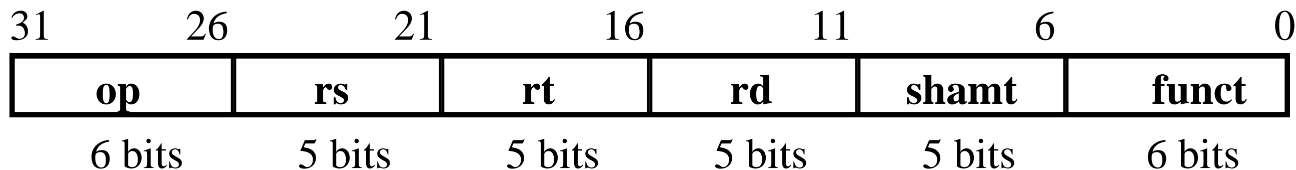
Zero

ALU result



Datapath for R-type Instructions (Contd.)

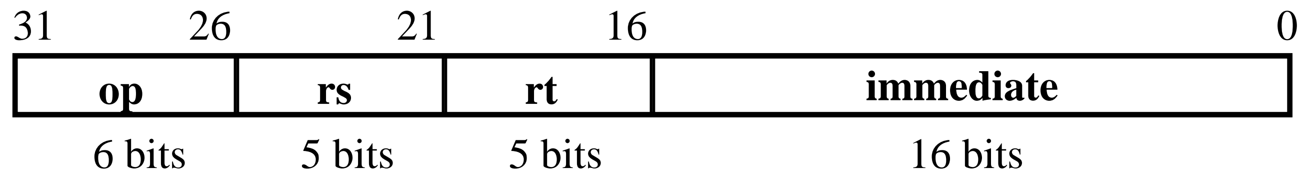
Instruction
rs
rt
rd
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
3
ALU operation
ALU
Zero
ALU result



	31 op 26	rs 21	rt 16	rd 11	shamt 6	funct 0
R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

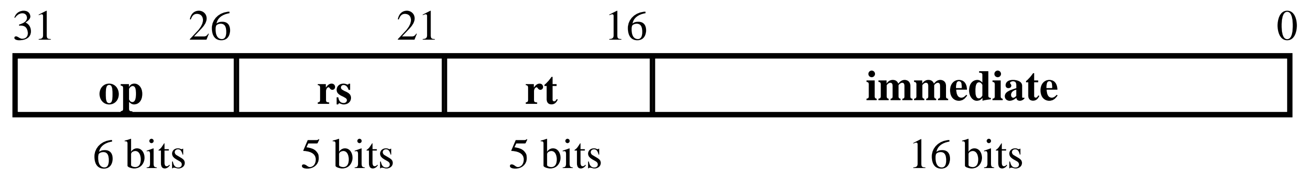
Datapath for Load/Store Instructions

Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
RegWrite
3
ALU operation
ALU
Zero
ALU result
MemWrite
Address
Read data
Data memory
Write data
MemRead
16
Sign extend
32



Datapath for Load Instructions

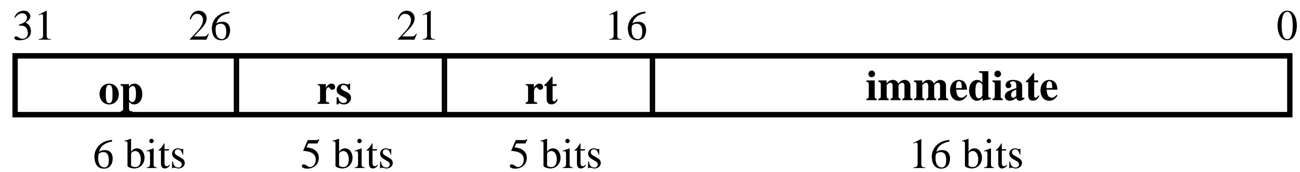
rs
Read register 1
Read register 2
Instruction
rt
Write register
Write data
Registers
Read data 1
Read data 2
RegWrite
immediate
16
Sign extend
32
3
ALU operation
ALU
Zero
ALU result
MemWrite
Address
Read data
Data memory
Write data
MemRead



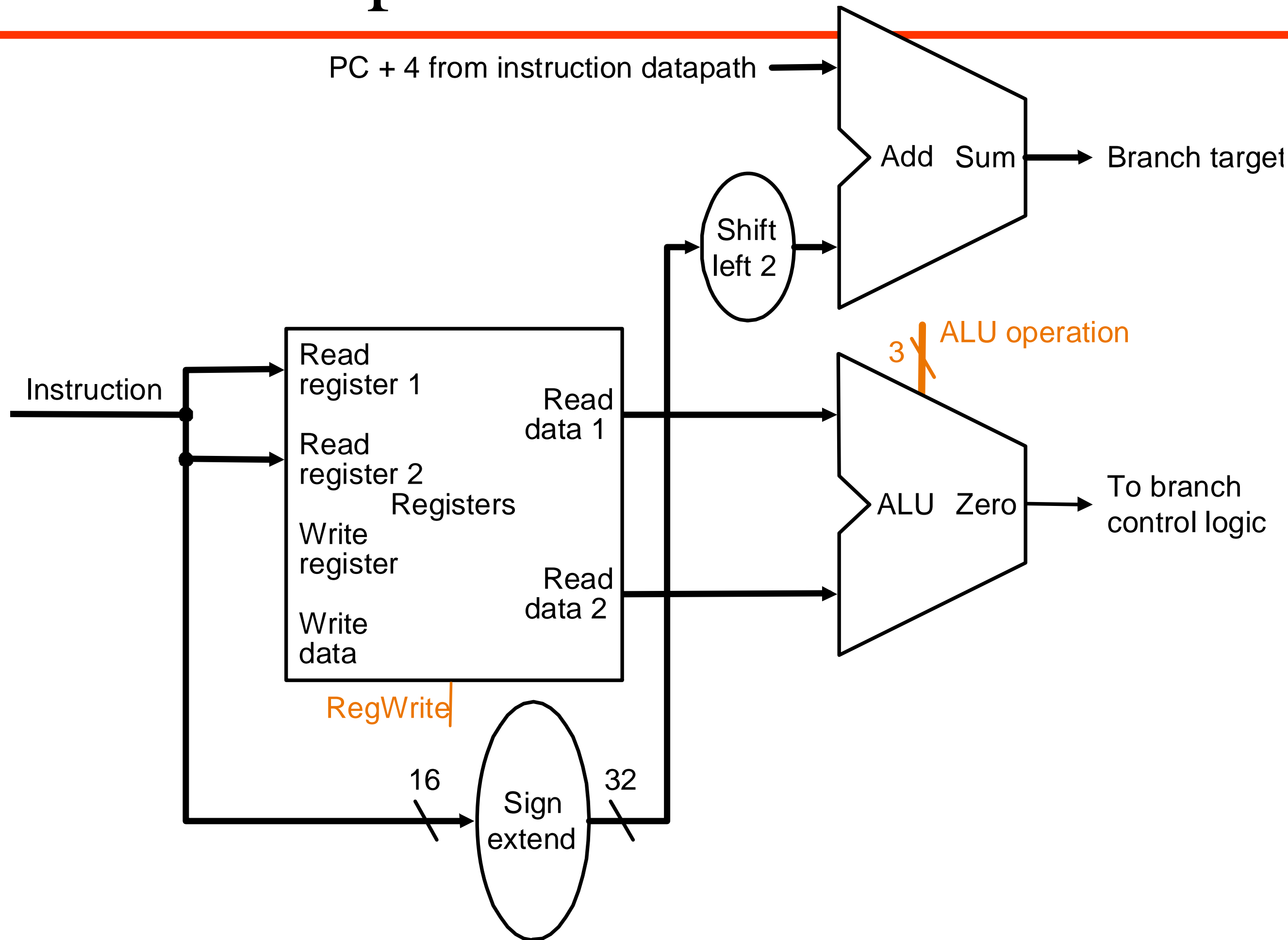
31 – 26	25 – 21	20 – 16	15 – 0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

Datapath for Store Instructions

rs
rt
Instruction
immediate
16
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
RegWrite
Sign extend
32
3
ALU operation
ALU
Zero
ALU result
MemWrite
Address
Read data
Data memory
Write data
MemRead

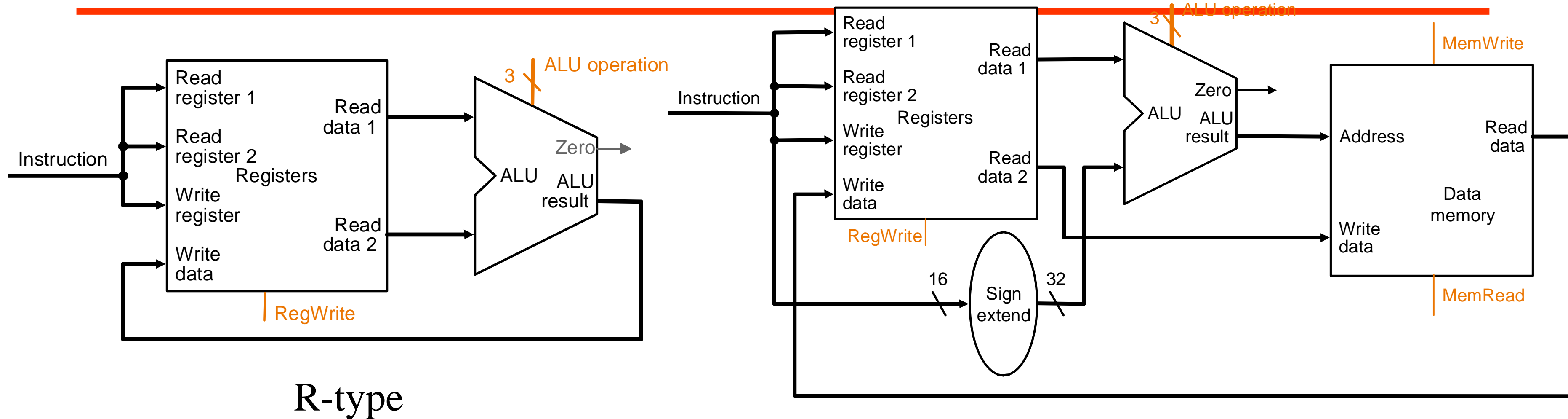


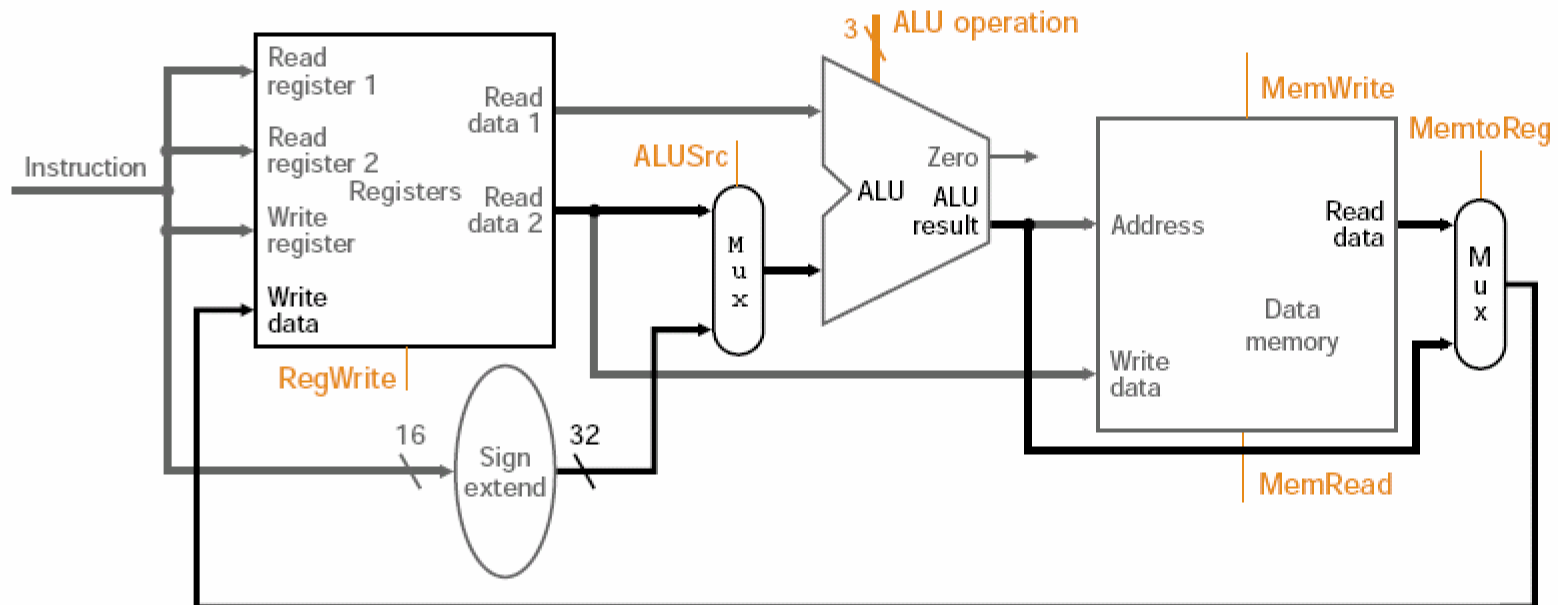
Datapath for Branch Instructions



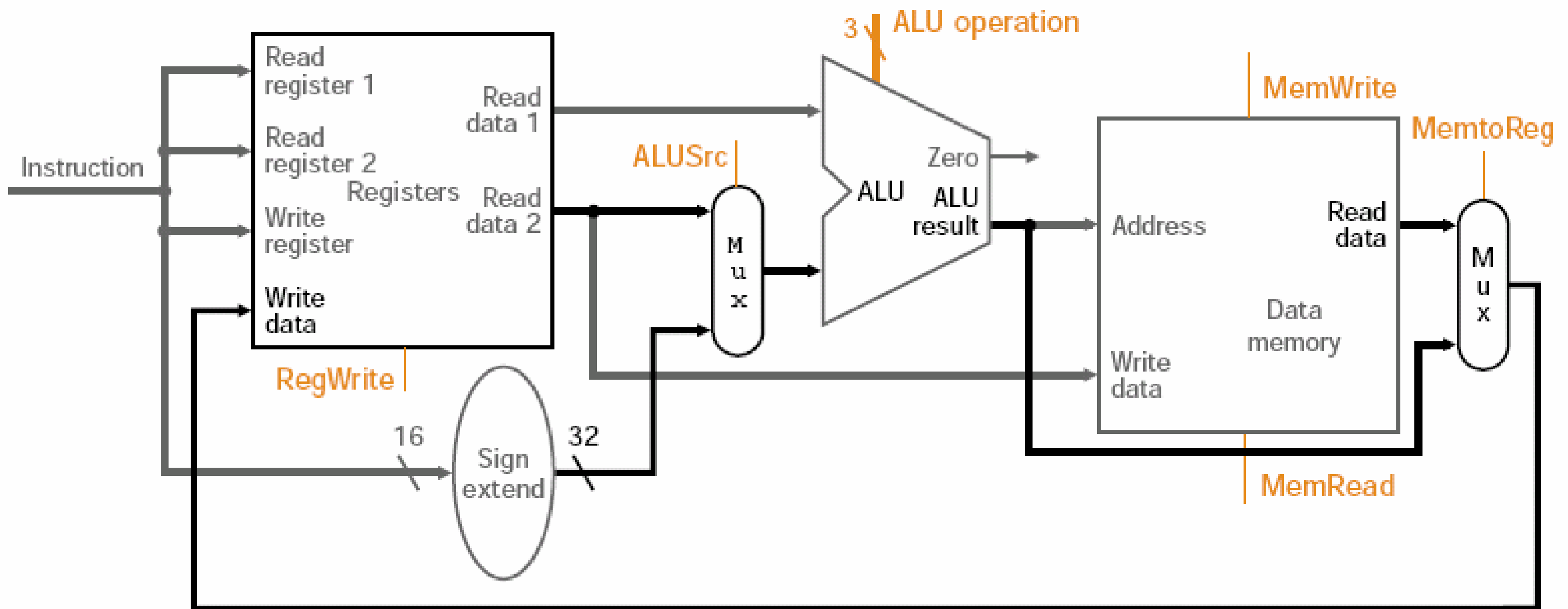
- The ALU is used to evaluate the branch condition and a separate adder is used to compute the branch target address as the sum of the incremented PC and the sign-extended lower 16 bits of the instruction shifted left by 2 bits

Memory & R-type Instructions: Combined Datapath

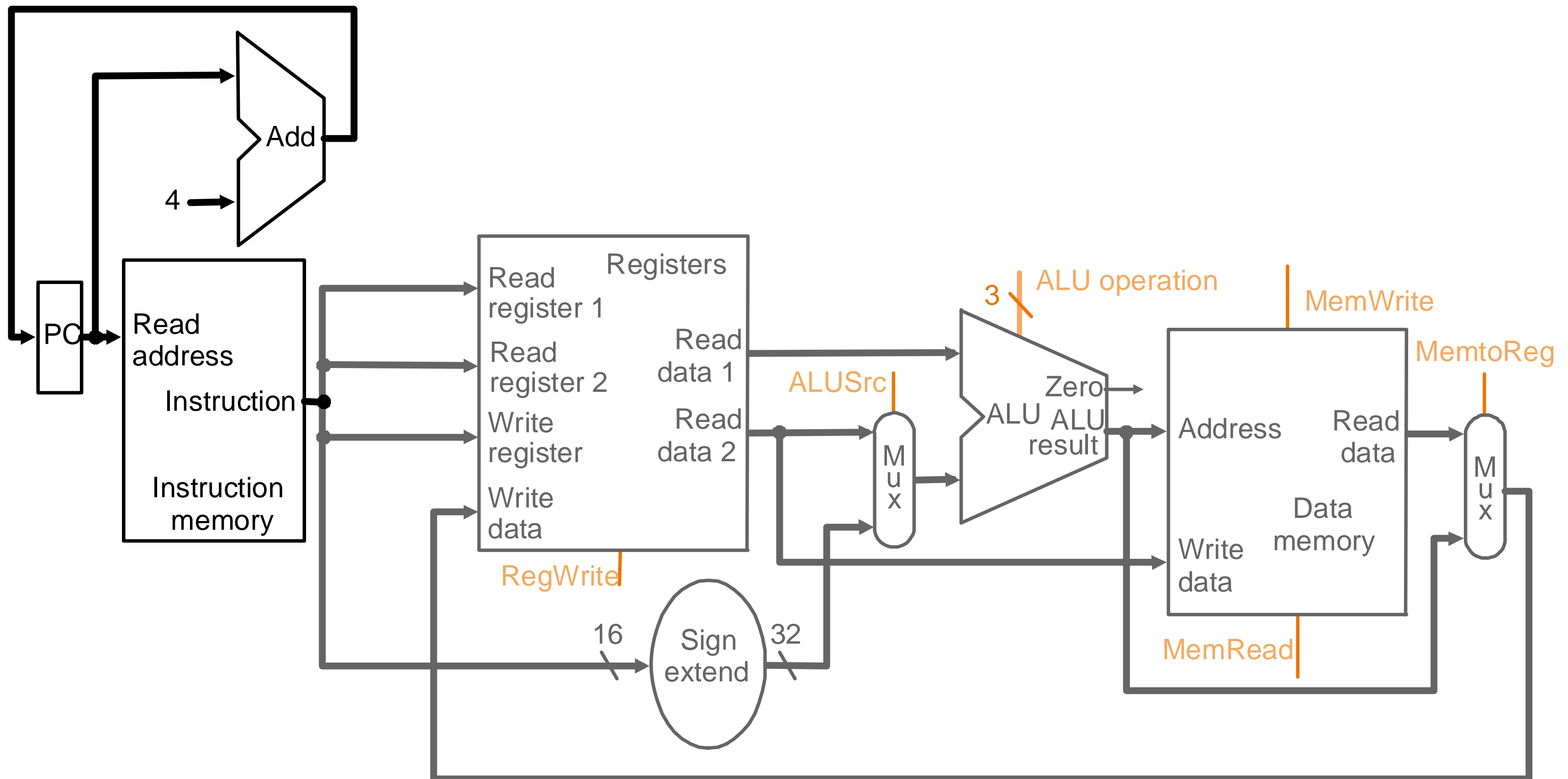




Using the Multiplexor



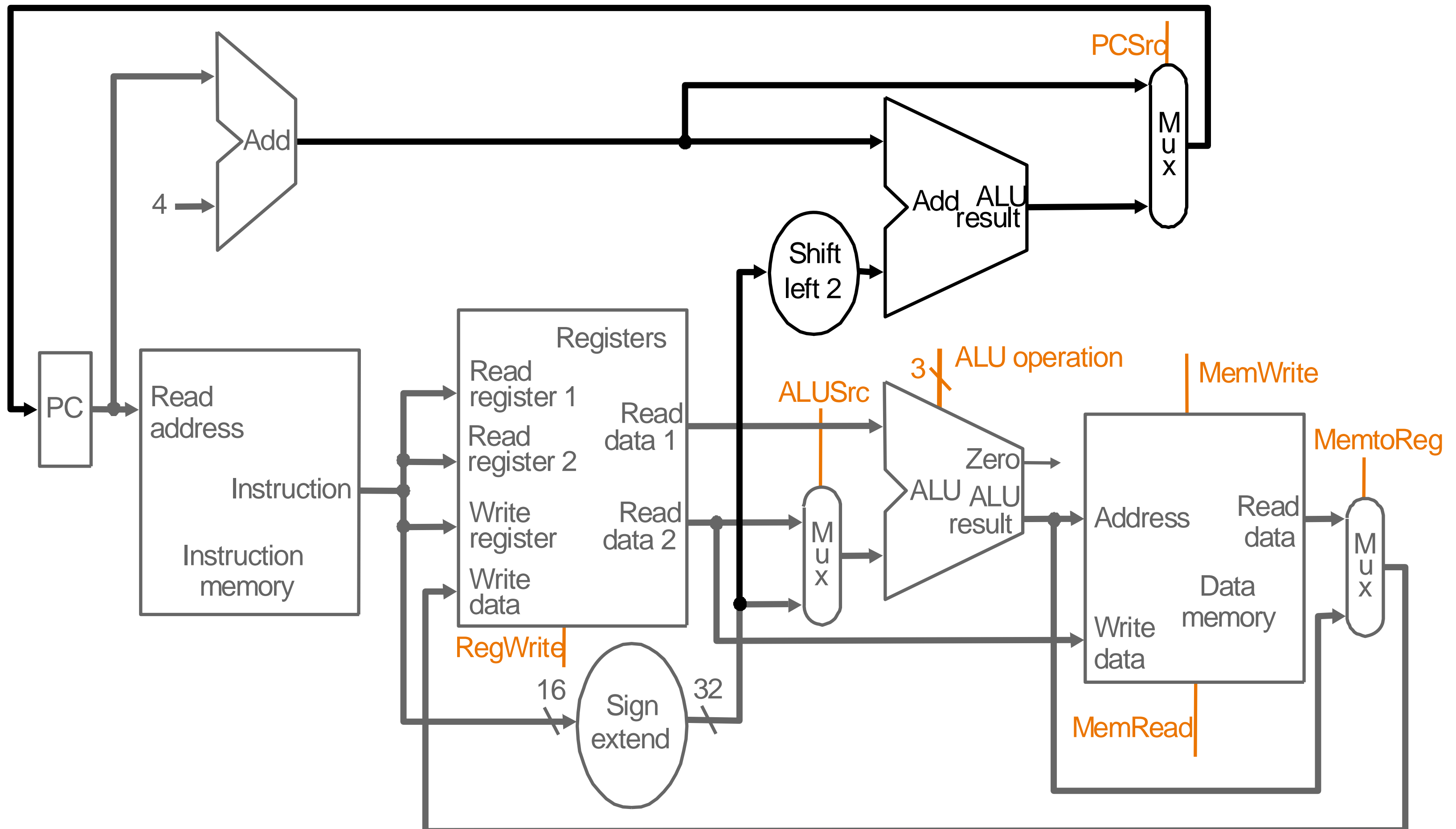
Adding “Instruction Fetch”



- The Instruction Fetch portion of the datapath has now been added to the previous datapath

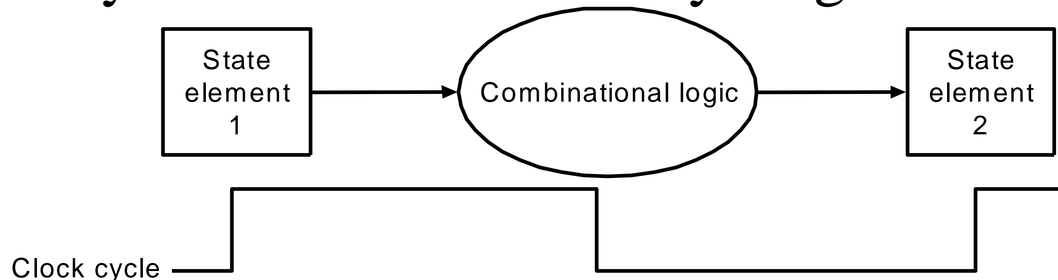
Simple Datapath for the MIPS Architecture

- Finally, adding the datapath for branch instructions

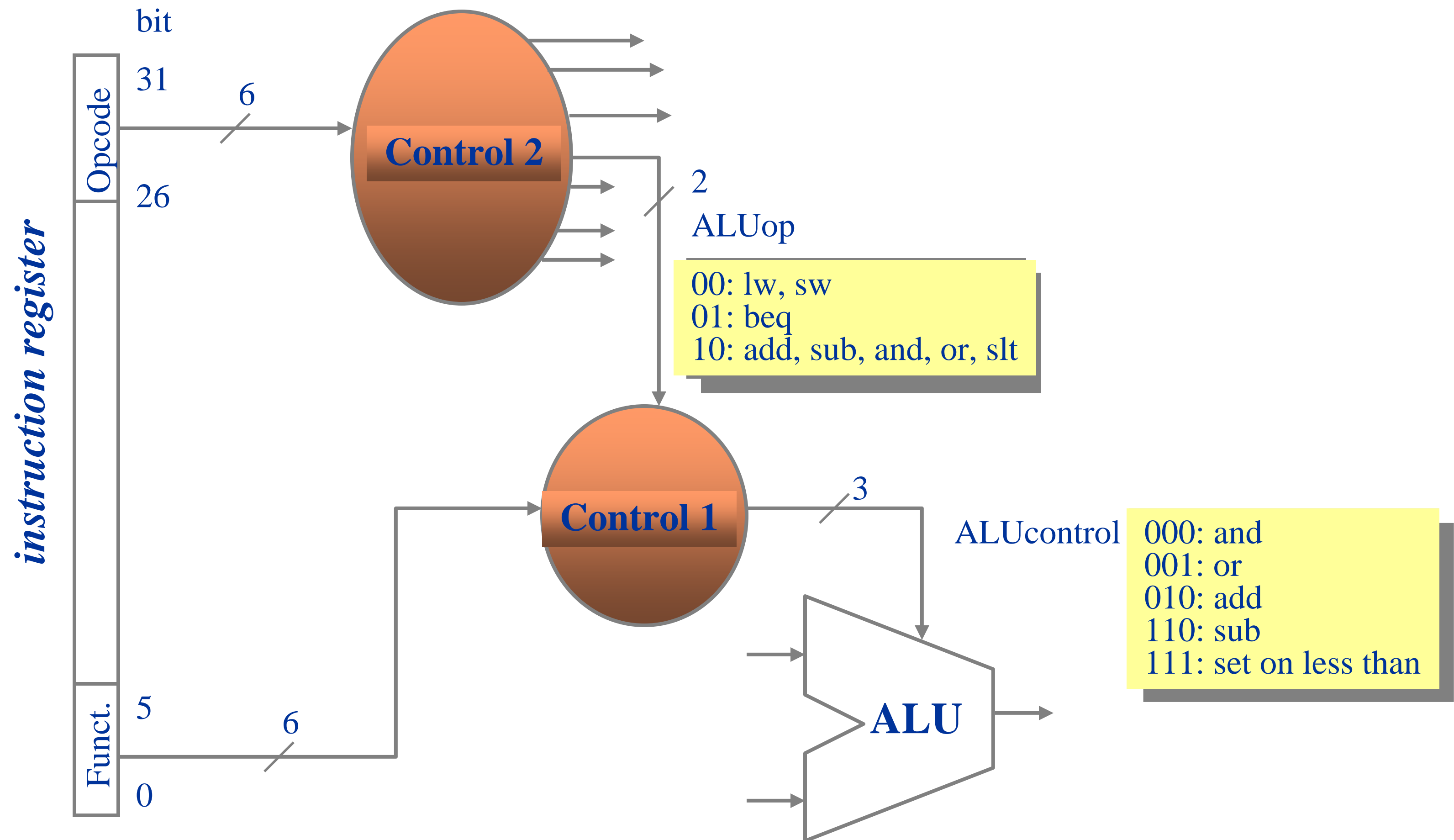


Simple Control Structure

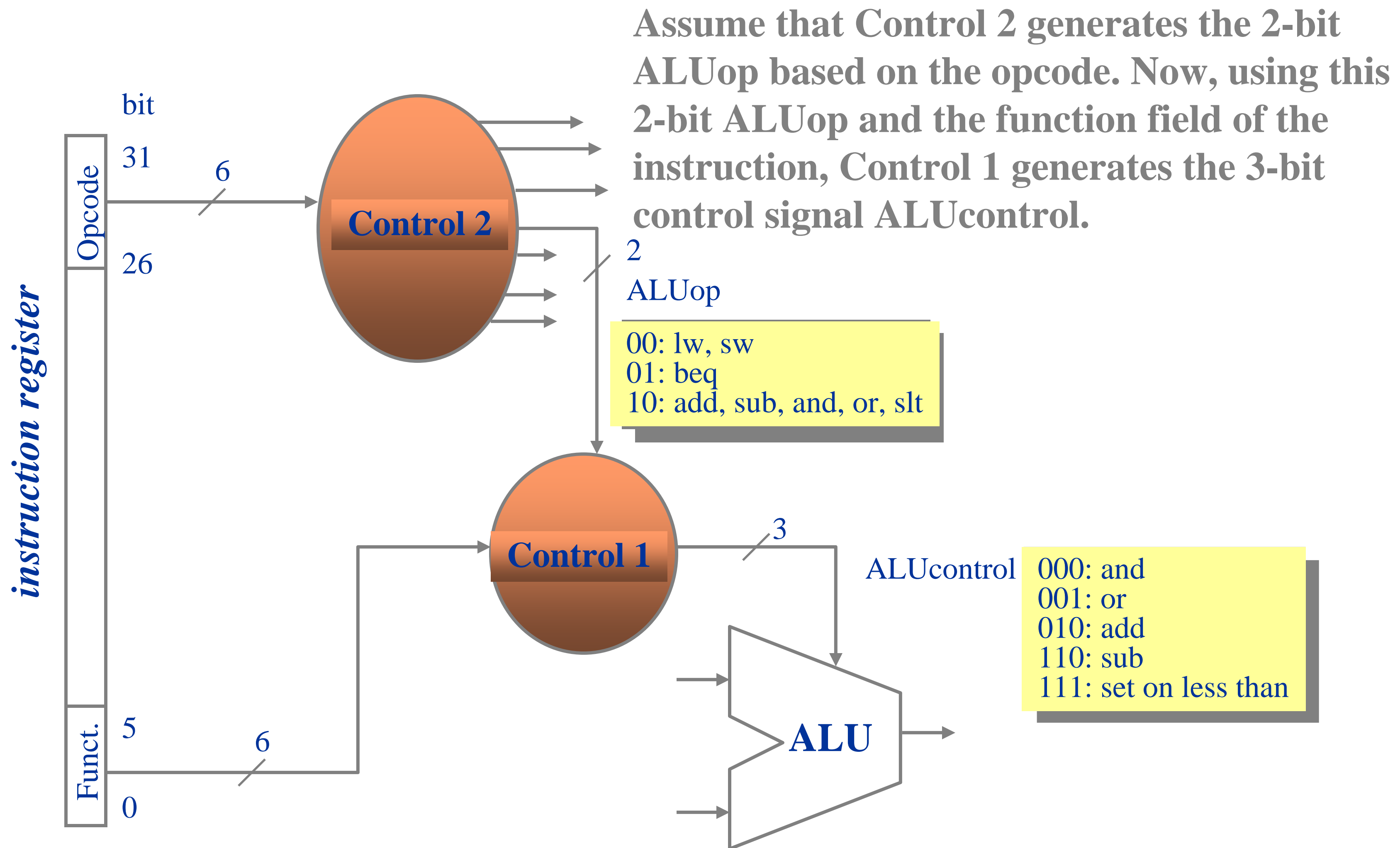
- All of the logic is combinational
- Wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - Use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



Control: Two-level implementation



Designing Control 1



ALUcontrol will determine the function that the ALU will perform (ADD, OR, etc.)

Deriving Control2 signals

Input 9 control (output) signals

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Determine these control signals directly from the opcodes:

R-format: 0

lw: 35

sw: 43

beq: 4

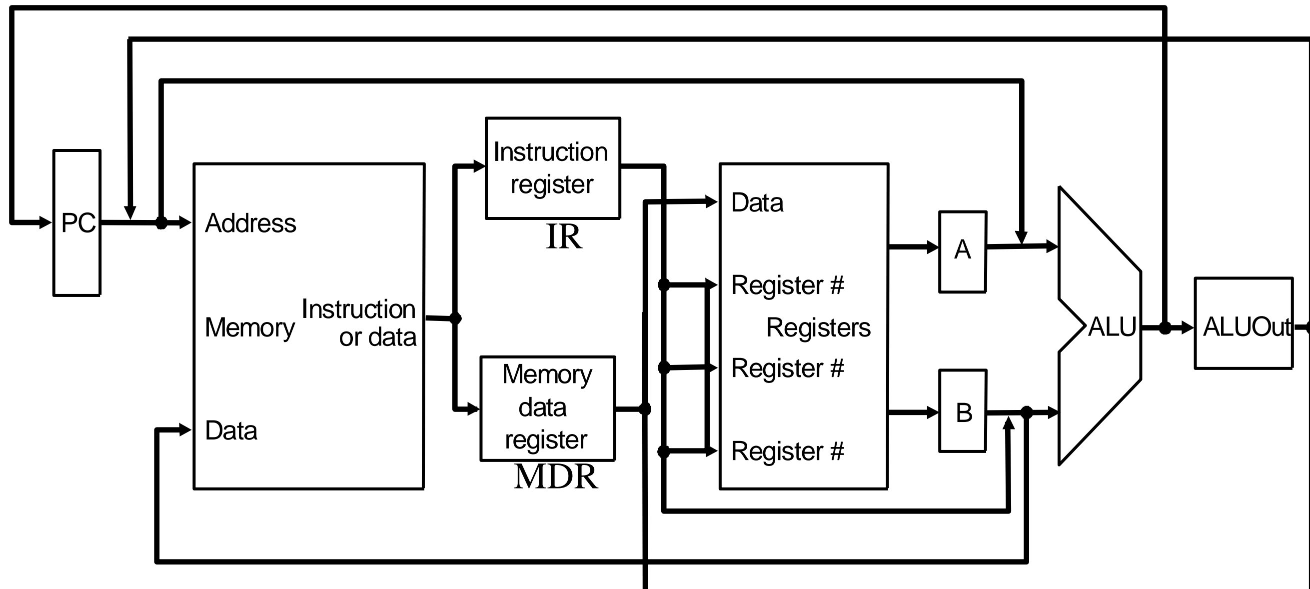
Similarly for the Other Instructions

- For each opcode, find the values of the control signals
- Construct the truth table
- Determine the logic that implements this truth table

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Where we are headed?

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area: *NO Sharing of Hardware resources*
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath:



Why single cycle implementation is not used?

- Assume the following access times: Memory (2ns), ALU & adders (2ns), reg. file access (1ns)
- Fixed length clock: longest instruction is the 'lw' which requires 8 ns
  - Load uses five functional units: instruction memory, register file, ALU, data memory, register file once again
  - Hence, clock cycle is 8ns
- Clock cycle is determined by the longest path in the machine (lw in this case)
- However, several other instructions could fit into a shorter clock cycle

Why single cycle implementation is not used?

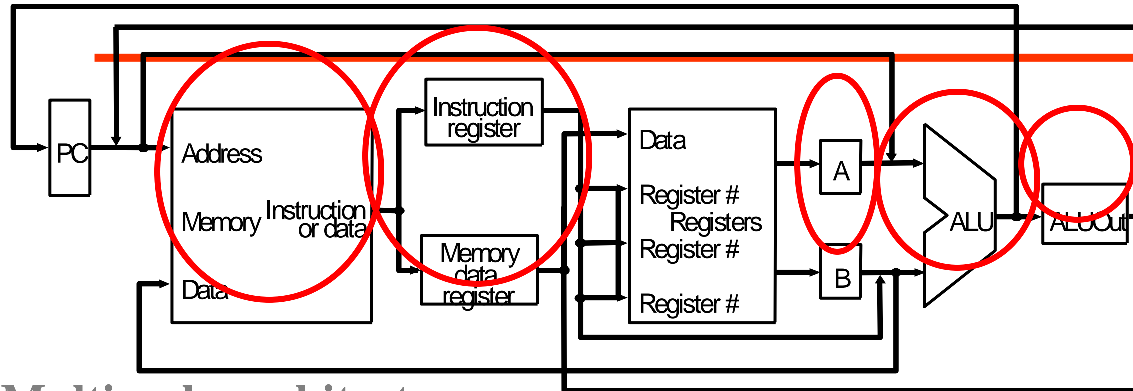
- **R-type:** Instruction fetch, Reg access, ALU, Reg access
- **Load:** Instruction fetch, Reg access, ALU, Mem access, Reg access
- **Store:** Instruction fetch, Reg access, ALU, Mem access
- **Branch:** Instruction fetch, Reg access, ALU
- **Jump:** Instruction fetch

Note the difference between Load and Jump. This difference becomes even more significant of there are floating-point instructions.

Multicycle implementation: Basics

- In the previous slide, the execution of each instruction was broken into several steps
- In a multicycle implementation, each such step executes in 1 clock cycle
- Hence, different instructions require different number of clock cycles
- Advantages:
  - More efficient
  - A functional unit can be used more than once per instruction, as long as it is used in different clock cycles (so less hardware is required)
- But the design is more complex

Single-Cycle versus Multicycle



Multicycle architecture

- **In a multicycle architecture:**
- Single memory unit for both instruction and data
- Single ALU, rather than one ALU and two adders
- One or more registers added after each functional unit to hold the output of that unit, until the value is used in the next clock cycle

Single cycle architecture

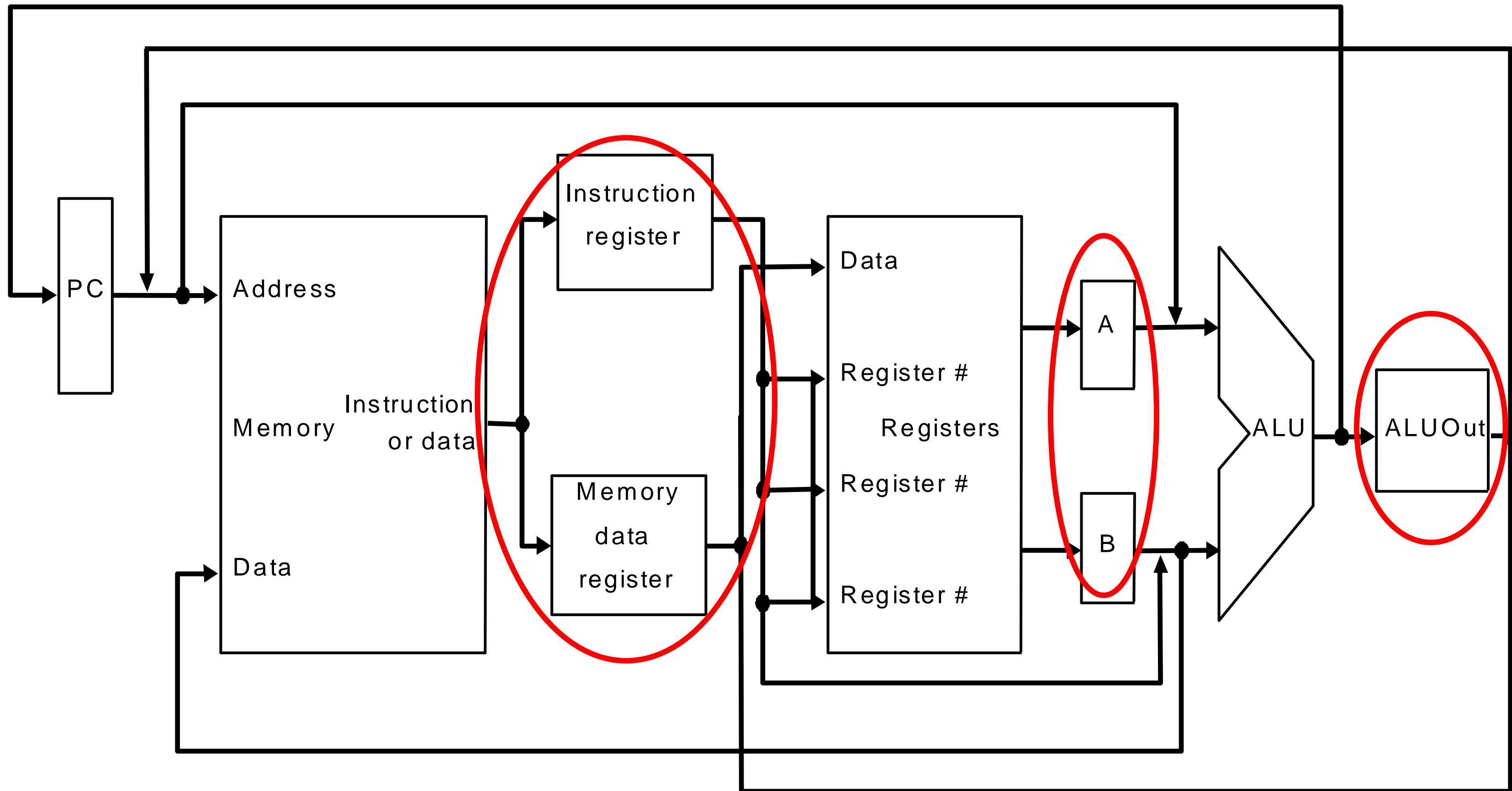
Multicycle implementation: Additional Registers

- Instruction Register, Memory Data Register, Registers A and B in front of the Reg file and ALUOut (reg in front of the ALU)
- At the end of each clock cycle, the data to be used in subsequent clock cycles is stored in a state element
  - data to be used in *subsequent instructions* in a later clock cycle is stored in a programmer-visible state element like reg file, PC or memory
  - data used by the *same instruction* in a later cycle is stored in one of the additional registers

Multicycle implementation

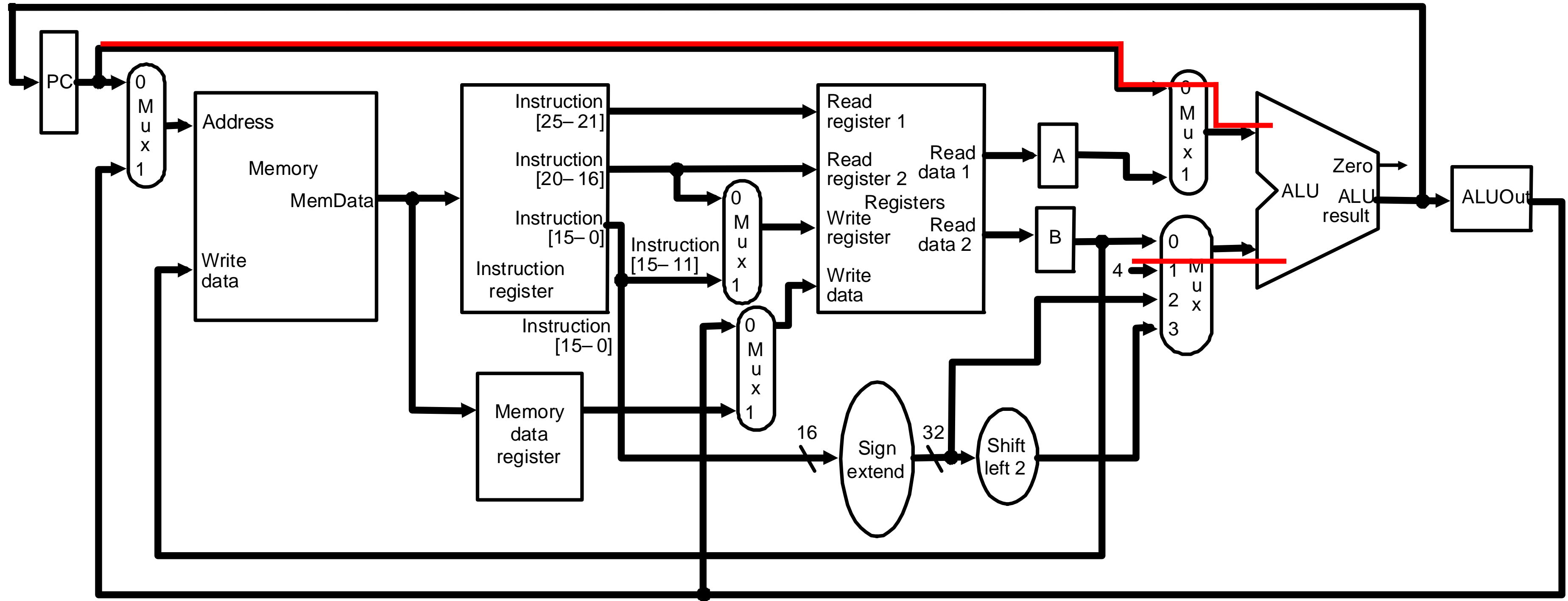
- Each clock cycle can accommodate at most one of the following operations:
  - a memory access
  - a register file access (two reads or one write)
  - an ALU operation
- Hence, any data produced by one of the above three functional units must be saved into a temporary register for use in a later cycle

Multicycle implementation: Additional Registers



All registers except the Instruction register (IR) hold data only between a pair of adjacent clock cycles (and hence do not need a write control signal)

Multicycle implementation: Examples



ALU used to compute PC = PC + 4

The same ALU is also used for R-type instructions, branch address computation, computing memory address in the case of lw/sw instructions

Multicycle Approach: Summary

- 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 implementation: Steps

- Instruction fetch
- Instruction decode and register fetch

common for all instructions

- Execution, memory address computation or branch completion
- Memory access or R-type instruction completion
- Memory read completion

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register
- Increment the PC by 4 and put the result back in the PC
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR = Memory[PC];
PC = PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

This step is common for all instructions (obviously!)

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- Previous two actions are done optimistically (no harm is done)
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC+(sign-extend(IR[15-0])<< 2);
```

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

This step is also common for all instructions

Step 3 (instruction dependent)

- ALU is performing one of four functions, based on instruction type
- Memory Reference:

```
ALUOut = A + sign-extend(IR[15-0]);
```

- R-type:

```
ALUOut = A op B;
```

- Branch:

```
if (A==B) PC = ALUOut;
```

- Jump:

```
PC = PC[31-28] || (IR[25-0]<<2)
```

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
MDR = Memory[ALUOut];
      or
Memory[ALUOut] = B;
```

- R-type instructions finish

```
Reg[IR[15-11]] = ALUOut;
```

Step 5: Write-back step

- Memory read completion step

```
Reg[IR[20-16]]= MDR;
```

Summary of execution steps

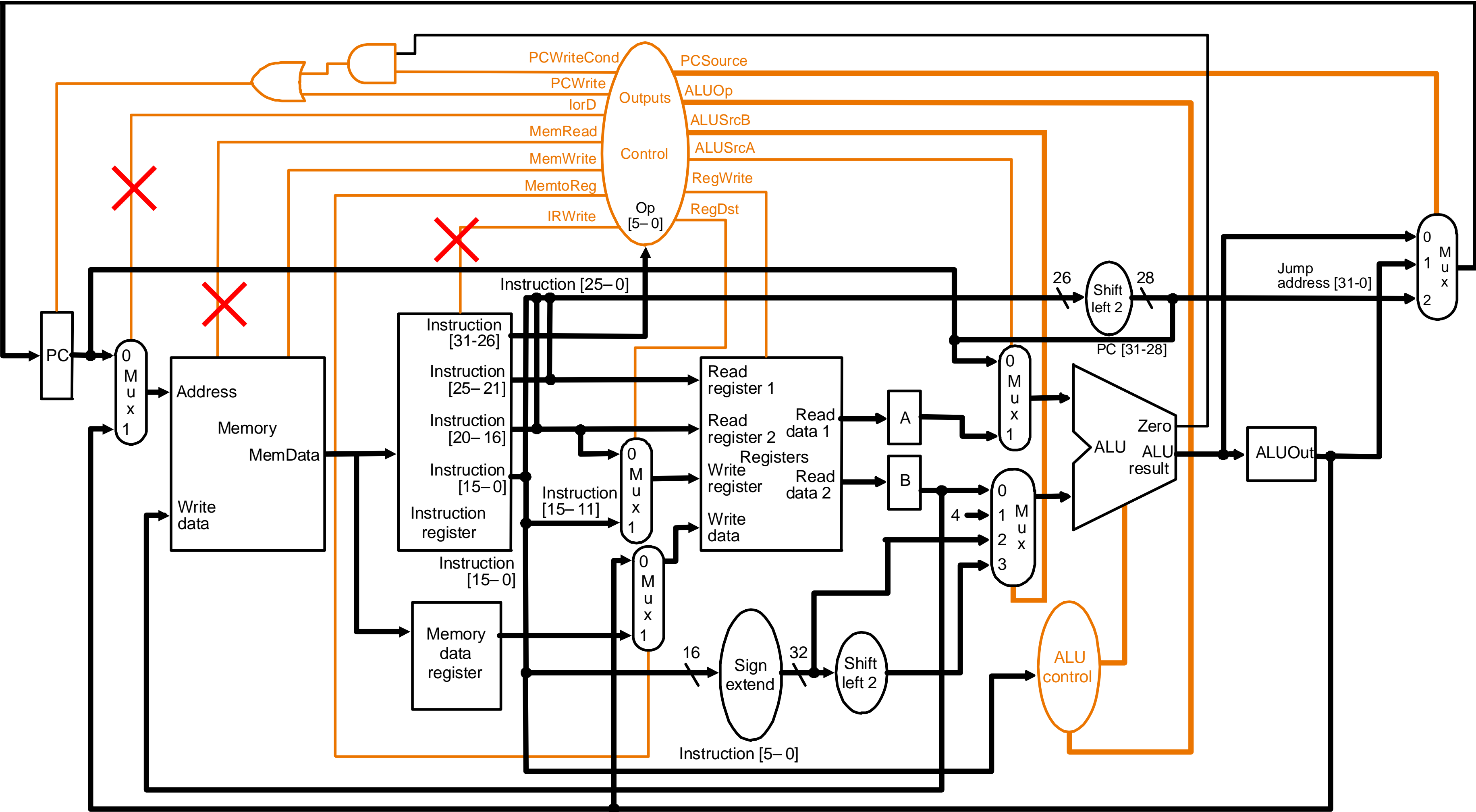
Steps taken to execute any instruction class

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Multicycle Control Unit

- We now need to determine the values of the control signals for each of the Steps 1 – 5
- The next few slides show how this is done for Step 1
- The procedure for the remaining steps is similar

Step 1: Instruction Fetch Step

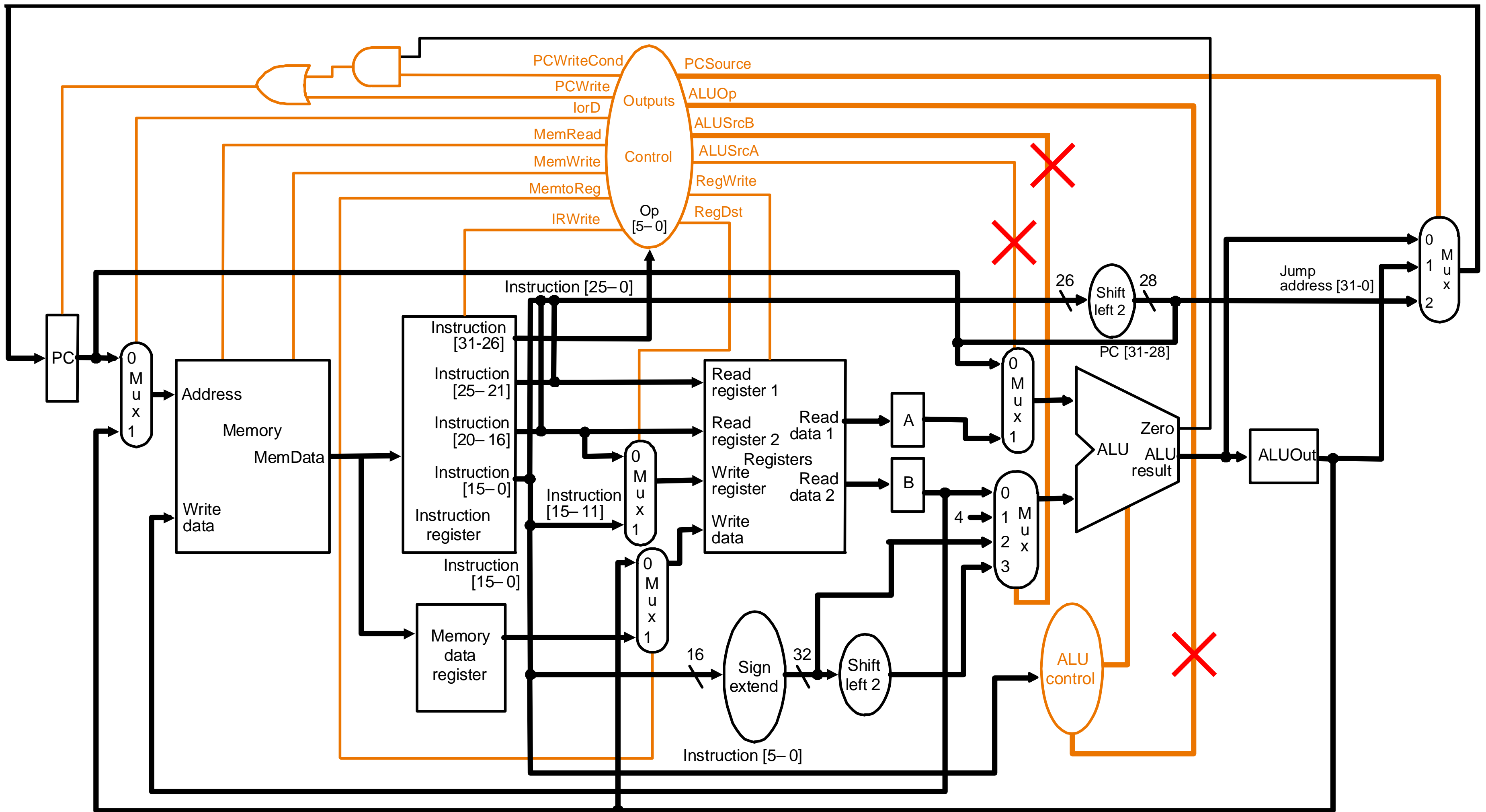


MemRead=1 IorD=0

IRWrite=1

```
IR = Memory[PC];
PC = PC + 4;
```

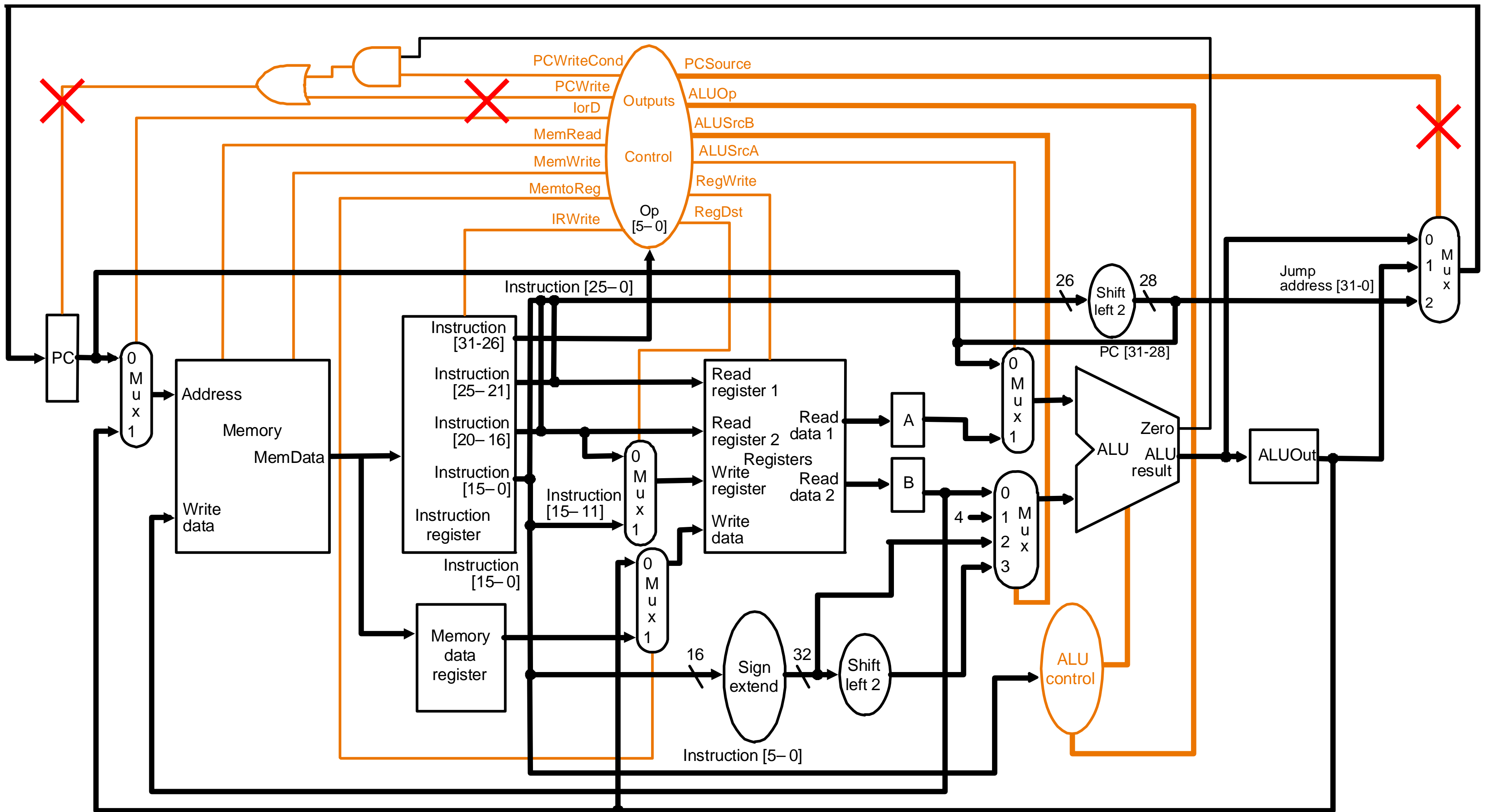
Step 1: Instruction Fetch Step



Increment PC by 4: ALUSrcA=0; ALUSrcB=01; ALUOp=00 (for ALU to ADD)

```
IR = Memory[PC];
PC = PC + 4;
```

Step 1: Instruction Fetch Step



Store incremented instruction address back to PC:
PCSource=00; PCWrite=1

```
IR = Memory[PC];
PC = PC + 4;
```

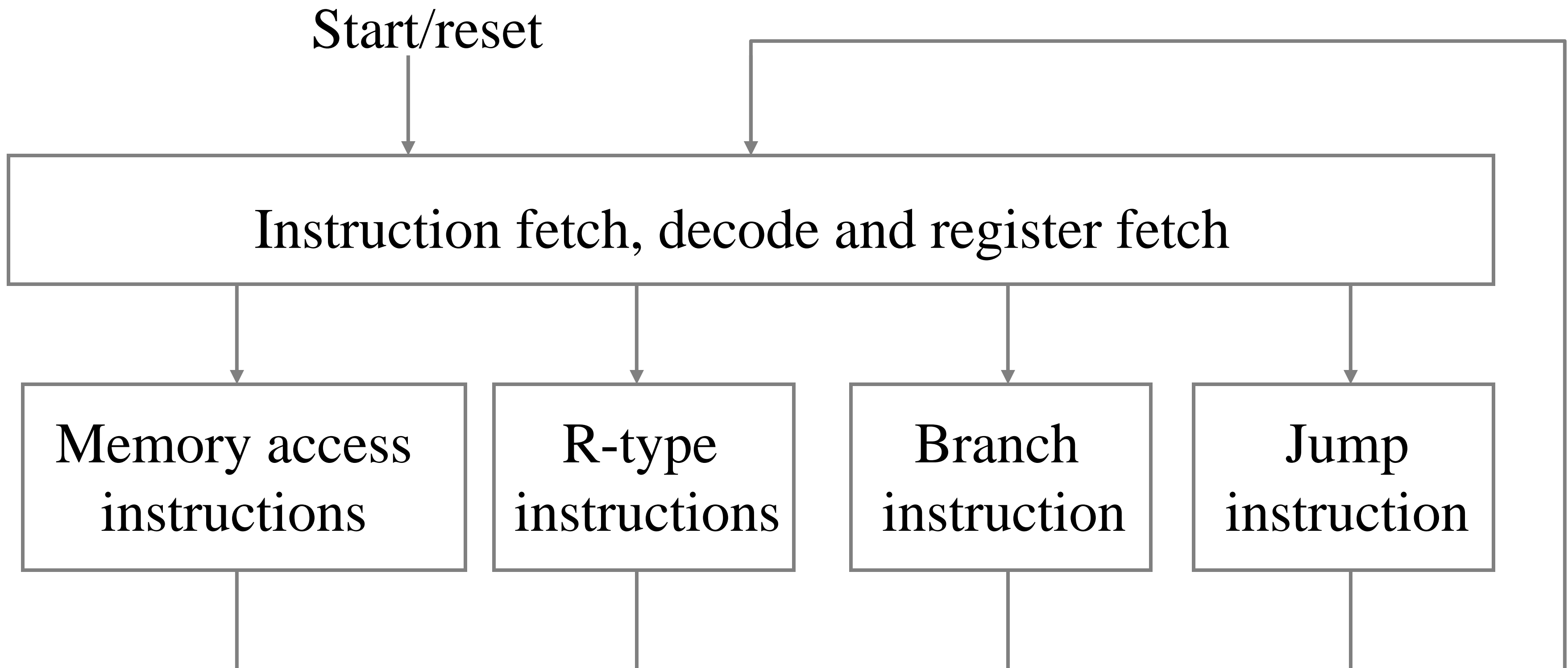
Steps 2 – 5

- Determining the values of the control signals for Steps 2 – 5 follows the same procedure
- The next stage is to design the control unit, which will generate these control signals

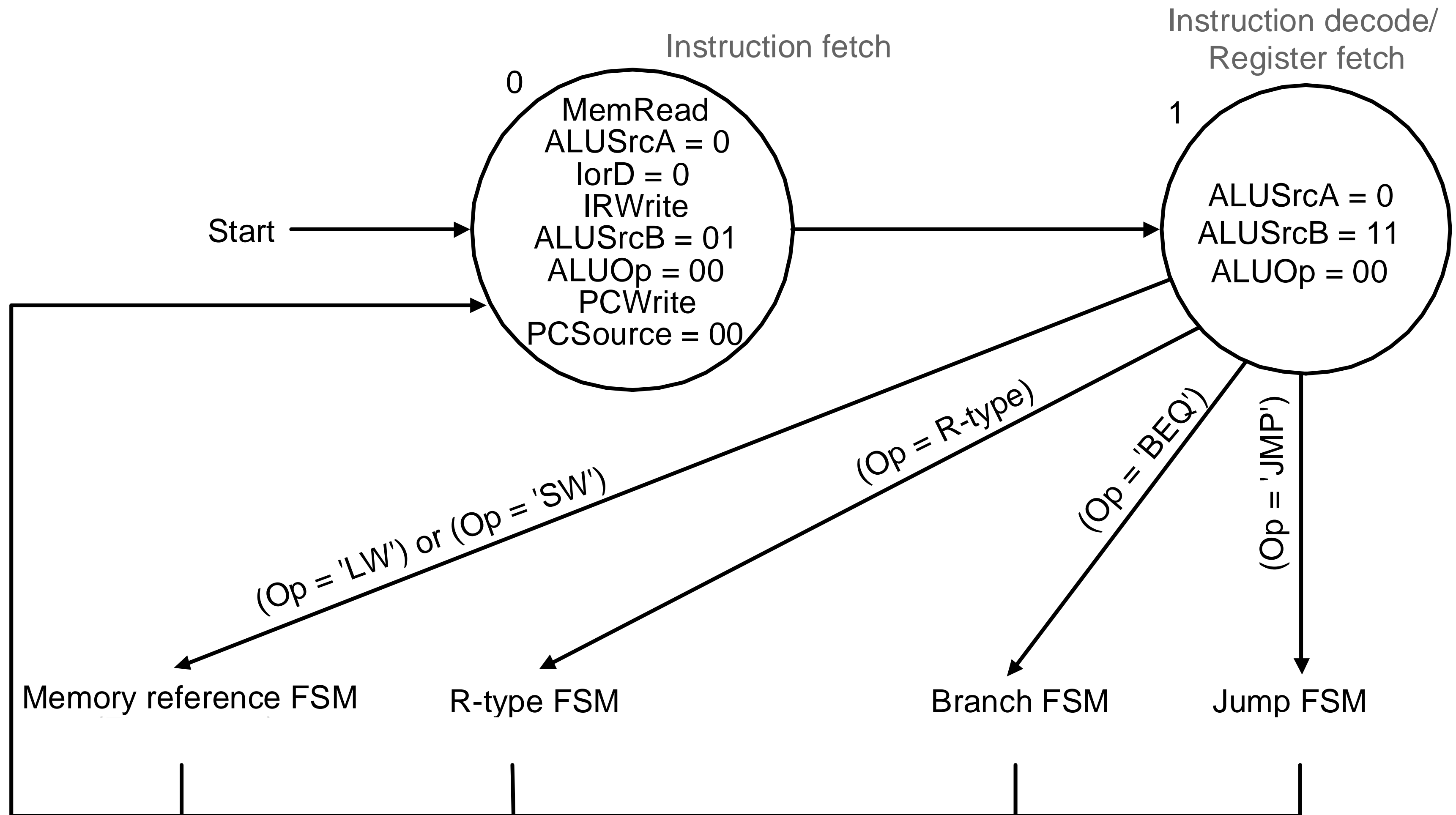
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we have accumulated (ex: control signals for Step 1) to specify a finite state machine (FSM)
  - specify the finite state machine graphically, or
  - use *microprogramming*
- Implementation can be derived from specification

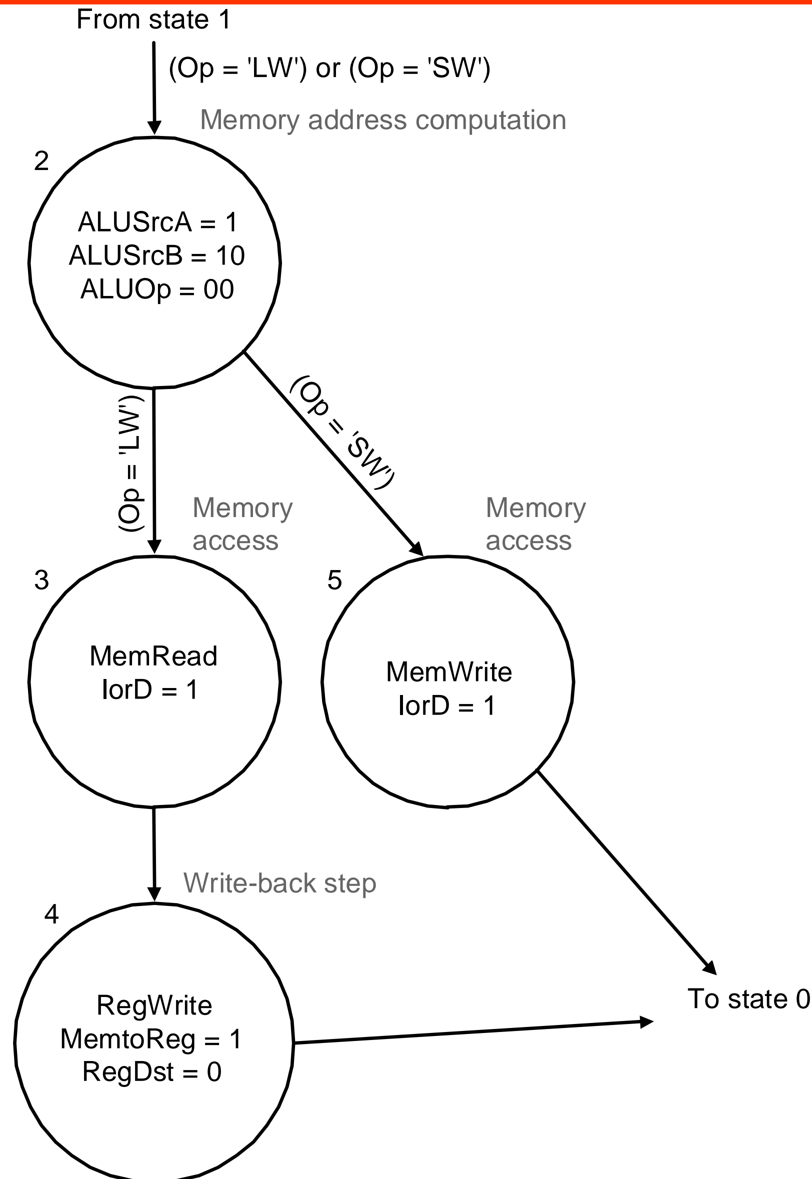
FSM: high level view



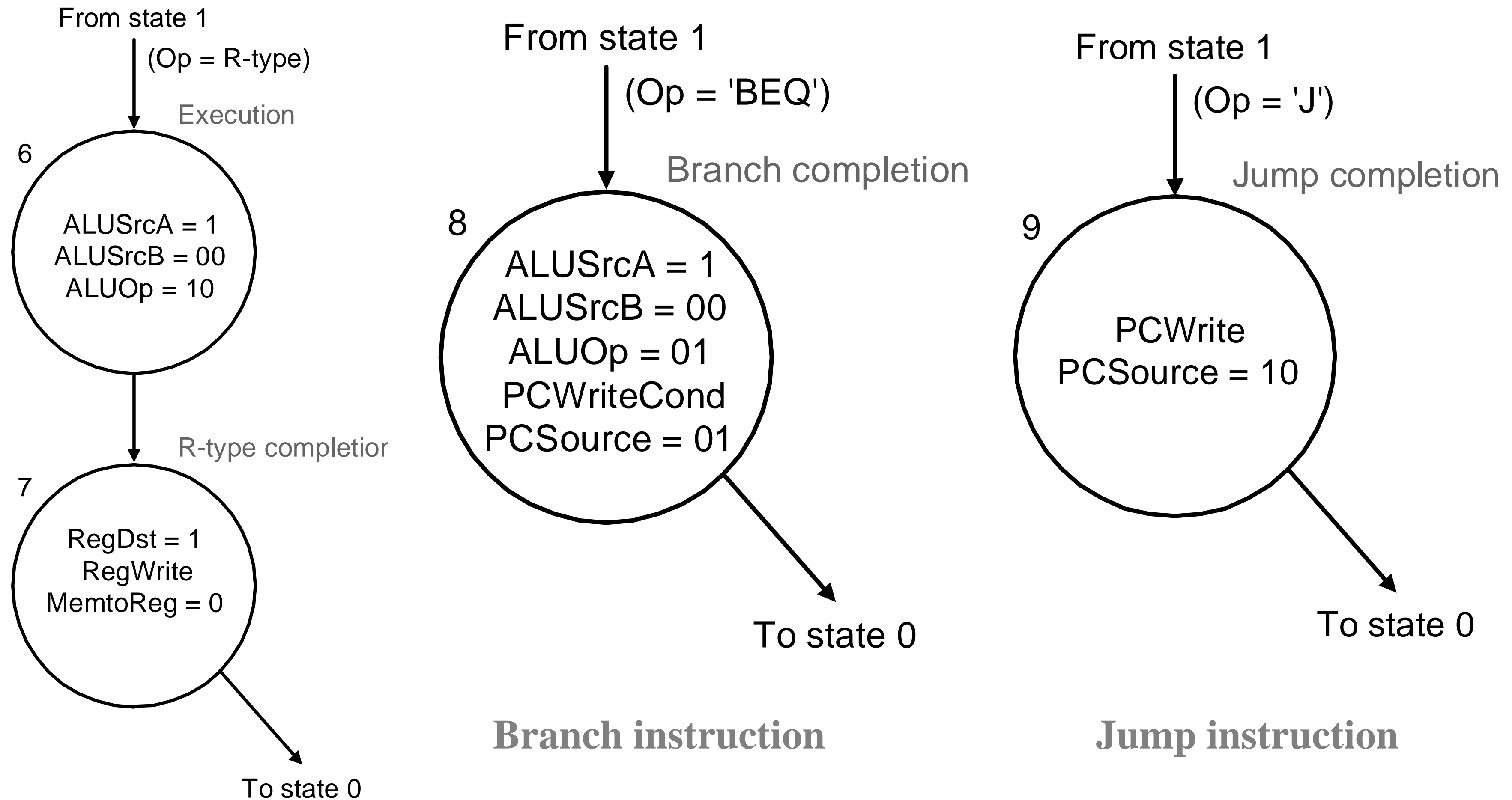
FSM implementation of the control unit



FSM for memory reference instructions



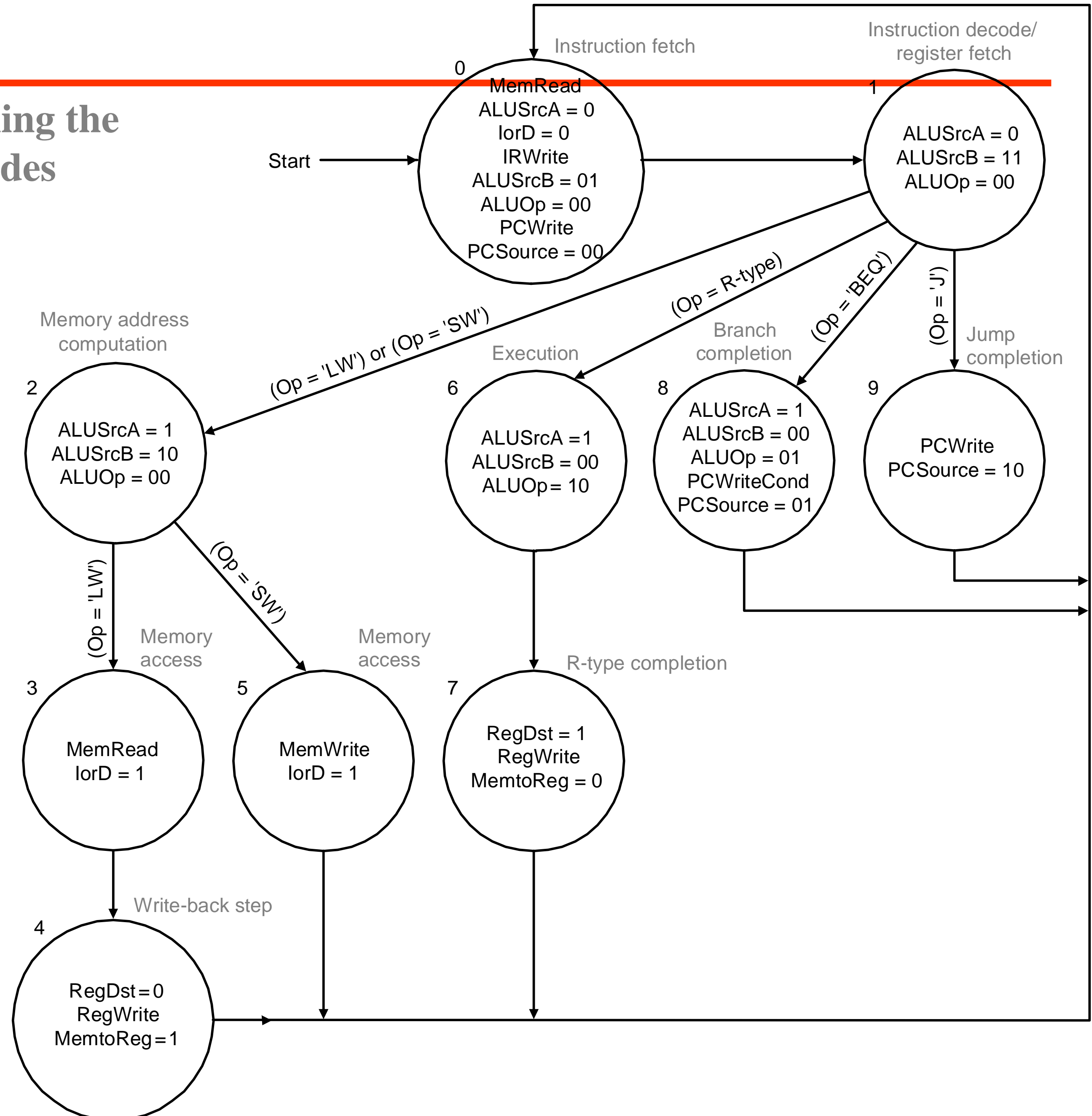
FSMs for other instructions



R-type instructions

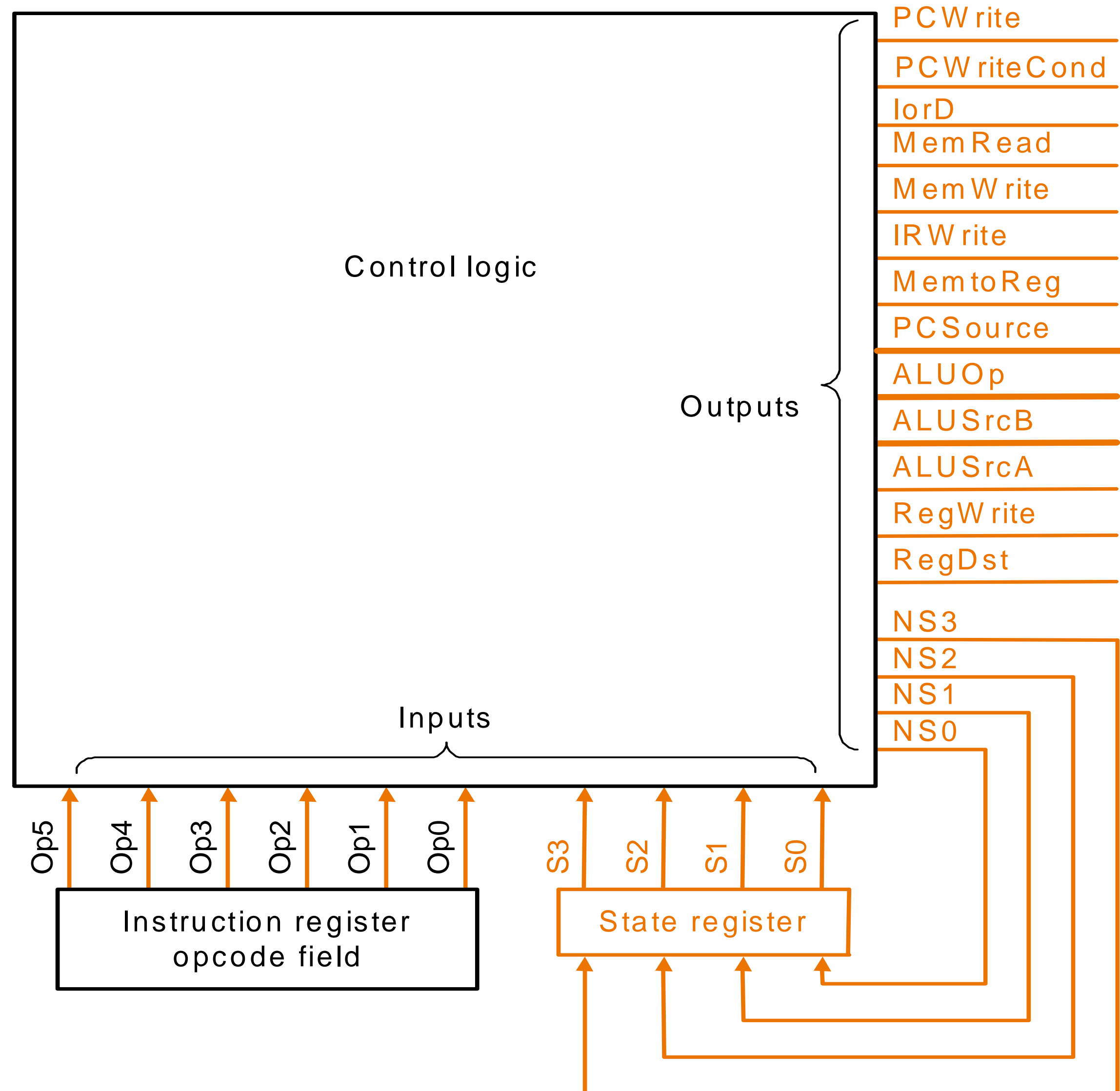
The Full FSM for the Control Unit

Obtained by simply joining the FSMs in the previous slides



Finite State Machine for Control

Implementation:

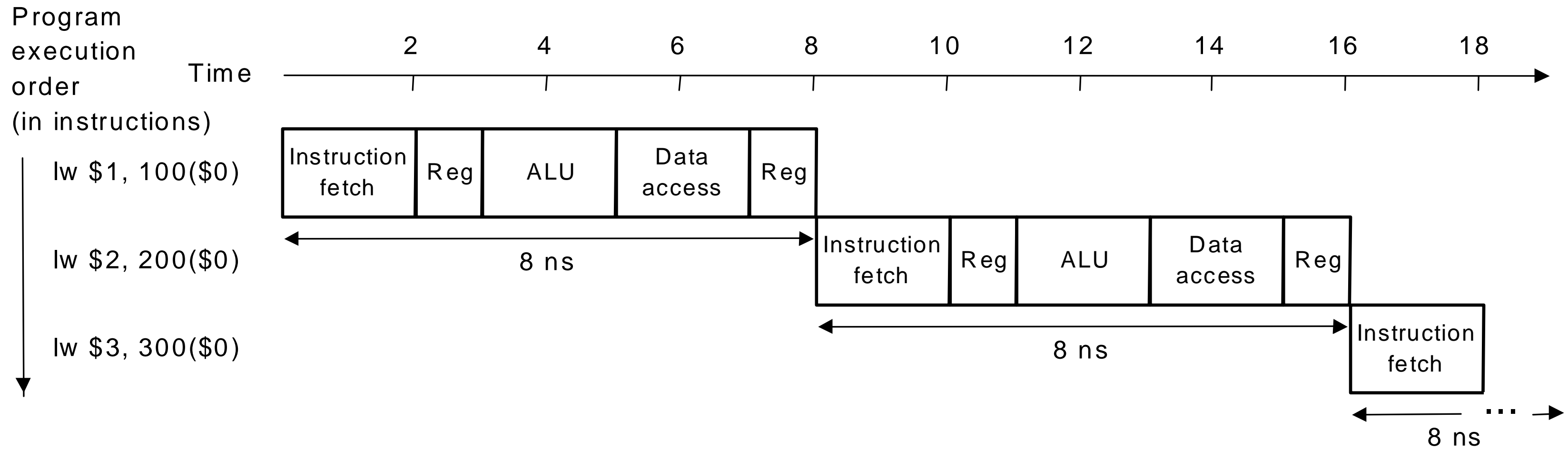


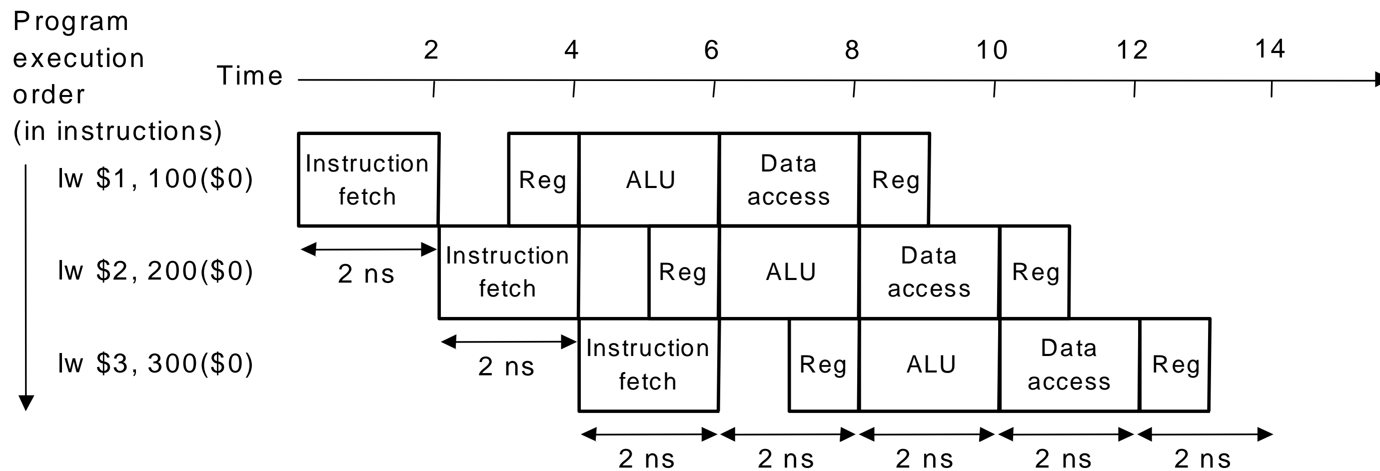
Further Improvement

- We saw that performance can be improved by using a multicycle implementation, compared to a single cycle one
- Most modern processors rely on further improvements by exploiting instruction-level parallelism (ILP), where multiple instructions are evaluated in parallel
- We will now briefly look at simple pipelining
- In the following classes, we will review the basic concepts of Superscalar and VLIW processors

Pipelining

Improve performance by increasing instruction throughput





Pipelining

- Ideal speedup = number of stages
- Do we achieve this?

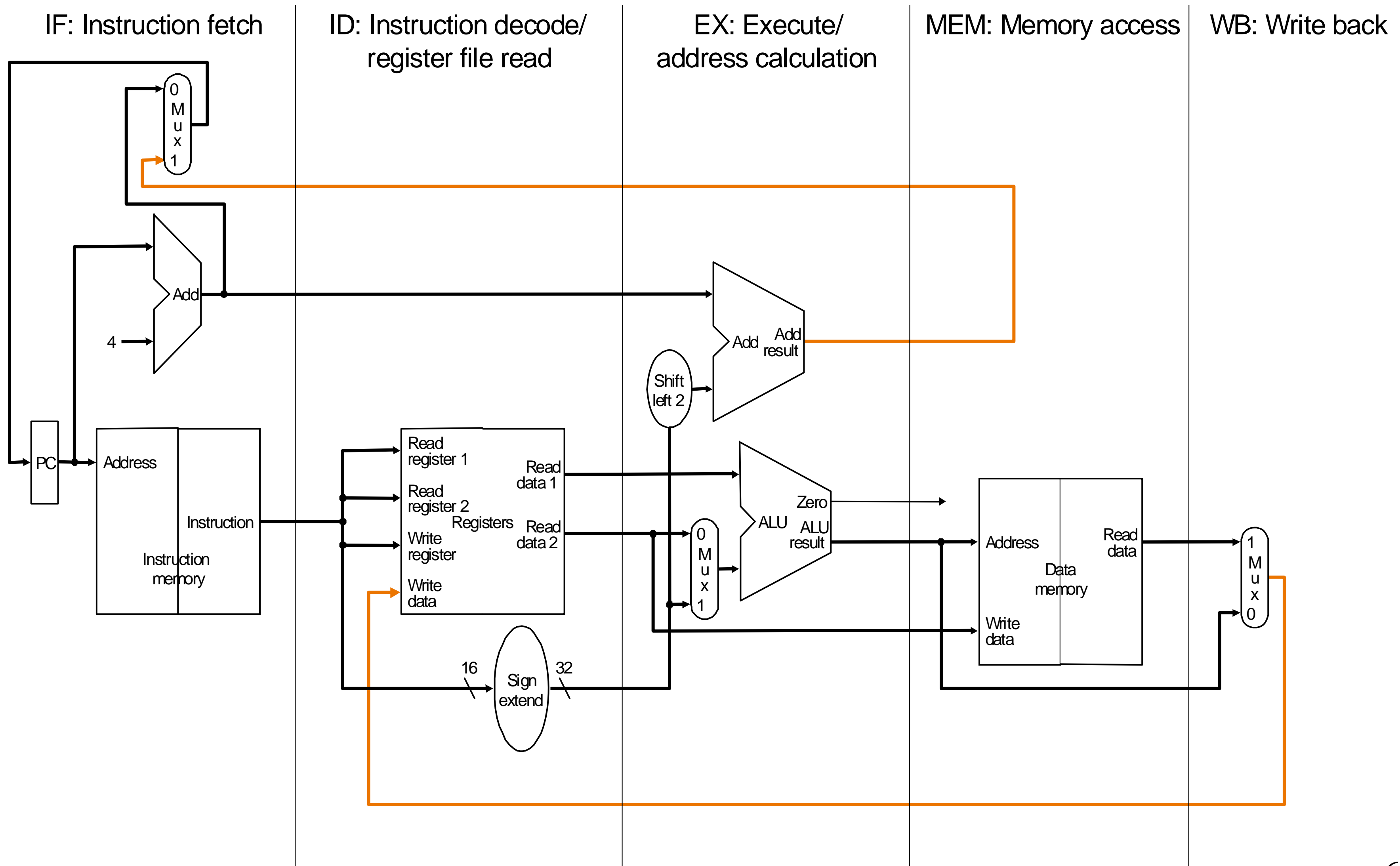
Pipelining

- What makes it easy
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores

- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction

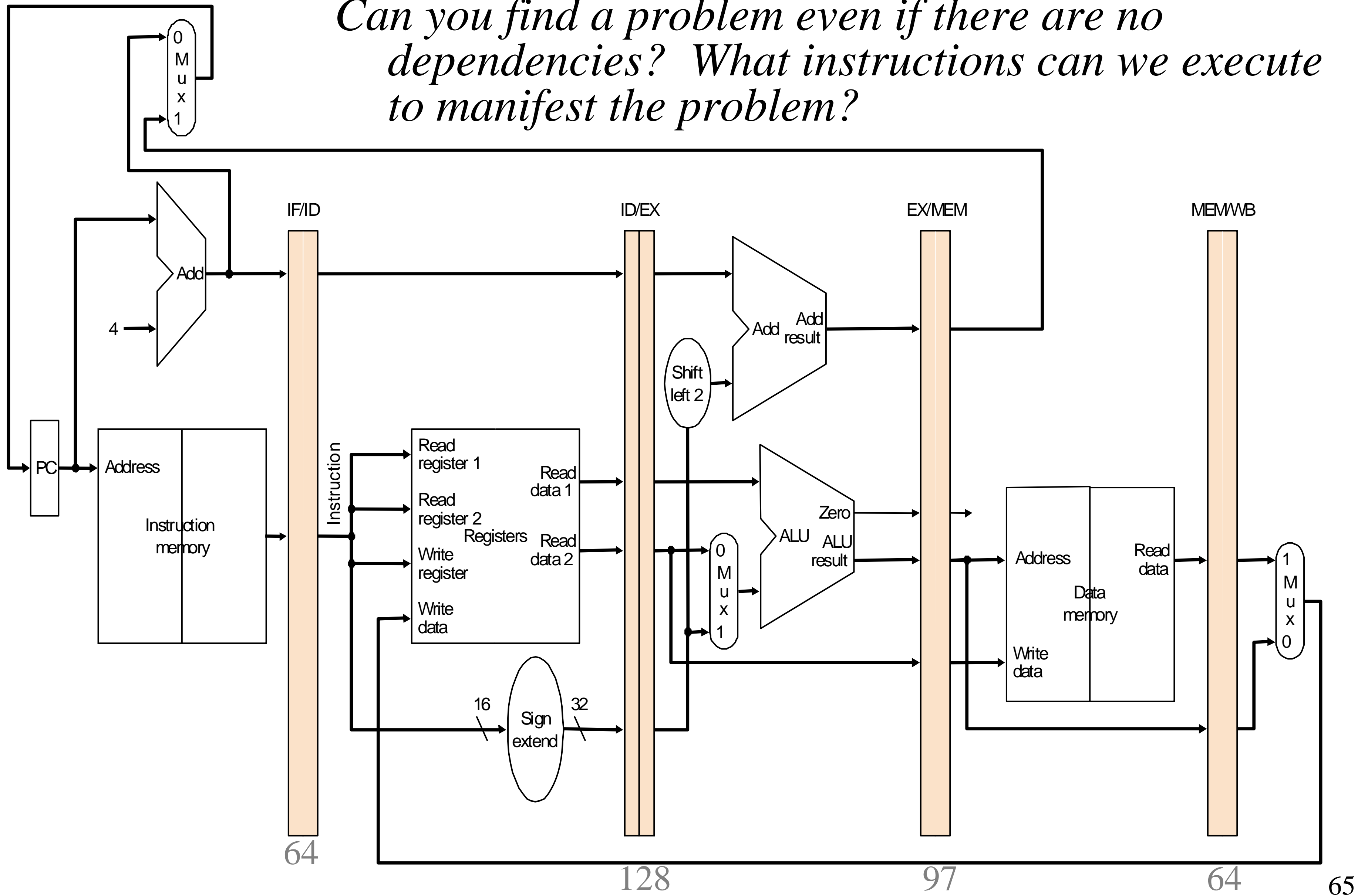
Basic Idea

What do we need to add to actually split the datapath into stages?

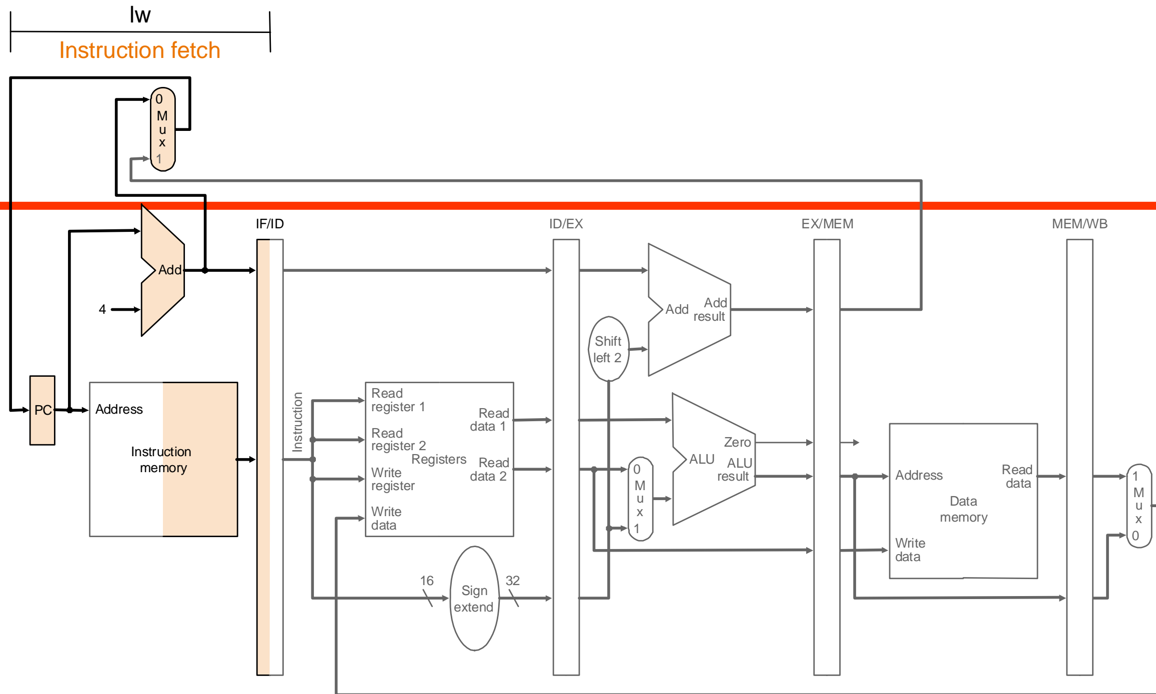


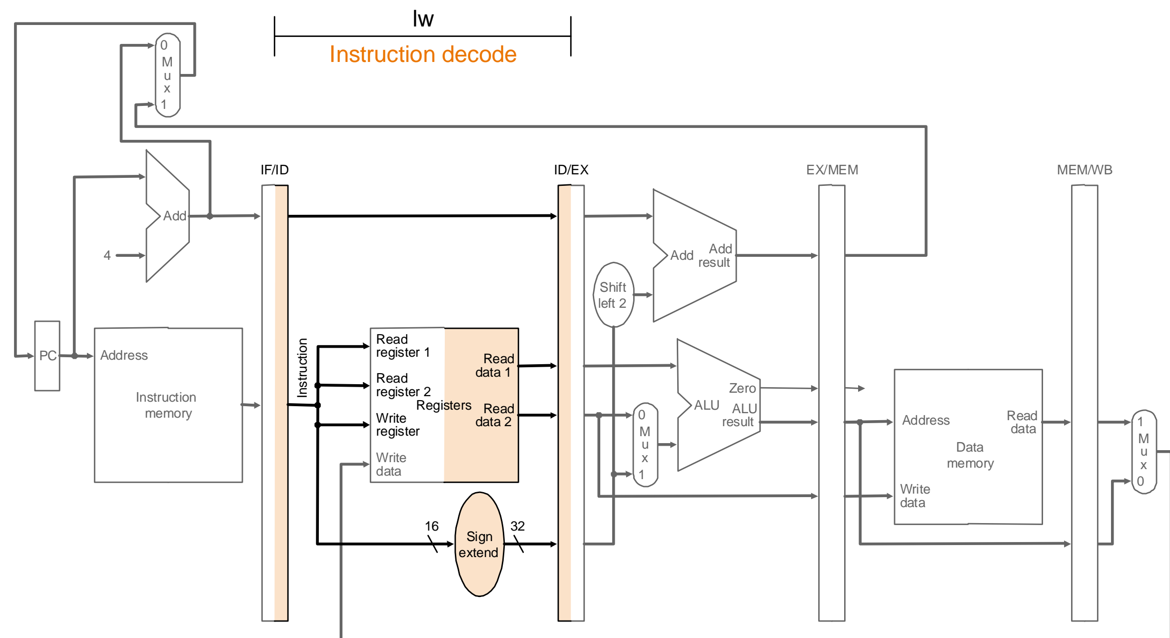
Pipelined Datapath

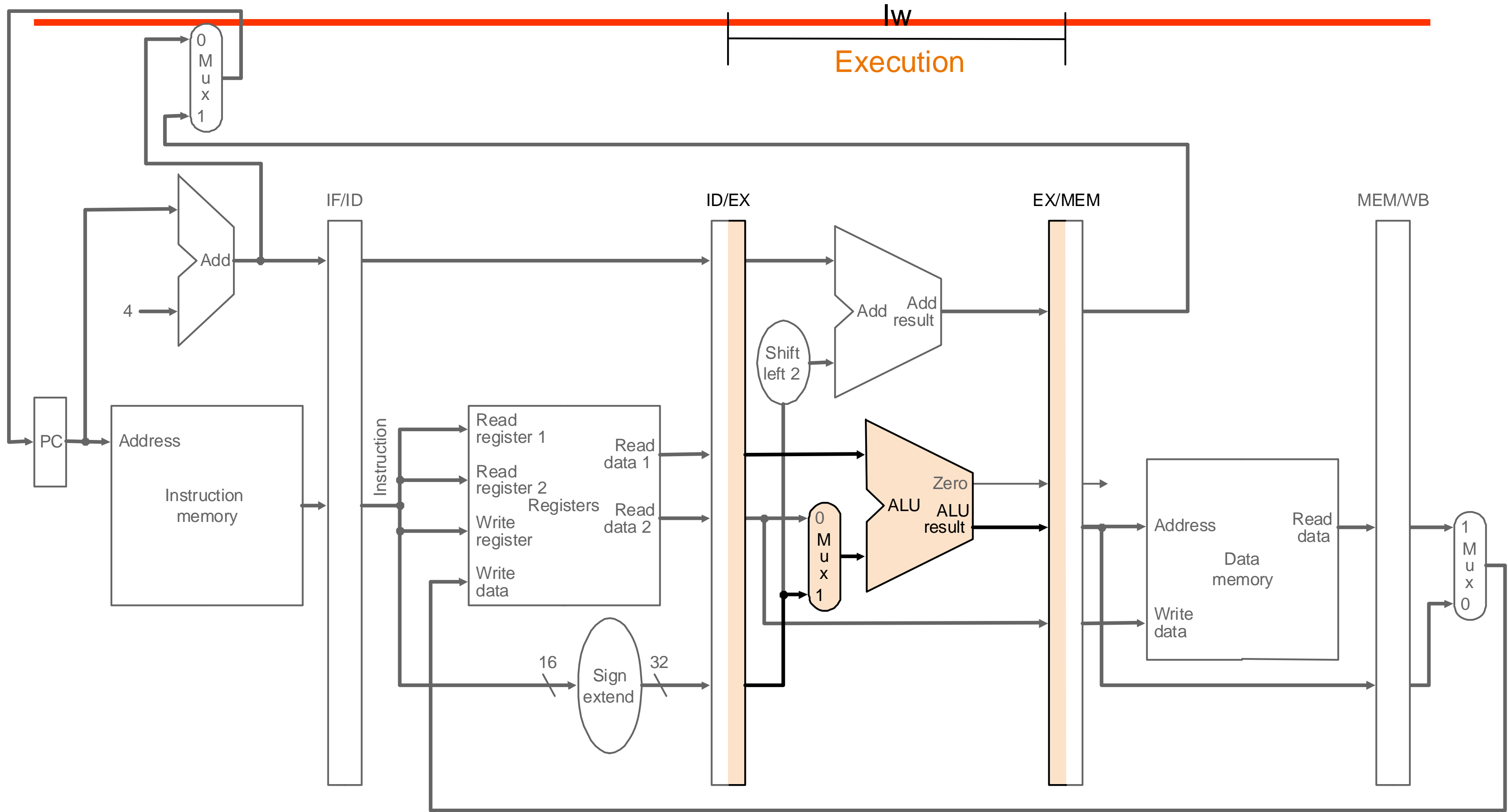
Can you find a problem even if there are no dependencies? What instructions can we execute to manifest the problem?



lw



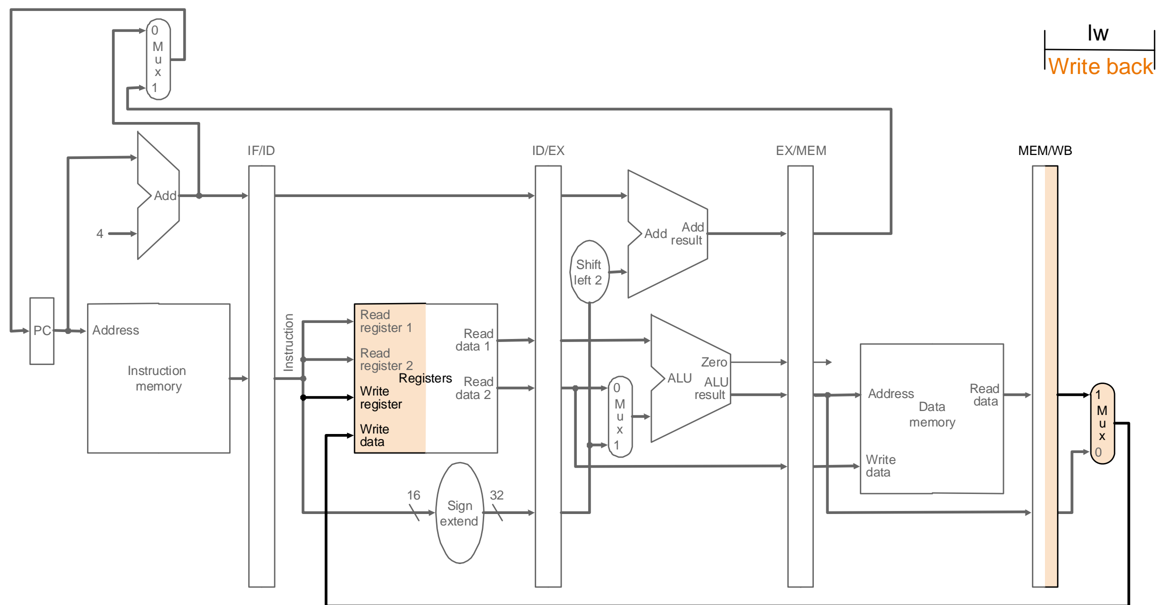



lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
0
M
u
x
1
Add
4
PC
Address
Instruction
memory
Instruction
Read
register 1
Read
register 2
Write
register
Write
data
Registers
Read
data 1
Read
data 2
16
Sign
extend
32
Shift
left 2
Add
Add
result
0
M
u
x
1
ALU
Zero
ALU
result
Address
Data
memory
Write
data
Read
data
1
M
u
x
0

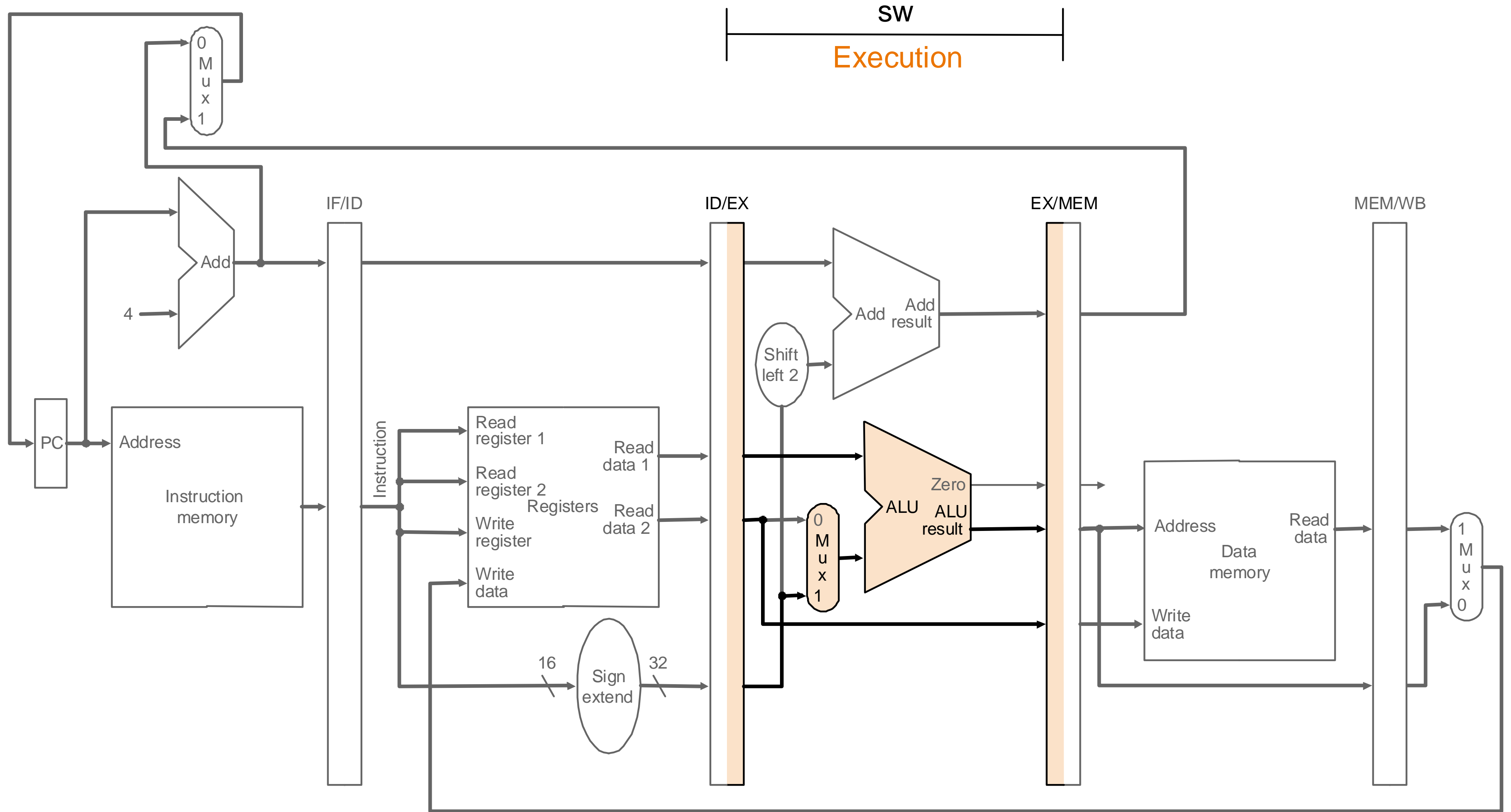
lw

Memory

IF/ID
ID/EX
EX/MEM
MEM/WB
0 M u x 1
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
0 M u x 1
Address
Data memory
Write data
Read data
1 M u x 0



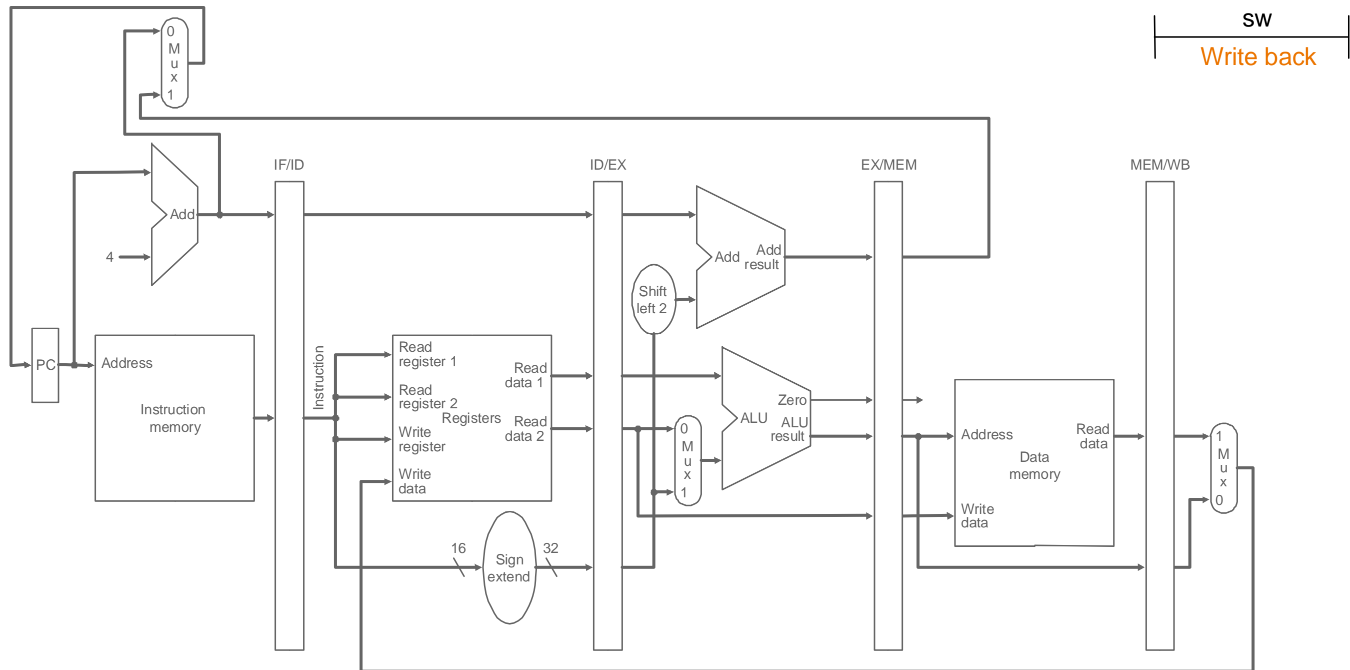
SW



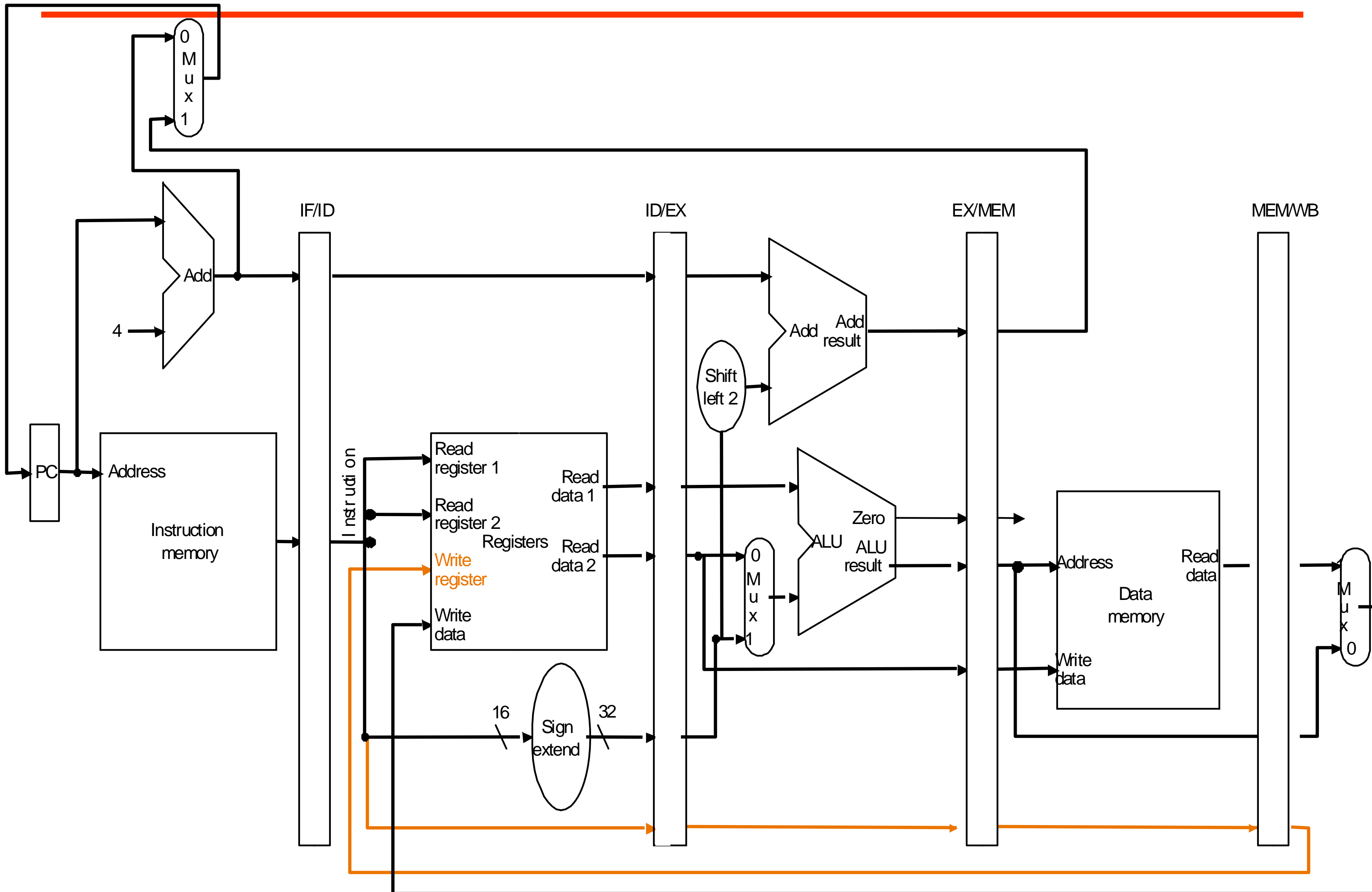
SW

SW
Memory

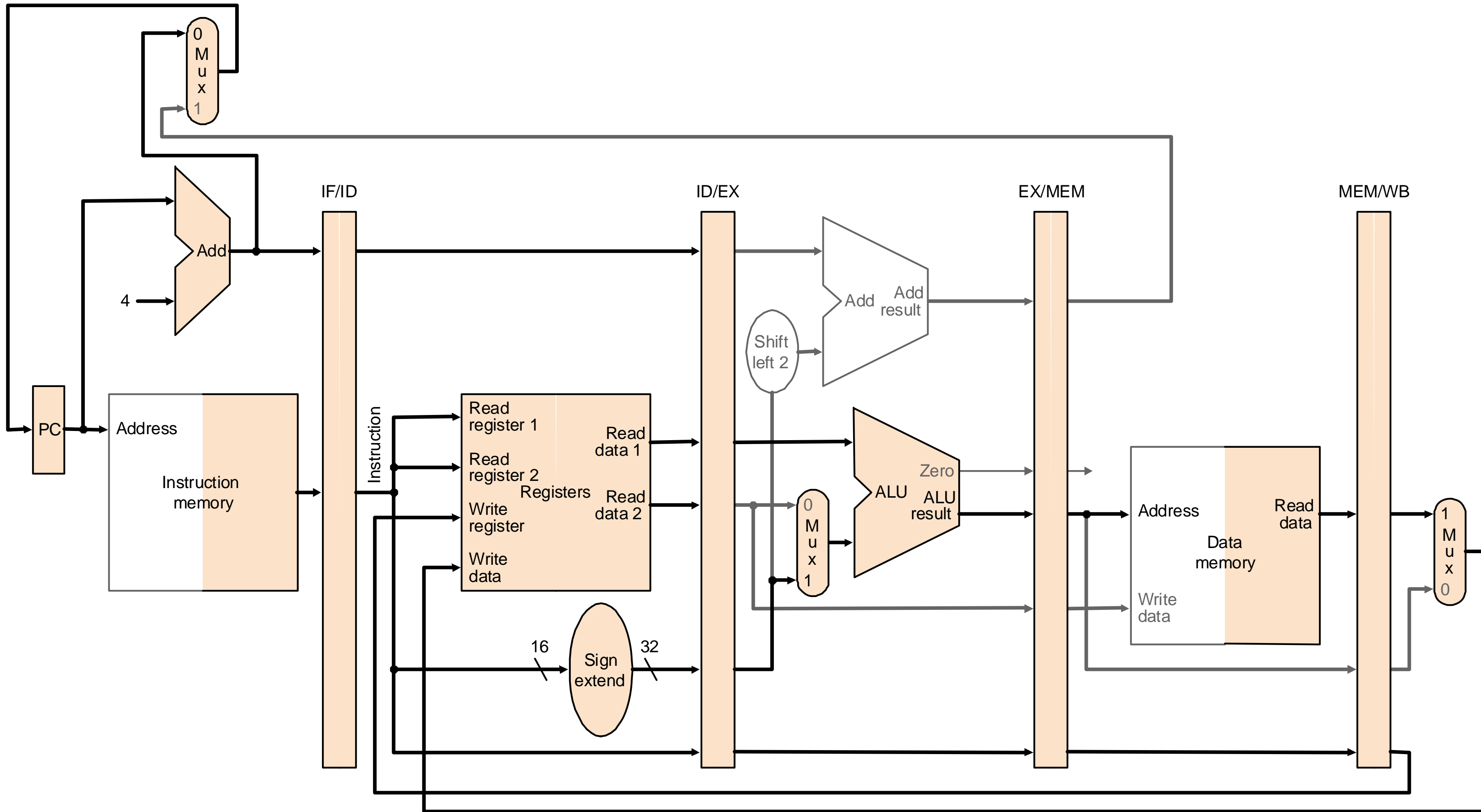
0 Mux 1
Add
4
PC
Address
Instruction memory
IF/ID
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign extend
32
ID/EX
Shift left 2
Add
Add result
0 Mux 1
ALU
Zero
ALU result
EX/MEM
Address
Data memory
Write data
Read data
MEM/WB
1 Mux 0



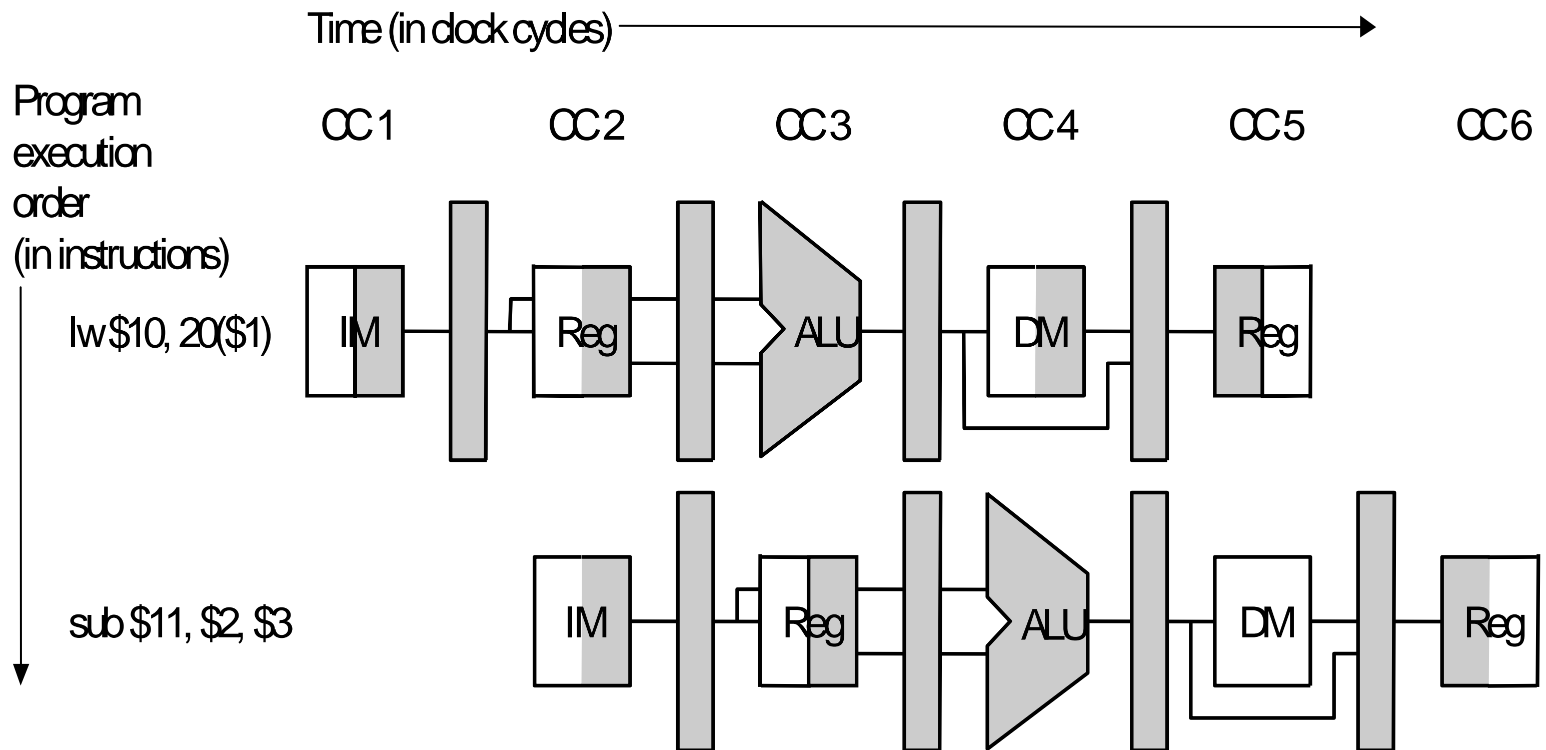
Corrected Datapath (lw)



Datapath used in all the five stages of lw

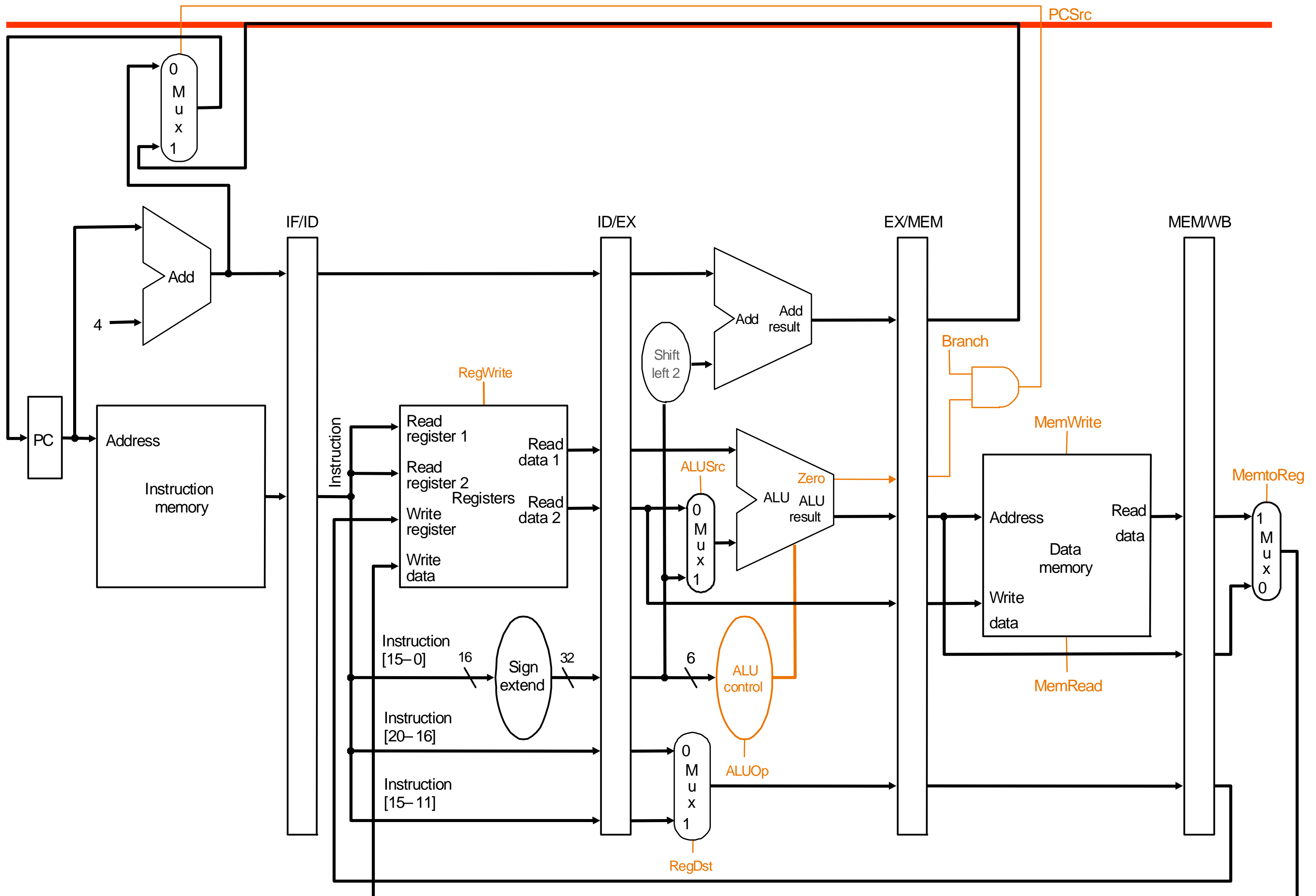


Graphically Representing Pipelines



- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

Pipeline Control



Pipeline control

- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back

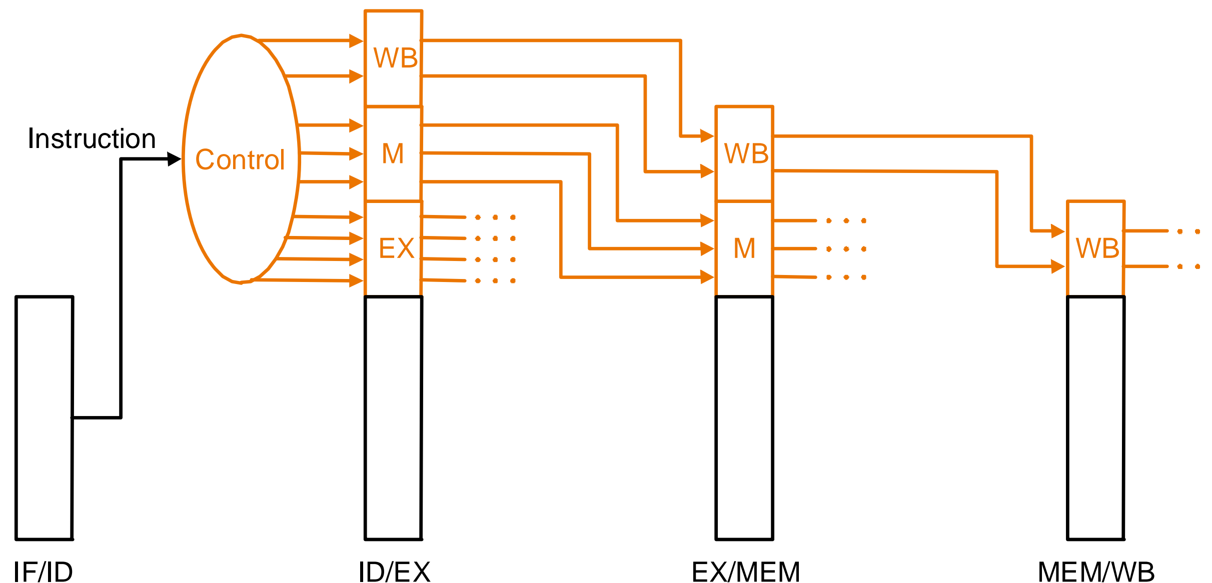
- How would control be handled in an automobile plant?
  - a fancy control center telling everyone what to do?
  - should we use a finite state machine?

Pipeline Control

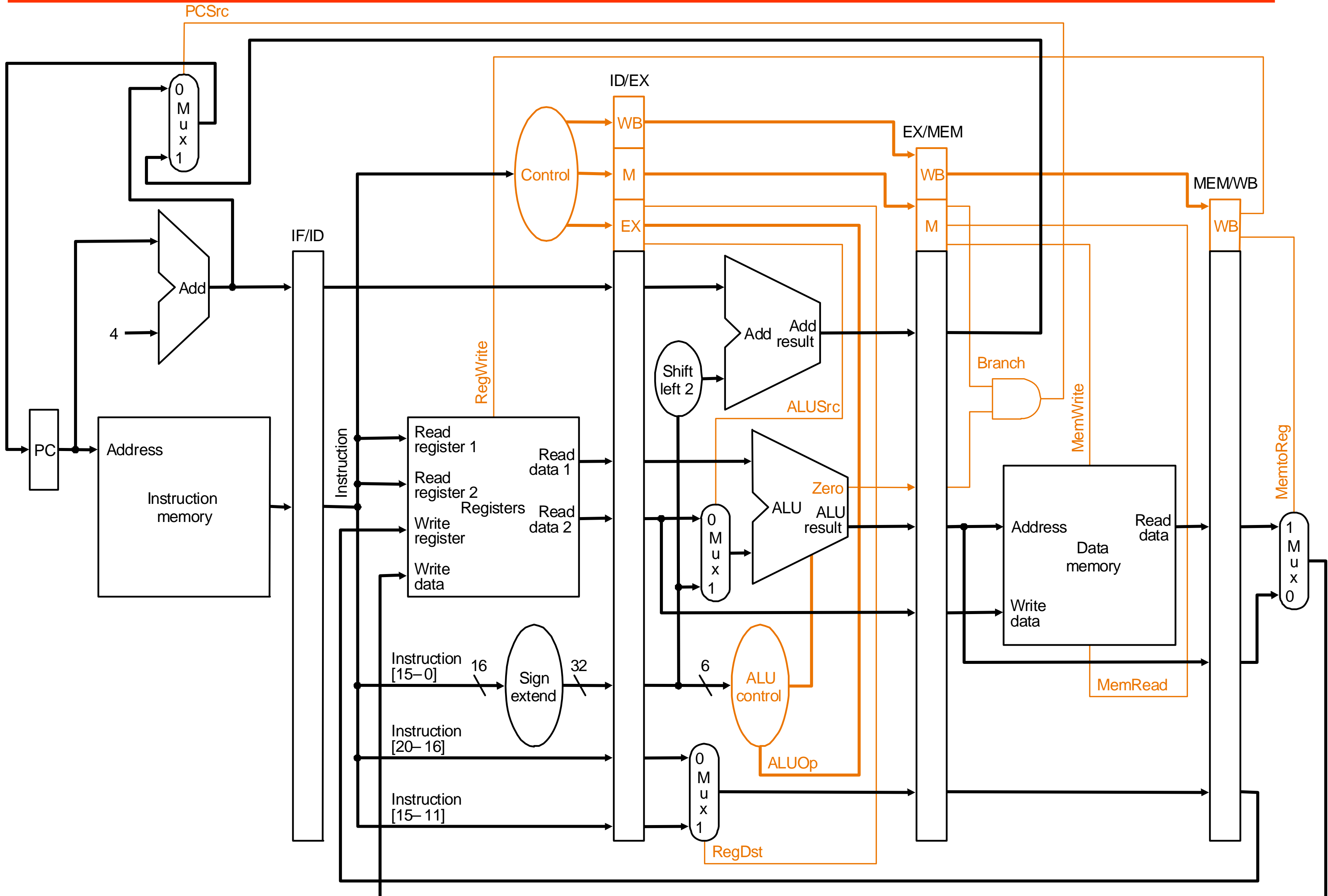
Pass control signals along just like the data

No control signals for IF and ID, but only for the remaining three stages

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X



Datapath with Control



Hazards

- Hazards: problems due to pipelining

- Hazard types:
- Structural
  - same resource is needed multiple times in the same cycle
- Data
  - data dependencies limit pipelining
- Control
  - next executed instruction is not the next specified instruction

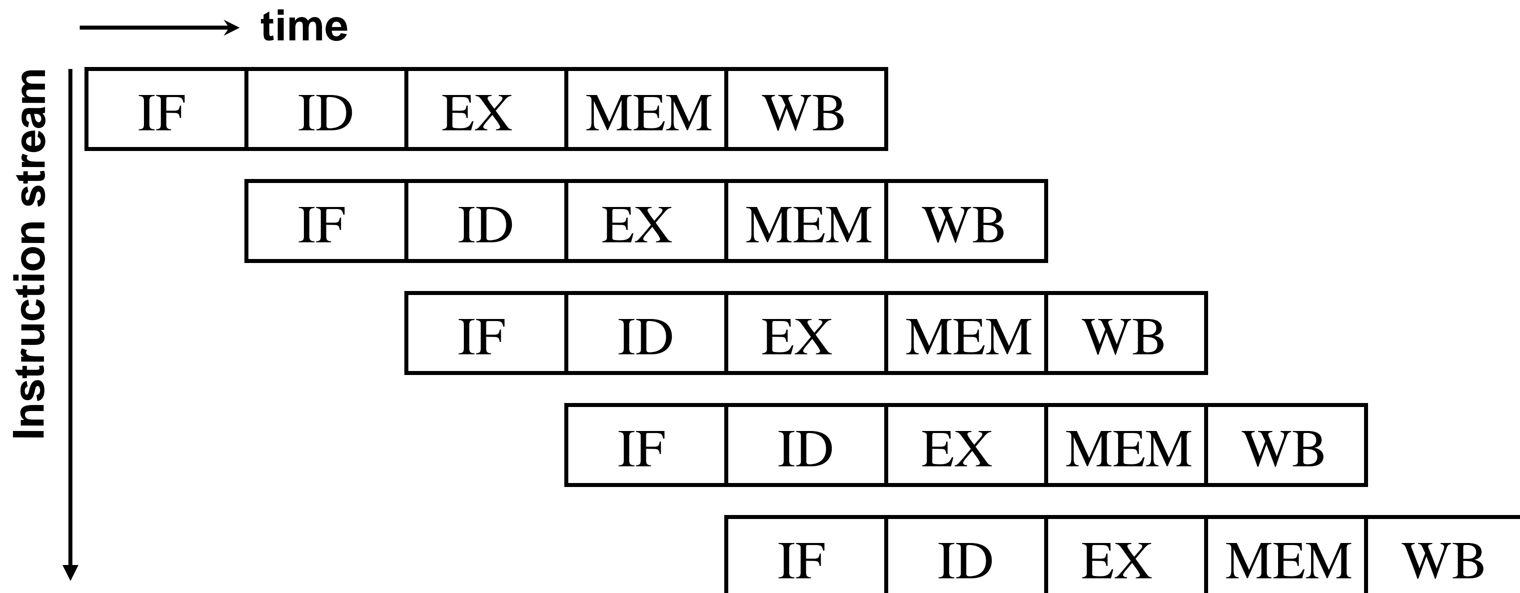
Structural hazards

- Examples:
- Two accesses to a single ported memory
- Two operations need the same function unit at the same time
- Two operations need the same function unit in successive cycles, but the unit is not pipelined

- Solutions:
- stalling
- add more hardware

Structural hazards on MIPS

Do we have structural hazards on our simple MIPS pipeline?



Data hazards

- Data dependencies:
  - RaW (read-after-write)
  - WaW (write-after-write)
  - WaR (write-after-read)
- Hardware solution:
  - Forwarding / Bypassing
  - Detection logic
  - Stalling
- Software solution: Scheduling

Control hazards

- Control operations may change the sequential flow of instructions
  - branch
  - jump
  - call (jump and link)
  - return
  - exception

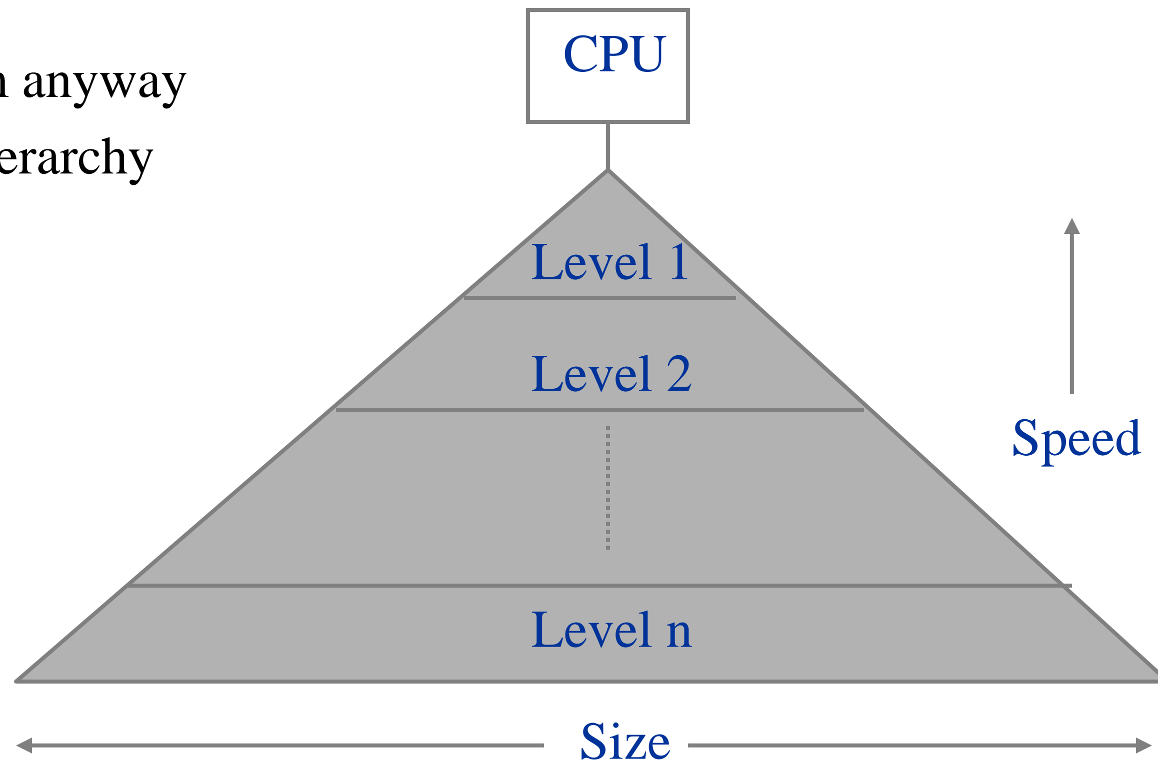
Memories: Review

- SRAM:
  - value is stored on a pair of inverting gates
  - very fast but takes up more space than DRAM (4 to 6 transistors)
  - access time: 5-25 ns
  - Cost (US$) per MByte in 1997: 100 to 250
  -
- DRAM:
  - value is stored as a charge on capacitor (must be refreshed)
  - very small but slower than SRAM (factor of 5 to 10)
  - access time: 60-120 ns
  - Cost (US$) per MByte in 1997: 5 to 10

Memory Hierarchy: why?

- Users want large and fast memories!
  SRAM access times are 2 - 25ns at cost of $100 to $250 per Mbyte.
  DRAM access times are 60-120ns at cost of $5 to $10 per Mbyte.
  Disk access times are 10 to 20 million ns at cost of $.10 to $.20 per Mbyte.

- Try and give it to them anyway
  - build a memory hierarchy



Memory Hierarchy: requirements

- If level is closer to Processor, it must…
  - Be smaller
  - Be faster
  - Contain a subset (most recently used data) of lower levels beneath it
  - Contain all the data in higher levels above it
- Lowest Level (usually disk or the main memory) contains all the available data

Locality

- A principle that makes having a memory hierarchy a good idea

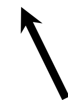
- If an item is referenced,
  - ***temporal locality***: it will tend to be referenced again soon
  - ***spatial locality*** : nearby items will tend to be referenced soon.

- Our initial focus: two levels (upper, lower)
  - block: minimum unit of data
  - hit: data requested is in the upper level
  - miss: data requested is not in the upper level

Cache

- Two issues:
  - How do we know if a data item is in the cache?
  - If it is, how do we find it?

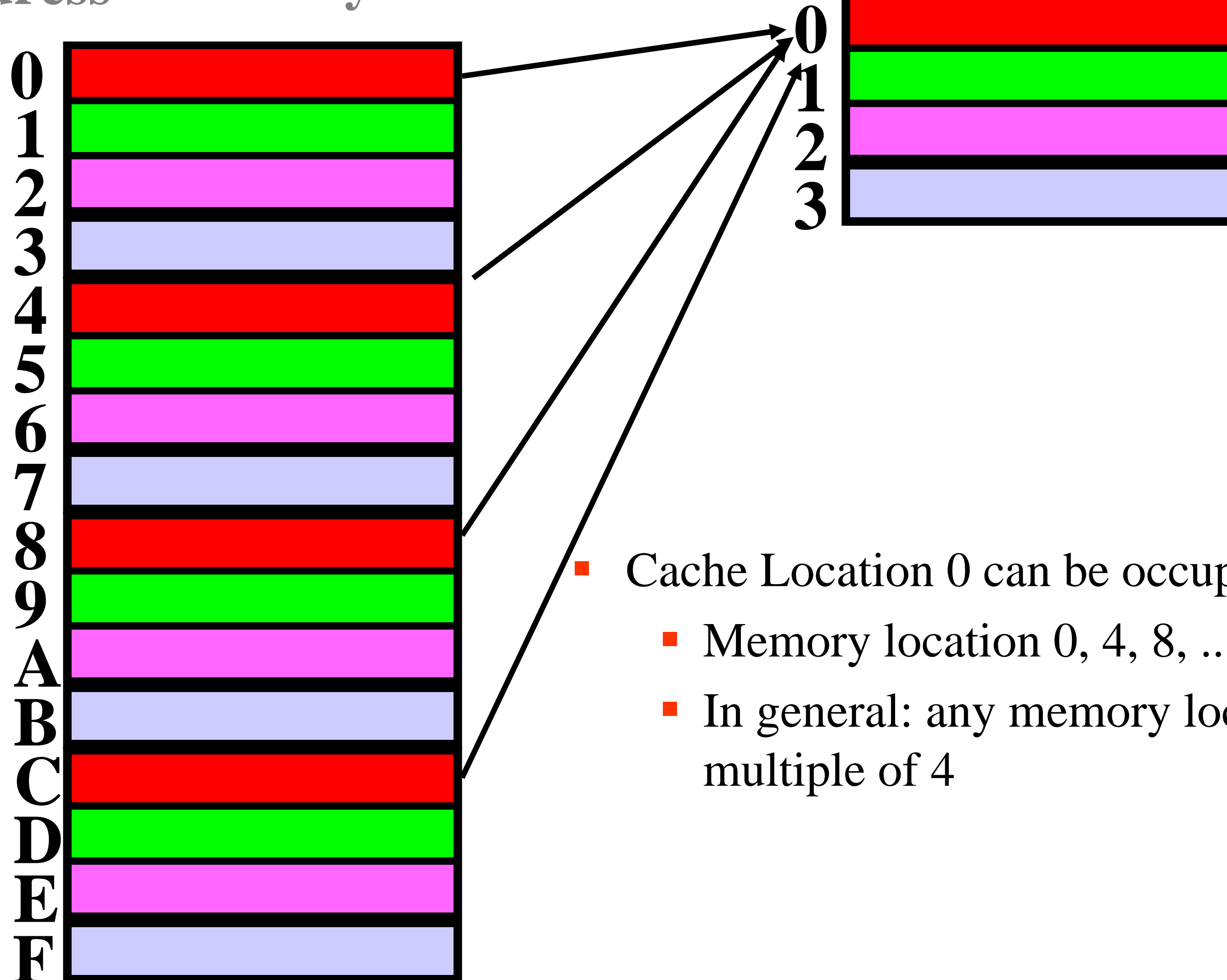
- Our first example:
  - *block size* is one word of data
  - "*direct mapped*"



For each item of data at the lower level,
there is exactly one location in the cache where it might be.

e.g., lots of items at the lower level share locations in the upper level

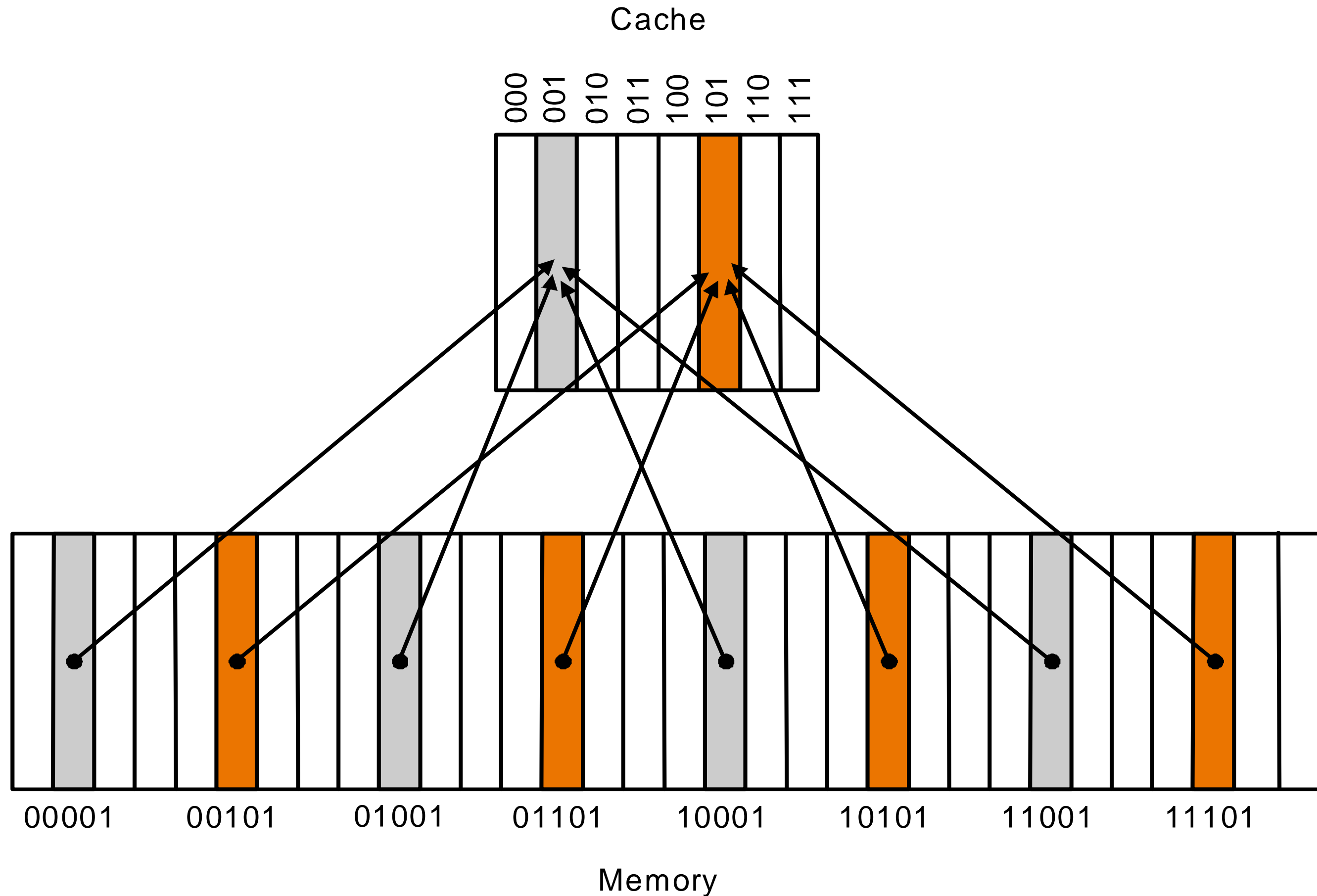
Direct Mapped Cache



- Cache Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ...
  - In general: any memory location that is multiple of 4

Direct Mapped Cache

- Mapping: address is modulo the number of blocks in the cache

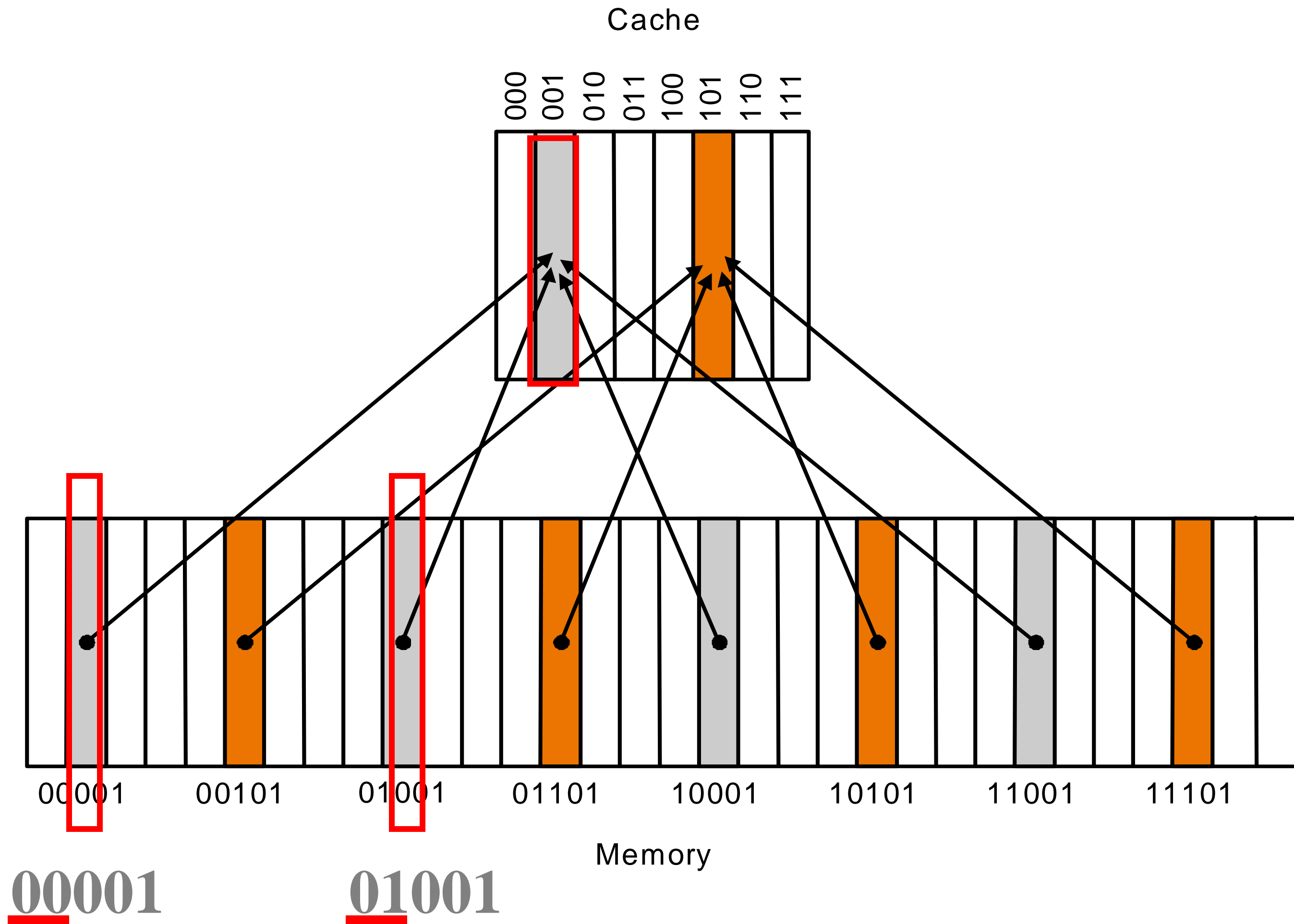


Issues with Direct Mapped Caches

- Since multiple memory addresses map to same cache index, how do we tell which one is in there?
- What if we have a block size > 1 byte?
  - Solution: divide memory address into three fields

tag to check if have correct block	index to select block	byte offset within block

Check if we have the correct block



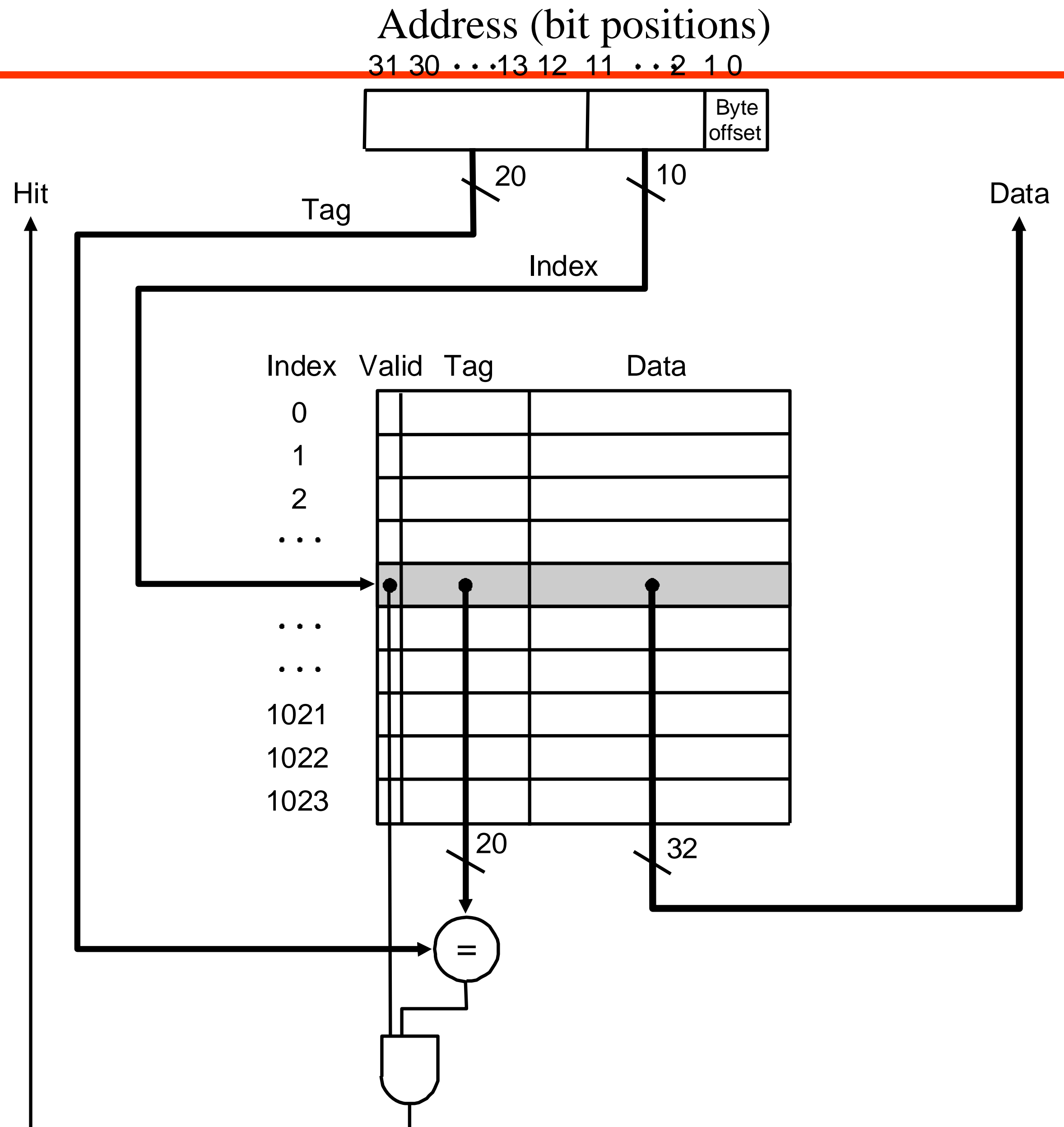
Direct Mapped Caches: Terminology

- All fields are read as unsigned integers.
- Index: specifies the cache index (which "row" of the cache we should look in)
- Offset: once we've found correct block, specifies which byte within the block we want
- Tag: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

tag to check if have correct block	index to select block	byte offset within block

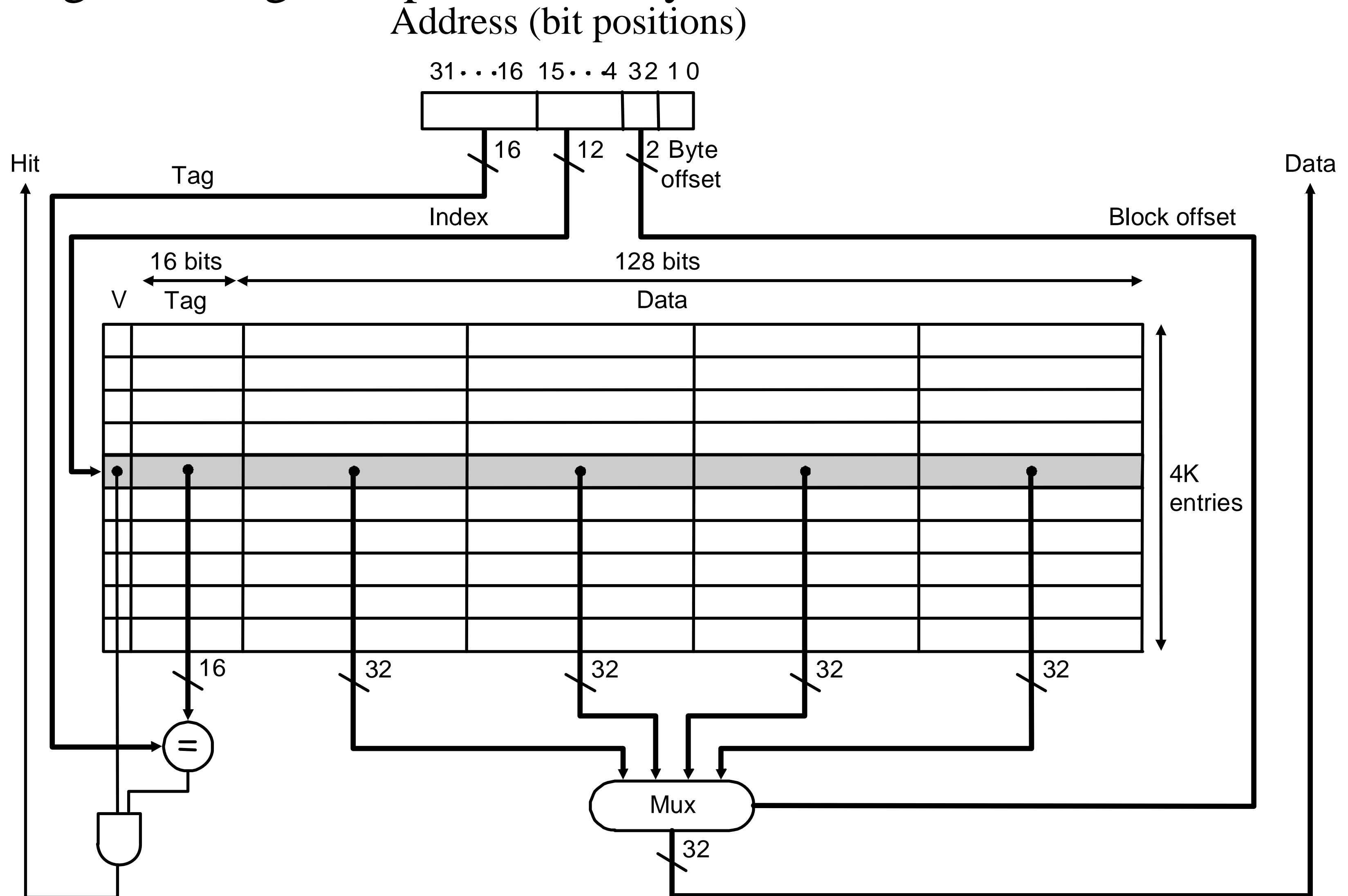
Direct Mapped Cache

- For MIPS:



Direct Mapped Cache

- Taking advantage of spatial locality:



Hits vs. Misses

- Read hits
  - this is what we want!
- Read misses
  - stall the CPU, fetch block from memory, deliver to cache, restart the load instruction
- Write hits:
  - can replace data in cache and memory (*write-through*)
  - write the data only into the cache (*write-back* the cache later)
- Write misses:
  - read the entire block into the cache, then write the word (*allocate* on write miss)
  - do not read the cache line; just write to memory (*no allocate* on write miss)

Improving performance

- Two ways of improving performance:
  - decreasing the miss ratio: *associativity*
  - decreasing the miss penalty: *multilevel caches*

Decreasing miss ratio with associativity

(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

2 blocks / set

block

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

4 blocks / set

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

8 blocks / set

Block replacement policy

- In a direct mapped cache, when a miss occurs, the requested block can go only at one position.

- In a set-associative cache, there can be multiple positions in a set for storing each block. If all the positions are filled, which block should be replaced?
- Least Recently Used (LRU) Policy
- Randomly choose a block and replace it

References

- Computer Organization and Design by Patterson and Hennessy (for the basic topics that we discussed today)
- Computer Architecture – A Quantitative Approach by Hennessy and Patterson, Chapters 3, 4 and 5 (for Superscalar and VLIW processors and memory hierarchy design)
- Virtual Machines by Smith and Nair, Appendix A (Real Machines) for overview of Computer Architecture and OS

Next Class

- Some System ISA issues (especially memory management)
- Overview of PowerPC ISA
- Overview of Intel IA-32 ISA
- Implementation of Process VMs using interpretation