Data Representation and Assembly Language Programming The ANT-97 Architecture

Data Representation
and
Assembly Language Programming
The ANT-97 Architecture

Daniel J. Ellard
Penelope A. Ellard

TR-15-98

January 11, 1998

Computer Science Group
Harvard University
Cambridge, Massachusetts

Data Representation
and
Assembly Language Programming

The ANT-97 Architecture

Daniel J. Ellard
Penelope A. Ellard

January 11, 1998

Chapter 1

Data Representation

In order to understand how a computer is able to manipulate data and perform computations, you must first understand how data is represented by a computer.

At the lowest level, the indivisible unit of data in a computer is a *bit*. A bit represents a single binary value, which may be either 1 or 0. In different contexts, a bit value of 1 and 0 may also be referred to as "true" and "false", "yes" and "no", "high" and "low", "set" and "not set", or "on" and "off".

The decision to use binary values, rather than something larger (such as decimal values) was not purely arbitrary– it is due in a large part to the relative simplicity of building electronic devices that can manipulate binary values.

1.1 Representing Integers

1.1.1 Unsigned Binary Numbers

While the idea of a number system with only two values may seem odd, it is actually very similar to the decimal system we are all familiar with, except that each digit is a bit containing a 0 or 1 rather than a number from 0 to 9. (The word "bit" itself is a contraction of the words "binary digit") For example, figure 1.1 shows several binary numbers, and the equivalent decimal numbers.

In general, the binary representation of 2^k has a 1 in binary digit k (counting from the *right*, starting at 0) and a 0 in every other digit. (For notational convenience, the ith bit of a binary number A will be denoted as A_i.)

The binary representation of a number that is not a power of 2 has the bits set

Figure 1.1: Binary and Decimal Numbers

Binary		Decimal
0	=	0
1	=	1
10	=	2
11	=	3
100	=	4
101	=	5
110	=	6
⋮	⋮	⋮
11111111	=	255

corresponding to the powers of two that sum to the number: for example, the decimal number 6 can be expressed in terms of powers of 2 as $1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$, so it is written in binary as 110.

An eight-digit binary number is commonly called a *byte*. In this text, binary numbers will usually be written as bytes (i.e. as strings of eight binary digits). For example, the binary number 101 would usually be written as 00000101– a 101 padded on the left with five zeros, for a total of eight digits.

Whenever there is any possibility of ambiguity between decimal and binary notation, the *base* of the number system (which is 2 for binary, and 10 for decimal) is appended to the number as a subscript. Therefore, 101_2 will always be interpreted as the binary representation for five, and never the decimal representation of one hundred and one (which would be written as 101_{10}).

1.1.1.1 Conversion of Binary to Decimal

To convert an unsigned binary number to a decimal number, add up the decimal values of the powers of 2 corresponding to bits which are set to 1 in the binary number. Algorithm 1.1 shows a method to do this. Some examples of conversions from binary to decimal are given in figure 1.2.

Since there are 2^n unique sequences of n bits, if all the possible bit sequences of length n are used, starting from zero, the largest number will be $2^n - 1$.

Algorithm 1.1 Binary to Decimal

To convert a binary number to decimal.

- Let X be a binary number, n digits in length, composed of bits $X_{n-1} \cdots X_0$.
- Let D be a decimal number.
- Let i be a counter.

1. Let $D = 0$.
2. Let $i = 0$.
3. While $i < n$ do:
   - If $X_i == 1$ (i.e. if bit i in X is 1), then set $D = (D + 2^i)$.
   - Set $i = (i + 1)$.

Figure 1.2: Examples of Conversion from Binary to Decimal

Binary						**Decimal**
00000000	=	0	=	0	=	0
00000101	=	$2^2 + 2^0$	=	$4 + 1$	=	5
00000110	=	$2^2 + 2^1$	=	$4 + 2$	=	6
00101101	=	$2^5 + 2^3 + 2^2 + 2^0$	=	$32 + 8 + 4 + 1$	=	45
10110000	=	$2^7 + 2^5 + 2^4$	=	$128 + 32 + 16$	=	176

1.1.1.2 Conversion of Decimal to Binary

An algorithm for converting a decimal number to binary notation is given in algorithm 1.2.

Algorithm 1.2 Decimal to Binary

To convert a positive decimal number to binary.

- Let X be an unsigned binary number, n digits in length.
- Let D be a positive decimal number, no larger than $2^n - 1$.
- Let i be a counter.

1. Let $X = 0$ (set all bits in X to 0).
2. Let $i = (n - 1)$.
3. While $i \geq 0$ do:
   (a) If $D \geq 2^i$, then
      - Set $X_i = 1$ (i.e. set bit i of X to 1).
      - Set $D = (D - 2^i)$.
   (b) Set $i = (i - 1)$.

1.1.1.3 Addition of Unsigned Binary Numbers

Addition of binary numbers can be done in exactly the same way as addition of decimal numbers, except that all of the operations are done in binary (base 2) rather than decimal (base 10). Algorithm 1.3 gives a method which can be used to perform binary addition.

When algorithm 1.3 terminates, if c is not 0, then an *overflow* has occurred– the resulting number is simply too large to be represented by an n-bit unsigned binary number.

Algorithm 1.3 Unsigned Binary Addition

Addition of unsigned binary numbers.

- Let A and B be a pair of n-bit binary numbers.
- Let X be a binary number which will hold the sum of A and B.
- Let c and $\hat{c}$ be carry bits.
- Let i be a counter.
- Let s be an integer.

1. Let $c = 0$.
2. Let $i = 0$.
3. While $i < n$ do:
   (a) Set $s = A_i + B_i + c$.
   (b) Set X_i and $\hat{c}$ according to the following rules:
      - If $s == 0$, then $X_i = 0$ and $\hat{c} = 0$.
      - If $s == 1$, then $X_i = 1$ and $\hat{c} = 0$.
      - If $s == 2$, then $X_i = 0$ and $\hat{c} = 1$.
      - If $s == 3$, then $X_i = 1$ and $\hat{c} = 1$.
   (c) Set $c = \hat{c}$.
   (d) Set $i = (i + 1)$.

1.1.2 Signed Binary Numbers

The major flaw with the representation that we've used for unsigned binary numbers is that it doesn't include a way to represent negative numbers.

There are a number of ways to extend the unsigned representation to include negative numbers. One of the easiest is to add an additional bit to each number that is used to represent the *sign* of the number– if this bit is 1, then the number is negative; otherwise the number is positive (or vice versa). This is analogous to the way that we write negative numbers in decimal– if the first symbol of the number is a negative sign, then the number is negative, otherwise the number is positive.

Unfortunately, when we try to adapt the algorithm for addition to work properly with this representation, this apparently simple method turns out to cause some trouble. Instead of simply adding the numbers together as we do with unsigned numbers, we now need to consider whether the numbers being added are positive or negative. If one number is positive and the other negative, then we actually need to do subtraction instead of addition, so we'll need to find an algorithm for subtraction. Furthermore, once we've done the subtraction, we need to compare the the unsigned magnitudes of the numbers to determine whether the result is positive or negative!

Luckily, there is a representation that allows us to represent negative numbers in such a way that addition (or subtraction) can be done easily, using algorithms very similar to the ones that we already have. The representation that we will use is called *two's complement* notation.

To introduce two's complement, we'll start by defining, in algorithm 1.4, the algorithm that is used to compute the negation of a two's complement number.

Figure 1.3 shows the process of negating several numbers. Note that the negation of zero is zero.

This representation has several important properties:

- The leftmost (most significant) bit also serves as a sign bit; if 1, then the number is negative, if 0, then the number is positive or zero.
- The rightmost (least significant) bit of a number always determines whether or not the number is odd or even– if bit 0 is 0, then the number is even, otherwise the number is odd.
- The largest positive number that can be represented in two's complement notation in an n-bit binary number is $2^{n-1} - 1$. For example, if $n = 8$, then the largest positive number is $01111111 = 2^7 - 1 = 127$.

Algorithm 1.4 Two's Complement Negation

Negation of a two's complement number.

1. Let $\bar{x}$ = the *logical complement* of x.

   The logical complement (also called the *one's complement*) is formed by flipping all the bits in the number, changing all of the 1 bits to 0, and vice versa.

2. Let $X = \bar{x} + 1$.

   If this addition *overflows*, then the overflow bit is discarded.

By the definition of two's complement, the resulting X is the negation of the original x.

Figure 1.3: Examples of Negation Using Two's Complement

	00000110	=	6
1's complement	11111001		
Add 1	11111010	=	-6
	11111010	=	-6
1's complement	00000101		
Add 1	00000110	=	6
	00000000	=	0
1's complement	11111111		
Add 1	00000000	=	0

- Similarly, the "most negative" number is -2^{n-1}, so if $n = 8$, then it is 10000000, which is $-2^7 = -128$. Note that the negative of the most negative number (in this case, 128) cannot be represented in this notation.

1.1.2.1 Addition and Subtraction of Signed Binary Numbers

The same addition algorithm that was used for unsigned binary numbers also works properly for two's complement numbers.

	00000101	=	5
+	11110101	=	-11
	11111010	=	-6

Subtraction is also done in a similar way: to subtract A from B, take the two's complement of A and then add this number to B.

The conditions for detecting overflow are different for signed and unsigned numbers, however. If we use algorithm 1.3 to add two unsigned numbers, then if c is 1 when the addition terminates, this indicates that the result has an absolute value too large to fit the number of bits allowed. With signed numbers, however, c is not relevant, and an overflow occurs when the signs of both numbers being added are the same but the sign of the result is opposite. If the two numbers being added have opposite signs, however, then an overflow *cannot* occur.

For example, consider the sum of 1 and -1:

	00000001	=	1	
+	11111111	=	-1	
	00000000	=	0	**Correct!**

In this case, the addition will overflow, but it is not an error, since the result that we get (without considering the overflow) is exactly correct.

On the other hand, if we compute the sum of 127 and 1, then a serious error occurs:

	01111111	=	127	
+	00000001	=	1	
	10000000	=	-128	**Uh-oh!**

Therefore, we must be very careful when doing signed binary arithmetic that we take steps to detect bogus results. In general:

- If A and B are of the same sign, but $A + B$ is of the opposite sign, then an overflow or wraparound error has occurred.
- If A and B are of different signs, then $A + B$ will never overflow or wraparound.

1.1.2.2 Shifting Signed Binary Numbers

Another useful property of the two's complement notation is the ease with which numbers can be multiplied or divided by two. To multiply a number by two, simply shift the number "up" (to the left) by one bit, placing a 0 in the least significant bit. To divide a number in half, simply shift the the number "down" (to the right) by one bit (but do not change the sign bit).

Note that in the case of odd numbers, the effect of shifting to the right one bit is like dividing in half, rounded towards $-\infty$, so that 51 shifted to the right one bit becomes 25, while -51 shifted to the right one bit becomes -26.

	00000001	=	1
Double	00000010	=	2
Halve	00000000	=	0
	00110011	=	51
Double	01100110	=	102
Halve	00011001	=	25
	11001101	=	-51
Double	10011010	=	-102
Halve	11100110	=	-26

1.1.2.3 Hexadecimal Notation

Writing numbers in binary notation can soon get tedious, since even relatively small numbers require many binary digits to express. A more compact notation, called *hexadecimal* (base 16), is usually used to express large binary numbers. In hexadecimal, each digit represents four unsigned binary digits.

Figure 1.4: Hexadecimal and Octal

Binary	0000	0001	0010	0011	0100	0101	0110	0111
Decimal	0	1	2	3	4	5	6	7
Hex	0	1	2	3	4	5	6	7
Octal	0	1	2	3	4	5	6	7

Binary	1000	1001	1010	1011	1100	1101	1110	1111
Decimal	8	9	10	11	12	13	14	15
Hex	8	9	A	B	C	D	E	F
Octal	10	11	12	13	14	15	16	17

Another notation, which is not as common currently, is called *octal* and uses base eight to represent groups of three bits. Figure 1.4 show examples of binary, decimal, octal, and hexadecimal numbers.

For example, the number 200_{10} can be written as 11001000_2, $C8_{16}$, or 310_8.

1.2 Representing Characters

Just as sequences of bits can be used to represent numbers, they can also be used to represent the letters of the alphabet, as well as other characters.

Since all sequences of bits represent numbers, one way to think about representing characters by sequences of bits is to choose a number that corresponds to each character. The most popular correspondence currently is the ASCII character set. ASCII, which stands for the American Standard Code for Information Interchange, uses 7-bit integers to represent characters, using the correspondence shown in table 1.5.

When the ASCII character set was chosen, some care was taken to organize the way that characters are represented in order to make them easy for a computer to manipulate. For example, all of the letters of the alphabet are arranged in order, so that sorting characters into alphabetical order is the same as sorting in numerical order. In addition, different classes of characters are arranged to have useful relations. For example, to convert the code for a lowercase letter to the code for the same letter in uppercase, simply set the 6th bit of the code to 0 (or subtract 32). ASCII is by no means the only character set to have similar useful properties, but it has emerged as

Figure 1.5: The ASCII Character Set

00 NUL	01 SOH	02 STX	03 ETX	04 EOT	05 ENQ	06 ACK	07 BEL
08 BS	09 HT	0A NL	0B VT	0C NP	0D CR	0E SO	0F SI
10 DLE	11 DC1	12 DC2	13 DC3	14 DC4	15 NAK	16 SYN	17 ETB
18 CAN	19 EM	1A SUB	1B ESC	1C FS	1D GS	1E RS	1F US
20 SP	21 !	22 "	23 #	24 $	25 %	26 &	27 '
28 (	29)	2A *	2B +	2C ,	2D -	2E .	2F /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4A J	4B K	4C L	4D M	4E N	4F O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5A Z	5B [	5C \	5D]	5E ^	5F _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6A j	6B k	6C l	6D m	6E n	6F o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z	7B {	7C |	7D }	7E ~	7F DEL

the standard.

The ASCII character set does have some important limitations, however. One problem is that the character set only defines the representations of the characters used in written English. This causes problems with using ASCII to represent other written languages. In particular, there simply aren't enough bits to represent all the written characters of languages with a larger number of characters (such as Chinese or Japanese). Already new character sets which address these problems (and can be used to represent characters of many languages side by side) are being proposed, and eventually there will unquestionably be a shift away from ASCII to a new multilanguage standard[1].

[1]This shift will break many, many existing programs. Converting all of these programs will keep many, many programmers busy for some time.

1.3 Representing Programs

Just as sequences of bits can be used to represent numbers, they can also be used to represent instructions for a computer to perform. Unlike the two's complement notation for integers, which is a standard representation used by nearly all computers, the representation of instructions, and even the set of instructions, varies widely from one type of computer to another.

The ANT architecture, which is the focus of the rest of this document, uses a relatively simple and straightforward representation. Each instruction is exactly 16 bits in length, and consists of several bit fields, as depicted in figure 1.6.

Figure 1.6: ANT Instruction Formats

4 bits	4 bits	4 bits	4 bits
op	des	reg1	reg2
op	des	reg1	4-bit constant
op	reg	8-bit constant	

The first four bits (reading from the left, or high-order bits) of each instruction are called the `op` field. The op field determines what operation the instruction represents. Depending on what the op is, the rest of the instruction may represent the names of registers or constants used by the op.

For example, the instruction 0234_{16} has an op of 0, which corresponds to the operation of addition.[2] With the addition operation, the three remaining 4-bit fields are interpreted as the names of the registers to use; instruction 0234_{16} adds the contents of registers 3 and 4, and places the sum in register 2. (The add instruction and the rest of the ANT instructions are described more fully in the rest of this document.)

1.4 Memory Organization

We've seen how sequences of binary digits can be used to represent numbers, characters, and instructions. In a computer, these binary digits are organized and ma-

[2]The fact that most of the instructions consist of four 4-bit fields makes hexadecimal notation particularly appropriate for expressing ANT instructions.

nipulated in discrete groups, and these groups are said to be the *memory* of the computer.

1.4.1 Units of Memory

The smallest of these groups, on most computers, is called a *byte*. On nearly all currently popular computers a byte is composed of 8 bits.

The next largest unit of memory is usually composed of 16 bits. What this unit is called varies from computer to computer– on smaller machines, this is often called a *word*, while on newer architectures that can handle larger chunks of data, this is called a *halfword*.

The next largest unit of memory is usually composed of 32 bits. Once again, the name of this unit varies– on smaller machines, it is referred to as a *long*, while on newer and larger machines it is called a *word*.

Finally, on the newest machines, the computer also can handle data in groups of 64 bits. On a smaller machine, this is known as a *quadword*, while on a larger machine this is known as a *long*.

1.4.1.1 Historical Perspective

There have been architectures that have used nearly every imaginable word size– from 6-bit bytes to 9-bit bytes, and word sizes ranging from 12 bits to 48 bits. There are even a few architectures that have no fixed word size at all (such as the CM-2) or word sizes that can be specified by the operating system at runtime.

Over the years, however, most architectures have converged on 8-bit bytes and 32-bit longwords. An 8-bit byte is a good match for the ASCII character set (which has some popular extensions that require 8 bits), and a 32-bit word has been, at least until recently, large enough for most practical purposes.

1.4.2 Addresses and Pointers

Each unique byte[3] of the computer's memory is given a unique identifier, known as its *address*. The *address of* a piece of memory is often refered to as a *pointer to* that

[3]In some computers, the smallest distinct unit of memory is not a byte. For the sake of simplicity, however, this section assumes that the smallest distinct unit of memory on the computer in question is a byte.

piece of memory– the two terms are synonymous, although there are many contexts where one is commonly used and the other is not.

The memory of the computer itself is often organized as a large array (or group of arrays) of bytes of memory. In this organization, the address of each byte of memory is simply the index of the memory array location where that byte is stored.

1.4.3 Summary

In this chapter, we've seen how computers represent integers using groups of bits, and how basic arithmetic and other operations can be performed using this representation.

We've also seen how the integers or groups of bits can be used to represent several different kinds of data, including written characters (using the ASCII character codes), instructions for the computer to execute, and addresses or pointers, which can be used to reference other data.

There are also many other ways that information can be represented using groups of bits, including representations for rational numbers (usually by a representation called *floating point*), irrational numbers, graphics, arbitrary character sets, and so on. These topics, unfortunately, are beyond the scope of this chapter.

Chapter 2

An ANT Tutorial

This section is a quick tutorial for ANT assembly language programming and the ANT environment. This chapter covers the basics of ANT assembly language, including arithmetic operations, simple I/O, conditionals, loops, and accessing memory.

2.1 What is Assembly Language?

As alluded to in the previous chapter, computer instructions can be represented as sequences of bits. Generally, this is the lowest possible level of representation for a program– each instruction is equivalent to a single, indivisible action of the CPU. This representation is called *machine language*, and it is the only form that can be "understood" directly by the machine.

A slightly higher-level representation (and one that is much easier for humans to use) is called *assembly language*. Assembly language is very closely related to machine language, and there is usually a straightforward way to translate programs written in assembly language into machine language. (This translation is usually implemented by a program called an *assembler*.) Assembly language is usually a direct translation of the machine language; one instruction in machine language corresponds to one instruction in the assembly language.

Because of the close relationship between machine and assembly languages, each different machine architecture usually has its own assembly language (in fact, a particular architecture may have several), and each is unique[1].

[1]For many years, considerable effort was spent trying to develop a portable assembly language that could generate machine language for a wide variety of architectures. Eventually, these efforts

2.2 Getting Started with Assembly: `add.asm`

To get our feet wet, we'll write an assembly language program named `add.asm` that adds 1 and 2, and stores the result in register `r2`.

2.2.1 Commenting

Before we start to write the executable statements of a program, however, we'll need to write a comment that describes what the program is supposed to do. In the ANT assembly language, any text between a pound sign (`#`) and the subsequent newline is considered to be a comment, and is ignored by the assembler. Good comments are absolutely essential! Assembly language programs are notoriously difficult to read unless they are well organized and properly documented. Therefore, we start by writing the following:

```
# Dan Ellard -- 11/2/96
# add.asm-- A program that computes the sum of 1 and 2,
#       leaving the result in register r2.
# Registers used:
# r2 - used to hold the result.

# end of add.asm
```

Even though this program doesn't actually do anything yet, at least anyone reading our program will know what this program is supposed to do, and who to blame if it doesn't work[2]. Unlike C programs, it is usually appropriate to comment every line, often with seemingly redundant comments. Uncommented code that seems obvious when you write it will be a deep mystery a few hours later. While a well-written but uncommented C program might be relatively easy to read by an experienced programmer, as we will soon see it is not true that even the most well-written assembly code is readable without plentiful and meaningful comments. Some programmers prefer to add comments that echo the steps performed by the assembly instructions in a higher-level language.

We are not finished commenting this program, but we've done all that we can do until we know a little more about how the program will actually work.

---

were abandoned as hopeless.

Some people consider C to be a portable assembly language.

[2]You should put your own name on your own programs, of course; Dan Ellard shouldn't take all the blame.

2.2.2 Finding the Right Instructions

Next, we need to figure out what instructions the computer will need to execute in order to add two numbers. Since the ANT architecture has very few instructions, it won't be long before you have memorized all of the instructions that you'll need, but as you are getting started you'll need to spend some time browsing through the lists of instructions, looking for ones that you can use to do what you want. Documentation for the ANT instruction set can be found in the appendix of this document.

Luckily, as we look through the list of arithmetic instructions, we notice the `add` instruction, which adds two numbers together.

The `add` instruction takes three operands, which appear in the following order:

1. A register that will be used to hold the result of the addition. For our program, this will be `r2`.

2. A register that contains the first number to be added. Therefore, we're going to have to place the value 1 into a register before we can use it as an operand of `add`. Checking the list of registers used by this program (which is an essential part of the commenting) we select `r3`, and make note of this in the comments.

3. A register that holds the second number to be added. We're also going to have to place the value 2 into a register before we can use it as an operand of `add`. Checking the list of registers used by this program we select `r4`, and make note of this in the comments.

We now know how we can add the numbers, but we have to figure out how to place 1 and 2 into the appropriate registers. To do this, we can use the `lc` (load constant value) instruction, which places an 8-bit constant into a register. Therefore, we arrive at the following sequence of instructions:

```
# Dan Ellard -- 11/2/96
# add.asm-- A program that computes the sum of 1 and 2,
#       leaving the result in register r2.
# Registers used:
# r2 - used to hold the result.
# r3 - used to hold the constant 1.
# r4 - used to hold the constant 2.

        lc      r3, 1           # r3 = 1
        lc      r4, 2           # r4 = 2
```

```
        add     r2, r3, r4      # r2 = r3 + r4.

# end of add.asm
```

2.2.3 Completing the Program

These three instructions perform the calculation that we want, but they do not form a complete program. Like C, an assembly language program must contain some additional information that tells the assembler where the program begins and ends. Unlike C, ANT programs always start with the first instruction; there is no `main`. The end of a program is defined in a very different way, however. Similar to C, where the `exit` function can be called in order to halt the execution of a program, the proper way to end an ANT program is with something analogous to calling `exit` in C. Unlike C, however, if you forget to "call `exit`" your program will not gracefully exit when it reaches the end of the `main` function. Instead, it will blunder on through memory, interpreting whatever it finds as instructions to execute. Generally speaking, this means that if you are lucky, your program will crash immediately; if you are unlucky, it will do something destructive and then crash.

The way to tell ANT that it should stop executing your program, and also to do a number of other useful things, is with a special instruction called `sys`. The `sys` instruction suspends the execution of your program and starts execution of the system. The system then looks at the second argument to `sys` to determine what it is that your program is asking it to do.

In this case, what we want is for the operating system to do whatever is necessary to exit or halt our program. Looking in table A.1.2, we see that this is done by calling the `sys` instruction with zero as the second argument (with the halt syscall, the first argument is unused, although with other syscalls it is used to pass an argument to or return a value from the system).

```
# Dan Ellard -- 11/2/96
# add.asm-- A program that computes the sum of 1 and 2,
#       leaving the result in register r2.
# Registers used:
# r2 - used to hold the result.
# r3 - used to hold the constant 1.
# r4 - used to hold the constant 2.

        lc      r3, 1           # load 1 into r3.
        lc      r4, 2           # load 2 into r4.
```

```
        add     r2, r3, r4      # r2 = r3 + r4.

        sys     r0, 0           # Halt - end execution.

# end of add.asm
```

2.2.4 The Format of ANT Assembly Programs

As you read `add.asm`, you may notice several formatting conventions– all the lines that contain instructions are indented, and each line contains at most one instruction. These conventions are *not* simply a matter of style, but are actually part of the definition of the ANT assembly language.

The first rule of ANT assembly formatting is that instructions must be indented. Comments do not need to be indented, but all of the code itself must be. The second rule of ANT assembly formatting is that only one instruction can appear on a line. (There are a few additional rules, but these will not be important until section 2.5.1.)

Unlike C, where the use of whitespace and formatting is largely a matter of style, in ANT assembly language some use of whitespace is required.[3]

2.3 Using ANT

At this point, we should have a working program. Now, it's time to try running it and see what happens.

Before running the program, we must *assemble* it. The assembler translates the program from the assembly language representation to the machine language representation. The assembler for ANT is called `aa`, so the appropriate command would be:

```
% aa add.asm
```

This will create a file named `add.ant` that contains the ANT machine-language representation of the program in `add.asm`.

Now that we have the assembled version of the program, we can test it by loading it into the ANT debugger in order to execute it. The name of the ANT debugger is `ad`, so to run the debugger, use the `ad` command followed by the name of the

[3] CS50 students may find it a useful exercise to enumerate the kinds of C constructs whose meaning can be altered by the addition or deletion of whitespace.

machine language file to load. For example, to run the program that we just wrote and assembled:

```
% ad add.ant
```

After starting, the debugger will display the following prompt: `>>`. Whenever you see the `>>` prompt, you know that the debugger is waiting for you to specify a command for it to execute.

Once the program is loaded, you can use the `r` (for *run*) command to run it:

```
>> r
```

The program runs, and then the debugger indicates that it is ready to execute another command. Since our program is supposed to leave its result in register `r2`, we can verify that the program is working by asking the debugger to print out the contents of all of the registers using the `p` (for *print*) command, to see if it contains the result we expect:

```
>> p
 r01  r02  r03  r04  r05  r06  r07  r08  r09  r10  r11  r12  r13  r14  r15
  00   03   01   02   00   00   00   00   00   00   00   00   00   00   00
   0    3    1    2    0    0    0    0    0    0    0    0    0    0    0
```

The `p` command displays the contents of each register. The first line lists the register names. The following line lists the value of each register in hexadecimal, and the last line lists the same number in decimal.

`ad` includes a number of features that will make debugging your ANT assembly language programs much easier. Type `h` at the `>>` prompt for a full list of the `ad` commands, or consult the manual page.

2.4 Reading and Printing: `add2.asm`

Our program to compute $1+2$ is not particularly useful, although it does demonstrate a number of important details about programming in ANT assembly language and the ANT environment. For our next example, we'll write a program named `add2.asm` that computes the sum of two numbers specified by the user at runtime, and displays the result on the screen.

The algorithm this program will follow is:

1. Read the two numbers from the user. We'll need two registers to hold these two numbers. We can use `r3` and `r4` for this.

2. Compute their sum. We'll need a register to hold the result of this addition. We can use `r2` for this.

3. Print the sum, followed by a newline.

4. Exit. We already know how to do this, using `sys`.

The only parts of the algorithm that we don't know how to do yet are to read the numbers from the user, and print out the sum. Fortunately, both of these operations can be done with `sys`. Looking again in Table A.1.2, we see that `sys 5` can be used to read an integer into a register, and `sys 2` can be used to print out the integer stored in a register.

For formatting purposes, we also want to print a newline after printing out the sum. We can use `sys 3` to print out a character.

This gives the following program:

```
# Dan Ellard -- 11/2/96
# add2.asm-- A program that computes and prints the sum
#       of two numbers specified at runtime by the user.
# Registers used:
# r2 - used to hold the result.
# r3 - used to hold the first number.
# r4 - used to hold the second number.
# r5 - used to hold the constant '\n'.

        sys     r3, 5           # read first number into r3
        sys     r4, 5           # read second number into r4
        add     r2, r3, r4      # compute the sum r2 = r3 + r4.
        sys     r2, 2           # print contents of r2.

        # Print out a newline
        lc      r5, '\n'        # load a newline character into r5
        sys     r5, 3           # print contents of r5

        sys     r0, 0           # Halt

# end of add2.asm.
```

2.5 Strings: `hello.asm`

The next program that we will write is the "Hello World" program. Looking in table A.1.2 once again, we note that there is a `sys` call to print out a string. All we need to do is to put the address of the string we want to print into the source register (the first argument), and execute `sys` *reg*`, 5`. The only things that we don't know how to do are how to define a string, and then how to determine its address.

The string `"Hello World"` cannot be part of the executable part of the program (which contains all of the instructions to execute), which is called the *instruction segment* or *text segment*. Instead, the string should be part of the *data* used by the program, which is stored in the *data segment*.

To put something in the data segment, all we need to do is to put a `.data` before we define it. Every `.data` command can be followed by up to eight (8) bytes of data. Data is put in the data segment starting at memory location zero. Each byte is put in the next consecutive memory location. Data is loaded into memory at assembly time. You will have to be careful not to overwrite your data during run-time.

ANT programs must have all of the `.data` items defined at the end of the program, after the special label `_data_`. The `_data_` label indicates to the assembler that all subsequent items are data.

2.5.1 Labels

A *label* is a symbolic name for an address in memory. In ANT assembler, a *label definition* is an identifier (following the same conventions as C identifiers) followed by a colon.

Labels must be the first item on a line, and must begin in the "zero column" (immediately after the left margin). Label definitions *cannot* be indented, but all other non-comment lines *must* be.

Since labels must begin in column zero, only one label definition is permitted on each line of assembly language, but a location in memory may have more than one label. Giving the same location in memory more than one label can be very useful. For example, the same location in your program may represent the end of several nested "if" statements, so you may find it useful to give this instruction several labels corresponding to each of the nested "if" statements.

When a label appears alone on a line, it refers to the following memory location. This is often good style, since it allows the use of long, descriptive labels without disrupting the indentation of the program. It also leaves plenty of space on the line

for the programmer to write a comment describing what the label is used for, which is very important since even relatively short assembly language programs may have a large number of labels.

The following program is an example of how to use labels and treat characters in memory as strings:

```
# Dan Ellard -- 11/2/96
# hello.asm-- A "Hello World" program.
# Registers used:
#       r2      - holds the address of the string

        lc      r2, $str_data   # load the address of the string into r2
        sys     r2, 4           # Print the characters in memory
        sys     r0, 0           # Halt

# Data for the program:
_data_:

str_data:
        .data   'H', 'e', 'l', 'l', 'o', ' '
        .data   'W', 'o', 'r', 'l', 'd', '\n', 0

# end of hello.asm
```

The label `str_data` is the symbolic representation of the memory location where the string begins in data memory.

Note that strings in ANT must be terminated by a 0 byte, as in C.

2.6 Conditional Execution: `larger.asm`

The next program that we will write will read two numbers from the user, and print out the larger of the two. The algorithm for this program is exactly the same as the one used by `add2.asm`, except that we're computing the maximum rather than the sum of two numbers.

Browsing through the instruction set again, we find a description of the ANT branching instructions. These allow the programmer to specify that execution should *branch* (or *jump*) to a location other than the next instruction. These instructions allow conditional execution to be implemented in assembly language (although in not nearly as clean a manner as higher-level languages provide).

In ANT assembler, there are three branching instructions: `bgt`, `beq` and `jmp`.

The `bgt` instruction takes three registers as arguments. If the number in the second register is larger than the number in the third, then execution will jump to the location specified by the first; otherwise it continues at the next instruction.

The `beq` instruction is similar to the `bgt` instruction, except that the branch occurs if the second and third registers contain the same value.

The `jmp` instruction takes two arguments, a register and an unsigned 8-bit constant. Execution jumps to the location specified by the constant (the register is ignored).

2.6.1 Branching Using Labels

Using the branching instructions and labels we can do what we want in the `larger.asm` program. Since the branching instructions take a register containing an address as their first argument, we need to somehow load the address represented by the label into a register. We do this by using the `lc` command. The `larger.asm` program illustrates how this is done.

```
# Dan Ellard -- 11/2/96
# larger.asm-- A program that computes and prints the larger
#       of two numbers specified at runtime by the user.
# Registers used:
# r2 - used to hold the first number.
# r3 - used to hold the second number.
# r4 - used to hold the larger of r2 and r3.
# r5 - used to hold the address of the label "r2_larger"
# r6 - used to hold the a "newline" character

        sys     r2, 5           # read a number into r2
        sys     r3, 5           # read a number into r3

        # put the larger of r2 and r3 into r4
        lc      r5, $r2_larger  # put the address of r2_larger into r5

        bgt     r5, r2, r3      # if r2 is larger, branch to r2_larger
        add     r4, r3, r0      # "copy" r3 into r4
        jmp     r0, $endif      # and then branch to endif
r2_larger:
        add     r4, r2, r0      # "copy" r2 into r4
endif:
        sys     r4, 2           # print contents of r4.
```

```
        lc      r6, '\n'        # load a newline character into r6
        sys     r6, 3           # print contents of r6

        sys     r0, 0           # Halt

# end of larger.asm.
```

Since ANT does not have an instruction to *copy* or *move* the contents of one register to another, in order to copy the value of one register to another register we've added 0 to one register and put the sum in the destination register in order to achieve the desired result. (Recall that register `r0` always contains the constant zero.)

2.7 Looping: `multiples.asm`

The next program that we will write will read two numbers A and B, and print out multiples of A from A to $A \times B$. The algorithm that our program will use is shown in the snippet of C code below:

```
int main (void)
{
        int A, B, top, multiple;

        A = GetInteger ();
        B = GetInteger ();

        if ((A == 0) || (B <= 0)) {
                exit (0);
        }
        top = A * B;
        for (multiple = A; multiple <= top; multiple += A) {
                printf ("%d", multiple);
                printf (" ");
        }
        printf ("\n");
        exit (0);
}
```

This algorithm translates easily into ANT assembler.

```
# Dan Ellard -- 11/2/96
```

```
# multiples.asm-- takes two numbers A and B, and prints out
#       all the multiples of A from A to A * B.
#       If B <= 0, then no multiples are printed.
# Registers used:
# r2 - used to hold A.
# r3 - used to hold B.
# r4 - used to store top, the sentinel value A * B.
# r5 - used to store multiple, the current multiple of A.
# r6 - used for address of labels
# r7 - used for holding and printing spaces and a newline

start:
        sys     r2, 5                   # read A into r2
        sys     r3, 5                   # read B into r3

        lc      r6, $A_ok               # r6 = the address of A_ok.
        bgt     r6, r2, r0              # make sure that A != 0.
        bgt     r6, r0, r2
        sys     r0, 0                   # if A == 0, exit.
A_ok:

        lc      r6, $B_ok               # r6 = the address of B_ok.
        bgt     r6, r3, r0              # make sure that B > 0.
        sys     r0, 0                   # if B <= 0, exit.
B_ok:

        mul     r4, r2, r3              # top = A * B.
        add     r5, r2, r0              # multiple = A

loop:
        sys     r5, 2                   # print out multiple (r5)

        lc      r6, $endloop            # r6 = the address of endloop
        beq     r6, r4, r5              # if multiple == top, we're done.
        add     r5, r5, r2              # otherwise, multiple += A.

        lc      r7, ' '
        sys     r7, 3                   # print a space

        jmp     r0, $loop               # go to top of the loop
endloop:
        lc      r7, '\n'
        sys     r7, 3                   # print a newline
```

```
        sys     r0, 0                   # Exit
# end of multiples.asm
```

2.8 Character I/O: echo.asm

Now that we have mastered loops and reading and printing integers, we'll turn our attention to reading and printing single characters. The program that we'll write in this section simply echos whatever you type to it, until EOF (aka end of input) is reached.

The way that EOF is detected in ANT is that when the EOF is reached, the syscall that reads a single character will put a non-zero value into register `r1`. (All of the syscalls place 0 in register `r1` to indicate success, non-zero to indicate failure.)

```
# Dan Ellard - 11/10/96
# Echos input until EOF.
# Register usage:
# r2 - holds each character read in.
# r3 - address of $print.

        lc      r3, $print
loop:
        sys     r2, 6           # r2 = getchar ();
        beq     r3, r1, r0      # if not at EOF, go to $print.
        jmp     r0, $exit       # otherwise, go to $exit.
print:
        sys     r2, 3           # putchar (r2);
        jmp     r0, $loop       # iterate, go back to $loop.
exit:
        sys     r0, 0           # Exit

# end of echo.asm
```

2.9 Load and Store: string_reverse.asm

The next program that we write will read in a string from the user and then print it out backwards. Characters are read until the user enters a newline, or the array used to store the string is exhausted.

The program reads input character-by-character, storing each character in data memory as it is read. Once it reads a newline or the space reserved for the string in

data memory is full, it prints out the characters in reverse order.

The first part of the program reads a string from input, character-by-character. If the character is not a newline, and the user has typed in less than the alloted number of characters, the character is stored in memory. Otherwise, the loop that reads the characters exits immediately.

The command for storing the contents of a register in memory is `st`. It takes three arguments: the register whose contents will be stored in memory, the register containing the base address of memory where the information will be stored (the start of the array), and a 4-bit constant (0 .. 15) that represents the offset from the base address (the index of the array). In our example, the character is read into `r4`, so that will be the first argument. The address of the data is represented by the value of `r7`.

The loop that reads characters and stores them to memory looks like:

```
read_loop :
        sys     r4, 6           # Read a character, put in r4
        beq     r5, r4, r8      # if it's a newline, exit read loop
        bgt     r5, r7, r10     # if char_array is full, exit read loop
        st      r4, r7, 0       # store character at r7
        inc     r7, 1           # i++
        jmp     r0, $read_loop  # go to top of loop

end_read:
```

Now that we have the string in memory, we want to print it out backwards. We know that in order to print out a character, it has to be in a register. The command for getting data out of memory and into a register is `ld`, which takes three arguments. The first is the register where the data will go, and, like `st`, the second and third arguments are the base address and offset from the base address where the data is stored in memory.

The code for printing the string in memory backwards is this:

```
        lc      r5, $end_print  # Re-Initialize r5 to end of print loop
        lc      r6, $print_loop # Re-Initialize r6 to start of print loop
        lc      r9, $char_array # r9 is the address of the first byte
                                # in char_array.
print_loop:
        inc     r7, -1          # i--
        bgt     r5, r9, r7      # Have we backed off the end of char_array?
                                # If so, then exit print loop.
```

```
        ld      r4, r7, 0       # load character at r7 into r4
        sys     r4, 3           # Print r4
        jmp     r0, $print_loop
```

The entire program looks like this:

```
# Penny Ellard -- 9/7/97
# string_reverse.asm-- A program that reads a string from the user,
#       then prints out the string in reverse order
# Registers used:
# r4  - hold characters as they are read in and printed out.
# r5  - address - used for conditional branches.
# r6  - address - used for conditional branches.
# r7  - the address of the next byte in char_array to visit.
# r8  - the constant '\n'.
# r9  - address of the start of char_array.
# r10 - the address of the last byte in the char_array.

initialize:
        lc      r5, $end_read   # Initialize r5 to end of read loop
        lc      r6, $read_loop  # Initialize r6 to start of read loop
        lc      r9, $char_array # r9 is the address of the start of char_array
        lc      r8, '\n'        # Initialize r8 to '\n'
        lc      r10, $end_array # Initialize r10 to the address of the
                                # location after the end of the char_array,
        inc     r10, -1         # and decrement r10 so that it is the address
                                # the last location in the char_array.
        add     r7, r9, r0      # r7 starts at the start of char_array

read_loop:
        sys     r4, 6           # Read a character, put in r4
        beq     r5, r4, r8      # if it's a newline, exit read loop
        bgt     r5, r7, r10     # if char_array is full, exit read loop
        st      r4, r7, 0       # store character at r7
        inc     r7, 1           # i++
        jmp     r0, $read_loop  # go to top of loop

end_read:
        lc      r5, $end_print  # Re-Initialize r5 to end of print loop
        lc      r6, $print_loop # Re-Initialize r6 to start of print loop
        lc      r9, $char_array # r9 is the address of the first byte
                                # in char_array.
```

```
print_loop:
        inc     r7, -1          # i--
        bgt     r5, r9, r7      # Have we backed off the end of char_array?
                                # If so, then exit print loop.
        ld      r4, r7, 0       # load character at r7 into r4
        sys     r4, 3           # Print r4
        jmp     r0, $print_loop

end_print:
        sys     r8, 3           # Print a newline

        sys     r0, 0           # Halt

_data_:

                                # enough space for 40 characters:
char_array:
        .data   0, 0, 0, 0, 0, 0, 0, 0
        .data   0, 0, 0, 0, 0, 0, 0, 0
        .data   0, 0, 0, 0, 0, 0, 0, 0
        .data   0, 0, 0, 0, 0, 0, 0, 0
        .data   0, 0, 0, 0, 0, 0, 0, 0
end_array:

# end of string_reverse.asm
```

Note that there is a way to write this program that uses about half the number of memory accesses (`ld` and `st` are the only commands in ANT that access memory). If we initialized `r7` to start at the *end* of `char_array`, and decremented it, we could then use the `print_string` system call, instead of loading each character and printing it out one at a time. You can try this, if you like; just make sure your string is 0-terminated and that you don't back up past the start of `char_array`!

2.10 Putting It All Together: `atoi.asm`

The next program that we'll write will look at a a line of text in memory, interpret it as an integer, and then print it out.

2.10.1 atoi-1

We will use a string in memory, and we know how to print out a number, so all we need is an algorithm to convert a string into a number. We'll start with the algorithm given in the snippet of C code shown below.

For our algorithm, we will take advantage of the fact that in ASCII, the numbers that represent the digits 0 through 9 are arranged consecutively, starting at '0'. Therefore, for any ASCII character x, the number represented by x is simply x - '0'.

```
int atoi (char *str)
{
        int sum = 0;
        int i;

        for (i = 0; str [i] != '\0'; i++) {
                sum *= 10;
                sum += (str [i] - '0');
        }
        return (sum);
}
```

The code for this algorithm then is simply:

```
# Register usage:
# r3 - used as scratch space to load each byte into.
# r4 - used to hold the sum.
# r5 - the address of the next byte to load.
# r6 - the location of the end of the main loop.
# r7 - used to hold the constant 10.
# r8 - used to hold the constant '0'.
# r9 - used to hold the constant '\n'.

        lc      r4, 0                   # Initialize sum to 0.
        lc      r5, $string_start       # Start at beginning of string.
        lc      r6, $end_sum_loop       # Location of end of the loop.
        lc      r7, 10                  # Initialize r7 to 10.
        lc      r8, '0'                 # Initialize r8 to '0'.

sum_loop:
        ld      r3, r5, 0               # load the byte *str into r3,
        beq     r6, r3, r0              # if r3 == 0, branch out of loop.
        mul     r4, r4, r7              # r4 *= 10.
        sub     r3, r3, r8              # r3 -= '0'.
```

```
        add     r4, r4, r3              # sum += r3.
        inc     r5, 1                   # increment str to the next char,
        jmp     r0, $sum_loop           #  and repeat the loop.
end_sum_loop:

        sys     r4, 2                   # print out the number

        lc      r9, '\n'                # put newline into r9
        sys     r9, 3                   # print out a newline

        sys     r0, 0                   # halt

_data_:

string_start:
        .data   '1', '0', '5', 0
```

2.10.2 More Error Checking of `atoi`

Although the algorithm used by `atoi-1` seems reasonable, it actually has several serious flaws. The first problem is that this routine cannot handle negative numbers. We can fix this easily enough by looking at the very first character in the string, and doing something special if it is a `'-'`. The easiest thing to do is to introduce a new variable to represent the sign of the number. If the number is positive, then the variable will be 1, and if negative then the variable will be -1. This makes it possible to leave the rest of the algorithm intact, and then simply multiply the result by the new variable in order to get the correct sign on the result at the end.

While this algorithm is better than the one used by `atoi-1.asm`, it is by no means free of bugs. The next problem that we must consider is what happens when *str* does not point to a proper string of digits, but instead points to a string that contains erroneous characters.

If we want to mimic the behavior of the UNIX `atoi` library function, then as soon as our program encounters any character that is not a digit (after an optional `'-'`) it must stop the conversion immediately and return whatever is in *sum* as the result. We can implement this by adding some extra tests on every character that gets processed inside `sum_loop`.

Even after correcting this problem, however, our program still has flaws. The original algorithm is generalized to work with *any* number. Unfortunately, register `r4`, which we use to represent *sum*, can only represent an 8-bit binary number, so it

is easy for the user to type in a number that is too large or too small for this program to deal with. Although there's not much that we can do to *prevent* this problem, we definitely want to *detect* this problem and indicate that an error has occurred.

There are two spots in our routine where an overflow might occur: when we multiply the contents of register `r4` by 10, and when we add in the value represented by the current character.

Detecting overflow during addition and multiplication is not hard, but it does require some care. In the ANT architecture, when multiplication and addition are performed, the result is actually stored in two 8-bit registers, the regular destination register (`des`) and `r1`. `des` contains the low-order 8 bits and `r1` contains the high-order 8 bits of the result. Therefore, if `r1` is non-zero after we do either of these operations, then the result was too large to fit into a single 8-bit word.

Appendix A

The ANT Instruction Set

This appendix gives an overview of the ANT instruction set and some of the details of the ANT assembler. The exact definition of the ANT instruction set and a specification for how ANT programs are executed are not given here.

A.1 ANT Architecture Overview

The ANT architecture is a load/store architecture; the only instructions that can access memory are the *load* and *store* (and in some sense the *sys*) instructions. All other operations access only registers.

The ANT CPU has 16 registers, named `r0` through `r15`. Register `r0` always contains the constant 0, and register `r1` is used to hold results related to previous operations (described later). `r0` and `r1` are *read-only* and cannot be used as destination registers. The other 14 registers (`r2` through `r15`) are general-purpose registers.

In the description of the instructions, the following notation is used:

des	Must always be a register, but never `r0` or `r1`.
reg	Must always be a register.
src1	Must always be a register.
src2	Must always be a register.
const8	Must be an 8-bit constant (-128 .. 127): an integer (signed), char, or label.
uconst8	Must be an 8-bit constant (0 .. 255): an integer (unsigned) or label.
uconst4	Must be a 4-bit constant integer (0 .. 15).

A.1.1 General Instructions

Op	Operands	Description
`add`	*des, src1, src2*	*des* gets *src1* + *src2*. `r1` gets any overflow from this addition.
`sub`	*des, src1, src2*	*des* gets *src1* - *src2*. `r1` gets any underflow from this subtraction.
`mul`	*des, src1, src2*	Multiply *src1* and *src2*, leaving the low-order byte in register *des* and the high-order byte in register `r1`.
`div`	*des, src1, src2*	Divide *src1* by *src2*, leaving the quotient in register *des* and the remainder in register `r1`.
`beq`	*reg, src1, src2*	Branch to *reg* if *src1* == *src2*. `r1` is set to the address of the instruction following the `beq`.
`bgt`	*reg, src1, src2*	Branch to *reg* if *src1* > *src2*. `r1` is set to the address of the instruction following the `bgt`.
`ld`	*des, src1, uconst4*	Load the byte at *src1* + *uconst4* into *des*. `r1` is unchanged.
`st`	*reg, src1, uconst4*	Store the contents of register *reg* to *src1* + *uconst4*. `r1` is unchanged.
`lc`	*des, const8*	Load the constant *const8* into *des*. `r1` is unchanged.
`jmp`	*reg, uconst8*	Branch unconditionally to the specified constant. *reg* is ignored.
`inc`	*reg, const8*	Add *const8* to the specified register.
`sys`	*reg, code*	Makes a system call. See A.1.2 for a list of the ANT system calls.

Note that for all instructions except `sys`, register `r1` is always updated *after* the rest of the instruction is done, so that it is always safe to use `r1` as a source register for these instructions. (`sys` sets `r0` to 0 before executing the syscall.)

A.1.2 System Calls Handling

All syscalls set `r1` to 0 if successful, and set `r1` to non-zero values to indicate failure.

Service	Code	Description
`halt`	0	Halt the processor.
`dump`	1	Dump core to file ant.core.
`put_int`	2	Print the contents of *reg* as a number.
`put_char`	3	Print the contents of *reg* as an ASCII character.
`put_string`	4	Print the 0-terminated ASCII string that starts at *reg*.
`get_int`	5	Read an integer into *reg*. *reg* must not be `r0` or `r1`. If EOF, `r1` is set to 1. *Does not check for illegal input.*
`get_char`	6	Read a character into *reg*. *reg* must not be `r0` or `r1`. If EOF, `r1` is set to 1.

A.2 The ANT Assembler

A.2.1 Comments

A comment begins with a `#` and continues until the following newline. The only exception to this is when the `#` character appears as part of an ASCII character constant (as described in section A.2.3).

A.2.2 The `_data_` Label

A special label, `_data_`, is used to mark the boundary between the instructions of the program (which must appear before the `_data_` label) and the data of the program (which appear afterward).

The `_data_` label itself should never be referenced by the program.

A.2.3 Constants

Several ANT assembly instructions contain 8-bit or 4-bit constants.

The 8-bit constants can be specified in a variety of ways: as decimal, octal, hexadecimal, or binary numbers, ASCII codes (using the same conventions as C), or labels.

The value of a label is the index of the subsequent instruction in instruction memory for labels that appear in the code, or the index of the subsequent `.data` item for labels that appear in the data.

The 4-bit constants must be specified as unsigned numbers (using decimal, octal, hexadecimal, or binary notation). ASCII constants or labels cannot be used as 4-bit constants, even if the value represented fits into 4 bits.

A.2.4 The `.data` Directive

Name	Parameters	Description
`.data`	*byte1* $\cdots$ *byteN*	Assemble the given bytes (8-bit integers) into the next available locations in the data segment. As many as 8 bytes can be specified on the same line. Bytes may be specified as hex, binary, decimal or C character constants (as described in A.2.3).