Assembly Language and Microcomputer Systems

Module 1

CS 272

Sam Houston State University

Dr. Tim McGuire

Learning Assembly Language

- Assembly language lets you talk to a computer in its “native tongue”
- All computer programs actually run in *machine language*
- High level languages such as C must be translated (*compiled*) into machine language
- Assembly language translates directly into machine language

Advantages of Assembly Language

- Low-level access to the computer
- Higher speed
- Total control over CPU
- (Must know what you are doing in order to make these advantages work)

Disadvantages of assembly language

- Increased risk of bugs
  - subtle mistakes can be more costly
- Reduced portability
  - programs run on only one type of CPU
- Absence of library routines
  - must write your own

Microcomputer Systems

- Before you can program a computer in assembly language, you must learn a little about its architecture
- In this module, you will learn about the main hardware components: the CPU, memory, and peripherals
- You will see what the computer does when it executes an instruction

Digital Circuits

- Integrated-circuit (IC) chips are used in the construction of computer circuits
  - Each IC chip may contain thousands or even millions of transistors
  - These IC circuits are known as ***digital circuits*** because they operate on discrete voltage signal levels, typically, a high voltage and a low voltage
- We use the symbols 0 and 1 to represent the low and high voltage signals
  - These symbols are called ***binary digits*** or ***bits***
  - All information processed by the computer is represented by strings of 0's and 1's; that is by bit strings

The System Board

- Inside the system unit is a main circuit board called the ***system board*** that contains the ***central processing unit*** (*CPU*) and memory circuits
- The system board is also called a ***motherboard*** because it contains expansion slots for additional circuit boards called ***add-in boards***
- In a microcomputer, the CPU is a single-chip processor called a ***microprocessor***

Bytes and Words

- Information processed by the computer is stored in its memory
- A memory circuit element can store one bit of data
- Memory circuits are usually organized into groups that can store eight bits of data, and a string of eight bits is called a ***byte***
- Each ***memory byte*** circuit is identified by a number that is called its address
- The first memory byte has address **0**
- The data stored in a memory byte are called its ***contents***

Address vs. Contents

- The address of a memory byte is fixed and is different from the address of any other memory byte in the computer
- The contents of a memory byte are not unique and are subject to change
- The figure shows the organization of memory bytes; the contents are arbitrary

Address	Contents
.	
.	. . . .
7	0 1 1 0 0 0 0 1
6	1 1 0 0 1 1 1 0
5	0 0 0 0 1 1 0 1
4	0 0 0 0 0 0 0 0
3	1 1 1 1 1 1 1 1
2	0 1 1 0 0 0 0 1
1	0 1 0 1 1 1 1 0
0	0 1 1 0 0 0 0 1

Addresses on Various Processors

- Another distinction between address and contents is that while the contents of a memory byte are always eight bits, the number of bits in an address depends on the processor
  - For example, the Intel 8086 assigns a 20-bit address, the Intel 80286 uses a 24-bit address, and the 386 uses a 32-bit address.
- The number of bits used in the address determines the number of bytes that can be accessed by the processor

Example

- Suppose a processor uses 20 bits for an address. How many memory bytes can be accessed?
- Solution:
  - A bit can have two possible values, so in a 20-bit address there can be 2^{20} = 1,048,576 different values, with each value being the potential address of a memory byte
  - In computer lingo, the number 2^{20} is called 1 ***mega*** -- thus a 20-bit address can be used to address one ***megabyte*** or 1*MB*

Words

- In a typical microcomputer, two bytes form a word
- The 80x86 family allows any pair of successive memory bytes to be treated as a single unit, called a memory word
  - The lower address of the two bytes is used as the address of the memory word
  - Thus the word with the address 2 is made up of the bytes with addresses 2 and 3
- The microprocessor can tell from the context whether an address refers to a byte or a word

Memory Operations

- The processor can perform two operations on memory:
  - **fetch** -- read the contents of a location
  - **store** -- write data to a location
- In a fetch, the processor only gets a copy of the data -- the original contents are unchanged
- In a store, the data written become the new contents of the location -- the original contents are lost

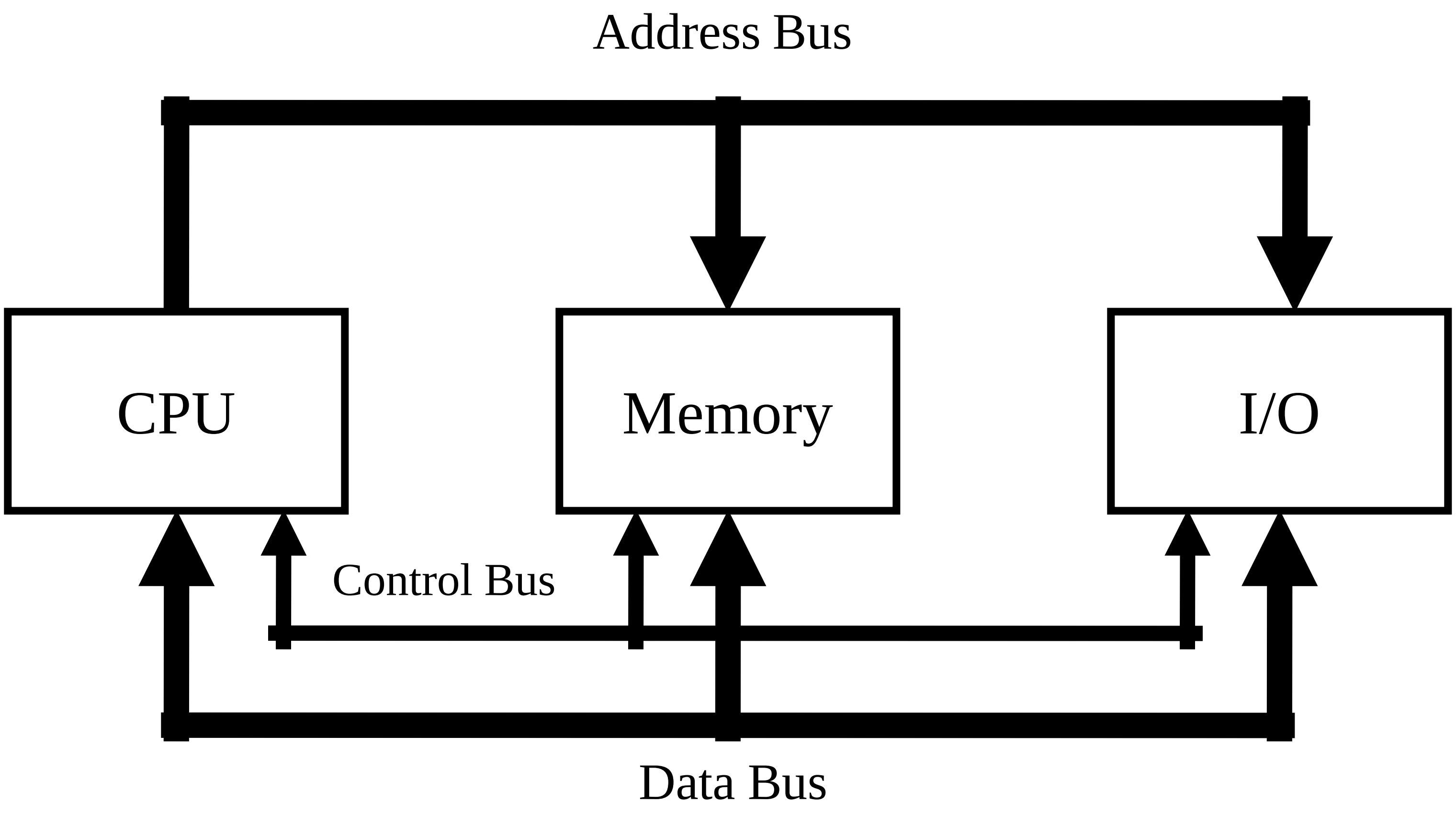
Buses

- A processor communicates with memory and I/O by using signals that travel along a set of wires called a ***bus*** that connect the different components
- There are three buses:
  - ***address bus***
  - ***data bus***
  - ***control bus***

Bus Example

- To fetch the contents of a memory location
  - The CPU places the address of the memory location on the address bus
  - The CPU sends a control signal to the memory circuits telling it to read from memory on the control bus
  - It receives the data, sent by the memory circuits, on the data bus

Bus Connections of a Microcomputer



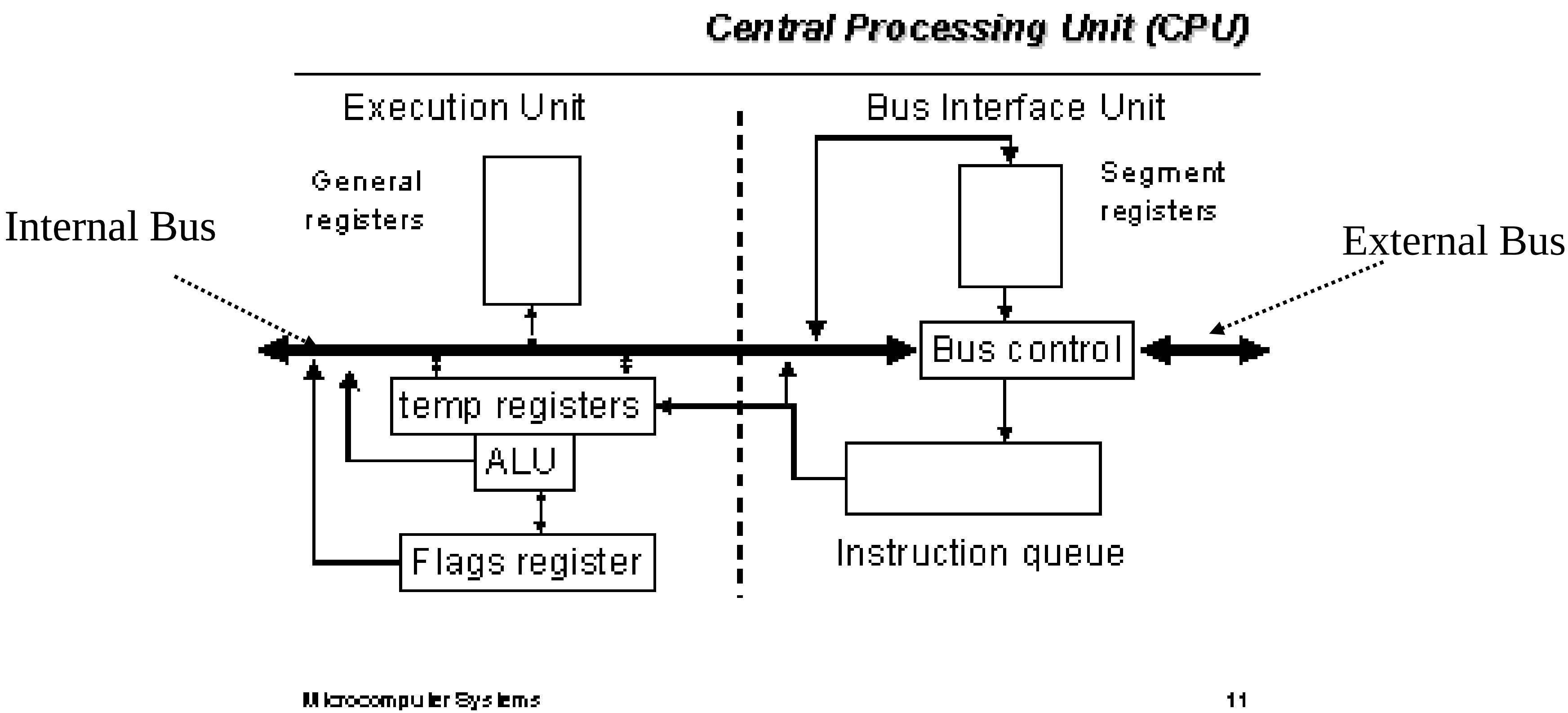
The CPU

- The CPU controls the computer by executing programs stored in memory
- Each instruction is a bit string
  - For the 80x86 family, instructions are from one to six bytes long
- This language of 0's and 1's is called ***machine language***

The Instruction Set

- The instructions performed by a CPU are its instruction set
  - The instruction set for each type of CPU is unique
- Machine language instructions are designed to be simple
- Complex programs are just a sequence of very basic operations

Intel 80x86 Microprocessor Organization



This is the 8086 – other family members are more complex, but the basic principles apply

Execution Unit (EU)

(not just in Huntsville)

- Executes instructions, obviously
- Contains a circuit called the ***arithmetic and logic unit (ALU)***
  - The ALU can perform arithmetic (+,-,*,/) and logic (AND, OR, NOT) operations
  - The data for the operations are stored in ***register*** circuits
- Registers are like memory, only
  - much faster
  - referenced by name instead of number address

The EU's Registers

- The EU has eight registers for storing data
  - AX, BX, CX, DX, SI, DI, BP, and SP
  - We will speak more of these later
- In addition, there are
  - unnamed temporary registers for holding operands for the ALU
  - the FLAGS register whose individual bits reflect the result of a comparison

Bus Interface Unit (BIU)

- Facilitates communication between the EU and the memory or I/O circuits
- Responsible for transmitting addresses, data, and control signals on the buses
- Its registers are named CS, DS, ES, SS, and IP; they hold addresses of memory locations
- *IP* (*instruction pointer*) contains the address of the next instruction to be executed by the EU

The Internal Bus

- The EU and BIU are connected by an internal bus -- they work together
- While the EU is executing an instruction, the BIU fetches up to six bytes of the next instruction and places them in the instruction queue
  - This operation is called ***instruction prefetch***
  - Its purpose is to speed up the processor

Instruction Execution

- A machine instruction has two parts:
  - an ***opcode***, and
  - ***operands***
- The opcode specifies the type of operation
- The operands are often memory addresses to the data

The Fetch-Execute Cycle

- The CPU goes through the following steps to execute a machine instruction:
- Fetch
  - fetch an instruction from memory
  - decode the instruction to determine the operations
  - fetch data from memory if necessary
- Execute
  - perform the operation on the data
  - store the result in memory if needed

Example

- The instruction that adds the contents of register AX to the contents of memory word 0 is

`00000001 00000110 00000000 00000000`

- The first byte of the instruction is stored at the location indicated by the IP
- **1. Fetch the instruction**
  - The BIU places a memory read request on the control bus and the address of the instruction on the address bus
  - Memory responds by sending the contents of the location specified -- namely, the instruction just given -- over the data bus
  - The CPU accepts the data and adds 4 to the IP so that the IP will contain the address of the next instruction

Example, continued

- **2. Decode the instruction**
  - On receiving the instruction, a decoder circuit in the EU decodes the instruction and determines that it is an ADD operation involving the word at address 0
- **3. Fetch data from memory**
  - The EU informs the BIU to get the contents of memory word 0
  - The BIU sends address 0 over the address bus and a memory read request is again sent over the control bus
  - The contents of memory word 0 are sent back over the data bus to the EU and placed in a holding register
- **4. Perform the operation**
  - The contents of the holding register and the AX register are sent to the ALU, which performs the addition and holds the sum

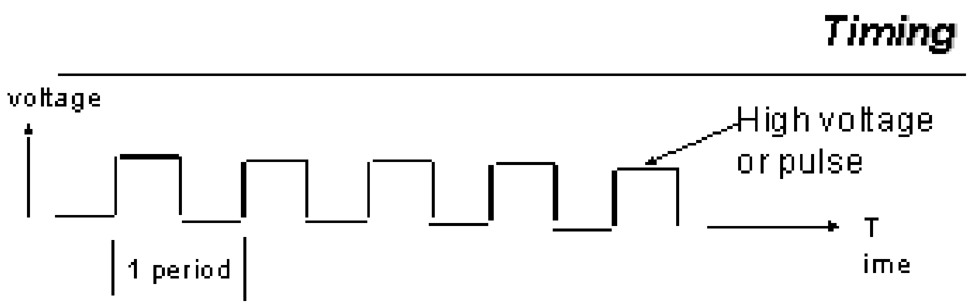
Example, continued

- **5. Store the result**
  - The EU directs the BIU to store the sum at address 0
  - To do so, the BIU sends out a memory write request over the control bus, the address 0 over the address bus, and the sum to be stored over the data bus
  - The previous contents of memory word 0 are overwritten by the sum

- The cycle is repeated for the instruction whose address is now contained in the IP

Timing

- Even though machine instructions are very simple, their execution is actually quite complex.
- A clock circuit controls the processor by generating a train of ***clock pulses***



Clock period - time interval between two pulses
Clock rate or speed - number of pulses per second
1 MHz = 1 million cycles per second

- The original IBM PC had a clock rate of 4.77MHz, but current Intel chips have clock rates of over 2000MHz, or 2GHz

Programming Languages

- **Machine Language**
  - A CPU can only execute machine language instructions (which are bit strings)

Machine Instruction	Operation
10100001 00000000 00000000	Fetch contents of memory word 0 and put it in register AX
00000101 00000100 00000000	Add 4 to AX
10100011 00000000 00000000	Store the contents of AX in memory word 0

- As you can imagine, writing programs in machine language is tedious and subject to error

Assembly Language

- A more convenient language to use is assembly language
- In it, we use symbolic names to represent operations, registers, and memory locations
- If location 0 is symbolized by A, the preceding program would look like this:

Assembly Language Example

Assembly Language Instruction	Operation
`MOV AX,A`	Fetch contents of memory word 0 and put it in register AX
`ADD AX,4`	Add 4 to AX
`MOV A,AX`	Store the contents of AX in memory word 0

- A program written in assembly language must be converted to machine language before the CPU can execute it
- A program called an ***assembler*** translates each assembly language statement into a single machine language instruction

High-Level Languages

- Assembly language is easier than machine language, but it's still difficult because the instruction set is so primitive
- That is why high-level languages like C were developed
- A program called a ***compiler*** is needed to translate a high-level language program into machine code
- A high-level language statement typically translates into many machine language instructions

Sample Assembly Language Program (Linux-nasm)

```
;  hello.asm  a first program
   SECTION .data           ; data section
msg:  db "Hello World",10 ; the string to print, 10=cr
len:  equ $-msg

   SECTION .text           ; code section
   global main             ; make label available to linker
main:                      ; standard  entry point
   mov edx,len             ; arg3, length of string to print
   mov ecx,msg             ; arg2, pointer to string
   mov ebx,1               ; arg1, where to write, screen
   mov eax,4               ; write sysout command
   int 80h                 ; interrupt 80 hex, call kernel
   mov ebx,0               ; exit code, 0=normal
   mov eax,1               ; exit command to kernel
   int 80h                 ; interrupt 80 hex, call kernel
```