CS271 Computer Architecture and Assembly Language
MASM 8.0 Language Guide

Defining constants

Ordinarily, constants are defined before the **`.data`** segment. The general form is:

```
name = value
```

Example:

```
MaxSize = 100        ;array size
```

Declaring variables

Variables are declared in the **`.data`** segment, one declaration per line. The general form is:

```
label     data_type   initializer(s)    ;comment
```

- *label* is the "variable name". Use meaningful names, that conform to the usual rules
- *data_type* is one of the types shown in the table below
- At least one *initializer* is required. If there is just one value for the initializer, that value will be assigned to the variable. If the "value" is ?, the variable is uninitialized. If the initializer is a positive integer *x* followed by DUP(*xxx*) … where *xxx* is a valid initializer value … an *x*-element array is defined with all elements assigned to *xxx*. Strings may also be initialed here (see below).
- *comment* should explain how the variable will be used

Type	Used for:
BYTE	1-byte unsigned integers [0 … 255], ASCII characters
SBYTE	1-byte signed integers [-128 … 127]
WORD	2-byte unsigned integers [0 … 65535], address
SWORD	2-byte signed integers [-32768 … 32767]
DWORD	4-byte unsigned integers [0 … 4294967295], address
SDWORD	4-byte signed integers [-2147483648 … 2147483647]
FWORD	6-byte integer
QWORD	8-byte integer
TBYTE	10-byte integer
REAL4	4-byte floating-point
REAL8	8-byte floating-point
REAL10	10-byte floating-point

Examples:

```
size      DWORD     100            ;class size
days      REAL4     31 DUP (0.0)   ;31-element array for
                                   ; daily costs.
response  BYTE      'Y'            ;positive answer
```

Declaring strings

Strings are declared as **BYTE**, and must end with an extra 0. At first they look different from other variables, but you will see that they are pretty much like everything else.

Examples:

```
warning   BYTE "Keep your seatbelt fastened ...", 0
                              ;part of the introduction
prompt    BYTE "Enter an integer : ", 0
                              ;to get data from the user
```

Registers

MASM refers to registers by name. Technically, most of the registers can be set to anything, but some of them are usually used for specific purposes.

eax, *ebx*, *ecx*, *edx*:	general purpose, but some instructions give you no choice about which registers to use. E.G., *ecx* is the counter for the *loop* instruction; *eax* holds the dividend and quotient for the *div* instruction, and *edx* hold the remainder.
ax, *ah*, *al*, etc.	"partial" registers (analogous names for *ebx*, *ecx*, *edx*), *ax* refers to the low 16 bits of *eax*, *al* refers to the low 8 bits of *ax*, and *ah* refers to the high 8 bits of *ax*.
esi	source index, often used to hold an address or an offset
edi	destination index, often used to hold an address or an offset
ebp	base pointer, often used to hold the address of the first element of an array
esp	stack pointer, maintained by the system when a *push* or *pop* instruction is executed
si, di, bp, sp	low 16 bits of *esi*, *edi*, *ebp*, and *esp* respectively

Executable statements

Instructions are placed inside procedures, which are defined in the **.code** segment. The general form of a procedure is

```
procname  PROC
; procedure description
; preconditions: (registers required)
; postconditions:(registers changed)

<executable statements>

procname  ENDP   initializer(s)    ;comment
```

The general form of an instruction is

```
label:    opcode    operand,[operand]   ;comment
```

- *label* is used to mark a target. Use meaningful names that conform to the usual rules. Suggestion: put each label appear on its own line.
- *opcode* is a MASM instruction (see Appendix B in the textbook)
- *operand*s are registers, constants, literals, variables, labels, etc. Depending on the opcode, one or more operands may be required
- *comment* should explain the purpose of the instruction. Suggestion: write an explanatory comment for each logical group of instructions. Instructions should be grouped so that they require no more than a one-sentence comment to explain their purpose.
- The fields of an instruction must appear in the order shown. Opcode and operands must be on the same line. Any field of an instruction may be empty (unless operands are required by the opcode).

The **`call`** instruction:

- used for calling a procedure … internal or library. Be sure that you understand the pre and post conditions for a call. Parameters can be passed in a variety of ways, including using the system stack. A procedure might use values in certain registers (pre-conditions) and/or change registers (postconditions).
- Example:

```
mov  ecx,IntegerCount
call ArraySum
```

The **`mov`** instruction:

- used for assigning a value to a register or variable. The first operand is the destination (register or memory). The second is the source or value to be assigned (register, memory, constant, or literal).
- Examples:

```
mov  ecx,integerCount
mov  response,'N'
mov  ebx,eax
```

Arithmetic instructions:

- used for adding, subtracting, etc., a value to a register or variable. The first operand is both an argument and the destination (register or memory). The second is the source or value to add, subtract, etc. (register, memory, constant, or literal).
- Examples:

```
add  ecx,25
sub  value,edx
sub  ebx,eax
```

- The *mul* and *div* instructions are used for unsigned operations, and use some implied operand registers. (See Appendix B in the textbook.) Before using either of these instructions, it is a good idea to set the *edx* register to zero (unless you intend to divide a QWORD). The *mul* operation multiplies its operand times *eax*, and puts the result in *eax* with overflow in *edx*. The *div* operation divides the *edx*:*eax* combination (high-order bits in *edx*, low-order bits in *eax*) by the operand, and puts the integer quotient in *eax* and the remainder in *edx*. Note that the operand must be a register or a variable (not literal or constant). Be sure that these implied registers are not being used for something else when a *mul* or *div* instruction is executed.
- Examples:

```
mov  eax,value
mul  ebx       ;result is in edx:eax
```

- or

```
mov  eax,value
xor  edx,edx   ;set edx to zero
div  ebx       ;quotient is in eax, remainder is in edx
```

- The *imul* and *idiv* instructions work in the same way as *mul* and *div*, but are used for signed values. However, instead of setting *edx* to zero for *idiv*, it is necessary to extend the sign of the value in *eax* into *edx*.
- Examples:

```
mov  eax,value
imul  ebx      ;result is in edx:eax
```

  - or

```
mov  eax,value
cdq            ;extend sign of eax into edx
idiv  ebx      ;quotient is in eax, remainder is in edx
```

<u>Comparison and branching instructions:</u>

- used for implementing decision and repetition control. There are several forms, but a few examples will probably suffice. Things might seem a bit primitive here …
- Example: Translate this decision structure into MASM.

  if (x > 10) do one thing, else do another thing

```
decide:
       mov  ebx,x
       cmp  ebx,10
       jle  lessEqual
greater:
       call doSomething
       jmp  endDecide
lessEqual:
       call doAnotherThing
endDecide:
```

  Note that labels *decide* and *greater* are not required; they just help to clarify the structure. Also note that the test is for the opposite condition in order to skip the "true" block. Don't forget to skip the "else" block if the "true" block is executed.
-
- Example: Translate this pre-test loop into MASM.

  while (user enters integers greater than 0) do something

```
initialize:
       call ReadInt
pretest:
       cmp  eax,0
       jle  endloop
       call doSomething
       call ReadInt
       jmp  pretest
endloop:
```

  ReadInt is from the *Irvine32* library, and puts its value into *eax*. Also note that the loop control condition must be set before the repeated code, and must be reset inside the loop before jumping back to the top of the loop.

- Example: Translate this counted loop into MASM.

  for (k = 1 to 10) do something

```
initialize:
        move ecx,10
forLoop:
        call doSomething
        loop forLoop
endFor:
```

  The *ecx* register is automatically decremented and tested by the *loop* instruction. The loop terminates when ecx becomes 0. To make this a true for loop, another test is required before entering the loop. Be sure that the loop body doesn't mess up *ecx*.

Input/output:

- Beginners should use the I/O procedures defined in the *Irvine32* library.
- Examples:

```
call ReadInt
mov value,eax       ;get value from the user

mov  eax,value
call WriteInt       ;display value

mov  edx,offset string1       ; address of string1
mov  ecx, SIZEOF string1      ; max number of characters
dec  ecx                      ; leave space for zero-byte
call ReadString               ; input the string
mov  strSize,eax              ; save the length
call Crlf                     ; new line

mov  edx,offset prompt        ; display a prompt
call WriteString
```

  Note the use of specific registers. String procedures use references (*offset*); this is called register indirect addressing. See examples in `Masm615\Examples`.

Addressing modes:

MASM has several ways to access data:

- **Immediate** Use constant as operand

  Examples:
  ```
  mov  eax,10
  add  eax,20h
  ```

- **Direct** Set register to address of global

  Example:
  ```
  mov  esi,OFFSET var1
  ```
- Register Use register as operand

  Examples:
  ```
  mov  var1,eax
  add  edx,eax
  ```

- **Register indirect** Access memory through address in a register

  Examples:
```
mov   [esi],25
add   [eax],ebx
mov   eax,[edi]
```
  Brackets [ ] mean "memory referenced by the address in". Note that the following instruction is **<u>invalid</u>** because it attempts to add memory to memory:
```
add   [eax],[edi]
```

- **Indexed** "array" element, using offset in register

  Examples:
```
mov   edi,0
mov   array[edi],eax
add   edi,4
mov   array[edi],ebx
```
  This means "add the value in [ ] to address of global".

- **Base-indexed** Start address in one register; offset in another; add the registers to access memory

  Examples:
```
mov   edi,OFFSET array
mov   ecx,12
mov   eax,[edi+ecx]
mov   edx,4
mov   ebx,[edi+edx]
mov   [edi+ecx],ebx
mov   [edi+edx],eax
```

- **Stack** Memory area specified and maintained as a stack; stack pointer in register *esp*

  Examples:
```
push  eax
add   eax,ebx
mov   var1,eax
pop   eax
```
  Note: the *esp* "stack pointer" is maintained automatically by the *push* and *pop* instructions.