Lecture #11

The Processor: Datapath

NUS | School of
National University | Computing
of Singapore

Lecture #11: Processor: Datapath (1/2)

1. Building a Processor: Datapath & Control

2. MIPS Processor: Implementation

3. Instruction Execution Cycle (Recap)

4. MIPS Instruction Execution

5. Let's Build a MIPS Processor
    5.1   Fetch Stage
    5.2   Decode Stage
    5.3   ALU Stage
    5.4   Memory Stage
    5.5   Register Write Stage

Lecture #11: Processor: Datapath (2/2)

6. The Complete Datapath!

7. Brief Recap

8. From C to Execution

   8.1 Writing C program

   8.2 Compiling to MIPS

   8.3 Assembling to Binaries

   8.4 Execution (Datapath)

1. Building a Processor: Datapath & Control

- Two major components for a processor

Datapath

- Collection of components that process data
- Performs the arithmetic, logical and memory operations

Control

- Tells the datapath, memory and I/O devices what to do according to program instructions

2. MIPS Processor: Implementation

- Simplest possible implementation of a subset of the core MIPS ISA:
  - **Arithmetic and Logical operations**
    - `add`, `sub`, `and`, `or`, `addi`, `andi`, `ori`, `slt`
  - **Data transfer instructions**
    - `lw, sw`
  - **Branches**
    - `beq, bne`
- Shift instructions (`sll`, `srl`) and J-type instructions (`j`) will not be discussed:
  - Left as exercises ☺

3. Instruction Execution Cycle (Basic)

Next Instruction

Instruction Fetch	
Instruction Decode	
Operand Fetch	
Execute	
Result Write	
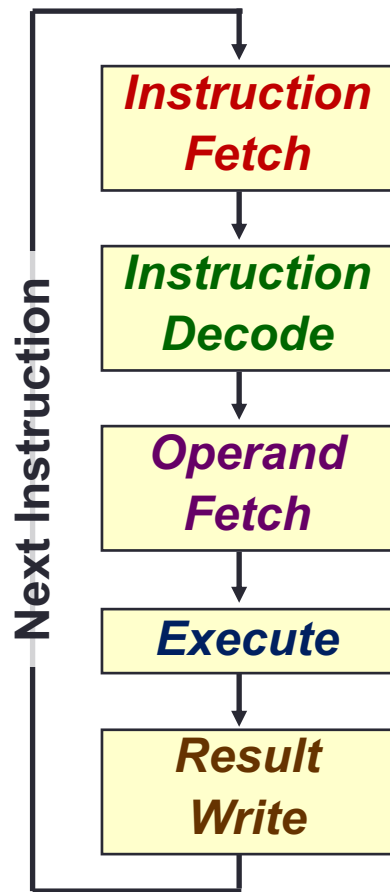
1. **Fetch:**
   - Get instruction from memory
   - Address is in **P**rogram **C**ounter (PC) Register

2. **Decode:**
   - Find out the operation required

3. **Operand Fetch:**
   - Get operand(s) needed for operation

4. **Execute:**
   - Perform the required operation

5. **Result Write (Store):**
   - Store the result of the operation

4. MIPS Instruction Execution (1/2)

- Show the actual steps for 3 representative MIPS instructions
- Fetch and Decode stages not shown:
  - The standard steps are performed

	`add $3, $1, $2`	`lw $3, 20($1)`	`beq $1, $2, ofst`
Fetch	*standard*	*standard*	*standard*
Decode			
Operand Fetch	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*	○ Read [**$1**] as *opr1* ○ Use **20** as *opr2*	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*
Execute	*Result = opr1 + opr2*	○ *MemAddr = opr1 + opr2* ○ Use *MemAddr* to read from memory	*Taken = (opr1 == opr2)?* *Target = (**PC**+4) + **ofst**×4*
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	if (*Taken*) **PC** = *Target*

- **opr** = operand
- **MemAddr** = Memory Address
- **ofst** = offset

4. MIPS Instruction Execution (2/2)

- Design changes:
  - Merge *Decode* and *Operand Fetch* – Decode is simple for MIPS
  - Split *Execute* into **ALU** (Calculation) and **Memory Access**

	`add $3, $1, $2`	`lw $3, 20($1)`	`beq $1, $2, ofst`
Fetch	Read inst. at [PC]	Read inst. at [PC]	Read inst. at [PC]
Decode & Operand Fetch	o Read **[$1]** as *opr1* o Read **[$2]** as *opr2*	o Read **[$1]** as *opr1* o Use **20** as *opr2*	o Read **[$1]** as *opr1* o Read **[$2]** as *opr2*
ALU	*Result = opr1 + opr2*	*MemAddr = opr1 + opr2*	*Taken = (opr1 == opr2)?* *Target = (**PC**+4) + **ofst**×4*
Memory Access		Use *MemAddr* to read from memory	
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	if (*Taken*) **PC** = *Target*

5. Let's Build a MIPS Processor

- What we are going to do:
  - Look at each stage closely, figure out the requirements and processes
  - Sketch a high level block diagram, then zoom in for each elements
  - With the simple starting design, check whether different type of instructions can be handled:
    - Add modifications when needed

- ➔ Study the design from the viewpoint of a designer, instead of a "tourist" ☺

5.1 **Fetch Stage**: Requirements

- Instruction **Fetch Stage**:

   1. Use the **P**rogram **C**ounter (**PC**) to fetch the instruction from **memory**

      - PC is implemented as a special register in the processor

   2. **Increment** the PC by 4 to get the address of the next instruction:

      - How do we know the next instruction is at PC+4?

      - Note the exception when branch/jump instruction is executed

- Output to the next stage (**Decode**):

   - The instruction to be executed

5.1 **Fetch Stage**: Block Diagram
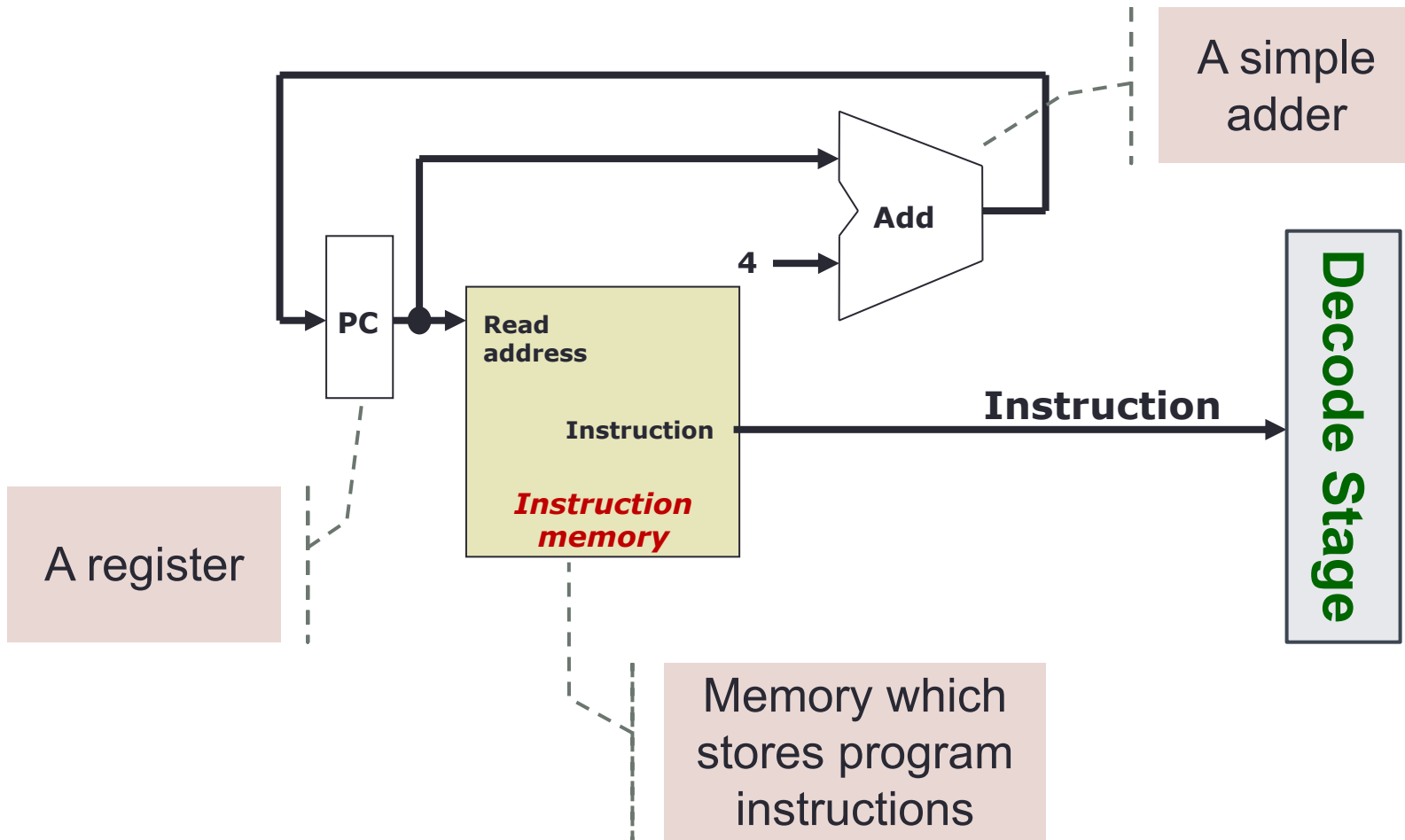
5.1 Element: **Instruction Memory**

- Storage element for the instructions
  - It is a **sequential circuit** (to be covered later)
  - Has an internal state that stores information
  - Clock signal is assumed and not shown

Instruction Address

Instruction

Instruction Memory
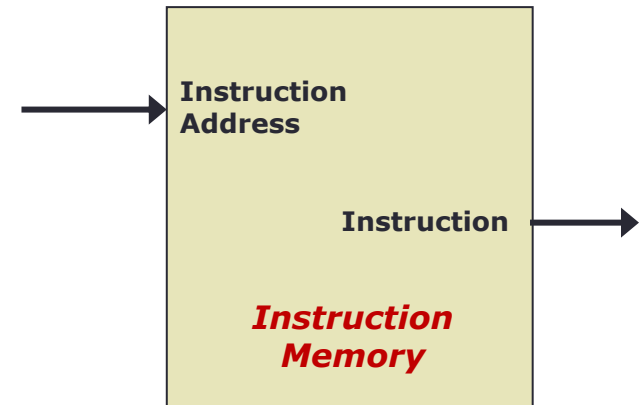
- Supply instruction given the address
  - Given instruction address M as input, the memory outputs the content at address M
  - Conceptual diagram of the memory layout is given on the right ➔

Memory

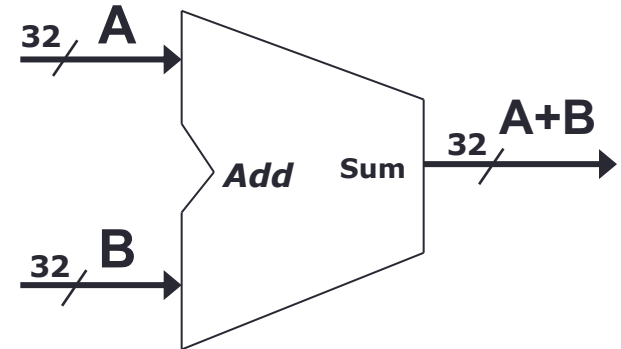|	
	....
2048	add $3, $1, $2
2052	sll $4, $3, 2
2056	andi $1, $4, 0xF
......	....

5.1 Element: **Adder**

- Combinational logic to implement the addition of two numbers

- **Inputs:**
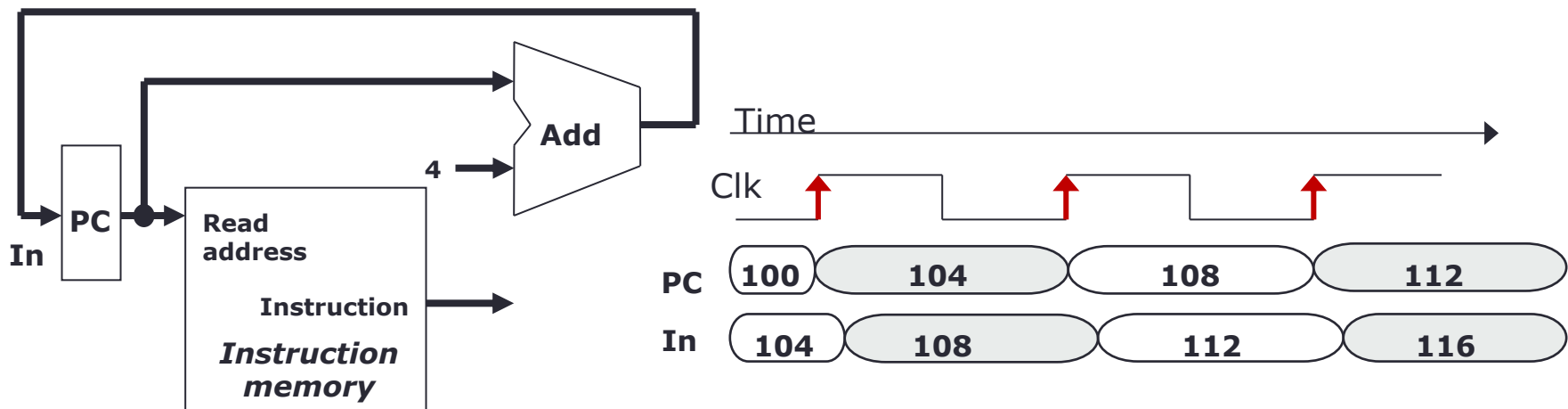 - Two 32-bit numbers **A**, **B**

- **Output:**
  - Sum of the input numbers, **A + B**

5.1 The Idea of Clocking

- It seems that we are reading and updating the PC at the same time:
  - How can it works properly?
- **Magic of clock**:
  - PC is read during the first half of the clock period and it is updated with PC+4 at the **next rising clock edge**

5.2 **Decode Stage**: Requirements

1.	Fetch
2.	**Decode**
3.	ALU
4.	Memory
5.	RegWrite

- Instruction **Decode Stage**:
  - Gather data from the instruction fields:
    1. Read the **opcode** to determine instruction type and field lengths
    2. Read data from all necessary registers
       - Can be two (e.g. `add`), one (e.g. `addi`) or zero (e.g. `j`)

- Input from previous stage (**Fetch**):
  - Instruction to be executed

- Output to the next stage (**ALU**):
  - Operation and the necessary operands

5.2 **Decode Stage**: Block Diagram
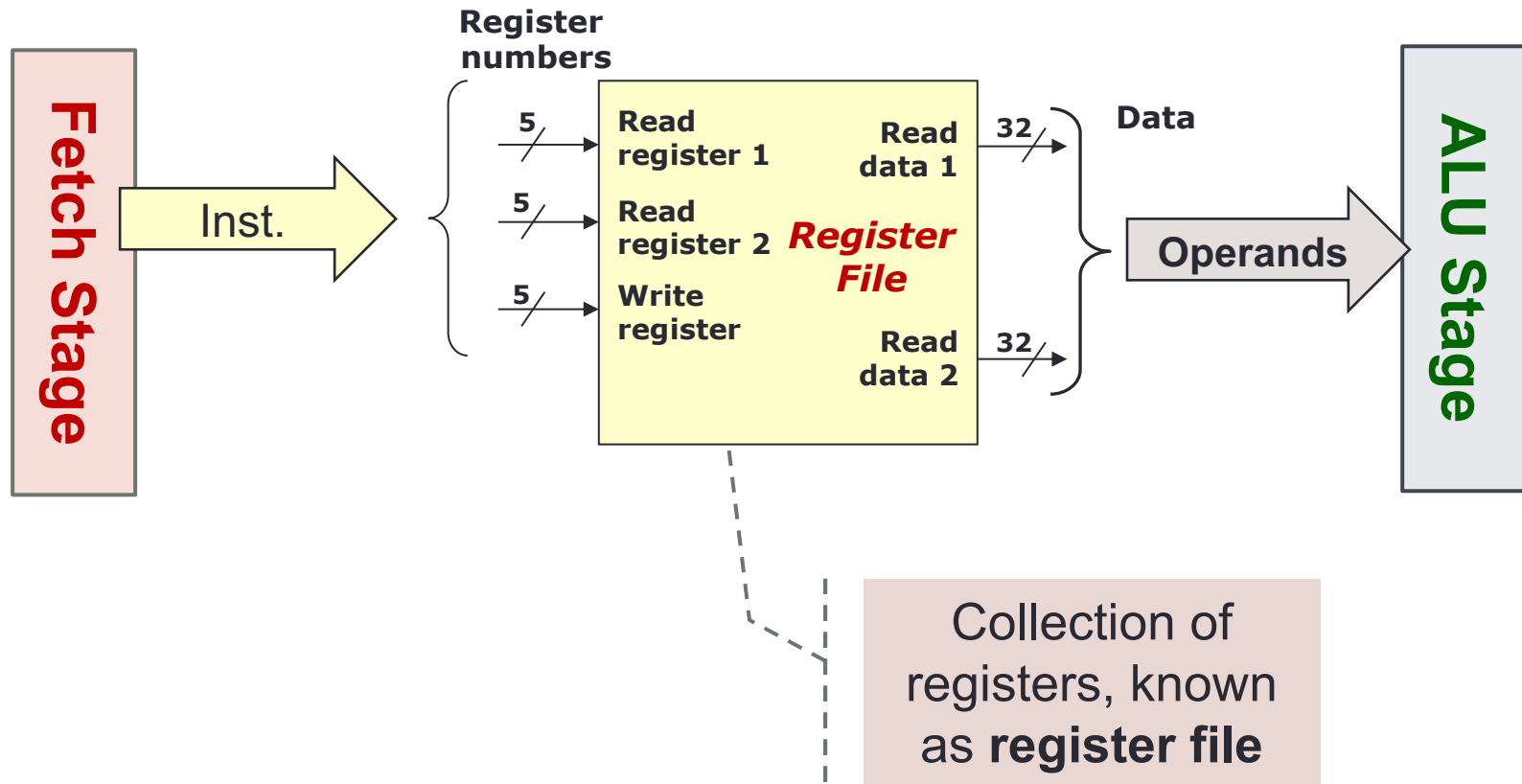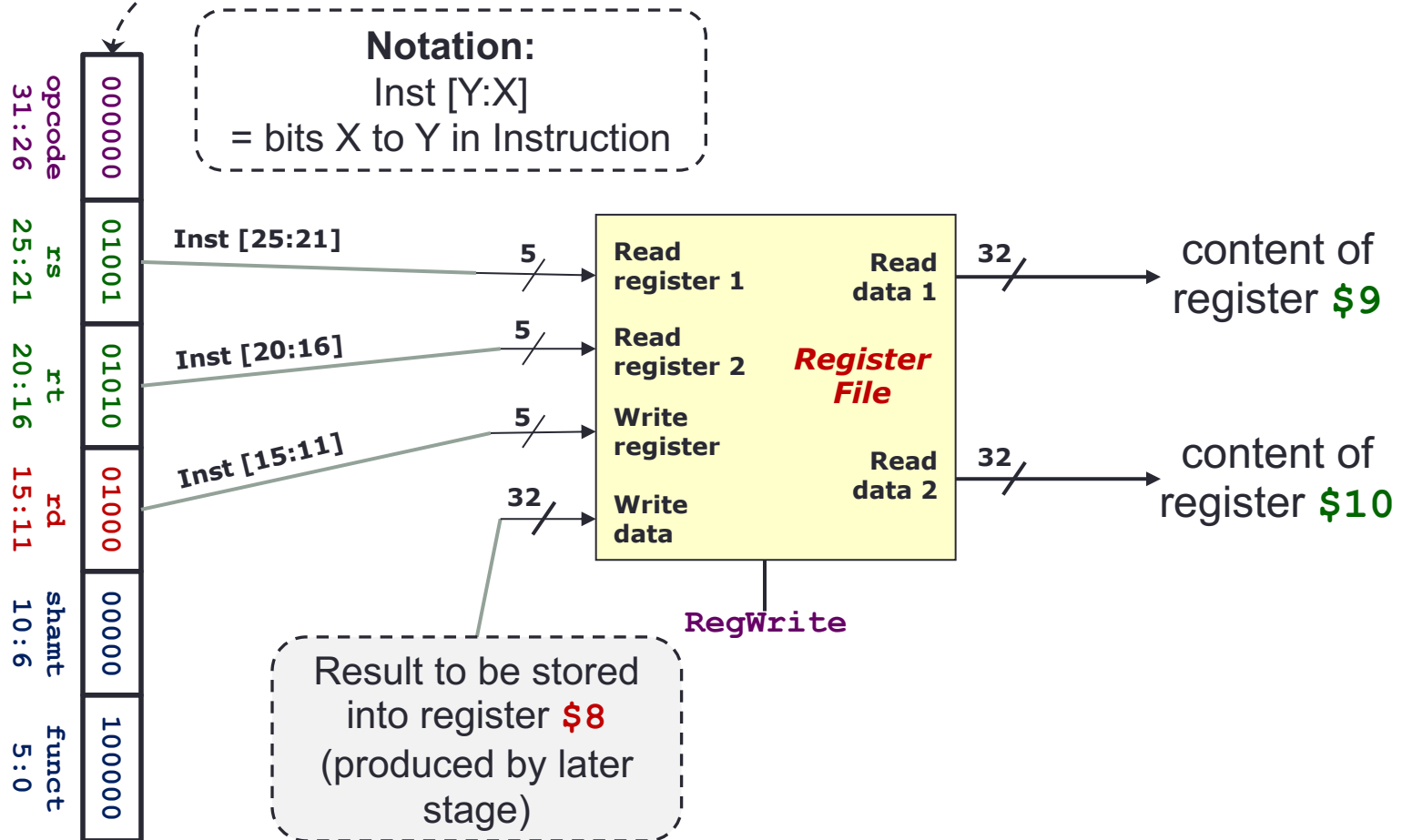
5.2 Element: **Register File**

- A collection of 32 registers:
  - Each 32-bit wide; can be read/written by specifying register number
  - Read at most two registers per instruction
  - Write at most one register per instruction

- **RegWrite** is a control signal to indicate:
  - Writing of register
  - 1(True) = Write, 0 (False) = No Write

5.2 Decode Stage: **R-Format Instruction**

add $8, $9, $10

Notation:
Inst [Y:X]
= bits X to Y in Instruction

opcode 31:26	000000
rs 25:21	01001
rt 20:16	01010
rd 15:11	01000
shamt 10:6	00000
funct 5:0	100000

Inst [25:21] → 5 → Read register 1

Inst [20:16] → 5 → Read register 2

Inst [15:11] → 5 → Write register

32 → Write data

Register File

Read data 1 → 32 → content of register **$9**

Read data 2 → 32 → content of register **$10**

RegWrite

Result to be stored into register **$8** (produced by later stage)

5.2 Decode Stage: **I-Format Instruction**

addi $21, $22, -50

opcode
31:26
`001000`

rs
25:21
`10110`

rt
20:16
`10101`

Immediate
15:0
`1111 1111 1100 1110`

Inst [25:21]

5

Inst [20:16]

5

Inst [15:11]

5

Read register 1

Read register 2

Write register

Write data

Register File

Read data 1   32 — content of register $22

Read data 2   32

RegWrite
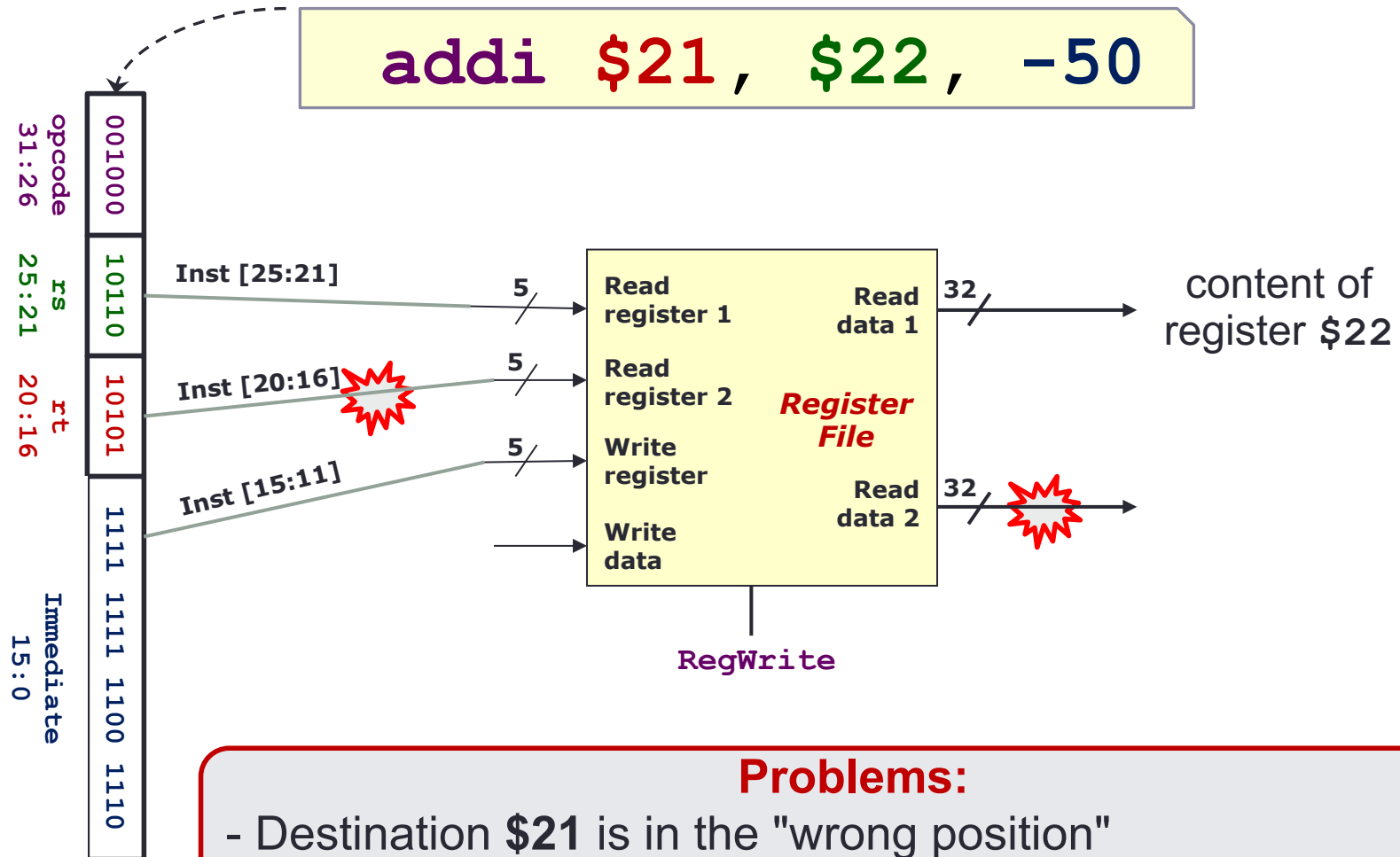
Problems:
- Destination **$21** is in the "wrong position"
- **Read Data 2** is an immediate value, not from register

5.2 Decode Stage: **Choice in Destination**

addi $21, $22, -50

opcode
31:26
`001000`

rs
25:21
`10110`

rt
20:16
`10101`

Immediate
15:0
`1111 1111 1100 1110`

Inst [25:21]

Inst [20:16]

Inst [15:11]

5

5

5

M U X

RegDst

Read register 1
Read register 2
Write register
Write data

Read data 1
Read data 2

Register File

32

32

content of register **$22**

RegWrite

RegDst:
A control signal to choose either Inst[20:16] or Inst[15:11] as the write register number
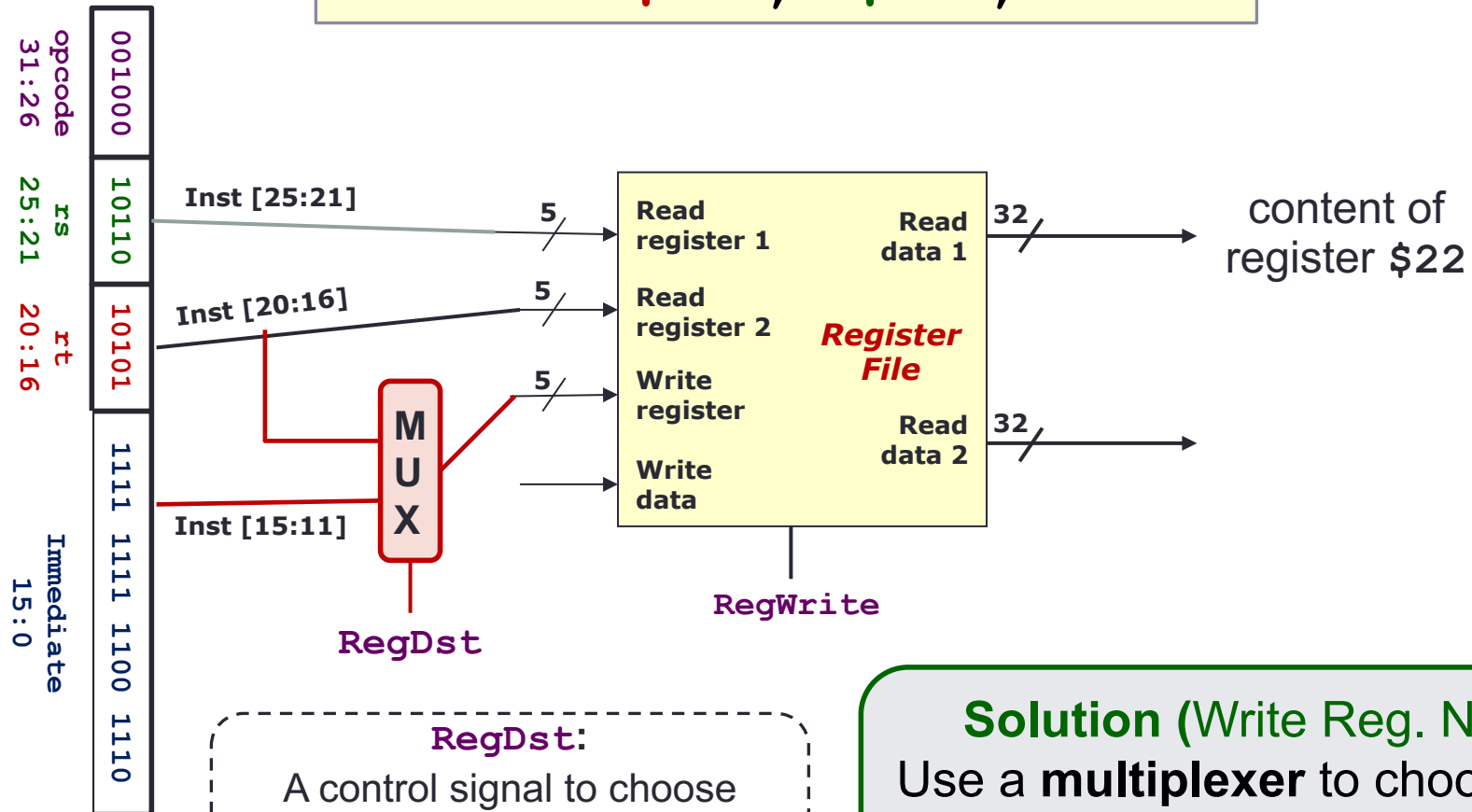
Solution (Write Reg. No.**):**
Use a **multiplexer** to choose the correct write register number based on instruction type

5.2 Multiplexer

- **Function:**
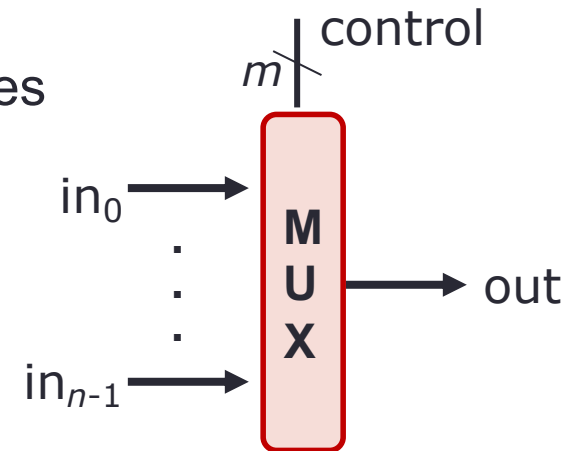 - Selects one input from multiple input lines

- **Inputs:**
  - n lines of same width

- **Control:**
  - m bits where $n = 2^m$
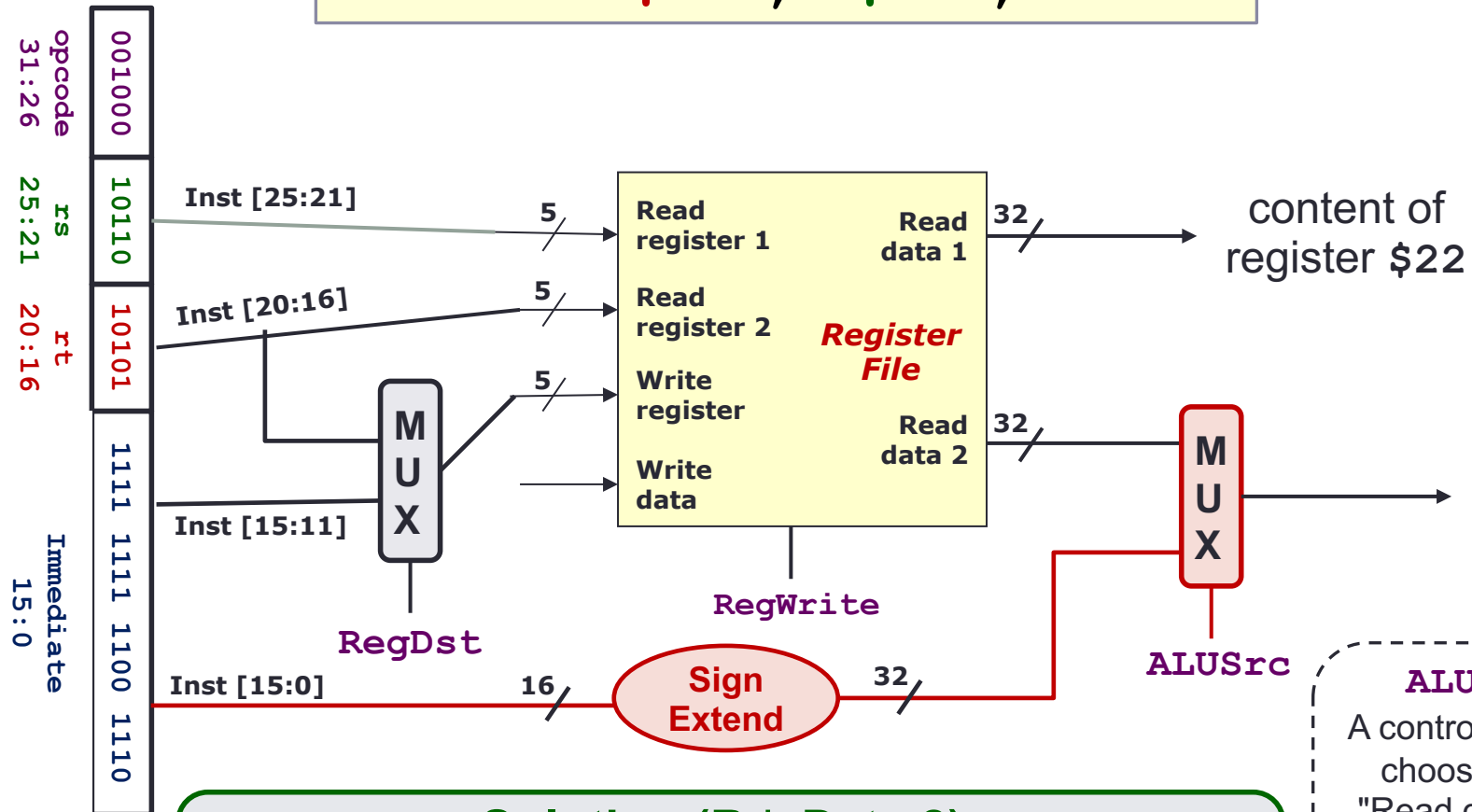
- **Output:**
  - Select i^{th} input line if control = i

control

m

in_0

.
.
.

in_{n-1}

M U X

out

Control=0 $\rightarrow$ select in_0 to out
Control=3 $\rightarrow$ select in_3 to out

5.2 Decode Stage: **Choice in Data 2**

addi $21, $22, -50

opcode 31:26 : 001000

rs 25:21 : 10110

rt 20:16 : 10101

Immediate 15:0 : 1111 1111 1100 1110

Inst [25:21] → 5 → Read register 1

Inst [20:16] → 5 → Read register 2

→ 5 → Write register

Inst [15:11]

M U X — RegDst

Read data 1 → 32 → content of register $22

Register File

Read data 2 → 32 →

Write data

RegWrite

Inst [15:0] → 16 → **Sign Extend** → 32 →

M U X — ALUSrc

ALUSrc: A control signal to choose either "Read data 2" or the sign extended Inst[15:0] as the second operand

Solution (Rd. Data 2**):**
Use a **multiplexer** to choose the correct operand 2.
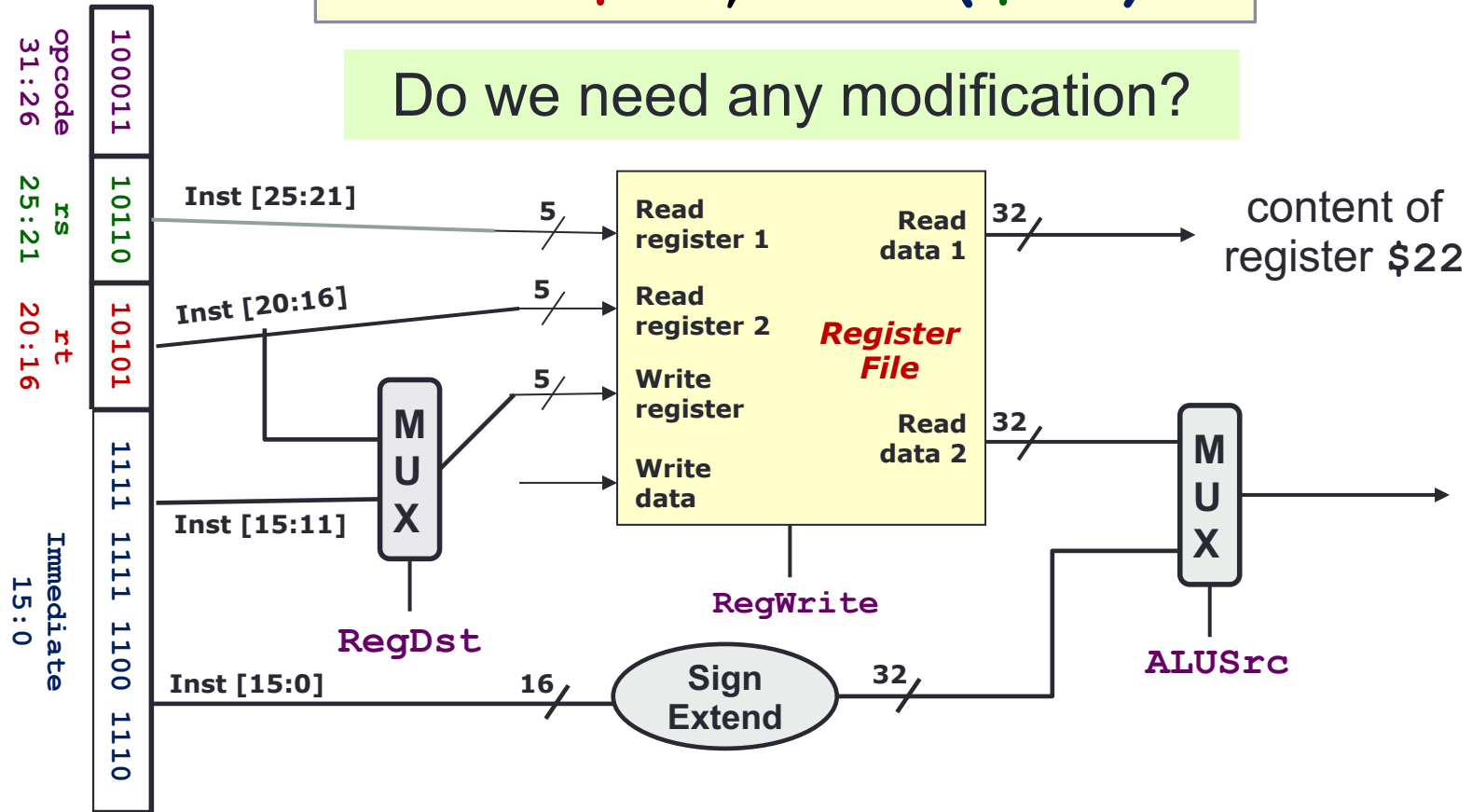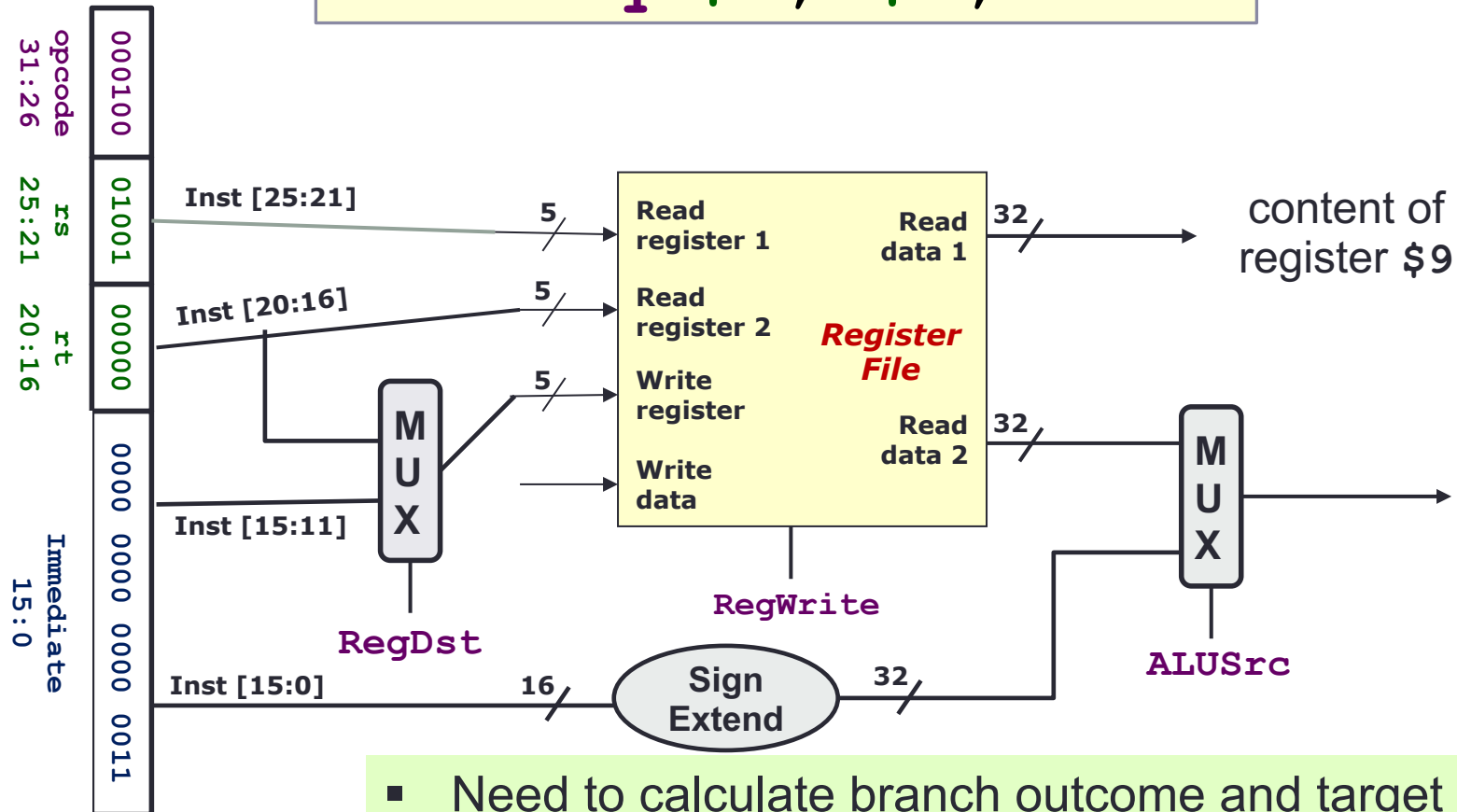Sign extend the 16-bit immediate value to 32-bit

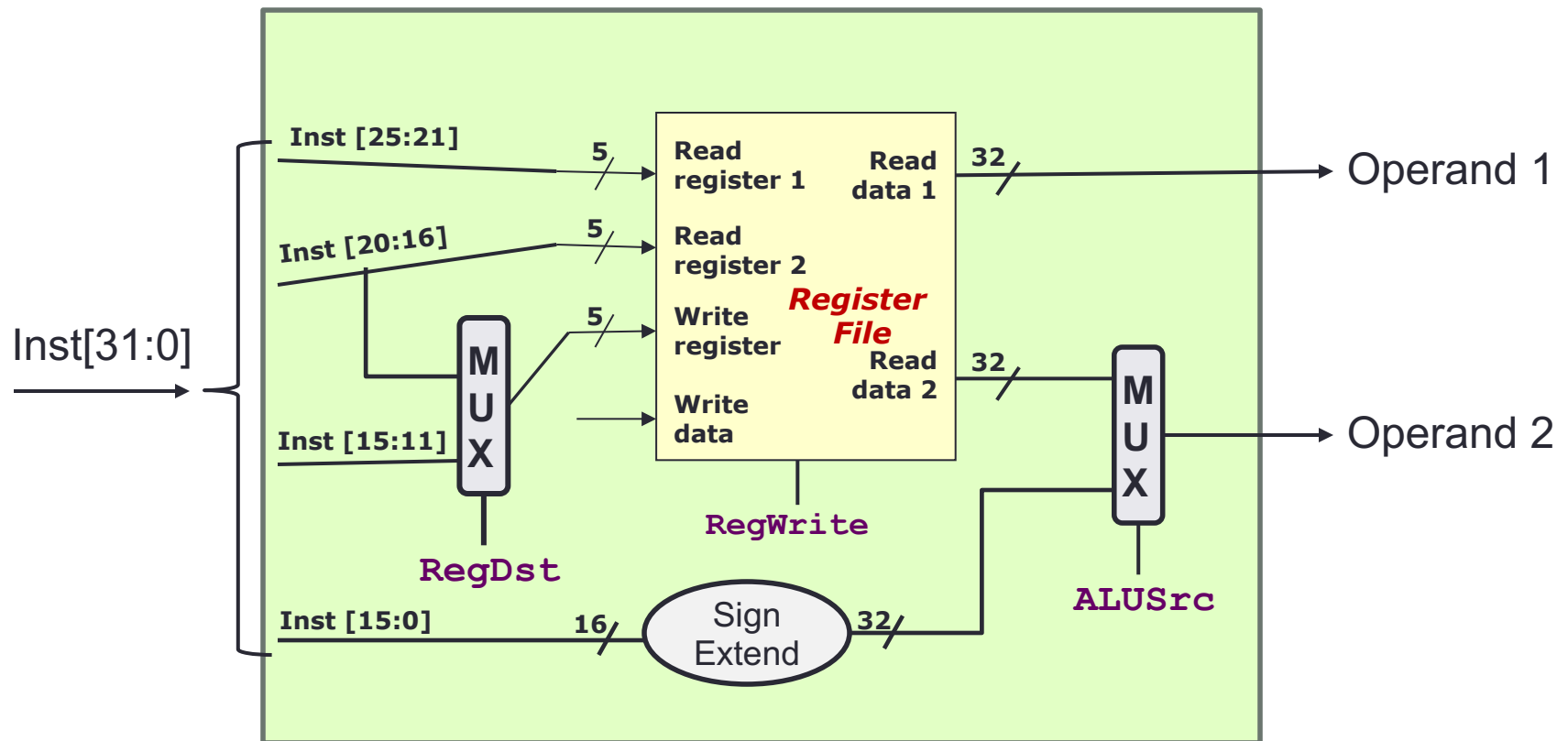# 5.2 Decode Stage: **Load Word Instruction**

5.2 Decode Stage: **Branch Instruction**

beq $9, $0, 3

- Need to calculate branch outcome and target at the same time!
- We will tackle this problem at the ALU stage
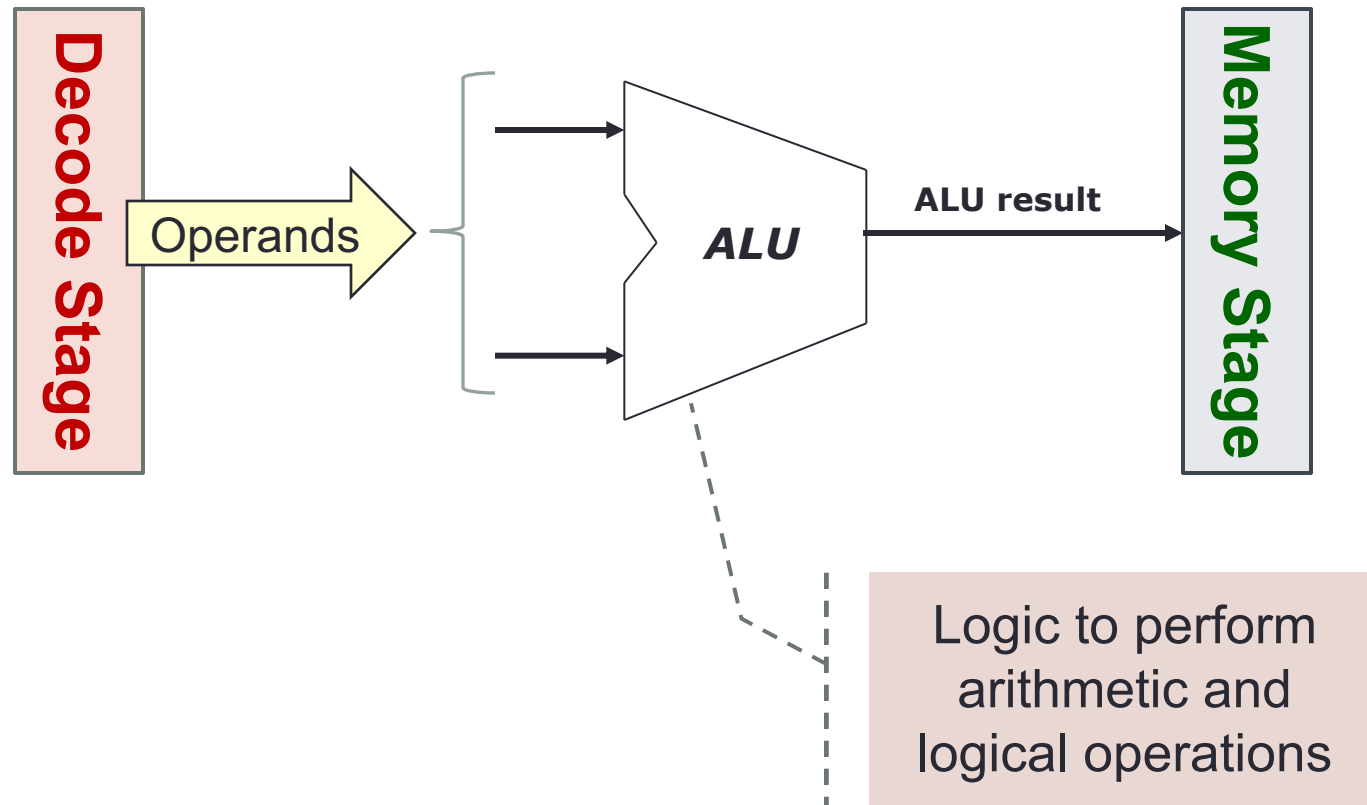
5.2 **Decode Stage**: Summary

5.3 **ALU Stage**: Requirements

- Instruction **ALU Stage**:
  - ALU = Arithmetic-Logic Unit
  - Also called the **Execution stage**
  - Perform the real work for most instructions here
    - Arithmetic (e.g. `add`, `sub`), Shifting (e.g. `sll`), Logical (e.g. `and`, `or`)
    - Memory operation (e.g. `lw`, `sw`): Address calculation
    - Branch operation (e.g. `bne`, `beq`): Perform register comparison and target address calculation

- Input from previous stage (**Decode**):
  - Operation and Operand(s)

- Output to the next stage (**Memory**):
  - Calculation result

5.3 **ALU Stage**: Block Diagram

Decode Stage

Operands

ALU

ALU result

Memory Stage

Logic to perform arithmetic and logical operations

5.3 Element: **Arithmetic Logic Unit**

- **ALU (Arithmetic Logic Unit**)
  - Combinational logic to implement arithmetic and logical operations

- **Inputs:**
  - Two 32-bit numbers

- **Control:**
  - 4-bit to decide the particular operation

- **Outputs:**
  - Result of arithmetic/logical operation
  - A 1-bit signal to indicate whether result is zero
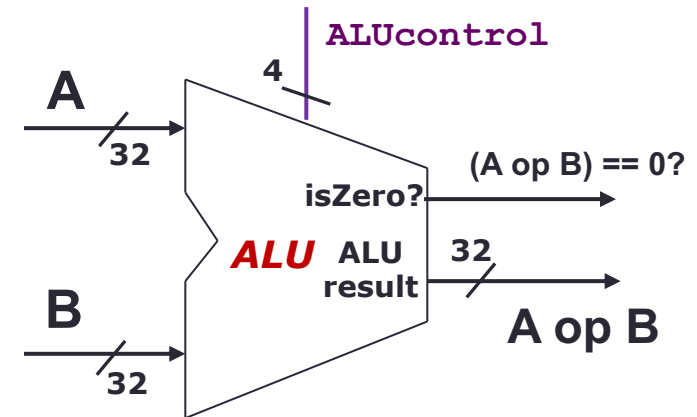
ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

5.3 ALU Stage: **Non-Branch Instructions**

- We can handle non-branch instructions easily:

add $8, $9, $10

opcode 31:26   000000
rs 25:21   01001
rt 20:16   01010
rd 15:11   01000
shamt 10:6   00000
funct 5:0   100000

Inst [25:21]   5   Read register 1   Read data 1   32
Inst [20:16]   5   Read register 2
                5   Write register   *Register File*
Inst [15:11]   Write data   Read data 2   32

MUX   RegDst

RegWrite

Inst [15:0]   16   Sign Extend   32

ALUSrc

ALUcontrol   4

ALU   isZero?

ALU result   32

ALUcontrol: Set using opcode + funct field (more in next lecture)

5.3 ALU Stage: **Branch Instructions**

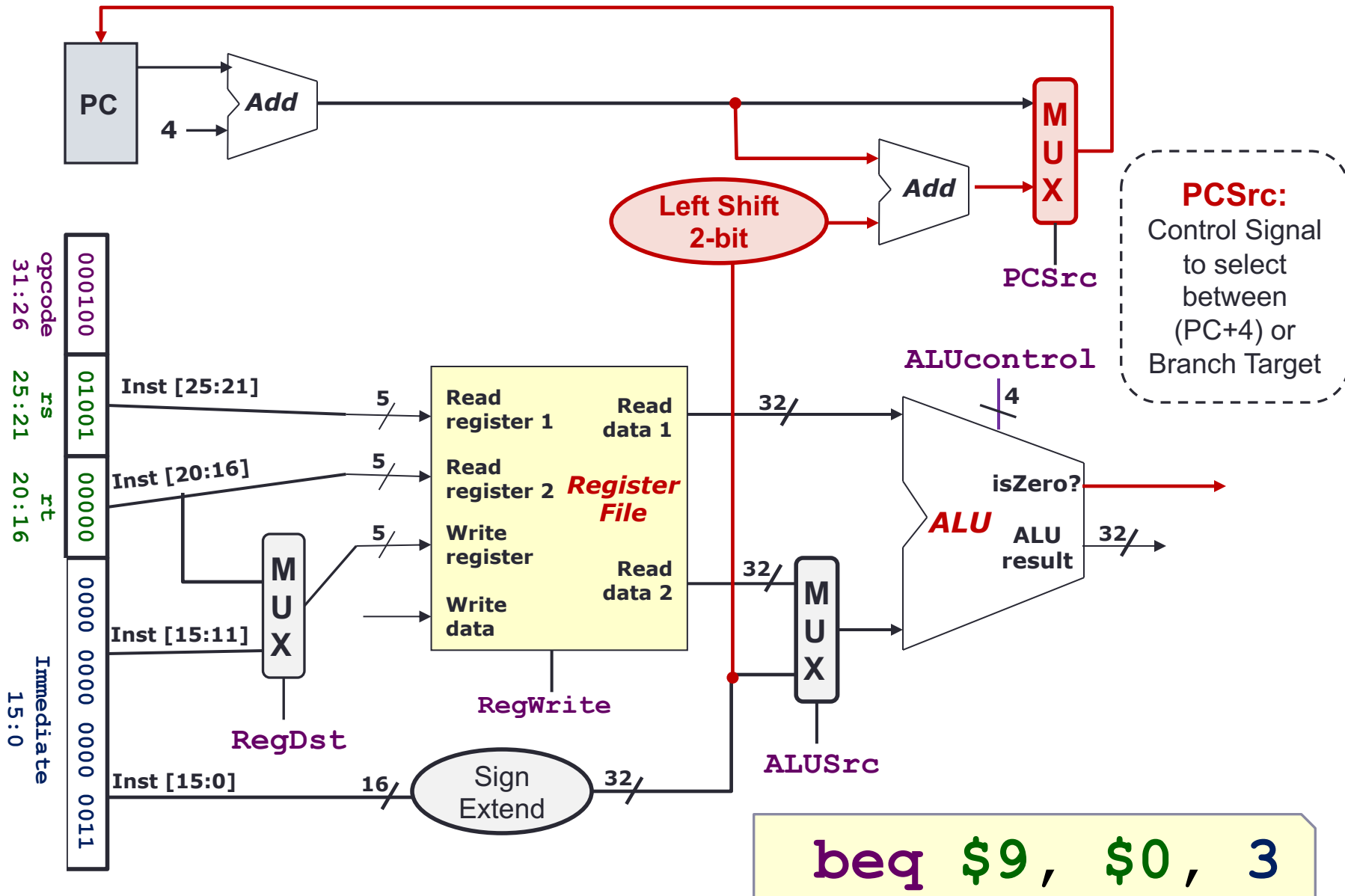- Branch instruction is harder as we need to perform two calculations:

- Example: "`beq $9, $0, 3`"
  1. **Branch Outcome:**
     - Use ALU to compare the register
     - The 1-bit "`isZero?`" signal is enough to handle equal/not equal check (how?)

  2. **Branch Target Address:**
     - Introduce additional logic to calculate the address
     - Need **PC** (from Fetch Stage)
     - Need **Offset** (from Decode Stage)

PC

Add

4

M
U
X

Add

Left Shift
2-bit

PCSrc

PCSrc:
Control Signal
to select
between
(PC+4) or
Branch Target

opcode
31:26

000100

rs
25:21

01001

rt
20:16

00000

Immediate
15:0

**0000
0000
0000
0011**

Inst [25:21]

Inst [20:16]

5

5

5

Inst [15:11]

M
U
X

RegDst

Inst [15:0]

16

Sign
Extend

32

Read
register 1

Read
register 2

Write
register

Write
data

*Register
File*

Read
data 1

Read
data 2

RegWrite

32

32

M
U
X

ALUSrc

ALUcontrol

4

ALU

isZero?

ALU
result

32

beq $9, $0, 3

5.4 **Memory Stage**: Requirement

1. Fetch
2. Decode
3. ALU
4. **Memory**
5. RegWrite

- Instruction **Memory Access Stage**:
  - Only the load and store instructions need to perform operation in this stage:
    - Use memory address calculated by ALU Stage
    - Read from or write to data memory
  - All other instructions remain idle
    - Result from ALU Stage will pass through to be used in Register Write stage (see section 5.5) if applicable

- Input from previous stage (**ALU**):
  - Computation result to be used as memory address (if applicable)

- Output to the next stage (**Register Write**):
  - Result to be stored (if applicable)

5.4 **Memory Stage**: Block Diagram
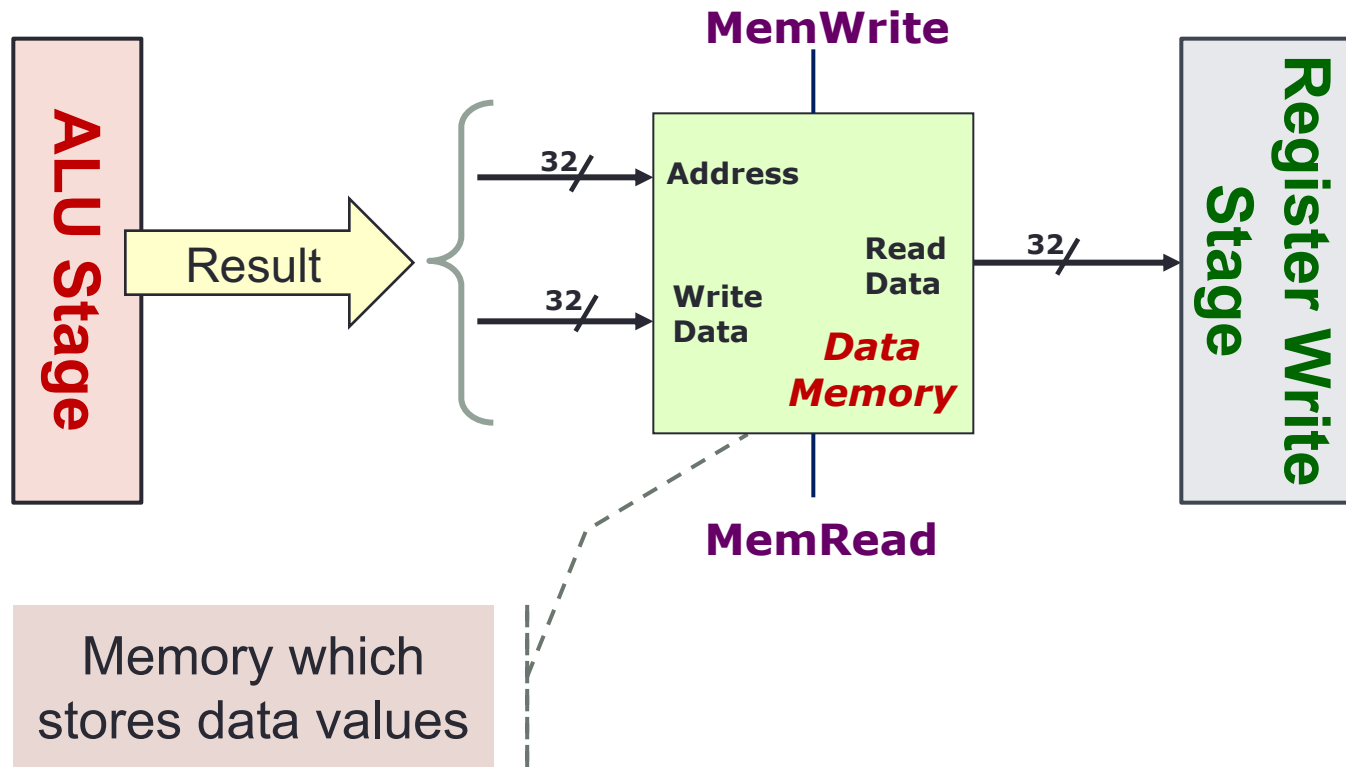
5.4 Element: **Data Memory**

- Storage element for the data of a program

- **Inputs:**
  - Memory Address
  - Data to be written (Write Data) for store instructions

- **Control:**
  - Read and Write controls; only one can be asserted at any point of time

- **Output:**
  - Data read from memory (Read Data) for load instructions
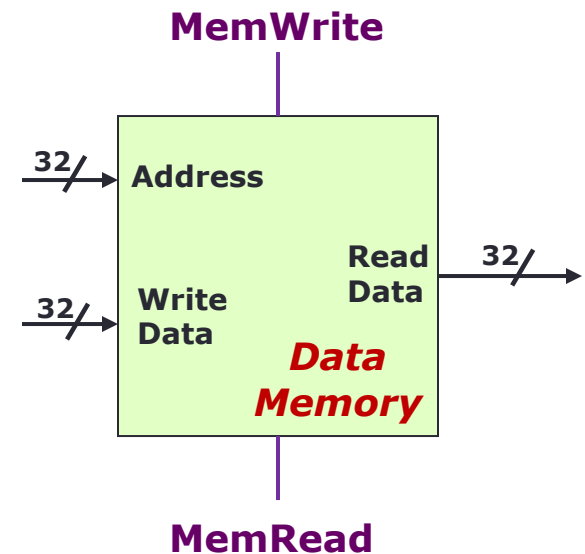
5.4 Memory Stage: **Load Instruction**

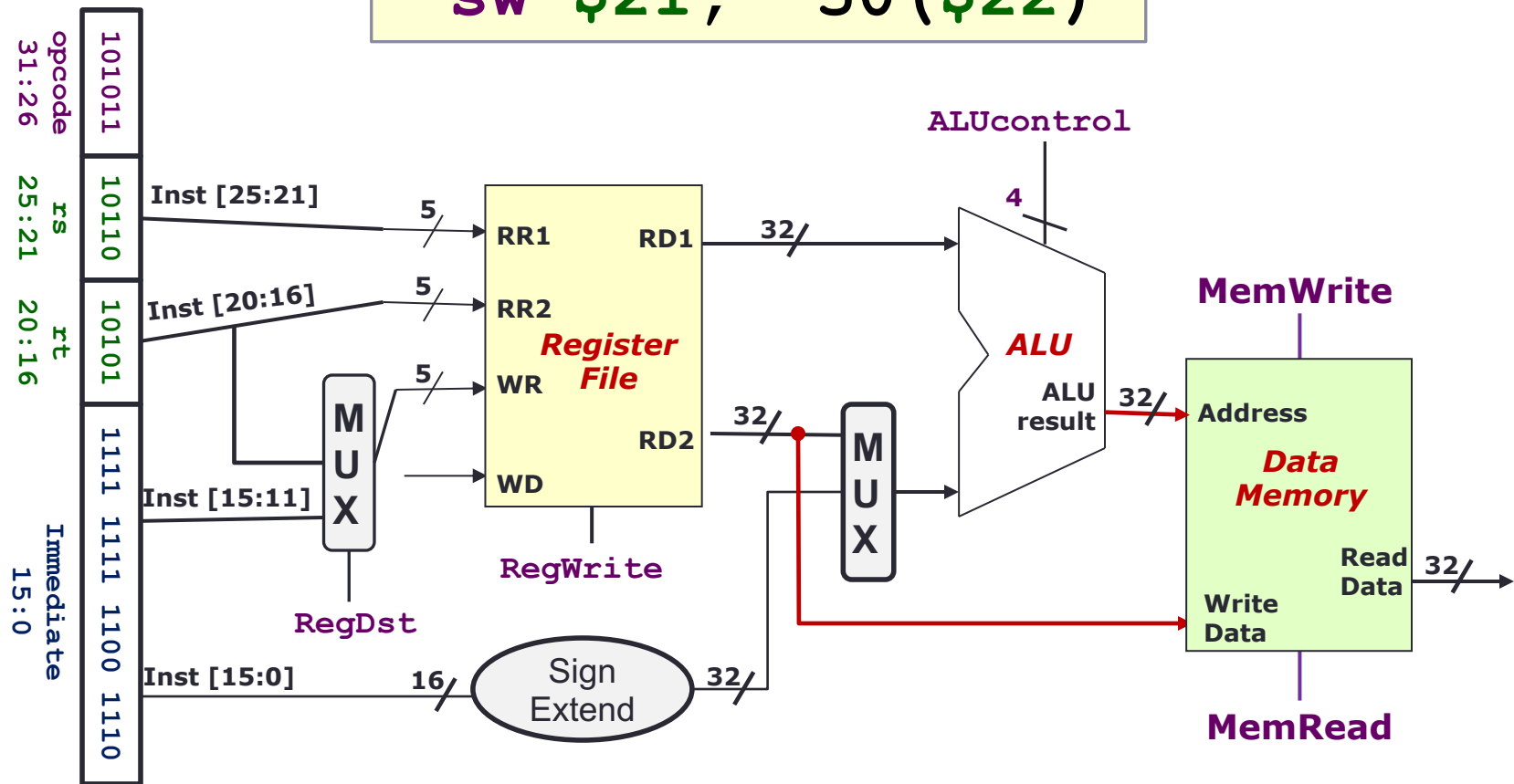- Only relevant parts of Decode and ALU Stages are shown


```
lw $21, -50($22)
```
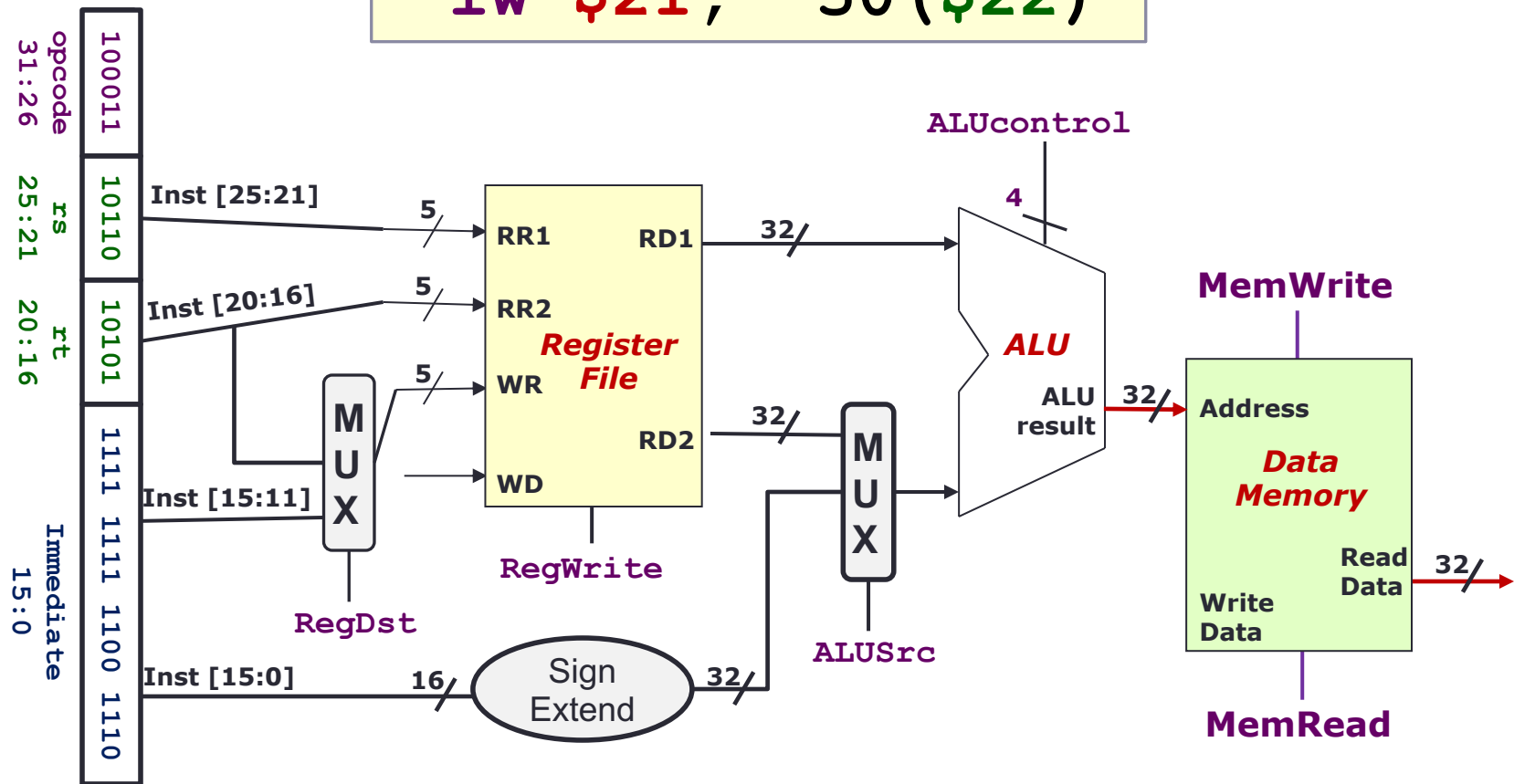
5.4 Memory Stage: **Store Instruction**

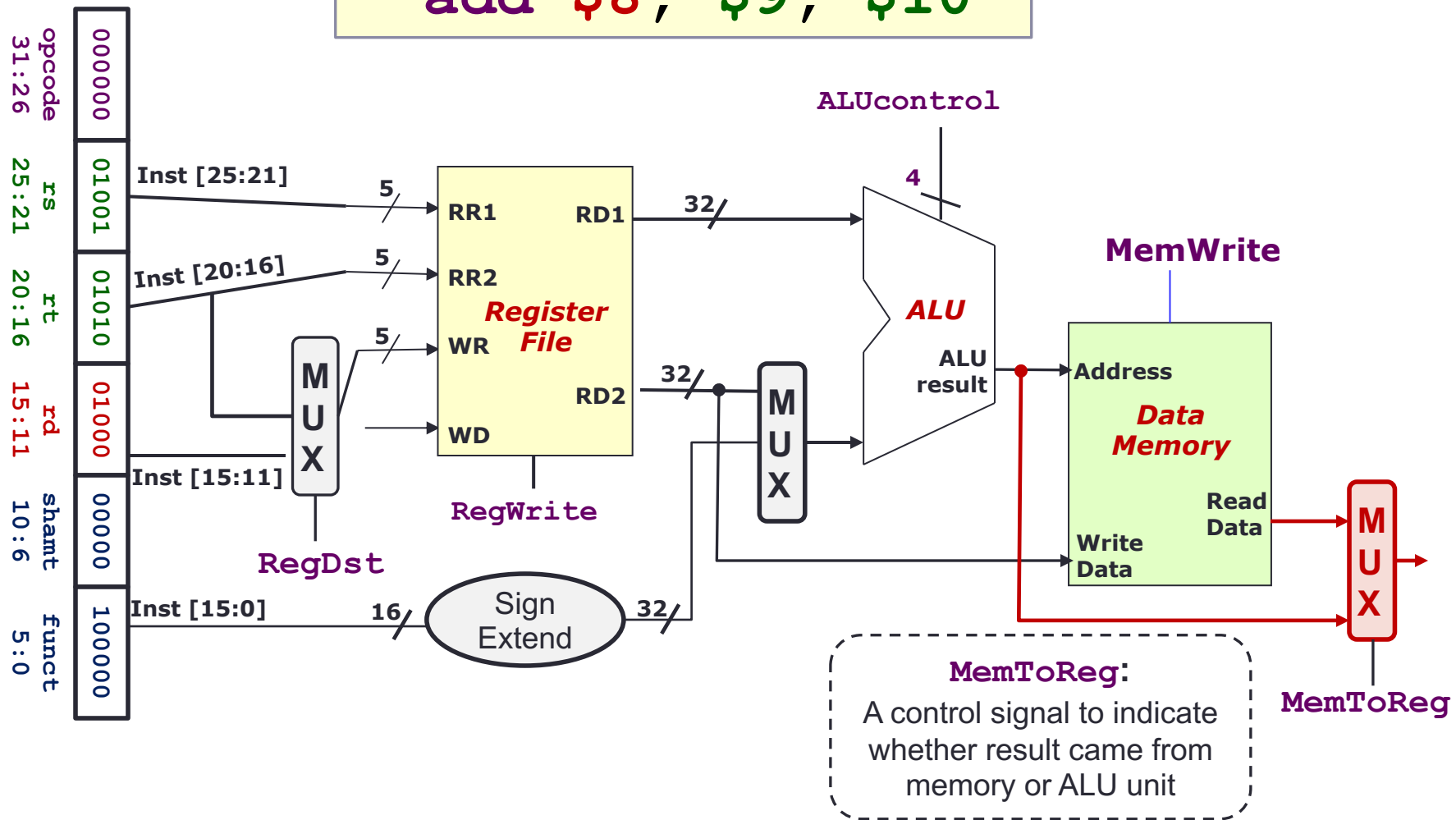- Need *Read Data 2* (from Decode stage) as the *Write Data*

```
sw $21, -50($22)
```

5.4 Memory Stage: **Non-Memory Inst.**

- Add a multiplexer to choose the result to be stored

> **add $8, $9, $10**

MemToReg:
A control signal to indicate whether result came from memory or ALU unit

5.5 **Register Write Stage**: Requireme

1. Fetch
2. Decode
3. ALU
4. Memory
5. **RegWrite**

- Instruction **Register Write Stage**:
  - Most instructions write the result of some computation into a register
    - Examples: arithmetic, logical, shifts, loads, set-less-than
    - Need destination register number and computation result
  - Exceptions are stores, branches, jumps:
    - There are no results to be written
    - These instructions remain idle in this stage

- Input from previous stage (**Memory**):
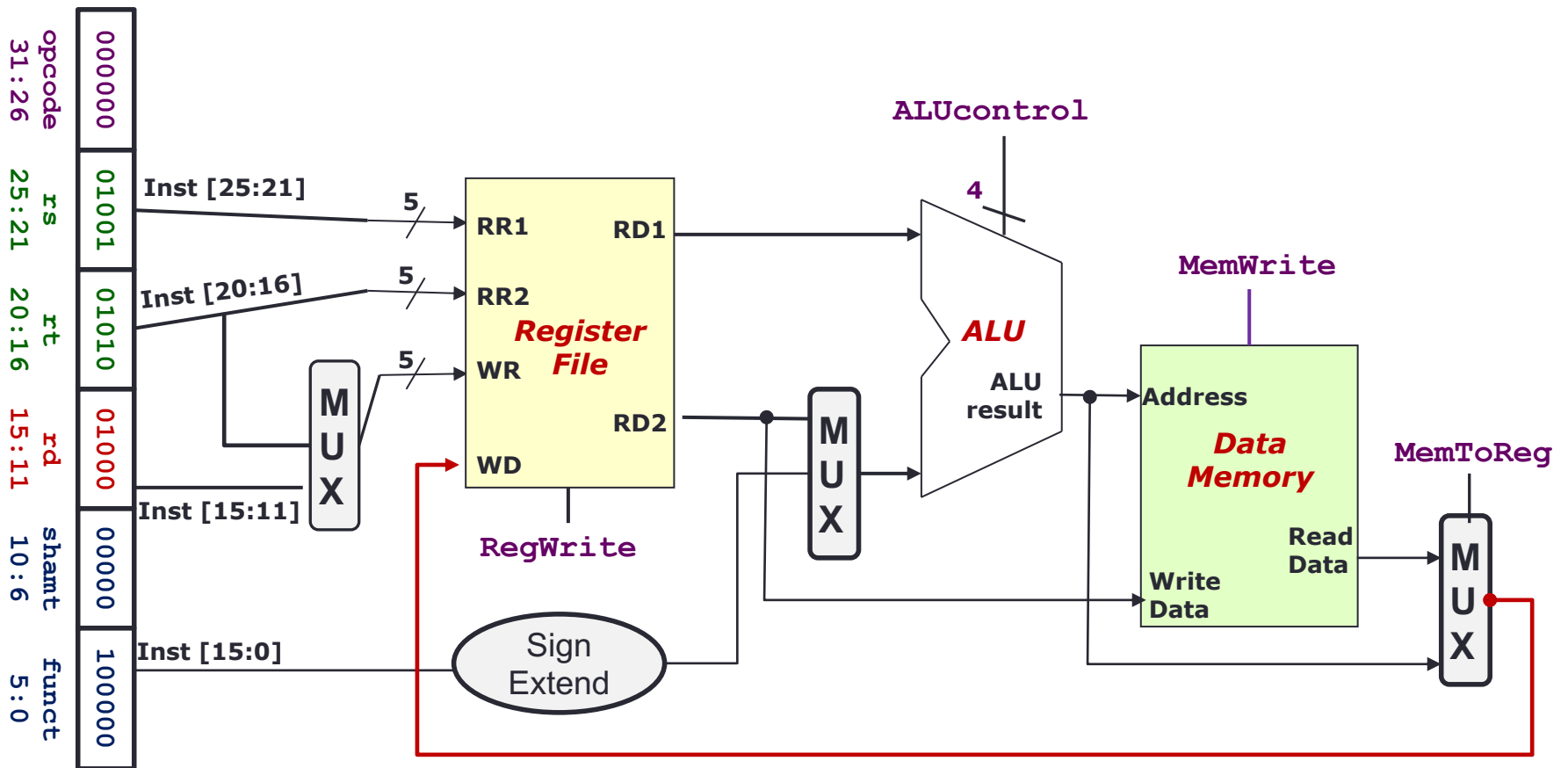 - Computation result either from memory or ALU

5.5 **Register Write Stage**: Block Diagram
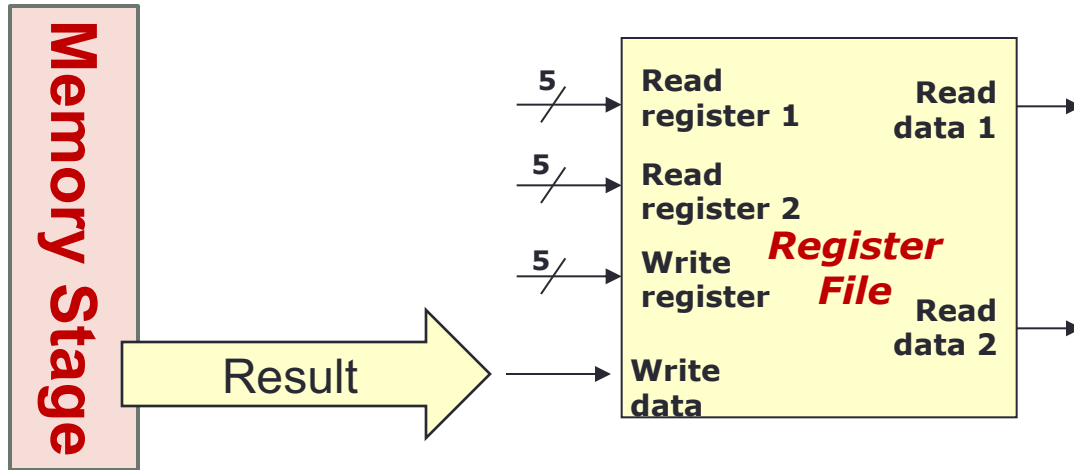
- Result Write stage has no additional element:
  - Basically just route the correct result into register file
  - The **Write Register** number is generated way back in the **Decode** Stage

5.5 Register Write Stage: **Routing**

add **$8**, **$9**, **$10**

6. The Complete Datapath!

- We have just finished "designing" the datapath for a subset of MIPS instructions:
  - Shifting and Jump are not supported
- Check your understanding:
  - Take the complete datapath and play the role of controller:
    - See how supported instructions are executed
    - Figure out the correct control signals for the datapath elements

- Coming up next lecture: **Control**

Complete Datapath
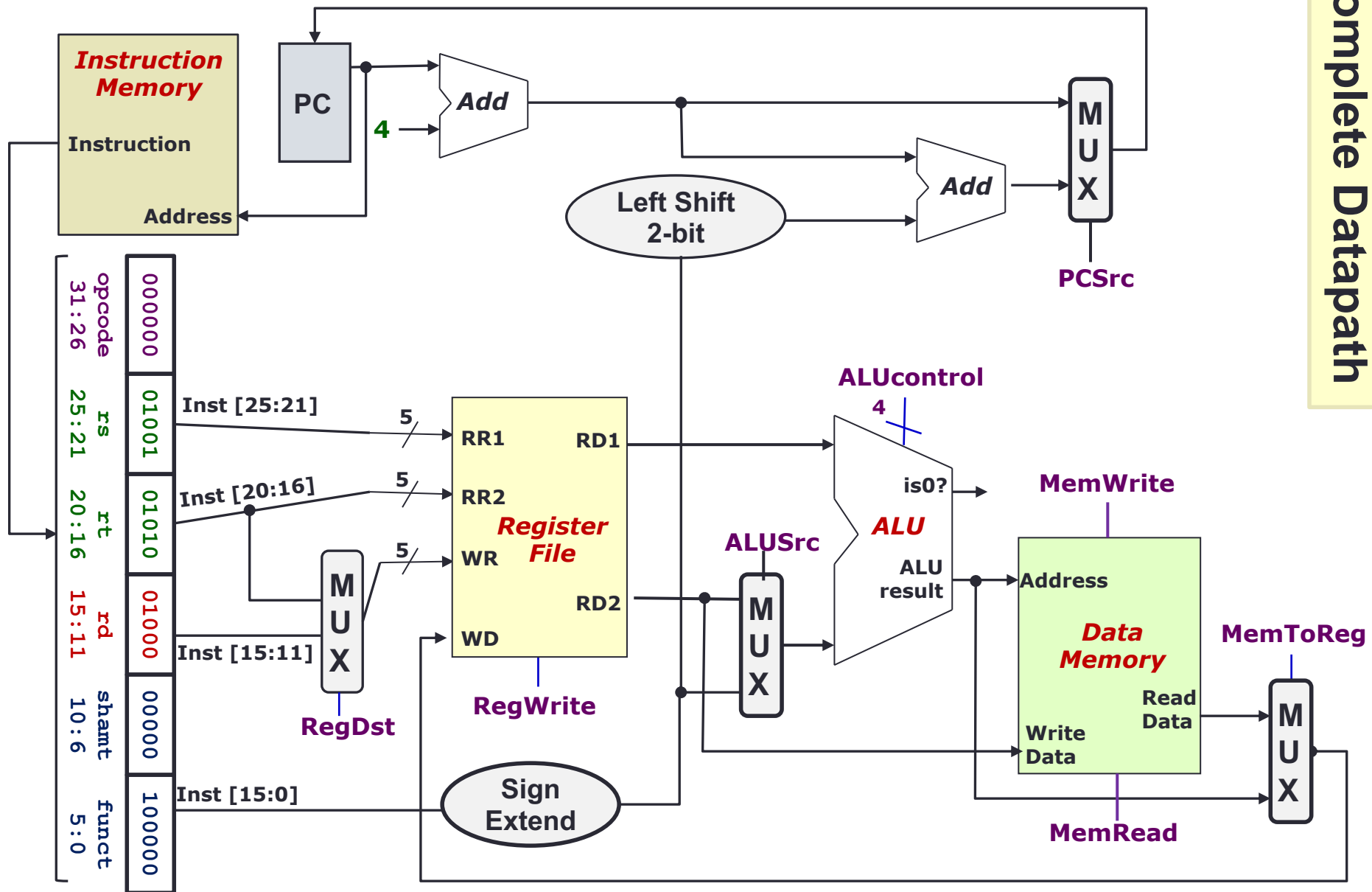
7. Brief Recap (1/4)

- Lecture #7, Slide 4

```
High-level          swap(int v[], int k)
language            {int temp;
program                 temp = v[k];
(in C)                  v[k] = v[k+1];
                        v[k+1] = temp;
                    }
```

                    Compiler

```
Assembly            swap:
language                muli $2, $5,4
program                 add  $2, $4,$2
(for MIPS)              lw   $15, 0($2)
                        lw   $16, 4($2)
                        sw   $16, 0($2)
                        sw   $15, 4($2)
                        jr   $31
```

                    Assembler

```
Binary machine      00000000101000010000000000011000
language            00000000000110000001100000100001
program             10001100011000100000000000000000
(for MIPS)          10001100111100100000000000000100
                    10101100111100100000000000000000
                    10101100011000100000000000000100
                    00000011111000000000000000001000
```

➤ Write program in high-level language (e.g., **C**)

```
if(x != 0) {
   a[0] = a[1] + x;
}
```

7. Brief Recap (2/4)

- ## Lecture #7, Slide 4

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Assembly language program (for MIPS)

```
swap:
       muli $2, $5,4
       add  $2, $4,$2
       lw   $15, 0($2)
       lw   $16, 4($2)
       sw   $16, 0($2)
       sw   $15, 4($2)
       jr   $31
```

Assembler

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111100000000000000001000
```
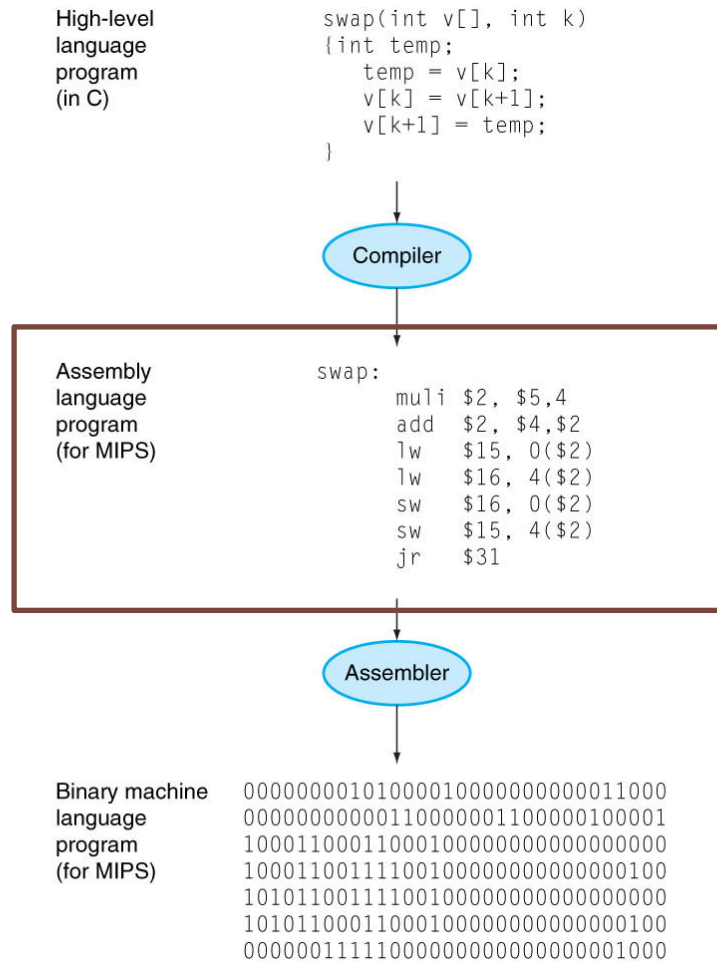
> Compiler translates to assembly language (e.g., **MIPS**)

```
        beq $16, $0, Else
        lw  $8, 4($17)
        add $8, $8, $16
        sw  $8, 0($17)
Else:
```

7. Brief Recap (3/4)

- Lecture #7, Slide 4

High-level
language
program
(in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Compiler

Assembly
language
program
(for MIPS)

```
swap:
    muli  $2, $5,4
    add   $2, $4,$2
    lw    $15, 0($2)
    lw    $16, 4($2)
    sw    $16, 0($2)
    sw    $15, 4($2)
    jr    $31
```

Assembler

Binary machine
language
program
(for MIPS)
```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

➢ Assembler translates to machine code (i.e., **binaries**)

```
0001 0010 0000 0000
0000 0000 0000 0011

1000 1110 0010 1000
0000 0000 0000 0100

0000 0010 0000 1000
0100 0000 0001 0100

1010 1110 0010 1000
0000 0000 0000 0000
```

7. Brief Recap (4/4)

- ## Lecture #7, Slide 4

> ➤ Processor executes the machine code (i.e., **binaries**)
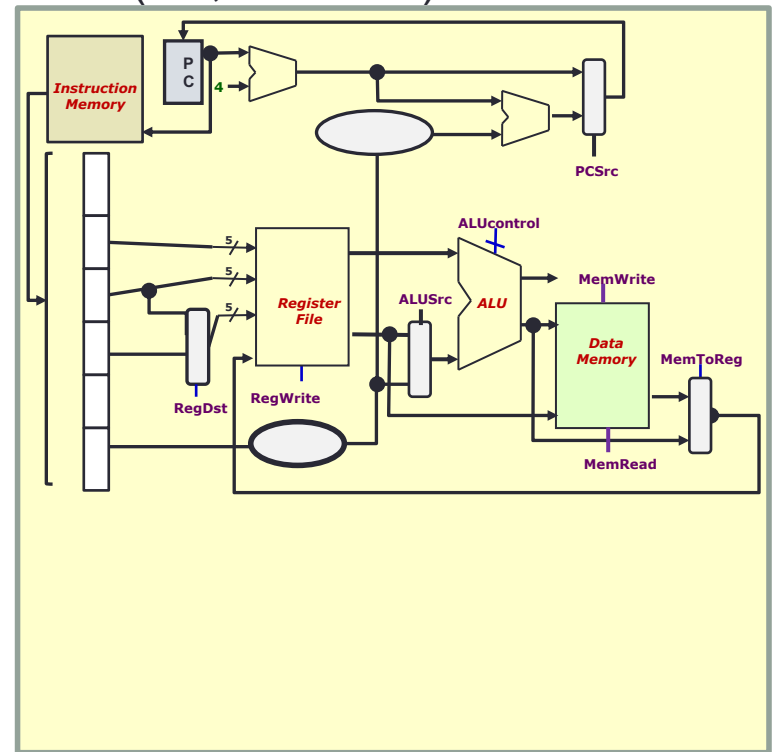
8. From C to Execution

- We play the role of Programmer, Compiler, Assembler, and Processor

  - Program:
    ```
    if(x != 0) {
      a[0] = a[1] + x;
    }
    ```

  - Programmer:
    - Show the workflow of compiling, assembling, and executing C program

  - Compiler:
    - Show how the program is compiled into MIPS

  - Assembler:
    - Show how the MIPS is translated into binaries

  - Processor:
    - Show how the datapath is activated in the processor

8.1   Writing C Program

- Edit, Compile, Execute: Lecture #2, Slide 5

Edit
eg: vim recap.c

produces

Source code
recap.c

```
if(x != 0) {
 a[0] = a[1] + x;
}
```

Compile
eg: gcc recap.c

produces

Executable code
a.out

Execute
eg: a.out

produces

Program output
No output expected

Cannot
compile?

Incorrect
result?

Edit   Compile

Execute

```
recap.c

if(x != 0) {
 a[0] =
    a[1] + x;
}
```
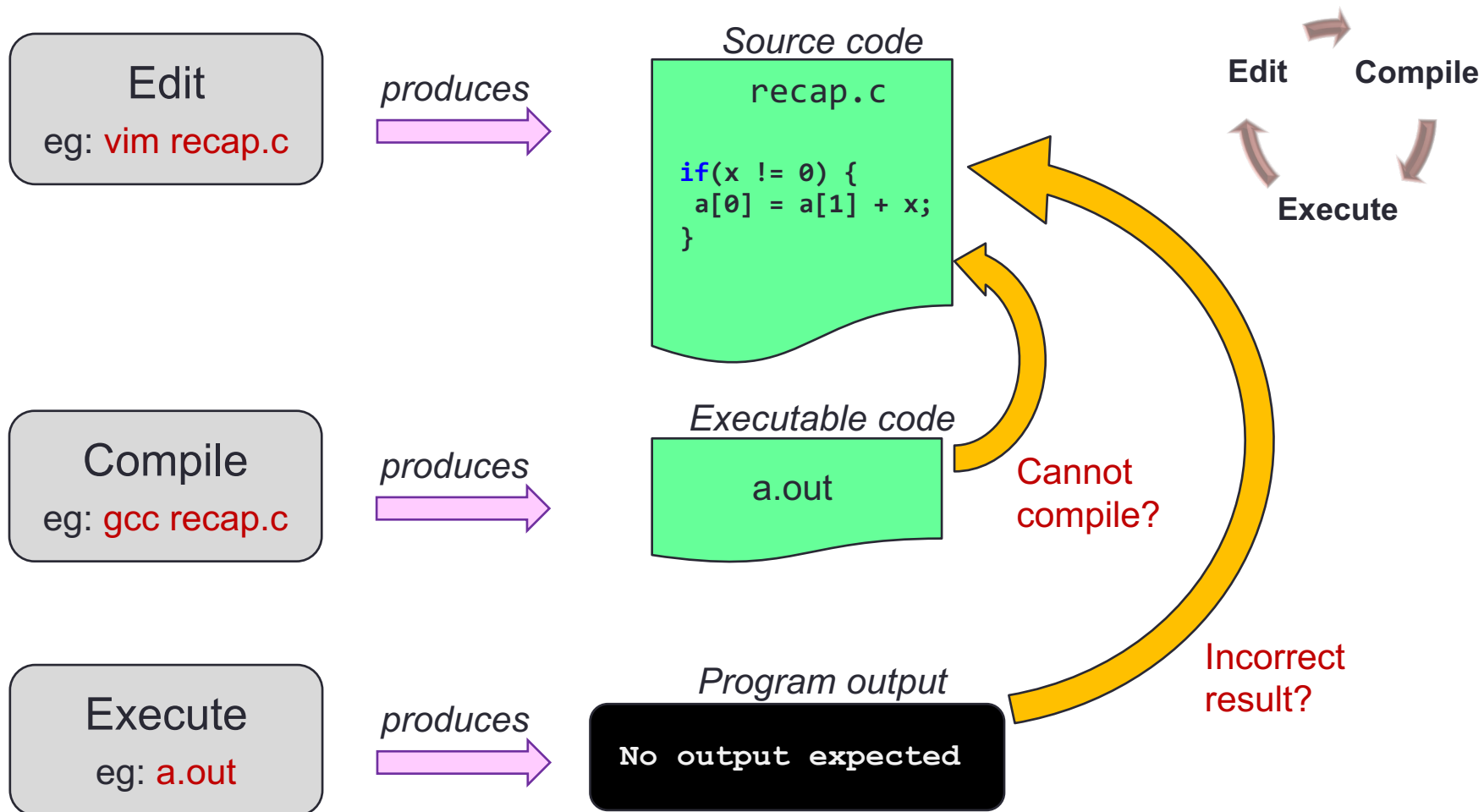
8.2   Compiling to MIPS (1/7)

- Key Idea #1:
Compilation is a *structured process*
```
if(x != 0) {
  a[0] = a[1] + x;
}
```
  - Each structure can be compiled *independently*

Inner Structure

```
  a[0] = a[1] + x;
```

Outer Structure

```
if(x != 0) {


}
```

8.2   Compiling to MIPS (2/7)

- Key Idea #2:
  Variable-to-Register Mapping

```
if(x != 0) {
  a[0] = a[1] + x;
}
```

  - Let the mapping be:

Variable	Register Name	Register Number
x	$s0	$16
a	$s1	$17

8.2   Compiling to MIPS (3/7)

```
Mapping:

x: $16
a: $17
```

```
recap.c

if(x != 0) {
  a[0] =
    a[1] + x;
}
```

- Common Technique #1:
  Invert the condition for shorter code
  (Lecture #8, Slide 22)

Outer Structure

```
if(x != 0) {



}
```

Outer MIPS Code

```
        beq $16, $0, Else

        # Inner Structure

Else:
```

8.2   Compiling to MIPS (4/7)

```
Mapping:

x: $16
a: $17
$t1: $8
```

```
recap.c

if(x != 0) {
 a[0] =
    a[1] + x;
}
```

- Common Technique #2:
Break complex operations, use temp register
(Lecture #7, Slide 29)

Inner Structure

```
 a[0] = a[1] + x;
```

Simplified Inner Structure

```
$t1  = a[1];
$t1  = $t1 + x;
a[0] = $t1;
```

8.2   Compiling to MIPS (5/7)

Mapping:

x: $16
a: $17
$t1: $8

recap.c

if(x != 0) {
  a[0] =
    a[1] + x;
}

- Common Technique #3:
  Array access is **lw**, array update is **sw**
  (Lecture #8, Slide 13)

Simplified Inner Structure

```
$t1  = a[1];
$t1  = $t1 + x;
a[0] = $t1;
```

Inner MIPS Code

```
lw  $8, 4($17)
add $8, $8, $16
sw  $8, 0($17)
```

8.2   Compiling to MIPS (6/7)

```
Mapping:

x: $16
a: $17
$t1: $8
```

```
recap.c

if(x != 0) {
 a[0] =
    a[1] + x;
}
```

- Common Error:
  Assume that the address of the next word can
  be found by incrementing the address in a
  register by 1 instead of by the word size in bytes

  - Example:

    - `$t1  = a[1];`
      is translated to
      `lw  $8, 4($17)`
      instead of
      `lw  $8, 1($17)`

```
Mapping:

x: $16
a: $17
$t1: $8
```

```
recap.c

if(x != 0) {
  a[0] =
    a[1] + x;
}
```

8.2   Compiling to MIPS (7/7)

- Last Step:
  Combine the two structures logically

Inner MIPS Code

```
    lw   $8, 4($17)
    add  $8, $8, $16
    sw   $8, 0($17)
```

Outer MIPS Code

```
        beq $16, $0, Else

              # Inner Structure

Else:
```

Combined MIPS Code

```
        beq $16, $0, Else
        lw  $8, 4($17)
        add $8, $8, $16
        sw  $8, 0($17)
Else:
```

```
recap.mips

   beq $16, $0, Else
   lw  $8, 4($17)
   add $8, $8, $16
   sw  $8, 0($17)
Else:
```

8.3   Assembling to Binary (1/6)

■ Instruction Types Used:

1.   R-Format: (Lecture #9, Slide 8)

   • **opcode** $rd, $rs, $rt

6	5	5	5	5	6
opcode	**rs**	**rt**	**rd**	**shamt**	**funct**

2.   I-Format: (Lecture #9, Slide 14)

   • **opcode** $rt, $rs, immediate

6	5	5	16
opcode	**rs**	**rt**	**immediate**

3.   Branch: (Lecture #9, Slide 22)

   • Uses I-Format
   • **PC = (PC + 4) + (immediate × 4)**

8.3   Assembling to Binary (2/6)

recap.mips

```
   beq $16, $0, Else
   lw  $8, 4($17)
   add $8, $8, $16
   sw  $8, 0($17)
Else:
```

- **beq $16, $0, Else**

  - Compute immediate value (Lecture #9, Slide 27)

    - **immediate** = 3

  - Fill in fields (*refer to MIPS Reference Data*)

6	5	5	16
4	16	0	3

  - Convert to binary

000100	10000	00000	0000000000000011

```
    beq $16, $0, Else
    lw  $8, 4($17)
    add $8, $8, $16
    sw  $8, 0($17)
Else:
```

+3

8.3   Assembling to Binary (3/6)

- **lw $8, 4($17)**

  - Fill in fields (*refer to MIPS Reference Data*)

6	5	5	16
35	17	8	4

  - Convert to binary

100011	10001	01000	0000000000000100

```
0001 0010 0000 0000 0000 0000 0000 0011
        lw  $8, 4($17)
        add $8, $8, $16
        sw  $8, 0($17)
Else:
```

8.3 Assembling to Binary (4/6)

```
recap.mips

      beq $16, $0, Else
      lw  $8, 4($17)
      add $8, $8, $16
      sw  $8, 0($17)
Else:
```

- **add $8, $8, $16**

  - Fill in fields (*refer to MIPS Reference Data*)

6	5	5	5	5	6
0	8	16	8	0	32

  - Convert to binary

000000	01000	10000	01000	00000	100000

```
0001 0010 0000 0000 0000 0000 0000 0011
1000 1110 0010 1000 0000 0000 0000 0100
      add $8, $8, $16
      sw  $8, 0($17)
Else:
```

8.3   Assembling to Binary (5/6)

- **sw $8, 0($17)**

  - Fill in fields (*refer to MIPS Reference Data*)

6	5	5	16
43	17	8	0

  - Convert to binary

101011	10001	01000	0000000000000000

```
0001 0010 0000 0000 0000 0000 0000 0011
1000 1110 0010 1000 0000 0000 0000 0100
0000 0001 0001 0000 0100 0000 0010 0000
        sw   $8, 0($17)
Else:
```

```
recap.mips

  beq $16, $0, Else
  lw  $8, 4($17)
  add $8, $8, $16
  sw  $8, 0($17)
Else:
```

8.3   Assembling to Binary (6/6)

■ Final Binary

  ■ Hard to read?

  ■ Don't worry, this is intended for machine
    not for human!

```
0001 0010 0000 0000 0000 0000 0000 0011
1000 1110 0010 1000 0000 0000 0000 0100
0000 0001 0001 0000 0100 0000 0010 0000
1010 1110 0010 1000 0000 0000 0000 0000
```

8.4   Execution (Datapath)

- ## Given the binary
  - Assume two possible executions:
    1. `$16 == $0`   (*shorter*)
    2. `$16 != $0`   (*Longer*)
  - Convention:

    Fetch: ⟶   Memory: ⟶

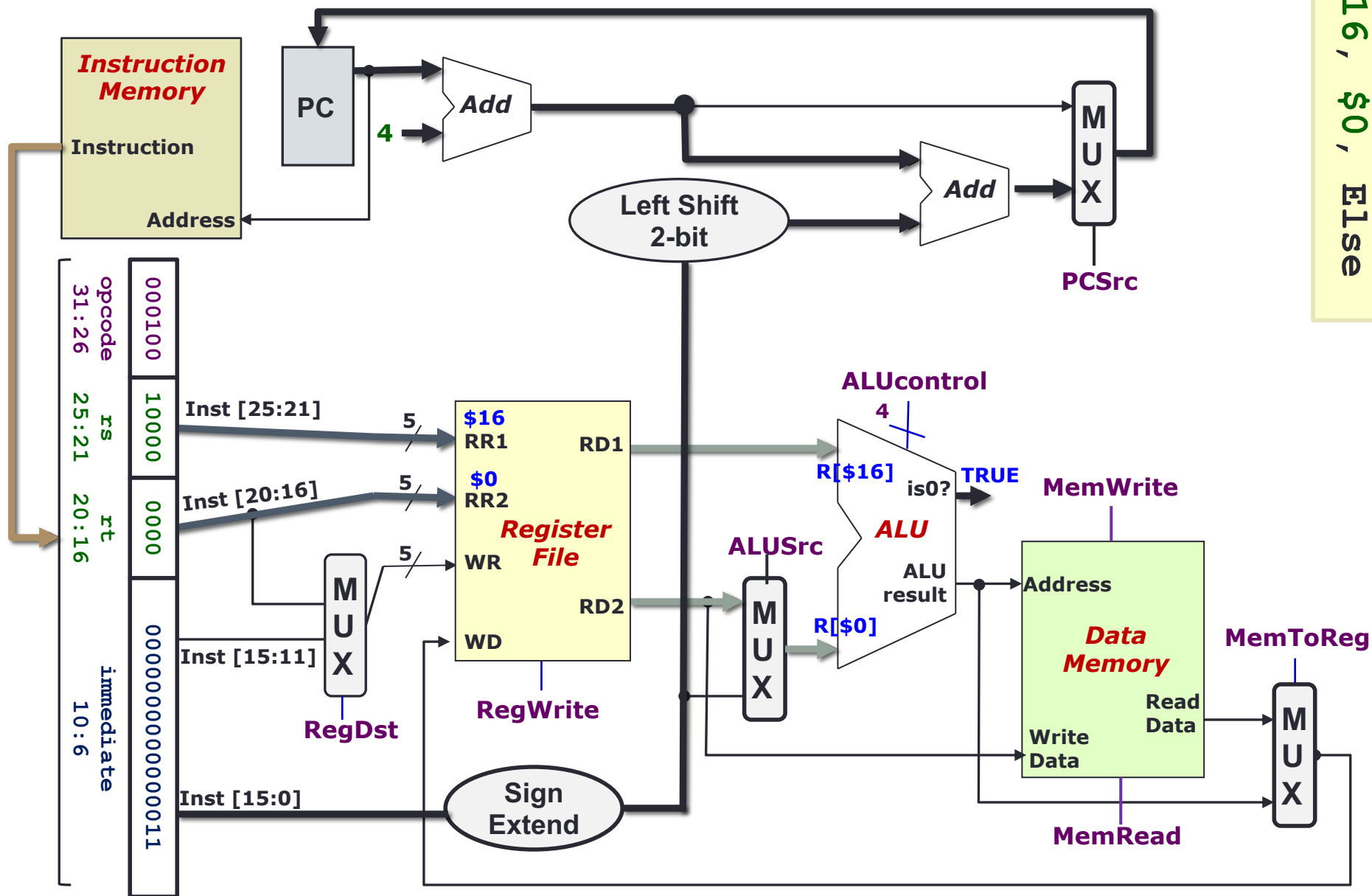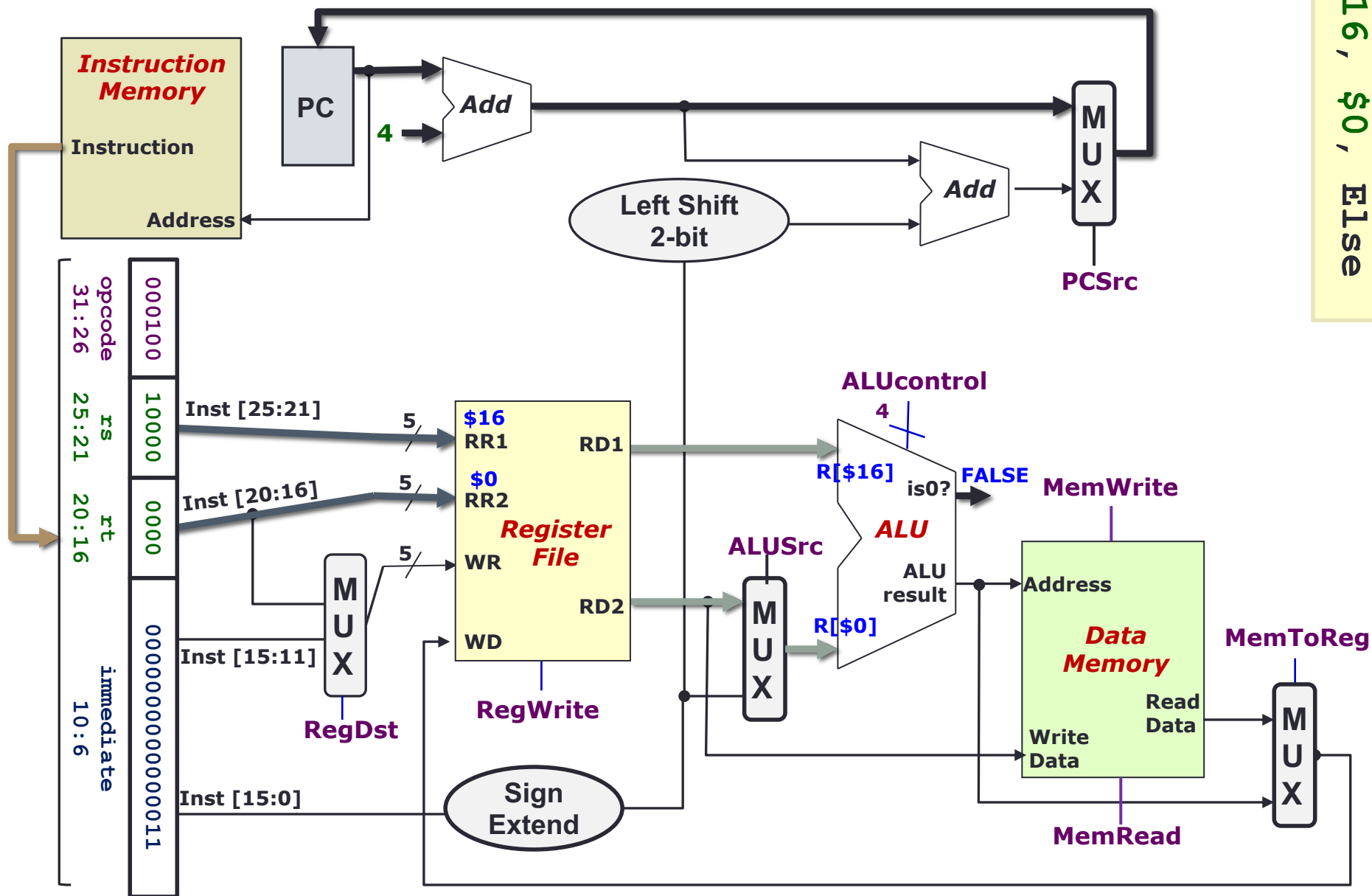   Decode: ⟶   Reg Write: ⟶

    ALU: ⟶   Other: ⟶

```
0001 0010 0000 0000 0000 0000 0000 0011
1000 1110 0010 1000 0000 0000 0000 0100
0000 0001 0001 0000 0100 0000 0010 0000
1010 1110 0010 1000 0000 0000 0000 0000
```
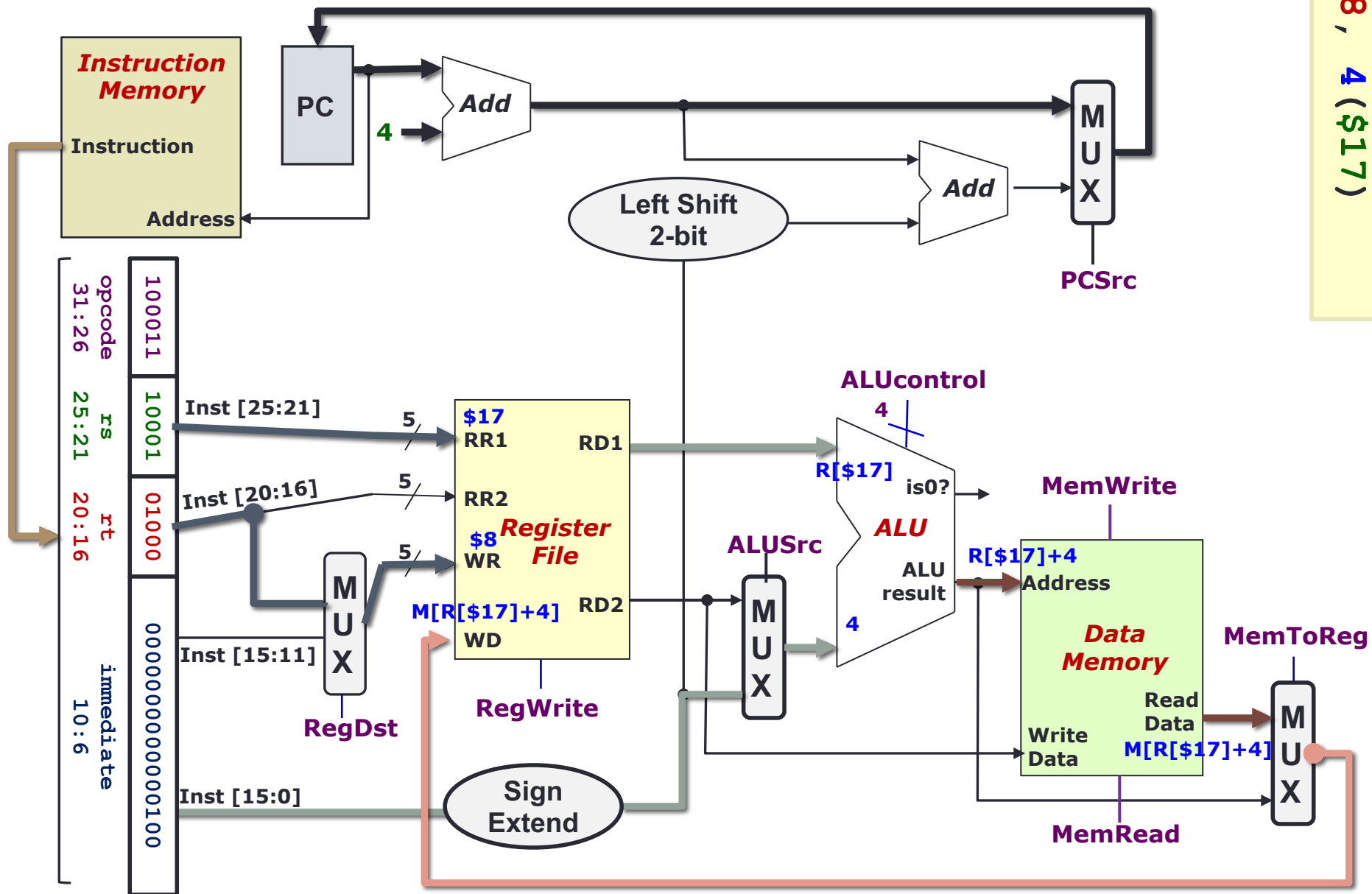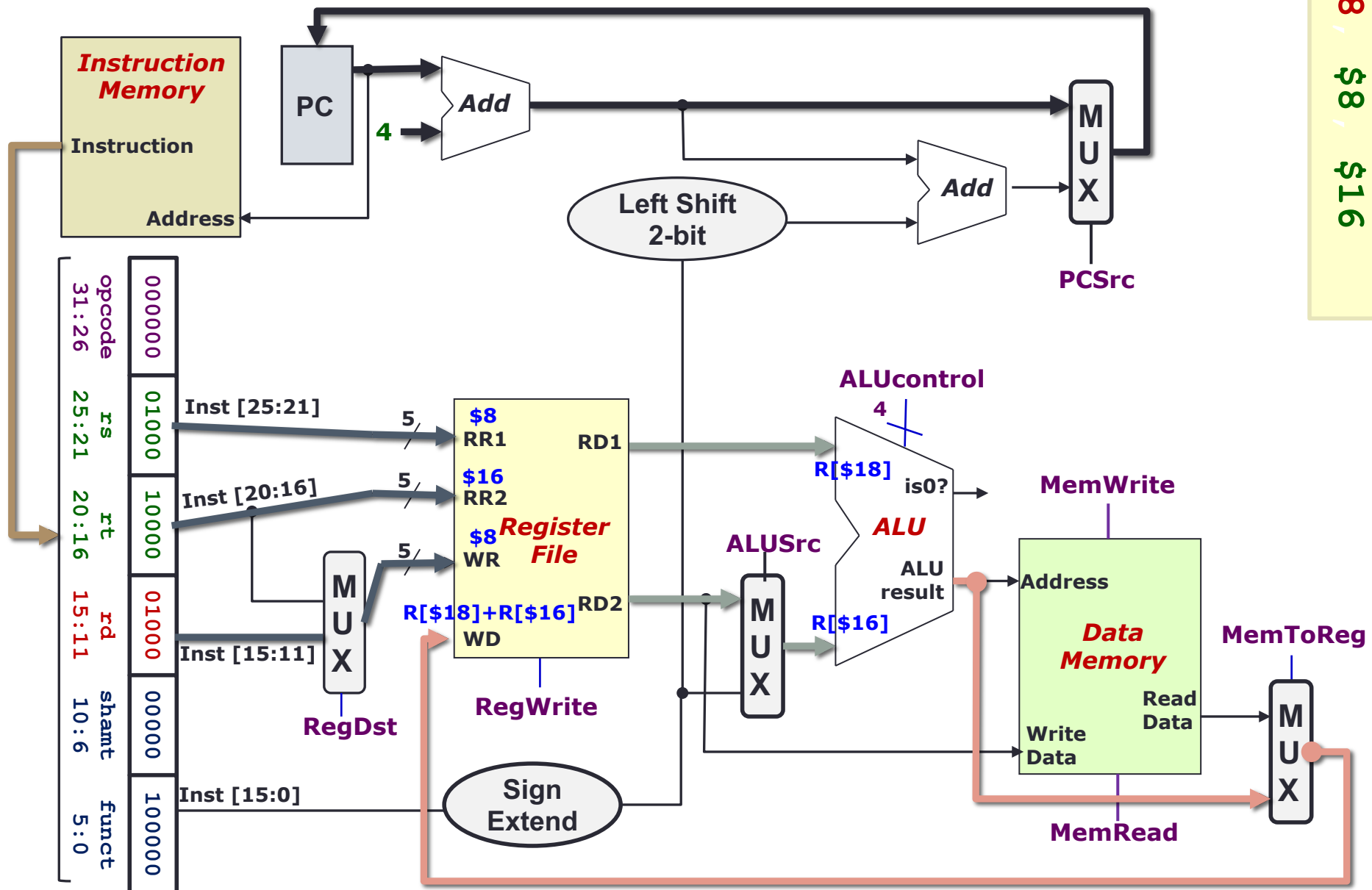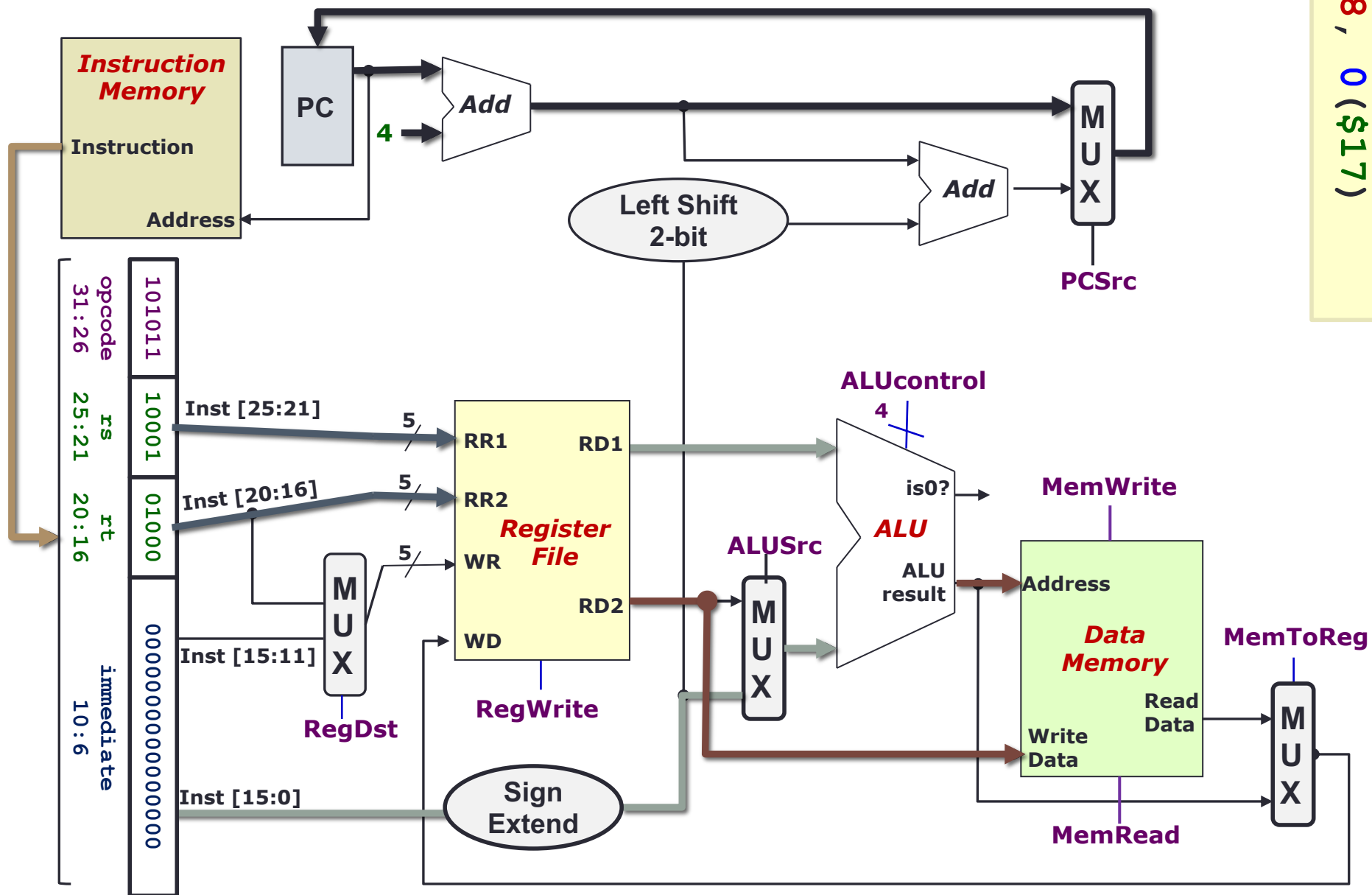
Assume $16 == $0

beq $16, $0, Else

Assume $16 != $0

beq $16, $0, Else

Assume $16 != $0

lw $8, 4($17)

Instruction Memory

Instruction

PC

Add

4

Address

Left Shift 2-bit

Add

MUX

PCSrc

opcode 31:26 — 100011

rs 25:21 — 10001

rt 20:16 — 01000

immediate 10:6 — 0000000000000100

Inst [25:21]

Inst [20:16]

Inst [15:11]

Inst [15:0]

5

5

5

M U X

RegDst

$17 RR1

RR2

$8 WR

M[R[$17]+4] WD

Register File

RegWrite

RD1

RD2

Sign Extend

ALUcontrol

4

M U X

ALUSrc

R[$17]

is0?

ALU

ALU result

R[$17]+4

4

MemWrite

Address

Data Memory

Write Data

Read Data

M[R[$17]+4]

MemRead

MemToReg

M U X

Assume $16 != $0

Assume $16 != $0

sw $8, 0($17)

Reading

- **The Processor: Datapath and Control**
  - COD Chapter 5 Sections 5.1 – 5.3 (3rd edition)
  - COD Chapter 4 Sections 4.1 – 4.3 (4th edition)

End of File