Assembly basics

CS 2XA3

Term I, 2018/19

Outline

What is Assembly Language?

In a high level language (HLL), one line of code usually translates to 2, 3 or more machine instructions. Some statements may translate to hundreds or thousands of machine instructions.

- In Assembly Language (AL), one line of code translates to one machine instruction; AL is a "human readable" form of machine language
- HLLs are designed to be "machine-independent", but machine dependencies are almost impossible to eliminate.
- ALs are NOT machine-independent. Each different machine (processor) has a different machine language. Any particular machine can have more than one assembly language

Assemblers

An assembler is a program that translates an assembly language program into binary code

- NASM Netwide Assembler
- TASM Turbo Assembler (Boorland)
- MASM Microsoft Assembler
- GAS GNU assembler

NASM

- We will use NASM (Netwide Assembler) in this course
- NASM is operating system independent
  - One of the two widely used Linux assemblers
  - The other is GAS (GNU assembler)

NASM

- We will not cover NASM syntax in full depth
  - We are interested in a basic machine interface, NOT a production assembler language
  - NASM has many syntactic constructs similar to C
  - NASM has an extensive preprocessor similar to the C preprocessor.

Base elements of NASM Assembler

- Character Set

  Letters `a..z A..Z`
  Digits `0..9`
  Special characters `? _ @ $ . ~`

- NASM (unlike most assemblers) is case-sensitive with respect to labels and variables

- It is not case-sensitive with respect to keywords, mnemonics, register names, directives, etc.

Literals

Literals are values that are known or calculated at assembly time. Examples:

```
'This is a string constant'
"So is this"
`Backquoted strings can use escape chars\n`
123
1.2
0FAAh
$1A01
0x1A01
```

Integers

- numeric digits (including `A..F`) with no decimal point
- may include radix specifier at end:
  - `b` or `y` binary
  - `d` decimal
  - `h` hexadecimal
  - `q` octal
- Examples
  - `200` decimal (default)
  - `200d` decimal
  - `200h` hex
  - `200q` octal
  - `10110111b` binary

Statemenmts

Syntax:

`[label[:]] [mnemonic] [operands] [;comment]`

- `[ ]` indicates optionality
- Note that **all** parts are optional $\rightarrow$ blank lines are legal
- `[label]` can also be `[name]`

  Variable names are used in data definitions

  Labels are used to identify locations in code
- Statements are free form; they need not be formed into columns
- Statement must be on a single line, max 128 chars

Statemenmts

- Example:

```
L100:  add eax, edx ; add subtotal to total
```

- Labels often appear on a separate line for code clarity:

```
L100:
add eax, edx ; add subtotal to total
```

Labels and Names

Names identify labels, variables, symbols, and keywords

- May contain:

  letters: `a..z A..Z`
  digits: `0..9`
  special chars: `? _ @ $ . ~`
- NASM is case-sensitive (unlike most x86 assemblers)
- First character must be a letter, `_` or `.` (which has a special meaning in NASM as a "local label" indicating it can be redefined)
- Names cannot match a reserved word (and there are many reserved words!)

Type of statements

1. Directives

```
limit EQU 100 ; defines a symbol limit
%define limit 100 ; like C #define
```

2. Data Definitions

```
msg db 'Welcome to Assembler!'
    db 0Dh, 0Ah
count dd 0
mydat dd 1,2,3,4,5
resd 100 ; reserves 400 bytes
```

3. Instructions

```
mov eax, ebx
add ecx, 10
```

Variable, labels, and constants

```
count1 db 100           ; variable called count1
count2 times 100 db (0); variable called count2 (100 bytes)
count3 EQU 100          ; constant called count3
count4 EQU $            ; const count4 = address of str1
str1 DB 'This is a string'      ; variable str1 16 bytes
slen EQU $-str1         ; const slen = 16
label1:  mov eax,0      ; label1 is the address of instruction
    ....                ; colon may be omitted

jmp label1
```

- `count1` is the address of a single byte, `count2` is the address of the first byte of 100 bytes of storage
- `count3` does not allocate storage; it is a textual EQUate (symbolic substitution; similar to C #define)
- The `$` has a special meaning: the location counter

The Location Counter

```
count1 db 100            ; variable called count1
count2 times 100 db (0); variable called count2 (100 bytes)
count3 EQU 100           ; constant called count3
count4 EQU $             ; const count4 = address of str1
str1 DB 'This is a string'      ; variable str1 16 bytes
slen EQU $-str1          ; const slen = 16
```

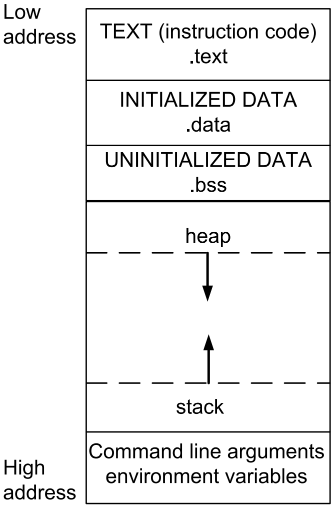
- The symbol `$` refers to the location counter
- As the assembler processes source code, it emits either code or data into the object code.
- The location counter is incremented for each byte emitted
- In the example above, `count4` is numerically the same value as `str1` (which is an address)
- With `slen EQU $-str1` the assembler performs the arithmetic to compute the length of `str1`
- Note the use `str1` in this expression as a numeric value (the address of the first byte)

Program structure

```
SECTION .data           ;data section
  msg:  db "Hello World",10  ;the string to print
                             ;10=newline
  len:  equ $-msg       ;len is value, not an addr.
SECTION .text           ;code section
  global main           ;for linker
main:                   ;standard gcc entry point
  mov edx, len          ;arg3, len of str.  to print
  mov ecx, msg          ;arg2, pointer to string
  mov ebx, 1            ;arg1, write to screen
  mov eax, 4   ;write sysout command to int 80 hex
  int 0x80     ;interrupt 80 hex, call kernel

  mov ebx, 0   ;exit code, 0=normal
  mov eax, 1   ;exit command to kernel
  int 0x80      ;interrupt 80 hex, call kernel
```

program layout



BSS came from "Block Started by Symbol", an assembler for IBM 704 in the 1950s.

Assembly program structure

```
%include "asm_io.inc"
segment .data
   ;initialized data
segment .bss
   ;uninitialized data
segment .text
  global asm_main
asm_main:
  enter 0,0     ;setup
  pusha         ;save all registers
  ;put your code here
  popa          ;restore all registers
  mov eax, 0    ;return value
leave
ret
```

I/O

- C: I/O done through the standard C library
- NASM: I/O through the standard C library
  `%include "asm_io.inc"`
- Contains routines for I/O

  `print_int` prints `EAX`
  `print_char` prints ASCII value of `AL`
  `print_string` prints the string stored at the address stored in `EAX`; must be 0 terminated
  `print_nl` prints newline
  `read_int` reads an integer into `EAX`
  `read_char` reads a character into `AL`

First program

```
;
; file:  first.asm
; First assembly program.  This program asks
for two integers as
; input and prints out their sum.
;
; To create executable:
;
; Using Linux and gcc:
; nasm -f elf first.asm
; gcc -o first first.o driver.c asm_io.o
```

```
%include "asm_io.inc" ;
; initialized data is put in the .data segment
;
segment .data
;
; These labels refer to strings used for output
;
prompt1 db "Enter a number:  ", 0 ; don't forget null
prompt2 db "Enter another number:  ", 0
outmsg1 db "You entered ", 0
outmsg2 db " and ", 0
outmsg3 db ", the sum of these is ", 0
;
; uninitialized data is put in the .bss segment
;
segment .bss
```

```
;
; These labels refer to double words used to store the
inputs;
;
input1 resd 1
input2 resd 1
;
; code is put in the .text segment
;
segment .text
     global asm_main
asm_main:
     enter 0,0 ; setup routine
     pusha
     mov eax, prompt1 ; print out prompt
     call print_string
     call read_int ; read integer
     mov [input1], eax ; store into input1
     mov eax, prompt2 ; print out prompt
```

```
    call print_string
    call read_int ; read integer
    mov [input2], eax ; store into input2
    mov eax, [input1] ; eax = dword at input1
    add eax, [input2] ; eax += dword at input2
    mov ebx, eax ; ebx = eax
    dump_regs 1 ; dump out register values
    dump_mem 2, outmsg1, 1 ; dump out memory
;
; next print out result message as series of steps
;
    mov eax, outmsg1
    call print_string ; print out first message
    mov eax, [input1]
    call print_int ; print out input1
    mov eax, outmsg2
    call print_string ; print out second message
    mov eax, [input2]
    print_int ; print out input2
```

```
mov eax, outmsg3
call print_string ; print out third message
mov eax, ebx
call print_int ; print out sum (ebx)
call print_nl ; print new-line
popa
mov eax, 0 ; return value
leave
ret
```

C driver

```c
#include "cdecl.h"
int PRE_CDECL asm_main( void ) POST_CDECL;
int main() {
    int ret_status;
    ret_status = asm_main();
    return ret_status;
}
```

- All segments and registers are initialized by the C system
- I/O is done through the C standard library
- Initialized data in .data
- Uninitialized data in .bss
- Code in .text
- stack later

Compiling

- `nasm -f elf first.asm`
  produces `first.o`
- ELF: executable and linkable format
- `gcc -c driver.c`
  produces `driver.o`
  option `-c` means compile only
- We need to compile `asm_io.asm`:
  `nasm -f elf -d ELF_TYPE asm_io.asm`
  produces `asm_io.o`
- On 64-bit machines, add the option `-m32` to generate 32-bit code, e.g. `gcc -m32 -c driver.c`

Linking

- Linker: combines machine code & data in object files and libraries together to create an executable
- `gcc -o first driver.o first.o asm_io.o`
- On 64-bit machines,
  `gcc -m32 -o first driver.o first.o asm_io.o`
- `-o outputfile` specifies the output file
- `gcc driver.o first.o asm_io.o`
  produces `a.out` by default