

# Computer Design Basics – Part B

Instruction cycle, the Control Word

**Gloria Beraldo** (gloria.beraldo@unipd.it)

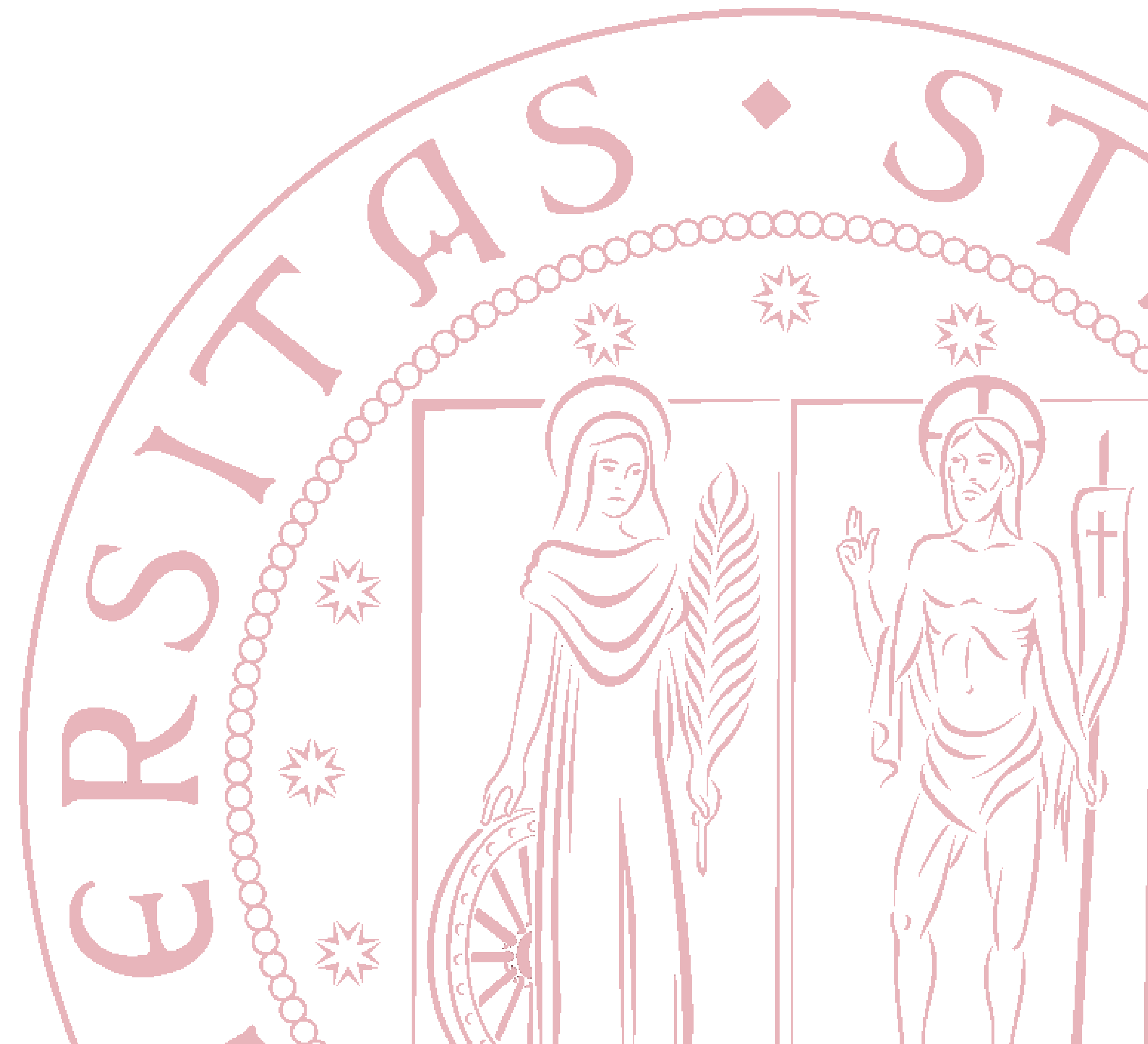
Department of Information Engineering, University of Padova

## Topics:

- Control and microoperations
- Instruction Set Architecture
- Single-cycle computer

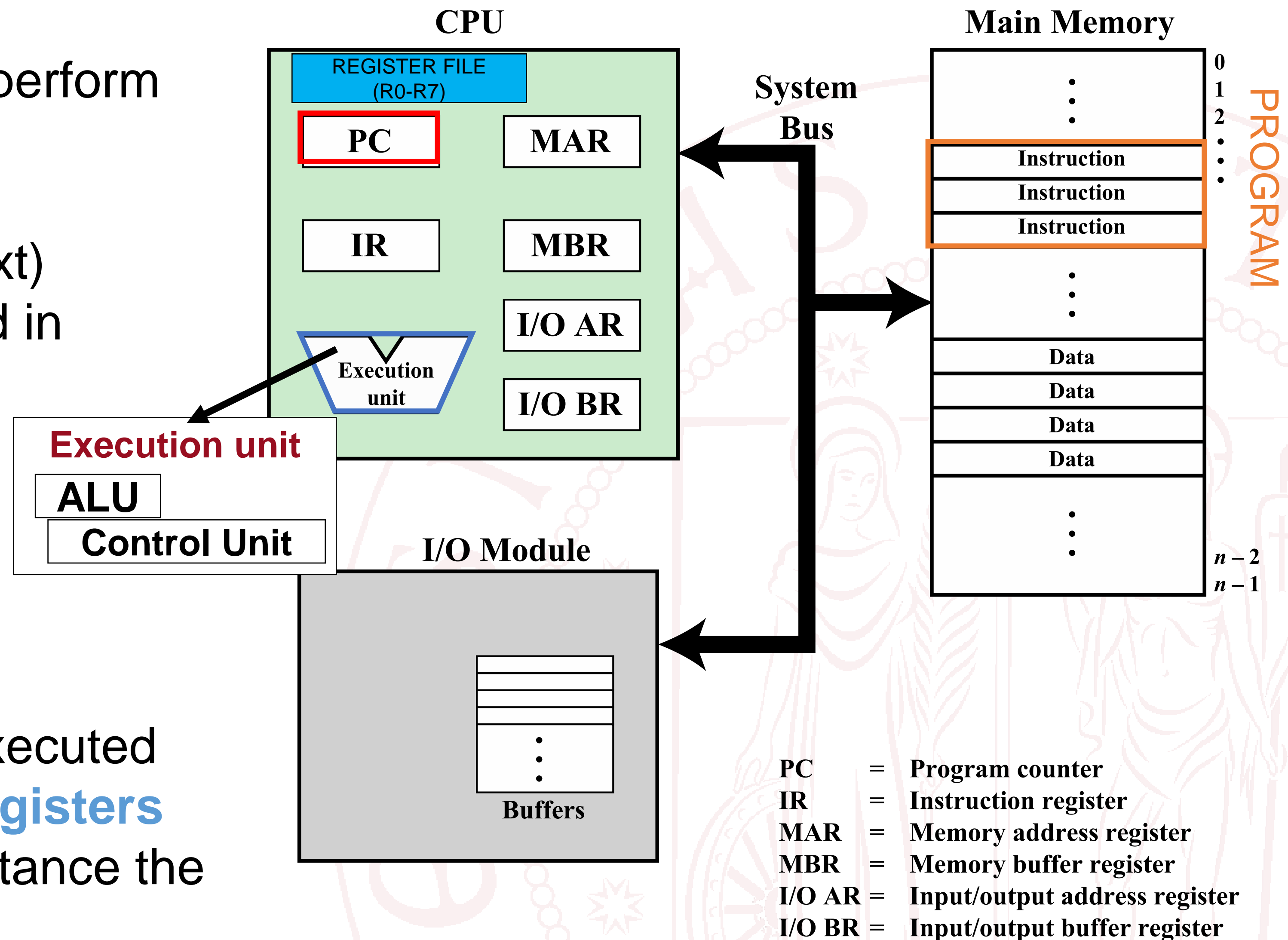
## Book Reference:

- Chapter 8



# Recap (I)

- A program is a sequence of instructions
- **Each instruction specifies:**
  - an operation the system has to perform
  - which operands to use
  - where to place the results
  - (which instruction to execute next)
- The **instructions** are usually stored in memory and they are loaded in sequence into registers.
- To execute the instructions in sequence, the register called **Program Counter (PC)** provides the address in memory of the instruction to be executed
- There is a set of **other general registers** to store data (e.g., R0-R7), for instance the operands



# Recap (II)

- Structure of the ALU

TABLE 8-2  
Function Table for ALU

Operation Select				Operation	Function
S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>	C <sub>in</sub>		
0	0	0	0	$G = A$	Transfer A
0	0	0	1	$G = A + 1$	Increment A
0	0	1	0	$G = A + B$	Addition
0	0	1	1	$G = A + B + 1$	Add with carry input of 1
0	1	0	0	$G = A + \overline{B}$	A plus 1s complement of B
0	1	0	1	$G = A + \overline{B} + 1$	Subtraction
0	1	1	0	$G = A - 1$	Decrement A
0	1	1	1	$G = A$	Transfer A
1	X	0	0	$G = A \wedge B$	AND
1	X	0	1	$G = A \vee B$	OR
1	X	1	0	$G = A \oplus B$	XOR
1	X	1	1	$G = \overline{A}$	NOT (1s complement)

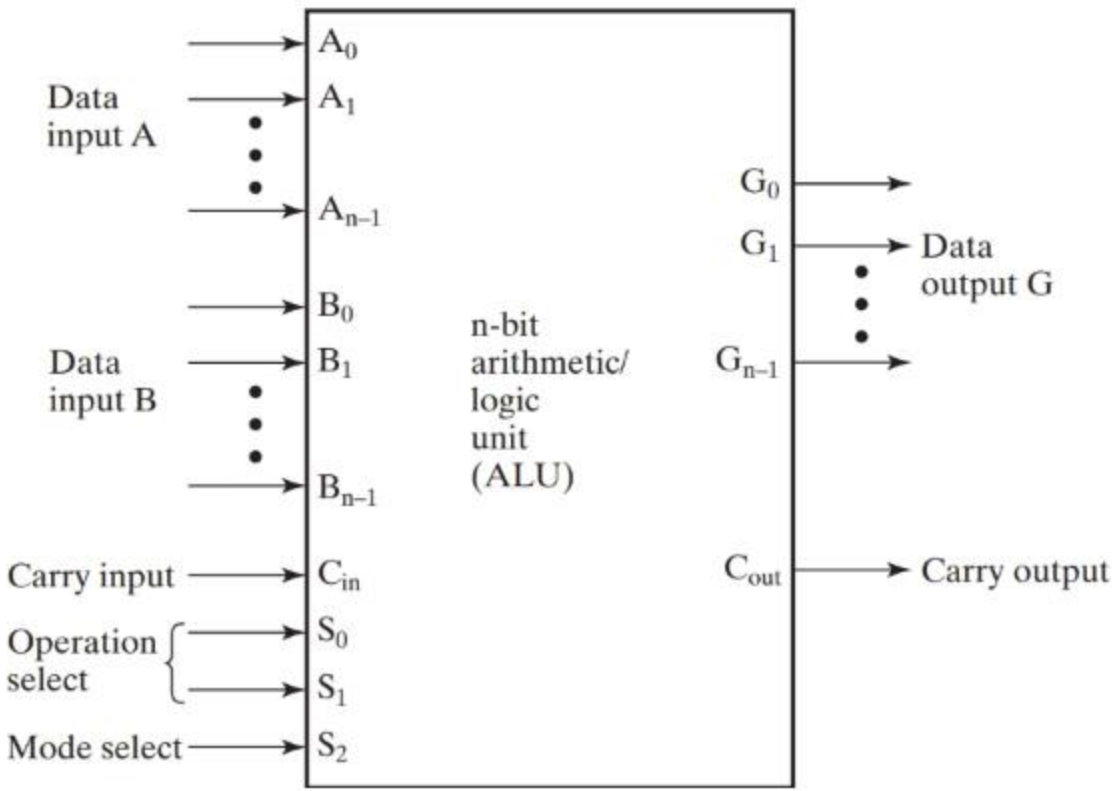


FIGURE 8-2  
Symbol for an  $n$ -Bit ALU

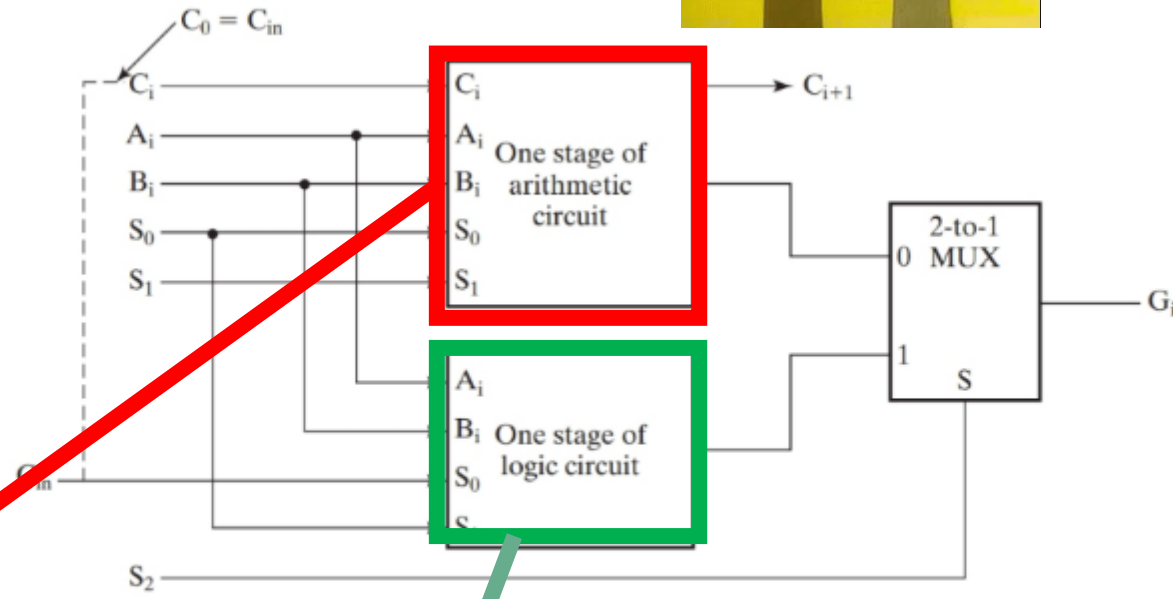


FIGURE 8-7  
One Stage of ALU

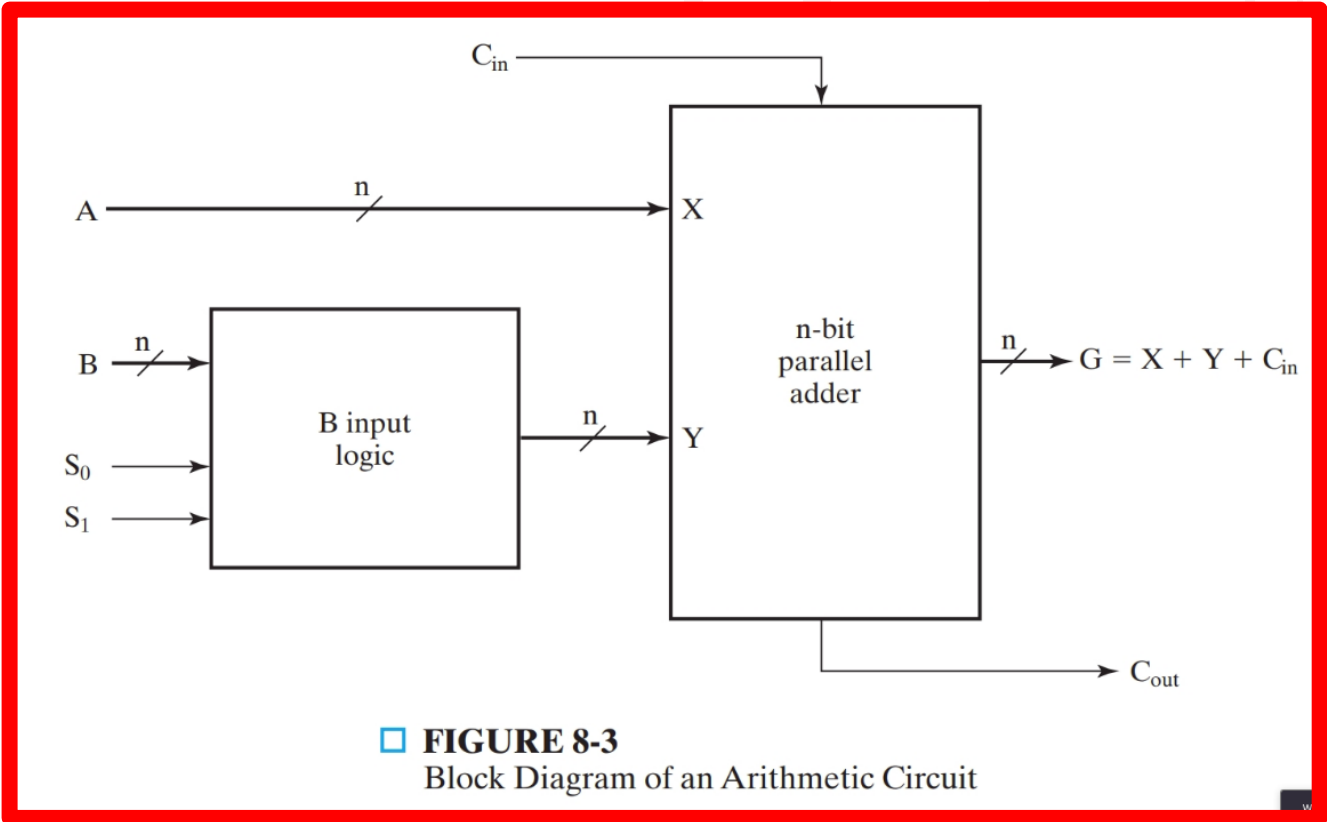


FIGURE 8-3  
Block Diagram of an Arithmetic Circuit

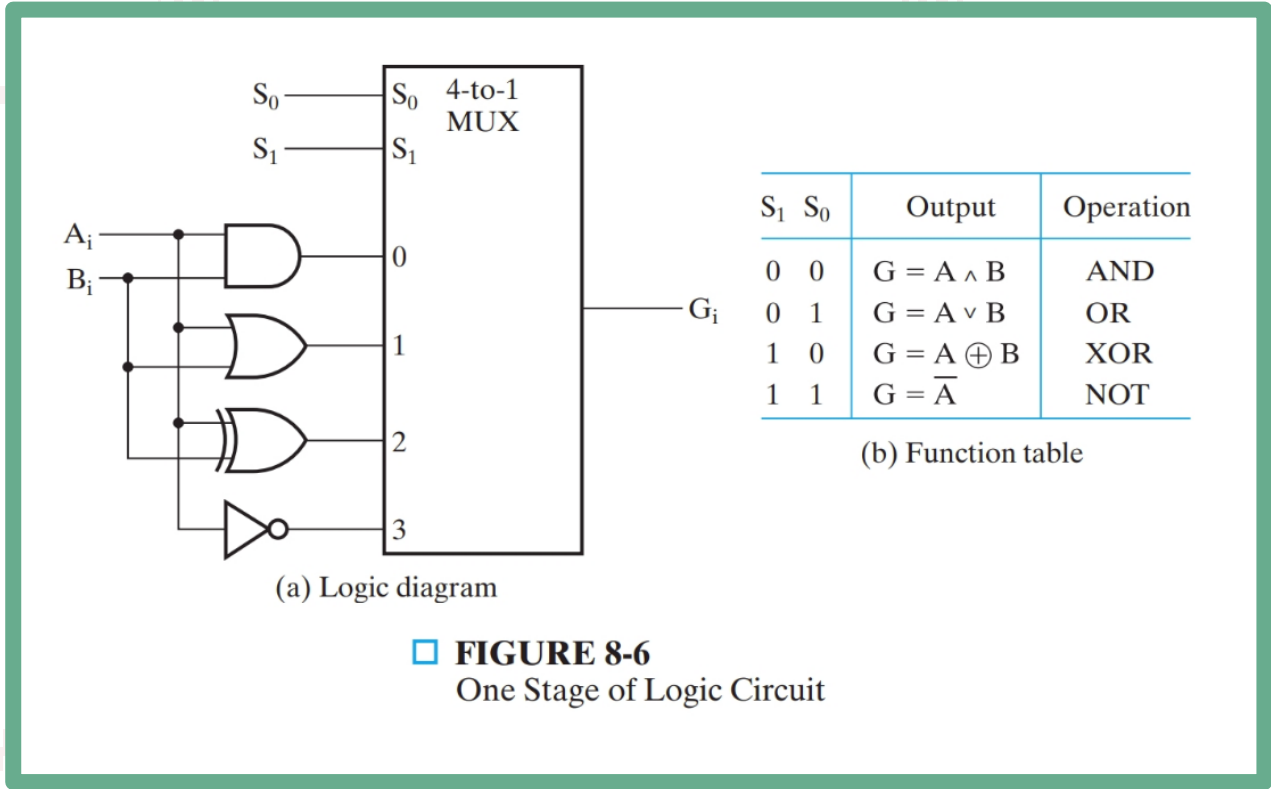


FIGURE 8-6  
One Stage of Logic Circuit

# Recap (III)

- Structure of the ALU
- Structure of the Shifter

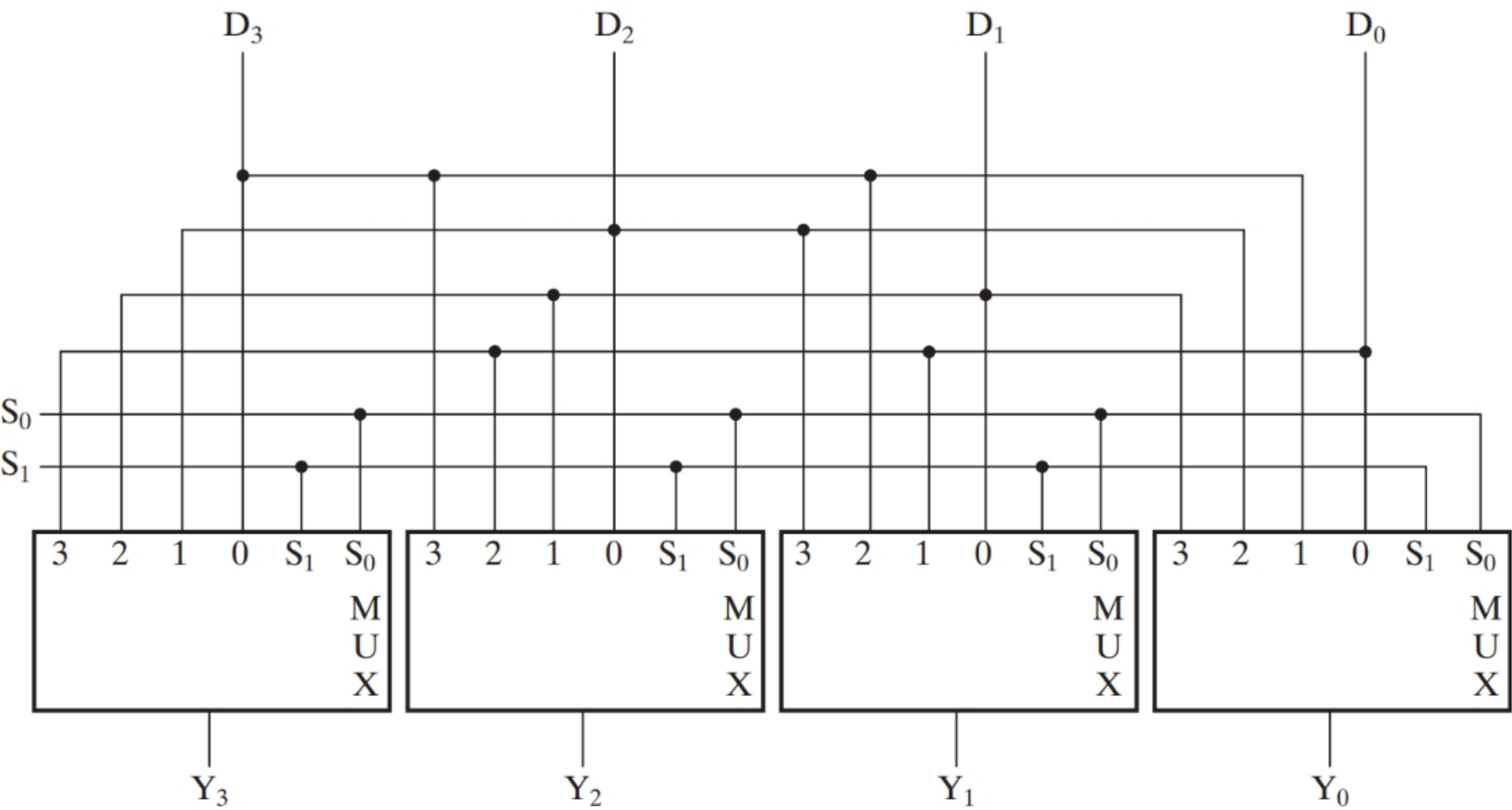


FIGURE 8-9  
4-Bit Barrel Shifter

It is often useful to shift the data by  $n$  bits positions in datapath applications

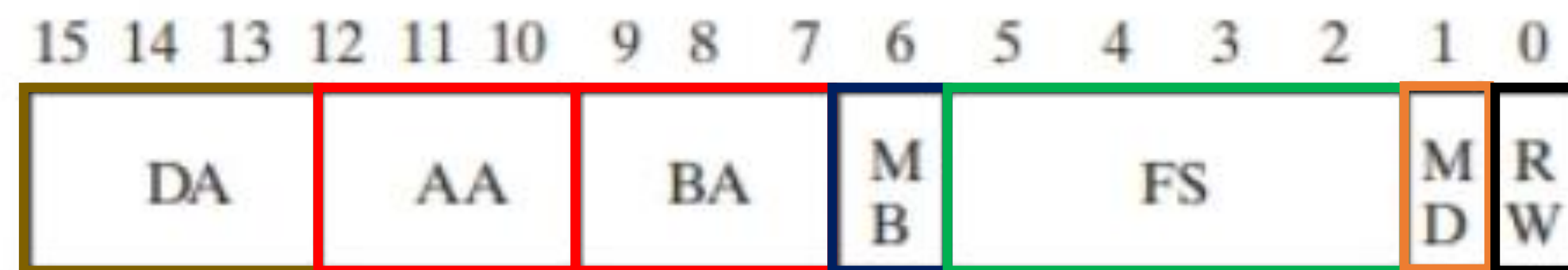
TABLE 8-3  
Function Table for 4-Bit Barrel Shifter

Select		Output				Operation
S <sub>1</sub>	S <sub>0</sub>	Y <sub>3</sub>	Y <sub>2</sub>	Y <sub>1</sub>	Y <sub>0</sub>	
0	0	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	No rotation
0	1	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>	D <sub>3</sub>	Rotate one position
1	0	D <sub>1</sub>	D <sub>0</sub>	D <sub>3</sub>	D <sub>2</sub>	Rotate two positions
1	1	D <sub>0</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	Rotate three positions

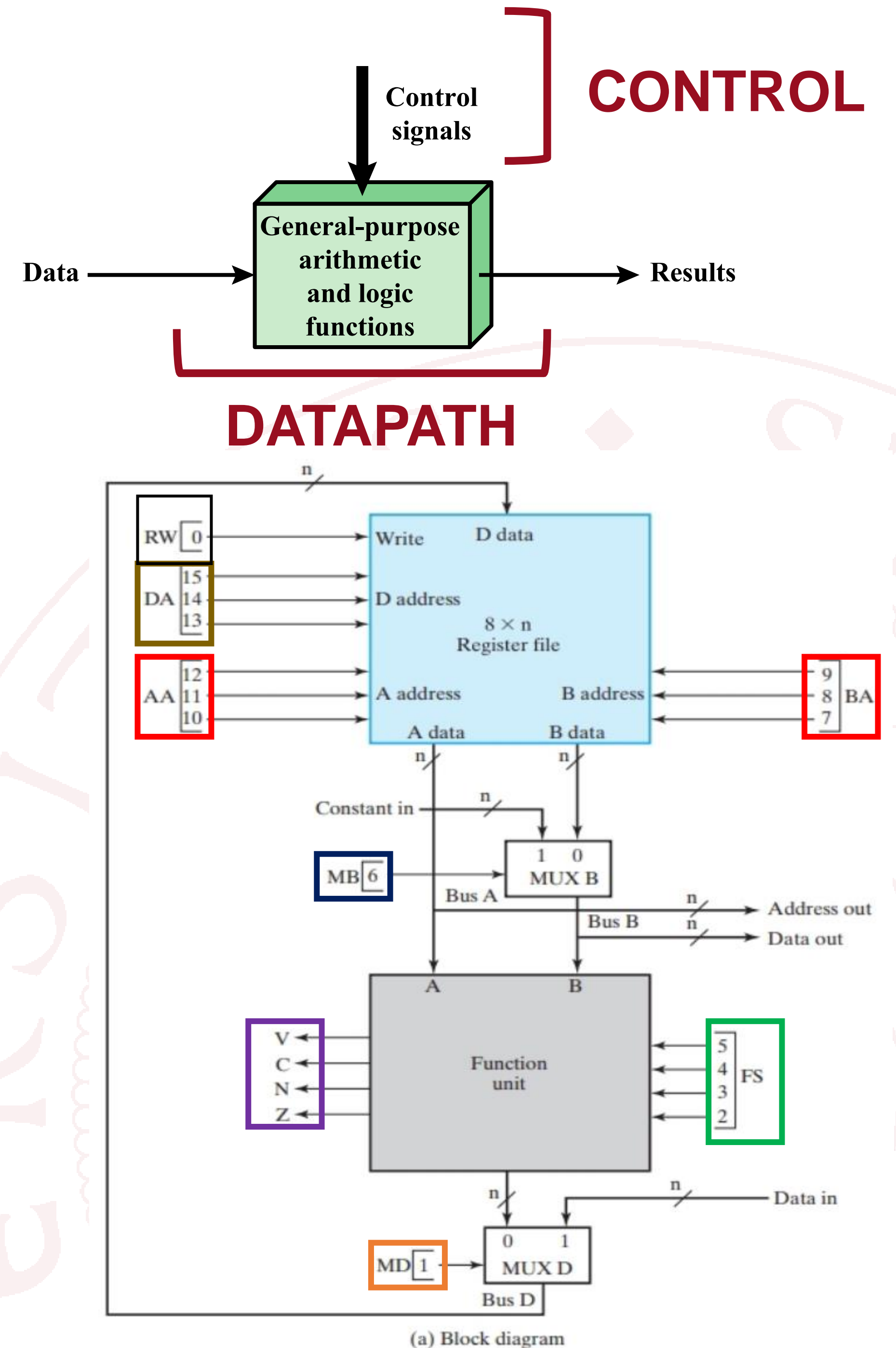


# Recap (IV)

- Registers in the microprocessor
- Block Diagram of a Generic Datapath
- Function of the control word
- Format of the control word

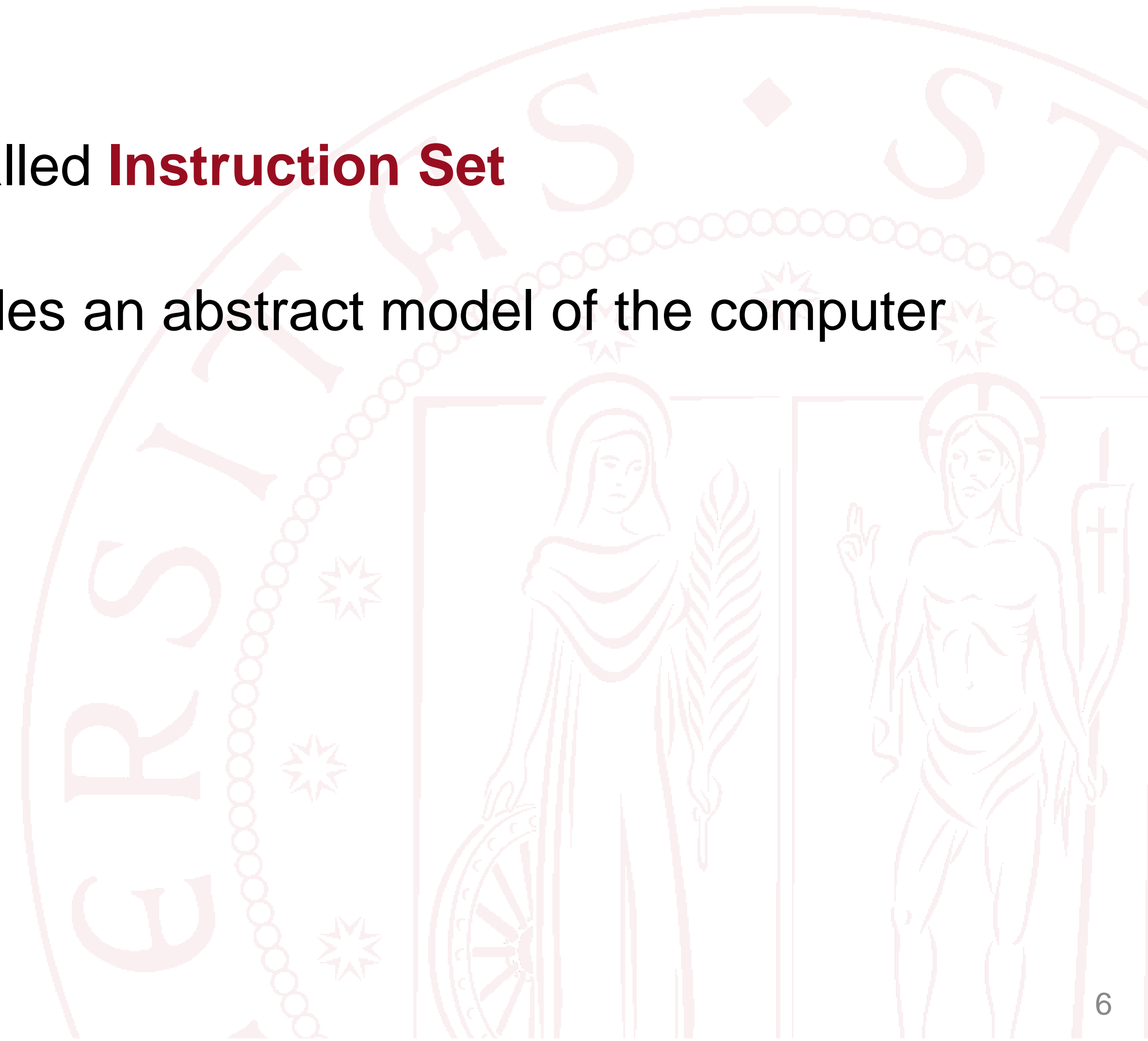


(b) Control word



# Instruction Set Architecture (ISA)

- An *instruction* is a collection of bits that instructs the computer to perform a specific operation.
- The collection of instructions for a computer is called **Instruction Set**
- A detailed description of the instruction set provides an abstract model of the computer that is called **Instruction Set Architecture (ISA)**
- The simplest ISA has three major components:
  - Storage resources
  - Instruction formats
  - Instruction specifications

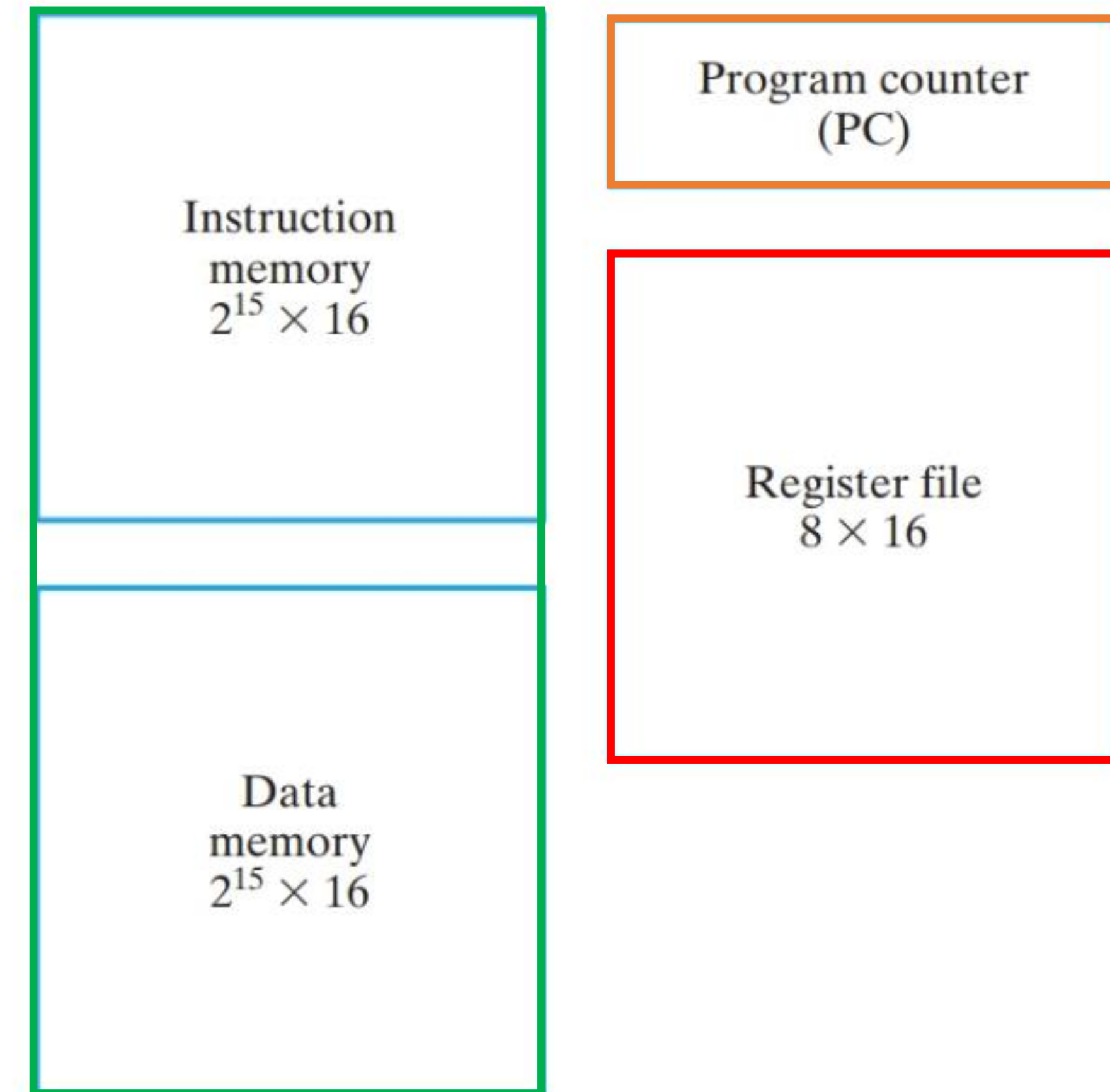


# ISA Storage Resources

# Storage resources

## Resource available to the programmer:

- **Instruction memory:** where the instructions are stored
- **Data memory:** where data are stored
- **Register file:** registers where the data are saved inside the processor
- **Program counter:** provides the address in memory of the instruction to be executed

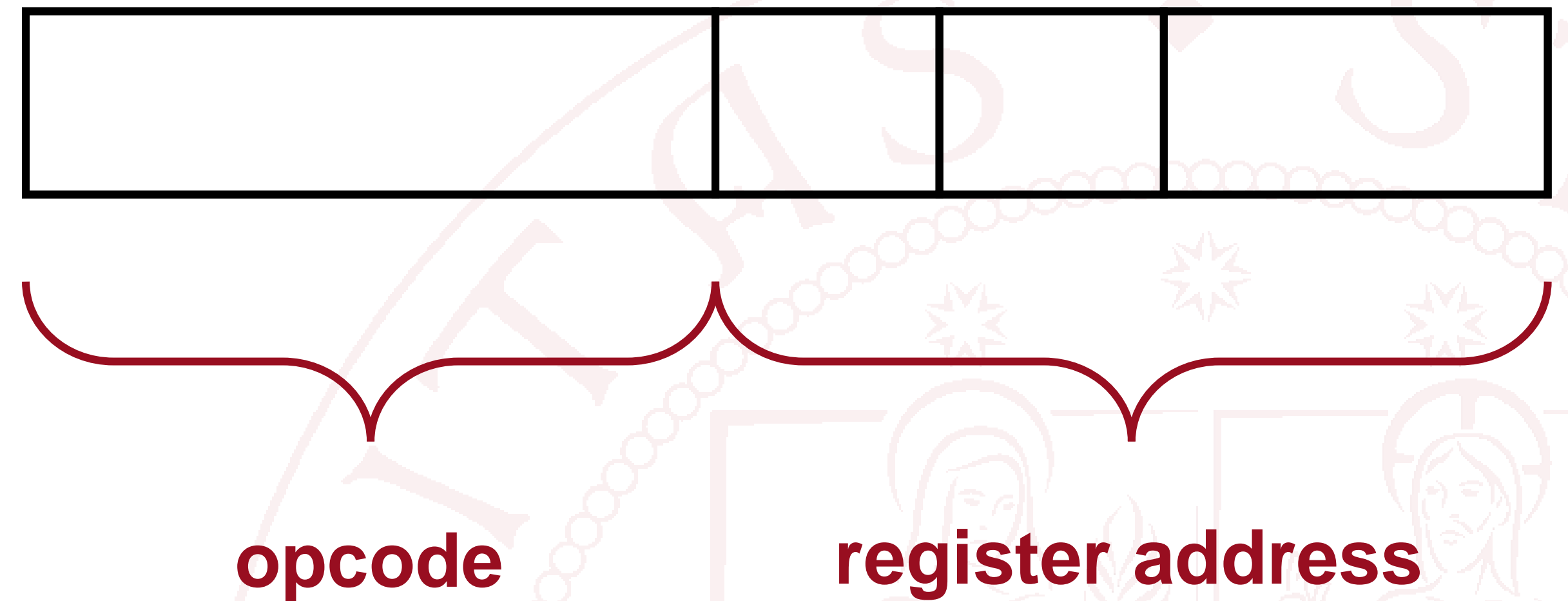


□ **FIGURE 8-13**  
Storage Resource Diagram for a Simple Computer



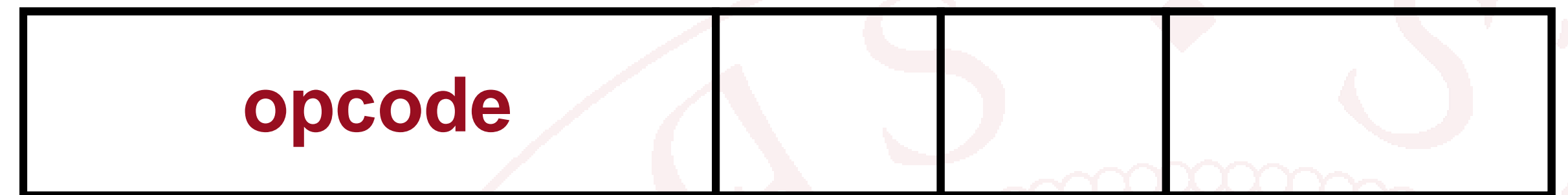
# Instruction formats

- The format of an instruction is usually depicted by a rectangular box symbolizing the bits of the instruction
- The representation of the bits is as they appear in memory words or in a control register
- The instruction is divided into fields that specify:
  - The operation code (opcode)
  - Register file address



# Instruction formats: operation code (opcode)

- The opcode is a group of bits in the instruction that specifies the operation to be performed (e.g., add, sub, shift..)
- The number of bits required for the opcode of an instruction depends on the total number of distinct operations in the instruction set ( $m$  bits for up to  $2^m$  distinct operations)
- The designer assigns a bit combination (code) to each operation.



For example:

- Processor with 128 distinct operations
- $2^m = 128 \rightarrow m = 7 \rightarrow$  opcode with 7 bit
- opcode: 0000010
- When the opcode is detected by the control unit, a sequence of control words is applied to the datapath

# Instruction formats: register addresses, operands

- The operation must be performed using data stored in computer registers or in a memory
- An instruction, therefore, must specify not only the operation, but also the **operands** and where the **result** has to be placed
- The operands may be specified in two ways:
  - **explicitly**, if the instruction contains special bits for its identification
  - **implicitly**, if it is included as a part of the definition of the operation itself (represented by the opcode)



## Explicit operands

For instance, the instruction performing an addition ( $R1 \leftarrow A + B$ ) may contain:

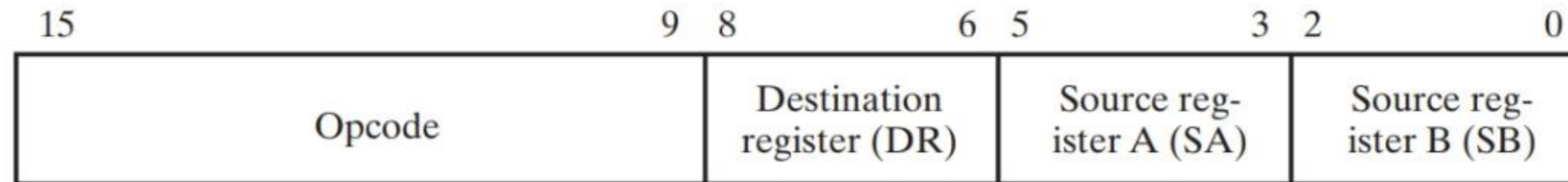
- 2 registers for the two operands ( $A, B$ )
- 1 register for the result ( $R1$ )

## Implicit operands

For instance, in an Increment Register operation ( $R7 \leftarrow R7 + 1$ ), one of the operands is *implicitly*  $+ 1$

# Instruction formats (1): on registers

- 8 registers: R0-R7
- 16 bits per instruction



(a) Register

- **opcode**: specifies the use of three or fewer registers as needed
- **Destination register (DR)**: destination for the output
- **Source register A (SA)**: operand A
- **Source register B (SB)**: operand B

Example 1:

- opcode identifies a subtract
- SA: 010 (R2); SB: 011 (R3); DR: 001 (R1)
- $R1 \leftarrow R2 - R3$

Example 2:

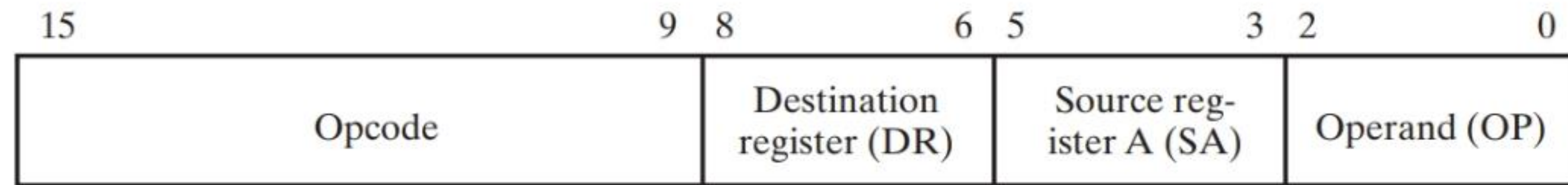
- opcode identifies a store in memory
- SA: 100 (R4); SB: 101 (R5); DR: XXX
- $M[R4] \leftarrow R5$
- It is worth noticing the use of  $M[ ]$  to identify the address in memory

DR has no effect, because the store prevent the writing on the register (RW=0)



# Instruction formats (2): immediate operand

- 8 registers: R0-R7
- 16 bits per instruction



(b) Immediate

- **opcode**: specifies an operation on two registers
- **Destination register (DR)**: destination for the output
- **Source register A (SA)**: operand A
- **Operand (OP)**: immediate operand (immediately available in the instruction, not from the registers)

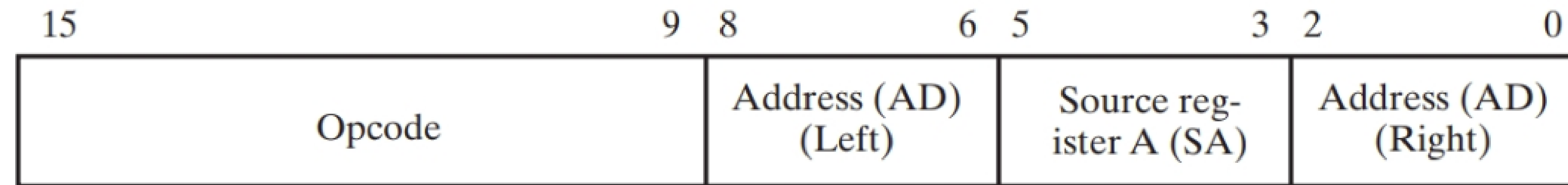
Example 1:

- opcode identifies an add immediate operation
- SA: 111 (R7); DR: 010 (R2); OP: 011 (value 3)
- $R2 \leftarrow R7 + 3$



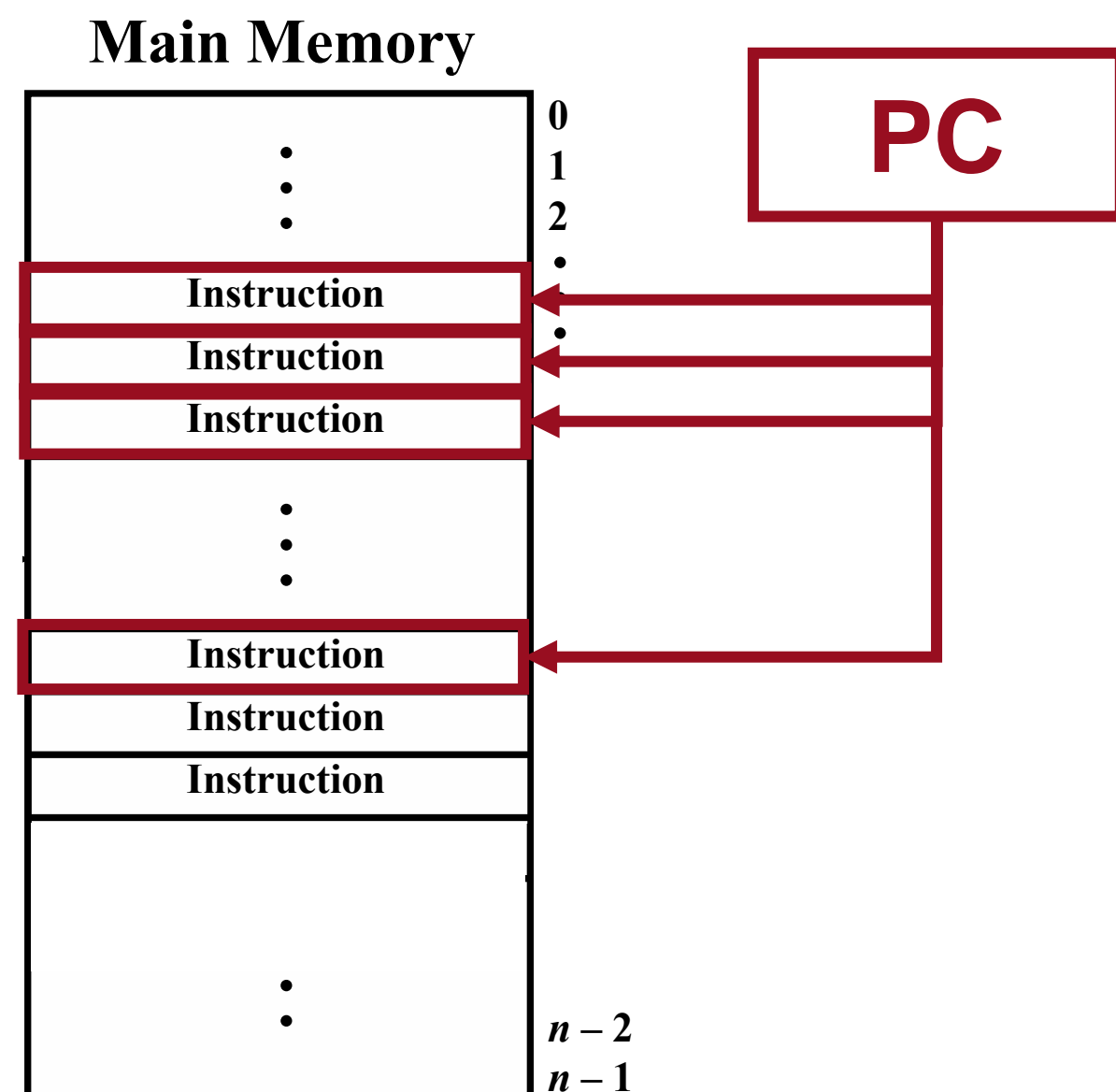
# Instruction formats (3): jump and branch

- 8 registers: R0-R7
- 16 bits per instruction



(c) Jump and branch

## What is a jump or a branch?



- The instructions are written in memory in sequence
- The location of an instruction to be fetched is determined by the PC (its value is updated as  $[PC+1]$  at each clock cycle)
- It is necessary sometimes to *change the order of execution* of the instructions based on the results of the processing performed (e.g., «*jump*» to the next instructions)
- Such changes in the order of instruction execution are called **jumps** or **branches**

# Jump and Branch (example)

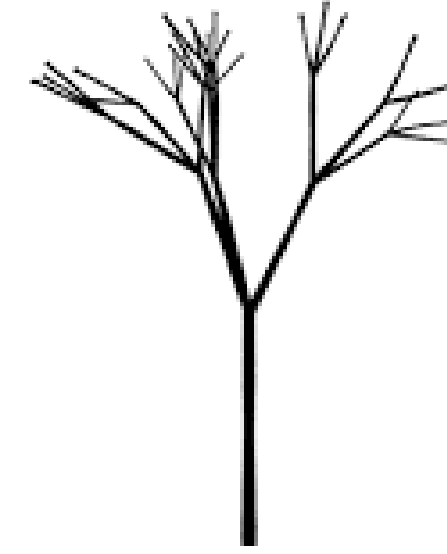
```
17 public class Test {  
18  
19  
20 public static void main( String[] args ) {  
21  
22     int year = 2023;  
23     String pippo = "Have a nice "+year;  
24  
25  
26  
27  
28  
29     System.out.println( pippo );  
30  
31 }  
32  
33 }
```

branch

jump

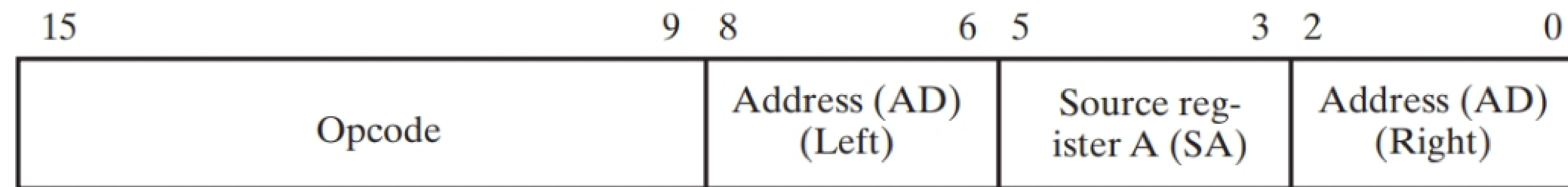
```
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035
```

```
/**  
 * Prints a String and then terminate the line. This method behaves as  
 * though it invokes {@link #print(String)} and then  
 * {@link #println()}.  
 *  
 * @param x The {@code String} to be printed.  
 */  
public void println(String x) {  
    if (getClass() == PrintStream.class) {  
        writeln(String.valueOf(x));  
    } else {  
        synchronized (this) {  
            print(x);  
            newLine();  
        }  
    }  
}
```



# Instruction formats (3): jump and branch

- 8 registers: R0-R7
- 16 bits per instruction



(c) Jump and branch

- **opcode**: specifies the jump or branch operation
- **Source register A (SA)**: operand A
- **Address (AD)**: split address (left and right) that identifies the destination for the branch

The addressing method is based on the **address offset** and it refers to the current value of the PC.

- The address offset is composed of 6 bits
- Address offset is treated as a signed 2s complement number
- A sign extension is applied to the PC to form a 16-bit offset before the addition

If the leftmost bit of the address field AD is a 1:  
The 10 bits to its left are filled with 1s to give a  
**negative 2s complement offset**

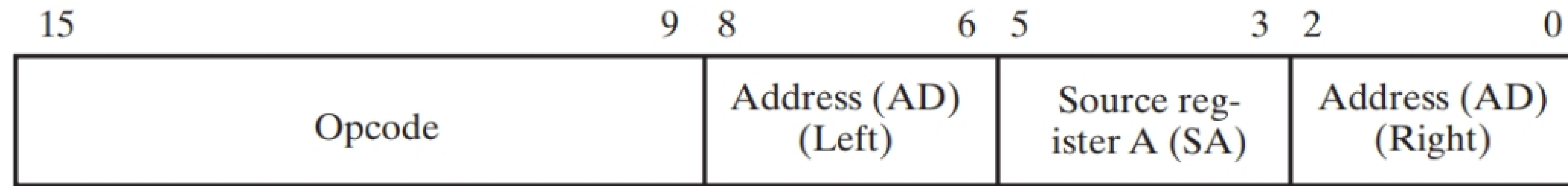
If the leftmost bit of the address field is 0:  
10 bits to its left are filled with 0s  
to give a **positive 2s complement offset**

**se**  
sign extension



# Instruction formats (3): jump and branch

- 8 registers: R0-R7
- 16 bits per instruction



(c) Jump and branch

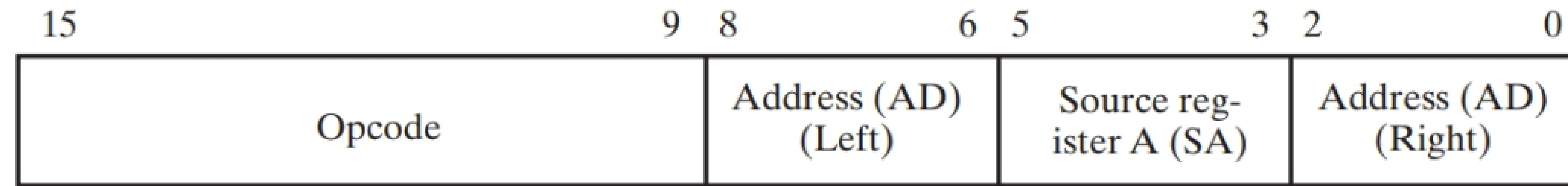
- **opcode**: specifies the jump or branch operation
- **Source register A (SA)**: operand A
- **Address (AD)**: split address (left and right) that identifies the destination for the branch

Example:

- PC = 55; Branch target address: 35
- The branch occurs if and only if the content of R6 is equal to zero
- Opcode specifies an instruction **branch-on-zero**
- SA: 110 (R6); AD (Left): 101; AD (Right): 100 → AD: 101100 (-20)
  - Se  $R6 = 0 \rightarrow PC = 55 + (-20) = 35$
  - Se  $R6 \neq 0 \rightarrow PC = 55 + 1 = 56$

# Instruction formats (3): jump and branch

- 8 registers: R0-R7
- 16 bits per instruction



(c) Jump and branch

- **opcode**: specifies the jump or branch operation
- **Source register A (SA)**: operand A
- **Address (AD)**: split address (left e right) that identifies the destination for the branch
- This addressing method alone provides a branch addresses **within a small range** below and above the PC value
- The jump provides a broader range of addresses by using the unsigned contents of a 16-bit register as the jump target
- The branch is conditional, the jump is not

# Instruction Specifications

- Describe each of the distinct instructions that can be executed by the system
- For each instruction, the **opcode** is given along with a short name called a **mnemonic**
- The mnemonic, along with a representation for each of the additional instruction fields in the format for the instruction, represents the notation to be used

Instruction	Mnemonic	Format	Description	Status bits
Addition	ADD	RD, RA, RB	$R[DR] \leftarrow R[SA] + R[SB]$	N,Z

- This symbolic representation is then converted to the binary representation of the instruction by a program called an **assembler**

# ISA


# Instruction Specifications



# Instruction Specifications

- Examples of instructions specifications for the Simple Computer
- The format of the instruction is specified (for example: ADD RD, RA, RB)
- The symbolic description: (for example:  $R[DR] \leftarrow R[SA] + R[SB]$ )
- The status bits related to the instuction (for example: N, Z)
- For all of the instructions with \*, PC is incremented by a unit for the next cycle ( $PC \leftarrow PC + 1$ )

TABLE 8-8  
Instruction Specifications for the Simple Computer

Instruction	Opcode	Mne- monic	Format	Description	Status Bits
Move A	0000000	MOVA	RD, RA	$R[DR] \leftarrow R[SA]^*$	N, Z
Increment	0000001	INC	RD, RA	$R[DR] \leftarrow R[SA] + 1^*$	N, Z
Add	0000010	ADD	RD, RA, RB	$R[DR] \leftarrow R[SA] + R[SB]^*$	N, Z
Subtract	0000101	SUB	RD, RA, RB	$R[DR] \leftarrow R[SA] - R[SB]^*$	N, Z
Decrement	0000110	DEC	RD, RA	$R[DR] \leftarrow R[SA] - 1^*$	N, Z
AND	0001000	AND	RD, RA, RB	$R[DR] \leftarrow R[SA] \wedge R[SB]^*$	N, Z
OR	0001001	OR	RD, RA, RB	$R[DR] \leftarrow R[SA] \vee R[SB]^*$	N, Z
Exclusive OR	0001010	XOR	RD, RA, RB	$R[DR] \leftarrow R[SA] \oplus R[SB]^*$	N, Z
NOT	0001011	NOT	RD, RA	$R[DR] \leftarrow \overline{R[SA]}^*$	N, Z
Move B	0001100	MOVB	RD, RB	$R[DR] \leftarrow R[SB]^*$	
Shift Right	0001101	SHR	RD, RB	$R[DR] \leftarrow sr\ R[SB]^*$	
Shift Left	0001110	SHL	RD, RB	$R[DR] \leftarrow sl\ R[SB]^*$	
Load Immediate	1001100	LDI	RD, OP	$R[DR] \leftarrow zf\ OP^*$	
Add Immediate	1000010	ADI	RD, RA, OP	$R[DR] \leftarrow R[SA] + zf\ OP^*$	N, Z
Load	0010000	LD	RD, RA	$R[DR] \leftarrow M[SA]^*$	
Store	0100000	ST	RA, RB	$M[SA] \leftarrow R[SB]^*$	
Branch on Zero	1100000	BRZ	RA, AD	if ( $R[SA] = 0$ ) $PC \leftarrow PC + se\ AD$ , N, Z if ( $R[SA] \neq 0$ ) $PC \leftarrow PC + 1$	
Branch on Negative	1100001	BRN	RA, AD	if ( $R[SA] < 0$ ) $PC \leftarrow PC + se\ AD$ , N, Z if ( $R[SA] \geq 0$ ) $PC \leftarrow PC + 1$	
Jump	1110000	JMP	RA	$PC \leftarrow R[SA]$  <b>ERROR!</b>	

\* For all of these instructions,  $PC \leftarrow PC + 1$  is also executed to prepare for the next cycle. **no PC+1 after JUMP !!**



# Instruction Specifications: execution

- Memory with 16 bits
- Instructions and data are placed in memory
- Four instructions:
  1. **Subtract**  
(opcode: 5 [0000101])
  2. **Store**  
(opcode: 32 [01000000])
  3. **Add immediate** (opcode: 66 [1000010])
  4. **Branch on zero**  
(opcode: 96 [11000000])

TABLE 8-9  
Memory Representation of Instructions and Data

Decimal Address	Memory Contents	Decimal Opcode	Other Fields	Operation
25	000010100101011	5 (Subtract)	DR:1, SA:2, SB:3	R1 ← R2 - R3
35	0100000000100101	32 (Store)	SA:4, SB:5	M[R4] ← R5
45	1000010010111011	66 (Add Immediate)	DR:2, SA:7, OP:3	R2 ← R7 + 3
55	1100000101110100	96 (Branch on Zero)	AD: 44, SA:6	If R6 = 0, PC ← PC - 20
70	00000000011000000	Data = 192. After execution of instruction in 35, Data = 80.		

Addresses in memory where the instructions are stored

# Instruction Specifications: execution

1. Address 25:  $R1 \leftarrow R2 - R3$

0000101 001 010 011

**SUB**     **R1, R2, R3**

DR:1 SA:2 SB:3

2. Address 35:  $M[R4] \leftarrow R5$

0100000 000 100 101

**ST**     **R4, R5**

SA:4 SB:5

3. Address 45:  $R2 \leftarrow R7 + 3$

1000010 010 111 011

**ADI**     **R2, R7, 3**

DR:2 SA:7 OP:3

In memory

TABLE 8-9  
Memory Representation of Instructions and Data

Decimal Address	Memory Contents	Decimal Opcode	Other Fields	Operation
25	0000101 001 010 011	5 (Subtract)	DR:1, SA:2, SB:3	$R1 \leftarrow R2 - R3$
35	0100000 000 100 101	32 (Store)	SA:4, SB:5	$M[R4] \leftarrow R5$
45	1000010 010 111 011	66 (Add Immediate)	DR:2, SA:7, OP:3	$R2 \leftarrow R7 + 3$
55	1100000 101 110 100	96 (Branch on Zero)	AD: 44, SA:6	If $R6 = 0$ , $PC \leftarrow PC - 20$
70	00000000011000000	Data = 192. After execution of instruction in 35, Data = 80.		

$R4 = 70 \rightarrow M[R4] = 192$

After the instruction 2 (35):

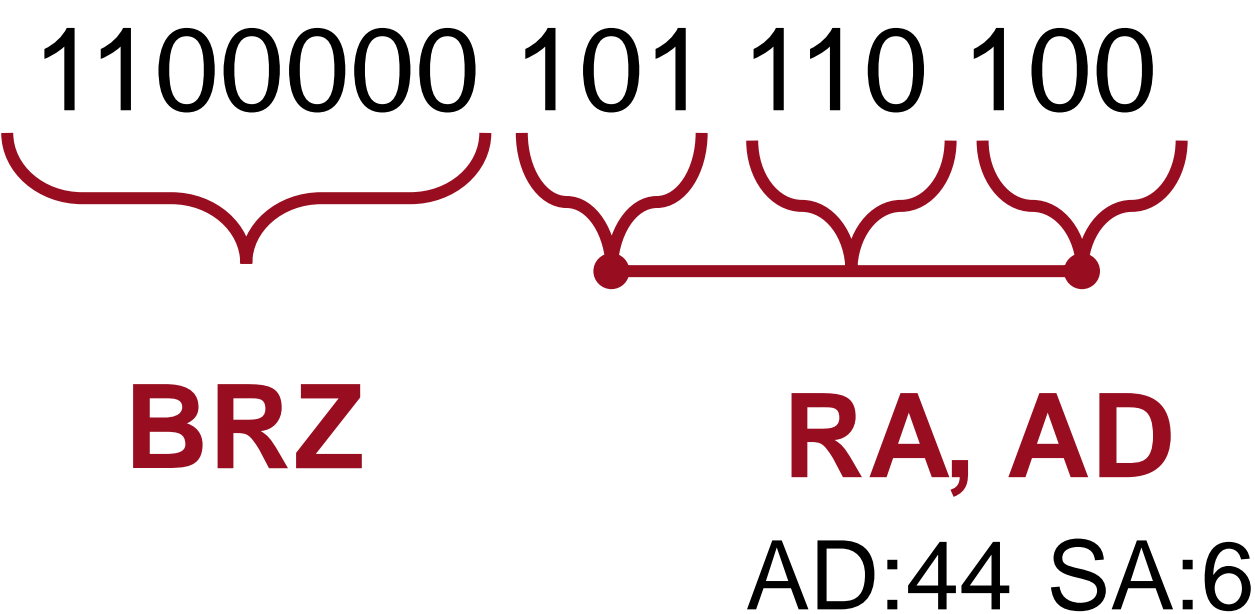
$R5 = 80 \rightarrow M[R4] = 80$

Immediate operand



# Instruction Specifications: execution

4. Address 55: **PC** ← **PC** – 20 (if R6=0)



- In this example, the word length is 16 bits
- In many recent computers, the word length is from 32 to 64 bits (higher range for immediate operands and addresses)
- The number of registers is often larger, so the register fields in the instructions must contain more bits

TABLE 8-9  
Memory Representation of Instructions and Data

Decimal Address	Memory Contents	Decimal Opcode	Other Fields	Operation
25	0000101 001 010 011	5 (Subtract)	DR:1, SA:2, SB:3	R1 ← R2 – R3
35	0100000 000 100 101	32 (Store)	SA:4, SB:5	M[R4] ← R5
45	1000010 010 111 011	66 (Add Immediate)	DR:2, SA:7, OP:3	R2 ← R7 + 3
55	1100000 101 110 100	96 (Branch on Zero)	AD: 44, SA:6	If R6 = 0, PC ← PC – 20
70	00000000011000000	Data = 192. After execution of instruction in 35, Data = 80.		



# Operations and microoperations

- A **computer operation** is specified by an instruction stored in binary, in the computer's memory.
  - The control unit in the computer uses the address or addresses provided by the PC to retrieve the instruction from memory (**fetch**)
  - The control unit decodes the instruction (e.g., opcode, register addresses) to perform the required microoperations for the execution of the instruction
- A **microoperation** is specified by the bits in a control word in the **hardware** which is decoded by the computer hardware to execute the microoperation.
- The execution of a **computer operation** often requires a **sequence or program of microoperations**, rather than a single microoperation.

# Single-Cycle Computer

- The fetch and the execution of an instruction are executed in a single clock cycle
- The **memory M** is attached to the Address Out, Data Out e Data In by connections to the datapath
- The **memory M** has a single control signal to enable/disable the writing
- The **control unit** retrieves and decodes the instruction (from the PC)
- The **control word** is **associated** with the datapath and it is executed in a single clock cycle
- Except the PC (sequential), the instruction decoder, branch control, function unit are combinational circuits (single cycle)

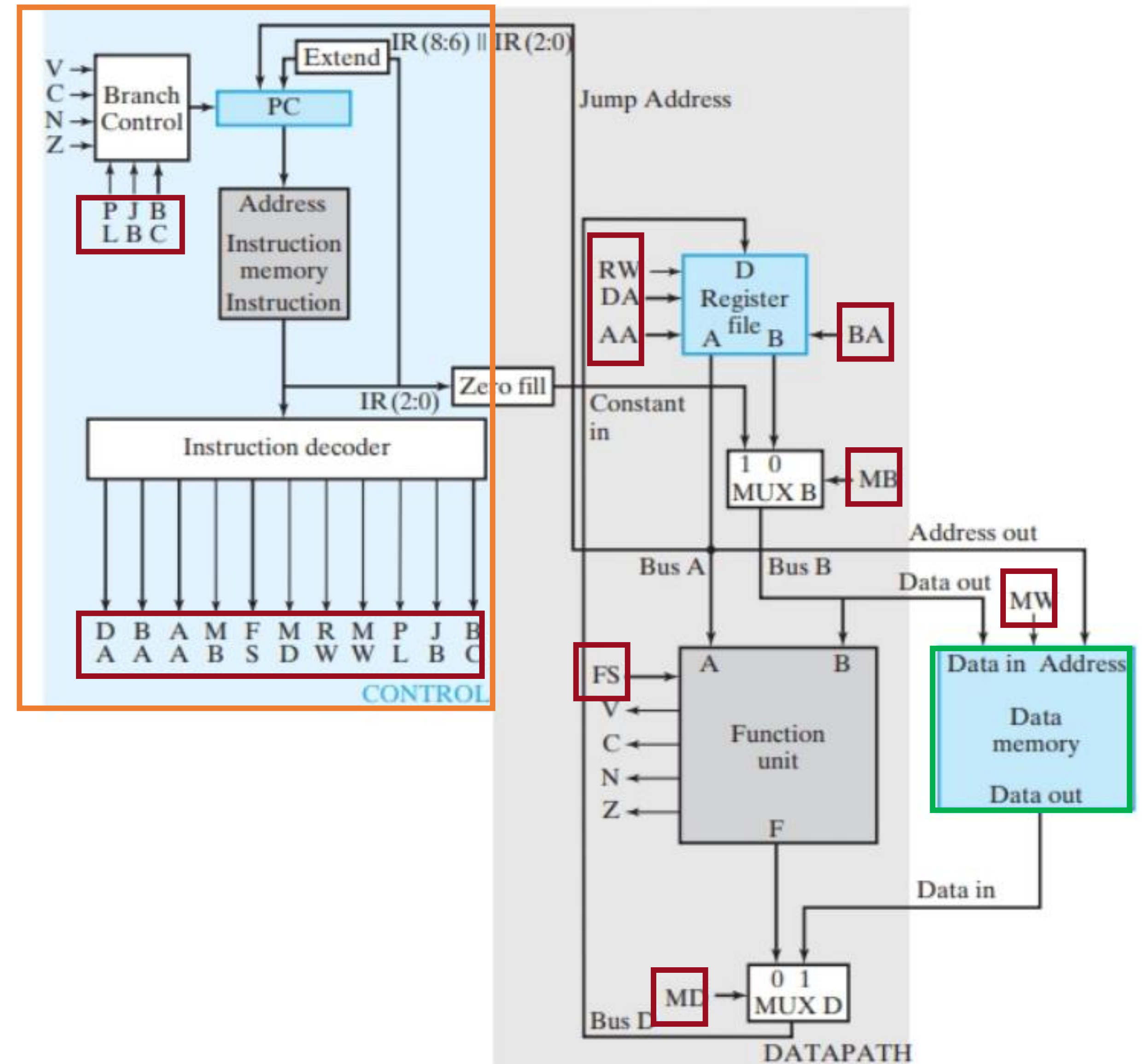


FIGURE 8-15  
Block Diagram for a Single-Cycle Computer



# Single-Cycle Computer: an example (1)

SUB R1, R2, R3 (R1←R2-R3)

- PC = 25
- Fetch of the instruction from the address in memory 25
- Instruction decoder:

DA	BA	AA	MB	FS	MD	RW	MW	PL	JB	BC
001	011	010	0	0101	0	1	0	0	X	X
R1	R3	R2	-	SUB	-	-	-	-	-	-

- Microoperation
- PC = PC + 1 = 26

In un single clock cycle

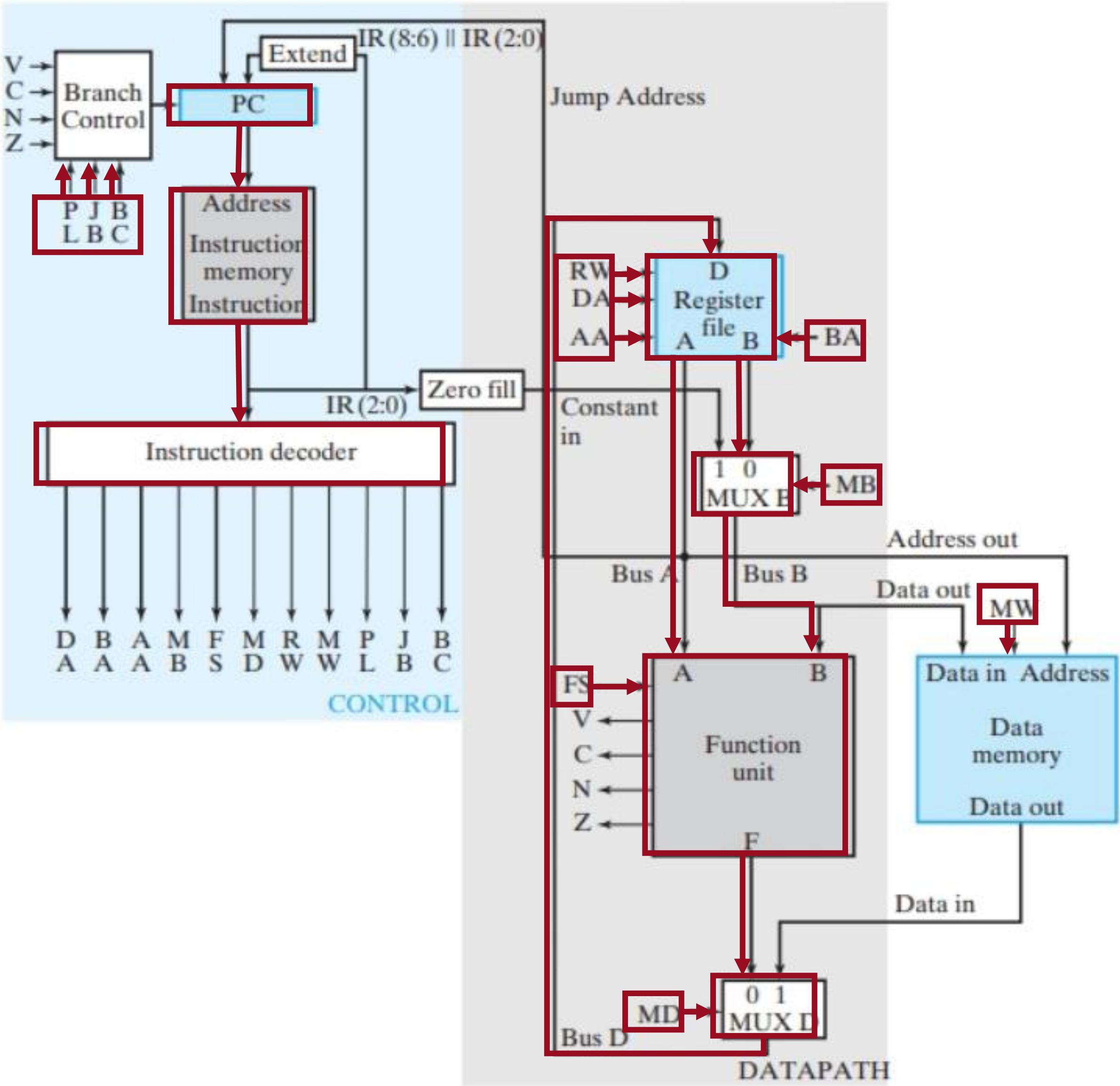


FIGURE 8-15  
Block Diagram for a Single-Cycle Computer

# Single-Cycle Computer: an example (2)

## ADI R2 ← R7 + 3

- PC = 55
- Fetch of the instruction from the address in memory 25
- Instruction decoder:

DA	BA	AA	MB	FS	MD	RW	MW	PL	JB	BC
010	011	111	1	1..10	0	1	0	0	X	X
R2	3	R7	-	ADI	-	-	-	-	-	-

- Microoperation
- PC = PC + 1 = 56

In un single clock cycle

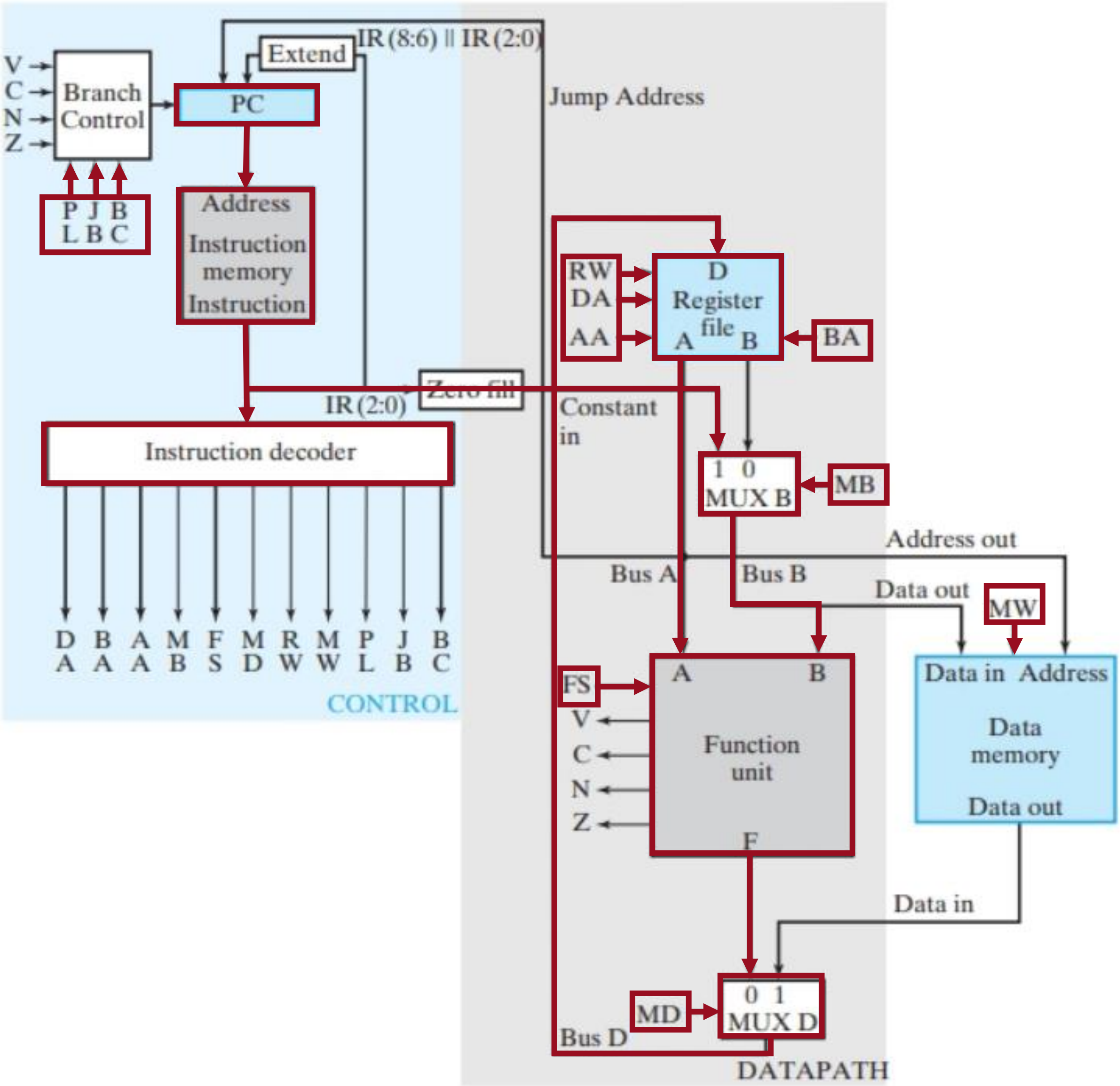
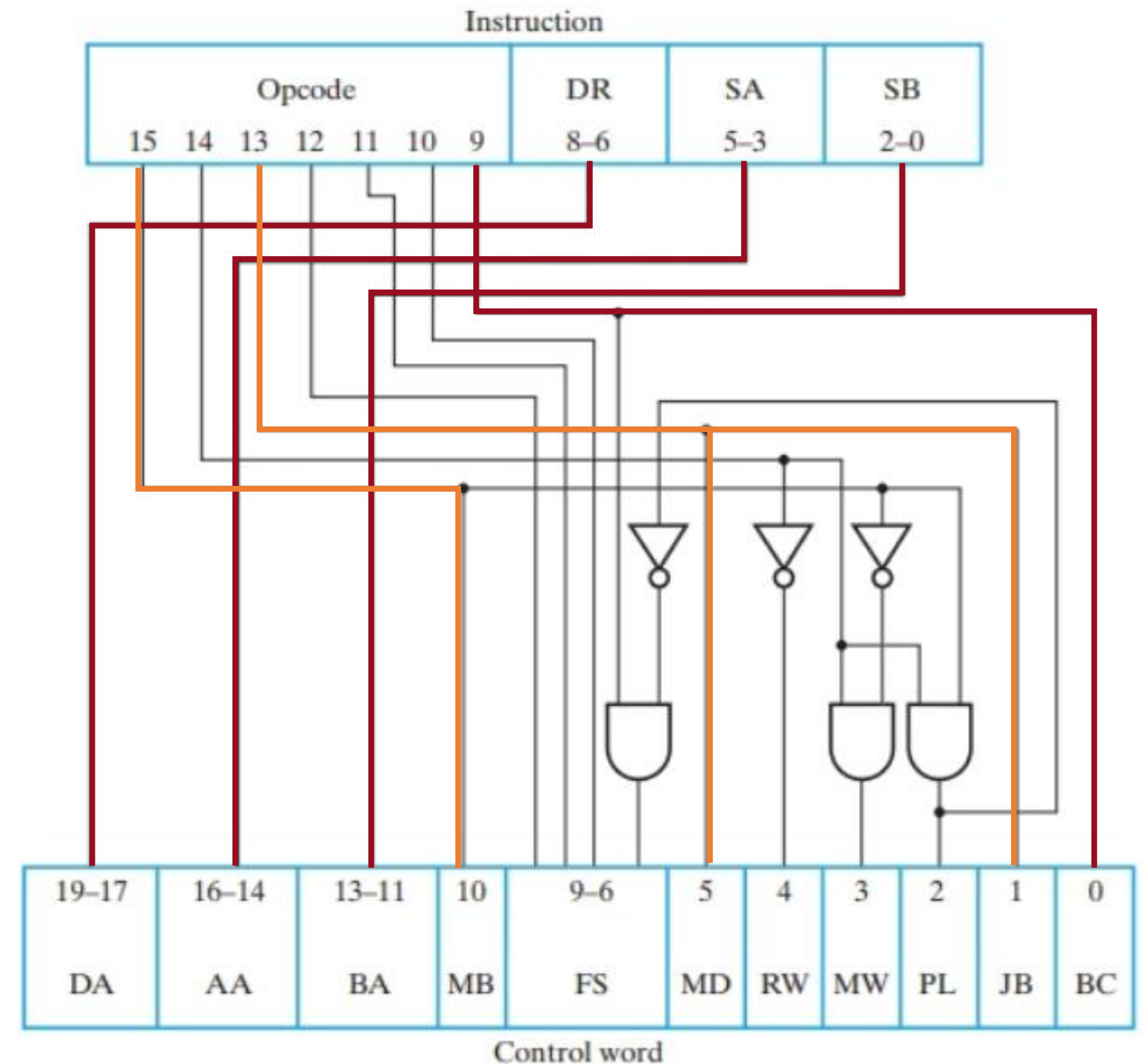


FIGURE 8-15  
Block Diagram for a Single-Cycle Computer



# Instruction Decoder

- It is a combinational circuit that provides all the control words for the datapath
- A number of fields of the control words can be obtained directly from the contents of the fields in the instruction  
 $(DA, AA, BA, BC \leftarrow DR, SA, SB, OpCode[9])$
- Some status bits are directly taken by the instruction  
 $(MB, MD, JB \leftarrow OpCode[15], OpCode[13])$
- Some of the single-bit-control-word fields require logic for their implementation
- There are two added bits for the control of the PC: PL e JB.
  - $PL = 1 \rightarrow$  if there is to be a jump or branch
  - $PL = 0 \rightarrow$  PC is incremented by one unit
  - $PL = 1, JB = 1 \rightarrow$  Jump
  - $PL = 1, JB = 0 \rightarrow$  Conditional branch

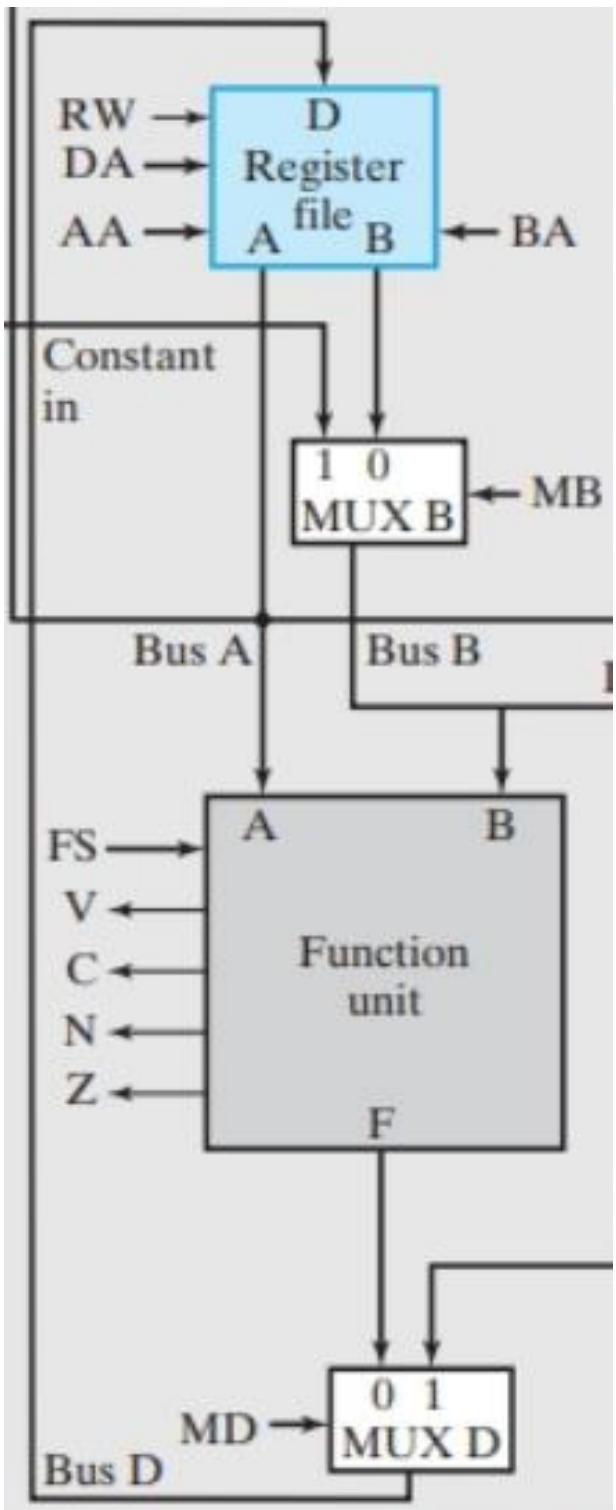


**FIGURE 8-16**  
Diagram of Instruction Decoder



# Design instruction decoder

- The instructions are divided into groups (function types)
- The instruction function types are based on the use of specific hardware resources
- For example, the first uses:
  - ALU
  - MUX B to use the Register file source (MB)
  - MUX D to use the Function unit output (MD)
  - Write to the Register file (RW)



**TABLE 8-10**  
**Truth Table for Instruction Decoder Logic**

Instruction Function Type	Instruction Bits				Control-Word Bits						
	15	14	13	9	MB	MD	RW	MW	PL	JB	BC
Function-unit operations using registers	0	0	0	X	0	0	1	0	0	X	X
Memory read	0	0	1	X	0	1	1	0	0	X	X
Memory write	0	1	0	X	0	X	0	1	0	X	X
Function-unit operations using register and constant	1	0	0	X	1	0	1	0	0	X	X
Conditional branch on zero ( <i>Z</i> )	1	1	0	0	X	X	0	0	1	0	0
Conditional branch on negative ( <i>N</i> )	1	1	0	1	X	X	0	0	1	0	1
Unconditional jump	1	1	1	X	X	X	0	0	1	1	X

# Design of the instruction decoder

Treating this table as a truth table and optimizing the logic functions, the logic for the single-bit outputs of the instruction decoder in the figure results.

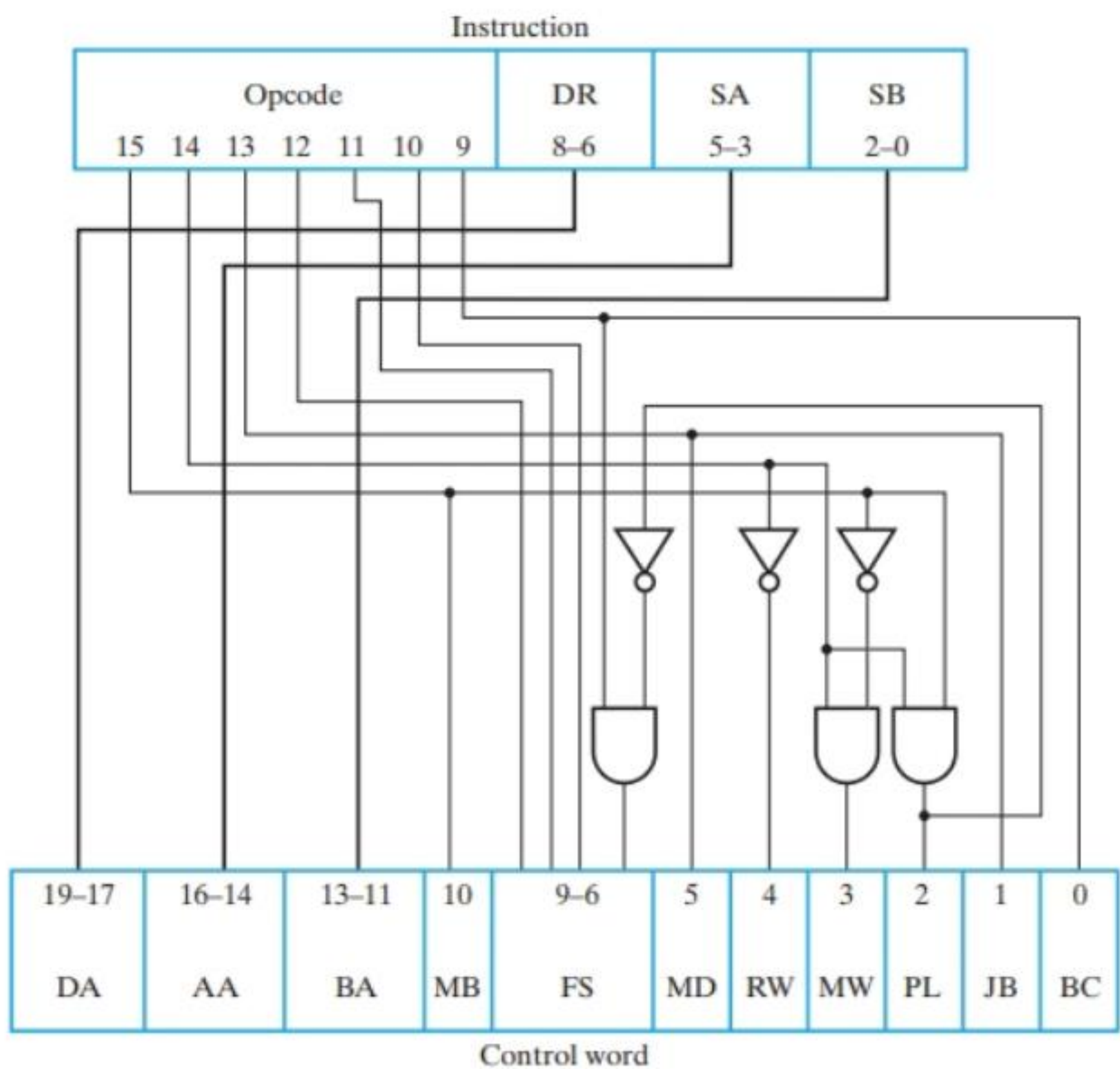


FIGURE 8-16  
Diagram of Instruction Decoder

TABLE 8-10  
Truth Table for Instruction Decoder Logic

Instruction Function Type	Instruction Bits				Control-Word Bits						
	15	14	13	9	MB	MD	RW	MW	PL	JB	BC
Function-unit operations using registers	0	0	0	X	0	0	1	0	0	X	X
Memory read	0	0	1	X	0	1	1	0	0	X	X
Memory write	0	1	0	X	0	X	0	1	0	X	X
Function-unit operations using register and constant	1	0	0	X	1	0	1	0	0	X	X
Conditional branch on zero (Z)	1	1	0	0	X	X	0	0	1	0	0
Conditional branch on negative (N)	1	1	0	1	X	X	0	0	1	0	1
Unconditional jump	1	1	1	X	X	X	0	0	1	1	X



# Design of the instruction decoder

Treating this table as a truth table and optimizing the logic functions, the logic for the single-bit outputs of the instruction decoder in the figure results.

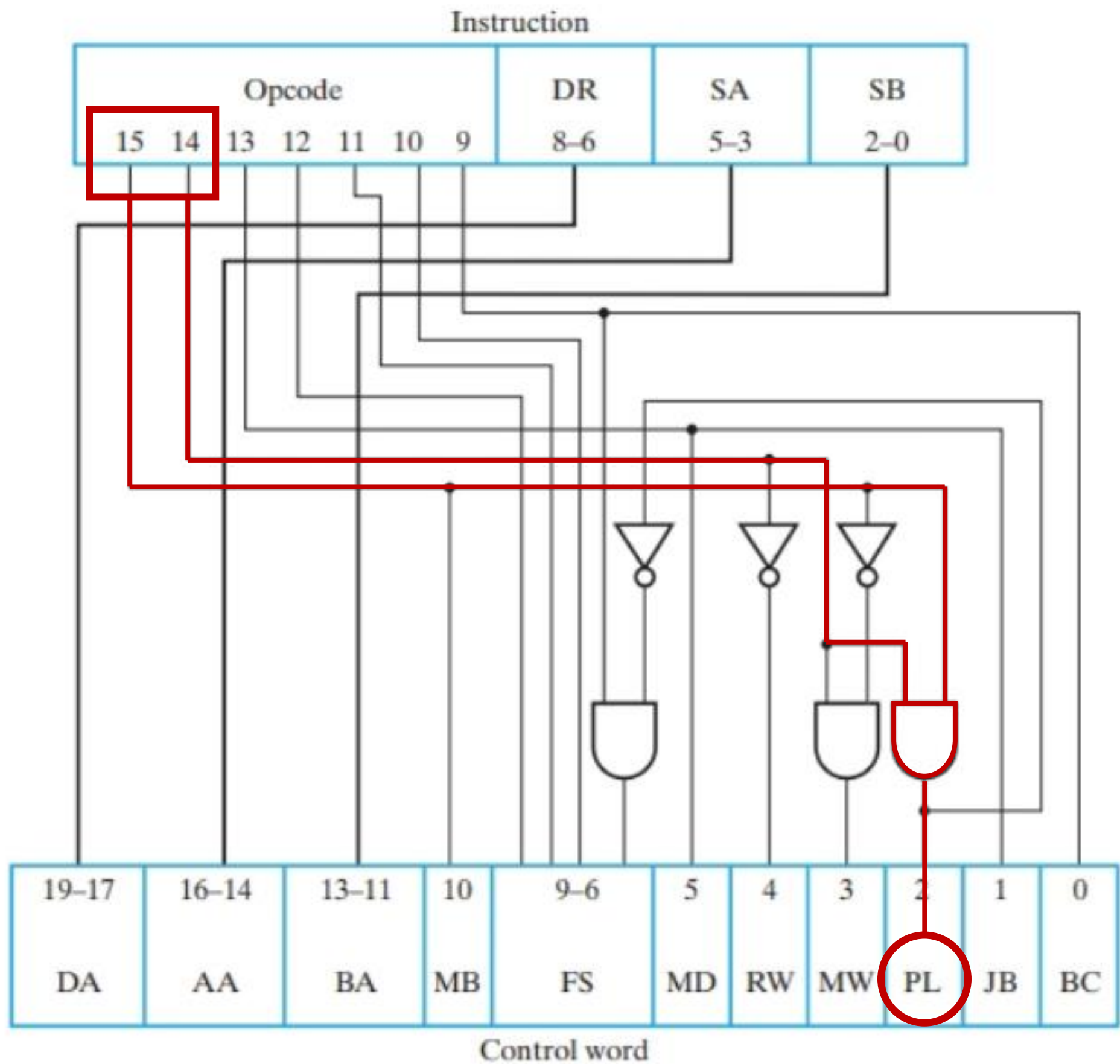


FIGURE 8-16  
Diagram of Instruction Decoder

TABLE 8-10  
Truth Table for Instruction Decoder Logic

Instruction Function Type	Instruction Bits				Control-Word Bits						
	15	14	13	9	MB	MD	RW	MW	PL	JB	BC
Function-unit operations using registers	0	0	0	X	0	0	1	0	0	X	X
Memory read	0	0	1	X	0	1	1	0	0	X	X
Memory write	0	1	0	X	0	X	0	1	0	X	X
Function-unit operations using register and constant	1	0	0	X	1	0	1	0	0	X	X
Conditional branch on zero (Z)	1	1	0	0	X	X	0	0	1	0	0
Conditional branch on negative (N)	1	1	0	1	X	X	0	0	1	0	1
Unconditional jump	1	1	1	X	X	X	0	0	1	1	X

Example: we can derive  
PL = 1 for the operation  
Jump and Branch from  
14th and 15th bit



# Design of the instruction decoder

TABLE 8-8  
Instruction Specifications for the Simple Computer

Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move A	0000000	MOVA	RD, RA	$R[DR] \leftarrow R[SA]^*$	N, Z
Increment	0000001	INC	RD, RA	$R[DR] \leftarrow R[SA] + 1^*$	N, Z
Add	0000010	ADD	RD, RA, RB	$R[DR] \leftarrow R[SA] + R[SB]^*$	N, Z
Subtract	0000101	SUB	RD, RA, RB	$R[DR] \leftarrow R[SA] - R[SB]^*$	N, Z
Decrement	0000110	DEC	RD, RA	$R[DR] \leftarrow R[SA] - 1^*$	N, Z
AND	0001000	AND	RD, RA, RB	$R[DR] \leftarrow R[SA] \wedge R[SB]^*$	N, Z
OR	0001001	OR	RD, RA, RB	$R[DR] \leftarrow R[SA] \vee R[SB]^*$	N, Z
Exclusive OR	0001010	XOR	RD, RA, RB	$R[DR] \leftarrow R[SA] \oplus R[SB]^*$	N, Z
NOT	0001011	NOT	RD, RA	$R[DR] \leftarrow \overline{R[SA]}^*$	N, Z
Move B	0001100	MOVB	RD, RB	$R[DR] \leftarrow R[SB]^*$	
Shift Right	0001101	SHR	RD, RB	$R[DR] \leftarrow sr\ R[SB]^*$	
Shift Left	0001110	SHL	RD, RB	$R[DR] \leftarrow sl\ R[SB]^*$	
Load Immediate	1001100	LDI	RD, OP	$R[DR] \leftarrow zf\ OP^*$	
Add Immediate	1000010	ADI	RD, RA, OP	$R[DR] \leftarrow R[SA] + zf\ OP^*$	N, Z
Load	0010000	LD	RD, RA	$R[DR] \leftarrow M[SA]^*$	
Store	0100000	ST	RA, RB	$M[SA] \leftarrow R[SB]^*$	
Branch on Zero	1100000	BRZ	RA, AD	if ( $R[SA] = 0$ ) $PC \leftarrow PC + se\ AD$ , N, Z if ( $R[SA] \neq 0$ ) $PC \leftarrow PC + 1$	
Branch on Negative	1100001	BRN	RA, AD	if ( $R[SA] < 0$ ) $PC \leftarrow PC + se\ AD$ , N, Z if ( $R[SA] \geq 0$ ) $PC \leftarrow PC + 1$	
Jump	1110000	JMP	RA	$PC \leftarrow R[SA]$	

\* For all of these instructions,  $PC \leftarrow PC + 1$  is also executed to prepare for the next cycle.

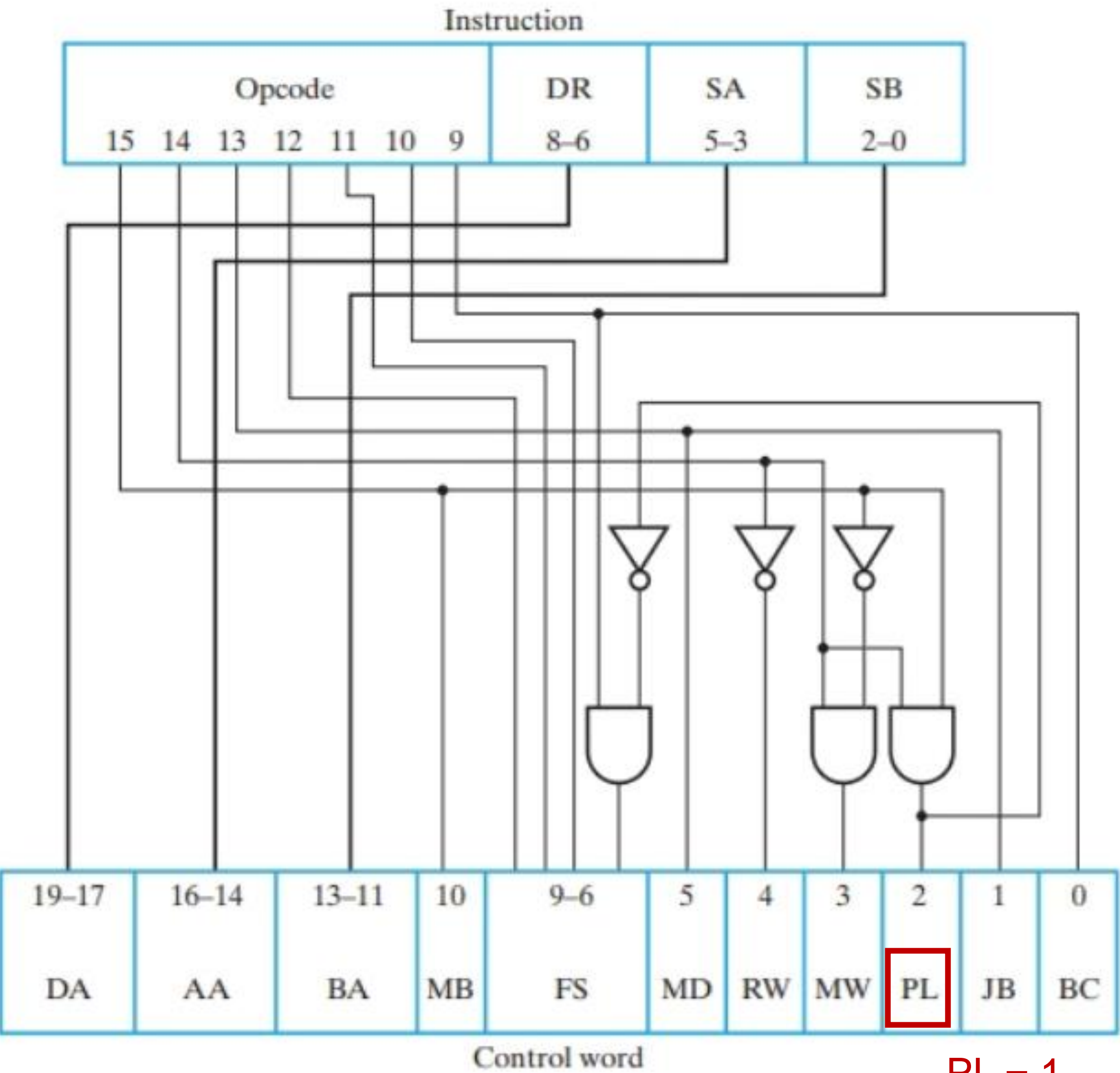


FIGURE 8-16  
Diagram of Instruction Decoder

PL = 1  
BIT 15th and  
14th = 1



# Examples of Instructions for a Single-Cycle Computer

□ **TABLE 8-11**  
Six Instructions for the Single-Cycle Computer

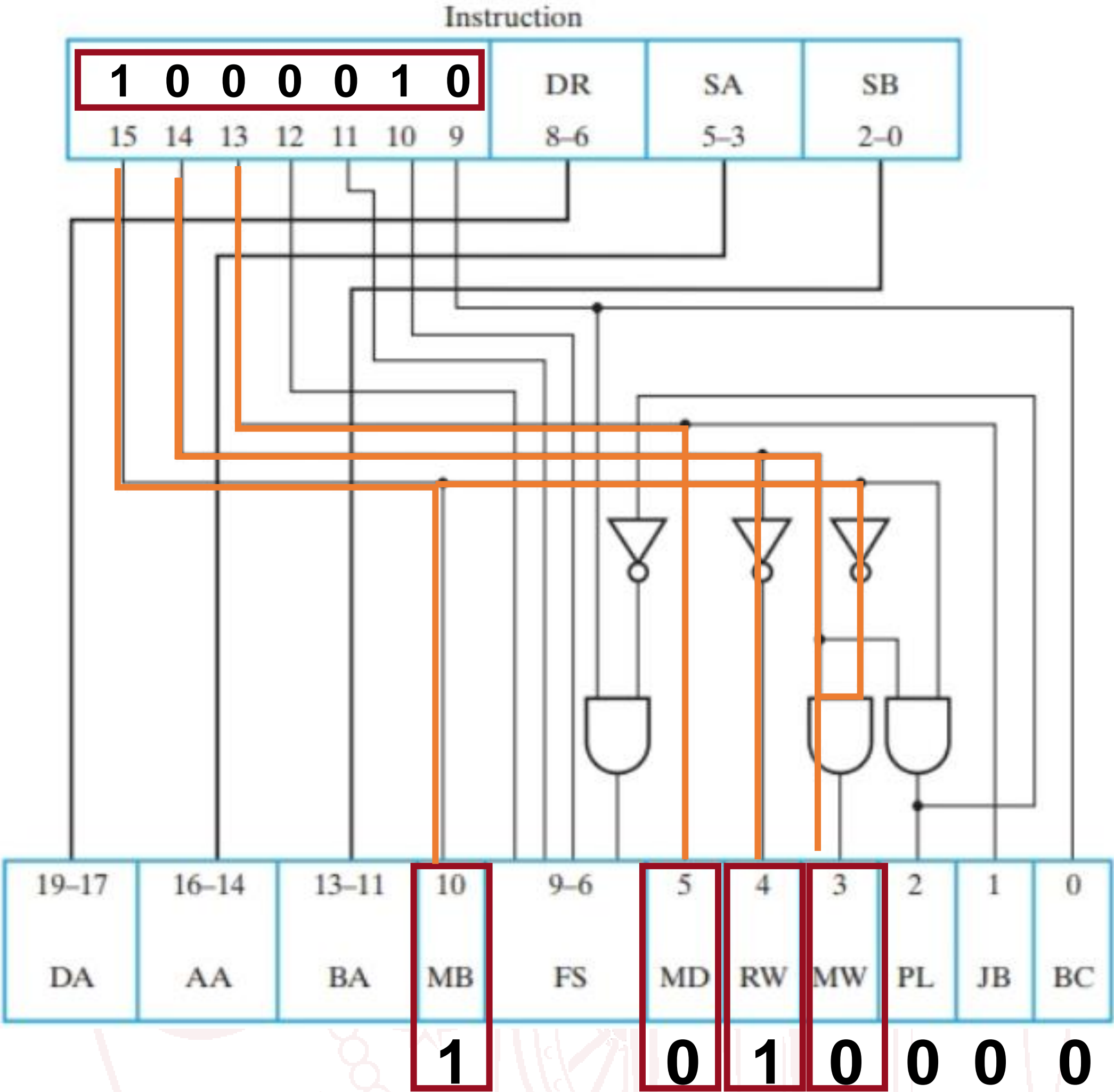
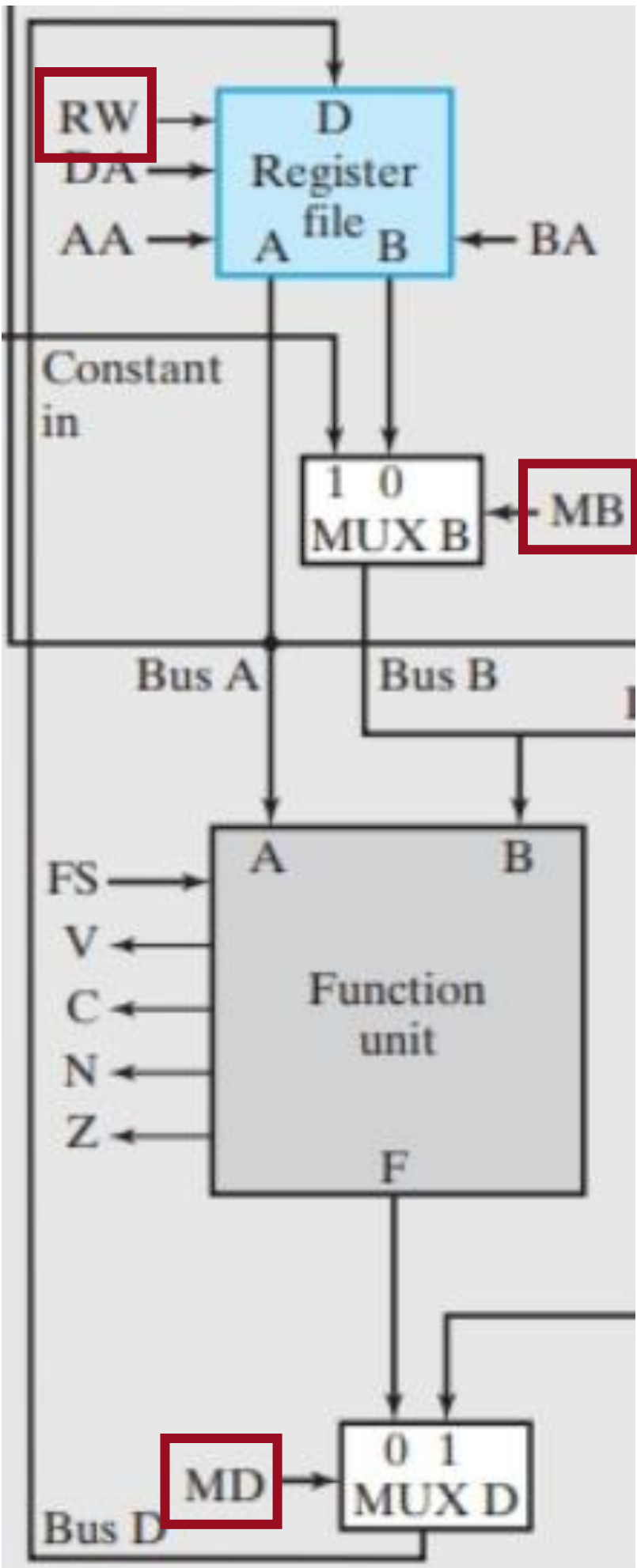
Operation Code	Symbolic Name	Format	Description	Function	MB	MD	RW	MW	PL	JB	BC
1000010	ADI	Immediate	Add immediate operand	$R[DR] \leftarrow R[SA] + \boxed{zf} I(2:0)$	1	0	1	0	0	0	0
0010000	LD	Register	Load memory content into register	$R[DR] \leftarrow M[R[SA]]$	0	1	1	0	0	1	0
0100000	ST	Register	Store register content in memory	$M[R[SA]] \leftarrow R[SB]$	0	1	0	1	0	0	0
0001110	SL	Register	Shift left	$R[DR] \leftarrow sl R[SB]$	0	0	1	0	0	1	0
0001011	NOT	Register	Complement register	$R[DR] \leftarrow \overline{R[SA]}$	0	0	1	0	0	0	1
1100000	BRZ	Jump/Branch	If $R[SA] = 0$ , branch to $PC + se AD$	If $R[SA] = 0$ , $PC \leftarrow PC + se \boxed{AD}$ If $R[SA] \neq 0$ , $PC \leftarrow PC + 1$	1	0	0	0	1	0	0



# Examples of Instructions for a Single-Cycle Computer

Operation Code	Symbolic Name	Format	Description	Function	MB	MD	RW	MW	PL	JB	BC
1000010	ADI	Immediate	Add immediate operand	$R[DR] \leftarrow R[SA] + zf\ I(2:0)$	1	0	1	0	0	0	0

- MB = 1, uses constant in the sum
- MD = 0, uses output function unit
- RW = 1, write in the register
- MW = 0, not write in the memory
- PL, JB = 0, no jump/branch



# Example of a program

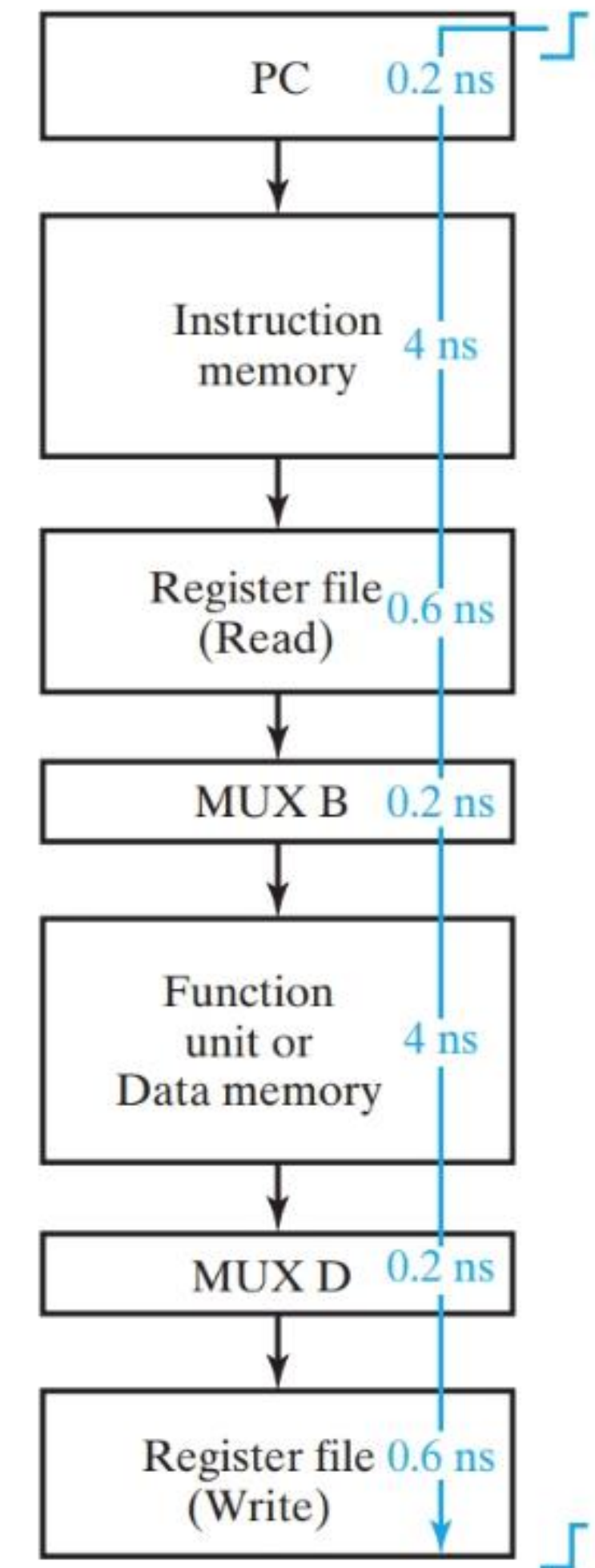
- Implementing the arithmetic expression:  $83 - (2 + 3)$
- Data of the problem:
  - $R3 = 248$
  - $M[248] = 2$
  - $M[249] = 83$
  - The result stored in  $M[250]$

1	LD	R1, R3	Load R1 with contents of location 248 in memory ( $R1 = 2$ )
2	ADI	R1, R1, 3	Add 3 to R1, $R1 + 3$ ( $R1 = 5$ )
3	NOT	R1, R1	Complement R1
4	INC	R1, R1	Increment R1 ( $R1 = -5$ )
5	INC	R3, R3	Increment the contents of R3 ( $R3 = 249$ )
6	LD	R2, R3	Load R2 with contents of location 249 in memory ( $R2 = 83$ )
7	ADD	R2, R2, R1	R1 Add contents of R1 to contents of R2 ( $R2 = 83 + (-5) = 78$ )
8	INC	R3, R3	Increment the contents of R3 ( $R3 = 250$ )
9	ST	R3, R2	Store R2 in memory location 250 ( $M[250] = 78$ )

# Single-Cycle Computer Issues

- Complex operations that cannot be executed in a single clock
  - For instance: bitwise multiplication
- Two read accesses of the memory are required
  - The first access to obtain the instruction (one cycle)
  - The second access to read and write the data (another cycle)
- The maximum frequency of the clock depends on the delay due to the propagation of the signal in the circuit (worst case)
  - Total delay along the path is 9.8 ns  $\rightarrow$  Max Freq Clock= 102 MHz

## Multiple-Cycle Computer

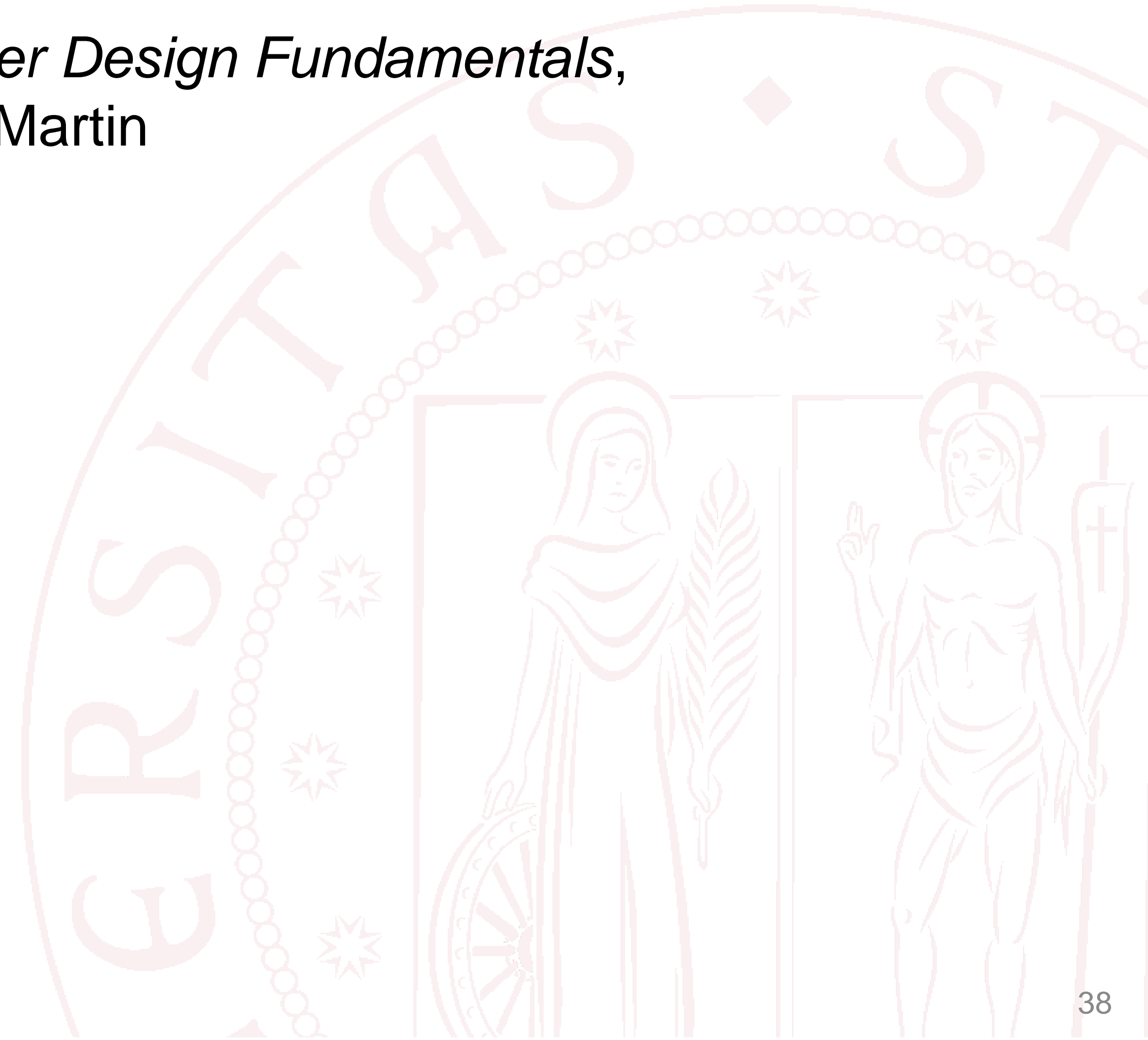


**FIGURE 8-17**  
Worst-Case Delay Path in Single-Cycle Computer

# Disclaimer

Figures from *Logic and Computer Design Fundamentals*,  
Fifth Edition, GE Mano | Kime | Martin

© 2016 Pearson Education, Ltd





# Questions

