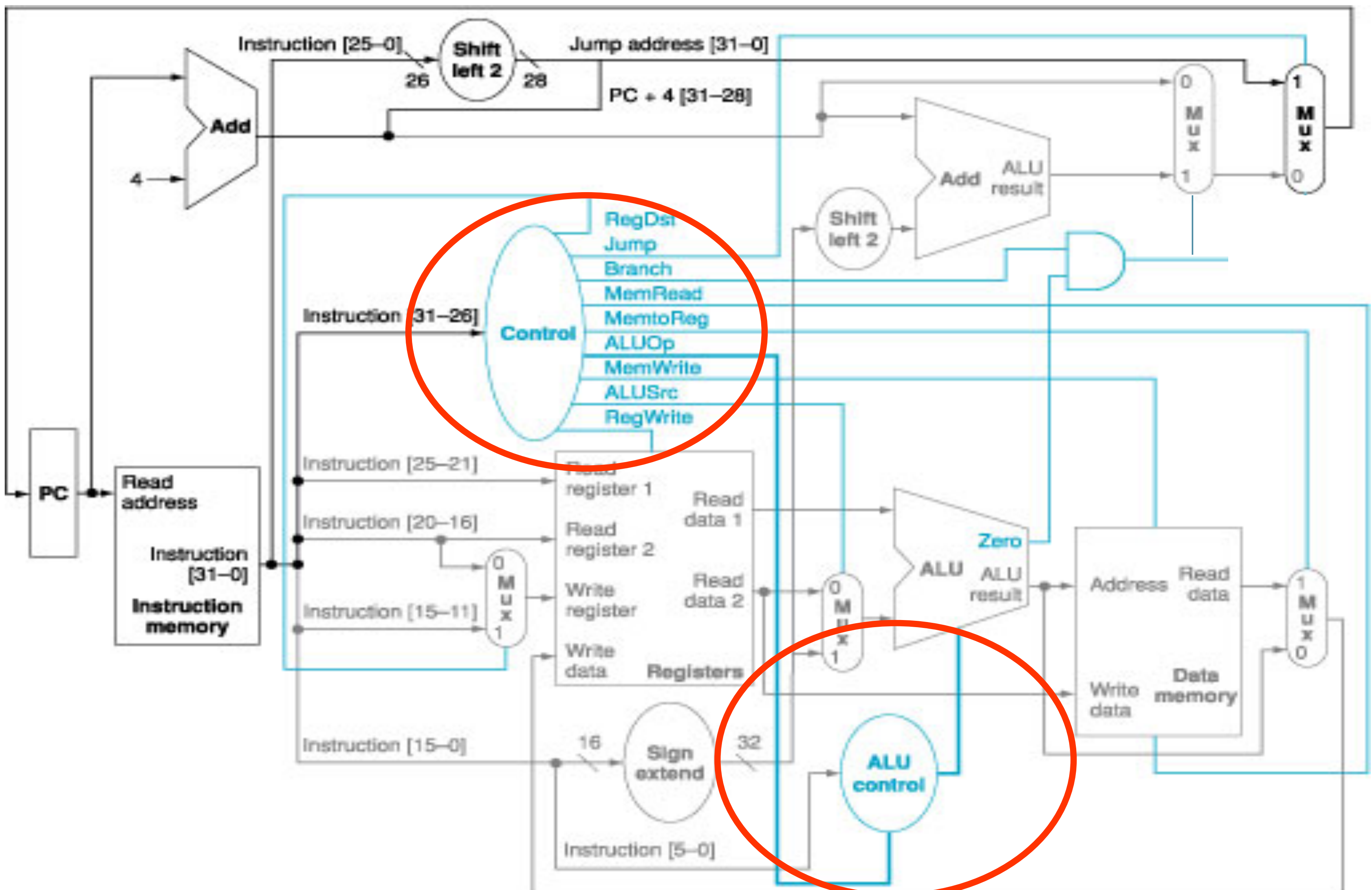
Lecture 4:
Building Single-Cycle Datapath and Control Unit



Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC + 4 [31–28]
Add
4
Add
ALU result
M u x
0
1
M u x
1
0
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
16
Sign extend
32
ALU control
Instruction [5–0]

How to Design a Processor: step-by-step

- 1. Analyze instruction set => datapath requirements
  - the meaning of each instruction is given by the *register transfers*
  - datapath must include storage element for ISA registers
  - datapath must support each register transfer
- 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 effect the register transfer.
- 5. Assemble the control logic

Step 1: The MIPS-lite Subset for today

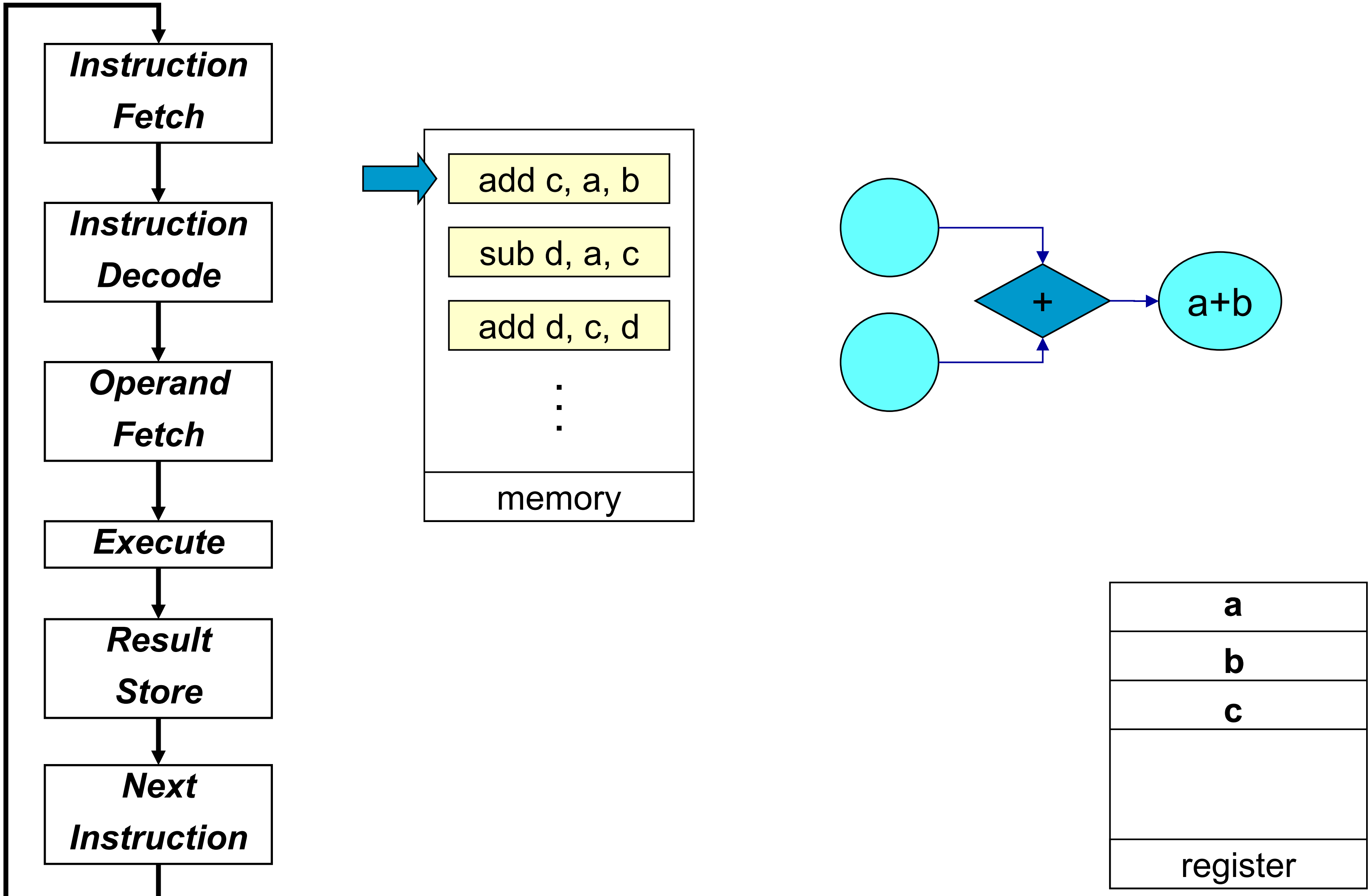
- ADD and SUB
  - addU rd, rs, rt
  - subU rd, rs, rt
- OR Immediate:
  - ori rt, rs, imm16
- LOAD and STORE Word
  - lw rt, rs, imm16
  - sw rt, rs, imm16
- BRANCH:
  - beq rs, rt, imm16
  - jump

31 – 26	21	16	11	6	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

31 – 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

31 – 26	0
op	target address
6 bits	26 bits

Execution Flow



Instruction <=> Register Transfers

- RTL (Register Transfer Languages) gives the meaning of the instructions
- All start by fetching the instruction

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

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

inst	Register Transfers	
ADDU	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
SUBU	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	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)] else PC <– PC + 4	

Step 1: Requirements of the Instruction Set

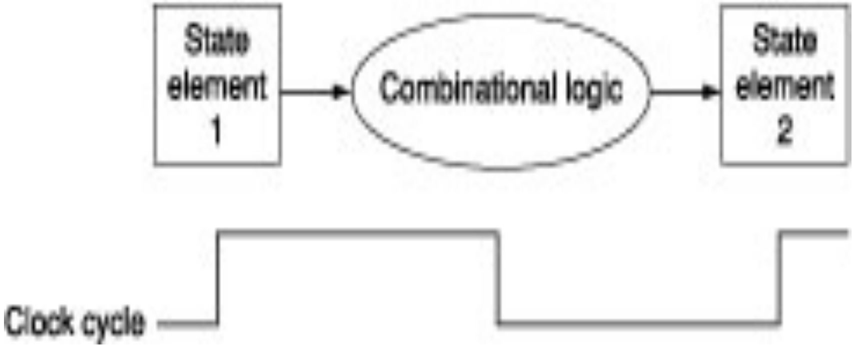
- Memory
  - instruction & data
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- PC
- Extender
- Add and Sub register or extended immediate
- Add 4 or extended immediate to PC

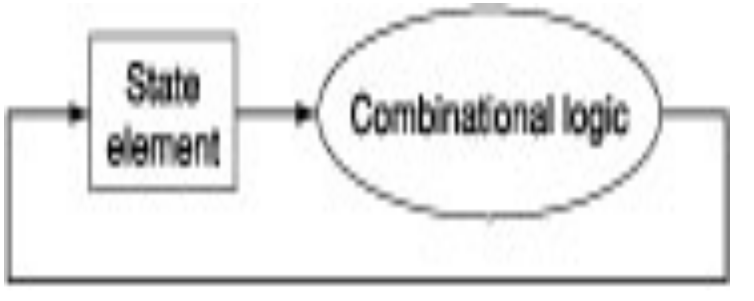
Step 2: Components of the Datapath

- **Combinational Elements**
  - The outputs only depend on the current inputs.
  - Example: ALU
- **Storage Elements** (state element)
  - The outputs depend on both their inputs and the contents of the internal state.
  - At least two inputs and one output
    - Inputs – input data and clock (clocking methodology)
    - Output – the value stored in a state element
  - Example: D Flip-Flop, register and memory

Clocking Methodology

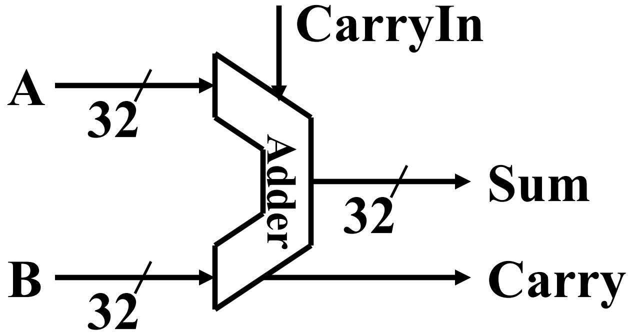
- Define when signals can be read and when they can be written
- Edge-triggered clocking
  - All state changes occur on a clock edge.
  - Active edge
    - Rising edge or Falling edge
  - No feedback in the same clock cycle
    - A state element could be read/written in the same clock cycle



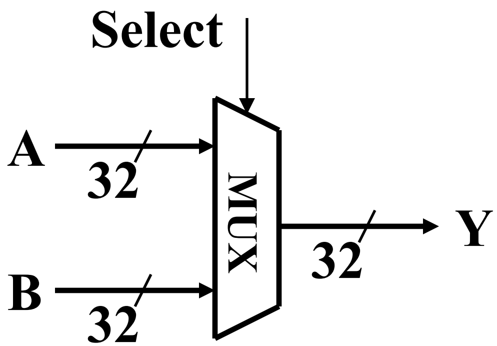


Combinational Logic Elements (Basic Building Blocks)

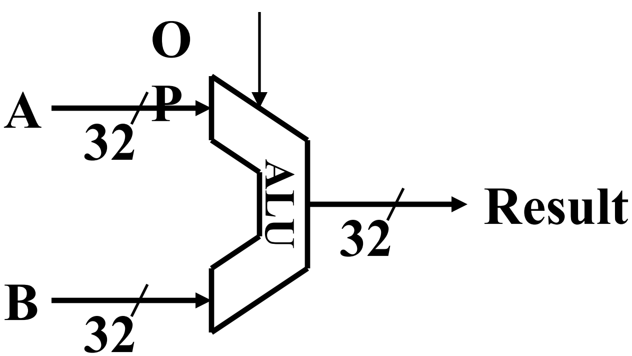
- Adder



- MUX

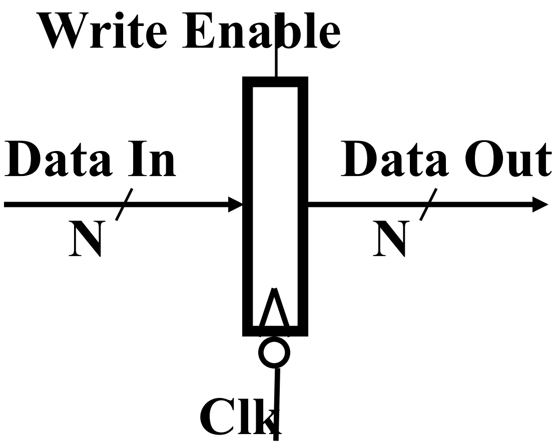


- ALU

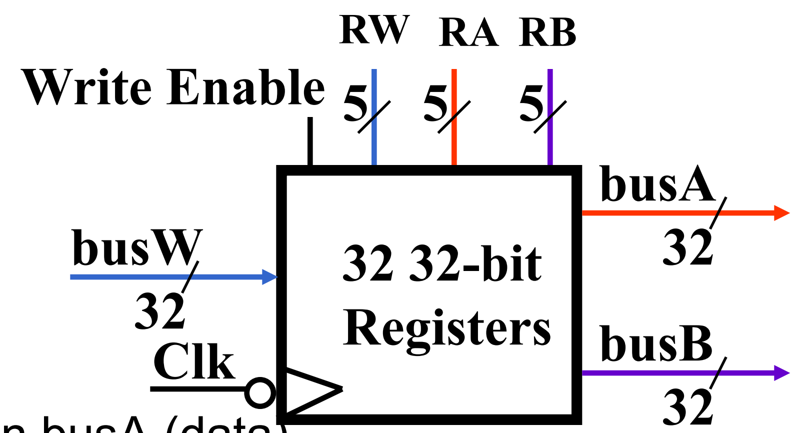


Storage Element: Register (Basic Building Block)

- Register
  - –Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
  - –Write Enable:
    - Asserted -> update the register contents



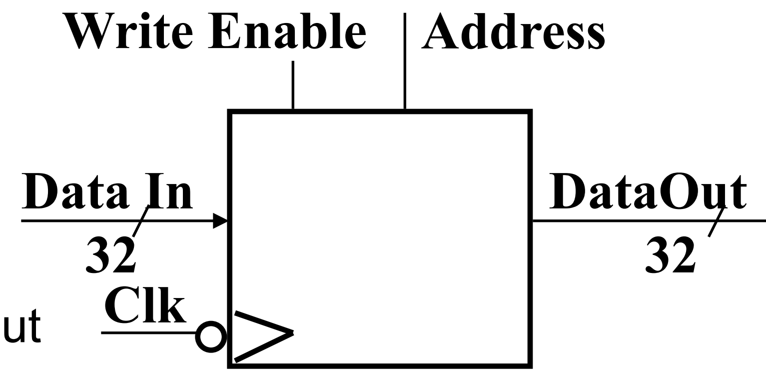
Storage Element: Register File



- Register File consists of 32 registers:
  – Two 32-bit output busses:
    busA and busB
  – One 32-bit input bus: busW
- Register is selected by:
  – RA (number) selects the register to put on busA (data)
  – RB (number) selects the register to put on busB (data)
  – RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (CLK)
  – The CLK input is a factor ONLY during write operation
  – During read operation, behaves as a combinational logic block:
    - RA or RB valid => busA or busB valid after "access time."

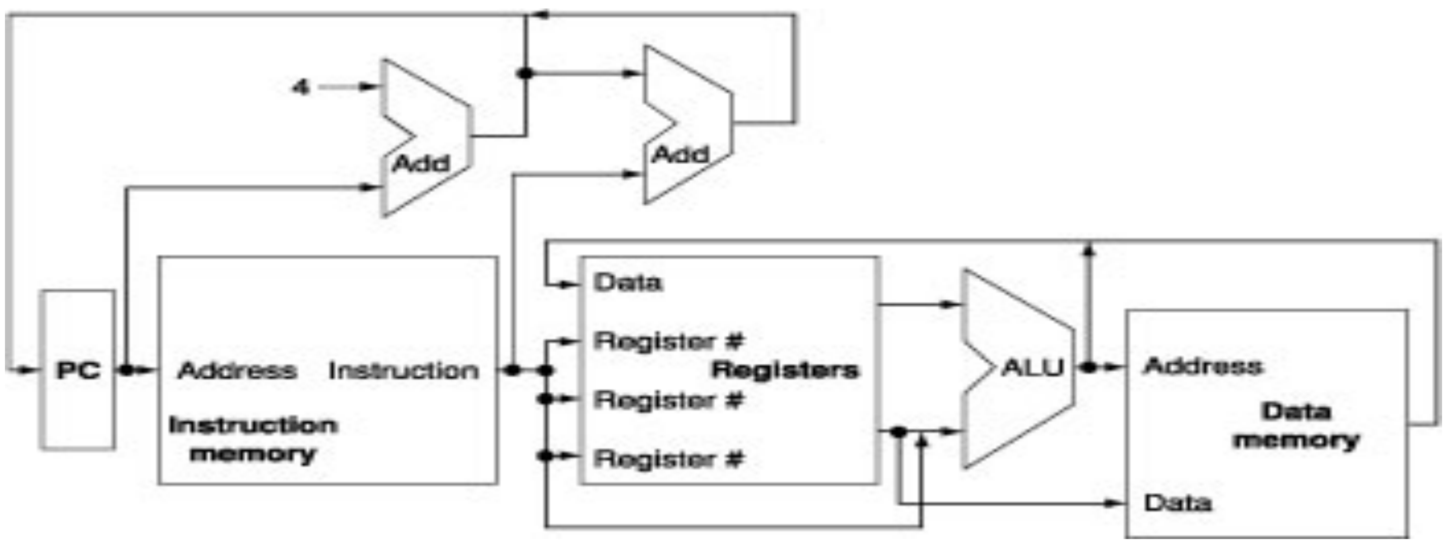
Storage Element: Memory

- Memory
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is selected by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after "access time."



Step 3 : Assemble Datapath

- Register Transfer Requirements –> Datapath Assembly
  – Instruction Fetch
  – Read Operands and Execute Operation
  – Memory Read/Write
  – Register Update



3a: Instruction Fetch Unit

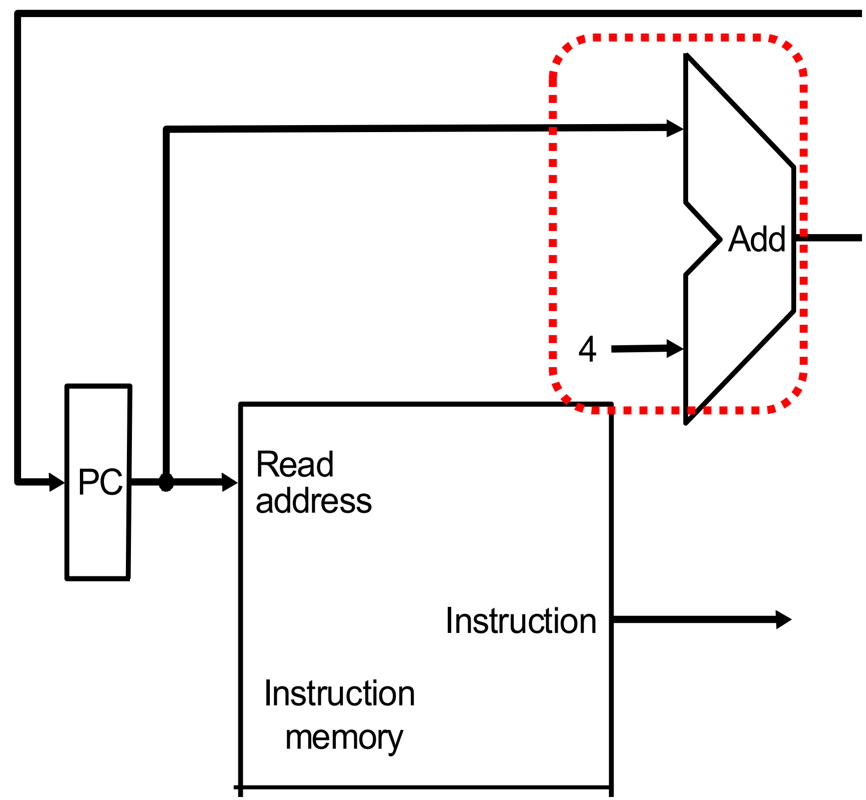
- Instruction fetch unit: common operations
  - Fetch the instruction: **`mem[PC]`**
  - Update the program counter:

    Sequential code

    `PC <- PC + 4`

    Branch and Jump

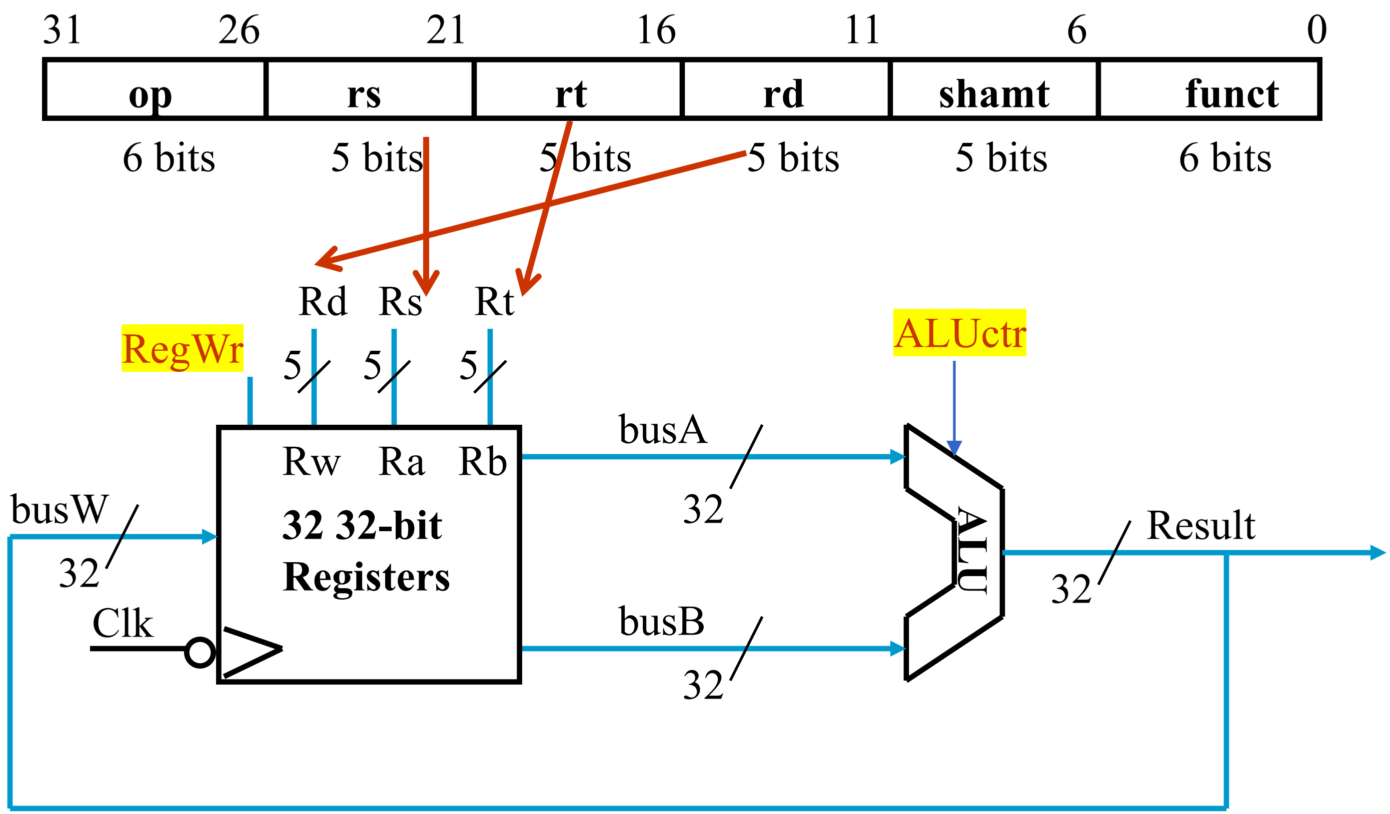
    `PC <- Target addr.`



3b: Add & Subtract

R[rd] <- R[rs] op R[rt]　　　　Example: addU　rd, rs, rt

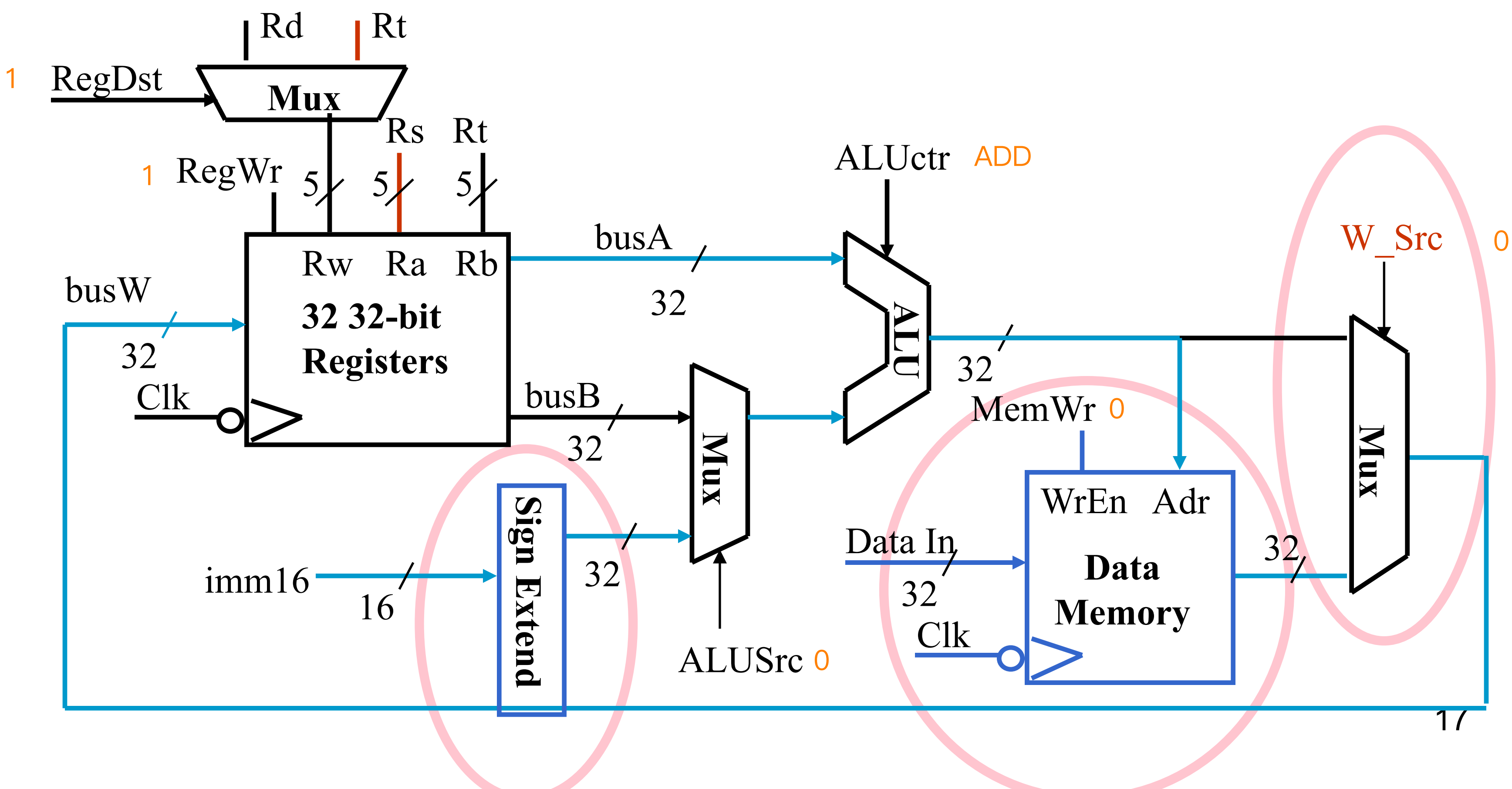
– Ra, Rb, and Rw come from instruction's rs, rt, and rd fields
– ALUctr and RegWr: control logic after decoding the instruction



3c: Load Operations

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

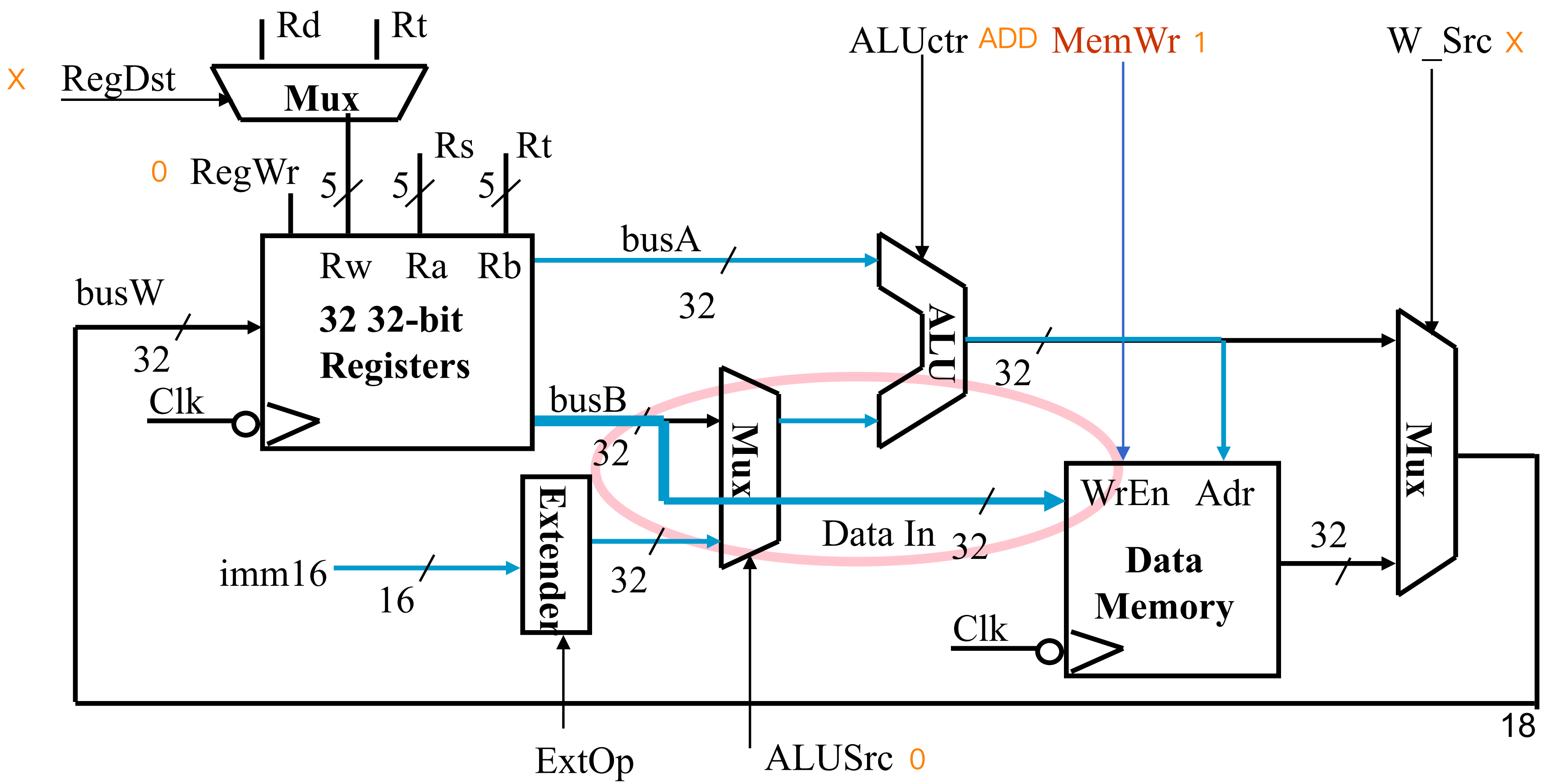
31–26	25–21	20–16	15–0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits



3d: Store Operations

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

31	26	21	16	0
op	**rs**	**rt**	**immediate**	
6 bits	5 bits	5 bits	16 bits	



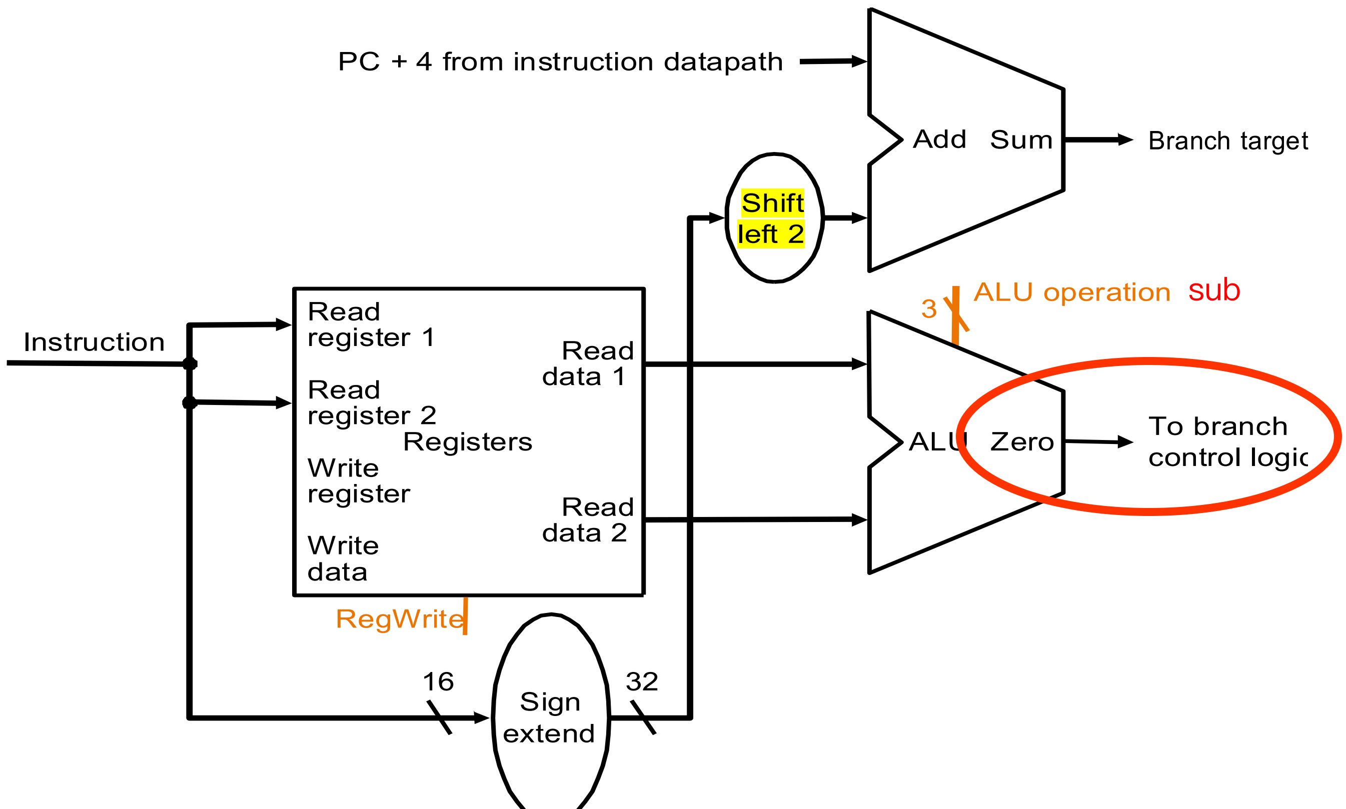
3e: Branch Operations

beq rs, rt, imm16

31	26	21	16	0
op	rs	rt	immediate	
6 bits	5 bits	5 bits	16 bits	

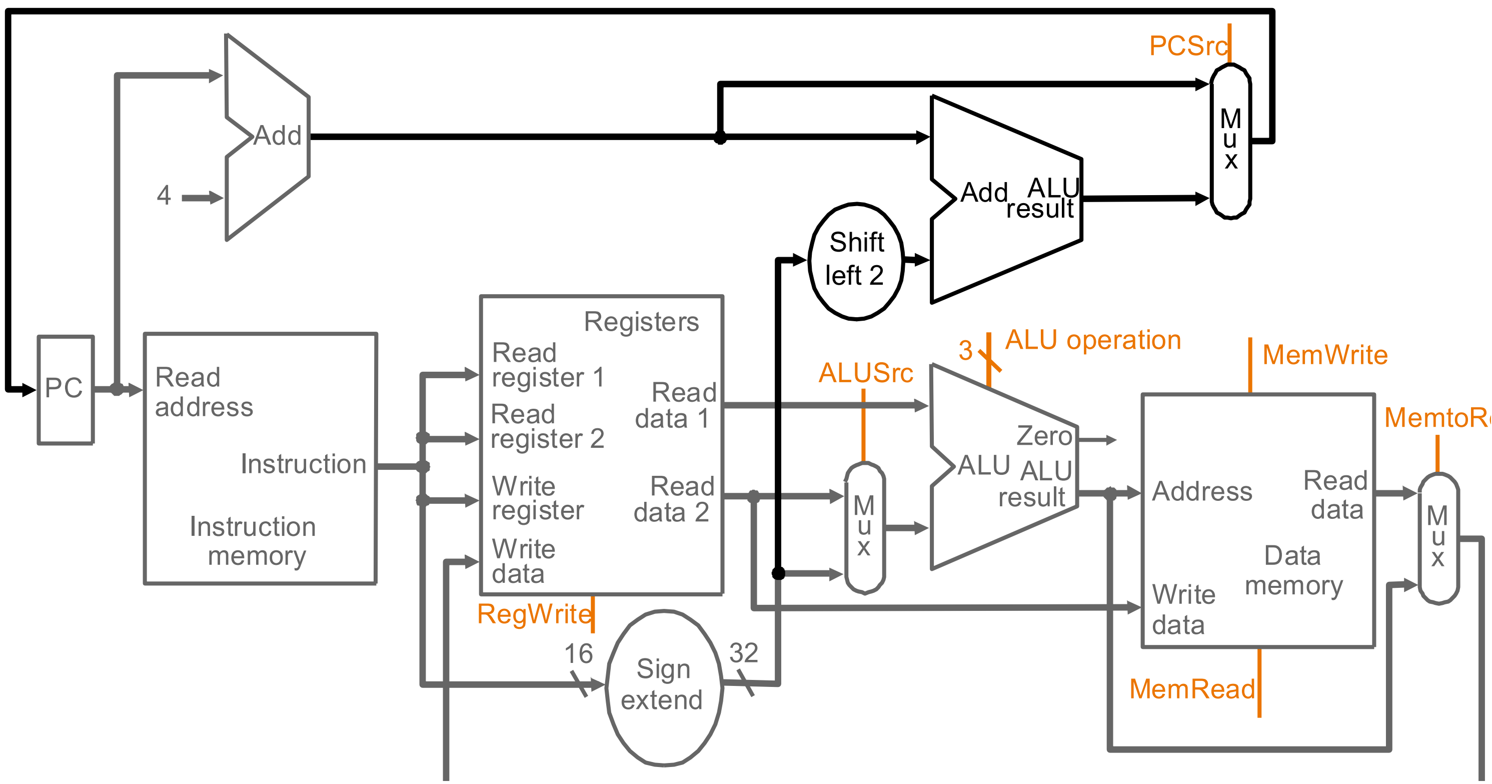
if (rs = = rt)
    PC <- PC + 4 + (SignExt(imm16) x 4)
Else
    PC <- PC + 4

immediate 怎麼來的
為什麼要*4

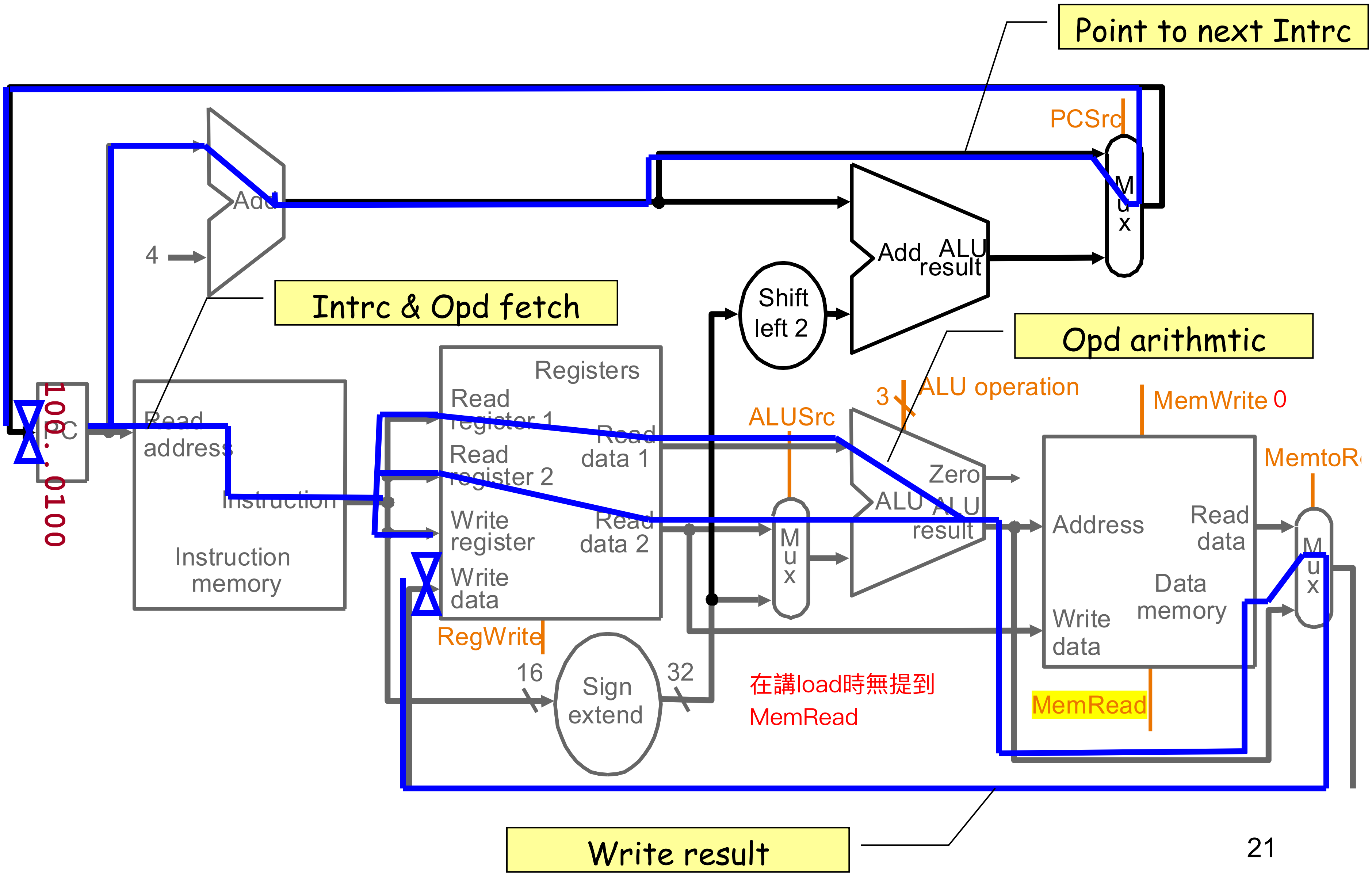


Single-Cycle Datapath

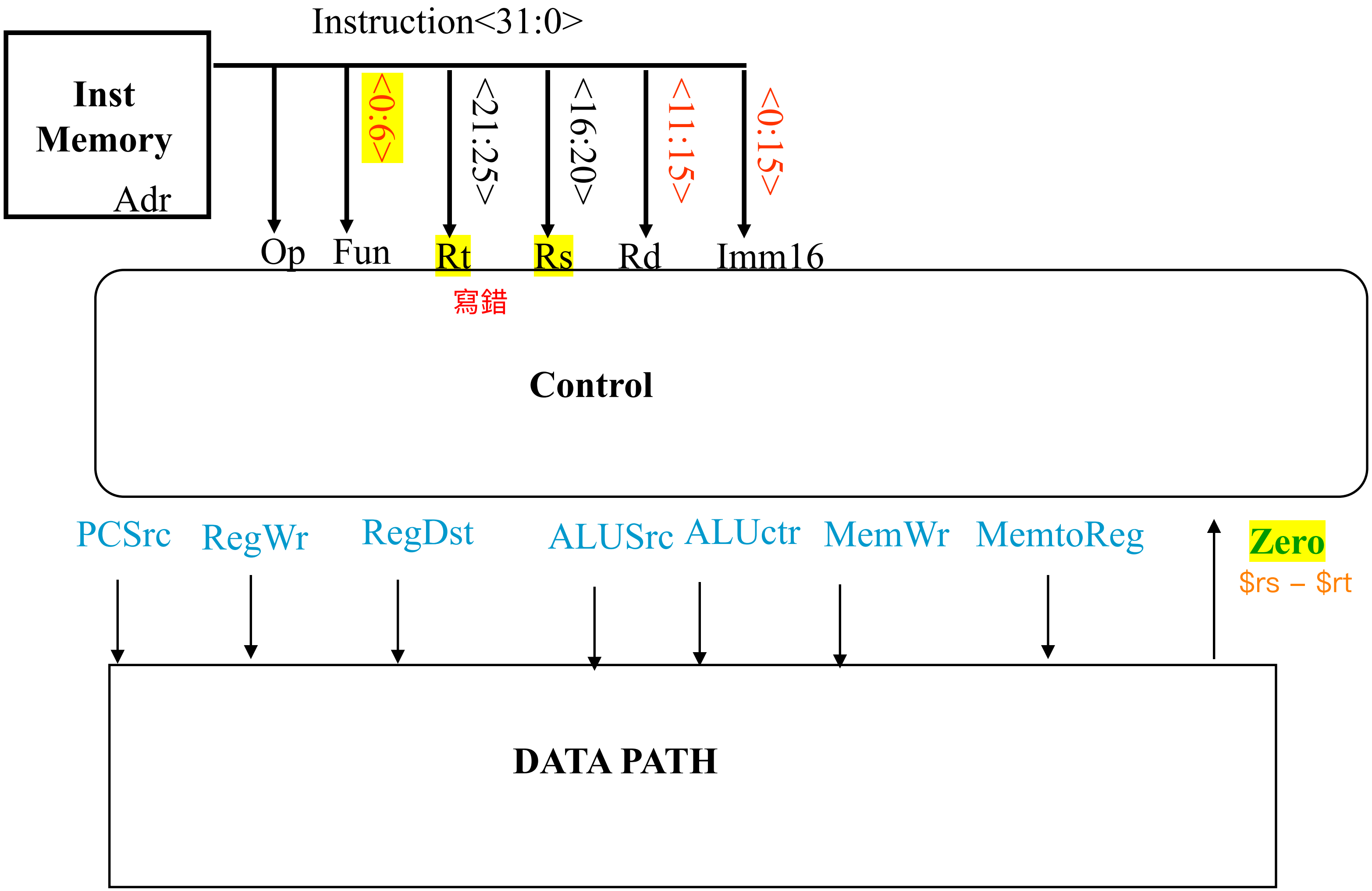
- Attempt to execute all instructions in one clock-cycle.



Data Flow of *add* Instruction



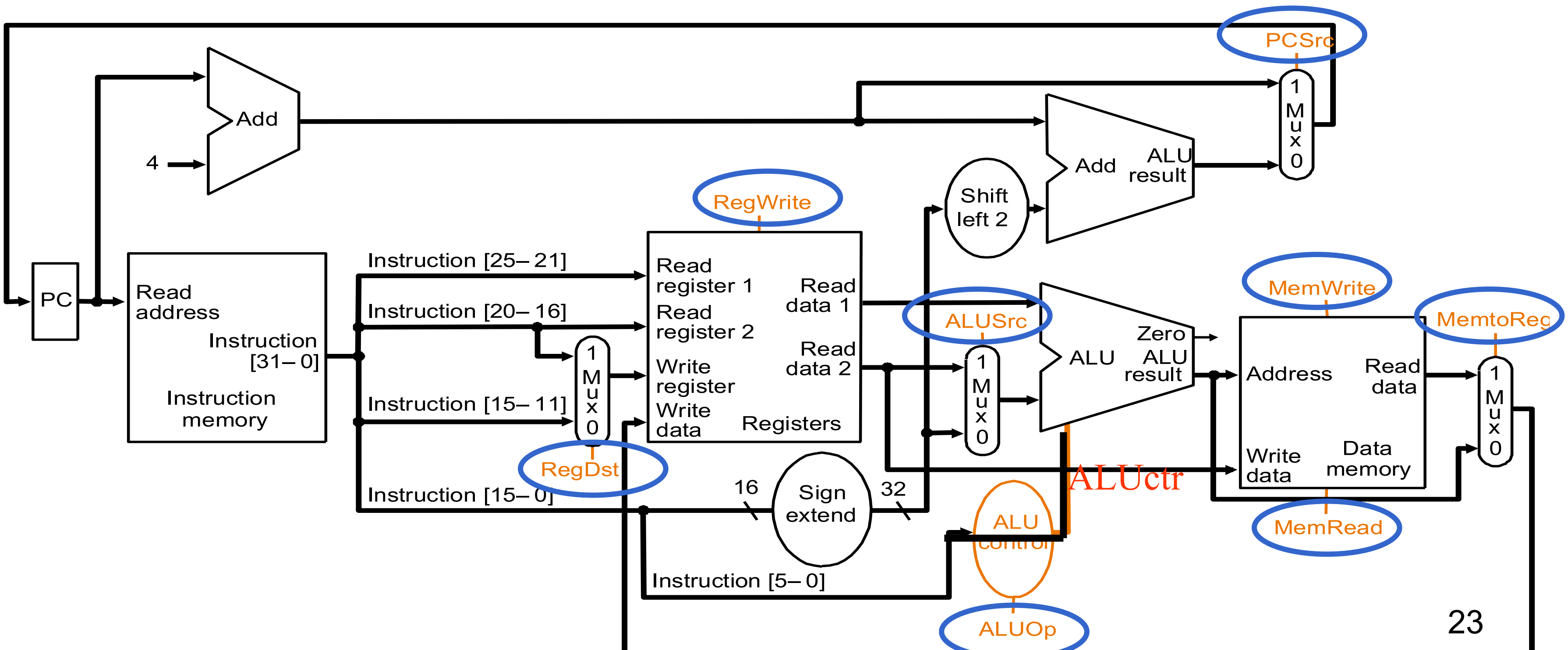
Step 4: Given Datapath: RTL -> Control



- Rs, Rt, Rd and Imed16 hardwired into datapath

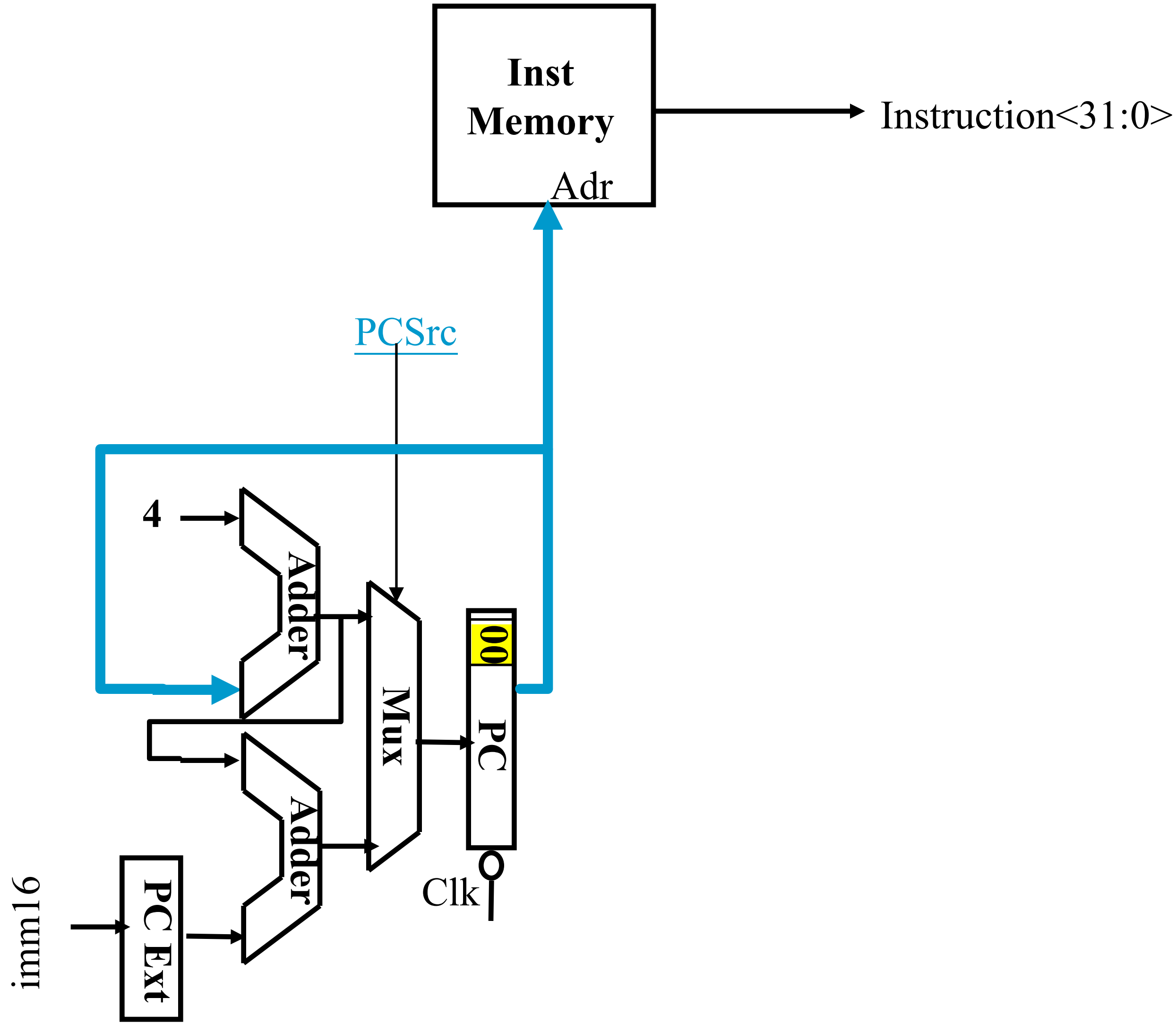
Meaning of Control Signals

- **MemWr:** **write memory**
- **MemtoReg:** **0 => ALU output 1 => Mem**
- **RegDst:** **0 => “rd”; 1 => “rt”**
- **RegWr:** **write dest register**
- **ALUsrc:** **1=> regB; 0=>immed**
- **ALUctr:** **“add”, “sub”**
- **PCSrc:** **1=> PC = PC + 4; 0=> PC = branch target address**
-



Examine control signals: **Add**

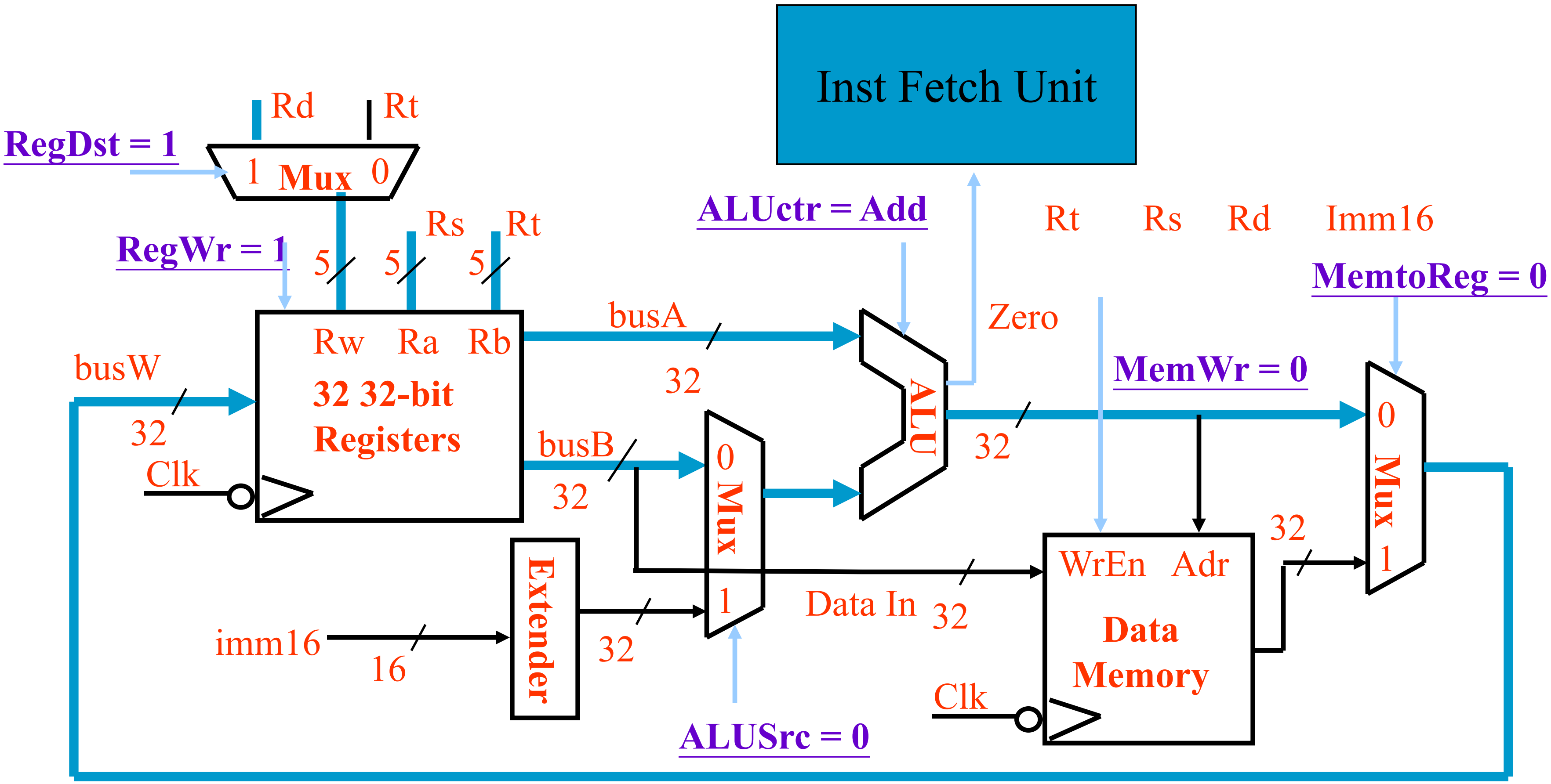
- Fetch the instruction from Instruction memory: Instruction <- mem[PC]
  - This is the same for all instructions



The Single Cycle Datapath during Add

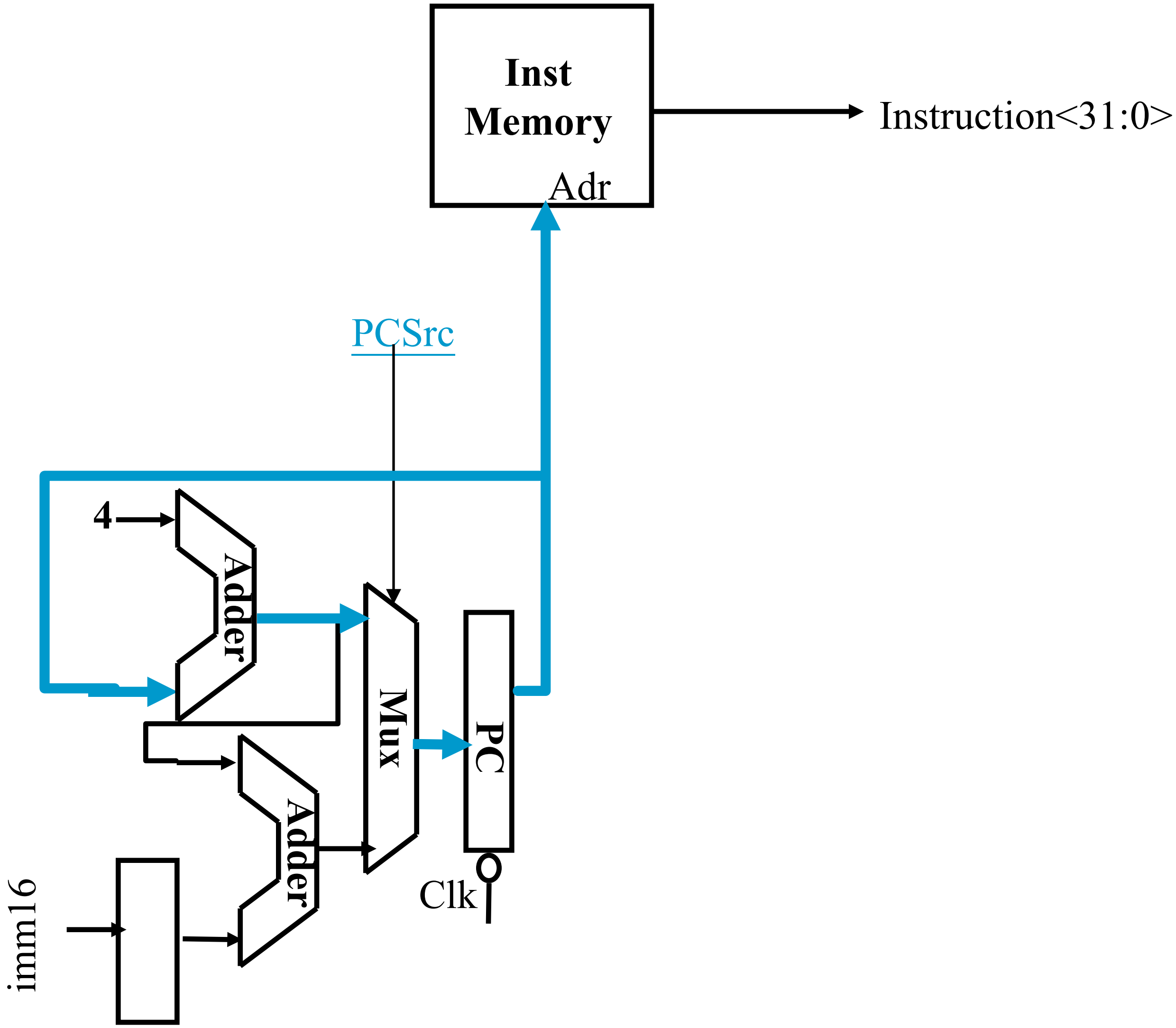
31 op 26	21 rs	16 rt	11 rd	6 shamt	0 funct

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



Instruction Fetch Unit at the End of Add

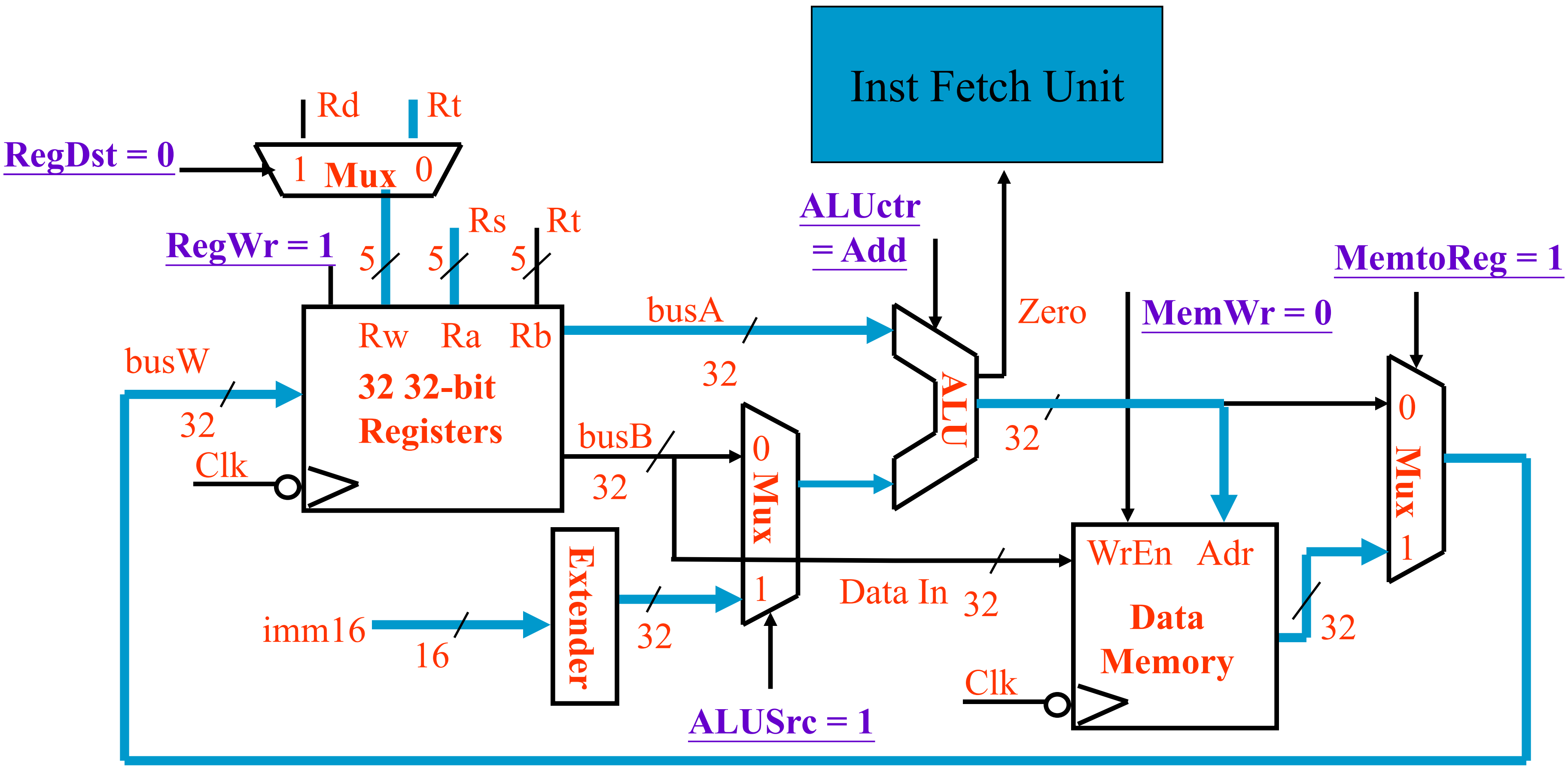
- PC <- PC + 4
  - This is the same for all instructions except: Branch and Jump



The Single Cycle Datapath during Load

31	26	21	16	0
op	rs	rt	immediate	

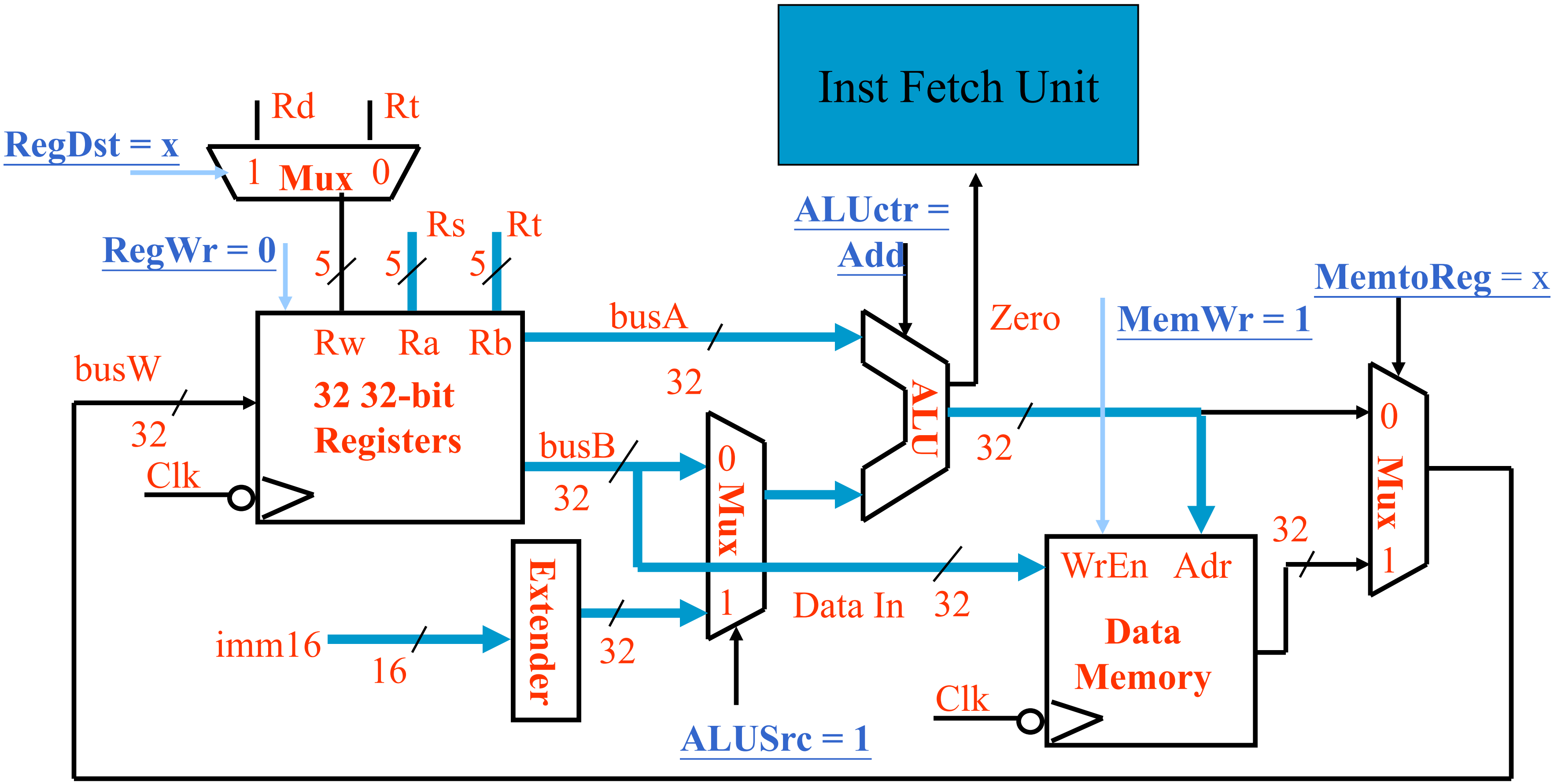
- R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



The Single Cycle Datapath during Store

31–26	25–21	20–16	15–0
op	**rs**	**rt**	**immediate**

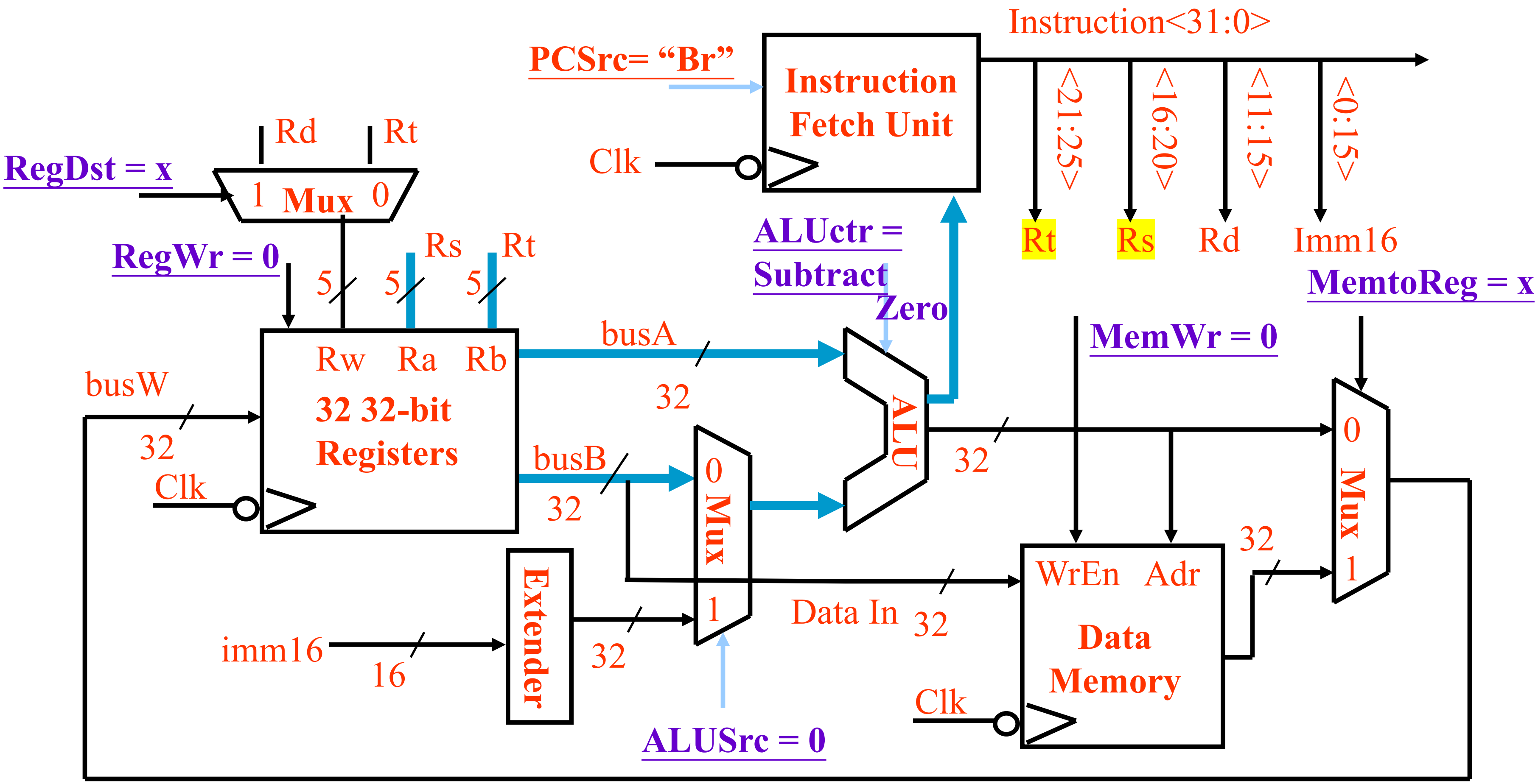
- Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



The Single Cycle Datapath during Branch (beq)

31	26	21	16	0
op	rs	rt	immediate	

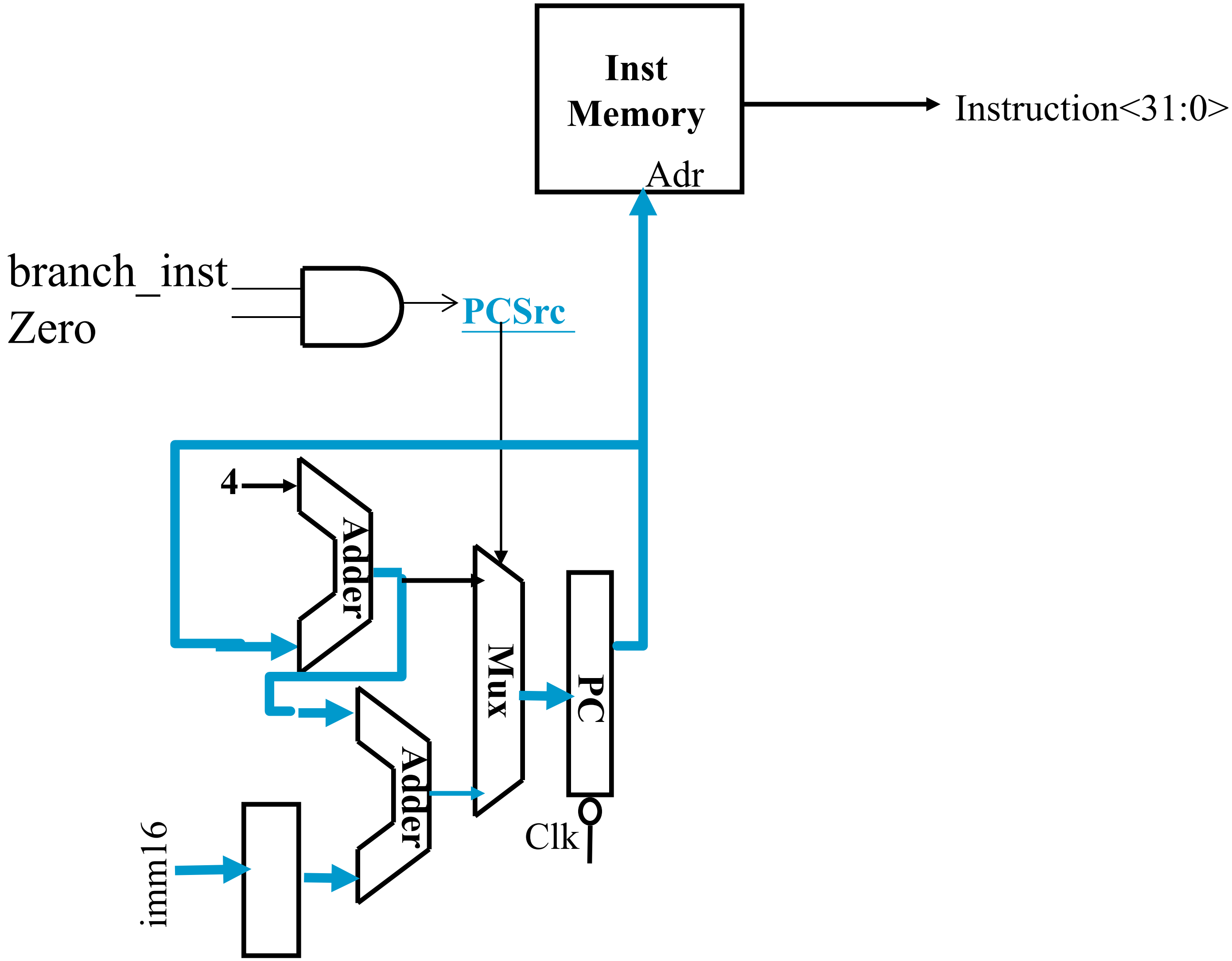
- if (R[rs] - R[rt] == 0) then Zero <- 1 ; else Zero <- 0



Instruction Fetch Unit at the End of Branch

31	26	21	16	0
op	**rs**	**rt**	**immediate**	

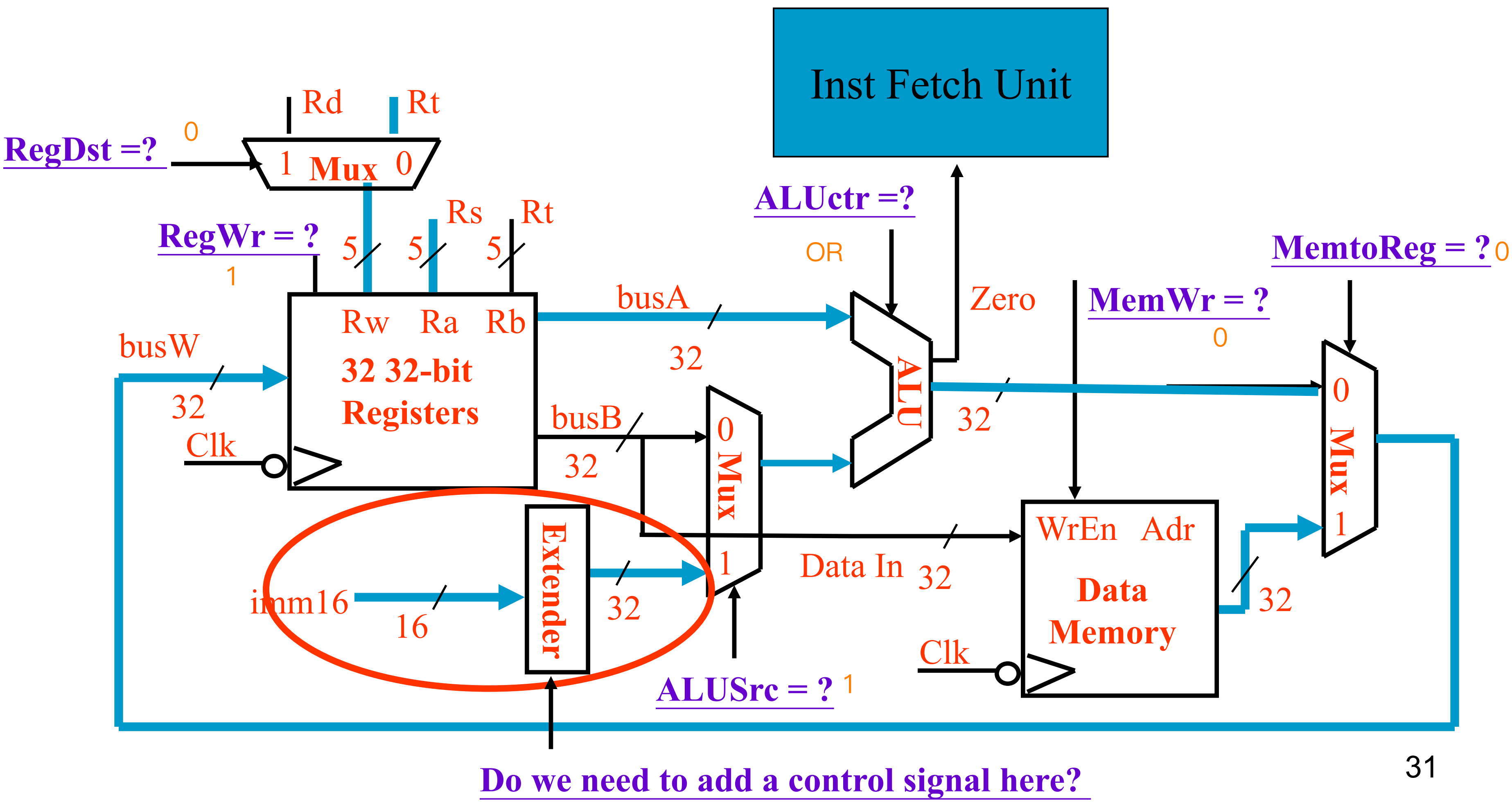
- if (Zero == 1)   then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4



Exercise: The Single Cycle Datapath during Ori

31	26	21	16	0
op	rs	rt	immediate	

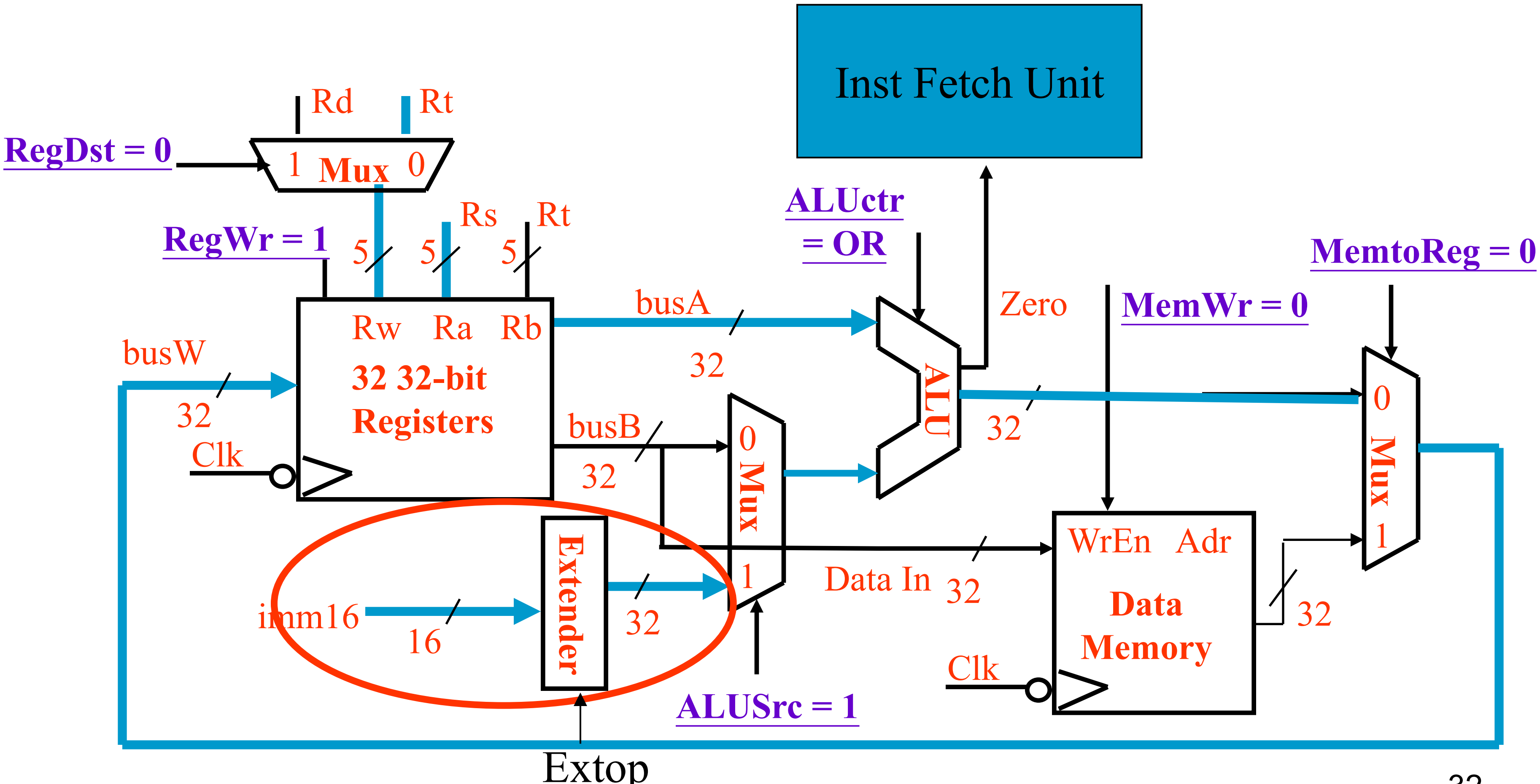
- R[rt] <- R[rs] or ZeroExt[imm16]}



Exercise: The Single Cycle Datapath during Ori

31	26	21	16	0
op	**rs**	**rt**	**immediate**	

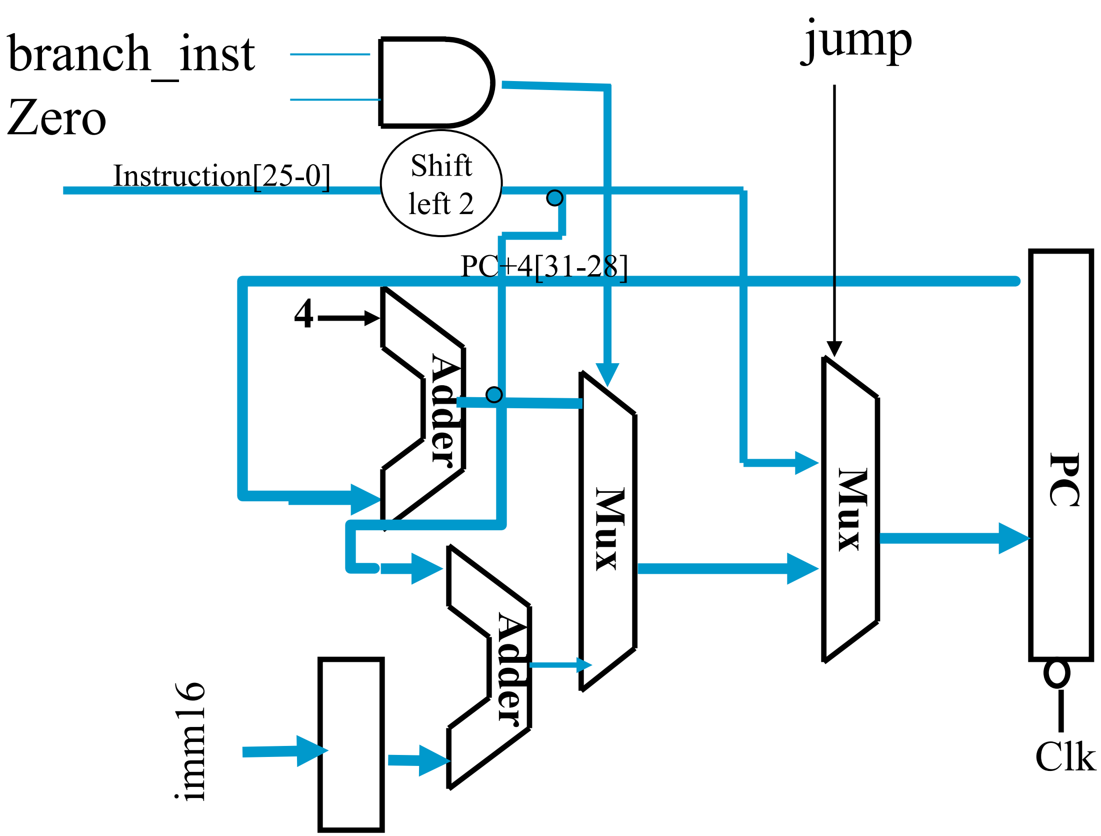
- R[rt] <- R[rs] or ZeroExt[imm16]}



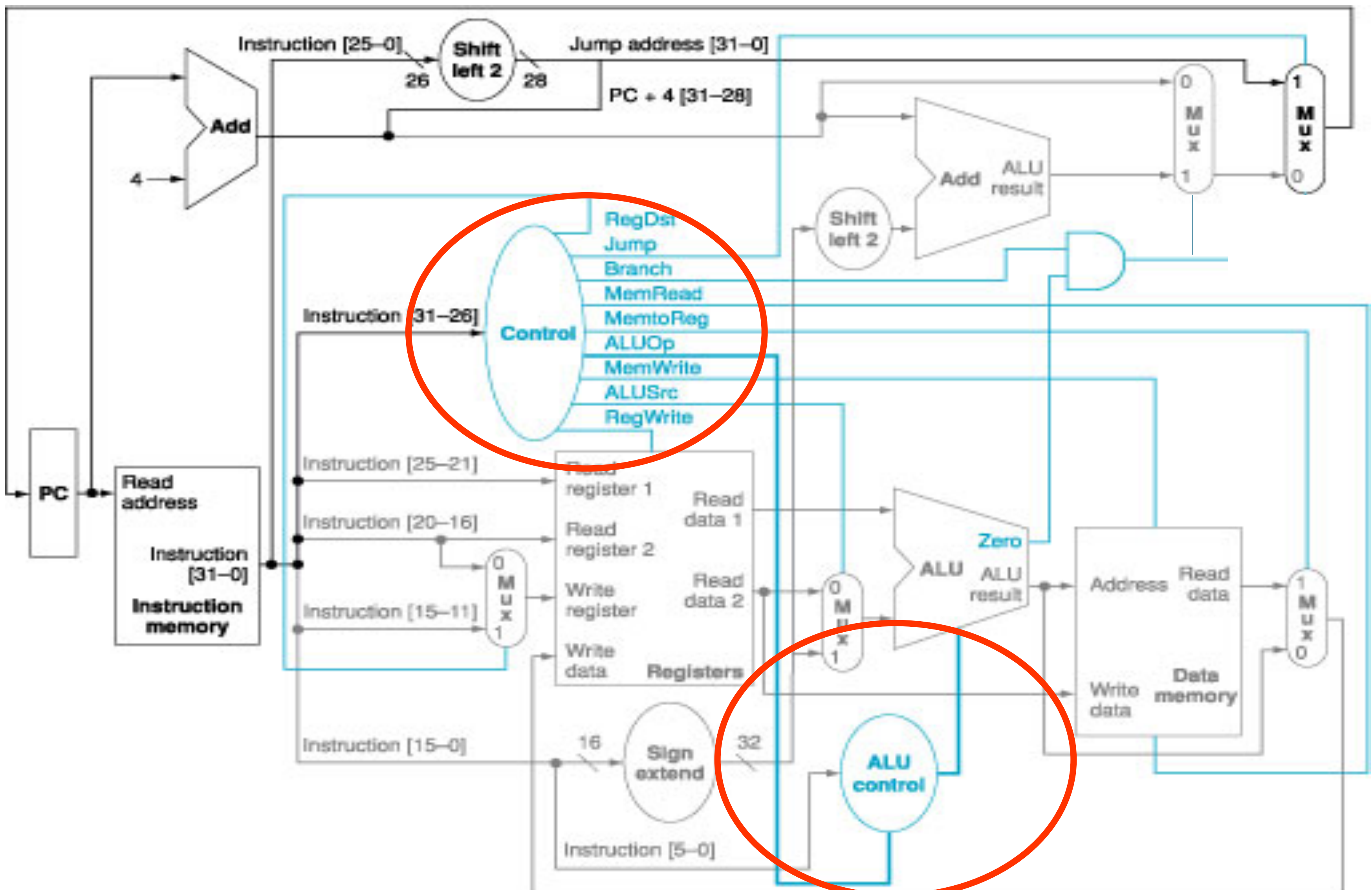
Exercise : Implementing Jumps

000010	address
31:26	25:0

- The jump address can be obtained by the concatenation of:
  - The upper 4 bits of the current PC+4
  - The 26-bit immediate field of the jump instruction



Step 5: Assemble the Control Unit



A Summary of the Control Signals

See MIPS instruction Set material

	func	10 0000	10 0010	We Don't Care :-)				
	op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
		add	**sub**	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst		1	1	0	0	x	x	x
ALUSrc		0	0	1	1	1	0	x
MemtoReg		0	0	0	1	x	x	x
RegWrite		1	1	1	1	0	0	0
MemWrite		0	0	0	0	1	0	0
Branch		0	0	0	0	0	1	0
Jump		0	0	0	0	0	0	1
ExtOp		x	x	0	1	1	x	x
ALUctr<2:0>		Add	Subtract	Or	Add	Add	Subtract	xxx

31 26 21 16 11 6 0

op	rs	rt	rd	shamt	funct

R-type

op	rs	rt	immediate

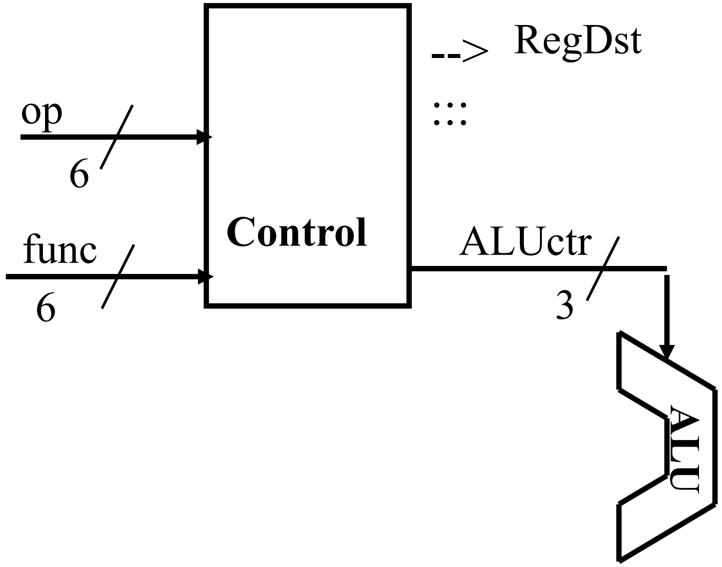
I-type

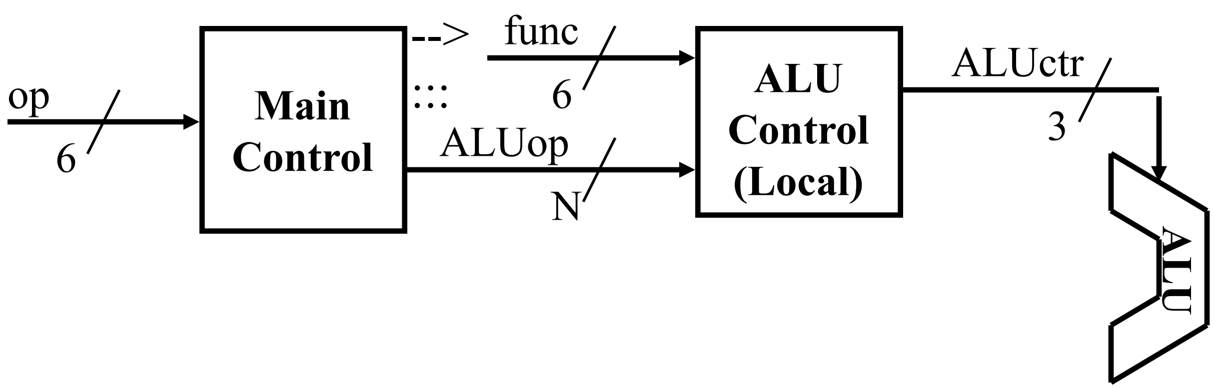
op	target address

J-type

ALU control lines	Function
000	AND
001	OR
010	add
110	subtract
111	set on less than

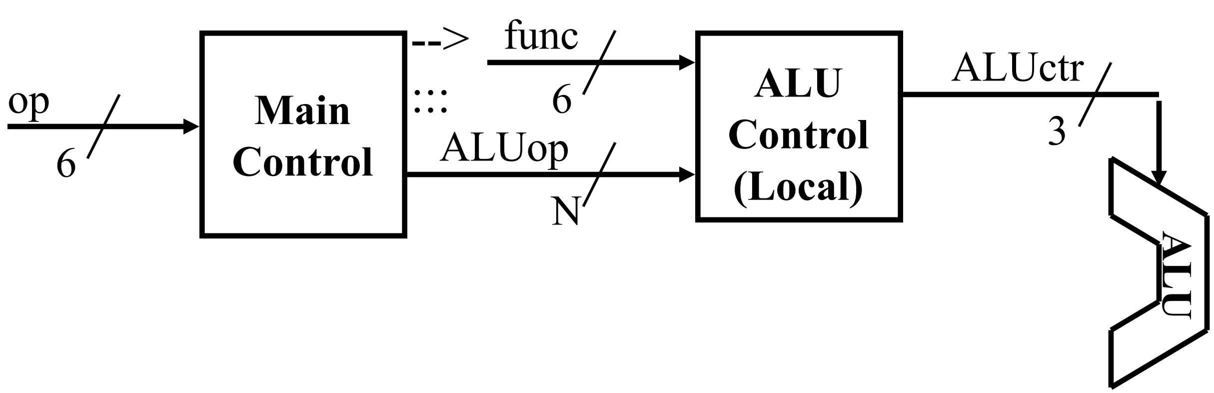
Control Unit





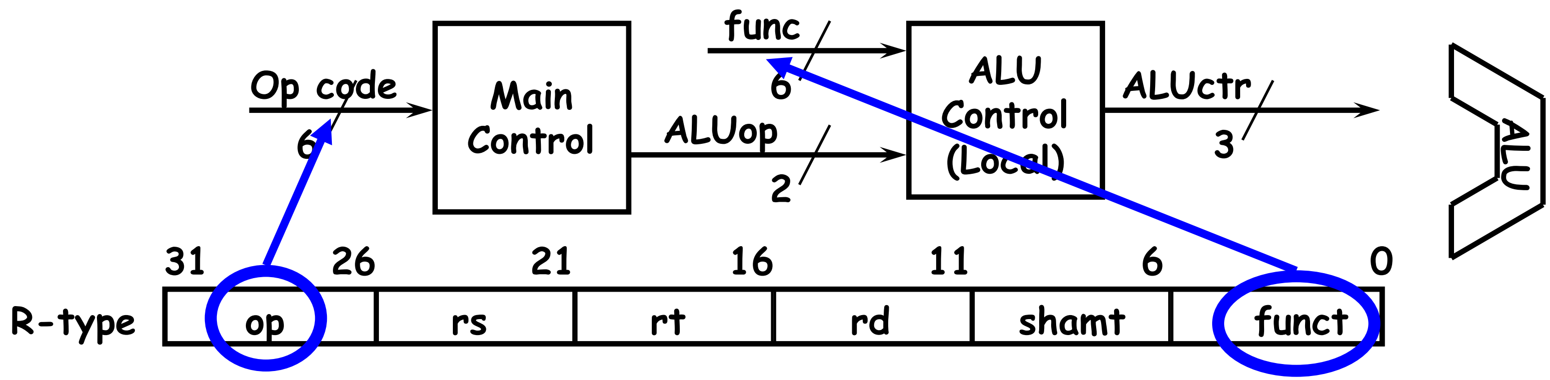
The Concept of Multi-level Decoding

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



ALU Control

- ALUop is 2-bit wide to represent:
  - "I-type" requiring the ALU to perform:
    - (00) add for load/store and (01) sub for beq and (10) for ori
  - "R-type" (11, need to refer to func field)



	R-type	lw	sw	beq	ori
ALUop (Symbolic)	"R-type"	Add	Add	Subtract	or
ALUop<2:0>	11	00	00	01	10

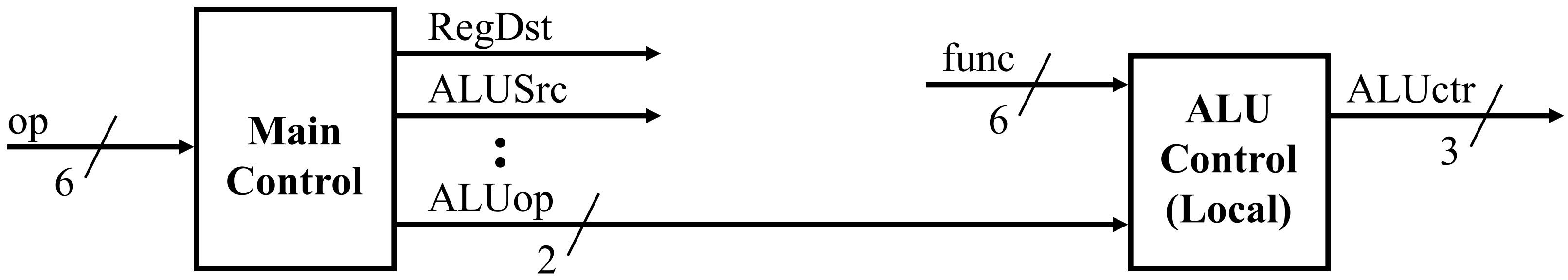
The Truth Table for ALUctr

funct<3:0>	Instruction Op.
100000	add
100010	subtract
100100	and
100101	or
101010	set-on-less-than

	R-type	ori	lw	sw	beq
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract
ALUop<1:0>	1 1	10	00	00	01

ALUop		**func**						**ALU Operation**	**ALUctr**		
bit<1>	**bit<0>**	**bit<5>**	**bit<5>**	**bit<3>**	**bit<2>**	**bit<1>**	**bit<0>**		**bit<2>**	**bit<1>**	**bit<0>**
0	0	x	x	x	x	x	x	Add	0	1	0
0	1	x	x	x	x	x	x	Subtract	1	1	0
1	0	x	x	x	x	x	x	Or	0	0	1
1	1	x	x	0	0	0	0	Add	0	1	0
1	1	x	x	0	0	1	0	Subtract	1	1	0
1	1	x	x	0	1	0	0	And	0	0	0
1	1	x	x	0	1	0	1	Or	0	0	1
1	1	x	x	1	0	1	0	Set on <	1	1	1

The “Truth Table” for the Main Control



op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop <1>	1	0	0	0	1	x
ALUop <0>	1	1	0	0	0	x

The “Truth Table” for RegWrite

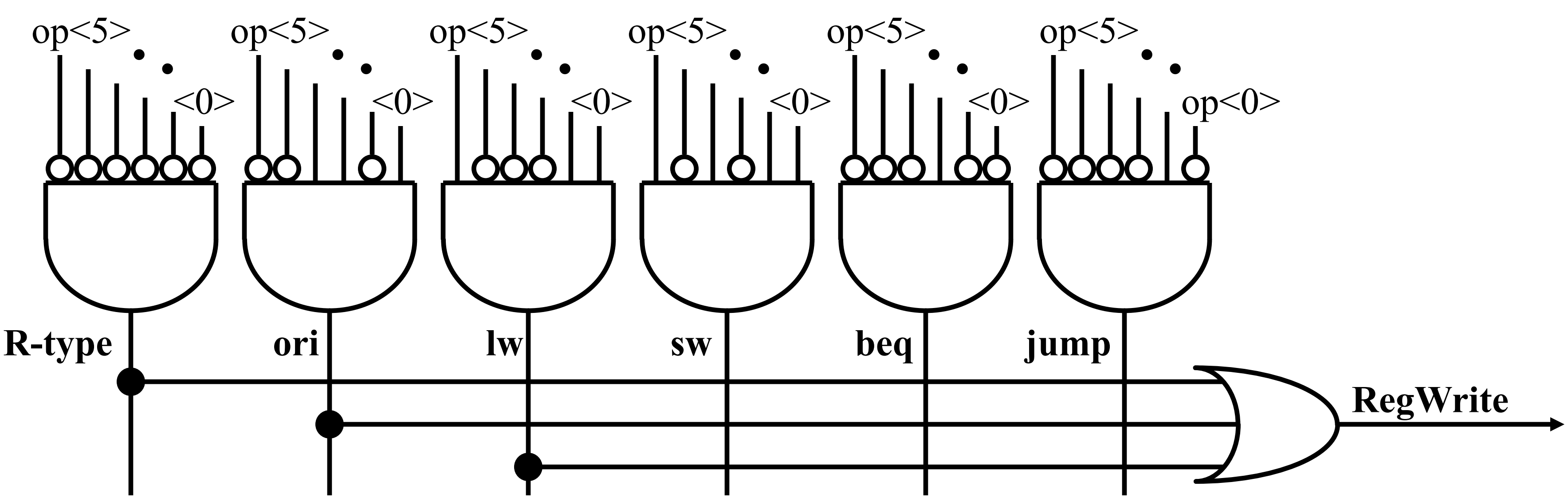
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	0	0	0

- RegWrite = R-type + ori + lw

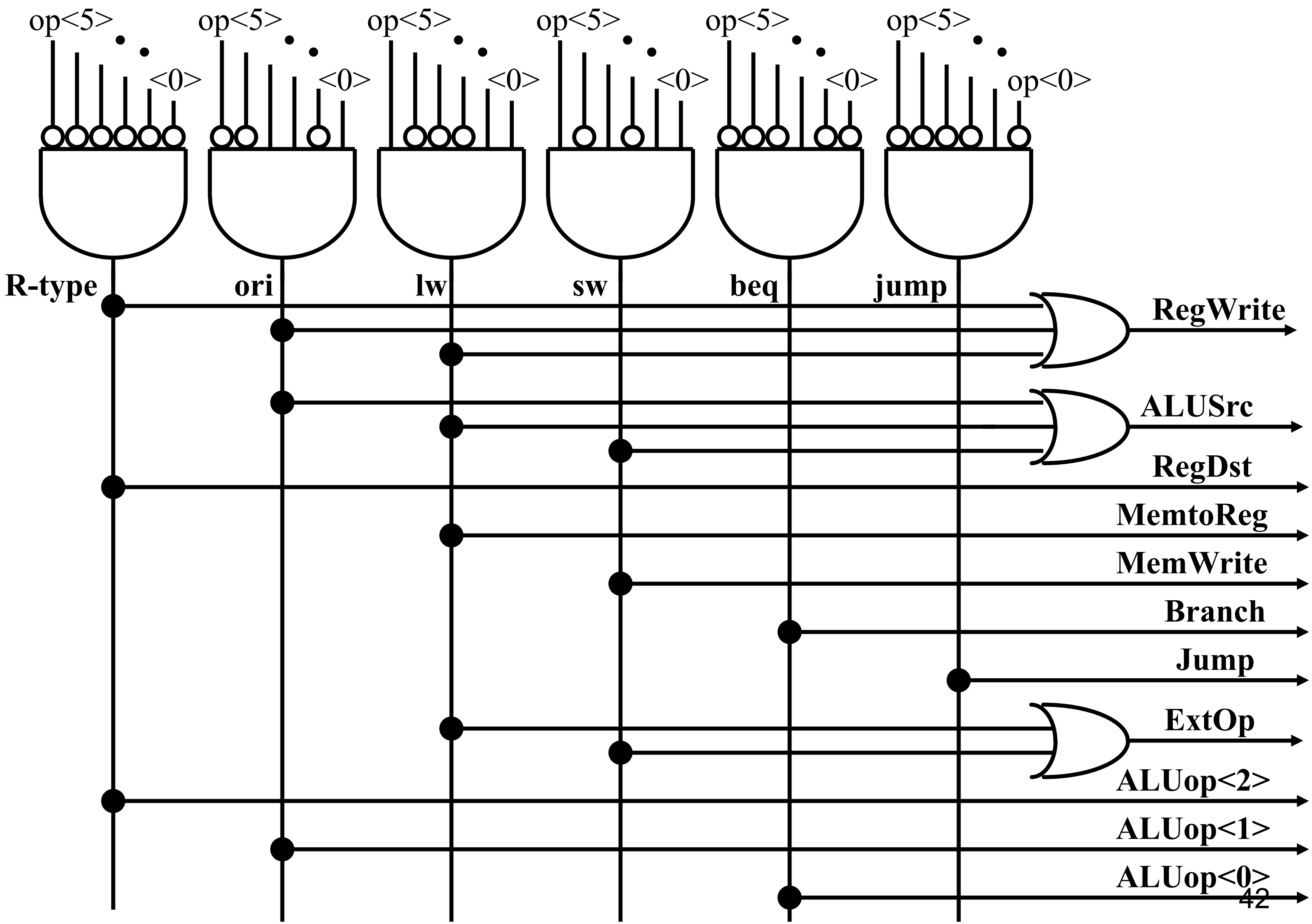
  = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

  \+ !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)

  \+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



PLA Implementation of the Main Control



Performance of Single-Cycle Machines

- **Assumption**
  - **Memory units: 200 ps**
  - **ALU and adders: 100 ps**
  - **Register file (read or write): 50 ps**
  - **Instruction mix:**
    - **25% loads, 10% stores, 45% ALU instructions, 15% branches, and 5% jumps.**
- **Comparison**
  - **Every instruction operates in 1 clock cycle of a fixed length.**
  - **Every instruction executes in 1 clock cycle using a variable-length clock.**

Instruction Class	Functional Units used by the instruction class				
R-Type	Inst Fetch	Register Access	ALU	Register Access	
Load Word	Inst Fetch	Register Access	ALU	Memory Access	Register Access
Store word	Inst Fetch	Register Access	ALU	Memory Access	
Branch	Inst Fetch	Register Access	ALU		
Jump	Inst Fetch				

Performance of Single-Cycle Machines (cont.)

- Recall

  CPU execution time = Instruction count $\times$ CPI $\times$ Clock cycle time

Since CPI must be 1, we can simplify this to

  CPU execution time = Instruction count $\times$ Clock cycle time

- For fixed clock cycle implementation
  - The clock cycle for each instruction is determined by the longest instruction, load, which is 600 ps (200+50+100+200+50).
- For variable clock cycle implementation
  - The average time per instruction with a variable clock is
  - 600 $\times$ 25% + 550 $\times$ 10% + 400 $\times$ 45% + 350 $\times$ 15% + 200 $\times$ 5% = 447.5 ps (see page 316)
- The variable clock implementation would be faster by

$$\frac{600}{447.5} = 1.34$$

NEXT TIME: Pipelining