Single Cycle MIPS Datapath

Computer Architecture

ECE 6913

Brandon Reagen

NYU TANDON SCHOOL OF ENGINEERING

What we'll cover today

1) Finish the MIPS calling convention we left off with last time
2) Brief review of the MIPS ISA and key features
3) Implementing a MIPS datapath
4) Release Lab 1! (implement MIPS datapath)

Why do we need conventions?

Let's think about functions

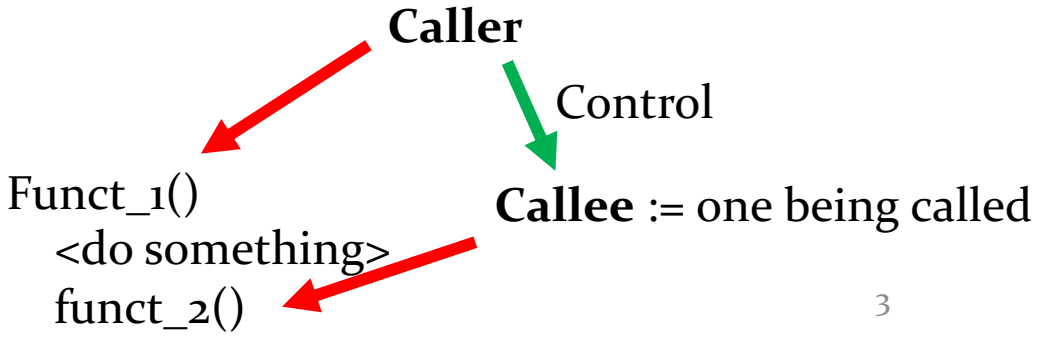
- What happens when you transfer control?
- View of registers verses memory

MIPS calling conventions

- We will be using a simplified convention
- The full convention is excessively detailed
  - All core concepts provided here
- A **great** description can be found here
  - https://sites.cs.ucsb.edu/~kyledewey/cs64w16/documentation/calling_convention/calling_convention.html

Caller: function calling another function

Callee: function being called



Supporting functions: Jumps and $ra

$ra: return address

- Special register to store next instruction of caller
  - What does that mean? Tells us how to get home!

Jump instructions help us call functions

- jal: “jump and link”
  - Will branch to specified location AND store PC+4 in $ra (i.e., R[31])
- jr: “jump register”
  - Will jump to location specified at register, like $ra!

We can use jal & jr to get in and out of functions

Supporting functions: arguments

How do we get arguments to functions?

- Use reserved registers $a0 - $a3
- To use:
  - Caller loads values into $a0 - $a3
  - Caller immediately jumps to function
  - Callee can read $a0 - $a3, will have correct values!
  - < do something >
  - Callee jumps back to Caller using: jr $ra

Note: when control returns to caller, $a0-$a3 may have changed

- This is the whole point of the convention

Supporting functions: return values

How do we get values back from function calls?

- $v0 - $v1 are reserved for returning values from functions
- In callee, can load $v0-$v1 with values
- When control returns to caller, can trust return values are there

Supporting functions: temporary registers

What happens if we need more than $a* and $v*?

- $t0 - $t9 can be used
- "All bets are off" between function calls
  - Callee assumes nothing about initial $t* values
  - Caller assumes nothing about values of $t* when control returns

Supporting functions: saved registers

What if you're in the middle of a computation and need a function?

- Values in $s* are preserved when control returns
  - Caller uses $s1 then calls a function
  - Callee first saves value of $s* in *memory*
  - Callee does its compute
  - Callee restores $s* values
  - Callee returns control
  - Caller assumes control, value of $s* is same as before function call

Supporting functions: saving registers

Register values are preserved in memory using the “stack”

- Recall: $sp register tracks the “top” of the stack (grows?)
- Functions can write register values to stack, then increment $sp
- When one calls another, it won’t look “backwards”
  - Can.. But that’s why conventions are important!
- When functions return control, they reset $sp to where caller left off

This is super important, why?

- Think about $ra

Contrived example

```
Initialize      li $s0, 7
                li $s1, 3
Some compute    sub $s2, $s0, $s1
Jump prep       move $a0, $s0
                move $a1, $s1
Jump!           jal compute_f1
Get return val  move $s3, $v0
Compute..       slt $s4, $s2, $s3
```

Q: what happens if a function calls another function..?

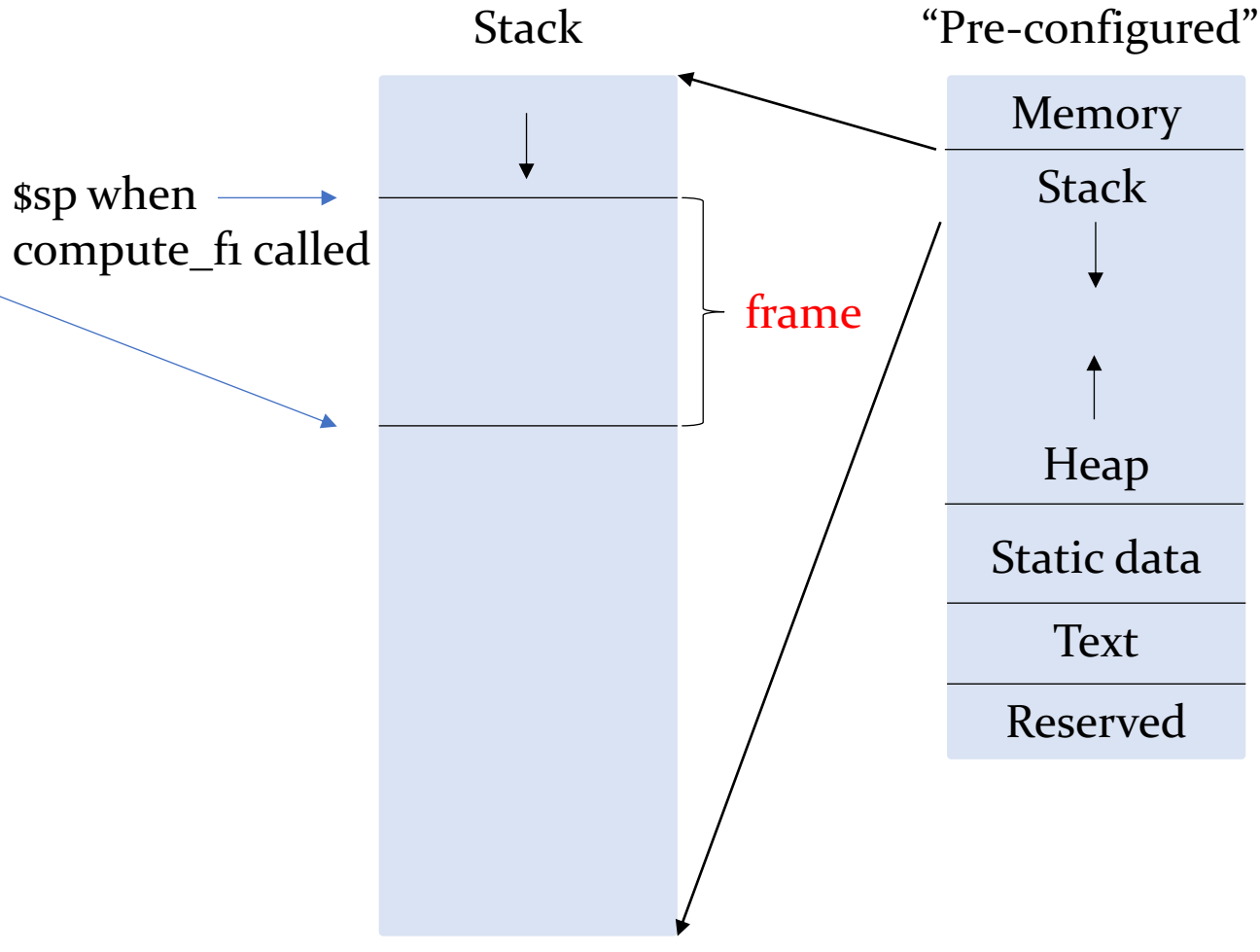
```
compute_f1:
Procedure prep:
save all used s* regs
                addi $sp, $sp, -12
                sw $s0, 8($sp)
                sw $s1, 4($sp)
                sw $s2, 0($sp)
Free to compute!
                add $s0, $a0, $a1
                addi $s1, $a1, 1
                sub $s2, $a0, $s1
                mult $s2, $s1
Return prep:
save return value
restore s* regs
restore sp!
                mflo $v0, $lo
                lw $s0, 8($sp)
                lw $s1, 4($sp)
                lw $s2, 0($sp)
                addi $sp, $sp, 12
Return control!
(Jump)          jr $ra
```

Visualizing the stack

```
compute_f1:
        addi $sp, $sp, -12
        sw $s0, 8($sp)
        sw $s1, 4($sp)
        sw $s2, 0($sp)
        .....
        lw $s0, 8($sp)
        lw $s1, 4($sp)
        lw $s2, 0($sp)
        addi $sp, $sp, 12
```

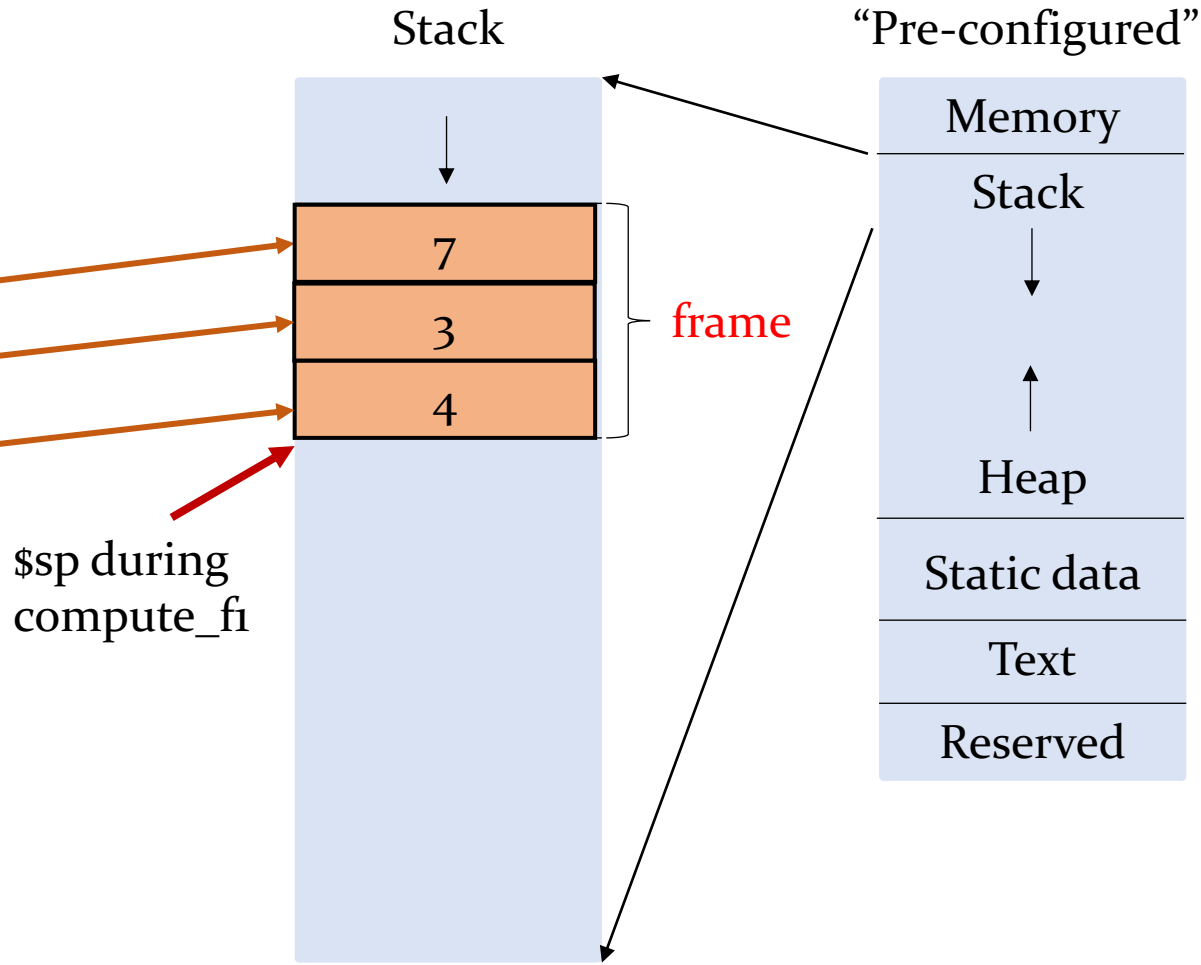
Setup

Teardown



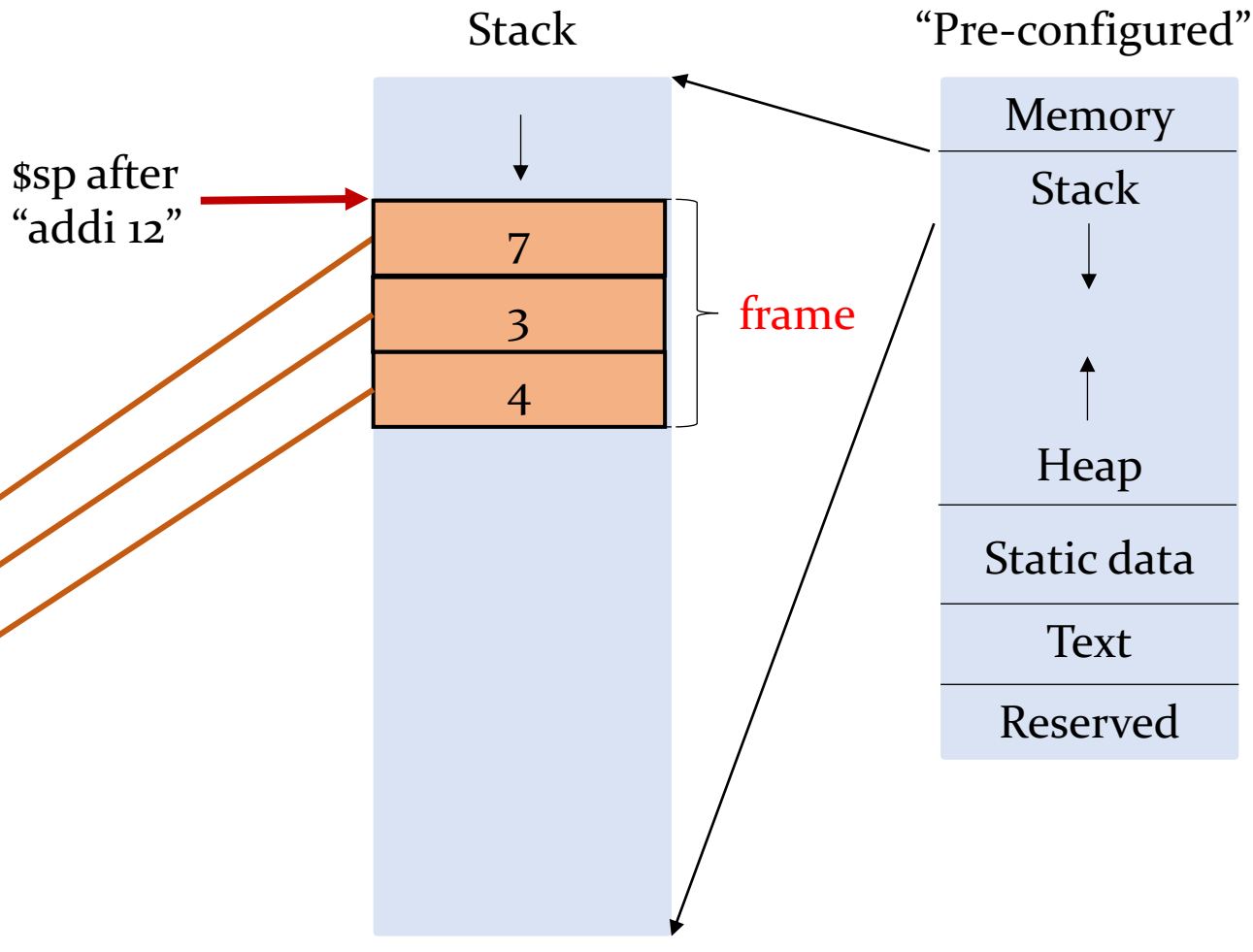
Visualizing the stack

```
compute_f1:
    Setup
        addi $sp, $sp, -12
        sw $s0, 8($sp)
        sw $s1, 4($sp)
        sw $s2, 0($sp)
        .....
    Teardown
        lw $s0, 8($sp)
        lw $s1, 4($sp)
        lw $s2, 0($sp)
        addi $sp, $sp, 12
```



Visualizing the stack

```
compute_f1:
        addi $sp, $sp, -12
Setup   sw $s0, 8($sp)
        sw $s1, 4($sp)
        sw $s2, 0($sp)
        .....
        lw $s0, 8($sp)
Teardown lw $s1, 4($sp)
        lw $s2, 0($sp)
        addi $sp, $sp, 12
```



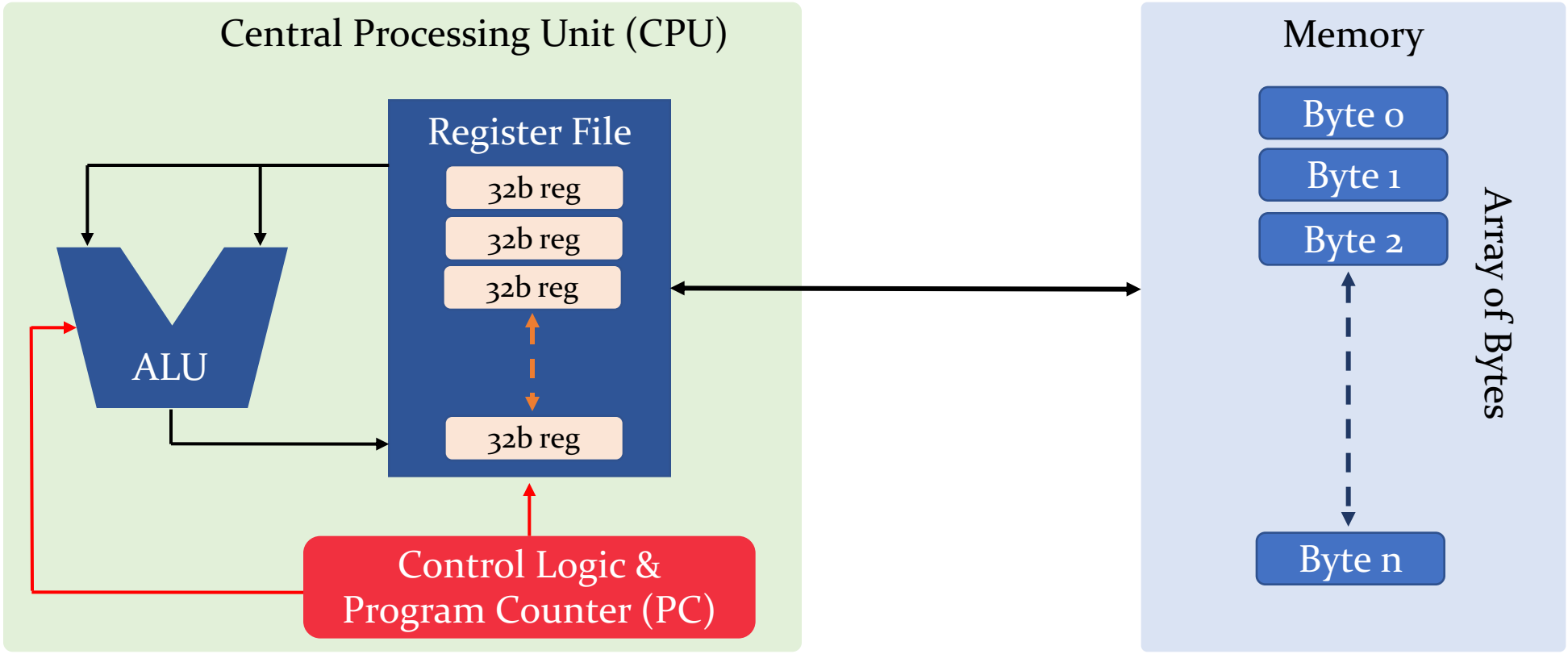
Q: what happens if a function calls another function..?

The MIPS register convention

$zero:	hardwired to zero	(register 0)	
$at:	assembler temporary	(1)	
$v0, $v1:	function return values	(2, 3)	
$a0 - $a3:	function arguments	(4-7)	
$t0 - $t9:	temporaries	(8-15, 24, 25)	[Can be overwritten by callee]
$s0-$s7:	saved	(16-23)	[Callee must save/restore before call]
$gp:	global pointer for static data	(28)	
$sp:	stack pointer	(29)	
$fp:	frame pointer	(30)	
$ra	return address	(31)	

*26-27 are kernel regs, we won't use them.

An abstract computer



MIPS Summary

32b RISC architecture
- Fixed instruction length
- 32b registers and memory addresses
- Byte addressable memory.. How much?

Load-store architecture
- All compute use values from registers
- Only special instructions access memory

32 general purpose registers
- We will assume the simplified calling convention
- Program counter for finding next instruction
- Hi/lo registers for multiplication

What makes a good ISA?

Simplicity favors regularity
- Instruction size, instruction format, data format
- Makes hardware implementation cleaner

Smaller is faster
- Fewer bit to move, write, and read per instruction
- Register file is faster than memory

Make the common case fast
- Constants tend to be small, immediate field optimized for this

Good design demands compromise
- Special formats for important exception
- E.g., jumping far away (as we saw)

Instruction types: Overview

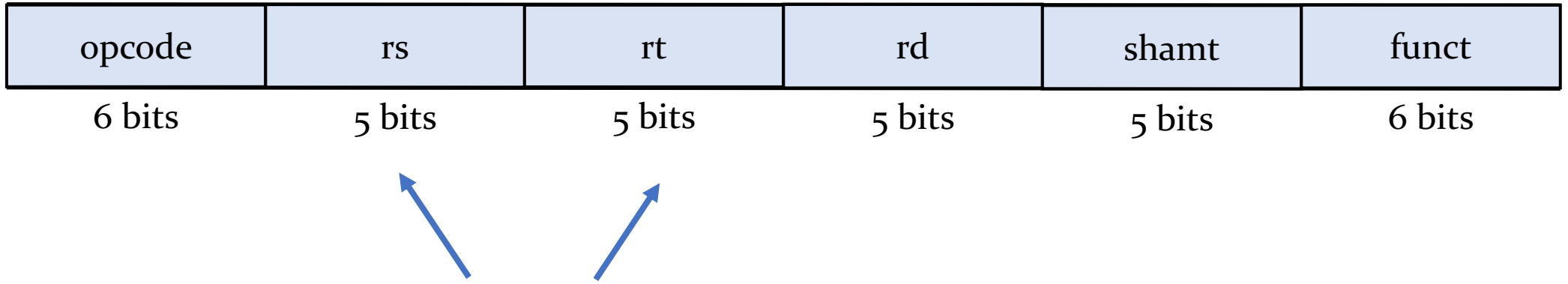
Types of MIPS instructions

- Compute:
  - Arithmetic
  - Logical
- Memory/Data transfer
  - Load
  - Store
- Control:
  - Conditional (branch)
  - Unconditional jump

MIPS has 3 instruction representations

- Register format
- Immediate format
- Jump format

MIPS Instructions: R-Type



opcode: denotes operation and function

rs & rt: Source operand registers

rd: Destination operand register

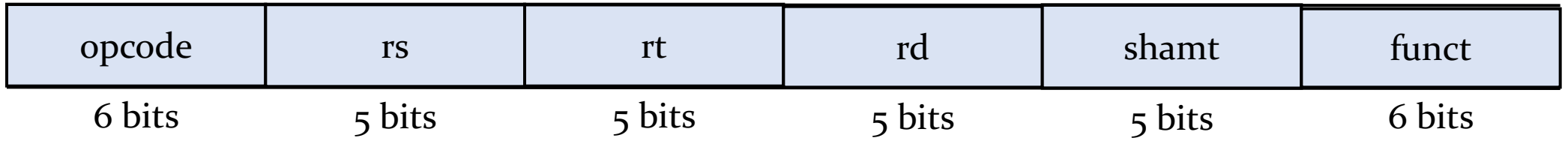
Shamt: Shift amount (sll, srl)

funct: Sub-opcode identifier

Note: oooooo Opcode for all R-type
Use "funct" field to specify add/sub…
Slight simplification earlier

Add: 10 0000 = 20_{hex}
Sub: 10 0010 = 22_{hex}
Or: 10 0101 = 25_{hex}
Sll: 00 0000 = 00_{hex}

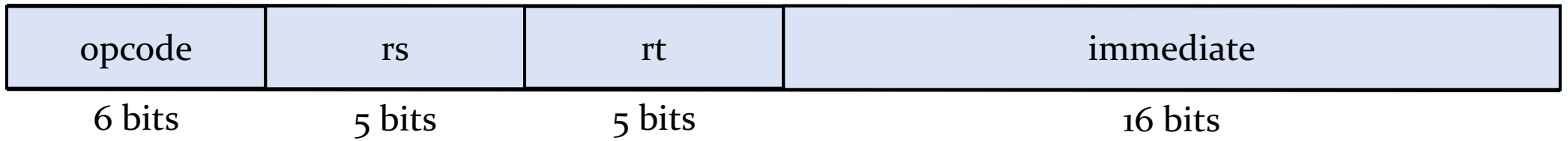
MIPS Instructions: R-Type




```
add     rd, rs, rt        // R[rd]  ← R[rs] + R[rt]; signed addition
sub     rd, rs, rs        // R[rd]  ← R[rs] - R[rt]; signed subtraction
or      rd, rs, rt        // R[rd]  ← R[rs] | R[rt]; bit-wise Boolean OR operation
sll     rd, rt, shamt     // R[rd]  ← R[rt] << shamt; logical shift left
mult    rs, rt            // {hi, lo} ← R[rt] * R[rs]; multiply rt, rs; access result with mfli $r1, mflo $r2
```

Note: *Machine language* register order doesn't match assembly language!

MIPS Instructions: I-Type

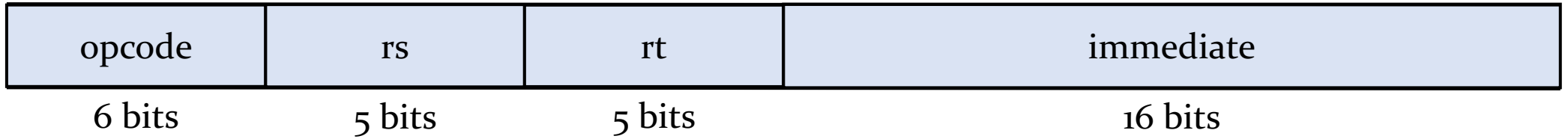


opcode:
denotes operation and function

rs & rt:
Source operand registers

immediate:
Constant or address

MIPS Instructions: I-Type



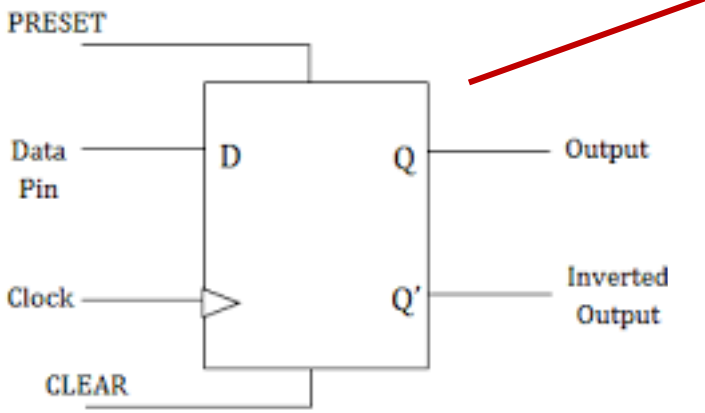

```
addi    rs, rt, imm     // R[rt]  ← R[rs] + {SignExtend, imm}; MSB of imm is extended to 32 bits
ori     rs, rt, imm     // R[rt]  ← R[rs] | {ZeroExtend, imm}; bit-wise Boolean OR operation

beq     rs, rt, imm     // if{R[rs] == R[rt]} branch to PC + 4 + BranchAddress;  ("PC relative")
                        // Else go to PC+4
                        // BranchAddress = {SignExtend, imm, 00}

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

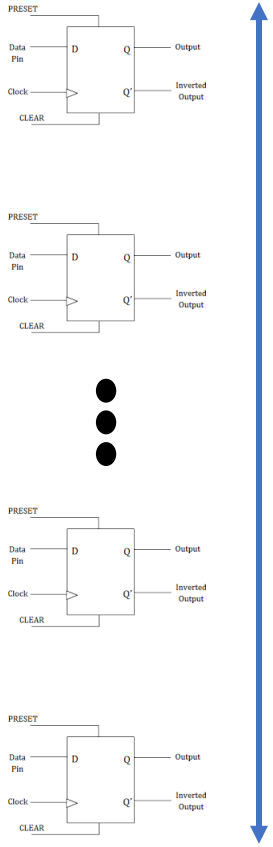
MIPS Instructions: J-Type


```
                    // JumpAddress = { (PC+4)[31:28], address, 2'b0 }
                    // (Pseudodirect)
j       address     // PC ← JumpAddress
jal     address     // R[31] ← PC+4; PC ← JumpAddress;
```

Branch instructions verses jump?

Microarchitectural building blocks: register





32 flip-flops

PC

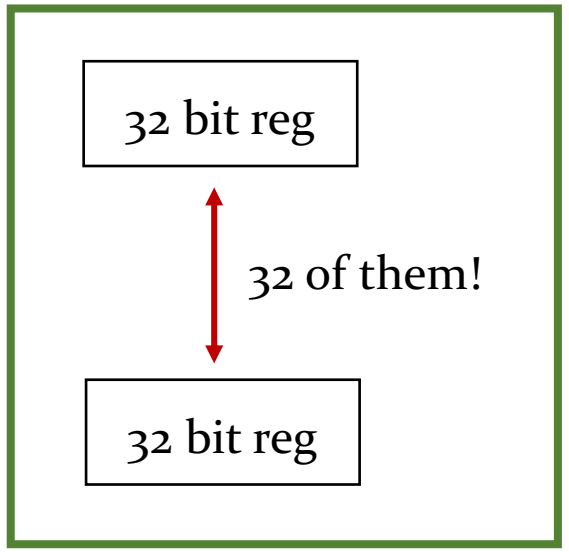
Recall D flip-flop
Stores state
1 flip-flop stores 1 bit

Program Counter
A 32-bit "register"
So what is a register?

Microarchitectural building blocks: register file

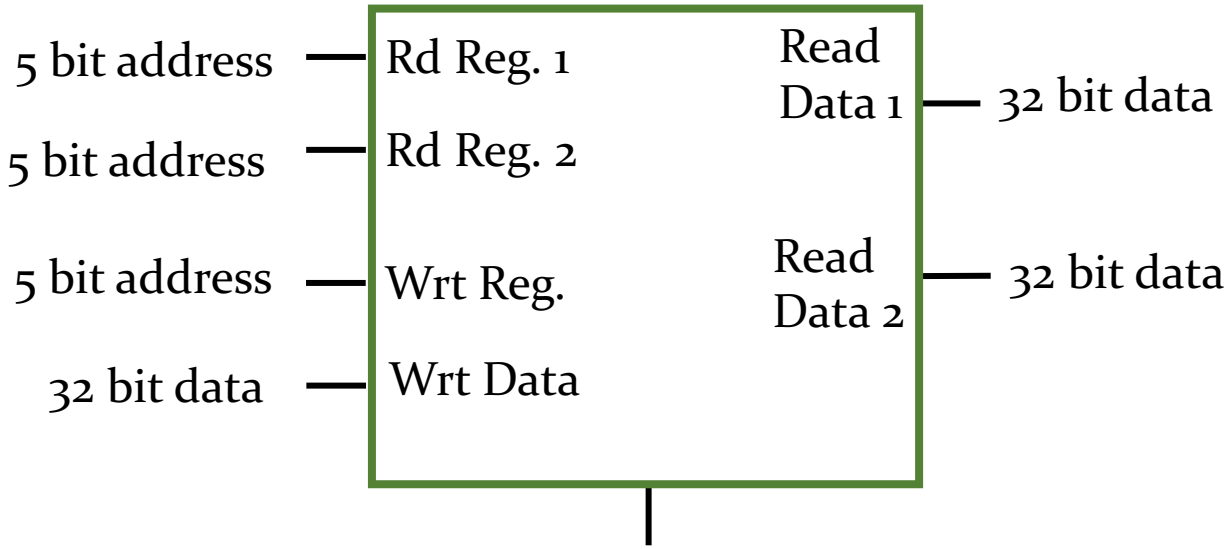
Need three things:
1) Data
2) Addresses
3) Control

RegFile internals



Register file is simply
a 1D array of registers

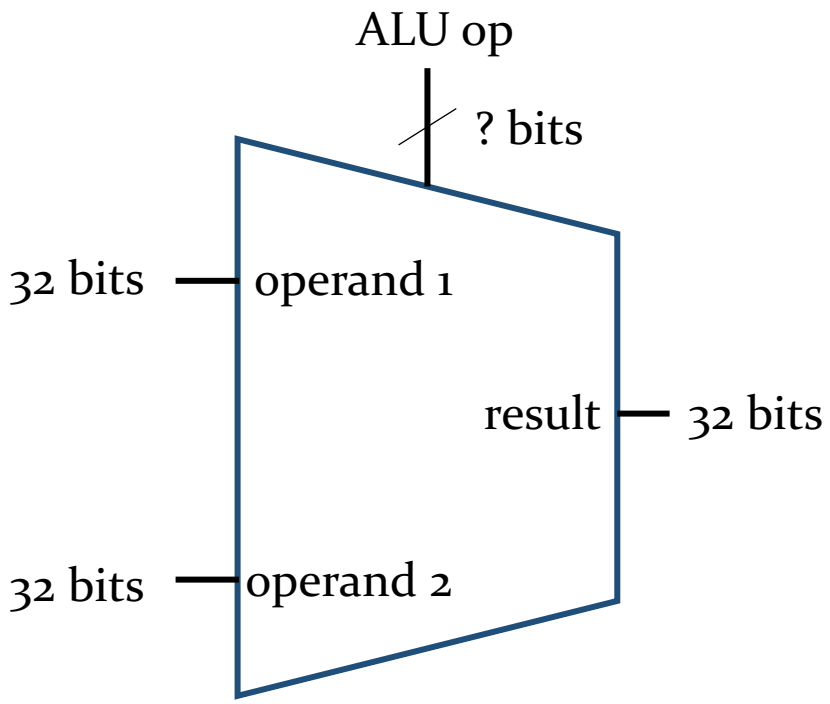
RegFile interface



Set to '1' to enable write

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

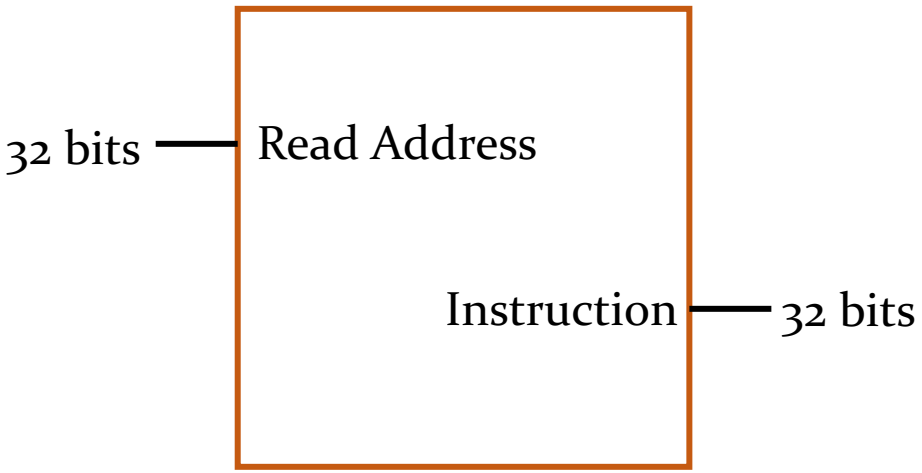
Microarchitectural building blocks: Arithmetic Logic Unit



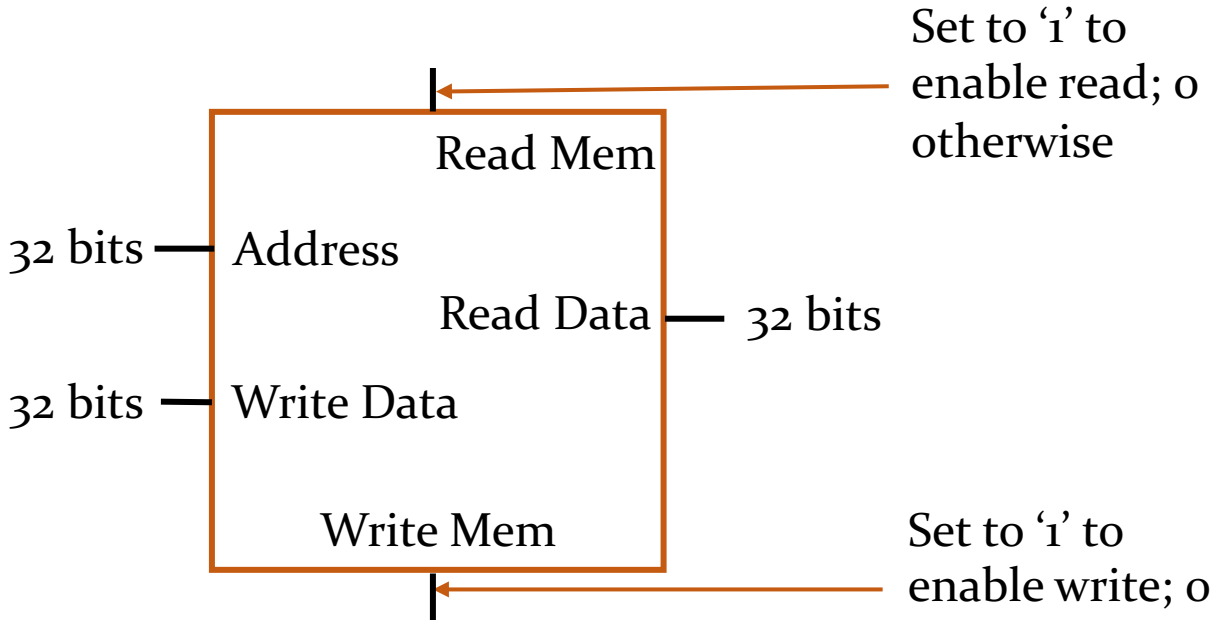
Arithmetic Logic Unit (ALU)

Implementation of the MIPS ISA: Memory

Memory is just a big 1D array of bytes
For this lecture, let's assume instructions and data are separate



Instruction Memory
(Read only)



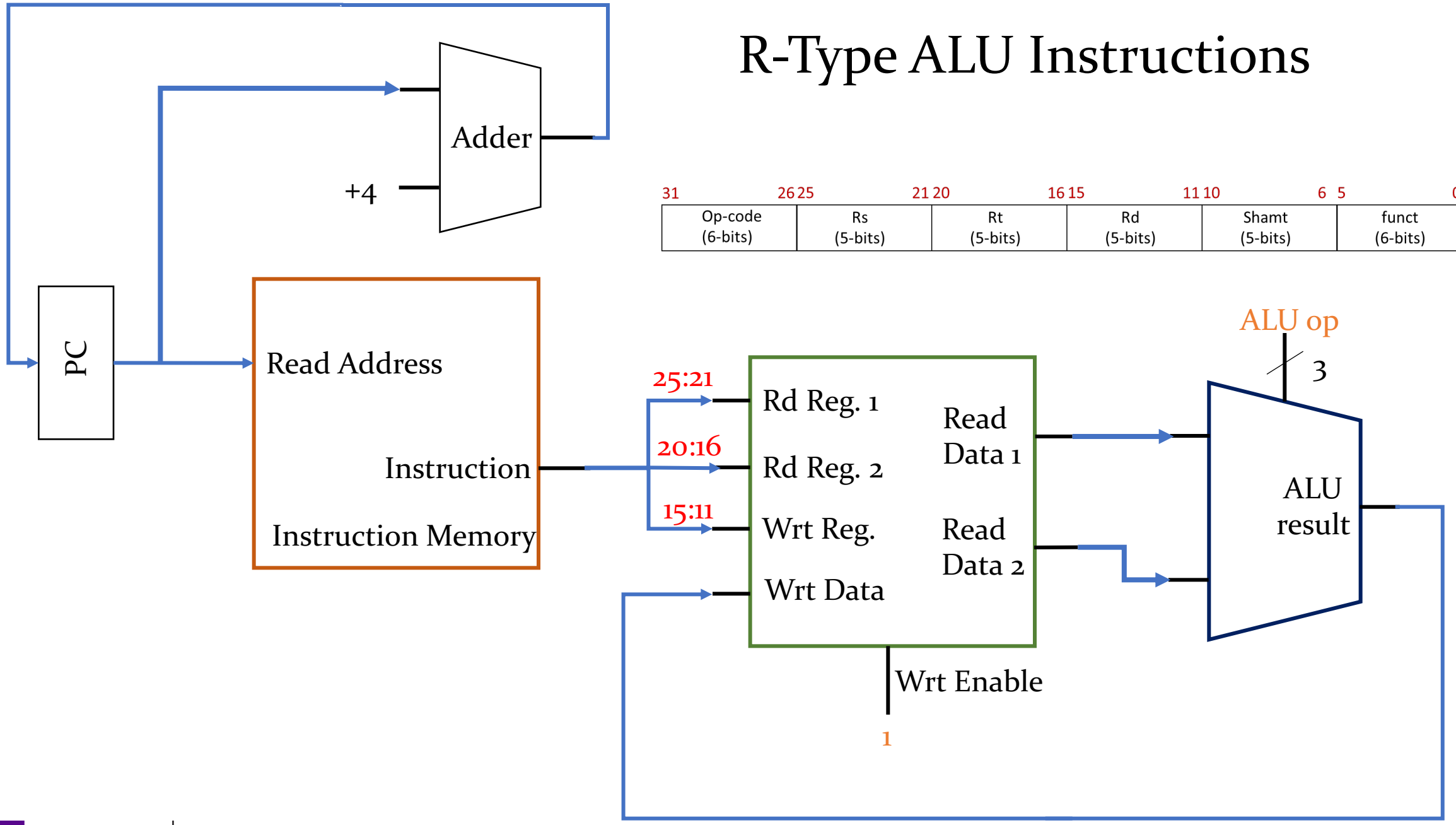
Data Memory
Read or write
but not simultaneously

Now we have all the pieces!

Rest of class: How do we connect and control them?

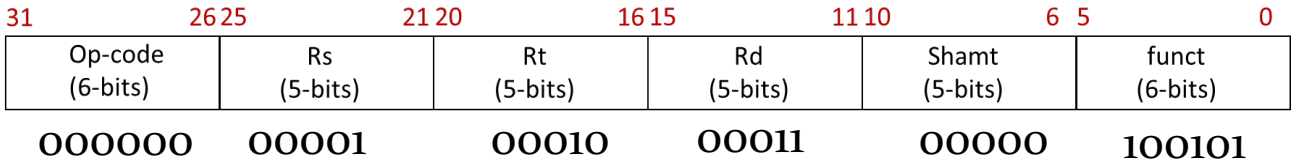
First:	R-type
Second:	I-type
Third:	Combined R/I datapaths
Forth:	Accessing memory
Finally:	J-Types

R-Type ALU Instructions



31 26	25 21	20 16	15 11	10 6	5 0
Op-code (6-bits)	Rs (5-bits)	Rt (5-bits)	Rd (5-bits)	Shamt (5-bits)	funct (6-bits)

Example: Instruction at PC=0

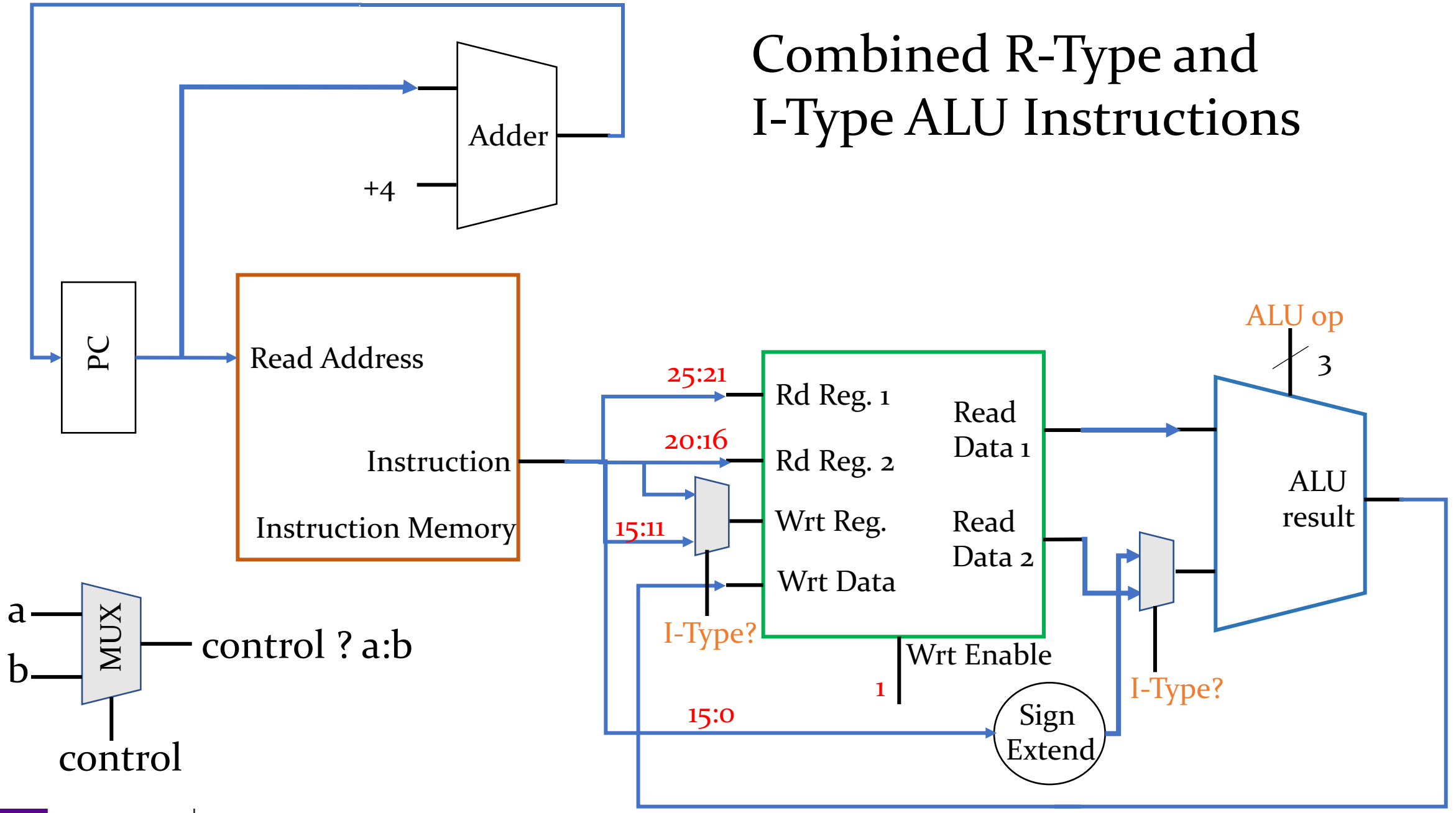


RF Contents

0000 0000	00000
A21F FFB3	00001
BB33 CD53	00010
1242 221E	00011
•	
•	
0001 0001	11111

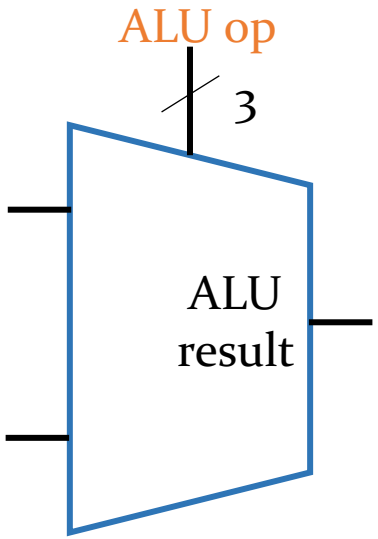
I-Type ALU Instructions

Combined R-Type and I-Type ALU Instructions



Combined R-Type and I-Type ALU Instructions

What about ALUop?



Add: 20_{hex} = 0010 0000
Addu: 21_{hex} = 0010 0001
And: 24_{hex} = 0010 0100
Or: 25_{hex} = 0010 0101
…

R-type Function Field

Addi: 8_{hex} = 1000
Addiu: 9_{hex} = 1001
Andi: c_{hex} = 1100
Ori: d_{hex} = 1101
…

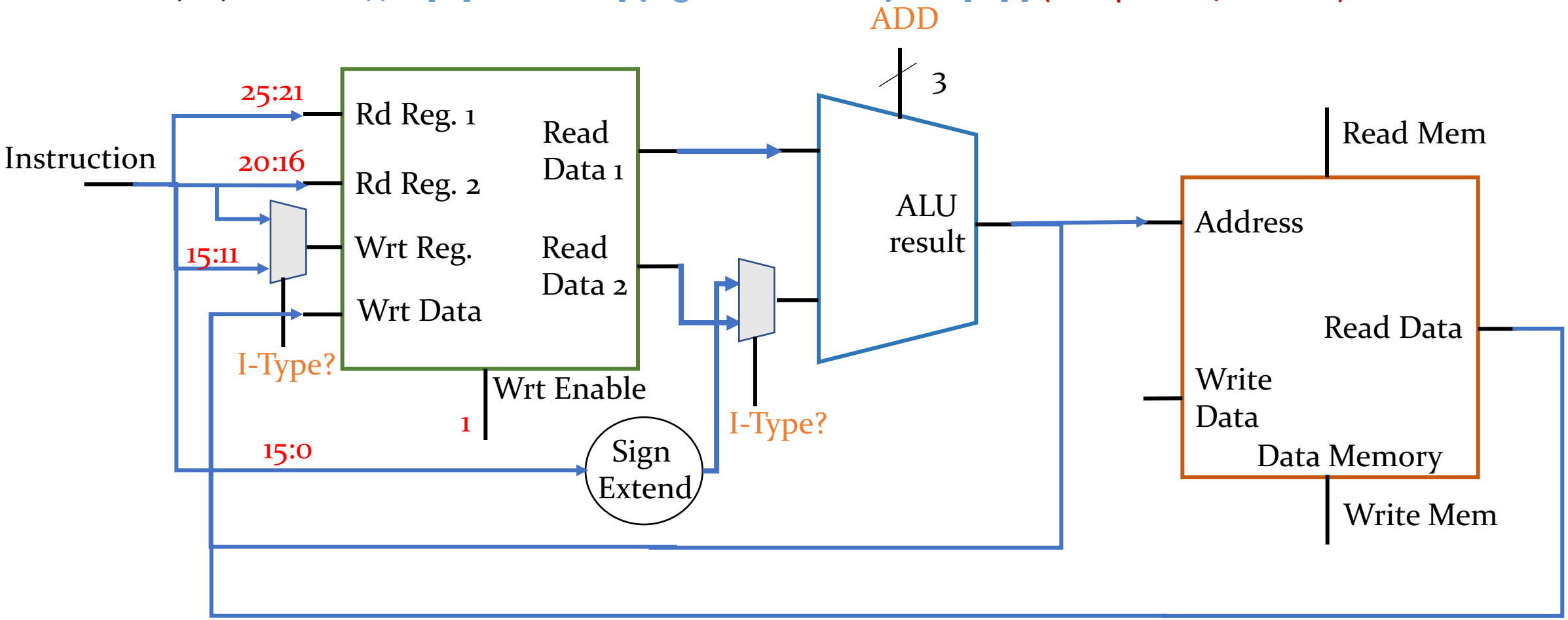
I-Type OpCode

Last 3 bits of function field or
Op Code uniquely identify ALU functionality!

I-Type Load Instructions

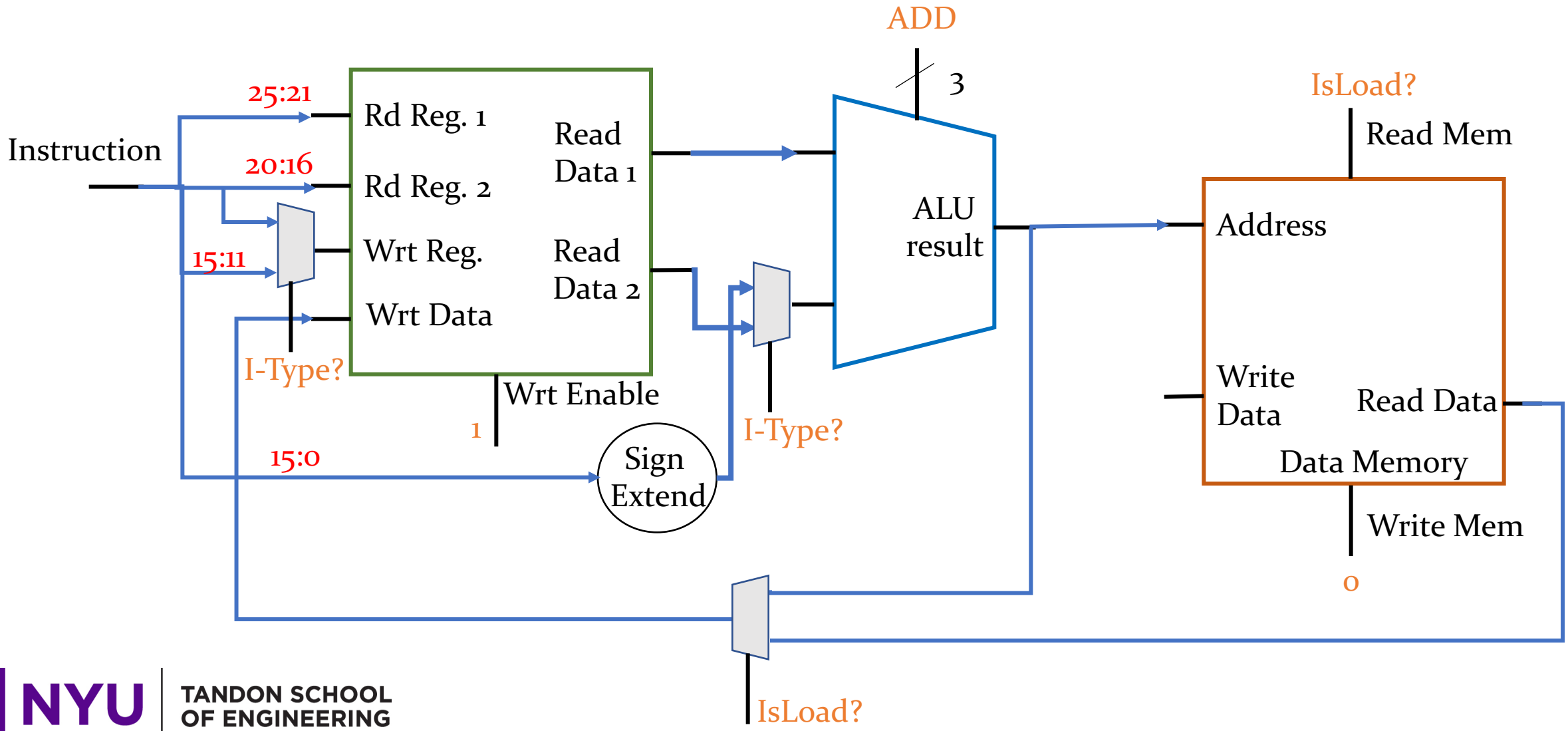
Instruction Fetch Part Not Shown

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



I-Type Load Instructions

Instruction Fetch Part Not Shown

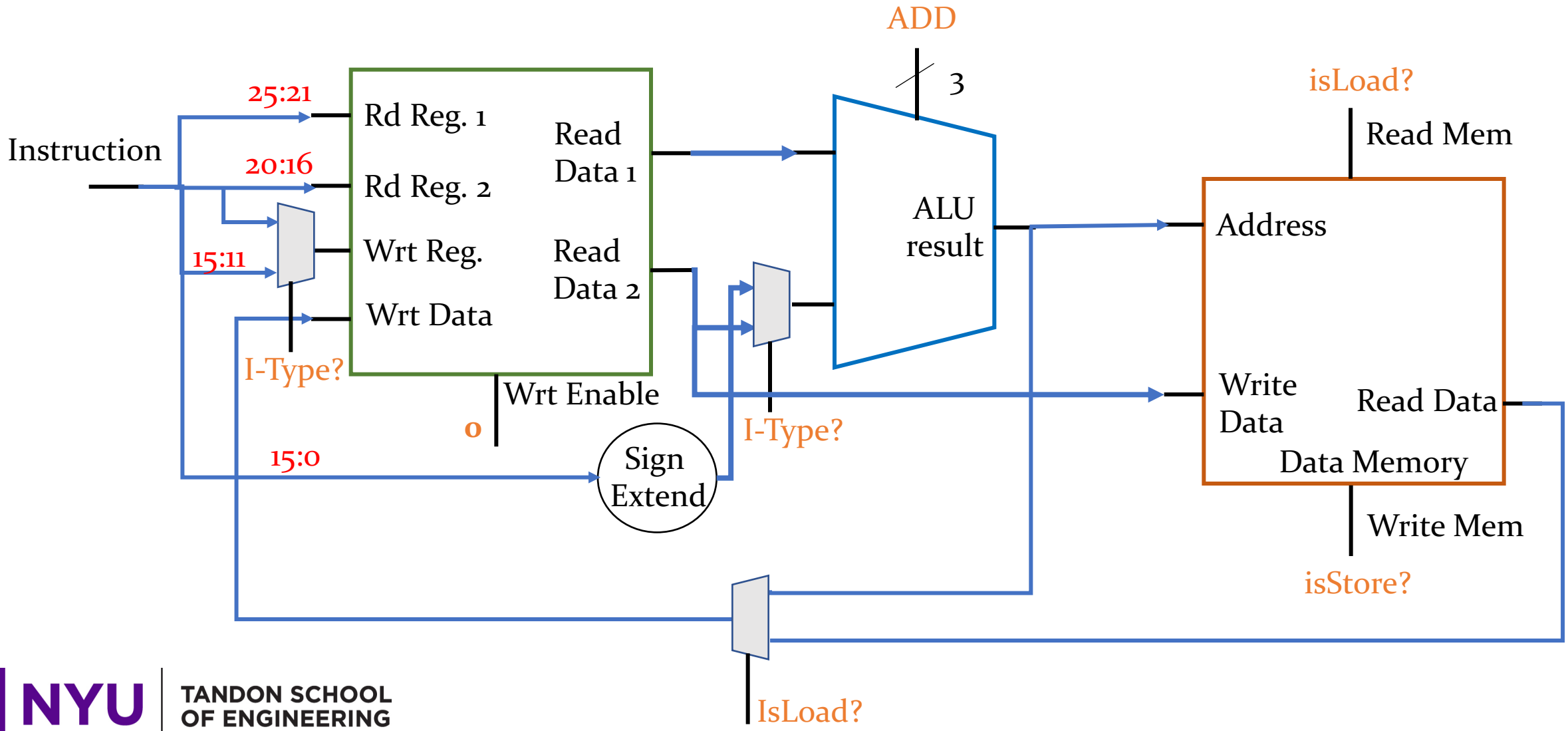


NYU TANDON SCHOOL OF ENGINEERING

I-Type Store Instructions

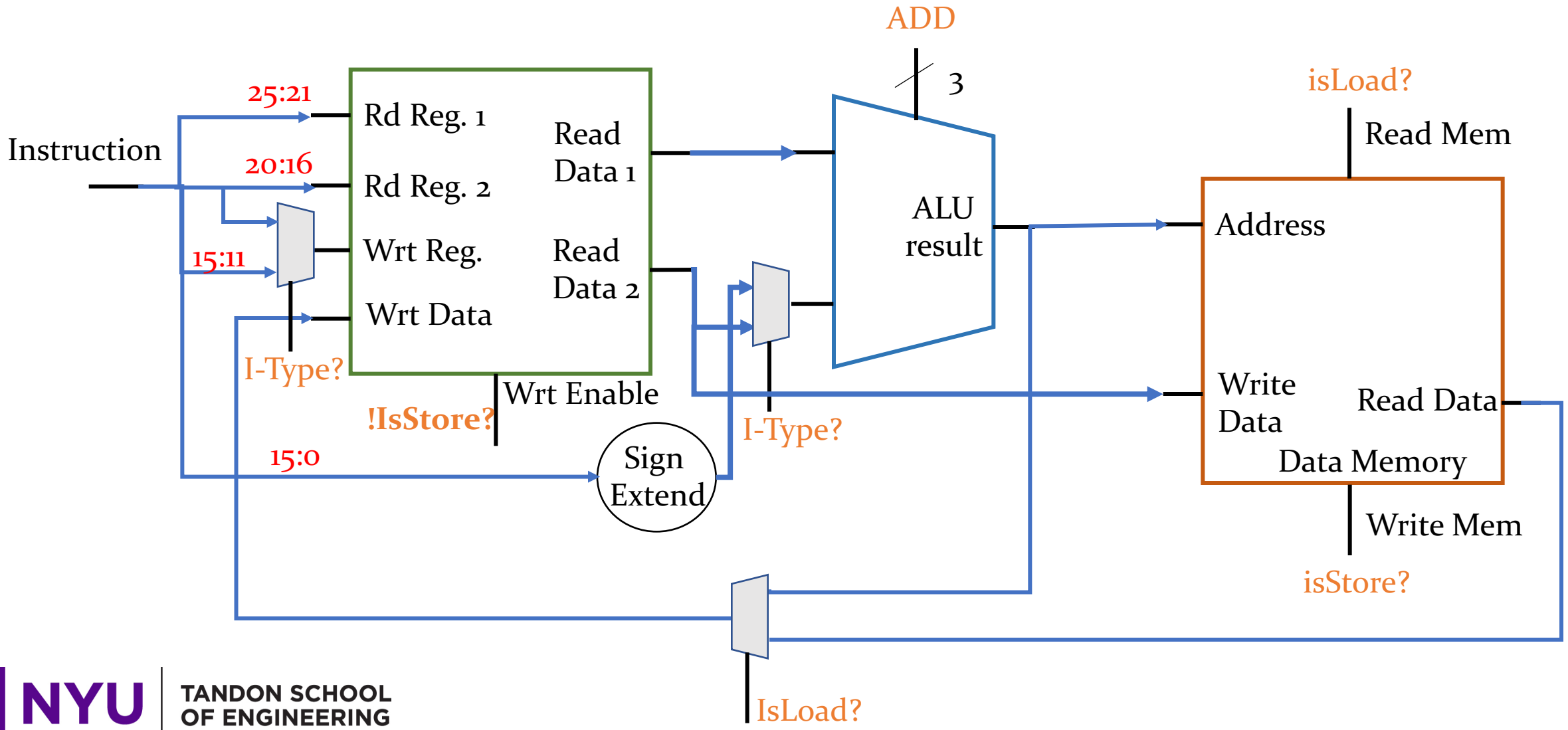
Did we miss something?

Instruction Fetch Part Not Shown

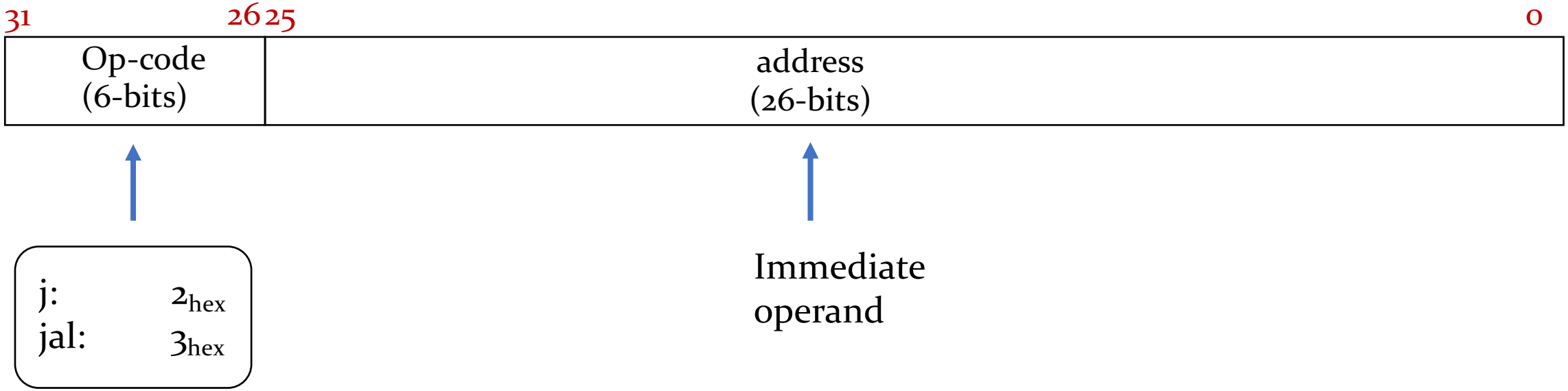


I-Type Store Instructions

Instruction Fetch Part Not Shown

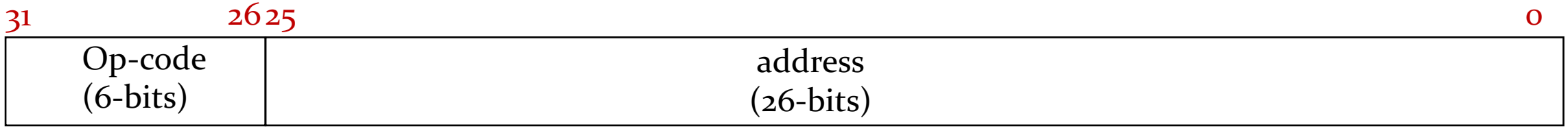


MIPS Instructions: J-Type



j address // PC ← {PC+4[31:28], address, 00}

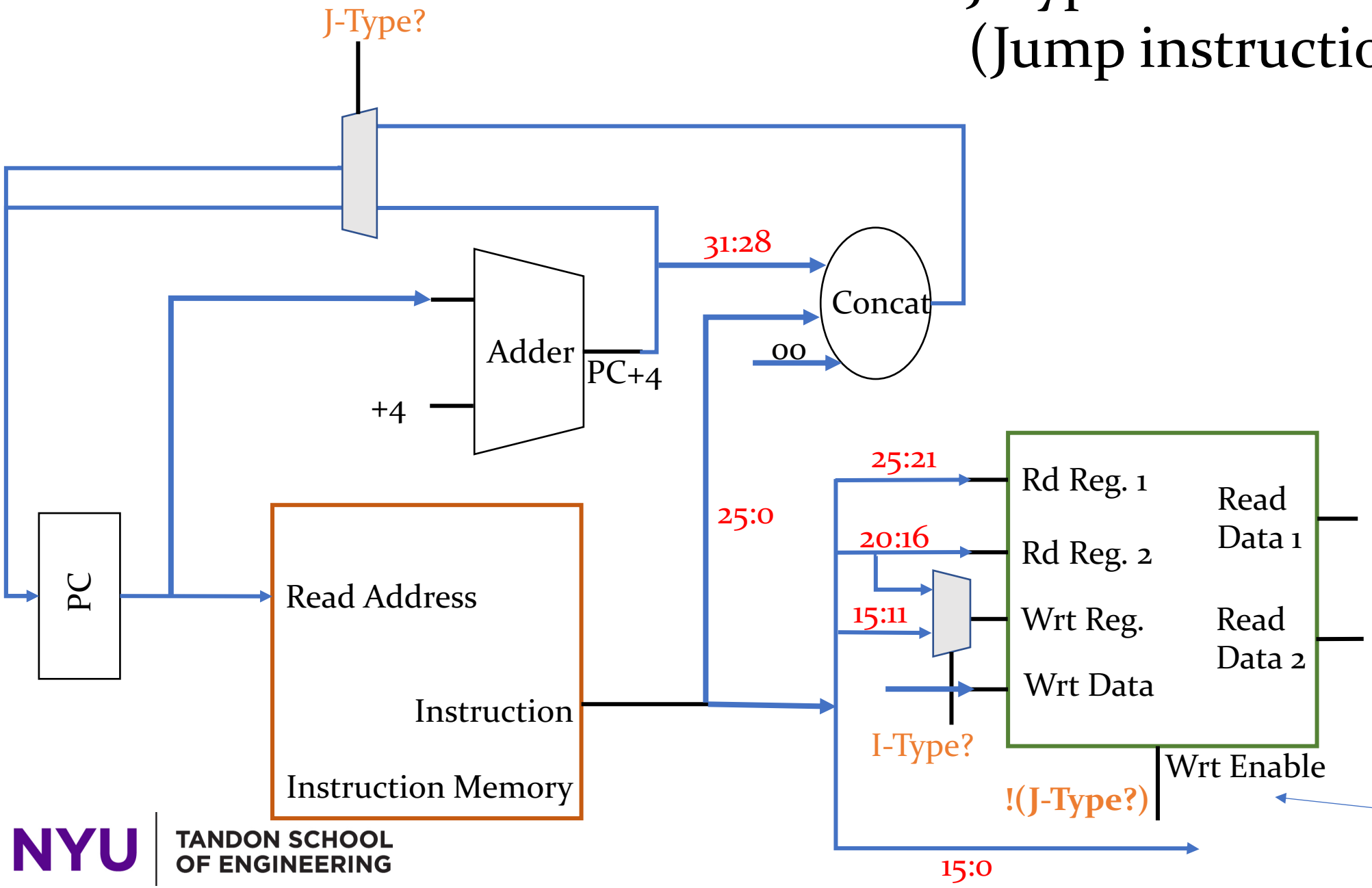
MIPS Instructions: J-Type



j address // PC ← {PC+4[31:28], address, 00}, opcode=2_{hex}

- 4 MSB bits taken from the 4 MSBs of PC+4
- Next 26 bits taken from the "address" field
- Last 2 bits are set to zero? (why?)

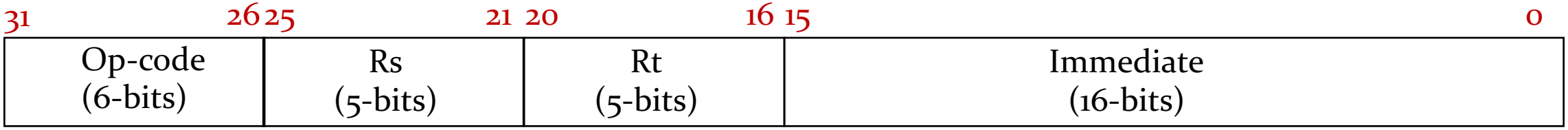
J-Type Control Instruction (Jump instruction)



RF Onwards Not Shown

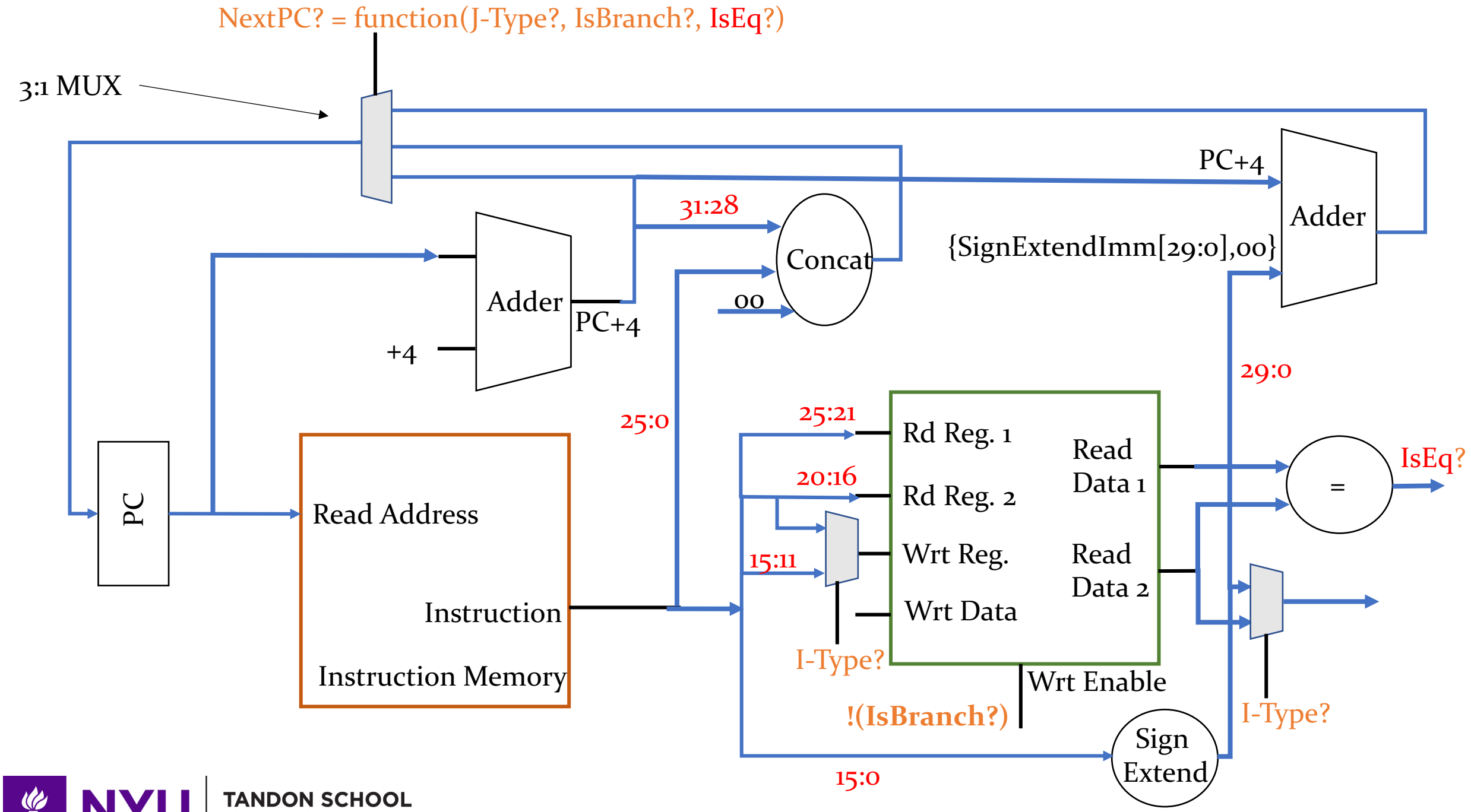
Control Flow

I-Type Branch Instruction

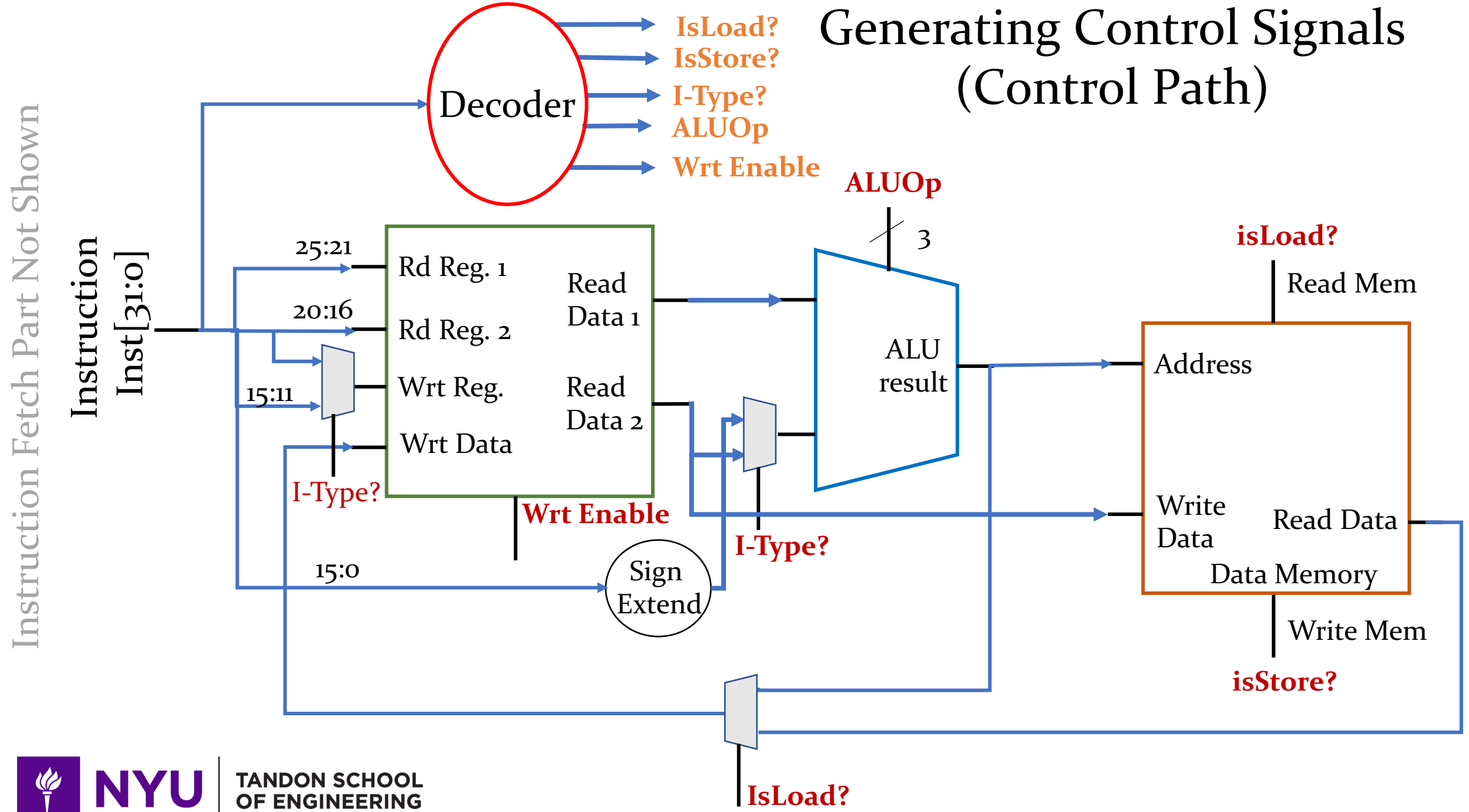


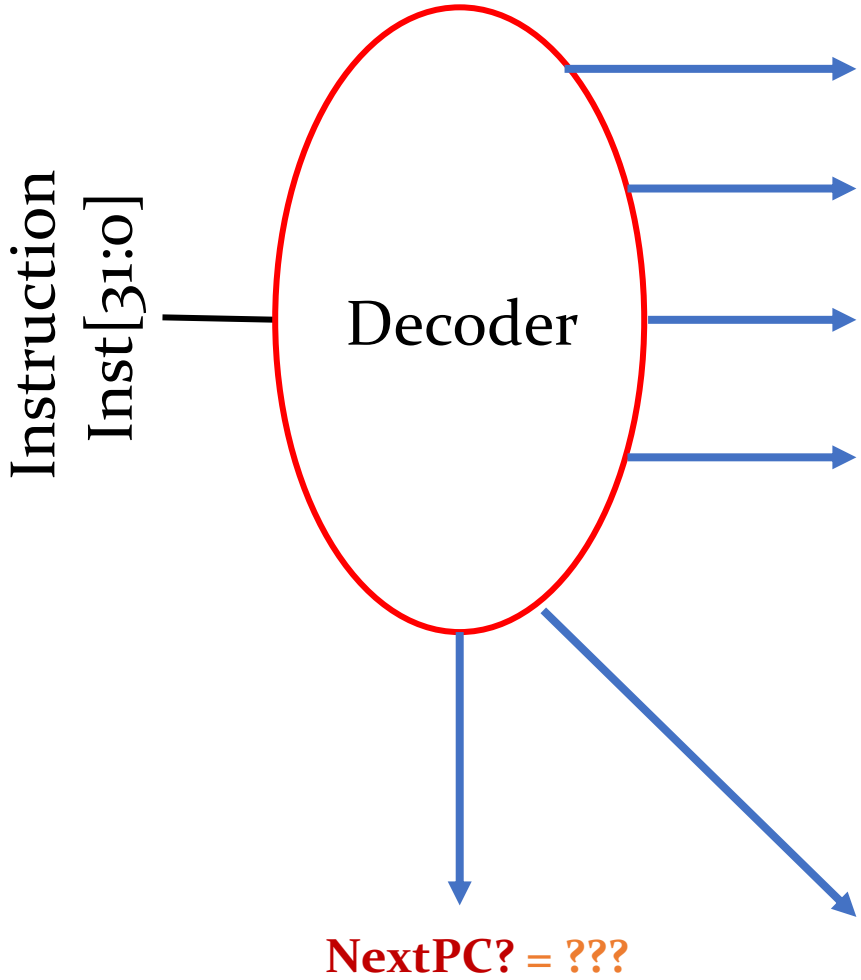

```
beq rs, rt, imm // if (RF[rs]==RF[rt])
                //      PC ← PC + 4 + {SignExtendImm, 00},
                //  else
                //      PC ← PC + 4
                // opcode=4hex
```

- First 30 MSBs from sign extended immediate
- 2 LSBs are always 0


NextPC? = function(J-Type?, IsBranch?, IsEq?)
3:1 MUX
PC+4
Adder
31:28
Concat
{SignExtendImm[29:0],00}
Adder
+4
PC+4
00
29:0
25:0
25:21
20:16
15:11
Rd Reg. 1
Rd Reg. 2
Wrt Reg.
Wrt Data
Read Data 1
Read Data 2
=
IsEq?
PC
Read Address
Instruction
Instruction Memory
I-Type?
Wrt Enable
!(IsBranch?)
I-Type?
Sign Extend
15:0

Generating Control Signals (Control Path)





IsLoad? = (Inst[31:26] == 100011) // Just check OpCode!

IsStore? = (Inst[31:26] == 101011) // Just check OpCode!

I-Type? = (Inst[31:26] != 00000) && (Inst[31:26] != 0001x)
//(Is not R-Type) AND (Is not J-Type)

ALUOp =

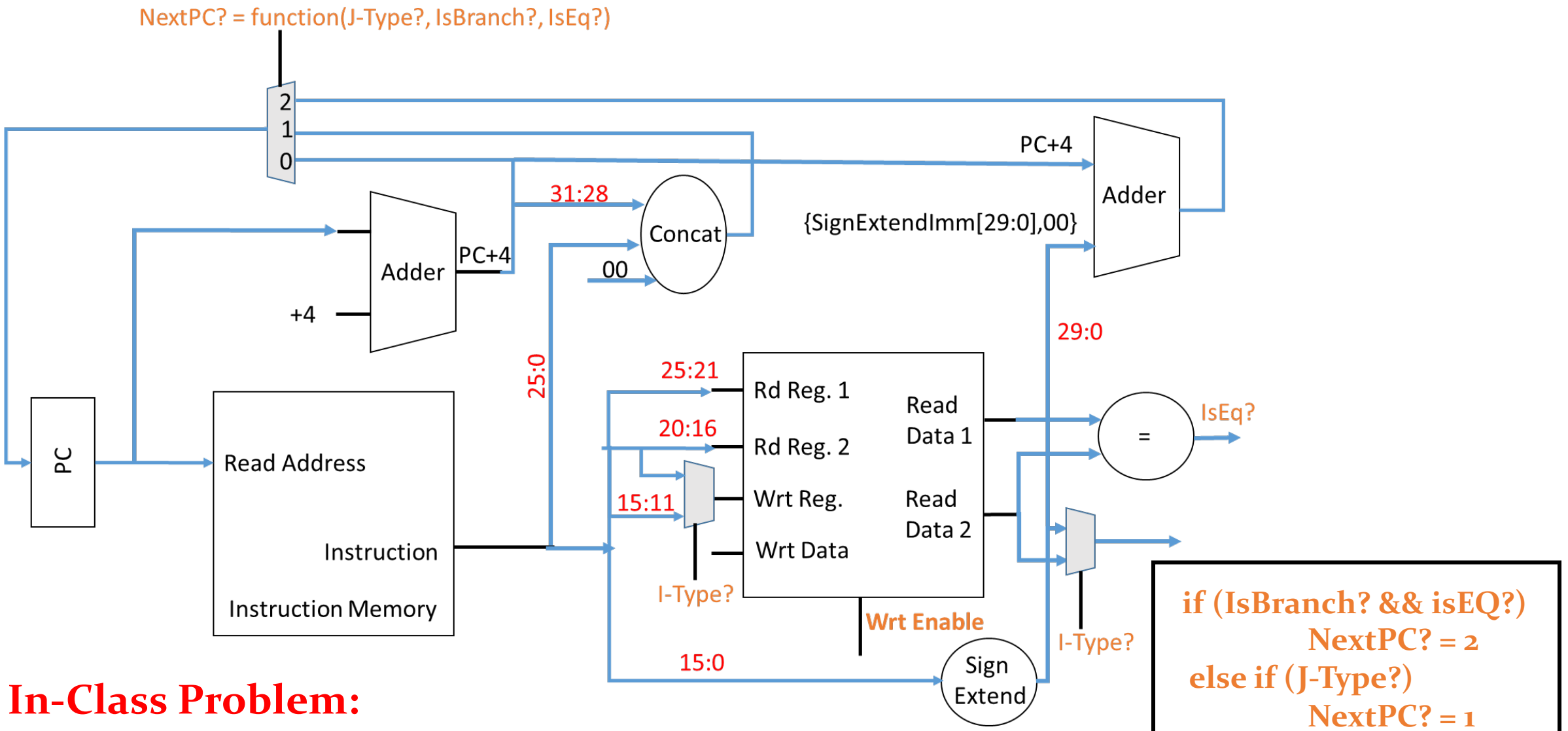
```
if (Inst[31:26] == 100011) | (Inst[31:26] == 101011)     // is it ld or st?
        ALUop = 001                                      // Add! Why?
else If (Inst[31:26]==000000)                            // Is it R-Type?
        ALUop = Inst[2:0]                                // 3 LSBs of func field
else                                                     // everything else
        ALUop = Inst[28:26]                              // 3 LSBs of OpCode!
```

Wrt Enable =

```
if (IsStore? | IsBranch? | J-Type?)
        WrtEnable = 0
else
        WrtEnable = 1
```

J-Type? = (Inst[31:26]==000010)

IsBranch? = (Inst[31:26]==000100)



In-Class Problem:
Write pseudo-code for NextPC?

```
if (IsBranch? && isEQ?)
        NextPC? = 2
else if (J-Type?)
        NextPC? = 1
else
        NextPC? = 0
```

Now we have all the pieces!

Rest of class: How do we connect and control them?

First:	R-type
Second:	I-type
Third:	Combined R/I datapaths
Forth:	Accessing memory
Finally:	J-Types