Ant-32 Assembly Language Tutorial (for version 3.1.0b)

Ant-32 Assembly Language Tutorial (for version 3.1.0b)

Daniel J. Ellard
and
Penelope A. Ellard

TR-09-02

Computer Science Group
Harvard University
Cambridge, Massachusetts

Ant-32
Assembly Language Tutorial
(for version 3.1.0b)

Daniel J. Ellard, Penelope A. Ellard

May 20, 2002

Contents

Preface

This document contains a brief tutorial for Ant-32 assembly language programming, the assembly language utilities, and a description of the general Ant-32 instruction set architecture. A complete specification of the Ant-32 architecture, including the exception handling mechanisms, the MMU, and related issues, is given in the *The Ant-32 Architecture (version Revision 3.1.0b)*, which is provided as part of the Ant distribution and is also available from the Ant web site (`www.ant.harvard.edu`).

The Ant-32 architecture is a 32-bit RISC architecture designed specifically for pedagogical purposes. It is intended to be useful to teaching a broad variety of topics, including machine architecture, assembly language programming, compiler code generation, operating systems, and VLSI circuit design and implementation.

0.1 Outline

- Chapter 1 gives a tutorial for Ant-32 assembly language programming. After reading this chapter, the reader should be able to write simple Ant-32 assembly language programs.
- Chapter 2 gives a tutorial for implementing function calls and related techniques in Ant-32 assembly language.
- Chapter 3 gives a summary of the Ant-32 instruction set.
- Appendix A describes the default machine configuration, including the basic boot sequence and utility functions provided in the ROM.
- Appendix B documents the assembler directives provided by the `aa32` assembler.

Chapter 1

Ant-32 Assembly Language Programming

This chapter is a tutorial for the basics of Ant-32 assembly language programming and the Ant-32 environment. This section covers the basics of the Ant-32 assembly language, including arithmetic operations, simple I/O, conditionals, loops, and accessing memory.

1.1 What is Assembly Language?

Computer instructions are represented, in a computer, 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 computer.

A slightly higher-level representation (and one that is much easier for humans to use) is called *assembly language*. Assembly language is 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 assembly language corresponds to one instruction in the machine language.

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

1.2 Getting Started with Ant-32 Assembly: `add.asm`

To get our feet wet, we'll write an assembly language program named `add.asm` that computes the sum of 1 and 2. Although this task is very simple, in order to accomplish it we will need to explore several key concepts in Ant-32 assembly language programming.

1.2.1 Registers

Like many modern CPU architectures, the Ant-32 CPU can only operate directly on data that is stored in special locations called *registers*. The Ant-32 hardware architecture has 64 general-purpose registers.

However, some of these registers are reserved for use by the assembler, and some are reserved for other special purposes.

In the Ant-32 software architecture, there are 56 general-purpose registers available. These are named `g0` through `g55`. Each of these registers can hold a single 32-bit value.

One of the registers that is defined to have a special meaning is the *zero register* (`ze`), which always contains the constant zero. Any values can be assigned to `ze`, but the assignment has no effect.

While most modern computers have many megabytes of memory, it is unusual for a computer to have more than a few dozen registers. Since most computer programs use much more data than can fit into these registers, it is usually necessary to juggle the data back and forth between memory and the registers, where it can be operated upon by the CPU. (The first few programs that we write will only use registers, but in section 1.4 the use of memory is introduced.)

1.2.2 Commenting

Before we start to write the executable statements of our program, it is important to write a comment that describes what the program is supposed to do, and what algorithm will be used to accomplish this task. In the Ant-32 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
# add.asm-- A program that computes the sum of 1 and 2,
#       leaving the result in register g0.
# Registers used:
# g0 - 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 perhaps who to blame if it doesn't work.

Unlike programs written in higher level languages, it is usually appropriate to comment every line of an assembly language program, often with seemingly redundant comments. Uncommented code that seems obvious when you write it will be baffling a few hours later. While a well-written but uncommented program in a high level language might be relatively easy to read and understand, even the most well-written assembly code is unreadable without appropriate comments. Some programmers prefer to add comments that paraphrase 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.

1.2.3 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-32 architecture has relatively few instructions, it won't be long before you have memorized

the names of all of the frequently-occurring instructions, but when you are getting started you'll need to spend some time browsing through the list of instructions, looking for ones that you can use to do what you want.) A summary of the user-level instructions is given in Chapter 3, on page 25.

Scanning through the list of instructions, we find the the `add` instruction, which adds two numbers together. The `add` instruction takes three operands, which must 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 `g0`.
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 `g1`, 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 `g2`, 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*) instruction, which places a constant into a register. Therefore, we arrive at the following sequence of instructions:

```
# Dan Ellard
# add.asm-- A program that computes the sum of 1 and 2,
#       leaving the result in register g0.
# Registers used:
# g0 - used to hold the result.
# g1 - used to hold the constant 1.
# g2 - used to hold the constant 2.

        lc      g1, 1           # g1 = 1
        lc      g2, 2           # g2 = 2
        add     g0, g1, g2      # g0 = g1 + g2.

# end of add.asm
```

It is important to note that the `lc` instruction is not always implemented by a single Ant-32 instruction. The `lc` instruction can handle any 32-bit constant, but the Ant-32 hardware architecture only contains instructions for dealing directly with 16-bit constants. In the case where the constant has a magnitude too large to fit into 16 bits, the assembler expands the `lc` instruction into two real instructions.

For the small constants in this program, we could use `lcl` (a native instruction) instead of `lc`, but it's easier to simply always use `lc` and let the assembler decide how to handle it.

1.2.4 Completing the Program

These three instructions perform the calculation that we want, but they do not really form a complete program. We have told the processor what we want it to do, but we have not told it to stop after it has done it!

Ant-32 programs always begin executing at the first instruction in the program. There is no rule for where the program ends, however, and if not told otherwise the Ant-32 processor will read past the end of the program, interpreting whatever it finds as instructions and trying to execute them. It might seem sensible (or obvious) that the processor should stop executing when it reaches the "end" of the program (in this case, the `add` instruction on the last line), but there are some situations where we might want the program to continue past the "end" of the program, or stop before it reaches the end. Therefore, the Ant-32 architecture contains an instruction named `halt` that *halts* the processor.

The `halt` instruction does not take any operands. (For more information about `halt`, consult Section 3.11 on page 30.)

```
# add.asm-- An Ant-32 program that computes the sum of 1 and 2,
#       leaving the result in register g0.
# g0 - used to hold the result.
# g1 - used to hold the constant 1.
# g2 - used to hold the constant 2.

        lc      g1, 1           # load 1 into g1.
        lc      g2, 2           # load 2 into g2.
        add     g0, g1, g2      # g0 = g1 + g2.

        halt                    # Halt - end execution.
```

1.2.5 The Format of Ant-32 Assembly Programs

As you read `add.asm`, you may notice several formatting conventions – every instruction is 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-32 assembly language.

The first rule of Ant-32 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-32 assembly formatting is that only one instruction can appear on a each line. (There are a few additional rules, but these will not be important until section 1.3.3.)

Unlike many programming languages, where the use of whitespace and formatting is largely a matter of style, in Ant-32 assembly language some use of whitespace is required.

1.2.6 Assembling and Running Ant-32 Assembly Language Programs

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

The principal way of running an Ant-32 program is to use the command-line tools: the assembler `aa32`, the debugger `ad32` and VM `ant32`.

Using the Command-line Tools

Before the command-line tools can run on a program, the program must be written in a file. This file must be plain text, and by convention Ant-32 assembly language files have a suffix of `.asm`. In this example, we will assume that the file `add.asm` contains a copy of the `add` program listed earlier.

Before we can run 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-32 is called `aa32`, so the appropriate command would be:

```
aa32 add.asm
```

This will create a file named `add.a32` that contains the Ant-32 machine-language representation of the program in `add.asm` (and some additional information that is used by the debugger).

Now that we have the assembled version of the program, we can test it by loading it into the Ant-32 debugger in order to execute it. The name of the Ant-32 debugger is `ad32`, so to run the debugger, use the `ad32` command followed by the name of the machine language file to load. For example, to run the program that we just wrote and assembled:

```
ad32 add.a32
```

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 `g0`, we can verify that the program is working by asking the debugger to print out the contents of the registers using the `p` (for *print*) command, to see if it contains the result we expect:

```
>> p
g0 :  00000003 00000001 00000002 00000000 00000000 00000000 00000000 00000000
g8 :  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
g16:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
g24:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
g32:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
g40:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
g48:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
ra:   00000000
sp:   00000000
fp:   00000000
```

The `p` command displays the contents of all of the registers. The first column shows what registers are displayed on that line. For example, the first line lists the values in registers `g0` through `g7`. The register values are printed in hexadecimal.

To print the value of particular registers, specify the names of those registers as part of the `p` command. For example, to print the values of only `g0`, `g1`, and `g2`:

```
>> p g0, g1, g2
g0 :  hex: 0x00000003  dec:           3  ascii: '\003'
g1 :  hex: 0x00000001  dec:           1  ascii: '\001'
g2 :  hex: 0x00000002  dec:           2  ascii: '\002'
```

Note that the format of the display is different when the `p` command includes specific registers. First the hexadecimal representation of the value in the register is printed, then the decimal representation, and finally the ASCII representation (if the value is in the ASCII range). If the ASCII value is printable, the corresponding character is displayed. Otherwise, the value is shown as a 3-digit octal number (as shown in this example).

Using the `p` command, we can examine the registers to make sure that the calculation was carried out properly. Then we can use the `q` command to exit the debugger.

`ad32` includes a number of features that will make debugging your Ant-32 assembly language programs much easier. Type `h` (for *help*) at the `>>` prompt for a full list of the `ad32` commands, or consult `ad32_notes.html` for more information.

Once your program is debugged, you can use the `ant32` program to execute your `.a32` files. `ant32` simply runs an Ant-32 program and then exits.

1.3 Branches, Jumps, and Conditional Execution: `larger.asm`

The next piece of code that we will write will compare two numbers (stored in registers `g1` and `g2`) and put the larger of the two in register `g0`.

The basic structure of this program is similar to the one used by `add.asm`, except that we're computing the maximum rather than the sum of two numbers. The difference is that the behavior of this program depends upon the values in `g1` and `g2`, which are unknown when the program is written. The program must be able to decide whether to execute instructions to copy the number from `g1` into `g0`, or copy the number from `g2` into `g0`. This is known as *conditional execution* – whether or not certain parts of program are executed depends on a condition that is not known when the program is written.

1.3.1 Comparison Instructions

Our program requires a way to compare two integers to determine whether the first is larger than the second. Fortunately, the Ant-32 instruction set contains several instructions that make comparing integers easy:

`eq`	Equal
`gts`	Greater Than (signed)
`ges`	Greater Than or Equal (signed)
`gtu`	Greater Than (unsigned)
`geu`	Greater Than or Equal (unsigned)

The result of a comparison operation is that 1 is placed in the destination register if the condition is true, 0 otherwise. For example,

```
gts     g0, g1, g2
```

will cause `g0` to get the value 1 if the value in register `g1` is greater than the value in register `g2` (when the values are interpreted as signed numbers).

1.3.2 Branching and Jumping

Ant-32 contains instructions that allow the programmer to specify that execution should *branch* (or *jump*) to a location other than the next instruction, or continue with the next instruction, based on the value stored in a register. These instructions allow conditional execution to be implemented in assembly language (although not nearly as succinctly as the methods provided in higher-level languages).

In Ant-32 assembler, there are several jump instructions. The one we will focus on for this program is `jez`, which stands for *jump if equal zero*. The format of `jez` is:

```
jez     des, cond, addr
```

where *des*, *cond*, and *addr* are the names of registers. If the value in the *cond* register is zero, then execution will jump to the address specified by the *addr* register; otherwise, execution will continue with the next instruction. In either case, the address of the currently executing instruction is stored in the *des* register. (Capturing the address of the `jez` instruction in the *des* register makes it possible to use `jez` to implement function calls, as discussed in Chapter 2.)

In addition to `jez`, Ant-32 includes several other jump constructs, such as `jnz` (*jump if not equal zero*), `jezi`, `jnzi`, and `j` (an unconditional jump).

In addition to the jump instructions, Ant-32 provides several branching instructions, such as `bez` and `bnz` (*branch if equal/not equal zero*), and `bezi`, `bnzi`, and `b`.

There is a potential for confusion between the terms “branching” and “jumping”. In their common usage as verbs to describe what happens in a program, they are nearly synonymous. In the actual hardware, however, there are two distinct kinds of instructions, which implement this notion in very different ways, and the distinction between them in very important. The jump instructions cause execution to transfer to an *absolute* address, while the branch instructions cause the execution to transfer to an address calculated *relative* to the current address. For example, consider the following instructions:

`j 12`	Continue executing at location 12 in memory.
`b 12`	Continue executing at the twelth instruction past the current instruction.

Which of these instructions is more appropriate in a particular context depends on a number of factors. It is much easier to write relocatable code using branches, but often more intuitive to write simple code using jumps. Human coders usually find jumps easier to understand, while compilers and other automatic code generators find it easier to use the branching instructions.

One particular difficulty with using the branching instructions is that some of the instructions in the assembly language expand to more than one hardware instruction, and the number of instructions in the expansion can depend on several things. For example, in order to know how many instructions an `lc` will really require, it is necessary to know how large the constant is. This makes using the branch instructions

difficult unless you entirely avoid using the synthetic instructions that can expand to more than one size – easy for a code generator to do, but awkward for a human.

1.3.3 Labels

In order to use a jump instruction, we need to know the address of the location in memory that we want to jump to. Keeping track of the numeric addresses in memory of the instructions that we want to jump to is troublesome and tedious at best – a small error can make our program misbehave in strange ways, and if we change the program at all by inserting or removing instructions, we will have have to carefully recompute all of these addresses and then change all of the instructions that use these addresses. This is much more than most humans can reasonably keep track of. Luckily, the computer is very good at keeping track of details like this, and so the Ant-32 assembler provides *labels*, a way to provide a human-readable shorthand for addresses.

A *label* is a symbolic name for an address in memory. In Ant-32 assembler, a *label definition* is an identifier followed by a colon. Ant-32 identifiers use the same conventions as Python, Java, C, C++, and many other contemporary languages:

- Ant-32 identifiers must begin with an underscore, an uppercase character (A-Z) or a lowercase character (a-z).
- Following the first character there may be zero or more underscores, or uppercase, lowercase, or numeric (0-9) characters. No other characters can appear in an identifier.
- Although there is no intrinsic limit on the length of Ant-32 identifiers, some Ant-32 tools may reject identifiers longer than 100 characters.

The definition of a label 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 label definitions 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.

Because labels represent addresses, they are effectively constants. Therefore, we can use `lc` to load the address represented by a label into a register, in the same manner as we loaded the constants 1 and 2 into registers in the `add.asm` program.

1.3.4 Jumping Using Labels

Using the comparison and jump instructions and labels we can do what we want in the `larger.asm` program. Since the jump 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` (*load constant*) command. The `larger.asm` program illustrates how this is done.

```
# Dan Ellard
# larger.asm-- An Ant-32 program that finds the larger of two numbers
#       stored in registers g1 and g2, and copies it into g0.
# g0 - the result.
# g1 - the first number.
# g2 - the second number.
# g3 - the result of comparing g1 and g2.
# g4 - the address of the label "g2_larger"
# g5 - the address of the label "endif"

        lc      g4, $g2_larger  # put the address of g2_larger into g4
        lc      g5, $endif      # put the address of endif into g5

        ges     g3, g1, g2      # g3 gets (g1 >= g2)
        jez     ze, g3, g4      # if g3 is zero, jump to g2_larger
        addi    g0, g1, 0       # Otherwise, "copy" g1 into g0
        jez     ze, ze, g5      # and then jump to endif
g2_larger:
        addi    g0, g2, 0       # "copy" g2 into g0
endif:
        halt                    # Halt
```

Note that Ant-32 does not have an instruction to *copy* or *move* the contents of one register to another. We can achieve the same result, however, by adding zero to the source register and saving the result in the destination register. (There are several other instructions we could use in a similar manner to achieve the same result, but using addition is straightforward.)

We can use the `add` instruction and use the zero register (`ze`) to supply a zero. Alternatively, we can use the `addi` instruction. The `addi` instruction (and the other arithmetic instructions that end in "`i`") are called *immediate* instructions because one of their operands is a constant.

1.3.5 Running `larger.asm` Using `ad32`

Like the previous example program, we need to assemble `larger.asm`, using `aa32`, to create the file `larger.a32`, before we can run the program. Once the program is assembled, we can run it using either `ant32` or `ad32`. Unfortunately, this program isn't very interesting – since it never loads any values into registers `g1` and `g2`, the result will always be the same. In a real program, we would take the numbers from the user at runtime – but unfortunately, reading in numbers is actually a complicated exercise by itself. Luckily, we can use the debugger to load values into registers, and this will allow us to test the logic of our program.

Figure 1.1: Using `lc` to initialize registers in `ad32`. User input is shown in bold font.

```
0x80000000:              lc      g4, $g2_larger  # put the address of g2_larger into g4
>> lc g1, 100

0x80000000:              lc      g4, $g2_larger  # put the address of g2_larger into g4
>> lc g2, 200

0x80000000:              lc      g4, $g2_larger  # put the address of g2_larger into g4
>> r
PC = 0x80000024, Status = CPU Halted
HALTED at (0x80000028)

>> p g0
g0 :  hex: 0x000000c8  dec:         200
```

The `lc` debugger command mimics the `lc` mnemonic in the assembly language. For example, the command

```
lc g1, 10
```

loads the number 10 into register `g1`.

To test our program, we can use the `lc` command to load numbers into registers `g1` and `g2`, the `r` command to run the program, and then the `p` command to see the result. An entire such debugger session is shown in Figure 1.1. The user commands are shown in a bold font. Note that `ad32` prints the address of the next instruction to be executed and the source code for that instruction (unless the processor is halted), before each prompt.

1.4 Strings and `cout`: `hello.asm`

The next program that we will write is the "Hello World" program, a program that simply prints the message "Hello World" to the screen and then halts.

Ant-32 includes a very simple text-based console, with instructions to read and write single characters. The instruction for writing a single character is named `cout` (for *console output*).

Because there is no way in Ant-32 to print out more than one character at a time, we must use a loop to print out each character of the string, starting at the beginning and continuing until we reach the end of the string.

The string "`Hello World`" is not part of the instructions of the program, but it is part of the memory used by the program. The assembler places all data values (not instructions) after all of the instructions in memory.

The way that the initial contents of data memory are defined is via the `.byte` directive. `.byte` looks like an instruction that takes as many as eight 8-bit constants, but it is not an instruction at all. Instead, it is a directive to the assembler to fill in the next available locations in memory with the given values.

Data and instructions are seperated by using two assembler directives: `.data` and `.text`. The `.data` directive tells the assembler to assemble the subsequent lines into the data area, and the `.text` directive tells the assembler to assemble the subsequent lines into the *text* or instruction memory. In the assembled version of your program, all of the text is placed at the beginning, and all of the data is placed immediately after the text.

Note that the assembler assumes that the program starts with instructions, so it is not necessary for the first line of the program to be a `.text`. (Since none of the earlier examples in this document used any data memory at all, they didn't need either the `.text` or `.data` directives, but almost all the programs we will see from this point forward will use them.)

In our programs, we will use the following convention for ASCII strings: a *string* is a sequence of characters terminated by a 0 byte. For example, the string "hi" would be represented by the three characters 'h', 'i', and 0. Using a 0 byte to mark the end of the string is a convenient method, used by several contemporary languages.

The program `hello.asm` is an example of how to use labels and treat characters in memory as strings:

```
# Dan Ellard
# hello.asm-- An Ant-32 "Hello World" program.
# g0  - holds the address of the string
# g1  - holds the address of the end of the loop
# g2  - holds the address of the start of the loop
# g3  - holds the next character to be printed.

        lc      g0, $str_data   # load the address of the string into g0
        lc      g1, $endloop    # load address of the end of the loop.
        lc      g2, $loop       # load address of the start of the loop.
loop:
        ld1     g3, g0, 0       # Get the first character from the string
        jez     ze, g3, g1      # If the char is zero, we're finished.
        cout    g3              # Otherwise, print the character.
        addi    g0, g0, 1       # Increment g0 to point to the next char
        jez     ze, ze, g2      # and repeat the process...
endloop:
        halt

        # Data for the program begins here:
        .data
str_data:
        .byte 'H', 'e', 'l', 'l', 'o', ' '
        .byte 'W', 'o', 'r', 'l', 'd', '\n'
        .byte 0
```

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

1.5 Character I/O: `echo.asm`

Now that we have mastered character output, we'll turn our attention to reading and writing single characters. The program we'll write in this section simply echoes whatever you type to it, until EOI (*end of input*) is reached.

The instruction for reading a character from the console is named `cin` (for *console input*). The way that EOI is detected in Ant-32 is that when the EOI is reached, any attempt to use `cin` to read more input will immediately fail, and a negative value will be placed in the destination register to indicate that there was an error. (If the `cin` succeeds, then the destination register gets a value between 0 and 255.)

Therefore, our program will loop, continually using `cin` to read characters, and checking after each `cin` to see whether or not the EOI has been reached.

```
# Dan Ellard
# echo.asm - An Ant-32 program that echos input until EOI
#       (End of Input) is reached.
# g0 - holds each character read in.
# g1 - address of $print.
# g2 - scratch.

        lc      g1, $print
loop:
        # We've reached End of Input when g0 gets -1.  To check for
        # -1, add 1 to g0 and check to see if the result is zero.

        cin     g0              # g0 = getchar ();
        addi    g2, g0, 1       # Looking for -1...
        jnz     ze, g2, g1      # if not at EOI, go to $print.
        j       $exit           # otherwise, go to $exit.
print:
        cout    g0              # putchar (g0);
        j       $loop           # iterate, go back to $loop.
exit:
        halt                    # Exit
```

Chapter 2

Advanced Ant-32 Programming

2.1 Introduction

Any of the general registers in the Ant-32 architecture can, in general, be used in whatever way the programmer wishes. The architecture imposes no restrictions or limitations (apart from the restriction that the zero register always contains the constant 0, and that for the operations that take a register pair as an operand, the register pair must begin with an even-numbered register).

Most software architectures, however, include some conventions about the use of specific registers. These conventions are principally focussed on supporting features of high-level languages, such as functions, recursion, and separate compilation.

In order to facilitate the implementation of higher-level software architectures using Ant-32, the Ant-32 tools support two register names and conventions.

The first is a very simple model, useful for introductory programming courses and demonstrating how function calls and recursion can be implemented. This convention is the focus of the rest of this document.

The second is a more advanced model, which refines the simple model in a manner that allows for more efficient code. It is described only briefly in this document.

2.2 Simple Register Use Conventions

The simple register use conventions implement a straight-forward stack architecture. The conventions are outlined in Figure 2.1, and described in more detail below.

2.2.1 `ze` - The zero register

The `ze` register is simply register zero, which always contains the number zero.

2.2.2 `ra` - The Return Address

The `ra` register is used to store the return address of the most recent function call.

Figure 2.1: Simplified Register Use Conventions

Mnemonic	Registers	Description
ze	r0	Always zero
ra	r1	Return address
sp	r2	Stack pointer
fp	r3	Frame pointer
g0-g55	r4 - r59	General-purpose registers
u0-u3	r60 - r63	Reserved registers

Figure 2.2: Implementing `push` and `pop`

```
# push register g0:
subi    sp, sp, 4
st4     g0, sp, 0

# pop into register g1:
ld4     g1, sp, 0
addi    sp, sp, 4
```

2.2.3 `sp` - The Stack Pointer

`sp` is used as the *stack pointer*. The stack grows "downward"; a push moves the stack pointer to a numerically lower address, and a pop moves the stack pointer toward numerically greater address.

The Ant-32 architecture does not contain native push or pop instructions, and these operations require more than one instruction to execute. The push and pop operations, for example, can be coded as shown in Figure 2.2.

In general, `sp` points to the "top" of the stack (although this may seem somewhat confusing, since the stack grows downward – so the top of the stack is located at the lowest address). This convention can be relaxed in order to implement groups of push or pop operations (see Figure 2.3), as long as the stack pointer is never moved past any values that are still on the stack.

The Ant-32 assembler provides macro implemenations of `push` and `pop`, using this method.

2.2.4 `fp` - The Frame Pointer

The `fp` register is used as a *frame pointer*. The frame is often used to implement activation records, or simplify the implementation of function calls.

Figure 2.3: Combining Multiple Push or Pop Operations

For consecutive pushes and pops, it can increase code efficiency to reduce the number of `addi` and `subi` instructions by aggregating the movement of the stack pointer, as shown in the following code fragment.

```
# push registers g0, g1, g2:
subi    sp, 12
st4     g0, sp, 8
st4     g1, sp, 4
st4     g2, sp, 0

# pop into registers g3, g4, g5:
ld4     g3, sp, 0
ld4     g4, sp, 4
ld4     g5, sp, 8
addi    sp, 12
```

2.2.5 `g0-g55` - General-Purpose Registers

These registers are free to be used for any purpose.

2.2.6 `u0-u3` - Reserved Registers

These registers are reserved for use by the assembler. They are used as scratch space for the expansion of macros. They should not be used for any other purpose, and programs should never make any assumptions about their contents.

2.3 Function Calls

This section describes how the stack pointer, frame pointer, and return address registers can be used to implement the abstraction of function calls. The description is divided into four steps:

1. Preparing to call the function and performing the call.
2. Function preamble.
3. Preparing to return from the function.
4. Cleaning up after the function call.

2.3.1 Preparing to Call: Using `call`

1. All of the `g`-registers whose values need to be preserved are pushed onto the stack. The order that they are pushed onto the stack is up to the caller.

   Before the function call takes place, the caller must save any registers that contain necessary values, because otherwise the function might overwrite these values.

2. The arguments to the function are pushed onto the stack, in the reverse order that they appear (from right to left).

   The stack only contains whole words (32-bit values). If the arguments to the function are 8 or 16-bit values, then they are still pushed as the lower 8 or 16 bits of a 32-bit value, requiring four bytes of storage. It is the responsibility of the called function to ignore the extra bits.

3. Jump or branch to the function (using `jez`, `jnz`, `bez`, or `bnz`), specifying the return address register `ra` as the destination register.

Note that the last step can be accomplished with the `call` macro.

2.3.2 Handling the Call: Using `entry`

1. The current value of the `fp` and `ra` registers are pushed onto the stack.

2. The frame pointer gets a copy of the stack pointer.

3. The stack pointer is decremented by the size of the local frame. The area of memory thus allocated between the stack pointer and the frame pointer is used for local storage – for example, the local variables of the current function.

   Note that the local frame size must always be a multiple of 4, so that the stack pointer is always aligned properly on a 4-byte boundary.

These steps can be accomplished by using the `entry` macro. This macro takes a single constant argument, which is the size of the stack frame to create.

After this preamble is finished, the stack contains the information about the function call in the order shown in Figure 2.4.

Note that the function can always access its arguments and local variables via fixed offsets relative to the frame pointer, and the stack pointer is free to move. For example, the first argument (arg_0) is accessible at the address `fp` + 8, while the second argument is at address `fp` + 12, and so forth.

During a function call, the stack pointer can be used to manage the allocation of dynamic but function-private storage. If the storage requirements of the function can be computed in advance, however, it can be just as convenient to allocate this space from the frame.

Figure 2.4: Stack at start of call.

Address	Contents	Description
⋮ ⋮	`g0` $\cdots$ `g55` ⋮	Saved copies of `g`-registers.
`fp` $+ 8 + (N \times 4)$ ⋮ `fp` $+ 8$	arg_N ⋮ arg_0	Arguments to the function.
`fp` $+ 4$	`fp`	The saved value of the `fp`.
`fp` $+ 0$	`ra`	The saved value of the `ra`.
`fp` $- 4$ ⋮ `fp` $-(4 + (M \times 4))$	⋮	*local variables*

2.3.3 Returning from a Call: Using `return`

1. The return value (if any) is put into register `g0`.

   Functions that return multiple values, or a single value that is too large to fit into a single register, use a more complicated method for returning their values. This method is not documented here.

2. The stack pointer is reset to contain a copy of the frame pointer.

3. The return address is popped into `ra`, and then the `ra` register is incremented by 4.

   This increment is necessary because when the function is called via a jump or branch instruction, `ra` gets the address of the instruction that performed the call. The address we want to return to is the address of the instruction after the call.

4. The frame pointer is popped into `fp`.

   At this point, the stack pointer is in the same position as it was before the function was called.

5. Use the `jez` instruction to jump to the `ra`.

For a function that returns a single value, the `return` macro is provided to perform all of these steps. The single operand to the `return` macro can be the name of the register that contains the value to return, or the constant to return.

2.3.4 Handling the Return

When the execution resumes in the caller, the stack is exactly the same as it was before the jump to the caller. All that remains is to save the results, and restore the rest of the environment to the way it was before

the call took place. This can be done by popping the parameters and then by popping the saved `g`-registers. Once the stack is restored, execution can resume as normal.

2.4 Examples of Functions

Program `add-func.asm` gives a very simple example of a function that takes two arguments and returns their sum.

```
# Dan Ellard
# add-func.asm - an example of an Ant-32 function call.
#
# A program to compute the sum of 100 and 200, using a very simple
# function.

        # compute addFunction(200, 100).  Note that because of the way
        # the stack is organized, arguments are pushed in the opposite
        # order that they appear.
        push    100
        push    200
        call    $addFunction

        # At this point, g0 contains the sum.  There's nothing else we
        # need to do except restore the stack pointer by popping the
        # parameters back off the stack.  Since we don't actually
        # care about the values of the parameters any more, we can
        # save time by simply incrementing the stack pointer:
        addi    sp, sp, 8

        halt

        # addFunction is a function that computes the sum of two
        # numbers and returns it.
addFunction:
        entry   0               #  No extra space needed.

        # Get the arguments from the stack and put them into
        # registers.  The first argument (which is 200 in this
        # example) is loaded into g0, and the second (which is 100 in
        # this example) is loaded into g1.
        ld4     g0, fp, 8
        ld4     g1, fp, 12

        # Compute the sum in g0, and return it.
        add     g0, g0, g1
        return  g0

# end of add-func.asm
```

Program `fibonacci.asm` gives an example of a recursive function.

```
# Dan Ellard
# fibonacci.asm - An Ant-32 program to recursively compute fibonacci numbers.
#
# To compute fibonacci(5), push the 5 on the stack and then use "call"
# to invoke the fibonacci function.  In this program, since we don't
# use g1 after calling the fibonacci function, we don't need to save
# and restore it.

main:
        lc      g1, 5
        push    g1
        call    $fibonacci      # Call Fibonacci(5)
        addi    sp, sp, 4       # Restore stack pointer
        halt                    # and halt.

# The fibonacci function:  computes the X'th Fibonacci number as the
# sum of the (X-1)'th and (X-2)'th Fibonacci numbers.  The base case
# is that if 0'th and 1'st Fibonacci numbers are 1.
#
# Takes a single argument X, accessible at fp + 8.  Assumes that X is
# positive or zero.  If negative, this function will fail!  Try it if
# you want to see what stack overflow looks like...

fibonacci:
        entry   0
        ld4     g1, fp, 8       # g1 gets a copy of the current X

        # If g1 is 0 or 1, then we've reached a base case.
        jezi    g1, $fibonacci_basecase
        subi    g1, g1, 1       # decrement g1 (computing X-1)
        jezi    g1, $fibonacci_basecase
fibonacci_recurse:
        push    g1              # push argument (X-1)
        call    $fibonacci      # recursively call fibonacci
        pop     g1              # pop argument (X-1)
        mov     g2, g0          # save value of fibonacci(X-1) in g2
        subi    g1, g1, 1       # g1 = X-2
        push    g2              # preserve g2
        push    g1              # push argument (X-2)
        call    $fibonacci
        pop     g1              # pop (X-2)
        pop     g2              # restore g2
        add     g0, g0, g2      # compute fibonacci(X-2) + fibonacci(X-1)
        return  g0              # return the sum...
fibonacci_basecase:
        return  1
```

2.5 Advanced Register Use Conventions

2.5.1 Optimizing Saving and Restoring of Registers

The function calling conventions described in the first part of this chapter can result in very inefficient code. For example, imagine that we have a function α that calls function β. Before α calls β, it has to save all the registers it is using. If α uses many registers, and β only uses a few, then it may be that many of α's registers didn't need to be saved, because their values weren't modified by β at all.

One solution to this particular problem is to change the responsibility for saving the registers to the called function – in this case, β would be responsible for saving and restoring the few registers that it uses. Unfortunately, in the opposite case, where α only uses a few registers and β uses many, then this approach results in the same kind of inefficiency as we saw initially.

Ideally, each function would have its own set of registers available for its exclusive use. Unfortunately, this is impossible: typical programs have thousands of functions but processors only have dozens of registers – and even if a huge number of registers were available, recursive functions would still be a problem.

However, there is a relatively straightforward way to solve most of this problem, by dividing the register set into two groups – one group which is caller-saved (like all the registers in the earlier convention) and a second which is callee-saved. Ideally, functions that call other functions will use the callee-saved registers, and *leaf functions* (functions that do not call other functions) or the base case code of recursive functions will use the caller-saved registers. If, in our previous example, β is a leaf function, then if α uses only callee-saved registers, and β uses only caller-saved registers, then no registers will need to be saved at all.

2.5.2 Optimizing Parameter Passing

Another cause of inefficiency in the normal function call conventions is the pushing of the parameters onto the stack, and then accessing them via the frame pointer. In terms of the number of instructions executed, this convention is not terribly inefficient – but in terms of the *kind* of instructions executed, it can be very slow. Passing the parameters on the stack means storing to memory and then loading from memory, and on most modern processors accessing memory is at least an order of magnitude slower than accessing values in registers.

Therefore, to optimize the passing of parameters, we reserve a small number of registers to use for passing parameters. If there are more parameters than will fit in these registers, the remainder are passed on the stack as before. Studies of existing bodies of software have shown, however, that six (or even four) argument registers are sufficient for an overwhelming majority of common functions.

2.5.3 The Advanced Conventions

The conventions are similar to the previous, except that the g-registers have been partitioned into four different kinds of registers: return value registers, argument value registers, saved registers, and temporary registers. These registers are described in more detail below.

Figure 2.5: Advanced Register Use Conventions

Mnemonic	Registers	Description
ze	r0	Always zero
ra	r1	Return address
sp	r2	Stack pointer
fp	r3	Frame pointer
v0-1	r4 - r5	Returned values
a0-5	r6 - r11	Argument registers
s0-23	r12 - r35	Callee-saved
t0-23	r36 - r59	Caller-saved
u0-3	r60 - r63	Reserved for the assembler

Return Value Registers: `v0` - `v1`

Values returned from a function. If the return value of the function requires more than two registers to express, the remainder of the return value is returned via the stack.

Argument Value Registers: `a0` - `a5`

Parameters to a function. If the function has more than six parameters, then the additional parameters are pushed onto the stack, in the opposite order that they appear (right to left).

Callee Saved Registers: `s0` - `s23`

If any of these registers are used by a function, then the function is responsible for saving their original values and then restoring them when the function returns.

How the values are preserved and restored is up to the implementation. For implementations of languages that permit recursive or reentrant functions, using the stack is an appropriate method.

Temporary (Caller Saved) Registers: `t0` - `t23`

If any of these registers contains live values when a function is called, they are preserved by the caller and then restored after the function has returned.

How the values are preserved and restored is up to the implementation. For implementations of languages that permit recursive or reentrant functions, using the stack is an appropriate method.

The distinction between the saved registers and the temporary registers allows some useful optimizations, especially with leaf functions (functions that do not call any other functions) or the base case of recursive functions. If these functions use can manage to exclusively use `t`-registers, and their callers use only `s`-registers, then these calls do not require saving and restoring any registers: it is the responsibility of the caller to save any `t`-registers it needs, and the callee to save any `s`-registers it needs, so if the caller only

uses `s`-registers and the callee only uses `t`-registers, a significant reduction in the overhead of function calls is obtained.

Chapter 3

Ant-32 Instruction Set Summary

3.1 Notation

The notations used to describe the instructions are summarized below.

`R(x)`	The value stored in register *x*.
const8	Any 8-bit constant.
const16	Any 16-bit constant.
const32	Any 32-bit constant. A label can be used as a 32-bit constant.
•	An instruction description that begins with a • symbol indicates that the instruction is *synthetic* (see Section 3.2).

3.2 Differences Between Assembly Language and Machine Language

The Ant-32 assembly language is closely related to the Ant-32 machine language, and there is always a simple mapping from instructions in the assembly language to the corresponding machine instructions. All of the machine language instructions are directly expressible in assembly language, but the assembly language also provides a slightly higher-level abstraction of the machine (called *synthetic instructions*) in order to reduce the tedium of programming in Ant-32 assembly language, and provides several directives to the assembler

In the tables of instructions that follow in this chapter, instructions that begin with a • are *synthetic instructions.* Synthetic instructions fall into two categories: mnemonic names for operations directly supported by the hardware, and names for sequences of instructions that implement operations not directly supported by the hardware.

For an example of the first type, consider the `mov` instruction. The Ant-32 hardware does not implement such an instruction, but the same functionality can be achieved by using the `add` instruction with the zero register as one of the operands.[1]

```
# copy the contents of g1 into g0
```

[1]The `mov` instruction can also be implemented in many other ways.

```
mov     g0, g1

# This is the same as writing:
add     g0, g1, ze
```

As an example of the second type, consider the `lc` (load constant) instruction. The Ant-32 hardware does not implement any method to load a 32-bit constant into a register, but the same effect can be achieved by using an `lcl` (which loads a 16-bit constant into the lower 16 bits of a register, performing sign extension), followed by an `lch` (which loads a 16-bit constant into the upper 16 bits of a register).

```
# load constant 0x12345678 into g0
lc      g0, 0x12345678

# this is the same as writing:
lcl     g0, 0x5678
lch     g0, 0x1234
```

In many cases, the synthetic instructions have the same form as native instructions. For example, `addi` (add immediate) exists in the native instruction set, but only for 8-bit constants. The assembler will allow `addi` to take a 32-bit constant, however, by using a synthetic sequence of instructions to implement the desired functionality. Note that the assembler chooses the best way to synthesize the instruction– for example, different sequences will be created to implement `addi` depending on whether the constant requires 8, 16, or 32-bits to express.

3.3 Loading Constants

	Mnemonic	Operands	Description
	`lch`	*rdes*, *const16*	Load *const16* into the top (high-order) 16 bits of *rdes*.
	`lcl`	*rdes*, *const16*	Load *const16* into the lower 16 bits of *rdes*, and perform sign extension to fill in the top 16 bits of *rdes*.
•	`lc`	*rdes*, *const32*	Load the *const32* into *rdes*.

3.4 Arithmetic Operations

	Mnemonic	Operands	Description
	`add`	*rdes*, *src1*, *src2*	*rdes* gets `R(src1)` + `R(src2)`.
	`addi`	*rdes*, *src1*, *const8*	*rdes* gets `R(src1)` + *const8*.
•	`addi`	*rdes*, *src1*, *const32*	*rdes* gets `R(src1)` + *const32*.
	`sub`	*rdes*, *src1*, *src2*	*rdes* gets `R(src1)` - `R(src2)`.
	`subi`	*rdes*, *src1*, *const8*	*rdes* gets `R(src1)` - *const8*.
•	`subi`	*rdes*, *src1*, *const32*	*rdes* gets `R(src1)` - *const32*.
	`mul`	*rdes*, *src1*, *src2*	*rdes* gets `R(src1)` $\times$ `R(src2)`.
	`muli`	*rdes*, *src1*, *const8*	*rdes* gets `R(src1)` $\times$ *const8*.
•	`muli`	*rdes*, *src1*, *const32*	*rdes* gets `R(src1)` $\times$ *const32*.
	`div`	*rdes*, *src1*, *src2*	*rdes* gets `R(src1)` / `R(src2)`.
	`divi`	*rdes*, *src1*, *const8*	*rdes* gets `R(src1)` / *const8*.
•	`divi`	*rdes*, *src1*, *const32*	*rdes* gets `R(src1)` / *const32*.
	`mod`	*rdes*, *src1*, *src2*	*rdes* gets `R(src1)` **modulo** `R(src2)`.
	`modi`	*rdes*, *src1*, *const8*	*rdes* gets `R(src1)` **modulo** *const8*.
•	`modi`	*rdes*, *src1*, *const32*	*rdes* gets `R(src1)` **modulo** *const32*.

The "o" arithmetic operations are similar to the ordinary arithmetic operations, except that they include the calculation of the "overflow", if any, from the operations. For these operations, *rdes* must be an even-numbered register. The result of the operation is stored in registers *rdes* and *rdes* $+1$. Consult the architecture reference for more information.

	Mnemonic	Operands	Description
	`addo`	*rdes*, *src1*, *src2*	Add with overflow.
	`addio`	*rdes*, *src1*, *const8*	Add immediate with overflow.
•	`addio`	*rdes*, *src1*, *const32*	
	`subo`	*rdes*, *src1*, *src2*	Subtract with overflow.
	`subio`	*rdes*, *src1*, *const8*	Subtract immediate with overflow.
•	`subio`	*rdes*, *src1*, *const32*	
	`mulo`	*rdes*, *src1*, *src2*	Multiply with overflow.
	`mulio`	*rdes*, *src1*, *const8*	Multiply immediate with overflow.
•	`mulio`	*rdes*, *src1*, *const32*	

3.5 Logical Bit Operations

	Mnemonic	Operands	Description
	`and`	*rdes*, *src1*, *src2*	*rdes* gets the bitwise AND of `R(src1)` and `R(src2)`.
•	`andi`	*rdes*, *src1*, *const32*	*rdes* gets the bitwise AND of `R(src1)` and *const32*.
	`nor`	*rdes*, *src1*, *src2*	*rdes* gets the bitwise NOR of `R(src1)` and `R(src2)`.
•	`nori`	*rdes*, *src1*, *const32*	*rdes* gets the bitwise NOR of `R(src1)` and *const32*.
	`or`	*rdes*, *src1*, *src2*	*rdes* gets the bitwise OR of `R(src1)` and `R(src2)`.
•	`ori`	*rdes*, *src1*, *const32*	*rdes* gets the bitwise OR of `R(src1)` and *const32*.
	`xor`	*rdes*, *src1*, *src2*	*rdes* gets the bitwise XOR of `R(src1)` and `R(src2)`.
•	`xori`	*rdes*, *src1*, *const32*	*rdes* gets the bitwise XOR of `R(src1)` and *const32*.

3.6 Bit Shifting Operations

	Mnemonic	Operands	Description
	`shl`	*rdes*, *src1*, *src2*	Shift `R(src1)` left by `R(src2)` bits.
	`shli`	*rdes*, *src1*, *const8*	Shift `R(src1)` left by *const8* bits.
•	`shli`	*rdes*, *src1*, *const32*	Shift `R(src1)` left by *const32* bits.
	`shr`	*rdes*, *src1*, *src2*	Shift `R(src1)` right by `R(src2)` bits.
	`shru`	*rdes*, *src1*, *src2*	Unsigned shift `R(src1)` right by `R(src2)` bits.
	`shri`	*rdes*, *src1*, *const8*	Shift `R(src1)` right by *const8* bits.
•	`shri`	*rdes*, *src1*, *const32*	Shift `R(src1)` right by *const32* bits.
	`shrui`	*rdes*, *src1*, *const8*	Unsigned shift `R(src1)` right by *const8* bits.
•	`shrui`	*rdes*, *src1*, *const32*	Unsigned shift `R(src1)` right by *const32* bits.

The left shift operation shifts the bits "left", towards the more significant bits, filling in the least significant bits with zeros. The right shift operations shift the bits toward the least significant bits. If the operation is "unsigned" then zeros are used to fill in the most significant bits, but if the operation is not "unsigned" then a copy of the most significant bit in the *src1* register is used to fill these bits.

3.7 Load/Store Operations

Mnemonic	Operands	Description
`st1`	*src1*, *src2*, *const8*	Store the least significant byte of `R(src1)` to the address `R(src2)` + *const8*.
`st4`	*src1*, *src2*, *const8*	Store `R(src1)` to the address `R(src2)` + *const8*.
`ld1`	*rdes*, *src1*, *const8*	Load the byte at address `R(src1)` + *const8* into *rdes*. The byte is sign-extended to 32-bits.
`ld4`	*rdes*, *src1*, *const8*	Load the word at address `R(src1)` + *const8* into *rdes*.
`ex4`	*rdes*, *src1*, *const8*	Exchange the contents of register *rdes* and the word at address `R(src1)` + *const8*

3.8 Comparison Instructions

	Mnemonic	Operands	Description
	`eq`	*rdes*, *src1*, *src2*	*rdes* gets 1 if `R(src1)` $==$ `R(src2)`, 0 otherwise.
	`ges`	*rdes*, *src1*, *src2*	*rdes* gets 1 if `R(src1)` $\geq$ `R(src2)`, 0 otherwise. The comparison uses signed numbers.
	`gts`	*rdes*, *src1*, *src2*	*rdes* gets 1 if `R(src1)` $>$ `R(src2)`, 0 otherwise. The comparison uses signed numbers.
	`geu`	*rdes*, *src1*, *src2*	Like `ges`, but using unsigned numbers.
	`gtu`	*rdes*, *src1*, *src2*	Like `gts`, but using unsigned numbers.
•	`les`	*rdes*, *src1*, *src2*	*rdes* gets 1 if `R(src1)` $\leq$ `R(src2)`, 0 otherwise. The comparison uses signed numbers.
•	`lts`	*rdes*, *src1*, *src2*	*rdes* gets 1 if `R(src1)` $<$ `R(src2)`, 0 otherwise. The comparison uses signed numbers.
•	`leu`	*rdes*, *src1*, *src2*	Like `les`, but using unsigned numbers.
•	`ltu`	*rdes*, *src1*, *src2*	Like `lts`, but using unsigned numbers.

3.9 Branch and Jump Instructions

	Mnemonic	Operands	Description
	`jez`	*rdes*, *src1*, *src2*	If `R(src1)` is zero, jump to the address `R(src2)`. *rdes* gets the address of the current instruction.
	`jnz`	*rdes*, *src1*, *src2*	If `R(src1)` is not zero, jump to the address `R(src2)`. *rdes* gets the address of the current instruction.
•	`jezi`	*rdes*, *src1*, *const32*	If `R(src1)` is zero, jump to *const32*. *rdes* gets the value of the current instruction.
•	`jnzi`	*rdes*, *src1*, *const32*	If `R(src1)` is not zero, jump to *const32*. *rdes* gets the value of the current instruction.
	`bez`	*rdes*, *src1*, *src2*	If `R(src1)` is zero, branch to the address of the current instruction plus `R(src2)`. *rdes* gets the address of the current instruction.
	`bnz`	*rdes*, *src1*, *src2*	If `R(src1)` is not zero, branch to the address of the current instruction plus `R(src2)`. *rdes* gets the address of the current instruction.
	`bezi`	*src1*, *const16*	If `R(src1)` is zero, branch to the address of the current instruction $+$ $(4 \times$ *const16*$)$.
	`bnzi`	*src1*, *const16*	If `R(src1)` is not zero, branch to the address of the current instruction $+$ $(4 \times$ *const16*$)$.

3.10 Console I/O Instructions

	Mnemonic	Operands	Description
	`cin`	*rdes*	Read a character from the console into *rdes*.
	`cout`	*src1*	Write the character `R(src1)` to the console.

3.11 Halting

	Mnemonic	Operands	Description
	`halt`		Stop the processor.

3.12 Artificial Instructions

	Mnemonic	Operands	Description
•	`mov`	*rdes, src1*	Copy `R(src1)` to *rdes*.
•	`j`	*const32*	Jump to *const32*.
•	`j`	*src1*	Jump to `R(src1)`.
•	`b`	*const32*	Branch to *const32*.
•	`b`	*src1*	Branch to `R(src1)`.
•	`push`	*src1*	Push `R(src1)` onto the stack.
•	`pop`	*rdes*	Pop the stack into *rdes*.
•	`call`	*const32*	Call a function (see Section 2.3.1).
•	`entry`	*const32*	Create a stack frame (see Section 2.3.2).
•	`return`	*src1*	Return `R(src1)` from a function (see Section 2.3.3).
•	`return`	*const32*	Return the *const32* from a function (see Section 2.3.3).

Appendix A

The Default Machine and ROM Routines

A.1 Introduction

This section describes the default Ant-32 implementation and the default ROM. The default ROM supplied with the implementation of the Ant-32 architecture contains a boot routine for initializing the machine and several utility functions to simplify writing small programs.

A.2 Hardware Overview

The default machine has 4 megs of physical RAM, contiguous from physical address 0 to physical address `0x3fffff`. Because of the manner in which physical memory is addressed in (unmapped) system mode, this means that this RAM appears to begin at virtual address `0x80000000` and ends at `0x803fffff`. Other RAM sizes are possible, however, so it is a mistake to assume that this is always the amount of RAM available.

In addition to the RAM, there are 4 pages (16K) of ROM located at the top of the physical address space. This small area of memory is where the default ROM is located.

The details of the ROM are best described in the source code for the ROM itself, and readers interested in more detail should refer to it. The source for the default ROM is provided as part of the normal Ant-32 distribution.

A.3 Initialization

When the machine is booted, if the memory image was constructed in the usual fashion (as described in `aa32_notes.html`), a short initialization routine located in the ROM is called before execution continues with the main program.

The boot ROM assumes that the memory image for the main program has already been loaded into memory, starting at memory location `0x80000000`.

Note that there is nothing sacred here– all of the initializations done here can be overridden by the main program. The purpose of the routine supplied in the ROM is simply to supply reasonable defaults so that it

is, for many purposes, unnecessary to override anything. The only really important thing that the main code needs to take into account is that exceptions are enabled by the ROM.

The steps taken by the initialization routine are as follows:

1. *Determine the size of physical memory.*

   This is done by iteratively probing each page of RAM, starting at physical location 0 and continuing until either the address space of physical RAM is exhausted (at 1 Gbyte) or an invalid page is encountered.

2. *Initialize the* `sp` *and* `fp` *registers.*

   The frame pointer and stack pointer are initialized to point to the "top" of physical memory (via addresses in the unmapped segment). Note that because of the way that the stack operations are implemented, the initial location pointed to by the frame pointer and stack pointer is actually one word *past* the end of physical memory.

   The initialization code assumes that it knows how much RAM is actually present. It is possible to write this routine in such a way that it first detects how much memory there is in the machine, but this has not been implemented yet.

3. *Prepare for Exceptions, and Zero the Cycle Counters*

   First, the exception handler is set to the address of a routine located in the ROM (named `antSysRomEH`) that prints an error message and halts if a run-time exception occurs. This is a minimal exception handler (since it doesn't really "handle" exceptions, it just makes the results a little less messy).

   Next, the *exception disable* flag is cleared, permitting exceptions to occur.

   Finally, the cycle counters and registers used by the probing routines are set to zero,

4. *Call the Main Code*

   The call to the main code of the program is implemented in the same manner as a zero-argument function call, so that if the "main" of the program returns, this code will be able to properly halt the machine.

A.4 ROM Routines

The functions in the ROM use the calling conventions described in Chapter 2. For routines that require more than one parameter, the parameters are listed in the order that they should be pushed onto the stack.

A.4.1 Memory Management

`antSysSbrkInit` Set the initial address of the boundary (aka *break*) between preallocated memory and memory available for dynamic memory allocation.

Note that all memory *after* this boundary (at higher addresses) is implicitly assumed to be available for dynamic allocation, which is not a completely accurate assumption, because the stack is also located

in this region, and grows down towards the break. If the break and the frontier of the stack cross, disaster is very likely. Detecting this situation without adding costly overhead to every `push` requires advanced techniques not described here.

`antSysSbrk` Takes a single argument *size*, which is the amount to move the break. The previous value of the break is returned.

If the *size* is positive, the break is advanced, effectively allocating memory. If the *size* is negative, memory is deallocated.

Note that the *size* is always rounded up (towards positive infinity) to the nearest multiple of 4, in order to ensure that the break is always properly aligned for any memory access operation. This can cause confusing behavior when trying to deallocate a small amount of memory. For example, using `antSysSbrk` with a size of 1 advances the break by 4 bytes (allocating 3 extra bytes), but using `antSysSbrk` with a value of -1 does not move the break at all, so no memory is actually deallocated.

A.4.2 Simple I/O Routines

`antSysPrintString` Print the zero-terminated ASCII string pointed to by the argument.

`antSysPrintSDecimal` Print the argument as a 32-bit signed decimal integer.

`antSysPrintUDecimal` Print the argument as a 32-bit unsigned decimal integer.

`antSysPrintHex` Print the argument as a 32-bit hexadecimal integer.

`antSysReadLine` Read characters until end-of-line or end-of-input is reached. (The behavior mimics the `fgets` function from the standard C library.)

This routine takes two parameters, which are pushed onto the stack in the following order:

buffer length The maximum number of characters to read from the console.

buffer address The address of the buffer to place the characters read from the console.

`antSysReadDecimal` Read characters from the console and interpret them as a 32-bit signed decimal number, which is returned.

Invalid input characters (such as non-digit characters, or a number too large to represent in 32 bits) will result in an arbitrary value being returned. No error checking is performed.

`antSysReadHex` Read characters from the console and interpret them as a 32-bit hexadecimal number, which is returned.

Invalid input characters (such as non-hex characters, or a number too large to represent in 32 bits) will result in an arbitrary value being returned. No error checking is performed.

Appendix B

Ant-32 Assembler Reference

B.1 Comments and Whitespace

A comment begins with a # and continues until the following end-of-line. The only exception to this is when the # character appears as part of an ASCII character constant (as described in section B.3).

Once comments have been removed, any line that is not indented defines a *label*. The name of the label begins with the first character of the line, and continues until a colon (`:`) has been reached. All other lines must be indented. The recommended level of indentation is at least one tab-stop; additional indentation may be used, at the discretion of the programmer, to clarify the program structure.

B.2 Summary of Directives

Name	Parameters	Description
`.text`		Assemble the following assembly language statements as program instructions. (This is the default.)
`.data`		Assemble the following assembly language statements as data.
`.define`	*name*, *value*	Bind the *value* to the *name*.
`.byte`	*byte1*, ⋯, *byteN*	Assemble the given byte values.
`.word`	*word1*, ⋯, *wordN*	Assemble the given word (4-byte) values.
`.ascii`	"*string*"	Assemble the given string. The string is not zero-terminated.
`.asciiz`	"*string*"	Assemble the given string, including the a zero-terminating byte.
`.align`	*size*	Force alignment to the next address used by the assembler to the given *size*, skipping over memory if needed.

B.3 Constants

Several Ant-32 assembly instructions contain 8, 16, or 32-bit constants. A 32-bit constant 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. Examples are shown in the following table:

Representation	Value	Decimal Value
Decimal (base 10)	`65`	65
Hexadecimal (base 16)	`0x41`	65
Octal (base 8)	`0101`	65
Binary (base 2)	`0b01000001`	65
ASCII	`'A'`	65
Decimal (base 10)	`10`	10
Hexadecimal (base 16)	`0xa`	10
Octal (base 8)	`012`	10
Binary (base 2)	`0b1010`	10
ASCII	`'\n'`	10

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 `.byte`, `.word`, or `.ascii` item for labels that appear in the data.

The 8 and 16-bit constants can be specified in all the same ways as the 32-bit constants *except* for labels, which are always 32 bits.

B.4 Symbolic Constants

Constants can be given symbolic names via the `.define` directive. This can result in substantially more readable code. The first operand of the `.define` directive is the symbolic name for the constant, and the second value is an integer constant. Unfortunately, the integer constant must not be a label or another symbolic constant.

```
.define ROWS, 10        # Defining ROWS to be 10
.define COLS, 10        # Defining COLS to be 10

lc      g2, ROWS        # Using ROWS as a constant
addi    g3, g3, COLS    # Using COLS as a constant
```

Note that `.define`'d constants can be redefined at any point.

B.5 The `.byte`, `.word`, and `.ascii` Directives

The `.byte` and `.word` directives are used to specify data values to be assembled into the next available locations in memory. `.byte` is used to assemble bytes, and `.word` is used to assemble 32-bit values.

Name	Parameters	Description
.byte	*byte1*, ⋯, *byteN*	Assemble the given bytes (8-bit values) 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 hexadecimal, octal, binary, decimal or character constants.
.word	*word1*, ⋯, *wordN*	Assemble the given words (32-bit values) into the next available locations in the data segment. As many as 8 words can be specified on the same line. Words may be specified as labels, hexadecimal, octal, binary, decimal or character constants.
.ascii	*"string"*	Assemble the given string (which must be enclosed in double quotes) as a sequence of 8-bit ASCII values. Note that a terminating zero is *not* added to string by .ascii, and must be placed there explicitly if desired.
.asciiz	*"string"*	Assemble the given string (which must be enclosed in double quotes) as a sequence of 8-bit ASCII values. Unlike .ascii, a terminating zero byte is added to the end of the string.

B.6 .align

The Ant-32 architecture only allows memory references that are *aligned* according to their size: 4-byte word reads and writes must always be aligned on 4-byte boundaries (their address must always be divisible by 4). Byte reads and writes do not have any alignment restrictions, since all addresses are divisible by 1.

The .align directive is used to ensure that an address is divisible by an arbitrary amount. The .align directive is used to ensure that addresses are properly aligned. The .align directive causes the assembler to skip to the next address which is a multiple of its argument *size*. (If the current address is a multiple of the *size*, then no skip is needed.)

For example, to ensure that the address of a .word is aligned in a 4-byte boundary after an .ascii string:

```
        .ascii  "hello"
        .align  4 # make sure that xxx is aligned on a word boundary
xxx :   .word   100
```

This will ensure the address xxx is aligned on a 4-byte boundary.

.align can also be used to align on other boundaries, such as page boundaries (by using a size of 4096).

Note that the alignment adjustment is done *after* the rest of the line is processed, and therefore it is usually incorrect to put a label definition on the same line as a .align, because the label will be assigned to a possibly misaligned address. For example:

```
xxx:    .align  4       # WRONG: xxx might not be aligned
        .word   100     # xxx might not be the address of this word.
```

```
        .align  4       # RIGHT: yyy will be aligned properly
yyy:    .word   100     # yyy will be the address of this word
```

Index