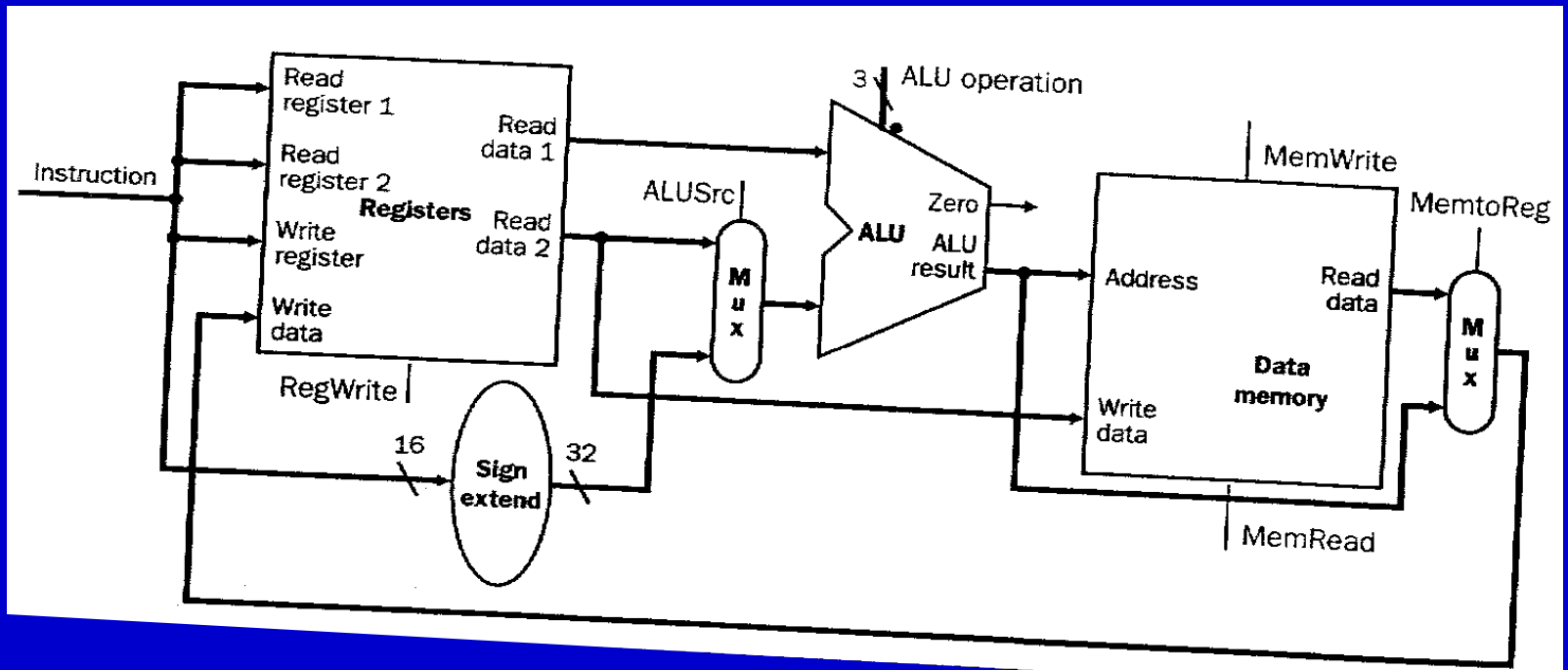


Single clock implementation

- Execute all instructions in one clock cycle
- No data path resource can be used more than once
- Memory for instructions separate from data
- Anywhere we want to choose a signal from two paths we use mux + selector

Combined R-type + Load/Store



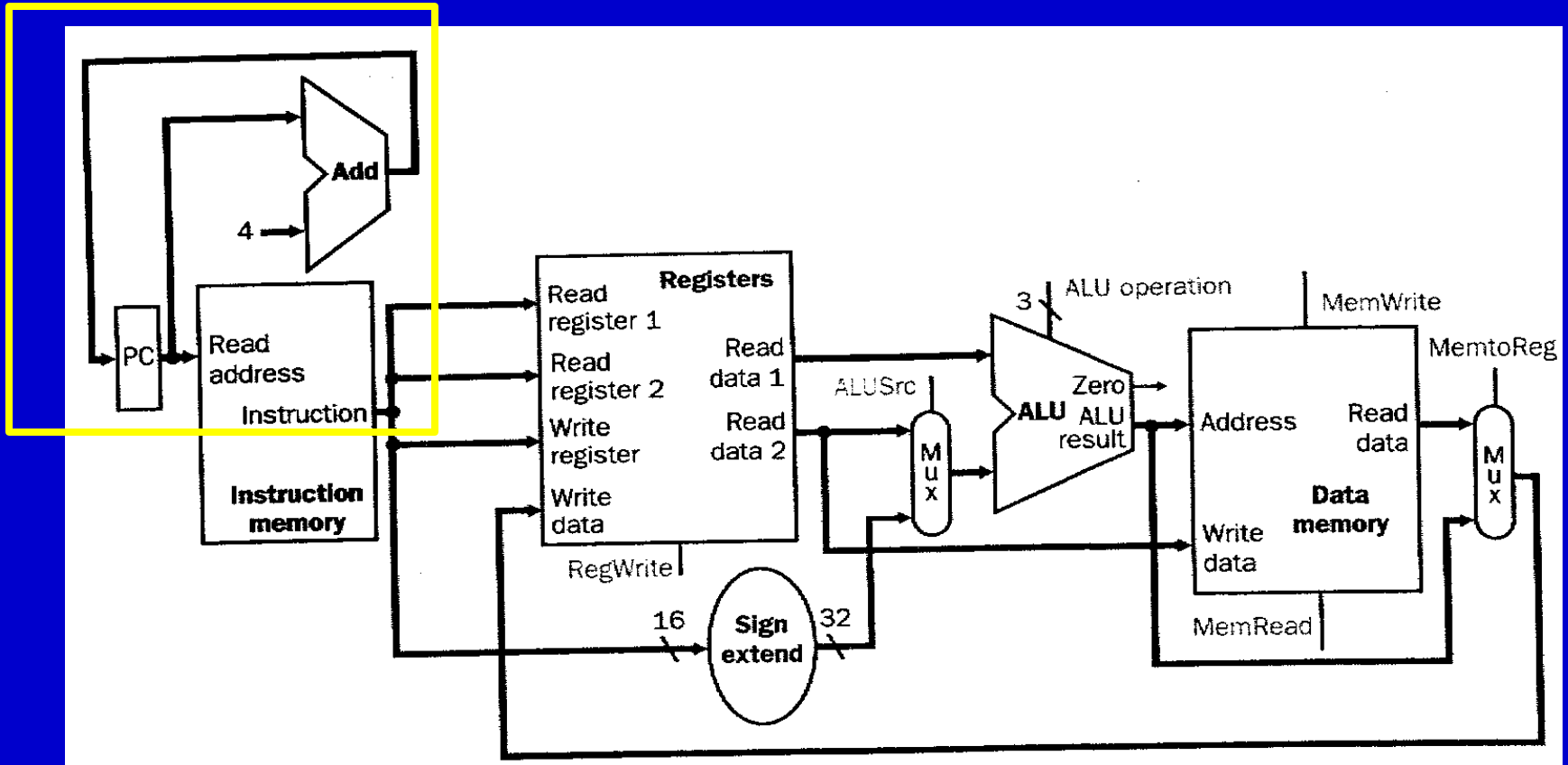
ALUSrc: 1 bit selector of

- a) data from rt (R-type)
- b) sign extended immediate (Load/Store)

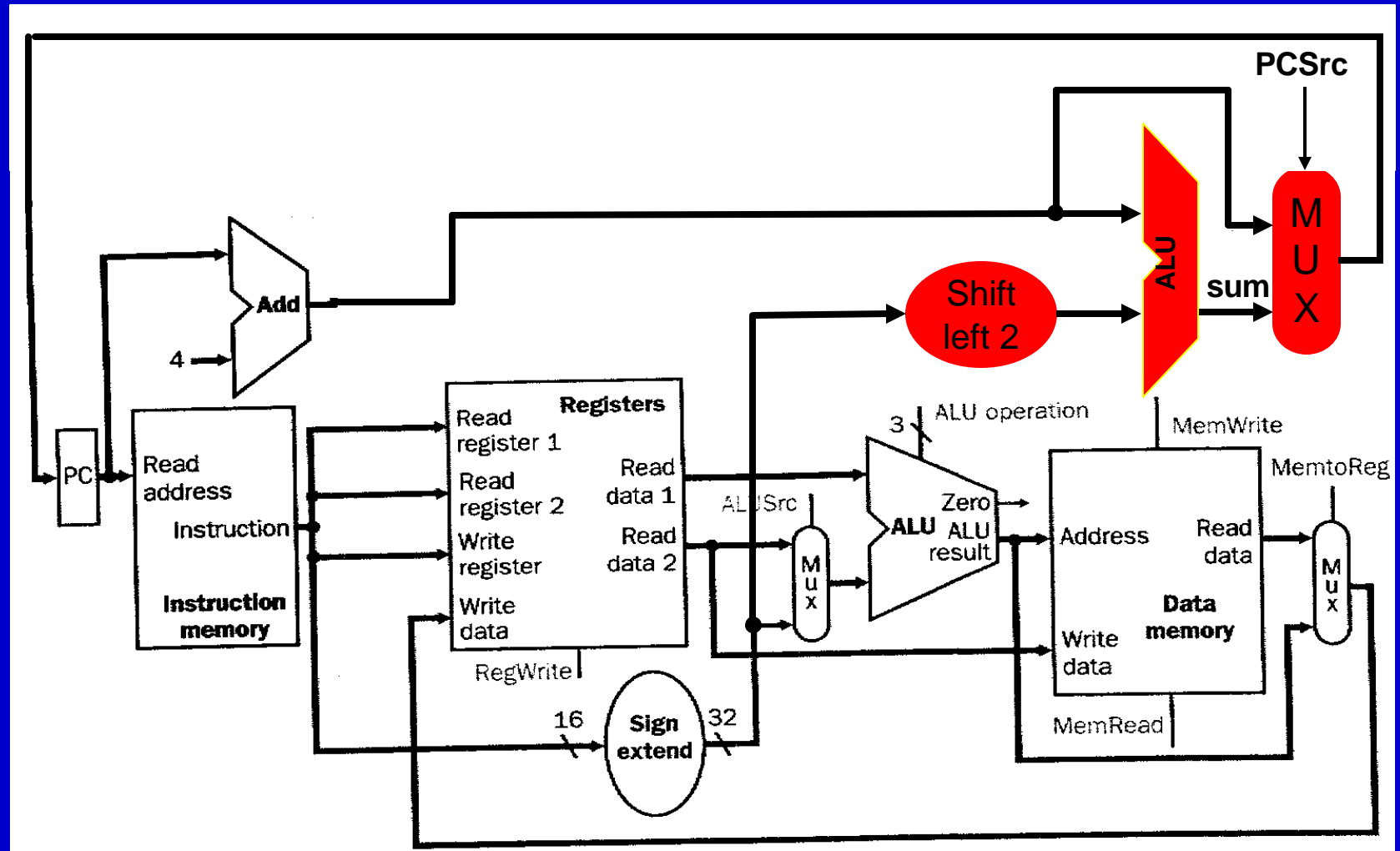
MemtoReg: 1 bit selector of

- a) data found at calculated address in memory (Load)
- b) ALU result from two register ALU operation

Adding instruction fetch



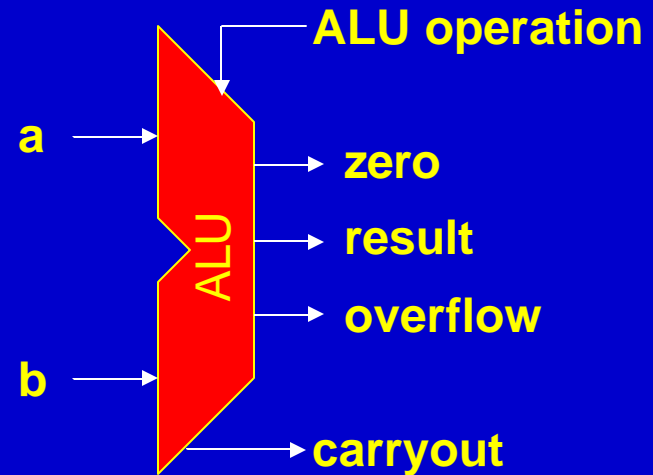
Adding branches

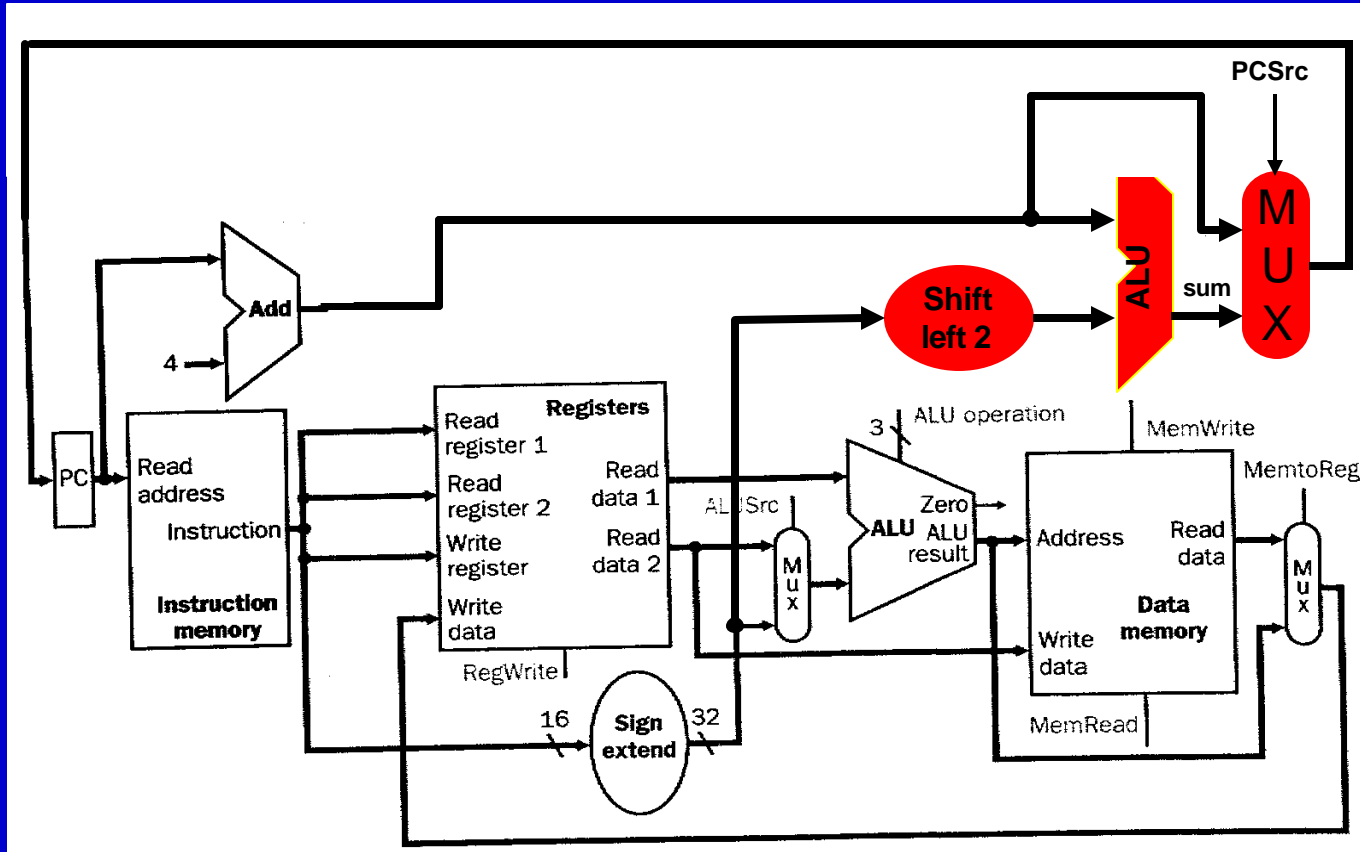


ALU control

- Our latest version of ALU needs 3-bits to control Mux's
 - Binvert: 1-bit control to operate on B or B-inverse (sub)
 - Operation: 2-bit control to perform AND, OR, add, slt

Combine Binvert and Operation into set of 3 control lines	
ALU operation	Function
000	AND
001	OR
010	ADD
110	SUB
111	Set on less than





ALU control input is 3-bits → AND(000), OR(001), ADD(010), SUB(110), SLT(111)

Instruction class determines ALU operation:

lw/sw → ALU computes mem address (ADD)

branch → ALU computes comparison (SUB)

R-type → ALU control determined by 6-bit func field

What function should ALU perform?

- Determined by instruction
- Op-code + function code => ALUOp
- Observation: There are 3 possibilities => 2 bits
 - Load/store uses ALU to add (always)
 - BEQ uses ALU to subtract (always)
 - R-type uses ALU to perform 1 of 5 operations
- Use the function-field and a 2-bit control field (ALUOp) as input to produce 3-bit ALU (operation) control

Determining ALU control bits

I-type instructions

Input signal to ALU

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 equal	01	branch equal	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 than	101010	set on less than	111

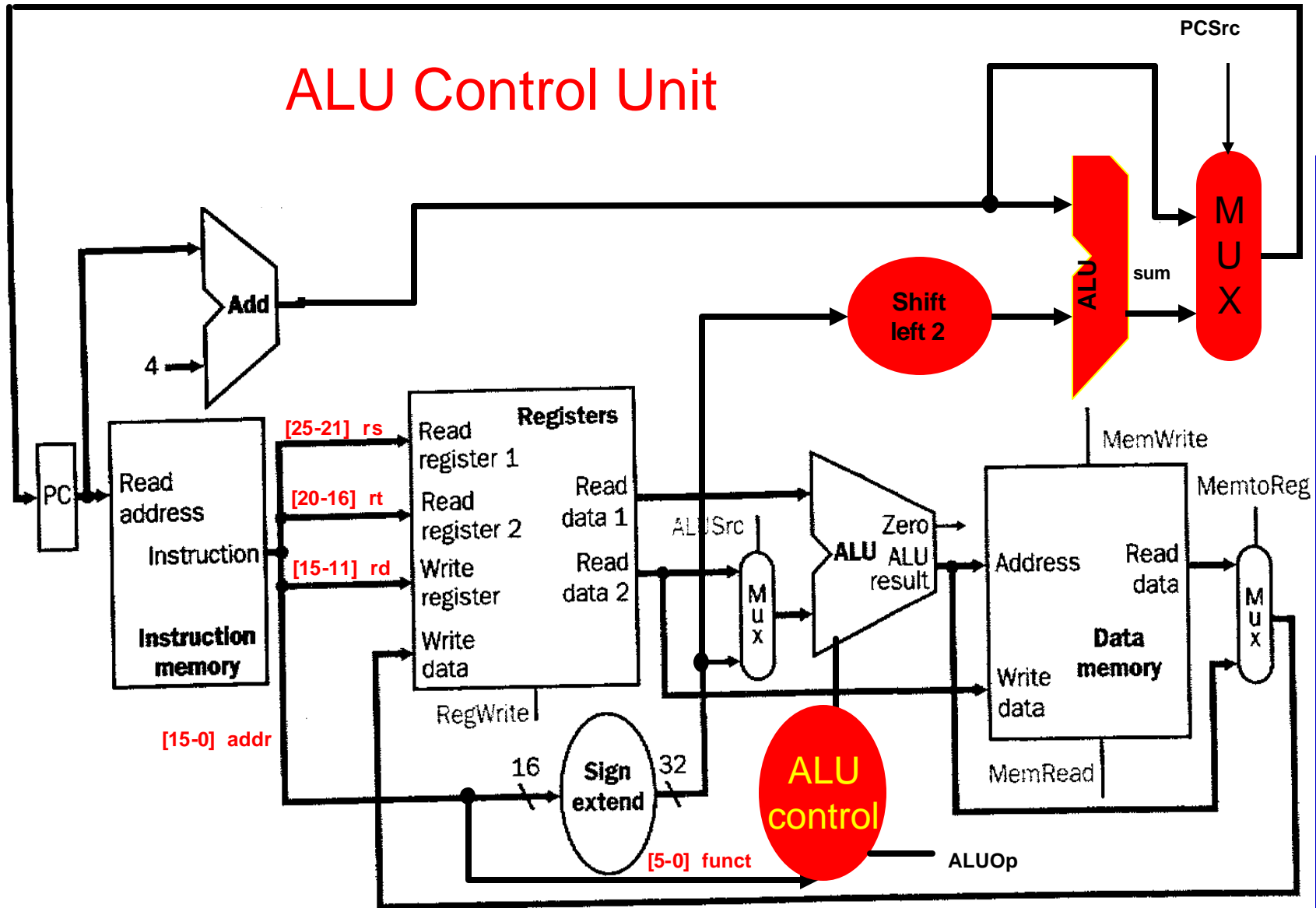
Based on op-code: generated by main control unit

Obtaining 3-bit ALU control from ALUOp + Funct

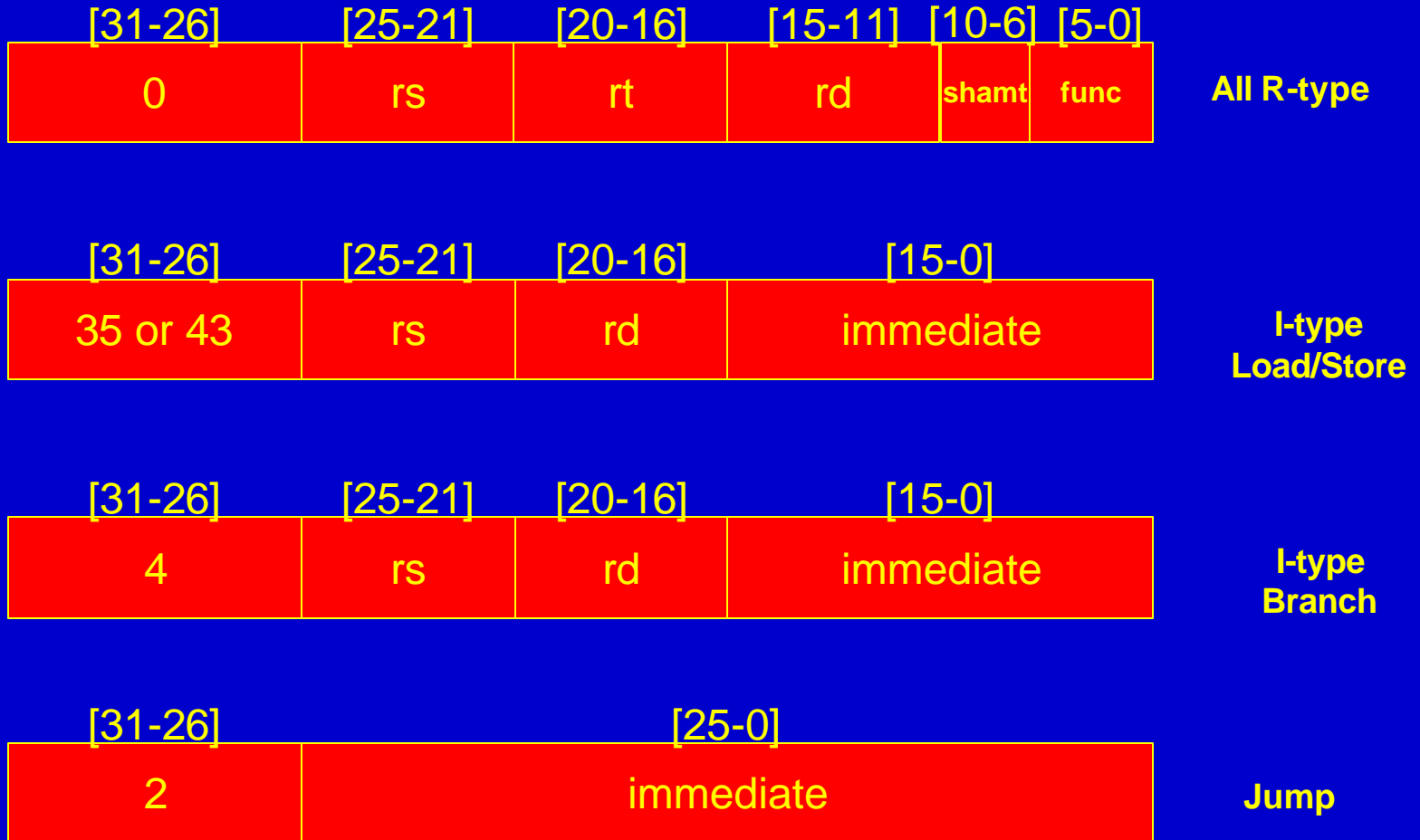
ALUOp		Funct field						Operation	Action
ALUOp1	ALUOp2	F5	F4	F3	F2	F1	F0		
0	0	X	X	X	X	X	X	010	Add
X	1	X	X	X	X	X	X	110	Sub
1	X	X	X	0	0	0	0	010	Add
1	X	X	X	0	0	1	0	110	Sub
1	X	X	X	0	1	0	0	000	And
1	X	X	X	0	1	0	1	001	Or
1	X	X	X	1	0	1	0	111	Set on less than

Resulting PLA is ALU control logic

ALU Control Unit

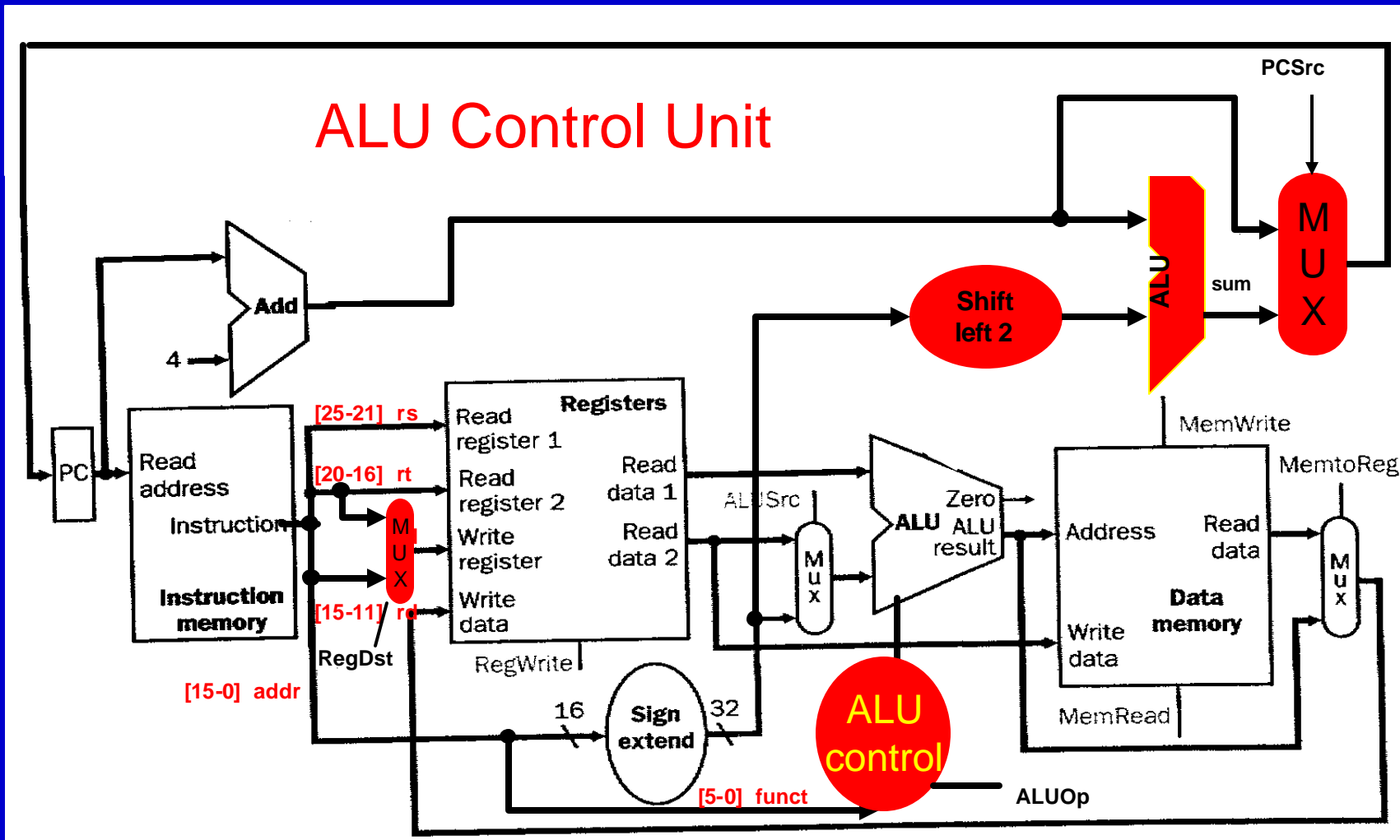


Instruction bits



Op Code Mapping

Instr type	OpCode in decimal	OpCode in binary					
		Instr[31]	Instr[30]	Instr[29]	Instr[28]	Instr[27]	Instr[26]
		Op5	Op4	Op3	Op2	Op1	Op0
R-type	0_{ten}	0	0	0	0	0	0
Load	35_{ten}	1	0	0	0	1	1
Store	43_{ten}	1	0	1	0	1	1
BEQ	4_{ten}	0	0	0	1	0	0
Jump	2_{ten}	0	0	0	0	1	0



What about write register?

For R-type use bits $[15-11]$

For I-type use bits $[20-16]$

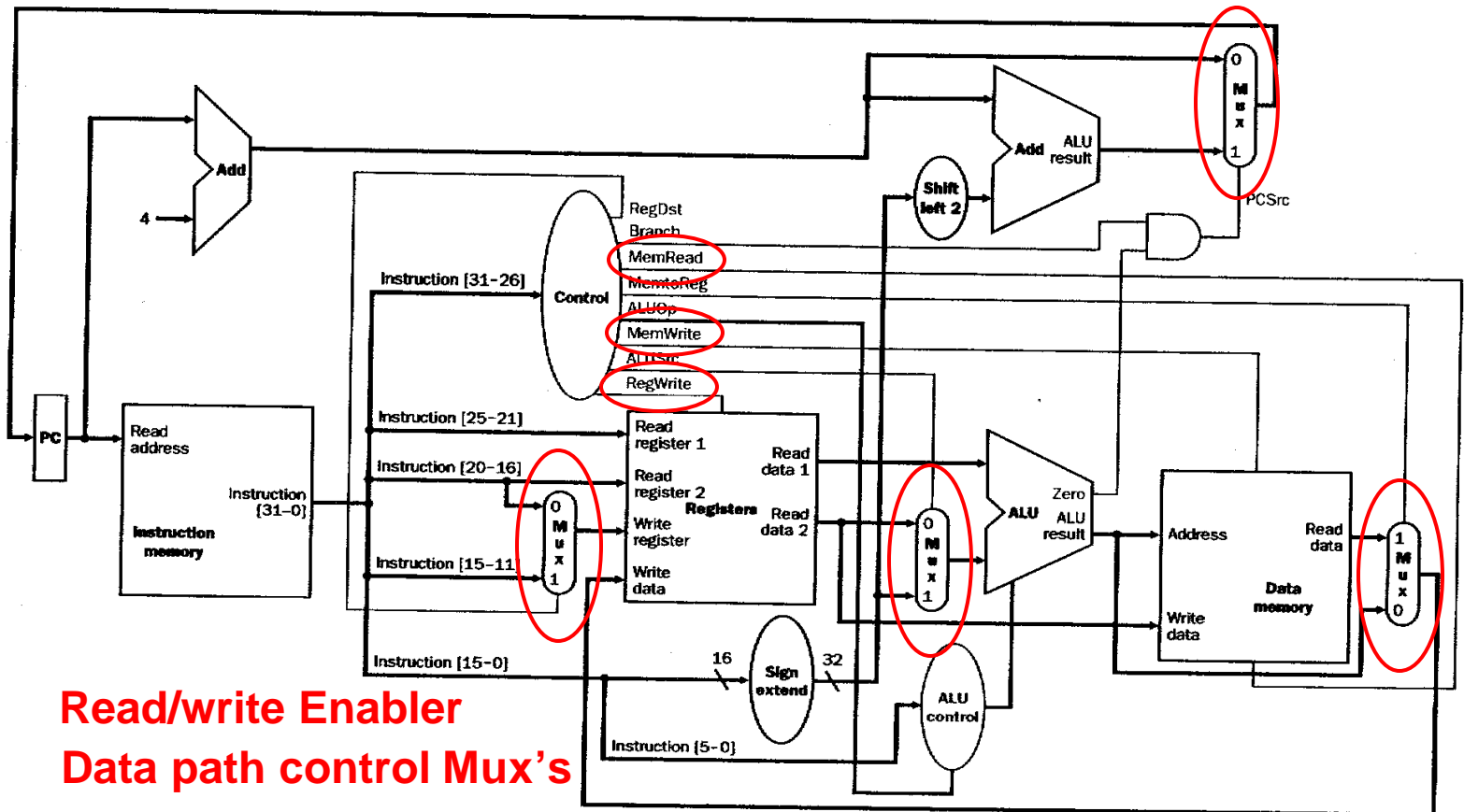
Single-cycle Instruction execution

- R-type
 - Fetch instruction and increment PC
 - Read two source registers from register file (set control)
 - Perform ALU operation on register operands
- Store/Load
 - Fetch instruction and increment PC
 - Read one register value from register file (set control)
 - ALU computes sum of immediate and register value
 - ALU result used as address to memory
 - Data from memory written back to register (Load only)
- Branch
 - Fetch instruction and increment PC
 - Read two source registers from register file (set control)
 - ALU performs subtract on register operands (target addr computed)
 - Zero result from ALU determines PC

Main Control Signals

Signal	Deasserted (0)	Asserted (1)
PCSrc (0/1mux)	PC = PC+4	PC ← branch target
RegDst (0/1mux)	Destination register for write uses [20-16]	Destination register for write uses [15-11]
ALUSrc (0/1mux)	Second ALU operand is data from register file (readdata2)	Second ALU operand is sign-extended lower 16 bits [15-0] of instruction
MemToReg (1/0mux)	Data to be written to local register comes from result of ALU	Data to be written to local register comes from data memory
RegWrite (enable)	No write	Register determined by RegDst is written with value on Write data input
MemRead (enable)	No read	Data at address is fetched and put on Readdata output
MemWrite (enable)	No write	Write data written to location specified by address

Data path and control



Read/write Enabler
Data path control Mux's

Designing Main control logic (R-type)

RegDst: Destination register is determined by [15-11] → 1

ALUSrc: ALU input 2 is from register file → 0

MemtoReg: ALU result written to register → 0

RegWrite: Write to register allowed → 1

MemRead: Read to mem not needed (no effect) → 0

MemWrite: Write to mem not needed (no effect) → 0

Branch: This is not a branch (PCSrc → 0) → 0

ALUOp1: TBD by function code → 1 (sets ALU control)

ALUOp0: TBD by function code → 0 (sets ALU control)

Instr type	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-type	1	0	0	1	0	0	0	1	0

Designing Main control logic (Load)

RegDst: Destination register is determined by [20-16] → 0

ALUSrc: ALU input 2 is from sign extended immediate → 1

MemtoReg: data found at address written to register → 1

RegWrite: Write to register allowed → 1

MemRead: Allow mem to be read → 1

MemWrite: Write to mem not needed (no effect) → 0

Branch: This is not a branch (PCSrc → 0) → 0

ALUOp1: TBD by function code → 0 (sets ALU control)

ALUOp0: TBD by function code → 0 (sets ALU control)

Instr type	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-type	1	0	0	1	0	0	0	1	0
Load	0	1	1	1	1	0	0	0	0

Designing Main control logic (Store)

RegDst: Don't care b/c no storage to regs (RegWrite=0) → X

ALUSrc: ALU input 2 is from sign extended immediate → 1

MemtoReg: Don't care b/c no storage to regs (RegWrite=0) → X

RegWrite: Write to register not allowed → 0

MemRead: Read to mem not needed (no effect) → 0

MemWrite: Allow mem to be written → 1

Branch: This is not a branch (PCSrc → 0) → 0

ALUOp1: TBD by function code → 0 (sets ALU control)

ALUOp0: TBD by function code → 0 (sets ALU control)

Instr type	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-type	1	0	0	1	0	0	0	1	0
Load	0	1	1	1	1	0	0	0	0
Store	X	1	X	0	0	1	0	0	0

Designing Main control logic (BEQ)

RegDst: Don't care b/c no storage to regs (RegWrite=0) → X

ALUSrc: ALU input 2 is from register file → 0

MemtoReg: Don't care b/c no storage to regs (RegWrite=0) → X

RegWrite: Write to register not allowed → 0

MemRead: Read to mem not needed (no effect) → 0

MemWrite: Write to mem not needed (no effect) → 0

Branch: This is a branch (PCSrc=(zero AND Branch)) → 1

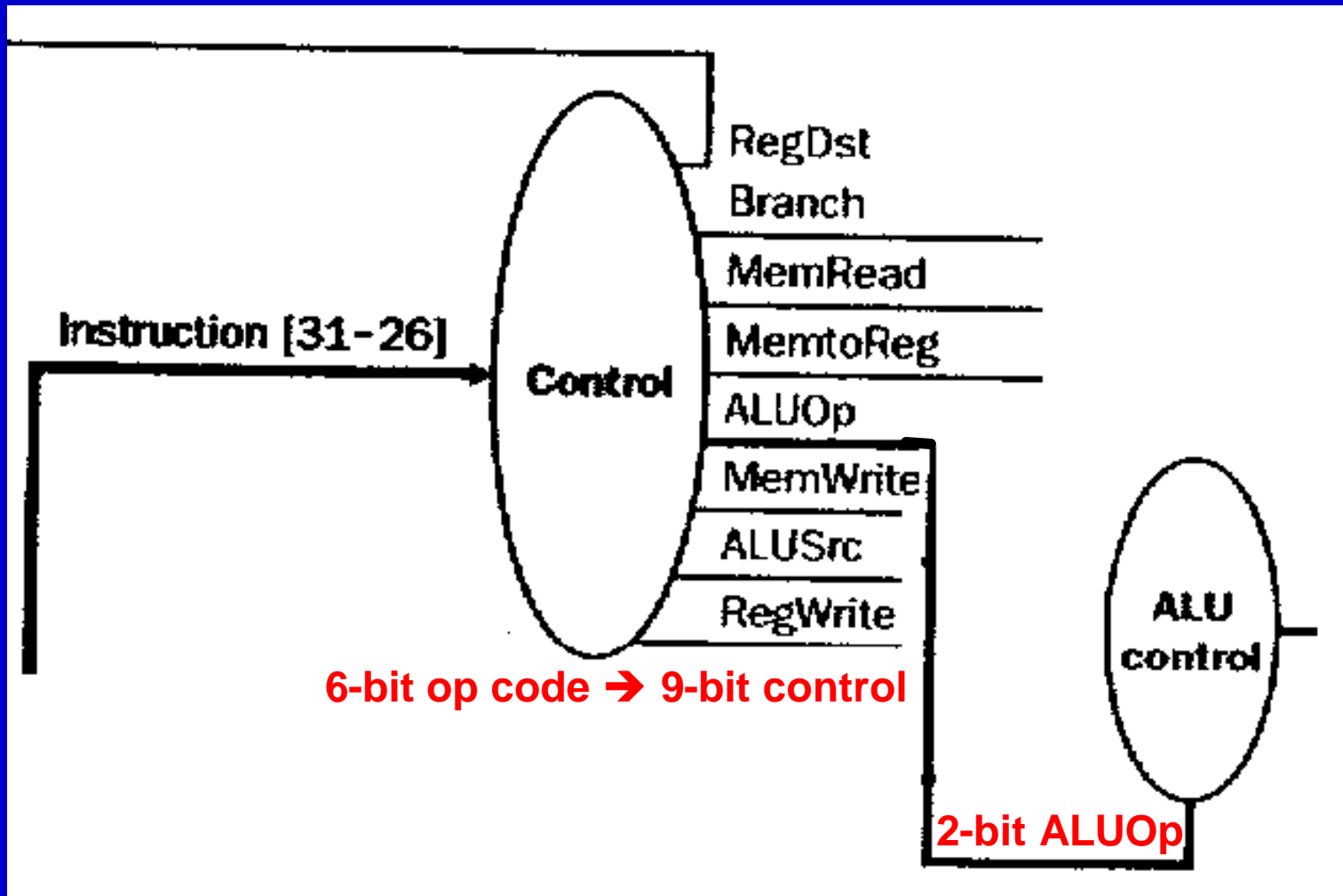
ALUOp1: TBD by function code → 0 (sets ALU control)

ALUOp0: TBD by function code → 1 (sets ALU control)

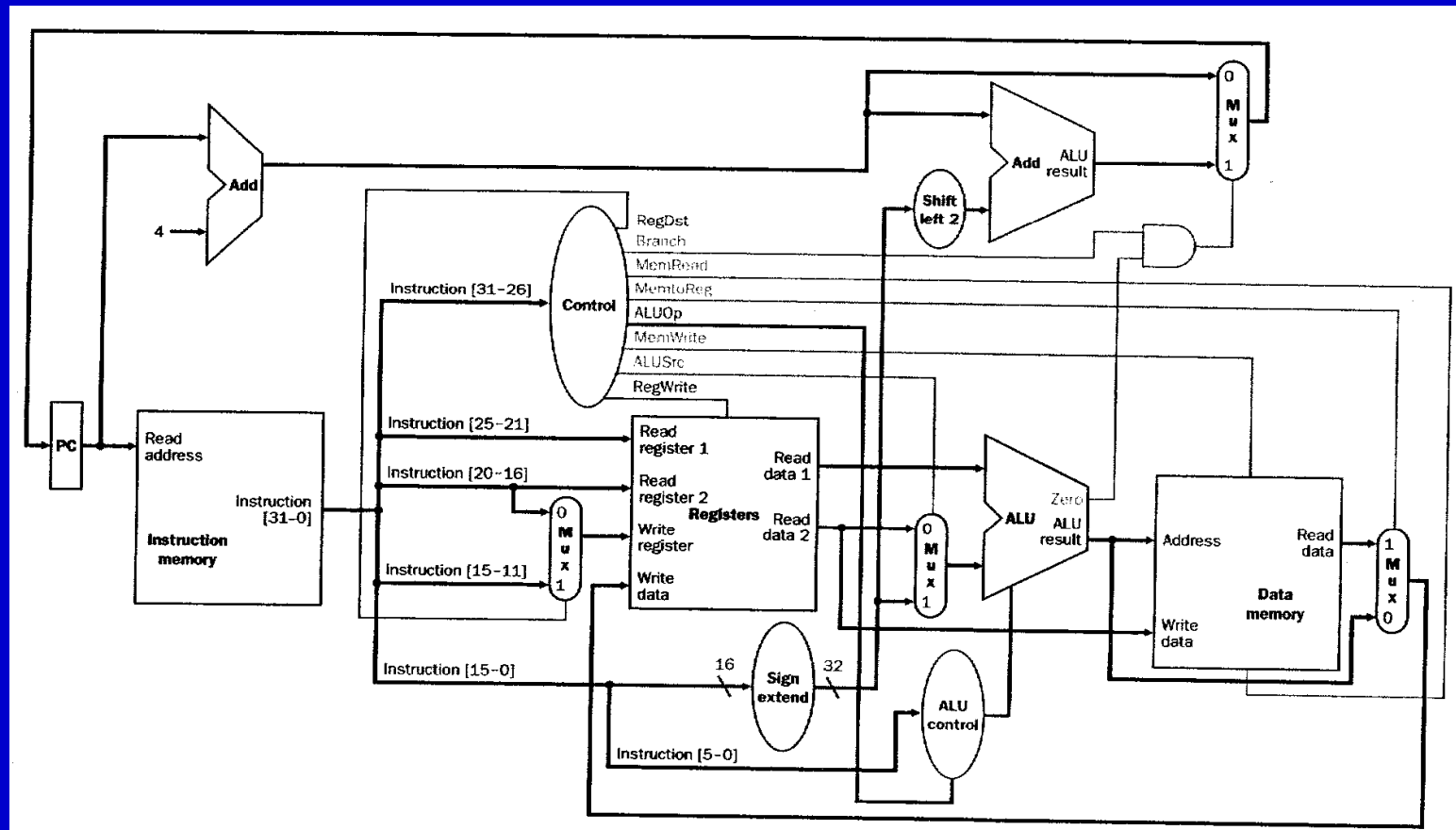
Instr type	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-type	1	0	0	1	0	0	0	1	0
Load	0	1	1	1	1	0	0	0	0
Store	X	1	X	0	0	1	0	0	0
BEQ	X	0	X	0	0	0	1	0	1

What about Jump?

Hierarchical Control Logic

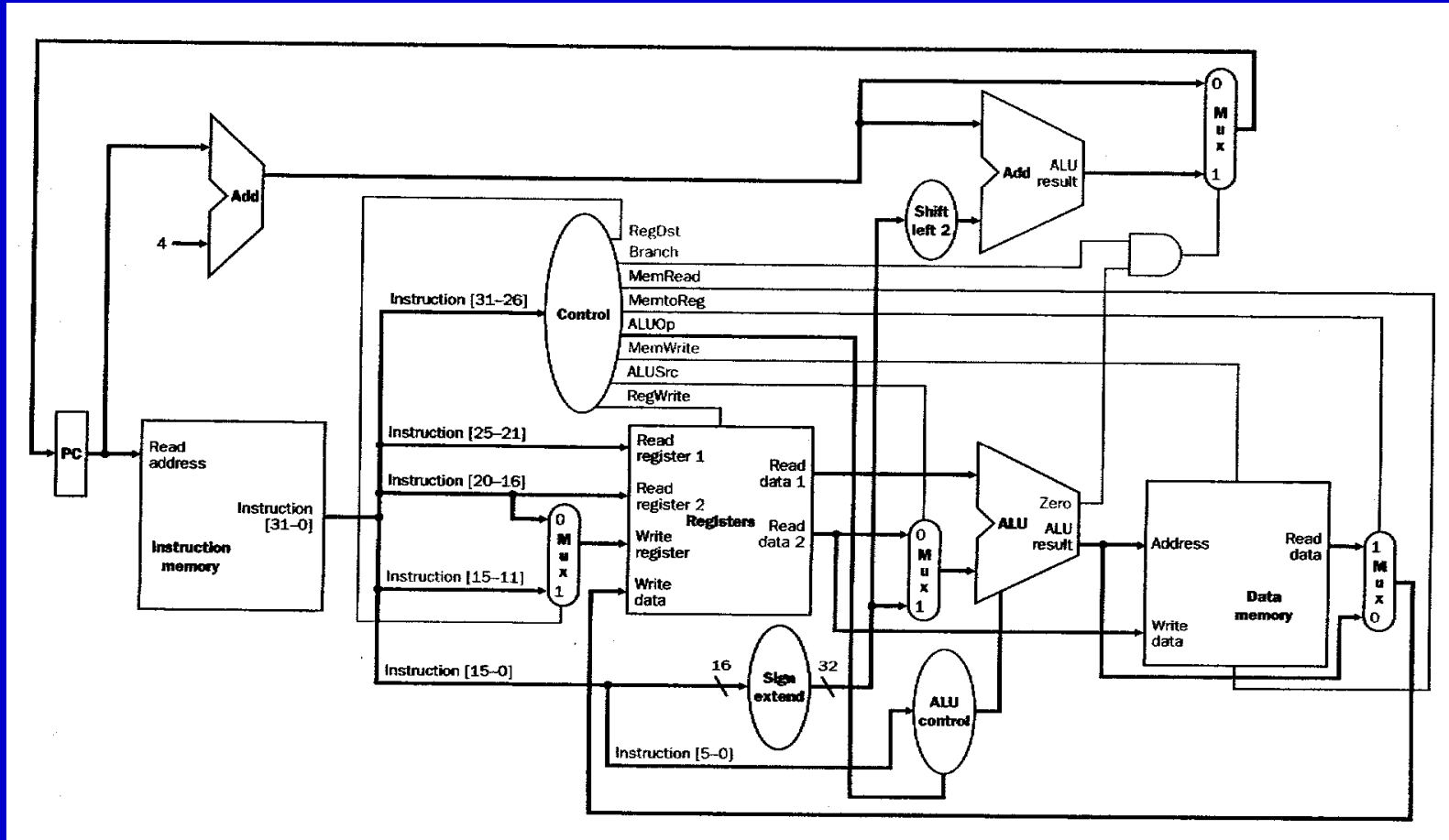


Active Units for R-type



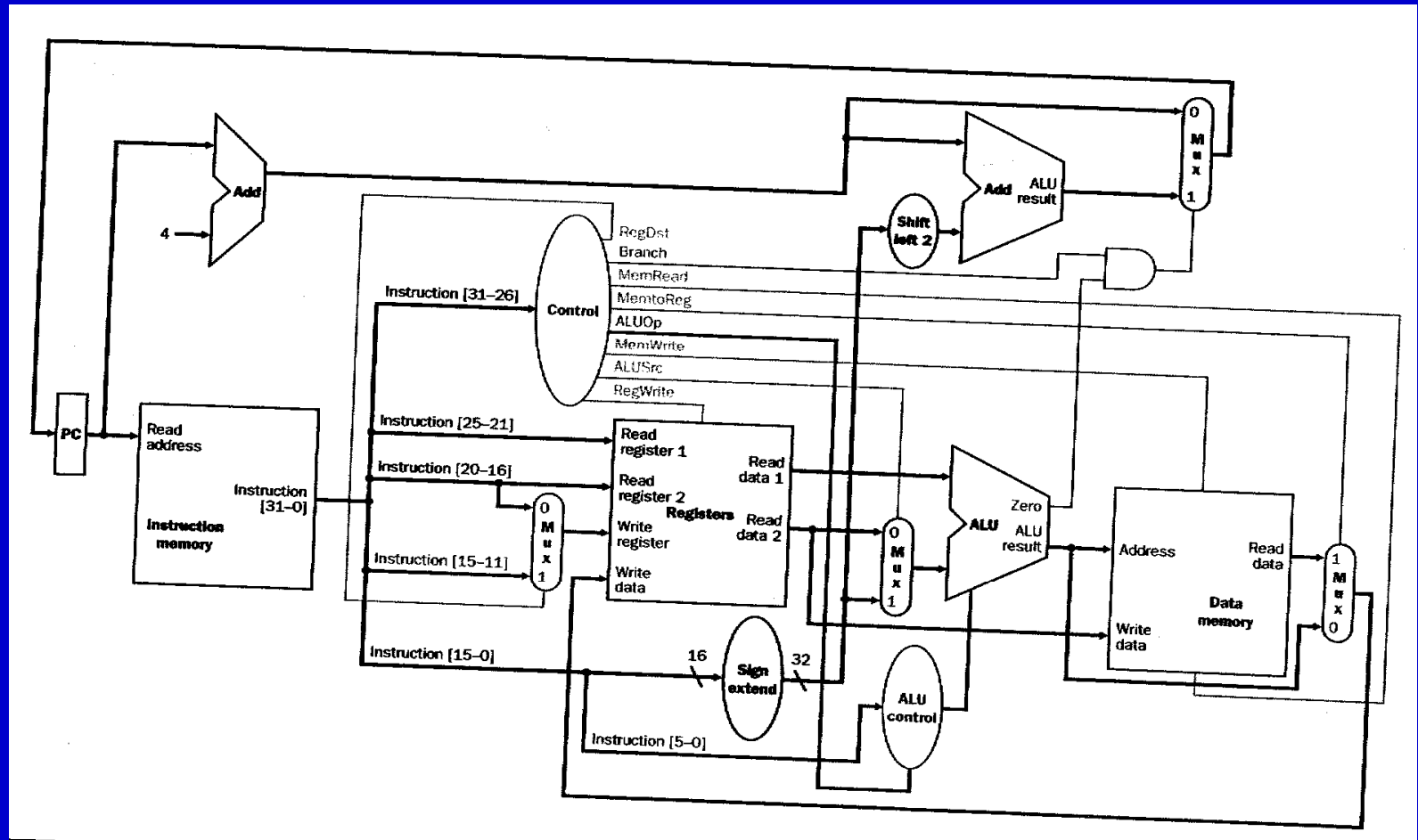
Instr type	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-type	1	0	0	1	0	0	0	1	0

Active Units for Load



Instr type	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
Load	0	1	1	1	1	0	0	0	0

Active Units for Branch



Instr type	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
BEQ	X	0	X	0	0	0	1	0	1

Design of Main Control

Instr type	OpCode in decimal	OpCode in binary					
		Instr [31]	Instr [30]	Instr [29]	Instr [28]	Instr [27]	Instr [26]
		Op5	Op4	Op3	Op2	Op1	Op0
R-type	0_{ten}	0	0	0	0	0	0
Load	35_{ten}	1	0	0	0	1	1
Store	43_{ten}	1	0	1	0	1	1
BEQ	4_{ten}	0	0	0	1	0	0

Instr type	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-type	1	0	0	1	0	0	0	1	0
Load	0	1	1	1	1	0	0	0	0
Store	X	1	X	0	0	1	0	0	0
BEQ	X	0	X	0	0	0	1	0	1

Resulting Truth Table

	Signal	R-type	Load	Store	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

Jumps

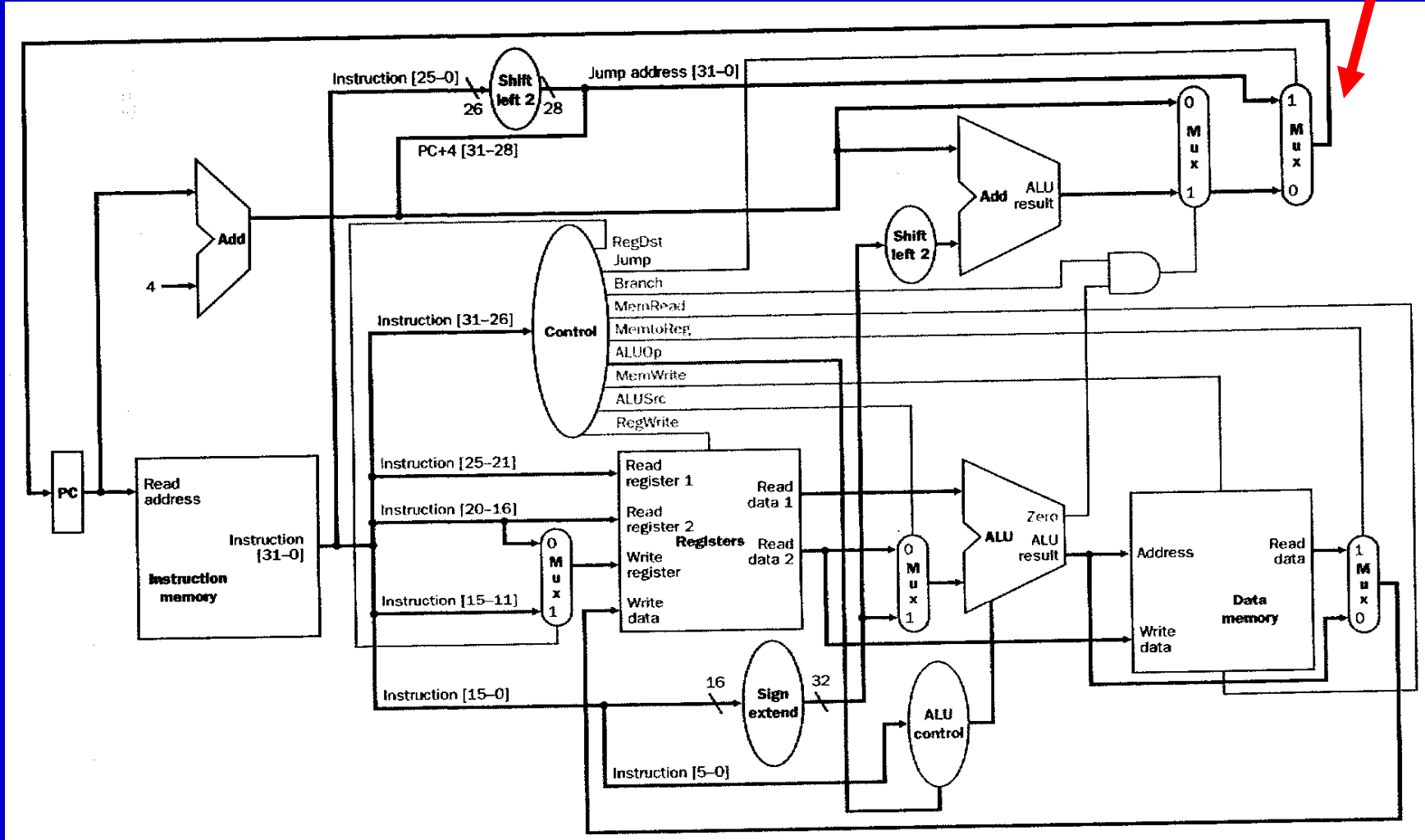
- Unconditional change to PC
- 26-bit immediate is word-aligned
- PC-relative offset, thus upper bits of PC give upper bits of target address



- Resulting 32-bit target address..



Jumps



Single-cycle performance

- Assume
 - No delay from mux's, control unit, PC access, sign extend, wiring
 - Delays: memory, 2ns; ALU 2ns; Registerfile 1ns
 - 24% loads, 12% stores, 44% R-type, 18% branch, 2% jumps
- Compare
 - 1 fixed-length clock cycle implementation
 - 1 variable-length clock cycle implementation (impractical)

Instruction type	Instruction memory	Register Read	ALU operation	Data memory	Register Write	Total
R-type	2	1	2	0	1	6
Load	2	1	2	2	1	8
Store	2	1	2	2		7
Branch	2	1	2			5
Jump	2					2

Single-cycle performance

Fixed length clock cycle: 8 ns clock length
avg time / instr = 8 ns

Variable length clock cycle:
avg time / instr = $8(.24)+7(.12)+6(.44)+5(.18)+2(.02)= 6.3$ ns

Speedup: $8/6.3=1.27$

Variable clock implementation is 1.27 times faster than fixed.

How can we gain efficiency without added complexity of variable clock length?

Drawbacks to single-cycle implementation

- Clock cycle length depends on critical path (loads)
- CPI=1 is good provided clock rate very fast
- Problem 1: many instructions execute faster than loads, thus inefficient (worse for fp)
 - Solution 1: use shorter clock cycle and require multiple clocks per instruction
- Problem 2: each functional unit can be used only once per clock
 - Solution 2: reuse functional units in data path through multi-cycle implementation