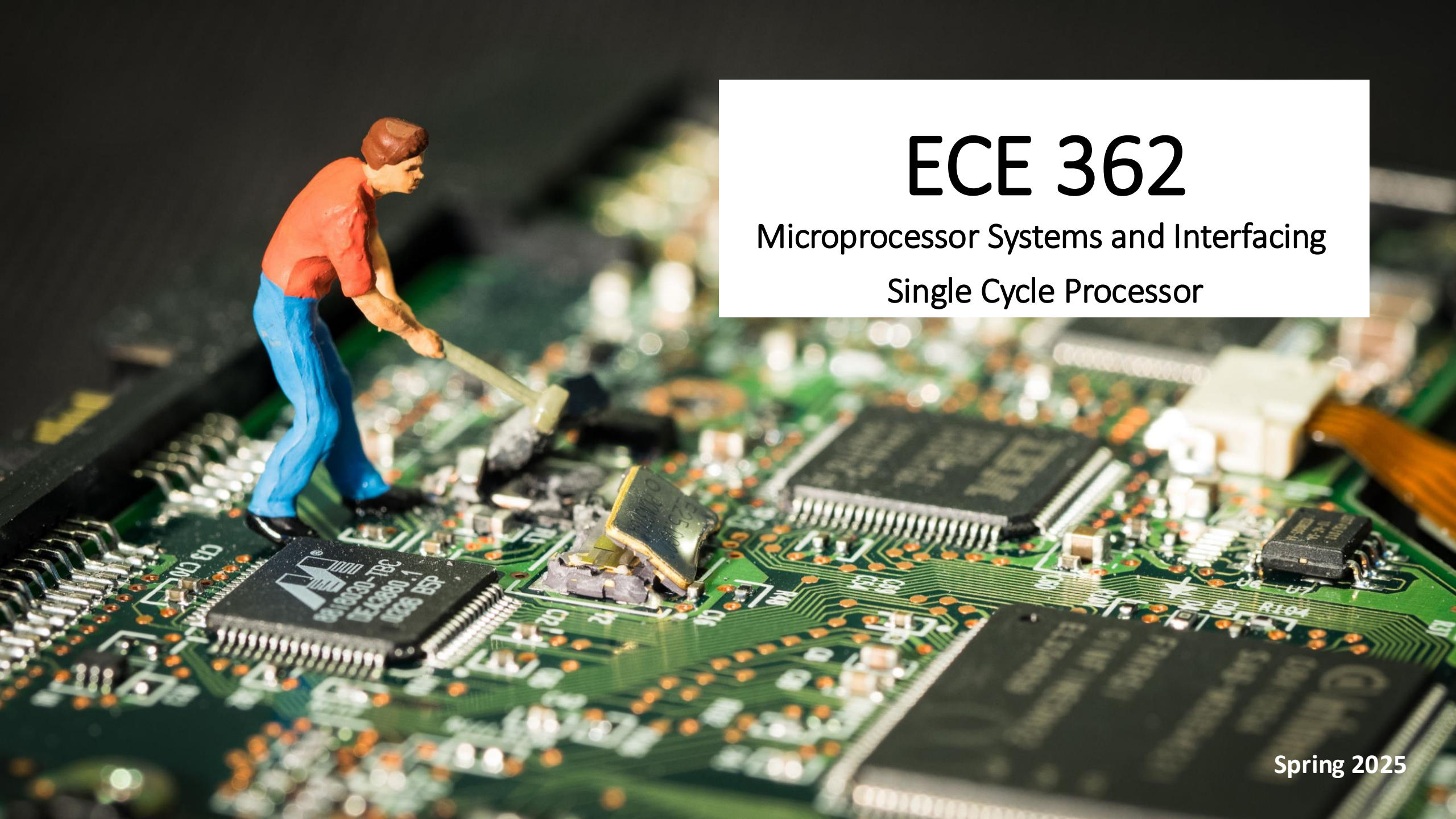


ECE 362

Microprocessor Systems and Interfacing

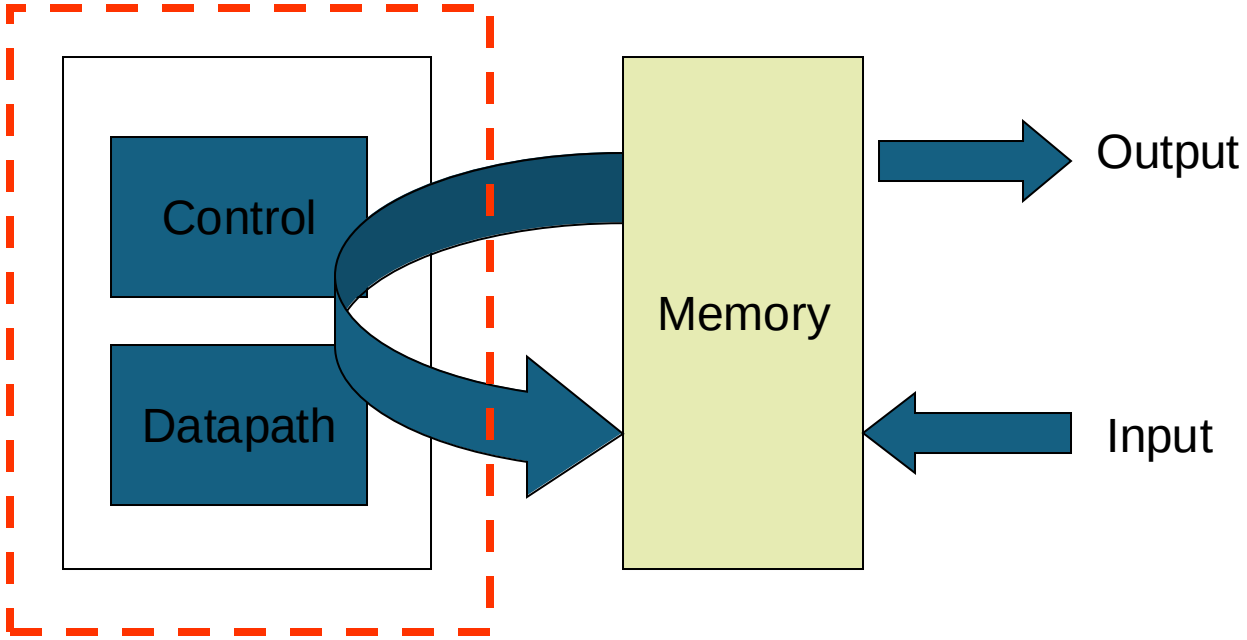
Single Cycle Processor

Spring 2025

Reading

- P&H textbook chapter 4.1-4.4: The Single Cycle Processor

Processor Implementation



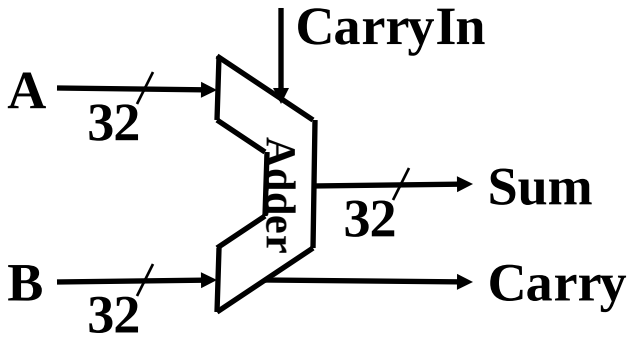
Outline

- Datapath - single cycle
  - single instruction
- Control

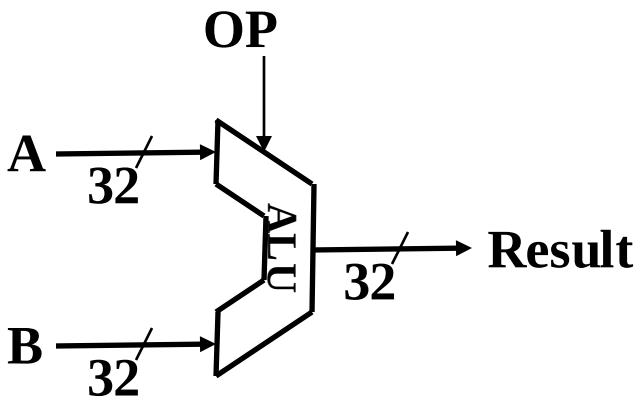
Datapath for Instructions

- Single-cycle datapath
- Compose using well-understood pieces
  - Mux, flip-flops and gates
  - ALU
  - Register file

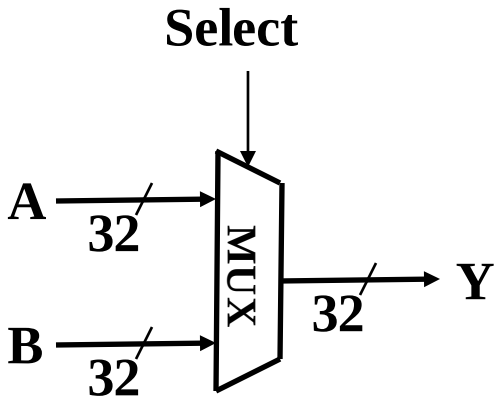
Comb. Logic Elements



Adder

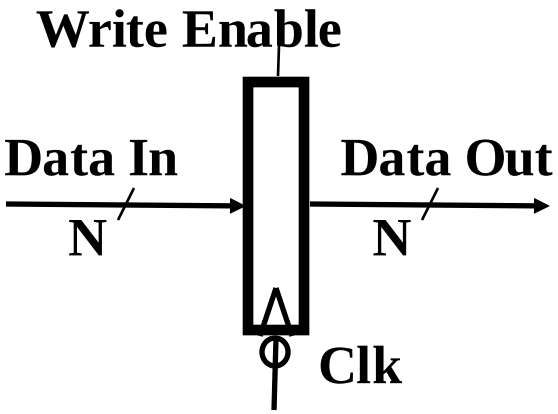


ALU

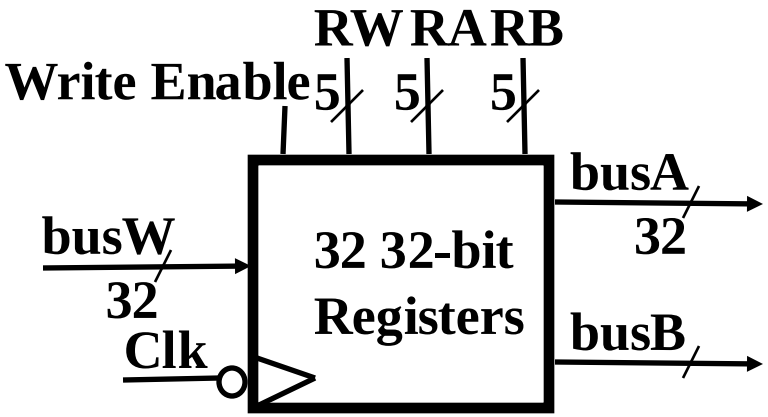


Mux

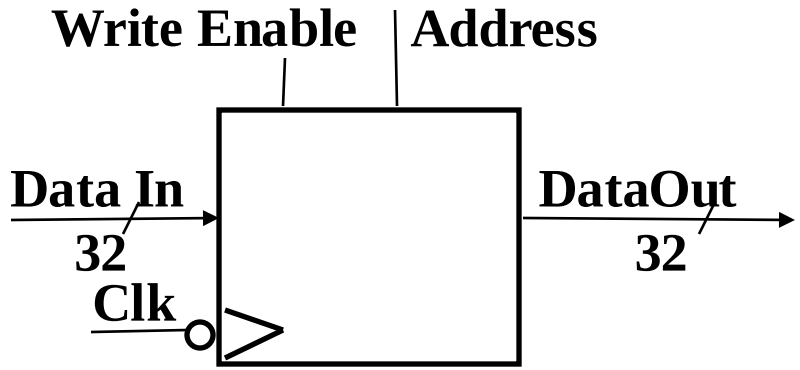
Storage Elements



- Register
  - for PC

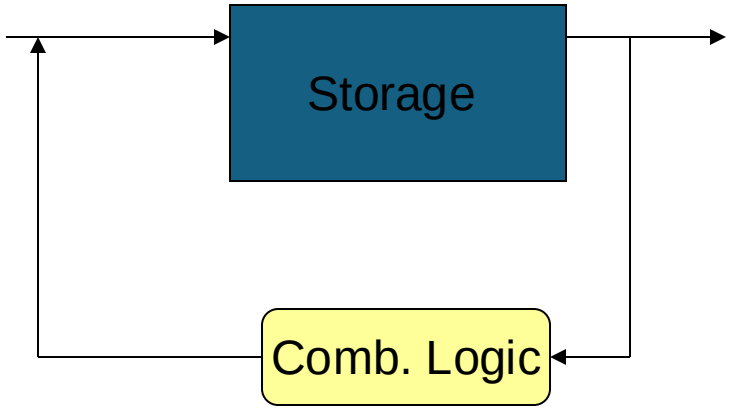


- Register file
  - 32 registers
  - 2 read ports/buses
  - 1 write port/bus



- Memory
  - 1 input bus
  - 1 output bus
  - Not bidirectional

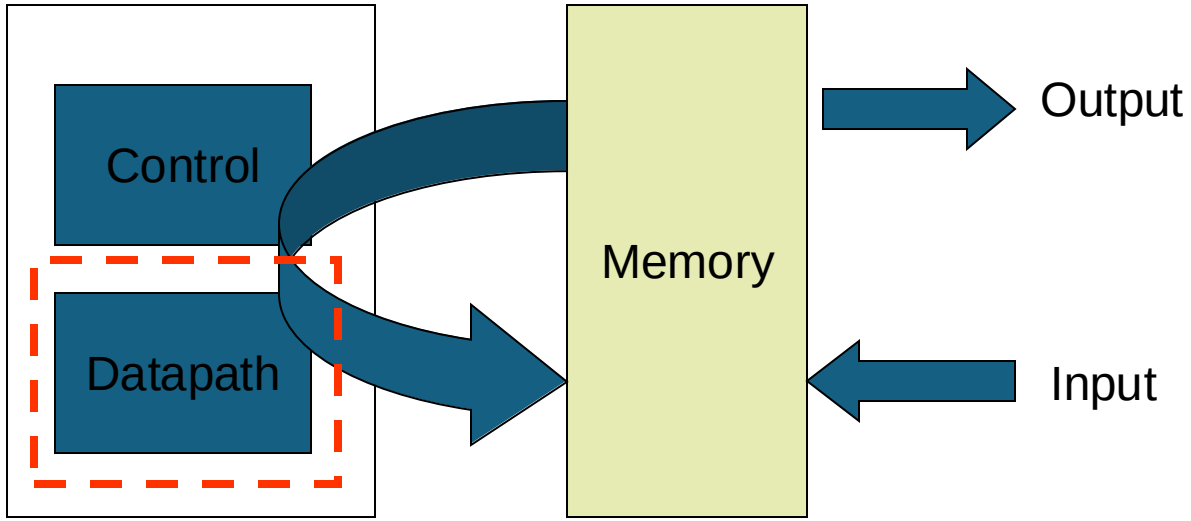
Computer as State Machine



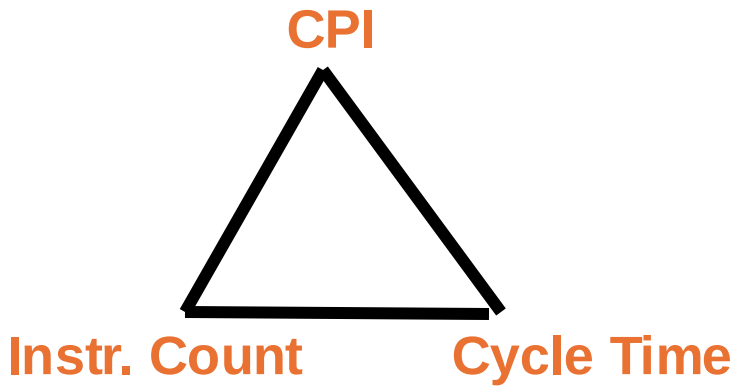
- Storage elements
  - Memory, Register file, PC
- Combinational elements
  - ALUs, Adders, Muxes

Processor Implementation

- This Lecture: Datapath



Processor Implementation

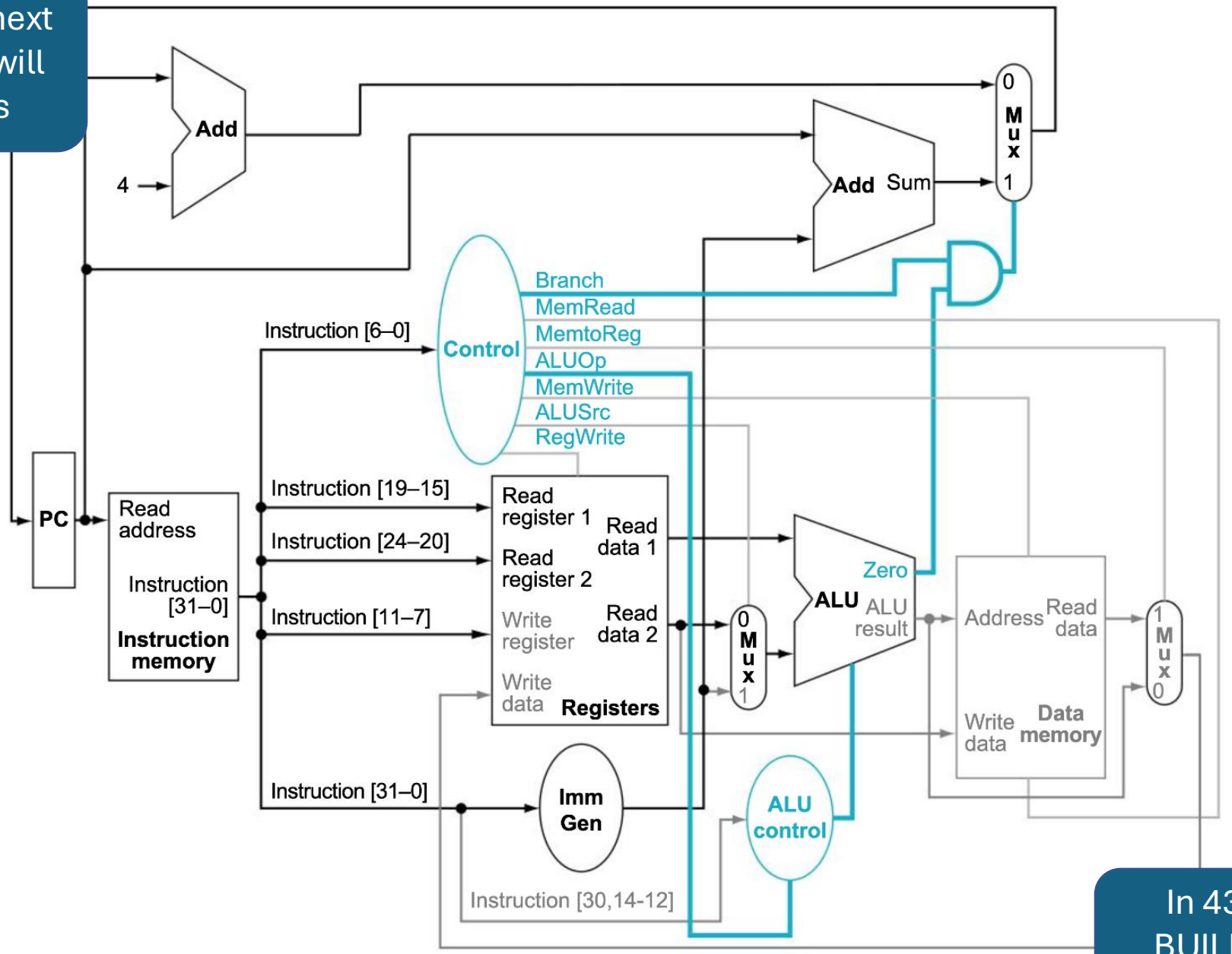


- Implementation determines
  - How many Clocks Per Instruction (CPI)
  - How long is the clock cycle (Cycle time)
- ISA, compiler determine
  - How many instructions in a program (Instr. count)
- 362 is focused on functionality only
  - Performance is the primary topic of 437

Datapath – Single cycle

- Assumption: Get one whole instruction done in one long clock cycle
  - **fetch, decode/read operands, execute, memory, writeback**
    - 5 steps you should NEVER forget!
  - useful way to represent steps and identify required datapath elements: RTL
- For single instruction
- Put it together

By the end of the next few lectures you will understand this



In 437 you will BUILD this! And much, much more.

A Simple Implementation

- add and sub
  - add rd, rs1, rs2
  - addi rd, rs1, imm12
  - sub rd, rs1, rs2
- lw/sw
  - lw rd, imm12(rs1)
  - sw rs2, imm12(rs1)
- branch:
  - beq rs1, rs2, imm12
- Logical ops:
  - and rd, rs1, rs2
  - andi rd, rs1, imm12
  - or rd, rs1, rs2
  - ori rs, rs1, imm12

Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

Register Transfer Language

- RTL gives the meaning of the instructions
- All start by fetching the instruction

fun7 | rs2 | rs1 | fun3 | rd | op = MEM[PC] // r-type

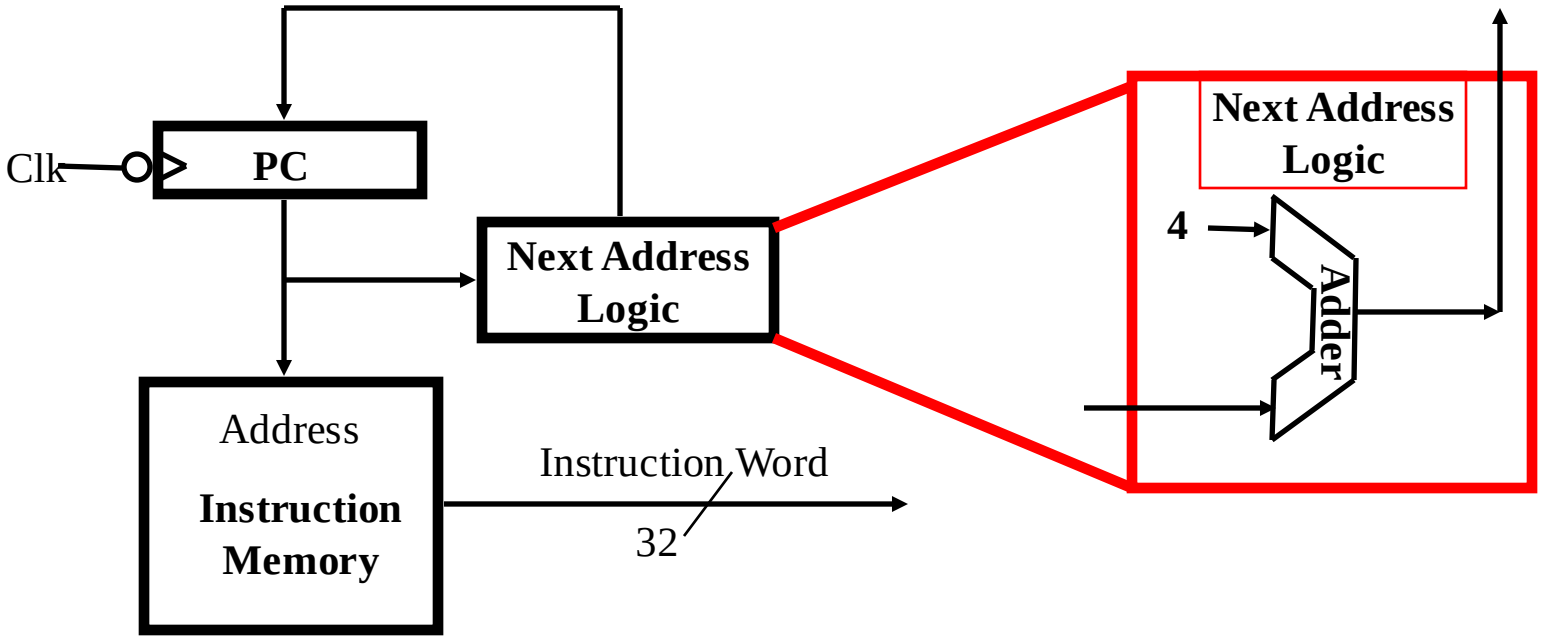
imm12 | rs1 | fun3 | rd | op = MEM[PC] // i-type

imm7 | rs2 | rs1 | fun3 | imm5 | op = MEM[PC] // s-type/sb-type

inst	Register Transfers	
ADD	R[rd] <– R[rs1] + R[rs2];	PC <– PC + 4
SUB	R[rd] <– R[rs1] – R[rs2];	PC <– PC + 4
LOAD	R[rd] <– MEM[R[rs1] + sign_ext(imm12)];	PC <– PC + 4
STORE	MEM[R[rs1] + sign_ext(imm7 || imm5)] <– R[rs2];	PC <– PC + 4
BEQ	if (R[rs1] == R[rs2]) then PC <– PC + sign_ext(imm7 || imm5)* || 0 else PC <– PC + 4	

Fetch Instructions

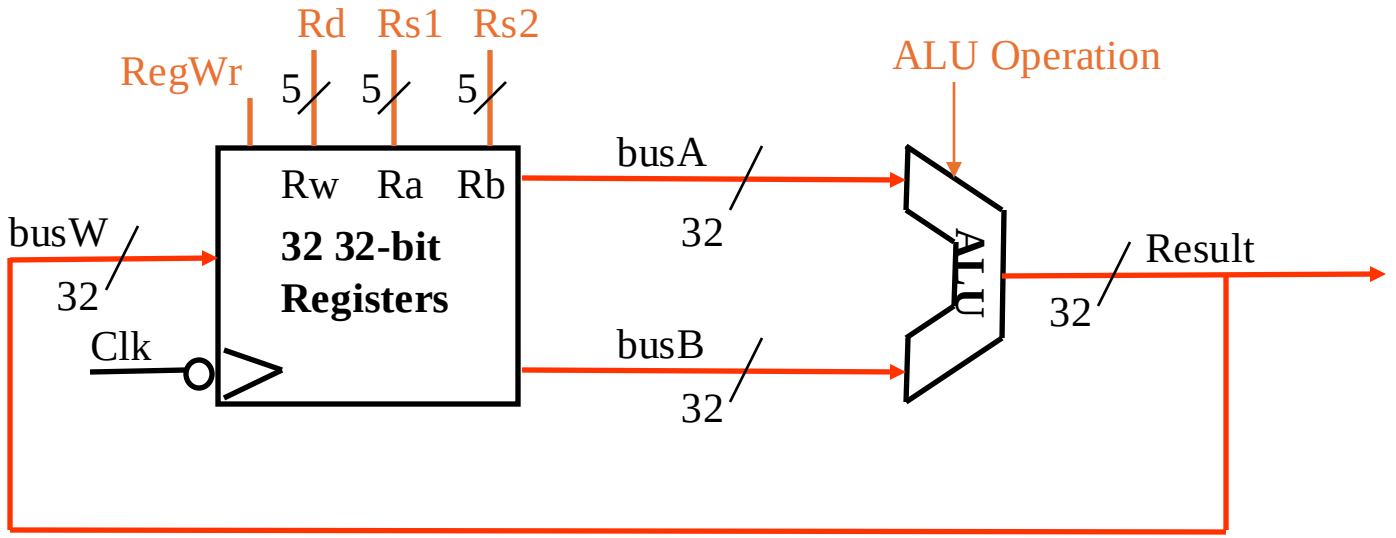
- Fetch instruction, then update PC
- PC updated (at the end of) every cycle
  - What if no branches or jumps?



ALU Instructions

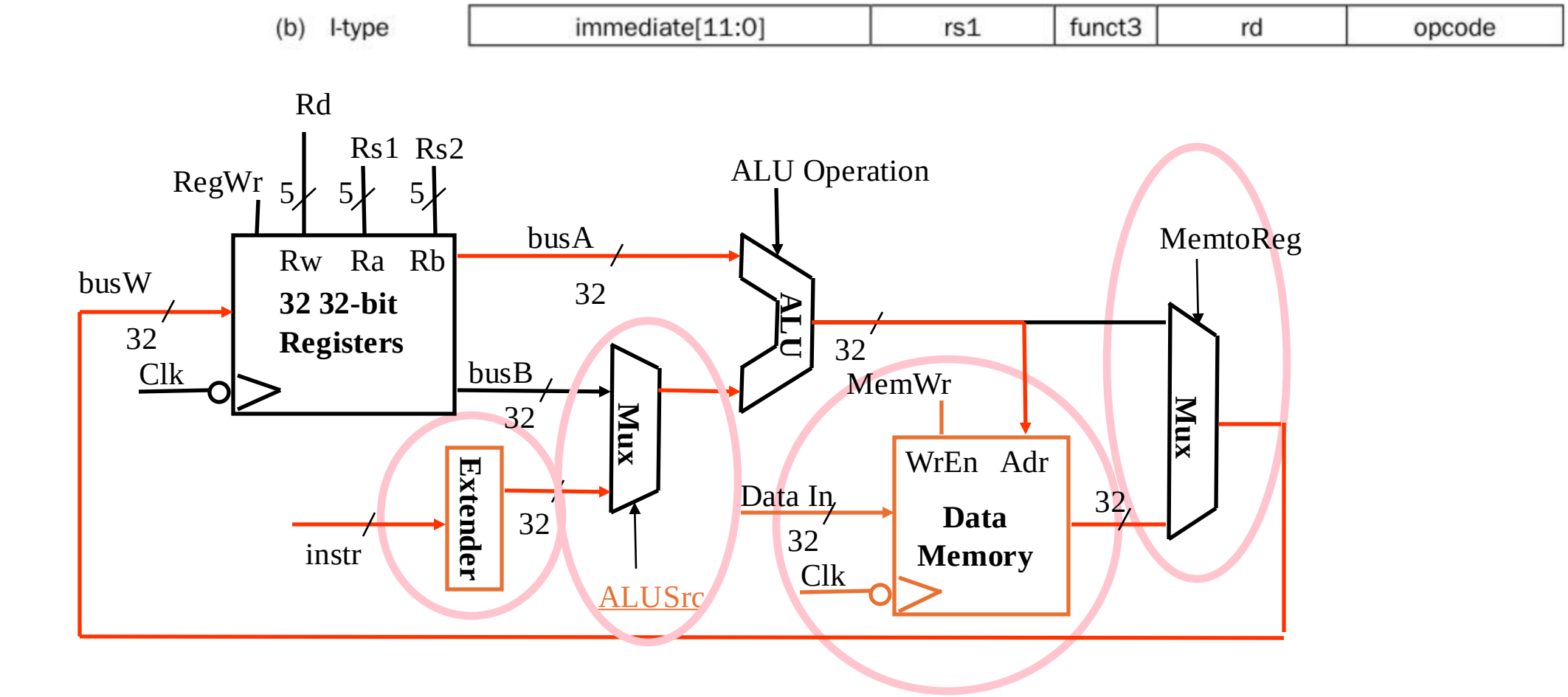
- R[rd] <- R[rs1] op R[rs2] Example: add rd, rs1, rs2
  - Ra, Rb, and Rw come from instruction's rs1, rs2, and rd fields
  - ALUOperation and RegWr: control logic after decoding the instruction

Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode



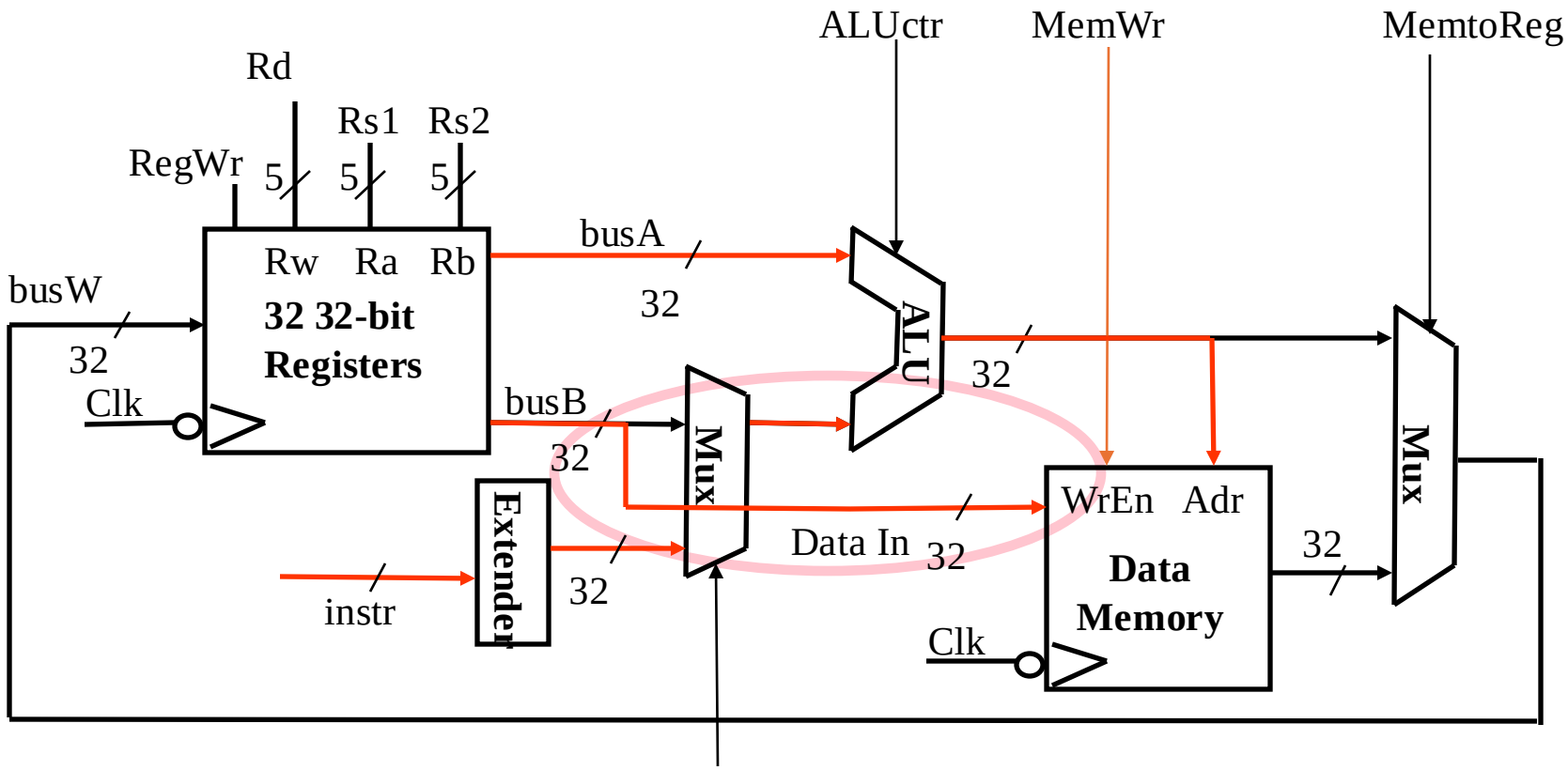
Load Instruction

- R[rd] <- Mem[R[rs1] + SignExt[imm12]]
- Example: lw rd, imm12(rs1)



Store Instruction

- Mem[R[rs1] + SignExt[imm11:5 || imm4:0]] <- R[rs2] Example: sw rs2, imm12(rs1)



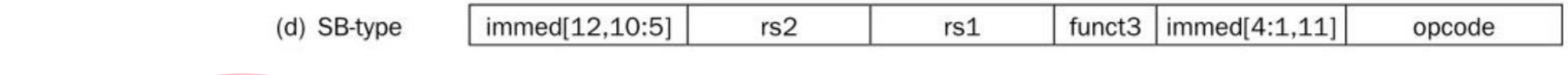
Conditional Branch Instruction

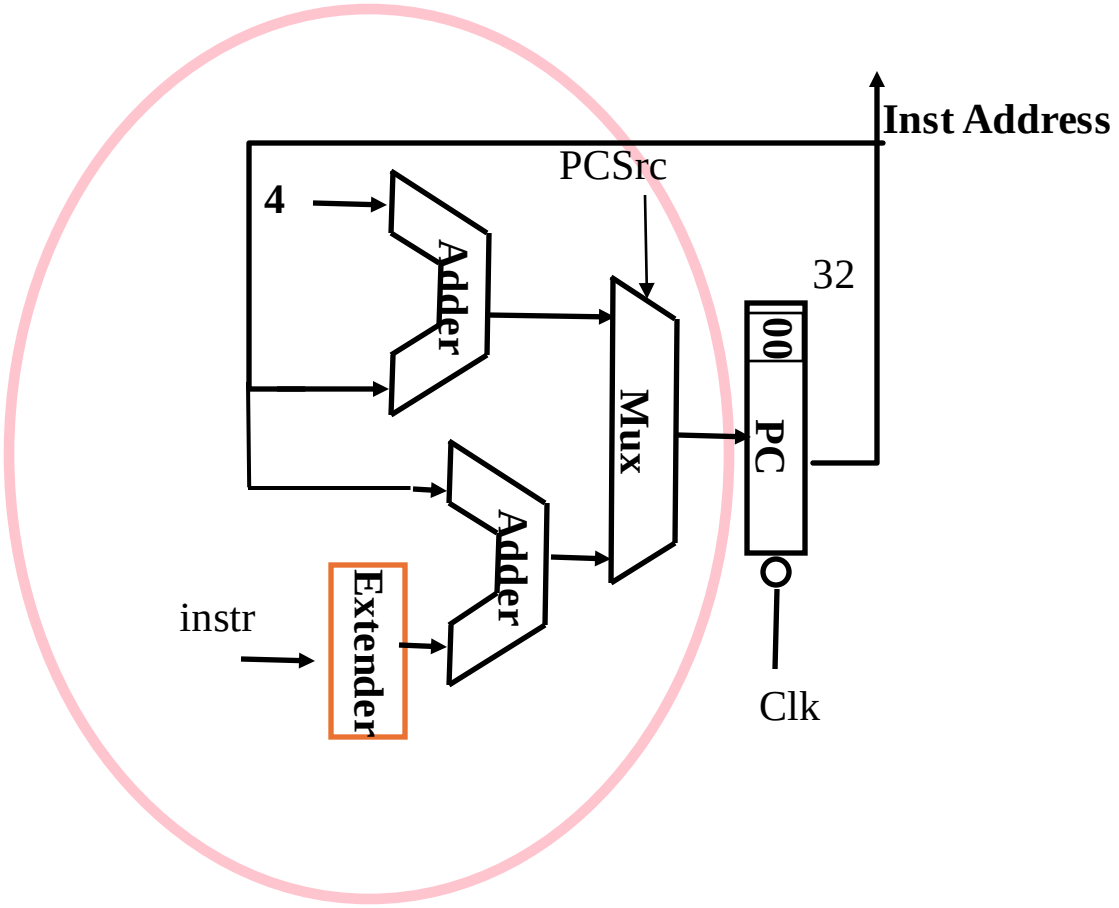
- beq rs1, rs2, imm12
  - IR = Mem[PC] // Fetch the instruction from memory
  - if (R[rs1] == R[rs2]) // Calculate the next instruction's address
    - PC <- PC + (SignExt(imm12) << 1)
  - else
    - PC <- PC + 4

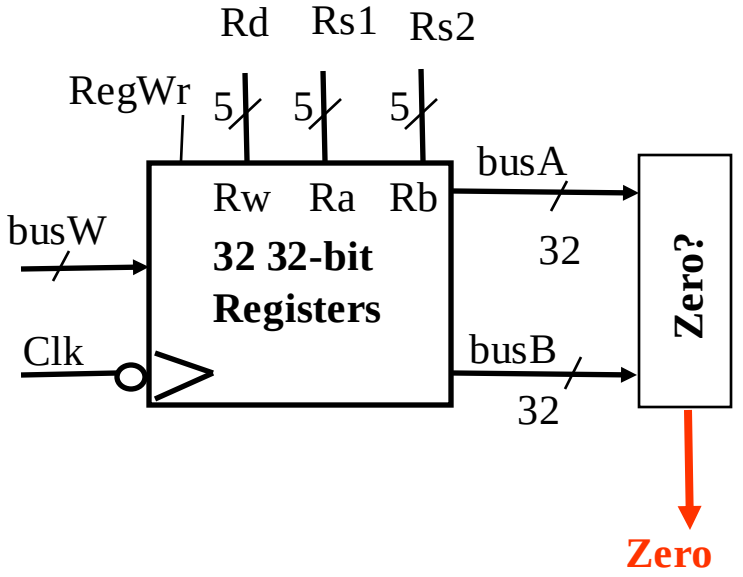
- Branches compute TWO things: branch condition and branch target

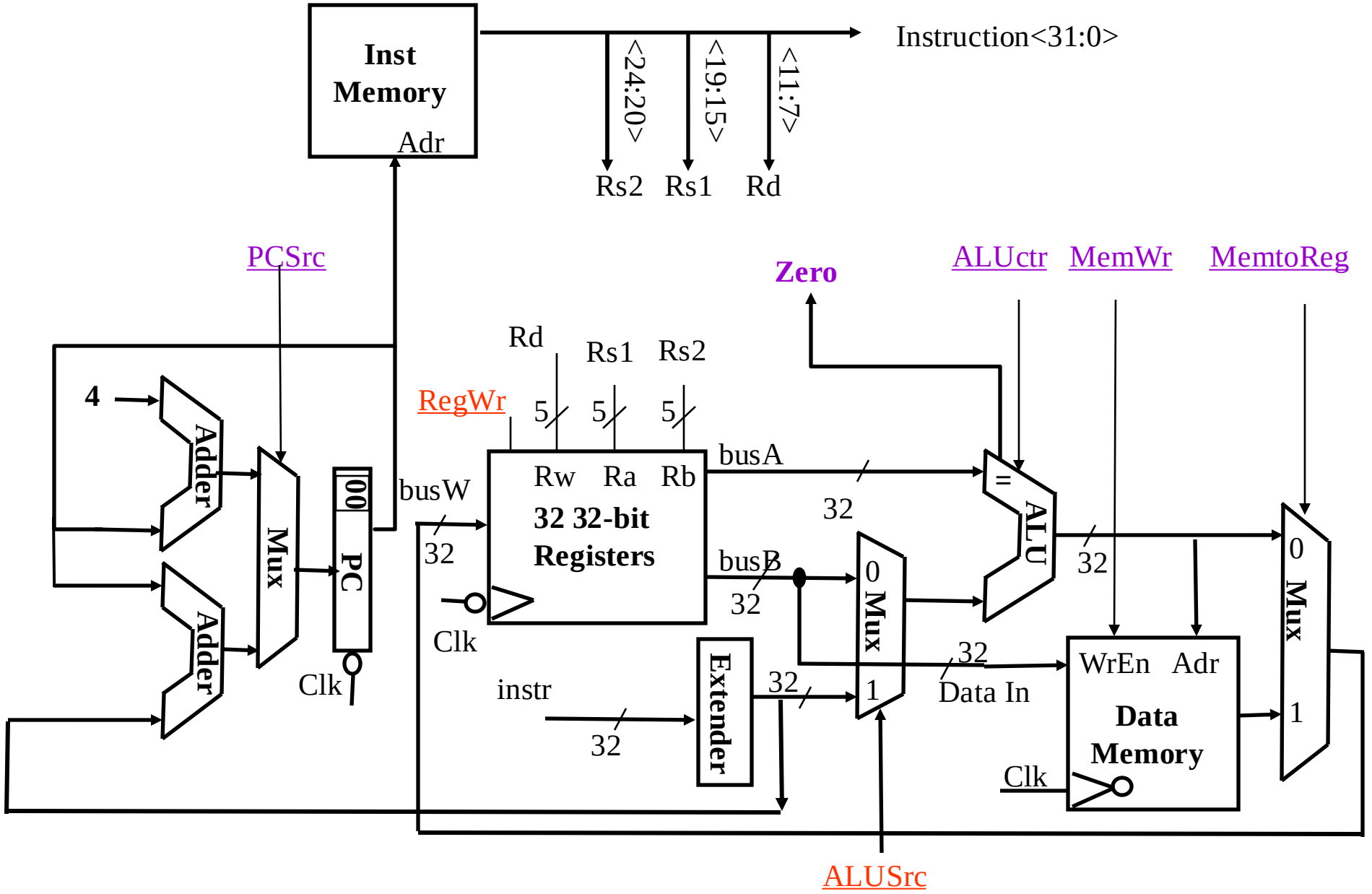
Datapath for ‘beq’

- beq rs1, rs2, imm12
  - Datapath generates branch condition (zero)








Inst Memory
Adr
Instruction<31:0>
<24:20>
<19:15>
<11:7>
Rs2
Rs1
Rd
PCSrc
Zero
ALUctr
MemWr
MemtoReg
4
Adder
Adder
Mux
00
PC
Clk
Rd
Rs1
Rs2
RegWr
5
5
5
busW
32
Rw
Ra
Rb
32 32-bit Registers
Clk
busA
32
busB
32
0
Mux
1
=
ALU
32
0
Mux
1
instr
32
Extender
32
32
Data In
WrEn
Adr
Data Memory
Clk
ALUSrc

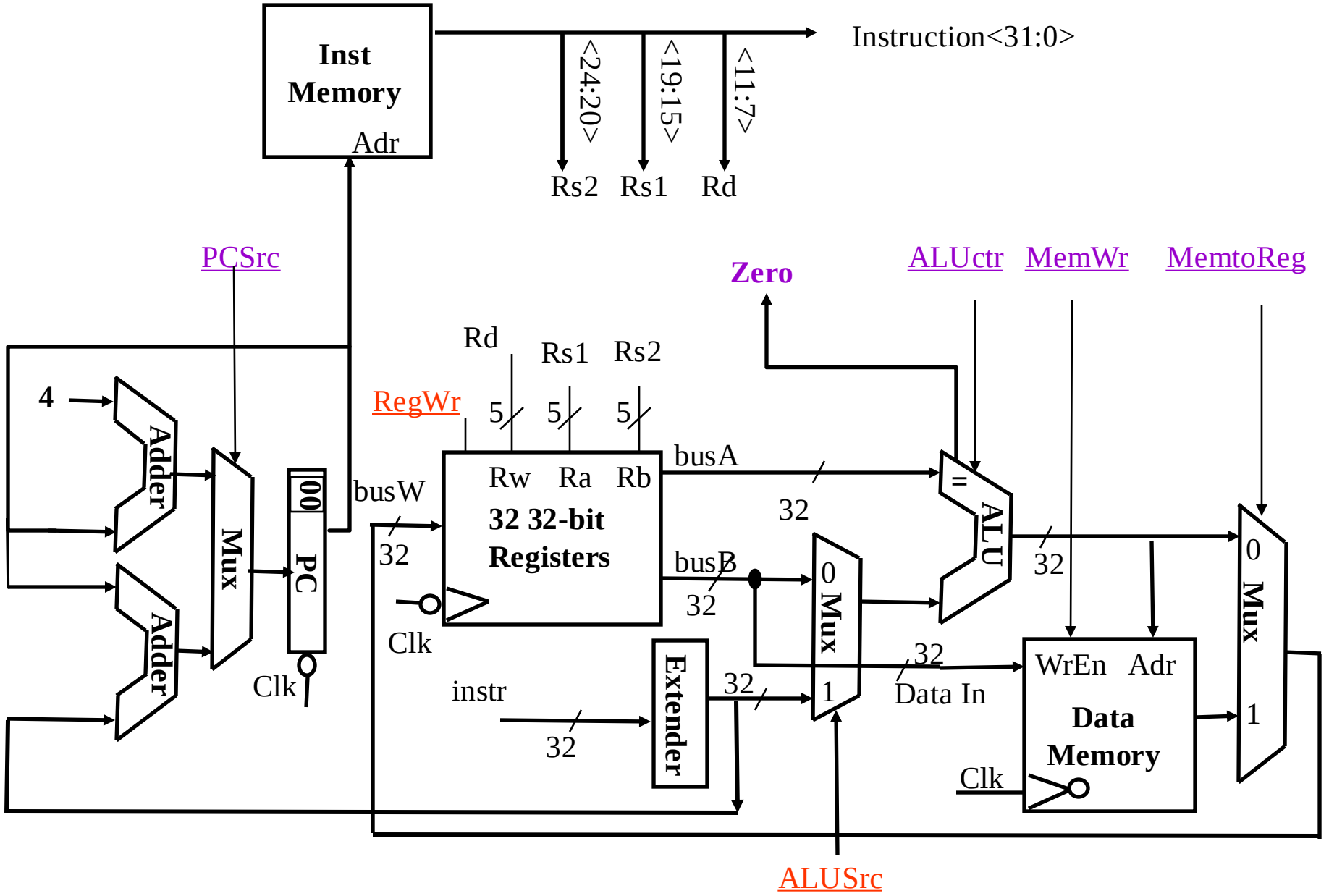
Summary

- For a given instruction
  - Describe operation in RTL
  - Use ALUs, Registers, Memory, adders to achieve required functionality
- To add instructions
  - Rinse and repeat; Reuse components via muxes
  - Not all blocks are used by all instructions so you need to ensure unused blocks don’t interfere
- Control: next
  - Selection controls for muxes
  - ALU controls for ALU ops
  - Register address controls
  - Write enables for registers/memory

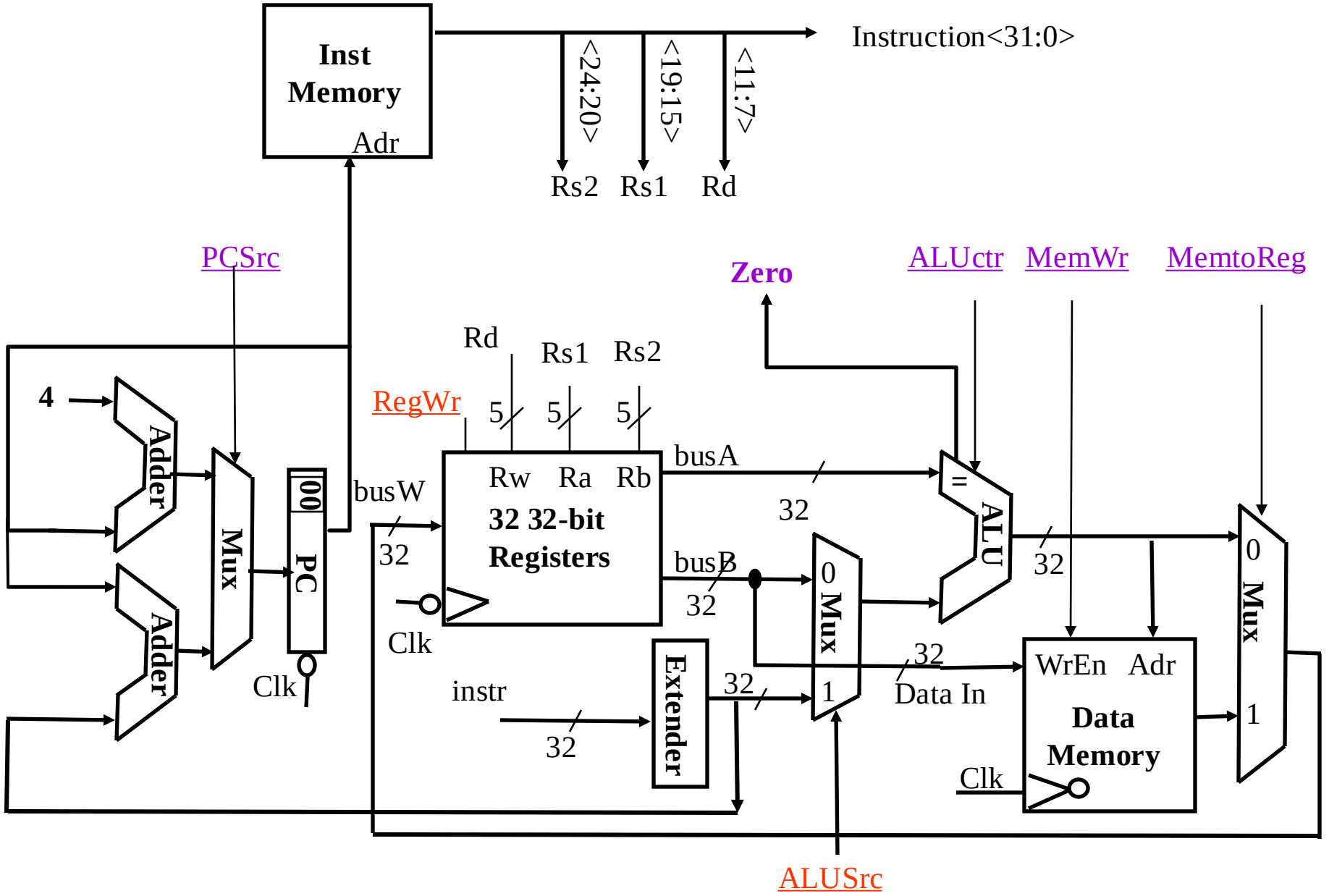
Exercise

- See worksheet
- Highlight active datapath for
  - add
  - beq
  - sw
  - lw

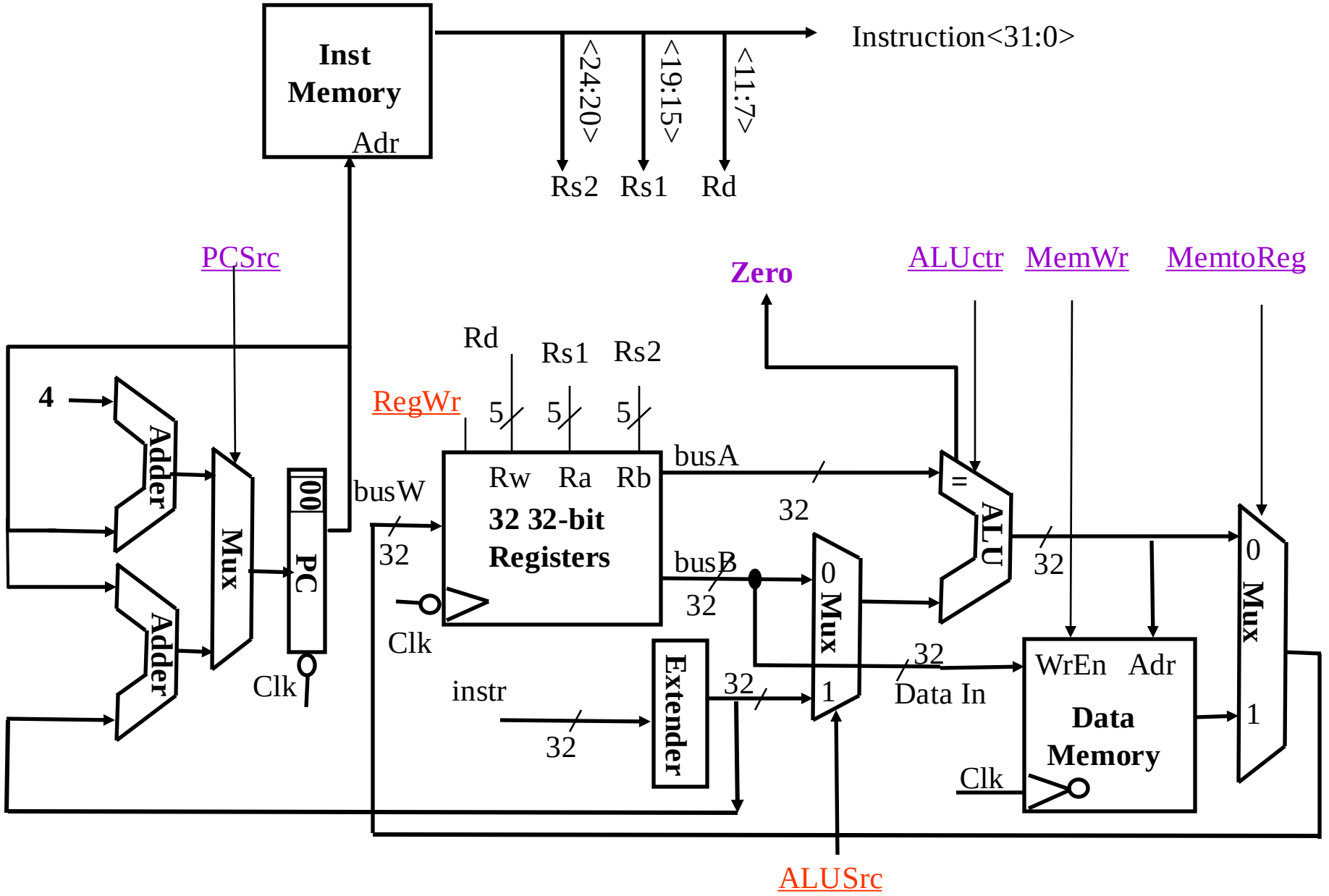
add


Inst Memory
Adr
Instruction<31:0>
<24:20>
<19:15>
<11:7>
Rs2
Rs1
Rd
PCSrc
Zero
ALUctr
MemWr
MemtoReg
Rd
Rs1
Rs2
RegWr
5
5
5
4
Adder
Adder
Mux
00
PC
Clk
busW
32
Rw
Ra
Rb
32 32-bit Registers
Clk
busA
32
busB
32
0
Mux
1
=
ALU
32
0
Mux
1
instr
32
Extender
32
32
Data In
WrEn
Adr
Data Memory
Clk
ALUSrc

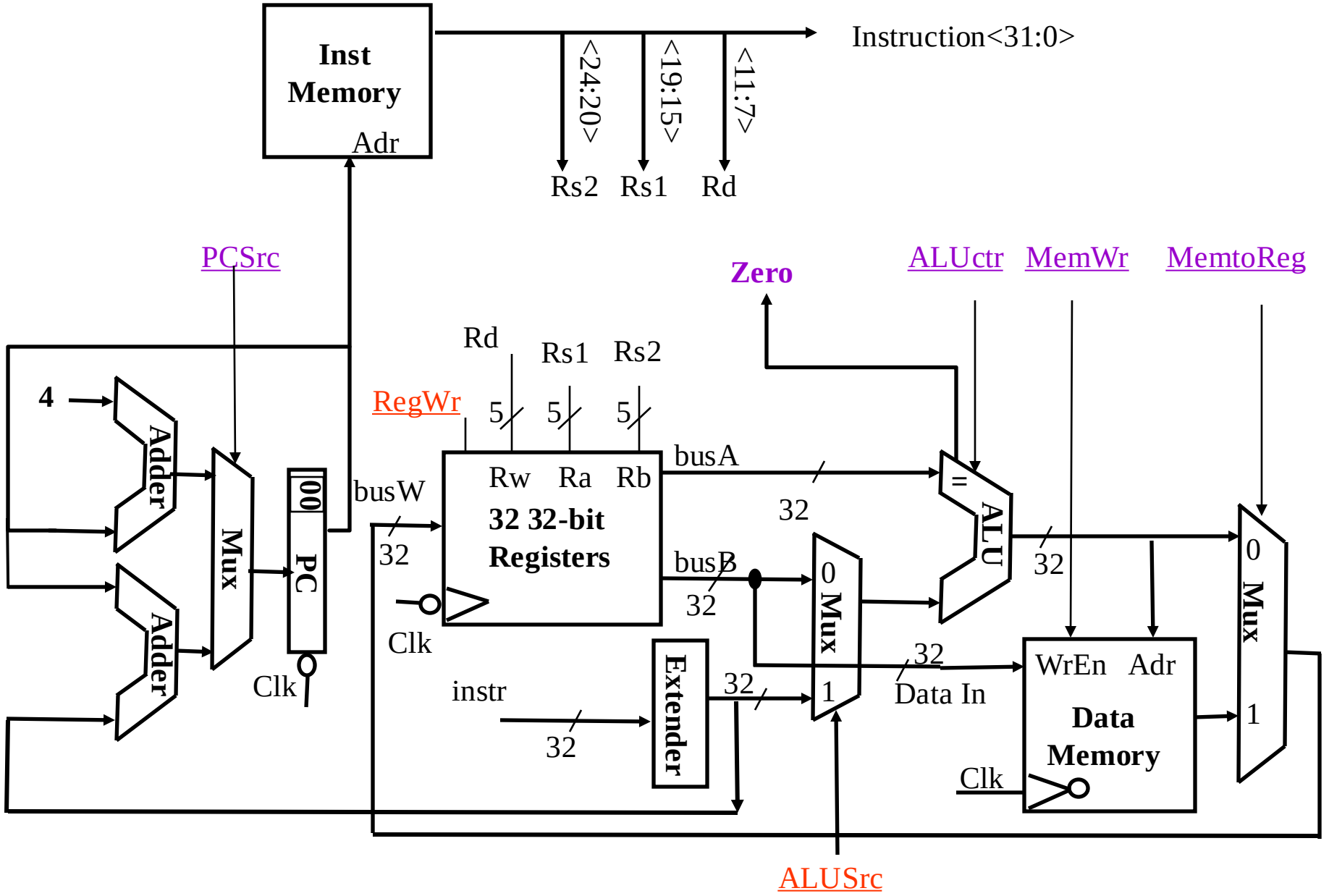
beq


Inst Memory
Adr
Instruction<31:0>
<24:20>
<19:15>
<11:7>
Rs2
Rs1
Rd
PCSrc
Zero
ALUctr
MemWr
MemtoReg
4
Adder
Adder
Mux
00
PC
Clk
busW
32
Rd
Rs1
Rs2
RegWr
5
5
5
Rw
Ra
Rb
32 32-bit Registers
busA
32
busB
32
Clk
=
ALU
32
0
Mux
1
instr
32
Extender
32
32
Data In
WrEn
Adr
Data Memory
Clk
0
Mux
1
ALUSrc

SW

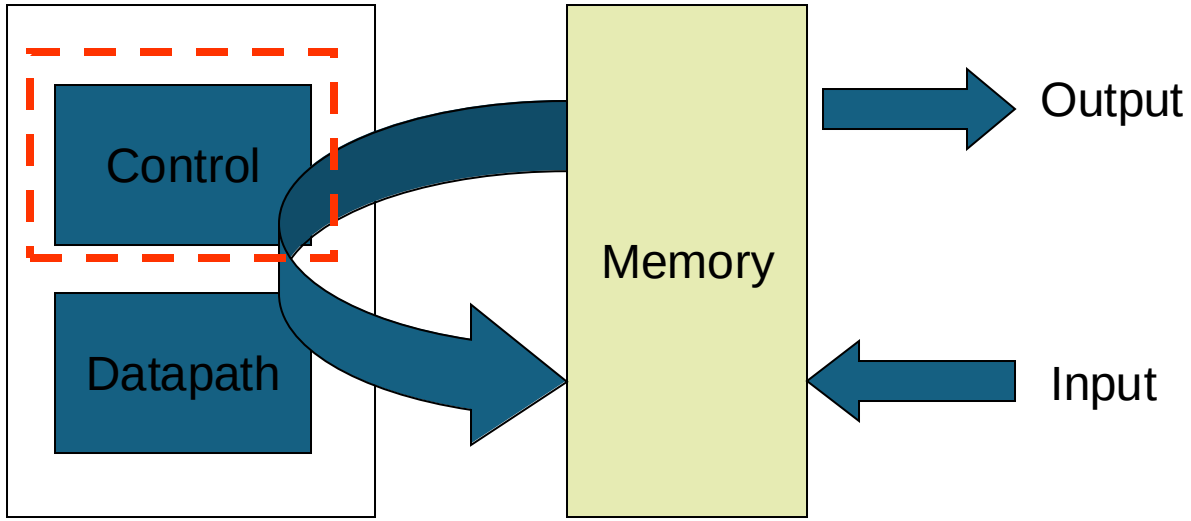

Inst Memory
Adr
Instruction<31:0>
<24:20>
<19:15>
<11:7>
Rs2
Rs1
Rd
PCSrc
Zero
ALUctr
MemWr
MemtoReg
4
Adder
Adder
Mux
00
PC
Clk
busW
32
Rd
Rs1
Rs2
RegWr
5
5
5
Rw
Ra
Rb
32 32-bit Registers
Clk
busA
32
busB
32
0
Mux
1
=
ALU
32
0
Mux
1
instr
32
Extender
32
32
Data In
WrEn
Adr
Data Memory
Clk
ALUSrc

lw

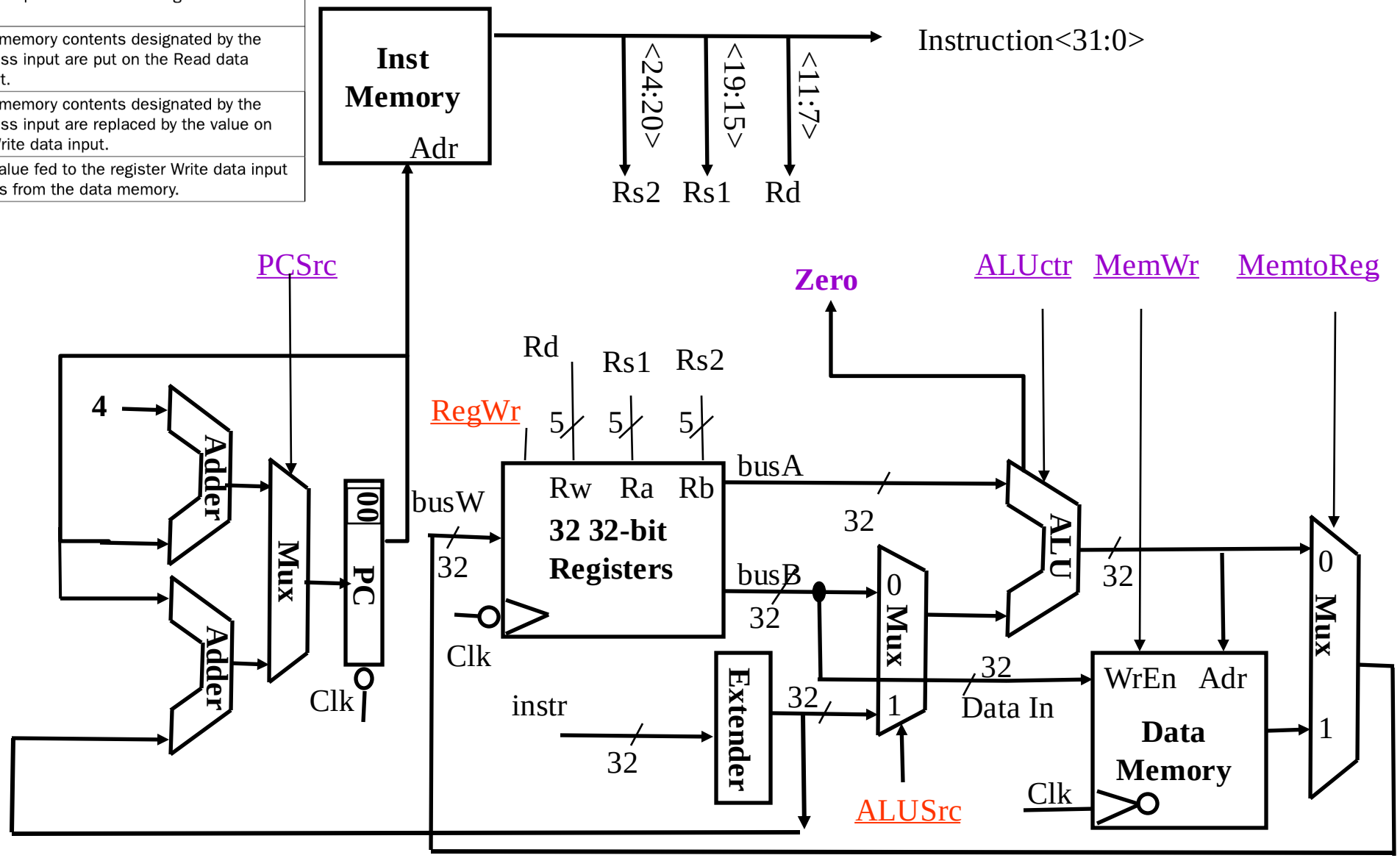

Inst Memory
Adr
Instruction<31:0>
<24:20>
<19:15>
<11:7>
Rs2
Rs1
Rd
PCSrc
Zero
ALUctr
MemWr
MemtoReg
4
Adder
Adder
Mux
00
PC
Clk
busW
Rd
Rs1
Rs2
RegWr
5
5
5
Rw
Ra
Rb
32 32-bit Registers
busA
32
busB
32
Clk
instr
32
Extender
32
0
Mux
1
=
ALU
32
32
Data In
WrEn
Adr
Data Memory
Clk
0
Mux
1
ALUSrc

Processor Implementation

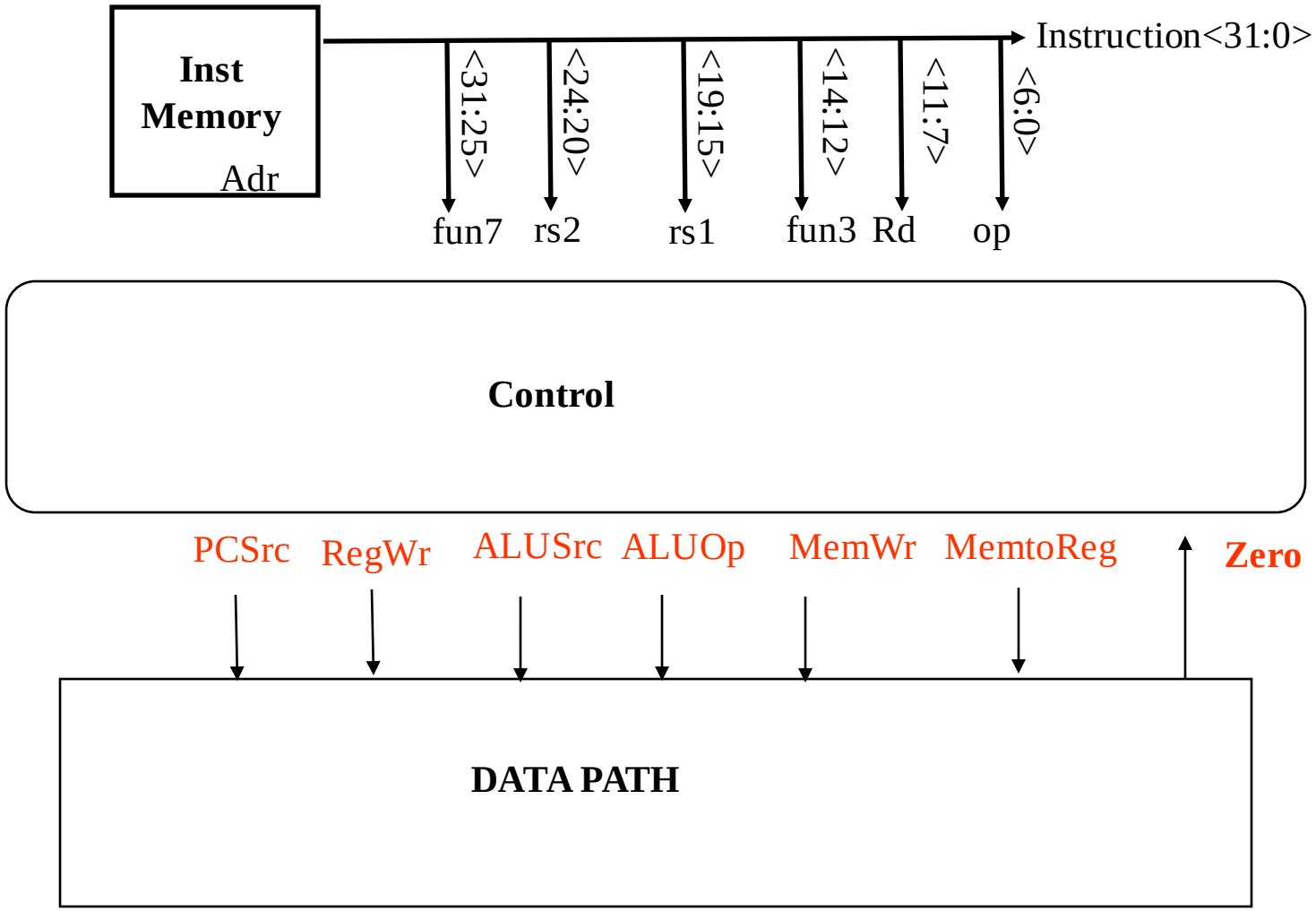
- This Lecture: Control



Signal name	Effect when deasserted	Effect when asserted
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, 12 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 Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on 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.

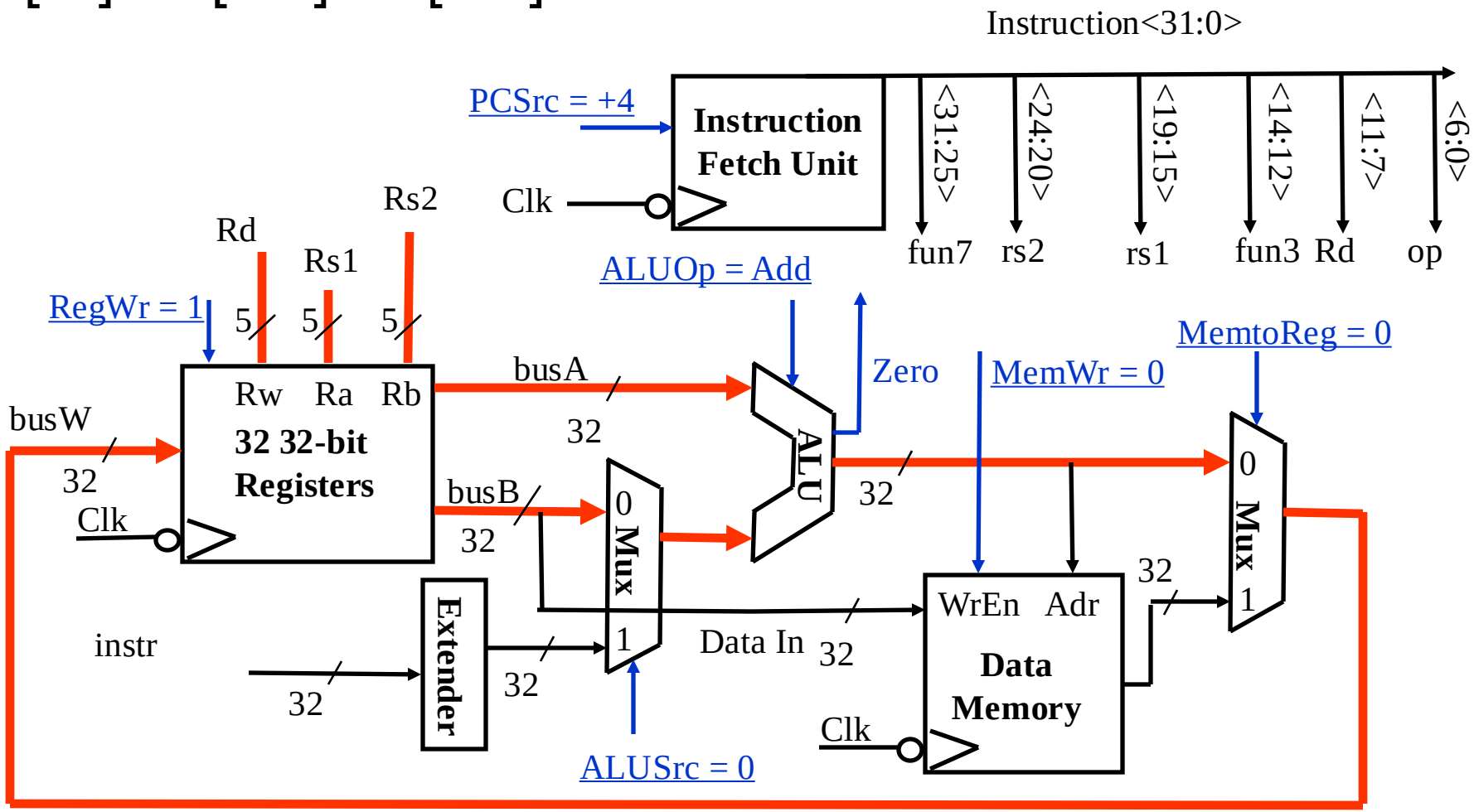


Control for Datapath



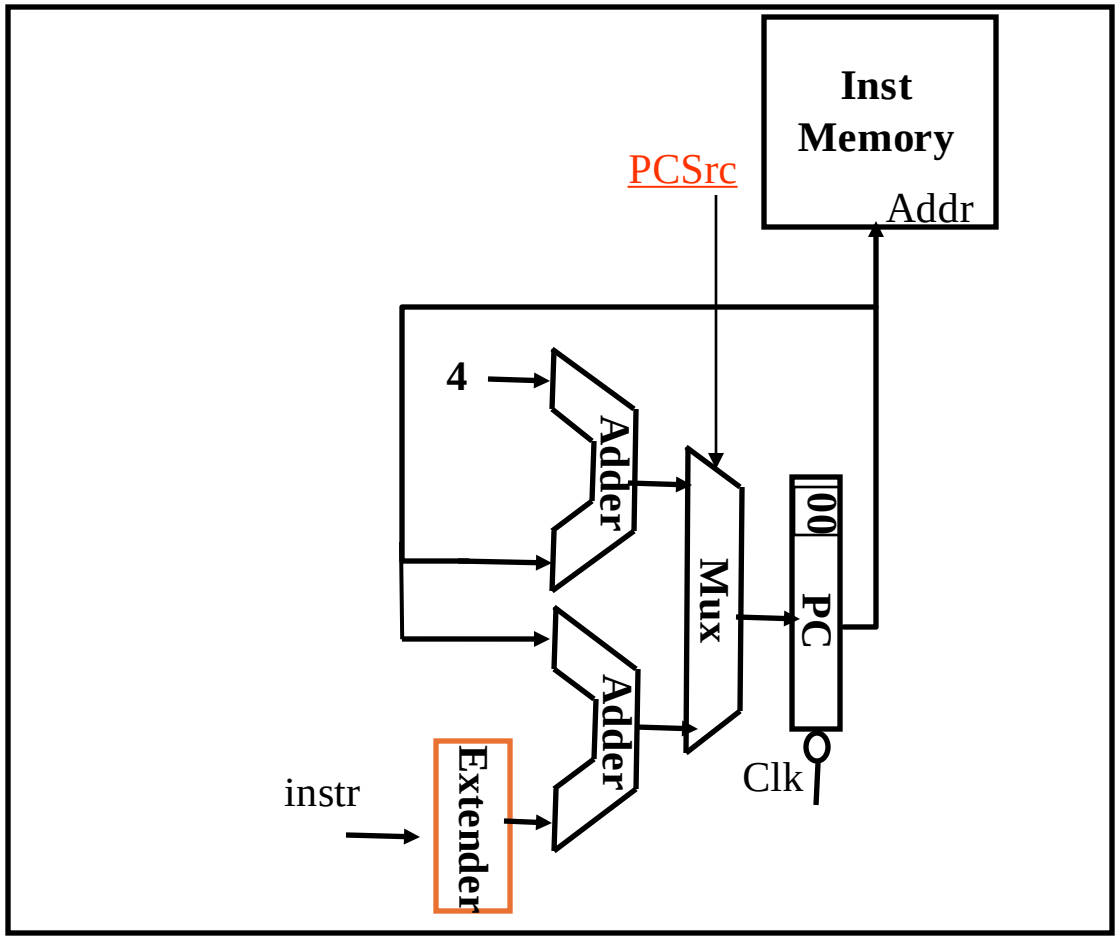
Controls for Add Operation

- R[rd] = R[rs1] + R[rs2]



Meaning of Control Signals

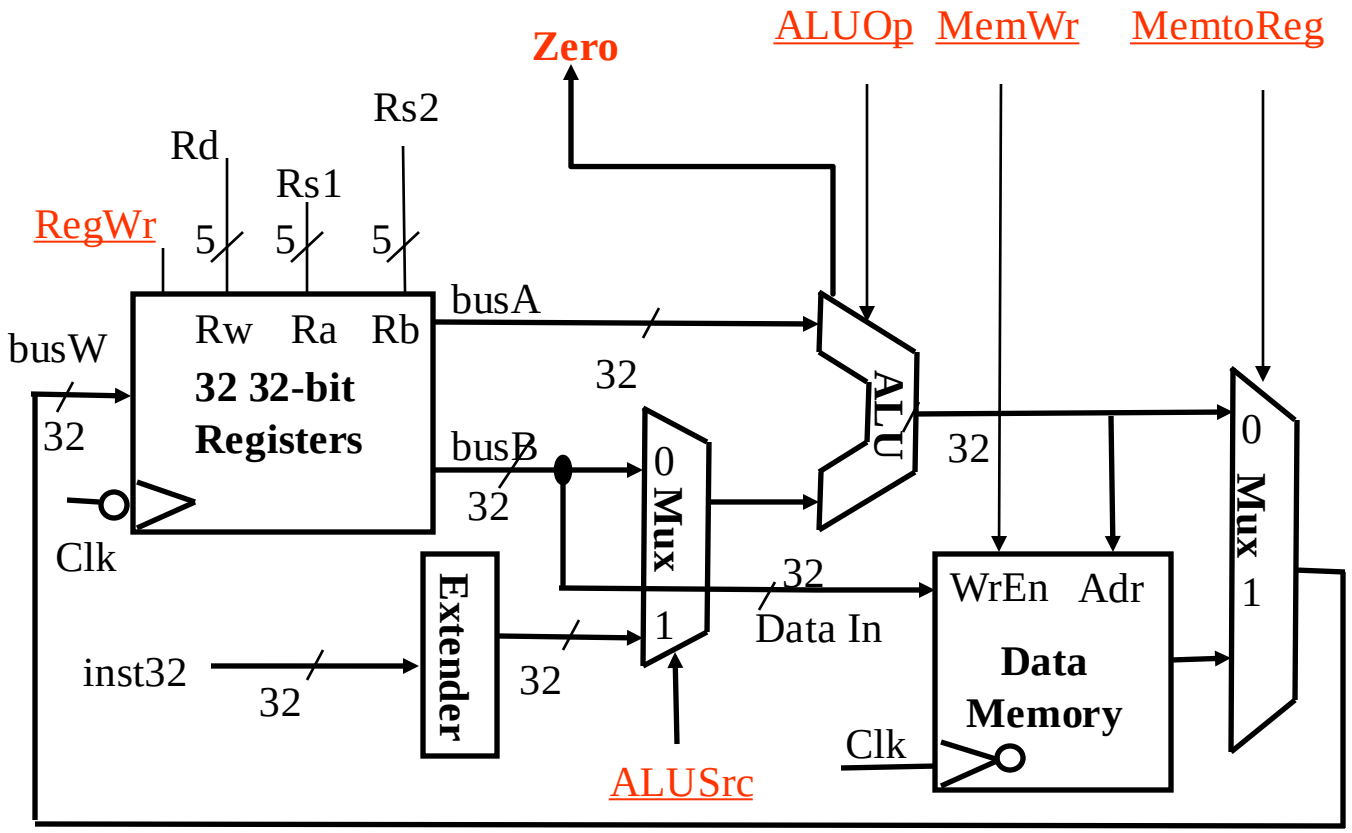
- rs1, rs2 and rd are hardwired in datapath
- PCSrc: 0 => PC <– PC + 4; 1 => PC <– PC + SignExt(imm12) || 0



Instruction Fetch Unit

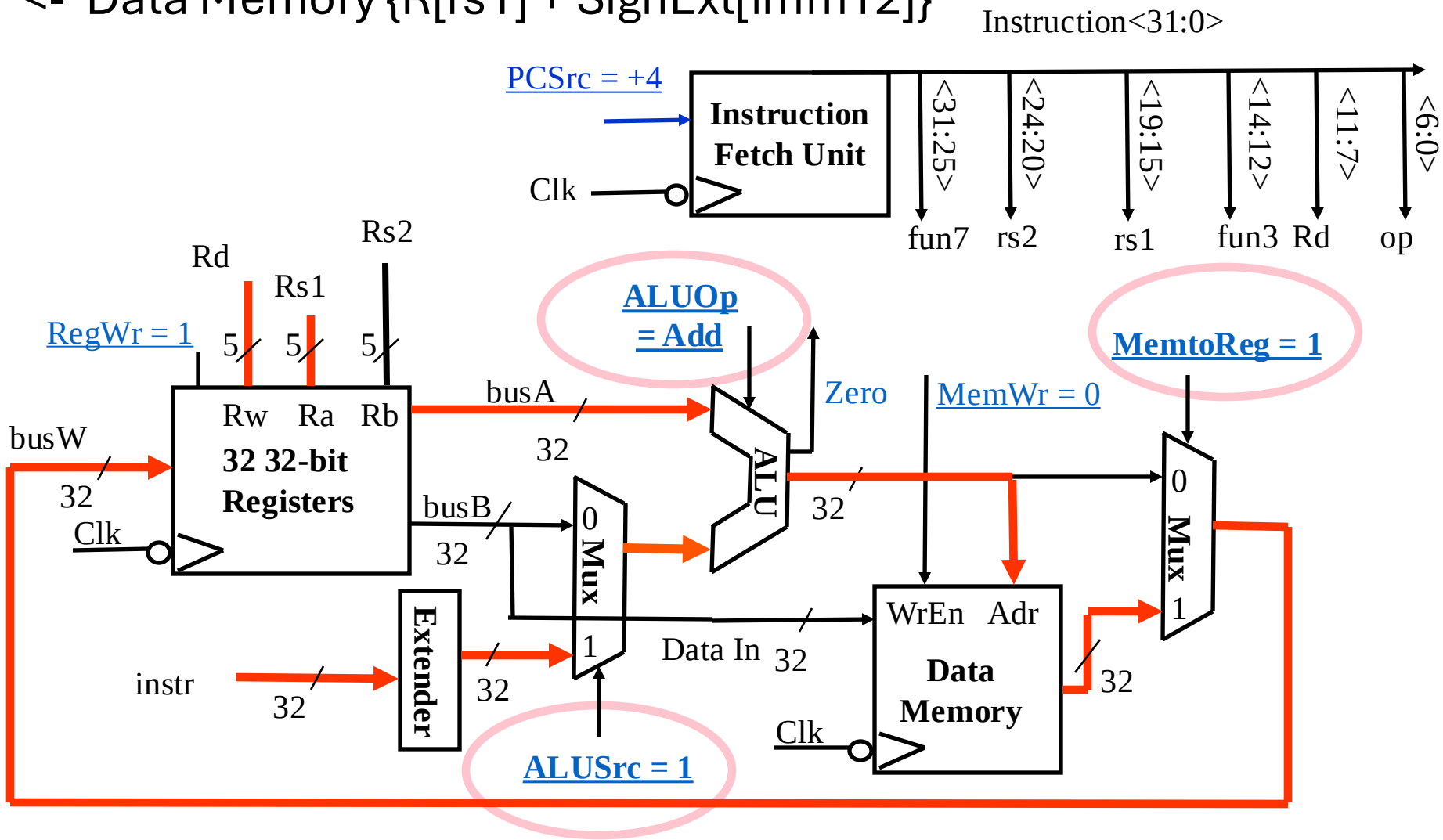
Meaning of Control Signals

- ALUsrc: 0 => regB; 1 => immed
- ALUOp: “add”, “sub”
- MemWr: write memory
- MemtoReg: 1=>Mem
- RegWr: write dest register



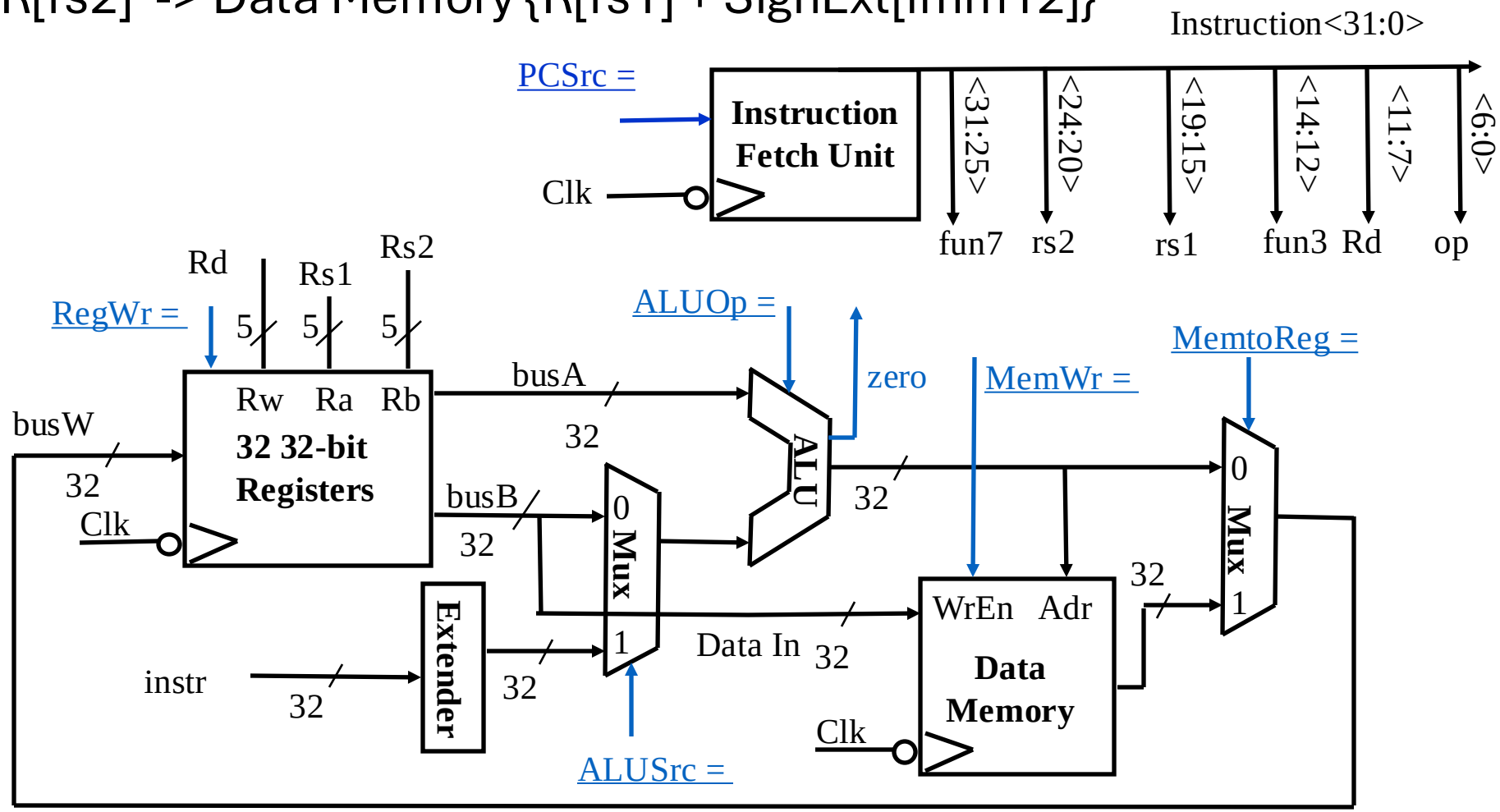
lw Controls

- R[rd] <- Data Memory {R[rs1] + SignExt[imm12]}



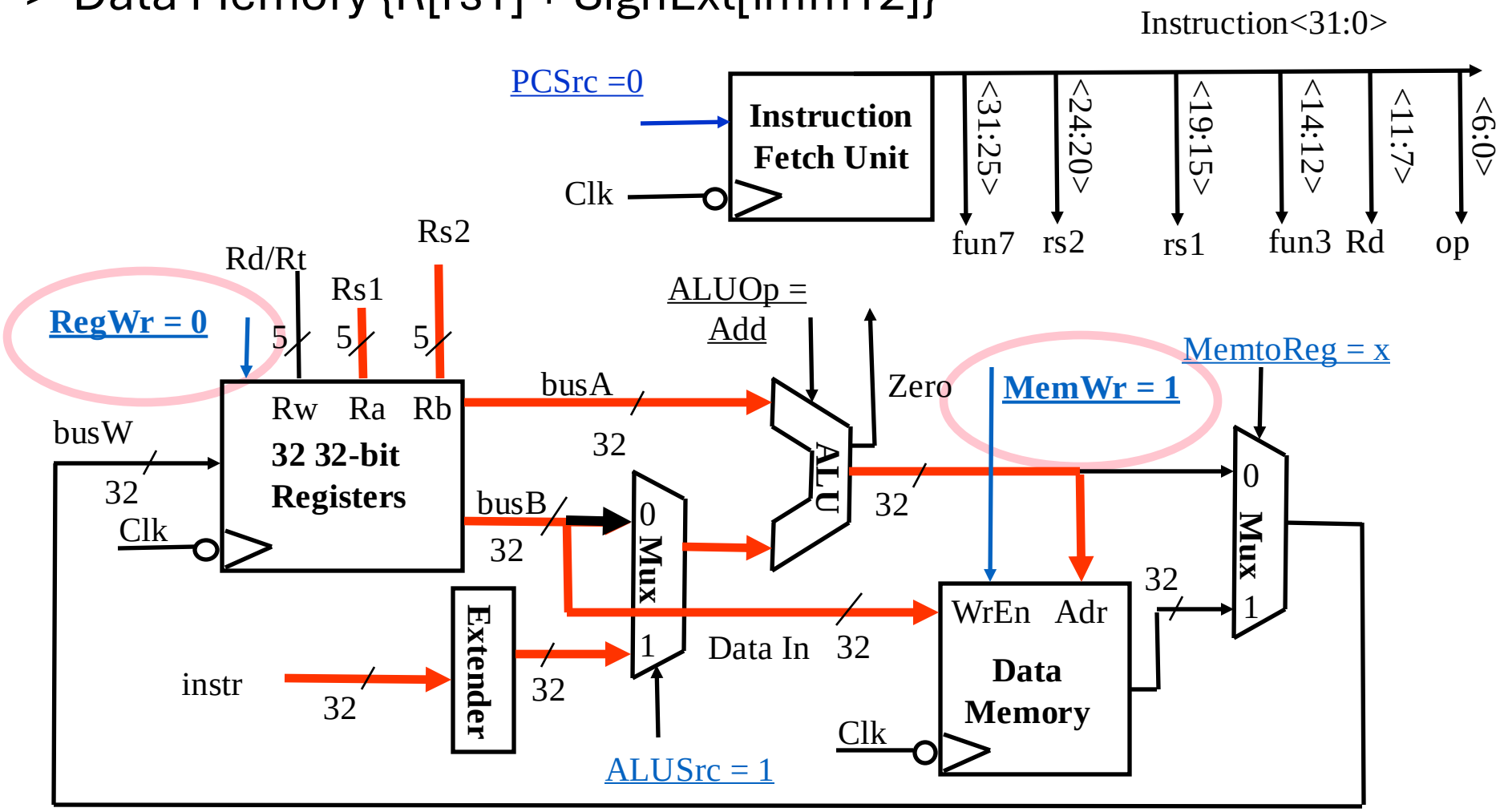
sw Controls: Worksheet

- R[rs2] -> Data Memory {R[rs1] + SignExt[imm12]}



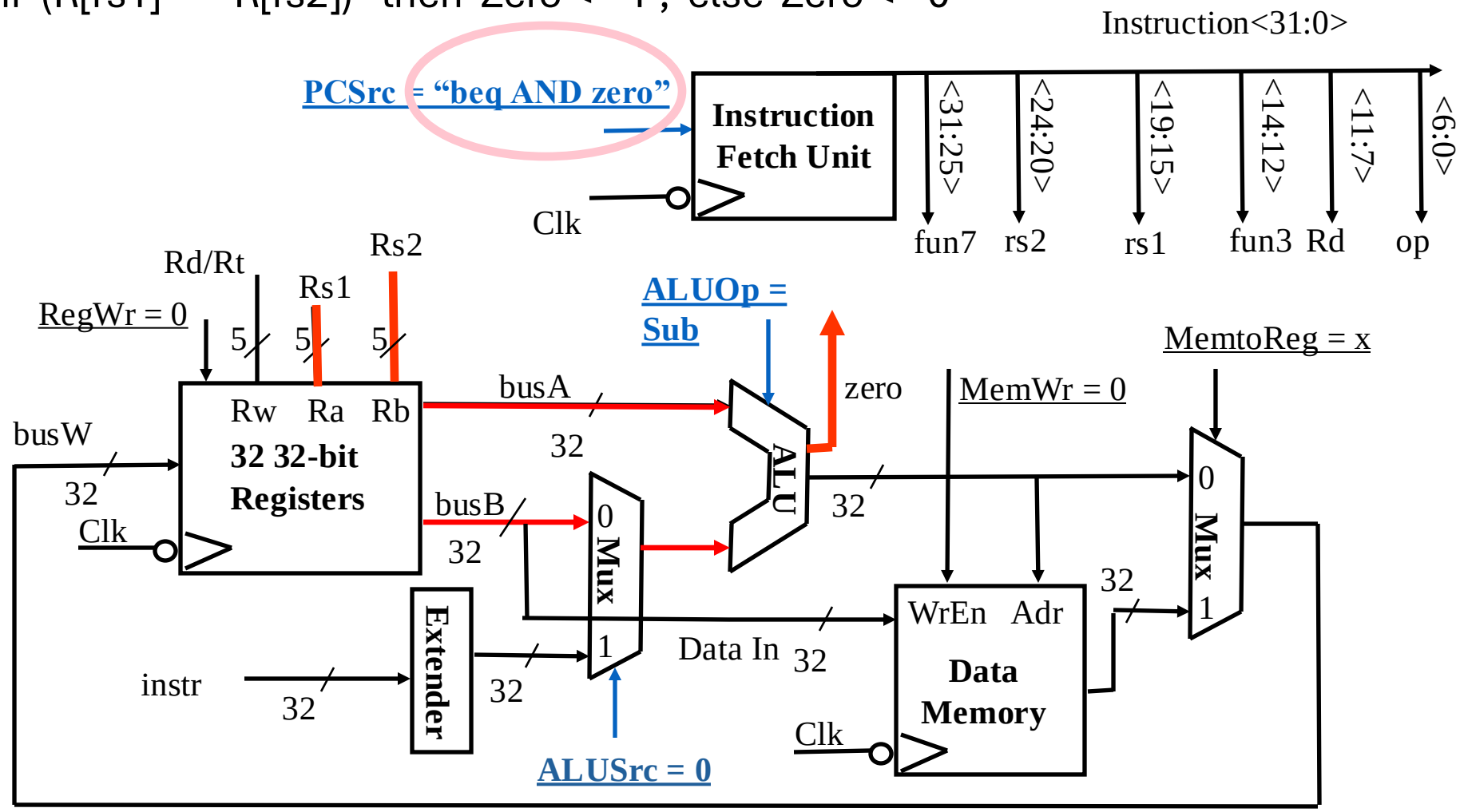
sw Controls: Solution

- R[rs2] -> Data Memory {R[rs1] + SignExt[imm12]}



beq controls

- if (R[rs1] == R[rs2]) then Zero <- 1 ; else Zero <- 0



Summary of Control Signals

inst	Register Transfer	
ADD	R[rd] <– R[rs1] + R[rs2];	PC <– PC + 4
	ALUsrc = RegB, ALUOp = “add”, RegWr, PCSrc = “+4”	
SUB	R[rd] <– R[rs1] – R[rs2];	PC <– PC + 4
	ALUsrc = RegB, ALUOp = “sub”, RegWr, PCSrc = “+4”	
LOAD	R[rd] <– MEM[R[rs1] + sign_ext(imm12)];	PC <– PC + 4
	ALUsrc = Im, ALUOp = “add”, MemtoReg, RegWr, PCSrc = “+4”	
STORE	MEM[R[rs1] + sign_ext(imm12)] <– R[rs2];	PC <– PC + 4
	ALUsrc = Im, ALUOp = “add”, MemWr, PCSrc = “+4”	
BEQ	if (R[r1] == R[r2]]) then PC <– PC + sign_ext(imm12)] || 0 else PC <– PC + 4;	
	PCSrc = “beq AND zero”, ALUOp = “sub”	

Control Logic

- Logic must generate appropriate signals for all instructions
- Summary slide (previous)
  - A way of representing the truth table
- Till now: Instr → signal, next: transpose
  - First: Equations in terms of opcodes
  - Next: Equations in terms of instruction bits

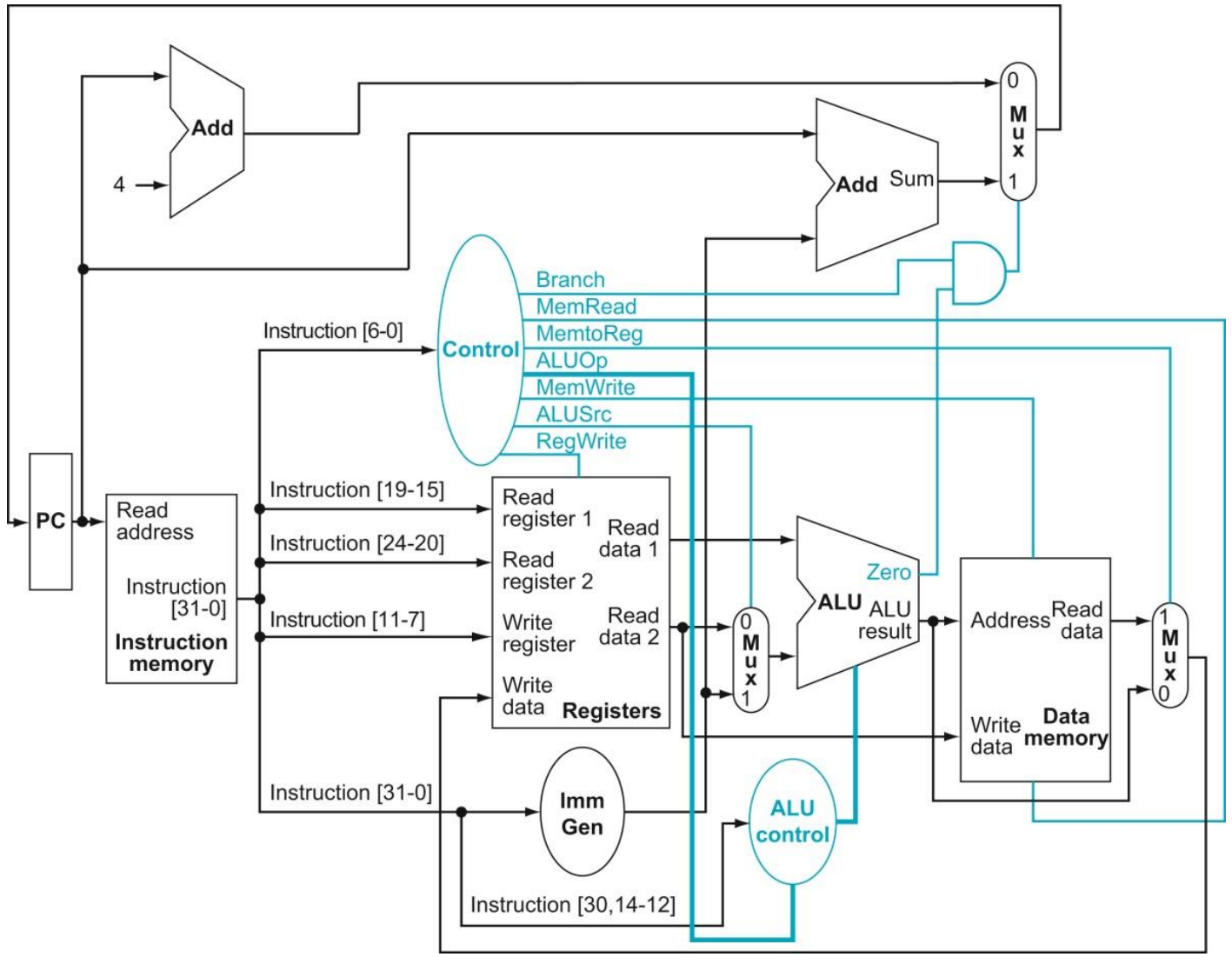
Controls: Logic equations

- PCSrc <= if (OP == BEQ) then "Zero" else 0
- ALUsrc <= if (OP == “R-type”) then “regB”
  elseif (OP == beq) then “regB”
  else “imm”
- ALUOp <= if (OP == “R-type”) then **funct**
  elseif (OP == beq) then “sub”
  else “add”
- MemWr <= _____
- MemtoReg <= _____
- RegWr: <= _____

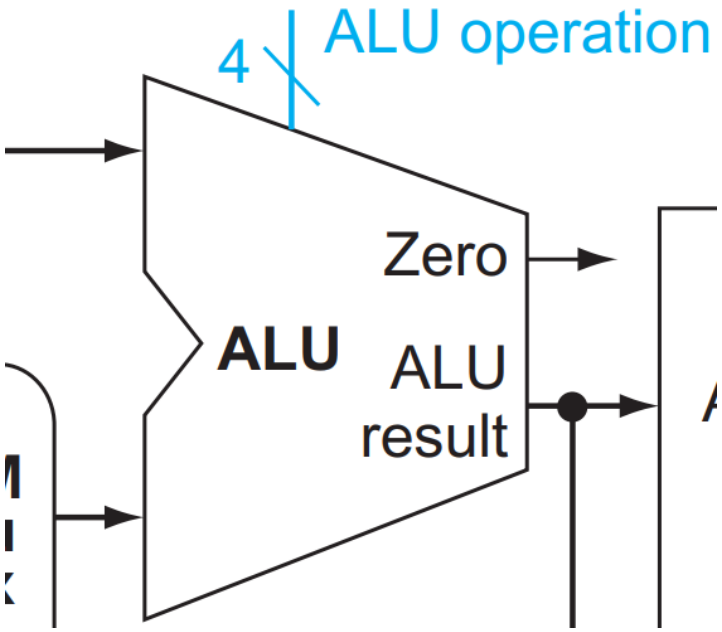
Controls: Logic equations

- PCSrc <= if (OP == BEQ) then Zero else 0
- ALUsrc <= if (OP == “R-type”) then “regB”
  elseif (OP == BEQ) then regB, else “imm”
- ALUOp <= if (OP == “R-type”) then **funct**
  elseif (OP == BEQ) then “passThrough”
  else “add”
- MemWr <= (OP == Store)
- MemtoReg <= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then 0 else 1

Full Data Path with control



ALU Operations



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Concrete signals for the ALU

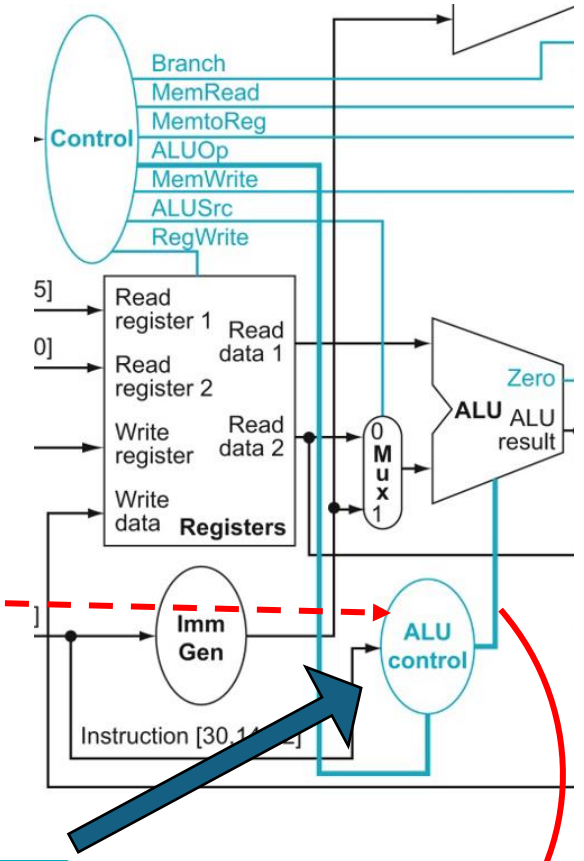
- We create a control signal called ALUOp
  - There are three different scenarios for ALU operation (lw/sw, beq, then R-type)
  - You only need to look at the func fields in R-Type

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Multi-level control

- Based on a combination of the ALUOp and the func fields: set the ALU Operation

Control contains combinational logic to set the signals



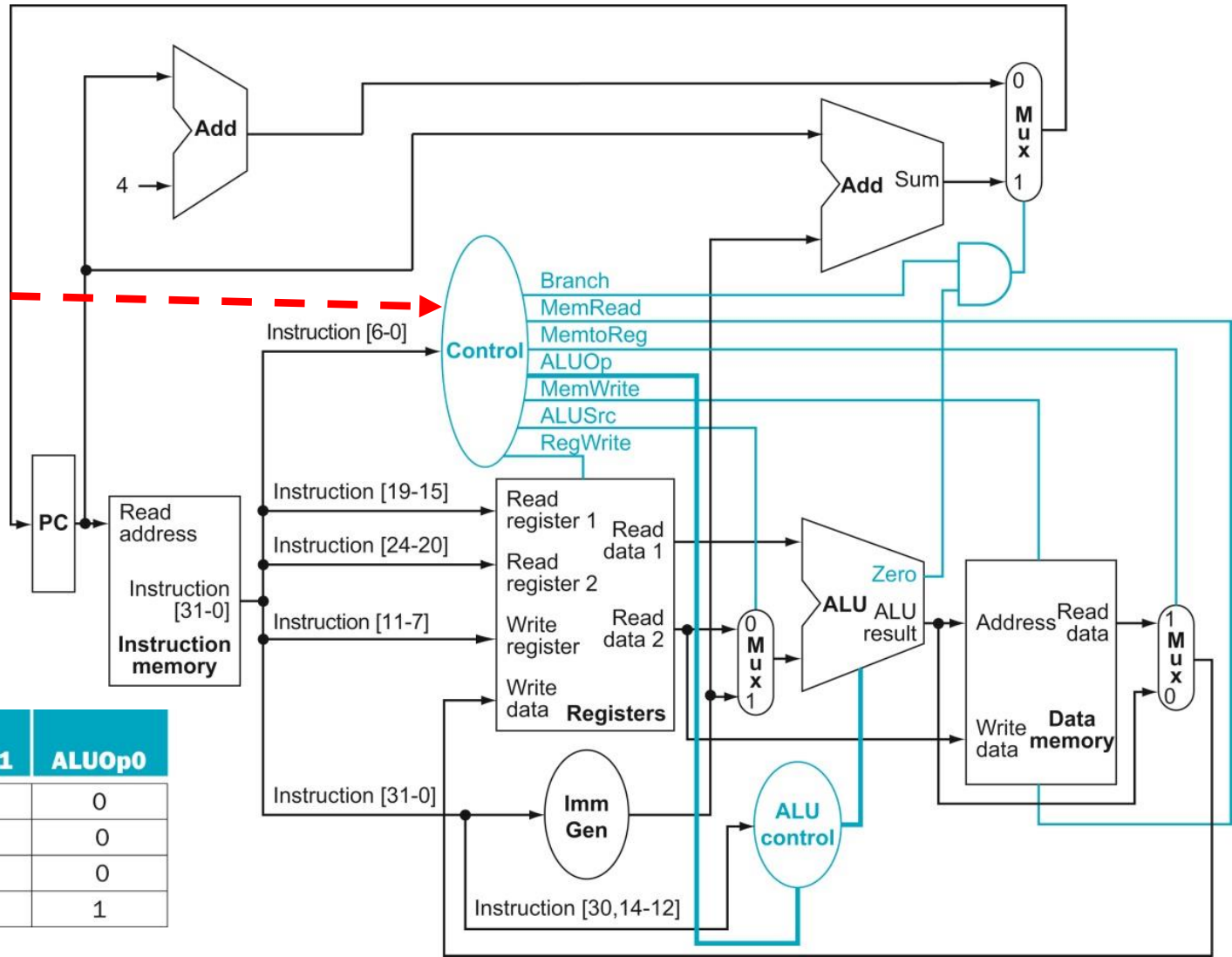
	ALUOp		Funct7 field							Funct3 field			Operation
	ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
lw/sw	0	0	X	X	X	X	X	X	X	X	X	X	0010
beq	X	1	X	X	X	X	X	X	X	X	X	X	0110
R-type	1	X	0	0	0	0	0	0	0	0	0	0	0010
	1	X	0	1	0	0	0	0	0	0	0	0	0110
	1	X	0	0	0	0	0	0	0	1	1	1	0000
	1	X	0	0	0	0	0	0	0	1	1	0	0001

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Main Control Unit

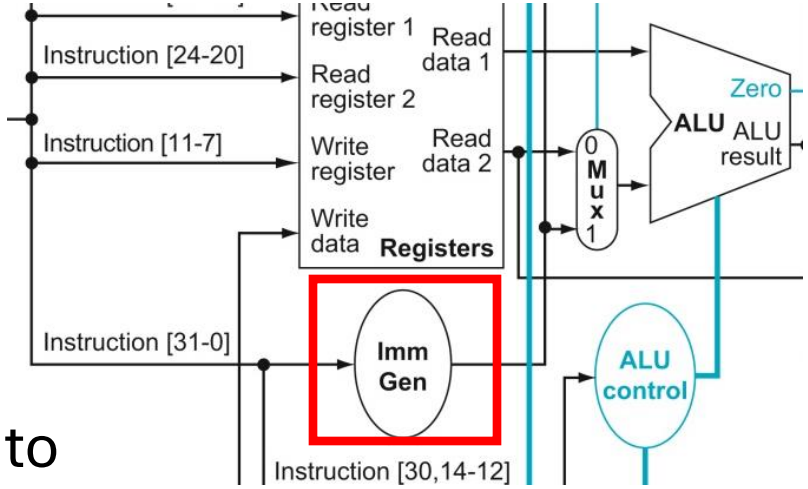
- Can determine these signals directly from the Op Code

Control contains combinational logic to set the signals



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

Immediate Generator



- Takes in the instruction, spits out a 32-bit immediate
- The table below maps the bits output by the unit to the instruction bits based on the instruction type
- **This table is without all the crazy bit-swizzling**

		Immediate Output Bit by Bit																													
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Format	Immediate Input Bit by Bit																															
Load, Arith. Imm.	I	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20
Store	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i11	i10	i9	i8	i7
Cond. Branch	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i30	i29	i28	i27	i26	i25	i24	"	"	"	"	0		
Uncond. Jump	U	"	"	"	"	"	"	"	"	"	"	"	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	i19	i18	i17	i16	i15	i14	i13	i12	"		
Load Upper Imm	U	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	i19	i18	i17	i16	i15	i14	i13	i12	0	0	0	0	0	0	0	0	0	0	0	"
Unique Inputs		1	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	4	3

The more inputs you have to select each bit, the wider the internal mux needed to implement the bit

Immediate Generator

Now, let's add the swizzling!

Name (Field size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

No Swizzle

		Immediate Output Bit by Bit																													
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Format	Immediate Input Bit by Bit																															
Load, Arith. Imm.	I	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20
Store	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i11	i10	i9	i8	i7
Cond. Branch	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i30	i29	i28	i27	i26	i25	i24	"	"	"	"	0
Uncond. Jump	U	"	"	"	"	"	"	"	"	"	"	"	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	i19	i18	i17	i16	i15	i14	i13	i12	"
Load Upper Imm.	U	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	i19	i18	i17	i16	i15	i14	i13	i12	0	0	0	0	0	0	0	0	0	0	0	"
Unique Inputs		1	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	4	3

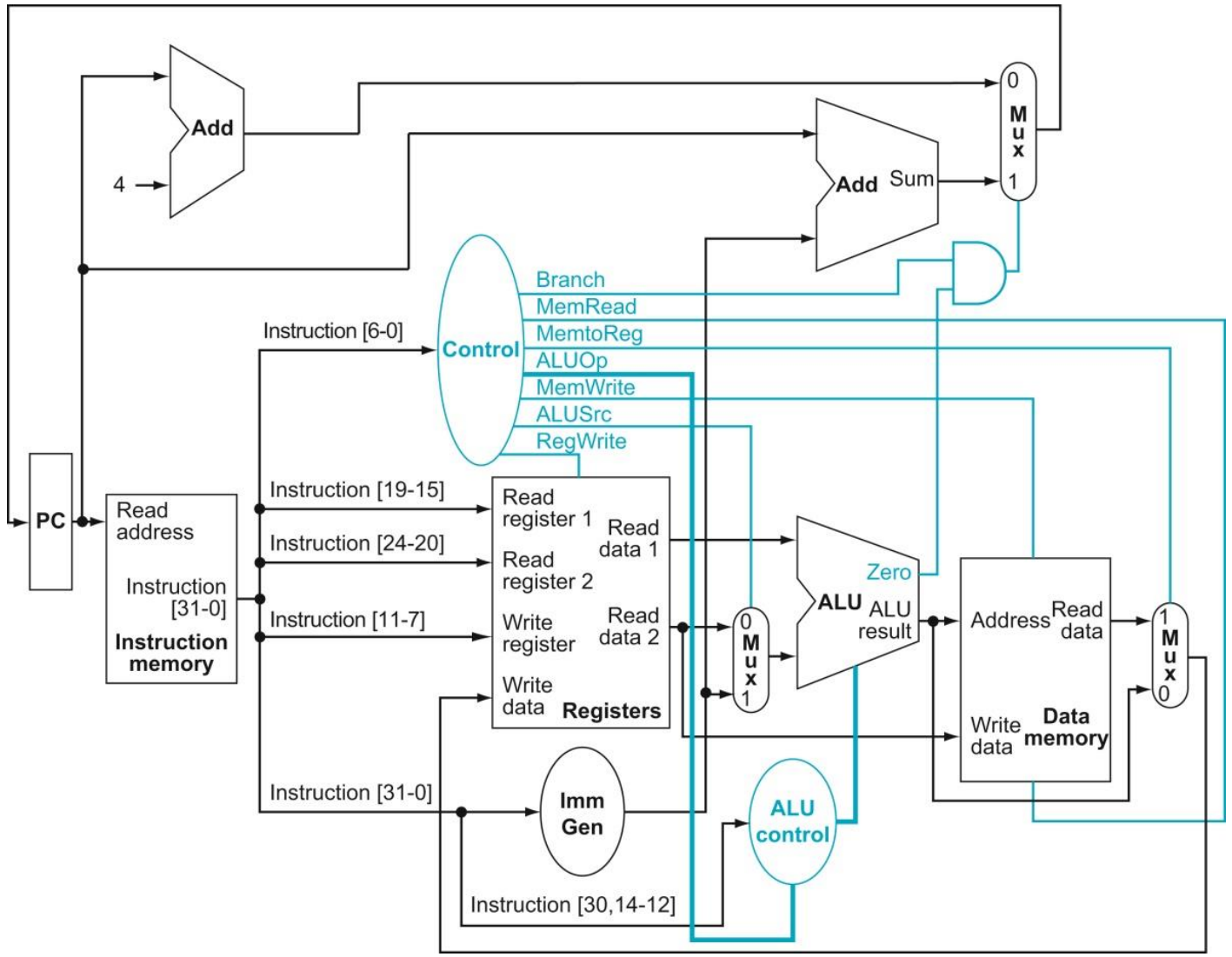
With Swizzle

		Immediate Output Bit by Bit																															
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Format	Immediate Input Bit by Bit																															
Load, Arith. Imm.	I	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20
Store	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i11	i10	i9	i8	i7
Cond. Branch	SB	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i7	"	"	"	"	"	"	"	"	"	"	"	"	0
Uncond. Jump	UJ	"	"	"	"	"	"	"	"	"	"	"	"	i19	i18	i17	i16	i15	i14	i13	i12	i20	"	"	"	"	"	"	"	"	"	"	"	"	"
Load Upper Imm.	U	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	"	"	"	"	"	"	"	"	0	0	0	0	0	0	0	0	0	0	0	"
Unique Inputs		1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	4	2	2	2	2	2	2	3	3	3	3	3

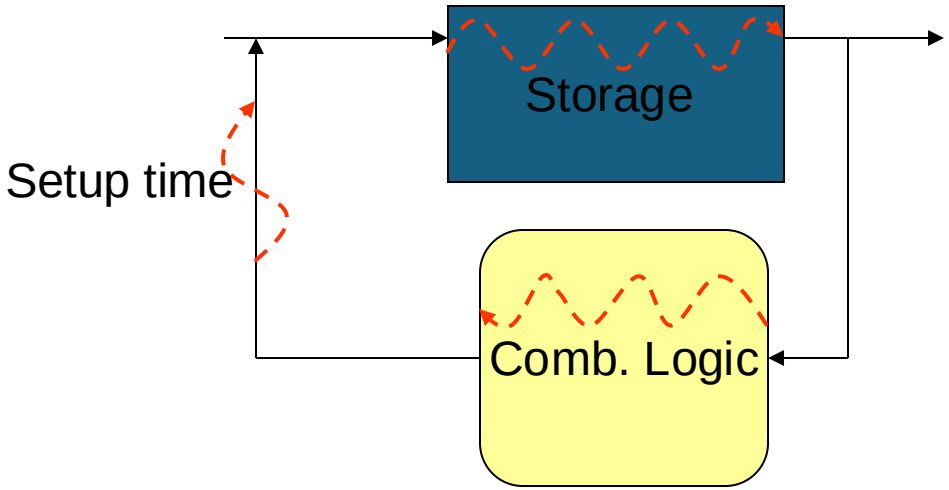
Why Swizzling

- Allows for simpler hardware
  - This small optimization doesn't matter in a big, complex implementation of RISC-V.
  - Saves area, energy, and latency in a simple implementation where every gate matters.
- It makes the assembler's life a little more complicated
  - But it is not too bad – program knows the pattern, repeats
  - **Does, however, make the binary encoding harder to read**
    - **And makes it harder to encode/decode by hand :)**

Now you've seen all of it!



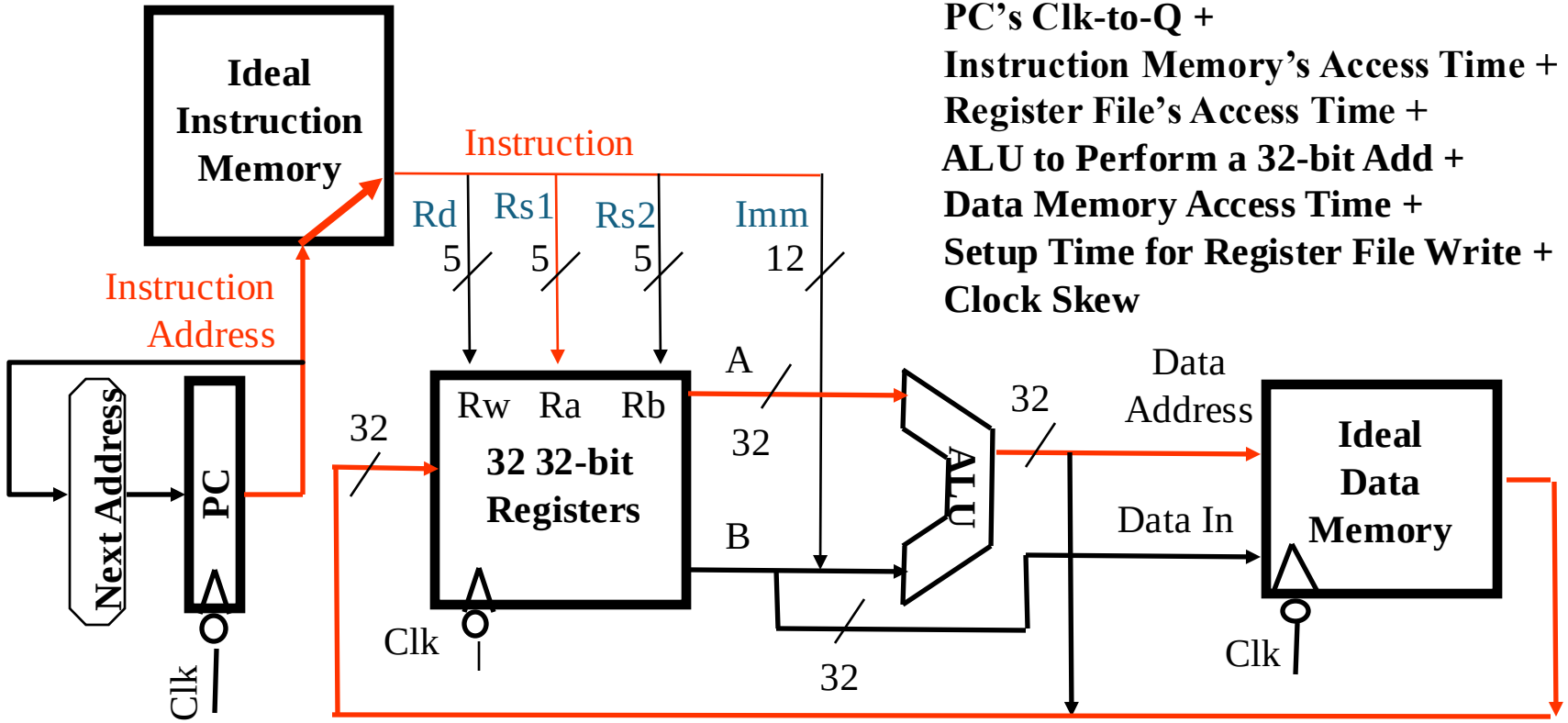
Cycletime



- What should the clock period be?
  - Enough to compute the next state values
    - Propagation clk-to-Q (new state)
    - Comb. Logic delay
    - Setup requirements

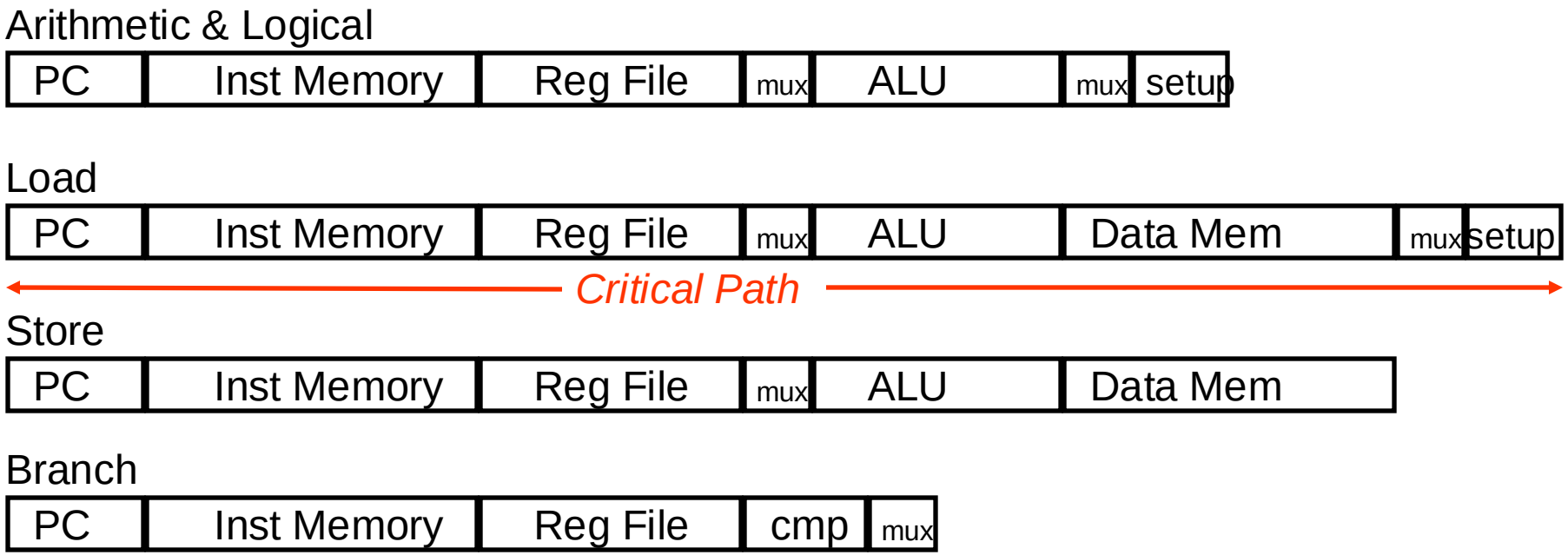
“lw” Instruction

- Longer critical path
  - lower bound on cycletime



Critical Path (Load Operation) =
PC’s Clk-to-Q +
Instruction Memory’s Access Time +
Register File’s Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

What's wrong with our processor?



- LOOOOONG Cycle Time
- ALL instructions take as much time as the slowest
- Real memory MUCH slower than our idealized memory
  - Today some 100ns (memory+bus+control) vs. 0.25ns CPU clock
  - cannot finish in one (short) cycle

Again, take 437!