

MIPS data path and Control

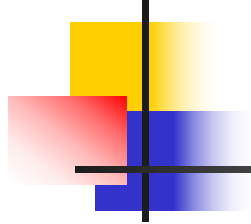


Presented By

Dr. Banchhanidhi Dash

**School of Computer Engineering
KIIT University**

11/18/2023



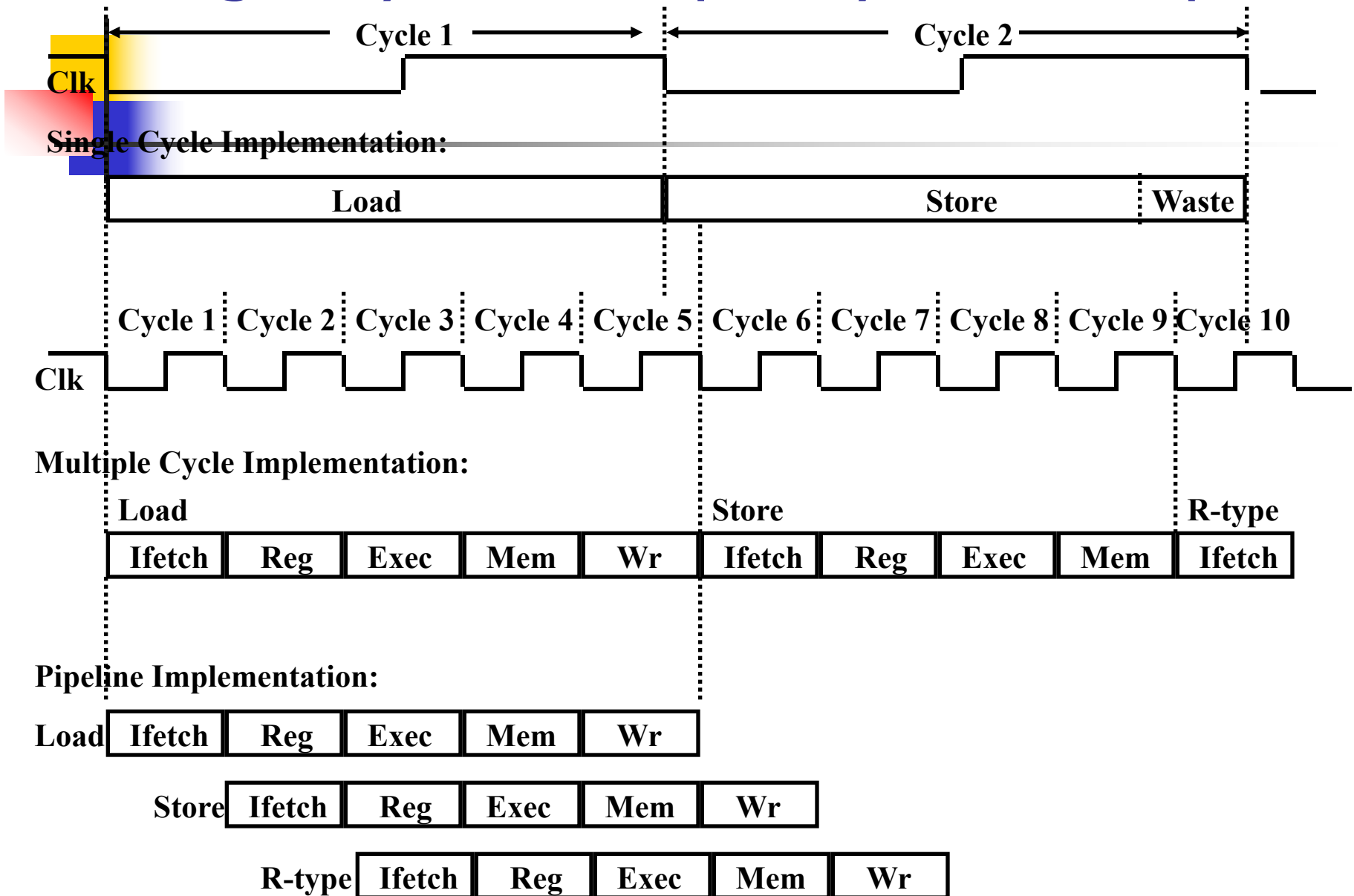
We're ready to look at an implementation
of the MIPS instruction set



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

Single Cycle, Multiple Cycle, vs. Pipeline





Single-cycle Implementation of MIPS

DATAPATH IN MIPS Architecture

The *datapath* and *control* are the two components that come together to be collectively known as the processor.

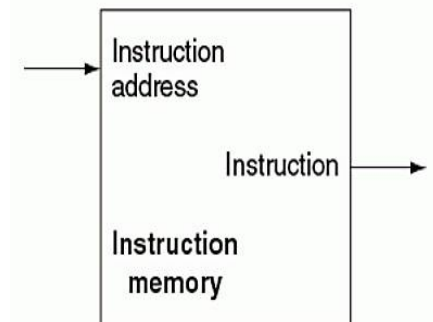
- Datapath consists of the functional units of the processor.
 - Elements that *hold data*.
 - Program counter, register file, instruction memory, etc.
 - Elements that *operate on data*.
 - ALU, adders, etc.
 - Buses for *transferring data* between elements.
- Control commands the datapath regarding when and how to route and operate on data.

Functional elements in DATAPATH

we will look at the datapath elements needed by every instruction.

First, we have *instruction memory*.

Instruction memory is a *state element* that provides *read-access* to the instructions of a program and, given an address as input, supplies the corresponding instruction at that address.



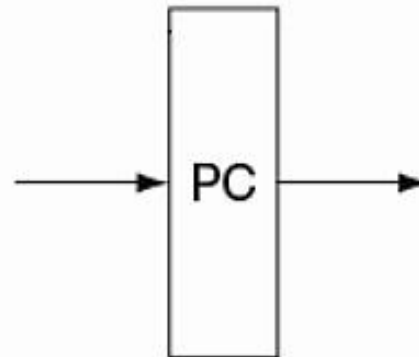
functional elements in DATAPATH

Next, we have the *program counter* or *PC*.

The PC is a state element that holds the *address of the current instruction*. Essentially, it is just a 32-bit register which holds the instruction address and is updated at the end of every clock cycle.

- Normally PC increments sequentially except for branch instructions

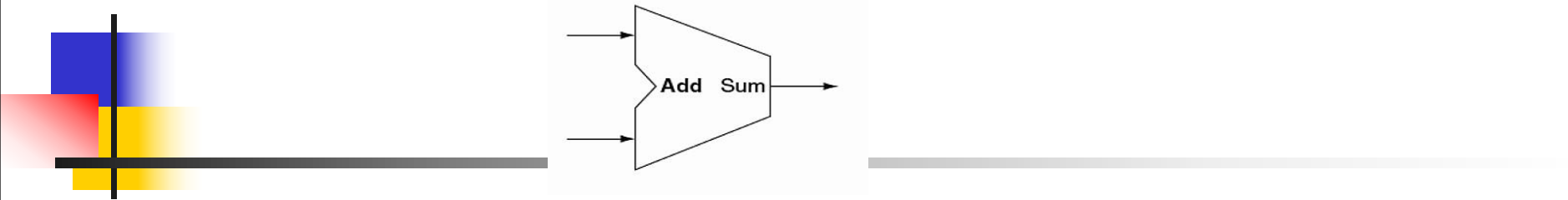
The arrows on either side indicate that the PC state element is both *readable* and *writable*.



Lastly, we have the *adder*.

The *adder* is responsible for *incrementing* the *PC* to hold the address of the next instruction.

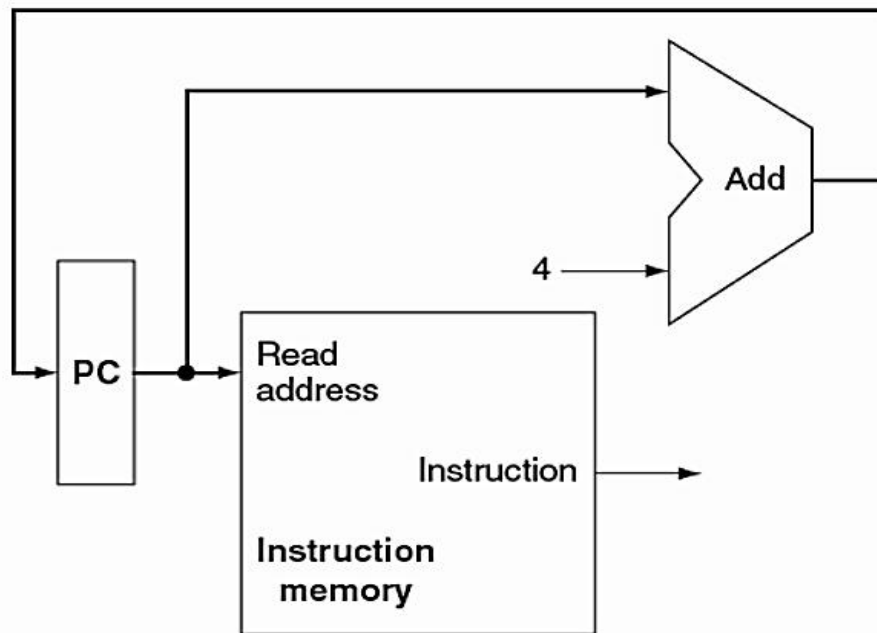
It takes two input values, adds them together and outputs the result.



So now we have instruction memory, *PC*, and adder datapath elements. Now, we can talk about the general steps taken to execute a program.

- *Instruction fetching*: use the address in the *PC* to fetch the current instruction from instruction memory.
- *Instruction decoding*: determine the fields within the instruction
- *Instruction execution*: perform the operation indicated by the instruction.
- Update the *PC* to hold the address of the next instruction.

- Fetch the instruction at the address in PC.
- Decode the instruction.
- Execute the instruction.
- Update the PC to hold the address of the next instruction.



Note: we perform $PC+4$ because MIPS instructions are word-aligned.

To support **R-format instructions**, we'll need to add a state element called a *register file*. A register file is a collection readable/writable registers.

- **Read register 1** - first source register. 5 bits wide.

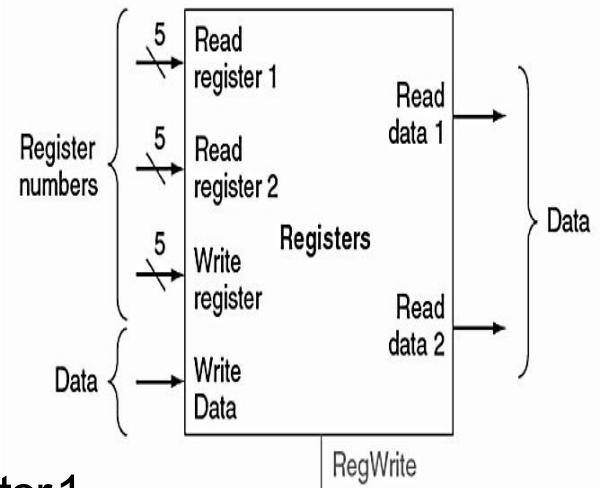
- **Read register 2** - second source register. 5 bits wide.

- **Write register** - destination register. 5 bits wide.

- **Write data** - data to be written to a register. 32 bits wide.

At the bottom, we have the **RegWrite** input. A writing operation only occurs when this bit is set.

The two output ports are:

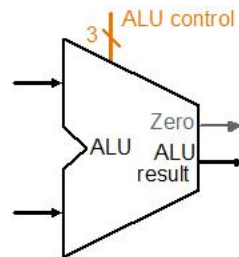


- **Read data 1** - contents of source register 1.
- **Read data 2** - contents of source register 2.

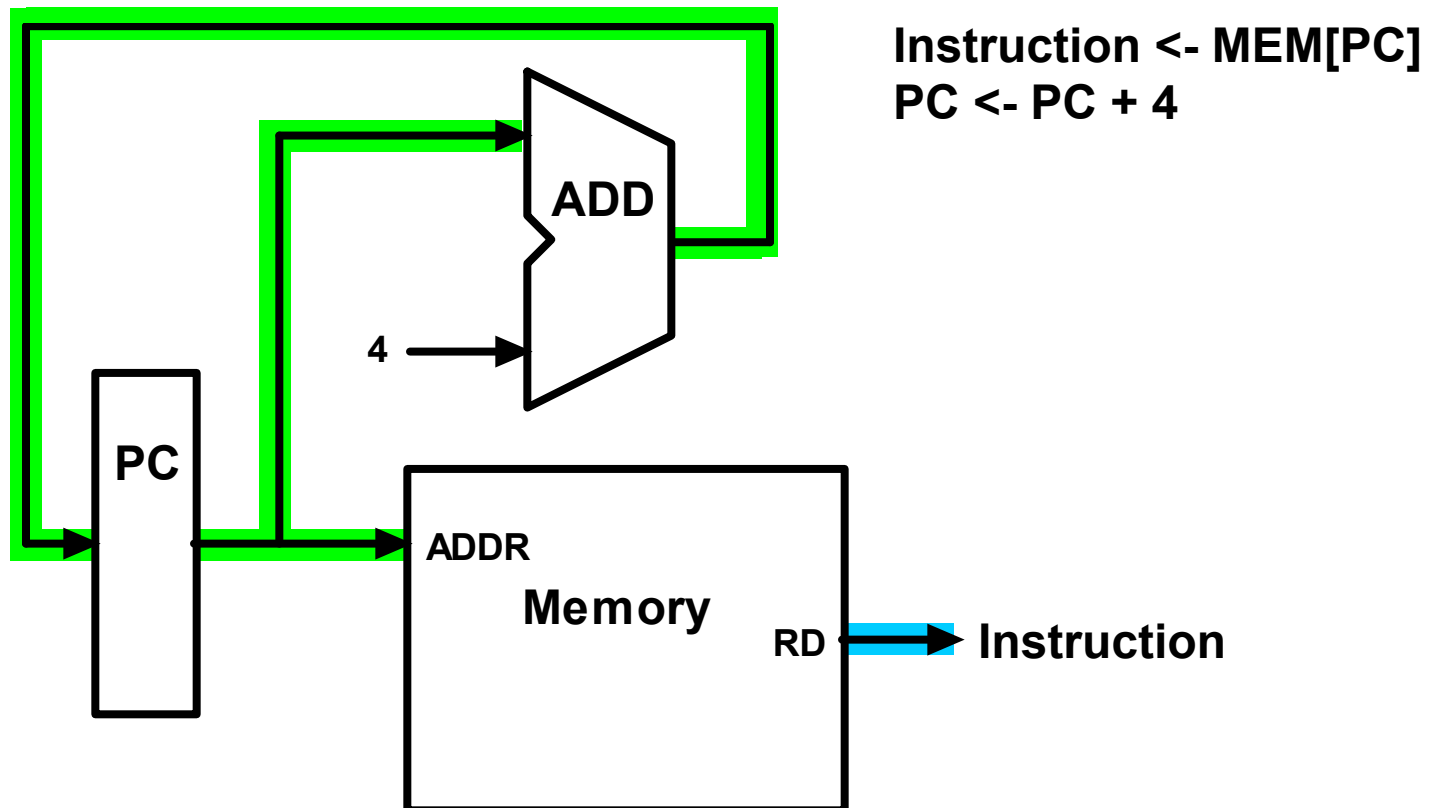
Register file with two read ports and one write port

to execute R-format instructions, we need to include the **ALU** element.

The **ALU** performs the operation indicated by the instruction. It takes two operands, as well as a 3-bit wide operation selector value. The result of the operation is the output value.



Animating the Datapath



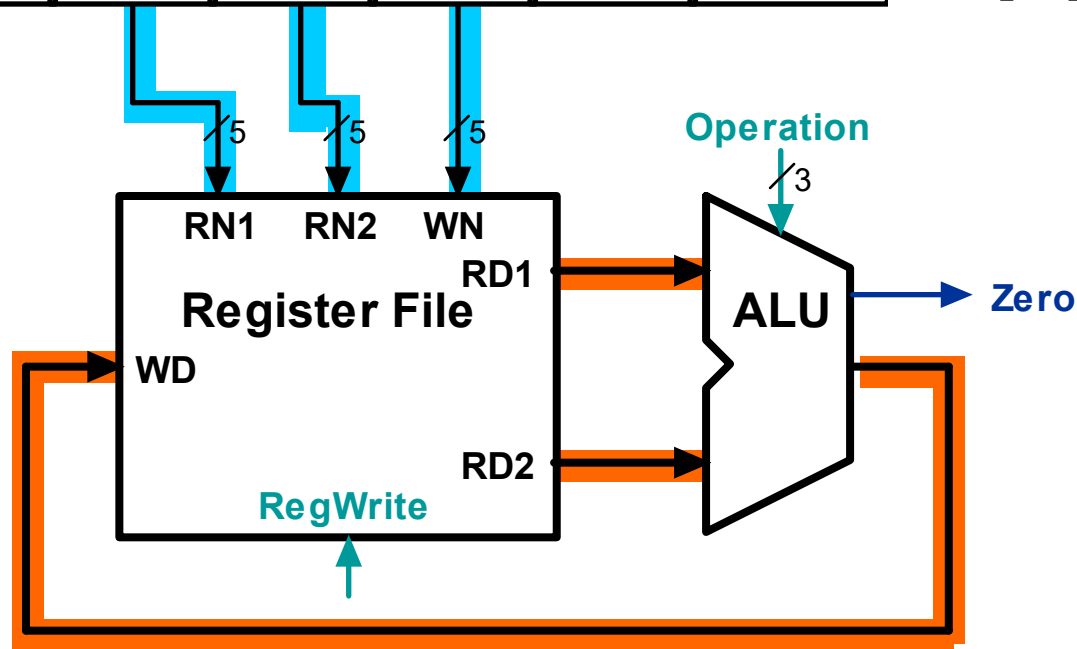
Animating the Datapath for R type Instruction

Instruction

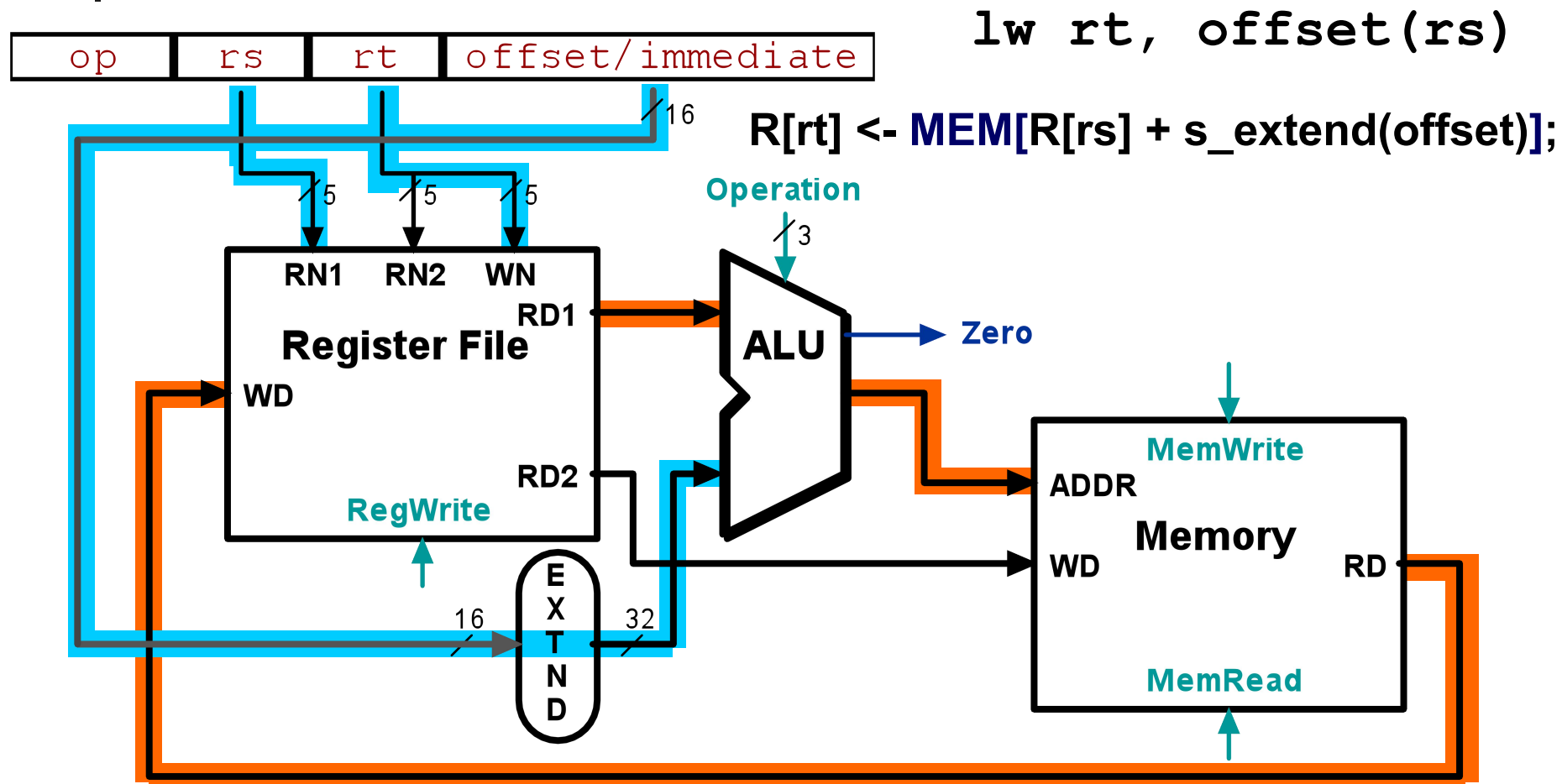


add rd, rs, rt

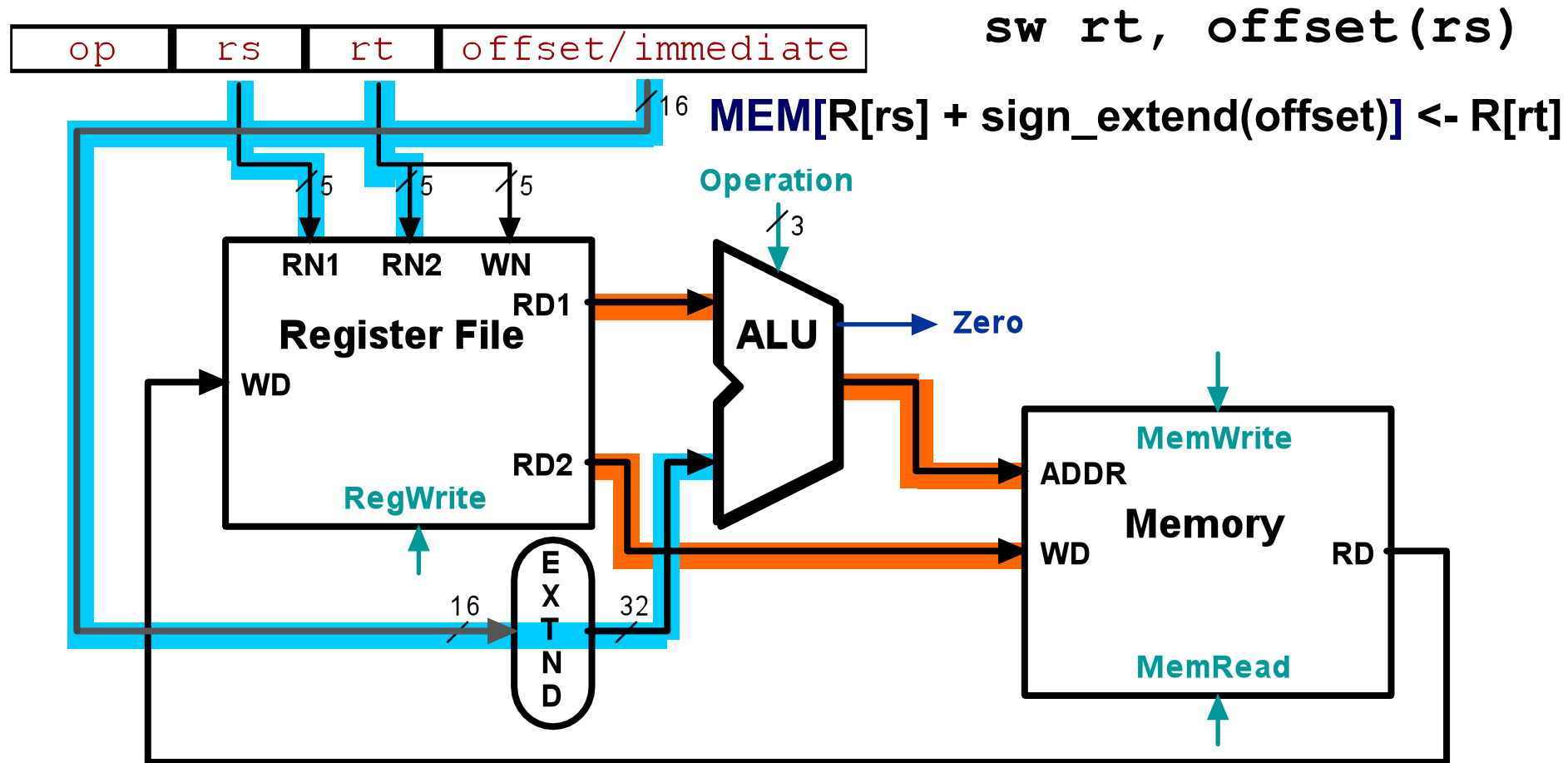
$R[rd] \leftarrow R[rs] + R[rt];$



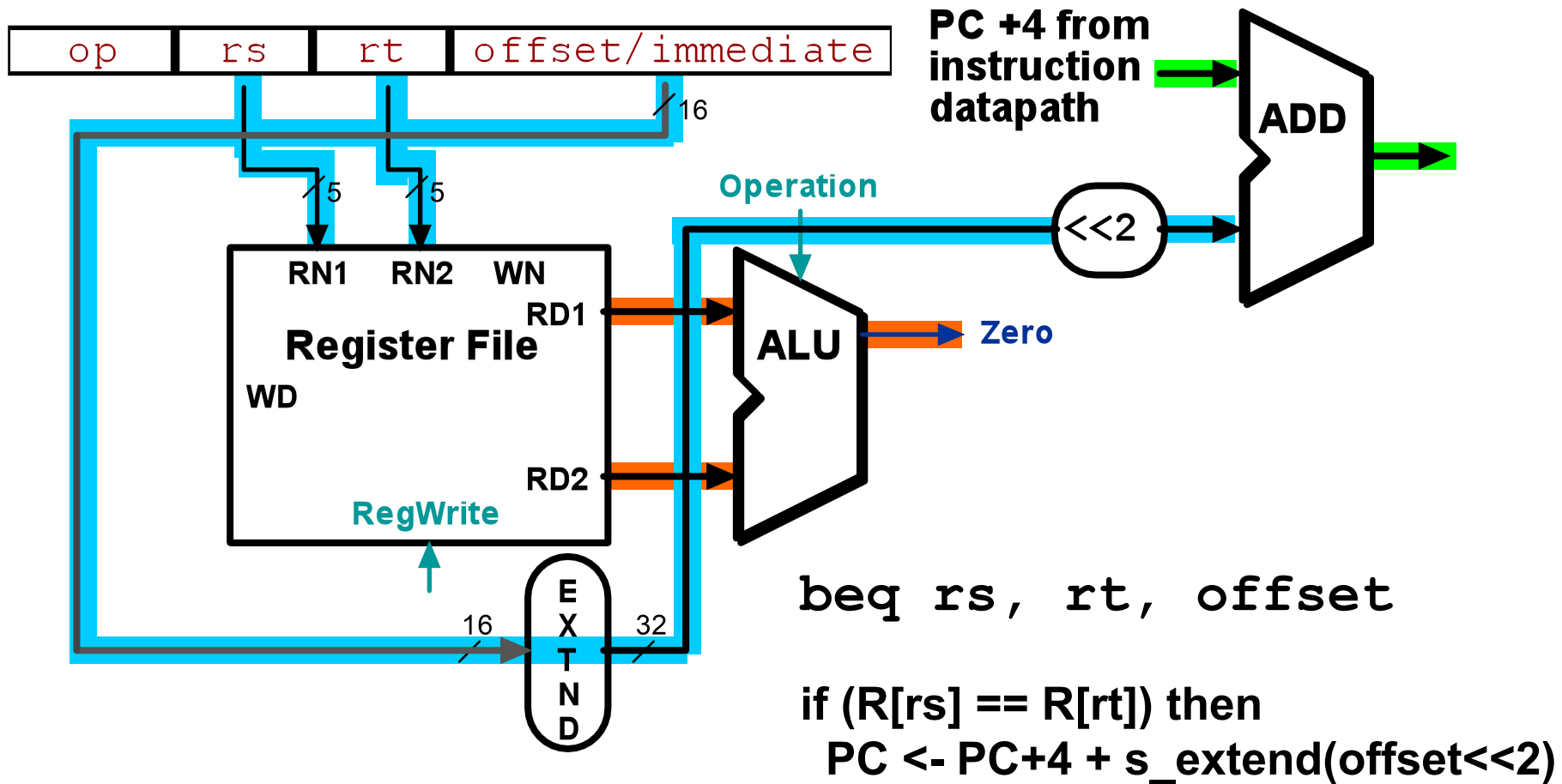
Animating the Datapath for load Instruction



Animating the Datapath store instruction



Animating the Datapath branch Instruction





J format Instruction

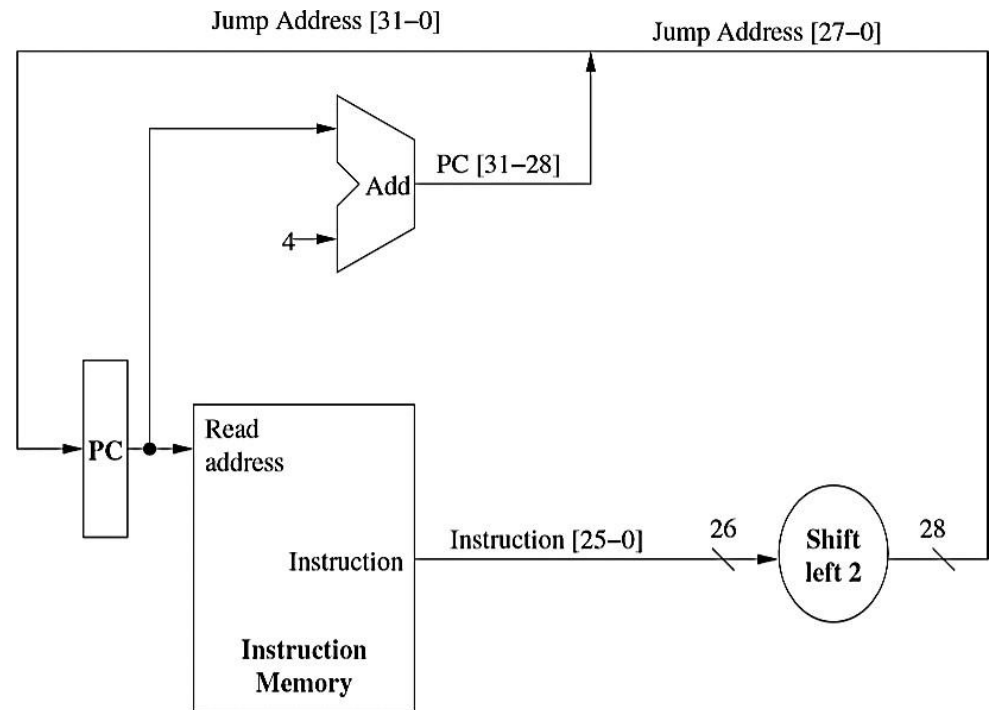
Note, we do not have enough space in the instruction to specify a full target address.

- Branching solves this problem by specifying an offset in words.
- Jump instructions solve this problem by specifying a *portion of an absolute address*
 - Take the 26-bit target address field of the instruction, left-shift by two (instructions are word-aligned),
 - concatenate the result with the upper 4 bits of PC+4.

datapath for J format

Here, we have modified the datapath to work only for the j instruction.

j targaddr



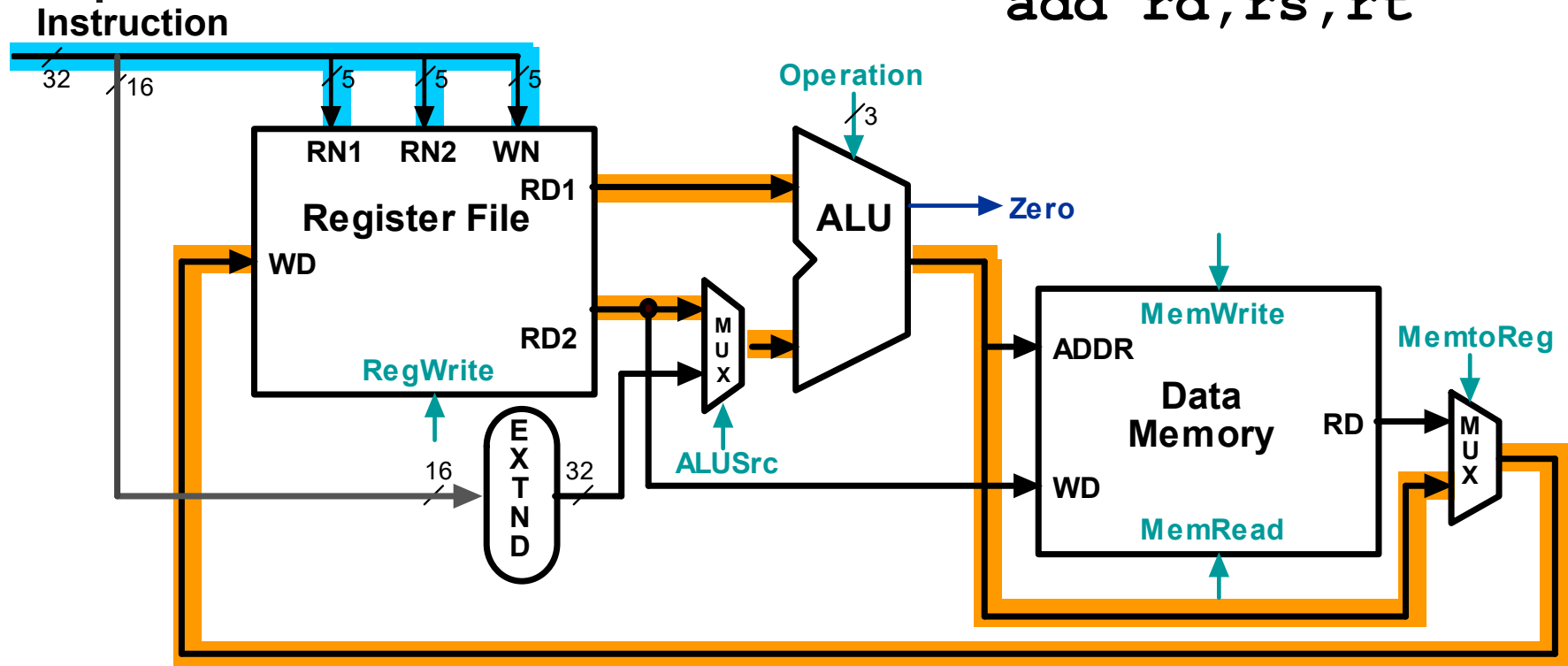


either
U (R-t
ory (lo

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

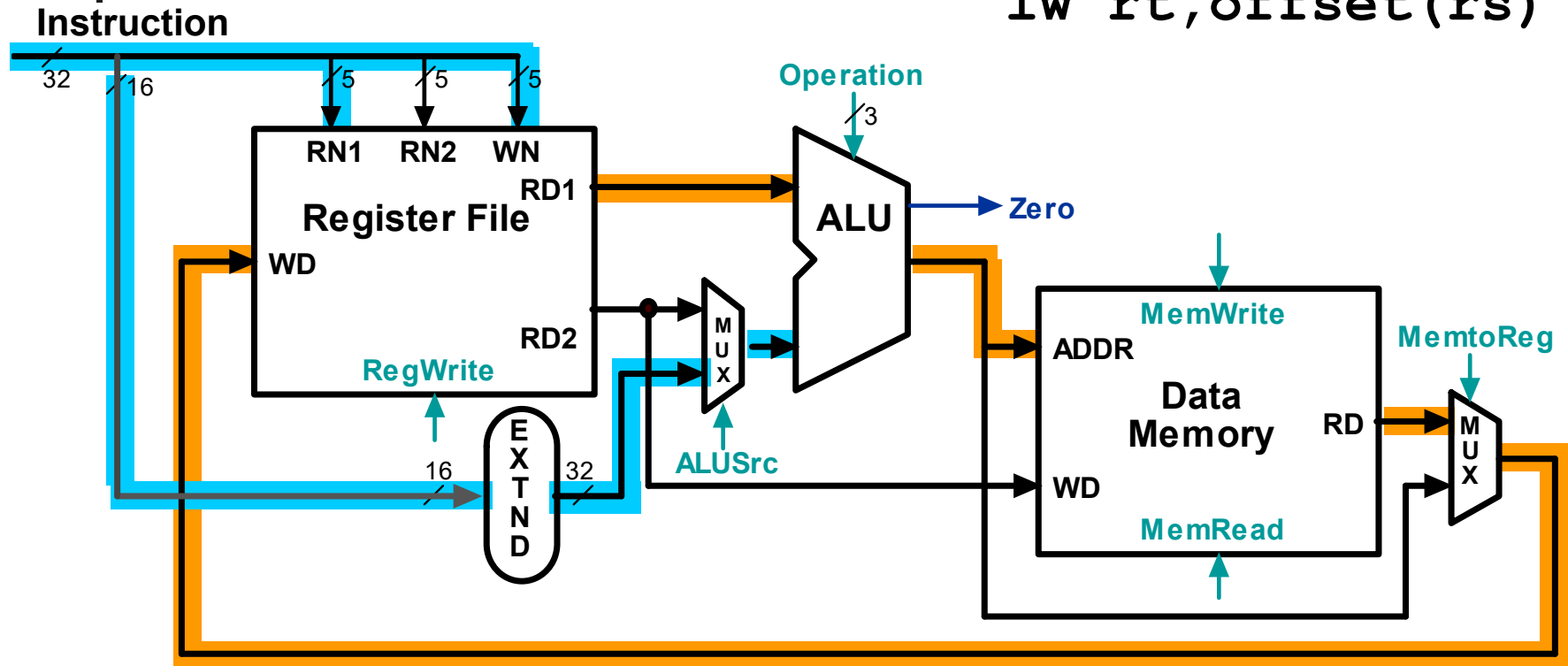
Animating the Datapath: R-type Instruction

`add rd,rs,rt`



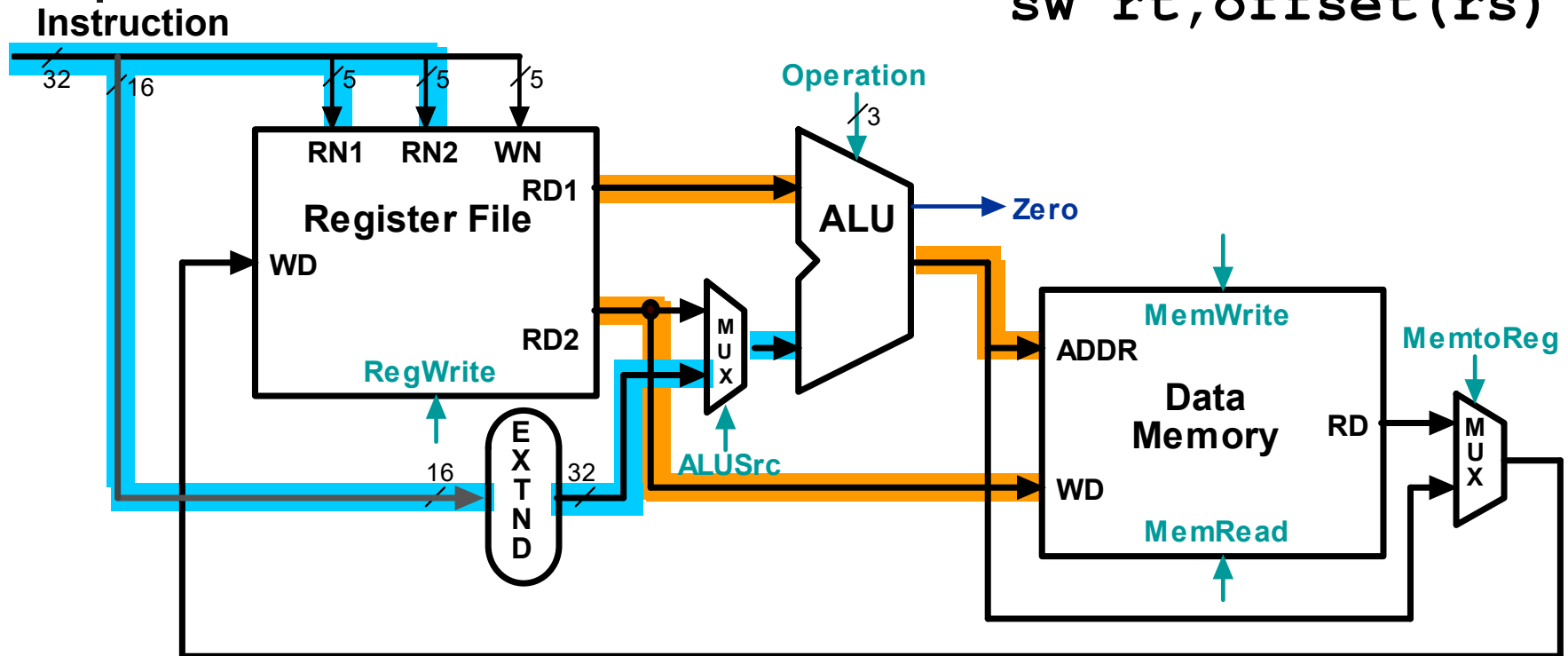
Animating the Datapath: Load Instruction

`lw rt,offset(rs)`

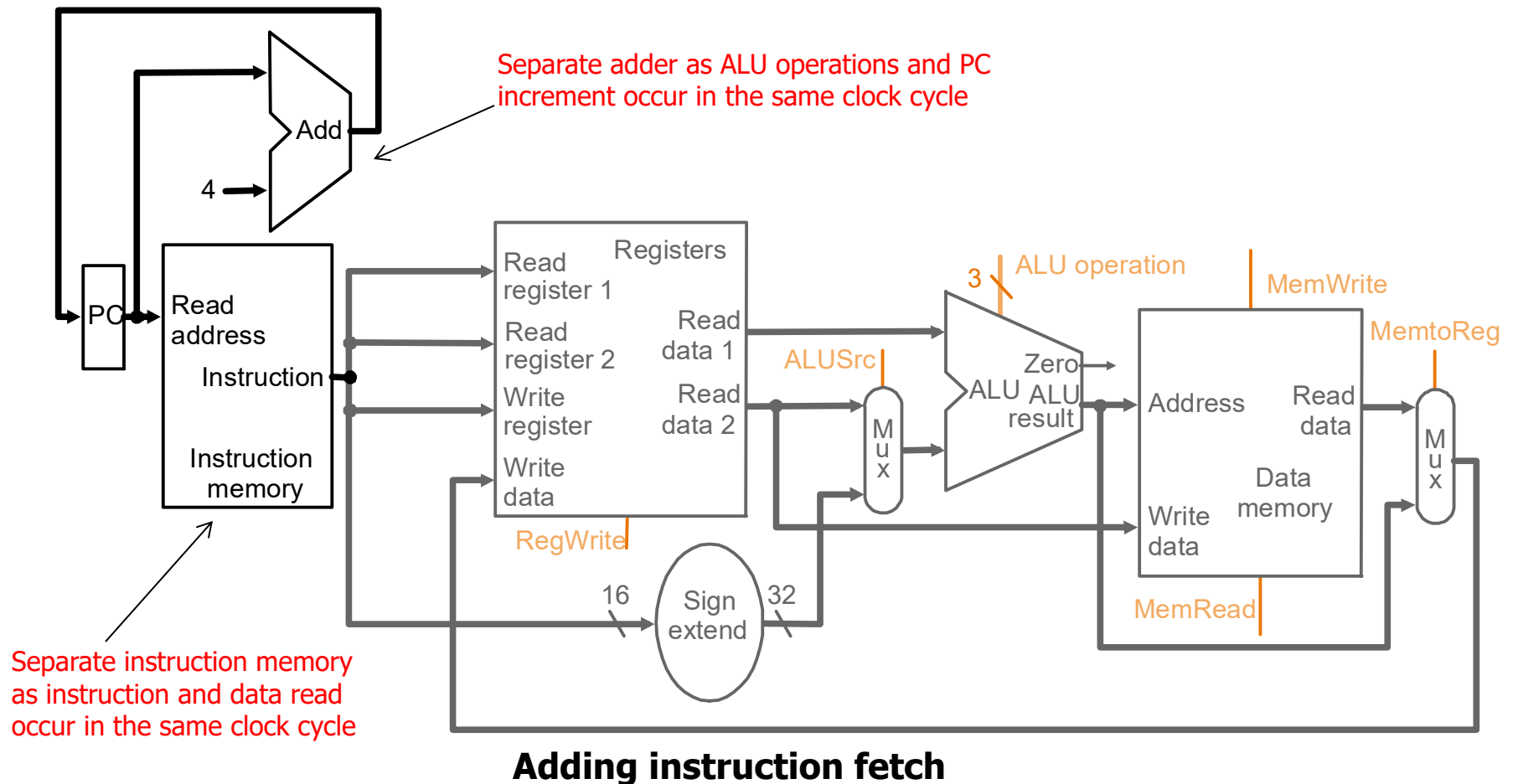


Animating the Datapath: Store Instruction

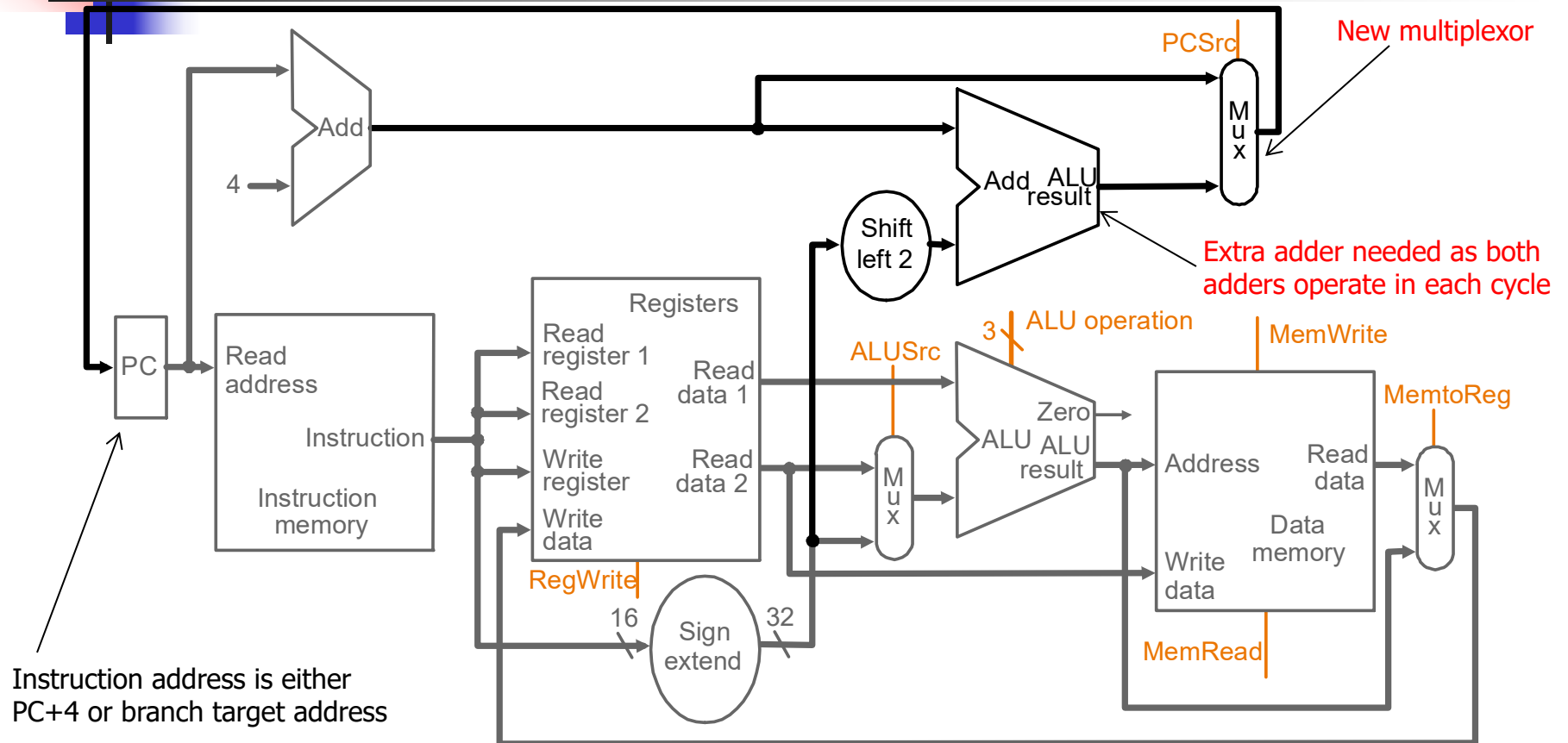
`sw rt,offset(rs)`



MIPS Datapath II: Single-Cycle

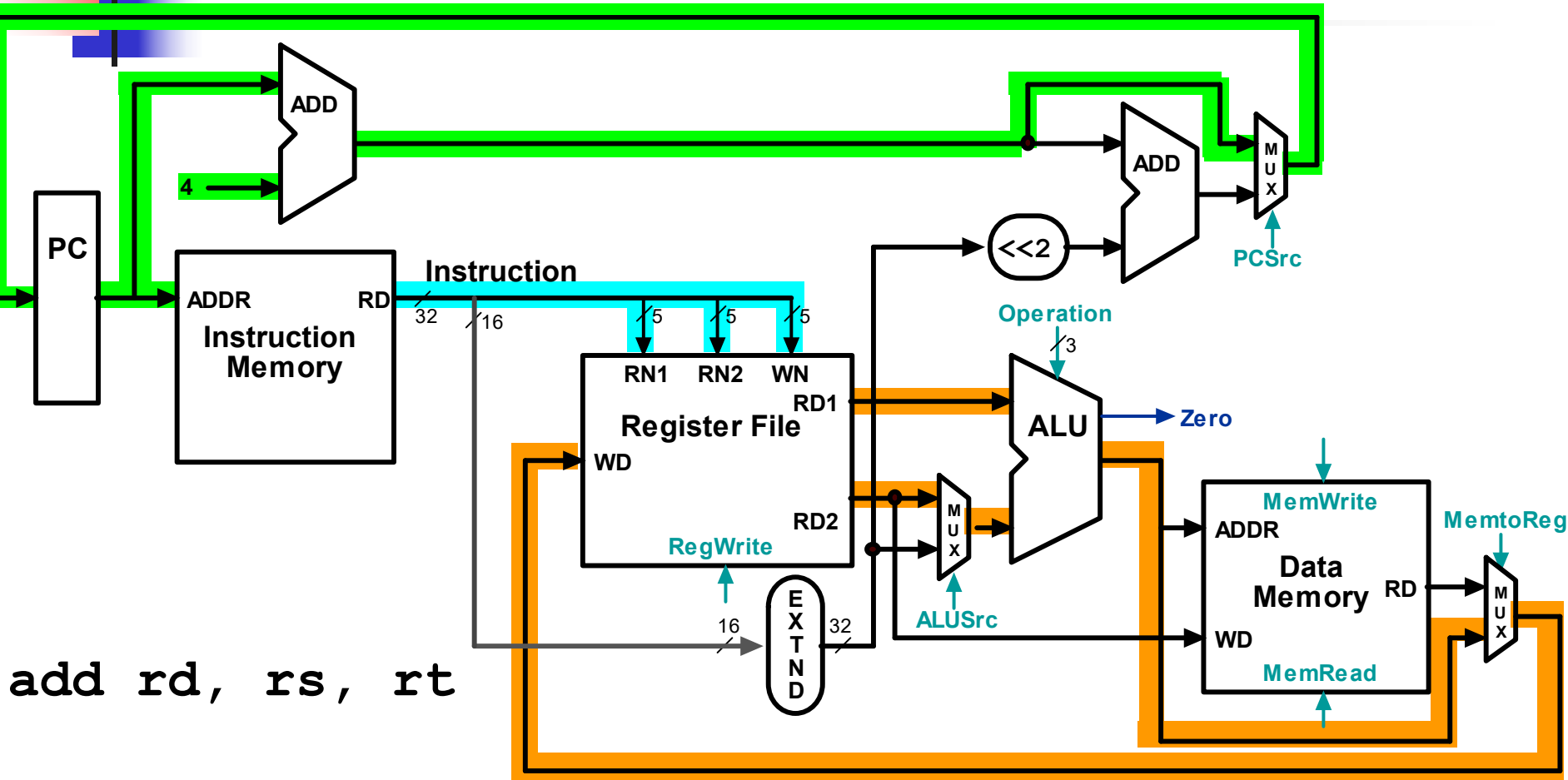


MIPS Datapath III: Single-Cycle

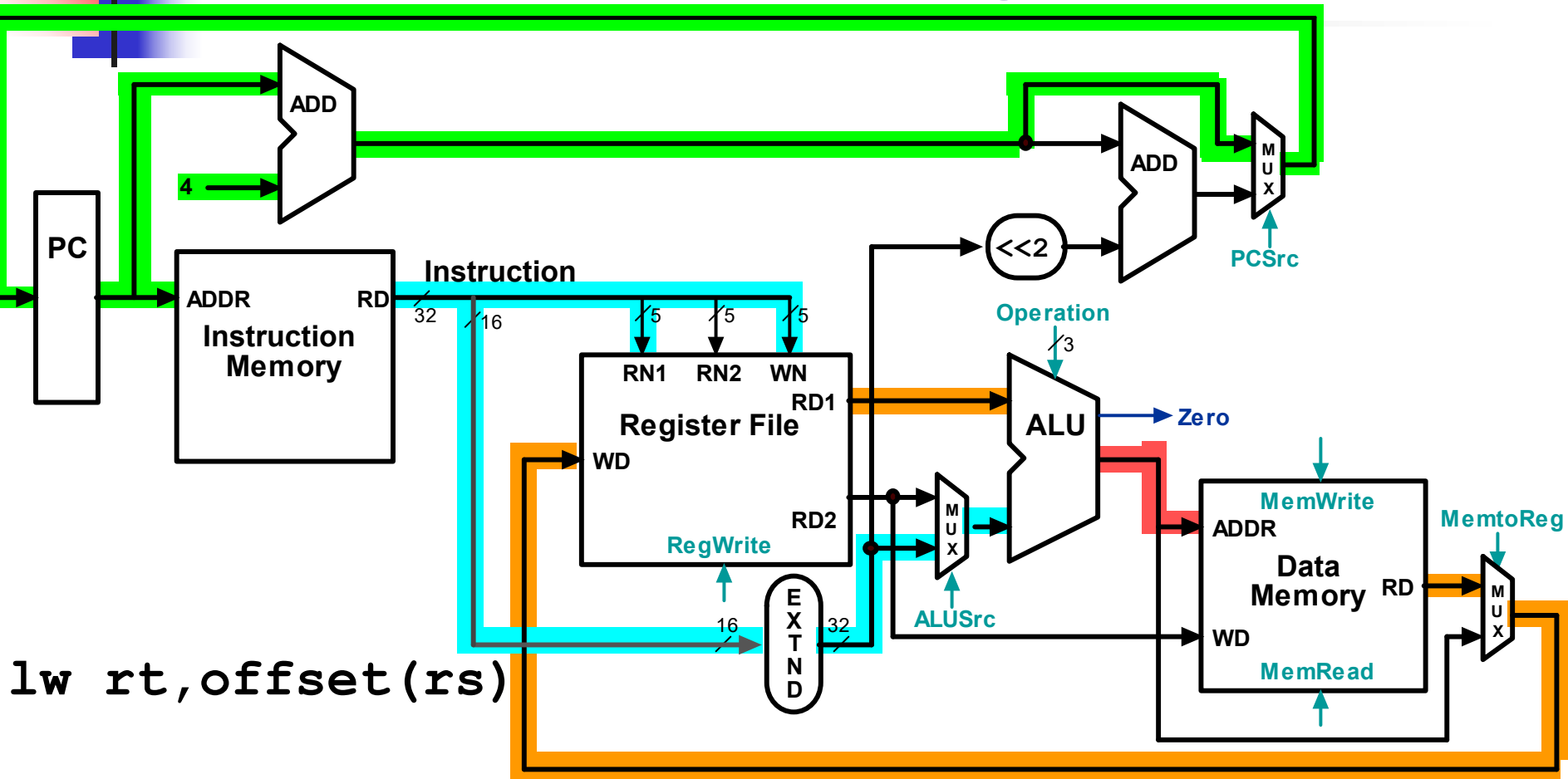


Adding branch capability and another multiplexor

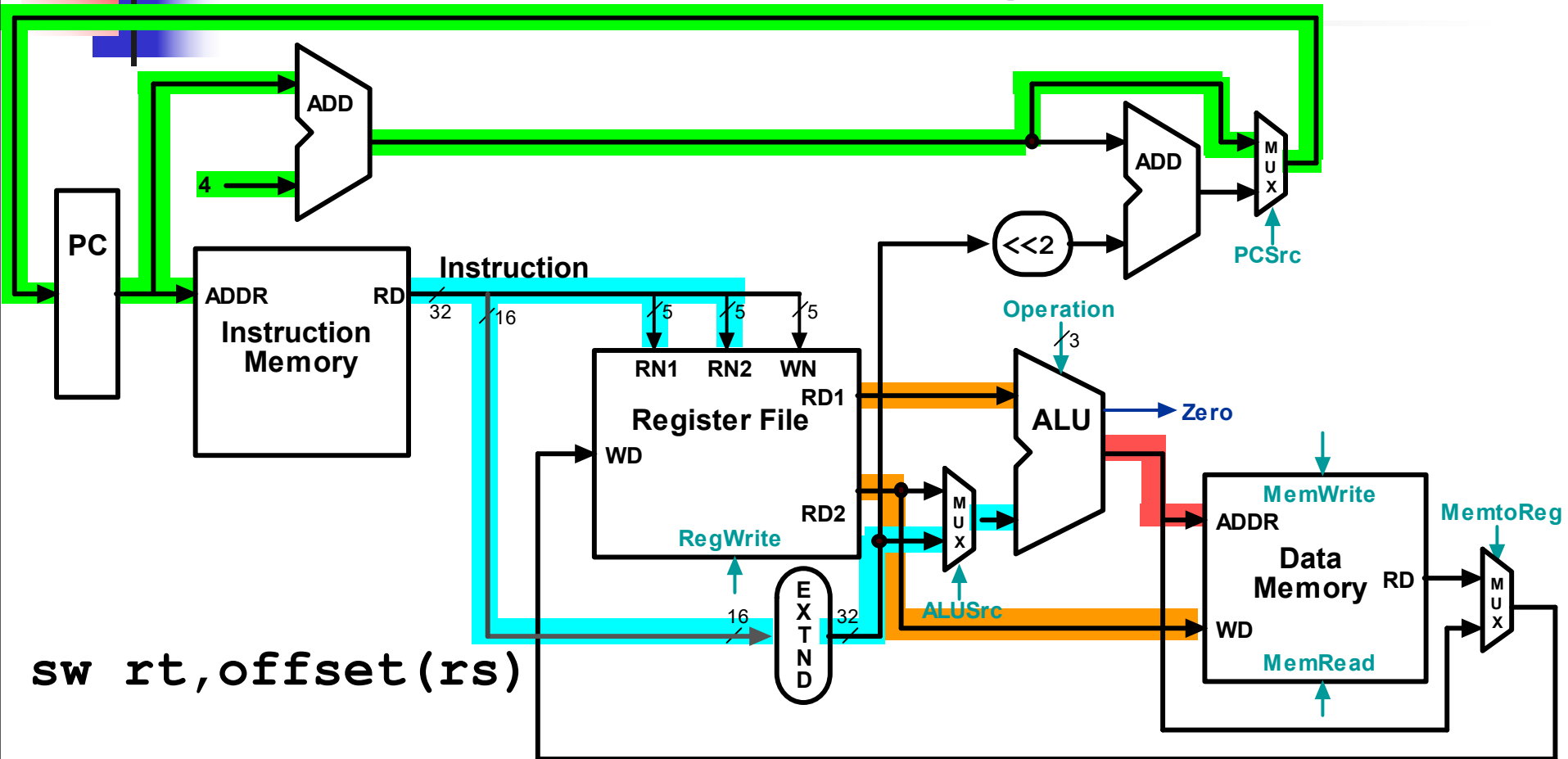
Datapath Executing `add`



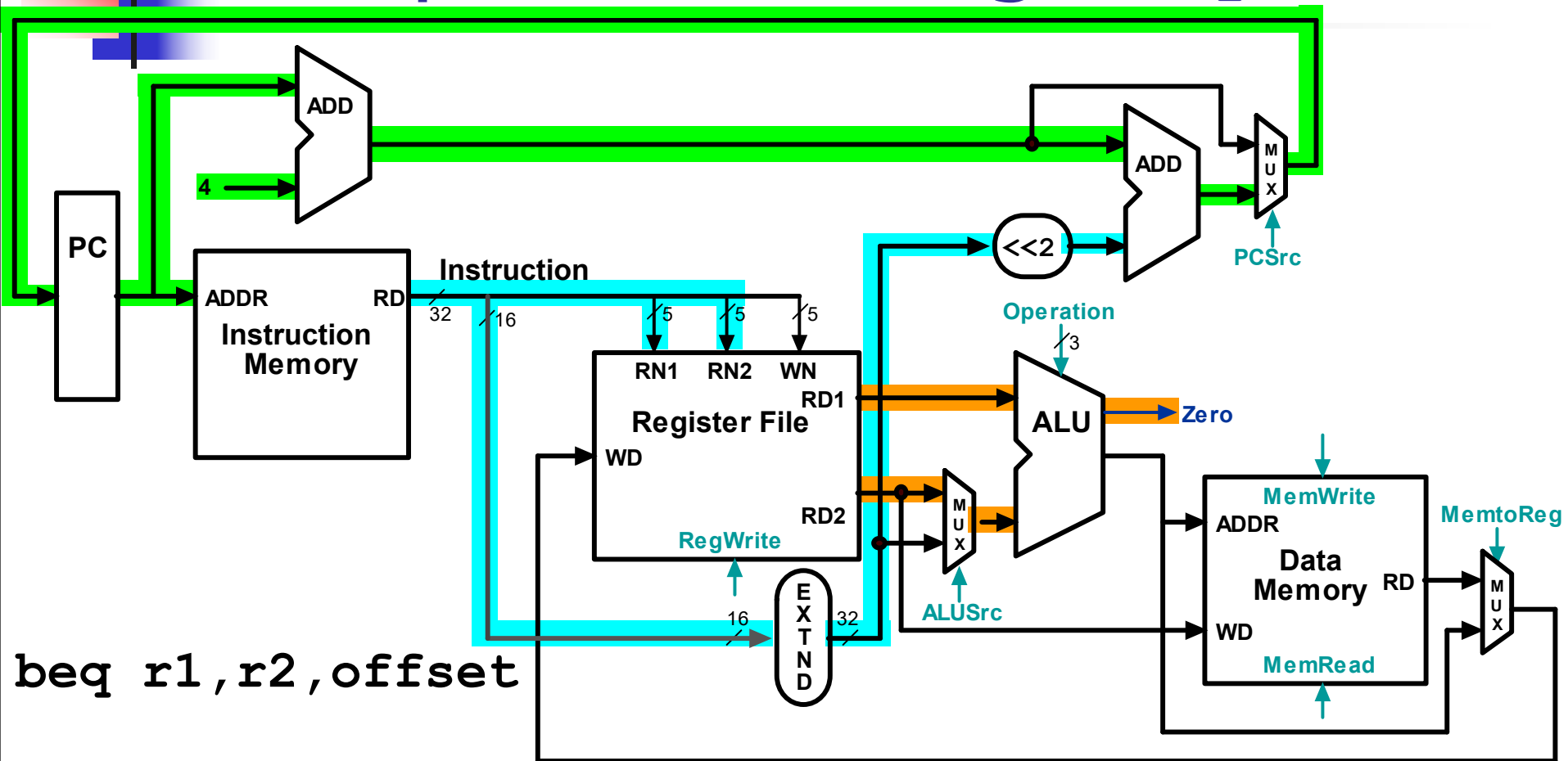
Datapath Executing lw



Datapath Executing sw



Datapath Executing `beq`







Control

- Q.design the single cycle datapath for the following instruction
- I1. SUB R3,R4,R5
- I2 LW R8,32(R9)
- I3 SW R4,8(R6)
- I4 BEQ R11,R12,16
- I5 J 32



Control

- Q. for the following instructions which control signals are to be asserted and deasserted.
- I1. SUB R3,R4,R5
- I2 LW R8,32(R9)
- I3 SW R4,8(R6)
- I4 BEQ R11,R12,16 (content of r1 and r2 is 2)
- I5 1 32



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

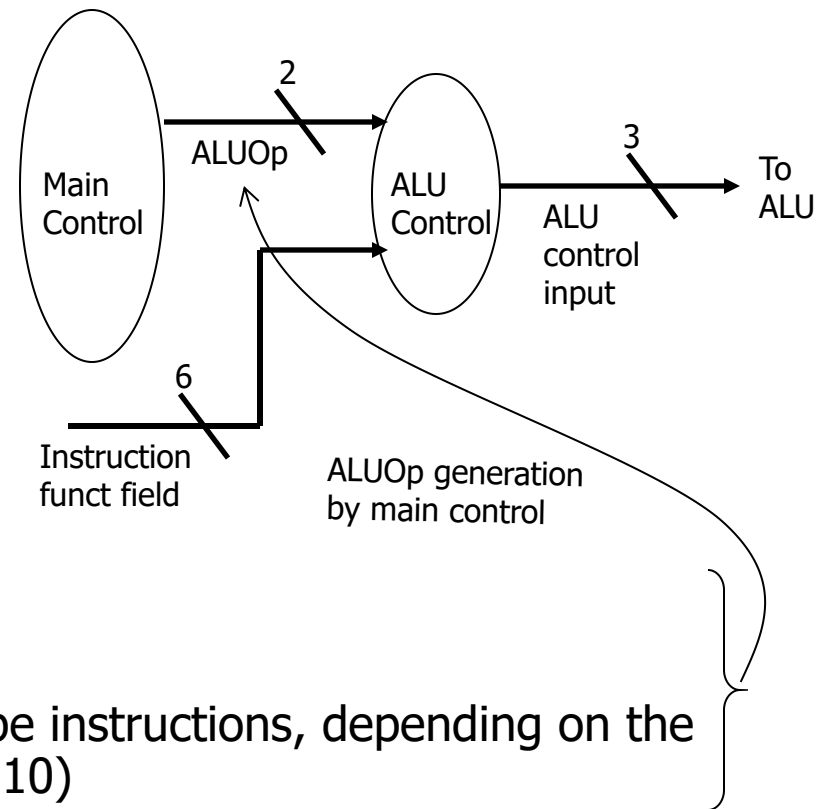
ALU Control

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

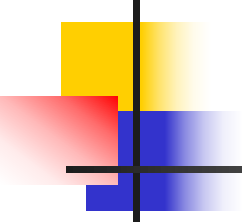
ALU control field	Function
000	and
001	or
010	add
110	sub
111	slt

■ 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



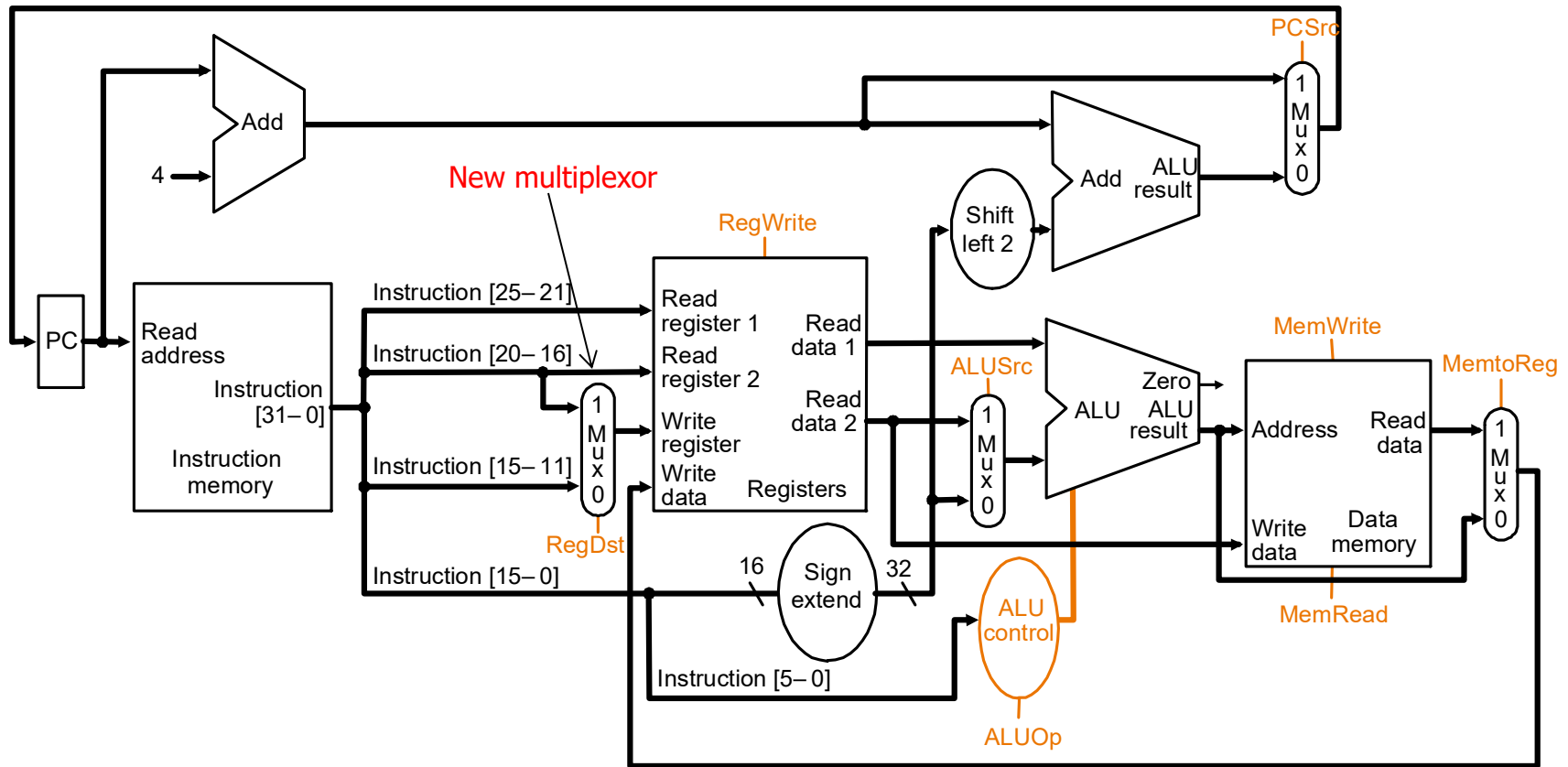
Designing the Main Control

R-type	opcode	rs	rt	rd	shamt	funct
	31-26	25-21	20-16	15-11	10-6	5-0

Load/store or branch	opcode	rs	rt	address
	31-26	25-21	20-16	15-0

- Observations about MIPS instruction format
 - opcode is always in bits 31-26
 - two registers to be read are always rs (bits 25-21) and rt (bits 20-16)
 - base register for load/stores is always rs (bits 25-21)
 - 16-bit offset for branch equal and load/store is always bits 15-0
 - 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):

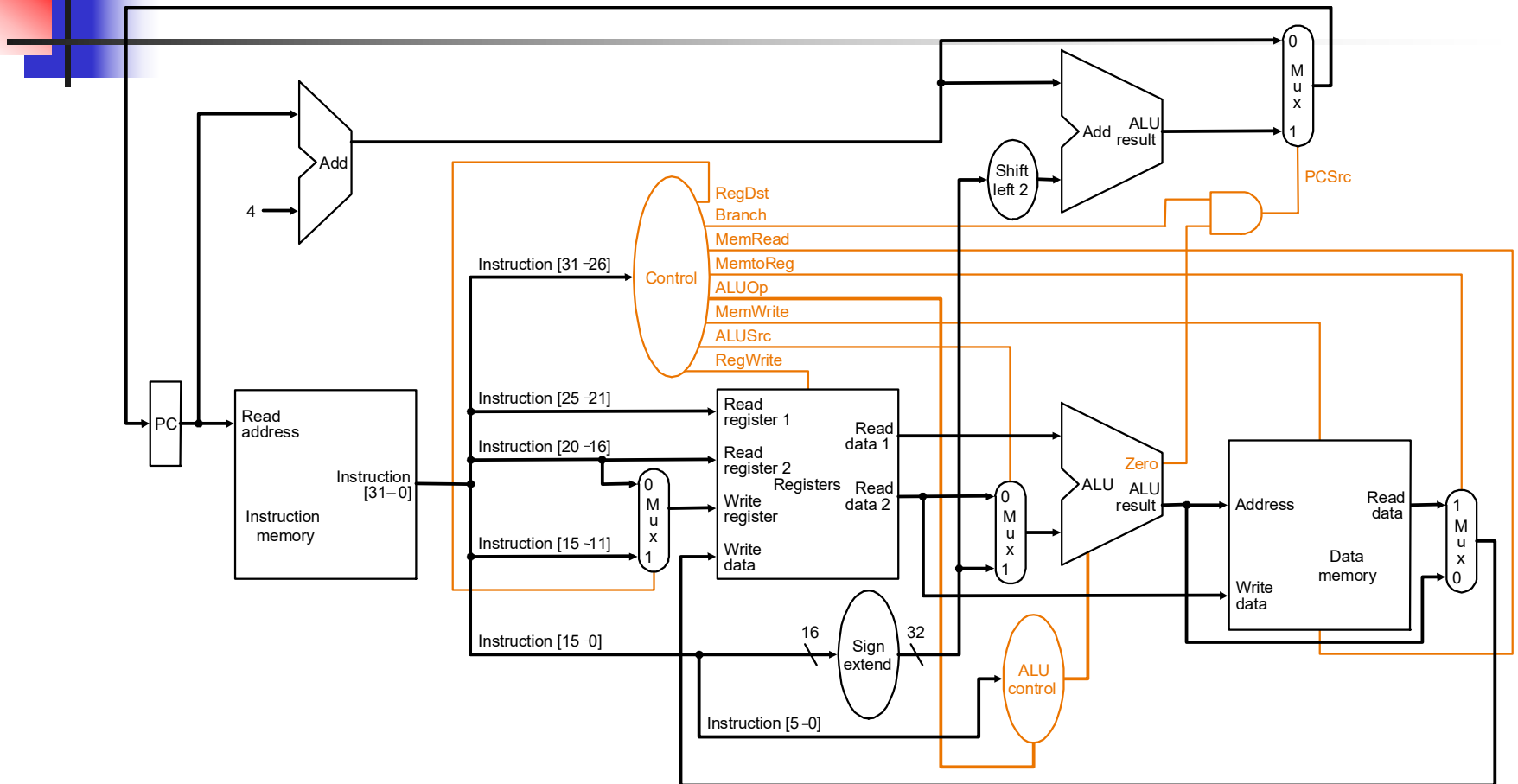


Control Signals

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

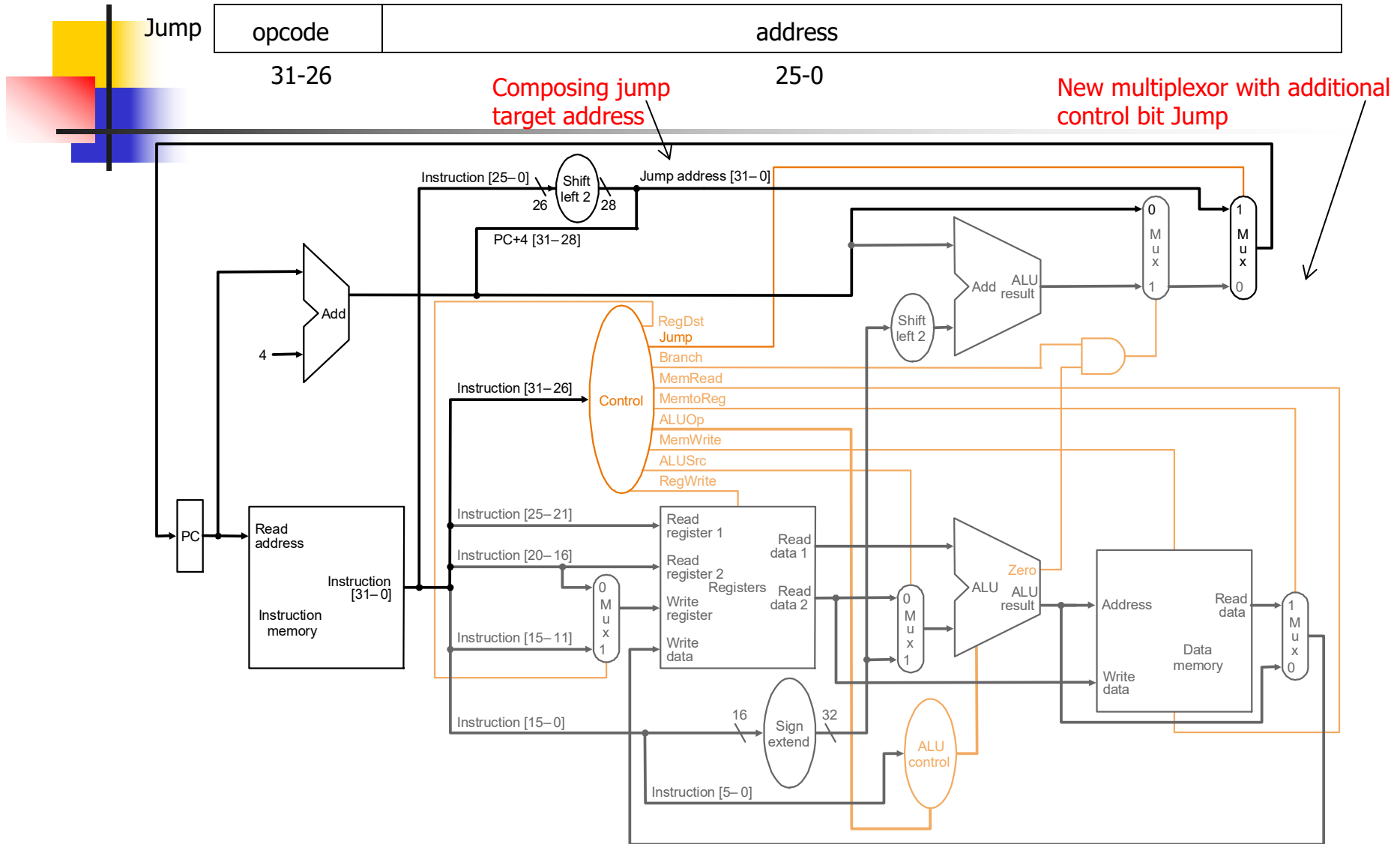
Effects of the seven control signals

Datapath with Control II



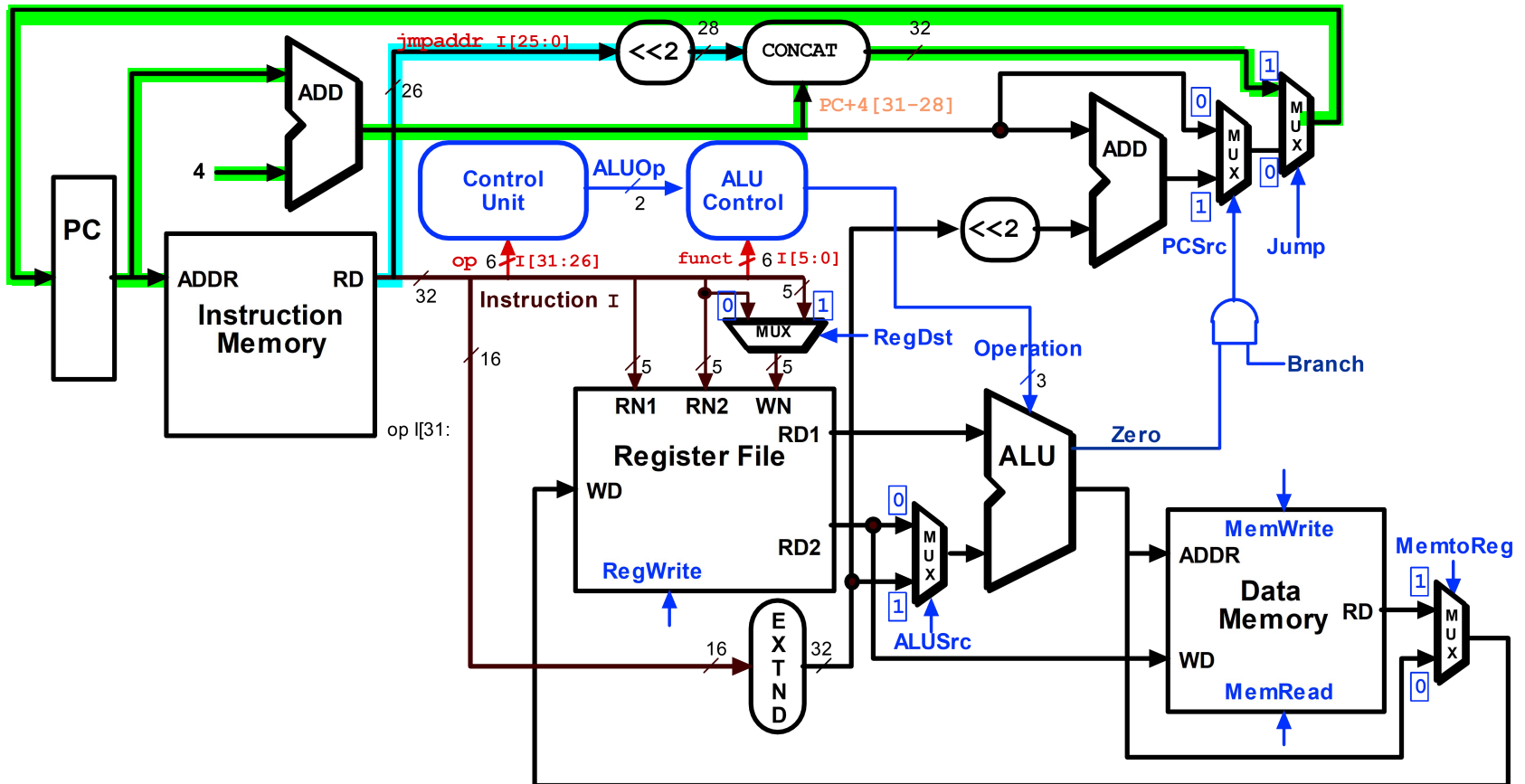
MIPS datapath with the control unit: input to control is the 6-bit instruction opcode field, output is seven 1-bit signals and the 2-bit ALUOp signal

Datapath with Control III



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

Datapath Executing j





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*



Control

- Q.design the single cycle datapath for the following instruction
- I1. SUB R3,R4,R5
- I2 LW R8,32(R9)
- I3 SW R4,8(R6)
- I4 BEQ R11,R12,16
- I5 J 32



Control

- Q. for the following instructions which control signals are to be asserted and deasserted.
- I1. SUB R3,R4,R5
- I2 LW R8,32(R9)
- I3 SW R4,8(R6)
- I4 BEQ R11,R12,16 (content of r1 and r2 is 2)
- I5 J 32



Pipeline Implementation of MIPS

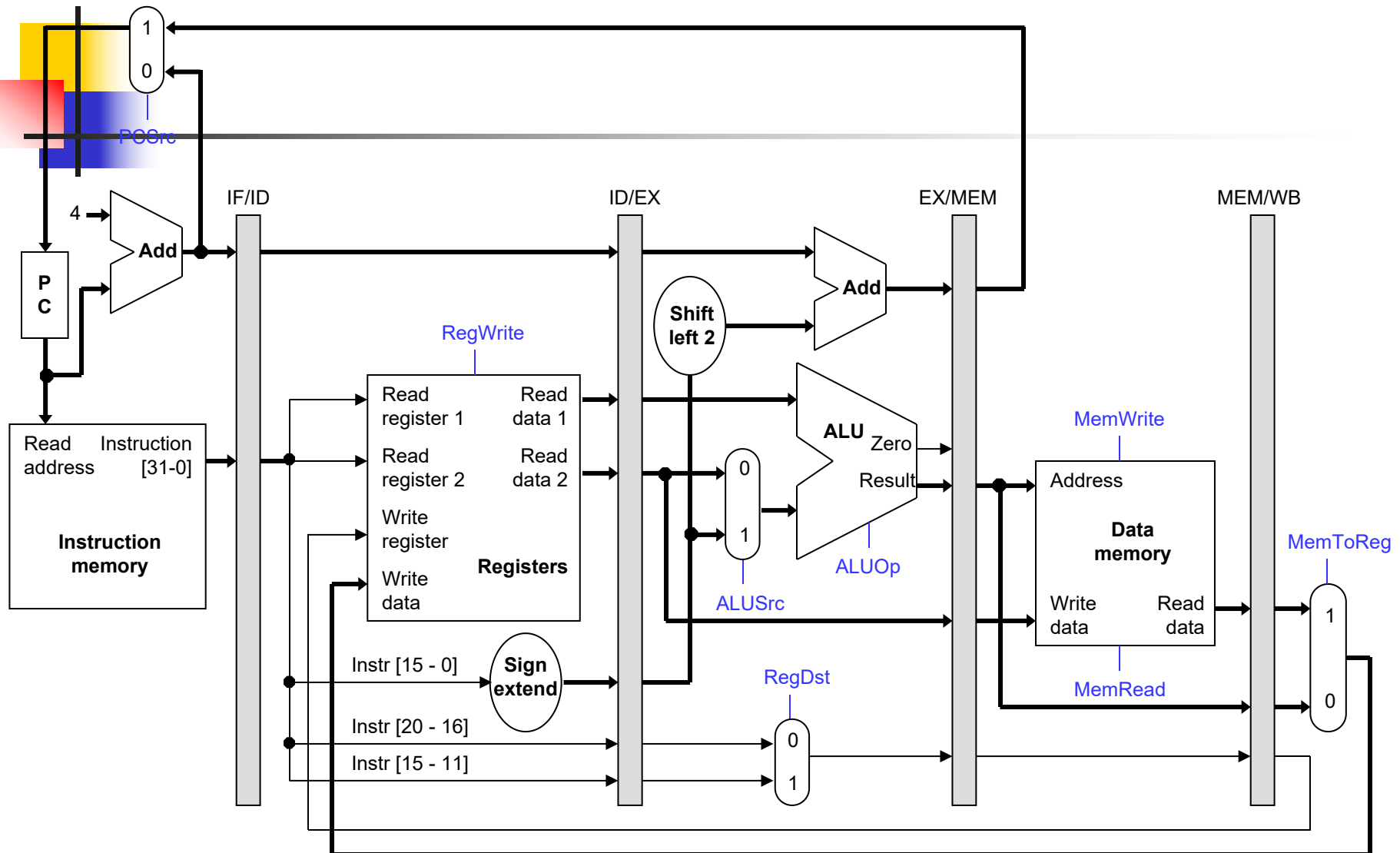


Pipeline registers

- In pipelining, we divide instruction execution into multiple cycles
 - IF ID EX MEM WB
- Information computed during one cycle may be needed in a later cycle:
 - Instruction read in IF stage determines which registers are fetched in ID stage, what immediate is used for EX stage, and what destination register is for WB
 - Register values read in ID are used in EX and/or MEM stages
 - ALU output produced in EX is an effective address for MEM or a result for WB
- A lot of information to save!
 - Saved in intermediate registers called **pipeline registers**
- The registers are named for the stages they connect:

IF/ID ID/EX EX/MEM MEM/WB
- No register is needed after the WB stage, because after WB the instruction is done

Pipelined datapath

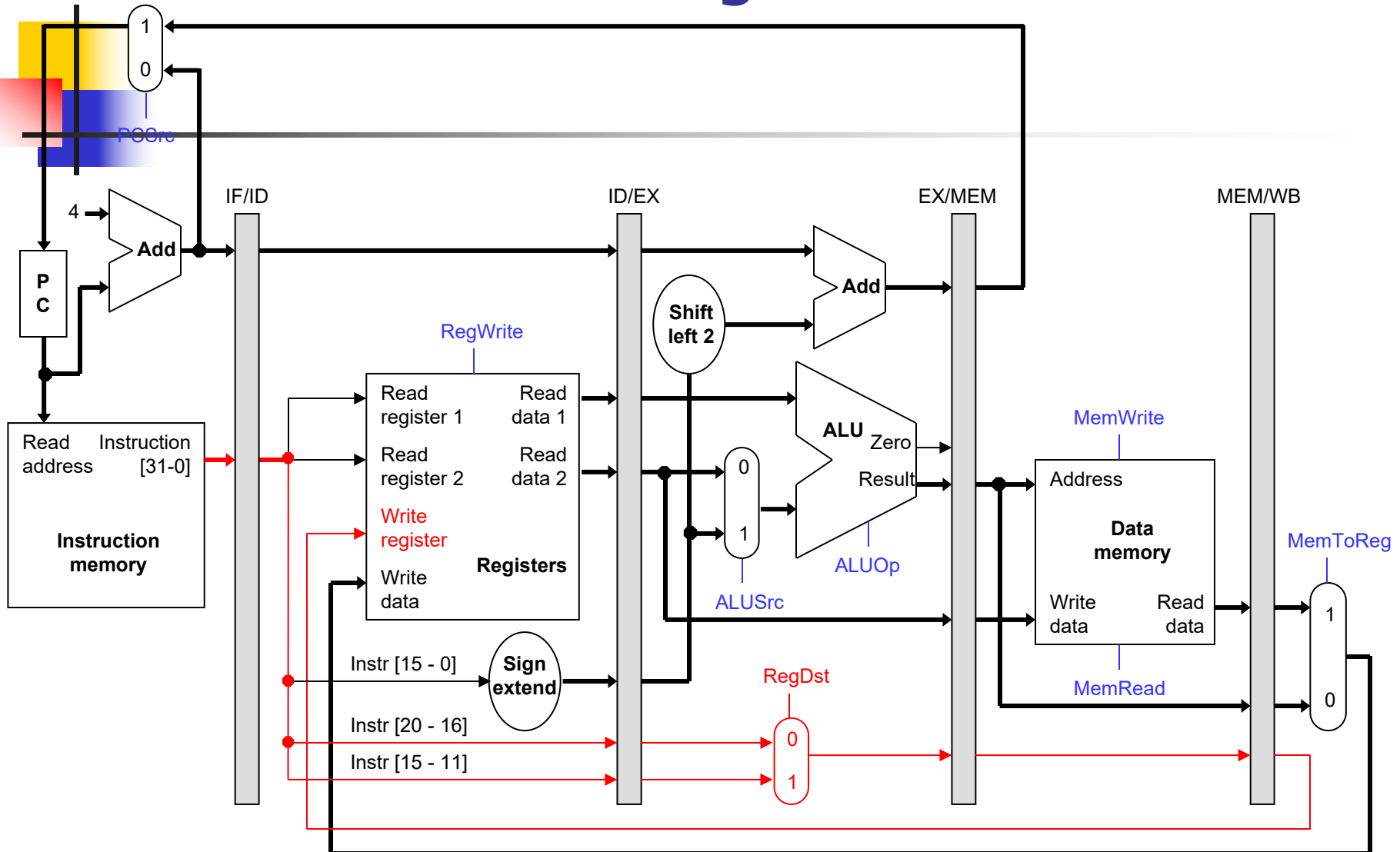




Propagating values forward

- Data values required *later* propagated through the pipeline registers
- The most extreme example is the destination register (r_d or r_t)
 - It is retrieved in IF, but isn't updated until the WB
 - Thus, it must be passed through *all* pipeline stages, as shown in red on the next slide
- Notice that we can't keep a single "instruction register," because the pipelined machine needs to fetch a new instruction every clock cycle

The destination register



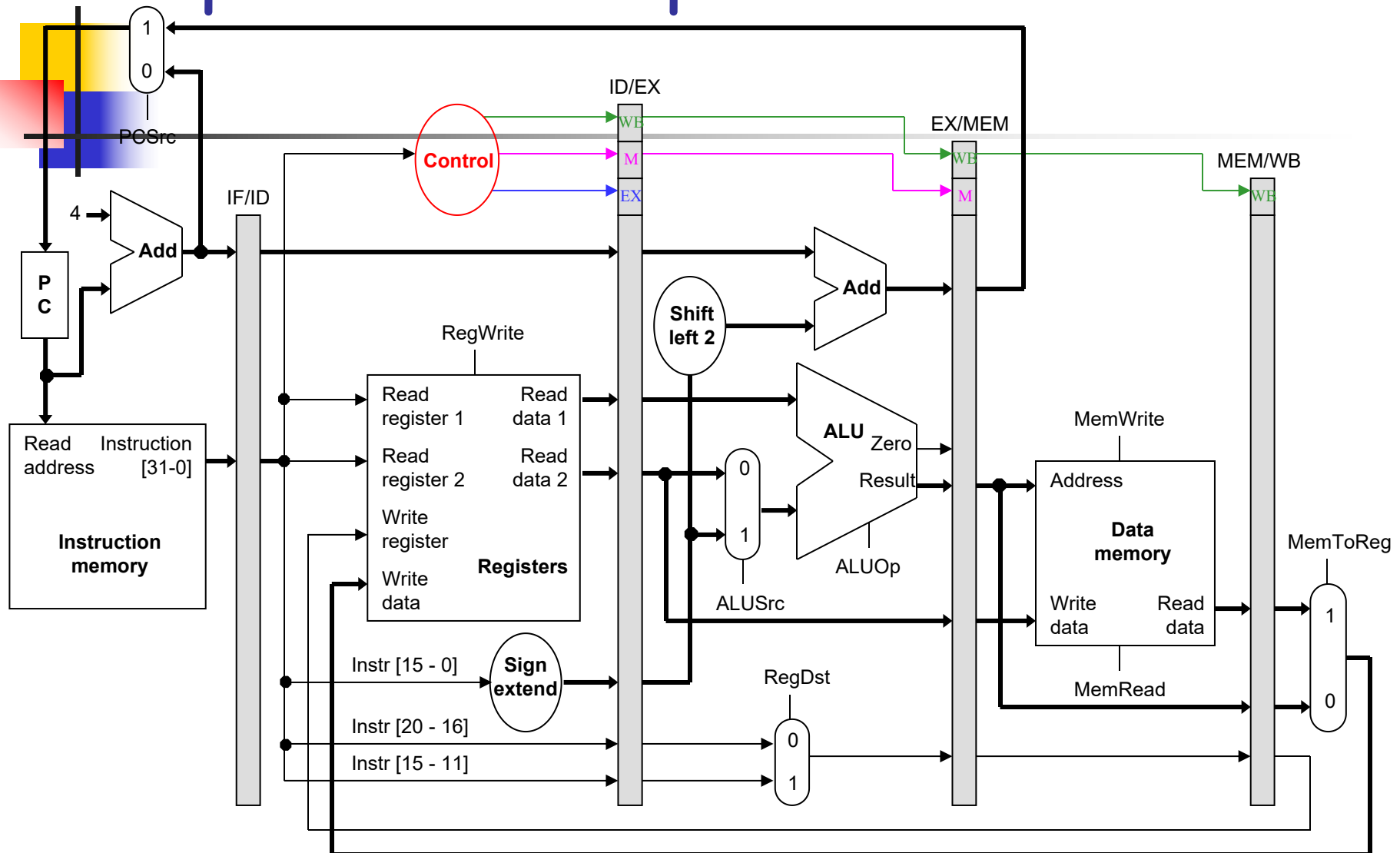


What about control signals?

- Control signals generated similar to the single-cycle processor
 - in the ID stage, the processor decodes the instruction fetched in IF and produces the appropriate control values
- Some of the control signals will not be needed until later stages
 - These signals must be propagated through the pipeline until they reach the appropriate stage
 - We just pass them in the pipeline registers, along with the data
- Control signals can be categorized by the pipeline stage that uses them

Stage	Control signals needed		
EX	ALUSrc	ALUOp	RegDst
MEM	MemRead	MemWrite	PCSrc
WB	RegWrite	MemToReg	

Pipelined datapath and control





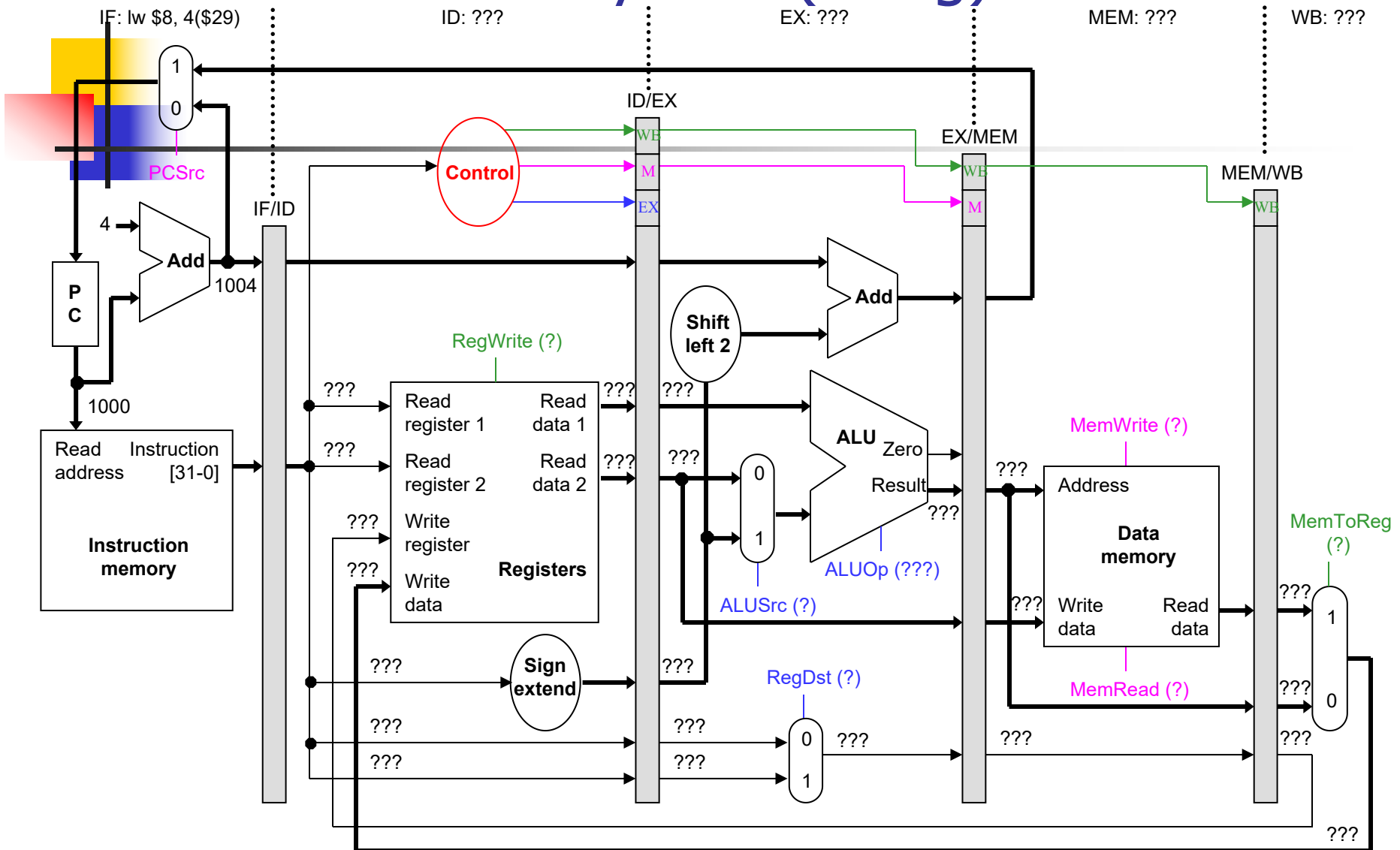
An example execution sequence

- Here's a sample sequence of instructions to execute

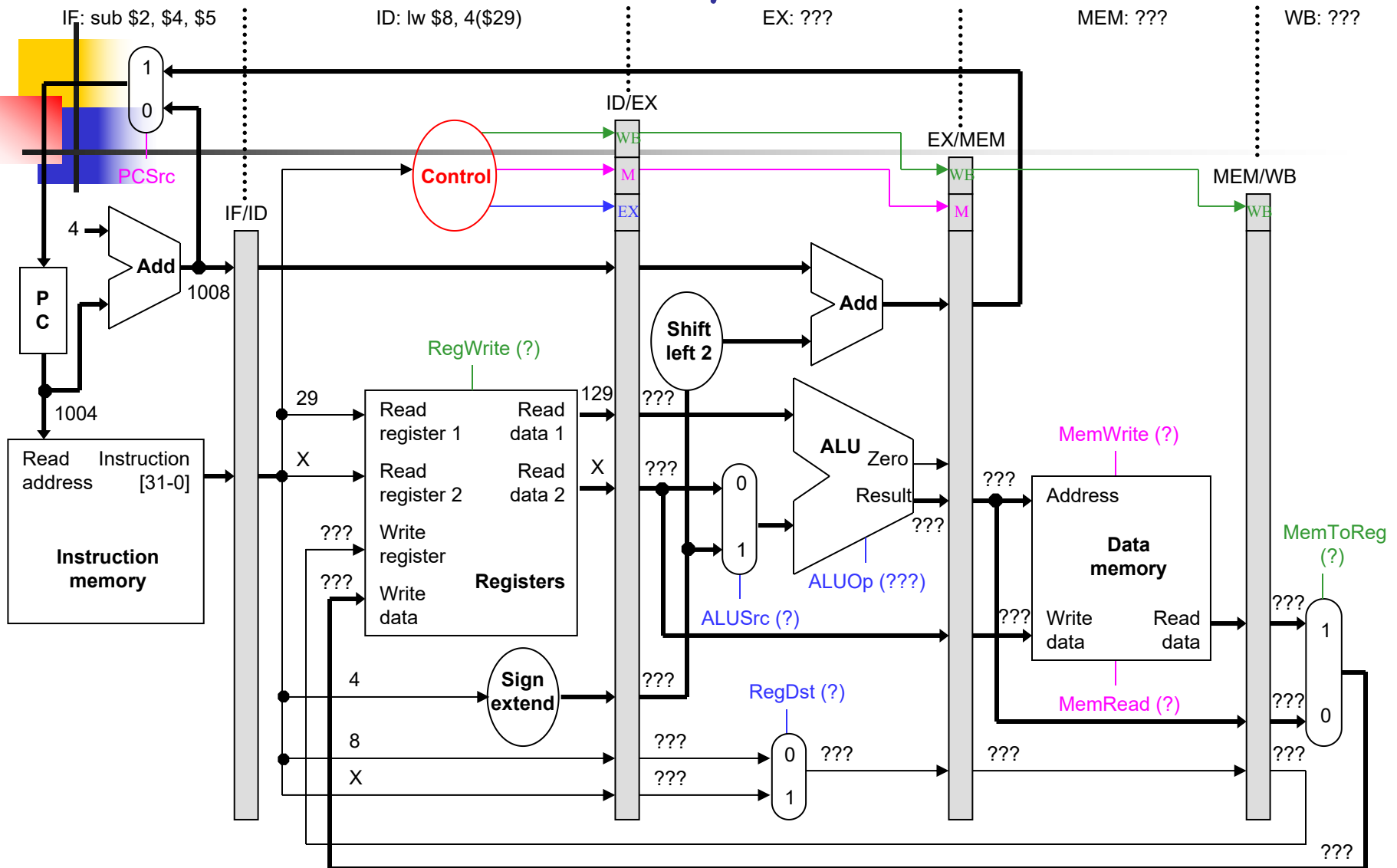
1000:	lw	\$8, 4(\$29)
1004:	sub	\$2, \$4, \$5
1008:	and	\$9, \$10, \$11
1012:	or	\$16, \$17, \$18
1016:	add	\$13, \$14, \$0

- We'll make some assumptions, just so we can show actual data values:
 - Each register contains its number plus 100. For instance, register \$8 contains 108, register \$29 contains 129, etc.
 - Every data memory location contains 99
- Our pipeline diagrams will follow some conventions:
 - An **X** indicates values that aren't important, like the constant field of an R-type instruction
 - Question marks **???** indicate values we don't know, usually resulting from instructions coming before and after the ones in our example

Cycle 1 (filling)



Cycle 2



Cycle 3

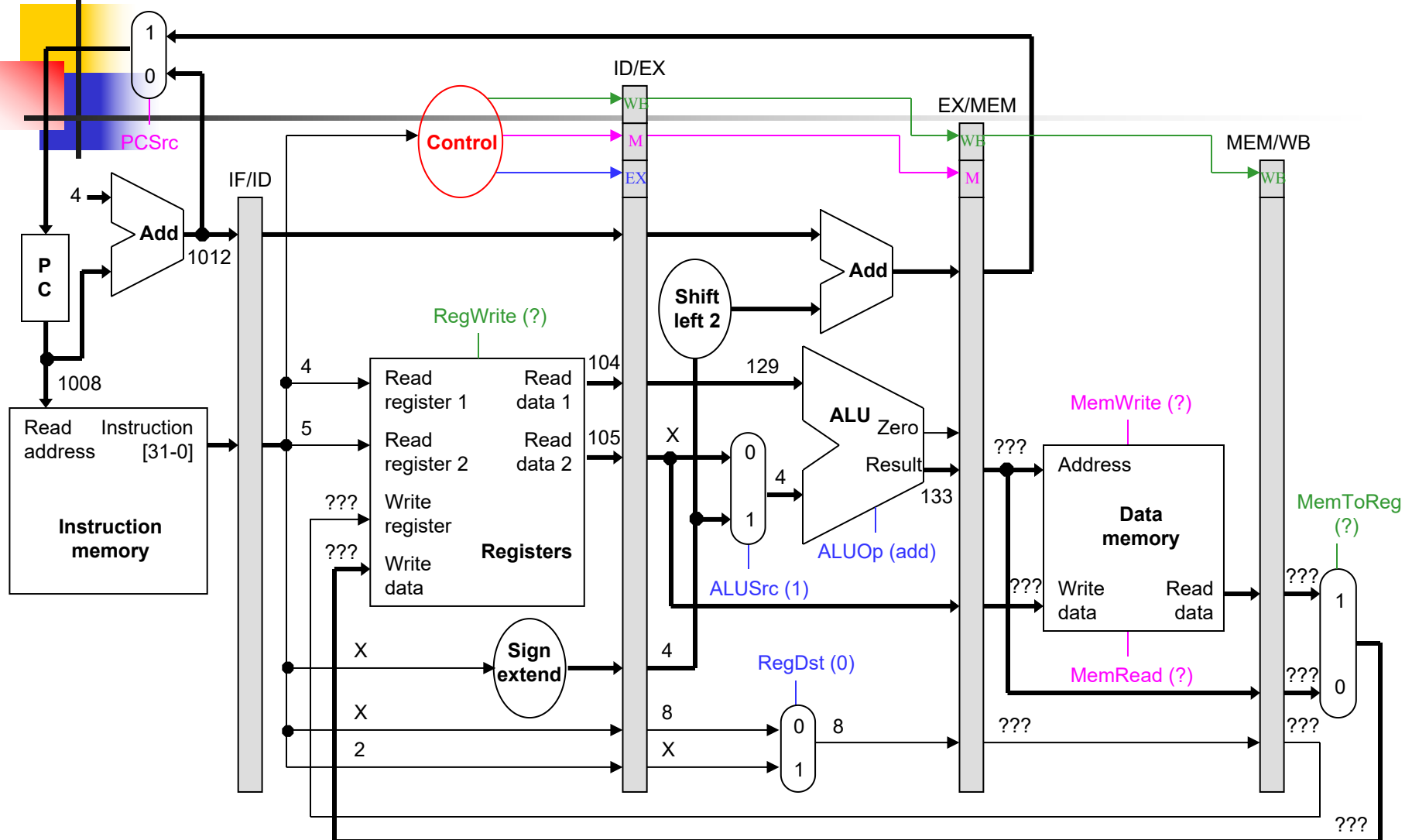
IF: and \$9, \$10, \$11

ID: sub \$2, \$4, \$5

EX: lw \$8, 4(\$29)

MEM: ???

WB: ???



Cycle 4

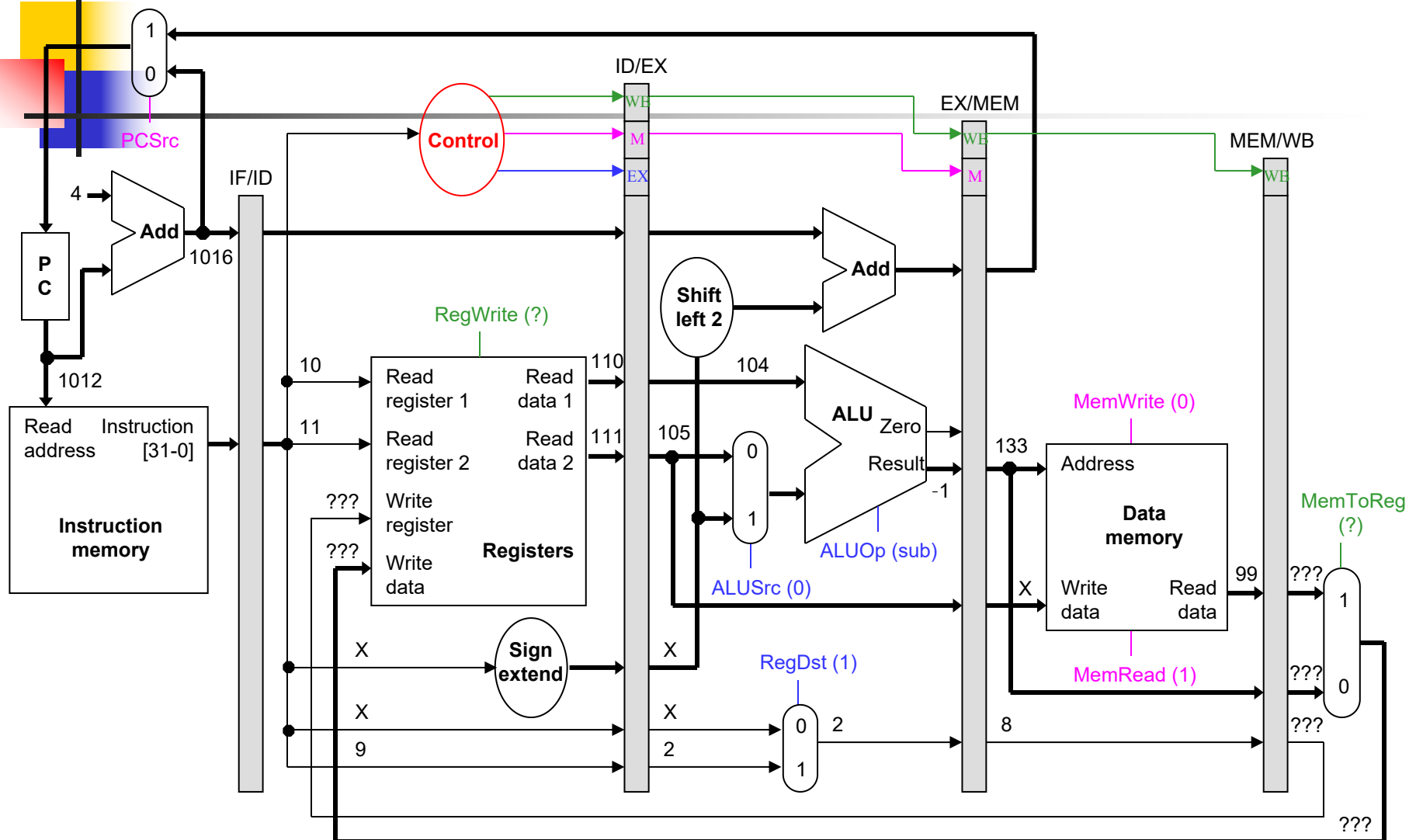
IF: or \$16, \$17, \$18

ID: and \$9, \$10, \$11

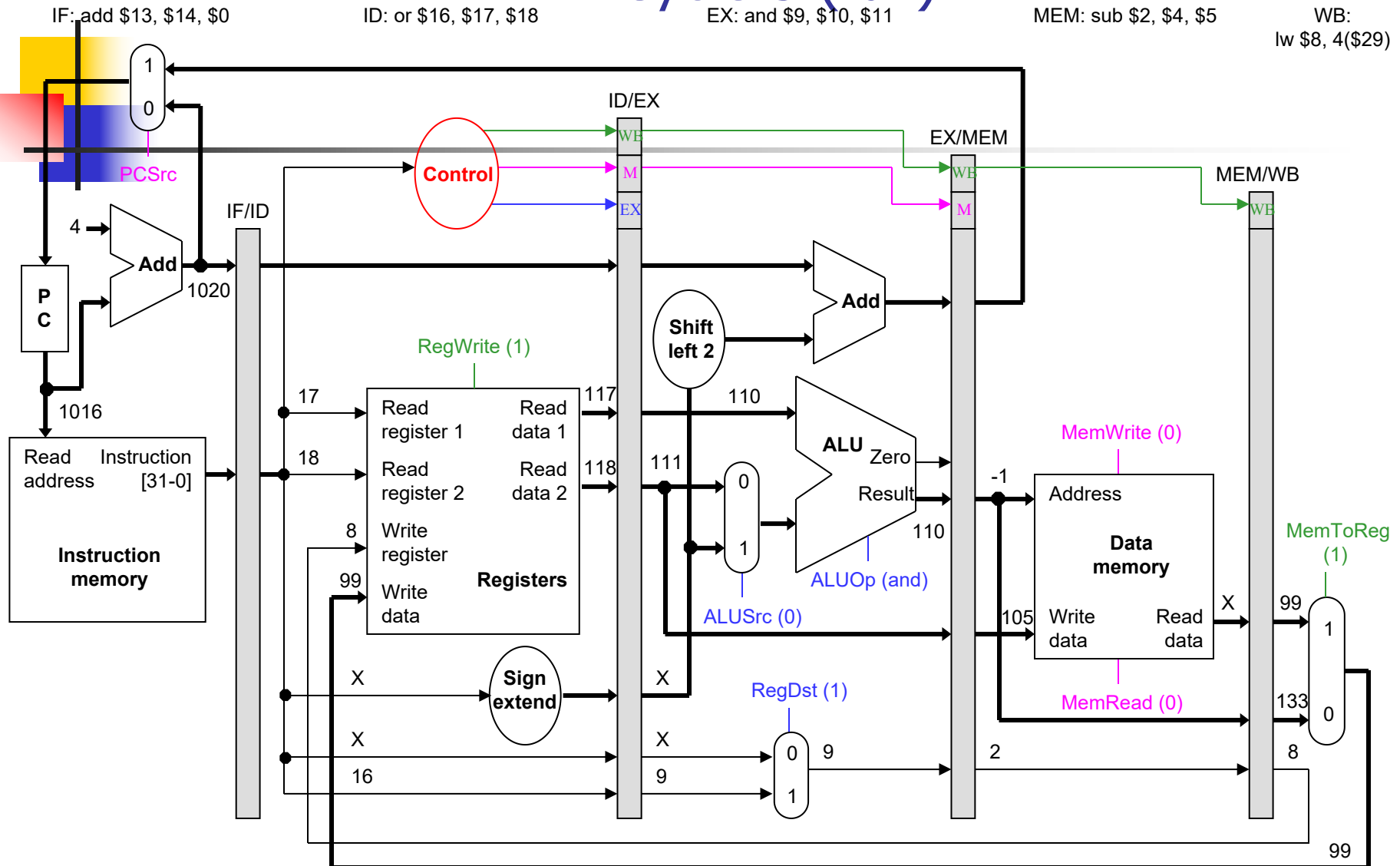
EX: sub \$2, \$4, \$5

MEM: lw \$8, 4(\$29)

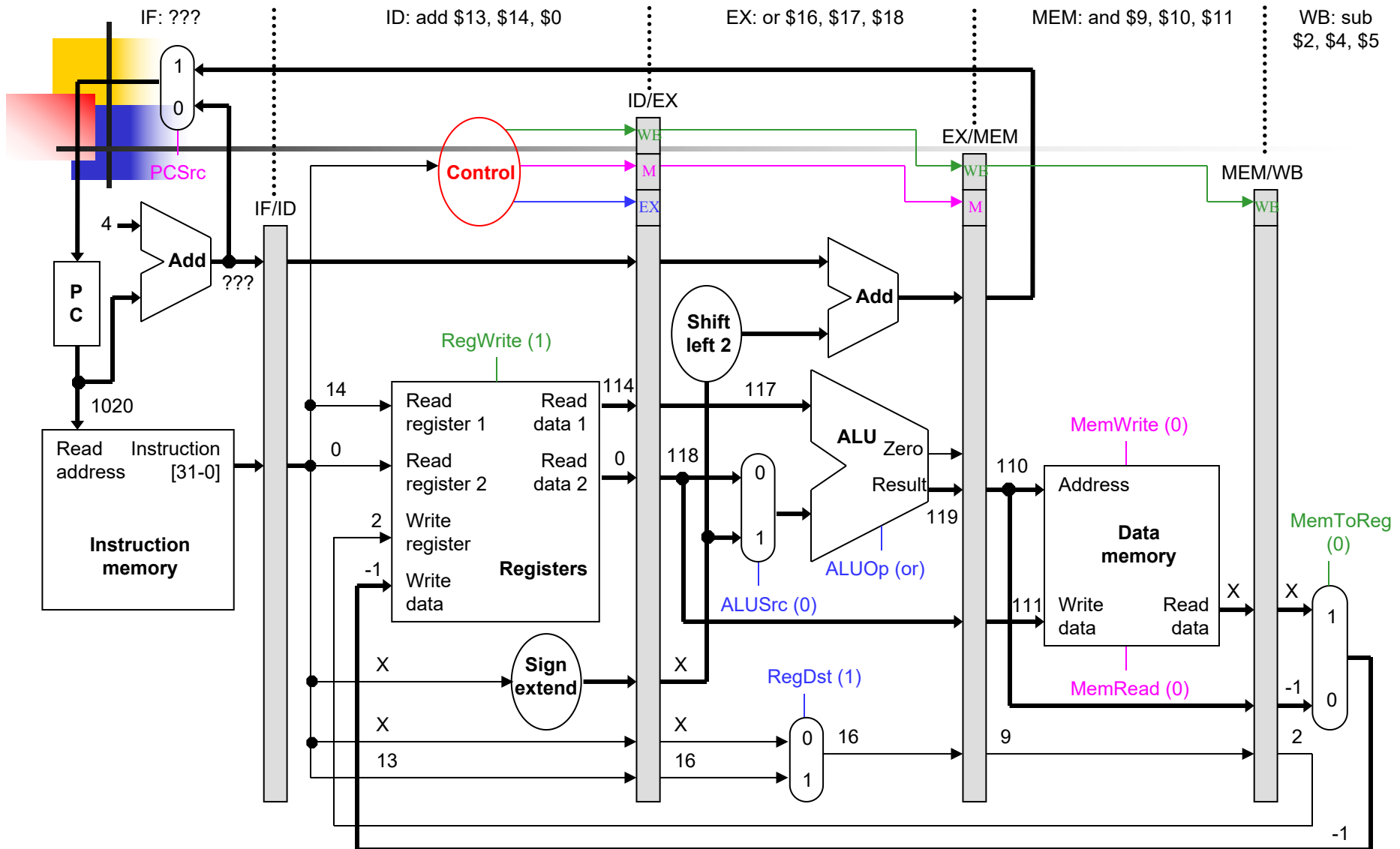
WB: ???



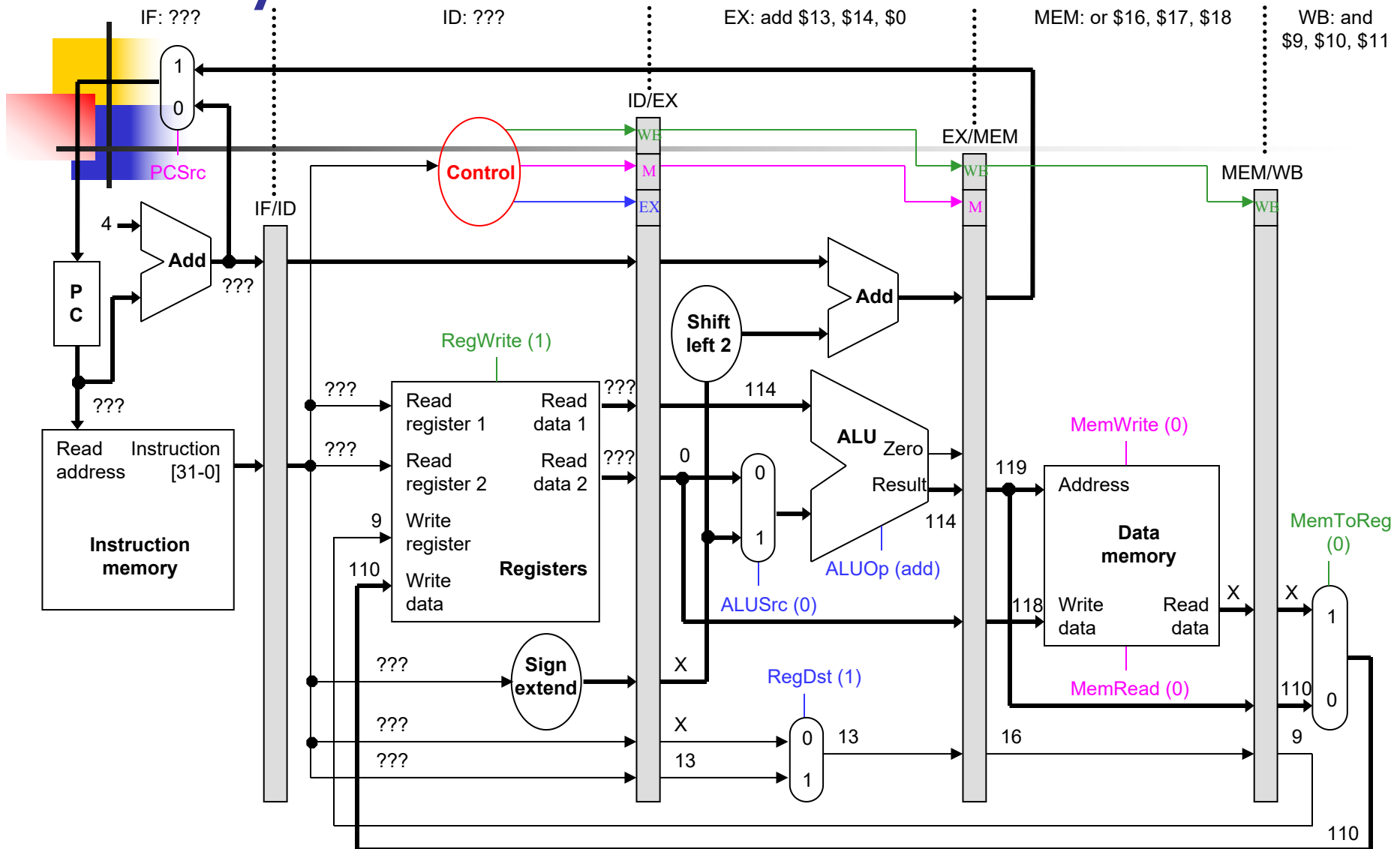
Cycle 5 (full)



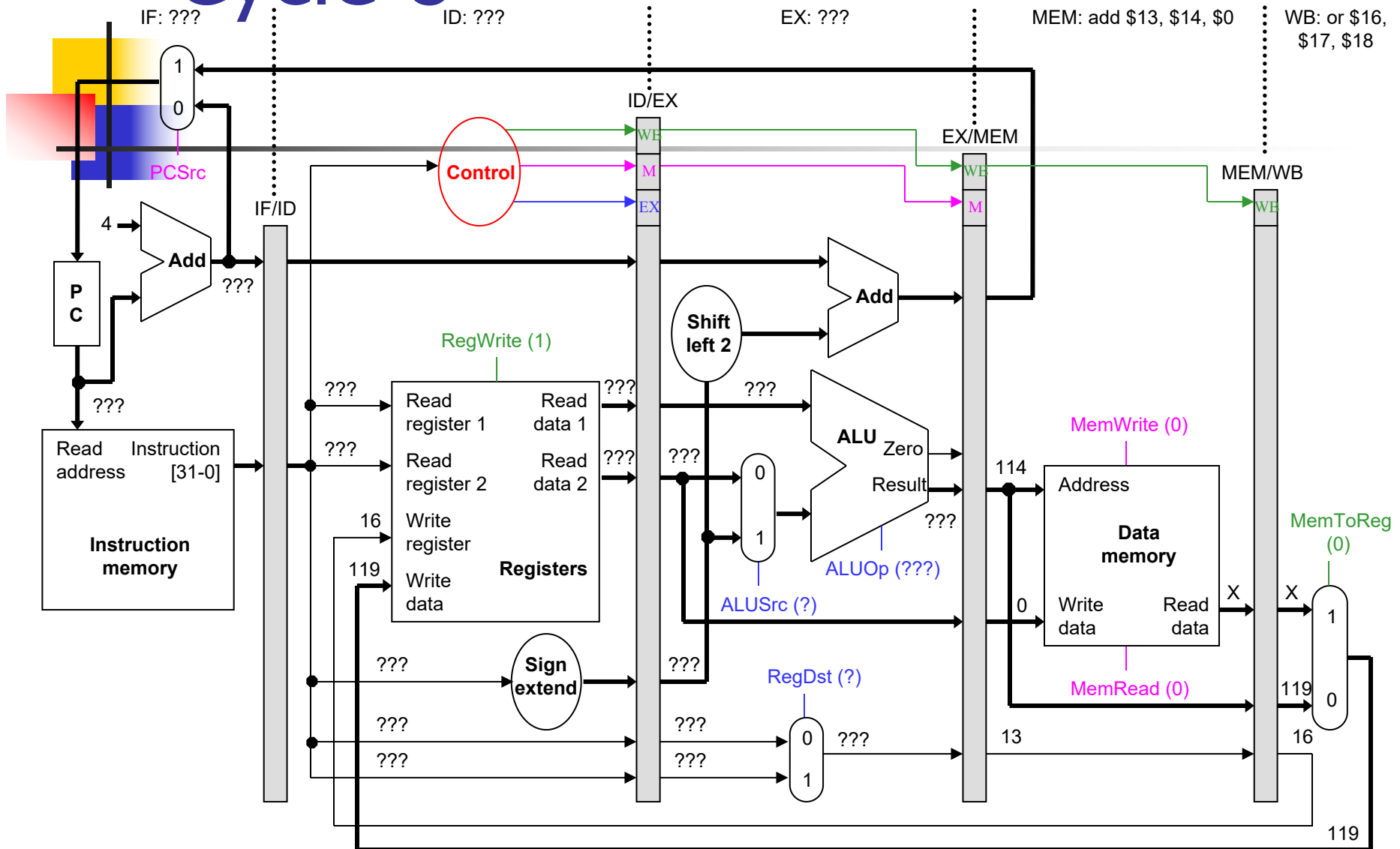
Cycle 6 (emptying)



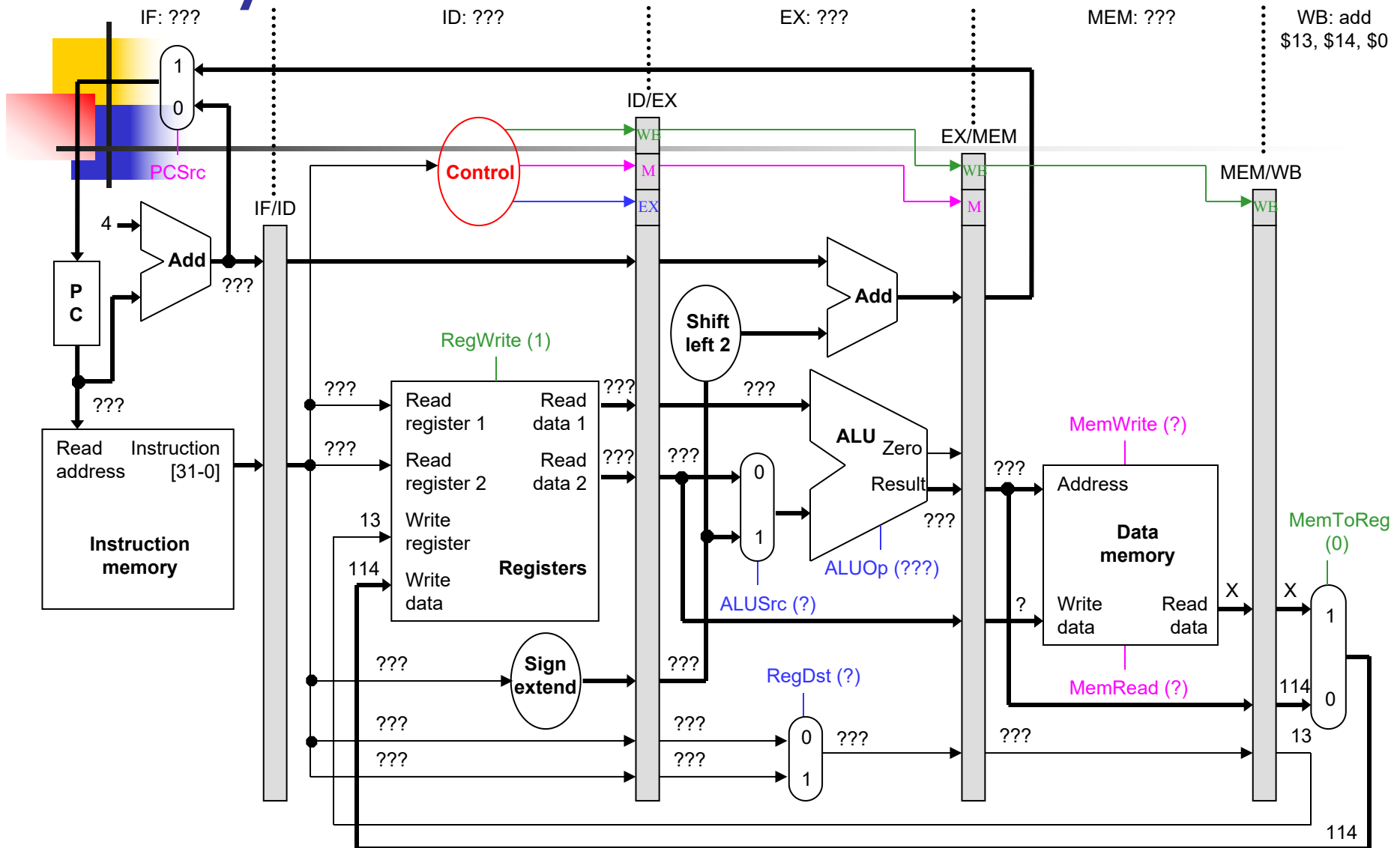
Cycle 7



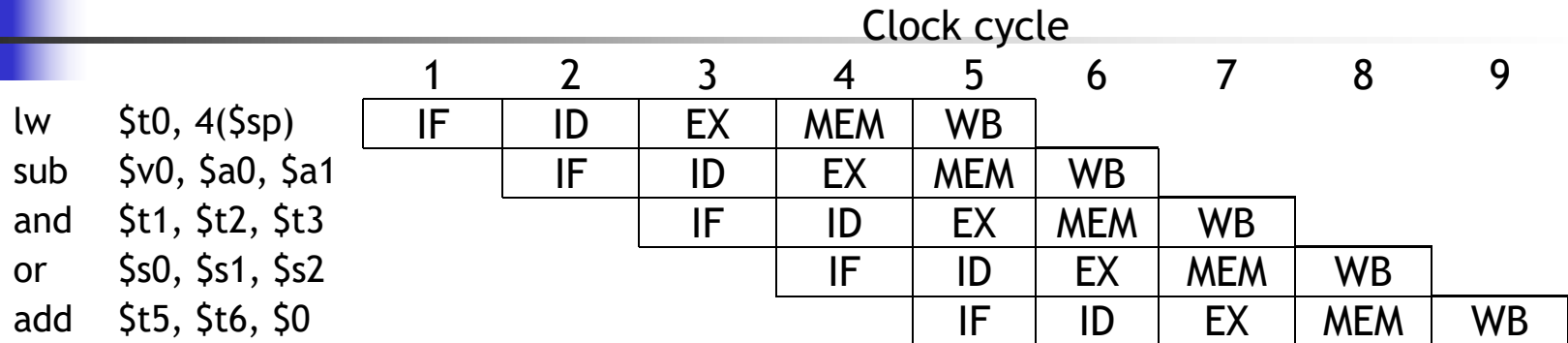
Cycle 8



Cycle 9



That's a lot of diagrams there



- Compare the last few slides with the pipeline diagram above
 - You can see how instruction executions are overlapped
 - Each functional unit is used by a *different* instruction in each cycle
 - The pipeline registers save control and data values generated in previous clock cycles for later use
 - When the pipeline is full in clock cycle 5, all of the hardware units are utilized. This is the ideal situation, and what makes pipelined processors so fast

Note how everything goes left to right, except ...

