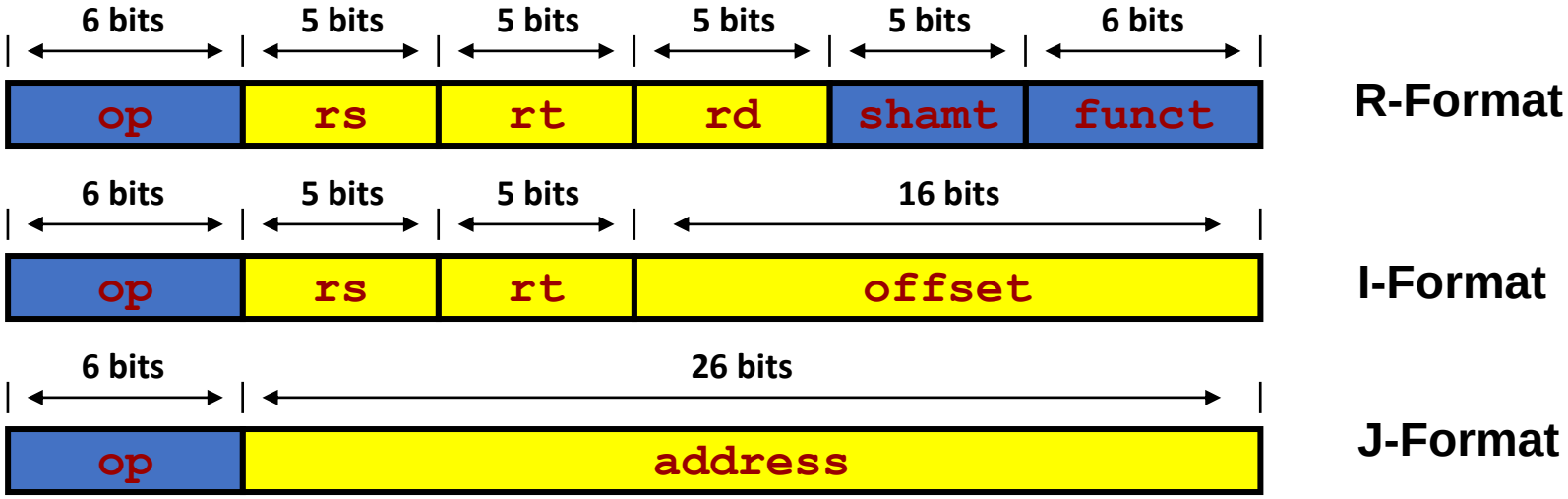
MIPS Processor Design

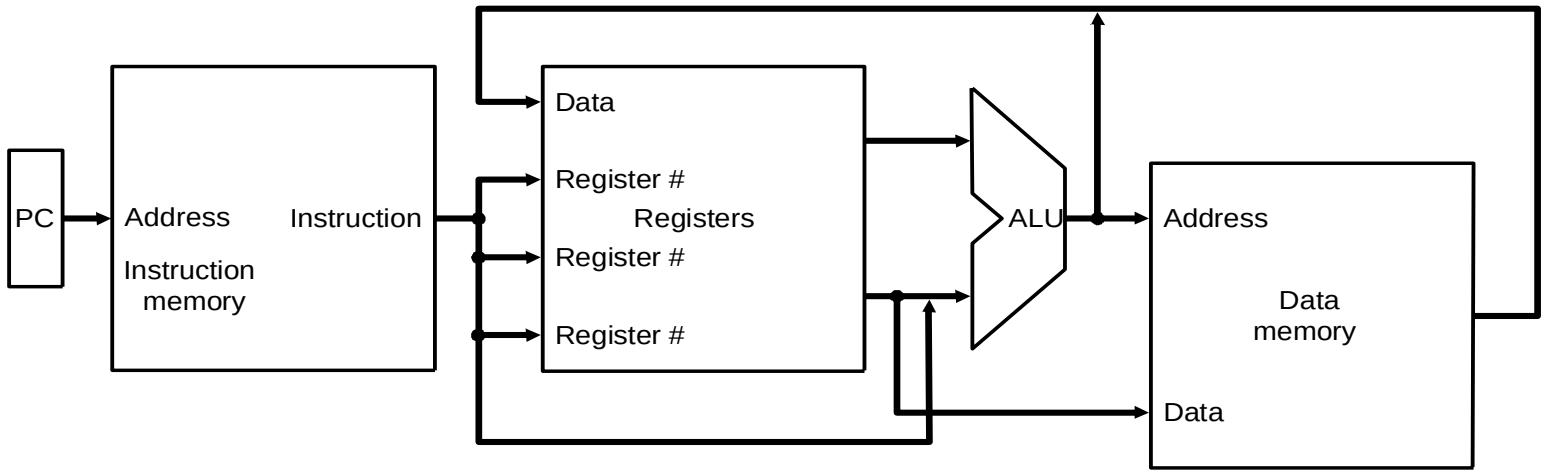
Introduction – Processor Design

- 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`**



Introduction – Processor Design

- 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

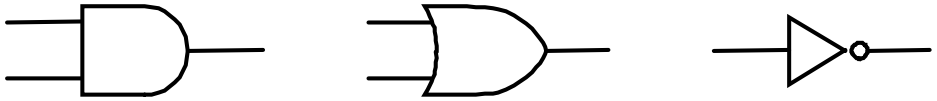
- **Single Cycle**
  - perform each instruction in 1 clock cycle
  - clock cycle must be long enough for slowest instruction; therefore,
  - disadvantage: only as fast as slowest instruction
- **Multi-Cycle**
  - break fetch/execute cycle into multiple steps
  - perform 1 step in each clock cycle
  - 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

Functional Elements

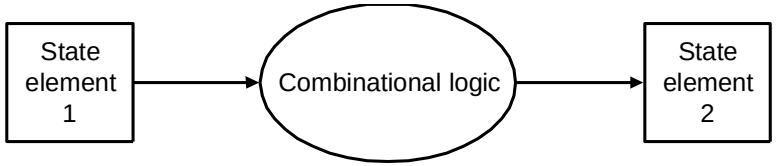
- Two types of functional elements in the hardware:
  - elements that operate on data (called **combinational elements**)
  - elements that contain data (called **state or sequential elements**)

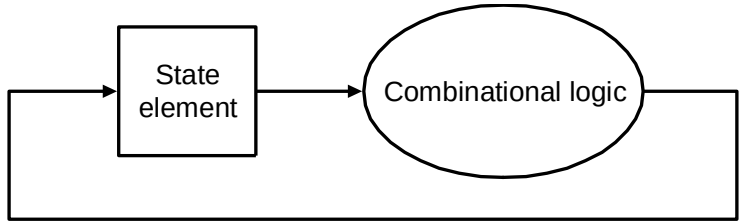
Combinational Elements

- Works as an *input* $\Rightarrow$ *output function*, e.g., ALU
- Combinational logic *reads input data from one register and writes output data to another, or same, register*
  - read/write happens in a single cycle – combinational element cannot store data from one cycle to a future one



Combinational logic hardware units



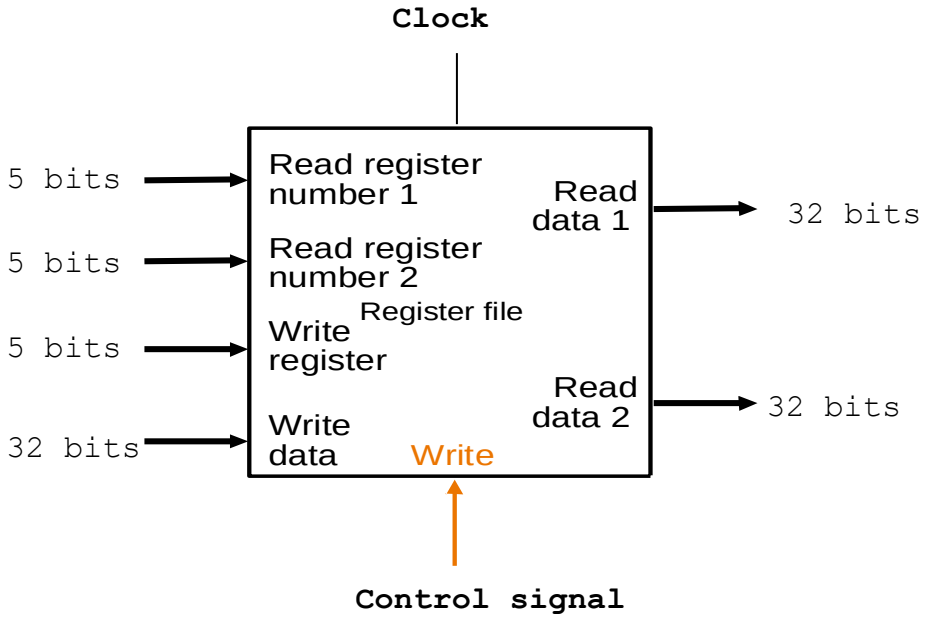


Sequential or State Elements

- State elements contain *data* in internal storage, e.g., *registers* and *memory*
- All state elements together *define* the *state of the machine*
  - *What does this mean? Think of shutting down and starting up again…*
- *Flipflops* and *latches* are 1-bit state elements, equivalently, they are *1-bit memories*
- The *output*(s) of a flipflop or latch *always* depends on the bit value stored, i.e., its state, and can be called *1/0* or *high/*low or *true/false*
- The *input* to a flipflop or latch can change its state depending on whether it is clocked or not…

State elements on the Datapath: Register File

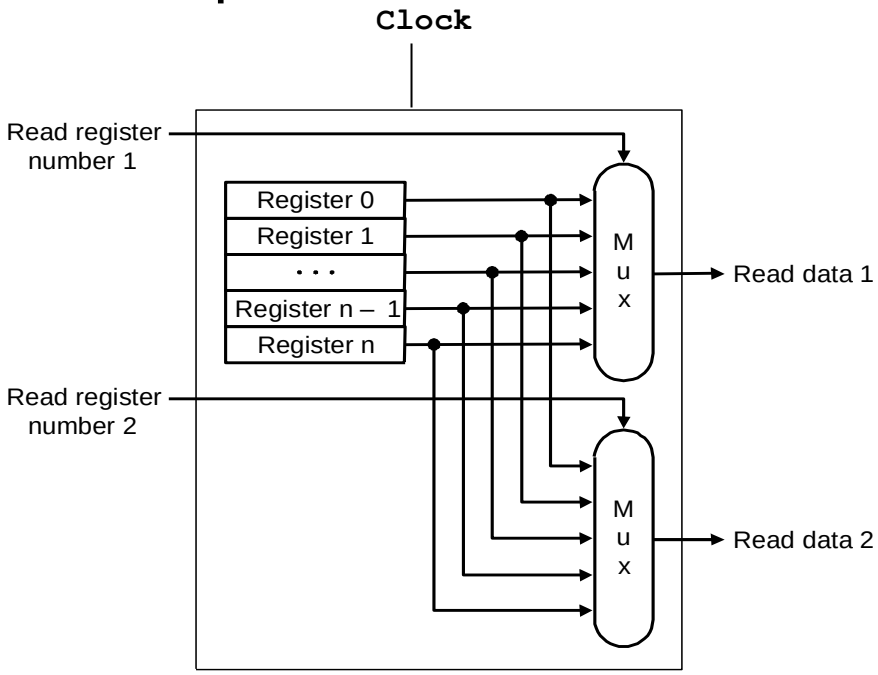
- Registers are implemented with arrays of D-flipflops



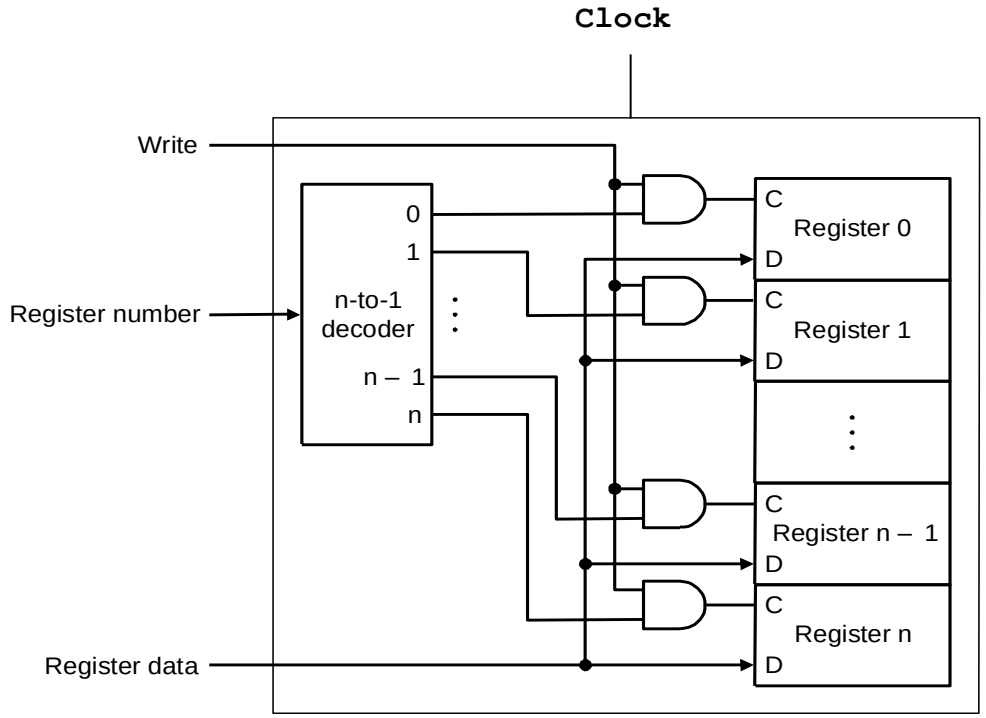
Register file with two read ports and one write port

State elements on the Datapath: Register File

- Port implementation:



Read ports are implemented with a pair of multiplexors – 5 bit multiplexors for 32 registers



Write port is implemented using a decoder – 5-to-32 decoder for 32 registers. Clock is relevant to write as register state may change only at clock edge

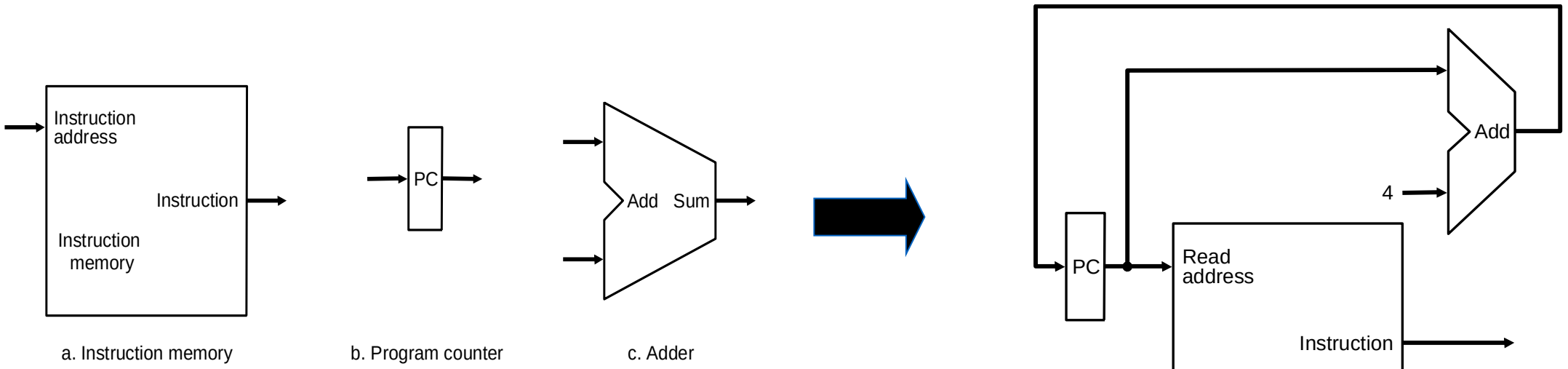
Single Cycle Implementation

- Our first implementation of MIPS will use a *single* long clock cycle for every instruction
- Every instruction begins on one up (or down) clock edge and ends on the next up (or, down) clock edge
- This approach is *not practical* as it is much slower than a *multicycle* implementation where different instruction classes can take different numbers of cycles
  - in a single-cycle implementation every instruction must take the same amount of time as the slowest instruction
  - in a multicycle implementation this problem is avoided by allowing quicker instructions to use fewer cycles
- Even though the single-cycle approach is not practical it is simple and useful to understand first.

General Sequence of Instruction Execution

- For every instruction:
  - Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.
  - Read one or two registers, using fields of the instruction to select the registers to read.
- For the load/store word instruction, we need to read only one register.
- Three types of instructions: memory-reference, arithmetic-logical, and branches.
- All instruction classes, except jump, use the ALU after reading the registers.

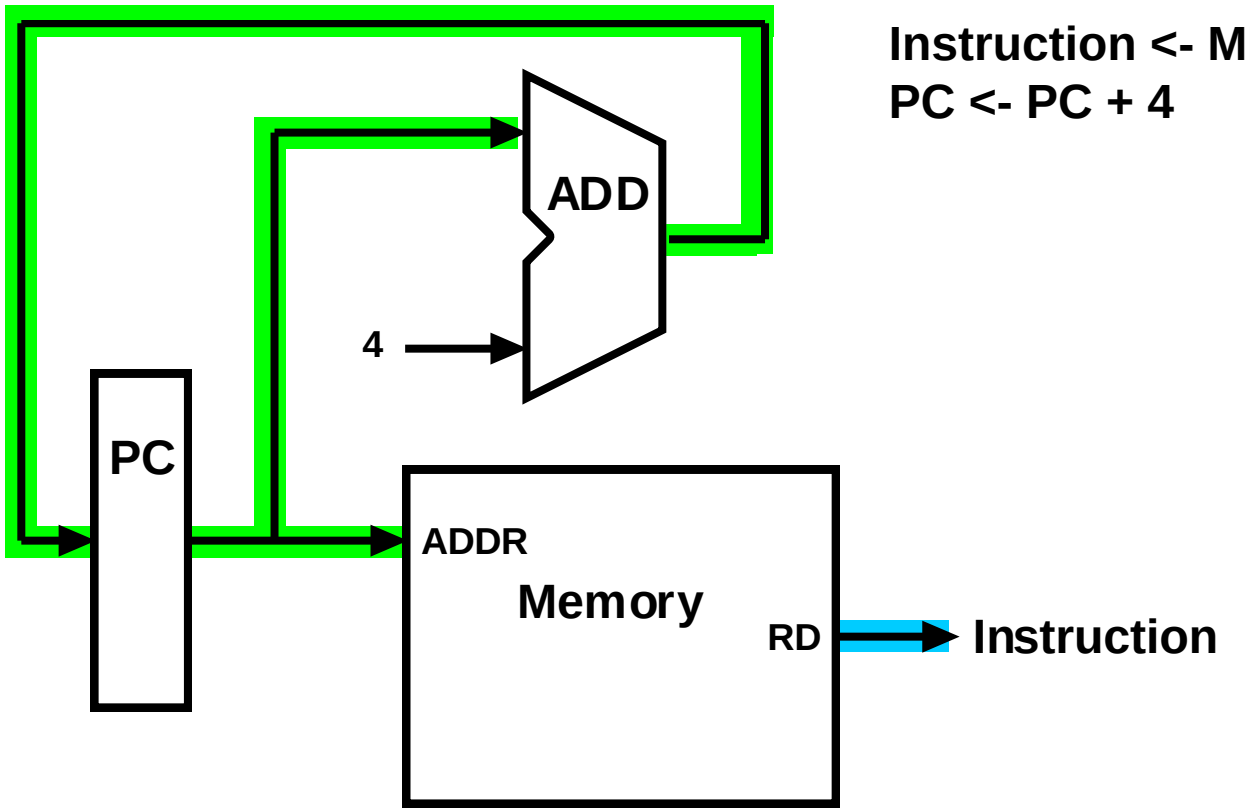
Datapath: Instruction Store/Fetch & PC Increment



Three elements used to store and fetch instructions and increment the PC

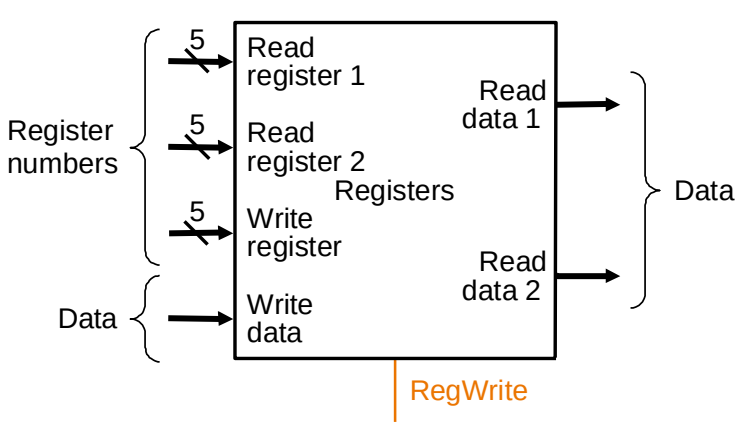
Datapath

Datapath - Visualizing

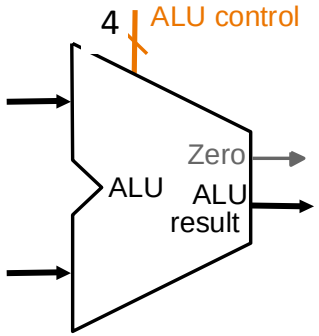


Instruction <- MEM[PC]
PC <- PC + 4

Datapath – R-type Instructions

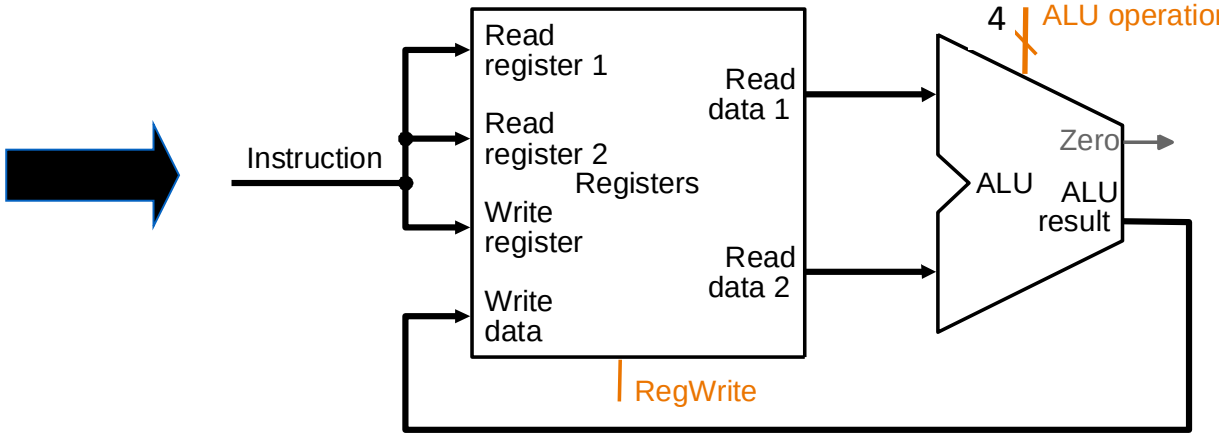


a. Registers



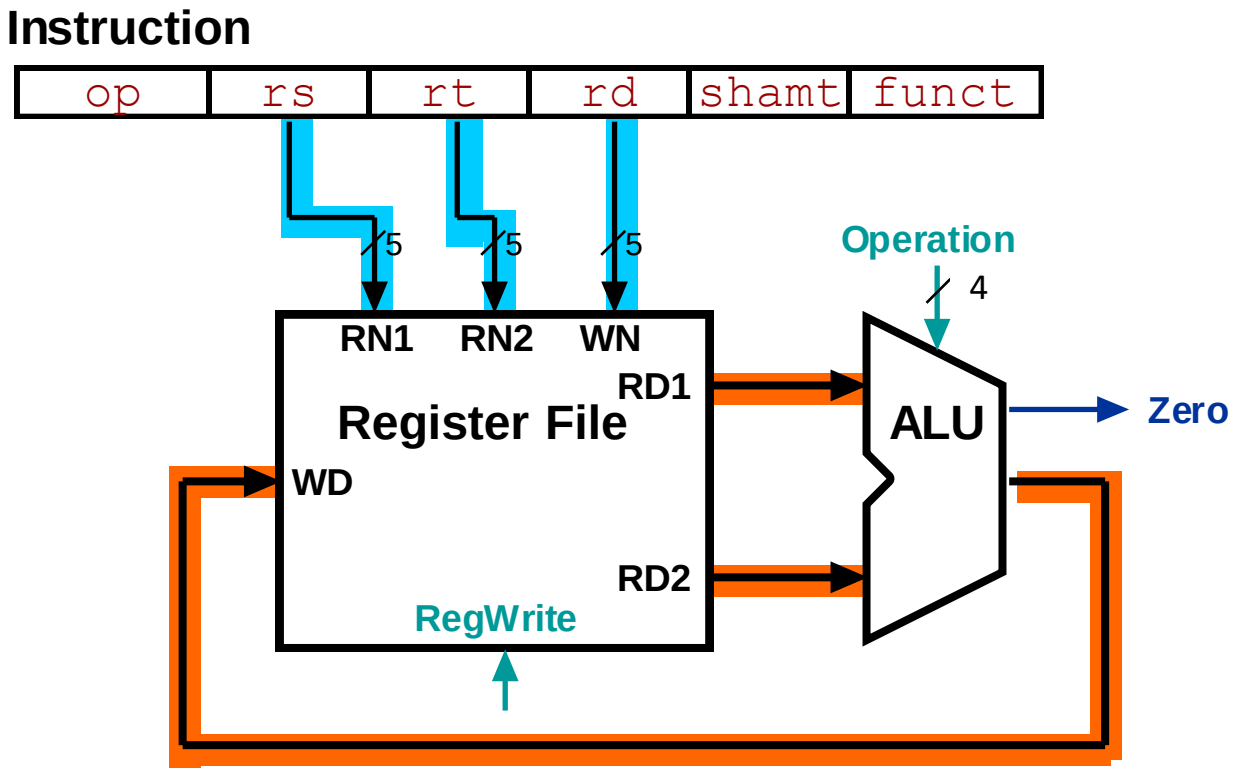
b. ALU

Two elements used to implement R-type instructions



Datapath

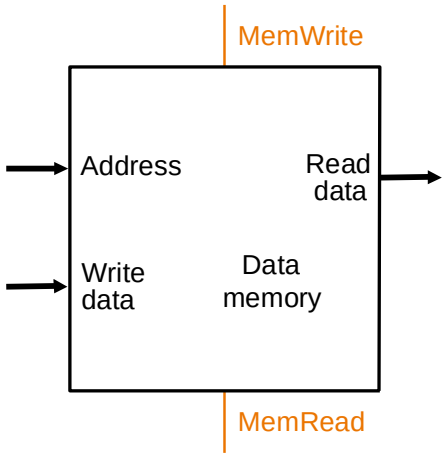
Datapath - Visualizing



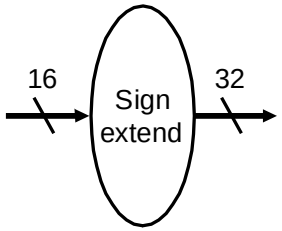

```
add rd, rs, rt
```

R[rd] <- R[rs] + R[rt];

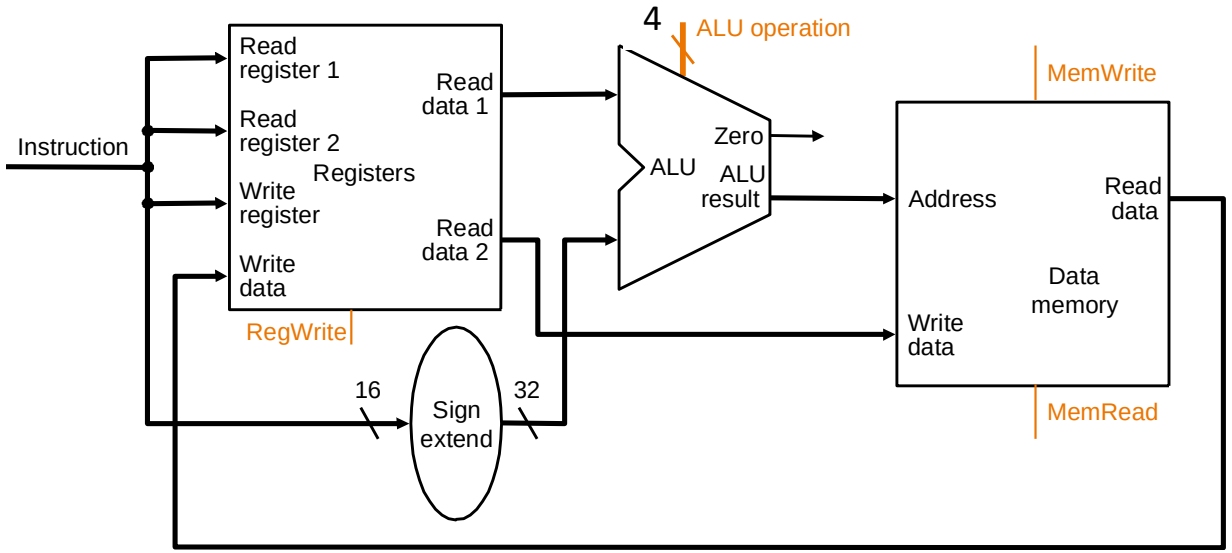
Datapath – Load/Store Instruction



a. Data memory unit



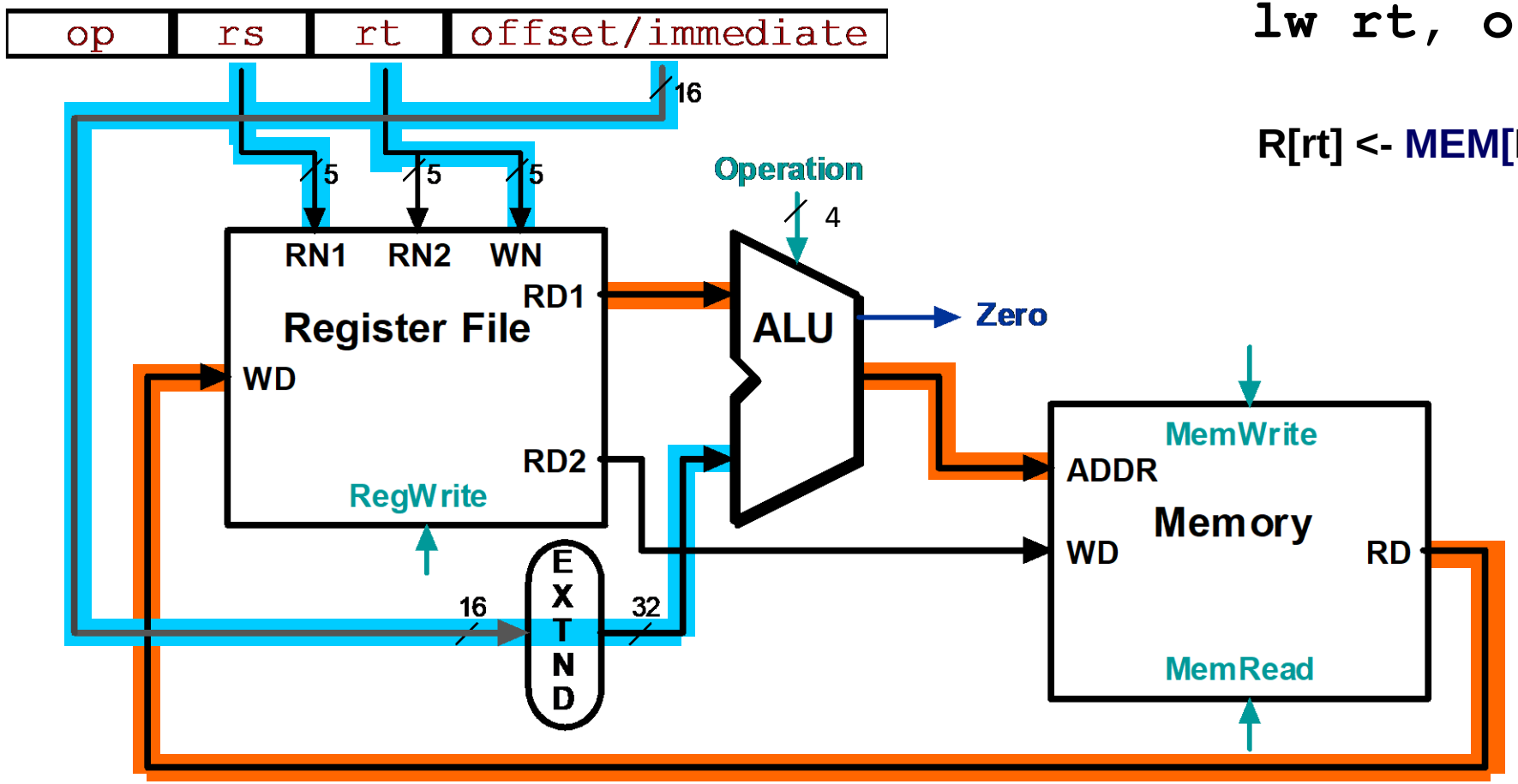
b. Sign-extension unit



Two additional elements used to implement load/stores

Datapath

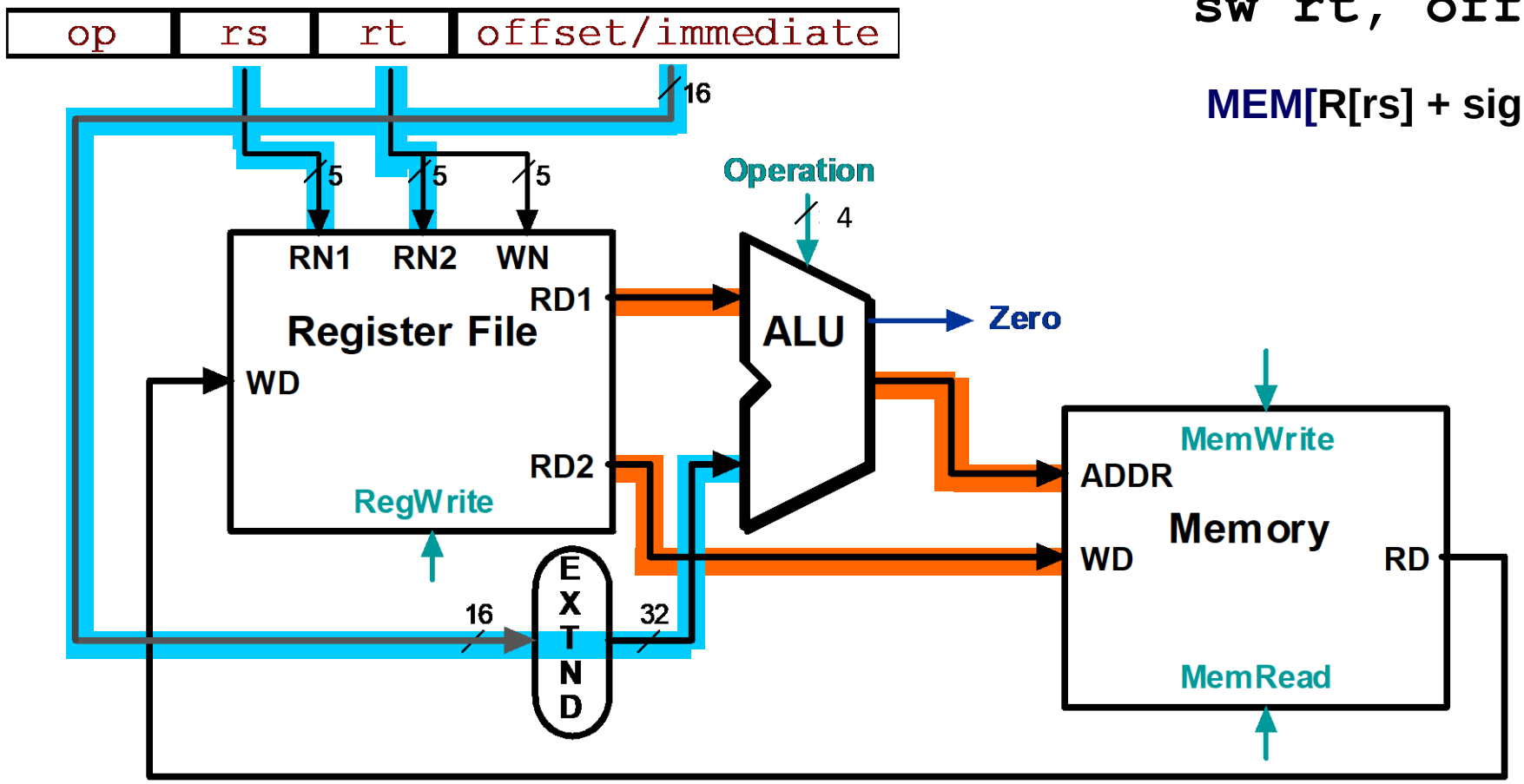
Datapath - Visualizing




```
lw rt, offset(rs)
```

R[rt] <- MEM[R[rs] + s_extend(offset)];

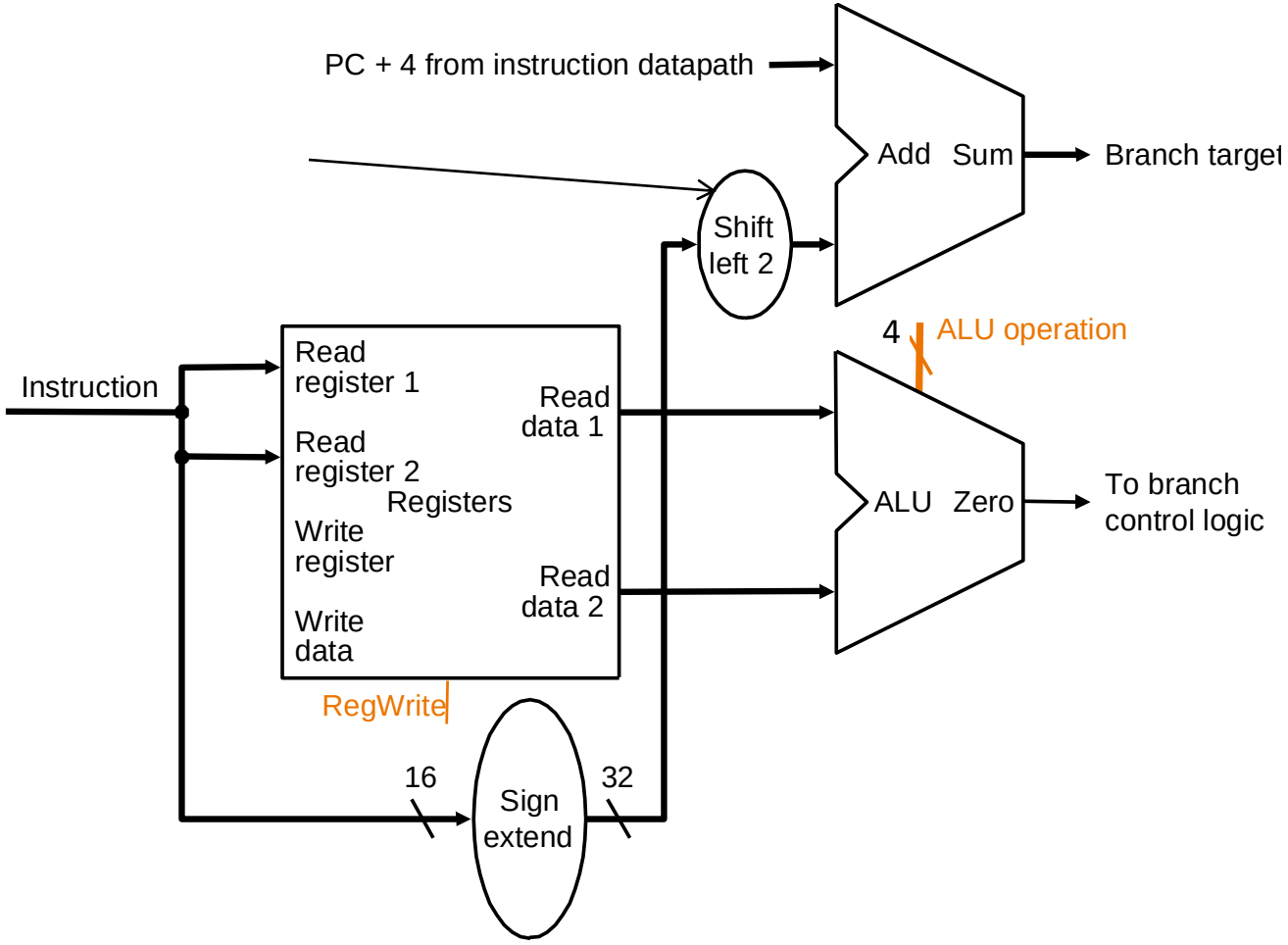
Datapath - Visualizing



`sw rt, offset(rs)`

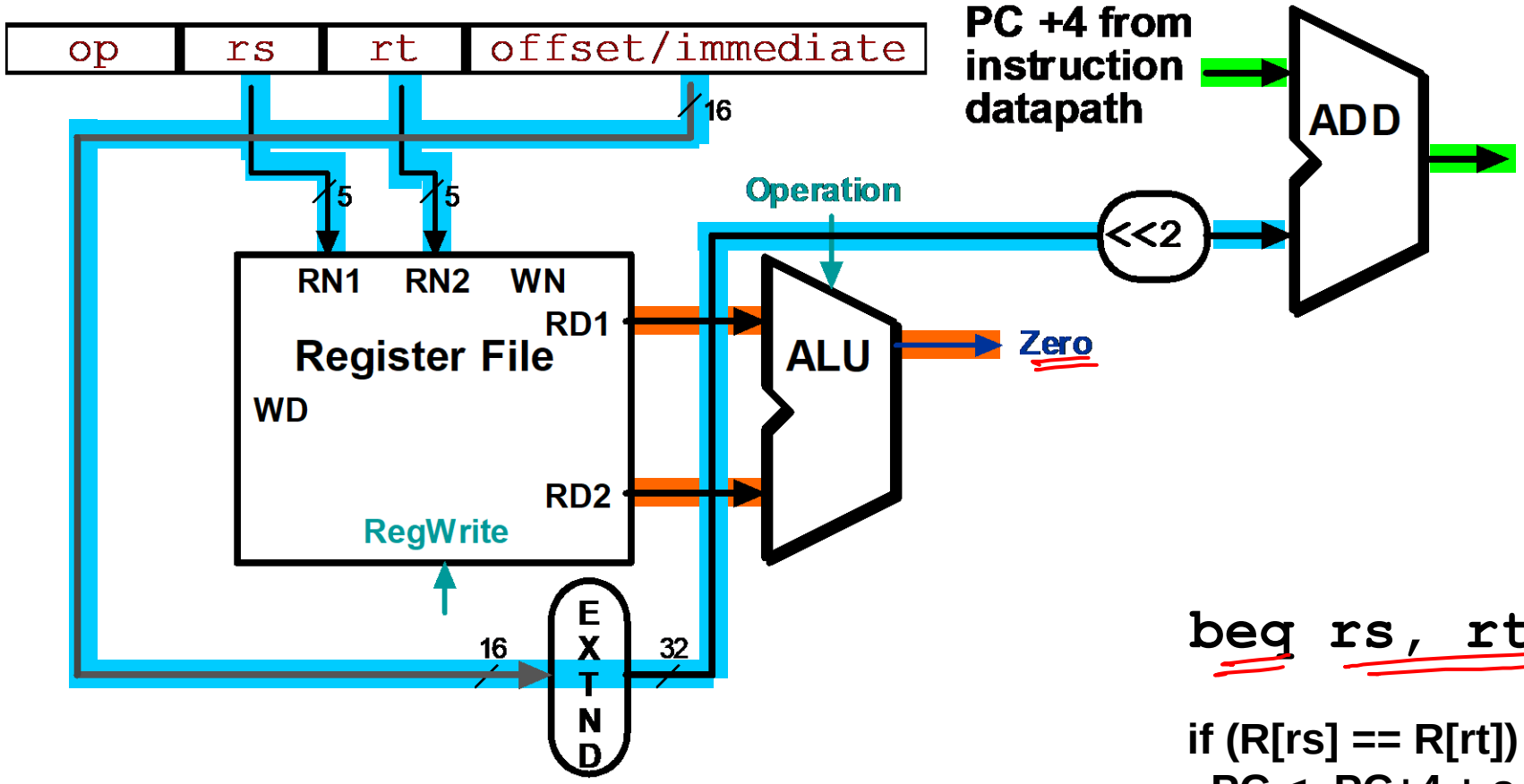
MEM[R[rs] + sign_extend(offset)] <- R[rt]

Datapath - Branch Instruction



No shift hardware required:
simply connect wires from input to output, each shifted left 2 bits

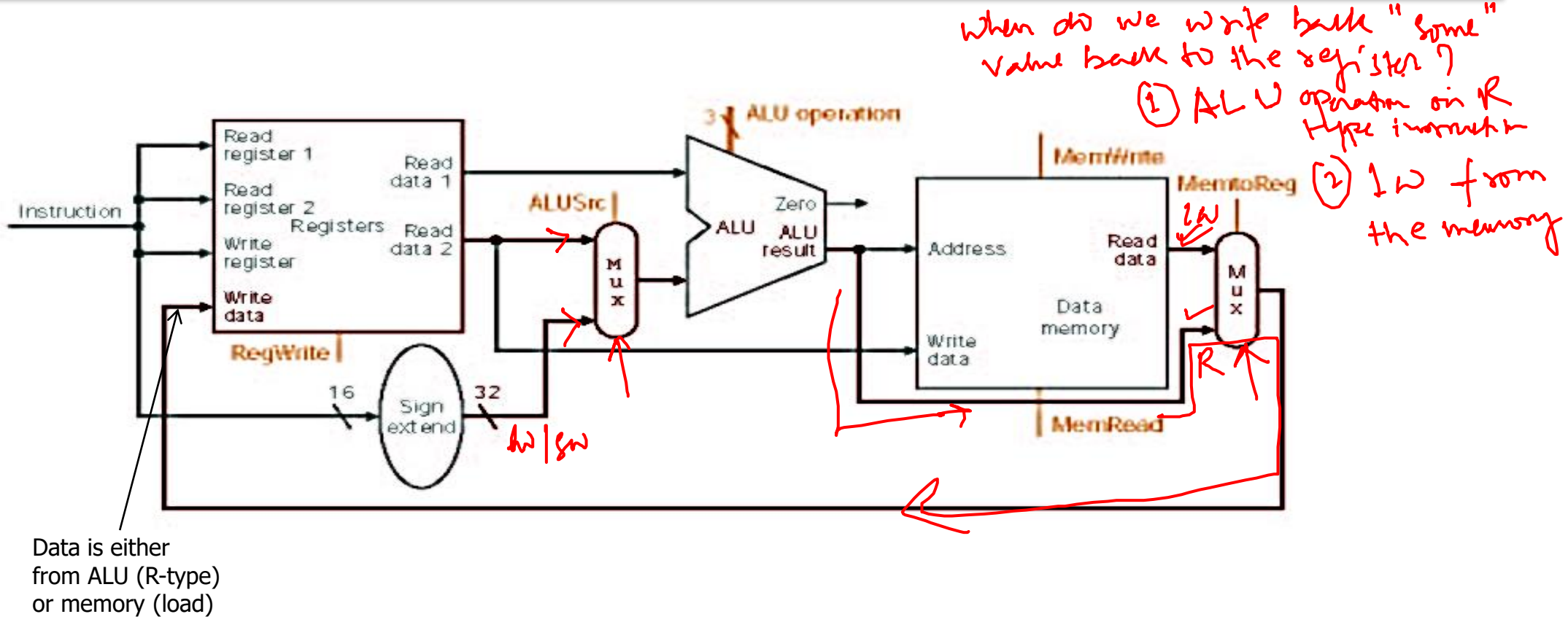
Datapath - Visualizing




```
beq rs, rt, offset
```

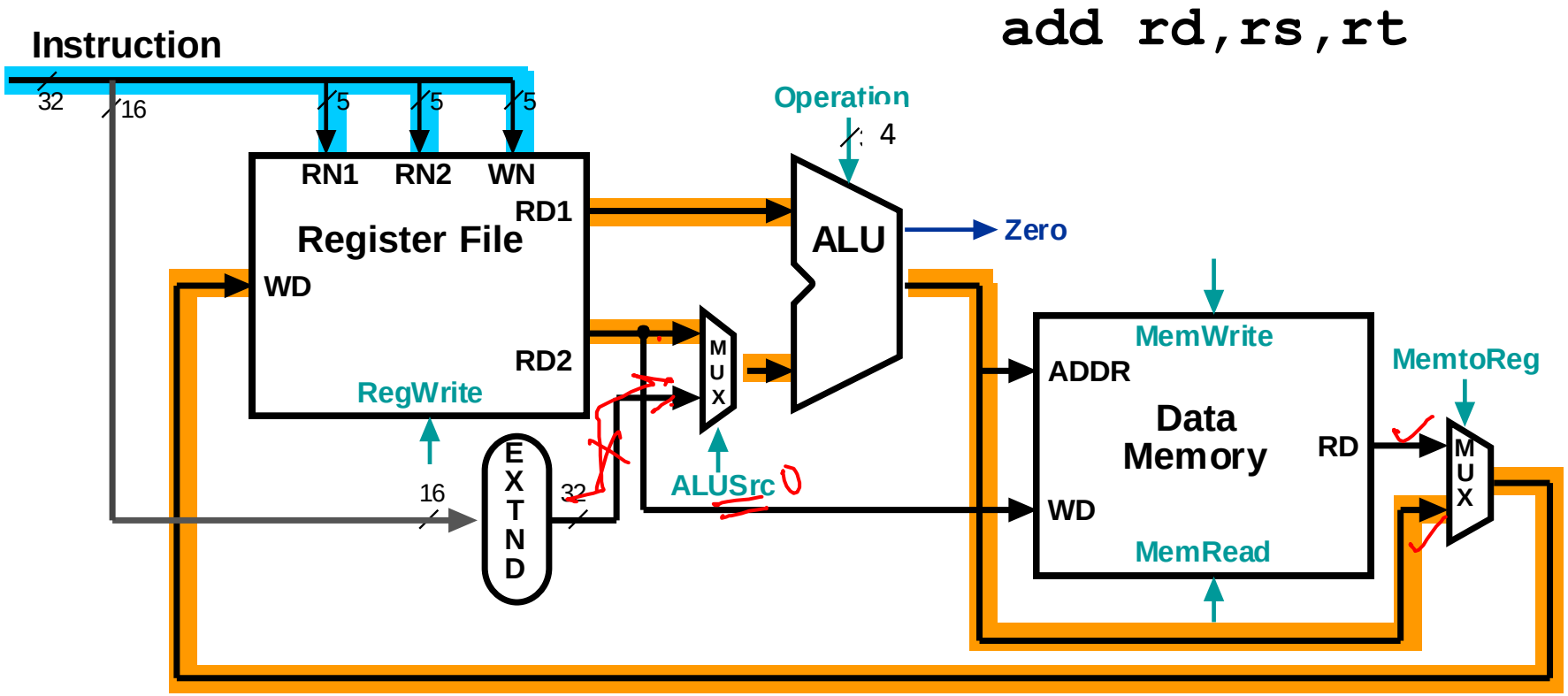
if (R[rs] == R[rt]) then
  PC <- PC+4 + s_extend(offset<<2)

MIPS Datapath-I: Single Cycle

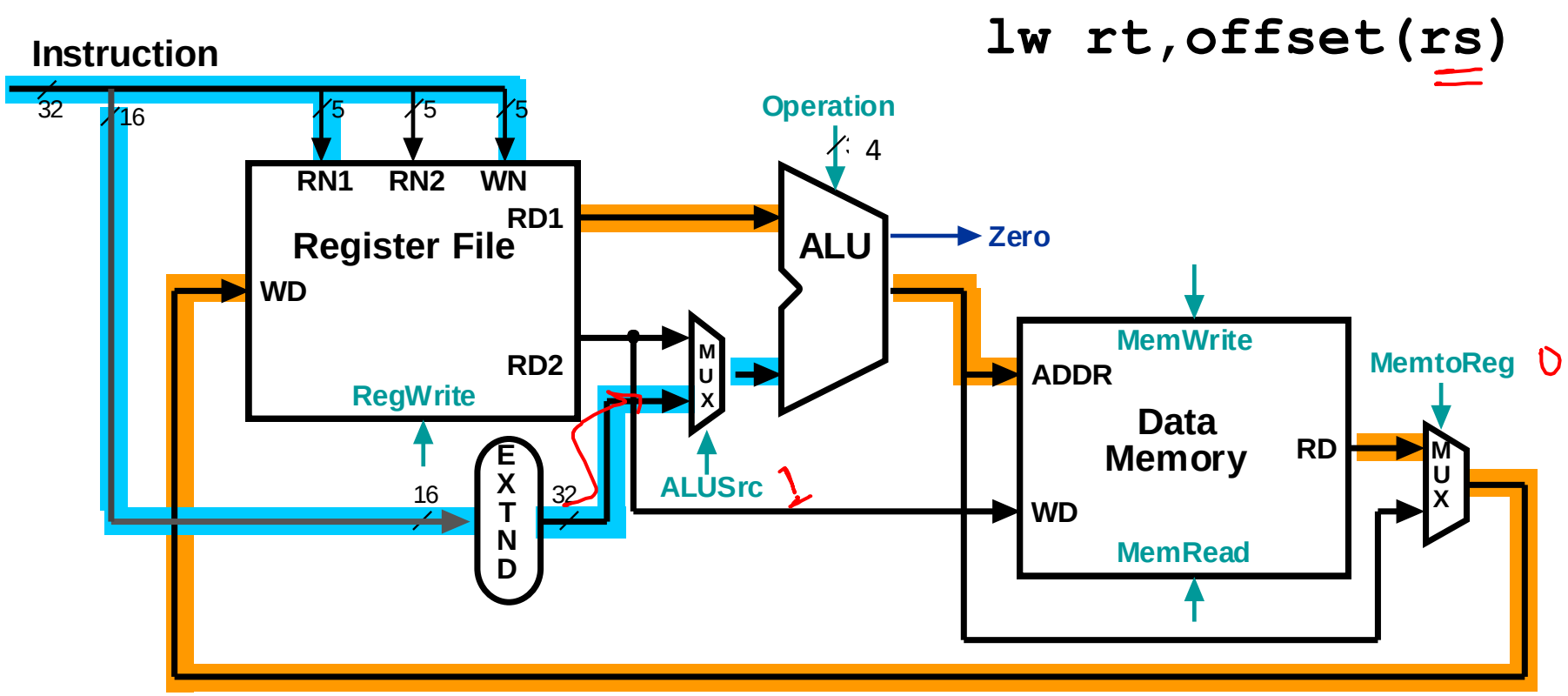


Combining the datapaths for R-type instructions and load/stores using two multiplexors

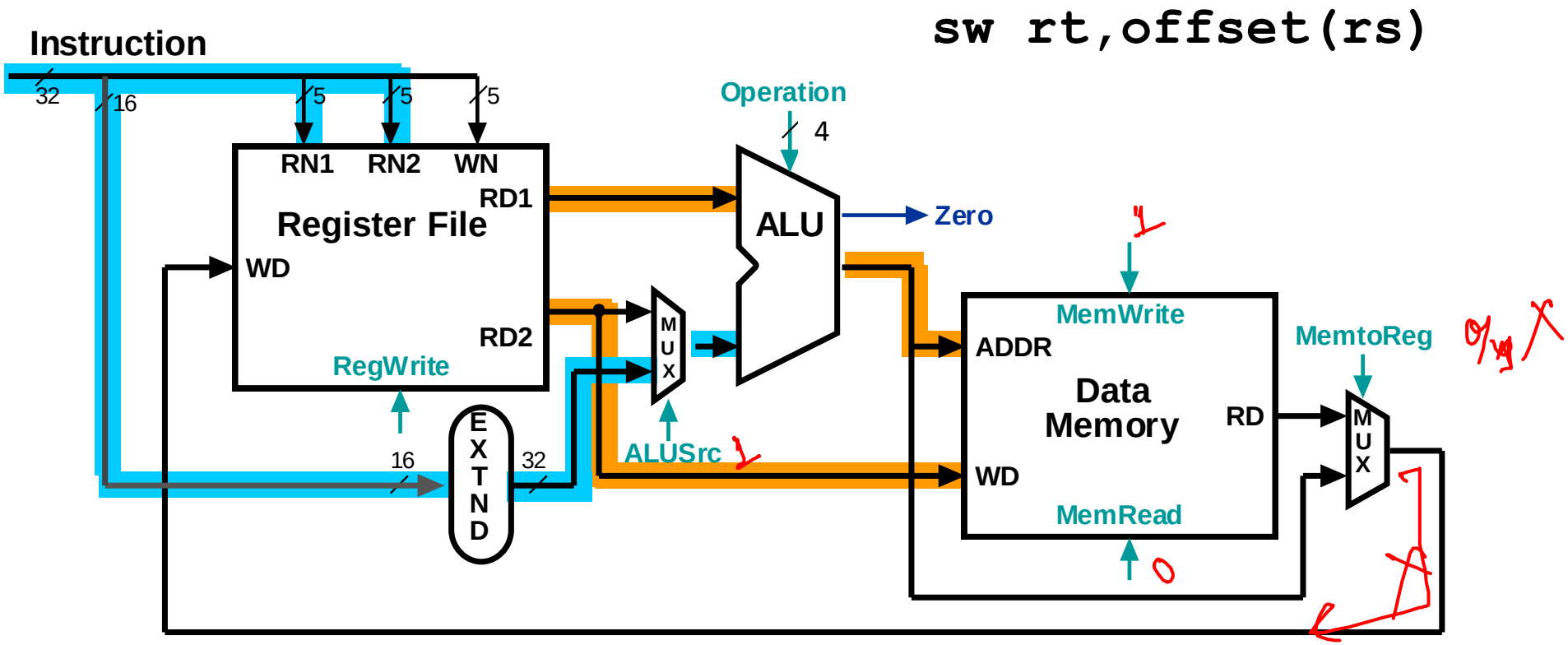
R-type Instruction



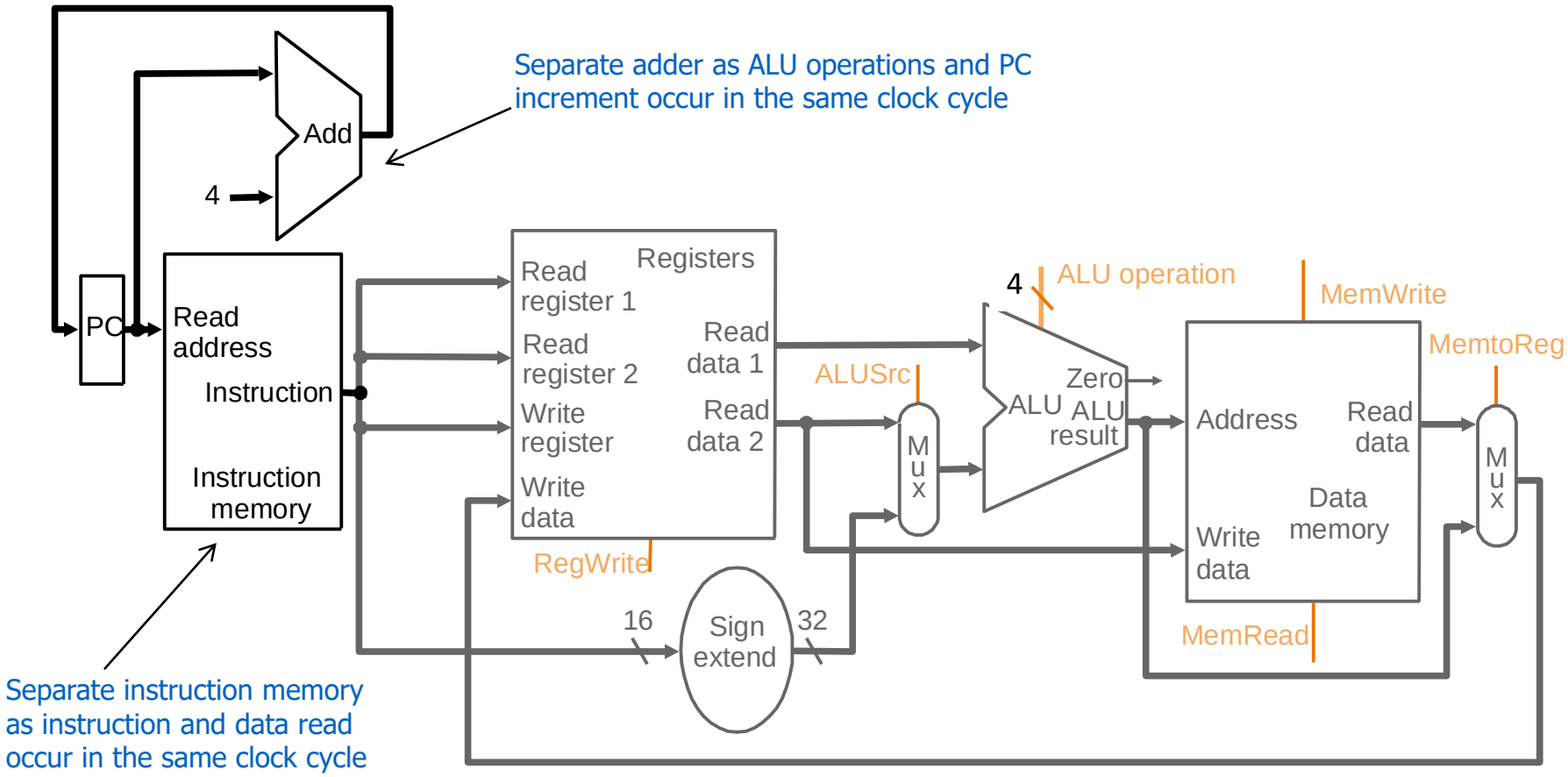
Load Instruction



Store Instruction

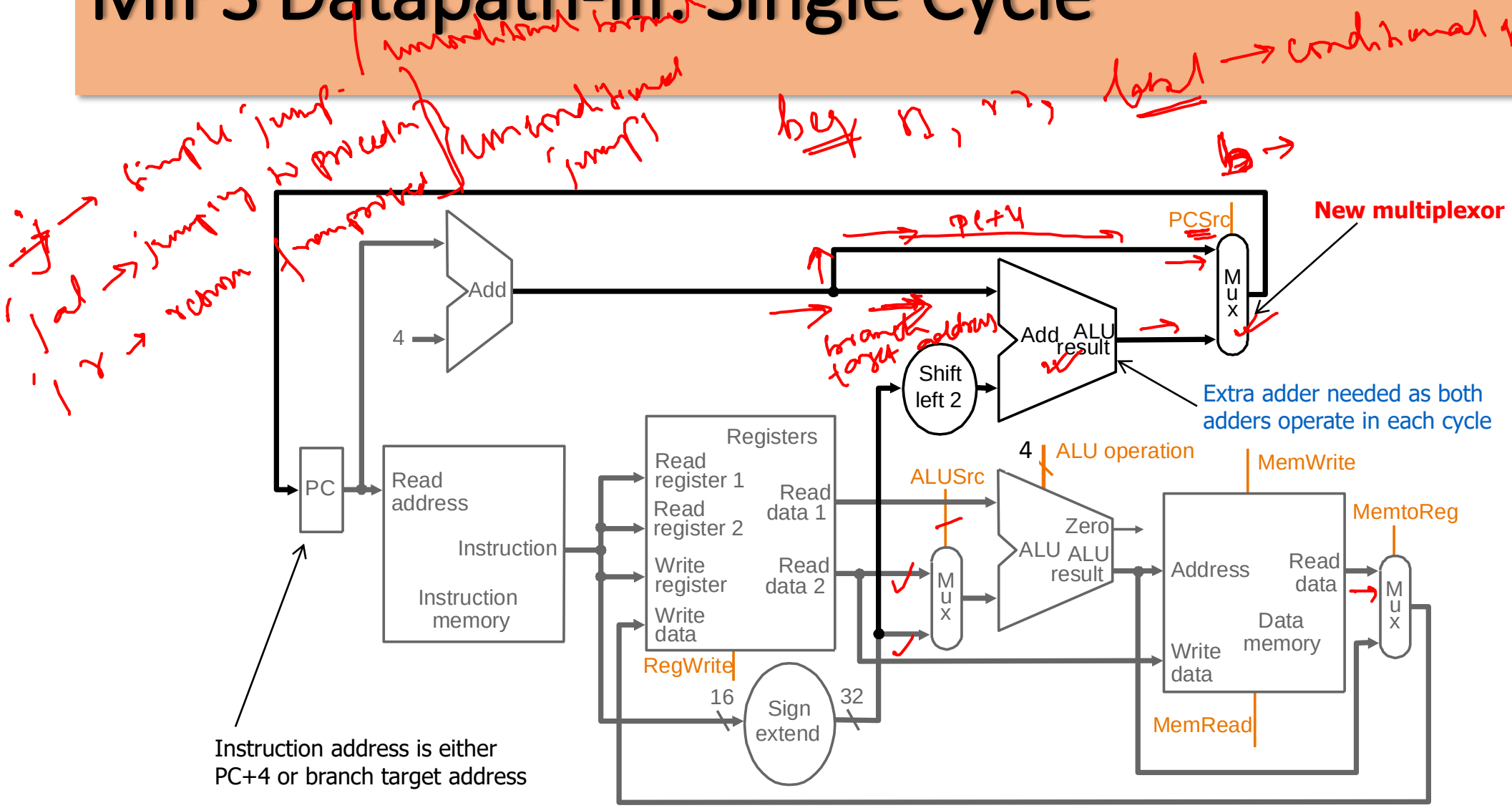


MIPS Datapath-II: Single Cycle



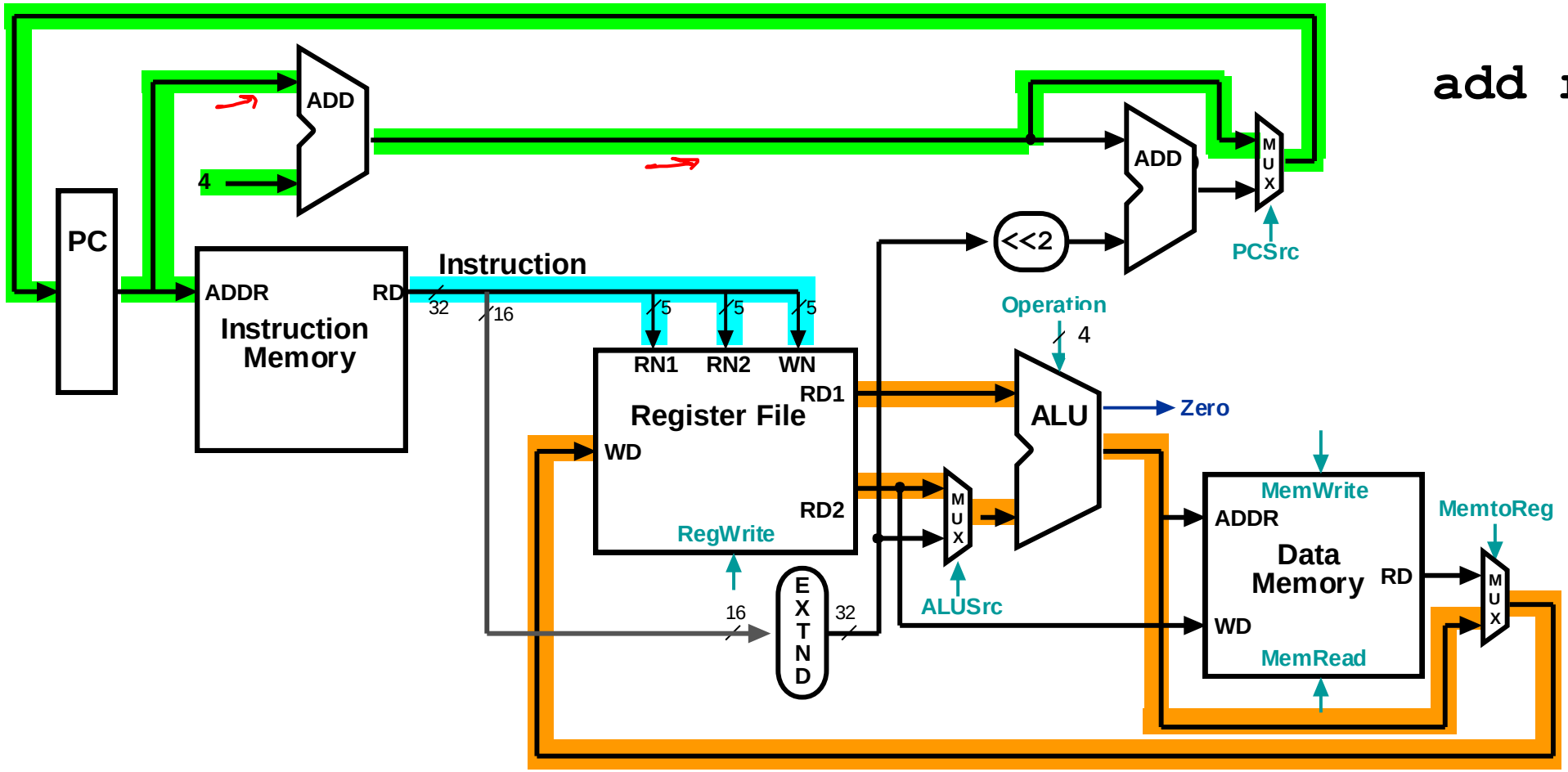
Adding instruction fetch

MIPS Datapath-III: Single Cycle



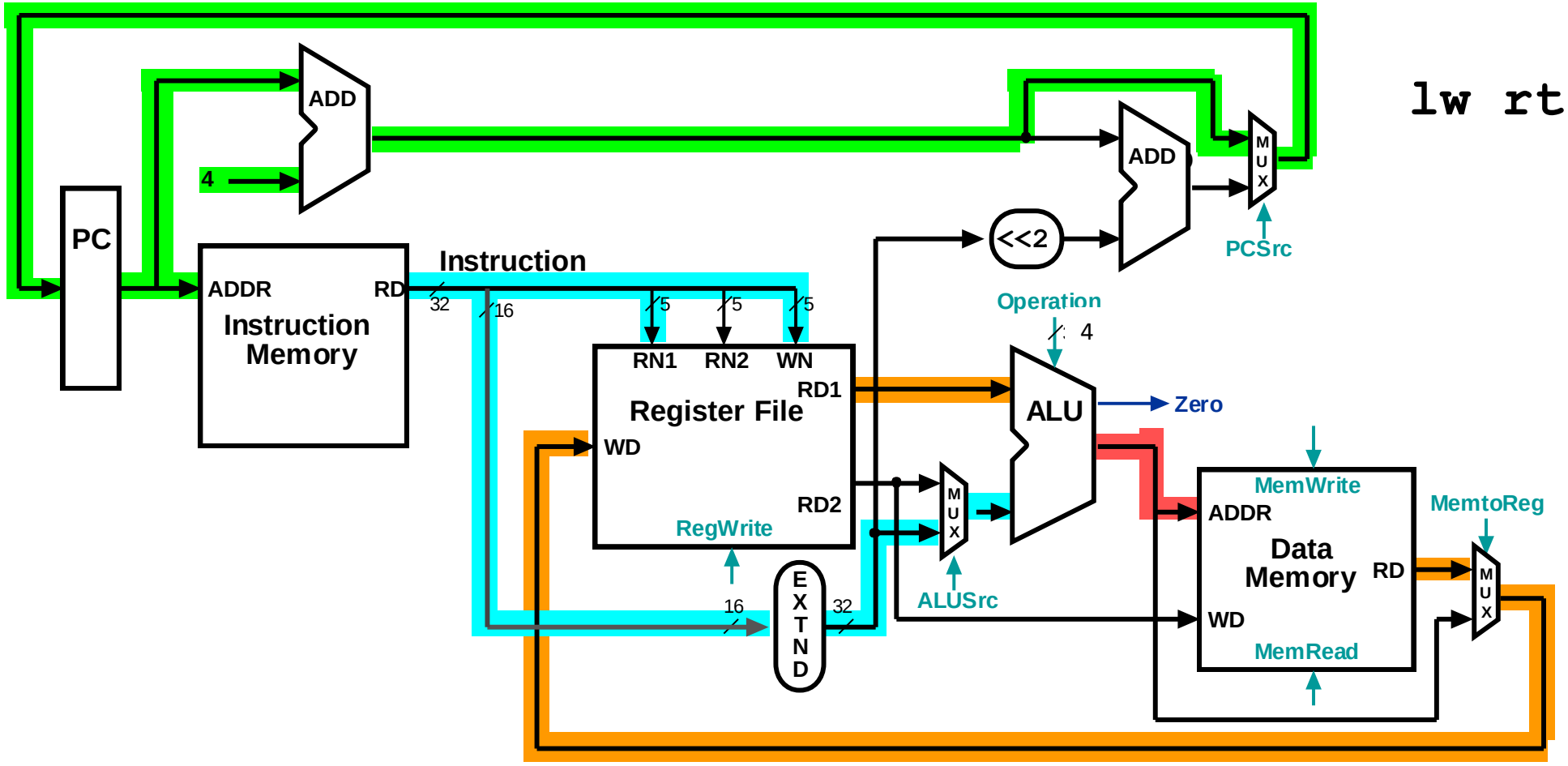
Adding branch capability and another multiplexor

Datapath - `add`



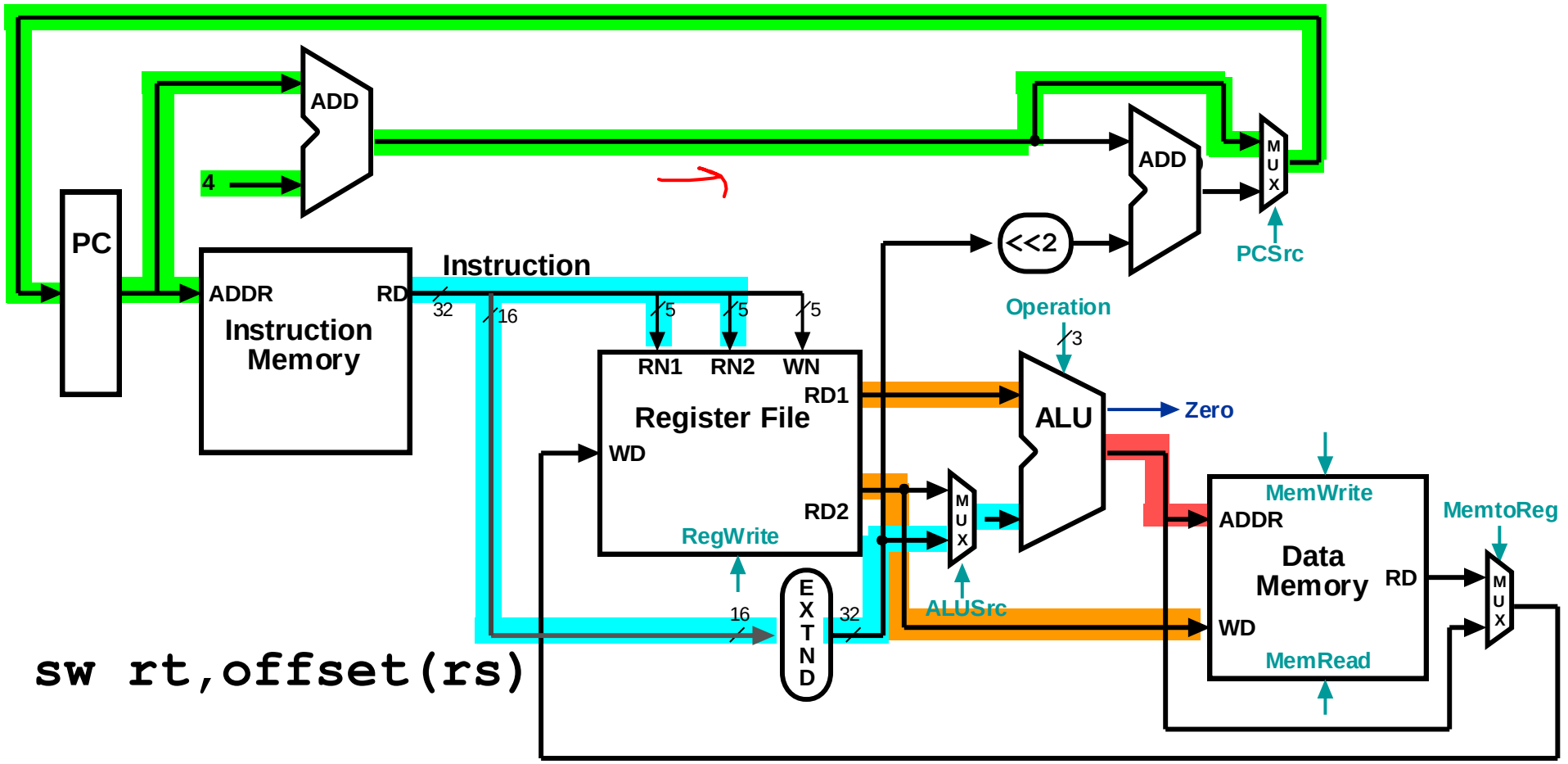

```
add rd, rs, rt
```

Datapath - `lw`



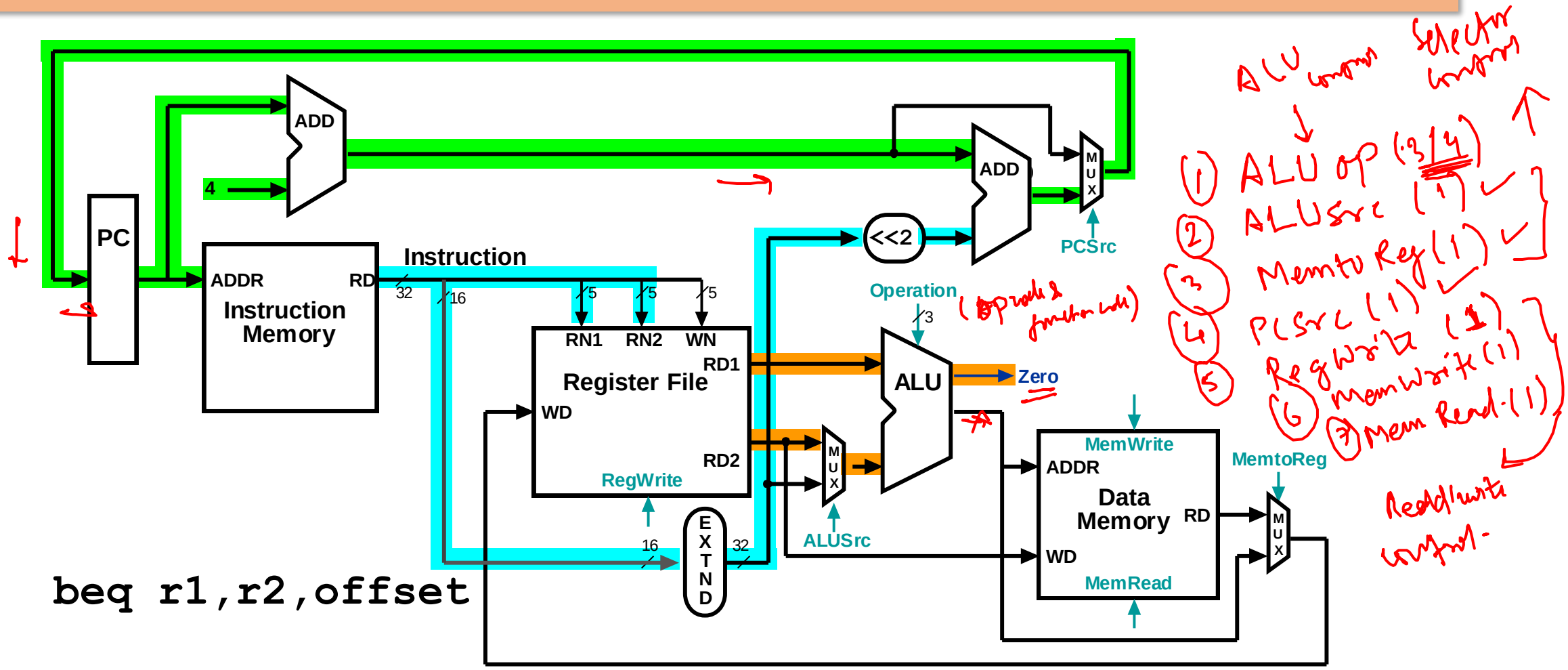

```
lw rt,offset(rs)
```

Datapath - `sw`



`sw rt,offset(rs)`

Datapath - beq



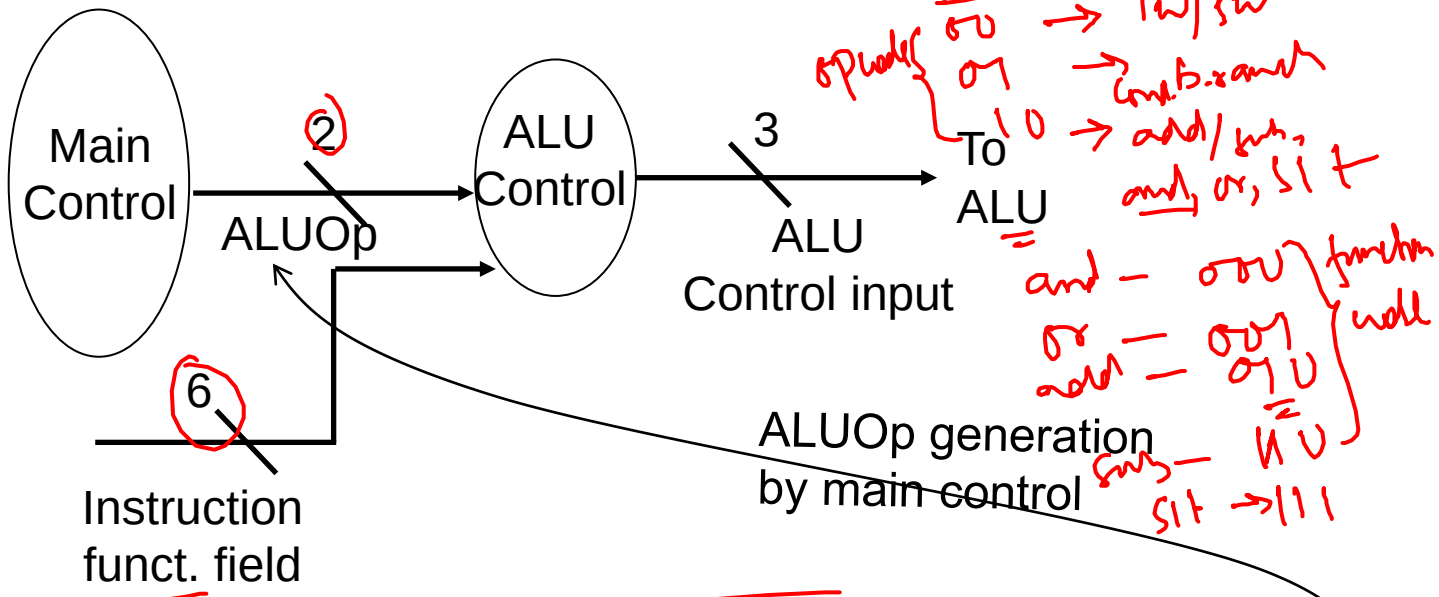
Control

- Control unit takes input from
  - the instruction opcode bits
- Control unit generates
  - ALU control input
  - write enable (possibly, read enable also) signals for each storage element
  - selector controls for each multiplexor

ALU Control

- Plan to control ALU: main control sends a 2-bit ALUOp control field to the ALU control. Based on ALUOp and funct field of instruction the ALU control generates the 3-bit ALU control field

ALU control field	Function
000	and
001	or
010	add
110	sub
111	slt



- ALU must perform
  - *add* for load/stores (ALUOp 00)
  - *sub* for branches (ALUOp 01)
  - one of *and*, *or*, *add*, *sub*, *slt* for R-type instructions, depending on the instruction's 6-bit funct field (ALUOp 10)

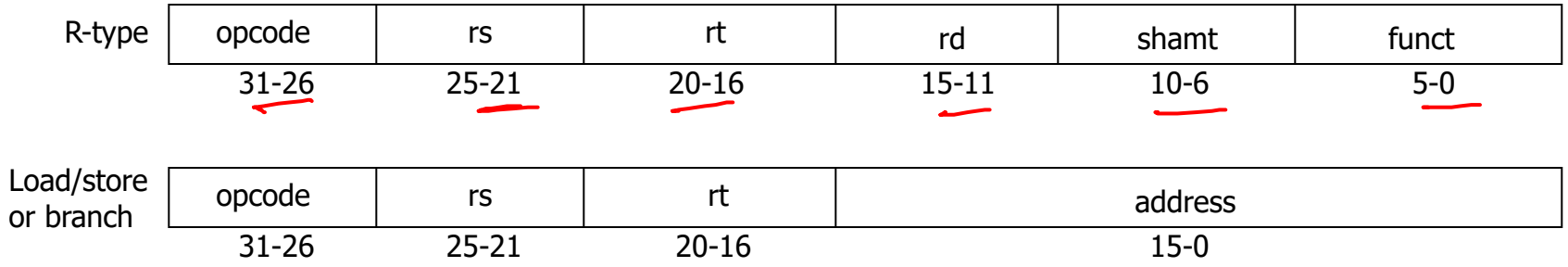
Setting ALU Control Bits

Instruction opcode	AluOp	Instruction operation	Funct Field	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch eq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less	101010	set on less	111

*

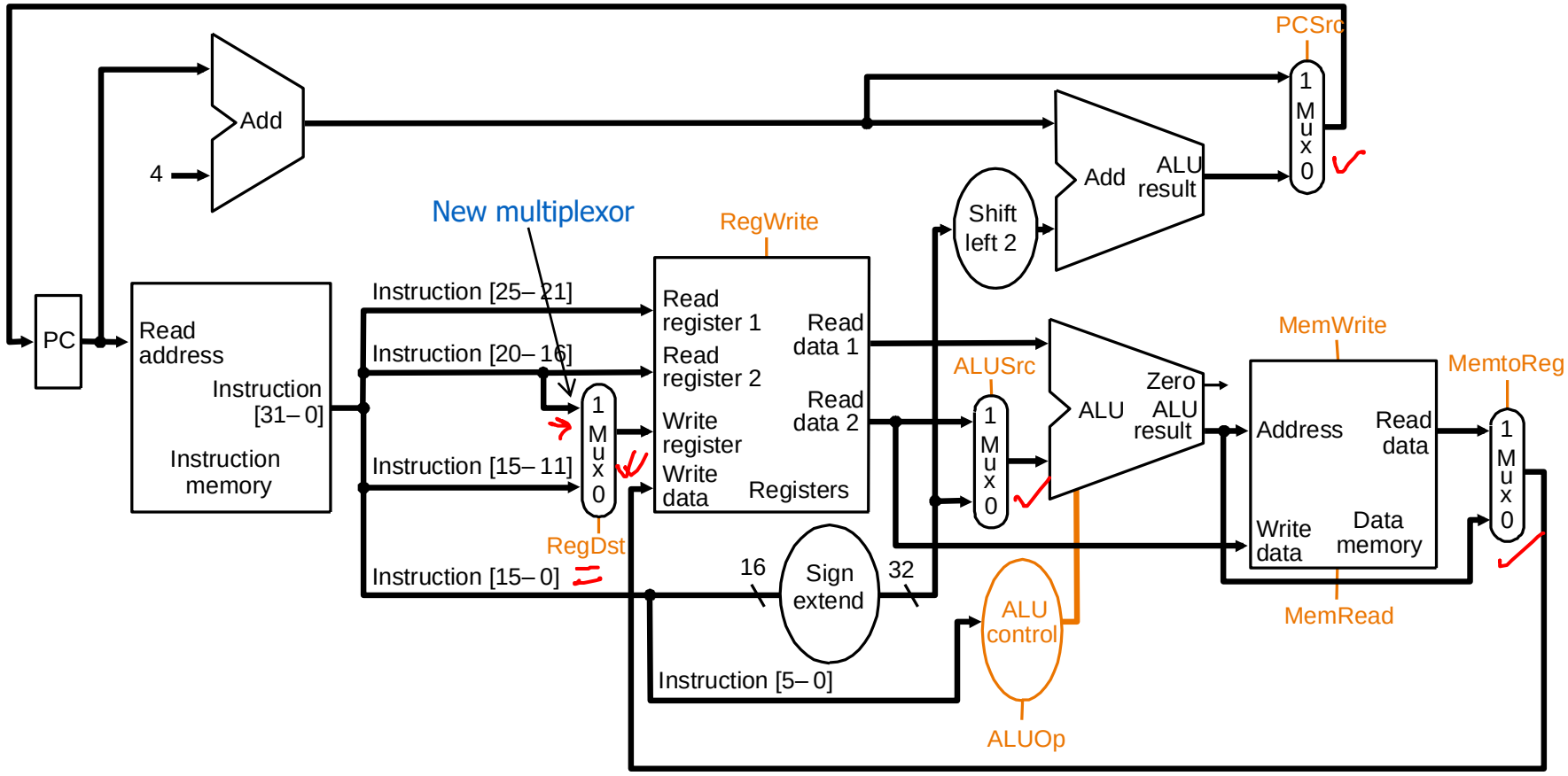
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Designing the Main Control Unit



- Observations about MIPS instruction format
  1. opcode is always in bits 31-26
  2. two registers to be read are always rs (bits 25-21) and rt (bits 20-16)
  3. base register for load/stores is always rs (bits 25-21)
  4. 16-bit offset for branch equal and load/store is always bits 15-0
  5. destination register for loads is in bits 20-16 (rt) while for R-type instructions it is in bits 15-11 (rd) (*will require multiplexor to select*)

Datapath with Control - I

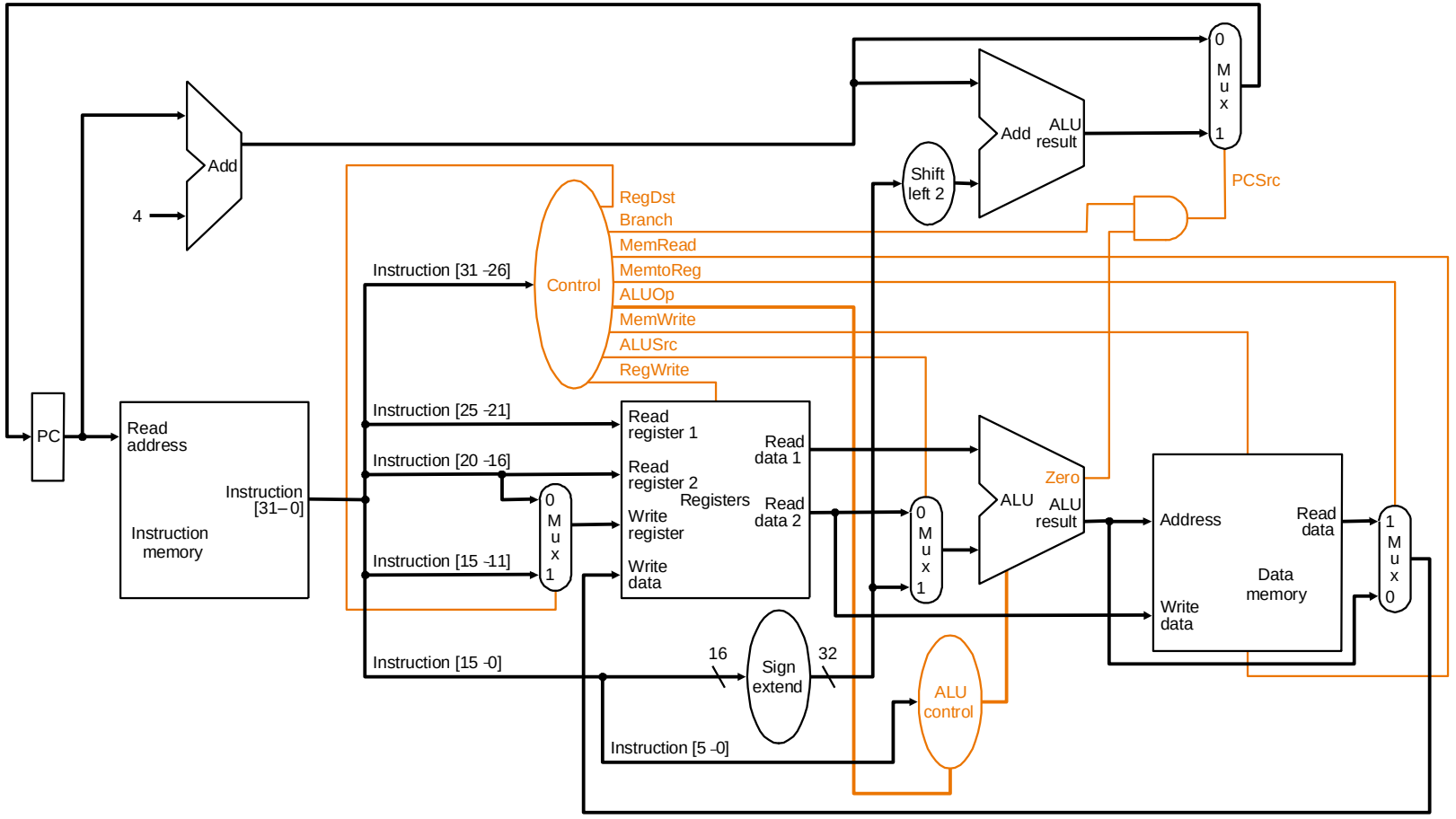


Adding control to the MIPS Datapath III (and a new multiplexor to select field to specify destination register): what are the functions of the 9 control signals?

Control Signals

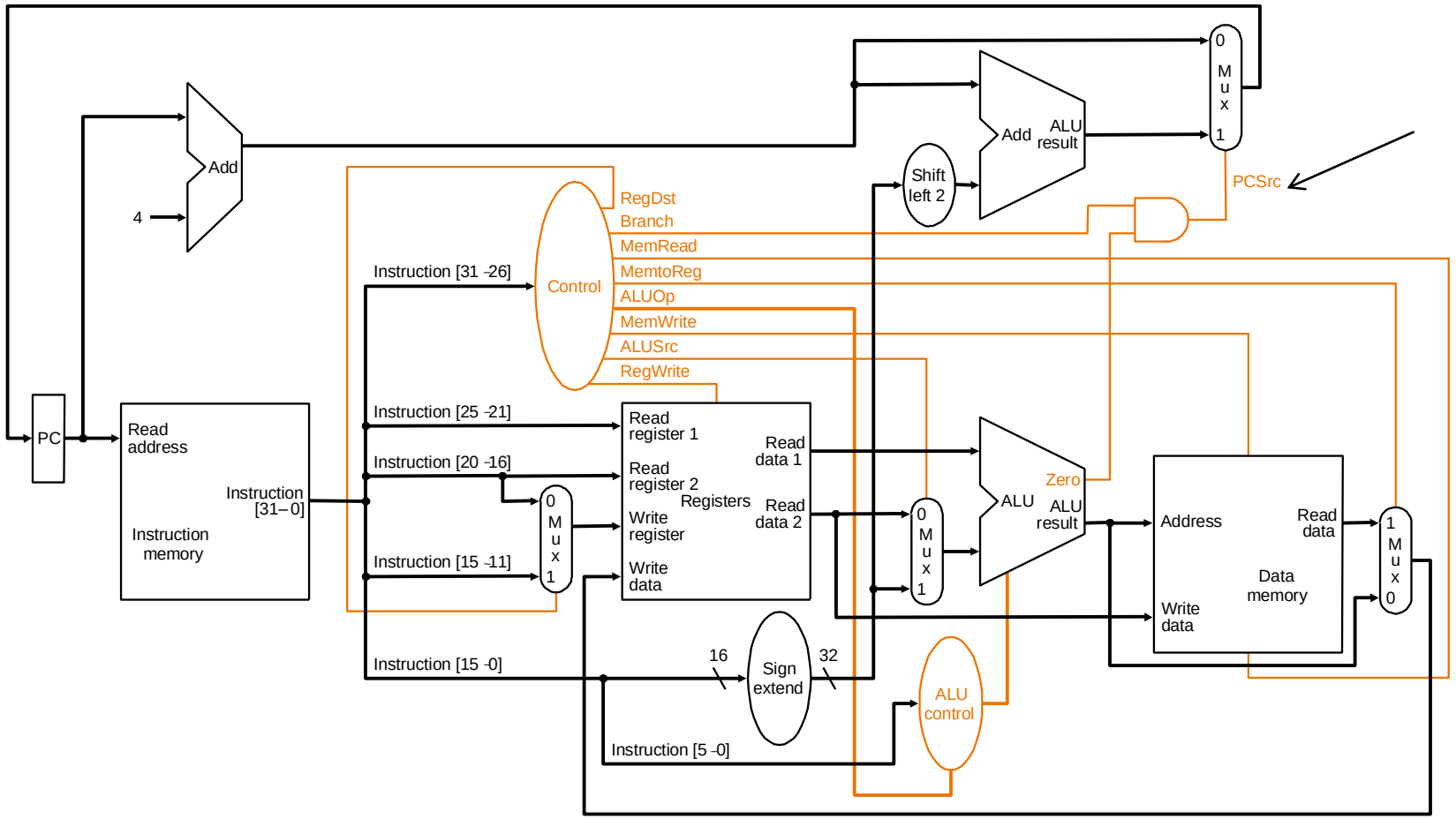
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20-16)	The register destination number for the Write register comes from the rd field (bits 15-11)
RegWrite	None	The register on the Write register input is written with the value on the Write data input
AlUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that computes the branch target
MemRead	None	Data memory contents designated by the address input are put on the first Read data output
MemWrite	None	Data memory contents designated by the address input are replaced by the value of the Write data input
MemtoReg	The value fed to the register Write data input comes from the ALU	The value fed to the register Write data input comes from the data memory

Datapath with Control - II



MIPS datapath with the control unit: input to control is the 6-bit instruction opcode field, output is seven 1-bit signals and the 2-bit ALUOp signal

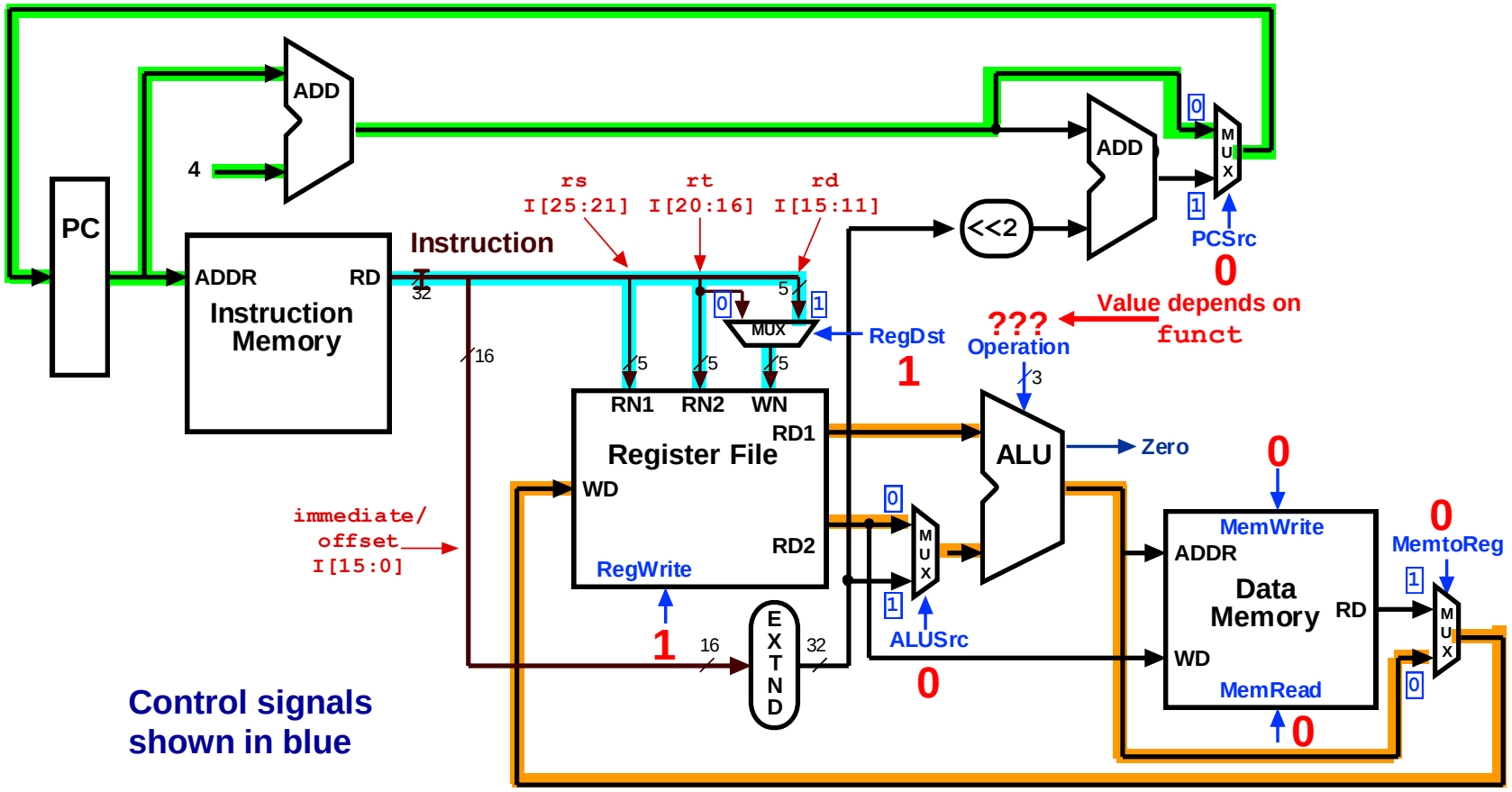
Datapath with Control – II (Contd…)



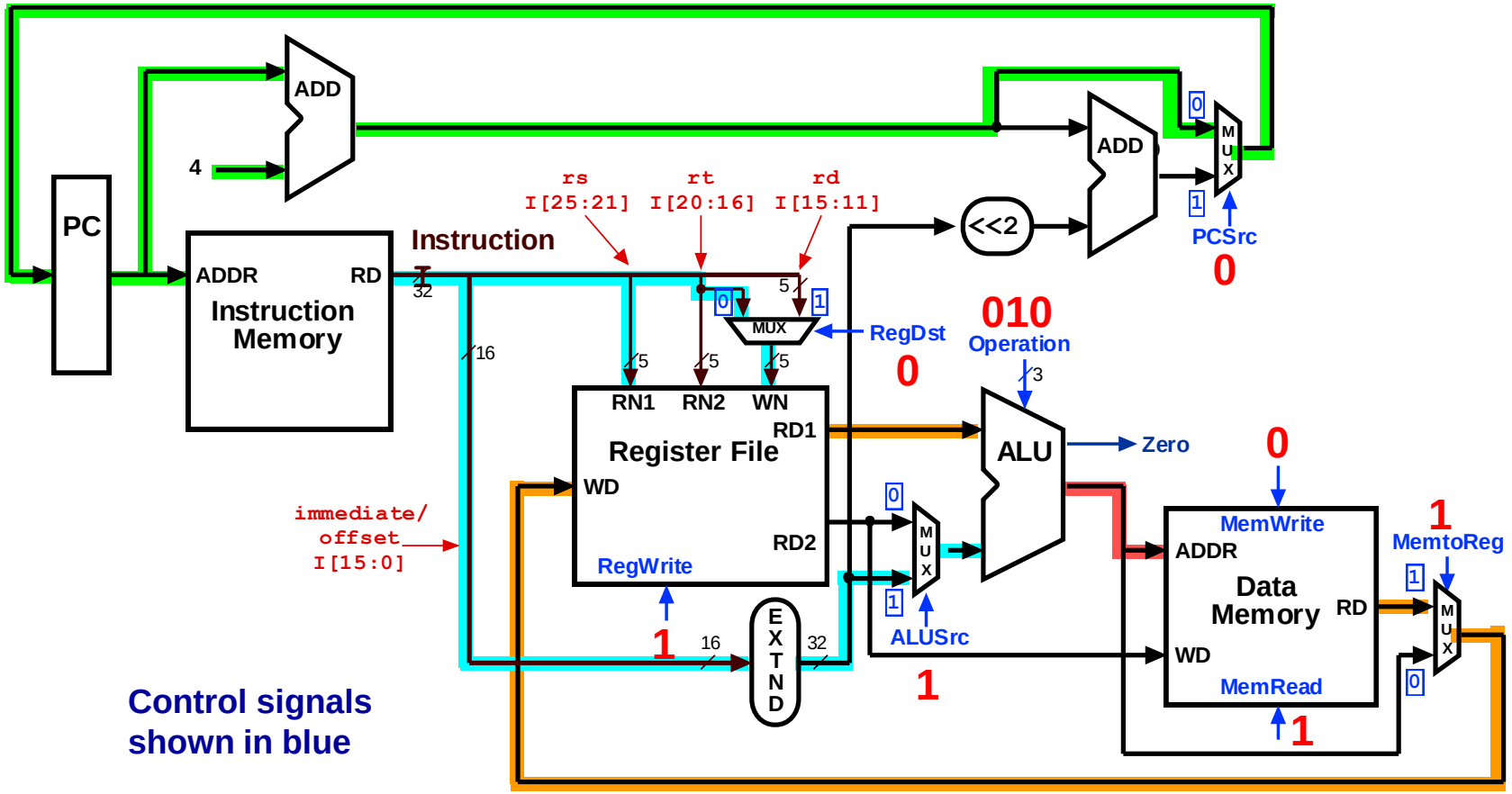
Determining control signals for the MIPS datapath based on instruction opcode

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

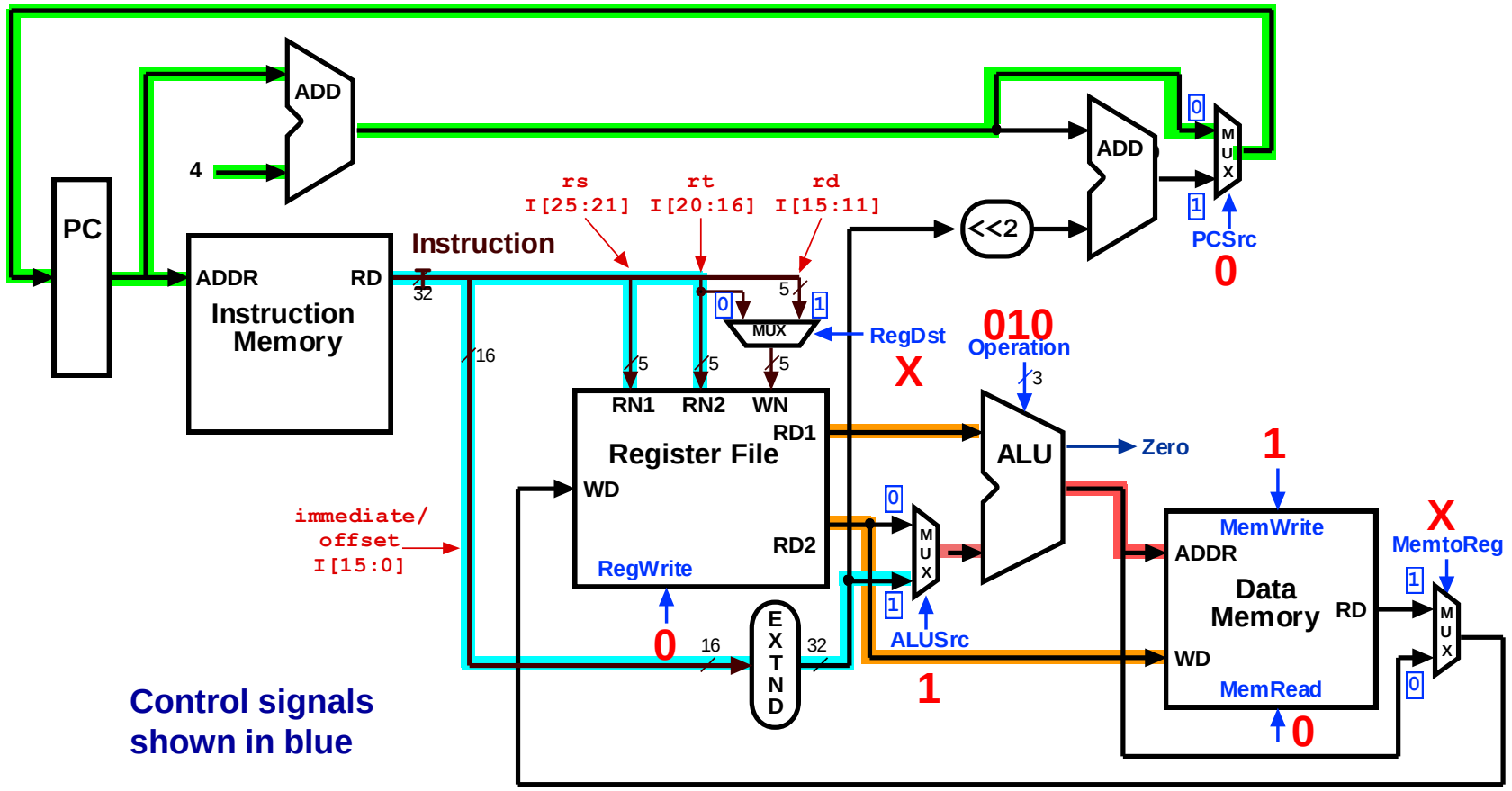
Control Signals: R-type instructions



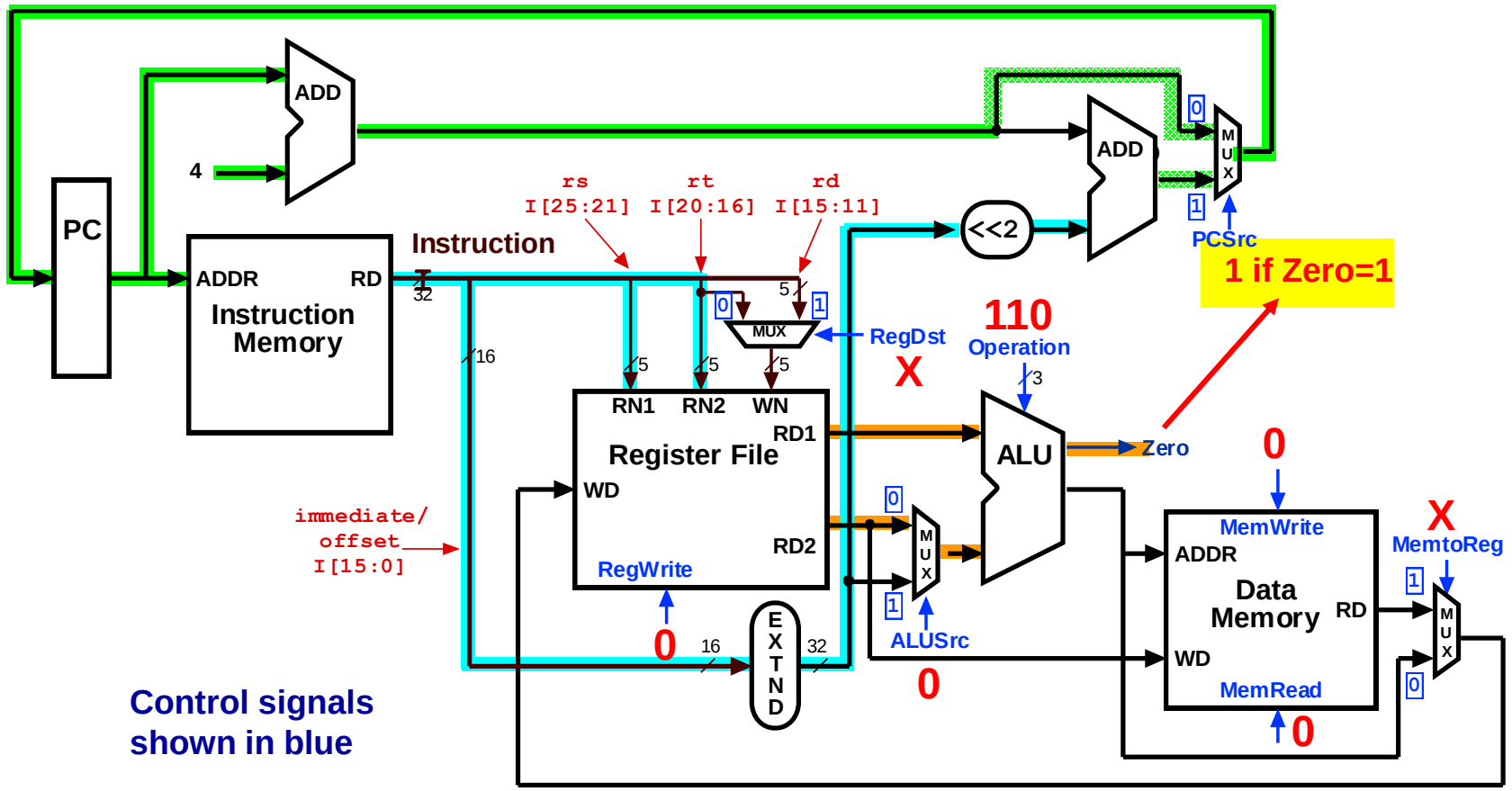
Control Signals: `lw` instruction



Control Signals: `sw` instruction



Control Signals: `beq` instruction

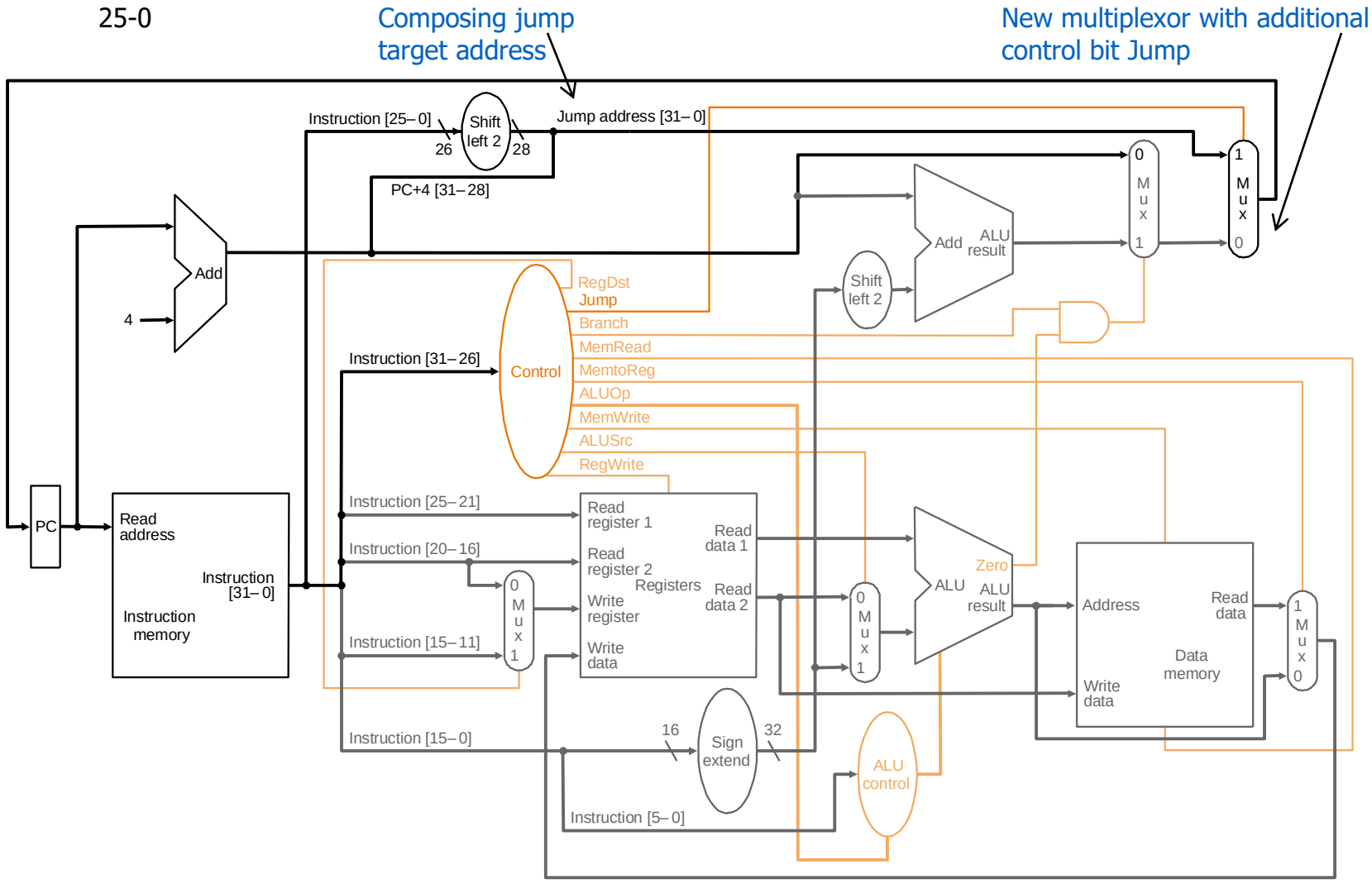


Datapath with Control - III

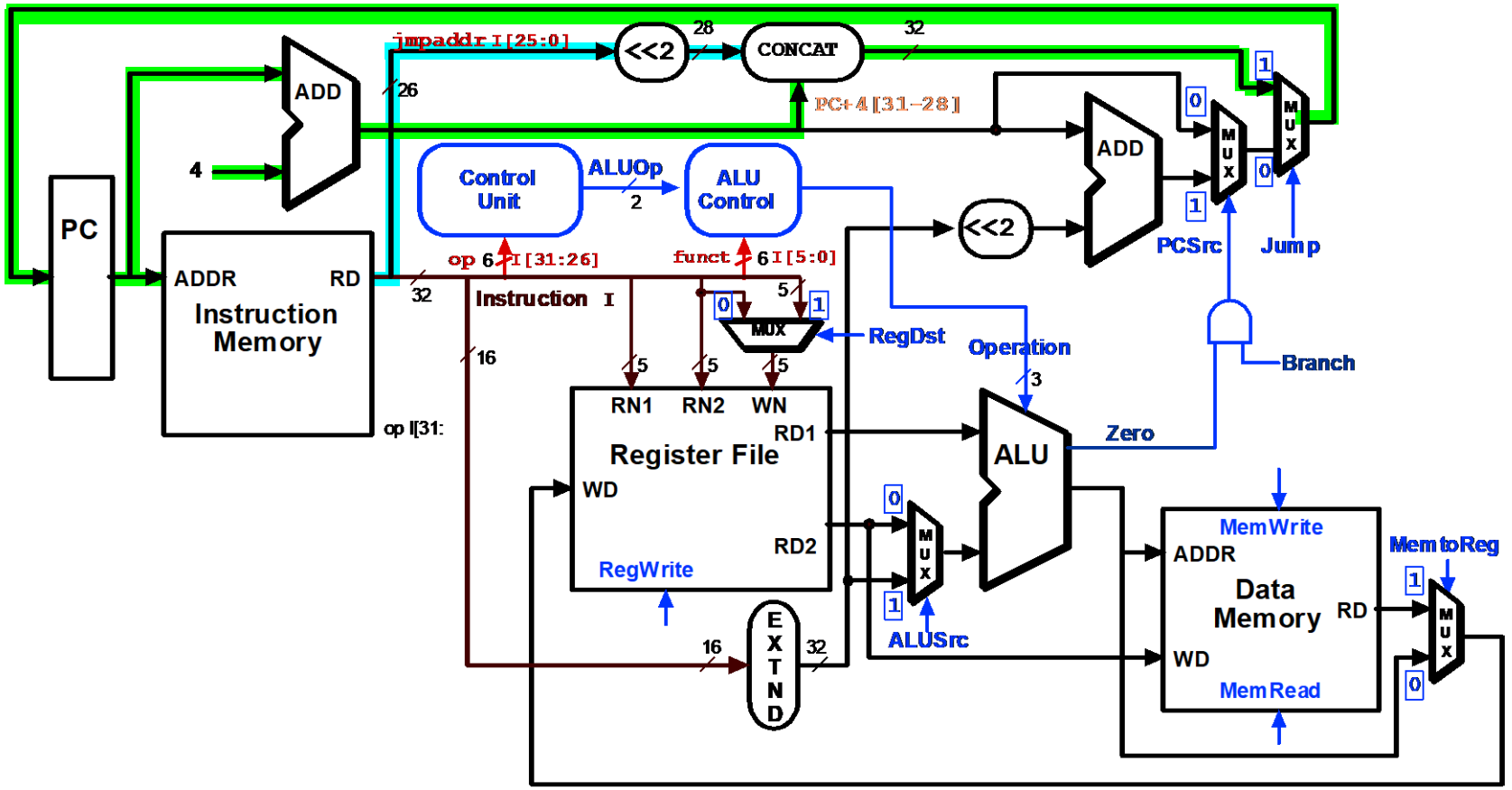
opcode	address
31-26	25-0

Jump

MIPS datapath extended to jumps: control unit generates new Jump control bit

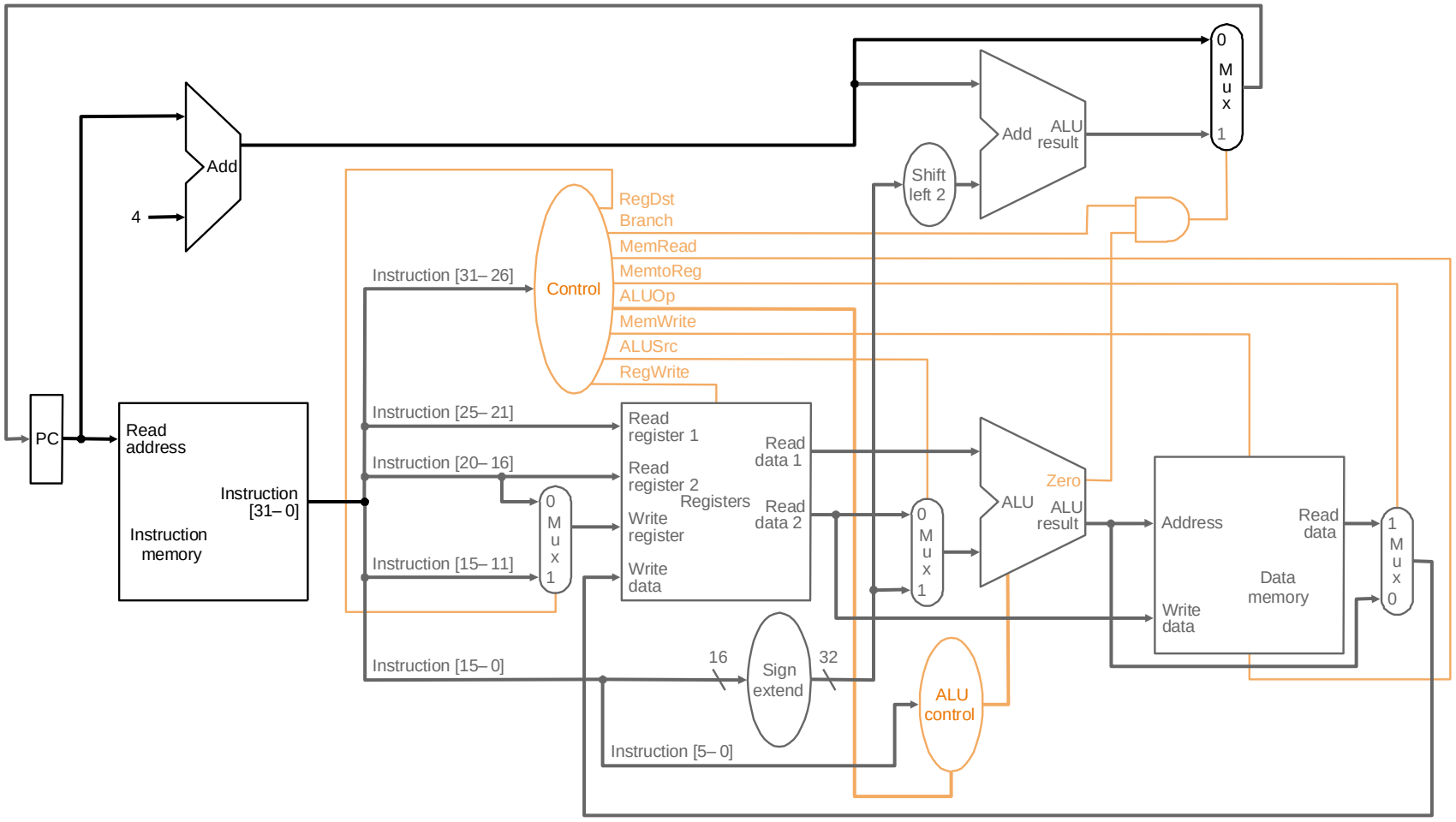


Datapath Executing - `j`



R-type Instruction: Step-1

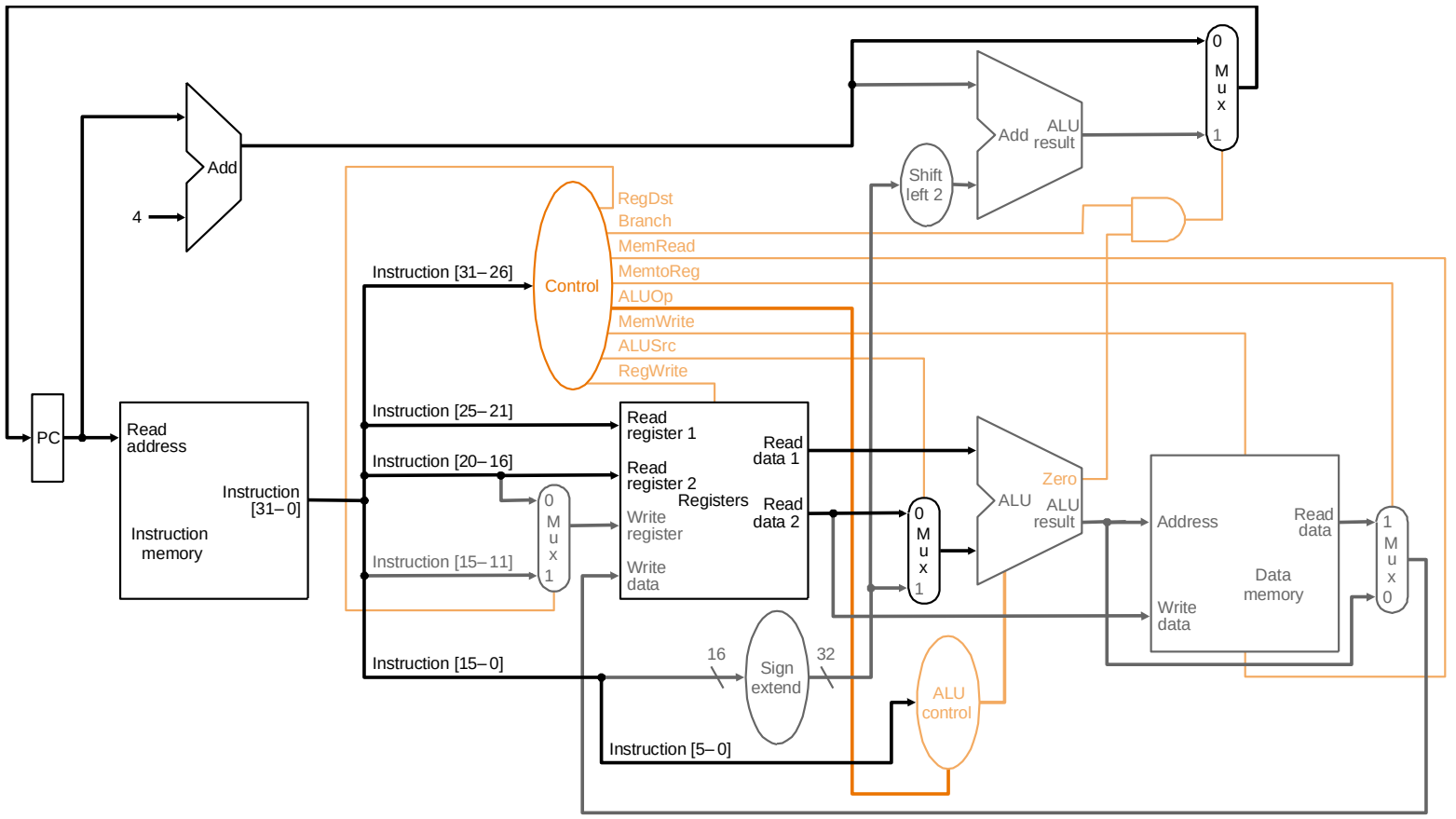
`add $t1, $t2, $t3`



Fetch instruction and increment PC count

R-type Instruction: Step-2

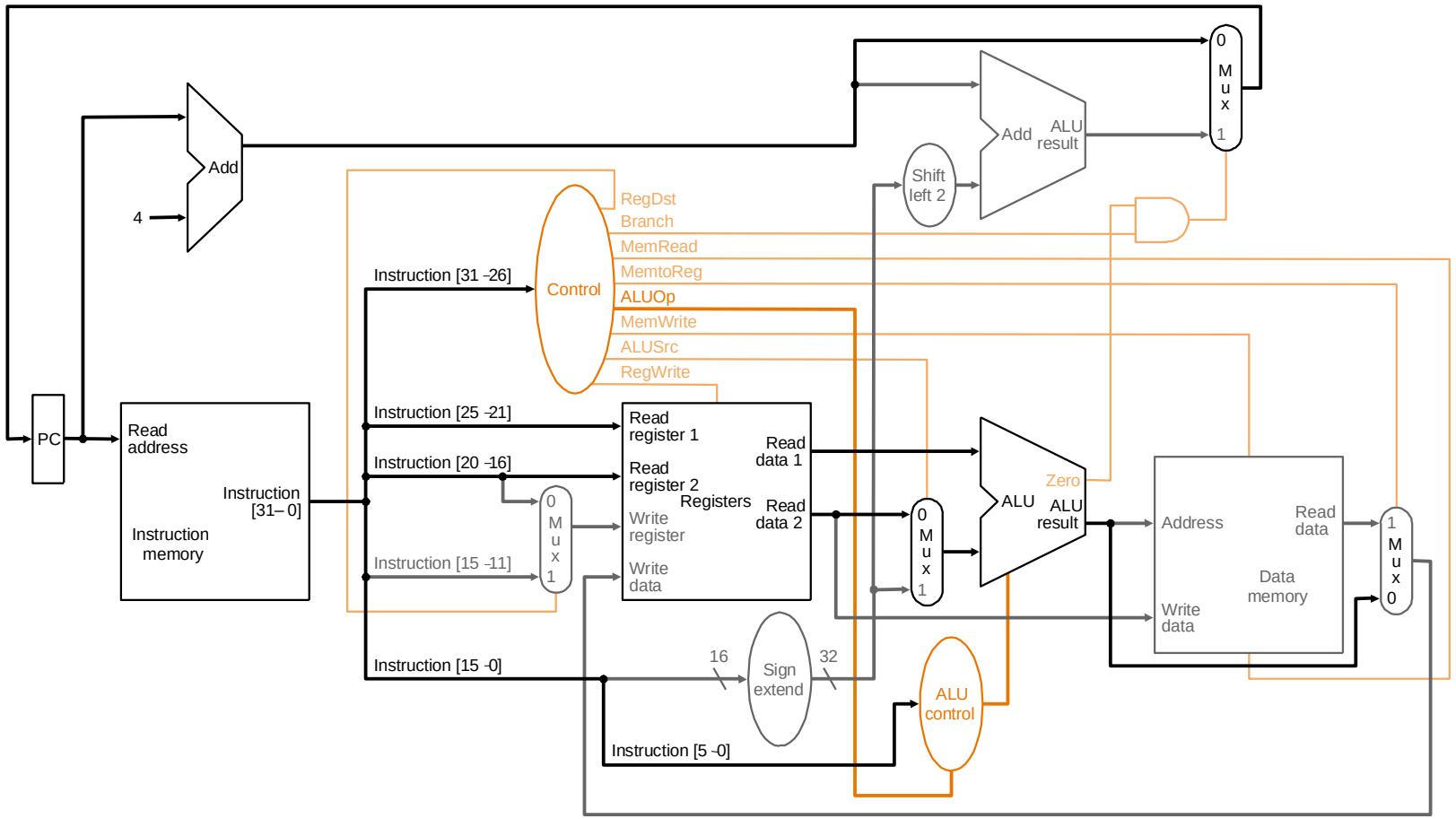
`add $t1, $t2, $t3`



Read two source registers from the register file

R-type Instruction: Step-3

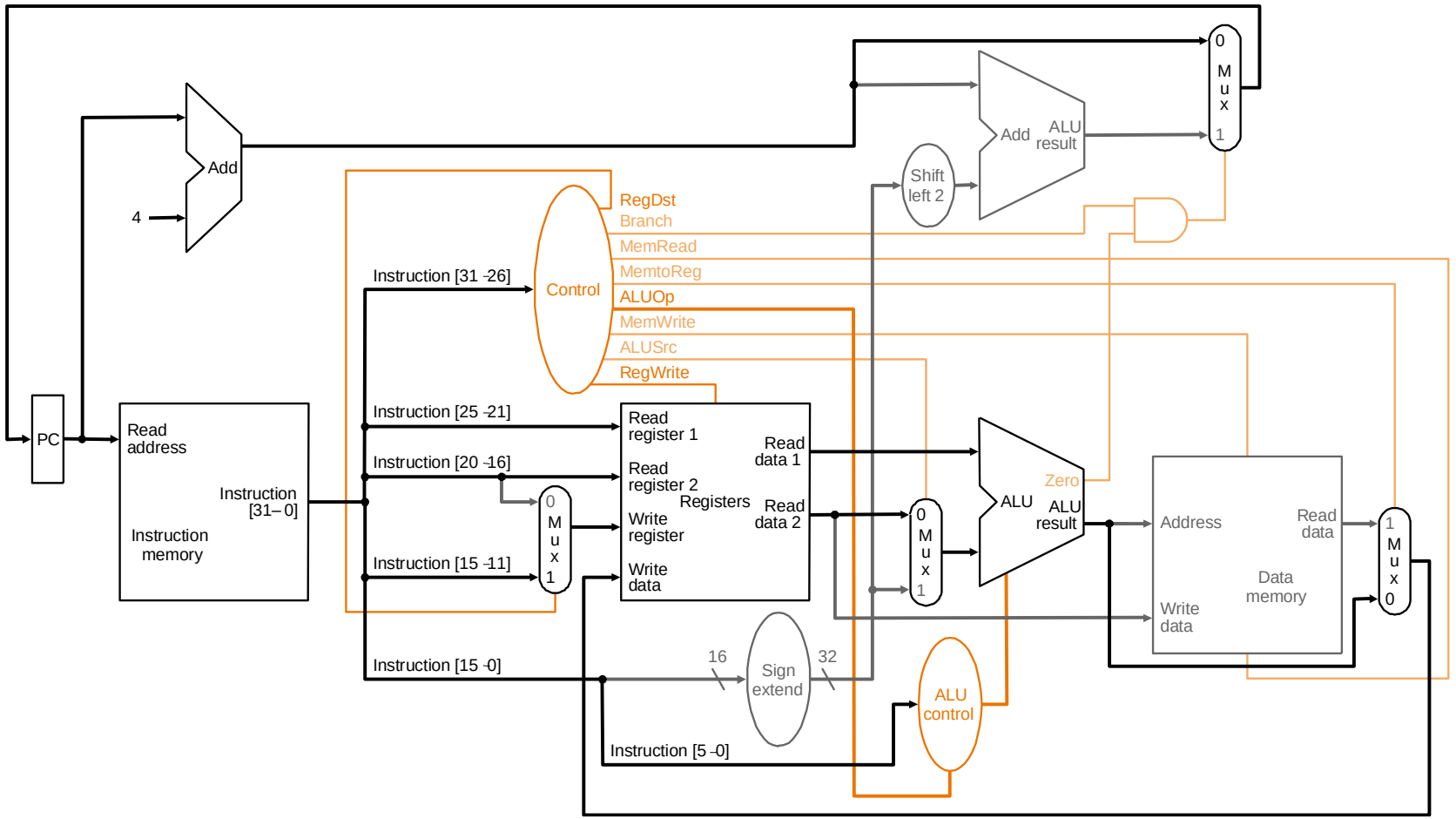
`add $t1, $t2, $t3`



ALU operates on the two register operands

R-type Instruction: Step-4

`add $t1, $t2, $t3`



Write result to register

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.

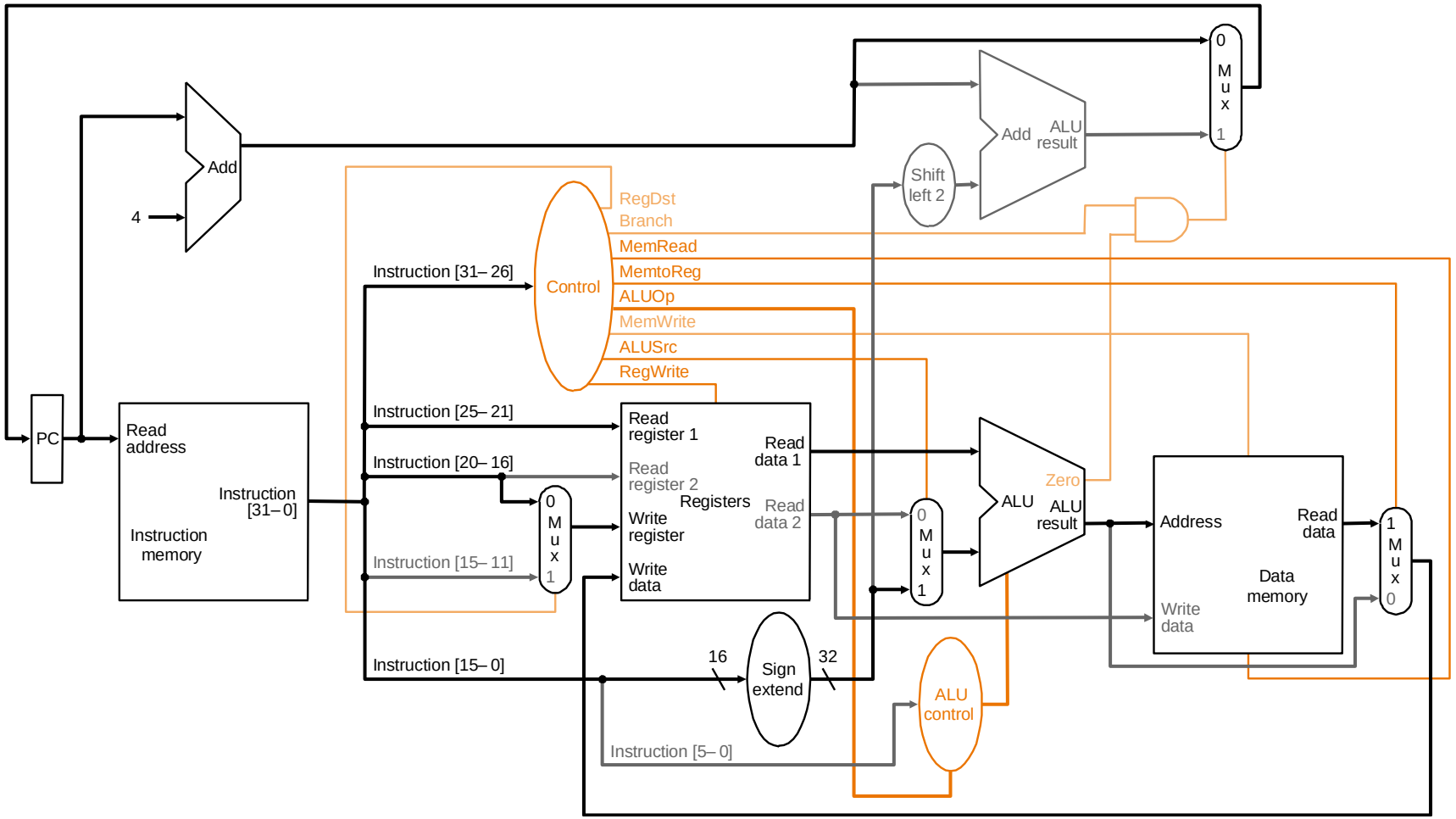
Load Instruction Steps

`lw $t1, offset($t2)`

1. Fetch instruction and increment PC
2. Read base register from the register file: the base register ($t2) is given by bits 25-21 of the instruction
3. ALU computes sum of value read from the register file and the sign-extended lower 16 bits (offset) of the instruction
4. The sum from the ALU is used as the address for the data memory
5. The data from the memory unit is written into the register file: the destination register ($t1) is given by bits 20-16 of the instruction

Load Instruction Steps

`lw $t1, offset($t2)`



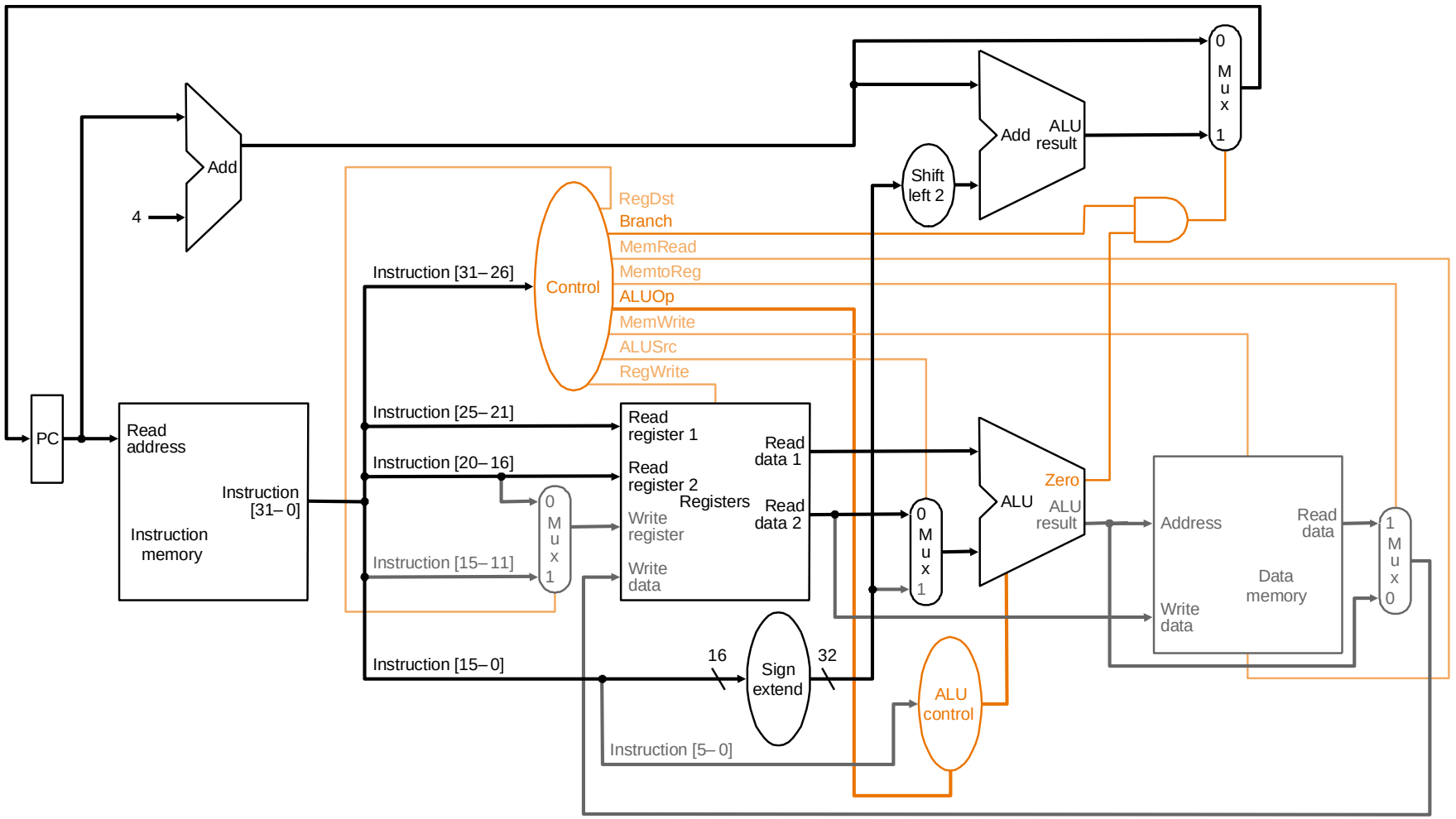
Branch Instruction Steps

`beq $t1, $t2, offset`

1. Fetch instruction and increment PC
2. Read two register ($t1 and $t2) from the register file
3. ALU performs a subtract on the data values from the register file; the value of PC+4 is added to the sign-extended lower 16 bits (offset) of the instruction shifted left by two to give the branch target address
4. The Zero result from the ALU is used to decide which adder result (from step 1 or 3) to store in the PC

Branch Instruction Steps

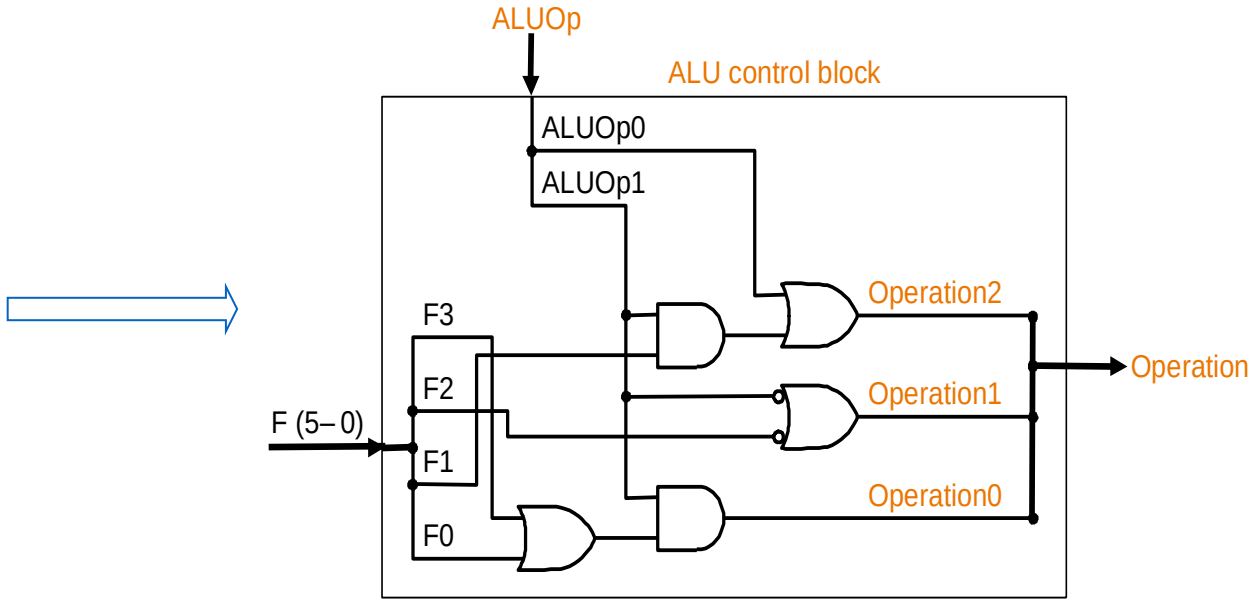
`beq $t1, $t2, offset`



Implementing the ALU Control Block

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits

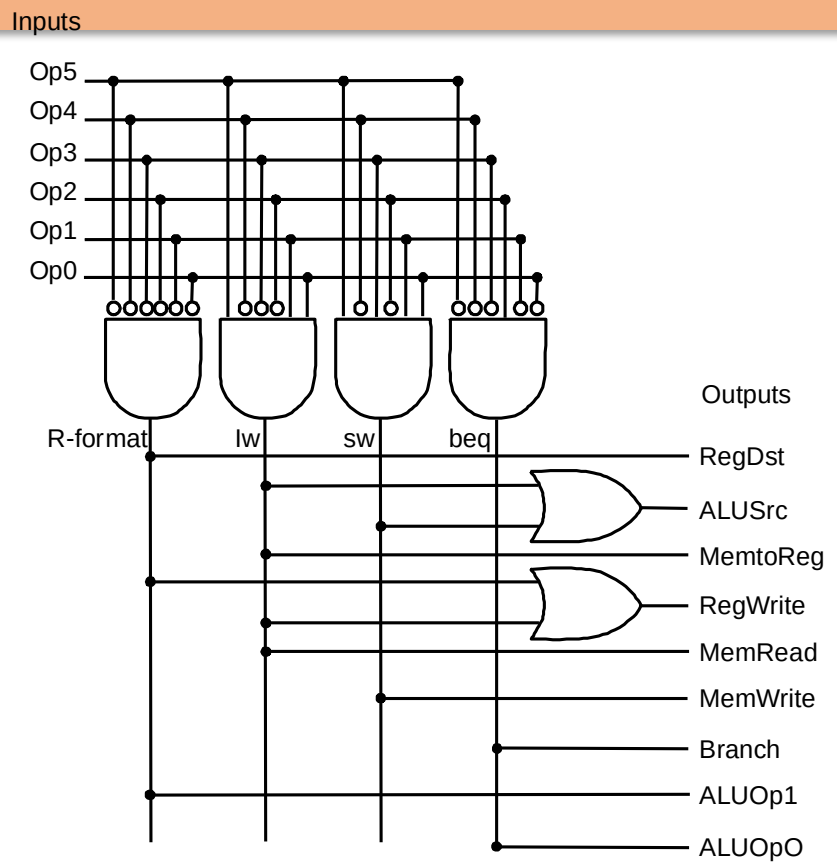


ALU control logic

Implementation: Main Control Block

	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOP2	0	0	0	1

Truth table for main control signals



Main control PLA (programmable logic array): principle underlying PLAs is that any logical expression can be written as a sum-of-products

Single-Cycle Design Problems

- Assuming fixed-period clock every instruction datapath 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*
    - consider if we have more complicated instructions like floating point!
  - resources used more than once in the same cycle need to be duplicated
    - *waste of hardware and chip area*

Example Problem

- Consider a machine with an additional floating point unit. Assume functional unit delays as follows
  - *memory*: 2 ns., *ALU* and *adders*: 2 ns., *FPU add*: 8 ns., *FPU multiply*: 16 ns., *register file access* (read or write): 1 ns.
  - *multiplexors*, *control unit*, *PC accesses*, *sign extension*, *wires*: no delay
- Assume instruction mix as follows
  - all loads take same time and comprise 31%
  - all stores take same time and comprise 21%
  - R-format instructions comprise 27%
  - branches comprise 5%
  - jumps comprise 2%
  - FP adds and subtracts take the same time and totally comprise 7%
  - FP multiplys and divides take the same time and totally comprise 7%
- *Compare the performance of (a) a single-cycle implementation using a fixed-period clock with (b) one using a variable-period clock where each instruction executes in one clock cycle that is only as long as it needs to be (not really practical but pretend it's possible!)*

Solution

Instruction class	Instr. mem.	Register read	ALU oper.	Data mem.	Register write	FPU add/ sub	FPU mul/ div	Total time ns.
Load word	2	1	2	2	1			8
Store word	2	1	2	2				7
R-format	2	1	2	0	1			6
Branch	2	1	2					5
Jump	2							2
FP mul/div	2	1			1		16	20
FP add/sub	2	1			1	8		12

- Clock period for fixed-period clock = longest instruction time = 20 ns.
- Average clock period for variable-period clock = 8 × 31% +

  7 × 21% + 6 × 27% + 5 × 5% + 2 × 2% + 20 × 7% + 12 × 7%

  = 7.0 ns

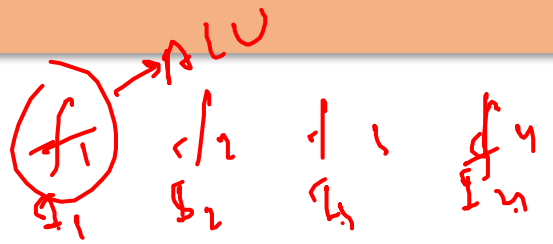
not practical

- Therefore, $\text{performance}_{\text{var-period}} / \text{performance}_{\text{fixed-period}} = 20/7 = 2.9$

Fixing the problem with single-cycle designs

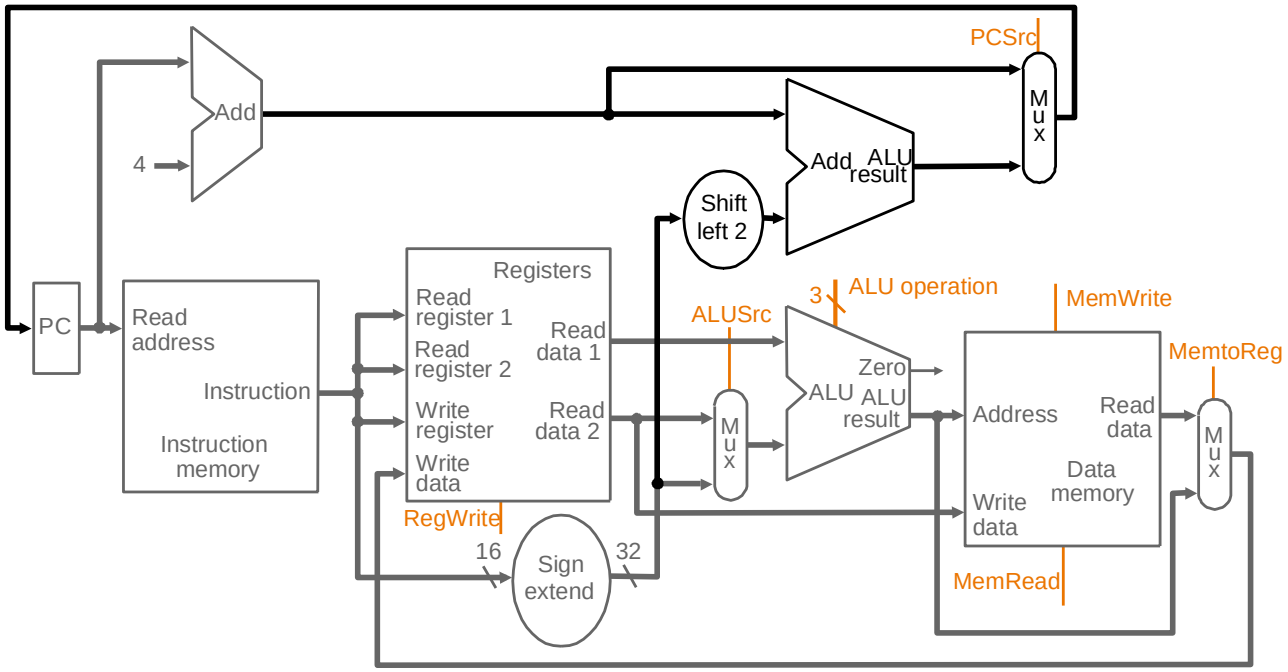
- One solution: a variable-period clock with different cycle times for each instruction class
  - *unfeasible*, as implementing a variable-speed clock is technically difficult
- Another solution:
  - use a smaller cycle time…
  - …have different instructions take different numbers of cycles
  - by breaking instructions into steps and fitting each step into one cycle
  - *feasible: multicyle approach*!

Multicycle Approach

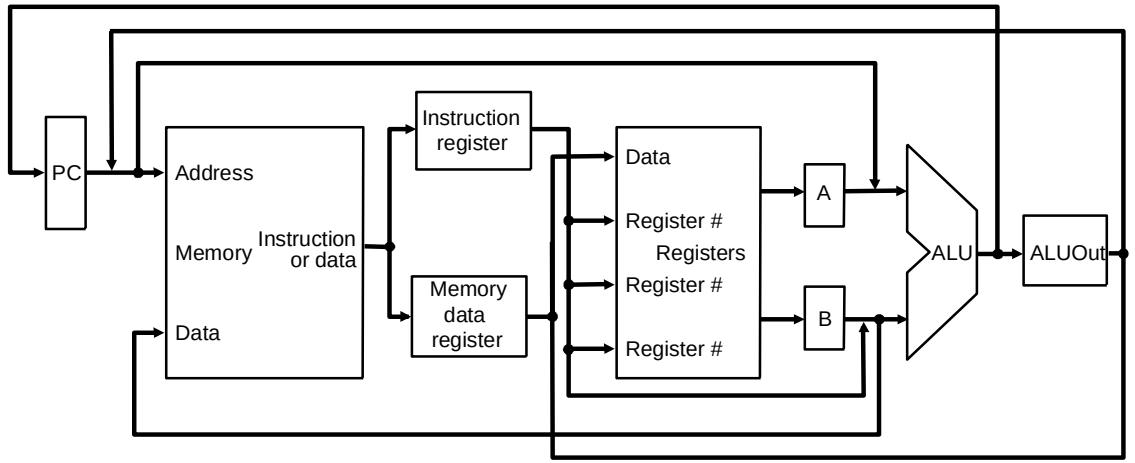


- Break up the instructions into *steps*
  - each step takes one clock cycle
  - balance the amount of work to be done in each step/cycle so that they are about equal
  - restrict each cycle to use at most once each major functional unit so that such units do not have to be replicated
  - functional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

Multicycle Approach



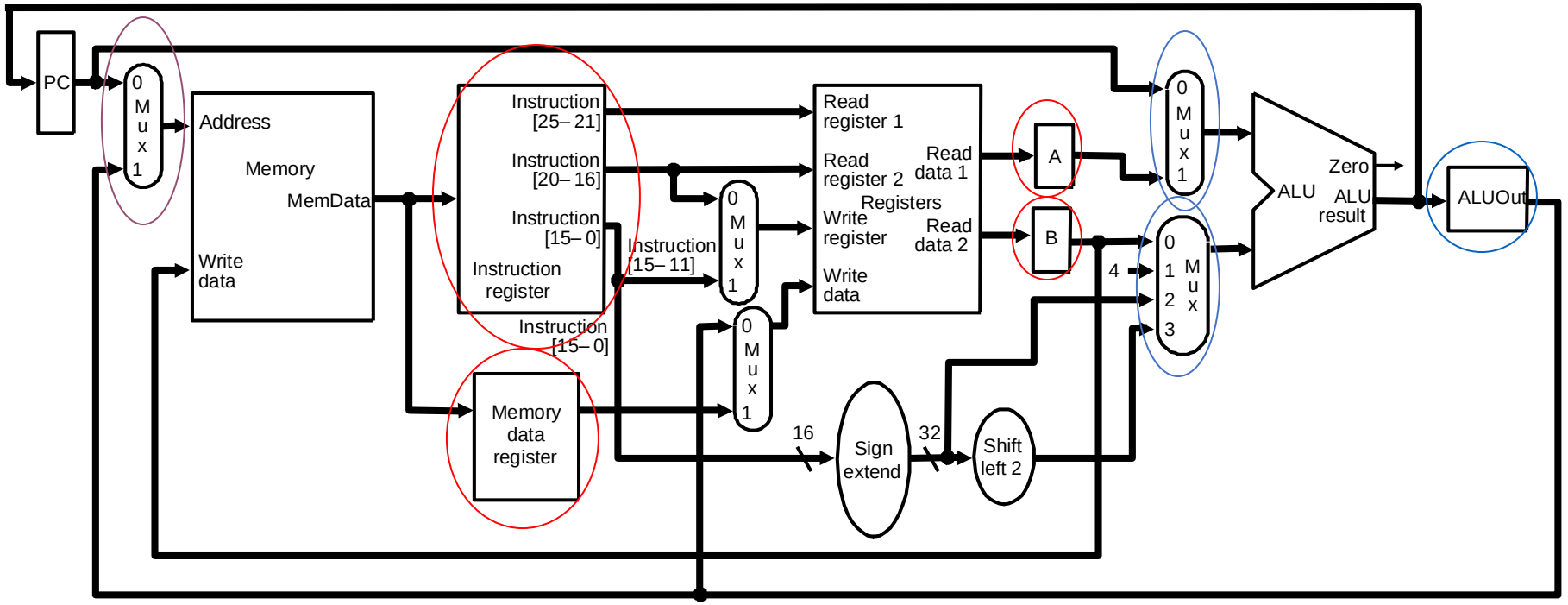
Single-cycle datapath



Multicycle datapath (high-level view)

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

Multicycle Approach



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

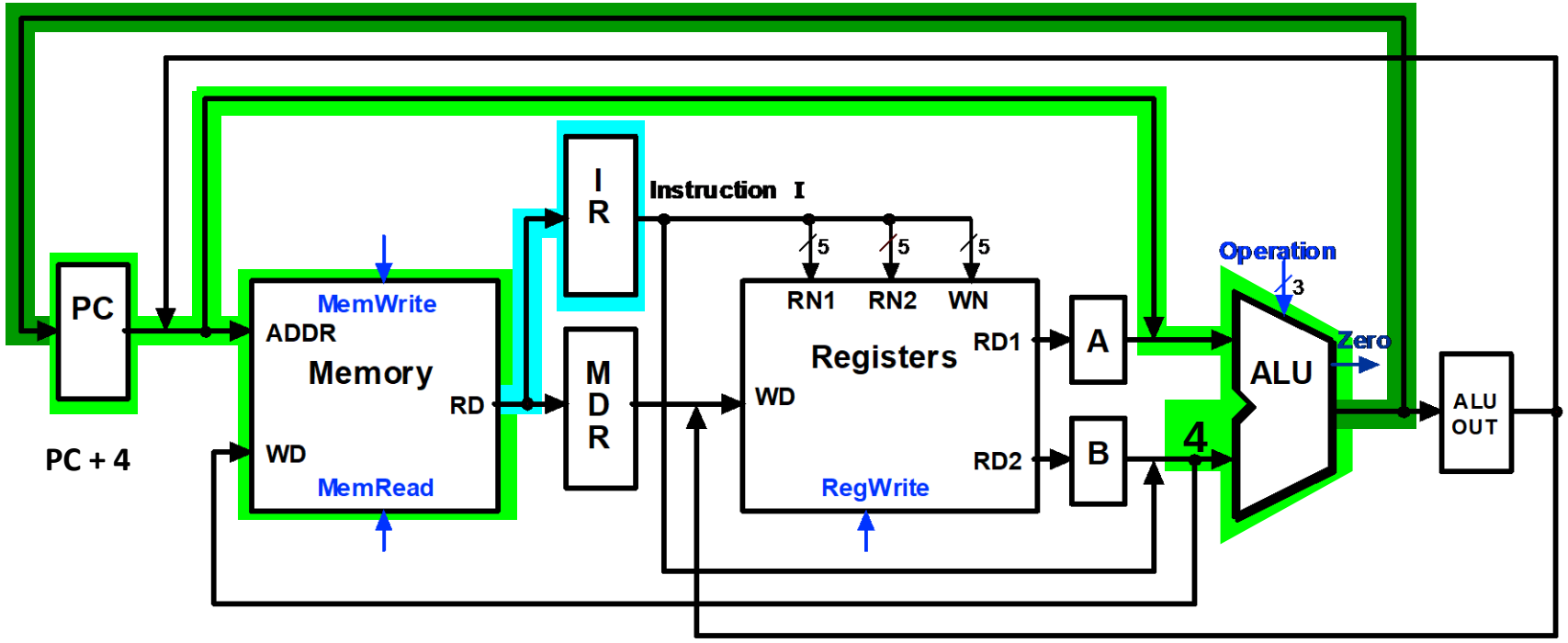
- The reason this is not done is that, to keep steps balanced in length, the design restriction is to allow each step to contain *at most* one ALU operation, or one register access, or one memory access.

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		

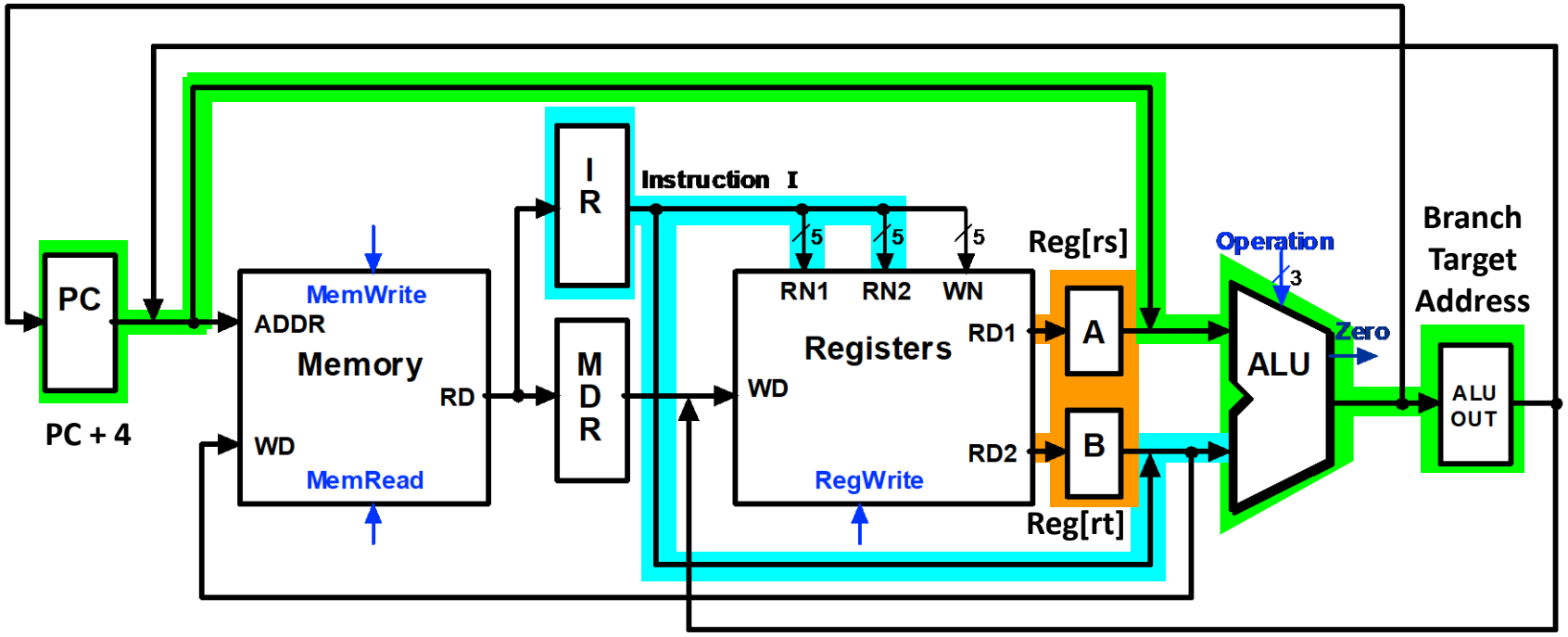
Multicycle Execution Step (1): Instruction Fetch

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



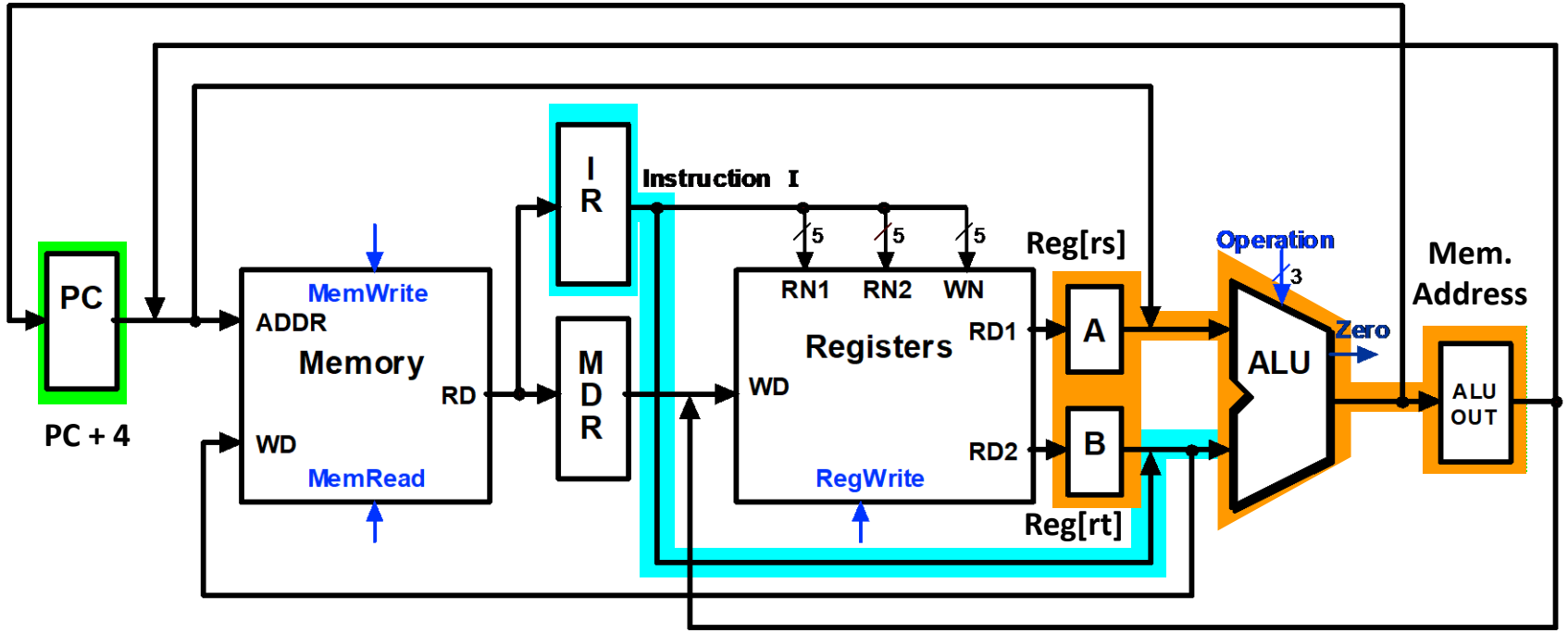
Multicycle Execution Step (2): Instruction Decode & Register Fetch

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



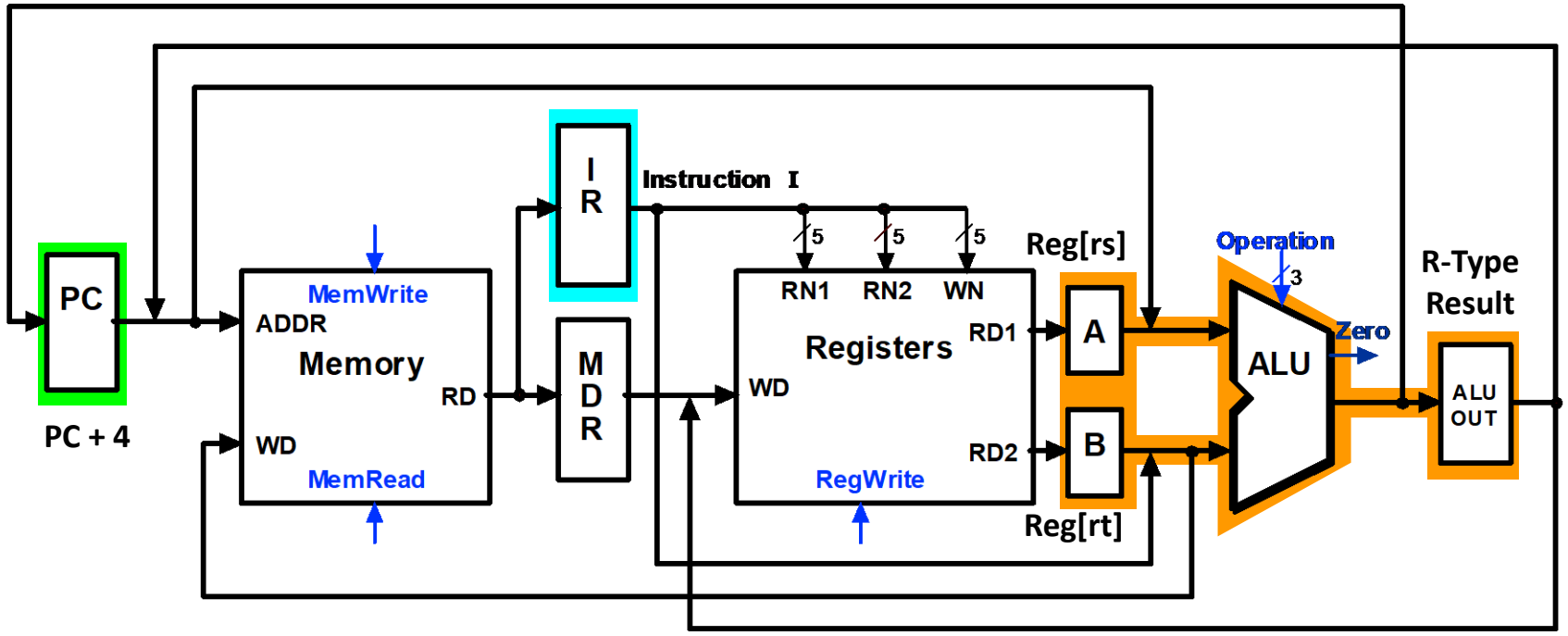
Multicycle Execution Step (3): Memory Reference Instructions

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



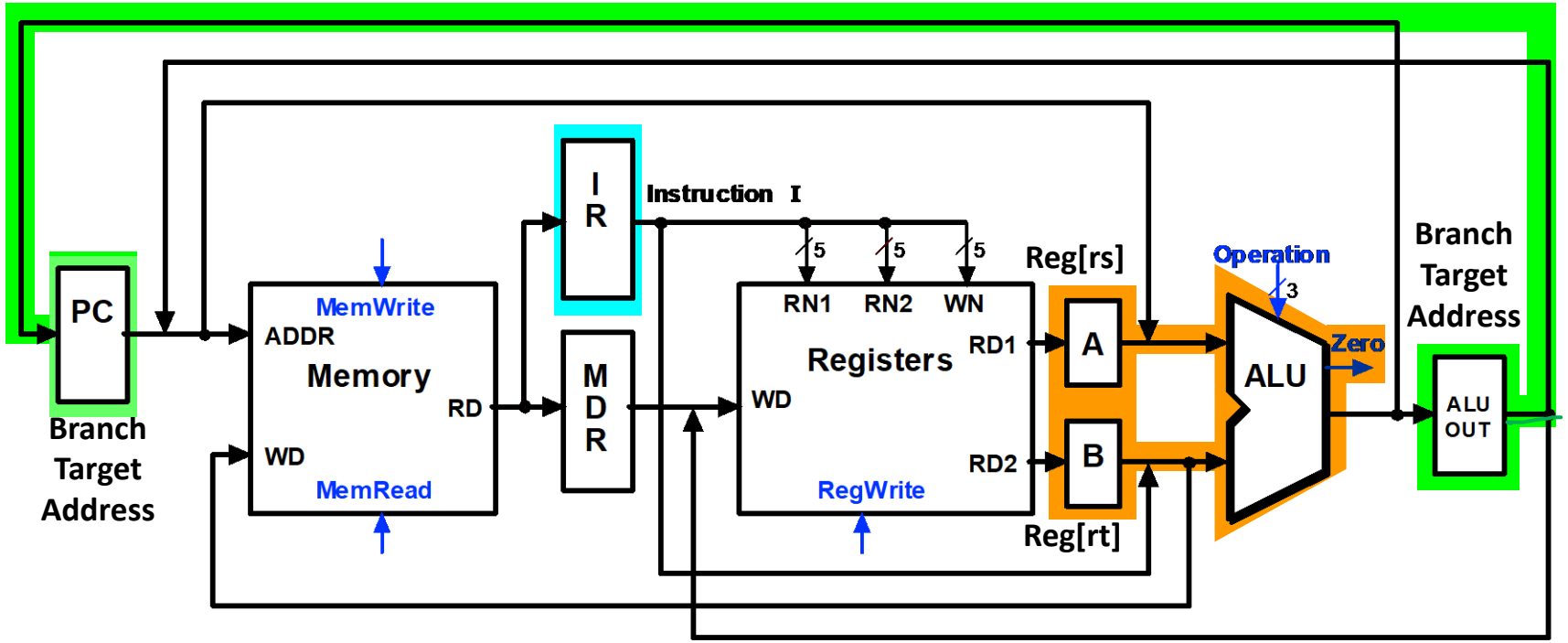
Multicycle Execution Step (3): ALU Instruction (R-Type)

```
ALUOut = A op B
```



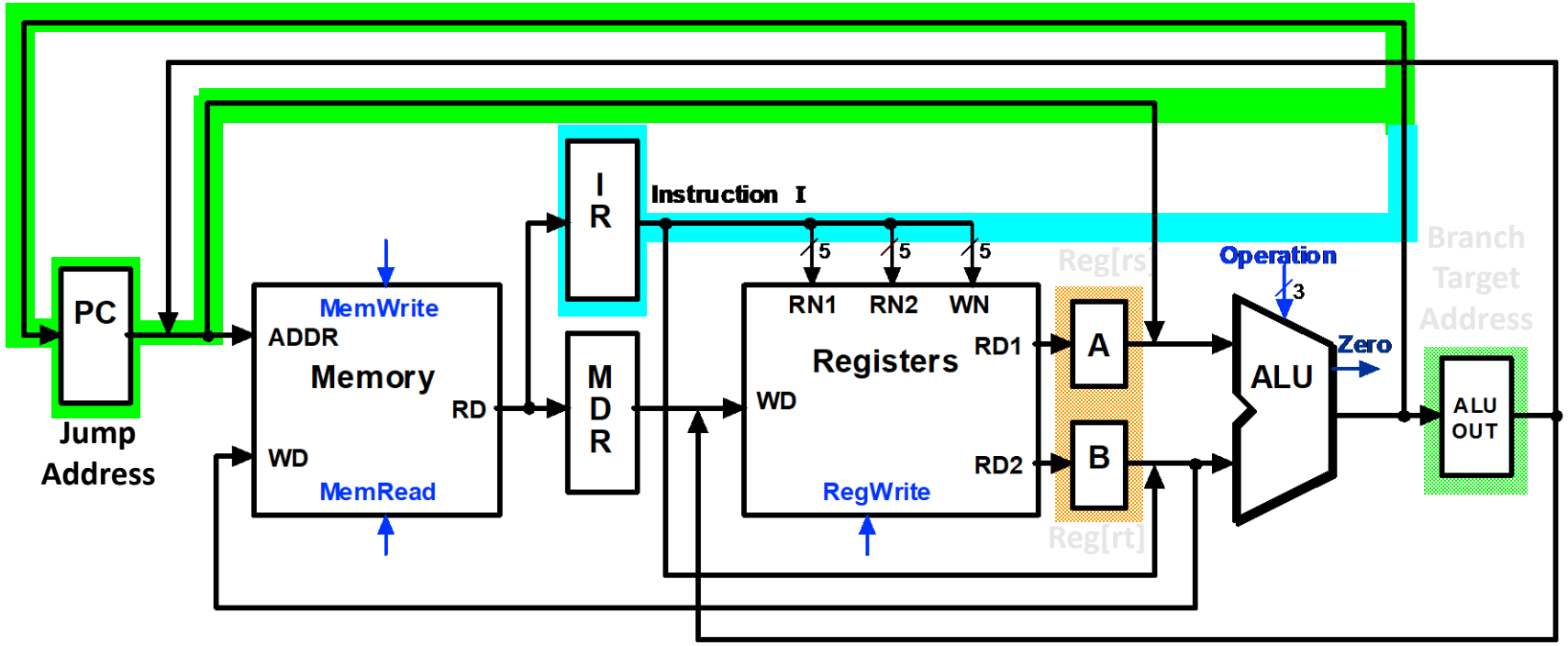
Multicycle Execution Step (3): Branch Instructions

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



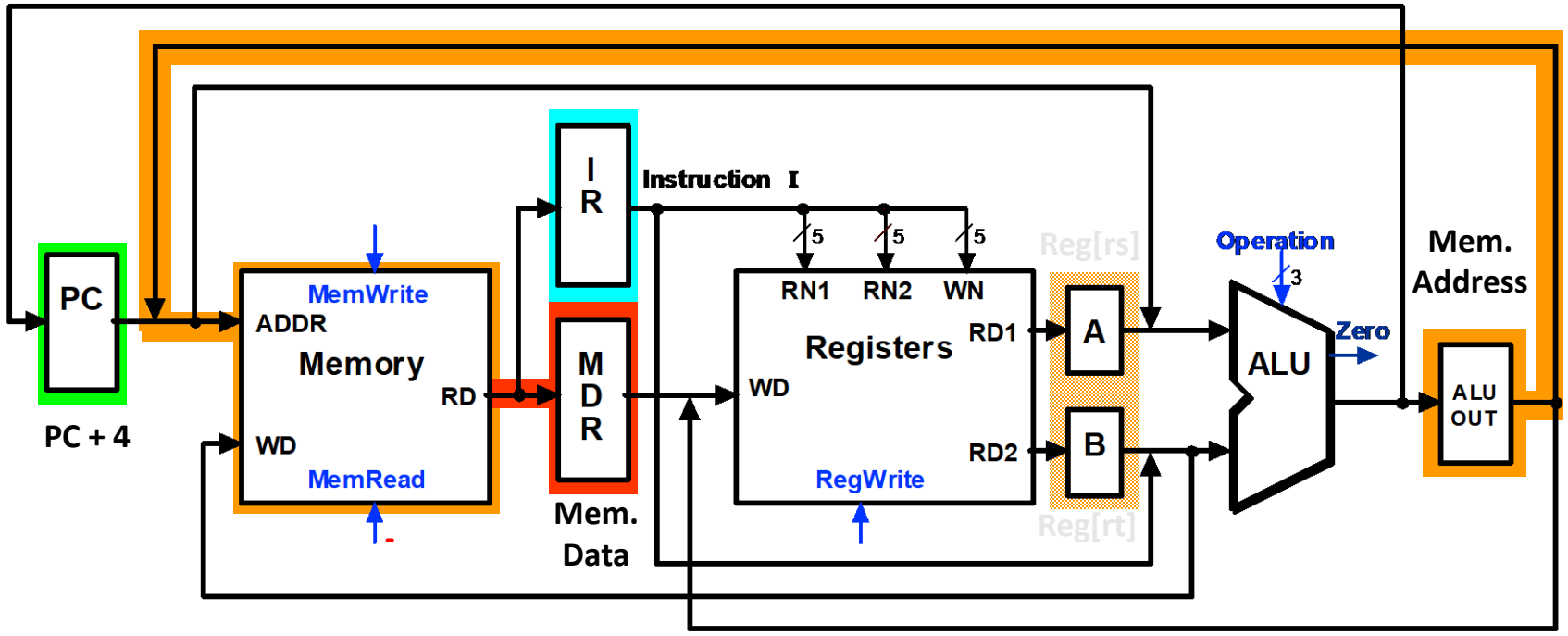
Multicycle Execution Step (3): Jump Instruction

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



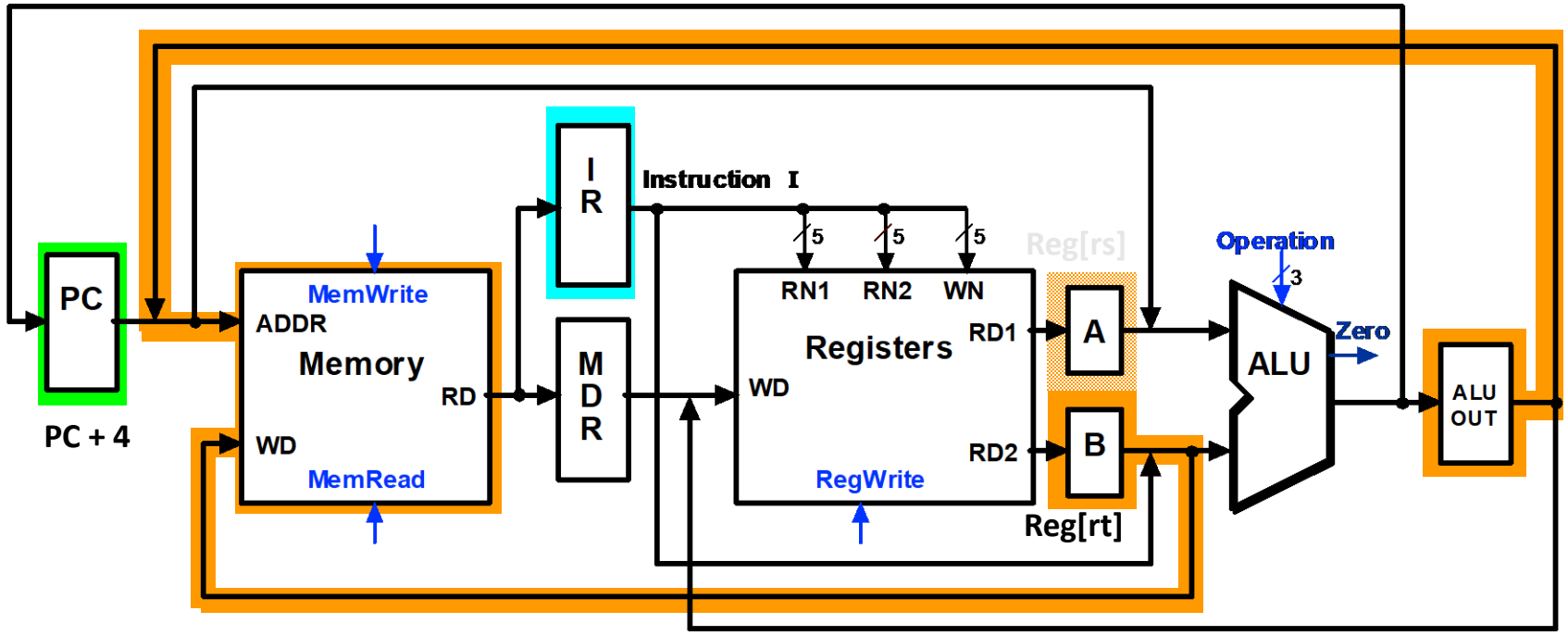
Multicycle Execution Step (4): Memory Access - Read (lw)

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



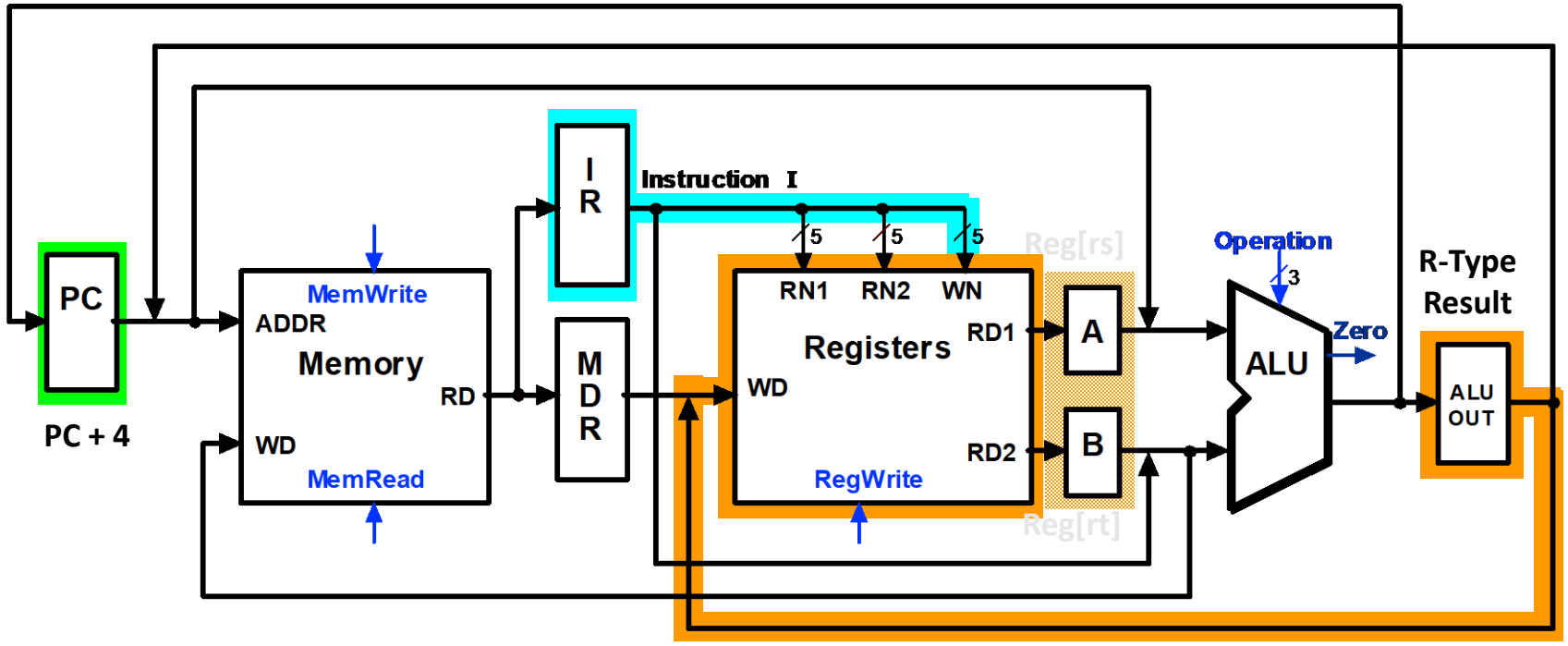
Multicycle Execution Step (4): Memory Access - Write (`sw`)

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



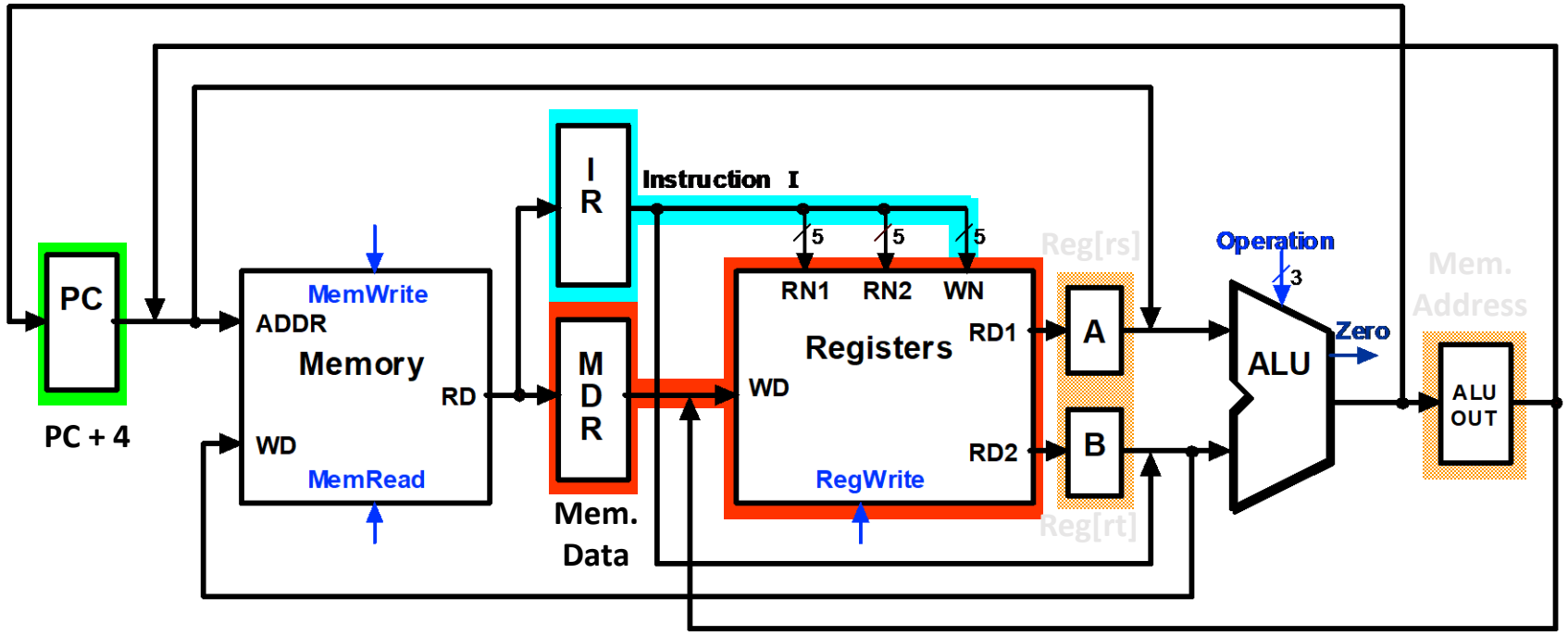
Multicycle Execution Step (4): ALU Instruction (R-Type)

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

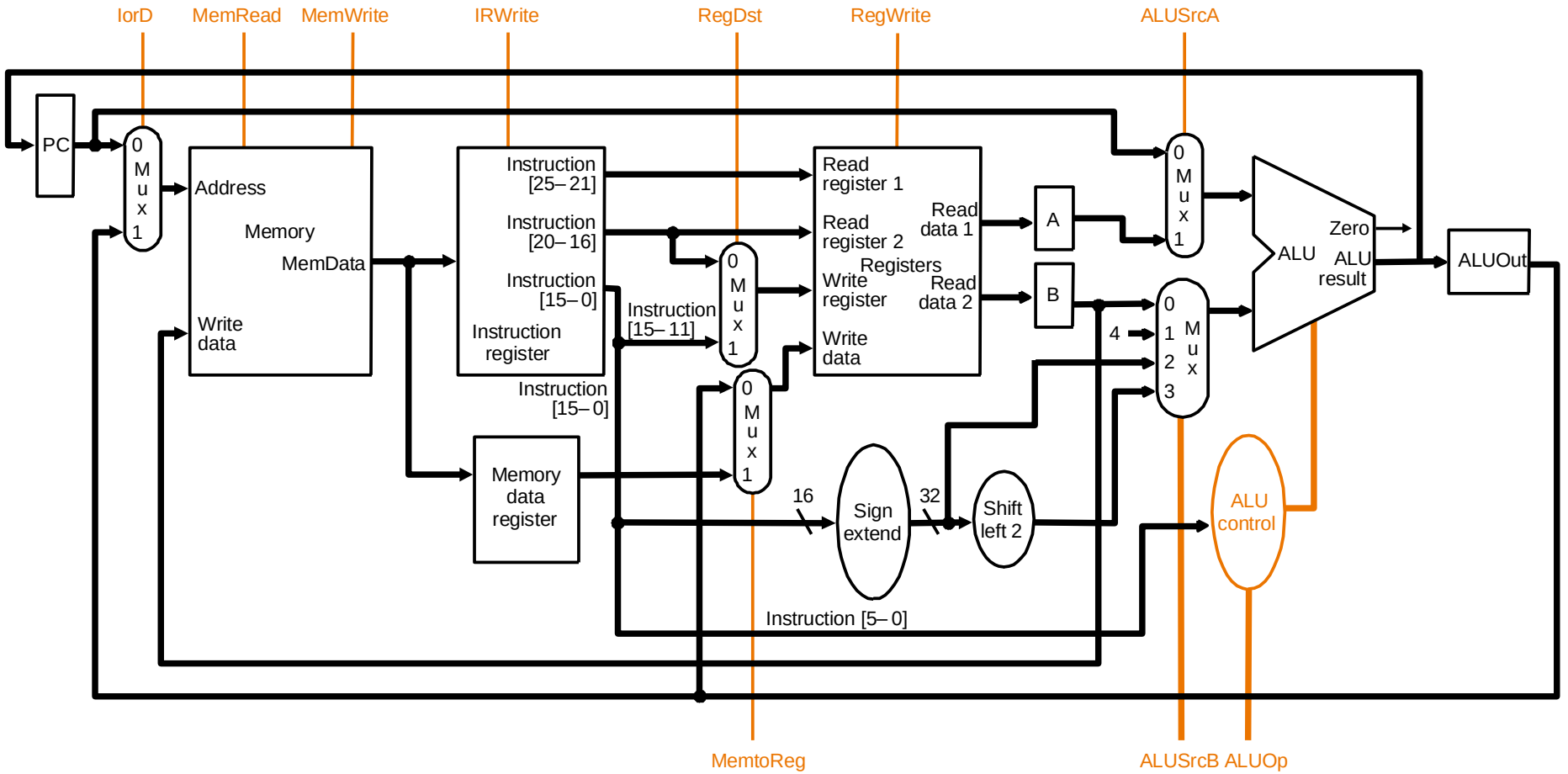


Multicycle Execution Step (5): Memory Read Completion (`lw`)

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

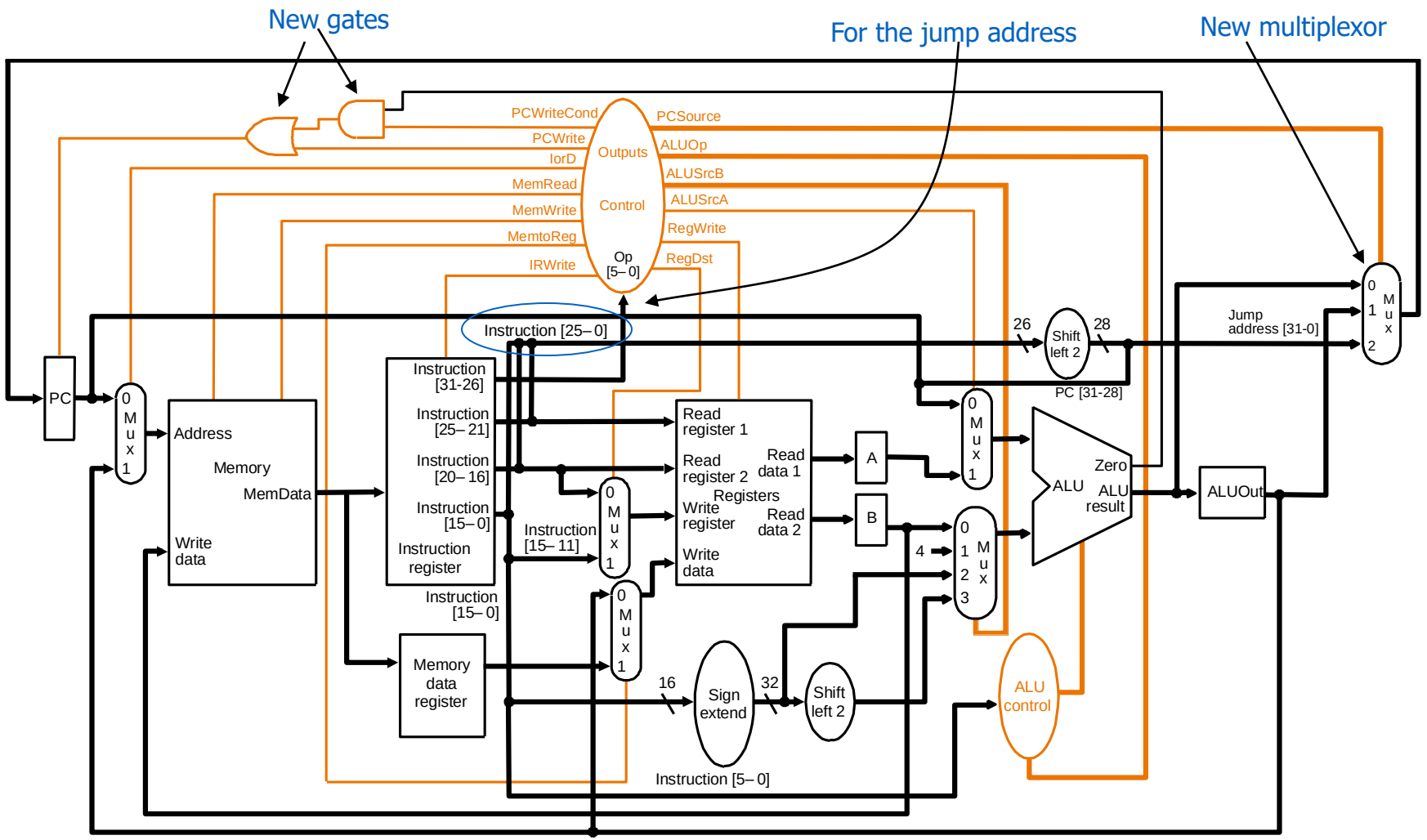


Multicycle Datapath with Control I



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

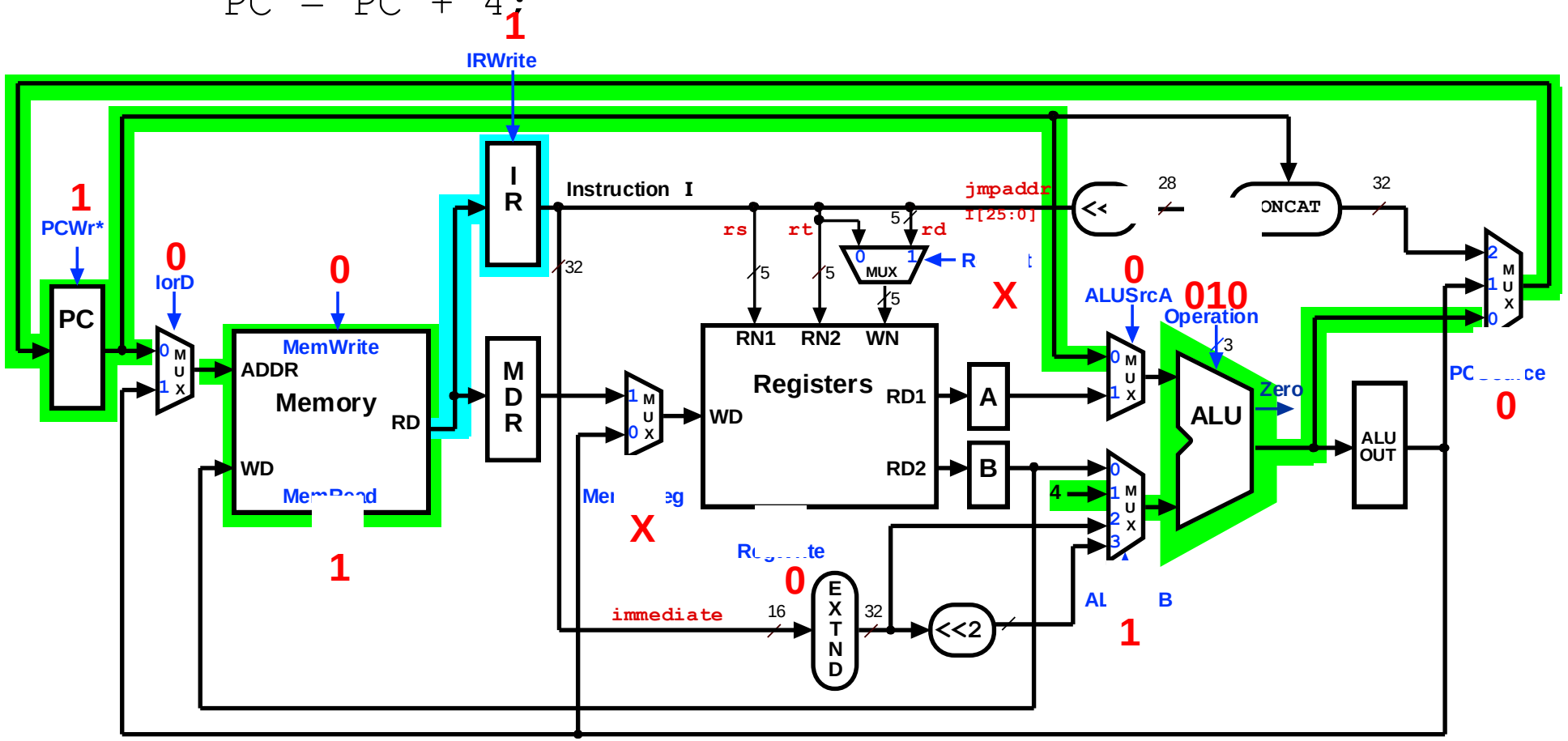
Multicycle Datapath with Control II



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

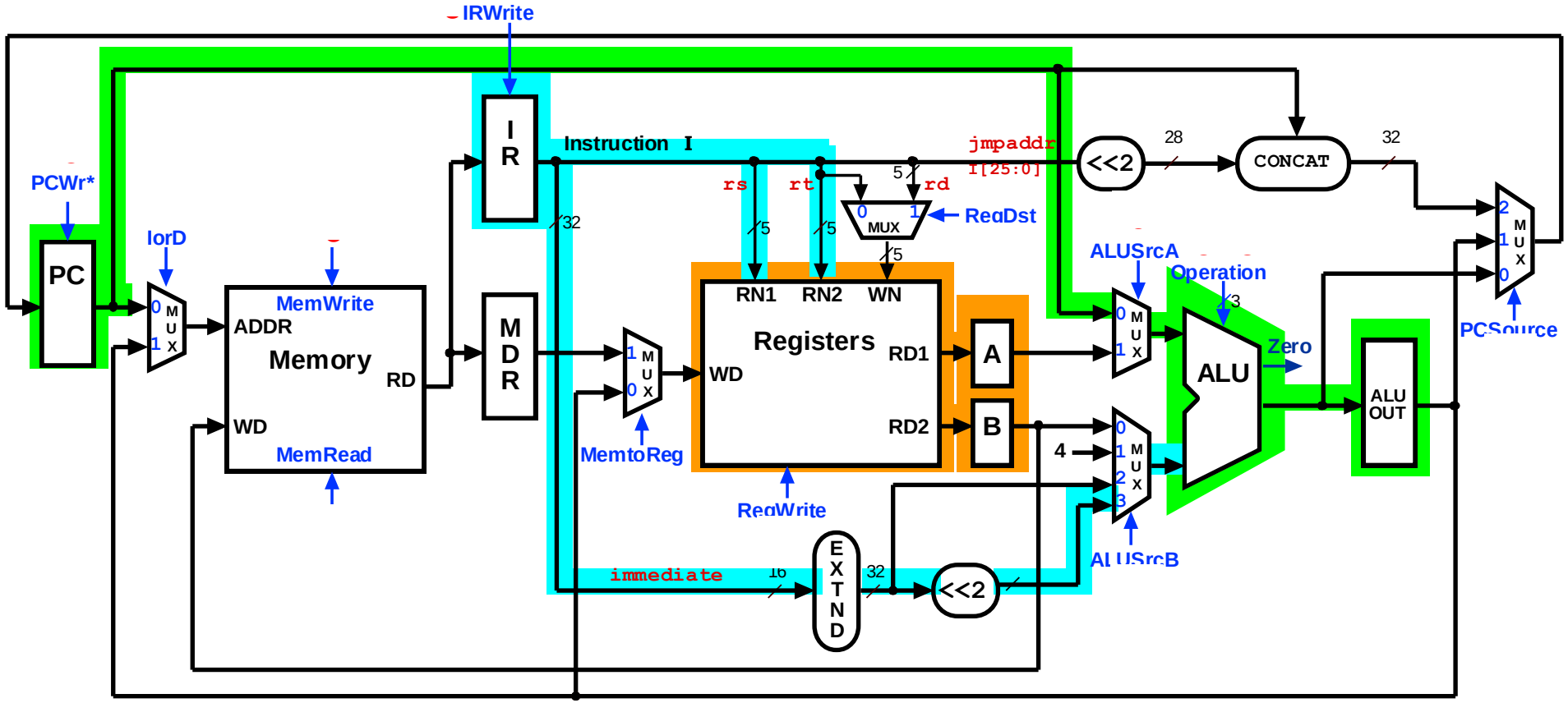
Multicycle Control Step (1): Fetch

```
IR = Memor   PC];
PC = PC + 4;
```



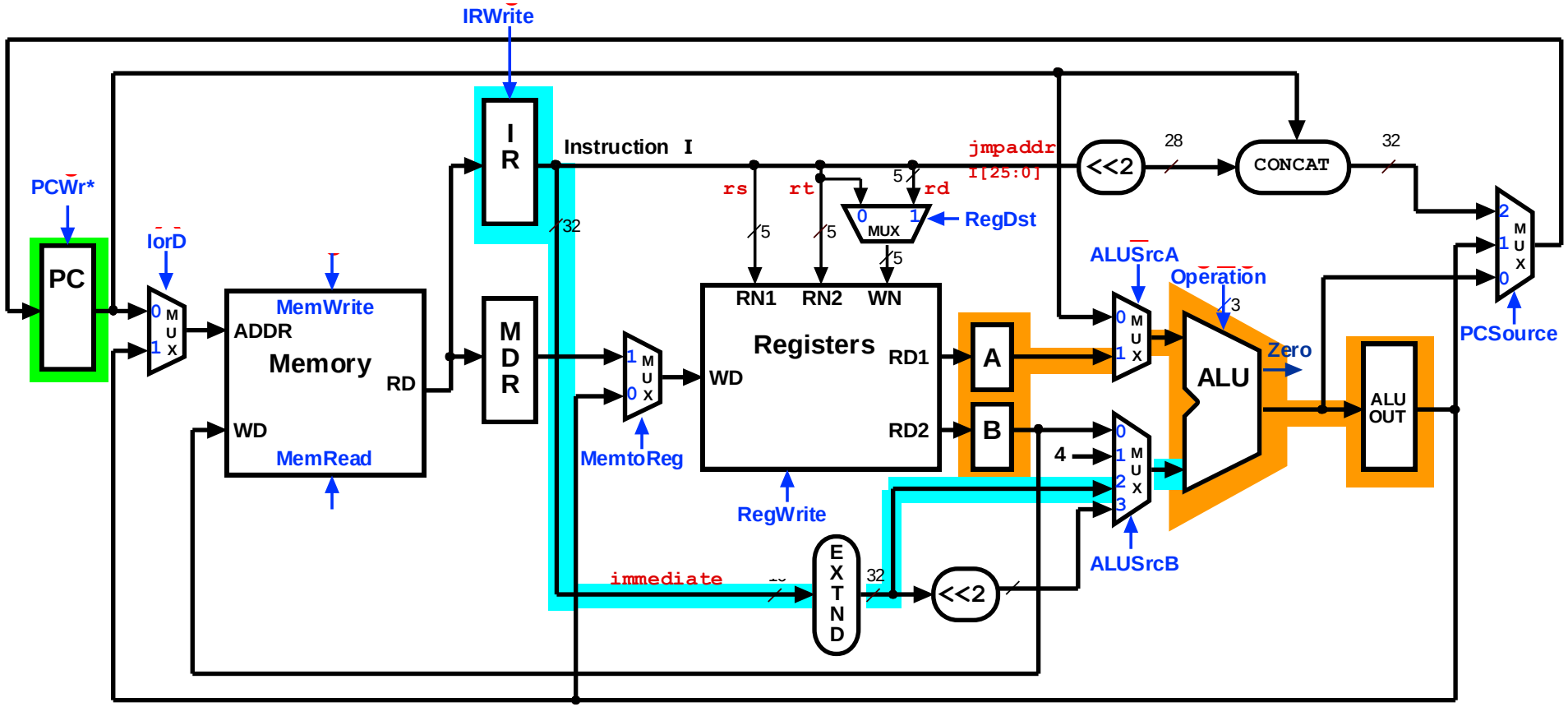
Multicycle Control Step (2): Instruction Decode & Register Fetch

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



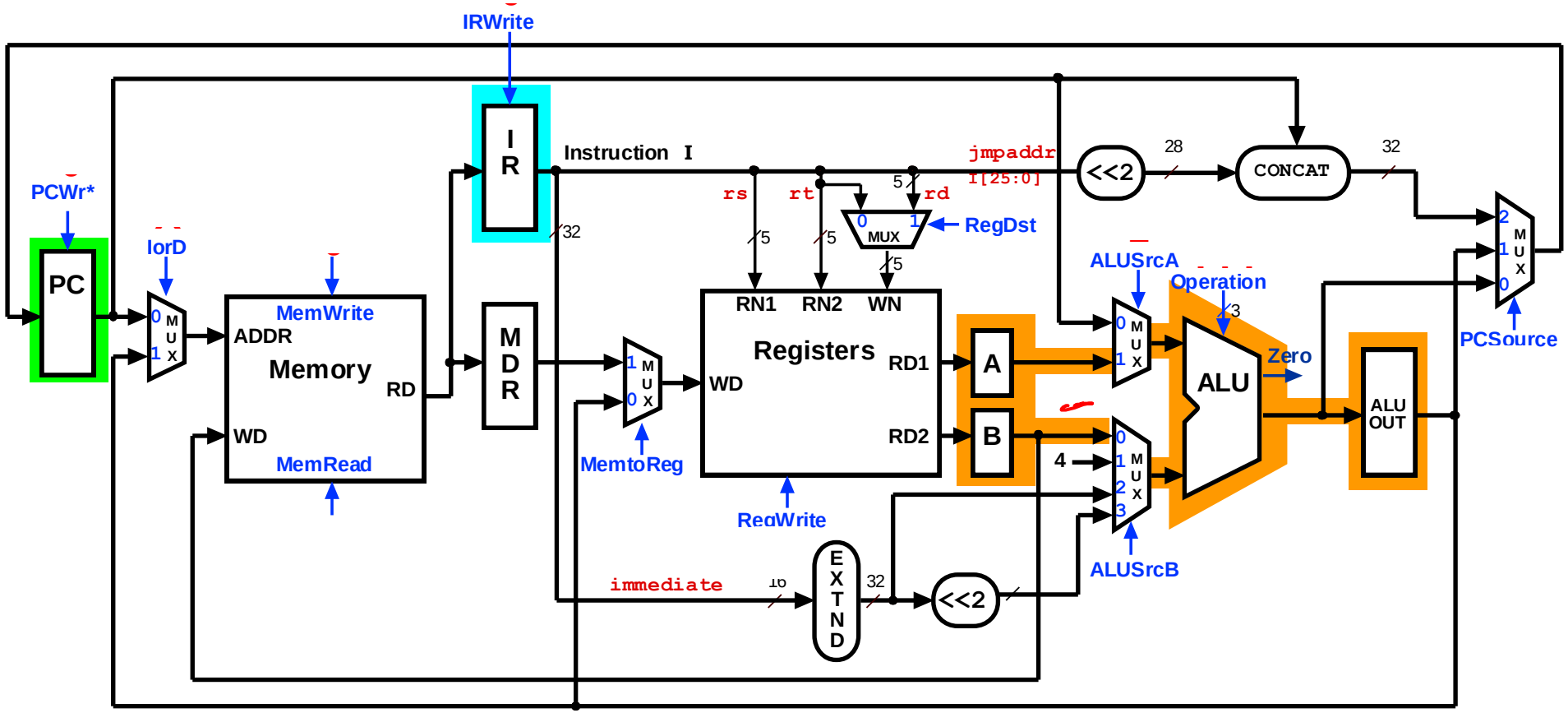
Multicycle Control Step (3): Memory Reference Instructions

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



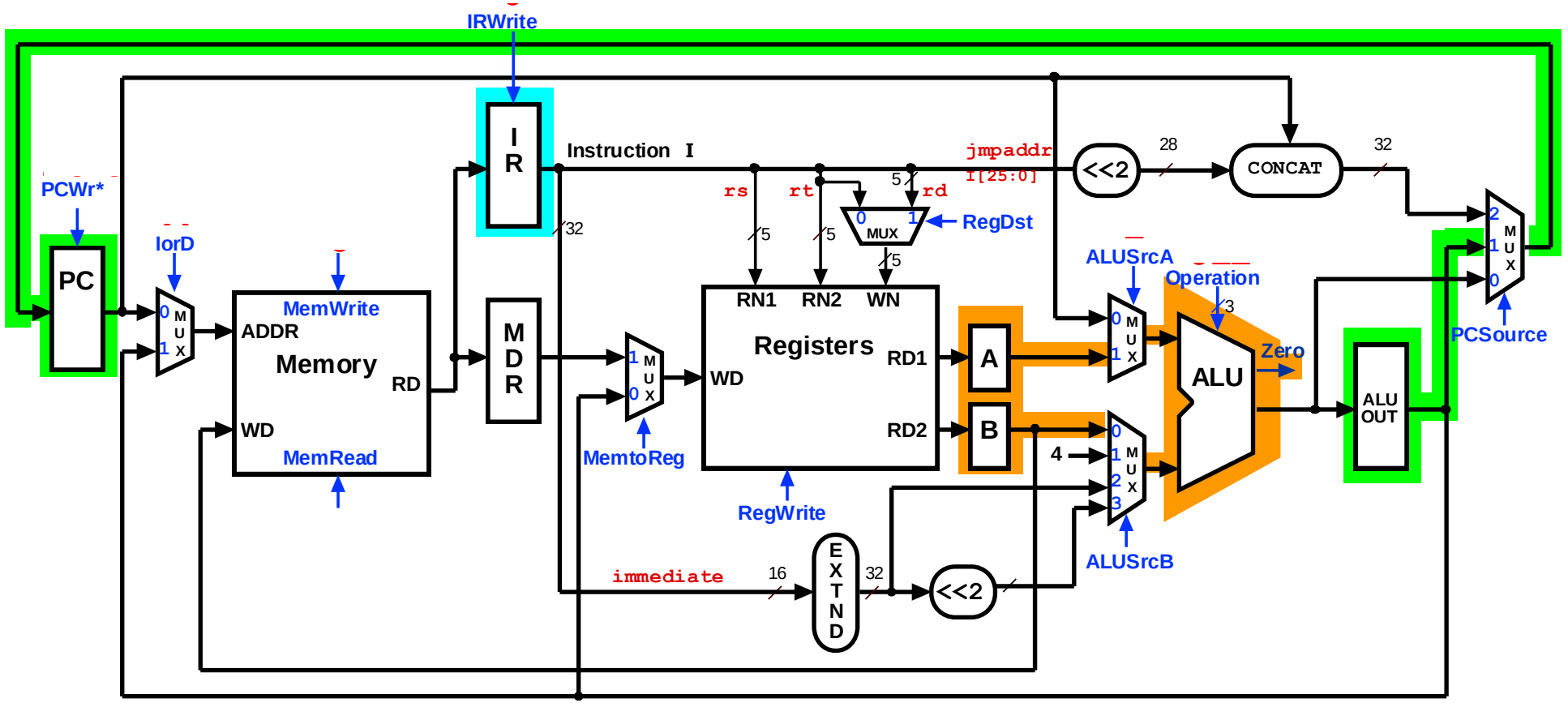
Multicycle Control Step (3): ALU Instruction (R-Type)

```
ALUOut = A op B;
```



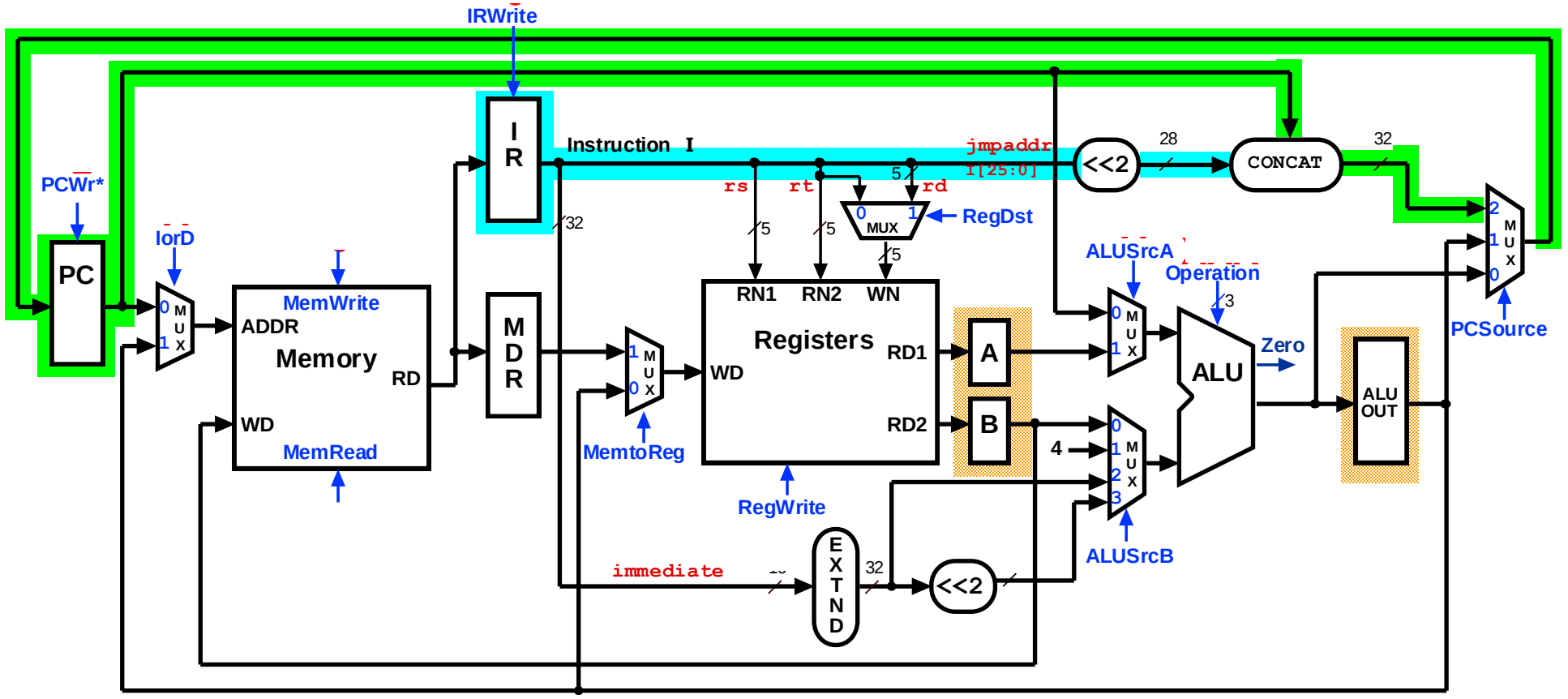
Multicycle Control Step (3): Branch Instructions

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



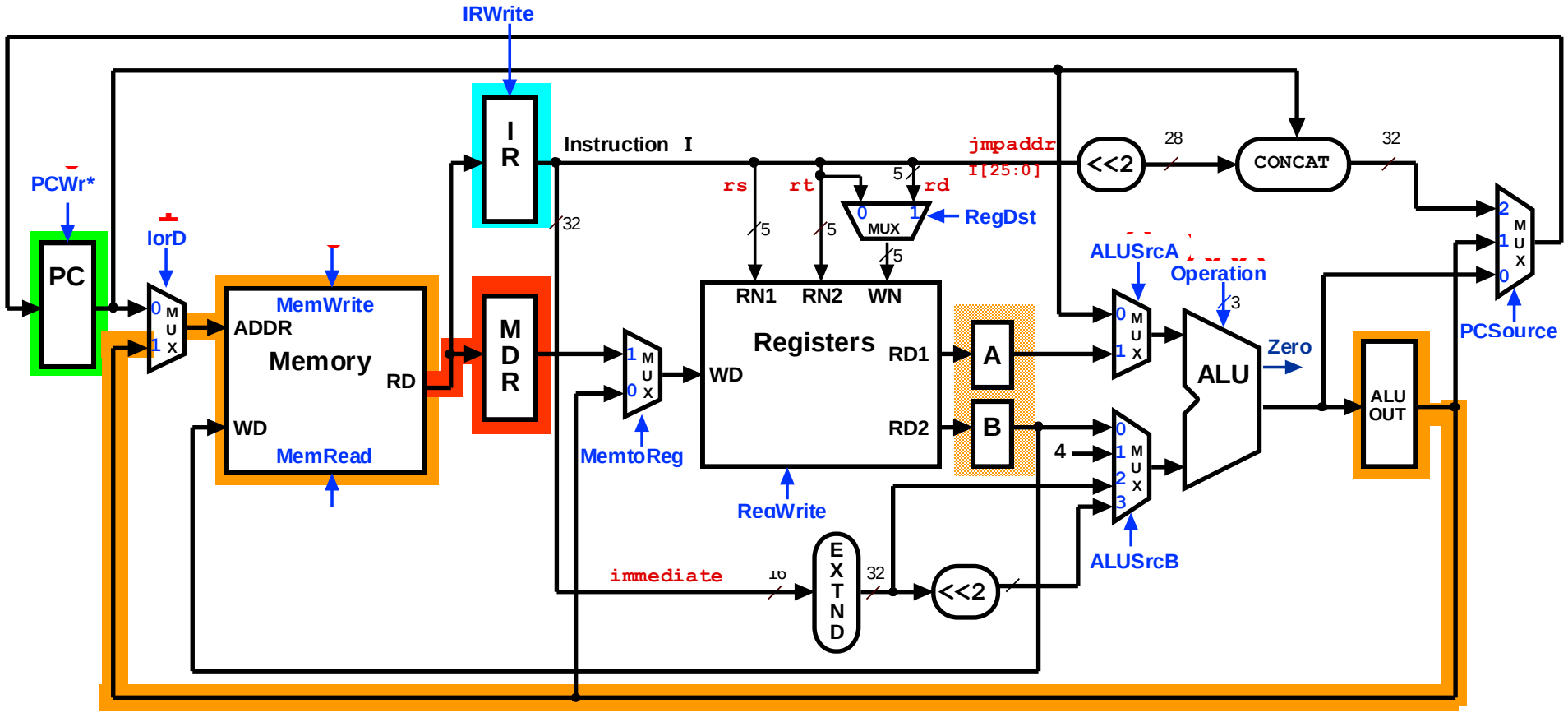
Multicycle Execution Step (3): Jump Instruction

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



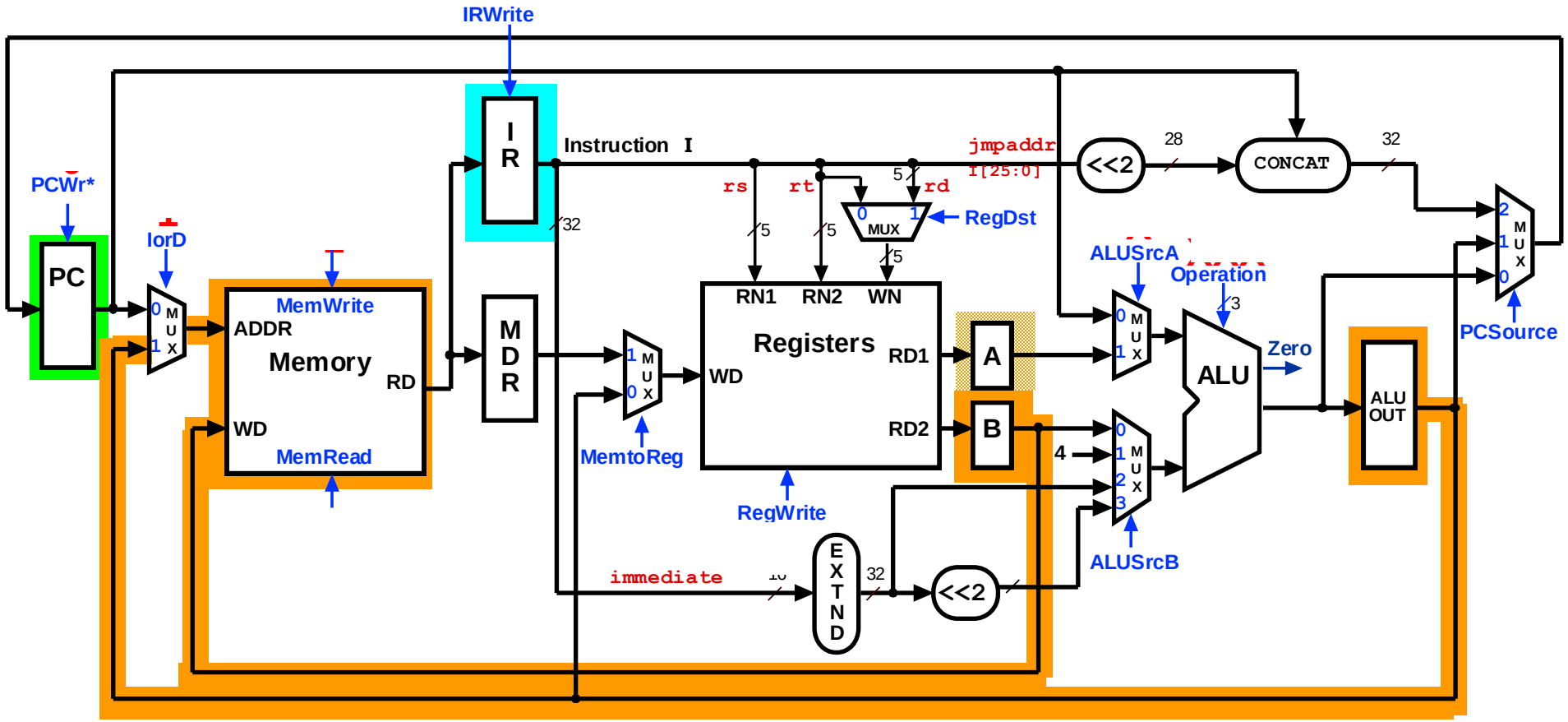
Multicycle Control Step (4): Memory Access - Read (`lw`)

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



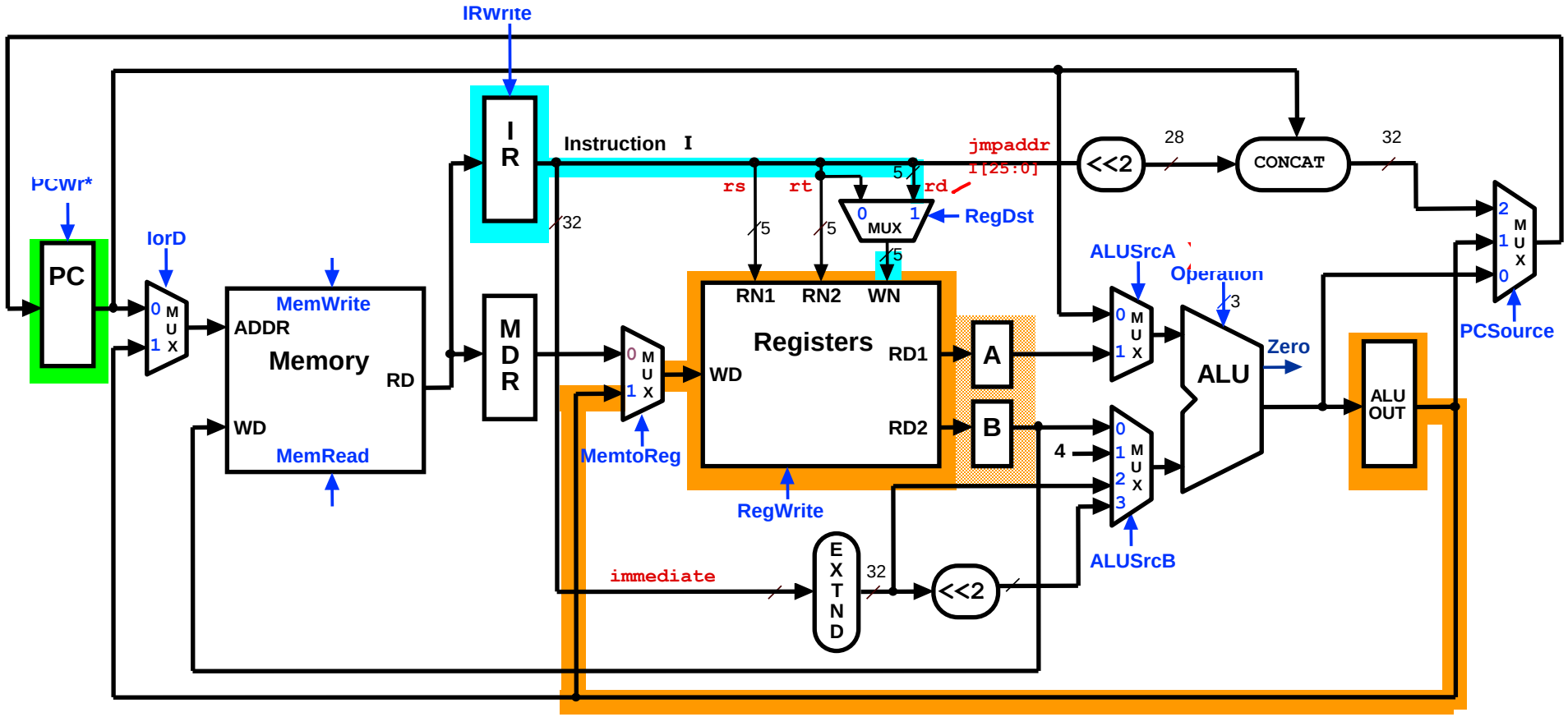
Multicycle Execution Steps (4)
Memory Access - Write (sw)

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



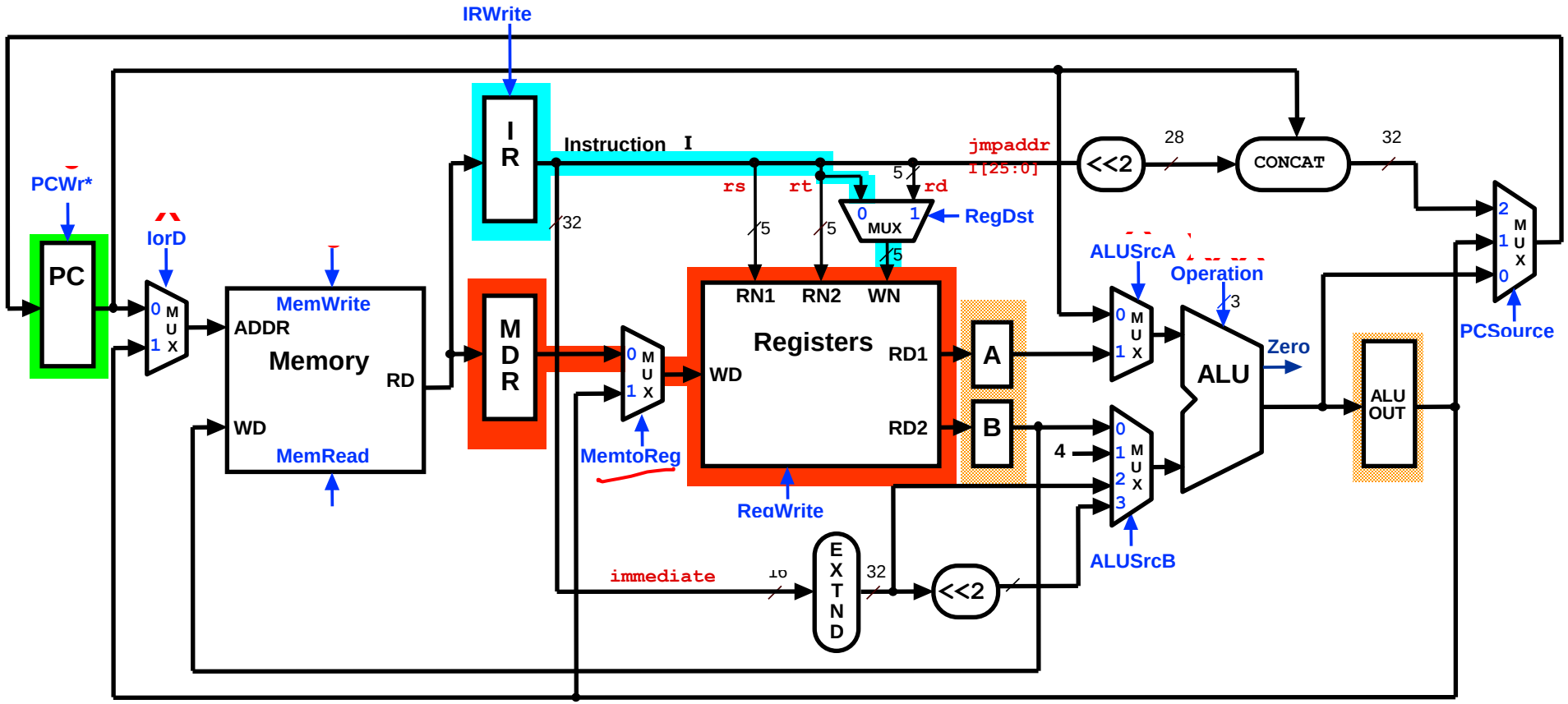
Multicycle Control Step (4): ALU Instruction (R-Type)

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



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

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



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?*

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

Clock time-line

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*