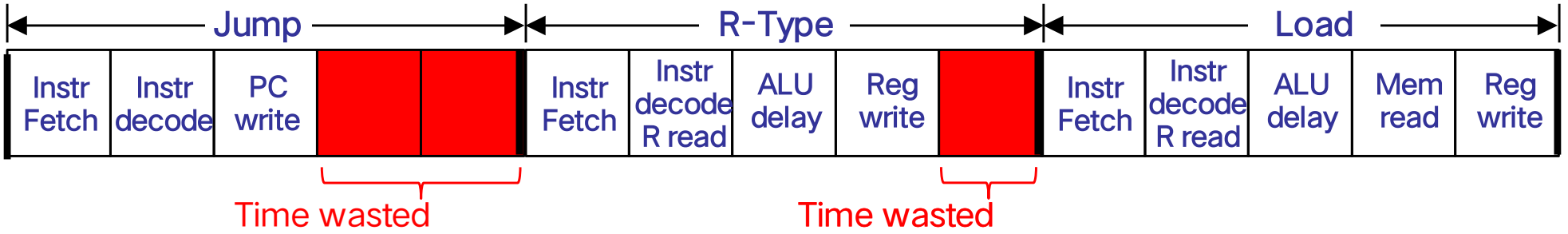
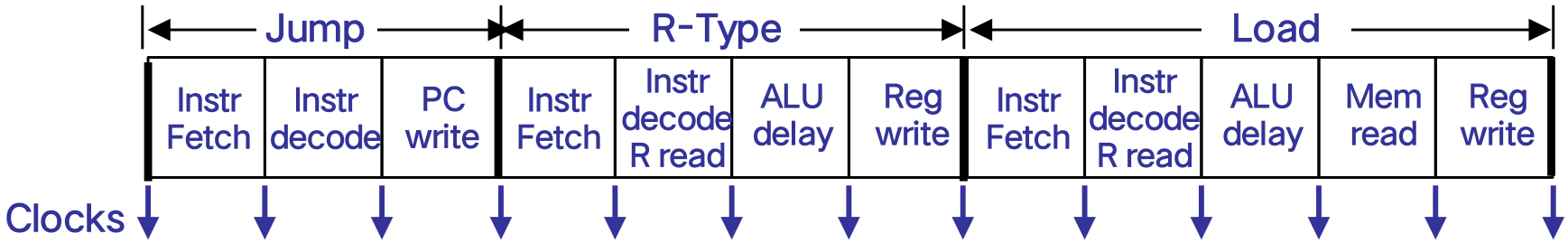
Multi Cycle Processor

Single Cycle Processor

- Not utilized in modern designs
  - Inefficient
  - Clock cycle must have the same length for every instruction
  - Determined by the longest possible path in the machine
  - Several of the instructions could fit in a shorter clock cycle



Multi Cycle Processor

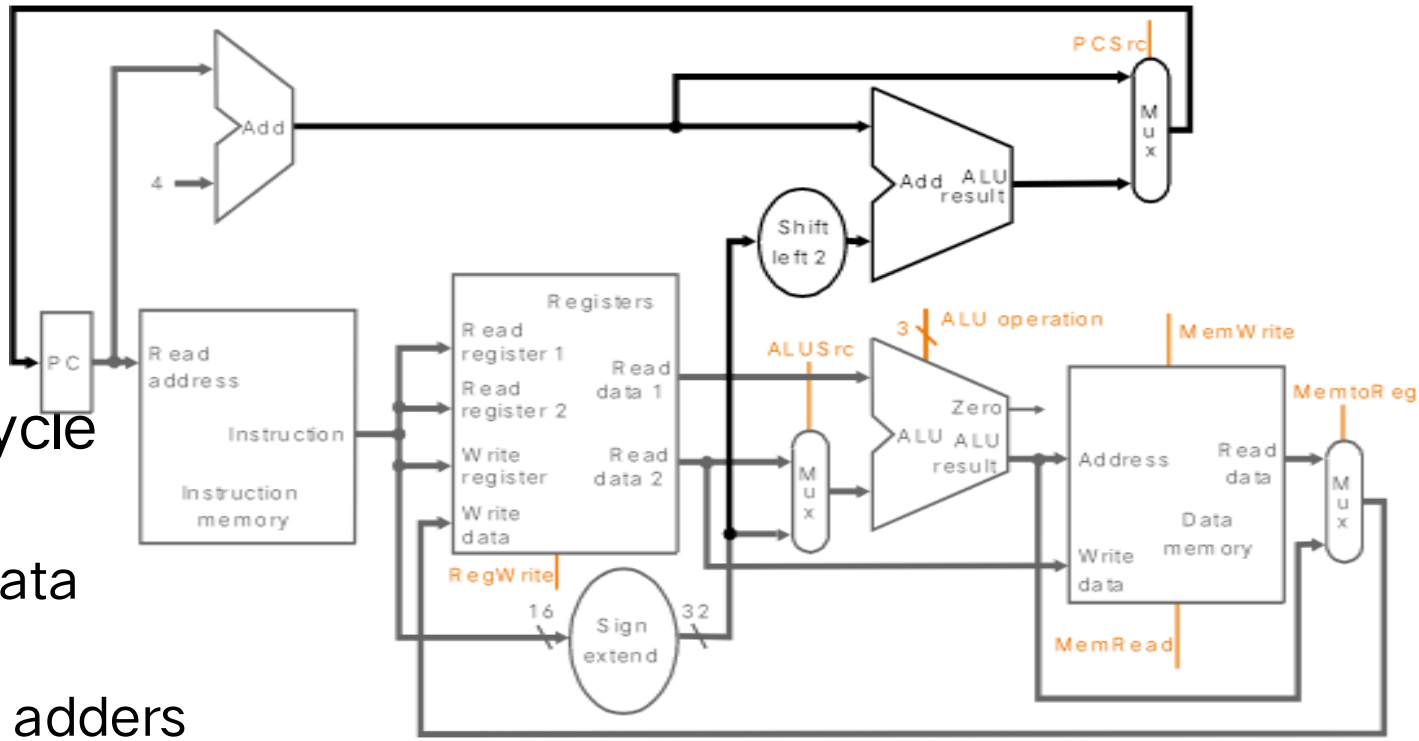


Multi Cycle Approach

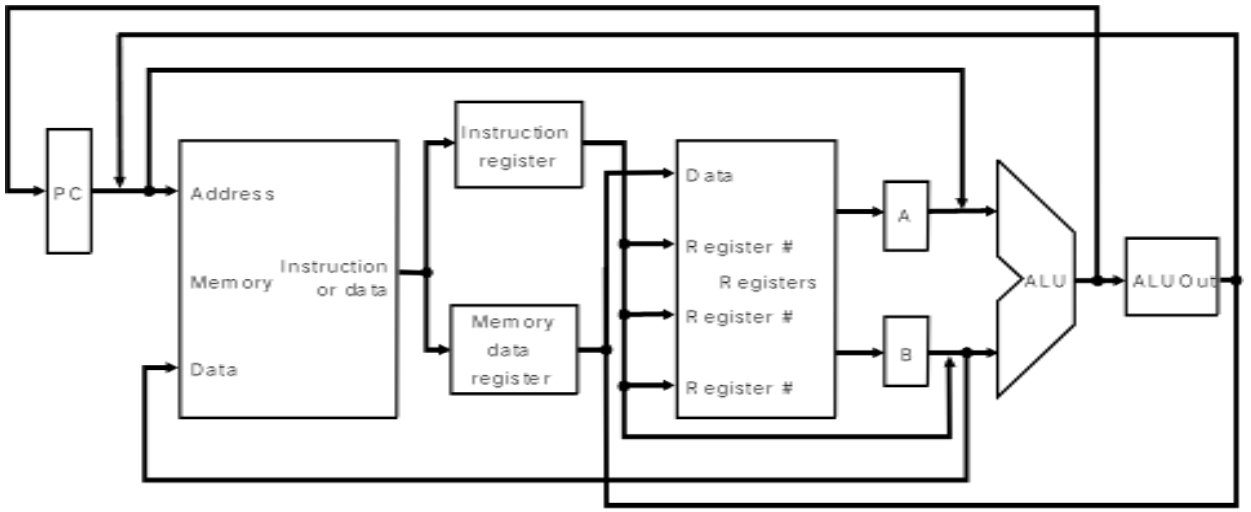
- Break up the instructions into *steps - GOAL*
  - each step takes one clock cycle
  - balance the amount of work to be done in each step/cycle so that they are about equal
  - restrict each cycle to use at most once each major functional unit so that such units do not have to be replicated
  - functional 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

Multicycle Approach

- Multicyle vs. single-cycle diagrams
  - single memory for data and instructions
  - single ALU, no extra adders
  - extra registers to hold data between clock cycles

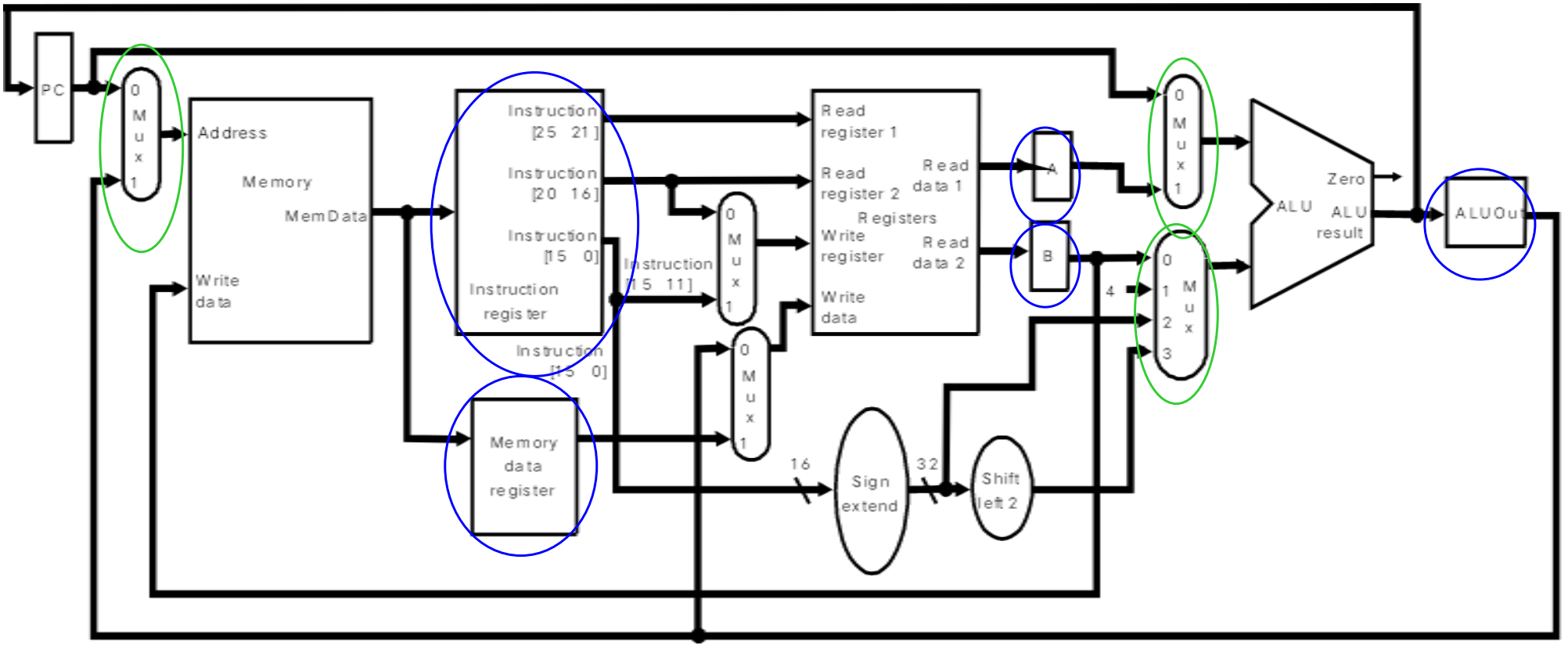


Single-cycle datapath



Multicycle datapath (high-level view)

Multicycle Datapath



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

Breaking instructions into steps

- Break instructions into the *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.

- Micro operations using *RTL* (*Register-Transfer Language*):

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

Step 2: Instruction Decode and Register Fetch (ID)

- Read registers rs and rt **if needed**
- Compute the branch address **if 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: Need of this step:

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

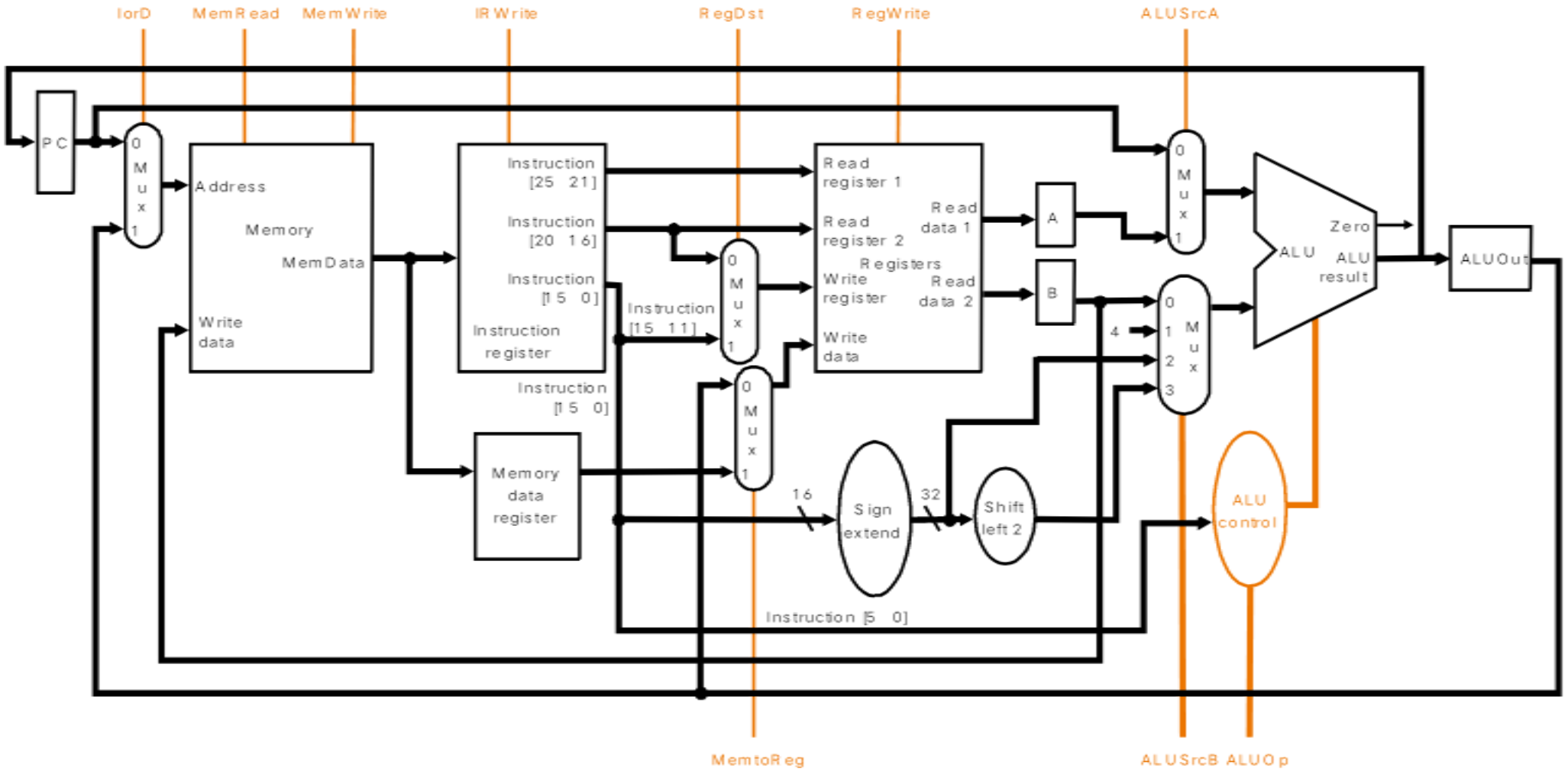
This would eliminate the MDR as well.

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

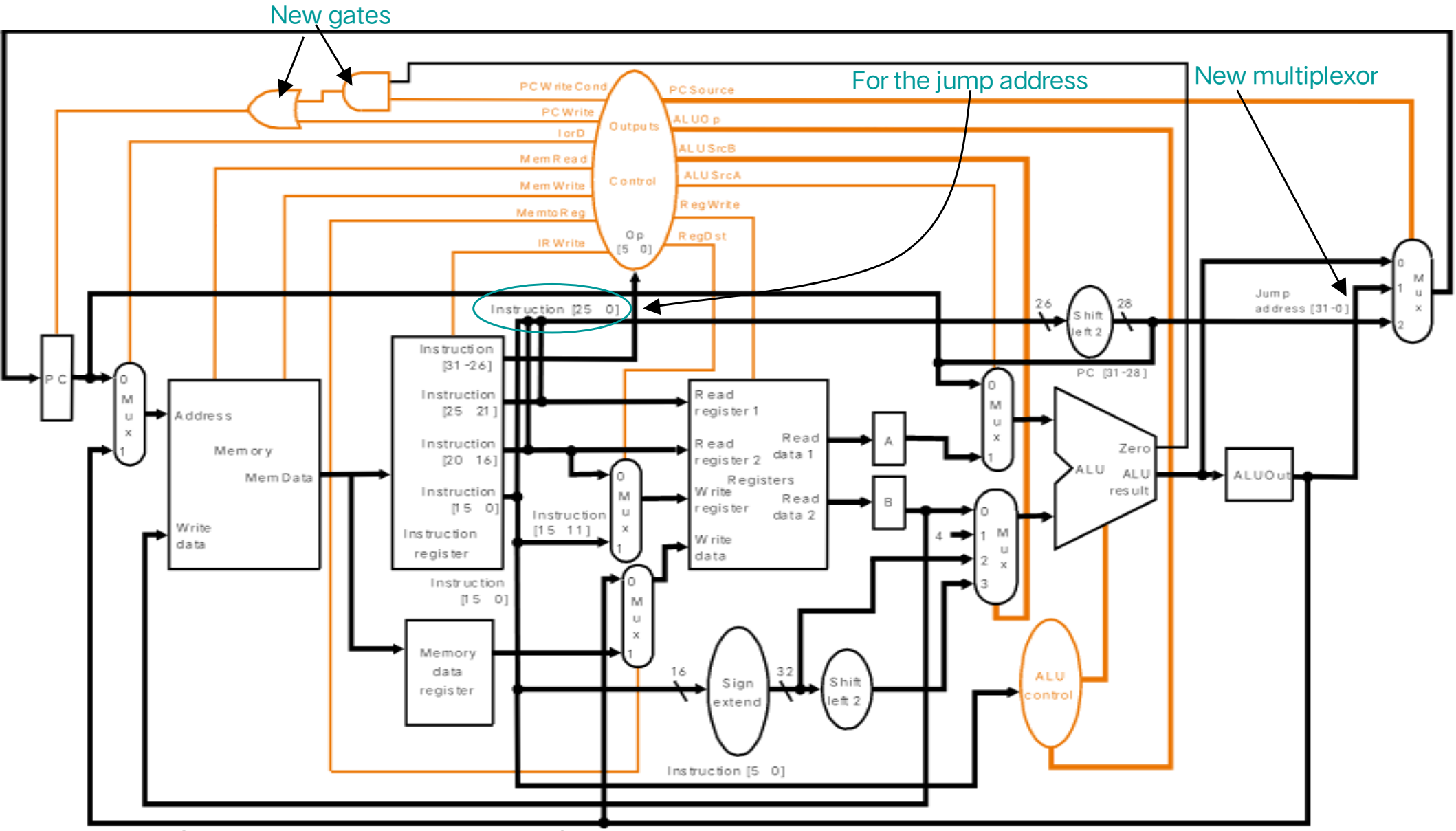
Step	Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
1: IF	Instruction fetch	IR = Memory[PC]			
		PC = PC + 4			
2: ID	Instruction	A = Reg [IR[25-21]]			
	decode/register fetch	B = Reg [IR[20-16]]			
		ALUOut = PC + (sign-extend (IR[15-0])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

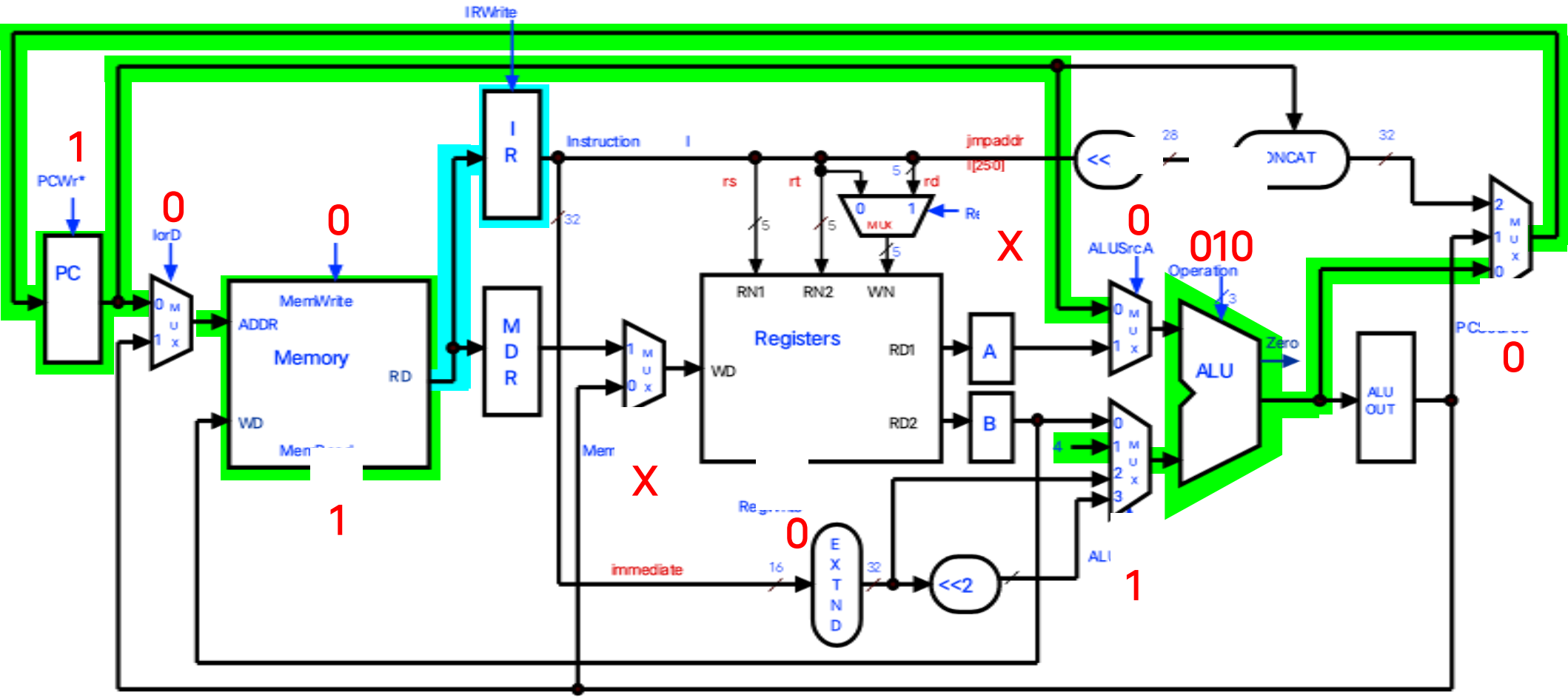
Multicycle Datapath with Control II



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

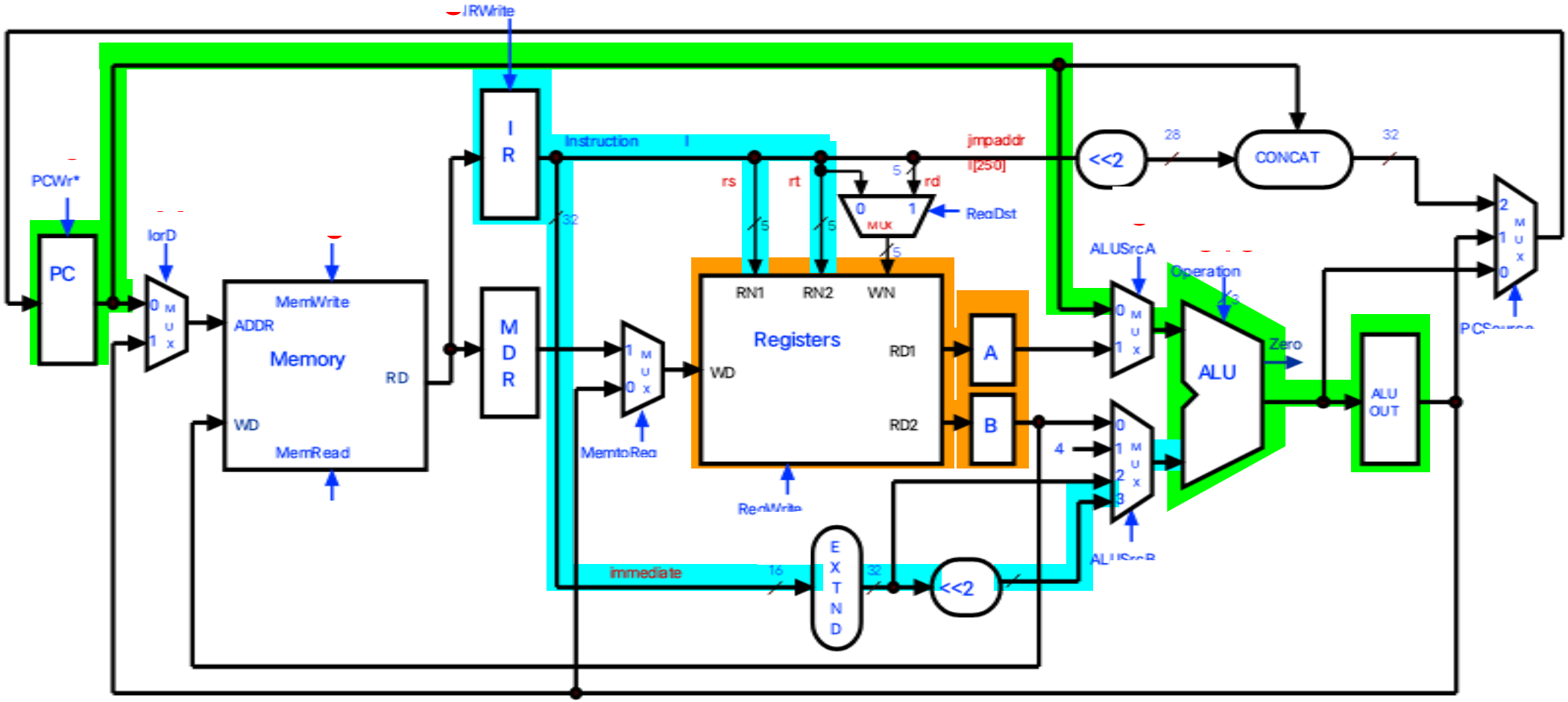
Multicycle Control Step (1): Fetch

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



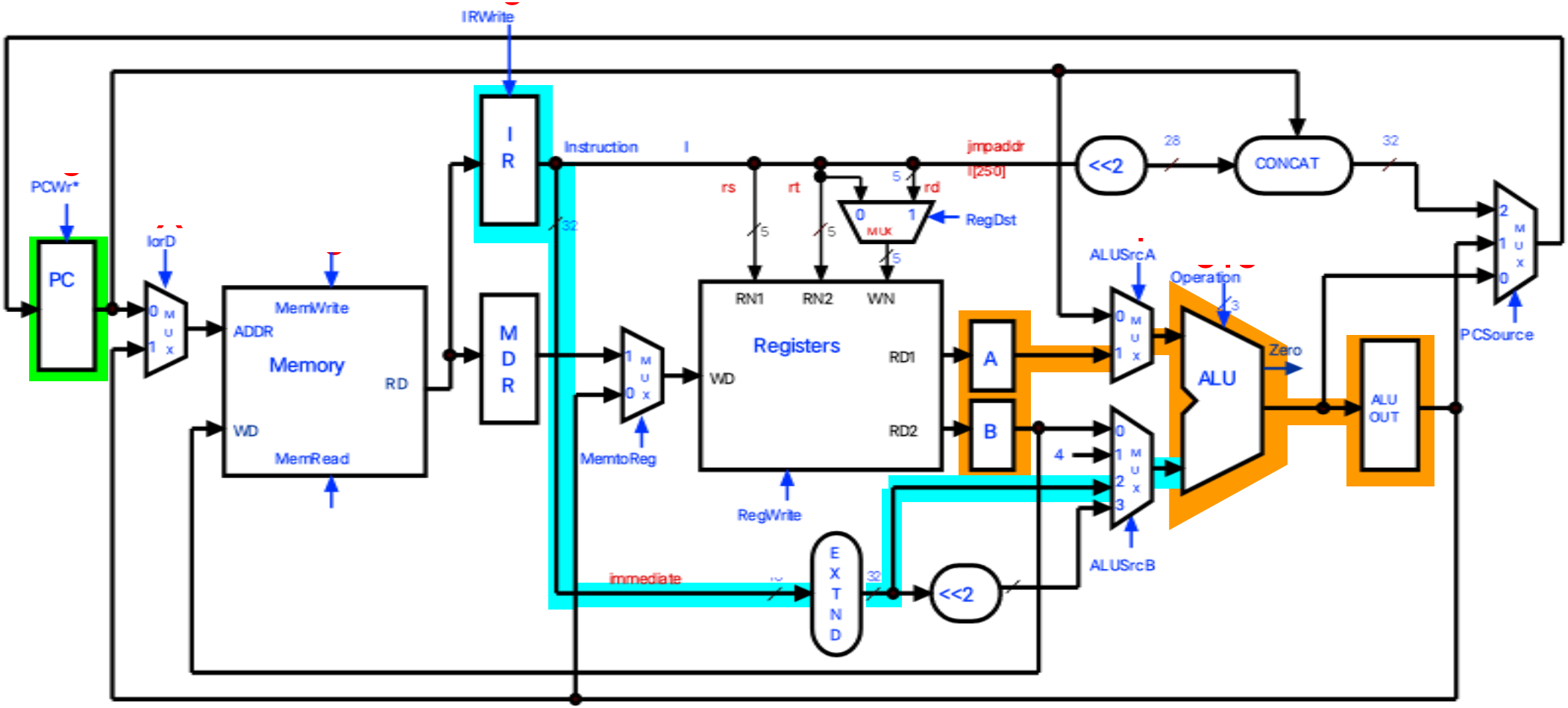
Multicycle Control Step (2): Instruction Decode & Register Fetch

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



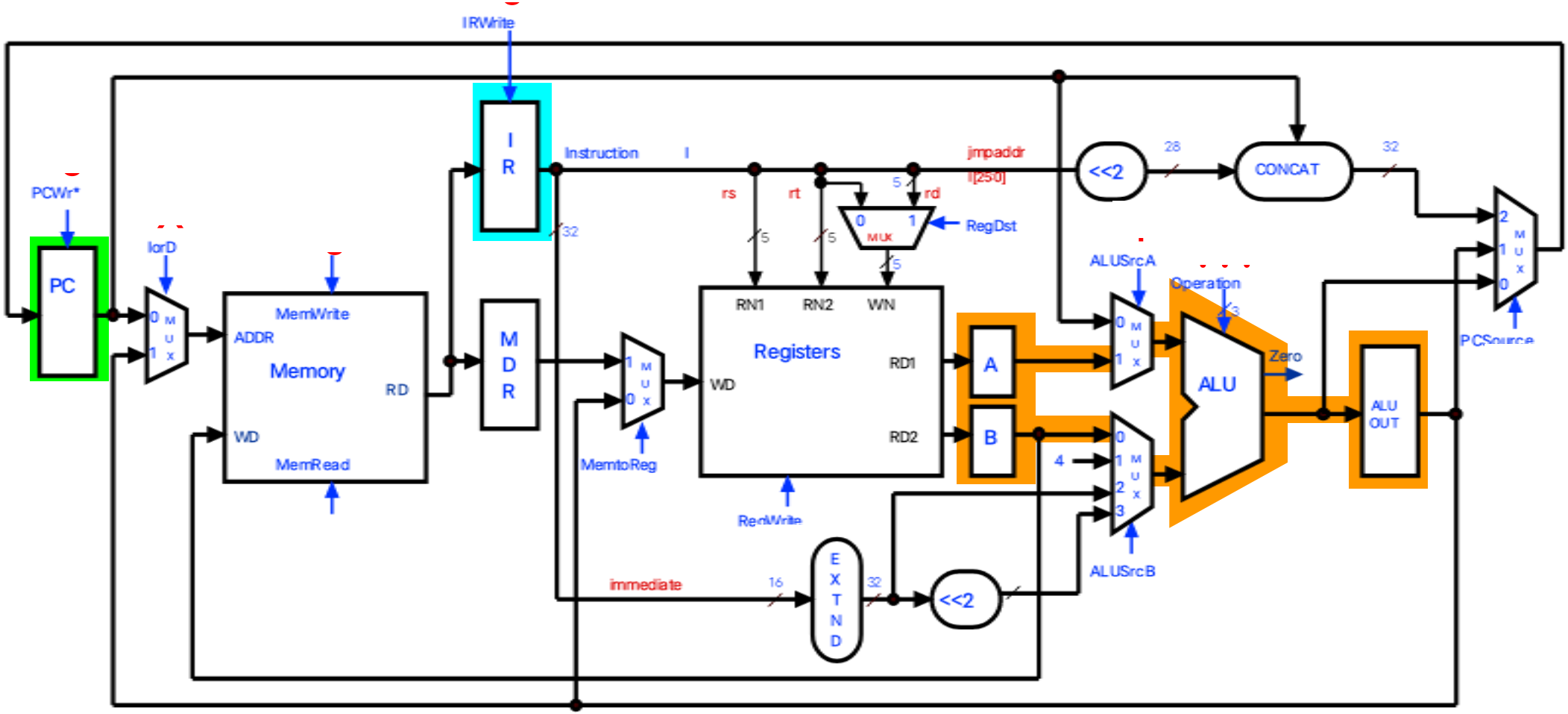
Multicycle Control Step (3): Memory Reference Instructions

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



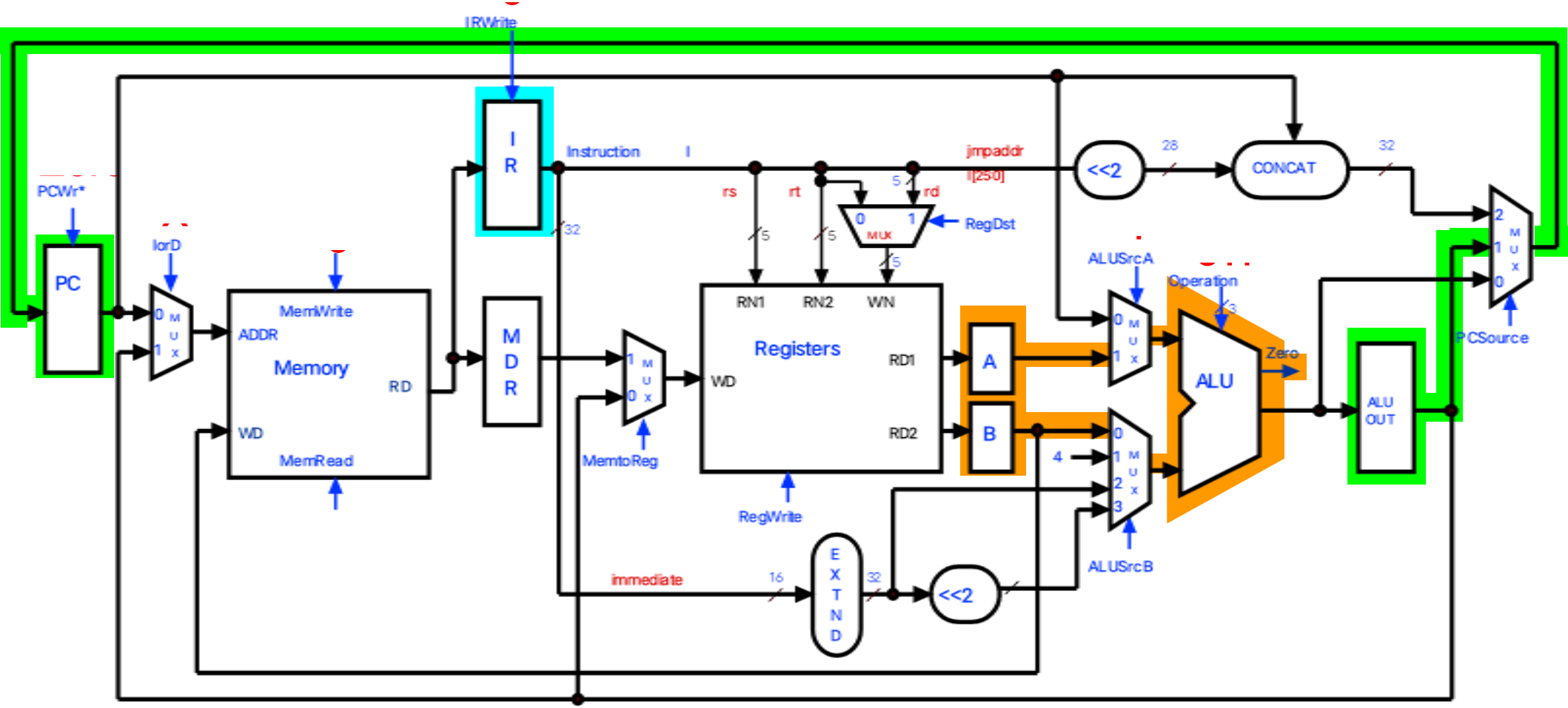
Multicycle Control Step (3): ALU Instruction (R-Type)

ALUOut = A op B;



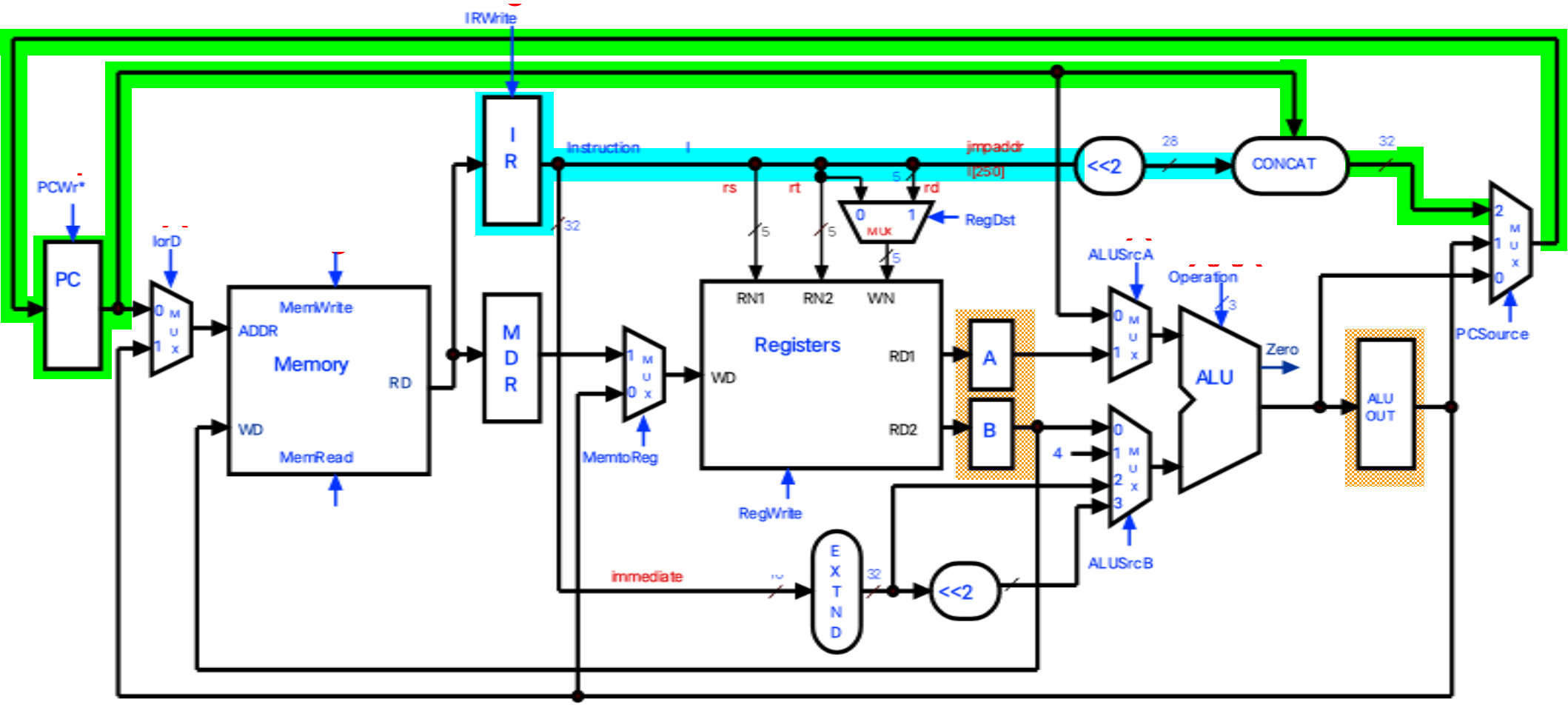
Multicycle Control Step (3): Branch Instructions

if (A == B) PC ← ALUOut;



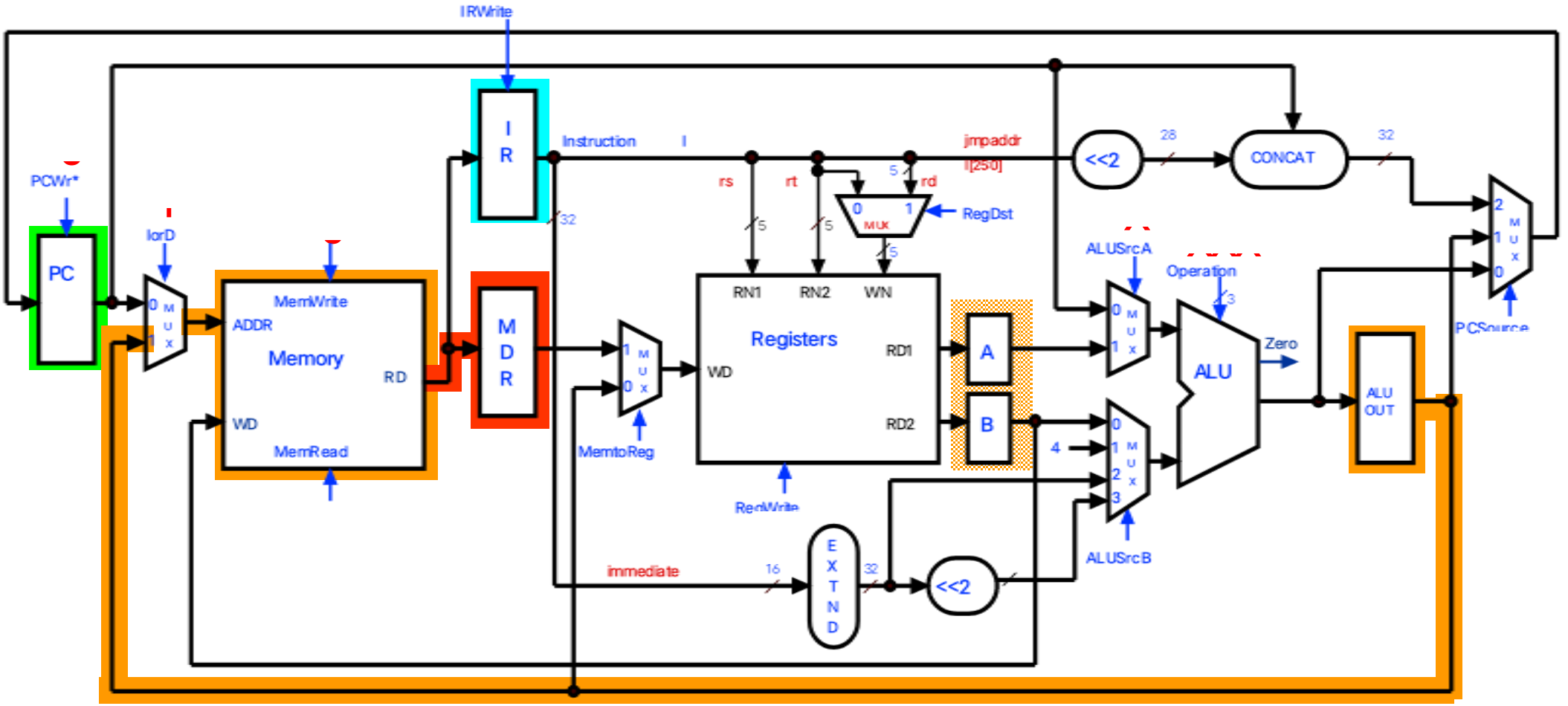
Multicycle Execution Step (3): Jump Instruction

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



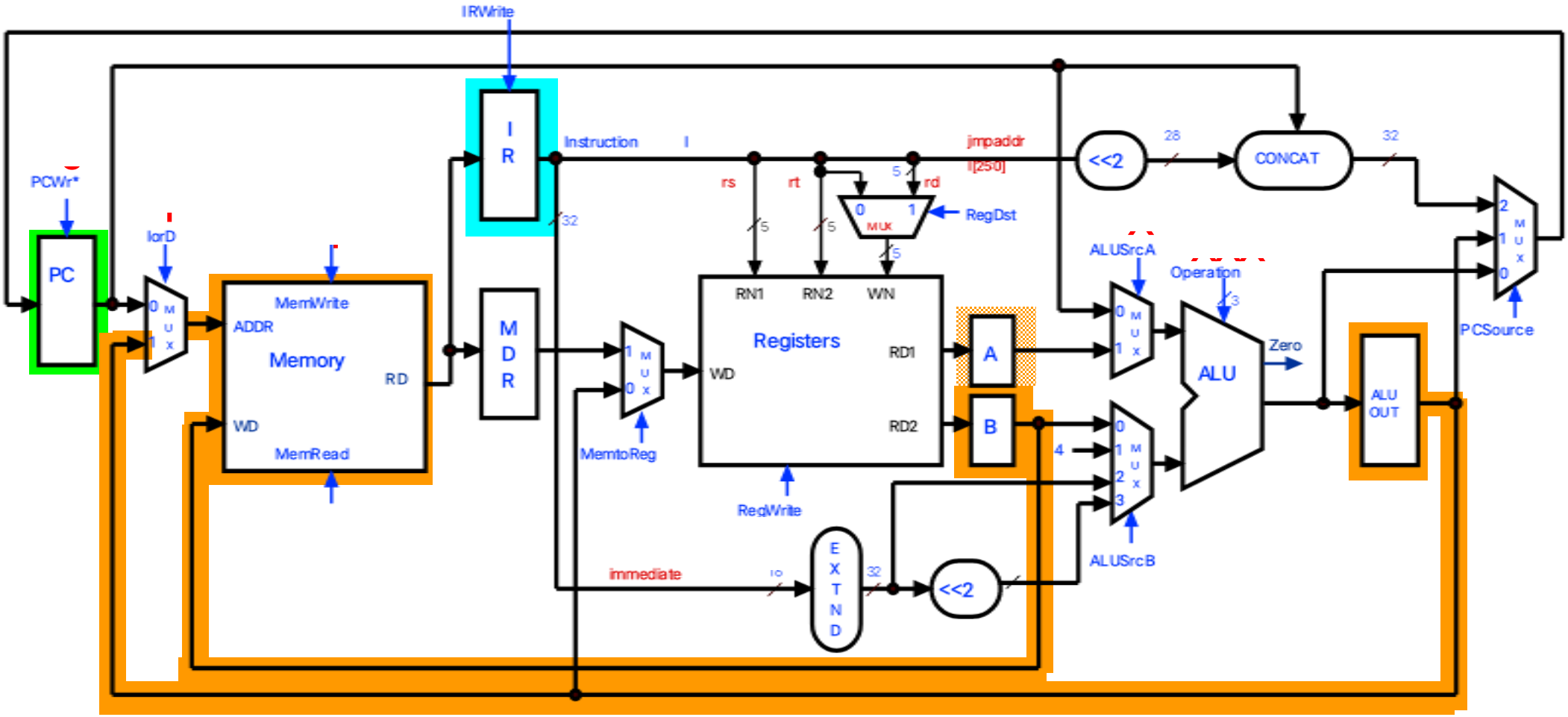
Multicycle Control Step (4): Memory Access - Read (lw)

MDR = Memory[ALUOut];



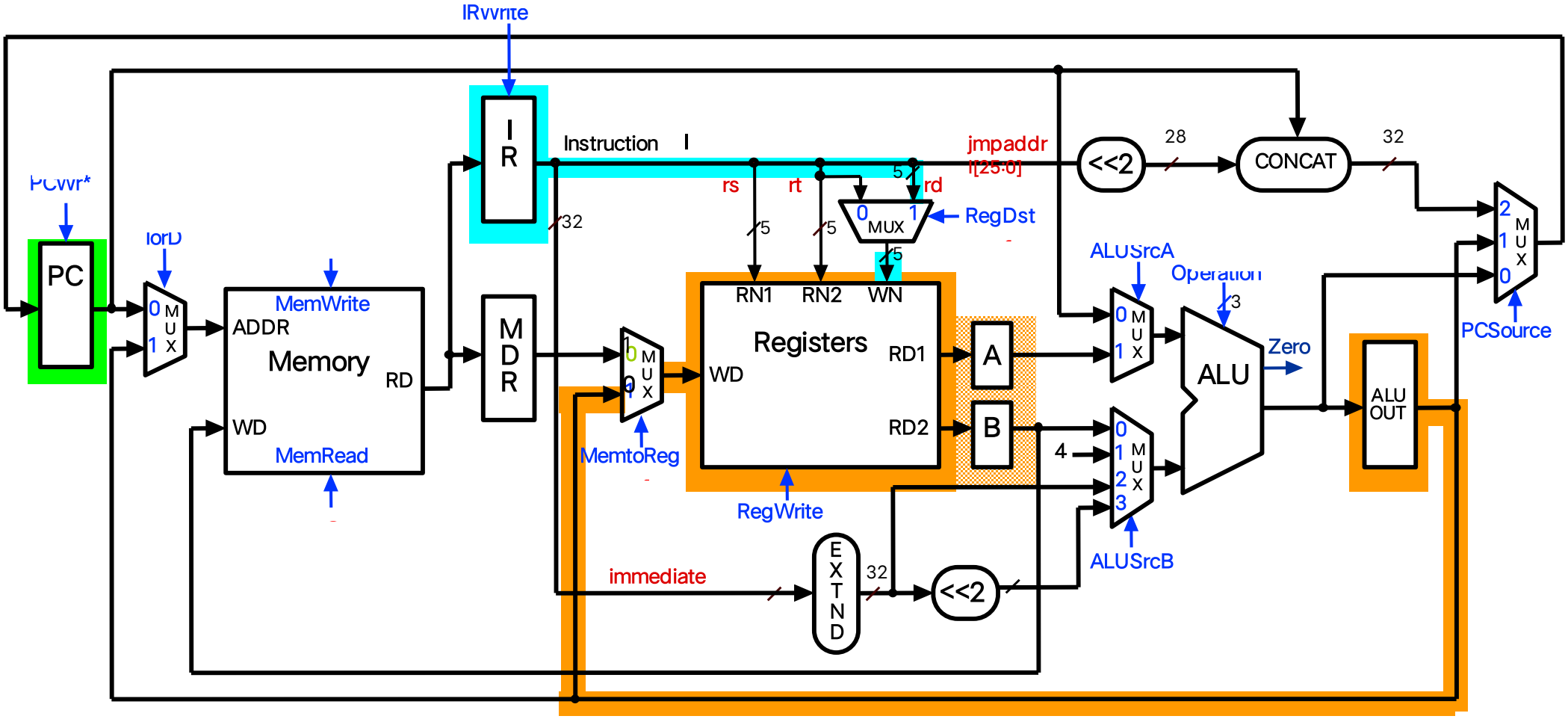
Multicycle Execution Steps (4)
Memory Access - Write (sw)

Memory[ALUOut] = B;



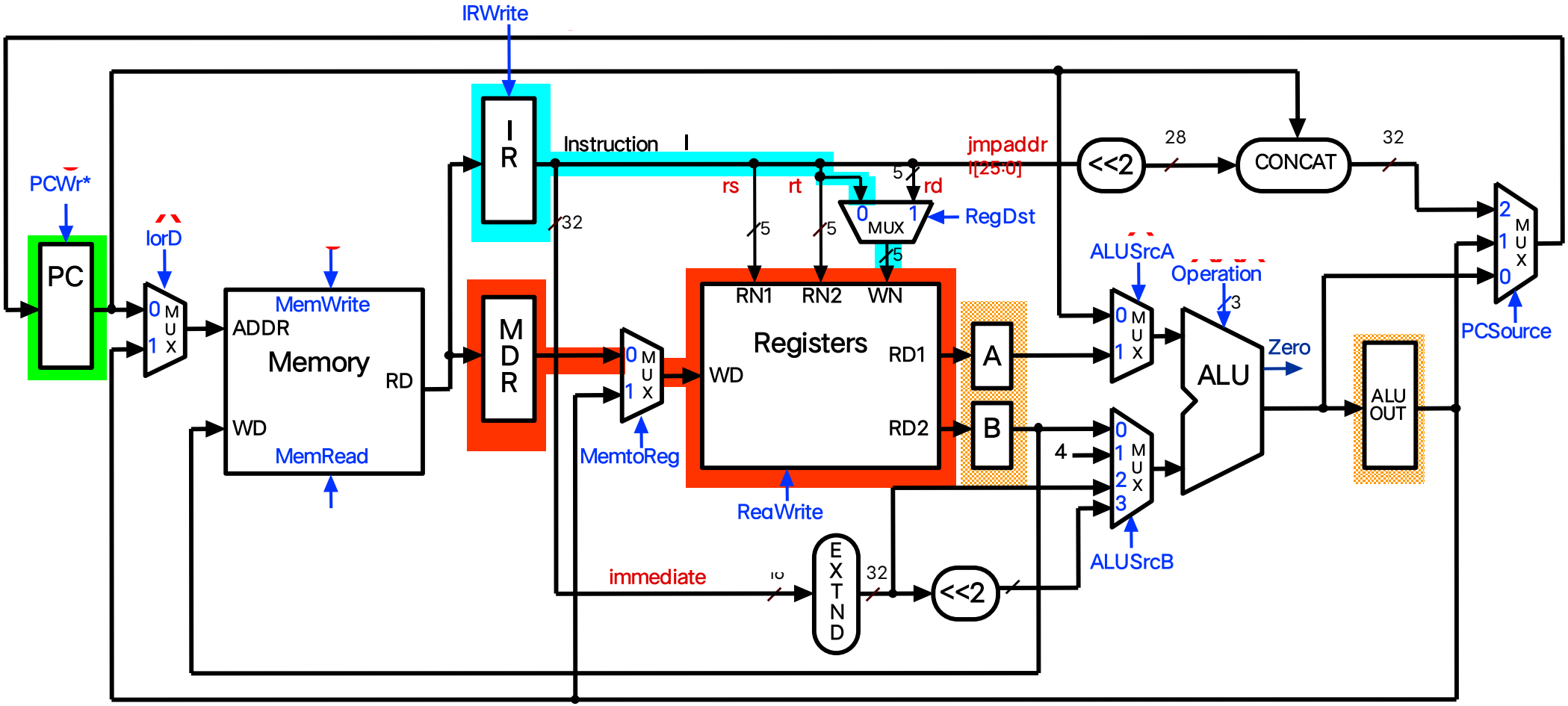
Multicycle Control Step (4): ALU Instruction (R-Type)

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



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

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



Tutorial 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?*

Implementing Control

- Up to this, we have determined what control signals are and when they must be asserted
  - So, now we can implement the CONTROL UNIT
- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed

Implementing Control

- Single Cycle CPU
  - We can use a set of truth tables that specify the settings of the controls signals based on the instruction class
- Multi Cycle CPU
  - The control is more complex
    - As the instruction is executed in a series of steps
  - We need to specify two things:
    - The signals to be set in any step
    - The next step in the sequence

Implementing Control

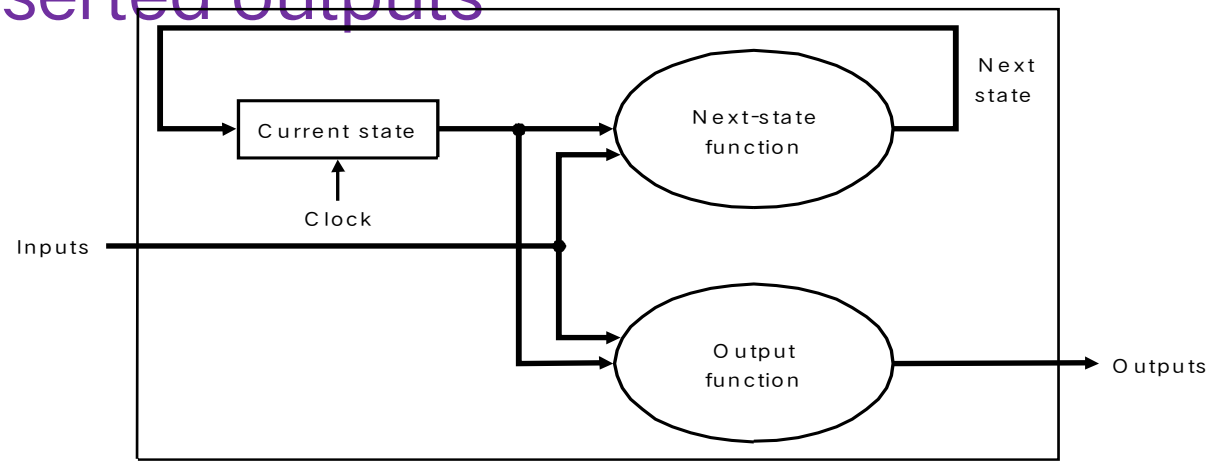
- Technique to specify the control
  - Finite State machine
    - Uses a graphical representation
  - Microprogramming
    - Uses a programming representation for control
- Both techniques use detailed implementation by gates, ROMs or PLAs

Implementing Control

- Use the information we have accumulated to specify a finite state machine
- Actual Hardware Implementation is then derived from the specification

Brief: Finite State Machines

- Finite state machines (FSMs):
  - A sequential logic function consisting of a set of inputs and outputs, a next state function that maps the current state and the inputs to a new state and an output function that maps the current state and possibly the inputs to a set of asserted outputs



  - *Moore machine*

Brief: Finite State Machines

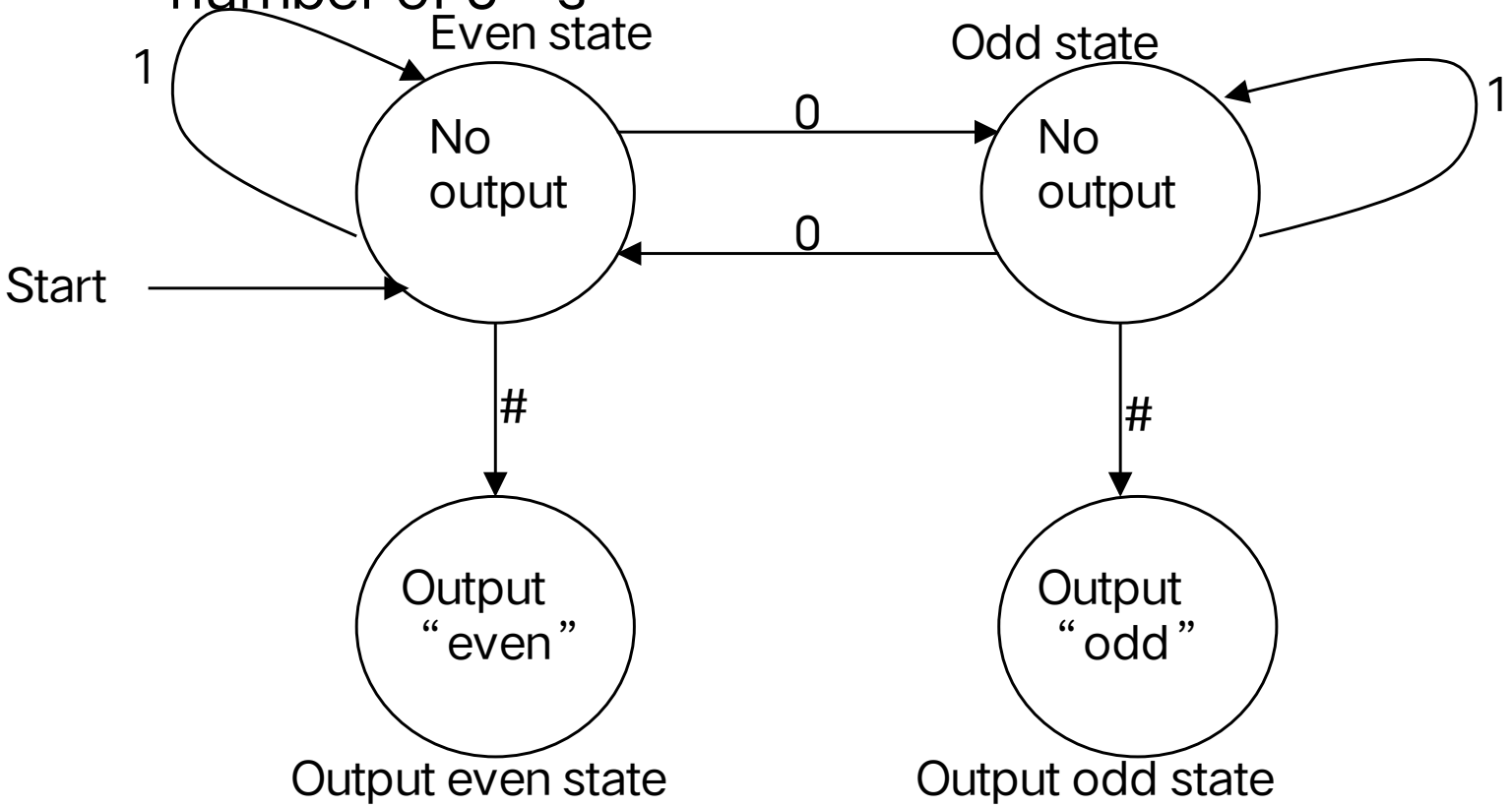
- Finite state machines (FSMs)
  - Each state also specifies a set of outputs that are asserted when the machine in that state
  - Assumes that all outputs that are not explicitly asserted are deasserted
    - No don’t care signals…

Example: Moore Machine

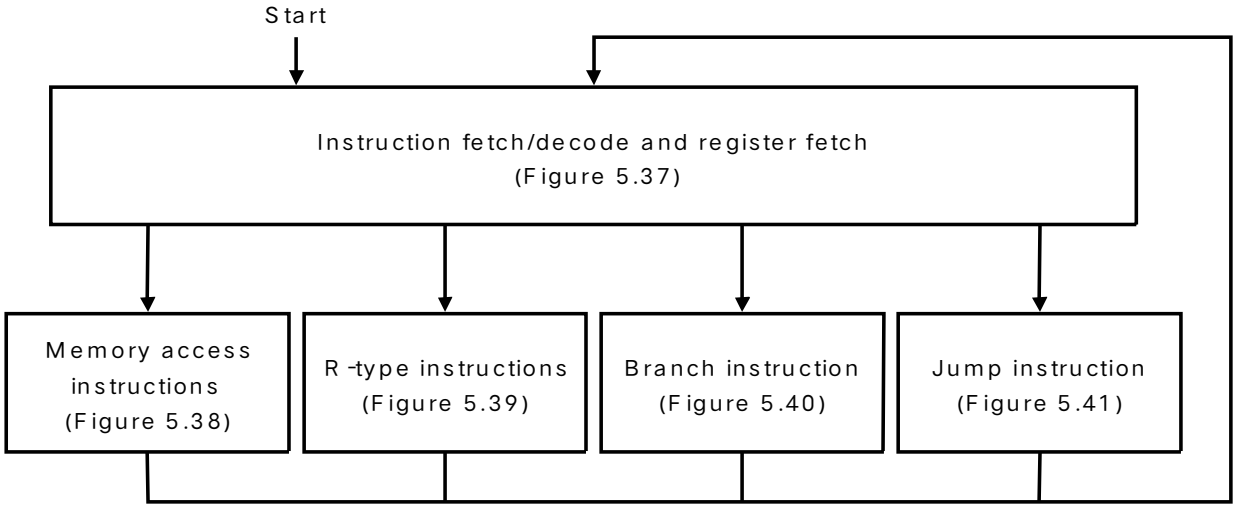
- The Moore machine: given *input* a binary string terminated by “#”, will *output* “even” if the string has an even number of 0’s and “odd” if the string has an odd number of 0’s

Example: Moore Machine

- The Moore machine below, given *input* a binary string terminated by "#", will *output* "even" if the string has an even number of 0's and "odd" if the string has an odd number of 0's

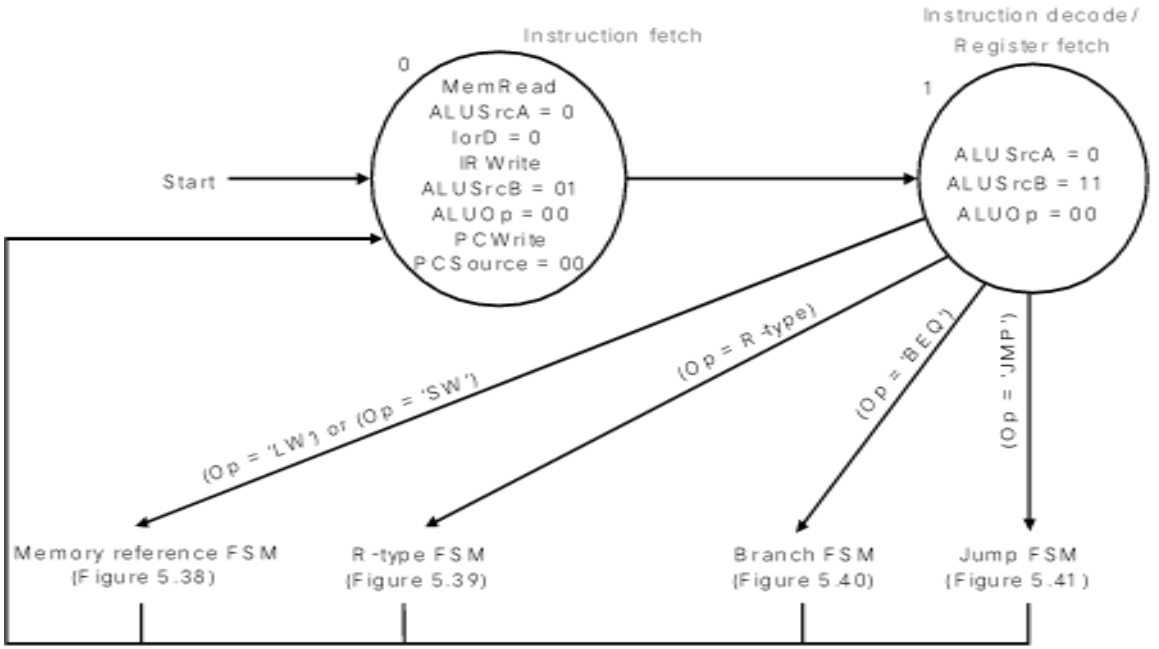


FSM Control: High-level View



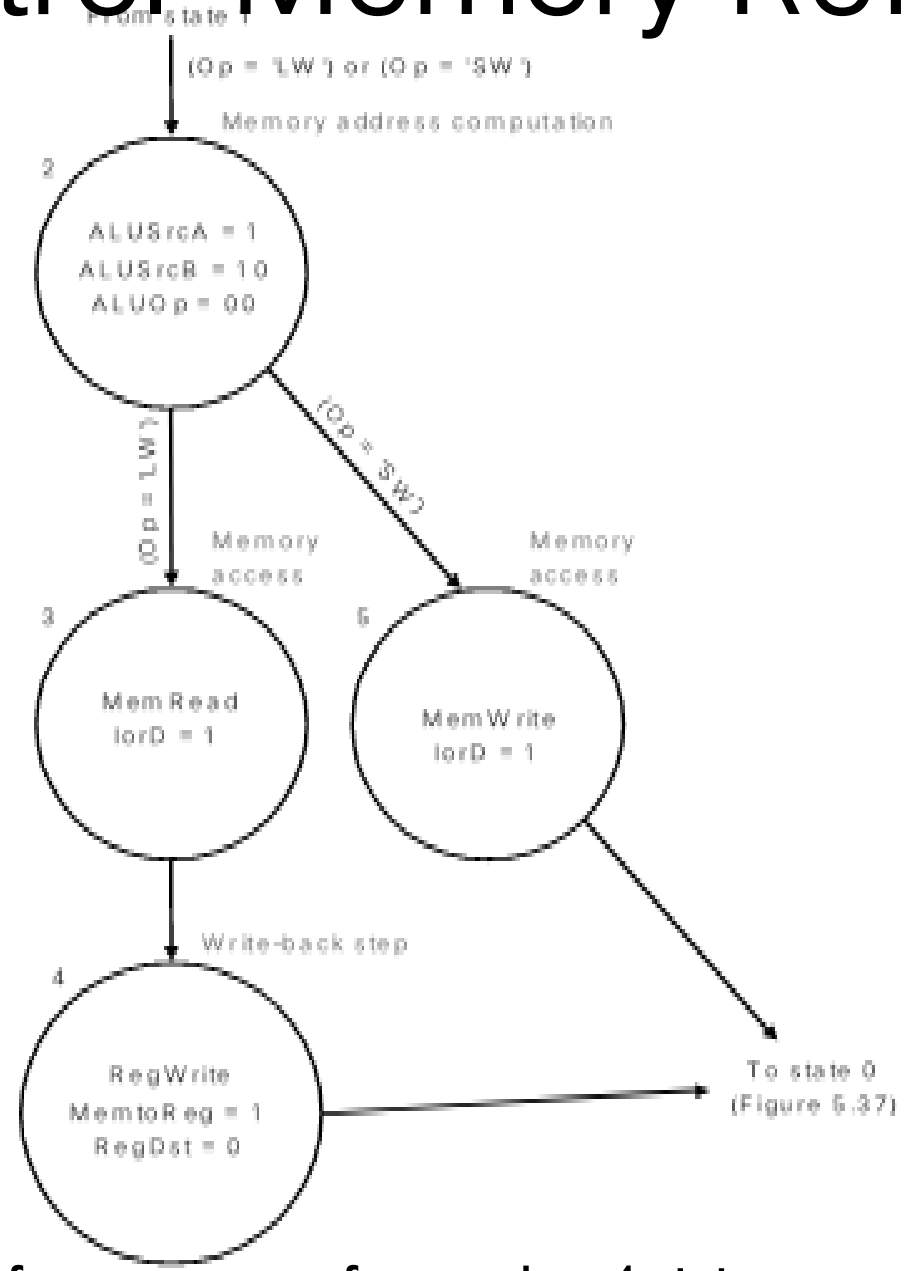
High-level view of FSM control

Asserted signals shown inside state circles



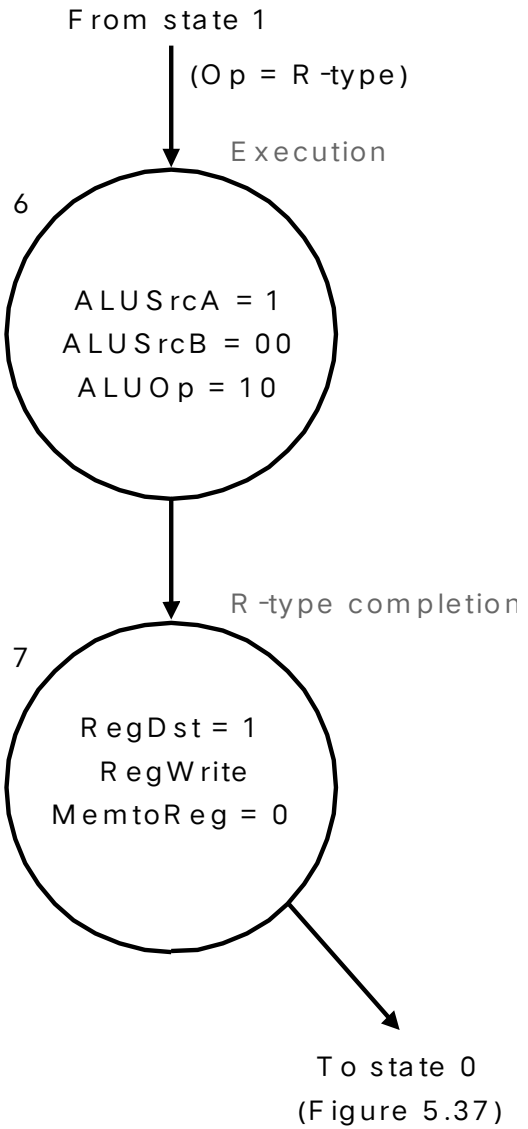
Instruction fetch and decode steps of every instruction is identical

FSM Control: Memory Reference



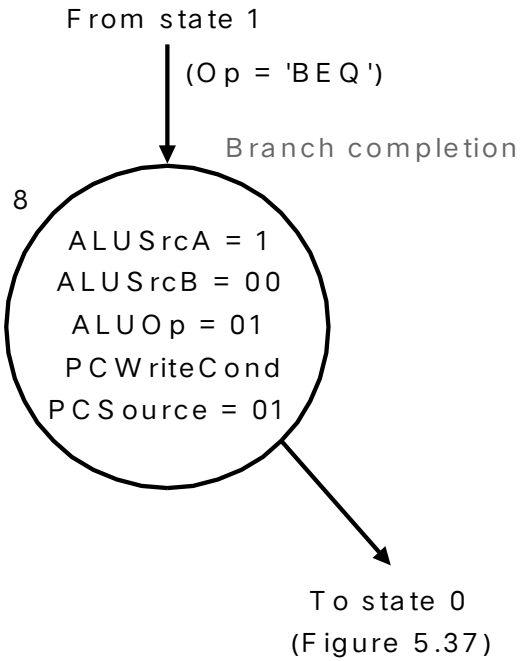
FSM control for memory-reference has 4 states

FSM Control: R-type Instruction



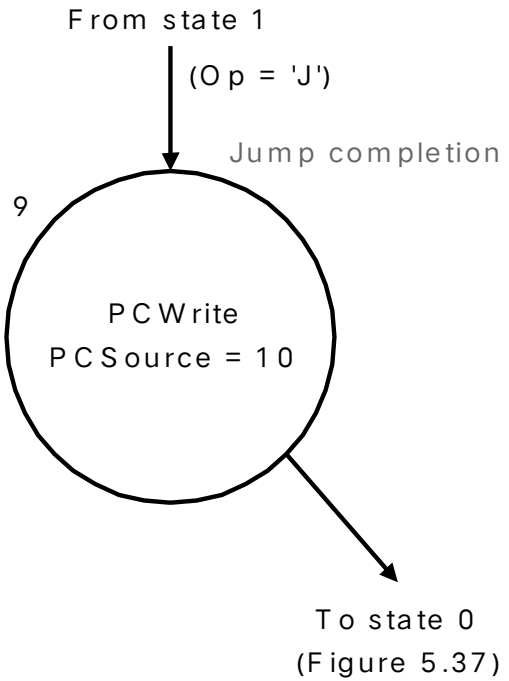
FSM control to implement R-type instructions has 2 states

FSM Control: Branch Instruction



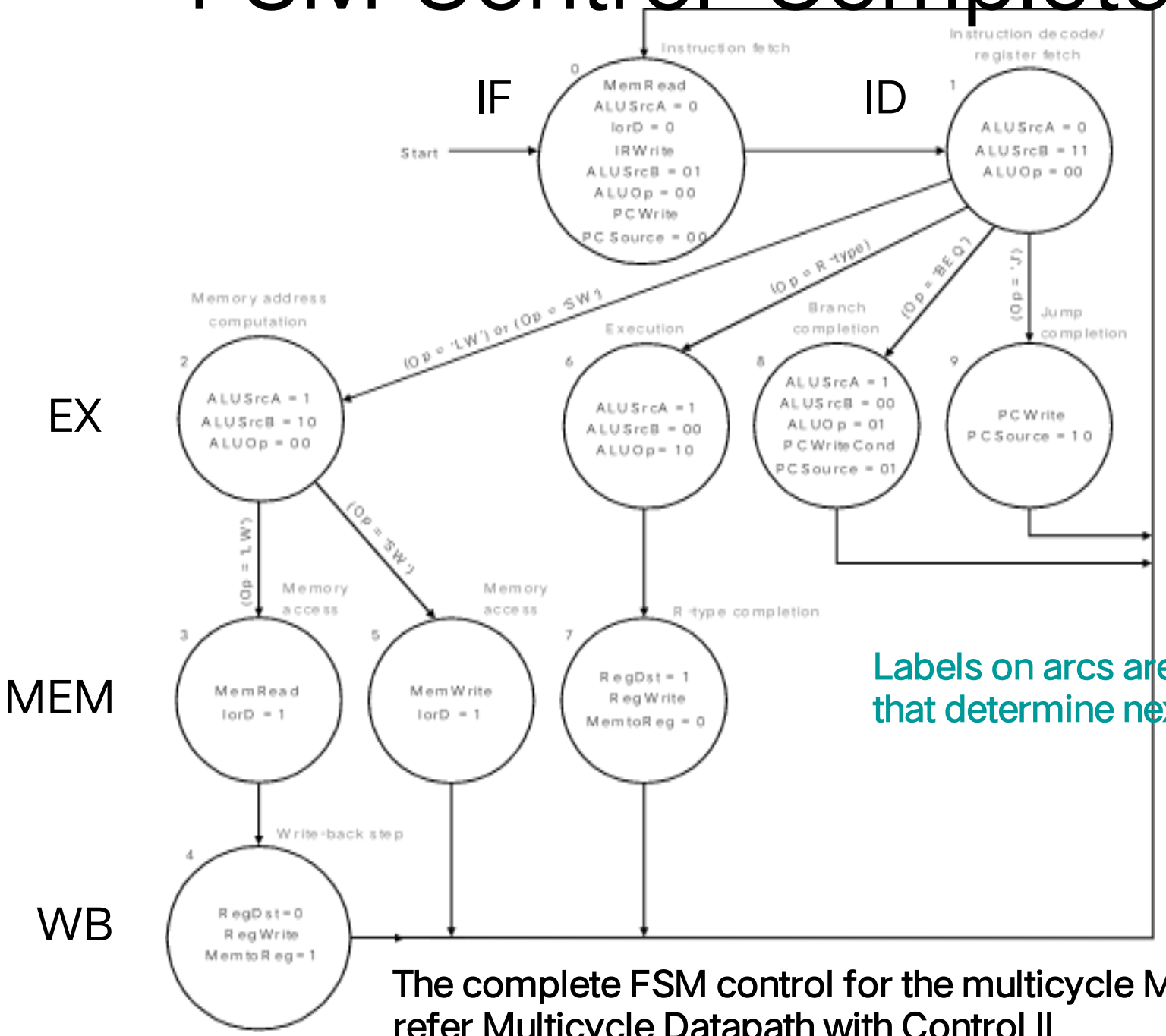
FSM control to implement branches has 1 state

FSM Control: Jump Instruction



FSM control to implement jumps has 1 state

FSM Control: Complete View



Labels on arcs are conditions that determine next state

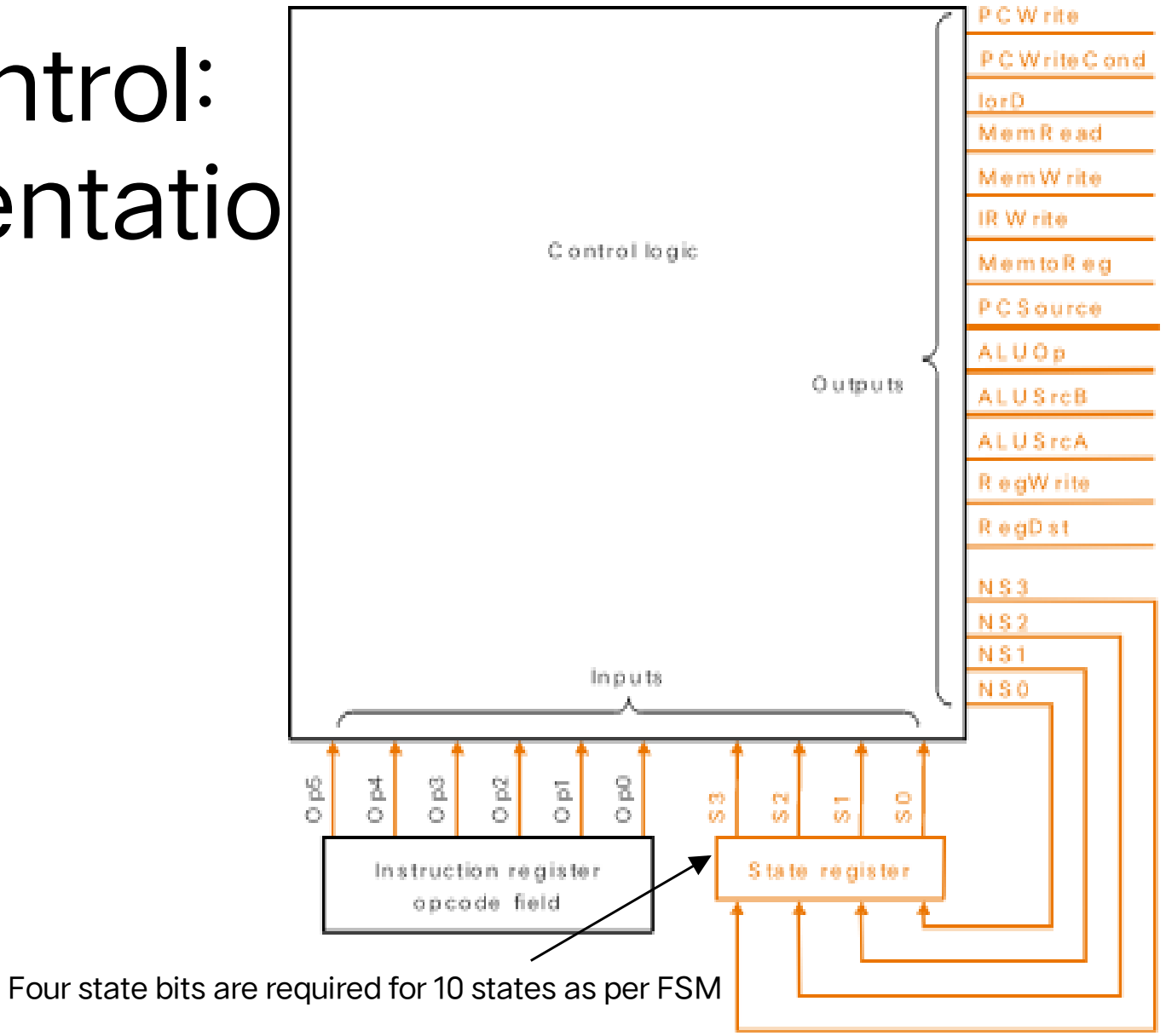
The complete FSM control for the multicycle MIPS datapath: refer Multicycle Datapath with Control II

FSM Control: Implementation

Requires

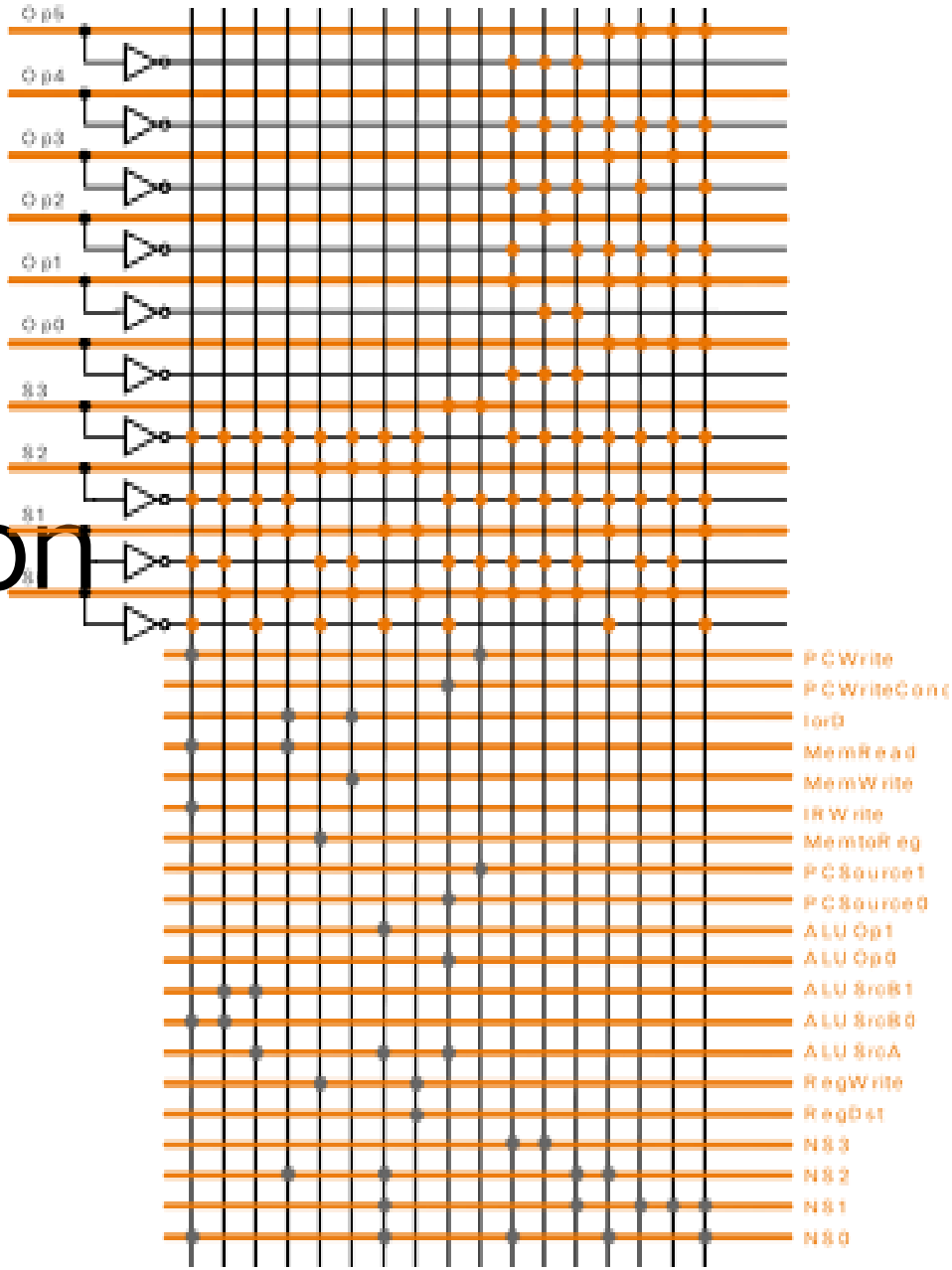
- A temporary register that holds the current state
- A block of combinational logic that determines both the data path signals to be asserted as well as the next state

FSM Control: Implementatio



High-level view of FSM implementation: inputs to the combinational logic block are the current state number and instruction opcode bits; outputs are the next state number and control signals to be asserted for the current state

FSMControl: PLA Implementation

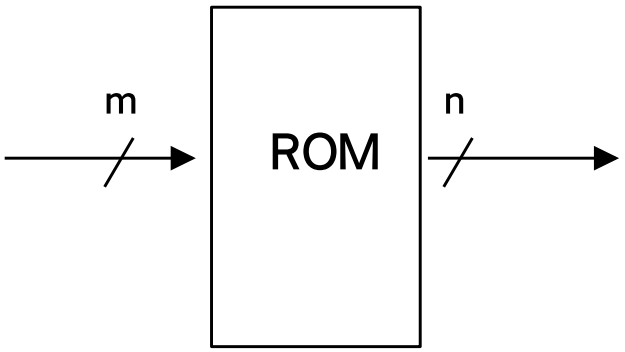


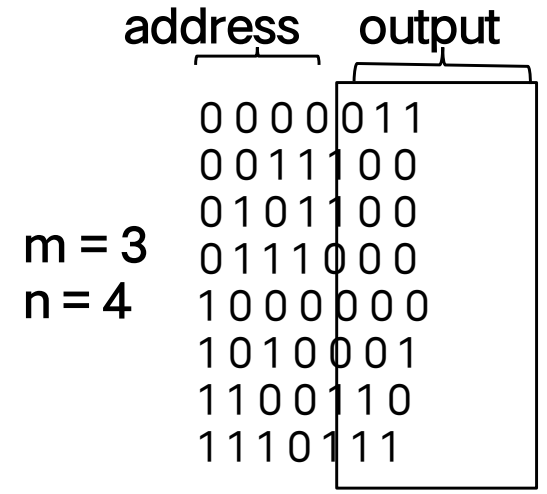
Upper half is the AND plane that computes all the products. The products are carried to the lower OR plane by the vertical lines. The sum terms for each output is given by the corresponding horizontal line

E.g., $IorD = S0.S1.S2.S3 + S0.S1.\overline{S2}.S3$

FSM Control: ROM Implementation

- ROM (Read Only Memory)
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address 2^m entries in the ROM
  - outputs are the bits of the entry the address points to



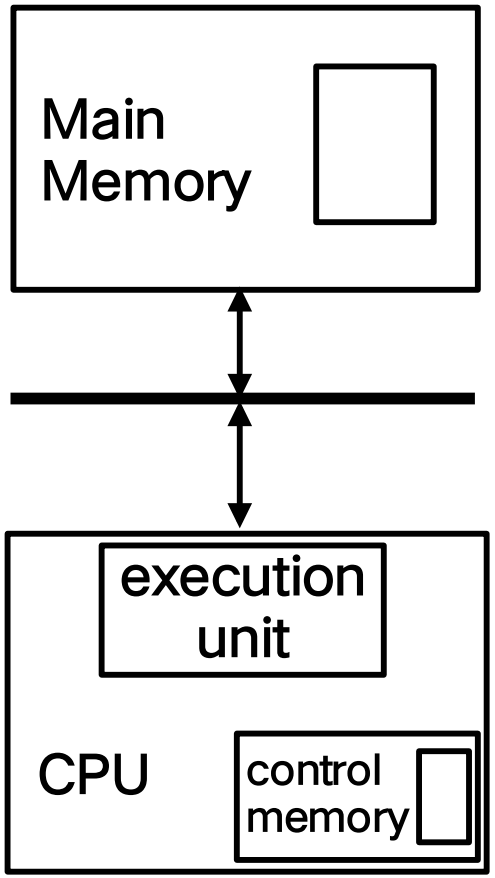


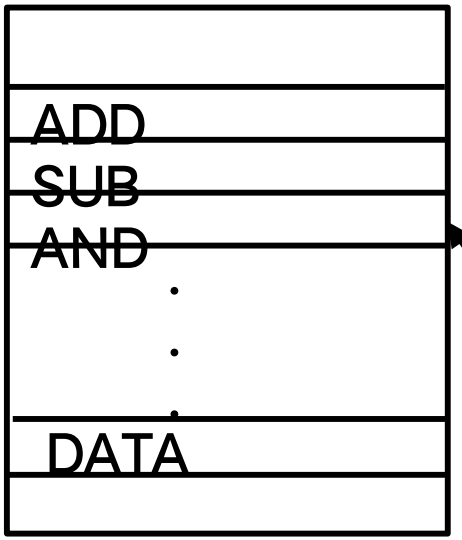
The size of an m-input n-output ROM is 2^m x n bits – such a ROM can be thought of as an array of size 2^m with each entry in the array being n bits

Microprogramming

- Microprogramming is a method of *specifying* FSM control that resembles a programming language – textual rather graphic
  - Appropriate when the FSM becomes very large, e.g., if the instruction set is large and/or the number of cycles per instruction is large
  - In such situations graphical representation becomes difficult as there may be thousands of states and even more arcs joining them
  - A microprogram is *specification* : *implementation* is by ROM or PLA

“Microinstruction” Interpretation

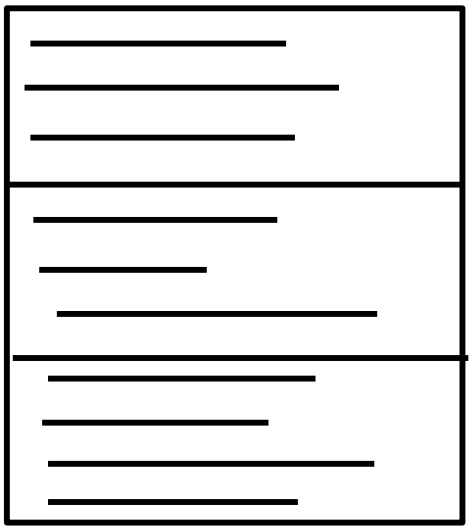




User program
plus Data

this can change!

one of these is
mapped into one
of these



AND microsequence

e.g., Fetch
Calc Operand Addr
Fetch Operand(s)
Calculate
Save Answer(s)

Microprogramming

- A *microprogram is a sequence of microinstructions*
  - each microinstruction has eight fields (label + 7 functional)
    - Label: used to control microcode sequencing
    - ALU control: specify operation to be done by ALU
    - SRC1: specify source for first ALU operand
    - SRC2: specify source for second ALU operand
    - Register control: specify read/write for register file
    - Memory: specify read/write for memory
    - PCWrite control: specify the writing of the PC
    - Sequencing: specify choice of next microinstruction

Microprogramming

- The *Sequencing* field value determines the execution order of the microprogram
  - value *Seq* : control passes to the sequentially next microinstruction
  - value *Fetch* : branch to the first microinstruction to begin the next MIPS instruction, i.e., the first microinstruction in the microprogram
  - value *Dispatch i* : branch to a microinstruction based on control input and a dispatch table entry (called *dispatching*):
    - Dispatching is implemented by means of creating a table, called *dispatch table*, whose entries are microinstruction labels and which is indexed by the control input. There may be multiple dispatch tables – the value *Dispatch i* in the sequencing field indicates that the *i* th dispatch table is to be used

Microprogram

Label	*ALU*	*SRC1*	*SRC2*	*Dest.*	*Memory*	*Mem. Reg.*	*PC Write*	*Sequencing*
Fetch:	Add	PC	4		Read	PC	IR	ALU Seq
	Add	PC	Extshft					Dispatch
Rtype:	Func	rs	rt					Seq
				rd	ALU			Fetch
Lw:	Add	rs	Extend					Seq
					Read	ALU		Seq
				rt	MEM			Fetch
Sw:	Add	rs	Extend					Seq
					Write	rt	ALU	Fetch
Beq:	Subt.	rs	rt				ALUoutCond.	Fetch

Control Microprogram

- The microprogram corresponding to the FSM control shown graphically earlier:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
R format1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

Microprogram containing 10 microinstructions

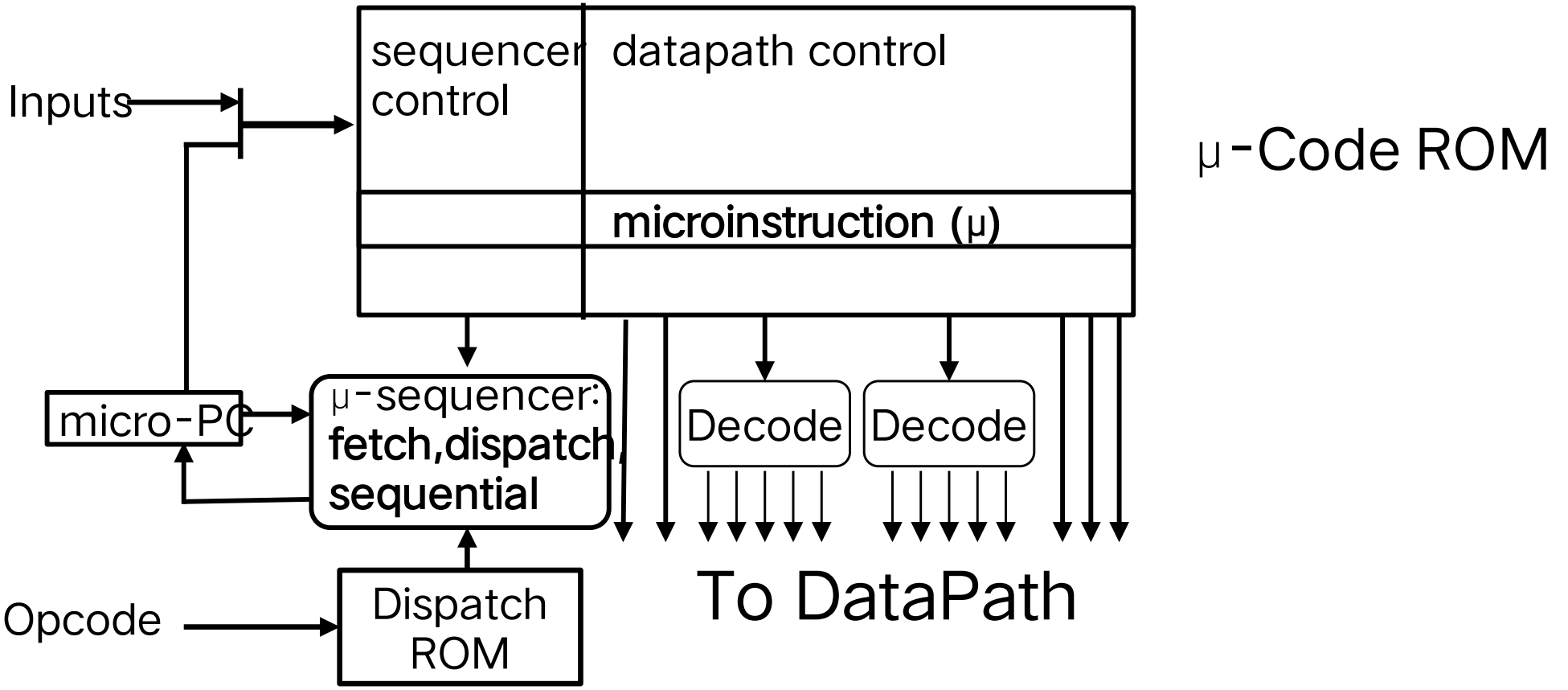
Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	Rformat1
000010	jmp	JUMP1
000100	beq	BEQ1
100011	lw	Mem1
101011	sw	Mem1

Dispatch Table 1

Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	LW2
101011	sw	SW2

Dispatch Table 2

Microprogramming



- Microprogramming is a fundamental concept
  - implement an instruction set by building a very simple processor and interpreting the instructions

Specific Sequencer

Sequencer-based control unit

– Called “microPC” or “µPC” vs. state register

Code Name Effect

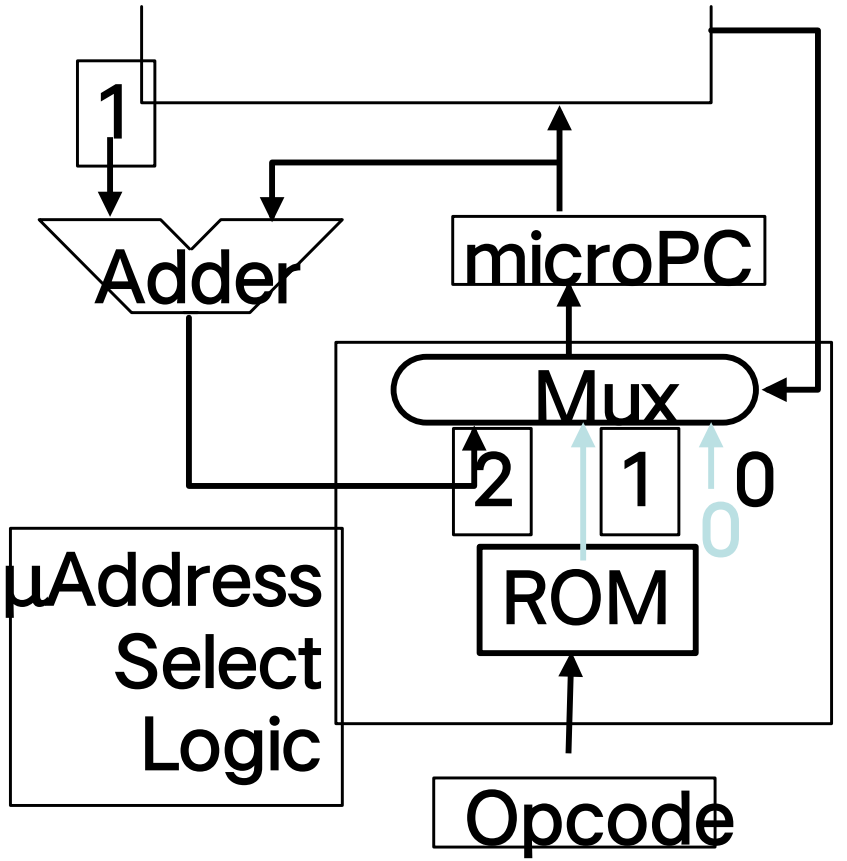
00 fetch Next µaddress = 0

01 dispatch Next µaddress = dispatch ROM

10 seq Next µaddress = µaddress + 1

ROM:

R-type	000000	0100
BEQ	000100	0011
ori	001101	0110
LW	100011	1000
SW	101011	1011



Microcode: Trade-offs

- Specification advantages
  - easy to design and write
  - typically manufacturer designs architecture and microcode in parallel
- Implementation advantages
  - easy to change since values are in memory (e.g., off-chip ROM)
  - can emulate other architectures
  - can make use of internal registers
- Implementation disadvantages
  - control is implemented nowadays on same chip as processor so the advantage of an off-chip ROM does not exist
  - ROM is no longer faster than on-board cache
  - there is little need to change the microcode as general-purpose computers are used far more nowadays than computers designed for specific applications

From Patterson Book

2nd Edition Copy – Refer from beginning to point 5.5(including)

Not to refer: 5.5 onwards…

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*

Tutorial Question

- 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?*

Tutorial

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$_{\text{class i}}$ × CPI$_{\text{class i}}$) / instruction count

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

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

    $= 4.04$

ACM Webinar

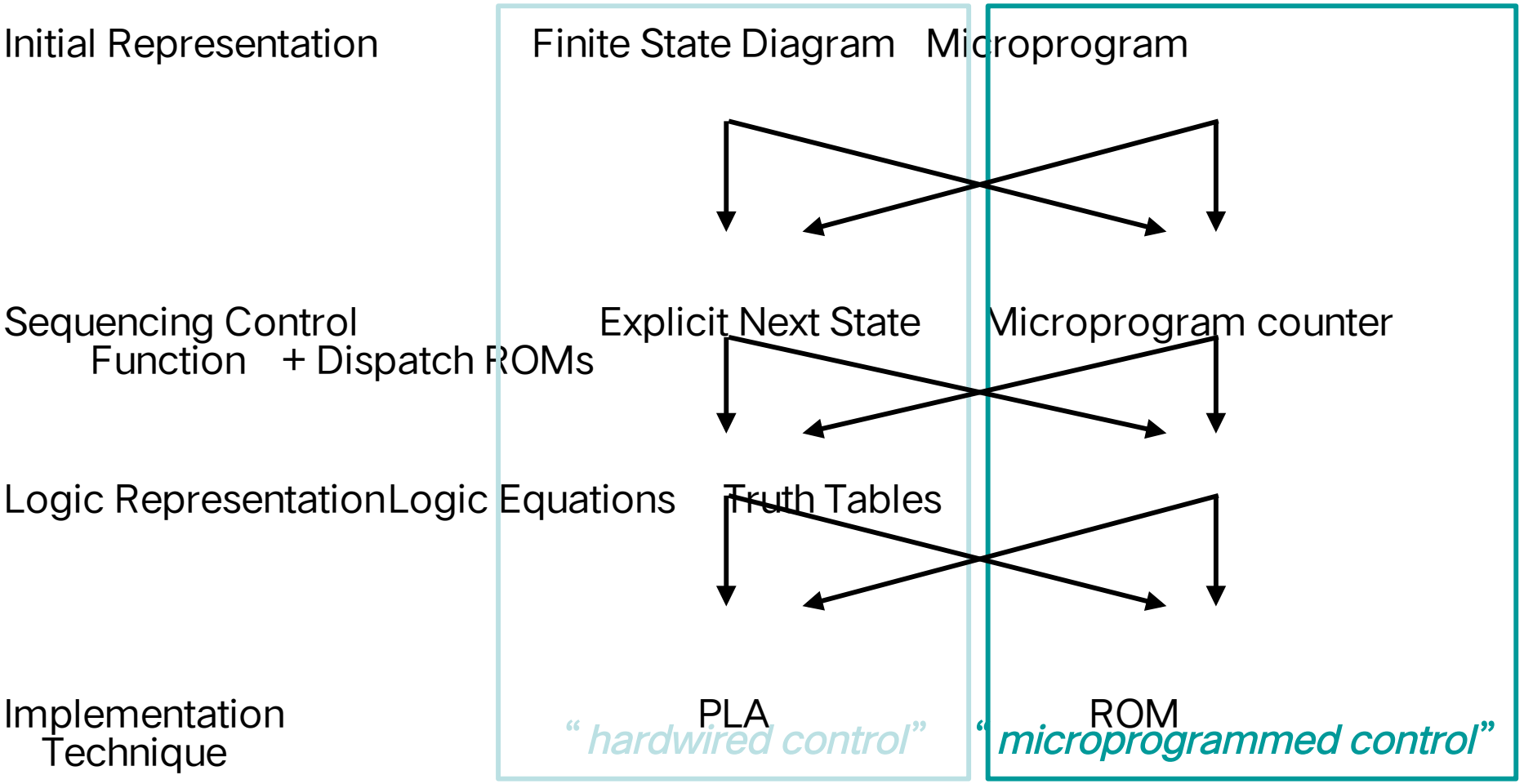
- Block Chain Technology

FSM Control: ROM vs. PLA

- First improve the ROM: break the table into two parts
  - 4 state bits give the 16 output signals – 2^4 x 16 bits of ROM
  - all 10 input bits give the 4 next state bits – 2^{10} x 4 bits of ROM
  - Total – 4.3K bits of ROM
- PLA is much smaller
  - can share product terms
  - only need entries that produce an active output
  - can take into account don't cares
- PLA size = (#inputs × #product-terms) + (#outputs × #product-terms)
  - FSM control PLA = (10x17)+(20x17) = 460 PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

Overview of Microprogrammed Control Unit

- Control may be designed using one of several initial representations.
- The choice of sequence control, and how logic is represented, can then be determine independently; the control can then be implemented with one of several methods using a structured logic technique.

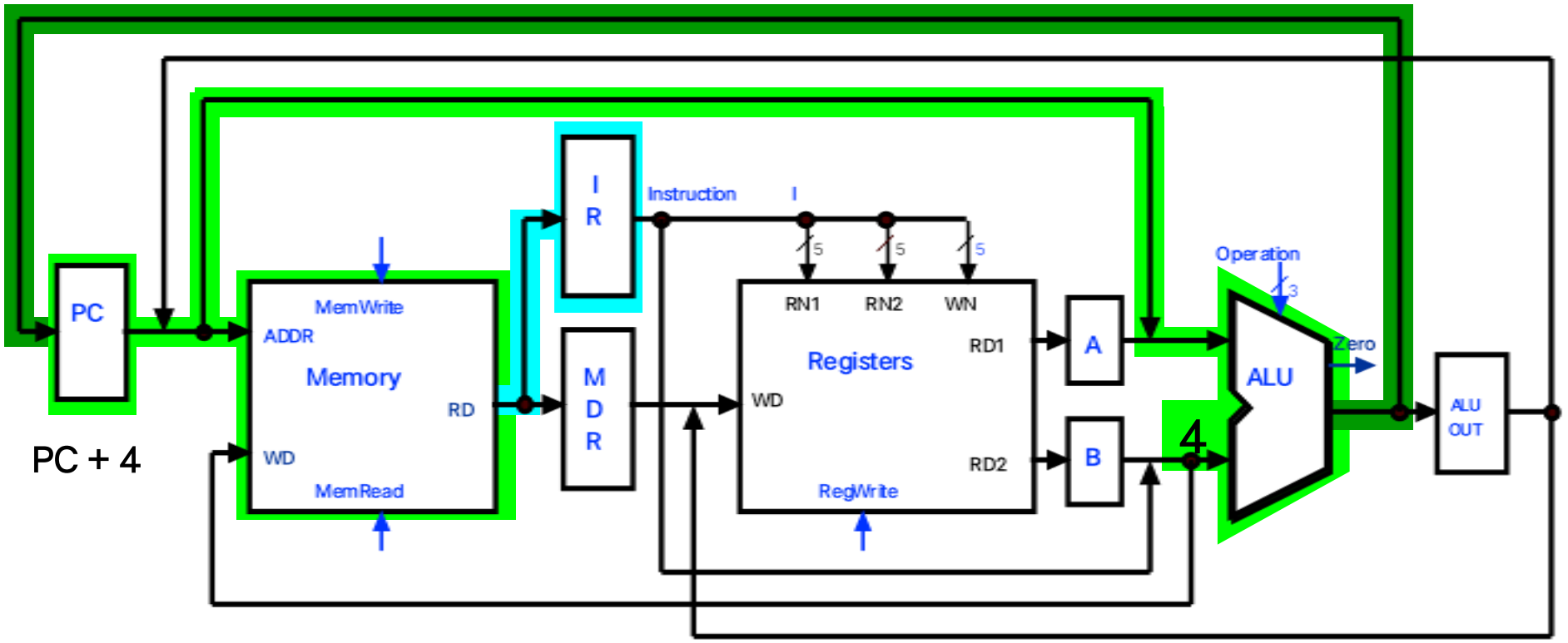


CORRECTIONs are required

NOT TO FOLLOW SLIDES FROM 57 onwards

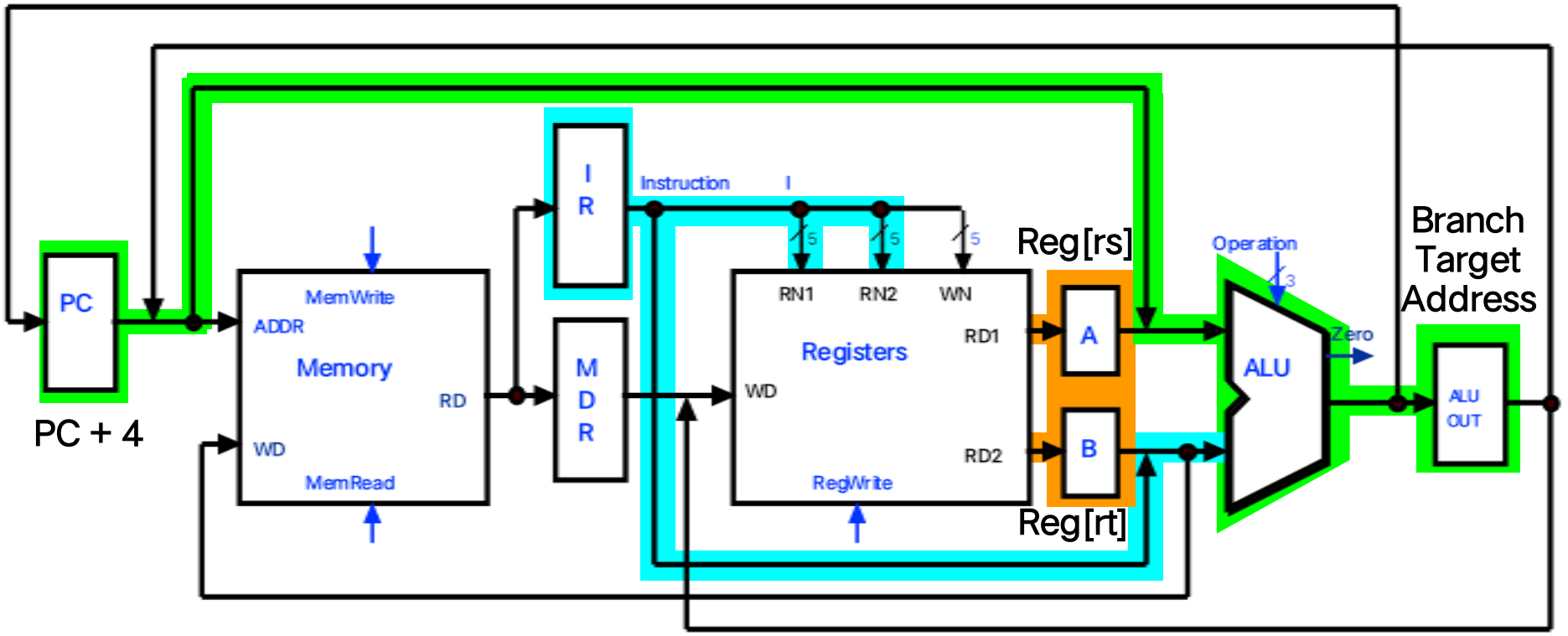
Multicycle Execution Step (1): Instruction Fetch

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



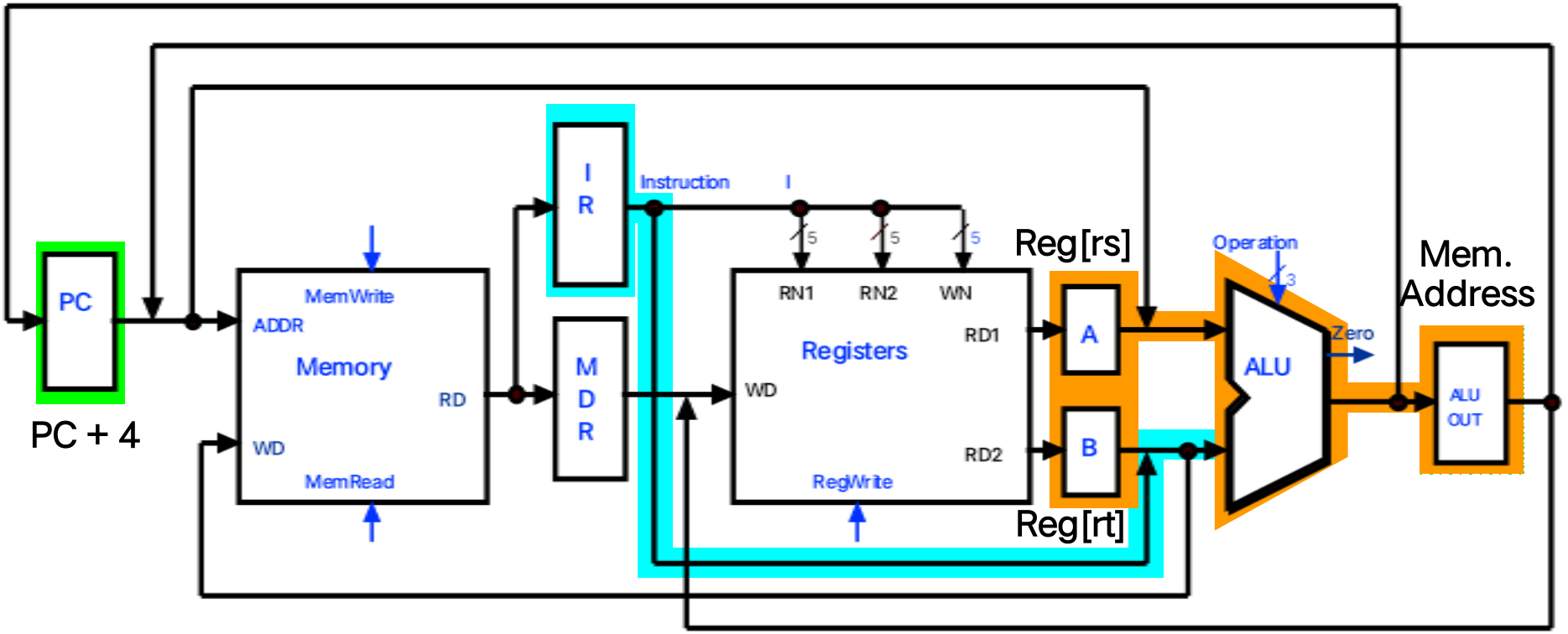
Multicycle Execution Step (2): Instruction Decode & Register Fetch

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



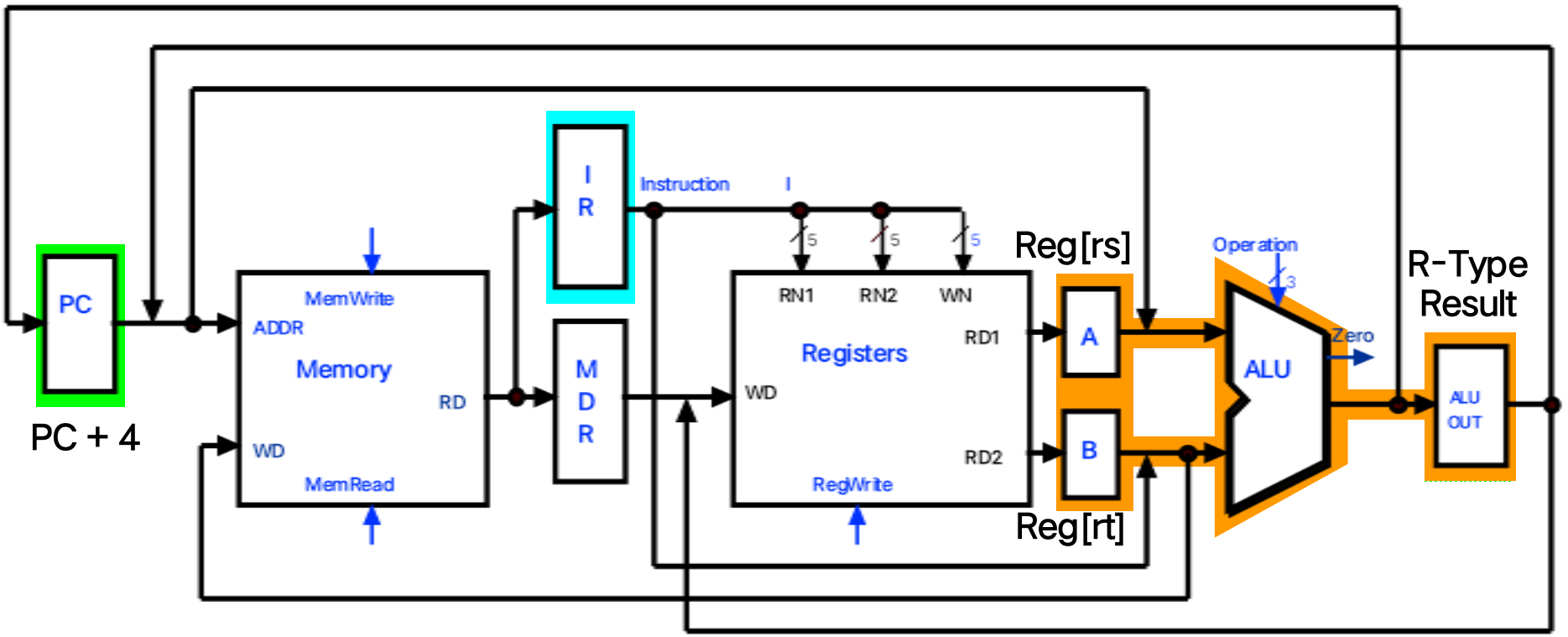
Multicycle Execution Step (3): Memory Reference Instructions

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



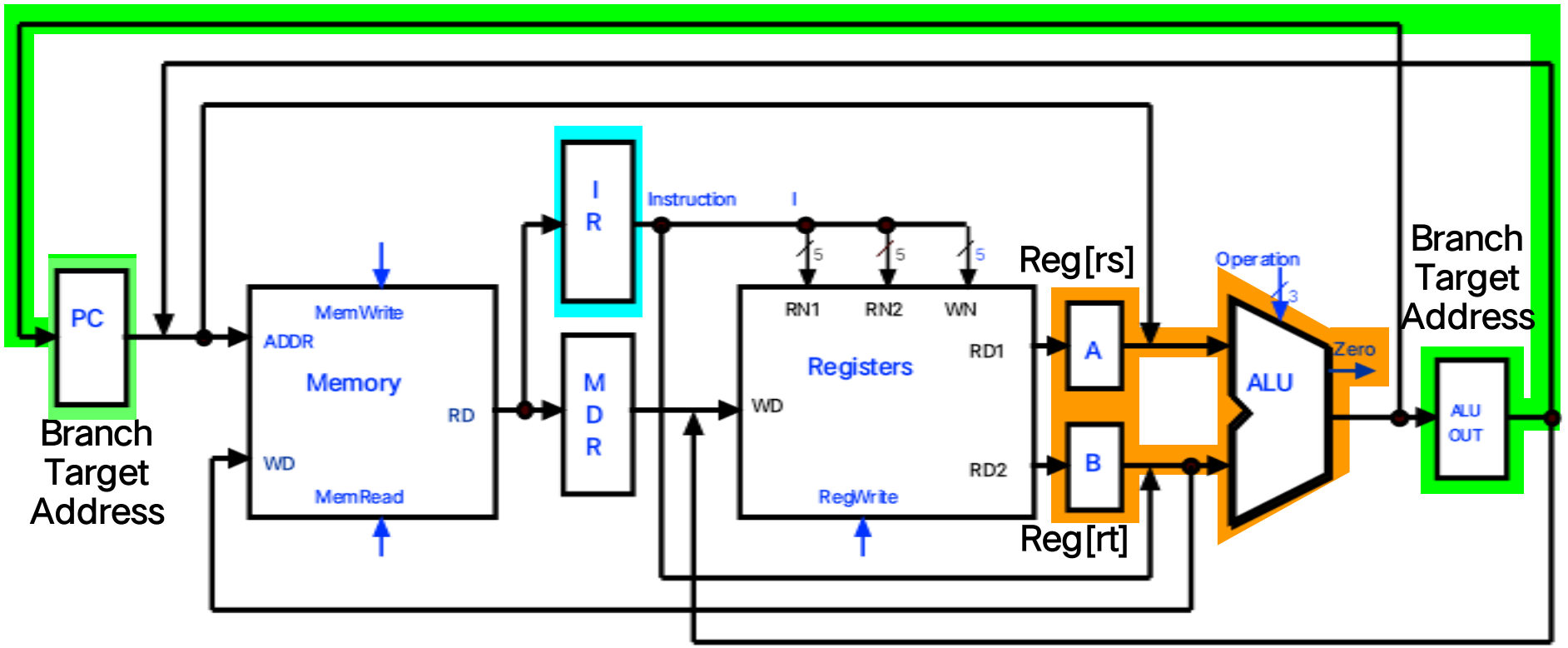
Multicycle Execution Step (3): ALU Instruction (R-Type)

ALUOut = A op B



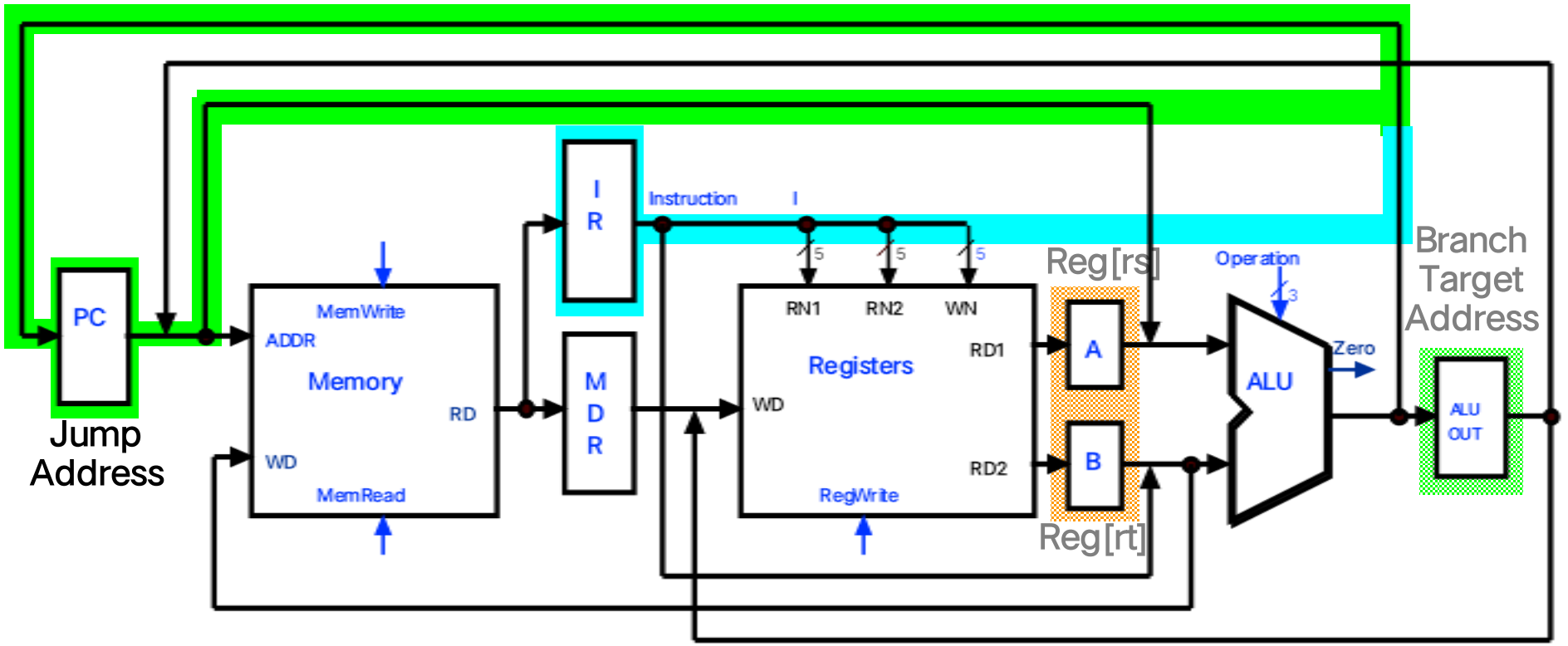
Multicycle Execution Step (3): Branch Instructions

if (A == B) PC = ALUOut;



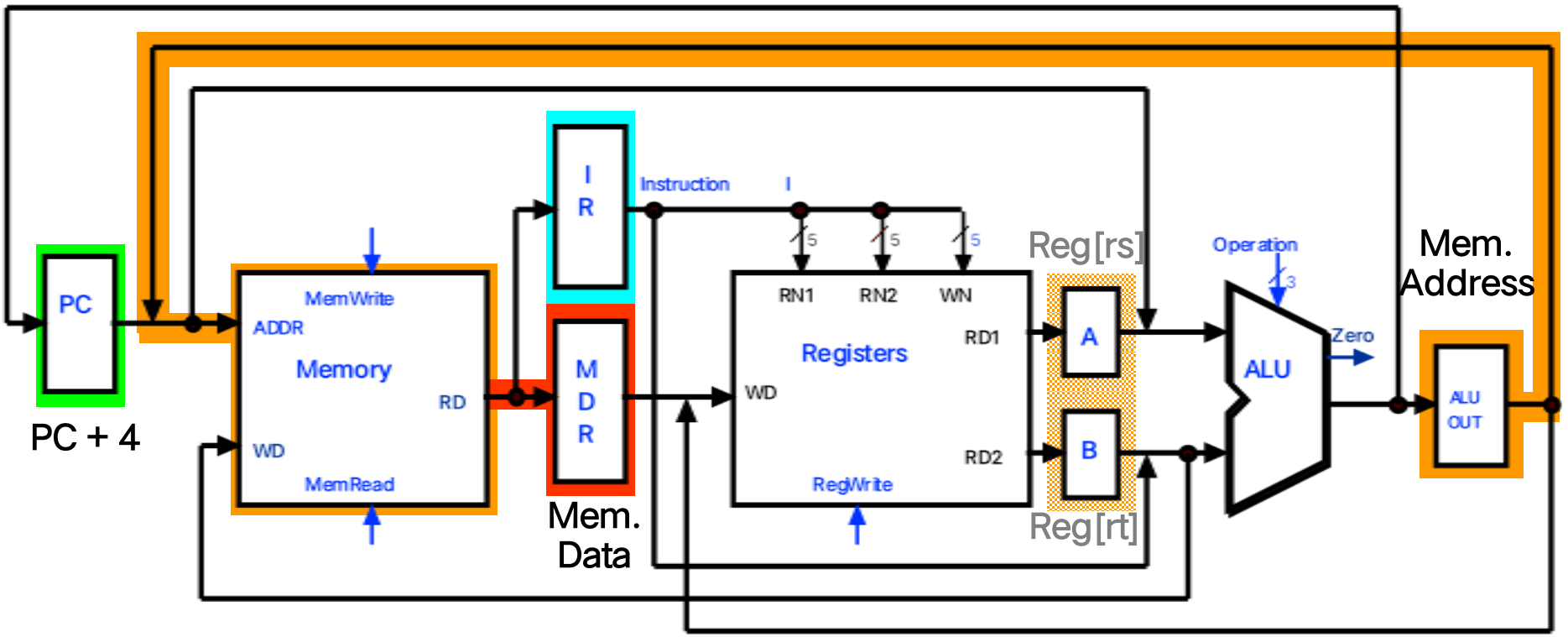
Multicycle Execution Step (3): Jump Instruction

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



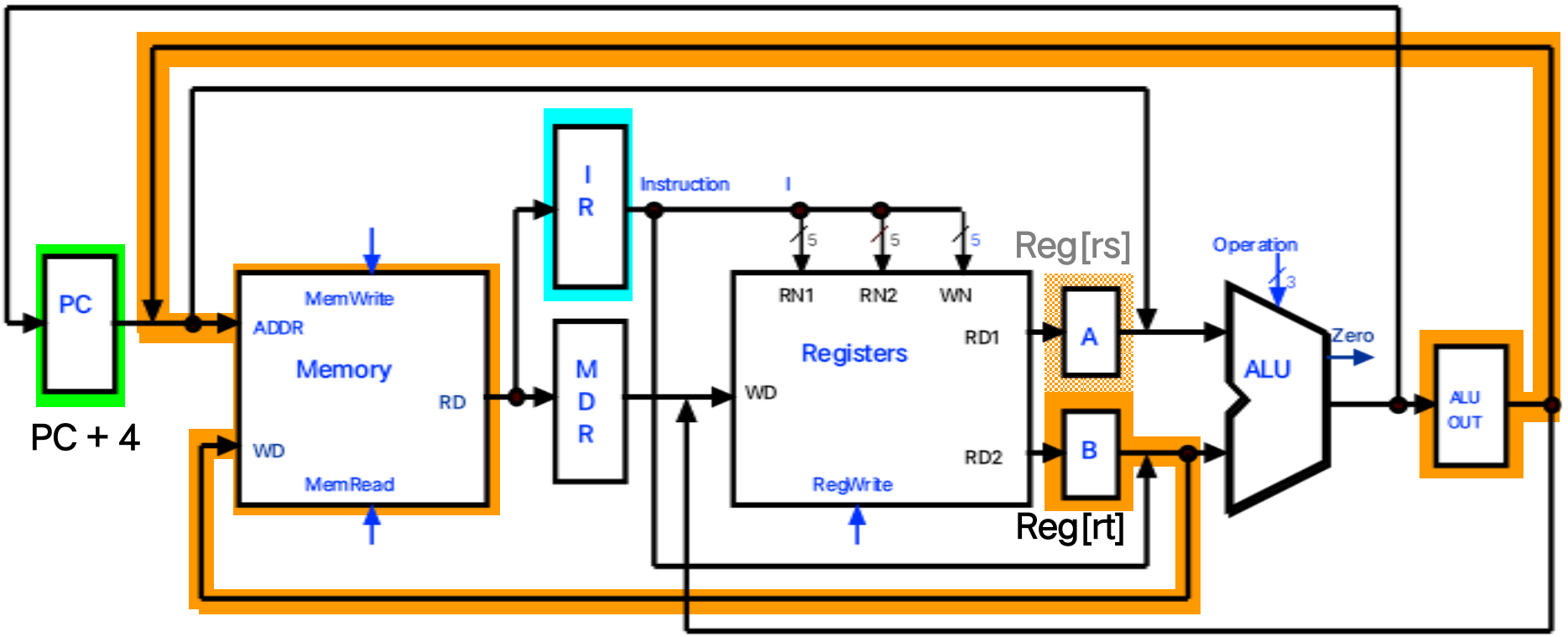
Multicycle Execution Step (4): Memory Access - Read (lw)

MDR = Memory[ALUOut];



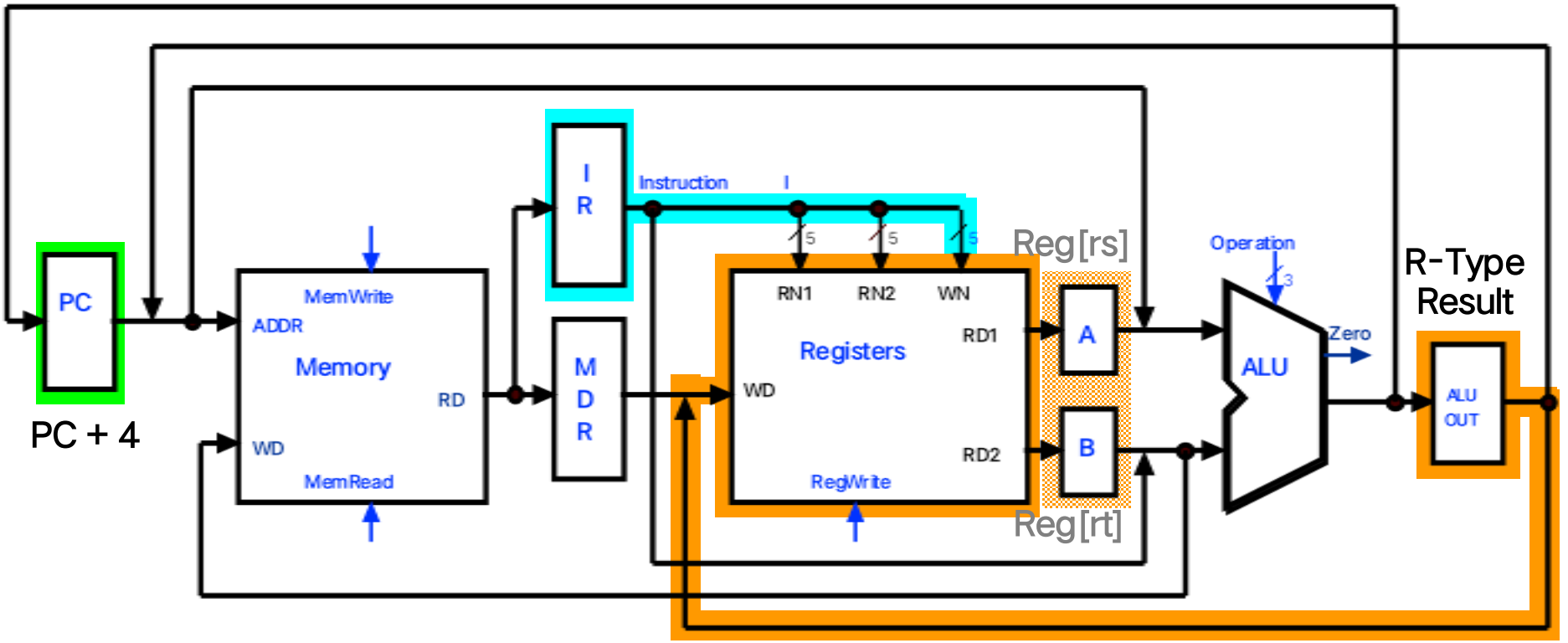
Multicycle Execution Step (4): Memory Access - Write (sw)

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;