COMP311: *COMPUTER ORGANIZATION!*

Lecture 17: Pipelining the RISC-V CPU

tinyurl.com/comp311-sp26

RISC-V Resources. We are getting into the thick of it now!

- I posted a “reading”. Please do this ASAP if you haven’t!
- Chapter 2 of the RISC-V reader/”Mona Lisa Book”
  - *I will give you p. 15-16 of this book on the next quizzes and on the final as a reference sheet (on the next slides, too)*
- Play around with the simulator!! Leonard built this!
  - *The best way to learn is by doing in this setting. (Think of other times you have learned a programming language ☺)*
- If you want even more details, you can read Andrew Waterman’s PhD dissertation
  - *https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf*

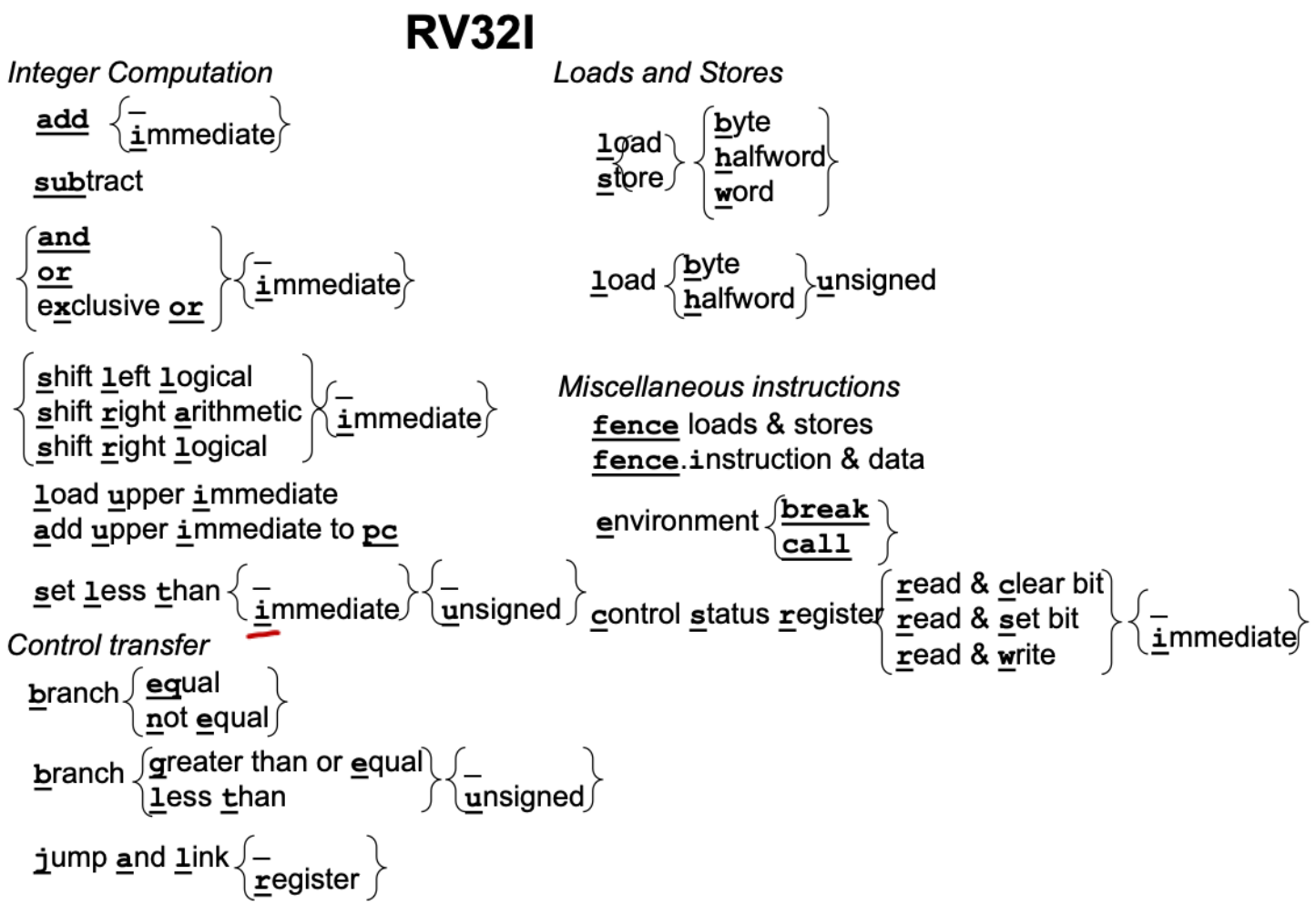


Figure 2.1: Diagram of the RV32I instructions. The underlined letters are concatenated from left to right to form RV32I instructions. The curly bracket notation { } means each vertical item in the set is a different variation of the instruction. The underscore _ within a set means that one option is simply the instruction name so far without a letter from this set. For example, the notation near the upper left-hand corner represents the following six instructions: `and`, `or`, `xor`, `andi`, `ori`, `xori`.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	I fence.i
000000000000		00000	000	00000	1110011	I ecall
000000000001		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I csrrsi
csr		zimm	111	rd	1110011	I csrrci

Figure 2.3: RV32I opcode map has instruction layout, opcodes, format type, and names. (Table 19.2 of [Waterman and Asanović 2017] is the basis of this figure.)

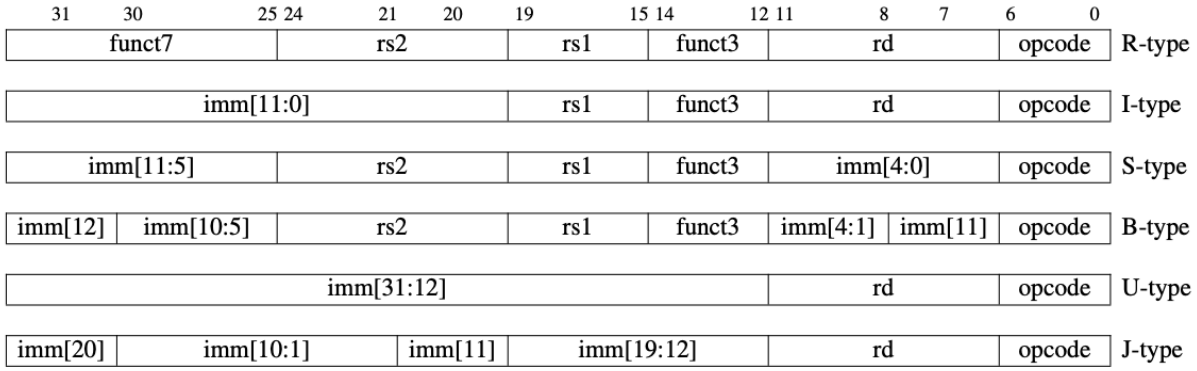


Figure 2.2: RISC-V instruction formats. We label each immediate subfield with the bit position (imm[*x*]) in the immediate value being produced, rather than the bit position in the instruction's immediate field as is usually done. Chapter 10 explains how the control status register instructions use the I-type format slightly differently. (Figure 2.2 of Waterman and Asanović 2017 is the basis of this figure).

Logistics Updates & Quiz Topics

- Computer Performance Metrics, Latency, Throughput
- Pipeline Diagrams
- Pipelining Concepts, How to analyze if a 2 vs. 3 stage pipelined approach (or single-cycle) approach is better
- More emphasis RISC-V Assembly
  - Load and Store Instructions, Reading & Writing to Memory
  - Going from RISC-V to C and C to RISC-V. The written assignment has some examples!

- WA5 is out and due next week!
- Next lab will be released in the next few days. Building the register file for the CPU. ☺
- Review Session for quiz tomorrow at 5pm in SN014

1.1 - D . srai x1, x1, 31

neg: all 1s → (-1)

non-neg: (0)

1.2 B - xori w/ -1

RISC-V REVIEW

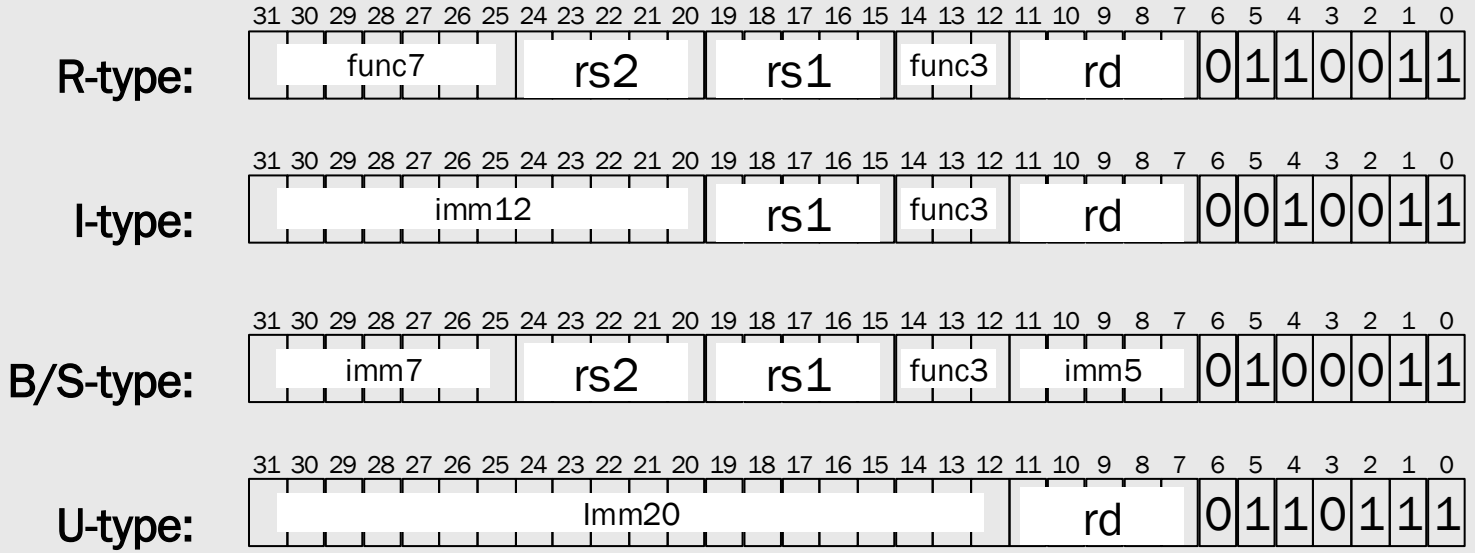
1.3 . C. sw x0, (x1)

src

memaddr

1.4 . E & was x1

The miniRISC-V ISA

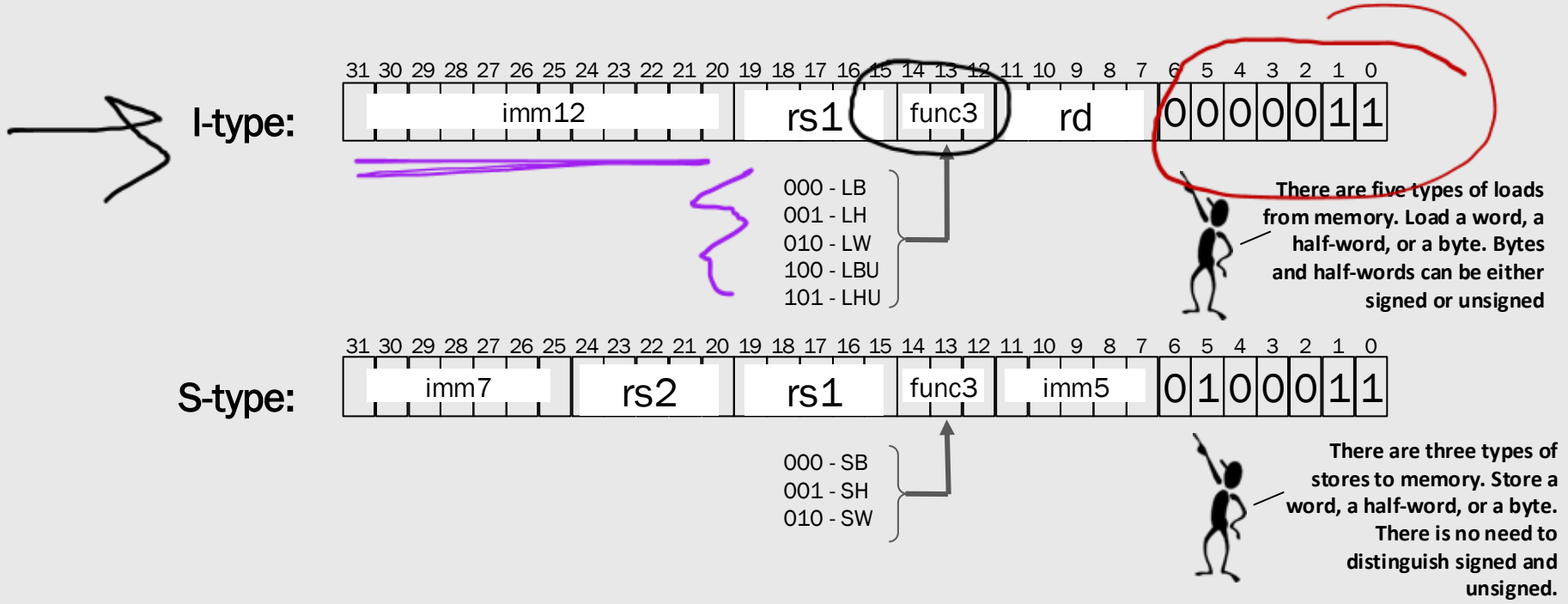


Four key instruction formats (each one has a corresponding opcode!):

1) ALU with two register operands
2) ALU with a register and an immediate operand. (**LOADS HAVE DIFF OPCODE!**)
3) Stores and Branches
4) Jumps and large constants (LUI, AUIPC)

Load and Store Instructions

RISC-V is a “Load/Store architecture”. That means that only a specific class of instructions are used to reference data in memory. As a rule, data is loaded into registers first, then processed, and the results are written back using stores. Load and Store instructions have their own format:



Load Word (lw)

- Used to read 4 bytes (one word) of data from memory
- `lw rd, offset(rs1)`
  - *`rs1` is a register that holds a memory address*
  - *`offset` is a 12 bit immediate value*
  - *The loaded word will be stored in `rd`*

R[rd] = M[R[rs1] + offset]

Store Word (sw)

- Used to store 4 bytes (one word) of data to memory
- `sw rs2 offset(rs1)`
  - *rs1 is a register that holds a memory address*
  - *offset is a signed 12-bit immediate value*
  - *rs2 is the value that will be written at this address*

$$M[R[rs1] + offset] = R[rs2]$$

Load Word vs Store Word

- Load Word
  - *Reads 4 bytes (one word) from memory*
- Store word
  - *Writes 4 bytes (one word) to memory*

Load and Store Options

RISC-V's load and store instructions are versatile. They provide a wide range of addressing modes. Only a subset is shown here.

```
lw   rd,imm12(rs)
```

Rd ← Memory[Rs + imm12]
Rd is loaded with the contents of memory at the address found by adding the contents of the base register, Rs, to the supplied constant

```
lb   rd,-4(rs)
```

Rd ← sign_extend(Memory[Rs - 4])
Offsets can be either added or subtracted, as indicated by a negative sign. The byte is signed-extend to fill the 32 bits of

```
lw   rd,(rs)
```

If no offset is specified it is assumed to be zero

```
sw   rd,12(rs)
sh   rd,(rs)
```

The contents of a register hold the address of the either the data value or a base address for a composite type (structure, object, array, or a stack)

MEMORY REVIEW

Memory

- The register file only contains 32, 32-bit registers
  - *We saw pipeline registers last time, but the programmer doesn't have access to these!*
- We can't store all of our data in the register file. So where does it all go?
- ***In the main memory***

RISC-V Memory Model

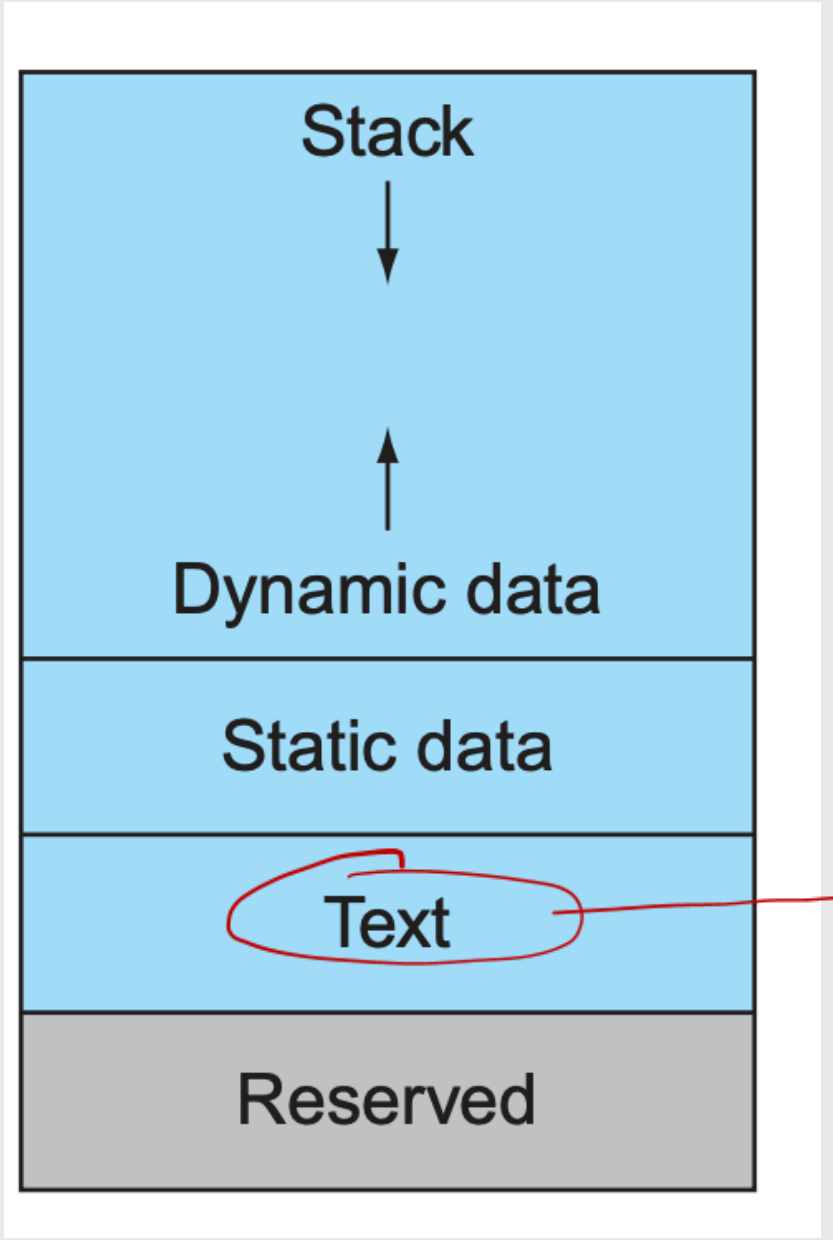
Used to manage function calls and local variables

Used for dynamic memory allocation

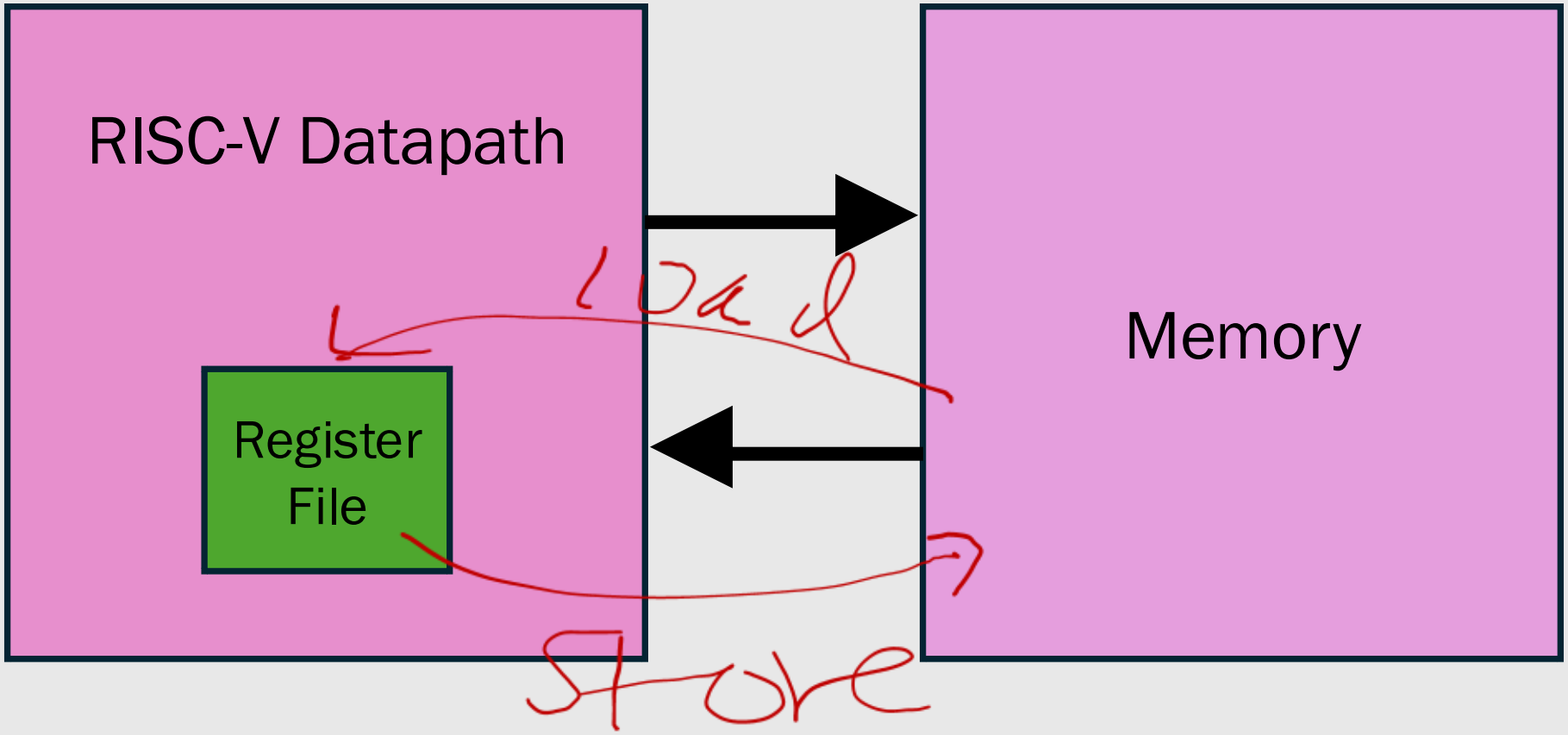
Variables allocated at compile-time

Programs

Reserved for the OS



Only reaches out to memory when a **load** or a **store** happens!



Memory vs. Register File

- So what's the point of the register file?
- It is MUCH, much..! faster to access the register file
- We'll store the values that we are currently working with in the register file

Memory

1. large
2. takes longer to access
3. not part of the CPU

Register File

1. small
2. quick access
3. part of the CPU

5	90	100	40	11
0	1	2	3	4

x10

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x11 after executing the following instruction?

```
lw x11, 0(x10)
```

x11 = 5

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x11 after executing the following instruction?

```
lw x11, 0(x10)
```

5

5	90	100	40	11
0	1	2	3	4

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x12 after executing the following instruction?

```
lw x12, 4(x10)
```

0x 4000 + 4 ⇒ 0x 4004

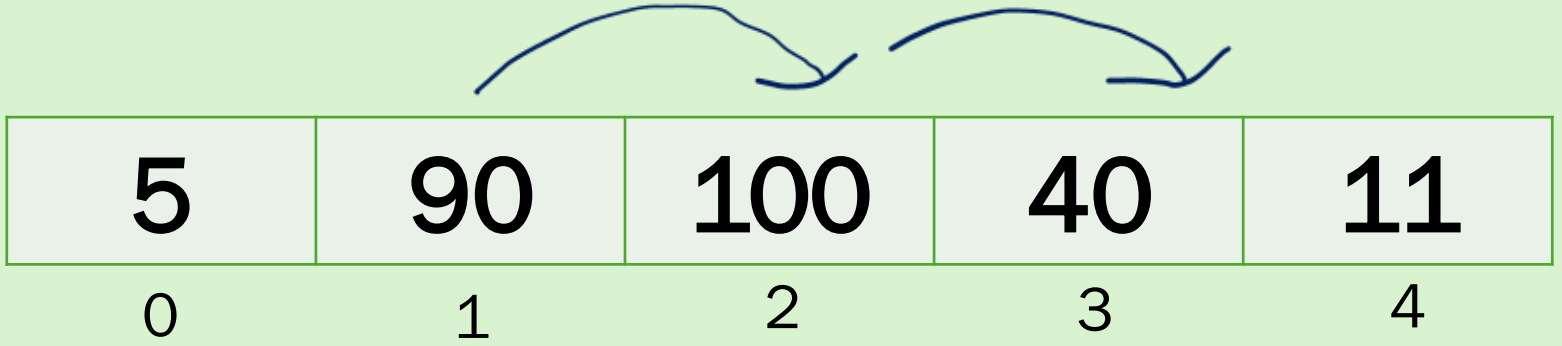
x12 = 90

5	90	100	40	11
0	1	2	3	4

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x12 after executing the following instruction?

```
lw x12, 4(x10)
```

90



- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x12 after executing the following instructions?

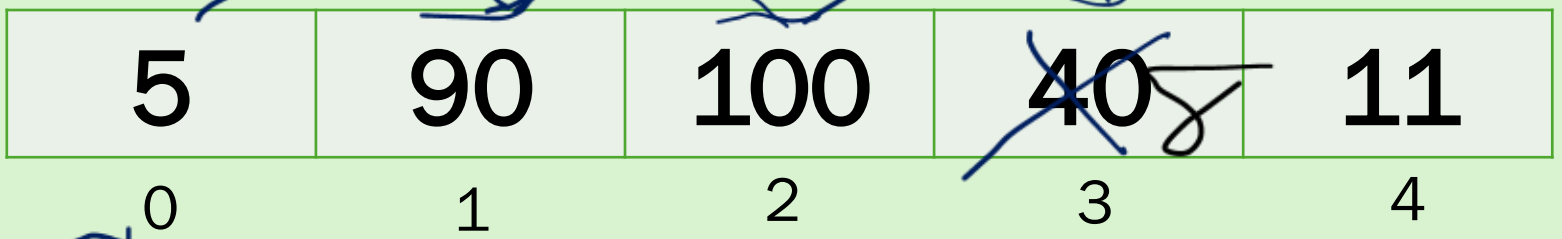
```
addi x10, x10, 4
lw x12, 8(x10)
```

x12 = 40

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x12 after executing the following instructions?

```
addi x10, x10, 4
lw x12, 8(x10)
```

40



- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000 and x7 = 8.
- What are the contents of the array after executing the following instruction?

```
sw x7, 12(x10)
```

5	90	100	40	11
0	1	2	3	4

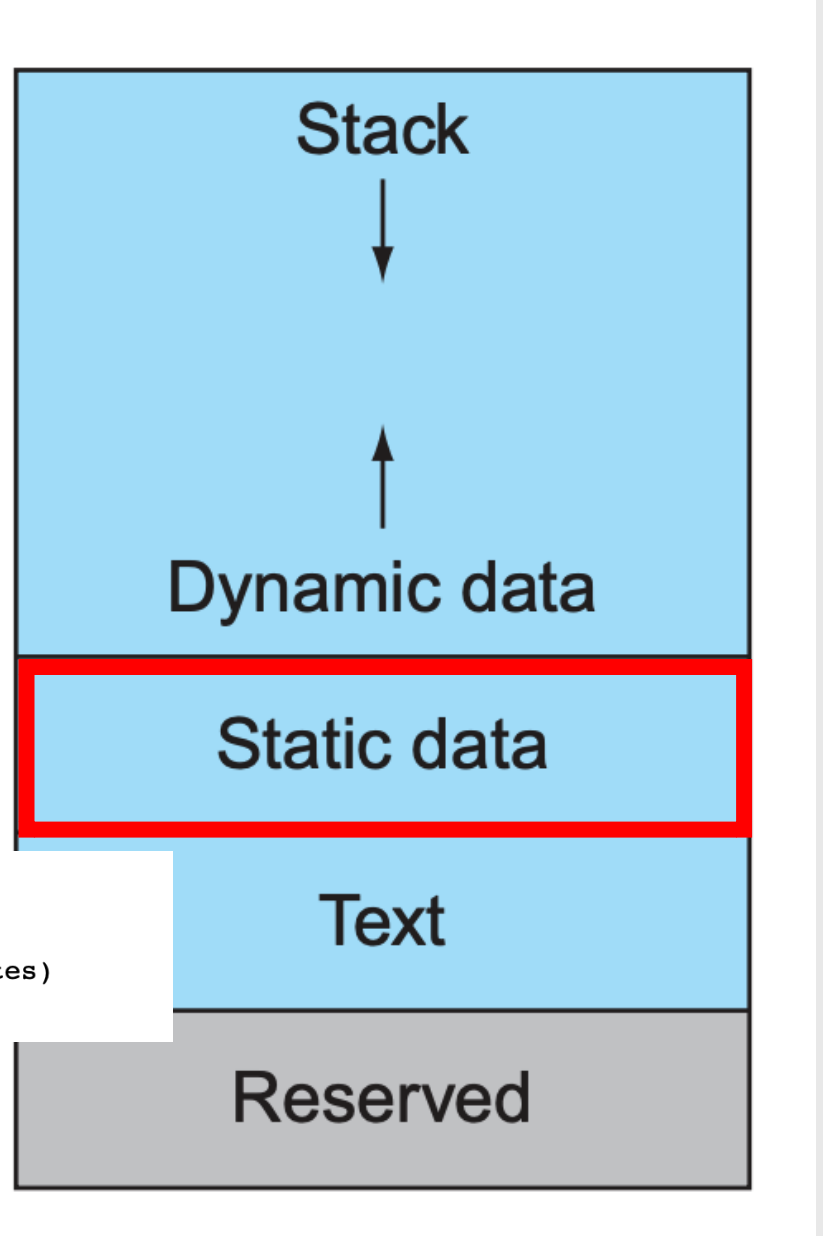
- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000 and x7 = 8.
- What are the contents of the array after executing the following instruction?

```
sw x7, 12(x10)
```

5	90	100	8	11
0	1	2	3	4

Storing Arrays in Memory

- We will store our statically allocated arrays in the static data segment
- To store data in the static data segment, we use an assembler directive.
  - *The miniRISCV assembler and simulator provides a set of directives for allocating space for data and initializing variables*




```
Fib:    .word   1,1,2,3,5,8,13,21                          # first 8 Fibonacci numbers
masks:  .word   0x000000ff,0x0000ff00,0x00ff0000,0xff000000# byte masks
Name:   .string"Leonard"                                   # 0-terminated string (8 bytes)
array:  .space  20                                         # 20 uninitialized words
```

Memory Example

la x14, A

- If the address of A is 0x00003000, what is the value of registers 14-18 after the program is run?

x14 == A0

```
A: .word     5 30 100 40 11

lw   x15, 0(x14)    # x15 = A[0]
addi x16, x14, 8    # x16 = &A[2]
lw   x17, 4(x16)    # x17 = A[3]
sw   x17, 4(x14)    # A[1] = x17
lw   x18, -4(x16)   # x18 = A[1]
```

x14: 0x3000

x15: 5

x16: 0x3008

x17: 40

x18: 40

Load Address

- How do you get the address of the array into a register?
  - *with an instruction called load address (la)*
- la rd Label
  - *places the address of* ***Label*** *in rd*
- This is actually a ***pseudoinstruction***

AUIPC is an instruction we will talk about more in a bit… it adds an immediate to the PC. Let's review what the PC is / related to program execution.

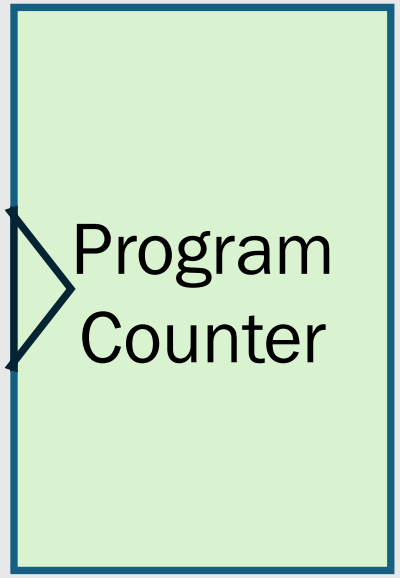
A pseudoinstruction is a "fake" assembly instruction that doesn't exist in HW, but assembler expands into one or more real RISC-V instructions!

Loads the address of the label into *Reg[rd]*. It is typically implemented using the two instruction sequence:

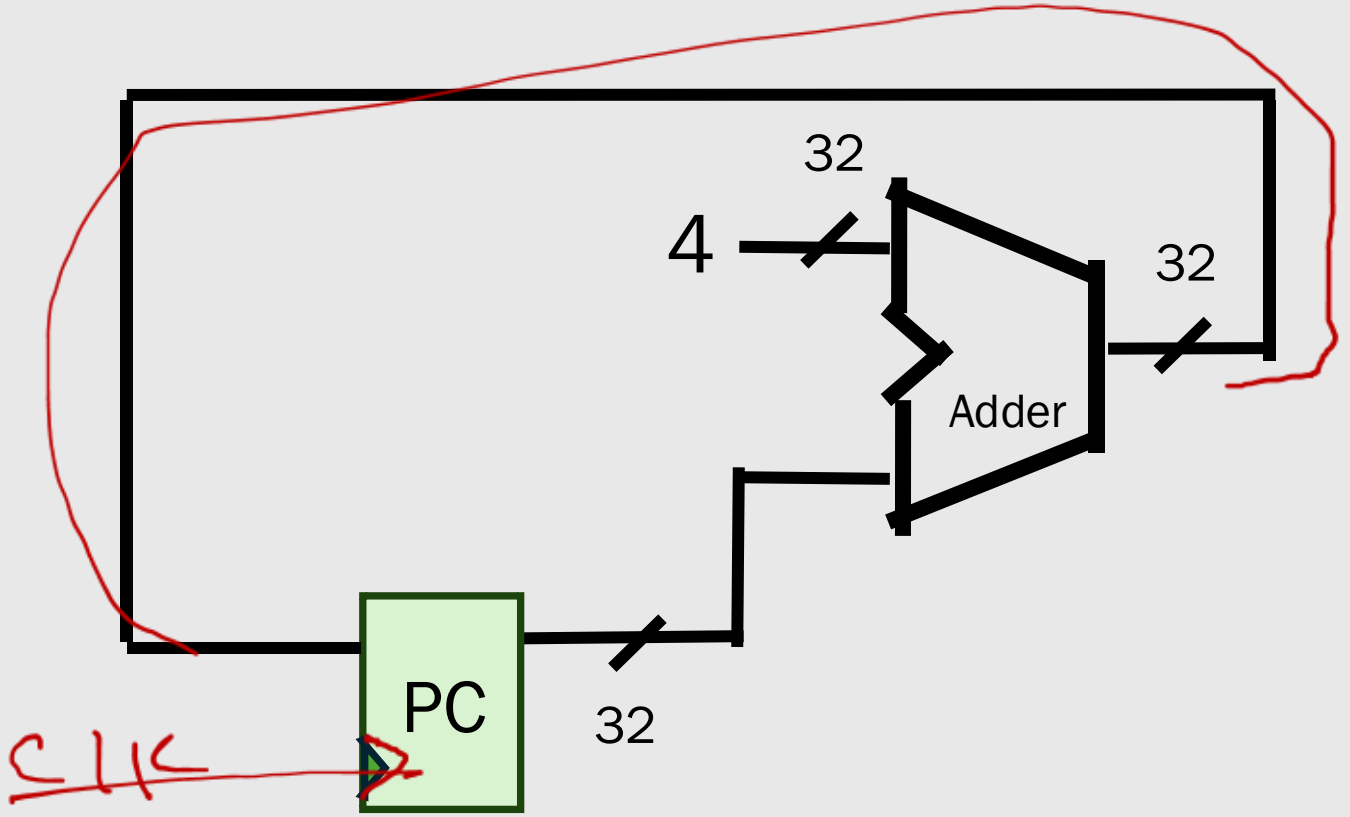
```
auipc rd, label
addi rd, rd, label
```

Program Counter (PC)

- The Program Counter (PC) is a 32-bit register
- The PC is not inside of the register file!!!
- The datapath executes one instruction per clock cycle
- On every clock cycle, PC is incremented by 4 to fetch the next instruction



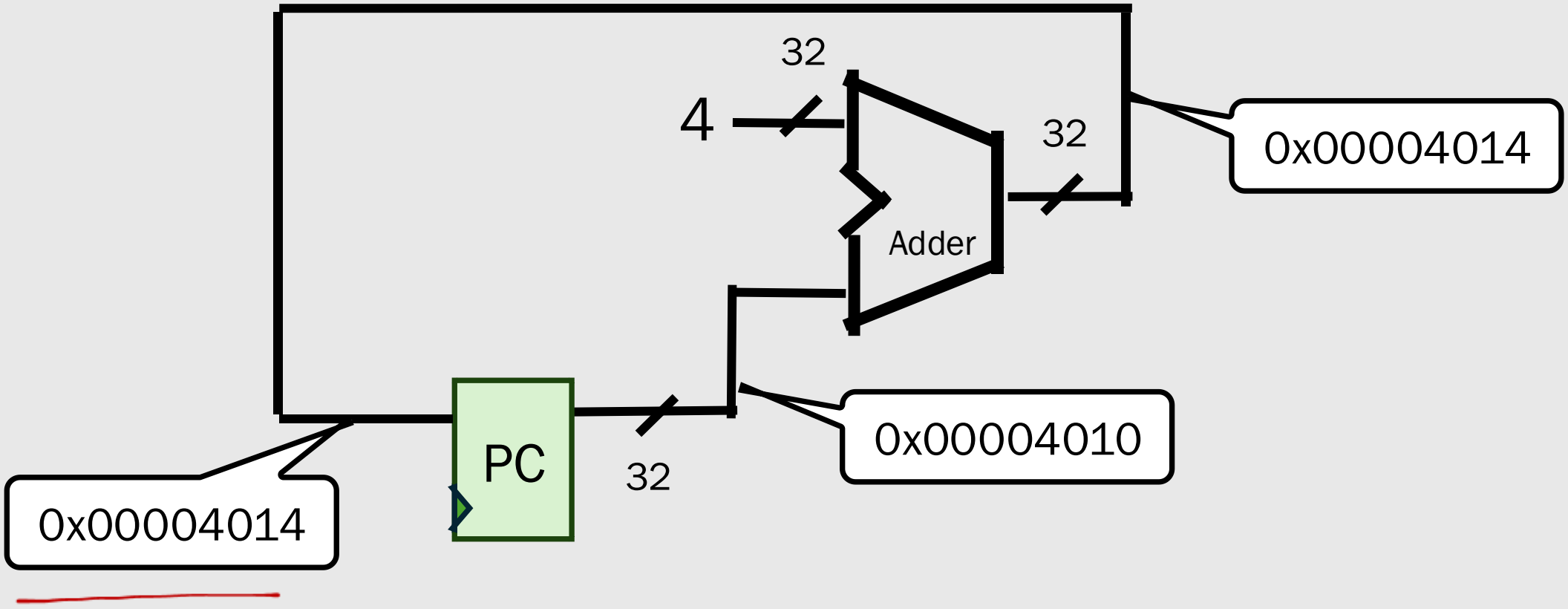
Program Counter (PC)



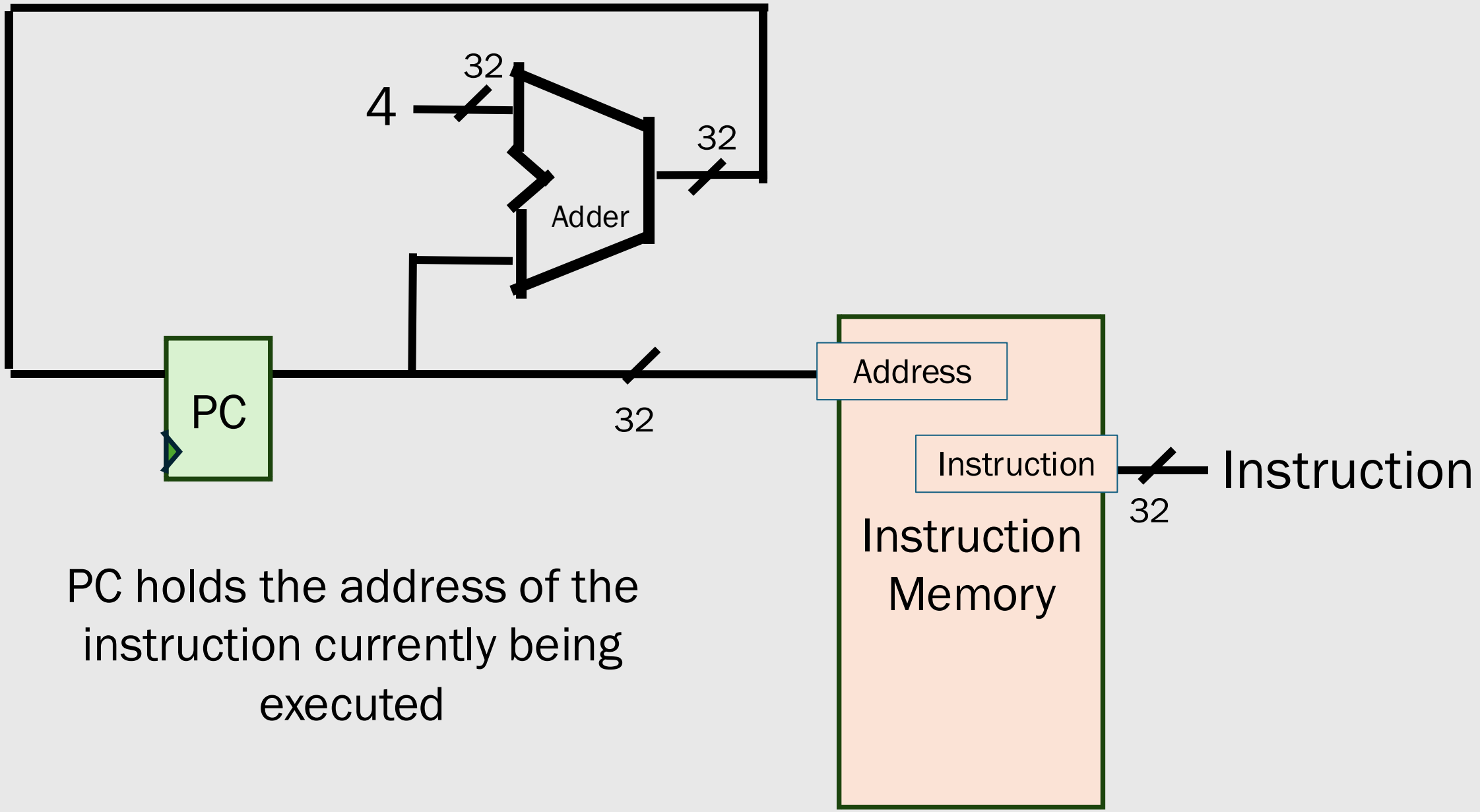
Increment the PC by 4 on every clock cycle

Program Counter (PC)

Example: Current PC = 0x00004010



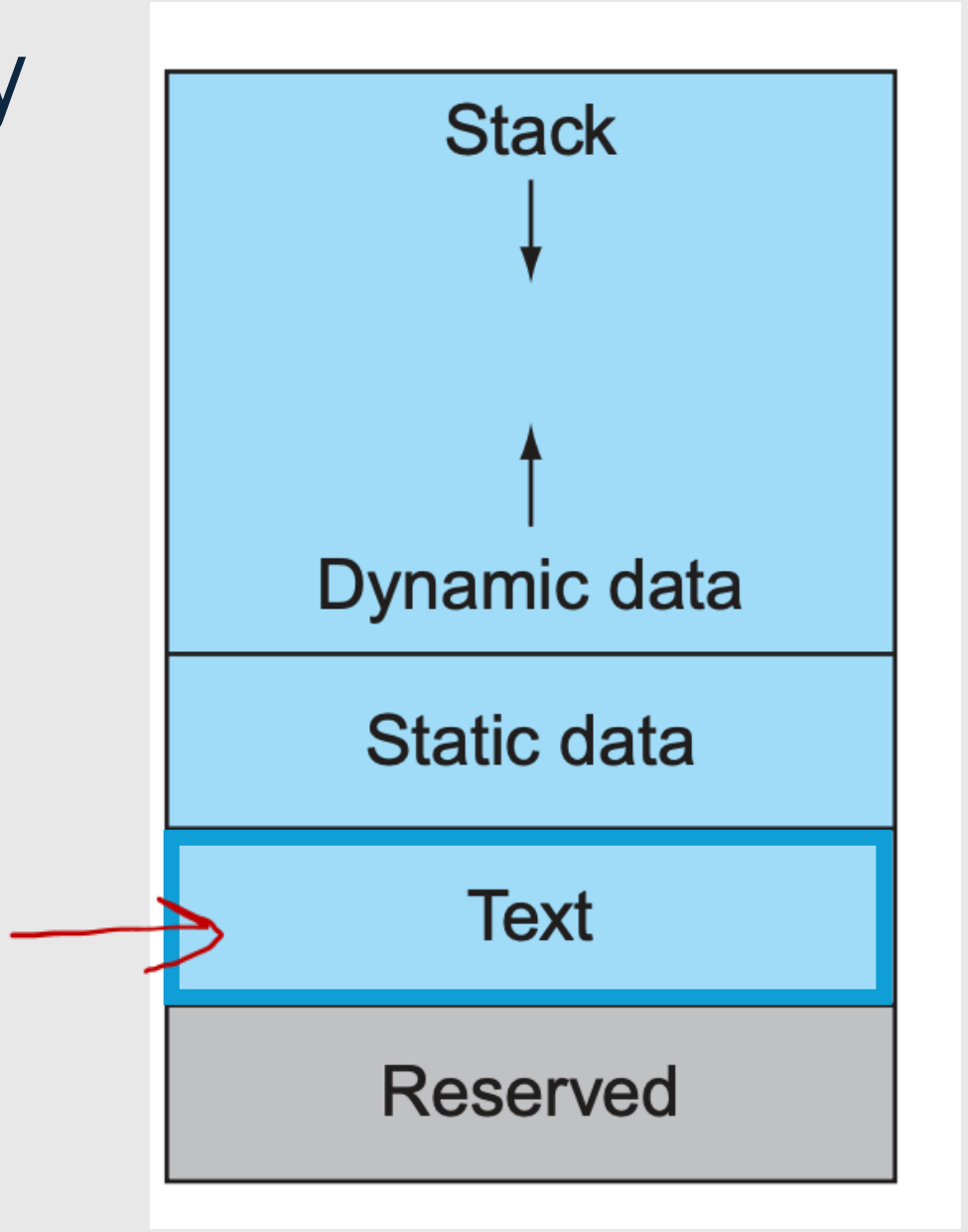
On the next rising edge of the clock, PC will become 0x00004014



PC holds the address of the instruction currently being executed

Storing Code in Memory

- Code is stored in the text segment
- To store code in the text segment, we use the .text directive



Load and Stores in action

Array access in C → in RISC-V: compute addr, load value, then use it!

An example of how loads and stores are used to access arrays.

C:

```
int sum = 0;
int values[10] = {1,3,5,7,9,11,
                  13,15,17,19};

int i;

for (i = 0; i < 10; i++)
    sum += value[i];
```

Assembly:

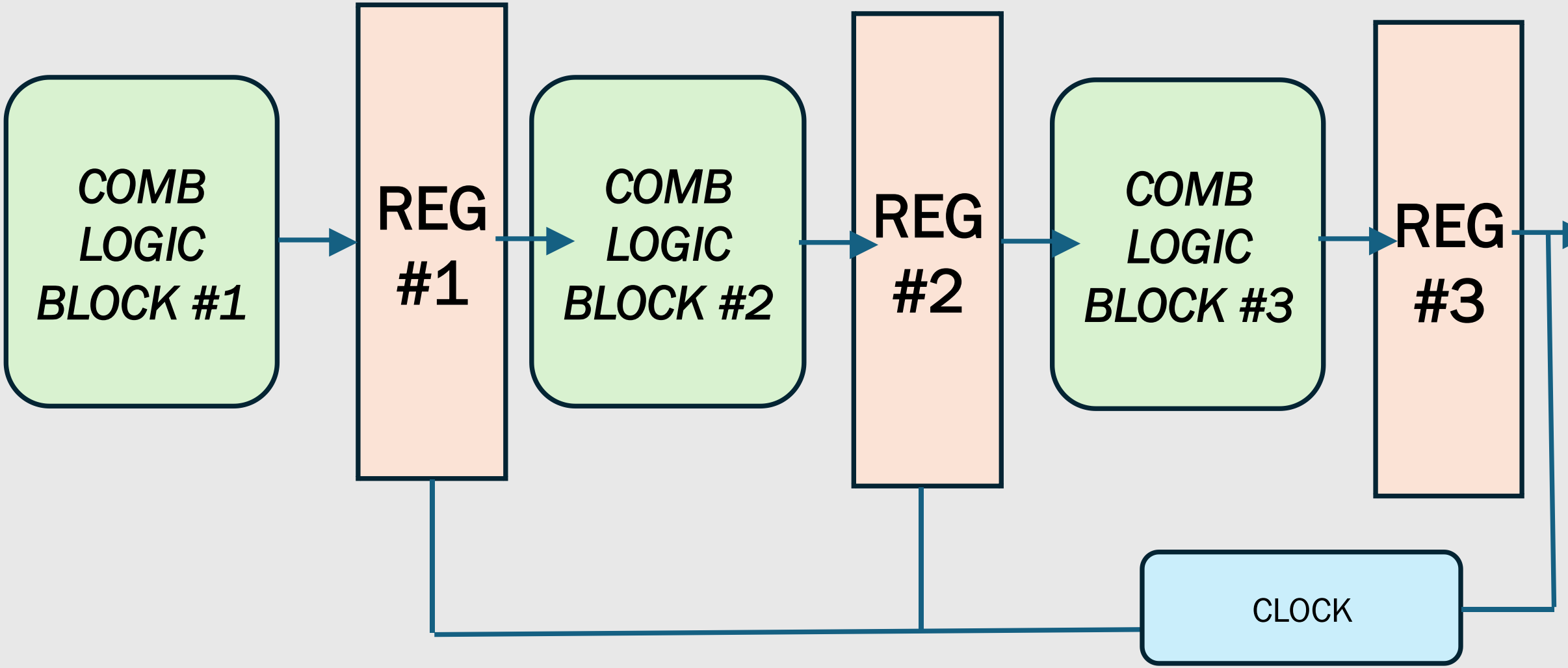
```
        addi    x31,x0,10
        addi    x5,x0,0         # x5 is i
loop:   slli    x6,x5,2
        lw      x6,values(x6)   # value[i]
        lw      x7,sum(x0)      # x5 is sum
        add     x7,x7,x6        # sum += value[i];
        sw      x7,sum(x0)
        addi    x5,x5,1
        blt     x5,x31,loop
*halt:  jal     x0,halt

sum:    .word   0
values: .word   1,3,5,7,9,11,13,15,17,19
```

PIPELINING THE RISC-V CPU!

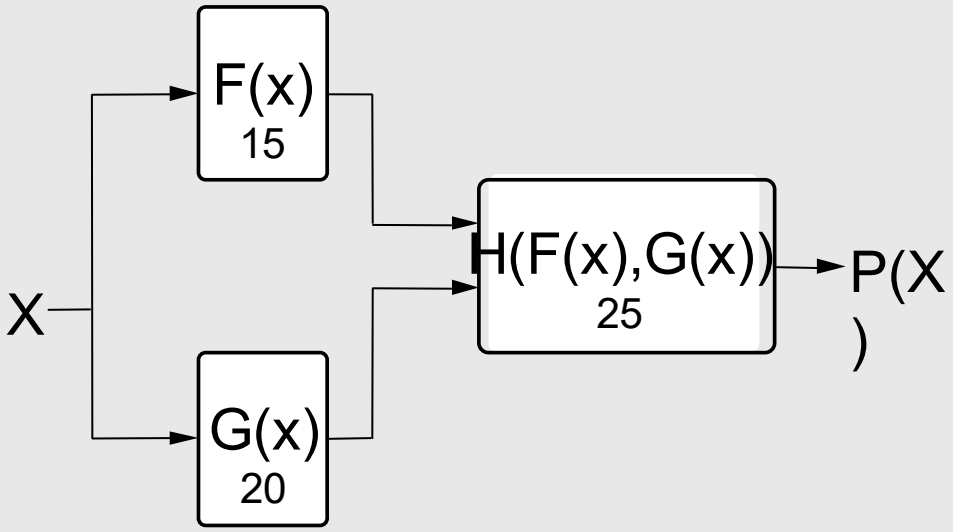
Recall: Pipeline Registers

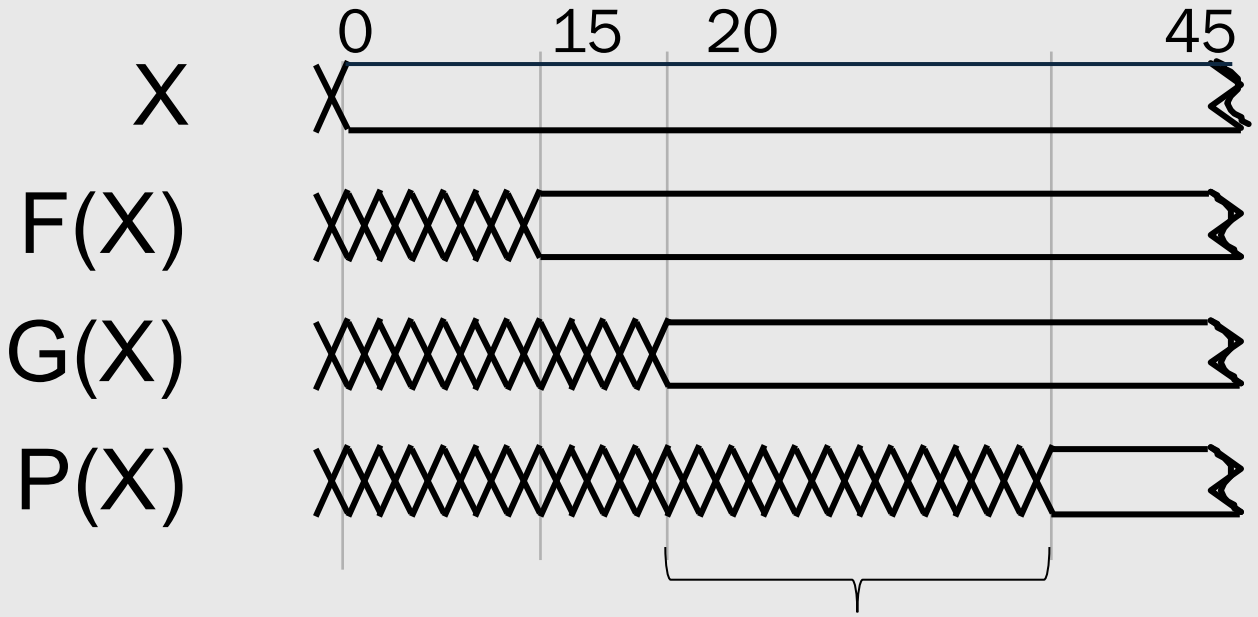
- Inserted between stages of execution to hold intermediate values.
  - *Ex: ALU operands that were decoded!*
- These registers are not visible / programmable by the user
  - *Not part of the 32 general purpose registers*
- Allow the **synchronization** of execution of multiple operations (AKA: **instructions)**
  - *Different stages operate on different instructions in each clock cycle*

Recall: Pipeline Registers



Some Review: Unpipelined Circuits





For combinational logic:

latency = t_{PD},

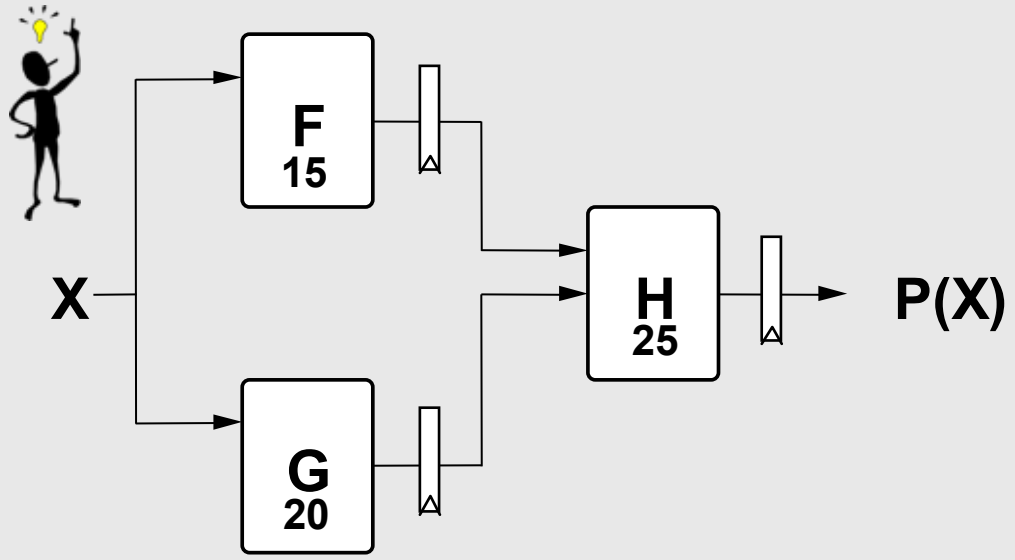
throughput = $1/t_{PD}$.

We can't get the answer faster. Are we making effective use of our hardware at all times?

F & G are "idle", just holding their outputs stable while H performs its computation

Pipelined Circuits

Use registers to hold H's inputs stable!



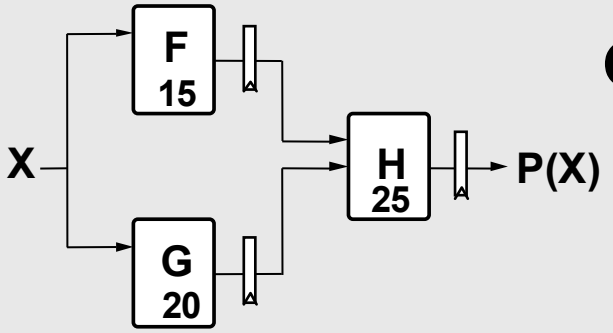
Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ***ideal zero-delay registers*** *(t_{pd} = 0):*

Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage pipeline: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

	latency	*throughput*
unpipelined	**45**	**1/45**
2-stage pipeline	**50**	**1/25**
	worse	***better***

Pipelining uses registers to improve the throughput of combinational circuits

Pipeline Diagrams



Clock cycle

Pipeline stages	i	i+1	i+2	i+3	
Input	X_i	X_{i+1}	X_{i+2}	X_{i+3}	…
F Reg		$F(X_i)$	$F(X_{i+1})$	$F(X_{i+2})$	
G Reg		$G(X_i)$	$G(X_{i+1})$	$G(X_{i+2})$	…
H Reg			$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

This is an example of parallelism. At any instant we are busy computing 2 results.

A pipeline diagram is just a depiction of what inputs are being processed during a given clock period. The results associated with a particular set of input data move *diagonally* through the diagram, progressing through one pipeline stage on each clock cycle.

Pipeline Conventions

DEFINITION:

A K-Stage Pipeline ("K-pipeline") is an acyclic circuit having exactly K registers on every path from an input to an output.

A COMBINATIONAL CIRCUIT is thus a 0-stage pipeline.

CONVENTION:

Every pipeline stage, hence every K-Stage pipeline, has a register on its OUTPUTS (as opposed to, alternatively, its inputs).

ALWAYS:

The CLOCK common to all registers ***must*** *have a period sufficient to allow for the propagation delays of all combinational paths (i.e. the worst case path) PLUS (input) register's* t_{PD} *PLUS (output) register's* t_{SETUP}. **[We haven't gotten to setup time yet]**

The LATENCY of a K-pipeline is K times the period of the clock common to all registers.

The THROUGHPUT of a K-pipeline is the frequency of the clock.

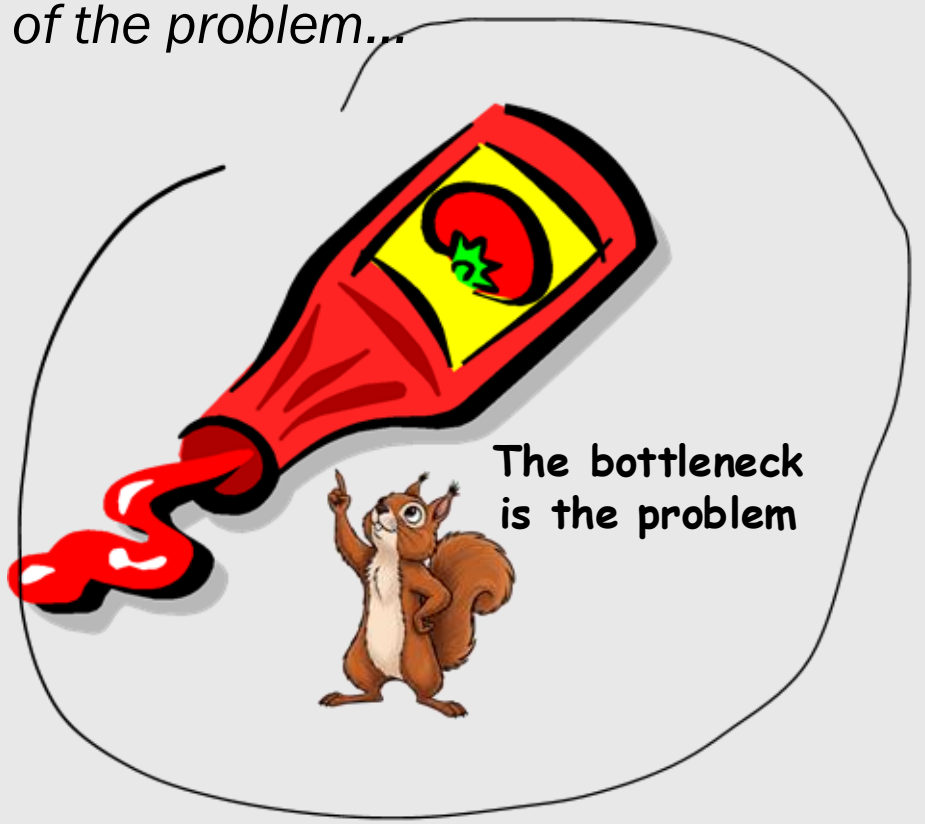
Pipelining Summary

Advantages:

- *Higher throughput than combinational system*
- *Different parts of the logic work on different parts of the problem…*

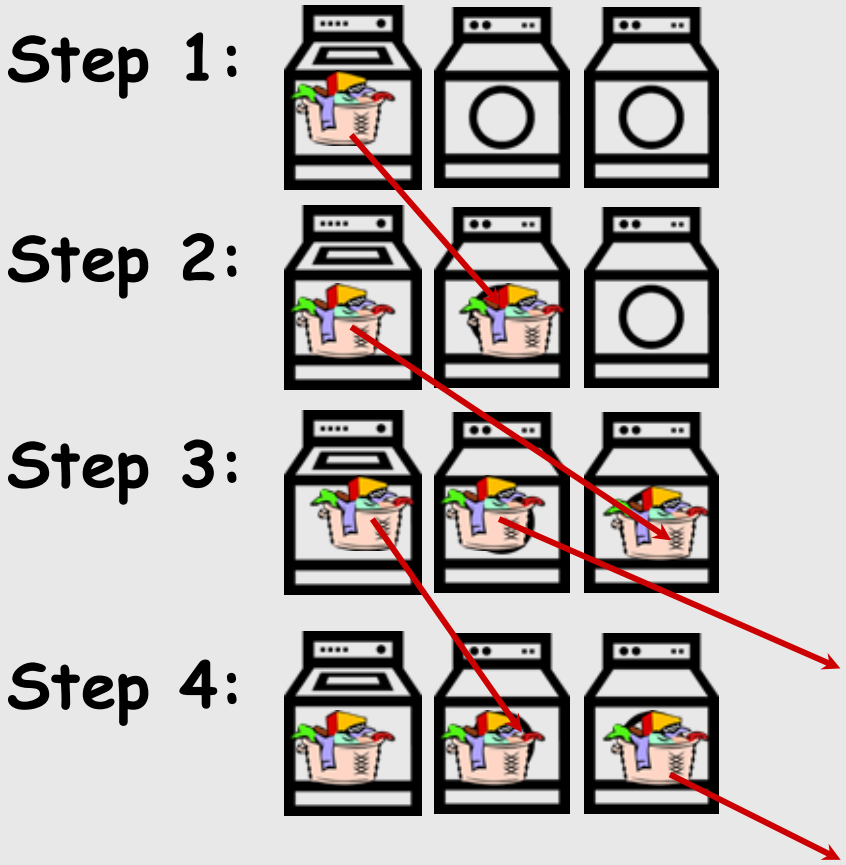
Disadvantages:

- *Generally, increases latency*
- *Only as good as the *weakest* link (often called the pipeline's BOTTLENECK)*



Is there a way around this "weak link"?

How UNC students REALLY do Laundry?



How to work around a bottleneck.

First, find a place with twice as many dryers as washers.

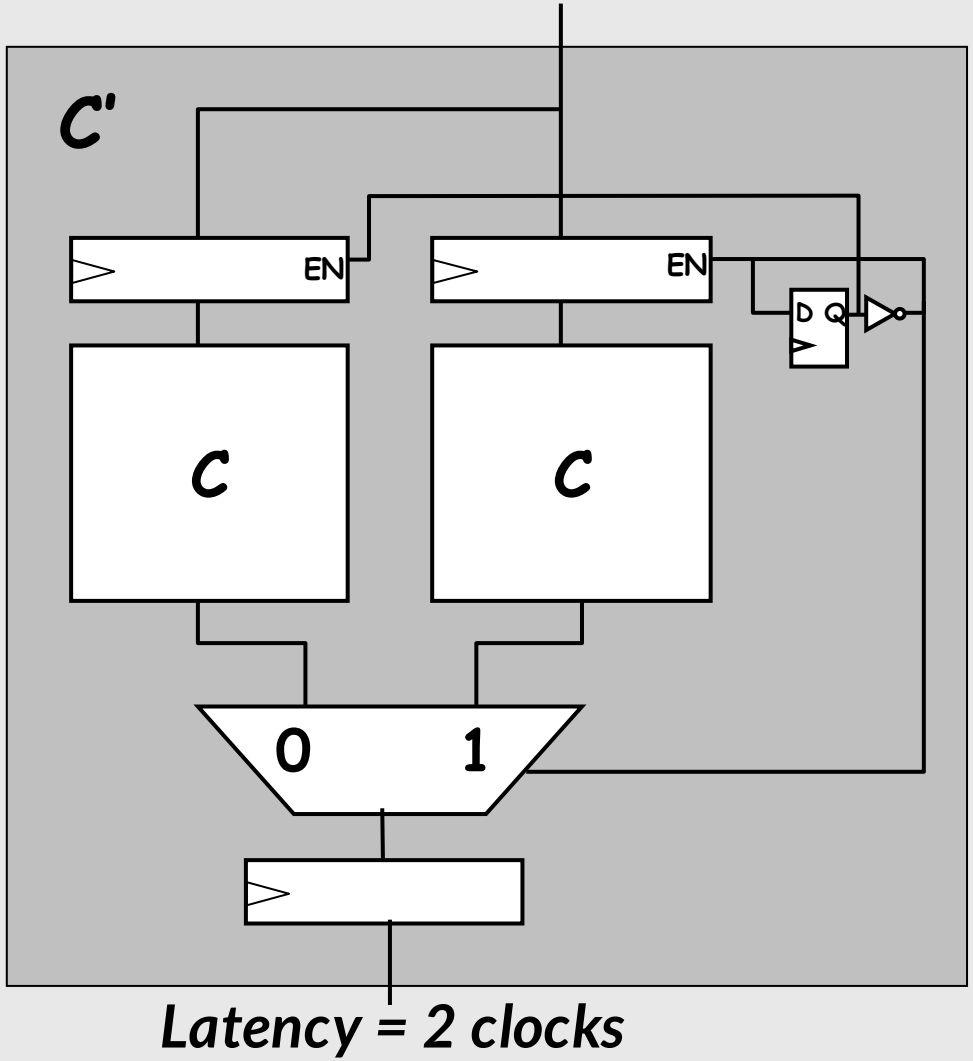
Throughput = 1/30 loads/min

Latency = 90 mins/load

This Speed-up is called "Interleaving"

One way to overcome a pipeline bottleneck is to **replicate the critical element** as many times as needed and *alternate* inputs between the various copies.

N-way interleaving is equivalent to how many pipeline Stages? **N**



Another Approach… Parallelism

We can combine interleaving and pipelining with parallelism.

Throughput = 2/30 / 1/15 load/min

Latency = 90 min

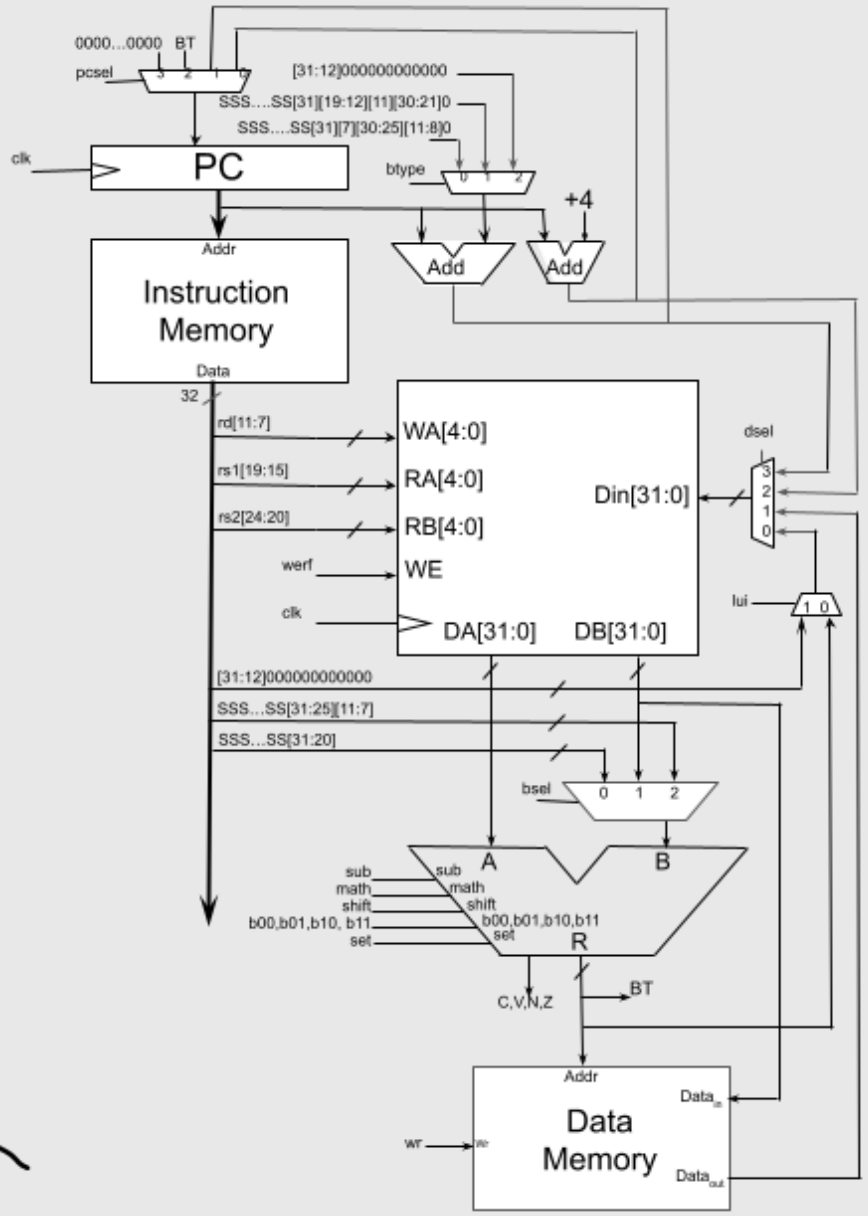
How to Pipeline This?

A CPU is a digital circuit like any other. Thus, we should be able to pipeline it to increase its throughput.

However, there are a few tricky issues.

- It already has registers that get updated on each clock (register file and PC)
- It has feedback, the ALU or Data Memory outputs are routed back to the register file

Pipeline hazards !



Our Goal

A simple 3-stage pipeline:

Fetch:

Instruction memory access

Decode:

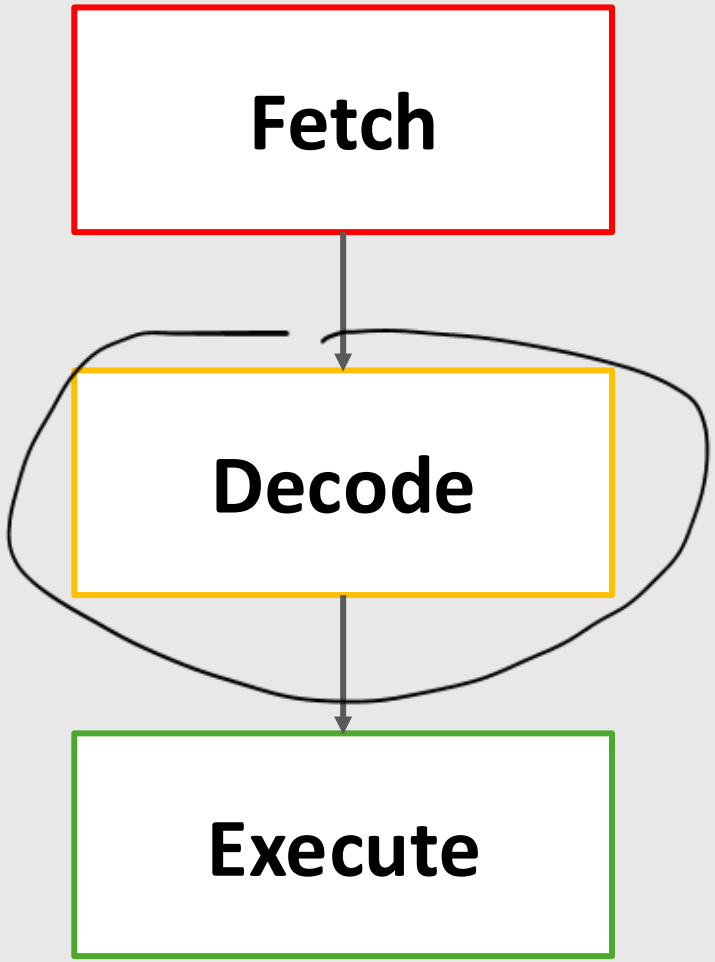
Decode instruction

Get source register operands

Execute:

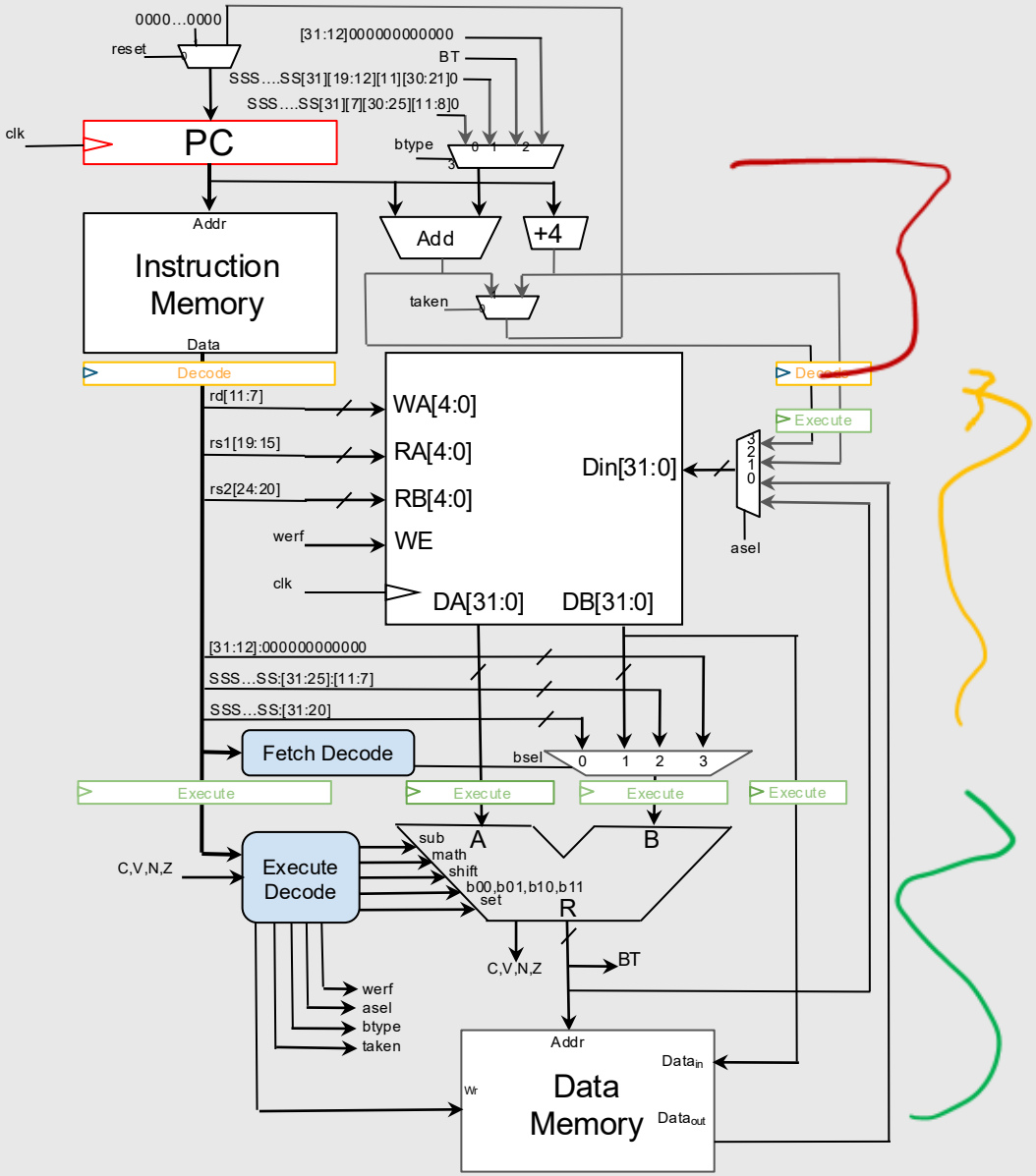
ALU operation

Write-back destination register



Pipelining the Datapath!

- Fetch, Decode, and Execute Stages
- Instructions are decoded in both the Fetch and Decode stages
- Register ports are “Read” in the Decode stage and “Written” at in the end of
  the Execute stage
- PC+4, and PC relative calculations are “delayed”
  for use in later stages



How Instructions flow

Consider the following instruction sequence:

Progress in a three-stage pipeline

Once filled, at every clock there are 3 instructions at various stages of execution.

```
...
sub     t1,t1,t2
addi    t2,t2,2
andi    t0,t0,1
slt     t2,t2,t0
```

Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	sub t1,t1,t2					
Decode						
Execute						

How Instructions flow

Consider the following instruction sequence:

Progress in a three-stage pipeline

Once filled, at every clock there are 3 instructions at various stages of execution.

```
...
sub     t1,t1,t2
addi    t2,t2,2
andi    t0,t0,1
slt     t2,t2,t0
```

Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	sub t1,t1,t2	addi t2,t2,2				
Decode		sub t1,t1,t2				
Execute						

How Instructions flow

Consider the following instruction sequence:

Progress in a three-stage pipeline

Once filled, at every clock there are 3 instructions at various stages of execution.

```
...
sub    t1,t1,t2
addi   t2,t2,2
andi   t0,t0,1
slt    t2,t2,t0
```

Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1			
Decode		sub t1,t1,t2	addi t2,t2,2			
Execute			sub t1,t1,t2			

How Instructions flow

Consider the following instruction sequence:

Progress in a three-stage pipeline

Once filled, at every clock there are 3 instructions at various stages of execution.

```
...
sub     t1,t1,t2
addi    t2,t2,2
andi    t0,t0,1
slt     t2,t2,t0
```

Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0		
Decode		sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0	
Execute			sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0

Source operands are fetched in this stage

How Instructions flow

Consider the following instruction sequence:

Progress in a three-stage pipeline

Once filled, at every clock there are 3 instructions at various stages of execution.

```
...
sub     t1,t1,t2
addi    t2,t2,2
andi    t0,t0,1
slt     t2,t2,t0
```

Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0		
Decode		sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0	
Execute			sub t1,t1,t2	addi t2,t2,2	andi t0,t0,1	slt t2,t2,t0

Source operands are fetched in this stage

Destination operands are updated in this stage

Simple Instruction flow

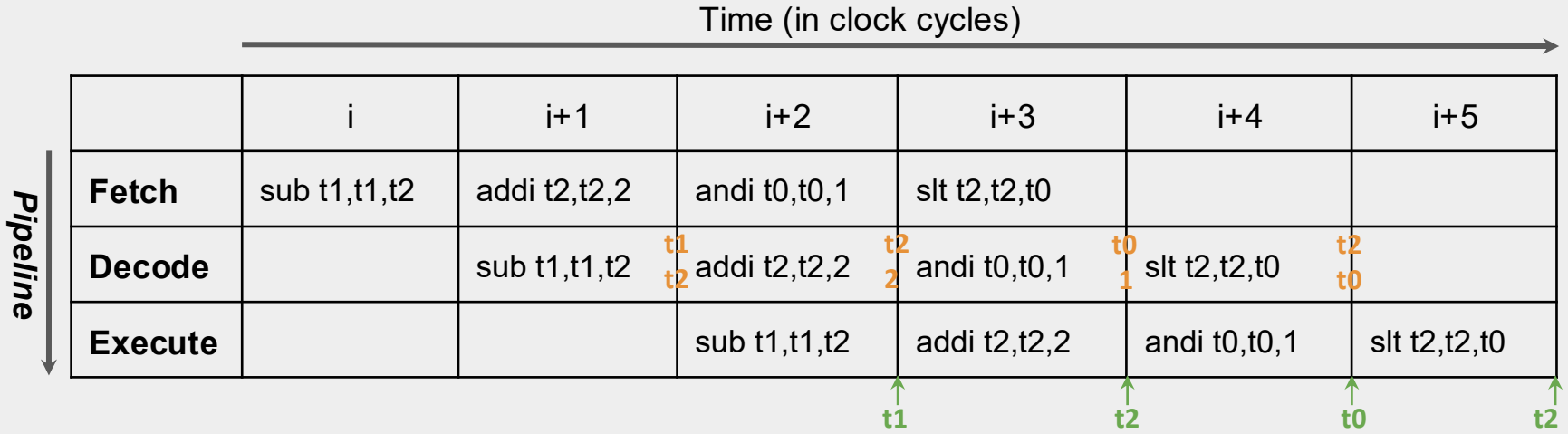
Consider the following instruction sequence:

Instruction becomes available at the end of the Fetch stage

Operands at the end of Decode

Destination and ALU flags are updated at the end of Execute

```
...
sub     t1,t1,t2
addi    t2,t2,2
andi    t0,t0,1
slt     t2,t2,t0
```



Splitting our CPU Datapath into 5 Stages

- Instruction Fetch
  - Reading the instruction from memory
- Instruction Decode
  - Splitting up the instruction, generating control signals, and reading the register file
  - Jump address calculation
- Execute
  - ALU Operation
  - Branch address calculation
- Memory
  - Reading/writing data memory
- Writeback
  - Writing the result to the register file

Instructions pass through *all 5 stages*, but different work happens in each stage depending on the instruction type.

Behavior of instructions starts to diverge

Summary Table

Instruction Type	IF	ID	EX	MEM	WB
`Arithmetic`	✓	✓	ALU	—	Write result
`LW`	✓	✓	Addr calc	Read mem	Write result
`SW`	✓	✓	Addr calc	Write mem	—
`Branch`	✓	✓	Compare + branch	—	—
`Jump`	✓	✓	Jump target*	—	—

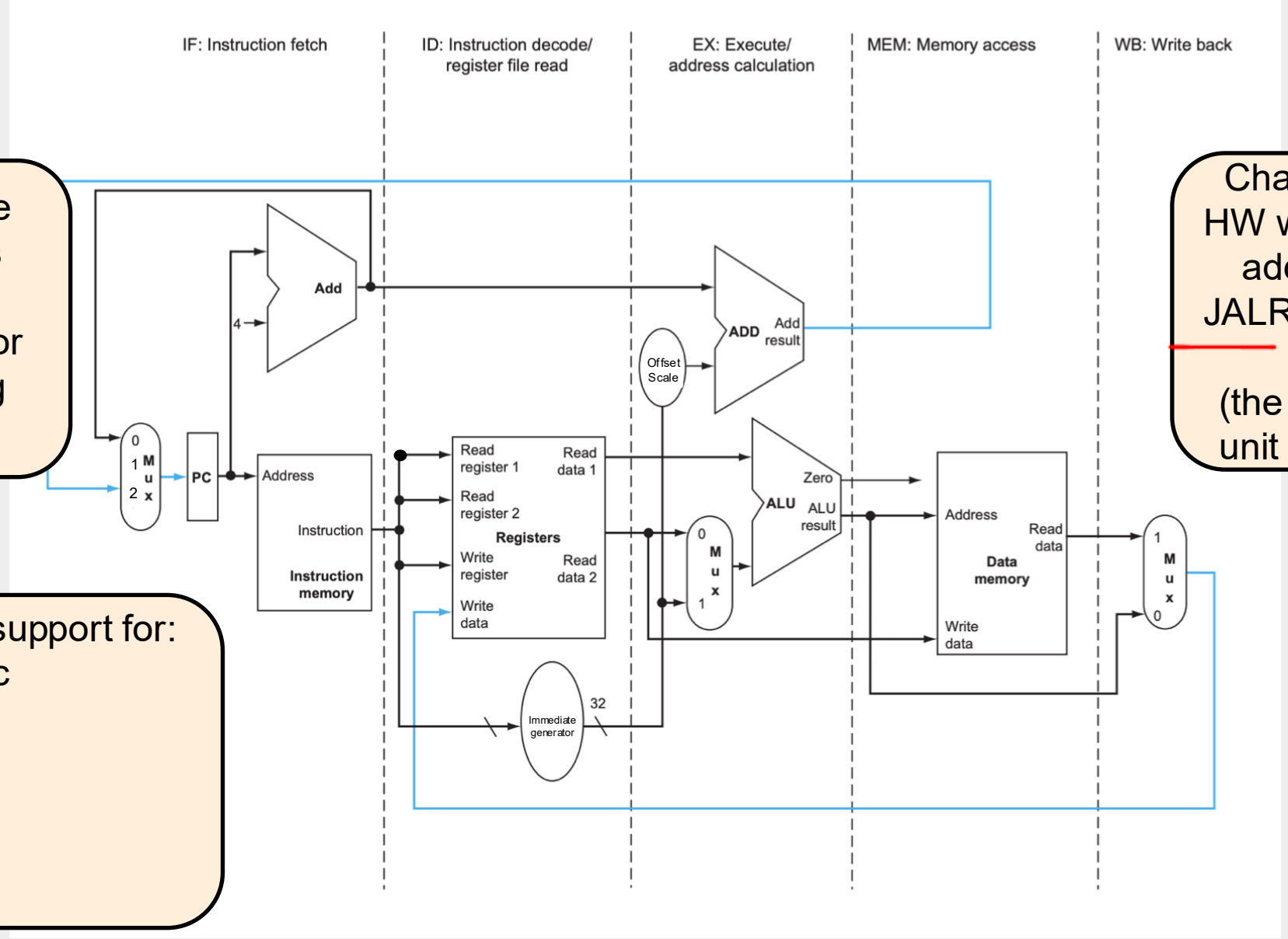
EX: Figure out what to do/action to take for specific instruction type

MEM: Only matters for loads/stores

WB: Only matters if we produce a value

Datapath Split into 5 Stages

This version of the full data path has some details abstracted away for readability/tracing purposes.



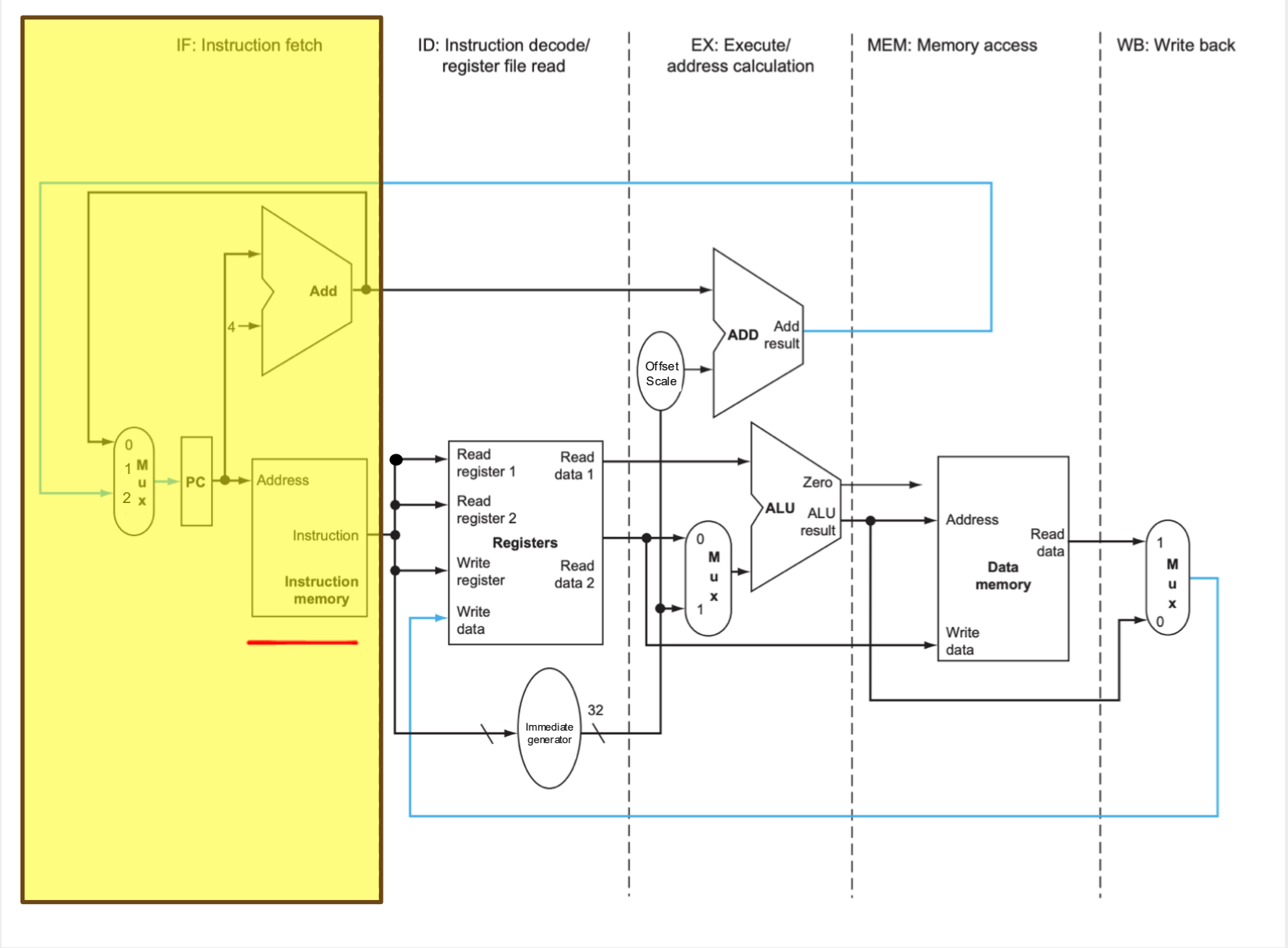
Challenge Q: What HW would need to be added to support JALR? Other types of branches?
(the full ALU control unit also missing…)

This datapath has support for:

- R-type arithmetic
- I-type arithmetic
- Loads
- Stores
- BEQ
- JAL

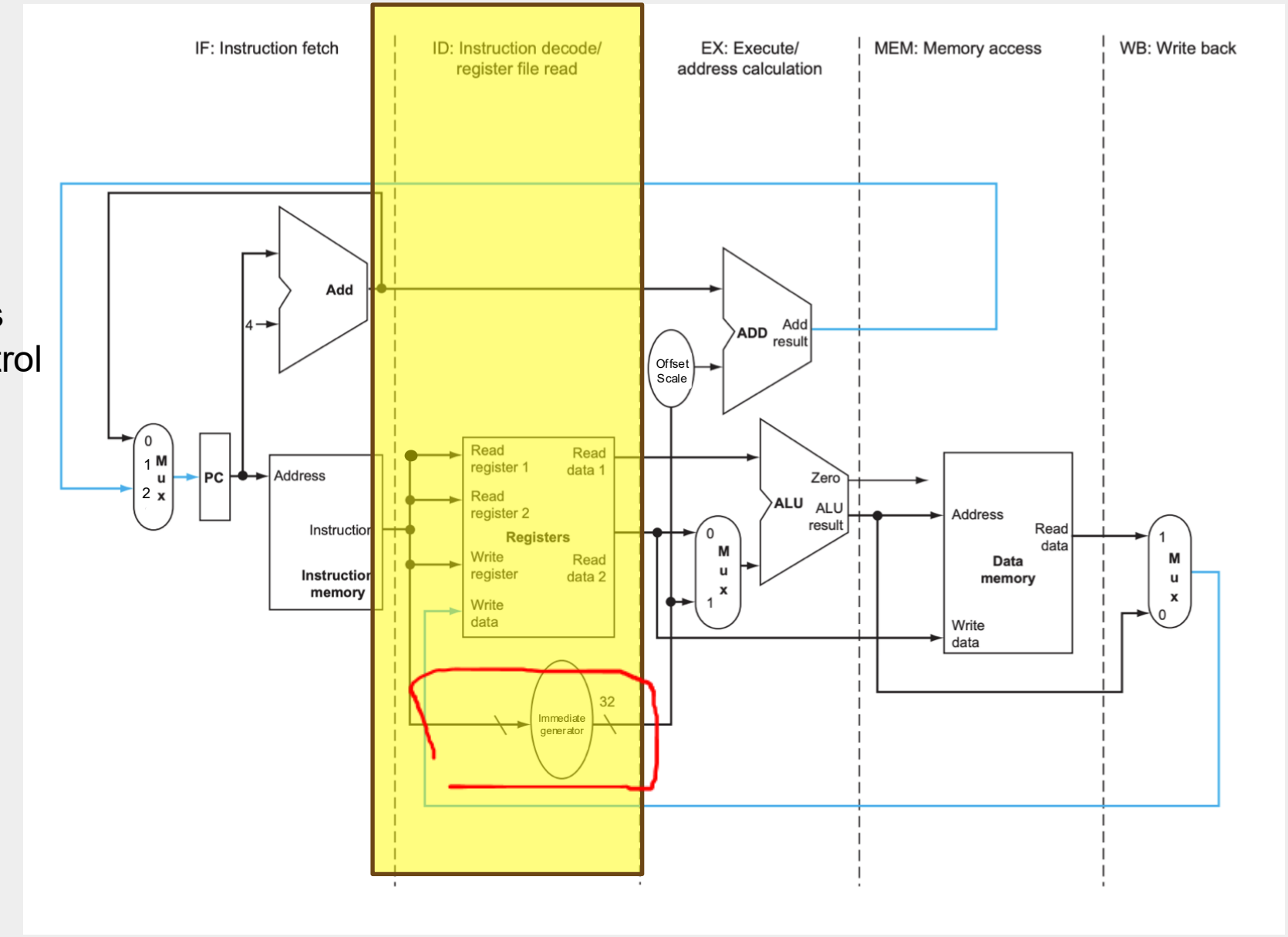
Instruction Fetch

Reading the instruction from memory



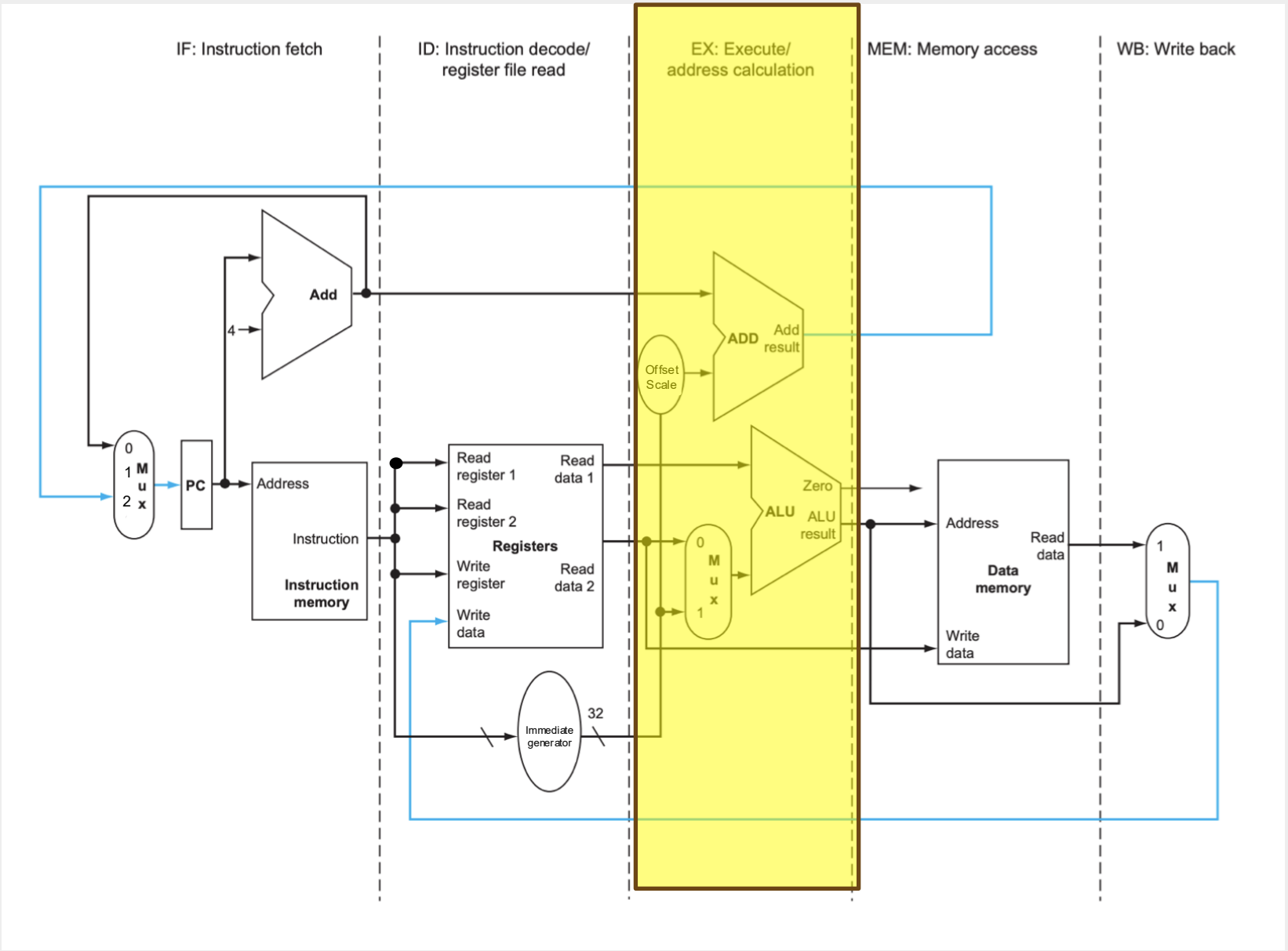
Instruction Decode

- Splitting up the instruction fields
- Generating control signals
- Reading the register file
- Jump address calculation



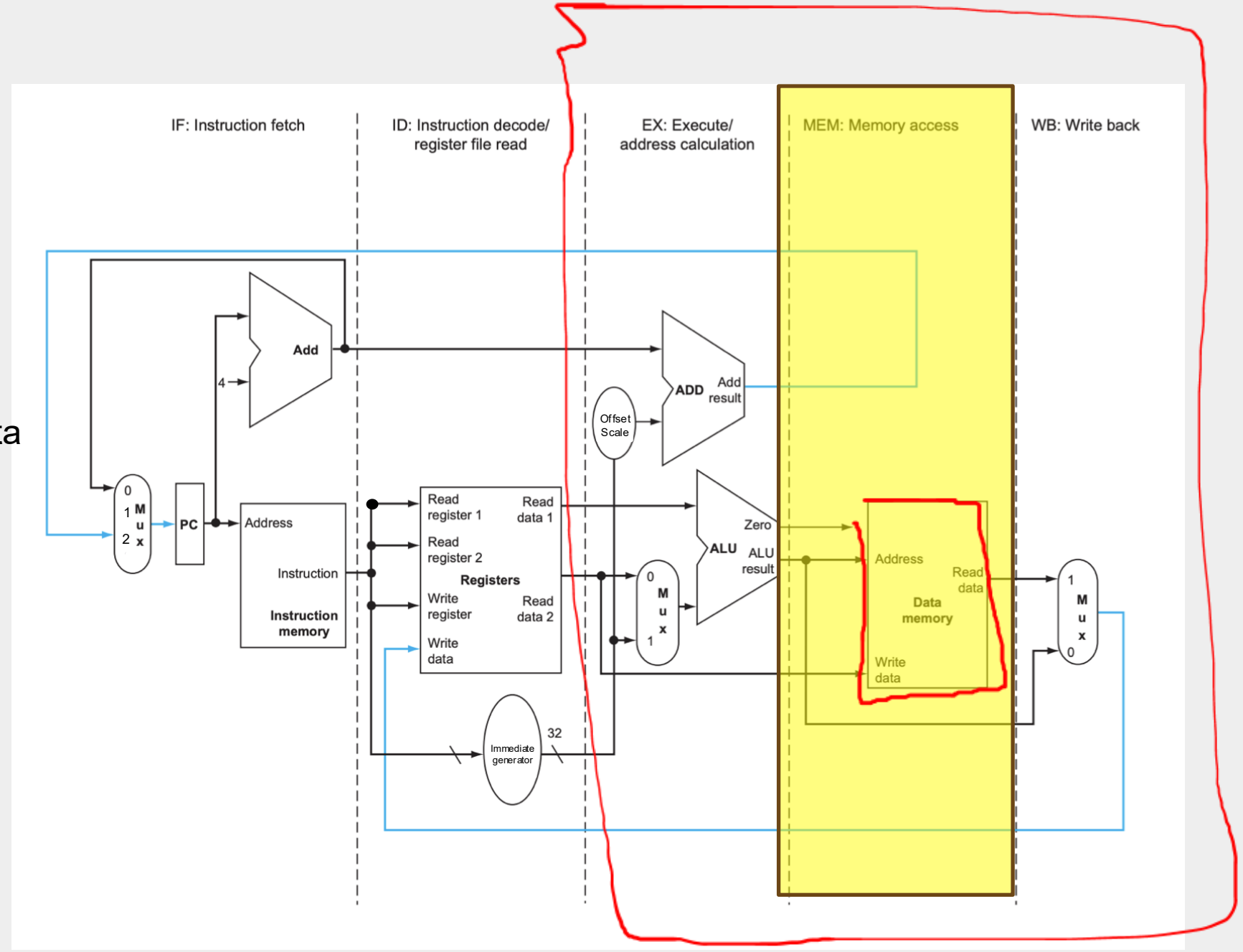
Execute

- ALU Operation
- Branch address calculation



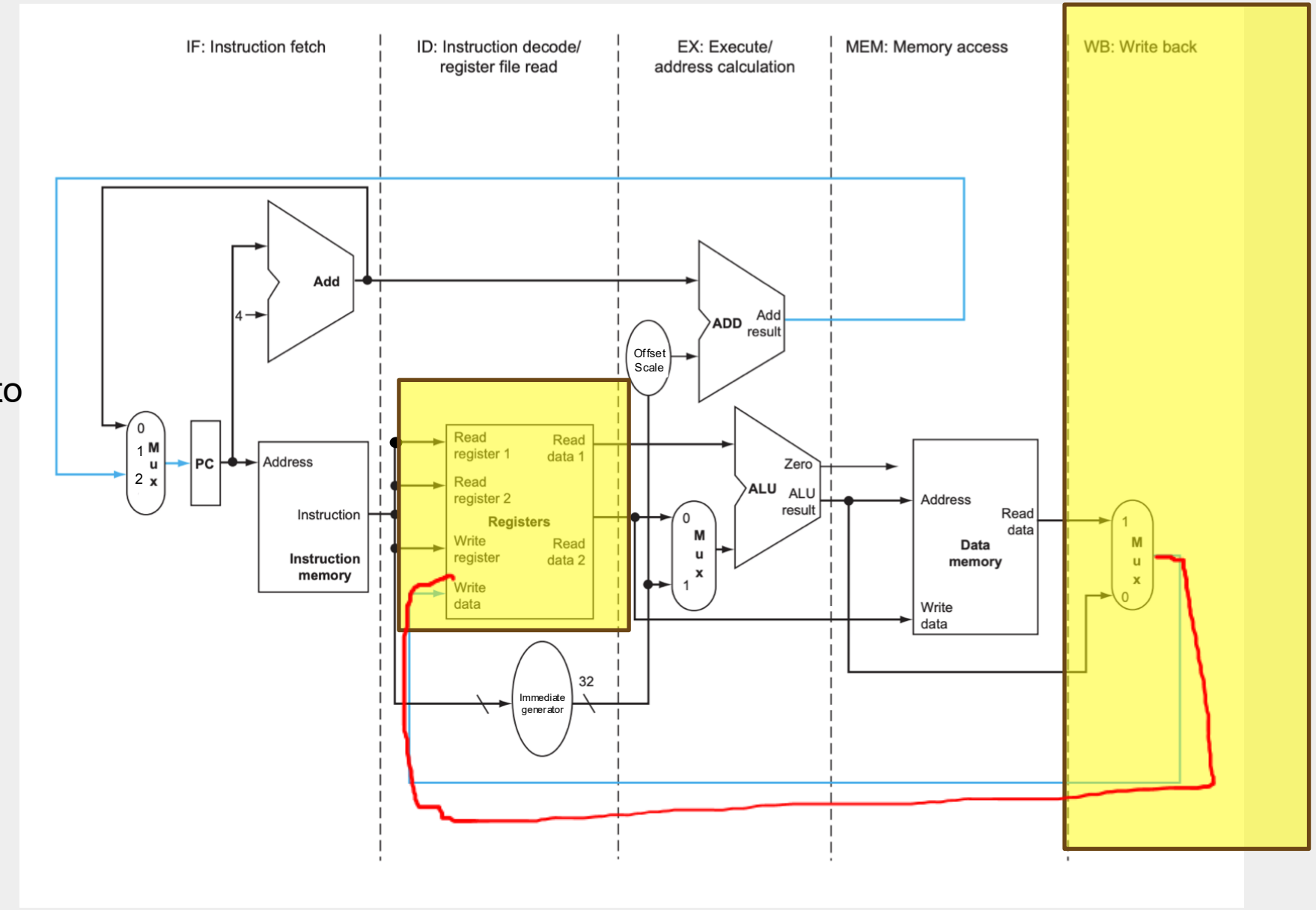
Memory

Reading/writing data memory

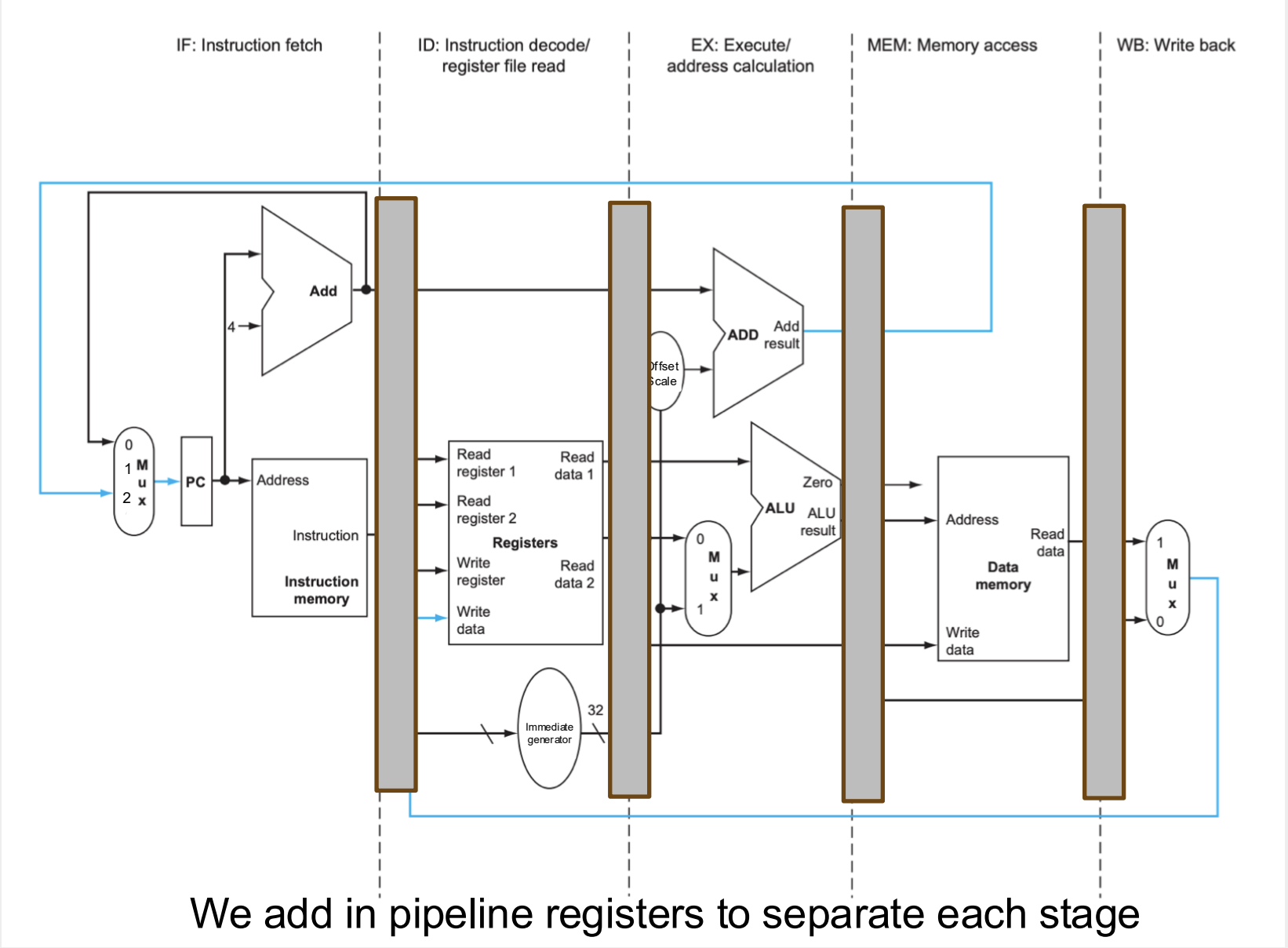


Writeback

Writing the result to the register file



5-Stage Pipeline



We add in pipeline registers to separate each stage