Lecture 7: Assembly Language Programs

- Basic elements of assembly language
- Assembler directives
- Data allocation directives
- Data movement instructions
- Assembling, linking, and debugging
- Using TASM

Constants and Expressions

- Numeric literal:
- Integer constants end with a radix symbol – h (hex), q or o (octal), d (decimal), b (binary).

Constants and Expressions (cont.)

- constant expression –
- symbolic constant –
- Character or string constants –

Statements

[name][mnemonic][operands][;comment]

- Free-form – any column, any number of spaces, can use blank lines
- Two types of statements:
  - Instruction – statements executed by the processor at run-time.
  - Directives – statements that give instructions to the assembler.

Names

- Four types of names:
  - Variable:
  - Label:
  - Symbol:
  - Keyword:
- Case-insensitive
- Can’t start with digits (and should avoid starting with @)
- Can’t match a reserved word

Program Structure Using Directives

```
title <your title here>
.model small
.stack 100h
.data
<your data here>
.code
main proc
     mov  ax, @data
     mov  ds,ax
    <your code here>
main endp
end main
```

Assembly Directives

- .code – marks start of code segment
- .data – marks start of data segment
- .stack – set the size of the stack segment
- .model – specify memory model (we will use .model small – 64K for memory, 64K for data)
- title – title of listing file
- proc – begin procedure
- endp – end of procedure
- end – end of program
- page – set page format

Alternative Structure

```
title <your title here>
.model small
.stack 100h
.data
<your data here>
.code
.startup
    <your code here>
.exit
end
```

.startup and .exit are always used in a pair. .startup sets up the data segment so you don’t need to do it!

Data Allocation Directives

- Data allocation directives allocate storage based on several predefined types:
  - DB – define byte (1 byte)
  - DW – define word (2 bytes)
  - DD – define doubleword (4 bytes)
  - … and more for larger data types (up to 10 bytes)

DB Example

```
.data
aList  db “ABCD”
```

offset	contents
0000	‘A’
0001	‘B’
0002	‘C’
0003	‘D’

Define Byte (DB)

- Example:

```
char1  db ‘A’        ;ASCII character
char2  db ‘A’ – 10 ; expression
signed1 db  –128  ;smallest signed val
signed2 db +127   ;largest signed val
unsig1  db 255      ;largest unsigned val
```

- Multiple initializers:

```
list db 1, 2, 3, 4
```

- Strings:

```
myString  db “Hello World”,0
```

- Can duplicate values using DUP:

```
db 2 dup(“ABC”)   ; 6 bytes “ABCABC”
```

Define Word (DW)

- Example:

```
dw 0, 65535  ;smallest/largest unsigned vals
dw –32768, 32767 ;smallest/largest signed
dw  256 * 2  ;calc expression (512)
dw 1000h, 4094, ‘AB’ ; multiple initializers
dw ?                          ;uninitialized
dw 5 dup(1000h)        ; 5 words, each 1000h
dw 5 dup(?)  ;5 words, uninitialized
```

- Pointer – the offset of a variable or subroutine can be stored in another variable (a pointer):

```
list  dw  23, 45, 22, 34
ptr    dw   list
```

- Reversed storage format

DW Example

- code:

```
.data
msg dw 'SC','02','11','L ',
  'ba','1 '
```

- data in memory:

```
-d 131d:0

131D:0000  43 53 32 30 31 31
  20 4C-61 62 20 31 2B D3 E8 DB
  CS2011 Lab 1+...
```

Define Doubleword (DD)

- Examples:

```
signed_val dd  0, 0BCDA1234h, -2147483648
block  dd 100h dup(?) ; 256  doublewords (1024
   bytes)
```

- Bytes in a doubleword are stored in reverse order – least significant digits at the lowest offset.

  12345678h:

  Offset: 00 01 02 03

  Value: 78 56 34 12

- Doublewords can hold the 32-bit segment-offset address of a variable or procedure:

```
pointer dd subroutine1
```

MOV Instruction

- MOV destination, source
- Basic forms:
  - MOV reg, reg
  - MOV mem, reg
  - MOV reg, mem
  - MOV mem, immed
  - MOV reg, immed
- Notice: no move from memory to memory!

MOV Examples

```
.data
count    db   10
total    dw   4126h
bigVal   dd   12345678h

.code
   mov  al, bl      ;8-bit register to register
   mov  bl, count ; 8-bit memory to register
   mov  count,26  ; 8-bit immediate to memory
   mov  bl, 1      ;8-bit immediate to register
   mov  dx, cx    ; 16-bit register to register
   mov  bx, 8FE2h ;16-bit immediate to register
   mov  total, 1000h ;16-bit immediate to memory
   mov  eax, ebx   ; 32-bit register to register
   mov  edx, bigVal  ;32-bit memory to register
```

Type Checking

- Assembler uses the size *you* give the variable:

```
.data
count dw 20h
.code
mov al, count  ; error
```

- The LABEL attribute:

```
.data
countb label byte ; byte attribute
countw dw 20h     ; word attribute
.code
mov al, countB   ; get low byte
mov cx, countW   ; get whole thing
```

XCHG

- XCHG exchanges the contents of two registers or a register and a variable:

XCHG reg, reg
XCHG reg, mem
XCHG mem, reg

Operands with Displacements

- You can add a displacement to the name of a variable using direct-offset addressing.
- Example:

```
arrayB   db  10h, 20h
arrayW   dw 100h, 200h
….
mov al, arrayB    ;AL = 10h
mov al, arrayB+1  ;AL = 20h
mov ax, arrayW    ;AX = 100h
mov ax, arrayW+2  ;AX = 200h
mov ax, arrayW+1  ;AX = ?
```

XCHG Example

- adding two variables from p. 73 of Irvine.

Assembling, Linking, and Debugging

- Multi-step process:
  - Use a text editor to create a *source file.*
  - Use the assembler program to read the source file and create an *object file.*
  - Use the linker to link the object file with any needed routines from the link library and create an *executable program.*
  - Use the operating system to run the executable.

Assembling using TASM

Without errors:

```
D:\Janet\Teaching\CS2011\Labs>tasm/z/zi lab1.asm

Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996
Borland International

Assembling file:   lab1.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  418k
```

- z option – source lines with errors should be displayed.
- zi option – include information needed by the debugger in the output file.

Assemble-Link-Execute Cycle

- figure 1 from p. 60 of Irvine
- Table 2 from p. 61 of Irvine

Assembling using TASM, cont.

With errors:

```
D:\Janet\Teaching\CS2011\Labs>tasm/z/zi lab1.asm

Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996
Borland  International

Assembling file:   lab1.asm
      mov     AX, [mst]
**Error** lab1.asm(11) Undefined symbol: MST
Error messages:    1
Warning messages:  None
Passes:            1
Remaining memory:  418k
```

Linking using TASM

```
D:\Janet\Teaching\CS2011\Labs>tlink/v lab1

Turbo Link  Version 7.1.30.1. Copyright
(c) 1987, 1996 Borland International
```

- /v option indicates that debug options should be included.
- Other options:
  - /3 – allow 32 bit registers (we won’t be using this)
  - /m – generate a map file

Debugging using TASM

- Use debugging options when assembling and linking
- Debugging information is stored in the .obj and .exe files, making them slightly larger.
- Starting the turbo debugger:
  - `td lab1`

Tracing Programs

- Some useful windows for looking at program information:
  - Stack Window (View/Stack) – lists all active procedures with most recent called listed first. This tells you how you got to where you are.
  - Execution History Window – this keeps a record of the last 400-3000 instructions executed!

Tracing Programs, cont.

- Stepping through your program:
  - run (F9) – runs through the program to the end, a breakpoint, or until ctrl-break is pressed
  - go to cursor (F4) – runs and stops before the line the cursor is on is executed
  - trace into (F7) – a single-step through the program that steps into subroutines
  - step over (F8) – a single-step through the program that skips over procedure calls (executing the procedure). This fully executes LOOP and INT instructions
  - see p. 616 of Irvine for more!

Breakpoints

- Breakpoints are very useful!
- A breakpoint is a marker that tells the debugger to pause in one of the following ways:
  - unconditionally on a particular statement
  - when a pre-set condition becomes true (say on the 3rd time through a loop)
  - when a memory location changes
- see Irvine p. 617 for things you can do with breakpoints.

Examining and Modifying Data

- There are many ways to examine and modify data using the debugger:
  - examining registers (View/Registers)
  - examine and modify variables (View/Variables)
  - watch windows (View/Watches) allow you to watch variables change as the program runs
  - view memory (View/Dump) – getting a hex memory dump (like we did in debug)
- Be sure to read Appendix D on how to use the Turbo Debugger!