Assembly Language Lab 2

What is Assembly Language?

- As we know, computers work only with 0's and 1's. Every program instruction or data element must be in binary to be manipulated by computer machine.
- Therefore, any program understood by machine has to be written in *machine language*, however machine language is too hard to write and maintain.
- Assembly language is developed to make programming easier than programming using machine language. *Assembly language* is a set of mnemonics (symbols) for machine code instructions plus other features that make programming easier.
- To run program written in assembly language, we should have a converter (or translator) which convert these labels and mnemonics to their corresponding machine codes in 0's and 1's. This converter is called *assembler*.

Terminology

Machine Language: is a set of binary codes (0's and 1's) that represent instructions of a specific machine. For example**, 8B D8** means copy content from AX register to BX register. (machine-dependent)

Assembly Language: is a machine-level programming language that uses mnemonics instead of numeric codes to simplify programming. For example, **mov BX, AX** means copy content from AX register to BX register. (machine-dependent)

High-level language: (like C, and C#) it has a lot of features and capabilities that simplify programming too much. (machine-**in**dependent)

Assembler converts assembly programs to an object code that is near to machine language code. **Compiler** converts high-level programs to object code also. **Linker** links many object files in a single executable file.

Why is Assembly Language important

- Assembly language gives the programmer an ability to perform technical tasks that would be difficult in high-level language including total control on the machine.

- Software written in assembly language runs faster than the same one written in high-level language and takes less amount of memory if the programmer well-optimized the assembly program code.
- Learning assembly language gives deep understanding of the computer organization and architecture and how programs run.

Numbering System

Conversion

Bin to Dec: $110110_2 = 2 + 4 + 16 + 32 = 54_{10}$

Dec to Bin: $23_{10} = 10111_2$

(How?) 23 / 2 = 11 *r* 1
11 / 2 = 5 *r* 1
5 / 2 = 2 *r* 1
2 / 2 = 1 *r* 0
1 / 2 = 0 *r* 1 (Quotient == 0? Stop)

Bin to Hex: $\underline{0101}\ \underline{1011}_2 = 5B_{16}$

Hex to Bin: $A6_{16} = \underline{1010}\ \underline{0110}_2$ (Note that 6 is converted in 4-bit also)

Addition and Subtraction

1's complement of $(1011010_2) = 0100101_2$

2's complement of $(1011010_2) = 0100110_2$ (1's complement +1) **N.B.:**
2's complement is used in representing negative numbers

Binary operations:

11001 + 10101 = **1**01110

(Note: the operation result does not fit in 5 bits, so the underlined 1 in the previous number is called *carry*)

11001 – 10101 = 11001 + 01011 = 00100 with carry = 1
(Carry = 1 in subtraction means: result is positive with *no borrow*)

10101 – 11001 = 10101 + 00111 = 11100 with carry = 0
(Carry = 0 in subtraction means: result is negative *with borrow*)

Hexadecimal operations:

23D9 + 94BE = B897
(How?) 9 + 14 = 23 – 16 = **7** with carry
1 + 13 + 11 = 25 – 16 = **9** with carry
1 + 3 + 4 = **8**
2 + 9 = **B**

59F – 2B8 = 2E7
(How?) 15 – 8 = **7**
(9 + 16) – 11 = 14 (**E**)
4 – 2 = **2**

ASCII Code

- All data stored in memory is numeric.
- Characters are stored by using a *character code* that maps numbers to characters.
- One of the most common character codes is known as ASCII (American Standard Code for Information Interchange). It is limited to code only 256 characters
- A new and more complete code that is supplanting ASCII is Unicode. It uses 2 bytes to encode characters. Therefore, it is capable to encode 65535 characters!

Computer Organization

Main Memory

- It is the place to store data (and instructions) temporarily.

- Each location (byte) in memory has content (***value***) and a unique label (***address***).
- Often, memory is used in larger chunks than single bytes. As shown next,

Nibble	4 bits
Byte	8 bits
Word	2 bytes
Double word	4 bytes
Quad word	8 bytes
Paragraph	16 bytes

CPU

- The Central Processing Unit (CPU) is the physical device that performs machine instructions, which are relatively very simple.
- Instructions may require operands (i.e. data items manipulated by instruction) to be stored in special locations in the CPU itself. These locations are called *registers*. Machine instructions generally operate on operands stored in registers only or in a register and a memory item. Therefore, we cannot, for example, add two values stored in memory directly, we have to load at least one of them to a register, then perform addition between a register and a memory item)
- The CPU can access data in registers much faster than those in memory. However, the number of registers in a CPU is limited, so the programmer should only keep currently used data in registers while the other in memory.

- Computers use a clock to synchronize the execution of the instructions. This clock pulses at a fixed frequency (known as the *clock speed*). Every machine instruction requires one or more clock cycle to execute depends on its operational complexity and CPU architecture.

CPU Family

CPU capabilities and organizations differ from one to another. IBM CPUs family (which we concern in this course) mainly includes: **8086.**

- 16-bit registers (AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP, FLAGS). It can address up to 1MB RAM,

- Program memory is divided into segments. Each segment cannot be larger than 64K.
- It runs in ***real mode***, which means a program may access any memory address, even the memory of other programs!

80286.

- It adds some new instructions to 8086.
- Its main new feature is 16-bit ***protected mode***. In this mode, it can access up to 16 megabytes and protect programs from accessing each other's memory.
- However, programs are still divided into segments that could not be bigger than 64K. **80386.**
- It extends many of machine registers to hold 32-bits (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP, EFLAGS) and adds two new 16-bit segment registers FS and GS.
- It also adds a new 32-bit protected mode. In this mode, it can access up to 4 gigabytes of memory.
- Programs are again divided into segments, but now each segment can also be up to 4 gigabytes in size!

Later series (80486, Pentium, Pentium II, III, IV…)

In each, new features are added like cache memory, pipelining, widening data bus and more.

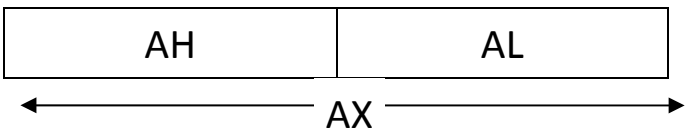
Note: all 80x86 series are **backward compatible**, which means ***architecture*** of later series is compatible with earlier one. In other words, ***programs*** written for earlier series will run as it is on later ones where the reverse is not the case.

Real Mode (8086)

16-bit registers

4 General-purpose registers: AX, BX, CX and DX.

- Each of these registers could be decomposed into two 8-bit registers.



- AH and AL are dependent on AX. Changing AX's value will change AH and AL values and vice versa.
- The general-purpose registers are used in many of the data movement and arithmetic instructions.

2 index registers: SI and DI.

- They are often used as pointers to memory items, but can be used for other purposes as the general-purpose registers.
- They cannot be decomposed into 8-bit registers.

BP and SP registers are used to point to data in ***stack*** and are called the ***Base Pointer*** and ***Stack Pointer***, respectively.

4 segment registers: CS, DS, SS and ES

They keep the starting address of memory chunk used for different parts of a program. CS stands for Code Segment, DS for Data Segment, SS for Stack Segment and ES for Extra Segment. ES is used as a temporary segment register.

Instruction Pointer (IP) register is used with the CS register to keep track of the address of the next instruction to be executed by the CPU. Normally, after an instruction is executed, IP is advanced (*incremented*) to point to the next instruction in memory.

The ***FLAGS*** register stores important information about the results of the last executed operation. This information is stored as individual bits in this register. For example, there is a specific bit called Zero flag (Z flag). This Z flag is 1 if the result of the last operation was zero otherwise Z flag is set to zero. Not all instructions modify the bits in FLAGS.

Memory Segments

- As shown before, memory in real mode (8086) is limited to only one megabyte (2^{20} bytes).
- Valid address range is from (in hex) 00000 to FFFFF. These addresses require a 20-bit number.
- Obviously, a 20-bit number will not fit into any of the 8086's 16-bit registers. So program memory should be divided into *segments* where its starting

address is stored in segment registers. In addition, an *offset* register is used to address memory locations within each segment.

- A segment begins on a *paragraph* boundary (i.e. its address is divisible by 16). Therefore, the starting address of any segment always begins with four 0-bits. By this assumption, it remains only 16 bits vary from segment address to another. Thus, segment address can fit in 16-bit register by only storing the least 16 bits of the address (since we know the other 4 bits are zero).
- A program is often divided into 3 segments, which are Code, Data, and Stack segments. The size of each segment is 64KB **at most**; according to the size the offset register that is, 16 bits (i.e. offset can take value from 0 to 64K).
- Usage of each segment, and segment and offset registers of each segment are shown in the following table:

Segment	Segment Register	Offset Register	Usage
Data	DS	BX, DI, SI	Store the program data
Code	CS	IP	Store the program machine instructions
Stack	SS	SP	Used in saving data elements and addresses temporarily

- To find the physical address (20-bit) from segment-offset pair, use the following relation:

16 * segment register + offset register

(N.B.: multiplying by 16 is equivalent to left shifting the binary value 4 times. This done to return the four 0 bits which not stored physically in the segment register) For example, the physical address referenced by 047C:0048 is given by: 047C0 + 0048 = 04808

Real mode segmented addresses have disadvantages:

- A single segment can only contain 64K of memory (the upper limit of the 16-bit offset register).
- Each byte in memory does not have a unique segmented address. The physical address 04808 can be referenced by 047C:0048, 047D:0038, 047E:0028,…

The First Assembly Program

- Assembly language program is a series of statements which are either computer *instructions* or statements called *directives*.
- Each assembly instruction represents exactly one machine instruction. However, directives do not generate any machine code. Directives are just pseudo-instructions which instruct the assembler how to translate the program into machine code.
- The general form of an assembly instruction is:

[label:] mnemonic [operands] [;comment]

- ***Label***: allows the program to refer to a line of code by a name. (Actually, it is a name for the address of the machine code of this line.)
- ***Mnemonic and operands***: together perform the real work of the program. Mnemonic can be an instruction (like MOV, ADD) or directive (like BYTE, END).
- ***Comment*** is any set of words preceded by a semi-colon.
- As we know, a program is divided into segments. There are directives that allow a programmer to define the starting and the ending of these segments. .DATA, .CODE, and .STACK directives are used to express the starting of data, code, and stack segments respectively. These directives tell the assembler that the next lines in the program belong to the specified segment. The segment ends by defining another segment directive or by the end of code file. *Data segment* is used to define program data. *Stack segment* is used to store data temporarily like saving registers values in the beginning of procedure call. *Code segment* contains the program instructions that do the required work.

- **General Program Skeleton :**

```
.386
.MODEL Flat, STDCALL
.STACK 4096
.DATA
;Your initialized data
.CODE
  <label>
;Your code
ret
  END <label>
```

Explanation for the above skeleton program:

.386

An assembler directive, telling the assembler to use 80386 instructions set and disable later series instructions (like those of 486, Pentium…).

.MODEL *memory_model, [language_type]*

It specifies the program memory model. Allowed models are [TINY, SMALL, COMPACT, MEDIUM, LARGE, HUGE, or FLAT]. *language_type* can be [C, BASIC, FORTRAN, PASCAL, SYSCALL, or STDCALL] and used to specify the compatibility with high level language in case there is an integration between the assembly code and another high level language code. The memory models differ in number of code and data segments and referencing type near or far (explained later). The following table shows the differences among memory models:

Memory Model	Default Code	Default Data	# code segments	# data segments
Tiny (DOS)	Near	Near	Single segment for both	
Small	Near	Near	1	1
Medium	Far	Near	multiple	1
Compact	Near	Far	1	multiple
Large	Far	Far	multiple	multiple
Huge	Far	Far	multiple	multiple
Flat (Windows)	Near	Near	Single segment for both	

Flat memory model and **stdcall** language type are often used with win32 assembly programs. We will use it with the most of programs we write during the course.

.STACK *[stack_size]*

A directive defines the stack segment and its size in bytes. The default size is 1024 byte. Nothing can be placed in the stack segment after its definition. .Stack directive just defines the size of the stack segment without allowing placing initial data in it.

.DATA

It defines the data segment where a programmer can place initialized data variables used in the program.

.CODE

It defines the code segment where assembly instructions are placed. The code segment or the main procedure in it must be ended by ***ret*** instruction (or anything similar that terminates the program execution) which returns control to the OS.

END *<label>*

End directive indicates the end of assembly code in this file. A label must be specified with END directive to tell the assembler where the program *entry point* exists (i.e. from which line the program should begin its execution).

Modified Skeleton Program

Text book we study offers a library which provides some basic operations required in many assembly programs. This library can be used by adding the following line in your program code:

```
INCLUDE Irvine32.inc
```

This line tells the assembler to include all defined functions in this file. It is like **#include** in C++ or **using** in C#. So we use the following modified skeleton program as a template for any program we write later on.

```
INCLUDE Irvine32.inc
.data
.code
main: ;define a label main
exit ;terminate the execution of the program
END main ;end the assembler work, and make the main as program
;entry point
```

Note that the following lines already exist in Irvine32.inc and so they are not written again in our program code.

```
.386
.MODEL Flat, STDCALL
.STACK 4096
```

Also, the code segment or the main procedure in it will be ended by ***exit*** which returns control to Windows after program terminate. It is an EQU for *exitprocess* API function and declared in Irvine32.inc.

Writing First program :

- Writing a program that flushes the values of the registers,
- The used command is DumpRegs

```
INCLUDE Irvine32.inc

.code
main PROC

Call DumpRegs

      exit
main ENDP

END main
```

```
INCLUDE Irvine32.inc

.code
main PROC

Call DumpRegs

    exit
main ENDP

END main
```