University Of Mauritius

Faculty of Information,Communication and Digital Technologies

Exploring the Memory Hierarchy and Architectural Features of Intel Core i7 Processors

Prepared by:

RAGHOONUNDUN Rishab – 2412024

THUMMANAH Kentish – 2411831

SOOBRAYEN Keshia – 2412920

BEEDASSY Nirvana Luxmi – 2413850

Course: BSc (Hons) Computer Science – Level 1

Module: ICT1206Y Computer Organisation and Architecture

Lecturer: Dr. Razvi Doomun

Date: 27th March 2025

Table of Contents

1.0 Introduction

The Intel Core i7 is a line of high-performance microprocessors developed by Intel, a leading technology company. It belongs to the Intel Core series and is designed for desktop and laptop computers. The Intel Core i7 processors are known for their powerful performance, advanced features, and multitasking capabilities. They offer excellent speed and can handle demanding tasks such as gaming, video editing and software development with ease. The latest generation of Intel Core i7 processors is the 14th generation. Moreover, with each new generation, Intel has introduced architectural improvements, increased core counts, and enhanced cache sizes to ameliorate processing capabilities and energy efficiency.

Furthermore, the purpose of this report is to analyse the architectural aspects of Intel Core i7 processors, focusing on memory hierarchy, cache organisation, addressing modes, system bus architecture, and pipelining mechanisms. The report also includes a practical simulation to demonstrate key features.

2.0 Memory Hierarchy Analysis

2.1 Overview of Memory Hierarchy

Intel Core i7 processors tend to have a memory hierarchy structured for both maximum data access speed and maximum storage capacity decisions that balance the two. The memory hierarchy of the Intel Core i7 processors is hierarchical meaning that each level of the hierarchy ranging from CPU registers to main memory where faster and smaller memory is accessed first. The memory hierarchy consists of :

1. CPU registers

Small and high-speed memory units located in the CPU. They are used to store the most frequently used data and instructions. Registers have the fastest access time and the smallest storage capacity typically ranging from 16 to 64 bits.

2. L1 Cache

The L1 cache is the closest level of cache to the CPU, it is also known as the primary cache. It is extremely fast but relatively small. The L1 cache is split into the Instruction Cache (L1i) and Data Cache (L1d), typically 32 KB each.

3. L2 Cache

The L2 cache is also known as the secondary cache typically 256 KB per core. When the L1 cache becomes overloaded , the L2 cache acts as the secondary cache, holding larger blocks of data. Hence it is often larger than the L1 cache.

4. L3 Cache

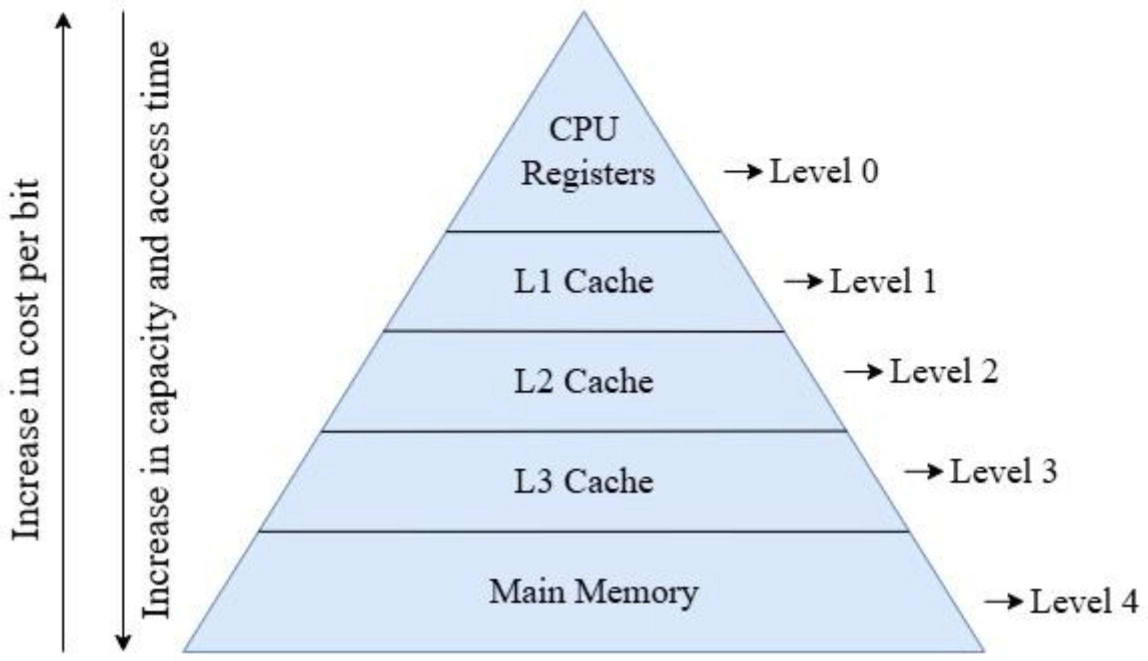
The L3 cache is also known as the last-level cache (LLC). It is a larger (up to 30 MB), shared cache memory located in the CPU. The L3 cache is slower than L1 and L2 caches but larger in capacity and is shared by all the cores. It helps maintain consistency among the cores and improves performance by storing frequently accessed data and instructions which speeds up data access times.

5. Main Memory

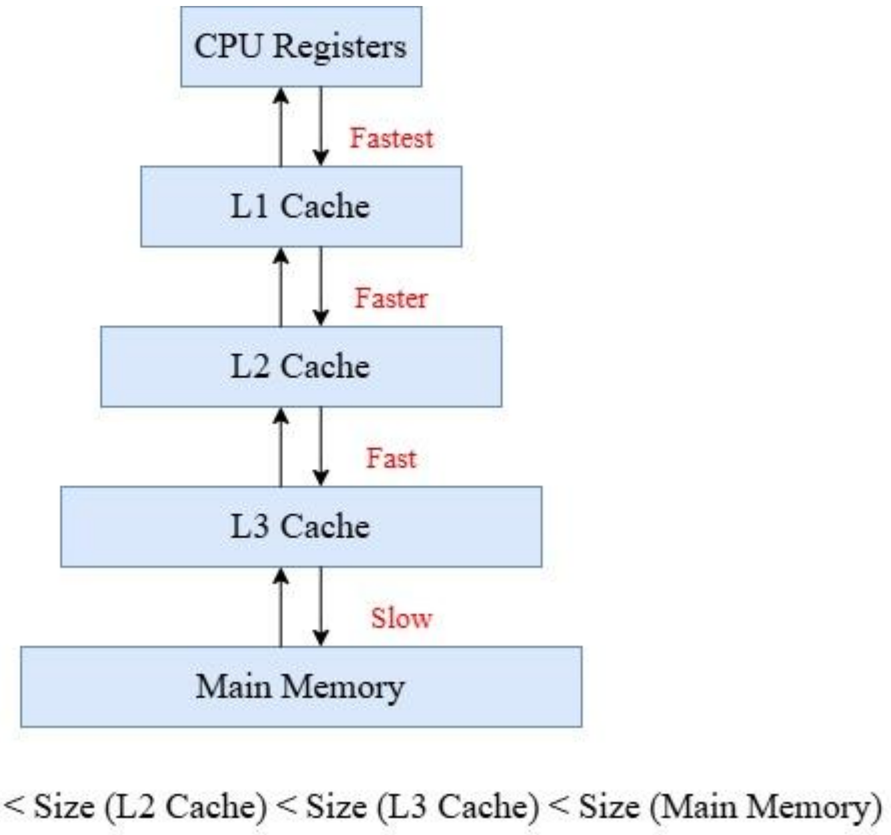
The Main Memory (RAM) also known as the primary memory is a volatile memory that provides fast storage and retrieval of data. It temporarily stores data that is being executed by the CPU. It has a storage capacity ranging from 4 GB to 16 GB.

2.2 Memory Hierarchy Levels

Memory hierarchy diagram



Hierarchical Access Memory Organisation Diagram


CPU Registers
Fastest
L1 Cache
Faster
L2 Cache
Fast
L3 Cache
Slow
Main Memory
Size (L1 Cache) < Size (L2 Cache) < Size (L3 Cache) < Size (Main Memory)

2.2.1 Roles of different levels in the Memory Hierarchy

1. Level 0

The CPU Registers hold immediate data and instructions that the CPU is actively processing. It also stores operands for arithmetic, logical and control operations. For example, the addition of two numbers is stored temporarily.

2. Level 1

The L1 Cache fetches and stores the most frequently used instructions and data. It also serves as the first point of contact when the CPU requests data. In addition, the L1 cache minimises latency by delivering information at CPU speeds. For example, storing loop variables or repeatedly used instructions.

3. Level 2

The L2 Cache is a backup for the L1 cache, it stores less frequently used data. It pre-fetches data from L3 cache or main memory to reduce access time. Furthermore, it improves performance by reducing main memory access. For example, storing data that is not immediately needed by likely to be re-used.

4. Level 3

The L3 Cache is a shared cache across multiple CPU cores. It stores larger blocks of data not present in L1 or L2 caches. It also reduces the number of main memory accesses for all cores. For example, storing shared data used by multiple cores, like system-level information.

5. Level 4

The Main Memory stores active programs and data required during execution. It supplies data to the CPU when it is not found in any cache. It also acts as a primary working area for the operating system and applications. For example, holding running programs like web browsers, games.

2.2.2 Impact on performance

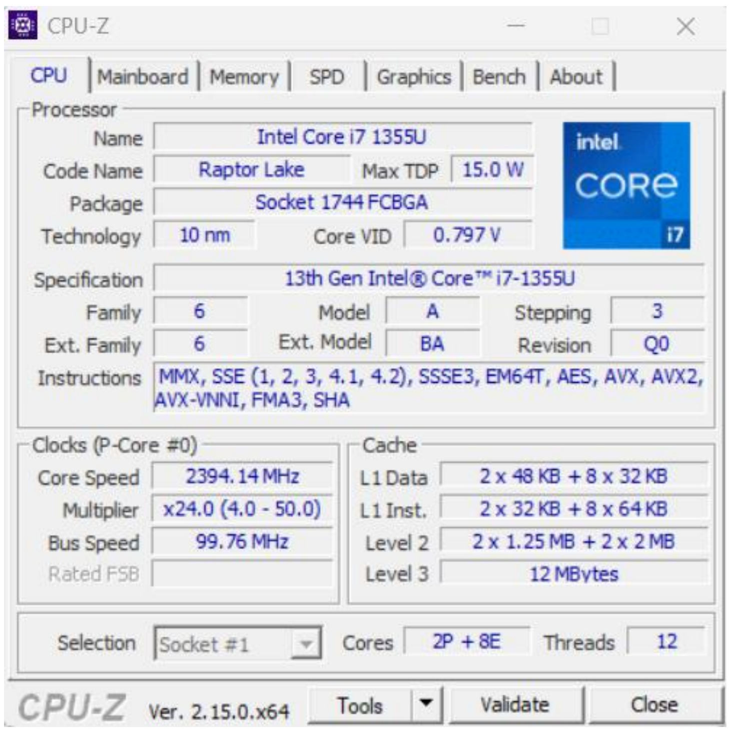
Memory Level	Access Time	Impact on performance
CPU Registers	Approx 1 clock cycle (Fastest)	Max performance (no delay).
L1 Cache	Approx 8 clock cycles	Fast execution, minimal CPU waiting.
L2 Cache	Approx 12 - 20 clock cycles	Balanced speed and size for workloads.
L3 Cache	Approx 30 - 45 clock cycles	Slower but improves data availability.
Main Memory	Approx 100+ clock cycles (Slowest)	Significant delay if cache misses.

3.0 Cache Design and Mapping Techniques

3.1 Cache Design

Consider the laptop ASUS-Vivobook 15 OLED (X1505) with processor 13th Gen Intel(R) Core(TM) i7-1355U.

Using CPU-Z,



CPU-Z

CPU | Mainboard | Memory | SPD | Graphics | Bench | About

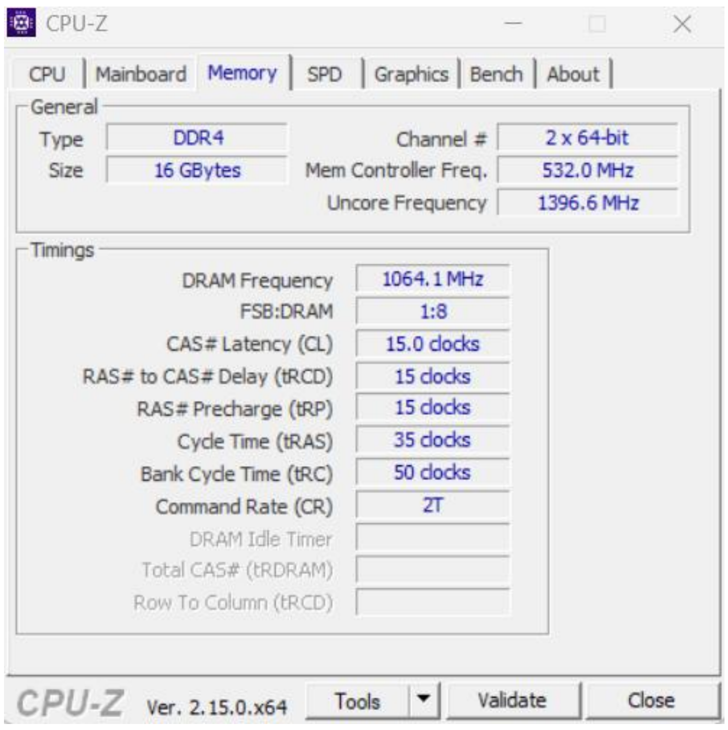
Processor

Name	Intel Core i7 1355U				
Code Name	Raptor Lake	Max TDP	15.0 W		
Package	Socket 1744 FCBGA				
Technology	10 nm	Core VID	0.797 V		
Specification	13th Gen Intel® Core™ i7-1355U				
Family	6	Model	A	Stepping	3
Ext. Family	6	Ext. Model	BA	Revision	Q0
Instructions	MMX, SSE (1, 2, 3, 4.1, 4.2), SSSE3, EM64T, AES, AVX, AVX2, AVX-VNNI, FMA3, SHA				

Clocks (P-Core #0)		Cache	
Core Speed	2394.14 MHz	L1 Data	2 x 48 KB + 8 x 32 KB
Multiplier	x24.0 (4.0 - 50.0)	L1 Inst.	2 x 32 KB + 8 x 64 KB
Bus Speed	99.76 MHz	Level 2	2 x 1.25 MB + 2 x 2 MB
Rated FSB		Level 3	12 MBytes

Selection Socket #1 Cores 2P + 8E Threads 12

CPU-Z Ver. 2.15.0.x64 Tools Validate Close



CPU-Z

CPU | Mainboard | Memory | SPD | Graphics | Bench | About

General

Type	DDR4	Channel #	2 x 64-bit
Size	16 GBytes	Mem Controller Freq.	532.0 MHz
		Uncore Frequency	1396.6 MHz

Timings

DRAM Frequency	1064.1 MHz
FSB:DRAM	1:8
CAS# Latency (CL)	15.0 clocks
RAS# to CAS# Delay (tRCD)	15 clocks
RAS# Precharge (tRP)	15 clocks
Cycle Time (tRAS)	35 clocks
Bank Cycle Time (tRC)	50 clocks
Command Rate (CR)	2T
DRAM Idle Timer	
Total CAS# (tRDRAM)	
Row To Column (tRCD)	

CPU-Z Ver. 2.15.0.x64 Tools Validate Close

From the above screenshots, it can be deduced that :

1. The CPU consists of 10 cores (8 efficiency cores and 2 performance cores).

2. L1 Data cache (L1d) = (2 * 48 KB) + (8 * 32 KB) = 352 KB

   L1 Instruction cache (L1i) = (2 * 32 KB) + (8 * 64 KB) = 576 KB

   L1 cache = L1d + L1i = 352 KB + 576 KB = 928 KB

3. L2 cache = (2 * 1.25 MB) + (2 * 2 MB) = 6.5 MB = 6656 KB

4. L3 cache = 12 MB = 12 288 KB

5. Main Memory (DDR4) = 16 GB

Total cache size for all 10 cores = L1 cache + L2 cache + L3 cache

= 928 KB + 6656 KB + 12 288 KB

= 19 872 KB

Block size = 4 KB = 4096 bytes

3.2 Cache Mapping Techniques

Cache Mapping is a technique by which the contents of main memory are brought into the cache memory (Doomun 2025). In other words, it refers to the technique used to store and use data in cache memory thus creating the mapping between memory addresses and cache locations (Kumar 2024). There are three types of cache mapping techniques mainly:

1. Direct Mapping
2. Fully Associative Mapping
3. Set Associative Mapping

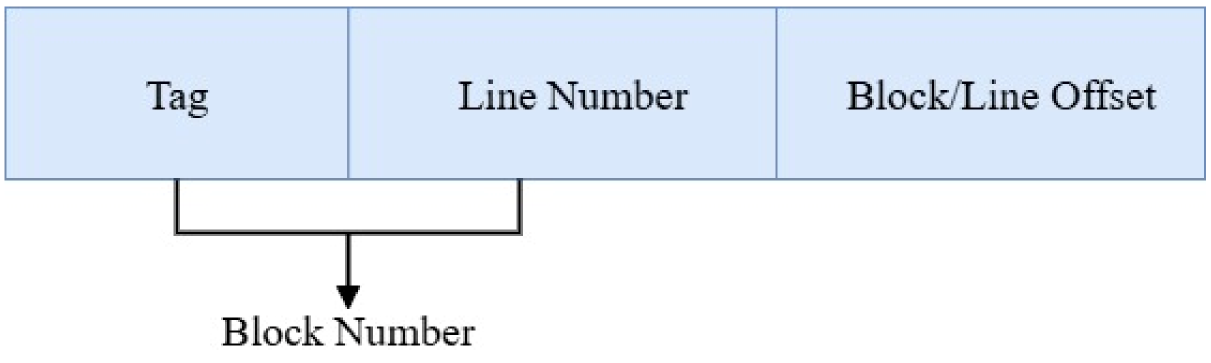
3.2.1 Direct Mapping

This technique involves a particular block of main memory which maps only to a particular line of the cache (Doomun 2025). Memory blocks are mapped to cache lines using a hashing or indexing algorithm.

In direct mapping, the physical address is divided into three components:

- Tag of the line is used to compare with the tag field of the CPU address to determine a cache hit or miss.
- Index bits are used to determine the cache line to which the memory block is mapped to.
- Block/Line Offset is used to identify the specific byte within a cache line that is being accessed.

<u>Division of physical address in Direct Mapping</u>



3.2.2 Fully Associative Mapping

This technique involves a block of main memory which maps to any freely available cache line (Doomun 2025). Memory blocks are mapped to cache lines using a replacement algorithm such as First Come First Serve or Least Recently Used algorithms.

In this mapping, the physical address is divided into two components:

- Tag is used to specify a block of main memory which is currently stored in a particular cache line.
- Block/Line Offset is used to identify the specific byte within a cache line where the requested data is situated.

Division of physical address in Fully Associative Mapping

Block Number/Tag	Block/Line Offset

3.2.3 K-Way Set Associative Mapping

Set associative mapping in cache systems is a compromise between direct and fully associative mapping, where the cache is divided into sets, and each memory block maps to a specific set but can be placed in any line within that set.

In this mapping, the physical address is divided into three components:

- Tag is used to specify a block of main memory which is currently stored in a particular cache line within a given set.
- Set Number bits decide that in which set of the cache the required block is present.
- Block/Line Offset is used to identify the exact byte within a cache line where the requested data is located.

Division of physical address in K-Way Set Associative Mapping

Tag	Set Number	Block/Line Offset

3.3 Mapping Techniques in Intel Core i7 Processors

In i7 processors in general, a multi-level hierarchy with three levels of caches are used L1, L2 and L3 (Bharatiya 2023):

- L1 Cache uses a 4-way set-associative cache mapping.
- L2 Cache uses an 8-way set associative cache mapping.
- L3 Cache uses a 12-way set associative cache mapping.

3.4 Cache Replacement Policies

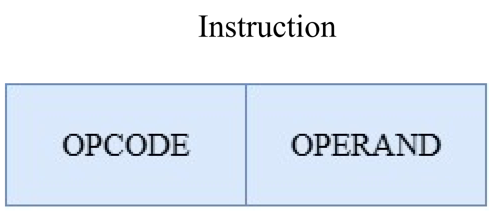
Cache replacement policies are algorithms that optimize instructions that a computer program uses to manage the stored cache information (Hyperskill n.d). There are three types of cache replacement policies:

- Random Replacement policy selects an item from the storage at random and discards it to make space for new incoming information thus keeping no information about how often a cache is used (Hyperskill n.d).
- First In, First Out is an algorithm that keeps of the buffer history and discards the information in order they were added without considering how often they were used (Hyperskill n.d).
- Least Recently Used is an algorithm that keeps track of access history of the information and discards information based on how least often they were used to make space for new ones (Hyperskill n.d).

4.0 Addressing Modes and Enhancing Processor Efficiency

4.1 Addressing Modes

Addressing modes are techniques used by the CPU to get the address of a specific data in an operation. They also define how the CPU calculates the Effective Address of an operand.



4.1.1 Implied Addressing Mode

In this addressing mode the operand is implicitly specified by the instruction itself. That is the CPU automatically knows where to fetch the operand. It is also known as Implicit addressing mode.

Example: DCR Ri ;Decrements the value in register Ri by 1.

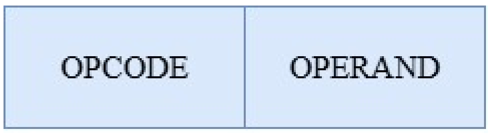
Operation: Ri ← Ri – 1

4.1.2 Immediate Addressing Mode

The operand is explicitly specified in the instruction, rather than being fetch from a register or a location.

Example: ADD Ri, #10 ;Adds the value 10 to the value in the register Ri.

Operation: Ri ← Ri + 10

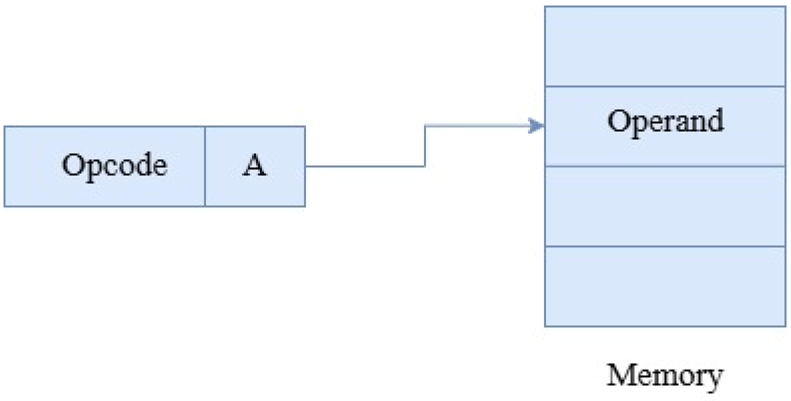


4.1.3 Direct Addressing Mode

In this memory addressing technique, the memory address of the operand is mentioned in the instruction itself. Only 1 reference to memory is required to fetch the operand. It is also known as absolute addressing mode.

Example: LOAD Ri, 2000 ;Loads the value from memory address 2000 into register Ri.

Operation: Ri ← M[2000]

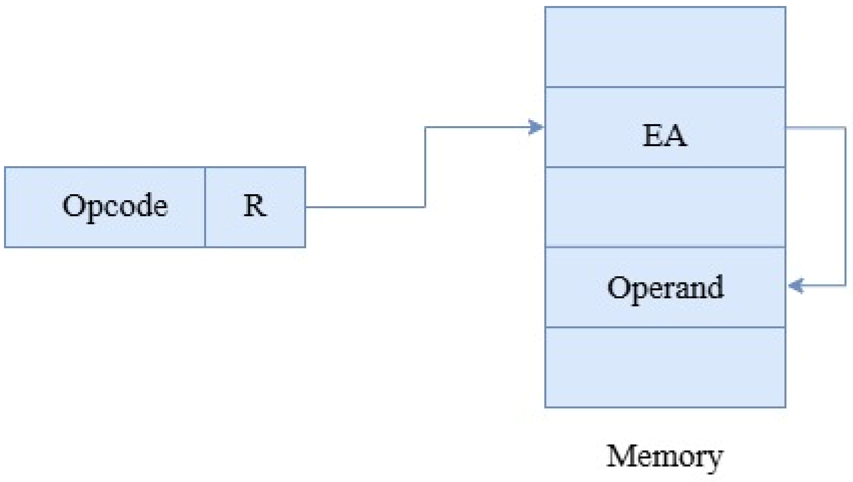


4.1.4 Indirect Addressing Mode

It is a technique where a register holds the memory address of the operand, rather than the operand itself. That is, the CPU fetches the data by looking at the address stored in the register. Two references to memory are required to fetch the operand.

Example: LOAD Ri, (100) ;Loads the value stored at the memory address found in location 100 into register Ri.

Operation: Ri ← M[M[100]]

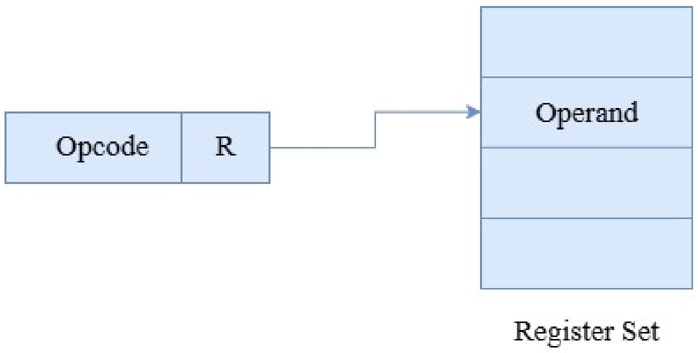


4.1.5 Register Direct Addressing Mode

In this addressing mode, the operand is stored in a register and the instruction directly refers to that register. No reference to memory is required to fetch the operand.

Example: ADD Ri, Rj ;Adds the value in register Rj to the value in Ri and stores the result in Ri.

Operation: Ri ← Ri + Rj

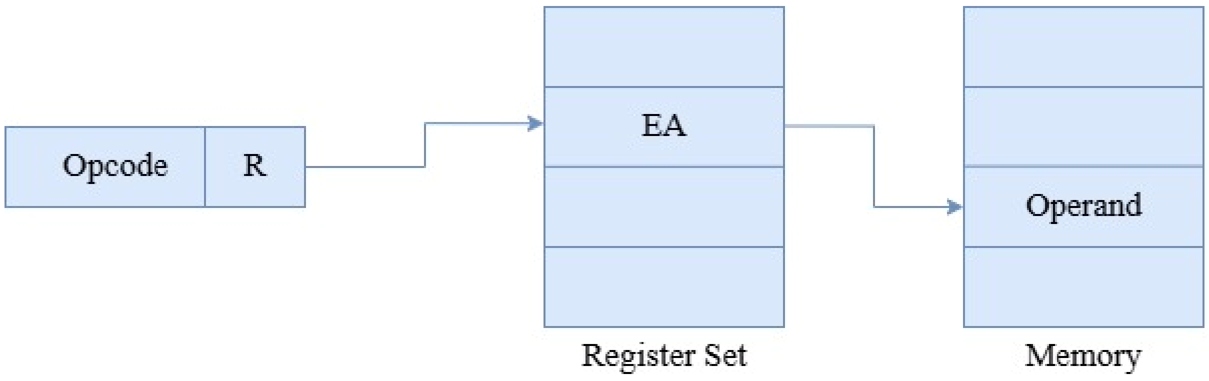


4.1.6 Register Indirect Addressing Mode

In this technique, rather than being directly specified in the instruction, the memory address of the operand is stored in a register. Only one reference to memory is required to fetch the operand.

Example: LOAD Ri, (Rj) ;Loads the value from the memory location stored in register Rj into register Ri.

Operation: Ri ← M[Rj]

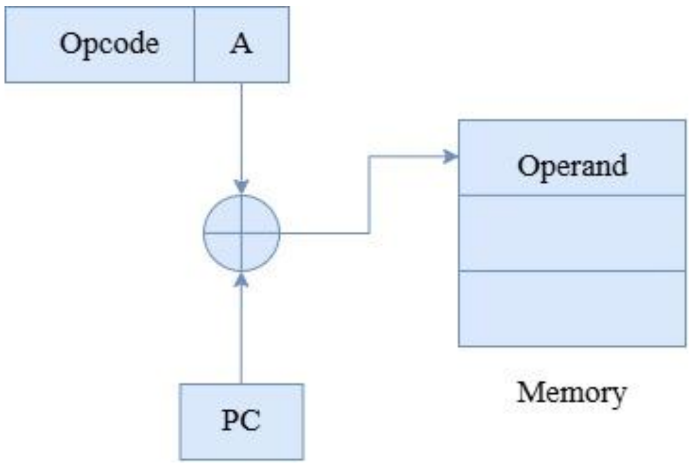


4.1.7 Relative Addressing Mode

The content of the program counter is added to the address part of the instruction to get the effective address of the operand.

Example: JUMP 1000(PC) ;Changes the Program Counter (PC) to PC + 1000, jumping to a new instruction relative to the current position.

Operation: PC ← PC + 1000

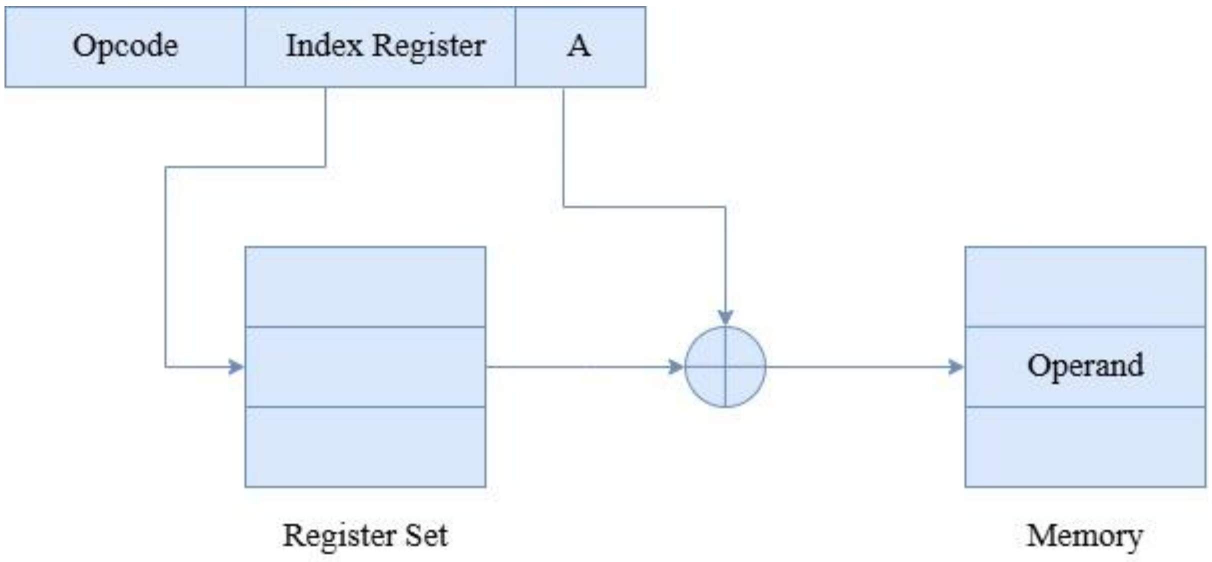


4.1.8 Indexed Addressing Mode

The content of the index register is added to the address part of the instruction to get the effective address of the operand.

Example: LOAD Ri, (Base + Index) ;Loads the value from the memory address calculated as Base + Index into register Ri, often used in arrays.

Operation: Ri ← M[Base + Index]

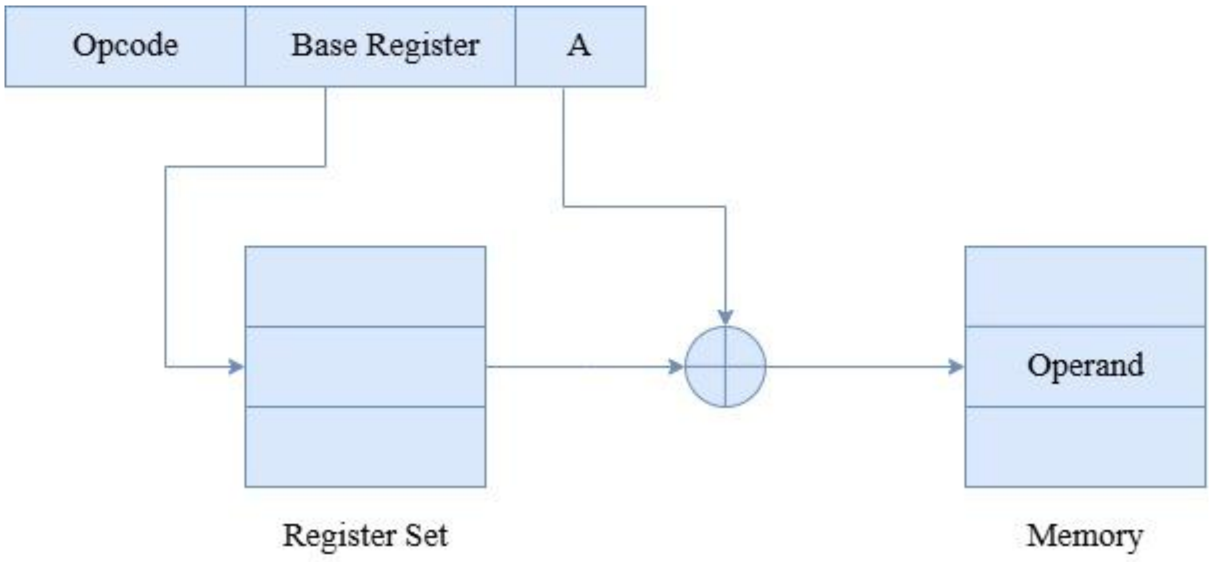


4.1.9 Base Register Addressing Mode

The content of the base register is added to the address part of the instruction to get the effective address of the operand.

Example: LOAD Ri, (Rb + Offset) ;Loads the value from the memory address Rb + Offset into Ri, where Rb is the base register and Offset is a fixed value.

Operation: Ri ← M[Rb + Offset]



4.2 How Addressing Modes Enhances Processor Efficiency

1. Implied Addressing Mode

- Since operands are pre-defined, the instructions are smaller and it enhances processor efficiency by making execution faster (Patterson & Hennessy, 2017).
- Smaller instructions require fewer bytes and no extra memory fetches are needed (Tanenbaum, 2016).
- Simplified decoding means the control unit requires less logic, therefore reducing hardware complexity (Clements, 2014).

2. Immediate Addressing Mode

- Since there is no need to retrieve data from memory, it reduces the number of cycles needed to fetch the operand (Patterson & Hennessy, 2017). This leads to faster execution.
- It reduces latency. The value is part of the instruction (e.g: MOV R1 #13), avoiding a separate memory load (Intel, 2023).

3. Direct Addressing Mode

- It reduces general purpose register usage by directly fetching from memory (Hennessy & Patterson, 2019).
- Fixed addresses allows better cache utilization (Patterson & Hennessy, 2017).
- No complex address calculation is needed (Clements, 2014).

4. Indirect Addressing Mode

- Instead of going through full memory addresses, the CPU only needs to specify a register, which reduces instruction size (Tanenbaum, 2016).
- Useful for jump tables (Hyde, 2010).
- Reduced program size. One instruction can access multiple locations indirectly (ARM, 2022).

5. Register Direct Addressing Mode

- It helps improve performance and efficiency by using less memory access (Hennessy & Patterson, 2019).
- Lower power consumption as no memory bus activity is needed (Patterson & Hennessy, 2017).

- Several register operations can execute simultaneously (Intel, 2023).

6. Register Indirect Addressing Mode

- Since registers are much faster than memory, it reduces access time (Hennessy & Patterson, 2019).
- Registers take fewer bits to encode than full memory addresses (ARM, 2022).
- Optimal for traversing arrays (Hyde, 2010).

7. Relative Addressing Mode

- It allows faster branching. Instead of multiple CMP and JMP instruction, a jump table allows direct jumps (Patterson & Hennessy, 2017).
- It reduces instruction size compared to absolute addresses (ARM, 2022).

8. Indexed Addressing Mode

- It reduces code complexity by using built-in indexing mechanisms instead of manually computing addresses (Hyde, 2010).
- Since the base address remains static while the index updates, it reduces overhead (Intel, 2023).

9. Base Register Addressing Mode

- It improves loop performance by removing repeated address calculations (Hennessy & Patterson, 2019).
- Useful for accessing local variables in function calls (Patterson & Hennessy, 2017).

5.0 System Bus Architecture

5.1 Introduction to System Bus in Modern Core i7

The system bus in Intel Core i7 processors progressed from standard parallel buses such as Front-Side Bus to integrated high-speed interconnects in their architecture. Core i7 CPUs in modern times use address, data and control signaling together with embedded protocols. For instance DMI, PCIe and memory channels (Intel, 2023).

Key buses in Core i7 Architecture:

1. Address bus

It enables the transportation of memory locations from the CPU to RAM and input/output units. Addditionally, it specifies the location from which certain information is to be fetched or to which information is to be stored within the domain. The maximum quantity of addressable memory is, to some extent, determined by the width of the address bus (Intel Corporation, 2023, p. 45).

2. Data bus

This bus facilitates the transfer of real information between the CPU, memory, and peripheral devices. A single cycle data transfer from Modern Core i7 CPUs operates through 64-bit data buses that enhance memory channel bandwidth (Patterson & Hennessy, 2017). The system bandwidth and overall performance increases when data bus width gets expanded (Stallings, 2022).

3. Control bus

This system portion handles the essential commands and signaling functions which enable CPU connectivity to other system elements (Intel Corporation, 2023). Such signals include both read/write commands and interrupts with clock timing signals between the CPU and other system components (Stallings, 2022). The control bus facilitates proper timing execution and system part synchronization thus preventing data corruption and operational conflicts (Patterson & Hennessy, 2017).

5.2 Roles of Different Buses in Intel Core i7 Processors

1. Address bus

The address bus in Intel Core i7 processors uses Physical Address Extensions (PAE) to enable access to extensive memory spaces while optimizing the 64-bit architectural limitations (Jacob, 2020). Modern Core i7 architectures incorporate the memory controller directly on the processor die, significantly reducing latency by removing traditional chipset intermediaries that were characteristic of earlier front-side bus designs (Drepper, 2020). This integrated approach supports memory-mapped I/O (MMIO) which is essential for high-speed communication with PCIe 4.0 and 5.0 devices such as GPUs and NVMe SSDs (Hennessy et al., 2019).

2. Data bus

Intel Core i7 processors feature an advanced 64-bit data bus system that enables fast, parallel data transfer between CPU components, memory, and peripherals (Intel Corporation, 2023; Hennessy et al., 2019). DDR memory channels and PCIe lanes create efficient pathways for RAM, storage, and GPU connectivity (Maynard et al., 2022). The Direct Media Interface (DMI) optimizes communication between the processor and chipset for I/O devices (Sewell et al., 2021). This architecture reduces bottlenecks, boosts system throughput, and improves multitasking performance by supporting simultaneous data transfers with expanded bandwidth (Drepper, 2020; Jacob, 2020).

3. Control bus

In Intel Core i7 processors, the control bus moves signals between the CPU, memory, and other components to manage internal operations (Intel Corporation, 2023). It directs data transfers, handles interrupt requests, and manages synchronization and read/write commands (Sorin et al., 2021). The control bus ensures proper instruction delivery for data fetching, execution timing, and device requests (Bhattacharjee, 2022). Synchronizing clock signals across components is key to maintaining task execution timing (Litz et al., 2020). Optimized control systems in Core i7 processors improve performance, reduce delays, and enhance overall functionality (Maynard et al., 2022).

6.0 Pipelining and Performance

6.1 Pipelining Mechanism in Intel Core i7 Processors

Pipelining consists of executing multiple instructions in parallel within a single processor. In Intel Core i7 processors, execution of instructions is divided into several stages, including (GeeksforGeeks 2021):

- Fetch involves retrieving instruction from memory.
- Decode interprets the instruction.
- Execute performs the operation specified by the instruction.
- Memory Access accesses memory if the instruction requires it.
- Write Back means writing the result back to a register or memory.

Figure of 5-Stage Pipeline (Eswer, Varuna & Dessai, Sanket. 2021)

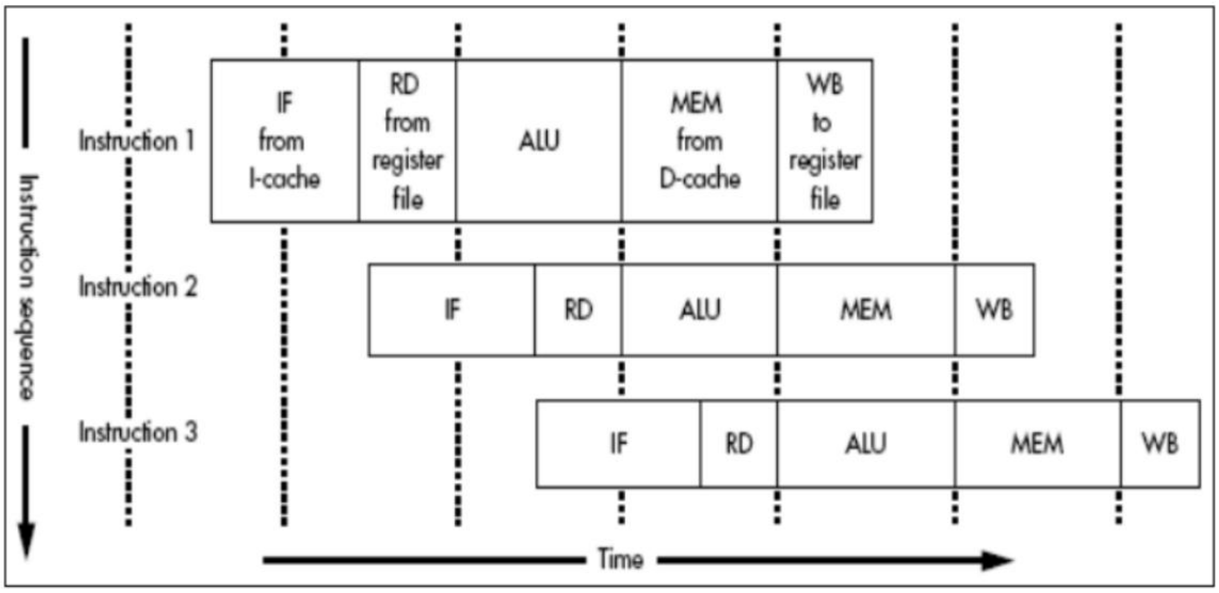


Figure 1

In Intel Core i7 processors, pipelining uses dynamic multiple issue, dynamic scheduling, out-of-order execution, and speculation across 14 stages instead of the traditional 5 (ResearchGate, 2023). Instructions are retrieved, translated into micro-operations, and executed at rates measured in MIPS. A dynamic pipeline manages these micro-operations, sustaining up to six executions per clock cycle (ResearchGate, 2023).

Block Diagram of Pipeline Functionality of the Skylake Microarchitecture

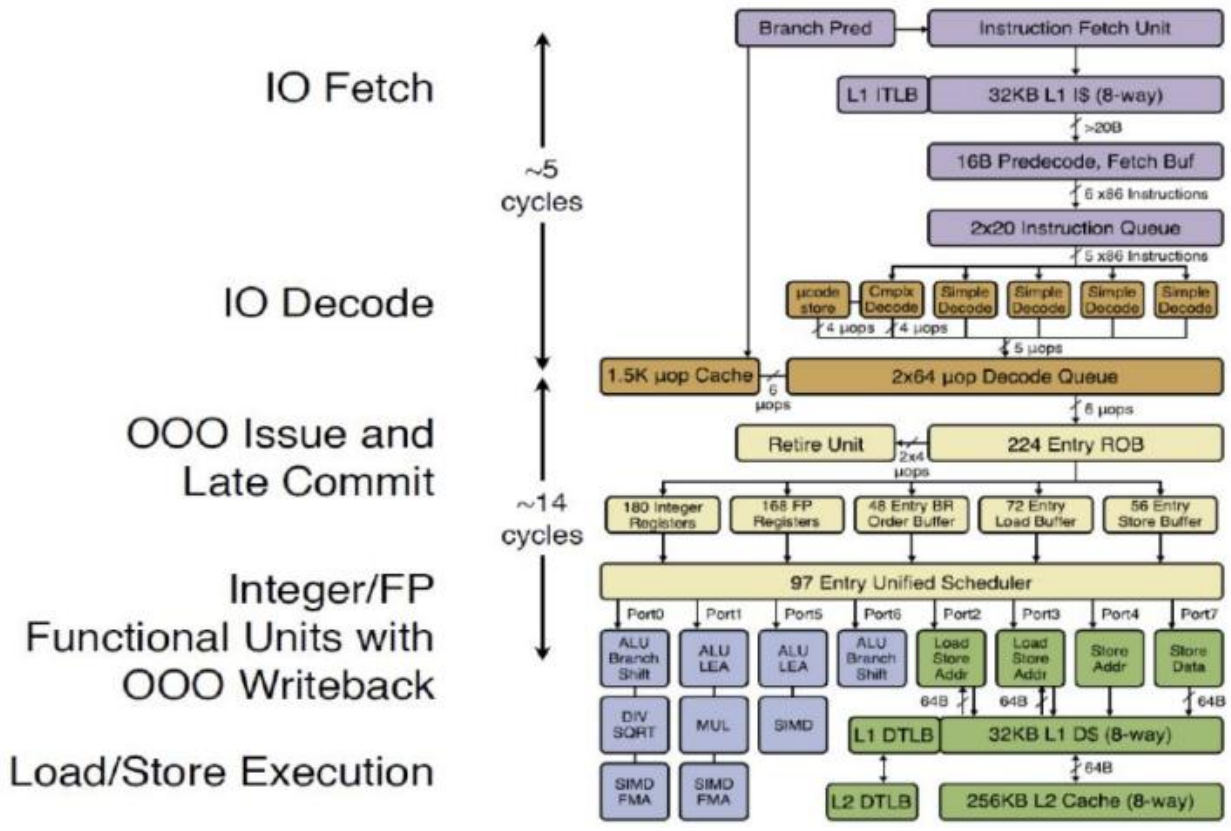


Figure 2

6.2 Challenges and Techniques to overcome Pipelining Hazards

Challenges such as read-after-read and write-after-read hazards are hazards that occur when a later instruction writes to a register or memory location that a previous instruction reads which conclude as lost update of data.

To overcome this hazard, in i7 processors:

- The execution part contains a component called "Renamer" that moves micro operations from the front end to the execution core (ResearchGate 2023).
- A Reorder buffer (ROB) restrains micro operations in various stages of completion, buffers completed micro operations, updates the architectural state in order and manages ordering of exceptions thus preventing data hazards (ResearchGate 2023).

Moreover, other challenges involve the microfusion in the fourth step of pipelining combines the micro operation pairs such as load of ALU operation and ALU operation and directs them to a reservation station, thus increasing the load of the buffer (ResearchGate 2023).

To overcome this hazard, in i7 processors:

- Macro operation fusion, which is beyond microfusion, combines clusters of x86 instructions into a single operation thus reducing the instruction count and decrease the pressure on the pipeline’s resources (ResearchGate 2023).

6.3 How Pipelining Enhances Processor Efficiency

In Intel Core i7 processors, the performance of the processor can be boosted with pipelining which reduces latency (TutorChase). The processor is not idle while waiting for the next instruction, thus decreases execution time. Moreover, all components of the processor are used at the same time as the next instruction is fetched while the current one is decoded or executed (TutorChase).

7.0 Practical Component

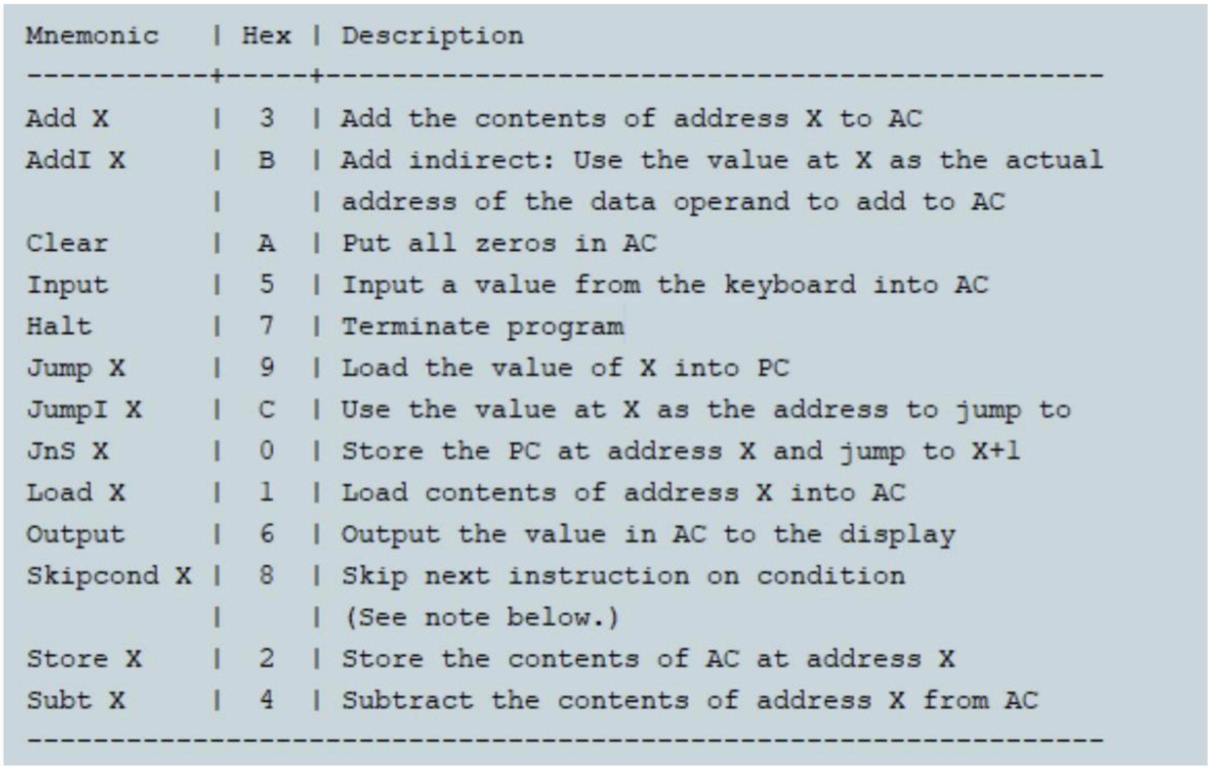
7.1 MARIE Simulator

The MARIE Simulator is a simple, educational CPU simulator used to teach and understand basic computer architecture concepts, including memory addressing, CPU instructions, and program execution. It is part of the MARIE Assembly Language Environment (MARIE-Sim), designed to provide hands-on learning about how a computer’s CPU, memory, and instructions interact.

Key Features:

- Visual representation of memory, registers, and output windows.
- Allows step-by-step execution to observe how instructions manipulate memory and registers.
- Supports basic addressing modes like Direct, Indirect, and Implied Addressing.

It supports basic instructions like:



```
Mnemonic    | Hex | Description
-----------+-----+-----------------------------------------------
Add X       |  3  | Add the contents of address X to AC
AddI X      |  B  | Add indirect: Use the value at X as the actual
            |     | address of the data operand to add to AC
Clear       |  A  | Put all zeros in AC
Input       |  5  | Input a value from the keyboard into AC
Halt        |  7  | Terminate program
Jump X      |  9  | Load the value of X into PC
JumpI X     |  C  | Use the value at X as the address to jump to
JnS X       |  0  | Store the PC at address X and jump to X+1
Load X      |  1  | Load contents of address X into AC
Output      |  6  | Output the value in AC to the display
Skipcond X  |  8  | Skip next instruction on condition
            |     | (See note below.)
Store X     |  2  | Store the contents of AC at address X
Subt X      |  4  | Subtract the contents of address X from AC
-----------------------------------------------------------------
```

7.1.1 Installation and Configuration

1. Installation

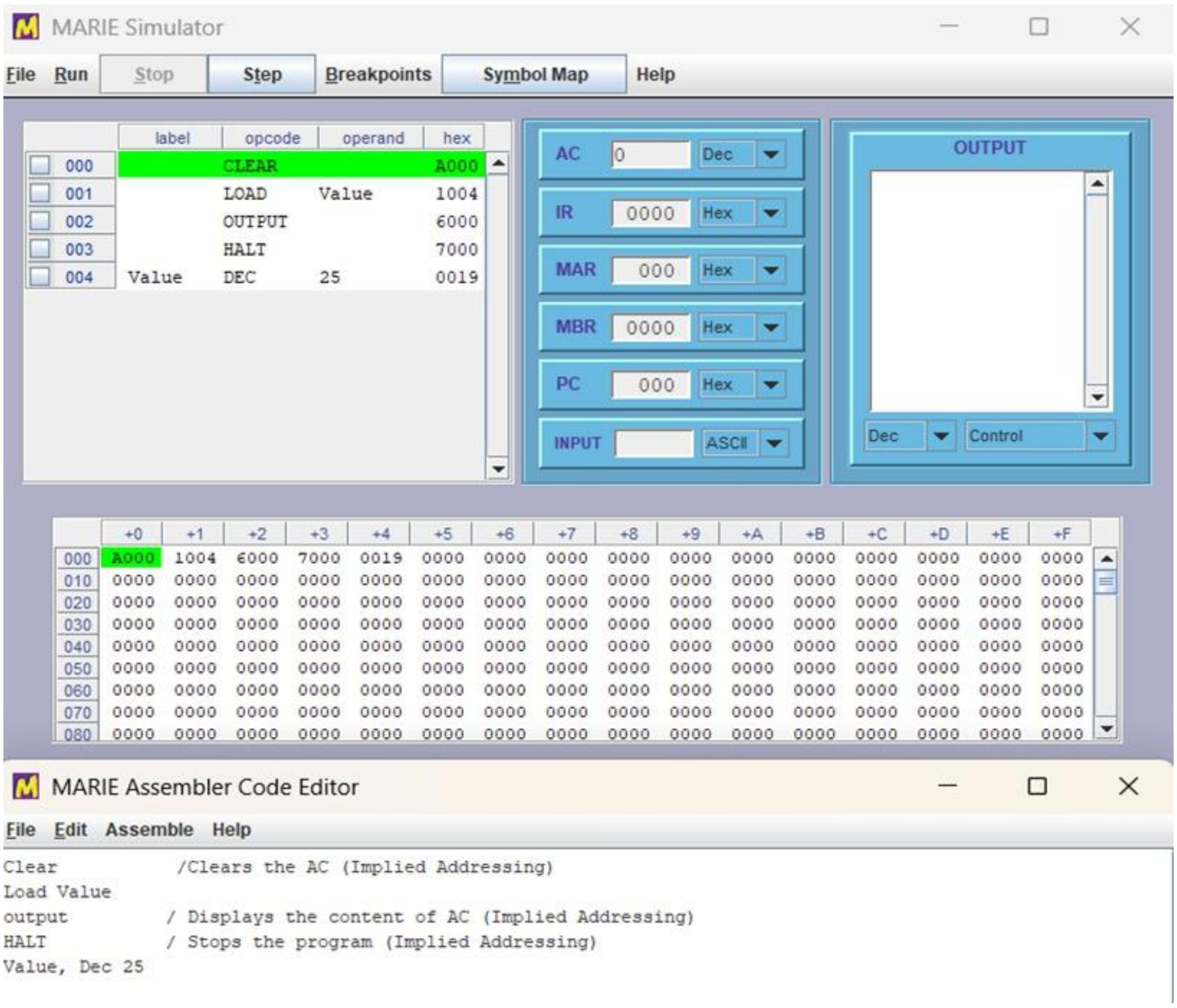
- ➢ Install Java virtual machine (JRE)
- ➢ Download MARIE Machine Simulator.

2. Configuration for task

- ➢ Memory Size: Default configuration (1000 memory cells).
- ➢ Instruction Set: Standard MARIE instructions.
- ➢ Assembler: Used to compile written programs into machine code.
- ➢ Debugger Mode: Enabled to step through the program and observe changes in registers and memory.
- ➢ Output Window: Configured to display results of the program.(In decimal).

7.2 Addressing Modes Simulation (Option 3)

7.2.1 Implied Addressing Mode


MARIE Simulator
File Run Stop Step Breakpoints Symbol Map Help
OUTPUT
Dec Control
MARIE Assembler Code Editor
File Edit Assemble Help

		label	opcode	operand	hex
☐	000		CLEAR		A000
☐	001		LOAD	Value	1004
☐	002		OUTPUT		6000
☐	003		HALT		7000
☐	004	Value	DEC	25	0019

AC	0	Dec
IR	0000	Hex
MAR	000	Hex
MBR	0000	Hex
PC	000	Hex
INPUT		ASCII

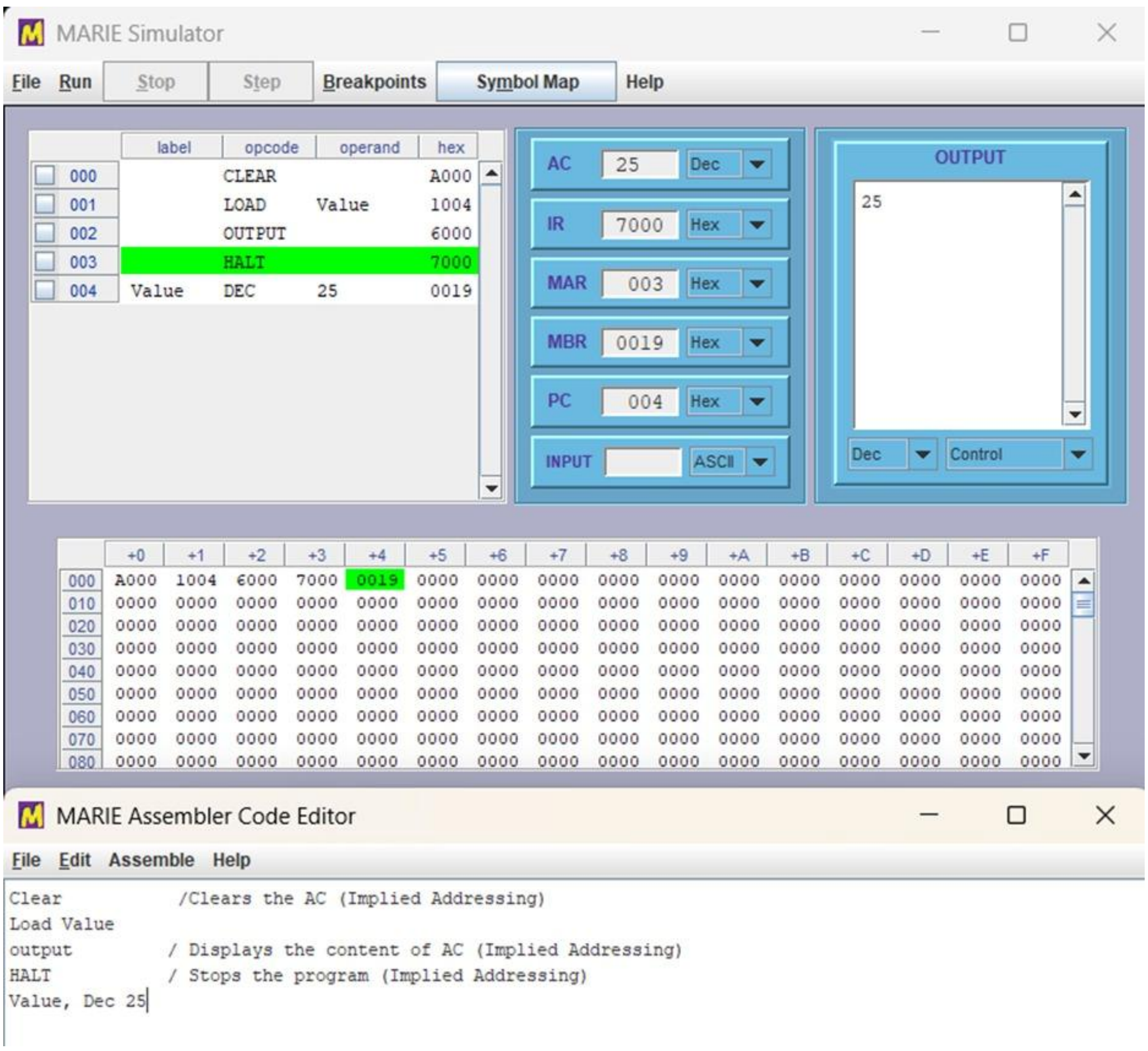
	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
000	A000	1004	6000	7000	0019	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
010	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
020	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
030	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
040	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

```
Clear           /Clears the AC (Implied Addressing)
Load Value
output         / Displays the content of AC (Implied Addressing)
HALT           / Stops the program (Implied Addressing)
Value, Dec 25
```

<u>Steps:</u>

- ➢ CLEAR does not specify any memory location; it directly affects the AC, setting it to 0.
- ➢ The AC value changed from 0 to 25 as shown in the simulator.
- ➢ OUTPUT uses Implied Addressing as it directly outputs the AC value.
- ➢ The HALT instruction successfully terminated the program also implied addressing.

Memory addressing results:


MARIE Simulator
File Run Stop Step Breakpoints Symbol Map Help
OUTPUT
Dec Control
MARIE Assembler Code Editor
File Edit Assemble Help

	label	opcode	operand	hex
000		CLEAR		A000
001		LOAD	Value	1004
002		OUTPUT		6000
003		HALT		7000
004	Value	DEC	25	0019

AC	25	Dec
IR	7000	Hex
MAR	003	Hex
MBR	0019	Hex
PC	004	Hex
INPUT		ASCII

```
25
```

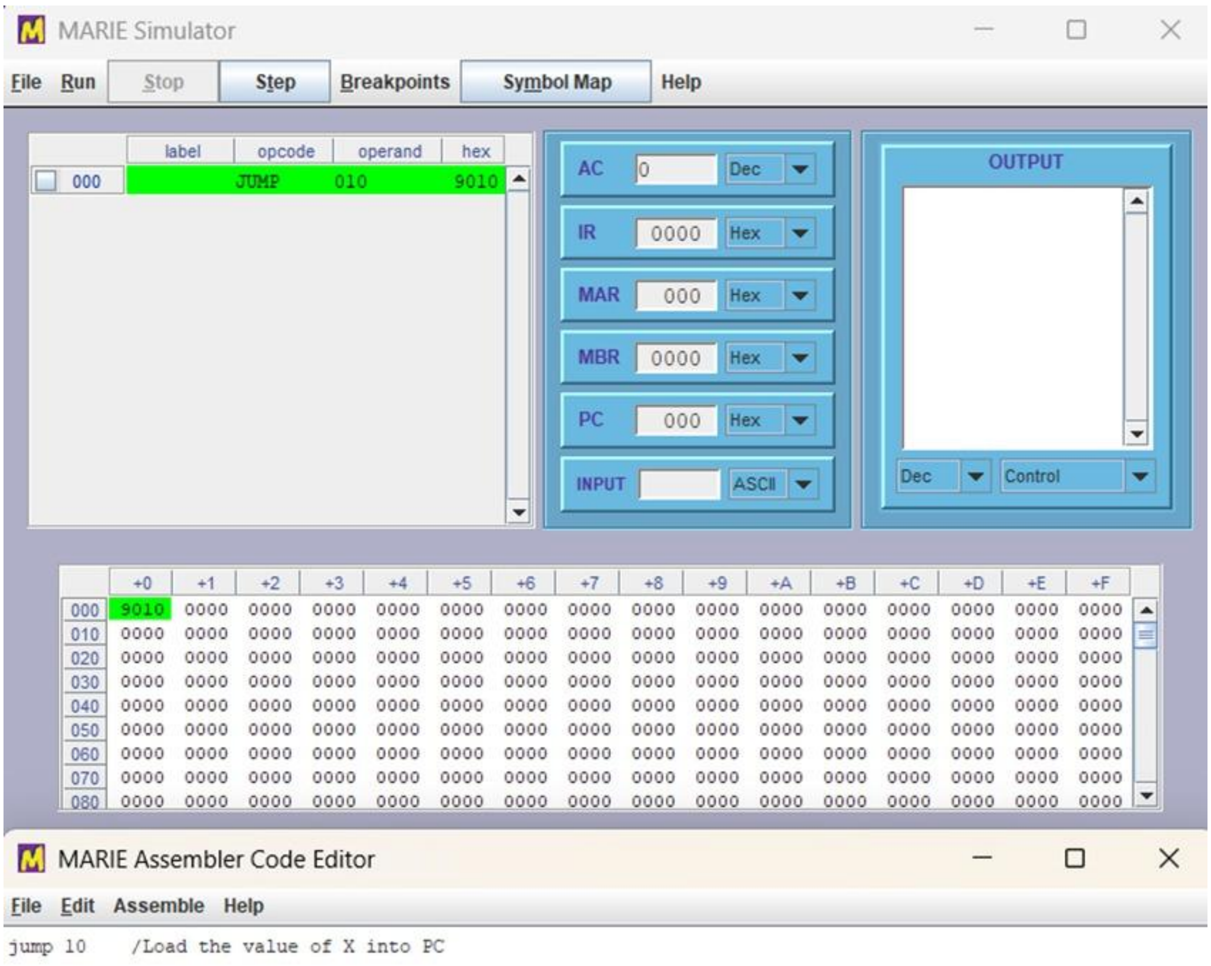
```
     +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  A000 1004 6000 7000 0019 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

```
Clear           /Clears the AC (Implied Addressing)
Load Value
output         / Displays the content of AC (Implied Addressing)
HALT           / Stops the program (Implied Addressing)
Value, Dec 25
```

While MARIE Simulator is a simplified educational tool, the principles demonstrated relate to modern processors like Intel Core i7. Core i7 Processors handle more complex instruction sets but retain the fundamental concepts of addressing modes:

- Implied Addressing Mode in MARIE directly manipulates registers (like the PC).
- In Core i7, similar operations are performed with dedicated registers (e.g., EAX, EBX, RAX in 64-bit mode).
- Instructions like INC RAX, DEC EAX, and RET are examples of Implied Addressing in Core i7.
- The Implied Addressing Mode is used for operations where the target is predefined or inherent to the instruction (like updating flags or manipulating dedicated registers).

7.2.2 Immediate Addressing Mode



	label	opcode	operand	hex
000		JUMP	010	9010

AC	0	Dec
IR	0000	Hex
MAR	000	Hex
MBR	0000	Hex
PC	000	Hex
INPUT		ASCII

Dec Control

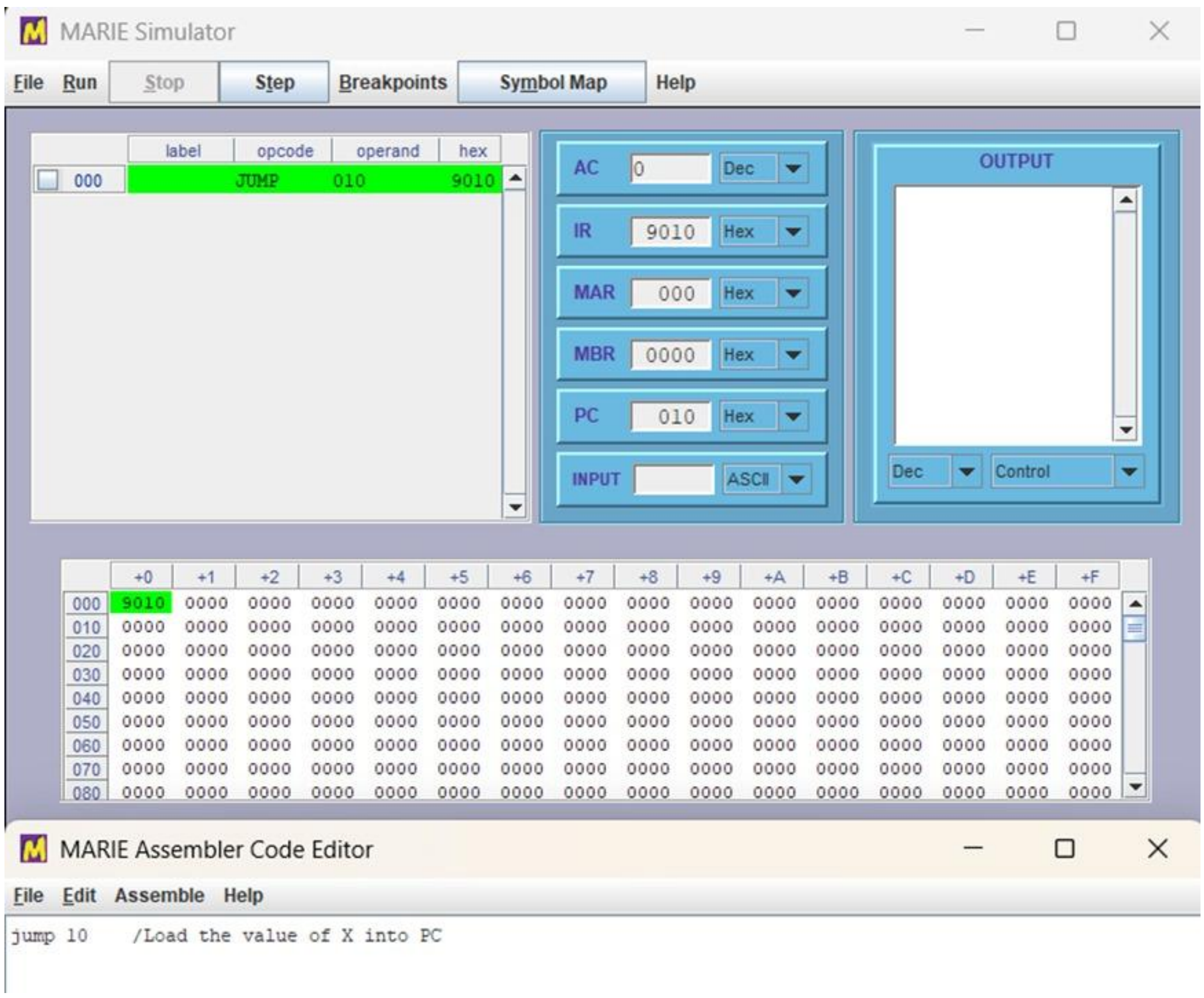
	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
000	9010	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
010	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
020	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
030	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
040	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

```
jump 10     /Load the value of X into PC
```

Steps:

- This instruction uses Immediate Addressing Mode where the operand (10) is directly provided as a constant value.
- It does not require memory lookup to fetch the value.
- The PC (Program Counter) is successfully updated to the value 10 as seen in the PC display section.

Memory addressing results:


MARIE Simulator
File Run Stop Step Breakpoints Symbol Map Help
OUTPUT
Dec Control
MARIE Assembler Code Editor
File Edit Assemble Help

	label	opcode	operand	hex
000		JUMP	010	9010

```
AC     0      Dec
IR     9010   Hex
MAR    000    Hex
MBR    0000   Hex
PC     010    Hex
INPUT         ASCII
```

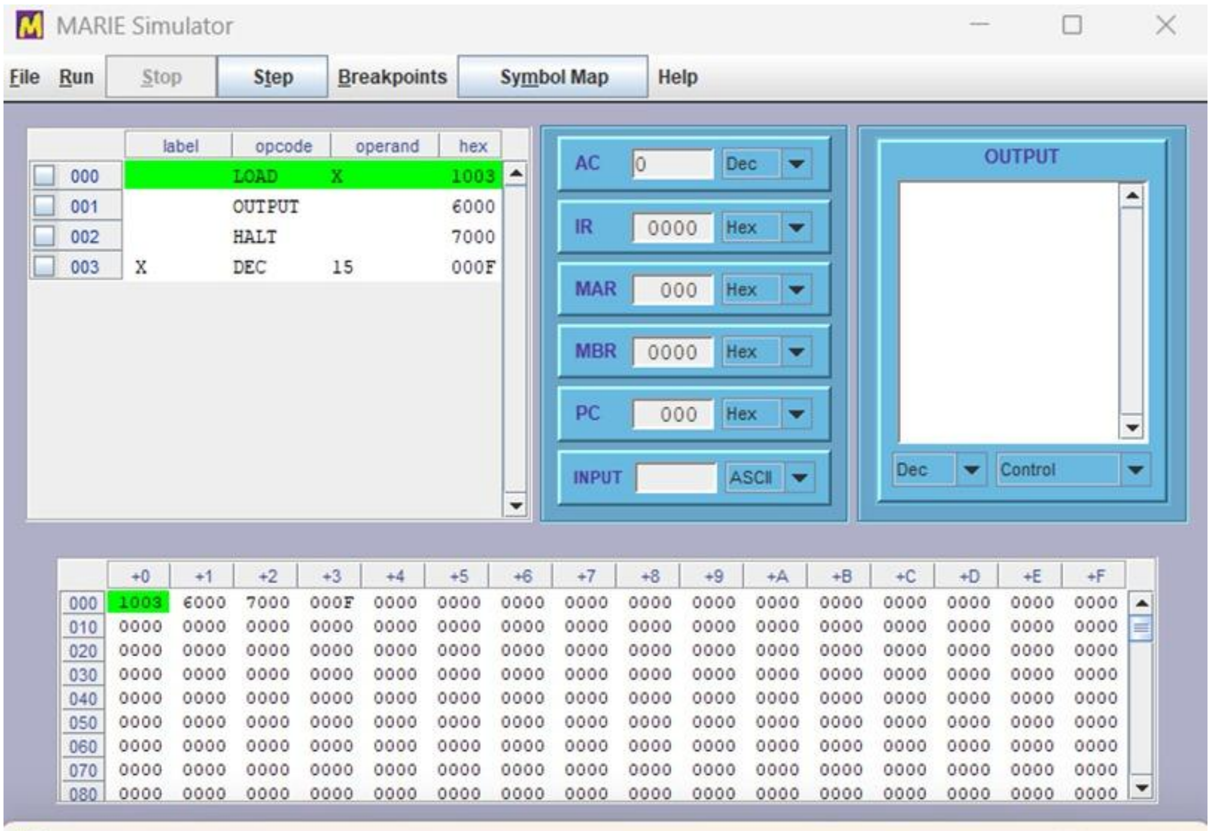
```
     +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  9010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

```
jump 10    /Load the value of X into PC
```

In Core i7, Immediate Addressing is supported through Immediate Operands within CISC (Complex Instruction Set Computing).

- Unlike MARIE, core i7 supports different sizes of immediate data (e.g., 8-bit, 16-bit, 32-bit, 64-bit) for greater flexibility.
- Faster Access: No memory access required, which reduces memory latency.
- Common Usage: Useful for setting initial values, loop counters, or constant-based arithmetic.

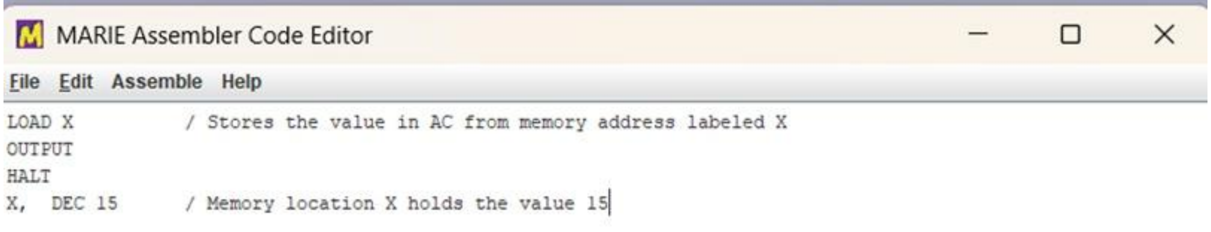
7.2.3 Direct Addressing Mode



	label	opcode	operand	hex
000		LOAD	X	1003
001		OUTPUT		6000
002		HALT		7000
003	X	DEC	15	000F

Register	Value	Mode
AC	0	Dec
IR	0000	Hex
MAR	000	Hex
MBR	0000	Hex
PC	000	Hex
INPUT		ASCII

```
     +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  1003 6000 7000 000F 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

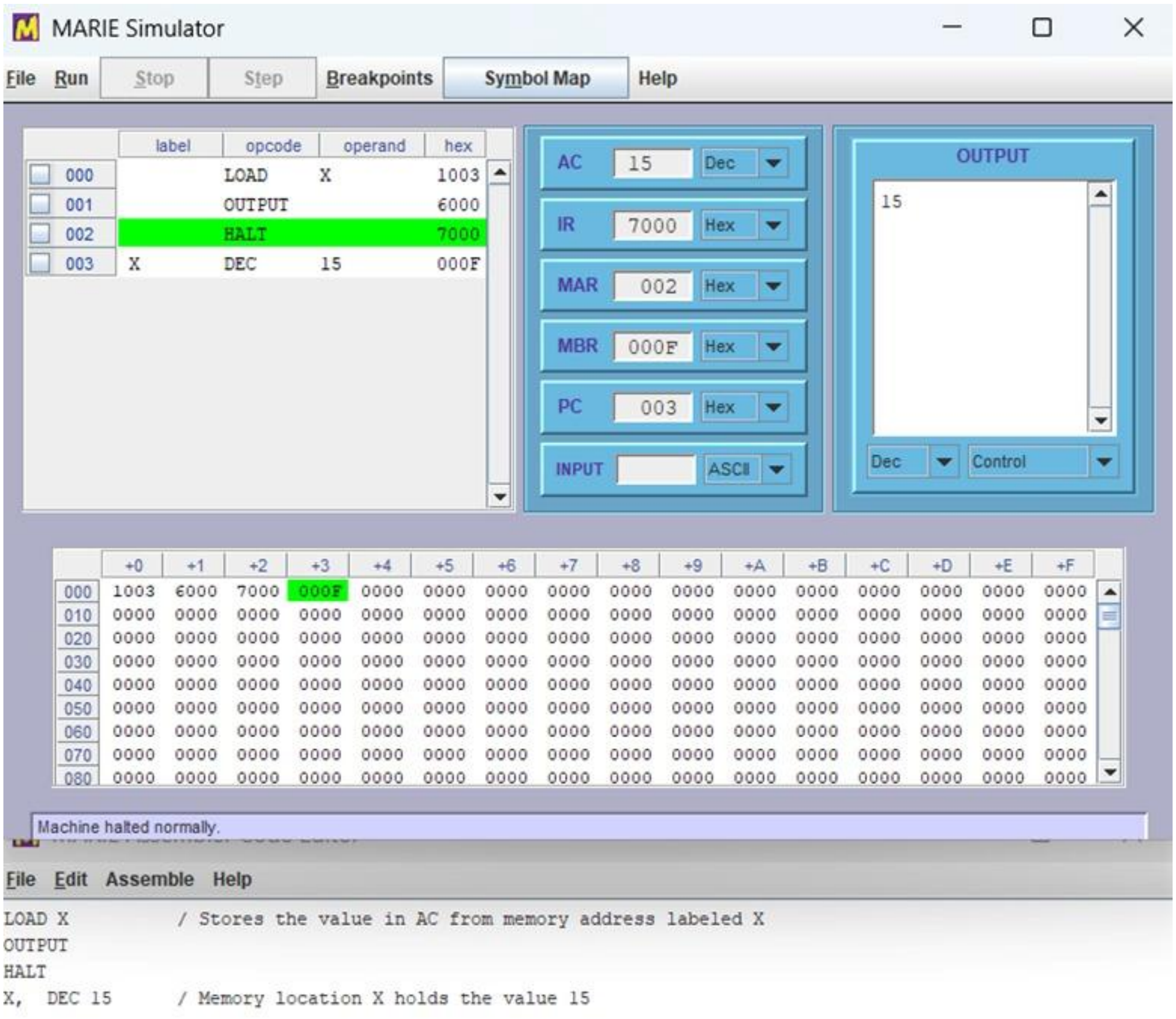


```
LOAD X          / Stores the value in AC from memory address labeled X
OUTPUT
HALT
X,   DEC 15     / Memory location X holds the value 15
```

Steps:

- ➢ The computer loaded the number 15 directly from memory into a place called the accumulator (AC).
- ➢ Then the program is stopped (HALT).

Memory addressing results:



	label	opcode	operand	hex
000		LOAD	X	1003
001		OUTPUT		6000
002		HALT		7000
003	X	DEC	15	000F

AC	15	Dec
IR	7000	Hex
MAR	002	Hex
MBR	000F	Hex
PC	003	Hex
INPUT		ASCII

OUTPUT

```
15
```

Dec   Control

```
     +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  1003 6000 7000 000F 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Machine halted normally.

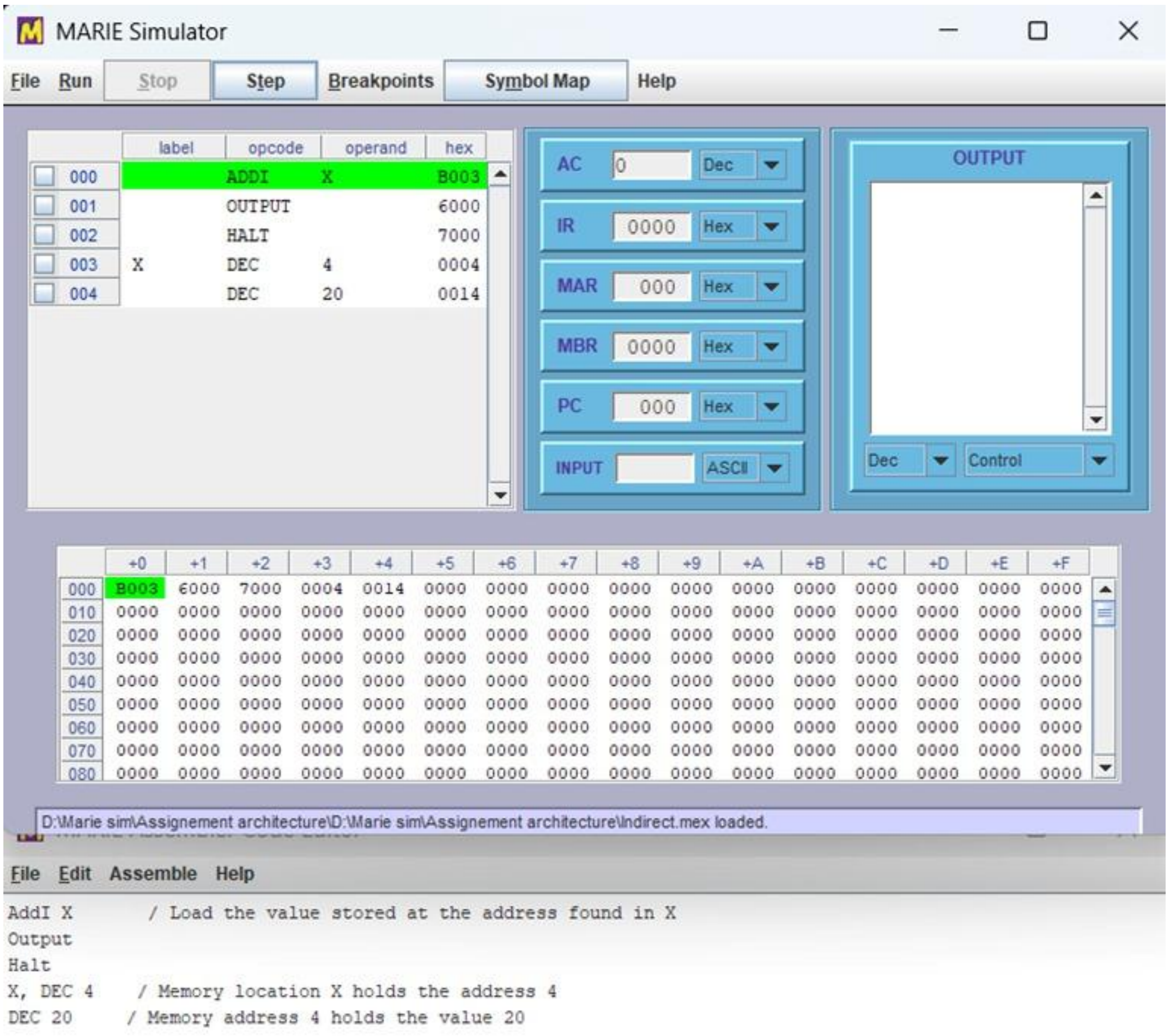
```
LOAD X          / Stores the value in AC from memory address labeled X
OUTPUT
HALT
X,  DEC 15      / Memory location X holds the value 15
```

MARIE is a very simple version of how real computers like an Intel i7 work. It helps you learn the basic steps every processor uses: Fetch ,Decode ,Execute.

An Intel i7 does the same things, but it's much faster and more powerful:

- ➢ It can run many instructions at once.
- ➢ It has multiple cores and more memory
- ➢ It uses advanced technology like cache and pipelining But at its core, it still follows the same idea as MARIE. So, learning MARIE helps you understand how more advanced CPUs like the i7 work behind the scenes.

7.2.4 Indirect Addressing Mode



	label	opcode	operand	hex
000		ADDI	X	B003
001		OUTPUT		6000
002		HALT		7000
003	X	DEC	4	0004
004		DEC	20	0014

Register	Value	Mode
AC	0	Dec
IR	0000	Hex
MAR	000	Hex
MBR	0000	Hex
PC	000	Hex
INPUT		ASCII

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
000	B003	6000	7000	0004	0014	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
010	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
020	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
030	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
040	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

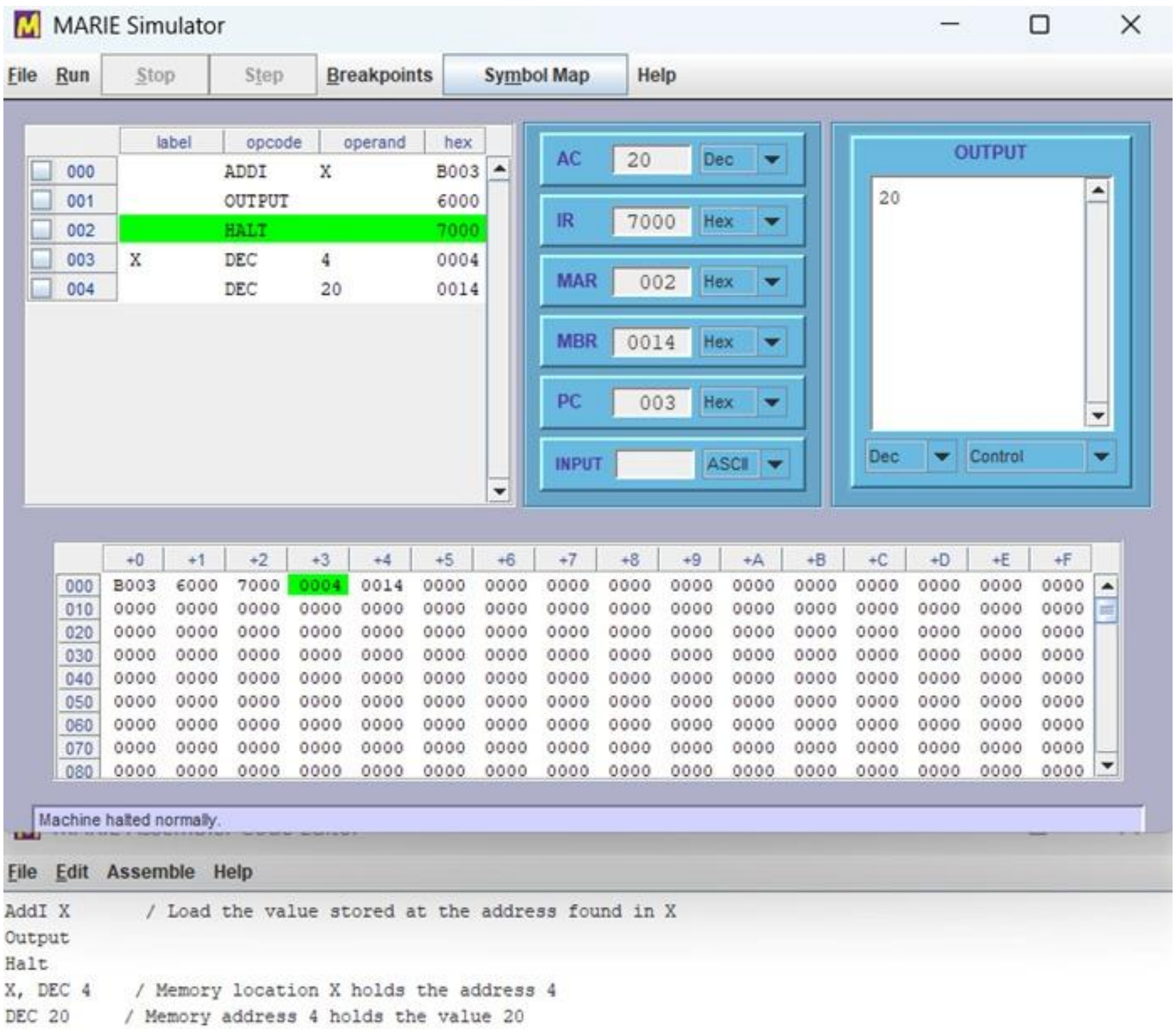
```
D:\Marie sim\Assignement architecture\D:\Marie sim\Assignement architecture\Indirect.mex loaded.
```

```
AddI X       / Load the value stored at the address found in X
Output
Halt
X, DEC 4     / Memory location X holds the address 4
DEC 20      / Memory address 4 holds the value 20
```

<u>Steps:</u>

- The program first looked at a memory location called X, which had the number 4.
- Then it went to address 4, where it found the number 20.
- That number (20) was loaded into the accumulator.

Memory addressing results:



	label	opcode	operand	hex
000		ADDI	X	B003
001		OUTPUT		6000
002		HALT		7000
003	X	DEC	4	0004
004		DEC	20	0014

AC	20	Dec
IR	7000	Hex
MAR	002	Hex
MBR	0014	Hex
PC	003	Hex
INPUT		ASCII

OUTPUT

```
20
```

Dec Control

```
     +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  B003 6000 7000 0004 0014 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

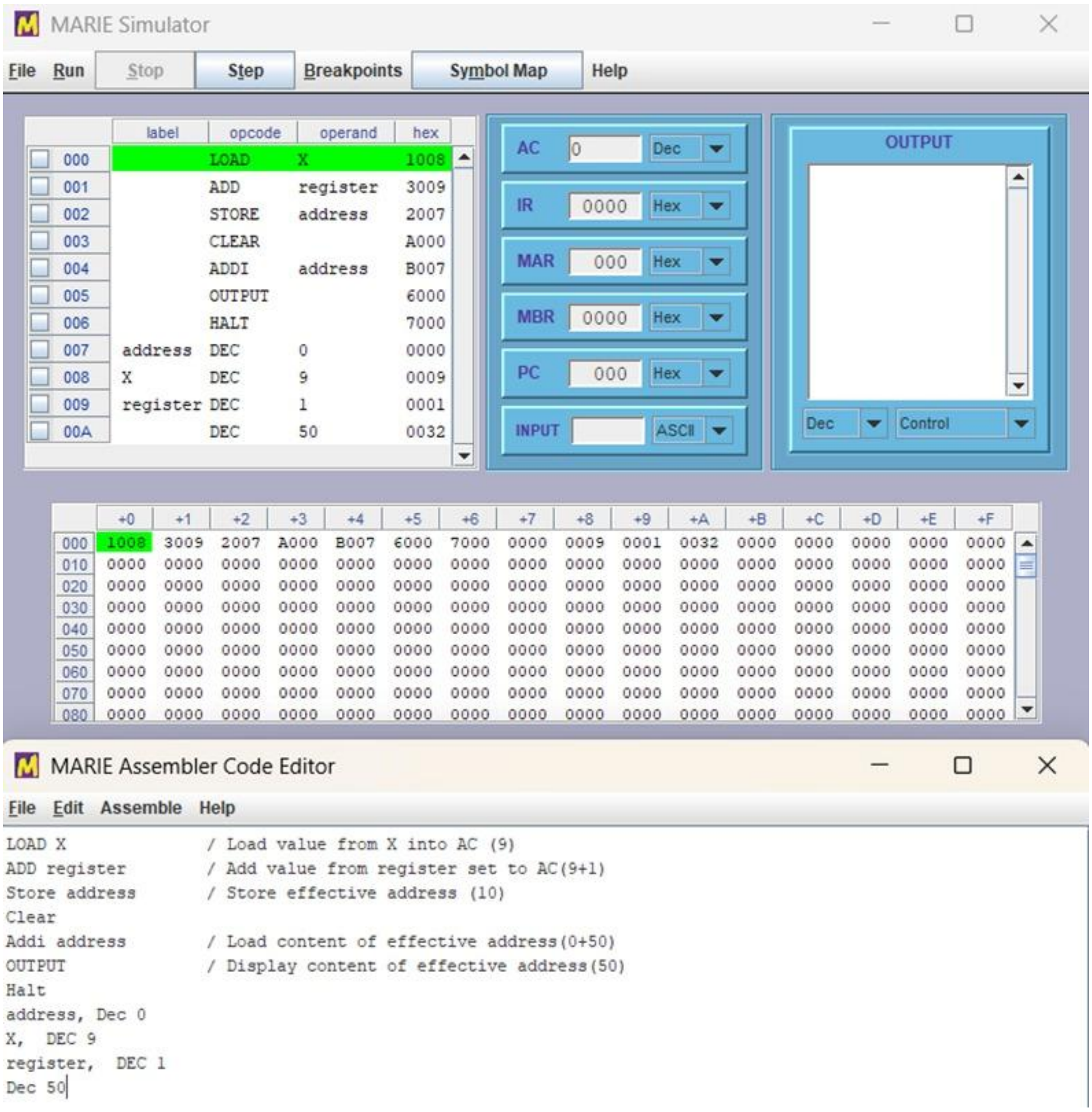
Machine halted normally.

```
AddI X       / Load the value stored at the address found in X
Output
Halt
X, DEC 4    / Memory location X holds the address 4
DEC 20     / Memory address 4 holds the value 20
```

Modern processors like the Intel i7 also use indirect addressing, pointers, and multi-step memory access but at a much faster and more complex level.

MARIE helps you understand how computers find and use data in memory, which is key to understanding real computer operations.

7.2.5 Register Direct Addressing Mode



	label	opcode	operand	hex
000		LOAD	X	1008
001		ADD	register	3009
002		STORE	address	2007
003		CLEAR		A000
004		ADDI	address	B007
005		OUTPUT		6000
006		HALT		7000
007	address	DEC	0	0000
008	X	DEC	9	0009
009	register	DEC	1	0001
00A		DEC	50	0032

AC	0	Dec
IR	0000	Hex
MAR	000	Hex
MBR	0000	Hex
PC	000	Hex
INPUT		ASCII

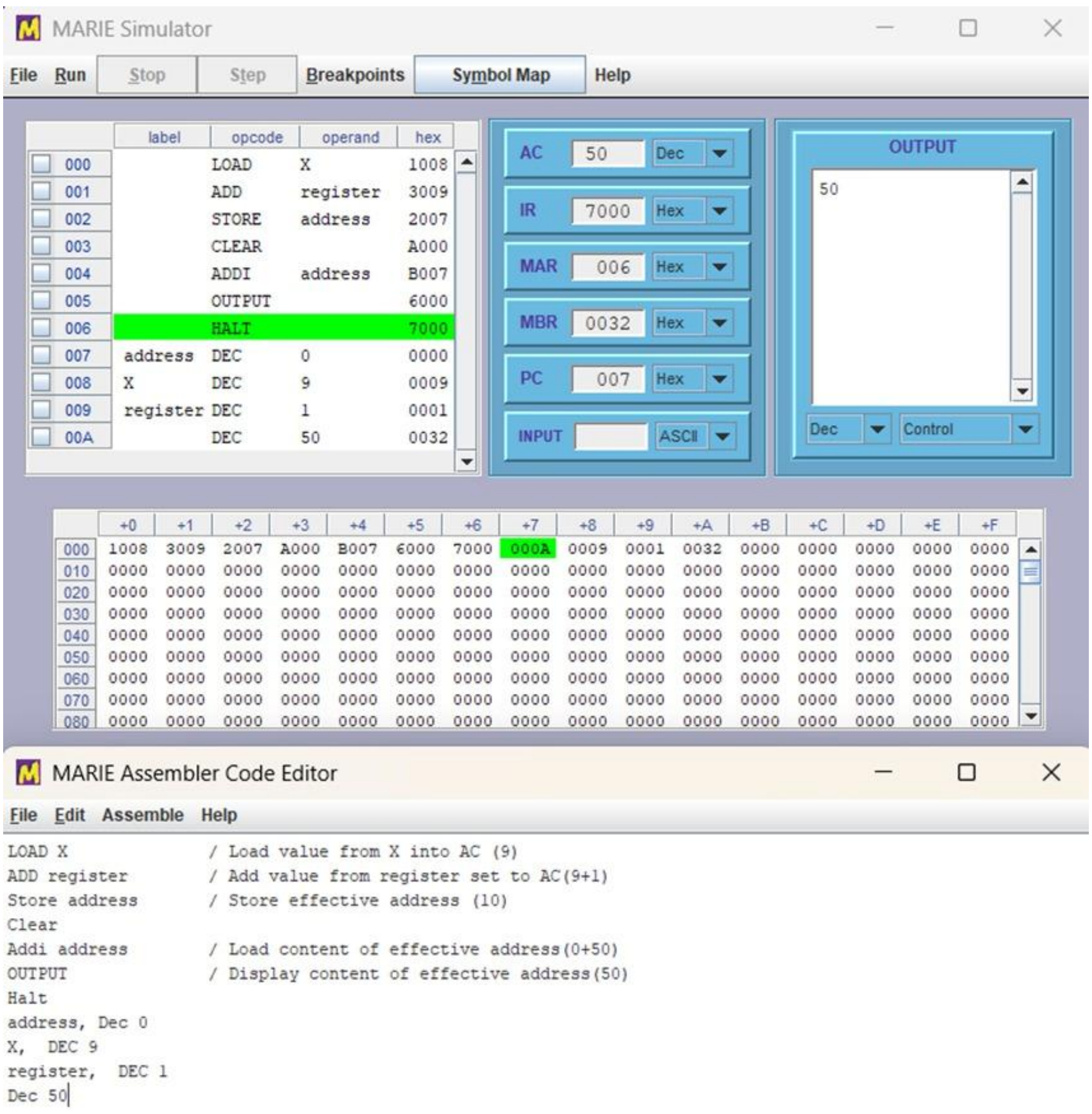
	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
000	1008	3009	2007	A000	B007	6000	7000	0000	0009	0001	0032	0000	0000	0000	0000	0000
010	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
020	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
030	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
040	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

```
LOAD X              / Load value from X into AC (9)
ADD register        / Add value from register set to AC(9+1)
Store address       / Store effective address (10)
Clear
Addi address        / Load content of effective address(0+50)
OUTPUT              / Display content of effective address(50)
Halt
address, Dec 0
X,  DEC 9
register,  DEC 1
Dec 50
```

Steps:

- Load the value from memory location labelled X into the AC.
- Add the value stored in the register set.
- Store the result into memory location labelled address.
- Reset AC to 0. Load the value from memory location address into the AC.

<u>Memory addressing results:</u>



	label	opcode	operand	hex
000		LOAD	X	1008
001		ADD	register	3009
002		STORE	address	2007
003		CLEAR		A000
004		ADDI	address	B007
005		OUTPUT		6000
006		HALT		7000
007	address	DEC	0	0000
008	X	DEC	9	0009
009	register	DEC	1	0001
00A		DEC	50	0032

AC	50	Dec
IR	7000	Hex
MAR	006	Hex
MBR	0032	Hex
PC	007	Hex
INPUT		ASCII

OUTPUT

```
50
```

Dec Control

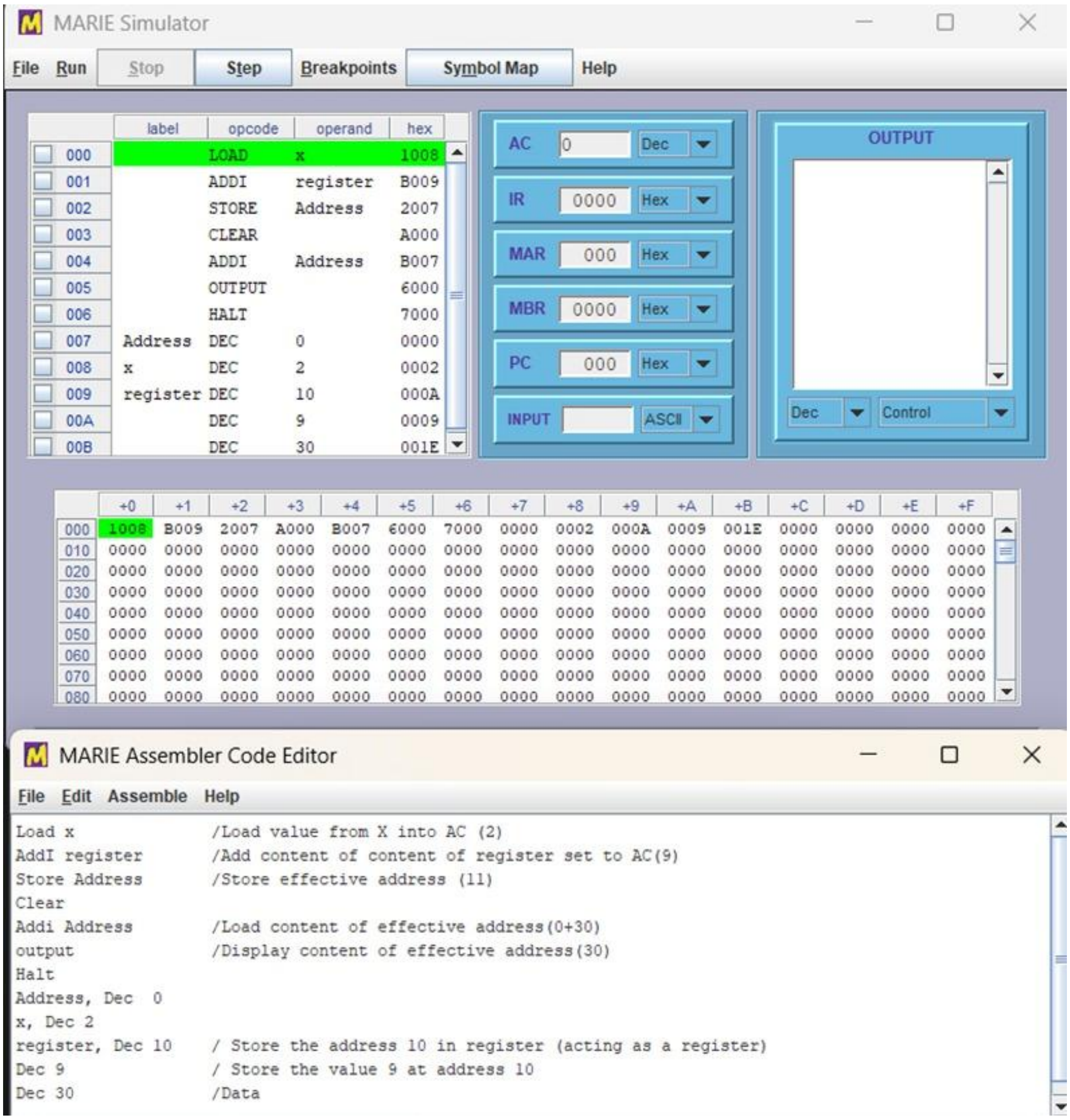
	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
000	1008	3009	2007	A000	B007	6000	7000	000A	0009	0001	0032	0000	0000	0000	0000	0000
010	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
020	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
030	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
040	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
050	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
060	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
070	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
080	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

```
LOAD X              / Load value from X into AC (9)
ADD register        / Add value from register set to AC(9+1)
Store address       / Store effective address (10)
Clear
Addi address        / Load content of effective address(0+50)
OUTPUT              / Display content of effective address(50)
Halt
address, Dec 0
X,  DEC 9
register,  DEC 1
Dec 50
```

Even though the i7 is way more powerful and faster, it uses the same idea:

- It also adds an index to a base address to find the effective address.
- This is helpful when working with lists or arrays.
- So just like MARIE adds 1 to 100 to get 101, the i7 does similar math, but much quicker and with more advanced instructions.

7.2.6 Register Indirect Addressing Mode



	label	opcode	operand	hex
000		LOAD	x	1008
001		ADDI	register	B009
002		STORE	Address	2007
003		CLEAR		A000
004		ADDI	Address	B007
005		OUTPUT		6000
006		HALT		7000
007	Address	DEC	0	0000
008	x	DEC	2	0002
009	register	DEC	10	000A
00A		DEC	9	0009
00B		DEC	30	001E

AC	0	Dec
IR	0000	Hex
MAR	000	Hex
MBR	0000	Hex
PC	000	Hex
INPUT		ASCII

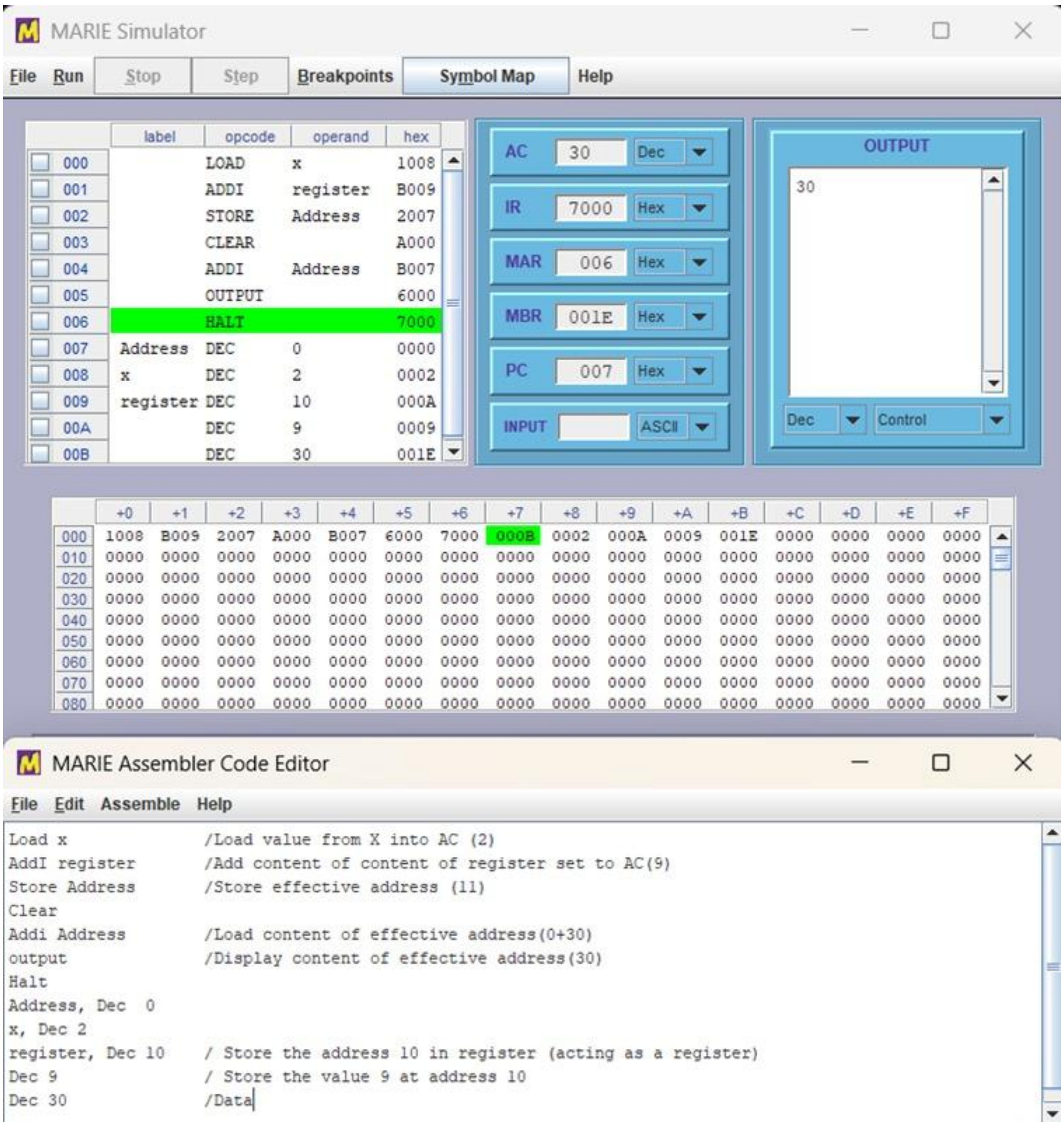
```
     +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  1008 B009 2007 A000 B007 6000 7000 0000 0002 000A 0009 001E 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

```
Load x              /Load value from X into AC (2)
AddI register       /Add content of content of register set to AC(9)
Store Address       /Store effective address (11)
Clear
Addi Address        /Load content of effective address(0+30)
output              /Display content of effective address(30)
Halt
Address, Dec  0
x, Dec 2
register, Dec 10    / Store the address 10 in register (acting as a register)
Dec 9               / Store the value 9 at address 10
Dec 30              /Data
```

<u>Steps:</u>

- ➢ Load the value from memory location labelled X into the AC.
- ➢ It indirectly adds the register set to AC to find effective address.

Memory addressing results:


MARIE Simulator
File Run Stop Step Breakpoints Symbol Map Help
OUTPUT
Dec Control
MARIE Assembler Code Editor
File Edit Assemble Help

	label	opcode	operand	hex
000		LOAD	x	1008
001		ADDI	register	B009
002		STORE	Address	2007
003		CLEAR		A000
004		ADDI	Address	B007
005		OUTPUT		6000
006		HALT		7000
007	Address	DEC	0	0000
008	x	DEC	2	0002
009	register	DEC	10	000A
00A		DEC	9	0009
00B		DEC	30	001E

```
AC     30     Dec
IR     7000   Hex
MAR    006    Hex
MBR    001E   Hex
PC     007    Hex
INPUT         ASCII
```

```
OUTPUT
30
```

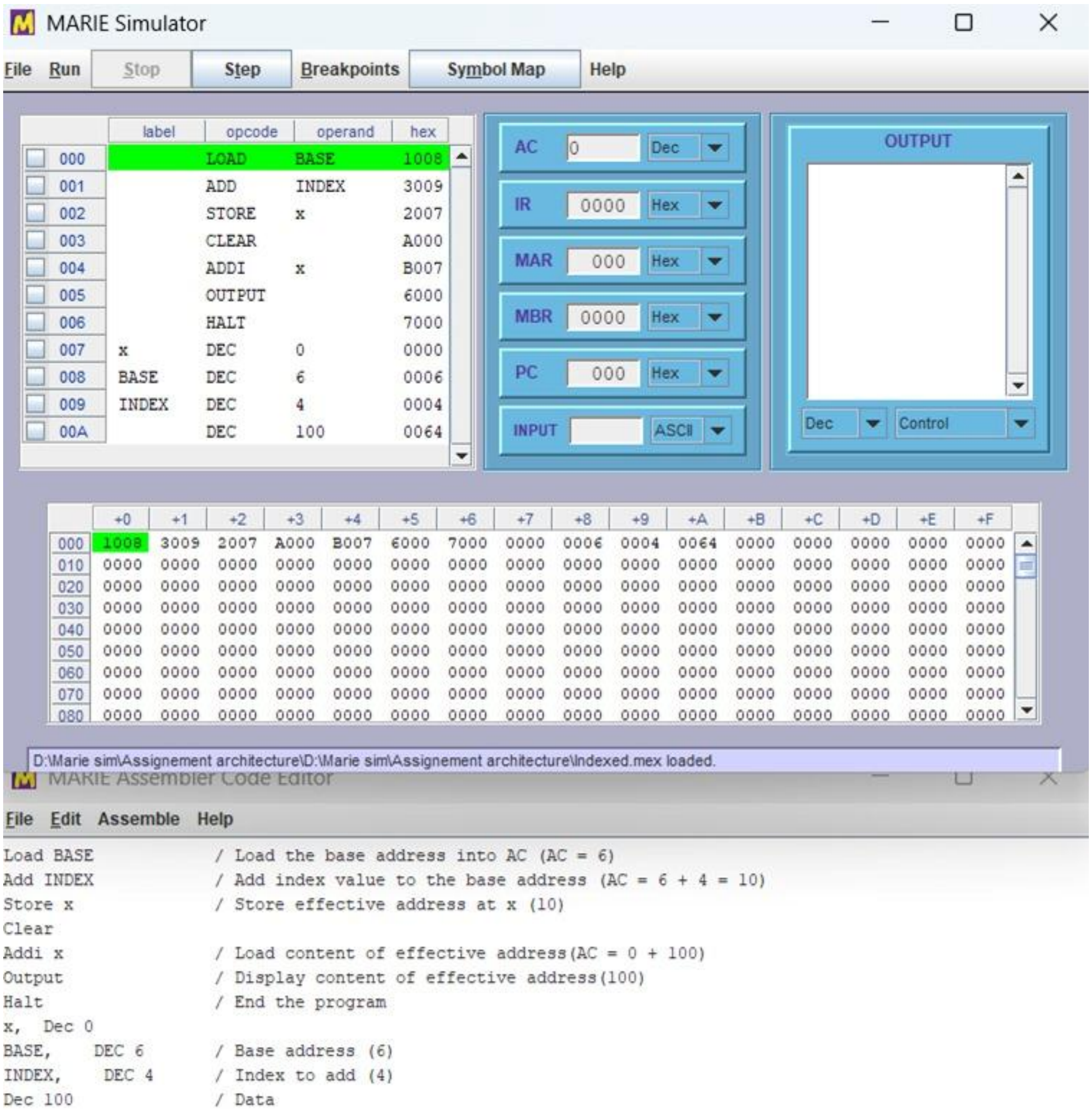
```
     +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  1008 B009 2007 A000 B007 6000 7000 000B 0002 000A 0009 001E 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

```
Load x              /Load value from X into AC (2)
AddI register       /Add content of content of register set to AC(9)
Store Address       /Store effective address (11)
Clear
Addi Address        /Load content of effective address(0+30)
output              /Display content of effective address(30)
Halt
Address, Dec  0
x, Dec 2
register, Dec 10    / Store the address 10 in register (acting as a register)
Dec 9               / Store the value 9 at address 10
Dec 30              /Data
```

Even though the Intel i7 is much faster and more powerful, it also works like this:

- It uses numbers in memory and adds them together to find where data is stored.
- This helps it quickly find information in big lists or complex data structures.

7.2.7 Index Addressing Mode



	label	opcode	operand	hex
000		LOAD	BASE	1008
001		ADD	INDEX	3009
002		STORE	x	2007
003		CLEAR		A000
004		ADDI	x	B007
005		OUTPUT		6000
006		HALT		7000
007	x	DEC	0	0000
008	BASE	DEC	6	0006
009	INDEX	DEC	4	0004
00A		DEC	100	0064

AC 0 Dec
IR 0000 Hex
MAR 000 Hex
MBR 0000 Hex
PC 000 Hex

```
     +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  1008 3009 2007 A000 B007 6000 7000 0000 0006 0004 0064 0000 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

D:\Marie sim\Assignement architecture\D:\Marie sim\Assignement architecture\Indexed.mex loaded.

MARIE Assembler Code Editor

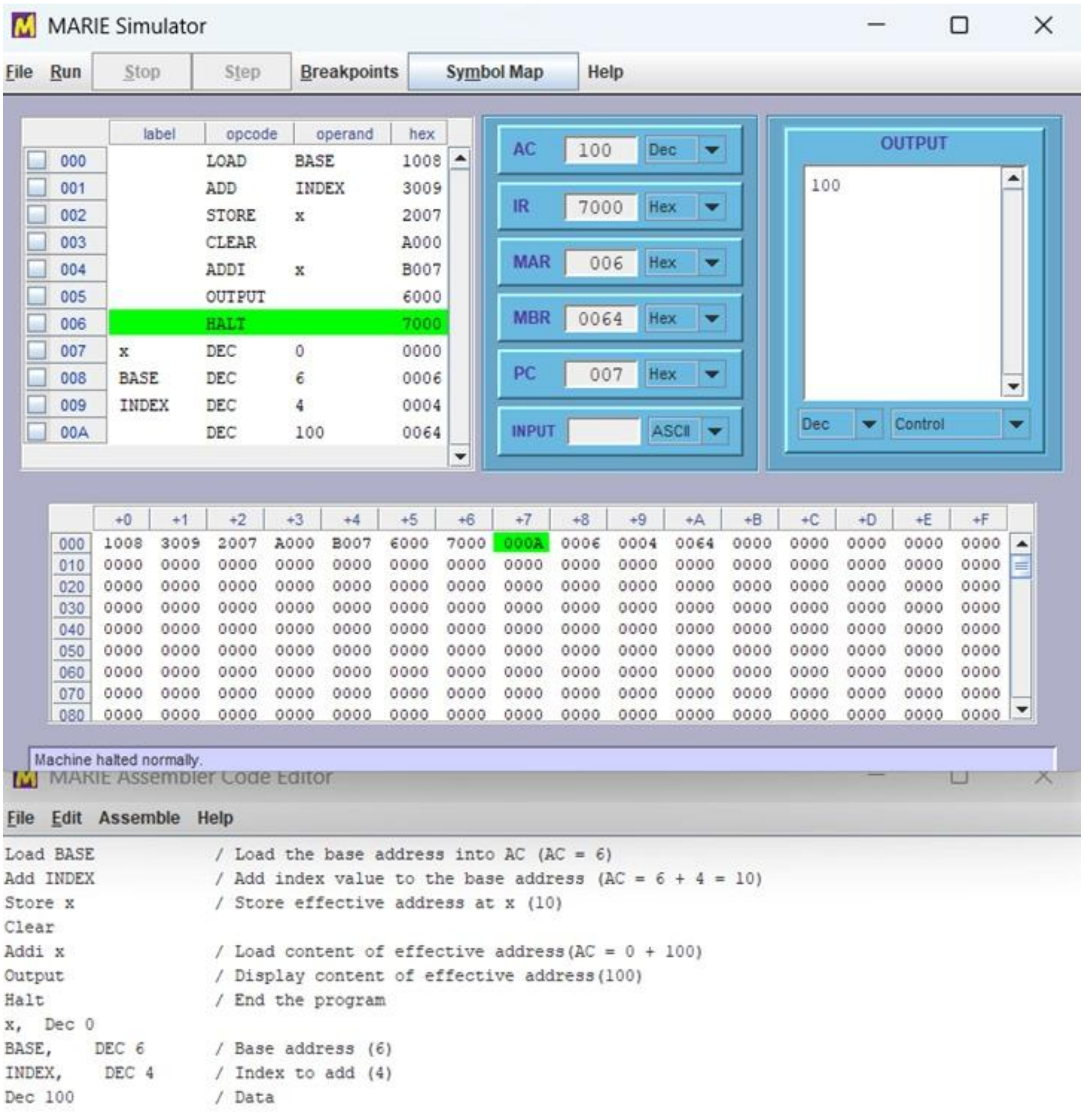
File Edit Assemble Help

```
Load BASE             / Load the base address into AC (AC = 6)
Add INDEX             / Add index value to the base address (AC = 6 + 4 = 10)
Store x               / Store effective address at x (10)
Clear
Addi x                / Load content of effective address(AC = 0 + 100)
Output                / Display content of effective address(100)
Halt                  / End the program
x,  Dec 0
BASE,     DEC 6       / Base address (6)
INDEX,     DEC 4      / Index to add (4)
Dec 100               / Data
```

<u>Steps:</u>

- ➢ It loads the base address.
- ➢ Then it adds an index to AC, to get the effective address.

Memory addressing results:



	label	opcode	operand	hex
000		LOAD	BASE	1008
001		ADD	INDEX	3009
002		STORE	x	2007
003		CLEAR		A000
004		ADDI	x	B007
005		OUTPUT		6000
006		HALT		7000
007	x	DEC	0	0000
008	BASE	DEC	6	0006
009	INDEX	DEC	4	0004
00A		DEC	100	0064

AC	100	Dec
IR	7000	Hex
MAR	006	Hex
MBR	0064	Hex
PC	007	Hex
INPUT		ASCII

OUTPUT: 100 (Dec, Control)

```
     +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  1008 3009 2007 A000 B007 6000 7000 000A 0006 0004 0064 0000 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

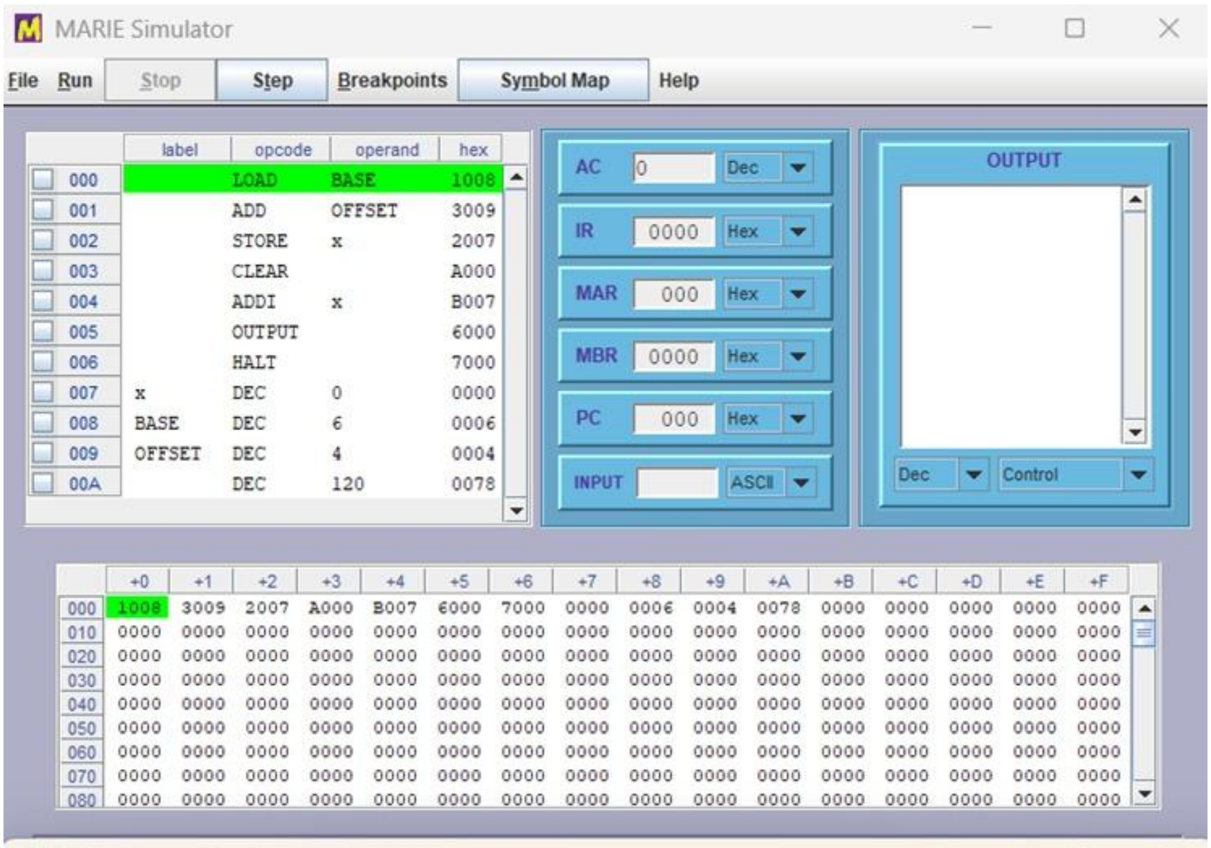
Machine halted normally.

```
Load BASE              / Load the base address into AC (AC = 6)
Add INDEX              / Add index value to the base address (AC = 6 + 4 = 10)
Store x                / Store effective address at x (10)
Clear
Addi x                 / Load content of effective address(AC = 0 + 100)
Output                 / Display content of effective address(100)
Halt                   / End the program
x,  Dec 0
BASE,    DEC 6         / Base address (6)
INDEX,    DEC 4        / Index to add (4)
Dec 100                / Data
```

The Intel i7 processor does similar things but much faster and smarter.

- It also uses base + index to find address in memory.
- This is super helpful when the computer is working with lists or arrays.
- The i7 just does it with more power and speed for real-world programs.

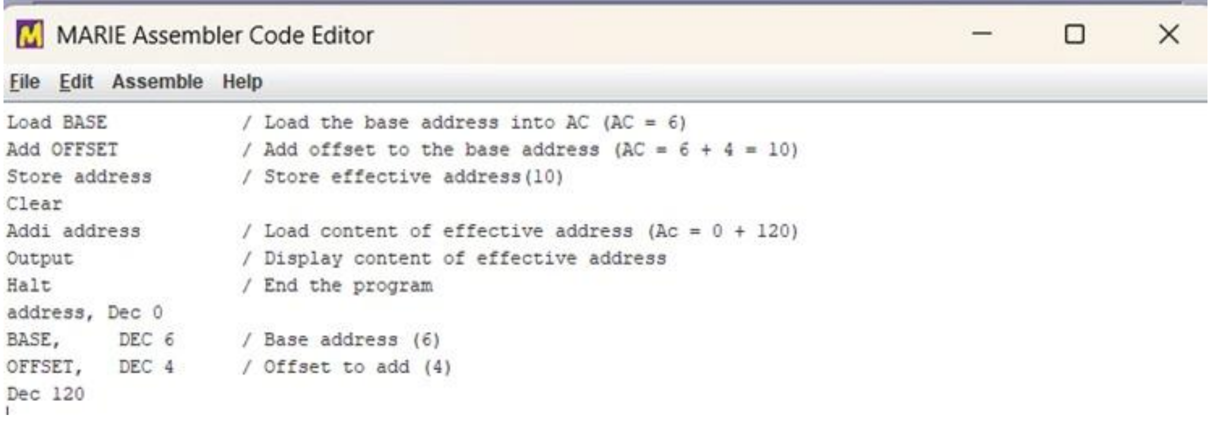
7.2.8 Base Addressing Mode



	label	opcode	operand	hex
000		LOAD	BASE	1008
001		ADD	OFFSET	3009
002		STORE	x	2007
003		CLEAR		A000
004		ADDI	x	B007
005		OUTPUT		6000
006		HALT		7000
007	x	DEC	0	0000
008	BASE	DEC	6	0006
009	OFFSET	DEC	4	0004
00A		DEC	120	0078

AC	0	Dec
IR	0000	Hex
MAR	000	Hex
MBR	0000	Hex
PC	000	Hex
INPUT		ASCII

```
      +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  1008 3009 2007 A000 B007 6000 7000 0000 0006 0004 0078 0000 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

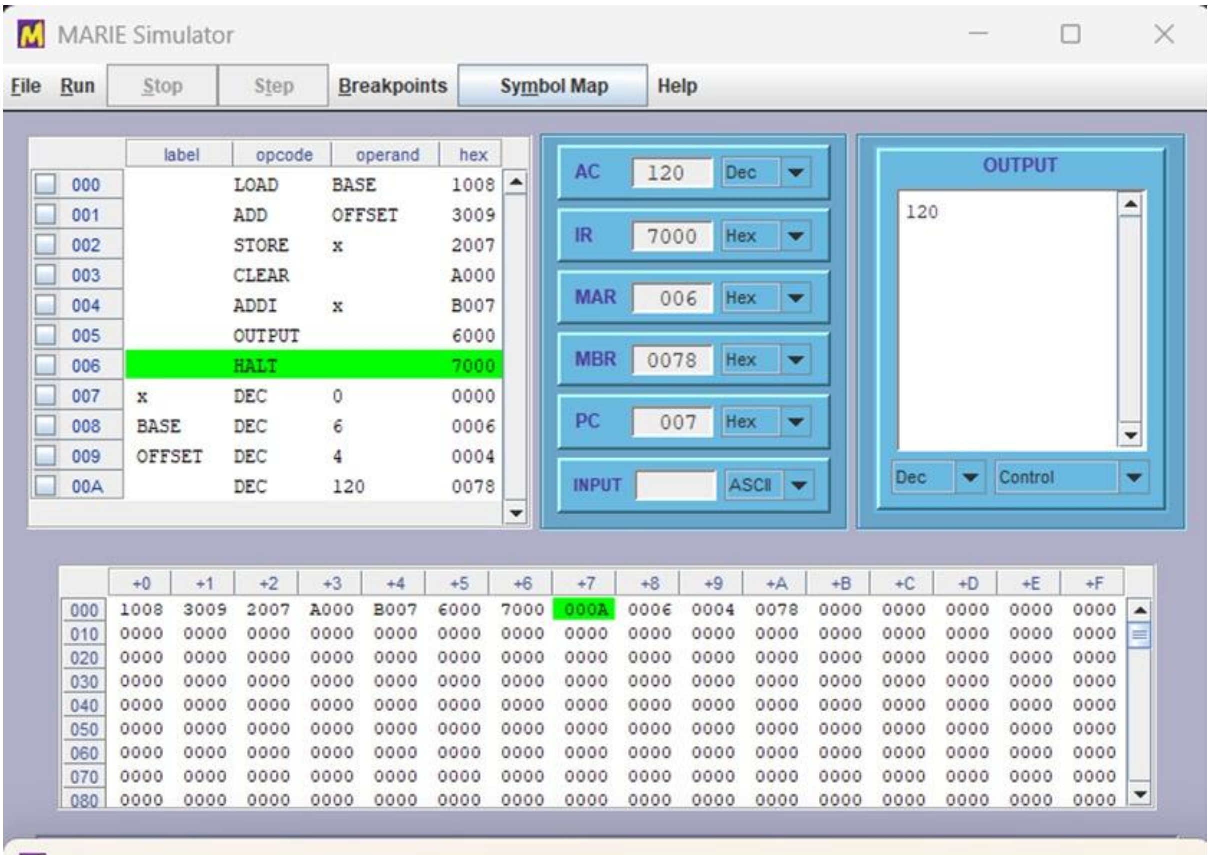


```
Load BASE              / Load the base address into AC (AC = 6)
Add OFFSET             / Add offset to the base address (AC = 6 + 4 = 10)
Store address          / Store effective address(10)
Clear
Addi address           / Load content of effective address (Ac = 0 + 120)
Output                 / Display content of effective address
Halt                   / End the program
address, Dec 0
BASE,     DEC 6        / Base address (6)
OFFSET,   DEC 4        / Offset to add (4)
Dec 120
```

<u>Steps:</u>

- ➢ It first takes a base number.
- ➢ Then it adds an offset.
- ➢ It becomes the final address we want. At that address (10), there is a value: 120.
- ➢ The program then shows 120 on the screen.

Memory addressing results:



	label	opcode	operand	hex
000		LOAD	BASE	1008
001		ADD	OFFSET	3009
002		STORE	x	2007
003		CLEAR		A000
004		ADDI	x	B007
005		OUTPUT		6000
006		HALT		7000
007	x	DEC	0	0000
008	BASE	DEC	6	0006
009	OFFSET	DEC	4	0004
00A		DEC	120	0078

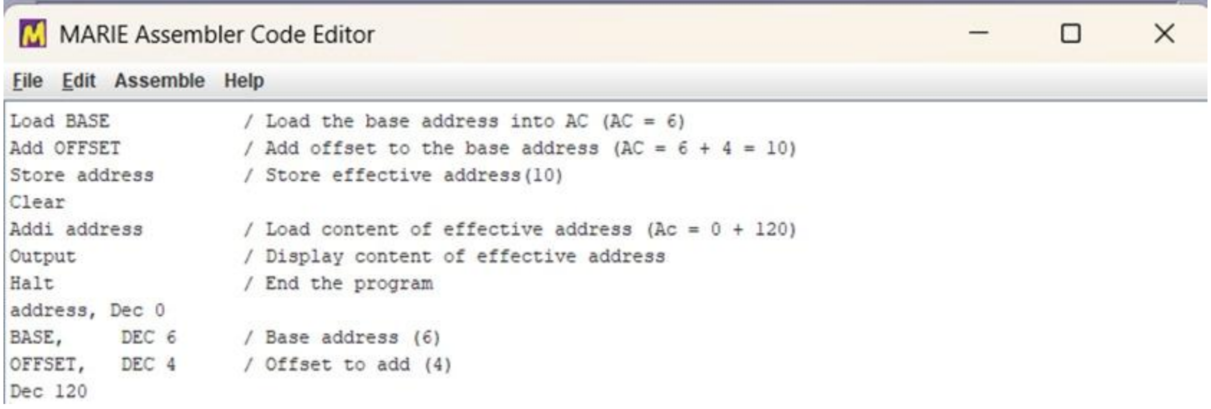
AC	120	Dec
IR	7000	Hex
MAR	006	Hex
MBR	0078	Hex
PC	007	Hex
INPUT		ASCII

OUTPUT

```
120
```

Dec Control

```
     +0   +1   +2   +3   +4   +5   +6   +7   +8   +9   +A   +B   +C   +D   +E   +F
000  1008 3009 2007 A000 B007 6000 7000 000A 0006 0004 0078 0000 0000 0000 0000 0000
010  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
020  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
030  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
040  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
050  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
060  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
070  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
080  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```



```
Load BASE               / Load the base address into AC (AC = 6)
Add OFFSET              / Add offset to the base address (AC = 6 + 4 = 10)
Store address           / Store effective address(10)
Clear
Addi address            / Load content of effective address (Ac = 0 + 120)
Output                  / Display content of effective address
Halt                    / End the program
address, Dec 0
BASE,     DEC 6         / Base address (6)
OFFSET,   DEC 4         / Offset to add (4)
Dec 120
```

Even modern processors like the Intel i7 do this:

- ➢ They combine numbers (like base + offset) to find where to get data from.
- ➢ It is how they find things in memory fast, like looking up items in a list.

8.0 References

1. Intel (n.d.). *Intel® Core^{TM} i7 Processors*. [online] Intel. Available at: https://www.intel.com/content/www/us/en/products/details/processors/core/i7.html.

2. www.lenovo.com. (n.d.). *What Is Intel® Core^{TM} i7? Can I Upgrade From an Older Intel® Core^{TM} Processor? | Lenovo US*. [online] Available at: https://www.lenovo.com/us/en/glossary/what-is-intel-i7/?orgRef=https%253A%252F%252Fwww.google.com%252F.

3. Lutkevich, B. (2020). *What is Cache Memory? Cache Memory in Computers, Explained*. [online] SearchStorage. Available at: https://www.techtarget.com/searchstorage/definition/cache-memory.

4. Intel. (n.d.). *Intel® Core^{TM} i7-1355U Processor (12M Cache, up to 5.00 GHz) - Product Specifications*. [online] Available at: https://www.intel.com/content/www/us/en/products/sku/232160/intel-core-i71355u-processor-12m-cache-up-to-5-00-ghz/specifications.html.

5. Intel. (n.d.). *How to Find the Size of L1, L2, and L3 Cache in Intel® Processors*. [online] Available at: https://www.intel.com/content/www/us/en/support/articles/000057727/processors.html.

6. Razvi (2022). *Razvi Doomun - ICT 1206Y Computer Org & Arch. Level 1 Semester 2*. [online] Google.com. Available at: https://sites.google.com/uom.ac.mu/razvi-doomun/ict-1206y-computer-org-arch-level-1-semester-2 [Accessed: 29 March 2025].

7. Kumar, D. (2023). *Baeldung*. [online] Baeldung on Computer Science. Available at: https://www.baeldung.com/cs/cache-direct-mapping. [Accessed: 29 March 2025].

8. Bharatiya, P. (2023) 'Cache Mapping - The Basics'. Available at:

https://data-intelligence.hashnode.dev/cache-mapping-the-basics [Accessed: 29 March 2025].

9. Hyperskill. (n.d.). *Cache replacement policies | Caching | Software construction | Essentials | Fundamentals | Computer science*. [online] Available at: https://hyperskill.org/learn/step/36825. [Accessed: 29 March 2025].

10. Seal, D. (n.d.). *ARM architecture reference manual*. Harlow, England Addison-Wesley [20]06.

11. Clements, A. (2014). Microprocessor Systems Design: 80x86 Hardware, Software, and Interfacing. Cengage Learning.

12. Hennessy, J.L. and Patterson, D.A. (2019). *Computer architecture : a quantitative approach*. Cambridge, Massachusetts: Elsevier.

13. Hyde, R. (2010). *The Art of Assembly Language, 2nd Edition*. No Starch Press.

14. Intel. (2023). Intel® 64 and IA-32 Architectures Software Developer Manuals. Intel Corporation.

15. Patterson, D. A., & Hennessy, J. L. (2017). Computer Organization and Design: The Hardware/Software Interface (5th ed.). Morgan Kaufmann.

16. Tanenbaum, A. S. (2016). Structured Computer Organization (6th ed.). Pearson.

17. Intel. (n.d.). *Intel® 64 and IA-32 Architectures Software Developer Manuals*. [online] Available at: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

18. In Praise of Computer Organization and Design: The Hardware/ Software Interface, Fifth Edition. (n.d.). Available at: https://theswissbay.ch/pdf/Books/Computer%20science/Computer%20Organization%20and%20Design-%20The%20HW_SW%20Inteface%205th%20edition%20-%20David%20A.%20Patterson%20%26%20John%20L.%20Hennessy.pdf.

19. geeksforgeeks (2016). *Computer Organization and Architecture | Pipelining | Set 1 (Execution, Stages and Throughput) - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/computer-organization-and-architecture-pipelining-set-1-execution-stages-and-throughput/.

20. Varuna Eswer and Naik, S. (2021). Processor performance metrics analysis and implementation for MIPS using an open source OS. *International Journal of Reconfigurable and Embedded Systems (IJRES)*, 10(2), pp.137–137. doi:https://doi.org/10.11591/ijres.v10.i2.pp137-148.

21. Muhammad Hataba (2020). *Pipelining in Modern Processors*. [online] doi:https://doi.org/10.13140/RG.2.2.34414.38720.

22. Anon, (2023). *How does pipelining improve CPU efficiency? | TutorChase*. [online] Available at: https://www.tutorchase.com/answers/ib/computer-science/how-does-pipelining-improve-cpu-efficiency [Accessed 12 Apr. 2025].