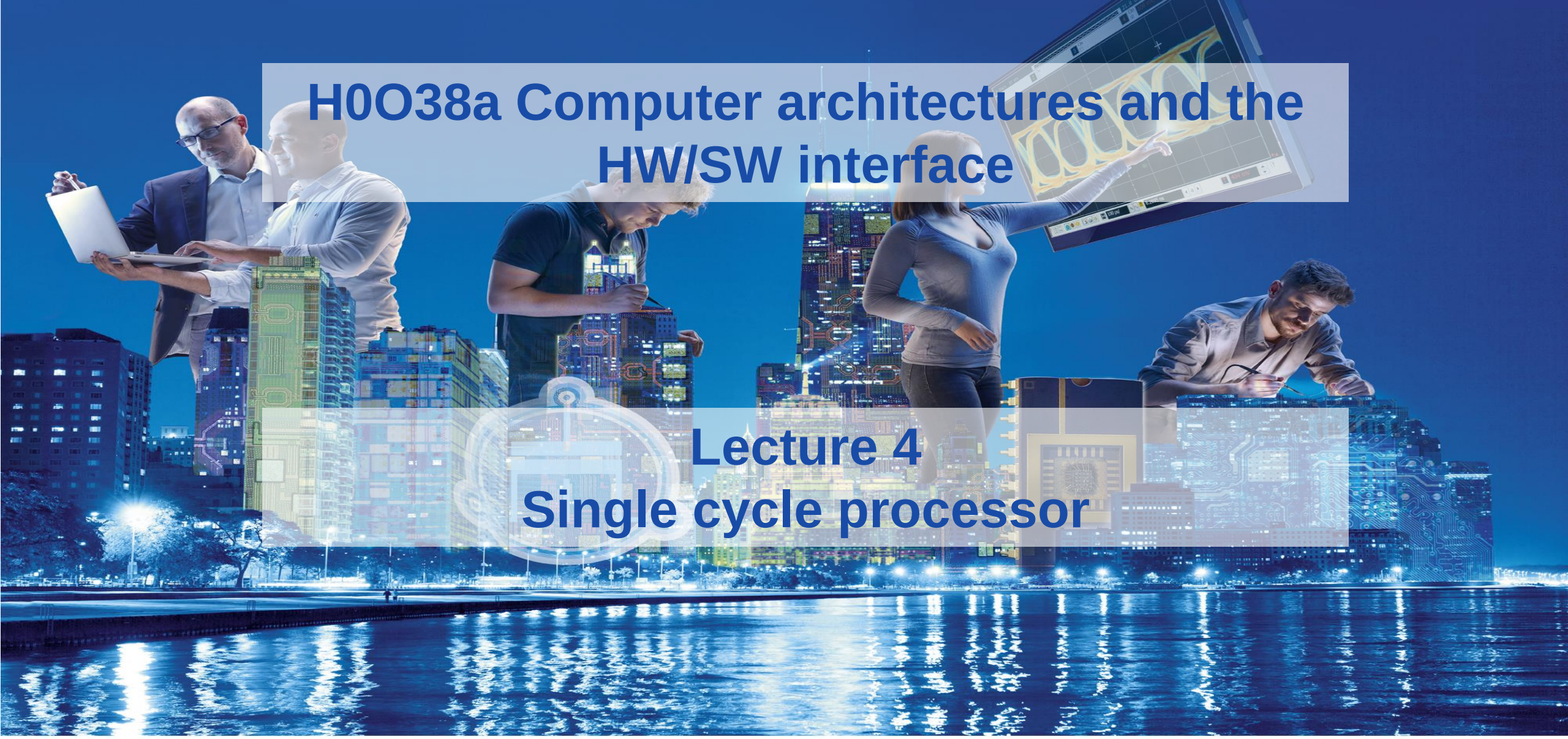


H0O38a Computer architectures and the HW/SW interface

Lecture 4
Single cycle processor

KU LEUVEN

M. Verhelst. version 2022.1

KU LEUVEN micas

Lecture series overview

- Part 1 - W. Dehaene
  - Basics about processors, performance, ...
  - Instructions and instruction set
  - Arithmetic
- Part 2 - Marian verhelst
  - internals of the microprocessor
  - micro-architecture

  + Exercise
- Part 3 - Caching and Memory aspects
  - Memory
  - Cache
  - Virtual memory

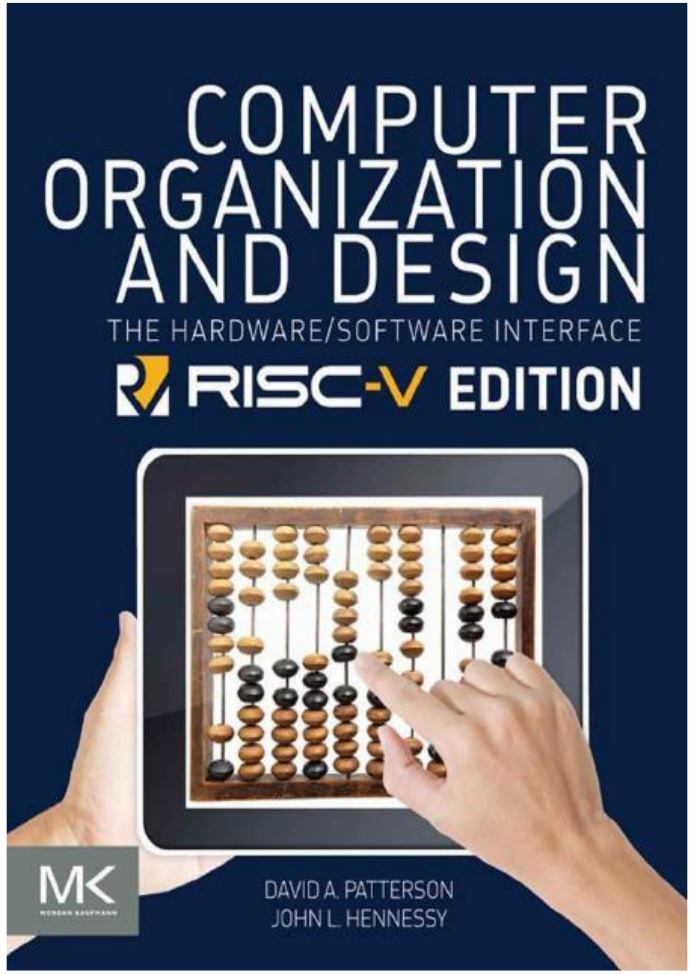
  + Exercise

Lecture series overview

- Part 2 - Marian verhelst
  - Lecture 4: Single cycle processor
  - Lecture 5: Pipelining and hazards
  - Lecture 6: Parallelism: SIMD, VLIW, Super-scalar

  + Exercise

Supporting material - the book



eBook

see toledo

Chapter 4 (and 6)

Lecture overview

- Lecture 4: Single cycle processor
  - Recap and goal
  - Building up the processor
  - Translating and starting a program
  - Pipelining basics

Lecture overview

- Lecture 4: Single cycle processor
  - **Recap and goal**
  - Building up the processor
  - Translating and starting a program
  - Pipelining basics

Recap Lecture 1

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Instr count	CPI	Clock cycle time

Program
Compiler
Instruction set

Implementation!

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Simple program compilation

```
int accum(int n)
{
        sum = 0;
        for (i = 0; i < y; i++)
            sum = sum + x;
}
```

compiler

```
                add x3, x0, x0
                add x4, x0, x0
  LoopBegin:
                beq x4, x2, LoopEnd
                add x3, x3, x1
                addi x4, x4, 1
                beq  x0,  x0, LoopBegin
  LoopEnd:
```

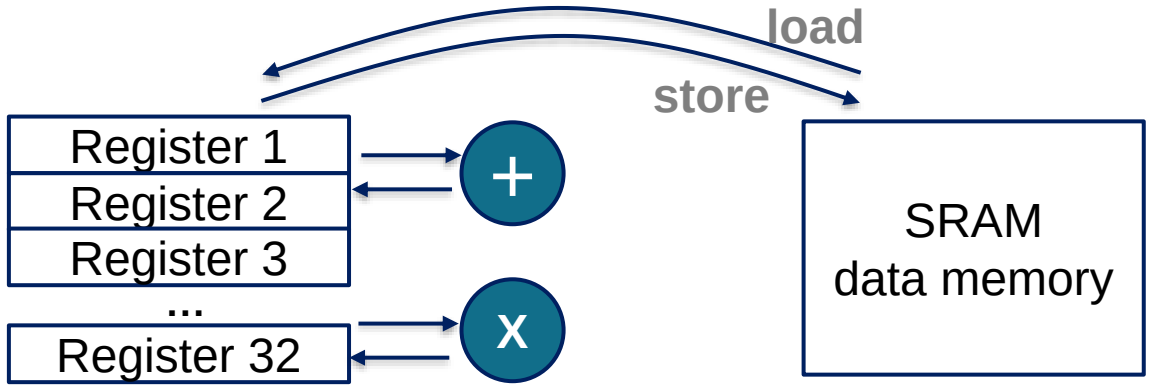
assembler

```
00000000101000010000000000011000
00000000000011000000110000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Put in instruction memory

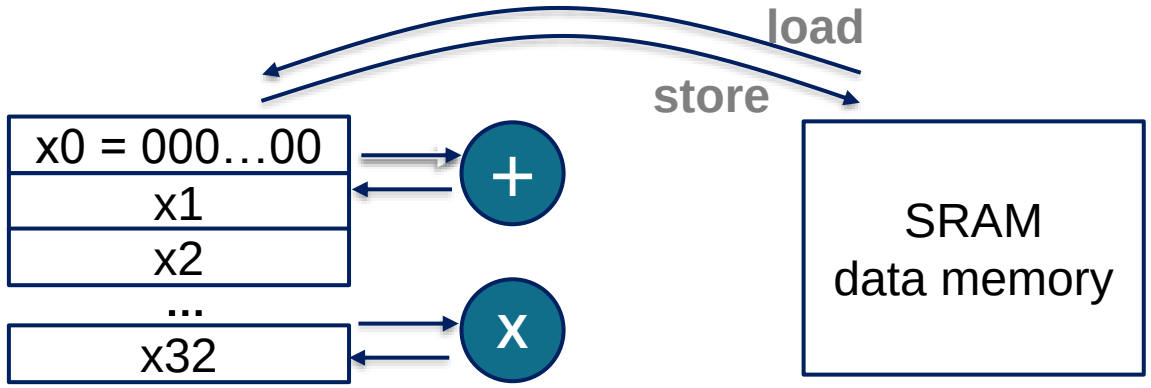
execute on HW

Registers vs memory (2)



- Result: A processor will be designed to only compute with data stored in registers (= fast access)
  - If we need to compute with data stored in memory, we will have to transfer it from the memory to the registers first, and later put the result back

Registers vs memory (3)



- Registers are always referenced with “x” (or “$”) and a register name.
- Not all 32 registers can be used for the same thing!
  - **‘temporary’, registers** are used for local computations, **‘saved’ registers** are used across function calls), there is a reg for stack pointers, …
  - x0 is a constant “0” and can not be overwritten

The Processor: Datapath & Control

- Let's design a simple processor ourselves…
- Reduced instruction set
  - memory-reference instructions: **`ld, sd`**
  - arithmetic-logical instructions: **`add, sub, addi`**
  - control flow instructions: **`beq, bneq`**

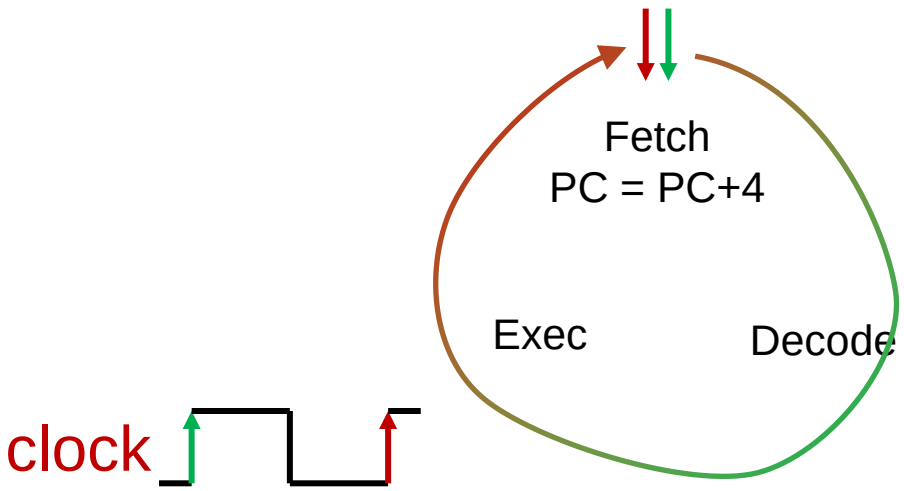
Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode

Lecture overview

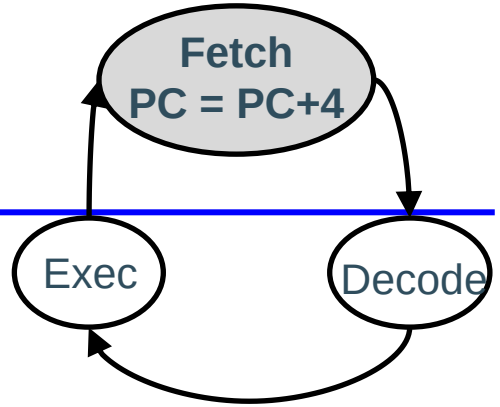
- Lecture 4: Single cycle processor
  - Recap and goal
  - **Building up the processor**
  - Translating and starting a program
  - Pipelining basics

The Processor: Datapath & Control

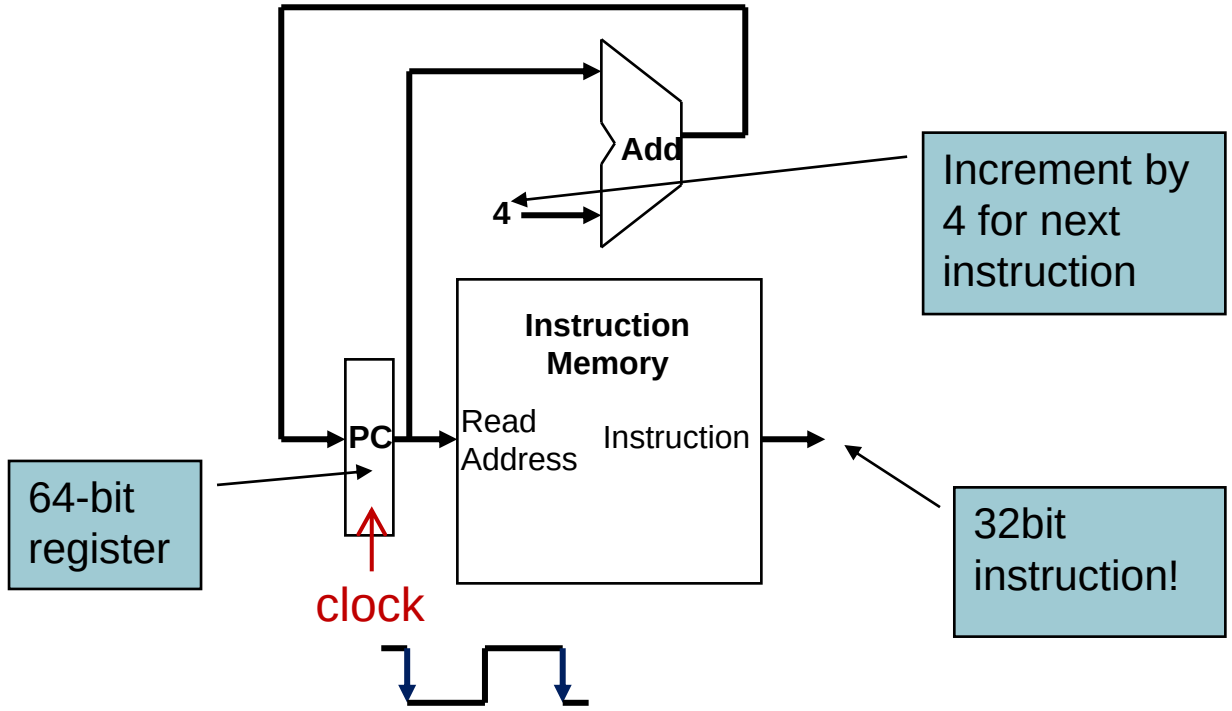
- Generic implementation: All instructions…
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction



Fetching Instructions

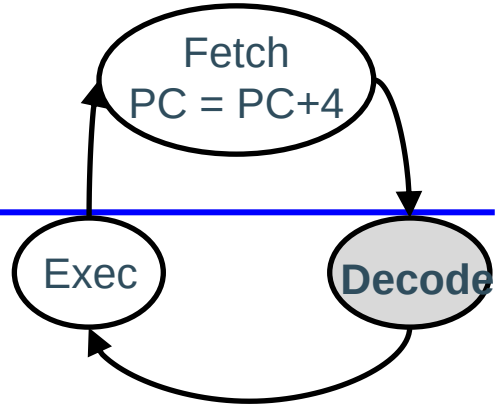


- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - PC = program counter = 64bit address where to read the 32bit instruction
  - updating the PC value to be the address of the next (sequential) instruction (+4)



Note: memory read = unclocked (or falling ege clock); memory write = clocked

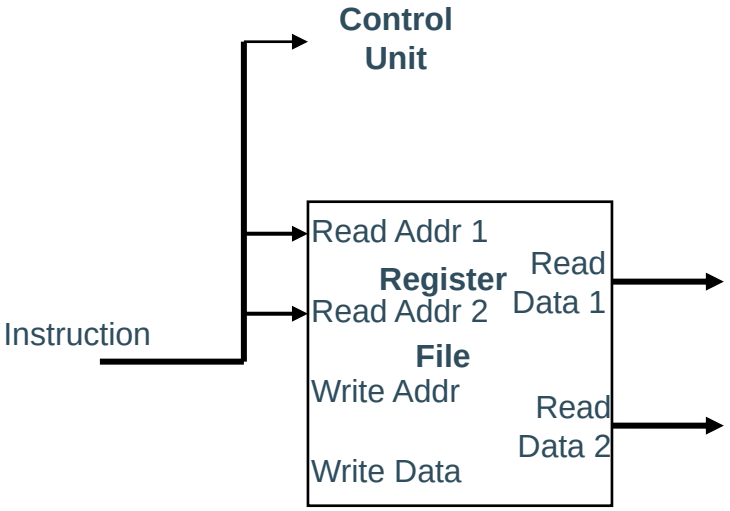
Decoding Instructions



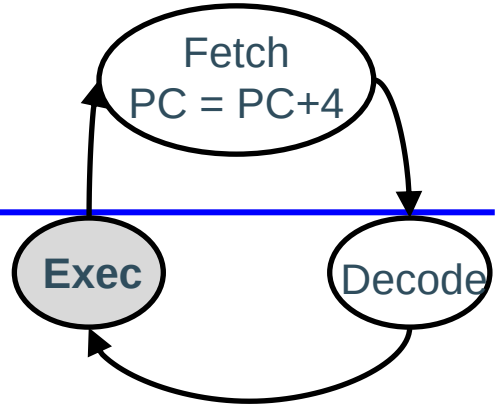
R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (Add)	0000000	00011	00010	000	00001	0110011	add x1,x2,x3
sub (Sub)	010000	00011	00010	000	00001	0110011	sub x1,x2,x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (Add Immediate)	001111101000		00010	000	00001	0010011	addi x1,x2, 1000
ld (Load doubleword)	001111101000		00010	011	00001	0000011	ld x1,1000

- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit
  - reading two values from the Register File

    Register File addresses are contained in the instruction



Executing R Format Operations




```
add x20, x21, x22
```

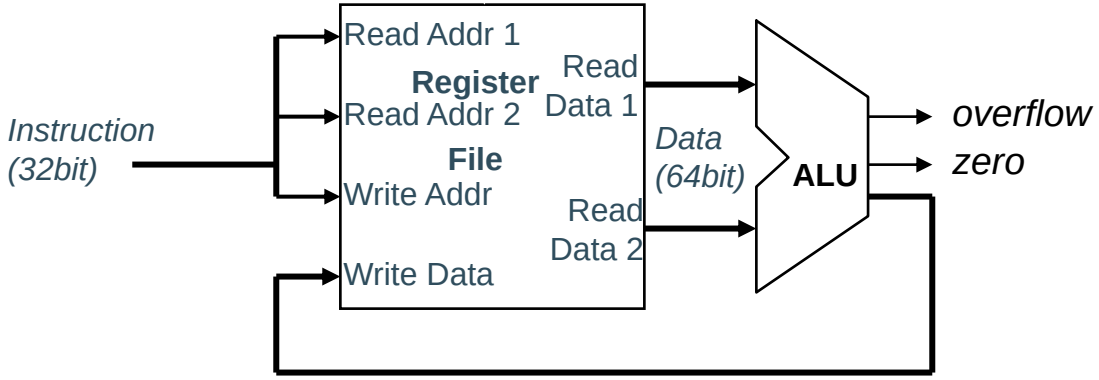
- R-type operations ( )

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (Add)	0000000	00011	00010	000	00001	0110011	add x1,x2,x3
sub (Sub)	010000	00011	00010	000	00001	0110011	sub x1,x2,x3

  - perform operation (opcode and funct7/3) on values in rs1 and rs2
  - store the result back into the Register File (into location rd)

RegWrite *ALU control*

Why needed?

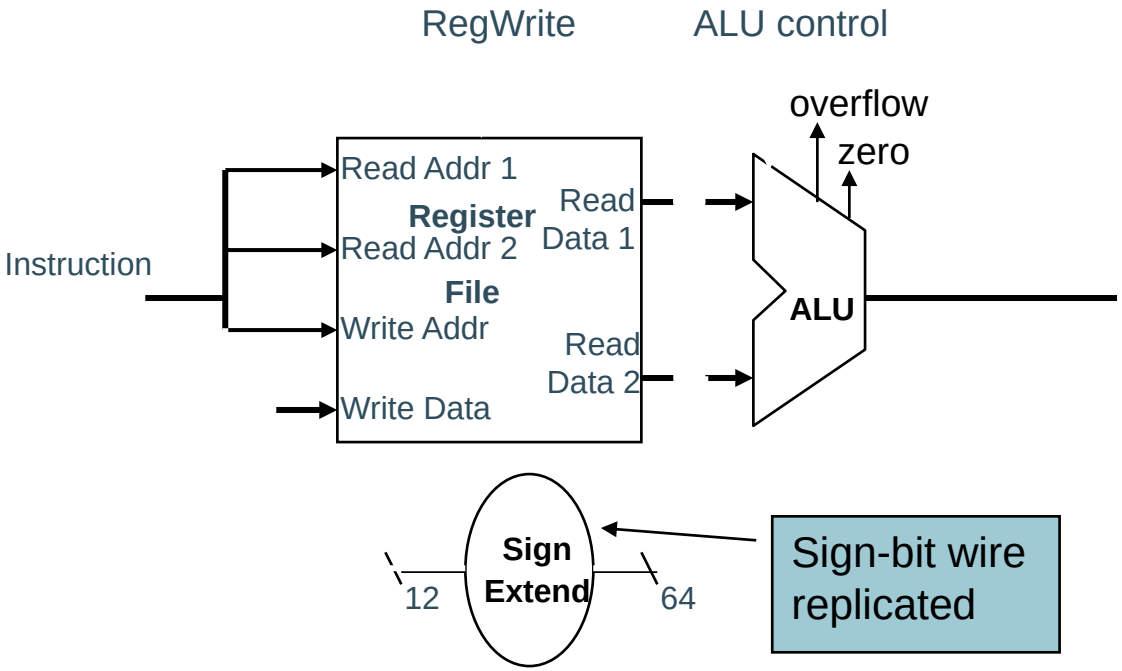


Executing I Format Operations

`addi x21, x22, 62`

- I-type operation
  - perform operation (opcode and funct3) on values in rs1 together with the immediate
  - store the result back into the Register File (into location rd)

I-type Instructions	immediate	rs1	funct3	rd	opcode	Example
addi (Add Immediate)	001111101000	00010	000	00001	0010011	addi x1,x2, 1000

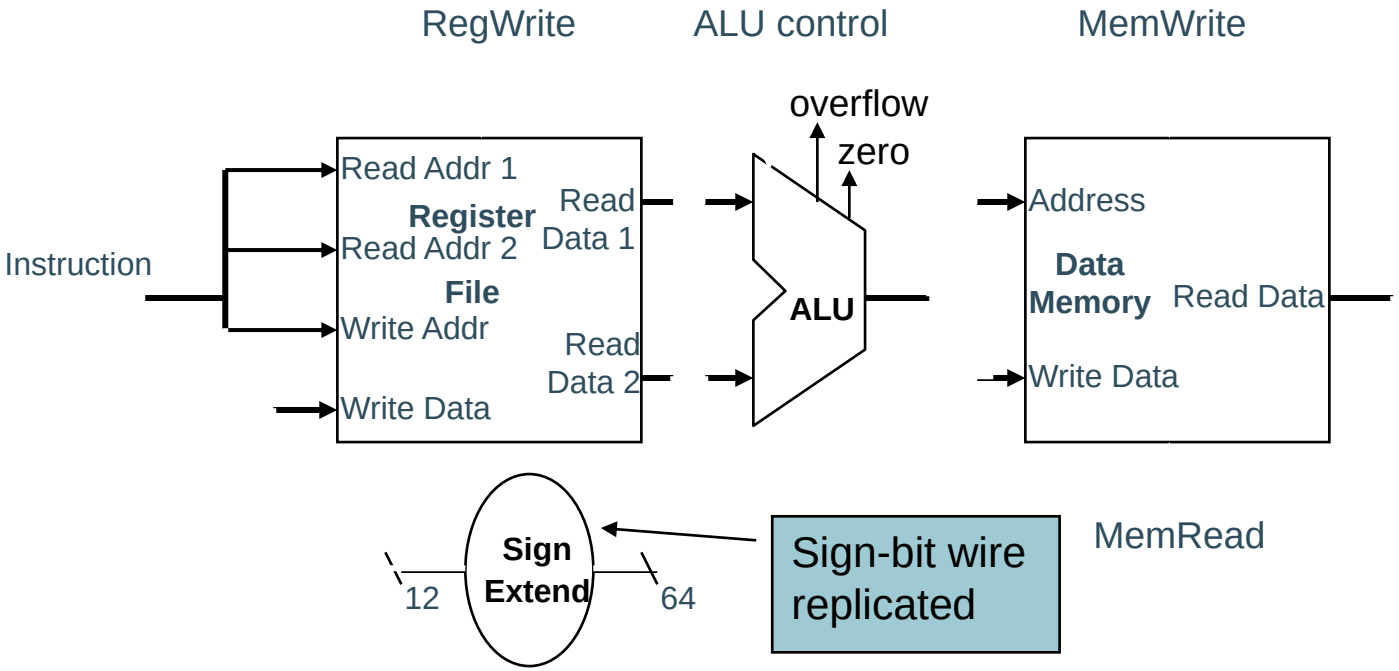


Executing Load Operations

```
ld x29, 24(x30)
```

I-type Instructions	immediate	rs1	funct3	rd	opcode	Example
ld (Load doubleword)	001111101000	00010	011	00001	0000011	ld x1,1000 (x2)

- Load operations involves
  - compute memory address with rs1 and immediate
  - Load value: read from the Data Memory, write to the Register File at location rd

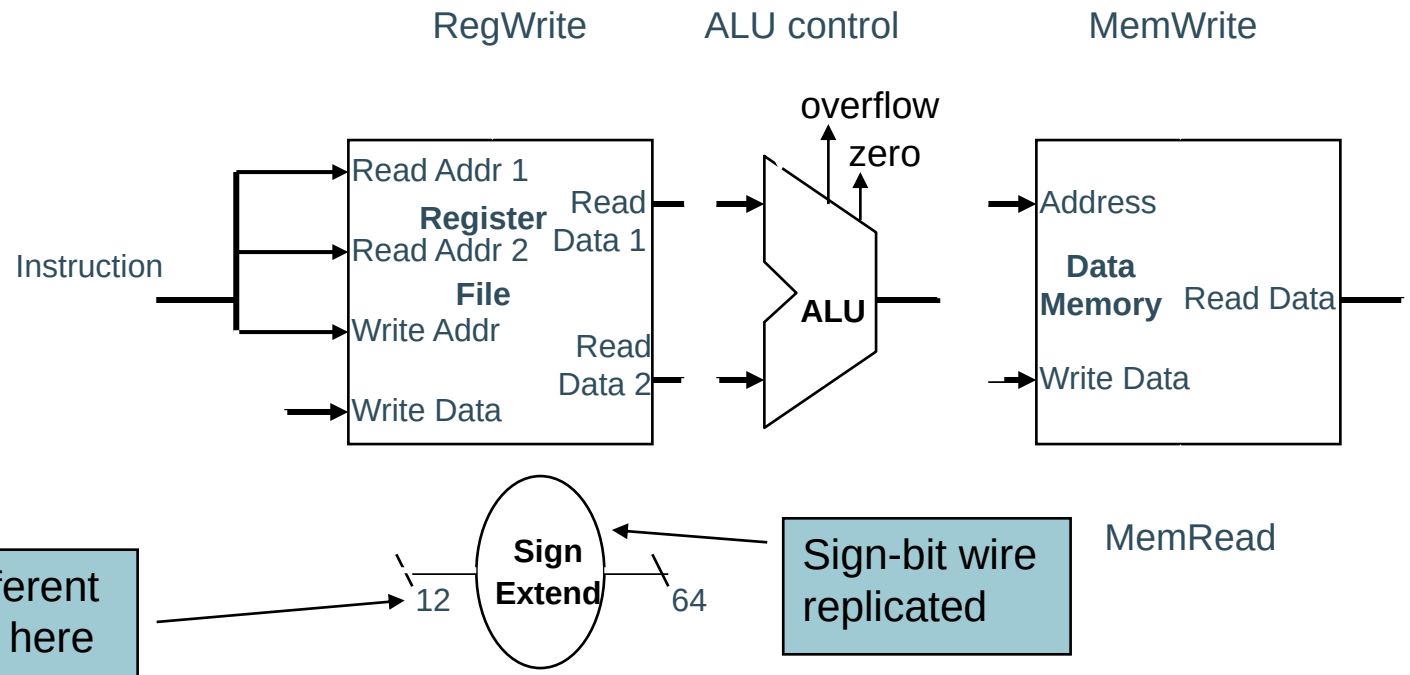


Executing Store Operations

`sd x29 32(x27)`

S-type Instructions	immed -iate	rs2	rs1	funct3	immed -iate	opcode	Example
sd (Store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1,1000(x2)

- Store operations involves
  - compute memory address with rs1 and immediate
  - Store value written to the Data Memory at location rs2 into the Data Memory



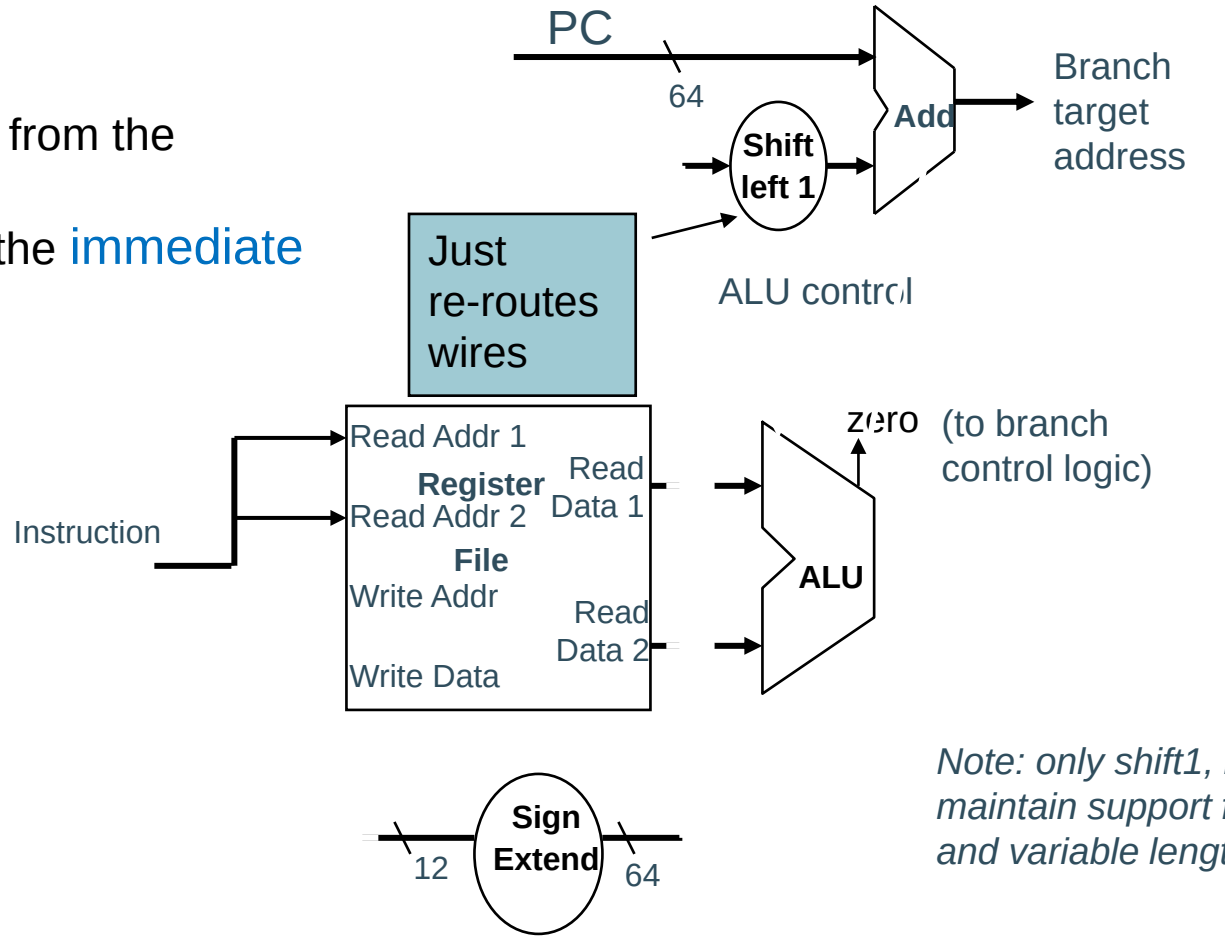
Executing Branch Operations

```
bne $t1, $s2, ADDR
```

S-type Instructions	immed -iate	rs2	rs1	funct3	immed -iate	opcode
bne / beq	0011111	00001	00010	011	01000	0100011

- Branch operations involves:
  - compare the operands rs1 and rs2 read from the Register File during decode for equality
  - compute the branch target address with the immediate
  - decide

➔can jump 2^10 instructions FW and BW

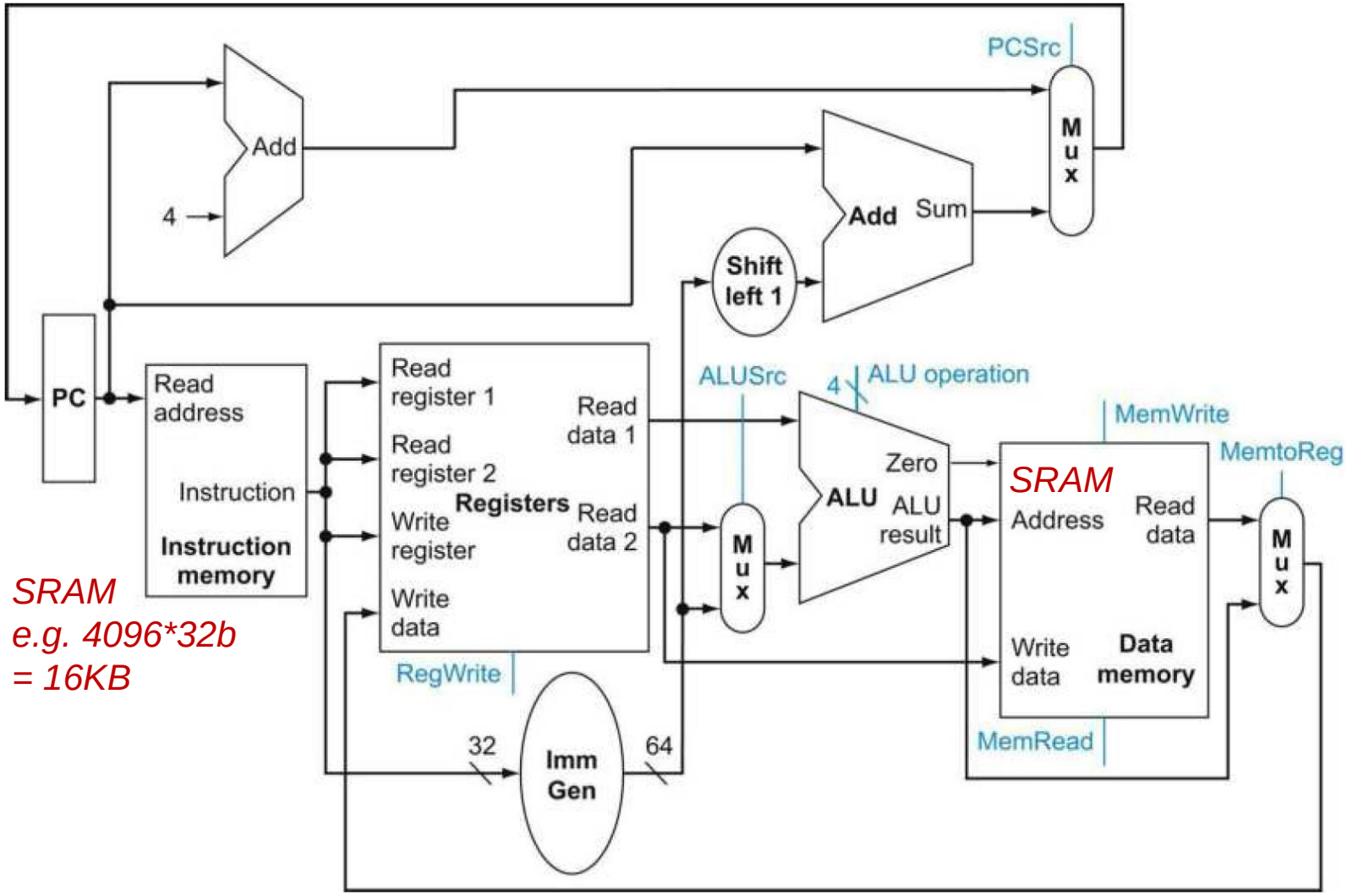


Note: only shift1, not shift2, to maintain support for compressed and variable length instructions.

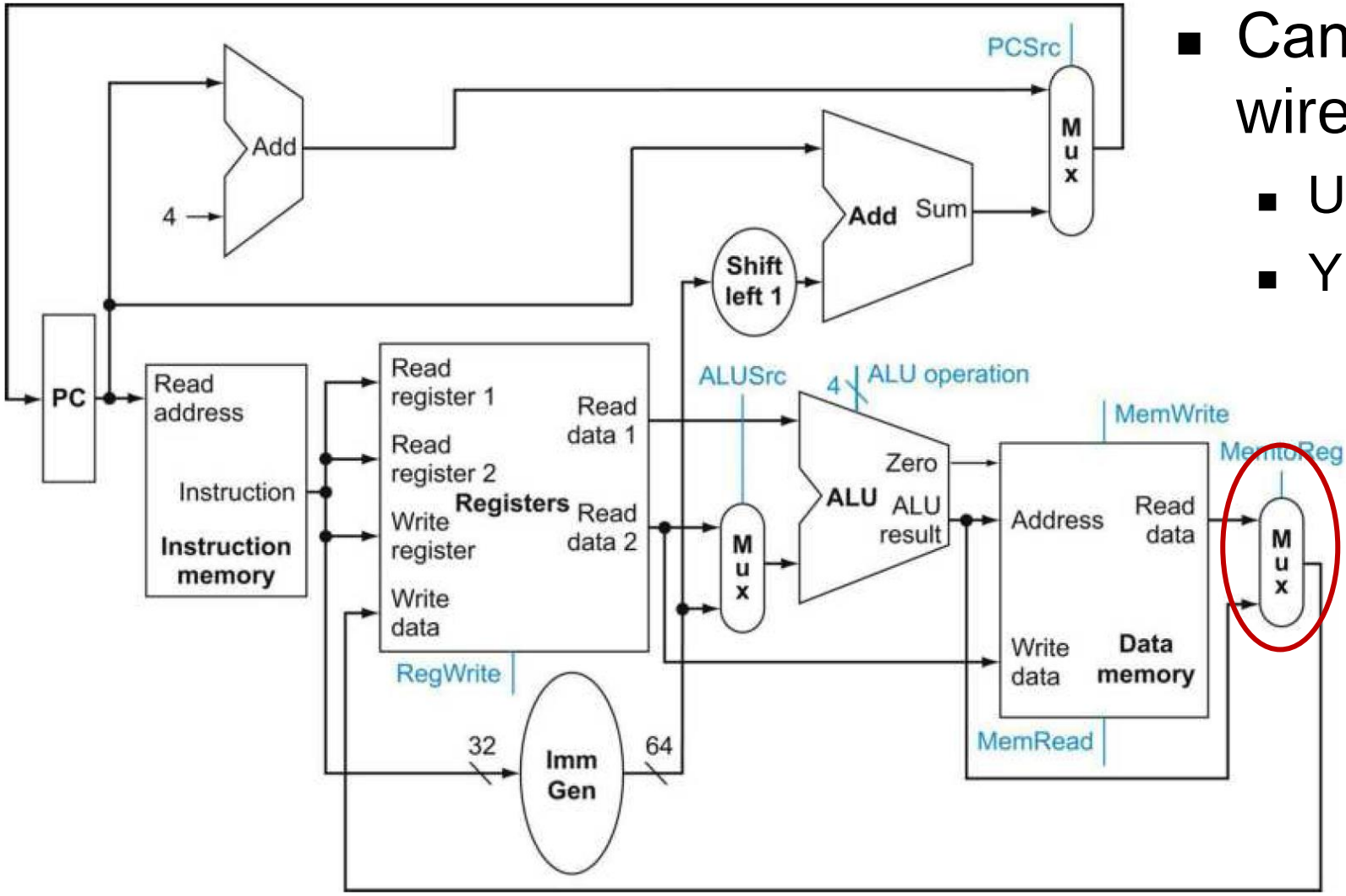
Creating a Single Datapath from the Parts

- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements
  - control to set multiplexors and resource configuration

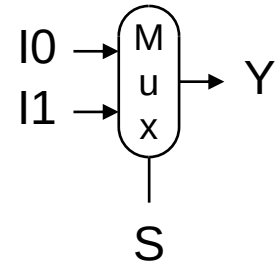
Full Datapath



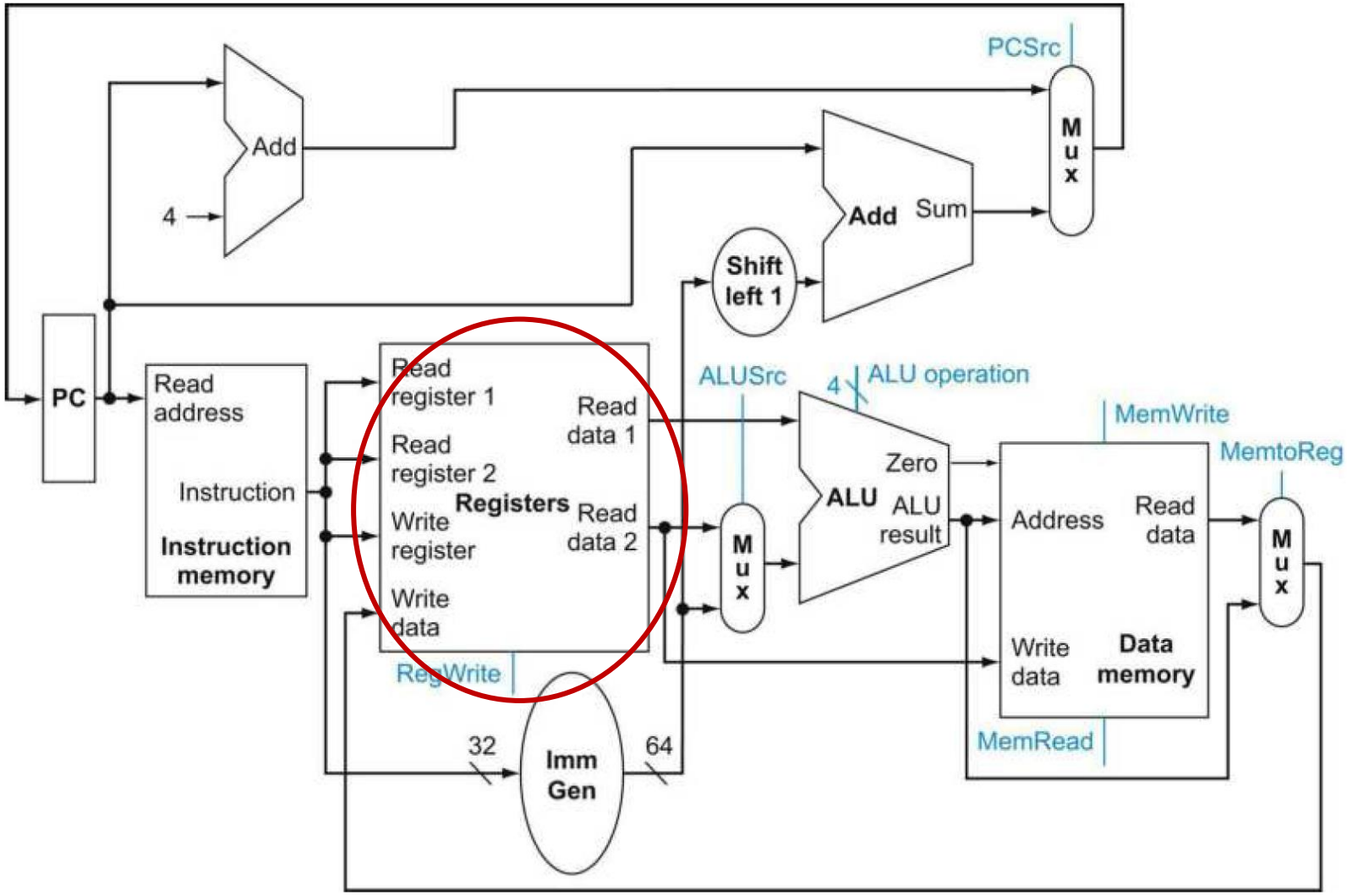
Full Datapath



- Can’t just join wires together
  - Use multiplexers
  - Y = S ? I1 : I0

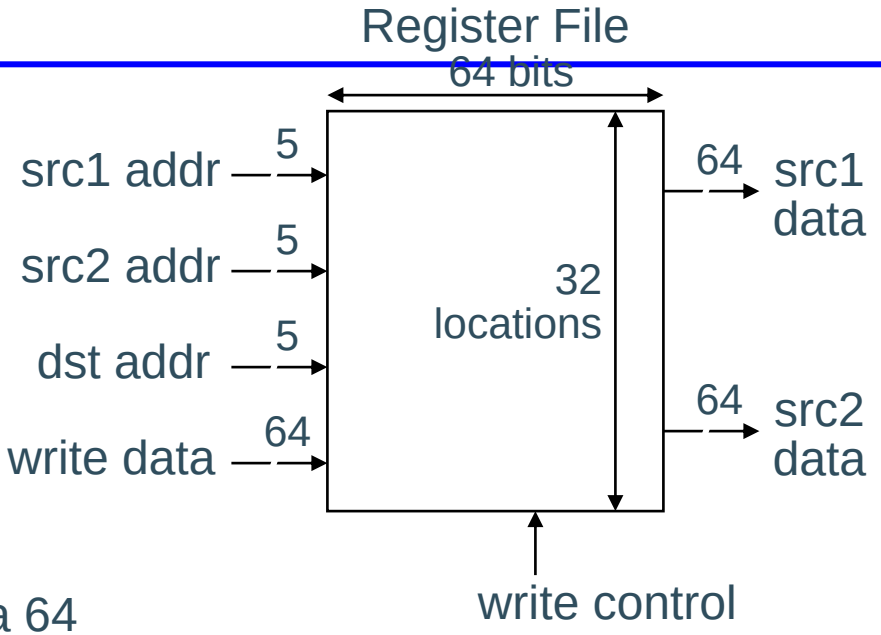


Full Datapath



Intermezzo: what is a Register File?

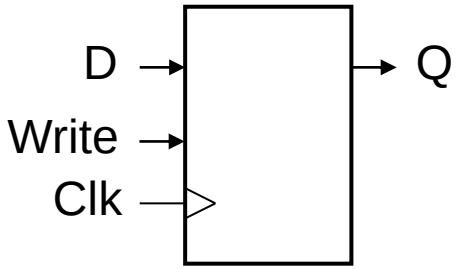
- Holds thirty-two 64-bit registers
  - Two read ports and
  - One write port

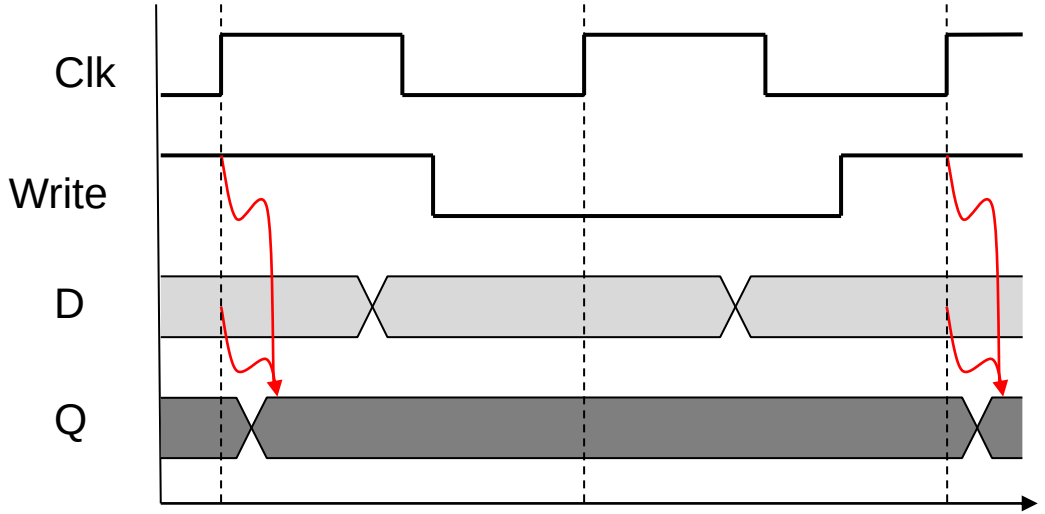


  - Registers are
    - Faster than main memory
      - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
      - Read/write port increase impacts speed quadratically
    - Improving code density
      - registers are named with fewer bits than a memory location
      - avoids load/store instructions
    - Only spill to memory for less frequently used variables

Sequential Elements

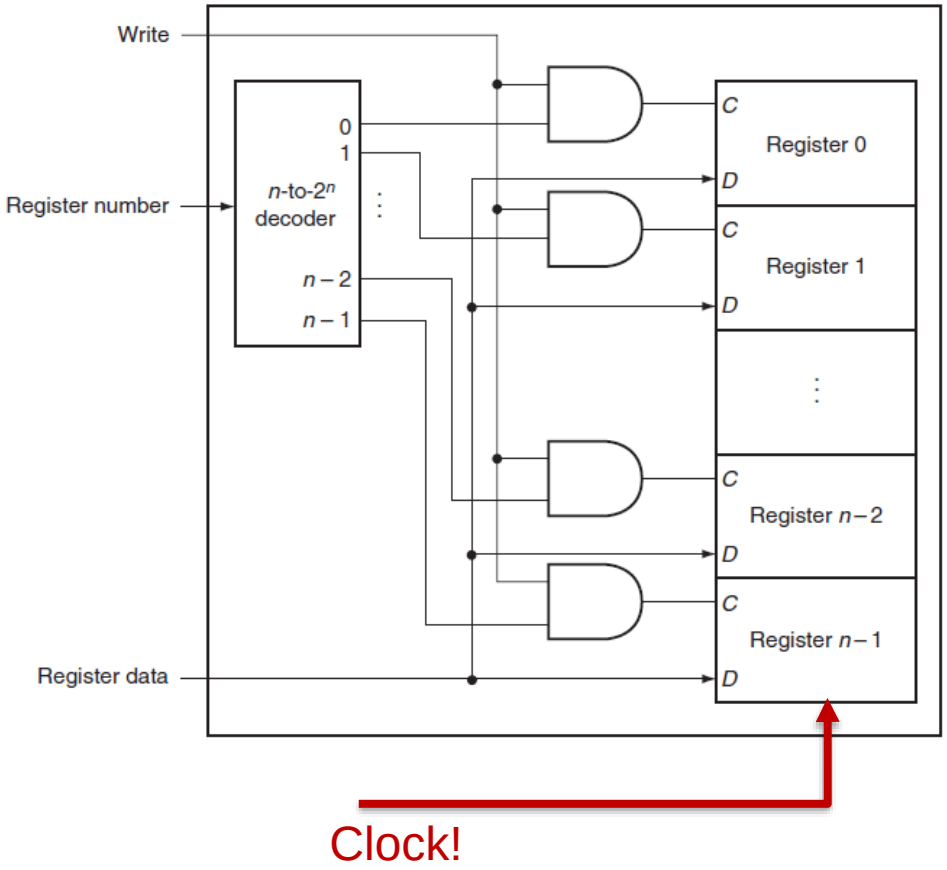
- Register (=flipflop) with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

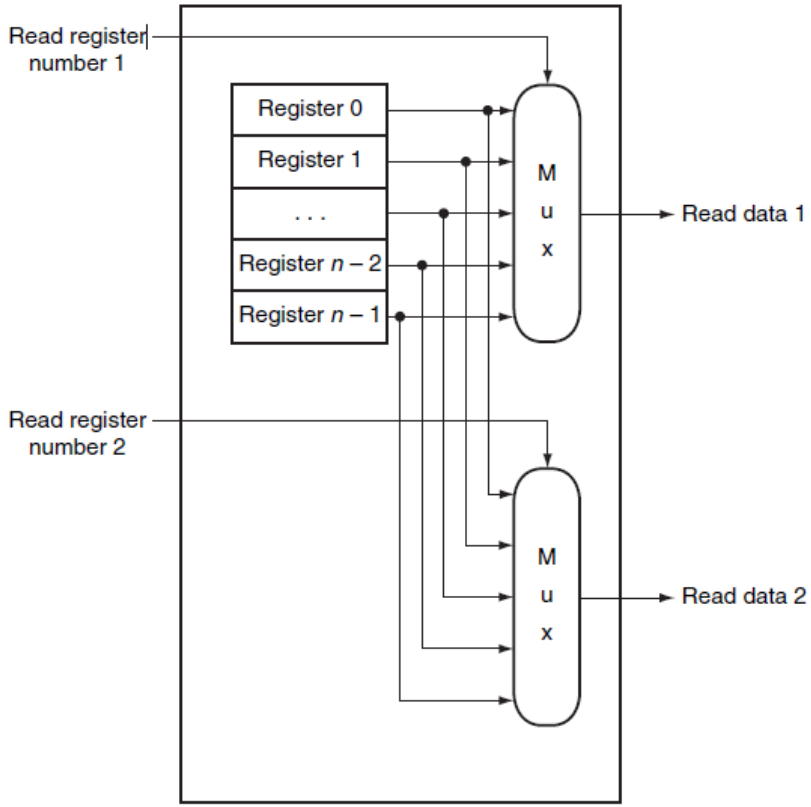




Register file internals

- Flipflops!, that can selectively be read (unclocked) and written (clocked!)



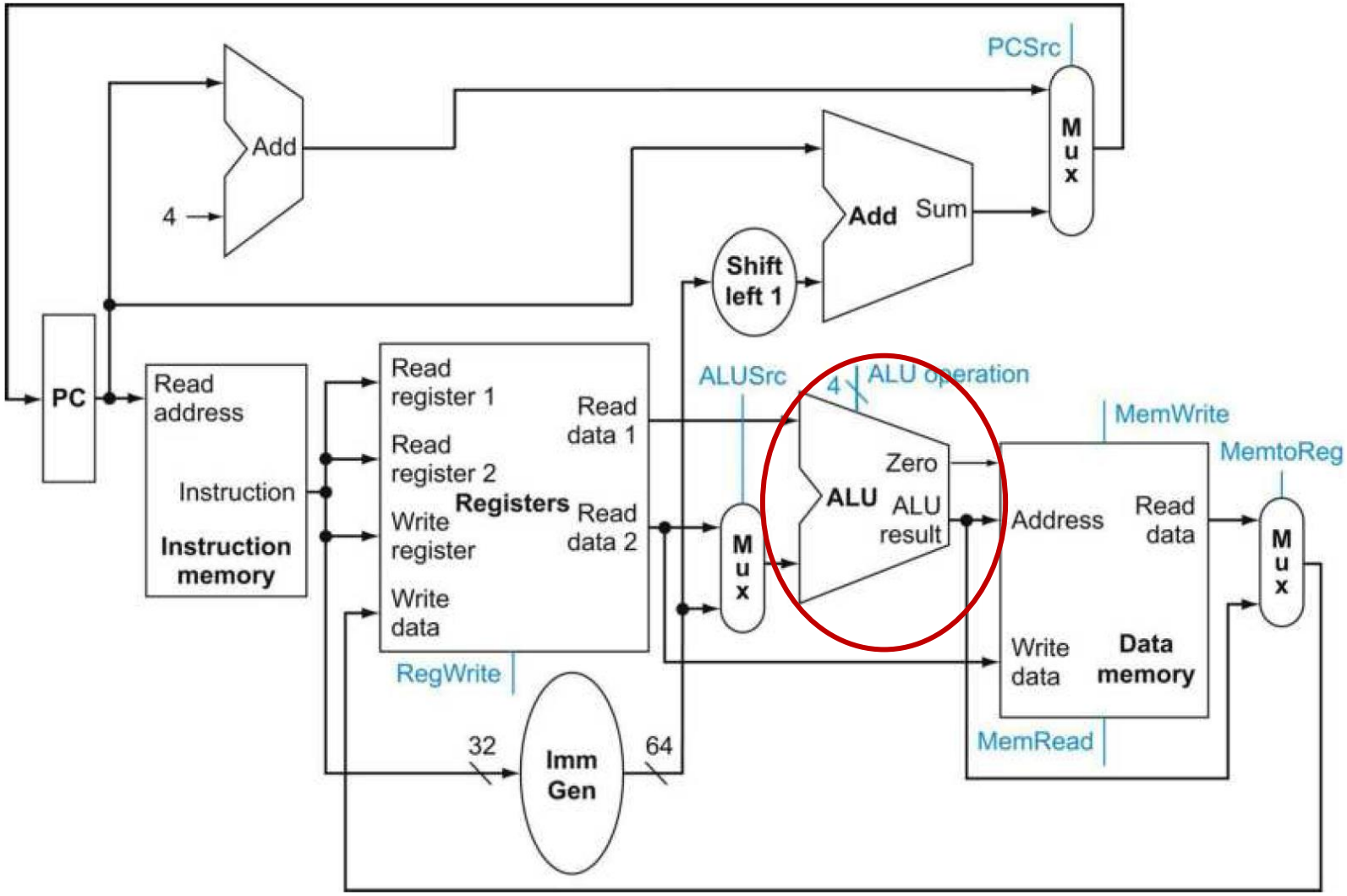


- Good for small memories <> SRAM for big memories!

Remember: RISC-V Register Convention (Lecture 2)

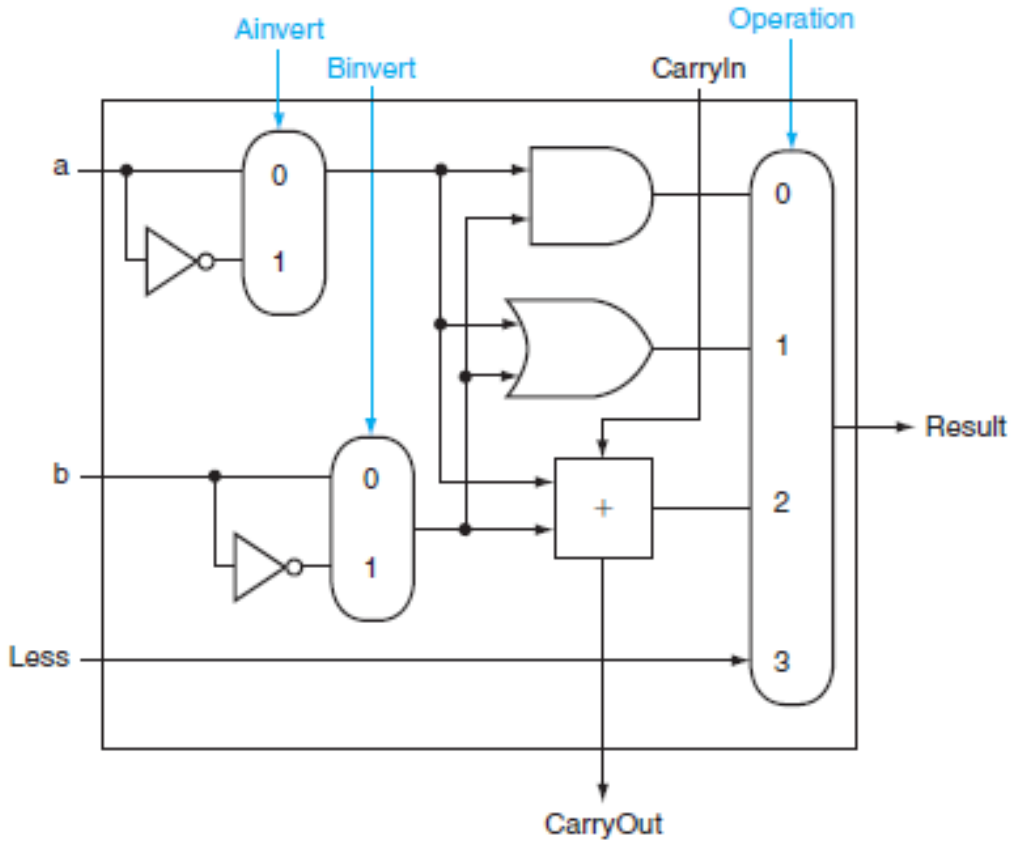
Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

Full Datapath



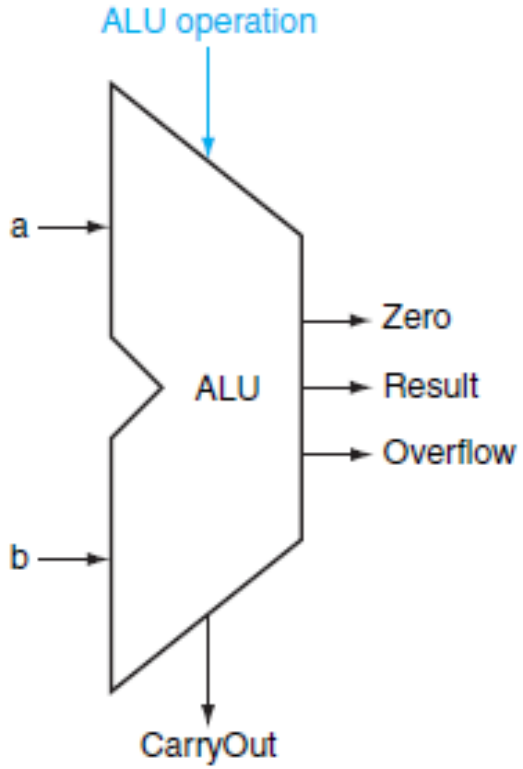
Intermezzo: What is an ALU?

- performs the arithmetic operations like ADD and SUB or logical operations like AND and OR, or comparisons like BEQ
- 1-bit ALU

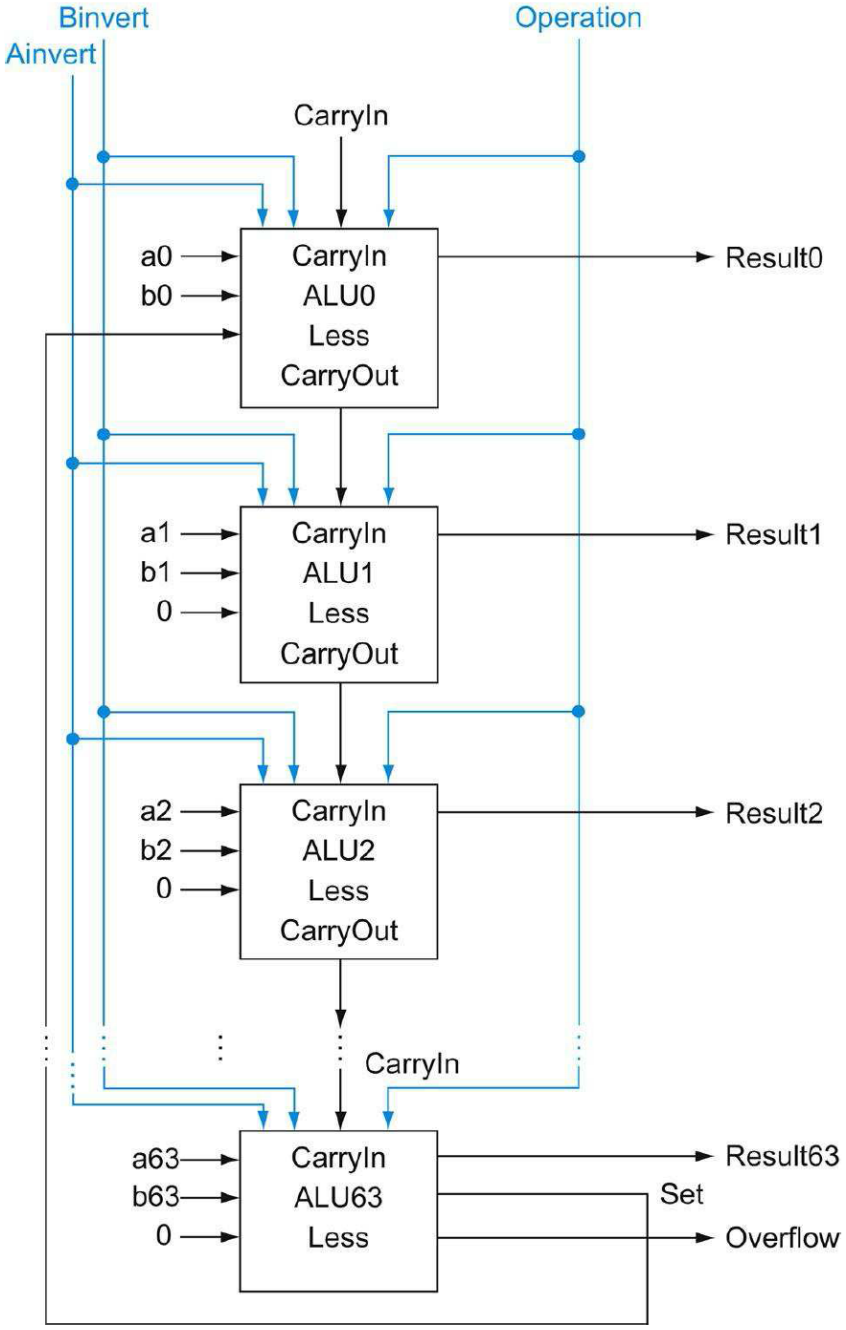


Intermezzo: What is an ALU

- Glue-ing it together for 64-bit ALU



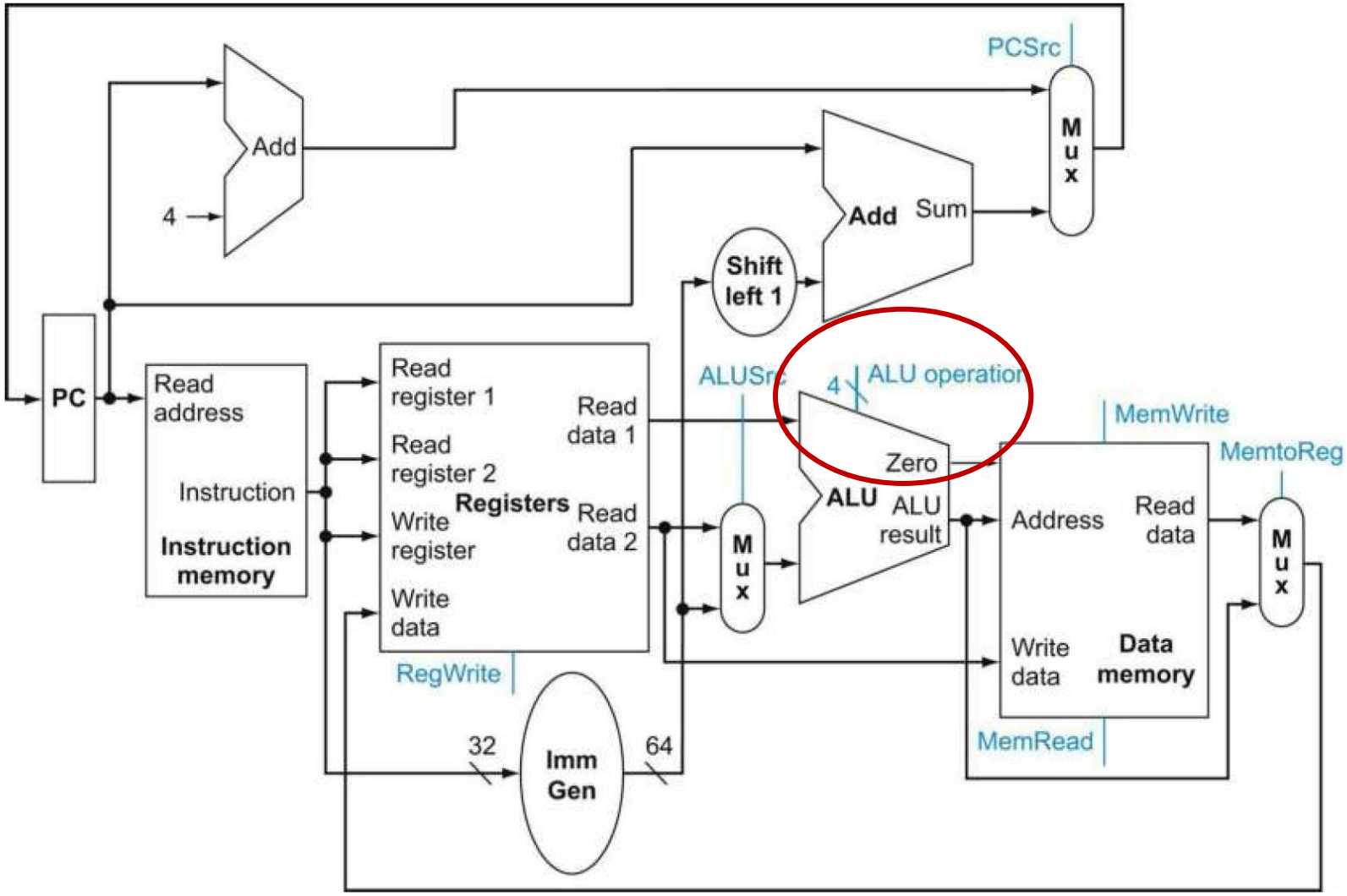
=



- Zero detect (`beq, bne`)

Full Datapath

How does instruction goes to register file and immediate?



ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

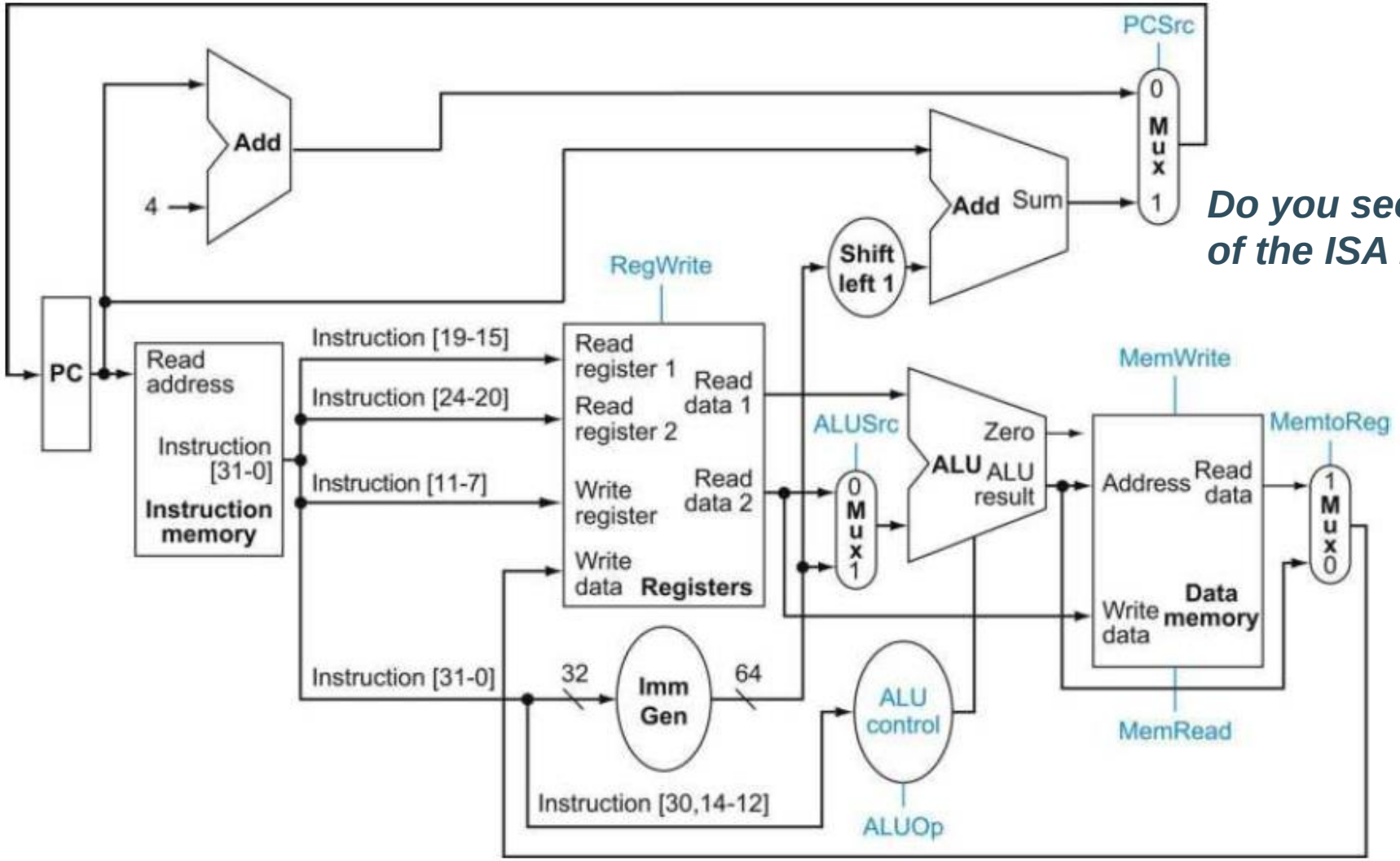
ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	XXXXXXXXXXX	add	0010
sd	00	store register	XXXXXXXXXXX	add	0010
beq	01	branch on equal	XXXXXXXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

Full Datapath

Who drives the blue wires?



Do you see why regularity of the ISA mattered?

Adding control

- Read/write addresses derived from instruction

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode

read, except for I-type

always read

write for R&I-type

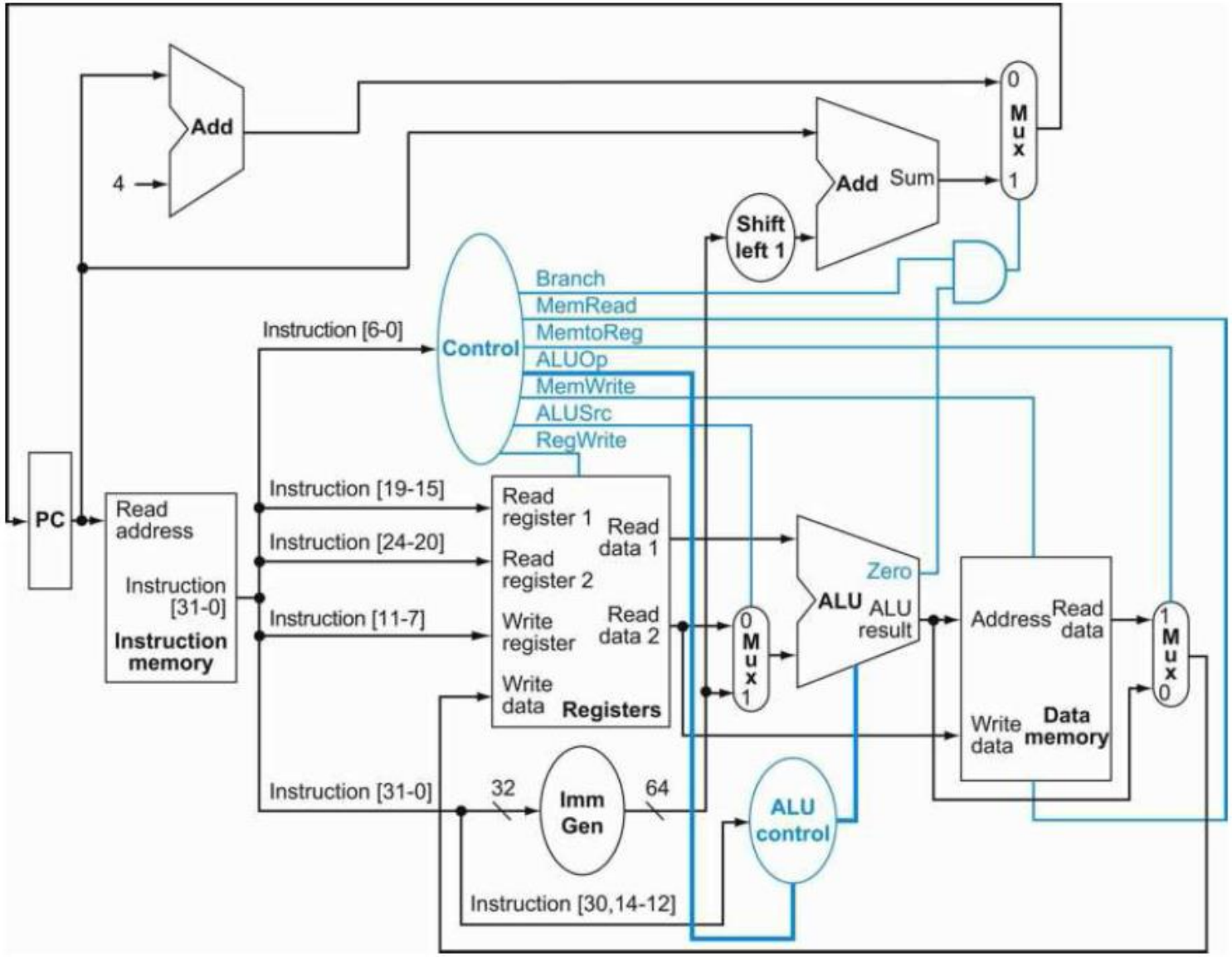
Adding control: The Control Unit

- Control signals derived from instruction

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode

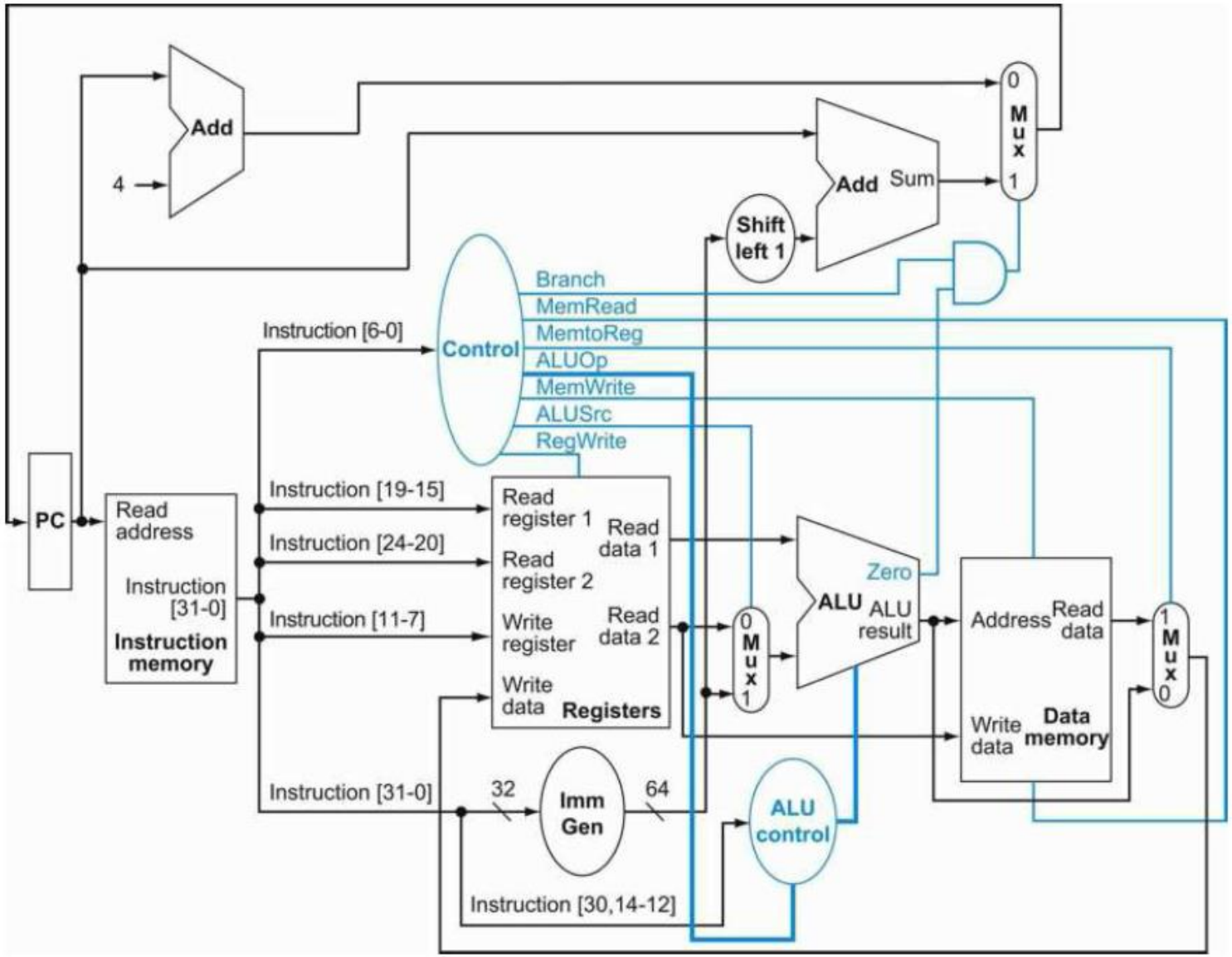
sign-extend funct7 for R-type	read, except for I-type	always read	funct	write for R&I-type	opcode

Datapath With Control

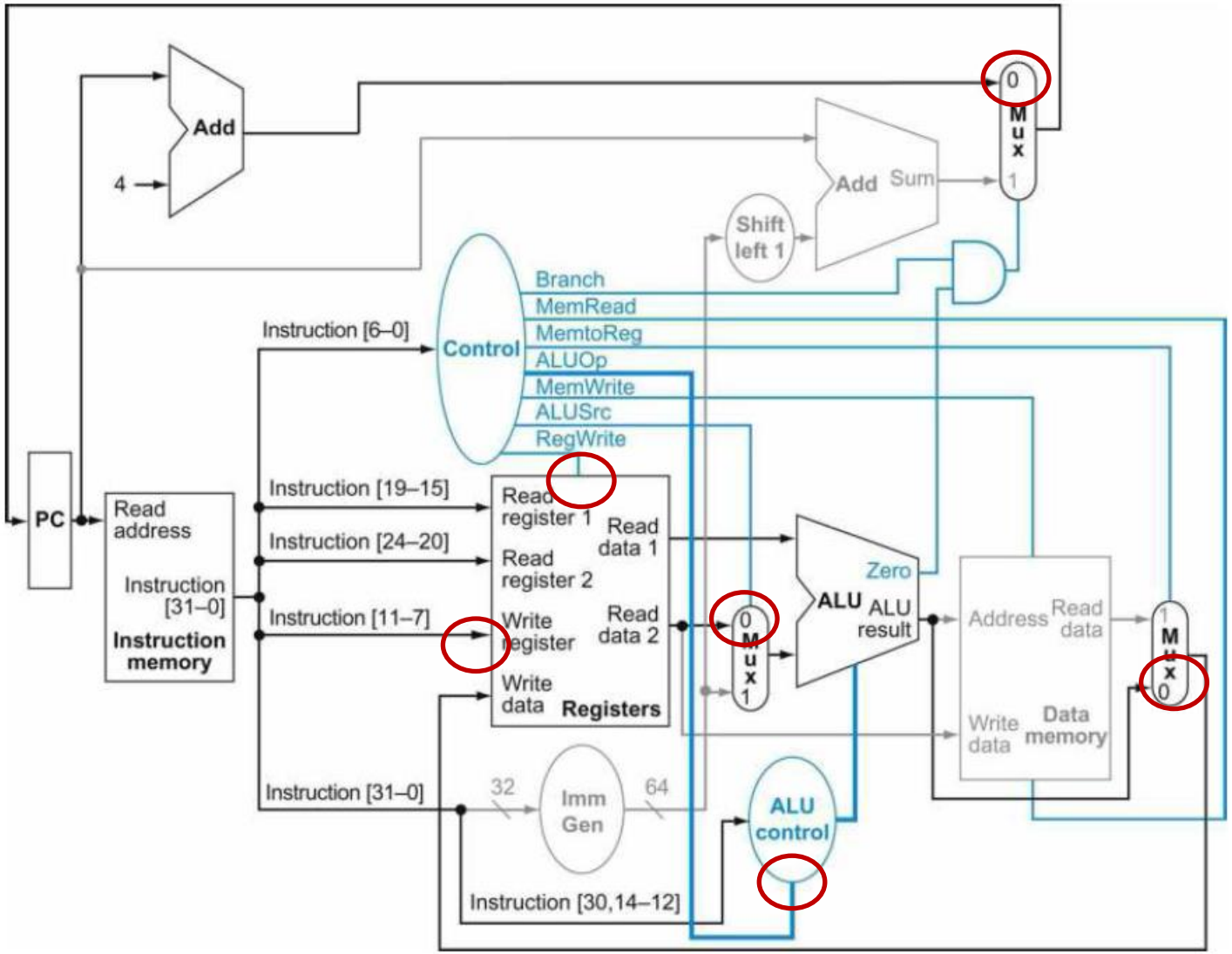


Exercise! R-Type Instruction?

mark the connections in the datapath that are active & show the state of the control lines.

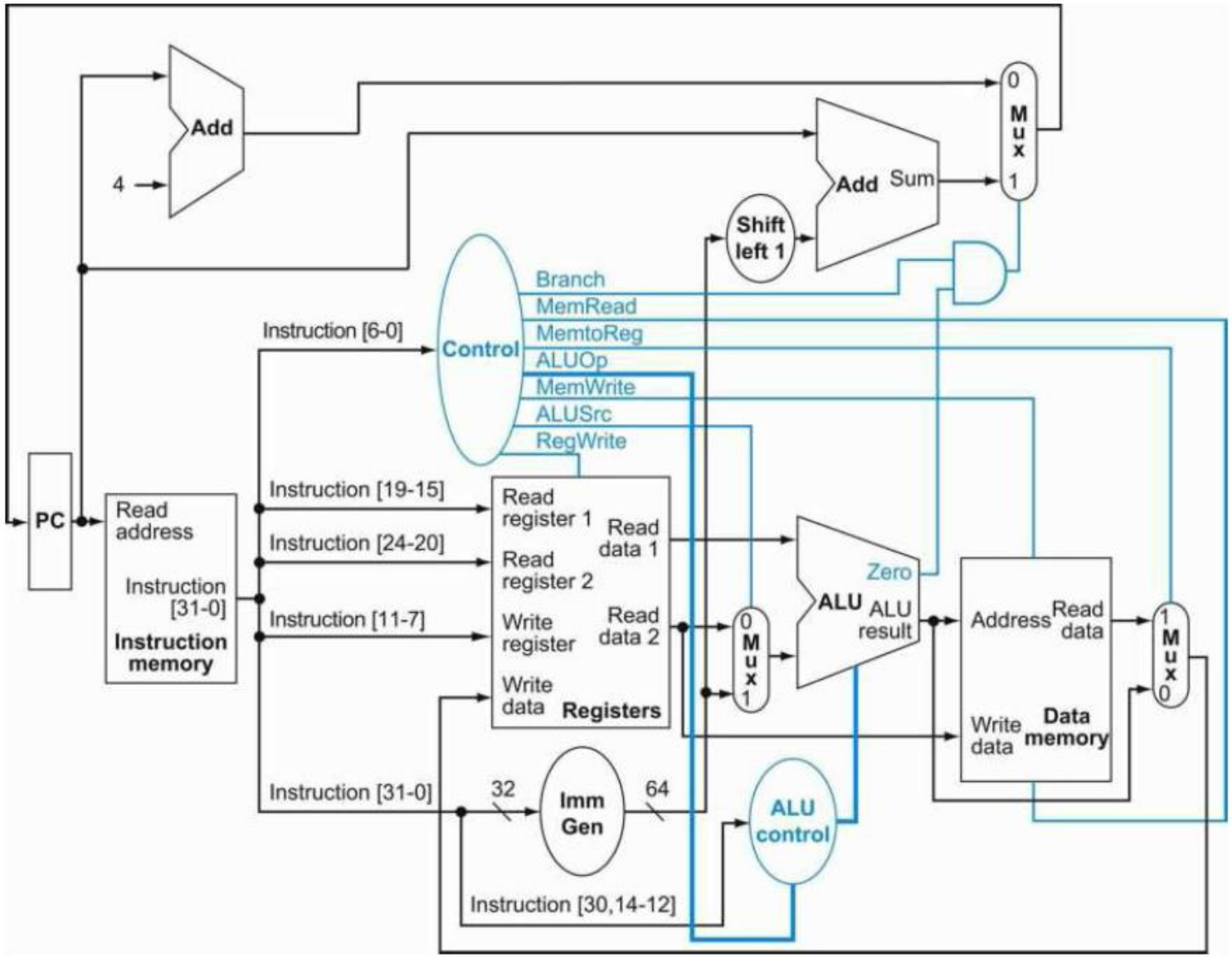


R-Type Instruction

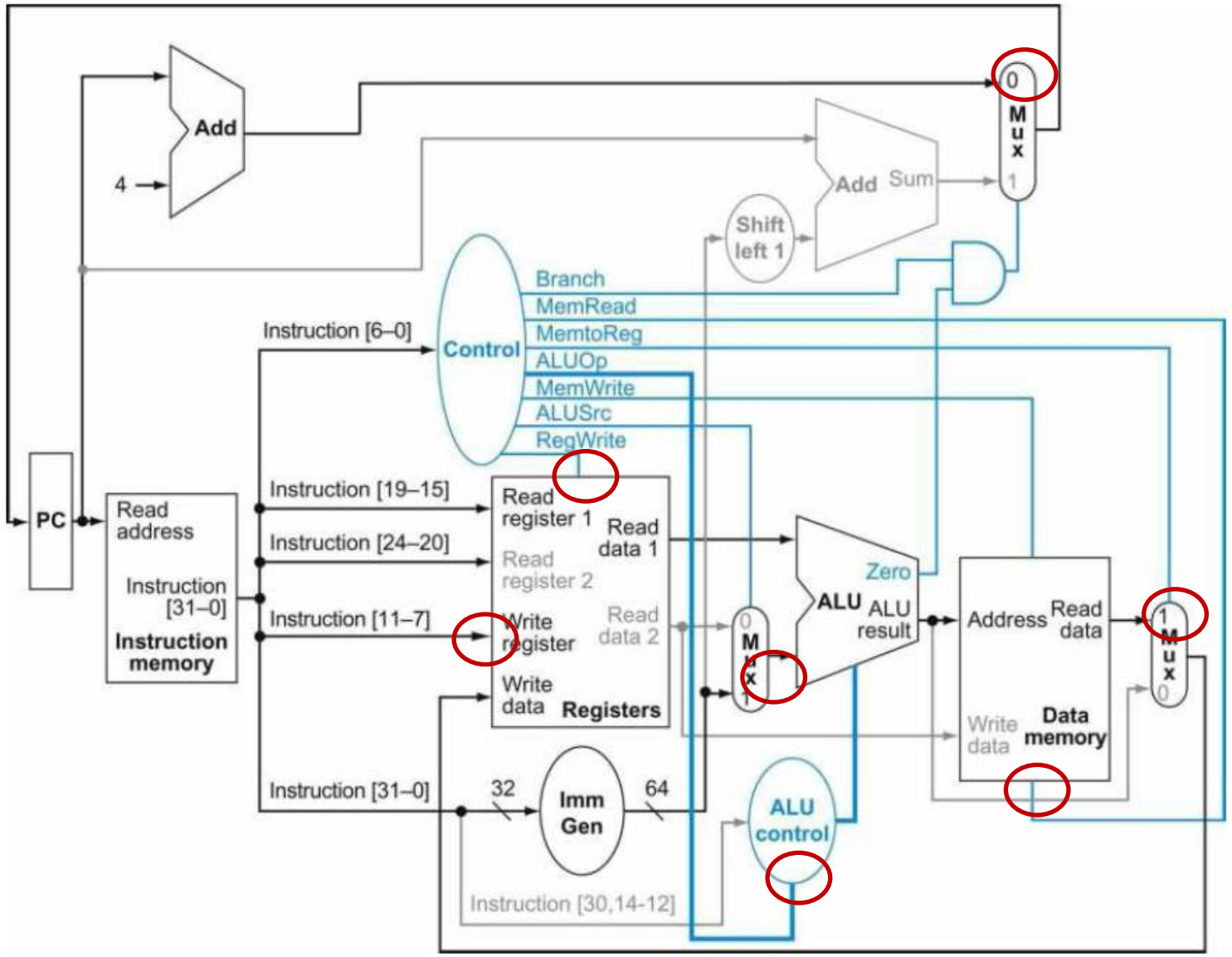


Exercise! Load Instruction?

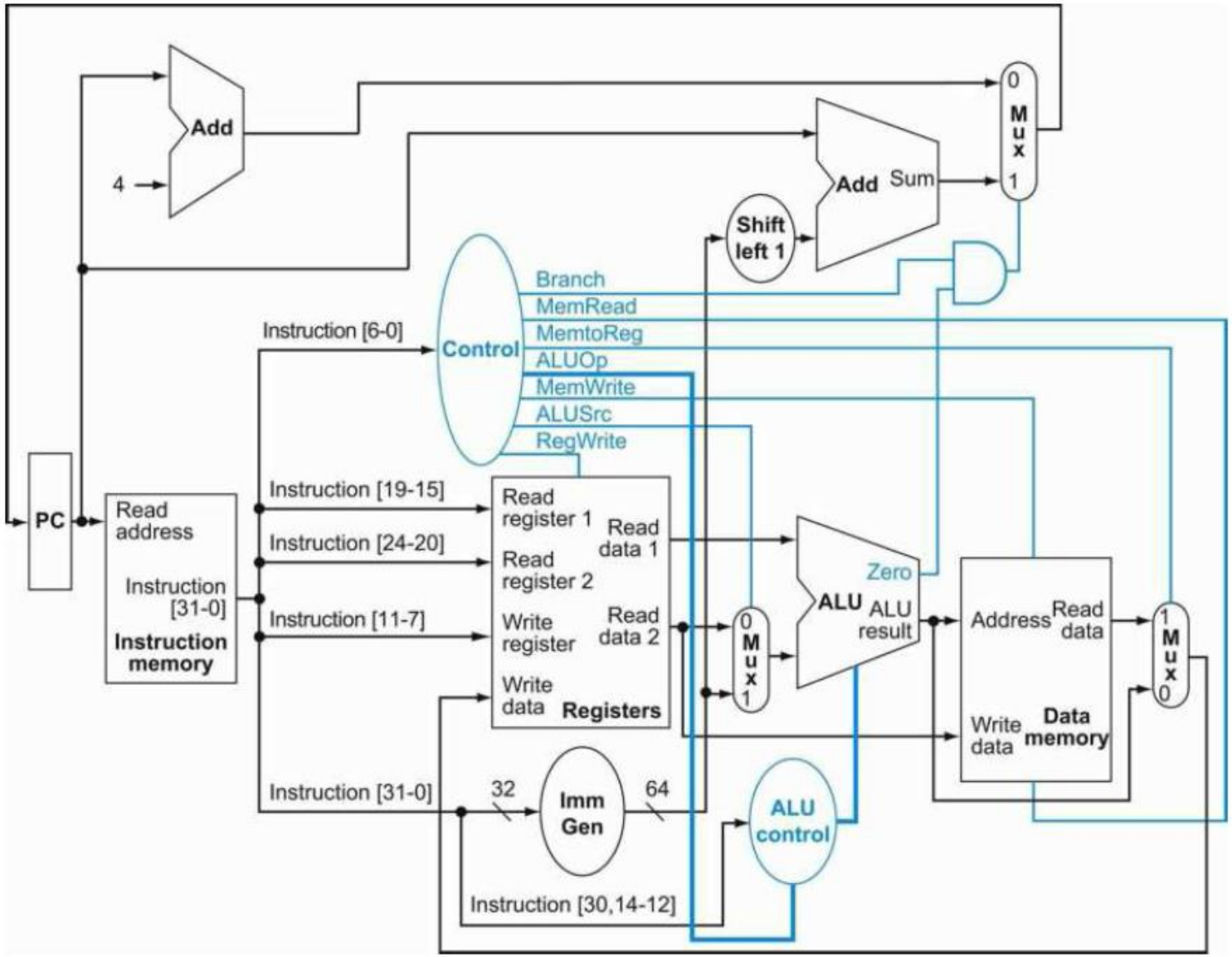
mark the connections in the datapath that are active & show the state of the control lines.



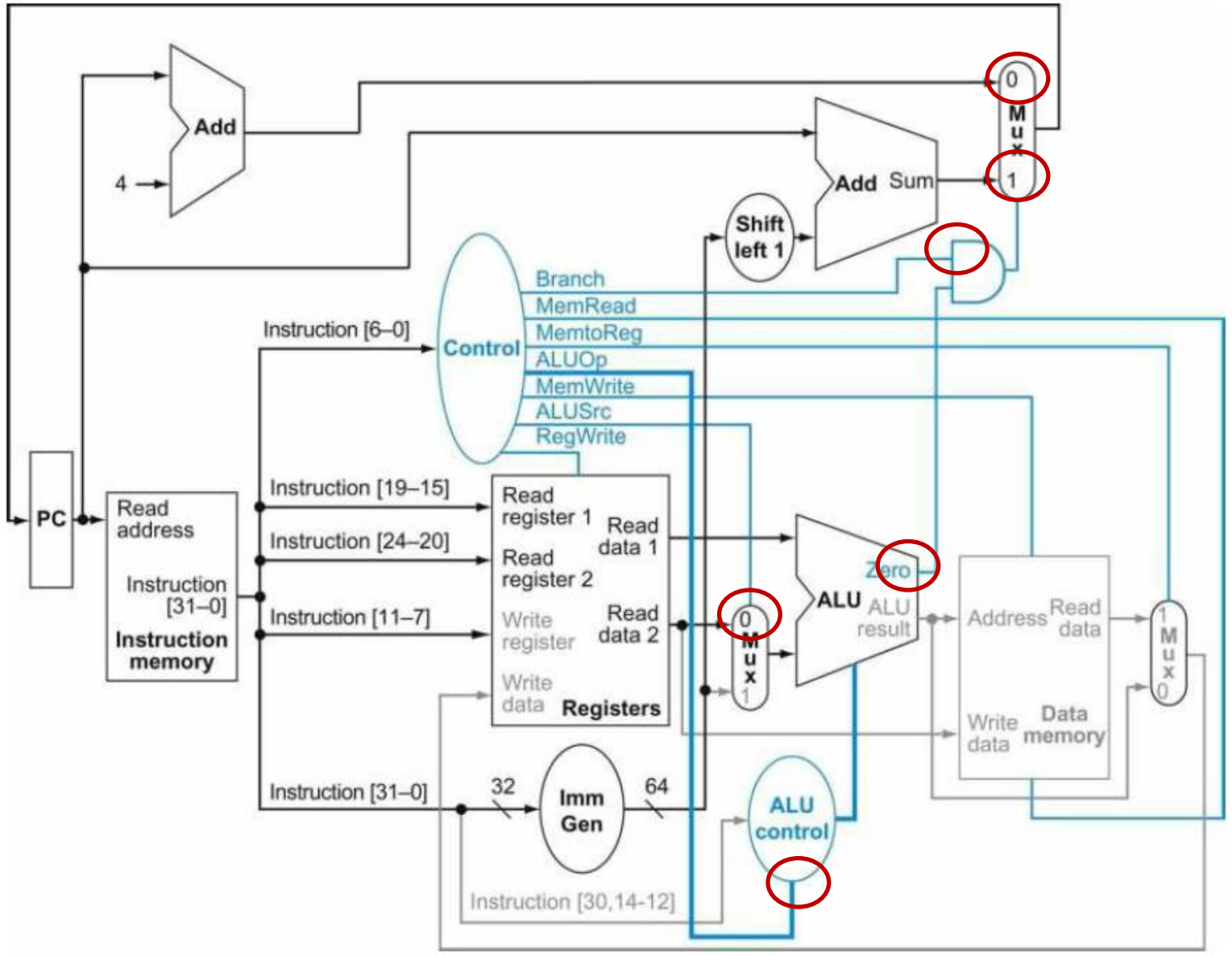
Load Instruction



Exercise! Branch-on-Equal Instruction?



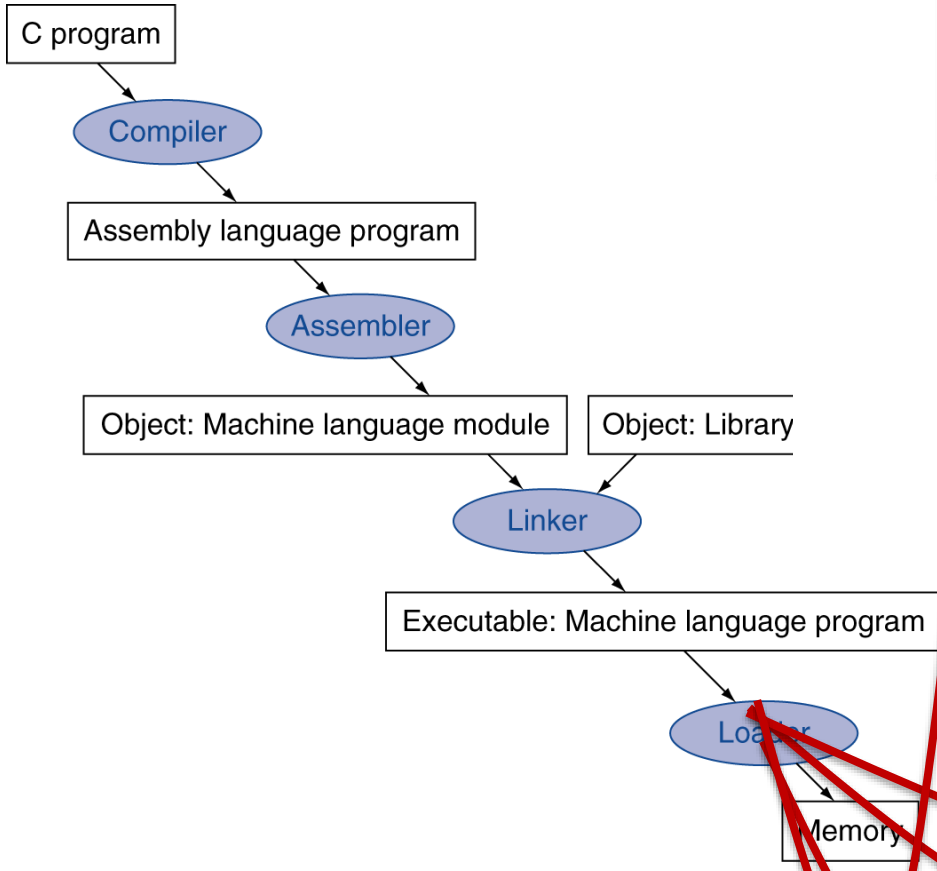
Branch-on-Equal Instruction

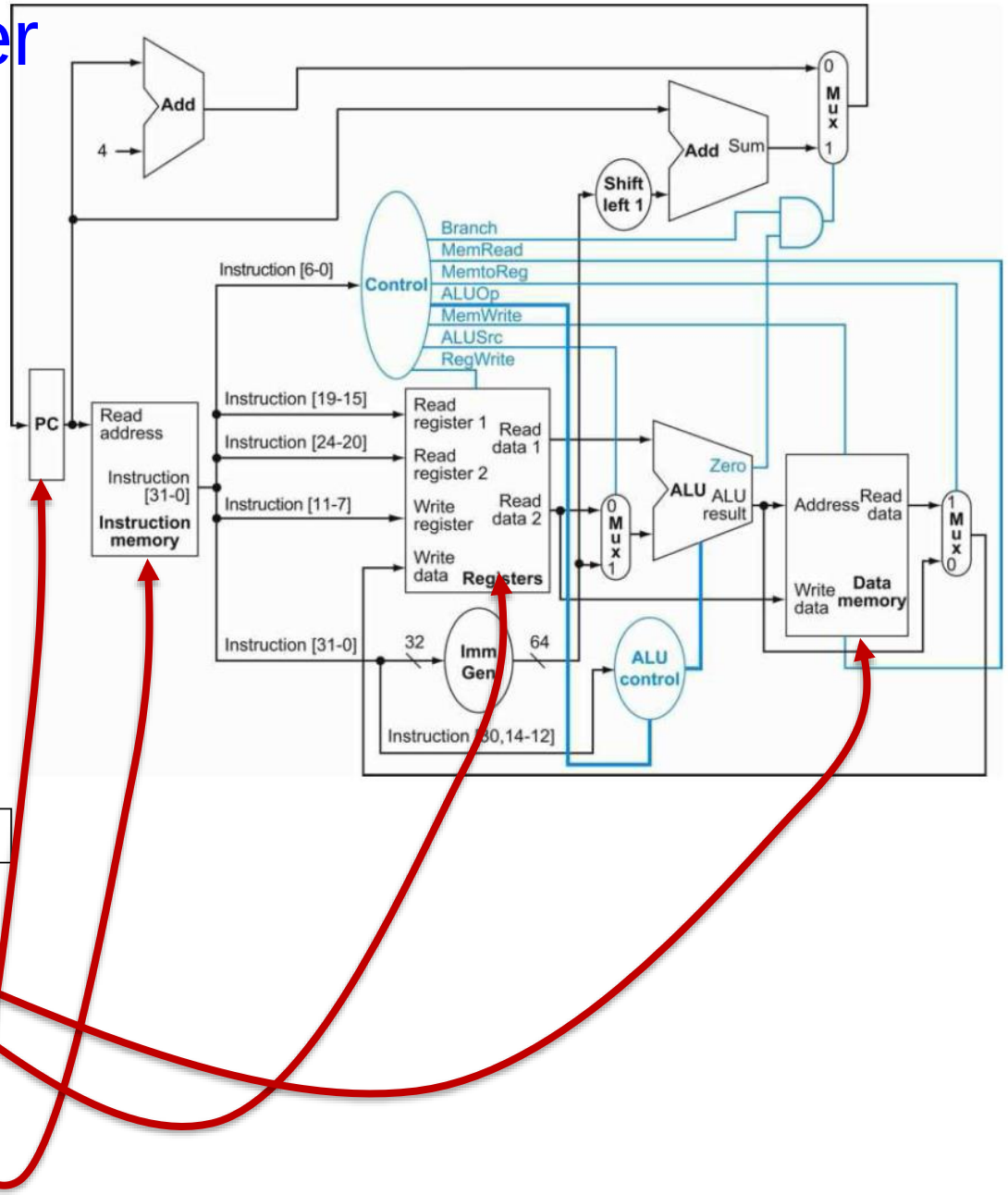


Lecture overview

- Lecture 4: Single cycle processor
  - Recap and goal
  - Building up the processor
  - **Translating and starting a program**
  - Pipelining basics

Bringing it all together





Loading a Program

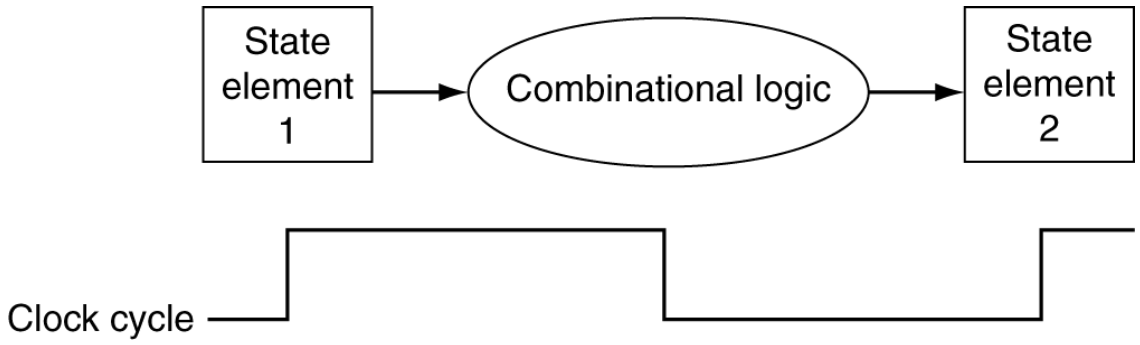
- Load from image file on disk into working memory
  1. Read header to determine imem and data sizes
  2. Create address space
  3. Copy imem and initialized data into memory
  4. Set up arguments for main on the stack
  5. Initialize registers (including $sp, etc.)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main

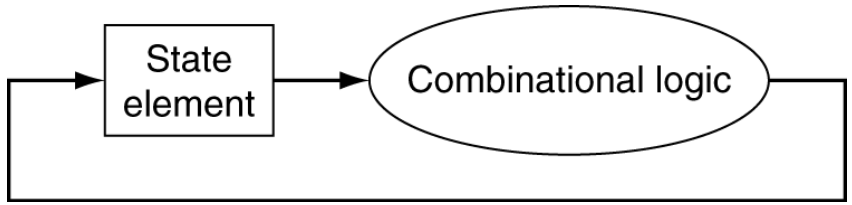
Lecture overview

- Lecture 4: Single cycle processor
  - Recap and goal
  - Building up the processor
  - Translating and starting a program
  - **Pipelining basics**

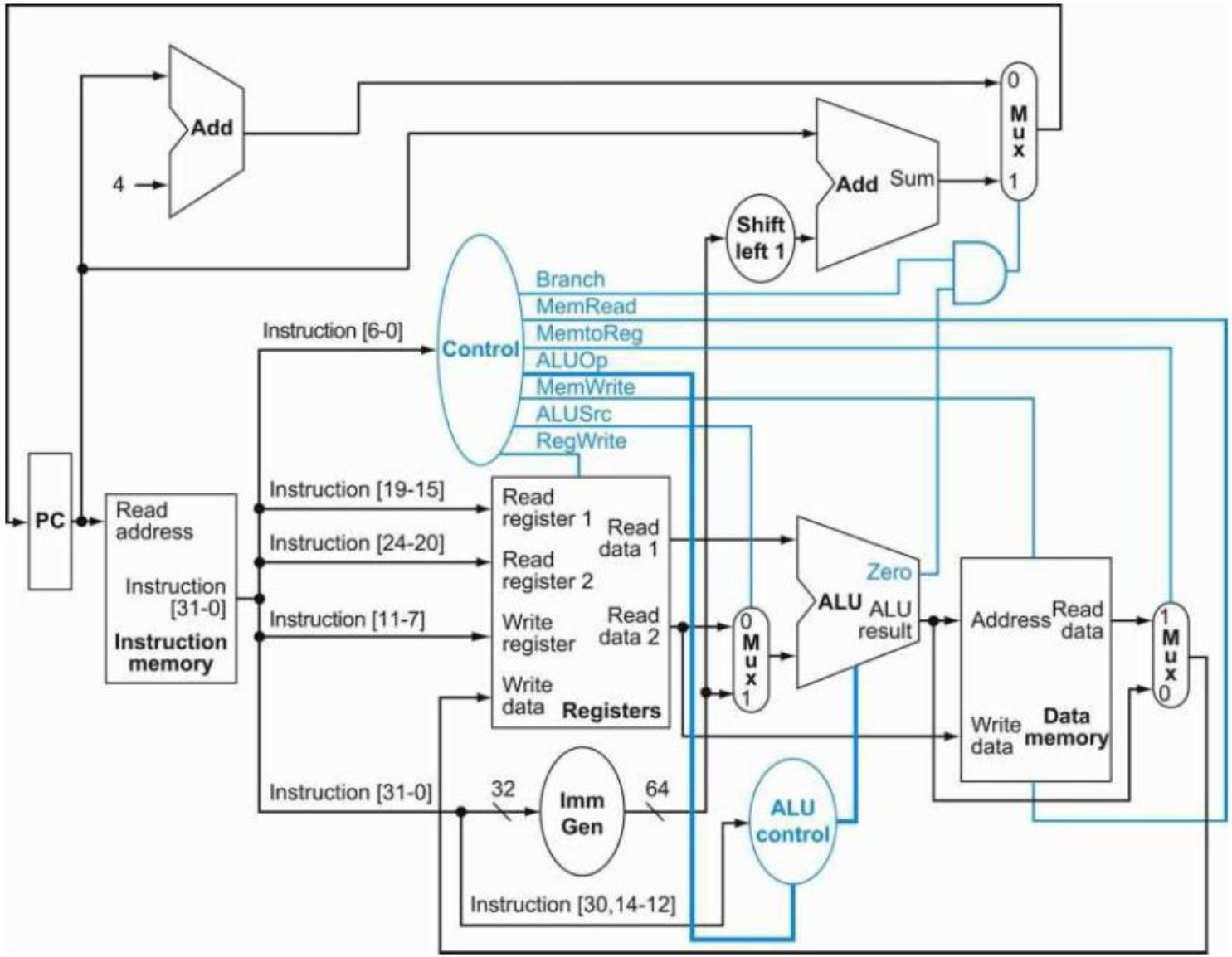
Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay (= critical path) determines clock period
  - Not feasible to vary period for different instructions (fixed clock)





Critical path?



Instruction Critical Paths

- Single cycle processor = clock is set by slowest path...
- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

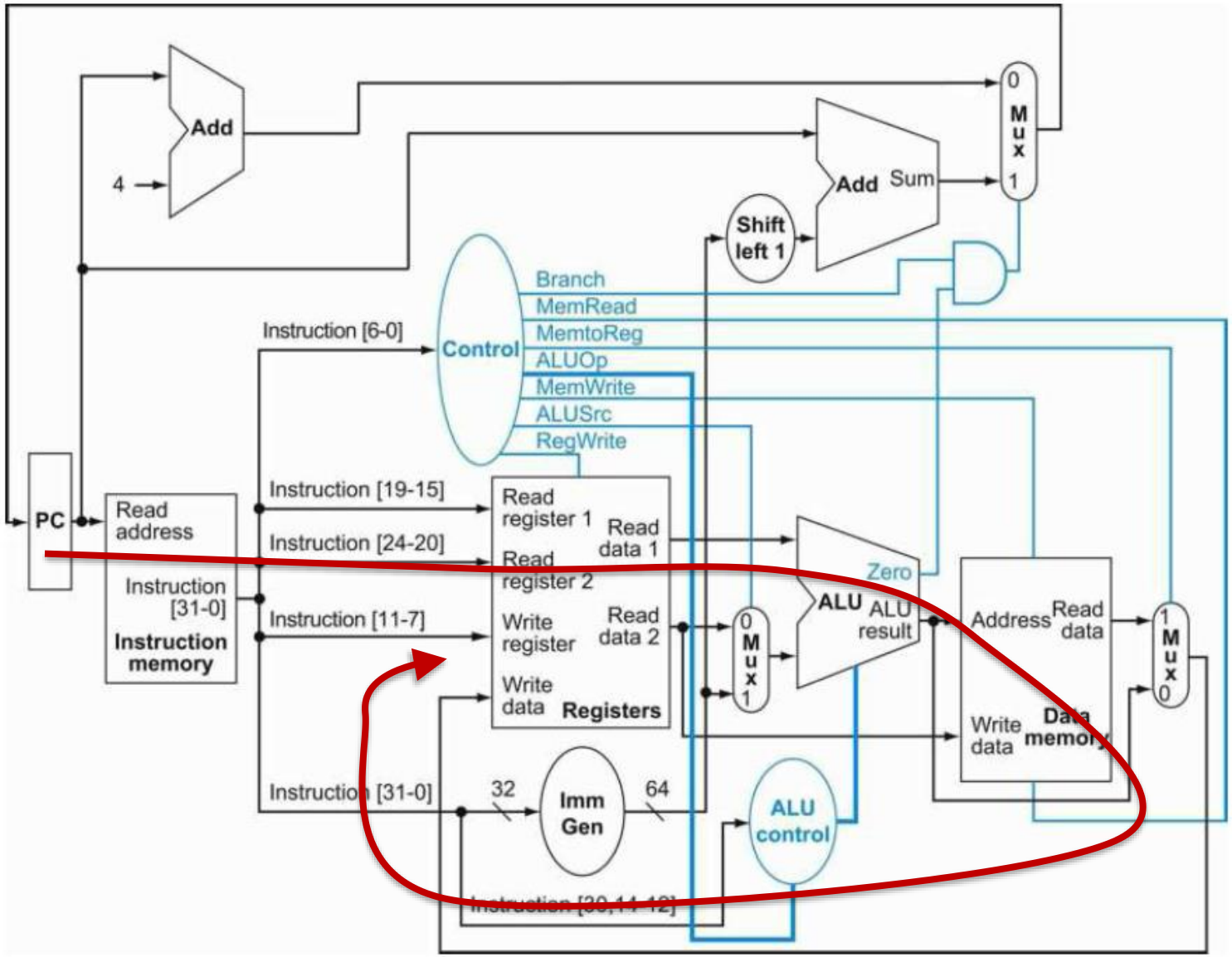
Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
beq	200	100	200			500

Critical path?

Remember:

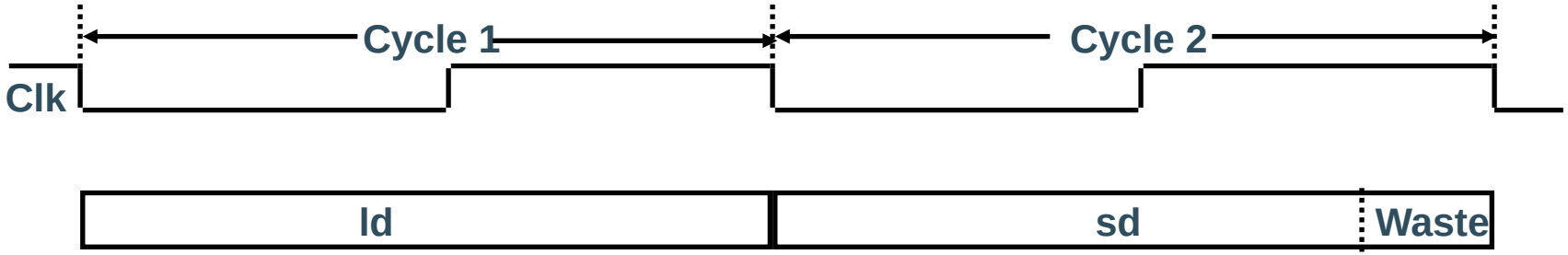
Memory read = unclocked

Memory write = clocked!!!



Single Cycle vs Pipelining

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply

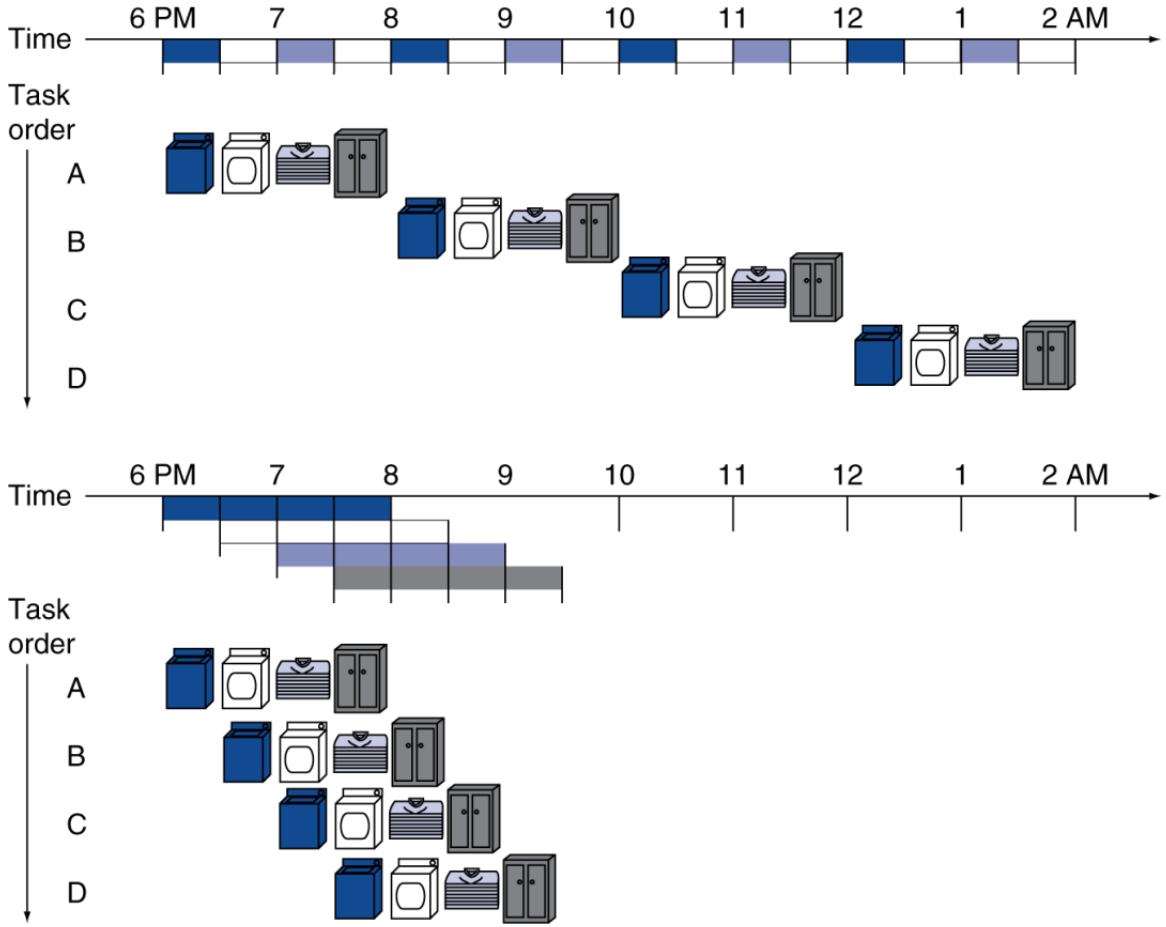


- Solution: Start fetching and executing the next instruction before the current one has completed
  - = Pipelining – in all modern processors
  - performance: seconds/task = IC * CPI * CC

➔ decrease CC, while keeping IC and CPI fixed…

Pipelining Analogy

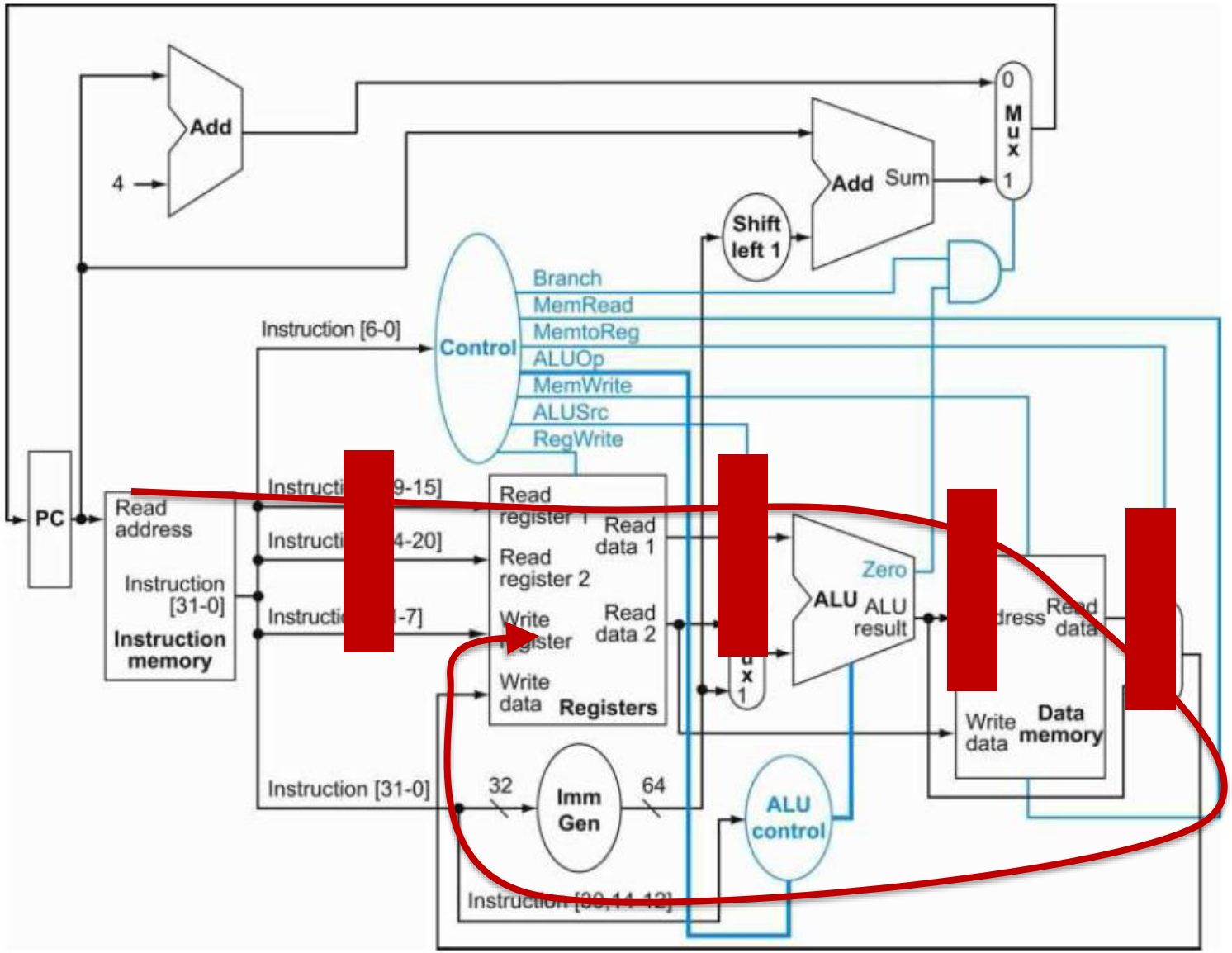
- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup
    = 8/3.5 = 2.3

- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

Break up the critical path!



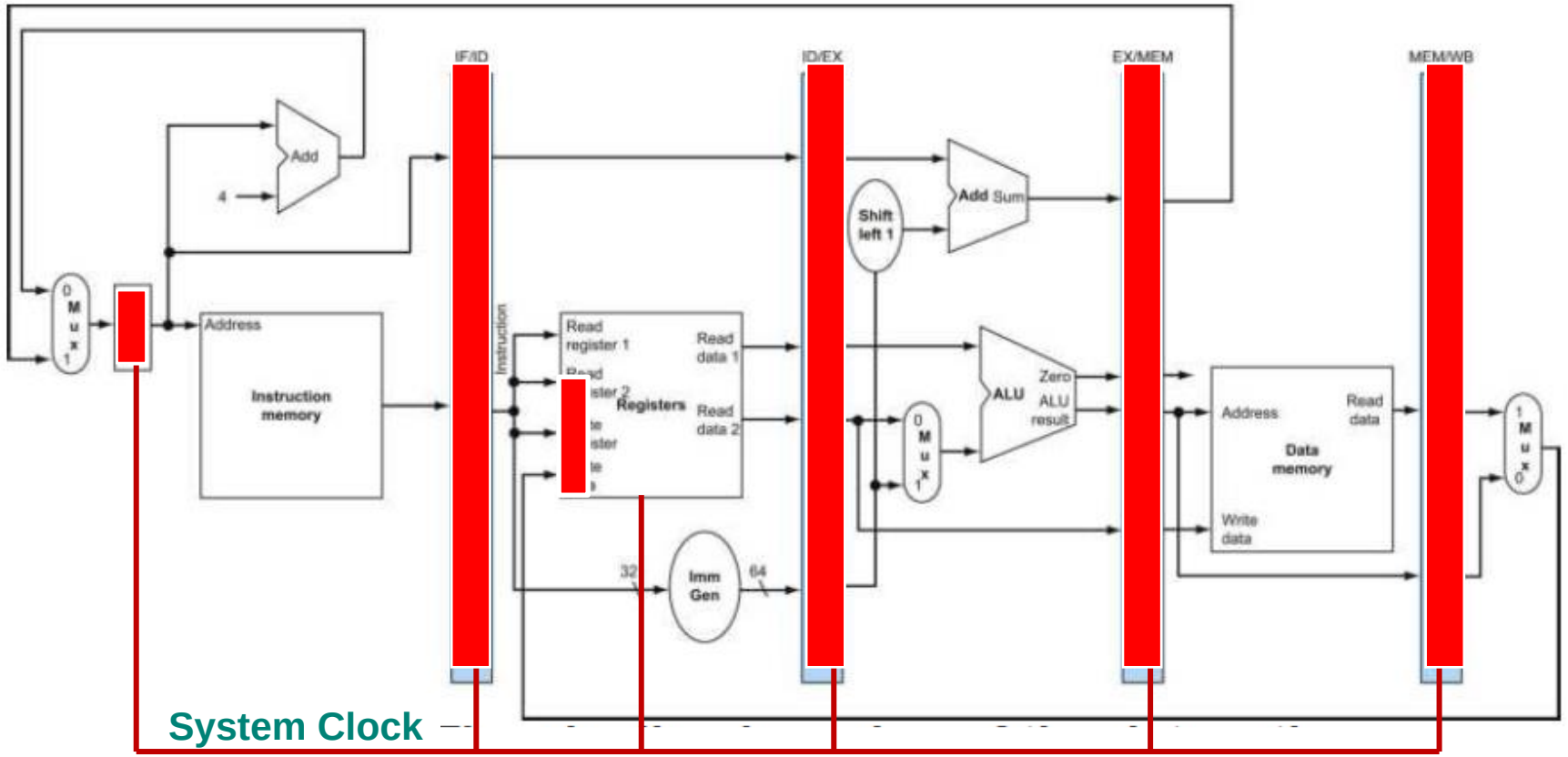
RISC-V Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Processor Pipeline Datapath Additions/Modifications

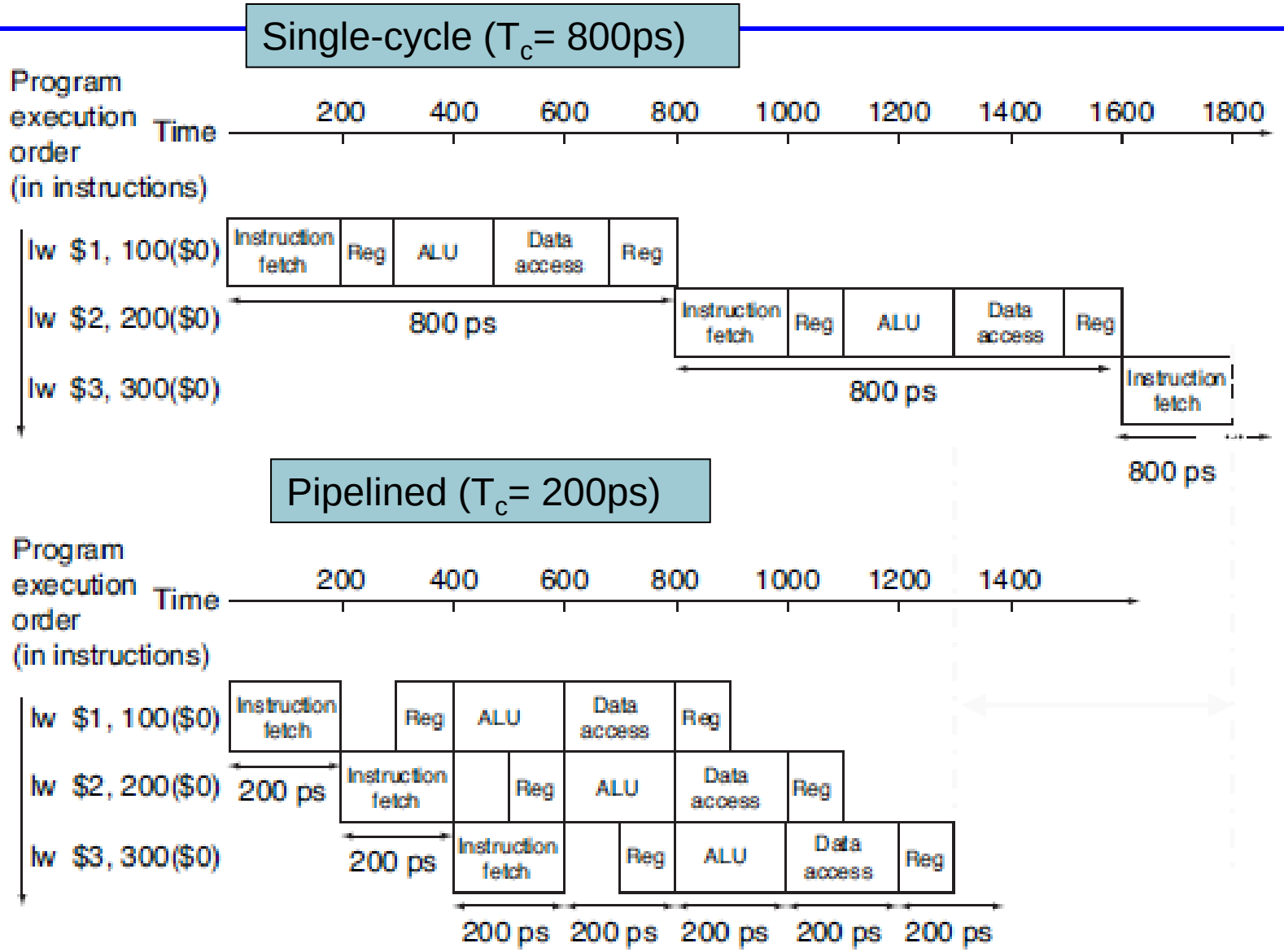
IF:IFetch **ID:Dec** **EX:Execute** **MEM: MemAccess** **WB: WriteBack**

End of WB stage?



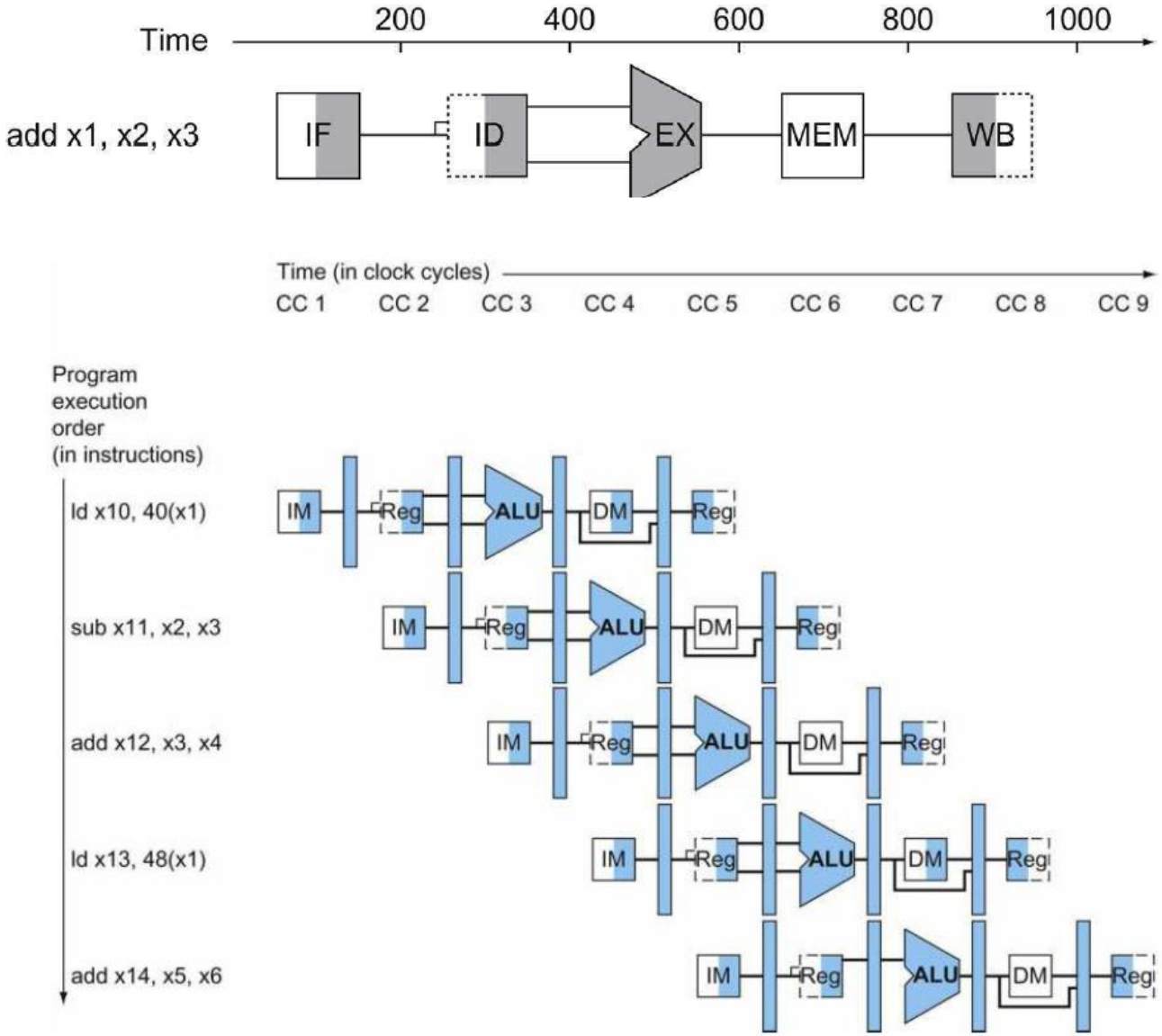
- State registers between each pipeline stage to isolate them

Single Cycle versus Pipeline



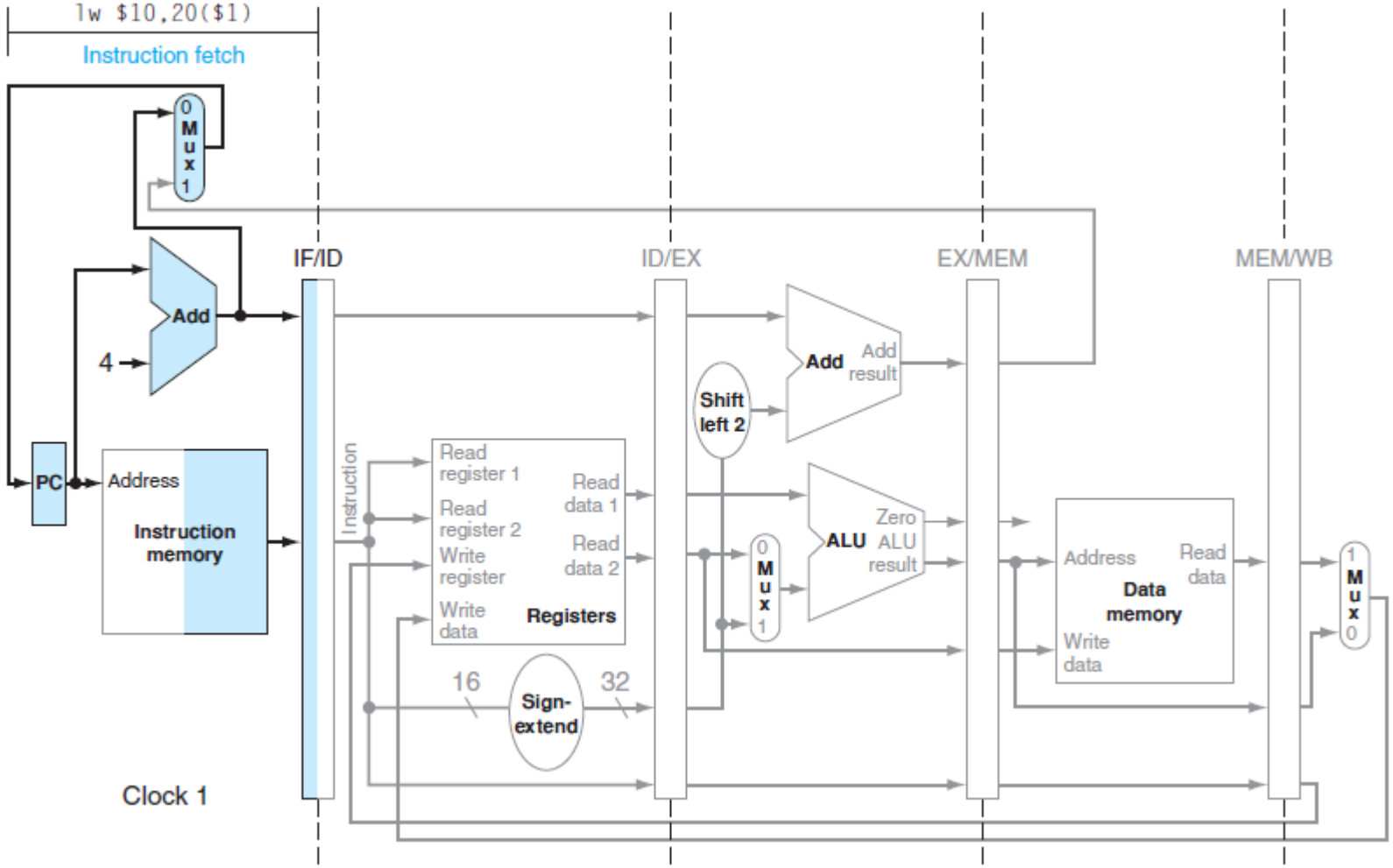
Lots of wasted time!
(e.g. in sw)

Single cycle vs pipeline (2)



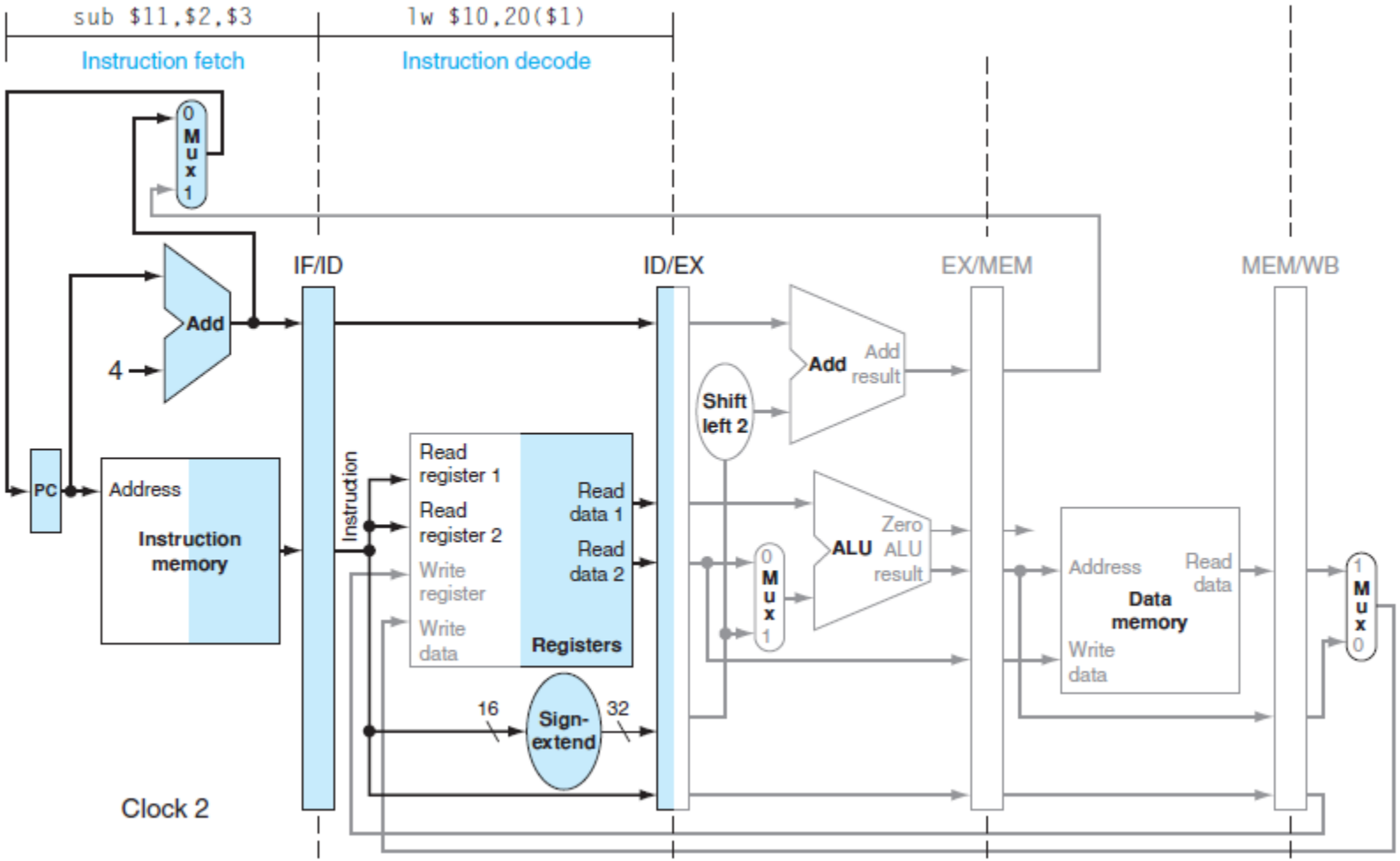
Pipelined example - Clk1

```
lw   $10, 20($1)
sub  $11, $2, $3
```



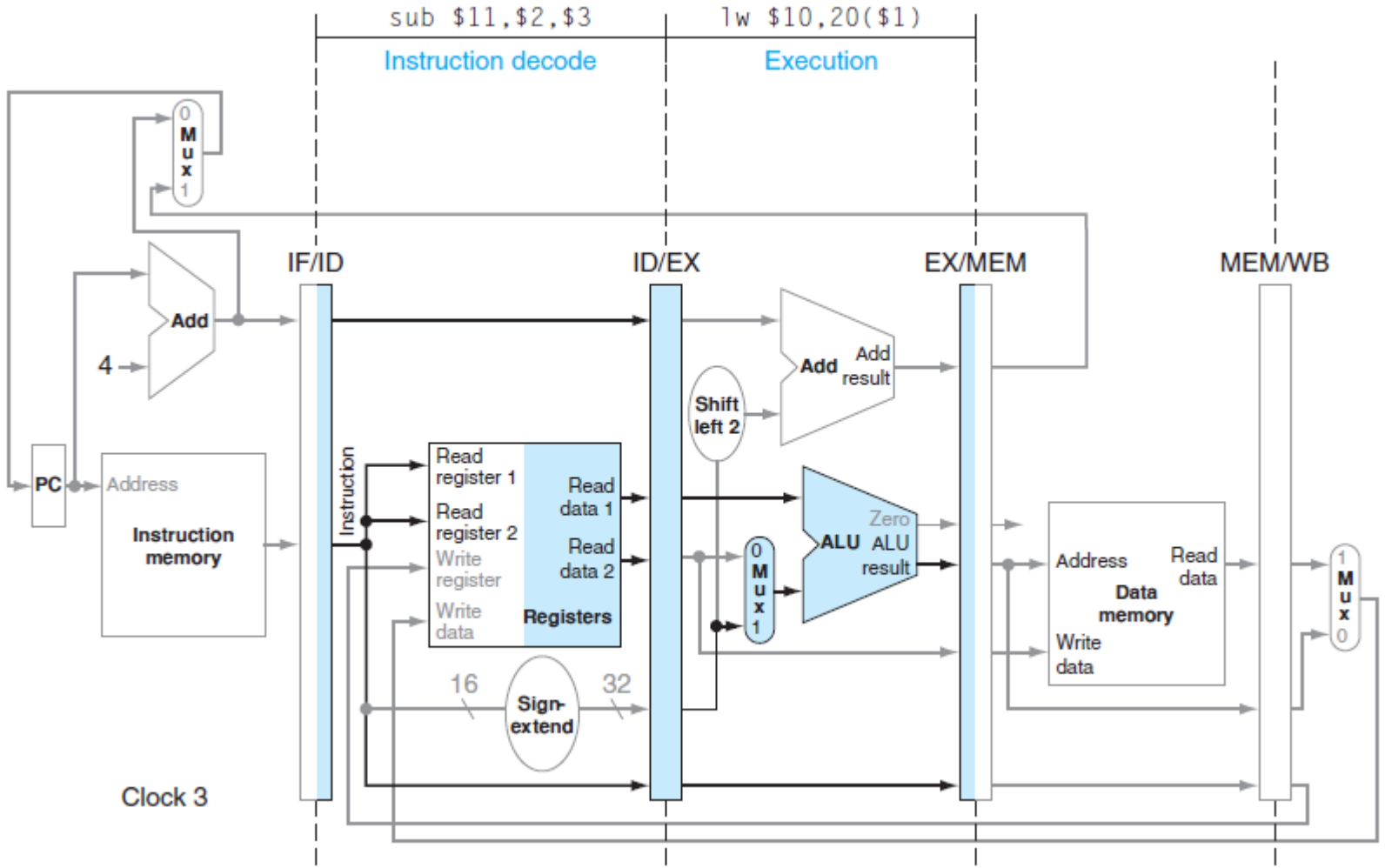
Pipelined example – Clk2

```
lw   $10, 20($1)
sub  $11, $2, $3
```



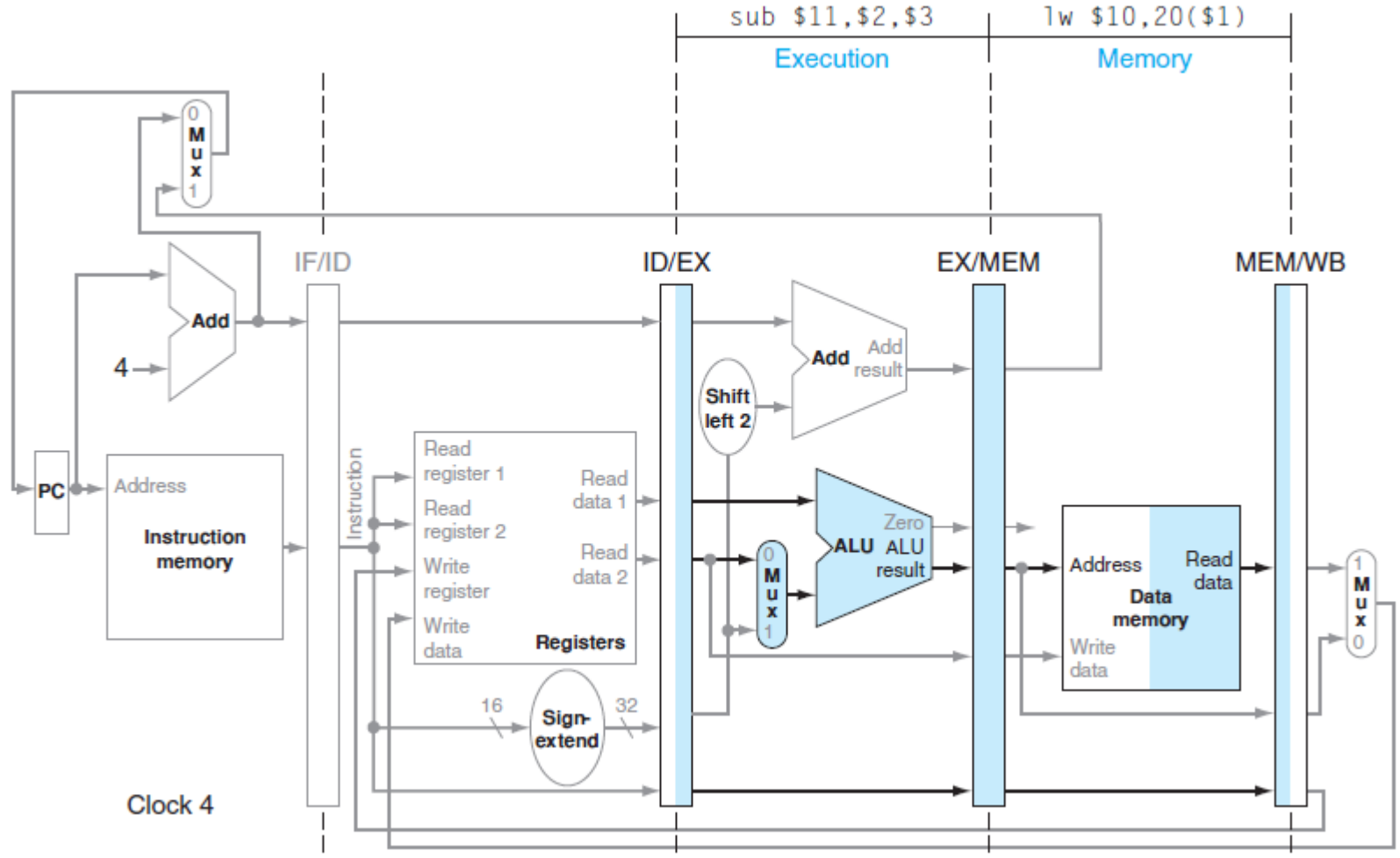
Pipelined example – Clk3

```
lw   $10, 20($1)
sub  $11, $2, $3
```



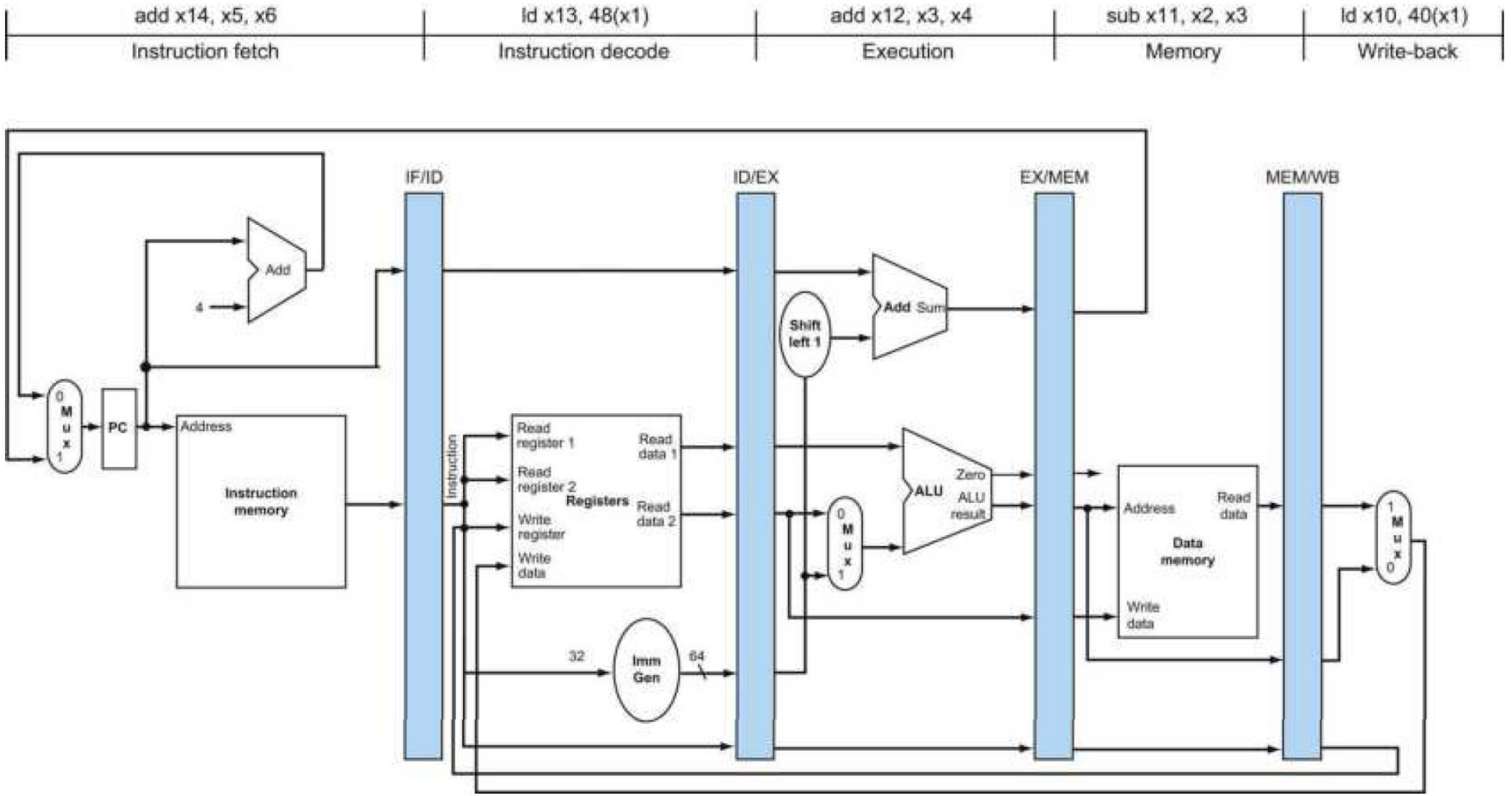
Pipelined example – Clk4

```
lw   $10, 20($1)
sub  $11, $2, $3
```



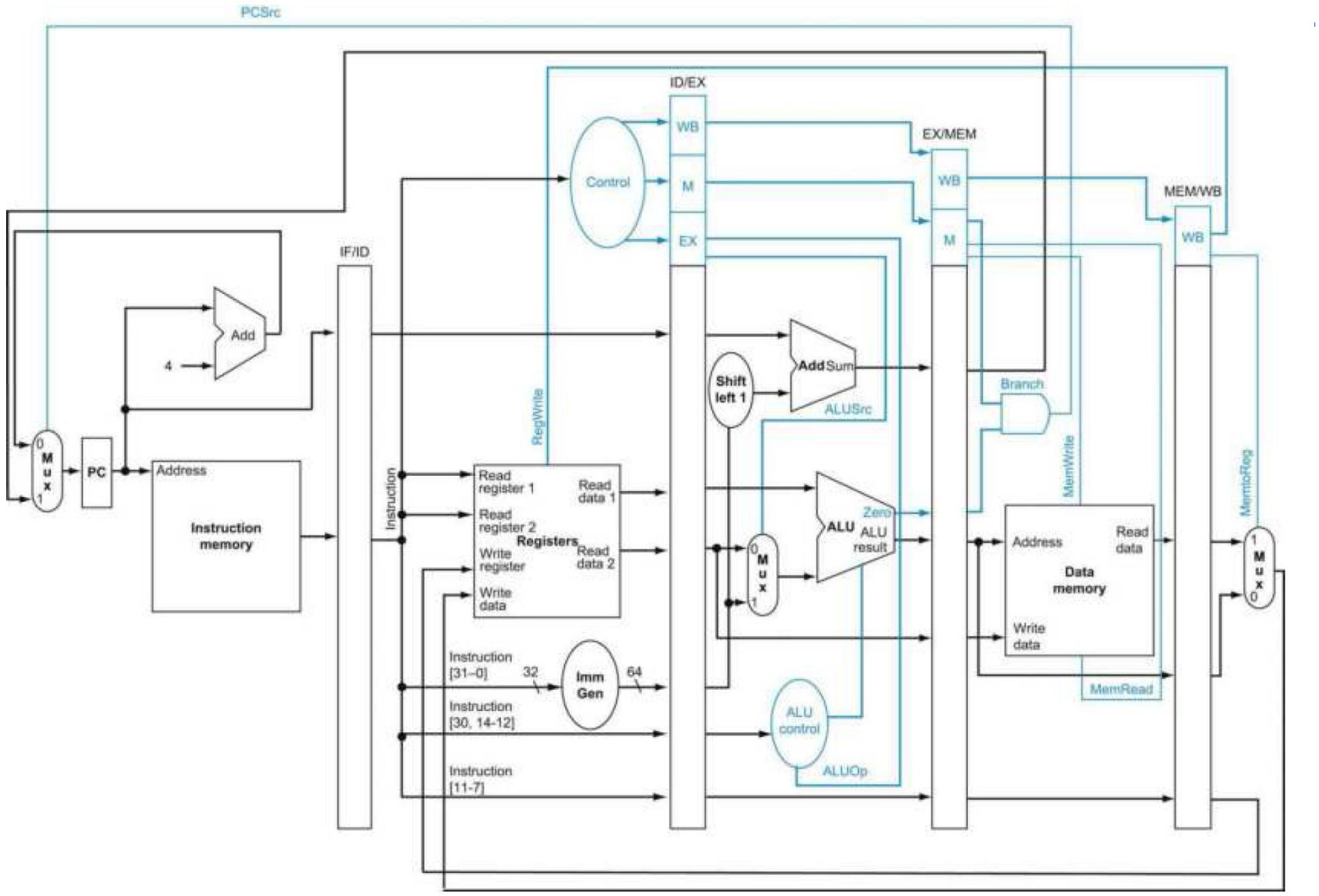
Now you: Clk5, Clk6?

What with the write address?



What with the control signals?

- Control signals can be determined during Decode and held in the state registers between pipeline stages



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease
- RISC-V ISA designed for pipelining
  - All instructions are 32-bits (can fetch and decode in 1 cycle)
  - Few and regular instruction formats
  - Load/store addressing (Can calculate address in 3rd stage, access memory in 4th stage)

Concluding Remarks

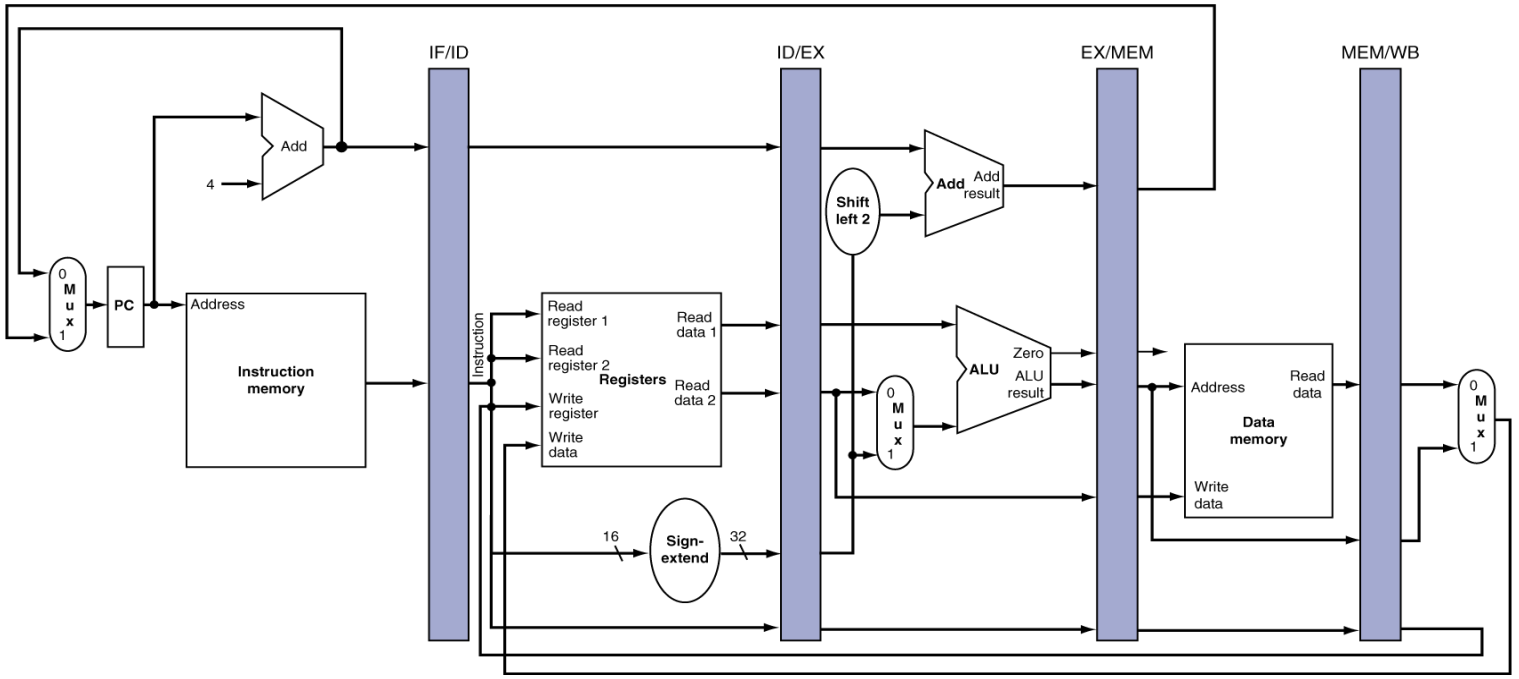
- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced

- Next class (online / recorded!) Hazards: structural, data, control

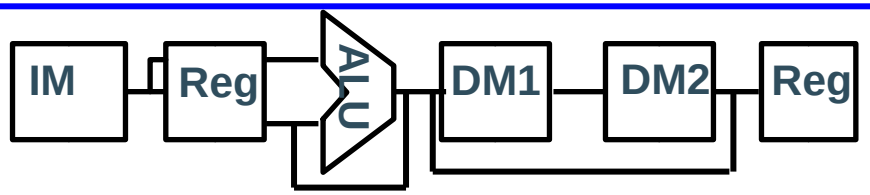
Representative exercises

Exercise1: Modify the pipeline...

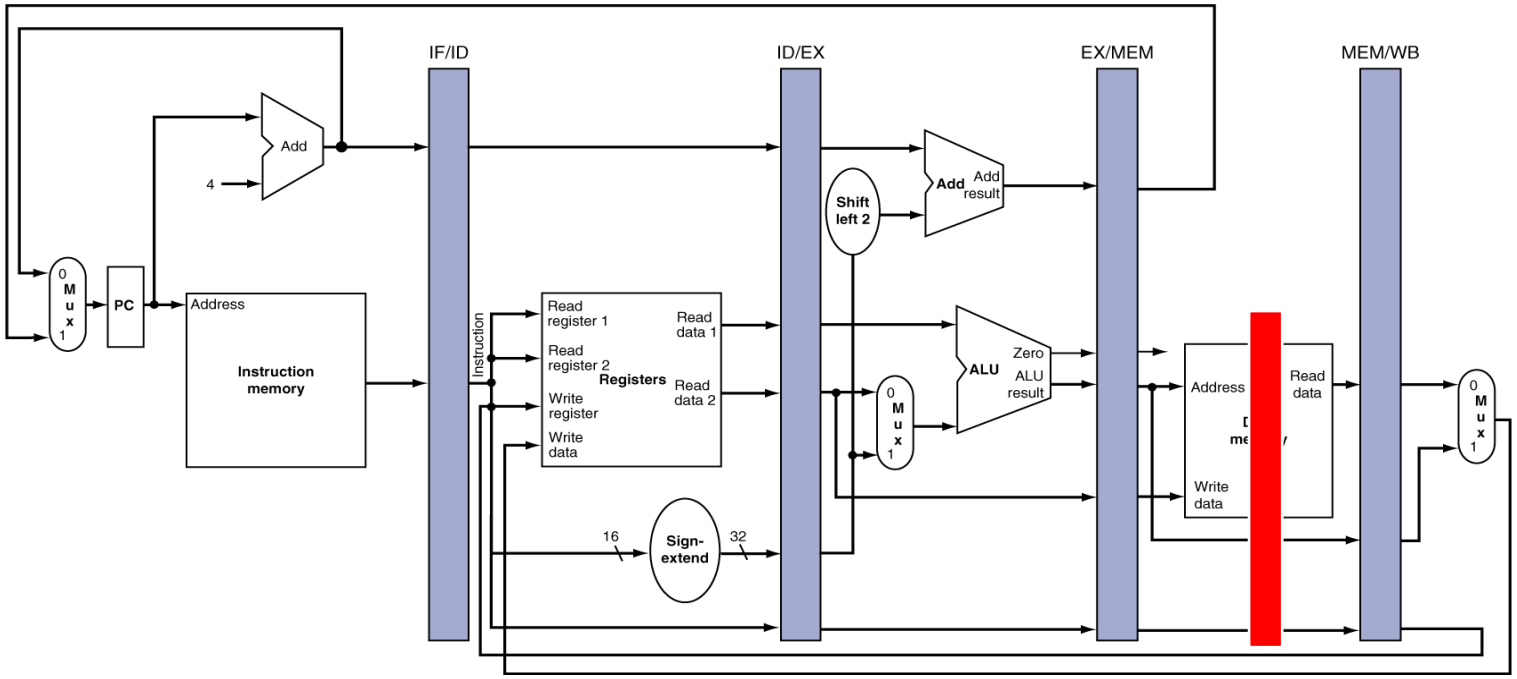
- In the processor underneath, what is the impact if the data memory access is twice as slow as the instruction memory?
- Can we do anything to mitigate this?



Exercise1: Solution



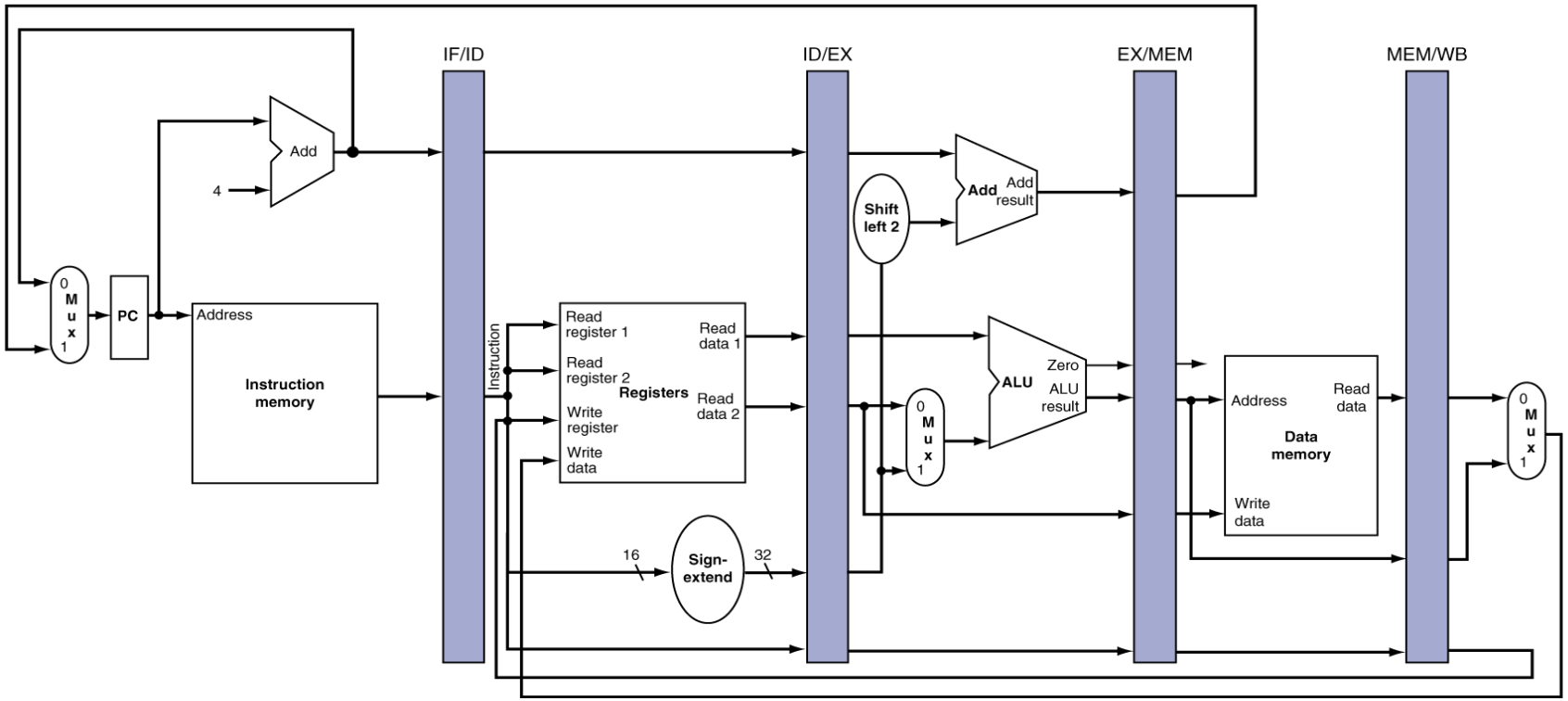
- make the clock twice as slow or …
- let data memory access take two cycles (and keep the same clock rate)



Exercise2: Modify the pipeline

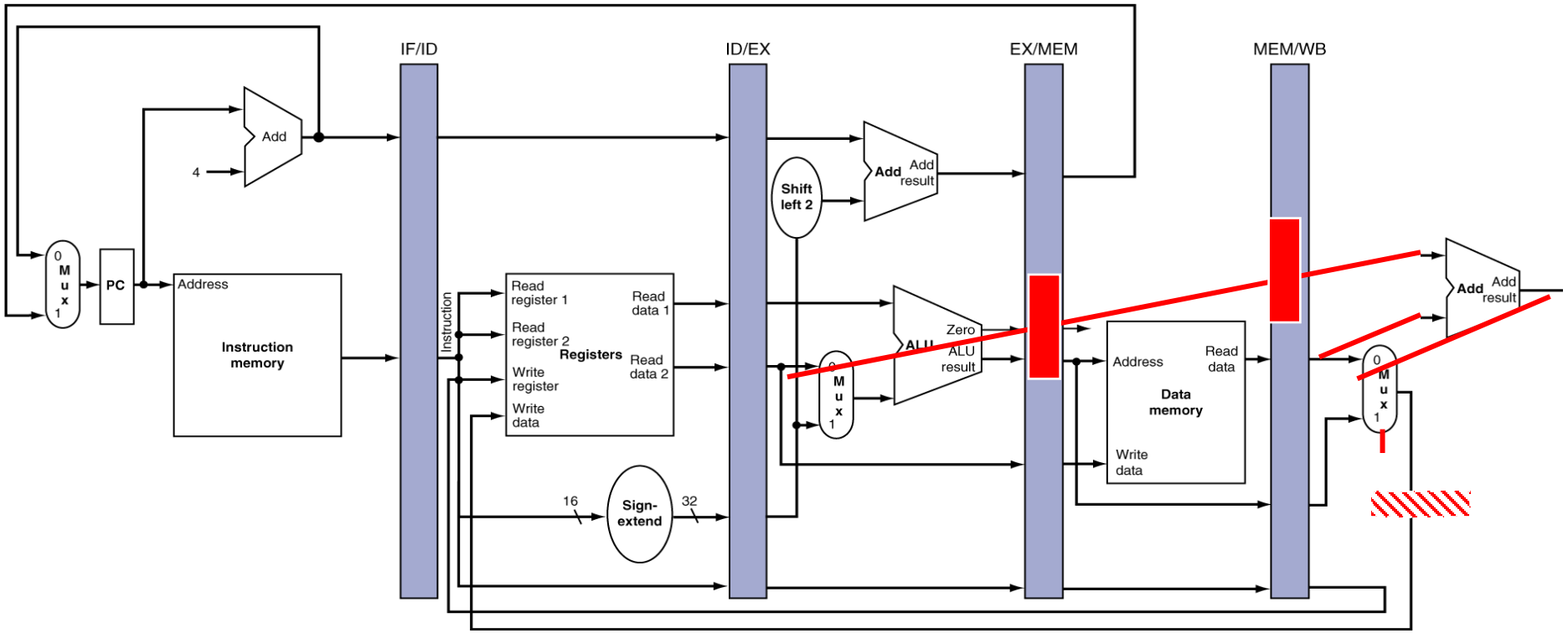
`addM $Rd, $Rt, $Rs` *Rd = Rt + Mem[Rs]*

- What must be changed in the pipeline to add this instruction to the MIPS ISA?
- New control signals needed?
- Where might this instruction be useful?


IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
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
Address
Read data
Data memory
Write data

Exercise2: Solution

- This instruction behaves like a normal load until the end of the MEM stage. After that, it behaves like an ADD, so we need another stage after MEM to compute the result, and we need additional wiring to get the value of Rt to this stage.
- We need to add a control signal that selects what the new stage does (just pass the value from memory through, or add the register value to it).
- can be used when trying to compute a sum of array elements.


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

Exercise3: Modify the pipeline...

- A dedicated multiply operation is very slow, about twice as slow as the ALU operation. Is it best to use a processor that will:
  A. Execute MULT as series of adds and shifts, not needing HW modifications, but needing many cycles per MULT.
  B. Add a dedicated MULT in EX and hence half the clock speed

  Or can we be smarter?
  - Hint: the multilply operation does not need the data memory...

Exercise3: Solution

- Increase the number of pipeline stages for the EX stage to two
- Let it intrude in the MEM stage...