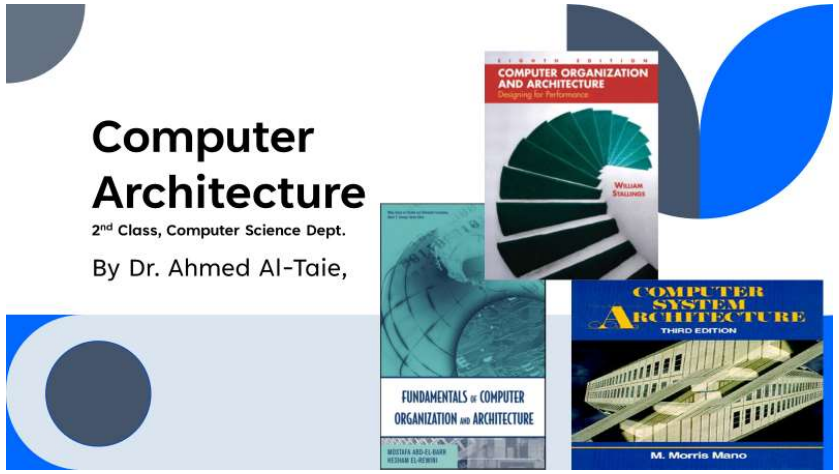
Computer Architecture

2nd Class, Computer Science Dept.

By Dr. Ahmed Al-Taie,

Chapter 3
Assembly Language Programming
Outline

- A Simple Machine
- Instructions Mnemonics and Syntax
- Assembler Directives and Commands
- Assembly and Execution of Programs
- **Example**: The X86 Family



Assembly Language Programming

- In **Chapter 2** we introduced the **basic concepts and principles** involved in the **design of an instruction set** of a machine.
- This chapter considers the issues related to **assembly language programming**. Although **high-level languages and compiler technology** have witnessed **great advances** over the years, **assembly language remains necessary** in some cases.
- **Programming in assembly** can result in **machine code** that is **much smaller** and **much faster** than that generated by **a compiler of a high level language**.
- **Small and fast code** could be **critical** in some **embedded and portable applications**, where **resources may be very limited**.
- In **such cases**, **small portions** of the **program** that may be **heavily used** can be **written in assembly language.**
- **Learning assembly languages** and **writing assembly code** can be **extremely helpful** in **understanding computer organization and architecture**.

Assembly Language Programming

- **A computer program** can be represented at **different levels** of **abstraction**.
- A **program** could be written in a **machine-independent, high-level language** such as **Java or C++.**
- A **computer** can **execute programs only** when they are represented **in machine language specific to its architecture**.
- A **machine language program** for a **given architecture** is a **collection of machine instructions** represented in **binary form**.
- **Programs written** at **any level higher than** the **machine language must be translated to the binary representation** before a **computer can execute them**.
- **An assembly language program** is a **symbolic representation** of the **machine language program.**

Assembly Language Programming

- **Machine language** is pure **binary code**, **whereas assembly language** is **a direct mapping** of the **binary code onto** a **symbolic form** that is **easier** for **humans to understand and manage**.
- **Converting the symbolic representation into machine language** is **performed by a special program** called **the assembler.**
- **An assembler** is **a program that accepts a symbolic language program (source)** and **produces its machine language equivalent (target).**
- In **translating a program into binary code**, **the assembler** will

(1) **replace symbolic addresses** by **numeric addresses**,

(2) **replace symbolic operation codes** by **machine operation codes**,

(3) **reserve storage for instructions and data**, and

(4) **translate constants into machine representation**.

Assembly Language Programming

- The **purpose** of this Lecture is to give you a **general overview of assembly language** and its **programming**.
- We start the chapter with a discussion of a **simple hypothetical machine**
- The **machine has** only **five registers** and its **instruction set has** only **10 instructions**.
- We will use this simple machine to **define a rather simple assembly language** that will be **easy to understand** and will **help explain the main issues** in **assembly programming.**
- We will **introduce instruction mnemonics** and **the syntax** and **assembler directives and commands**.
- A **discussion** on the **execution of assembly programs** is then **presented.**
- We end the chapter by showing a **real-world example** of **the assembly language for the X86 Intel CISC family.**

A SIMPLE MACHINE

Machine language is the native language of a given processor. **Since assembly language is the symbolic form of machine language**, **each different type of processor has its own unique assembly language.**

- Before we study the assembly language of a given processor, we need first to understand the details of that processor.
- We need to know the **memory size** and **organization**, the **processor registers**, the **instruction format, and the entire instruction set**.
- In this section, we **present** a very **simple hypothetical processor**, which will be used in **explaining** the **different topics in assembly language** throughout the chapter.

A SIMPLE MACHINE

Our simple machine is an **accumulator-based processor**, which has **five 16-bit registers**: **Program Counter (PC), Instruction Register (IR), Address Register (AR), Accumulator (AC), and Data Register (DR).**

- **The PC contains** the **address** of the **next instruction** to be **executed**.
- The **IR contains** the **operation code portion** of the **instruction being executed**.
- The **AR contains** the **address portion (if any)** of the **instruction being executed**.
- The **AC serves** as the **implicit source and destination** of **data**.
- The **DR is used** to **hold data**.

The **memory unit** is **made up** of **4096 words** of **storage**. The **word size is 16 bits**. **The processor** is shown in **Figure 3.1**.

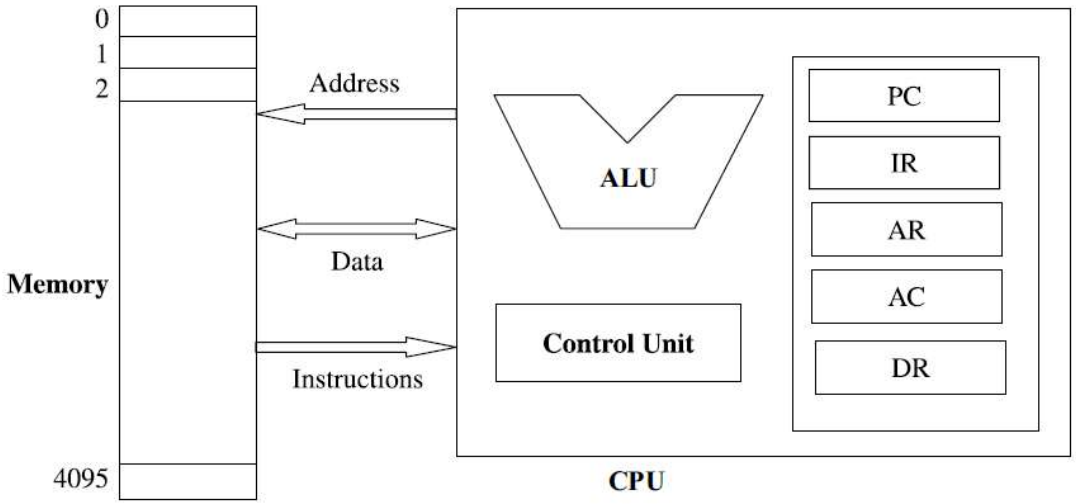


Figure 3.1 A simple machine

A SIMPLE MACHINE

- We assume that **our simple processor** supports **three types** of **instructions**: **data transfer, data processing, and program control.**
- The **data transfer operations** are **load, and store, and copy, move data** between the **registers AC and DR**.
- The **data processing instructions** are **add, subtract, and, and not**.
- The **program control instructions** are **jump and conditional jump**.

- The **instruction set of our processor** is **summarized** in **Table 3.1**.
- The **instruction size** is **16 bits**, **4 bits for the operation code and 12 bits for the address** (when appropriate).

TABLE 3.1 Instruction Set of the Simple Processor

Operation code	Operand	Meaning of instruction
0000		Stop execution
0001	*adr*	Load operand from memory (location adr) into AC
0010	*adr*	Store contents of AC in memory (location adr)
0011		Copy the contents AC to DR
0100		Copy the contents of DR to AC
0101		Add DR to AC
0110		Subtract DR from AC
0111		And bitwise DR to AC
1000		Complement contents of AC
1001	*adr*	Jump to instruction with address adr
1010	*adr*	Jump to instruction adr if AC = 0

A SIMPLE MACHINE : Example 1

- **Example 1:** Let us **write a machine language program** that
- **adds the contents of memory location 12 (00C-hex)**, **initialized to 350** and
- **memory location 14 (00E-hex), initialized to 96,** and
- **store the result in location 16 (010-hex), initialized to 0. The program** is **given in binary instructions** in **Table 3.2**. The **first column** gives the **memory location in binary** for each **instruction and operand.**
- The **second column** lists the **contents** of the **memory locations**.
- **For example**, the **contents** of **location 0** is an instruction with **opcode**: **0001**, and **operand address**: **0000 0000 1100**.
- **Please note** that in the case of **operations** that do **not require operand**, the **operand portion** of the **instruction** is shown as **zeros.**

TABLE 3.2 Simple Machine Language Program in Binary (Example 1)

Memory location (bytes)	Binary instruction	Description
0000 0000 0000	0001 0000 0000 1100	Load the contents of location 12 in AC
0000 0000 0010	0011 0000 0000 0000	Move contents of AC to DR
0000 0000 0100	0001 0000 0000 1110	Load the contents of location 14 into AC
0000 0000 0110	0101 0000 0000 0000	Add DR to AC
0000 0000 1000	0010 0000 0001 0000	Store contents of AC in location 16
0000 0000 1010	0000 0000 0000 0000	Stop
0000 0000 1100	0000 0001 0101 1110	Data value 350
0000 0000 1110	0000 0000 0110 0000	Data value is 96
0000 0001 0000	0000 0000 0000 0000	Data value is 0

A SIMPLE MACHINE : Example 1

- The program is expected to be stored in the indicated memory locations starting at location 0 during execution.
- If the program will be stored at different memory locations, the addresses in some of the instructions need to be updated to reflect the new locations.
- It is clear that programs written in binary code are very difficult to understand and, of course, to debug.
- Representing the instructions in hexadecimal will reduce the number of digits to only four per instruction.
- Table 3.3 shows the same program in hexadecimal.

TABLE 3.3 Simple Machine Language Program in Hexadecimal (Example 1)

Memory location (bytes)	Hex instruction
000	100C
002	3000
004	100E
006	5000
008	2010
00A	0000
00C	015E
00E	0060
010	0000

INSTRUCTION MNEMONICS AND SYNTAX

- **Assembly language** is the **symbolic form** of **machine language**.
- **Assembly programs** are **written** with **short abbreviations** called **mnemonics**.
- A mnemonic is an **abbreviation** that **represents** the **actual machine instruction.**
- **Assembly language programming is the writing of machine instructions in mnemonic form,** where **each machine instruction** (**binary or hex** value) is replaced by **a mnemonic**.

- **Clearly the use of mnemonics is more meaningful than that of hex or binary values, which would make programming at this low level easier and more manageable.**
- **An assembly program consists** of **a sequence of assembly statements**, where statements are written one per line. Each line of an assembly program is split into the following four fields: label, operation code (opcode), operand, and comments.
- **Figure 3.2** shows the **four-column format of an assembly instruction**.

Label (Optional)	Operation Code (Required)	Operand (Required in some instructions)	Comment (Optional)

Figure 3.2 Assembly instruction format

- **Labels are used to provide symbolic** names for **memory addresses**.
- A **label is an identifier** that can be used on a program line in order to branch to the labeled line. It can also be used to access data using symbolic names.

INSTRUCTION MNEMONICS AND SYNTAX

- The **maximum length** of a **label** **differs from** one **assembly language** to another.
- Some allow up to **32 characters** in length, **others may be stricted to six characters.**
- **Assembly languages** for some processors require a **colon after** each label while **others do not**.
- For example, **SPARC** assembly requires **a colon** after every label, but **Motorola** assembly **does not.**
- The Intel assembly requires colons after code labels but not after data labels
- The **operation code (opcode)** field contains the **symbolic abbreviation o**f a given operation.
- The **operand field consists** of additional information or data that the opcode requires.
- The operand field may be used to specify constant, label, immediate data, register, or an address.
- The comments field provides a space for documentation to explain what has been done for the purpose of debugging and maintenance.
- For the simple processor described in the previous section, we assume that the label field, which may be empty, can be of up to six characters.
- There is no colon requirement after each label. Comments will be preceded by "/".

INSTRUCTION MNEMONICS AND SYNTAX

- The simple mnemonics of the ten binary instructions of Table 3.1 are summarized in Table 3.4.
- Let us consider the following assembly instruction:

START LD X \ copy the contents of location X into AC

The label of the instruction LD X is START, which means that it is the memory
address of this instruction. That label can be used in a program as a reference as shown in the following instruction:
BRA START \ go to the statement with label START

TABLE 3.4 Assembly Language for the Simple Processor

Mnemonic	Operand	Meaning of instruction
STOP		Stop execution
LD	*x*	Load operand from memory (location x) into AC
ST	*x*	Store contents of AC in memory (location x)
MOVAC		Copy the contents AC to DR
MOV		Copy the contents of DR to AC
ADD		Add DR to AC
SUB		Subtract DR from AC
AND		And bitwise DR to AC
NOT		Complement contents of AC
BRA	*adr*	Jump to instruction with address *adr*
BZ	*adr*	Jump to instruction *adr* if AC = 0

INSTRUCTION MNEMONICS AND SYNTAX

- The **jump instruction** will make the **processor jump to** the **memory address associated** with the **label START,** thus executing the instruction LD X immediately after the BRA instruction.
- In addition to program instructions, an assembly program may also include pseudo instructions or assembler directives. Assembler directives are commands that are understood by the assembler and do not correspond to actual machine instructions. For example, the assembler can be asked to allocate memory storage.
- In our assembly language for the simple processor, we assume that we can use the pseudo instruction W to reserve a word (16 bits) in memory.

For example, the following pseudo instruction reserves a word for the label X and initializing it to the decimal value 350:

X W 350 \ reserve a word initialized to 350
Again, the label of the pseudo instruction W 350 is X, which means it is the memory address of this value. The following is the assembly code of the machine language program of Example 1 in the previous section.

```
LD X \ AC X
MOV AC \ DR AC
LD Y \ AC Y
ADD \ AC AC þ DR
ST Z \ Z AC
STOP
X W 350 \ reserve a word initialized to 350
Y W 96 \ reserve a word initialized to 96
Z W 0 \ result stored here
```

INSTRUCTION MNEMONICS AND SYNTAX

- **Example 2:** In this example, we will write an assembly program to perform the multiplication operation: Z XY, where X, Y, and Z are memory locations.
- As you know, the assembly of the simple CPU does not have a multiplication operation. We will compute the product by applying the add operation multiple times.
- In order to add Y to itself X times, we will use N as a counter that is initialized to X and decremented by one after each addition step.
- The BZ instruction will be used to test for the case when N reaches 0. We will use a memory location to store N but it will need to be loaded into AC before the BZ instruction is executed.
- We will also use a memory location ONE to store the constant 1. Memory location Z will have the partial products and eventually the final result.
- The following is the assembly program using the assembly language of our simple processor.
- We will assume that the values of X and Y are small enough to allow their product to be stored in a single word.

INSTRUCTION MNEMONICS AND SYNTAX

- For the sake of this example, let us assume that X and Y are initialized to 5 and 15, respectively.

```
LD X \ Load X in AC
ST N \ Store AC (X original value) in N
LOOP LD N \ AC N
BZ EXIT \ Go to EXIT if AC ¼ 0 (N reached 0)
LD ONE \ AC 1
MOVAC \ DR AC
LD N \ AC N
SUB \ subtract 1 from N
ST N \ store decrements N
LD Y \ AC Y
MOVAC \ DR AC
LD Z \ AC Z (partial product)
ADD \ Add Y to Z
ST Z \ store the new value of Z
BRA LOOP
EXIT STOP
X W 5 \ reserve a word initialized to 5
Y W 15 \ reserve a word initialized to 15
Z W 0 \ reserve a word initialized to 0
ONE W 1 \ reserve a word initialized to 1
N W 0 \ reserve a word initialized to 0
```

ASSEMBLER DIRECTIVES AND COMMANDS

- In the previous section, we introduced the reader to **assembly and machine languages**.
- We provided several **assembly code segments** written using our simple machine model.
- In **writing assembly language programs** for a **specific architecture**, a **number** of **practical issues** need to be considered.
- Among these issues are the **following**:
  - **Assembler directives**
  - **Use of symbols**
  - **Use of synthetic operations**
  - **Assembler syntax**
  - **Interaction with the operating system**

- The use of **assembler directives**, also called **pseudo-operations**, is an important issue in writing **assembly language programs**.
- **Assembler directives** are **commands** that are **understood** by the **assembler** and do not **correspond** to **actual machine instructions**.
- **Assembler directives** affect **the way the assembler performs** the **conversion** of **assembly code** to **machine code**.
- **For example**, **special assembler directives** can be used to **instruct the assembler** to **place data items** such that they **have proper alignment**.

ASSEMBLER DIRECTIVES AND COMMANDS

- **Alignment of data** in **memory is required** for **efficient implementation of architectures**.
- For **proper alignment of data, data** of **n-bytes width** must be **stored** at **an address** that is **divisible by n, for example**, a **word that has a two-byte width** has to be stored **at locations** having **addresses divisible by two**.
- In **assembly language programs symbols** are used to **represent numbers**, for example, **immediate data**.
- This is **done** to **make the code easier to read**, **understand, and debug**.
- **Symbols** are **translated to** their **corresponding numerical values by the assembler**.
- The use of **synthetic operations** helps **assembly programmers** to use **instructions** that are not **directly supported** by the **architecture**.
- These are then translated by the assembler to a set of instructions defined by the architecture.
- **For example**, **assemblers** can allow the use of (a **synthetic**) **increment instruction** on **architectures** for which an **increment instruction** is not **defined through** the use of some other **instructions** such as the **add instruction**.

ASSEMBLER DIRECTIVES AND COMMANDS

- **Assemblers** usually **impose** some **conventions** in **referring to hardware components** such as **registers** and **memory locations**.
- One such convention is the **prefixing** of **immediate values** with the **special characters (#)** or a **register name** with the **character (%).**
- The **underlying hardware** in some **machines** cannot be accessed **directly by a program**.
- The **operating system (OS) plays** the **role of mediating access** to **resources** such as **memory and I/O facilities**.
- Interactions with operating systems (OS) can take place in the form of a code that causes the execution of a function that is part of the OS. These functions are called system calls.