

---

# **Chapter Four**

## **The Processor: Datapath and Control**

# 4.1 Introduction

---

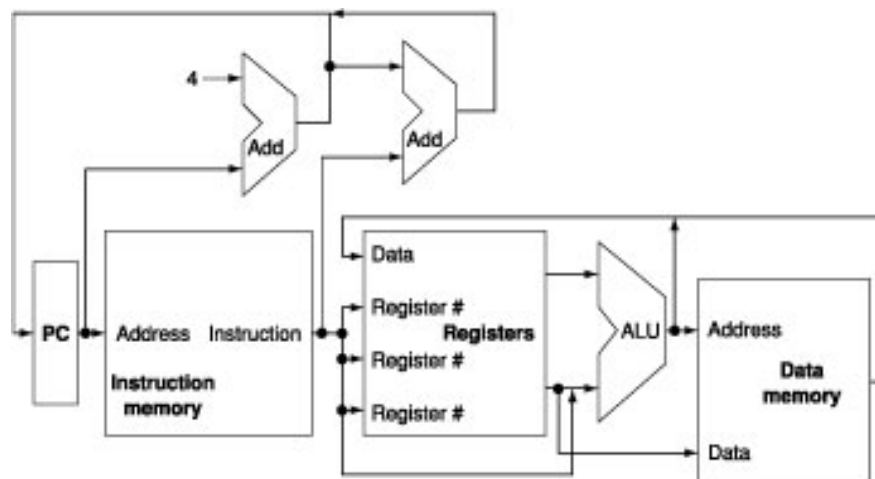
## A Basic MIPS Implementation

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw`, `sw`
  - arithmetic-logical instructions: `add`, `sub`, `and`, `or`, `slt`
  - control flow instructions: `beq`, `j`
- 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
- All instructions use the ALU after reading the registers
  - Why? memory-reference? arithmetic? control flow?

# An Overview of the Implementation

---

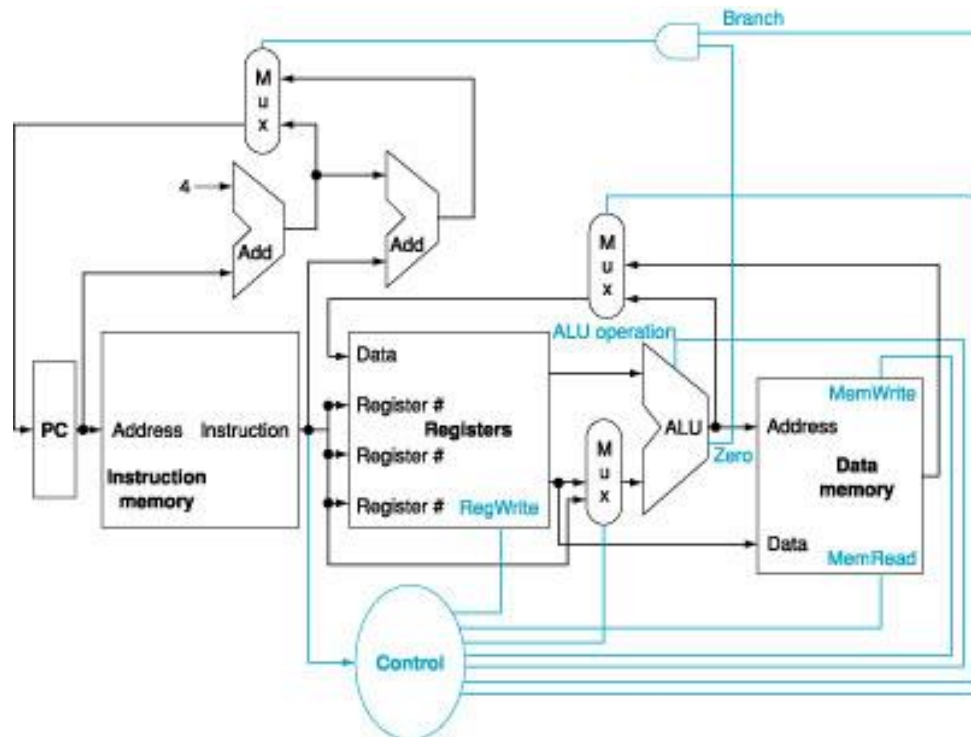
- For most instructions: **fetch** instruction, **fetch** operands, **execute**, **store**.
- An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them



- Missing **Multiplexers**, and some **Control lines** for read and write.

# Continue

- The basic implementation of the MIPS subset including the necessary multiplexers and control lines.



- **Single-cycle** datapath (long cycle for every instruction.
- **Multiple clock** cycles for each instruction>

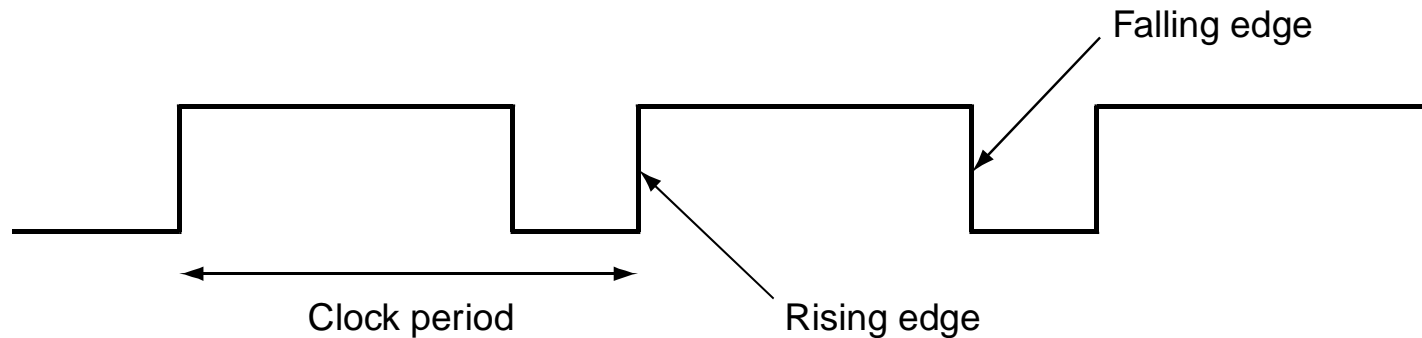
## 4.2 Logic Design Conventions

---

- **Combinational elements & State elements**

### State elements

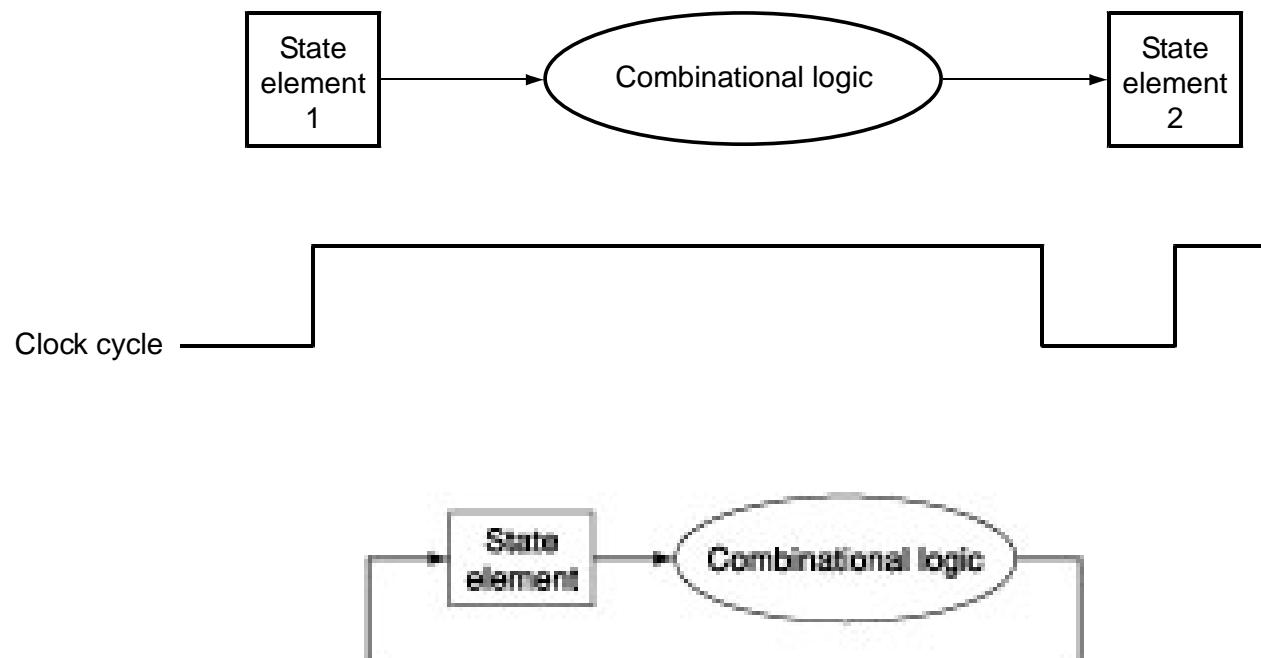
- **Unclocked vs. Clocked**
- **Clocks used in synchronous logic**
  - when should an element that contains state be updated?



# Clocking Methodology

---

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements

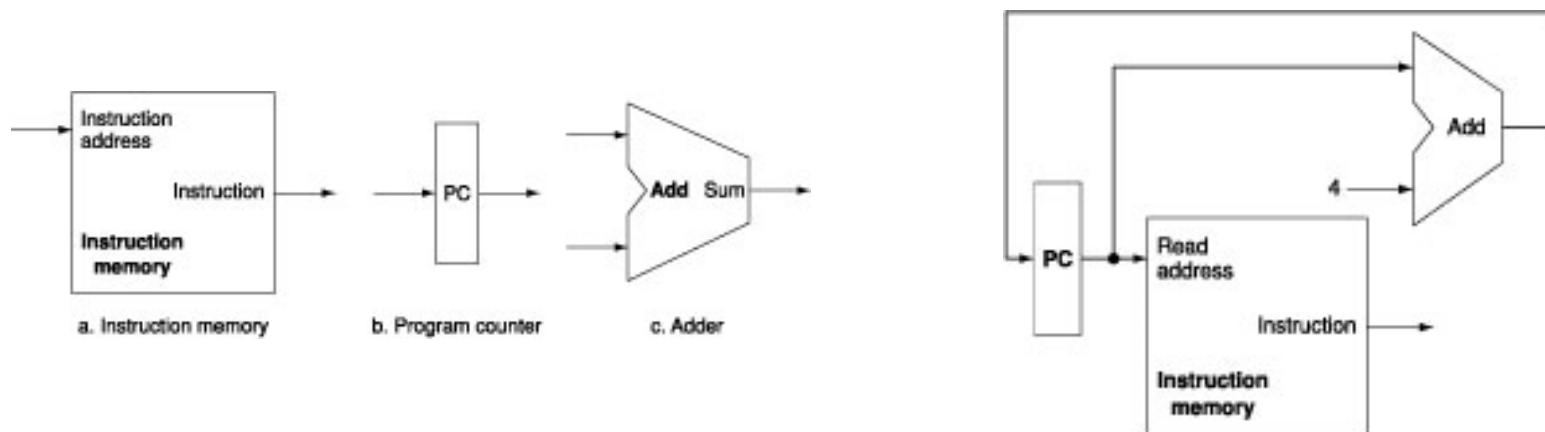


## 4.3 Building a Datapath

We need functional units (datapath elements) for:

1. Fetching instructions and incrementing the PC.
2. Execute arithmetic-logical instructions: `add`, `sub`, `and`, `or`, and `slt`
3. Execute memory-reference instructions: `lw`, `sw`
4. Execute branch/jump instructions: `beq`, `j`

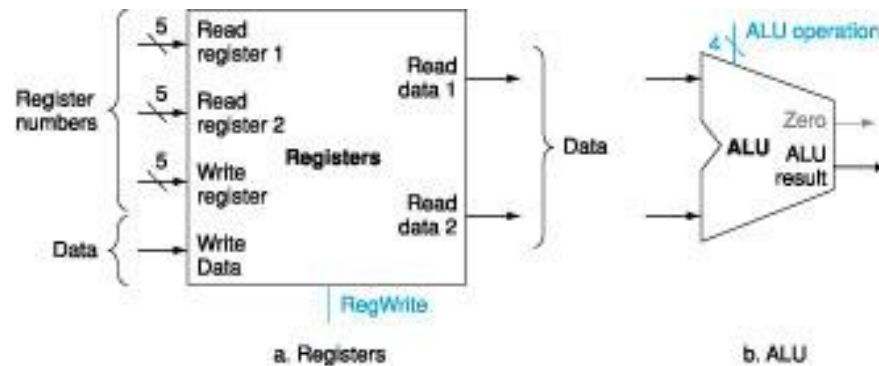
### 1. Fetching instructions and incrementing the PC.



# Continue

## 2. Execute arithmetic-logical instructions: **add**, **sub**, **and**, **or**, and **slt**

**add** \$t1, \$t2, \$t3    # t1 = t2 + t3



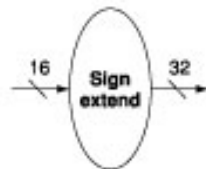
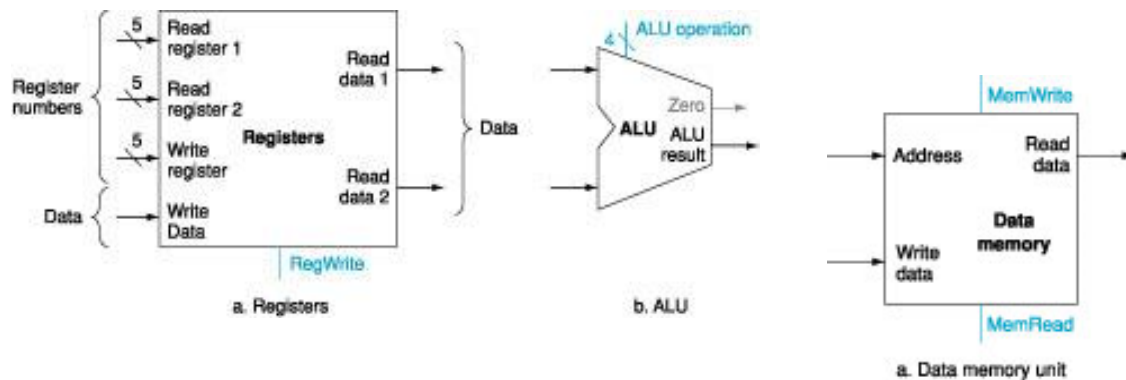


# Continue

## 3. Execute memory-reference instructions: `lw`, `sw`

`lw $t1, offset_value($t2)`

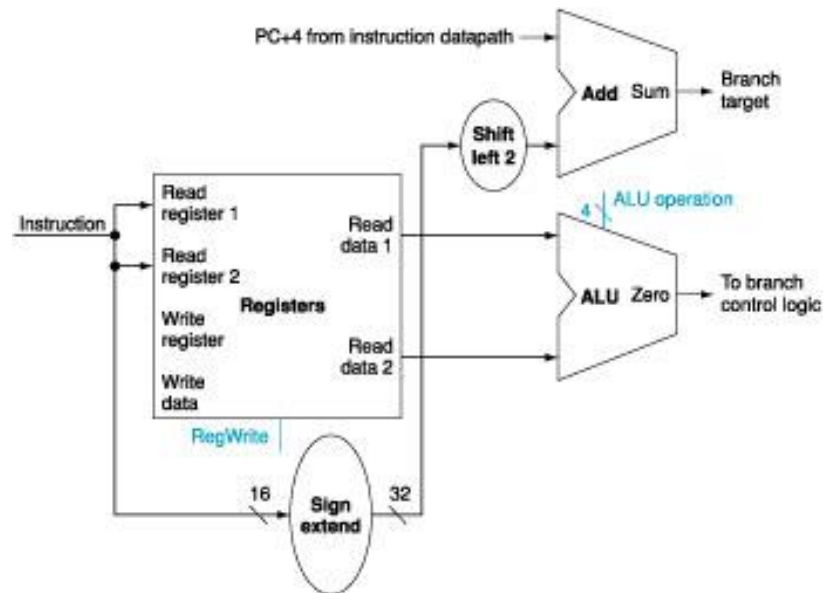
`sw $t1, offset_value($t2)`



b. Sign-extension unit

#### 4. Execute branch/jump instructions: `beq`, `j`

`beq $t1, $t2, offset`

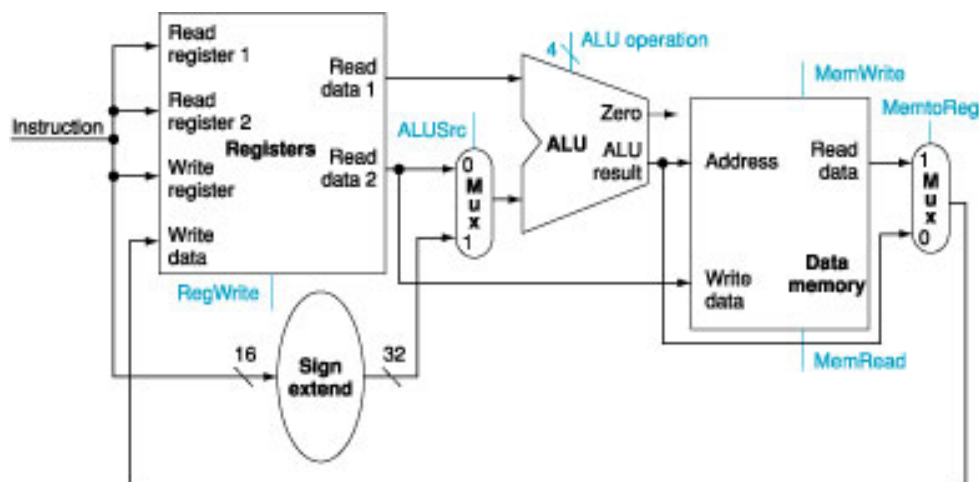


# Creating a Single Datapath

- Sharing datapath elements

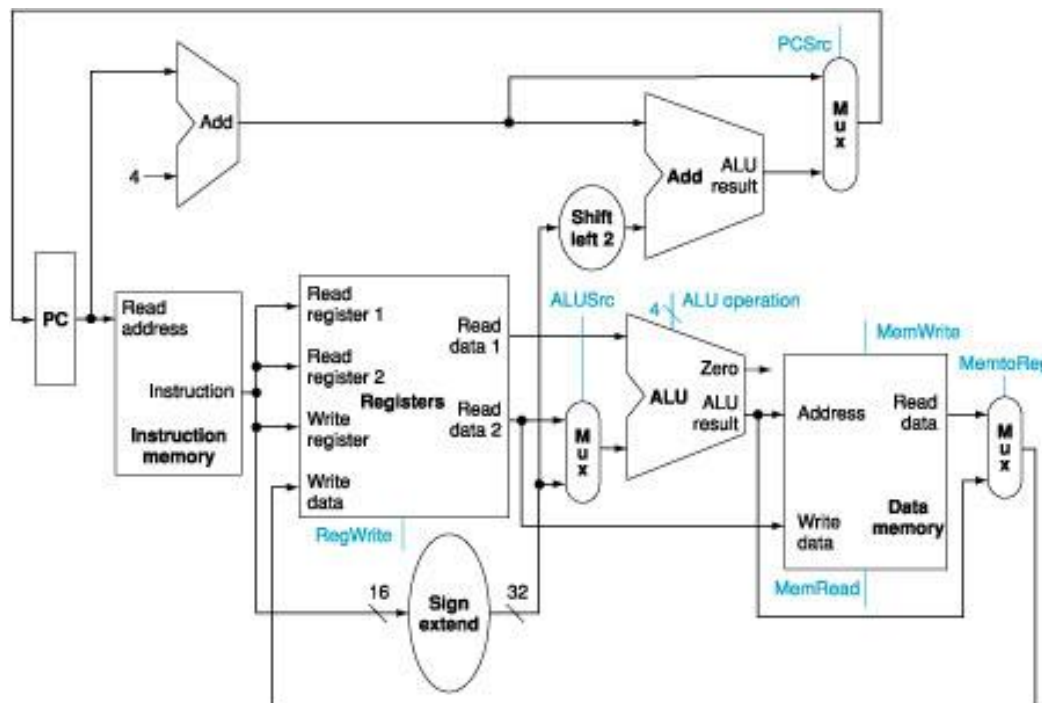
## Example:

Show how to built a datapath for arithmetic-logical and memory reference instructions.



# Continue

Now we can combine all the pieces to make a simple datapath for the MIPS architecture:



## 4.4 A Simple Implementation Scheme

### The ALU Control

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

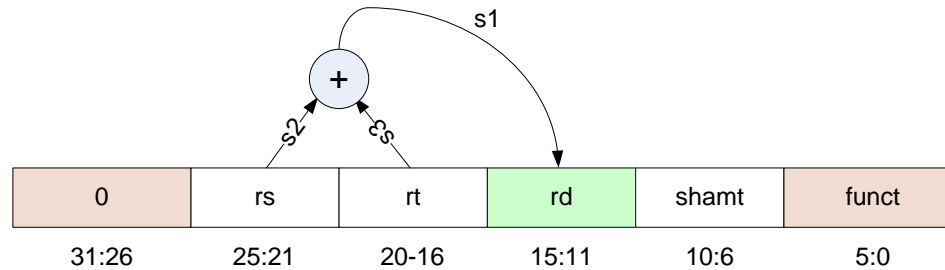
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 5.13 The truth table for the three ALU control bits (called Operation). The inputs

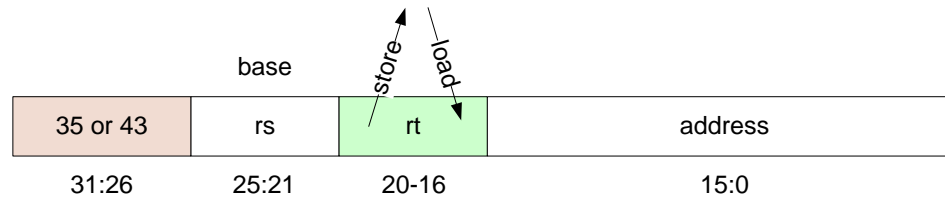
Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

FIGURE 5.12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction. The opcode, listed in the first column,

# Designing the Main Control Unit

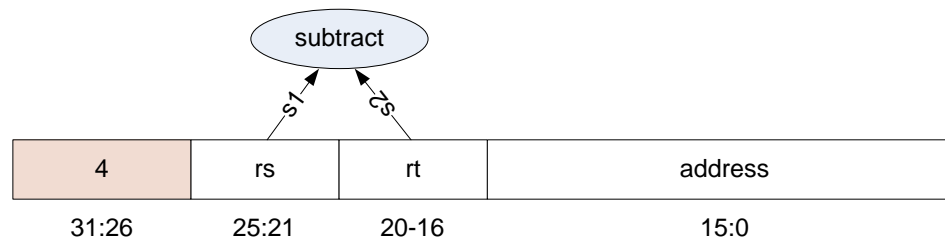


```
add    $s1, $s2, $s3
```



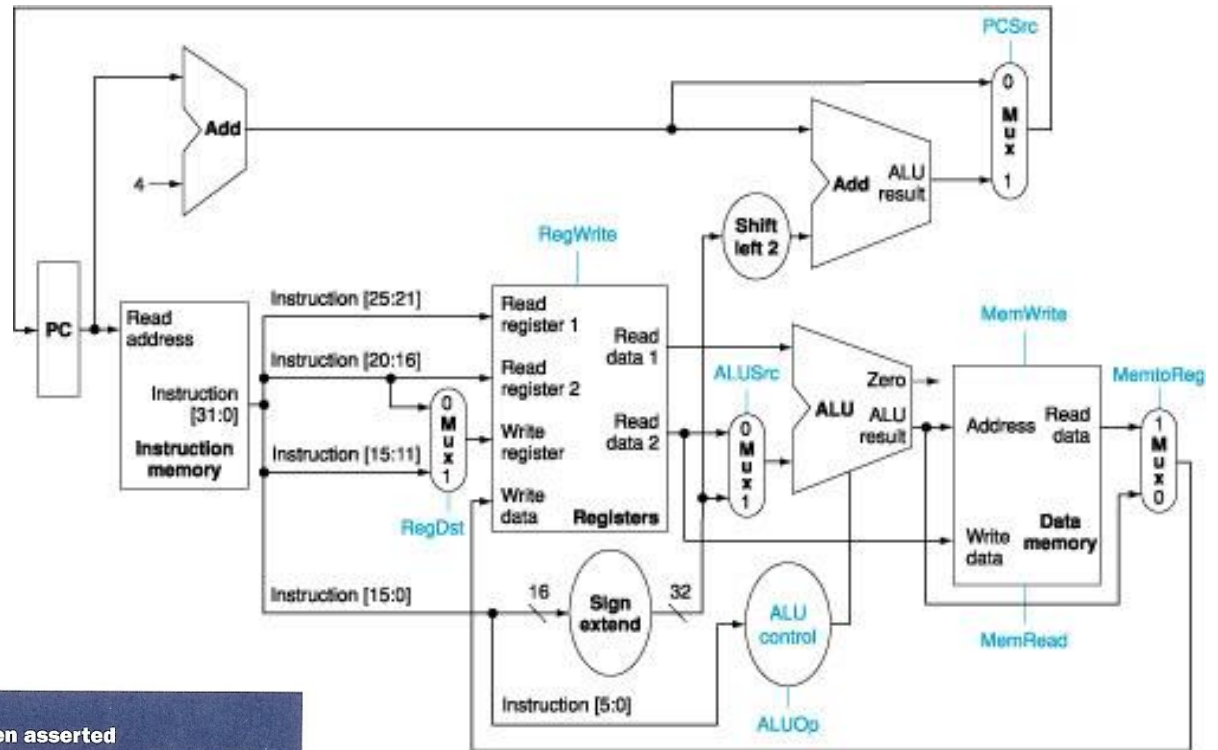
```
lw     $s1, 100($s2)
```

```
sw     $s1, 100($s2)
```



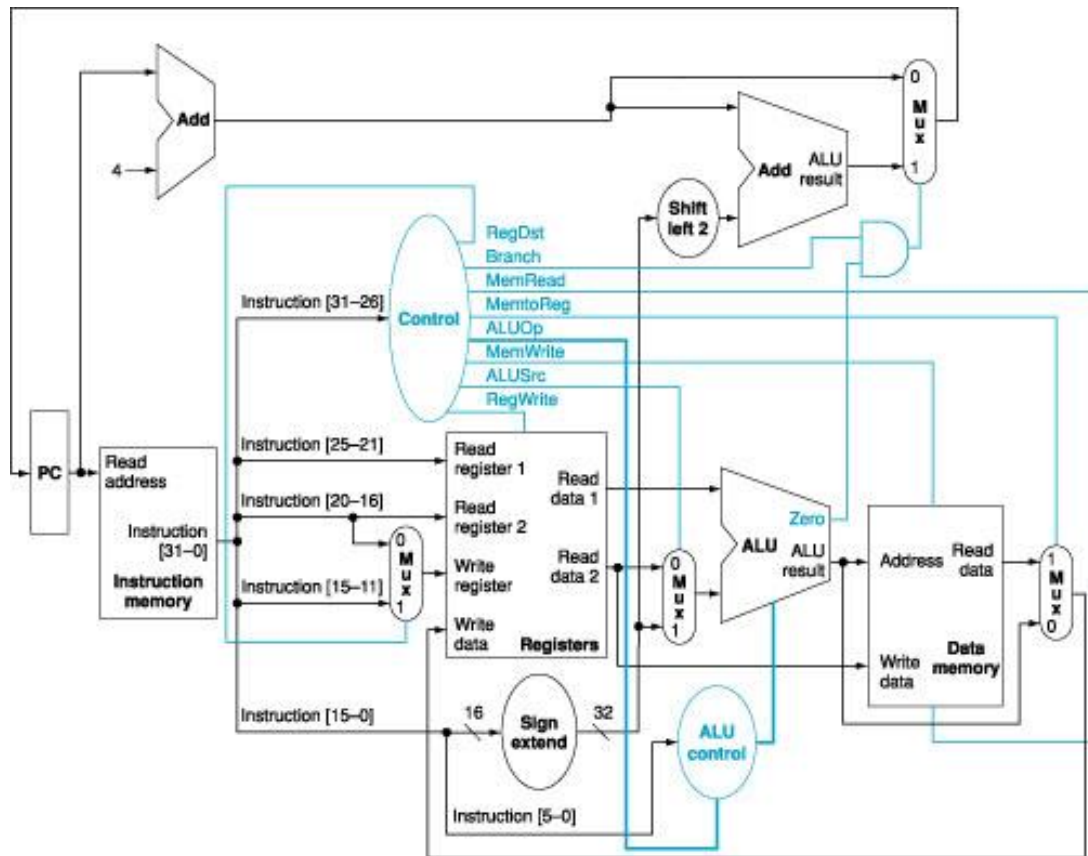
```
beq    $s1, $s2, L
```

# Continue



Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
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, lower 16 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.

# Continue



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

**FIGURE 5.18** The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of



# Finalizing the Control

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

**FIGURE C.2.1** The truth table for the three ALU control bits (called **Operation**) as a function of the ALUOp and function code field. This table is the same as that shown Figure 5.13.

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
X	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

a. The truth table for Operation2 = 1 (this table corresponds to the left bit of the Operation field in Figure C.2.1)

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

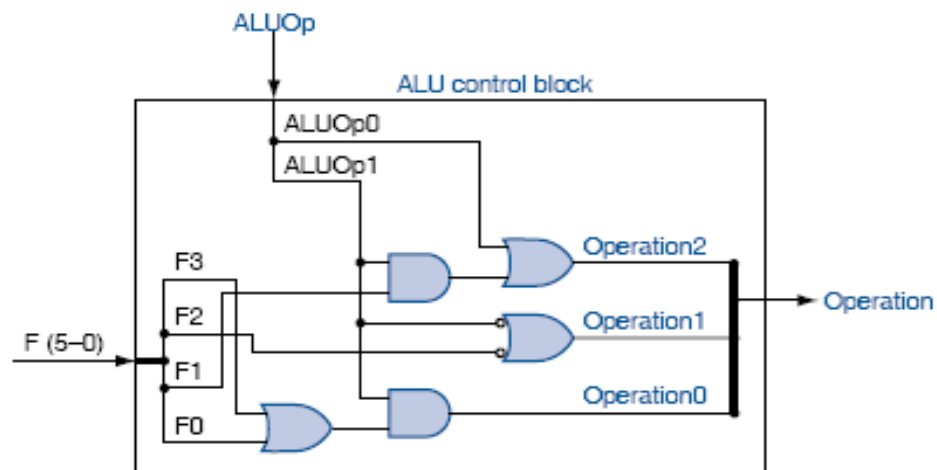
b. The truth table for Operation1 = 1

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X

c. The truth table for Operation0 = 1

**FIGURE C.2.2** The truth tables for the three ALU control lines. Only the entries for which the

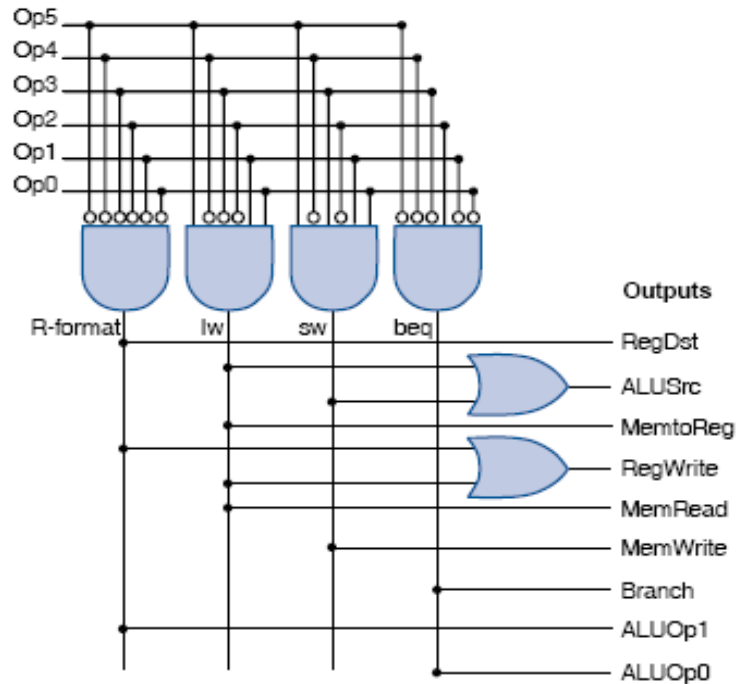
# Continue



**FIGURE C.2.3** The ALU control block generates the three ALU control bits, based on the function code and ALUOp bits. This logic is generated directly from the truth table in Figure C.2.2. Only

# Continue

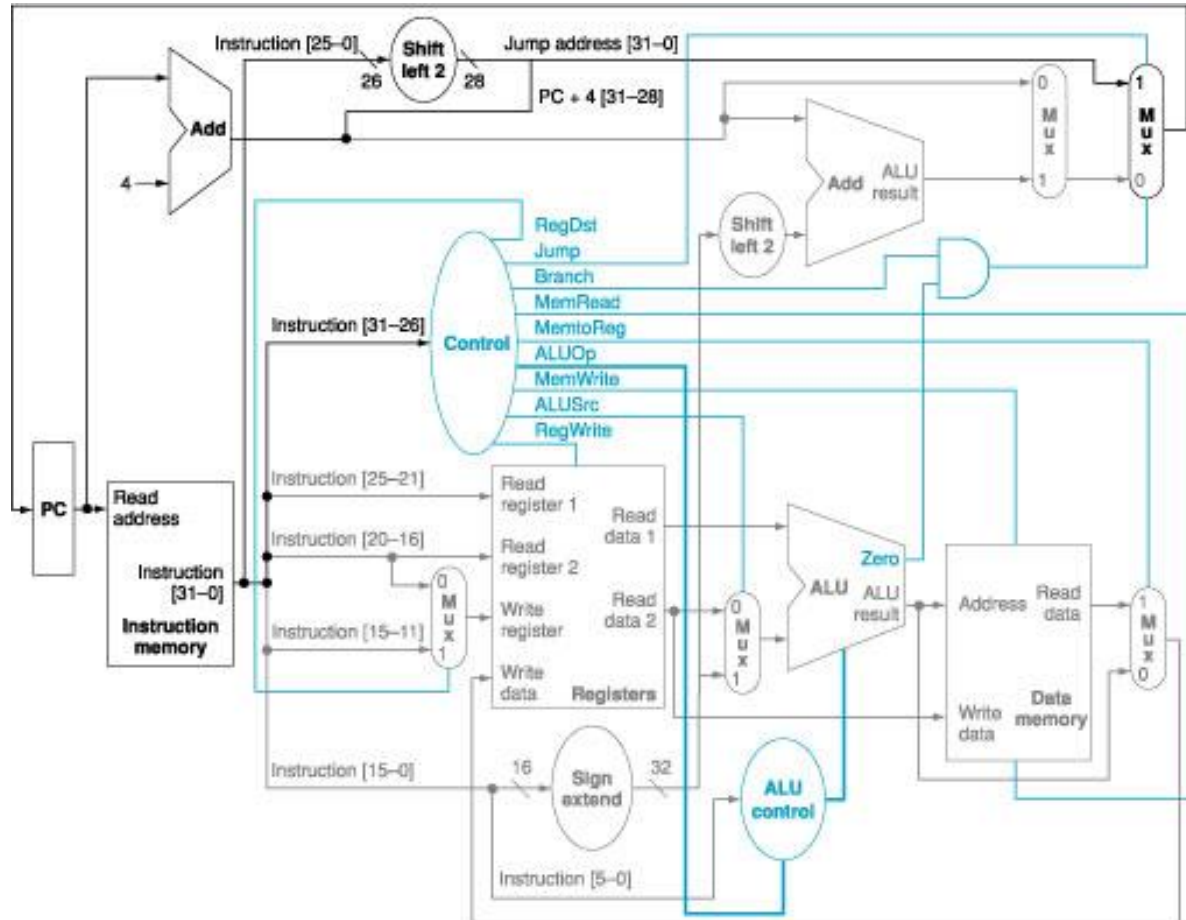
Inputs



Control	Signal name	R-format	lw	sw	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
	ALUOp0	0	0	0	1

**FIGURE C.2.4** The control function for the simple one-clock implementation is completely specified by this truth table. This table is the same as that shown in Figure 5.22.

# Example: Implementing Jumps



000010	address
31:26	25:0

# Why a Single-Cycle Implementation Is Not Used Today

---

## Example: Performance of Single-Cycle Machines

Calculate cycle time assuming negligible delays except:

- memory (200ps),
- ALU and adders (100ps),
- register file access (50ps)

Which of the following implementation would be faster:

1. When every instruction operates in 1 clock cycle of fixed length.
2. When every instruction executes in 1 clock cycle using a variable-length clock.

To compare the performance, assume the following instruction mix:

25% loads

10% stores

45% ALU instructions

15% branches, and

5% jumps

# Continue

memory (200ps),  
ALU and adders (100ps),  
register file access (50ps)

45% ALU instructions  
25% loads  
10% stores  
15% branches, and  
5% jumps

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Using these critical paths, we can compute the required length for each instruction class:

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

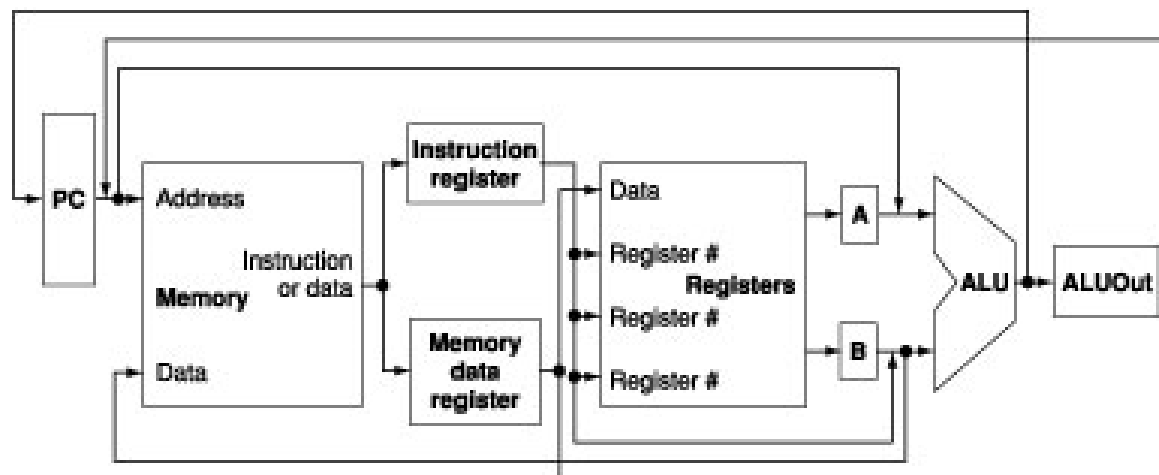
CPU clock cycle (option 1) = **600 ps.**

CPU clock cycle (option 2) =  $400 \times 45\% + 600 \times 25\% + 550 \times 10\% + 350 \times 15\% + 200 \times 5\%$   
= **447.5 ps.**

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

## 4.5 A Multicycle Implementation

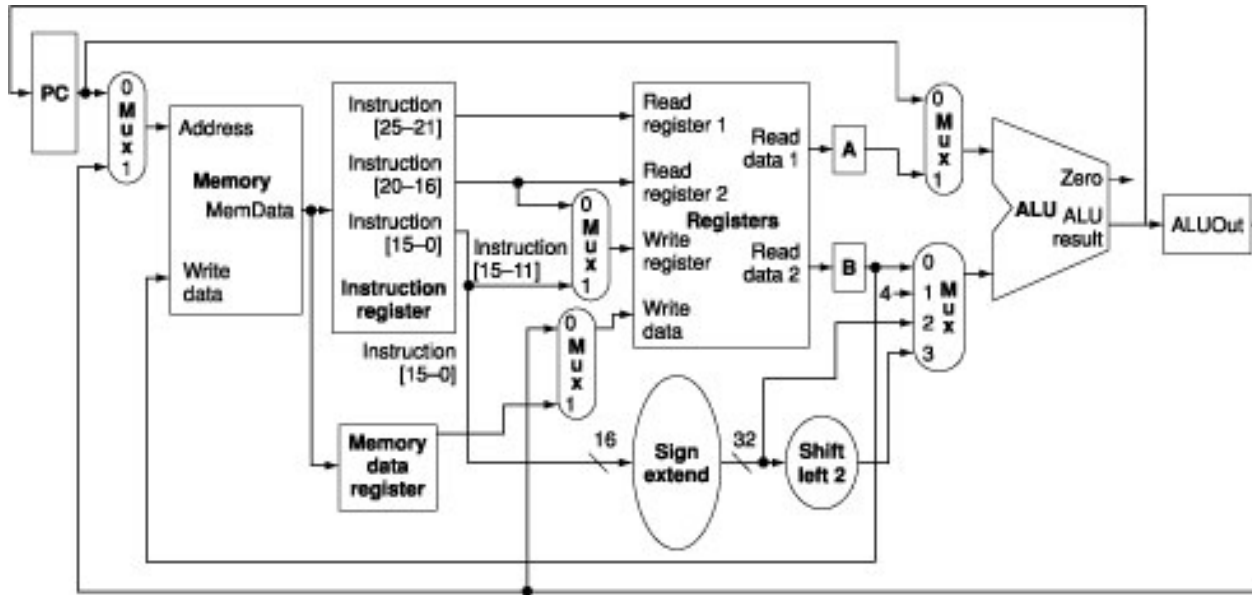
---



- A single memory unit is used for both instructions and data.
- There is a single ALU, rather than an ALU and two adders.
- One or more registers are added after every major functional unit.

# Continue

Replacing the three ALUs of the single-cycle by a single ALU means that the single ALU must accommodate all the inputs that used to go to the three different ALUs.

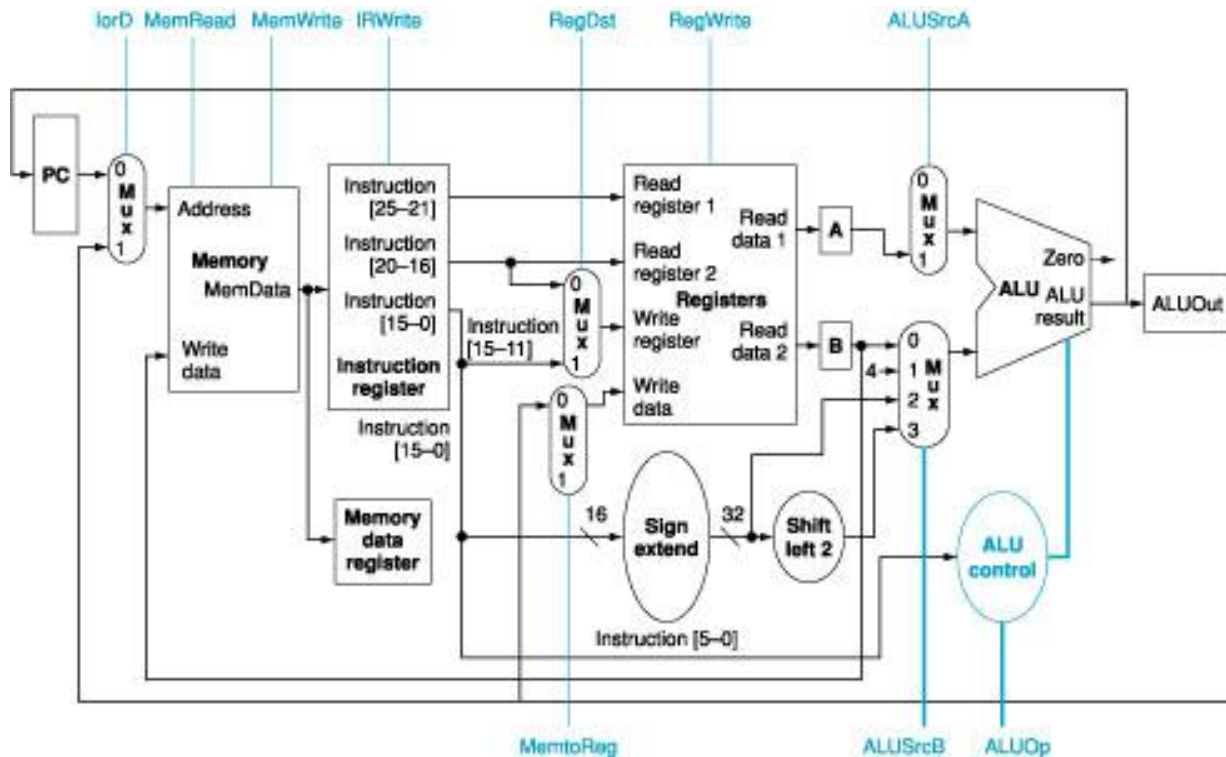




# Continue

## Control signals:

- The programmer-visible state units (PC, Memory, Register file) and IR → **write**  
Memory → **Read**
- ALU control: same as single cycle
- Multiplexor single/two control lines



# Continue

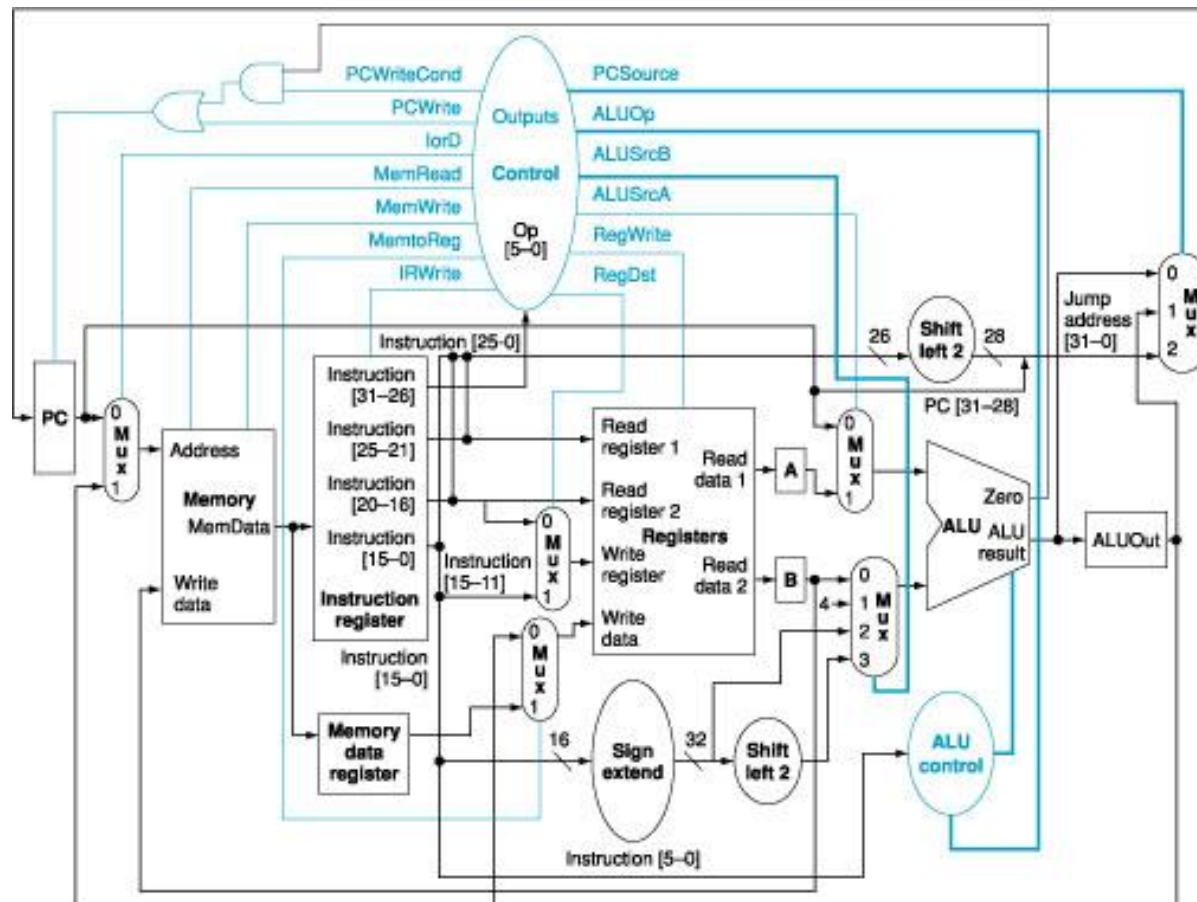
Three possible sources for the PC:

1. PC+4
2. ALUOut : address of the **beq**
3. Address for jump ( **j** )

PC write control signal:

PCWrite : PC+4 and **j** jump

PCWriteCond : **beq**



Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
lorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSrc.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

## Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSrc	00	Output of the ALU ( $PC + 4$ ) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address ( $IR[25:0]$ shifted left 2 bits and concatenated with $PC + 4[31:28]$ ) is sent to the PC for writing.

**FIGURE 5.29** The action caused by the setting of each control signal in Figure 5.28 on page 323. The top table describes the 1-bit control signals, while the bottom table describes the 2-bit signals. Only those control lines that affect multiplexors have an action when they are deasserted. This information is similar to that in Figure 5.16 on page 306 for the single-cycle datapath, but adds several new control lines (IRWrite, PCWrite, PCWriteCond, ALUSrcB, and PCSrc) and removes control lines that are no longer used or have been replaced (PCSrc, Branch, and Jump).

# Breaking the Instruction Execution into Clock Cycles

## 1. Instruction fetch step

```
IR <= Memory[PC];  
PC <= PC + 4;
```

IR <= Memory[PC];

MemRead

IRWrite

lorD = 0

PC <= PC + 4;

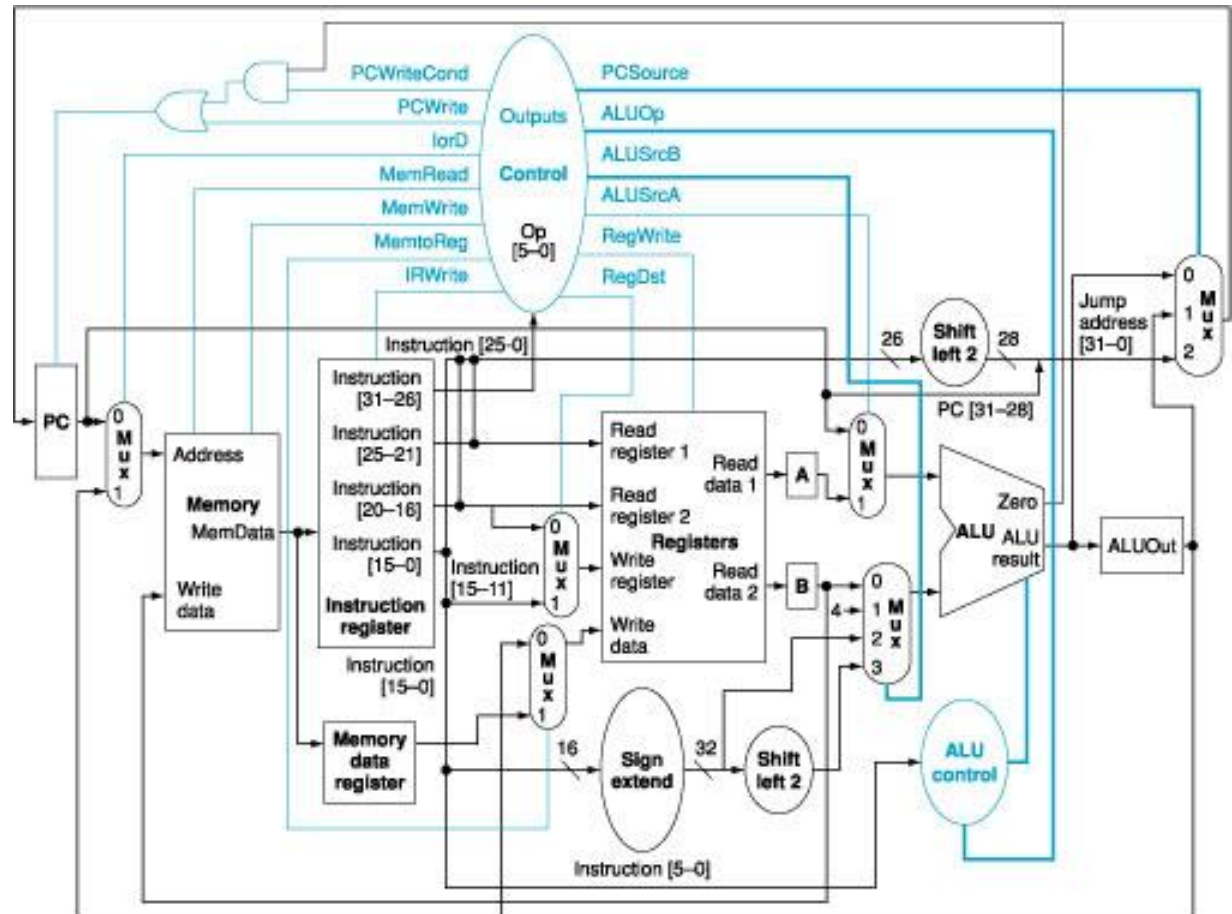
ALUSrcA = 0

ALUSrcB = 01

ALUOp = 00 (for add)

PCSource = 00

PCWrite



The increment of the PC and instruction memory access can occur in parallel, how?

# Breaking the Instruction Execution into Clock Cycles

---

## 2. Instruction decode and register fetch step

- Actions that are **either** applicable to all instructions
- **Or** are not harmful

```
A <= Reg[IR[25:21]];
```

```
B <= Reg[IR[20:16]];
```

```
ALUOut <= PC + (sign-extend(IR[15:0] << 2));
```

## 2. Instruction decode and register fetch step

$A \leftarrow \text{Reg}[\text{IR}[25:21]];$

$B \leftarrow \text{Reg}[\text{IR}[20:16]];$

$\text{ALUOut} \leftarrow \text{PC} + (\text{sign-extend}(\text{IR}[15:0] \ll 2));$

$A \leftarrow \text{Reg}[\text{IR}[25:21]];$

$B \leftarrow \text{Reg}[\text{IR}[20:16]];$

Since A and B are overwritten  
on every cycle  $\rightarrow$  Done

$\text{ALUOut} \leftarrow \text{PC} + (\text{sign-extend}(\text{IR}[15:0] \ll 2));$

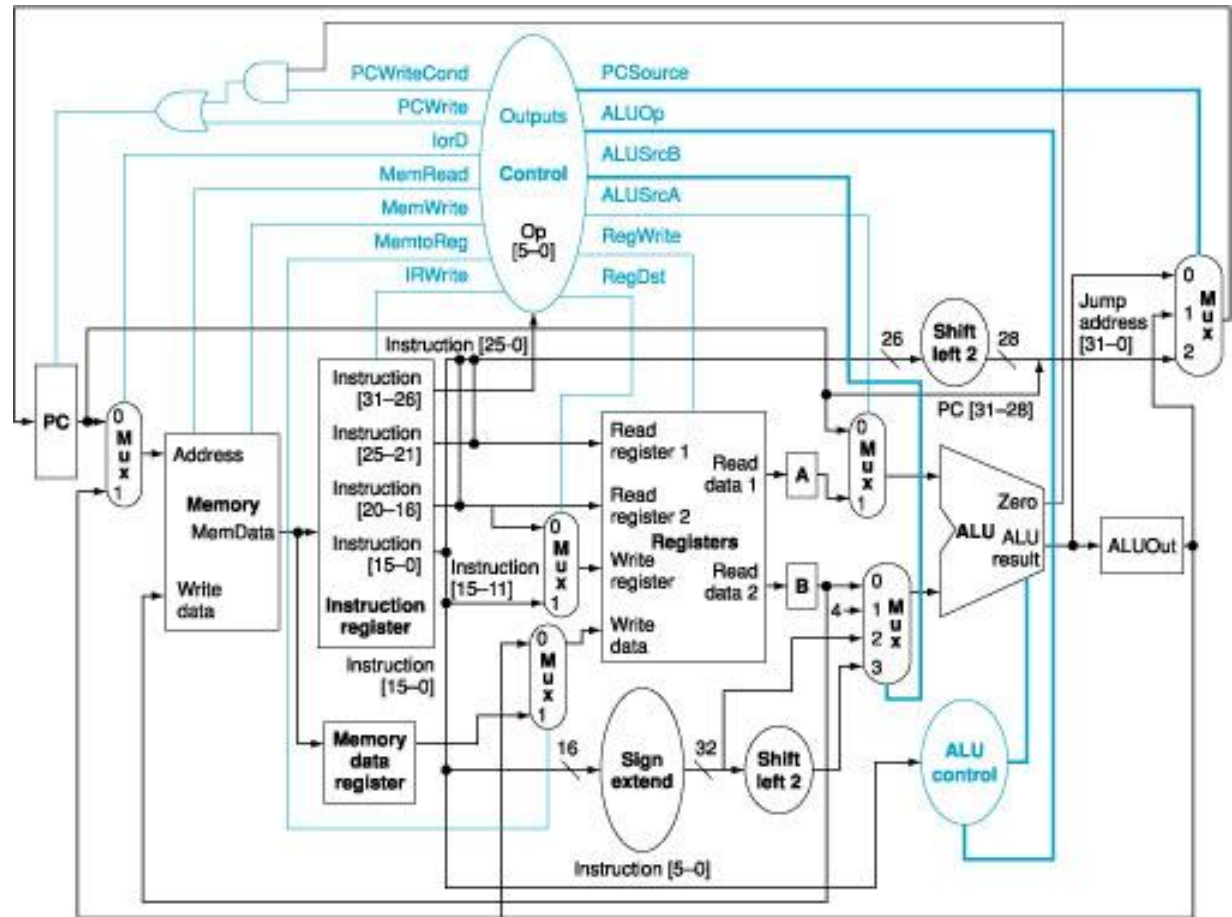
This requires:

$\text{ALUSrcA} \rightarrow 0$

$\text{ALUSrcB} \rightarrow 11$

$\text{ALUOp} \rightarrow 00$  (for add)

branch target address will be  
stored in  $\text{ALUOut}$ .



The register file access and computation of branch target occur in parallel.



# Breaking the Instruction Execution into Clock Cycles

---

## 3. Execution, memory address computation, or branch completion

### *Memory reference:*

```
ALUOut <= A + sign-extend(IR[15:0]);
```

### *Arithmetic-logical instruction:*

```
ALUOut <= A op B;
```

### *Branch:*

```
if (A == B) PC <= ALUOut;
```

### *Jump:*

```
PC <= { PC[31:28], (IR[25:0], 2'b00) };
```

### 3. Execution, memory address computation, or branch completion

#### Memory reference:

$ALUOut \leq A + \text{sign-extend}(IR[15:0]);$

$ALUSrcA = 1 \ \&\& \ ALUSrcB = 10$

$ALUOp = 00$

#### Arithmetic-logical instruction:

$ALUOut \leq A \ op \ B;$

$ALUSrcA = 1 \ \&\& \ ALUSrcB = 00$

$ALUOp = 10$

#### Branch:

$\text{if } (A == B) \ PC \leq ALUOut;$

$ALUSrcA = 1 \ \&\& \ ALUSrcB = 00$

$ALUOp = 01 \ (\text{for subtraction})$

$PCSource = 01$

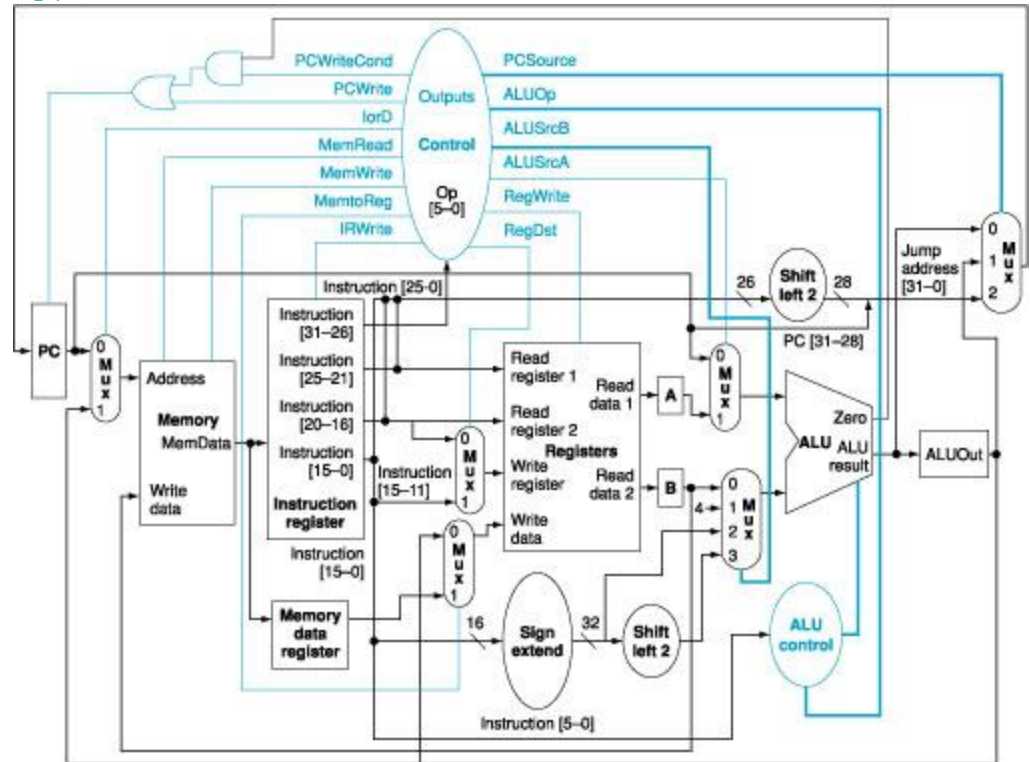
$PCWriteCond$

#### Jump:

$PC \leq \{ PC[31:28], (IR[25:0], 2'b00) \};$

$PCSource = 10$

$PCWrite$





# Breaking the Instruction Execution into Clock Cycles

---

## 4. Memory access or R-type instruction completion step

*Memory reference:*

<code>MDR &lt;= Memory [ALUOut];</code>	<code>MemRead</code>	} <code>lorD=1</code>
or		
<code>Memory [ALUOut] &lt;= B;</code>	<code>MemWrite</code>	

*Arithmetic-logical instruction (R-type):*

<code>Reg[IR[15:11]] &lt;= ALUOut;</code>	<code>RegDst=1</code>	<code>RegWrite</code>
	<code>MemtoReg=0</code>	

## 5. Memory read completion step

*Load:*

<code>Reg[IR[20:16]] &lt;= MDR;</code>	<code>MemtoReg=1</code>	<code>RegWrite</code>
<code>RegDst=0</code>		

# Breaking the Instruction Execution into Clock Cycles

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$IR \leftarrow \text{Memory}[PC]$ $PC \leftarrow PC + 4$			
Instruction decode/register fetch	$A \leftarrow \text{Reg}[IR[25:21]]$ $B \leftarrow \text{Reg}[IR[20:16]]$ $ALUOut \leftarrow PC + (\text{sign-extend}(IR[15:0]) \ll 2)$			
Execution, address computation, branch/jump completion	$ALUOut \leftarrow A \text{ op } B$	$ALUOut \leftarrow A + \text{sign-extend}(IR[15:0])$	if $(A == B)$ $PC \leftarrow ALUOut$	$PC \leftarrow \{PC[31:28], (IR[25:0], 2'b00)\}$
Memory access or R-type completion	$\text{Reg}[IR[15:11]] \leftarrow ALUOut$	Load: $MDR \leftarrow \text{Memory}[ALUOut]$ or Store: $\text{Memory}[ALUOut] \leftarrow B$		
Memory read completion		Load: $\text{Reg}[IR[20:16]] \leftarrow MDR$		

# Defining the Control

---

Two different techniques to design the control:

- Finite state machine
- Microprogramming

## Example: CPI in a Multicycle CPU

Using the SPECINT2000 instruction mix, which is: 25% **load**, 10% **store**, 11% **branches**, 2% **jumps**, and 52% **ALU**.

What is the CPI, assuming that each state in the multicycle CPU requires 1 clock cycle?

## Answer:

The number of clock cycles for each instruction class is the following:

- Load: 5
- Stores: 4
- ALU instruction: 4
- Branches: 3
- Jumps: 3

# Example Continue

---

The CPI is given by the following:

$$CPI = \frac{CPU \text{ clock cycles}}{Instruction \text{ count}} = \frac{\sum Instruction \text{ count}_i \times CPI_i}{Instruction \text{ count}}$$

$$CPI = \sum \frac{Instruction \text{ count}_i}{Instruction \text{ count}} \times CPI_i$$

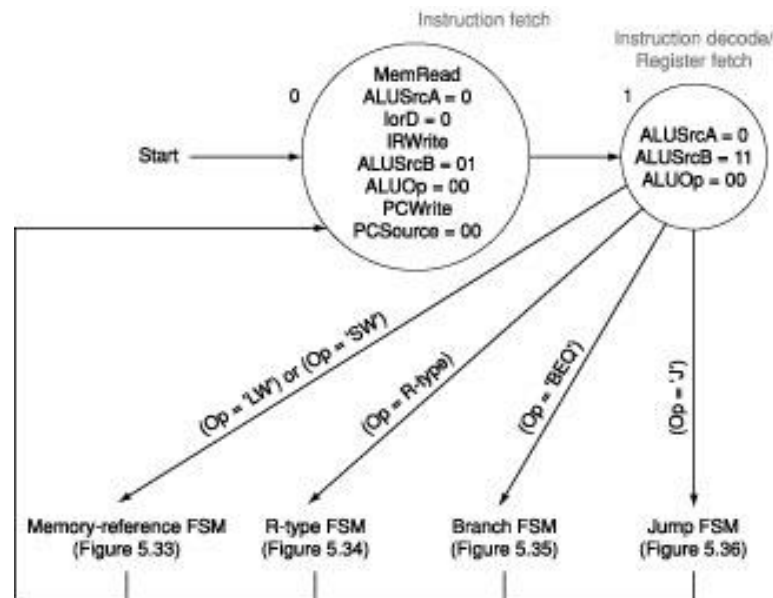
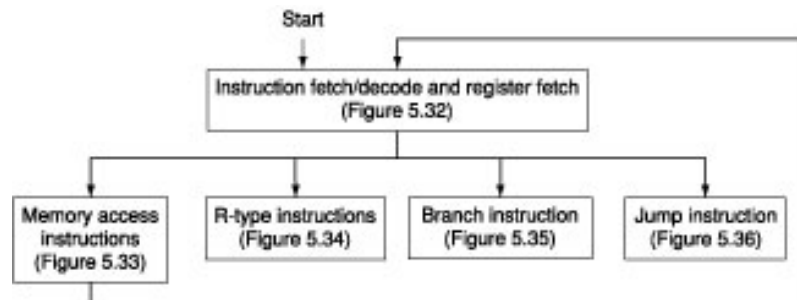
The ratio

$$\frac{Instruction \text{ count}_i}{Instruction \text{ count}}$$

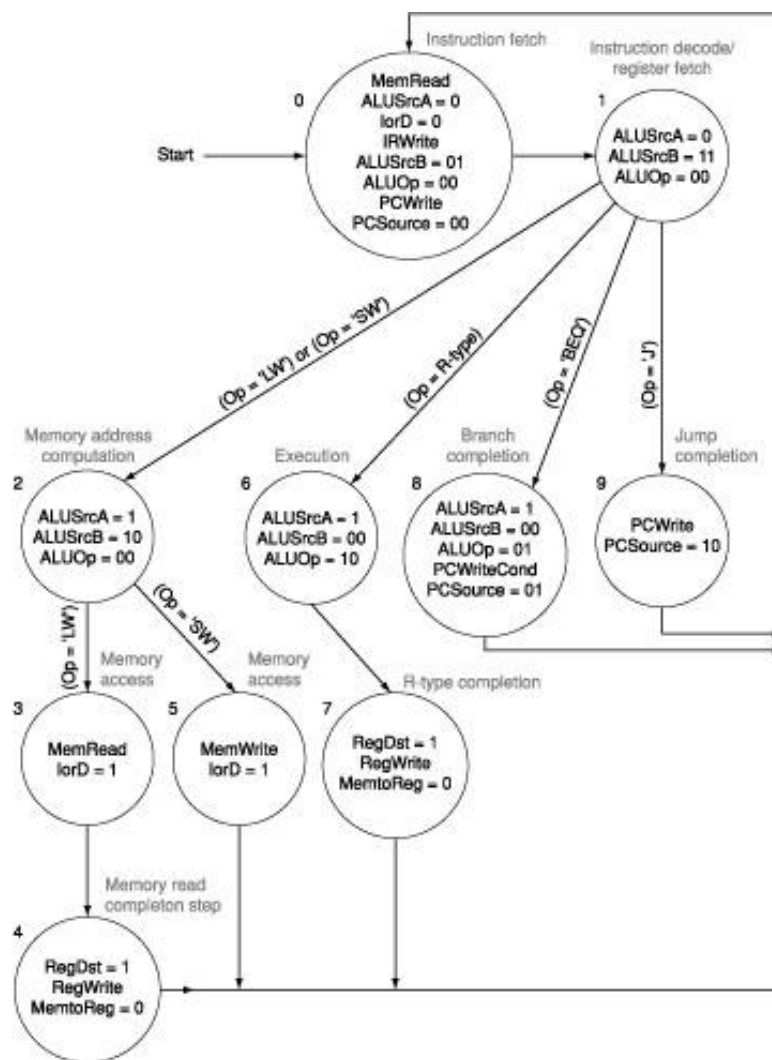
is simply the instruction frequency for the instruction class i. We can therefore substitute to obtain:

$$CPI = 0.25 \times 5 + 0.10 \times 4 + 0.52 \times 4 + 0.11 \times 3 + 0.02 \times 3 = 4.12$$

This CPI is better than the worst-case CPI of 5.0 when all instructions take the same number of clock cycles.



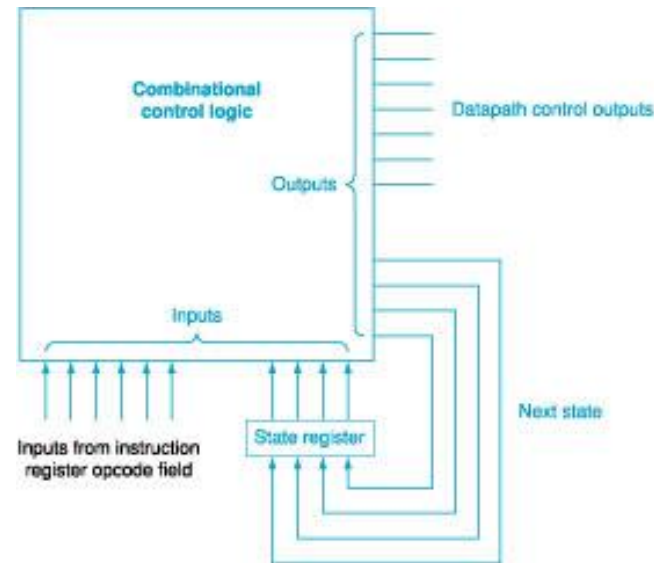
# The complete finite state machine control



# Continue

---

- **Finite state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state.**



## 4.6 Exceptions

---

- Exceptions
- Interrupts

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunction	Either	Exception or interrupt



# How Exception Are Handled

---

To communicate the reason for an exception:

1. a status register ( called the Cause register)
2. vectored interrupts

Exception type	Exception vector address (in hex)
Undefined instruction	C000 0000 <sub>hex</sub>
Arithmetic overflow	C000 0020 <sub>hex</sub>

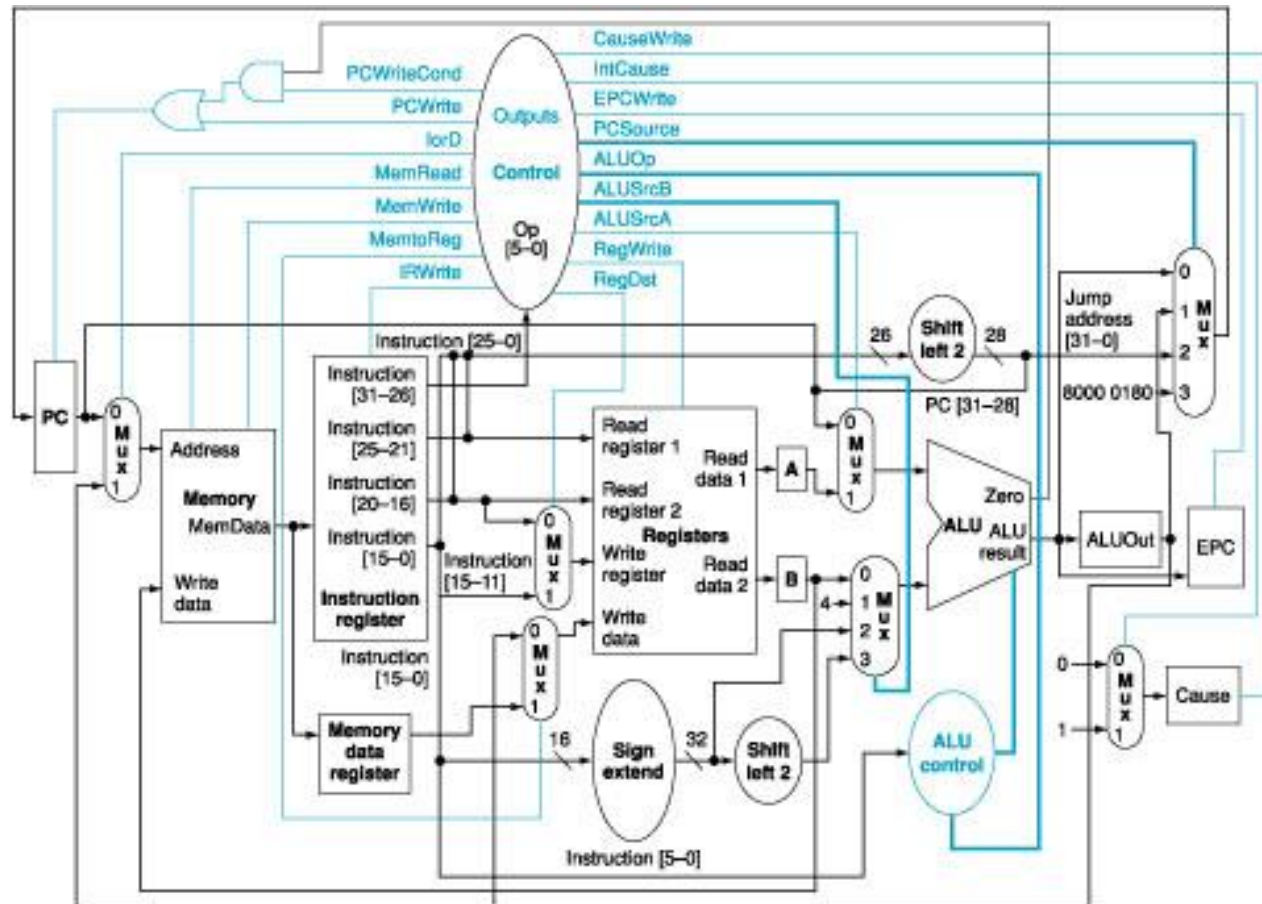
# How Control Checks for Exception

---

**Assume two possible exceptions:**

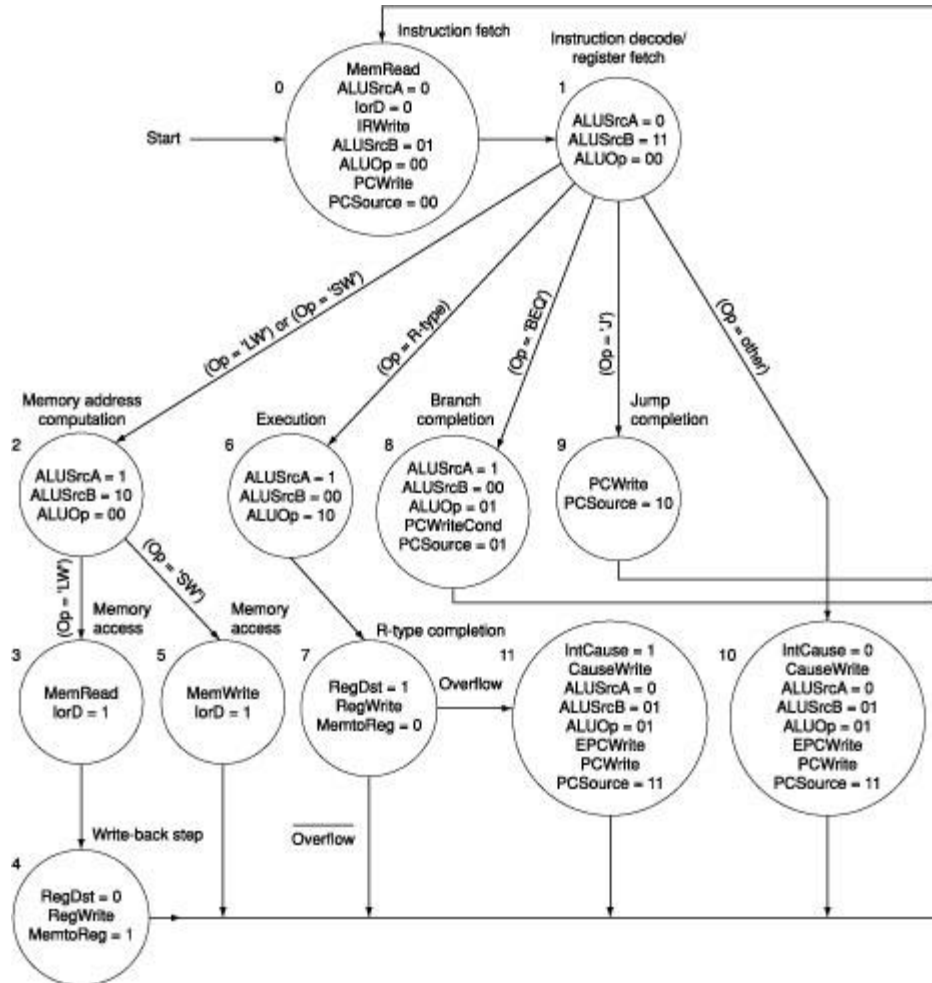
- **Undefined instruction**
- **Arithmetic overflow**

# Continue



The multicycle datapath with the addition needed to implement exceptions

# Continue



The finite state machine with the additions to handle exception detection

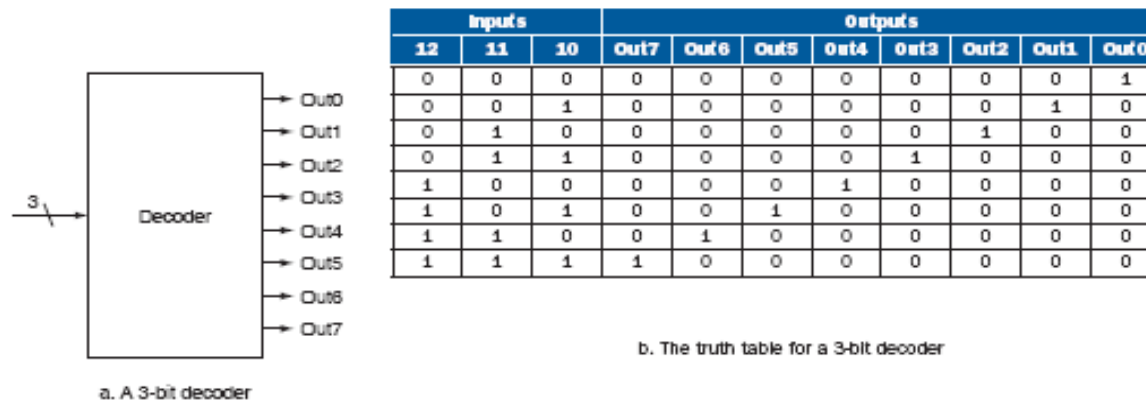
---

# **Appendix B**

## **The Basic of Logic Design**

## B.3 Combinational Logic

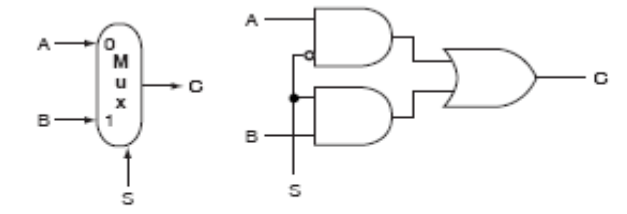
### Decoders:



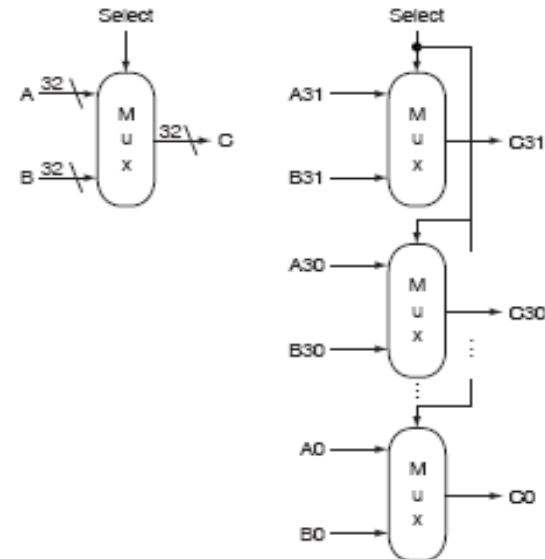
**FIGURE B.3.1** A 3-bit decoder has 3 inputs, called 12, 11, and 10, and  $2^3 = 8$  outputs, called Out0 to Out7. Only the output corresponding to the binary value of the input is true, as shown in the truth table. The label 3 on the input to the decoder says that the input signal is 3 bits wide.

# Multiplexors

## 2×1 Mux:



## A 32-bit wide 2×1 Mux:

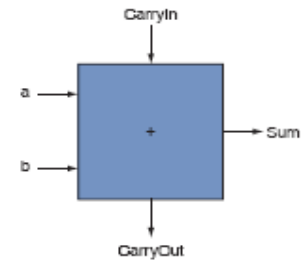
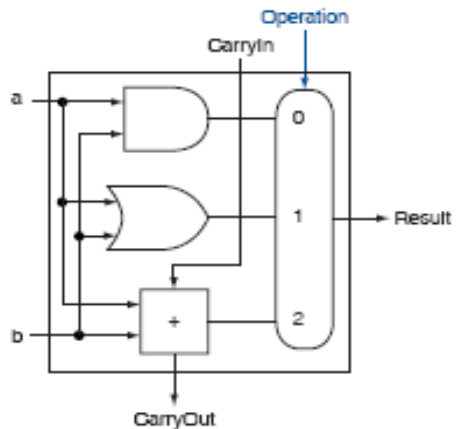
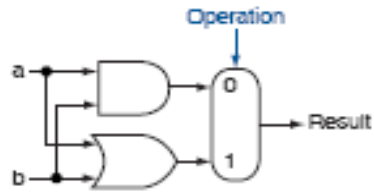


a. A 32-bit wide 2-to-1 multiplexor

b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

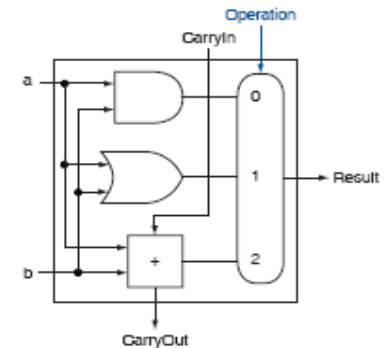
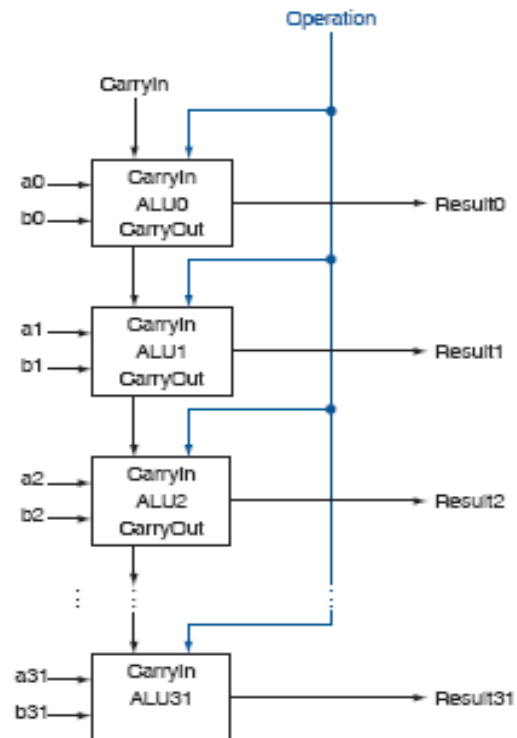
## B.5 Constructing a Basic Arithmetic Logic Unit

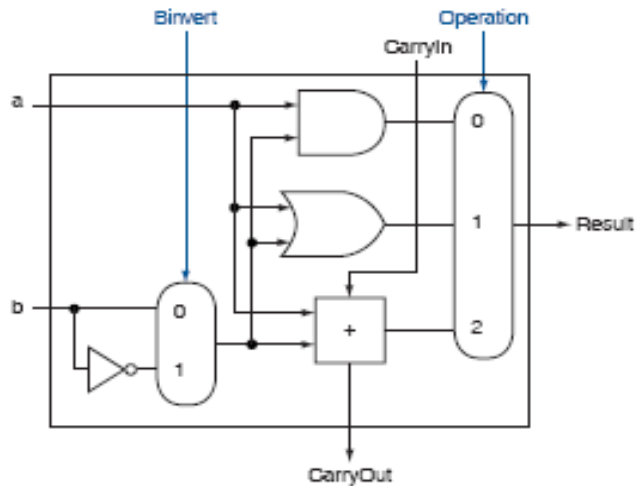
### A 1-Bit ALU



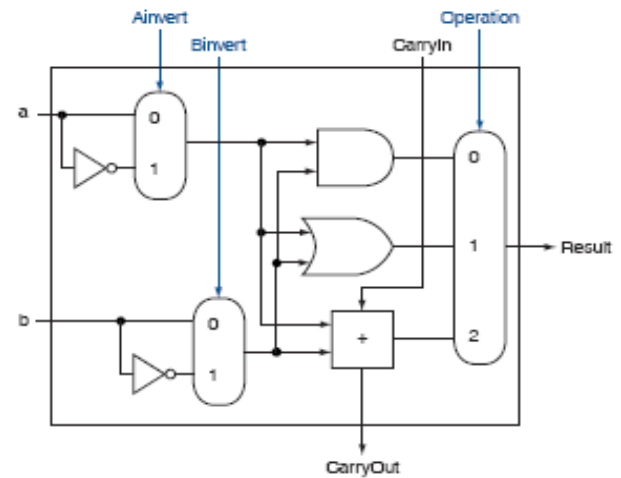


## A 32-Bit ALU





**A 1-bit ALU that perform AND, OR, addition and subtraction:**



**NOR**

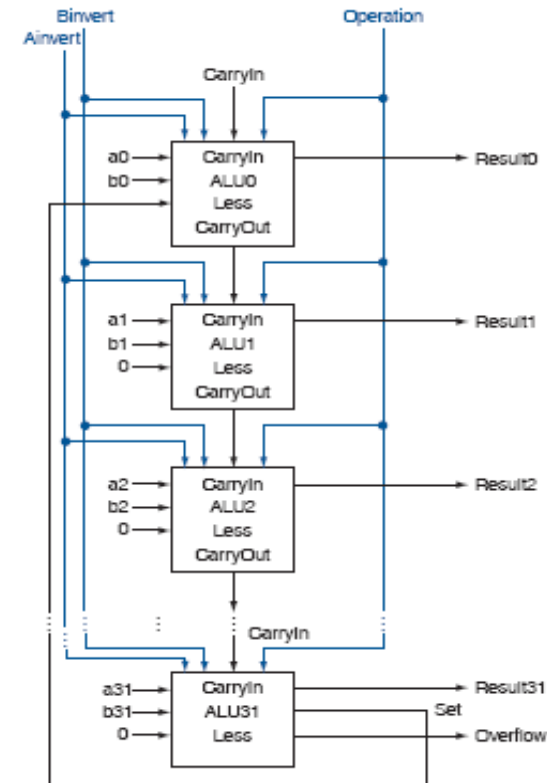
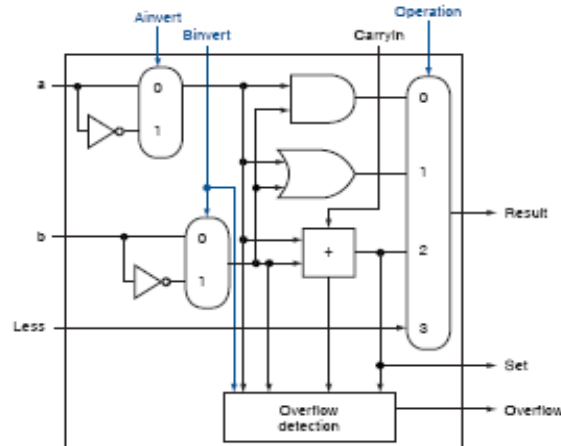
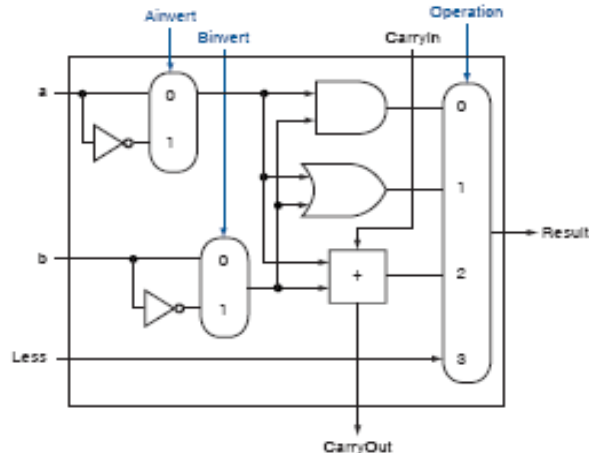
$$\overline{a + b} = \overline{a} \cdot \overline{b}$$

# Tailoring the 32-Bit ALU to MIPS

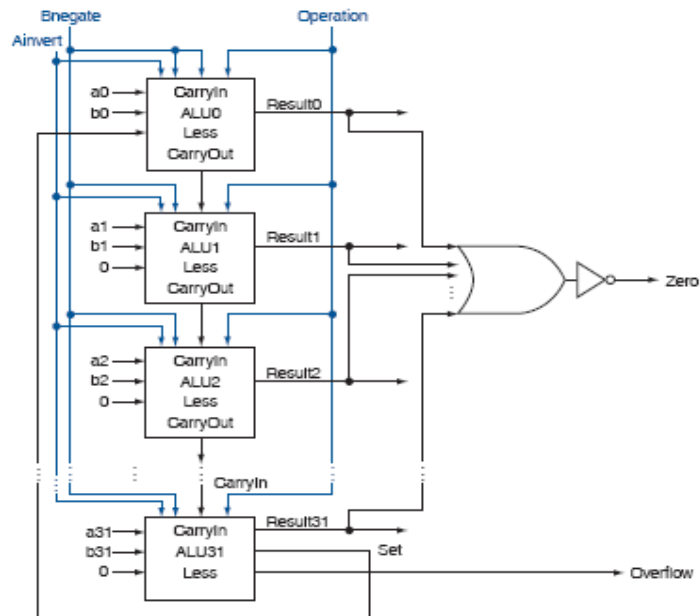
- slt

$$(a - b) < 0 \Rightarrow ((a - b) + b) < (0 + b)$$

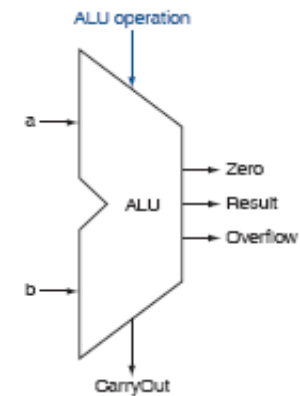
that is, a 1  $\Rightarrow a < b$  if  $a - b$  is negative and 0 if it's positive  $\rightarrow$  sign bit



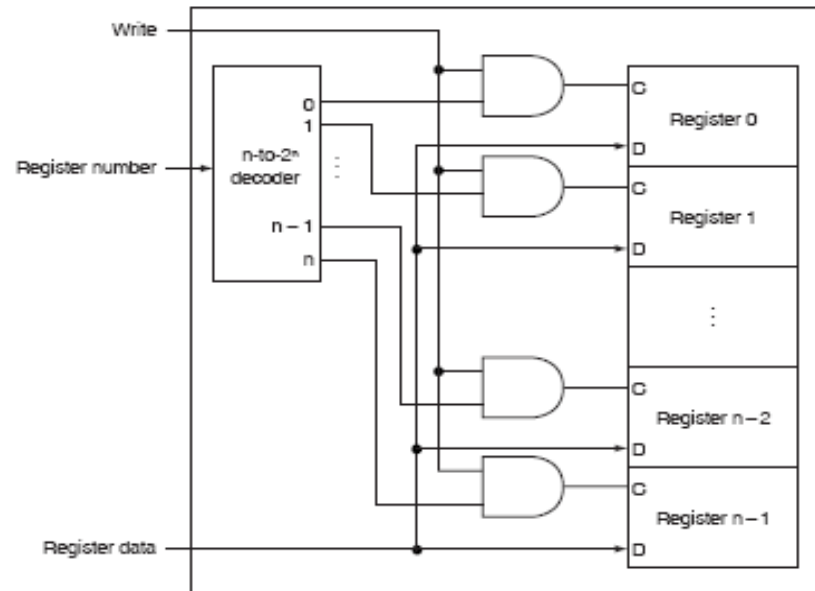
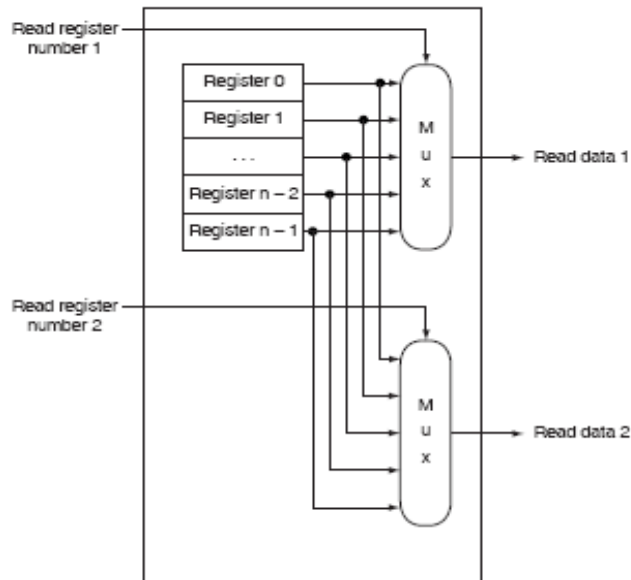
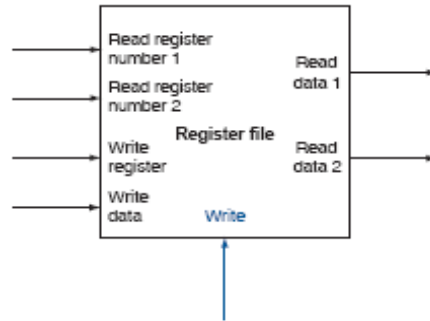
# The final 32-bit ALU



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

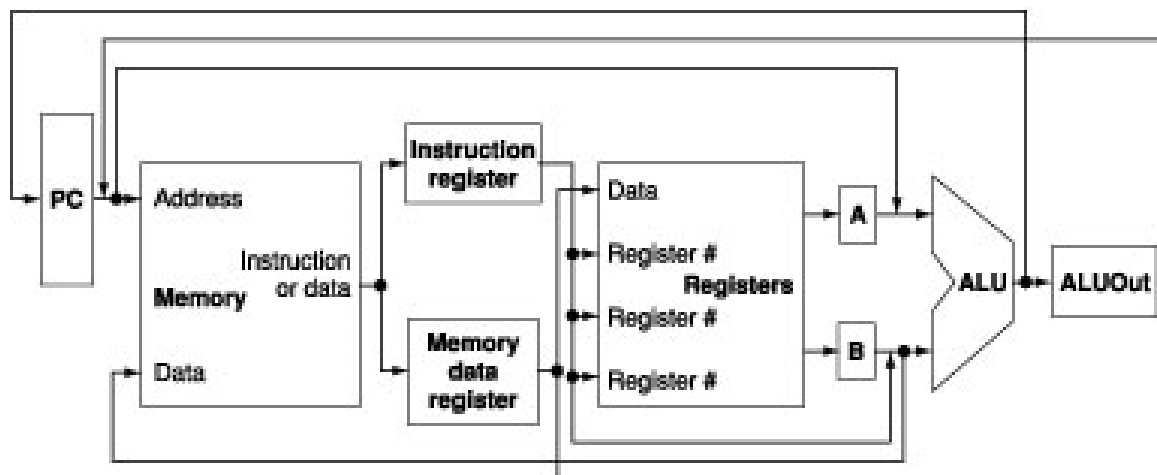


## B.8 Register Files



## 5.5 A Multicycle Implementation

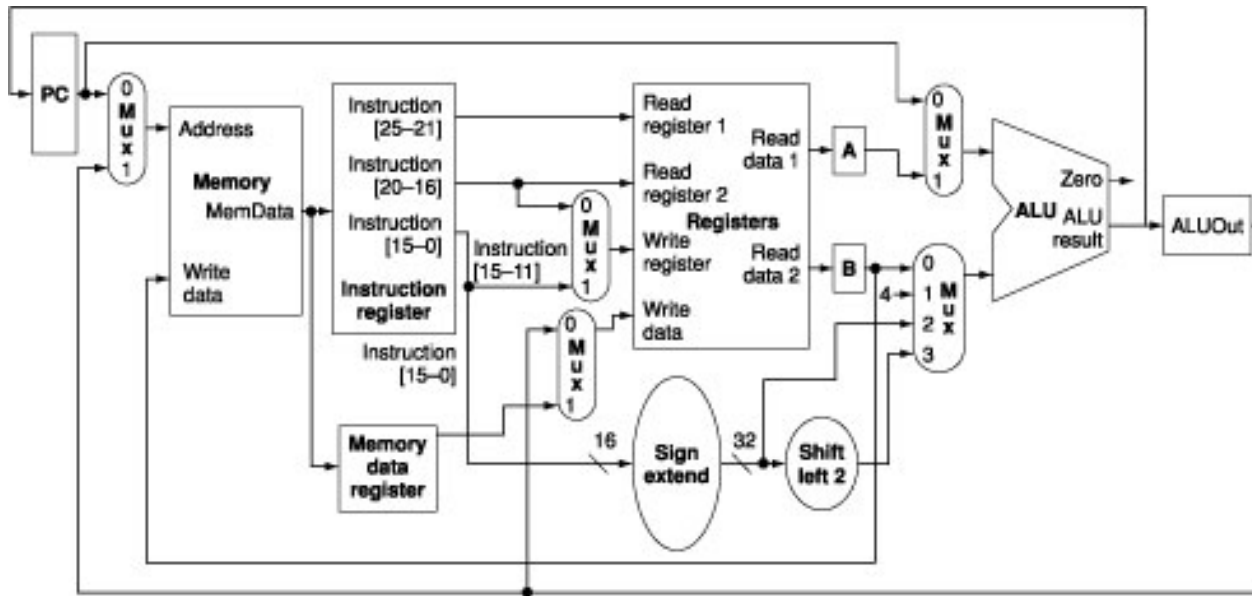
---



- A single memory unit is used for both instructions and data.
- There is a single ALU, rather than an ALU and two adders.
- One or more registers are added after every major functional unit.

# Continue

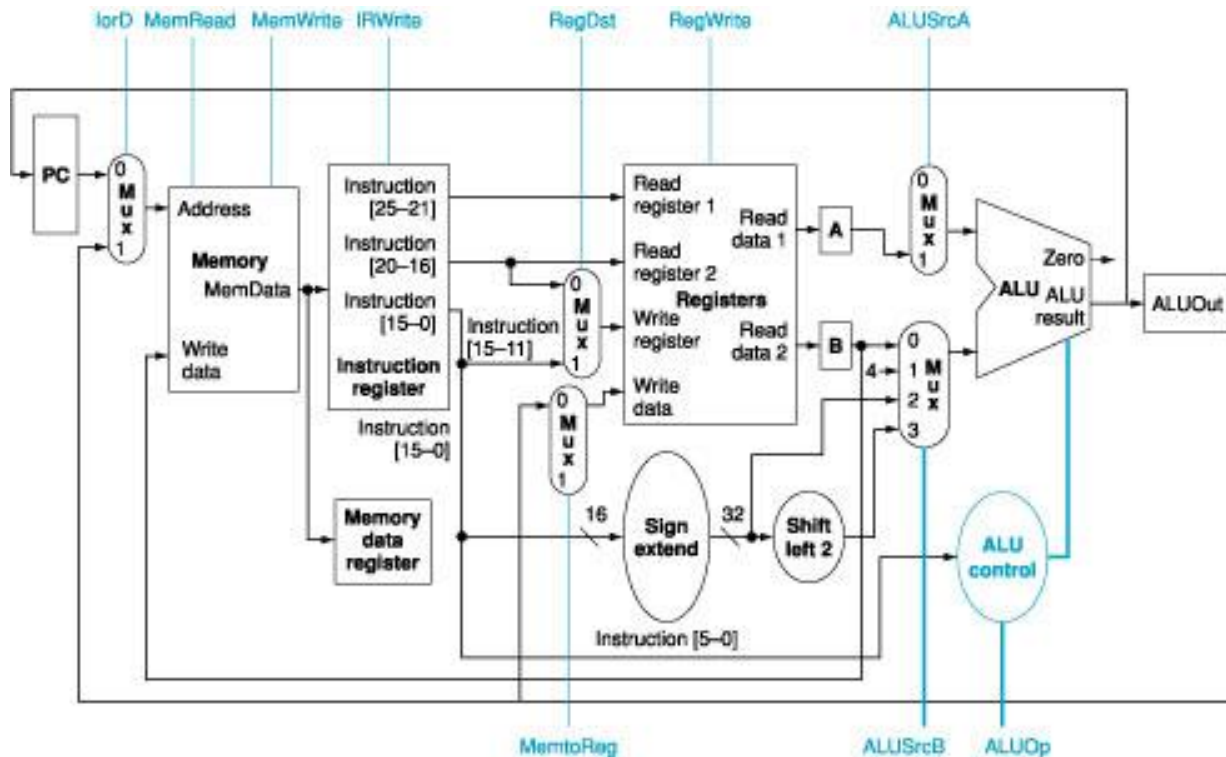
Replacing the three ALUs of the single-cycle by a single ALU means that the single ALU must accommodate all the inputs that used to go to the three different ALUs.



# Continue

## Control signals:

- The programmer-visible state units (PC, Memory, Register file) and IR → **write**  
Memory → **Read**
- ALU control: same as single cycle
- Multiplexor single/two control lines





# Continue

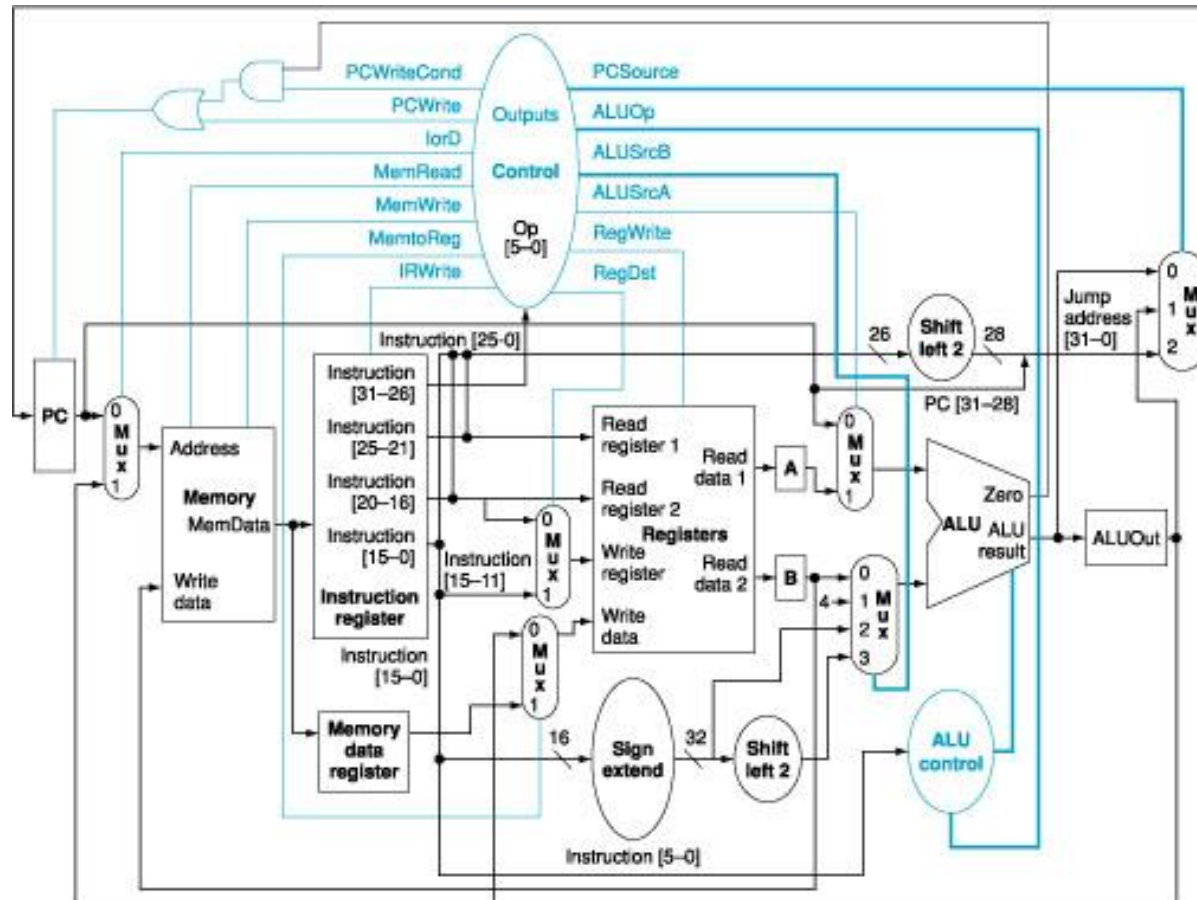
Three possible sources for the PC:

1. PC+4
2. ALUOut : address of the **beq**
3. Address for jump ( **j** )

PC write control signal:

PCWrite : PC+4 and **j** jump

PCWriteCond : **beq**



Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
lorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSrc.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

## Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSrc	00	Output of the ALU ( $PC + 4$ ) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address ( $IR[25:0]$ shifted left 2 bits and concatenated with $PC + 4[31:28]$ ) is sent to the PC for writing.

**FIGURE 5.29** The action caused by the setting of each control signal in Figure 5.28 on page 323. The top table describes the 1-bit control signals, while the bottom table describes the 2-bit signals. Only those control lines that affect multiplexors have an action when they are deasserted. This information is similar to that in Figure 5.16 on page 306 for the single-cycle datapath, but adds several new control lines (IRWrite, PCWrite, PCWriteCond, ALUSrcB, and PCSrc) and removes control lines that are no longer used or have been replaced (PCSrc, Branch, and Jump).

## Breaking the Instruction Execution into Clock Cycles

```
IR <= Memory[PC];
```

# IRWrite

**lorD = 0**

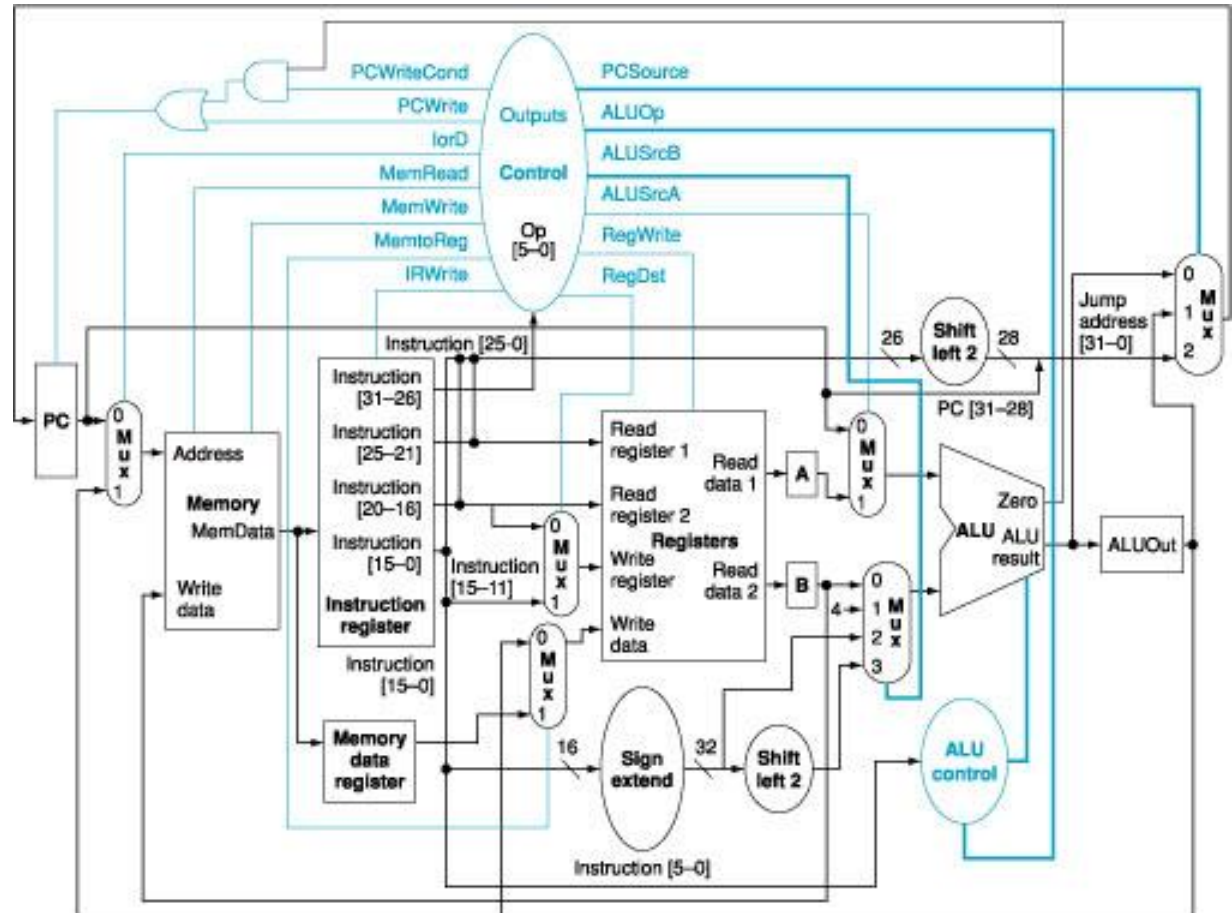
```
PC <= PC + 4;
```

## ALUSrcB = 01

**ALUOp = 00 (for add)**

**PCSource = 00**

# PCWrite



## The increment of the PC and instruction memory access can occur in parallel, how?

# Breaking the Instruction Execution into Clock Cycles

---

## 2. Instruction decode and register fetch step

- Actions that are **either** applicable to all instructions
- **Or** are not harmful

```
A <= Reg[IR[25:21]];
```

```
B <= Reg[IR[20:16]];
```

```
ALUOut <= PC + (sign-extend(IR[15:0] << 2));
```

## 2. Instruction decode and register fetch step

```
A <= Reg[IR[25:21]];
```

```
B <= Reg[IR[20:16]];
```

```
ALUOut <= PC + (sign-extend(IR[15-0] << 2 ));
```

```
A <= Reg[IR[25:21]];
```

```
B <= Reg[IR[20:16]] ;
```

**Since A and B are overwritten on every cycle → Done**

```
ALUOut <= PC + (sign-  
    extend(IR[15-0]<<2);
```

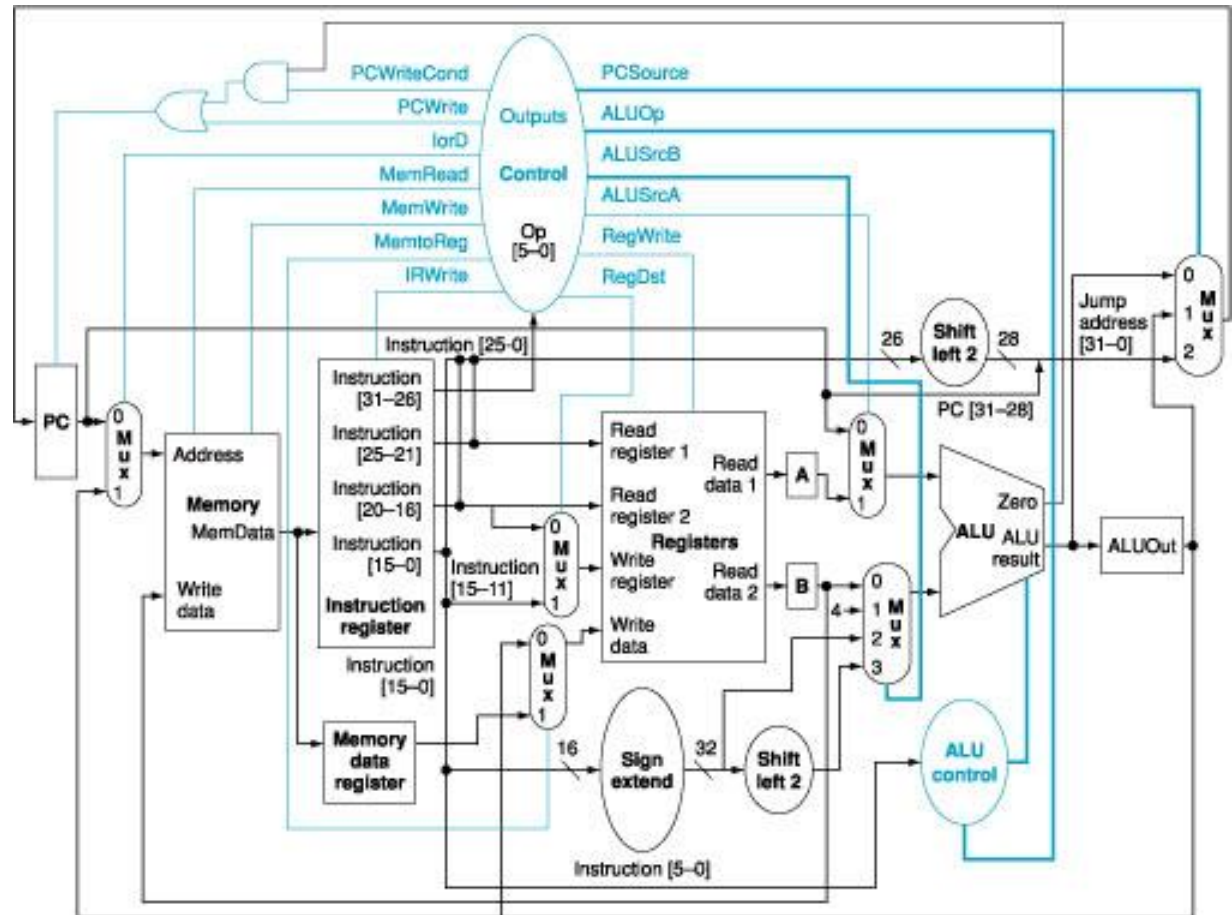
## This requires:

**ALUSrcA → 0**

**ALUSrcB → 11**

**ALUOp** → 00 (for add)

branch target address will be stored in **ALUOut**.



**The register file access and computation of branch target occur in parallel.**

# Breaking the Instruction Execution into Clock Cycles

---

## 3. Execution, memory address computation, or branch completion

### *Memory reference:*

```
ALUOut <= A + sign-extend(IR[15:0]);
```

### *Arithmetic-logical instruction:*

```
ALUOut <= A op B;
```

### *Branch:*

```
if (A == B) PC <= ALUOut;
```

### *Jump:*

```
PC <= { PC[31:28], (IR[25:0], 2'b00) };
```



### 3. Execution, memory address computation, or branch completion

#### Memory reference:

$ALUOut \leq A + \text{sign-extend}(IR[15:0]);$

$ALUSrcA = 1 \ \&\& \ ALUSrcB = 10$

$ALUOp = 00$

#### Arithmetic-logical instruction:

$ALUOut \leq A \ op \ B;$

$ALUSrcA = 1 \ \&\& \ ALUSrcB = 00$

$ALUOp = 10$

#### Branch:

$\text{if } (A == B) \ PC \leq ALUOut;$

$ALUSrcA = 1 \ \&\& \ ALUSrcB = 00$

$ALUOp = 01 \ (\text{for subtraction})$

$PCSource = 01$

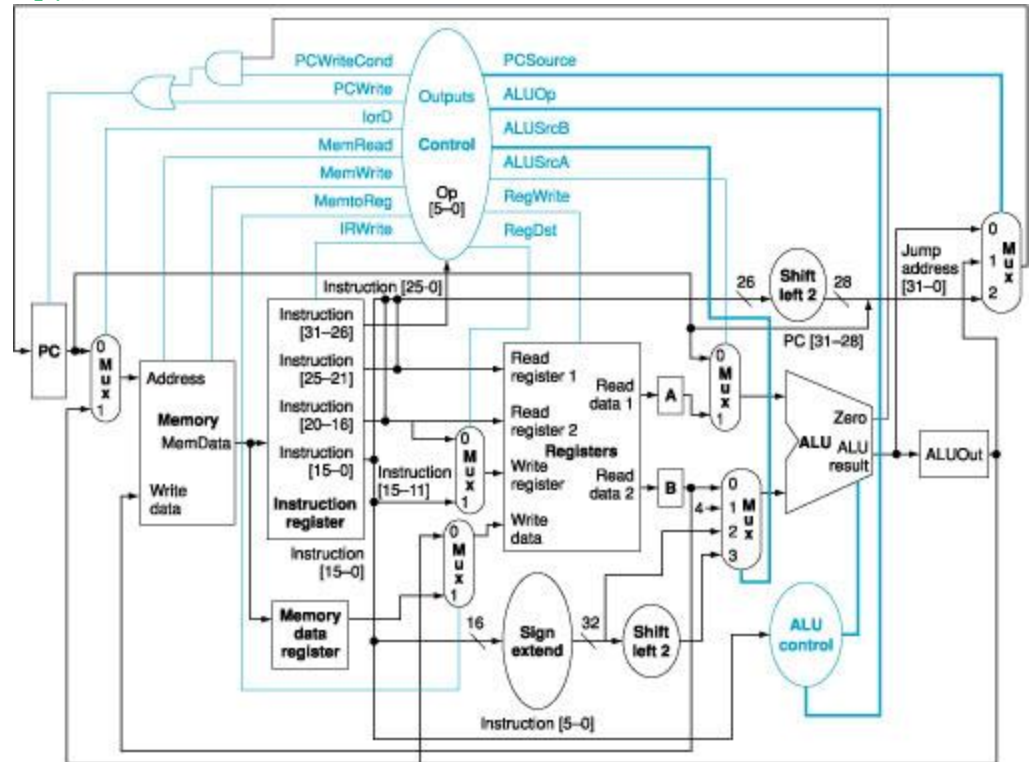
$PCWriteCond$

#### Jump:

$PC \leq \{ PC[31:28], (IR[25:0], 2'b00) \};$

$PCSource = 10$

$PCWrite$



# Breaking the Instruction Execution into Clock Cycles

---

## 4. Memory access or R-type instruction completion step

*Memory reference:*

MDR <= Memory [ALUOut];	MemRead	} lrd=1
or		
Memory [ALUOut] <= B;	MemWrite	

*Arithmetic-logical instruction (R-type):*

Reg[IR[15:11]] <= ALUOut;	RegDst=1 RegWrite
	MemtoReg=0

## 5. Memory read completion step

*Load:*

Reg[IR[20:16]] <= MDR;	MemtoReg=1 RegWrite
RegDst=0	



# Breaking the Instruction Execution into Clock Cycles

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$IR \leftarrow \text{Memory}[PC]$ $PC \leftarrow PC + 4$			
Instruction decode/register fetch	$A \leftarrow \text{Reg}[IR[25:21]]$ $B \leftarrow \text{Reg}[IR[20:16]]$ $ALUOut \leftarrow PC + (\text{sign-extend}(IR[15:0]) \ll 2)$			
Execution, address computation, branch/jump completion	$ALUOut \leftarrow A \text{ op } B$	$ALUOut \leftarrow A + \text{sign-extend}(IR[15:0])$	if $(A == B)$ $PC \leftarrow ALUOut$	$PC \leftarrow \{PC[31:28], (IR[25:0], 2'b00)\}$
Memory access or R-type completion	$\text{Reg}[IR[15:11]] \leftarrow ALUOut$	Load: $MDR \leftarrow \text{Memory}[ALUOut]$ or Store: $\text{Memory}[ALUOut] \leftarrow B$		
Memory read completion		Load: $\text{Reg}[IR[20:16]] \leftarrow MDR$		

# Defining the Control

---

Two different techniques to design the control:

- Finite state machine
- Microprogramming

## Example: CPI in a Multicycle CPU

Using the SPECINT2000 instruction mix, which is: 25% **load**, 10% **store**, 11% **branches**, 2% **jumps**, and 52% **ALU**.

What is the CPI, assuming that each state in the multicycle CPU requires 1 clock cycle?

## Answer:

The number of clock cycles for each instruction class is the following:

- Load: 5
- Stores: 4
- ALU instruction: 4
- Branches: 3
- Jumps: 3

# Example Continue

---

The CPI is given by the following:

$$CPI = \frac{CPU\ clock\ cycles}{Instruction\ count} = \frac{\sum Instruction\ count_i \times CPI_i}{Instruction\ count}$$

$$CPI = \sum \frac{Instruction\ count_i}{Instruction\ count} \times CPI_i$$

The ratio

$$\frac{Instruction\ count_i}{Instruction\ count}$$

is simply the instruction frequency for the instruction class i. We can therefore substitute to obtain:

$$CPI = 0.25 \times 5 + 0.10 \times 4 + 0.52 \times 4 + 0.11 \times 3 + 0.02 \times 3 = 4.12$$

This CPI is better than the worst-case CPI of 5.0 when all instructions take the same number of clock cycles.