

Intro to Software Engineering EK327

Giacomo Cappelletto

21/1/25

Contents

Chapter 1	Digital Logic	Page 2
1.1	Adding Two Single-Bit Binaries Using XOR and AND Gates	2
1.2	Adding Two Single-Byte Binary Numbers Using XOR and AND	2
1.3	One's Complement for a Single Byte	3
1.4	Carrying When Adding Negatives in One's Complement	4
1.5	Two's Complement for a Single Byte	5
1.6	Carrying When Adding Negatives in Two's Complement	6
Chapter 2	Microarchitecture	Page 8
2.1	Computer Architecture Overview	8
	Address Bus — 8 • Data Bus — 9 • Processor Components — 9	
Chapter 3	Instruction Set	Page 11
3.1	Assembly Pseudocode Conventions	11
3.2	Instruction Encoding for 16-bit Memory Cells	13
3.3	Assemble	14
3.4	Binary to Hexadecimal Conversion	16
3.5	Machine Code Decoding	16
	Hexadecimal Machine Code — 16 • Instruction Breakdown — 16 • Program Behavior — 17	
3.6	Fetch-Decode-Execute-Store Cycle	18
	Fetch — 18 • Decode — 18 • Execute — 18 • Store — 18	
3.7	Five-Stage Pipeline for Operations	18
	1. Fetch (IF) — 19 • 2. Decode (ID) — 19 • 3. Execute (EX) — 19 • 4. Memory Access (MEM) — 19 • 5. Write Back (WB) — 19 • Example of parallel processing in 5 stage pipeline — 19	
3.8	Stalling and Forwarding in a 5-Stage Pipeline	19
	Stalling — 20 • Forwarding — 21	
3.9	Handling Simultaneous Memory Reads in IF and WB Stages	21
	Stalling — 21 • Harvard Architecture — 22	
3.10	Memory Pointers	22
3.11	Memory Indirection in C++	23

Chapter 1

Digital Logic

1.1 Adding Two Single-Bit Binaries Using XOR and AND Gates

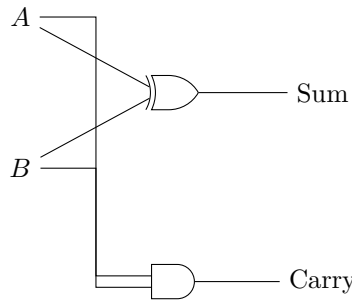
To add two single-bit binary numbers A and B , we need to calculate: 1. The **sum bit**, which is the XOR of A and B . 2. The **carry bit**, which is the AND of A and B .

Logic - Sum Bit: $\text{Sum} = A \oplus B$ - **Carry Bit:** $\text{Carry} = A \cdot B$

The following truth table summarizes the operation:

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The addition is implemented using XOR and AND gates as shown below:



The XOR gate produces the sum bit, while the AND gate produces the carry bit. This forms the fundamental building block of a full binary adder.

1.2 Adding Two Single-Byte Binary Numbers Using XOR and AND

Binary addition can be performed on two single-byte numbers using bitwise operations. Specifically: - The **XOR (Exclusive OR)** operation gives the sum of two bits without considering the carry. - The **AND** operation identifies where a carry will occur. - The carry is then shifted left by one position and added to the result in subsequent iterations.

Algorithm: 1. Compute the sum without carry using XOR:

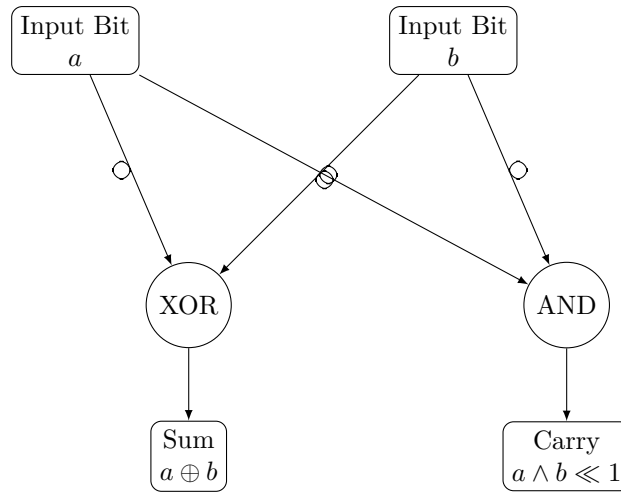
$$\text{sum} = a \oplus b$$

2. Compute the carry using AND and shift it left by one bit:

$$\text{carry} = a \wedge b \ll 1$$

- Repeat the process by adding the carry to the sum until there is no carry left.

Diagram: Below is a diagram that illustrates the process for a single bit addition.



Example: Consider two single-byte binary numbers:

$$a = 01101101, \quad b = 10101001$$

- Compute the XOR for the sum:

$$\text{sum} = a \oplus b = 11000100$$

- Compute the AND for the carry and shift left:

$$\text{carry} = a \wedge b \ll 1 = 00001010$$

3. Add the sum and carry: - New sum: $\text{sum} = 11000100 \oplus 00001010 = 11001110$ - New carry: $\text{carry} = 11000100 \wedge 00001010 \ll 1 = 00000000$

- Final result: 11001110.

1.3 One's Complement for a Single Byte

The **one's complement** of a binary number is obtained by flipping all the bits, changing every 1 to 0 and every 0 to 1. This operation is commonly used in binary arithmetic, particularly in representing negative numbers in early computing systems.

Steps to Compute One's Complement: 1. Write down the binary number. 2. Flip all bits: - Change each 0 to 1. - Change each 1 to 0.

Example: Consider the binary number $a = 01101101$.

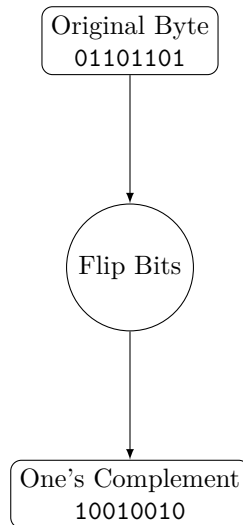
- Original binary number:

$$a = 01101101$$

- One's complement (flip all bits):

$$\text{one's complement of } a = 10010010$$

Diagram: The following diagram illustrates the transformation:



Properties of One's Complement: - A binary number and its one's complement always sum to all 1s (i.e., 11111111 for a single byte). - One's complement is useful in representing signed integers: - Positive numbers are represented as-is. - Negative numbers are represented by the one's complement of their positive counterparts.

Example with Signed Integers: For a single byte: 1. 5 in binary: 00000101 2. -5 in one's complement: 11111010

Verification: Adding 5 and -5 in one's complement arithmetic:

00000101 11111010 = 11111111 all bits are 1, representing zero in one's complement arithmetic.

1.4 Carrying When Adding Negatives in One's Complement

In one's complement representation, negative numbers are represented by flipping all the bits of their positive counterparts. When adding two negative numbers, a carry might be generated, which needs to be added back to the result to obtain the correct answer.

Example: Adding -3 and -4.

1. Represent -3 and -4 in one's complement for a single byte:

$$3 = 00000011, \quad -3 = \text{one's complement of } 3 = 11111100$$

$$4 = 00000100, \quad -4 = \text{one's complement of } 4 = 11111011$$

2. Add the two numbers:

$$\begin{array}{r} 11111100 \\ 11111011 \\ \hline 11110111 \end{array}$$

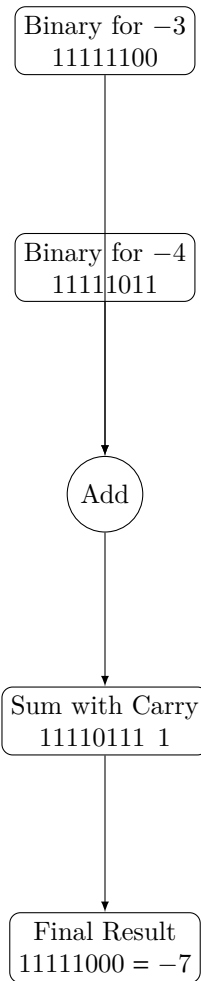
3. Handle the carry: - The result of the addition is 11110111, with a carry bit of 1. - Add the carry back to the least significant bit:

$$11110111 \ 1 = 11111000$$

4. Final result:

$$11111000 = \text{one's complement of } 7 = -7$$

Explanation: - When the sum generates a carry, it must be added back to the result to comply with one's complement rules. - The result of $-3 - 4 = -7$, as expected.



Note: This approach works for signed integers in one's complement and highlights the importance of handling the carry bit to ensure accurate results.

1.5 Two's Complement for a Single Byte

The **two's complement** of a binary number is obtained by flipping all the bits (as in one's complement) and then adding 1 to the result. This method is widely used in modern computer systems to represent signed integers.

Steps to Compute Two's Complement: 1. Write down the binary number. 2. Flip all bits (as in one's complement). 3. Add 1 to the flipped number.

Example: Consider the binary number $a = 01101101$.

1. Original binary number:

$$a = 01101101$$

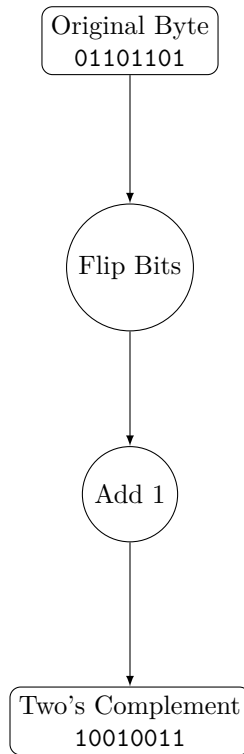
2. Flip all bits (one's complement):

$$10010010$$

3. Add 1 to the result:

$$\text{two's complement of } a = 10010010 + 1 = 10010011$$

Diagram: The following diagram illustrates the transformation:



Properties of Two's Complement: - The two's complement of 0 is 0, and the two's complement of the maximum negative value is itself. - Negative numbers are represented by their two's complement. - Addition and subtraction with two's complement do not require separate subtraction logic, simplifying arithmetic operations.

1.6 Carrying When Adding Negatives in Two's Complement

In two's complement representation, negative numbers are represented by flipping all bits of the positive number and adding 1. When adding two negative numbers, the carry generated during addition is discarded.

Example: Adding -3 and -4 .

1. Represent -3 and -4 in two's complement for a single byte:

$$3 = 00000011, \quad -3 = \text{two's complement of } 3 = 11111101$$

$$4 = 00000100, \quad -4 = \text{two's complement of } 4 = 11111100$$

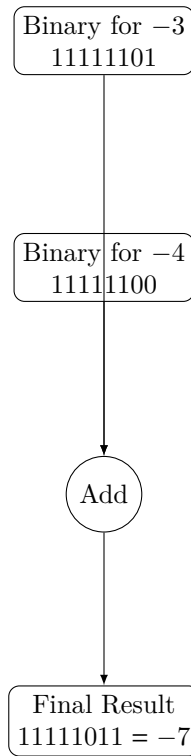
2. Add the two numbers:

$$\begin{array}{r} 11111110 \\ 11111100 \\ \hline 11111011 \end{array}$$

3. Handle the carry: - The result of the addition is 11111011. - In two's complement, the carry is ignored, so the final result is:

$$11111011 = -7$$

Explanation: - In two's complement, the carry bit is discarded, unlike in one's complement. - The result of $-3 - 4 = -7$, as expected.

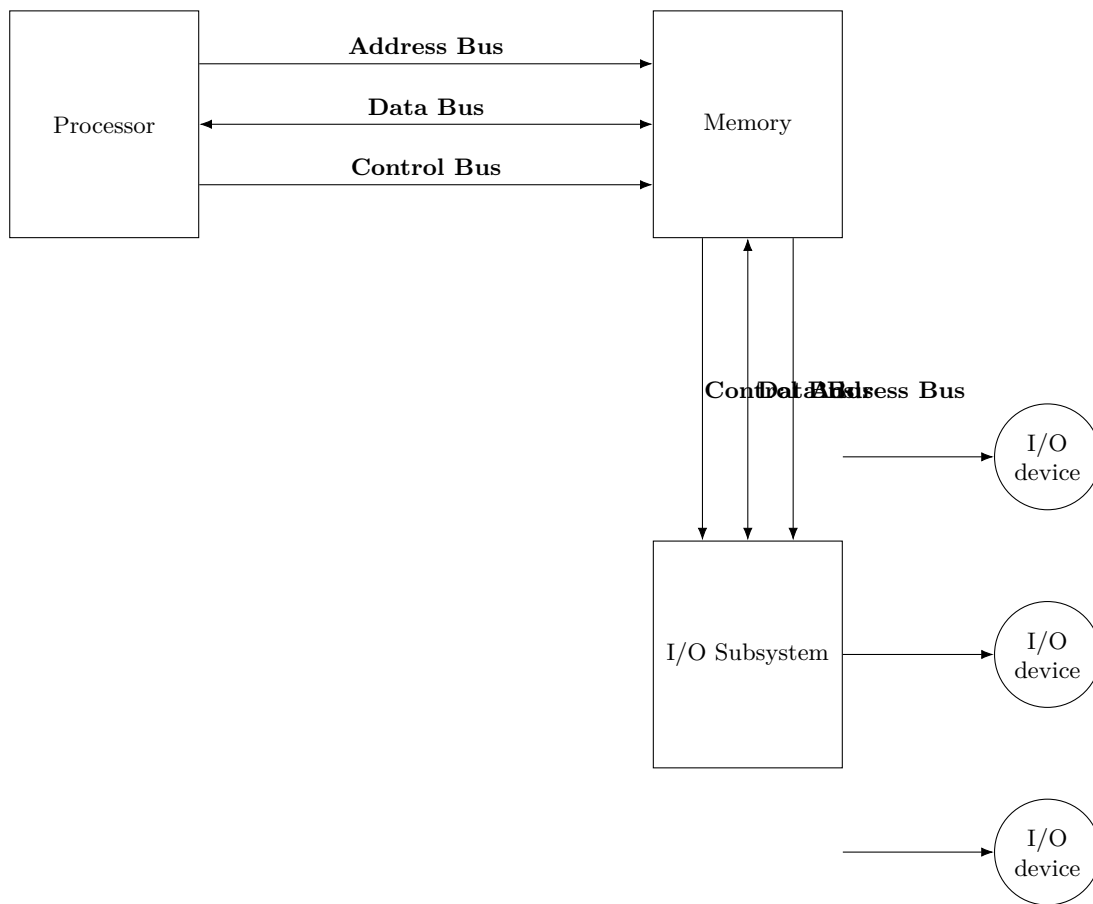


Note: Two's complement simplifies arithmetic operations by eliminating the need to add back the carry. It is the most commonly used method for representing signed integers in modern computing systems.

Chapter 2

Microarchitecture

2.1 Computer Architecture Overview



2.1.1 Address Bus

The address bus is a critical component in a computer's architecture, responsible for specifying the memory addresses that the processor will read from or write to. The width of the address bus determines the maximum amount of memory that can be addressed by the system.

Address Bus Width and Memory Capacity:

- **8-bit Address Bus:** - Can address $2^8 = 256$ memory locations. - Each memory location typically holds 1 byte of data. - Total addressable memory: 256 bytes.

- **16-bit Address Bus:** - Can address $2^{16} = 65,536$ memory locations. - Each memory location typically holds 1 byte of data. - Total addressable memory: 65,536 bytes (or 64 KB).

- **32-bit Address Bus:** - Can address $2^{32} = 4,294,967,296$ memory locations. - Each memory location typically holds 1 byte of data. - Total addressable memory: 4,294,967,296 bytes (or 4 GB).
- **64-bit Address Bus:** - Can address $2^{64} = 18,446,744,073,709,551,616$ memory locations. - Each memory location typically holds 1 byte of data. - Total addressable memory: 18,446,744,073,709,551,616 bytes (or 16 exabytes).

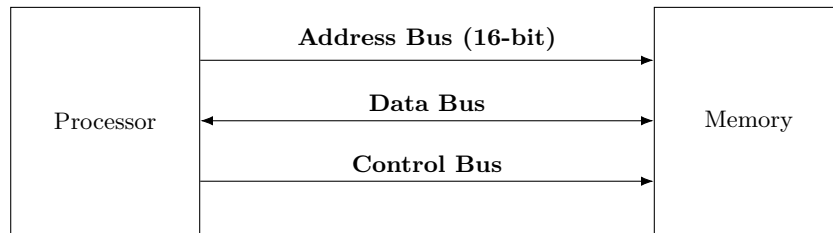
Example: 16-bit Address Bus

For a 16-bit address bus, the number of unique addresses is calculated as follows:

$$2^{16} = 65,536 \text{ addresses}$$

Each address corresponds to a unique memory cell, allowing the processor to access up to 65,536 different memory locations. If each memory location holds 1 byte, the total addressable memory is 65,536 bytes (or 64 KB).

Diagram:



The address bus plays a crucial role in determining the system's memory capacity and overall performance. A wider address bus allows for more memory to be accessed, which is essential for modern applications and operating systems.

2.1.2 Data Bus

The data bus is a bidirectional pathway that transfers data between the processor, memory, and I/O devices. It plays a crucial role in the fetch-execute cycle, enabling the processor to read instructions and data from memory and write results back to memory.

Multidirectionality in the Fetch-Execute Cycle: 1. **Fetch:** The processor uses the data bus to read an instruction from memory. 2. **Decode:** The instruction is decoded internally by the processor. 3. **Execute:** The processor may read or write data to/from memory or I/O devices using the data bus.

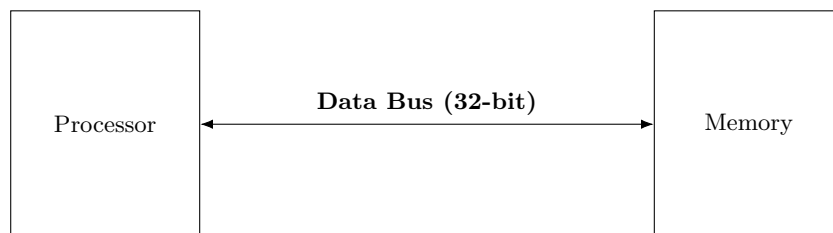
Bandwidth and Cell Size: - The width of the data bus determines the amount of data transferred in one cycle. - A wider data bus can transfer more data per cycle, increasing overall system performance. - For example, a 32-bit data bus can transfer 4 bytes per cycle, while a 64-bit data bus can transfer 8 bytes per cycle.

Example: 32-bit Data Bus

For a 32-bit data bus, the amount of data transferred per cycle is calculated as follows:

$$\text{Data per cycle} = 32 \text{ bits} = 4 \text{ bytes}$$

Diagram:



The data bus's bidirectional nature and bandwidth are critical for efficient data transfer, directly impacting the system's speed and performance.

2.1.3 Processor Components

The processor, also known as the Central Processing Unit (CPU), is the brain of the computer. It performs calculations, executes instructions, and manages data flow within the system. Key components of the processor include registers, the program counter (PC), and the EFLAGS register.

Registers

Registers are small, fast storage locations within the CPU used to hold data temporarily during execution. They are crucial for the processor's operation, allowing quick access to frequently used values. In this architecture, we have general-purpose registers and special-purpose registers.

General-Purpose Registers: - The CPU has six general-purpose registers, named $r0$ to $r5$. - Each register can store 16 bits of data. - These registers are used for various operations, such as arithmetic, logic, and data manipulation.

Example:

Register	Value (Binary)
$r0$	0000000000000001
$r1$	0000000000000010
$r2$	0000000000000011
$r3$	0000000000000100
$r4$	0000000000000101
$r5$	0000000000000110

Program Counter (PC)

The Program Counter (PC) is a special-purpose register that holds the address of the next instruction to be executed. It plays a critical role in the fetch-execute cycle, ensuring the CPU processes instructions in the correct sequence.

Functionality: - The PC is automatically incremented after each instruction fetch, pointing to the next instruction in memory. - If a jump or branch instruction is executed, the PC is updated to the target address, altering the flow of execution.

Example:

PC = 0000000000001000 Next instruction address

EFLAGS Register

The EFLAGS register is a special-purpose, read-only register that holds the results of recent operations. It contains several flags that provide information about the state of the CPU and the outcome of arithmetic and logical operations.

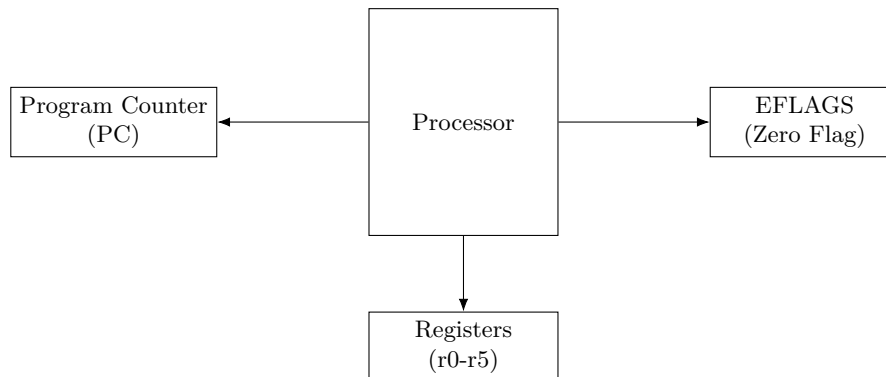
Key Flags: - **Zero Flag (ZF):** Indicates whether the result of an operation is zero. - Set to 1 if the result is zero. - Set to 0 if the result is non-zero.

Example:

EFLAGS =															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	ZF

Usage: - The Zero Flag is commonly used in conditional branching. For example, a jump instruction may depend on whether the Zero Flag is set, allowing the CPU to make decisions based on the results of previous operations.

Diagram:



These components work together to enable the processor to execute instructions efficiently, manage data flow, and make decisions based on the results of operations.

Chapter 3

Instruction Set

- We will go over all of the instructions that we will cover in this class by example, including their assembly code.
- We will then learn how to represent each instruction in binary, so that the hardware can understand it.
- We will work with them in binary only to start.
- Later, when we get to the Assembly level, we will be able to represent the instructions in more readable form.

3.1 Assembly Pseudocode Conventions

- R_n : register n (where n is 0 through 5)
- $*R_n$: data in register n
- R_n : memory cell at the address stored in register n
- $R_n \leftarrow x$: contents of register n are now x

Example 3.1.1 (Addition)

assembly:

$$addR_n, R_m$$

pseudocode:

$$R_n \leftarrow *R_n + *R_m$$

Example 3.1.2 (Subtraction)

assembly:

$$subR_n, R_m$$

pseudocode:

$$R_n \leftarrow *R_n - *R_m$$

Example 3.1.3 (XOR)

assembly:

$$xorR_n, R_m$$

pseudocode:

$$R_n \leftarrow *R_n \oplus *R_m$$

Example 3.1.4 (Compare)

assembly:

 $cmpR_n, R_m$

pseudocode:

 $EFLAGS \leftarrow compare *R_n - *R_m$ **Example 3.1.5** (increment)

assembly:

 $incrementR_n$

pseudocode:

 $R_n \leftarrow *R_n + 1$ **Example 3.1.6** (Move: copy data from register R_n into the absolute address in memory stored in R_m)

assembly:

 $movR_n, R_m$

pseudocode:

 $Mem * R_n \leftarrow *R_m$

Intuition for busses: The address bus carries the memory address from the processor to the memory. The data bus transfers the actual data between the processor and memory. The control bus carries control signals from the processor to coordinate the operations of the memory and I/O devices.

Example 3.1.7 (Move: copy data from the absolute address in memory stored in R_m into register R_n)

assembly:

 $movR_n, R_m$

pseudocode:

 $R_n \leftarrow Mem * R_m$

Intuition for busses: The address bus carries the memory address from the processor to the memory. The data bus transfers the actual data between the processor and memory. The control bus carries control signals from the processor to coordinate the operations of the memory and I/O devices.

Example 3.1.8 (Jump if equals)

assembly:

 $jeqR_n$

pseudocode:

 $if ZF == 1 \text{ then } PC \leftarrow *R_n$

Intuition: read (jump to that address) R_n if the zero flag is 1.

Example 3.1.9 (move num to R_n)

assembly:

 $movR_n, num$

pseudocode:

 $R_n \leftarrow num$

Question 1: Starting Conditions and Execution Flow

Given the following initial program state and instructions:

- PC = 0010
- Mem0010 = mov R0, 1010
- Mem0011 = mov R1, 0001
- Mem0100 = mov R4, 0110
- Mem0101 = mov R5, 1011
- Mem0110 = inc R1
- Mem0111 = cmp R0, R1
- Mem1000 = je R5
- Mem1001 = cmp R0, R0
- Mem1010 = je R4
- Mem1011 = halt

Explain, step by step, how this program executes until **halt**.

Solution: Here is the execution walkthrough:

1. **PC = 0010:** mov R0, 1010. This loads 1010 (binary) into R_0 . Then $PC \leftarrow 0011$.
2. **PC = 0011:** mov R1, 0001. Loads 0001 into R_1 . Then $PC \leftarrow 0100$.
3. **PC = 0100:** mov R4, 0110. Loads 0110 into R_4 . Then $PC \leftarrow 0101$.
4. **PC = 0101:** mov R5, 1011. Loads 1011 into R_5 . Then $PC \leftarrow 0110$.
5. **PC = 0110:** inc R1. Increments R_1 from 0001 to 0010. Then $PC \leftarrow 0111$.
6. **PC = 0111:** cmp R0, R1. Compares R_0 (1010) with R_1 (0010). Since they differ, the zero/equal flag is cleared. Then $PC \leftarrow 1000$.
7. **PC = 1000:** je R5 (jump if equal). The zero flag is 0, so no jump occurs. Then $PC \leftarrow 1001$.
8. **PC = 1001:** cmp R0, R0. Compares R_0 with itself, which sets the zero flag to 1. Then $PC \leftarrow 1010$.
9. **PC = 1010:** je R4. Because the zero flag is 1, we jump to the address stored in R_4 , which is 0110. So $PC \leftarrow 0110$, looping back.
10. The loop repeats from **inc R1** until eventually R_1 equals R_0 (1010). At that point, the compare (**cmp R0, R1**) sets the zero flag, and **je R5** at PC = 1000 finally jumps to $R_5 = 1011$, which contains the **halt** instruction.

This mechanism effectively keeps incrementing R_1 until it matches R_0 , then halts.

3.2 Instruction Encoding for 16-bit Memory Cells

Have been talking about 16-bit memory cells. Thus, our instructions and their parameters need to be able to be described and differentiated in 16-bits.

Encoding Rules

- **Zero-argument instructions**
 - `halt` (only one): 0000 0000 0000 0000
- **Single-argument instructions**
 - First two bits are always 11.
 - Use the first five bits to encode the instruction.
 - Use the remaining eleven bits to represent the argument.
- **Two-argument instructions**
 - Use the first four bits to encode the instruction.
 - Use the next six bits for the first argument.
 - Use the final six bits for the second argument.

3.3 Assemble

To assemble a program is to convert it from human-readable instructions to their binary representations

Instruction	Binary Code	Description	Arguments	Example
halt	0000 0000 0000 0000	halt execution	none	0000 0000 0000 0000
inc R _n	1100 0	register R _n gets (contents of R _n) + 1	11-bit argument representing <i>n</i>	inc R3
jmp num	1100 1	jump to <i>relative</i> address <i>num</i> by adding <i>num</i> to the program counter	11-bit argument representing <i>num</i>	1100 0000 0000 0011 jmp 25
jne num	1101 0	jump to relative address <i>num</i> if last cmp was unequal	11-bit argument representing <i>num</i>	1100 1000 0001 1001 jne 13
je [R _n]	1101 1	jump to <i>absolute</i> address in R _n if last cmp was equal	11-bit argument representing <i>n</i>	1101 0000 0000 1101 je [R3]
add R _n , R _m	0001	R _n gets (R _n + R _m)	Two 6-bit arguments (first = n, next = m)	1101 1000 0000 0011 add R3, R5
sub R _n , R _m	0010	R _n gets (R _n - R _m)	Two 6-bit arguments (first = n, next = m)	0001 0000 1100 0101 sub R3, R2
xor R _n , R _m	0011	R _n gets bitwise XOR of R _n and R _m	Two 6-bit arguments (first = n, next = m)	0010 0010 0100 1000 xor R2, R3
cmp R _n , R _m	0100	compare R _n and R _m ; set flags accordingly	Two 6-bit arguments (first = n, next = m)	0011 0000 1000 0011 cmp R5, R3
mov R _n , num	0101	R _n gets the value <i>num</i>	Two 6-bit arguments (first = n, next = m)	0100 0001 1100 0011 mov R5, 61
mov R _n , R _m	0110	R _n gets the contents of R _m	Two 6-bit arguments (first = n, next = m)	0101 0001 1111 1101 mov R3, R1
mov [R _n], R _m	0111	copy contents of R _m to memory address stored in R _n	Two 6-bit arguments (first = n, next = m)	0110 0001 0100 0001 mov [R3], R1
mov R _n , [R _m]	1000	copy from memory address in R _m to R _n	Two 6-bit arguments (first = n, next = m)	0111 0001 0100 0001 mov R3, [R1]
				1000 0001 0100 0001

Table 3.1: Summary of instructions, binary codes, and examples.

3.4 Binary to Hexadecimal Conversion

Converting binary numbers to hexadecimal is straightforward because each hexadecimal digit represents four binary digits (bits). Here are the steps:

1. **Group the binary digits into sets of four**, starting from the right. Add leading zeros if necessary to make a complete set.
2. **Convert each group of four binary digits to its hexadecimal equivalent** using the following table:

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Example: Convert the binary number 11010110 to hexadecimal.

1. Group the binary digits: 1101 0110.
2. Convert each group:

$$1101 = D, \quad 0110 = 6$$

3. Combine the hexadecimal digits: $D6$.

Thus, $11010110_2 = D6_{16}$.

3.5 Machine Code Decoding

3.5.1 Hexadecimal Machine Code

5000 504A 5094 7040 C000 4042 D7FC 0000

3.5.2 Instruction Breakdown

- **5000**
 - Binary: 0101 0000 0000 0000
 - Instruction: MOV R0, 0
- **504A**
 - Binary: 0101 0000 0100 1010
 - Instruction: MOV R1, 10
- **5094**

- Binary: 0101 0000 1001 0100
- Instruction: MOV R2, 20
- **7040**
 - Binary: 0111 0000 0100 0000
 - Instruction: MOV [R1], R0
- **C000**
 - Binary: 1100 0000 0000 0000
 - Instruction: INC R0
- **C001**
 - Binary: 1100 0000 0000 0001
 - Instruction: INC R1
- **4042**
 - Binary: 0100 0000 0100 0010
 - Instruction: CMP R1, R2
- **D7FC**
 - Binary: 1101 0111 1111 1100
 - Instruction: JNE -4
- **0000**
 - Binary: 0000 0000 0000 0000
 - Instruction: HALT

3.5.3 Program Behavior

1. MOV R0, 0 sets register R0 to 0.
2. MOV R1, 10 sets register R1 to 10.
3. MOV R2, 20 sets register R2 to 20.
4. MOV [R1], R0 writes the contents of R0 into memory at the address held by R1.
5. INC R0 increments R0.
6. INC R1 increments R1.
7. CMP R1, R2 compares R1 and R2.
8. JNE -4 jumps back 4 instructions if R1 is not equal to R2.
9. HALT ends execution.

Summary: This program uses a loop to:

- Increment R0 and R1
- Store R0's value at memory address R1
- Stop once R1 reaches R2 (i.e., 20)

In effect, it writes an increasing sequence of values (0, 1, 2, ...) into consecutive memory locations starting at address 10, and halts once it has written the value 9 at address 19 (because at that point R1 becomes 20, matching R2, so the loop ends).

3.6 Fetch-Decode-Execute-Store Cycle

The fetch-decode-execute-store cycle is the fundamental process by which a computer executes instructions. This cycle involves several key components: the Program Counter (PC), Instruction Register (IR), and various other registers and buses.

3.6.1 Fetch

1. The PC holds the address of the next instruction to be executed. 2. The address is sent to memory via the address bus. 3. The instruction at that address is fetched from memory and loaded into the IR.

3.6.2 Decode

1. The control unit reads the instruction in the IR. 2. The instruction is decoded to determine the operation to be performed and the operands involved.

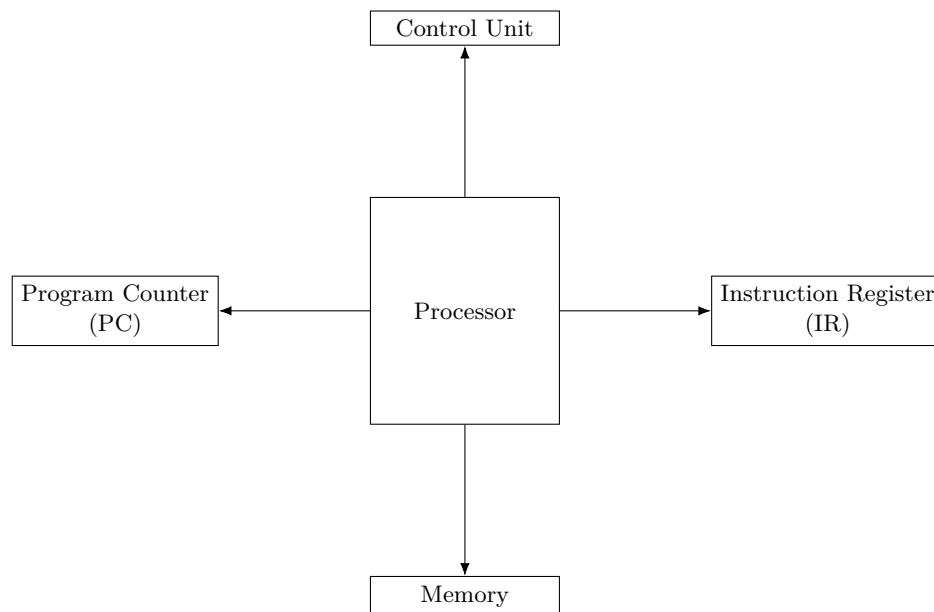
3.6.3 Execute

1. The control unit signals the appropriate components (ALU, registers, etc.) to perform the operation. 2. The ALU or other execution units carry out the operation using the operands.

3.6.4 Store

1. The result of the operation is stored in the appropriate location (register or memory). 2. The PC is updated to point to the next instruction, and the cycle repeats.

Diagram:



This cycle ensures that instructions are processed sequentially and efficiently, allowing the computer to perform complex tasks by breaking them down into simpler operations.

3.7 Five-Stage Pipeline for Operations

The five-stage pipeline is a common technique used in computer architecture to improve instruction throughput. Each instruction passes through five distinct stages: Fetch, Decode, Execute, Memory Access, and Write Back. Below are the details of each stage:

3.7.1 1. Fetch (IF)

Instruction Fetch: - The instruction is fetched from memory. - The Program Counter (PC) holds the address of the instruction to be fetched. - The fetched instruction is stored in the Instruction Register (IR).

Key Points: - The PC is incremented to point to the next instruction. - This stage involves reading from memory, which can introduce delays if the memory access time is significant.

3.7.2 2. Decode (ID)

Instruction Decode: - The fetched instruction is decoded to determine the operation and the operands. - The control unit generates the necessary control signals based on the decoded instruction. - The source operands are read from the register file.

Key Points: - This stage involves interpreting the binary instruction and preparing the necessary data for execution. - The control signals generated will guide the subsequent stages.

3.7.3 3. Execute (EX)

Execution: - The actual operation specified by the instruction is performed. - This could involve arithmetic or logical operations performed by the Arithmetic Logic Unit (ALU). - For branch instructions, the target address is calculated.

Key Points: - The ALU performs the required computation. - The result of the computation is temporarily stored for the next stage.

3.7.4 4. Memory Access (MEM)

Memory Access: - If the instruction involves memory access (e.g., load or store), the memory address is accessed. - For load instructions, data is read from memory and stored in a temporary register. - For store instructions, data is written to the specified memory address.

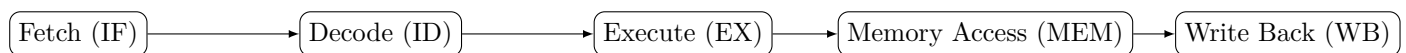
Key Points: - This stage is crucial for instructions that interact with memory. - Memory access can introduce delays due to varying memory access times.

3.7.5 5. Write Back (WB)

Write Back: - The result of the instruction is written back to the register file. - This updates the destination register with the computed value or the data read from memory.

Key Points: - This stage ensures that the results of the instruction are stored for future use. - The pipeline is now ready to process the next instruction.

Diagram:



The five-stage pipeline allows for overlapping execution of multiple instructions, significantly improving the instruction throughput and overall performance of the processor.

3.7.6 Example of parallel processing in 5 stage pipeline

3.8 Stalling and Forwarding in a 5-Stage Pipeline

In a 5-stage pipeline, conflicts can arise when instructions that depend on the results of previous instructions are executed concurrently. Two common methods to overcome these conflicts are stalling and forwarding.

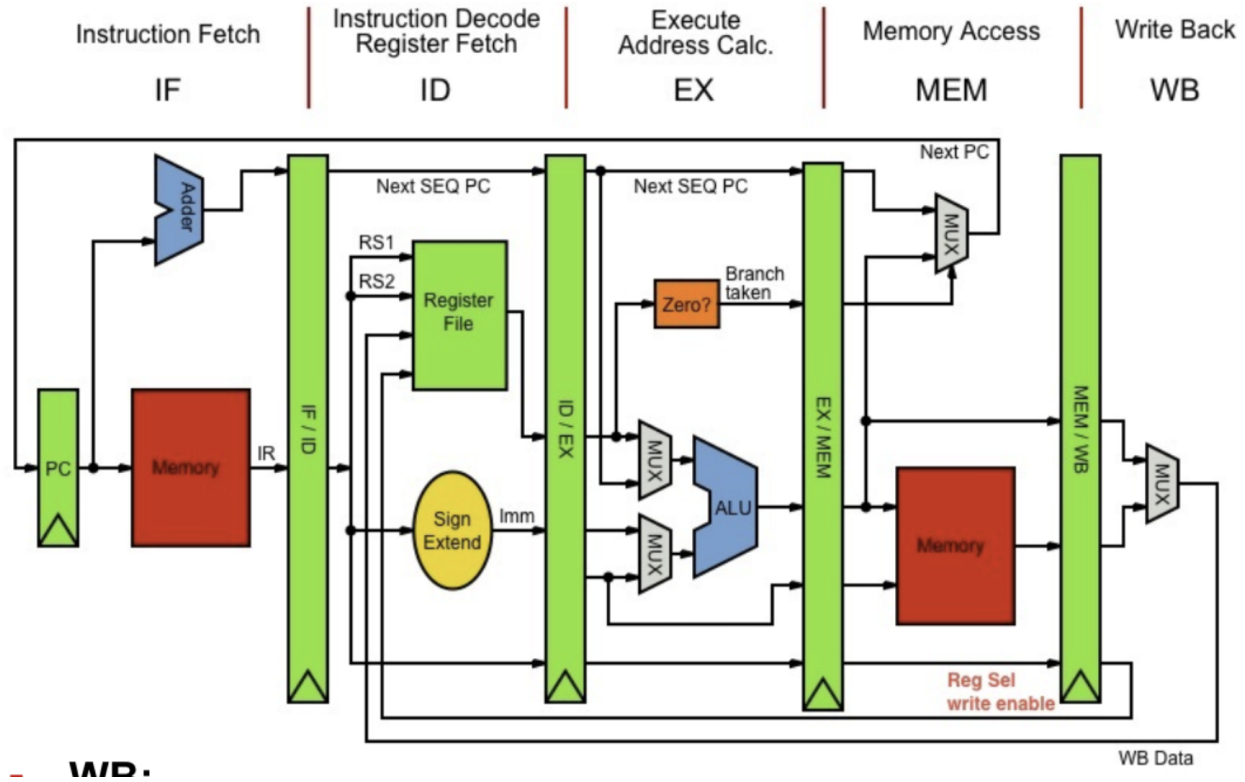


Figure 3.1: Five-Stage Pipeline for Operations

Cycle	IF	ID	EX	MEM	WB
1	MOV R1 [R2]				
2	ADD R3 R4	MOV R1 [R2]			
3	MOV [R5] R6	ADD R3 R4	MOV R1 [R2]		
4		MOV [R5] R6	ADD R3 R4	MOV R1 [R2]	
5			MOV [R5] R6	ADD R3 R4	MOV R1 [R2]
6				MOV [R5] R6	ADD R3 R4
7					MOV [R5] R6

Table 3.2: Pipeline Execution Table

3.8.1 Stalling

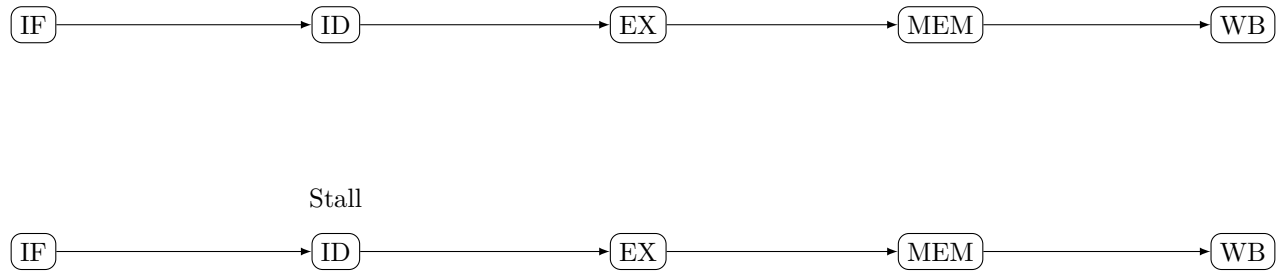
Stalling, also known as pipeline interlock, involves pausing the pipeline until the data required by an instruction is available. This method ensures that instructions are executed in the correct order, but it can reduce the overall performance due to idle cycles.

Example: Consider the following sequence of instructions:

1. MOV R1, [R2]
2. ADD R3, R1

In this case, the ADD instruction depends on the result of the MOV instruction. To resolve this conflict, the pipeline can be stalled until the MOV instruction completes.

Diagram:



3.8.2 Forwarding

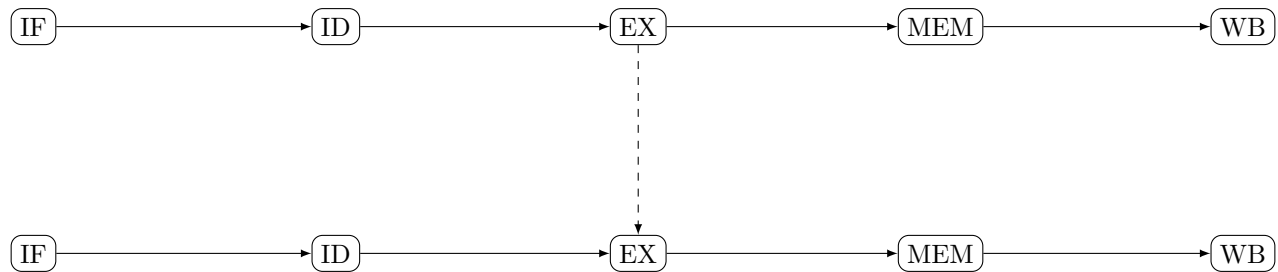
Forwarding, also known as bypassing, involves passing the result of an instruction directly to a subsequent instruction that needs it, without waiting for it to be written back to the register file. This method reduces the number of idle cycles and improves performance.

Example: Consider the following sequence of instructions:

1. MOV R1, [R2]
2. ADD R3, R1

In this case, the result of the MOV instruction can be forwarded directly to the ADD instruction, allowing it to proceed without stalling.

Diagram:



By using stalling and forwarding, the pipeline can handle data hazards and ensure correct execution of instructions while maintaining high performance.

3.9 Handling Simultaneous Memory Reads in IF and WB Stages

In a pipelined processor, simultaneous memory reads in the Instruction Fetch (IF) and Write Back (WB) stages can lead to conflicts and performance degradation. Two common approaches to address this issue are stalling and using the Harvard architecture.

3.9.1 Stalling

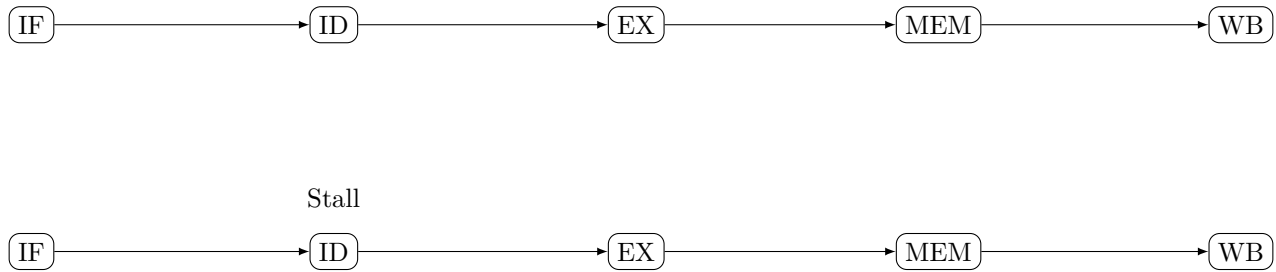
Stalling involves pausing the pipeline to resolve memory access conflicts. When a memory read conflict is detected, the pipeline is stalled until the memory access is completed. This ensures that the correct data is read, but it can reduce overall performance due to idle cycles.

Example: Consider the following sequence of instructions:

1. MOV R1, [R2]
2. ADD R3, R1

If both instructions require memory access at the same time, the pipeline can be stalled to resolve the conflict.

Diagram:

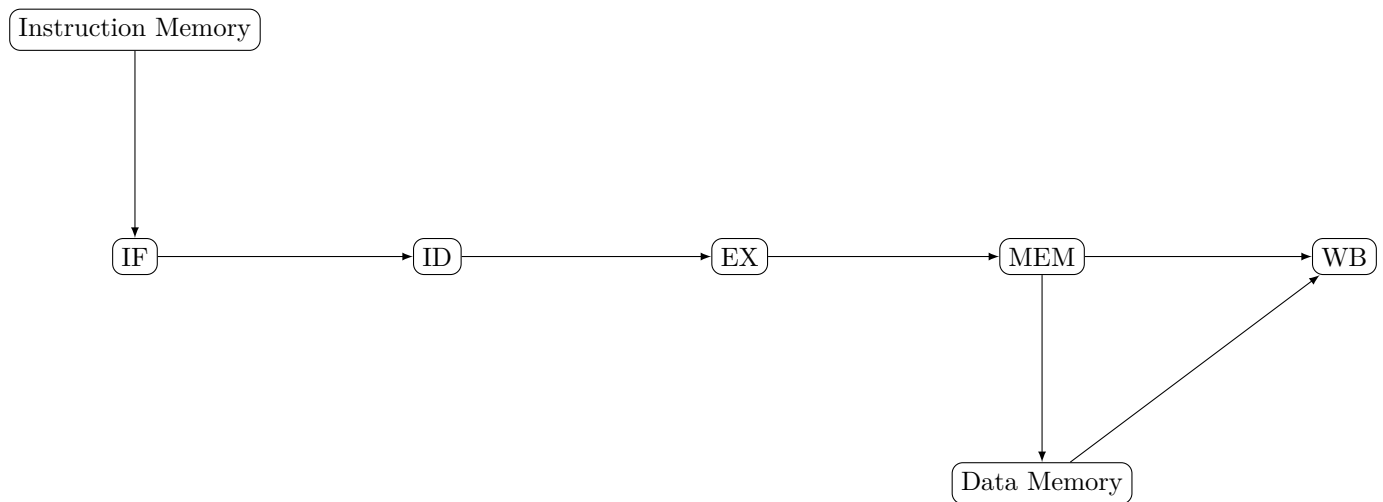


3.9.2 Harvard Architecture

The Harvard architecture addresses memory access conflicts by using separate memory spaces for instructions and data. This allows simultaneous access to both instruction and data memory, eliminating conflicts and improving performance.

Example: In the Harvard architecture, the instruction memory and data memory are accessed independently, allowing the IF stage to fetch instructions while the WB stage writes data.

Diagram:



By using separate memory spaces for instructions and data, the Harvard architecture allows for more efficient memory access and reduces the need for stalling, leading to improved overall performance.

3.10 Memory Pointers

- When we get into C (and C++ later), we will talk a **lot** about memory indirection (pointers).
- But you have already been exposed to the concept with our minimal Instruction Set!
- **What is memory indirection??**
- Which one feels like memory indirection?
 - MOV R1 [0x27]
 - MOV R1 [R2]
- In the first case, we are reading an absolute memory address into **register-1**.
- In the second case, we are reading the memory address specified by the value of **register-2** into **register-1**.
- **The second case is memory indirection, and register-2 is acting as a pointer!**

3.11 Memory Indirection in C++

```
int numbers[] = {10, 20, 30, 40, 50};
int size = sizeof(numbers) / sizeof(numbers[0]);
// Pointer to the first element of the array
int *ptr = numbers;
for (int i = 0; i < size; i++) {
    printf("Element %d: %d\n", i, *(ptr + i));
}
```

Explanation:

This code demonstrates memory indirection using pointers in C. Here's a breakdown of how it works:

1. `int numbers[] = {10, 20, 30, 40, 50};` - An array of integers is declared and initialized with values.
2. `int size = sizeof(numbers) / sizeof(numbers[0]);` - The size of the array is calculated by dividing the total size of the array by the size of one element.
3. `int *ptr = numbers;` - A pointer `ptr` is declared and initialized to point to the first element of the array.
4. `for (int i = 0; i < size; i++) {` - A loop is used to iterate through the array elements.
5. The value of each element is accessed using the pointer `ptr` with memory indirection (`*(ptr + i)`) and printed.

Memory Indirection: - The expression `*(ptr + i)` uses memory indirection to access the value at the memory address pointed to by `ptr` plus `i`. - This allows accessing array elements using pointer arithmetic, demonstrating how pointers can be used to indirectly access memory locations.