Chapter 3: Assembly Language and Memory

Objectives:
(a) Explain the operations of several basic assembly language commands, including mov, cmp, jmp, jle, jne and inc.
(b) Demonstrate the ability to debug a running C program in memory, to include the inspection of processor registers and arbitrary memory locations
(c) Demonstrate the ability to analyze C programs that employ if-else statements and for loops.
(d) Apply Boolean logic to evaluate how selection structures can alter program flow.
(e) Analyze existing programs for flaws with the gdb debugger.
(f) Explain the distinction between machine code, assembly code and high-level source code.

I. A Little More C: The if-else selection structure and the for repetition structure

The order in which program statements are executed is called flow of control. All of the programs that we have seen so far consist of statements executed in order, one after the other. As we will see, we often need to vary the order in which statements are executed.

1. The if-else statement  Consider the following example.

   Write a program that accepts the user’s GPA as an input and prints “You’re on the Dean’s List!” if her GPA is greater than or equal to 3.5, and prints, “Keep trying!” if her GPA is less than 3.5.

Right now, we can’t solve this simple problem because we have no way for a program to choose between alternatives. To solve this problem, C provides an instruction that allows the user to select which statements to execute based on the value of one or more variables. This useful C instruction is the if-else statement.

The program that solves the problem above is shown below:

```c
#include<stdio.h>
int main( )
{
    float gpa;
    printf( "Enter GPA: ");
    scanf( "%f" , &gpa ) ;
    if ( gpa  >=  3.5 )
    {
        printf("\nYou’re on the Deans List!\n");
    }
    else
    {
        printf( "\nKeep trying!\n" );
    }
    printf( "\nGo Navy!\n\n" );
}
```

If the value of the variable gpa is greater than or equal to 3.5, all of the statements between these two braces will execute. The statements within the braces after the else are skipped.

If the value of the variable gpa is less than 3.5, all of the statements between these two braces will execute. The statements within the braces following the if will be skipped.
In the code above—immediately after the word if—we have the Boolean expression in parenthesis:

\[ gpa \geq 3.5. \]

If this Boolean expression is true (i.e., if the value of the variable \( gpa \) is indeed greater than or equal to 3.5), the statements contained within the first set of braces (following the word if) will be executed, and the statements within the second set of braces (following the word else) will be skipped.

If, on the other hand, the Boolean expression is false, the statements within the braces following the word else will execute (and the statements within the braces following the word if will be skipped).

Shown below are two separate executions of the program shown above. Note that in both cases, the statement `printf( "\nGo Navy!\n\n" );` is executed.

Boolean expressions always evaluate to either true or false. The simplest Boolean expression compares numbers and/or variables using a comparison operator. You should be familiar with the usual operators: >, >=, < and <=, == and !=. In C, we can check for equality by using two equals signs in a row, with no space between them. So, for example, a Boolean expression that can be used to check if a float variable named hours is equal to forty would be

\[ \text{hours} == 40 \]

In C, we can check for inequality by using an exclamation sign followed by an equals sign. So, for example, a Boolean expression that can be used to check if a char variable named grade is not equal to F would be

\[ \text{grade} != 'F' \]

There are two modifications we can make to the if-else statement. The first modification is that we don’t have to have the else part. In this case, the program performs the statements in braces following the word if when the Boolean expression is true, and skips these statements if the Boolean expression is false. Consider our earlier program without the else portion, and the corresponding screen captures:

```c
#include<stdio.h>

int main( )
{
    float gpa;
    printf( "Enter GPA: ");
    scanf( "%f" , &gpa );

    if ( gpa >= 3.5 )
    {
        printf("\nYou’re on the Deans List!\n" );
    }
    printf( "\nGo Navy!\n\n" );
}
```
The second modification is that if there is only a single statement within the braces, then the braces are optional. The programs shown above will work just as well without the braces surrounding the `printf` statements.

2. The `for` statement  Many programs include some actions that must be performed again and again, some number of times—that is, we may want to repeat sections of our program again and again. A part of a program that repeats a number of statements is called a loop. Let's jump right into examining a program that uses a `for` loop, along with its corresponding output.

```c
#include<stdio.h>
int main()
{
    int count;
    for( count = 1 ; count <= 5 ; count = count + 1)
    {
        printf( "%d\n" , count );
    }
}
```

Any statements within braces following the word `for` are the statements that will be executed each time the loop iterates. In this example, there is only one statement within the body of the `for` loop:

```
printf( "%d\n" , count );
```

and so each time the loop iterates, the program will print out the value of the variable `count`, followed by a new line. The question remains: What controls the number of times the loop will iterate?

In this example, the variable `count` will be used to determine the number of times the loop executes. When we enter the `for` loop, the loop control variable (i.e., `count`) is initialized:

```
for( count = 1 ; count <= 5 ; count = count + 1)
```

This tells how the loop control variable is initialized. This initialization occurs only once.

Next, the program checks to see if the Boolean expression is true:

```
for( count = 1 ; count <= 5 ; count = count + 1)
```

The loop control variable is compared to 5. This Boolean expression is used to determine if the loop should execute.

Since the variable `count` (at this point in time) is equal to 1, the Boolean expression is true and we execute the statement in the body of the loop.

```
midshipman@EC310:~/work $ ./a.out
1
2
3
4
5
```

When we finish executing the body of the loop, we update the loop control variable:

```
for( count = 1 ; count <= 5 ; count = count + 1 )
```

The loop control variable is updated.
The loop control variable `count` is now equal to 2. We once again return to the Boolean expression:

```c
for( count = 1 ; count <= 5 ; count = count + 1)
```

and see that it is true (2 is indeed less than or equal to 5) and we again execute the body of the loop.

When we finish executing the body of the loop, we update the loop control variable:

```c
for( count = 1 ; count <= 5 ; count = count + 1 )
```

and `count` becomes 3. We then return to the Boolean expression, note that it is true, execute the loop, and update the loop control variable to 4. The loop executes again, and `count` is then updated to 5. The loop executes again (since 5 is less than or equal to 5) and `count` is then updated to 6. When `count` is updated to 6 the Boolean expression becomes false and we exit the loop.

Note that in the `for` loop the initialization is done only once, and we then "bounce back and forth" between the Boolean expression and the update of the loop control variable.

An [flowchart](#) for the `for` loop is shown below:

- **Initialization of loop’s control variable** occurs only once!
- The update happens after the body of the loop is performed!
Practice Problem

For each of the for loops shown below, state how many times the loop will iterate.

(a) \[\text{for}( \text{i} = 1 ; \text{i} \leqslant 100 ; \text{i} = \text{i} + 1)\]
(b) \[\text{for}( \text{i} = 3 ; \text{i} > 1 ; \text{i} = \text{i} - 1)\]
(c) \[\text{for}( \text{i} = 7 ; \text{i} \leqslant 21 ; \text{i} = \text{i} + 7)\]

Solution:

(a) \(n\) (b) \(n\) (c) \(n\)

Practice Problem

Examine the following C program and describe the expected output.

```c
#include<stdio.h>
int main( )
{
    int count;
    for( count = 1 ; count <= 2 ; count = count + 1 )
    {
        if( count > 1 )
            printf( "Cyber\n" );
        else
            printf( "Fun\n" );
    }
}
```

Solution:

II. Machine and Assembly Language

To understand the damage that an adversary can inflict on your host computer, you have to know a little bit about programming, since, after all, a computer will only do what it is told to do, and a computer is told to do things via programs.

But the programs—the software—are only half the story. To understand how a program can damage your computer, you have to know how the hardware interacts with the software. We examine the relationship between software and hardware by focusing on hardware that runs the x86 instruction set, the so-called x86 chip. This is by far the most common hardware implementation in PCs and servers.

So, now that we know a little bit about software, let's go back to the machine!
1. **Machine Language**  Examine the C program shown below. What does it do?

```c
#include<stdio.h>
int main( )
{
    int x = 7;
    x = 2001;
}
```

Suppose we enter this program using nano, and then compile it using gcc. Remember that the gcc compiler converts the source code (which we humans like) to machine language (which the computer likes). The machine language code is written in the specific machine language for the x86 processor, which is the CPU in your computer. The file containing the machine language code (i.e., the executable file) is named `a.out`. We can run our program by entering: `./a.out`.

What is `a.out`? And why do I need to put a dot and a slash in front of `a.out` to execute my program?

Since the CPU can only execute machine language instructions, a C program that you write must be converted into a machine language program before it can be executed by the CPU. This conversion is performed by the gcc compiler.

By default, the compiler gives the name `a.out` to the file containing the machine language program. When you recompile a program, a new `a.out` replaces (overwrites) any file named `a.out` that may already exist in the working directory.

To execute the machine language code (i.e., to run your program), you have to specify the relative pathname to the file named `a.out`. Recall that a single dot (.) can be used as a shorthand for your current working directory. Thus, the relative path name to your machine language file is `./a.out`.

Later, you will learn how to change the name of your executable code to a file name of your choosing.

Remember that a CPU can only interpret very simple instructions that have been “hardwired” as electronic circuits on the CPU chip. This set of simple hardwired instructions is termed the *instruction set*. Each instruction in the CPU’s instruction set has associated with it a unique string of bits that the CPU can interpret. So, compilation converts the source code instructions to the correct bit strings that correspond to instructions from the CPU’s instruction set.

You may be wondering: If a computer can only carry out a small number of tasks (the limited number of simple instructions that have been “hardwired” as circuits on the CPU chip), how are computers able to perform complex operations? To gain an insight into the answer, consider that the complete works of Shakespeare, the English translation of the Bible and the US Constitution are all written using 26 letters, a space symbol and a
few punctuation symbols. Similarly, massive programs can be built by combining the limited number of machine language instructions in various ways.

So… what does the machine language program for our simple C program look like? Here is a picture of the machine language code for our program, beginning at the line that says `int main( );`.

![Machine Language Code](image)

Machine language is supposed to be bits… where are the bits? Recall that we use hexadecimal to represent binary more compactly. The machine language shown above is in hexadecimal.

The machine language code is on the right. On the far left are the addresses in main memory where the machine language instructions are stored. The addresses are also presented in hexadecimal. So, let's add headings to our picture:

<table>
<thead>
<tr>
<th>address</th>
<th>machine language instruction</th>
</tr>
</thead>
</table>

Remember that any program that you run—MSWORD, Firefox, a video game—must be in main memory. The operating system decides where a program will actually be placed in memory. So, the line that reads

```
8048344: 55
```

means that at memory address `08048344` there exists the machine language instruction `55`. Pictorially:

![Address and Instruction](image)

So, the first instruction listed is `55` (remember, this is all hexadecimal—think of this first instruction as `0x55`).

**Practice Problem**

How many bits are in an address?

**Solution:**

---

1 Note that the first line of the program, `#include<stdio.h>`, creates object code too—but this standard program opening produces standard object code. We are primarily interested in the part of the program that we write (which comes after the line `int main( );`) so we’ll only focus on that.

2 Notice that in the address listing above, the very top line shows the full address, but subsequent lines do not show the leading zeros.
Practice Problem

How many bits are represented by the hexadecimal number 55?

Solution:

And this number 0x55 means... what?

If I looked up the x86 instruction set (e.g.: at [http://sparksandflames.com/files/x86InstructionChart.html](http://sparksandflames.com/files/x86InstructionChart.html)) I would see that the instruction 55 means to push a specific specialized CPU variable into a location in memory where the CPU can retrieve it again later.

So, if the first machine code instruction (55) takes one byte, where will the next instruction be located?

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>08048374</td>
<td>55</td>
</tr>
<tr>
<td>08048375</td>
<td>89 e5</td>
</tr>
<tr>
<td>08048376</td>
<td>83 ec 08</td>
</tr>
<tr>
<td>08048377</td>
<td>83 e4 f0</td>
</tr>
<tr>
<td>08048378</td>
<td>b8 00 00 00</td>
</tr>
<tr>
<td>08048379</td>
<td>29 c4</td>
</tr>
<tr>
<td>08048380</td>
<td>c7 04 24 74</td>
</tr>
<tr>
<td>08048381</td>
<td>84 04 08</td>
</tr>
<tr>
<td>08048382</td>
<td>e8 10 ff ff</td>
</tr>
<tr>
<td>08048383</td>
<td>c9</td>
</tr>
<tr>
<td>08048384</td>
<td>c3</td>
</tr>
</tbody>
</table>

The answer: At address 08048345. Each byte has its own address, and memory is numbered sequentially.

So... why is the third instruction located at address 8048347? Shouldn’t it be 8048346?

The instruction at 8048345 (89 e5) is two bytes. So, this instruction uses addresses 8048345 and 8048346. Similarly, we see that the next instruction consumes three bytes, so the following instruction is stored at address 804834a. (Recall, in hexadecimal, the number after 9 is a.)

What would you guess that last machine language instruction (c3) does? If you guessed "finishes the program and returns to the operating system", you are correct!

So, our program residing in main memory is shown below in Figure 3.1. The numbers to the left (e.g., 08048344) are the addresses in main memory. The contents of the boxes show the values stored at the memory locations. So, memory location 08048344 holds the value 0x55.

So, our program residing in main memory is shown below. The numbers to the left (e.g., 08048374) are the addresses in main memory. The contents of the boxes show the values stored at the memory locations. So, memory location 08048374 holds the value 0x55.
I’m sure you would agree: *Machine language is fun!* (Don’t worry… we won’t see a lot of machine language.)

2. **Processor Registers** The CPU fetches an instruction (like the instruction 0x55 at address 0x08048374 in Figure 3.1), decodes the instruction, and then executes the instruction. After the CPU executes an instruction, it fetches the next instruction. The sequence of steps *fetch-decode-execute* repeats until the program is finished.
How does the CPU keep track of which instruction it is at in memory?

The CPU has some specialized variables that it uses to execute programs. Unlike variables that you declare in, say, a C program, these CPU variables are actually implemented in high-speed hardware called registers. The x86 has 16 of these variables, each already named and each intended for a specific purpose. Each register holds 32 bits.

The most important CPU variable is eip. Memorize this name. This variable is the Instruction Pointer 3: This variable holds the address of the next instruction the CPU intends to execute.

Many text books refer to the instruction pointer as the program counter. These two terms are synonyms.

So, if the executable program shown at the top of the prior page is loaded into memory, the address 08048344 is placed in eip.

Let’s add two more registers to our repertoire. You should also memorize these (along with eip):

- esp: The CPU reserves a section of memory, called the stack, to store values that the CPU might want to retrieve later. The esp variable is used to store the address of the "top" of the stack. The name esp stands for extended stack pointer, but it is usually just called the stack pointer.

- ebp: This variable is called the base pointer. This CPU variable is used to point to the "bottom" of the stack. (To be more precise, we will see later that ebp actually points to the very first address after the bottom of the stack.)

3. Assembly Language

Okay, we want to see precisely what is going on in the CPU, but we can’t keep our sanity if we have to look at pictures like this:

This picture shows machine language. Midshipmen do not like machine language, and if there is any recurring theme at USNA, it is the need to keep midshipmen happy. But unless we can get “into” the CPU, we don’t really know what is going on. So…we have to deal with the CPU instructions (machine language) without dealing with bits (or hexadecimal).

The answer is to use assembly language!

Remember that in assembly language, each machine instruction is replaced by an “English-like” word or mnemonic.

Looking at the machine code above, we mentioned that the last instruction, c3, had us finish execution and return to the operating system. In assembly language, this instruction maps to ret (short for return).

---

3 The “e” in eip stands for “extended.” The original instruction pointer was 16 bits, but it was later extended to 32 bits.
There is a one-to-one mapping between the assembly language instructions and the machine language instructions. Thus, assembly language is just an easier way to read machine language.

Our simple program:

```c
#include<stdio.h>
int main( )
{
    printf( "Go Navy!\n" );
}
```

is shown below in assembly language. For convenience, the machine language is repeated in the middle. The assembly language appears on the right.

<table>
<thead>
<tr>
<th>address</th>
<th>machine language instruction</th>
<th>assembly language instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>08048344</td>
<td>push ebp</td>
<td></td>
</tr>
<tr>
<td>8048344:</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>8048345:</td>
<td>89 e5</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>8048347:</td>
<td>83 ec 08</td>
<td>sub esp, 0x8</td>
</tr>
<tr>
<td>804834a:</td>
<td>83 e4 f0</td>
<td>and esp, 0xfffffffff0</td>
</tr>
<tr>
<td>804834d:</td>
<td>b8 00 00 00 00 00</td>
<td>mov eax, 0x0</td>
</tr>
<tr>
<td>8048352:</td>
<td>29 c4</td>
<td>sub esp, eax</td>
</tr>
<tr>
<td>8048354:</td>
<td>c7 45 fc 07 00 00 00</td>
<td>mov DWORD PTR [ebp-4], 0x7</td>
</tr>
<tr>
<td>804835b:</td>
<td>c7 45 fc d1 07 00 00</td>
<td>mov DWORD PTR [ebp-4], 0x7d1</td>
</tr>
<tr>
<td>8048362:</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>8048363:</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

Now… you might be looking at the assembly language and thinking: “That’s easier???” Well, it will take some getting used to, but you will pick it up fast. For example, what do you think _mov_ means? If you guessed _move_, you’re right. If you’re guessing that _sub_ means _subtract_, right again! And note that we see the CPU variables _ebp_ and _esp_ that we talked about earlier flying around in the code.

Some assembly language instructions just specify an operation and do not have any operands, e.g.:

```
leave
ret
```

Some assembly language instructions specify an operation and a single operand, e.g.:

```
push ebp
```

Some assembly language instructions specify an operation and two operands, e.g.:

```
mov ebp, esp
sub esp, eax
```

For the two-operand assembly language instructions, it is important to note that first operand is the destination and the second operand is the source. So the instruction

```
mov ebp, esp
```

means: “Move the value of _esp_ to _ebp_”

and the instruction

```
sub esp, eax
```

means: “Subtract the value of _eax_ from _esp_ (so that _esp_ is reduced by the amount _eax_).”

Here is a cheat sheet of common assembly language instructions. It is suggested that you _not_ grapple with this
cheat sheet right now. Rather, it is suggested that you refer back to it when you later encounter an assembly language instruction that is unfamiliar.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
<th>Example</th>
<th>Explanation of the example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>move</td>
<td>mov DWORD PTR [esp],0x804848a</td>
<td>Place the value 0x804848a in the location specified by the address in the esp register.</td>
</tr>
<tr>
<td>cmp</td>
<td>compare</td>
<td>cmp DWORD PTR [ebp],0x4</td>
<td>Compare the value 4 to the value stored in the address contained within the ebp register.</td>
</tr>
<tr>
<td>jne</td>
<td>jump if not equal</td>
<td>jne 0x804839f</td>
<td>This instruction will always follow a comparison (cmp). If the two items in the prior comparison were not equal, then jump to the instruction stored at address 0x804839f.</td>
</tr>
<tr>
<td>jle</td>
<td>jump if less than or equal</td>
<td>jle 0x804839f</td>
<td>This instruction will always follow a comparison (cmp). If the first item in the prior comparison is less than the second item in the prior comparison, then jump to the instruction stored at address 0x804839f. For example, if the prior comparison was cmp DWORD PTR [ebp],0x4, then if the value stored in the address pointed to by the ebp register is less than or equal to 4, we would jump to the instruction stored at address 0x804839f.</td>
</tr>
<tr>
<td>jmp</td>
<td>jump</td>
<td>jmp 0x804839f</td>
<td>Jump to the instruction located at address 0x804839f.</td>
</tr>
<tr>
<td>inc</td>
<td>increment</td>
<td>inc DWORD PTR [eax]</td>
<td>Increment the value stored at the memory location contained within the eax register by one.</td>
</tr>
</tbody>
</table>

4. **Program Autopsy: Case 1**  Now, to really see what is going on, we can run this program one line at a time, and, at each step in the process, examine the CPU’s special variables (the registers) and any other memory locations we care to. We can step through an executable file and examine registers and memory by using a debugger. A debugger is a program that allows you to test and examine other programs. Here’s how to get started:

**Step 1.** Start up VMware Workstation, navigate to your work directory be entering: cd work. Then using nano, open a new file named ch3demo.c by entering: nano ch3demo.c. Then enter the following program:

```c
#include<stdio.h>
int main( )
{
    int x = 7;
    x = 2001;
}
```

Compile the program and ensure that it contains no syntax errors (recall that to compile your program you enter gcc ch3demo.c). Then run the program (by entering: ./a.out). You should see the results shown in the screen capture below.
Wait – what happened? This program is very simple – it merely stores and changes the value of the variable \( x \) in memory. It doesn’t get input from the user (\texttt{scanf}), and it doesn’t display output either (\texttt{printf}), so there’s not much to see “on the outside” when the program is run. But what’s happening “on the inside” (in memory)? The debugger will help us figure that out.

**Step 2.** Start the debugger by entering the following seven lines of code. Enter the commands below (don’t include the comments! – those are provided just to explain what is accomplished by each command). You should look at the screen capture below to follow along as you are entering commands. Your screen should look the same!

\begin{verbatim}
gcc -g ch3demo.c // The \texttt{-g} part of this is new! Adding this provides some extra functionality // for the debugger.
gdb -q ./a.out // \texttt{gdb} is the name of the debugger. So, we are running the debugger on the // executable file named \texttt{a.out}
set dis intel // This displays the assembly code in standard Intel lingo
list // This repeats the source code for convenience
disassemble main // This shows the assembly code starting with the line that has \texttt{main}
break main // This sets a “breakpoint” at \texttt{main}. So, when we run the program, it will stop // at the first line of executable code that follows the line that has \texttt{main}
r
run // This starts executing the program up to the first line of executable code // that follows the line that has \texttt{main}.
\end{verbatim}
So, the program’s execution is "frozen" at the first real line of code (the first line of executable code that follows the line that has `main`). So... where did the program freeze?

**Practice Problem**

In the screen capture above, what assembly language instruction did the program stop at—i.e., what is the next instruction that will execute, and where in main memory is this instruction stored?

Solution:

You might be wondering: What about all the instructions before this one? Does that matter? The answer: This is code that the compiler has generated to set up memory for the program. We can safely ignore this for now.

Since we know that `leave` and `ret` are basically mop-up operations, we really only have to concentrate on the two lines:

0x08048354 <main+16>:   mov DWORD PTR [ebp-4],0x7
0x0804835b <main+23>:   mov DWORD PTR [ebp-4],0x7d1

What do we make of these two cryptic lines? To find out, we introduce two powerful commands: the `info` command and the `examine` command.

**Step 3. The info command.** To look at the value of a register, we use the `info (i)` command. For example, to examine the `eip` register, you would enter the command

```
ir eip
```

and to examine the `esp` register, you would enter the command

```
ir esp
```

**Practice Problem**

What is the value stored in the `eip` register? Does this answer make sense?

Solution:

**Step 4. The examine command.** To examine the value stored at a memory location, we use the `examine (x)` command. The format for the `x` command is:

```
x/display_option location we want to display
```

- use `x` for hexadecimal
- use `u` for decimal
- use `i` to display assembly language
- use `s` to display the result as a string of characters

- to see the contents of an address, simply use the address
- to see the contents of an address in a register, use the register name preceded by a dollar sign
So, the command starts with an $x$ followed by a slash. Then we tell the debugger how we would like the memory location contents to be displayed. If we want the value to be displayed in hexadecimal, the display option is $x$. If we want the value to be displayed in decimal, the display option is $u$.

If we want to display the contents of a memory location, we simply supply the memory location as the last argument. If we instead want to see the contents of a memory location whose address is in a register, we supply the register name preceded by a dollar sign.

By default, the debugger displays 4 bytes for its answer. If we only want to display a single byte, we place the letter $b$ right after the display option. To display two bytes, we place the letter $h$ right after the display option. To display 4 bytes, we place the letter $w$ after the display option. To display 8 bytes, we place the letter $g$ after the display option.

To summarize the examine command:

<table>
<thead>
<tr>
<th>x/</th>
<th>location we want to display</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first position specifies the format for the display. Use this table:

<table>
<thead>
<tr>
<th>x</th>
<th>hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>decimal</td>
</tr>
<tr>
<td>i</td>
<td>assembly language</td>
</tr>
<tr>
<td>s</td>
<td>string</td>
</tr>
</tbody>
</table>

The second position specifies the number of items we want to display. Use this table:

<table>
<thead>
<tr>
<th>b</th>
<th>byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>half-word (2 two bytes)</td>
</tr>
<tr>
<td>w</td>
<td>word (four bytes)</td>
</tr>
<tr>
<td>g</td>
<td>giant (eight bytes)</td>
</tr>
</tbody>
</table>

To see the contents of a memory location, simply place the memory location here.

To see the contents of an address whose location is stored in a register, place the register here, preceded by a dollar sign (e.g., $\texttt{esp}$)

If the foregoing paragraphs have you bewildered, do not fear! We will do many examples!
Practice Problem

Refer to the picture shown on pages 80-81. What should be printed out by each of the following commands? In each case, enter the command to confirm your answer.

(a) \texttt{x/xb 0x08048354}

(b) \texttt{x/xb 0x08048355}

(c) \texttt{x/xb 0x08048356}

(d) \texttt{x/xb 0x08048357}

Solution: (a) (b) (c) (d)

The above example is depicted in the extract from Figure 3-1, shown below.

\begin{center}
\begin{tabular}{|c|c|}
\hline
0848354 & c7 \\
0848355 & 45 \\
0848356 & fc \\
0848357 & 07 \\
0848358 & 00 \\
\hline
\end{tabular}
\end{center}

Now, recall that when we use \texttt{b} in the examine command, as in \texttt{x/xb}, the \texttt{b} stands for byte. When we issue the command

\texttt{x/xb 0x08048354}

we are saying: "Show me the contents of main memory, starting at address 0x08048384, but only going one single byte into memory."

If we want to see the contents of memory starting at address 0x08048354, but going two bytes (i.e., a half-word) into memory, we would enter: \texttt{x/xh 0x08048354}.

Practice Problem

What do you think will be displayed by the command: \texttt{x/xh 0x08048354} ? Confirm your result.

Solution:

The x86 processor stores values in so-called \emph{little-endian} order. If I have a four byte quantity, the least significant byte goes in the first address, the second-least-significant byte goes in the next address, and so on. So, if we are to interpret a four-byte quantity as a single unit, the bytes must be reversed. The debugger does this for us automatically.

This is confusing, so let's look at this a little more carefully. As we mentioned, the program is halted at the instruction at address 08048384. We looked earlier at this section of main memory, exploring the results as machine language and assembly language:
So, the assembly language instruction at address 08048354 is

\[ \text{mov DWORD PTR [ebp-4], 0x7} \]

This assembly language instruction is stored in memory locations 08048354 through 0804835a inclusive.

Here is the key point:

The assembly language instruction \( \text{mov DWORD PTR [ebp-4], 0x7} \) is actually equivalent to the machine language

\[ \text{00 00 00 07 fc 45 c7} \]

The question faced by the designers of the x86 was: In what order should we store 00 00 00 07 fc 45 c7 in memory?

The answer for the x86 processor is to store the least significant byte in memory first, and then continue downward. So, the least significant byte (c7) goes into memory first (at address 08048354) then the next-least-significant byte (45) goes into the next address (08048355), and so forth.

The debugger automatically reverses the little-Endian notation for us, restoring the proper order.

**Practice Problem**

What do you think will be displayed by the command: \( \text{x/xw 0x08048354} \)? Confirm your result.

**Solution:**

**Step 5. Using the examine command with registers.**

If we instead want to see the contents of a memory location whose address is in a register, we supply the register name preceded by a dollar sign. So, the command

\[ \text{x/xb $eip} \]
means the following: "The instruction pointer holds an address (specifically, the address of the next instruction to be executed). Go to that address. Then tell me what is stored at that address, but only proceed one byte into memory please."

Practice Problem

What do you think will be displayed by the command: `x/xb $eip` . Confirm your result.

Solution:

The preceding example is explained by the picture below. The command `x/xb $eip` means that we should proceed to the memory location that is contained in the instruction pointer, and read off one byte.

```
eip 08048354 8048354 c7
8048355 45
8048356 fc
8048357 07
8048358 00
8048359 00
804835a 00
804835b c7
45
```

Practice Problem

What do you think will be displayed by the command: `x/xh $eip` . Confirm your result.

Solution:

Practice Problem

What do you think will be displayed by the command: `x/xw $eip` . Confirm your result.

Solution:

Practice Problem

What do you think will be displayed by the command: `x/i $eip` . Confirm your result.

Solution:

Step 6. Wonderful... so what does the program actually do? We mentioned that our program has two lines of code we care about:

```
0x08048354 <main+16>:    mov    DWORD PTR [ebp-4],0x7
0x0804835b <main+23>:    mov    DWORD PTR [ebp-4],0x7d1
```

We know that the `eip` contains the first instruction's address: `0x8048354`. If we were to execute one instruction and then freeze again, the instruction executed would be

```
mov    DWORD PTR [ebp-4],0x7
```

What does this cryptic instruction do?
For starters, the register \texttt{ebp} is the base pointer (which, you may recall from earlier in this chapter, points to the memory address immediately below the bottom of the stack). The stack is a section of memory that our program has available to store any values it needs. The \texttt{esp} register contains the address of the "top" of the stack, and the \texttt{ebp} contains the address below the bottom.

For starters, the register \texttt{esp} is the stack pointer. The stack is a section of memory that our program has available to store any values it needs. The \texttt{esp} register contains the address of the "top" of the stack.

This assembly language instruction means (in plain English):

\begin{quote}
Move the value \texttt{0x7} into the address pointed to by \texttt{ebp-4} (the base pointer, minus 4).
\end{quote}

The base pointer contains an address; this instruction will write the value \texttt{0x00000007} into the address 4 above the address contained in the base pointer.

Let's look at a picture of the bottom of the stack. Suppose the base pointer contained the address \texttt{0xbffff818}. Then that would mean that my program is storing all the information it needs (for example, variables) just above address \texttt{0xbffff818}. See the picture below:

So... If I know the value \texttt{0x00000007} is going to be placed in the address 4 above the \texttt{ebp} in memory, how does that change the image above?  
First, let’s figure out the address where the 7 is placed (\texttt{ebp-4}):

\[
\begin{array}{c}
\texttt{0xbffff818} \\
- \texttt{4} \\
\hline
\texttt{0xbffff814}
\end{array}
\]

That’s not so bad. So the 4-byte value \texttt{0x00000007} is going to begin at address \texttt{0xbffff814}.

Next, we have to remember the order in which those bytes are stored. (If you’re thinking, \textit{Little Endian} – GREAT!)

Remember, little endian order means that the least significant byte goes in the first address, the second-least-significant byte goes in the next address, and so on, so let’s take a look at how that applies to a 4-byte integer.
The integer “7” is represented by the following 4 bytes: 

\[ 0x\ 00\ 00 \ 00 \ 07 \]

In memory, the least significant byte goes in the first address, like this:

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>07</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

To tie it all together - the “big picture,” if you will – the 4 bytes are placed in memory, with the least significant byte beginning at address \(0xbffff814\), like this:

The Stack: The area in main memory that the program has available to store any information it needs.

That’s probably enough pontificating about what will happen when the next instruction is executed... Let’s actually execute a single instruction, and then freeze again! Enter the command:

```
exti```

After you enter this command, you should see:

\[ 0x0804835b \ 5 \quad x = 2001; \]

**Practice Problem**

When you execute a command (as you just did), what happens to the instruction pointer (eip)?

**Solution:**
Practice Problem

What is the value stored in the `eip` register? Does this answer make sense?

Solution:

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Value stored at this address</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048351</td>
<td>00</td>
</tr>
<tr>
<td>8048352</td>
<td>29</td>
</tr>
<tr>
<td>8048353</td>
<td>c4</td>
</tr>
<tr>
<td>8048354</td>
<td>c7</td>
</tr>
<tr>
<td>8048355</td>
<td>45</td>
</tr>
<tr>
<td>8048356</td>
<td>fc</td>
</tr>
<tr>
<td>8048357</td>
<td>07</td>
</tr>
<tr>
<td>8048358</td>
<td>00</td>
</tr>
<tr>
<td>8048359</td>
<td>00</td>
</tr>
<tr>
<td>804835a</td>
<td>00</td>
</tr>
<tr>
<td>804835b</td>
<td>c7</td>
</tr>
<tr>
<td>804835c</td>
<td>45</td>
</tr>
<tr>
<td>804835d</td>
<td>fc</td>
</tr>
<tr>
<td>804835e</td>
<td>d1</td>
</tr>
<tr>
<td>804835f</td>
<td>07</td>
</tr>
<tr>
<td>8048360</td>
<td>00</td>
</tr>
<tr>
<td>8048361</td>
<td>00</td>
</tr>
<tr>
<td>8048362</td>
<td>c9</td>
</tr>
</tbody>
</table>

`08048344 <main>:`

- 8048344: 55
- 8048345: 89 e5
- 8048347: 83 ec 08
- 804834a: 83 e4 f0
- 804834d: b8 00 00 00 00
- 8048352: 29 c4
- 8048354: c7 45 fc 07 00 00 00
- 804835b: c7 45 fc d1 07 00 00
- 8048362: c9
- 8048363: c3

`0x08048352 <main+14>:`

- `sub esp,eax`

`0x08048354 <main+16>:`

- `mov DWORD PTR [ebp-4],0x7`

`0x0804835b <main+23>:`

- `mov DWORD PTR [ebp-4],0x7d1`

`0x08048362 <main+30>:`

- `leave`
Practice Problem

Sketch what you expect the stack to look like after the instruction at address \texttt{0x0804835b} is executed.

Solution:

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{The Stack: The area in main memory that the program has available to store any information it needs} \\
\hline
\textbf{bffff810} \\
\textbf{bffff811} \\
\textbf{bffff812} \\
\textbf{bffff813} \\
\textbf{bffff814} \\
\textbf{bffff815} \\
\textbf{bffff816} \\
\textbf{bffff817} \\
\textbf{bffff818} \\
\hline
\end{tabular}
\end{center}

Let’s execute a single instruction, and then freeze again! Enter the command:

\texttt{nexti}

Practice Problem

What two things happen when \texttt{nexti} is entered?

Solution: 1.

2.

Practice Problem

What should you type to examine memory for the hex values you sketched in Practice Problem 3.18? (Confirm your result!)

Solution:
Practice Problem

What should you type to examine memory for the integer 2001? (Confirm your result!)

Solution:

Congratulations! You've completed your first program autopsy!
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Null</td>
<td>32</td>
<td>20</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>B</td>
<td>96</td>
<td>60</td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Start of heading</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>Start of text</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>End of text</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>End of transmitter</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>Enquiry</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>Acknowledge</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>Audible bell</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>Backspace</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>Horizontal tab</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>Line feed</td>
<td>42</td>
<td>2A</td>
<td>^</td>
<td>74</td>
<td>4A</td>
<td>J</td>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>Vertical tab</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>Form feed</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>Carriage return</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>Shift out</td>
<td>46</td>
<td>2E</td>
<td>,</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>Shift in</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>Data link escape</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Device control 1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>Device control 2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>Device control 3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Device control 4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>Negative acknowledge</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>Synchronous idle</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>End trans. block</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>Cancel</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>End of medium block</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Substitution</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>Escape</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>[</td>
<td>123</td>
<td>7B</td>
<td>(</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>File separator</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
<td>124</td>
<td>7C</td>
<td>]</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>Group separator</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
<td>125</td>
<td>7D</td>
<td>)</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>Record separator</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>Unit separator</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>_</td>
<td>127</td>
<td>7F</td>
<td>_</td>
</tr>
</tbody>
</table>
Appendix: Memory Storage Example

This material (Chapter 3) is the toughest chapter in EC312. Midshipmen in the past have struggled with the Chapter 3 material because it introduces a slew of new concepts (the debugger with its many cryptic new commands, assembly code, a first look at registers and memory organization, etc.), all of which are alien to anything you have seen before in any other USNA class. You should rest assured that with some effort the concepts will solidify. The remainder of this chapter contains an extended memory storage example. While this example will not be covered in class, it is recommended that you take time to work through it. Note that in each case that a question is asked, the correct answer follows.

Suppose I have the following variable declarations in a C program:

```c
int zz = 206578;
char letter1 = 'v';
char letter2 = 'N';
int y = 154;
```

Note:
- decimal $206578_{10}$ is $0x326F2$ in hexadecimal
- character ‘v’ is $0x76$ in hexadecimal (from ASCII table)
- character ‘N’ is $0x4E$ in hexadecimal (from ASCII table)
- decimal $154_{10}$ is $0x9A$ in hexadecimal

Suppose when the program gets compiled with `gcc`, the compiler sets aside storage space in the main memory (RAM) for the program and its variables, and variable `zz` gets stored at the first memory address below, then `letter1`, then `letter2`, then `y`.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Data at that Memory Address (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048374</td>
<td></td>
</tr>
<tr>
<td>0x08048375</td>
<td></td>
</tr>
<tr>
<td>0x08048376</td>
<td></td>
</tr>
<tr>
<td>0x08048377</td>
<td></td>
</tr>
<tr>
<td>0x08048378</td>
<td></td>
</tr>
<tr>
<td>0x08048379</td>
<td></td>
</tr>
<tr>
<td>0x0804837A</td>
<td></td>
</tr>
<tr>
<td>0x0804837B</td>
<td></td>
</tr>
<tr>
<td>0x0804837C</td>
<td></td>
</tr>
<tr>
<td>0x0804837D</td>
<td></td>
</tr>
<tr>
<td>0x0804837E</td>
<td></td>
</tr>
<tr>
<td>0x0804837F</td>
<td></td>
</tr>
<tr>
<td>0x08048380</td>
<td></td>
</tr>
<tr>
<td>0x08048381</td>
<td></td>
</tr>
</tbody>
</table>

1. How many total bytes are used to store these variables in memory?
   
   Answer: 4 bytes for `zz`, 1 byte for `letter1`, 1 byte for `letter2`, 4 bytes for `y`: 10 bytes total

2. What are the actual bit values that will be stored in the memory? Give your answer as hexadecimal values.
   
   Answer:
   - Variable `zz` is an integer, so is stored in 4 bytes (which is 8 hexadecimal digits). In memory, its value looks like: $0x000326F2$
   - Variable `letter1` is stored in one byte (which is 2 hexadecimal digits). In memory, its value looks like: $0x76$
Variable `letter2` is also stored in one byte, and in memory its value looks like: \(0x4E\)

Variable `y` is an integer, so it is stored in 4 bytes (8 hexadecimal digits), and in memory its value looks like: \(0x0000009A\)

3. How will the values be stored in the memory?

Answer:
- `char` values are stored in one byte, so they look as is.
- `int` values are stored in “little endian” format, so the least significant byte is stored FIRST in the memory location, and the most significant byte is stored LAST (this is the reverse order of what you’d think it should be).

The memory values will look as follows:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Memory Address</th>
<th>Data at that Memory Address (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>zz</code></td>
<td>0x08048374</td>
<td>F2</td>
</tr>
<tr>
<td></td>
<td>0x08048375</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>0x08048376</td>
<td>03</td>
</tr>
<tr>
<td></td>
<td>0x08048377</td>
<td>00</td>
</tr>
<tr>
<td><code>letter1</code></td>
<td>0x08048378</td>
<td>76</td>
</tr>
<tr>
<td><code>letter2</code></td>
<td>0x08048379</td>
<td>4E</td>
</tr>
<tr>
<td><code>y</code></td>
<td>0x0804837A</td>
<td>9A</td>
</tr>
<tr>
<td></td>
<td>0x0804837B</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>0x0804837C</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>0x0804837D</td>
<td>00</td>
</tr>
<tr>
<td>Garbage</td>
<td>0x0804837E</td>
<td>10</td>
</tr>
<tr>
<td>bits</td>
<td>0x0804837F</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>0x08048380</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>0x08048381</td>
<td>A3</td>
</tr>
</tbody>
</table>

4. What are the values and addresses of the variables?

Answer:
- `zz` = 206578 (which is \(0x000326F2\) in hex), and the address of `zz` is `0x08048374`
- `letter1` = ‘v’ (which is 76 in hex), and the address of `letter1` is `0x08048378`
- `letter2` = ‘N’ (which is 4E in hex), and the address of `letter2` = `0x08048379`
- `y` = 154 (which is \(0x0000009A\) in hex), and the address of `y` is `0x0804837A`