Computer Architecture I

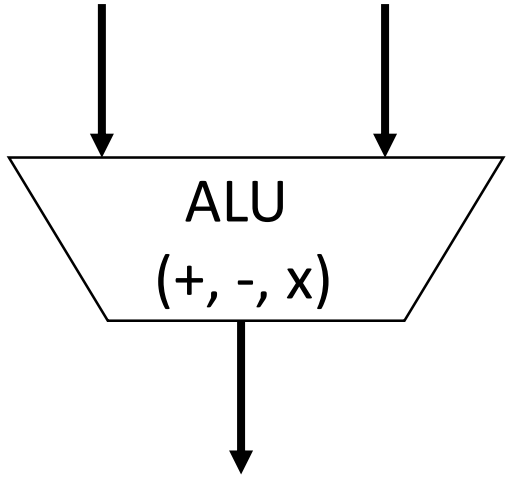
CS-GY-6133

Topic: Introduction and Instruction Set Architectures (ISA)

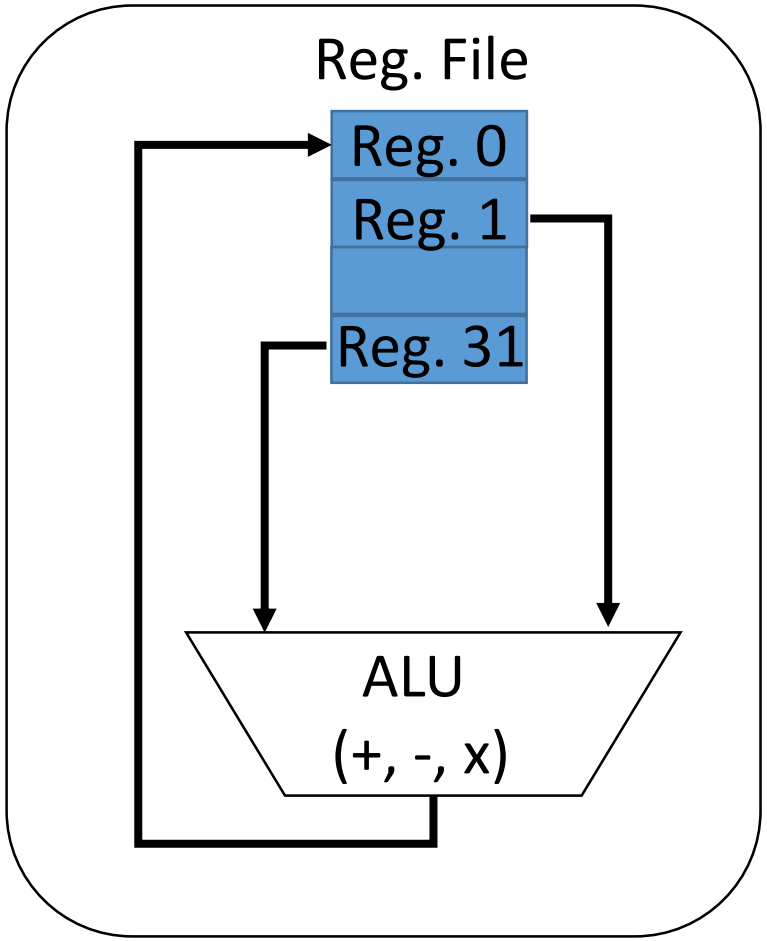
Instructor: Siddharth Garg

ISA Design: Operand Addresses

Where do the operands come from?



Where is the result written?



add rs, rt, rd
($rd ← $rs + $rt)

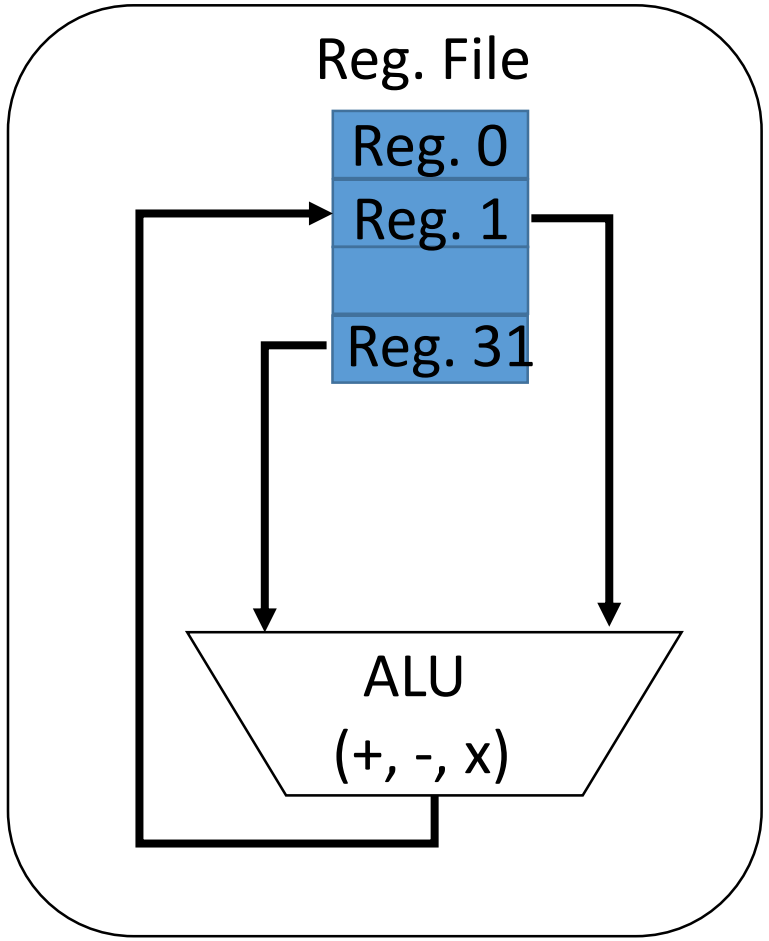
Value of register rt

sub rs, rt, rd
($rd ← $rs - $rt)

Instruction specifies 2 source and 1 destination register

"3-Address" ISA (E.g., MIPS ISA)

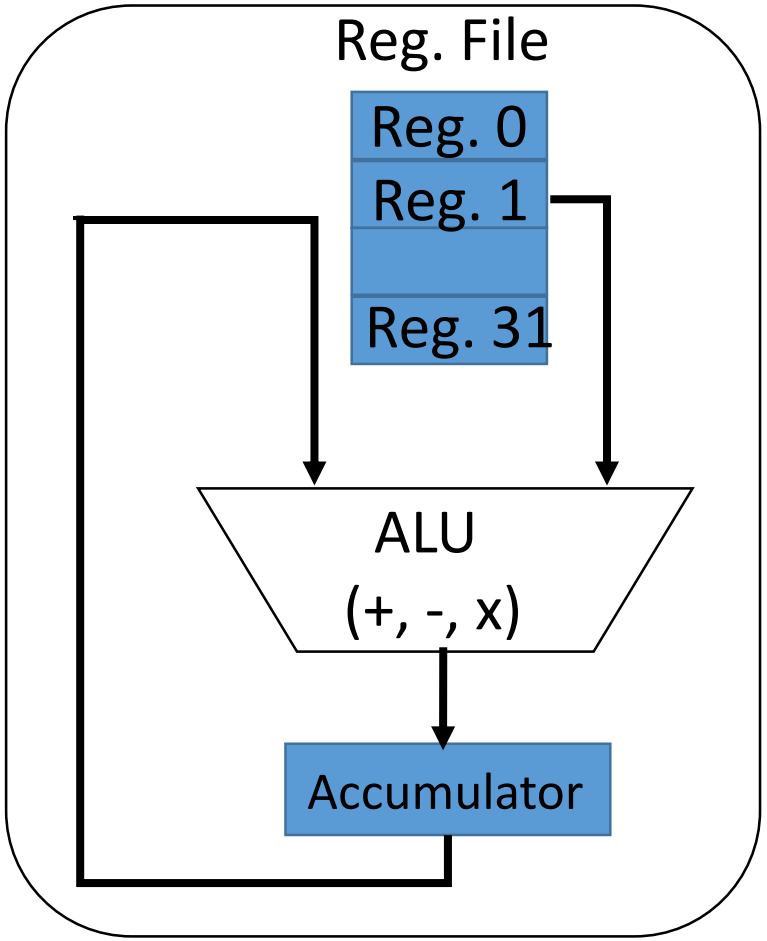
1- and 2-Address ISAs



Write result back to a source register

"2-Address" ISA

(Have we seen an example today?)



Special purpose "Accumulator" serves as an operand and as the destination

"1-Address" ISA

(Also called an "Accumulator Architecture")

So Which is "Better"?

- Metrics?
  - Code size (Bytes)
  - Design complexity

```
add r2, r3, r2
sub r2, r3, r3
sub r2, r3, r2
```

Assume 5-bit opcode for all 3 ISAs

How many bits requires to address 32 registers?

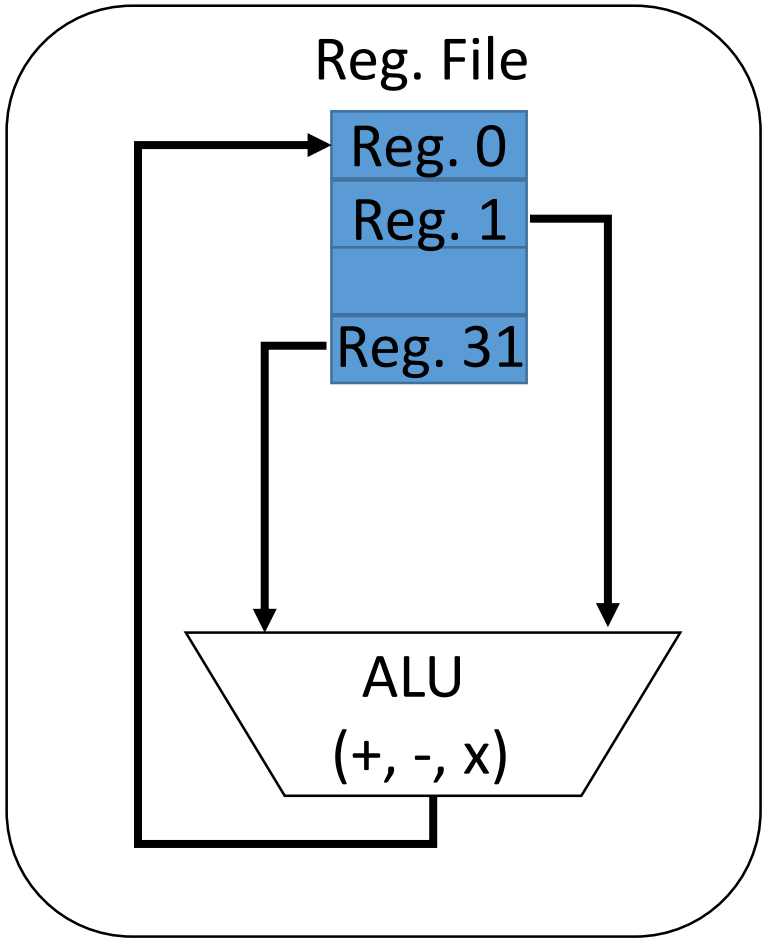
```
add r3, r2
sub r4, r4
add r4, r3
sub r3, r4
sub r3, r4
add r3, r2
 sub r2, r3
```

```
mvfrm r2
add r3
mvto r2
sub r3
mvto r3
mvfrm r2
sub r3
mvto r2
```

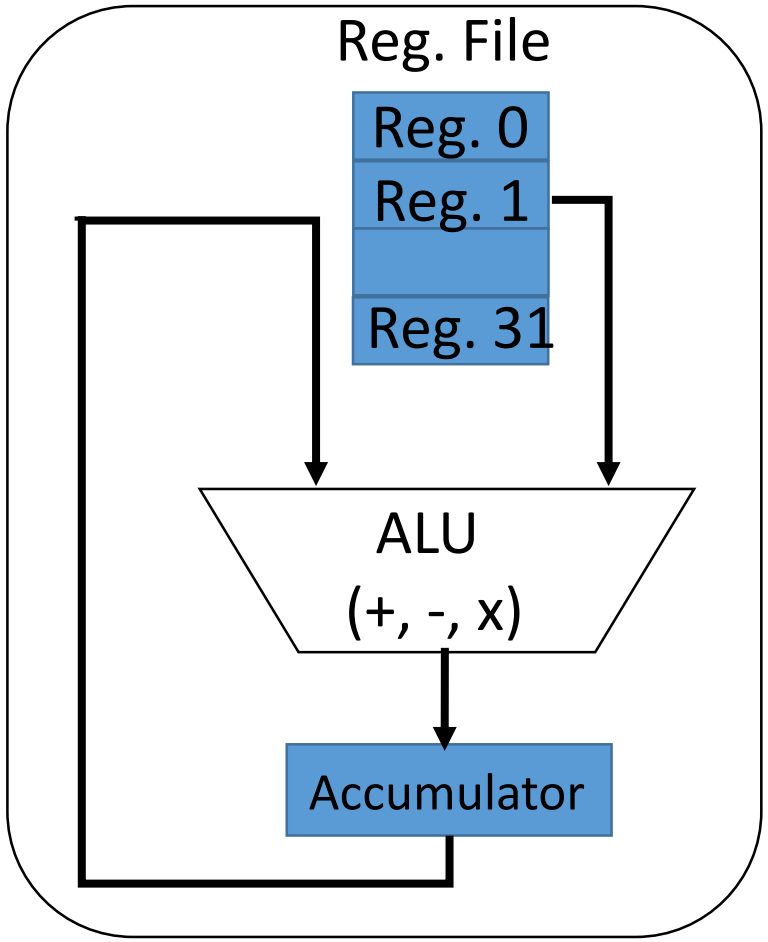
- Code size = Num Insts x Bits/Inst
  - 3-Address ISA
  - 2-Address ISA
  - 1-Address ISA

So Which is “Better”?

- Metrics?
  - Code size (Bytes)
  - Design complexity

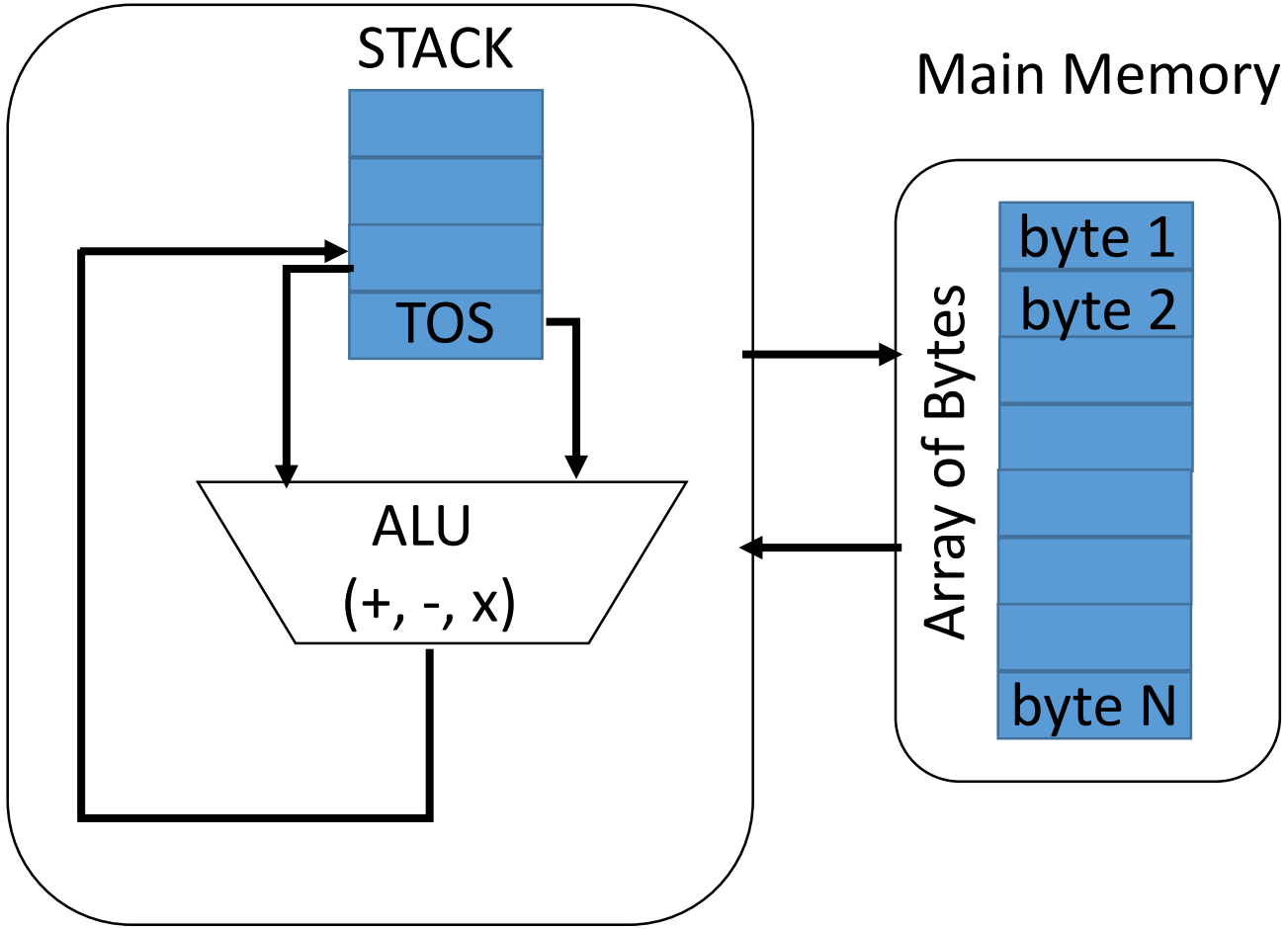


2 Read ports
1 Write port



1 Read ports
+ 1 R/W accum

0-Address ISA (Stack Machines)



pop M
Mem[M] ← $TOS
TOS ← TOS-1

push M
TOS ← TOS+1
$TOS ← Mem[M]

add
$TOS-1 ←$TOS + $(TOS-1)
//Pull out two operands from stack,
//replace with result
TOS ←TOS-1
//Update stack pointer

In-Class Problem 1

Variables x,y,w,z are stored at addresses &x, &y, &w and &z in main memory.

Write stack machine code to compute:

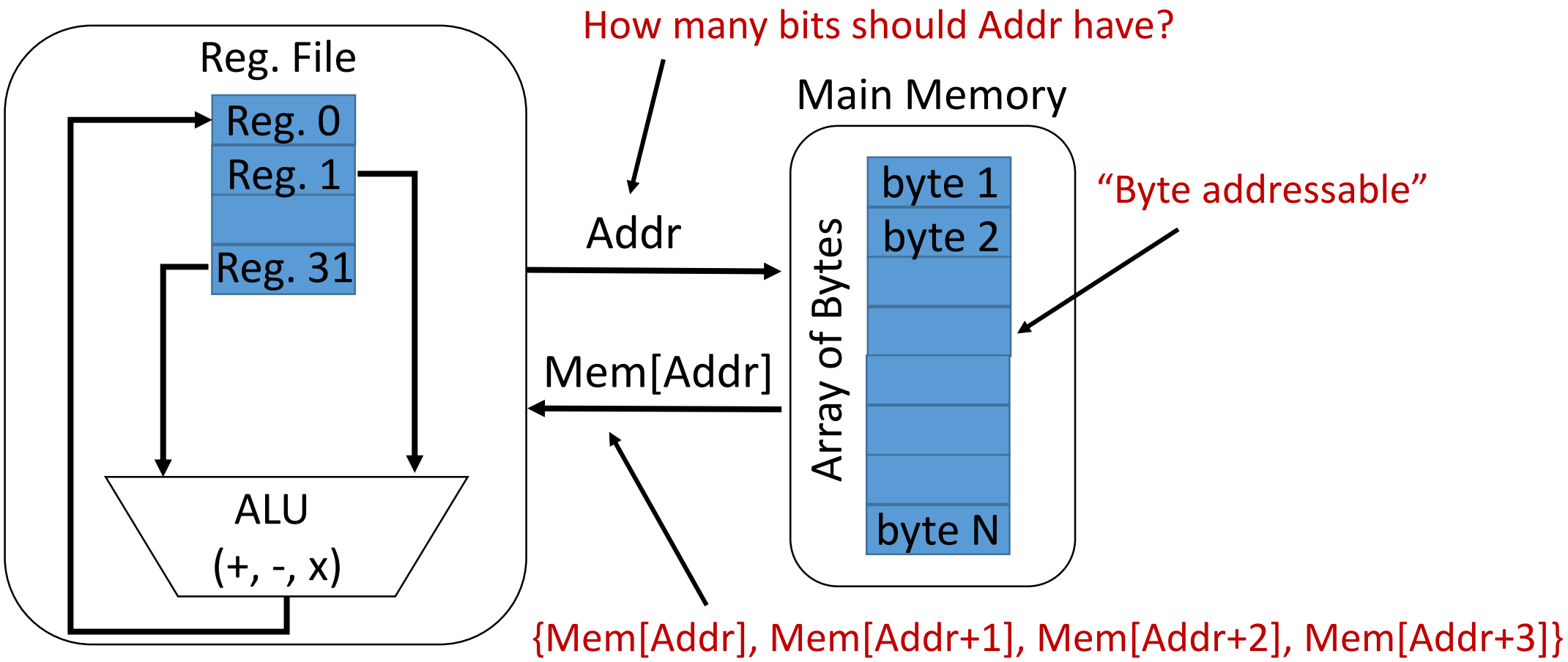
f = ((x * y) + w)*z

and store the result in memory location &f.

Assume that the *add* and *mult* operations are available.

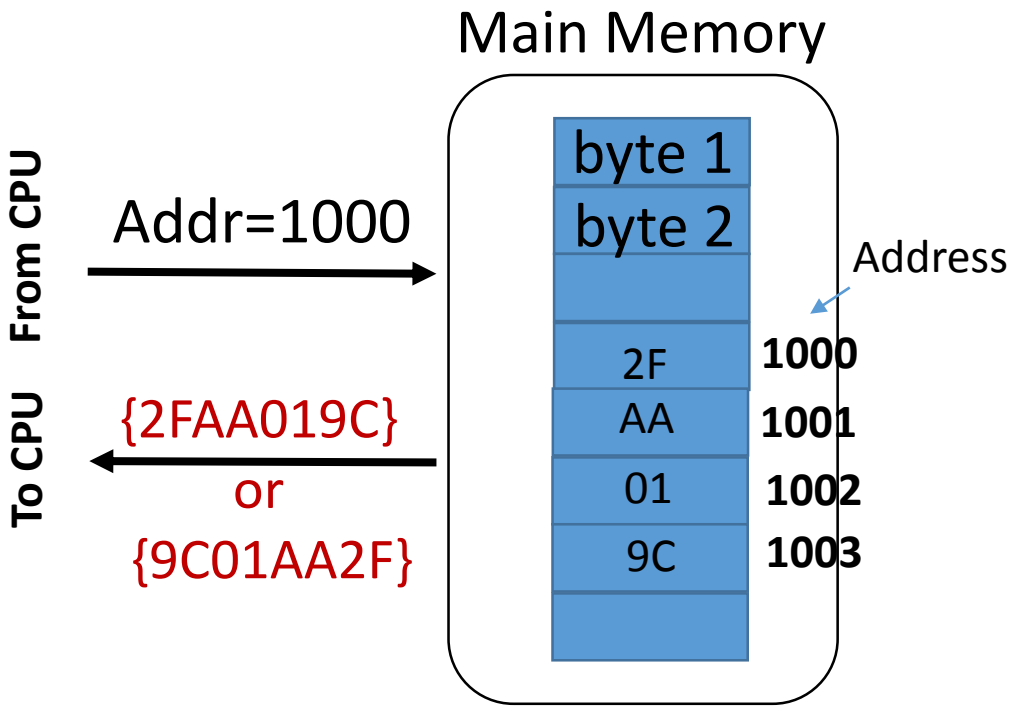
```
push &x
push &y
mult   //$TOS = x*y
push &w
add //$TOS = (x*y) + w
push &z
mult //$TOS = ((x*y) + w)*z
pop &f
```

How is Memory Accessed



- Memory can also be bit-addressable, 32-bit addressable, 64-bit addressable, etc.

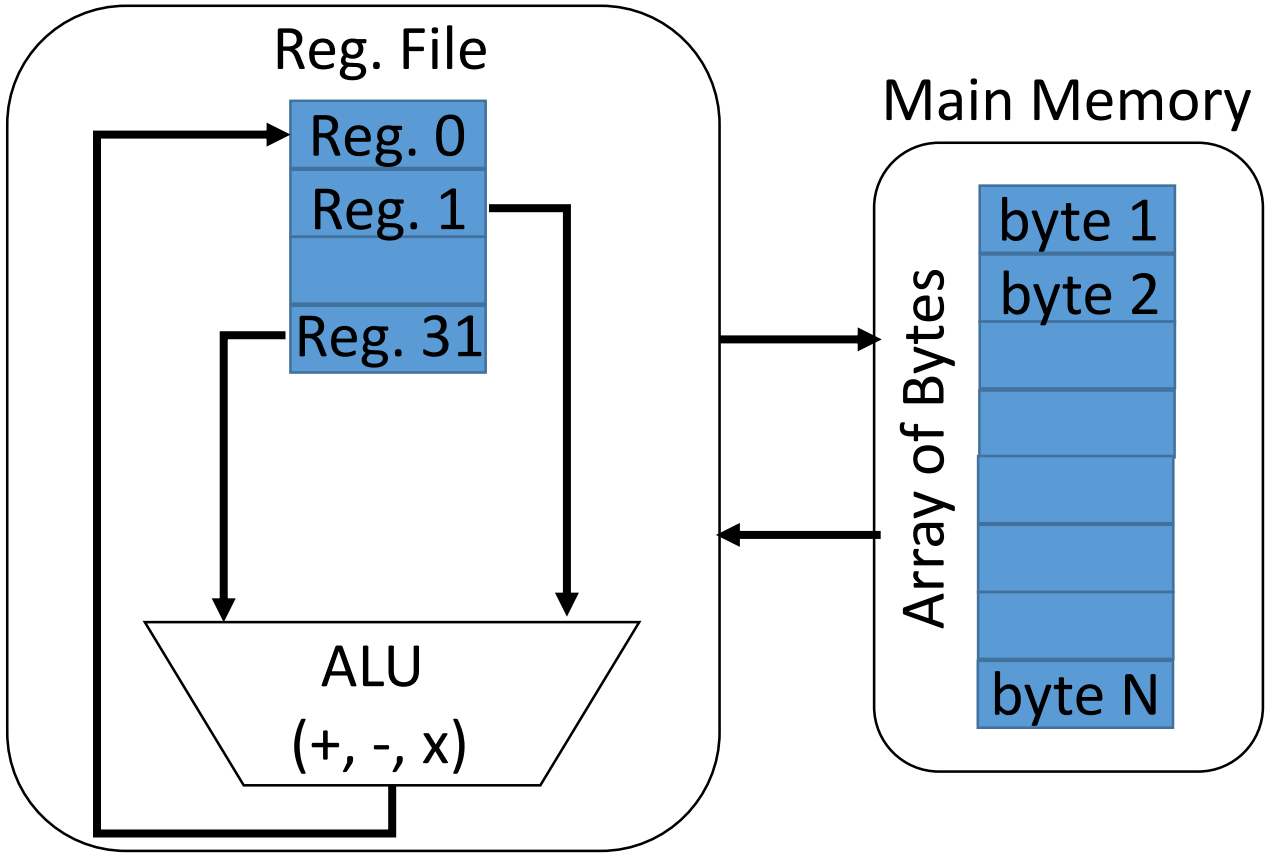
Little/Big Endian



- Assuming an instruction or data word is 32 bits of data from locations [Addr,Addr+3] are fetched from Main Memory.
- Big Endian: Data = {Mem[Addr], Mem[Addr+1], Mem[Addr+2], Mem[Addr+3]}
  - Examples: IBM360
- Little Endian: Data = {Mem[Addr+3], Mem[Addr+2], Mem[Addr+1], Mem[Addr]}
  - Examples: x86, x86-64, AMD64
- Bi-Endian: both supported! (Ex: MIPS)

Load-Store ISA

- Instructions like *add*, *sub* etc. operate *only* on registers (examples: MIPS, ARM, 3/2/1/0-Address ISAs we have seen so far)
  - Separate Load/Store instructions to fetch data from memory into registers and to write data back from registers to memory



add rs, rt, rd
($rd ←$rs + $rt)

load rt, M
($rt ←Mem[M])

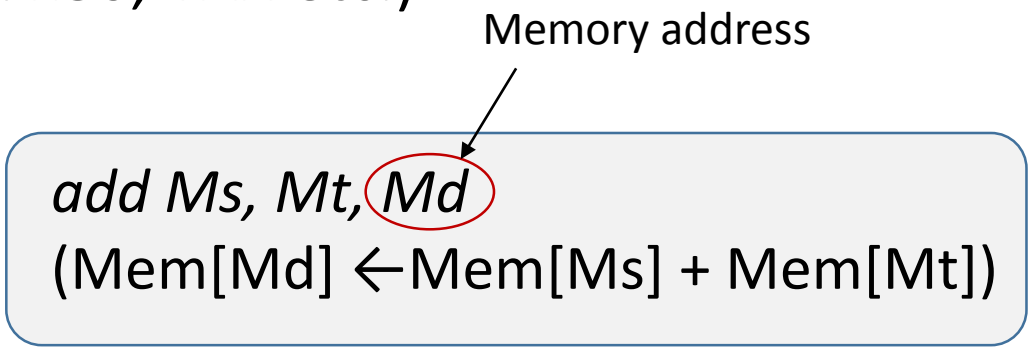
Load data at memory location M into register rt

store rt, M
(Mem[M] ←$rt)

Store data in register rt to memory location M

Memory-Memory ISA

- Allow instructions like add/sub to operate *directly* on data in memory (Examples include: x86, VAX etc.)



- Most generally, each of the two operands and the result can be read-from/written-to either memory or register file
  - Pros?
  - Cons?

Addressing Modes

- How to specify where to read from/write to

Absolute

load rt, M
($rt ←Mem[M])

Directly provide memory address

Register Indirect

load rs, rt
($rt ←Mem[$rs])

Memory address from register rs

Displaced/Based

load rs, rt, offset
($rt ←Mem[offset+$rs])

Add an offset to address stored in register rs

Indexed

load rs, rt, rd
($rd ←Mem[$rs + $rt])

Base memory address and offset from registers

Memory Indirect

load rt, rs
($rt ←Mem[Mem[$rs]]

Use value at Mem[$rs] as address

Auto-increment

load rt, rs
($rt ←Mem[$rs]
$rs ←$rs + 1)

Same as Register indirect + value in rs is automatically incremented

- Orthogonal ISA: each op supports every addressing mode (ex: VAX)

Addressing Modes

Absolute

load rt, M
($rt ←Mem[M])

32-bit instruction and can't possible store a 32-bit address!

Register Indirect

load rs, rt
($rt ←Mem[$rs])

1. Pointers
2. Emulate absolute addressing (more later)

Displaced/Based

load rs, rt, offset
($rt ←Mem[offset+$rs])

Data is frequently stored *relative* to a base location

Indexed

load rs, rt, rd
($rd ←Mem[$rs + $rt])

Memory Indirect

load rt, rs
($rt ←Mem[Mem[$rs]]

Pointer to a pointer!

Auto-increment

load rt, rs
($rt ←Mem[$rs]
$rs ←$rs + 1)

Array accesses (rs stores pointer to an array)

What Makes a Good ISA

- Ease of hardware implementations
  - Decoder for fixed vs. variable instruction lengths
  - Simple vs. complex addressing modes

- Ease of software implementation
  - Can the programmer/compiler use the ISA easily?
  - How many assembly instructions to represent a single line of code in a high-level language? “semantic gap”

- Backwards compatibility
  - ISA will have to be supported into the future; new instructions can be added but existing instructions cannot be removed

CISC Vs. RISC

- CISC: “Complex” Instruction Set Computer
- RISC: “Reduced” Instruction Set Computer
  - Each CISC instruction = multiple RISC instructions

- CISC vs. RISC debate dominated the 80s/early 90s but is largely resolved now
  - CISC ISAs like x86 are “broken down” internally into simpler RISC-like instructions (microcode)
  - x86 is “externally CISC but internally RISC”
  - Other popular ISAs like ARM and MIPS are RISC

ISA Comparison

	VAX	x86	MIPS
ISA Type	CISC	CISC (but internally RISC)	RISC
Instruction Length	Variable	Variable	Fixed (32 bits)
Addressing Modes	• 3-Address • Memory-memory • Orthogonal addressing modes	• 2-address • Register-memory	• 3-Address • Load/store ISA

In-Class Problem 2

We will explore the advantages of

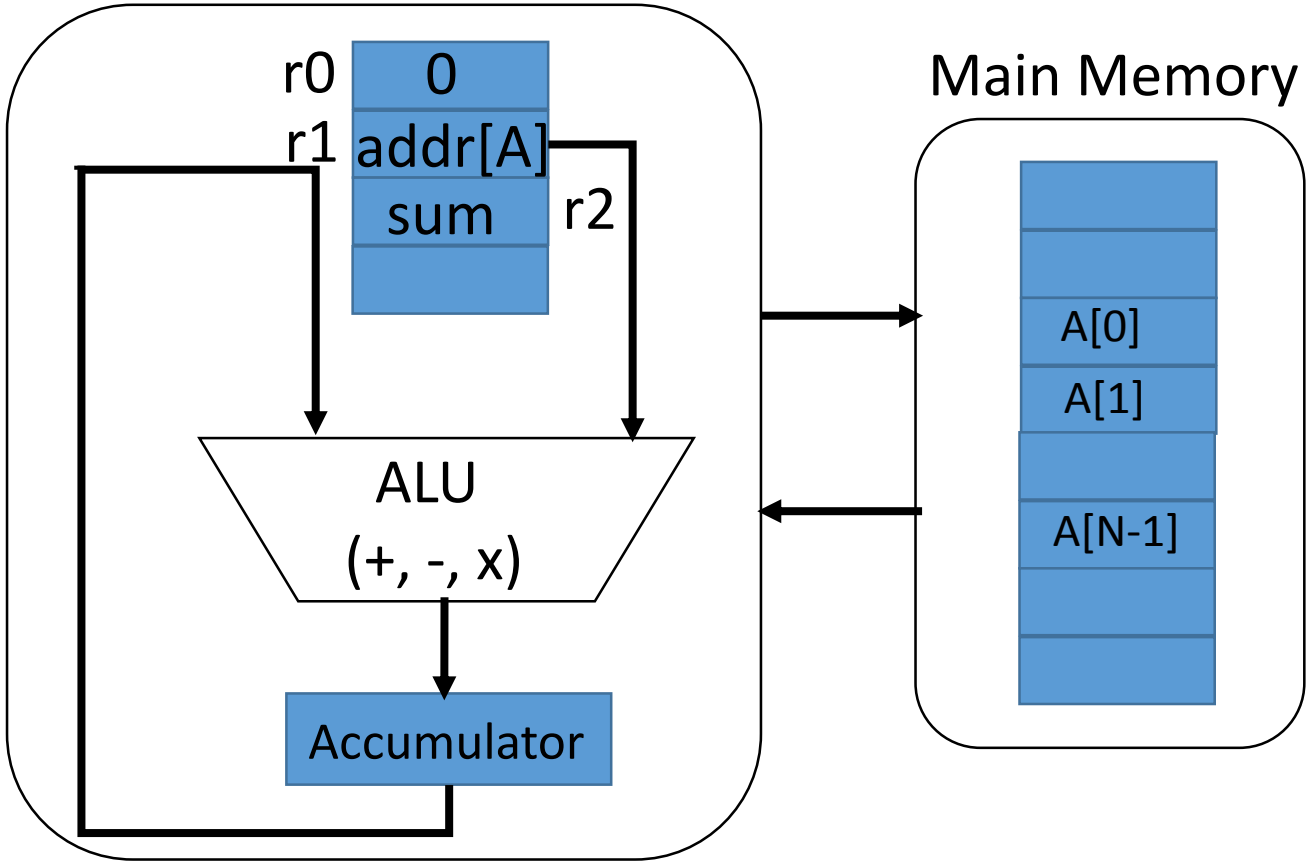
(i) The auto-increment addressing mode

(ii) Orthogonal ISA

mvto rs
($rs ←$acc)

mvfrm rs
($acc ←$rs)

add rs
($acc ←$acc + $rs)



addinc rs, rt
($acc ←$acc + Mem[$rs + $rt]
$rs ←$rs + 1)

ble rs, imm, addr
(if ($rs<=imm)
goto addr)

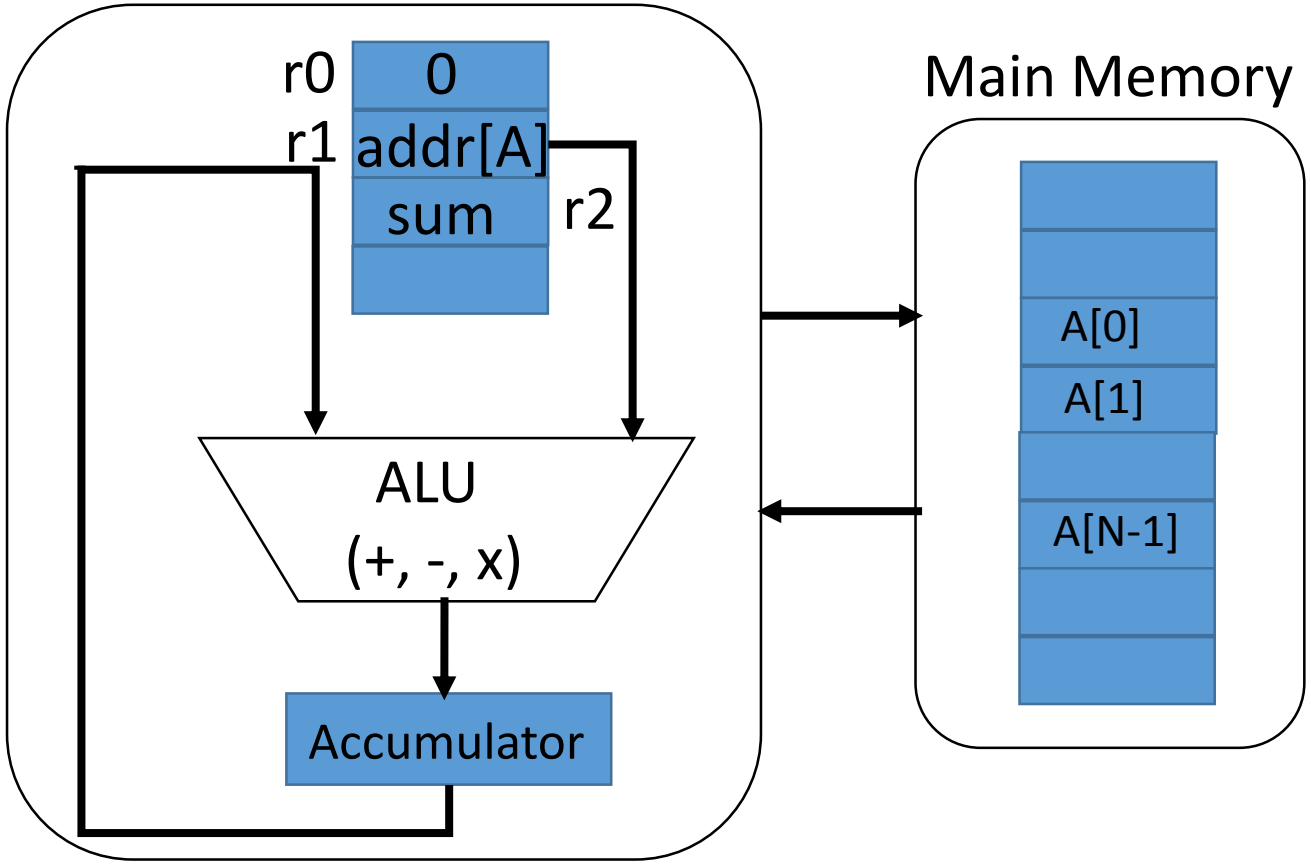
Write code to sum the array and store the sum in register r2. You can assume the acc is initialized to zero.

In-Class Problem 2

We will explore the advantages of

(i) The auto-increment addressing mode

(ii) Orthogonal ISA




```
mvfrm r0
L: addinc r0, r1
ble r0, N-1, L
mvto r2
```

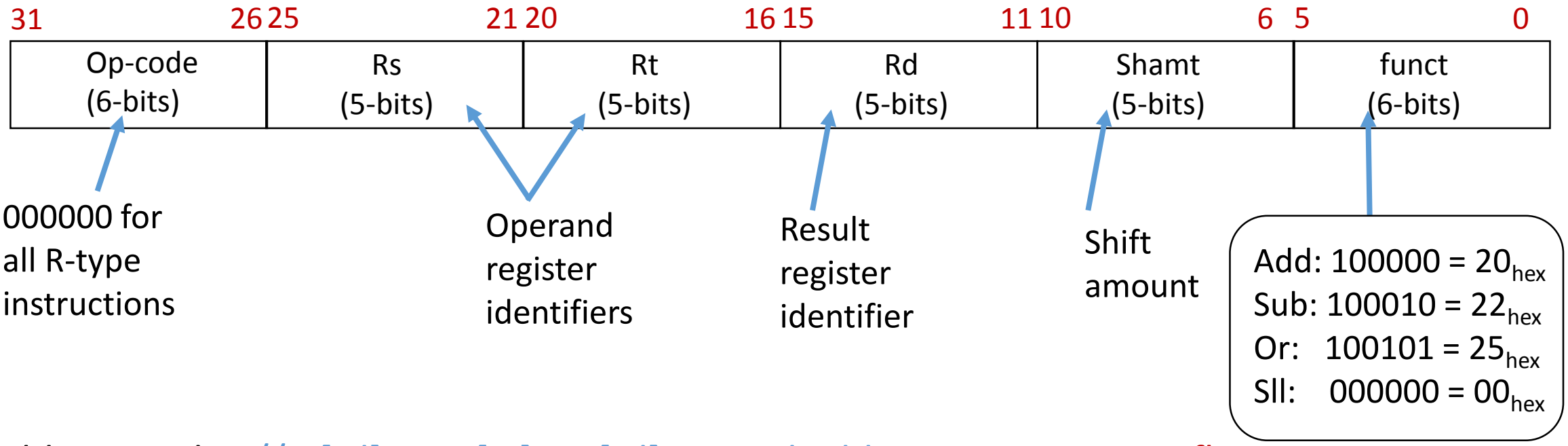
MIPS ISA Overview

- 32-bit ISA (what does this mean?)
  - Instruction length: 32 bits
  - Data word length: 32 bits

- 32 general purpose registers ($0, $1, $2, … $31); 32-bits each
  - Register $0 is hard-wired to 0 (constant)
  - All other registers can be used as read/write
  - But, certain registers dedicated for specific purposes by compiler
    - Global pointer ($28), Stack pointer ($29), Frame pointer ($30), Return Address ($31)
    - We wont worry about these conventions in this class!

- Two dedicated 32-bit registers *hi* and *lo* for multiplication ops
- Separate regs/unit for floating point ops (discussed later)

MIPS Instructions: R-Type



add rs, rt, rd // R[rd] ← R[rs] + R[rd]; signed addition; trap on overflow

sub rs, rt, rd // R[rd] ← R[rs] - R[rt]; signed subtraction; trap on overflow

or rs, rt, rd // R[rd] ← R[rs] | R[rt]; bit-wise Boolean OR operation

sll rt, rd, shamt // R[rd] ← R[rt] << shamt; logical shift left

…..(many others)

MIPS R-type Instructions: Add vs Addu

add rs, rt, rd // R[rd] ← R[rs] + R[rd]; signed addition; trap on overflow

- rs, rt and rd are assumed to contain 32-bit signed, 2's complement numbers
  - MSB (32nd bit) indicates if the number is positive or negative
  - Review 2's complement representation (https://en.wikipedia.org/wiki/Two%27s_complement)

- Example: assume only 4-bit numbers in range [-8,7]

rs = 0101 (5)

rt =+ 0110 (6)

rd= 1011 (-4) → Overflow!

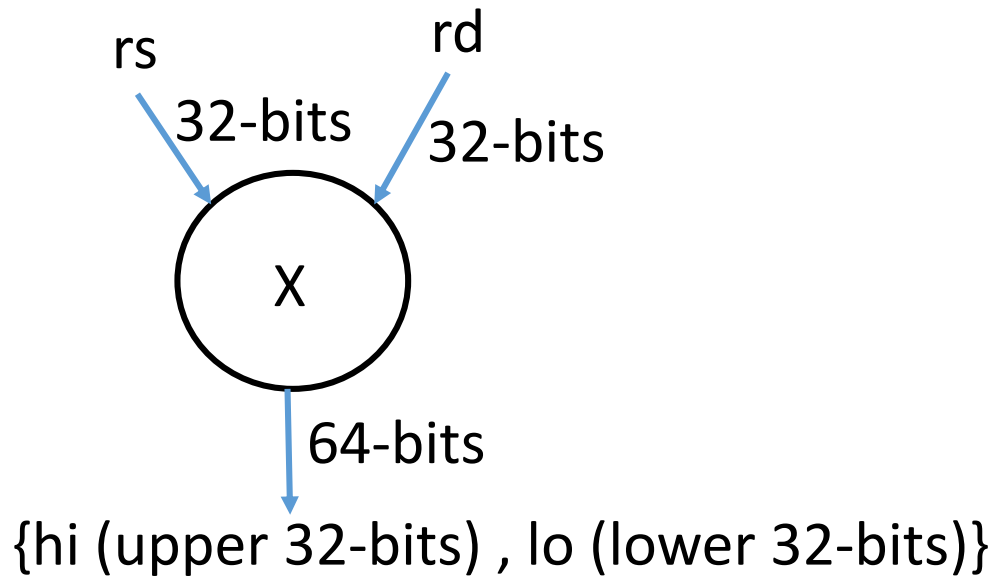
addu rs, rt, rd // R[rd] ← R[rs] + R[rd];

NO trap on overflow

MIPS R-type Instructions: Mult

mult rs, rt // {hi,lo} ← R[rs] + R[rd]; signed multiplication

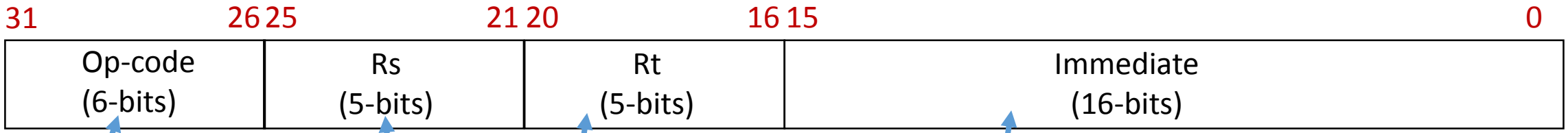
- *hi* and *lo* are two *special purpose 32-bit registers* that store the result of a multiplication operation
  - Why? Multiplication result of two 32-bit values needs at least 64 bits



What does this remind you of?

mvhi rd //R[rd] ← R[hi]
mvlo rd //R[rd] ← R[lo]

MIPS Instructions: I-Type



addi: 8_{hex}
beq: 4_{hex}
ori: 25_{hex}
lw: 23_{hex}

Operand register

Result register ("PC relative")

Immediate operand

addi rs, rt, imm // R[rt] ← R[rs] + {SignExtend, imm}; MSB of imm is extended to 32 bits

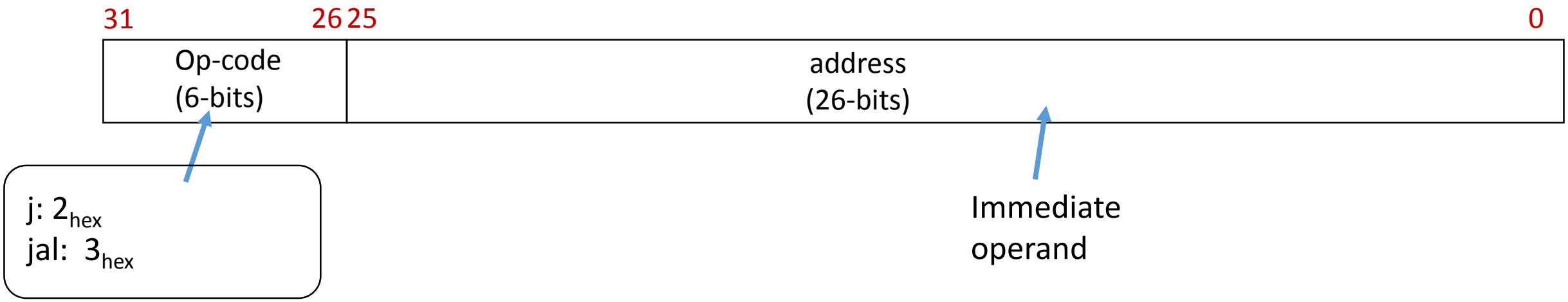
beq rs, rt, imm // if{R[rs] == R[rt]} branch to PC + 4 + imm; ("PC relative")

ori rs, rt, imm // R[rt] ← R[rs] | {ZeroExtend,imm}; bit-wise Boolean OR operation

lw rs, rt, imm // R[rt] ← Mem[{SignExtendimm} + R[rs]] ("Displaced/based")

…..(many others)

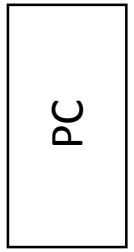
MIPS Instructions: J-Type



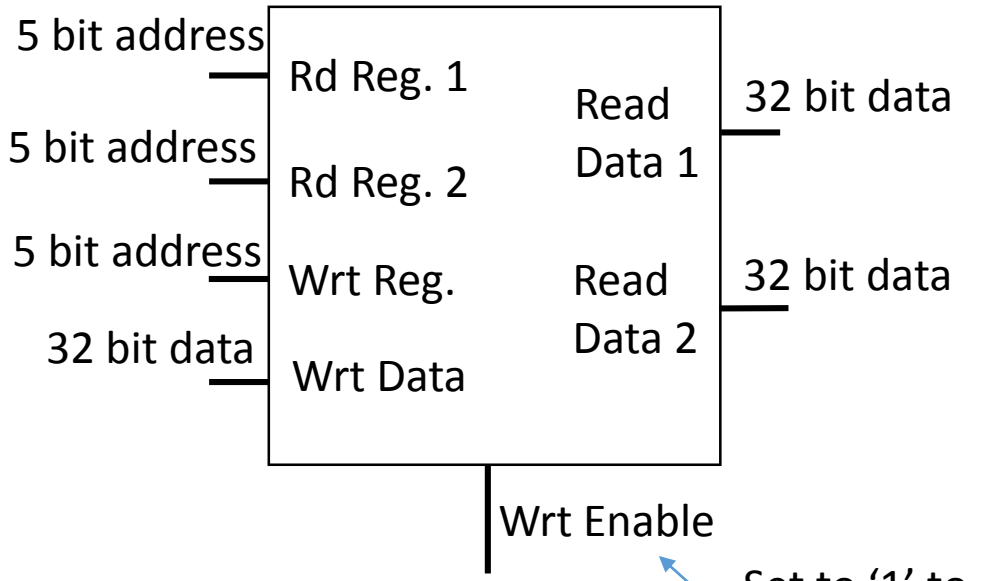

```
j address    // PC ← address
jal address // R[31] ← PC+8; PC ← address;
// Also, R-type jump
jr rs // PC ← R[rs]
```

Implementation of the MIPS ISA

- Hardware building blocks

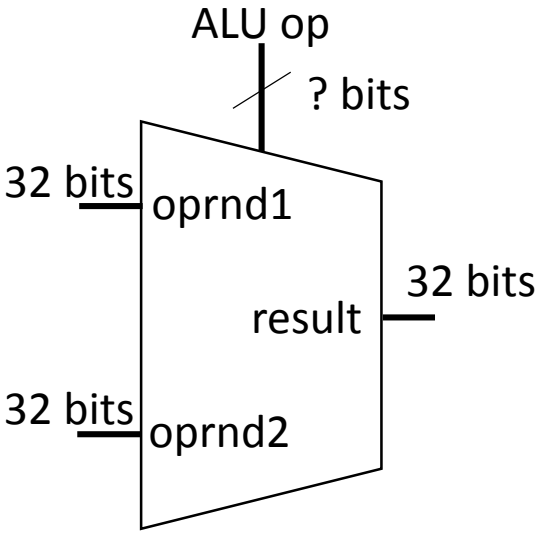


Program Counter
A 32-bit “register”
(can store information)



Register File
32 32-bit registers
(2 Read ports, 1 Write Port)

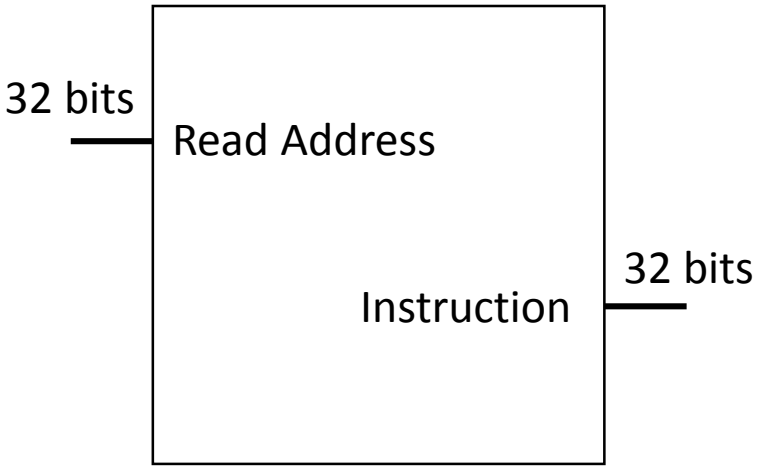
Arithmetic/Logic Unit (ALU)



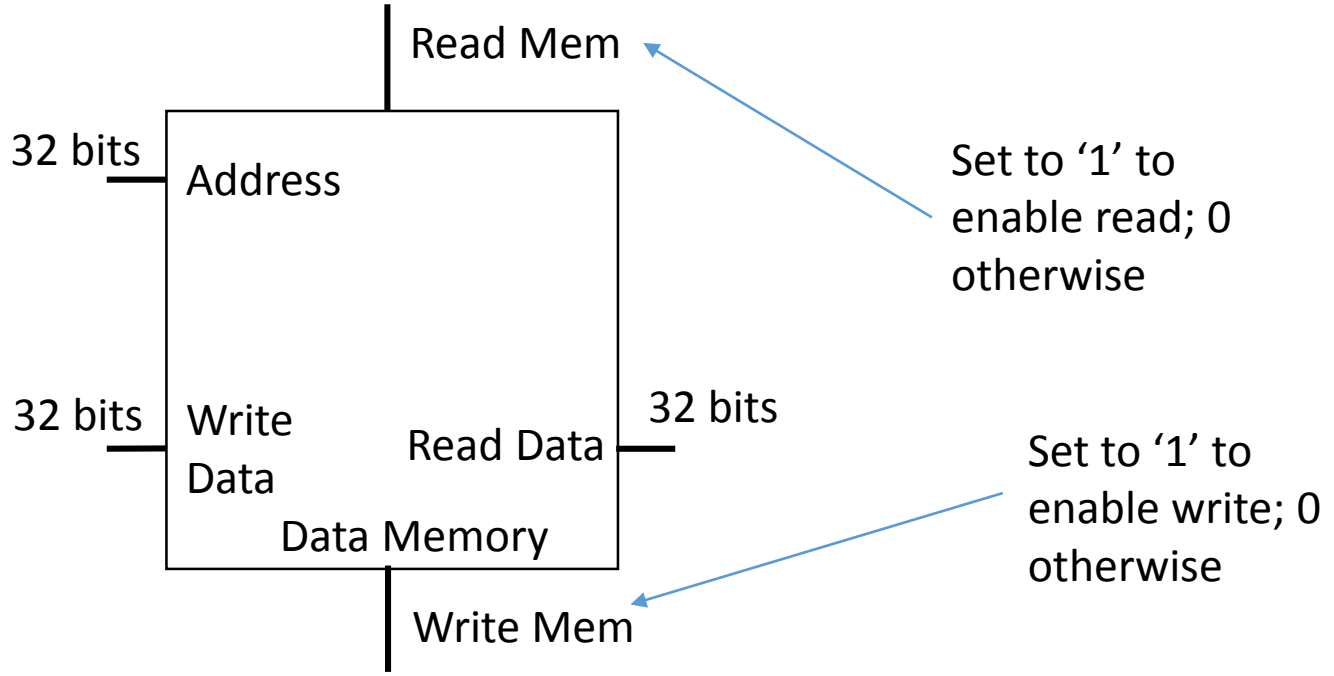
Add: 20_{hex}
Addu: 20_{hex}
Sub: 23_{hex}
Subu: 24_{hex}
And: 25_{hex}
Or: 24_{hx}
Nor: 27_{hex}

Implementation of the MIPS ISA

- Hardware building blocks



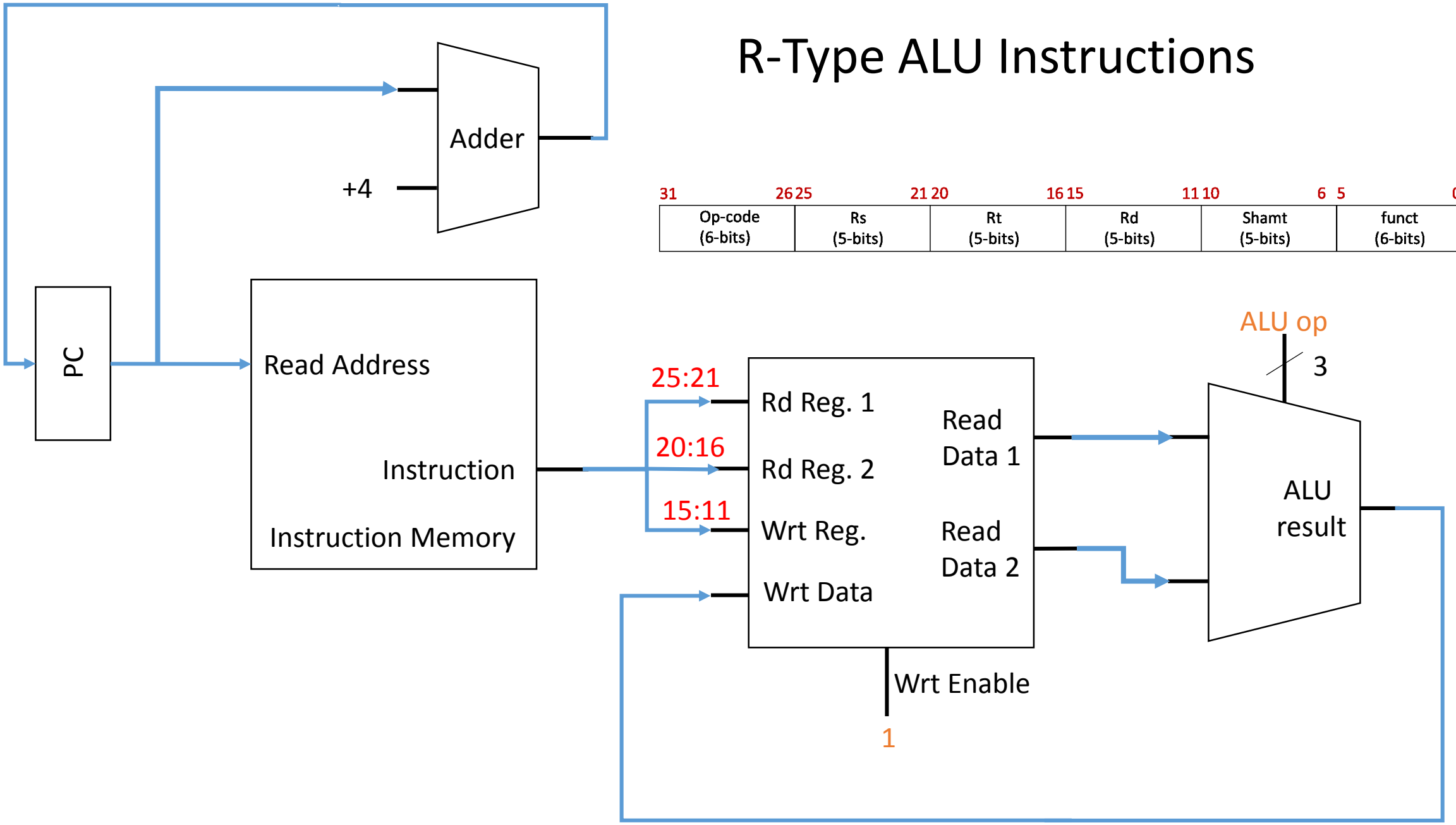
Instruction Memory
(Read only)



Data Memory
(Either read or write, but not both together)

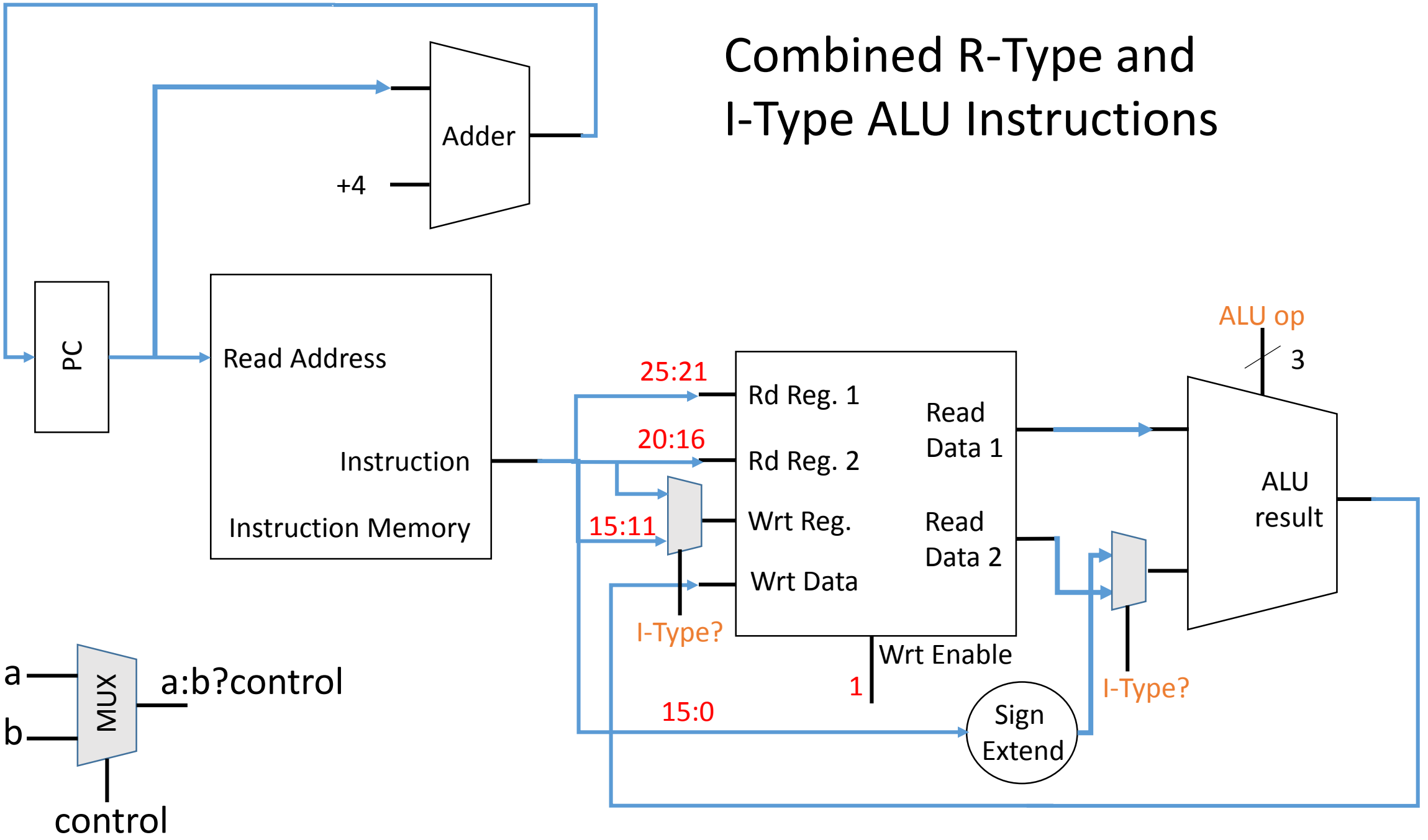
- Is this a von Neumann architecture?

R-Type ALU Instructions



I-Type ALU Instructions

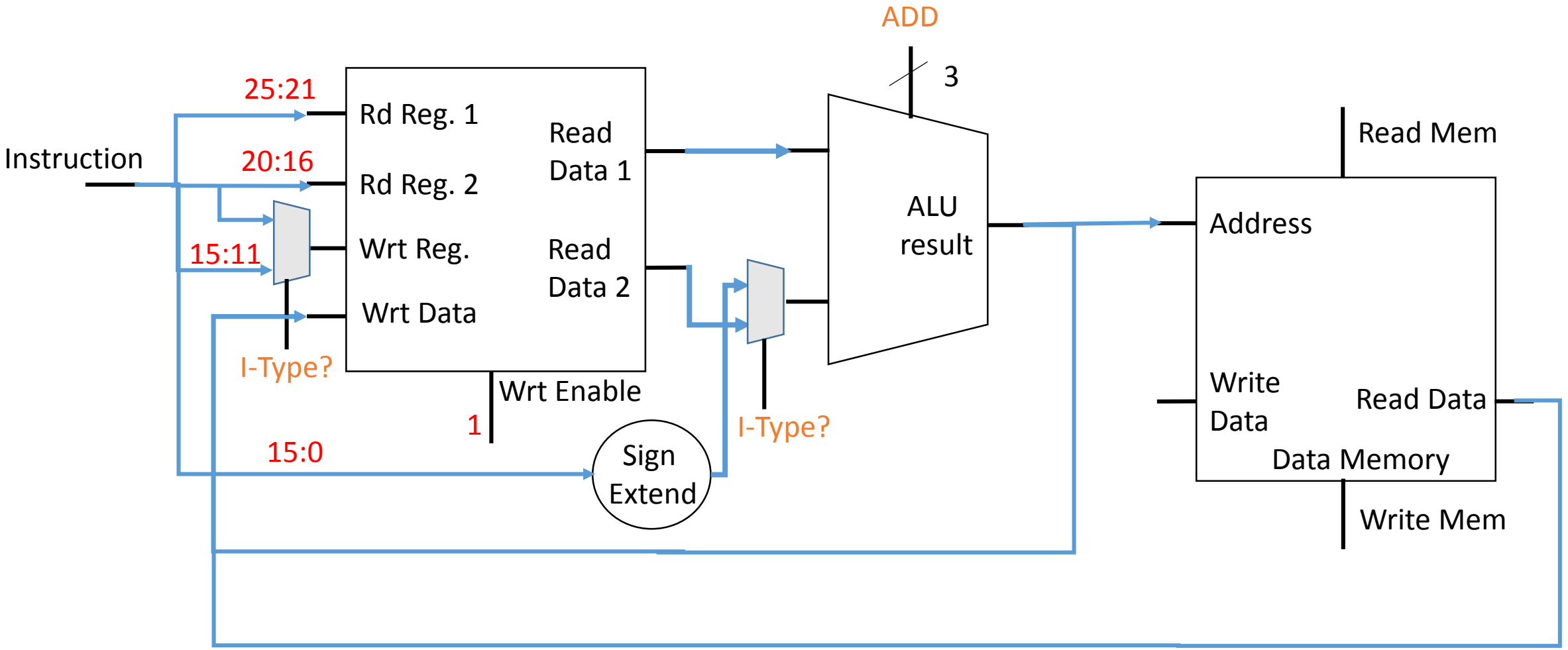
Combined R-Type and I-Type ALU Instructions



I-Type Load Instructions

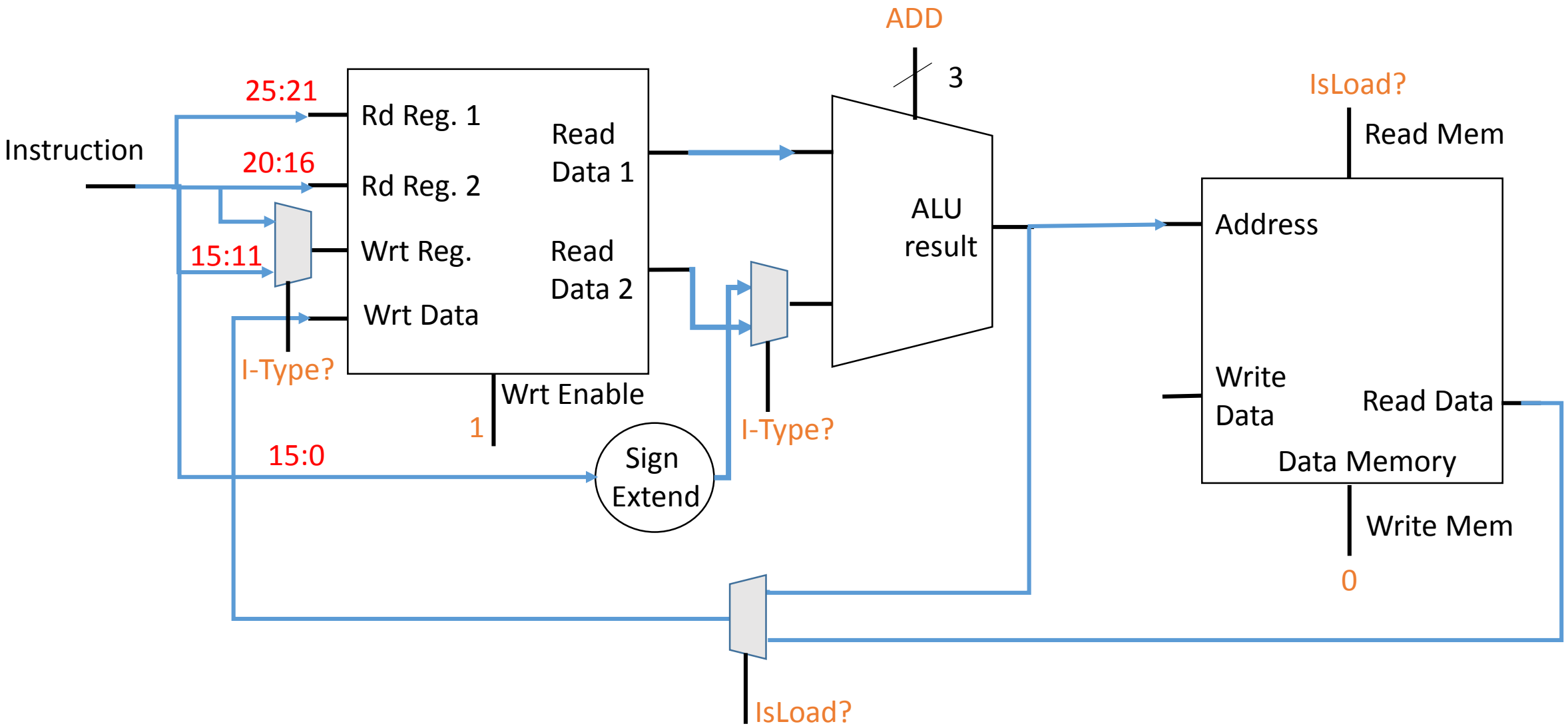
lw rs, rt, imm // R[rt] ← Mem[{SignExtendimm} + R[rs]] ("Displaced/based")

Instruction Fetch Part Not Shown



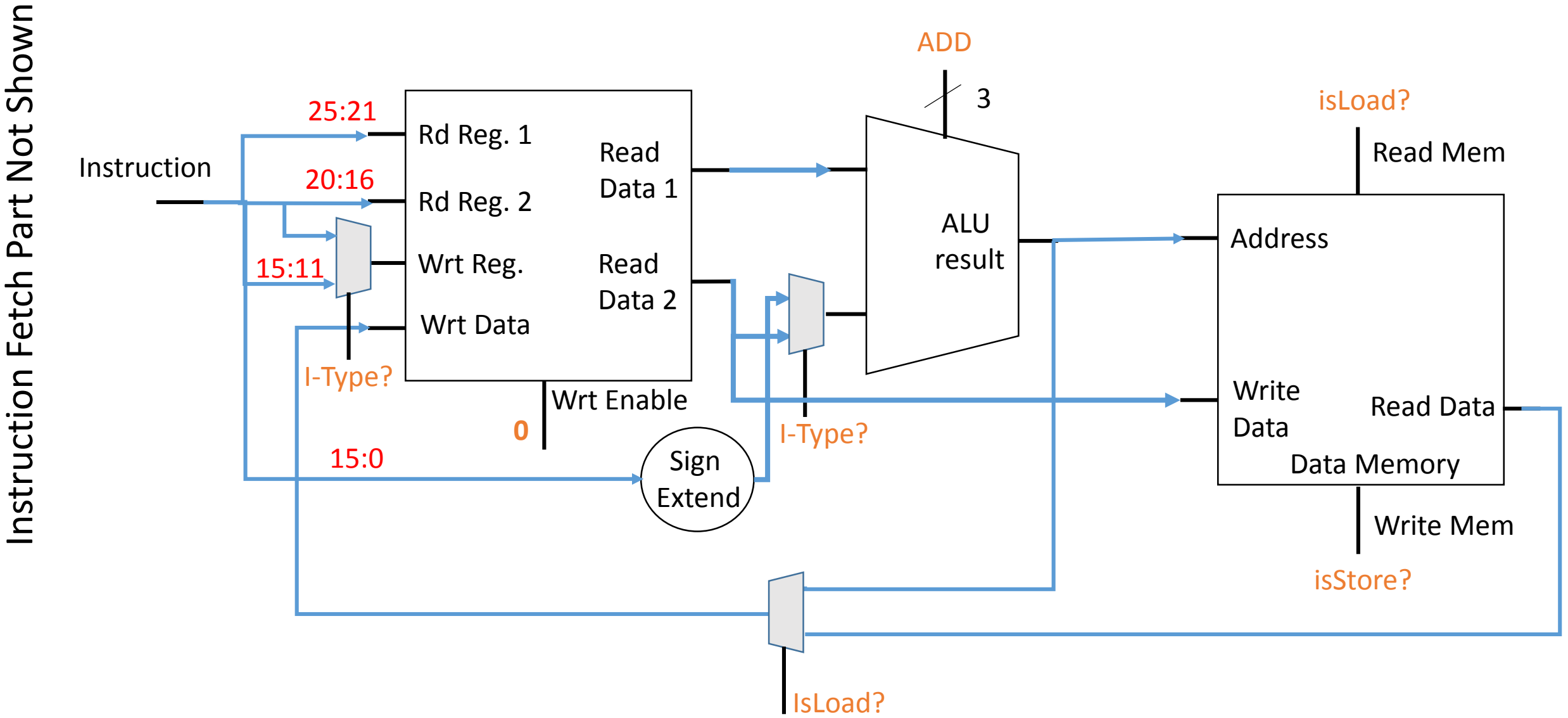
I-Type Load Instructions

Instruction Fetch Part Not Shown



I-Type Store Instructions

Did we miss something?



I-Type Store Instructions