

CMSC 313 Lecture 10

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

Project 3: External Records

Due: Tue 10/07/03, Section 0101 (Chang) & Section 0301 (Macneil)
Wed 10/08/03, Section 0201 (Patel & Bourner)

Objective

The objective of this programming project is to gain experience writing more complex assembly language programs and to use indexed addressing modes.

Assignment

Your assembly language program for this project will work with an externally defined array of records. This array is defined in a C program as follows:

```
struct {
    char  realname[32] ;
    char  nickname[16] ;
    char  alignment[20] ;
    char  role[20] ;
    int   points ;
    int   level ;
} records[10] ;

int num_records = 10 ;
```

The records in the array have pre-initialized values not shown here. The full text of the C program is available on the GL file system at: `/afs/umbc.edu/users/c/h/chang/pub/cs313/records.c`

Your assembly language program must search through the array and find the record with the least number of points and the record with the alphabetically first nickname. It must then print out the `realname` field of these two records. E.g.,

```
Lowest Points: James Pressman
First Nickname: Dan Gannett
```

Implementation Notes

- The sample data in `records.c` contains 10 records, but your program should work with any number of records. The number of records is stored in the `int` variable `num_records`.
- In order to access the externally defined array and integer variable, you must have the following declaration in your assembly language program:

```
extern    records, num_records
```

- You must also make your own test cases. The example in `records.c` does not fully exercise your program. Your program will be graded based upon other test cases.
- You will need to link your assembly language program with the data defined in the C program:

```
gcc -c records.c
nasm -f elf report.asm
ld records.o report.o
```

- An important part of this project is deciding how to use indexed addressing to access the data in the records. Think this through carefully. A clean and logical approach to this problem will yield clean and logical code that is easier to construct and, more importantly, easier to debug.

- Your program should be reasonably robust and report errors encountered (e.g., empty array) rather than crashing.
- Note that the strings stored in the array are C-style null-terminated strings.
- Nicknames should be compared using dictionary ordering. For example, any string starting with the letter 'a' comes before any string that starts with 'b'. In the case that one string is a prefix of another, the shorter string come first. E.g., "egg" comes before "egghead".
- To access each field of the record, you should use an offset from the address of the record. You should use `%define` constants instead of magic numbers. E.g.,

```
%define NickOffset 32
%define AlignOffset 48
%define RoleOffset 68
%define PointsOffset 88
%define LevelOffset 92
%define RecSize 96
```

- Project 4 will be based upon Project 3, so keep in mind that you will need to extend/modify this program.

Turning in your program

Use the UNIX `submit` command on the GL system to turn in your project. You should submit at least 4 files: your assembly language program, at least 2 of your own test cases and a typescript file of sample runs of your program. The class name for submit is `cs313_0101`, `cs313_0102` or `cs313_0103` for respectively sections 0101 (Chang), 0201 (Patel & Bourner) or 0301 (Macneil). The name of the assignment name is `proj3`. The UNIX command to do this should look something like:

```
submit cs313_0103 proj3 report.asm myrec1.c myrec2.c typescript
```

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 )
```

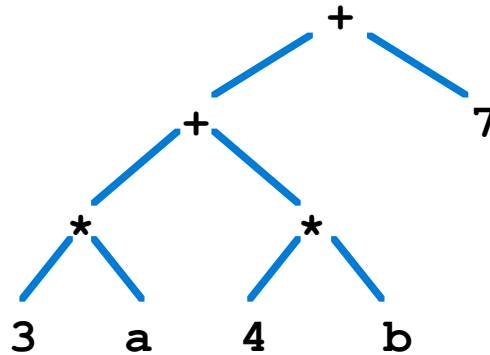
Deriving $3 * a + 4 * b + 7$:

```
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 $3 * a + 4 * b + 7$:



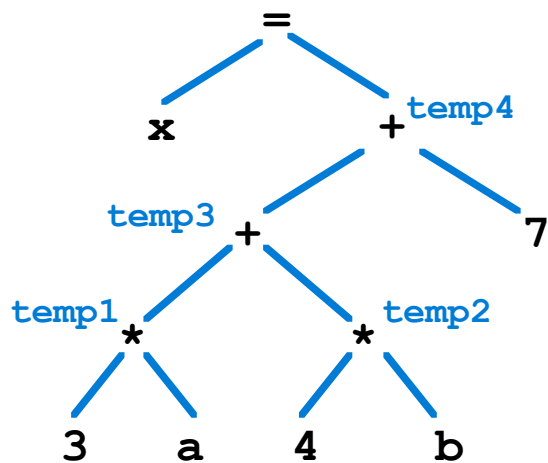
Code Generation

- 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

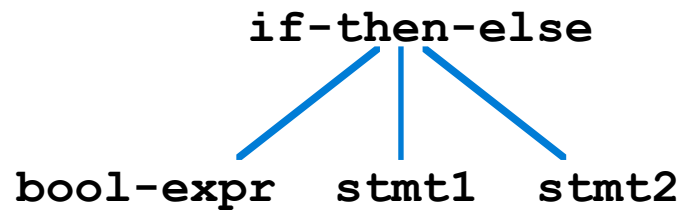
Parse tree for $x = 3 * a + 4 * b + 7$:



```
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**

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



Instruction Encoding

“S” represents a bit in the address of the source register

“D” represents bits in the address of the destination

One byte instruction

Conditional code is specified separately

Instruction	First Byte	Second Byte	4 More
MOV reg, reg	89	11SS SDDD	
MOV reg, imm	1011 1DDD		-----
ADD reg, reg	01	1100 0DDD	
ADD reg, imm	81	11SS SDDD	-----
SUB reg, reg	29	1110 1DDD	
SUB reg, imm	81		-----
INC reg	0100 0DDD		
DEC reg	0100 1DDD		
IN EAX, [DX]	ED		
OUT [DX], EAX	EF		
RET	C3		
JMP imm	E9		-----
JZ imm	0F	84	-----
JNZ imm	0F	85	-----
JS imm	0F	88	-----
JNS imm	0F	89	-----



Encoding MOV Instructions

<u>Label</u>	<u>Source Code</u>	<u>Address</u>	<u>Machine Code</u>
	MOV EDX, 0	0	BA 00
		2	00 00
		4	00
	IN EAX, [DX]	5	ED
	MOV ECX, EAX	6	89 C1
	IN EAX, [DX]	8	ED
	MOV EDX, EAX	9	89 C2
ORD:	SUB EAX, ECX	B	29 C8
	JZ GCD	D	0F 84
		F	11 00
		11	00 00
	JNS NXT	13	0F 89
		15	04 00
		17	00 00
	MOV EAX, ECX	19	89 C8
	MOV ECX, EDX	1B	89 D1
NXT:	MOV EDX, EAX	1D	89 C2
	JMP ORD	1F	E9 E7
		21	FF FF
		23	FF
GCD:	MOV EAX, EDX	24	89 D0
	MOV EDX, 1	26	BA 01
		28	00 00
		2A	00
	OUT [DX], EAX	2B	EF
	RET	2C	C3

$1011\ 1DDD \rightarrow (1011\ 1010)_2 = (BA)_{16}$
 $11SS\ SDDD \rightarrow (1100\ 0001)_2 = (C1)_{16}$

(00000001)₁₆ is stored 01 00 00 00
 (little endian byte order)



Relative Jumps

<u>Label</u>	<u>Source Code</u>	<u>Address</u>	<u>Machine Code</u>
	MOV EDX, 0	0	BA 00
		2	00 00
		4	00
	IN EAX, [DX]	5	ED
	MOV ECX, EAX	6	89 C1
	IN EAX, [DX]	8	ED
	MOV EDX, EAX	9	89 C2
ORD:	SUB EAX, ECX	B	29 C8
	JZ GCD	D	0F 84
		F	11 00
		11	00 00
	JNS NXT	13	0F 89
		15	04 00
		17	00 00
	MOV EAX, ECX	19	89 C8
	MOV ECX, EDX	1B	89 D1
NXT:	MOV EDX, EAX	1D	89 C2
	JMP ORD	1F	E9 E7
		21	FF FF
		23	FF
GCD:	MOV EAX, EDX	24	89 D0
	MOV EDX, 1	26	BA 01
		28	00 00
		2A	00
	OUT [DX], EAX	2B	EF
	RET	2C	C3

Jump encoding uses relative addresses rather than absolute addresses when possible

Relative encoding specifies the number of bytes to jump starting from the beginning of the next instruction

Target instruction is 4 bytes away (2 MOV instructions in between)

Backward jump to an instruction that is 25 bytes away $(\text{FFFFFFE7})_{16} = (-25)_{10}$

Forward jumps use positive increments while negative ones allow backward jumps



Short Jumps

- ❑ Most conditional and unconditional jumps are statistically found to be made to close points in the program (principle of spatial locality)
- ❑ Short jump instructions are used for small address increments to optimize code size and performance (fewer bytes to read from memory and to load to registers)
- ❑ NASM encodes JMP using relative addressing with one byte increment (other assembler might require explicit suffix)

Example: JNS NXT → 0F 89 04 → 79 04 (drop 0F and subtract 10H)
 JMP ORD → EB E7 (EB is the code for short JMP)

- ❑ The “JMP NEAR” instruction is used for jump to points in the program that are more distant (more bytes are dedicated to the increment)
- ❑ The “JMP FAR” instruction allows jumps to remote execution points using absolute addresses



Opcode Space

❑ The general format for a machine instruction is: opcode operands

❑ Operands are described using the following notation:

r: register *m*: memory *i*: immediate *v*: 32 bits or 16 bits

b: byte **2**: 2 bytes *e*: indicates E for 32-bit registers

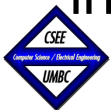
Example: ADD *rv*, *rmv* means that the 1st operand is either 32 or 16-bit register and the 2nd operand is 32 or 16-bit register or memory

Example: MOV *eAX*, [*iv*] means that the 1st operand is either the AX or EAX register and the 2nd operand a memory location addressable with a 32 or 16-bit immediate value or memory

❑ In few cases one byte suffices to completely specify the instruction, e.g. F4 specifies the halt instruction HLT

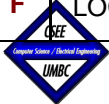
❑ For some instructions, a second (ModRM) byte is needed to specify the operand types (e.g. MOV *rmv*, *rv* and MOV *rv*, *rmv*)

❑ If the first byte is 0F, the second describe the operation code of an instruction that works only for 80386 (not backward compatible)



First Byte of Opcode Space

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD rmb, rb	ADD rmv, rv	ADD rb, rmb	ADD rv,rmv	ADD AL,ib	ADD eAX ,iv	PUSH ES	POP ES	OR rmb,rb	OR rmv,rv	OR rb,rmb	OR Rv,rmv	OR AL,ib	OR eAX,iv	PUSH CS	386 space
1	ADC rmb, rb	ADC rmv, rv	ADC rb, rmb	ADC rv,rmv	ADC AL,ib	ADC eAX ,iv	PUSH SS	POP SS	SBB rmb,rb	SBB rmv,rv	SBB rb,rmb	SBB Rv,rmv	SBB AL,ib	SBB eAX,iv	PUSH DS	POP DS
2	AND rmb, rb	AND rmv, rv	AND rb, rmb	AND rv,rmv	AND AL,ib	AND eAX ,iv	ES:	DAA	SUB rmb,rb	SUB rmv,rv	SUB rb,rmb	SUB Rv,rmv	SUB AL,ib	SUB eAX,iv	CS:	DAS
3	XOR rmb, rb	XOR rmv, rv	XOR rb, rmb	XOR rv,rmv	XOR AL,ib	XOR eAX ,iv	SS:	AAA	CMP rmb,rb	CMP rmv,rv	CMP rb,rmb	CMP Rv,rmv	CMP AL,ib	CMP eAX,iv	DS:	AAS
4	INC eAX	INCeCX	INC eDX	INC eBX	INC eSP	INC eBP	INC eSI	INC eDI	DEC eAX	DEC eCX	DEC eDX	DEC eBX	DEC eSP	DEC eBP	DEC eSI	DEC eDI
5	PUSH eAX	PUSH eCX	PUSH eDX	PUSH eBX	PUSH eSP	PUSH eBP	PUSH eSI	PUSH eDI	POP eAX	POP eCX	POP eDX	POP eBX	POP eSP	POP eBP	POP eSI	POP eDI
6	PUSHA PUSHAD	POPA POPAD	BOUND rv,m2v	ARPL rm2,r2	FS:	GS:	OpLen	AdLen	PUSH iv	IMUL Rv,rmv,iv	PUSH ib	IMUL rv,rmv,ib	INSB	INSD	OUTSB	OUTSD
7	JO ib	JNO ib	JB ib	JAE lb	JE ib	JNE ib	JBE ib	JA lb	JS lb	JNS lb	JP ib	JNP ib	JL ib	JGE ib	JLE ib	JG ib
8	Immed rmb, ib	Immed rmv, iv		Immed rmv,ib	TEST rmb,rb	TEST rmv,rv	XCHG rmb, rb	XCHG rmv, rv	MOV rmb,rb	MOV rmb,rv	MOV rb,rmb	MOV rv,rmv	MOV rm,segr	LEA rv,rm	MOV segr,rm	POP rmv
9	NOP	XCHG eAX,eCX	XCHG EAX,eDX	XCHG eAX,eBX	XCHG eAX,eSP	XCHG eAX,eBP	XCHG eAX,eSI	XCHG eAX,eDI	CWDE CBW	CWQ CDQ	CALL FAR s:m	FWAIT	PUSHF	POPF	SAHF	LAHF
A	MOV AL,[iv]	MOV eAX,[iv]	MOV [iv],AL	MOV [iv],AX	MOV SB	MOV SD	CMP SB	CMP SD	TEST AL,rb	TEST eAX,iv	STOSB	STOSD	LODSB	LODSD	SCASB	SCASD
B	MOV AL,ib	MOV CL,ib	MOV DL,ib	MOV BL,ib	MOV AH,ib	MOV CH,ib	MOV DH,ib	MOV BH,ib	MOV eAX,iv	MOV eCX,iv	MOV Edx,iv	MOV eBX,iv	MOV eSP,iv	MOV eBP,iv	MOV eSI,iv	MOV eDI,iv
C	Shift rmb,ib	Shift rmv,ib	RET i 2	RET	LES rm,rmp	LDS rm, rmp	MOV rmb,ib	MOV rmv,iv	ENTER i2,ib	LEAVE	RETF i2	RETF	INT 3	INT ib	INTO	IRET
D	Shift rmb,1	Shift rmv,1	Shift rmb,CL	Shift rmv,CL	AAM	AAD		X LATB	87 space	87 space	87 space	87 space	87 space	87 space	87 space	87 space
E	LOOPNE short	LOOPE short	LOOP short	JeCXZ short	IN AL, [ib]	IN eAX,[ib]	OUT [ib],AL	OUT [ib],eAX	CALL iv	JMP iv	JMP FAR s:m	JMP lb	IN AL,[DX]	IN eAX,[DX]	OUT [DX],AL	OUT [DX],eAX
F	LOCK		REPNE	REP REPE	HLT	CMC	Unary rmb	Unary rmv	CLC	STC	CLI	STI	CLD	STD	IncDec rmb	Indir rmv



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

MOD		R			M		
7	6	5	4	3	2	1	0

Instruction ModRM byte

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

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

ADD EDI, EAX → 03 F8

MOD		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)



Non-register R bits

- ❑ In some cases, the first byte is insufficient to specify an operation, but the R bits in the ModRM byte are used to complete the specification
- ❑ Example: The value of the R bits determine the type of shift operation

<i>/r</i>	<i>/0</i> 000	<i>/1</i> 001	<i>/2</i> 010	<i>/3</i> 011	<i>/4</i> 100	<i>/5</i> 101	<i>/6</i> 110	<i>/7</i> 111
Immed	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
Shift	ROL	ROR	RCL	RCR	SHL	SHR	SAR	
Unary	TEST <i>i</i>		NOT	NEG	MUL	IMUL	DIV	IDIV
IncDec	INC	DEC						
Indir	INC	DEC	CALL <i>m</i>	CALL FAR <i>m</i>	JMP	JMP FAR	PUSH	

- ❑ Example: the instruction “SUB *rmv*, *ib*” can be coded out of “Immed *rmv*, *ib*” with 83 in the first byte and setting the R bits to 101

SUB ECX, 32 → 83 E9 20

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

Operand Size

32-bit and 16-bit Code

- ❑ The encoding of 32-bit and 16-bit operations in the 80386 machine language is the same (e.g. the machine code for both “MOV ECX, EAX” and “MOV CX, AX” is 89 C1)
- ❑ A configuration bit in the processor is responsible for determining the default operand (word) size
- ❑ Operand size prefix “66” is used to force word size different than the default

	32 –Bit Code	16-Bit Code
MOV ECX, EAX	89 C1	66 89 C1
MOV CX, AX	66 89 C1	89 C1

The 8-bit Registers

- ❑ 8-bit registers have dedicated opcodes
- ❑ Again the ModRM byte is used to determine operand types
- ❑ Example: MOV AL, CL → 8A C1

Codes for 8-Bit Registers			
AL	000	AH	100
CL	001	CH	101
DL	010	DH	110
BL	011	BH	111

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



```

1           ; File: add2.asm
2           ;
3           ; Various addressing modes with the add operation.
4           ;
5
6           section .data
7
8 00000000 2A000000      x:      dd      42           ; 4-byte word
9
10
11          section .text
12          global _start
13
14 00000000 90          _start: nop
15
16          ; initialize
17
18 00000001 B811000000    start:  mov      eax, 17       ; eax := 17
19 00000006 BB[00000000]    mov      ebx, x           ; ebx := address of x
20 0000000B B909000000    mov      ecx, 9          ; ecx := 9
21
22 00000010 0503000000    add     eax, 3           ; add immediate
23 00000015 01C8        add     eax, ecx         ; add 32-bit registers
24 00000017 6601C8      add     ax, cx           ; add 16-bit registers
25 0000001A 0305[00000000] add     eax, [x]         ; add memory
26 00000020 0303        add     eax, [ebx]      ; add register indirect
27 00000022 8105[00000000]0500-  add     [x], dword 5    ; add immediate to mem
28 0000002A 0000
29 0000002C 0105[00000000]    add     [x], eax        ; add register to mem
30
31          ; these two are not allowed (commented out):
32          ;      add     [x], [x]           ; add mem to mem
33          ;      add     [x], [ebx]        ; add reg indirect to mem
34

```

ADD—Add

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

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.



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

			AL AX	CL CX	DL DX	BL BX	AH SP	CH BP ¹	DH SI	BH DI	
			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI	
			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7	
			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7	
			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)								
[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	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 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.

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.

```
; 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
```

```

1          ; File: twopass.asm
2          ;
3          ; Demonstrating a two-pass assembler
4
5          section .data
6 00000000 87          x:      db      87h
7 00000001 9214       y:      dw      1492h
8          zalias equ $
9 00000003 01207617   z:      dd      17762001h
10
11         calc equ (x-y)*2
12         x4 equ x+1
13
14         section .text
15         global _start
16
17 00000000 A1[03000000] _start: mov     eax, [zalias]
18 00000005 668B1D[01000000]      mov     bx, [y]
19 0000000C 668B0D[01000000]      mov     cx, [x4]
20 00000013 6639CB          cmp     bx, cx
21 00000016 7514          jne     error
22
23 00000018 6601D8          OK:    add     ax, bx
24 0000001B A2[00000000]          mov     [x], al
25 00000020 BB00000000        mov     ebx, 0          ; 0=normal exit
26
27 00000025 B801000000          done:  mov     eax, 1          ; syscall number for exit
28 0000002A CD80          int     080h
29
30 0000002C BB11000000          error: mov     ebx, 17          ; abnormal exit
31 00000031 E9FFFFFFF          jmp     done
32
33

```

```
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	1	df	*ABS*	00000000	twopass.asm
00000000	1	d	*ABS*	00000000	
00000000	1	d	.data	00000000	
00000000	1	d	.text	00000000	
00000000	1		.data	00000000	x
00000001	1		.data	00000000	y
00000003	1		.data	00000000	zalias
00000003	1		.data	00000000	z
fffffffe	1		*ABS*	00000000	calc
00000001	1		.data	00000000	x4
00000018	1		.text	00000000	OK
00000025	1		.text	00000000	done
0000002c	1		.text	00000000	error
00000000	g		.text	00000000	_start

```
linux3% objdump -t a.out
```

```
a.out:      file format elf32-i386
```

```
SYMBOL TABLE:
```

```
08048080 1      d  .text  00000000
080490b8 1      d  .data  00000000
080490bf 1      d  .bss   00000000
00000000 1      d  .comment      00000000
00000000 1      d  *ABS*  00000000
00000000 1      d  *ABS*  00000000
00000000 1      d  *ABS*  00000000
00000000 1      df *ABS*  00000000 twopass.asm
080490b8 1      .data  00000000 x
080490b9 1      .data  00000000 y
080490bb 1      .data  00000000 zalias
080490bb 1      .data  00000000 z
fffffffe 1      *ABS*  00000000 calc
080490b9 1      .data  00000000 x4
08048098 1      .text  00000000 OK
080480a5 1      .text  00000000 done
080480ac 1      .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     %a1,0x80490b8
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>
```

```
; File: sep1.asm
;
; File 1 for separate compilation example
```

```
global gvar1, _start
extern gvar2, add_these
```

```
        section .data
foo:    db      12h
gvar1:  dd      17h
lvar1:  dd      42h

        section .text
_start: mov     eax, [gvar1]
        mov     ebx, [gvar2]
        mov     ecx, [lvar1]

        call    add_these      ; gvar1 := eax+ebx+ecx
        mov     ebx, [gvar1]   ; store in return code
        mov     eax, 1         ; syscall number for exit
        int     080h          ; bye-bye
```

```
; File: sep2.asm
;
; File 2 for separate compilation example
```

```
global gvar2, add_these
extern gvar1
```

```
        section .data
bar:    dw      07h
gvar2:  dd      03h
lvar1:  dd      02h      ; same name as other lvar1, OK

        section .text
add_these:                                ; no regs altered!
        mov     [gvar1], dword 0         ; clear destination
        add     [gvar1], eax
        add     [gvar1], ebx
        add     [gvar1], ecx
        ret
```



```

1          ; File: sep1.asm
2          ;
3          ; File 1 for separate compilation example
4
5          global gvar1, _start
6          extern gvar2, add_these
7
8          section .data
9
10         00000000 12          foo:      db      12h
11         00000001 17000000    gvar1:   dd      17h
12         00000005 42000000    lvar1:   dd      42h
13
14         section .text
15         00000000 A1[01000000]  _start:  mov     eax, [gvar1]
16         00000005 8B1D[00000000]  mov     ebx, [gvar2]
17         0000000B 8B0D[05000000]  mov     ecx, [lvar1]
18
19         00000011 E8(00000000)    call    add_these      ; gvar1 := eax+ebx+ecx
20         00000016 8B1D[01000000]  mov     ebx, [gvar1]   ; store in return code
21         0000001C B801000000     mov     eax, 1         ; syscall number for exit
22         00000021 CD80          int     080h          ; bye-bye

```

```

1           ; File: sep2.asm
2           ;
3           ; File 2 for separate compilation example
4
5           global gvar2, add_these
6           extern gvar1
7
8           section .data
9
10          00000000 0700          bar: dw 07h
11          00000002 03000000      gvar2: dd 03h
12          00000006 02000000      lvar1: dd 02h ; same name as other lvar1, OK
13
14          section .text
15          add_these:                ; no regs altered!
16          00000000 C705[00000000]0000- mov [gvar1], dword 0 ; clear destination
17          00000008 0000
18          0000000A 0105[00000000]    add [gvar1], eax
19          00000010 011D[00000000]    add [gvar1], ebx
20          00000016 010D[00000000]    add [gvar1], ecx
21          0000001C C3                ret

```

```
linux3% nasm -f elf -l sep1.lst sep1.asm
linux3% nasm -f elf -l spe2.lst sep2.asm
linux3% ld sep1.o sep2.o
linux3% a.out
linux3% echo $?
92
linux3%
```

```
linux3% objdump -h sep1.o
```

```
sep1.o:      file format elf32-i386
```

```
Sections:
```

Idx	Name	Size	VMA	LMA	File off	Algn
0	.data	00000009	00000000	00000000	00000180	2**2
		CONTENTS,	ALLOC,	LOAD,	DATA	
1	.text	00000023	00000000	00000000	00000190	2**4
		CONTENTS,	ALLOC,	LOAD,	RELOC,	READONLY,
		CODE				
2	.comment	0000001c	00000000	00000000	000001c0	2**0
		CONTENTS,	READONLY			

```
linux3%
```

```
linux3% objdump -t sep1.o
```

```
sep1.o:      file format elf32-i386
```

```
SYMBOL TABLE:
```

00000000	1	df	*ABS*	00000000	sep1.asm
00000000	1	d	*ABS*	00000000	
00000000	1	d	.data	00000000	
00000000	1	d	.text	00000000	
00000000	1		.data	00000000	foo
00000005	1		.data	00000000	lvar1
00000000			*UND*	00000000	gvar2
00000000			*UND*	00000000	add_these
00000001	g		.data	00000000	gvar1
00000000	g		.text	00000000	_start

```
linux3% objdump -h sep2.o
```

```
sep2.o:      file format elf32-i386
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.data	0000000a	00000000	00000000	00000180	2**2
		CONTENTS,	ALLOC,	LOAD,	DATA	
1	.text	0000001d	00000000	00000000	00000190	2**4
		CONTENTS,	ALLOC,	LOAD,	RELOC,	READONLY,
		CODE				
2	.comment	0000001c	00000000	00000000	000001b0	2**0
		CONTENTS,	READONLY			

```
linux3%
```

```
linux3% objdump -t sep2.o
```

```
sep2.o:      file format elf32-i386
```

SYMBOL TABLE:

00000000	1	df	*ABS*	00000000	sep2.asm
00000000	1	d	*ABS*	00000000	
00000000	1	d	.data	00000000	
00000000	1	d	.text	00000000	
00000000	1		.data	00000000	bar
00000006	1		.data	00000000	lvar1
00000000			*UND*	00000000	gvar1
00000002	g		.data	00000000	gvar2
00000000	g		.text	00000000	add_these

```
linux3% objdump -h a.out
```

```
a.out:      file format elf32-i386
```

```
Sections:
```

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	0000004d	08048080	08048080	00000080	2**4
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
1	.data	00000016	080490d0	080490d0	000000d0	2**2
			CONTENTS, ALLOC, LOAD, DATA			
2	.bss	00000002	080490e6	080490e6	000000e6	2**0
			CONTENTS			
3	.comment	00000038	00000000	00000000	000000e8	2**0
			CONTENTS, READONLY			

```
linux3% objdump -t a.out
```

```
a.out:      file format elf32-i386
```

```
SYMBOL TABLE:
```

08048080	l	d	.text	00000000	
080490d0	l	d	.data	00000000	
080490e6	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	sep1.asm
080490d0	l		.data	00000000	foo
080490d5	l		.data	00000000	lvar1
00000000	l	df	*ABS*	00000000	sep2.asm
080490dc	l		.data	00000000	bar
080490e2	l		.data	00000000	lvar1
080480cd	g	O	*ABS*	00000000	_etext
080480b0	g		.text	00000000	add_these
08048080	g		.text	00000000	_start
080490de	g		.data	00000000	gvar2
080490e6	g	O	*ABS*	00000000	__bss_start
080490d1	g		.data	00000000	gvar1
080490e6	g	O	*ABS*	00000000	_edata
080490e8	g	O	*ABS*	00000000	_end

```
linux3% objdump -d a.out
```

```
a.out:      file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048080 <_start>:
```

```
8048080:      a1 d1 90 04 08      mov     0x80490d1,%eax
8048085:      8b 1d de 90 04 08   mov     0x80490de,%ebx
804808b:      8b 0d d5 90 04 08   mov     0x80490d5,%ecx
8048091:      e8 1a 00 00 00     call   80480b0 <add_these>
8048096:      8b 1d d1 90 04 08   mov     0x80490d1,%ebx
804809c:      b8 01 00 00 00     mov     $0x1,%eax
80480a1:      cd 80              int     $0x80
80480a3:      90                nop
80480a4:      90                nop
80480a5:      90                nop
80480a6:      90                nop
80480a7:      90                nop
80480a8:      90                nop
80480a9:      90                nop
80480aa:      90                nop
80480ab:      90                nop
80480ac:      90                nop
80480ad:      90                nop
80480ae:      90                nop
80480af:      90                nop
```

```
080480b0 <add_these>:
```

```
80480b0:      c7 05 d1 90 04 08 00  movl   $0x0,0x80490d1
80480b7:      00 00 00
80480ba:      01 05 d1 90 04 08   add    %eax,0x80490d1
80480c0:      01 1d d1 90 04 08   add    %ebx,0x80490d1
80480c6:      01 0d d1 90 04 08   add    %ecx,0x80490d1
80480cc:      c3                ret
```

```
linux3%
```

```
linux3% objdump -s a.out
```

```
a.out:      file format elf32-i386
```

```
Contents of section .text:
```

```
8048080 a1d19004 088b1dde 9004088b 0dd59004 .....
8048090 08e81a00 00008b1d d1900408 b8010000 .....
80480a0 00cd8090 90909090 90909090 90909090 .....
80480b0 c705d190 04080000 00000105 d1900408 .....
80480c0 011dd190 0408010d d1900408 c3 .....
```

```
Contents of section .data:
```

```
80490d0 12170000 00420000 00000000 07000300 .....B.....
80490e0 00000200 0000 .....
```

```
Contents of section .bss:
```

```
80490e6 0000 ..
```

```
Contents of section .comment:
```

```
0000 00546865 204e6574 77696465 20417373 .The Netwide Ass
0010 656d626c 65722030 2e393800 00546865 embler 0.98..The
0020 204e6574 77696465 20417373 656d626c Netwide Assembl
0030 65722030 2e393800 er 0.98.
```

```
linux3% exit
```

```
; File: sep3.asm
;
; File 3 for separate compilation example
```

```
extern _start, add_these
```

```
section .data
```

```
lvar1: dd 03h ; same name as other lvar1, OK
```

```
section .text
```

```
test3: ; no regs altered!
```

```
cmp [lvar1], dword 7
```

```
jne _start
```

```
jmp add_these
```

```
linuxserver1% nasm -f elf sep3.asm
```

```
linuxserver1% objdump -t sep3.o
```

```
sep3.o: file format elf32-i386
```

```
SYMBOL TABLE:
```

```
00000000 1 df *ABS* 00000000 sep3.asm
00000000 1 d *ABS* 00000000
00000000 1 d .data 00000000
00000000 1 d .text 00000000
00000000 1 .data 00000000 lvar1
00000000 1 .text 00000000 test3
00000000 *UND* 00000000 _start
00000000 *UND* 00000000 add_these
```

```
linuxserver1% ld sep1.o sep2.o sep3.o
```



```
linuxserver1% objdump -t a.out
```

```
a.out:          file format elf32-i386
```

```
SYMBOL TABLE:
```

```
08048080 l      d  .text  00000000
080490e8 l      d  .data  00000000
080490fc 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 sep1.asm
080490ec l          .data  00000000 lvar1
00000000 l      df *ABS*  00000000 sep2.asm
080490f4 l          .data  00000000 lvar1
00000000 l      df *ABS*  00000000 sep3.asm
080490f8 l          .data  00000000 lvar1
080480d0 l          .text  00000000 test3
080480b0 g          .text  00000000 add_these
08048080 g          .text  00000000 _start
080490f0 g          .data  00000000 gvar2
080490fc g          *ABS*  00000000 __bss_start
080490e8 g          .data  00000000 gvar1
080490fc g          *ABS*  00000000 _edata
080490fc g          *ABS*  00000000 _end
```

```
linuxserver1% objdump -d a.out
a.out:          file format elf32-i386
```

Disassembly of section .text:

08048080 <_start>:

```
8048080:      a1 e8 90 04 08      mov     0x80490e8,%eax
8048085:      8b 1d f0 90 04 08   mov     0x80490f0,%ebx
804808b:      8b 0d ec 90 04 08   mov     0x80490ec,%ecx
8048091:      e8 1a 00 00 00     call   80480b0 <add_these>
8048096:      8b 1d e8 90 04 08   mov     0x80490e8,%ebx
804809c:      b8 01 00 00 00     mov     $0x1,%eax
80480a1:      cd 80              int     $0x80
```

080480b0 <add_these>:

```
80480b0:      c7 05 e8 90 04 08 00  movl   $0x0,0x80490e8
80480b7:      00 00 00
80480ba:      01 05 e8 90 04 08   add    %eax,0x80490e8
80480c0:      01 1d e8 90 04 08   add    %ebx,0x80490e8
80480c6:      01 0d e8 90 04 08   add    %ecx,0x80490e8
80480cc:      c3                ret
```

080480d0 <test3>:

```
80480d0:      81 3d f8 90 04 08 07  cmpl   $0x7,0x80490f8
80480d7:      00 00 00
80480da:      0f 85 a0 ff ff ff   jne    8048080 <_start>
80480e0:      e9 cb ff ff ff     jmp    80480b0 <add_these>
```

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/>>
- **Machine code slides courtesy of Prof. Younis.**