CS 012 • Handout 1

A Brief Introduction to Computer Organization

wagner@cs.ucr.edu
Department of Computer Science and Engineering
University of California, Riverside

September 25, 2003

Representing Numbers

Consider the interesting fact that we can enumerate and write down — in other words, we can *represent* — every integer quantity, no matter how large, using only ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. That is possible because, in the standard way of writing numbers, the value of a digit depends on its position.

Take, for instance, the quantity represented by the number 1383.[1] The first and third digits (reading from *right to left*) are both 3's but they represent different amounts, namely, 3 units and 3 *hundred* units. Expressed differently, we have

> the quantity represented by 1383 =
> 3 units + 8 × (ten units) + 3 × (hundred units) + 1 × (thousand units).

Notice that the number of units multiplying each digit is *ten* times larger than the previous one, that is to say, each unit is a *power of ten*. This is no coincidence but, alas, has to do with the fact that we have only *ten* digits at our disposal. Let's see if we can understand why.

Say that we have nine objects. We can easily represent that quantity with the number 9. Now, suppose we acquire another object. How can we represent the quantity *ten*? We can if we define a new unit, a super-unit so to speak, equivalent to *ten* regular units. Then, we can say that we have 1 super-unit and 0 regular units, that is, 10.

[1] Note that I'm being careful to distinguish a *quantity* from its *numerical representation*. The reason for this distinction will become clear as we move along.

Suppose, instead, that we have ninety-nine objects. That quantity is represented by the number 99 because we have 9 super-units and 9 regular units. If we acquire another object, how can we represent the quantity *one-hundred*? Once again, we must define a new unit, this time a super-super-unit, equivalent to *ten super-units* or *one hundred regular units*, and we thus represent the total number of objects with the number 100, meaning 1 super-super-unit, 0 super-units and 0 regular units. And so on and so forth for larger quantities.

This whole discussion may seem obvious and useless, but it shows us that there is nothing special about the quantity *ten*. If we were to use a different number of symbols, we would be able to build a number system just as good as the number system based on ten symbols (known, to no one's surprise, as the *decimal system*).

In particular, suppose we decide to use only *two* symbols (known as *bits*, an abbreviation for *bi*nary digi*ts*), say, 0 and 1. How do we count in this so-called *binary system*? We do so by following a strategy analogous to that used in the decimal system:[2]

quantity	number_2
zero	0
one	1
two	?

In order to represent the quantity *two* without using any more symbols, we need to create a new unit, equivalent to the total we have already accumulated (two). We then have 1 super-unit (equivalent to *two* regular units) and 0 regular units. In other words, in the binary system, we represent the quantity *two* with the number 10.

Wait a second, now. Isn't that 10 a *ten*? No, not in the binary system. The *quantity* ten and the *number* 10 are different concepts, equivalent only in the decimal system. That's why I've been very careful above to write quantities using their full-length English names, rather than their numerical representation.

How about the binary representation of the quantity *three*? Well, it's 1 super-unit (two) plus 1 regular unit, so its numerical representation in binary is 11. And *four*? Once again, we have used up all symbols at our disposal in the positions we have so far, so yet again we need to create a new unit, a super-super-unit equivalent to *two* super-units (since our total is now *four*). Hence, the binary representation of *four* is 100: 1 super-super-unit (equivalent to *four* regular units), 0 super-units (equivalent to *two* regular units), and 0 regular units. By reasoning this way, it's not difficult to see that the table above develops into the one shown next.

[2]Note that I'm using a subscript 2 to indicate the fact that the numbers on the table's right column are written in binary. This is a common practice.

quantity	number_2	quantity	number_2	quantity	number_2
zero	0	seven	111	fourteen	1110
one	1	eight	1000	fifteen	1111
two	10	nine	1001	sixteen	10000
three	11	ten	1010	seventeen	10001
four	100	eleven	1011	eighteen	10010
five	101	twelve	1100	nineteen	10011
six	110	thirteen	1101	...	...

Notice how the expression written before for the quantity represented by the decimal number 1383 has an analogous form in the binary system, provided that we make the appropriate changes. For example,

the quantity represented by $10011_2 =$

1 unit + 1 × (two units) + 0 × (four units) + 0 × (eight units) + 1 × (sixteen units) =

one + two + zero + zero + sixteen =

the quantity nineteen.

Remember that we're reading the digits from *right to left*. Notice how each new unit is *twice* as large as the previous unit, just as each unit in the decimal system is *ten* times larger than the previous one.

Now, what if we were to use eight symbols — 0, 1, 2, 3, 4, 5, 6, and 7 — instead of two or ten? Or how about sixteen symbols? The two number systems we'd obtain by making those choices are called the *octal* and the *hexadecimal* number systems, respectively. Any number system with a *base* larger than 10 (that is, with more than ten symbols) requires additional symbols beyond the first ten (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). It's customary in those cases to use letters of the alphabet. Thus, for example, the hexadecimal system uses the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. See if you can understand the material presented so far by trying to explain the table on the next page.

Can you see a simple relation between the binary, octal, and hexadecimal representations of a given quantity? Look, for instance, at the quantity represented by 1387_{10}. Its binary, octal, and hexadecimal representations are 10101101011_2, 2553_8, and $56B_{16}$. Can you see a pattern? Here's a hint: write 10101101011 as 010 101 101 011 (that is, in groups of 3 digits, starting from the right, adding zeros to the left if necessary) and compare with 2553_8. Then write 10101101011 as 0101 0110 1011 (groups of 4 digits, again starting from the right and with zeros on the left as needed) and compare with $56B_{16}$. If you didn't see the pattern, here it is: $\underbrace{010}_{2_8}\,\underbrace{101}_{5_8}\,\underbrace{101}_{5_8}\,\underbrace{011}_{3_8}$ and $\underbrace{0101}_{5_{16}}\,\underbrace{0110}_{6_{16}}\,\underbrace{1011}_{B_{16}}$.

num_{10}	num_2	num_8	num_{16}	num_{10}	num_2	num_8	num_{16}
0	00000	000	00	1375	10101011111	2537	55F
1	00001	001	01	1376	10101100000	2540	560
2	00010	002	02	1377	10101100001	2541	561
3	00011	003	03	1378	10101100010	2542	562
4	00100	004	04	1379	10101100011	2543	563
5	00101	005	05	1380	10101100100	2544	564
6	00110	006	06	1381	10101100101	2545	565
7	00111	007	07	1382	10101100110	2546	566
8	01000	010	08	1383	10101100111	2547	567
9	01001	011	09	1384	10101101000	2550	568
10	01010	012	0A	1385	10101101001	2551	569
11	01011	013	0B	1386	10101101010	2552	56A
12	01100	014	0C	1387	10101101011	2553	56B
13	01101	015	0D	1388	10101101100	2554	56C
14	01110	016	0E	1389	10101101101	2555	56D
15	01111	017	0F	1390	10101101110	2556	56E
16	10000	020	10	1391	10101101111	2557	56F
17	10001	021	11	1392	10101110000	2560	570
...	...	...	...	...	...	...	...

Various quantities, represented in bases 10, 2, 8, and 16, respectively.

Bits, Bytes, and Representation Ranges

We've already seen that a single binary digit is referred to as a *bit*. Another important definition is that of a *byte*, which is a collection of 8 bits in succession. It's also equivalent to 2 hexadecimal digits in succession.

Now, it should be intuitive that the more bits we have, the larger a quantity we can represent. What is the largest integer positive quantity that can be represented with a given number of bits? With 4 bits, for example, we can represent all integers between 0_{10} ($= 0000_2 = 0_{16}$) and 15_{10} ($= 1111_2 = F_{16}$). With a byte, however, we can go as high as 255_{10} ($= 1111\,1111_2 = FF_{16}$). In general, with n bits, we can represent all integers between 0_{10} and $(2^n - 1)_{10}$, including those extremes. Most computers use 32 bits for integers and 64 bits for so-called long integers.

How about negative integers? How can we represent those? A *signed* integer is an integer which can be either positive or negative (or zero, of course). One way to represent signed integers is to use the *left-most* bit to indicated the sign: 0 for positive, 1 for negative.

Of course, we would have to give up representing half the numbers we had before, since we're using one bit for the sign. With 4 bits, for example, we'd have:

num_2	num_{10}	num_2	num_{10}
0000	0	1000	-0
0001	1	1001	-1
0010	2	1010	-2
0011	3	1011	-3
0100	4	1100	-4
0101	5	1101	-5
0110	6	1110	-6
0111	7	1111	-7

One possible way to represent negative 4-bit integers.

This is *not* how computers represent negative integers, however, because in this representation we end up with two distinct ways of representing zero. Instead, the universally adopted representation, called *2's complement*, is as follows:

num_2	num_{10}	num_2	num_{10}
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

2's complement representation of 4-bit integers.

Notice how, in the 2's complement representation, the left-most bit still indicates the sign, but now every represented integer is represented in a unique way. The 'rule' to find the negative of a number represented in 2's complement is simple: flip every bit (that is, replace 0's by 1's and vice-versa), then add 1. Thus, for example, the 8-bit number 11010011_2 is the negative of $(00101100_2 + 1_2)$. But, $00101100_2 + 1_2 = 00101101_2 = 45_{10}$, so $11010011_2 = -45_{10}$.

To recap, an n-bit binary number can represent any *unsigned* integer in the range $[0_{10}, (2^n - 1)_{10}]$ or any *signed* integer in the range $[-(2^{n-1})_{10}, (2^{n-1} - 1)_{10}]$, if we use 2's complement.

Representing Characters

Now that we've seen how it is possible to represent any integer number in whatever base system we care to choose, the next question is how to represent characters, such as the letters of the alphabet or unusual symbols such as $\otimes$, ϕ, ♠, and ♯. The answer is simple: associate each symbol with a particular number and then represent that number instead of the symbol. Of course, we also need some means of recording the fact that these numbers are not numbers per se but merely stand-ins for characters. We'll get back to this question momentarily.

The first and simplest character table associating numbers to characters is the so-called ASCII table, part of which is shown below:

char	code	char	code	char	code	char	code	char	code	char	code
!	33	1	49	A	65	Q	81	a	97	q	113
”	34	2	50	B	66	R	82	b	98	r	114
#	35	3	51	C	67	S	83	c	99	s	115
$	36	4	52	D	68	T	84	d	100	t	116
%	37	5	53	E	69	U	85	e	101	u	117
&	38	6	54	F	70	V	86	f	102	v	118
’	39	7	55	G	71	W	87	g	103	w	119
(	40	8	56	H	72	X	88	h	104	x	120
)	41	9	57	I	73	Y	89	i	105	y	121
*	42	:	58	J	74	Z	90	j	106	z	122
+	43	;	59	K	75	[	91	k	107	{	123
,	44	¡	60	L	76	\	92	l	108	|	124
-	45	=	61	M	77	]	93	m	109	}	125
.	46	¿	62	N	78	ˆ	94	n	110	˜	126
/	47	?	63	O	79	_	95	o	111		
0	48	@	64	P	80	‘	96	p	112		

Part of the ASCII table. Codes are written in the decimal representation.

The complete ASCII table has only 256 entries and, therefore, is not capable of mapping more than that many characters. In order to accommodate languages containing thousands of characters, such as Chinese, and the multitude of symbols other than alphabetic characters, other schemes, or *character sets*, had to be invented. The idea, however, remains the same: attach to each character a different number and then manipulate the number instead of the character.

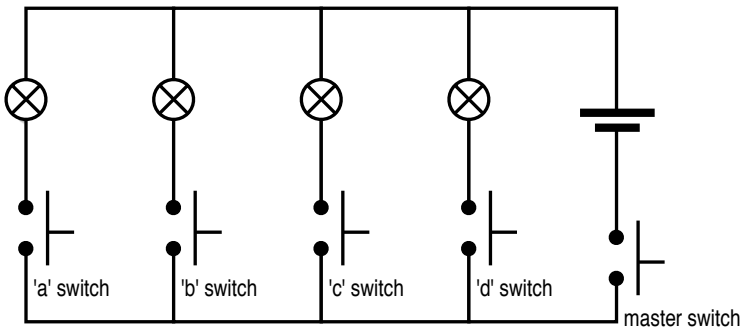
Now let's get back to the question of differentiating regular numbers from numbers used

to represent characters. The distinction is accomplished by defining a *type*. A *char* is a numerical type used to represent characters from the ASCII character set and is represented by a single unsigned byte. An *int* is another numerical type, but it differs from a *char* in the range it represents. As alluded to earlier, most computers use 4 bytes (32 bits) to represent an *int*.

A *Really* Simple Computer

Why have we spent so much time discussing number systems? And, if all number systems are equally good in representing quantities, why at all should we consider alternative systems, such as the binary or the hexadecimal? The answer to both questions rests on the important fact that *any* device capable of detecting (and storing) the difference between two states can be used as a computing device, because we can associate to each of those states one of the two binary digits of the binary system.

Let's look at the storage problem first. Consider the electric circuit shown below. The circles with $\times$'s inside are lights, and the two horizontal bars to the right represent a power source. If the master switch is turned on, the circuit is capable of storing any decimal number from 0 to 15. Here's how: we associate a 'light on' state with the binary digit 1 and a 'light off' state with the binary digit 0. Thus, for example, when switches a, b, and d, but not c, are turned on, we have a situation which can represent the binary number 1101_2. In a way, this circuit functions as a memory or storage device, albeit one which requires continuous power to retain the stored value. In essence, it works much like the RAM memory in your computer.[3]



Now, having a storage device alone isn't sufficient to make a computer; we also need some means of manipulating the numbers stored in memory. In other words, we need an

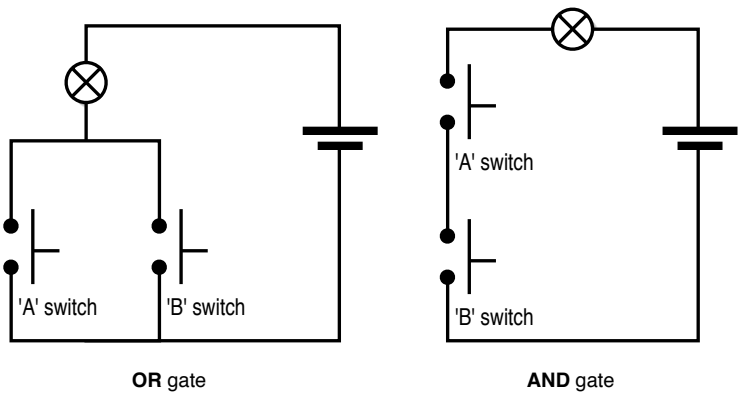
[3]Of course, the RAM memory in your computer is *not* made of light bulbs. In addition, it can store many more numbers than simply 16. We could say that this circuit is a *4-bit* memory device.

arithmetical and logical unit, or *ALU*, which is a device capable of adding and subtracting binary numbers, as well as capable of comparing two binary numbers. Later in your studies in Computer Science, you will learn that ALU's are built from elementary constructs called *logic gates*. A logic gate is a device which accepts a number of input signals (each in the form of high and low voltages which, yet again, we can abstract as a '1' or a '0') and produces one output signal, also in the form of high or low voltages.[4] Each logic gate performs a very specific action. For example, the OR gate outputs 0 (low voltage) only if *all* its inputs are also 0. If at least one of its inputs is a 1, the OR outputs a 1. Similarly, an AND gate outputs a 1 only if *all* its inputs are 1. Otherwise, it outputs a 0. The table below, known as a *truth table*, summarizes the operating conditions for OR and AND gates with two inputs (A and B):

A	B	OR	AND
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

The truth table for OR and AND gates with two inputs.

I'm not going to show you here how ALU's are built from OR's, AND's and other gates, but I will show you how you could build these two kinds of gates using circuits similar to that shown earlier:



Note that in the first case, the light will be turned on (output 1) if either or both switches

[4]High and low, of course, are relative terms. The high volage value in many logic gates is as low as 5.0 volts, which isn't really high.

are turned on, whereas in the second case, the light will turn on *only* when *both* switches are turned on. These are exactly the behaviors of the OR and AND gates, respectively.

Let's now abstract our discussion a little and assume that we have an ALU and some memory. What else do we need? We need input and output devices (such as the keyboard and mouse for input, and the monitor for output) and a more permanent form of storage (such as a hard disk). But, more importantly, we need a means of coordinating, or synchronizing, the actions of the various parts so that the signals travel from one part to another at the appropriate time. This task is performed by the *clock device*, which works very much like a musical metronome, providing an appropriate *tempo* or rhythm to the internal parts of our very simplified computer. Finally, we also need a *central processing unit* (*CPU*) or, more simply, a *processor*. If the clock acts as a metronome, the processor acts like the conductor of an orchestra, telling each member what to do. It is the processor who is responsible for interpreting the instructions fed to the computer in the form of a computer program.

Programming Our Simple Computer

In the very early days of Computer Science, computers had to be programmed in the only language they understand: binary numbers. This language, known as *machine code*, is simply a binary representation of the operations understood by the processor, much like each character can be represented by a number. Soon after getting the first computers to work, however, people realized that programming directly in machine code is extremely inefficient. Thus, people wrote (in machine code) a program known as an *assembler*, which took a more abstract version of the operations the processor could perform and converted it to the corresponding machine code. Now, instead of writing binary numbers, people could write their programs in an *assembly language*, with instructions such as MOV, to move some value from one place in memory to another, and ADD, which adds two numbers in memory.

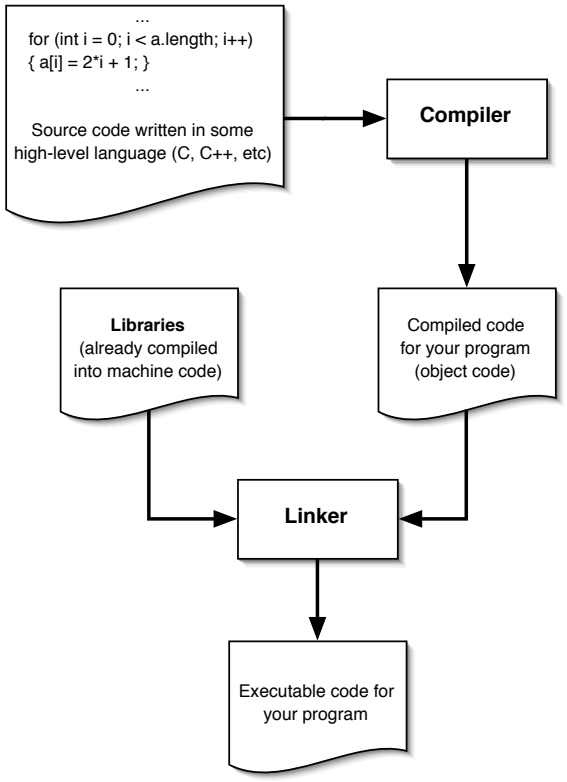
This process of abstracting away the more complicated details by automating the creation of low-level code proved to be extremely successful, so more abstraction came along. This time, people wrote another program (in assembly language), called a *compiler*, which could translate directly into machine code a program written in a very high-level language, such as C, C++, Pascal, etc.

Compilers made life as a computer programmer so much easier that people started to write *libraries*, which are programs specialized in performing certain functions. For example, most languages (C, C++, Java, and so on) have math libraries, string manipulation libraries, graphic libraries, and so on.

But libraries create a problem. When the computer is ready to run your program, how

does it figure out how to piece together your program and all the libraries that it uses? This task is accomplished by yet another program, called the *linker*. The linker's function is basically to solve a jigsaw puzzle, namely, to put together in one contiguous chunk of memory all the code from your program along with the code from the libraries that your program uses. The reason why all this code must be laid out contiguously in memory will become clear in the next section.

We can thus summarize the process of writing a computer program, and getting the computer to accept it, as shown in the figure below.



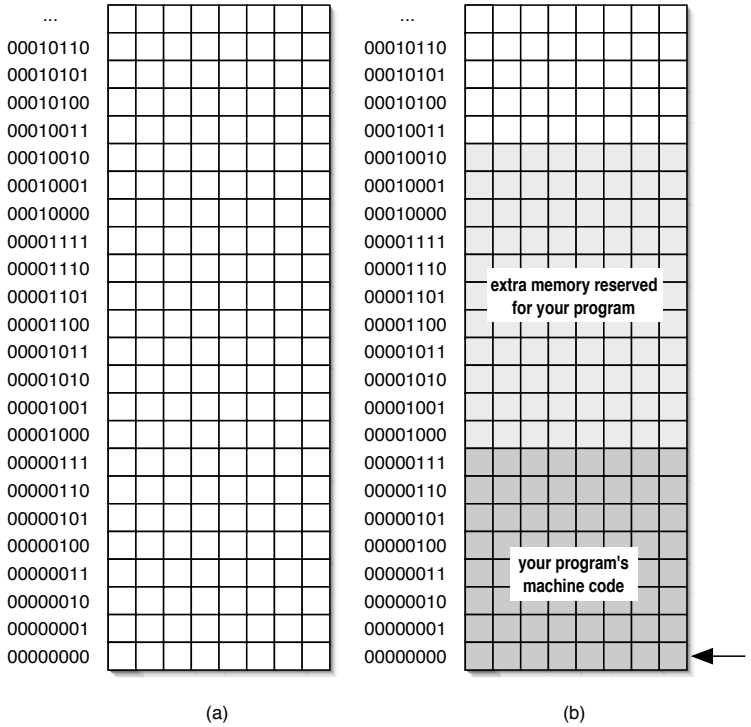
Running A Program

What exactly happens when you run or execute a computer program? Before we answer that question, we need to look with a little more detail at the organization of our simple computer. In particular, we need to look at the way *memory* is organized.

Memory Organization

First, recall that hard disk memory is abundant but extremely slow compared to the transient memory installed on your computer. In fact, most RAM memory chips have access times in the nanosecond range (one-thousandth of one millionth of a second) whereas the most common disk drives, which are mechanical devices, have access times in the millisecond range (one-thousandth of a second). Thus, RAM chips are about a million times faster than hard disks. This explains why you want most, if not all, of your program's code to be resident in memory while it's being executed.

In the simplest case, the computer's memory is a collection of contiguous memory cells, each capable of storing a certain number of bits. Four our purposes here, we'll assume that each memory cell stores one byte, that is, 8 bits. These memory cells are stacked and may number in the billions. In order to be useful, these cells need to be numbered in some way so that the processor can refer to them. This is exemplified in figure (a) below, where the cells are numbered in binary from the bottom up.



Program Execution

Now, imagine that you try to run your program. What happens? Typically, another program running all the time in your computer, the *Operating System* (*OS*), will attempt to reserve a portion of memory for your program, with some amount to spare. This extra amount is going to be used by your program to store and manipulate all the variables your program creates while it's running. This is the situation shown in figure (b) above.

Once this memory allocation is completed and your program's machine code is loaded from the hard disk into its portion of memory, the OS assigns to your program a *program counter*, which is responsible for keeping track of which instruction of your program is to be executed next. Incidentally, it should be clear now why the linker is necessary; it's important to have the entire program code, including the code from the libraries it uses, in one contiguous portion of memory to make it easier for the program counter to perform its function. If there were gaps in your program's code, a single program counter would not suffice. The program counter is indicated in figure (b) above by an arrow. In reality, the program counter is just another cell in memory, holding the *address* of the next instruction to be executed. In the example of figure (b) the address stored in the program counter is 00000000.

Your program is then executed. A simplified version of how that happens is as follows: the processor checks the address stored in the program counter, fetches and executes the instruction stored in that address, then increments the program counter to the next instruction to be executed. This process continues until your program terminates.

For instance, suppose that your program contains the following function:[5]

```
int f(int a, int b)
{
    int x = a + b;
    int y = a - b;
    return x * y;
}
```

Without getting into too many details, the machine code generated by the compiler will contain the following instructions, in this order:

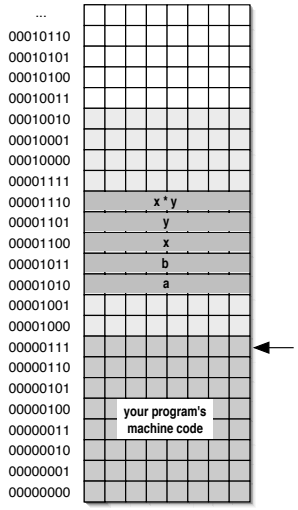
- load a copy of a into memory;
- load a copy of b into memory;

[5]This function computes the value $a^2 - b^2$ performing only one multiplication, rather than two.

- load a copy of a into an ALU's register;[6]
- add b to that ALU's register;
- load a copy of the result from the ALU into memory. This is the value of x;
- load a copy of a into an ALU's register;
- subtract b from that ALU's register;
- load a copy of the result from the ALU into memory. This is the value of y;
- load a copy of x into an ALU's register;
- multiply y into that ALU's register;
- load a copy of the result from the ALU into memory. This is the value of $x * y$;

Before the processor can execute these instructions, however, it needs to store some information about the point in the program where the function got called so that execution can return to that point after the function exits. Then, as those instructions are executed, the various values are stored in memory in the fashion displayed below.[7]

Finally, when the function returns, all those values in memory are effectively lost. You'll learn more about this process when we discuss *function activation frames*, in the next handout. ■



[6] A register is similar to a memory cell, but it resides inside the ALU and is used by the ALU only.

[7] The picture assumes that integers occupy a single byte in memory, which is not the case in real life. This inaccuracy does not affect the conclusions.