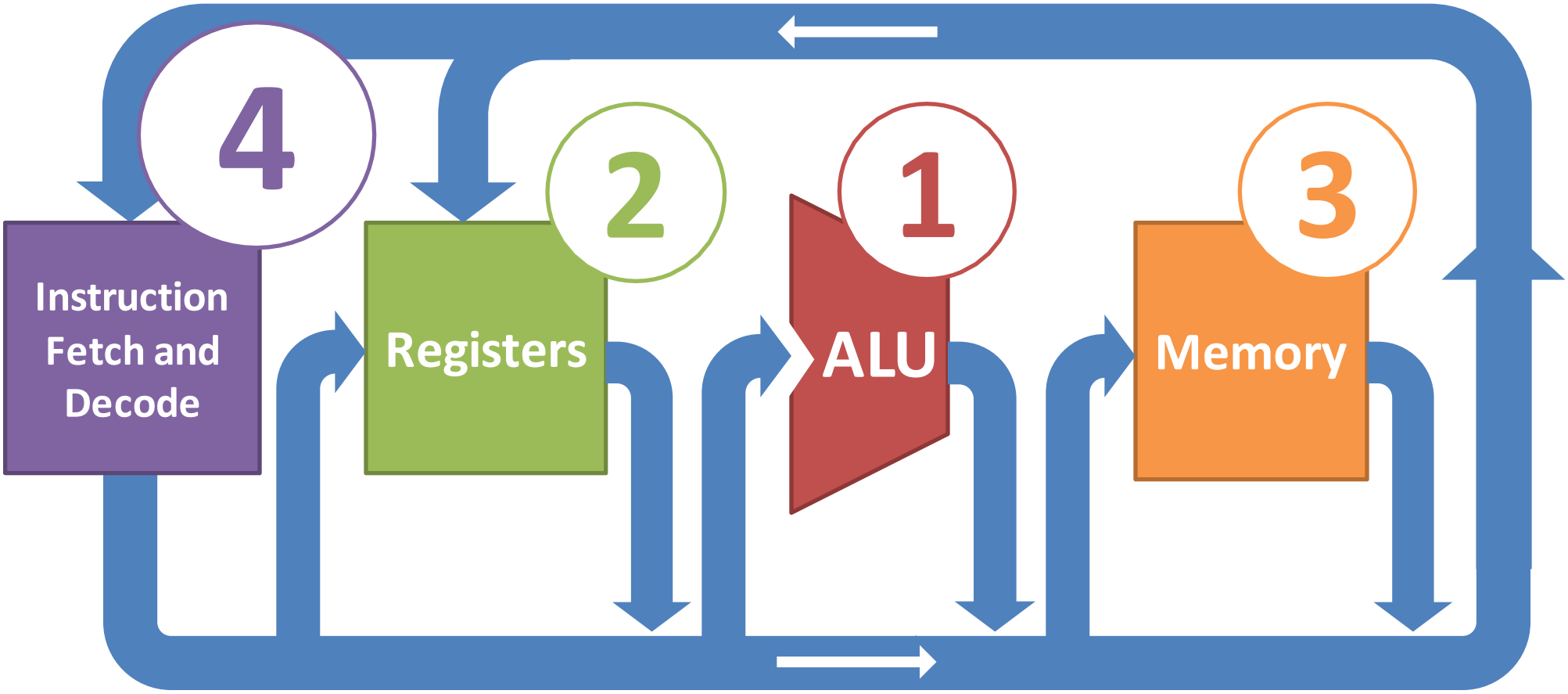
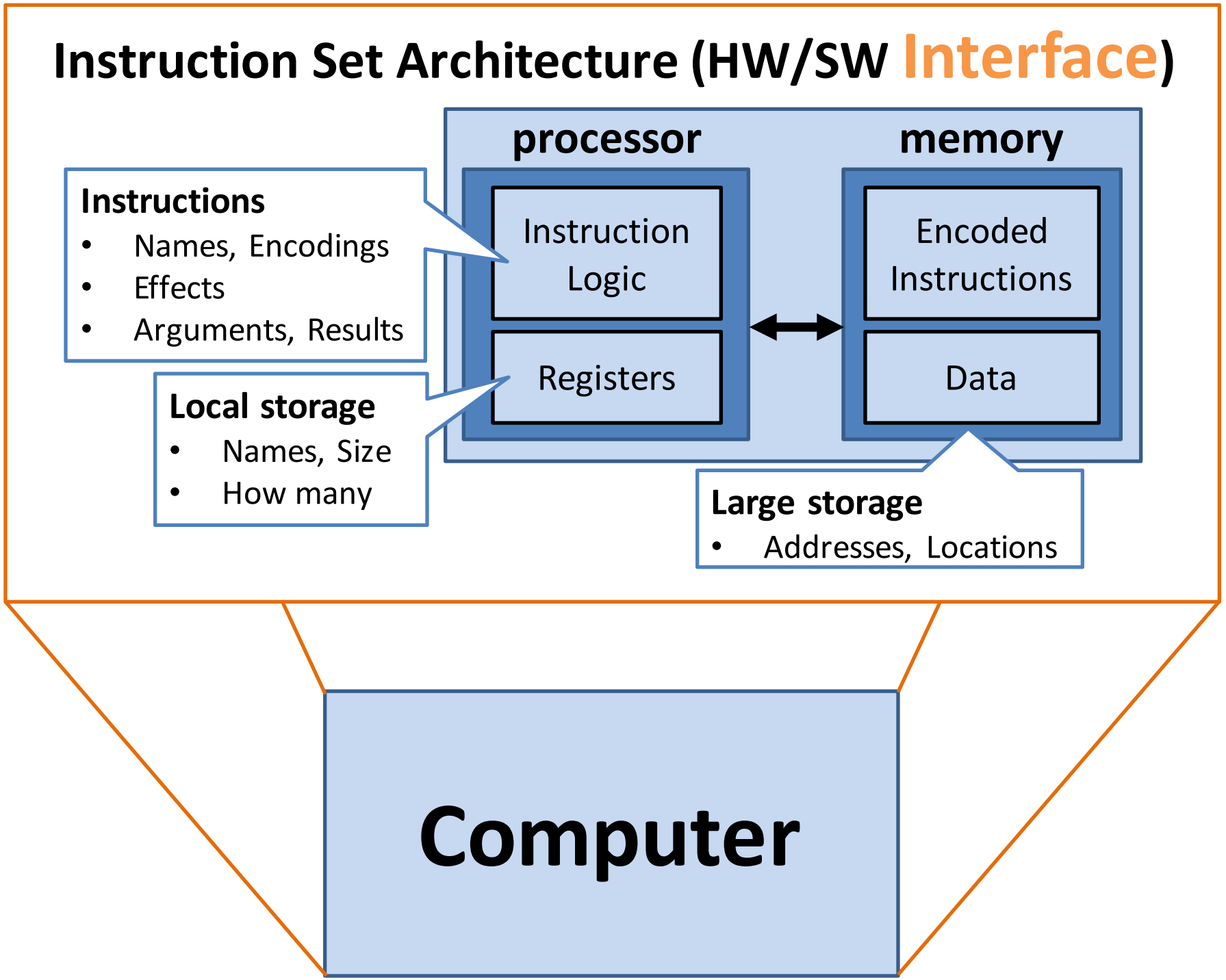
Toy ISA and microarchitecture




Instruction Set Architecture (HW/SW Interface)
processor
memory
Instruction Logic
Registers
Encoded Instructions
Data
Instructions
• Names, Encodings
• Effects
• Arguments, Results
Local storage
• Names, Size
• How many
Large storage
• Addresses, Locations
Computer

Toy Instruction Set Architecture

(a.k.a. Mini-MIPS)

- **Word size = 16 bits, data bus = 16 bits.**
  - Register size = 16 bits.
  - ALU computes on 16-bit values.
- **Memory is byte-addressable**, also access 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*.**
- Each instruction executes in a single clock cycle.
- Special **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
…	…

Instruction Set

MSB **16-bit Encoding** LSB

Mnemonic	Meaning	Opcode	Rs	Rt	Rd
ADD R*s*, R*t*, R*d*	R[*d*] ← R[*s*] + R[*t*]	0010	*s*	*t*	*d*
SUB R*s*, R*t*, R*d*	R[*d*] ← R[*s*] - R[*t*]	0011	*s*	*t*	*d*
AND R*s*, R*t*, R*d*	R[*d*] ← R[*s*] & R[*t*]	0100	*s*	*t*	*d*
OR R*s*, R*t*, R*d*	R[*d*] ← R[*s*] | R[*t*]	0101	*s*	*t*	*d*
LW R*t*, *offset*(R*s*)	R[*t*] ← M[R[*s*] + *offset*]	0000	*s*	*t*	*offset*
SW R*t*, *offset*(R*s*)	M[R[*s*] + *offset*] ← R[*t*]	0001	*s*	*t*	*offset*
BEQ R*s*, R*t*, *offset*	If R[*s*] == R[*t*] then [PC] ← [PC]+2 + *offset**2 Else [PC] ← [PC]+2	0111	*s*	*t*	*offset*
JMP *offset*	[PC] ← *offset**2	1000	*o* *f*	*f* *s*	*e* *t*

(R = register file, M = memory)

Instruction Encoding: 3 formats

All have 4-bit opcode in MSBs

Arithmetic instructions:

- 2 source register IDs (Rs,Rt)
- 1 destination register ID (Rd)

15:12	11:8	7:4	3:0
Op	*Rs*	*Rt*	*Rd*

Memory/branch instructions:

- address/source register ID (Rs)
- data/source register ID (Rt)
- 4-bit offset

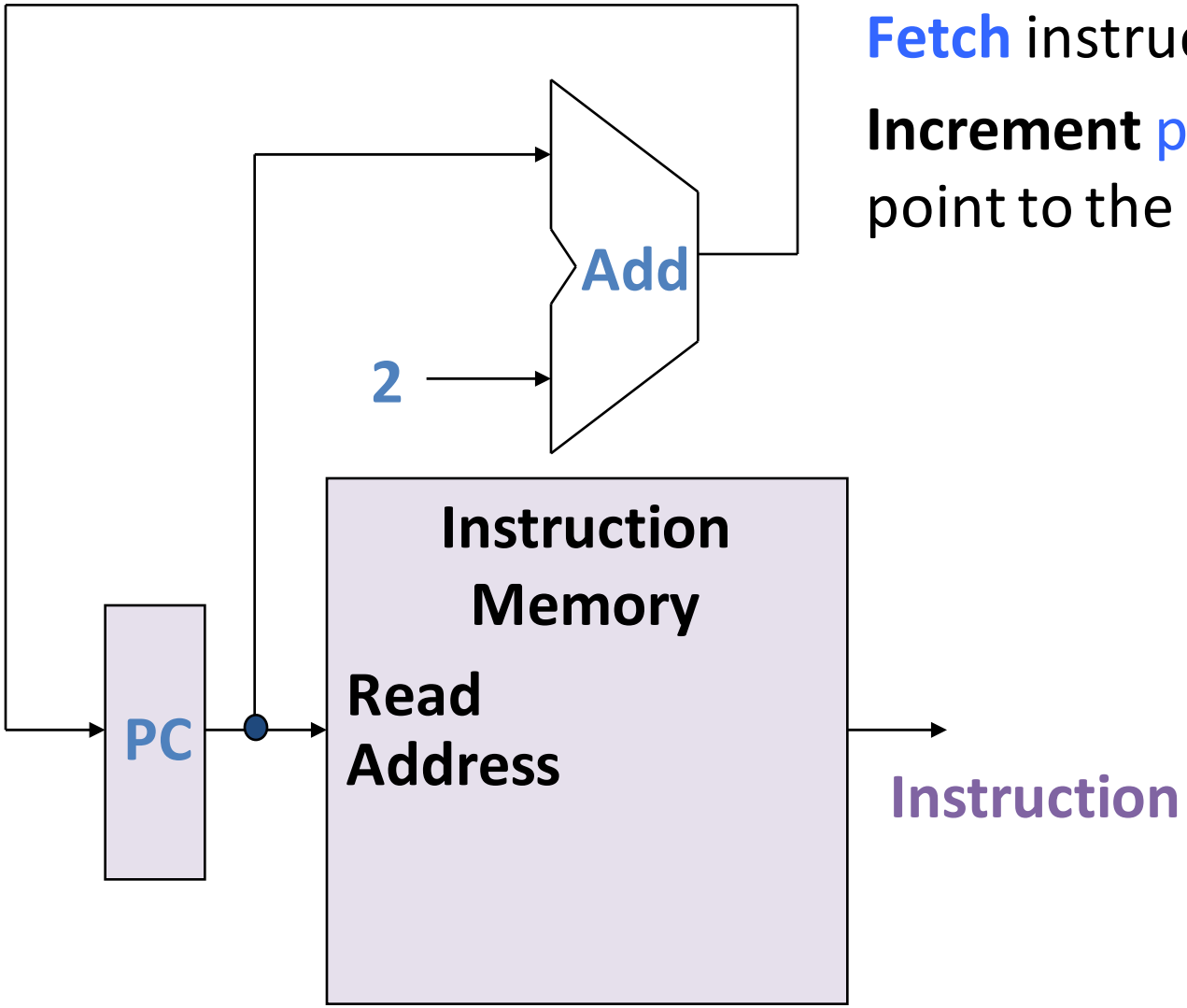
15:12	11:8	7:4	3:0
Op	*Rs*	*Rt*	*offset*

Jump instruction:

- 12-bit offset

15:12	11:0
Op	*offset*

Instruction Fetch



Fetch instruction from memory.

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

Arithmetic Instructions

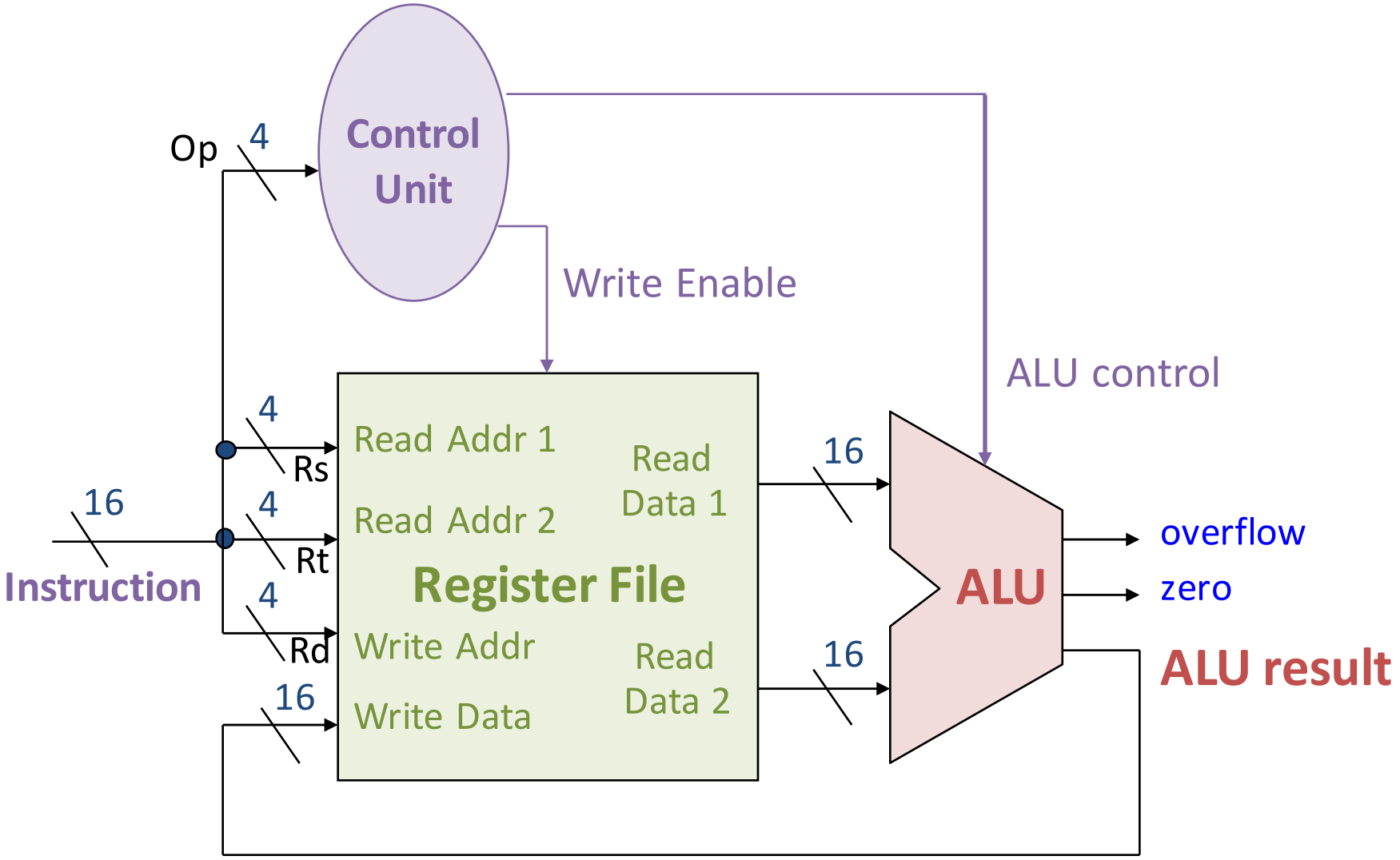
ADD R3, R6, R8

Op	Rs	Rt	Rd
0010	0011	0110	1000

16-bit Encoding

Instruction	Meaning	Opcode	Rs	Rt	Rd
ADD *Rs, Rt, Rd*	*R[d]* ← *R[s]* + *R[t]*	0010	0-15	0-15	0-15
SUB *Rs, Rt, Rd*	*R[d]* ← *R[s]* – *R[t]*	0011	0-15	0-15	0-15
AND *Rs, Rt, Rd*	*R[d]* ← *R[s]* & *R[t]*	0100	0-15	0-15	0-15
OR *Rs, Rt, Rd*	*Rd* ← *R[s]* | *R[t]*	0101	0-15	0-15	0-15
…					

Instruction Decode, Register Access, ALU



Memory Instructions

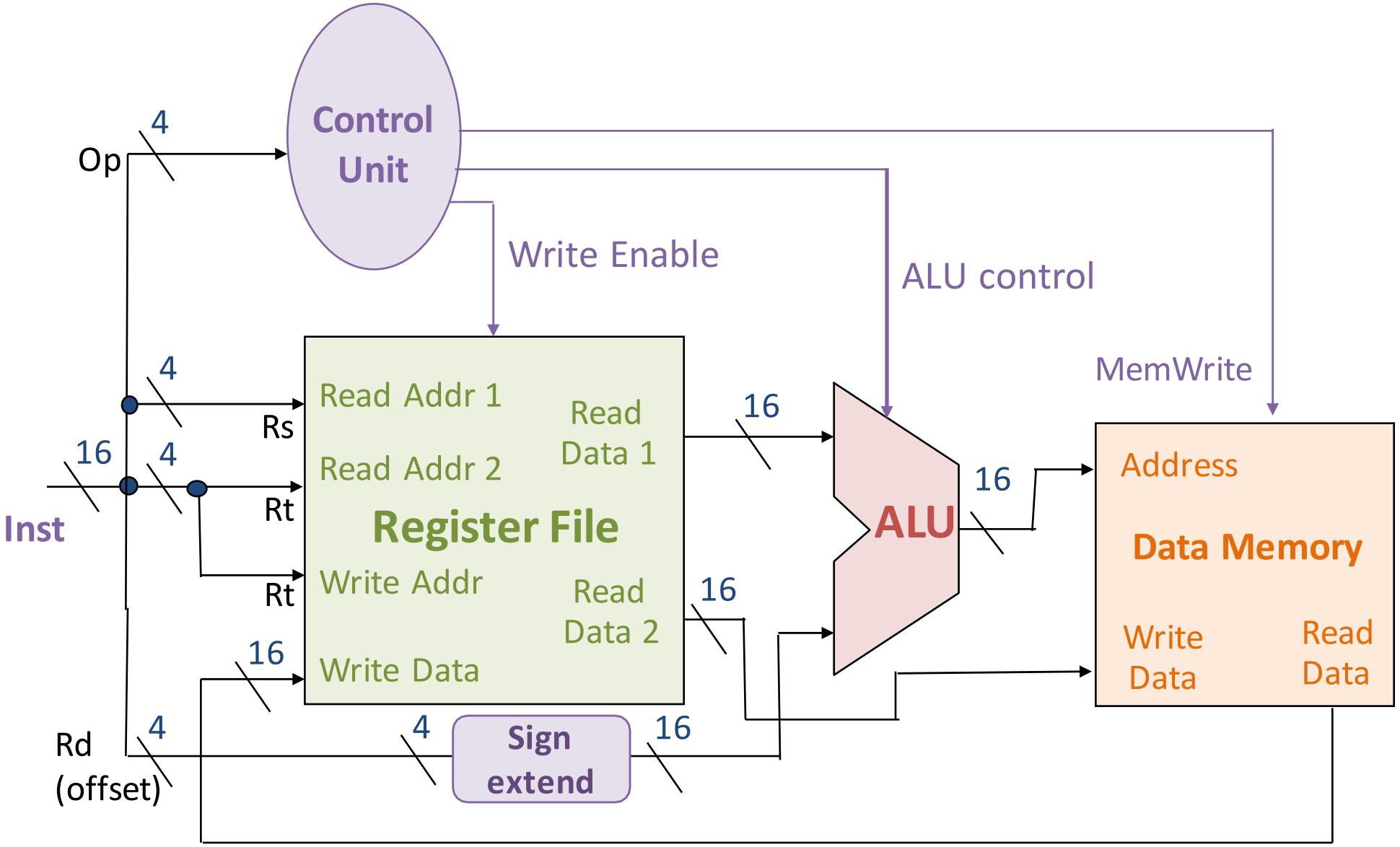
SW R6, 8(R3)

Op	Rs	Rt	Rd
0001	0011	0110	1000

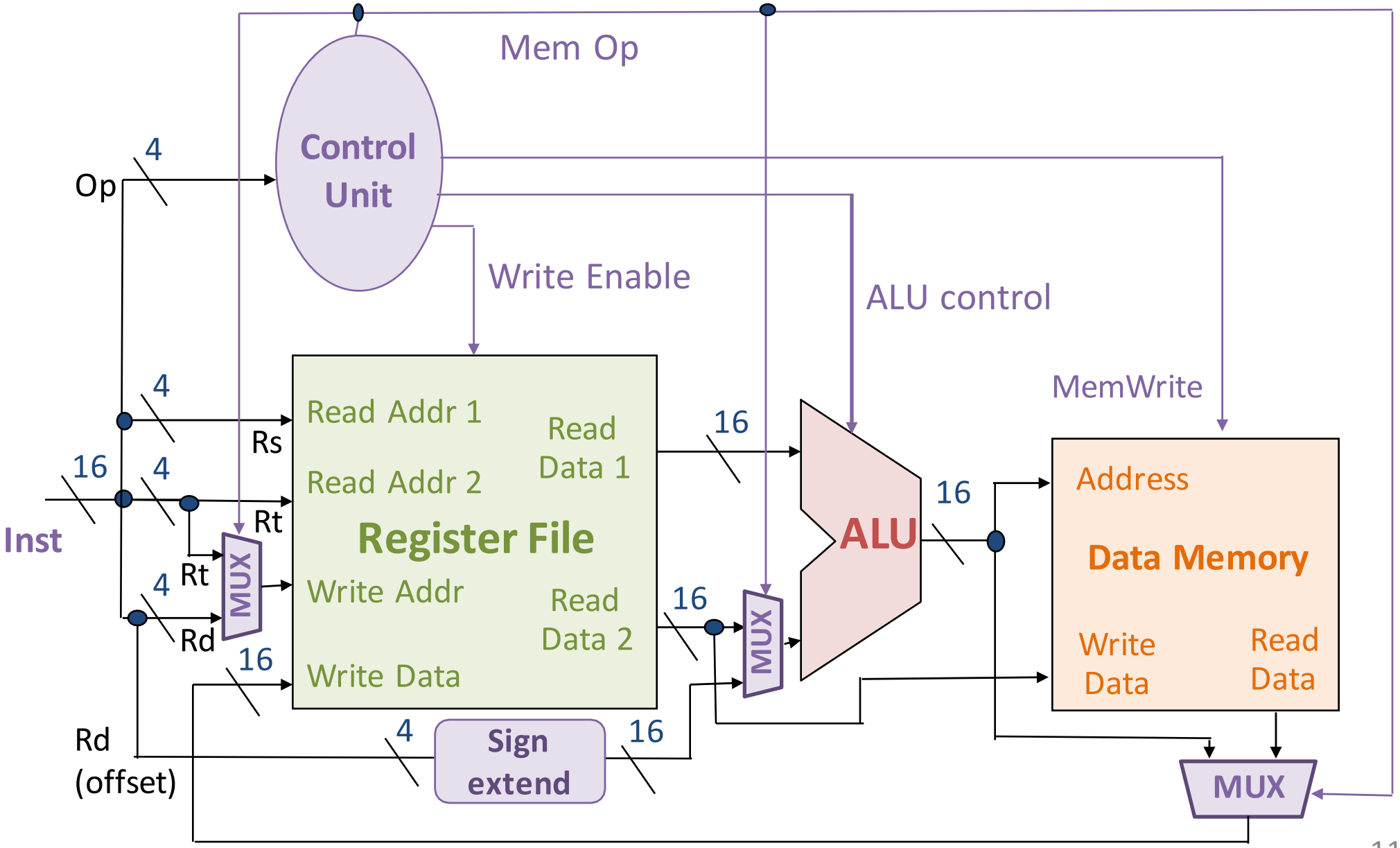
Instruction	Meaning	Op	Rs	Rt	Rd
LW *Rt, offset(Rs)*	*R[t] ← Mem[R[s] + offset]*	0000	0-15	0-15	offset
SW *Rt, offset(Rs)*	*Mem[R[s] + offset] ← R[t]*	0001	0-15	0-15	offset
…					

Memory access

How can we support arithmetic **and** memory instructions? What's shared?



MUXes to the rescue!



Control-flow Instructions

BEQ R1, R2, 28

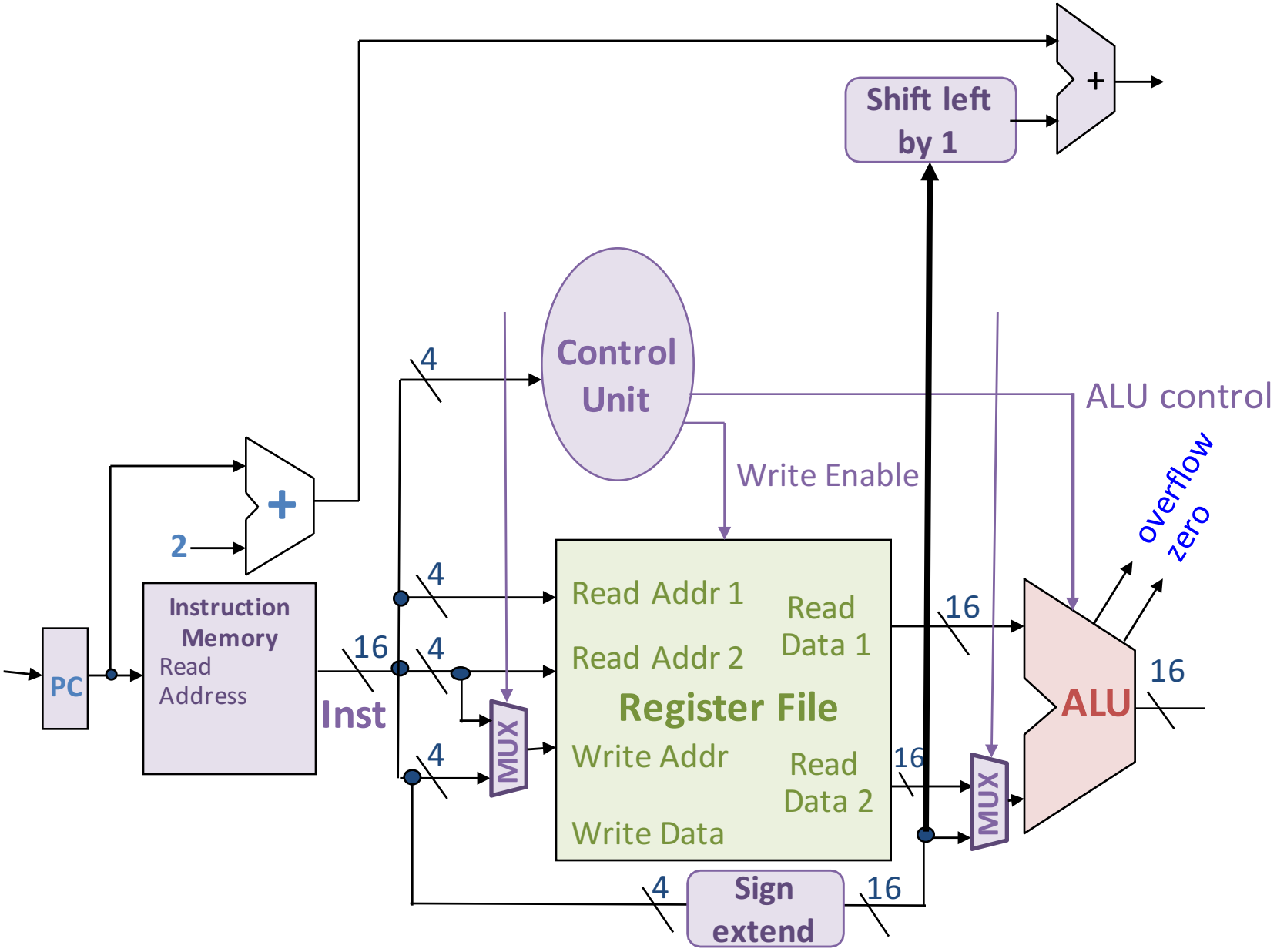
Op	Rs	Rt	Rd
0111	0001	0010	1110

16-bit Encoding

Instruction	Meaning	Op	Rs	Rt	Rd
BEQ *Rs, Rt, offset*	*If R[s] == R[t] then* *PC ← PC+2 + offset*2* *Else* *PC ← PC+2*	0111	0-15	0-15	offset
JMP *offset*	*PC ← offset*2*	1000	o f	f s	e t
…					

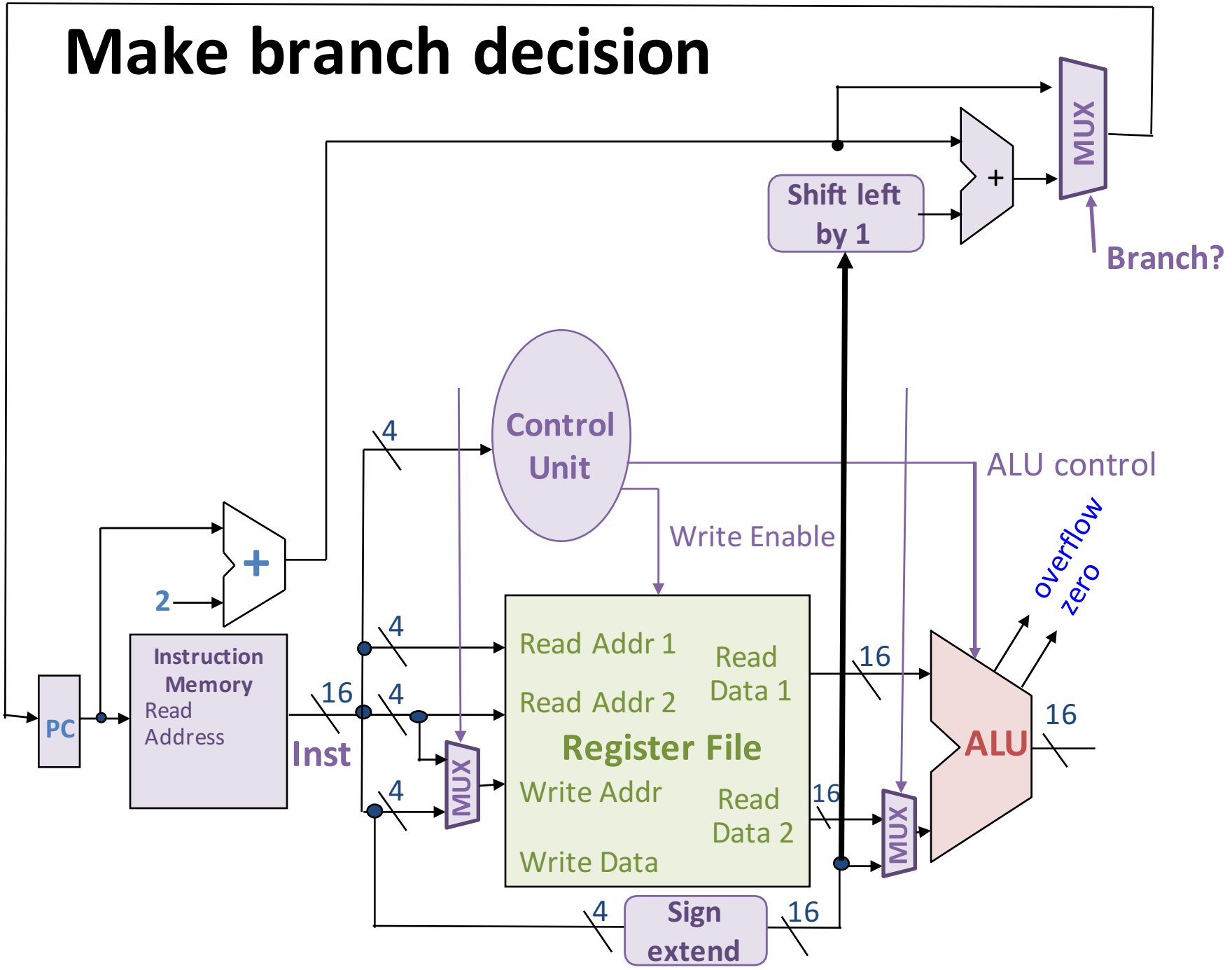
Use these to implement: conditionals, loops, etc.

Compute branch target

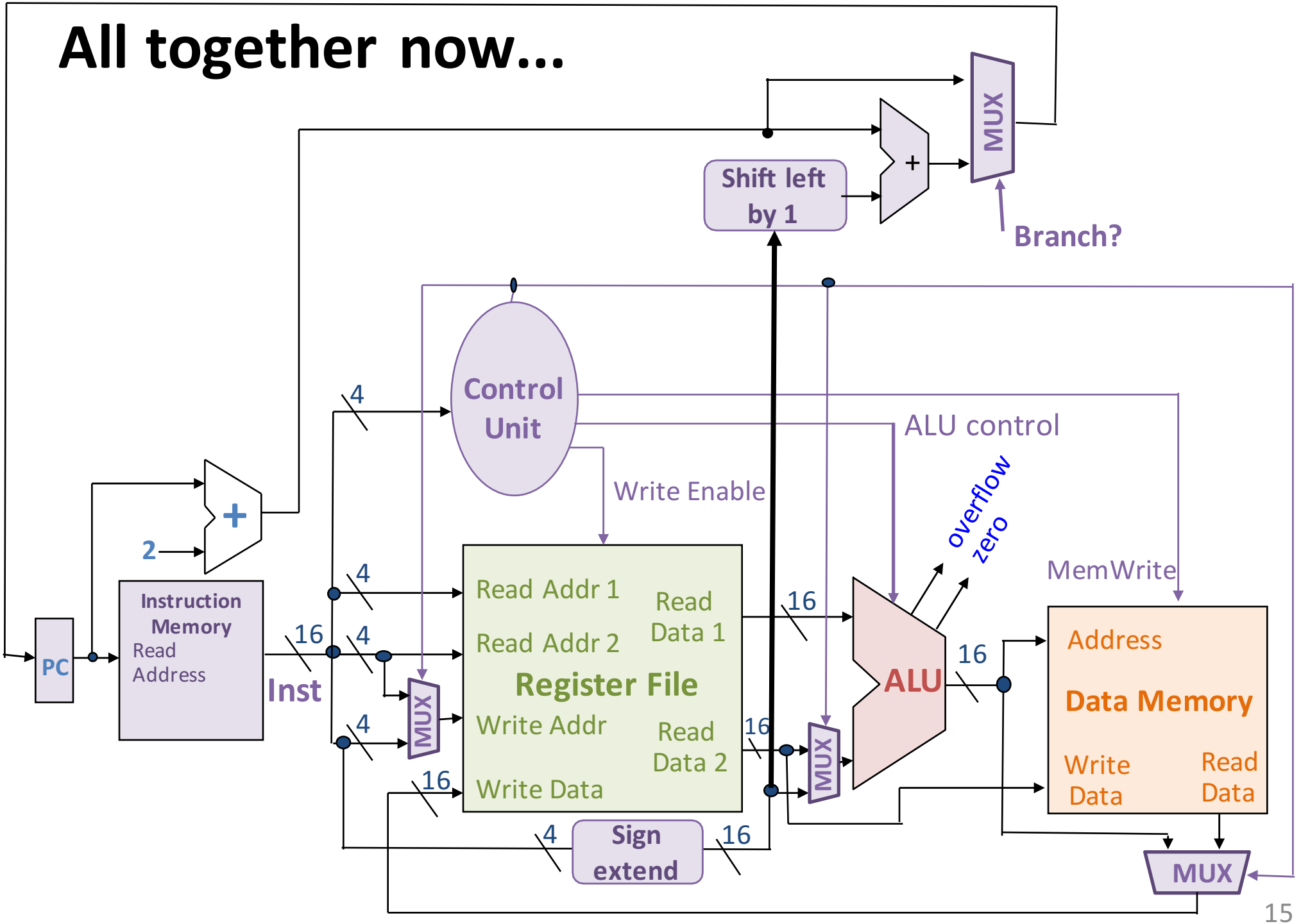


Make branch decision

ex



All together now...



Microarchitecture

Single-cycle architecture

- Simple, "easily" fits on a slide (and in your head).
- One instruction takes one clock cycle.
- Slowest instruction determines minimum clock cycle.
- Inefficient.

Could it be better?

- How? Performance, energy, debugging, security, reconfigurability, …
- Pipelining
- OoO: out-of-order execution
- SIMD: single instruction multiple data
- Caching
- Microcode vs. direct hardware implementation
- … enormous, interesting design space of **Computer Architecture**