Principles of Computer Architecture

CSE 240A

Fall 2024

Hadi Esmaeilzadeh

hadi@ucsd.edu

University of California, San Diego

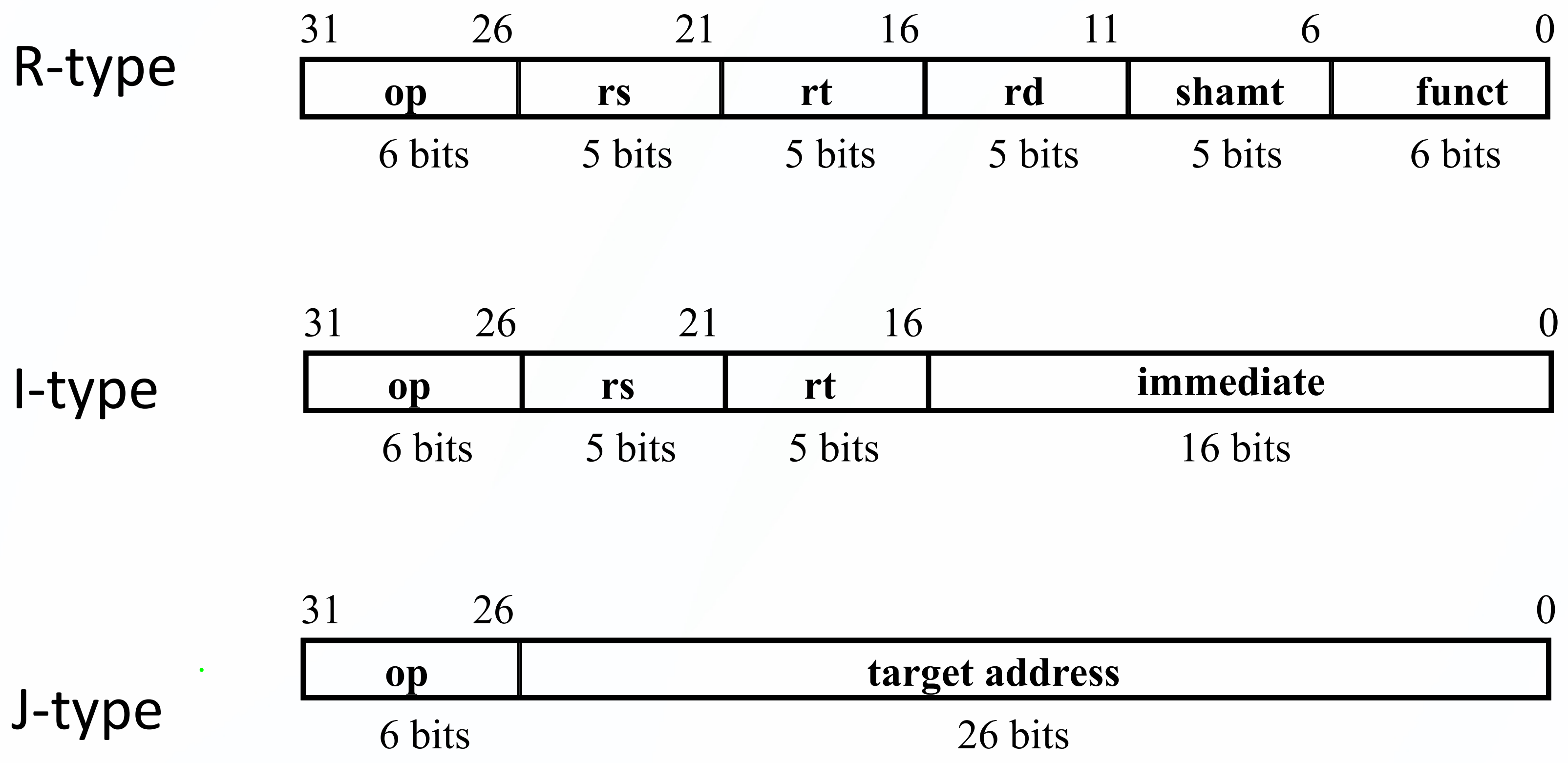
Processor Cheat Sheet

The Processor: Datapath & Control

- We're ready to look at an implementation of MIPS simplified to contain only:
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - arithmetic-logical instructions with immediates: `addi, subi, andi, ori, slt`
  - memory-reference instructions: `lw, sw`
  - control flow instructions: `beq, jr, j, jal`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- Two sets of registers
  - Architecturally visible: PC, Reg File
  - Architecturally invisible: Multi-cycle physical registers, Pipe registers

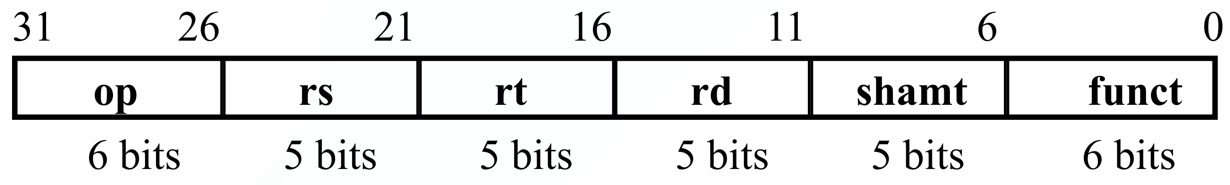
The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:

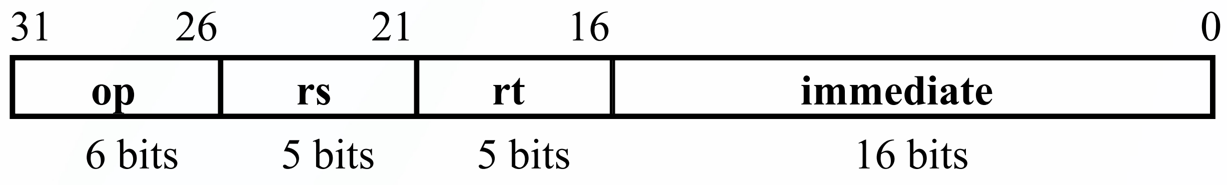


The MIPS Subset

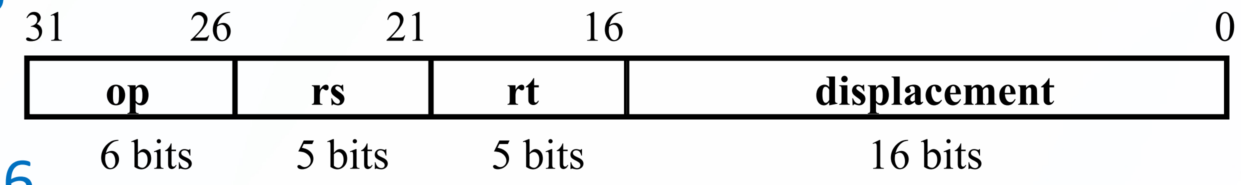
- R-type
  - *add rd, rs, rt*
  - sub, and, or, slt
  - jr rs => Function Call / Function Call Return: jr $ra = jr $r31



- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16



- BRANCH:
  - beq rs, rt, imm16



- JUMP
  - j label
  - jal label => Function call

Category	Instr	Op Code	Example	Meaning
Arithmetic (R format)	**add**	**0 and 32**	**add $s1, $s2, $s3**	**$s1 = $s2 + $s3**
	subtract	**0 and 34**	**sub $s1, $s2, $s3**	**$s1 = $s2 - $s3**
Data transfer (I format)	**load word**	**35**	**lw $s1, 100($s2)**	**$s1 = Memory($s2+100)**
	store word	**43**	**sw $s1, 100($s2)**	**Memory($s2+100) = $s1**
	load byte	**32**	**lb $s1, 101($s2)**	**$s1 = Memory($s2+101)**
	store byte	**40**	**sb $s1, 101($s2)**	**Memory($s2+101) = $s1**
Cond. Branch	**br on equal**	**4**	**beq $s1, $s2, L**	**if ($s1==$s2) go to L**
	br on not equal	**5**	**bne $s1, $s2, L**	**if ($s1 !=$s2) go to L**
	set on less than	**0 and 42**	**slt $s1, $s2, $s3**	**if ($s2<$s3) $s1=1 else $s1=0**
Uncond. Jump	**jump**	**2**	**j 2500**	**go to 10000**
	jump register	**0 and 8**	**jr $t1**	**go to $t1**
	jump and link	**3**	**jal 2500**	**go to 10000; $ra=PC+4**

$ra is the Return Address register, which is R31

Namesapce Data Transfer Language

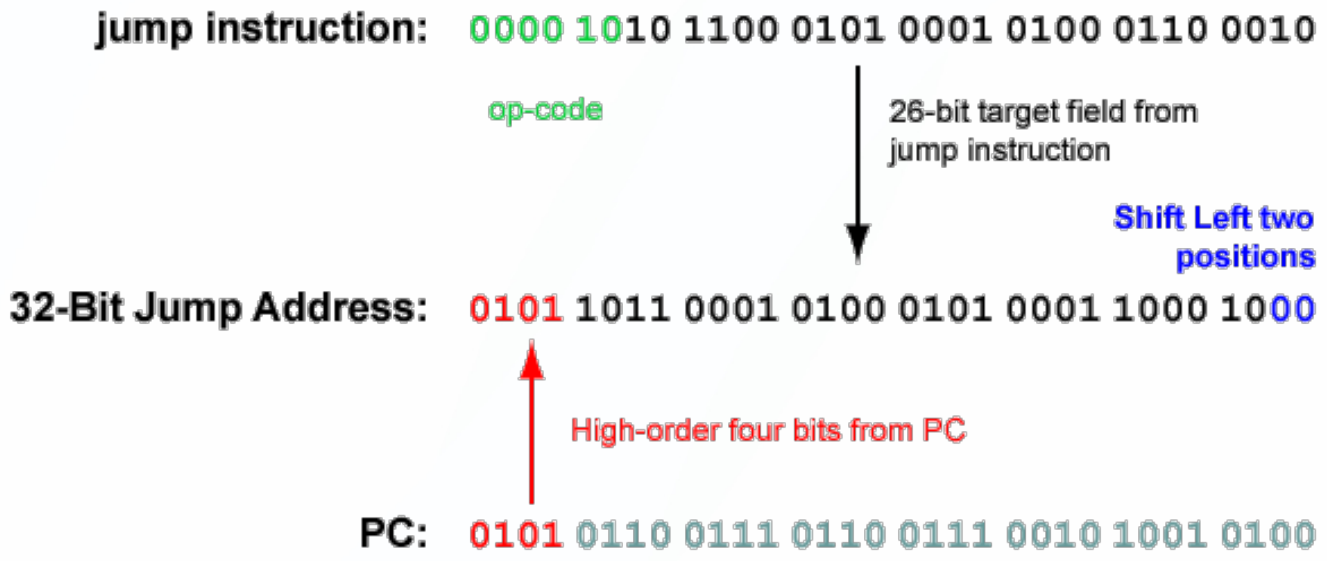
- is a mechanism for describing the movement and manipulation of data between storage elements:

```
R[3] <- R[5] + R[7]
PC <- PC + 4 + R[5]
R[rd] <- R[rs] + R[rt]
R[rt] <- Mem[R[rs] + immed]
```

First Step: Namespace Data Transfer Language Implementation (Single Cycle)

- arithmetic-logical instructions:
  `add, sub, and, or, slt`
  - `PC <- PC + 4`
  - `RegFile[rd] <- RegFile[rs] + RegFile[rt]`
- arithmetic-logical instructions with immediates:
  `addi, subi, andi, ori, slt`
  - `PC <- PC + 4`
  - `RegFile[rt] <- RegFile[rs] + sign_ext(imm)`
- memory-reference instructions: `lw, sw`
  - `PC <- PC + 4`
  - `RegFile[rt] <- Mem[rs + sign_ext(imm)]`
  - `Mem[rs + sign_ext(imm)] <= RegFile[rt]`
- control flow instructions: `beq/bne`
  - `if (rs ==/!= rt) {PC <- PC + sign_ext(imm << 2)} else {PC <- PC + 4}`
- jump instructions:
  - `jr : PC <- RegFile[rs]`
  - `j: PC <- {PC[31:28], target_address << 2}`

  `jal`
  - `RegFile[31] <- PC + 4 // $r31 = $ra = return address register`
  - `PC <- {PC[31:28], target_address << 2}`


jump instruction: 0000 1010 1100 0101 0001 0100 0110 0010
op-code
26-bit target field from jump instruction
Shift Left two positions
32-Bit Jump Address: 0101 1011 0001 0100 0101 0001 1000 1000
High-order four bits from PC
PC: 0101 0110 0111 0110 0111 0010 1001 0100

MIPS operands

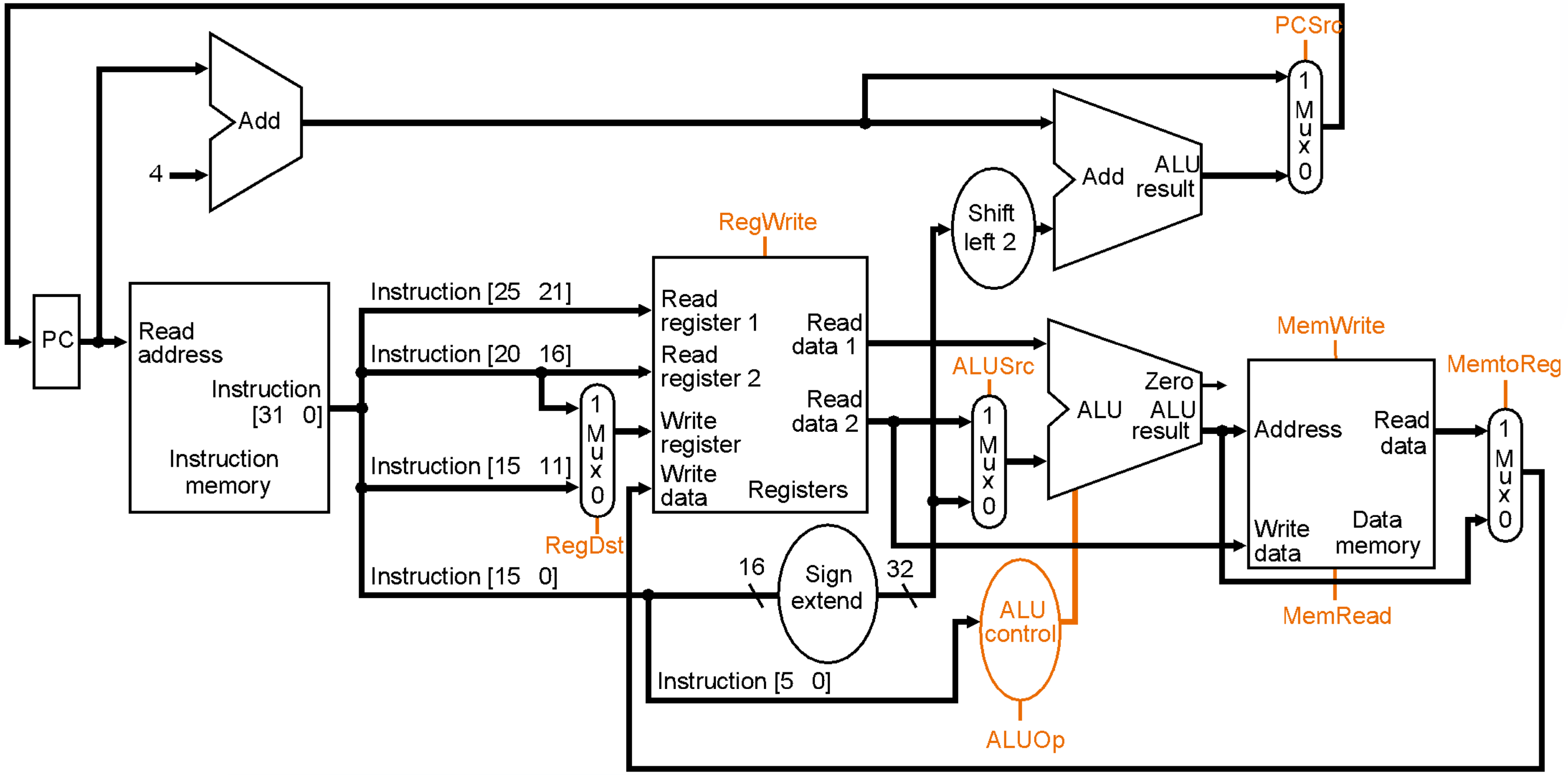
Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

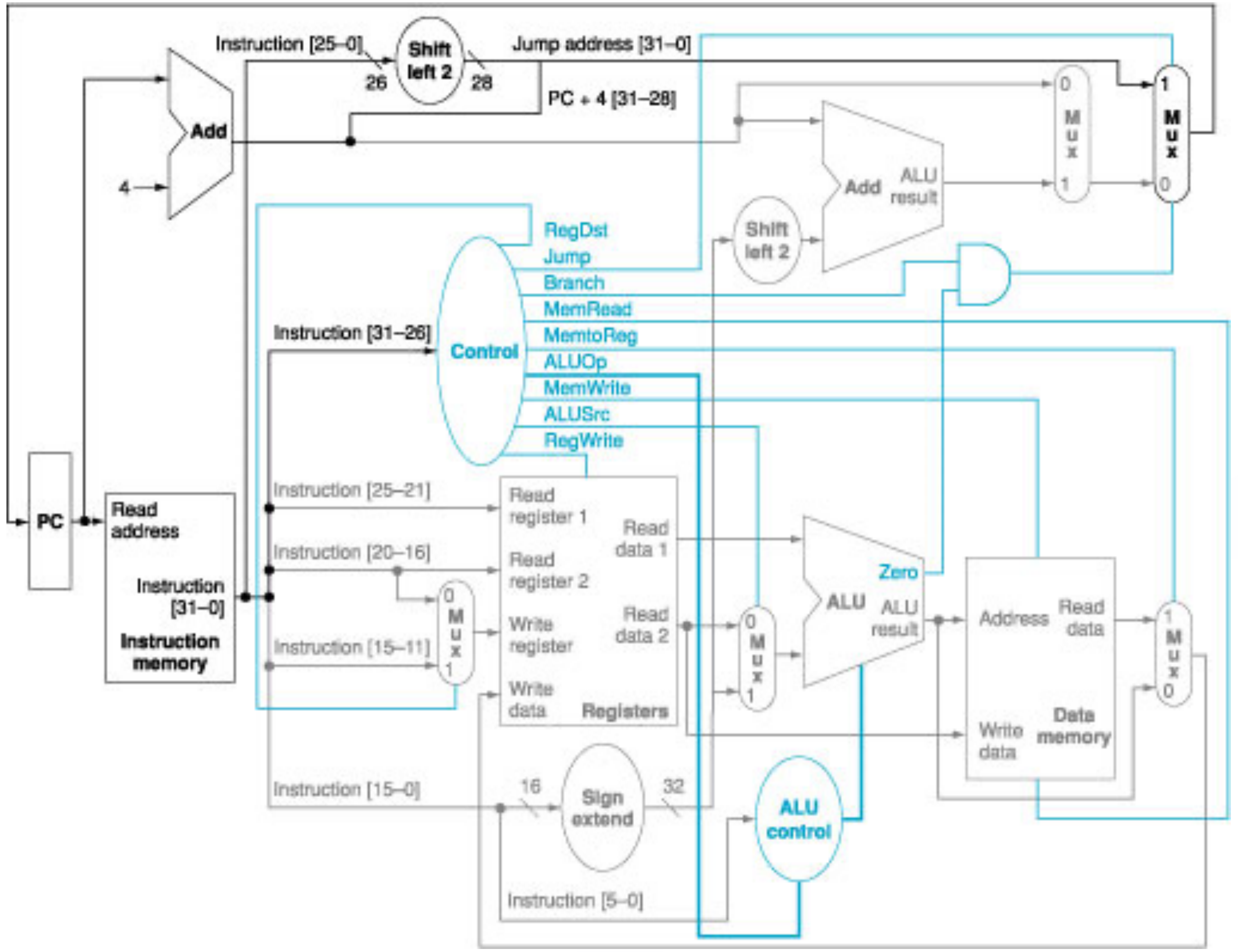
MIPS assembly language

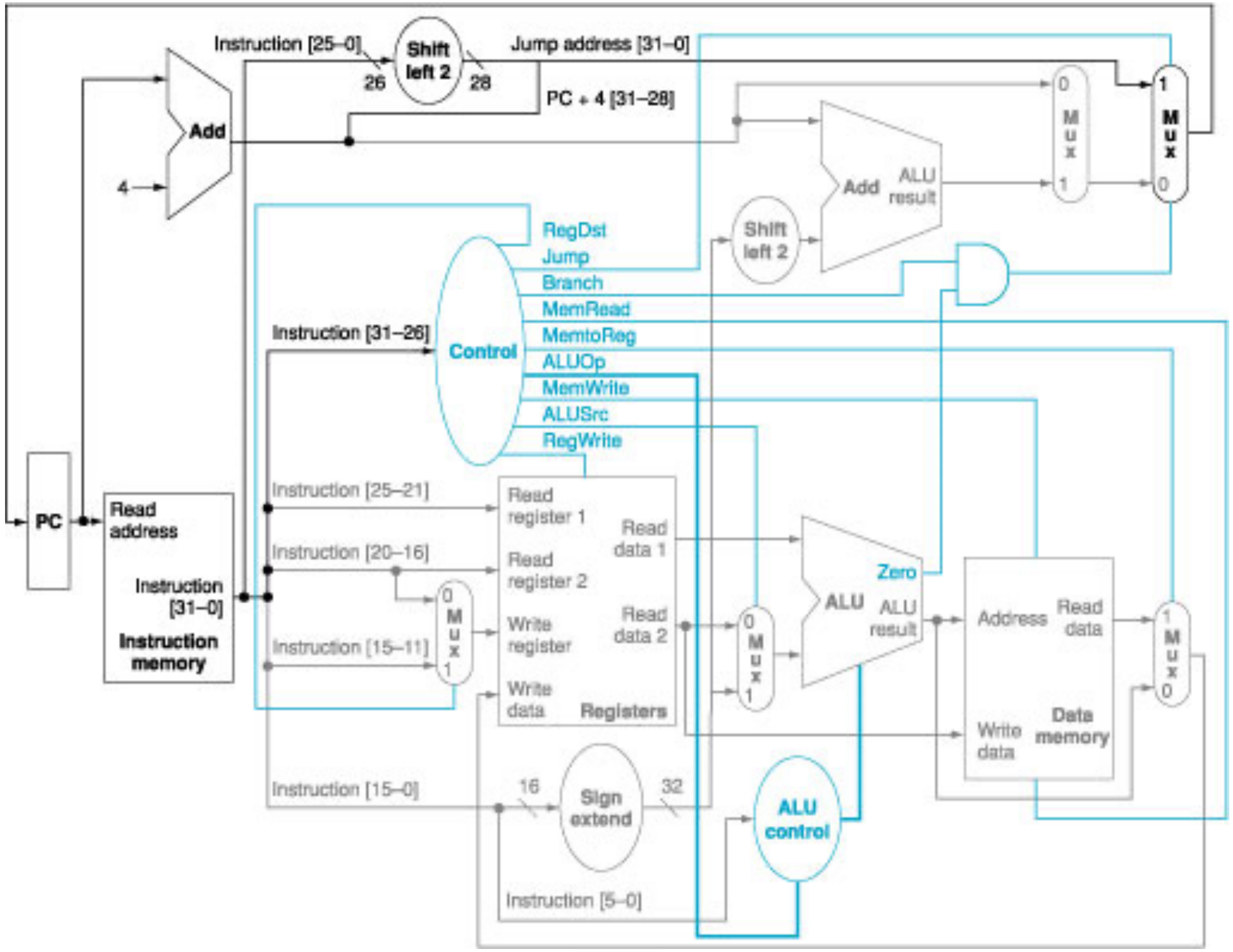
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	lb $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1, 100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equa	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Putting it All Together: A Single Cycle Datapath

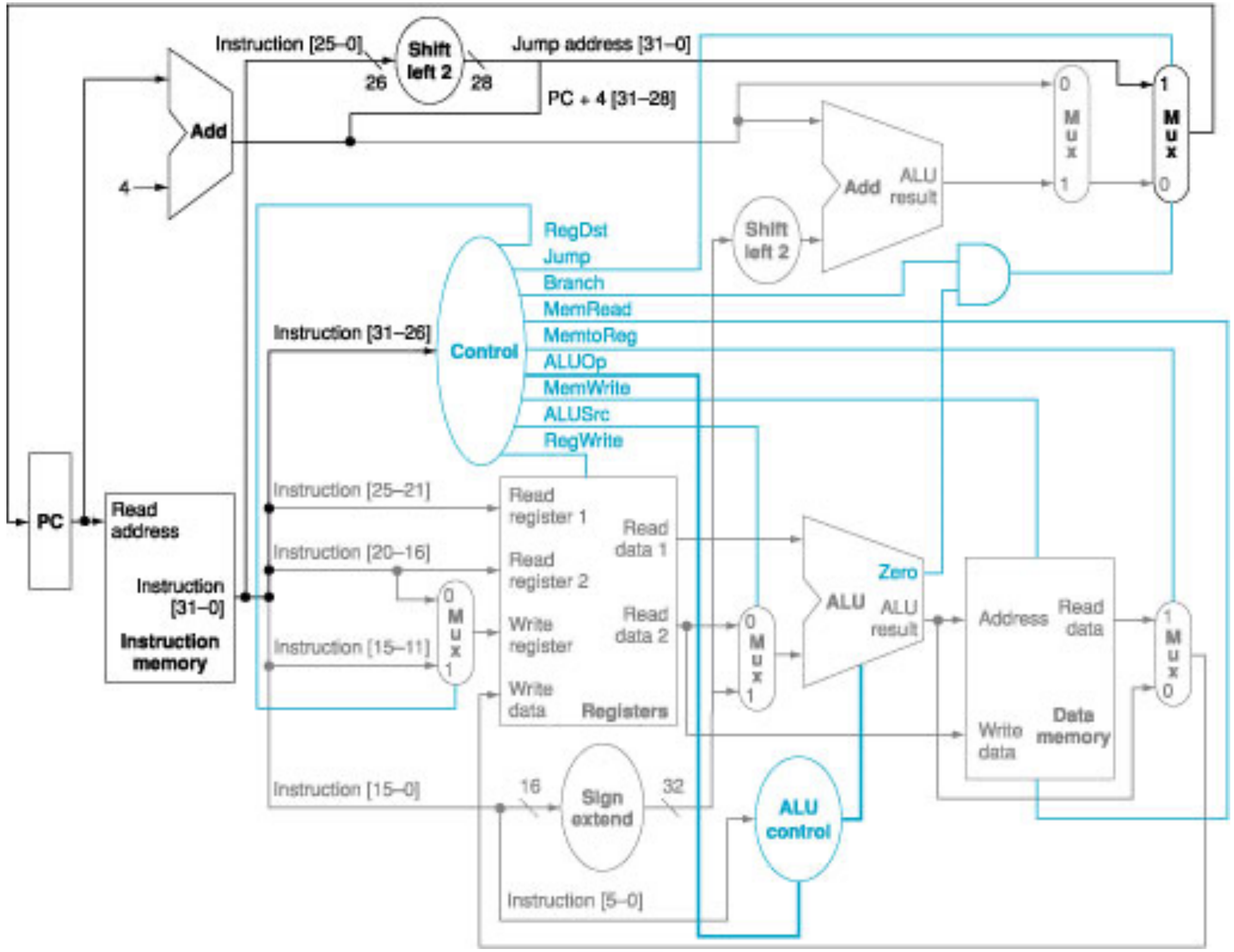
- We have everything except control signals

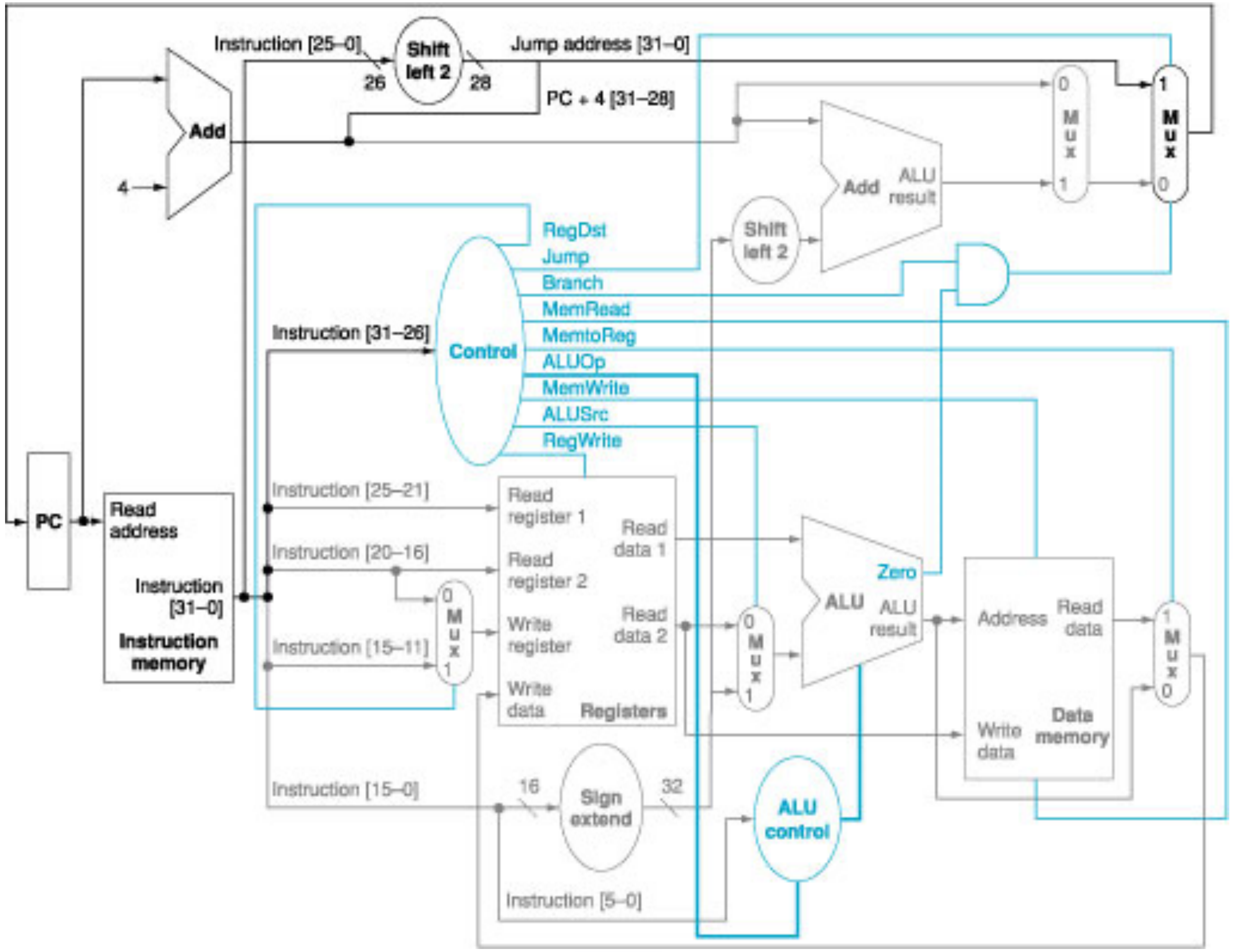


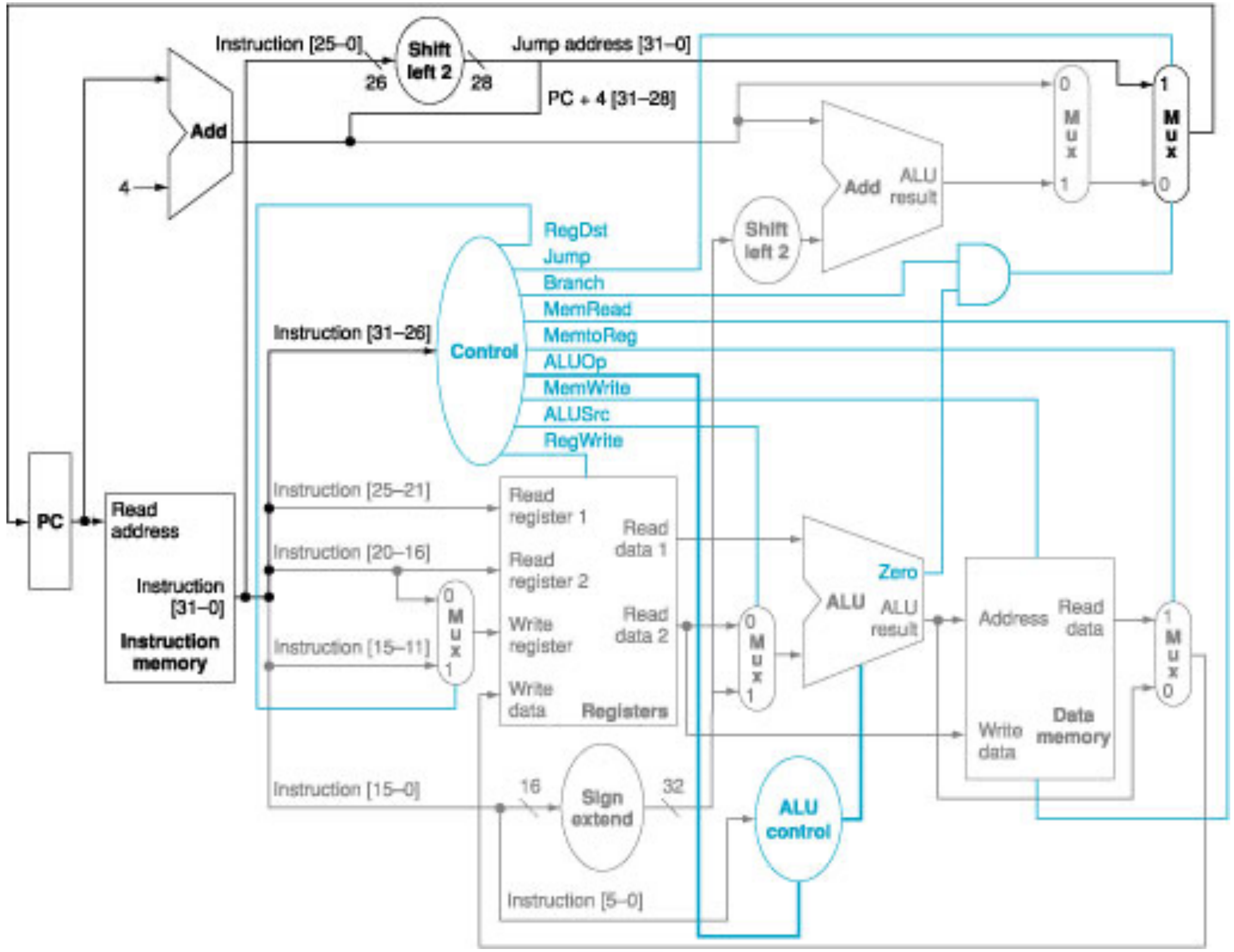

Instruction [25–0]
26
Shift left 2
28
Jump address [31–0]
PC + 4 [31–28]
Add
4
0
M u x
1
1
M u x
0
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction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



Instruction [25–0]
26
Shift left 2
28
Jump address [31–0]
PC + 4 [31–28]
Add
4
Add
ALU result
Shift left 2
Mux
0
1
Mux
1
0
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]
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
Instruction [15–0]
16
Sign extend
32
ALU control
Instruction [5–0]


Instruction [25–0]
26
Shift left 2
28
Jump address [31–0]
PC + 4 [31–28]
Add
4
0
M u x
1
1
M u x
0
Add
ALU result
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]
0
M u x
1
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
0
M u x
1
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
1
M u x
0
Instruction [15–0]
16
Sign extend
32
ALU control
Instruction [5–0]


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



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

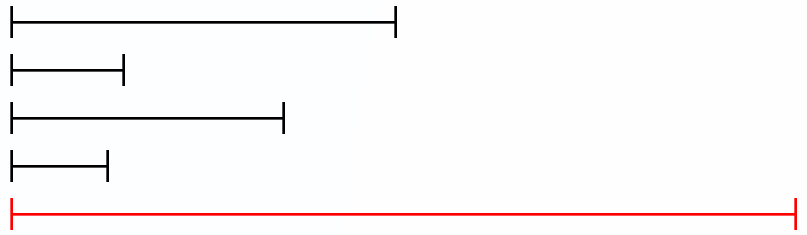
Single Cycle

	CPI	CT
Single Cycle		

Why a Multiple Clock Cycle CPU?

- The problem => single-cycle cpu has a cycle time long enough to complete the longest instruction in the machine
- The solution => break up execution into smaller tasks, each task taking a cycle, different instructions requiring different numbers of cycles or tasks
- Other advantages => reuse of functional units (e.g., ALU, memory)

- ET = IC * CPI * CT



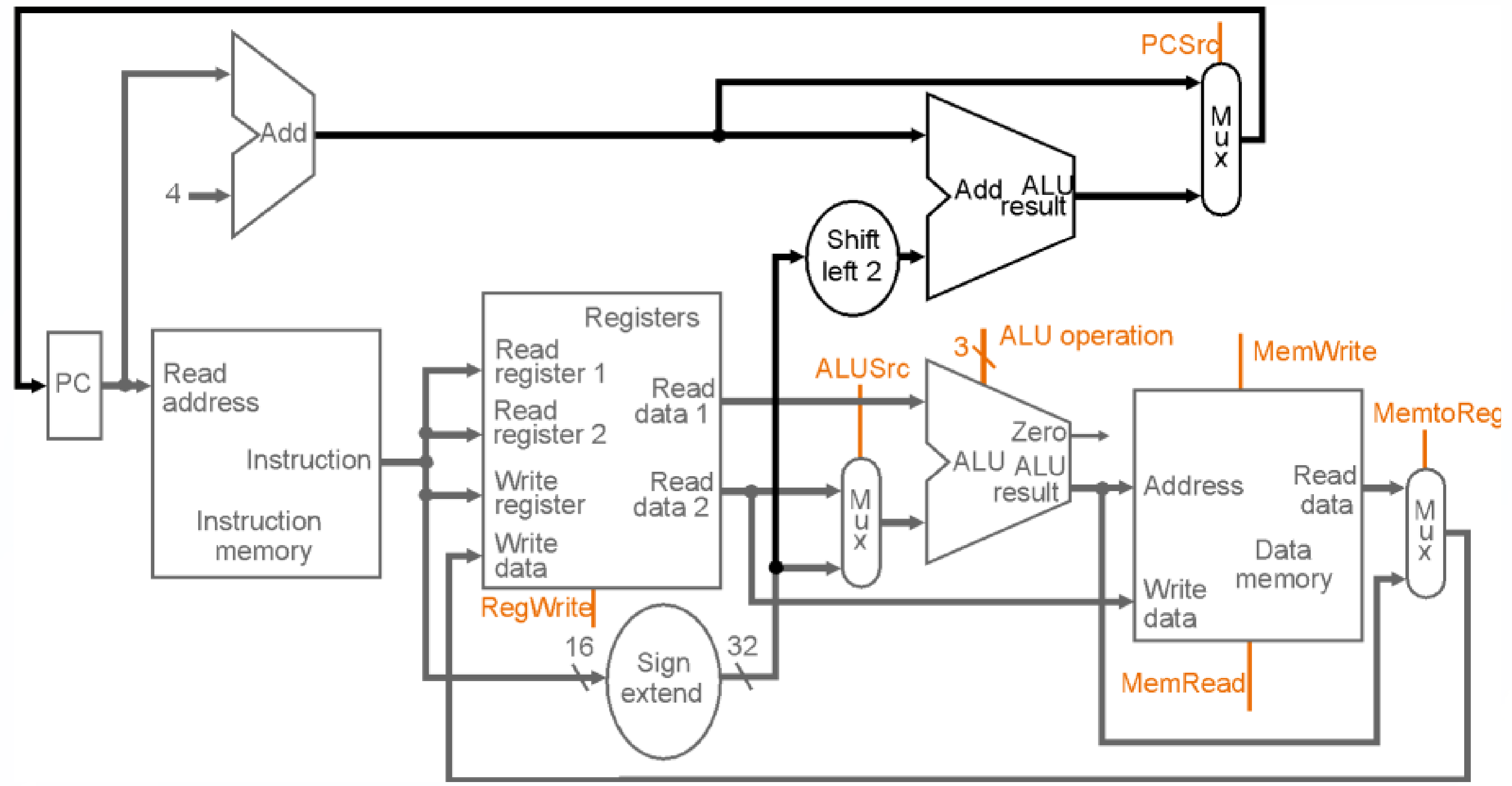
Breaking Execution Into Clock Cycles

- Introduces extra registers when:
  - signal is computed in one clock cycle and used in another, AND
  - the inputs to the functional block that outputs this signal can change before the signal is written into a state element.
- The goal is to balance the amount of work done each cycle.
- Significantly complicates control. **Why?**

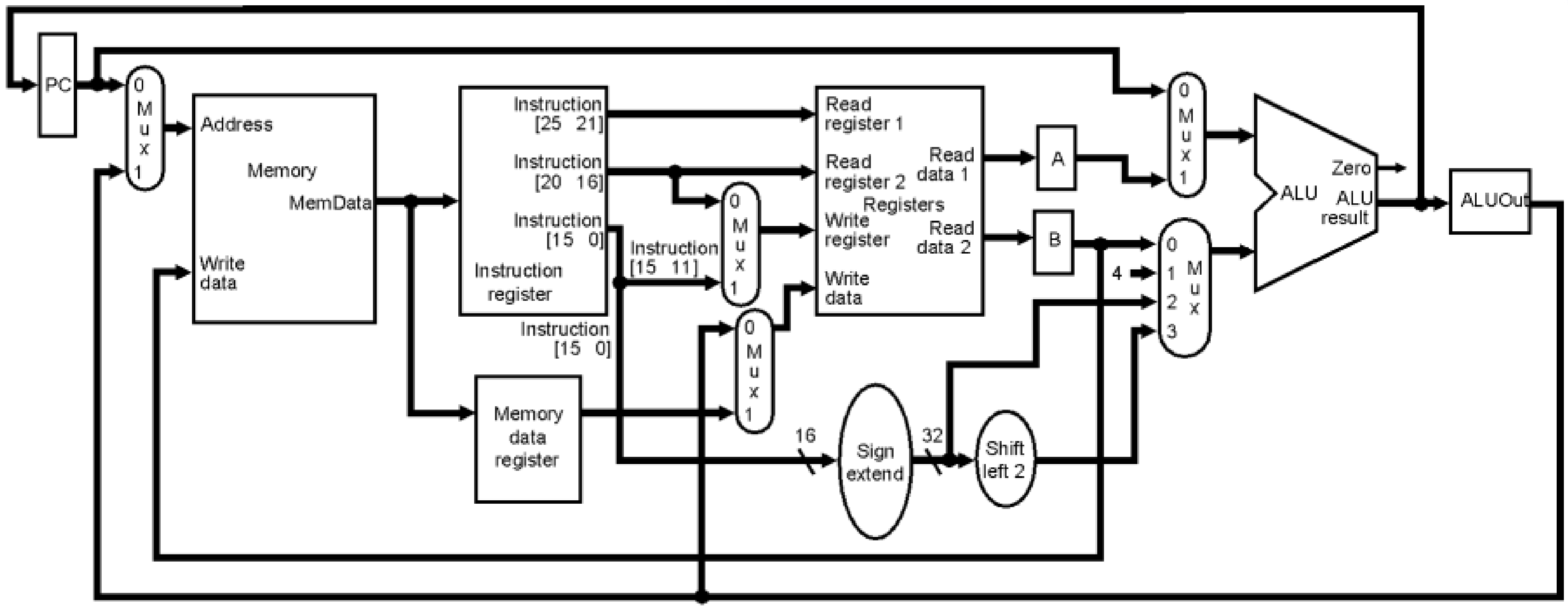
Multi-cycle Review: Breaking Execution Into Clock Cycles

- We will have five execution steps (not all instructions use all five)
  - fetch
  - decode & register fetch
  - execute
  - memory access
  - write-back

Cutting up Single Cycle

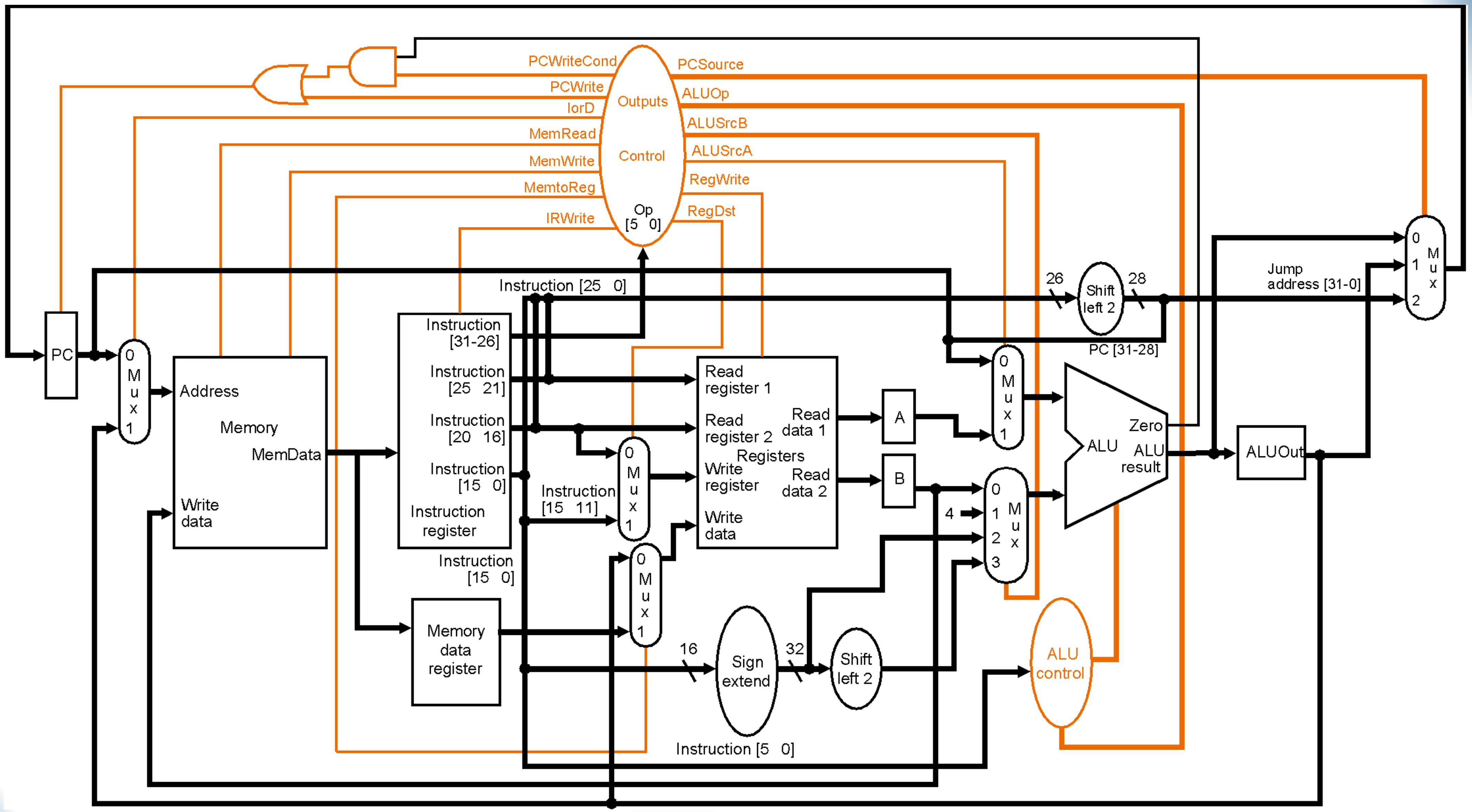


Multicycle datapath



Intermediate latches.
One ALU
One memory (can we do self-modifying code?)

Complete Multicycle Datapath



Summary of execution steps

We can use Namespace Data Transfer Language to describe these steps

Step	R-type	Memory	Branch

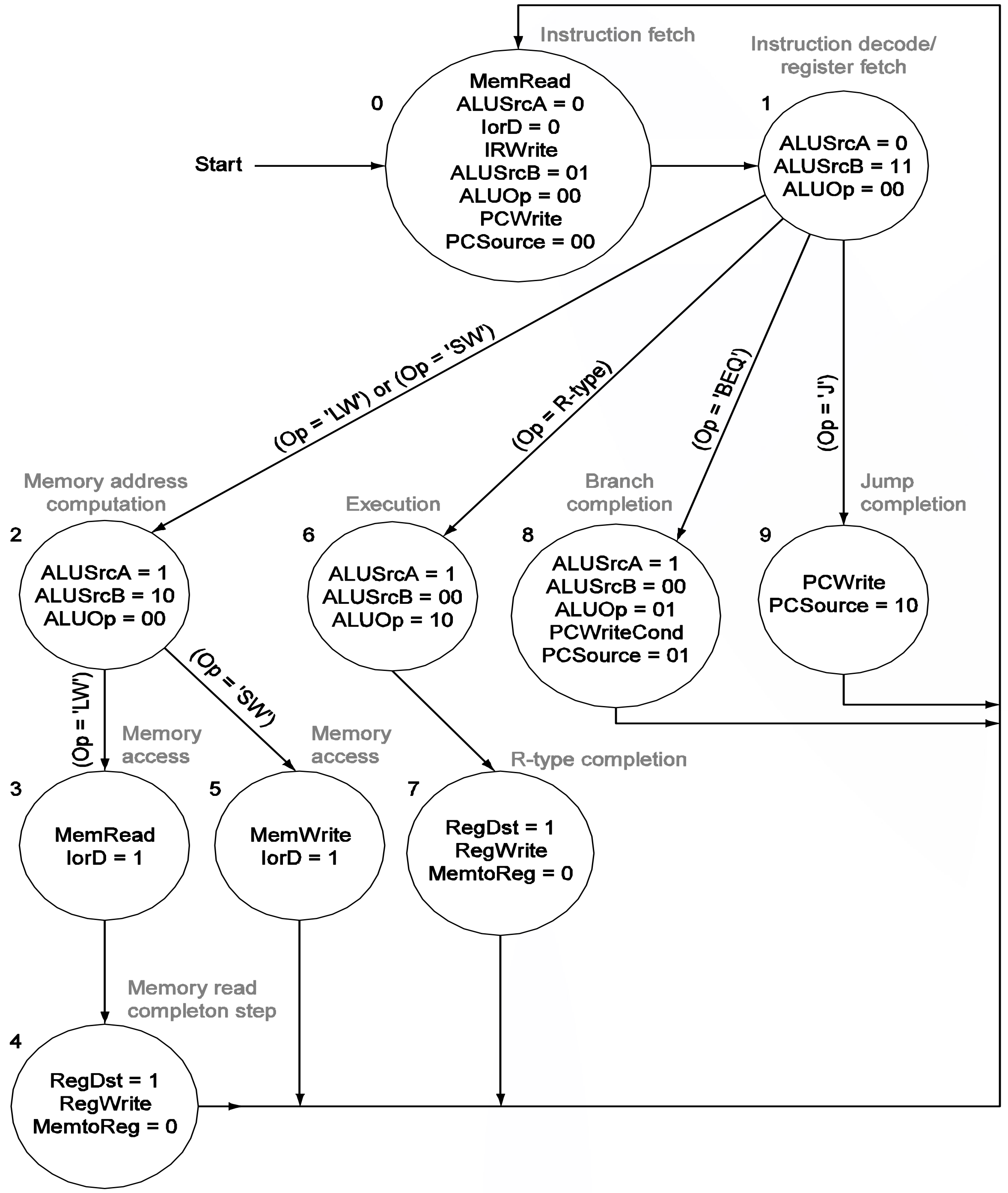
Instruction Fetch	IR = Mem[PC]
	PC = PC + 4		

Instruction Decode/ register fetch	A = Reg[IR[25-21]]
	B = Reg[IR[20-16]]
	ALUout = PC + (sign-extend(IR[15-0]) << 2)		

Execution, address computation, branch completion	ALUout = A op B	ALUout = A + sign-extend(IR[15-0])	if (A==B) then PC=ALUout

Memory access or R-type completion	Reg[IR[15-11]] = ALUout	memory-data = Mem[ALUout]
		or
		Mem[ALUout]=B	

Write-back		Reg[IR[20-16]] = memory-data	


Instruction fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
Instruction decode/
register fetch
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Memory read
completon step
4
RegDst = 1
RegWrite
MemtoReg = 0

Single Cycle vs. Multi-cycle

		CPI	CT
Single Cycle			
Multi-cycle			
lw			
sw			
beq			
r-type			

IF = 200ps

ID = 200ps

EX = 200ps

M = 200ps

WB = 200ps

A coworker at MIPS told you she thinks she can improve the MIPS single cycle processor performance. She tells you that she wants to remove base+displacement memory addressing and replace it with register direct addressing since only 10% of instructions are memory instructions which use a non-zero displacement.

Your coworker asks you whether or not this would be a good idea. You say?
(Hint. Think about implications to the ISA as well as hardware)

Selection	Good idea?	Reason
A	Yes	IC stays the same. CPI stays the same. CT decreases (20%)
B	Yes	IC increases by ~10%. CPI stays the same. CT decreases (20%)
C	No	IC stays the same. CPI stays the same. CT increases (20%)
D	No	IC increases by ~10%. CPI stays the same. CT stays the same.
E	No	IC stays the same. CPI stays the same. CT stays the same. Complexity increases.

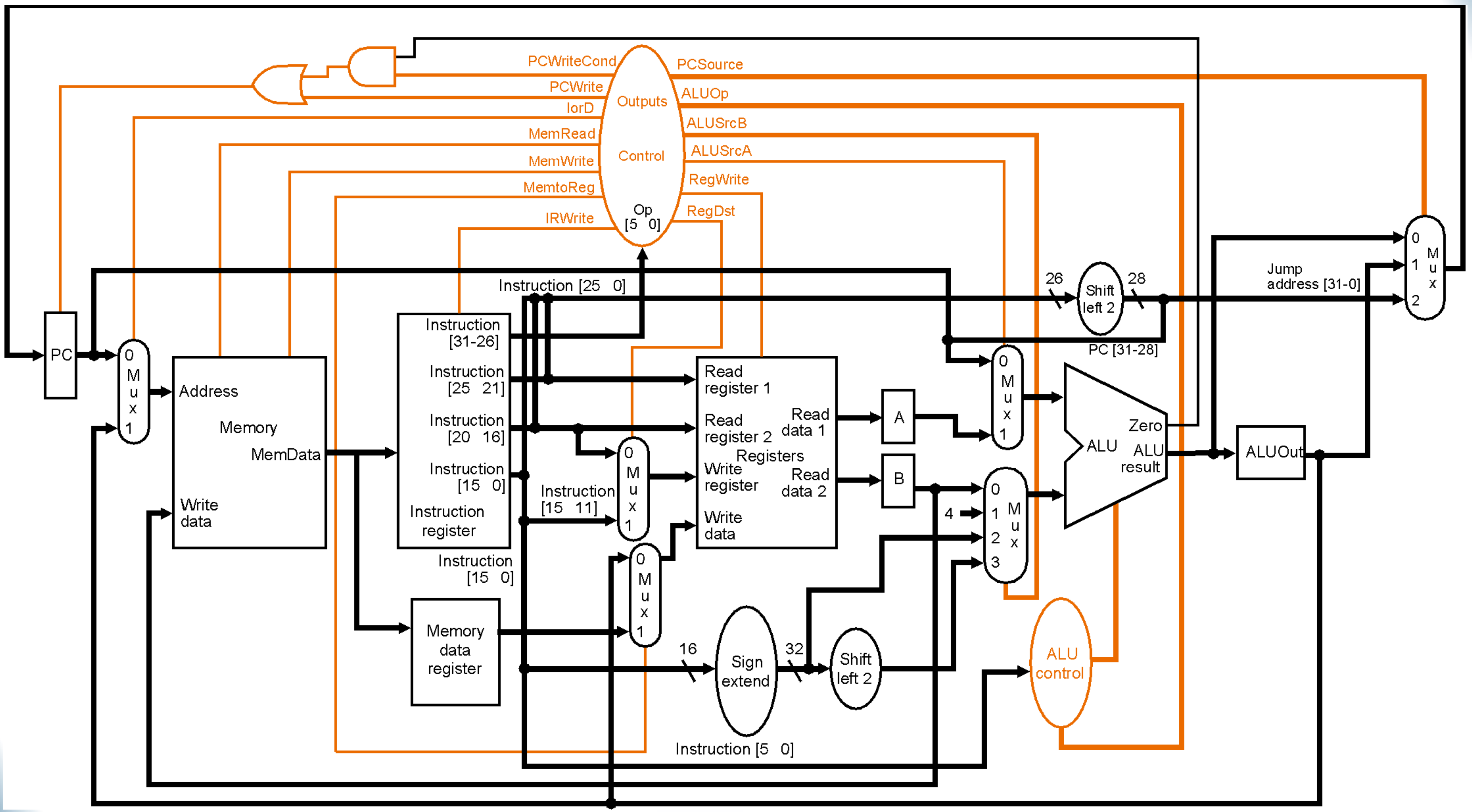
What type of processor encouraged having many different CISC instructions?

	Processor Type	Best explanation
A	Single-Cycle	Specialized CISC instructions improved performance
B	Single-Cycle	The specialized nature of CISC instructions meant a variety of instructions were needed just to perform most tasks
C	Multi-Cycle	Specialized CISC instructions improved performance
D	Multi-Cycle	The specialized nature of CISC instructions meant a variety of instructions were needed just to perform most tasks
E	Pipeline	Specialized CISC instructions improved performance

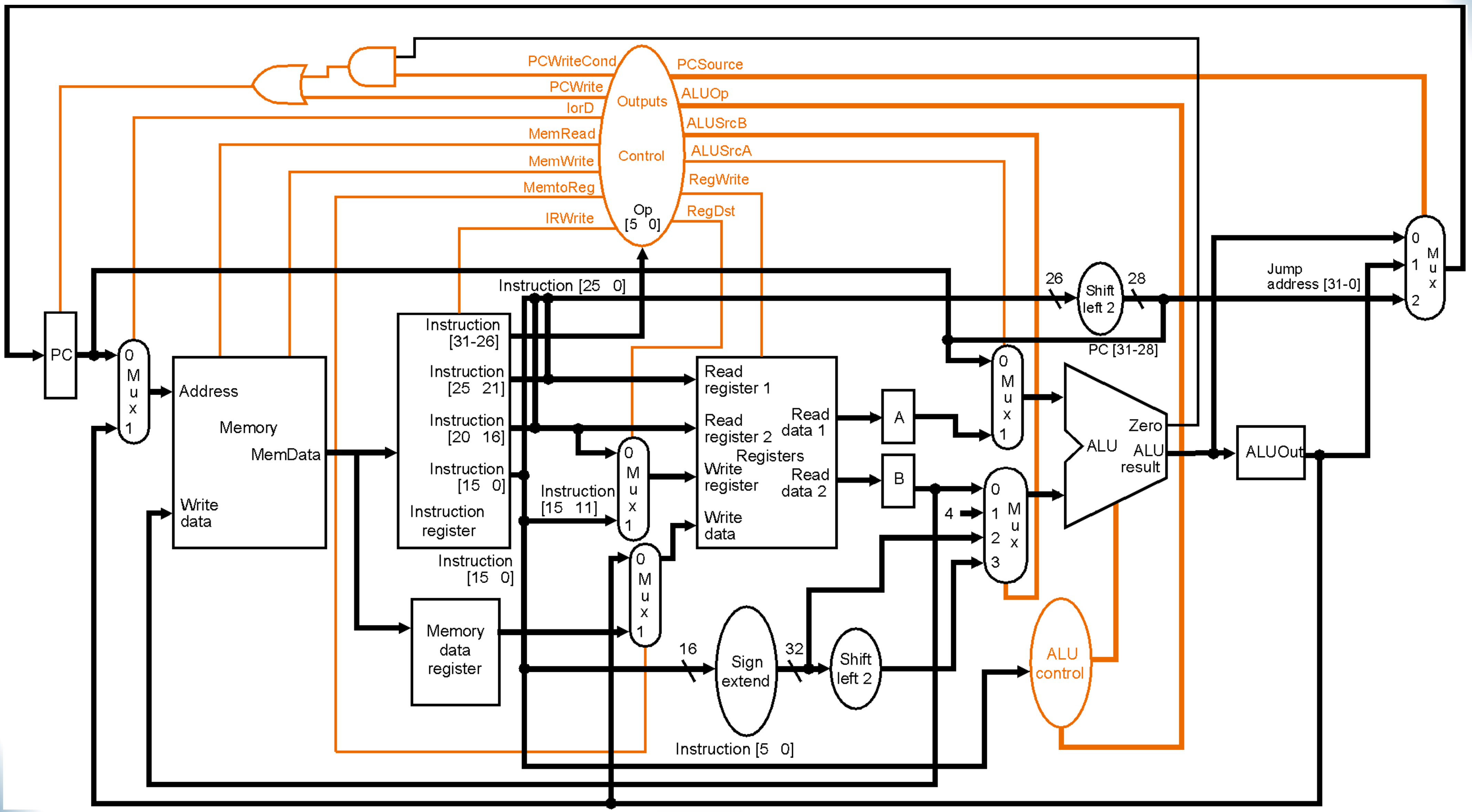
Suppose you work on an embedded ***multi-cycle MIPS32 processor*** and your software team tells you that every program which executes has to go through memory and zero 1k bytes of data fairly often (averages 10% of ET). You realize you could just have a single instruction do this called zero1k (rs) which does:
M[rs] = 0 … M[rs+1020] = 0.
Your coworker thinks you are crazy. You reply?

Selection	Crazy?	Reason
A	Yes	The complexity of such an instruction combined with no performance gain is just silly.
B	Yes	The complexity of such an instruction combined with minimal performance gain (<5%) is not worth it.
C	No	The minimal performance gains (<5%) still rationalize this simple instruction.
D	No	The significant performance gains (>5%) clearly rationalize this complex instruction.
E	Maybe	None of the above.

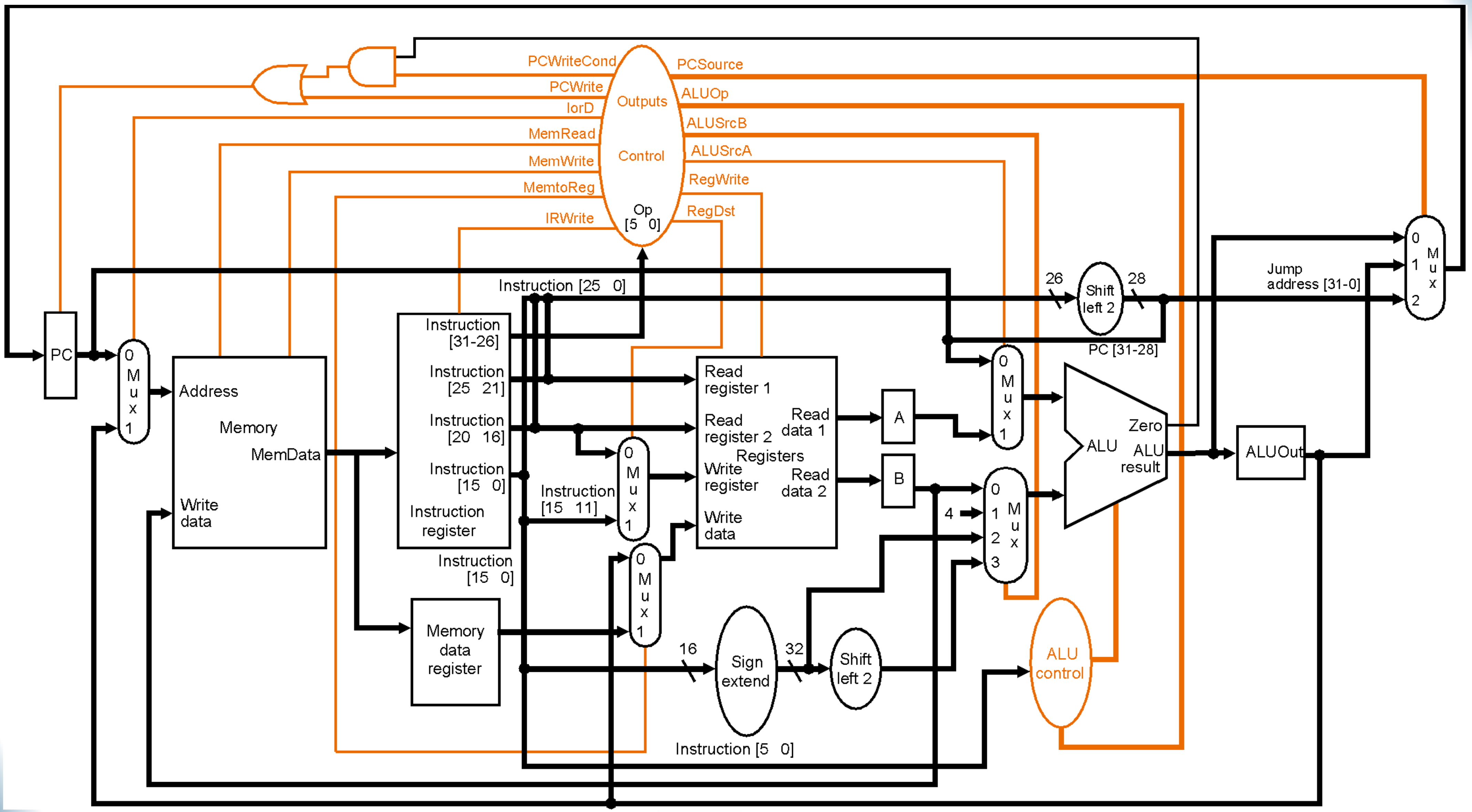
Complete Multicycle Datapath



Complete Multicycle Datapath



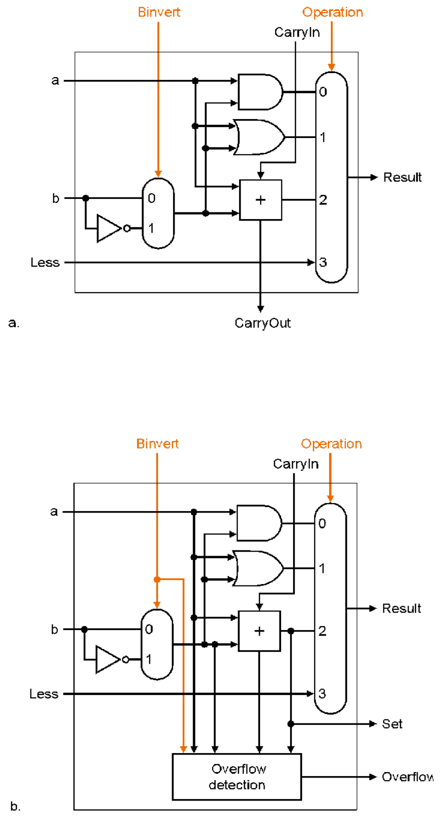
Complete Multicycle Datapath

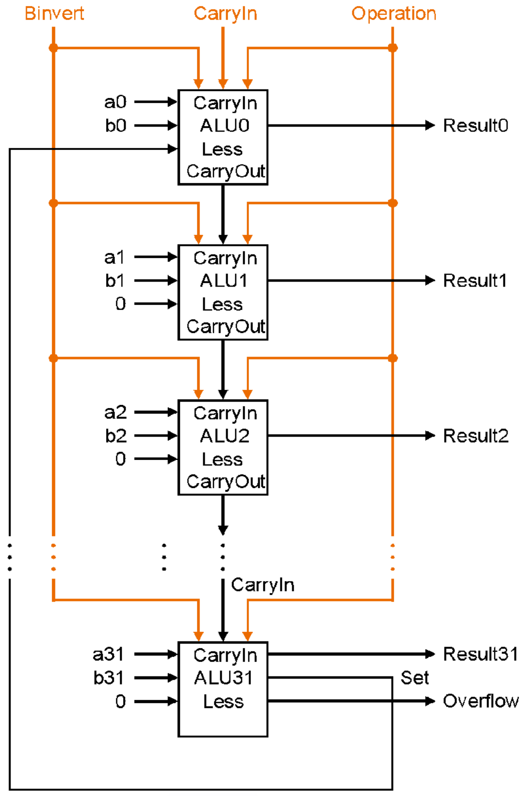


ALU Control SIGNALS

Full ALU

Consolidate to 3 wires since Binvert and CIn are always the same





what signals accomplish:

	Binvert	CIn	Oper
and?			
or?			
add?			
sub?			
beq?			
slt?			

And	0	0	0
Or	0	0	1
Add	0	0	2
Sub	1	1	2
Beq	1	1	2
Slt	1	1	3

ALU control bits

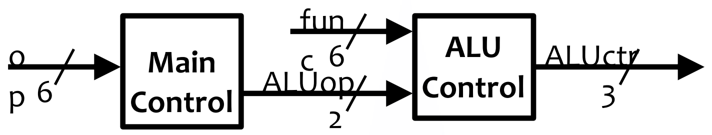
- Recall: 5-function ALU

Opcode (31-26) Function code (5-0)

ALU control input	Function	Operations
000	And	and
001	Or	or
010	Add	add, lw, sw
110	Subtract	sub, beq
111	Slt	slt

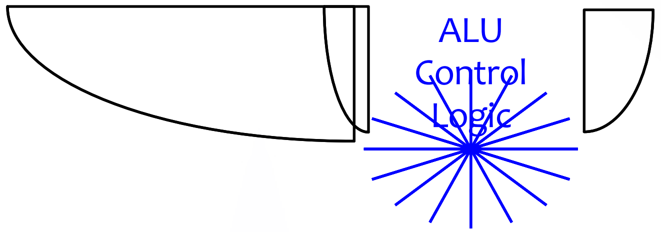
- based on Opcode (31-26) and Function code (5-0) from instruction
- ALU doesn't need to know all opcodes--we will summarize opcode with ALUOp (2 bits):

  00 - lw,sw 01 - beq 10 - R-format



Generating ALU control

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
lw	00	load word	xxxxxx	add	010
sw	00	store word	xxxxxx	add	010
beq	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	slt	101010	slt	111



Essentially a truth table, and we can design logic to do this.