Introduction to x86 Assembly

Ted Obuchowicz

December 30, 2020

Contents

1 Introduction

The following 32 ones and zeros represent an Intel x86 computer program instruction:

```
0110 0110 1011 1000 0000 0101 0000 0000
```

It is most likely that the reader does not know what instruction these bits represent. Computers are good at ones and zeros, humans not so much. The ones and zeros represent the particular instruction at the most primitive level known as *binary machine code*. The same instruction can be representedin a more compact version of machine code, hexadecimal machine code where a group of 4 bits is represented by single hexadecimal (base-16) digit:

```
6 6 B 8 0 5 0 0
```

In hexadecimal notation, we use the digits 0 through 9 and then the letters A to F. It is still likely the case that the reader has no clue as to what intruction these 8 hex digits represents.

The instruction expressed in a more human readable form is:

```
mov ax,5
```

This form is called "assembly language". Humans are better at understanding assembly language than machine code. The instruction moves (or loads) the value `5` into the processor register called `AX`.

Every processor has its own "flavor" of assembly language. The number and names of the registers varies with processor to processor. The number and type of assembly language instructions also depends on the type of processor. Here are the Motorola 68000 and ARM processor equivalent instructions to move the data `5` into a CPU register:

```
move #5,d0
```
and
```
mov r1,#5
```

In the early days of computing, programming was performed at the machine code level by physically connecting wires either to power or ground (a connection to the positive power supply represented a "1" and a ground connection represented a "0"). This was done for all the instructions in the program. This was a most tedious and error-prone procedure.

A program called an *assembler* is used to convert assembly language into machine code. This document is a brief introduction to Intel x86 assembly language programming. It is not a complete reference, refer to a textbook on Intel x86 assembly language for more details.

2 Instruction Format

Intel assembly language insructions are (generally) written in the form:

```
opcode destination_operand, source_operand
```

The opcode is a *mnemonic* (memory aid) which uses language0-like words to represent the particular instruction. Sometimes, an abbreviate spelling is used. The historical reason being to save data tranmission time when sending the program over a slow teletype 300 bits per second connection.

The destination and source operands may either be CPU *registers* or some location in main memory. Some instructions restrict an operand to be a CPU register and other instructions may involve some implicit CPU register which is loaded with the required data by a preceding instruction.

3 Intel x86 Processor Organization

A typical processor contains digital logic hardware which is capable of performing arithmetic and logical operations on binary data. Data is typically found in the (off-processor) memory. Registers within the processor are used to store the data to be operated upon. All processors contain instructions used to transfer data between the processor and main memory.

The early Intel processors (8086/8088) processor were so called *16-bit* processors since the CPU registers were capable of holding 16 bits of information. The registers visible to the programmer, or allowed to be specified as a source/destination operand in an assembly language instruction, are:

- AX, the "accumulator"
- BX, the "base index"
- CX, the "count" register
- DX, the "data" register
- BP, the "base pointer" register
- DI, the "destination index" register
- SI, the "source index" register

Of these 7 general purpose registers, the first 4 (`AX`, `BX`, `CX`, `DX`) are subdivided into two 8 bit halves:

Register	Upper Half	Lower Half
`AX`	`AH`	`AL`
`BX`	`BH`	`BL`
`CX`	`CH`	`CL`
`DX`	`DH`	`DL`

The H half holds the high order 8 bits (bits 15 down to bit 8), while the L part of the half holds the low order eight bits (bits 7 down to 0). Moving data into a particular half of one of these registers does not affect the data currently stored in the other half.

For example;

```
mov ax, 0x1234 ; AH will hold 0x12
               ; AL will hold 0x34
mov al, 0x56   ; AL will now hold 0x56
               ; and AH still holds the
               ; original 0x12
               ; the 0x prefix designates that the
               ; number is in base-16 (hexadecimal)
               ; representation
```

We have introduced the use of comments in the above code snippets. In Intel x86 assembly language a comment is commenced with the `;` character, everything following the semicolon is considered to be a comment and will be ignored by the assembler. Note also the use of the `0x` prefix to signify that hexadecimal is used to represent a number (the default base in the absence of any base specifier prefix is decimal).

The `BP`, `DI`, and `SI` registers cannot be accessed in 8-bit halves, they can only be used to hold 16 bit data.

Some of these general purpose registers also have special purposes. For example, the `AX` register is used by some arithmetic instructions to hold one of the operands, only the other operand need be specified by these instructions as the "other" operand is implied to be the `AX` register. The `CX` register is used together with a special instruction to conveniently control the number of times a loop is executed - hence its name "the count" register.

When Intel developed the 32-bit Pentium (80386) family of processors, they maintained backwards compatibility with these existing registers and simply "added" another 16 bits to each and named them as:

- `EAX`
- `EBX`
- `ECX`
- `EDX`
- `EBP`
- `EDI`
- `ESI`

A programmer can now access either the complete 32 bits of a register:

```
mov eax, 0x12345678
```

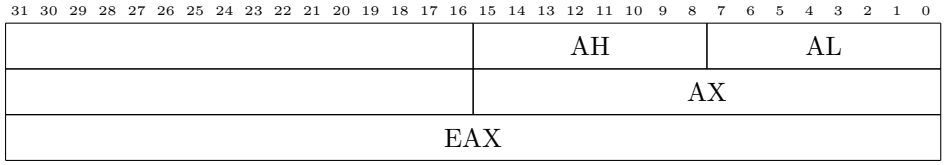
or access the low-order 16 bits:

```
mov ax, 0x1234
```

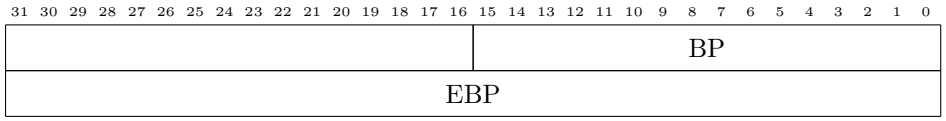
or in the case of `EAX`, `EBX`, `ECX`, `EDX` access a particular "half" of the low-order 16 bits:

```
mov al, 0xac mov ah,0xdc
```

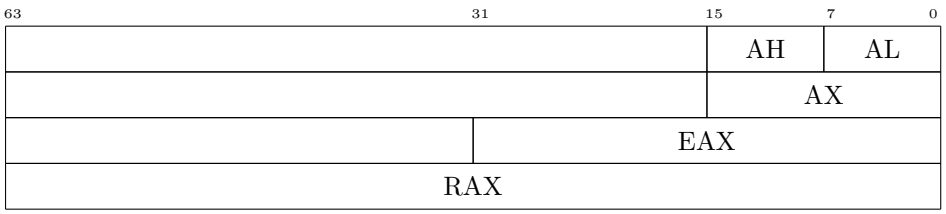
We can represent each "portion" of the registers by specifying the bit numbers with bit 0 appearing in the least significant left most position, and bit 31 in the most significant right most position.



The `EBX`, `ECX`, and `EDX` registers are subdivided in a similar fashion. The `EBP`, `EDI` and `ESI` registers are available in only the "full" 32-bit versions or the low-order 16 bit portions:



In 2003, AMD introduced its Opteron processor and with it, the AMD64 instruction set extension. This extension would become the defacto 64-bit extension for x86 as we know today. AMD64 is also known as `x64`. In a simplified view, this expanded the registers to 64 bit sizes, allowing them the prefix "R". With this in mind, the 64 bit version of `AX` is known as `RAX` and so on. The subdivisions mentioned previously are applicable to their 64-bit variants. As an example, the least significant 8 bits of `RAX` can be referenced as an operand by using `AL` still. **For the rest of the text, the 32-bit version will be used.** The following diagram shows the subdivision of a 64 bit register:



3.1 Special Purpose Registers

In addition to the general purpose registers, the Intel x86 processor contain registers which are used by the processor for very specific purposes. These are:

- Intruction Pointer, 16 bit version (`IP`)
- Stack Pointer, 16 bit version (`SP`)

The 80386 and above processors have the "expanded" 32-bit versions (`EIP` and `ESP`) and eventually, x64 processors have their versions (`RIP` and `RSP`).

The Instruction Pointer register is used to point to the address in main memory of the next instruction to be executed. Assembly language instructions and the data to be operated on are stored in main memory. As a program executes, instructions are *fetched* from main memory and stored in the processor. The hardware within the processor automatically adjusts the `IP` register so that it points to the next instruction in main memory as each instruction is executed. The Instruction Pointer register can also be loaded with a new address by certain specific instructions (i.e. a subroutine call, a return from a subroutine, a conditional branch instruction and so on). The Instruction Pointer is not considered a "programmer- visible register" as it cannot be specified as an operand in a typical instruction such as a `mov`. Attempting to use mov with the `IP` (or `EIP`, `RIP`) register such as:

```
mov eip,0 ; will this assemble?
```

Will result in the assembler reporting an error :

```
move_eip.asm:20:  error:  symbol 'eip' undefined
```

The stack pointer (`SP`, `ESP`, `RSP`) is used in instructions which manipulate the stack which is a reserved portion of main memory. It is usually manipulated automatically by certain instructions (`push`, `pop`). Unlike the instruction pointer, the stack pointer may be specified as an operand in an instruction (without causing an assemle-time error), but in general this is not a good thing to do (as it can cause catastrophic program execution). The following instruction will assemble:

```
mov esp,0          ; will this assemble?
                   ; yes, but in general we do not
                   ; manipulate the stack pointer directly
                   ; in such a manner.
```

3.2 Flags Register

Consider a hypothetical processor which contains 4-bit registers. If this processor were to add (in binary) the following 4-bit unsigned binary numbers:

	1	0	0	1	(9 in decimal)
+	1	0	0	0	(8 in decimal)
1	0	0	0	1	(17 in decimal)

Since the registers are only 4 bits wide, the register which is used internally by the processor to hold the sum stores only the 4 least significant bits of the sum and the carry bit is "lost". Specific bits with in a register can be used by the processor to "keep track" of losing a carry and other conditions such a result of zero from an arithmetic operation or a negative value. The processor updates this flag register after execution of certain instructions (usually the arithmetic/logic instructions).

The most frequently used flags and their bit positions within the `EFLAGS` for 32-bit x86 and, `RFLAGS` for x64 register are:

- C(arry), bit 0

  Used to indicate whether a carry resulted from an addition, or a borrow from a subraction. `C = 1` if a carry (or borrow) resulted, `C = 0` if no carry (no borrow) resulted.

- Z(ero), bit 6

  Used to indicate whether an arithmetic/logic instruction resulted in an answer of zero. `Z = 1` if result was zero, `Z = 0` if result was non-zero.

- S(ign), bit 7

  Used to indicate the sign (in two's complement notation) of the result of an arithmetic/logic instruction. `S = 1` means a negative result, `S = 0` indicates a positive result.

- O(verflow), bit 11

  Used to indicate overflow when signed numbers are added/subtracted. `O = 1` indicates the result was beyond the range of representable integers, `O = 0` indicates the result was within the range of representable integers. For example, with 8-bit two's complement the range of integers is from -128 to +127. If we were to add +1 to +127 :

	0	1	1	1	1	1	1	1	(+127)
+	0	0	0	0	0	0	0	1	(+1)
0	1	0	0	0	0	0	0	0	(+127)

  The result is beyond the range of 8-bit two's complment representation so the Overflow bit would be set in the flags register. The result of 1000 0000 would be stored in the CPU register holding the result, but the value is meaningless as the Overflow bit has been set.

  Similarly, if we were to add -1 to -128 :

	1	1	1	1	1	1	1	1	(-1)
+	1	0	0	0	0	0	0	0	(-128)
	1	0	1	1	1	1	1	1	1	(+127)

As the true result of -129 is beyond the range of representable numbers, the Overflow bit would be set (and the 0111 1111 result stored in the register holding the result).

3.3 Segment Register

The Intel x86 processors also contain several 16-bit "segment registers" (`CS`,`DS`, `SS`, `ES`, `FS`, and `GS`) which are used to generate complete memory addresses (in conjunction with other registers such as `EAX`, `EBX` etc). These registers are a hold-over from the original 8086/8088 processors which used a complicated manner of generating 20 bit main memory addresses from a pair of two 16 bit registers. We will not concern ourselves with the details of the use of these segment registers (they are not programmer-visible). They have been mentioned only for sake of completeness.

4 Data Representation in Intel x86

The Intel x86 family of processors store integers and perform arithmetic operations on integer data using two's complement represensetation. Characters are represented by their ASCII encoding.

The 80386 (and higher) processors can operate on byte (8 bits), word (2 bytes) and doubleword (4 bytes) integer data depending on whether the data is stored in a "half register" (eg. `AL`, `BL`, etc), a 16 bit register (`AX`, `BX`), or a full 32 bit register (`EAX`, `EBX`). In the case of x64, the option for a 64-bit register such as `RAX` or `RBX` is available to store a quad word of 64 bits.

5 A Complete NASM Assembly Language Program

Every programming language has its own unique syntax - the set of rules which specify how to properly write a program. Every assembler also has its own particular syntax. In this course, we will be using the Netwide Assembler (NASM), a free Linux assembler for the Intel x86 instruction set.

Consider the following C++ program:

```
// T. Obuchowicz
// June 25, 2020

#include <iostream>
using namespace std;
```

```
int mick = 3 ;  // declare and intialize an integer variable
int keith = 2 ; // declare and initialize an integer variable
int ron ;       // declare an integer variable

int main()
{
 ron = mick + keith ;

 return 0;
}
```

The Intel x86 assembly language "equivalent" program is:

```
; Ted Obuchowicz
; hello_world.asm
; June 25, 2020

section .data

; put your data in this section using
; db (define byte) , dw (define word) , dd (define double word)
; assembler directives

mick  db 3
keith db 2


section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5  )
;     resw (reserve word)    or
;     resd (reserve double)

ron resb 1


section .text
        global _start

_start:
        mov al, [mick] ; load data found in memory at address mick
                       ;  into register al.
        mov bl, [keith]; load data found in memory at address keith
                       ; into register bl
```

```
add al,bl       ; al = al + bl
mov [ron],al    ; store contents of register al back into
                ; memory at address ron


mov eax,1             ; The system call for exit (sys_exit)
mov ebx,0             ; Exit with return code of 0 (no error)
int 80h               ; control returned to Linux prompt
```

The assembly source code consists of three portions. The `.data` section is used to define locations in main memory and intialize these locations with some values. The assembler directives `db`, `dw`, and `dd` are used to define a byte, word, and double word. The labels `mick` and `keith` will be associated with the main memory addresses storing the two values.

The `.bss` (block segment symbol) section is used to simply reserve a specified number of memory locations and to associate the address of the reserved memory with the specified label. *Note that these assembler directives do not produce any executable machine code.*

The `.text` section is where the instructions composing the program are written. The special label `_start` is used to define the entry point of the program. When the program is loaded into main memory, program execution will start with the `mov al, [mick]` instruction.

Let us examine this instruction in greater detail. The use of the `[ ]` around the label `mick` indicates to the assembler that the destination operand of the `mov` instruction is the contents of the byte of main memory found at address `mick` (the data `3`). Thus, the value 3 will be loaded into the `AL` register.

Similarly, the `mov bl, [keith]` instruction will load into the `BL` register the byte-sized data found in main memory at address associated with the label `keith`. Thus, the value 2 will be loaded into register `BL`.

This association of physical main memory addresses with programmer chosen names is performed automatically when the program is loaded into memory, more details on this will be given at a later point.

The `add al, bl` instruction adds the current contents of register `BL` (2), to the current contents of register `AL` (3), and stores the resulting sum of 5 into the `AL` register. The Intel x86 is set to be a "two-address" instruction format as the arithmetic type of instructions are of the form:

```
operation source2 , source1
```

with the understanding that this performs:

```
source2 = source2 operation source1
```

In other words, the `source2` operand acts initially as one of the operands to the arithemetic operation and is eventually the destination of where the result is to be stored (overwriting the initial value).

Some microprocessors (the ARM family) allow for "three-address" type of instructions in which all three operands (destination, source1, and source2) are specified:

```
add r3, r2, r1
```

The instruction `mov [ron], al` is used to store the resulting sum found in register `AL` into the memory location associated with label `ron` (which was defined in the `.bss` secion). Since the initial contents of this memory location was immaterial, we simply reserved 1 byte without any concern with its contents.

The last three lines of the program:

```
mov eax,1               ; The system call for exit (sys_exit)
mov ebx,0               ; Exit with return code of 0 (no error)
int 80h                 ; control returned to Linux prompt
```

are used to return "gracefully" to the Linux operating system when the program is run from the Linux command prompt. If the program did not contain these three instructions, the processor would continue continue to fetch and execute instructions after the `mov [ron], al` instruction. Since there it is not known what is stored in main memory after this instruction, the program would behave in an unpredictable manner. Here is a sample run of the program, where the last three lines have been intentionally removed:

```
ted@localhost NASM]$ ./hello_world_bad
Segmentation fault (core dumped)
```

The infamous Linux "Segmentation fault (core dumped)" error message was produced as a consequence of omitting the last three lines. Since every NASM program will follow the above general format, it is useful to create an ASCII text file (save it with filename `template.asm`) with the following contents:

```
; template file for a NASM assembly language program

section .data

; put your data in this section using
; db , dw, dd directions


section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5
; sum2 resw 5 ( reserve 5 words )
; sum3 resd 5 ( reserve 5 double words)

section .text
        global _start

_start:
        ; put your code here.THIS IS THE FIRST ASSEMBLY LANGUAGE INSTRUCTION
```

```
keith:  ; THIS IS THE SECOND ASSEMBLY LANGUAGE INSTRUCTION
        ;
        ; THE REST OF THE PROGRAM IS ENTERED HERE

        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

6 Addressing Modes

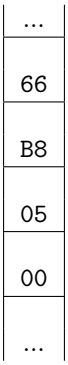
The previous code examples have introduced the concept of addressing modes. Addressing modes determine how an instruction specifies how the data to be operated upon is to be accessed. The four basic modes are: immediate, register, direct, register indirect.

6.1 Immediate Mode

In immediate mode, the data is contained within the instruction:

```
mov ax,5 ;  load the immediate data into register ax
```

The hex machine code for this instruction would be stored in memory as:



The instruction occupies four bytes of main memory and the numeric data `0005` is stored as part of the instruction. Intel x86 processors store multi-byte numbers in a "backwards" manner, more on this later.

6.2 Register Mode

The data is contained within the specified register:

```
mov al,bl ;
mov ebx,eax ;
```

6.3 Absolute Mode

The data is found in memory at the address associated with the given label in the instruction. Recalling the earlier program which added the contents of two main memory locations:

```
section .data
mick  db 3
keith db 2


_start:
        mov al,[mick];
        mov bl,[keith];
```

The labels `mick` and `keith` would be associated with the memory addresses holding the data `3` and `2` respectively. Let us assume these addresses are `1000` and `1001` as indicated:

Memory	Address
	...
3	1000
2	1001

...

In the instruction `mov al, [mick]`, the destination operand (register `AL`) is specified with the register addressing mode and the source operand is specified the direct (sometimes called *absolute*) addressing mode. The 32 bits of the address associated with label mick would be stored as part of the insruction. Once the instruction has been fetched into the processor, memory would have to be accessed again to fetch the desired operand at address `1000`.

The following constrast can be made about direct mode with the following:

```
mov eax,mick ; load register eax with immediate data mick
```

As the comment indicates, the source operand is specified with immediate mode. The memory address associated with label `mick` would the the immediate data stored as part of the instruction. Register `EAX` would be loaded with the 32 bits of the address associated with label `mick`.

6.4 Register Indirect Mode

In register indirect mode, a processor register contains the address of where the data is found. For example, we could, as in the earlier example, first perform:

```
mov eax, mick
```

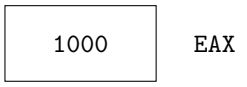
to load register `EAX` with the address `mick`.

Subsequently, the data found at address `mick` (i.e the number 3), could be loaded into a register with the following instruction:

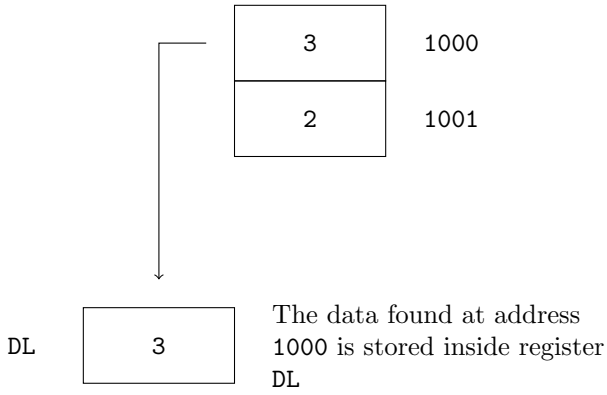
```
mov dl, [eax]
```

Here is a figure explaining the concept of register indirect (we assume label `mick` is associated with address `1000`):

```
    mov eax,mick ; loads register eax with the
                 ; with the address mick
```




```
mov dl, [eax] ; means use the address contained
              ; in register eax as a 'pointer'
              ; to where the data is found in main
              ; memory, go to that address, read the
              ; number that is stored there and load
              ; it into the  specified destination (dl) register
```



It is essential to fully understand the concept of the various addressing modes. Here is a program which illustates the basic modes and also gives more examples of the various assembler directives (`db`, `dw`, `dd`, `resb`). The reader is well advised to read and understand the program:

```
; T. Obuchowicz
; addressing_modes_new.asm

section .data

; put your data in this section using
; db , dw, dd directions

; examples of db (DEFINE BYTE assembler directive )

num1 db -5 ; decimal number -5
num2 db 0xa2 ; hexadecimal
num3 db 0a2h ; alternative form of specifying hexadecimal notation
num4 db $0a2 ; hex yet again, the leading 0 IS REQUIRED
num5 db 10100010b ; same number specified in binary

; examples of dw (DEFINE WORD assembler directive)

num6 dw 321 ; decimal 321 as a WORD of memory = 2 BYTES
num7 dw 0x2ab6 ; how will this word be stored in memory?
               ; which byte is stored first?

num8 dw 0010101010110110b ; same number as above but in binary

; examples of dd (DEFINE DOUBLEWORD assembler directive)

num9 dd 0x12345678  ; 4 bytes of memory required to store a double word

; we can even define ASCII characters and ASCII character strings

letter db 'a'  ; ASCII code for 'a' will be stored in memory location
               ; letter
more_letters db 'b','c','d','e'  ; if more than one letter,
                                 ; separate with commas
keith1 db 'keith' ; a character string is enclosed in single quotes
mick db 'mick'    ; another string

section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5


num10 resb 10 ; reserve 10 bytes
num11 resw 2 ; reserve 2 words
```

```
section .text
        global _start

_start:
  ; put your code here
        mov eax, num1 ;  IMMEDIATE

; register eax is loaded with the address of memory location
; symbolically referred to by 'num1'
; the loader will convert the symbol 'num1' into some physical
; memory address at load  time.

keith:        mov ebx, 0x0000

; another form of IMMEDIATE addressing this time using
; a numeric constant specified in hexadecimal as opposed
; to a symbolic label num1 in the above instruction

mov ecx, ebx    ; REGISTER addressing

; the  source data is found in the register ebx and
; the destination of the mov instruction is register ecx

mov bl, [eax] ;  REGISTER INDIRECT addressing

;  means go to the memory location
;  pointed to by the contents of eax  (which is num1), read the
;  data found at this address and load it into register bl
;  bl gets loaded with number -5

mov cl, [num1] ;  DIRECT or ABSOLUTE addressing

;  cl will be loaded with the 1 byte of data found at memory
;  location num1, i.e. with the number -5



mov eax,1       ; The system call for exit (sys_exit)
mov ebx,0       ; Exit with return code of 0 (no error)
int 80h
```

7 Instruction Set Summary

There are 4 main categories of instructions within any assembly language:

1. Data movement intructions
2. Arithmetic/logic instructions
3. Flow control (branching/subroutines)
4. Input/output

A summary of the most commonly used Intel x86 instructions in the first three groups is presented in the following subsections.

7.1 Data Movement

The basic format for a `mov` insruction is:

```
mov destination_operand, source_operand
```

Data may be transferred:

- From register to another register
- An immediate data to a register
- From a memory location to a register
- From a register to a memory location.

The Intel x86 instruction set does not allow a memory to memory data transfer. The following program gives examples of different valid combinations of the `mov` instruction and intentionally includes three invalid combinations which cause assembly time errors:

```
; Ted Obuchowicz
; June 26, 2020
; data_movs.asm

section .data

; put your data in this section using
; db , dw, dd directions
mick db 5


section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5
```

```
ron resb 1

section .text
        global _start

_start:
        mov ax,5          ; load immediate data into register ax
                          ; ax register is loaded with 0005

        mov bl,[mick]     ; from memory to register
                          ; bl is loaded with 5

        mov [ron],bl      ; from register to main memory
                          ; 5 is stored in memory at location ron

        mov [ron],[mick]  ; not allowed will cause assembly error

        ; the following are non-sensical and will cause assembly errors

        mov 5, cl         ; the destination cannot be immediate data
        mov 6,5           ; this too is incorrect for the same reason



        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

Self-Study Question:
How would one achieve a memory-to-memory data transfer?
Hint: Two separate `mov` *instructions are required.*

7.2 Arithmetic Instructions

7.2.1 The ADD Instruction

The possible combinations of destination and source operands for the `add` instruction are:

- `add destination_register, source_register`
- `add destination_memory, source_register`
- `add destination_register, source_memory`
- `add destination_register, immediate_data_source`
- `add destination_memory, immediate_data_source`

As with the move, both the destination and source operands cannot be a memory location at the same time.

```
; Ted Obuchowicz
; June 26, 2020
; adds.asm

section .data

; put your data in this section using
; db , dw, dd directions
mick  db 3
keith db 2


section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5


section .text
        global _start

_start:
        mov al,1
        mov bl,1
        add al,bl ; al = al + bl ( al has 2 after the add)

        add al,[mick] ; al = al +  3
                      ; al has 5 after the add
        add [keith],al ; memory location keith now
                       ; contains  2 + 5 = 7
    ;   add [keith],[mick] ; will cause an error

        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

Caveat: When performing register to register addition, the sizes of the registers must both be the same (ie. byte, word, doubleword) or else an assemble time error will occur. For example:

```
        mov ax,1
        mov bl,1
        add ax,bl ; will not assemble since operands are
                  ; not the same size
```

NASM will report the following error:

```
adds2.asm:25: error: invalid combination of opcode and operands
```

This situation is similar to the proverbial "comparing apples and oranges". With a little thought (and understanding of how binary addition is performed), one understands why the an attempt to add the contents of an 8-bit register to that of a 16-bit register makes no sense:

```
      AX  =  0000  0000  0000  0001
+     BX  =  ????  ????  0000  0001
```

The addition cannot be performed since the bits represented with the "?" do not have any value (since they do not exist within an 8-bit register), hence the NASM assembler generates an error.

One has to be aware of the sizes of each operand as there can sometimes be unexpected results. Consider the following situation, where one of the operands to an `add` instruction is a memory location and the other is a 16-bit register. However, through some kind of oversight, the programmer defined only a single byte with a `db` directive:

```
; Ted Obuchowicz
; June 26, 2020
; add3s.asm

section .data

; put your data in this section using
; db , dw, dd directions
mick  db 3    ; intentionally define only a single byte
              ; some random values will exist in memory
              ; in the subsequent bytes following mick

section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5


section .text
        global _start

_start:
        add ax,[mick]   ; will add the 2 bytes in memory at
                        ; locations mick and mick+1 to the
                        ; contents of 16 bit ax register
                        ; so one has to be aware of the the
                        ; sizes of the operands
```

```
        mov eax,1               ; The system call for exit (sys_exit)
        mov ebx,0               ; Exit with return code of 0 (no error)
        int 80h
```

Since the one of the operands to the add is the 16-bit `AX` register, the 2 bytes of memory at addresses `mick` and `mick + 1` will form the second operand. However, only 1 byte of memory data was actually defined with a known value (that at address mick with value of 3, the subsequent byte has not specified value). Since both operands of the `add` instruction must be of the same size, the assembler was able to determine the size to be 2 bytes (by the destination register `AX`). In cases similar to this, the programmer can explicity tell the assembler that the `[mick]` is referring to two bytes (a word) of data with the use of the `word` modifier:

```
 add ax, word [mick]   ; explicitly specify that the source
                       ; operand should be the word starting
                       ; at location mick (even though the assembler
                       ; will determine this on its own from the fact
                       ; that the register operand is 16 bits
```

Self-Study Question:

What will be stored in the `EAX` register after the `add` instruction has been executed in the following program:

```
; Ted Obuchowicz
; June 26, 2020
; adds4.asm

section .data

; put your data in this section using
; db , dw, dd directions
mick  db 3    ; intentionally define only a single byte
              ; some random values will exist in memory
              ; in the subsequent bytes following mick

section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5


section .text
        global _start
```

```
_start:
        add eax,mick ;  what will the contents of eax be
                     ;  after the add instruction has been
                     ;  executed ????


        mov eax,1            ; The system call for exit (sys_exit)
        mov ebx,0            ; Exit with return code of 0 (no error)
        int 80h
```

7.2.2 The INC Instruction

The `inc` instruction is used to to add 1 to either a register or a memory location. It is worth asking why there is a special instruction to add 1 to destination operand when the same can be performed with a simple add instruction with an immediate operand of 1:

```
add ax,1  ; will add 1 to contents of ax register
inc ax    ; will also add 1 to contents of ax register
```

The machine code for the two instructions quite clearly shows the difference between the two instructions:

```
00000000 6683C001    add ax,1
00000004 6640        inc ax
```

The machine code for the `add ax, 1` instruction occupies 4 bytes of memory, whereas the `inc ax` instruction occupies only 2 bytes of machine code.

7.2.3 The Add With Carry Instruction

An addition of either a byte, word, or doubleword operands may result in a carry being generated from the most significant bit position. The `adc` (add with carry instruction) performs addition by adding the value of the carry bit in the flags register. In this manner, addition of operands larger than what would fit inside a CPU register can be performed. The multiple operands may reside in memory and can be "added with carry" from the least significant operands towards the most siginificant operands:

	$\texttt{carry}_{n-1}$	$\texttt{carry}_{n-2}$	...	$\texttt{carry}_0$	
	$\texttt{addened}_n$	$\texttt{addened}_{n-1}$	...	$\texttt{addened}_1$	$\texttt{addened}_0$
+	$\texttt{augend}_n$	$\texttt{augend}_{n-1}$	...	$\texttt{augend}_1$	$\texttt{augend}_0$
$\texttt{carry}_n$	$\texttt{sum}_n$	$\texttt{sum}_{n-1}$	...	$\texttt{sum}_1$	$\texttt{sum}_0$

The value of $\texttt{sum}_0$ is obtained with an:

$$\texttt{add addend}_0\texttt{, augend}_0$$

The value of $\texttt{sum}_1$ (and subsequent **sums**) is obtained by using the "add with carry" form of the **add** such that any carry generated by the first addition is added together with $\texttt{addend}_1$ and $\texttt{augend}_1$:

```
adc addend_1, augend_1 ; addend_1 = addend_1 + augend_1 + carry flag
```

Here is a example program which performs the hexadecimal addition of:

```
    DEADBEEFC0CAACDC
+   ACDCC0CABEEFDEAD
-------------------

1   8B8A7FBA7FBA8B89
```

Note: Both the addend and the augend (fancy names for the two operands in an addition operation) are 64-bit numbers. They are too large to fit into 32-bit registers.

```
; Ted Obuchowicz
; July 2, 2020

section .data

; put your data in this section using
; db , dw, dd directions
addend dd 0xdeadbeef, 0xc0caacdc
augend dd 0xacdcc0ca, 0xbeefdead

section .bss

; put UNINITIALIZED data here using

sum resd 3 ;  reserve 3 double words to hold the sum (and a
           ;  possible carry out
           ;  in the order  carry   high_sum   low_sum


section .text
        global _start

_start:
        mov eax,[addend + 4] ; get the low order addend
        mov ebx,[augend + 4] ; get the low order augend
        add eax, ebx         ; add the two low order operands
        mov [sum+8],eax      ; store low order sum
        mov eax, [addend]    ; get the high order addend
        mov ebx, [augend]    ; get the high order augend
        adc eax,ebx          ; add the two high order operands together
```

```
                        ; with any carry generated from the
                        ; addition of the low order operands
    mov [sum+4],eax     ; store the high order sum
    mov eax,0           ; prepare to add the last carry to 0
    adc eax,eax         ; eax = 0 + 0 + carry
                        ; so that the
    mov [sum],eax       ; resulting carry  may be stored in memory


    mov eax,1           ; The system call for exit (sys_exit)
    mov ebx,0           ; Exit with return code of 0 (no error)
    int 80h
```

7.2.4 The Subtraction Instruction

The assembly language insructions for subtraction are very similar to those for addition:

```
sub destination, source ; destination = destination - source

sbb destination, source ; destination =
                        ; destination
                        ; - source
                        ; - borrow

dec destination         ; destination =
                        ; destination - 1
```

The restrictions on the allowable addressing modes for the destination and source operands are the same as those for the `add` instruction.

7.2.5 Multiplication Instructions

There are two instructions for both unsigned and signed multiplication. They are:

- `mul` - Unsigned multiplication
- `imul` - Signed multiplication

Operands can be byte (8 bits), word (16 bits), doubleword (32 bits) or quadword (64 bits). Multiplying two 8-bit operands results in a 16-bit product, two-16 bit operands result in a 32-bit product, and the multiplication of two 32-bit operands yields a 64-bit product (which is stored in a pair of 32 bit registers). Lastly, the multiplication of two 64-bit operands yields a 128-bit result, stored in two 64-bit registers.

Multiplication is defined as:

	`mutiplicand`	(`m` bits wide)
×	`multiplier`	(`n` bits wide)
	`product`	(at most `m+k` bits wide)

The multiplicand is stored in either the `AL` register (8-bit case) or the `AX` register (16-bit case), the `EAX` register (32-bit case) or lastly, `RAX` for the 64-bit case. The multiplier can be in a register or a memory location. The product is stored either in the `AX` register (16-bit product resulting from two 8-bit operands), `DX:AX` register pair (32-bit product from two 16-bit 16-bit operands), `EDX:EAX` register pair (64-bit product from two 32-bit operands), or `RDX:RAX` (128-bit product from two 64-bit operands). The following program contains several examples of the `mul` and `imul` instructions with different sized operands:

```
; Ted Obuchowicz
; July 3, 2020
; mults.asm

section .data

; put your data in this section using
; db , dw, dd directions
multiplier1 db 3
multiplier2 dw 4
multiplier3 dd 5


section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5


section .text
        global _start

_start:
        mov al, 2           ; load 8 bit multiplicand
        mul byte [multiplier1] ;  ax = al x 3
                           ;  ax = 6

        mov al, 2           ; load another 8 bit multiplicand
        mov bl, -3          ; put mulitplier in a register
        imul bl            ; ax = al x 3
                           ; ax  will contain -6 in 2s complement
```

```
 ; 16 bit operands

 mov ax, 3          ; load 16 bit multiplicand
 mul word [multiplier2] ; dx:ax = ax x  4

; 32 bit operands

mov eax, 3          ; load 32 bit multiplicand
mov ebx, 2          ; load 32 bit multiplier
mov edx, -5         ; put some number in EDX to show
                    ; that EDX will get overwritten with the
                    ; high-order 32 bits of the product
mul ebx             ; edx:eax = eax x ebx

; 32 bit operands with mulitplier in a doubleword in memory

mov eax, 3
mul dword [multiplier3] ; edx:eax = eax x  15

   mov eax, 1            ; The system call for exit (sys_exit)
   mov ebx, 0            ; Exit with return code of 0 (no error)
   int 80h
```

The astute reader may have noticed the use of the byte, word, and dword specifiers in the three `mul` instructions in which the multiplier was stored in main memory:

```
mul byte  [multiplier1]
mul word  [multiplier2]
mul dword [multiplier3]
```

They are required as the assembler is not able to determine the size of the multiplier memory operand without their presence:

```
mul [multiplier1] ; is the memory operand a byte, word, or doubleword
                  ; it cannot be unambiguosly determined, hence the
                  ; the assembler will report an error of the form
                  ; error: operation size not specified
```

7.2.6 Division Instructions

Similar to multiplication, there are signed and unsigned division instructions. They are:

- `div` - Unsigned division
- `idiv` - Signed division

Integer division of a dividend by a divisor yields a integer quotient and an integer remainder:

$\frac{8}{3}$= Quotient of 2 and remainder of 2

Or as performed in grade school:

$$\begin{array}{r} 2 \\ 3\overline{)\,8} \\ \underline{6} \\ 2 \end{array}$$

Or, stated in other terms:

$$\begin{aligned} \text{8 (dividend)} &= \text{2 (quotient)} \times \text{3 (divisor)} + \text{2 (remainder)} \\ &= 6 + 2 \\ &= 8 \end{aligned}$$

The `div` instruction performs unsigned division, the `idiv` is used with the quotient can be positive or negative result and the remainder taking on the sign of the dividend:

$$\begin{aligned} \text{8 (dividend)} &= \text{-2 (quotient)} \times \text{-3 (divisor)} + \text{2 (remainder)} \\ &= 6 + 2 \\ &= 8 \end{aligned}$$

$$\begin{aligned} \text{-8 (dividend)} &= \text{-2 (quotient)} \times \text{3 (divisor)} + \text{-2 (remainder)} \\ &= -6 + (-2) \\ &= -8 \end{aligned}$$

The following table lists the sizes of operands are allowed, and the the CPU rdegisters to hold the operands and results of a divide operation:

Dividend	**Divisor**	**Quotient**	**Remainder**
16-bit `AX`	Any 8-bit reg or memory location	8-bit `AL`	8-bit `AH`
32-bit `DX:AX`	Any 16-bit reg or memory location	16-bit `AX`	16-bit `DX`
64-bit `EDX:EAX`	Any 32-bit reg or memory location	32-bit `EAX`	32-bit `EDX`
128-bit `RDX:RAX`	Any 64-bit reg or memory location	64-bit `RAX`	64-bit `RDX`

7.2.7 Sign Extension

A positive dividend can be extended into a larger number of bits by simply inserting 0s in front of the bits.

```
                0000   1000    (+8 as a 8-bit unsigned integer)
00000000        0000   1000    (+8 as a 16-bit unsigned integer)
```

This padding with 0s can easily be done with a `mov` instruction:

```
mov al,8
mov ah,0 ; now ah:ax consists of
         ; 0000 0000 0000 1000
```

For signed dividends, the most significant bit is the dividend will be a 1, this 1 must then be inserted in front of the bits of the original dividend to sign extend.

```
            1111   1000    (-8 as a 8-bit signed 2's complement integer)
11111111    1111   1000    (-8 as a 16-bit signed 2's complement integer)
```

Special instructions in the Intel x86 instruction set are used to sign extend a signed dividend into a larger (more bits) signed dividend:

```
cbw   ;  sign extend 8 bit number in al to 16 bit number in ax
      ;  called the 'convert byte to word' instruction

cwd   ;  sign extend 16 bit number in ax to 16 bit signed number stored
      ;  in the dx:ax register pair
      ; 'convert word to double'

 cdq  ; sign extend the 32 bit number in eax to a 64 bit signed number
      ; stored in the edx:eax register pair
      ; 'convert double word to quad word
```

The following program gives various examples of both unsigned and signed division with various sized operands:

```
; Ted Obuchowicz
; July 3, 2020
; divides.asm

section .data

; put your data in this section using
; db , dw, dd directions
divisor1 db 3
divisor2 dw 3
```

```
divisor3 dd 3


section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5


section .text
        global _start

_start:
        mov ax, 8 ; 16 bit dividend / 8 bit divisor
        mov bl, -3
        idiv bl ; quotient = -2 (AL)  and remainder = 2 (AH)
     ;  idiv byte [divisor1] ; example of memory divisor operand

        ; 32 bit dividend / 16 bit divisor
        mov dx, 0
        mov ax, 8 ; dx:ax holds 32 bit value of unsigned dividend
        mov bx, 3
        div bx  ; quotient = 2 (AX) remainder = 2 (DX)
    ;   div word [divisor2] ; example of word memory operand

       ; 32 bit signed dividend  / 16 bit divisor
       mov ax, -8
       mov bx, 3
       cwd   ; will sign extend the sign bit of ax register
             ; into the dx register thus forming the signed 32
             ; version of dividend in the dx:ax registers
       idiv bx ; quotient = -2 (AX) remainder = -2 (DX)

       ; 64 bit dividend / 32 bit divisor
       mov edx, 0
       mov eax, 8 ; edx:eax holds 64 bit unsigned dividend
       mov ebx, 3 ; 32 bit divisor
       div ebx   ; quotient = 2 (EAX) remainder = 2 (EDX)
    ;  div dword [divisor3] ; example of double word memory operand

      ; 64 bit signed dividend / 32 bit divisor
      mov eax, -8 ; 16 bit signed dividend in eax
      cdq         ; convert double word in eax to a signed quad word
                  ; stored in EDX:EAX registers
      mov ebx, 3  ; 32 bit divisor
      idiv ebx    ; quotient = -2 (EAX) remainder = -2 (EDX)
```

```
        mov eax, 1              ; The system call for exit (sys_exit)
        mov ebx, 0              ; Exit with return code of 0 (no error)
        int 80h
```

7.3 Logical Operations

The Intel x86 instruction set contains several Boolean logic operators. The following program illustrates typical use and also shows how to express immediate data in different bases (binary and hex). Operands may be either immediate data, registers, or memory locations.

```
; Ted Obuchowicz
; logical.asm
; July 6, 2020

section .data

; put your data in this section using
; db , dw, dd directions
mask db 00000001


section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5
; sum2 resw 5 ( reserve 5 words )
; sum3 resd 5 ( reserve 5 double words)

section .text
        global _start

_start:
        mov al, 01010101b ; load al with some random bits
        and al, 10101010b ; al = al and immediate data
                          ; al will be 00000000
        or al, [mask]     ; al = 00000000 or 00000001
                          ; al = 00000001
        not al            ; al = 11111110
        mov bl,11111111b
        xor al,bl         ; al = al exclusive-or bl
                          ; al = 00000001
```

```
        mov eax,1               ; The system call for exit (sys_exit)
        mov ebx,0               ; Exit with return code of 0 (no error)
        int 80h
```

There are additional instructions to test the value of a particular bit in an operand, and to shift bits either left or right.

7.4 Flow Control

All the sample assembly language programs exhibited what is commonly referred to as "straightline execution". Instructions are executed one after another in the order that they appear written as in the source code. In order to be able to write more sophisticated assembly language programs, the programmer needs to be able to alter, at will, the default straight line execution of instructions. The jump (unconditional and conditional) statements allow for flow control.

Consider the following C++ program which makes use of the controversial `goto` statement. The `goto` statement causes execution of the program to continue with the statement identified with a symbolic label:

```
// T. Obuchowicz
// July 6, 2020
// goto.C
// A C++ program illustrating the use of the imfamous goto statement

#include <iostream>
using namespace std;


int main()
{

 int sum = 0 ;

// example of the C++ goto statement which is
// used to create an (intentional) infinite loop
//
top_of_loop:    sum = sum + 1 ;
next_statement: goto top_of_loop ;  // in C++ any statement can have
                                    // a label

 return 0;
}
```

This program creates an infinite loop in which 1 is added to the variable `sum` (initialized to 0 outside of the loop).

The assembly language equivalent of this program is:

```
; Ted Obuchowicz
; July 6, 2020
; goto.asm
; an assembly language program need not always
; contain a section .data nor a section .bss


section .text
        global _start

_start:
              mov al, 0
top_of_loop:  add al,1            ; labels begin in column 1
              jmp top_of_loop     ; the next statement to be
                                  ; executed will be the add al,1

              ; these statements will not be executed as the
              ; jump statement changes the straight line execution

              mov eax,1            ; The system call for exit (sys_exit)
              mov ebx,0            ; Exit with return code of 0 (no error)
              int 80h
```

7.4.1 The Jump Instruction

The general form of the `jmp` (read as "jump" as in Jmping Jack Flash) instruction is:

```
jmp some_label
```

Where `some_label` is an identifier for a label placed somewhere within the source code. A `jmp` instruction will cause the program execution to continue with the statement identified by the label (rather than the next immediately proceeding instruction), thus altering the default straight line execution of code. The label `top_of_loop:` is placed in front of the instruction which corresponds to the target of the `jmp` statement. The placement of the label in column 1 (and the : after the label) are merely conventions. All the following will assemble properly:

```
_start:
              mov al, 0
top_of_loop   add al,1              ; labels need not have the :
              jmp top_of_loop       ; the next statement to be
                                    ; executed will be the add al,1
```

```
_start:
                mov al, 0
                top_of_loop  add al,1    ; labels  can be in any columns
                                         ; but this is a bit difficult to read
                jmp top_of_loop          ; the next statement to be
                                         ; executed will be the add al,1
```

However, if a label appears on a line by itself otherwise, some assemblers may return an error. `nasm` reports a warning but generates the correct machine code:

```
_start:
                mov al, 0
                top_of_loop
                add al,1                ; labels begin in column 1
                jmp top_of_loop        ; the next statement to be
                                        ; executed will be the add al,1
```

The warning generated by `nasm` from the use of the label on a separate line without the semicolon is:

```
goto3.asm:12: warning: label alone on a line without a colon might be in error
                       [-w+orphan-labels]
```

Adopt a convention for your use of lables and be consistent with it; this will result in easy to understand code.

7.4.2 Conditional Jumps

It is often required to alter the flow of control based upon some sort of condition:

- Is one number smaller than another?
- Is the answer of some sort of arithmetic operation equal to 0?
- Is the answer a negative number.

All computers have the ability to execute conditional jumps. Consider the simple task of funding the sum of

$$5 + 4 + 3 + 2 + 1 = 15$$

Using a loop approach, the pseudocode is:

```
   initialize ax register = 0 ; ax will be used to hold the running sum
   initialize bx register = 5 ; bx hold the number to be added to the sum

top: ax = ax + bx
     decrement bx register
     jump to top provided bx register is not equal to 0, otherwise continue
     store value of ax register in memory location
```

Using C++, the implementation is:

```
// T. Obuchowicz
// July 6, 2020
// sum_of_5.C

#include <iostream>
using namespace std;


int main()
{

int ax = 0;
int bx = 5;
int sum;

keith: ax = ax + bx;
       bx--;
       if (0 != bx)
       {
         goto keith;
       }
       // else continue with straightline code
       sum = ax;
       cout << sum << endl;

       // end of program

       return 0;
}
```

Here is the assembly language version:

```
; Ted Obuchowicz
; sum_of_5.asm
; July 6, 2020

section .data
```

```
; put your data in this section using
; db , dw, dd directions


section .bss

; put UNINITIALIZED data here using
sum resw 1

section .text
        global _start

_start:
        mov ax,0 ; initialize sum to 0
        mov bx,5 ; initialize bx to starting value
keith:  add ax,bx ; form the running sum
        dec bx    ; decrement bx by 1 and check if reached zero
        jne keith ; jump not equal to zero to top of loop and keep adding
        mov [sum],ax ; otherwise exit the loop and store result in memory

        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

The `jne keith` statement is the conditional jump to label `keith` (any name can be used for a label, `keith` was chosen to show there is nothing special about the label `top` used in the pseudocode). The jump to label `keith` is taken when the result of decrementing the `BX` register is non-zero. When the `BX` register reaches the value 0, the condition becomes false and program flow continues with the storing of the sum (the contents of the `AX` register) into memory.

The general form of a conditional jump instruction is:

`j(condition) some_label`

Where (condition) refers to a specific condition such as but not limited to:

Condition	Description
`jeq`	Jump if equal to zero
`jne`	Jump if not equal to zero
`jle`	Jump if less than or equal to zero
`jc`	Jump if carry flag is set
`jnc`	Jump if carry flag is not set

The microprocessor hardware examines certain flags in the flags register to determine whether the specified condition is true or false. This begs the question, does the order of statements in a loop "matter"?

When programming in a high-level language, the order of statements as they appear in a loop is sometimes not important, meaning that two statements can be interchanged without affecting the final outcome of the program. For example, we can re-write the sum program as:

```
// T. Obuchowicz
// July 6, 2020
// sum_of_5_order.C

#include <iostream>
using namespace std;


int main()
{

int ax = 0;
int bx = 6;
int sum;

keith: bx--; // we change the order in the loop so bx
             // starts from 6
       ax = ax + bx ; // however the final sum will be the same
       if (0 != bx)
       {
         goto keith;
       }
       // else continue with straightline code
       sum = ax;
       cout << sum << endl;

       // end of program

       return 0;
}
```

Both versions of the program give the same value for `sum`, being 15. Does the order of assembly language statements within a loop matter? Or, in other words, would the following version of the assembly language program still give 15 as the result?

```
; Ted Obuchowicz
; sum_of_5_order.asm
; July 6, 2020

section .data

; put your data in this section using
; db , dw, dd directions


section .bss

; put UNINITIALIZED data here using
sum resw 1

section .text
        global _start

_start:
        mov ax,0 ; initialize sum to 0
        mov bx,6 ; initialize bx to starting value
keith:  dec bx ;  ; decrement bx by 1
        add ax,bx ; form the running sum and check if condition
        jne keith ; jump not equal to zero to top of loop and keep adding
        mov [sum],ax ; otherwise exit the loop and store result in memory

        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

The key to the answer to the question is to know when the values of the various flags are updated by the microprocessor hardware. The flags are updated after every arithmetic/logical instruction. So, there is most definitely a difference between the two following subsets of code:

```
keith: add ax,bx   ; for the running sum
       dec bx      ; decrease bx and check if not zero
       jne keith

keith: dec bx      ; this will update the flags
       add ax,bx   ; this will update the flags one more time
                   ; so consequently when we do the condtional jump
       jne keith   ; we are checking the the flags at the 'wrong' time.
```

7.4.3 The Compare Instruction

There is a very useful instruction within the x86 instruction set called `cmp`:

```
cmp first,second
```

This instruction performs the following arithmetic operation with updating the flags but without modifying the operands:

```
first - second
```

The `cmp` instruction is often used together with a conditional jump to peform the logical equivalent of an `if` statement in a high-level programming language:

```
if ( al > bl)
{
  // the 'true' stuff to do
    mov cl, 1
}
else
{
  // the 'false' stuff to do
   mov  cl,0
}

// the 'next' statement which is executed after the if
// regardless of the outcome of the if statement

mov dl, 5
```

We would implement the logic of the above if statement in assembly with:

```
ron:        cmp al, bl     ; performs al - bl and updates flags
jack:       jg  al_bigger  ; jump  result greater than 0 ( al > bl)
bl_bigger:  mov cl , 0 ; this is the 'false' stuff
            jmp next       ; now we have to jump over the 'true' stuff
al_bigger:  mov cl, 1   ; this is the 'true' stuff and we continue with
next:       mov dl, 5      ; the next statement
```

Alternatively, it may also be written as:

```
ron:        cmp al, bl       ; performs al - bl and updates flags
jack:       jl bl_bigger     ; jump result less than 0 ( al < bl)
al_bigger:  mov cl,1         ; this is the 'true stuff' we get here if al>bl
            jmp next         ; remember to jump over the 'false' stuff
bl_bigger:  mov cl,0         ; this is the 'false' stuff we get here is al<bl
next:       mov dl,5         ; this is next statement, we always get here
```

Here is a complete assembly language program which implements the two versions of the `if`:

```
; Ted Obuchowicz
; July 6, 2020
; if.asm


section .text
        global _start

_start:
        mov al, 11111111b
        mov bl, 11111110b

;  the following code performs
;  if ( al > bl)
;  {
      mov cl, 1
;  }
;  else
;  {
;     mov  cl,0
;  }
;
;  mov dl, 5

ron:        cmp al, bl
jack:       jg  al_bigger
bl_bigger:  mov cl , 0
            jmp next
al_bigger:  mov cl, 1
next:       mov dl, 5


; here is an alternate version

ron2:       cmp al, bl       ; performs al - bl and updates flags
jack2:      jl bl_bigger2     ; jump result less than 0 ( al < bl)
al_bigger2: mov cl,1         ; this is the 'true stuff' we get here if al>bl
            jmp next2         ; remember to jump over the 'false' stuff
bl_bigger2: mov cl,0         ; this is the 'false' stuff we get here is al<bl
next2:      mov dl,5         ; this is next statement, we always get here

        mov eax,1            ; The system call for exit (sys_exit)
        mov ebx,0            ; Exit with return code of 0 (no error)
        int 80h
```

The astute reader may be wondering the following, "How does the computer know whether the numbers inside the `AL` and `BL` registers (loaded by the two instructions `mov al, 11111111b` and `mov bl, 11111110b`) represent unsigned integers or signed integers (in two's complement format)?"

The answer is that there are two different sets of mnemonics to be used to specify the condtion in a jump instruction. One set tells the hardware that the numbers are to be intepreted as unsigned integers and another set of mnemonics is used when the numbers are to be intepreted as signed nuumbers. The hardware then examines a particular set of flags to determine the "trueness" or "falseness" of the `cmp` instruction.

Jumps may be based upon a single flag setting, or a combination of flags. The following program summarizes the different mnemonics for the available conditional jumps and it lists the two sets of mnemonics for the condition to be tested depending on whether the operands are to be considered as unsigned or signed integers:

```
; Ted Obuchowicz
; jump.asm

section .data

; put your data in this section using
; db , dw, dd directions


section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5


section .text
        global _start

_start:

; jumps based upon testing a single flag in the status register

; jc   jump if carry
; jnc  jump if no carry
; jo   jump if overflow
; jno  jump if no overflow
; jp    jump if parity
; jnp  jump if no parity
; jpe  jump if parity even
; jpo  jump if parity odd
```

```
; js   jump if sign flag set
; jns  jump if sign flag clear
; jz   jump if zero
; jnz  jump if not zero


; jumps based after cmp first, second
;   result of compare      signed operands          unsigned operands
;     =                       je                       je;
;     <>                      jne                      jne
;     >                       jg                       ja
;     <                       jl                       jb
;    >=                       jge                      jae
;    <=                       jle                      jbe


        mov eax,1            ; The system call for exit (sys_exit)
        mov ebx,0            ; Exit with return code of 0 (no error)
        int 80h
```

We can now employ what we have thus far learned to write a program which finds the sum of 10 numbers which are stored in successive memory locations. Two versions of the program will be presented: a brute force method and a more refined version which makes use of register indirect addressing and loops.

The brute force program is:

```
; Ted Obuchowicz
; July 7, 2020
; brute_force.asm

section .data

; define 10 contiguous words in memory

num0 dw 1
num1 dw 2
num2 dw 3
num3 dw 4
num4 dw 5
num5 dw 6
num6 dw 7
num7 dw 8
num8 dw 9
num9 dw 10

section .bss
```

```
sum resw 1 ; reserve 1 word to hold the sum of the 10 numbers

section .text
        global _start

_start:
        mov eax,0     ; clear the entire 32 bits of the register
        add ax,[num0] ;  use brute force to find sum ax = 0 + first number
        add ax,[num1] ; ax = 0 + first number + second number
        add ax,[num2] ; ax = 0 + first number + second number + third number
        add ax,[num3] ; etcetera
        add ax,[num4]
        add ax,[num5]
        add ax,[num6]
        add ax,[num7]
        add ax,[num8] ; brute force is not a practical method for
        add ax,[num9] ; long lists.. can you imagine doing this for
                      ; 1000 numbers?

        ; now store the sum back into memory

        mov [sum],ax

        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

With a little thought, a more intelligent version which makes use of a loop to find the sum can be written. We note that the list of numbers is stored in successive memory locations, each number occupying two byte of memory. Pictorially, the list can be represented as:

Address	Contents
10000000	0100
10000002	0200
10000004	0300
10000006	0400
10000008	0500
1000000A	0600
1000000C	0700
1000000E	0800
10000010	0900
10000012	0A00

For simplicity, we assume the list is stored beginning at address 10000000. Since the `dw` directive was use to define the numbers in the list, each number

occupies two bytes. If we know the address of ith number in the list, adding two to this address yields the address of the i+1'th number in the list. A register can be loaded with the starting address of the list with the instruction:

```
mov ebx, nums
```

And then register indirect mode can be used to access each number in the list and add it to the current contents of `AX`:

```
add ax,[ebx]
```

Another register (`ECX`) can be initialized to 10 and used as a loop counter together with a `jne` instruction to setup a loop.

The complete program is:

```
; Ted Obuchowicz
; July 7, 2020
; use_intelligence.asm

section .data

; define 10 contiguous words in memory
; we don't really require 10 distinct
; labels for the 10 numbers in the list
; one label for the first number will suffice
; since all the numbers are stored in adjacent memory
; location one after another and to access any number
; all that is required is the starting address of the first
; number

nums dw 1,2,3,4,5,6,7,8,9,10


section .bss

sum resw 1 ; reserve 1 word to hold the sum of the 10 numbers

section .text
        global _start

_start:
        mov eax,0      ; clear the entire 32 bits of the register
        mov ebx,nums   ; load starting address of the list into ebx
        mov ecx,10     ; ecx used as a counter to keep track of how many
                       ; numbers in the list have been added

; we make use of register indirect addressing and conditional jumps
; to setup a loop in which the 10 numbers are added
```

```
top:    add ax,[ebx] ; ebx points to first number so use register indirect
                     ; mode to get the source operands and add it to ax
        add ebx,2    ; make ebx POINT to the NEXT number in the list
                     ; by adding 2 since every number is a word ( 2 bytes)
        dec cx       ; decrement the loop counter and check with a
        jne top      ; conditional jump not equal to 0 to see if end of list

        ; now store the sum back into memory

        mov [sum],ax

        mov eax,1            ; The system call for exit (sys_exit)
        mov ebx,0            ; Exit with return code of 0 (no error)
        int 80h
```

7.4.4 The Loop instruction

The x86 instruction set contains a convenient instruction used to set up programming loops. It works together with the `ECX` register which is intially loaded with the desired number of times the loop is to iterate over. The general form is:

```
     mov ecx, the_desired_number_of_times_you_want_the_loop_to_run
top: first_instruction_of_loop_body
     second_instruction_of_loop_body
     third_instruction_of_loop_body
     etc.
     ...
     ...
     last_instruction_of_loop_body
     loop top
     next_instruction
```

The loop instructions performs a decrement of the `ECX` register, followed by a conditional jump to the specified label if `ECX` is not equal to 0. When ecx becomes 0, program execution continues with `next_instruction`.

Basically, the `loop` instruction is a programming convenience and all it does is relieve the programmer from explicitly decrementing the `ECX` register, followed by a `jnz top` instruction. Had the engineers at Intel not incorporated the `loop` instruction into the intruction set, programmers would simply have to do:

```
     mov ecx, the_desired_number_of_times_you_want_the_loop_to_run

top: first_instruction_of_loop_body
     second_instruction_of_loop_body
```

```
    third_instruction_of_loop_body
    etc.
    ...
    ...
    last_instruction_of_loop_body
    dec ecx  ; manually decrement ecx
    jnz top  ; jump to top if ecx still not equal to 0
    next_instruction
```

Here is the sum of the numbers $1 + 2 + 3 + 4 + 5 = 15$ program rewritten making use of the loop instruction:

```
; Ted Obuchowicz
; sum_of_5_with_loop.asm
; July 9, 2020

section .data

; put your data in this section using
; db , dw, dd directions


section .bss

; put UNINITIALIZED data here using
sum resw 1

section .text
        global _start

_start:
        mov ax,0      ; initialize sum to 0
        mov bx,5      ; initialize bx to starting value
        mov ecx,5     ; initialize loop counter to 5
keith:  add ax,bx     ; form the running sum
        dec bx        ; decrement bx by 1 and
        loop keith    ; keep looping as long as ecx != 0
                      ; the loop instruction automatically decrements ecx
                      ; and branches to specified label as long as ecx != 0
                      ; it is just a programming convenience and provides for
                      ; a handy method of setting up a loop
        mov [sum],ax ; otherwise exit the loop and store result in memory

        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

8 Subroutines

8.1 Subroutine Basics

Consider the following C++ program which determines the larger element of two integer array on an element-by-element basis:

```
// T. Obuchowicz
// July 10, 2020
// max_of_array.C

#include <iostream>
using namespace std;


int main()
{

int num1[4] = {1,2,3,4};
int num2[4] = {5,1,7,2};
int ans[4];  // will eventually hold the larger element from each
             // array i.e  [ 5 2 7 4 ]

for(int i = 0 ; i <=3 ; i++)
{
  if ( num1[i] >= num2[i] )
  {
    ans[i] = num1[i];
  }
  else
  {
    ans[i] = num2[i];
  }
}

// display the answer array

cout << ans[0] << " " <<  ans[1] << " " << ans[2] << " " << ans[3] << endl;

 return 0;
}
```

The C++ program can be rewritten to make use of a function which receives two argments and returns the larger of the two as the explicit return value:

```
// T. Obuchowicz
// July 10, 2020
```

```
// max_of_array_with_function.C

#include <iostream>
using namespace std;

// define a function which returns the larger of
// two ints

int max(int mick, int keith)
{
  if (mick >= keith)
   return mick;
  else
   return keith;
}



int main()
{

int num1[4] = {1,2,3,4};
int num2[4] = {5,1,7,2};
int ans[4];

for(int i = 0 ; i <=3 ; i++)
{
  // instead of using the cumbersome if statement to determine
  // the larger of the two array elements, simply pass them
  // as arguments to the function and invoke  the function do the work
  // for you

  ans[i] = max(num1[i],num2[i]);
}

// display the answer array

cout << ans[0] << " " <<  ans[1] << " " << ans[2] << " " << ans[3] << endl;

 return 0;
}
```

In C++, a function invocation causes a change in the program flow of control. The program momentarily "halts" its execution and control then transfers to the function and the function then executes its code. A *return* statement in C++ is used to return back to where the main program "halted".

Since all high-level programs are first converted into the native assembly language version and then assembled into machine code, every assembly language contains specific instructions to support the concept of functions (which are called *subroutines* in assembly language). These functions are:

- `call my_subroutine`

  Invoke the subroutine named with the label `my_subroutine`, can be any label)

- `ret`

  Return to whoever called the subroutine

Here is the assembly language version of the C++ program. It defines a subroutine called `max` within the `.text` section portion of the source code file. The subroutine is written with the expectation that the two numbers to be compared (to determine which is the larger) are to be found in some predetermined CPU registers (`AL` and `BL` in this case), and the "answer" (the so-called return value) is to be stored in the `DL` register. Upon a return from this subroutine, the main loop continues by storing the value in `DL` into memory reserved to hold the answer array:

```
; T. Obuchowicz
; max_subroutine.asm
; July 10, 2020

section .data

; define the two arrays
num1 db 1,2,3,4
num2 db 5,1,7,2

section .bss

; reserve 4 bytes to hold the answer array

ans resb 4

section .text


; define a subroutine which determines the larger
; of two arguments which are passed via the al and bl registers
; the subroutine returns the result in the dl register
; note the use of the label max: to denote the first
; instruction of the subroutine

max:        cmp al, bl      ; is al >= bl ?
```

```
            jae al_bigger  ; if true, then max is al
            mov dl, bl      ; else bl is the max so mov it into 'result' reg. dl
            ret             ; and return from the subroutine
al_bigger:  mov dl, al      ; al >= bl, so load dl with al
            ret             ; and return to the caller


        global _start

_start:
keith:   mov ecx , 4 ; initialize loop counter to 4
         mov edi, num1 ; load first address
         mov esi, num2 ; load second address
         mov ebp, ans
again:   mov al, [edi] ; get first number from memory into "parameter" register
         mov bl, [esi] ; get second number from memory into "parameter" register
         call max      ; invoke the subroutine
                       ; the call instruction causes program execution
                       ; to continue with the instruction at label max
                       ; the instructions comprising the subroutine
                       ; are fetched and executed and when the 'return'
                       ; instruction is executed, program control returns
                       ; to the instruction immediately following the
                       ; call max instrruction, i.e control resumes with
                       ; the mov [ebp], dl instruction
         mov [ebp] , dl ; save the max into memory
         inc edi
         inc esi
         inc ebp
         loop again


        mov eax,1            ; The system call for exit (sys_exit)
        mov ebx,0            ; Exit with return code of 0 (no error)
        int 80h
```

8.2 Subroutine Linkage

How does a subroutine find its way back home to the next instruction following a call upon executing a ret instruction? The answer to this question involves the concept of subroutine linkage. The following is a simplified representation of the `max_subroutine.asm` program as it would partially appear in a computer's memory:

Address	Contents
1000	`cmp al,bl`
1001	`jae al_bigger`
1002	`mov dl,bl`
1003	`ret`
1004	`mov dl,al`
1005	`ret`
1006	`mov ecx,4`
1007	`mov edi, num1`
...	...
1012	`call 1000`
1013	`mov [ebp],dl`
1014	`inc edi`
...	...

The assembler will replace the label `max` with the main memory address corresponding to where the first instruction of the subroutine is stored in memory. Let us assume this is address `1000`. We have also taken the liberty of assuming that each location in memory is sufficiently large enough to store one instruction (in reality, this is often not the case).

Let us suppose the computer is about to execute the `call 1000` instruction. This means that the `EIP` (instruction pointer) register will contain the value `1012`:

Memory	Instruction	Comment
		...
1012	`call 1000`	← `EIP` points to address `1012`
1013	`mov [ebp], dl`	This is where the subroutine must return to once finished

...

As part of the *fetch-execute cycle*, the instruction pointed to by the `EIP` register is copied into a special CPU register (usually called the instruction register) and then the `EIP` is adjusted to point to the next insruction. In this example, it is the instruction at address 1013. This is the so called "return address". This return address, which is the current value of `EIP`, must be saved somewhere. Once it has been saved, the execute steps of a `call` instruction then load the `EIP` with the address specified in the call. In this case, the address 1000 is the value loaded into the `EIP`. A new fetch-execute cycle then commences and the instruction at address 1000 is brought into the CPU and executed.

When the subroutine fetches and executes the `ret` instruction, the return address of 1013 is read from where it was stored and loaded into `EIP`. So, the next instruction to be fetched and executed will be the `mov [epb], dl` instruction.

To summarize, the call instruction performs the following:

1. Save the value of `EIP` which points to the address of the instruction immediately following the `call` instruction
2. Load the `EIP` register with the address contained within the `call` instruction.

A special portion of main memory called the *stack space* is used to store the return address (it is also used to store other values that a subroutine may need). A special CPU register called the *stack pointer* (`ESP` for 32-bit x86 processors) points to the current top of the stack. Items may be placed onto the stack and may be retrieved from the stack. Depending on the design of the processor, the stack may grow (when items are placed onto the stack) from a high memory address towards a low address, and the opposite when an item is retrieved. Intel processors grow from high to low addresses.

Address	Current Location	Comments
4999		
5000	*	← `ESP` points to an initially empty stack

A `call` instruction saves the return address onto the stack by:

1. First decrmenting the `ESP` (to 4999 in this example)
2. Saving the contents of `EIP` into main memory at the address pointed to by `ESP`. In other words, it effectively does the equivalent of `mov [esp], eip`

The situation in the stack now looks like:

Address	Current Location	Comments
4999	1013	← `ESP` points now to this address
5000	*	

To achieve, a return back to the calling program, the `ret` instruction:

1. Retrieves the return address stored in the stack pointed to by `ESP` (address 4999) and loads the value of `1013` back into `EIP`
2. Adjusts `ESP` so that it now points to address 5000 by incrementing `ESP`

The stack now looks like:

Address	Value	Comments
4999	1013	
5000	*	← `ESP` points now back to bottom of stack

The writing of the return address to the stack during a call, and the reading from the stack of return address during a ret contribute to the so-called *subroutine overhead*. Main memory is typically 10 times slower than the processor, so a program which is making many function calls usually runs slower than one which does not make as many function calls.

To summarize, a `ret` instruction performs the following:

1. Move the data stored in main memory pointed by the stack pointer into `EIP`. Effectively doing `mov eip, [esp]`
2. Increment `ESP`

Note how the word *effectively* was used when describing the actions performed by the `call`/`ret` sequence. The instruction pointer is not a programmer visible register, so it may **not** be used as an operand of a `mov` instruction. Attempting to do so would result in an assemble time error:

```
; Ted Obuchowicz
; eip.asm
; July 10, 2020


section .text
        global _start

_start:
        mov eip, [esp]    ; does this assemble ???
                          ; NO. an error is reported:
                          ; eip.asm:10: error: symbol `eip' undefined

        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

8.3 Writing To and Reading From The Stack

Items may be written to the stack space by the following procedure:

1. Decrement `ESP`
2. `mov [esp], the_item_to_be_stored_into_the_stack`

The current top of the stack may be retrieved by:

1. `mov the_destination_you_want_the_retrieved_item_stored_into, [esp]`
2. Increment `ESP`

Here is a program which writes and retrieves a word to and from the stack:

```
; Ted Obuchowicz
; fake_push_pop.asm
; July 10, 2020


section .text
        global _start

_start:
        sub esp, 2          ; decrement esp by 2 bytes
        mov byte [esp], 0xAC   ; write a word of data into the stack
        ; now read it from the stack into some register
        mov ax, [esp]
        add esp, 2          ; make esp point to original top
        mov eax,1            ; The system call for exit (sys_exit)
        mov ebx,0            ; Exit with return code of 0 (no error)
        int 80h
```

The operations of writing data into the stack, and reading values of the stack are so fundamental that most processors have dedicated assembly language instructions to perform them (and automatically perform the required adjustment of the stack pointer register). They are usually called `push` (write into the stack) and `pop` (read from the stack).

Here is the `push` and `pop` version of the earlier program:

```
; Ted Obuchowicz
; push_pop_word.asm
; July 10, 2020


section .text
        global _start

_start:
        push word 0xACDC  ; push a word onto the stack
        pop ax       ; now pop it from the stack into ax
                     ; no need to manually adjust esp when using
                     ; the push and pop instructions.. a handy
                     ; programming convenience
        mov eax,1            ; The system call for exit (sys_exit)
```

```
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

The Intel 80386 and higher processor is capable of pushing a word (2 bytes) or a double word (4 bytes). In the case of a x64-compatible CPU, a quad word (8 bytes). The `nasm` assembler directive `word` in the above `push` instruction tells the assembler that 2 bytes are to be pushed (in the absence of any size specifier, the push insruction defaults to a push of 4 bytes. No special size specifier is required for the `pop` instruction since the destination of the `pop` is a register and thus the number of bytes to be popped can be determined from the size of the specified register. However, in the case that the destination operand of a `pop` is a memory location, a size specifier may be required to distinguish from popping 2 bytes or 4 bytes (or 8 in the case of a 64-bit system).

Here is the 4 byte version of the push/pop program:

```
; Ted Obuchowicz
; push_pop.asm
; July 10, 2020


section .text
        global _start

_start:
        push 0xACDC0CA  ;  push 4 bytes onto the
        pop eax         ; now pop it from the stack into eax
                        ; no need to manually adjust esp when using
                        ; the push and pop instructions.. a handy
                        ; programming convenience
        mov eax,1             ; The system call for exit (sys_exit)
        mov ebx,0             ; Exit with return code of 0 (no error)
        int 80h
```

8.4 Saving/Restoring Registers Within A Subroutine

Suppose it is critical to the correct functioning of a program that certain registers have the same value immediately upon return from a subroutine. Clearly, if the subroutine does not use these registers for its own purpose, then there is nothing to be done within the subroutine. However, if the subroutine needs to use these registers for its own local processing, it is the responsibility of the subroutine to save these registers before modifying them as part of its exection. As well, it must also restore the registers to the saved values before returning. The stack provides a convenient location to save any registers. Registers to be saved are pushed onto the stack. Once saved, the subroutine may use these registers for whatever purpose, and before returning, the registers are restored by popping the stack. **The registers must be popped in REVERSE order from**

which they were pushed onto the stack. The following code gives an example:

```
; Ted Obuchowicz
; August 4, 2020
; save_restore_reg.asm

section .data

; put your data in this section using
; db , dw, dd directions


section .bss

; put UNINITIALIZED data here using
; sum resb 5 format (read as sum reserve bytes 5
; sum2 resw 5 ( reserve 5 words )
; sum3 resd 5 ( reserve 5 double words)

section .text

mysub:  push ax  ; save the ax register onto the stack
        push bx  ; save the bx register onto the stack
        mov ax, -3 ; use ax within the subroutine
        mov bx, -2 ; use bx within the subroutine
        pop bx     ; restore the saved registers by
        pop ax     ; popping in reverse order
        ret

        global _start
_start:
        mov ax, 5  ; load some value into ax
        mov bx, 4  ; and some value into bx
        call mysub

        ; upon return from mysub
        ; ax will contain 5 and
        ; bx will contain 4 and the
        ; program can continue with its work
        ; such as adding ax and bx with the knowledge
        ; that the values contained in ax and bx are what
        ; they were BEFORE calling the subroutine

        mov eax,1            ; The system call for exit (sys_exit)
        mov ebx,0            ; Exit with return code of 0 (no error)
```

```
int 80h
```

8.5 Using The Stack For Subroutine Arguments

The x86 CPU has only a limited number of registers. It can happen that a subroutine requires more arguments than there are available registers. What can be done in such a case? The stack can be used to pass arguments to a subroutine before calling the subroutine. Within the subroutine, the arguments passed on the stack may be accessed using the register indirect addressing mode with the `ESP` register with an offset.

The following is a variation of the max subroutine in which the main program pushes the arguments, which are stored in registers `AX` and `BX`, onto the stack prior to calling the subroutine. The main program executes two pushes:

```
push ax
push bx
```

The stack resembles:

Address	Value	Comments
4996	BL	← ESP Points here
4997	BH	
4998	AL	
4999	AH	
5000	*	Original Top of Stack

We arbitrarily used `5000` as the original value of the start of the stack. Each location in memory corresponds to 1 byte. The main program then calls the subroutine by executing

```
call max
```

Let us suppose that the return address is `0x08049039`. This 4 byte return address is stored on the top of the stack. The situation in the stack now looks like:

Address	Value	Comments
4992	39	← ESP Points here (Low byte of address)
4993	90	
4994	04	
4995	08	(High byte of address)
4996	BL	
4997	BH	
4998	AL	
4999	AH	
5000	*	Original Top of Stack

Note that the value of ESP+4 points to the start of the second argument (the second item pushed onto the stack), and ESP+6 points to the start of the first argument (the first item pushed onto the stack). The subroutine may access the these arguments using register indirect mode with ESP as the index register together with the appropriate offset value. For example, the subroutine can perform:

```
mov ax, [esp+6]  ; get first parameter from stack
mov bx, [esp+4]  ; get second parameter from stack
```

Upon return, the situation in the stack is now:

Address	Value	Comments
4996	BL	← ESP Points here
4997	BH	
4998	AL	
4999	AH	
5000	*	Original Top of Stack

Note how the two items pushed onto the stack are still in the stack. More importantly, the value of ESP points to the second item pushed onto the stack at address 4996. There is a variation of the ret instruction which performs a return and adds a given value to ESP to account for any pushed items onto the stack prior to a call instruction. In this particular example, the ret 4 instruction would be used to return from the subroutine and then add 4 to ESP such that ESP points to the original top of stack at address 5000. Note, that the items pushed onto the stack are still stored in the memory locations, all that has been done is that the esp points to the original top of stack:

Address	Value	Comments
4996	BL	
4997	BH	
4998	AL	
4999	AH	
5000	*	← ESP Points here, Original Top of Stack

The reader may be wondering why bother with the ret 4 instruction, after all it's only 4 bytes of memory that was "wasted" and most systems have lots of memory. In isolation, the unnecessary waste of 4 bytes incurred by not performing the ret 4 instruction is of no consequece. However, if the above was done within a loop of 1000000 iterations, then 40000000 bytes of memory would have been needlessly wasted.

The complete program is:

```
; Ted Obuchowicz

section .data

num1 db 1,2,3,4
num2 db 5,1,7,2

section .bss

; put UNINITIALIZED data here using

ans resb 4

section .text


; define a subroutine which determines the larger
; of two arguments which have been PUSDed onto the stack
; by the calling program
; the subroutine returns the result in the dl register

max:        mov ax, [esp + 6]  ; get first parameter from stack
            mov bx, [esp + 4]  ; get second parameter from stack

            cmp al, bl     ; is al >= bl ?
            jae al_bigger  ; if true, then max is al
            mov dl, bl     ; else bl is the max so mov it into
                           ;'result' reg. dl
            ret 4          ; and return from the subroutine by
                           ; popping ret address and add 4 to esp
                           ; to account for the two pushed arguments
                           ; on the stack
al_bigger:  mov dl, al     ; al >= bl, so load dl with al and return
            ret 4          ; and return ( and add 4 to esp to account
                           ; for the two pushed args)


        global _start

_start:
keith:   mov ecx , 4    ; initialize loop counter to 4
ron:     mov edi, num1  ; load first address
         mov esi, num2  ; load second address
         mov ebp, ans
```

```
again:   mov al, [edi]  ; get first number from memory into
                        ; "parameter" register
         mov bl, [esi]  ; get second number from memory into
                        ;"parameter" register
         push   ax      ; only allowed to push words or double words
                        ;  never byte
         push   bx
         call max       ; invoke the subroutine
         mov [ebp] , dl ; save the max into memory
         inc edi
         inc esi
         inc ebp
         loop again


         mov eax,1            ; The system call for exit (sys_exit)
         mov ebx,0            ; Exit with return code of 0 (no error)
         int 80h
```