# Computer Architecture

Trần Trọng Hiếu
Information Systems Department
Faculty of Technology
College of Technology
hieutt@vnu.edu.vn

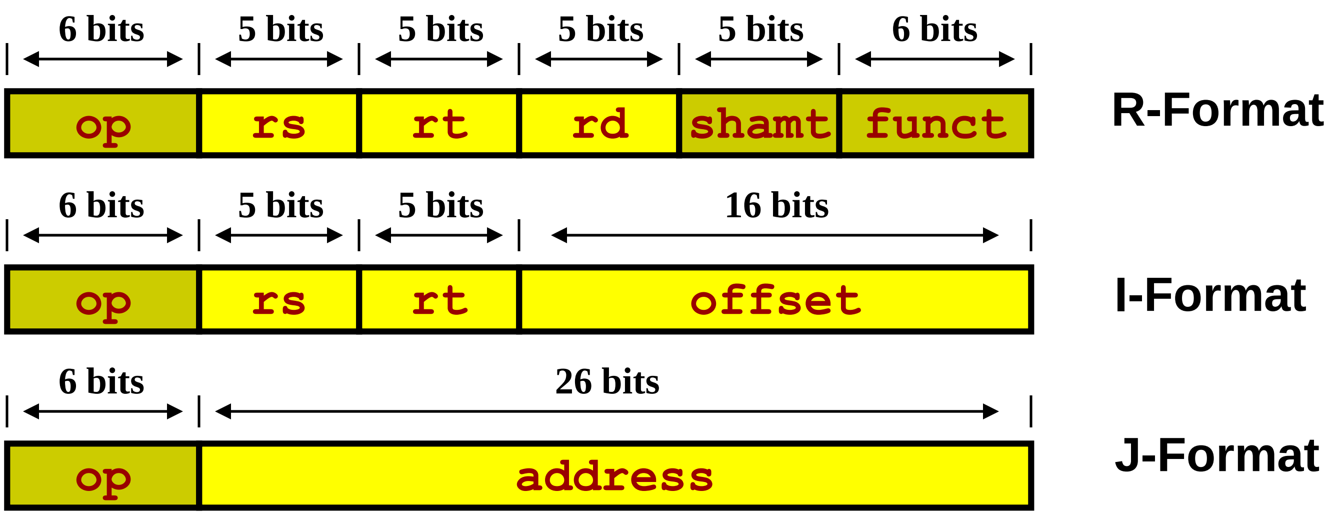
# More on the Processor

# Instruction Execution

- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC $\leftarrow$ target address or PC + 4

# Implementing MIPS

- We're ready to look at an implementation of the MIPS instruction set
- Simplified to contain only
  - arithmetic-logic instructions: `add, sub, and, or, slt`
  - memory-reference instructions: `lw, sw`
  - control-flow instructions: `beq, j`



# ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

| ALU control | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | set-on-less-than |
| 1100 | NOR |

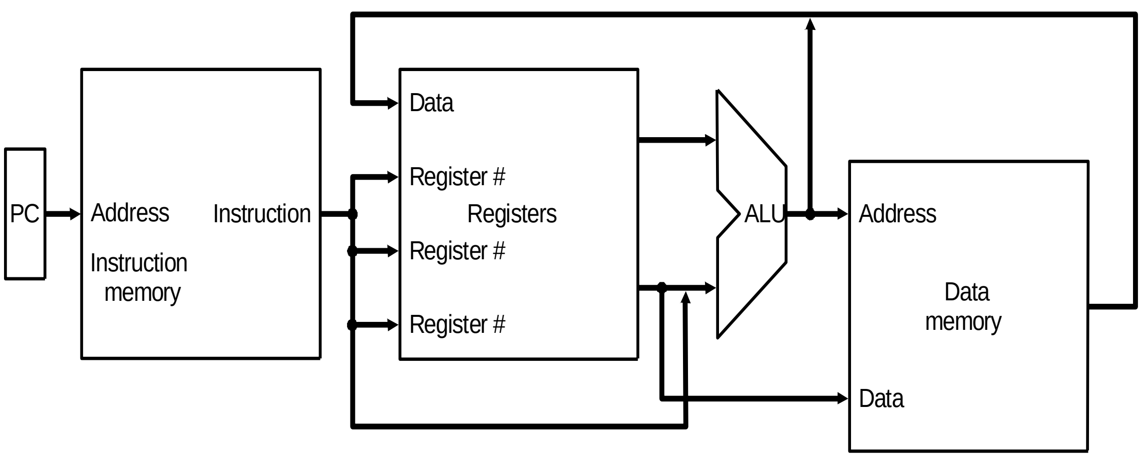
# ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

| opcode | ALUOp | Operation | funct | ALU function | ALU control |
|---|---|---|---|---|---|
| lw | 00 | load word | XXXXXX | add | 0010 |
| sw | 00 | store word | XXXXXX | add | 0010 |
| beq | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| | | subtract | 100010 | subtract | 0110 |
| | | AND | 100100 | AND | 0000 |
| | | OR | 100101 | OR | 0001 |
| | | set-on-less-than | 101010 | set-on-less-than | 0111 |

# Implementing MIPS: the Fetch/Execute Cycle

- High-level abstract view of *fetch/execute* implementation
  - use the program counter (PC) to read instruction address
  - *fetch* the instruction from memory and increment PC
  - use fields of the instruction to select registers to read
  - *execute* depending on the instruction
  - repeat…



# Overview: Processor Implementation Styles

- Single Cycle
  - perform each instruction in 1 clock cycle
  - clock cycle must be long enough for slowest instruction; therefore,
  - disadvantage: only as fast as slowest instruction
- Multi-Cycle
  - break fetch/execute cycle into multiple steps
  - perform 1 step in each clock cycle
  - advantage: each instruction uses only as many cycles as it needs
- Pipelined
  - execute each instruction in multiple steps
  - perform 1 step / instruction in each clock cycle
  - process multiple instructions in parallel – assembly line

# Breaking instructions into steps

- Our goal is to break up the instructions into *steps* so that
  - each step takes one clock cycle
  - the amount of work to be done in each step/cycle is about equal
  - each cycle uses at most once each major functional unit so that such units do not have to be replicated
  - functional units can be shared between different cycles within one instruction
- Data at end of one cycle to be used in next *must be stored* !!

# Breaking instructions into steps

- We break instructions into the following *potential* execution steps – not all instructions require all the steps – each step takes one clock cycle
  1. Instruction fetch and PC increment (**IF**)
  2. Instruction decode and register fetch (**ID**)
  3. Execution, memory address computation, or branch completion (**EX**)
  4. Memory access or R-type instruction completion (**MEM**)
  5. Memory read completion (**WB**)

- Each MIPS instruction takes from 3 – 5 cycles (steps)

# Step 1: Instruction Fetch & PC Increment (IF)

- Use PC to get instruction and put it in the instruction register.

  Increment the PC by 4 and put the result back in the PC.

- Can be described succinctly using *RTL* (*Register-Transfer Language*):

```
IR = Memory[PC];
PC = PC + 4;
```

# Step 2: Instruction Decode and Register Fetch (ID)

- Read registers rs and rt in case we need them.

  Compute the branch address in case the instruction is a branch.

- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) <<
2);
```

# Step 3: Execution, Address Computation or Branch Completion (EX)

- ALU performs one of four functions *depending* on instruction type
  - memory reference:
    `ALUOut = A + sign-extend(IR[15-0]);`
  - R-type:
    `ALUOut = A op B;`
  - branch (instruction *completes*):
    `if (A==B) PC = ALUOut;`
  - jump (instruction *completes*):
    `PC = PC[31-28] || (IR(25-0) << 2)`

# Step 4: Memory access or R-type Instruction Completion (MEM)

- Again *depending* on instruction type:
- Loads and stores access memory
  - load

    `MDR = Memory[ALUOut];`
  - store (instruction *completes*)

    `Memory[ALUOut] = B;`

- R-type (instructions *completes*)

  `Reg[IR[15-11]] = ALUOut;`

# Step 5: Memory Read Completion (WB)

- Again *<u>depending</u>* on instruction type:
- Load writes back (instruction *completes*)

```
Reg[IR[20-16]]= MDR;
```

Important: There is no reason from a datapath (or control) point of view that Step 5 cannot be eliminated by performing

```
Reg[IR[20-16]]= Memory[ALUOut];
```

for loads in Step 4. This would eliminate the MDR as well.

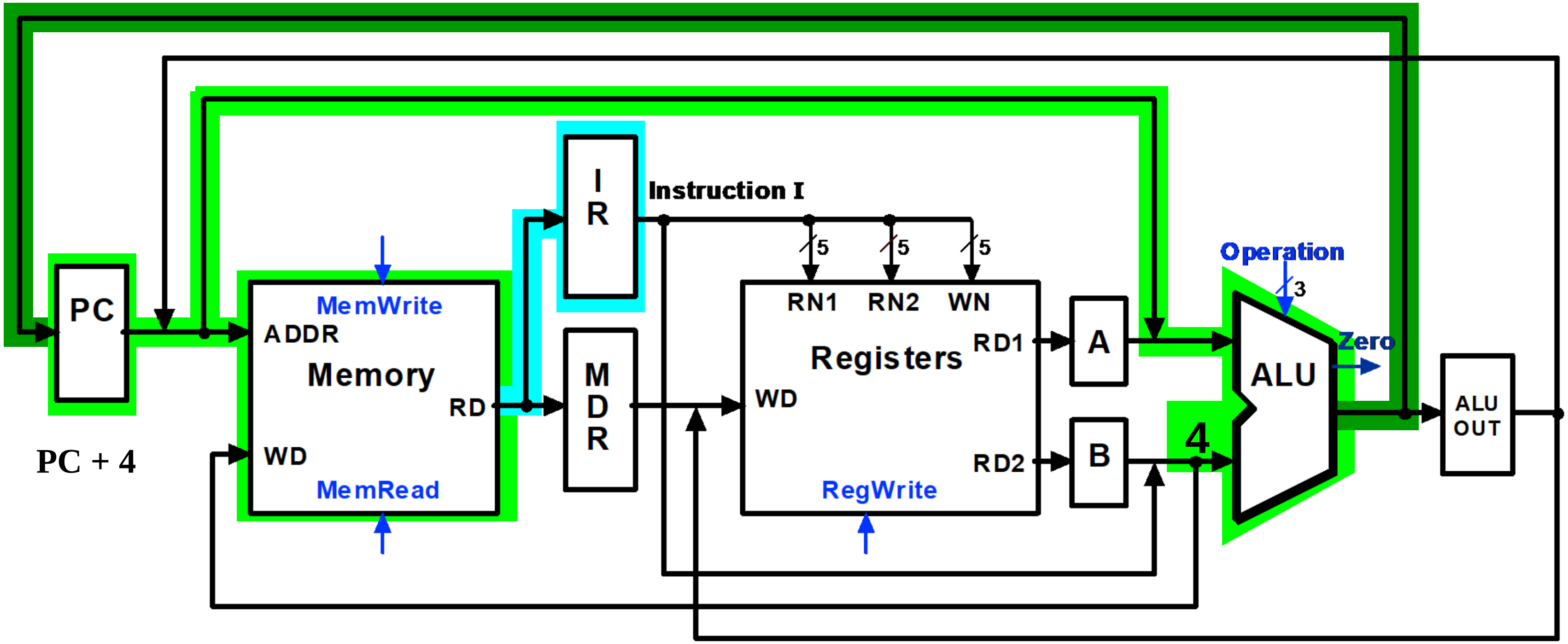
The reason this is not done is that, to keep steps balanced in length, the design restriction is to allow each step to contain *at most* one ALU operation, or one register access, or one memory access.

# Summary of Instruction Execution

| **Step** | **Step name** | **Action for R-type instructions** | **Action for memory-reference instructions** | **Action for branches** | **Action for jumps** |
|---|---|---|---|---|---|
| **1: IF** | Instruction fetch | IR = Memory[PC]<br>PC = PC + 4 | | | |
| **2: ID** | Instruction decode/register fetch | A = Reg [IR[25-21]]<br>B = Reg [IR[20-16]]<br>ALUOut = PC + (sign-extend (IR[15-0]) << 2) | | | |
| **3: EX** | Execution, address computation, branch/ jump completion | ALUOut = A op B | ALUOut = A + sign-extend (IR[15-0]) | if (A ==B) then PC = ALUOut | PC = PC [31-28] II (IR[25-0]<<2) |
| **4: MEM** | Memory access or R-type completion | Reg [IR[15-11]] = ALUOut | Load: MDR = Memory[ALUOut]<br>or<br>Store: Memory [ALUOut] = B | | |
| **5: WB** | Memory read completion | | Load: Reg[IR[20-16]] = MDR | | |

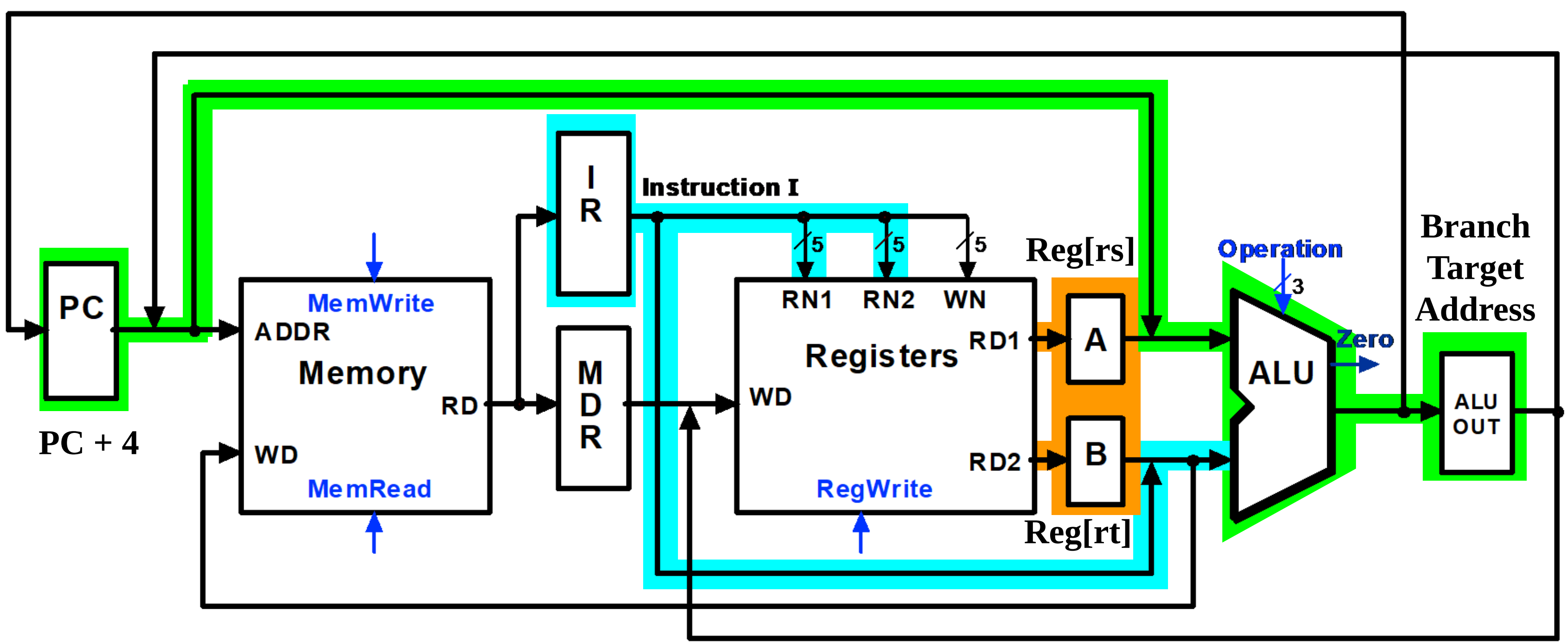
# Multicycle Execution Step (1): Instruction Fetch

```
IR = Memory[PC];
PC = PC + 4;
```



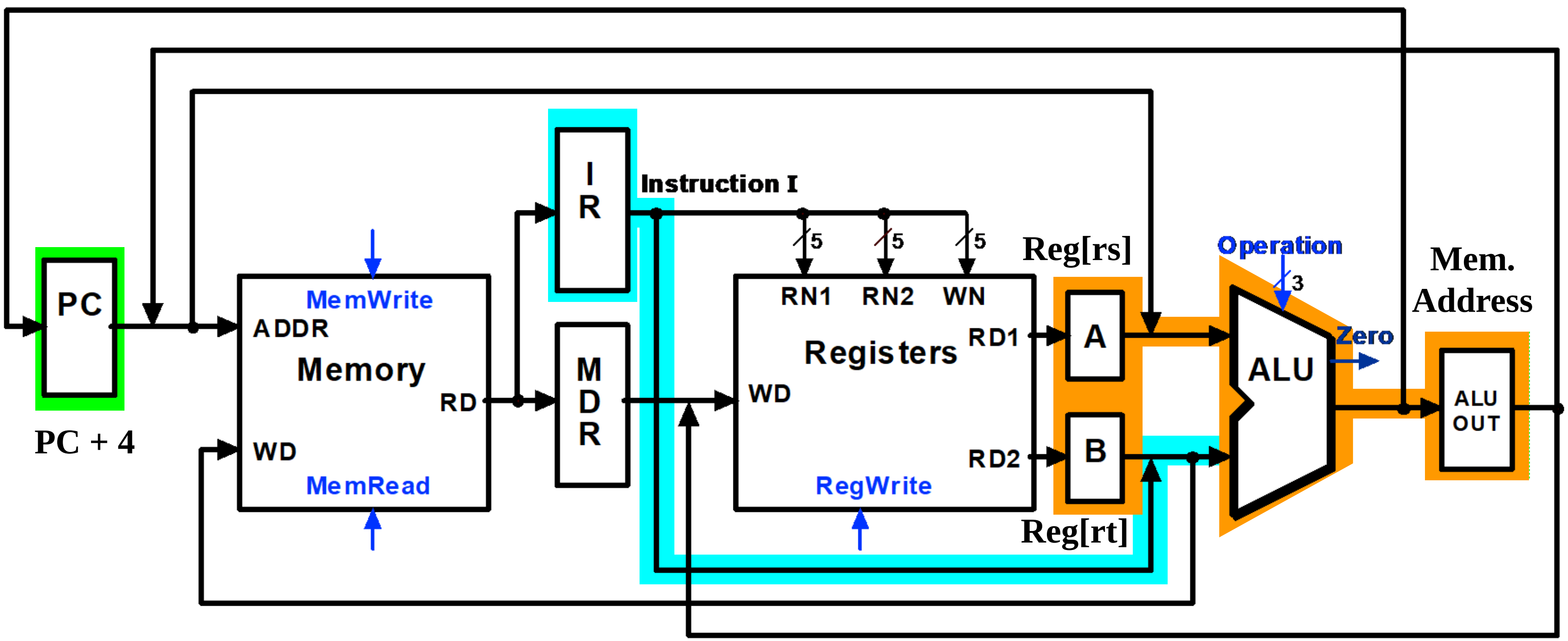
# Multicycle Execution Step (2): Instruction Decode & Register Fetch

```
A = Reg[IR[25-21]];                 (A = Reg[rs])
B = Reg[IR[20-15]];                 (B = Reg[rt])
ALUOut = (PC + sign-extend(IR[15-0]) << 2)
```



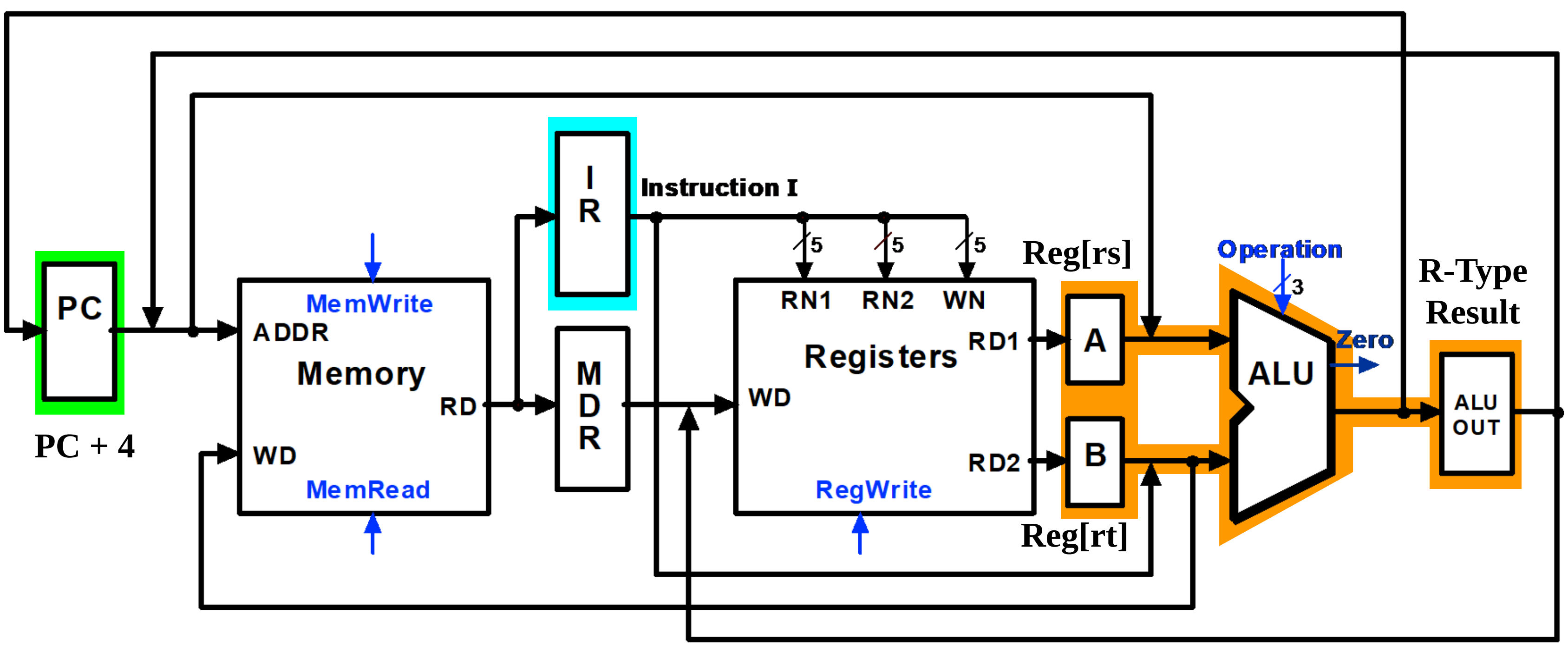
# Multicycle Execution Step (3): Memory Reference Instructions

```
ALUOut = A + sign-extend(IR[15-0]);
```



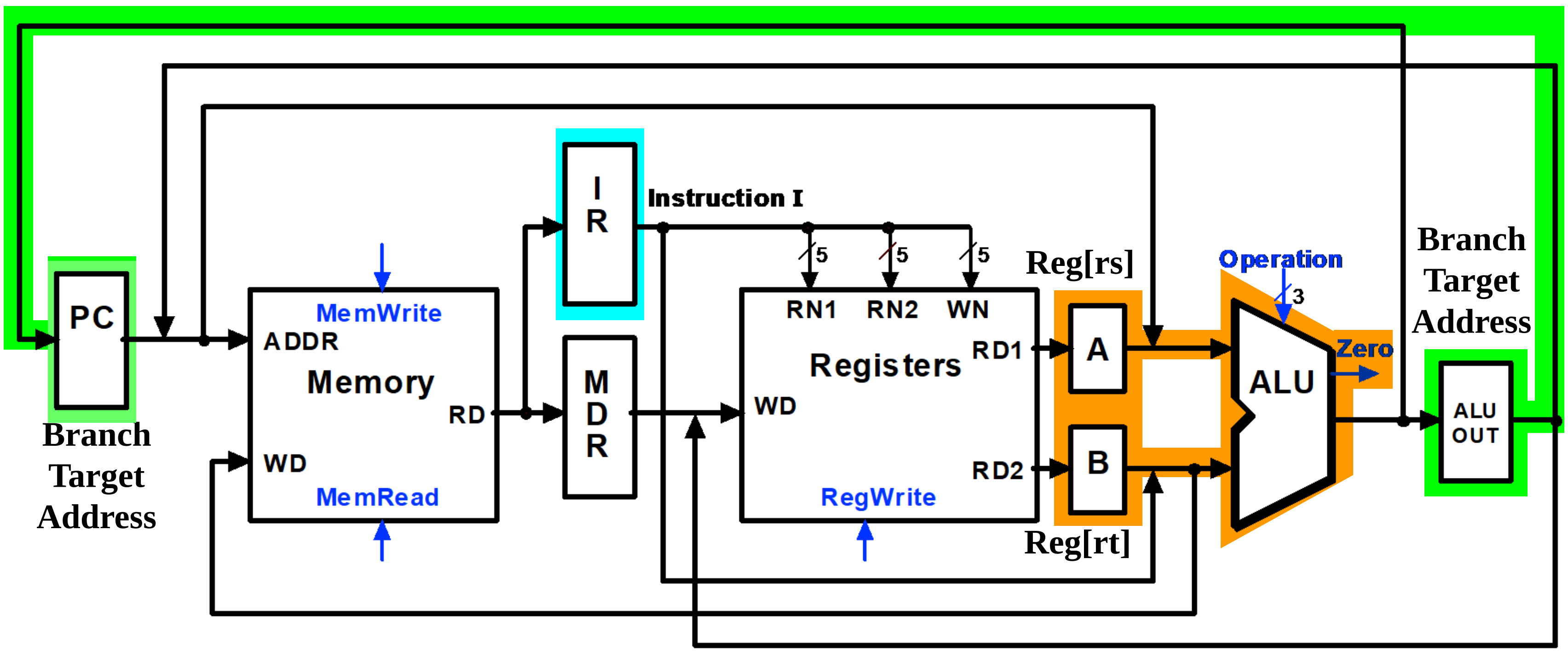
# Multicycle Execution Step (3): ALU Instruction (R-Type)

```
ALUOut = A op B
```



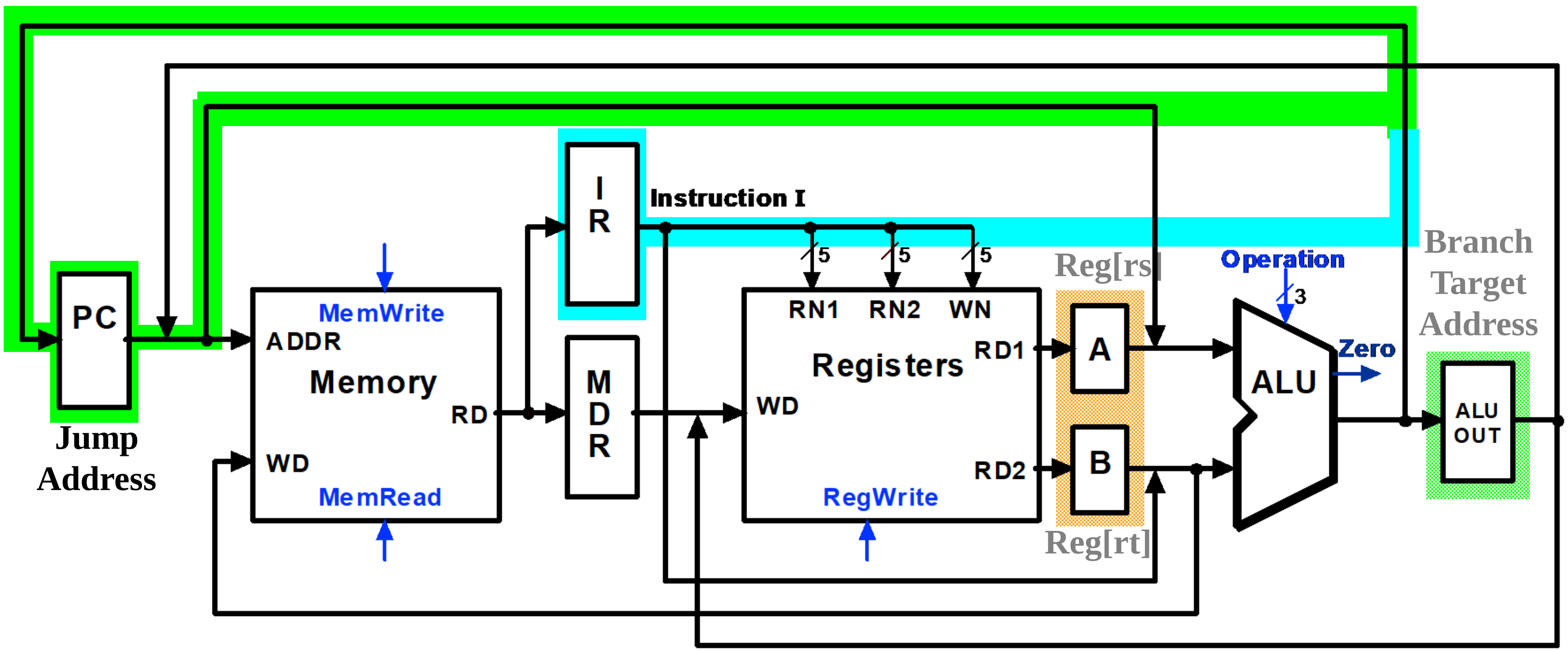
# Multicycle Execution Step (3): Branch Instructions

```
if (A == B) PC = ALUOut;
```



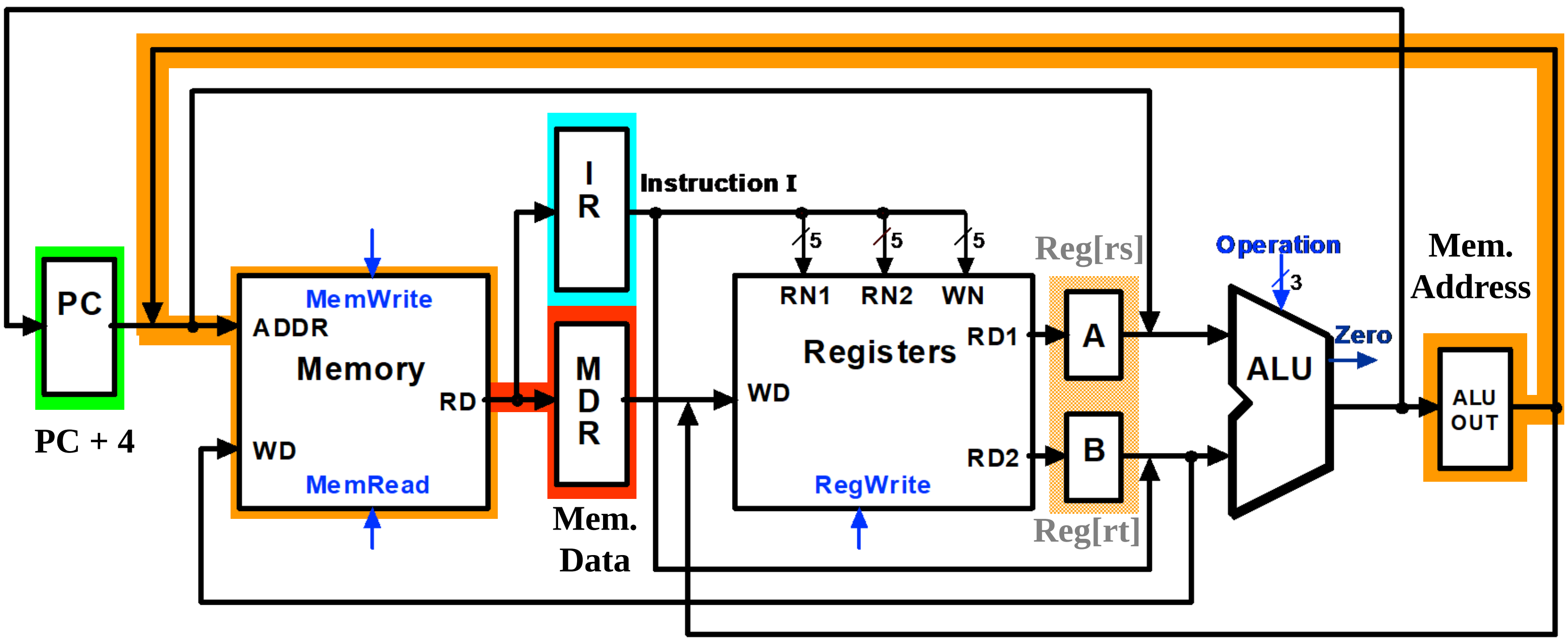
# Multicycle Execution Step (3): Jump Instruction

```
PC = PC[31-28] concat (IR[25-0] << 2)
```



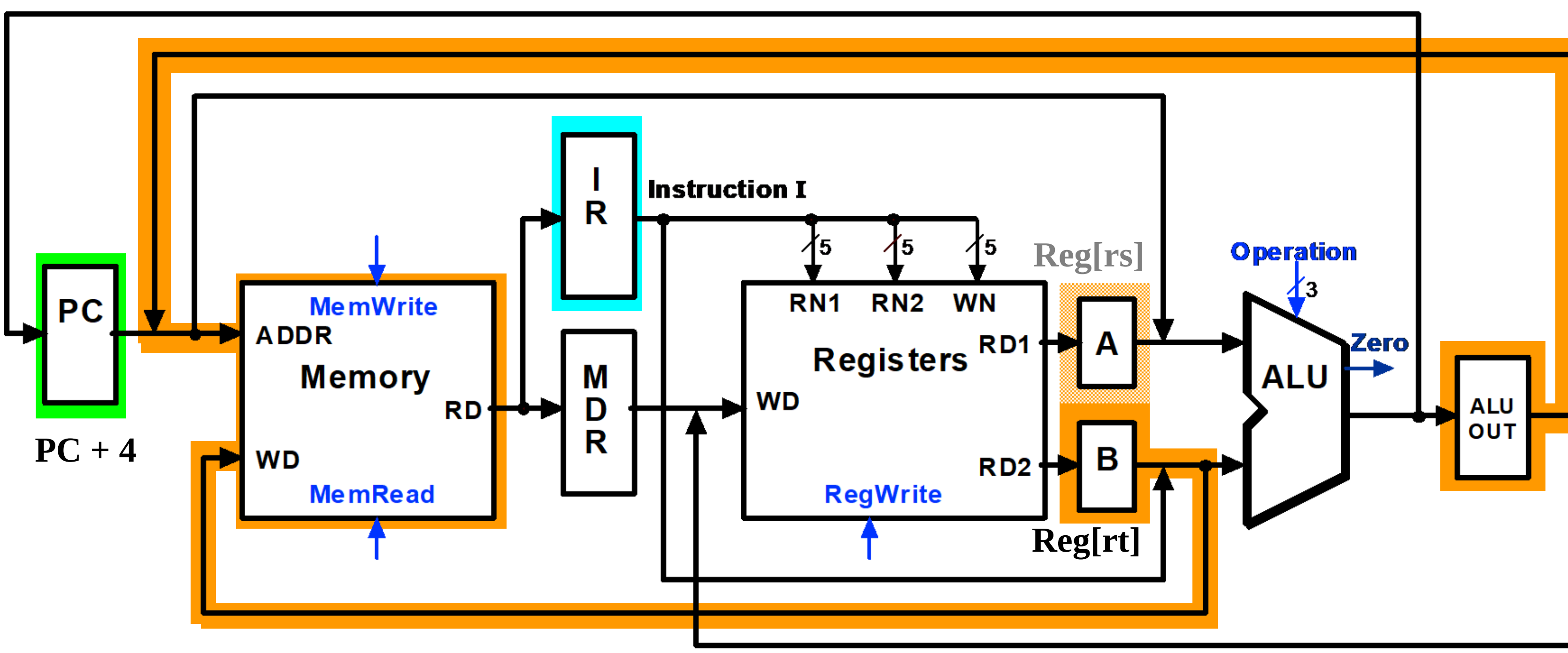
# Multicycle Execution Step (4): Memory Access - Read (`lw`)

```
MDR = Memory[ALUOut];
```



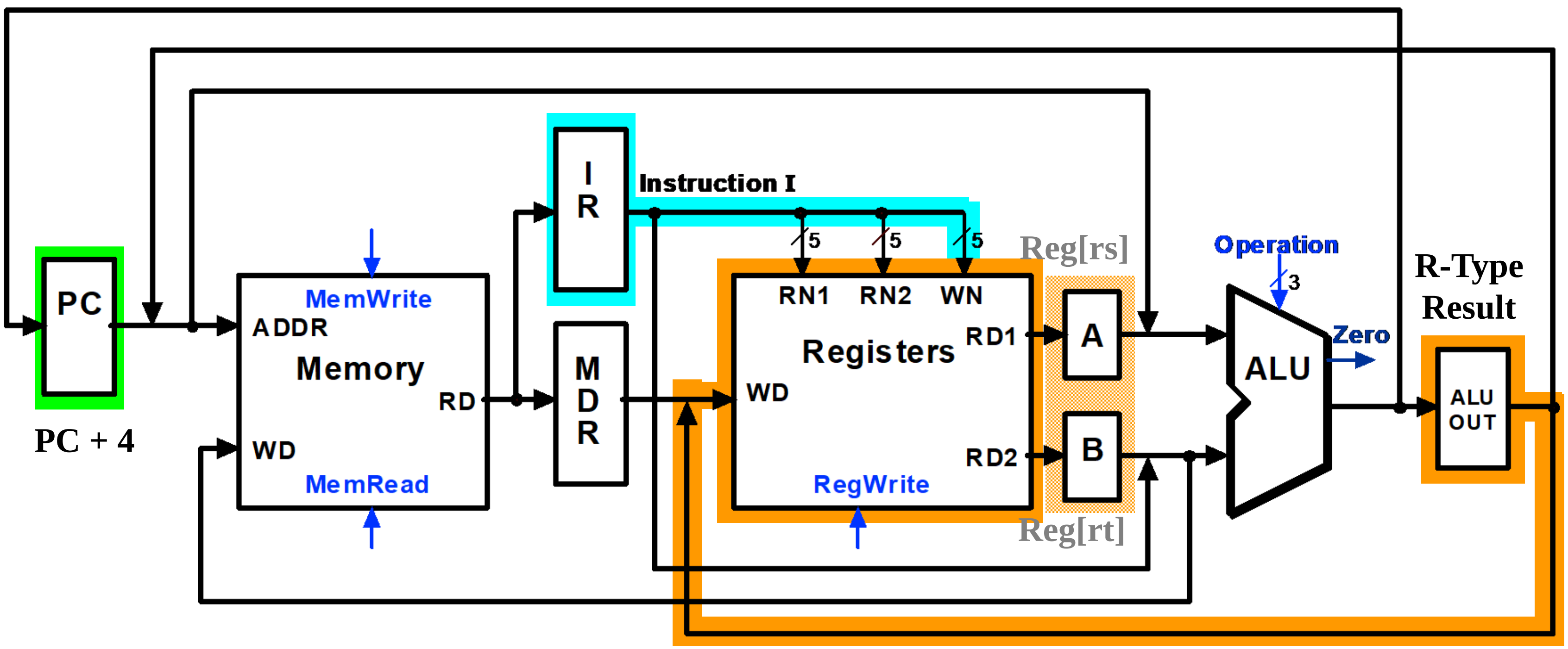
# Multicycle Execution Step (4): Memory Access - Write (`sw`)

```
Memory[ALUOut] = B;
```



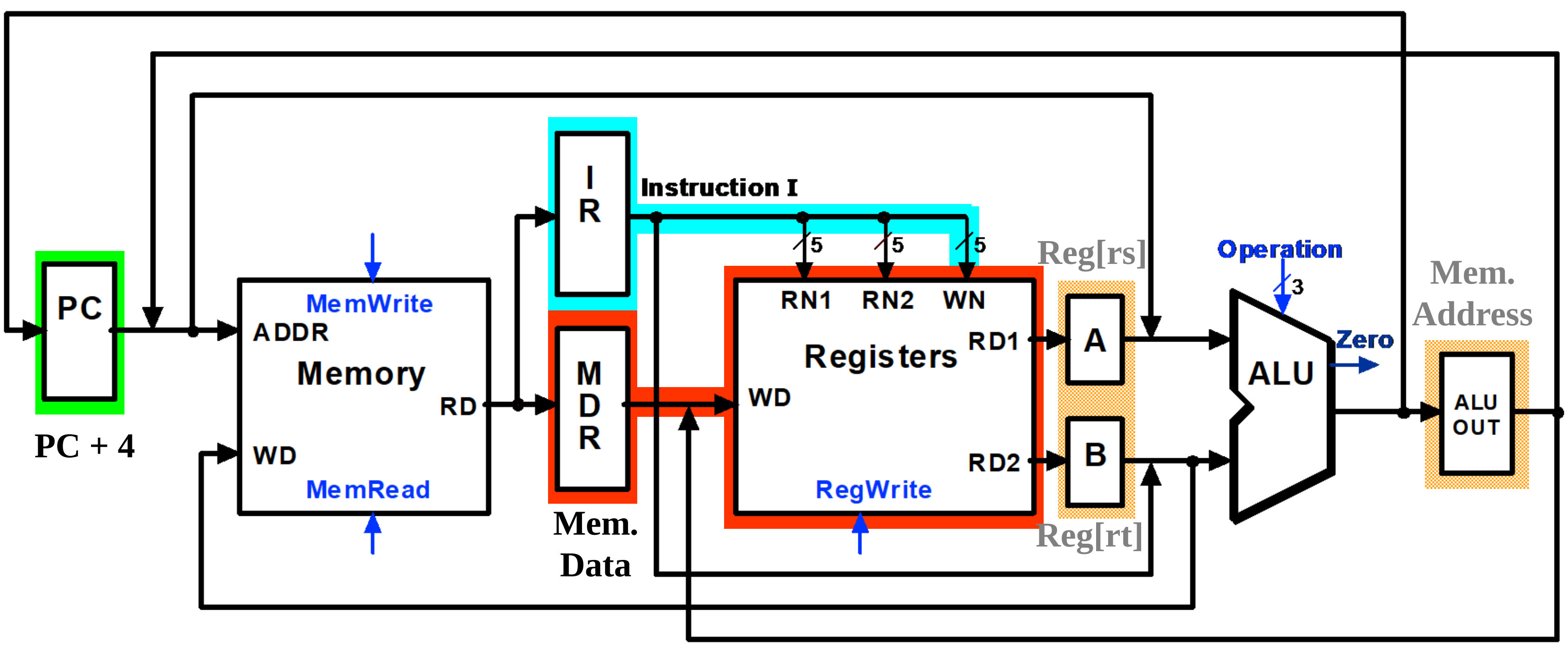
# Multicycle Execution Step (4): ALU Instruction (R-Type)

`Reg[IR[15:11]] = ALUOUT`



# Multicycle Execution Step (5): Memory Read Completion (`lw`)

```
Reg[IR[20-16]] = MDR;
```



# Simple Questions

- *How many cycles will it take to execute this code?*

```
            lw $t2, 0($t3)
            lw $t3, 4($t3)
            beq $t2, $t3, Label #assume not equal
            add $t5, $t2, $t3
            sw $t5, 8($t3)
Label:      ...
```

- *What is going on during the 8th cycle of execution?*

**Clock time-line**

- *In what cycle does the actual addition of $t2 and $t3 takes place?*

# Example: CPI in a multicycle CPU

- Assume
  - the control design of the previous slide
  - An instruction mix of 22% *loads*, 11% *stores*, 49% *R-type operations*, 16% *branches*, and 2% *jumps*
- *What is the CPI assuming each step requires 1 clock cycle?*

- Solution:
  - Number of clock cycles from previous slide for each instruction class:
    - *loads* 5, *stores* 4, *R-type instructions* 4, *branches* 3, *jumps* 3
  - CPI = CPU clock cycles / instruction count

    $= \Sigma$ (instruction count$_{\text{class i}} \times$ CPI$_{\text{class i}}$) / instruction count

    $= \Sigma$ (instruction count$_{\text{class I}}$ / instruction count) $\times$ CPI$_{\text{class I}}$

    $= 0.22 \times 5 + 0.11 \times 4 + 0.49 \times 4 + 0.16 \times 3 + 0.02 \times 3$

    $= 4.04$

# Summary

- *Techniques described in this chapter to design datapaths and control are at the core of all modern computer architecture*
- Multicycle datapaths offer two great advantages over single-cycle
  - functional units can be reused within a single instruction if they are accessed in different cycles – reducing the need to replicate expensive logic
  - instructions with shorter execution paths can complete quicker by consuming fewer cycles
- Modern computers, in fact, take the multicycle paradigm to a higher level to achieve greater instruction throughput:
  - *pipelining* (next topic) where multiple instructions execute simultaneously by having cycles of different instructions overlap in the datapath
  - *the MIPS architecture was designed to be pipelined*