Datapath (Processor)

- encoded binary patterns → processor
- 2 components in processor
  - **Datapath** — collection of components that process data (read/write, ALU). performs arithmetic, logical and memory operations.
  - **Control** — tells the datapath, memory and I/O devices what to do according to program instructions. → Control signals.

- Core MIPS ISA:
  - **Arithmetic & Logic operations**: add, sub, and, or, addi, slt
    (C andi, ori not supported)
    ↳ always do "sign extension" on immediate values.
  - **Data transfer instructions**: lw, sw
  - **Branches**: beq, bne

- Instruction Execution Cycle

next instruction

1. **Fetch**
   - get instr. from memory
   - address is in Program Counter (PC) Register. — stores addr. of next instr. to be executed.
2. **Decode**
   - find out the operation required. — eg. add $s0, $s1, $s2.
3. **Operand Fetch**
   - get operand(s) needed for operation — eg. $s1, $s2.
4. **Execute**
   - perform the required operation ALU ⇒ $opr_1 + opr_2$ → $s0.
5. **Result Write (Store)**
   - Store the result of the operation.

Same for all 3 types of instruction.

	`add $rd, $rs, $rt`	`lw $rt, ofst($rs)`	`beq $rs, $rt, ofst` (16 bit immed.)
Fetch	*standard*	*standard*	*standard*
Decode			
Operand Fetch	○ Read [**$rs**] as *opr1* ○ Read [**$rt**] as *opr2*	○ Read [**$rs**] as *opr1* ○ Use **ofst** as *opr2*	○ Read [**$rs**] as *opr1* ○ Read [**$rt**] as *opr2*
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

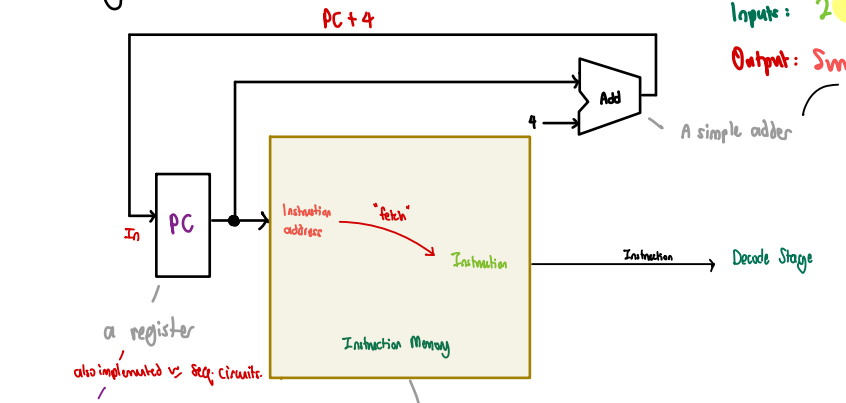
	add $rd, $rs, $rt	lw $rt, ofst($rs)	beq $rs, $rt, ofst
1 Fetch	Read instr from [PC]	Read instr from [PC]	Read instr from [PC]
2 Decode / Operand Fetch	○ Read [$rs] as *opr1* ○ Read [$rt] as *opr2*	○ Read [$rs] as *opr1* ○ Use ***ofst*** as *opr2*	○ Read [$rs] as *opr1* ○ Read [$rt] as *opr2*
3 ALU	*Result* = opr1 + opr2	*MemAddr* = opr1 + opr2	*Taken* = (opr1 == opr2)? calc. here! *Target* = (PC+4) + ofst×4
4 Memory Access		Use *MemAddr* to read from memory	–
5 Result Write	*Result* stored in $rd	*Memory* data stored in $rt	if (*Taken*) PC = *Target*

merged as decode is simple for MIPS.

split execute stage.

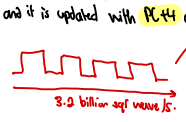
1. Fetch Stage

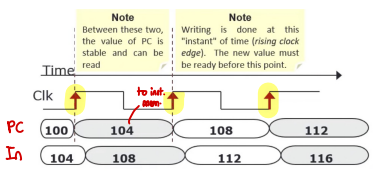
- Requirements
  1. Use the Program Counter (PC) to fetch the instruction from memory.
     - implemented as a special register in the processor.
     - stores addr. of next instr.
  2. Increment the PC by 4 to get the address of the next instruction
     - 32 bits → size of instr.
     - note the exception when branch/jump instruction is executed.
  3. Output to the next stage (Decode)
     - the instruction to be executed.
- Block diagram



Combinational logic to implement the addition of 2 numbers
Inputs: 2 32-bit numbers A, B.
Output: Sum of the input numbers A+B.

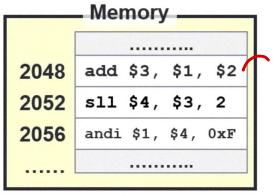
Clock signal
- PC is read during first half of the clock period and it is updated with PC+4 at the next rising clock edge.





special circuits used for storing data. i.e. instructions.

- It is a sequential circuit
- Has an internal state that stores information
- Clock signal is assumed and is 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



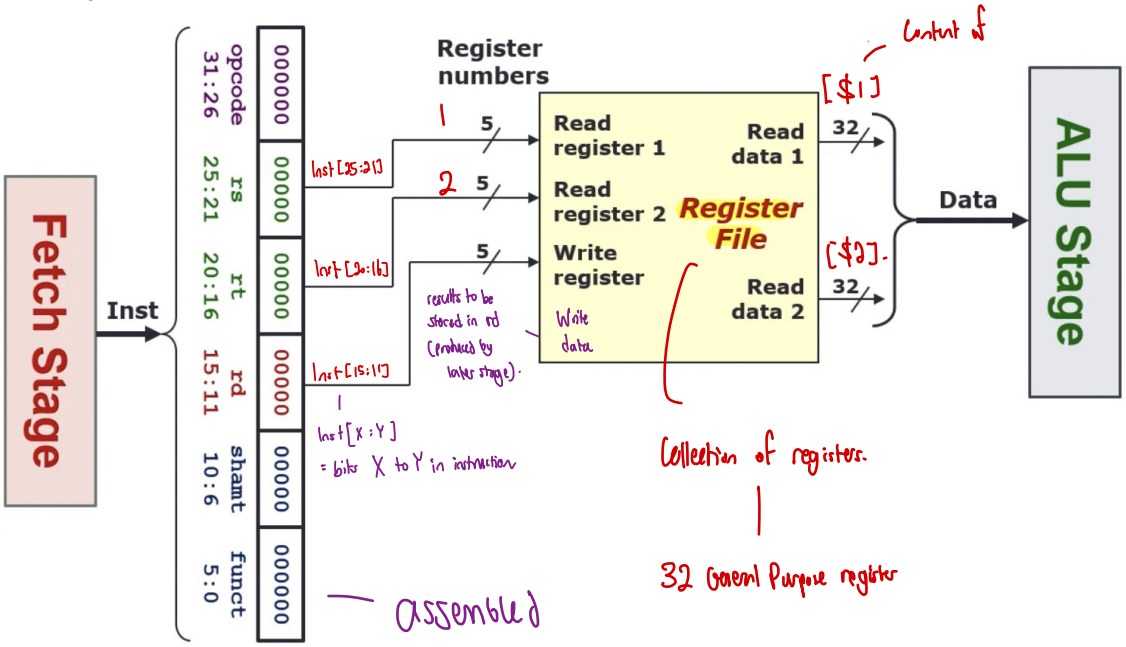
as binary

As a Function

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

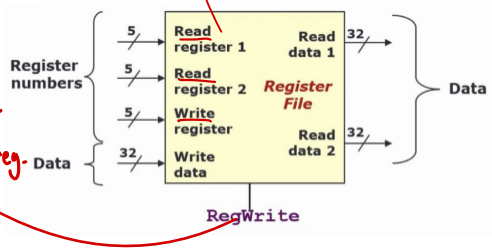
2. Decode Stage

- Requirements
  - Gather Data from the instruction fields (32-bit binary)
    1. Read the opcode to determine instruction type and field lengths
    2. Read data from all necessary registers
       i.e. 2 - add , 1 - addi , 0 - j format
  - Input from previous stage Fetch
    - Instruction to be executed.
  - Output to the next stage ALU
    - Operation and the necessary operands.
- Block Diagram:



- each 32-bit wide; can be read/written by specifying register number
- read at most 2 registers per instruction
- write at most 1 register per instruction
- RegWrite is a control signal to indicate:
  - writing of register
  - 1 (True) = Write
  - 0 (False) = No Write

Control signal
→ data should be written in write register or not.
→ 1 : write data to write reg.
0 : nothing happens.

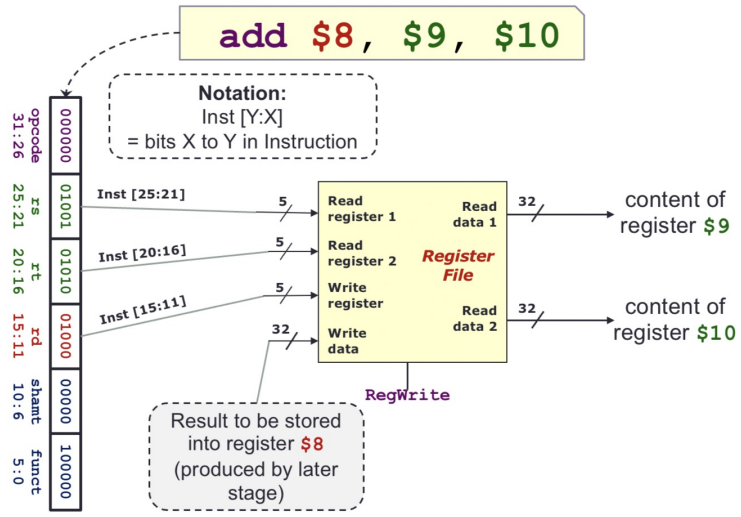


As a Function

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

Examples

R-format Instructions

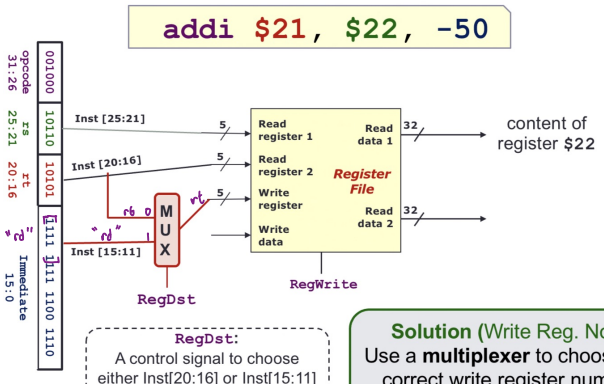


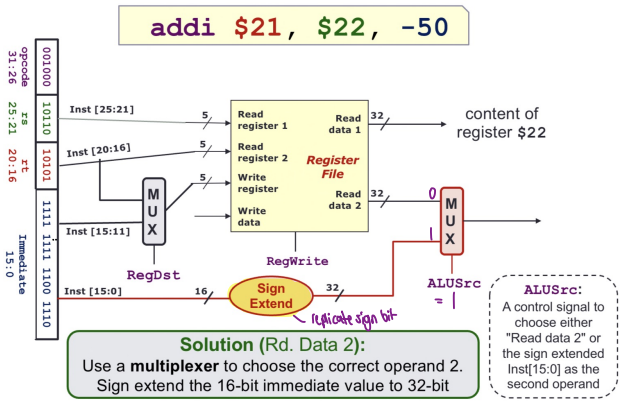
I-format Instructions.

- problems:
  1. Choose Write registers
  2. Choose operand 2 for ALU
- Fix:

1. Use multiplexer – a device w/ 2 inputs, choose either input
   ↳ using control signal

2. Use another multiplexer





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

- always 0 in I-format to pass in rt
- always 1 in R-format

- Function: select one input from multiple input lines.
- Inputs: n lines of some width.
- Control: m bits where $n = 2^m$ (in binary)
- Output: select i^{th} input line if control = i

– don't need to change anything for load word

....

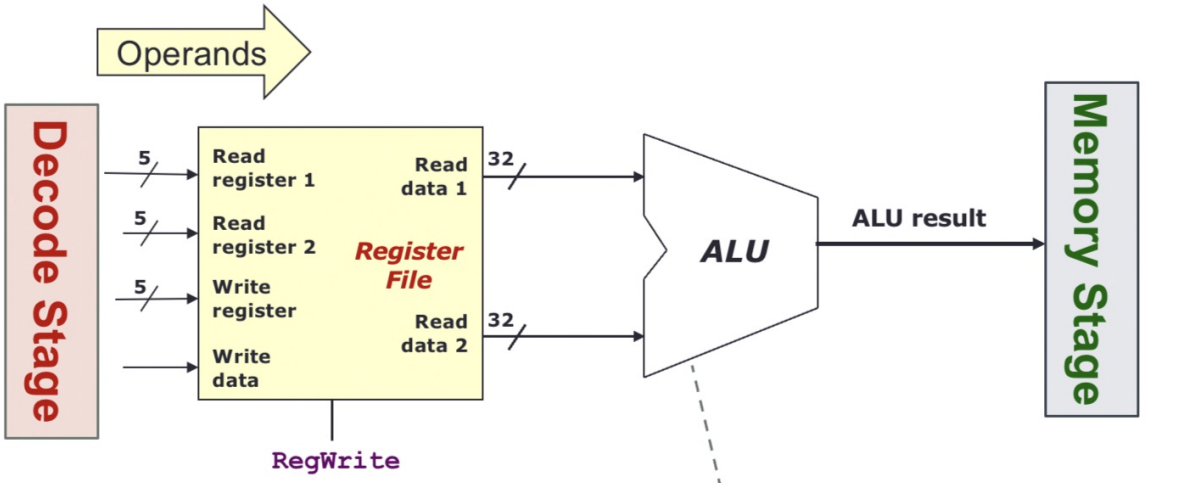
3. ALU stage (Execute Stage)

- Requirements:

1. Instruction @ ALU stage
   - ALU = Arithmetic Logic Unit
   - AKA Execution Stage (Ex)
   - Perform the real work for most instructions here
     - Arithmetic (eg. add, sub), Shifting (eg. sll), Logical (eg. and, or)
     - Memory Operation (eg. lw, sw) – Address Calculations
     - Branch operation (eg. bne, beq) – Perform register comparison and target address calculation.

2. Input from prev. Stage (Decode)
   - Operations & operands

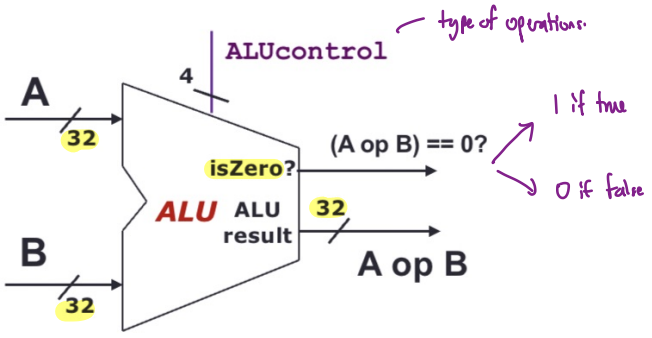
3. Output to next stage (Memory)
   - Calculation result.



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];
  :
}
```

- **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

A AND B.
A OR B
⋮

Control Path vs Data Path.
- Control signals
  - RegDst < rt / rd
  - ALUsrc < Rd2 / immed
  - ALUctrl
- Read data.
- ALU operations on data.

Branch in ALU

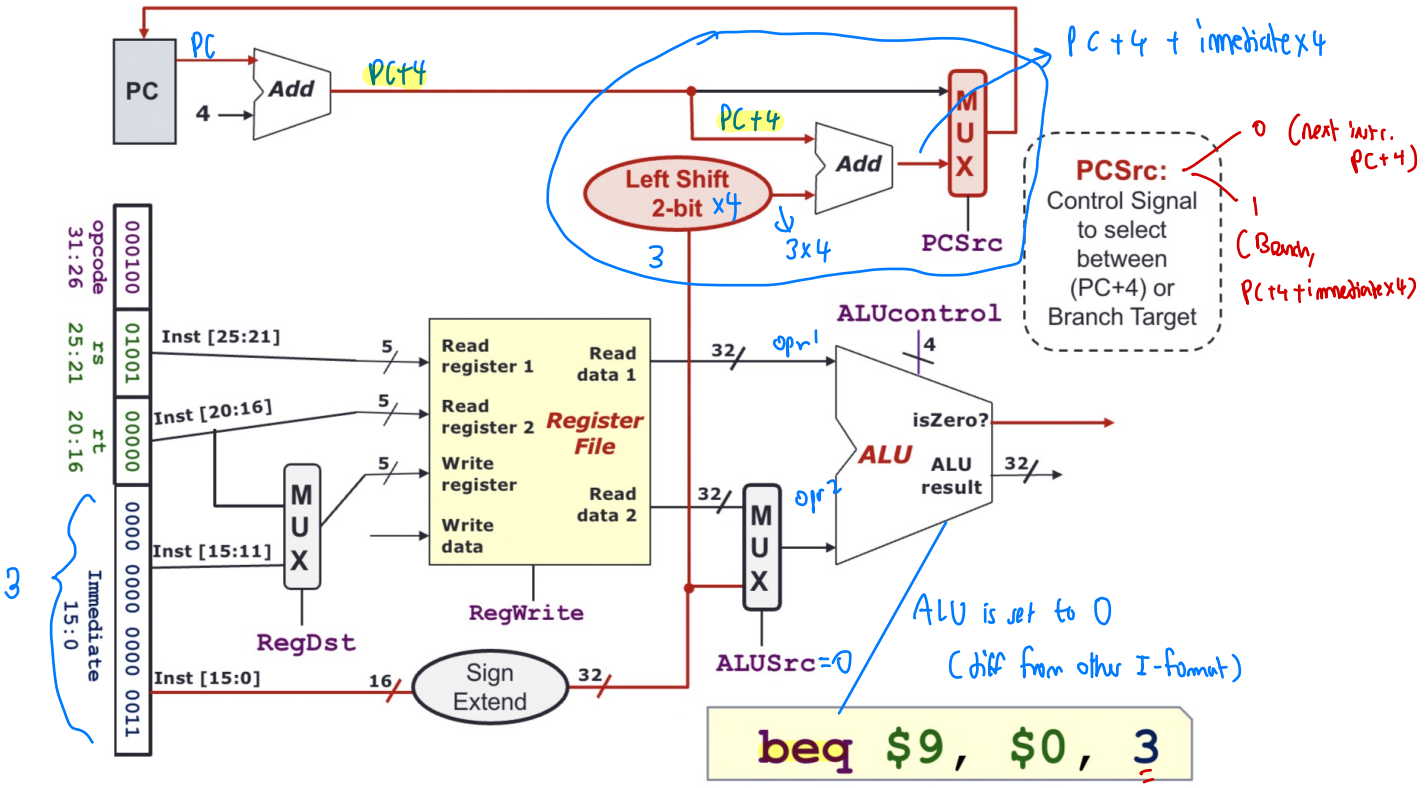
1. Branch Outcome
   - Use ALU to compare the register
   - The 1 bit "isZero" signal is enough to handle equal/not equal check

2. Branch Target Address
   - Introduce additional logic to calculate the address
   - Need PC — from fetch Stage.
   - Need offset - from Decode Stage.

PC+4 + immed×4.
offset.

→ 2 things need to happen to take branch
1. The instruction is a branch instruction.
2. The condition of the branch is true

Complete ALU stage for branch instruction



eg. beq $9 $0 label

1 add $0 $1 $2

2 sub $s0 $s0 $t2

3 addi $t0 $t1 100

Label: sub $s0 $s1 $t0

4. Memory Stage

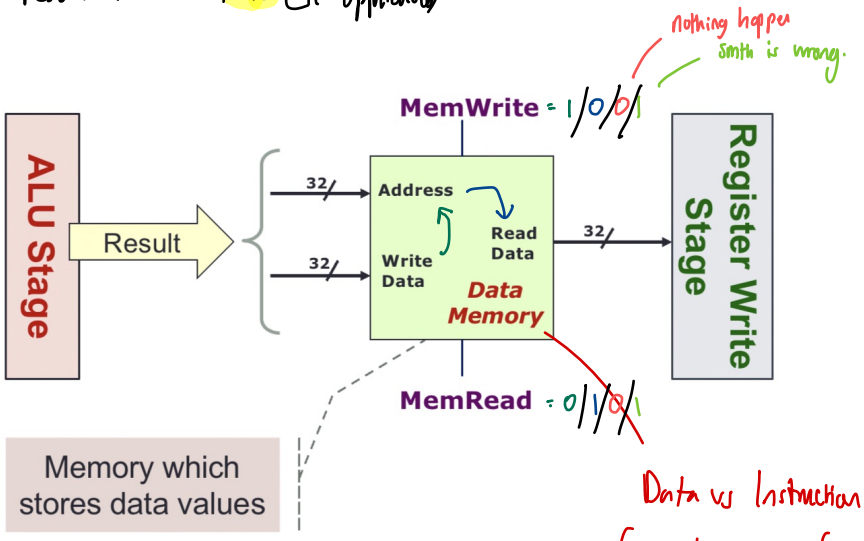
. Requirements:

1. Instruction @ Memory 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 (if applicable).

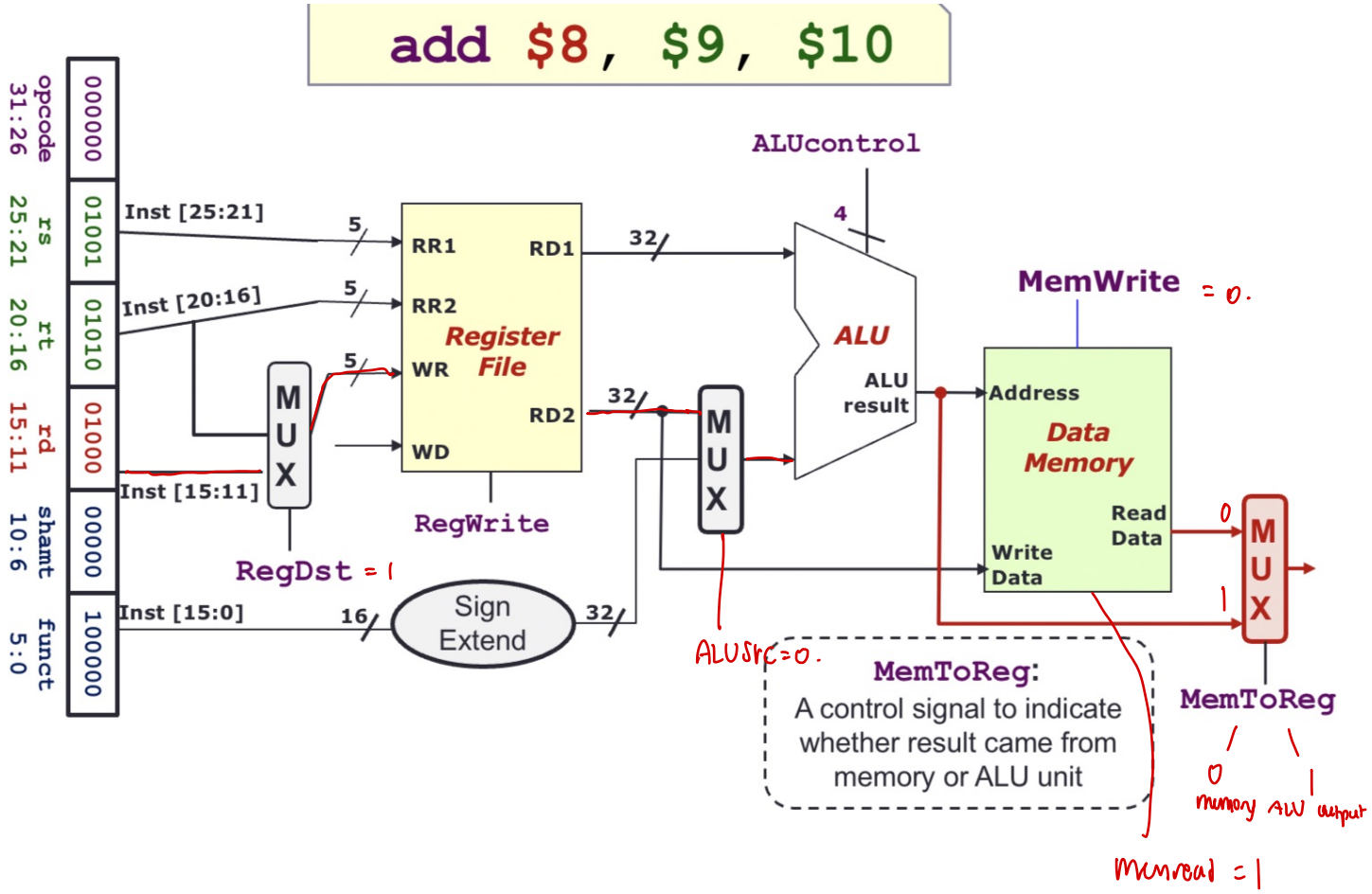
RegValue from Decode Stage

2. Input from previous stage (ALU):
   - computational result to be used as memory address (if applicable)

3. Output to the next stage (Register Write)
   - result to be stored (if applicable).



Data vs Instruction Memory
(prog data) (prog. instr.)


add $8, $9, $10
opcode 31:26
rs 25:21
rt 20:16
rd 15:11
shamt 10:6
funct 5:0
000000
01001
01010
01000
00000
100000
Inst [25:21]
Inst [20:16]
Inst [15:11]
Inst [15:0]
5
5
5
16
RR1
RR2
WR
WD
RD1
RD2
Register File
RegWrite
RegDst = 1
MUX
Sign Extend
32
32
32
ALUcontrol
4
ALU
ALU result
ALUSrc=0.
MemWrite = 0.
Address
Data Memory
Read Data
Write Data
MemRead = 1
0
1
MemToReg
0
1
memory
ALU output
MemToReg:
A control signal to indicate whether result came from memory or ALU unit

5 Result Write Stage.

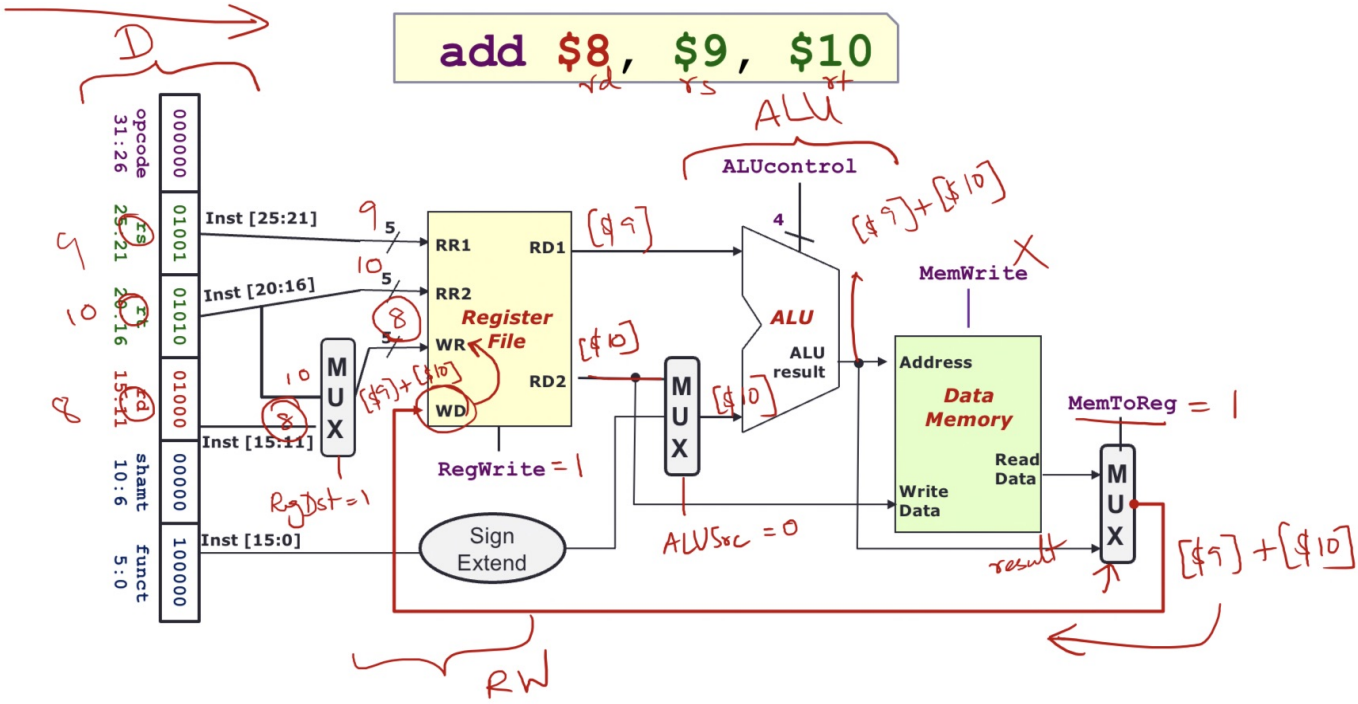
- Requirements:
  1. Instruction @ Register Write.
     - Most instr. write the result of some computation into a register.
       - eg. arithmetic, logical, shifts, loads, set-less-than.
       - need destination register number and computation result.
     - Exceptions are stores, branches, jumps.
       - there are no results to be written
       - instructions remain idle in this stage.

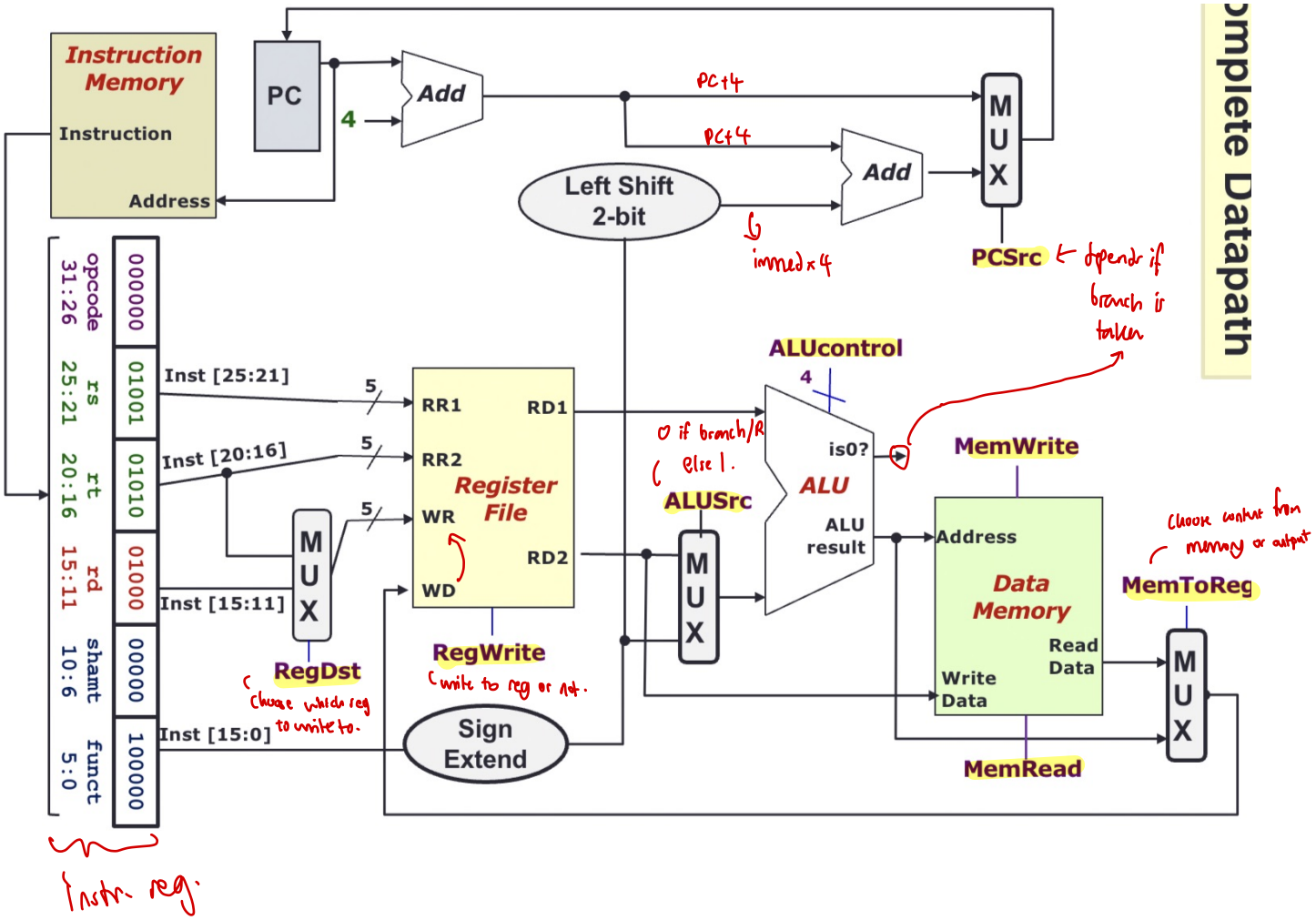
① test reg. no.

② result to write to destination reg

  2. Input from previous stage (Memory)
     - Computation result either from memory or ALU.

- Result write stage has no additional elements.
  - just route the correct result into register file.
  - the Write Register number is generated way back in the decode stage.





Assembly Instruction to Binary.

- See opcode 000000 ⇒ R-format

  000010 ⇒ J-format

  else ⇒ I-format.