



A Simple Processor

1. A simple Instruction Set Architecture
2. A simple microarchitecture (implementation):
Data Path and Control Logic

Software

Program, Application

Programming Language

Compiler/Interpreter

Operating System

Instruction Set Architecture

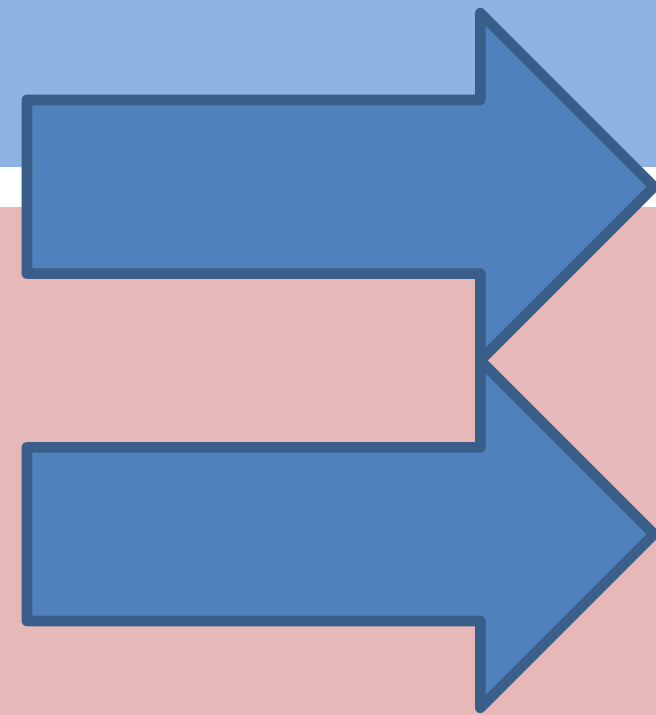
Microarchitecture

Digital Logic

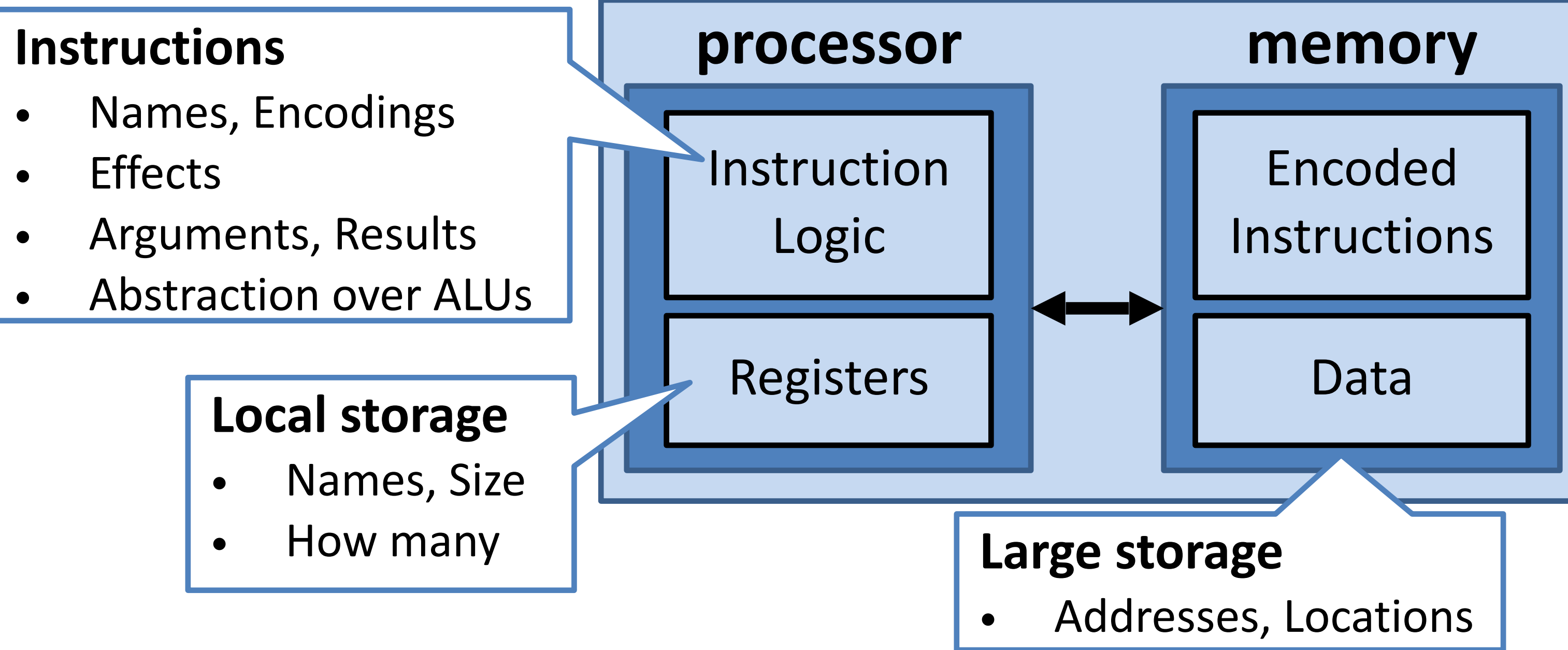
Devices (transistors, etc.)

Solid-State Physics

Hardware

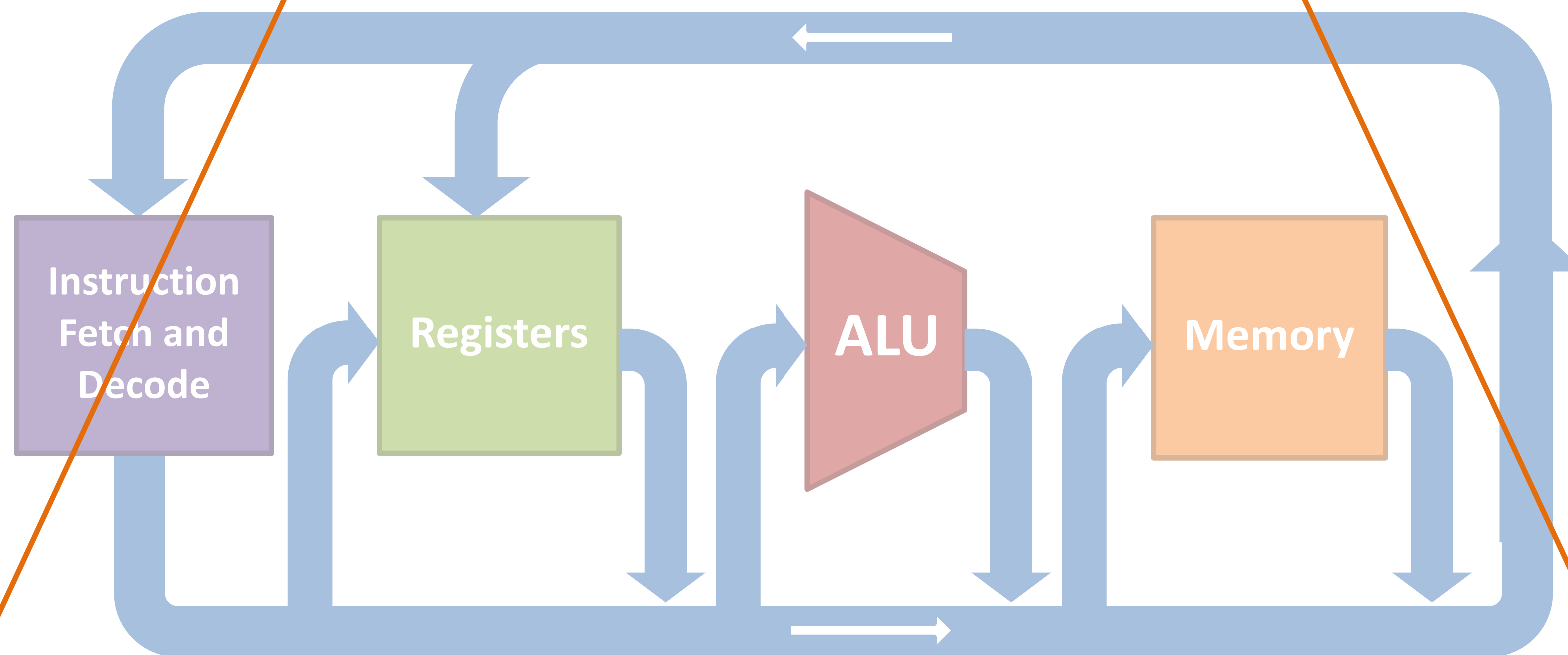


Instruction Set Architecture (HW/SW **Interface**)



Computer

Microarchitecture (Implementation of ISA)



HW ISA

An example, made-up instruction set architecture for CS240

Word size = 16 bits (smaller than most real CPUs)

- Register size = 16 bits.
- ALU computes on 16-bit values.

Memory is byte-addressable, accesses full words (byte pairs)

16 registers: R0 - R15

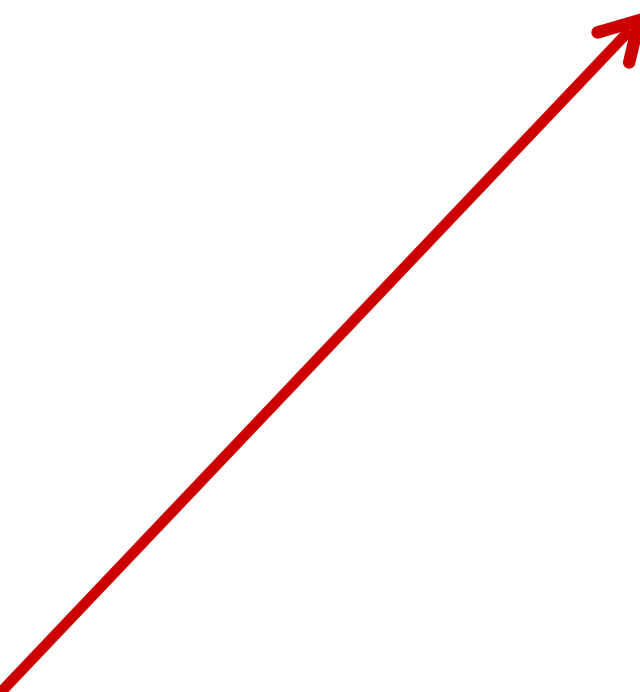
- R0 always holds hardcoded 0
- R1 always holds hardcoded 1
- R2 – R15: general purpose

Instructions are 1 word in size.

Separate *instruction memory*.

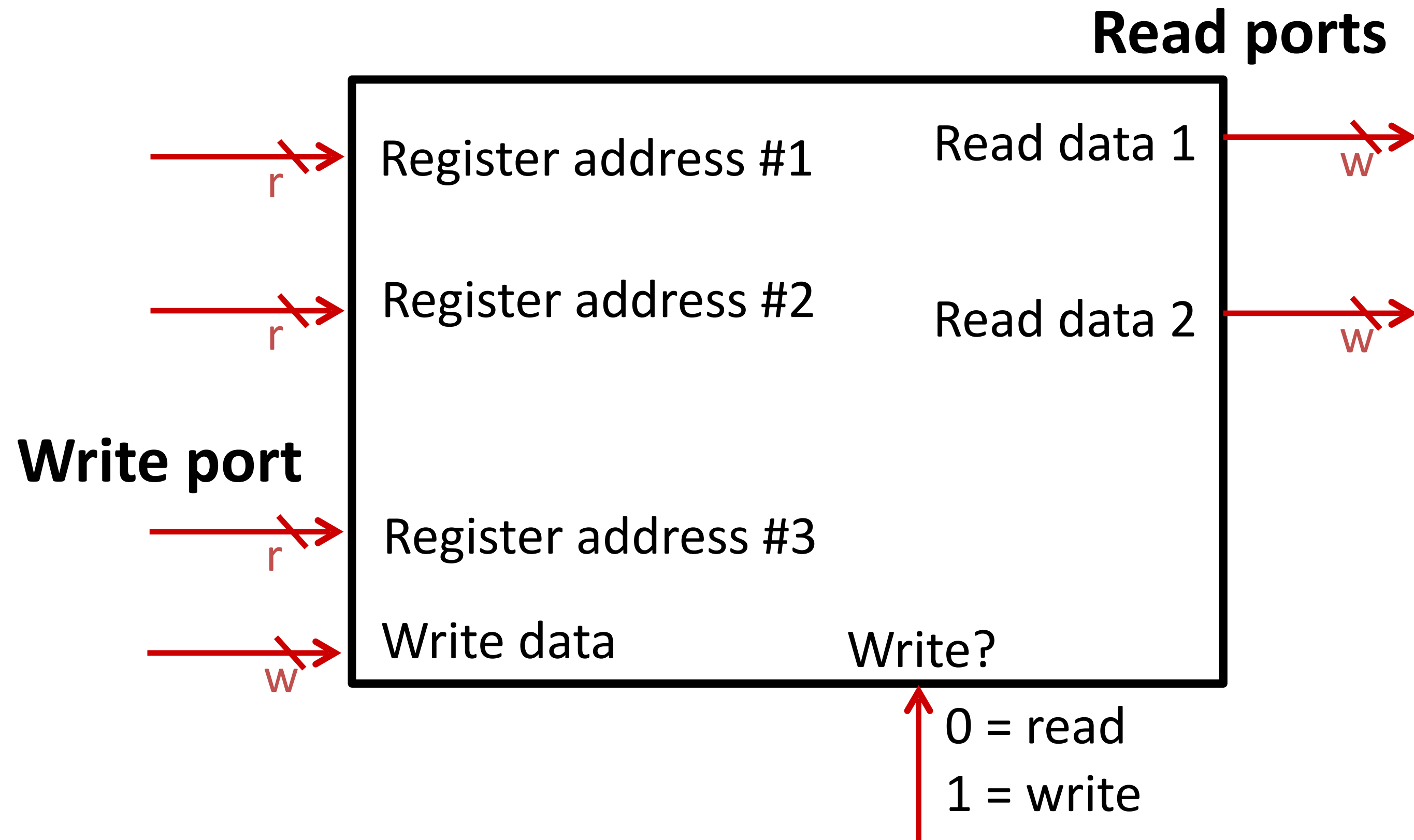
Program Counter (PC) register

- holds address of next instruction to execute.



Address	Contents
0	First instruction, low-order byte
1	First instruction, high-order byte
2	Second instruction, low-order byte
...	...

HW ISA **R: Register File**

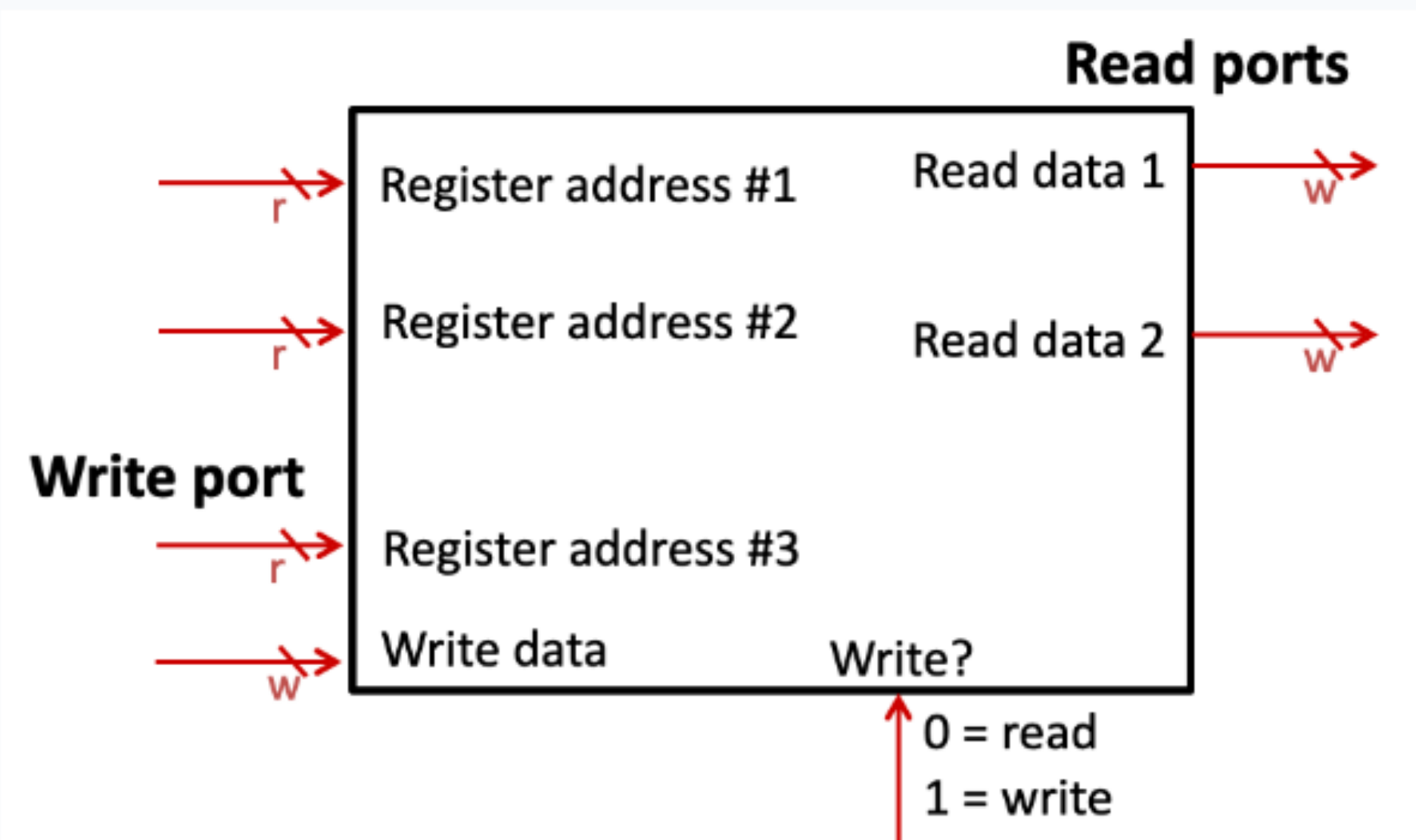


Word size = 16 bits, # registers = 16

ex

r = ?
w = ?

Bus size for registers



Word size = 16 bits, # registers = 16

ex

r = ?
w = ?

r = 8, w = 8

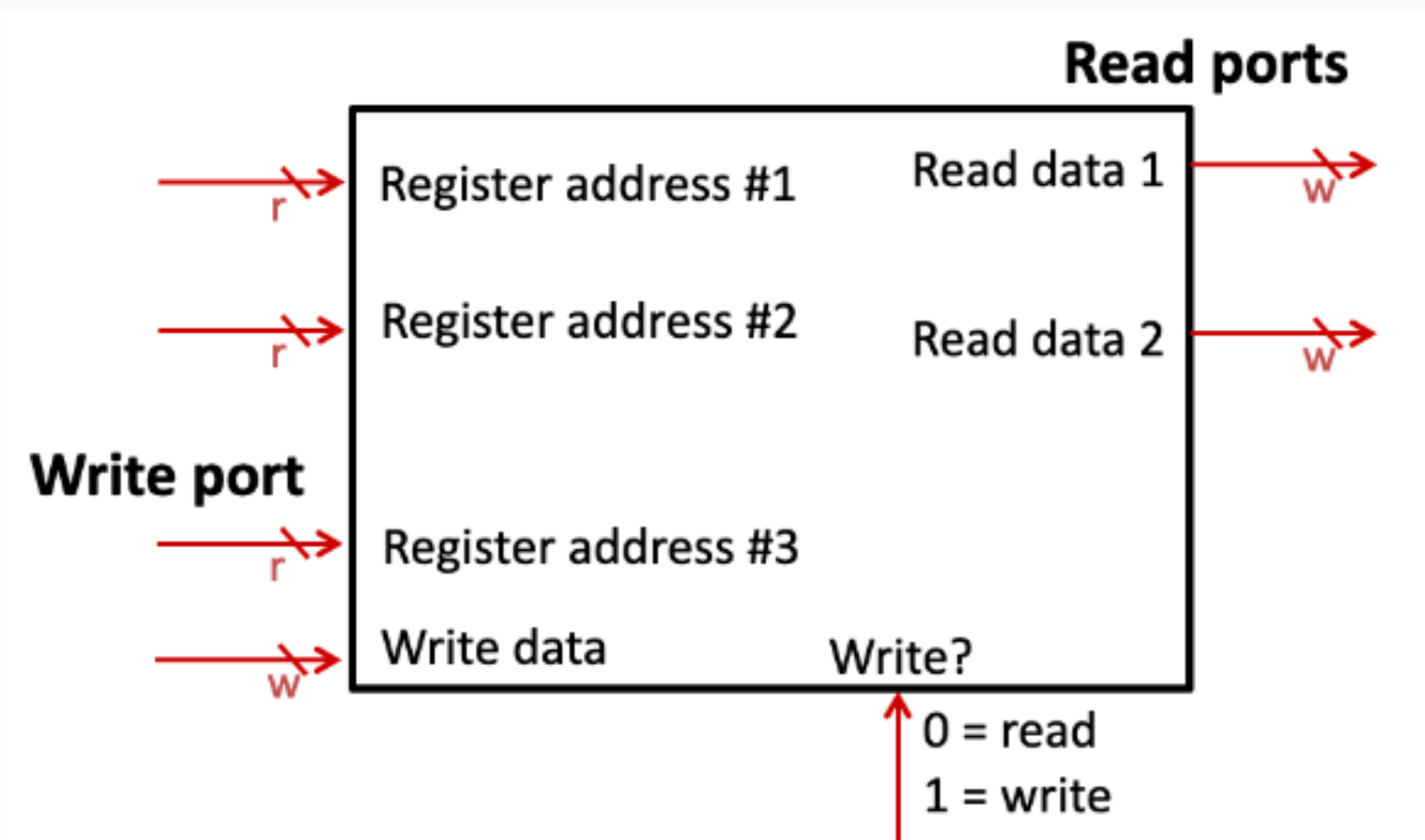
r = 16, w = 16

r = 4, w = 16

r = 16, w = 4

None of the above

Bus size for registers

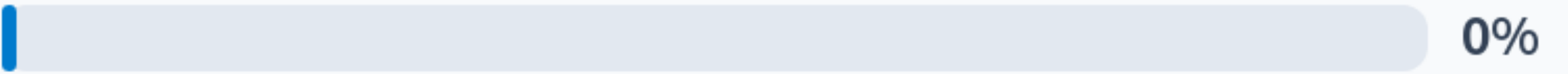


Word size = 16 bits, # registers = 16

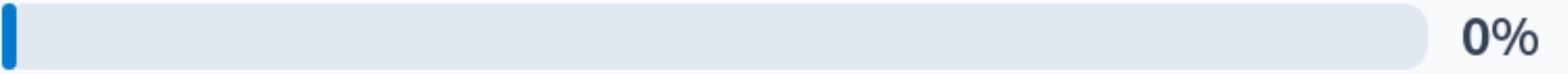
ex

r = ?
w = ?

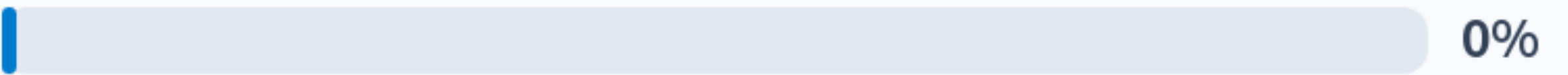
r = 8, w = 8



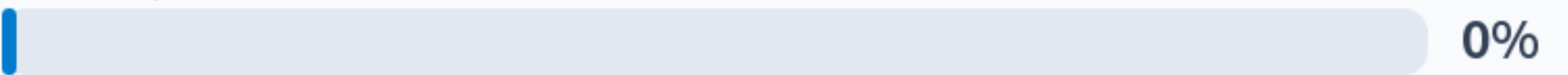
r = 16, w = 16



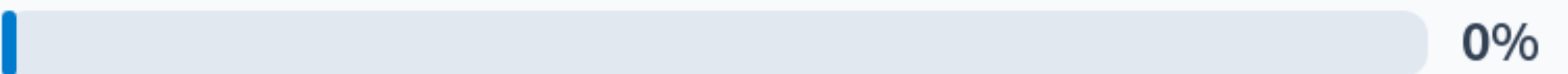
r = 4, w = 16



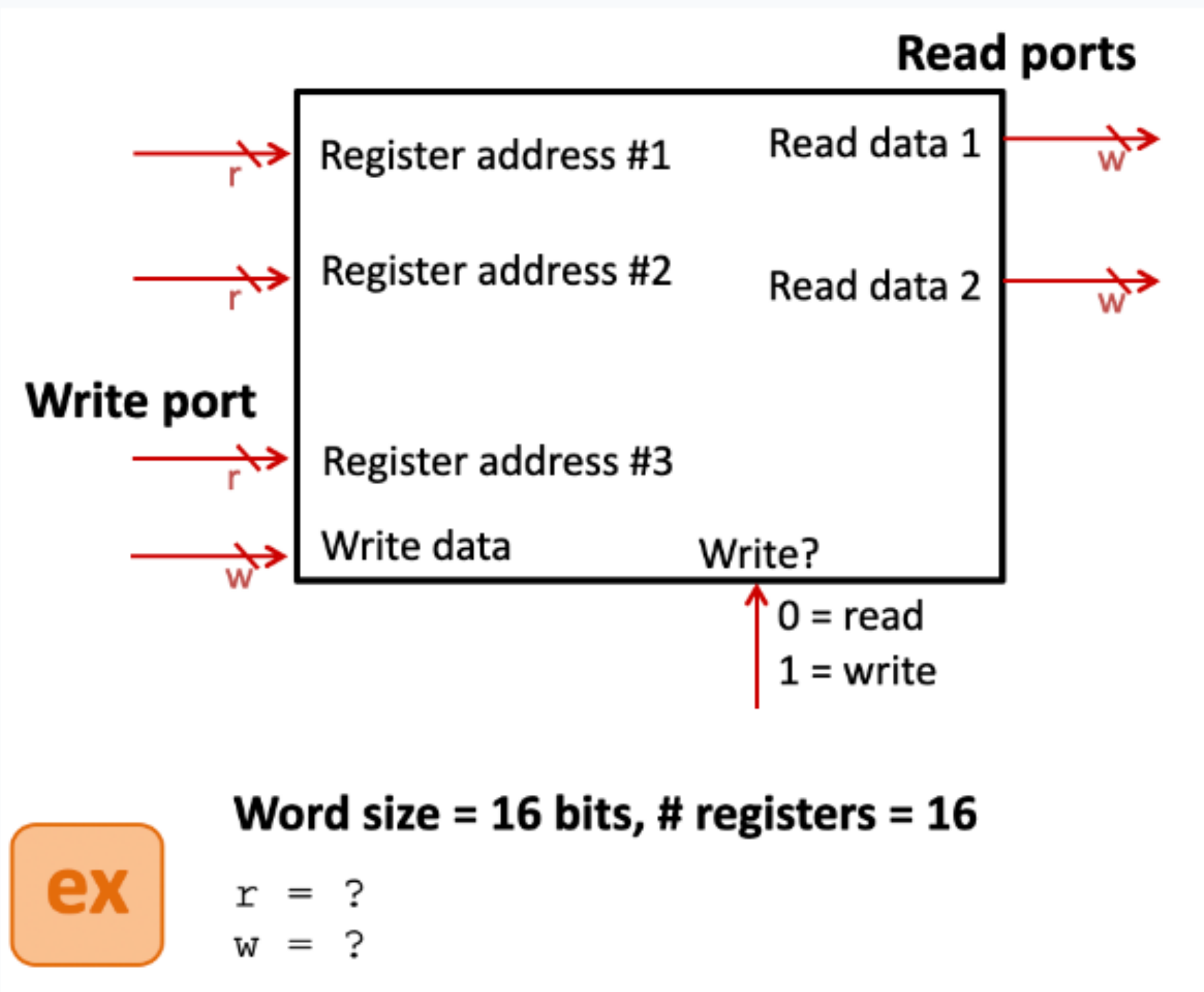
r = 16, w = 4



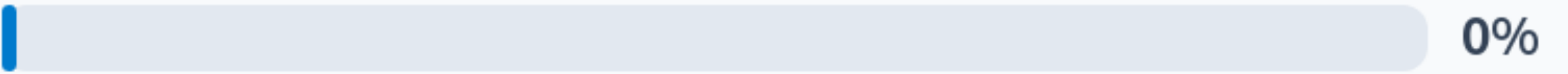
None of the above



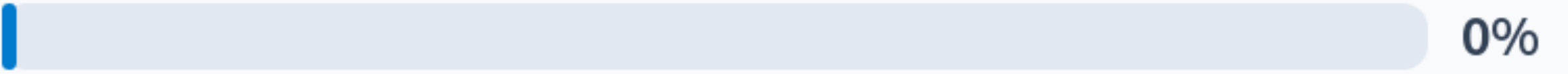
Bus size for registers



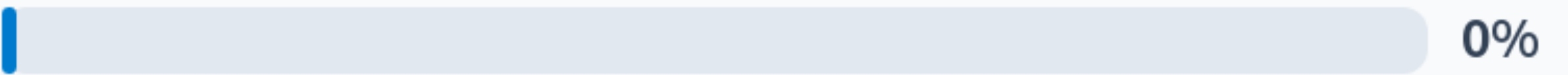
r = 8, w = 8



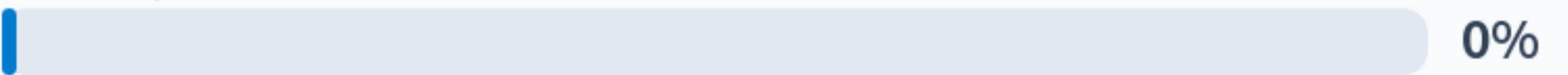
r = 16, w = 16



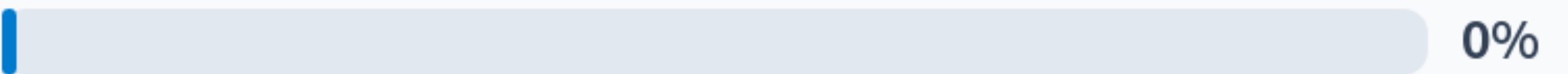
r = 4, w = 16



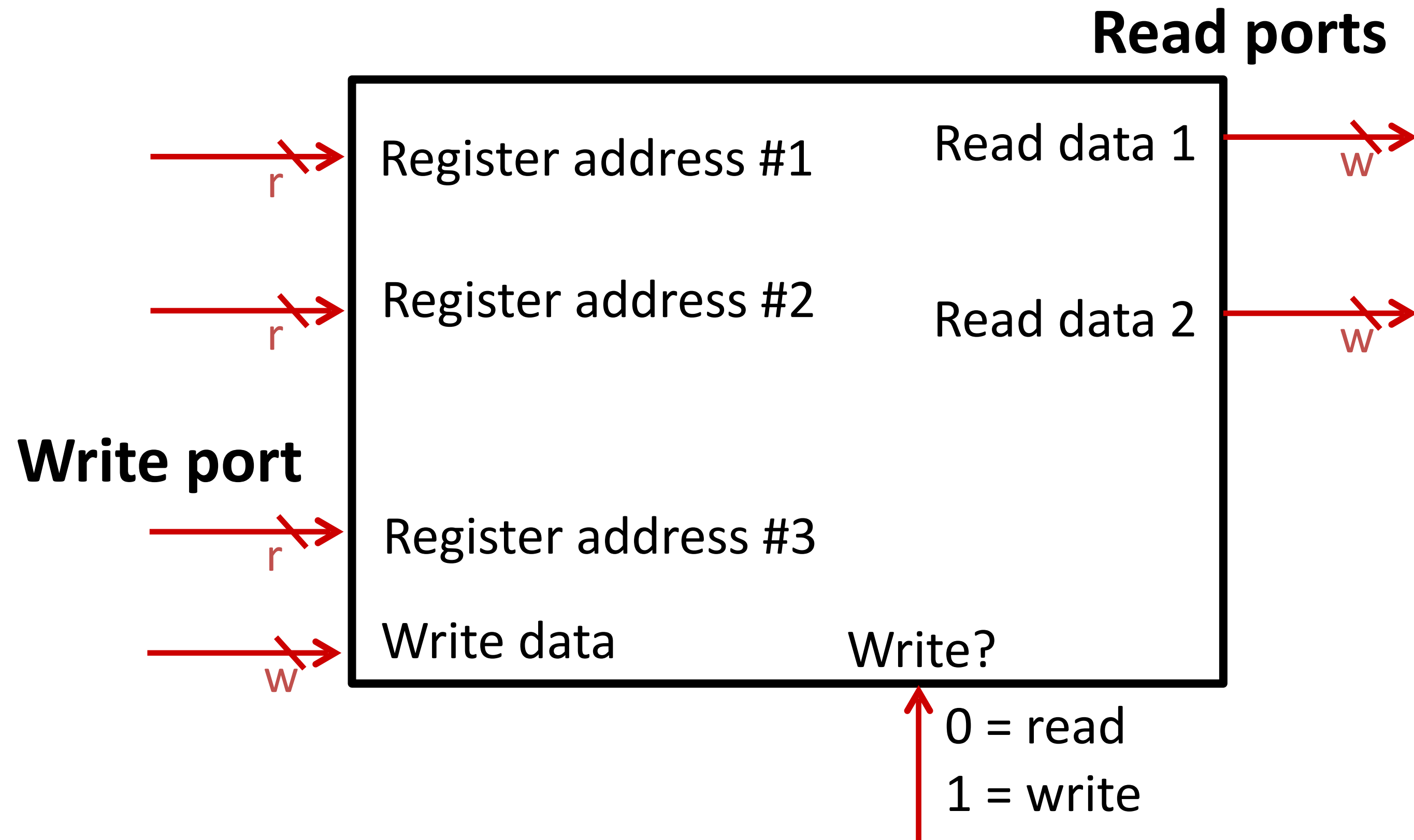
r = 16, w = 4



None of the above



HW ISA R: Register File



Abstraction!

We'll think of the register file like this:

R0 always holds hardcoded 0
R1 always holds hardcoded 1

How should we write this?

Reg	Contents
R0	
R1	
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	

ex

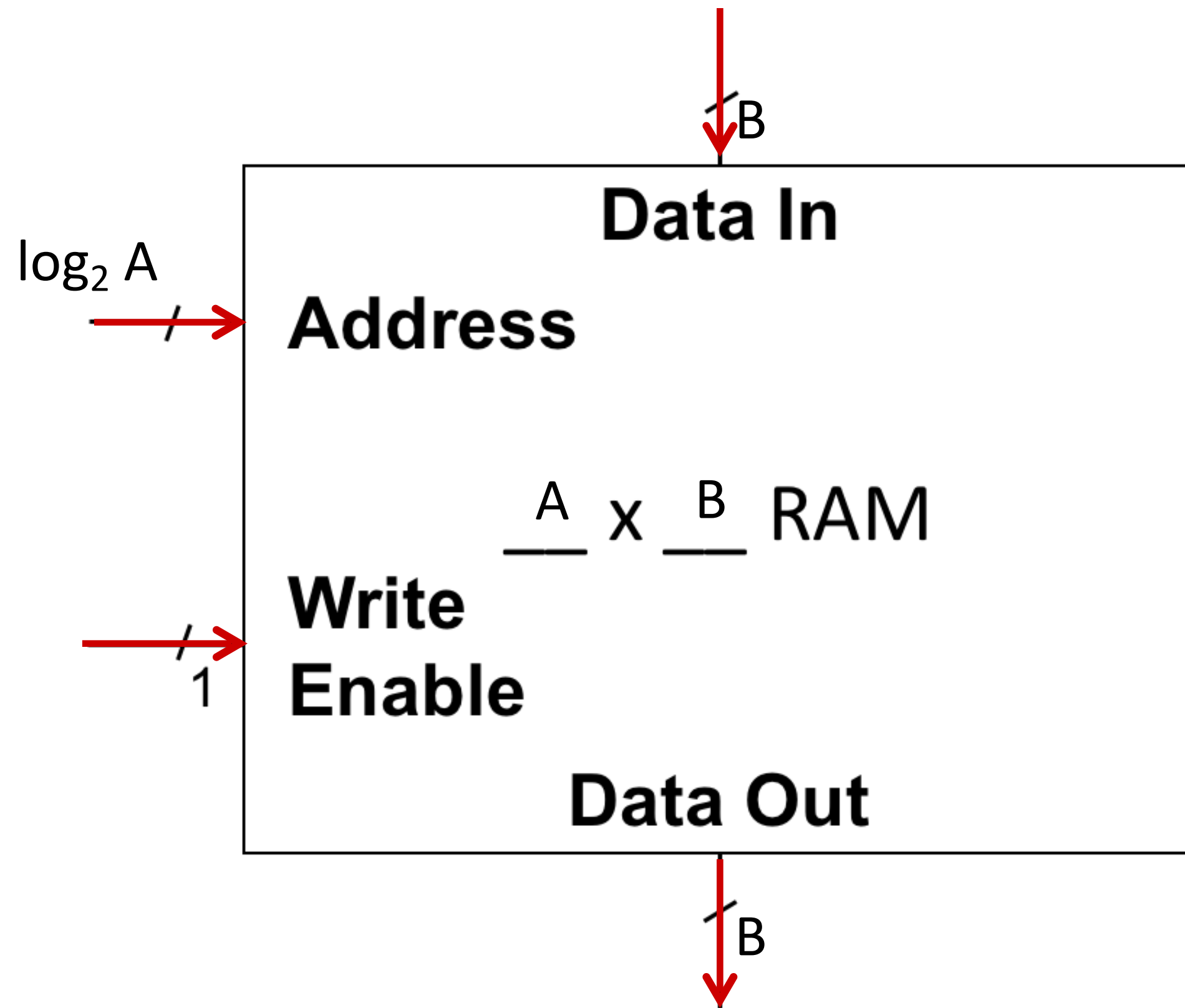
Word size = 16 bits, # registers = 16

r = ?
w = ?

HW ISA M: Data Memory



We'll think of the data memory like this:



Address	Contents	
0x0 – 0x1		
0x2 – 0x3		
0x4 – 0x5		
0x6 – 0x7		
0x8 – 0x9		
0xA – 0xB		
0xC – 0xD		
...		

Memory is byte-addressable, accesses full words (16 bits)

A = 4
B = 16

HW ISA **IM**: Instruction Memory

Instructions are 1 word in size.

Separate *instruction memory*.

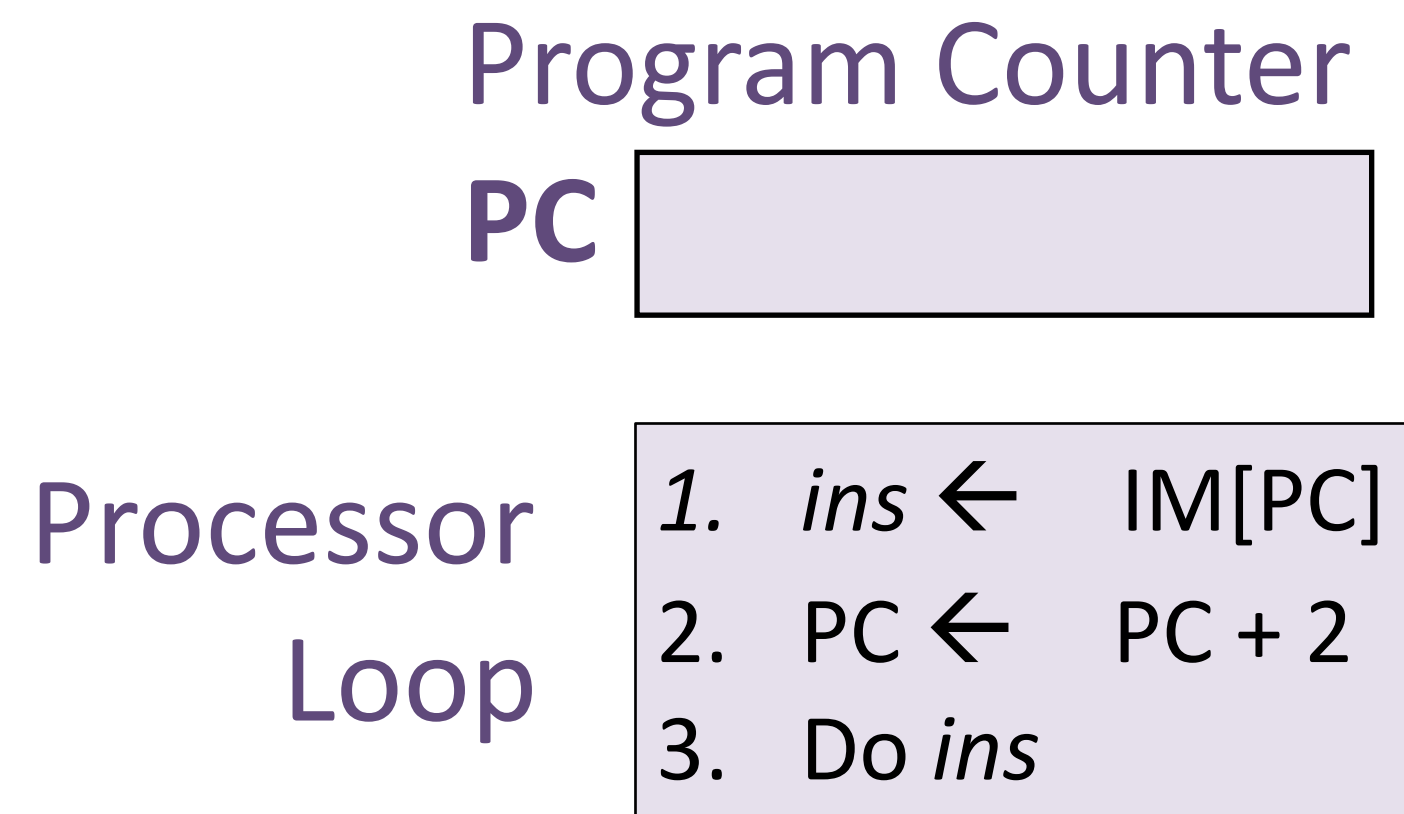
Program Counter (PC) register

- holds address of next instruction to execute.



Abstraction!

We'll think of the instruction memory like this:



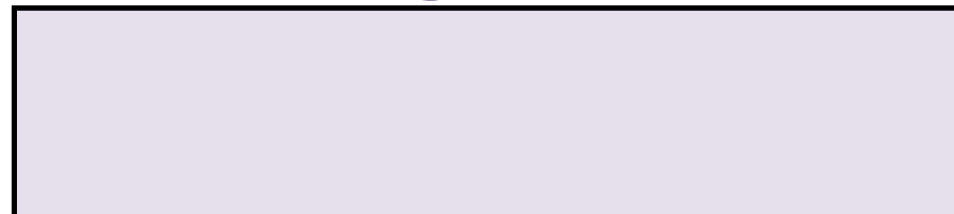
Address	Contents
0x0 – 0x1	
0x2 – 0x3	
0x4 – 0x5	
0x6 – 0x7	
0x8 – 0x9	
...	

HW ISA



Abstract Machine

PC: Program Counter



Processor Loop

1. $ins \leftarrow IM[PC]$
2. $PC \leftarrow PC + 2$
3. Do ins

M: Data Memory

Address	Contents	
0x0 – 0x1		
0x2 – 0x3		
0x4 – 0x5		
0x6 – 0x7		
0x8 – 0x9		
0xA – 0xB		
0xC – 0xD		
...		

IM: Instruction Memory

Address	Contents	
0x0 – 0x1		
0x2 – 0x3		
0x4 – 0x5		
0x6 – 0x7		
0x8 – 0x9		
...		

R: Register File

Reg	Contents
R0	0x0000
R1	0x0001
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	

HW ISA Instructions

MSB **16-bit Encoding** LSB

Assembly Syntax	Meaning (R = register file, M = data memory)	Opcode	Rs	Rt	Rd
ADD <i>Rs, Rt, Rd</i>	$R[d] \leftarrow R[s] + R[t]$	0010	<i>s</i>	<i>t</i>	<i>d</i>
SUB <i>Rs, Rt, Rd</i>	$R[d] \leftarrow R[s] - R[t]$	0011	<i>s</i>	<i>t</i>	<i>d</i>
AND <i>Rs, Rt, Rd</i>	$R[d] \leftarrow R[s] \& R[t]$	0100	<i>s</i>	<i>t</i>	<i>d</i>
OR <i>Rs, Rt, Rd</i>	$R[d] \leftarrow R[s] R[t]$	0101	<i>s</i>	<i>t</i>	<i>d</i>
LW <i>Rt, offset(Rs)</i>	$R[t] \leftarrow M[R[s] + offset]$	0000	<i>s</i>	<i>t</i>	<i>offset</i>
SW <i>Rt, offset(Rs)</i>	$M[R[s] + offset] \leftarrow R[t]$	0001	<i>s</i>	<i>t</i>	<i>offset</i>
BEQ <i>Rs, Rt, offset</i>	If $R[s] == R[t]$ then $PC \leftarrow PC + 2 + offset * 2$	0111	<i>s</i>	<i>t</i>	<i>offset</i>
JMP <i>offset</i>	$PC \leftarrow offset * 2$	1000	<i>offset</i>		
HALT	Stops program execution	1111			

JMP offset is
unsigned
All others are
signed

HW ISA



Exercise #0 (example)

Fill in the rest of the machine state based on this initial state

PC: Program Counter

Processor Loop

1. $ins \leftarrow IM[PC]$
2. $PC \leftarrow PC + 2$
3. Do ins

M: Data Memory

Address	Contents	
0x0 – 0x1	0xEB	0xCA
0x2 – 0x3	0xBD	0x56
0x4 – 0x5		
0x6 – 0x7		
0x8 – 0x9		
0xA – 0xB		
0xC – 0xD		
...		

IM: Instruction Memory

Address	Contents
0x0 – 0x1	ADD R1, R1, R2
0x2 – 0x3	SW R2, 4(R0)
0x4 – 0x5	HALT
0x6 – 0x7	
0x8 – 0x9	
...	

R: Register File

Reg	Contents
R0	0x0000
R1	0x0001
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	

Execution Table for *Exercise #0* (shows step-by-step execution)

Solutions



PC	Instr	State Changes
0x0	ADD R1, R1, R2	$R[2] \leftarrow R[1] + R[1] = 1 + 1 = 0x0002$; $PC \leftarrow PC + 2 = 0 + 2 = 2$
0x2	SW R2, 4(R0)	$M[R[0] + 4] = M[4] \leftarrow R[2] = 0x0002$; $PC \leftarrow PC + 2 = 2 + 2 = 4$
0x4	HALT	<i>Program execution stops</i>

These bytes will be stored in so-called **Little Endian** order when we store them to memory **M**.

That is, the byte pair 0x02 will be stored in the “little” end of the word—the lower address of the pair of addresses that store the word. 0x00 will be stored at the higher address.

HW ISA



Solutions

Exercise #0 (example)

Fill in the rest of the machine state based on this initial state

PC: Program Counter

Processor Loop

1. $ins \leftarrow IM[PC]$
2. $PC \leftarrow PC + 2$
3. Do ins

M: Data Memory

Address	Contents	
0x0 – 0x1	0xEB	0xCA
0x2 – 0x3	0xBD	0x56
0x4 – 0x5	0x02	0x00
0x6 – 0x7		
0x8 – 0x9		
0xA – 0xB		
0xC – 0xD		
...		

IM: Instruction Memory

Address	Contents
0x0 – 0x1	ADD R1, R1, R2
0x2 – 0x3	SW R2, 4(R0)
0x4 – 0x5	HALT
0x6 – 0x7	
0x8 – 0x9	
...	

R: Register File

Reg	Contents
R0	0x0000
R1	0x0001
R2	0x0002
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	

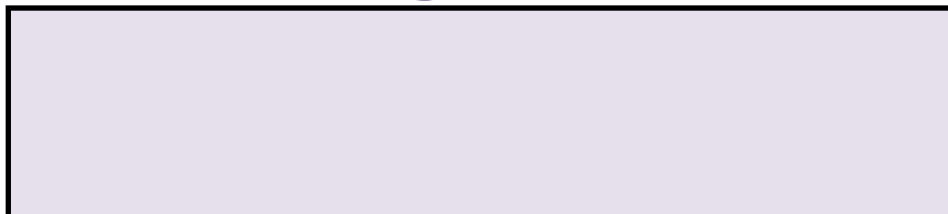
HW ISA



Exercise #1:

Fill in the rest of the machine state based on this initial state

PC: Program Counter



Processor Loop

1. $ins \leftarrow IM[PC]$
2. $PC \leftarrow PC + 2$
3. Do ins

M: Data Memory

Address	Contents	
0x0 – 0x1	0xEB	0xCA
0x2 – 0x3	0xBD	0x56
0x4 – 0x5		
0x6 – 0x7		
0x8 – 0x9		
0xA – 0xB		
0xC – 0xD		
...		

IM: Instruction Memory

Address	Contents
0x0 – 0x1	LW R3, 0(R0)
0x2 – 0x3	LW R4, 2(R0)
0x4 – 0x5	AND R3, R4, R5
0x6 – 0x7	SW R5, 4(R0)
0x8 – 0x9	HALT
...	

R: Register File

Reg	Contents
R0	0x0000
R1	0x0001
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	

Execution Table for *Exercise #1* (shows step-by-step execution)



PC	Instr	State Changes
0x0	LW R3 0(R0)	

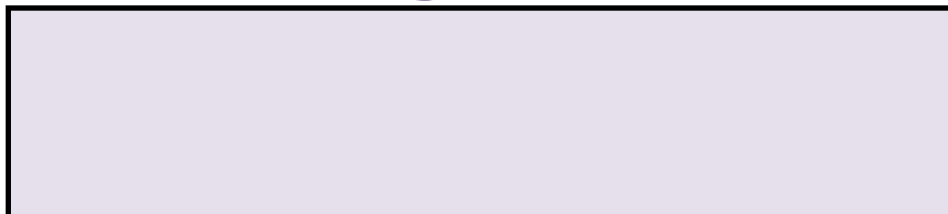
HW ISA



Exercise #2:

Fill in the rest of the machine state based on this initial state

PC: Program Counter



Processor Loop

1. $ins \leftarrow IM[PC]$
2. $PC \leftarrow PC + 2$
3. Do ins

M: Data Memory

Address	Contents	
0x0 – 0x1	0xEB	0xCA
0x2 – 0x3	0xBD	0x56
0x4 – 0x5		
0x6 – 0x7		
0x8 – 0x9		
0xA – 0xB		
0xC – 0xD		
...		

IM: Instruction Memory

Address	Contents
0x0 – 0x1	SUB R8, R8, R8
0x2 – 0x3	BEQ R9, R0, 3
0x4 – 0x5	ADD R10, R8, R8
0x6 – 0x7	SUB R9, R1, R9
0x8 – 0x9	JMP 1
0xA – 0xB	HALT
...	

R: Register File

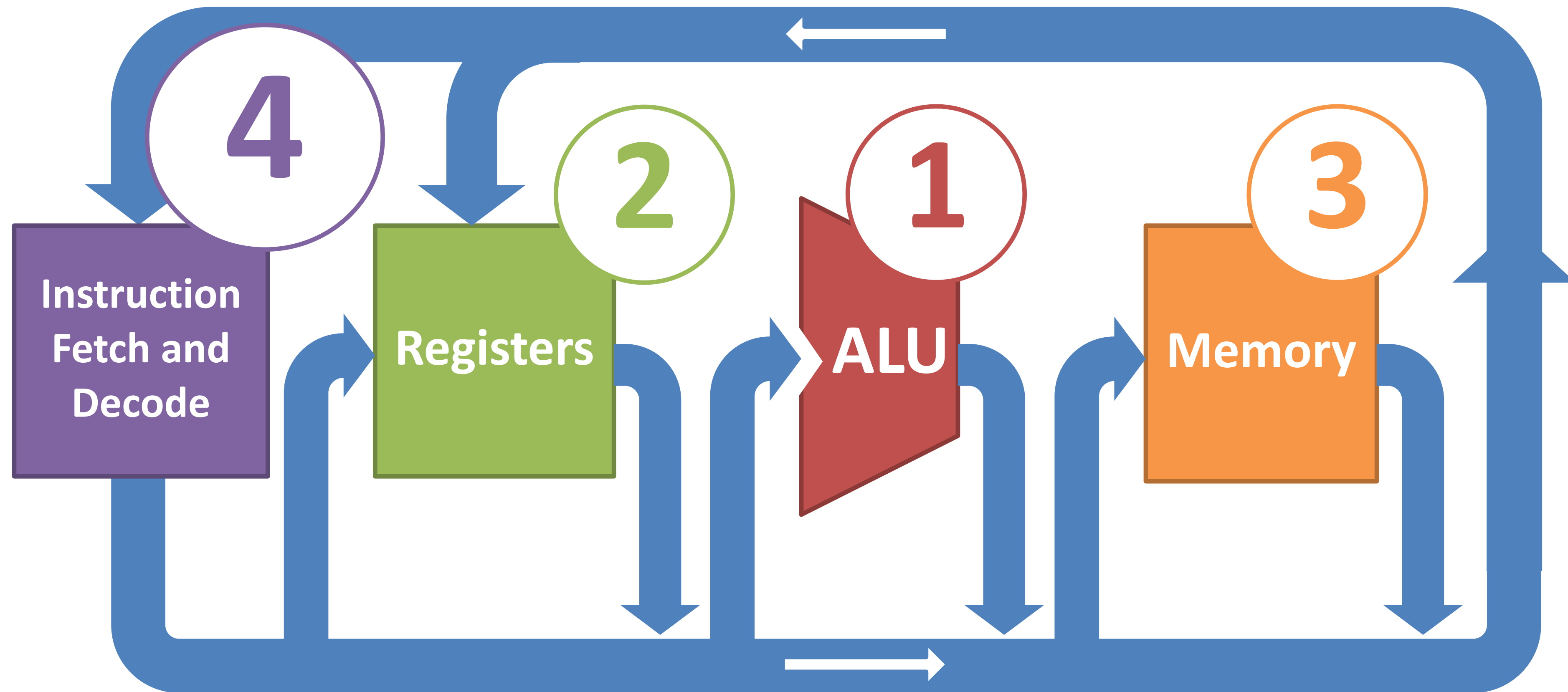
Reg	Contents (time: →)
R0	0x0000
R1	0x0001
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	0x0002
R10	0x0003
R11	
R12	
R13	
R14	
R15	

Execution Table for *Exercise #2* (shows step-by-step execution)



PC	Instr	State Changes
0x0	SUB R8, R8, R8	

HW ARCH **microarchitecture**



One possible hardware implementation of the HW ISA

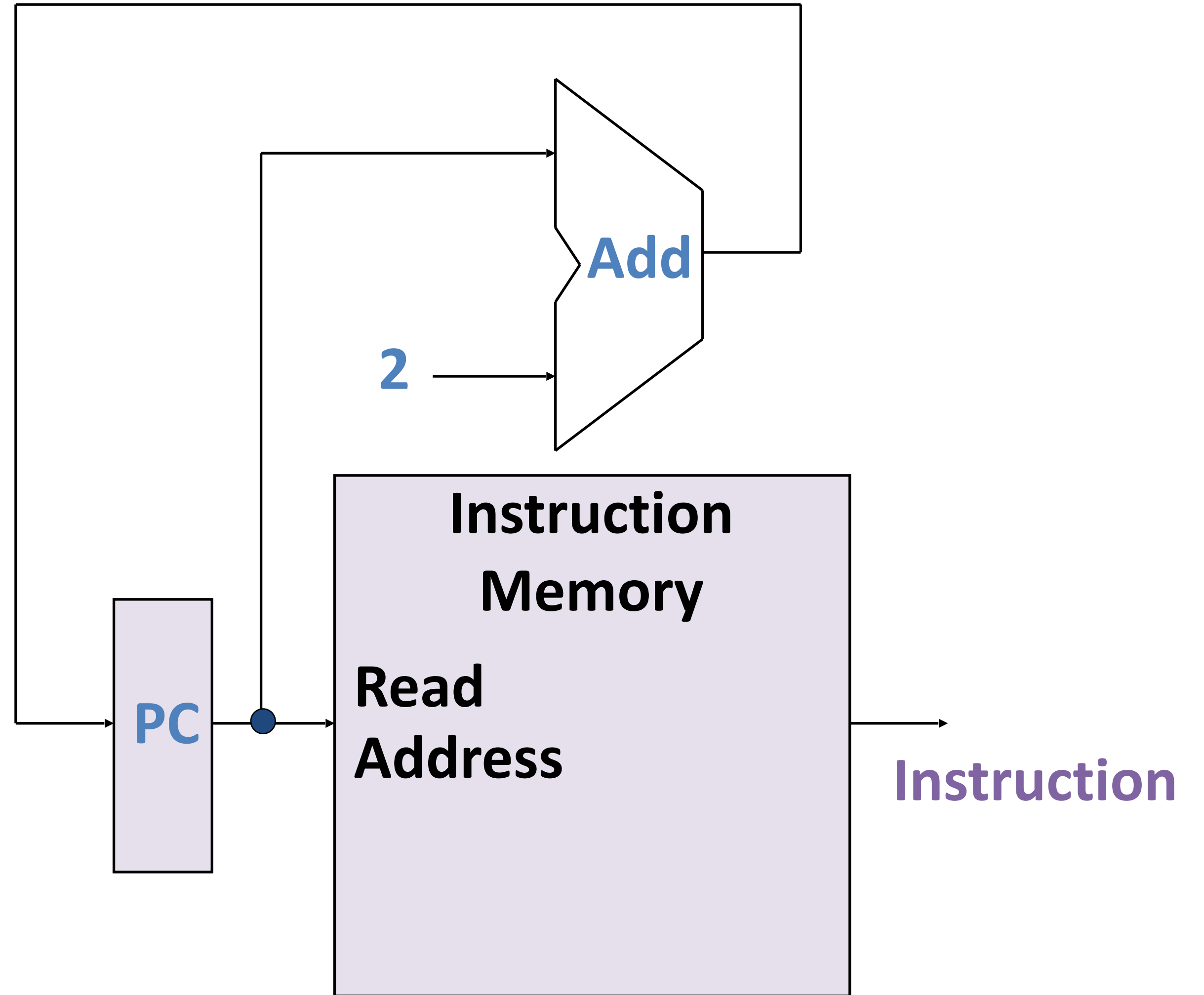
Instruction Fetch

(default, unless branch or jump)

Fetch instruction from memory.
Increment program counter (PC)
to point to the next instruction.

Processor
Loop

1. $ins \leftarrow IM[PC]$
2. $PC \leftarrow PC + 2$
3. Do ins



Instruction Encoding: 3 formats

All have 4-bit opcode in MSBs

Arithmetic instructions:

- 2 source register IDs (R_s, R_t)
- 1 destination register ID (R_d)



15:12	11:8	7:4	3:0
<i>opcode</i>	R_s	R_t	R_d

Memory/branch instructions:

- address/source register ID (R_s)
- data/source register ID (R_t)
- 4-bit offset

15:12	11:8	7:4	3:0
<i>opcode</i>	R_s	R_t	<i>offset</i>

Jump instruction:

- 12-bit offset

15:12	11:0
<i>opcode</i>	<i>offset</i>

Arithmetic Instructions

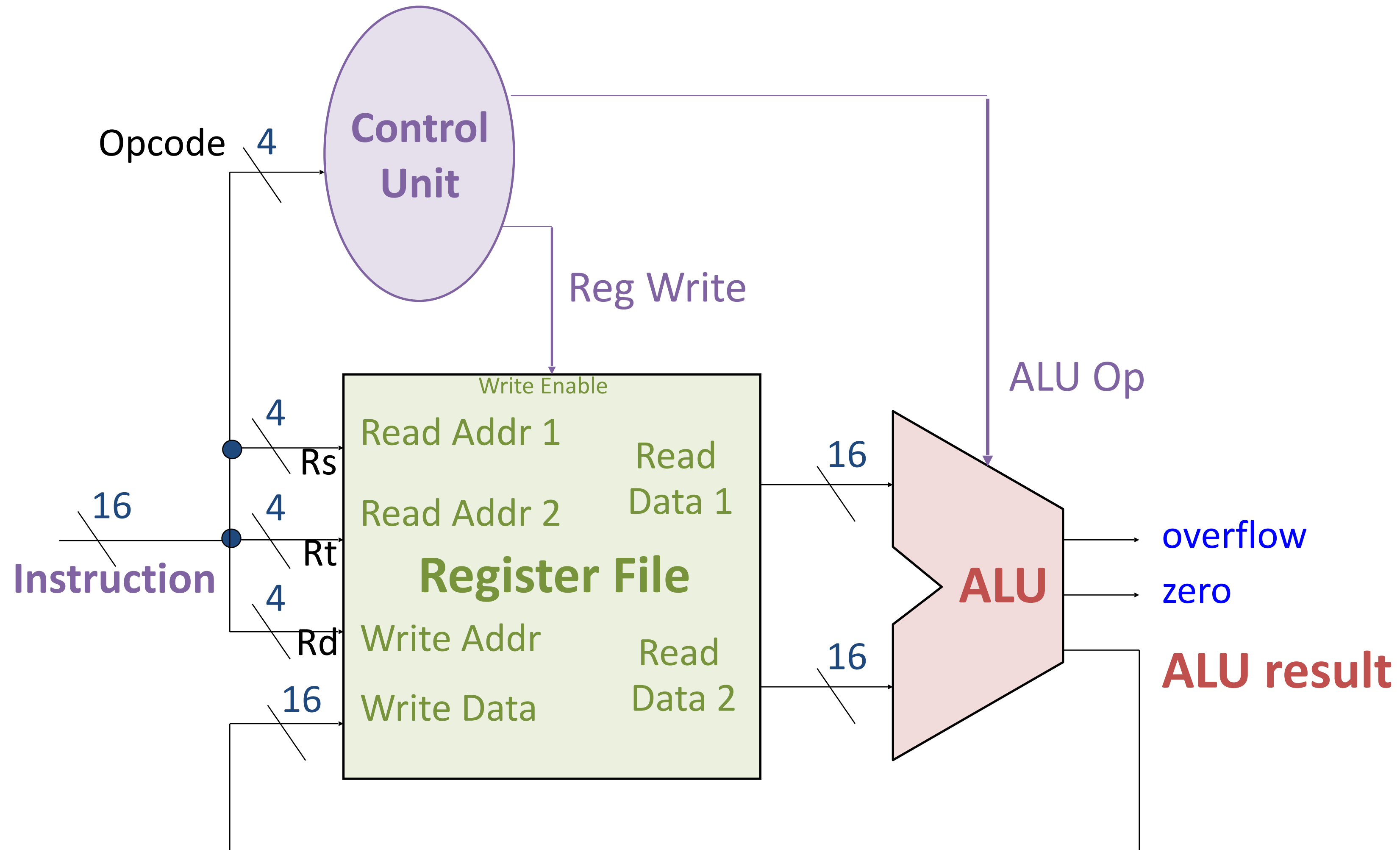
ADD R3, R6, R8

Opcode	Rs	Rt	Rd
0010	0011	0110	1000

16-bit Encoding

Instruction	Meaning	Opcode	Rs	Rt	Rd
ADD R_s, R_t, R_d	$R[d] \leftarrow R[s] + R[t]$	0010	0-15	0-15	0-15
SUB R_s, R_t, R_d	$R[d] \leftarrow R[s] - R[t]$	0011	0-15	0-15	0-15
AND R_s, R_t, R_d	$R[d] \leftarrow R[s] \& R[t]$	0100	0-15	0-15	0-15
OR R_s, R_t, R_d	$R_d \leftarrow R[s] R[t]$	0101	0-15	0-15	0-15
...					

Arithmetic Instructions: Instruction Decode, Register Access, ALU



Memory Instructions

SW R6, -8(R3)

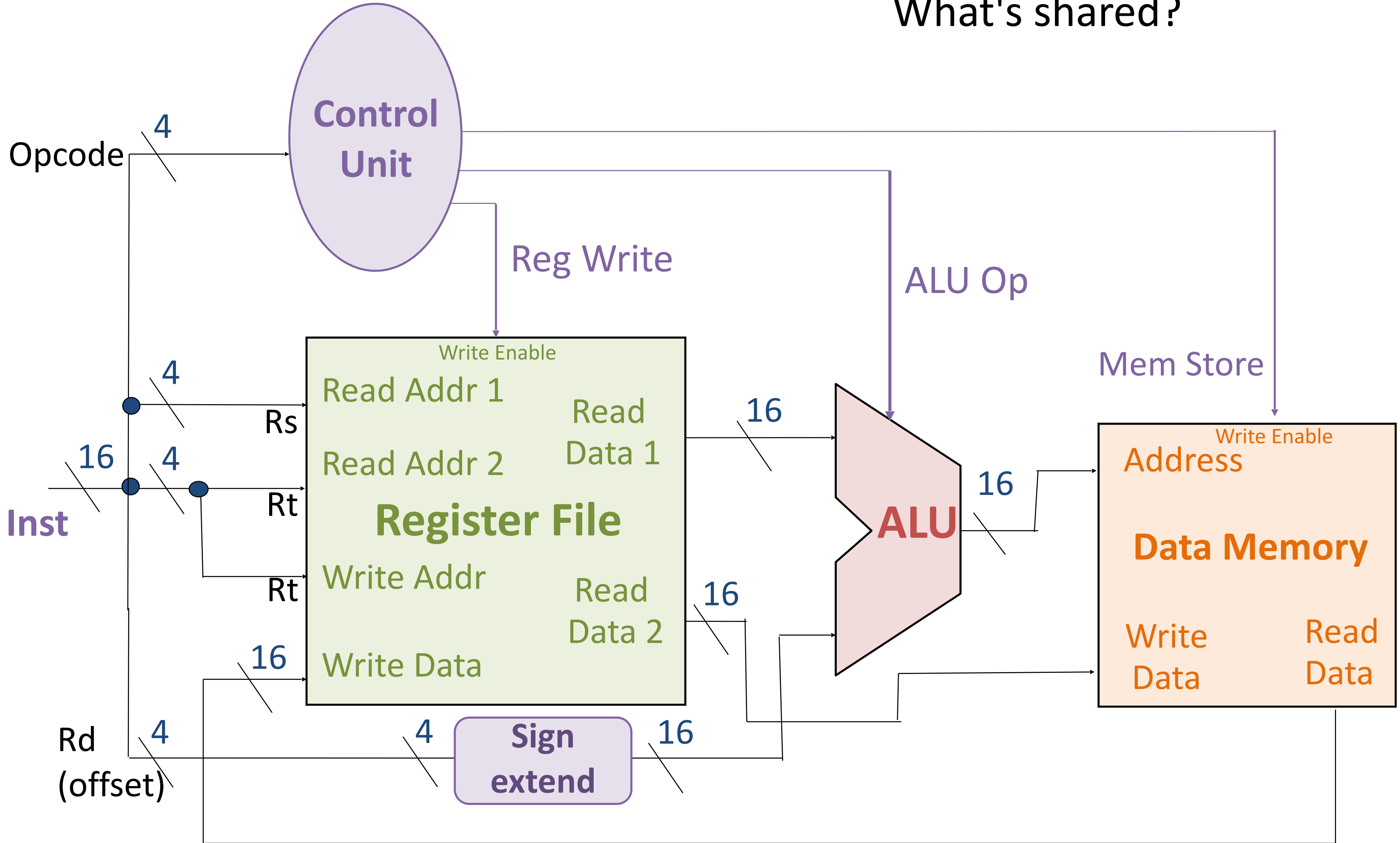
Opcode	Rs	Rt	Rd
0001	0011	0110	1000

Instruction	Meaning	Op	Rs	Rt	Rd
LW <i>Rt</i> , <i>offset</i> (<i>Rs</i>)	$R[t] \leftarrow \text{Mem}[R[s] + \text{offset}]$	0000	0-15	0-15	<i>offset</i>
SW <i>Rt</i> , <i>offset</i> (<i>Rs</i>)	$\text{Mem}[R[s] + \text{offset}] \leftarrow R[t]$	0001	0-15	0-15	<i>offset</i>
...					

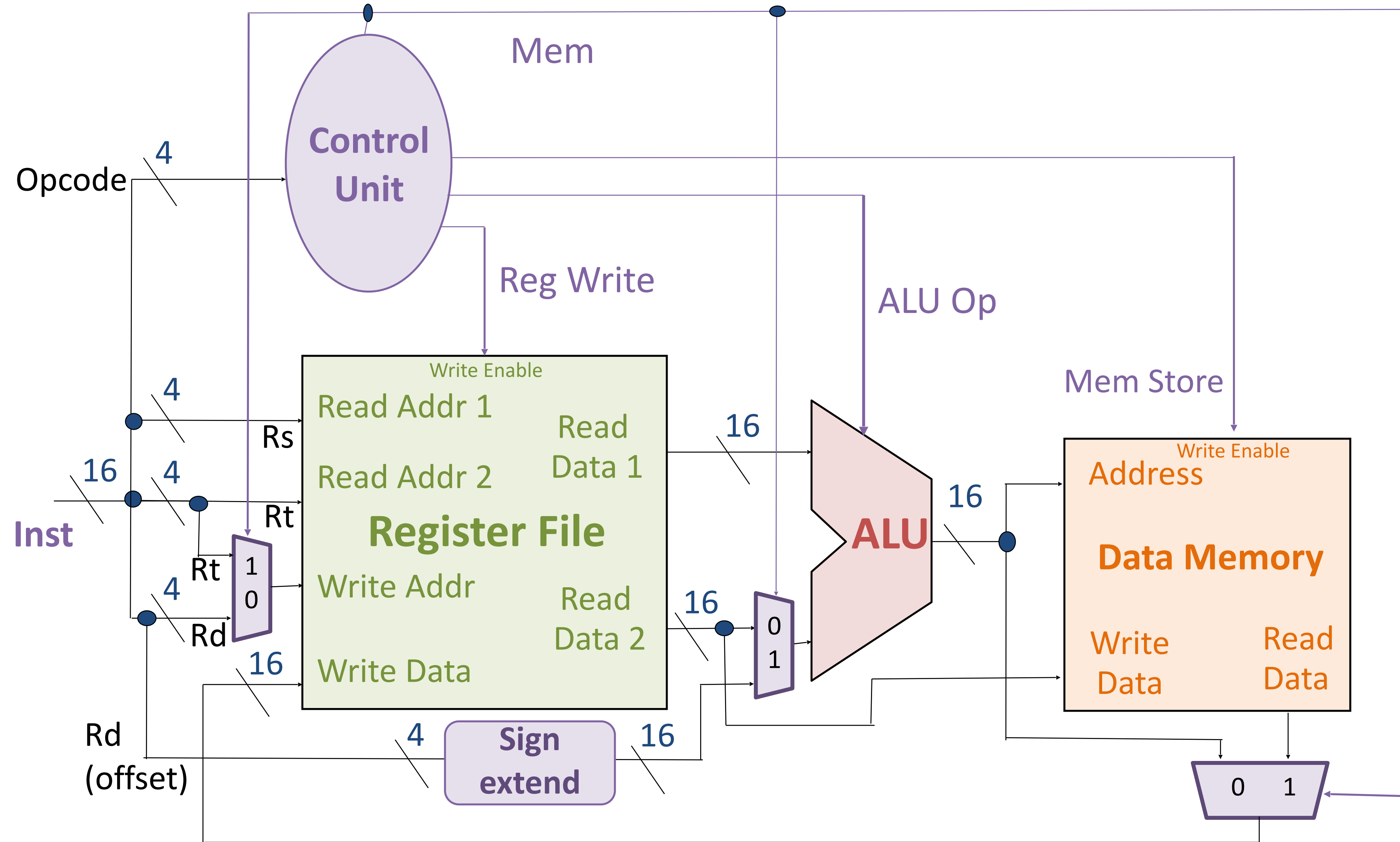
Memory Instructions: Instruction Decode, Register/Memory Access, ALU

How can we support arithmetic
and memory instructions?

What's shared?



Choose between Arithmetic/Memory instructions with MUXs



Control-flow Instructions

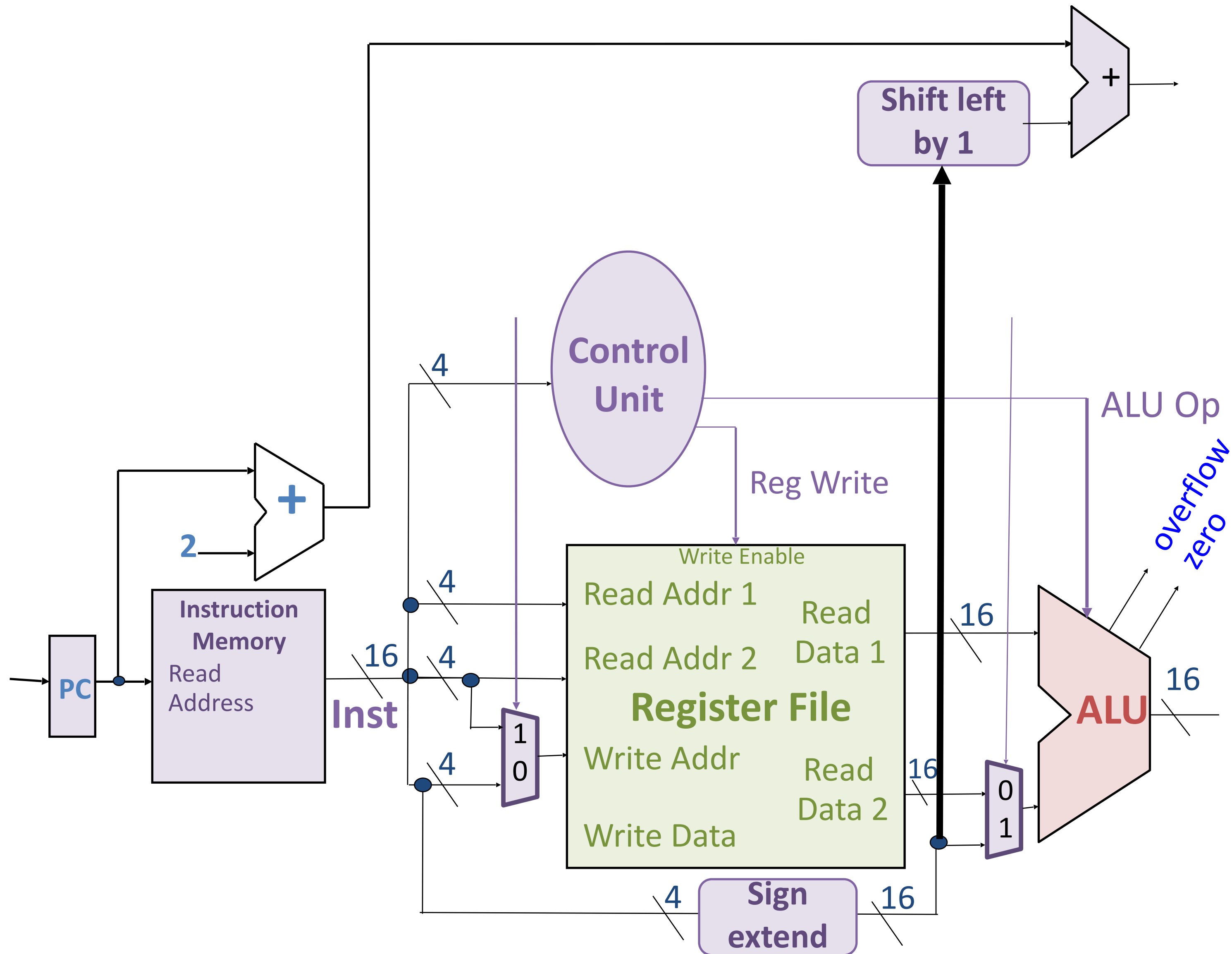
BEQ R1, R2, -2

Op	Rs	Rt	Rd
0111	0001	0010	1110

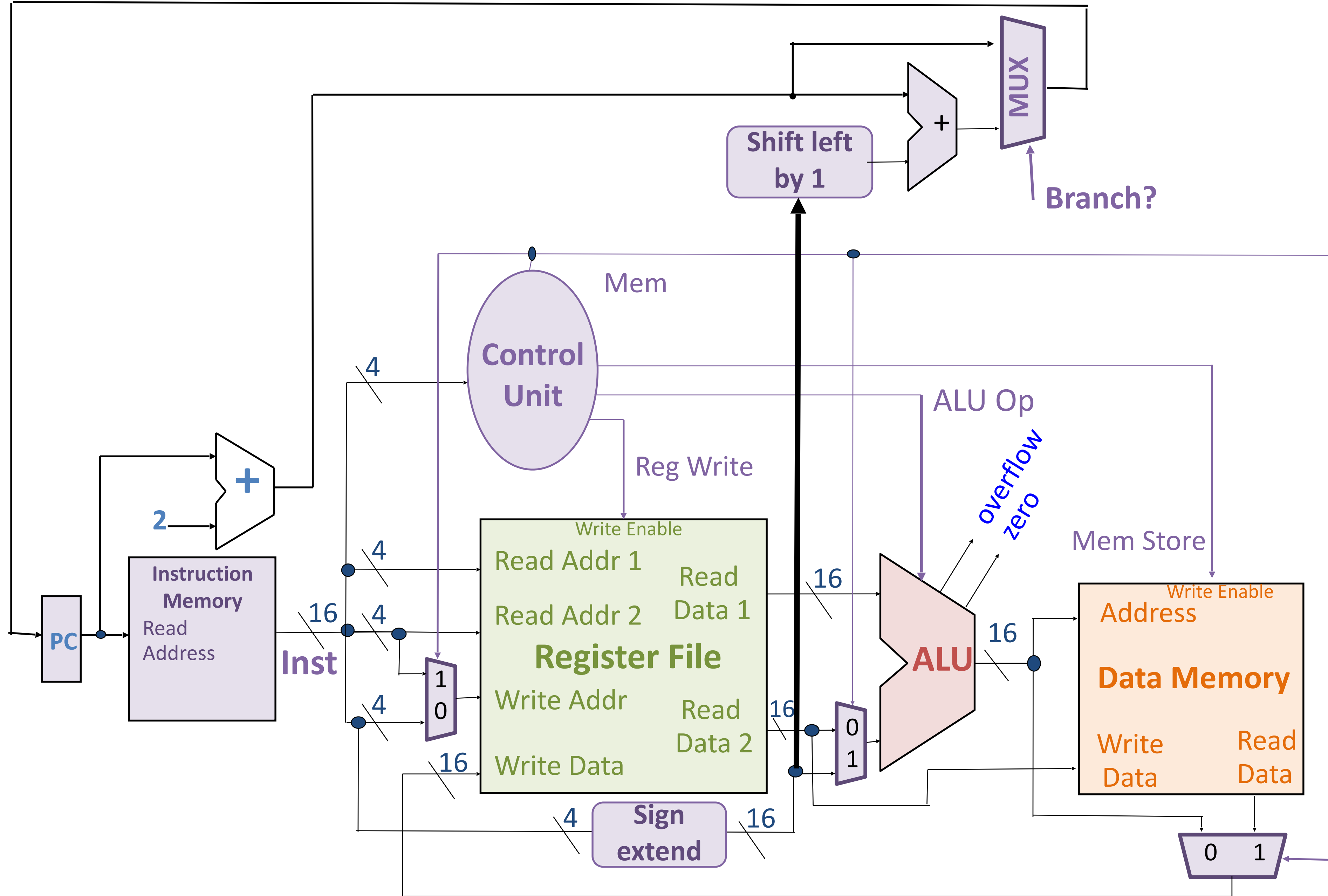
16-bit Encoding

Instruction	Meaning	Op	Rs	Rt	Rd
BEQ <i>Rs, Rt, offset</i>	<i>If $R[s] == R[t]$ then $PC \leftarrow PC + 2 + offset * 2$</i>	0111	0-15	0-15	<i>offset</i>
...					

Compute branch target for BEQ



Make branch decision



What's missing from what we covered in lecture?

- Details of Control Unit
 - ALU op is **not** instruction opcode; some translation involved
 - Reg Write bit (for ADD, SUB, AND, OR, LW)
 - Mem Store bit (for SW)
 - Mem bit (arithmetic/memory MUX bit)
 - Branch bit (for BEQ)
- Implementation of JMP
- Implementation of HALT (basically stops the clock running the computer; we won't implement this)

See **Lab 5: Processor Datapath** and **Arch Assignment!**

HW ARCH **not the only implementation**

Single-cycle architecture

- Simple, (barely!) fits on a slide (and in our heads).
- One instruction takes one clock cycle.
- Slowest instruction determines minimum clock cycle.
- Inefficient.

Could it be better?

- Performance, energy, debugging, security, reconfigurability, ...
- Pipelining
- OoO: Out-of-order execution
- SIMD: single instruction multiple data (“vector” instructions)
- Caching
- Microcode vs. direct hardware implementation
- ... enormous, interesting design space of **Computer Architecture**

Conclusion of unit: Computational Building Blocks (HW)

Lectures

- Digital Logic
- Data as Bits
- Integer Representation
- Combinational Logic
- Arithmetic Logic
- Sequential Logic
- A Simple Processor

Labs

- 1: Transistors to Gates
- 2: Data as Bits
- 3: Combinational Logic & Arithmetic
- 4: ALU & Sequential Logic
- 5: Processor Datapath

Topics

- Transistors, digital logic gates
- Data representation with bits, bit-level computation
- Number representations, arithmetic
- Combinational and arithmetic logic
- Sequential (stateful) logic
- Computer processor architecture overview

Assignments

- Gates
- Zero
- Bits
- Circuits
- Arch

Mid-semester exam 1: HW
October 19
(2 weeks from Thursday)