CS 271
Computer Architecture & Assembly Language

Lecture 3
MASM Syntax and First MASM Program
1/11/22, Tuesday
Odds and Ends

• My office hours are confirmed:
  • M 11-12, W 12-1 @ KEC 3114, R 3:30-5:30 @ KEC 3057

• From now on, you will have access to lecture recordings from W21 on Canvas
  • I will make a class announcement once I’ve done so

• Assignment 1 clarifications

• Questions?
Lecture Topics:

• Introduction to MASM assembly language
• Writing a MASM program
Introduction to MASM assembly language
TITLE Program Template (template.asm)

; Author:

; Course/project ID

; Description:

 Date:

INCLUDE Irvine32.inc

<insert constant definitions here>

.data

<insert variable definitions here>

.code

main PROC

<insert executable instructions here>

exit ; exit to operating system

main ENDP

<insert additional procedures here>

END main
MASM syntax and style

• MASM is **not** case-sensitive!!
  • **Constants** usually ALL CAPS
• Segments start with `.`
  • `main` should be the first procedure in the `.code` segment
  • Beginning of next segment (or `END main`) is end of segment
• Comments start with `;`
  • Can start anywhere in a line
  • Remainder of line is ignored by the assembler
  • End of line is end of comment
• Use indentation and sufficient white space to make sections easy to find and identify
MASTM identifier syntax

• Identifiers: Names for variables, constants, procedures, and labels
• 1 to 247 characters (no spaces)
  • Use concise, meaningful names
• Not case sensitive!
• Start with letter, _, @, or $
  • For now, start with letter only
• Remaining characters are letters, digits, or _
• Cannot be a reserved word
  • E.g.: proc, main, eax, ... etc.
Memory Locations

• May be named
  • Name can refer to a variable name or a program label

• Interpretation of contents depends on program instructions
  • Numeric data
    • Integer, floating point
  • Non-numeric data
    • Character, string
  • Instruction
  • Address
  • etc.
# MASM data types syntax

<table>
<thead>
<tr>
<th>Type</th>
<th>Used for:</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>Character, string, 1-byte integer</td>
</tr>
<tr>
<td>WORD</td>
<td>2-byte integer, address</td>
</tr>
<tr>
<td>DWORD</td>
<td>4-byte unsigned integer, address</td>
</tr>
<tr>
<td>FWORD</td>
<td>6-byte integer</td>
</tr>
<tr>
<td>QWORD</td>
<td>8-byte integer</td>
</tr>
<tr>
<td>TBYTE</td>
<td>10-byte integer</td>
</tr>
<tr>
<td>REAL4</td>
<td>4-byte floating-point</td>
</tr>
<tr>
<td>REAL8</td>
<td>8-byte floating-point</td>
</tr>
<tr>
<td>REAL10</td>
<td>10-byte floating-point</td>
</tr>
</tbody>
</table>
MASM Data definition syntax

• In the .data segment

• General form is

  `label  data_type  initializer ;comment`

  • `label` is the “variable name”
  • `data_type` is one of (see previous slides)
  • At least one `initializer` is required
    • May be `?` (value to be assigned later)

• Examples:

  `size     DWORD    100 ;class size`
  `celsius  WORD     -10 ;current Celsius temp`
  `response BYTE    'Y' ;positive answer`
  `myName   BYTE     "Wile E. Coyote", 0`
  `gpa      REAL4    ? ;my GPA`
Data in Memory

• “variables” are laid out in memory in the order declared

• Example:

```assembly
.data
size DWORD 100 ; class size
celsius WORD -10 ; current Celsius
response BYTE 'Y' ; positive answer
myName BYTE "Wile E. Coyote", 0
  ; my GPA
  ; my GPA

  # space

• Suppose that the data segment starts at memory address 1000

size is address 1000 (DWORD uses 4 bytes)
celsius is address 1004 (WORD uses 2 bytes)
Response is address 1006 (BYTE uses 1 byte)
myName is address 1007 (Each character uses 1 byte)
  (Blank spaces and the terminating 0 are characters too!)
gpa is address 1022
```
Data in Memory

- Each name is a constant:
  - i.e. the system substitutes the memory address for each occurrence of a name
- The contents of a memory location may be variable.

\[
\begin{align*}
\text{size} & \quad \text{is address 1000} \quad \text{(DWORD uses 4 bytes)} \\
\text{celsius} & \quad \text{is address 1004} \quad \text{(WORD uses 2 bytes)} \\
\text{Response} & \quad \text{is address 1006} \quad \text{(BYTE uses 1 byte)} \\
\text{myName} & \quad \text{is address 1007} \quad \text{(Each character uses 1 byte)} \\
\text{gpa} & \quad \text{is address 1022} \\
\end{align*}
\]

(Blank spaces and the terminating 0 are characters too!)
**Literals**

- Actual values, named constants
  - Integer
  - Floating point
  - Character
  - String (only in `.data` segment or named constant)

- Used for:
  - Initializing variables (in the `.data` segment)
  - Defining constants
  - Assigning contents of registers
  - Assigning contents of memory (in the `.code` segment)
MASM Literals syntax

• Integer
  • Optional radix: b, q/o, d, h
    • Digits must be consistent with radix (e.g., 1011b, 235q, 2012d, 30h)
    • Hex values that start with a letter must have a leading 0 (e.g., 0A3h)
      • Or use the 0x prefix instead of the radix (e.g., 0xA3)
  ✓ Default is decimal

• Floating-point (decimal real)
  • Optional sign
  • Standard notation (e.g., -3.5 +5. 7.2345)
  • Exponent notation (e.g., -3.5E2 6.15E-3)
  • Must have a decimal point
MASM Literals syntax

• Character
  • Single character in quotes
    • ‘a’ 97  “*” $2  ‘3’ $1
    • Single quotes recommended

• String
  • 2 or more characters in quotes
    • “always”, 0
    • ‘123 * 456’, 0
    • Double quotes recommended
    • Embedded quotes must be different
      • “It’s”, 0  ‘Title: “MASM”’, 0

• String must be null-terminated
  • Always end with zero-byte
MASM Instruction syntax

• Each instruction line has 4 fields:
  • Label
  • Opcode
  • Operands
  • Comments

• Depending on the opcode, one or more operands may be required
  • Otherwise, any field may be empty
  • If empty opcode field, operand field must be empty
MASM Instruction syntax

- **Opcode** (specifies what to do)
  - Mnemonic (e.g., ADD, MOV, CALL, etc.)

- **Zero, one, or two Operands** (specify the opcode’s target)
  - Different number of operands for different opcodes

```
opcode
destination
opcode destination, source
```

```
int x = 30;
ADD eax, ebx
```
Specific “addressing modes” are permitted for the operands associated with each opcode.

• Basic (used in first programming assignment)
  • Immediate
  • Register
  • Direct
  • Offset

• Advanced (used in later assignments)
  • Register indirect
  • Indexed
  • Base-indexed
  • Stack

See the MASM list of instructions
Writing a MASM program

- Rules & Regulations
- Syntax and semantics
MASM Instructions

• For now, know how to use
  • mov, add, sub, mul, div, call

• Some instructions use implied operands

• See textbook (Appendix) or on-line instructions
Easy Instructions

• For 2-operand instructions, the 1st operand is the destination, and the 2nd operand is the source.
• 2-operand instructions require at least one of the operands to be a register (or op2 must be literal).
  • Note: **op1 cannot be a literal**

```
mov   op1, op2   ; op2 is copied to op1
add   op1, op2   ; op2 is added to op1
sub   op1, op2   ; op2 is subtracted from op1
inc   op1        ; add 1 to op1
dec   op1        ; subtract 1 from op1
```
Instructions with implied operands

- **mul** implied operand must be in **EAX**
- **mul** op2 ; result is in **EDX:EAX**
- Example:

```
mov eax, 10
mov ebx, 12
mul ebx ; result is in eax (120)
; with possible overflow in edx
; edx is changed!
```
Instructions with implied operands

- `div` implied operand is in **EDX:EAX**
- So **extend EAX into EDX** before division
- `div op2`; quotient is in **EAX**
  - remainder is in **EDX**

Example:

```
mov eax, 100
cdq ; extend the sign into edx
mov ebx, 9
div ebx ; quotient is in eax (11)
          ; remainder is in edx (1)
```
## Operand notation (See Instruction list)

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r8</td>
<td>8-bit general-purpose register: AL, AH, BL, BH, CL, CH, DL, DH</td>
</tr>
<tr>
<td>r16</td>
<td>16-bit general-purpose or multi-purpose register: AX, BX, CX, DX, SI, DI, BP, SP</td>
</tr>
<tr>
<td>r32</td>
<td>32-bit general-purpose or multi-purpose register: EAX, EBX, ECX, EDX, ESI,EDI, EBP, ESP</td>
</tr>
<tr>
<td>reg</td>
<td>any general-purpose or multi-purpose register</td>
</tr>
<tr>
<td>accum</td>
<td>AL, AX, or EAX (depending on operand size)</td>
</tr>
<tr>
<td>mem</td>
<td>8-bit, 16-bit, or 32-bit memory location (depending on operand size)</td>
</tr>
<tr>
<td>segreg</td>
<td>16-bit segment register: SS, CS, DS, ES, FS, GS</td>
</tr>
<tr>
<td>r/m8</td>
<td>8-bit register or memory location</td>
</tr>
<tr>
<td>r/m16</td>
<td>16-bit register or memory location</td>
</tr>
<tr>
<td>r/m32</td>
<td>32-bit register or memory location</td>
</tr>
<tr>
<td>imm8</td>
<td>8-bit literal value</td>
</tr>
<tr>
<td>imm16</td>
<td>16-bit literal value</td>
</tr>
<tr>
<td>imm32</td>
<td>32-bit literal value</td>
</tr>
<tr>
<td>imm</td>
<td>8-bit, 16-bit, or 32-bit literal value (depending on operand size)</td>
</tr>
</tbody>
</table>
## Examples

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV mem,accum</td>
<td>mov total,eax</td>
</tr>
<tr>
<td></td>
<td>mov response,al</td>
</tr>
<tr>
<td>MOV accum,mem</td>
<td>mov al,char</td>
</tr>
<tr>
<td></td>
<td>mov eax,size</td>
</tr>
</tbody>
</table>

Notes:

- **accum** means “eax or some valid part of eax”
- **imm** means “a literal or constant”

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV mem,imm</td>
<td>mov color,7</td>
</tr>
<tr>
<td></td>
<td>mov response,’y’</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV reg,imm</td>
<td>mov ecx,256</td>
</tr>
<tr>
<td></td>
<td>mov edx,OFFSET myString</td>
</tr>
</tbody>
</table>
### Examples

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV reg,reg</td>
<td>mov dh,bh</td>
</tr>
<tr>
<td></td>
<td>mov edx,ecx</td>
</tr>
<tr>
<td></td>
<td>mov ebp,esp</td>
</tr>
<tr>
<td>MOV mem,reg</td>
<td>mov count,ecx</td>
</tr>
<tr>
<td></td>
<td>mov num1,bx</td>
</tr>
<tr>
<td>MOV reg,mem</td>
<td>mov ebx,pointer</td>
</tr>
<tr>
<td></td>
<td>mov al,response</td>
</tr>
</tbody>
</table>

**Notes:**

- **mem8** means “BYTE”
- **mem16** means “WORD”
- **mem32** means “DWORD”
- **sreg** means CS, DS, ES, FS, GS, or SS

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV sreg,reg16</td>
<td>mov ds,ax</td>
</tr>
<tr>
<td>MOV sreg,mem16</td>
<td>mov es,pos1</td>
</tr>
<tr>
<td>MOV reg16,sreg</td>
<td>mov ax,ds</td>
</tr>
<tr>
<td>MOV mem16,sreg</td>
<td>mov stack_save,ss</td>
</tr>
</tbody>
</table>
Invalid MOV statements

.data
bVal BYTE 100
bVal2 BYTE  ?
wVal WORD 2
dVal DWORD 5

.code
mov ds,45 immediate move to DS not permitted
mov esi,wVal size mismatch
mov eip,dVal EIP cannot be the destination
mov 25,bVal immediate value cannot be destination
mov bVal2,bVal memory-to-memory move not permitted

25 = bVal
Libraries

• We will use Irvine’s library (for now) to handle the really awful stuff
  • Input/output
  • Screen control
  • Timing
  • etc.

• Check IrvineLibHelp, or find the descriptions in your textbook.
Library Procedures – Overview 1

• **Clrscr** – clear the screen
  • **Preconditions**: none
  • **Postconditions**: screen cleared, and cursor is at upper left corner

• **Crlf** – New line
  • **Preconditions**: none
  • **Postconditions**: cursor is at beginning of next new line

$\frac{5}{3} \quad 10$
• **ReadInt** – Reads an integer from keyboard, terminated by the Enter key
  • **Preconditions**: none
  • **Postconditions**: value entered is in EAX

• **ReadString** – Reads a string from keyboard, terminated by the Enter key
  • **Preconditions**: OFFSET of memory destination in EDX
    Size of memory destination in ECX
  • **Postconditions**: String entered is in memory
    Length of string entered is in EAX
• **WriteInt, WriteDec** – Writes an integer to the screen
  • **Preconditions**: value in EAX
  • **Postconditions**: value displayed
  • WriteInt displays +/-  

• **WriteString** – Writes a null-terminated string to the screen
  • **Preconditions**: OFFSET of memory location in EDX
  • **Postconditions**: String displayed
Calling a Library Procedure

• The INCLUDE directive copies the procedure prototypes (declarations) into the program source code.

• Call a library procedure using the CALL instruction.
In-line Comments

- Start with `;`
- May be on separate line or at the end of a line
- Use comments to clarify lines or sections
- Preferred ...

  ```
  ; Calculate the number of students on-line today.
  mov    eax, size
  sub    eax, absent
  mov    present, eax
  ```

- OK ...

  ```
  mov    eax, size ; start with class size
  sub    eax, absent ; subtract absentees
  mov    present, eax ; number present
  ```

- Terrible ...

  ```
  mov    eax, size ; move size into eax
  sub    eax, absent ; subtract absent from eax
  mov    present, eax ; move eax to present
  ```
Example Problem Definition

Write a MASM program to perform the following tasks:

1. Introduce yourself to the user.
2. Get the user’s name and number of yards.
3. Greet the user, and report the yards in inches.
4. Say good-bye to the user.

Requirements:

1. The user’s name and yards must be entered by the user, and must be stored and accessed as data segment variables.
2. The “yard-to-inch factor” (36) must be defined as a constant.
Program Design

• Decide what the program should do
• Define algorithm(s)
• Decide what the output should show
• Determine what variables/constants are required
Implementing a MASM program

• Open project
• Start with template, “save as” <.asm file in the program directory>
  • This is the source code file
• Fill in identification block information
• Create comment outline for algorithms
• Define constants
  • Test/fix (syntax check, nothing happens)
• Declare variables (.data section)
  • Test/fix (syntax check, nothing happens)
• Enter the output code
  • Test/fix (no calculations, usually everything show 0)
• Enter the input code
  • Test/fix (no calculations, echo input)
• Enter the calculation code
  • Test/fix (logic check, verify)

*First try Debug, Start Without Debugging (more later on using the debug system)
Writing a MASM program

• Demo