GNU ASSEMBLY LANGUAGE MANUAL

PROF. GODFREY C. MUGANDA
NORTH CENTRAL COLLEGE

This manual seeks to provide enough background on GNU Assembly language to allow students of computer organization to program the Intel 80X86 family of processors in assembly language.

1. Basic Architecture

This family of CPUs has a 32 bit architecture, that is, its CPU registers are 32 bits wide. Memory addresses are 32 bits wide, allowing a machine address to fit nicely into a single register. Thus the 386 can address up to 2^{32} bytes, or 4GB of RAM. The architecture has instructions for manipulating data one byte at a time (byte operations), one word at a time (word operations, a word being 16 bits[1]), or one double word at a time (a double word is 32 bits). One can specify the address of any byte in RAM, or the address of any word in RAM, or the address of any double word in RAM. The address of a word is the lower of the addresses of the two bytes that constitute that word; the address of a double word is the minimum of the 4 addresses of the 4 bytes that constitute the double word. Byte addresses begin at 0, word addresses are always even, double word addresses are always divisible by 4. Intel architectures store numbers with the least significant byte in low memory, and the more significant bytes in higher memory.

The system stack is in high memory, and grows downward in memory as data is pushed onto it. Thus pushing a double word on the stack decrements the stack pointer by 4, whereas popping a word increments the stack pointer by 2.

2. General Purpose Registers

This family of processors has six 32 bit *general purpose registers* named EAX, EBX, ECX, EDX, ESI, EDI. These registers are 32 bit extensions of the 16 bit registers AX, BX, CX, DX, SI, and DI, which were originally present in the 16 bit precursors of the venerable 80386. Thus the lower 16 bit portion of the 32 bit register EAX is actually itself a 16 bit register named AX; and changing AX affects the value in EAX and vice versa.

The 16 bit registers AX, BX, CX, and DX are themselves further subdivided into pairs of 8 bit registers:

[1]The terminology used by the assembler harks back to the days of 16-bit hardware processors when a "word" was 16 bits; hardware advances have since increased the size of machine words to 32, and even 64 bit words.

AH	AL
BH	BL
CH	CL
DH	DL

General purpose registers are used by programs to store application-specific data such as values and addresses of memory variables.

3. Special Purpose Registers:

In addition to the general purpose registers, this family of processors has a set of *special purpose registers* used to store data that controls the operation of the CPU. Among these are the *program counter*, the *stack pointer*, and the *frame pointer*.

The *program counter* holds the address of the next instruction to be executed. Because we will not be directly manipulating the program counter in our programs, we do not need a name for it. The *stack pointer* is named ESP (it is a 32 bit extension of the 16 bit register SP) and always points to the item that was last pushed onto the stack. The *frame pointer* is called EBP (it is the 32 bit extension of the 16 bit BP register). and is used to access local variables of procedures in block structured languages. Space for local variables defined in a block is allocated on the stack when control enters the block in a contiguous chunk of memory called a *stack frame*, and then each local variable is accessed using its offset from the beginning of the stack frame. The *frame pointer* register is used to hold the address of the base of stack frame corresponding to the block. It is this function of pointing to the *base* of the stack frame that gives this register its name of BP: Base Pointer.

4. Instruction Formats

Each CPU has an instruction set: the set of hardware instructions that it understands and is designed to execute. A hardware instruction is characterized by a binary *operation code* (also called an *opcode*) denoting the runtime operation to be performed when the instruction is executed. For example, an opcode may specify that some arithmetic or logical operation be performed. In addition, an instruction may specify one or more *operands* representing the data to be operated on by the opcode. Thus a hardware instruction consists of an opcode field, optionally followed by up some operands.

opcode

opcode	operand1

opcode	operand1	operand2

5. Addressing Modes

An *addressing mode* specifies how the contents of the operand field in a hardware instruction are to be interpreted to yield the operand. Here are the commonly used addressing modes:

(1) *Immediate addressing*: the contents of the operand are the operand. In immediate addressing, if the operand field contains a bit combination which represents 234, then the operand is 234.
(2) *Direct addressing*: the contents of the operand give the memory address of the operand. For example, if the contents of the operand denote the number 234, then the operand is the value currently stored at memory address 234.
(3) *Register (direct) addressing*: the contents of the operand field specify a number denoting a register (the hardware assigns each register a unique number). The contents of this register are the operand.
(4) *Register Indirect addressing*: the contents of the operand field specify a register, the contents of the register are used as the memory address of the operand. For example, if the operand specifies the register EBX, and EBX contains 234, then the operand is the contents of memory at location 234. This mode is also called *register indexed addressing.*
(5) *Register indexed addressing with displacement*: The operand field specifies a register, say EBX, and a constant displacement, say d. The sum of the displacement and the contents of the register are added together and used as the memory address of the operand. For example, if $d = 10$ and EBX contains 234, then the operand is at memory address 244.

The X86 actually has additional addressing modes: there is a form of register indexed addressing which uses 2 registers: the contents of the two registers are added together to give the memory address of the operand. There is also a version of Register indexed with displacement which uses two registers, the contents of the two registers are added to the displacement and the sum is used to index into memory. These two addressing modes are sometimes called *base index addressing*, with one of the registers being called a base register (typically used to hold base address of an array) and the other register being called the index register. Finally, one can specify that the contents of the index register be multiplied by a *scaling factor* of 1, 2, 4, or 8 before being added to the base register and displacement to provide the final value used to index memory.

6. GNU Assembly Language

An assembly language program is line oriented: each line contains either an assembler directive or an instruction. *Assembler directives* are also called *pseudoinstructions*: they direct the operation of the assembler and can be used to convey information to both the assembler and the linker about the intended use of various symbols. The assembly language *instructions* specify operations to be performed at run time, and are translated into equivalent machine language instructions. Assembler directives are sort of like variable declarations in a high-level language, while the assembly instructions are like the statements.

An assembly language instruction has the following form:

label:	opcode	operand list	comment

All components except the opcode are optional. It is permissible to have a label by itself on a line: this is interpreted as a null statement.

A pseudo-instruction has a similar structure, the only difference being that the opcode is replaced by a *pseudo opcode*:

label:	pseudo opcode	operand list	comment

Comments start with a # and terminate at the end of the line. C-style and C++ comments can also be used[2]

7. Accessing C library functions

To avoid having to write our own IO routines, we will make library calls to C-library functions. The C standard library then has to be linked in with our assembly language programs. The simplest way to do this is to assemble and link your assembly code using the GNU C compiler `gcc`. Write your code with a `.s` extension, say `myprog.s`, and then assemble and link using the command

```
gcc myprog.s -omyprog
```

to produce an executable `myprog.exe`. The -o option is used to specify the name of the executable file. Omitting the -o option gives an executable whose name defaults to `a.exe.`

The executable program created by the C compiler contains, in addition to the translated version of your assembly code, C *start-up and run-time support code* inserted by the compiler. In fact, it is this start-up code that begins to execute when your program starts running. After the start-up code has initialized global variables that will be used by library code, it transfers control to a function named `_main`. (C compilers prefix all identifiers with an underscore to avoid possible clashes with assembler reserved words, this allows C programs to be linked with assembly code.) Thus your assembly code must have a main function. A return from the main function returns to the C start-up code, which will then terminate the application.

8. Structure of a GNU Assembly Language Program

Programs have 4 segments when laid out in memory:

(1) Stack: contains the application stack. This is defined in the C start-up code, so we don't have to worry about it.
(2) Text: this segment contains the part of the program which is not modified while the program is running. This includes instructions for the application as well as constant data, for example, literal character strings.
(3) Data: contains initialized global variables (data in this section can be modified during execution).
(4) Bss: contains uninitialized global variables.

You can do away with the Bss section by initializing all your variables. This makes for bigger executables: the advantage of a Bss section is that no space needs be

[2]A C++ style comment begins with a // and terminates at the end of the line. Avoid C++ style comments as they sometimes cause mysterious errors during the assembly process.

allocated in the executable while it is stored on disk: space is allocated only when the application is loaded into RAM to be executed.

9. The C calling sequence

High level languages usually pass parameters to procedures on the stack. The C procedure calling convention is to evaluate the actual parameters, and then push them onto the stack in the *reverse* order in which they are encountered (from right to left). The call to the function is then executed. When the function returns, the calling routine removes the parameters from the stack by adding the number of bytes occupied by the parameters to the stack pointer (ESP).

10. A First Assembly program

Here is the venerable "Hello World." program. The program just pushes the address of the string "Hello World" to pass it to the `_printf` function, and then removes the address of the string from the stack when `_printf` returns.

```
           .text
Greeting:  .ascii "Hello World\0"
           .globl _main
_main:     pushl $Greeting
           call _printf
           addl $4,%esp
           ret
```

The `.text` tells the assembler to put whatever follows into the text segment. the next line sets aside memory to hold a string of ascii bytes, and binds the label `Greeting` to the beginning of that memory location. The `.globl` assembler directives tells the assembler that the symbol `_main` will be defined in this module, and that it should be exported to other modules (i.e made visible to the linker so that `_main` can be called by code in other files, namely the C start up code). The next line binds the label `_main` to the location in which the next instruction, in this case `pushl` will be assembled. This is how procedures are defined in assembly language, procedures are no different from labels of instructions. The rest should be clear from what has already been explained (and perhaps from what is to follow.)

11. General Information

When a symbol, label of a memory location, or number is used, it is automatically derefenced to obtain the value stored in the memory location whose address is given by the value of the symbol or number. To make a value immediate, you prefix it with a $. Names of registers are prefixed with % to avoid collisions with user defined identifiers.

Many of the data manipulation opcodes have byte, word, and long (double word) versions. Thus,

`addb`	adds a byte to a byte
`addw`	adds a word to a word
`addl`	adds a long to a long

Operands are characterized as *source* or *destination*, depending on whether they supply the value to be operated on, or receive the value that is the result of the operation. In an instruction with 2 operands, the first is the source, and the second is destination. Thus

```
popl %eax //eax is the destination (stack is source)
pushl %eax //eax is the source (stack is destination)
addl %eax, %ebx //eax is source, ebx is destination
movb %al, $bh //move value of al (source), to bh (destination)
subl $12, %eax // subract 12(source) from eax (destination)
```

An immediate operand can never be a destination. In multi-operand instructions, at most one operand may reside in memory.

12. Procedure call and return instructions

The `call procSpec` instruction pushes a return address onto the stack, computes an address from `procSpec` and puts that address into the program counter.

The `ret` instruction pops a 32 bit value from the top of the stack and stuffs it into the program counter.

13. Procedure entry and exit

By convention, procedures will take their parameters from the stack. If a procedure returns a value that will fits in a register, it will be returned in AL, AX, or EAX, depending on its size. A procedure will in general have local variables as well. The starndard method of accessing both local variables and parameters is to "mark" a place on the stack, and then address both parameters and local variables by their offsets from the "mark." This mark or reference point is stored in the EBP register. Since the procedure that called the current one will already be using EBP to locate its own local variables, we need to save EBP before we store the new reference point in it. We save it by pushing it onto the stack, and then store the new top of the stack (current value of ESP) as our reference point in EBP:

```
pushl %ebp
movl  %esp, %ebp
```

Now the parameters will have been pushed onto the stack, then the return address is pushed by the `call`, and then the old value of EBP is pushed. This means that the last parameter pushed now has 8 bytes of data on top of it, and is at an offset of 8 bytes from the top of the stack, which is the new EBP. By using register indirect with an offset of 8, we can locate the last parameter pushed. The parameter before that, if any, will be at an offset of 8 plus the number of bytes occupied by the last parameter pushed, and so on. Thus, for example, to grab the value of the last parameter pushed and store it in EAX, we write:

```
              movl  8(%ebp), %eax
```

(Putting parenthesis around a register indicates indirection).

Once the procedure has done its job, it is ready to return. Before it returns, is should restore the stack and EBP register to the state they were in when the procedure was called. This is done by executing the sequence

```
              movl %ebp, %esp
              popl %ebp
```

or by executing the single instruction

```
              leave
```

which is equivalent to the above two. This leaves the return address exposed on the stack, which is the precondition for the correct execution of the `ret` instruction.

The following example illustrates access to parameters by using a function which is passed the address of a string. The function, `printString( )` prints the string by passing it to `printf`.

```
             .text
             .globl _main
_string:     .ascii  "This is a string\0"
_main:
              pushl $_string
              call _printString
              addl  $4, %esp
              ret
_printString:
              pushl %ebp
              movl %esp, %ebp
              pushl 8(%ebp)
              call _printf
              addl $4, %esp
              leave
              ret
```

14. Accessing local variables

Space for local variables is allocated on the stack by decrementing the stack pointer by the number of bytes needed to store the local variables. This is done after the frame pointer has been pushed and EBP has been set to mark the current top of the stack. Local variables can then be addressed using indirect addressing with respect to EBP, using negative offsets. Assuming the first local variable occupies 4 bytes, it will be at an offset of -4 relative to EBP, since the old EBP is already at offset 0.

Storage for local variables is deallocated by adding to the stack pointer.

Here is a program which has a function `int sum(int, int)` which takes two integer parameters on the stack (offsets will be 8, 12). The function stores copies of the parameters into two local variables (offsets will be -4, -8). It then adds up those 2 local variables, putting the sum into EAX, deallocates the local variable storage, and returns. The main function just calls `sum` with the values 75 and 25, and prints the result that `sum` returns.

```
               .text
               .globl  _main
_format:       .ascii  "Sum is %d\0"
_main:
                pushl  $75
                pushl  $25
                call _sum
                addl  $8, %esp
                pushl  %eax
                pushl  $_format
                call _printf
                addl $8, %esp
                ret
//int sum(int, int)
_sum:
                pushl %ebp
                movl %esp, %ebp
                subl $8, %esp
                movl 12(%ebp), %ecx
                movl %ecx, -4(%ebp)
                movl 8(%ebp), %ebx
                movl %ebx, -8(%ebp)
                movl -8(%ebp), %eax
                addl -4(%ebp), %eax
                leave
                ret
```

15. Using Global variables

Global variables can be created using the `.long`, `.word` and `.byte` pseudo opcodes all of which take an initial value for the memory location as an operand. Storage for arrays can be allocated using the `.space` pseudocode, which takes a number of bytes to allocate as an operand.

```
x:             .long 12
y:             .word 34
b:             .byte 1
ch:            .byte 'A'
myArray:       .space 20
```

The last directive allocates an array of 5 longs.

The following program initializes two global variables to 12, 3, then adds them up and prints the sum.

```
           .data
x:         .long 12
y:         .long 3
           .text
           .globl _main
format:    .ascii " %d\0"
_main:
            movl  x, %eax
            addl  y, %eax
            pushl %eax
            pushl $format
            call _printf
            addl $8, %esp
            ret
```

16. Reading from the keyboard

Reading from the keyboard is easily accomplished through the use of `scanf`. The following example is a modification of the above: Two integers are read in and their sum is printed.

```
           .data
x:         .space 4
y:         .space 4
           .text
           .globl _main
format:    .ascii " %d\0"
format2:   .ascii " %d  %d\0"
promptformat:
           .ascii "Enter 2 numbers\0"
_main:
//prompt
            push $promptformat
            call _printf
            addl $4, %esp
//read 2 numbers using scanf(" %d %d", &x, &y)
            pushl $y
            pushl $x
            pushl $format2
            call _scanf
            addl $12, %esp
//now add and print
            movl  x, %eax
            addl  y, %eax
            pushl %eax
            pushl $format
            call _printf
            addl $8, %esp
            ret
```

17. Arrays

The following fragment demonstrates the use of indexed addressing with a scaling factor to address array elements. The syntax used is

```
   immed32(baseReg,indexReg,scaleFactor)
```

where `immed32` is an immediate 32 bit value which usually is a label. The label usually indicates the base address of the array. The index register holds the index of the desired array item, and the scale factor is the size of a single array item. The scaling factor is multiplied by the index register (second register in the parenthesis), The resulting sum gives the address of the desired array component in memory. The base register is usually omitted for one-dimensional arrays, it can be useful for multi-dimensional arrays. Alternatively, one can load the base address of the array in the base register and omit the immediate value:

```
    (baseReg, indexReg, scaleFactor)
```

If the base register is not desired, it is simply omitted:

```
   immed32( ,indexReg,scaleFactor)
```

The program uses the compare opcodes `cmpl` which compares its destination and source operands and sets flags in the CPU to indicate whether the destination is equal, less than, less than or equal, greater, greater than or equal, to the source. These flags can then be acted on by one of the conditional jumps, in this case `jl` which jumps if the destination was less than the source.

```
             .data
myArray:     .long  50
             .long  40
             .long  30
             .long  20
             .long  10
             .text
             .globl _main
format:      .ascii " %d\0"
prompt:      .ascii "Enter 5 integers: \0"
_main:

//loop to print the array in reverse order
              movl $5, %edx
nextPosition:
              decl %edx
//are We done?
              cmpl $0, %edx
              jl   weAreDone
```

```
//print next array entry
              pushl myArray(,%edx,4)
              pushl $format
              call _printf
              addl $8, %esp
              jmp nextPosition
weAreDone:
              ret
```

18. Selected Opcodes

We now list some of the most commonly used opcodes for the X86 architecture.

General information: Most assembly language instructions have either one or two operands. An operand is characterized as being a *source* operand if it supplies data for the instruction, and it is characterized as a *destination* if it receives data that is the result of the instruction execution. An immediate operand cannot be a destination. An instruction may reference at most one memory operand. In an instruction that has both source and destination operand, the source operand is written first.

Instructions for floating point numbers are not covered.

All opcodes take a suffix of l, w, or b to indicate the size of the operand as a long word (4 bytes) a word (2 bytes) or a byte.

Data Movement Instructions: These move data from a source to a destination, or exchanges the contents of two operands (register or memory).

```
movl source, dest
movb
movw
xchgl dest, dest
xchgw
xchgb
```

Arithmetic Instructions: These add a source to a destination, subtract a source from a destination, increment or decrement a destination by 1, negate a destination, and perform multiplication and division of signed integers. Other opcodes, `mul` and `div` exist to perform multiplication and division of unsigned integers.

```
addl source, dest
addw
addb
subl source, dest
subw
subb
incl dest
incw
incb
decl dest
```

```
decw
decb
negl dest
negw
negb
imull source
imulw
imulb
idivl source
idivw
idivb
```

In the multiplication and division case, the destination is assumed to be the accumulator (with size of the operand being taken into account), and the source operand may not be an immediate operand. Multiplication results in a double length result and division requires a double length dividend. The size extension opcodes discussed below are useful in converting an operand to double length prior to a division.

Size extension opcodes: All the following opcodes require the operand to be converted to be in the accumulator.

```
cbw
cbtw     //convert byte to word
cwd
cwtd     //word to double
cdq      //double word to quad word
cltd     //long to double long
```

Logical and Bitwise Operations: These perform logical (boolean) operations on a bitwise basis.

```
andl source, dest
andw
andb
orl source, dest
orw
orb
xorl source, dest
xorw
xorb
notl dest
notw
notb
```

Comparison: These opcodes simulate subtracting the source from the destination, and set the condition flags in the flags register so they can be tested by a subsequent conditional jump instruction. The flags reflect the result of the simulated subtraction. Thus the greater than flag is set if the result would have been greater than 0, or equivalently, if the destination is greater than the source.

```
cmpl source, dest
cmpw
cmpb
```

Conditional Jumps: These check the condition flags for the result of the last ALU operation, and jump to the specified address if the tested condition is satisfied.

```
jz address //zero
je        //equal
jne     //not equal
jnz
jl      //less
jle     //less or equal
jg      //greater
jge
jcxz    //cx is zero
jecxz   //ecx is zero
```

Looping Instructions: Decrements count register and jumps to specified address if count is not zero.

```
loop address
```

Address Manipulation: Computes the effective address of the source using whatever addressing mode is specified, and stores the result in the destination. Since the source is a memory operand (it does have an effective address) the destination must be a register.

```
leal  source, dest
```