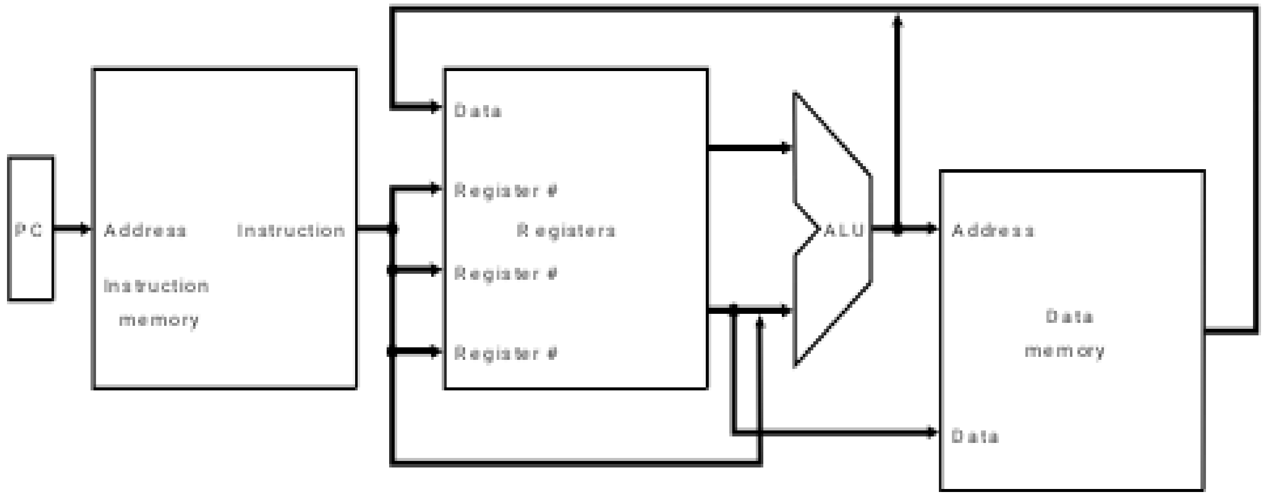
Implementing MIPS: the Fetch/Execute Cycle

- 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…



Single-Cycle Design: disadvantage

- Assuming fixed-period clock every instruction data-path uses one clock cycle implies:
  - CPI = 1
  - **cycle time determined by length of the longest instruction path (load)**
    - **but several instructions could run in a shorter clock cycle:** ***waste of time***
  - **resources used more than once in the same cycle need to be duplicated**
    - ***waste of hardware*** *and chip area*

Performance of Single-Cycle Machines

Assume that the operation times for the major functional units in this implementation are the following:

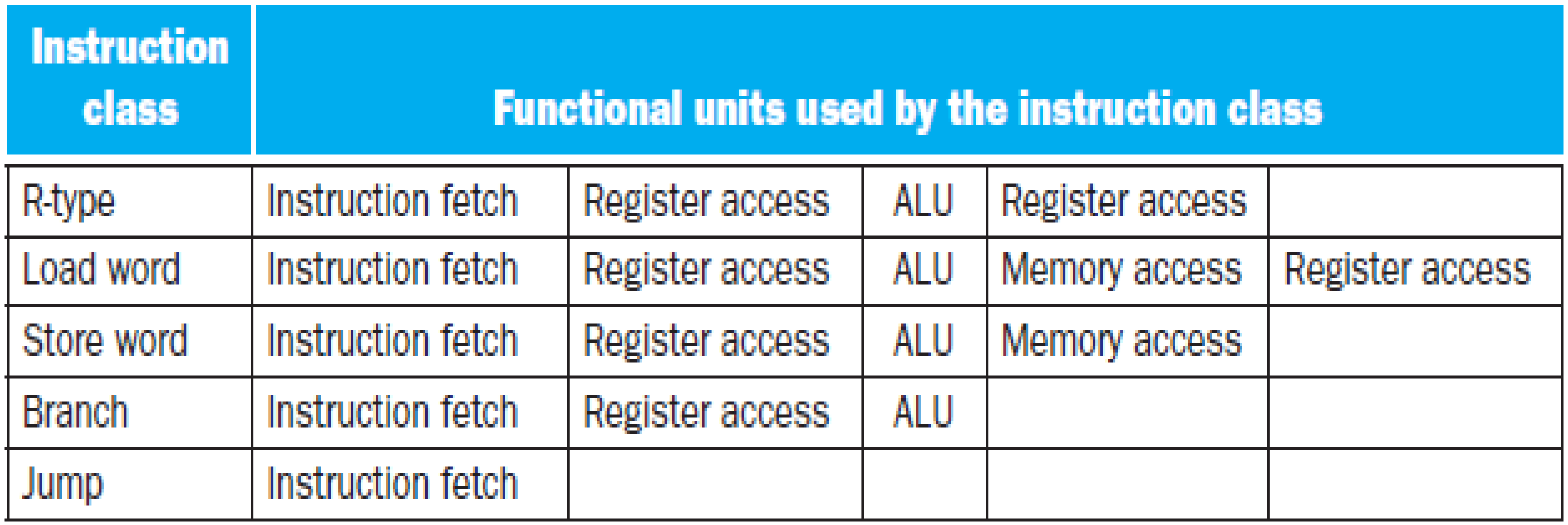
- Memory units: 200 picoseconds (ps)
- ALU and adders: 100 ps
- Register file (read or write): 50 ps

Assuming that the multiplexors, control unit, PC accesses, sign extension unit, and wires have no delay, which of the following implementations would be faster and by how much?

1. An implementation in which every instruction operates in 1 clock cycle of a fixed length.
2. An implementation where every instruction executes in 1 clock cycle using a variable-length clock, which for each instruction is only as long as it needs to be. (Such an approach is not terribly practical, but it will allow us to see what is being sacrificed when all the instructions must execute in a single clock of the same length.)

To compare the performance, assume the following instruction mix: 25% loads, 10% stores, 45% ALU instructions, 15% branches, and 5% jumps.

Critical path for different instruction classes



Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

The required length for each instruction

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

Fixing the problem with single-cycle designs

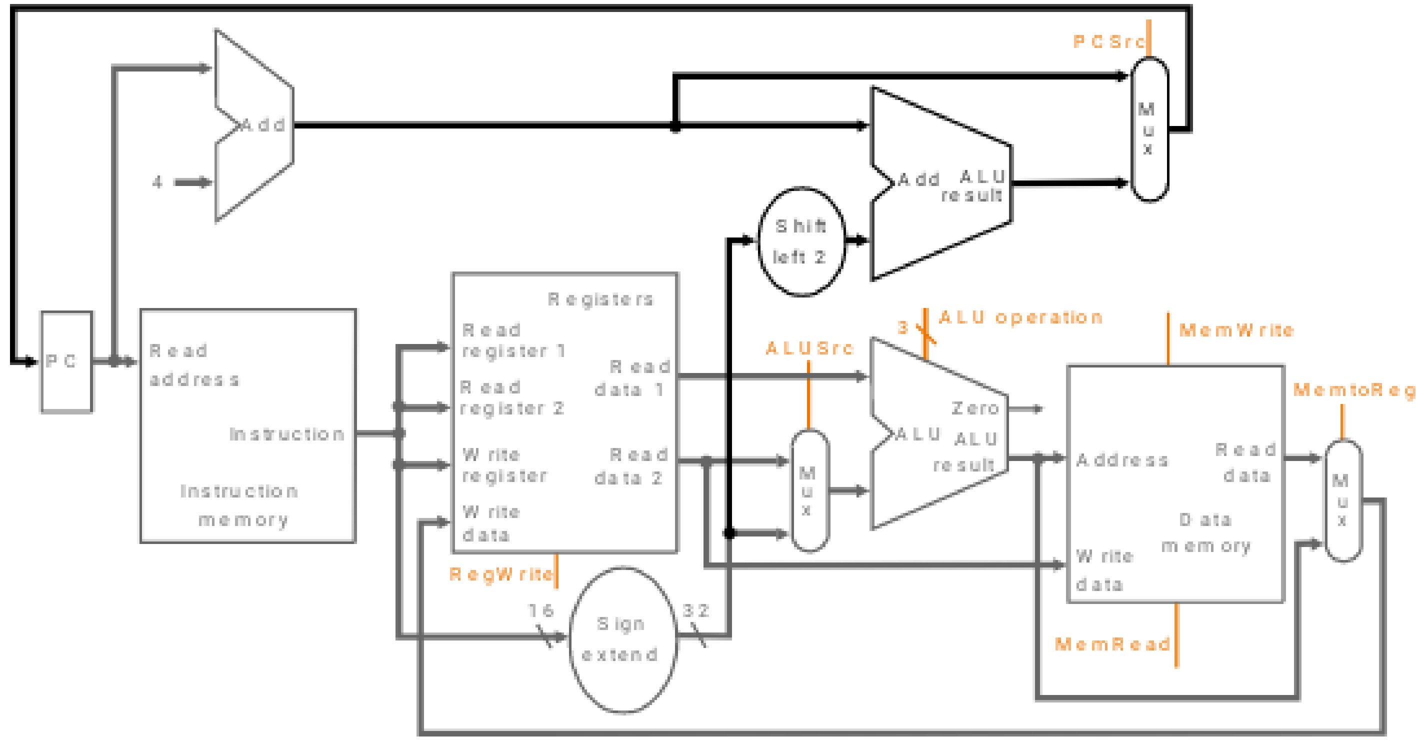
- One solution: a variable-period clock with different cycle times for each instruction class
  - ***unfeasible*** , as implementing a variable-speed clock is technically difficult
- Another solution:
  - use a smaller cycle time…
  - …have different instructions take different numbers of cycles

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

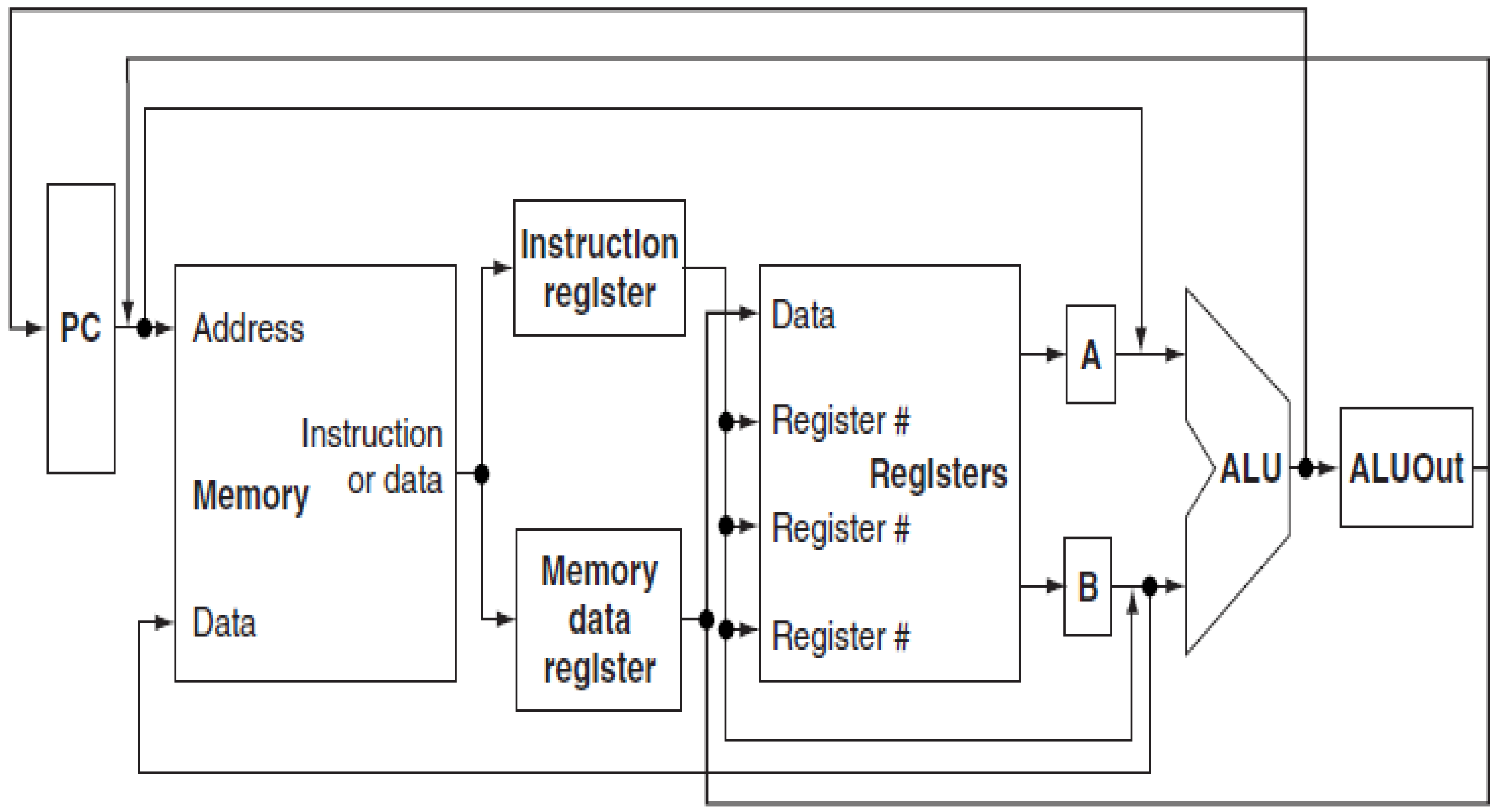
Multi cycle implementation

- **An implementation in which an instruction is executed in multiple clock cycle**
- Break up the instructions into *steps*
  - each step takes one clock cycle
  - It allows a functional unit to be used more than once per instruction, since it is used on different clock
  - This Sharing can reduce amount of hardware required

Single-cycle datapath



Multi-cycle : abstract implementation



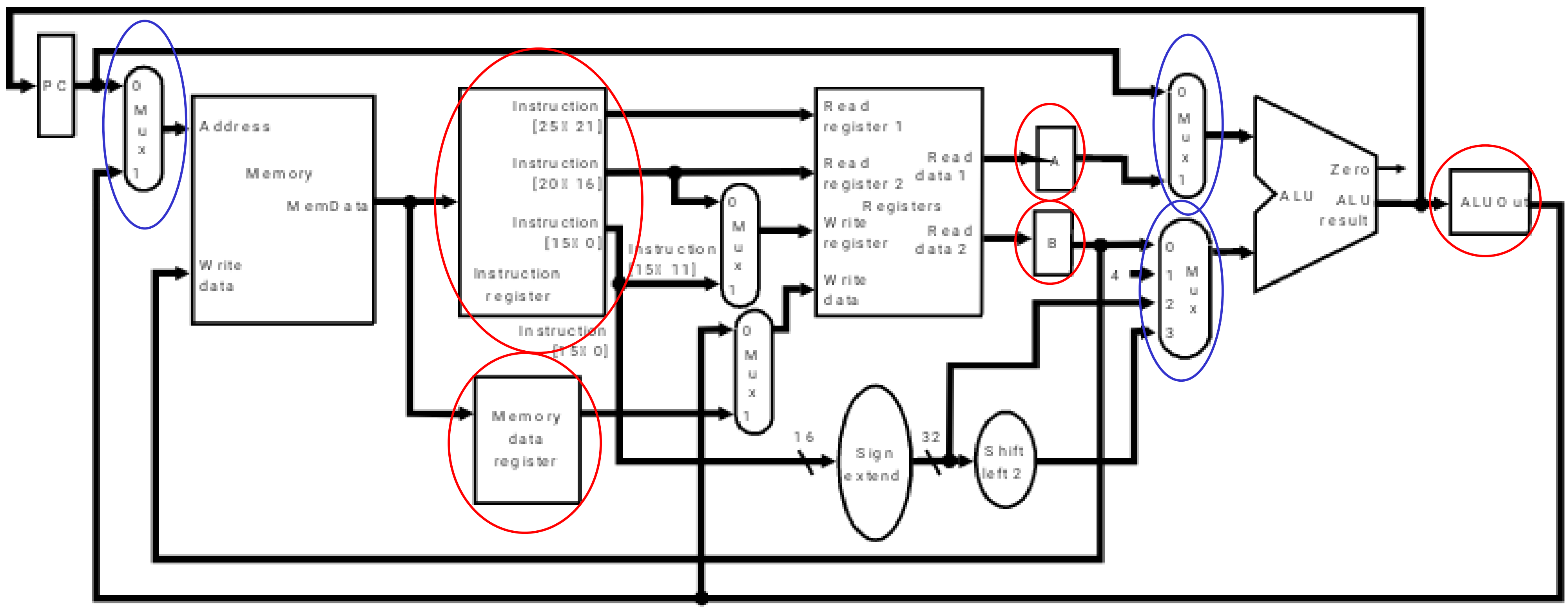
Multicycle Vs single cycle

- Note particularities of multicyle vs. single-cycle diagrams
  - single memory for data & instructions
  - single ALU, no extra adders
  - extra registers to hold data between clock cycles
- Data used by same instruction in a later cycle must be stored into additional registers

Internal registers

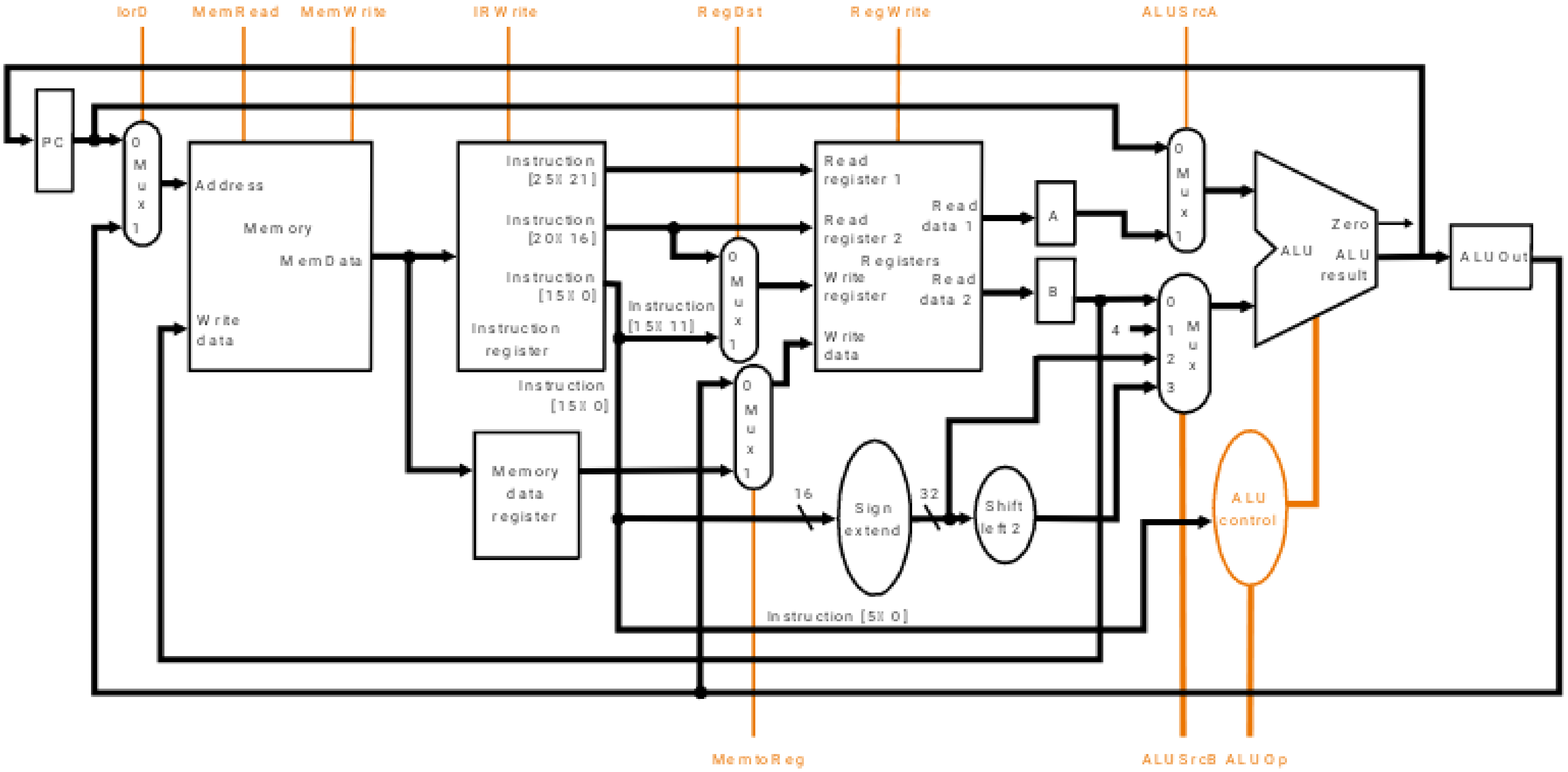
- Between steps/cycles
  - At the end of one cycle store data to be used in later cycles of the same instruction
    - need to **introduce additional internal (programmer-invisible) registers** for this purpose
- Temporary registers are
  - Instruction Registers (IR), Memory Data Register (MDR) to save output of memory
  - Registers A, B to hold register operands
  - ALUout register holds the output of ALU

Multicycle Datapath



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

Multicycle Datapath with Control I

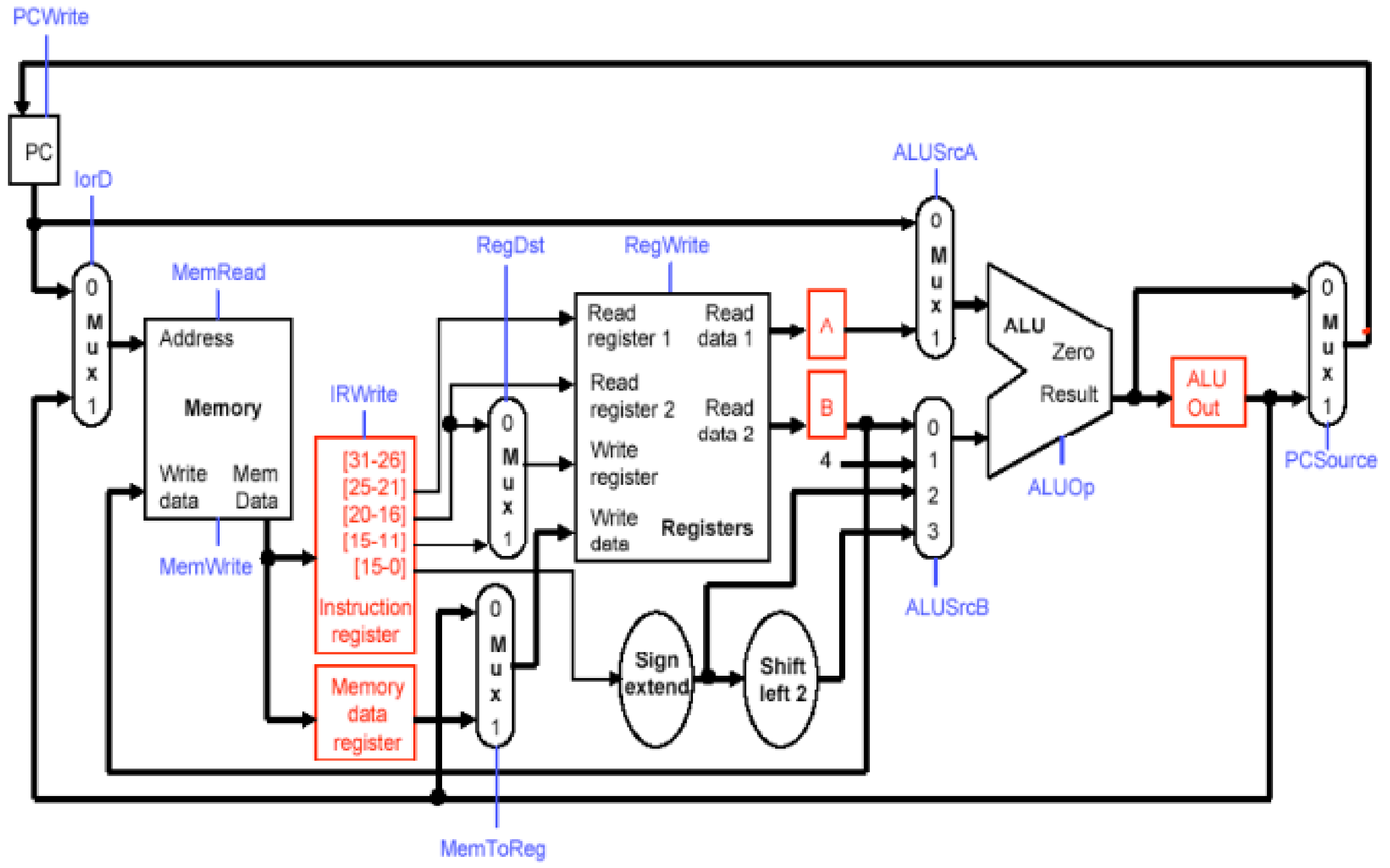


… with control lines and the ALU control block added – *not all* control lines are shown

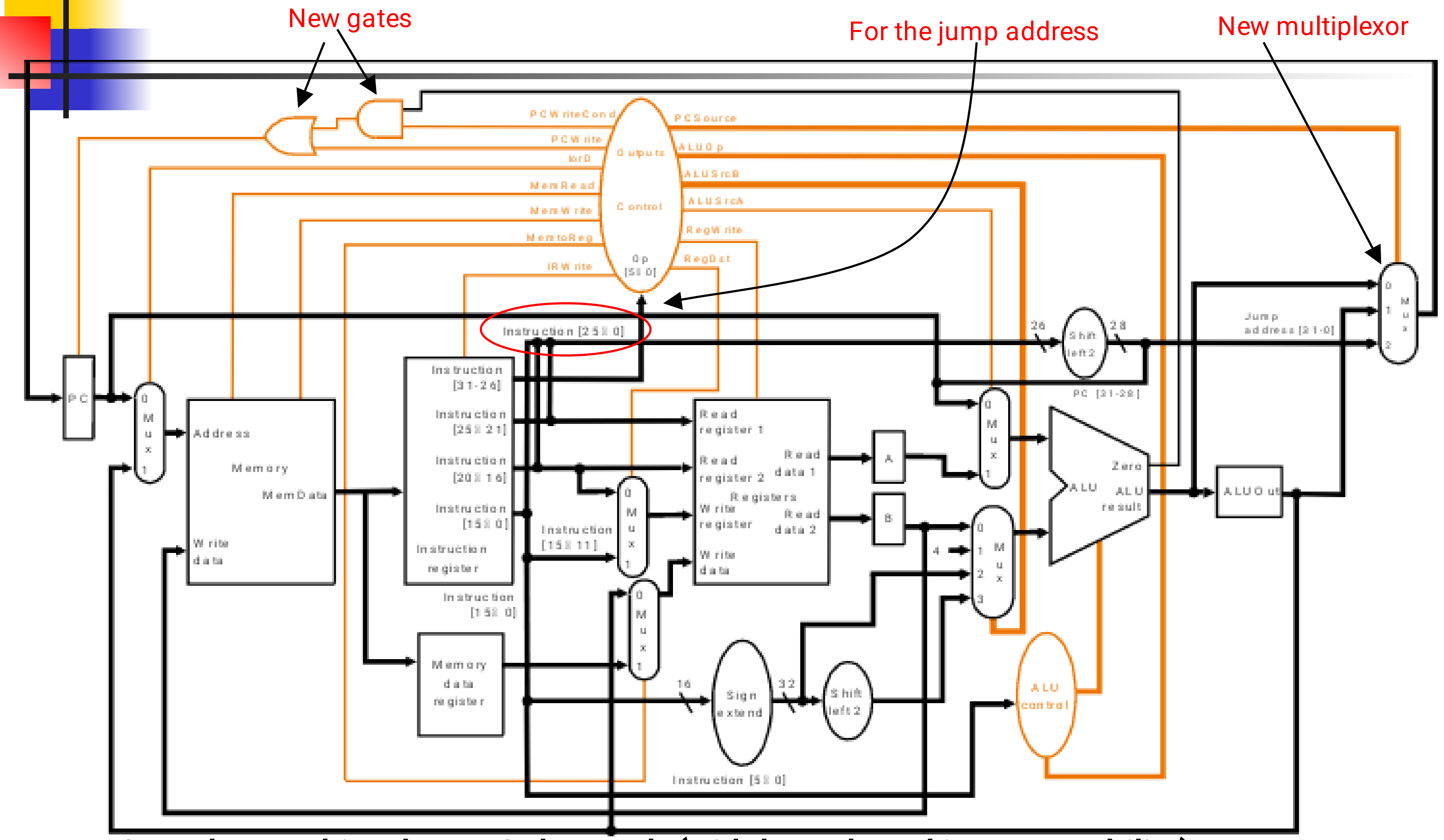
Multi-cycle design:Advantages

- The ability to allow instructions to take different numbers of clock cycles
- The ability to share functional units with in the execution of a single instruction

Multi cycle data path with control signals.



Multicycle Datapath with Control II



Complete multicycle MIPS datapath (with branch and jump capability) and showing the main control block and all control lines

Breaking instructions into steps

- Our goal is to break up the instructions into *steps* so that
  - **each step takes one clock cycle**
  - 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* !!**

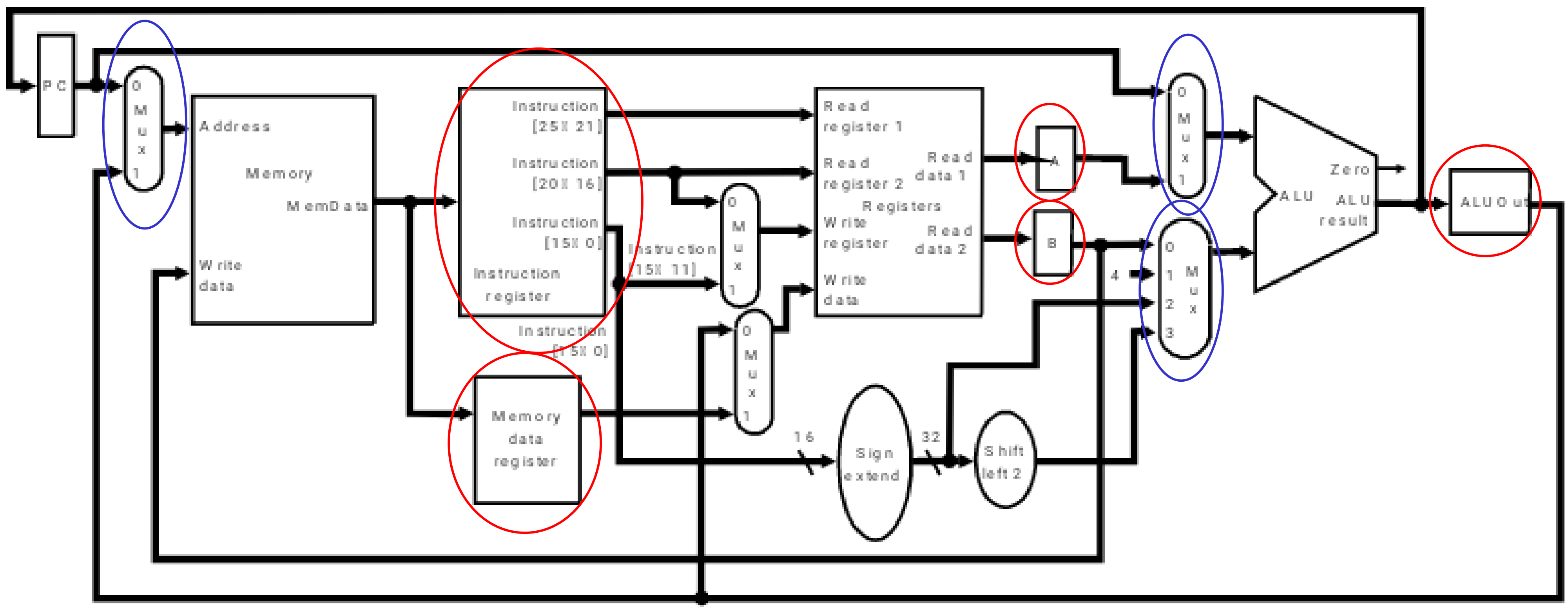
- Multi-cycle implementation attempt to keep the amount of work per cycle roughly equal.
  - Restrict each step to contain at most
    - One ALU operation
    - One register file access
    - One memory access
- With this restriction clock cycle could be as short as the longest of these operations.

Breaking instructions into steps

- We break instructions into the following*potential* execution 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. Writing data back to register file(**WB**)

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

Multicycle Datapath



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

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.

(Instruction Fetch)

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.

```
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:

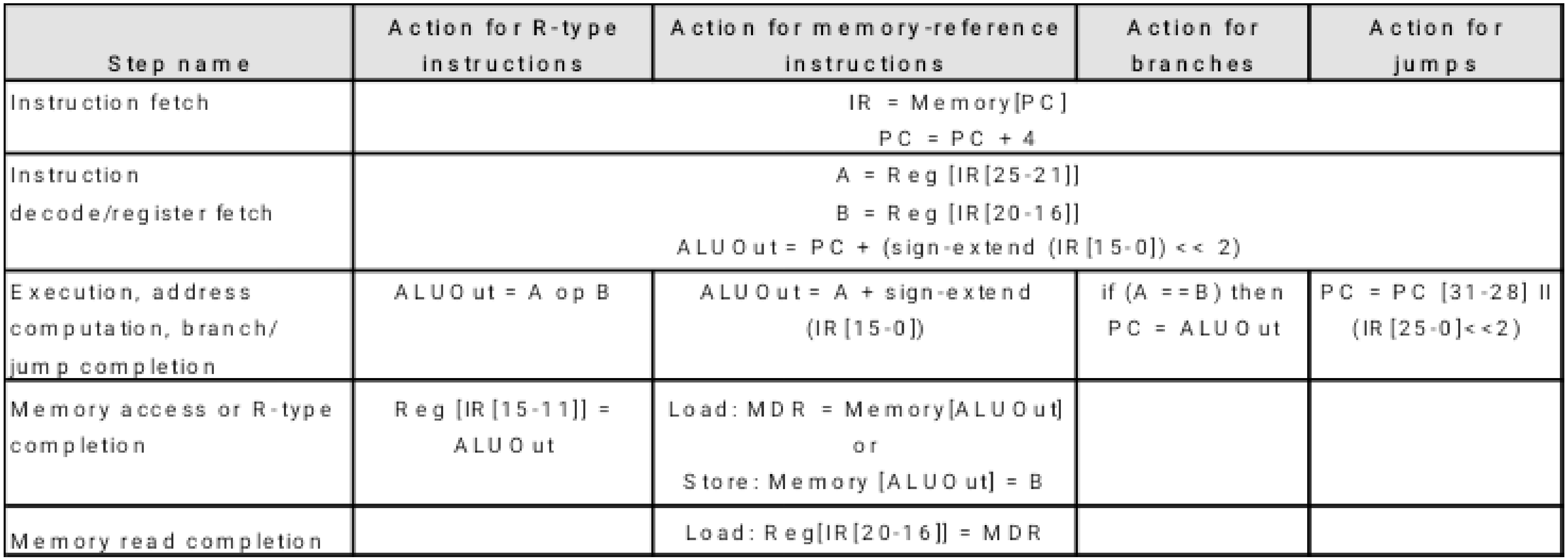
- Load writes back (instruction *completes*)

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.**

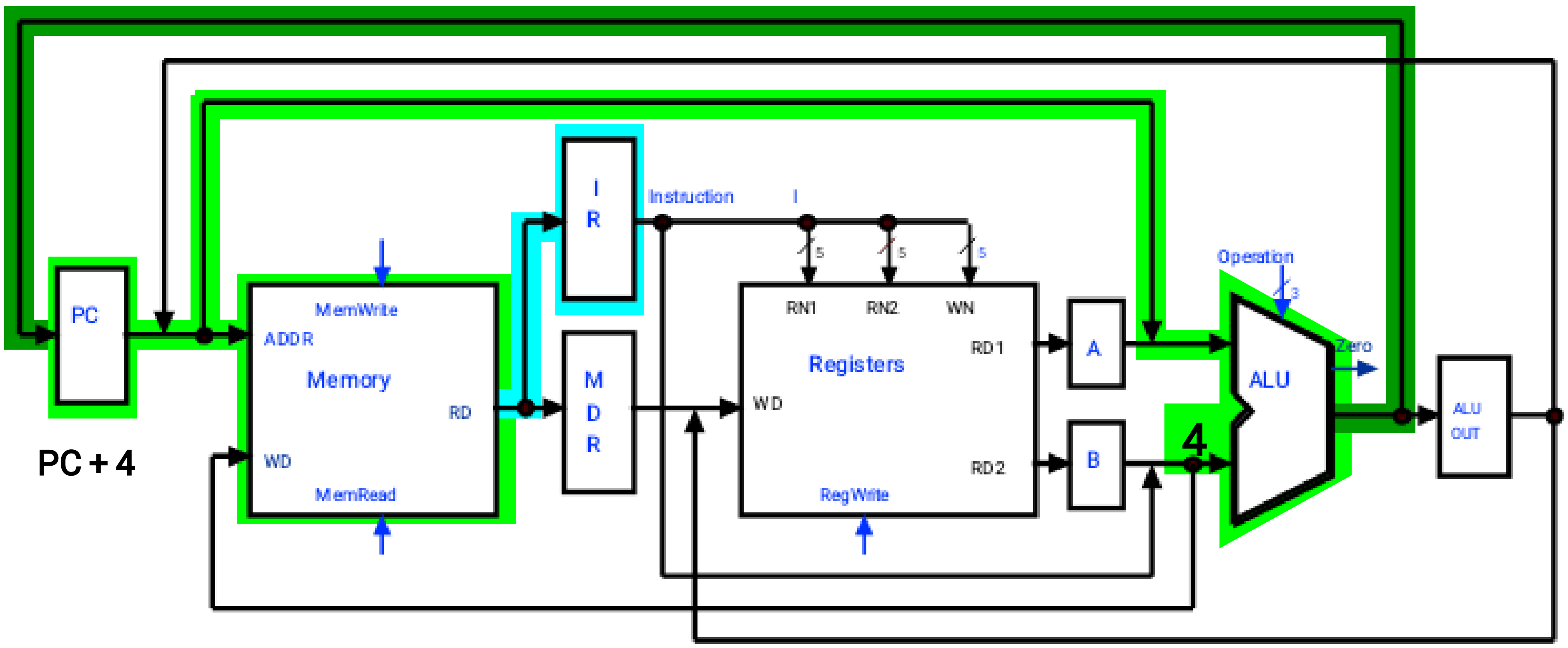
Summary of Instruction Execution



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 + 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] or Store: Memory [ALUOut] = B		
5: WB	Memory read completion		Load: Reg[IR[20-16]] = MDR		

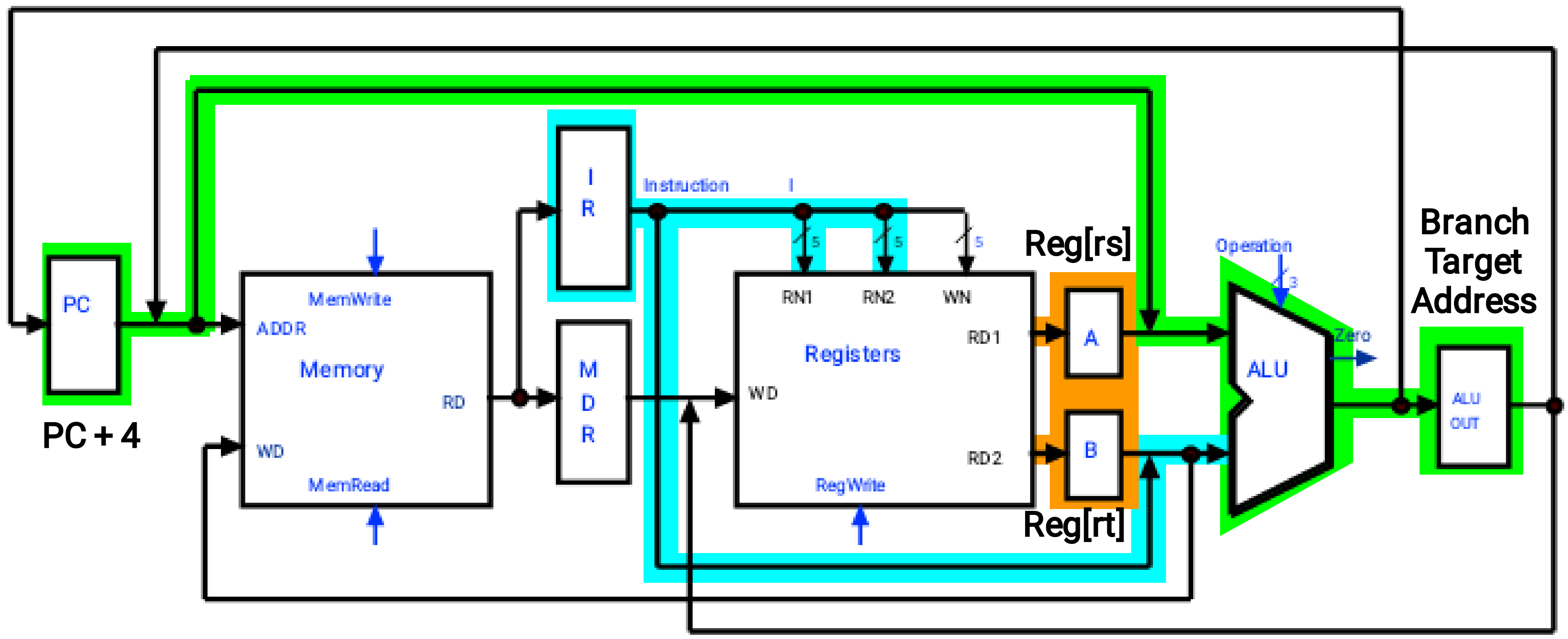
Multicycle Execution Step (1): Instruction Fetch

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



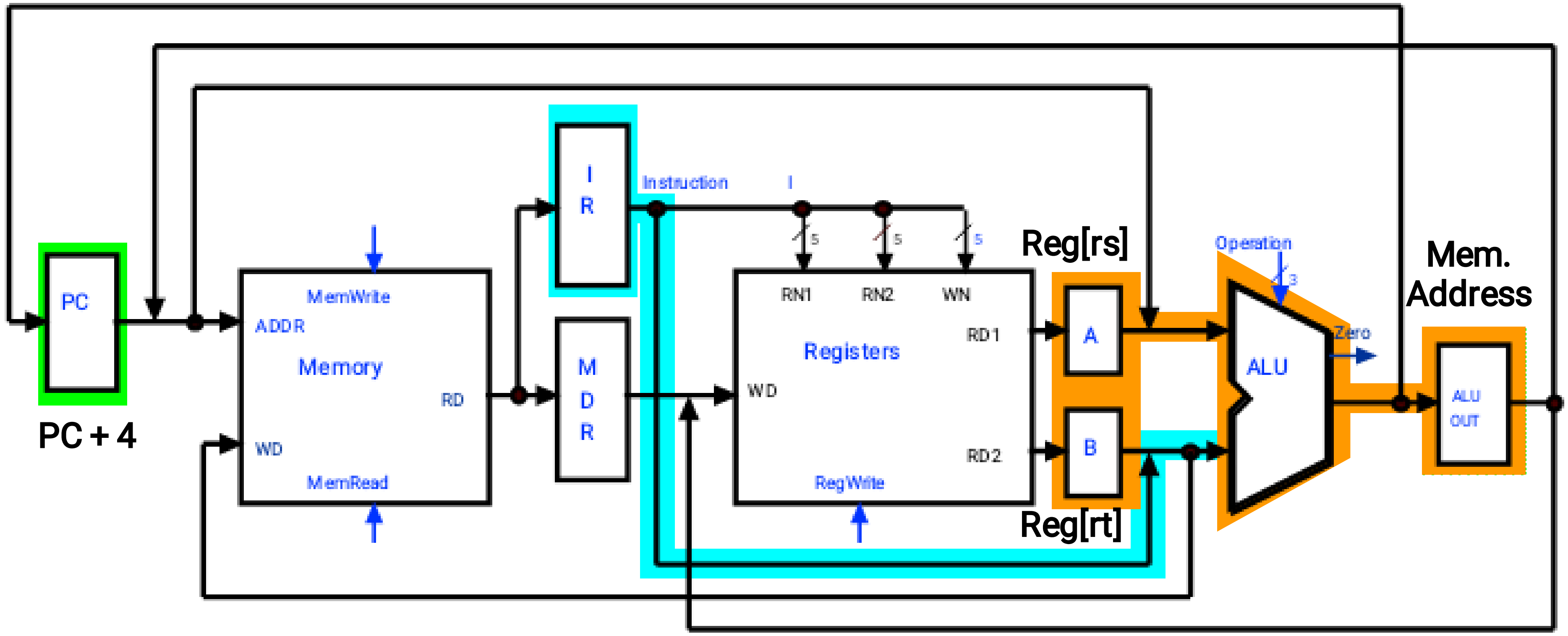
Multicycle Execution Step (2): Instruction Decode & Register Fetch

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



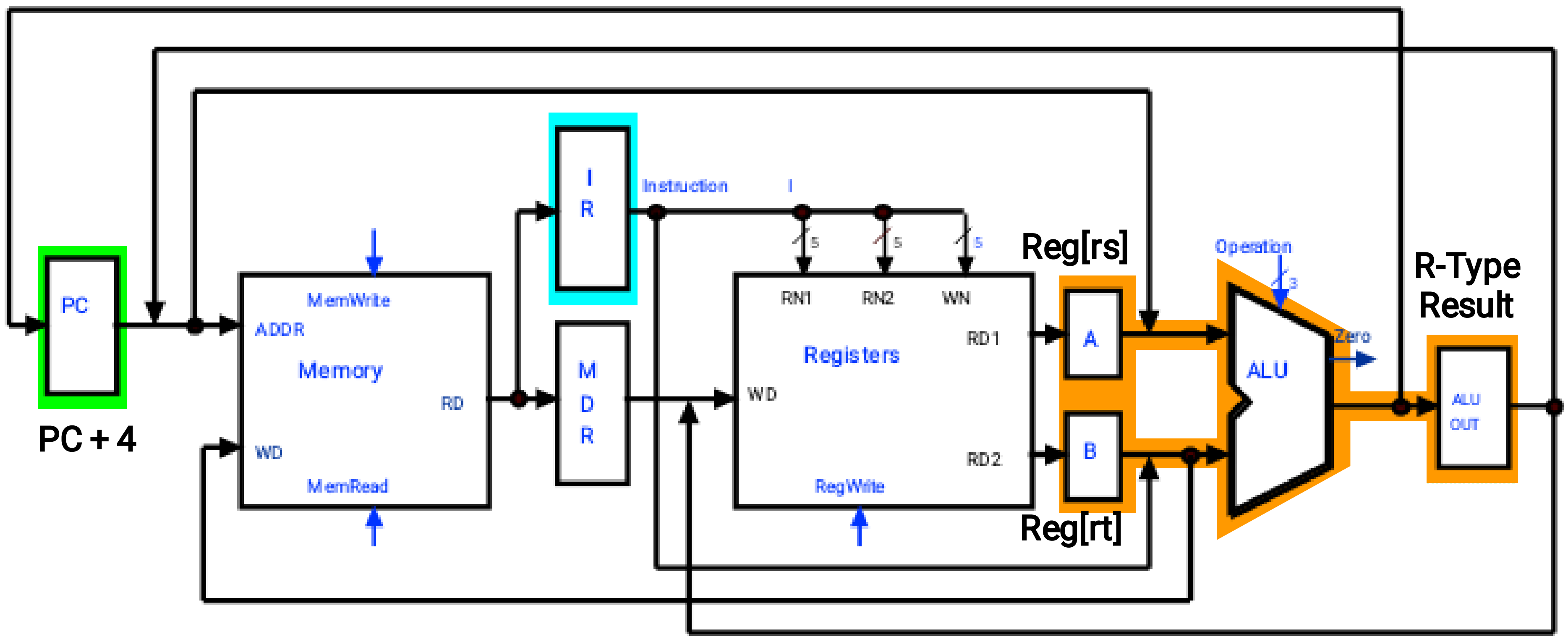
Multicycle Execution Step (3): Memory Reference Instructions

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



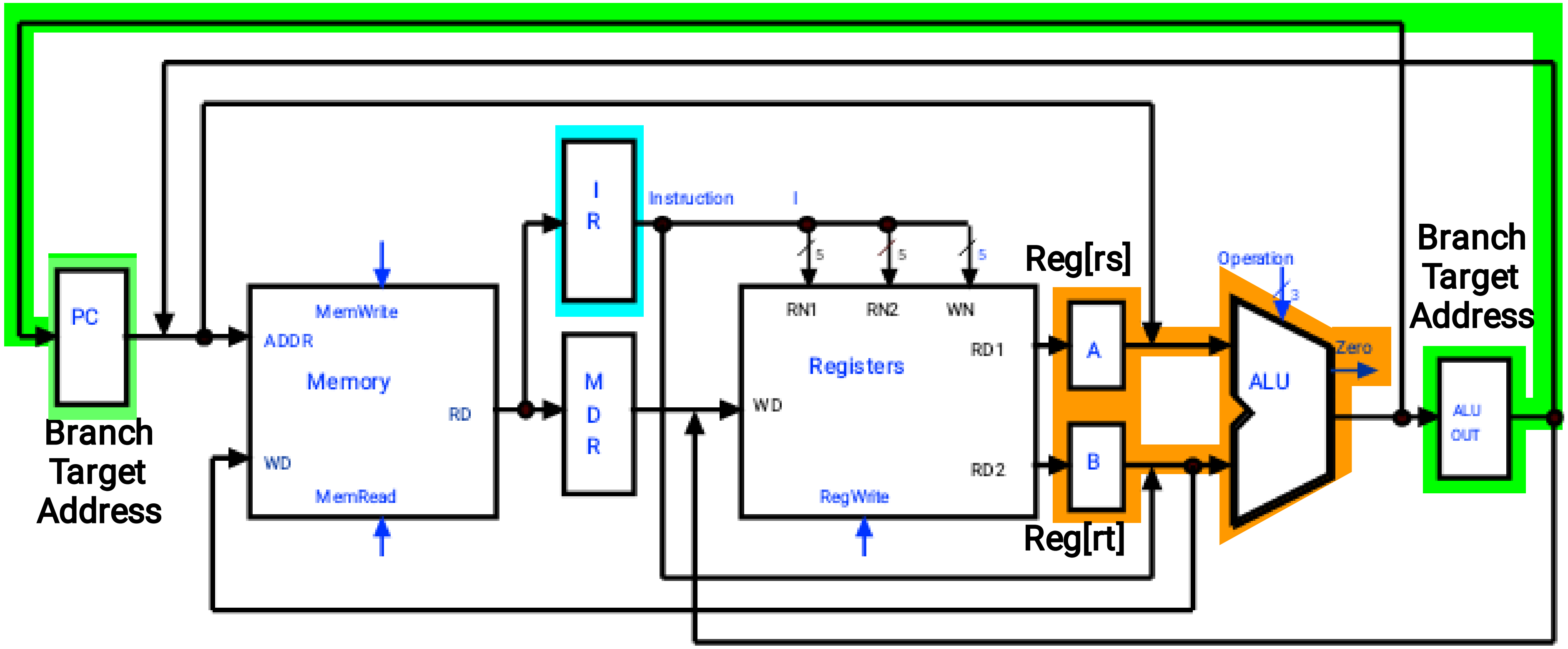
Multicycle Execution Step (3): ALU Instruction (R-Type)

ALUOut = A op B



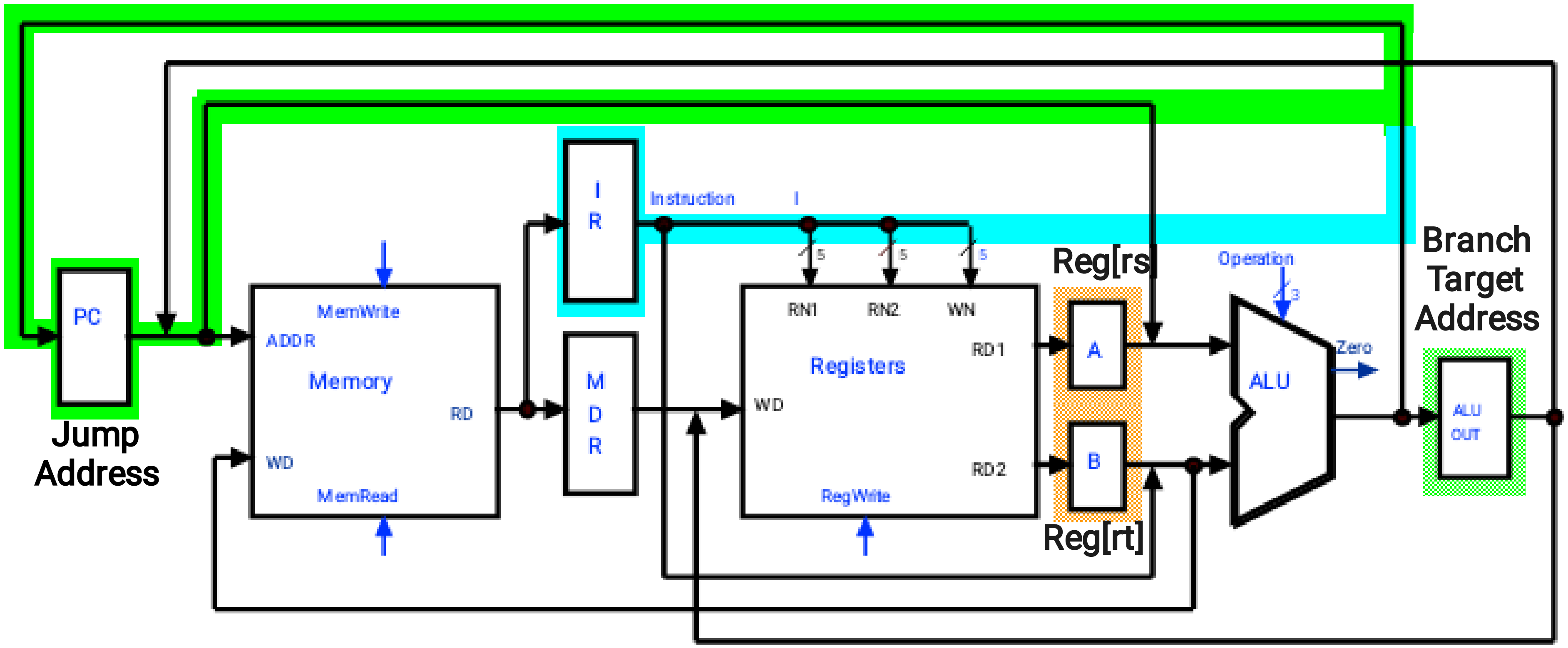
Multicycle Execution Step (3): Branch Instructions

if (A == B) PC = ALUOut;



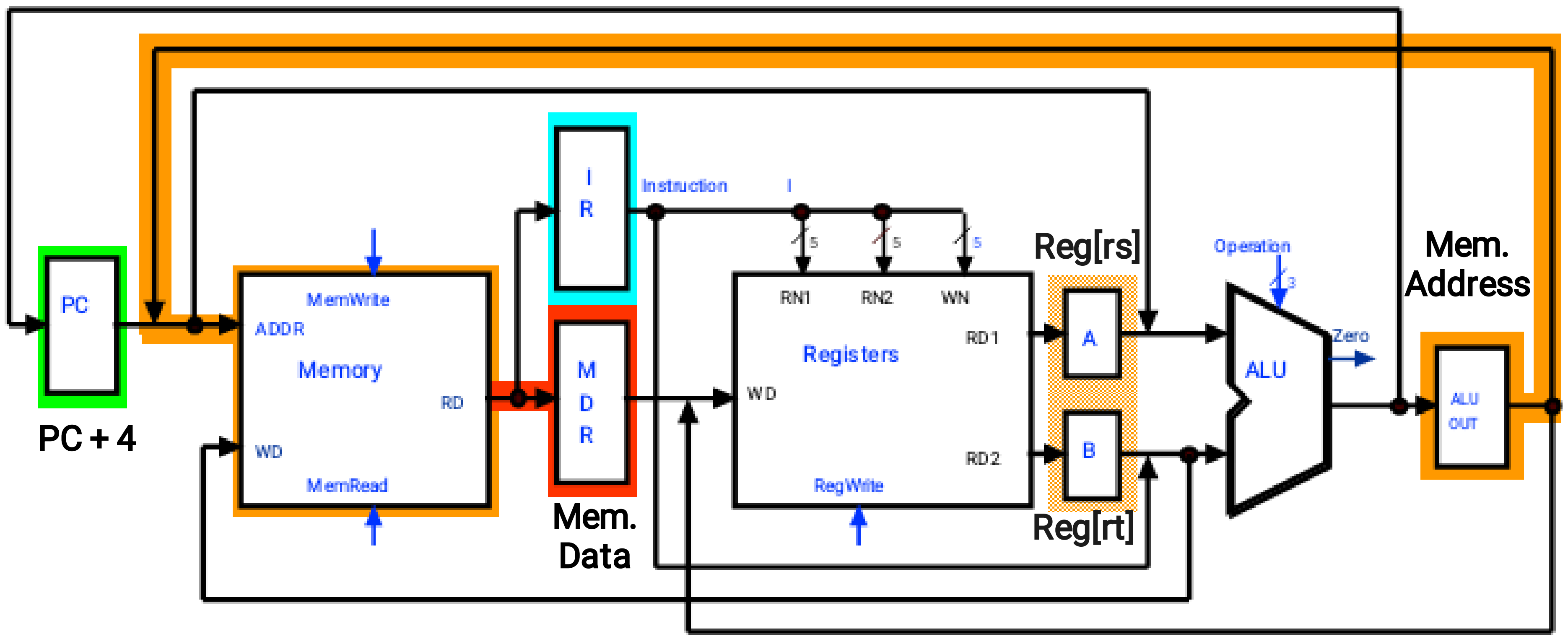
Multicycle Execution Step (3): Jump Instruction

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



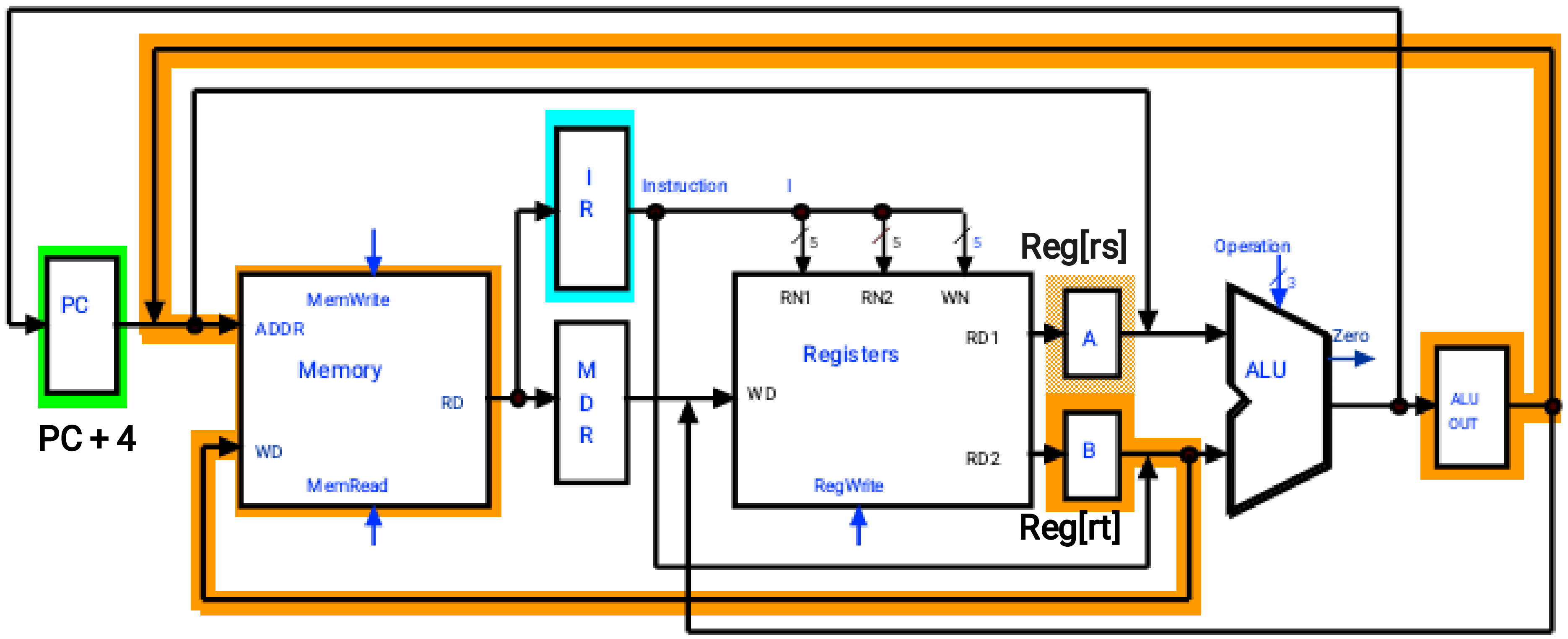
Multicycle Execution Step (4): Memory Access - Read (lw)

MDR = Memory[ALUOut];



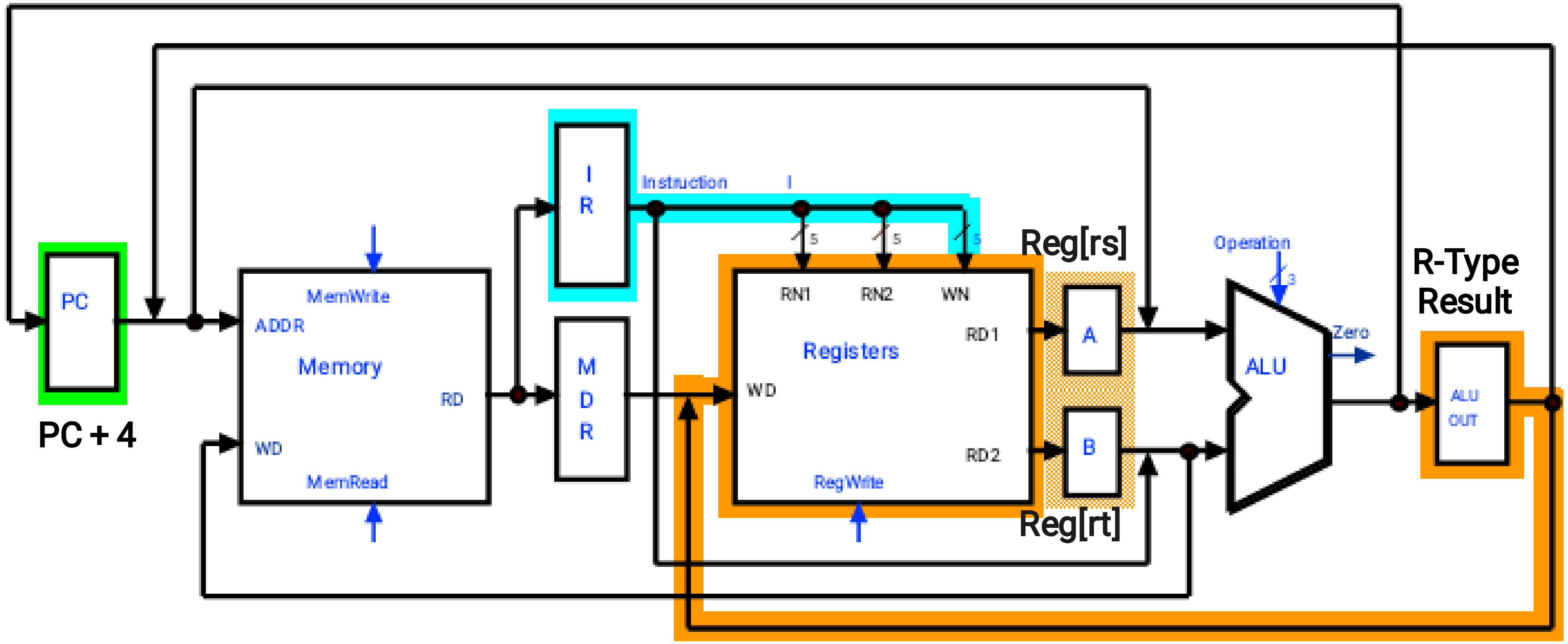
Multicycle Execution Step (4): Memory Access - Write (sw)

Memory[ALUOut] = B;



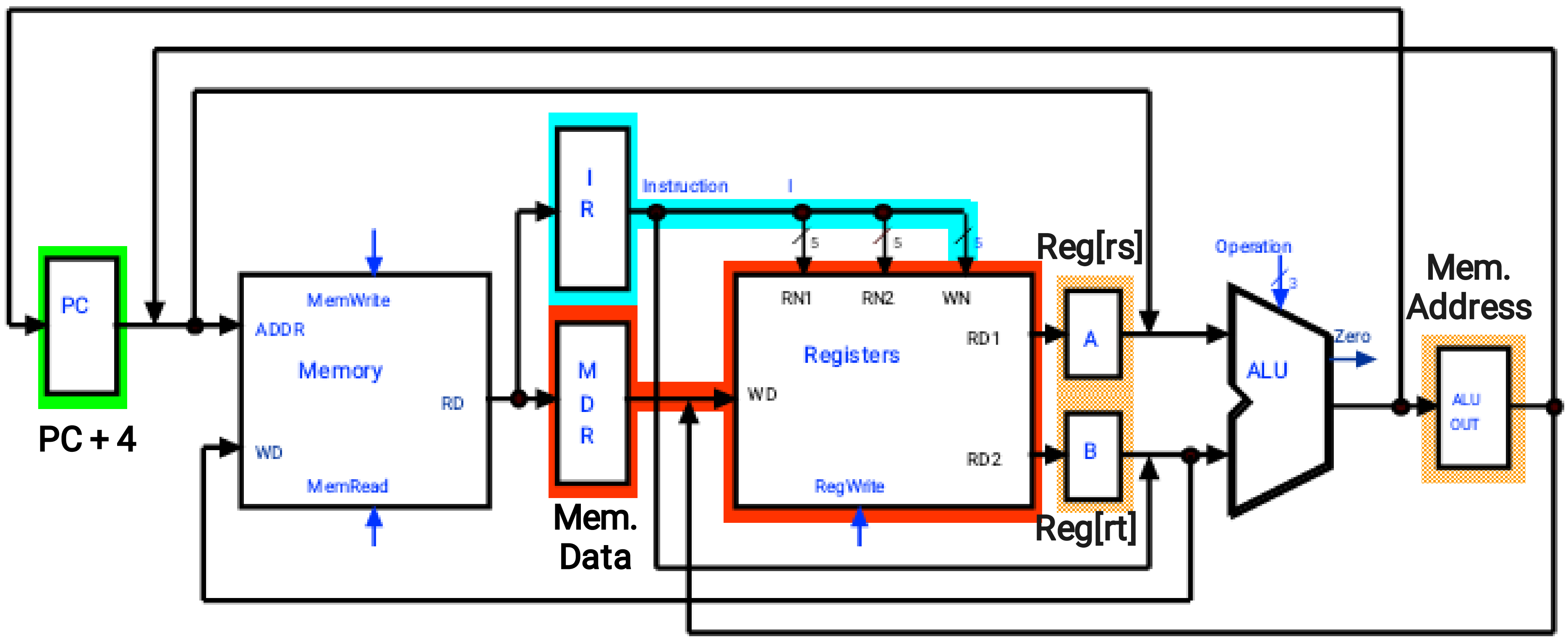
Multicycle Execution Step (4): ALU Instruction (R-Type)

Reg[IR[15:11]] = ALUOUT



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

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



Instruction execution review

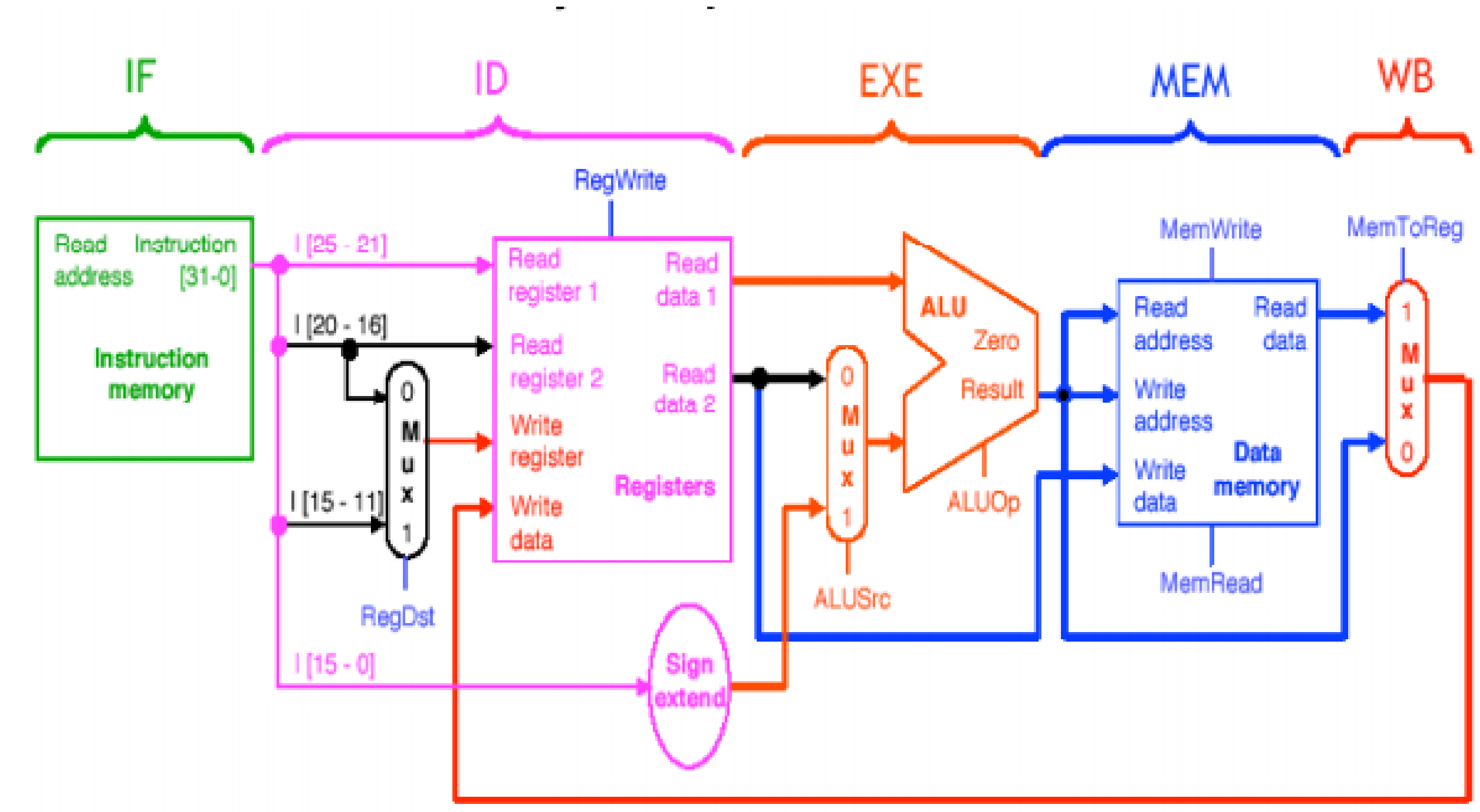
- Executing a MIPS instruction can take up to five steps.

Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

- However, as we saw, not all instructions need all five steps.

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

Break data path into 5 stages


IF
ID
EXE
MEM
WB
RegWrite
Read address
Instruction [31-0]
Instruction memory
I [25 - 21]
I [20 - 16]
I [15 - 11]
I [15 - 0]
0 Mux 1
RegDst
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Sign extend
0 Mux 1
ALUSrc
ALU
Zero
Result
ALUOp
MemWrite
Read address
Write address
Write data
Read data
Data memory
MemRead
MemToReg
1 Mux 0