An Introduction to x86 Assembly and Disassembly

UNO CTF

Contents

1 Binary and Hexadecimal

In order to better understand the composition of x86 assembly instructions, it helps to understand hexadecimal byte notation as well as the relationship between hex and binary.

Binary numbers are base 2 numbers, so they have two possible values in each digit: 0 and 1. Each position in a binary number is a power of two. So for an eight bit (one byte) binary number representing the number 7, it would look like this:

Bit 7 2^7	Bit 6 2^6	Bit 5 2^5	Bit 4 2^4	Bit 3 2^3	Bit 2 2^2	Bit 1 2^1	Bit 0 2^0
0	0	0	0	0	1	1	1

Hexadecimal numbers are base 16 numbers, so they have 16 possible symbols. These are 0-9, and for the final 16 numbers, these are A, B, C, D, E, and F.

While binary requires 8 digits to represent a single byte (as a byte is 8 bits), hexadecimal only requires two. An unsigned byte can represent any decimal number from 0 to 255. In an unsigned binary byte, 0 is represented as 00000000 and 255 is represented as 11111111. In hexadecimal, 0 is represented as 00 and 255 is represented as FF.

To easily translate from a binary byte to a hexadecimal byte, you can cut the binary byte down the middle and directly translate the lower four bits to the lower hexadecimal digit, and the higher four bits to the higher hexadecimal digit–four bits allow for a range of (decimal) 0 to 15, so you need four bits to represent the hex range 0-F (F again being 15 in decimal).

So if you split the binary representation of 7 above across the middle, you have 0000|0111, and 0000 in binary (and just about anything else) is 0 in hex, while 0111 is 7 in decimal, so this would be 07 in hex notation.

Let's try a larger number: 62 in decimal. Let's try to reduce this quickly to a hex number. Because this is between 0 and 255 decimal, we only need two hex digits to represent this number. Because hexadecimal deals in powers of 16, we can go ahead and divide 62 by 16, and the result will be the higher digit. So decimal 62 divided by decimal 16 results in the number 3, with a remainder of 14. Because we're dealing in hex, and 0-15 decimal in hex each has a single digit representation, we will turn the 14 into E. So the final hex representation of 62 is 3E.

Let's take hex 3E and turn it into binary. Splitting them up, we can look at the high digit first, because it's only 3. So three in binary is 0011, because we have a 2^0 (1) and a 2^1 (2).

Generally when splitting a number into binary digits, you want to start with setting the binary digit with the highest power not exceeding the number you're dealing with to 1, and then determining which smaller powers you need to add to get your number.

Let's now split the lower hex digit. The largest binary power in E, or decimal 14, is 8 (2^3). So we have a 2^3. The next binary power is 2^2 (4), and adding that to 8 is decimal 12. We know that 2^1 is 2, and so adding that to decimal 12, we have our decimal 14. So we have a 2^3, a 2^2, a 2^1, and no 2^0, so this will be 1110 in binary. Adding this to binary 00110000 (I added zeroes to the previously calculated 0011, because our 0011 represents a count of 16^1, with the rest being a count weighted at 16^0), we get 00111110 in binary.

Often, you may be dealing with different representations of numbers at the same time in code or disassembly. So to indicate that a number is in binary, you will often see the prefix 0b. So you might see the binary number we just calculated as 0b00111110. Hexadecimal numbers are often prefixed with 0x, so this number in hexadecimal would be shown as 0x3E. Less frequently, hexadecimal numbers may be suffixed instead with "h", so 3Eh is another way to show it.

1.1 Two's Complement

So far we've only dealt with unsigned numbers, so numbers that are always non-negative. If you need to represent positive and negative numbers, you need to sign your numbers. We can represent these numbers in the same space as unsigned numbers, but to do so, we can use one of our bits (typically the most significant bit) to represent the sign rather than a power of 2. While in an unsigned byte we can represent the numbers from 0 to 255, in a signed byte, we can represent the numbers -128 to 127. The maximum positive is one less than the negative, because 0 is represented with the positive sign. The signed bit will be 1 for negative numbers and 0 for positive numbers.

However, we don't just flip a single bit to change from a negative to a positive number and vice versa. To change a positive number to a negative number, you must flip all of the bits and add 1 to the number.

So for a positive decimal 62, we have 0b00111110. To get the negative, first we flip all the bits, resulting in 0b11000001. Then we add one, resulting in 0b11000010. Let's translate this to hex. So, splitting them into two digits, on the most significant digit, we have a 2^3 (8), a 2^2 (4), and no 2^1 or 2^0. So we have a decimal 12, or a hex C. Then for the lesser hex digit, we have 0010, so only a 2^1, or 2. So the negative representation of 62 or 0x3E is 0xC2.

2 x86 Assembly

2.1 x86 Processor

x86 is a family of 16, 32, and 64-bit processors. The number of bits is also known as the word size–the native size of registers and memory that the processor handles. The instructions decoded by the processor are of variable length, and can extend past the processor's word size, but for now, we're generally concerned with data size, registers, and memory.

The remainder of this document will refer to a 32-bit x86 processor for simplicity.

2.1.1 Registers

Registers can be viewed as small, discrete words of memory that are available directly on the processor. If data is required from memory and is not in a register itself, the processor will generally load that piece of data into one of its registers in order to perform the desired operation.

As before, each register on our processor will be 32-bits. The processor is backwards-compatible in that it allows for the usage of 16-bit registers and words, but when this happens, these registers are simply 16-bit subsets of the native 32-bit registers.

The registers are as follows[1]:

- General-Purpose Registers

Register	Purpose
`EAX`	Accumulator–This register is generally used for arithmetic operations
`ECX`	Counter–used when needed to increment and decrement, occasionally for other purposes as well
`EDX`	Data–Used for arithmetic and I/O operations
`EBX`	Base–Used to point to data
`ESP`	Stack Pointer–points to the top of the stack; stack variable access is generally done relative to the stack pointer
`EBP`	Stack Base Pointer–Used to point to the base of the stack; the address contained in this register is used to restore the stack when the frame is no longer needed (e.g., when a function returns)
`ESI`	Source Index–used to hold pointer to the source for streaming data transfers
`EDI`	Destination Index–used to hold pointer to the destination for streaming data transfers

- Segment Registers

  Occasionally segmentation comes into play; different data may lie in different areas of memory, but generally with more modern applications, this is not so much of a concern. The segment pointer is used somewhat to serve as a base pointer to the type of data in use.

[1] From `https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture`. Source is not necessarily used word-for-word, but core information is borrowed from this source in terms of discussion of x86.

Register	Purpose
`SS`	Points to the program stack
`CS`	Points to the program code
`DS`	Points to the program data
`ES`	General-purpose data segment
`FS`	Additional general-purpose data segment
`GS`	Additional general-purpose data segment

- EFLAGS Register Flags

  After instructions are carried out, particular one bit flags are used to somewhat determine the result of instructions. For example, if subtracting the contents of one register from another results in a negative number, the sign flag (`SF`) will be set. If it is zero, the zero flag (`ZF`) will be set. Here are a few examples of the flags in this register.

Flag	Purpose
`CF`	Carry Flag–if an operation results in a carry outside of the size of a register, this flag will be set
`ZF`	Zero Flag–if an operation results in a zero value, this flag will be set.
`SF`	Sign Flag–if an operation results in a negative value, this flag will be set.
`DF`	Direction Flag–For stream operations, this determines whether the pointer will be incremented or decremented as memory is read for the operation.
`OF`	Overflow Flag–If an arithmetic operation results in over or underflow, this flag will be set.

- Instruction Pointer

Register	Purpose
`EIP`	Contains the address of the next instruction to be executed in the program. This register is not manually modified, but is also changed in the event of a branch in control flow. In this case, it will point to the instruction at the destination of the branch.

2.2 Main Memory

Main memory, generally referred to as *RAM*, is where programs are loaded when they are to be executed. Program data is also kept here. Processors generally load their instructions from main memory as they are to be executed, and they pull other data from here into registers as necessary.

2.3 Secondary Memory

Secondary memory is generally your hard disk. This is used for persistent storage. When programs are loaded into main memory, they are usually read from a file in the secondary memory.

2.4 x86 Instructions

x86 instructions, as specified by the x86 Instruction Set Architecture (or ISA), are the instructions that make up those used in an assembly language. They are composed of an operation, specified by an *opcode*, as well as a set of registers and/or memory locations which are operated upon–these registers and/or memory locations are known as the *operands*.

2.4.1 Opcodes

Opcodes are the representation of an operation in the memory. They take some of the length of an instruction. For example, the opcode to add the contents of one 32-bit register to another might be 01. The opcode to compare the contents of two registers (by subtraction) might be 39. The opcode to unconditionally jump to a piece of code elsewhere in a program might be E9 (note that these opcodes are in hex).

2.4.2 Operands

An instruction optionally acts on operands; generally registers or memory addresses

2.4.3 Mnemonics

While assembly instructions are made up of binary, the programmer can program assembly in terms of mnemonics, which are then assembled into binary for the machine to execute. Mnemonics are shorthand. There are two main formats of mnemonics: Intel syntax and AT&T syntax. Typically in Intel syntax, an instruction first contains the mnemonic for the opcode, such as ADD or JMP. Then, if operands are necessary for the instruction, the destination operand is then provided, and then the source. AT&T typically requires that the source operand be provided first, and then the destination. AT&T syntax also adds a number of additional details that are abstracted away in Intel syntax, such as the length of operands being operated upon.

Here are some examples of instructions written in their Intel and AT&T syntax mnemonics, as well as their meanings.

Intel	AT&T	Meaning
`mov eax, ebx`	`movl %ebx, %eax`	Move the contents of the ebx register into the eax register
`sub ebx, ecx`	`subl %ecx, %ebx`	Subtract the contents of ecx from the contents of ebx, and place in ebx
`mov [var], eax`	`movl %eax, (var)`	Move the contents of eax into the memory location pointed to by var
`mov eax, [var]`	`movl (var), %eax`	Move the contents of the memory location pointed to by var into eax

Note that AT&T syntax specifies that the move and subtract operations are operating on long integers (32-bit), whereas Intel abstracts this away. Also, the square bracket (or parentheses, in the case of AT&T) syntax indicates that the operation is working on the data contained at the memory address specified within the square brackets, rather than the data specified within the square brackets themselves–if the instruction is `mov eax, [8080F583]`, then the data located at the address 8080F583 is loaded into eax, not the value 8080F583 itself. This is known as dereferencing a pointer, and this is done frequently in C and C++ as well.

2.4.4 Sample Assembly Program

Here is a short assembly program that prints the string "Hello world!" to the terminal[2].

```
section .data
str: db 'Hello world!', 0Ah
str_len: equ $ - str


section .text
global _start

_Start:
    mov eax, 4
    mov ebx, 1

    mov ecx, str
    mov edx, str_len
    int 80h

    mov eax, 1
    mov ebx, 0
    int 80h
```

You may have noticed some declarations before the actual Intel x86 assembly instructions you have been introduced to. These are pseudo-instructions, which are not directly interpreted by the assembler, but are a form of shorthand used to set up the program when it is assembled. For example, the string "Hello world!" is placed in the data section along with its calculated length.

With this setup, you can refer to data by name within the assembly code, as in the line "`mov ecx, str`", which places a pointer to the memory address at which `str` begins.

Also, the `int` instruction is important. This stands for interrupt, or software interrupt. Int 80h is used to make system calls, the exact choice of which is specified within eax. System call 4 (which was placed into eax) is used to write to a file descriptor, which is provided through ebx. File descriptor 1 is standard output, so int 80h is used to write to the screen. So here, eax contains the system call number for writing to a file descriptor, ebx contains the file descriptor, ecx contains the string to write, and edx contains the length of the string to write. The second int 80h call, using system call 1, is used to exit a program. ebx contains the exit status. The exit status provided here is 0, which indicates that the program exited successfully.

2.4.5 Assembly Calling Conventions and the Stack

When writing assembly programs, you can make use of subroutines that either you declare or use from a dynamic library. A calling convention refers to how parameters are provided to subroutines and how (and by whom) the parameters are cleaned.

[2]From `https://en.wikipedia.org/wiki/X86_assembly_language#.22Hello_world.21.22_program_for_Linux_in_NASM_style_assembly` This was copied instruction by instruction without comments.

For example, in `cdecl`, the standard C calling convention, arguments are first pushed onto the stack in reverse order. Then the subroutine is called. Once the subroutine returns to the caller (with its return value typically placed into eax), the caller then removes the arguments it previously placed onto the stack.

Here is an example[3]:

```
caller:
    ; make new call frame
    push ebp
    mov ebp, esp
    ; push call arguments
    push 3
    push 2
    push 1
    ; call subroutine 'callee'
    call callee
    ; remove arguments from frame
    add esp, 12
    ; use subroutine result
    add eax, 5
    ; restore old call frame
    pop ebp
    ; return
    ret
```

When a push occurs to place a value onto the stack, the value is placed at `esp`, and then esp is decremented by the size of the value (four bytes each, so 32 bits each in this case). The size of the push is generally determined by the processor word size. To restore the stack to its original position after the subroutine call is complete, the program could either perform 3 pops to match the three pushes, or it can manually restore esp, which is done here by adding 12 bytes back to esp.

Some calling conventions may require the callee subroutine to clean up the stack rather than the caller. Other calling conventions may only place arguments and return values in registers, requiring no usage of the stack.

When programming in assembly language, the programmer needs to know the calling conventions of subroutines to correctly interact with them.

3 x86 Disassembly

If you're provided with a compiled C program and you want to know its inner-workings (maybe to be assured of its legitimacy or to find vulnerabilities), you will need to *disassemble* it. Disassembling a program refers to translating the binary source back into mnemonics that a programmer can read.

Programs that can perform this process are known as *disassemblers*, and the resulting mnemonic representation of the code is known as *disassembly.* Many debuggers such as GDB can disassemble programs, and there are other programs such as IDA or radare that are primarily disassemblers.

[3]From `https://en.wikipedia.org/wiki/X86_calling_conventions`, copied exactly as displayed.

3.1 GDB Example 1: Hello world

You've been provided with an example program, called `helloworld`. This is a compiled C version of the assembly Hello World program introduced above. Open it with the command `gdb helloworld`.

1. You'll be provided with a command prompt that looks like "(gdb)". By default, GDB uses AT&T syntax, so let's change that. Enter the command `set disassembly-flavor intel`.
2. In order to approach disassembling `helloworld`, you'll want to see what functions are in the program. Type `info functions`, and you'll be presented with a list of all defined functions in the program. It may be a bit daunting, but we're only really concerned with one function: the `main` function. Most of the other functions are a result of boilerplate code that results from the compilation process–there would be much less code if this were assembled by hand–as you've already seen.
3. Run the command `disass main`. This will show the disassembly of the main function. If you don't see "End of assembler dump." at the bottom, hit `[return]` until you do.
4. You'll see some boilerplate code, but for now, we're interested in the _printf function call. First note the `sub esp,0x10` instruction. This instruction subtracts the stack pointer, providing for enough space on the stack for 0x10 bytes of local variables.
5. Let's take a look at the line that says `call 0x80482f0 <printf@plt>`. This is of course a call to *printf*. Taking a look at the preceding line, you'll see an instruction that looks like `mov DWORD PTR [esp],0x80484d0`. In the cdecl calling convention, arguments are passed by pushing them onto the stack. However, as the program previously made space on the stack by subtracting a value from the stack pointer, so we can place the value right in the stack, right where the stack pointer points. The address 0x80484d0 is where the "Hello world!" string resides, so this address is being provided to the printf function call.
6. The `leave` function destroys the current stack frame.
7. `retn` returns from the function. As main contains the only user programmed code, we are essentially done with the program.

3.2 GDB Example 2: echo

You've been provided with an example program, called `echo`. Open it with the command `gdb echo`.

1. You'll be provided with a command prompt that looks like "(gdb)". By default, GDB uses AT&T syntax, so let's change that. Enter the command `set disassembly-flavor intel`.
2. In order to approach disassembling `echo`, you'll want to see what functions are in the program. Type `info functions`, and you'll be presented with a list of all defined functions in the program. It may be a bit daunting, but we're only really concerned with one function: the `main` function. Most of the other functions are a result of boilerplate code that results from the compilation process–there would be much less code if this were assembled by hand.
3. Run the command `disass main`. This will show the disassembly of the main function. If you don't see "End of assembler dump." at the bottom, hit `[enter]` until you do. There are some serious security flaws in this program, but we won't discuss those in this lab (see if you can identify them).

4. You'll see a lot of boilerplate code, but for now, we're interested in the three function calls. First note the `sub esp,0x20` instruction. This instruction subtracts the stack pointer, allowing for enough space on the stack for 0x20 bytes of local variables.

5. Let's take a look at the line that says `call 0x8048340 <malloc@plt>`. This is of course a call to the *malloc* function.

   When allocating memory for variables, there are two types of allocation. Static allocation is done when you know the exact size of the variable you are allocating space for. For example, when you create an int variable, you know that you will need 16 bits, but for a string, you need an indeterminate amount of memory.

   Static allocation is done from the stack. When this happens, we know the exact size of the variables, and so we can push to and pop from the stack as necessary for these variables.

   Dynamic allocation, for when you may need a variable amount of memory for data such as strings or arrays, is done from the heap. This requires a call to a function such as `malloc`, which takes a request for size of a buffer and returns a pointer to the start of that buffer on the heap if there is enough memory left on the heap to fulfill the request.

   If you look at the `malloc` manpage, you'll see it takes one argument: size. In the cdecl convention, arguments are pushed onto the stack. However, if you'll notice the line right before the call to `malloc`, the instruction `mov DWORD PTR [esp],0xff` is performed. This places the value 0xFF right at the stack pointer. No push is performed to do this, because if you remember, we manually moved the stack pointer to allow for some space. This value, 255 decimal, is being provided to malloc, which means we're requesting a buffer of length 255 bytes. The `mov DWORD PTR [esp+0x1c],eax` instruction saves the result of malloc (a pointer to the buffer) in the stack.

6. The next 8 instructions following `mov DWORD PTR [esp+0x1c],eax` simply check for malloc success, and jump to exit if there is a failure.

7. Let's assume malloc succeeded. If we succeeded, we will eventually end up at main+61–the instruction `mov eax,DWORD PTR [esp+0x1c]`. This places a pointer to the allocated buffer in eax. You'll then notice the movement of that pointer from eax to the current stack pointer location, preparing for another function call.

8. At main+68, the `gets` (or get string) function is called. This function takes user input from the keyboard, and once the `[enter]` key is hit, it places the input into a buffer–the pointer to which was just recently provided.

9. At main+73 (`mov eax,DWORD PTR [esp+0x1c]`), you'll notice the variable that holds the pointer to our buffer is again placed into eax. Then at main+77, the new contents of eax are placed right at the stack pointer location.

10. At main+80, `puts` (or put string) is called. This function prints the provided string to standard output, or the terminal.

11. Once puts is called, there's not a whole lot else to do. The contents at esp+0x18 are placed into eax, and this variable is simply used as a return value–a value of 0 indicates that the function ran successfully. The final two instructions are `leave` and `ret`. `leave` destroys the stack frame, and `ret` returns to the caller. But because `main` is the only function with user-written code, this means that our program is essentially complete.

12. Now that we've gone over the program, you'll notice its only real purpose was to take input from the keyboard and print it back to the screen, hence the name `echo`. This was a somewhat simple program; it only gets more complex from here. At this point, you'll likely understand why more specialized programs are used for disassembly–following control flow without some sort of graph is difficult enough! Programs like IDA and radare can easily show control flow with arrows and even separated code blocks with arrows pointing from one to another with each jump. GDB is fine for disassembly of smaller programs, but it's best used primarily as a debugger.

13. You may want to open the smaller hello world program to see how much simpler and easier to understand hand-written code can be! And you may even want to open `echo` in IDA to see if you find it any easier to read there.