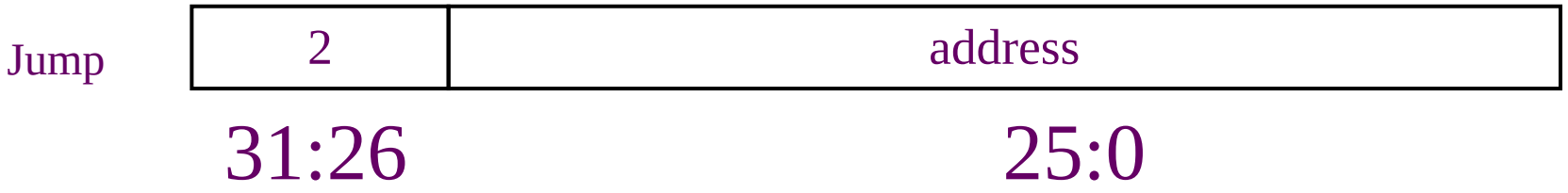
COMPUTER ORGANIZATION (IS F242)

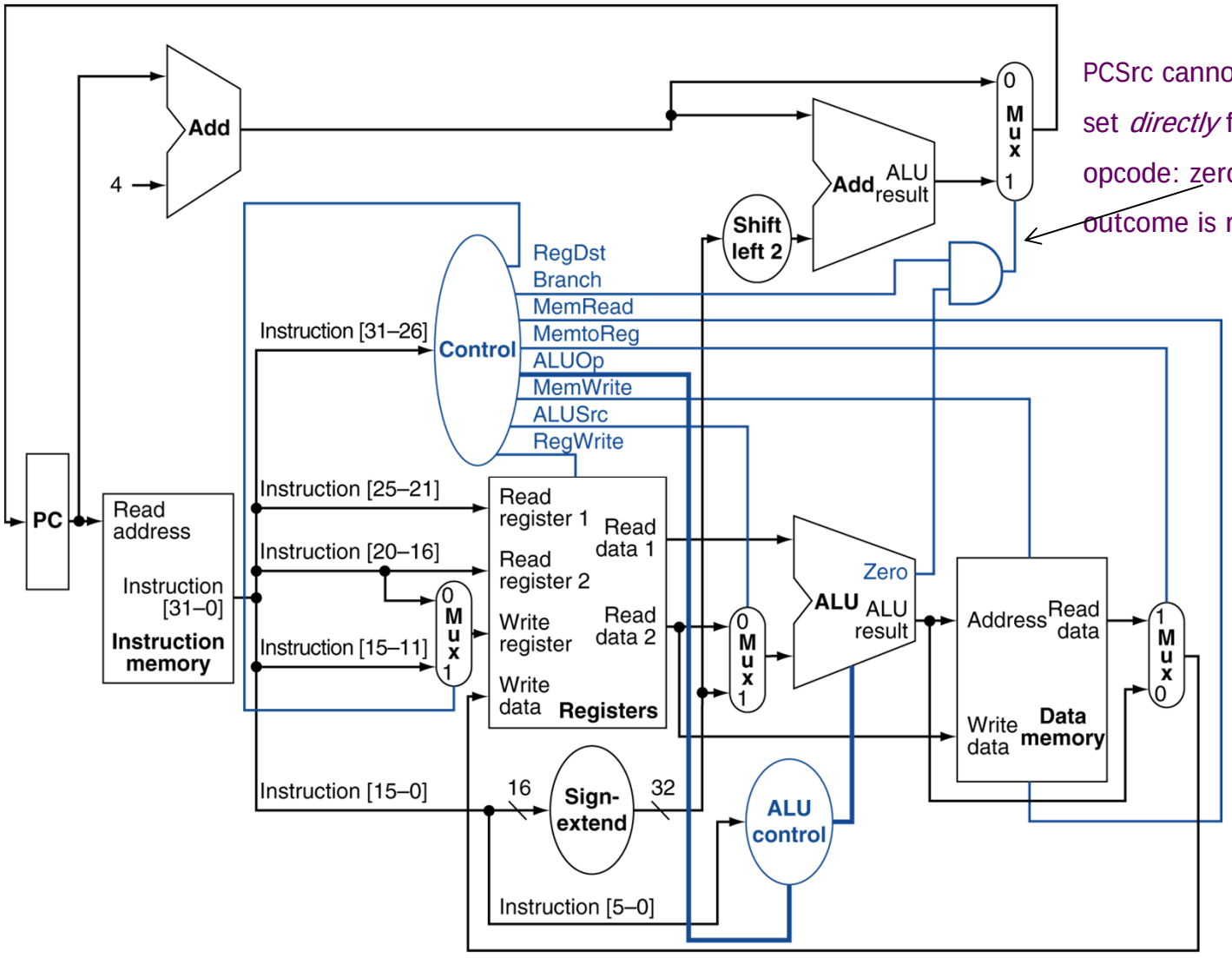
LECT 30_31: MIPS ARCHITECTURE

Implementing Jumps

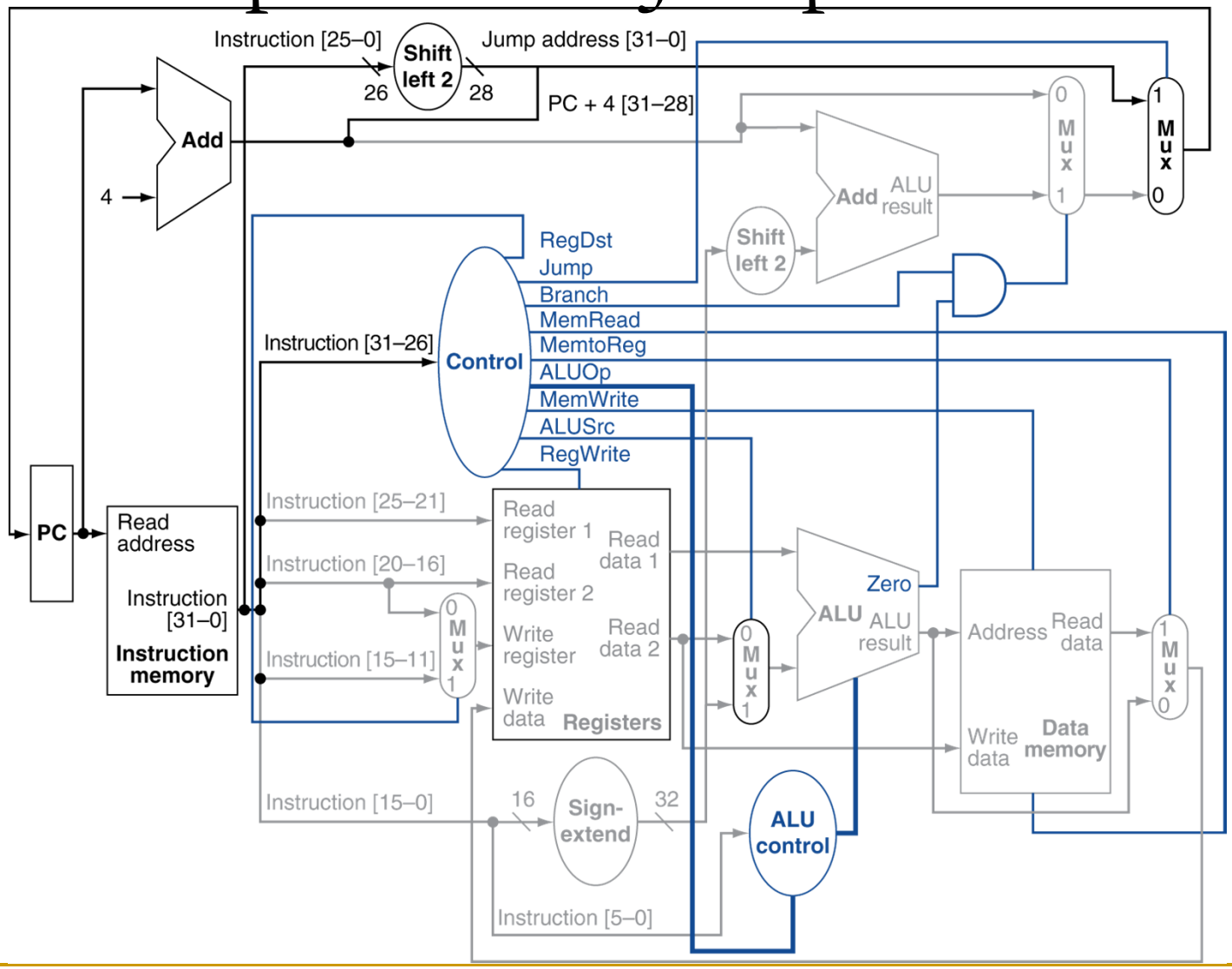


- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
- Need an extra control signal decoded from opcode

Datapath Without j



Datapath With Jumps Added



Single-cycle Implementation Notes

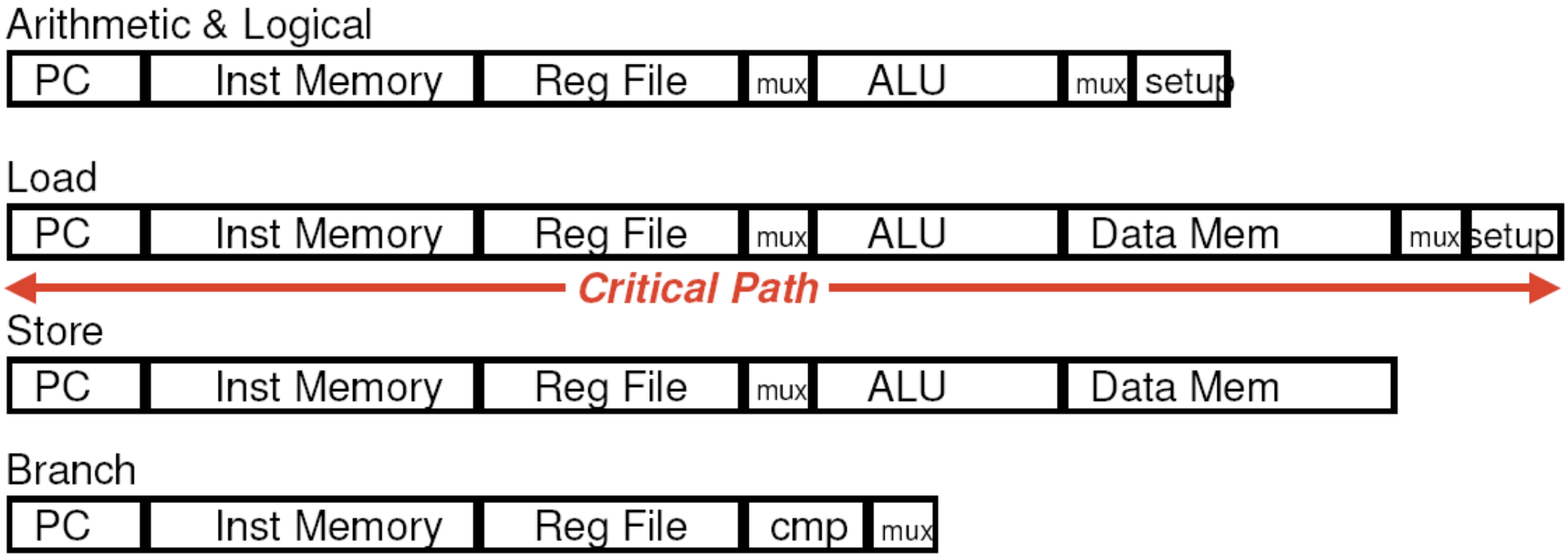
- *The steps are not really distinct* as each instruction completes in exactly one clock cycle – they simply indicate the sequence of data flowing through the datapath
- *The operation of the datapath during a cycle is purely combinational* – nothing is stored during a clock cycle
- Therefore, the machine is stable in a particular state at the start of a cycle and reaches a new stable state only at the end of the cycle

Performance Issues of Single Cycle

- Longest delay determines clock period
  - but several instructions could run in a shorter clock cycle: *waste of time*
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- CPI = 1
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
  - resources used more than once in the same cycle need to be duplicated
    - *waste of hardware and chip area*

Drawbacks of Single Cycle Datapath

- For a single cycle datapath CPI = 1
  - Critical path is the maximum length instruction's execution time (load instruction in MIPS).
  - Clock frequency will be very low



Drawbacks of Single Cycle Datapath

- Assume the following
  - Memory access = **2ns**
  - Register Read/Write = **1ns**
  - ALU function = **2ns**

Load	**8ns**
Add	**6ns**
Store	**7ns**
Branch	**5ns**

Single Cycle Each cycle is 8ns
Assume a program with a million instructions T = I * CPI * Cycle Time
= 10^6 * 1 * 8ns
= 8ms

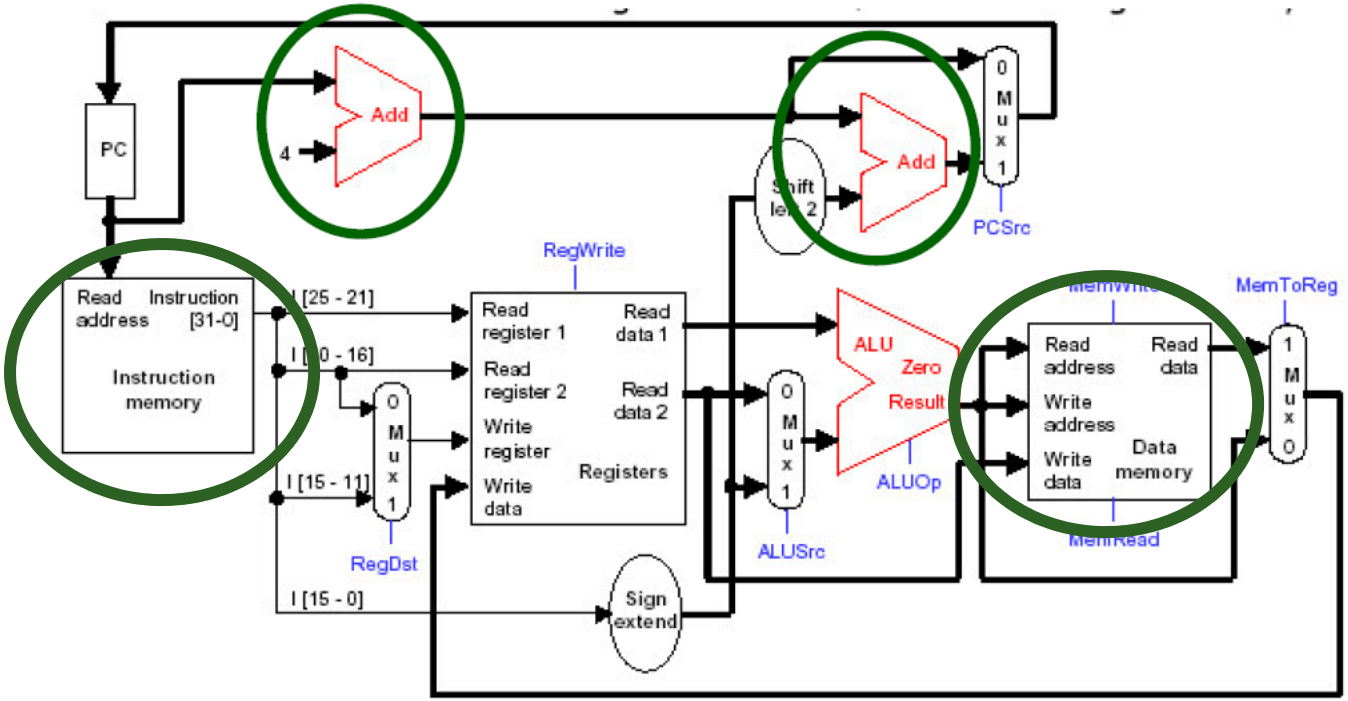
Arithmetic	**48%**
Loads	**22%**
Stores	**11%**
Branch	**19%**

Actual- Assume a program with a million instructions
Avg. CPI = 0.48*6 + 0.22*8 + 0.11*7 + 0.19*5 = 6.36ns
T = I * CPI * Cycle Time = 10^6 * 6.36 * 1 = 6.36 ms
Single Cycle Datapath is 8/6.36 = 1.26 times slower

Drawbacks of Single Cycle Datapath

- Long Cycle Time
- All instructions take as much time as the slowest
- Real Memory is not as nice as our idealized memory
- Can not pipeline (overlap) the processing of one instruction with its previous instruction

- What if the instructions are more complex than what we discussed?
  - Multiplication & Division, FP operations ….
- Requires extra hardware



Fixing the problem with single-cycle designs

- One solution: a variable-period clock with different cycle times for each instruction class
  - *unfeasible*, as implementing a variable-speed clock is technically difficult
- Another solution:
  - use a smaller cycle time…
  - …have different instructions take different numbers of cycles

    by breaking instructions into steps and fitting each step into one cycle
  - *feasible: multicyle approach*!

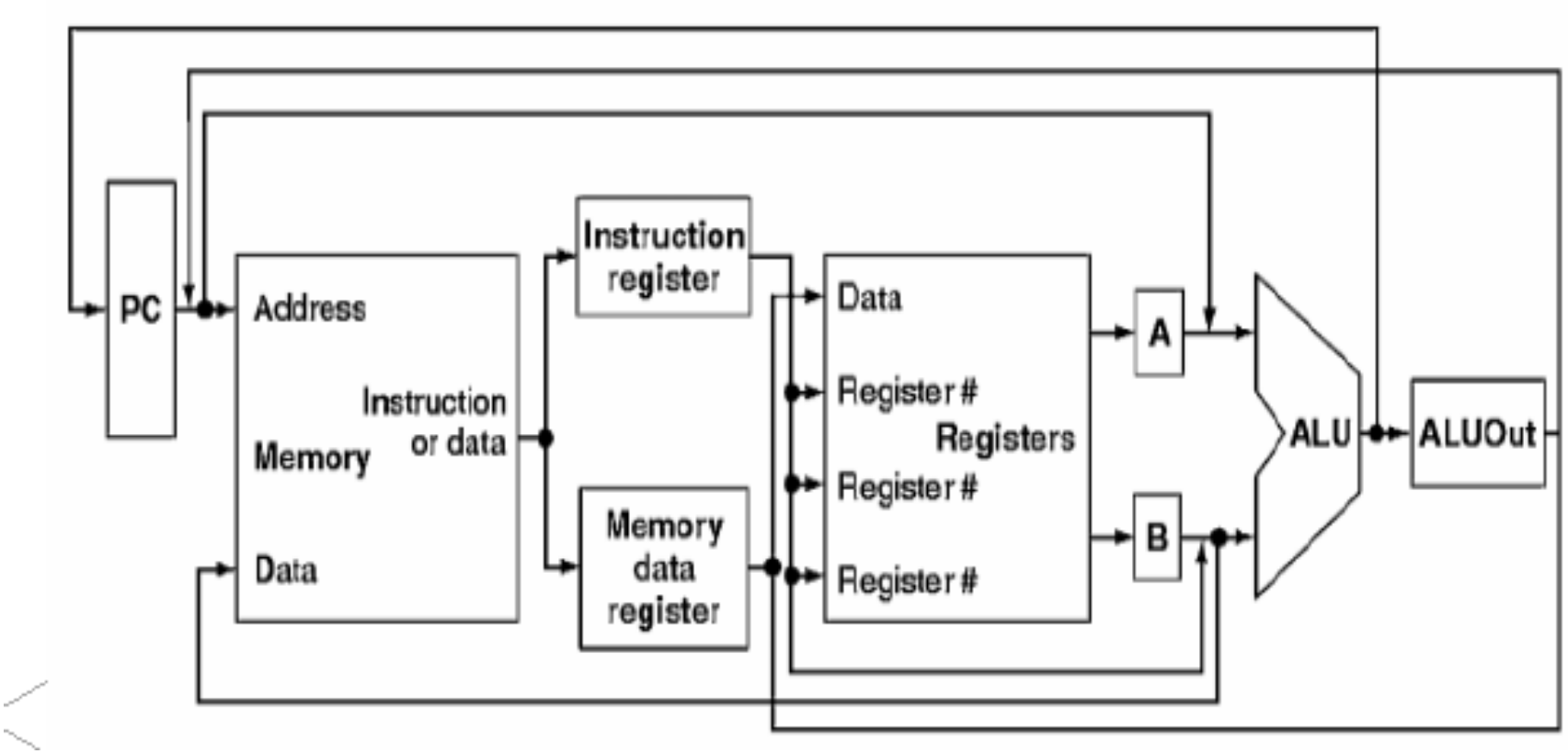
Multicycle Approach

- Break up the instructions into *steps*
  - each step takes one clock cycle
  - balance the amount of work to be done in each step/cycle so that they are about equal
  - restrict each cycle to use 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
- Between steps/cycles
  - At the end of one cycle store data to be used in *later cycles of the same* instruction
    - need to introduce additional *internal* (programmer-invisible) registers for this purpose
  - Data to be used in *later instructions* are stored in programmer-visible state elements: the register file, PC, memory

Requires Additional Storage elements

- Instruction register
- A & B Registers
- ALU Out Register
- Memory Data Register (MDR)

Multicycle implementation



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 *depending* on instruction type:
- Load writes back (instruction *completes*)

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

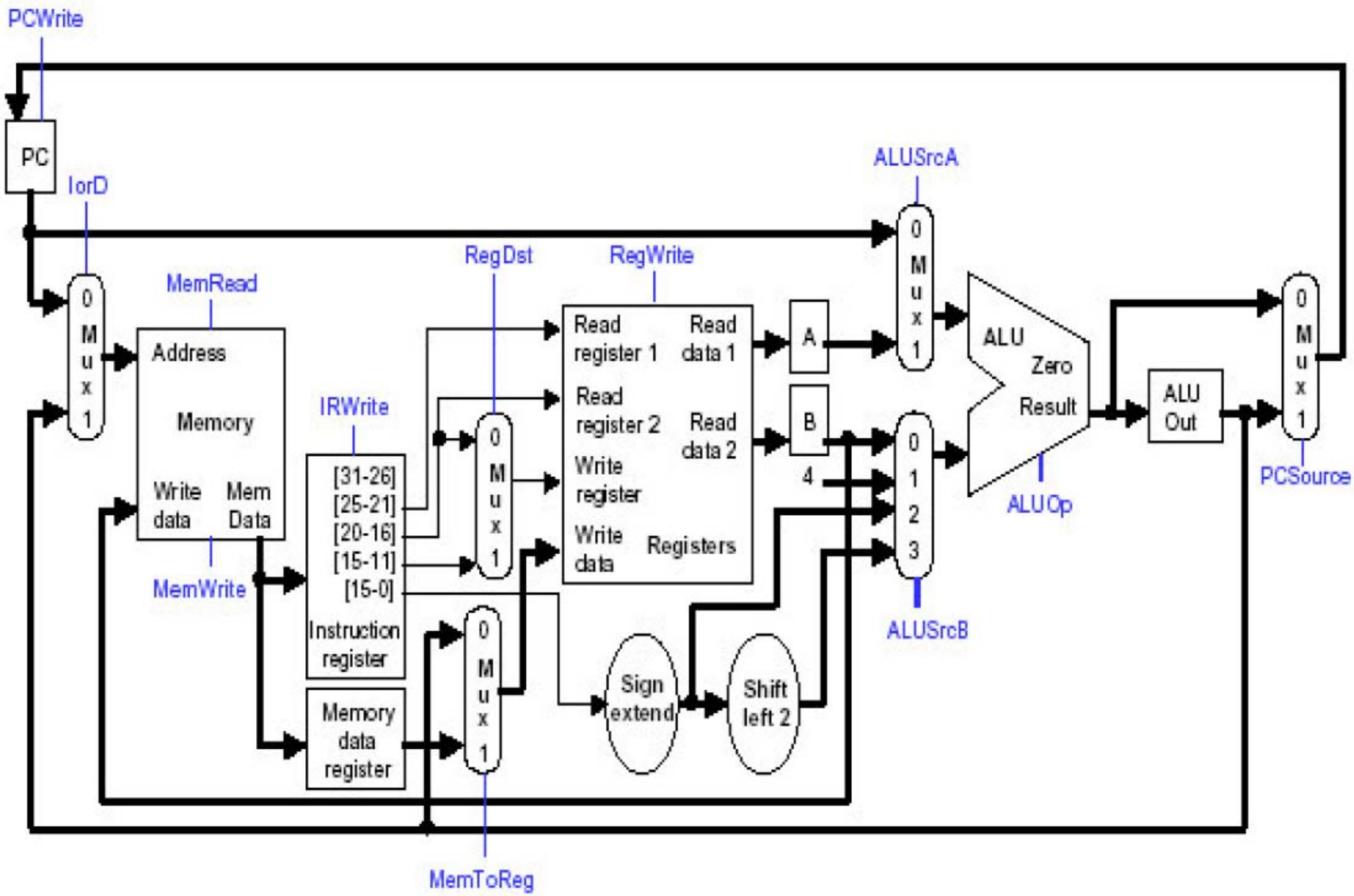
Important: There is no reason from a datapath (or control) point of view that Step 5 cannot be eliminated by performing

```
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.

Multi cycle Datapath

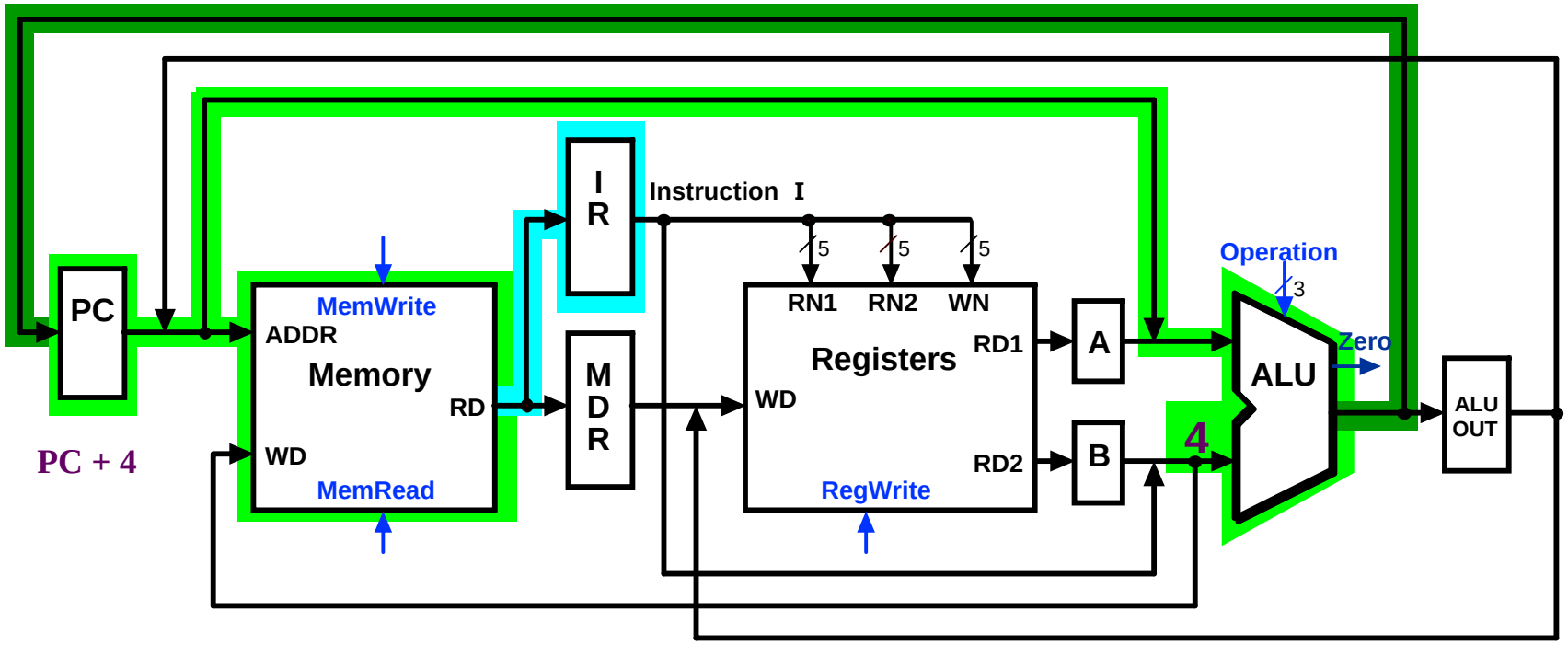


Summary of Instruction Execution

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]			
		PC = PC + 4			
2: ID	Instruction decode/ register fetch	A = Reg [IR[25-21]]			
		B = Reg [IR[20-16]]			
		ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
3: EX	Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + SEXT(IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28 ll (IR[25-0]<<2)
4: MEM	Memory access or R-type completion	Reg [IR[15-11]]= ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
5: WB	Memory read completion		Load: Reg[IR[20-16]] = MDR		

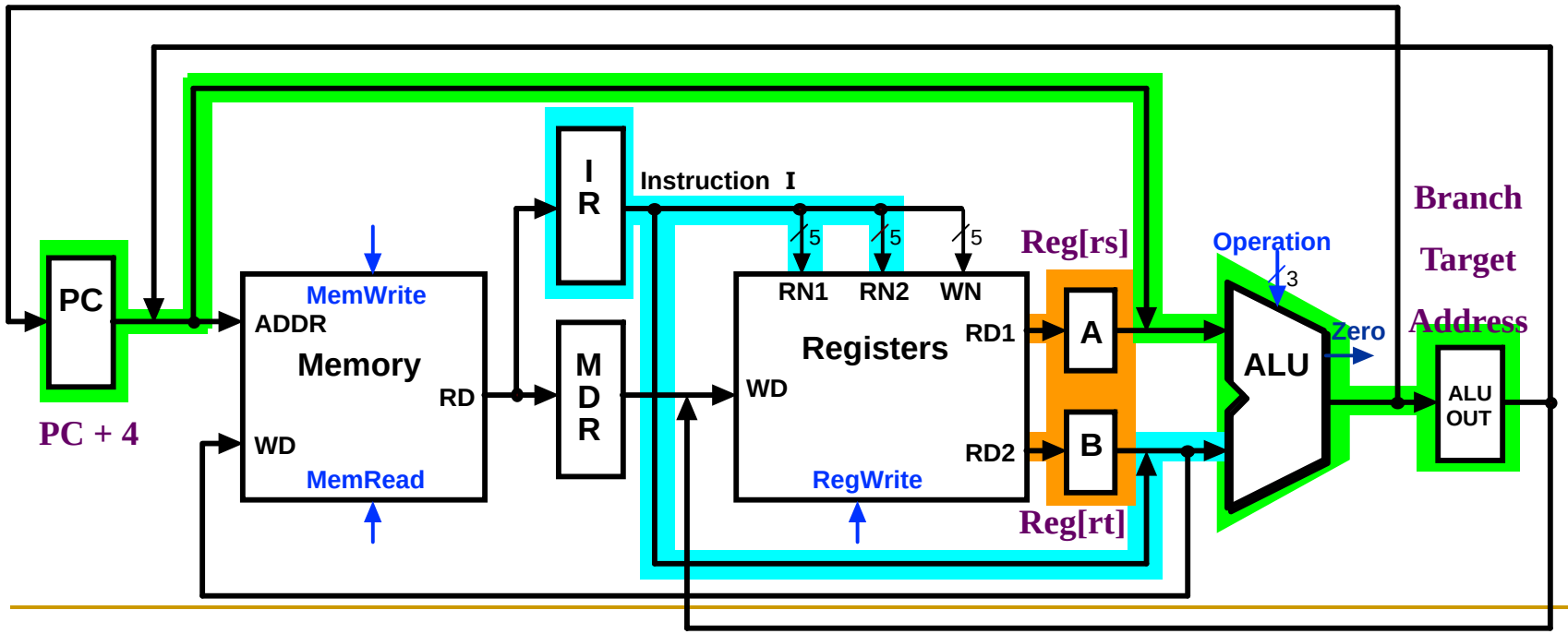
Multicycle Execution Step (1): Instruction Fetch

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



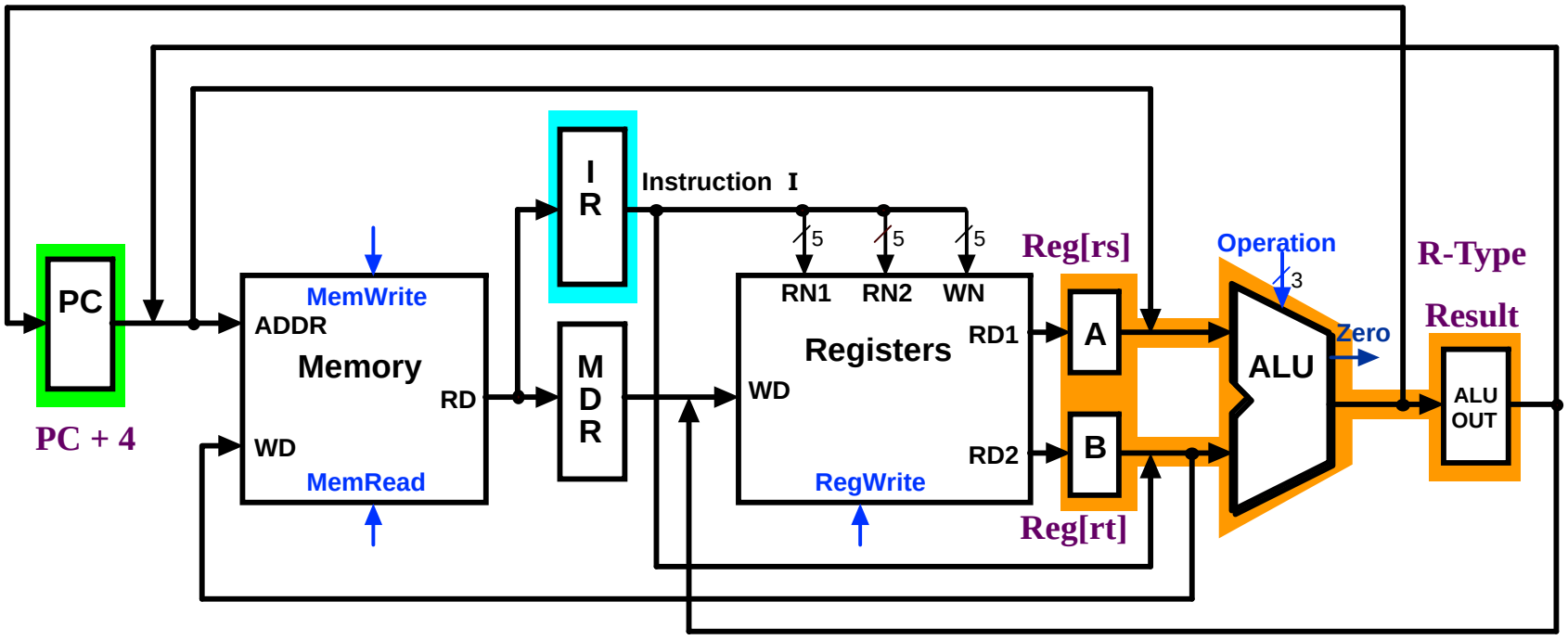
Multicycle Execution Step (2): Instruction Decode & Register Fetch

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



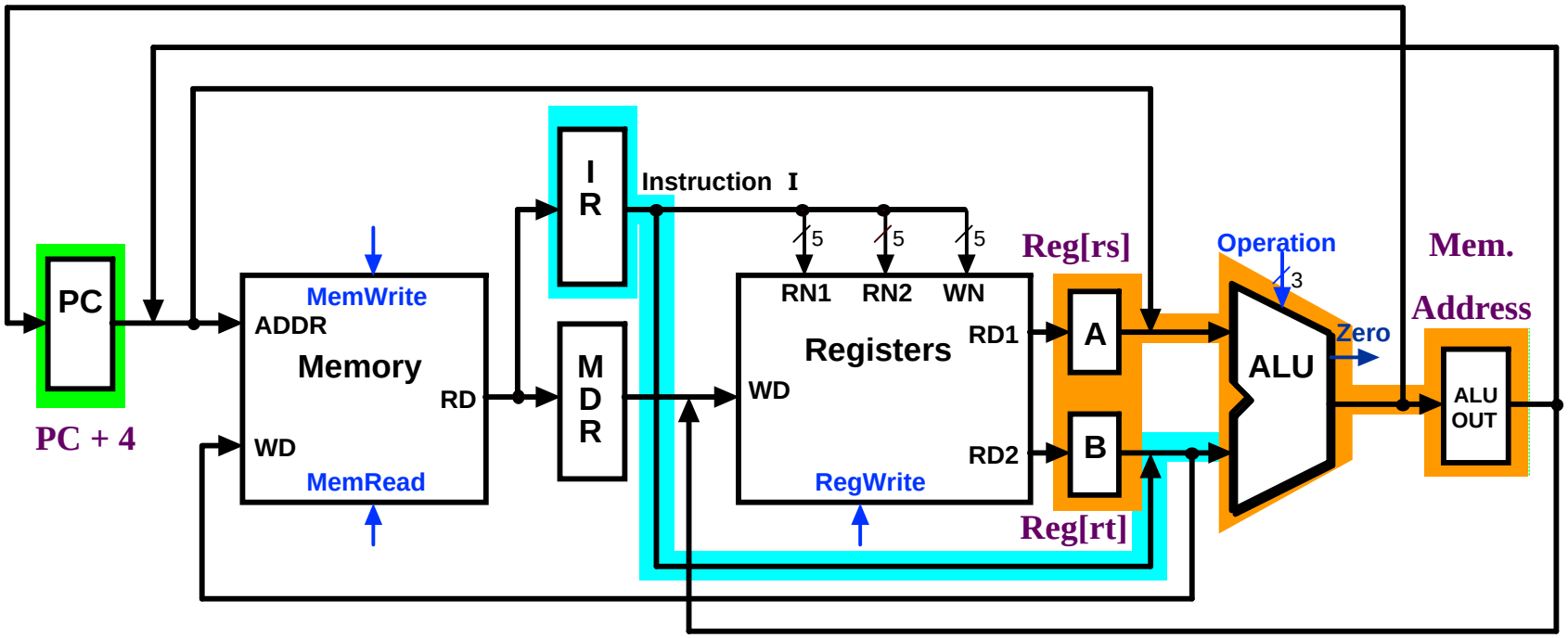
Multicycle Execution Step (3): ALU Instruction (R-Type)

```
ALUOut = A op B
```



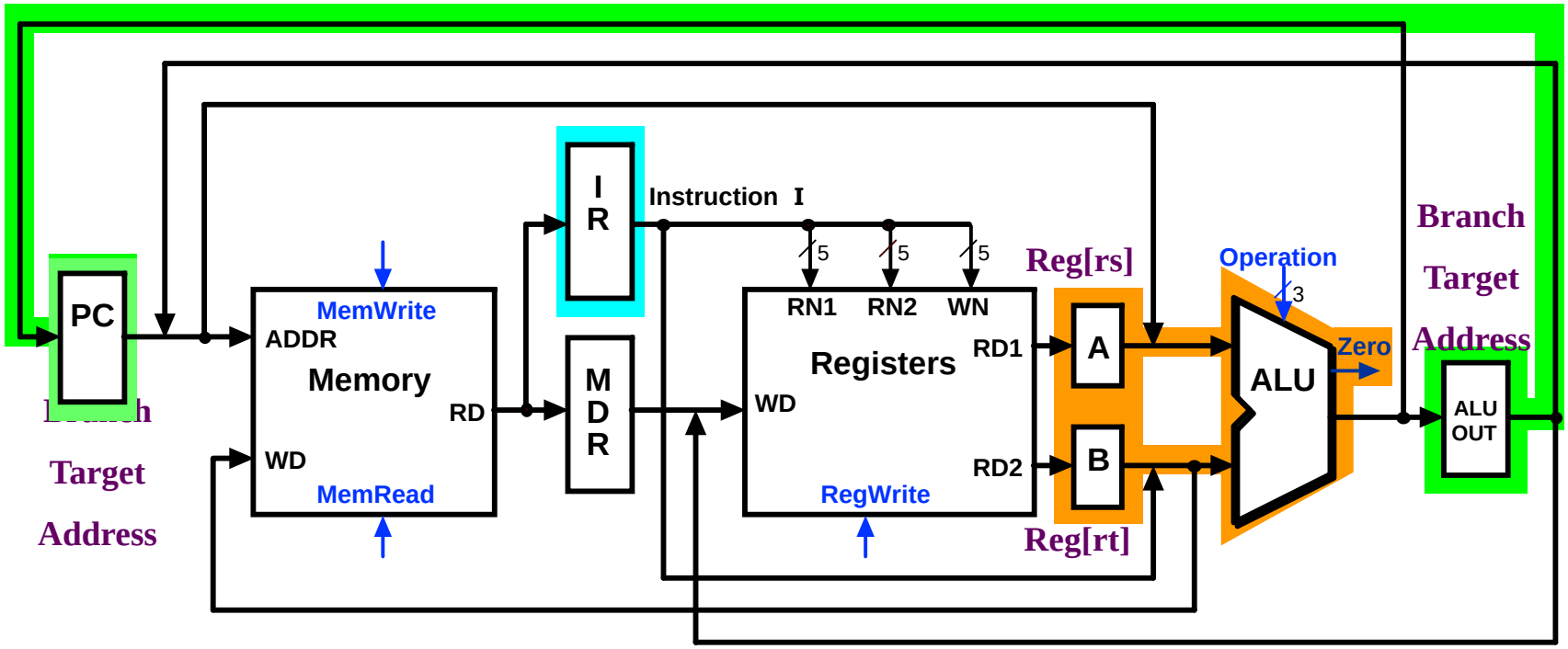
Multicycle Execution Step (3): Memory Reference Instructions

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



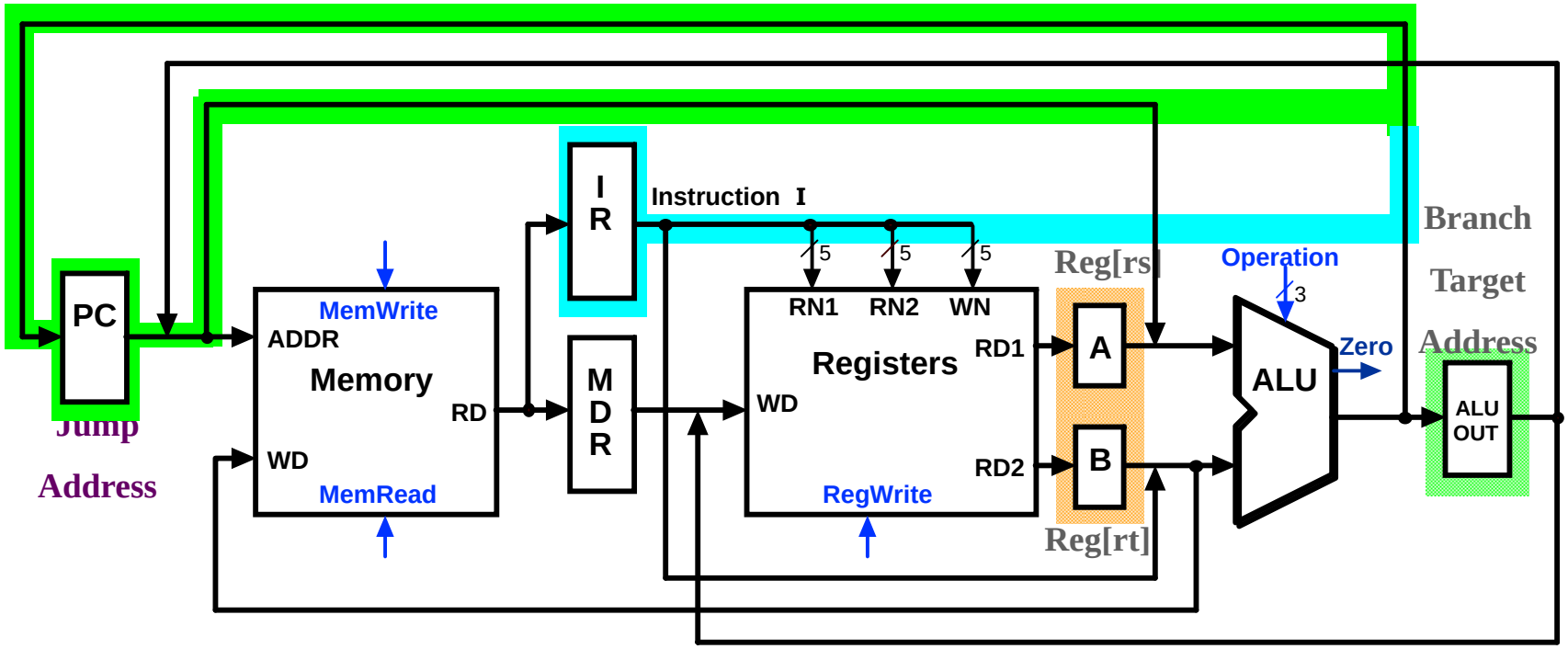
Multicycle Execution Step (3): Branch Instructions

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



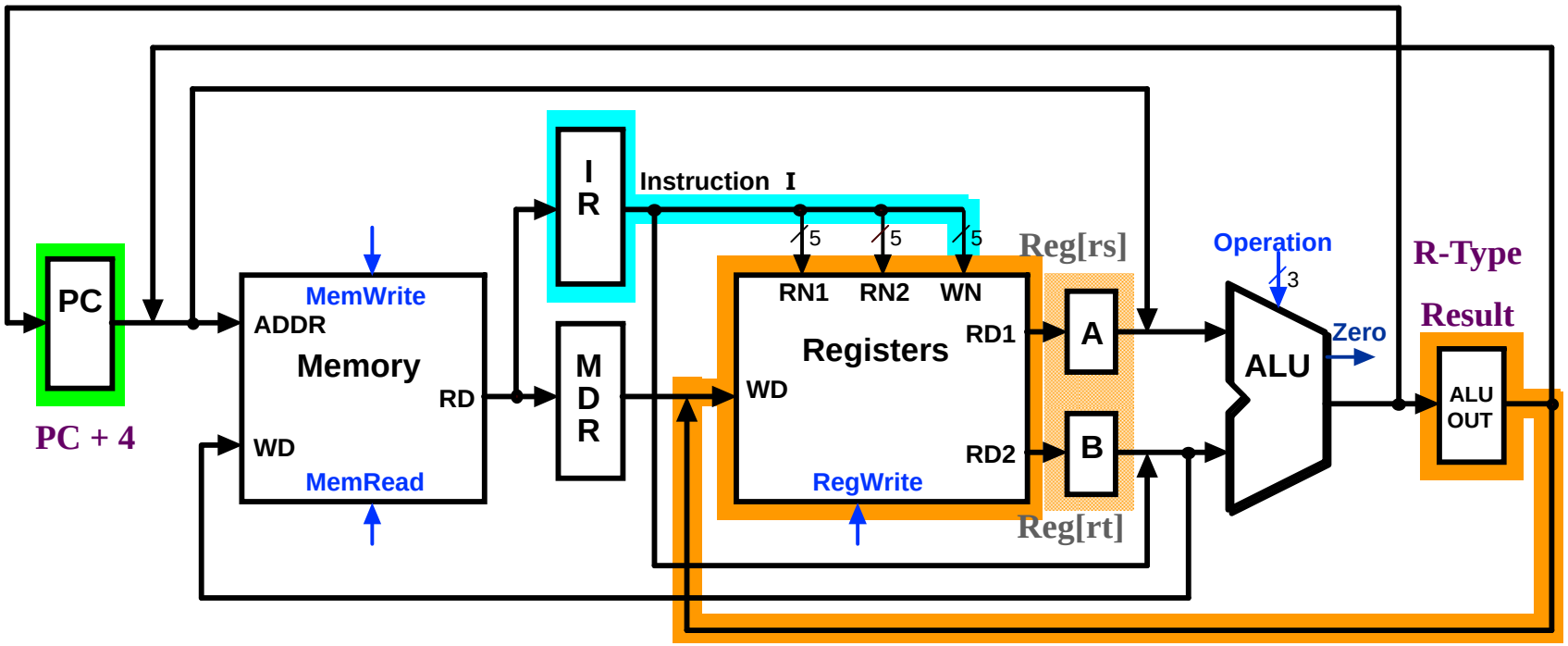
Multicycle Execution Step (3): Jump Instruction

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



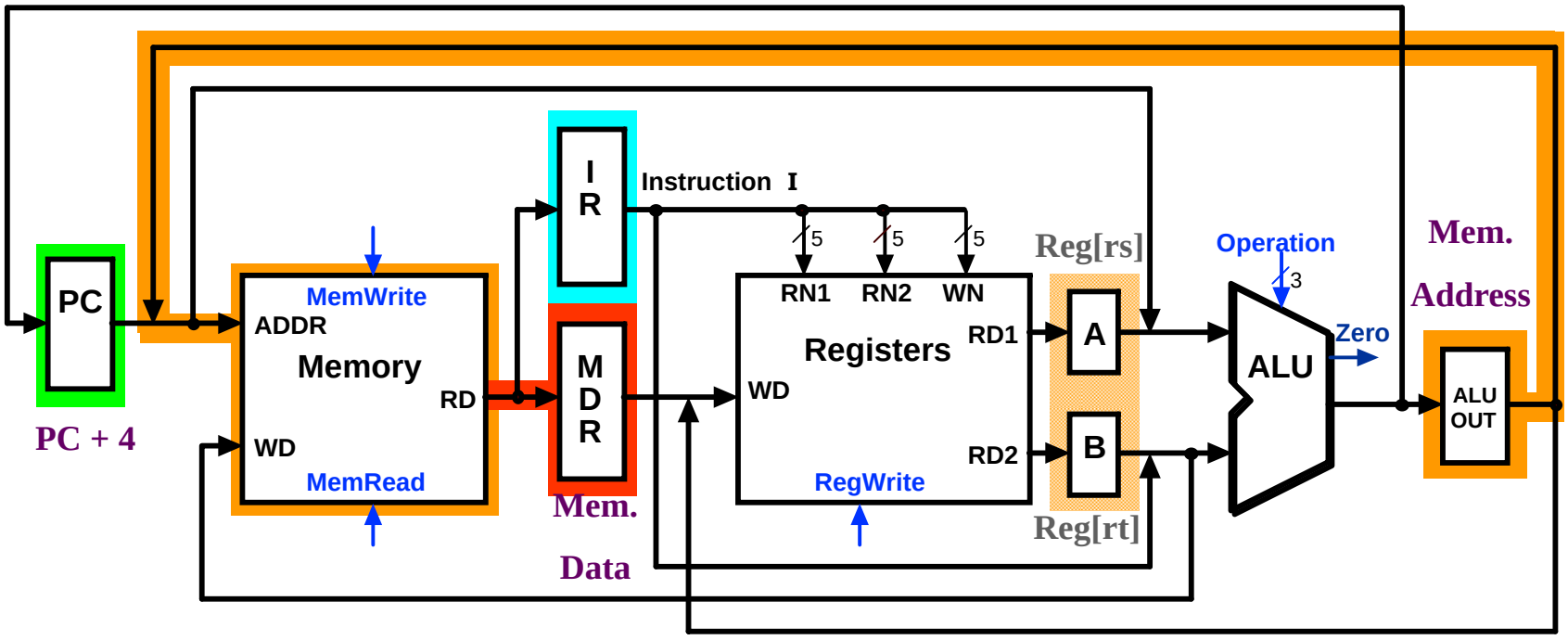
Multicycle Execution Step (4): ALU Instruction (R-Type)

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



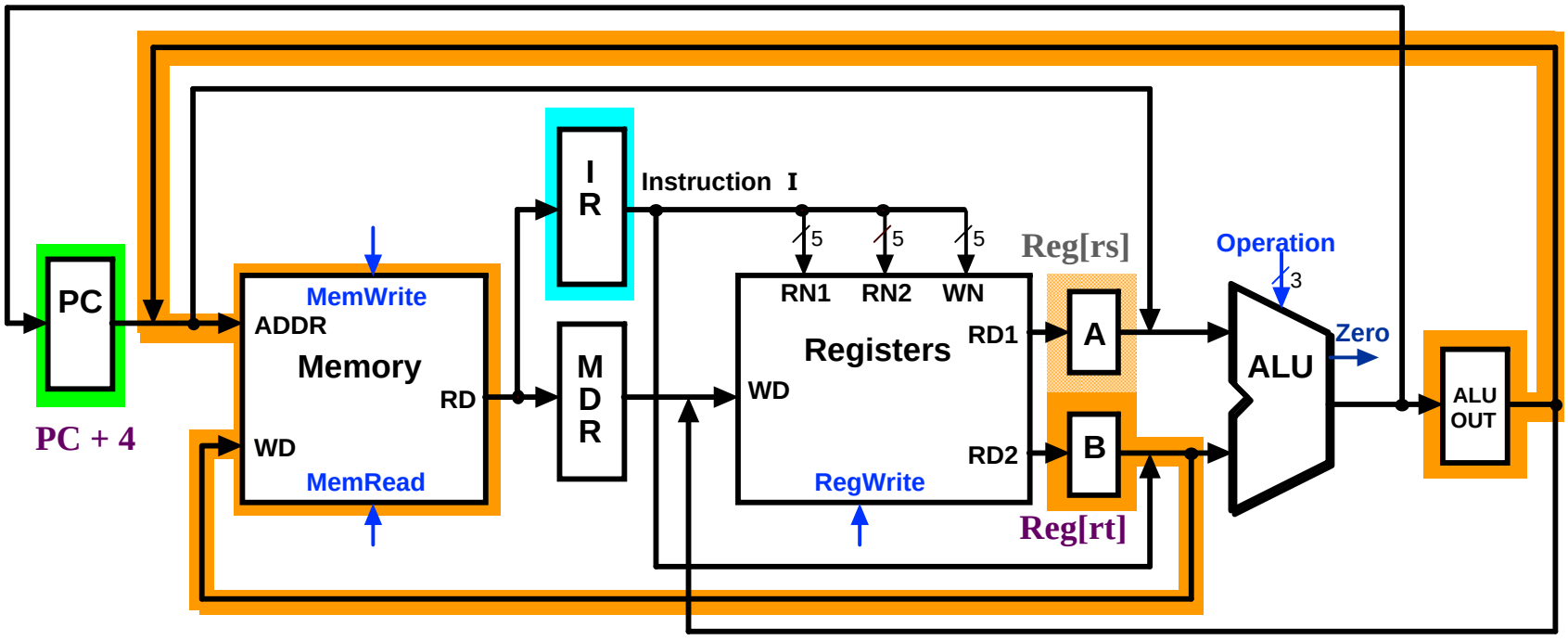
Multicycle Execution Step (4): Memory Access - Read (`lw`)

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



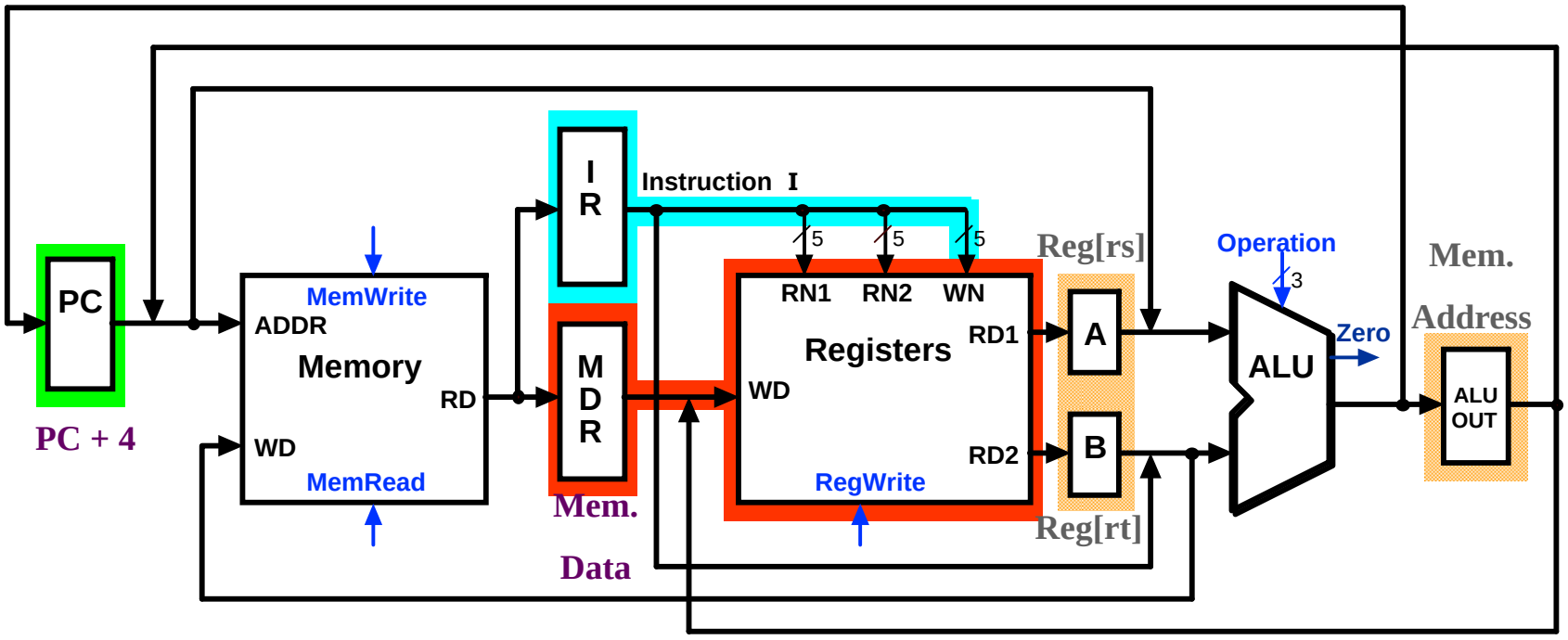
Multicycle Execution Step (4): Memory Access - Write (`SW`)

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



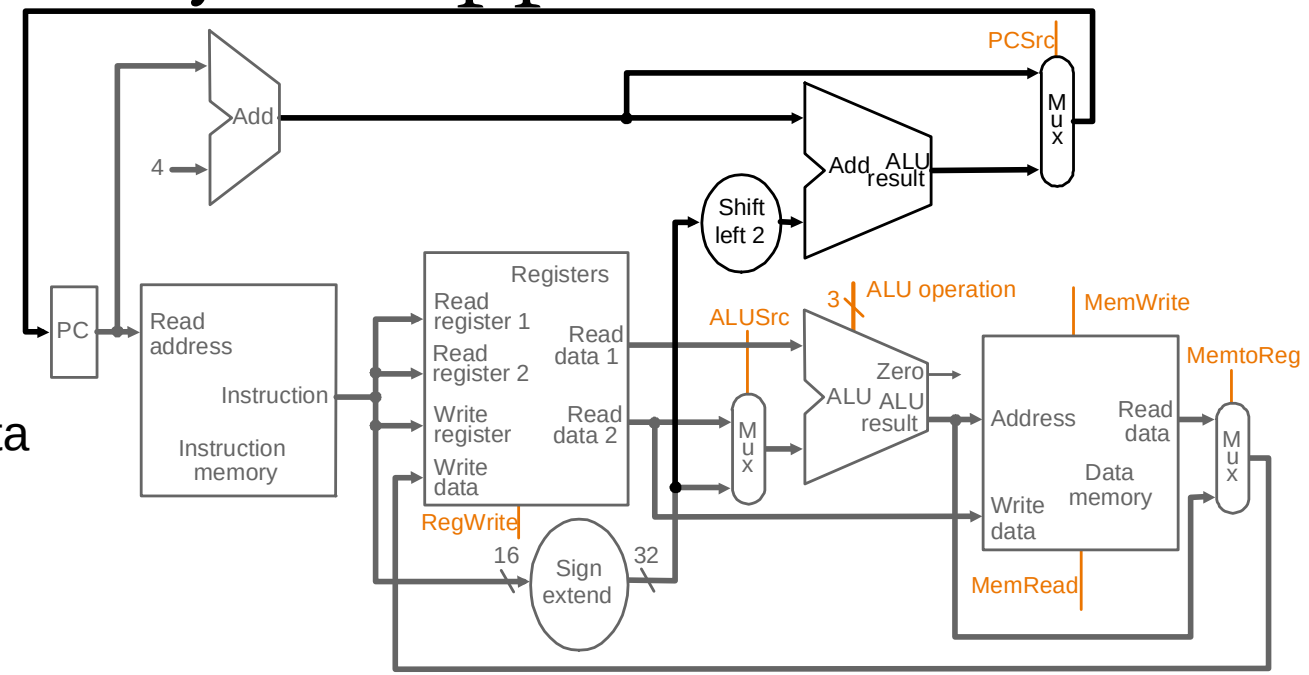
Multicycle Execution Step (5): Memory Read Completion (lw)

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

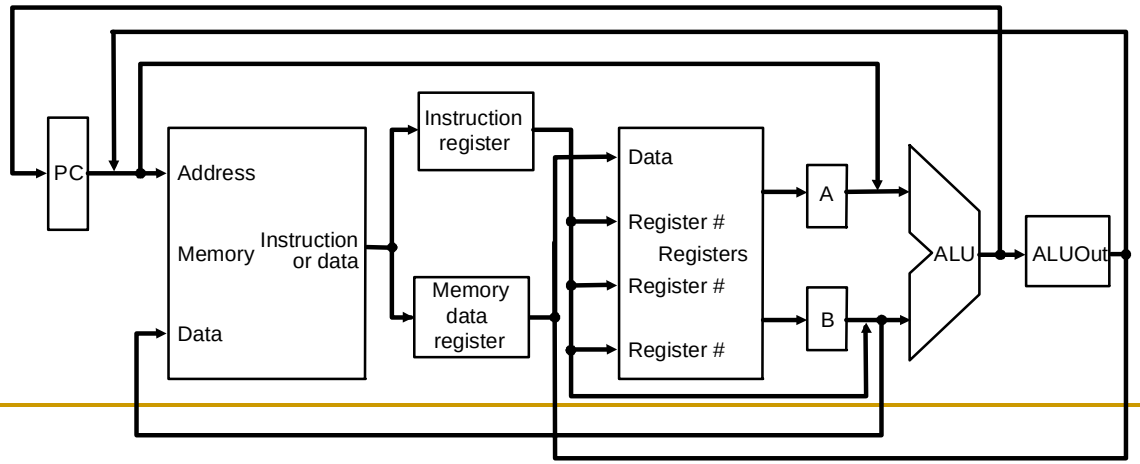


Multicycle Approach

- Note particularities of multicyle vs. single-diagrams
  - single memory for data and instructions
  - single ALU, no extra adders
  - extra registers to hold data between clock cycles

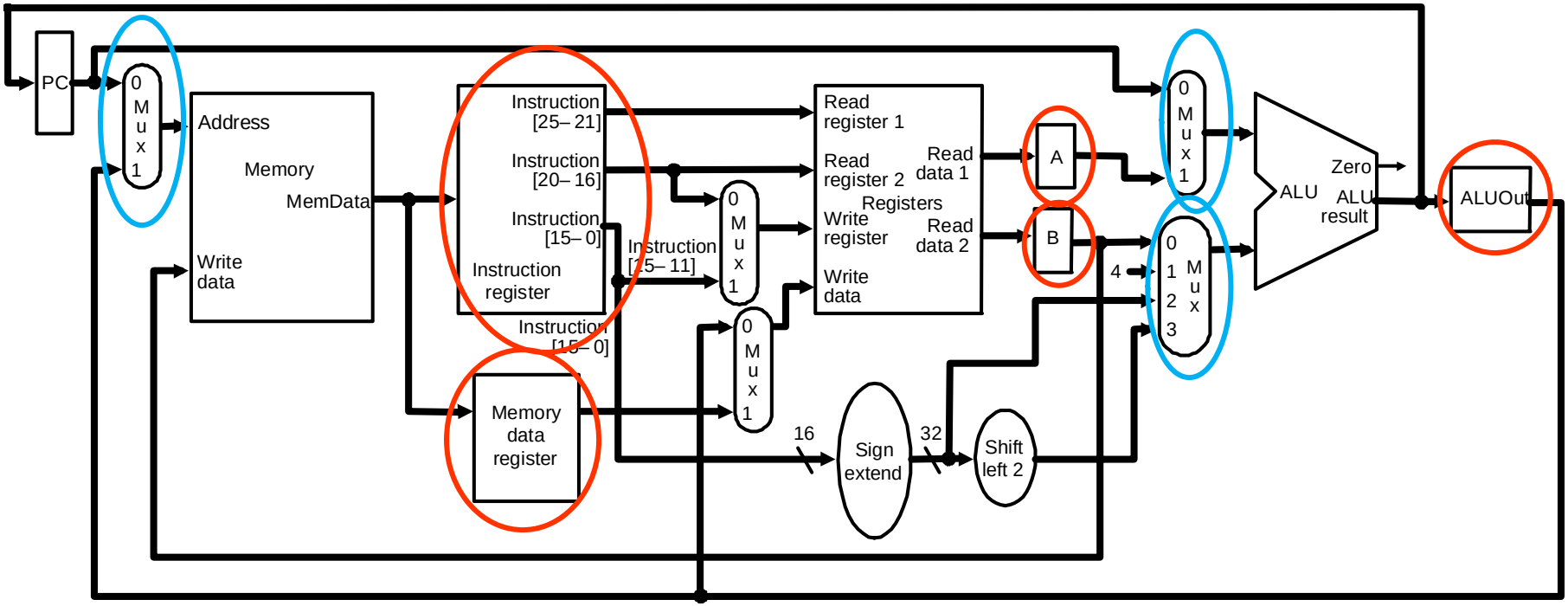


Single-cycle datapath



Multicycle datapath (high-level view)

Multicycle Datapath



Basic multicycle MIPS datapath handles R-type instructions and load/stores: new internal register in red ovals, new multiplexors in blue ovals