Processor: Datapath and Control

Book of David A. Patterson

[Single cycle and Multicycle Processor]

Implementing Jump

- **<u>Format of J-type Instruction:</u>**

opcode	Addresses

6 bits 26 bits

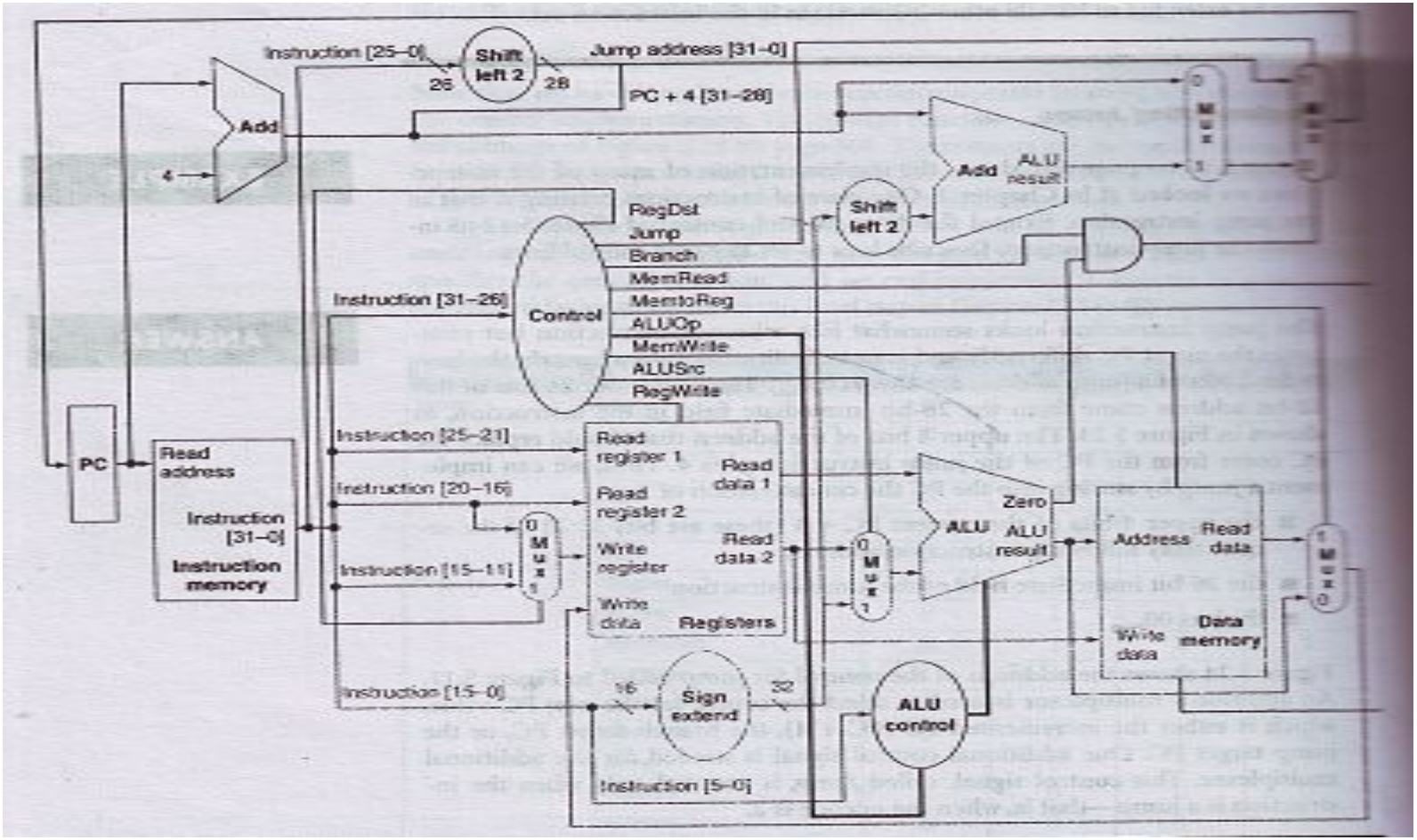
- Jump address is calculated as follows:

  The upper 4-bits of the current PC+4 [31:28] + 26 bits immediate field of the Jump instruction + 00_2

- Implementation of Jump requires:

  1. An additional multiplexor
  2. Control signal *Jump* from the main control unit.

Control and Datapath to Handle the Jump Instruction



Drawback of Single Cycle Processor

- The clock cycle must have same length for every instruction. The cycle time must be long enough for the load instruction.

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 performance is not good since, several of the instruction classes could fit in a shorter clock cycle.

Limitations of Single Cycle Processor

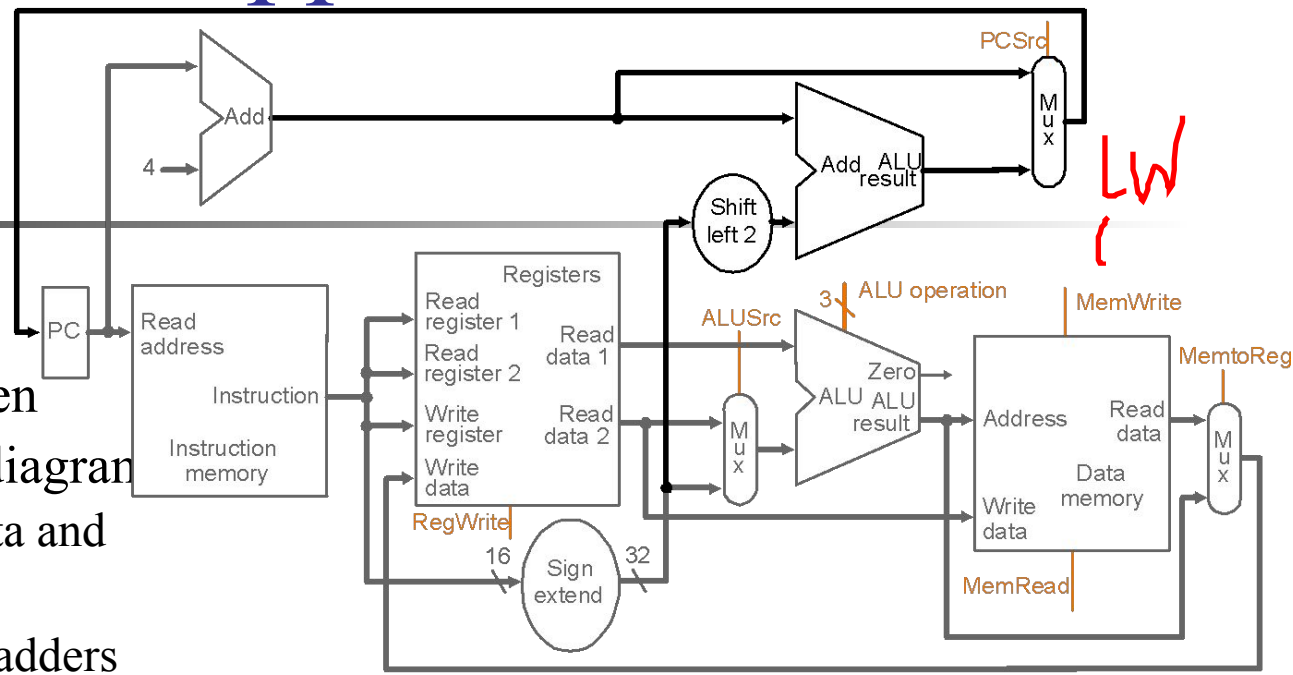
- Each instruction is executed in one clock cycle.
- Clock cycle length is long enough to accommodate the load instruction.
- Cycle time is much longer than needed for all other instructions.
- Single cycle execution requires the duplication of several resources.

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
  - Data to be used in later clock cycles are stored in the additional registers.

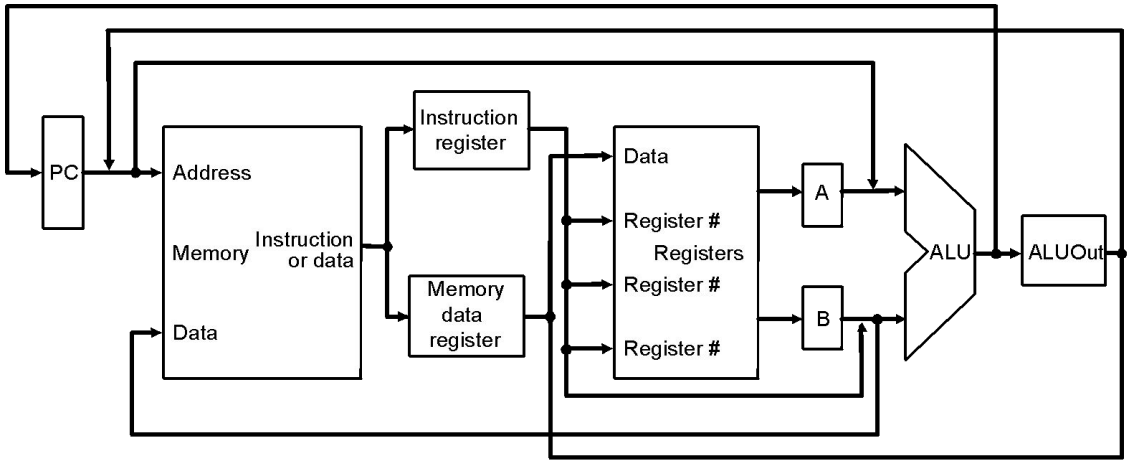
Multicycle Approach

- Note differences between multicyle vs. single cycle diagram
  - single memory for data and instructions
  - single ALU, no extra adders
  - extra registers to hold data between clock cycles



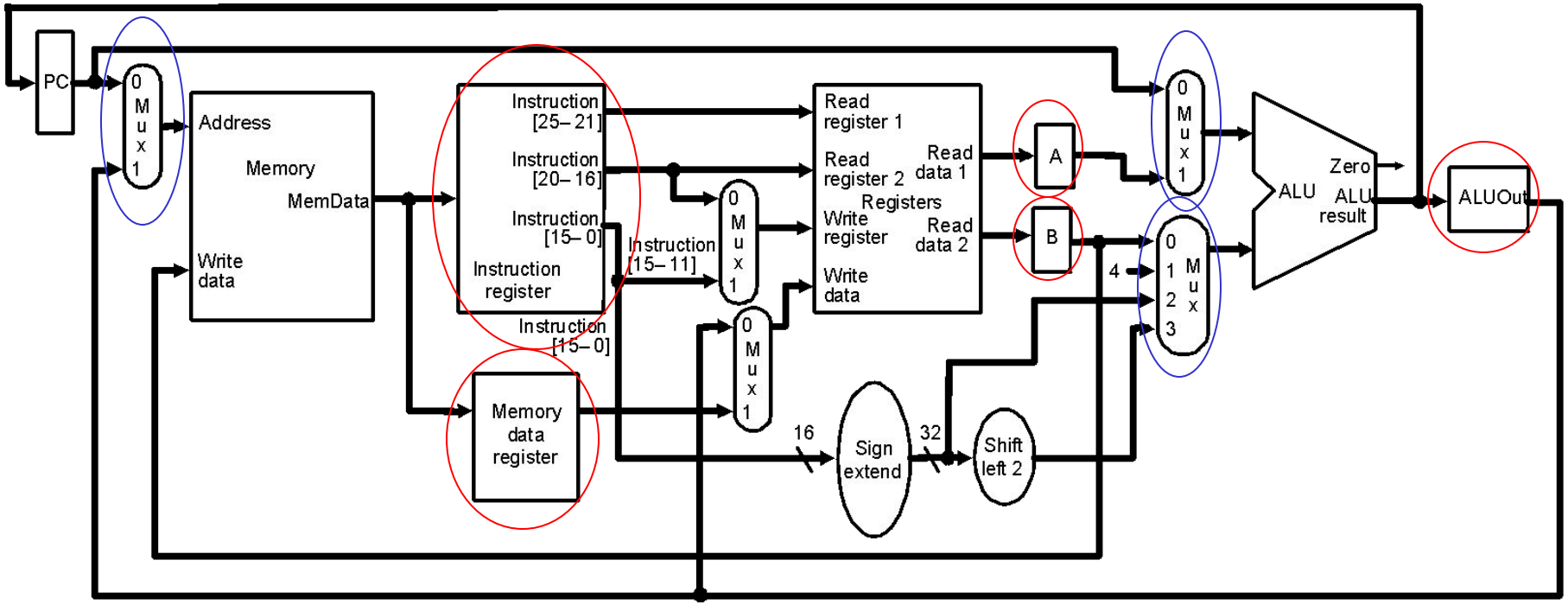
LW

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

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)

Instruction	Description	Inside processor's task
Add Sub	R[rd] <– R[rs] + R[rt]; PC <– PC + 4 CLK CYCLE :– 4	1. IR <– MEM[pc]; PC <– PC + 4 2. A<– R[rs]; B <– R[rt] 3. S <– A + B 4. R[rd] <– S;
Load	R[rt]<– MEM[R[rs]+SExt(Im16)]; PC <– PC + 4 CLK CYCLE :- 5	1. IR <– MEM[pc]; PC <– PC + 4 2. A<– R[rs]; 3. S <– A + SExt(Im16) 4. M <– MEM[S]; 5. R[rt] <– M;
Store	MEM[R[rs]+SExt(Im16)] <– R[rt]; PC <– PC + 4 CLK CYCLE :– 4	1. IR <– MEM[pc]; PC <– PC + 4 2. A<– R[rs]; B <– R[rt]; 3. S <– A + SExt(Im16); 4. MEM[S] <– B;
Branch	if R[rs] == R[rt] then PC <-PC + 4+SExt(Im16) || 00 else PC <= PC + 4 CLK CYCLE :- 3	1. IR <– MEM[pc]; PC <– PC + 4 3. E<– (R[rs] = R[rt]) if !E then *do nothing* 2. Else PC<-PC+SExt(Im16)||00

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

Br

Branch

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)

lw

st

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;

op

Step 5: Memory Read Completion (**WB**)

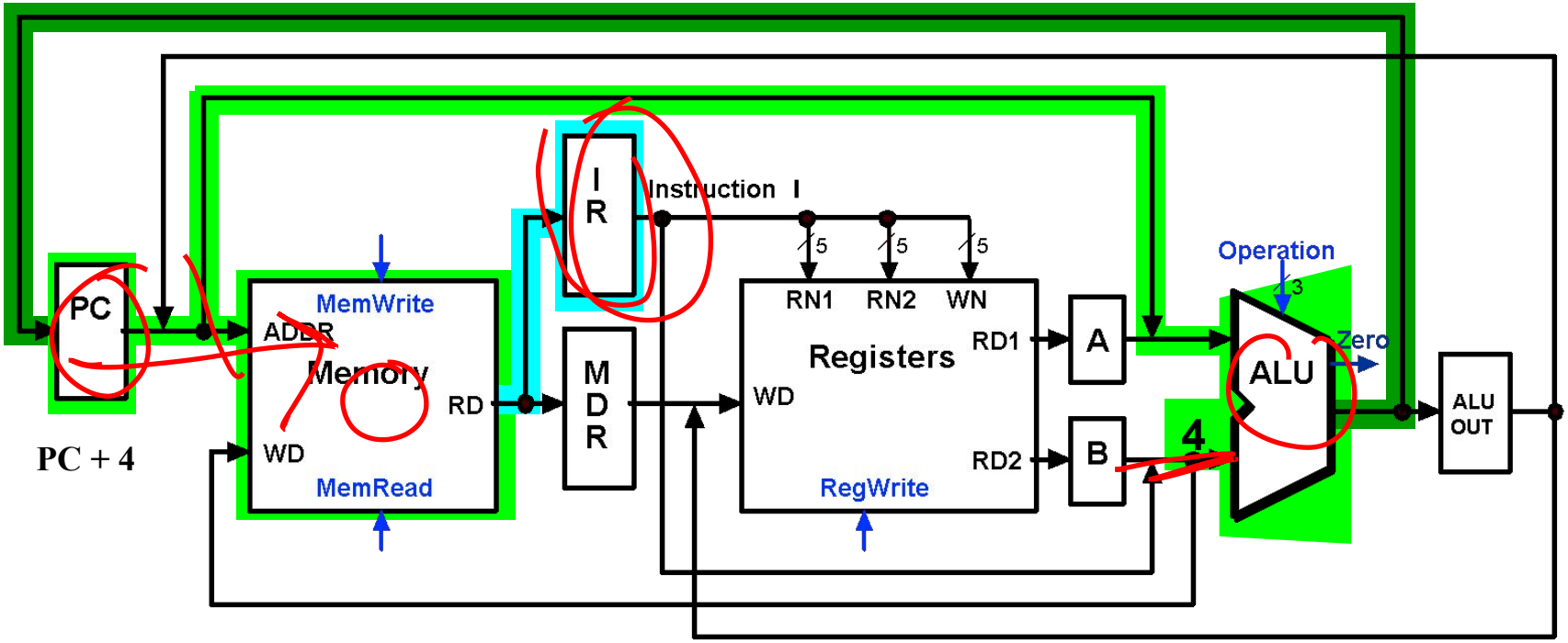
- Again *depending* on instruction type:
- Load writes back (instruction *completes*)
- Reg[IR[20-16]]= MDR;

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		

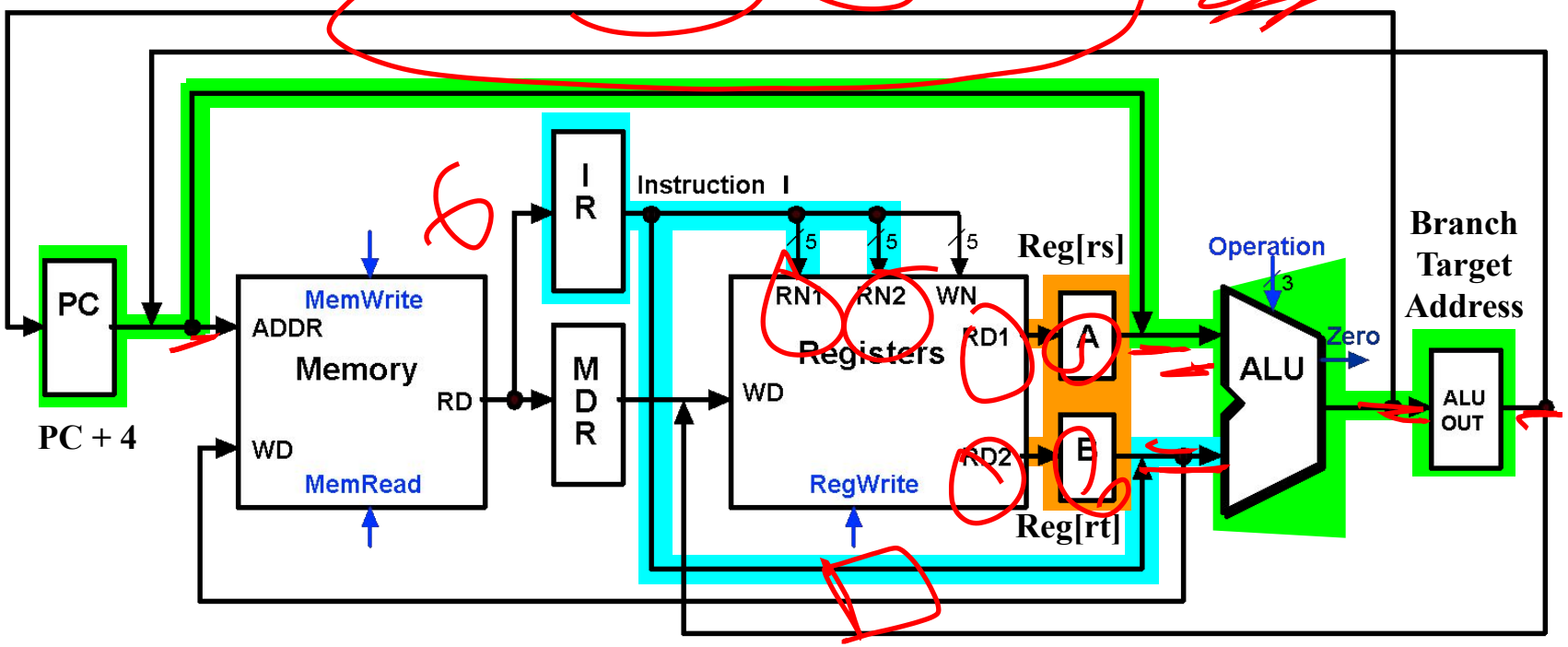
Multicycle Execution Step (1): Instruction Fetch

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



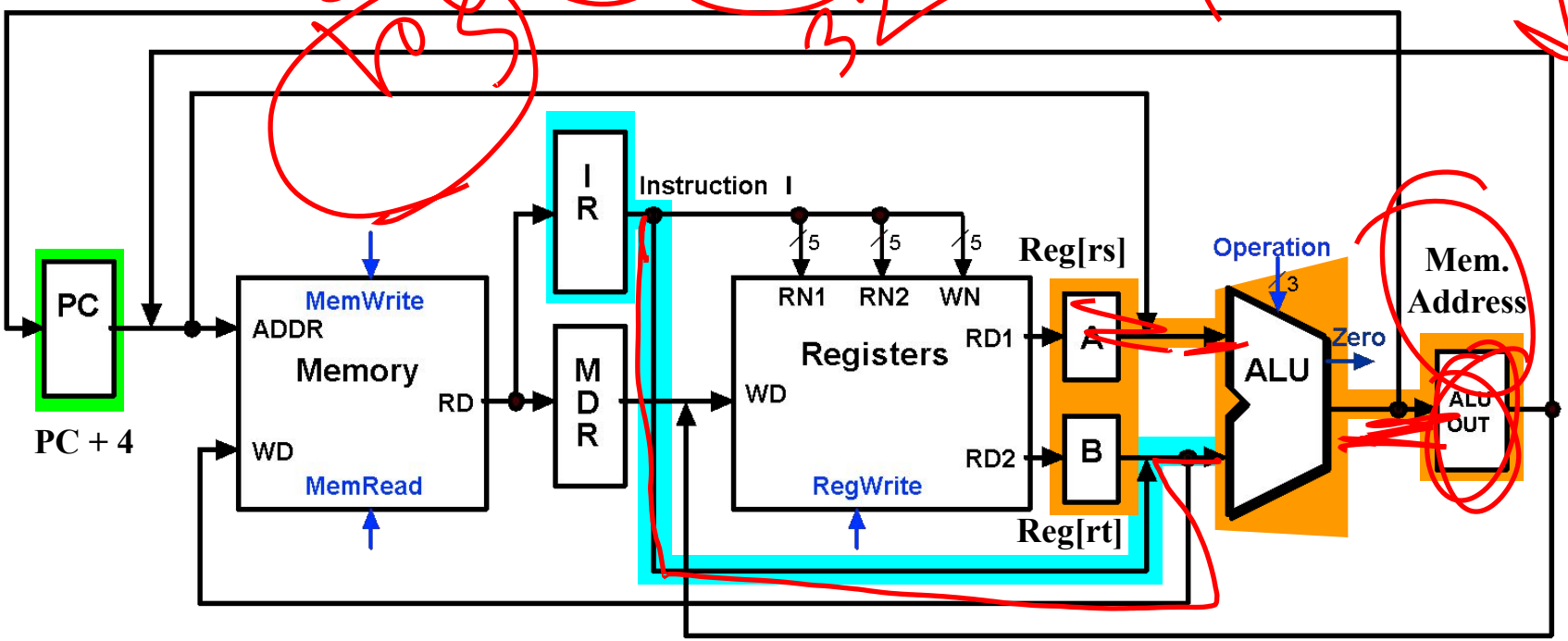
Multicycle Execution Step (2): Instruction Decode & Register Fetch

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



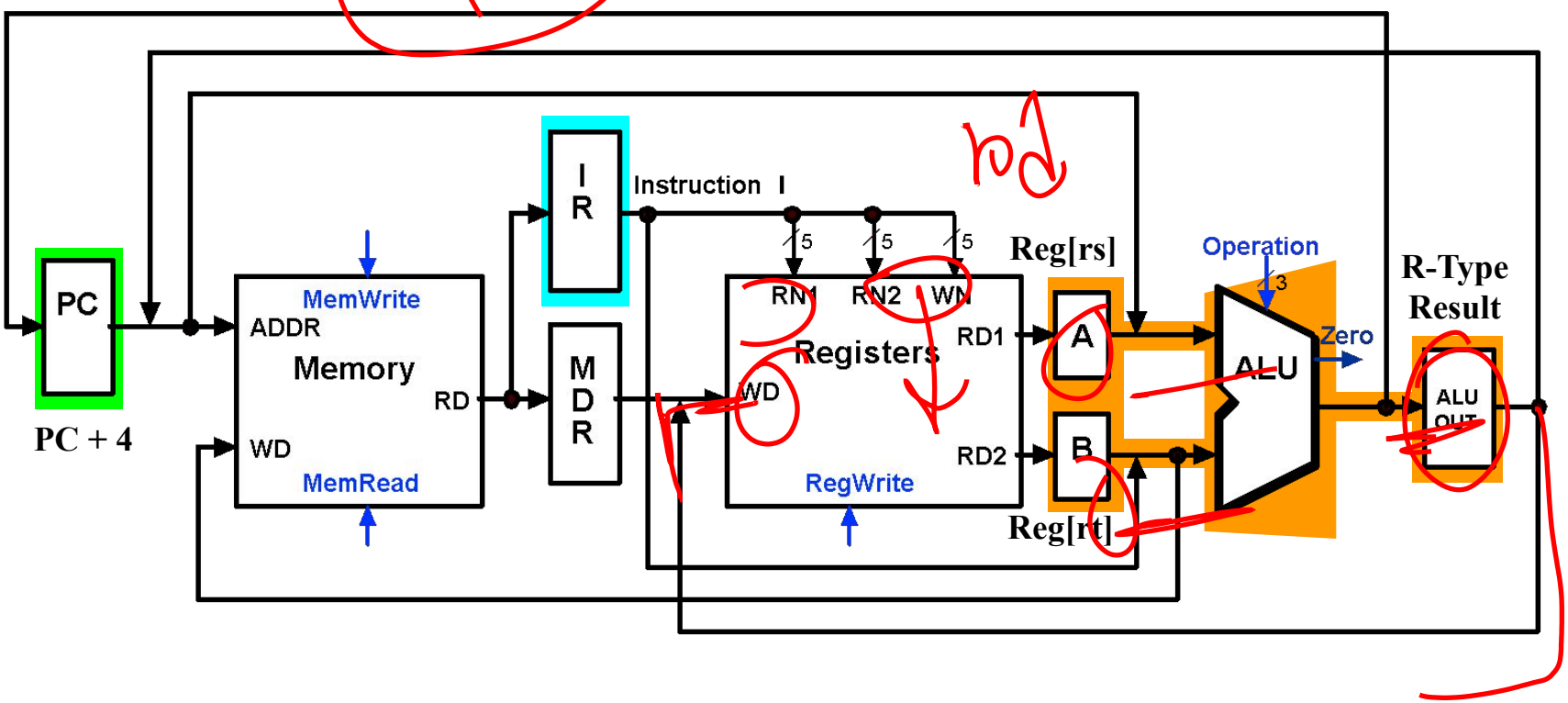
Multicycle Execution Step (3): Memory Reference Instructions

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



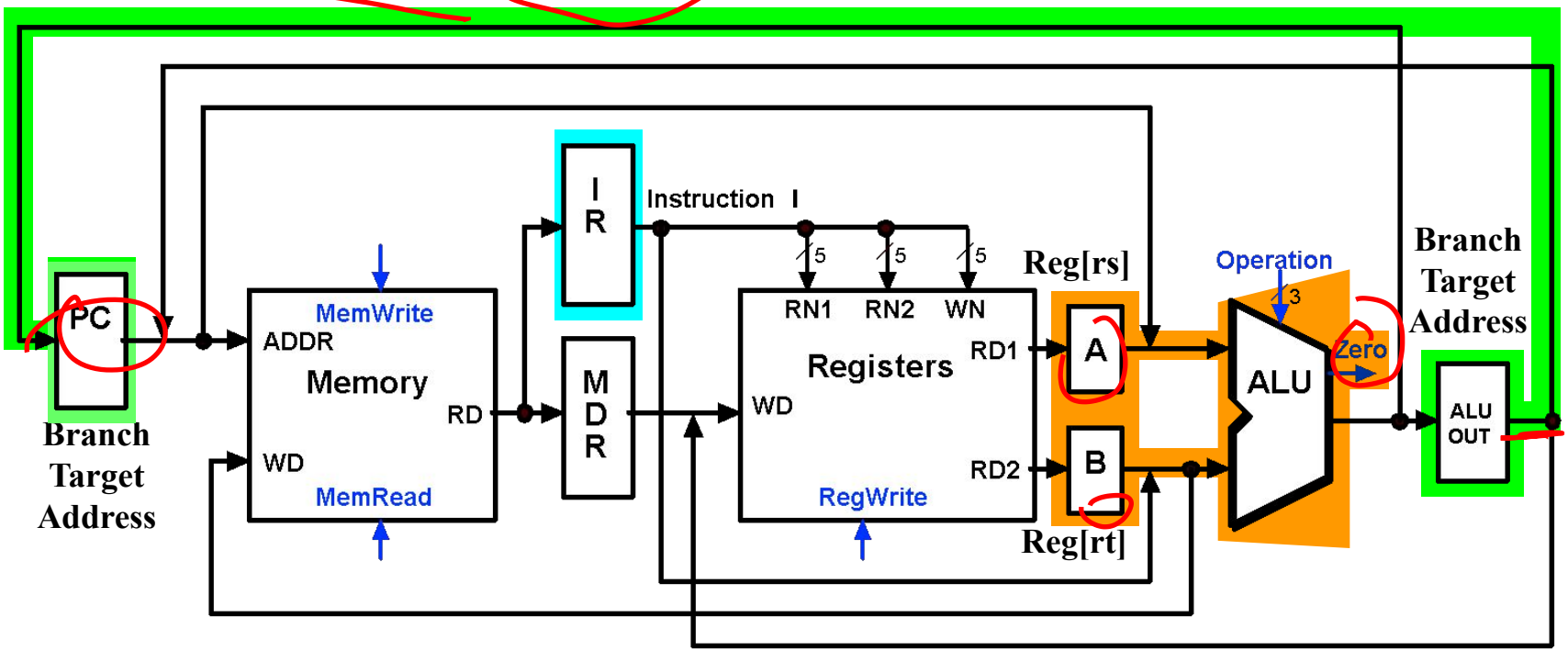
Multicycle Execution Step (3): ALU Instruction (R-Type)

ALUOut = A op B



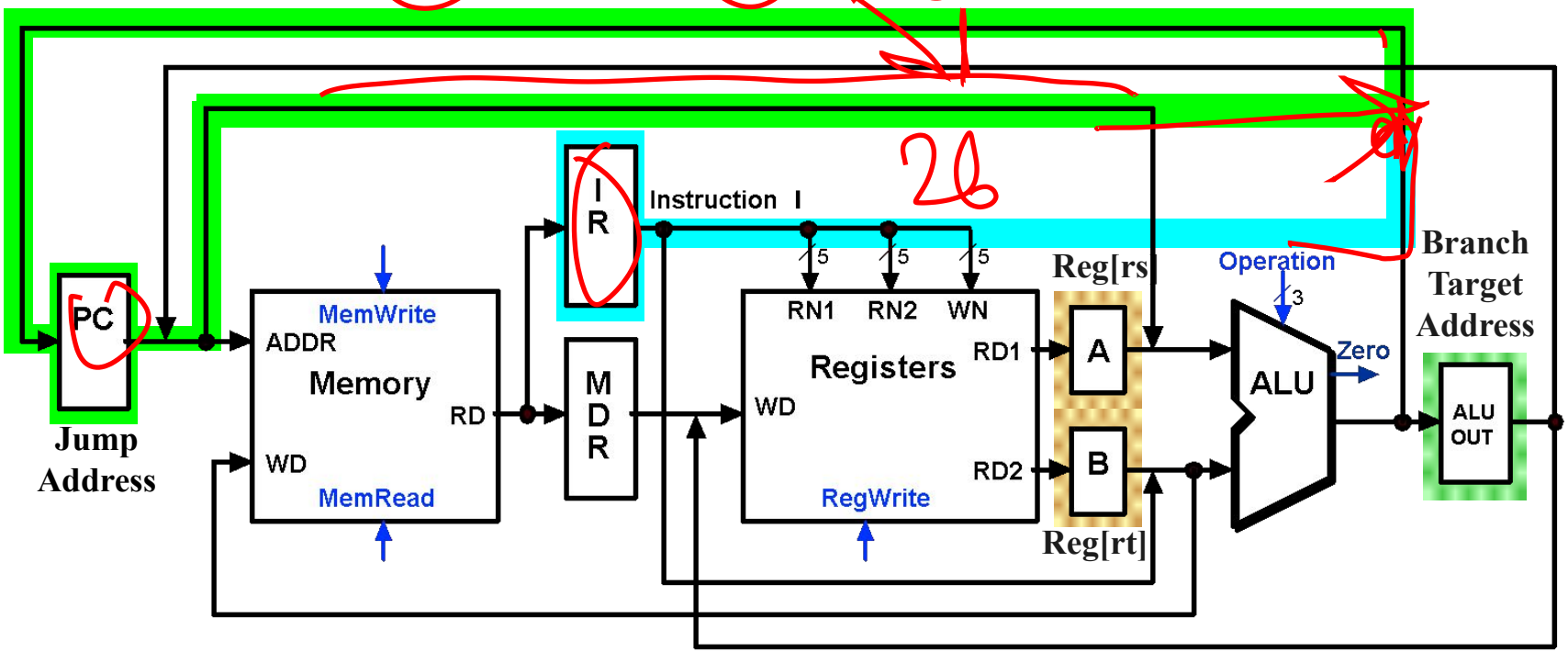
Multicycle Execution Step (3): Branch Instructions

if (A == B) PC = ALUOut;



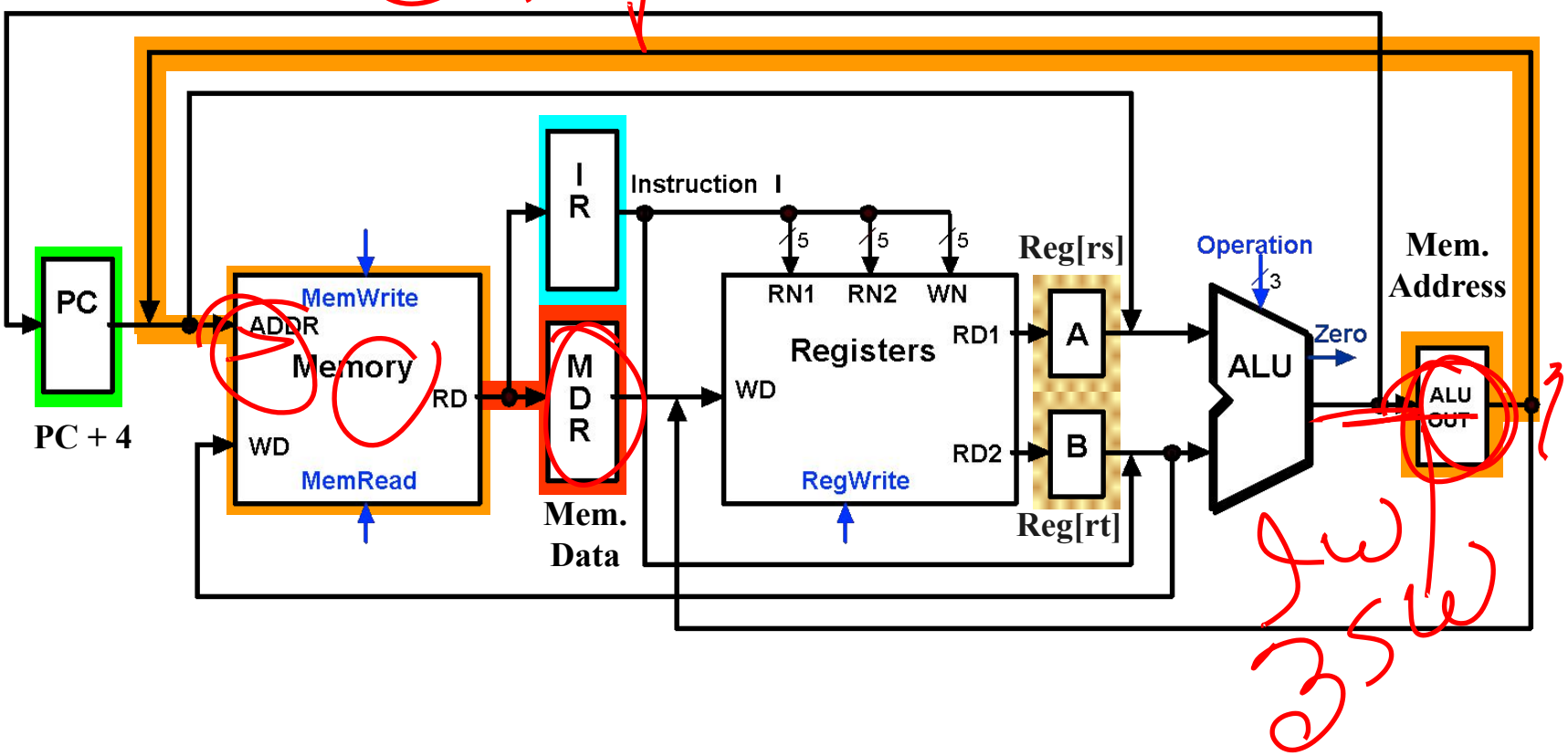
Multicycle Execution Step (3): Jump Instruction

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



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

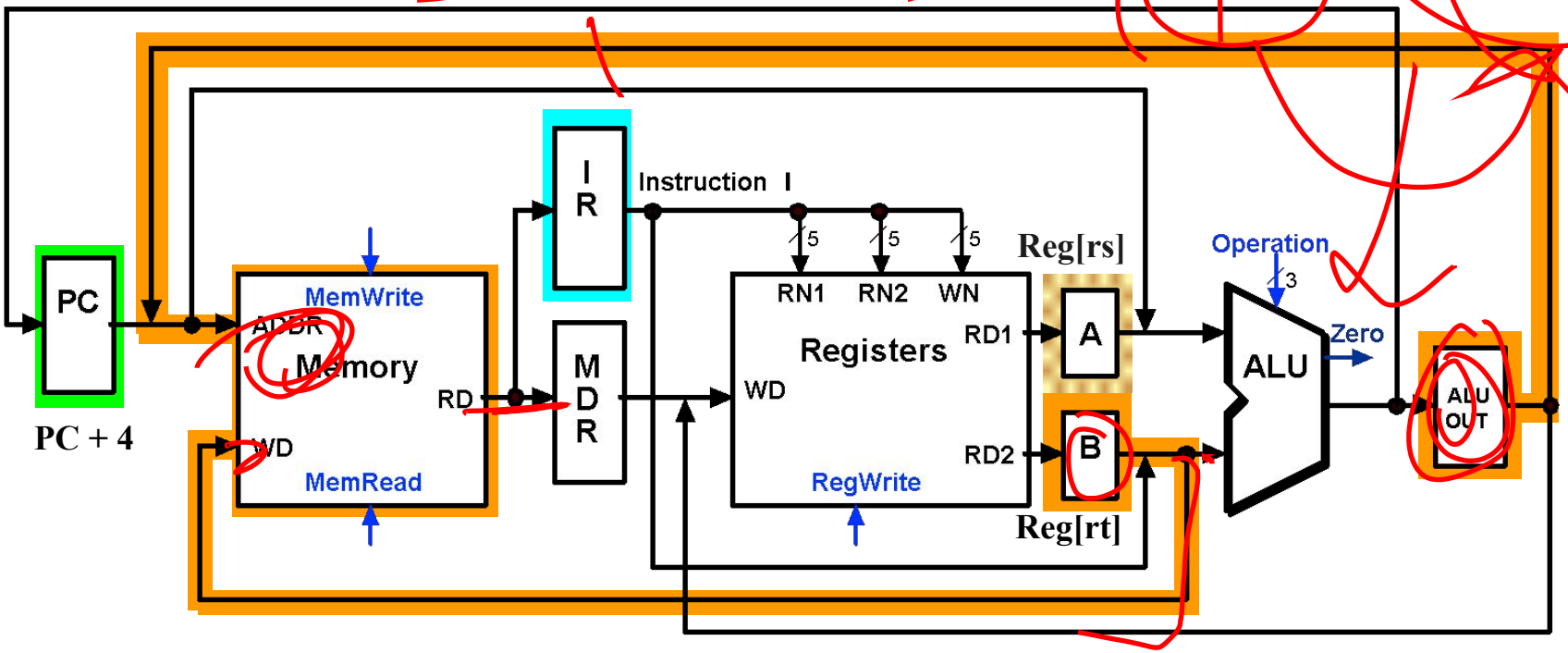
MDR = Memory[ALUOut];



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

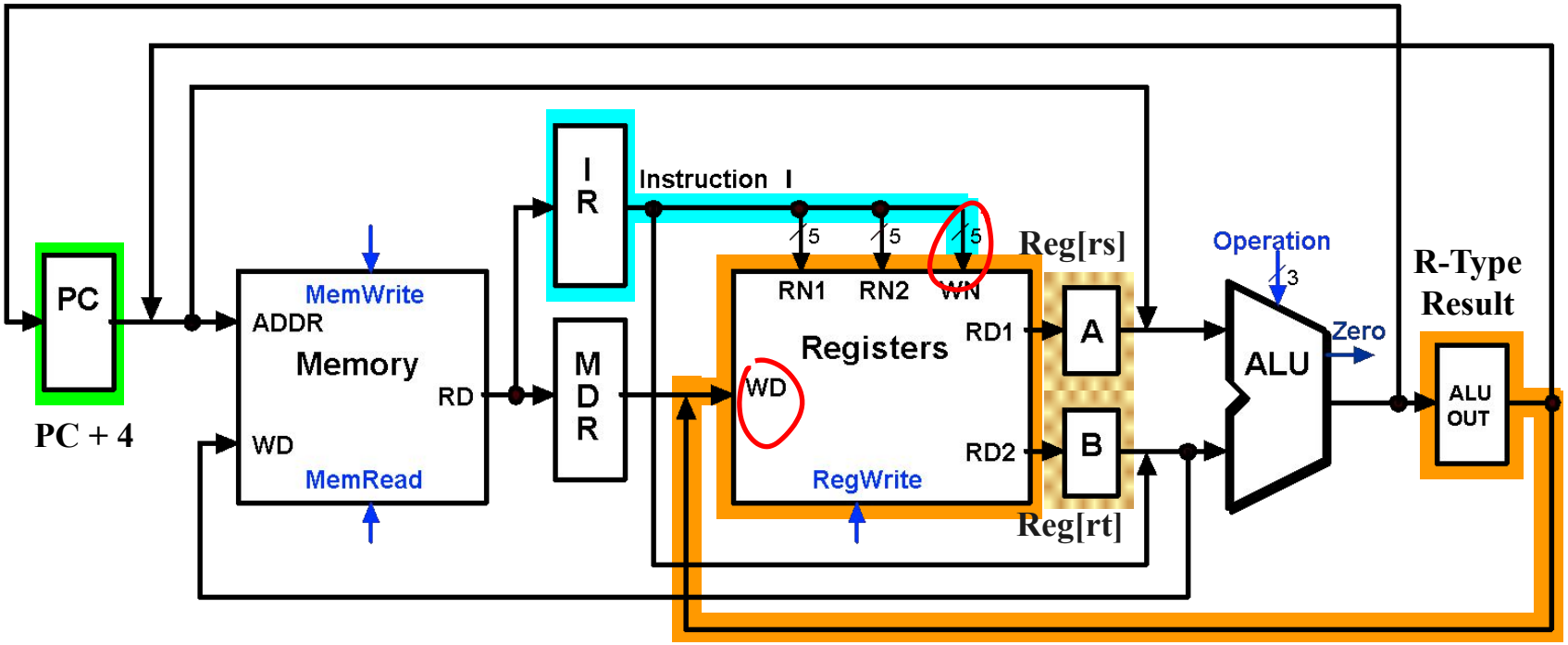
Memory[ALUOut] = B;

sw



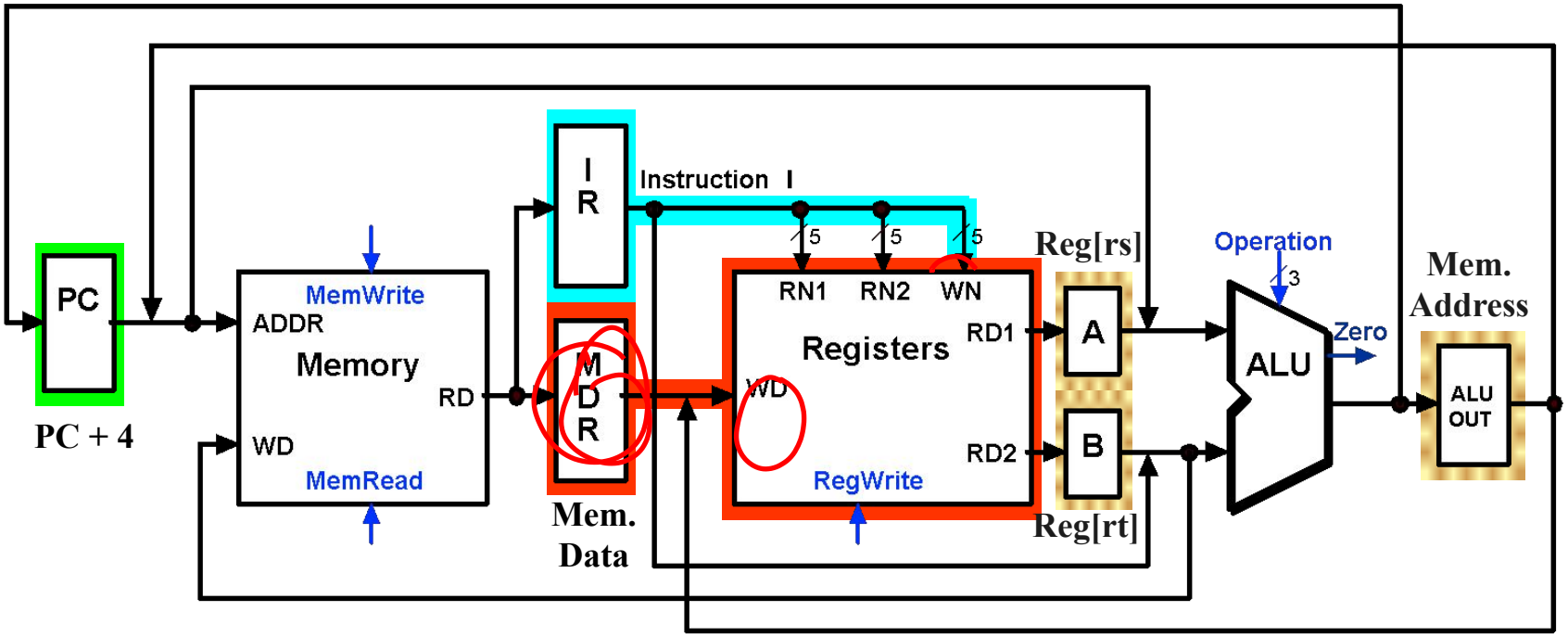
Multicycle Execution Step (4): ALU Instruction (R-Type)

Reg[IR[15:11]] = ALUOUT

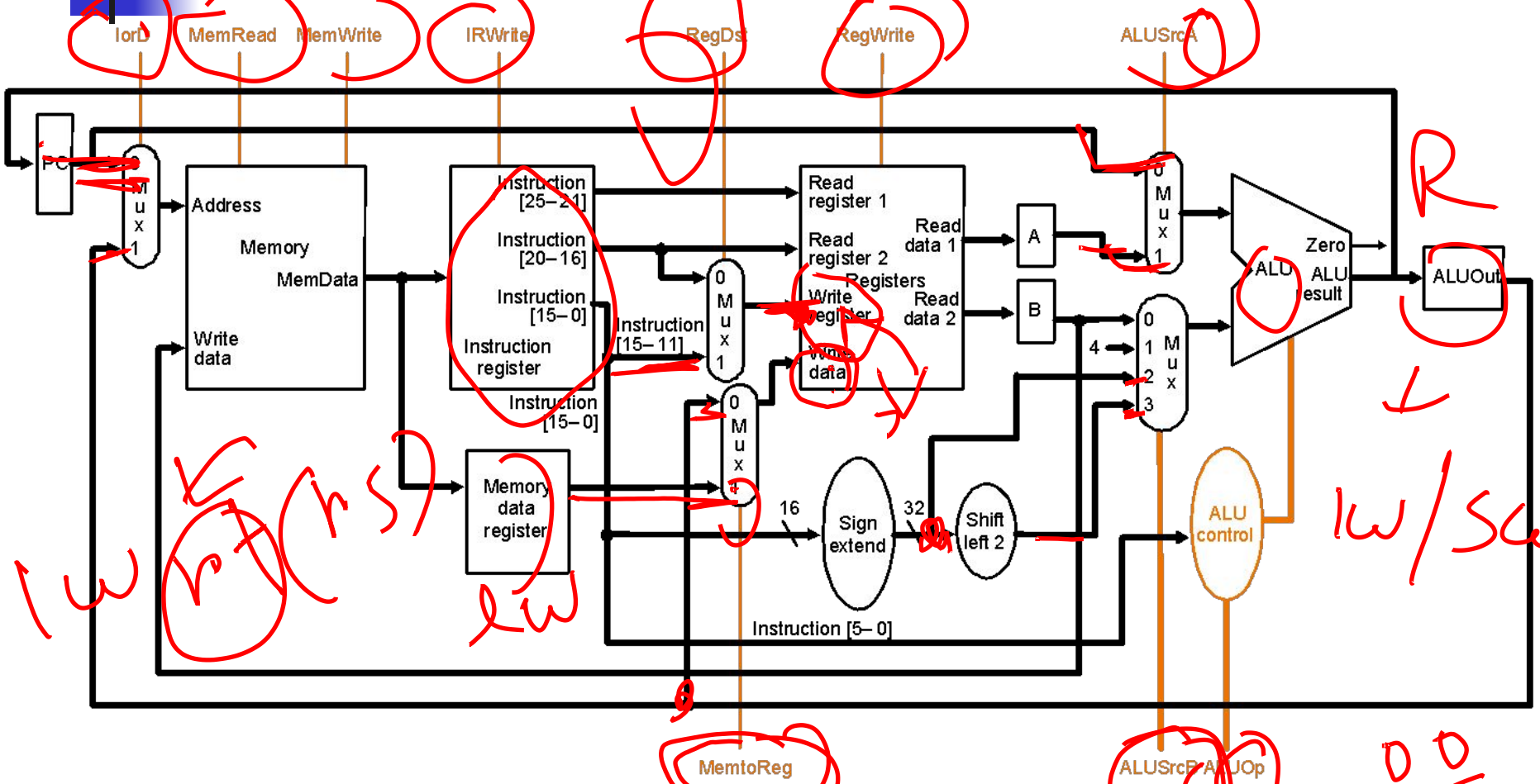


Multicycle Execution Step (5): Memory Read Completion (lw)

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

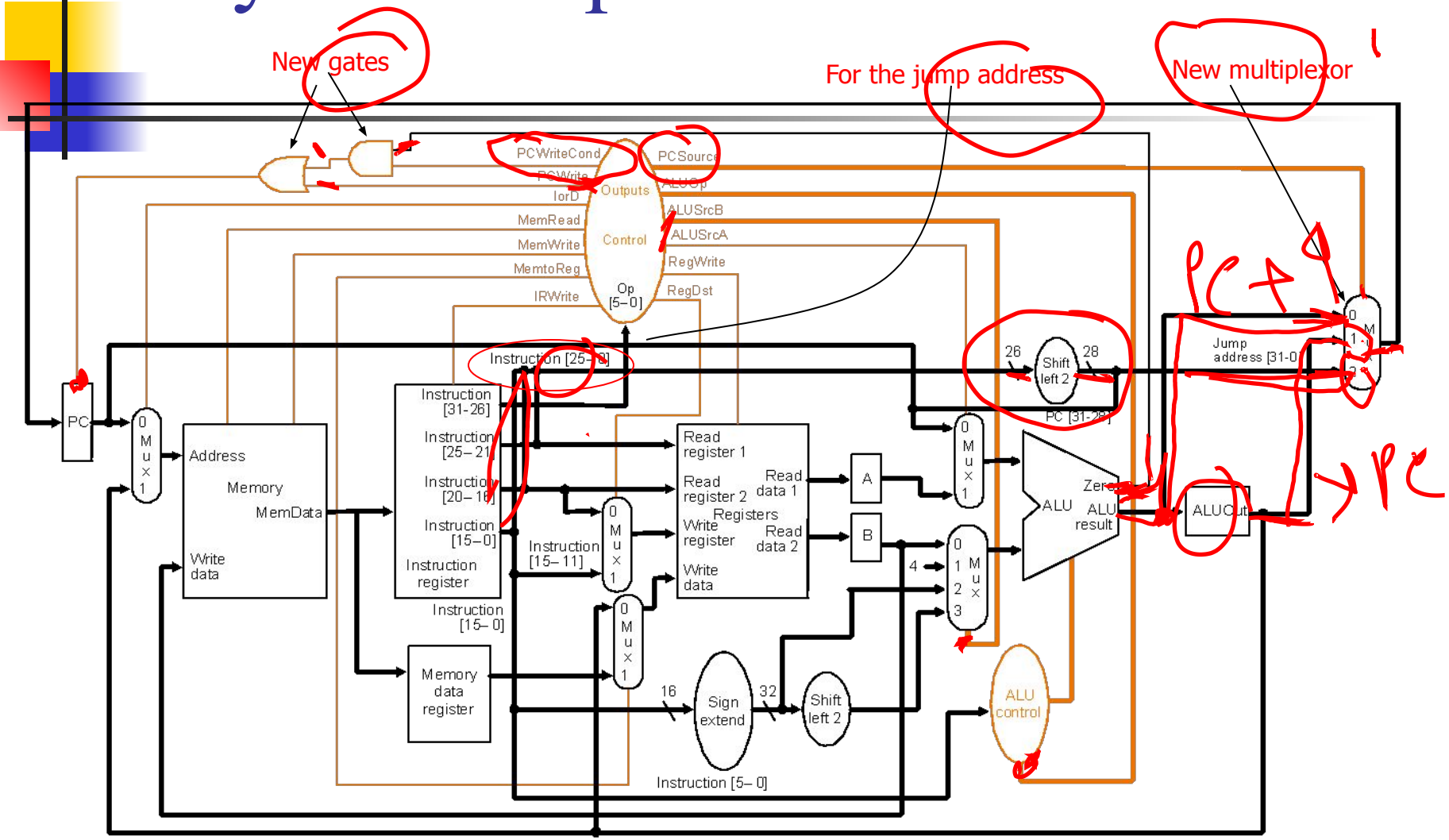


Multicycle Datapath with Control I



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

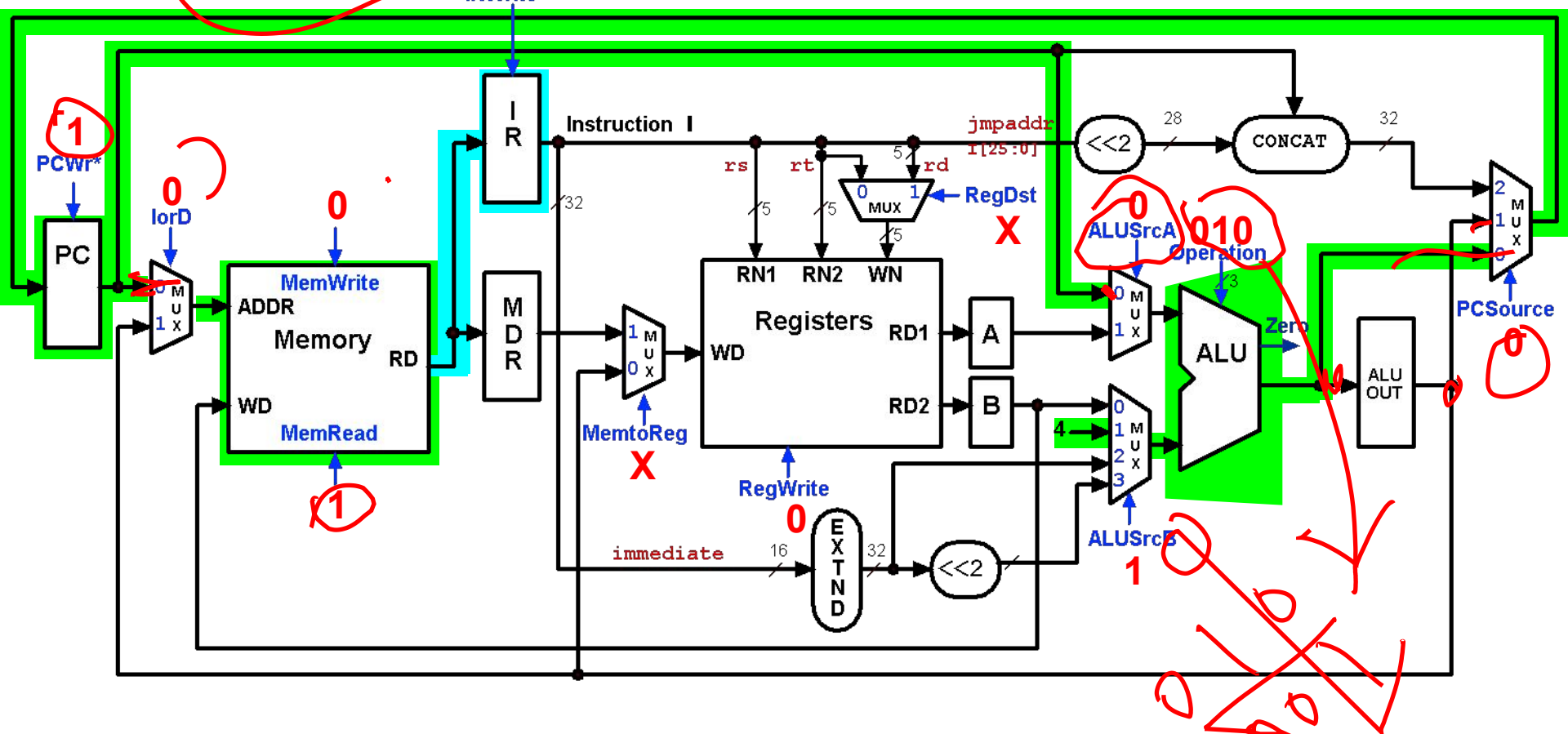
Multicycle Datapath with Control II



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

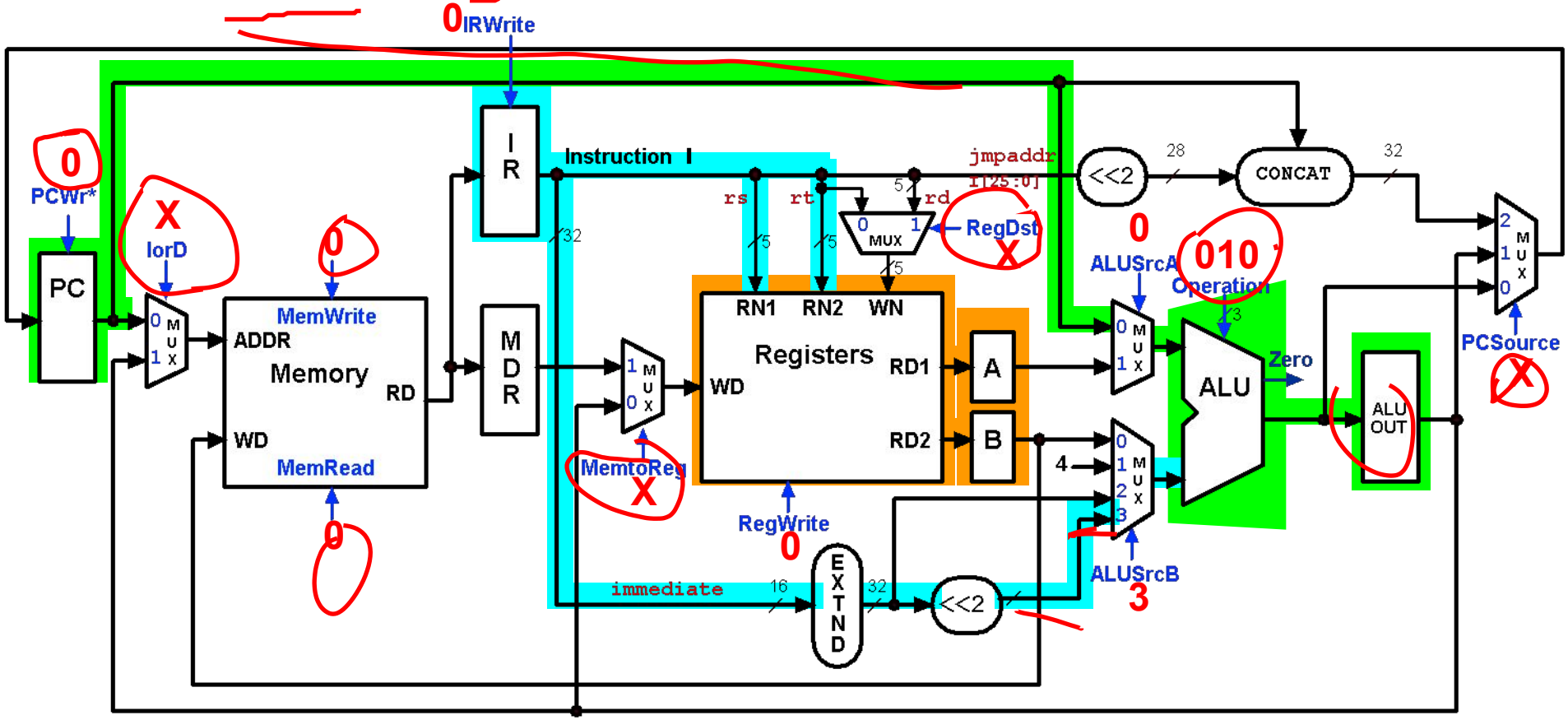
Multicycle Control Step (1): Fetch

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



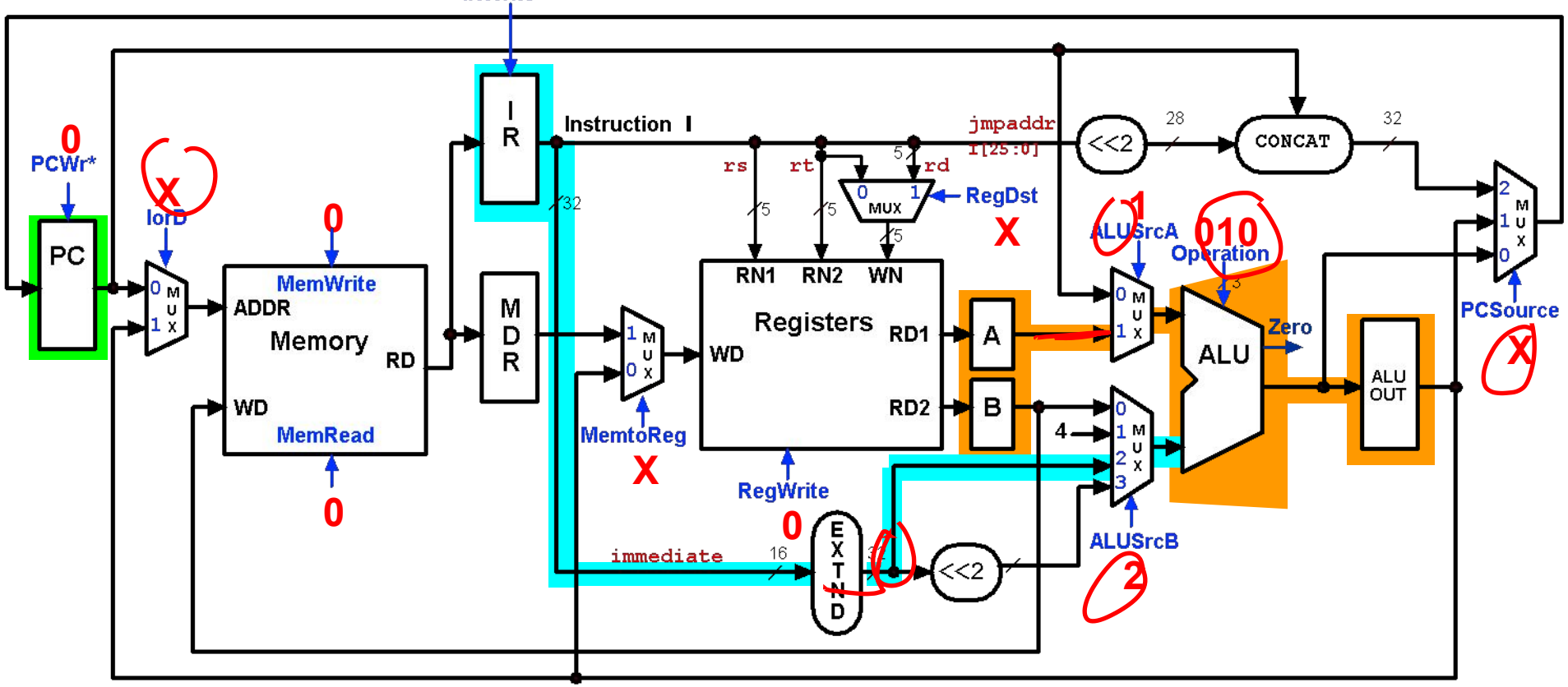
Multicycle Control Step (2): Instruction Decode & Register Fetch

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



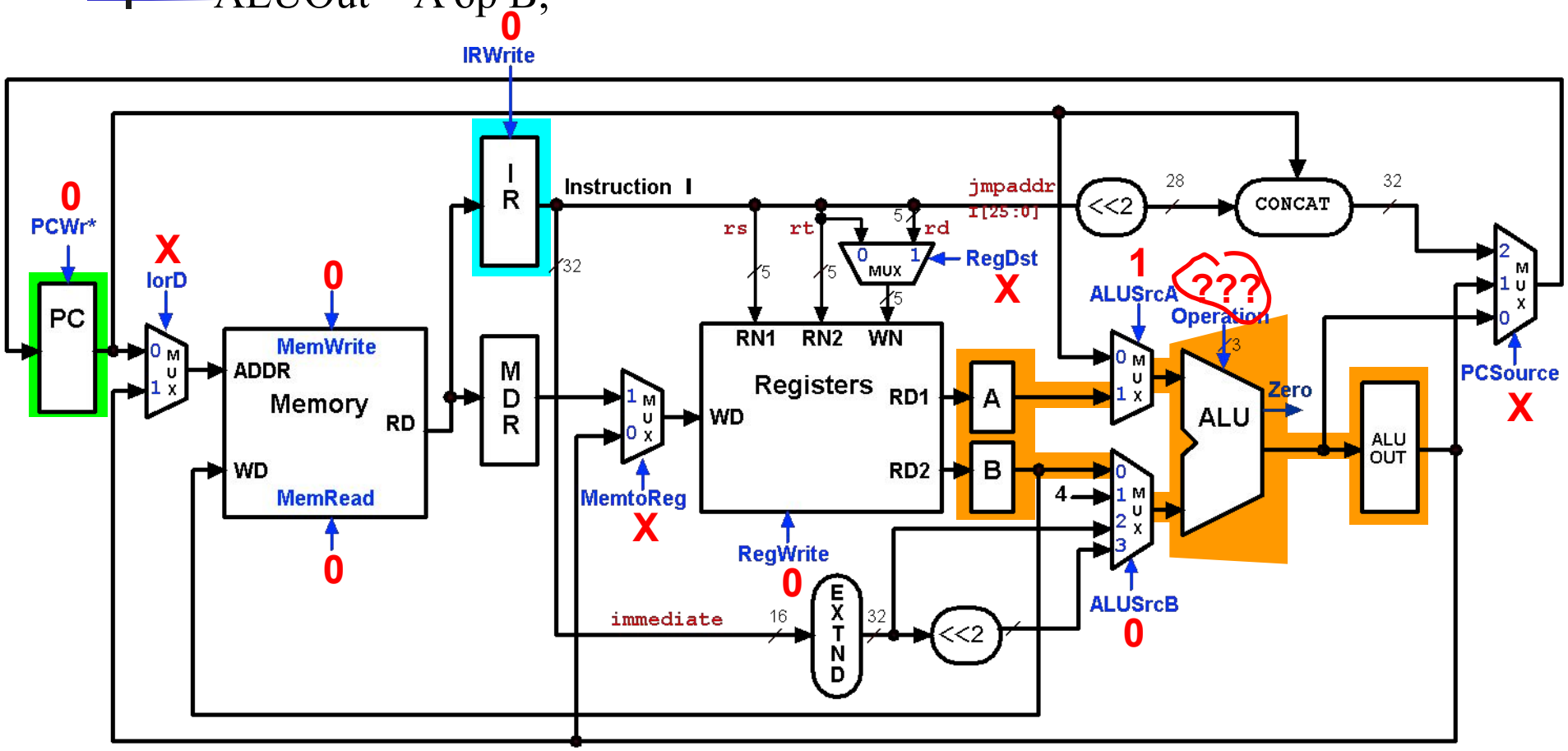
Multicycle Control Step (3): Memory Reference Instructions

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



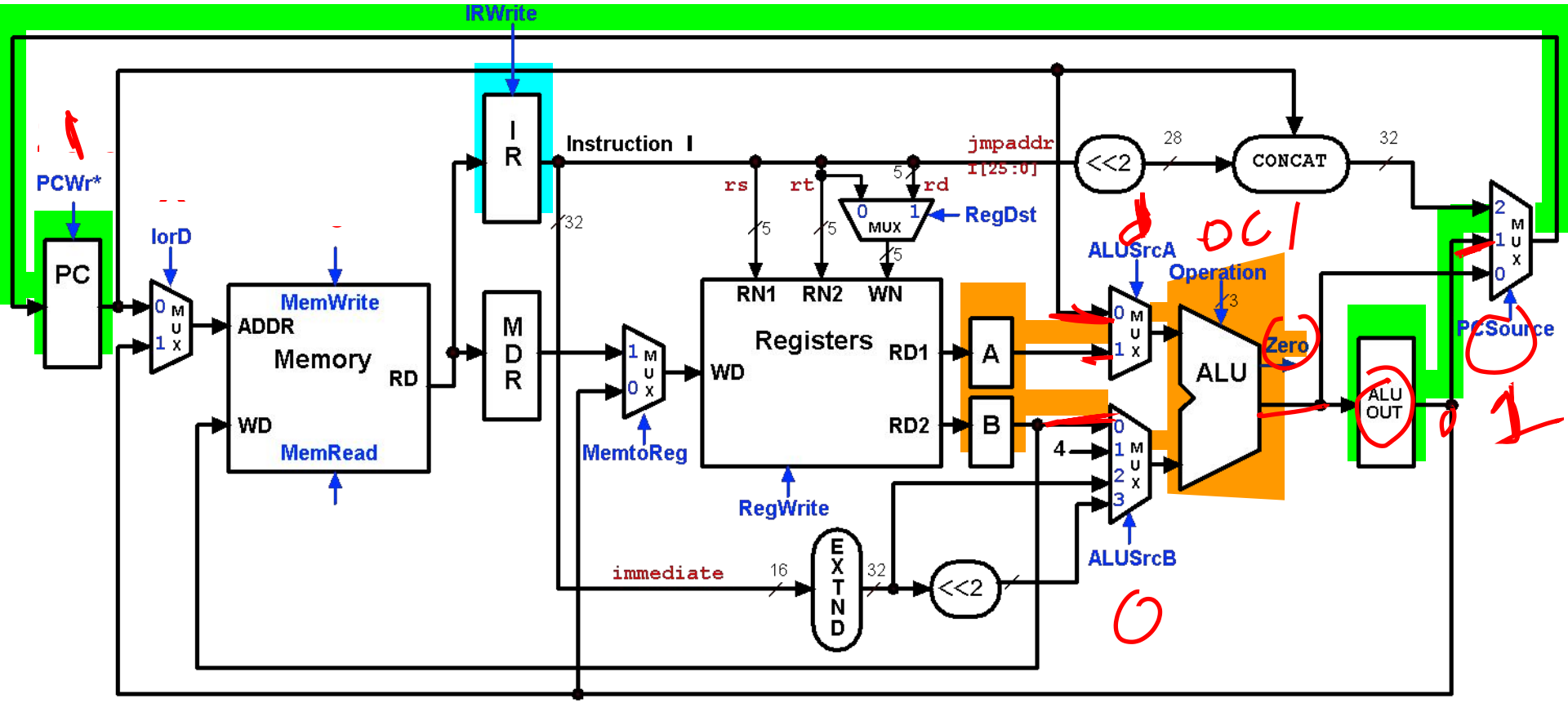
Multicycle Control Step (3): ALU Instruction (R-Type)

ALUOut = A op B;



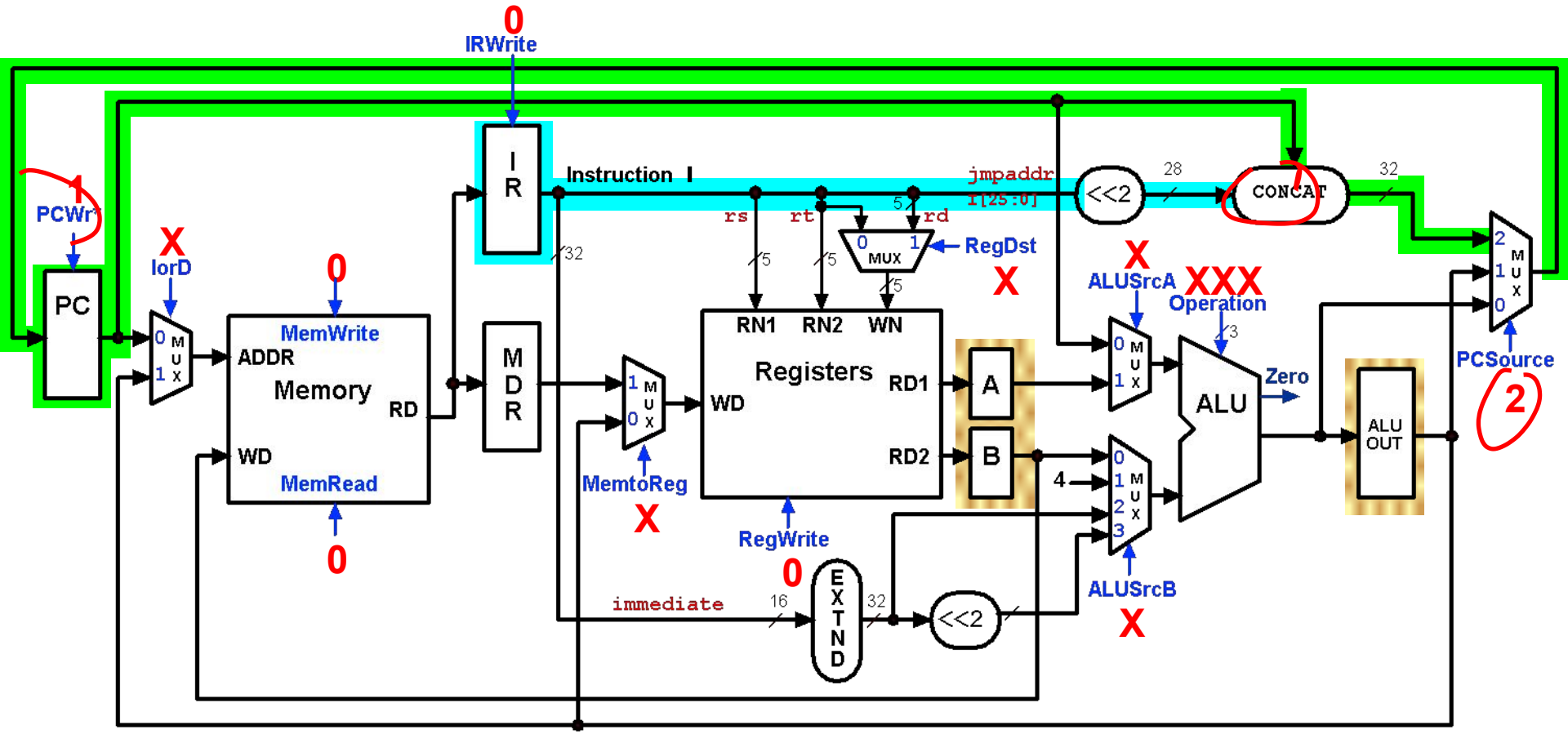
Multicycle Control Step (3): Branch Instructions

if (A == B) PC ALUOut;



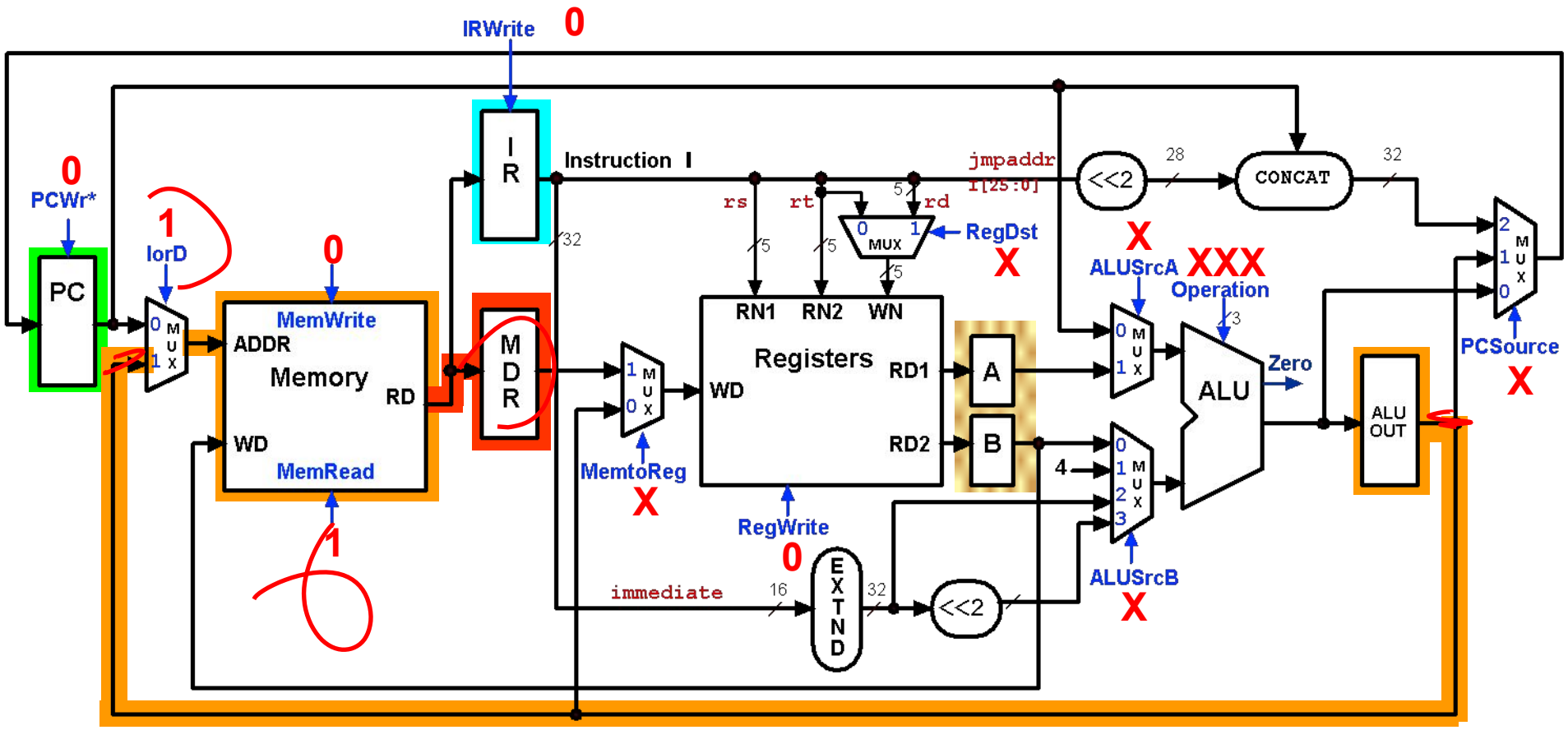
Multicycle Execution Step (3): Jump Instruction

PC = PC[21-28] concat (IR[25-0] << 2);



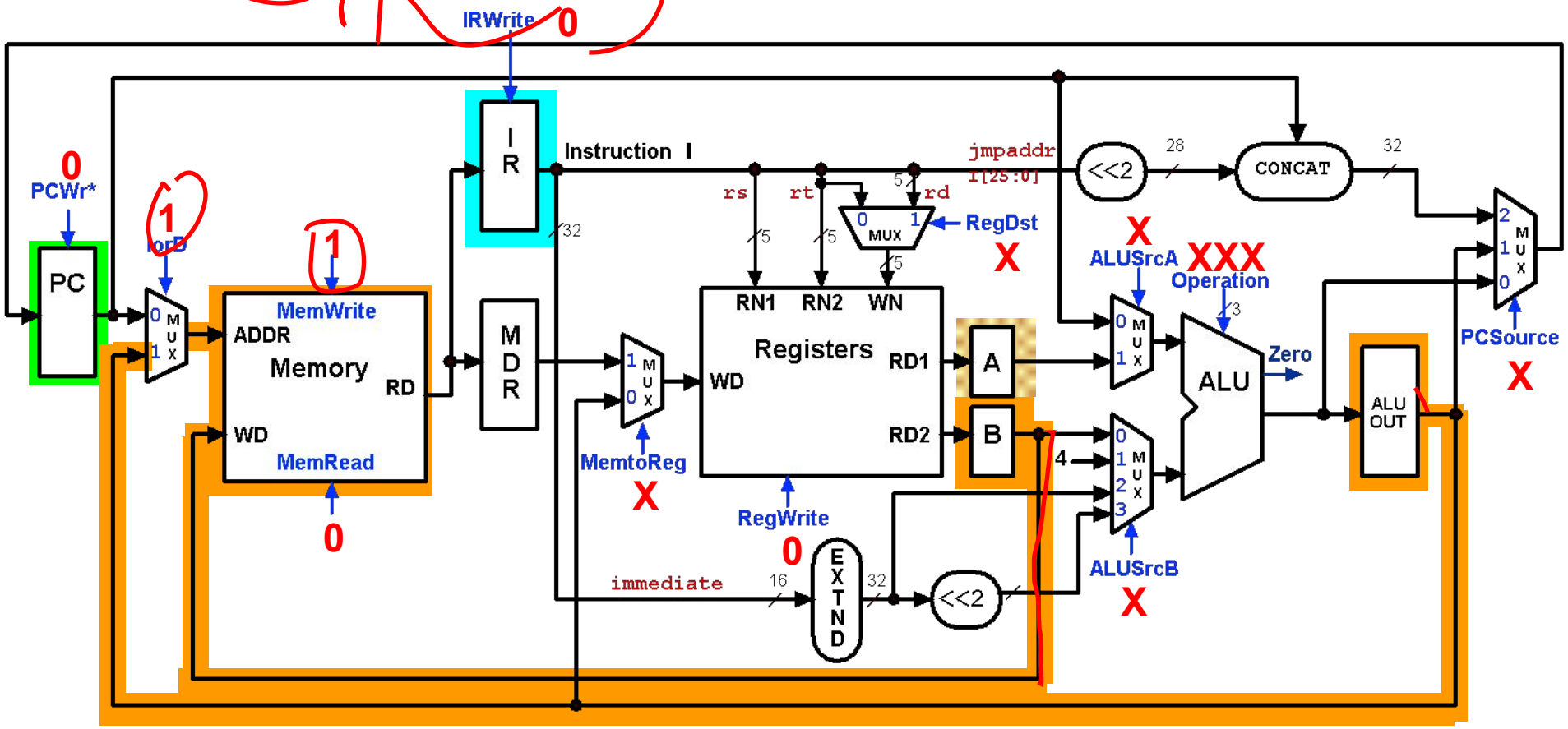
Multicycle Control Step (4): Memory Access - Read (lw)

MDR = Memory[ALUOut];



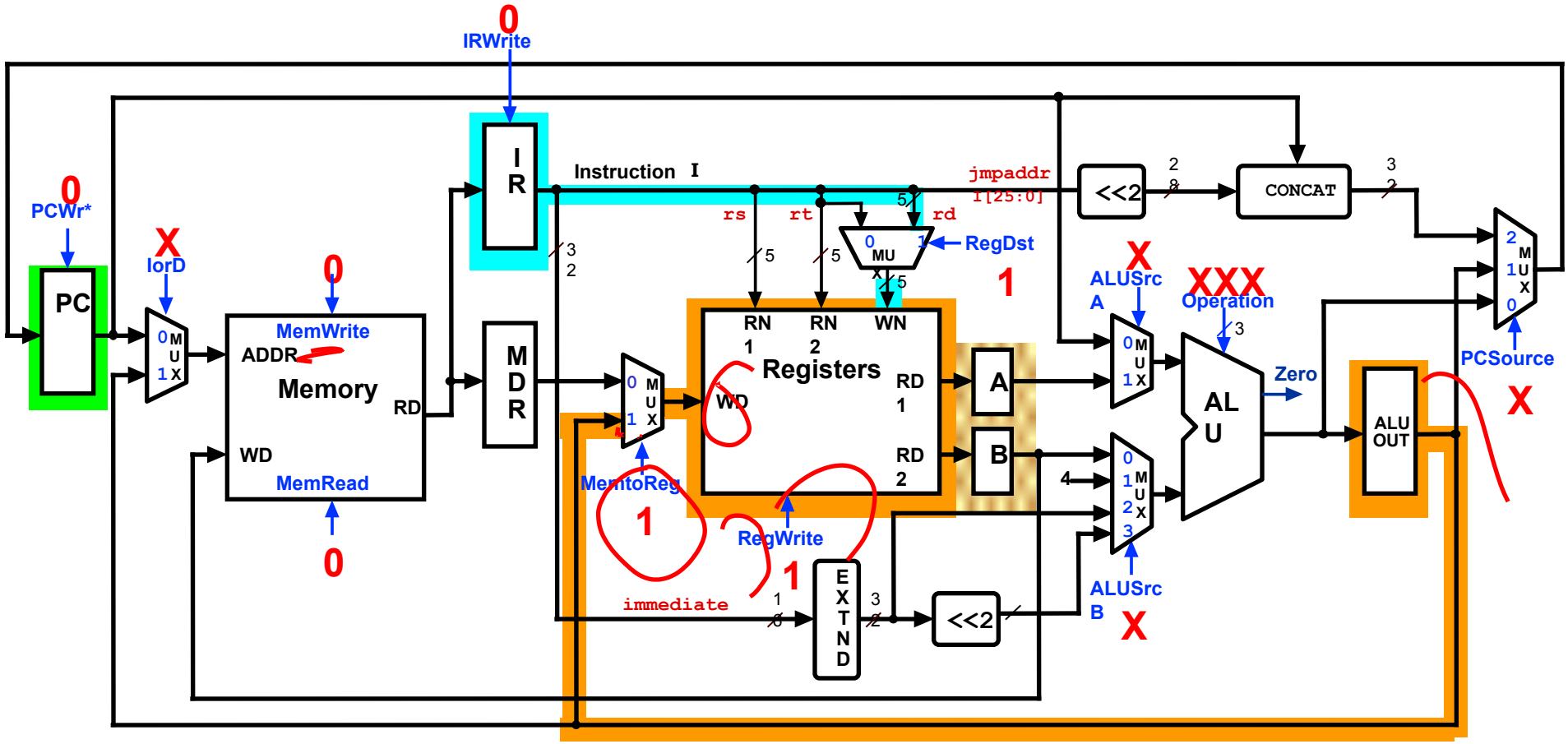
Multicycle Execution Steps (4)
Memory Access - Write (sw)

Memory[ALUOut] = B;



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

Reg[IR[15:11]] = ALUOut; (Reg[Rd] = ALUOut)



Multicycle Execution Steps (5)
Memory Read Completion (lw)

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