Roadmap

- 1. Analyze instruction set ⇒ datapath requirements √
  - MIPS Lite: arithmetic/logical, data moving, branch
- 2. Select set of datapath components and establish clocking methodology √
  - Combinational and sequential elements
- 3. Assemble datapath meeting the requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer
- 5. Assemble the control logic

Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Fetch instructions
  - Read operands and execute instructions

Register Transfer Language

- RTL gives the meaning of the instructions
- First step is to fetch/decode instruction from memory

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

← Instruction Fetch

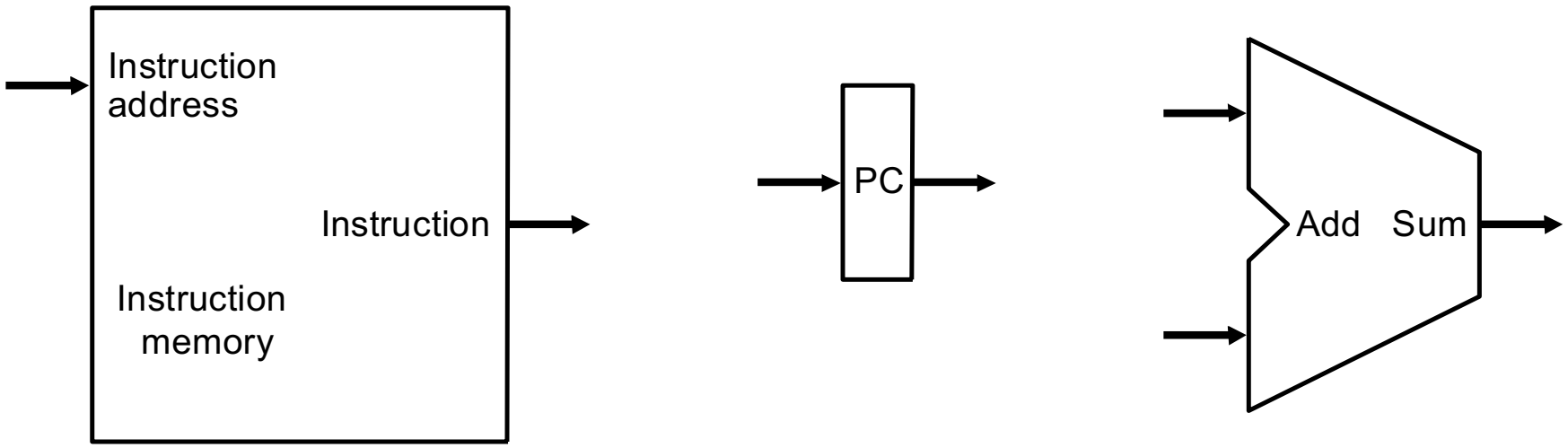
PC updating

inst	Register Transfers	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
OR	R[rd] <– R[rs] | R[rt];	PC <– PC + 4
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rt];	PC <– PC + 4
BEQ	if (R[rs] == R[rt]) then PC <– PC +4 + (sign_ext(Imm16) <<2) else PC <– PC + 4	

3a: Instruction Fetch Unit

- The common RTL operations
  - Fetch the instruction: mem[PC]
  - Update the program counter:
    - Sequential code: PC <- PC + 4
    - Branch and jump: PC <- "something else"
    - We don't know if instruction is a branch/jump or one of the other instructions until we have fetched and interpreted the instruction from memory
    - So all instructions initially increment the PC by 4

Components to Assemble

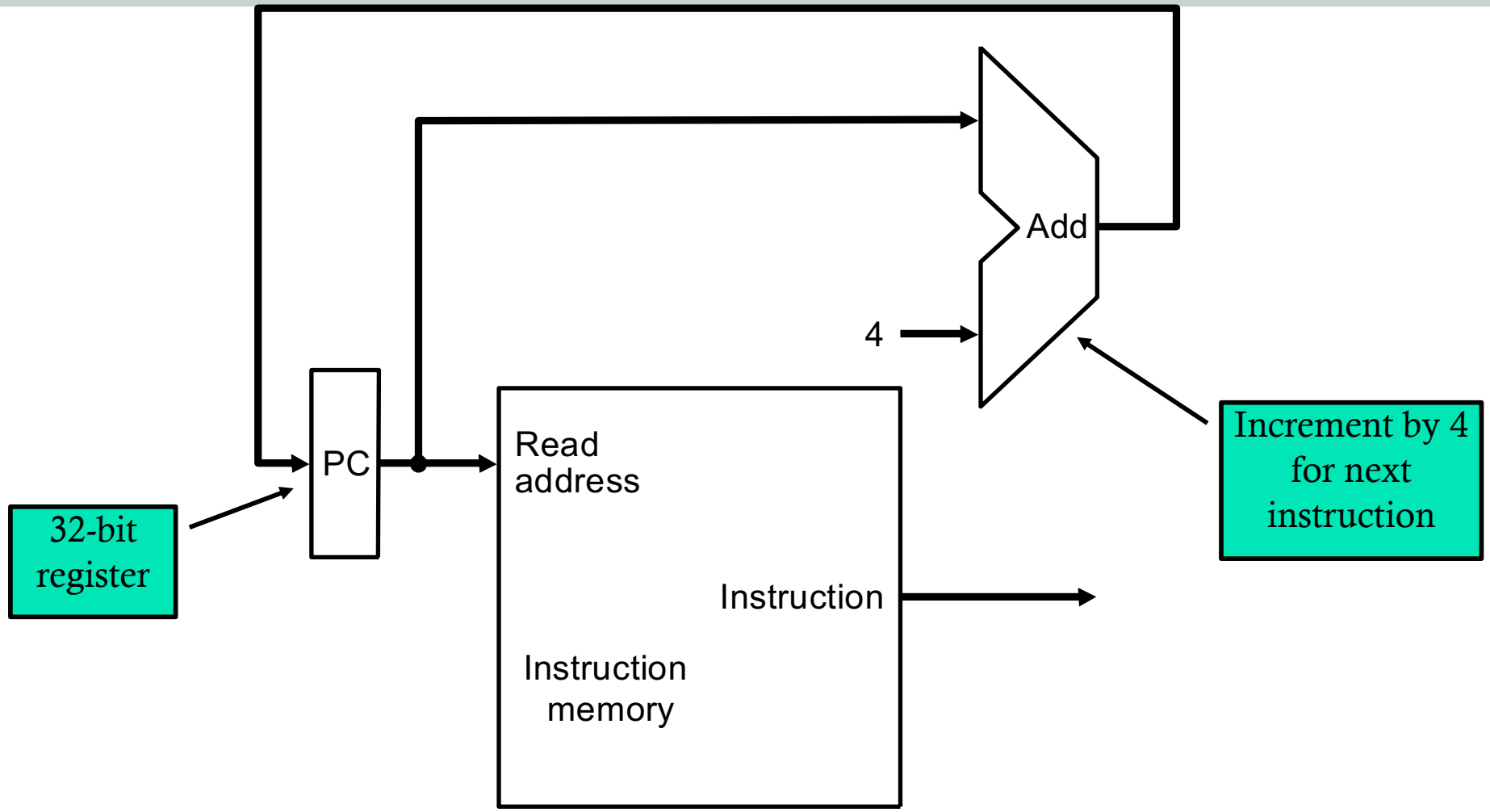


a. Instruction memory

b. Program counter

c. Adder

Datapath for Instruction Fetch

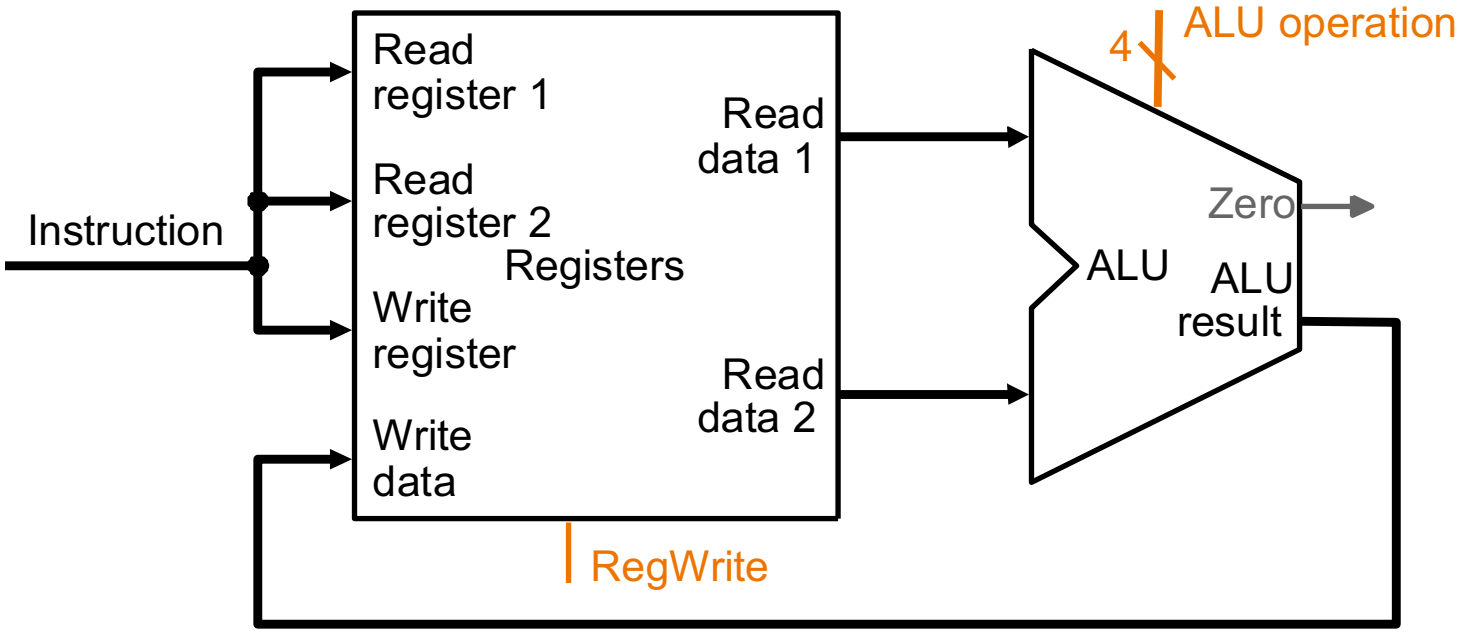


Step 3b: R-format Instructions

- Example instructions: add, sub, and, or
- R[rd] <- R[rs] op R[rt]
  - Read two registers
  - Perform arithmetic/logical operation
  - Write register result

Datapath for R-format Instructions

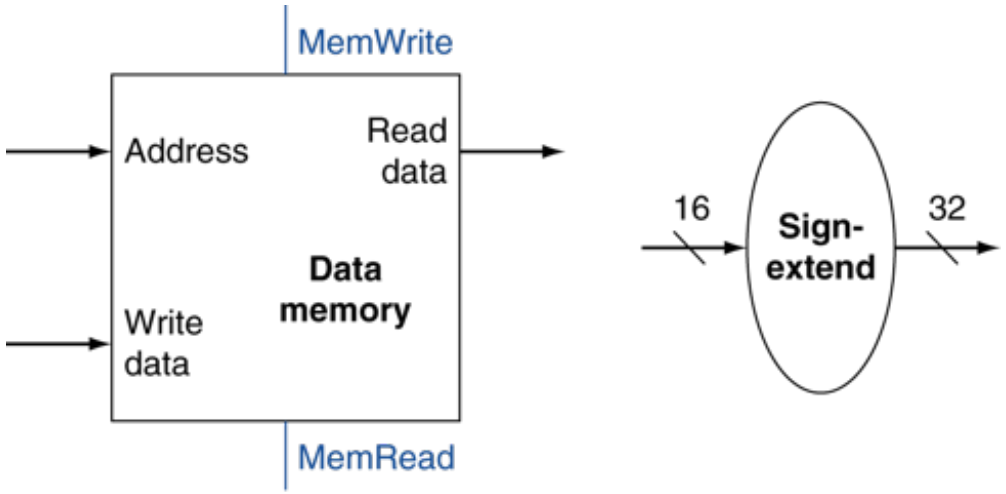
- Read register 1, Read register 2, and Write register come from instruction's rs, rt, and rd fields
- ALU control and RegWrite: control logic after decoding the instruction



Load/Store Instructions

- R[rt] <- Mem[R[rs] + SignExt[imm16]] # lw rt, rs, imm16
- Mem[R[rs] + SignExt[imm16]]<- R[rt] # sw rt, rs, imm16

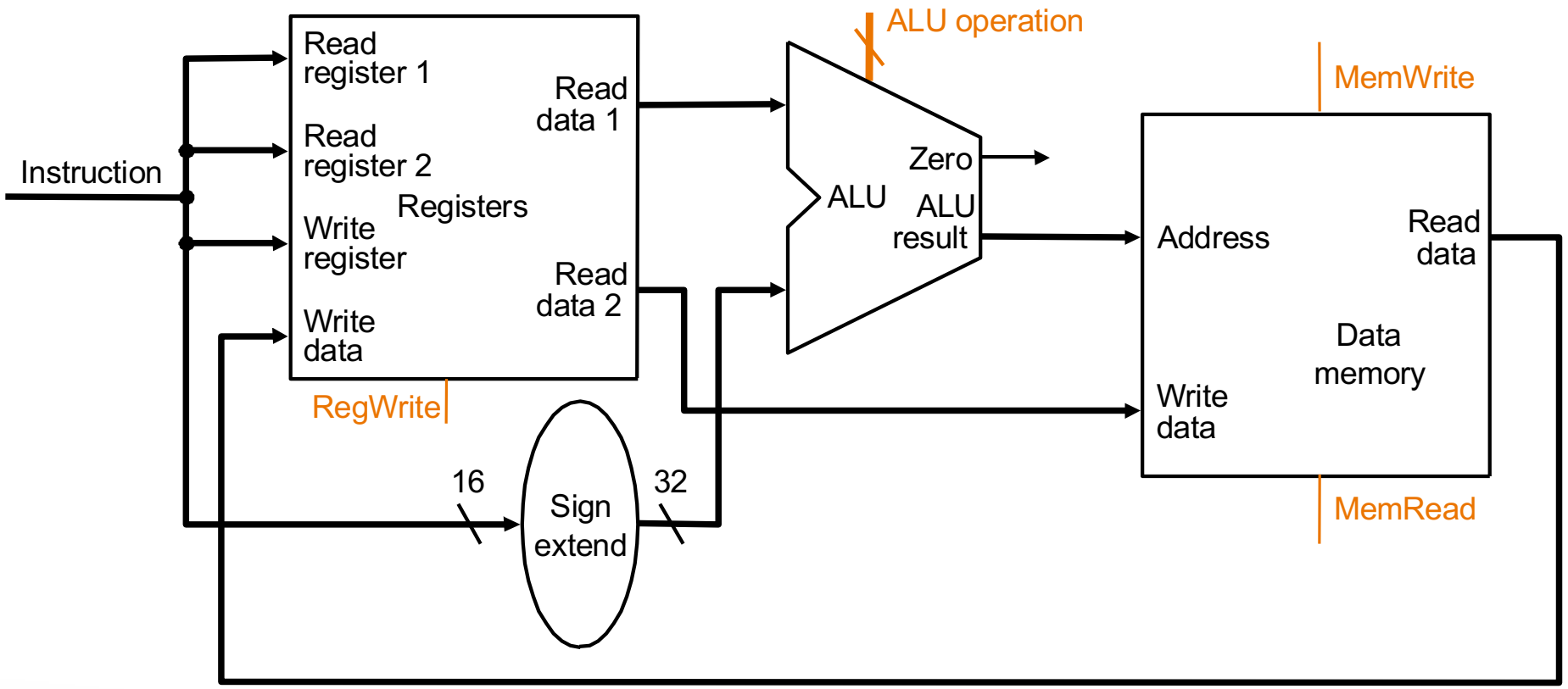
- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset



a. Data memory unit b. Sign extension unit

Datapath for LW & SW

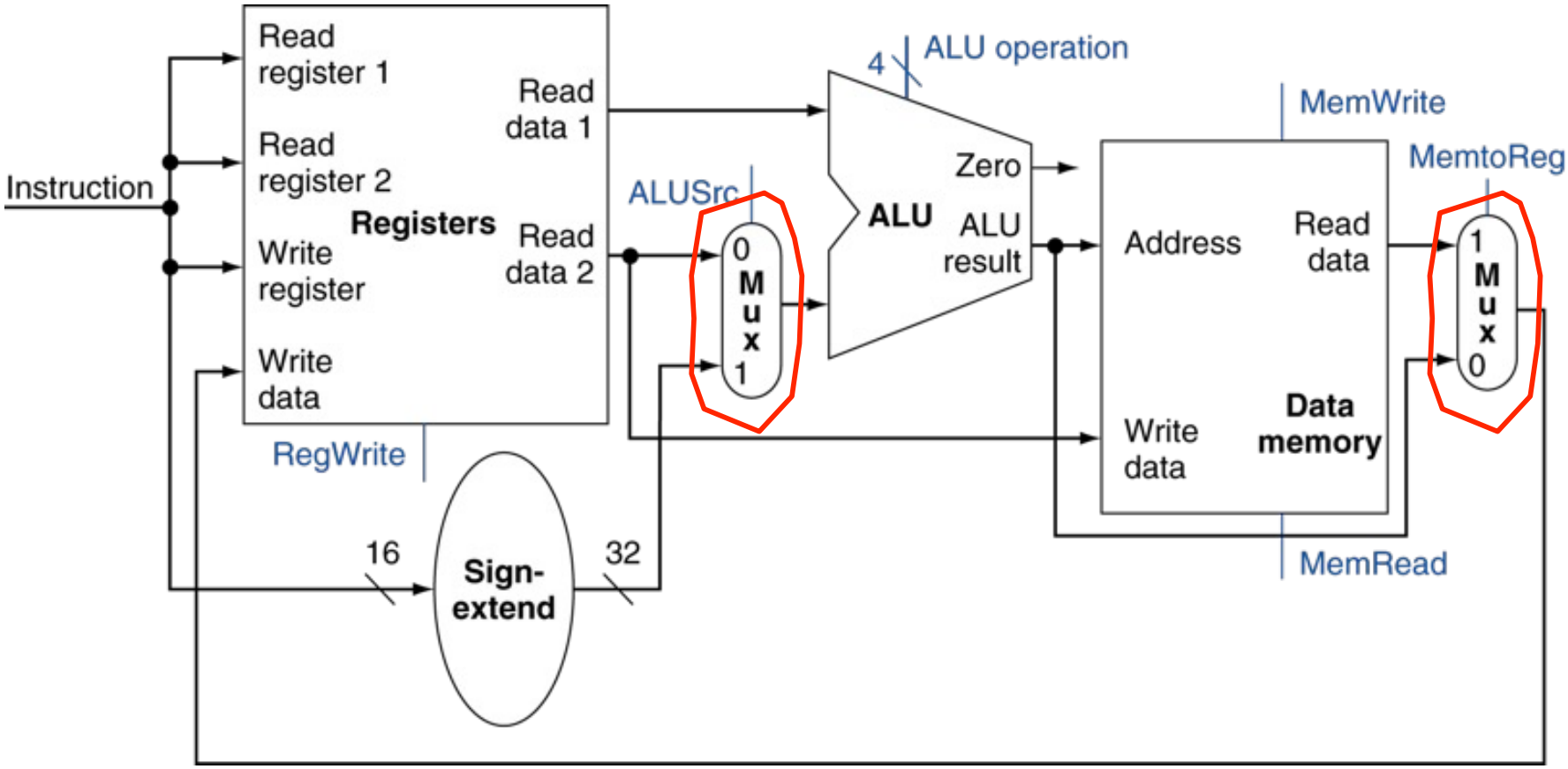
- R[rt] <- Mem[R[rs] + SignExt[imm16]] # lw rt, rs, imm16
- Mem[R[rs] + SignExt[imm16]]<- R[rt] # sw rt, rs, imm16



Composing the Elements

- First-cut data path completes one instruction in each clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Combine datapaths for R-type and data moving instructions
  - Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



ALU operand? How to update register? Which register to update?

Register Transfer Language

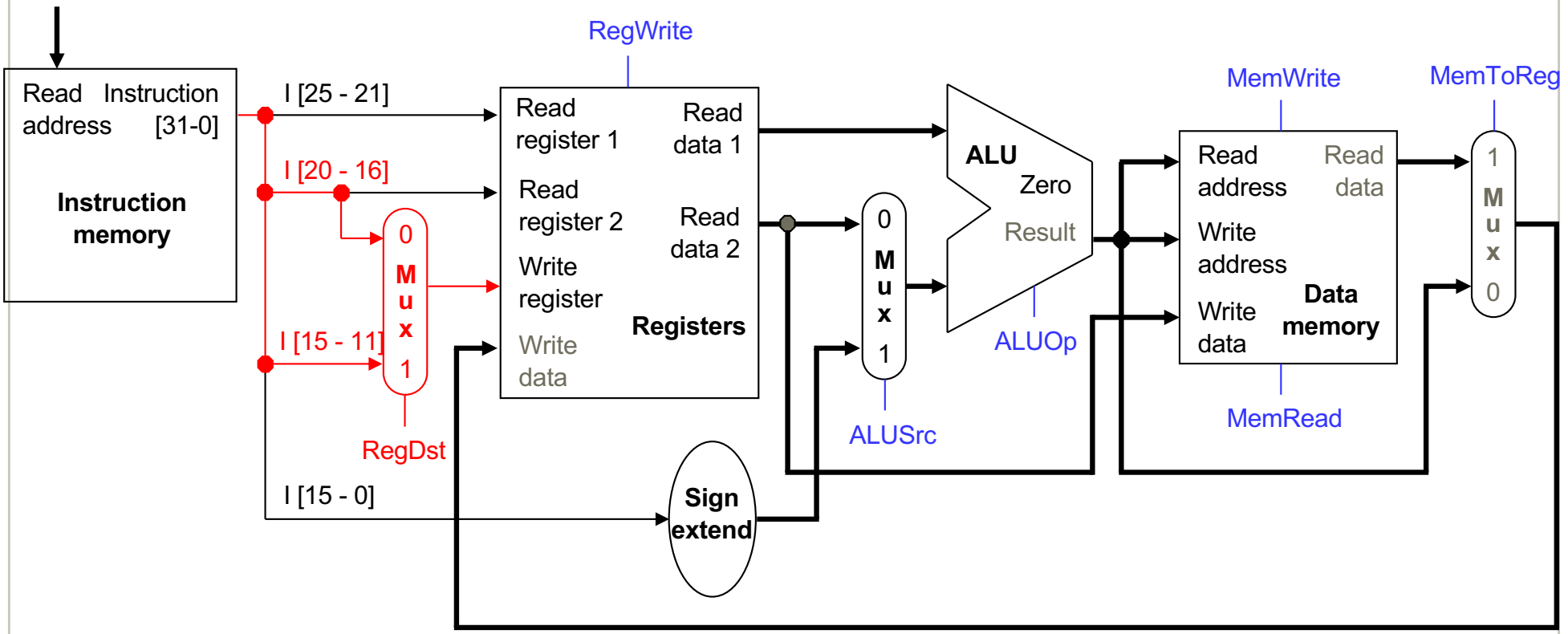
- RTL gives the meaning of the instructions
- First step is to fetch the instruction from memory

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

inst	Register Transfers	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
OR	R[rd] <– R[rs] | R[rt];	PC <– PC + 4
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rt];	PC <– PC + 4
BEQ	if (R[rs] == R[rt]) then PC <– PC +4 + (sign_ext(Imm16) || 00)	else PC <– PC + 4

R-Type/Load/Store Datapath



Branch Instructions

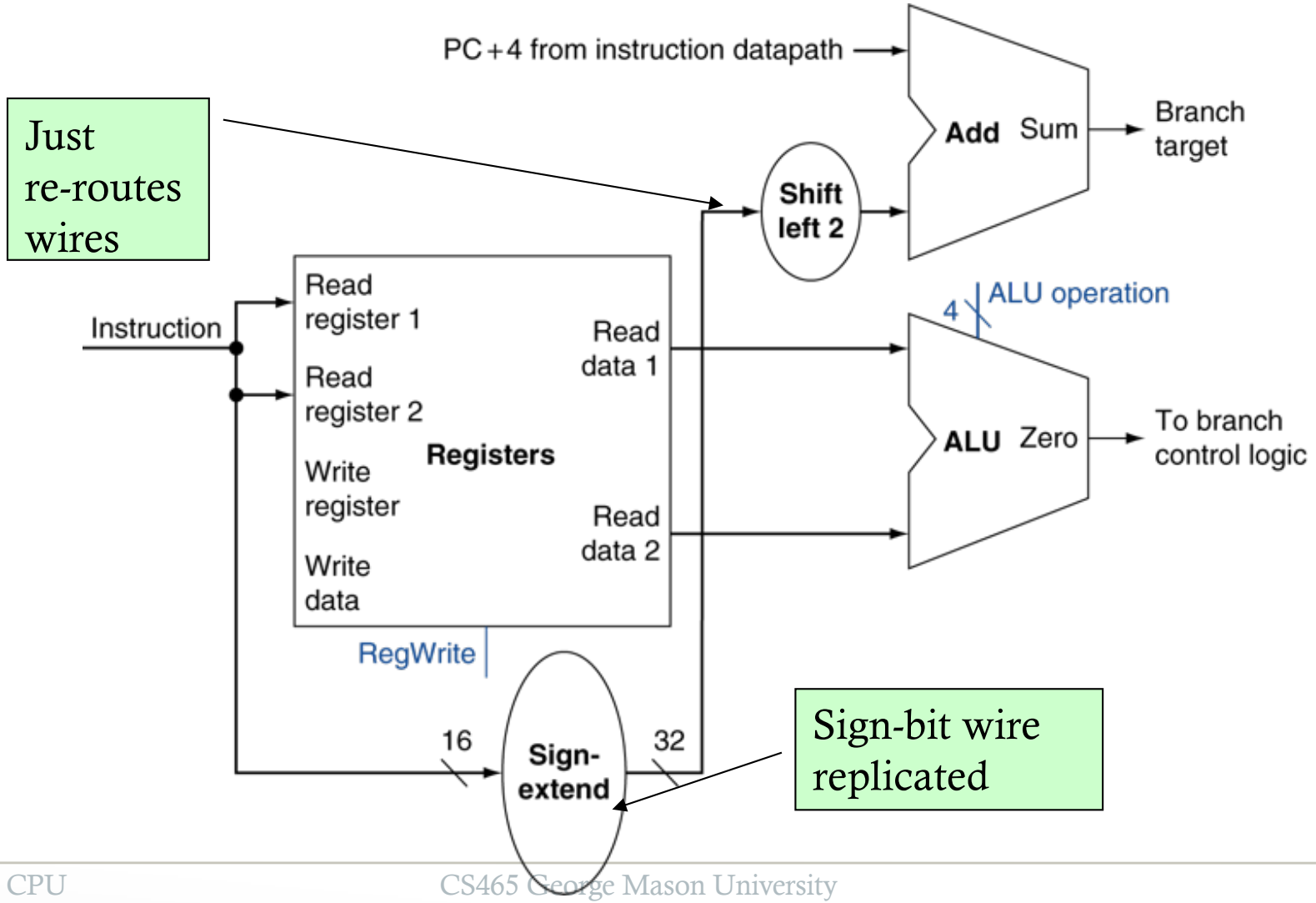
- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

beq rs, rt, L

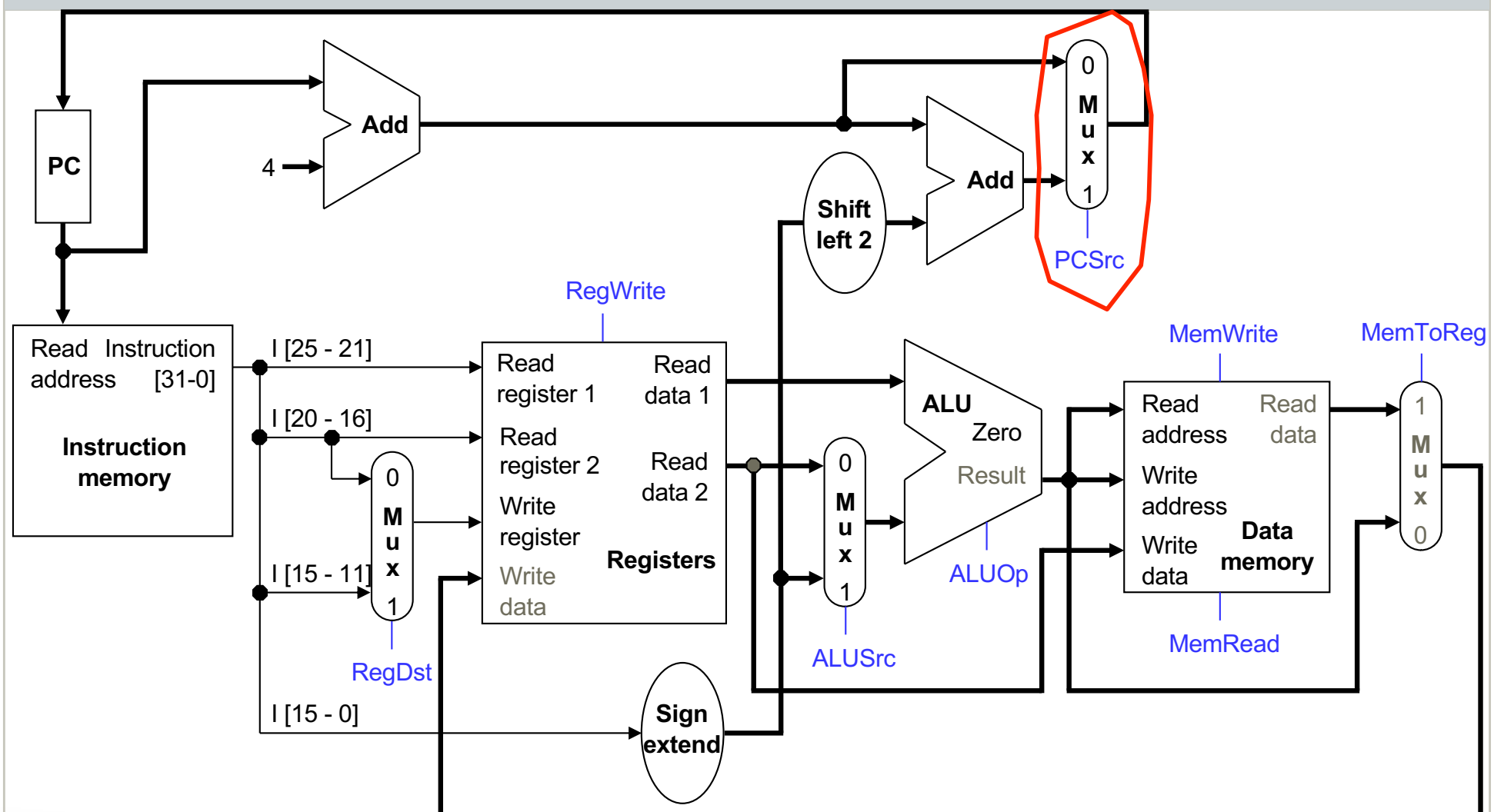
if (R[rs] == R[rt]) then PC <– PC +4 + (sign_ext(Imm16) <<2)

else PC <– PC + 4

Branch Instructions



Full Datapath



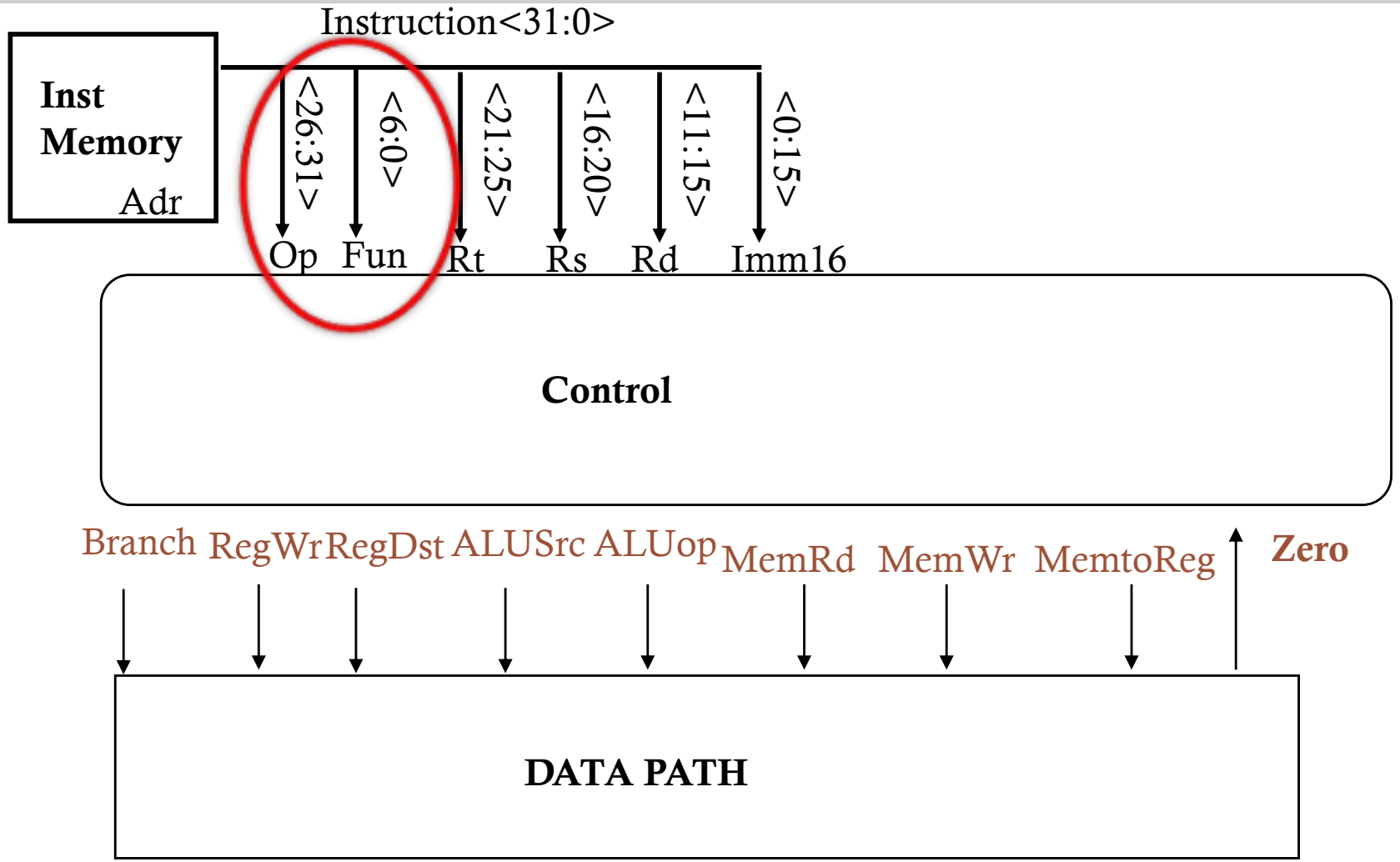
How to Design a Processor

- 1. Analyze instruction set ⇒ datapath requirements √
- 2. Select set of datapath components and establish clocking methodology √
- 3. Assemble datapath meeting the requirements √
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer
- 5. Assemble the control logic

Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information mostly comes from the 32 bits of the instruction
  - PCSrc (beq) as an exception

Step 4: Datapath: RTL -> Control



ALU Control

- ALU discussed in out last chapter comes with a 4-bit control signal
  - We need to generate the correct signals that match the required functionality

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

Register Transfer Language

- RTL gives the meaning of the instructions
- First step is to fetch the instruction from memory

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

inst	Register Transfers	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
OR	R[rd] <– R[rs] | R[rt];	PC <– PC + 4
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rt];	PC <– PC + 4
BEQ	if (R[rs] == R[rt]) then PC <– PC +4 + sign_ext(Imm16) <<2	
	else PC <– PC + 4	

ALU Control

- ALU used for
  - Load/Store: F = add (0010)
  - Branch: F = subtract (0110)
  - R-type: F depends on funct field (varied)
- Possible approach:

  - Concern: much more complicated than other control signals
  - Alternative approach: 2-step decoding

ALU Control

- ALU used for
  - Load/Store: F = add (0010)
  - Branch: F = subtract (0110)
  - R-type: F depends on funct field (varied)
- Two-level decoding/control
  - Level 1: check opcode only (based on instruction class)
    - 2-bit ALUOp generated
    - Output: Load/store: 00; branch: 01; R-type: 10
    - This is part of the main control unit
  - Level 2: further checking of funct if needed
    - Output: 4 bit ALU control

ALU Control

- Construct an additional control unit
  - Input1: 2-bit ALUOp derived from opcode
  - Input2: 6-bit funct field from instruction
  - Output: 4-bit ALU control signal

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111