Intro to GNU Assembly Language on Intel Processors

Prof. Godfrey C. Muganda
North Central College

February 29, 2004

1 Basic Machine Architecture

This family of processors has a 32-bit architecture: its CPU registers are 32 bits wide. In a good machine design, addresses should be the same width as the registers, to enable addresses to be stored in registers and operated on as data. The Intel Pentium follows this tenet of good design, and accordingly, has 32 bit addresses. This means that the Pentium can address up to 2^{32} bytes, or 4GB of RAM. The architecture can manipulate, and has instruction for manipulating data one byte at a time (byte operations), one word at a time (word operations, a word being 16 bits), 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 lowest of the addresses of the four 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 bytes of numbers in the *Little Endian* format, 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.

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 32-bit processors which have descended from the venerable 80386. Thus the lower 16-bit portion of the 32-bit register EAX is actually

itself a 16-bit register named AX; 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:

AH	AL
BH	BL
CH	CL
DH	DL

General purpose registers are used by programs to store application specific data, for example, addresses of variables in memory, values of variables, etc.

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. These registers are typically referred to as 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, 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). The frame pointer 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 in a contiguous chunk of memory called a *stack frame* when control enters the block. Each local variable is then 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 the stack frame corresponding to the block.

2 Introduction to Assembly Language

An assembly language program is typed into a file in line-oriented fashion: each line contains either an instruction, a pseudo-instruction, or a comment.

Instructions specify an operation to be performed by the CPU at runtime. Each instruction is actually translated into an equivalent machine instruction that appears in the object code.

Pseudo-instructions specify an operation to be performed by the assembler at assembly time. A pseudo-instruction

1. affects the way that the assembler assembles the source file.
2. conveys information to the assembler that needs to be communicated to the linker or loader.
3. defines values for mnemonic symbols that are later used during the assembly process.
4. causes storage to be allocated, and may specify values to be used in the initialization of such storage.

Because they are commands to the assembler, pseudo-instructions are also called *assembler directives*.

3 Pseudo-Instructions

The structure of a pseudo-instruction is:

label	pseudo-opcode	list of 0 to 3 operands

Pseudo-instructions can be grouped into the following categories:

1. *Data definition directives*: allocate memory and define mnemonic symbols to be used to refer to these memory locations, and may specify initial values to be stored.
2. *Segment definition directives*: specify that a part of the assembly code be assembled into a specific segment of the run-time memory of the executing program.
3. *Scope and linkage directives*: specify visibility of symbols defined in a file: for example, a directive may specify that a symbol defined in the present file be made accessible to code in other files.

The label field of a pseudo-instruction is optional. As we will see, instructions may also have labels attached to them: in this case, the value of the label is defined to be the run-time address of the instruction being labelled. The label can then be referenced in other instructions to transfer control to this instruction at runtime, or it can be used in a pseudo-instruction to define the value of other symbols.

3.1 Data Definition Pseudo-opcodes

Global integer-based 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.

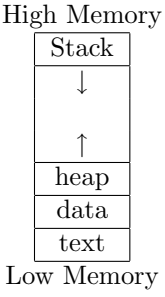
Strings can be defined using the `.ascii` and `.asciz`. Both allocate a consecutive array of bytes and fill it with the ascii codes of a given string; the `.asciz` gratuitously adds a null terminator. Thus, the following assembler directives are equivalent:

```
Message:        .ascii  "Hello World\0"
Message:        .asciz  "Hello World"
```

3.2 Segment definition Directives

When linked and loaded into memory, the address space of an executing program consists of a

1. *text segment*: contains the executable instructions, and any other data that will not be modified at run time. Such "constant" data can include strings and constant variables.
2. *data segment*: contains data that will be modified at run time.
3. *stack*: used by the program to keep track of function calls and returns.
4. *heap*: provides storage that is used for dynamic memory allocation.



Note that the stack is in high memory and grows *downward*. The heap space is allocated right after the data, and grows *upward* toward the stack.

The relevant segment definition directives are

```
.text
.data
```

We will see examples of their use in the sequel.

4 Instructions

The structure of an instruction is:

label	opcode	list of 0 to 3 operands

As in the case of the pseudo-instruction, the label field is optional. In general, an instruction consists of an optional label, an opcode, and zero, one, two, or three operands. Three-operand instructions are very rare, and we shall not mention them further. Each operand is classified as as either a *source* operand or a *destination* operand. A source operand is like a *const value* parameter in the definition of a C++ function: it provides a value to be used by the instruction, but cannot be written to. A destination operand is like a non-const reference parameter; it can be written to as a result of the execution of the instruction. In some one-operand instructions (e.g. `inc`), the same operand may serve as source and destination: in this case, it is considered to be a destination operand.

We next consider the addressing modes, which specify both the syntax of an operand in an assembly language instruction, and the rule for using the syntax of the operand field to specify the location of the operand.

4.1 Syntax of Addressing Modes

1. *Register Addressing* specifies the name of a register prefixed by a percent symbol %. The 32-bit general purpose registers are `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`. The 16-bit general purpose registers are `ax`, `bx`, `cx`, `dx`, `si`, `di`. The 8-bit general purpose registers are `ah`, `al`, `bh`, `bl`, `ch`, `cl`, `dh`, `dl`. Here are some examples of instructions that use register addressing:

```
mov %eax, %ebx
add %ax, %bx
```

2. *Immediate Addressing* specifies a compile-time constant prefixed by a dollar sign $. For example,

```
mov $4, %eax
add $-45, %esi
```

3. *Register indirect* specifies a register enclosed in parenthesis. The following example uses register indirect for the destination operand:

```
mov $45, (%ebx)
```

   It stores the value 45 into the memory location whose address is in the register `ebx`.

4. *Register indexed* specifies a compile-time constant *offset* with an *index* register. In the following example, the destination operand uses 12 for the offset and `ebx` for the index register:

```
mov $45, 12(%ebx)
```

5. *Base indexed addressing* specifies two registers and an offset.

```
mov $45, 12(%ebx, %esi)
```

6. *Direct Addressing* specifies any compile-time constant to represent an address. This may be a constant integer, or a previously defined label. Examples are

```
        .data
x:      .long 12
        .text
L1:     mov x, %eax   //move value at x into eax,
                      //this is direct addressing
L2:     mov $x, %eax  //move address of x into eax,
                      //this is immediate addressing
        jmp  L1       //another example of direct addressing
```

There a number of constraints that are placed on instructions:

1. A destination operand may not use the immediate addressing mode.
2. An instruction can have at most one memory operand. Thus, in a two-operand instruction in which one of the operands references memory, the other operand must be a register operand or an immediate operand.
3. The source and destination operands must be the same size: for example, you cannot move a sixteen bit register into a 32-bit register, or vice-versa:

```
mov %eax, %si
```

   is not permitted because the registers `eax` and `si` have different sizes.

4.2 Function Calls and Returns

We need to discuss how the hardware provides support for function calls and returns. The processor's *program counter* register holds the address of the next instruction that will be executed. A function call is effected by executing an instruction

```
call fun
```

which makes the hardware push the function's *return address* (the current contents of the program counter) onto the program's stack and load the program counter with the address of the function being called. This will result in the hardware fetching and executing the first instruction in the called function.

There are two different hardware instructions that can be used to return from function calls:

```
ret
ret x
```

The first version simply pops the value at the top of the stack into the program counter. It is the programmer's responsibility to make sure that when the `ret` instruction is being executed, the value at the top of the stack is the return address that was pushed by the corresponding call.

The second version, `ret x`, acts like `ret`, but in addition, it discards x number of bytes from the stack *after* popping the return address into the program counter. This is used by function calls that use the Pascal calling convention to clear the stack of parameters passed during the function call. This is explained below when we discuss calling conventions.

Because the stack is used in the management of function calls, there is hardware support for working with stacks. There are push and pop instructions for

```
push source
pop dest
```

to add and remove items from the stack. In addition, the stack can be directly manipulated by adjusting the ESP register, which always points to value that was last pushed onto the stack. On Intel machine, ESP points to the lowest byte of the value at the top of the stack.

We now need to consider the *function calling conventions* that are commonly used by compilers of high level languages. The two that are most prominent are

1. The Pascal calling convention
2. The C calling convention

A calling convention is a contract between the caller and the callee that specifies where the caller will leave parameters that are being passed to the callee, and where the callee will leave a value that is being returned to the caller. In addition, if parameters are being passed on the stack, the calling convention must specify whether it is the caller or the callee who should remove the parameters from the stack when the function call ends.

In the Pascal Calling Convention,

1. The caller passes parameters to the callee by pushing them onto the stack in left to right order, and then calls the function. Thus, the calling sequence for a function `fun(A, B, C)` will be

   ```
   push A
   push B
   push C
   call fun
   ```

2. The callee is responsible for removing the parameters from the stack at the time the function is returning. This is accomplished by the hardware instruction

   ```
   ret x
   ```

   where x is the number of bytes occupied by all the parameters on the stack.

In the C calling convention,

1. The caller passes parameters to the callee by pushing them onto the stack in right to left order, and then calls the function. Upon the function's return, the caller removes the previously pushed parameters from the stack by adding to the stack pointer a number equal to the number equal to the total number of bytes of memory occupied by the parameters. For example, to call the function `fun(A, B, C)` the caller would execute the following calling sequence:

   ```
   push C
   push B
   push A
   call fun
   add $12, %esp
   ```

2. The callee returns by executing a simple

   ```
   ret
   ```

   instruction. It is the responsibility of the caller to remove the parameters from the stack.

5 Mixing C and Assembly Language

Our programs will call functions in the C standard library to perform input, output, and other tasks.

5.1 Calling C library Functions

To call C library functions, we first need to link in the C library functions and also link in C startup code, which initializes some global variables needed for the C library functions to work correctly. C startup code is also called C run-time support code. The simplest way to do this is to assemble your assembly code with the gcc command. The gcc command will assemble your assembly code and automatically link it with the C startup code. When you execute a program linked with C-startup code, the startup code executes first, initializes the global variables used by C library functions, and then calls the main function expected to be in every C program. In our case, this main function will be written in assembly language. By convention, C compilers prefix all identifiers in a C program with an underscore. This means that we need to call our function `_main` so the startup code can link to it.

Here a first program, that calls the library function `puts` to print "Hello World".

```
            .text
Message:    .asciz "Hello World!"
_main:
            push $Message
            call _puts
            add $4, %esp
            ret
            .global _main
```

The `.global` declaration is necessary to enable the C start-up code, which is housed in another file, to call our `_main` function.

5.2 Introduction to C

C is similar to C++. The main differences are that

1. you cannot use classes,
2. you must place all definitions and declarations of local variables in a block statement *before* any executable statements,
3. you cannot define the loop control variable for a *for* statement inside the for loop,

4. and you cannot use the C++ style of comments that start with // and end at the end of the line: instead, you must start your comments with /* and end them with */.

Thus the C++ code

```
{
   // C++ style comment
   int x;
   x = x + 1;
   int y;
   y = x + 3;
   for (int k = 0; k < 3; k++)
   {
      y = y + k;
   }
}
```

has to be written as

```
{
   /* C style comment */
   int x;
   int y;
   int k;
   x = x + 1;
   y = x + 3;
   for (k = 0; k < 3; k++)
   {
      y = y + k;
   }
}
```

Here are some C-library functions that are useful.

1. `char *puts(char *s)` prints a C-string s to standard output and outputs a newline. It returns the same string printed.
2. `char *gets(char *s)` reads a C-string from standard input and stores it at the address s. It returns a pointer to the string that was read: that is, it returns s itself.
3. `int printf(char *format, ...)` is a multi-purpose function that can be used to output a string, called a *format string*, with different types of values embedded in it. The ellipses ... means that this function can take any number of arguments after the format argument. To use *printf*, you

place conversion characters within the format string to mark the places at which values to be printed should be inserted, and you list the values themselves in order after the format string, separating them with commas. Here are the conversion characters you can use:

   (a) `%s`: print a C-string specified by a pointer to char that represents the C-string.

   (b) `%i`: print an integer in decimal format. The integer is specified by any integer expression.

   (c) `%d`: same as `%i`.

   (d) `%x`: print an integer in hexadecimal format.

   (e) `%c`: print single character. The character is specified by any char variable, or an integer giving its ascii code.

4. `int scanf(char *format, ...)` is the counterpart of `printf` that performs input. The format string consists of the same conversion characters as printf, but the arguments that follow the format string must be *addresses* of variables to hold the values read.

6 Some Intel Pentium Instructions

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.

This tutorial does not cover instructions for working with floating point numbers.

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. The suffix can be omitted when the size of the operand can be inferred from the context. For example, in a two-operand instruction where one of the operands is a 32-bit register, the size of the other operand will be assumed to be 32-bits as well.

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 result. 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        //not zero
jl         //less
jle        //less or equal
jg         //greater
jge        //greate or equal
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
```

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.

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
```

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
```