CMSC 313 Lecture 05

- Recap i386 Basic Architecture
- Second assembly program: "toupper.asm"
- EFLAGS Register & Branching Instructions
- Project 1

Recap i386 Basic Architecture

- Registers are storage units inside the CPU.
- Registers are much faster than memory.
- 8 General purpose registers in i386:
  - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
  - subparts of EAX, EBX, ECX and EDX have special names
- The instruction pointer (EIP) points to machine code to be executed.
- Typically, data moves from memory to registers, processed, moves from registers back to memory.
- Different addressing modes used.

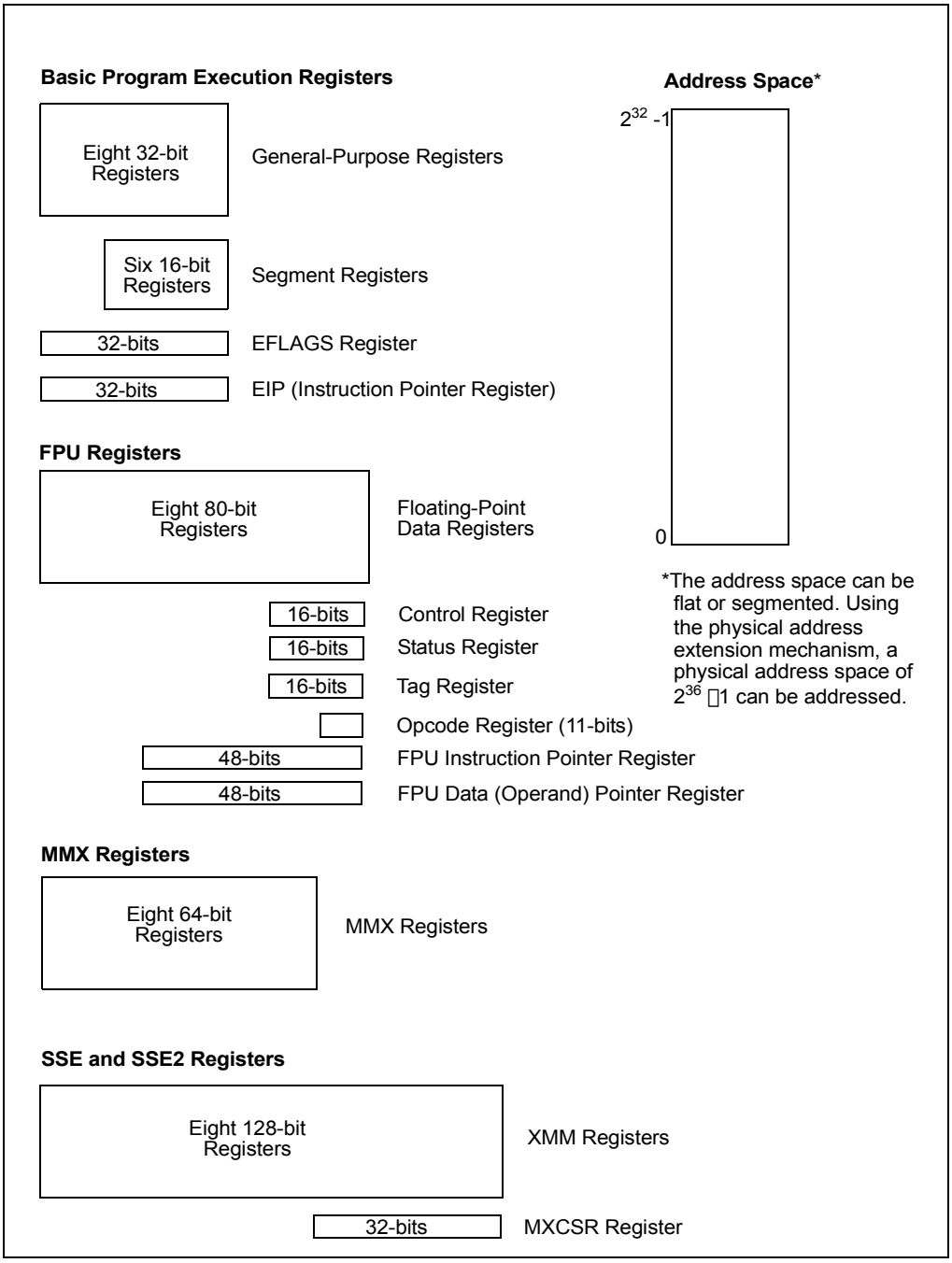

Figure 3-1. IA-32 Basic Execution Environment

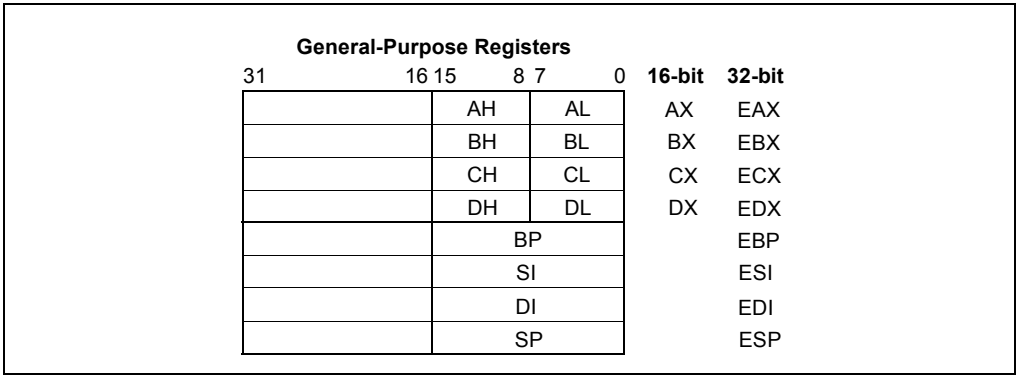

Figure 3-4. Alternate General-Purpose Register Names

toupper.asm

- Prompt for user input.
- Use Linux system call to get user input.
- Scan each character of user input and convert all lower case characters to upper case.
- How to:
  - work with 8-bit data
  - specify ASCII constant
  - compare values
  - loop control

```
 1  ; File: toupper.asm last updated 09/26/2001
 2  ;
 3  ; Convert user input to upper case.
 4  ;
 5  ; Assemble using NASM:  nasm -f elf toupper.asm
 6  ; Link with ld:  ld toupper.o
 7  ;
 8
 9  %define STDIN 0
10  %define STDOUT 1
11  %define SYSCALL_EXIT  1
12  %define SYSCALL_READ  3
13  %define SYSCALL_WRITE 4
14  %define BUFLEN 256
15
16
17          SECTION .data                   ; initialized data section
18
19  msg1:   db "Enter string: "             ; user prompt
20  len1:   equ $-msg1                      ; length of first message
21
22  msg2:   db "Original: "                 ; original string label
23  len2:   equ $-msg2                      ; length of second message
24
25  msg3:   db "Convert:  "                 ; converted string label
26  len3:   equ $-msg3
27
28  msg4:   db 10, "Read error", 10         ; error message
29  len4:   equ $-msg4
30
31
32          SECTION .bss                    ; uninitialized data section
33  buf:    resb BUFLEN                     ; buffer for read
34  newstr: resb BUFLEN                     ; converted string
35  rlen:   resb 4                          ; length
36
37
38          SECTION .text                   ; Code section.
39          global  _start                  ; let loader see entry point
40
41  _start: nop                             ; Entry point.
42  start:                                  ; address for gdb
43
44          ; prompt user for input
45          ;
46          mov     eax, SYSCALL_WRITE      ; write function
47          mov     ebx, STDOUT             ; Arg1: file descriptor
48          mov     ecx, msg1               ; Arg2: addr of message
49          mov     edx, len1               ; Arg3: length of message
50          int     080h                    ; ask kernel to write
51
```

```
52          ; read user input
53          ;
54          mov     eax, SYSCALL_READ       ; read function
55          mov     ebx, STDIN              ; Arg 1: file descriptor
56          mov     ecx, buf                ; Arg 2: address of buffer
57          mov     edx, BUFLEN             ; Arg 3: buffer length
58          int     080h
59
60          ; error check
61          ;
62          mov     [rlen], eax             ; save length of string read
63          cmp     eax, 0                  ; check if any chars read
64          jg      read_OK                 ; >0 chars read = OK
65          mov     eax, SYSCALL_WRITE      ; ow print error mesg
66          mov     ebx, STDOUT
67          mov     ecx, msg4
68          mov     edx, len4
69          int     080h
70          jmp     exit                    ; skip over rest
71  read_OK:
72
73
74          ; Loop for upper case conversion
75          ; assuming rlen > 0
76          ;
77  L1_init:
78          mov     ecx, [rlen]             ; initialize count
79          mov     esi, buf                ; point to start of buffer
80          mov     edi, newstr             ; point to start of new string
81
82  L1_top:
83          mov     al, [esi]               ; get a character
84          inc     esi                     ; update source pointer
85          cmp     al, 'a'                 ; less than 'a'?
86          jb      L1_cont
87          cmp     al, 'z'                 ; more than 'z'?
88          ja      L1_cont
89          and     al, 11011111b           ; convert to uppercase
90
91  L1_cont:
92          mov     [edi], al               ; store char in new string
93          inc     edi                     ; update dest pointer
94          dec     ecx                     ; update char count
95          jnz     L1_top                  ; loop to top if more chars
96  L1_end:
97
98
```

```
 99          ; print out user input for feedback
100          ;
101          mov     eax, SYSCALL_WRITE      ; write message
102          mov     ebx, STDOUT
103          mov     ecx, msg2
104          mov     edx, len2
105          int     080h
106
107          mov     eax, SYSCALL_WRITE      ; write user input
108          mov     ebx, STDOUT
109          mov     ecx, buf
110          mov     edx, [rlen]
111          int     080h
112
113          ; print out converted string
114          ;
115          mov     EAX, SYSCALL_WRITE      ; write message
116          mov     EBX, STDOUT
117          mov     ECX, msg3
118          mov     EDX, len3
119          int     080h
120
121          mov     EAX, SYSCALL_WRITE      ; write out string
122          mov     EBX, STDOUT
123          mov     ECX, newstr
124          mov     EDX, [rlen]
125          int     080h
126
127
128          ; final exit
129          ;
130  exit:   mov     EAX, SYSCALL_EXIT       ; exit function
131          mov     EBX, 0                  ; exit code, 0=normal
132          int     080h                    ; ask kernel to take over
```

Read The Friendly Manual (RTFM)

- **Best Source: Intel Instruction Set Reference**
  - Available off the course web page in PDF.
  - Download it, you'll need it.
- **Next Best Source: Appendix A NASM Doc.**
- **Questions to ask:**
  - What is the instruction's basic function? (e.g., adds two numbers)
  - Which addressing modes are supported? (e.g., register to register)
  - What side effects does the instruction have? (e.g. OF modified)

Branching Instructions

- JMP = unconditional jump
- Conditional jumps use the flags to decide whether to jump to the given label or to continue.
- The flags were modified by previous arithmetic instructions or by a compare (CMP) instruction.
- The instruction

```
CMP     op1, op2
```

computes the unsigned and two's complement subtraction op1 - op2 and modifies the flags. The contents of op1 are not affected.

Example of CMP instruction

- Suppose AL contains 254. After the instruction:

```
CMP     AL, 17
```

CF = 0, OF = 0, SF = 1 and ZF = 0.

- A JA (jump above) instruction would jump.
- A JG (jump greater than) instruction wouldn't jump.
- Both signed and unsigned comparisons use the same CMP instruction.
- Signed and unsigned jump instructions interpret the flags differently.

Table 7-4. Conditional Jump Instructions

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CF or ZF)=0	Above/not below or equal
JAE/JNB	CF=0	Above or equal/not below
JB/JNAE	CF=1	Below/not above or equal
JBE/JNA	(CF or ZF)=1	Below or equal/not above
JC	CF=1	Carry
JE/JZ	ZF=1	Equal/zero
JNC	CF=0	Not carry
JNE/JNZ	ZF=0	Not equal/not zero
JNP/JPO	PF=0	Not parity/parity odd
JP/JPE	PF=1	Parity/parity even
JCXZ	CX=0	Register CX is zero
JECXZ	ECX=0	Register ECX is zero
Signed Conditional Jumps		
JG/JNLE	((SF xor OF) or ZF) =0	Greater/not less or equal
JGE/JNL	(SF xor OF)=0	Greater or equal/not less
JL/JNGE	(SF xor OF)=1	Less/not greater or equal
JLE/JNG	((SF xor OF) or ZF)=1	Less or equal/not greater
JNO	OF=0	Not overflow
JNS	SF=0	Not sign (non-negative)
JO	OF=1	Overflow
JS	SF=1	Sign (negative)

The destination operand specifies a relative address (a signed offset with respect to the address in the EIP register) that points to an instruction in the current code segment. The J*cc* instructions do not support far transfers; however, far transfers can be accomplished with a combination of a J*cc* and a JMP instruction (see "J*cc*—Jump if Condition Is Met" in Chapter 3 of the *Intel Architecture Software Developer's Manual, Volume 2*).

Table 7-4 shows the mnemonics for the J*cc* instructions and the conditions being tested for each instruction. The condition code mnemonics are appended to the letter "J" to form the mnemonic for a J*cc* instruction. The instructions are divided into two groups: unsigned and signed conditional jumps.

Project 1: ROT13 Due Thursday, September 23, 2004

Objective

This project is a finger-warming exercise to make sure that everyone can compile an assembly language program, run it through the debugger and submit the requisite files using the systems in place for the programming projects.

Background

The ROT13 format is used on USENET newsgroups to mask potentially offensive postings, movie spoilers, etc. The idea is that readers who think they might be offended by a controversial remark will simply not "decode" the posting and thus not be offended. Many news readers and email clients support ROT13.

The encoding is very simple. The characters 'a'–'m' are mapped to 'n'–'z' and vice versa. Upper case letters are transformed analogously. All other characters (e.g., digits and punctuation marks) are left alone. For example, "There was a man from Nantucket" becomes "Gurer jnf n zna sebz Anaghpxrg" after ROT13 transformation. To decode a message in ROT13, you simply apply the ROT13 transformation again.

Assignment

For this project, you must do the following:

1. Write an assembly language program that prompts the user for an input string and prints out the ROT13 encoding of the the string. A good starting point for your project is the program `toupper.asm` (shown in class) which converts lower case characters in the user's input string to upper case. The source code is available on the GL file system at:

   ```
   /afs/umbc.edu/users/c/h/chang/pub/cs313/
   ```

2. Using the UNIX `script` command, record some sample runs of your program and a debugging session using `gdb`. In this session, you should fully exercise the debugger. You must set several breakpoints, single step through some instructions, use the automatic display function and examine the contents of memory before and after processing. The script command is initiated by typing `script` at the UNIX prompt. This puts you in a new UNIX shell which records every character typed or printed to the screen. You exit from this shell by typing `exit` at the UNIX prompt. A file named `typescript` is placed in the current directory. You must exit from the `script` command *before* submitting your project. Also, remember not to record yourself editing your programs — this makes the `typescript` file very large.

Turning in your program

Use the UNIX `submit` command on the GL system to turn in your project. You should submit two files: 1) the modified assembly language program and 2) the typescript file of your debugging session. The class name for submit is `cs313_0101`, the project name is `proj1`. The UNIX command to do this should look something like:

```
submit cs313_0101 proj1 rot13.asm typescript
```

Notes

Additional help on running NASM, `gdb` and making system calls in Linux are available on the assembly language programming web page for this course:

<http://www.csee.umbc.edu/~chang/cs313.f04/assembly.shtml>

Recall that the project policy states that programming assignments must be the result of individual effort. *You are not allowed to work together.* Also, your projects will be graded on five criteria: correctness, design, style, documentation and efficiency. So, it is not sufficient to turn in programs that assemble and run. Assembly language programming can be a messy affair — neatness counts.