CS2100 COMPUTER ORGANISATION

http://www.comp.nus.edu.sg/~cs2100/

Note

You will only learn the components in the next half of the semester. For now, it is easier to just think about them as a function BUT created as a hardware. So, the explanation is a functional explanation.

Lecture #11

The Processor: Datapath

NUS National University of Singapore | School of Computing

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

Questions?

IMPORTANT: DO NOT SCAN THE QR CODE IN THE VIDEO RECORDINGS. THEY NO LONGER WORK

Ask at
https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

OR

Scan and ask your questions here!
(May be obscured in some slides)

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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)

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

2. MIPS Processor: Implementation

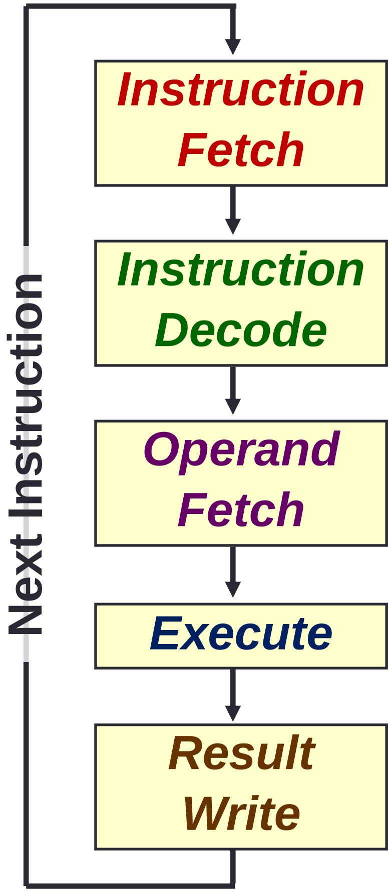
- Simplest possible implementation of a subset of the core MIPS ISA:
  - **Arithmetic and Logical operations**
    - `add`, `sub`, `and`, `or`, `addi`, ~~`andi`~~, ~~`ori`~~, `slt`
  - **Data transfer instructions**
    - `lw`, `sw`
  - **Branches**
    - `beq`, `bne`
- Shift instructions (`sll`, `srl`)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

Note
`andi` and `ori` is not supported in this current processor design because we always do "sign extension" on immediate value

Note
`sll` and `srl` can be done by multiplication (*which can be done by add with loop*). `j` can be done by `beq $zero, $zero, label` if we ignore the difference related to 256MB blocks

https://sets.netlify.app/module/66988ae3322ba68d110

3. Instruction Execution Cycle (Basic)



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

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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 $rd, $rs, $rt`	`lw $rt, ofst($rs)`	`beq $rs, $rt, labl`
Fetch **Decode**	***standard***	***standard***	***standard***
Operand Fetch	○ Read [**$rs**] as *opr1* ○ Read [**$rt**] as *opr2*	○ Read [**$rs**] as *opr1* ○ Use ***ofst*** as *opr2*	○ Read [**$rs**] as *opr1* ○ Read [**$rt**] 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 **$rd**	*Memory* data stored in **$rt**	if (*Taken*) **PC** = *Target*

- **opr** = operand
- **MemAddr** = address
- **ofst** = offset

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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 $rd, $rs, $rt`	`lw $rt, ofst($rs)`	`beq $rs, $rt, ofst`
Fetch	***standard***	***standard***	***standard***
Decode / **Operand Fetch**	○ Read [**$rs**] as *opr1* ○ Read [**$rt**] as *opr2*	○ Read [**$rs**] as *opr1* ○ Use ***ofst*** as *opr2*	○ Read [**$rs**] as *opr1* ○ Read [**$rt**] 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 **$rd**	*Memory* data stored in **$rt**	if (*Taken*) **PC** = *Target*

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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" ☺

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

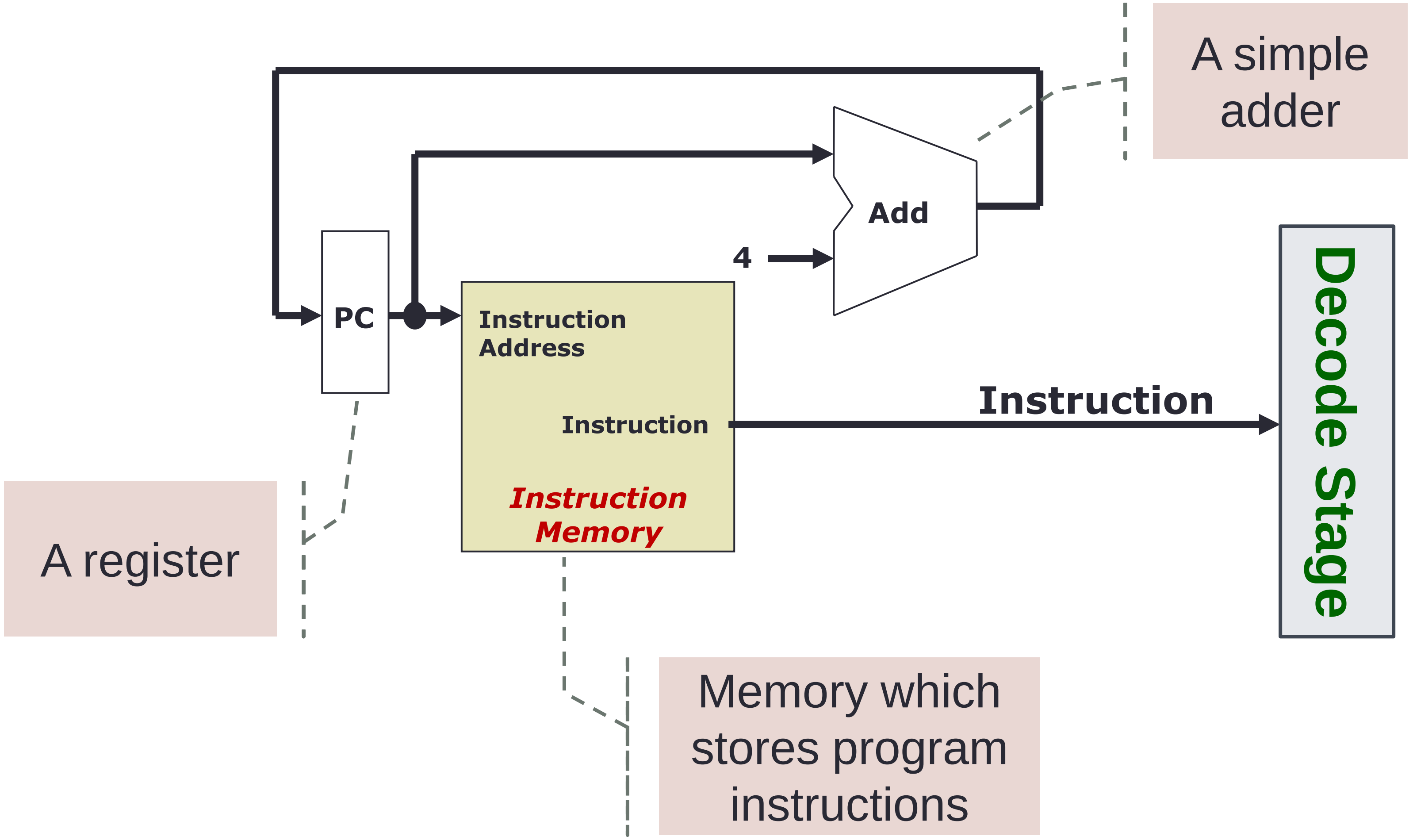
5.1 **Fetch Stage**: Requirements

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

- 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

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

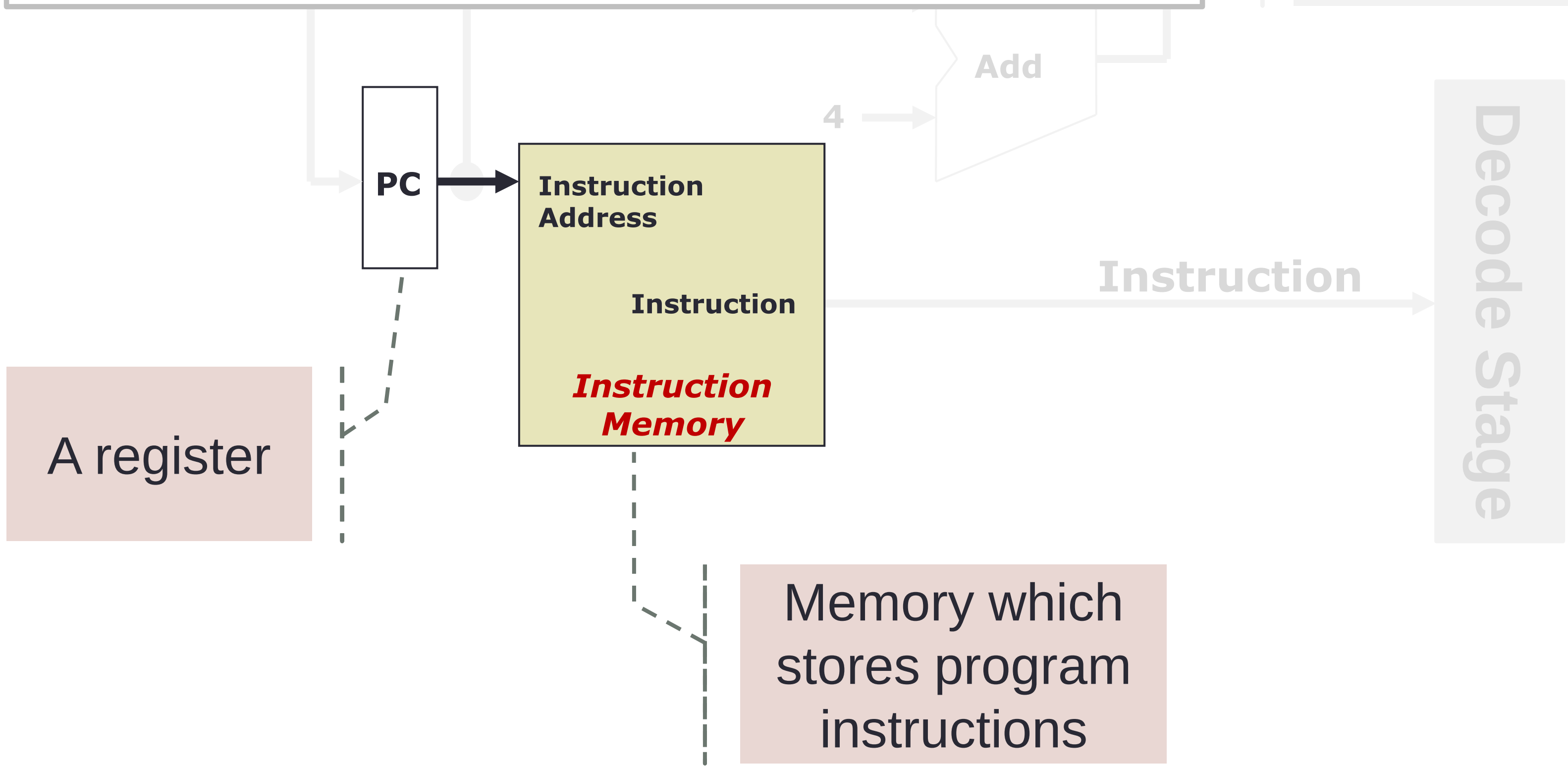
5.1 **Fetch Stage**: Block Diagram



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

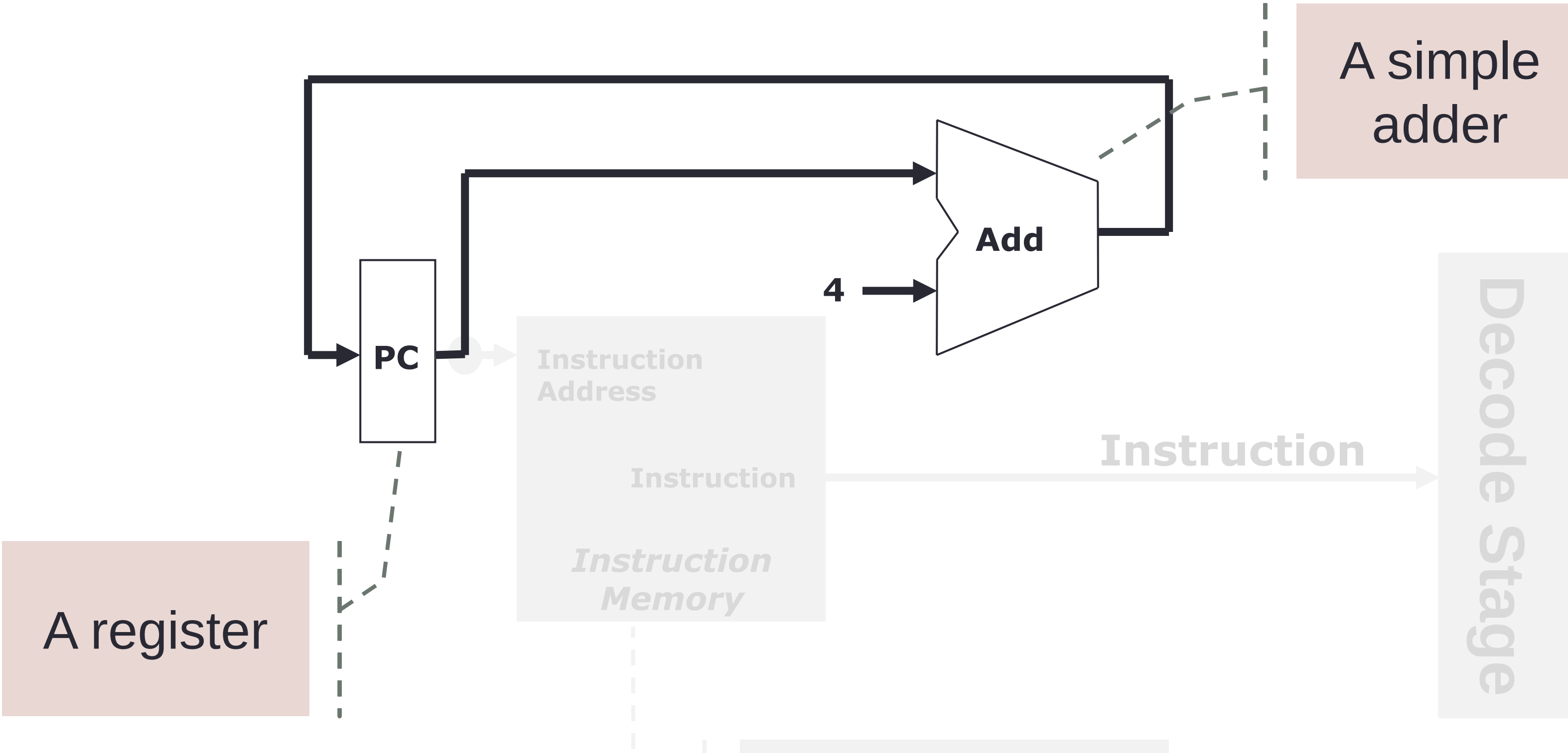
5.1 **Fetch Stage**: Block Diagram

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



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

5.1 **Fetch Stage**: Block Diagram

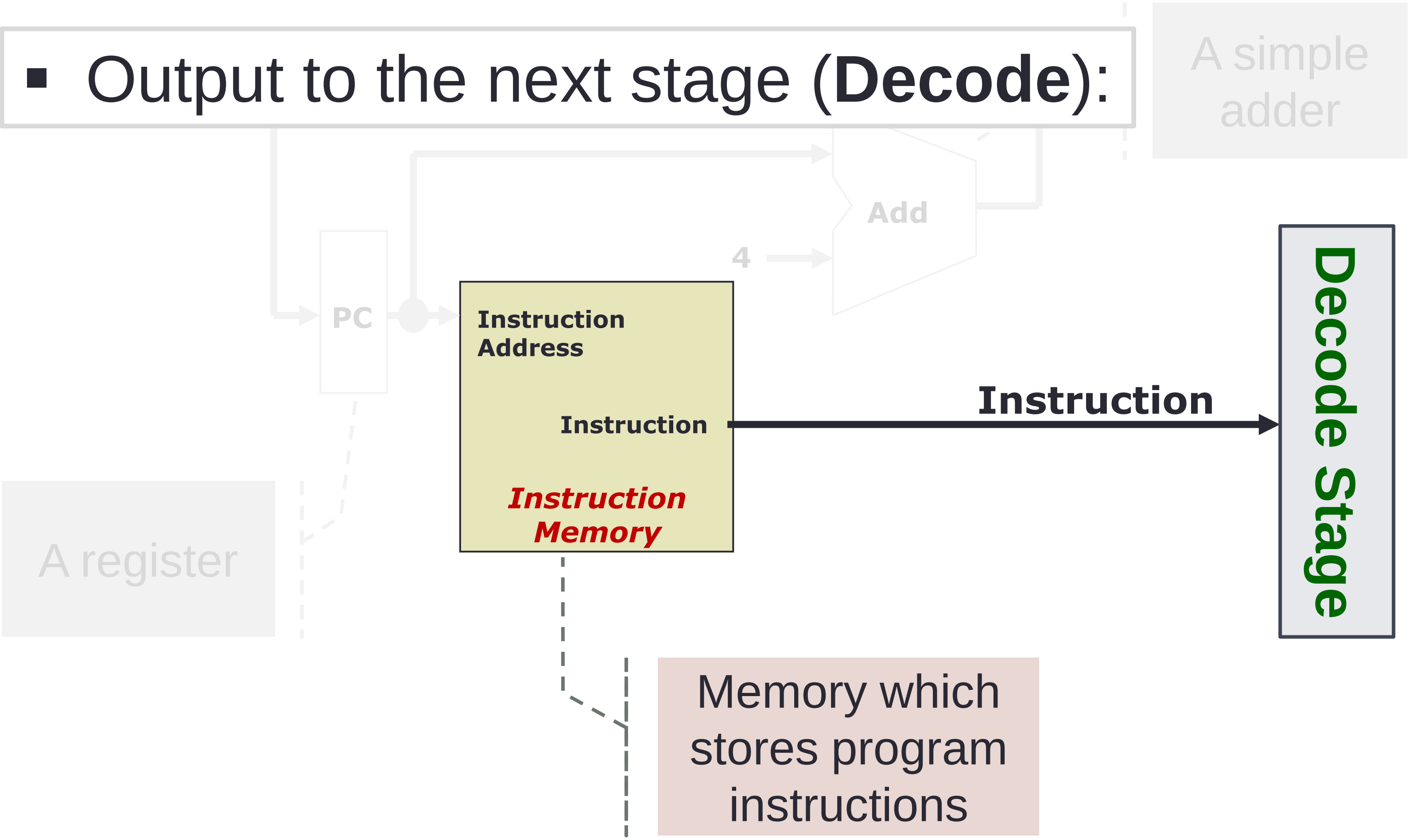


2. **Increment** the PC by 4 to get the address of the next instruction:

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

5.1 **Fetch Stage**: Block Diagram

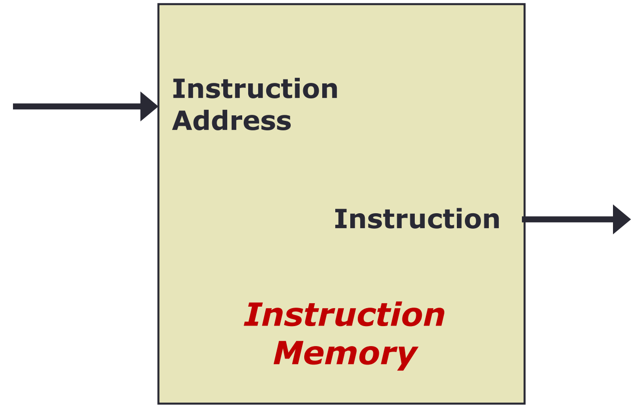
- Output to the next stage (**Decode**):




https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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



- 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
……	………..

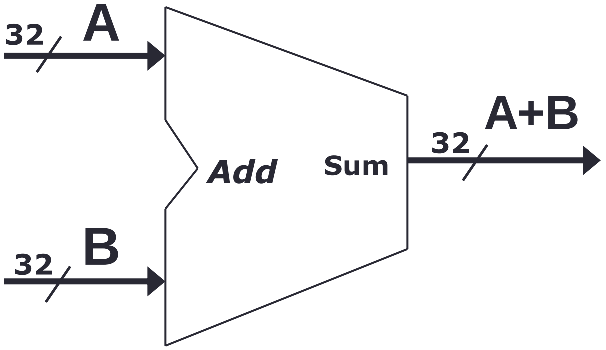
As a Function

```
function IM(addr) {
  return Mem[addr];
}
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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**



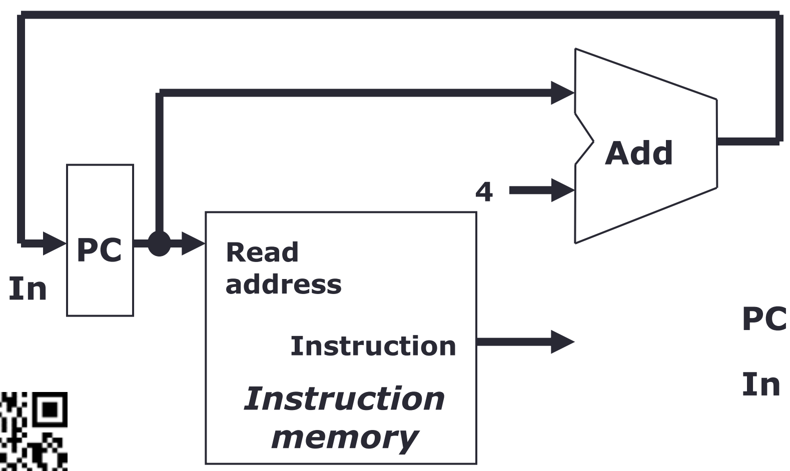
As a Function

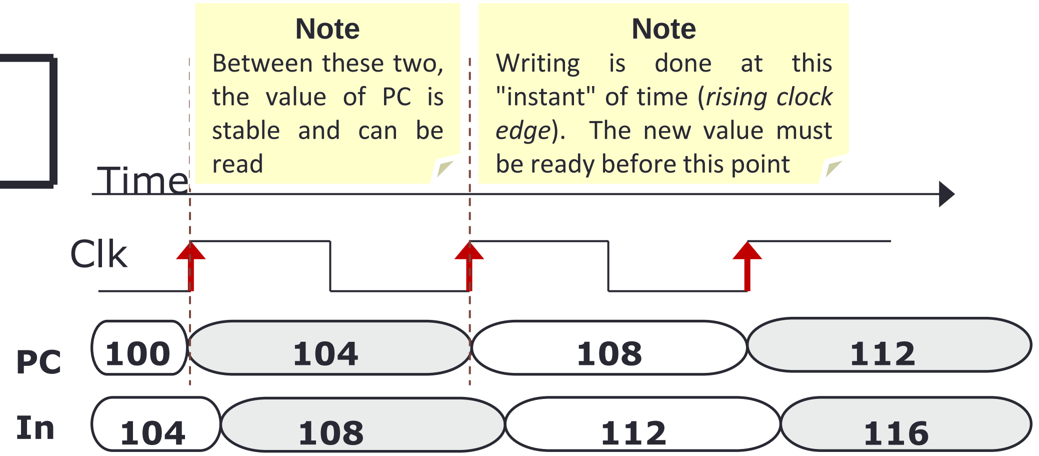
```
function add(A,B) {
  return A+B;
}
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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**





https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

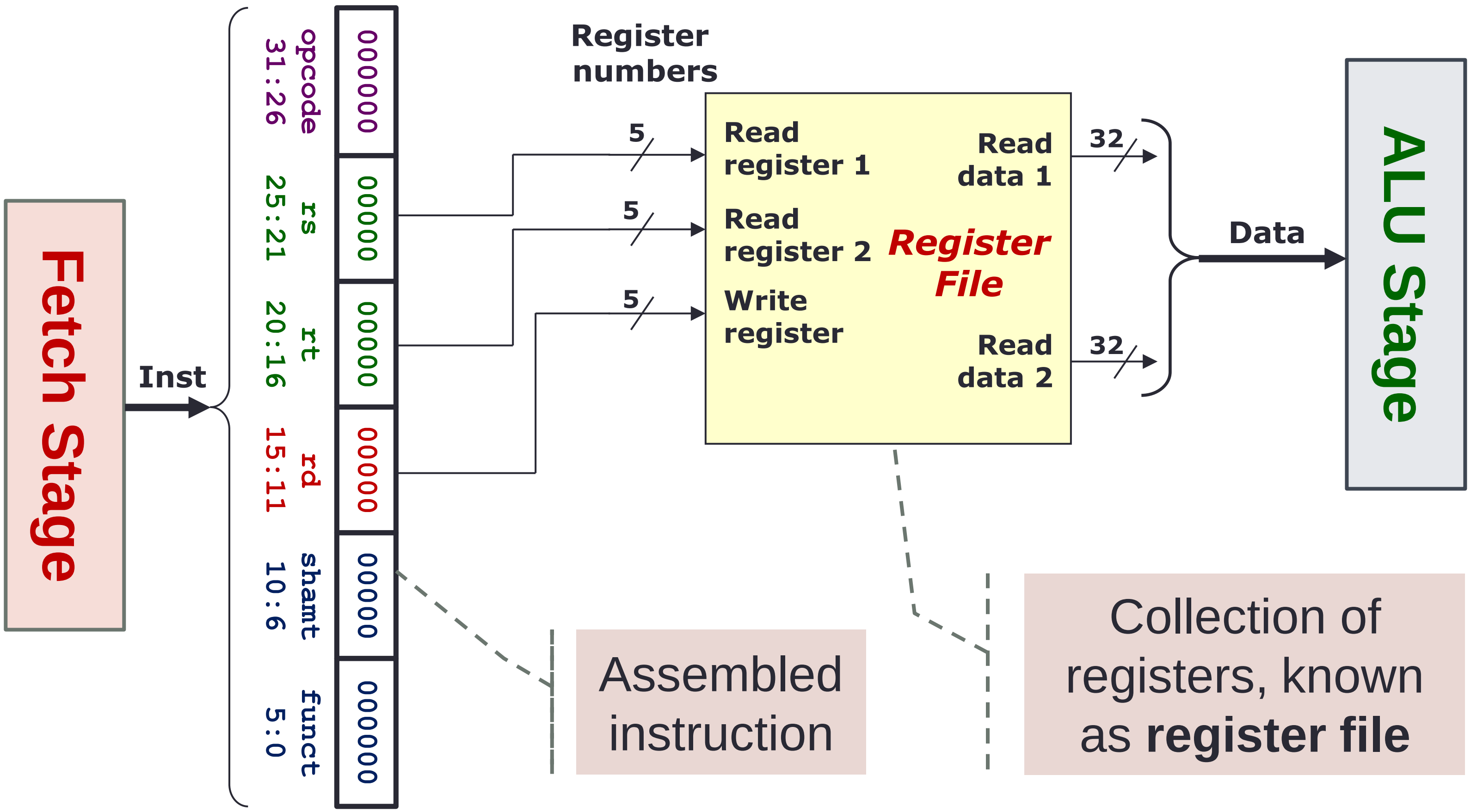
5.2 **Decode Stage**: Requirements

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

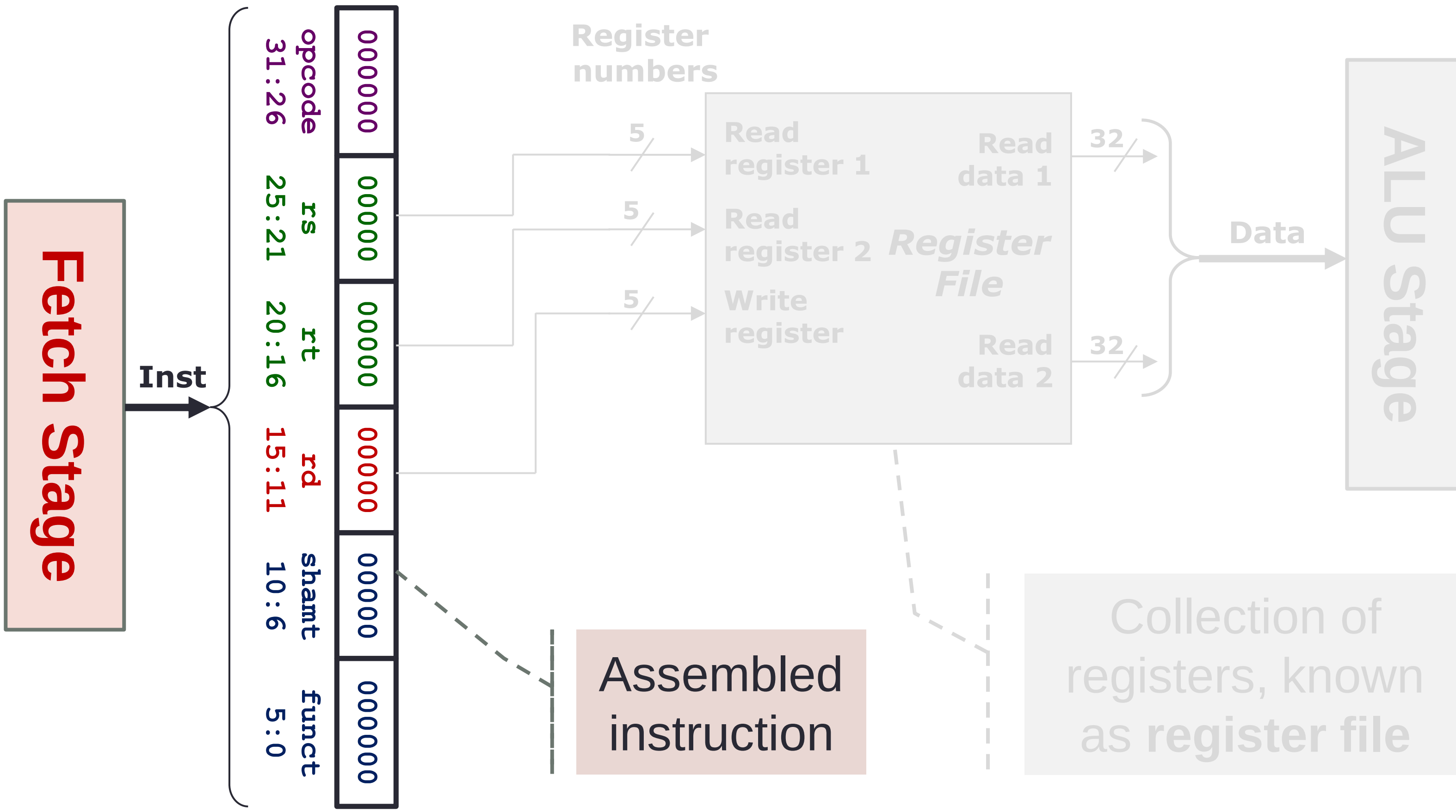
https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

5.2 **Decode Stage**: Block Diagram



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

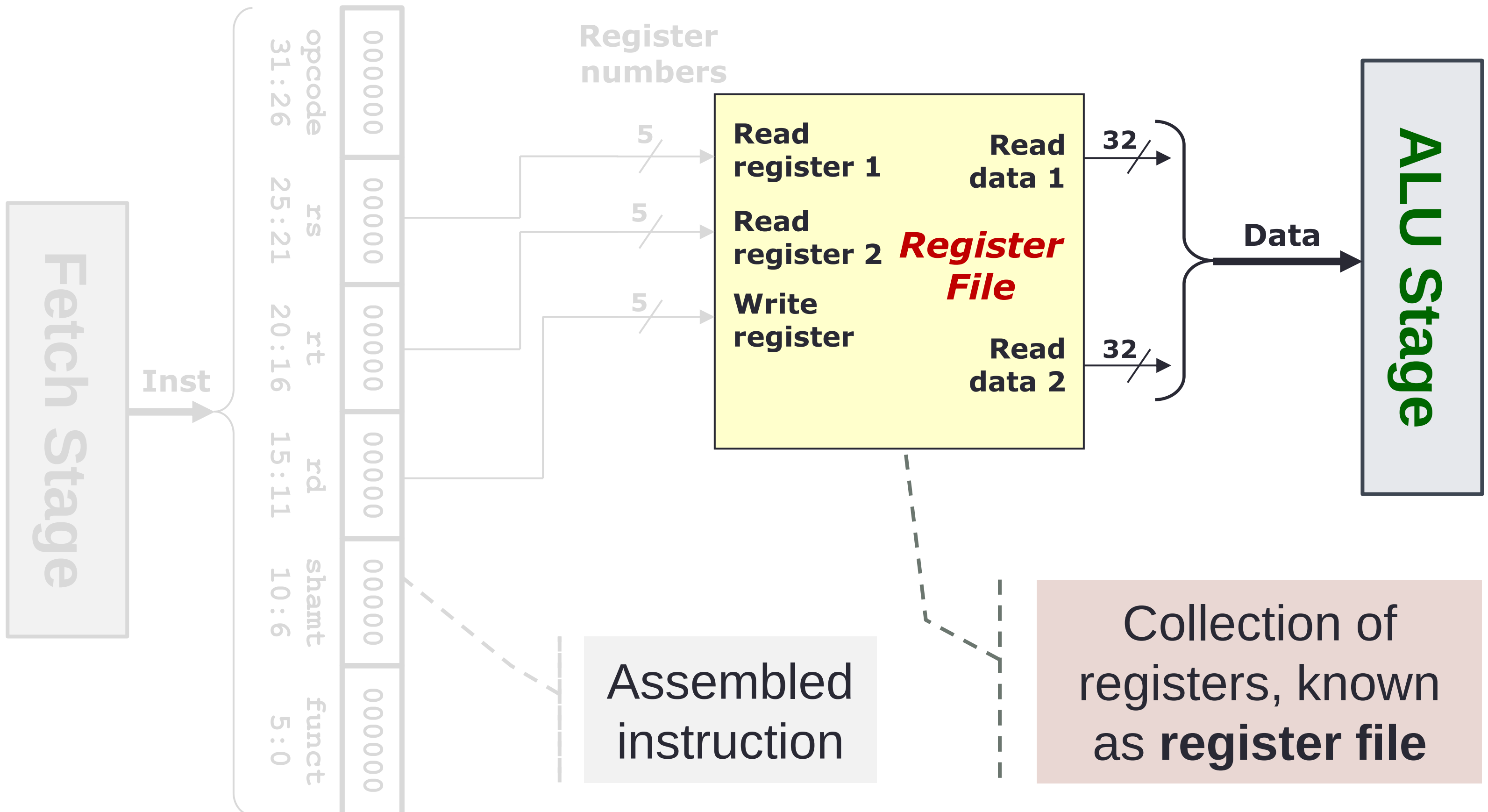
5.2 **Decode Stage**: Block Diagram



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

https://sets.netlify.app/module/66988ae3322ba08d1105fdd4

5.2 **Decode Stage**: Block Diagram

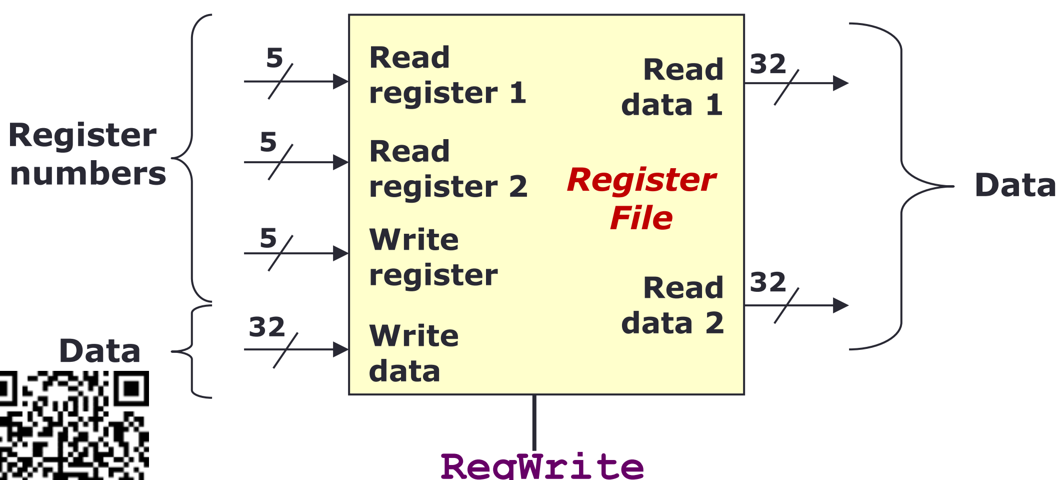


- Output to the next stage (**ALU**):
  - Operation and the necessary operands

https://sets.netlify.app/module/669886ae3322ba68d1165fdd4

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

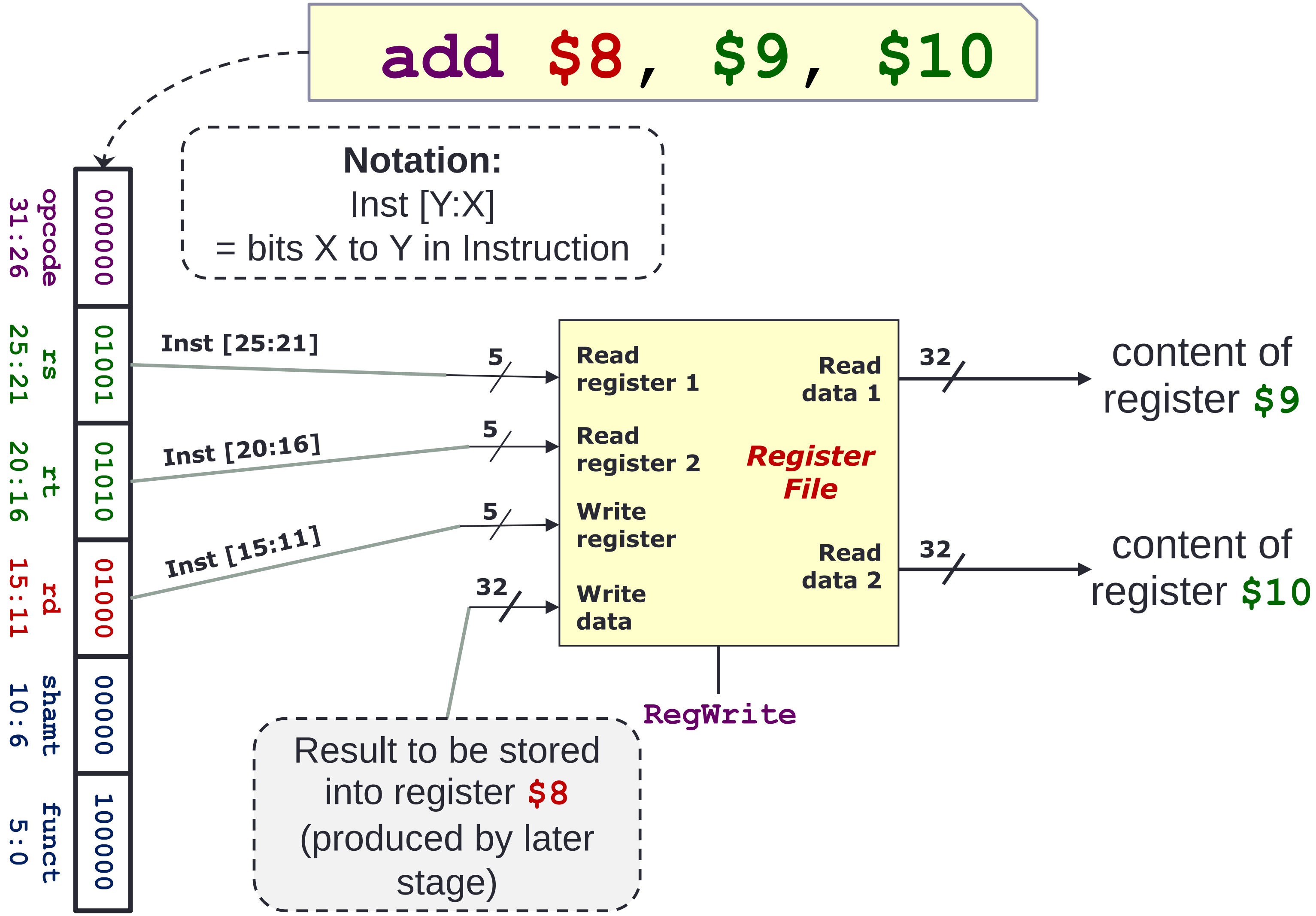


As a Function

```
// Decode Stage
function RegRead(RR1, RR2) {
  return [Reg[RR1], Reg[RR2]];
}
// Writeback Stage
function RegWrite(WR, WD, RegWrite) {
  if(RegWrite) {
    Reg[WR] = WD;
  }
}
```

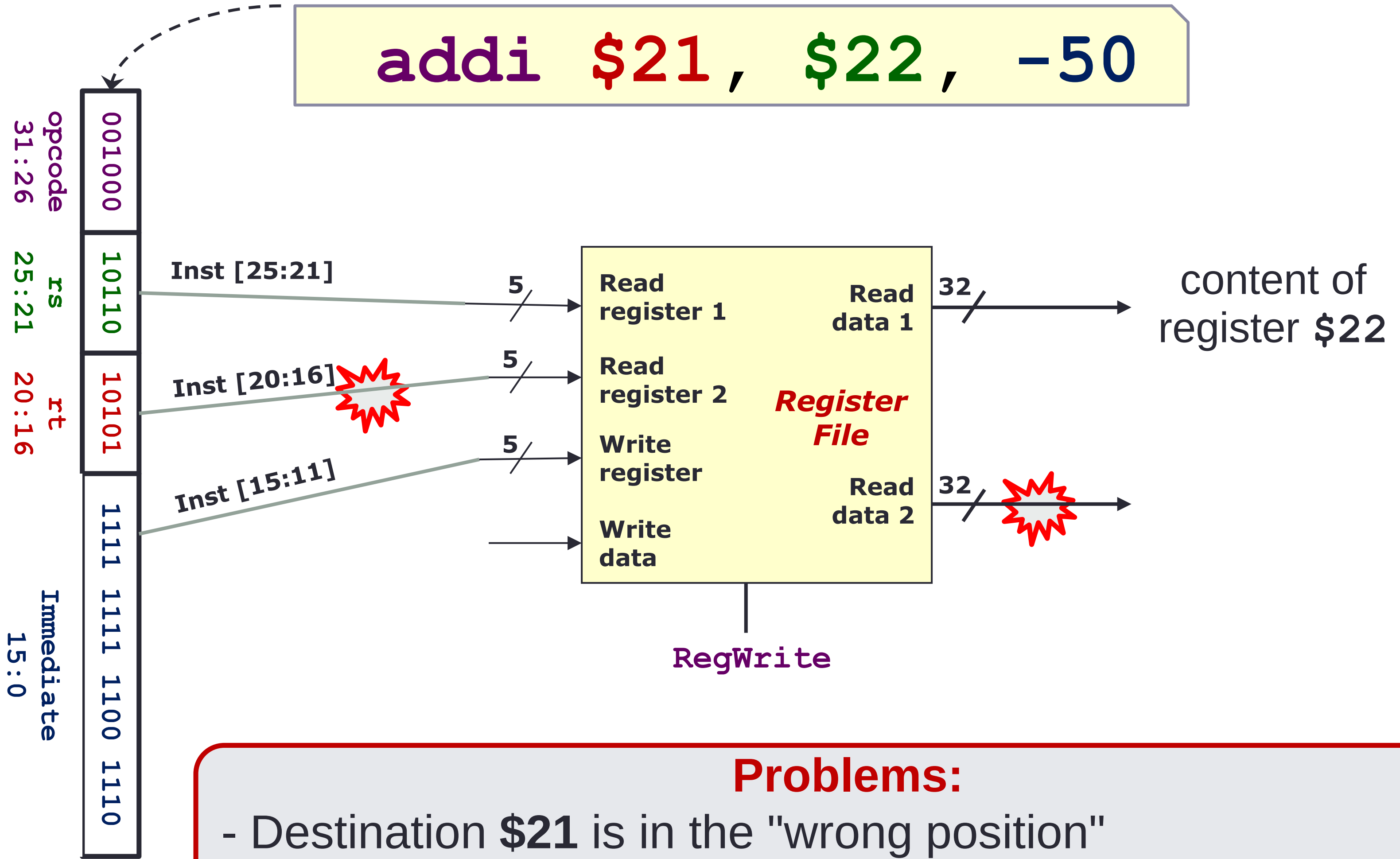
https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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

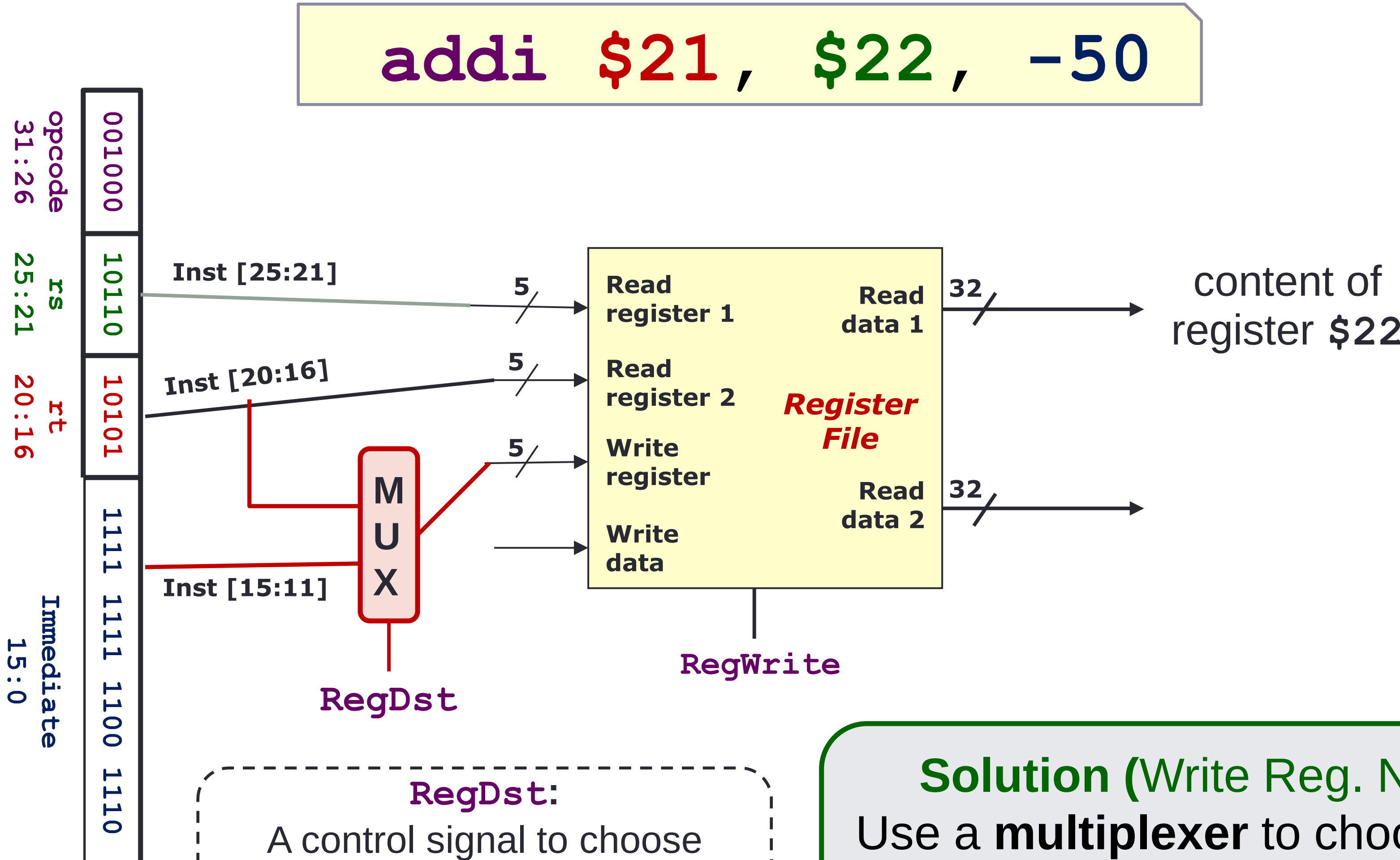


Problems:

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

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

5.2 Decode Stage: **Choice in Destination**



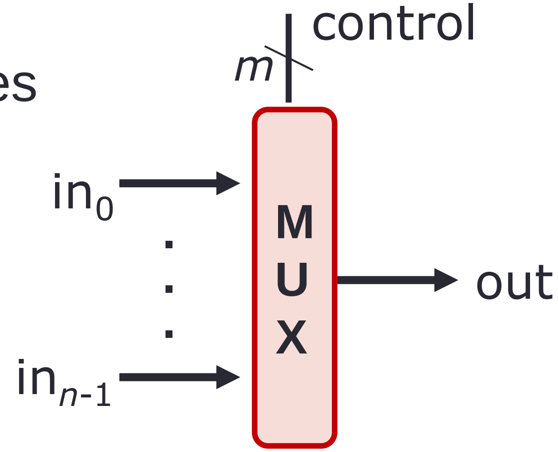
RegDst:
A control signal to choose either Inst[20:16] or Inst[15:11] as the write register number

Solution (Write Reg. No.**):**
Use a **multiplexer** to choose the correct write register number based on instruction type

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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=0 → select in_0 to out
Control=3 → select in_3 to out

As a Function

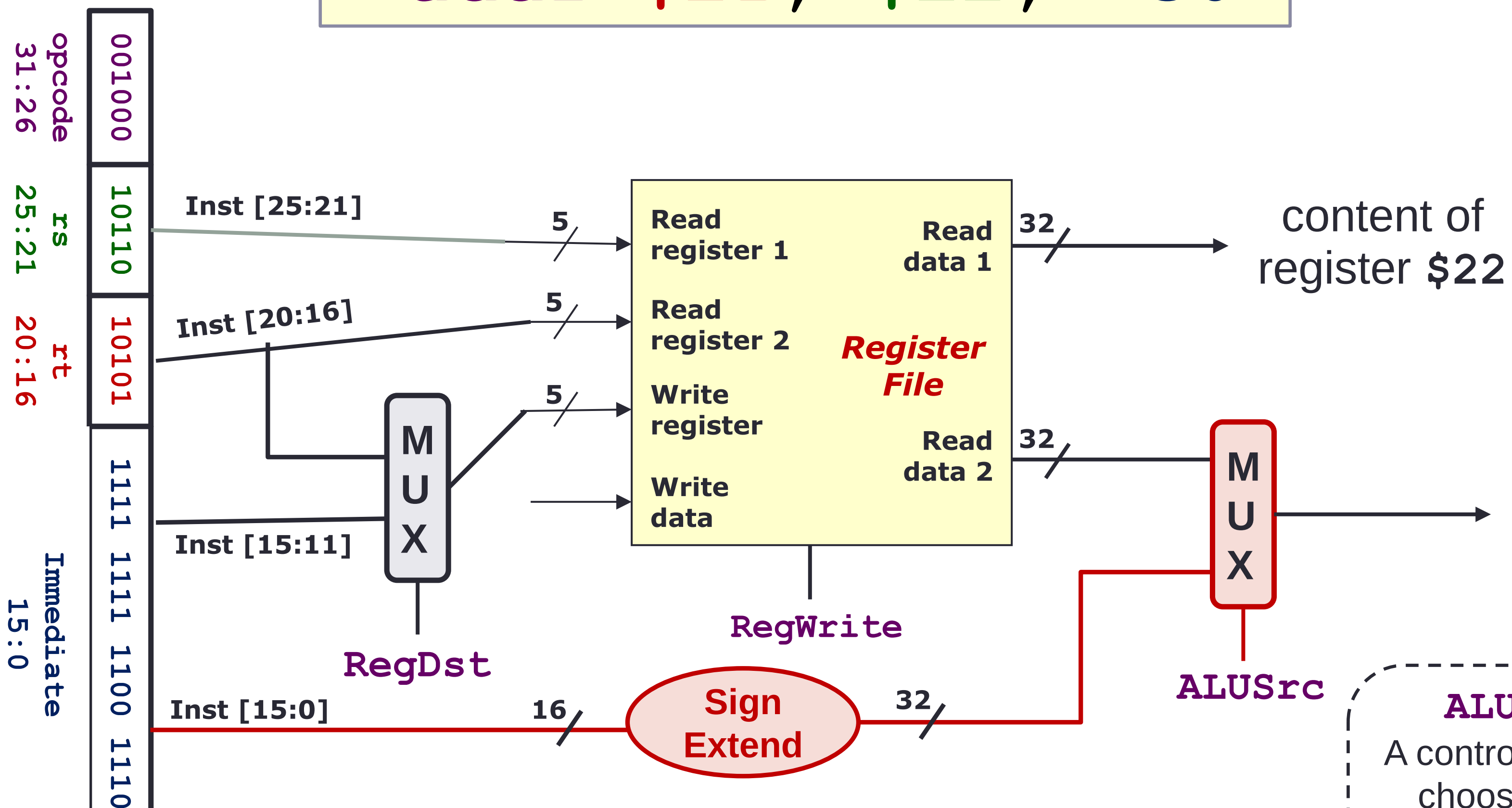
```
// 2 input + 1 control + 1 output
function Mux(in0, in1, ctrl) {
  if(!ctrl) {
    return in0;
  } else {
    return in1;
  }
}
```

Can be Combined to Form Larger MUX

```
function Mux2(in0,in1,in2,in3,ctrl0,ctrl1) {
  return Mux(Mux(in0,in1,ctrl0),
             Mux(in2,in3,crtl0),
             ctrl1);
}
```

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

`addi $21, $22, -50`

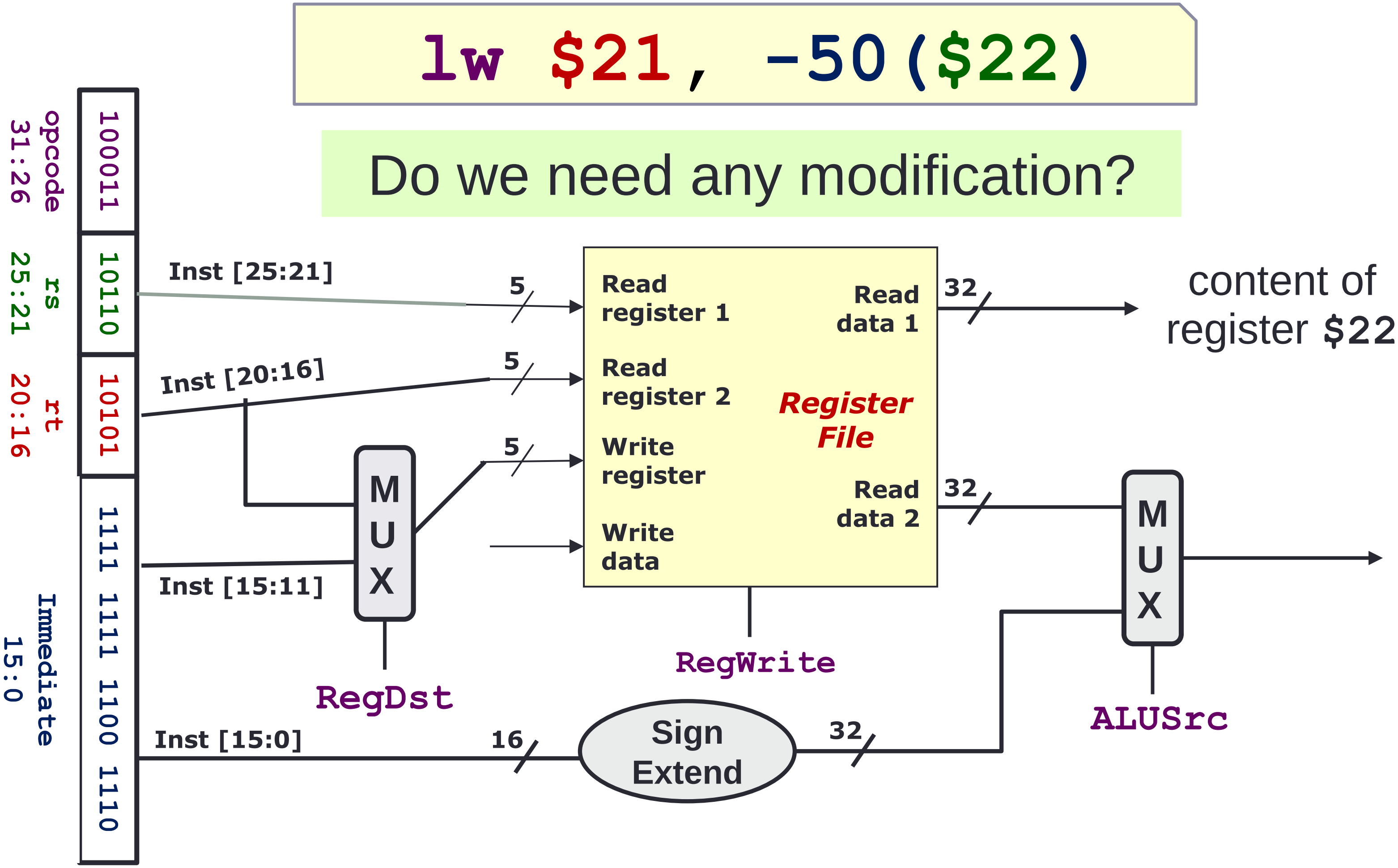


Solution (Rd. Data 2**):**
Use a **multiplexer** to choose the correct operand 2.
Sign extend the 16-bit immediate value to 32-bit

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

https://s

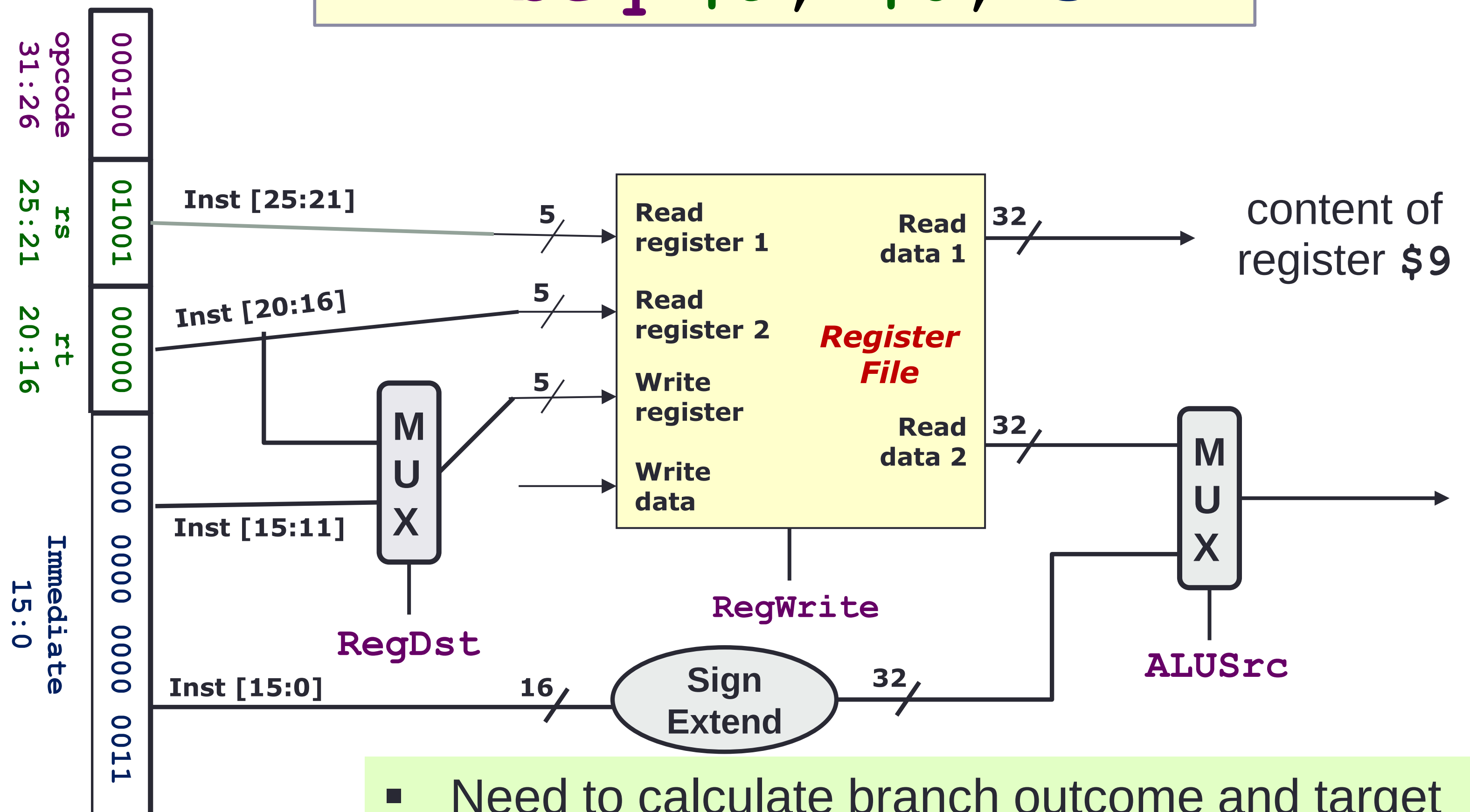
5.2 Decode Stage: **Load Word Instruction**



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

5.2 Decode Stage: **Branch Instruction**

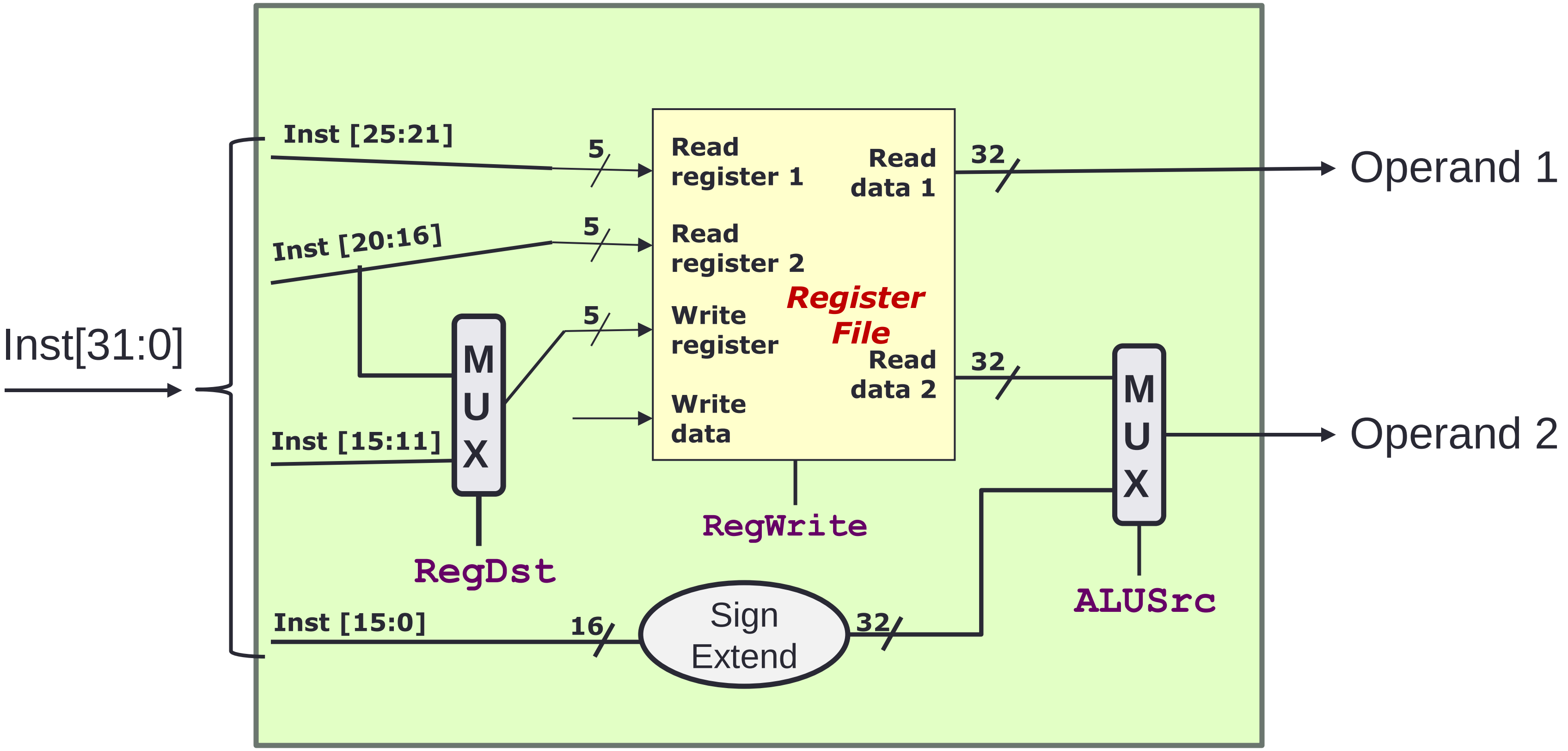
`beq $9, $0, 3`



- Need to calculate branch outcome and target at the same time!
- We will tackle this problem at the ALU stage

https://sets.netlify.ap

5.2 **Decode Stage**: Summary



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

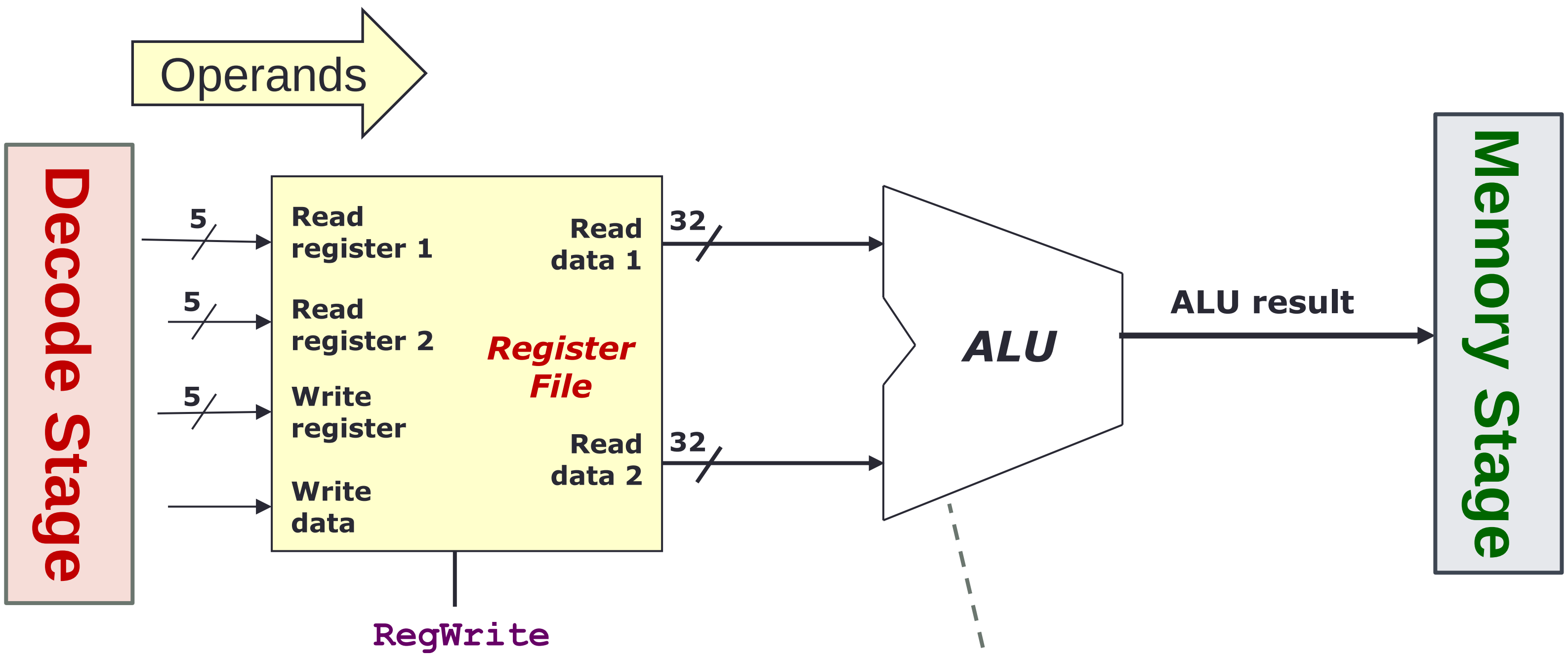
5.3 **ALU Stage**: Requirements

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

- 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

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

5.3 **ALU Stage**: Block Diagram



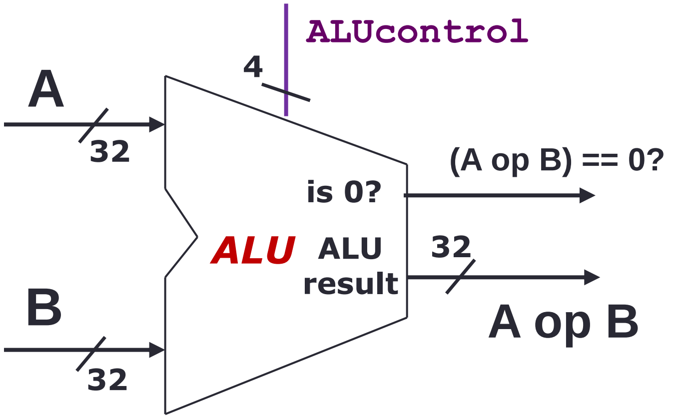
Logic to perform arithmetic and logical operations

ALU as a Function (see next slide for cases)

```
function ALU(A,B,ALUcontrol) {
  case 0000:
    return [A&B, A&B==0];
  case 0001:
    return [A|B, A|B==0];
  :
}
```

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

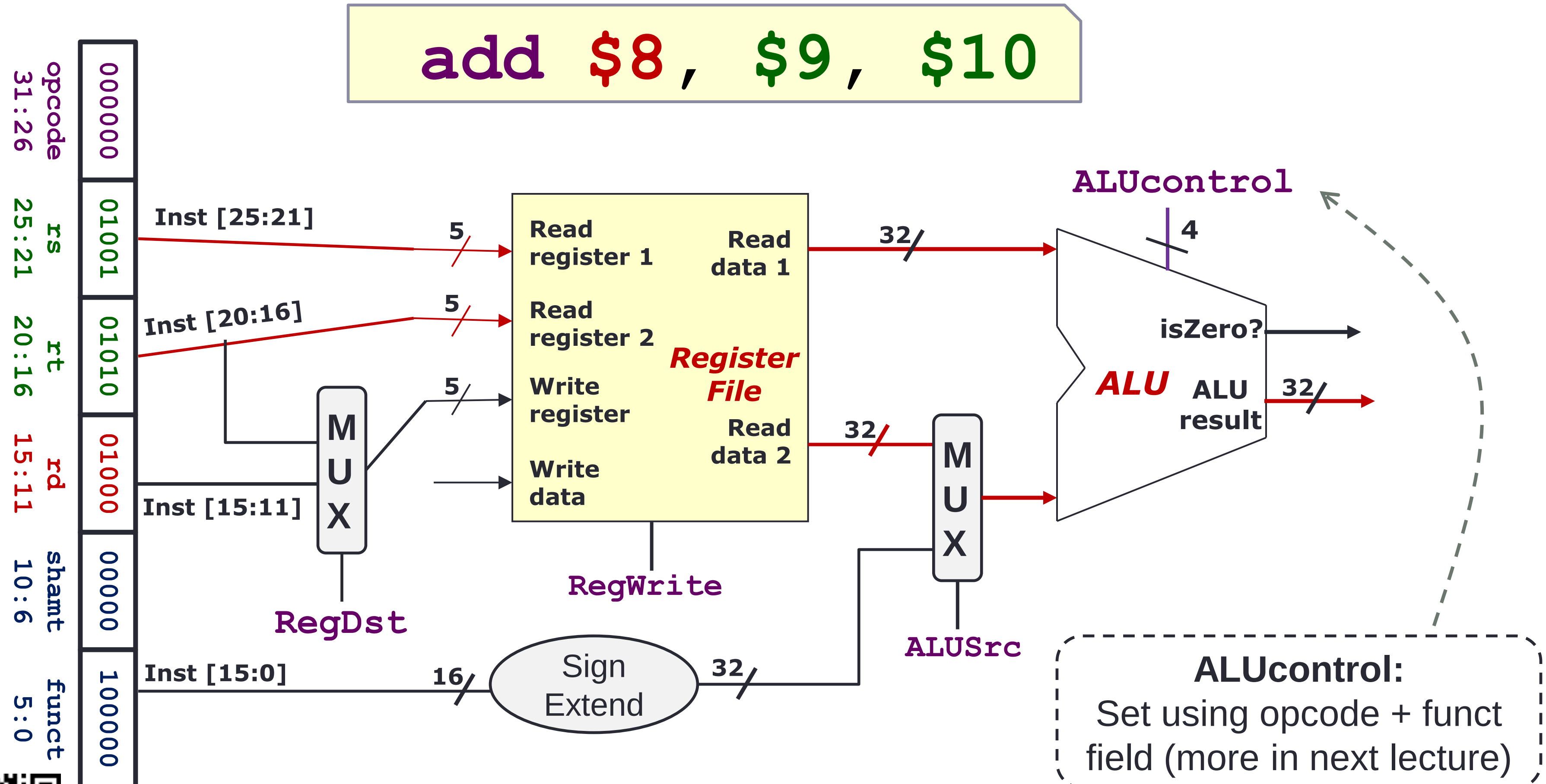


ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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

- We can handle non-branch instructions easily:



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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)

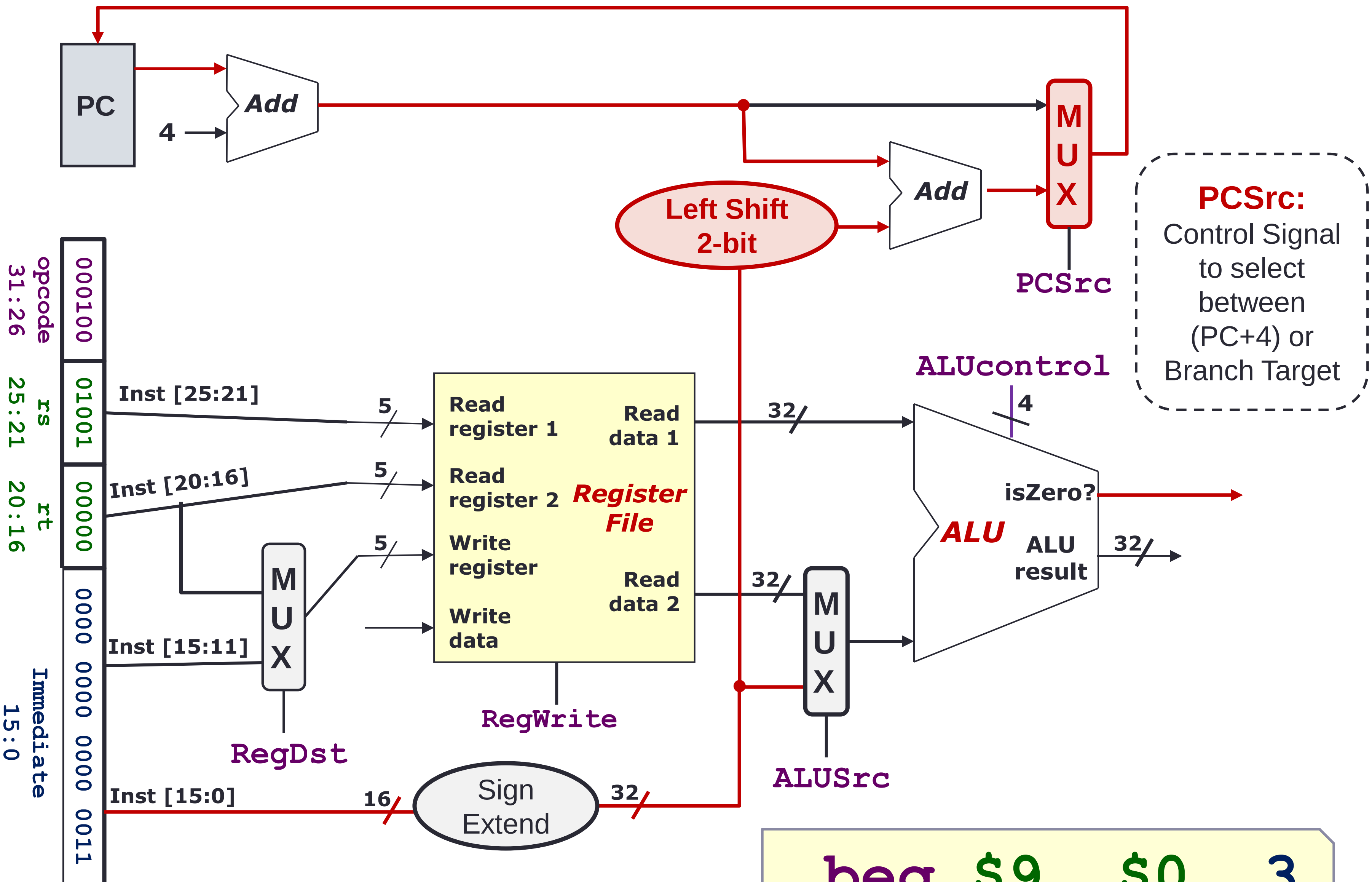
Note

Two things need to happen to actually take the branch

1. The instruction is a branch instruction
2. The condition of the branch is true

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

Complete ALU Stage



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

beq $9, $0, 3

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

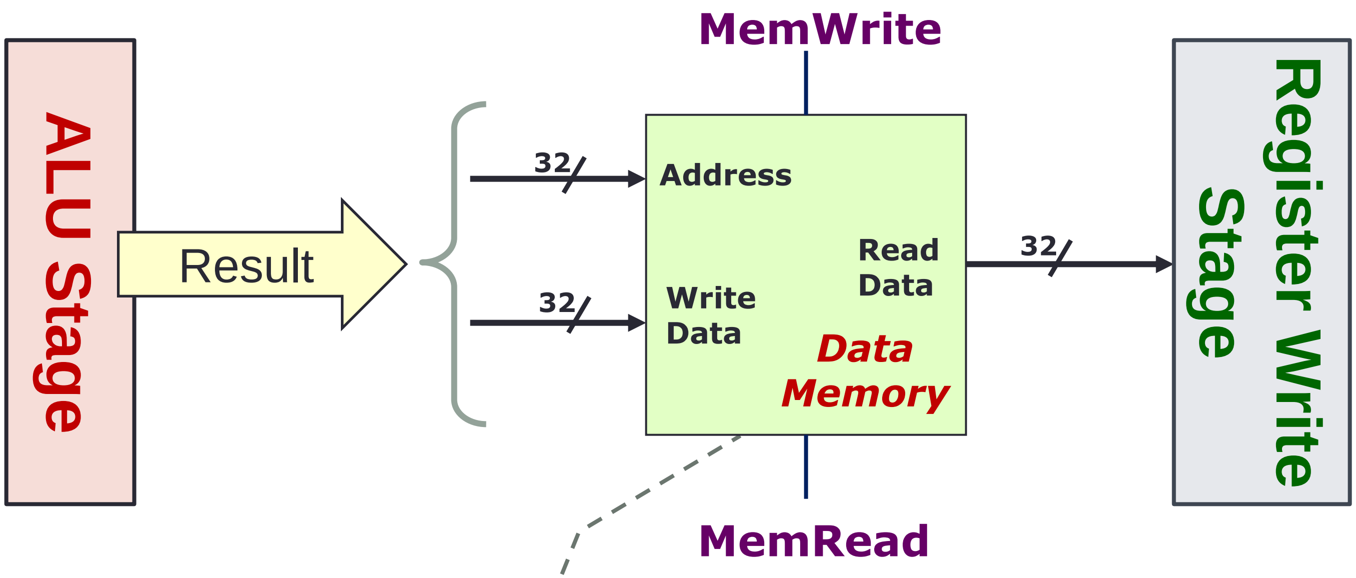
5.4 **Memory Stage**: Requirement

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)

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

5.4 **Memory Stage**: Block Diagram

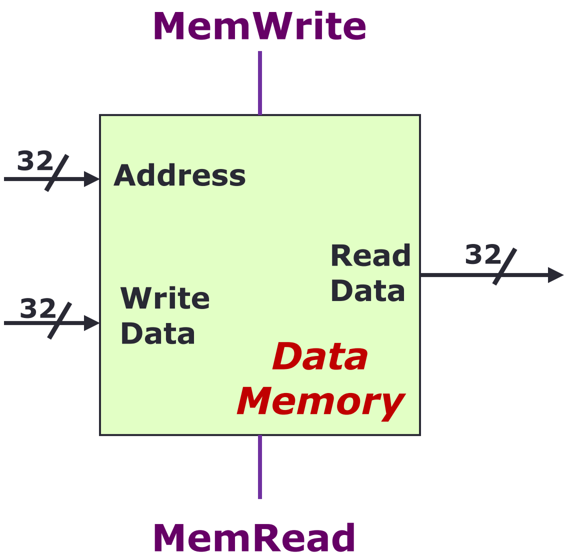


Memory which stores data values

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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



As a Function

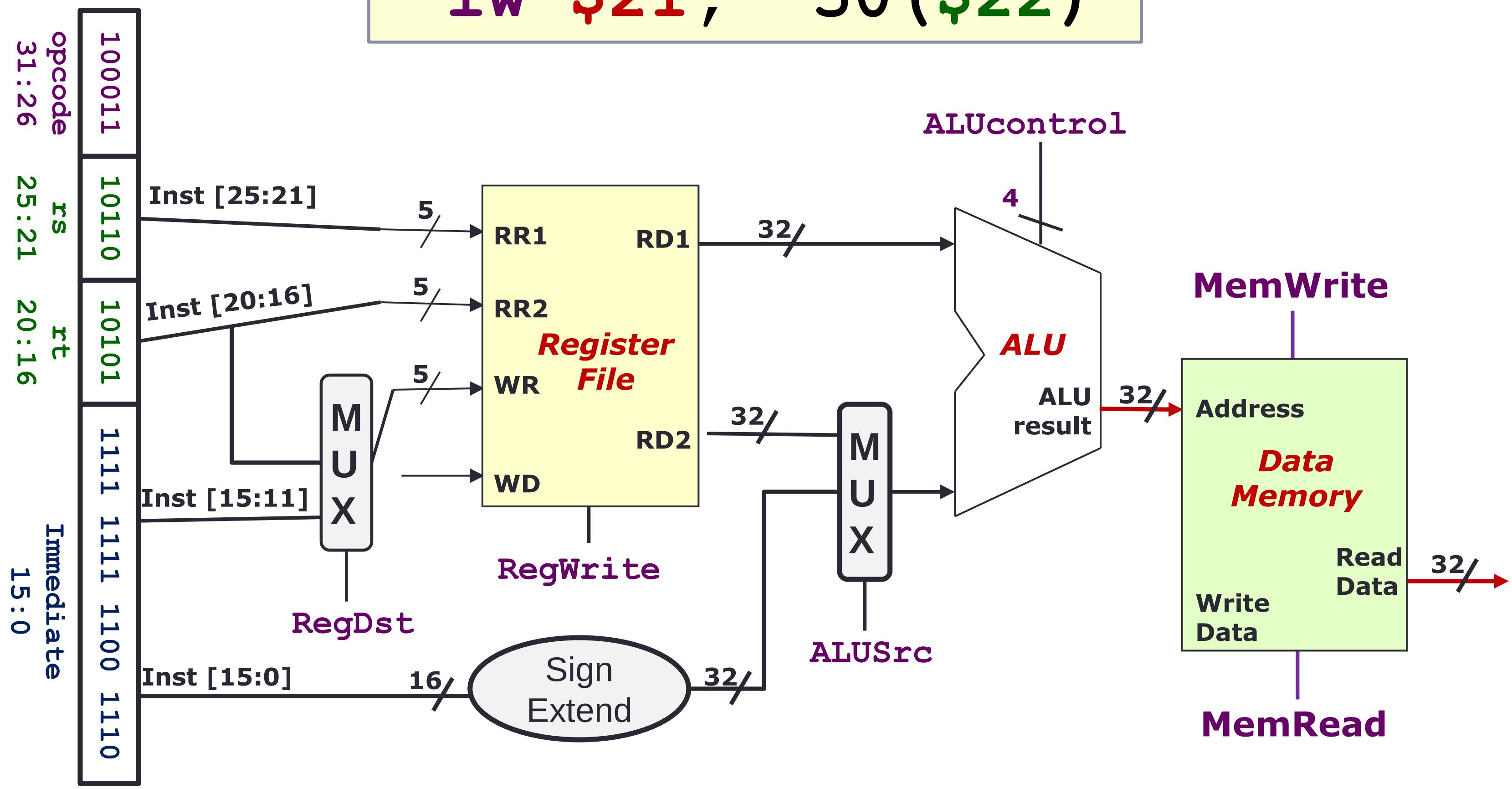
```
function DataMem(addr,WD,MW,MR) {
  if(MW) {
    Mem[addr] = WD;
  } else if(MR) {
    return Mem[addr];
  }
}
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

5.4 Memory Stage: **Load Instruction**

- Only relevant parts of Decode and ALU Stages are shown

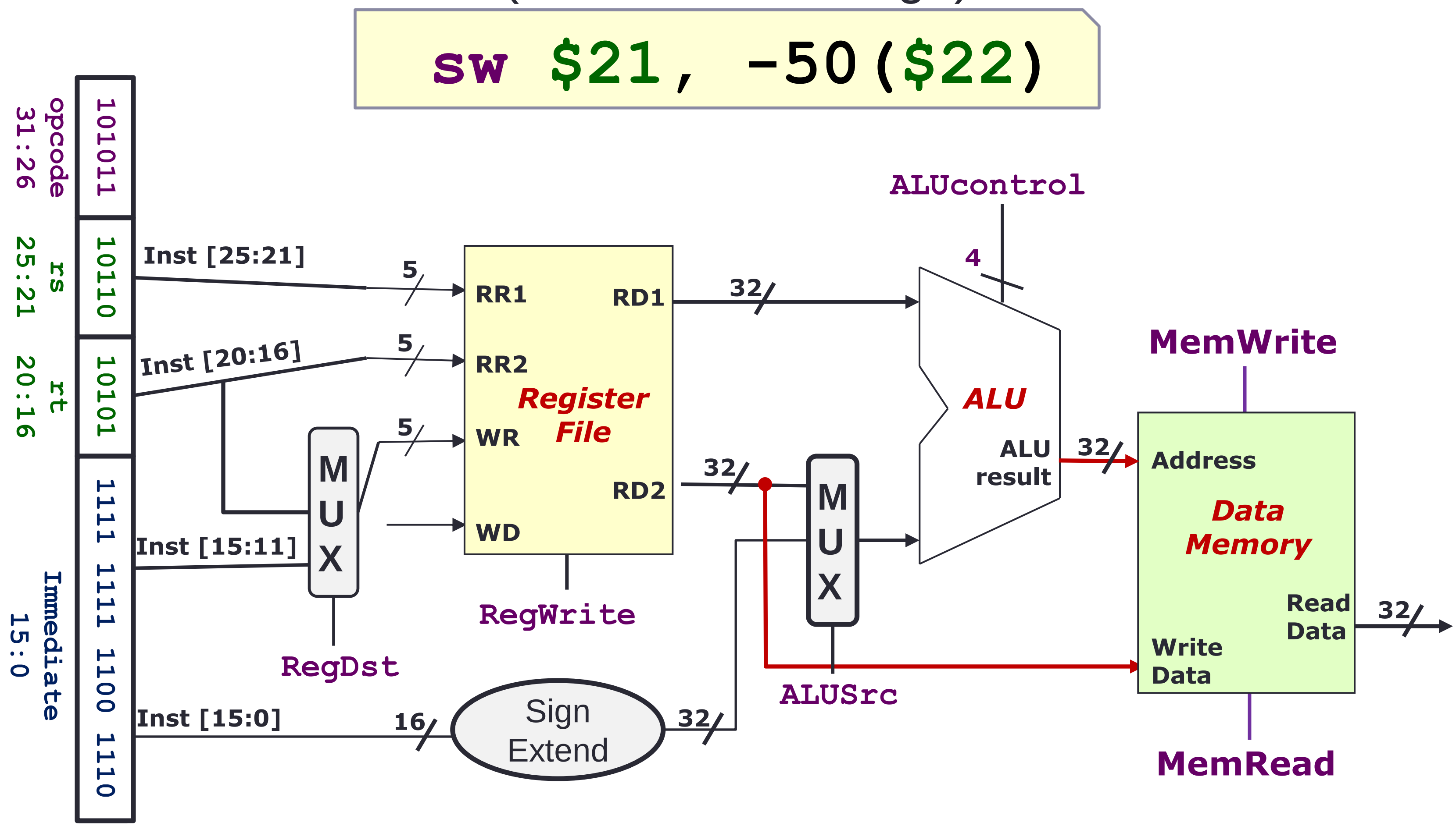
```
lw $21, -50($22)
```



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

5.4 Memory Stage: **Store Instruction**

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

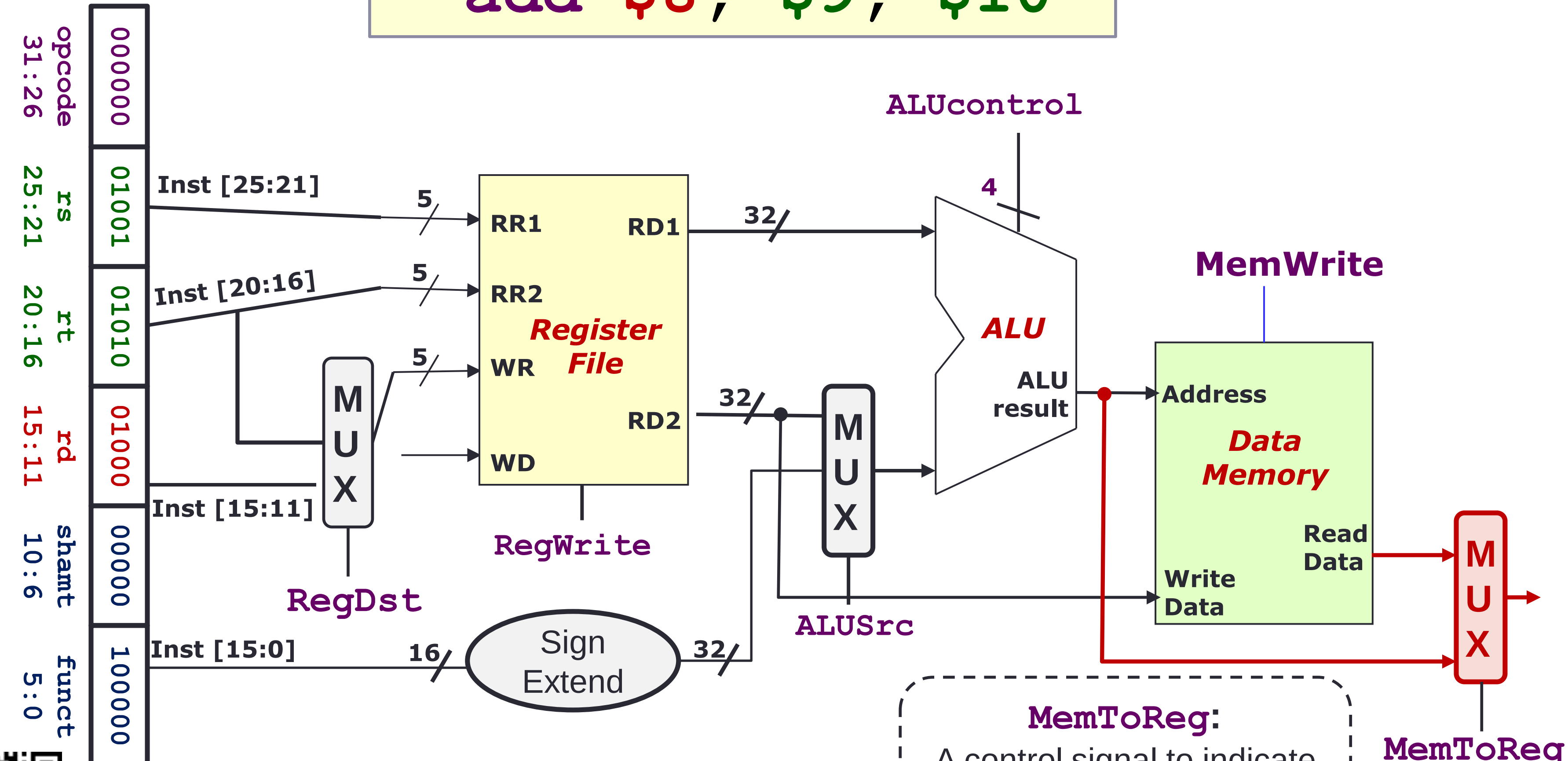


https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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

https://sets.netlify.app/module/66988ae3322ba68d1103fd...

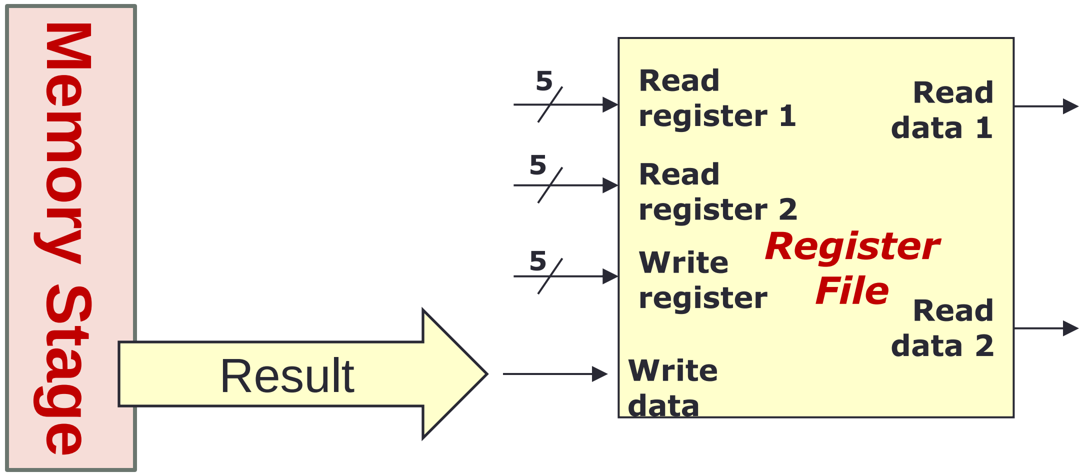
5.5 **Register Write Stage**: Requirement

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

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

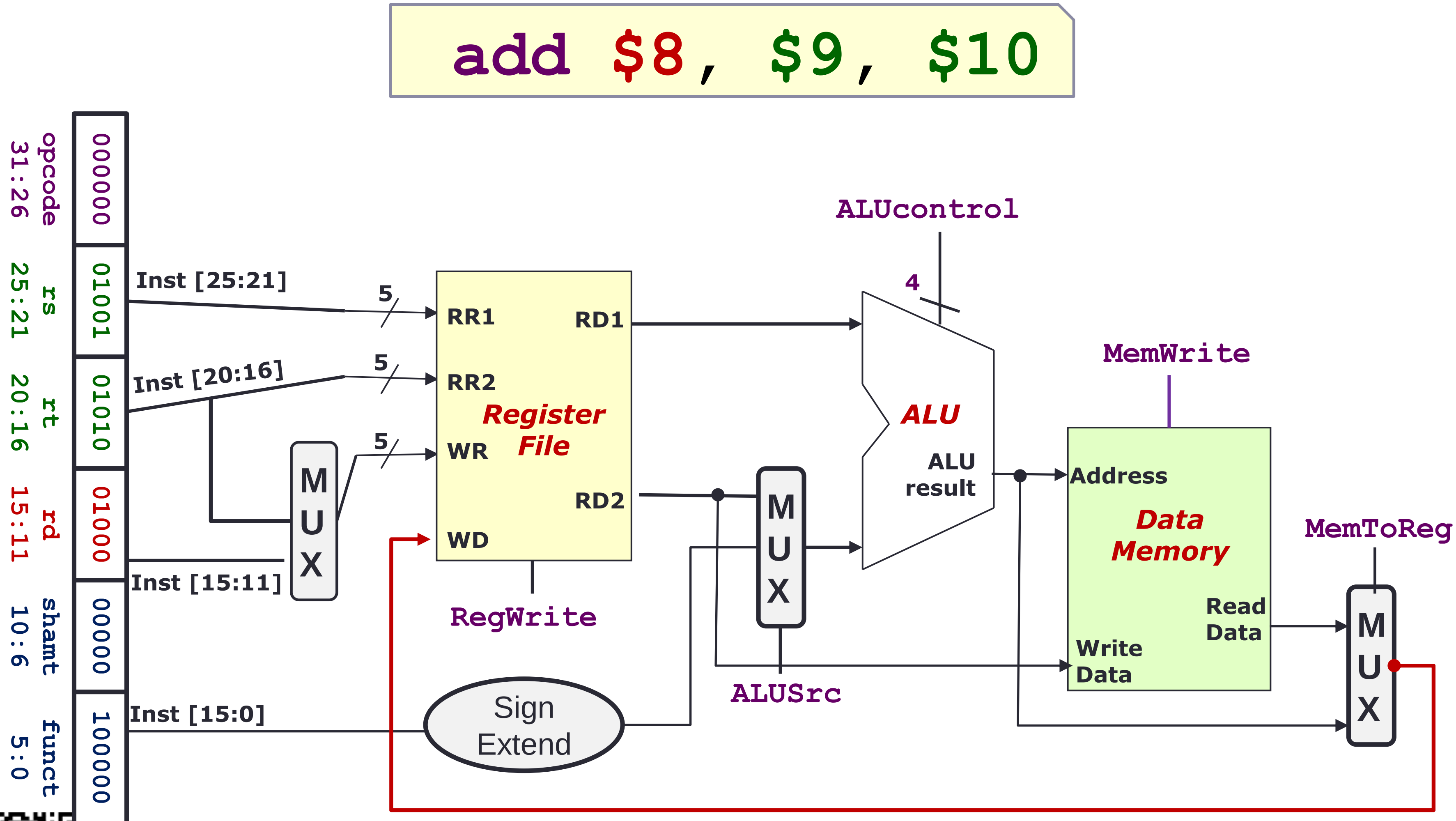
5.5 **Register Write Stage**: Block Diagram



- 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

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

5.5 Register Write Stage: **Routing**



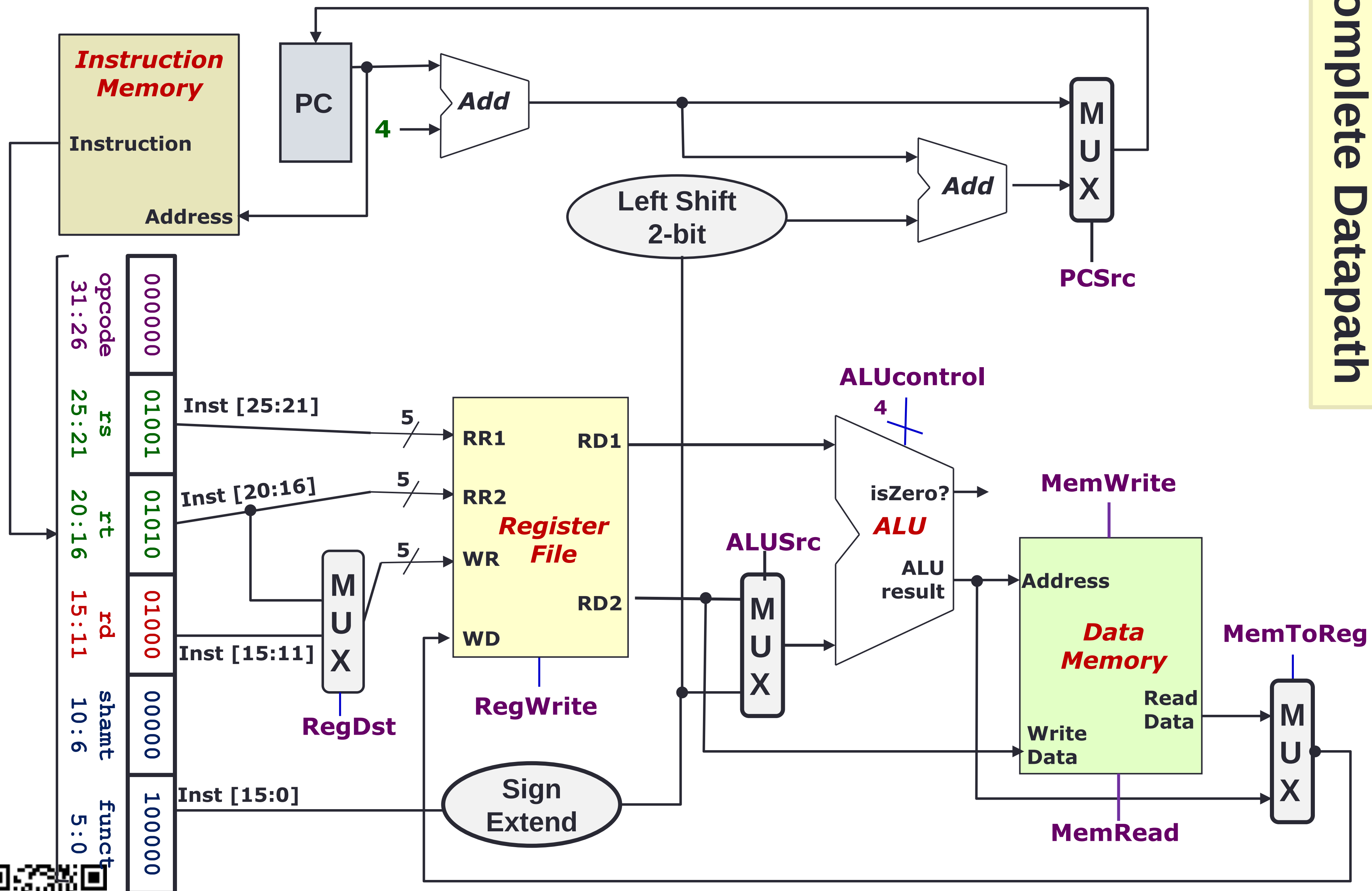
https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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**

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

Complete Datapath



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

Stages Summaries

- Since the components are already described as functions, we will also describe the stages as functions
- You should look back at the components as a function to understand how we can call them
- This is merely an alternative explanation, you should depend on the original explanation rather than this

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

Stages Summaries

```
function FETCH() {
  inst = IM(PC);   // read instruction at address from PC
  PC = Add(PC, 4); // update PC to PC+4
  return inst;
}
function DECODE(inst) {
  // 1. Read Register
  RR1 = inst[21:25];           // $rs
  RR2 = inst[16:20];           // $rt
  RD1,RD2 = RegRead(RR1, RR2); // read both registers
  // 2. Store WR for later use
  WR  = Mux(inst[16:20],       // $rt
            inst[11:15],       // $rd
            RegDst);           // control signal
  // 3. Choose output
  IMM = SignExtend(inst[0:15]);         // immediate value
  return [RD1, Mux(RD2, IMM, ALUSrc)]; // choose imm or rd2
}
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

Stages Summaries

```
function ALU(A,B,ALUcontrol) {
  case 0000: return [A&B    , A&B    ==0]; // AND
  case 0001: return [A|B    , A|B    ==0]; // OR
  case 0010: return [A+B    , A+B    ==0]; // ADD
  case 0110: return [A-B    , A-B    ==0]; // SUB
  case 0111: return [A<B    , A<B    ==0]; // SLT
  case 1100: return [~(A|B), ~(A|B)==0]; // NOR
}

function MEM(alu_res,data) {
  mem_res = DataMem(alu_res, data, MemWrite, MemRead);
  return Mux(alu_res, mem_res, MemToReg);
}

function WRITEBACK(data) {
  RegWrite(data, WR, RegWrite); // WR is global var from DECODE
}
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

Stages Summaries

```
function DATAPATH() { // one cycle
  // Variables:
  inst, RR1, RR2, RD1, RD2, WR, IMM, A, B, alu_res, mem_res, data;
  // Controls are assumed to be already set correctly
  inst    = FETCH();
  A,B     = DECODE(inst);
  alu_res = ALU(A, B, ALUControl);
  data    = MEM(alu_res, RD2, MemWrite, MemRead);
            WRITEBACK(data, WR, RegWrite);
}

// Then we can do this in a loop
while(true) {
  DATAPATH(); // This is your processor
}
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

7. Brief Recap (1/4)

- Lecture #7, Slide 5 (4 in video)

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)

```
0001001000000000 0000000000000011
1000111000101000 0000000000000100
0000001000001000 0100000000010100
1010111000101000 0000000000000000
```

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

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

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

7. Brief Recap (2/4)

- Lecture #7, Slide 5 (4 in video)

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)

```
0001001000000000 0000000000000011
1000111000101000 0000000000000100
0000001000001000 0100000000010100
1010111000101000 0000000000000000
```

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:
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

7. Brief Recap (3/4)

- Lecture #7, Slide 5 (4 in video)

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)

```
0001001000000000 0000000000000011
1000111000101000 0000000000000100
0000001000001000 0100000000010100
1010111000101000 0000000000000000
```

- 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
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

7. Brief Recap (4/4)

- Lecture #7, Slide 5 (4 in video)

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)

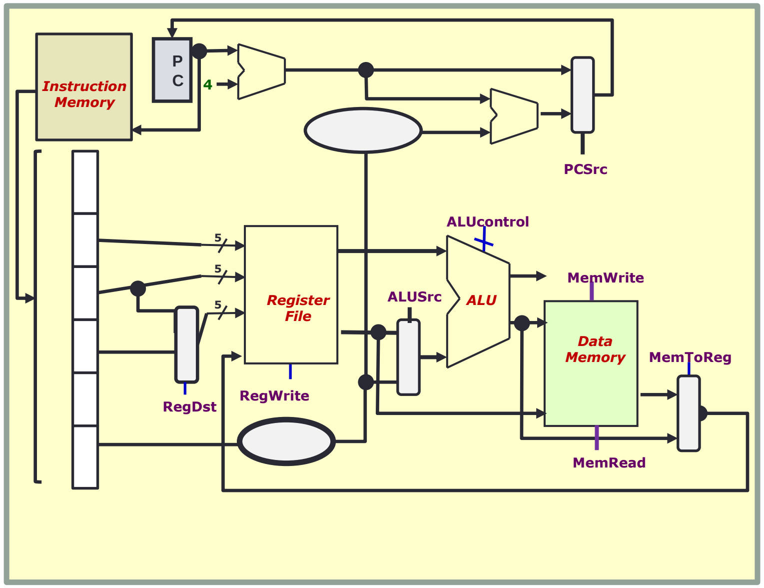
```
swap:
       muli $2, $5, 4
       add $2, $4, $2
       lw $15, 0($2)
       lw $16, 4($2)
       sw $16, 0($2)
       sw $15, 4($2)
       jr $31
```

Assembler

Binary machine language program (for MIPS)

```
0001001000000000 0000000000000011
1000111000101000 0000000000000100
0000001000001000 0100000000010100
1010111000101000 0000000000000000
```

Processor executes the machine code (i.e., **binaries**)



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

8. From C to Execution

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

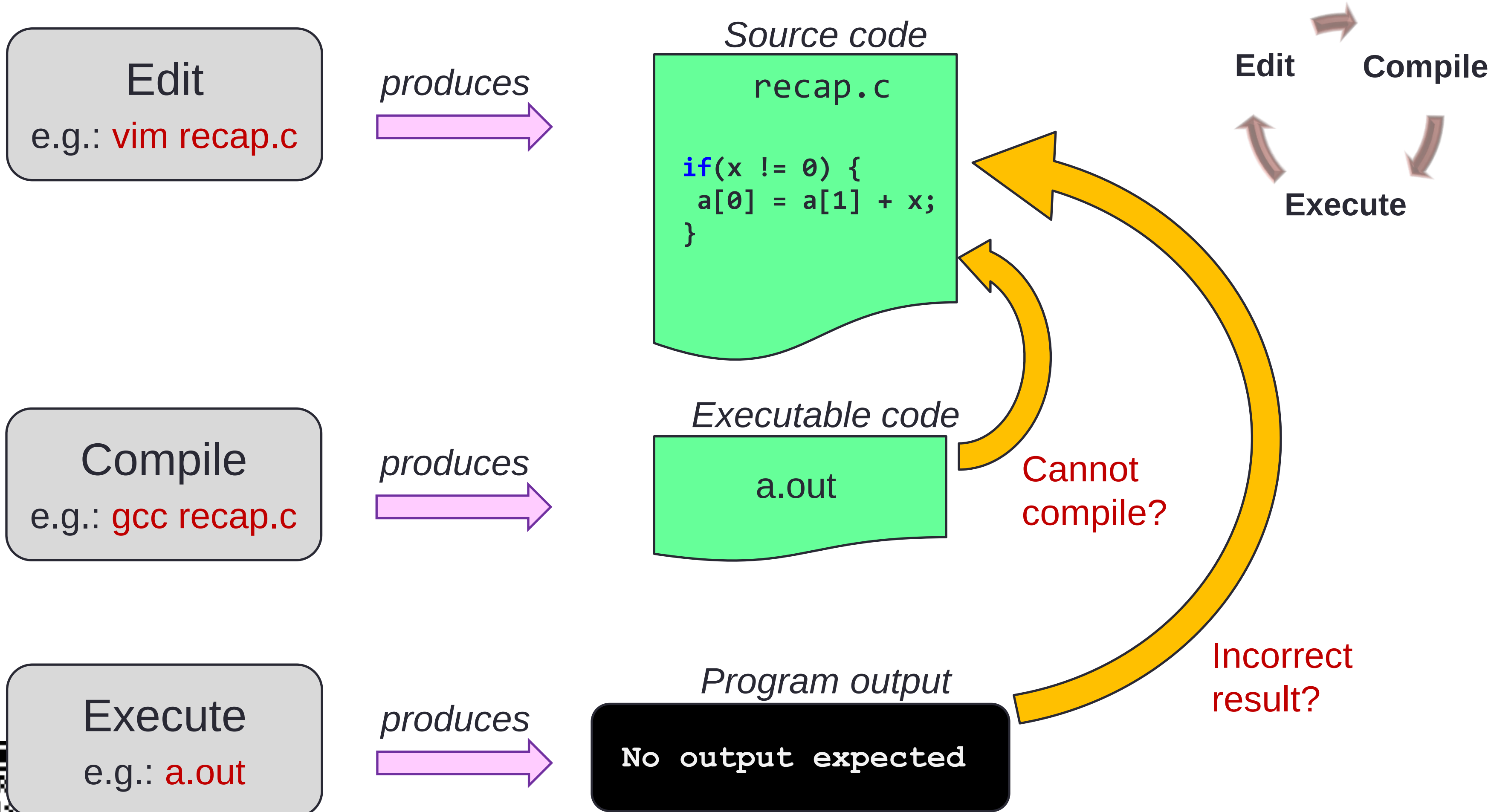
```
if(x != 0) {
  a[0] = a[1] + x;
}
```

  - Programmer:
    - Show the workflow of compiling, assembling and executing C program
  - Compiler:
    - Show how the program is compiled into MIPS
  - Assembler:
    - Show how MIPS assembly is translated into binaries
  - Processor:
    - Show how the datapath is activated in the processor

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

8.1 Writing C Program

- Edit, Compile, Execute: Lecture #2, Slide 6



https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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;
}
```

WARNING

Independently BUT be careful of the following:

1. The label name used have to be all unique.
2. Do not use temporary register named used outside UNLESS you're sure it will not affect the program

  - Each structure can be compiled *independently*

Inner Structure

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

Outer Structure

```
if(x != 0) {

}
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

recap.c

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

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

Note
As a good practice, use $t0-$t9 for registers storing temporary values (e.g., for intermediate computation)

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

Mapping:

```
x: $16
a: $17
```

recap.c

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

8.2 Compiling to MIPS (3/7)

- Common Technique #1:
  Invert the condition for shorter code
  (Lecture #8, Slide 22 (21 in video))

Outer Structure

```
if(x != 0) {



}
```

Outer MIPS Code

```
      beq $16, $0, Else

      # Inner Structure

Else:
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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 (28 in video))

Inner Structure

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

Simplified Inner Structure

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

Note

Another name for this is "name and extract" or "name and conquer". The later is likely to be used by Donald Knuth in his book Concrete Mathematics

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

```
Mapping:

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

```
recap.c

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

8.2 Compiling to MIPS (5/7)

- Common Technique #3:
  Array access is **lw**, array update is **sw**
  (Lecture #8, Slide 13 (12 in video))

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)
```

Note

We really need to know what is the type here. In this case, it is an integer which has 4 bytes of data. So index 1 is at offset of 4

Additional Note

Sometimes, we have to do "detective work" like in the case of tutorial (*spoiler alert*). In Q3 (binary search), there is no type given BUT we have the index being multiplied by 4 that gives indication that it is an int

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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)`

Note

Part of the problem is that in C, we do `ptr++` regardless of the type of the pointer variable `ptr`. But when it is an `int`, we increment by 4 while for `char`, increment by 1

Again, need to really look through the entire question (sometimes do detective work) to determine the type appropriately

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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:
```

Note

Logically here is a fancy name to really just say put the inner structure inside the outer structure (*sandwich*)

8.3 Assembling to Binary (1/6)

recap.mips

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

- 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 17 (16 in video))
     - opcode $rt, $rs, immediate

6	5	5	16
opcode	rs	rt	immediate

  3. Branch: (Lect #9, Slide 24 (23 in vid))
     - Uses I-Format
     - **PC = (PC + 4) + (immediate × 4)**

Note

Note the operand position in MIPS.

```
• add  $rd, $rs, $rt
• sll  $rd, $rt, shamt
• addi $rt, $rs, immediate
• beq  $rs, $rt, label
• lw   $rt, offset($rs)
```

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

recap.mips

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

8.3 Assembling to Binary (2/6)

- **`beq $16, $0, Else`**
  - Compute immediate value (Lect #9, Slide 30 (29 in vid))
    - `immediate` = 3

Note
The steps below can work for both branch up or down. Branch up means the number (step 4) is negative instead

  - 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      +3
     sw  $8, 0($17)
Else:
```

Finding Immediate
4 Easy Steps:
1. Draw line *below* current instruction
2. Draw line above label
3. Close the lines to form rectangle
4. Count number of *instructions!*

LABELS ARE NOT INSTRUCTIONS

```
recap.mips

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

8.3 Assembling to Binary (2/6)

- **`beq $16, $0, Else`**
  - Compute immediate value
    (Lecture #9, Slide 30 (29 in video))
    - **`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:
```

Finding Immediate

4 Easy Steps:
1. Draw line *below* current instruction
2. Draw line above label
3. Close the lines to form rectangle
4. Count number of *instructions!*

LABELS ARE NOT INSTRUCTIONS

```
recap.mips

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

8.3 Assembling to Binary (2/6)

- `beq $16, $0, Else`
  - Compute immediate value (Lecture #9, Slide 30 (29 in video))
    - `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:
```

Finding Immediate

4 Easy Steps:

1. Draw line *below* current instruction
2. Draw line above label
3. Close the lines to form rectangle
4. Count number of *instructions!*

LABELS ARE NOT INSTRUCTIONS

```
recap.mips

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

8.3 Assembling to Binary (2/6)

- **`beq $16, $0, Else`**
  - Compute immediate value
    (Lecture #9, Slide 30 (29 in video))
    - **`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:
```

Finding Immediate

4 Easy Steps:

1. Draw line *below* current instruction
2. Draw line above label
3. Close the lines to form rectangle
4. Count number of *instructions!*

LABELS ARE NOT INSTRUCTIONS

recap.mips

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

8.3 Assembling to Binary (2/6)

- `beq $16, $0, Else`
  - Compute immediate value (Lect #9, Slide 30 (29 in vid))
    - `immediate = 3`
  - Fill in fields (*refer to MIPS Reference Data*)

Note
Branch down means positive value. The number of instructions contained is 3. So immediate = +3

6	5	5	16
4	16	0	3

  - Convert to binary

000100	10000	00000	0000000000000011

```
      beq $16, $0, Else
      lw  $8, 4($17)
+3    add $8, $8, $16
      sw  $8, 0($17)
Else:
```

Finding Immediate
4 Easy Steps:
1. Draw line *below* current instruction
2. Draw line above label
3. Close the lines to form rectangle
4. Count number of *instructions!*

LABELS ARE NOT INSTRUCTIONS

recap.mips

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

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:
```

recap.mips

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

8.3 Assembling to Binary (4/6)

- **`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:
```

```
recap.mips

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

Decode:

ALU:

Memory:

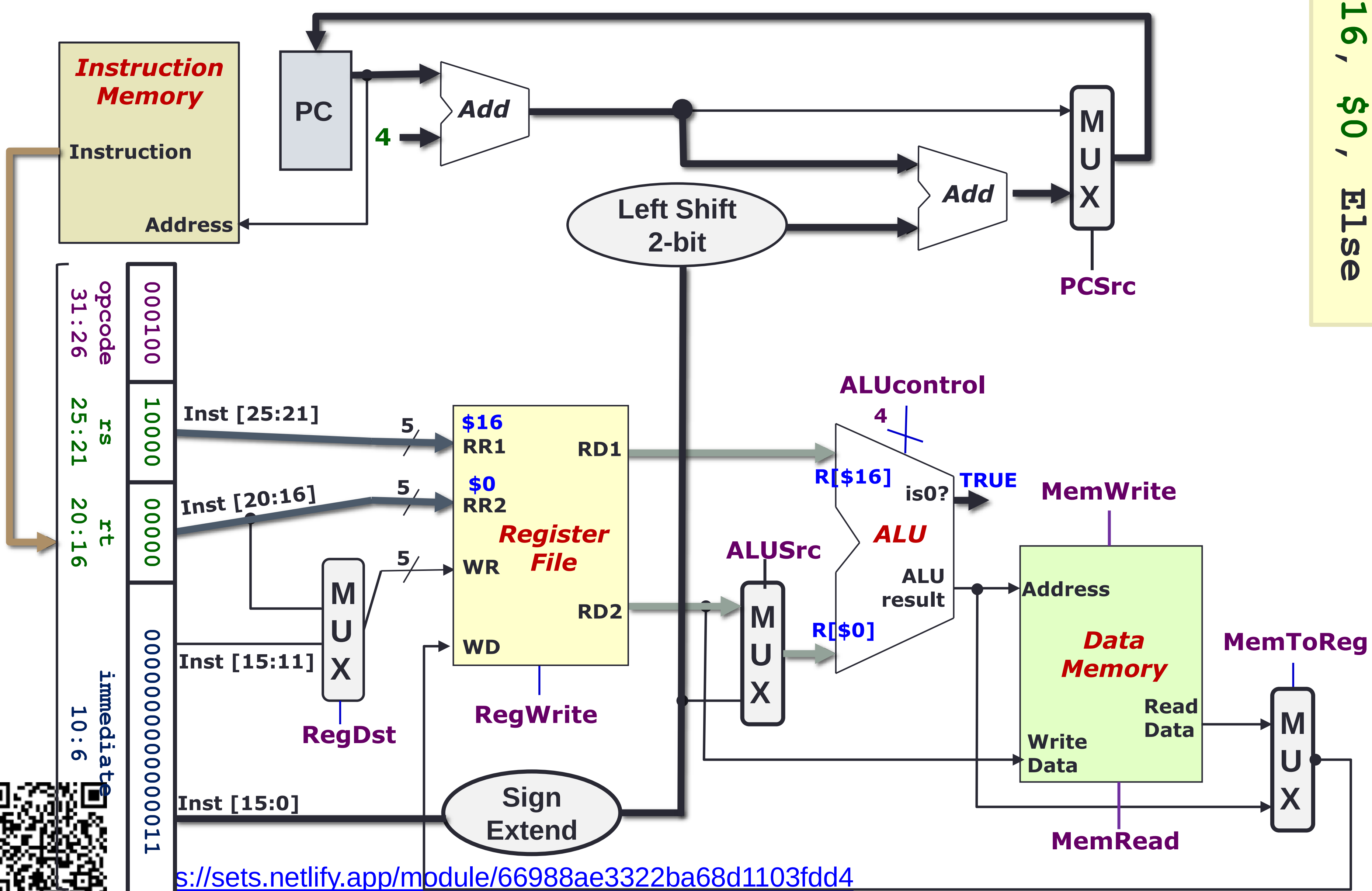
Reg Write:

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
```

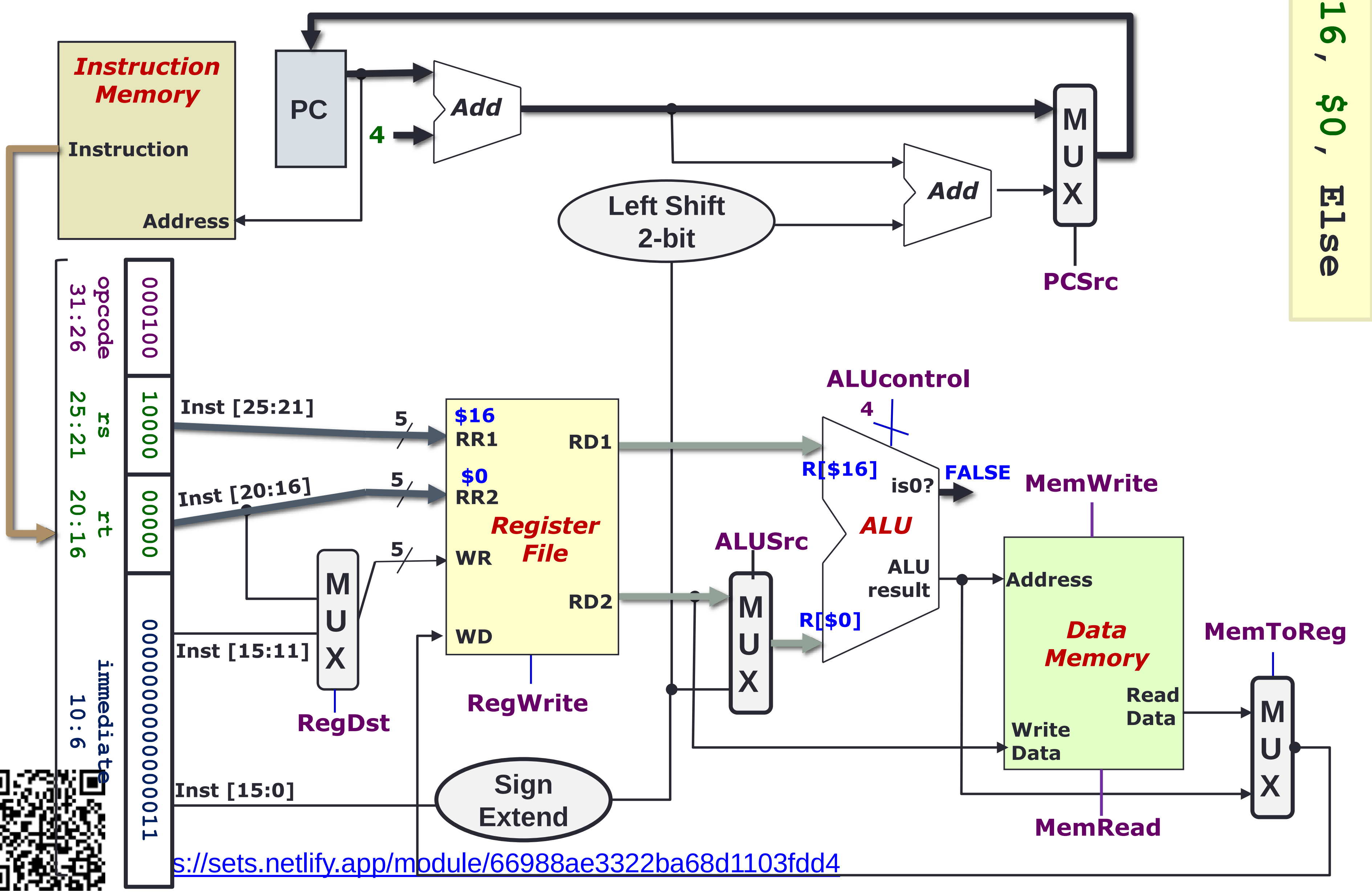
`beq $16, $0, Else`

- Assume `$16 == $0`



s://sets.netlify.app/module/66988ae3322ba68d1103fdd4

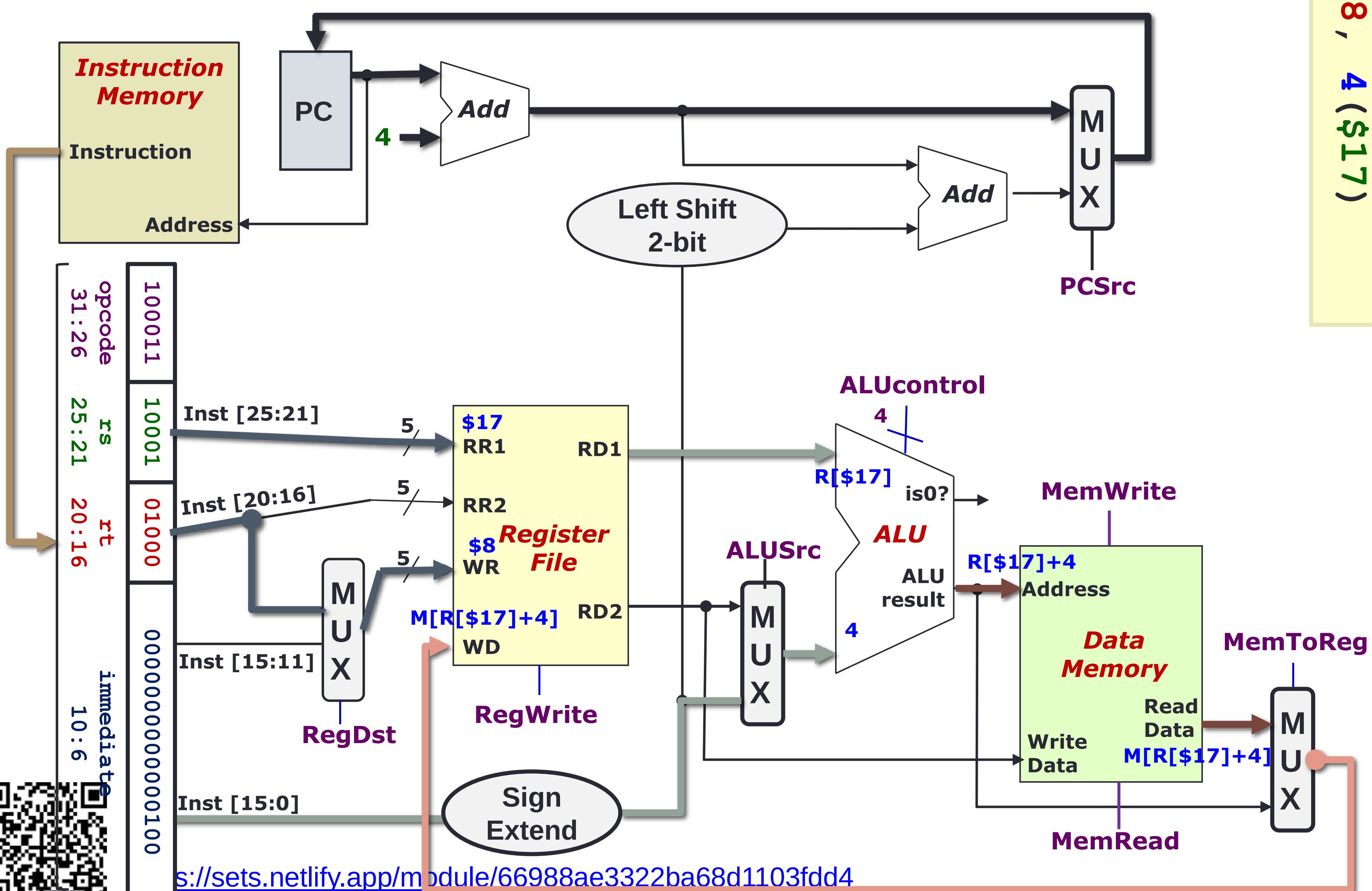
- Assume $16 != $0



s://sets.netlify.app/module/66988ae3322ba68d1103fdd4

- Assume $16 != $0

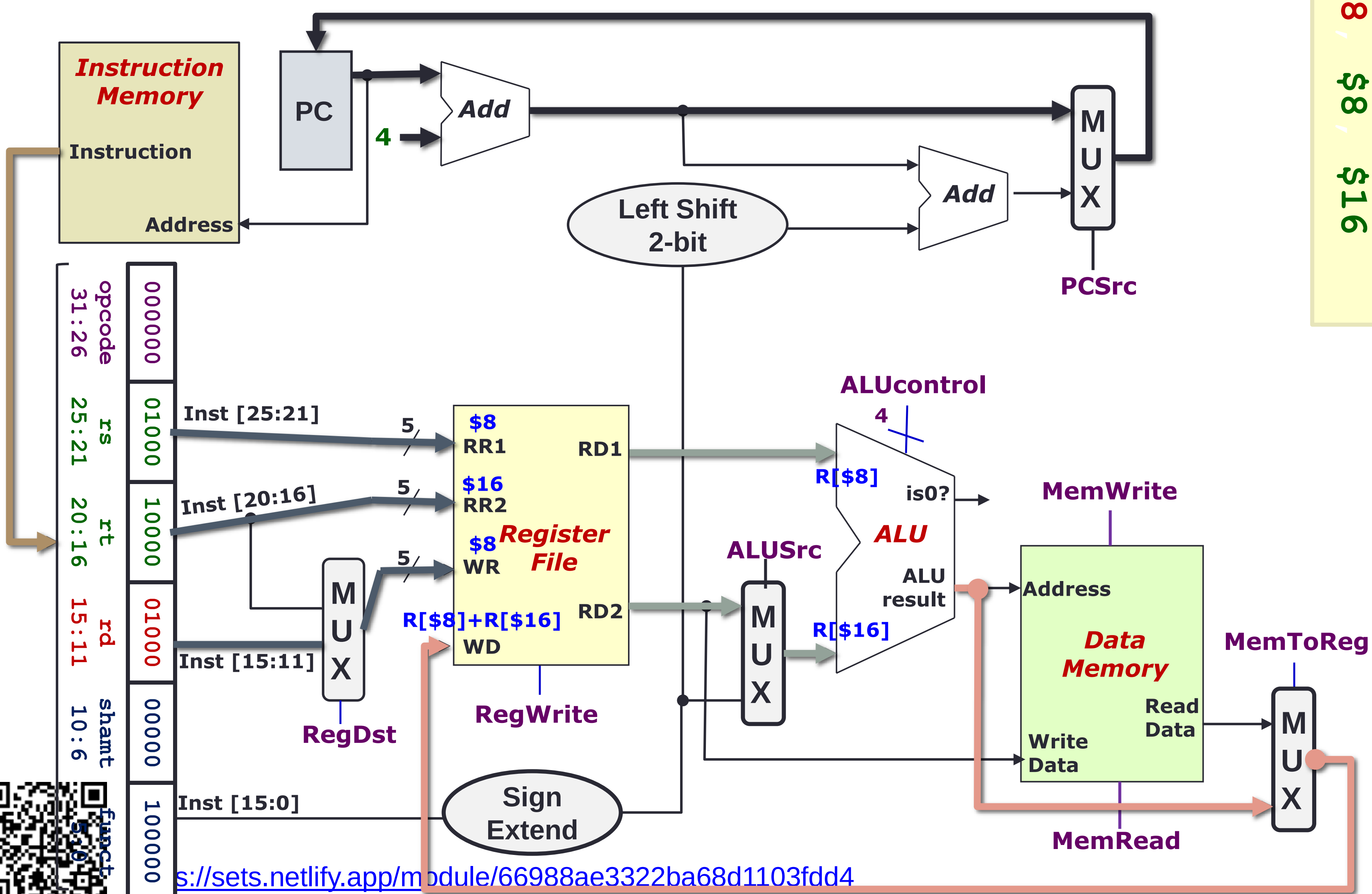
lw $8, 4($17)



s://sets.netlify.app/m?dule/66988ae3322ba68d1103fdd4

- Assume $16 != $0

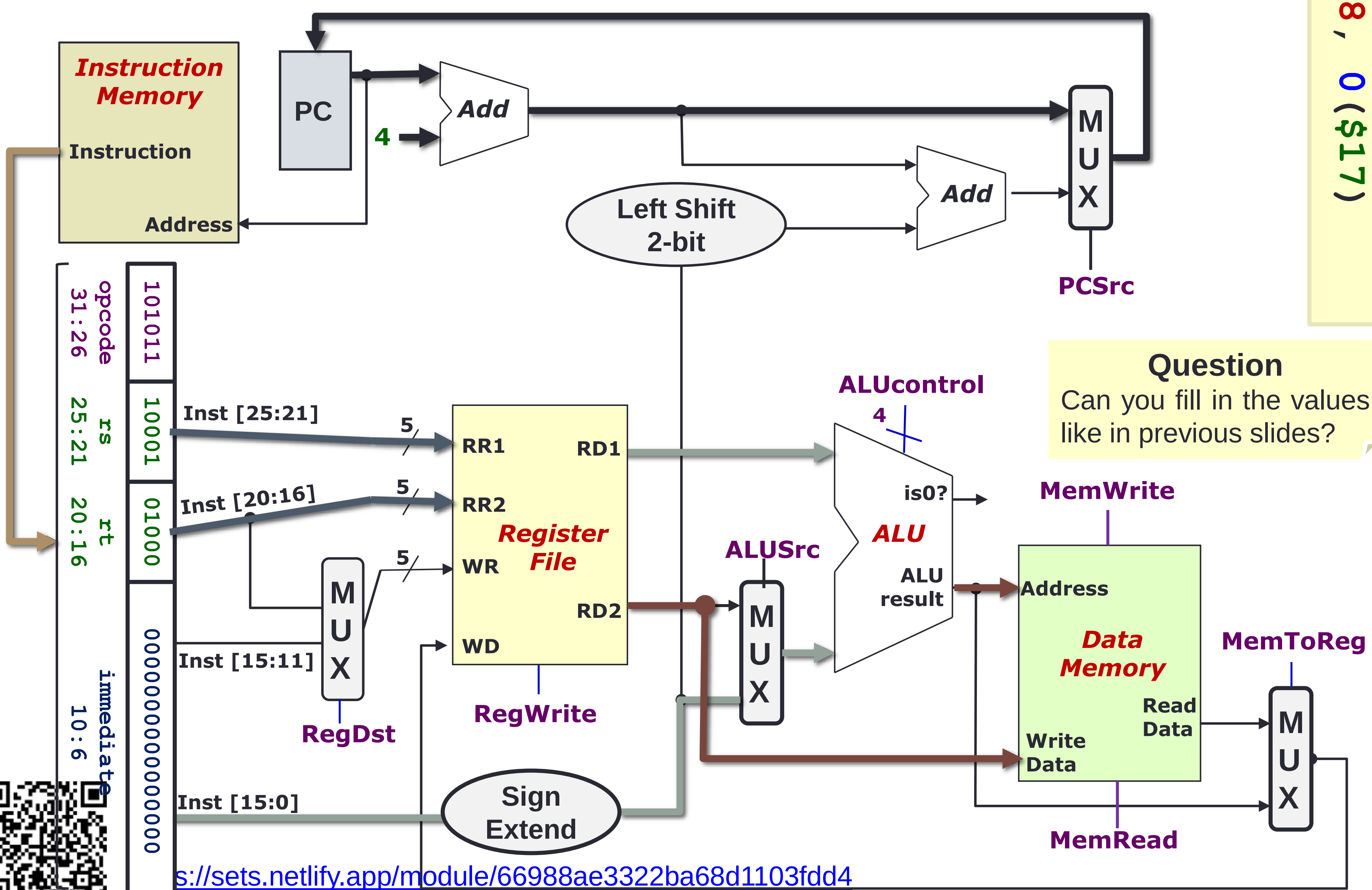
add $8, $8, $16



s://sets.netlify.app/module/66988ae3322ba68d1103fdd4

sw $8, 0($17)

- Assume `$16 != $0`



Question

Can you fill in the values like in previous slides?

s://sets.netlify.app/module/66988ae3322ba68d1103fdd4

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)

https://sets.netlify.app/module/66988ae3322ba68d1103fdd4

End of File