
Lecture 3: Basic Instructions

Dragan Mirkovic
Department of Computer Science
University of Houston

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Introduction

- Announcements:
 - Homework #1 due today
 - Quiz #1 on Thursday, 9/11/2003
 - Number representations
 - Assembler overview
 - Questions?
- Today's lecture:
 - Basic Instructions in IBM PC Assembler
 - Addressing, Macros, Integer arithmetic

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

The Big Picture

- We are using a hands-on approach
- At this point our knowledge of MASM consists only of bits and pieces that allow us to write simple programs
- How those fit into the big picture?
- What is the big picture anyway?
- How do we define a programming language?

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Language definition

- Complete specification of the of the rules used to define a computer program
 - (formal) grammar (syntax rules)
 - semantics (meaning)
- Language reference manual
 - Example: ANSI C Standard

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Program Elements

- One or more *translation units* stored in files
- The 1st phase of translation produces a sequence of tokens
- Token classes:
 - Identifiers
 - Reserved words (keywords)
 - Constants and string literals
 - Operators
 - Separators (white space: blanks, tabs, comments,...)

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Identifiers

- An identifier is a sequence of letters and digits representing variables, constants, procedure names, labels, segment names, and user defined data types
- Restrictions:
 - Length: 246 characters (247 generates an error)
 - The 1st char can be (A-z) or any of @_\$_?
 - The other characters (A-z), @_\$_?, (0-9)
- Recommendations:
 - Avoid starting an identifier with @

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Reserved words

- Have special meaning fixed by the language
 - Instructions (ex. `mov`, `int`, `lea`)
 - Directives (ex. `.DATA`, `.CODE`, ...)
 - Operators: (ex. `(,)`, `[,]`, `+`, `-`, `*`, `LENGTH`, ...)
 - Operands and attributes (ex. `BYTE`, `WORD`, `$`, `C`, `DD`, ...)
 - Predefined symbols (ex. `@data`, `@code`, `@Cpu`, `@Date`, `@Line`, ...)

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Constants and constant expressions

- Integers
 - Sequence of digits + radix specifier
 - Ex. `31`, `0B5h`
 - Default radix is decimal
 - The specifiers are:
 - `y` or `b` for binary
 - `o` or `q` for octal
 - `t` or `d` for decimal
 - `h` for hexadecimal
 - Lower or upper case
 - You can change the default with the `.RADIX` directive
- Floating point – latter
 - Ex. `myshort REAL4 21.2345 ;`

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Symbolic Integer constants

- Defined using the data assignment directives
 - EQU doesn't allow redefinition
 - = (the equal sign) allows redefinition
- Examples:

```
col EQU 80 ;
row EQU 25 ;
scr EQU col * row ;
age = 33 ;
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Operators

- Used in expressions whose value is determined at assembly time and it does not change at run-time
- Should not be confused with processor instructions (**ADD** and **+**)
- Assembler evaluates expressions according to the following rules:
 1. Parentheses
 2. Binary operations
 3. Left to right for equal precedence
 4. Unary operations of equal precedence are performed right to left

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Using Addresses

- Programming segmented addresses
 - Before a segmented address is used, segment registers need to be initialized
 - Simplified segment directives handle most of the initialization process
 - Near addresses have an implied segment name associated with it
- Addressing operands
 - Registers `mov ax, bx`
 - Immediate `mov cx, 20`
 - Direct memory `mov ax, var`
 - Indirect memory `mov ax, [bx]`

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Addressing operators

- The address consists of the segment and the offset part
- The **OFFSET** Operator:
 - Returns the offset of a memory location
 - Example
`mov bx, OFFSET var ;`
- The **SEG** Operator
 - Returns the segment of the memory location
 - Example
`mov ax, SEG farvar ;`
`mov es, ax`

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Addressing operators...

- One can use expressions to specify memory locations
- Operators: Plus (+), Minus (-), Index ([]), ...
- We will see more details later

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Macro instructions

- Writing directly in assembly language is wordy and tedious
- Many things are often repeated
- One can use a single name for frequently used sequences of instructions
- Macro Instructions
- Definition using the MACRO directive

```
Name    MACRO
...
        ENDM
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Macro Example

```
_Exit    MACRO    ReturnVal
IFNB    <ReturnVal>
        mov     al, ReturnVal
ENDIF

        mov     ah, 4ch
        int     21h
        ENDM
```

- This macro is from PCMAC.INC
- Note: IFNB/ENDIF conditional assembly directives
IFB, (IFNB) the argument is blank, (not blank)

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Using Macros

```
.CODE
Hello  PROC
        mov     ax, @data
        mov     ds, ax
        mov     dx, OFFSET Message
        mov     ah, 9h
        int     21h
        mov     al, 0 ;
        mov     ah, 4ch;
        int     21h
Hello  ENDP
END    Hello ;

        .CODE
Hello  PROC
        _Begin
        _PutStr Message
        _Exit 0;
Hello  ENDP
END    Hello
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Using Macros...

- Macros may change the register values
- Example:

```
Mov     dx, 14
_PutStr Message
Mov     A, dx;      Does NOT set A to 14
```

- Macros could save and restore the registers, but this is inefficient in most of the cases

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Using Macros...

- There is a convenient set of macros in PCMAC.INC and you should learn how to use them
- You have to include this file using INCLUDE directive
- There are two versions of the macros: safe and unsafe
- Macros are used to simplify writing of your codes but you should understand the instructions they hide (at least in due time)

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Integer Arithmetic

- Addition and subtraction

```
add    dest, source; dest += source
sub    dest, source; dest -= source
```
- The operands are the same as for `mov`

```
add/sub    reg/mem, reg/mem/const
```
- The operands must be of the same size, and at least one operand must not be memory.
- Remember:

<code>eax, ebx, ...</code>	double word	32-bit
<code>ax, bx, ...</code>	word	16-bit
<code>ah, al, ...</code>	byte	8-bit

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Examples

- Assembly code for: $A = A + B$

```
mov    ax, B;    ax = B
add    A, ax;    A = A + B
```
- Note that

```
add    A, B;    ILLEGAL - two memory operands
```
- Assembly code for: $A = B - C + 3$

```
mov    ax, B;    ax = B
sub    ax, C;    ax = B - C
add    ax, 3;    ax = B - C + 3
mov    A, ax;    A = B - C + 3
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Examples...

- The increment/decrement and sign change instructions

$A = A + 1$, $A = A - 1$, $A = -A$

occur frequently

- Implemented in hardware on many microprocessors

- On 80x86 instructions are

<code>inc/dec</code>	<code>reg/mem</code>
<code>inc ax</code>	<code>ax = ax + 1</code>
<code>dec ax</code>	<code>ax = ax - 1</code>
<code>neg ah</code>	<code>ah = -ah</code>

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Integer Multiplication

- Integer multiplication and division are much more complicated operations

- The result of multiplication is bigger than the factors

- The special set of registers is assumed depending on the operand size
- One factor must be in the accumulator register

- There is no simple way for dealing with the sign:

- 80x86 has two different operations implemented
 - `mul` for unsigned integers
 - `imul` for signed integers

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Use of Registers for Multiplication

Operand size	Multiplicand	Multiplier	Product
BYTE	AL	Register or Memory	AX
WORD	AX		AX(low) and DX(high)
DWORD	EAX		EAX(low) and EDX(high)

- Only the multiplier is specified explicitly
- The other operands are in predefined positions
- Constant multiplier is not allowed
 - Constant needs to be loaded into register first
 - Three additional operand combinations on 80186 and up including the multiplication by a constant

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Examples

- If we want to calculate $A = B * C$, and A, B and C are signed word variables

```
mov    ax, B
imul   C
mov    A, ax; high word ignored!
```

- The carry and overflow flags are set if high byte/word/dword is not 0.
- Example:

```
mov    al, 20 ;
mov    bl, 15 ;
mul    bl ; Product in AX, overflow and carry set
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Debugger Example

- Microsoft DEBUG is convenient tool for assembling and testing short programs
- Other (more sophisticated) tools: PWB, CodeView, etc.
- DEBUG displays the contents of memory, registers and flags as they change during the execution

```
-a 100 ; Assemble at CS:100h
0B1D:0100 mov al, 20
0B1D:0102 mov bl, 15
0B1D:0104 mul bl
0B1D:0106
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B1D ES=0B1D SS=0B1D CS=0B1D IP=0100 NV UP EI PL NZ NA PO NC
0B1D:0100 B020 MOV AL,20
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Debugger Example

```
0B1D:0100 B020 MOV AL,20
-t
AX=0020 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B1D ES=0B1D SS=0B1D CS=0B1D IP=0102 NV UP EI PL NZ NA PO NC
0B1D:0102 B315 MOV BL,15
-t
AX=0020 BX=0015 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B1D ES=0B1D SS=0B1D CS=0B1D IP=0104 NV UP EI PL NZ NA PO NC
0B1D:0104 F6E3 MUL BL
-t
AX=02A0 BX=0015 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B1D ES=0B1D SS=0B1D CS=0B1D IP=0106 OV UP EI PL NZ NA PO CY
```

Result in AX

Overflow and carry flags set

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Integer division

- Needs more space for the dividend (similar to product in `mul`)
- The result is quotient q and remainder r where
 $a/b = (q, r)$ imply $a = q*b + r$
- The instructions are:
`div reg/mem; unsigned divide`
`idiv reg/mem; signed divide`
- The registers used for Integer division

Operand size	Dividend	Divisor	Quotient and Remainder
BYTE	AX	Register or Memory	AL and AH
WORD	AX and DX		AX and DX
DWORD	EAX and EDX		EAX and EDX

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Integer division...

- Usually all the operands we use are of the same size
 - BYTE, WORD, DWORD
- How to extend the dividend into a word/dword/qword
- Easy for unsigned and positive numbers – zero the high-order bits
- Example: $A = B/C$, A,B,C are unsigned words

```
mov ax, B
mov dx, 0 ; or sub dx, dx
div C
mov A, ax;
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Sign Extension

- For negative signed numbers put ones in the high part
- 80x86 has a special instructions for sign extension

```
cbw      Convert Byte to Word   (AL->AX)
cwd      Convert Word to Dword  (AX->DX:AX)
cdq      Convert DWord to QWord (EAX->EDX:EAX)
cwde     Convert Word to Dword  (AX->EAX)
```

- Example: $A = B/C$, A,B,C are unsigned words

```
mov     ax, B
cwb
idiv   C
mov     A, ax;
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Important Reminders

- Multiplication and division can take place **only** in the specified register combinations.
- There is no unsigned multiply or any divide by a **constant**.
- You **must** convert dividends (8 to 16, 16 to 32 and 32 to 64-bit) before `div/divid`.

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Examples

```
mov    al, -1;
mov    bl, -1;
mul    bl;
```

- What is in `ax` ?
`ax = 0FE01h = 65,025 = 255*255`
- The numbers are treated as unsigned

```
mov    al, -1;
mov    bl, -1;
imul   bl;
```

- Now `ax = 1`

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Computer Performance

- How do we measure the performance of our code on a given architecture?
- Elapsed Time
 - counts everything, including the I/O time and the CPU time dedicated to other users and tasks
 - a useful number but often not good performance indicator
- CPU Time
 - Doesn't count I/O or time spent running other programs
 - Better indicator than the elapsed time

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

CPU Time

- Time spent executing the lines of code in our program
- It usually requires reading of the OS internal clock
- Low resolution clock
 - You need to repeat small sections of the code many times in order to get meaningful results
 - Inefficient in many applications

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Elapsed time

- Accurate enough for short sequences of instructions if the machine is not overused
- There are high-resolution hardware timers that can be used
- These usually count machine cycles

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Clock Cycles

- Instead reporting execution time in seconds we often use cycles
- Clock “ticks” indicate when to start activities
- Parameters:
 - Cycle time = time between ticks
 - Clock rate (frequency) = cycles/second (1Hz = 1 cycle/sec)
- A 1GHz clock has a $1/10^9 = 10^{-9} = 1\text{ns}$ cycle time
- We can calculate the elapsed time easily if we know the number of elapsed cycles and the clock frequency.

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Performance metrics

- Performance is always determined by execution time
- Performance is proportional to the reciprocal value of the execution time
- Several performance metrics are often used
 - Average # of instruction per second (MIPS= 10^6 instructions/sec)
 - Average # of floating point instructions per second (MFLOPS = 10^6 floating point instructions/sec)
 - For numerical applications

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Timing of instructions

- Very difficult task on modern microprocessors
 - The architecture is very complex,
 - out-of-order execution, memory hierarchy, instruction-level parallelism, etc.
 - It used to be quite simple some time ago
 - Tables with instruction execution times
- Timings of the small code segments is often very inaccurate due to
 - Timing overhead
 - Low clock resolution

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Timing tools

- The importance of the accurate timings has forced the architecture developers to include a high-resolution hardware “clock”
- Starting with Pentium Intel 32-bit architecture includes a hardware cycle counter accessible through `rdtsc` instruction
- **rdtsc**
 - Loads the current value of the processor's time-stamp counter into the EDX:EAX registers
 - TSC counts the number of cycles since the CPU was powered up or reset. It uses 64-bit MSR.

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Example

- Here is one example that uses only the low dword of the cycle counter
 - Simpler to use, less overhead
 - The probability that the high word has changed is very small (every 4 billion cycles)
 - The probability that the whole 64-bit counter has overflowed is negligible.

```
rdtsc
mov     ebx, eax      ; save starting count
;; Instrucitons to be timed here
rdtsc
sub     eax, ebx      ; eax contains elapsed cycles
call   PutDDec       ; Output the number of cycles
```

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Using rdtsc

```
.DATA                                .DATA
Msg  DB    'Elapsed cycles = $'      Msg  DB    'Elapsed cycles = $'
.CODE
EXTERN  PutDDec : NEAR              EXTERN  PutDDec : NEAR
Tim  PROC
  _Begin
  rdtsc
  mov  ebx, eax;
  rdtsc
  sub  eax, ebx;
  _PutStr Msg
  call PutDDec ;
  _Exit 0;
Tim  PROC
  _Begin
  rdtsc
  mov  ebx, eax;
  rdtsc
  sub  eax, ebx;
  ;; _PutStr Msg
  call PutDDec ;
  _Exit 0;
```

- The output of the code on the left is 2340 and on the right 32 cycles.
- How is that possible?

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Word of caution

- Issues affecting the cycle count
 - Out-of-order execution
 - The RDTSC instruction is not a serializing instruction.
 - It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed.
 - Solution: include serializing instruction in the timing loop
 - Cache effects
 - Using rdtsc on the same section of code often produces very different results due to cache effects
 - Solution: cache “warming”, averaging, etc.
 - Register overwriting
 - Counter overflow

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003

Homework

- Read Ch. 3 and Ch. 4 in Jones
- Problems:
 - Exercises 4.1 (pp. 65) 1-4.
 - 4.2 (pp. 71) 1 and 2.
- Optional:
 - Read Ch. 2 in Patterson & Hennessy (more details about the performance)

D. Mirkovic, COSC 2410: Computer Organization and Programming, Fall 2003