CMSC 313 Lecture 10

- Project 3 Questions
- The Compilation Process: from *.c to a.out

Project 3: An Error-Correcting Code Due: Thursday October 14, 2004

Objective

The objectives of this programming project are 1) for you to gain familiarity with data manipulation at the bit level and 2) for you to write more complex assembly language programs.

Background

In Project 2, we saw that checksums can be used detect corrupted files. However, there is not much we can do after we have detected the corruption. An error-correcting code is able to fix errors, not just detect them.

In this project, we will use a 31-bit Hamming code that can correct a 1-bit error in each 32-bit codeword. Each 32-bit codeword encodes 3 bytes of the original data. The format of the codeword is shown on the next page.

Assignment

Write an assembly language program that encodes the input file using the codeword format described below. As in Project 2, use Unix input and output redirection:

```
./a.out <ifile >ifile.ham
```

Some details:

- Your program must read a block of bytes from the input. You should not read from the input one byte at a time or three bytes at a time. (That would be terribly inefficient.)
- You may assume that when the operating system returns with 0 bytes read that the end of the input file has been reached. On the other hand, you may not assume that the end of the file has been reached when the operating system gives you fewer bytes than your block size. Similarly, you may not assume that the operating system will comply with your request for a number of input bytes that is divisible by 3.
- The 32-bit codewords must be written out in little-endian format.

The C source code for two programs `decode.c` and `corrupt.c` are provided in the GL file system in the directory: `/afs/umbc.edu/users/c/h/chang/pub/cs313`. These two programs can be used to decode an encoded file and to corrupt an encoded file. You can use these programs to check if your program is working correctly. Both programs use I/O redirection.

Record some sample runs of your program using the Unix `script` command. You should show that you can encode a file using your program, then decode it and obtain a file that is identical to the original. Use the Unix `diff` command to compare the original file with the decoded file. You should also show that this works when the file is corrupted.

Implementation Notes

- The parity flag PF is set to 1 if the result of an instruction contains an even number of 1's. Unfortunately, PF only looks at the lowest 8 bits of the result. For this project, you will need to compute 32-bit parities. Here's a simple way to comput the parity of the EAX register.

```
mov     ebx,eax
shr     eax,16
xor     ax,bx
xor     al,ah
jp      even_label
```

  Note that the EAX and EBX registers are modified in this process, so you may need to use different registers.

- A main issue in this project is handling the "extra characters" at the end of a block of input after you have processed all the 3-byte "groups". E.g., if your block size 128, then you will have 2 characters left over after processing 42 three-byte groups (42 x 3 = 126). These 2 extra characters must be grouped with the first character of the next block (if there is a next block). Think about this situation *before* you begin coding.

- Another main issue is the last 32-bit word output by your program. Note that the bits m1 and m0 must be set *before* you compute the parity bits p4, p3, p2, p1 and p0.

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 assembly language program and 2) the typescript file of sample runs of your program. The class name for submit is `cs313_0101`. The name of the assignment name is `proj3`. The UNIX command to do this should look something like:

```
submit cs313_0101 proj3 encode.asm typescript
```

Codeword format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
a7	a6	a5	a4	a3	a2	a1	a0	b7	b6	b5	b4	b3	b2	b1	p4	b0	c7	c6	c5	c4	c3	c2	p3	c1	c0	m1	p2	m0	p1	p0	0

bit 0 is not used and always holds a 0.

1st byte of data = a7 a6 a5 a4 a3 a2 a1 a0

2nd byte of data = b7 b6 b5 b4 b3 b2 b1 b0

3rd byte of data = c7 c6 c5 c4 c3 c2 c1 c0

p4, p3, p2, p1 and p0 are used to ensure that these bit positions have an even number of 1's:

```
p0:  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31
p1:  2  3  6  7 10 11 14 15 18 19 22 23 26 27 30 31
p2:  4  5  6  7 12 13 14 15 20 21 22 23 28 29 30 31
p3:  8  9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
p4: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

m1 and m0 are only used in the last word of the encoded file. They depend on the original file size (in number of bytes).

m1 m0 = 00 if the file size mod 3 is 0

m1 m0 = 01 if the file size mod 3 is 1

m1 m0 = 10 if the file size mod 3 is 2

The Compilation Process: Major Steps

- **Lexical Analysis**
  - Converts source code to a token stream
- **Parsing**
  - Construct a parse tree from the token stream
- **Code Generation**
  - Produce native assembly language code from parse tree
- **Assembling**
  - Produce machine language code from assembly language source
- **Linking & Loading**
  - Resolve external references
  - Assign addresses to code and data sections

Lexical Analysis

- Groups together characters into "tokens"
- recognizes keywords, identifiers, constants, ...
- strips out comments, white space, ...
- Unix tool for lexical analysis: **lex**

```
[if] [(] [x] [+] [y] [<=] [74.2] [)] [{]
    [a] [=] [x] [+] [7] [;]
[else] [{]
    [printf] [(] ["Out of bounds!\n"] [)] [;]
[}]
```

Parsing

- Uses context-free grammar (a.k.a. Backus-Naur Form) for the language to construct a parse tree.

A simple grammar:

```
E -> E + T
E -> E - T
E -> T
T -> T * F
T -> T / F
T -> F
F -> <id>
F -> <const>
F -> ( E )
```

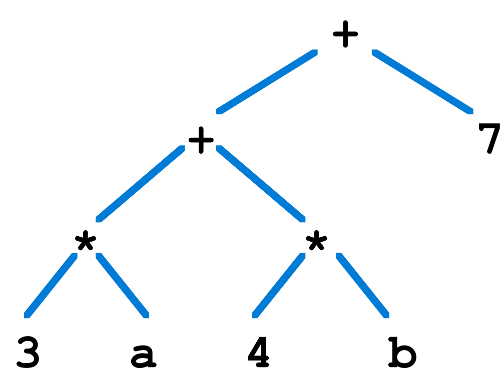
Deriving 3 * a + 4 * b +7:

```
E -> E + T
  -> E + T + T
  -> T + T + T
  -> T * F + T + T
  -> F * F + T + T
  -> 3 * F + T + T
  -> 3 * a + T + T
  -> 3 * a + T * F + T
  -> 3 * a + F * F + T
  -> 3 * a + 4 * F + T
  -> 3 * a + 4 * b + T
  -> 3 * a + 4 * b + 7
```

Parse Trees

- Constructing a parse tree is essentially the reverse of the derivation process
- Unix tool: **yacc** (yet another compiler compiler)

Parse tree for 3 * a + 4 * b +7:



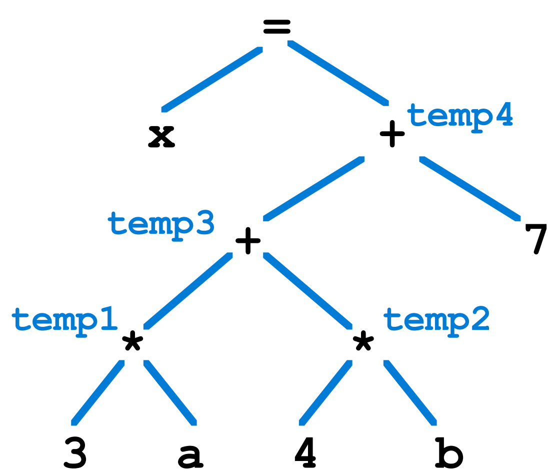
Code Generation

- Produce “intermediate“ code from parse tree.
- Produce native assembly language code from intermediate code.
- Code optimization may be used in both steps.

Code Generation Example 1

- Use EAX to perform +, *, ...
- Store result in temporary location

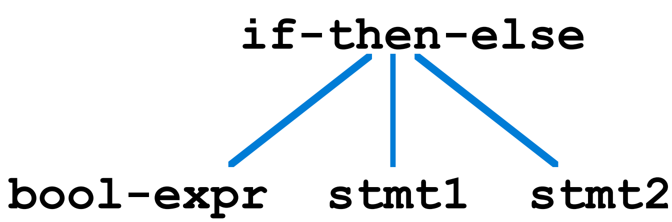
Parse tree for x = 3 * a + 4 * b +7:




```
mov     eax, 3
imul    eax, [a]
mov     [temp1], eax
mov     eax, 4
imul    eax, [b]
mov     [temp2], eax
mov     eax, [temp1]
add     eax, [temp2]
mov     [temp3], eax
mov     eax, [temp3]
add     eax, 7
mov     [temp4], eax
mov     eax, [temp4]
mov     [x], eax
```

Code Generation Example 2

Parse tree for if-then-else statements




```
bool_expr:
        .
        .
        .
        mov     eax, [temp1]
        cmp     eax, 0
        je      stmt2

stmt1:
        .
        .
        .
        jmp     end_if

stmt2:
        .
        .
        .

end_if
```

Assembling

- Line-by-line translation of assembly language mnemonics to machine code
- two passes needed to resolve forward jumps

```
                               ; File: add2.asm
                               ;
                               ; Various addressing modes with the add operation.
                               ;

                                      section .data

00000000 2A000000              x:     dd      42              ; 4-byte word


                                      section .text
                                      global _start

00000000 90                    _start: nop

                                      ; initialize

00000001 B811000000            start:  mov     eax, 17         ; eax := 17
00000006 BB[00000000]                  mov     ebx, x          ; ebx := address of x
0000000B B909000000                    mov     ecx, 9          ; ecx := 9

00000010 0503000000                    add     eax, 3          ; add immediate
00000015 01C8                          add     eax, ecx        ; add 32-bit registers
00000017 6601C8                        add     ax, cx          ; add 16-bit registers
0000001A 0305[00000000]                add     eax, [x]        ; add memory
00000020 0303                          add     eax, [ebx]      ; add register indirect
00000022 8105[00000000]0500-           add     [x], dword 5    ; add immediate to mem
0000002A 0000
0000002C 0105[00000000]                add     [x], eax        ; add register to mem

                               ; these two are not allowed (commented out):
                               ;       add     [x], [x]        ; add mem to mem
                               ;       add     [x], [ebx]      ; add reg indirect to mem
```

Machine Code

- Instructions are represented internally as sequences of bits
- The assembly process translates the mnemonic into machine code
- The length 80386 machine instructions varies from one to six bytes
- Instructions containing memory addresses are the longest
- Decoding the operation code within the instructions determine its length and the number and type of operands
- The –l option of NASM create a list file that includes both the assembly mnemonic and their machine language counterpart
- Since the 80386 has 8 registers, register's address has to be 3 bits
- Instructions that assume some defaults registers, e.g. IN and OUT, do not to specify the address of register operands

3- Bit Register Codes			
EAX	000	ESP	100
ECX	001	EBP	101
EDX	010	ESI	110
EBX	011	EDI	111

ADD—Add

Opcode	Instruction	Description
04 *ib*	ADD AL,*imm8*	Add *imm8* to AL
05 *iw*	ADD AX,*imm16*	Add *imm16* to AX
05 *id*	ADD EAX,*imm32*	Add *imm32* to EAX
80 /0 *ib*	ADD *r/m8,imm8*	Add *imm8* to *r/m8*
81 /0 *iw*	ADD *r/m16,imm16*	Add *imm16* to *r/m16*
81 /0 *id*	ADD *r/m32,imm32*	Add *imm32* to *r/m32*
83 /0 *ib*	ADD *r/m16,imm8*	Add sign-extended *imm8* to *r/m16*
83 /0 *ib*	ADD *r/m32,imm8*	Add sign-extended *imm8* to *r/m32*
00 /*r*	ADD *r/m8,r8*	Add *r8* to *r/m8*
01 /*r*	ADD *r/m16,r16*	Add *r16* to *r/m16*
01 /*r*	ADD *r/m32,r32*	Add r32 to *r/m32*
02 /*r*	ADD *r8,r/m8*	Add *r/m8* to *r8*
03 /*r*	ADD *r16,r/m16*	Add *r/m16* to *r16*
03 /*r*	ADD *r32,r/m32*	Add *r/m32* to *r32*

Description

Adds the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
DEST    DEST + SRC;
```

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

The ModRM Byte

- The ModRM byte is used to carry operand information when the first byte cannot do so
- The ModRM byte distinguishes between the different versions of the same instruction, e.g. ADD rv, rmv

M O D		R			M		
7	6	5	4	3	2	1	0

Instruction ModRM byte

- When the MOD bits are 11, the M bits designate a register (interpreted for memory otherwise)

rmv, rv Coding

ADD EDI, EAX ➔ 01 C7

MOD		R			M		
7	6	5	4	3	2	1	0
1	1	0	0	0	1	1	1

rv, rmv Coding

M O D		R			M		
7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0

ADD EDI, EAX ➔ 03 F8

M O D		R			M		
7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	0

ADD EDI, [EAX+5] ➔ 03 78 05

ADD EDI, [EAX+87654321H] ➔ 03 B8 21 43 65 87 (MOD = 10)

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
/digit (Opcode)			0	1	2	3	4	5	6	7
REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
disp8[EAX][3]	01	000	40	48	50	58	60	68	70	78
disp8[ECX]		001	41	49	51	59	61	69	71	79
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
disp8[EBX];		011	43	4B	53	5B	63	6B	73	7B
disp8[--][--]		100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB
disp32[--][--]		100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
disp32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement following the SIB byte, to be added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index.

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP[1] EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16[2]		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BP+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI]+disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI]+disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI]+disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP]+disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX]+disp16		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	EQ	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AHMM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The “disp16” nomenclature denotes a 16-bit displacement following the ModR/M byte, to be added to the index.
3. The “disp8” nomenclature denotes an 8-bit displacement following the ModR/M byte, to be sign-extended and added to the index.

```
; File: twopass.asm
;
; Demonstrating a two-pass assembler

        section .data
x:      db      87h
y:      dw      1492h
zalias  equ $
z:      dd      17762001h

calc equ (x-y)*2
x4 equ x+1

        section .text
        global _start

_start: mov     eax, [zalias]
        mov     bx, [y]
        mov     cx, [x4]
        cmp     bx, cx
        jne     error

OK:     add     ax, bx
        mov     [x], al
        mov     ebx, 0          ; 0=normal exit

done:   mov     eax, 1          ; syscall number for exit
        int     080h

error:  mov     ebx, 17         ; abnormal exit
        jmp     done
```

```
                               ; File: twopass.asm
                               ;
                               ; Demonstrating a two-pass assembler

                                       section .data
00000000 87                    x:      db      87h
00000001 9214                  y:      dw      1492h
                               zalias  equ $
00000003 01207617              z:      dd      17762001h

                               calc equ (x-y)*2
                               x4 equ x+1

                                       section .text
                                       global _start

00000000 A1[03000000]          _start: mov     eax, [zalias]
00000005 668B1D[01000000]              mov     bx, [y]
0000000C 668B0D[01000000]              mov     cx, [x4]
00000013 6639CB                        cmp     bx, cx
00000016 7514                          jne     error

00000018 6601D8                OK:     add     ax, bx
0000001B A2[00000000]                  mov     [x], al
00000020 BB00000000                    mov     ebx, 0          ; 0=normal exit

00000025 B801000000            done:   mov     eax, 1          ; syscall number for exit
0000002A CD80                          int     080h

0000002C BB11000000            error:  mov     ebx, 17         ; abnormal exit
00000031 E9EFFFFFFF                    jmp     done
```

Linking & Loading

- Linking resolves external references to data and code, including calls to library functions.
- References are often raw addresses without type.
- Loading assigns addresses to data & code sections.
- The loader must patch every absolute memory reference in the code with the assigned address:

```
MOV     EAX, [x]        ; value of x is patched by the loader
```

- In UNIX, `ld` performs both linking & loading.

```
linux3% nasm -f elf -l twopass.lst twopass.asm
linux3% ld twopass.o

linux3% a.out  ; echo $?
0

linux3% objdump -t twopass.o

twopass.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 twopass.asm
00000000 l    d  *ABS*  00000000
00000000 l    d  .data  00000000
00000000 l    d  .text  00000000
00000000 l       .data  00000000 x
00000001 l       .data  00000000 y
00000003 l       .data  00000000 zalias
00000003 l       .data  00000000 z
fffffffe l       *ABS*  00000000 calc
00000001 l       .data  00000000 x4
00000018 l       .text  00000000 OK
00000025 l       .text  00000000 done
0000002c l       .text  00000000 error
00000000 g       .text  00000000 _start
```

```
linux3% objdump -t a.out

a.out:     file format elf32-i386

SYMBOL TABLE:
08048080 l    d  .text  00000000
080490b8 l    d  .data  00000000
080490bf l    d  .bss   00000000
00000000 l    d  .comment       00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  *ABS*  00000000
00000000 l    df *ABS*  00000000 twopass.asm
080490b8 l       .data  00000000 x
080490b9 l       .data  00000000 y
080490bb l       .data  00000000 zalias
080490bb l       .data  00000000 z
fffffffe l       *ABS*  00000000 calc
080490b9 l       .data  00000000 x4
08048098 l       .text  00000000 OK
080480a5 l       .text  00000000 done
080480ac l       .text  00000000 error
080480b6 g     O *ABS*  00000000 _etext
08048080 g       .text  00000000 _start
080490bf g     O *ABS*  00000000 __bss_start
080490bf g     O *ABS*  00000000 _edata
080490c0 g     O *ABS*  00000000 _end


linux3% objdump -h a.out

a.out:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000036  08048080  08048080  00000080  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000007  080490b8  080490b8  000000b8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000001  080490bf  080490bf  000000bf  2**0
                  CONTENTS
  3 .comment      0000001c  00000000  00000000  000000c0  2**0
                  CONTENTS, READONLY
linux3%
```

```
linux3% objdump -d a.out

a.out:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       a1 bb 90 04 08          mov    0x80490bb,%eax
 8048085:       66 8b 1d b9 90 04 08    mov    0x80490b9,%bx
 804808c:       66 8b 0d b9 90 04 08    mov    0x80490b9,%cx
 8048093:       66 39 cb                cmp    %cx,%bx
 8048096:       75 14                   jne    80480ac <error>

08048098 <OK>:
 8048098:       66 01 d8                add    %bx,%ax
 804809b:       a2 b8 90 04 08          mov    %al,0x80490b8
 80480a0:       bb 00 00 00 00          mov    $0x0,%ebx

080480a5 <done>:
 80480a5:       b8 01 00 00 00          mov    $0x1,%eax
 80480aa:       cd 80                   int    $0x80

080480ac <error>:
 80480ac:       bb 11 00 00 00          mov    $0x11,%ebx
 80480b1:       e9 ef ff ff ff          jmp    80480a5 <done>
```

Next Time

- Subroutines & the stack

References

- Some figures and diagrams from *IA-32 Intel Architecture Software Developer's Manual, Vols 1-3*

  <http://developer.intel.com/design/Pentium4/manuals/>