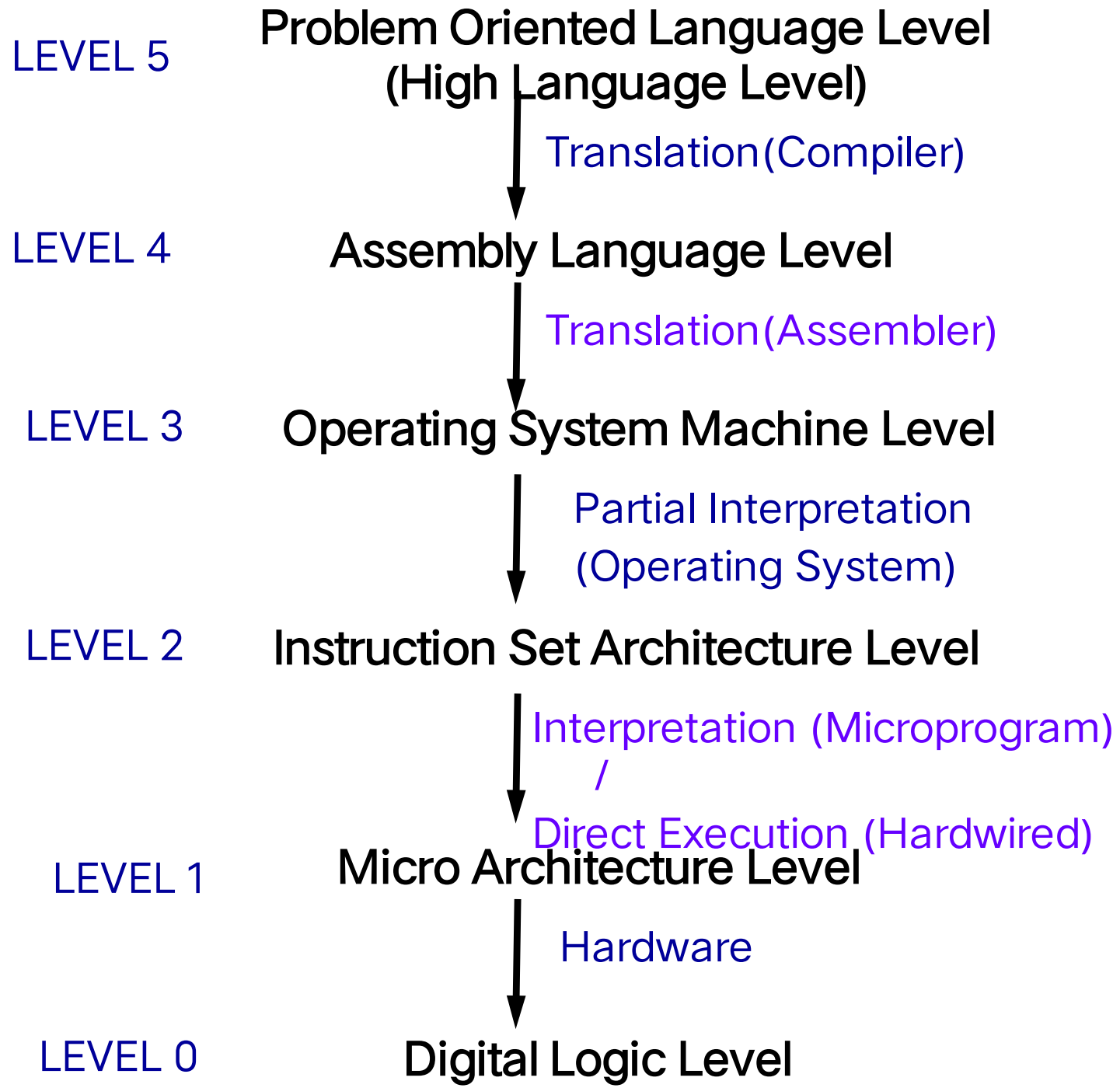
Computer Organization

Micro Architecture Level
The Processor: Datapath and Control

Six Level Computer



GOAL: Datapath and control unit

- Design a computer architecture
  - To support the defined ISA
  - To fetch and execute each instruction
- Study of
  - Processor designs
    - Single cycle
    - Multiple cycle
    - Pipelined

GOAL: Datapath and control unit

- Study of
  - Datapath
    - Basic components
    - How a datapath is constructed?
    - How the speed of the CPU is determined?
  - Control unit
    - Why do we need control unit?
    - How control unit relates instructions and datapath?
    - Control unit design

Instruction Set Architecture (ISA)

- A programmable system uses
  - A sequence of *instructions* to control its operation
- A typical instruction specifies:
  - Operation to be performed
  - Operands to use
  - Where to place the result
  - Which instruction to execute next

Instruction Set Architecture (ISA) (cont.)

- Instructions are stored in
  - RAM or ROM as a *program*
- The addresses for instructions in a computer are provided by
  - A *program counter (PC)* that can
    - Count up
    - Load a new address based on an instruction and, optionally, status information

Instruction Set Architecture (ISA) (cont.)

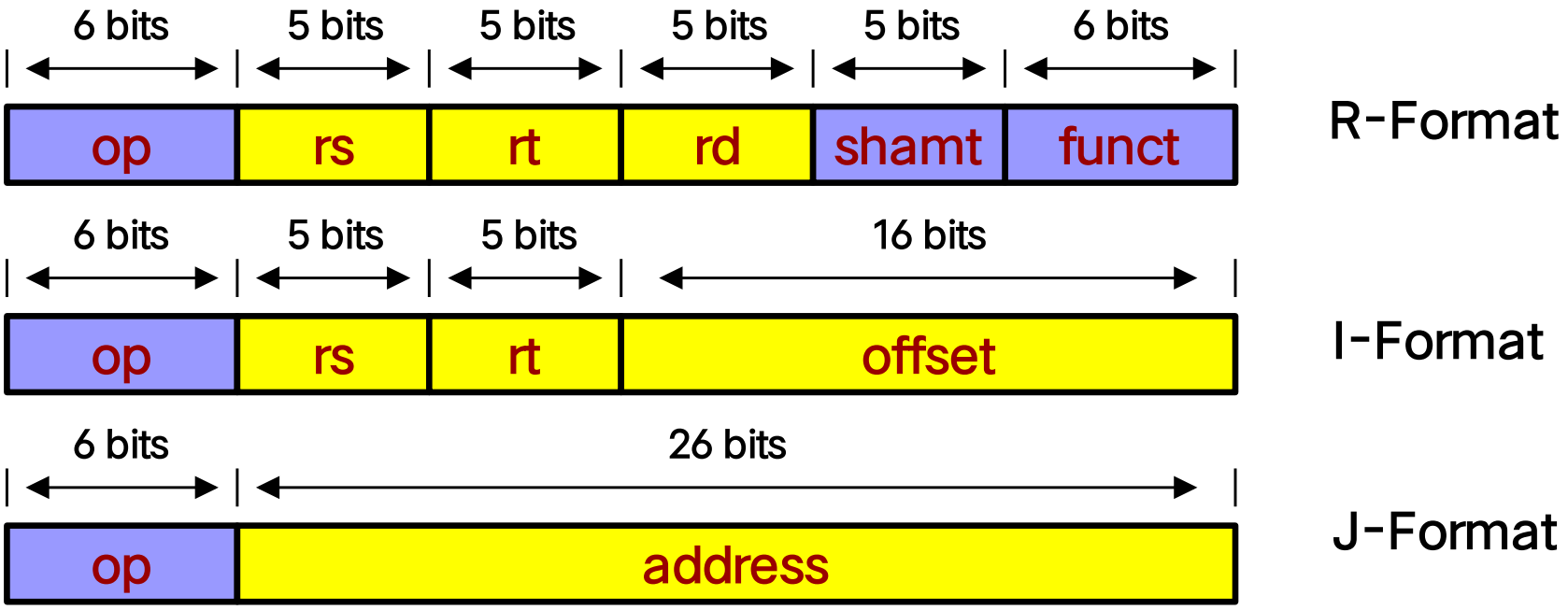
- Part of the Control Unit to execute an instruction
  - The PC and associated control logic
- Executing an instruction –
  - **Activating the necessary sequence of operations specified by the instruction**
  - Controlled by the control unit and performed,
    - **In the datapath**
    - **In the control unit**
    - **In external hardware such as memory or input/ output**

ISA: Instruction Format

- A instruction consists of a bit vector
  - The *fields* of an instruction are subvectors representing specific functions and having specific binary codes defined
  - The *format* of an instruction defines the subvectors and their function
- An ISA usually contains multiple formats

MIPS Instruction & instuction Formats

- Simplified only for
  - Arithmetic-logic instructions : add, sub, and, or, slt
  - Memory-reference instructions : lw, sw
  - Control-flow instructions : beq, j



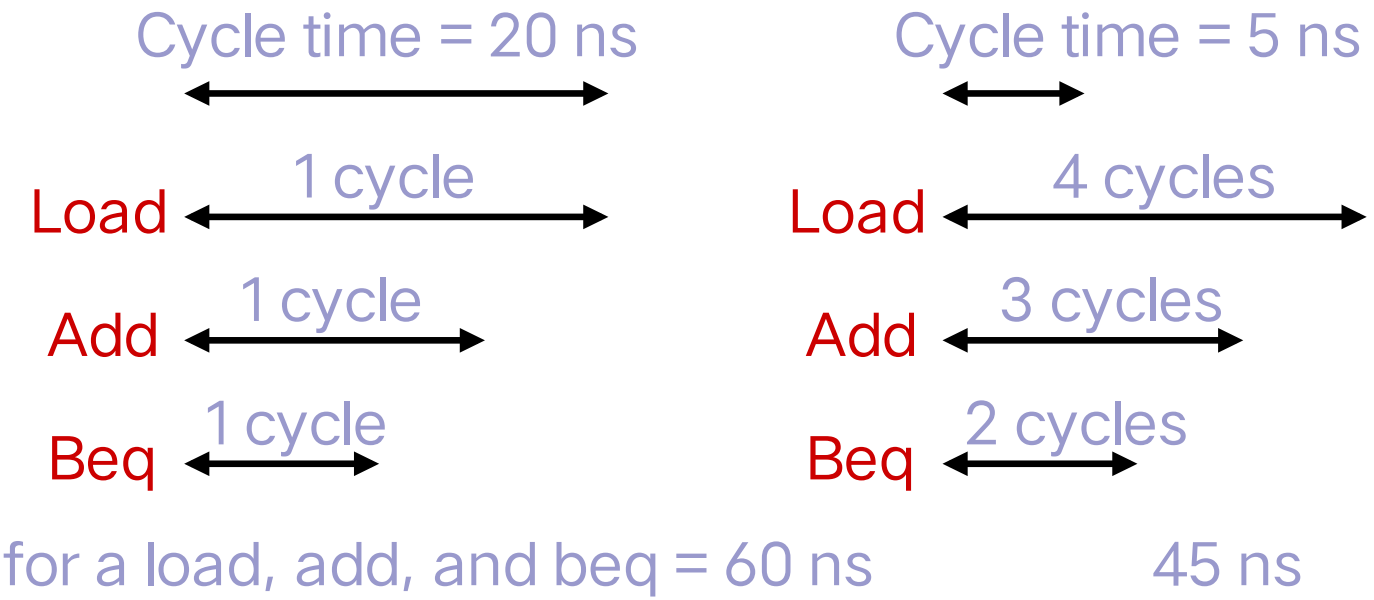
Implementing MIPS: the Fetch/Execute Cycle

- High-level abstract view of *fetch/execute* implementation
  - use the program counter (PC) to read instruction address
  - *fetch* the instruction from memory and increment PC
  - use fields of the instruction to select registers to read
  - *execute* depending on the instruction
  - repeat…

Processor Implementation Styles

- Single Cycle
  - Perform each instruction in 1 clock cycle
  - Clock cycle must be long enough for slowest instruction
- Multi-Cycle
  - Break fetch/execute cycle into multiple steps
  - Perform 1 step in each clock cycle
- Pipelined

Single Vs. Multi-Cycle Processor Machine



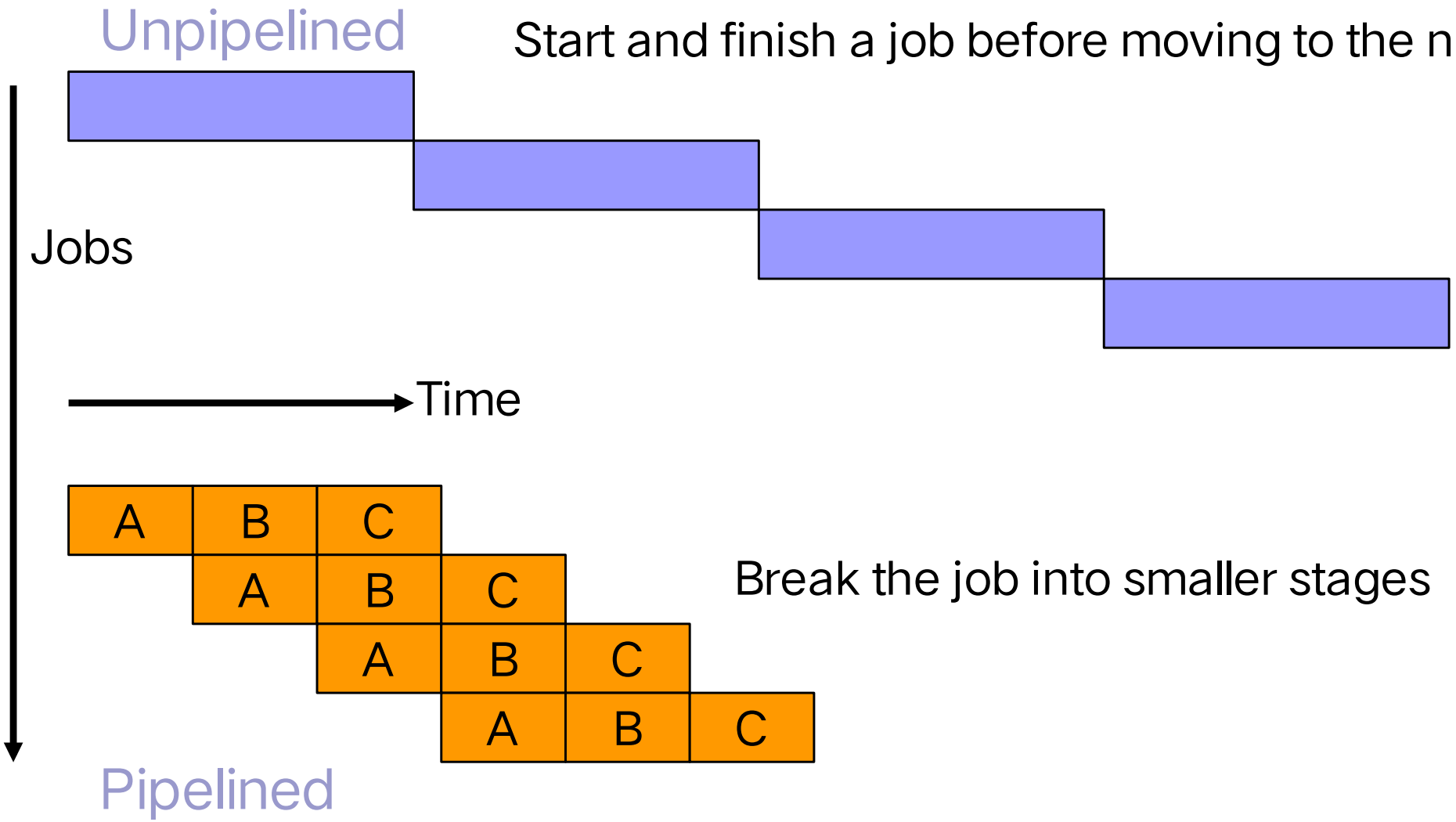
- In the Single cycle implementation, every instruction requires one cycle to complete, so cycle time = time taken for the slowest instruction

- If the execution was broken into multiple (faster) cycles, the shorter instructions can finish sooner

Processor Implementation Styles

- Single Cycle
  - Perform each instruction in 1 clock cycle
  - Clock cycle must be long enough for slowest instruction
  - Disadvantage: only as fast as the 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

Processor Implementation Styles



Processor Implementation Styles

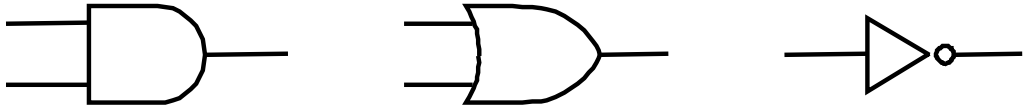
- Single Cycle
  - Perform each instruction in 1 clock cycle
  - Clock cycle must be long enough for slowest instruction
  - Disadvantage: only as fast as the slowest instruction
- Multi-Cycle
  - Break fetch/execute cycle into multiple steps
  - Perform 1 step in each clock cycle
  - Advantage: each 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

Functional Elements

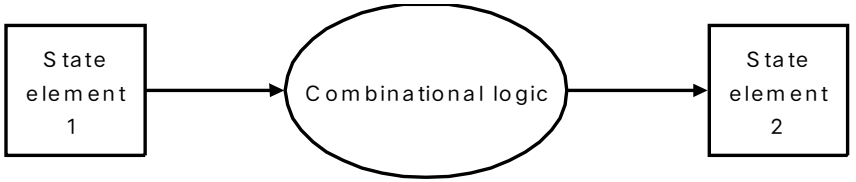
- Two types of functional elements in the hardware:
  - Elements that *operate on* data
    - Called combinational elements
  - Elements that *contain* data
    - Called state or sequential elements

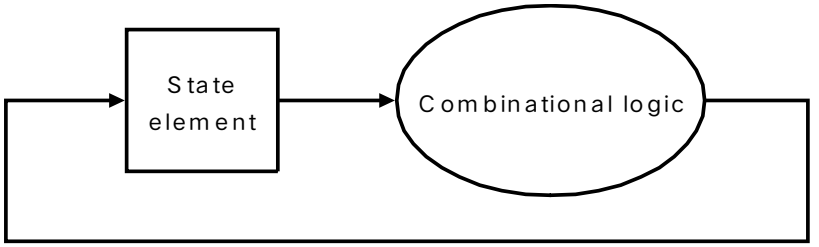
Combinational Elements

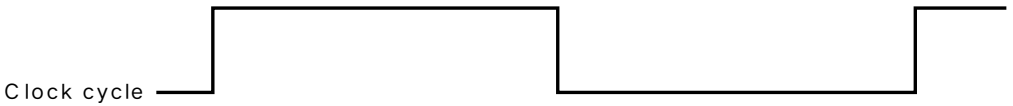
- Works as an *input* ⇒ *output function*, e.g., ALU
- Combinational logic *reads input data from one register and writes output data to another*, or same, register
  - *read/write happens in a single cycle* – combinational element *cannot store data* from one cycle to a future one



Combinational logic hardware units





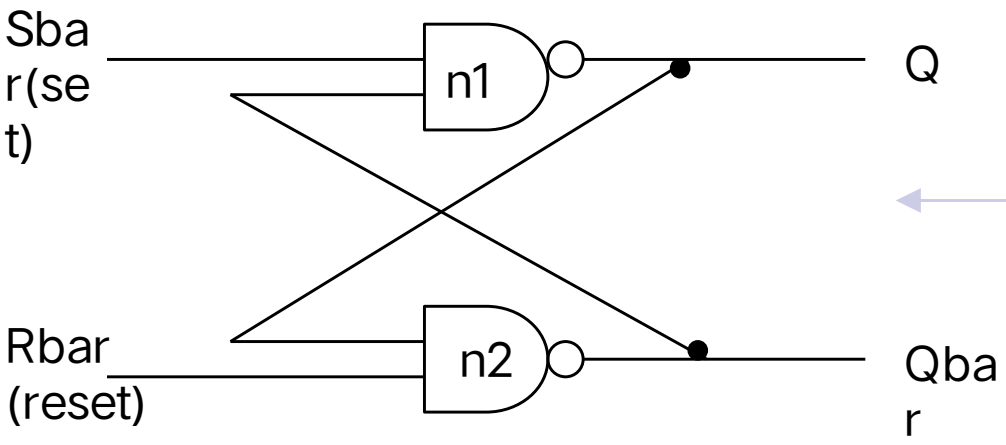


State Elements

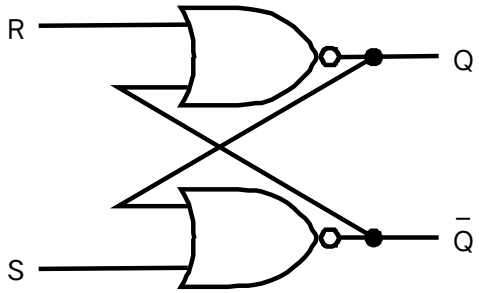
- Contain *data* in internal storage, e.g., *registers* and *memory*
- All state elements together *define* the *state of the machine*
- *Flipflops* and *latches* are 1-bit state elements, equivalently, they are *1-bit memories*
- The *output*(s) of a flipflop or latch *always* depends on the bit value 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…

Set-Reset (SR-) latch (unclocked)

Think of Sbar as S, the inverse of *set* (which sets Q to 1), and Rbar as R, the inverse of *reset*



equivalently with nor gates



Basic memory unit

Made from two cross-coupled *nand* gates

When both Sbar and Rbar are 1, then either *one of the following two* states is *stable*:

a) Q = 1 & Qbar = 0

b) Q = 0 & Qbar = 1

and the latch will *continue* in the current stable state.

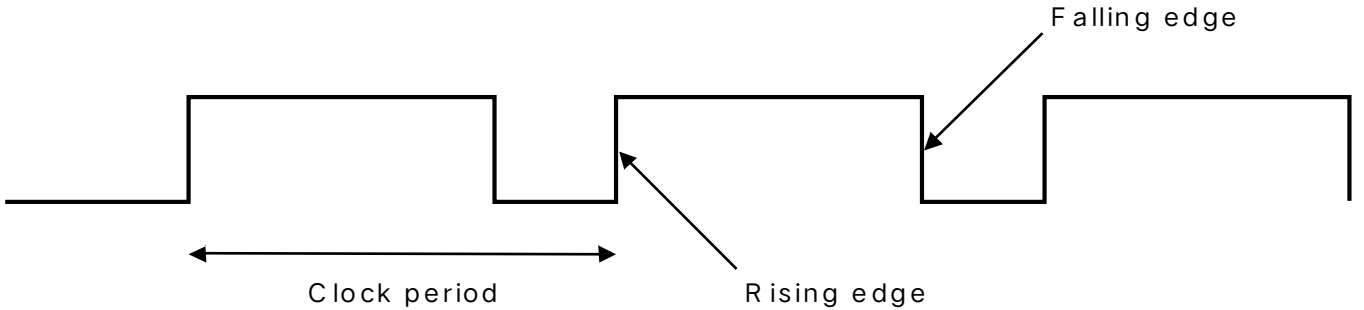
If Sbar changes to 0 (while Rbar remains at 1), then the latch is forced to the *exactly one* possible stable state (a). If Rbar changes to 0 (while Sbar remains at 1), the latch is forced to the *exactly one* possible stable state (b).

So, the latch *remembers* which of Sbar or Rbar was last 0 *during* the time they are both 1.

When both Sbar and Rbar are 0 the *exactly one* stable state is Q = Qbar = 1. However, if after that both Sbar and Rbar return to 1, the latch may then *jump non-deterministically* to one of stable states (a) or (b), which is undesirable behavior

Synchronous Logic: Clocked Latches and Flipflops

- Clocks are used in *synchronous* logic to determine *when* a state element is to be updated
  - in *level-triggered* clocking methodology either the state changes only when the clock is high or only when it is low (technology-dependent)

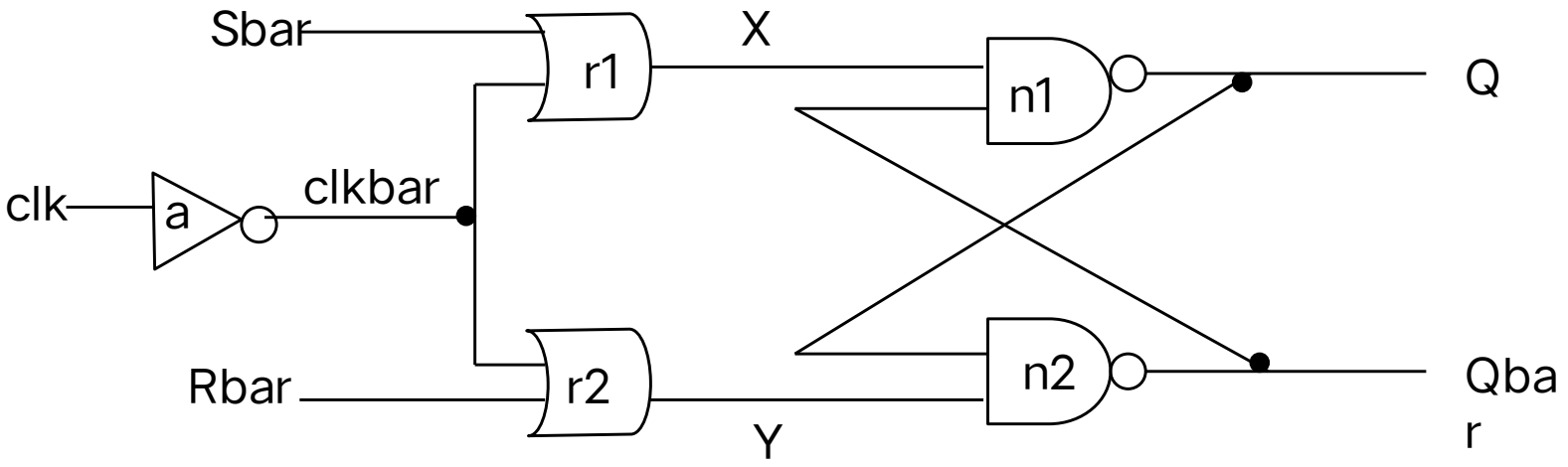


  - in *edge-triggered* clocking methodology either the *rising edge* or *falling edge* is active (depending on technology) – i.e., states change only on rising edges or only on falling edge

- Latches are level-triggered
- Flipflops are edge-triggered

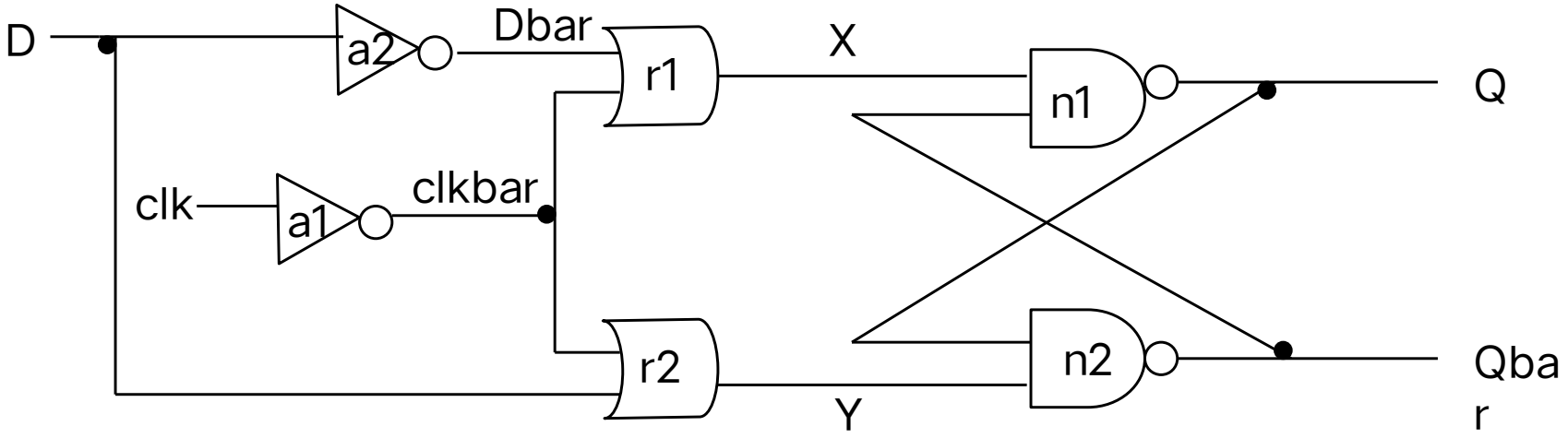
Clocked SR-latch

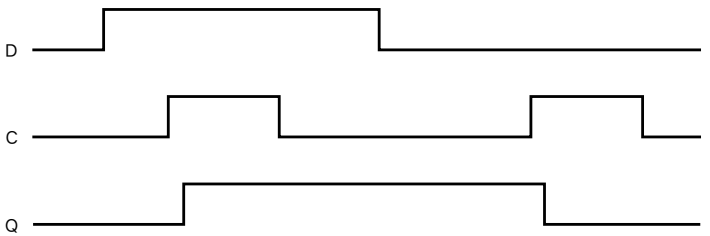
- State can change only when clock is *high*
- *Potential problem* : both inputs Sbar = 0 & Rbar = 0 will cause non-deterministic behavior



Clocked D-latch

- State can change only when clock is *high*
- Only *single* data input (compare SR-latch)
- *No problem* with non-deterministic behavior

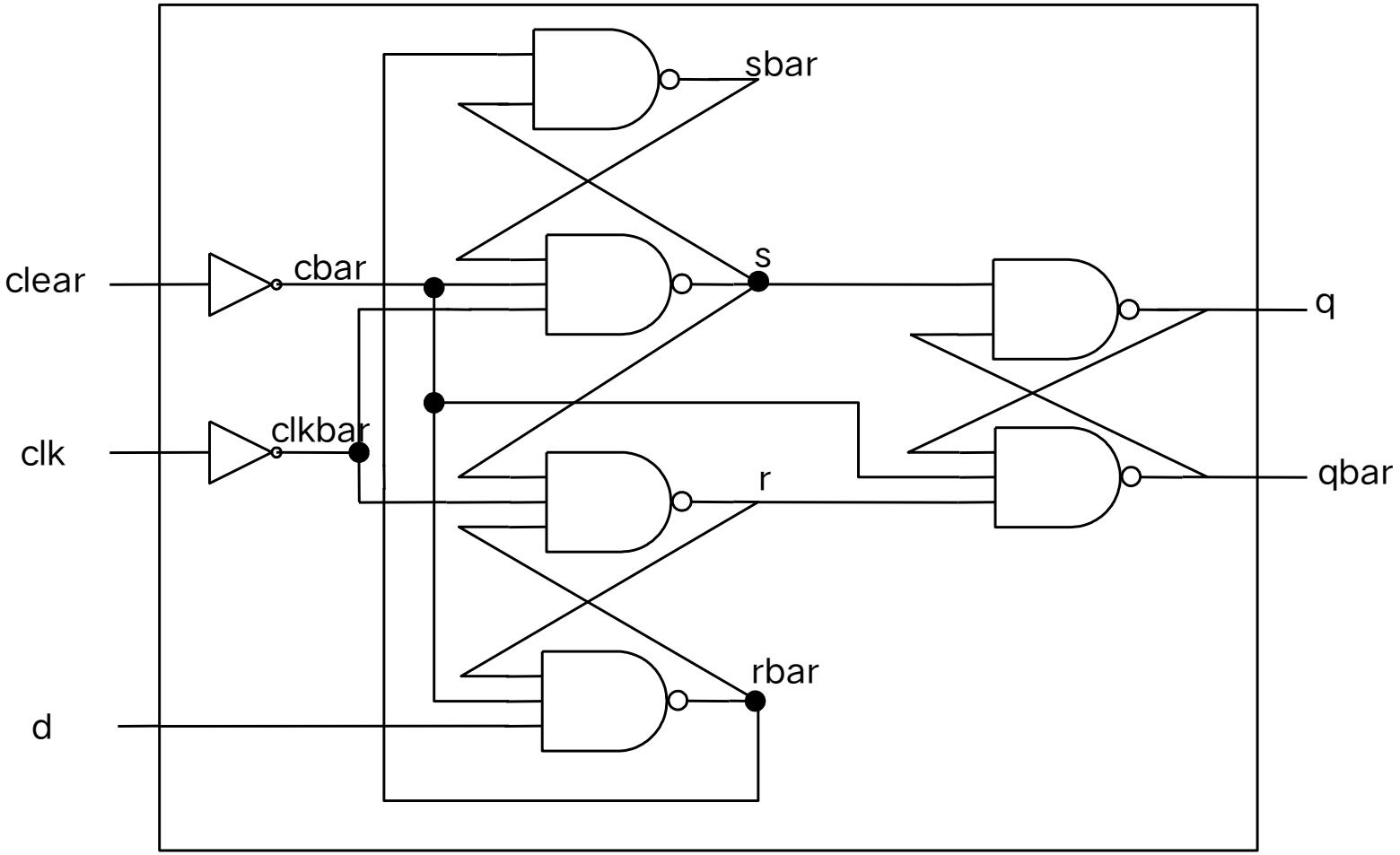




Timing diagram of D-latch

Clocked D-flipflop

- *Negative edge-triggered*
- Made from *three SR-latches*

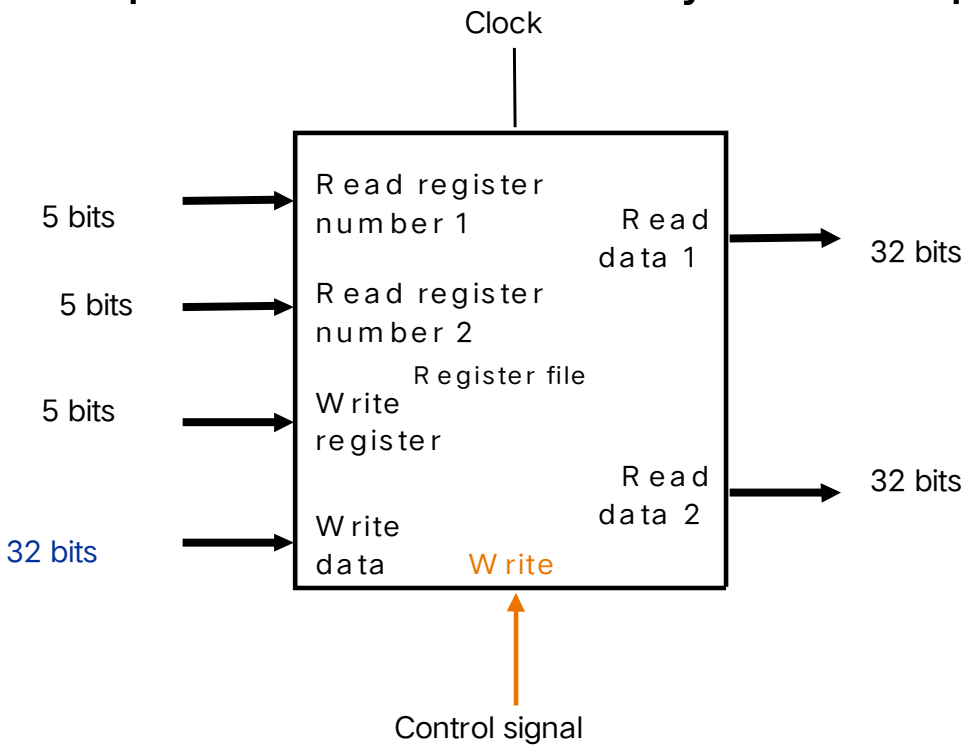


Logi Sim

- All components that we have discussed – and shall discuss – can be fabricated using Logi Sim

State Elements on the Datapath: Register File

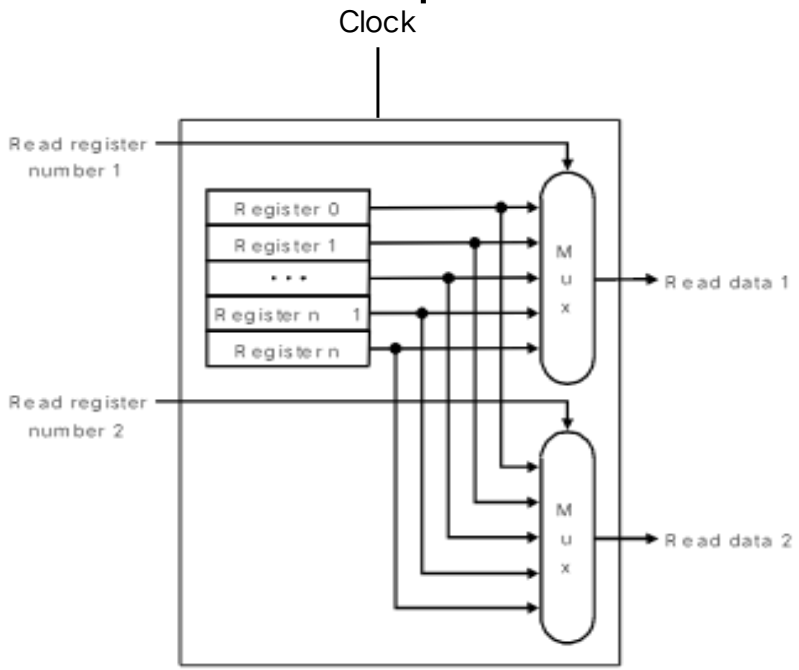
- Registers are implemented with arrays of D-flipflops



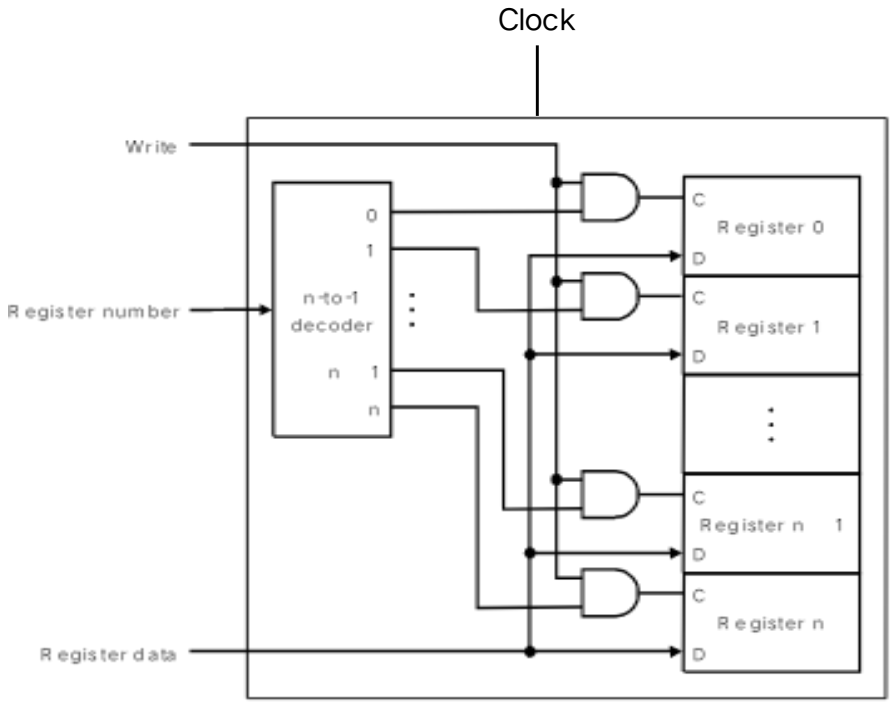
Register file with two read ports and one write port

State Elements on the Datapath: Register File

- Port implementation:



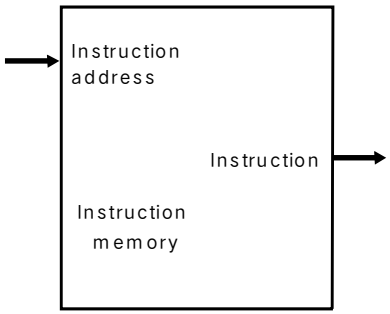
Read ports are implemented with a pair of multiplexors – 5 bit multiplexors for 32 registers



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

Datapath: Instruction Store/Fetch & PC Increment

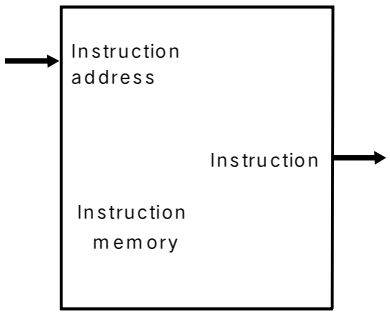
Three elements used to store and fetch instructions and increment the PC



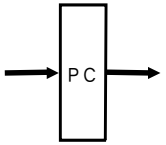
a. Instruction memory

Datapath: Instruction Store/Fetch & PC Increment

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



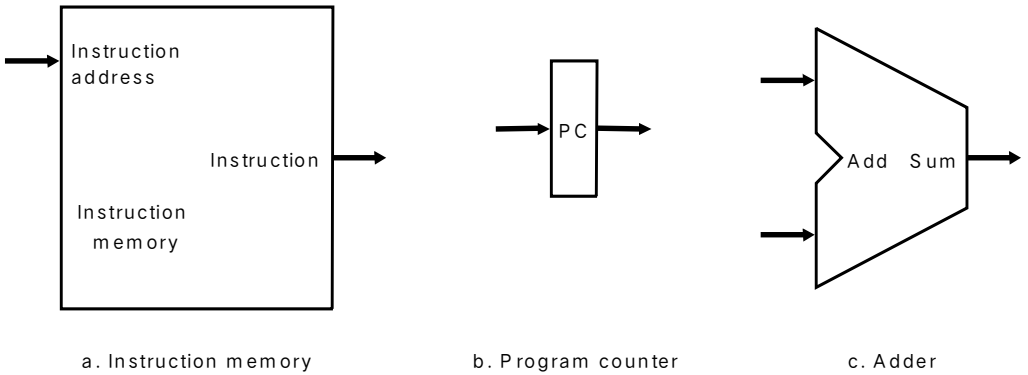
a. Instruction memory



b. Program counter

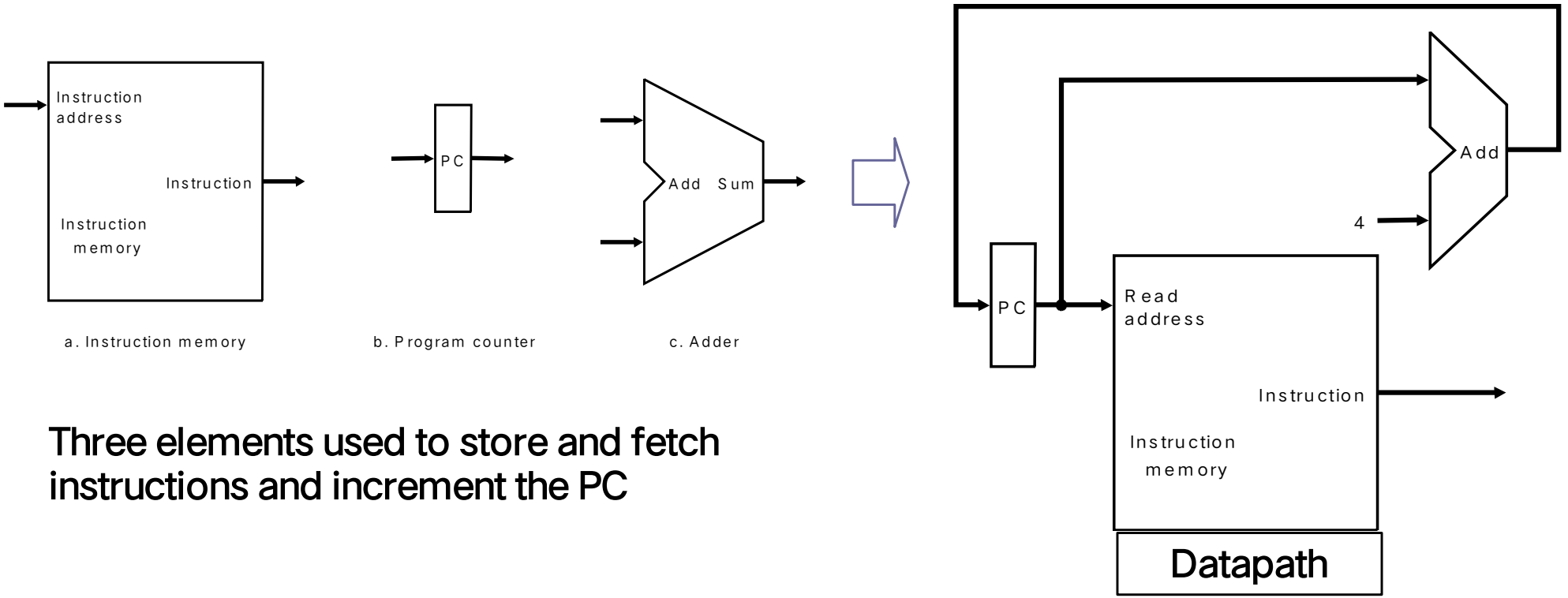
Datapath: Instruction Store/Fetch & PC Increment

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



a. Instruction memory b. Program counter c. Adder

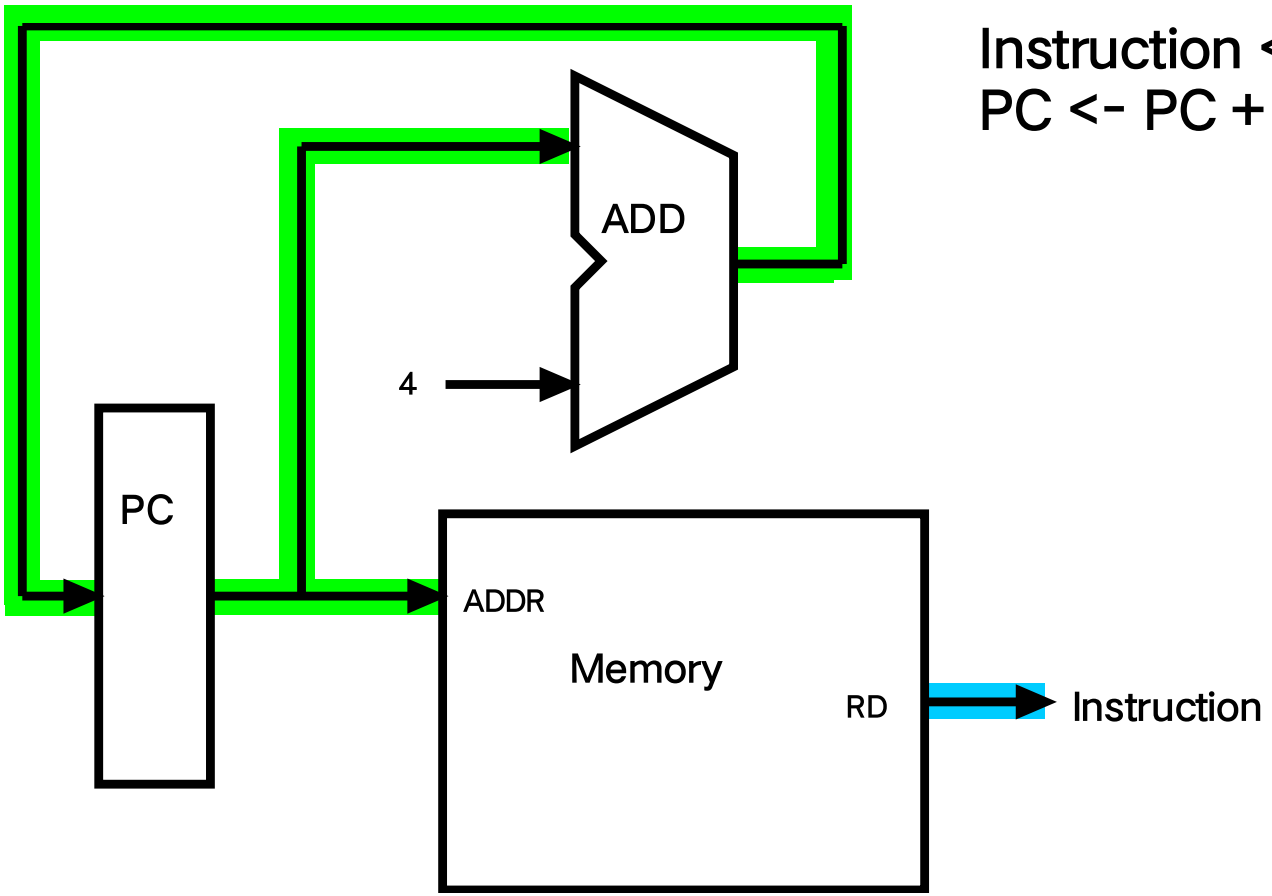
Datapath: Instruction Store/Fetch & PC Increment



a. Instruction memory b. Program counter c. Adder

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

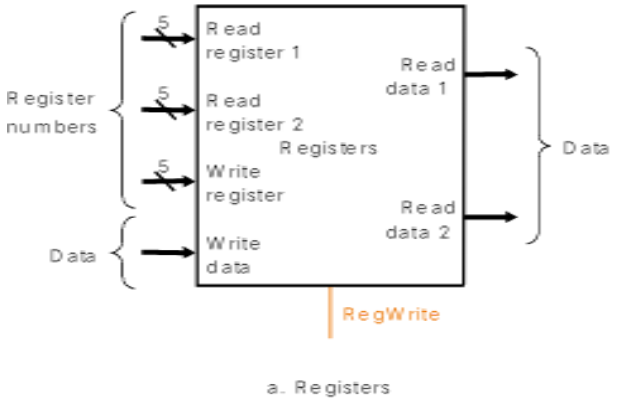
Animating the Datapath



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

Datapath: R-Type Instruction

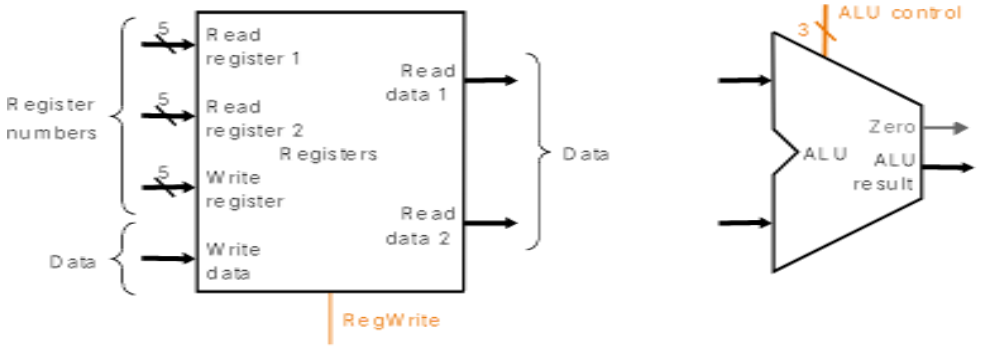
Two elements used to implement R-type instructions



a. Registers

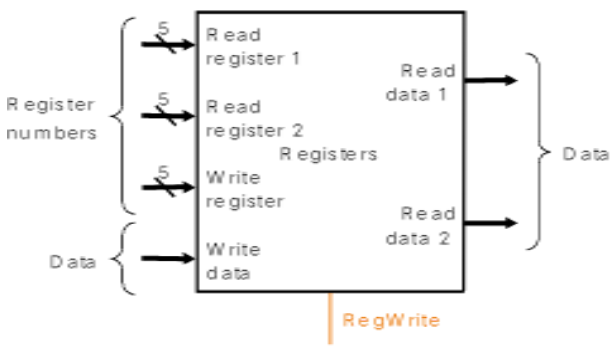
Datapath: R-Type Instruction

Two elements used to implement R-type instructions

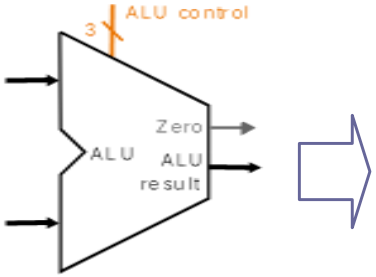


Datapath: R-Type Instruction

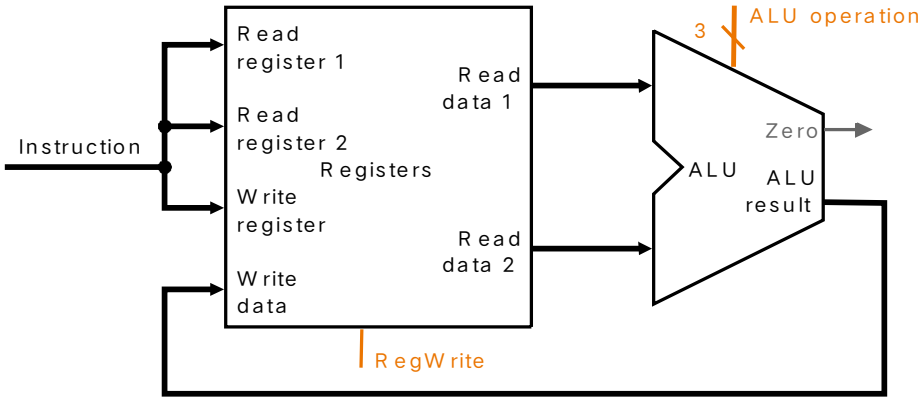
Two elements used to implement R-type instructions



a. Registers



b. ALU

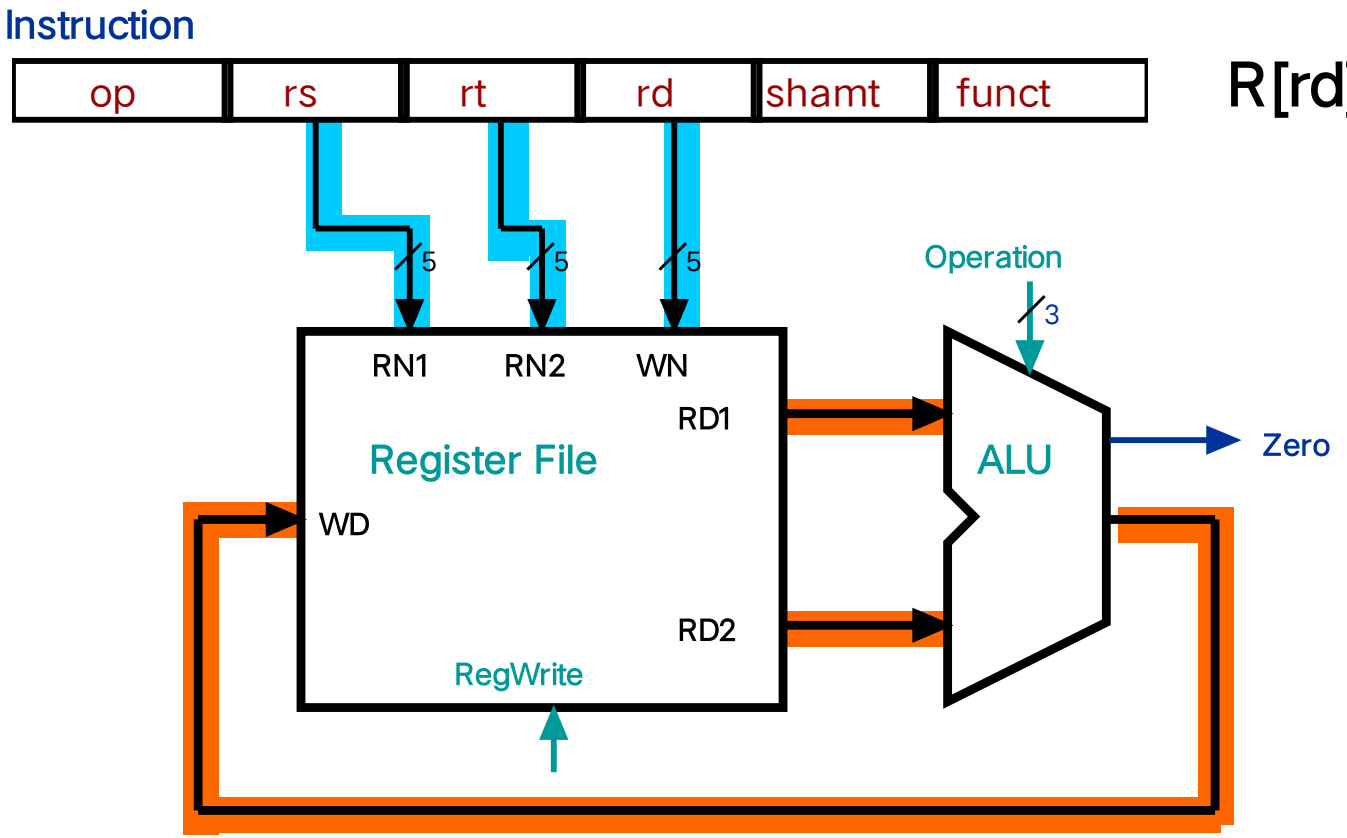


Datapath

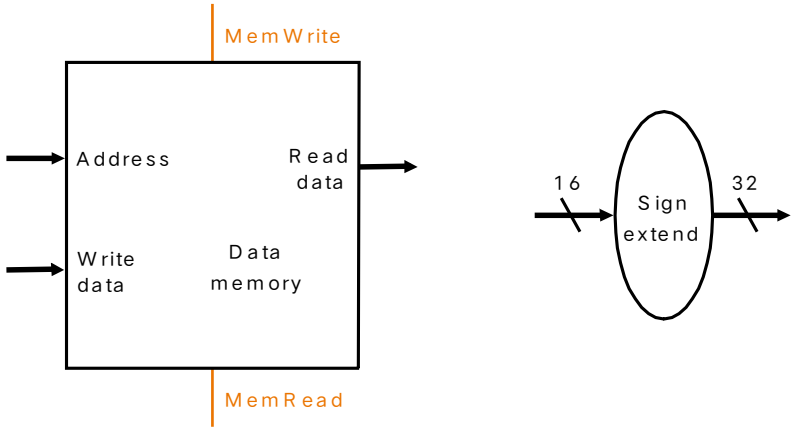
Animating the Datapath

add rd, rs, rt

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



Datapath: Load/Store Instruction



a. Data memory unit

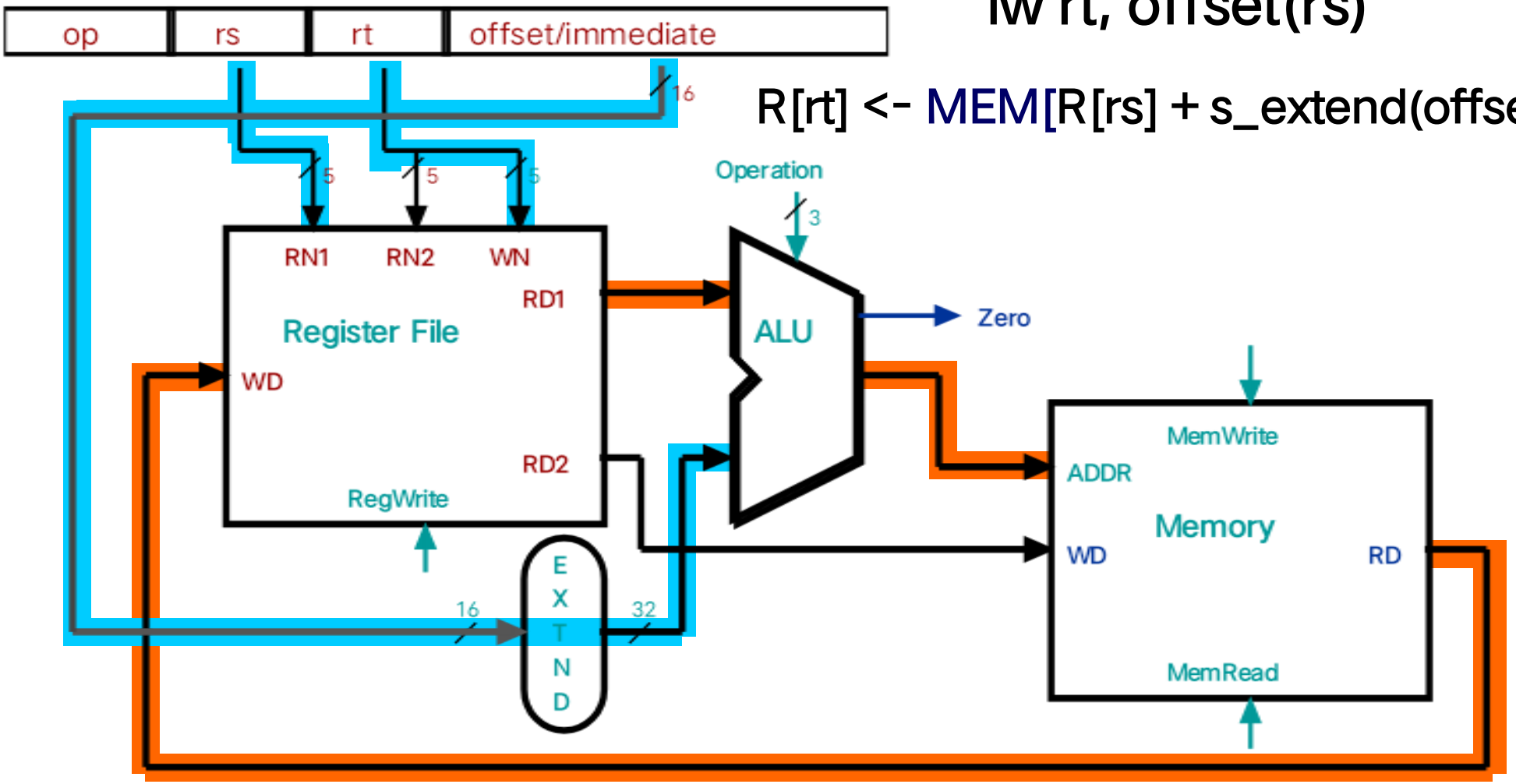
b. Sign-extension unit

Two additional elements used
To implement load/stores

Animating the Datapath

lw rt, offset(rs)

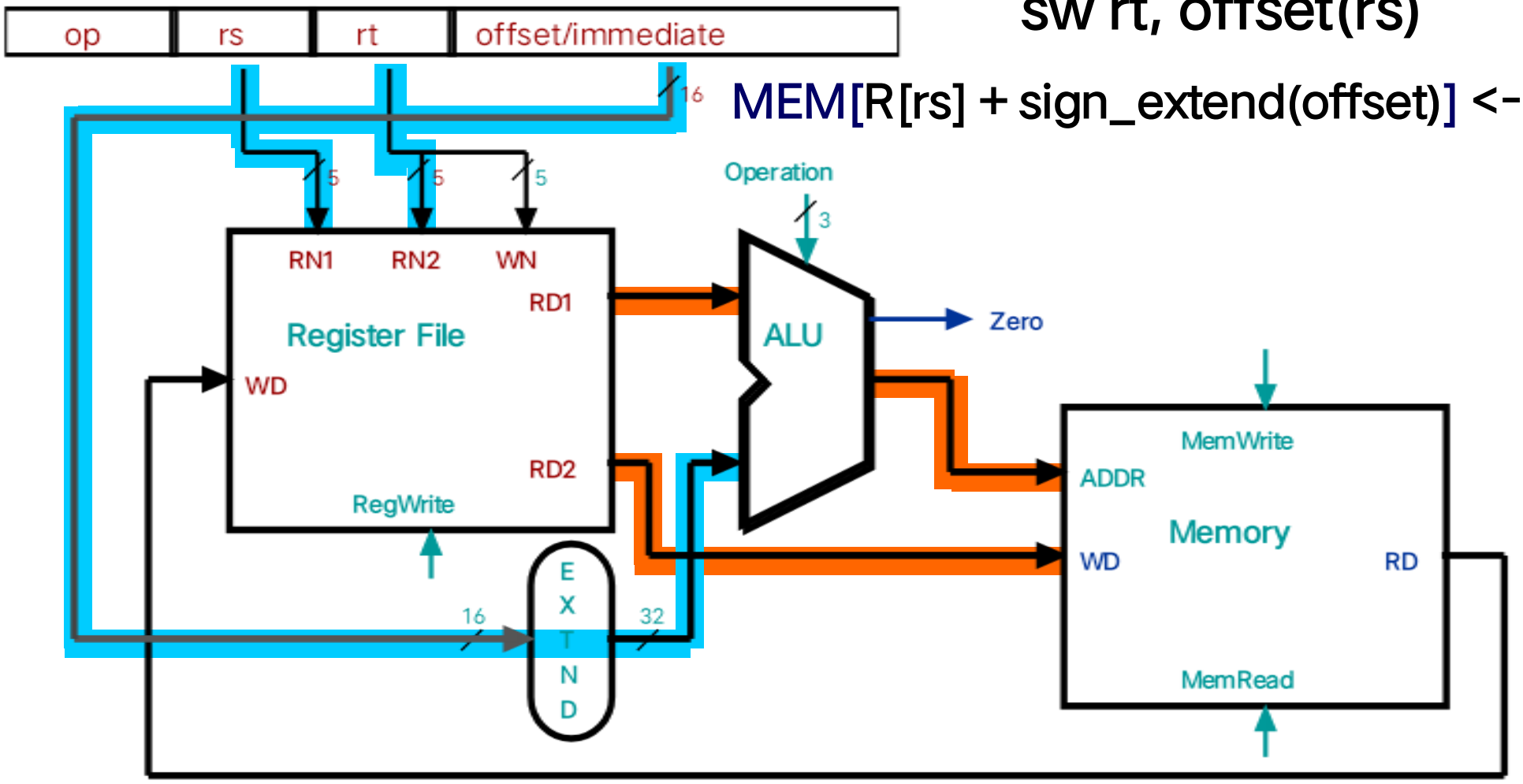
R[rt] <- MEM[R[rs] + s_extend(offset)];



Animating the Datapath

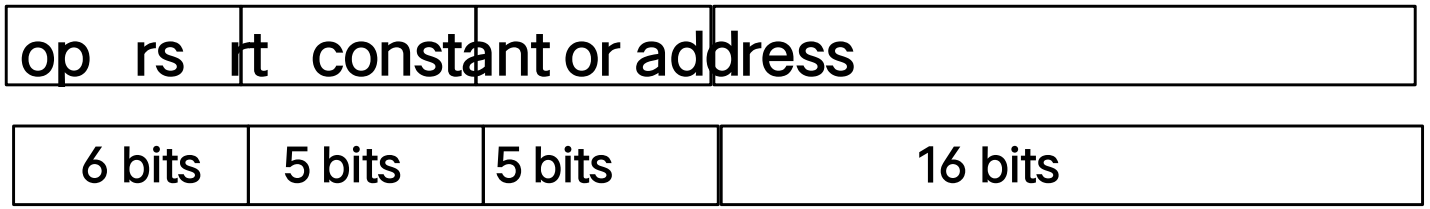
sw rt, offset(rs)

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



Recall: Branch Addressing

- Branch instructions:
  - beq rs, rt, L1
  - bne rs, rt, L1
  - Specify:
    - Opcode, two registers, target address
- Format:



- 16 bit Address ?

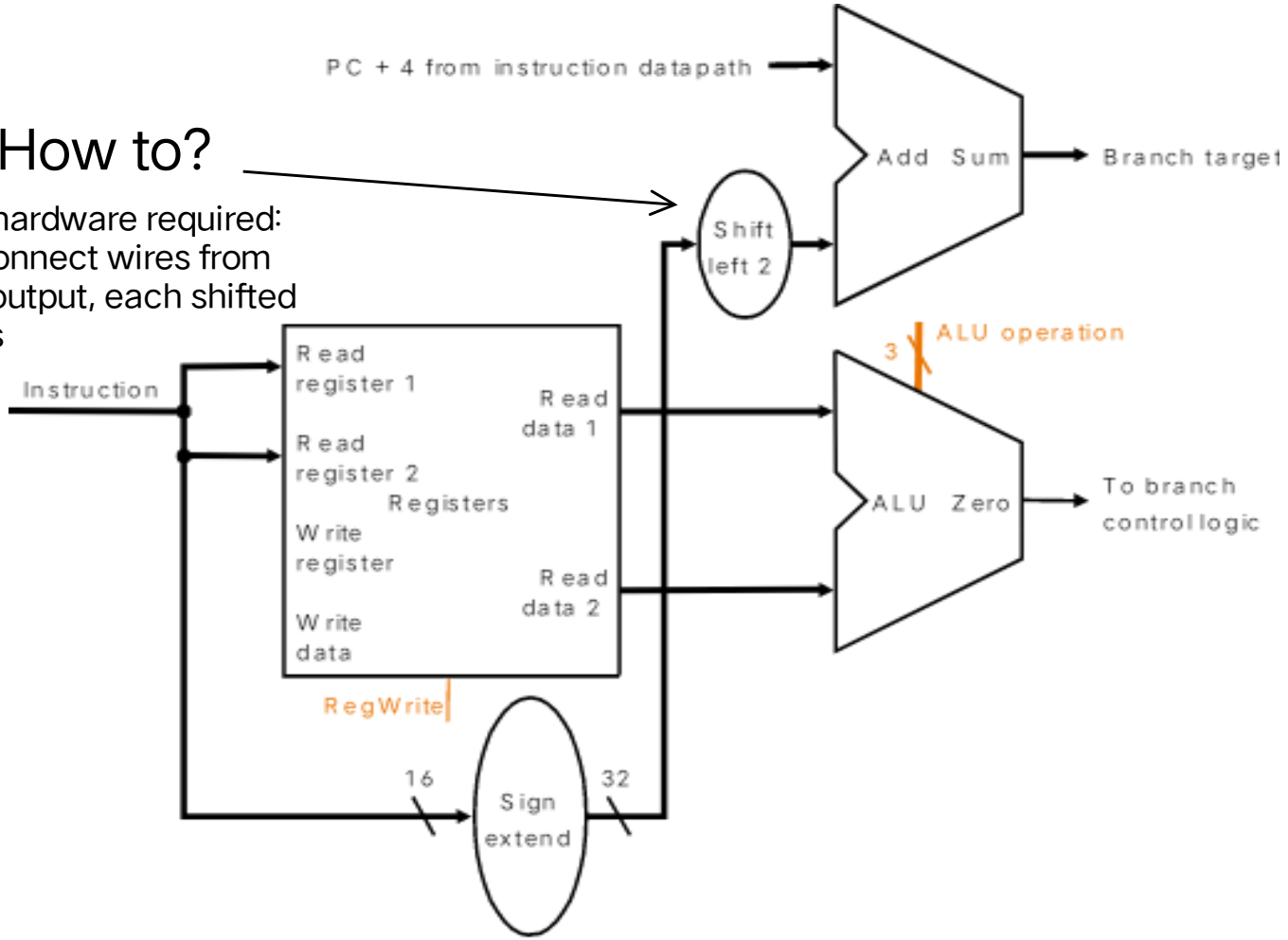
Recall: Branch Addressing

- 16 bits is too small a reach in a 2^{32} address space
- Solution:
  - Principle of locality
    - Most branch targets are near branch
    - Forward or backward Direction
  - Use PC (= program counter), called *PC-relative* addressing based on Principle of Locality
  - PC-relative addressing
    - Target address = PC + offset × 4
    - PC already incremented by 4 by this time

Datapath: Branch Instruction

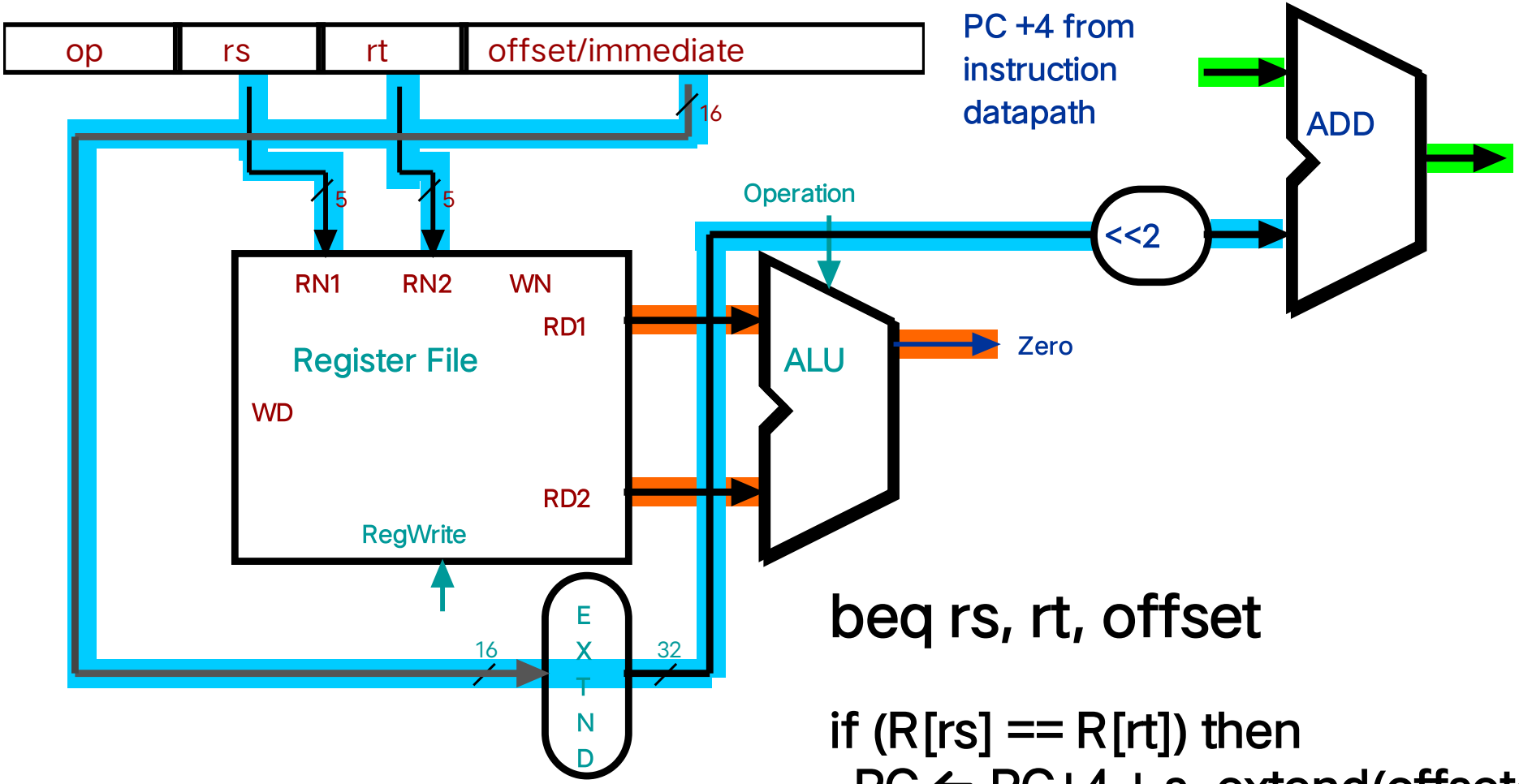
How to?

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



Datapath

Animating the Datapath



beq rs, rt, offset

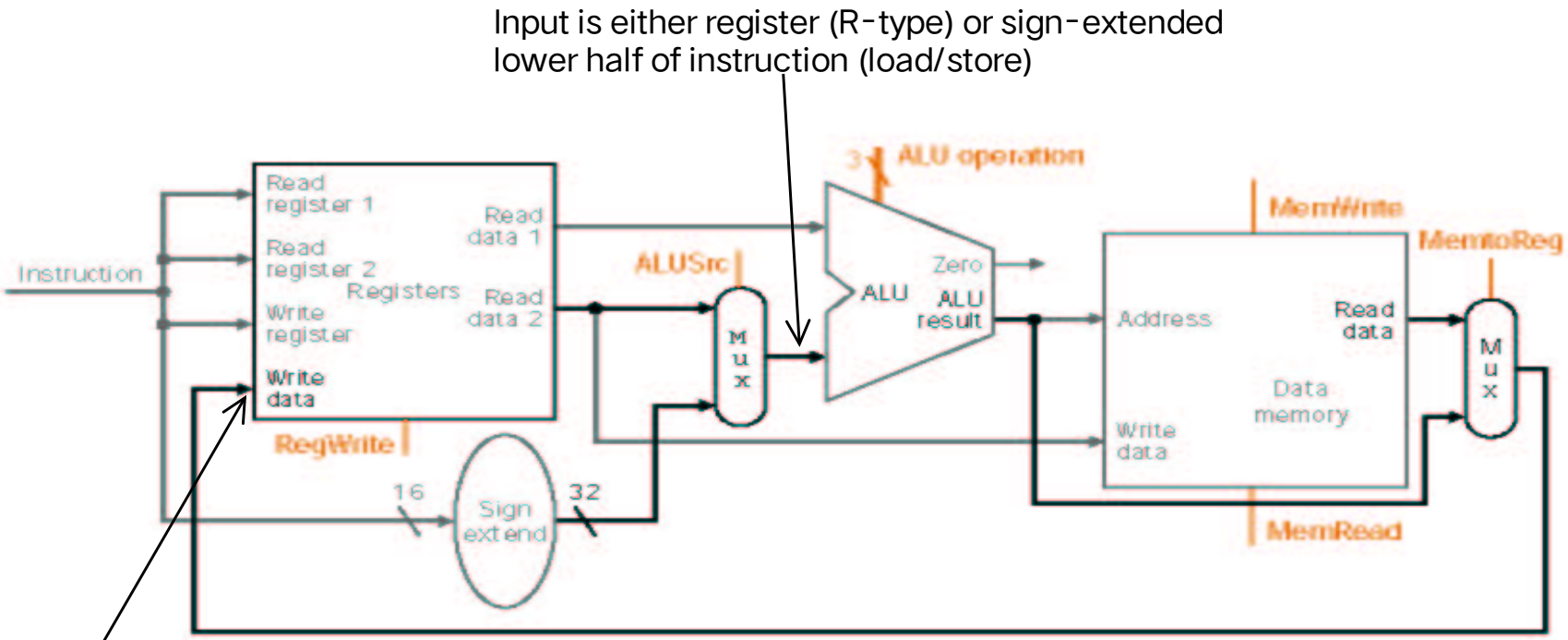
if (R[rs] == R[rt]) then
  PC ← PC+4 + s_extend(offset<<2)

Composing of the Elements

Each datapath element can only do one function at a time

- Check possibility
  - Need separate instruction and data memories
  - Use multiplexers where alternate data sources are used for different instructions

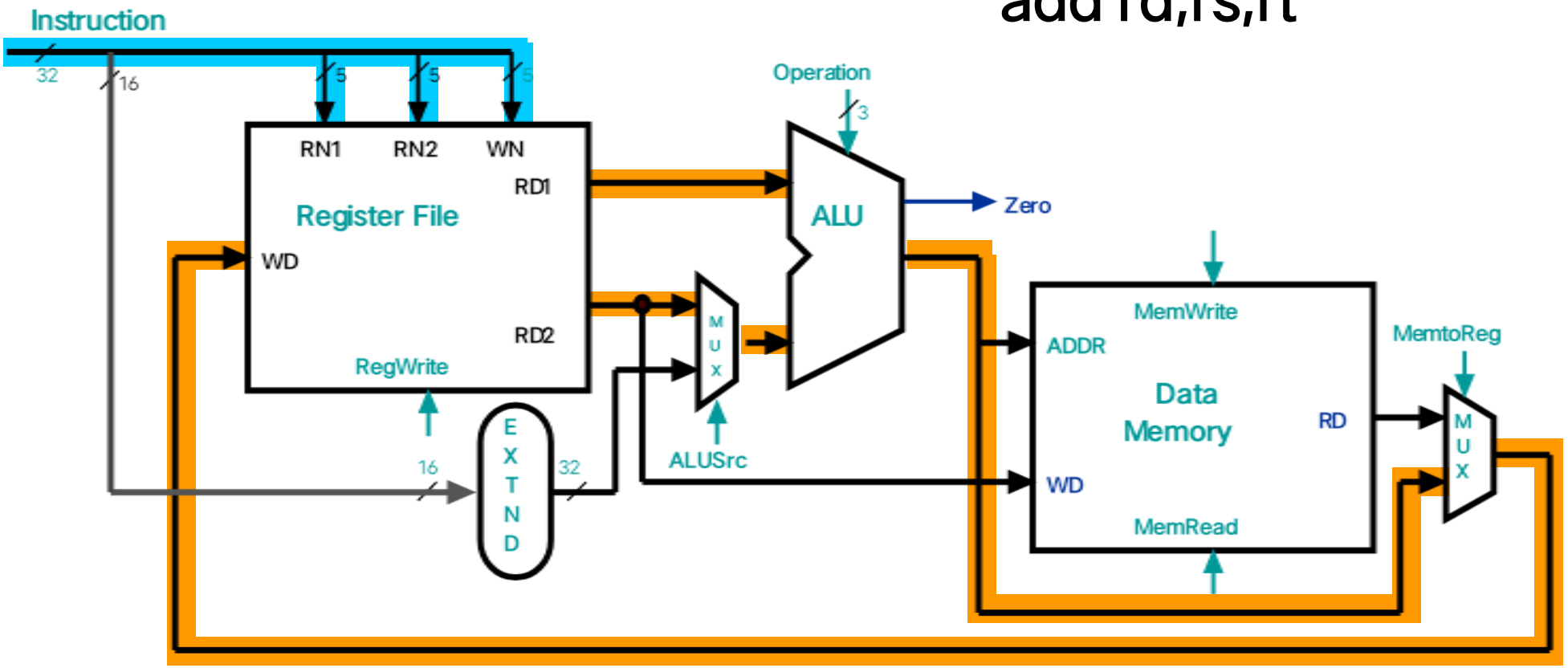
MIPS Datapath I: Single-Cycle



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

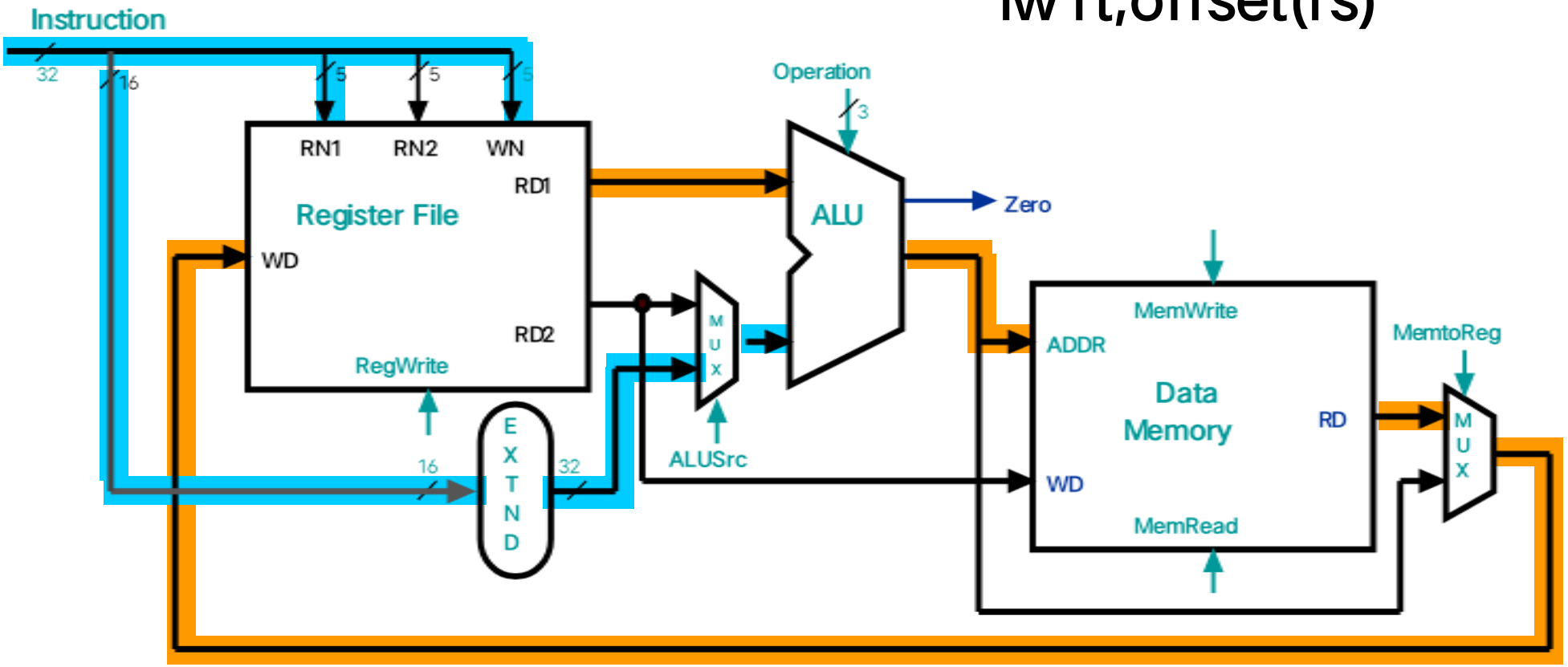
Animating the Datapath: R-type Instruction

add rd,rs,rt



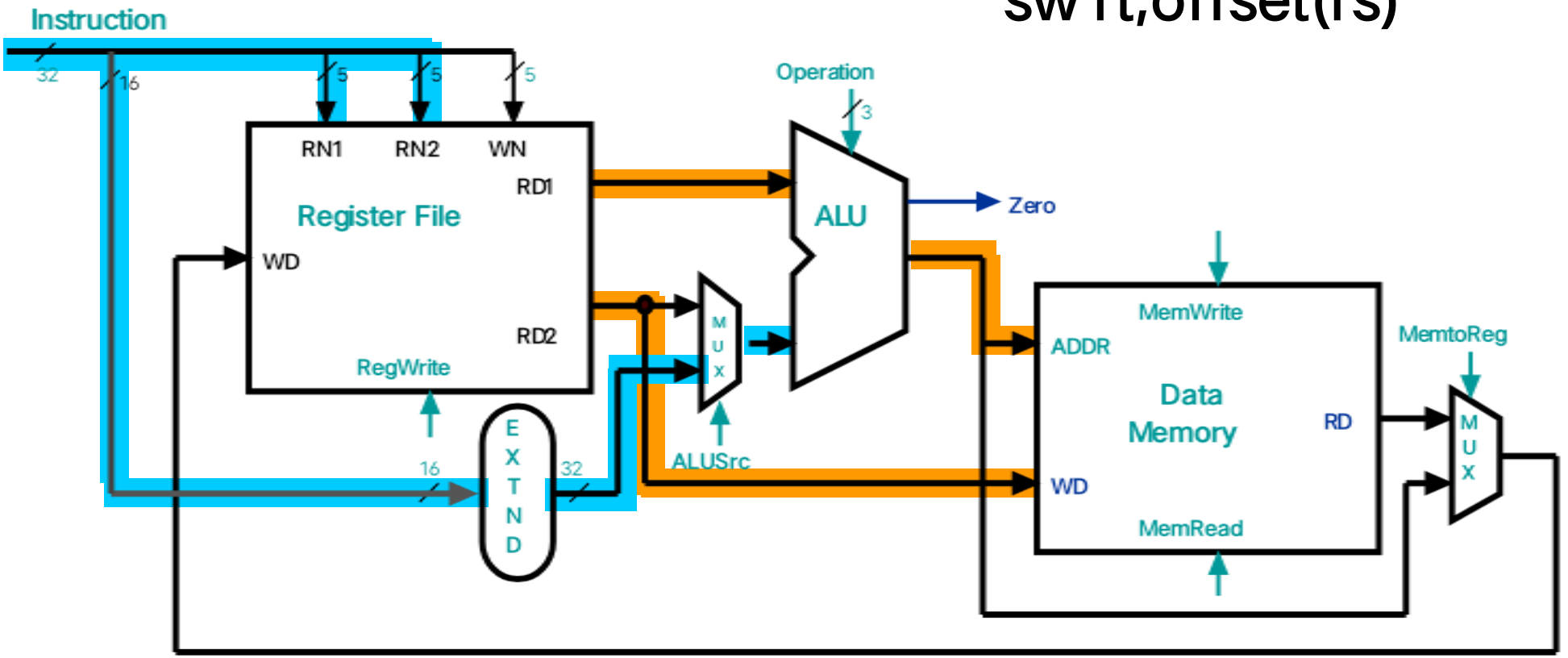
Animating the Datapath: Load Instruction

lw rt,offset(rs)

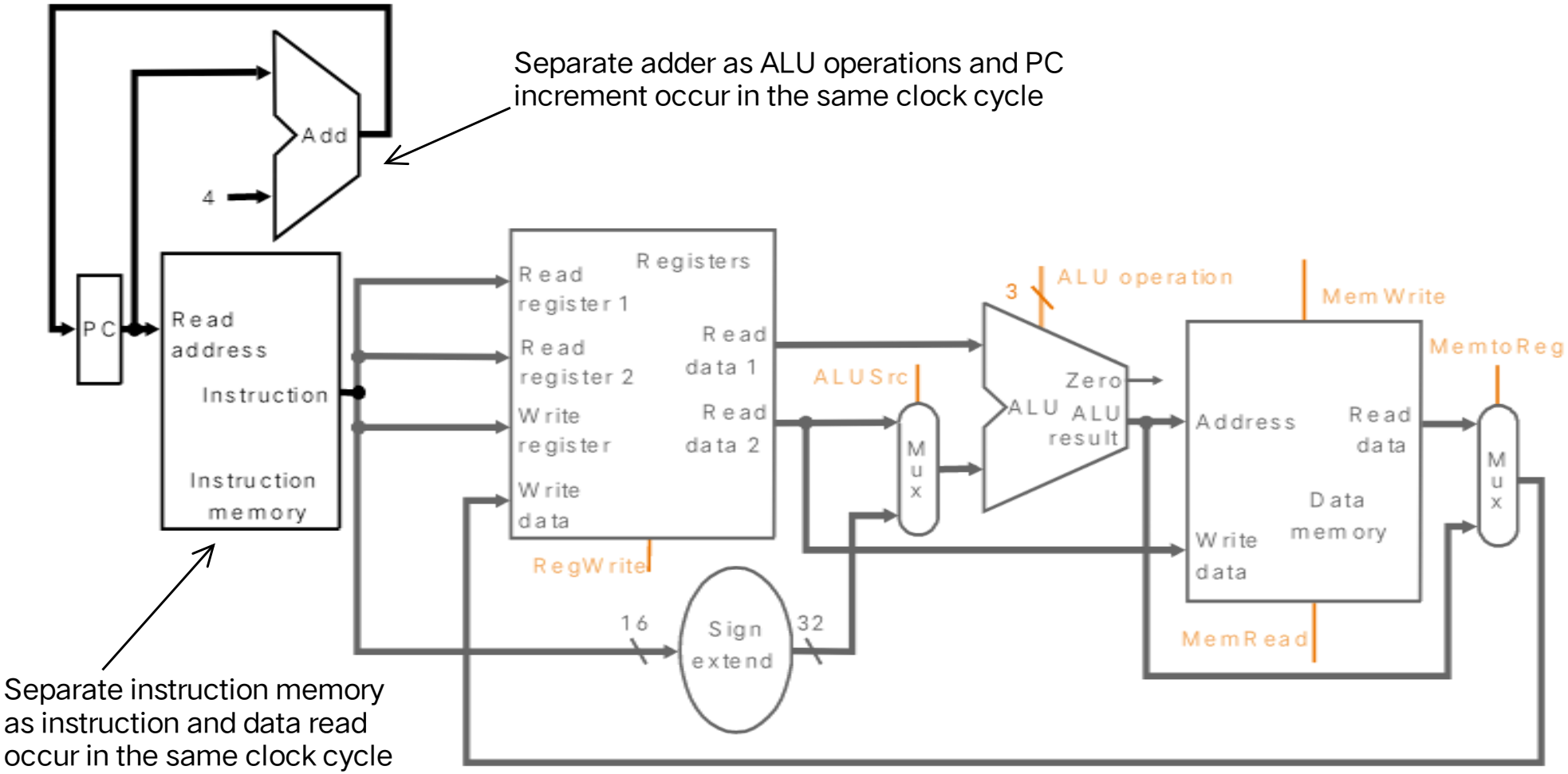


Animating the Datapath: Store Instruction

sw rt,offset(rs)

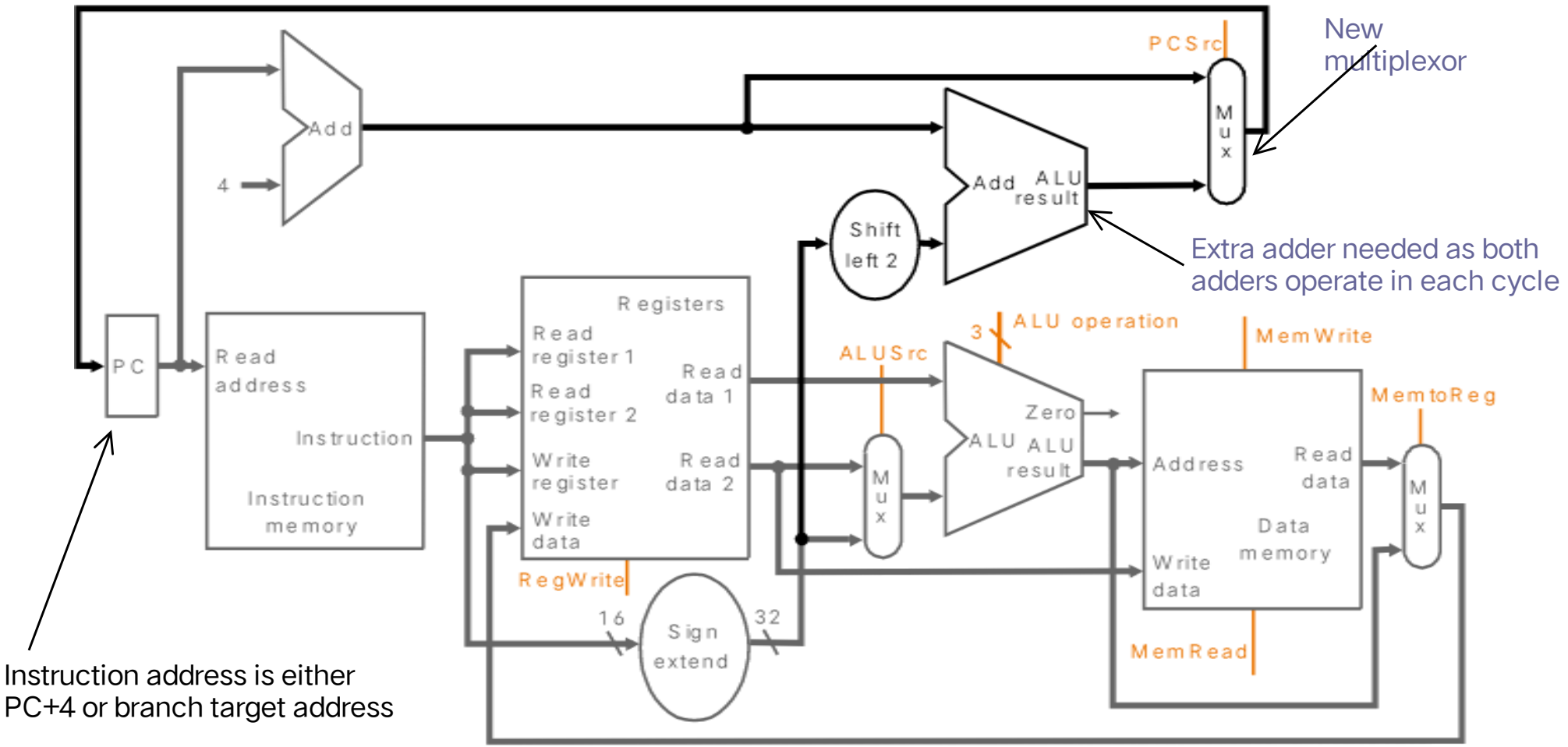


MIPS Datapath II: Single-Cycle



Adding instruction fetch

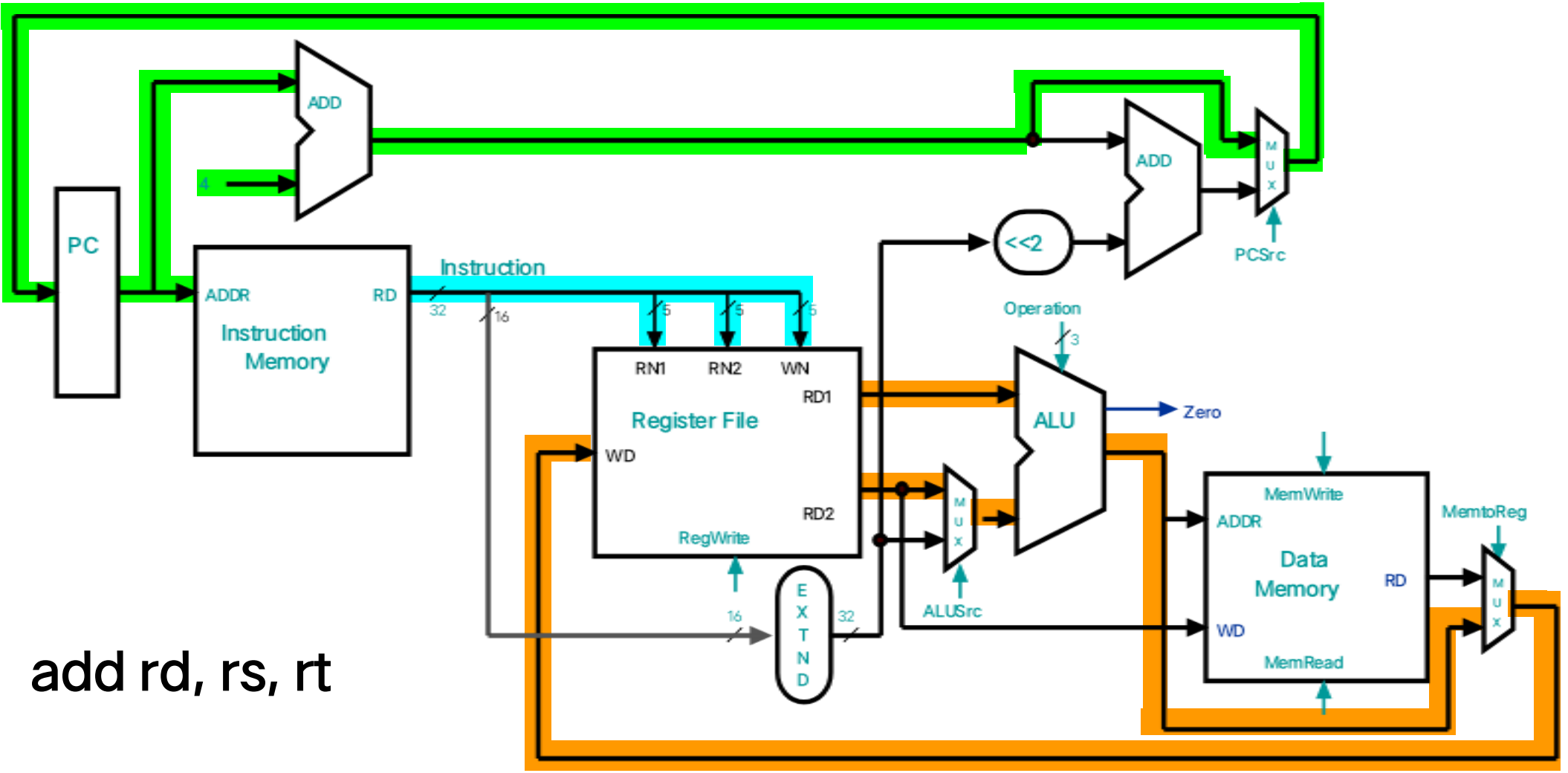
MIPS Datapath III: Single-Cycle



Adding branch capability and another multiplexor

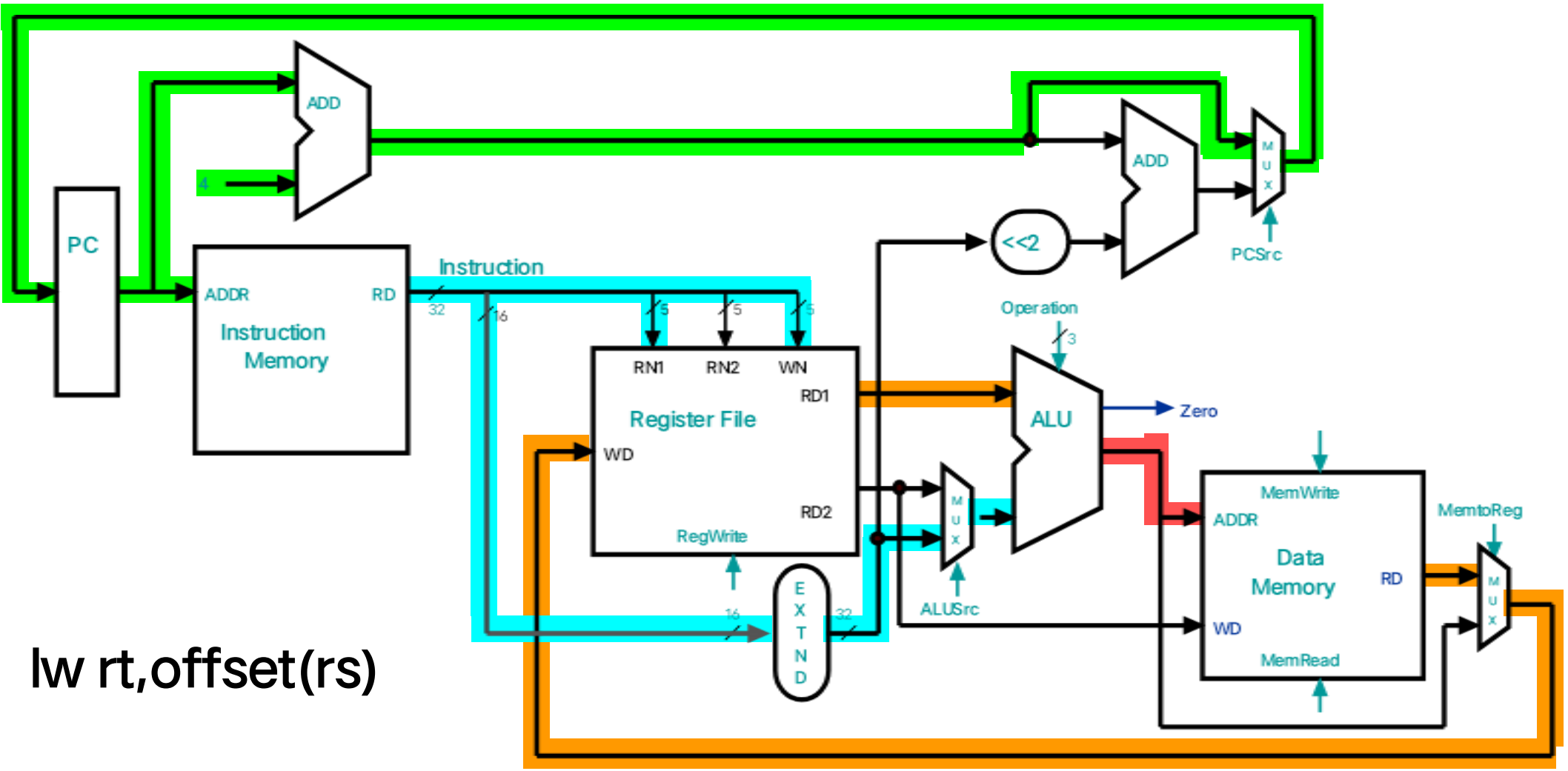
Important note: in a single-cycle implementation data cannot be stored
during an instruction – it only moves through combinational logic

Datapath Executing add



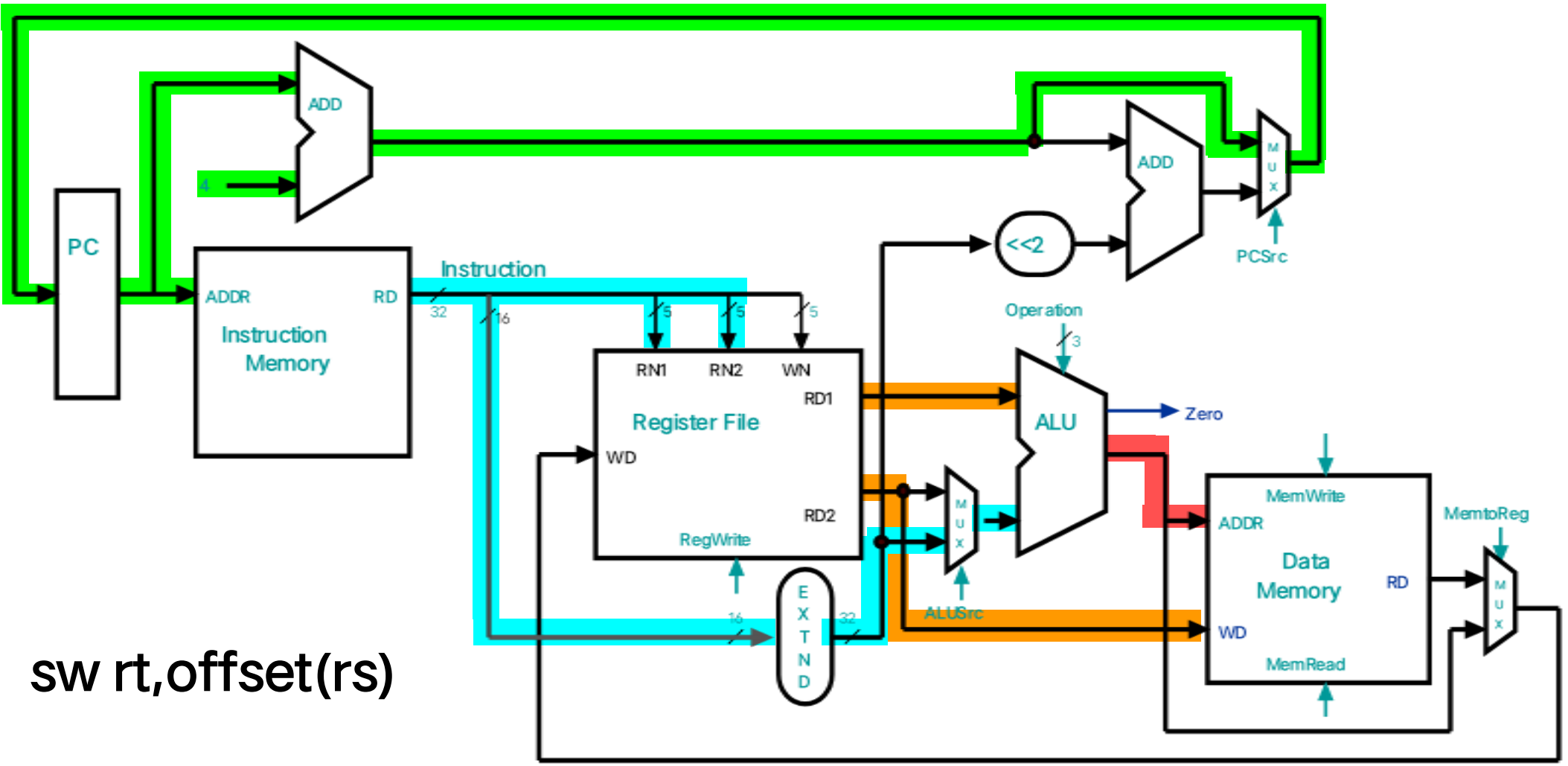
add rd, rs, rt

Datapath Executing lw



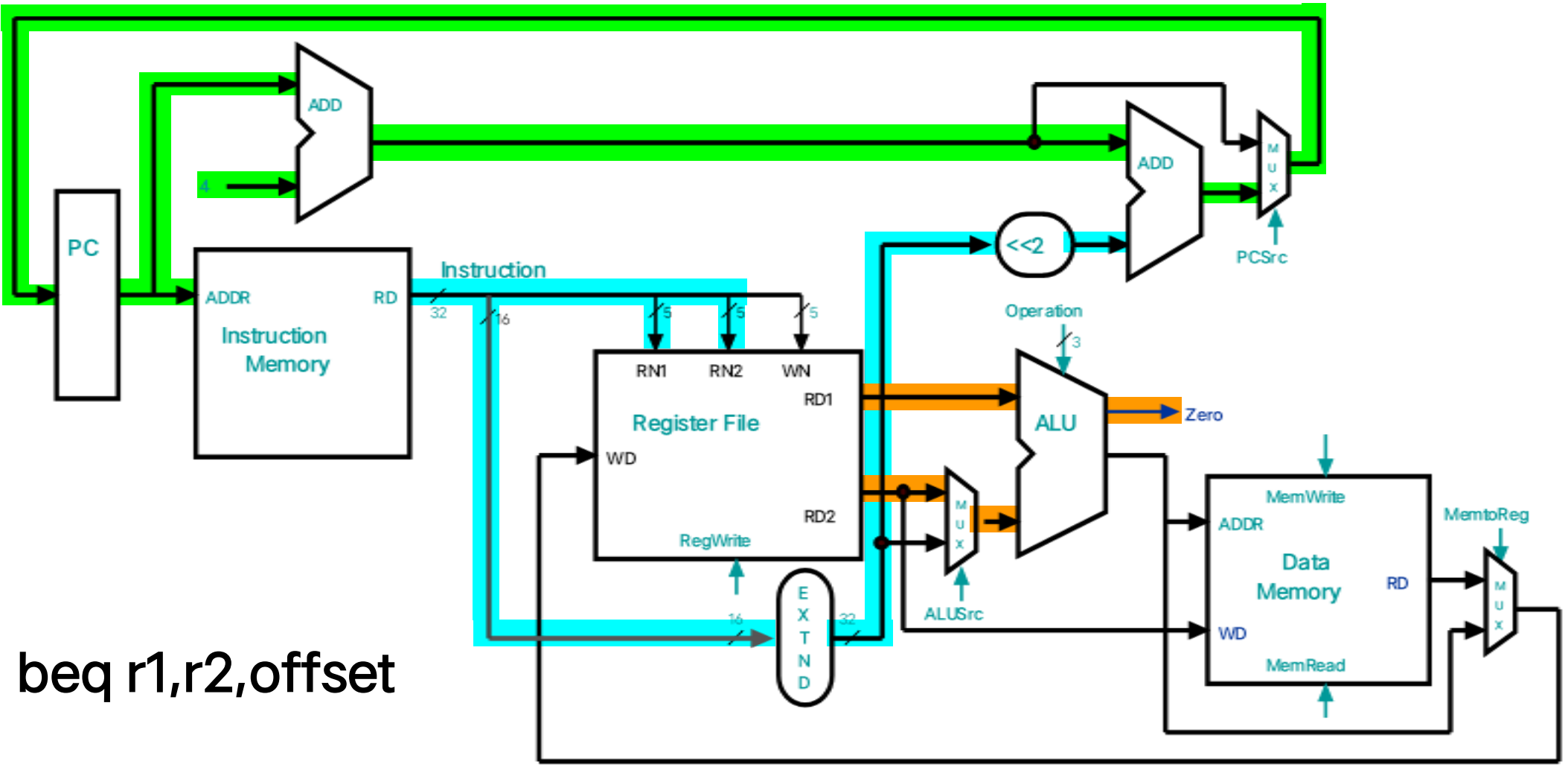
lw rt,offset(rs)

Datapath Executing sw



sw rt,offset(rs)

Datapath Executing beq

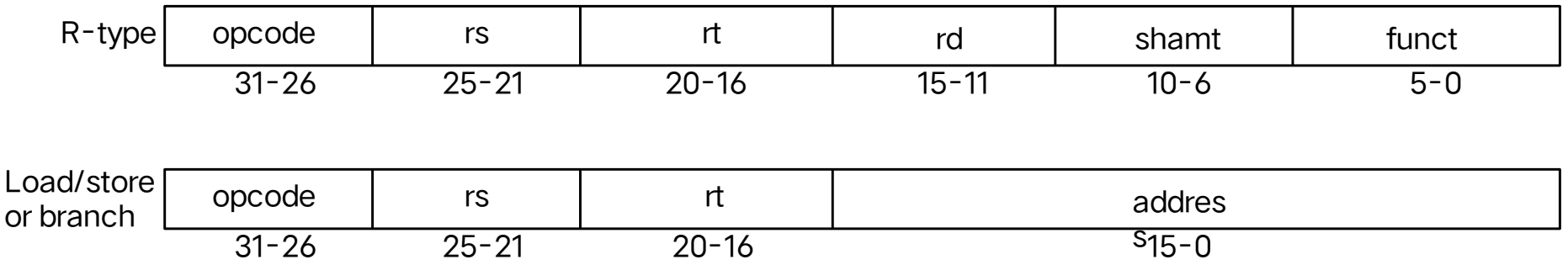


beq r1,r2,offset

Control

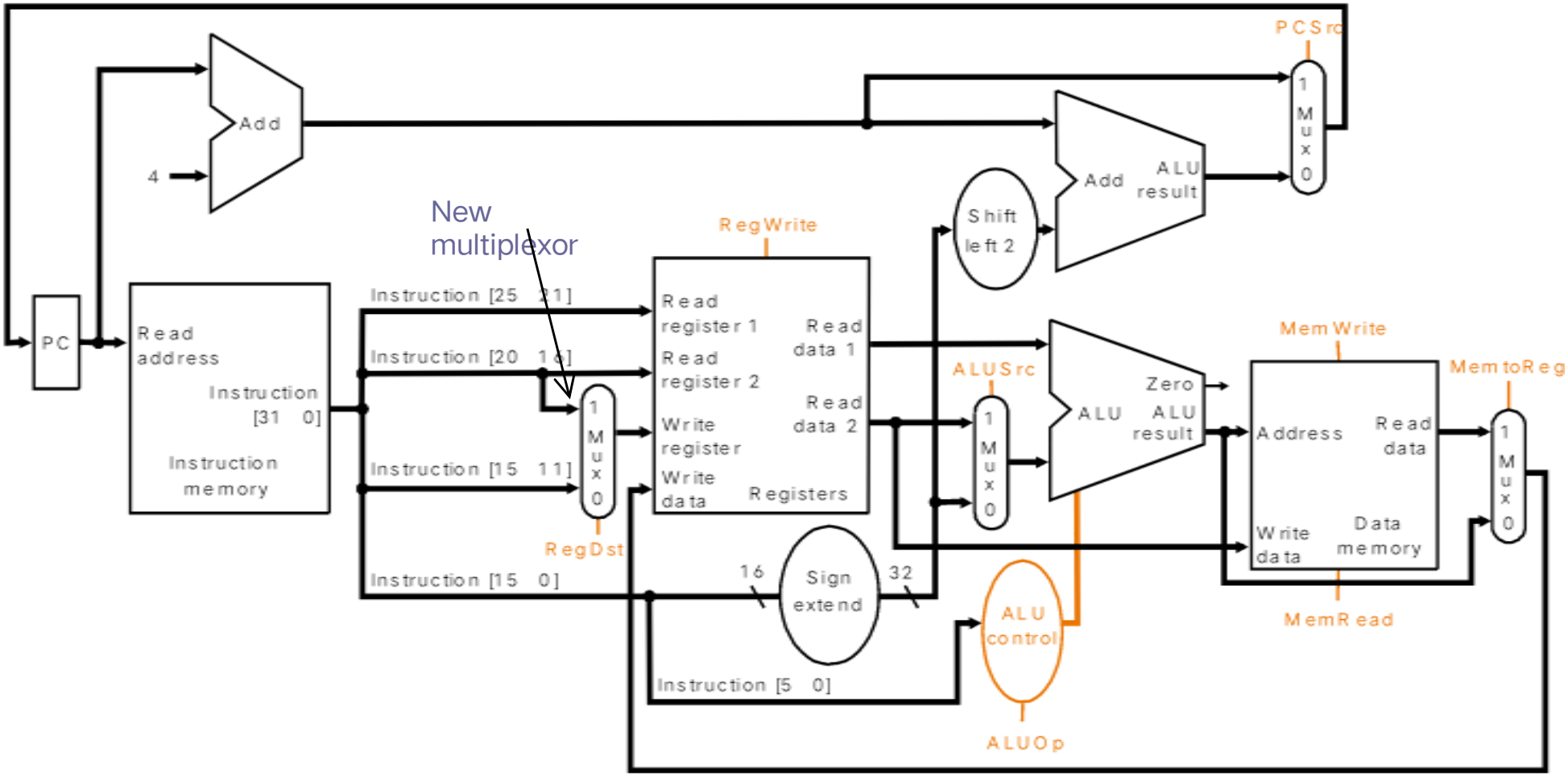
- Control unit takes input from
  - The instruction opcode bits
- Control unit generates
  - Write enable (possibly, read enable also) signals for each storage element
  - Selector controls for each multiplexor
  - ALU control input

Designing the Main Control



- 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



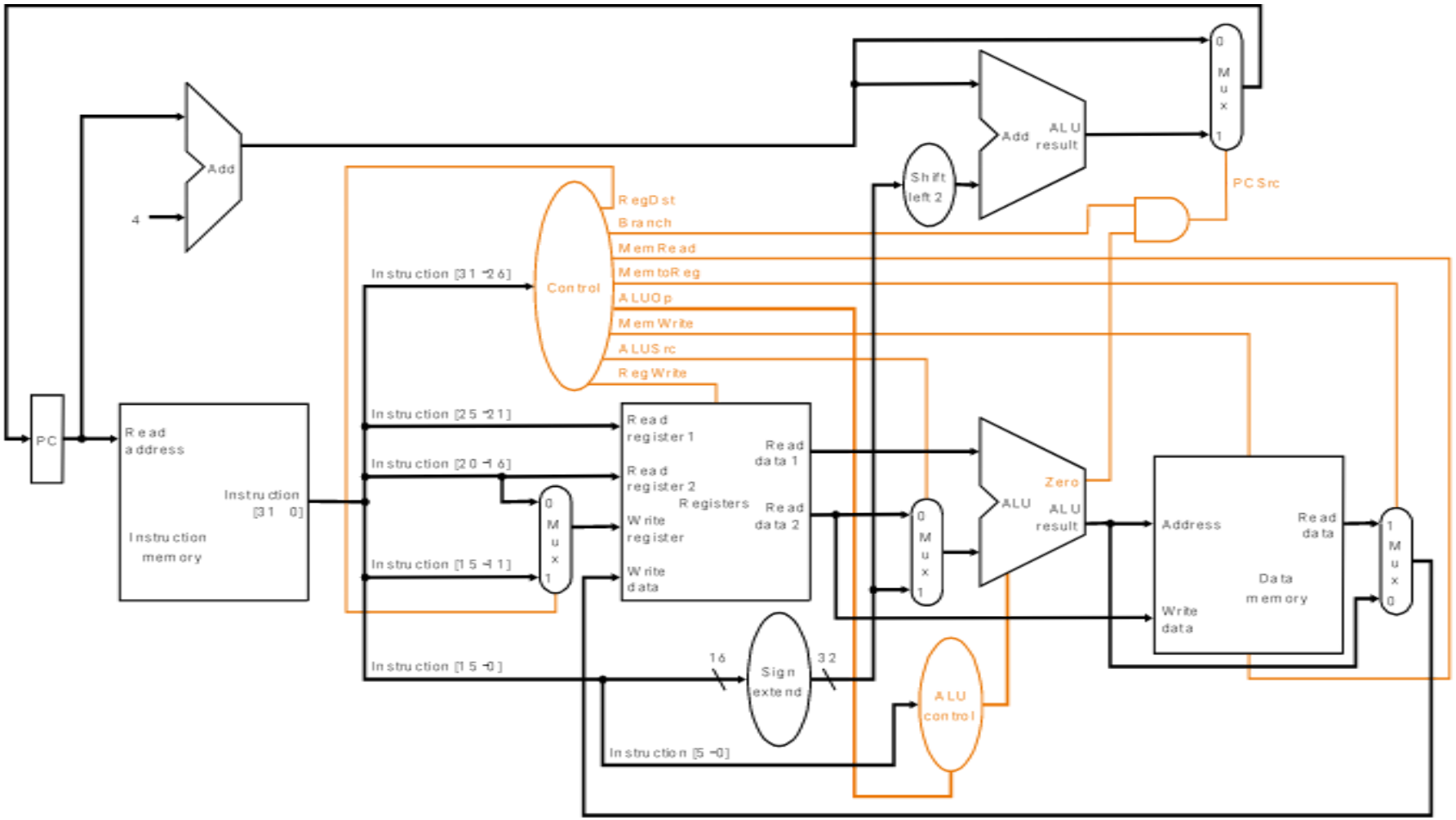
Adding control to the MIPS Datapath III (and a new multiplexor to select field to specify destination register)

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

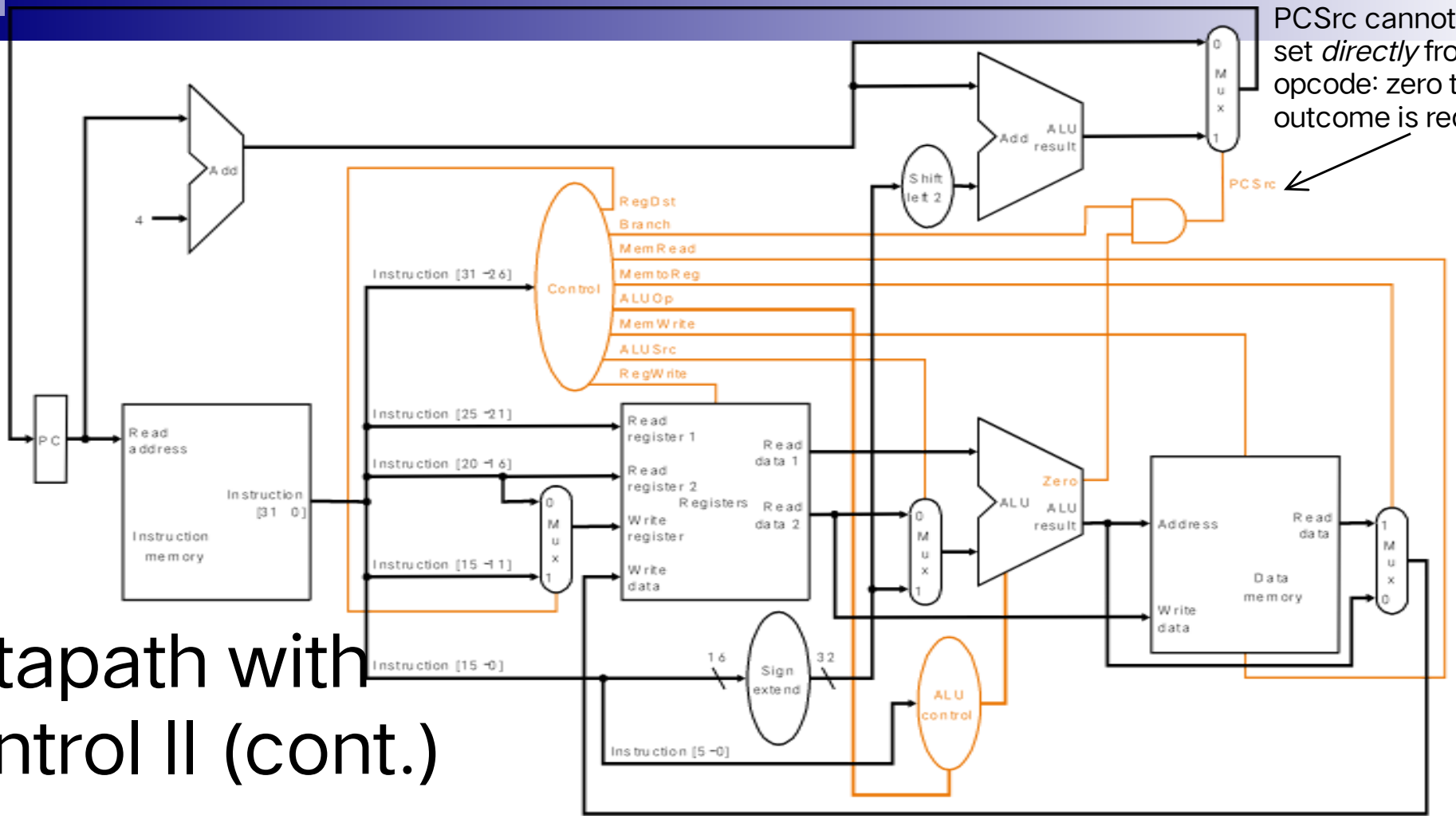
Effects of the seven control signals

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

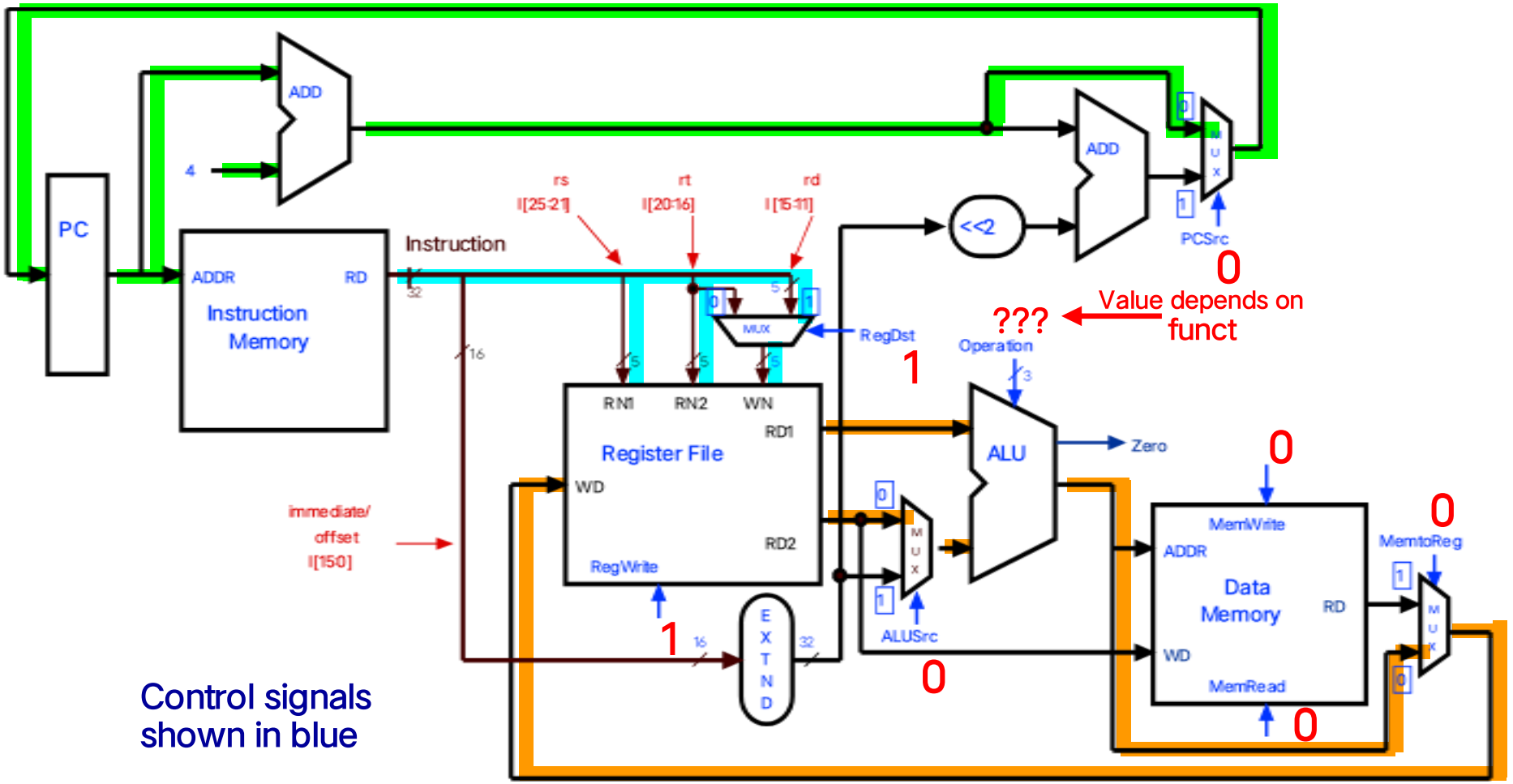
Datapath with Control II (cont.)



Determining control signals for the MIPS datapath based on instruction opcode

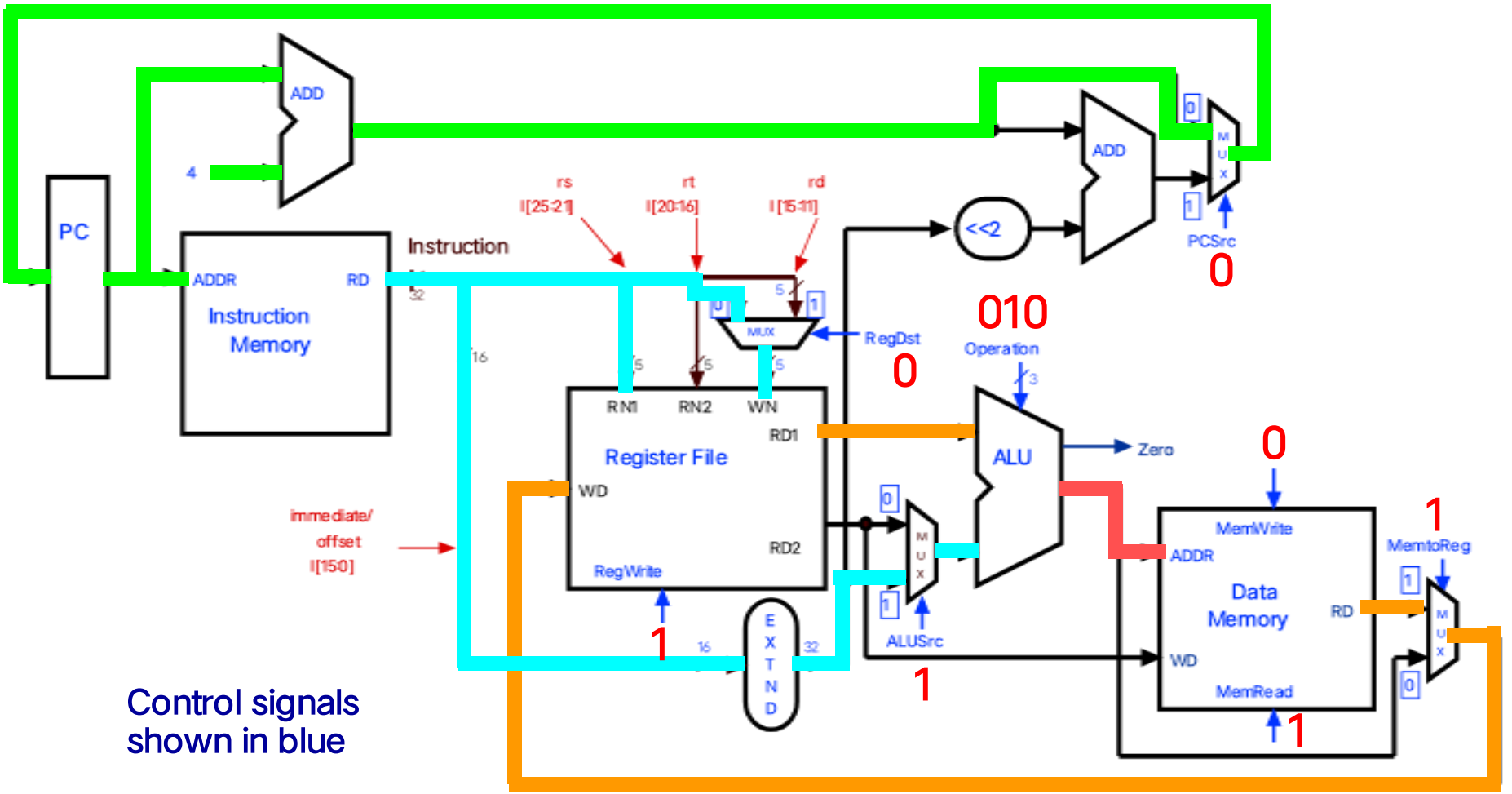
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Control Signals: R-Type Instruction

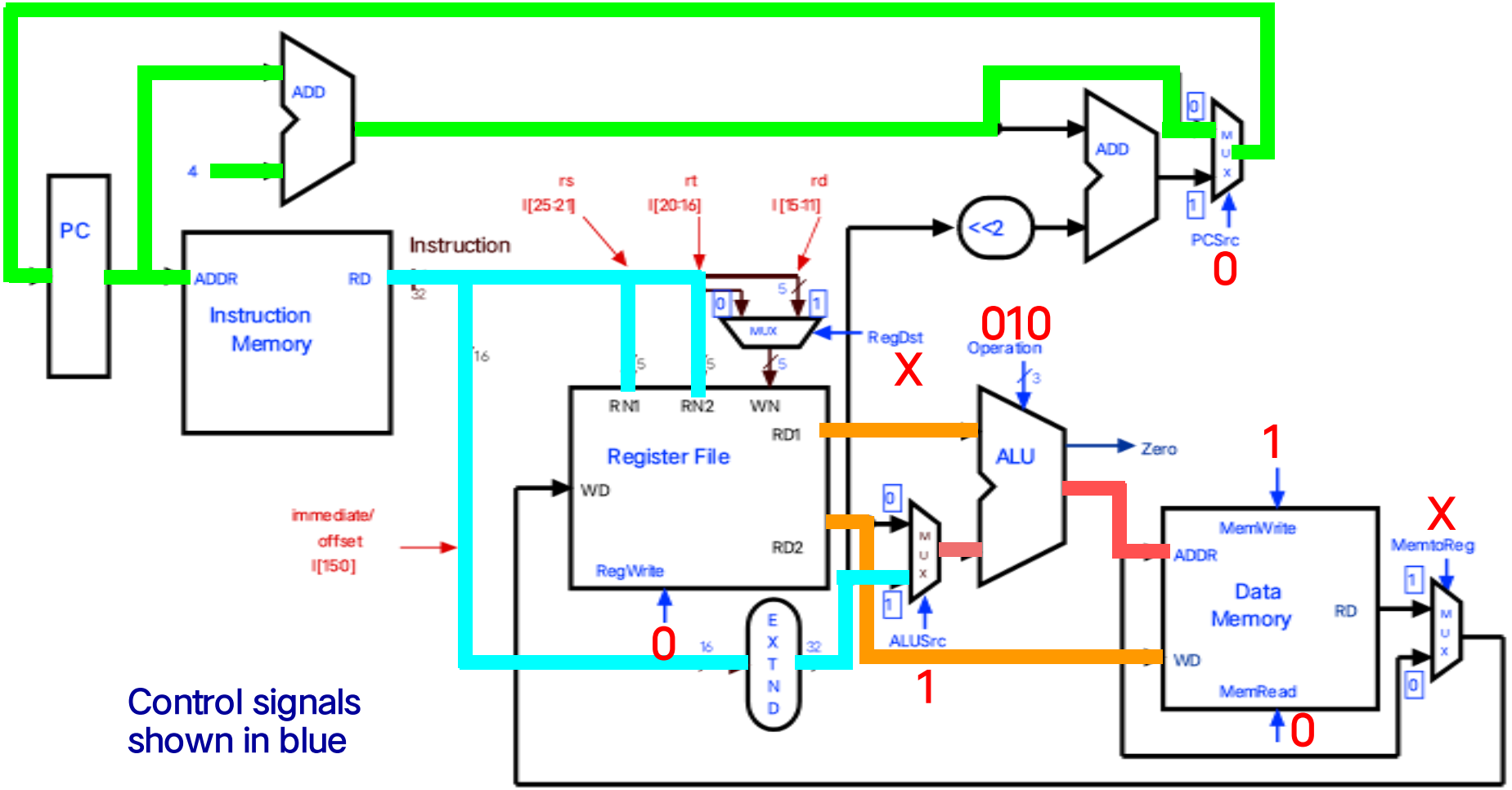


Control signals shown in blue

Control Signals: lw Instruction

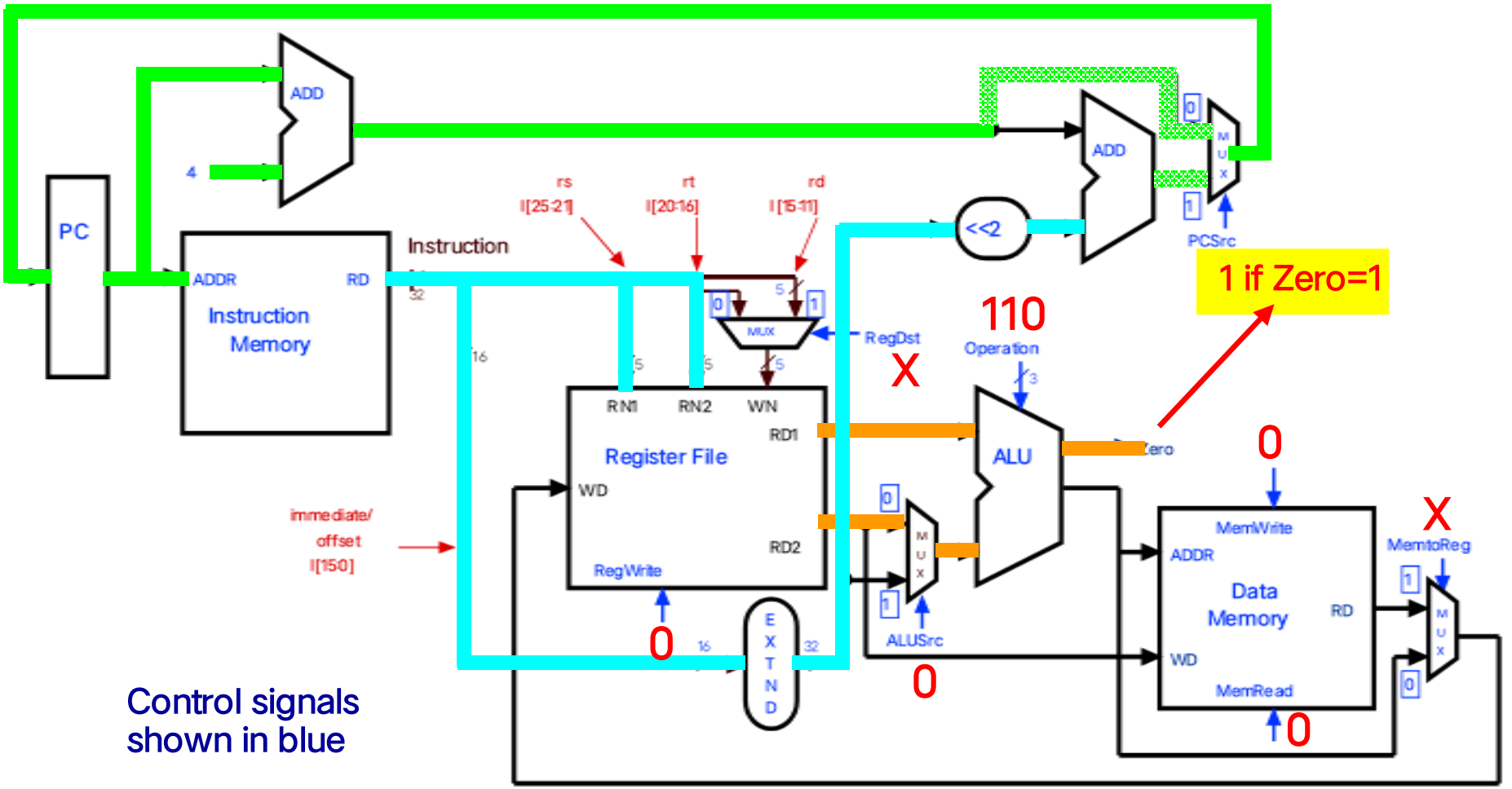


Control Signals: sw Instruction

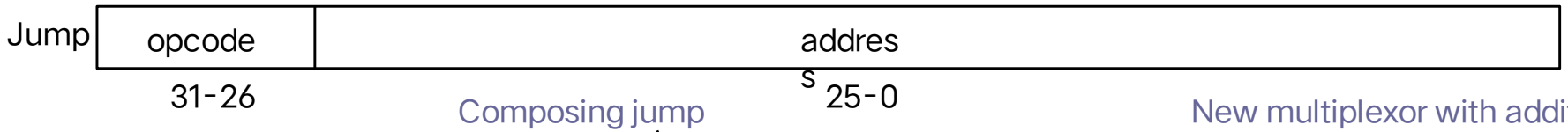


Control signals shown in blue

Control Signals: beq Instruction

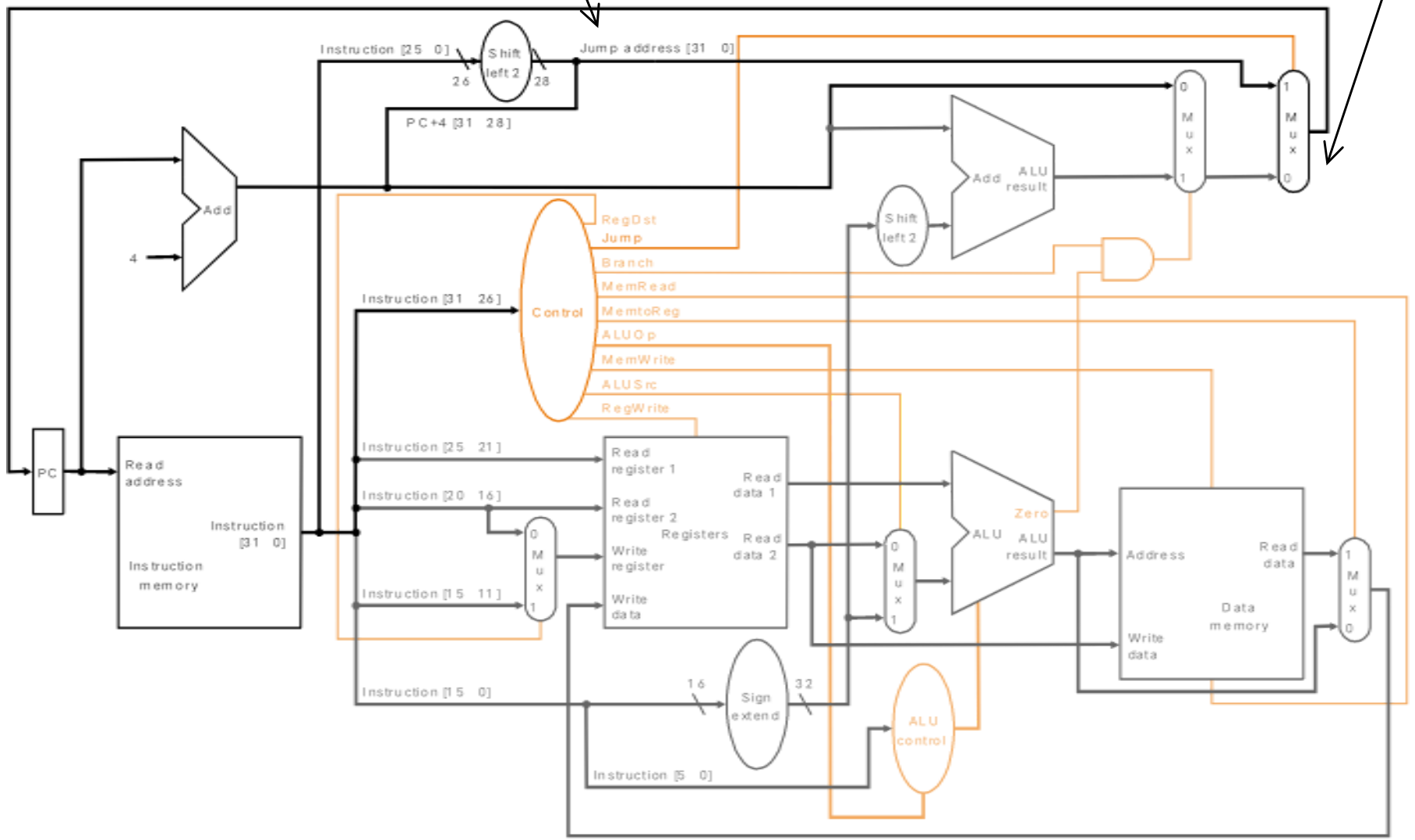


Datapath with Control III



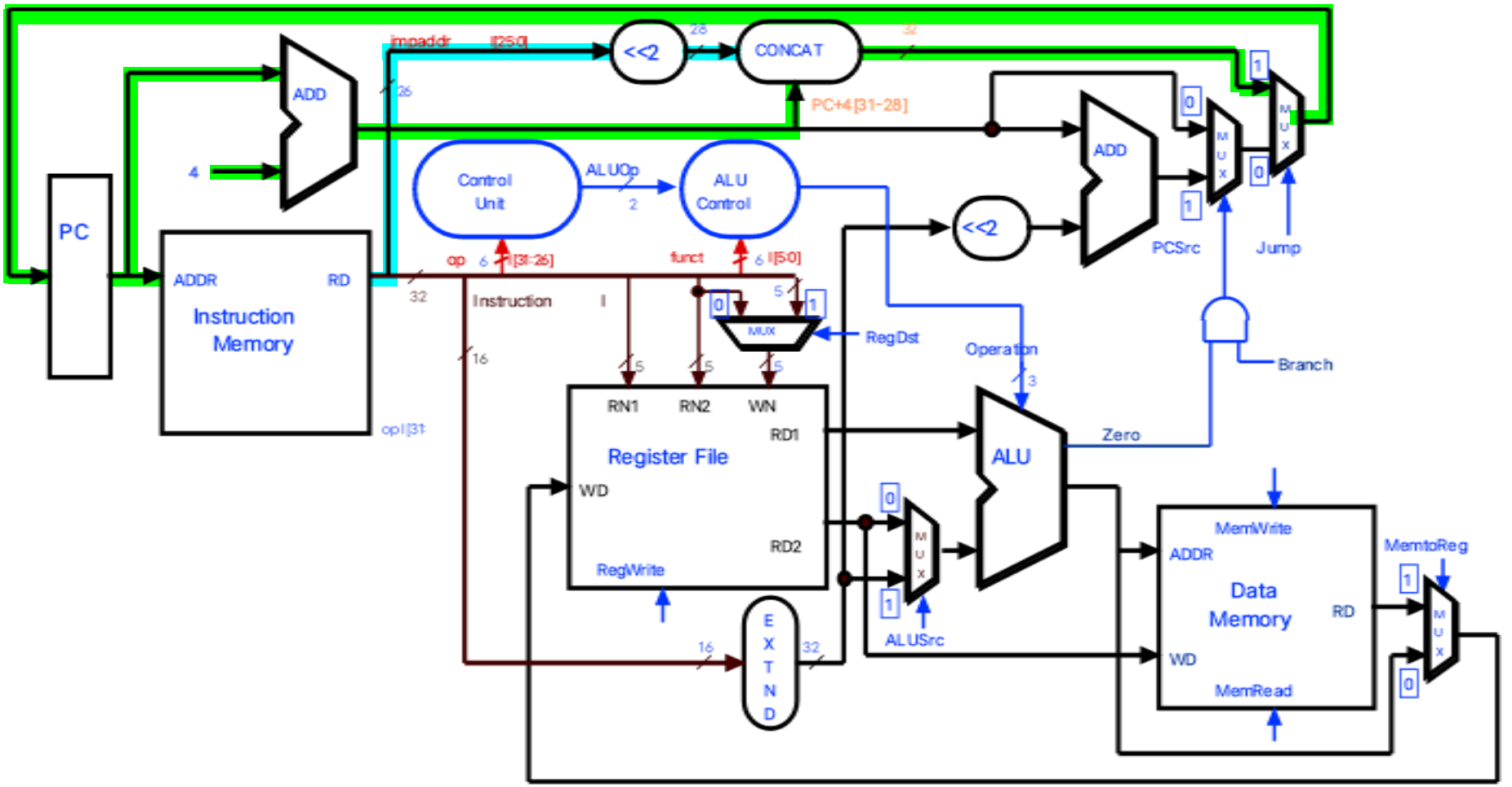
Composing jump target address

New multiplexor with additional control bit Jump



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

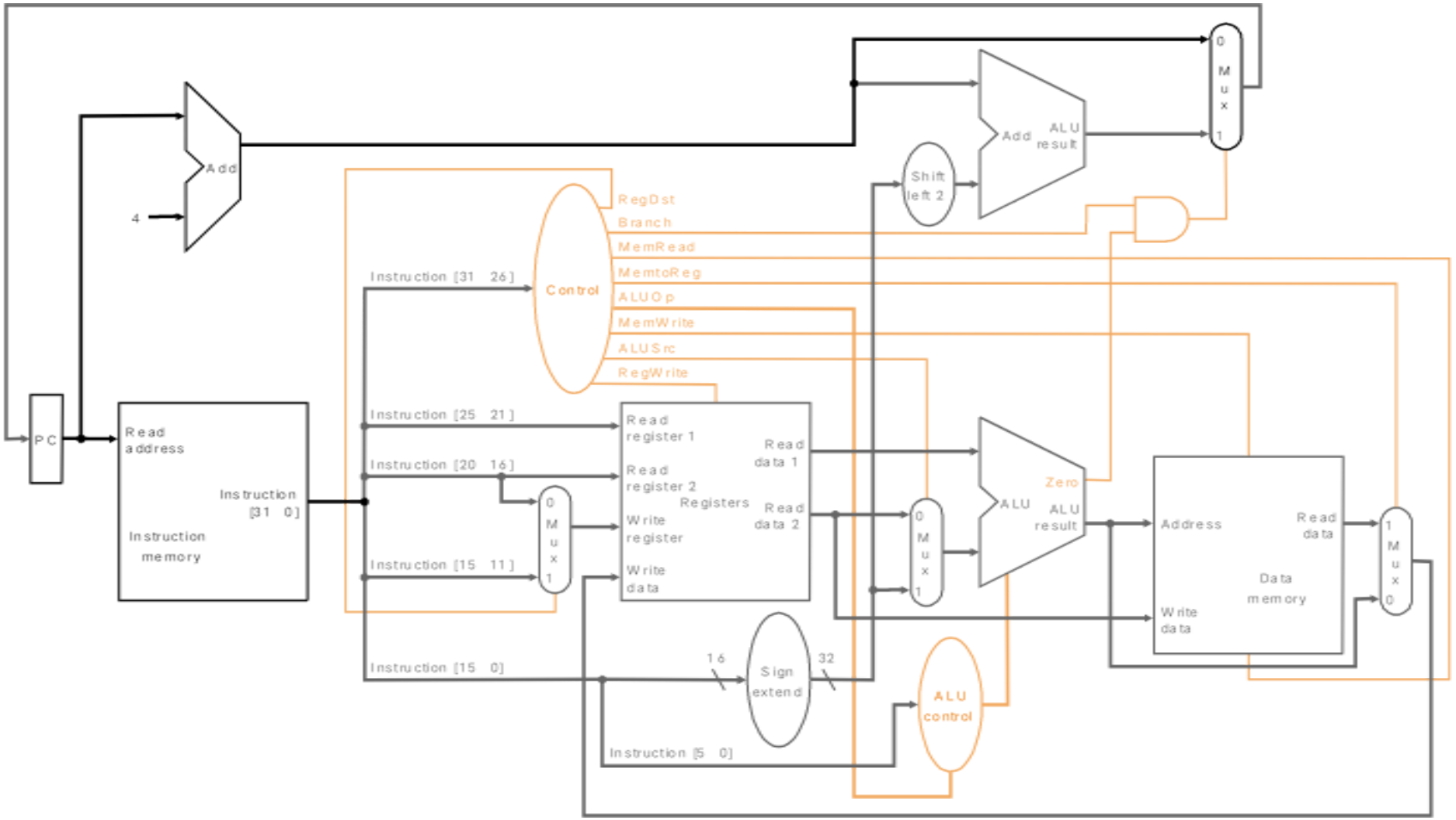
Datapath Executing j



R-Type Instruction Steps
add $t1, $t2, $t3

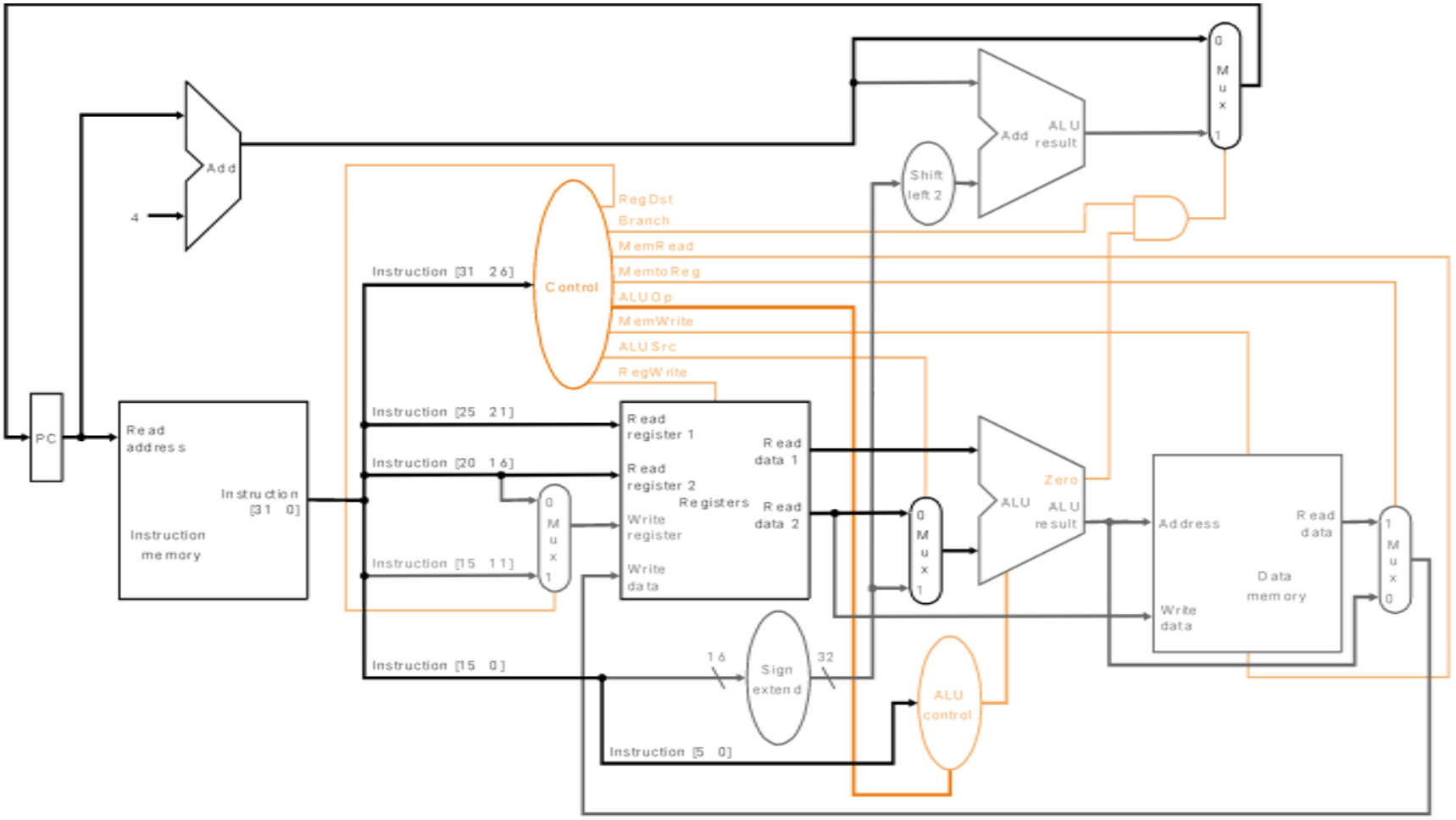
1. Fetch instruction and increment PC
2. Read two source registers from the register file
3. ALU operates on the two register operands
4. Write result to register

R-type Instruction: Step 1
add $t1, $t2, $t3 (active = bold)



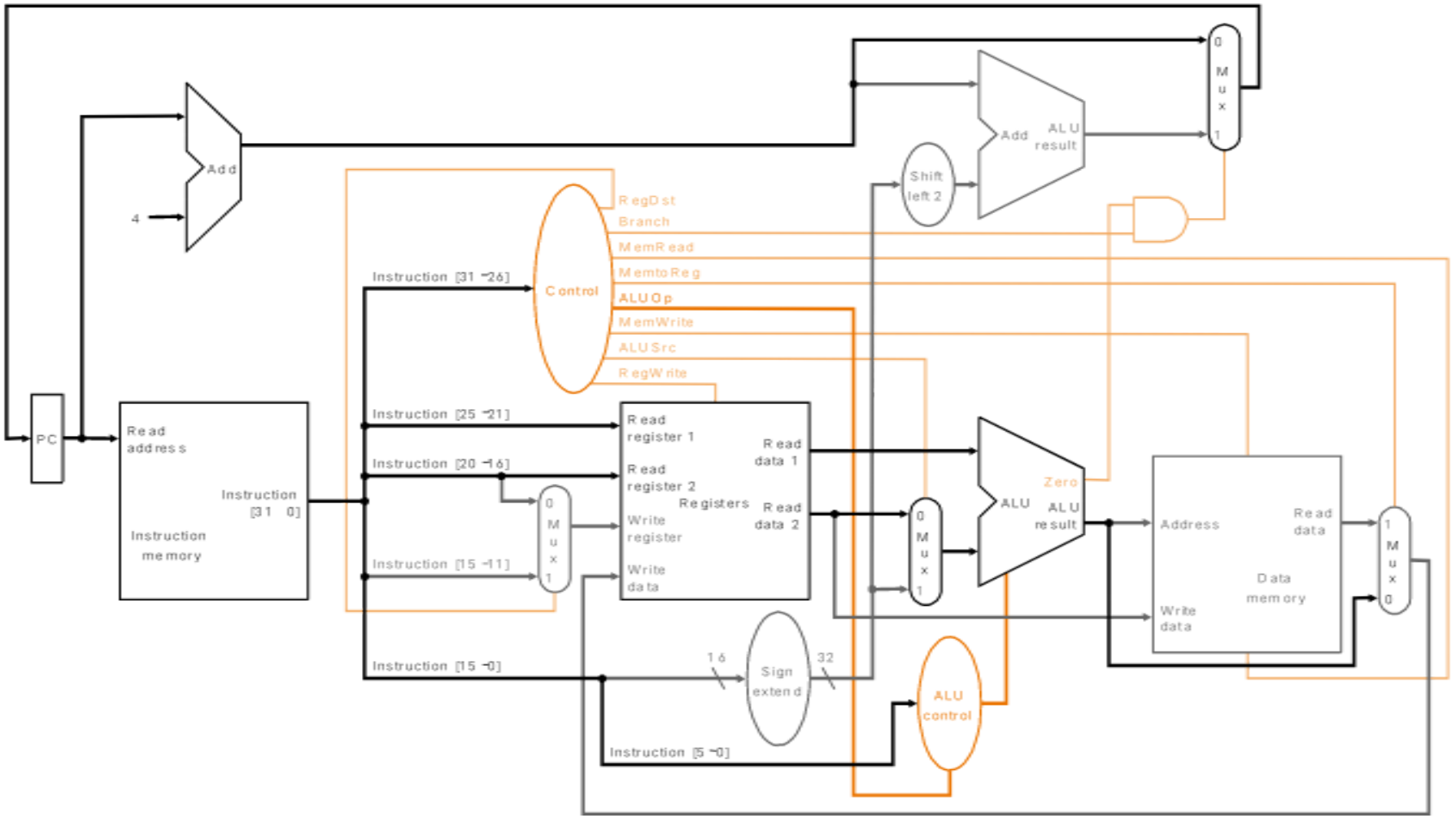
Fetch instruction and increment PC count

R-type Instruction: Step 2
add $t1, $t2, $t3 (active = bold)



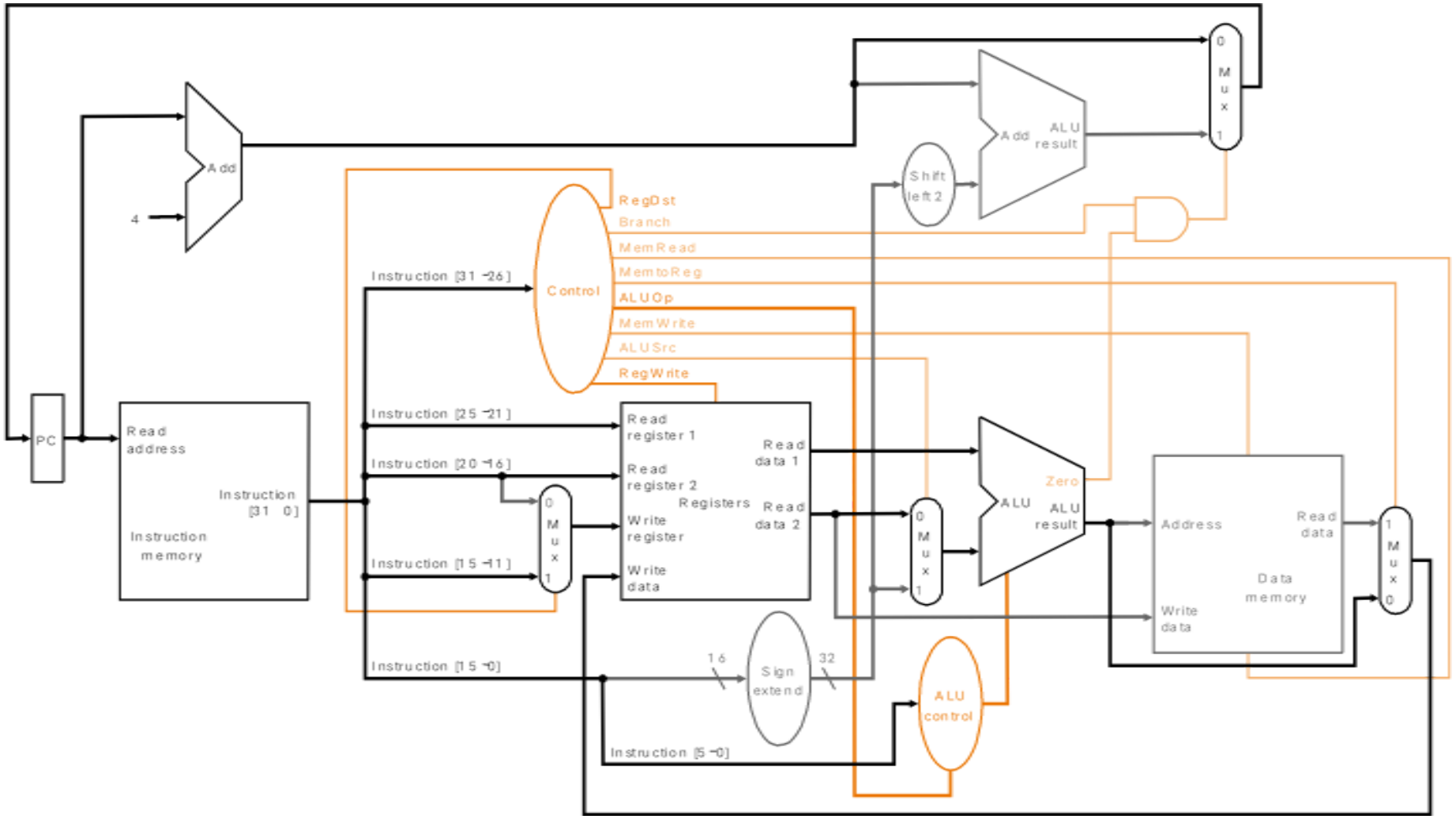
Read two source registers from the register file

R-type Instruction: Step 3
add $t1, $t2, $t3 (active = bold)



ALU operates on the two register operands

R-type Instruction: Step 4
add $t1, $t2, $t3 (active = bold)

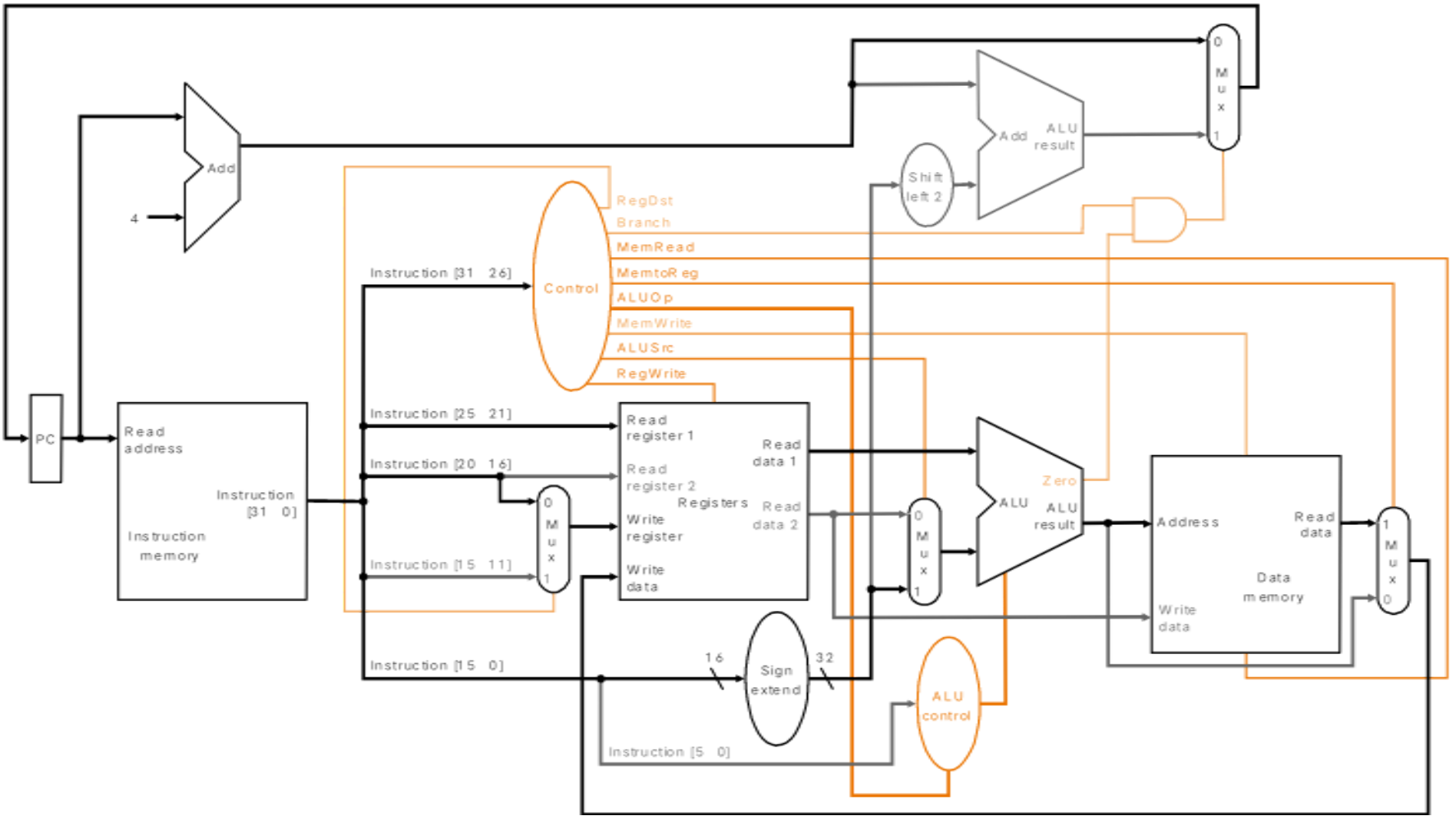


Write result to register

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

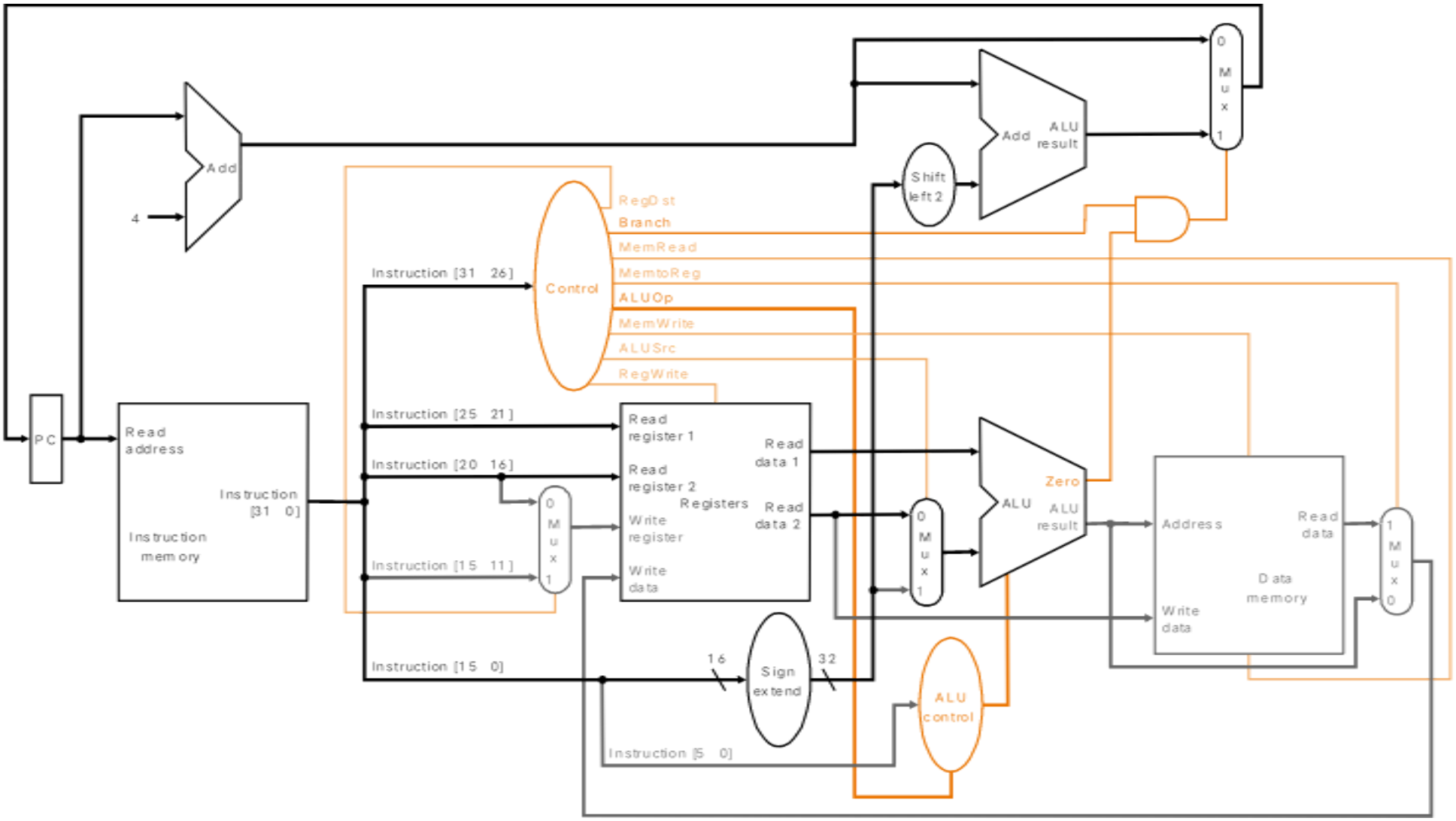
Load Instruction
lw $t1, offset($t2)



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
beq $t1, $t2, offset



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

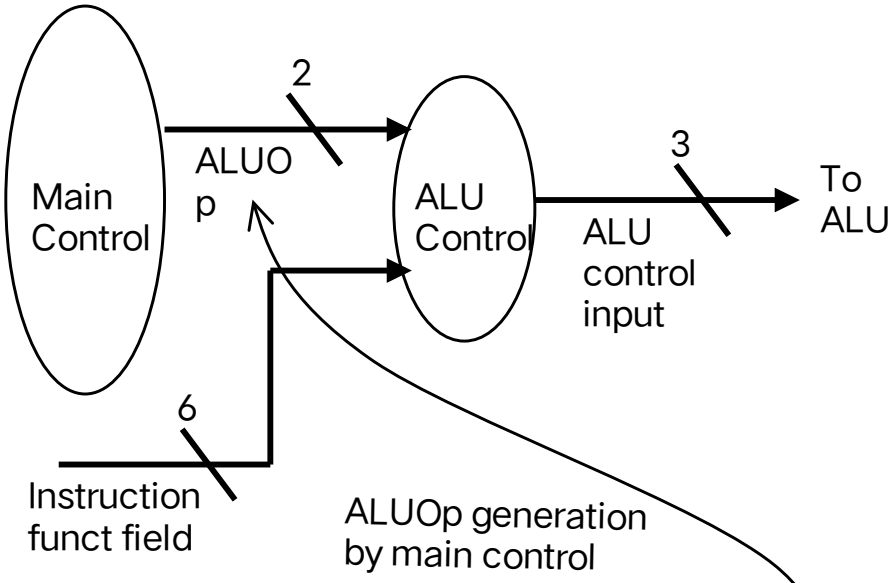
ALU control field	Func-tion
000	and
001	or
010	add
110	sub
111	slt

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

Recall from ALU Design

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

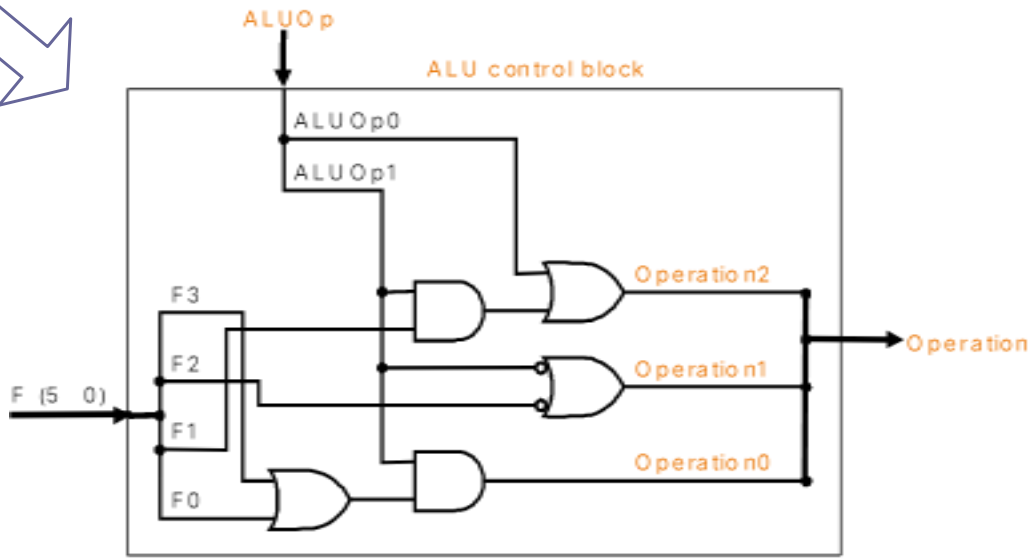
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits

Implementation: 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

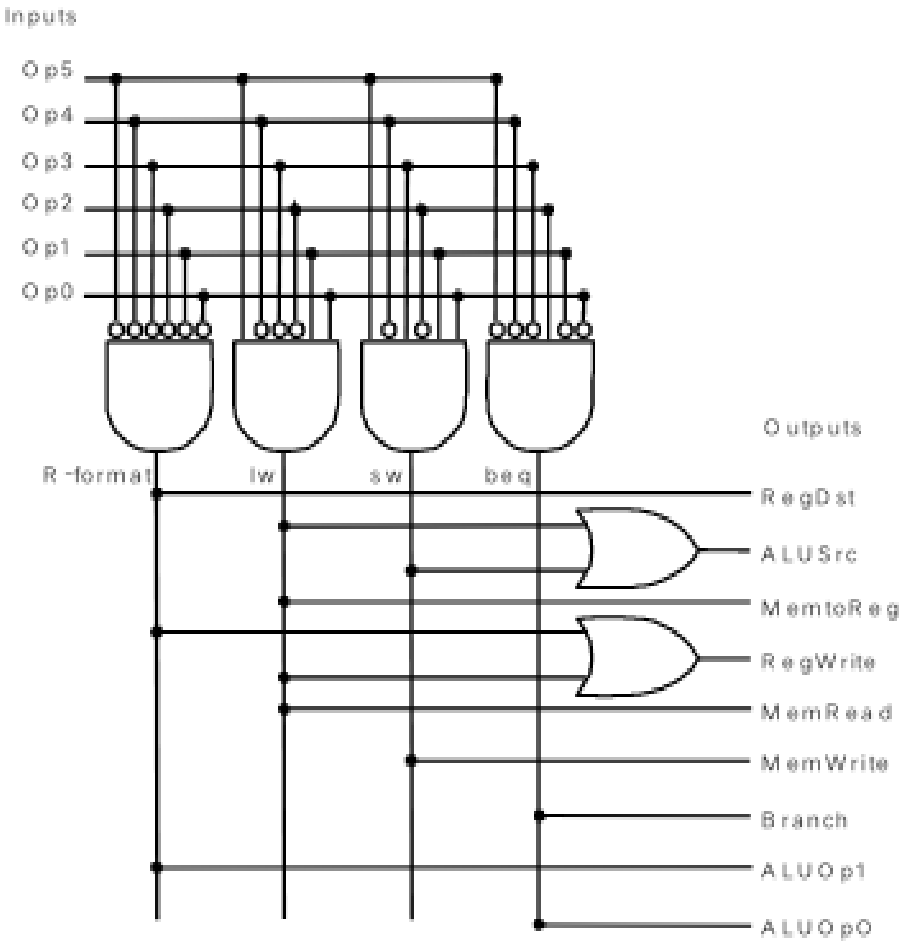


ALU control logic

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

Logi Sim

- All components that we have discussed – and shall discuss – can be fabricated using Logi Sim

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

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* when having 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: Fixed-period clock vs. variable-period clo in a single-cycle implementation

- 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.*

Example (Cont.)

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

Instruction class	Instr. mem.	Register read	ALU oper.	Data mem.	Register write	FPU add/ sub	FPU mul/ div
Load word	2	1	2	2	1		
Store word	2	1	2	2			
R-format	2	1	2	0	1		
Branch	2	1	2				
Jump	2						
FP mul/div	2	1		1			16
FP add/sub	2	1		1		8	

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.
- 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*!

Next

- Multi Cycle

Implementing MIPS: the Fetch/Execute Cycle

- High-level abstract view of *fetch/execute* implementation
  - use the program counter (PC) to read instruction address
  - *fetch* the instruction from memory and increment PC
  - use fields of the instruction to select registers to read
  - *execute* depending on the instruction
  - repeat…