DAKOTA STATE UNIVERSITY

Assembly Language and Machine language (part 2)

Jihene Kaabi & Youssef Harrath

Data Transfer Instructions

The mov instruction

- The mov instruction, which requires two operands and has the syntax

  mov destination,source

- The data are copied from source to destination, and the source operand remains unchanged.

- Both operands should be of the same size.

DAKOTA STATE UNIVERSITY

The mov instruction (cont.)

- The mov instruction can take one of the following five forms:

mov Form	Restrictions
mov register, register	• Destination register cannot be CS or (E)IP registers. • Both registers cannot be segment registers.
mov register, immediate	Register cannot be a segment register.
mov memory, immediate	
mov register, memory	
mov memory, register	

There is no move instruction to transfer data from memory to memory, as the Pentium does not allow it.

Arithmetic Instructions

- The Pentium provides several instructions to perform simple arithmetic operations.
- we will describe five instructions to perform addition, subtraction, and comparison.
- Arithmetic instructions update the status flags, to record the result of the operation.

Increment/Decrement Instructions

- These instructions can be used to either increment or decrement the operands.
- The **inc** (increment) instruction adds one to its operand.
- The **dec** (decrement) instruction subtracts one from its operand.
- Both instructions take a single operand. The operand can be either in a register or in memory.
- ***e.g.,*** **inc ECX ; ECX++**

ADD Instructions

- The add instruction can be used to add two 8-, 16-, or 32-bit operands.
    - The syntax is ***add destination, source***
- As with the mov instruction, add can also take the five basic forms depending on how the two operands are specified.
- The semantics of the add instruction are ***destination = destination + source***
- As a result, destination loses its contents before the execution of add but the contents of source remain unchanged.

DAKOTA STATE UNIVERSITY

Subtract Instructions

- The ***sub*** (subtract) instruction can be used to subtract two 8-, 16-, or 32-bit numbers.
  - The syntax is ***sub destination, source***

- The source operand is subtracted from the destination operand and the result is placed in the destination.

destination = destination - source

Compare Instruction

- The ***cmp*** (compare) instruction is used to compare two operands (equal, not equal, and so on).

  cmp destination, source

- Subtracts the source operand from the destination operand but does not alter any of the two operands, as shown below:

  destination – source

- The flags are updated as if the sub operation were performed.
- The main purpose of the ***cmp*** instruction is to update the flags so that a subsequent conditional jump instruction can test these flags.

Directives

- A directive is an artifact of the assembler not the CPU.
- They are generally used to either instruct the assembler to do something or inform the assembler of something.
- They are not translated into machine code.
- Common uses of directives are:
  - define constants
  - define memory to store data into group memory into segments
  - conditionally include source code
  - include other files

Directives (cont.)

- NASM (Netwide Assembler) code passes through a preprocessor just like C.
- It has many of the same preprocessor commands as C.
- However, NASM's preprocessor directives start with a **%** instead of a # as in C.

The EQU Directive

- The ***EQU directive*** can be used to define a symbol.
- Symbols are named constants that can be used in the assembly program.
- The format is: ***symbol EQU value***
- Symbol values can not be redefined later.
- E.g.

```
Num_Of_Students    EQU  90
mov AX, Num_Of_Students
```

The %define Directive

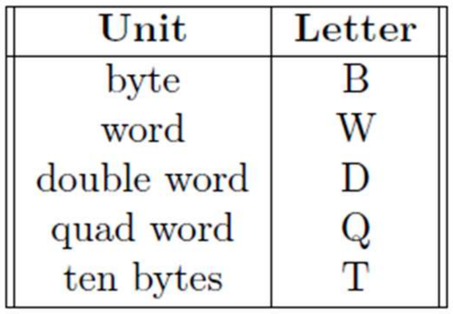
- This directive is like C's #define directive. It is most used to define constant macros just as in C.

```
%define SIZE 100
 mov eax, SIZE
```

- The above code defines a macro named SIZE and shows its use in a MOV instruction.
- Macros are more flexible than symbols in two ways.
  - Macros can be redefined and can be more than simple constant numbers.

The Data Directive (1/4)

- Data directives are used in data segments to define room for memory.
- There are two ways memory can be reserved.
  - The first way only defines room for data;
  - The second way defines room and an initial value.
- The first method uses one of the RESX (Reserve Memory) directives. The X is replaced with a letter that determines the size of the object (or objects) that will be stored.



Unit	Letter
byte	B
word	W
double word	D
quad word	Q
ten bytes	T

: Letters for `RESX` and `DX` Directives

The Data Directive (2/4)

- The second method (that defines an initial value, too) uses one of the DX directives. The X letters are the same as those in the RESX directives.
- It is very common to mark memory locations with labels. Labels allow one to easily refer to memory locations in code. Below are several examples:

```
L1    db    0        ; byte labeled L1 with initial value 0
L2    dw    1000     ; word labeled L2 with initial value 1000
L3    db    110101b  ; byte initialized to binary 110101 (53 in decimal)
L4    db    12h      ; byte initialized to hex 12 (18 in decimal)
L5    db    17o      ; byte initialized to octal 17 (15 in decimal)
L6    dd    1A92h    ; double word initialized to hex 1A92
L7    resb  1        ; 1 uninitialized byte
L8    db    "A"      ; byte initialized to ASCII code for A (65)
```

The Data Directive (3/4)

```
L1    db    0        ; byte labeled L1 with initial value 0
L2    dw    1000     ; word labeled L2 with initial value 1000
L3    db    110101b  ; byte initialized to binary 110101 (53 in decimal)
L4    db    12h      ; byte initialized to hex 12 (18 in decimal)
L5    db    17o      ; byte initialized to octal 17 (15 in decimal)
L6    dd    1A92h    ; double word initialized to hex 1A92
L7    resb  1        ; 1 uninitialized byte
L8    db    "A"      ; byte initialized to ASCII code for A (65)
```

- Double quotes and single quotes are treated the same.
- Consecutive data definitions are stored sequentially in memory.
  - That is, the word L2 is stored immediately after L1 in memory.

The Data Directive (4/4)

- The DD directive can be used to define both integer and single precision floating point constants.
- However, the DQ (**D**efine **Q**uad word) can only be used to define double precision floating point constants.
- For large sequences, NASM's ***TIMES*** directive is often useful. This directive repeats its operand a specified number of times.
  - E.g.,
    - L12 times 100 db 0 ; equivalent to 100 (db 0)'s
    - L13 resw 100 ; reserves room for 100 words

The Data Directive: Labels

- Labels can be used to refer to data in code.
- There are two ways that a label can be used.
    - If a plain label is used, it is interpreted as the address (or offset) of the data.
    - If the label is placed inside square brackets ([ ]), it is interpreted as the data at the address.
- In other words, one should think of a label as a pointer to the data and the square brackets dereferences the pointer just as the asterisk does in C.
- In 32-bit mode, addresses are 32-bit.

Labels : Examples

```
mov    al, [L1]      ; copy byte at L1 into AL
mov    eax, L1       ; EAX = address of byte at L1
mov    [L1], ah      ; copy AH into byte at L1
mov    eax, [L6]     ; copy double word at L6 into EAX
add    eax, [L6]     ; EAX = EAX + double word at L6
add    [L6], eax     ; double word at L6 += EAX
mov    al, [L6]      ; copy first byte of double word at L6 into AL
```

- Labels can be used to refer to data in code.
- An important property of NASM is that the assembler does not keep track of the type of data that a label refers to.
- It is up to the programmer to make sure that he (or she) uses a label correctly.
- Later it will be common to store addresses of data in registers and use the register like a pointer variable in C.

Labels : Examples (cont.)

- Again, no checking is made that a pointer is used correctly. In this way, assembly is much more error prone than even C.
- Consider the following instruction:

```
mov [L6], 1 ;   store a 1 at L6
```

  - This statement produces an operation size not specified error. Why?
  - Because the assembler does not know whether to store the 1 as a byte, word or double word.
  - To fix this, add a size specifier:

```
mov dword [L6], 1 ;     store a 1 at L6
```

  - This tells the assembler to store a 1 at the double word that starts at L6.
- Other size specifiers are: BYTE, WORD, QWORD and TWORD (defines a ten byte area of memory).

Input and Output

- Input and output are very system dependent activities. It involves interfacing with the system's hardware.
- High level languages, like C, provide standard libraries of routines that provide a simple, uniform programming interface for I/O.
  - Assembly languages provide no standard libraries.
  - They must either directly access hardware (which is a privileged operation in protected mode) or use whatever low-level routines that the operating system provides.

Input and Output Routines

print_int	prints out to the screen the value of the integer stored in EAX
print_char	prints out to the screen the character whose ASCII value stored in AL
print_string	prints out to the screen the contents of the string at the *address* stored in EAX. The string must be a C-type string (*i.e.* null terminated).
print_nl	prints out to the screen a new line character.
read_int	reads an integer from the keyboard and stores it into the EAX register.
read_char	reads a single character from the keyboard and stores its ASCII code into the EAX register.

Print Routines

- To use one of the print routines, you must load EAX with the correct value and uses a CALL instruction to invoke it.
  - The CALL instruction is equivalent to a function call in a high-level language. It jumps execution to another section of code but returns to its origin after the routine is over.
  - E.g.,

    ```
    mov eax, 123
    call print_int
    call print_nl
    ```

Debugging Routines

- Debugging routines display information about the state of the computer without modifying the state.
- These routines are really ***macros*** that preserve the current state of the CPU and then make a subroutine call.
- Macros are used like ordinary instructions. Operands of macros are separated by commas.
- There are four debugging routines named **dump regs**, **dump mem**, **dump stack**, and **dump math**; they display the values of registers, memory, stack and the math coprocessor, respectively.

dump_regs macro

- This macro prints out the values of the registers (in hexadecimal) of the computer to stdout (i.e. the screen).
- It takes a single integer argument that is printed out as well.
    - This can be used to distinguish the output of different dump regs commands.

```
dump_regs 1  ; print out the values of the registers
```

dump_mem macro

- This macro prints out the values of a region of memory (in hexadecimal) and also as ASCII characters.
- It takes three comma delimited arguments.
  - The first is an integer that is used to label the output (just as dump regs argument).
  - The second argument is the address to display. (This can be a label.)
  - The last argument is the number of 16-byte paragraphs to display after the address.
- The memory displayed will start on the first paragraph boundary before the requested address.

`dump_mem 1 , msg, 1` ; print out the memory

dump_stack macro

- This macro prints out the values on the CPU stack. (The stack will be covered in Chapter 4.)
- The stack is organized as double words, and this routine displays them this way.
- It takes three comma delimited arguments.
  - The first is an integer label (like dump regs).
  - The second is the number of double words to display below the address that the EBP register holds.
  - The third argument is the number of double words to display above the address in EBP.

```
dump_stack 3 , 8, 8 ; print out the stack
```

Dump_math macro

- This macro prints out the values of the registers of the math coprocessor.
- It takes a single integer argument that is used to label the output just as the argument of dump regs does.

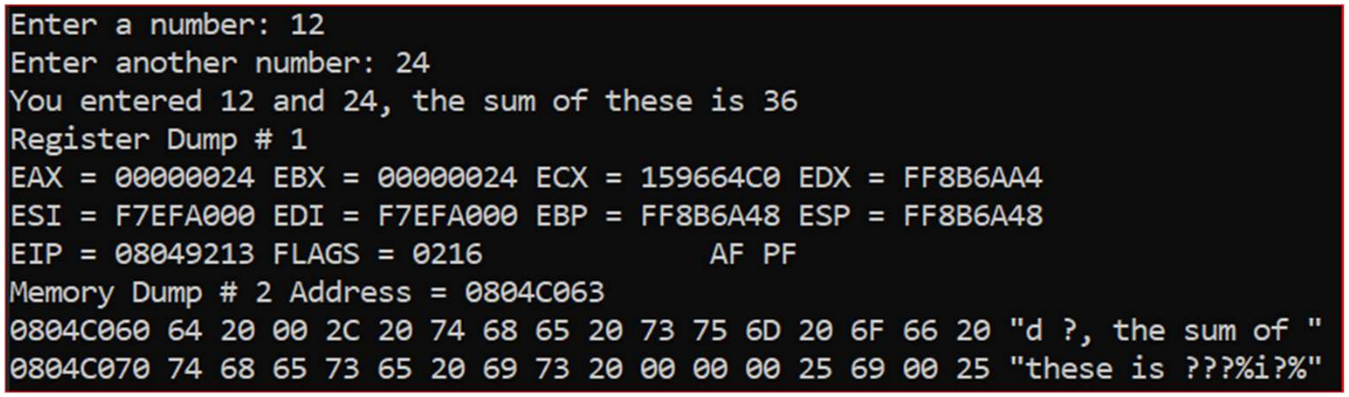
`dump_math 1` ; print out the registers of the math coprocessor

Exercise

Write an assembly program that

1. prompts the user to enter two numbers,
2. prints a message to show to the user the entered numbers,
3. calculate the difference between the two entered numbers,
4. prints the results to the screen, and
5. prints the values of the registers and the memory.

- **The output should look like:**



```
Enter a number: 12
Enter another number: 24
You entered 12 and 24, the sum of these is 36
Register Dump # 1
EAX = 00000024 EBX = 00000024 ECX = 159664C0 EDX = FF8B6AA4
ESI = F7EFA000 EDI = F7EFA000 EBP = FF8B6A48 ESP = FF8B6A48
EIP = 08049213 FLAGS = 0216          AF PF
Memory Dump # 2 Address = 0804C063
0804C060 64 20 00 2C 20 74 68 65 20 73 75 6D 20 6F 66 20 "d ?, the sum of "
0804C070 74 68 65 73 65 20 69 73 20 00 00 00 25 69 00 25 "these is ???%i?%"
```