1 Introduction

This manual provides tutorials, experiments and documentation for the University of Toronto Ultragizmo board and its associated hardware. This chapter gives an overall description of the board.

The Ultragizmo board, shown in Figure 1, contains a Motorola 68306 integrated microprocessor containing a M68000 processor core, an Altera 10K70 field programmable gate array (FPGA) chip and supporting hardware. Figure 2 gives a block diagram of the board, showing the set of chips and their functions. The microprocessor and its digital memory functions as a stand-alone microcomputer. The FPGA provides a convenient hardware platform for quick implementation of peripheral digital circuits.

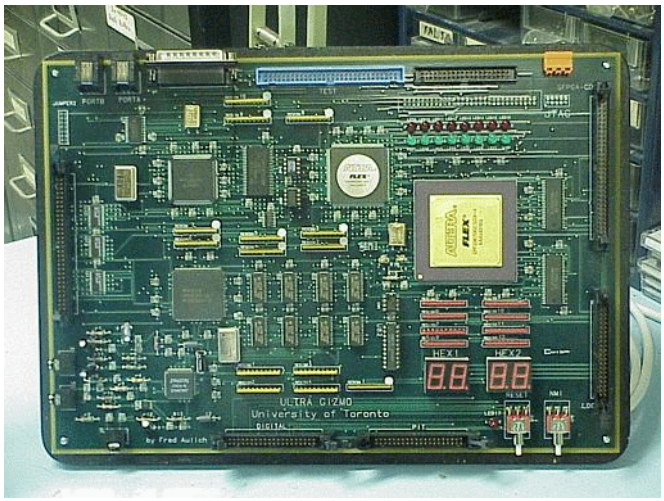


Figure 1 - *Picture of Ultragizmo Board*

The CPU of the board is the MC68306 integrated microprocessor. It has a clock frequency of 16.67 MHz. The processor contains an M68000 processor core, which is fully functionally compatible with the original M68000 microprocessor. Beside the processor core, each MC68306 chip also contains various peripherals, including an interrupt controller, and a 2-channel serial I/O system (DUART). In addition, the board contains 10 MBytes of TMS417400 Dynamic Random Access Memory (DRAM) for main memory, controlled through a National DP8422AV DRAM controller.

A majority of the glue logic of the Ultragizmo board is implemented in the CFPGA, which is an Altera 10K20 FPGA device. Through the CFPGA and on-chip peripherals of the MC68306, the M68000 processor core has access to a variety of I/O devices including two serial I/O ports, a high speed Centronics parallel port, and the PIT, which is a parallel I/O port. The M68000 bus is also directly connected to a buffered 60-pin I/O connector. The FBUG monitor, a simple operating system running on the Ultragizmo board which allows communication with the PC, is stored in 2 MB of AMD AM29F200 Flash ROM.

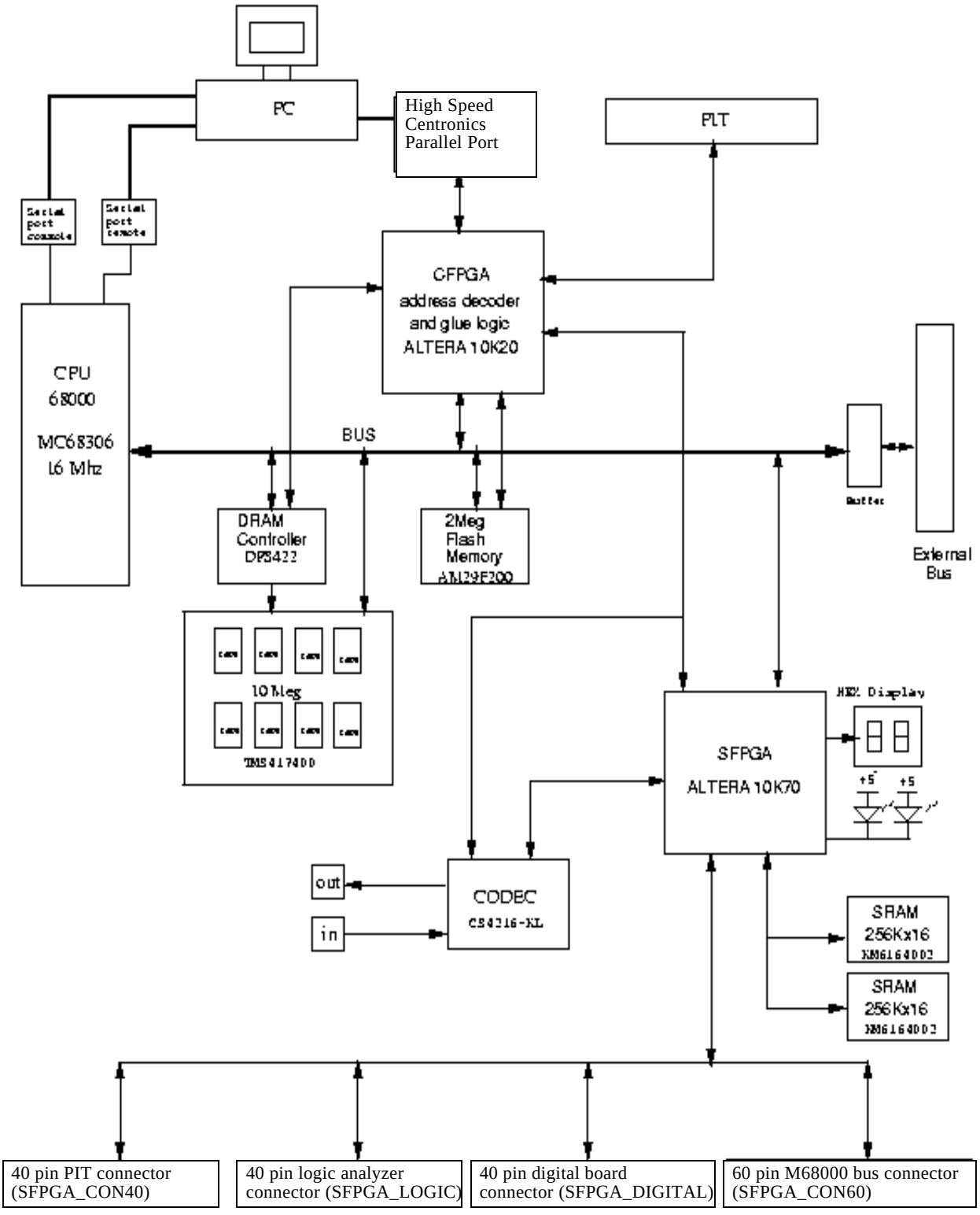


Figure 2 - *Functional Diagram of Ultragizmo Board*

Also attached to the M68000 bus is an Altera 10K70 FPGA, called the SFPGA. Complex digital systems can be implemented using the device. The 10K70 FPGA contains 3744 look-up tables and 18432 bits of user programmable RAM. The large capacity of the 10K70 can be used to implement systems like RISC processors and 2-D graphics accelerators. The 10K70 is attached to several useful digital components including a CS4216-KL stereo audio CODEC chip, 1 MByte of Static Random Access Memory (SRAM), four I/O connectors, four HEX displays and 16 LEDs.

The setup of the Ultragizmo board in the lab is shown in Figure 3. The board is connected to a PC workstation through two serial ports and a Centronics parallel port. Through one serial port, the PC provides simple keyboard input and screen output services for the board. The other serial port is used for downloading microprocessor programs from the PC. The parallel port is used for downloading FPGA chip configurations and M68000 programs from the PC. From the PC, communication to the Ultragizmo board is performed through the FBUG monitor program for M68000 programs, and through Max+PlusII for Altera 10K70 configuration.

This manual is divided into 11 chapters. Chapter 2 is a tutorial on assembly language programming. Chapter 3 is a tutorial on FPGA programming. Laboratory experiments are given in Chapter 4 and Chapter 5. Chapter 6 and Chapter 7 give detailed descriptions on assembly language programming and the Ultragizmo monitor program, respectively. Chapter 8 gives the detailed specifications of the board, including pin descriptions of all components. Chapter 9 and Chapter 10 describe various peripheral devices. URLs for web datasheets for Ultragizmo components are given in Chapter 12.

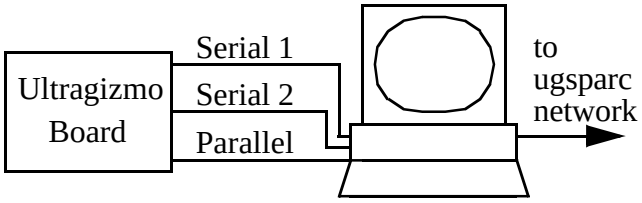


Figure 3 - *Ultragizmo Board Environment*

For More Information

For more information on the Ultragizmo board and for Ultragizmo board updates, please refer to both the */cad2/ultragizmo/Updates* directory on the *ugsparc* network, and to Fred Aulich's web page, located at *http://www.eecg.toronto.edu/~aulich.*

2 Assembly Language Programming Tutorial

This chapter provides an introduction and step-by-step tutorial on the use of the Ultragizmo board for assembly language programming. It assumes that you are capable of programming in a high level programming language, such as C, FORTRAN, or Turing. The chapter is divided into two sections. Section 2.1 provides a brief introduction to assembly-language programming. Section 2.2 shows how to assemble, debug, download, and execute assembly language programs onto the Ultragizmo board.

2.1 Assembly Language and Machine Code

Before describing how the Ultragizmo board is programmed, we provide a brief discussion on the differences between high-level language programming and the form of code the microprocessor actually executes. We assume that you are familiar with high-level language programming of computers.

Consider the following typical high-level language addition statement:

A = B + C

This statement describes addition of variables B and C as well as the assignment of the result into variable A. A microprocessor does not typically execute statements in this form. As you will learn in your class lectures, the microprocessor contains **registers**, which hold data internal to the processor. The Motorola 68000 has, among others, eight data registers named d0, d1, ... d7. Assume, for simplicity, that the variables A, B, and C are assigned to registers d0, d1, and d2, respectively.

A microprocessor can only execute very simple functions. Several of them must be put together to perform the high-level function described above. For the Motorola 68000, the instructions corresponding to the above high-level language statement are:

```
add.w     d1,d2     (1)
move.w    d2,d0     (2)
```

The assembly language statement labelled (1) adds the numbers in registers d1 and d2, and places the result in register d2 (which destroys the original value of B). The statement labelled (2) copies (“moves”) the contents of register d2 into register d0, which places the result of the addition into the register assigned to variable A. This is an example of the typical instructions that a microprocessor actually executes, one at a time. There are many other simple operations that you will learn about, including subtraction, multiplication, division, testing (related to high-level ‘if’ statements) and branching (related to high-level ‘goto’ statements). Can you think of a sequence of instructions which could be used instead and which would not destroy the original value of B?

There is one more thing to note: the microprocessor does not directly read the statements such as the “add.w” statement above. This must be translated, by a program called an *assembler*, into numerical codes which are called *machine code.* For example, the machine code for the add assembly language statement above (1) is d441 in base sixteen, and the code for the move state-

ment is 3002 in base 16. (The digits used in the hexadecimal base 16, which we will use in this manual extensively, are the usual **0** through **9**, and then **a** through **f**, where $a_{16} = 10_{10}$, $b_{16} = 11_{10}$, ..., $f_{16} = 16_{10}$).

The machine code of any program is stored in the memory of the microprocessor system (in our case it is the Dynamic Random Access Memory (DRAM), illustrated in Figure 2). The microprocessor fetches these codes and executes the appropriate instruction.

In the University of Toronto Ultragizmo system as we currently have it set up, the workstations perform the translation from assembly language into machine code (that is, the *assembler* program is executed on the workstations). The machine code is then copied onto the Ultragizmo board and executed. The mechanics of this process are described in Section 2.2.1 on page 6.

A Note About Variables

In programming with high-level languages, you are no doubt accustomed to thinking of variables as existing only in the computer's memory. When dealing at the assembly language level there are at least two physical locations for variables that you need to understand: some variables can reside inside **registers** inside the CPU, instead of in the main memory. On the Motorola 68000, these registers have separate names, such as d0, d1, a0 or a6, and are referred to directly by these names in the M68000 assembly language. Other variables reside in main memory. At the assembly language programming level you need to be aware that each memory location has a numerical address numbered, in order, starting from 0. We refer, for example, to the 'byte' at location 143.

A picture of the two kinds of memory inside the Ultragizmo board is illustrated in Figure 4. Your assembly language program must distinguish between the two.

As illustrated in Figure 4, the M68000 is a **byte-addressable** machine, which means that each 8-bit byte has its own separate address. Note that the M68000 assembly language (and processor) also allows you to access a **word** (which is two consecutive bytes beginning at an even address) or a **long word** (which is four consecutive bytes, also beginning at an even address). To tell the microprocessor what size of data you wish to access, the assembly-language instruction *operation code* is appended with a size designator. An instruction is of the form **op.s**, where **op** is the operation code (such as "move" or "add") and **s** is the size designator. For example, the assembly language statement

```
move.b    140,d0
```

means to copy the **byte** at memory address 140 into register d0. Alternatively, the statement

```
move.w    140,d0
```

means to copy the **word** beginning at memory location 140 (including the bytes at 140 and 141) into register d0. Finally the statement

```
move.l    140,d0
```

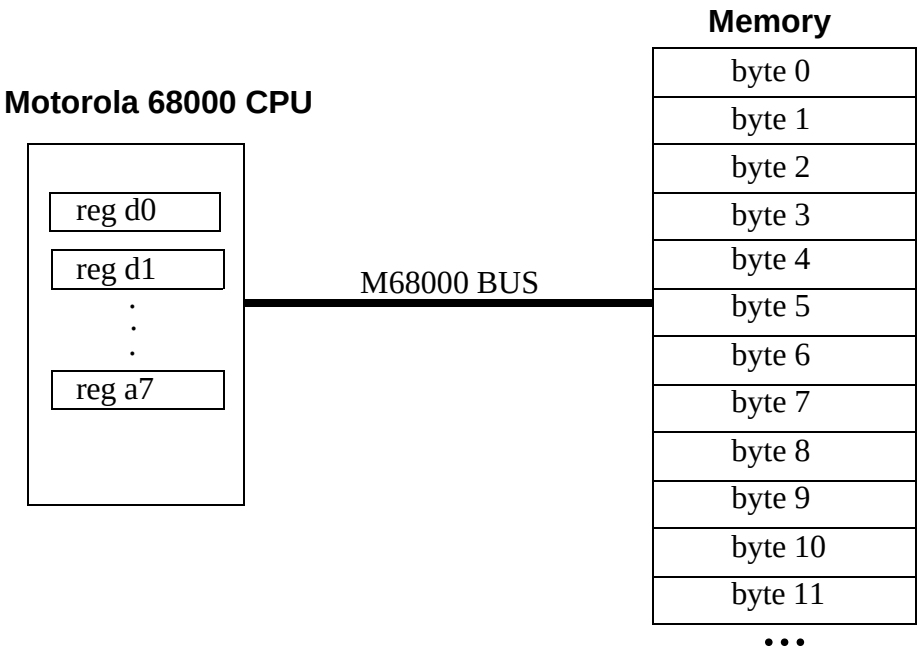


Figure 4 - *Location of Variables Inside a Microprocessor System*

means to copy the **long word** beginning at memory location 140 (including the bytes at 140, 141, 142 and 143) into register d0.

The registers d0 to d7 and a0 to a7 are 32 bits in size. If a smaller size number is copied into these registers, then only the low-order bits are affected (e.g. bits 0 to 7 in the case of a byte).

2.2 Tutorial

The purpose of this tutorial is to familiarize you with the University of Toronto Ultragizmo board and its environment. The first part of the tutorial will show you how to create, download and execute an assembly language program. The second part will illustrate several kinds of errors that can occur in assembly language programs, and how to fix them. The third part is a tutorial on how to use the facilities of the FBUG monitor to analyze and debug a running program. The fourth part illustrates some basic concepts of machine code and microprocessor system memory.

More detailed information about the Ultragizmo board and M68000 assembly language can be found in other chapters of this manual. Chapter 6 is a reference guide for writing Motorola 68000 assembly language programs. Chapter 7 is a reference guide on the debugging facilities of the Ultragizmo board. Chapter 8 provides details on the various hardware components on the Ultragizmo board while Chapter 9 and Chapter 10 describe other hardware that can be connected to the Ultragizmo board.

2.2.1 Logging In, Creating and Running a Simple Motorola 68000 Program

To do this tutorial you should be sitting in front of a PC attached to an Ultragizmo board. Note that you can save time in the lab by typing the example programs in Figure 7 and Figure 9 into a

text file in advance of your lab time, or you can copy them from */cad2/ultragizmo/MLabs/prog1.s* and */cad2/ultragizmo/MLabs/prog2.s* on the *ugsparc* system. Note that you can access your *ugsparc* account from the digital systems laboratories.

First, turn on the power to the board and the PC. This means both the power supply box connected to the Ultragizmo board and the power bar that the power supply is plugged into. After the PC has displayed its login window, enter your *ugsparc* login name and password.

Once you have logged onto the PC, select **Start->Run** to bring up the **Run** dialog window. In the **Run** dialog window, type in the command **maphome <login_name>**, where the <**login_name**> is your *ugsparc* account login name. Press **Enter**. A window as shown in Figure 5 will appear on your desktop. If you are a second year student, select 2. If you are a third year student, select 3, etc. This will properly connect your PC workstation to the *ugsparc* system server. You *ugsparc* home directory will be mapped to the *W:* drive on your PC. Besides the *C:* and the *W:* drives, the PC also contains an *X:* drive where system programs are located.

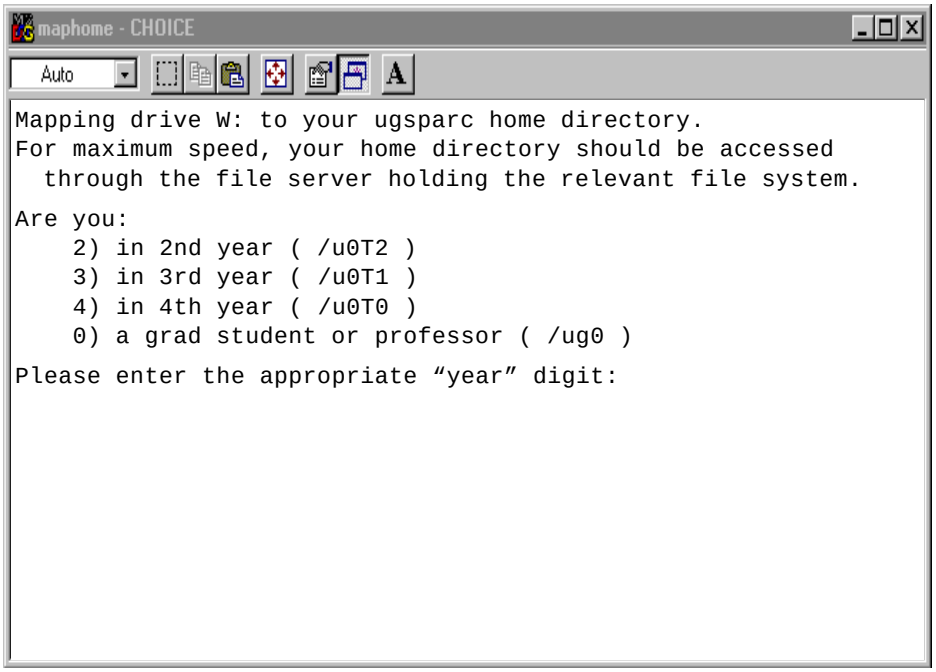


Figure 5 - *The Maphome Window*

Once the **maphome** program is finished, double click the **CONPORT** icon 

. This will open up a **Tera Term** window connected to COM1 of the PC. COM1 is connected to one of the Ultragizmo serial ports providing keyboard input and character output services for the board. After the **CONPORT** window is displayed (see Figure 6), press the reset button on the Ultragizmo board. At this point the **CONPORT** window should display an angle bracket prompt (Ultrag>).

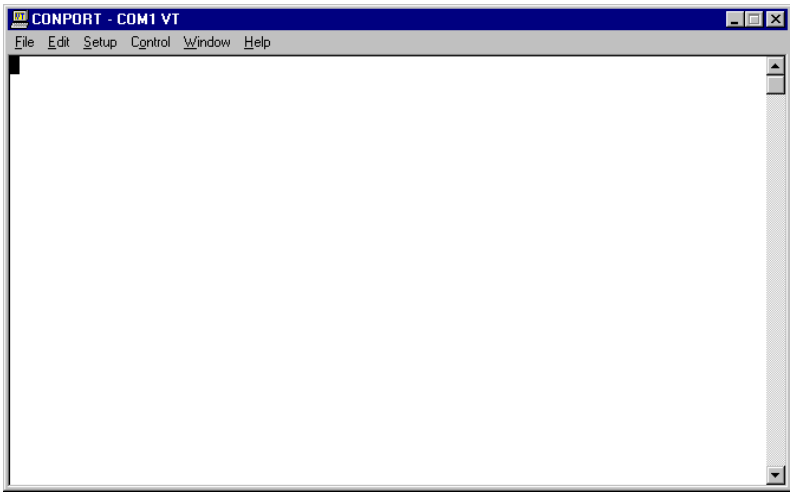


Figure 6 - *The CONPORT Window*

Next, you need to connect to one of the *ugsparc* workstations. To do this, double click the **TERA TERM SSH** icon  . Enter the login machine name as *ugsparcX*, where *X* is an available *ugsparc* machine number. Enter your *ugsparc* login name and password as requested.

You should now have logged into an *ugsparc* machine through the **TERA TERM SSH** window. Now create a directory to work in, and change directory into it. Continue to work in the **TERA TERM SSH** window. Using your favourite text editor (vi, jove, emacs, or pico), create a file, named *prog1.s*, and enter the program shown in Figure 7 into it. (If you don't know how to use a text editor, you will have to learn now! The simplest editor to learn is probably pico). To save time during the lab itself, it would be best if you created this file in advance, using the *ugsparc* facilities.

A few notes on entering this assembly language program: **Be sure** to place the labels (such as SRA, TBA, START, OUTCHR, etc.) in the **first** column of a line; the non-labelled statements should **NOT** begin in the first column. Spaces can be tabs or ordinary spaces. Make sure you copy the right number of zeroes in all of these numbers!

After creating the file *prog1.s*, you must convert the assembly language into machine code, using the **a68** program, as follows ("prompt%" is your workstation prompt):

prompt% **a68 prog1.s**

If **a68** complains about an "Unknown operator" on almost every line of your file, your path is set so that the **gnu a68** assembler is being used. This assembler uses a different assembly language syntax than the standard Motorola one. Change your path or type in the full pathname (*/local/bin/a68*) to assemble your program properly.

```
SRB       equ     $fffff7f3
TBB       equ     $fffff7f7

          org     $20000

START     move.b  #'A',d0     ; start with an A
OUTCHR    btst.b  #2,SRB      ; check if display ready
          beq     OUTCHR
          move.b  d0,TBB      ; send character to display
          addq.b  #1,d0       ; create next character
          cmpi.b  #'Z',d0     ; check if done
          ble     OUTCHR      ; if not, go back and display it
          trap    #15         ; exit
```

Figure 7 - *M68000 Assembly Language Program prog1.s*

The **a68** assembler requires instructions and register names to be written in lower-case letters. As well, it dislikes spaces between operands. For example, the addq.b #1,d0 line would not compile if it were written as ADDQ.B #1, D0.

The **a68** assembler creates a file in your directory called *prog1.srec*. It contains the machine code for your program, in the *S-record* standard format, which is a Motorola-standard file format for numerical data. The next thing to do is to "download" this program into the memory of the Ultragizmo board. To do that, you need to open a third window, the **DEVPORT** window (see Figure 8). Double click the **DEVPORT** icon 

. This will open a new **Tera Term** window connected to COM2 of the PC.

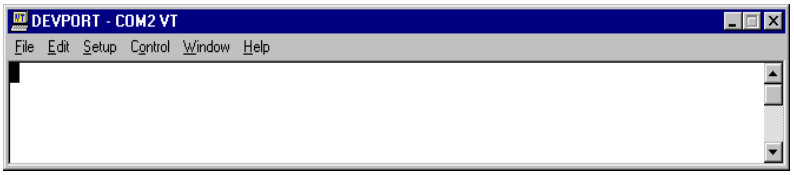


Figure 8 - *The DEVPORT Window*

COM2 is connected to the other serial port of Ultragizmo; through it program files are downloaded into the microprocessor's main memory. In the **DEVPORT** window do the following:

- Select **File->Send File**.
- Select network neighborhood under **Look in**.

- Go to the directory where your *.srec* file is. Note: the *W:* drive is mapped to your *ugsparc* home directory.
- Select (single click) the *.srec* file to be downloaded. Do not open it yet.
- Click on the **CONPORT** window.
- Type **lo [return]** in the **CONPORT** window.
- Click on the **DEVPORT** window.
- Click open. The file will be downloading to the Ultragizmo board and dots will appear.

With a successful download, all that remains is to execute the program, which is done with the monitor **go** command. Switch to the **CONPORT** window. In the **CONPORT** window type:

Ultrag> **go 20000**

This causes the monitor to begin executing the machine language code at location $20000 (note the $ sign which indicates a hexadecimal value), which is the starting address of the program in Figure 7 (the 'org' statement in that program tells the system where to place the machine code). The program should print out the letters from A to Z, and return to the monitor. If this didn't happen, you must look for a problem and fix it. The next section discusses various kinds of problems and errors that can occur. The program as given should work, however.

2.2.2 Assembly Errors and Monitor Tutorial

Errors and bugs are a fact of life in any kind of computer programming. In this tutorial section we introduce you to the types of errors you will encounter, and demonstrate the tools that you will use to find them. You will use these tools extensively in the programs that you create in this laboratory.

Again, before the lab you should create an assembly language source program text file called *prog2.s*, this time copied from the program given in Figure 9. The function of this program is to add a list of ten numbers in memory, beginning at memory location LIST, and place the result into memory location SUM. However, there are several errors in the program, of various types, which we will discover and correct in this tutorial.

After creating the source file, assemble the program on the *ugsparc* workstation in the **TERA TERM SSH** window:

prompt% **a68 prog2.s**

Because there are errors in the program, the assembler returns with these error statements:

error undefined symbol: lop

```
        org     $20000
const   dc.w    $c0
        move.l  #LIST,a0        ; a0 = addr of current list item
        move.b  #10,d0          ; d0 contains loop counter
        clr.b   d1              ; d1 contains SUM, init to 0
loop    add.w   (a0)+,d1        ; add next list item
        subq.b  #2,d0           ; subtract loop counter
        bgt     lop             ; check if done 10

        move.b  d1,SUM          ; put answer in location SUM
        trap    #15             ; exit - return to monitor

        org     $20101
        dc.b    1,2,3,4,5,6,7,8,9,10; data input
        org     $20200
SUM     ds.b    1               ; answer goes here
```

Figure 9 - *M68000 Assembly Language Program prog2.s (with Errors)*

error undefined symbol: LIST

error (13): Non-relocatable expression

There are two assembly language errors. The “bgt lop” instruction has a typing mistake and should read “bgt loop” to correspond with the label preceding the “add.w” instruction. The assembler says that “lop” is an undefined symbol because it used but not defined anywhere in the program.

The second error is that the statement “move.l #LIST,a0” refers to a label that doesn’t exist. To correct this, the label “LIST” should be added to the third last line of the program, so that it reads

```
LIST    dc.b        1,2,3,4,5,6,7,8,9,10
```

Correct these two errors, and re-assemble the program, using the **a68** program. Assuming there are no further assembly errors, download the program into the Ultragizmo as before:

- In the **CONPORT** window, type **lo [return]**.
- In the **DEVPORT** window, select **transfer->send text file**.
- Type **prog2.srec** in the File Name field.
- Click on **Open**.

And then execute the program on the M68000, by typing the following in **CONPORT** window:

Ultrag> **go 20000**

You will see the monitor display the error:

Illegal Instruction at $00020000

This is a rather obscure way of telling you that the M68000 tried to execute an instruction whose machine code it didn't understand. Observe the second line of the program in Figure 9, "const dc.w $c0." This is actually a piece of data that isn't an instruction - it shouldn't ever be executed. Go back to the **TERA TERM SSH** window, delete the "const dc.w $c0" statement, and then re-assemble and download the program.

Now re-execute the program on the M68000:

Ultrag> **go 20000**

You will see the M68000 display the error:

Address Error

This is caused by the statement "loop add.w (a0)+,d1" which is trying to add a word-size operand, but the original address being accessed through register a0 is an odd number (according the "org $20101" statement). Words cannot begin on an odd boundary, and so the processor had to *trap*. The problem is that the program was meant to operate on bytes, not words and so the ".w" extension on the add statement should be a ".b". Change the statement to be "loop add.b (a0)+,d1" and re-assemble and download the program.

Now re-execute the program on the M68000:

Ultrag> **go 20000**

The 68000 should now return immediately with a prompt, since the program only takes a few tens of microseconds to execute. The purpose of the program is to calculate the sum of the numbers from 1 to 10. What should the answer be? What is the answer in base 16? Can you calculate it without a calculator?

To find out if the program worked correctly, we need to check the memory location SUM, so we need to know where in memory the SUM variable is. If you look at the statement above the "SUM ds.b 1" statement in Figure 9, there is an "org $20200" statement which tells the assembler to place the next statement (or variable) at memory location $20200. (Normally you don't have to do this for variables, you just let the assembler assign any location. Since we need to know the explicit location for the purposes of this tutorial, we have used the **org** statement.) Thus the answer should be in memory location $20200.

To examine the contents of a memory location, use the monitor memory display command:

Ultrag> **md -b 20200:1**

This statement says to display (as indicated by the '**md**') the byte (as indicated by the '**-b**') at memory location $**20200**, and display one byte only, as indicated by the '**:1**' (Note: **:1** is a hexadecimal number. How many bytes would be shown if the command **md -b 20200:12** were used?). The answer is displayed in base 16. This should cause the monitor to print the result of the program. You'll notice the answer, in base 16, is *incorrect.* We have planned it that way so that you can learn several debugging skills to determine the error, so you should follow the procedure below to determine the cause of the error.

One way to determine what is going wrong inside a program is to execute each statement one at a time. The monitor provides a *single-step* (also called *trace*) command which does this. Type the following:

Ultrag> **tr 20000**

This command causes the instruction at location $20000 to be executed and to return control to the monitor. The monitor then prints out the contents of all of its registers and the assembly language statement for the *next* instruction to be executed. The last line of the displayed information is a version of the second line of the program, "move.l #10,d0".

You can continue through subsequent statements by simply typing a carriage return - the single step command will be automatically executed. As you do so, check the contents of the registers and see if you can determine the logical error in the program.

The key is to see if the correct operation is being done by each statement, as you understand what the program **should** do. After you reach the "subq.b" statement, you will notice (hopefully) that it is subtracting 2 from the loop counter, rather than 1. This causes the program to add only half the numbers, and hence to produce the incorrect result. Change the "subq.b" statement to subtract only 1 (subq.b #1,d0), re-assemble the program and download it. Check that the number in memory location SUM ($20200) is correct.

By the way, it is a common mistake to correct the source file (*prog2.s* in this case) and then forget to re-assemble the program, and simply re-execute the old program that still resides in memory. It is also easy to forget to re-download the program after assembling it.

2.2.3 Useful Monitor Commands

The monitor has several other commands that you should gain experience with so that you can use them with other programs in this laboratory. A reference guide for all of the monitor commands appears in Chapter 7 of this manual.

In this section, we will walk you through the use of several monitor commands. In these descriptions, we will use the following syntax: if an argument is optional, it is enclosed in square brackets '*[ ]*'. The monitor prompt is the character string "Ultrag>". Also, the following notation will be used:

<*addr*> *:* a hexadecimal integer specifying the memory address

<*count*> *:* a hexadecimal integer specifying the number of elements

<*size*> *:* size specifier **-b**, **-w** or **-l** (byte, word, long word)

<*data*> *:* a hexadecimal integer of size specified by <*size*>

Memory Display Command

The **Memory Display** command is used, as above, to inspect the contents of memory. It has the following general format:

md [<size>] <addr>[:<count>]

The memory display command allows the user to view memory. The size used to display the memory is determined by the <*size*> option. If no option is used the default is word. The starting address is specified by the <*addr*> field. The optional <*count*> field determines the number of memory elements to be displayed.

For example, type the following command which examines 16 bytes of memory beginning at address $20000.

Ultrag> **md -b 20000:10**

It should display a line that looks something like this:

020000: 20 7c 00 02 01 01 10 3c 00 0a 42 01 d2 18 53 00

Now type the next command, which examines eight words (1 word = 2 bytes) of memory beginning at $20000.

Ultrag> **md -w 20000:8**

It should result in something like this:

020000: 207c 0002 0101 103c 000a 4201 d218 5300

Block Fill Command

The **Block Fill** command places data directly into the memory. It has the following general syntax:

bf [<size>] <addr>:<count> <data>

The block fill command fills the specified range of memory with the data listed. If the size option is not specified the default size used is word. The fields have the same meanings as for the **Mem-**

ory Display command except that the <*data*> argument is the data value to be written into the memory locations. It must be supplied as an argument.

Type the following command which will deposit the data value $12 into 16 byte locations starting at address $20000.

Ultrag> **bf -b 20000:10 12**

The next command will deposit the data value $0023 into 8 word locations starting at $20000.

Ultrag> **bf -w 20000:8 23**

You can now use the **Memory Display** command to check that the specified memory locations were filled correctly.

The Disassemble Command

The **Disassemble** command translates machine code instructions back into the original assembly language statement. It has the general form:

md -di <addr>

To illustrate the use of this command you should once more download the (corrected) program of Figure 9 into the memory of the Ultragizmo board. That is, use the **lo** command to copy the *prog2.srec* file back into the M68000 memory, which the above work would have overwritten. The following command interprets the contents of the memory locations starting at $20000 until several instructions are disassembled:

Ultrag> **md -di 20000**

With the program of Figure 9 still in memory, this command will result in:

020000: movea.l #0x20100,a0

and so on.

Note that the disassembler displays hexadecimal numbers (base 16) with a 0x preface. What happens when you use the disassemble command with an incorrect address, e.g. **md -di 20002**?

The Register Display Command

The **Register Display** command displays the contents of the M68000 internal address and data registers. It has the following format:

Ultrag> **rd**

Try this command and see if you can figure out which registers are which. In this tutorial we have referred to registers d0 to d7, and a0 to a7. In your class lectures you will learn about the other registers displayed.

The Trace Command

As described earlier in the tutorial, the **Trace** command (also called the **single-step** command) allows assembly instructions to be executed one at a time. The general form of the Trace command is as follows:

tr [<addr>]:[<count>]

The processor executes the instruction located at address <*addr*> and then returns control back to the monitor. If the <*count*> argument is supplied, then the number of instructions indicated by <*count*> are executed before returning control to the monitor.

Ordinarily, when control is returned to the monitor, it will print out the program counter value (the address of the next instruction to be executed), the dis-assembly of the next instruction to be executed, and the current value of the general purpose registers. Subsequent single step commands may be issued by pressing the return key.

Breakpoint Commands

In some cases, when programs are very large and execute many instructions, it isn't practical to single-step through all of them. An alternative is to use the **Breakpoint** command, which causes the program to temporarily halt when a particular instruction is reached, so that registers and memory locations can be examined.

The general form of the breakpoint command is:

br [-r] <addr>[:<count>]

The breakpoint command causes a break in instruction execution at the specified instruction address. Whenever a breakpoint is encountered during the execution of the user's program, control is returned to the monitor **before** the instruction with the breakpoint is executed. If execution is continued (using the **co** command, as described below) it will start with the breakpoint instruction itself. If the count argument is specified, then execution of the user's program will be interrupted only after the breakpoint address is encountered <*count*> times.

To set a breakpoint, you need to know the physical address of the instruction you want to stop at. To learn the physical addresses, you need to assemble your program using the **a68 -l** command.

prompt% **a68 -l prog2.s > prog2.out**

In the *prog2.out* file you will find the listing as shown in Figure 10.

```
                                    org     #20000

020000: 207c00020101                move.l  #LIST,a0
020006: 103c000a                    move.b  #10,d0
02000a: 4201                        clr.b   d1

02000c: d218            loop        add.b   (a0)+,d1
02000e: 5300                        subq.b  #1,d0
020010: 6e00fffa                    bgt     loop
020014: 13c100020200                move.b  d1,SUM
02001a: 4e4f                        trap    #15

                                    org     $20101
020101: 01020304050607  LIST        dc.b     1,2,3,4,5,6,7,8,9,10
020108: 08090a
                                    org     $20200
                         SUM        ds.b    1
```

Figure 10 - *Assembly Listing of (Correct) prog2.s*

The address of the "move.b d1,SUM" instruction is $20014. Return to the monitor (**CONPORT** window) and set a breakpoint to halt execution when this instruction is reached by typing:

Ultrag> **br 20014**

Now, re-execute the program:

Ultrag> **go 20000**

Notice that the program halts at the required instruction. To continue on from this point type the **Continue** command:

Ultrag> **co**

To remove the breakpoint, use the *-r* option of the breakpoint command:

Ultrag> **br -r 20014**

Note that it is not possible to single-step through or add breakpoints to any terminal input/output routines because single stepping also uses the terminal I/O.

The Help Command

The **Help** command prints a short summary of the monitor commands and their arguments. To get that listing, type:

Ultrag> **help**

2.2.4 Machine Code Tutorial

To truly understand machine code and the final form that assembly-level instructions actually take, you must do some manual labour. As discussed in Section 2.1 on page 4, the assembler program that you used above translates the human-understandable assembly language into the numeric machine codes that the processor understands. These codes are put into the Ultragizmo board memory and then fetched and directly executed by the processor. In this section you will enter the machine code, provide its input data and "manually" inspect the output, using the monitor commands.

The following program calculates the sum of the two numbers in memory locations $20100 and $20101, and places the result in memory location $20103 (note again that the $ sign means that the number is in base 16, or hexadecimal):

```
move.b    $20100,d0    ; copy contents of location $20100 into d0
add.b     $20101,d0    ; add contents of loc $20101 to d0,
move.b    d0,$20103    ; place answer in location $20103
trap      #15          ; exit and return to monitor
```

Read through the above program and try to understand it. The listing below gives the machine code associated with each of the four instructions.

```
020000: 103900020100    move.b    $20100,d0
020006: d03900020101    add.b     $20101,d0
02000c: 13c000020103    move.b    d0,$20103
020012: 4e4f            trap      #15
```

The numbers on the left hand side give the address of the instruction (the first "move.b" instruction is placed beginning at memory location $20000), a colon (:) and then the machine code of the instruction. The machine code for "move.b $20100,d0" is $103900020100. Notice that the memory address within the instruction ($20100) actually appears as part of the machine code.

As part of the tutorial, you should now place this machine code directly into the Ultragizmo board memory using the monitor **block fill** command:

Ultrag> **bf -l 20000:1 10390002**

Ultrag> **bf -l 20004:1 0100d039**

Ultrag> **bf -l 20008:1 00020101**

Ultrag> **bf -l 2000c:1 13c00002**

Ultrag> **bf -l 20010:1 01034e4f**

Note that the four instructions were entered using five long words. Why is that?

Next, put the two (byte-sized) numbers to be added into memory locations $20100 and $20101:

Ultrag> **bf -b 20100:1 5**

Ultrag> **bf -b 20101:1 6**

Now, execute the program using the monitor **go** statement;

Ultrag> **go 20000**

This will execute in less than the blink of an eye, and return to the monitor. Use the monitor **memory display** command to see if the correct value is in memory location $20103:

Ultrag> **md -b 20103:1**

If the answer is printed in base 16, what should it be? Is it correct?

The purpose of this exercise was to illustrate two things:

- that the processor directly executes numeric codes, called machine codes, from memory.
- that the data that a program operates on is stored in that same memory. You entered the data in the same manner in which you entered the code!

Now that you've done these tutorials, you're ready to begin creating your own programs!

3 FPGA Tutorial

The University of Toronto Ultragizmo board contains an Altera FLEX 10K70 Field-Programmable Gate Array (FPGA), called the SFPGA, and is programmed using the Max+plusII CAD software. This chapter provides an introduction and step-by-step tutorial on the use of the Altera 10K70 FPGA and Max+plusII. It assumes that you have some knowledge of a hardware description language (HDL) such as VHDL or Verilog.

This chapter is organized into three sections. Section 3.1 describes a CAD flow for hardware design using FPGAs. Section 3.2 is a tutorial on Max+plusII. Section 3.3 is a guide to the bugs encountered while using Max+plusII and describing hardware using VHDL.

3.1 Programmable Logic

A field-programmable gate array (FPGA) is an integrated circuit containing a number of programmable logic blocks, programmable interconnect, and possibly memory. FPGAs can be reprogrammed to implement different digital circuits. The mapping of a hardware circuit onto the FPGA is usually performed with the aid of a CAD tool.

In a typical CAD flow, a designer describes a circuit using a hardware description language such as VHDL or Verilog. The CAD tool reads in the HDL files, and transforms the design into a set of logic equations, which are then transformed into a netlist of logic units available on the FPGA. This netlist is then physically placed on the FPGA. Routing is then performed, physically connecting the netlist using the interconnect resources. The circuit can then be simulated, using the timing information specific to the FPGA. If the simulation satisfies the design specifications, then a configuration bit string is generated and downloaded onto the FPGA, instantiating the circuit. If design specifications are not met, then the circuit is redesigned and recompiled.

The SFPGA is an Altera FLEX 10K70 FPGA. It contains 70 000 “typical” logic gates (this can be a misleading number - Altera claims that this represents 46 000 to 118 000 “usable” gates, including logic and memory), and nine 2048-bit memory arrays called EABs. The 10K70 FPGA is programmed using the Max+plusII CAD tool, described in the next section.

3.2 Max+plusII Tutorial

3.2.1 Setup

To do this tutorial you should be sitting at a PC workstation attached to an Ultragizmo board. Log onto the workstation using your *ugsparc* login and password.

A typical Max+plusII CAD flow is shown in Figure 11. First, a VHDL description of your circuit is entered. Max+plusII then compiles it onto the selected FPGA device. After that, a timing analyzer determines the critical path through the circuit and maximum operating frequency. Upon meeting this requirement, the circuit can be simulated by stimulating the inputs with test vectors and observing the outputs. If design specifications are met, then the FPGA chip can be pro-

grammed. Otherwise the design can be modified and recompiled. This tutorial will show you how to perform each of these steps.

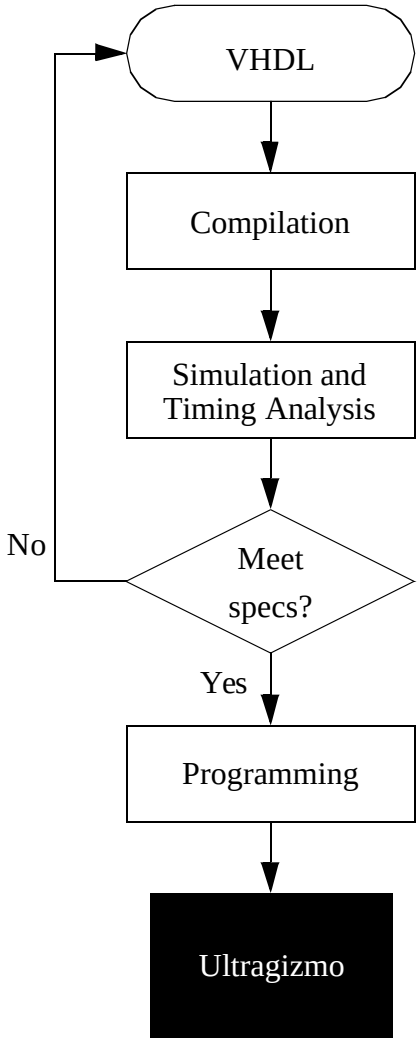


Figure 11 - *Max+plusII CAD Flow*

This tutorial will use files located in the */cad2/ultragizmo/FLabs/tutorial* directory on the *ugsparc* network. Copy the four files (*fulladder.vhd*, *wrapper.vhd*, *wrapper.acf*, and *wrapper.scf*) into a directory called *mp2/tutorial* on your *ugsparc* account. The *fulladder.vhd* file contains the description of a full adder to be used in this tutorial. The *wrapper.vhd* file is a generic VHDL file whose input and output pins correspond to the pins on the Ultragizmo board. See Section 8.5 on page 117 for more details on the signals defined in the *wrapper.vhd* file. The top-level entity of your circuit (the full adder, in our case), is instantiated in the wrapper file, mapping Ultragizmo inputs and outputs to your circuit. The *wrapper.acf* file gives compiler directions; this includes the selection of the 10K70 FPGA as target device as well as pin assignments. The *wrapper.scf* file will be described later in this tutorial. For a description of VHDL please refer to your course's VHDL reference manual.

In the *wrapper.vhd* file, you will notice that the three full adder inputs have been mapped onto the **SFPGA_DIGITAL** connector pins 1, 3, and 5. Referring to Figure 29 on page 131, notice that these correspond to switches SW1, SW2, and SW3 on the digital protoboard. Also, the full adder outputs have been mapped onto led(0) (for output Cout) and led(1) (for output S). The switches and LEDs will be used eventually for testing purposes.

3.2.2 Starting Max+plusII

In order to start Max+plusII, go in the Windows **Start** menu and select **Run**. At the prompt type

max <username>

where **<username>** is your ugsparc login name. You will then be prompted for your year digit. Enter '2' if you are in second year, '3' if you are in third year, '4' if you are in fourth year, and '0' if you are a professor or a graduate student. After a few seconds the Max+plusII window will appear on your screen, and should appear similar to Figure 12.

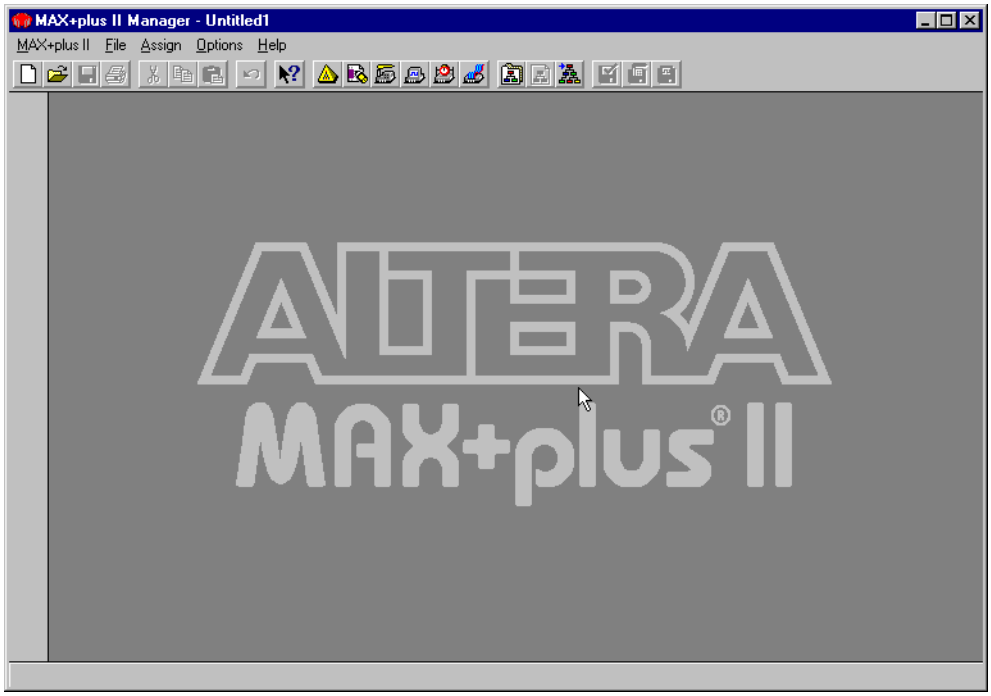


Figure 12 - *The Max+plusII Window*

At the top of the window is the menu bar. The leftmost menu is the Max+plusII menu and includes the list of tools available in Max+plusII. The following tools should be included in the list: Hierarchy Display, Graphic Editor, Symbol Editor, Text Editor, Waveform Editor, Floorplan Editor, Compiler, Simulator, Timing Analyzer, Programmer, and Message Processor. As you use each of these tools, menus relevant to the tool you are using will appear in the menu bar.

Right now we will use the text editor to view and edit our VHDL file. Open the top-level VHDL entity, *wrapper.vhd*:

File -> Open...

NOTE: Max+plusII sometimes has trouble with long file names. If your file isn't displayed in the list, then type in the name in the space provided.

When you start Max+plusII, your *ugsparc* account maps onto the *W:* drive through the **maphome** program (see Section 2.2.1 on page 6 for more details on the **maphome** program). Navigate to your *mp2/tutorial* directory, which you created earlier, select the *wrapper.vhd* file and click OK. You are now in the Max+plusII text editor.

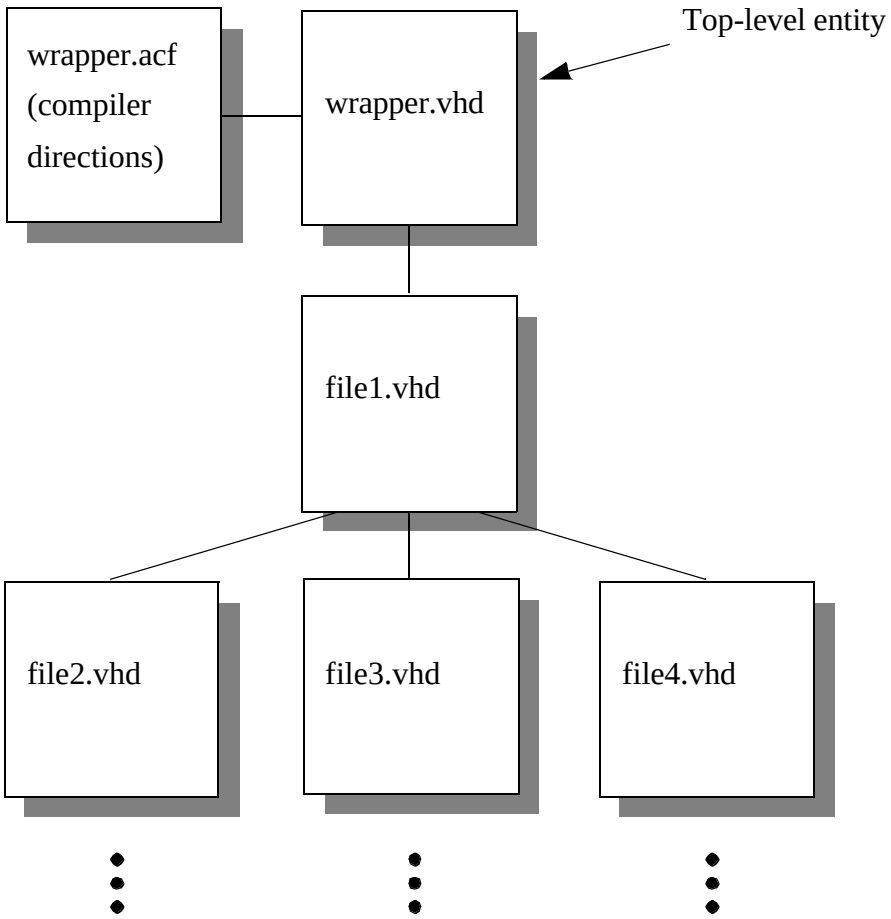


Figure 13 - *Max+plusII VHDL File Structure*

Configuration statements, familiar to those with VHDL experience, are not required in Max+plusII. Instead, Max+plusII associates all files relevant to a design with a project. A typical file structure is shown in Figure 13, including a top-level entity, lower-level entities, and compiler direction files. Instead of including all the relevant files in the project, you only need to point the project to the top-level VHDL entity, which in our case is the currently opened file, *wrapper.vhd*. Max+plusII will look for a *.acf* file of the same name as the top-level entity (the *.acf* file contains the compiler directions), and will find all *.vhd* files pointed to by the top-level file, as long as they are located in the same directory as the top-level entity. In order to let Max+plusII know which file is the top-level entity, do the following:

File -> Project -> Set Project to Current File.

3.2.3 Compiling Your Design

Before compiling a project you must specify the target device, which in our case is the Altera 10K70 FPGA. First select

Assign -> Device...

In the pop-up window scroll down the list of available devices, select EPF10K70GC503-4, and click OK. If you can't find that device in the list, unselect the "Show Only Fastest Speed Grades" box located under the list. A more complete list of devices will then appear. You should notice that the 10K70 is already selected, since it was preselected in the .acf file.

Now you can compile your design. Open the Max+plusII compiler.

Max+plusII -> Compiler

A window with the Max+plusII CAD flow as well as **Start** and **Stop** buttons will appear, as shown in Figure 14. Click on **Start** to start compiling. As each step in the CAD flow is completed, the colour of its box will change. As well, a window containing error and warning messages will appear. In this tutorial you should get two errors and no warning messages. The first refers to an undeclared signal, and the second announces that the compiler can't finish compiling due to errors.

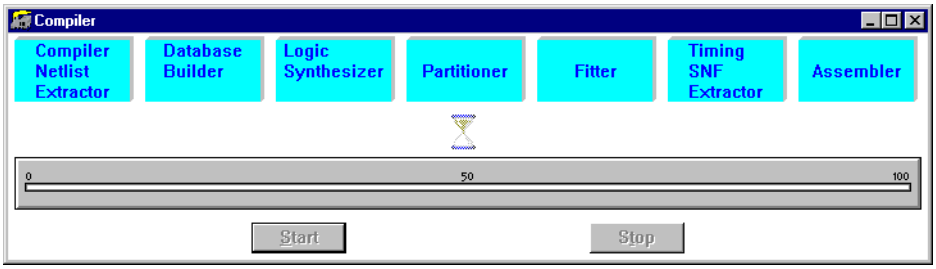


Figure 14 - *The Compiler Window*

Select the first error message (the one pointing to line 22). There are two things you can do to get information about the error. First, you can click on **Help on Message**, which will pop up a help page with instructions on how to correct the error. Second, you can click on **Locate**, which will open the relevant VHDL file and point to the source of the error. In this case, it points to line 22 of the *fulladder.vhd* file, where the signal **Sum** hasn't been declared. In the port map you will notice the signal named **S**. Change **Sum** to **S** in the *fulladder.vhd* file, save it, and recompile. There should be no error messages and several warning messages. Most of these warning messages refer to pins which were mentionned in the wrapper file but weren't actually used in the design. You should always check the warning messages, but in this case you can ignore them.

3.2.4 Timing Analysis

The Max+plusII timing analyzer can be used to determine the critical path of your circuit. To do this, open the timing analyzer window.

Max+plusII -> Timing Analyzer

The timing anayzer window should now be open, with a grid. On the left side you will put your circuit's inputs, and on the top side the circuit's outputs. Select

Node -> Timing Analysis Source...

In the window that pops up, click on **List**, select **sfpga_digital(1)**, **sfpga_digital(3)**, and **sfpga_digital(5)**, click on the right arrow (=>), and click **OK**. The inputs now appear on the left side of the grid. Now select

Node -> Timing Analysis Destination...

Similarly, click on the **List** button, select **led(1)** and **led(0)**, click on the right arrow, and then on the **OK** button. The outputs now appear at the top of the grid.

Then click on the **Start** button. After a few seconds the grid will be updated with delays from each input to each output. The maximum delay should be from **sfpga_digital(3)** to **led(0)**, with a delay of 28.3ns, as shown in Figure 15.

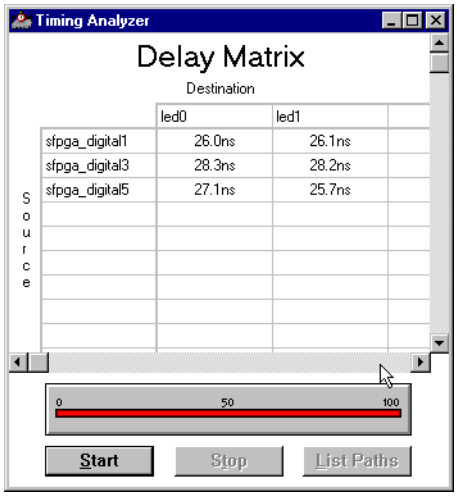


Figure 15 - *The Timing*

3.2.5 Simulating Your Design

Now you are ready to simulate your design. First open the simulation window.

Max+plusII -> Simulator

A window with a start time box, an end time box, and a start button will pop up. However, before simulating you must specify input waveforms. In order to do this you must go into the waveform editor, as follows:

Max+plusII -> Waveform Editor

A waveform window will pop up. You must then select the signals relevant to your design. For each node (**sfpga_digital(1)**, **sfpga_digital(3)**, **sfpga_digital(5)**, **led(0)**, and **led(1)**), select

Node -> Insert Node...

and click on **List** in the window that pops up. A list of nodes will then show up, from which you can select the desired node. When you are done, the five signals should all be shown in the waveform window, and you are ready to specify inputs.

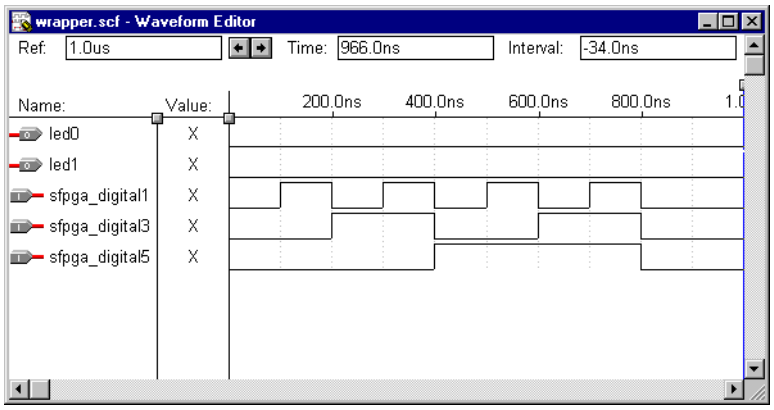


Figure 16 - *Waveform Editor Window*

For each input, the value currently specified is zero. You can change this using the tool bar located on the left hand side of the waveform editor. For example, select one time step from the **sfpga_digital(1)** line. Then click on the tool bar button that contains a 1. You will notice that the waveform changes to a 1 for that time interval. Using this method specify all 8 combinations of the three inputs. You might need to zoom out (**View -> Zoom Out**) in order to see all the 8 combinations at once. On the left you should notice the tools used to create other signal values, and which you should try out (you can create a '0', '1', 'X' (undefined), or 'Z' (high impedance)). The final result should look as shown in Figure 16.

Once you are done specifying an input pattern, you need to save your waveform. Click on

File -> Save

A pop-up window will ask for a file name to save to. Save it as *wrapper.scf* which should be the default. A copy of the *wrapper.scf* file has also been created and is stored in the */cad2/ultragizmo/FLabs/tutorial* directory.

You can now return to the simulator, and click on **Start**. When the simulation is done, return to the waveform window to view the simulation results. They should look like those in Figure 17.

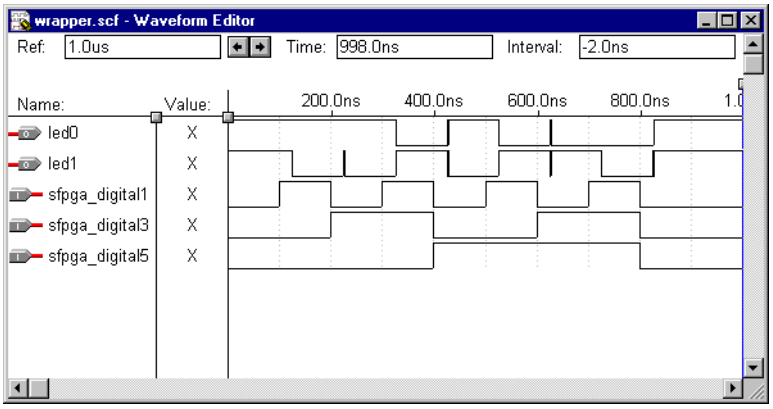


Figure 17 - *Simulation Results*

Notice the glitches and delays. When you compiled your design, you asked the compiler to produce timing information for your circuit. This timing information was used in the simulation to generate the delays you see on the waveforms. You can perform a purely functional (i.e. no timing information) simulation by returning to the compiler and selecting

Processing->Functional SNF extractor

as opposed to the Timing SNF extractor, and by then recompiling and resimulating. Note that you won't be able to download your circuit to the Ultragizmo board if you do a functional extraction.

3.2.6 Programming the Ultragizmo Board

Once you are satisfied that your design is functional and meets your design specifications, you can program the SFPGA chip. You should first ensure that the power to the Ultragizmo board is on and reset the board using its **RESET** button. Then open the Max+plusII programmer:

Max+plusII -> Programmer.

The programming window, and menus now appear. Before programming you must select the port through which the Ultragizmo board is programmed. Select

Options -> Hardware Setup...

A window pops up, in which you select a hardware type. From the pull-down menu select **Byte-Blaster** if it is not already selected. From the other pull-down menu select **LPT1**. Then click **OK**. If there is no check mark next to it in the menu, select

FLEX -> Multi-Device FLEX Chain.

The check mark should appear. Then select

FLEX -> Multi-Device FLEX Chain Setup...

Click on *wrapper.sof*, click **Add**, and then click **OK**.

Now you can return to the programming window and click on **Configure**. You will see the thermometer at the bottom of the programming window indicating the progress of the download. Once completed, a 'Configuration Complete' window should pop up. Your design is now programmed onto the Ultragizmo board.

3.2.7 Testing Your Design

You are now ready to test your design. For this tutorial, the inputs will be provided using the digital protoboard, and the outputs will be displayed on the Ultragizmo board's LED display.

Use a 40-pin connector to connect the protoboard to the Ultragizmo board's **SFPGA_DIGITAL** connector (see Figure 42 on page 92 for the location of the connector). The three inputs to the full adder correspond to protoboard switches SW1, SW2, and SW3. The outputs are displayed on the Ultragizmo's $\overline{\text{LED2}}$ (Sum) and $\overline{\text{LED1}}$ (Cout). You can now use the protoboard's switches to test all combinations of inputs. You will notice that the LEDs give the answers you should be expecting. Note the negations in the LED assignments in the *wrapper.vhd* file. This is because the LEDs are active-low.

3.2.8 Changing Pin Assignments in the Wrapper Files

When you create your own designs you will need to change pin information in the wrapper files. Unused pins on the SFPGA become outputs and are driven low by default. To ensure that unused SFPGA pins do not drive the M68000 bus, all unused pins connected to the bus have to be defined as inputs or as high impedance outputs.

Suppose you want to change the $\overline{\text{DTACK}}$ signal in the wrapper files from an input to an output. You would follow these steps. First, change the $\overline{\text{DTACK}}$ declaration in the *wrapper.vhd* file from an **input** to an **output**. Next, you need to change the pin information in the *wrapper.acf* file. Open the pin assignment window:

Assign->Pin/Location/Chip...

From the **Existing Pin/Location/Chip Assignments** scroll menu, select the **dtack** pin, labelled as an **input** on pin AG3. Then, from the Pin Type scroll menu, select **output**, and then click on **Add**. You will notice that Max+plusII added a $\overline{\text{DTACK}}$ **output** pin on AG3, but also left the old **input** pin. You need to delete the $\overline{\text{DTACK}}$ **input** pin. Select it, click on **Delete**, and then on **OK**.

You are now done the tutorial and can move on to more interesting and useful designs!

3.3 Error Messages in Max+plusII

This section is an aid in troubleshooting your design.

Error message about the design being too large

This indicates that your design is too large to fit onto the 10K70 FPGA. You need to either make it smaller, target it to a larger FPGA, or partition it onto several 10K70 FPGAs which you can connect together via a 40-pin connector.

"Found illegal use of a statement in a declarative part"

You have instantiated a component inside a process. Remember that statements inside a process occur sequentially every time a process is invoked; that means that the instantiation happens every time the process gets called, and you end up with millions, billions... of copies of your component. That's probably not what you want... and Max+plusII doesn't like that either. You can fix this by putting your component instantiation outside the process.

"Conditional statement in this region for signals not supported"

You have an IF statement dependent on a clock nested inside another IF statement. Max+plusII (and synthesis tools in general) has trouble mapping this into hardware. You must rewrite your code so that the outer IF statement is the one dependent on the clock.

"SRAM load unsuccessful"

This usually happens when you haven't properly specified which parallel port to use. Go into Options -> Hardware Setup... and RE-select the LPT1 port.

4 Assembly Language Laboratory Experiments

This section of the manual contains several laboratory experiments. Each requires you to write a Motorola 68000 assembly-language program and typically specifies an amount of preparation to be done in advance of the lab. It is usually not possible to do the preparation during the lab period itself, so it is strongly advised that you prepare in advance. Your TA might require you to hand in a lab preparation in which should always include a listing file of your assembled program (see Section 2.2.3).

It is essential that you enter your program into the *ugsparc* workstation computers in advance of the lab, and that you assemble the program without assembly errors.

Chapter 6 through Chapter 12 of this manual contain information that may be necessary to complete these experiments. The title of a lab gives in parentheses the main components of the Ultragizmo board that you are to use for that lab.

Lab M1 (M68000) Assembly Language Programming

The purpose of this lab is to familiarize you with Motorola 68000 assembly language instructions, addressing modes and debugging.

You are to write a Motorola 68000 assembly language program that takes two numbers, **s** and **p**, as input. (These numbers will be incorporated as part of the source code of the program, but you should treat them as variables, not constants).

The program should create a matrix of **bytes**, (of size **s** x **s**): $M_{0,0}$ $M_{0,1}$... $M_{s-1,s-1}$ in memory, and fill all of it, except the diagonal, with zeroes. The diagonal (elements $M_{0,0}M_{1,1}$... $M_{s-1,s-1}$) should be filled with the value **p**.

Hint: An **s** x **s** matrix of bytes, beginning at address **A**, is represented in memory by **s** consecutive sets of **s** bytes. The **address** of matrix element (*i, j*), where $0 \le i < s$ and $0 \le j < s$ is given by:

$$\text{Address of element } M_{i,j} = \mathbf{A} + i \times \mathbf{s} + j$$

Check that your program works by using several different values of **s** and **p**.

Preparation: Write and enter the M68000 assembly language program into the workstation computers. Read Section 6.1 to learn the syntax rules for writing an M68000 assembly language program. Assemble the program and eliminate any errors reported by the assembler.

Lab M2 (M68000) A Simple Instruction Interpreter

The purpose of this lab is to familiarize you with the Motorola 68000 address modes and assembly language programming.

You are to write a Motorola 68000 program that acts as a simple "interpreter." Your program will "read" the "instructions" (not M68000 instructions, but a simple set described below) from memory and do one of a few simple calculations depending on an operation code. The instruction in memory will also contain the data to be operated on, and the address of the next instruction to be executed.

The calculations are stored in a word-length "accumulator" located in main memory.

The format of the instructions, which consist of three items of different size, is as follows:

Item #1 (word size):Op Code:0 = Clear accumulator
1 = Add number to accumulator
2 = Subtract number from accumulator
3 = Exit program
Item #2 (word size): Number to be operated on, if applicable
Item #3 (long word): Address of next "instruction," if applicable

Your program should assume that the first "instruction" is always labelled **START**. For example, the instruction created by the following assembly language define statements:

```
INSTR   dc.w    1
        dc.w    107
        dc.l    $20500
```

is an instruction to add the number 107 to the accumulator, and then to go find the next instruction at address $20500. As another example, the following "program" adds the number 77, subtracts the number 15, and then exits. Notice the use of **org** statements illustrates that different instructions do not have to follow each other in consecutive memory locations.

```
        org     $20000
START   dc.w    0
        dc.l    INSTR1
        org     $20100
INSTR1  dc.w    1
        dc.w    77
        dc.l    INSTR2
        org     $20400
INSTR2  dc.w    2
        dc.w    15
        dc.l    FIN
        org     $20500
FIN     dc.w    3
```

To summarize, you are to write a program that reads these “instructions” from memory, beginning with the instruction at address **START**, and executes the instructions as described above.

If the “program” contains an op-code other than a legal one (0, 1, 2, or 3), your program should exit with the number $FFFF in the accumulator.

You should test your program with different “programs” that you create yourself. Use the monitor program to check the contents of your accumulator after the “program” has executed.

Preparation: The program described above, along with a listing file of the assembled program. Don’t forget to comment your program. Include with your preparation a sample input “program” for your interpreter.

Lab M3 (M68000) Subroutines and Fibonacci Sequence

As your programs get larger, you will find that they become more difficult to understand. With all the registers being used for different purposes, and branch instructions sending control every which way, you will find that there will come a point when you can no longer understand your own program! Subroutines provide a solution to this problem. By structuring your programs with subroutines, you can better understand your program. Properly-written subroutines allow you to make assumptions about parts of your program, so you can use a subroutine knowing what result to expect but without understanding exactly how it works. This means you don't have to keep your entire program all in your head at the same time, which is very important for large programs.

A subroutine has the following properties:

1. It is always called with "bsr" (Branch to Subroutine) or "jsr" (Jump to Subroutine).
2. It always ends with "rts" (ReTurn from Subroutine).
3. You never branch out of the middle of one subroutine into another (although using bsr or jsr is fine).
4. There is a convention (caller-save or callee-save) for preserving the contents of the registers across a subroutine call. You will learn more about calling conventions in lectures. The Lab M4 handout gives a more detailed discussion on calling conventions. Using caller-save, the program calling the subroutine pushes the registers onto the stack and pops them when the subroutine ends. Using callee-save, the subroutine does the pushing and popping.

The purpose of this lab is to practice writing and using subroutines. You must compute the terms of the Fibonacci sequence, in which each term is the sum of the two previous terms. The first two terms (terms number zero and one) are both equal to one, and the sequence progresses as follows:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Your assignment is the following:

1. Write a subroutine which, given a term number (i.e. the n*th* Fibonacci number), computes the value of that term. You must use recursion to perform this computation. You must use either the caller-save or callee-save convention to save all registers needed by the subroutine, except the one which contains the result.
2. Write a program which calls this subroutine, passing it a value from a certain memory location. It should then store the result in another memory location.

Preparation: Write and enter the M68000 assembly language program into the workstation. Assemble the program and eliminate any erros reported by the assembler.

Bonus (1 mark): Write two versions of the program: one using caller-save and the other using callee-save.

Lab M4 (M68000) Recursive Binary Search

The purpose of this lab is to gain experience using subroutines and manipulating the program stack in a consistent manner. The manner in which the program stack grows, shrinks, and is accessed is known as a calling convention. You will learn about caller-save and callee-save calling conventions in your lectures. To increase the opportunities for re-use, compiler writers and assembly-language programmers for a particular processor such as the M68000 or the MIPS typically agree upon a common calling convention that will be followed. Thus, any compiler for the M68000 processor would use one convention while compilers for the MIPS or Pentium processors might use a different convention. There are many advantages to using a common calling convention. It makes the compiler's job easier. It allows files to be compiled separately and later be linked together with little difficulty. It also allows subroutines that are developed by one programmer to be used by any other programmer. This is how libraries of functions (e.g., the C math library or string library) work. Finally, a common calling convention allows inter-language calls. For example, a C subroutine can call a FORTRAN subroutine and vice-versa.

To gain experience with manipulating the program stack, you will implement the following recursive subroutine:

```
/* Global Variables */
int key;
int Numbers[100] = {
   28, 37, 44, 60, 85, 99, 121, 127, 129, 138,
   143, 155, 162, 164, 175, 179, 205, 212, 217, 231,
   235, 238, 242, 248, 250, 258, 283, 286, 305, 311,
   316, 322, 326, 351, 355, 364, 366, 376, 391, 398,
   408, 410, 415, 418, 425, 437, 441, 452, 474, 488,
   506, 507, 526, 532, 534, 547, 548, 583, 585, 595,
   603, 621, 640, 661, 666, 690, 692, 713, 719, 750,
   755, 768, 775, 776, 784, 785, 791, 797, 798, 804,
   828, 842, 846, 858, 884, 887, 890, 893, 908, 936,
   939, 953, 960, 970, 978, 979, 981, 990, 1002, 1007,
}
int BinarySearch( startIndex, endIndex, NumCalls )
int startIndex, endIndex, NumCalls;
{
 int keyIndex,middleIndex;
 NumCalls++;
 if (startIndex > endIndex) return -1;
 middleIndex = startIndex + (endIndex-startIndex)/2;
 if ( key < Numbers[middleIndex] ) {
   keyIndex = BinarySearch( startIndex, middleIndex-1, NumCalls );
 } else if ( key == Numbers[middleIndex] ) {
   keyIndex = middleIndex;
 } else { /* key > Numbers[middleIndex] */
   startIndex = middleIndex+1;
   keyIndex = BinarySearch( middleIndex+1, endIndex, NumCalls );
 }
 Numbers[ middleIndex ] = -NumCalls;
 return keyIndex;
}
```

This subroutine uses a recursive binary search to look through a sorted list of 100 positive numbers (`Numbers`) for a number (`key`) and returns the position of that number in the sorted list (`keyIndex`). Positions are numbered from 0 to 99. If the number is not in the list then -1 is returned. Additionally, the number of times `BinarySearch`() is called is recorded in the local variable `NumCalls`. After an element in the `Numbers` array is examined, it is replaced by `-NumCalls`. By examining `Numbers`, you can see which elements have been examined and in what order they were looked at. Negative values are stored to distinguish them from the original numbers.

As an example, searching for key=418 returns the position 43 and changes `Numbers` as follows:

```
 28   37   44   60   85   99  121  127  129  138
143  155  162  164  175  179  205  212  217  231
235  238  242  248   -2  258  283  286  305  311
316  322  326  351  355  364   -3  376  391  398
408  410   -4   -6  425   -5  441  452  474   -1
506  507  526  532  534  547  548  583  585  595
603  621  640  661  666  690  692  713  719  750
755  768  775  776  784  785  791  797  798  804
828  842  846  858  884  887  890  893  908  936
939  953  960  970  978  979  981  990 1002 1007
```

Although there may be simpler and more efficient ways to program a binary search, the purpose of this lab is not to implement an efficient binary search. The reason for implementing binary search in this way is to provide a short example that requires you to grow, shrink, and access the program stack in a consistent manner. If you do not manipulate the stack in a consistent manner, the recursive function will most likely cause the stack to grow and use up all the Ultragizmo board's memory.

You are to write the M68000 assembly code for the above subroutine. Store the global variable `key` in register d0 and the address of the global array `Numbers` in a0. Pass the parameters `startIndex` and `endIndex` in the registers d1 and d2, and return the value in register d3. Use the caller-save convention for saving and restoring registers. Also write a main routine that will make the first call to your subroutine as follows: `BinarySearch(0,99,0)`. A copy of the list of numbers can be found in *ugsparc:/cad2/ultragizmo/MLabs/binarySearch.s*.

Preparation: Write the M68000 program described above. Make sure you comment your subroutines well and describe how various registers are being used in your subroutine. For example, you should state which registers store the variables `middleIndex` and `keyIndex`.

In the lab: Show that your program works correctly by searching for different numbers and after each search, examining the contents of register d3 and the `Numbers` array in memory.

Bonus (1 mark): Use the callee-save convention for saving and restoring registers.

Lab M5 (DUART) Program-Controlled Input and Output (Polling)

1 Introduction

The purpose of this lab is to demonstrate the method of computer Input/Output known as **programmed I/O** or **polling I/O** (see Section 8.1 for a description of polling on the DUART). Its purpose is also to make clear the difference between the representation of data inside a computer, and its interpretation once it is sent to an I/O device.

The I/O device that we will use in this lab is the terminal that you have already used to communicate with the Ultragizmo board. The board is connected to the terminal in the following way:

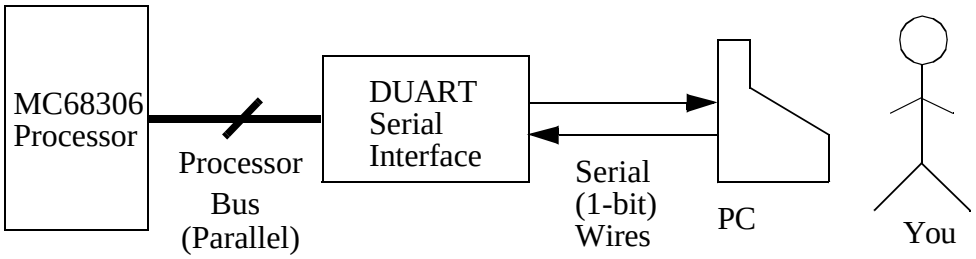


Figure 18 - *Processor, DUART, and PC Interface*

Communication from the processor through the DUART (Dual Universal Asynchronous Receiver-Transmitter) to the terminal is done by accessing registers on the DUART as described in class. Note that while these registers are accessed as ordinary memory addresses, they are located not in the main memory DRAM chips but in the DUART itself. Section 8.1 on page 94 explains how to access the DUART registers and how to program the DUART so that you can communicate with the terminal.

2 Program

You are to write a program that reads two 2-digit **octal** numbers from the terminal, adds them, and prints out the result in octal. (To be clear: each digit of the 2-digit combination is selected from 0s, 1s, 2s,..., and 7s). Do not assume that exactly two digits will be typed in, but that the end of each number is indicated by a carriage return character. If only one digit is typed, followed by a carriage return, then the number should be considered as a 2-digit number with 0 as the left digit and the typed digit as the right digit. If several digits are typed, then the last two digits to be typed before a carriage return should be considered to be the two valid digits. You should “echo” the numbers on the screen as they are typed. You do not need to consider the case where the sum of the two numbers is larger than 2 digits.

Note that when a character comes in from the terminal, it is in ASCII code. The ASCII code for the *character* 0 is $30, and for the character 1 it is $31. The ASCII code for a carriage return is $0d, and for a line feed is $0a. Other ASCII codes can be found in Table D.2 of the **Computer Organization**, by Hamacher, Vranesic, and Zaky, Fourth Edition.

Note also that the terminal interprets bytes sent to it to be displayed as ASCII characters. You may also wish to use terminal control sequences as described in Table 7, “Terminal Control Sequences,” on page 96 to clear the screen and move the cursor.

Preparation: Read Section 8.1 on page 94 which describes how to program the DUART. Write the assembly language program as described above. Be sure to have it typed in and assembled without errors before the lab begins. Remember to hand in the listing file.

In the Lab: You will not be able to use breakpoints or single-stepping to debug your program. See Item 1 on page 89 for a fuller explanation.

Lab M6 (DUART & Interrupts) Interrupt-Driven Output and Polling-Driven Input

The purpose of this lab is to illustrate the use of interrupt-driven Input/Output and compare it to the polling-driven I/O used in the previous lab. It will also be an exercise in understanding the complexity of handling “simultaneous” input and output.

You are to write a program that continuously outputs characters to the terminal display and simultaneously reads input from the keyboard.

- The output MUST be done using Interrupt-Driven I/O, as described in class.
- The input may be either polling-driven or interrupt-driven.

The program is to work as follows: The output should continuously output the character ‘N’ to the screen if no input has been typed. Whenever a character has been typed, then that character should be printed 100 times (or a sufficient number of times so that you will be able to see it, since the output will go by very quickly).

A sample output stream might look like:

NNNNNNNNNNNNNNNNNNNNNNaaaaaaaaaaaa...aaaaaa[100 times a] NNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNbbbbbbbbbb...bbbbbbbbc-
ccccccccccc....ccccccccNNNNNNNNNNNNNNNNNNNNN....

Here the user has typed an ‘a’ then a ‘b’ then a ‘c’.

For a description of DUART interrupts, read Section 8.1.2 on page 97.

Preparation: The program.

In the Lab: You will not be able to use breakpoints or single-stepping to debug your program. See Item 1 and Item 2 on page 89 for a fuller explanation.

Bonus (2 marks): Augment your program to allow nested interrupts.

Lab M7 (DUART & Interrupts) Input and Output Using Polling and Interrupts

The purpose of this lab is to contrast the use of polling (program)-driven Input/Output with interrupt-driven I/O. You will implement the same program using both methods.

1 The Basic Program

The basic program reads four characters from the terminal and outputs those four characters 20 times each. That is, if the user types ABCD, then the program prints “AAAAA....” [20 times] then “BBBB...”[20 times] and so on. In this simple version, the user then types another four characters and they are then displayed 20 times each, and so on. **No characters should be printed until four characters have been typed. The program should then wait until another four characters have been typed, and so on.**

2 Simultaneous Input and Output Using Polling I/O

That’s the easy part. To get a feel for polling I/O control of simultaneous tasks, the program should operate in the following way: it should appear to be generating output and receiving input at the same time. The program should continuously, (and slowly), output the most recently entered four characters in the manner described above. At the beginning, before any numbers are entered, it should output only X’s. Put a delay loop (carefully) between each character output, so that the output appears slowly, about one character every quarter of a second. Note that the “dbra” instruction operates on words, not long words. **While the output is going on,** your program should also be reading the keyboard to receive a new set of four characters, which, once entered, will cause the output to change. From this discussion, it should be clear that the program operates continuously, receiving input characters and displaying the most recent set. **In this part, you should use only the polling-driven input/output technique.**

3 Interrupt-Driven Version

Implement the same program as in part 2, only this time using interrupts. That is, continuously read in the four characters, while continuously displaying the repetitions of the most recently entered set of four characters. The input of characters should be done using interrupts. **You should use the DUART interrupt method as described in class and in Section 8.1.2.** In the main program, the output should be done using the polling technique. If you wish, however, you may try to do both the input and the output using interrupts. This is somewhat more challenging. **To the user, both programs should appear to operate the same way.**

Preparation: The programs for parts 2 and 3. If you think it is helpful, you can write a separate program that implements part 1, but it isn’t required. For the interrupts, read Section 8.1.2 on page 97.

In the Lab: You will not be able to use breakpoints or single-stepping to debug your program. See Item 1 and Item 2 on page 89 for a fuller explanation.

Lab M8 (DUART & Interrupts) Counter and Interrupts

In programs dealing with many different I/O devices, it becomes cumbersome and wasteful to poll them all. A typical system might have to check a keyboard, terminal, mouse, disk, network, or modem, while performing other computations at the same time. With all these devices, a system could spend all its time polling, and never get any work done!

Fortunately, polling is not the only way to interact with a device. You can cause a device to interrupt your program when a certain event occurs. This allows your program to go about its business, ignoring the device until something happens. This makes your program more efficient, and in many cases it makes the program simpler as well, since most of the program does not have to worry about that device.

In this lab, you will write two programs which will demonstrate the differences between polling and interrupts. Both programs will have the same functionality: they will simply increment a counter, and print the counter's value on the screen (in octal) whenever any key is pressed. You should print each value at the beginning of a new line on the screen. You must use a counter capable of counting to at least one billion (how many bits does this require?).

Your task is the following:

- Write the program using polling to receive the keyboard input.
- Write the program again using interrupts to receive the keyboard input.

Your programs will be marked partly based on their speed of incrementing, relative to the standard benchmark programs. You should put some effort into making your programs efficient.

Preparation: The two programs. For interrupts, read Section 8.1.2 on page 97 and Section 8.2 on page 100.

In the Lab: You will not be able to use breakpoints or single-stepping to debug your program. See Item 1 and Item 2 on page 89 for a fuller explanation.

Lab M9 (PIT) Hex Keypad

In this lab you are to connect a hex keypad to the Ultragizmo board using the Parallel Interface/Timer (PIT) port. This involves hooking up the wires and writing the software to make the interface work. A description of the hex keypad is given in Section 9.3 on page 135.

Write a program to do the following:

- Determine when a button on the hex keypad is pressed, and which button is pressed. When a key is pressed, because it is slow and mechanical, it will bounce open and closed for about 10 milliseconds. Make sure that the physical connection has stopped "bouncing" - that is, it has made a firm connection. You can do this by waiting about 10 milliseconds after the first connection is detected and checking to see that the same button has been pressed. A 10 millisecond delay can be generated by executing any short loop roughly 600 times.

- Print out the button label on the screen.

1 Exercise 1 Memory-Mapped I/O Exercise

This exercise is described in Section 8.3.2 on page 113.

2 Exercise 2 Interface Suggestions/Partial Solution

1. In a similar manner to Exercise 1, using the protoboard, plug the hex keypad into the PIT connector. Make sure that the ground on the Ultragizmo board is connected to the ground on the proto-board so that the logic probe will work correctly. If the keyed connectors are correct, then the following connections will be made:

PA0 -> R0	PA4 -> C0
PA1 -> R1	PA5 -> C1
PA2 -> R2	PA6 -> C2
PA3 -> R3	PA7 -> C3

2. In your program, using the Port A Data Direction Register, configure PA0, PA1, PA2, PA3 as inputs (0's in the PADDR positions), and PA4, PA5, PA6, PA7 as outputs (1's in the PADDR positions).

3. Write 0's into wires PA4 -> PA7 (the outputs).

4. Any input that is not connected to anything (an unconnected wire) is set up inside the PIT to be read as a binary 1. You need this fact in the following suggested procedure.

Read PA0 -> PA3. If any button is pressed, then one of these bits will be 0. Otherwise it will be a 1. If none of the wires are 0, then no key is pressed. Keep checking the input until one of them is 0, since this will tell you when a key is down.

5. After a key is pressed, make PA0 -> PA3 into outputs and PA4 -> PA7 into inputs.

6. Write 0's into PA0 -> PA3.

7. Read PA4 -> PA7.

8. With the information from steps 4 and 7 (reading PA0 -> PA3 and PA4 -> PA7) you have enough information to determine which key has been pressed. Figure it out, and print out the letter of the key on the terminal. Remember to wait until the key stops bouncing.

Preparation: The first exercise is to be done before completing the lab. It is intended to show, in a direct way, the link between memory locations and the physical world. The second exercise consists of suggestions on how to do the lab. The techniques learned in Exercise 1 will be very helpful in debugging this lab.

Lab M10 (PIT) Parallel I/O and the LEGO Motors and Sensors

The purpose of this lab is to learn how to use the Parallel Interface/Timer (PIT) to drive the LEGO motors and read the LEGO sensors.

- Do the "Getting Started Tutorial" in Section 10.3. This will save you time in learning how the LEGO system works.
- You are to build a light-tracking device using the Ultragizmo board, the LEGO driver board and the LEGO kit, as pictured below.

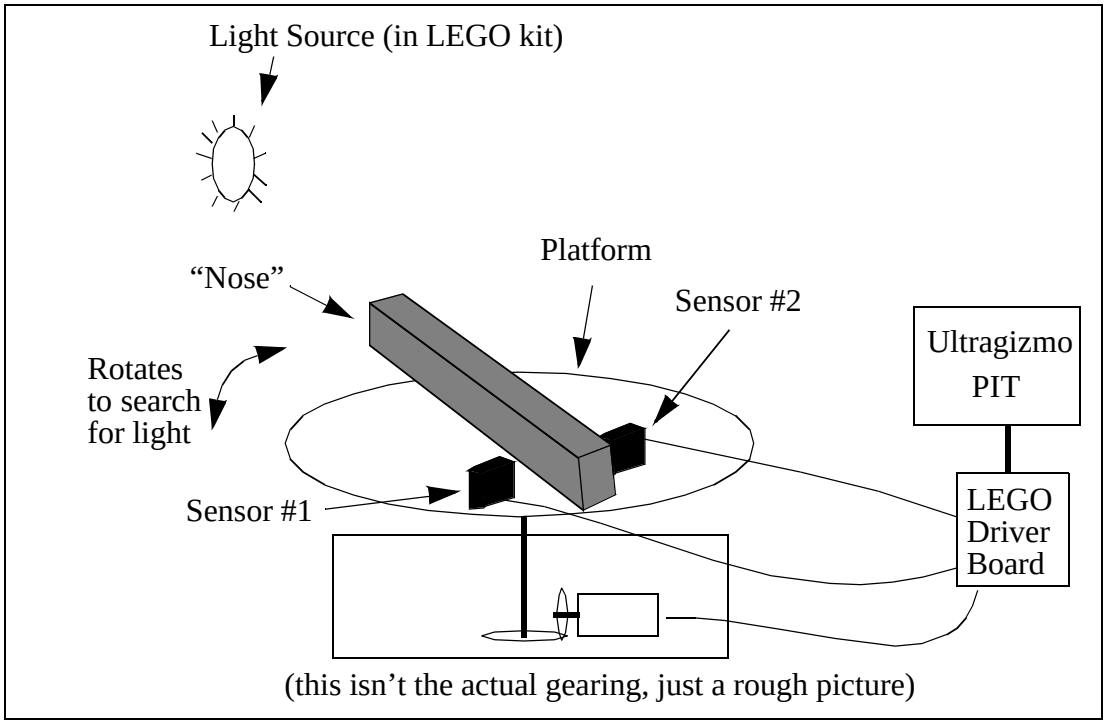


Figure 19 - *LEGO Nose Diagram*

- The basic idea is this: the M68000 will control a motor (via the PIT and the LEGO driver board) that rotates a platform, as illustrated above. On that platform are two light sensors, separated by a "nose." You must build the platform and write an M68000 program that will rotate the platform until the nose is pointing at a light source. The trick is that the nose prevents the light source from shining on both sensors until the light is directly in front of the nose. Your program should attempt to line up the nose with the light source as quickly as possible. This means your program should choose the direction of rotation of the platform intelligently – it shouldn't just always rotate in one direction until both sensors are illuminated.

Each LEGO kit contains several booklets of sample constructions. Booklet E describes the building of a LEGO robot arm. Use the diagrams in this booklet to build the base of the device described above. Steps 1 through 8 on pages 3 through 5 (and sub-steps 1-4 on page 5) show you

how to create a rotating platform. After that you have to construct the platform, nose and sensor assembly.

The light will be a LEGO light that can be powered directly from the LEGO control board, through the connector on the top (see Figure 55 in the LEGO Control Board section of this manual).

Preparation: Read Chapter 10 of this manual, which describes how the LEGO motors and sensors work, and how they are connected through the LEGO driver board to the Ultragizmo board's PIT. Study the LEGO pictures and try to get a sense of how to build the device described above. Be sure to read the tutorial in Section 10.3. Write and assemble the assembly language program described above.

Lab M11 (CODEC) Playing with Sound: A/D, D/A Conversion & Signal Processing

The purpose of this lab is to use the capabilities of the CODEC (COder-DECoder) to do something interesting with sound. Along the way, you'll encounter some basic Digital Signal Processing concepts. You may be motivated to use this hardware in your course project.

In this lab you are to use the CODEC, described in Section 8.6 on page 127, to do two things: make a simple voice delay box, and a voice "speed changer." In a voice delay box, when you speak into the microphone, your voice comes out of the speaker a programmable amount of time later. In the speed changer, you record your voice (or any sound) and play it back at different rates, altering its pitch.

In this lab, you are to write two programs and do three parts:

1 Basic Program

A program that simply takes the input signal and outputs it is in *ugsparc:/cad2/ultragizmo/MLabs/codec.s*. It will read in the a value from the microphone and output it to the speaker immediately, in an infinite loop. This makes the system act just like a wire - what comes in goes right out again. Input this program and execute it to be sure that it works.

2 Delay Loop

Write a program to delay the output of your voice for two seconds of time after you speak it. This would be implemented by storing enough samples from the CODEC input channels in memory so you can wait for the required amount of time to output it. Be sure that your program is always reading in the sounds as it is writing it back out (i.e. it should always be reading and playing back at the same time in a continuous loop. Marks will be deducted for any program that reads blocks of sound and plays them back later without reading at the same time).

3 Frequency Shift

Write a program to read in several seconds of sound, and then play it back at a different rate (faster or slower, programmably). Hint: in the first two cases the CODEC status register provided the timing between output samples. In this case, you must still output values whenever the CODEC status register is ready, but you must be careful which values to output. Note that in this part you should read a block a data into memory, and then output the result; you don't need to produce sound continuously.

Preparation: The programs for parts 2 and 3. You should describe your method for altering the pitch in part 3.

Lab M12 (PIT & Interrupts) Interrupt-Driven I/O Using the LEGO Sensors

The purpose of this lab is to learn interrupt-driven synchronization of inputs. You are to build and program a motor controller that is controlled by an interrupt-driven light sensor. Whenever the light beam is interrupted the motor should change direction. The main program should also read commands from the keyboard: if a ‘d’ is typed, then the interrupt should be disabled. If an ‘e’

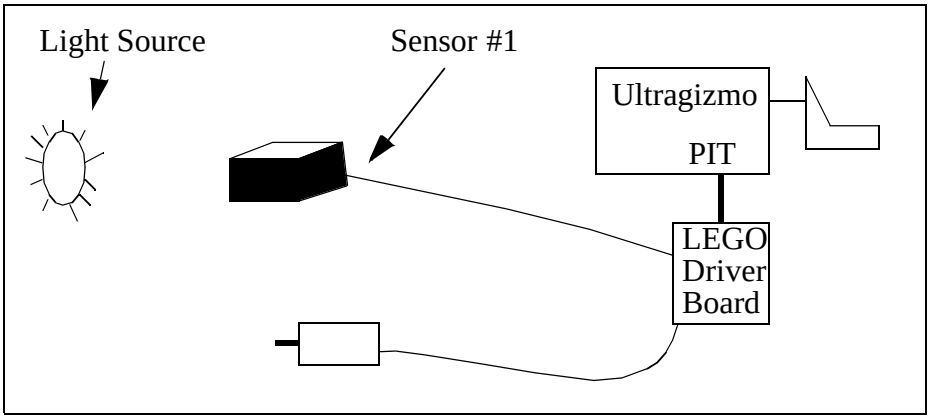


Figure 20 - *LEGO Sensor and Light Source*

is typed, then the interrupt should be enabled. If a ‘q’ is typed, then the program should exit, after turning off all of the interrupt enables.

Your program should have the following parts:

- Initialization - sets up the interrupts.
- Main program - reads the keyboard input as described above.
- Interrupt Service Routine - changes direction of the motor, resets interrupts.

If you have time, build the motor into a car (you can make one using the base you made in Lab M10, or look at the back of booklet E in the LEGO kits for some ideas), that is controlled by waving your hand! (optional).

Preparation: Read Section 10.7 of this manual, which describes how interrupts can be used with all four sensors. Write the program described above.

In the Lab: You will not be able to use break-points or single-stepping to debug your program. See Item 1 and Item 2 on page 89 for a fuller explanation.

Bonus (1 Mark): Attach a second light sensor to the LEGO board and rewrite your motor controller so that whenever the light beam to this sensor is interrupted the motor turns on or off. You program will therefore have to enable interrupts from both sensors, and your interrupt service routine must determine which sensor caused the interrupt and take the appropriate action (either reversing the motor direction or turning the motor on or off).

5 FPGA Laboratory Experiments

This section of the manual contains several laboratory experiments involving the Ultragizmo board's SFPGA. Each requires you to design a piece of digital hardware using VHDL, and some require some M68000 assembly language coding. The descriptions typically specify an amount of preparation to be done in advance of the lab. It is usually not possible to do the preparation during the lab period itself, so it is strongly advised that you prepare in advance.

It is essential that you should have entered your VHDL code into a workstation computer in advance of the lab, and compiled it without errors. Other chapters of this manual contain information that may be necessary to complete these experiments.

For each lab you will require wrapper files. The files, *wrapper.vhd* and *wrapper.acf*, are located in the */cad2/ultragizmo/FLabs* directory on the *ugsparc* network. You should copy the relevant files before each lab. As well, for some labs, you may be given some VHDL skeleton code, to which you have to add your own code. This code is located in the same directory, under a subdirectory with the lab name (e.g. Lab F3 is under */cad2/ultragizmo/FLabs/F3*).

Lab F1 Altera Software Introduction and Use

The purpose of this lab is to learn the basics of the Altera Max+plusII design software, including design entry, simulation and compilation. It will also introduce a large-scale programmable logic device, the FLEX 10K70 (SFPGA on the Ultragizmo board), and show you how to download a circuit onto the device. There are two parts to the lab. The first is the design of a simple combinational logic function, with the primary goal of learning how to use the Altera software. The second part is the design of a more complex logic circuit that will be useful in later parts of this course.

1 Preparation

1. Do the tutorial in Chapter 3.
2. Design, enter and simulate a circuit, using VHDL as the primary entry method, that implements the following logic function:

   You are to design a circuit which adds four 4-bit numbers, A, B, C, and D. Your design should include register to store the sum, an active-low reset, which resets the sum to 0, and a clock. The four numbers are to be entered one at a time on positive clock edges. The result should be displayed on the LED display on the Ultragizmo board. Remember that you need to store 6 output bits.

   Show truth tables for for each bit position in the output, and derive the corresponding Boolean expressions. Enter the simplified Boolean expressions into Max+plusII as VHDL code that represents the adder.
3. You should modify the wrapper file in the */cad2/Ultragizmo/FLabs* directory to instantiate your circuit, and to connect the appropriate input and output pins. Your inputs should come from the **SFPGA_DIGITAL** inputs in the *wrapper.vhd* file.

In the Lab:

1. Before downloading your designs into the board, connect a 40-pin cable from the Ultragizmo board **SFPGA_DIGITAL** port into a digital protoboard - a TA will demonstrate this. Notice that the cable can only plug into the header one particular way, because the headers have “keys” which prevent incorrect insertion. The header itself is numbered as follows, as viewed from the top:

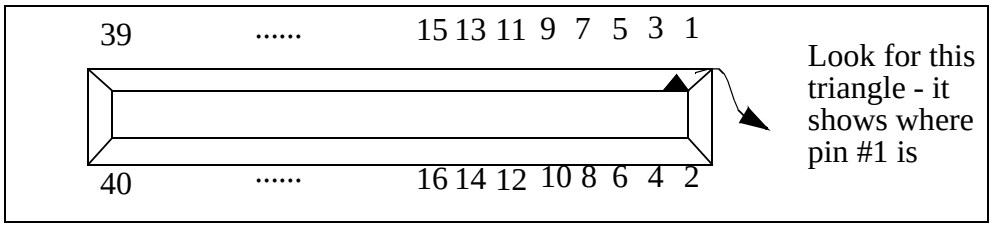


Figure 21 - *40-Pin Header View from Top*

Be sure that the power to the Ultragizmo board is off and the power to the protoboard is off.

2. Connect your input signals from the digital board switches to the correct input pins for your circuit on the protoboard header - i.e to those pins that correspond to the pins you selected in the wrapper file.
3. Turn on the power to the Ultragizmo board and protoboard.
4. Download and test your circuit, using the instructions from Chapter 3, and demonstrate it to a TA.

2 Background

A seven-segment display is often used on computers, watches, VCRs and many electronic devices to display numbers and some characters. It consists of seven independent lights in an “8” configuration as shown below in Figure 22. By turning on different segments, you can display different numbers and some letters.

Preparation

You are to create two logic circuits to drive one of the four 7-segment displays on the Ultragizmo board. Please see below for details on how to use the seven-segment displays. In particular, note that to turn a segment on, you must drive the corresponding pin to a logical “0”.

1. Design a circuit that takes a four bit (X_3, X_2, X_1, X_0) input from the digital switch board and drives digit #0 on the Ultragizmo board as described in the table below. Note that for the letters, some are capitalized and some are not. (The reason is that a capital B, for example, would come out the same as an 8 on a 7-segment display, so we will display a lower case b instead).

$X_3\ X_2\ X_1 X_0$	**Display (note the capitalization)**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	b
1100	C
1101	d
1110	E
1111	F

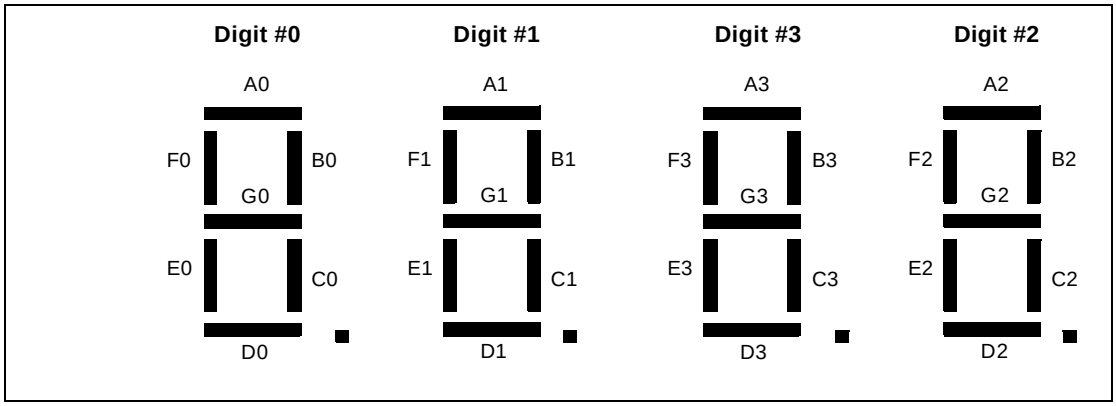
Table 1 - *Hex Display Truth Table*

2. Determine the equations for the 7-segment display segments, and minimize them using the Karnaugh-map method. Write VHDL code to represent the logic function for each segment as a Boolean equation. Simulate and test your equations using the Max+plusII functional or timing simulation on the SFPGA.

In the Lab: Download and test your circuits from the preparation. Show each working circuit to the TA.

Connections Between the 10K70 and the Seven-Segment Displays

The Ultragizmo board has four seven segment displays attached directly to the pins of the chip. Figure 22 shows the naming of each segment in the wrapper file. NOTE: to turn a segment on, you must drive the associated pin with a “0.” (not a 1). Also, note the numbering of the hex digits.



HOUTx(6)	Ax
HOUTx(5)	Bx
HOUTx(4)	Cx
HOUTx(3)	Dx
HOUTx(2)	Ex
HOUTx(1)	Fx
HOUTx(0)	Gx

Figure 22 - *7-Segment Display Organization on the Ultragizmo Board*

Lab F2 Hierarchical Design

The purpose of this lab is to create a design using hierarchical design techniques. The use of flip-flops in VHDL will also be explored.

Preparation

The purpose of this lab is to generate a sequence of Fibonacci numbers. Recall that the Fibonacci numbers are defined by $x_i = x_{i-1} + x_{i-2}$, with $x_0 = 1$ and $x_1 = 1$, so that the sequence of numbers begins 1,1,2,3,5,8,13,21,... In this lab, you are to construct a circuit that produces the Fibonacci numbers and displays them on the 7-segment displays. Note that since the display uses hexadecimal, the sequence of values that you show should proceed as 0001, 0001, 0002, 0003, 0005, 0008, 000D, 0015,... A block diagram of one method for producing this sequence is shown below. Each of the thick lines represents a bus of 16 wires. Two 16-bit registers are used to store x_{i-1} and x_{i-2} respectively, the latter one of which is also connected to the 7-segment displays. A 16-bit adder produces the sum of the two previous numbers, and at each clock edge, a new number is loaded into one of the registers. The reset signal sets both of the registers to 0000000000000001 binary.

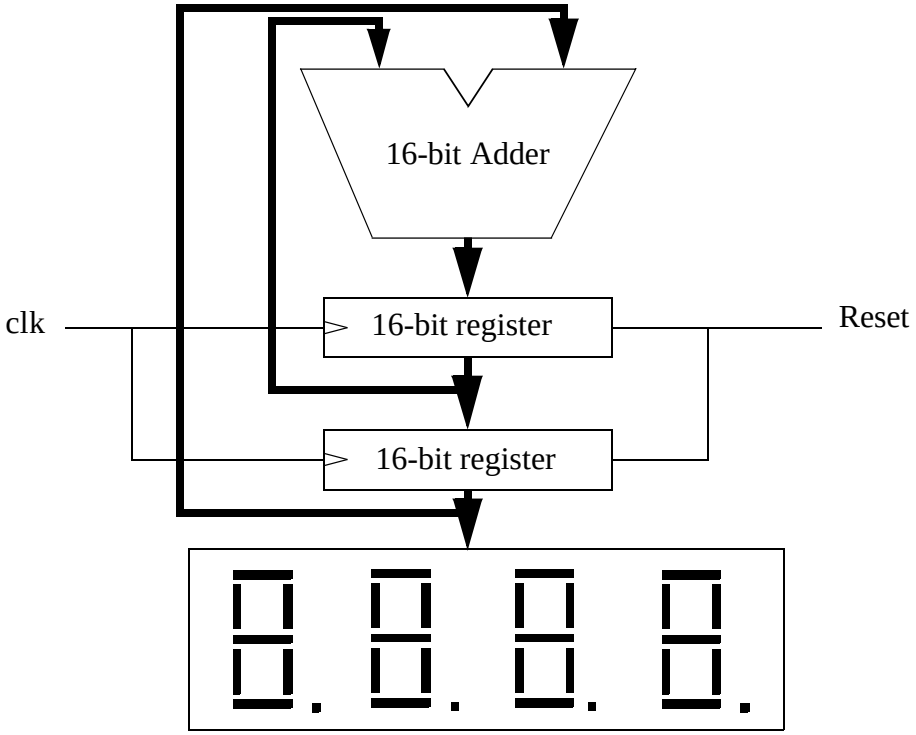


Figure 23 - *Block Diagram of Fibonacci Counter*

Design a single bit slice that contains a 1-bit adder, and two flip-flops. Use the Max+plusII DFF component for the flip-flops. Ideally, we would ike to cascade 16 identical copies of this slice to form the system, however, the reset state of the system makes this difficult. This is

because the least significant bit resets to 1, and all of the other bits reset to 0. One way to get around this difficulty is to design two slightly different versions of the 1-bit slice, one of which resets to 0, and one of which resets to 1. Both of these should use the 1-bit adder cells. Consequently, your VHDL code should have entities that correspond to the following diagram. The bit-0 slice and bit-1 slice differ only in the way that they respond to the reset signal. The Fibonacci generator has 1 instance of the bit-1 slice, and 15 instances of the bit-0 slice.

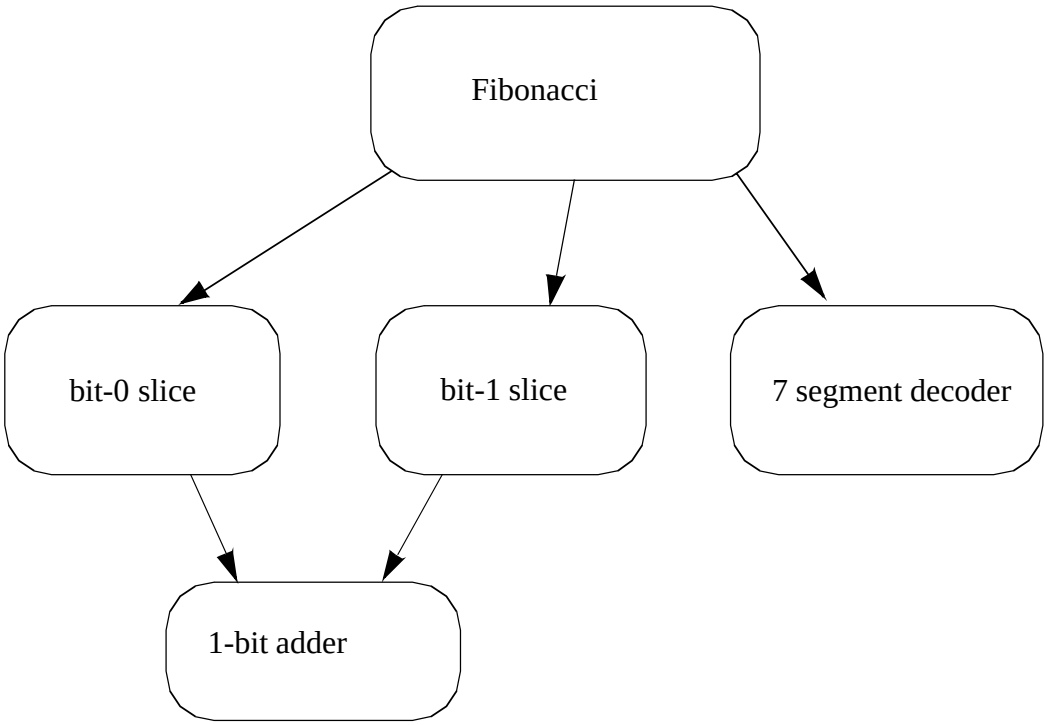


Figure 24 - *VHDL Entity Organization for Fibonacci Counter*

Preparation: Prepare and compile all of your VHDL code before the lab.

In the Lab: Implement and test the circuit that you designed above, and demonstrate it to a TA.

Lab F3 System Design Using State Machines and Arithmetic

The purpose of this lab is to perform a larger scale design using state machines and arithmetic circuits. The goal is to design a watch and stopwatch combination.

Description

The purpose of this lab is to construct a watch/stopwatch combination, but for brevity we will just call it the watch. It will have three switches to control it, and display the result, but operates as either a watch or a stopwatch.

The control buttons are called mode, select, and start/stop.

The watch has three modes: **watch** mode (W), **stopwatch** mode (S), and **set_watch** mode (SW). A 0 to 1 transition on the mode switch changes the mode of the watch from one to the next, in the following order W -> S -> SW -> W, etc.

In any mode, the watch keeps track of time in a MM:SS.HH format, where MM is the time in minutes, SS is the time in seconds, and HH is the time in hundredths of a second. Because we have only four digits that can be displayed at any one time, the select button is used to toggle between three different representations of the time. A 0 to 1 transition on select will change the display in the following order: MM:SS -> SS.HH -> HH,MM, etc.

The watch is always running. In watch mode, the display shows the time according to the digits selected.

In **stopwatch** mode, the display shows the amount of time that the stopwatch has run. The start/stop button controls the stopwatch. A 0 to 1 transition on start/stop will start the watch if it is stopped, and conversely, stop the watch if it is already started. However, a 1 on the start/stop switch for 2 seconds or more will set the stopwatch back to 00:00.00, and set the stopwatch mode back to stopped.

In **set_watch** mode, the current value of the watch will be displayed, and the watch will stop counting. The right time display will flash on and off at 1 Hz rate (i.e., on for .5 sec, followed by off for .5 sec.) In this mode, when the start/stop switch is 1, the value being displayed will increment at 2 counts per second. A roll-over of the display (eg. from 59 back to 0) will not affect the digits to the left, some of which might not visible at that time.

Preparation

Design your watch in VHDL. In a large design, it is usually valuable to test pieces of the design incrementally, as they are completed. Simulate each of the components, using several different designs that include progressively larger pieces of the watch.

The watch contains a large number of arithmetic components as well as state machines. You should use VHDL that expresses arithmetic operations for the arithmetic. Use one process for each distinct piece of combinational logic, and one process for each state register.

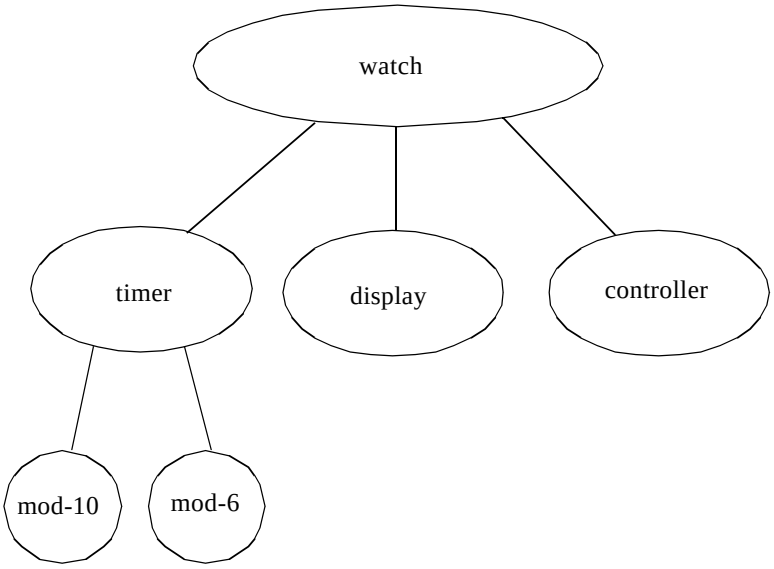


Figure 25 - *VHDL Entity Organization for Watch*

A possible way to organize the design is shown above in Figure 25. The watch contains a controller, two timers, and a display. The timer is made up of mod-10 and mod-6 counters. The controller monitors the buttons, and tells the display what to show, and whether to flash or not. It also controls the counting of each of the timers.

You may choose to use a package to define a type that represents the state of the display, and can be shared between the watch, timer, and controller. If you decide to do so, make sure you use the *interfaces->vhdl reader* menu to set the VHDL version to 1993.

Predefined Modules

The watch is a fairly complicated design. In order to simplify your work, a number of predefined design entities will be made available in the */cad2/ultragizmo/FLabs/F3* directory on the *ugsparc* network. These are as follows:

Modulo Counters

Modulo-6 and modulo-10 counters will be provided in *count6.vhd* and *count10.vhd*. Their component declarations are below. The clk signal is the clock, and both an asynchronous and synchronous reset are provided. Only use the asynchronous reset for master reset of the entire chip. Each counter provides an enable signal to enable counting, and output value, and a carry_out that is asserted when the counter is at its maximum count.

```
component count6 is
port (
 clk: in std_logic;
```

```
  sync_reset: in std_logic;
  async_reset: in std_logic;
  enable: in std_logic;
  count: buffer std_logic_vector (3 downto 0);
  carry_out: out std_logic);
end component;

component count10 is
port (
  clk: in std_logic;
  sync_reset: in std_logic;
  async_reset: in std_logic;
  enable: in std_logic;
  count: buffer std_logic_vector (3 downto 0);
  carry_out: out std_logic);
end component;
```

Timer

A skeleton of a timer will be provided. This includes the interface, and the declaration of control signals and instantiation of the 6 counters needed for a MM:SS.HH timer. You must complete the control logic for the count enable signals and output selection. The output selection is defined using a signal of type **display_select**. The interface to a timer is:

```
entity timer is
port (
  clk: in std_logic;
  sync_reset: in std_logic;
  async_reset: in std_logic;
  run_enable: in std_logic;
  increment_display: in std_logic;
  display_which: in display_select;
  display_3: out std_logic_vector (3 downto 0);
  display_2: out std_logic_vector (3 downto 0);
  display_1: out std_logic_vector (3 downto 0);
  display_0: out std_logic_vector (3 downto 0));
end timer;
```

Timer Output Select

A predefined type called **display_select** is available to communicate the desired digits between the watch and the timers. This is defined as follows in *watch_types.vhd*:

```
package watch_types is
  type display_select is (sel_seconds, sel_minutes);
end watch_types;
```

This is probably the cleanest way to define the structure of the information passed between the controller and the timers. Unfortunately, there is a bug in Max+plusII which prevents it from

working correctly in the present version. You should use *watch_types_const.vhd* instead, which explicitly assigns logic values to the **display_select** type.

Seven-segment Decoder

A 7-segment decoder with enable is available in *seg7.vhd*. The enable signal must be a 1 for the display to be enabled, otherwise all of the LEDs will be off.

Clock Generator and Watch Main Module

A skeleton of the watch is provided in */cad2/ultragizmo/FLabs/F3.* This includes the definition of a 100Hz clock, a 1Hz clock, and a timing signal that is enabled for one 100Hz cycle, twice per second.

The definition of these uses the M68000's 16.67MHz internal clock. This clock is divided down to a 100Hz clock. The clock divider uses the ratio 16.67MHz/100Hz/4 = 41675 as part of the clock division chain. To make simulations run faster, the clock division ratio is initially set to 3. You should simulate your watch using the division ratio of 3, and change the number to 41675 when you are ready to compile for execution on the Ultragizmo board.

The watch main module also provides a timer for the start/stop button to detect when it has passed 2 seconds. Initially, the timeout value is set to .5 second for faster simulation. You should change this to 2 seconds when you want to compile for the Ultragizmo board.

In the lab:

- Week 1: Design, test, and demonstrate a single timer, capable of toggling between the display states. Use of the Ultragizmo's clock chip is not necessary at this point, but would be very useful!

- Week 2: Complete the watch, and demonstrate it to a TA. For complete marks your watch must have all the capabilities described above.

Lab F4 System Design and Memory Arrays

The purpose of this lab is to investigate the use of memory on the SFPGA. This will be done by modifying the design from Lab F3. As part of this, the issue of design modularity and parameterization of a design will be discussed. The specific goal used to illustrate these concepts is the modification of your watch design from Lab F3 to remember up to 8 times that are timed by the stopwatch.

Description

The goal of this lab is to allow the user to save and examine up to 8 times that have been determined by using the stopwatch. However, the watch will not require this feature.

Recall that the control buttons are called **mode**, **select**, and **start/stop**.The watch will be extended to have four modes: **watch** mode (W), **stopwatch** mode (S), **stopwatch memory** mode (M), and **set_watch** mode (SW). A 0 to 1 transition on the mode switch changes the mode of the watch from one to the next, in the following order W -> S -> M -> SW -> W, etc. The watch has 8 memory cells that can each store a time.

As before, in all modes, the watch keeps track of time in a MM:SS.HH format, where MM is the time in minutes, SS is the time in seconds, and HH is the time in hundredths of a second, and the select button is used to toggle between three different representations of the time.The operation of the W, S, and SW modes is the same as in Lab F3.

The watch is augmented to have 8 memory cells, and a memory cell stores a time in the MM:SS.HH format. In M mode, the display shows the value recorded by the current memory cell address. The current memory cell address is a number from 0 to 7, and is adjusted by use of the buttons, as described as below. A 0 to 1 transition on the start/stop button increments the memory cell address that is being displayed by 1. The memory cell address will roll over from 7 to 0. If the start/stop button is held down for 2 seconds, then the value currently recorded in the stopwatch is stored in the current memory cell. The select button works as usual in M mode.

Use the Altera EABs to implement the memory cells. Each EAB can store up to 256 words of 8 bits, so the simplest method is to use 3 EABs, each storing 8 bits of the time. You will only need to use 8 words of each EAB. Be careful in your VHDL code in configuring the input and output registers and clocking of the EAB.

The design issue of interest here is how to perform a modular design, while specializing it to include the memory feature only for the stopwatch and not the watch. The advantage of a modular design is that you can define a module once, then create multiple identical copies of it. Since it is necessary to have several slightly different versions of the module, the design needs to be structured carefully to maximize modularity. For this problem, there are two approaches that can be taken.

Parameterization of a Module

One approach is to define a general module that includes all of the features that you could want, but by including certain parameters, can be specialized to only contain a subset of the particular features in each distinct copy. To do this, you would redefine the timer module to contain a memory controller, which contains the memory array and control circuit. Since the watch does not require the memory controller, an additional parameter would be required to specify whether or not to include the memory controller. VHDL includes the **generic** feature and **generate** statements to let you control whether or not the memory controller is instantiated. Unfortunately, Max+plusII does not support the necessary features in VHDL, so you can't do this!

An alternative which is acceptable in this lab, because we have the extra hardware, is to implement the memory controller for both the watch and the stopwatch, by including it in the timer module unconditionally. This is a waste of hardware, but we have some extra EABs so this is reasonable for this lab. However, you should be able to show which parts of VHDL code for the timer should only be instantiated for the stopwatch timer.

Finer Grained Modularity

Another approach is to break up the design into smaller modules. You can expand the interface of the timer so that the time in MM:SS.HH is also output, and define a memory entity that has an input and output in MM:SS.HH format. You can then connect the memory entity up to the stopwatch instance of the timer. Because both the timer and the memory need to select between MM, SS, and HH, you should also split off the output selection logic into a separate entity. Finally, you will need to include more complex display selection logic in the watch. This approach requires a finer degree of control over the design entities.

Either of these two approaches is acceptable, but the parameterized modules are somewhat more modular and clean. This approach makes it possible to create any number of timers, with or without associated memory.

Preparation: Prepare your VHDL, compile and simulate it before the lab.

In the Lab: Test your design and demonstrate it to a TA.

Lab F5 Data Transfer on the M68000

The objective of this lab is to study data transfer on the bus of the M68000 microprocessor, and to become familiar with design of memory interfaces. In the first part of the lab you will use the SFPGA to implement a simple register that can be accessed via M68000 assembly instructions. In the second part of the lab, you will implement an SRAM controller using the SFPGA and the Ultragizmo's SRAM memory. In the third part of the lab, you will implement a DRAM controller using the SFPGA and a standard DRAM chip. You will test your design using M68000 assembly language. This lab will likely take two weeks to complete.

Data Transfers on the M68000

The M68000 bus is asynchronous and uses a handshake to transfer data. The following description refers to the processor as the device issuing read and write data transfer requests. The signals involved in data transfer are:

A_{23-1} Address lines. The processor transmits the word address of the data to be transferred on these lines. When only one byte of that word is to be transferred, the processor uses the Data Strobe signals (see below) to indicate whether the low or high order byte is involved. There is no bus line for the least-significant address bit A_0.

$\overline{AS}$ Address Strobe. Asserted by the processor to indicate that the address lines carry a valid address.

D_{15-0} Data lines. These carry the data being transferred. Driven by the addressed device during read operations and by the processor during write operations.

$\overline{LDS}$ Lower data strobe. Asserted by the processor when the low-order byte of a word is to be transferred.

$\overline{UDS}$ Upper data strobe. Asserted by the processor when the high-order byte of a word is being transferred. During a word transfer, the processor asserts both $\overline{LDS}$ and $\overline{UDS}$.

$\overline{DTACK}$ Data acknowledge. Asserted by the device being addressed after it has completed the requested operation.

R/$\overline{W}$ Used by the processor to indicate whether it is requesting a read or a write operation.

The address strobe is the timing signal used in conjunction with the address lines. The $\overline{LDS}$ and $\overline{UDS}$ signals perform the function of the request signal in the handshaking protocol.

A read operation proceeds as shown in Figure 26. The processor transmits the address and then asserts $\overline{AS}$, after an address setup time t_{as} of at least 20 ns. It asserts $\overline{LDS}$ and $\overline{UDS}$ at the same time to request data from the addressed device. The device responds by placing the data on the bus and asserting $\overline{DTACK}$. In doing so, it must guarantee that the data is valid on the data lines no later than 65 ns after $\overline{DTACK}$ is asserted.

It may appear odd that the device may send the data after it asserts $\overline{DTACK}$. The reason this is allowed is that the processor waits for one clock cycle (100 ns) after receiving $\overline{DTACK}$ before it strobes the data into its input buffer. Remember that while the bus signals are asynchronous, the

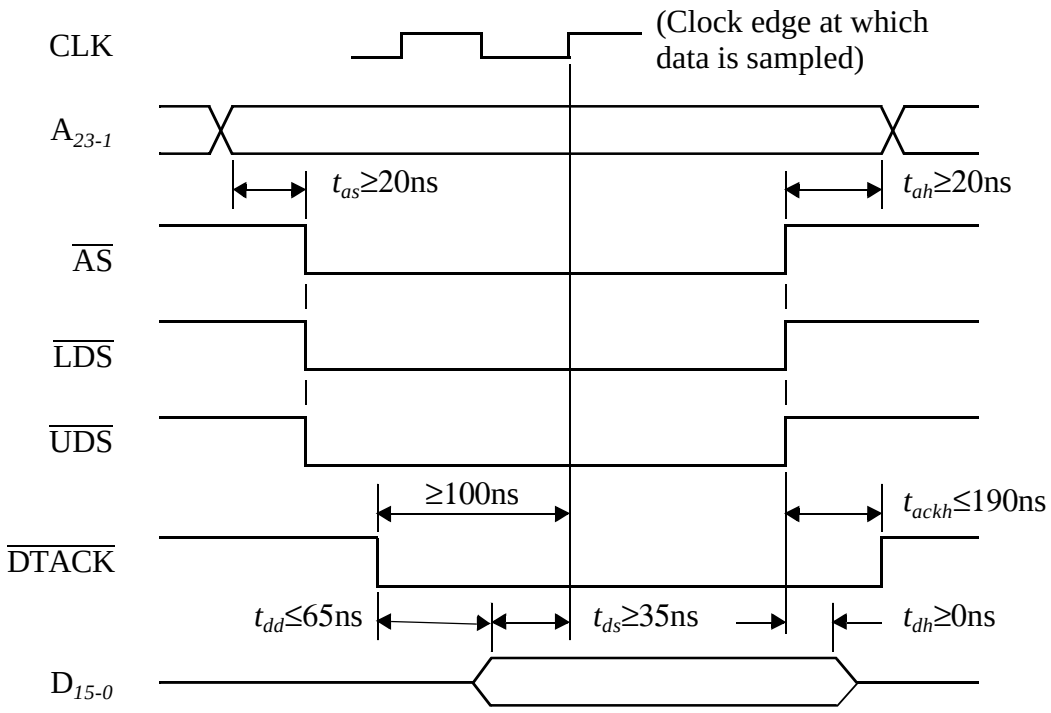


Figure 26 - *Timing Diagram for a Read Operation on the M68000 Bus*

processor itself is a synchronous machine. It will detect the asserted state on $\overline{\text{DTACK}}$ at one clock edge, then strobe the data at the next clock edge. Thus, the condition $t_{dd} \leq 65$ ns means that the actual setup time for the input buffer of the processor, t_{ds}, is greater than 35 ns.

After strobing the data, the processor negates $\overline{\text{AS}}$, $\overline{\text{LDS}}$, and $\overline{\text{UDS}}$ to end the transfer cycle. The device responds by removing the data and negating $\overline{\text{DTACK}}$.

1 Implementing a Register

The Ultragizmo board's address space from $a00000 to $bfffff is not used. In this lab, you are required to add a 16-bit register in this available range. To reduce the amount of wiring, you are allowed to use aliasing. That is, instead of locating your register at a specific location, you can allow any address in the range $a00000 to $bfffff to access your register. This means you only need to decode the upper address bits and can leave the lower bits unconnected.

The register should reset to 0, and the current contents of the register should always be shown on the 7-segment displays.

- Design the circuitry needed to interface the SFPGA to the M68000 bus with timing as shown in Figure 26. Make sure that your design handles both byte and word operations correctly. Pay special attention to the timing of $\overline{\text{DTACK}}$, and remember that it is an open-collector signal.

- Write an assembly language program to test your circuit. The program should read the register, add 1 to the value, and then write the sum back to the register. You will need to add a lot of NOPs to your code to make the result visible on the 7-segment display.

2 SRAM Controller

The objective of this section is to design an SRAM controller using the SFPGA on the M68000 bus. In the lab a controller and datapath are created to provide the address and control signals required by the SRAM memory located on the Ultragizmo board.

The purpose of this lab is to implement a 1M byte memory using two 256K * 16 SRAMs. An overview of the system is shown below.

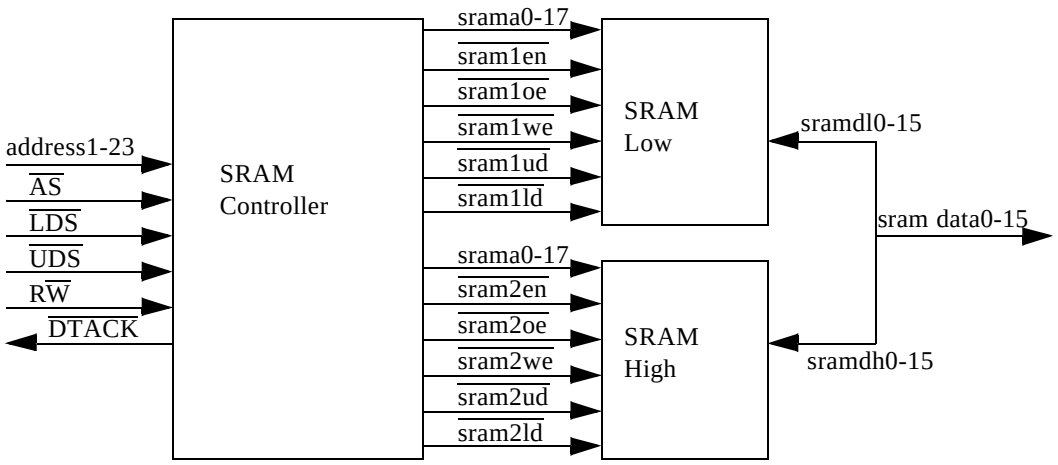


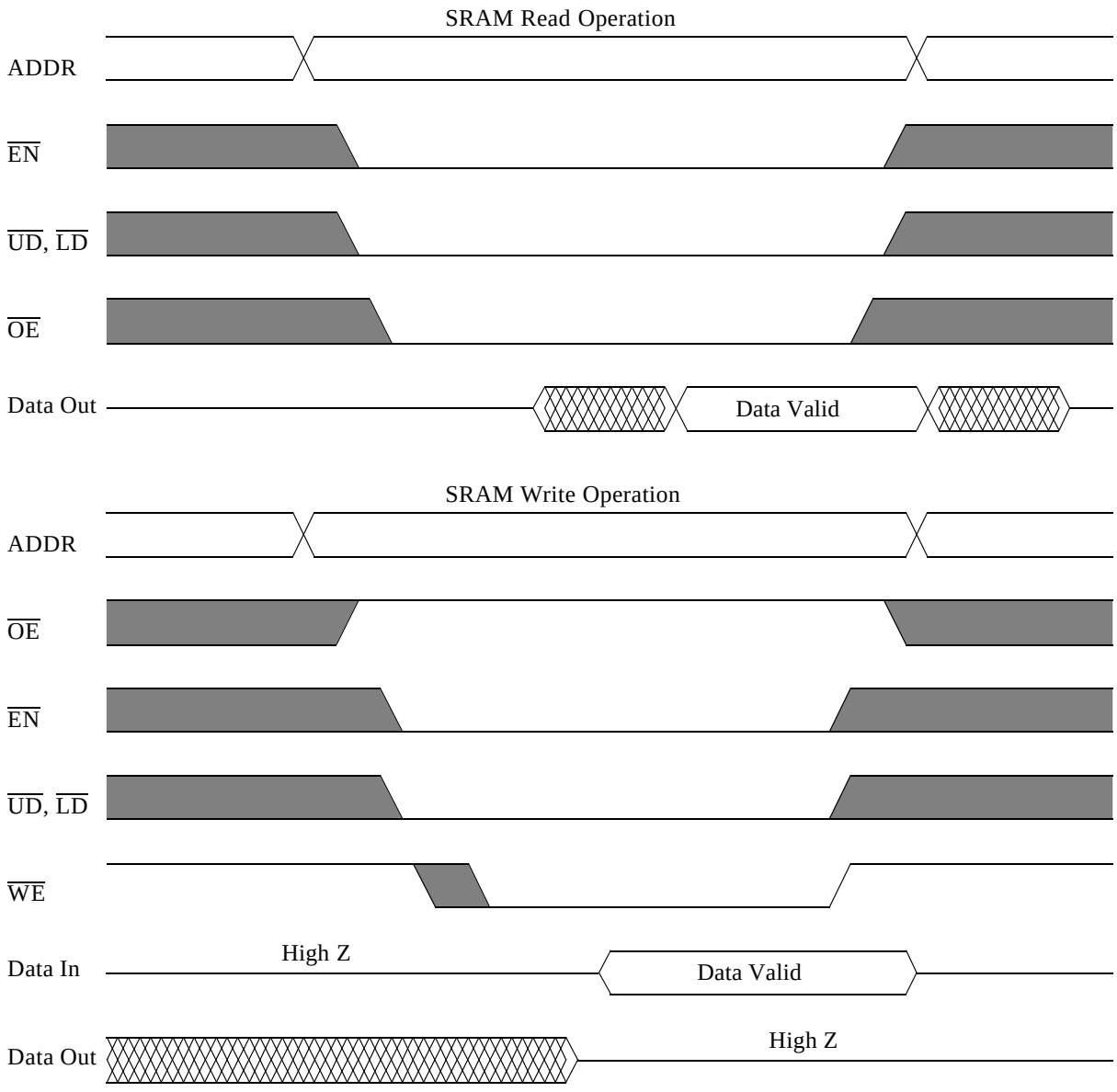
Figure 27 - *Diagram of SRAM and Controller Interconnections*

For the purposes of this lab we will ignore SRAM timing issues by assuming that each timing interval is implemented within a clock cycle.

An SRAM has 6 control signals, four of which control the addressing of the RAM, and two of which control the reading and writing of data. The SRAMs you will be using are 256K * 16, which means that 18 address bits are required to address a particular 16 bit word in the RAM. The SRAM is enabled when $\overline{EN}$ is asserted; when $\overline{LD}$ is asserted, the lower 8 data bits are enabled; similarly when $\overline{UD}$ is asserted, the upper 8 data bits are asserted.

The signals $\overline{OE}$ and $\overline{WE}$ control whether the cycle is a read or a write cycle. If $\overline{OE}$ is asserted, the cycle is a read and the data appears on the I/O pins shortly after the address pins become stable. If the cycle is a write, then $\overline{WE}$ should be asserted and the data is driven on the I/O pins for at least 8ns before the end of the $\overline{WE}$ assertion. Details of the two types of cycle appear in Figure 28.

Figure 28 - *SRAM Read and Write Timing Diagrams*



3 DRAM Controller

The objective of this lab is to investigate the design of a DRAM controller using the M68000 bus. This exercise represents the design of a complex controller and datapath needed to provide the address and control signals required by a DRAM memory.

The purpose of this lab is to implement a 256K byte memory using 256K * 4 DRAMs. Because a large number of data wires would be required to implement a word-wide memory, your memory only needs to implement the lower byte of 256K consecutive words. That is, for any word in the 256K word (512KB) region that your memory implements, a read or write to that word should access a word with the lower byte correct, but the upper byte may contain garbage. For example, if the CPU writes the value $1234 into location $a00000, and later reads $a00000, then the lower byte should contain $34, but the upper byte may contain any value. To implement

this you can use two 256K*4 DRAMs with their data lines connected to the lower 8 bits of the data bus D0-7. An overview of the system is shown below. A more detailed diagram is shown at the end of the lab handout.

The system contains an address decoder and DRAM controller which generates the timing signals $\overline{RAS}$, $\overline{CAS}$, $\overline{OE}$, and $\overline{WE}$ for the DRAMs. Because the DRAMs implement a 256K word memory, (although only the lower byte of each word) address lines A1-18 are used to address the memory. Other address lines may be used to decode the address. You should choose a 256K word region in the range $a00000 to $bfffff for your memory. You may have the memory appear at multiple locations if this simplifies your hardware.

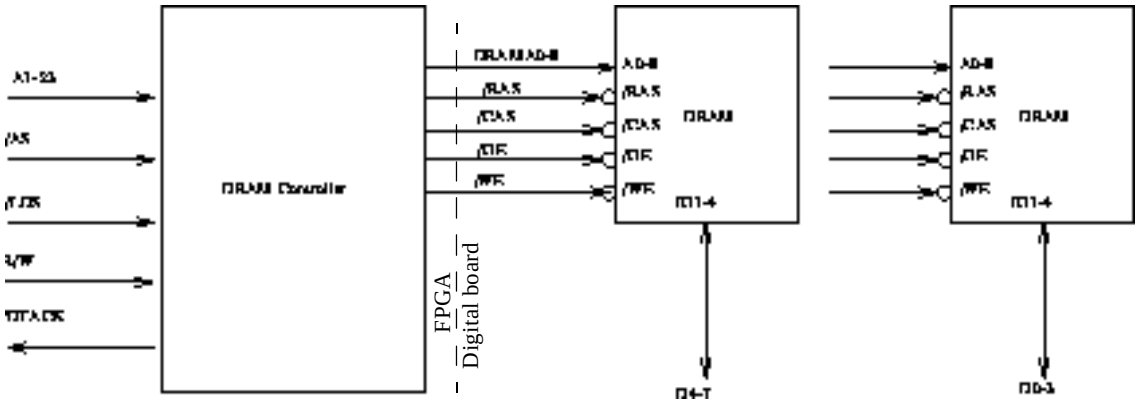


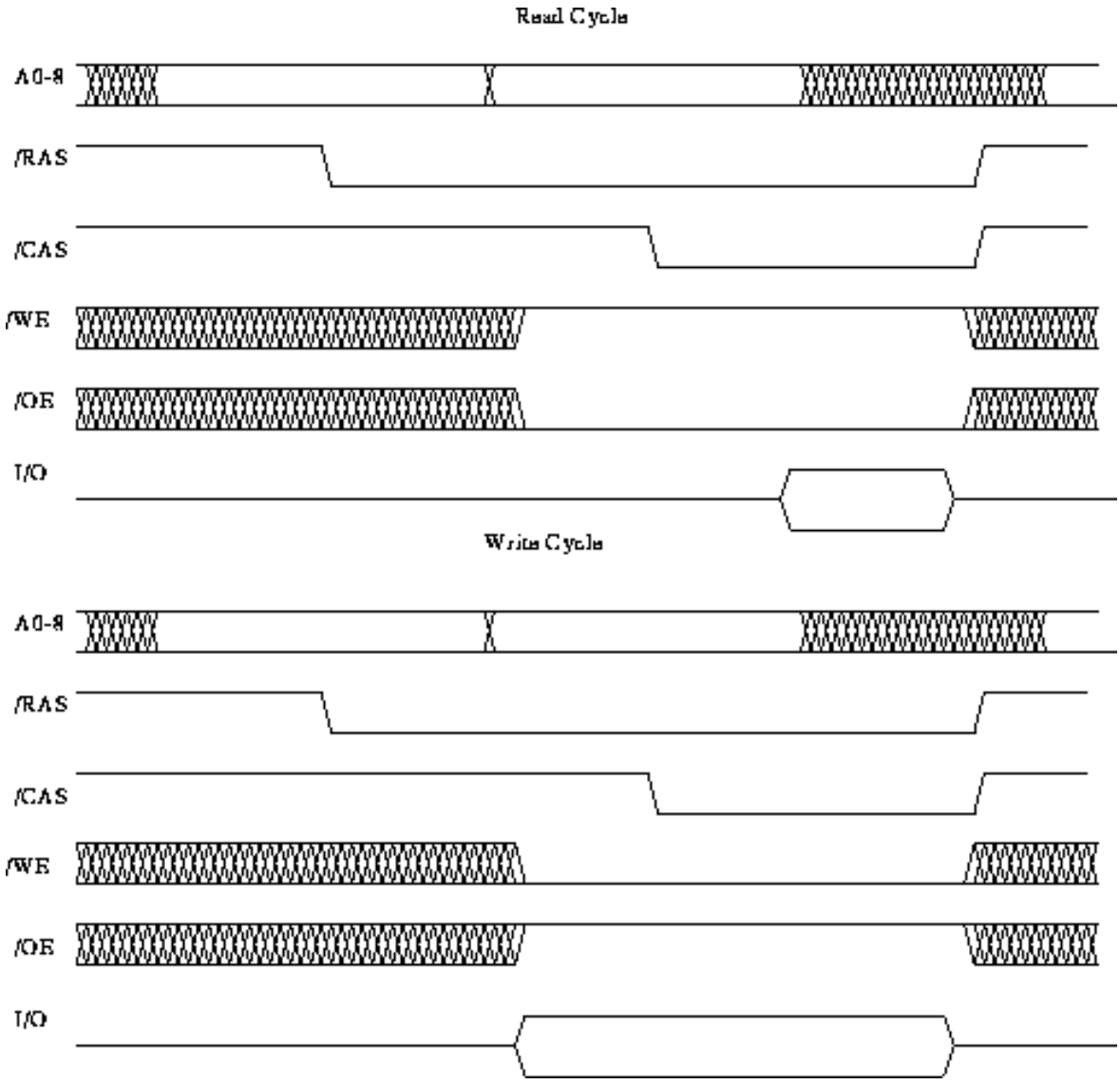
Figure 29 - *Diagram of DRAM and Controller Interconnections*

The detailed timing of a DRAM is a complex specification with many subtle issues. For the purposes of this lab we will ignore all of them by assuming that each timing interval is implemented with two clock cycle. This is much slower than is actually required, since DRAMs we use are capable of accessing in 80ns. However, this assumption greatly simplifies the design of the hardware.

A DRAM has 4 major control signals, two of which control the addressing of the RAM, and two of which control the reading and writing of data. The first two signals are $\overline{RAS}$ and $\overline{CAS}$, which stand for Row Address Strobe and Column Address Strobe respectively. The DRAMs you will be using are 256K * 4, which means that 18 address bits are required to address the particular 4 bit word in the RAM. DRAMs avoid the need for 18 address pins on each chip by multiplexing the address bits over 9 pins. The steps involved in addressing the DRAM for read and write cycles are shown in Figure 30. In each case, half of the address bits are driven on the address lines, $\overline{RAS}$ is asserted, the second half of the address bits are driven on the address lines, and $\overline{CAS}$ is asserted.

The signals $\overline{OE}$ and $\overline{WE}$ control whether the cycle is a read or write cycle. If $\overline{OE}$ is asserted, the cycle is a read and the data appear on the I/O pins shortly after $\overline{CAS}$ is asserted. If the cycle is a write, then $\overline{WE}$ should be asserted and the data driven on the I/O pins before $\overline{CAS}$ is asserted. See Figure 30 for details of the two types of cycles.

Figure 30 - *DRAM Read and Write Timing Diagrams*



Because DRAMs are dynamic, they will forget the information stored unless each cell is periodically read. Each access to the DRAM reads an entire row of 512*4 bits, and restores the signal levels in that row of cells. Each row must be read at least every 8ms. There are a total of 512 such rows. The easiest way to do this is to read one of the rows every 8ms/512=15.6 microseconds. However, the DRAM provides some help in doing this. The DRAM has an internal counter that is consecutively incremented when a CAS-before-RAS refresh cycle is performed (see Figure 31). A refresh cycle reads the data internal to the DRAM, but does not drive it on any of the I/O pins. A CAS-before-RAS refresh cycle is performed by asserting $\overline{\text{CAS}}$, waiting a while, and then

asserting $\overline{\text{RAS}}$. The address pins may be driven with any value, since the DRAMs internal counter supplies the address for this type of cycle.

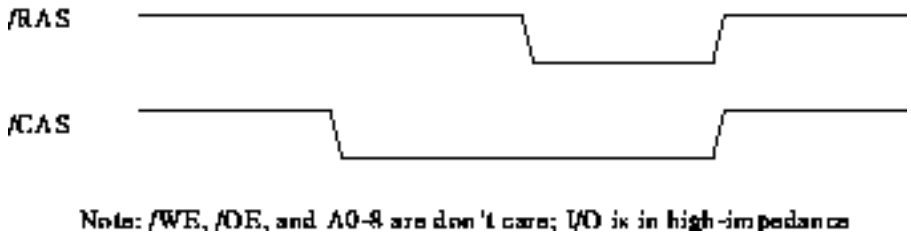


Figure 31 - *DRAM CAS-before-RAS Refresh*

This establishes the need for a timer that periodically (every 15 microseconds or less) determines when it is necessary to perform a refresh. Because of the need for servicing requests for both refresh and read/write accesses from the M68000, the controller must be designed to accept requests from two sources. The controller should examine requests for refresh, as well as the bus of the M68000. When a request is seen, the controller should perform the requested type of cycle. The controller design must be done carefully, so that if a refresh and a M68000 request happen simultaneously, one cycle is performed, then the other. It is suggested that the refresh be done first, to avoid the possibility of indefinitely postponing the refresh (if the M68000 continually accesses the DRAM).

A suggested hardware structure is shown in Figure 32. There are three main components. A set of nine 2-to-1 multiplexers is used to multiplex the M68000 address bits A1-18 onto the DRAMs 9 address bits DRAMA0-8. A refresh counter is used to generate a request for refresh every 8 microseconds. 8 microseconds is 128 clock cycles using the M68000 clock, so is a convenient number for this purpose. Note that the refresh timer must assert the refresh request until a refresh cycle is performed, because the DRAM may be in the middle of doing a read or write cycle when the refresh request occurs. The easiest way to detect that the refresh has begun is to detect that $\overline{\text{CAS}}$ is asserted and $\overline{\text{RAS}}$ is negated.

For testing purposes you will need to obtain a DRAM chip from your TA. You can place your chip on the digital protoboard and wire it to the SFPGA using a 40-pin connector connected to the **SFPGA_DIGITAL** port.

In the lab:

- Week 1: Design, test, and demonstrate parts 1 and 2 (the register and SRAM controller).
- Week 2: Design, test, and demonstrate part 3 (the DRAM controller).

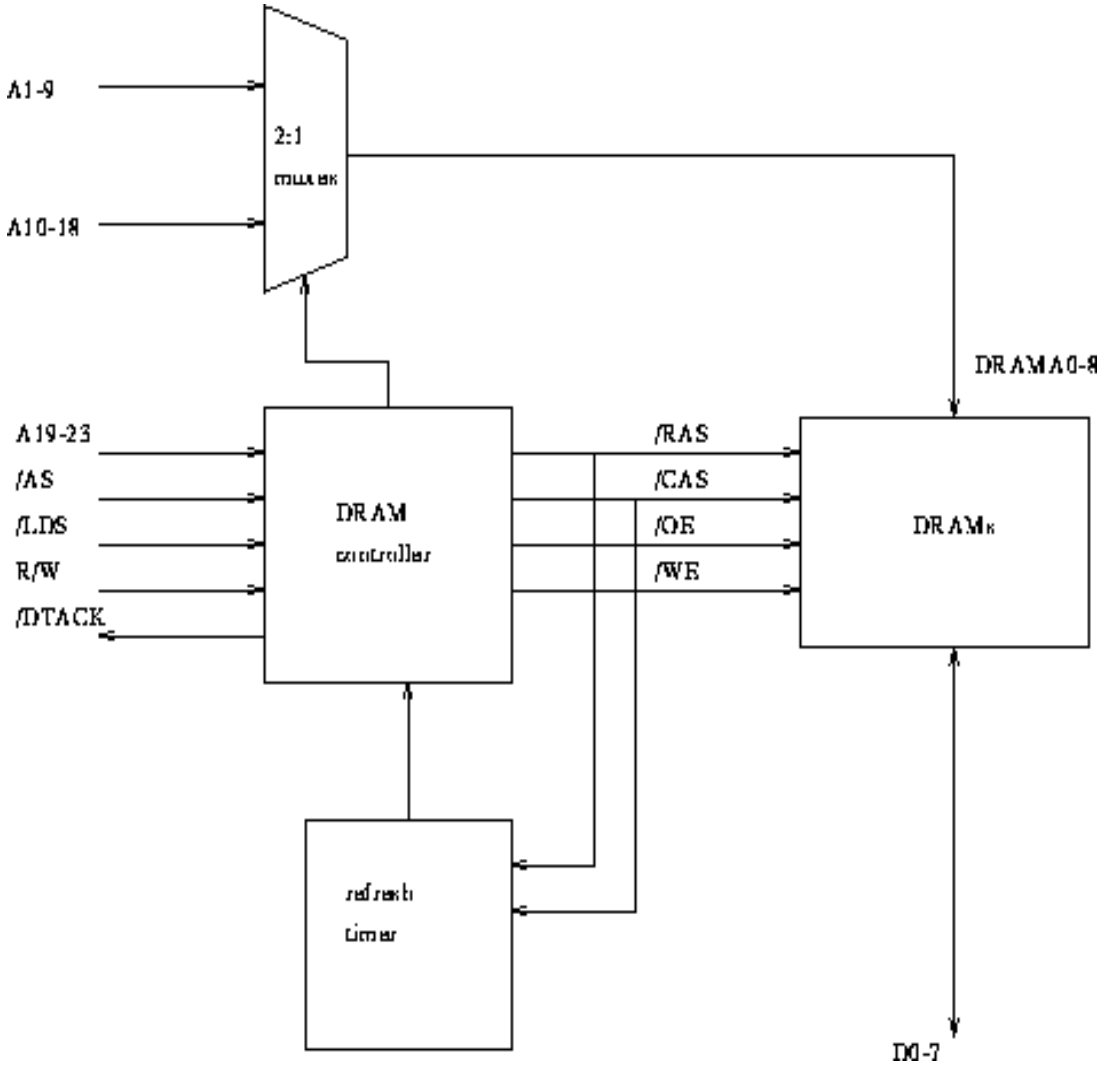


Figure 32 - *DRAM Controller Block Diagram*

Lab F6 Bus Arbitration on the M68000

The objective of this lab is to study bus arbitration on the bus of the M68000 microprocessor, and to become familiar with the design of DMA engines. In this lab you will use the SFPGA to implement a simple DMA engine that can be controlled via M68000 assembly instructions. This lab might take two weeks to complete.

Direct Memory Access

The figure below shows the M68000, its bus, the memory, and the SFPGA. In this lab you will write a DMA engine to copy data from the memory to the 10K70 and vice-versa.

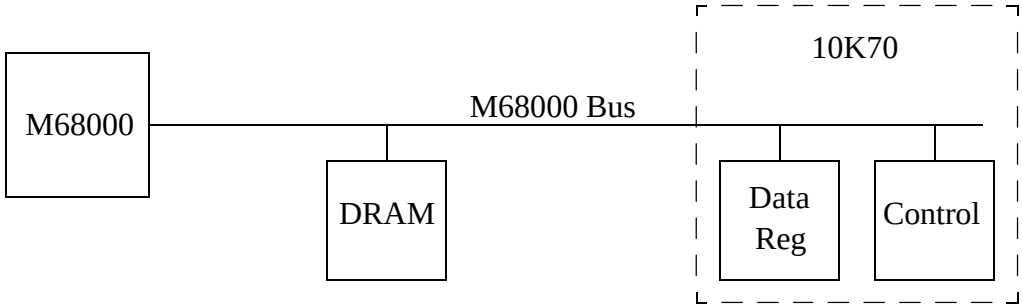


Figure 33 - *Block Diagram of M68000 Bus*

By using the M68000 to write to registers in your circuit, you will cause your circuit to gain control of the M68000 bus and perform read and write data transfers. The control registers in your circuit will allow you to specify the target address and amount of data to be transferred, and to specify the data being written or examine the data being read.

Bus Arbitration on the M68000

Bus arbitration on the M68000 is performed using three active low signals; **Bus Request** ($\overline{BR}$), **Bus Grant** ($\overline{BG}$), and **Bus Grant Acknowledge** ($\overline{BGACK}$). The waveform for bus arbitration is shown below in Figure 34.

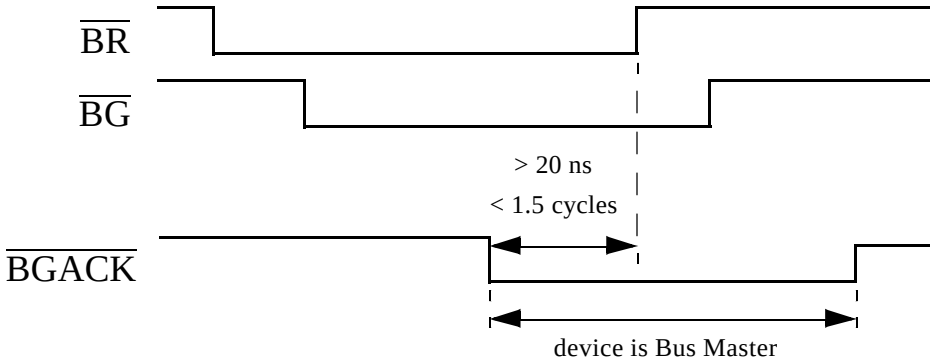


Figure 34 - *Timing Diagram for Bus Arbitration on the M68000 Bus*

The device that wants to become master asserts $\overline{BR}$. The CPU then asserts $\overline{BG}$, which is typically connected in a daisy chain between devices. In this lab, your circuit is the only potential bus master (other than the CPU), so you never need to pass the grant.

The device can begin driving bus signals as soon as it receives the grant and the bus is free. The bus is free when $\overline{AS}$, $\overline{UDS}$, $\overline{LDS}$, $\overline{DTACK}$, and $\overline{BGACK}$ are all unasserted. At this point the device becomes Master by asserting $\overline{BGACK}$ and then releasing $\overline{BR}$. $\overline{BGACK}$ must be asserted at least 20 ns before $\overline{BR}$ to ensure that all devices recognize that the bus is busy. $\overline{BR}$ must be released within 1.5 clock cycles after the assertion of $\overline{BGACK}$, or the device may receive a second grant (sometimes this is desirable). The CPU will release $\overline{BG}$ after sampling $\overline{BGACK}$ asserted. The device must keep $\overline{BGACK}$ asserted until it is finished driving the bus.

Control Registers

This lab builds on the register interface you designed in Lab F5. In order to control your DMA engine, you need to implement 3 control registers that can be accessed by the CPU. You may place your registers anywhere in the free memory space ($a00000 to $bfffff) and may alias them to multiple memory locations in order to reduce the amount of wiring.

- DMA Address Register (8 bits, suggested location $a00000):

All DMA data transfers should end at memory location $8XX00 in main memory, where XX is taken from this 8-bit register. Note that the 8 LSB's of the address are 0. This is done to ensure that all transfers are aligned to a word address, and that the count can be used directly as part of the address. This should simplify your design.

- DMA Data Register File (16 bits x 8 locations, suggested starting at $b00000):

This register file contains the data transferred by the DMA engine. If a DMA write is to be performed, these values will be written into the main memory. If a DMA read is performed, these are the values read from main memory.

- DMA Control Register (8 bits, suggested location $a00002):

The 3 LSB's contain the count of the number of words that are to be transferred. To simplify your design, you can count down from this value to 0. The value of the counter can be used directly for some of the address lines, and the transfers should stop when the count reaches 0.

The 2 MSB's are used for read and write controls. Writing a 1 to the MSB of this register should cause your DMA engine to perform a write transfer of <*count*> words with a data value of (data register) ending at address $8XX00. Writing a 1 to the 2nd MSB of this register should cause your DMA engine to perform a read transfer of <*count*> words ending at address $8XX00.

Implementation

In addition to the logic required to access and implement the control registers, you will need a circuit to request the bus and follow the bus arbitration protocol for the M68000. In addition, you

will need to implement a bus master state machine to perform the data transfers to and from main memory.

If it simplifies your circuit, you may make the Address register and Control register write only. If a read is performed to a write only register, $\overline{\text{DTACK}}$ must be asserted normally, but garbage data may be returned.

Figure 35 on the next page shows one possible top level design for your circuit.

The Address Register Logic Block implements the register to hold the XX part of the DMA address. It also combines the count with $8XX00 and drives this address on the bus when enabled by the DMA engine.

The Register Access Control Block handles reads and writes to the 3 control registers. The logic in this block should be very similar to Lab F5.

The Data Register Logic Block implements the register file that holds either the data to be written for a DMA write, or the data that was read for a DMA read. It must be able to store or drive the data on the bus as instructed by either the Register Access Control Block or the DMA Engine.

The Control Register Logic Block implements the counter and the DMA read or write control logic. When the counter value is non-zero, this block asserts either read or write to instruct the DMA Engine to perform a data transfer. As each data transfer completes, the DMA Engine will decrement the count by 1.

The DMA Engine has two functions. When instructed by the Control Register Logic, it requests the M68000 bus and obeys the bus arbitration protocol. Once it is granted the bus, it becomes the bus master and performs read or write transfers using the M68000 data transfer protocol. In the lab, demonstrate a DMA read and a DMA write to your TA.

Excerpts from the Motorola MC68306 Manual

The following is an excerpt from the Motorola MC68306 User's Manual (p. 3-4) concerning a bus read cycle:

"A bus cycle consists of eight states. The various signals are asserted during specific states of a read cycle as follows:

STATE 0 The read cycle starts in state 0 (S0). The processor places valid function codes on FC0-FC2, a valid address on the bus, and drives R/$\overline{\text{W}}$ high to identify a read cycle.

STATE 1 During state 1 (S1), no bus signals are altered.

STATE 2 On the rising edge of state 2 (S2), the processor asserts $\overline{\text{AS}}$ and $\overline{\text{UDS}}$/$\overline{\text{LDS}}$.

STATE 3 During state 3 (S3), no bus signals are altered.

STATE 4 During state 4 (S4), the processor waits for a cycle termination signal ($\overline{\text{DTACK}}$

or $\overline{\text{BERR}}$). If neither termination signal is asserted before the falling edge at the end of S4, the processor inserts wait states (full clock cycles) until either $\overline{\text{DTACK}}$ or $\overline{\text{BERR}}$ is asserted.

STATE 5 During state 5 (S5), no bus signals are altered.

STATE 6 Sometime between state 2 (S2) and state 6 (S6), data from the device is driven onto the data bus.

STATE 7 On the falling edge of the clock entering state 7 (S7), the processor latches data from the addressed device and negates $\overline{\text{AS}}$ and $\overline{\text{UDS}}$/$\overline{\text{LDS}}$. The device negates $\overline{\text{DTACK}}$ and $\overline{\text{BERR}}$ at this time."

NOTE: It is our experience that the $\overline{\text{DTACK}}$ negation in the read state 7 in a DMA is short and can go unnoticed, leading to a hung bus. Not checking for a negated $\overline{\text{DTACK}}$ can solve the problem.

The following is an excerpt from the Motorola MC68306 User's Manual (p. 3-7) concerning a bus write cycle:

"The descriptions of the eight states of a write cycle are as follows:

STATE 0 The write cycle starts in state 0 (S0). The processor places valid function codes on FC2-FC0, a valid address on the address bus, and drives R/$\overline{\text{W}}$ high (if a preceding write cycle has left R/$\overline{\text{W}}$ low).

STATE 1 During state 1 (S1), no bus signals are altered.

STATE 2 On the rising edge of state 2 (S2), the processor asserts $\overline{\text{AS}}$ and drives R/$\overline{\text{W}}$ low.

STATE 3 During state 3 (S3), the data bus is driven out of the high-impedance state as the data to be written is placed on the bus.

STATE 4 At the rising edge of S4, the processor asserts $\overline{\text{UDS}}$ and/or $\overline{\text{LDS}}$. The processor waits for a cycle termination signal ($\overline{\text{DTACK}}$ or $\overline{\text{BERR}}$). If neither termination signal is asserted before the falling edge at the end of S4, the processor inserts wait states (full clock cycles) until either $\overline{\text{DTACK}}$ or $\overline{\text{BERR}}$ is asserted.

STATE 5 During state 5 (S5), no bus signals are altered.

STATE 6 During state 6 (S6), no bus signals are altered.

STATE 7 On the falling edge of the clock entering state 7 (S7), the processor negates $\overline{\text{AS}}$, $\overline{\text{UDS}}$, and/or $\overline{\text{LDS}}$. As the clock rises at the end of S7, the processor places the data bus in the high-impedance state, and drives R/$\overline{\text{W}}$ high. The device negates $\overline{\text{DTACK}}$ and $\overline{\text{BERR}}$ at this time."

NOTE: It is our experience that the $\overline{\text{DTACK}}$ negation in the write state 7 in a DMA is short and can go unnoticed, leading to a hung bus. Not checking for a negated $\overline{\text{DTACK}}$ can solve the problem.

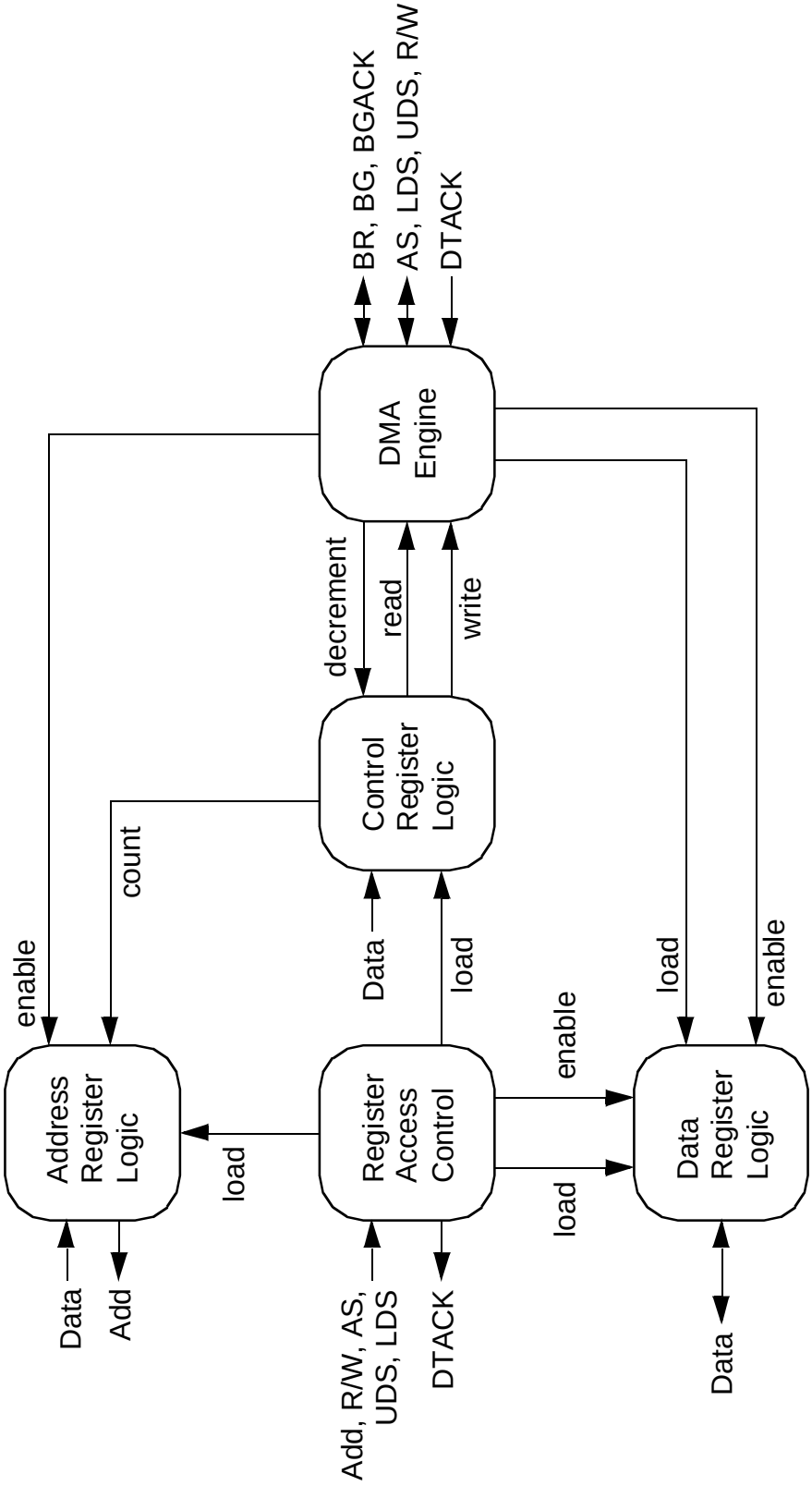


Figure 35 - *Block Diagram of DMA Controller*

Lab F7 (CODEC) Playing with Sound Using the Altera 10K70

The objective of this lab is to use the CODEC (COder-DECoder) to read and write sound, and to do manipulations on the sound. There are three parts to the lab. In the first you will take the input from the CODEC and output it directly, thus implementing a wire from input to output. In the second part, you will implement a programmable delay from input to output. In the last part you will record the input sound, manipulate it, and output it. A functional description of the CODEC is given in Section 8.6 on page 127.

1 Direct Input to Output

Given the timing requirements stated in section 8.6, implement a circuit which reads the input from the CODEC and writes the result back to the CODEC, in one cycle of the **ssync** pulse. For this lab you will need to implement a shift register as described in Section 8.6 on page 127.

2 Input to Output with Delay

In this section you will read values from the CODEC into a FIFO buffer, which will implement a delay. The length of the delay will be determined by the value of a 16-bit M68000 register, identical to the one designed in lab F5. The FIFO buffer should be implemented using the SRAM memory available on the Ultragizmo board. The SRAM contains 1 MB of memory. Since each read operation stores 4 Bytes, there is space for 256K values. At a sampling frequency of 48 kHz, this represents 5.5s of sound (i.e. a maximum delay of 5.5s if continuously sampling). You should continuously sample and output values. A block diagram of the circuit is shown in Figure 36.

3 Change in Pitch

In this part you will store 256K values in the SRAM, and output them afterwards (i.e. not concurrently) at a different rate than the sampling rate. This will have the effect of changing the pitch of the input sound. The output rate should be determined by the value stored in a 16-bit register accessed through the M68000 bus. Remember that the data are sampled at 48 kHz and that the output rate is also 48 kHz. To decrease the pitch you could for example output the same values for two consecutive **ssync** pulses. To increase the pitch you might send every second stored value to the output.

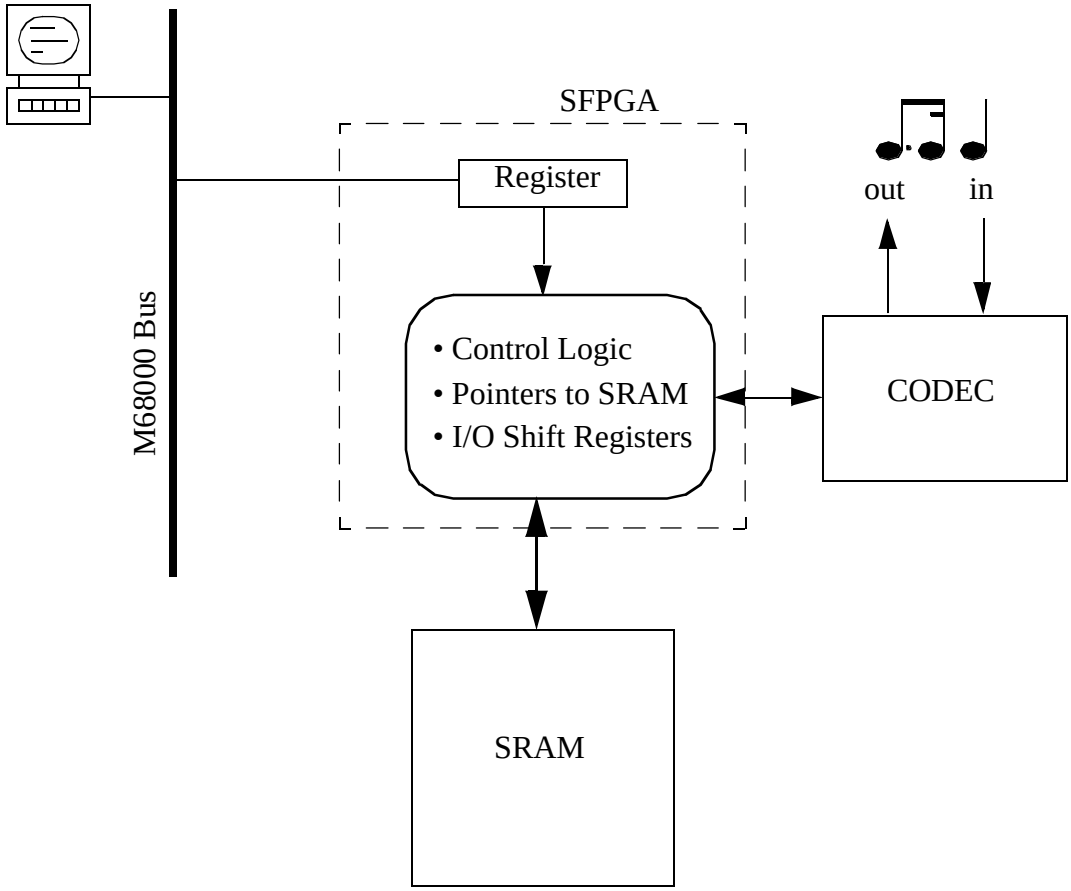


Figure 36 - *Block Diagram of Delay Circuit*

6 M68000 Assembly Language Programs

The tutorials in Chapter 2 give examples of actual M68000 assembly language programs and explain how to execute such programs on the University of Toronto Ultragizmo board. This chapter provides more detailed information. The first section summarizes the M68000 assembly language, the second section explains how to assemble a M68000 assembly language program, and the third section explains how to execute such a program. The fourth section shows what you should expect to see on your terminal when creating, assembling, and executing an M68000 program. The final section explains the procedure for compiling C code for the Ultragizmo board.

6.1 M68000 Assembly Language

Each line of an M68000 assembly language program can be either a comment, a statement, a directive, or a blank line. This section describes each of these basic constructs. It is your job to figure out which of these building blocks to use when writing your M68000 assembly language program.

Before delving into the details of the M68000 language, you should understand what the assembly location counter is. The assembly location counter is used by the assembler to keep track of where an instruction or datum is in memory. The assembler uses this information to translate symbolic labels into actual addresses.

The initial default value of the assembly location counter is 0. After processing an instruction or datum, the assembler automatically updates this counter by the number of bytes needed to store that instruction or datum. You can also change the value of the location counter at any point in an assembly language program by using an **org** directive at the appropriate spot in your program. For example, by placing an **org** directive before the first instruction in your program, the initial value of the assembly location counter will be set to the value specified in the **org** directive. (**org** directives are described in more detail in Section 6.1.3.)

Each assembly language program to be executed on the Ultragizmo board will have to have a starting address greater than or equal to $8001. The Ultragizmo board reserves locations $0 to $8000 for the FBUG monitor program, so your program **must** start with an **org** directive using an address of at least $8001. Refer to Table 4, “Memory Map of the Ultragizmo Board,” on page 93 for the Ultragizmo board’s memory map.

6.1.1 Comment

A line is treated as a comment if the character in column 1 is a star (*). A comment line is ignored by the assembler. See also the note on comments in the next section.

6.1.2 Statement

A line in an M68000 program can be a statement which contains an M68000 instruction that tells the M68000 processor what to do. Each statement contains up to four fields that appear in the following order:

```
            Label       Operator OperandList     Comment
```

An example of a statement that uses all four fields is:

```
            loop        move.l   #10,d1          ;move 10 into register d1
```

The fields are separated by white spaces.

A `Label` is a user-defined symbol which is assigned the current value of the assembly location counter. It must satisfy the following requirements:

1. It must be composed of alphanumeric characters, of which the first character must be a letter. Upper and lower cases are treated differently. For example, “Alpha” is a different label than “alpha”.
2. It may have any number of characters, but the assembler only looks at the first eight.
3. Each label must be unique within the first eight characters from any other label.

The `Label` field is optional. If a label is present, then it **must** begin in column 1 of the source line. Otherwise, one or more spaces (and/or tabs) **must** be typed to indicate the absence of a label. A label may appear on a line with no operator.

An `Operator` is an instruction mnemonic for an M68000 instruction and the **a68** assembler **requires** it to be in lower cases. In the example statement above, `move.l` is an operator. For a full listing of instruction (opcode) mnemonics, see the text, **Computer Organization**, by Hamacher, Vrenesic, and Zaky, fourth edition, or consult a Motorola manual.

An `OperandList` consists of one or more addressing modes that are separated by commas with no spaces. In the example statement, two addressing modes—immediate (`#10`) and register (`d1`)—are specified in the `OperandList`. For a full listing of addressing modes used to specify an operand, see **Computer Organization** or consult a Motorola manual. Parts of some addressing modes, such as absolute addressing or immediate addressing, are written as **expressions**. Section 6.1.2.1 below describes what an **expression** is.

The `Comment` field is ignored by the assembler; hence it may contain any characters. In the example statement, the comment begins a semicolon (;). This is not necessary but it makes the program easier to read for humans.

6.1.2.1 Expression

An **expression** can be used wherever a numerical value is needed. For example, it can be used in the `OperandList` of a statement (see Section 6.1.2) or in an **org**, **dc**, or **ds** directive (see Section 6.1.3). An **expression** is composed of **numbers**, **ASCII characters**, and **symbols** which can be either added or subtracted from each other to obtain an overall numeric value. Some examples are:

```
$cfdc
'A'-6
loop+3      where loop is a label defined in the program
```

6.1.2.2 Number

A **number** may be given in any one of hexadecimal, decimal, octal or binary notations. A hexadecimal number is denoted by a leading ‘$’ character or by a leading ‘0x’, an octal number by a leading ‘0’ character, and a binary number by a leading ‘%’ character. If there is no special leading character, then the number is assumed to be in decimal notation. Thus the decimal number 27 can be expressed as:

`$1B`	hexadecimal notaion
`$0x1b`	hexadecimal notation
`27`	decimal notaion
`033`	octal notation

If a negative number (the two's complement of a number) is desired, precede the number by a minus sign.

6.1.2.3 ASCII Character

When an **ASCII character** is preceded and followed by an apostrophe in an expression, it is assigned the 7-bit ASCII value of the character. For example, 'A' is assigned the value $41.

6.1.2.4 Symbol

A **symbol** must satisfy the following requirements:

1. It must be composed of alphanumeric characters, of which the first character must be a letter. Upper and lower cases are treated differently.
2. It may have any number of characters, but the assembler only looks at the first eight.
3. Each symbol, with the exception of "**d0**", "**d1**", "**d2**", "**d3**", "**d4**", "**d5**", "**d6**", "**d7**", "**a0**", "**a1**", "**a2**", "**a3**", "**a4**", "**a5**", "**a6**", "**a7**", "**sp**", "**sr**", and "**.**" (see below for an explanation), must be defined by appearing as a **label** within the program or by a direct assignment (see **equ** directive in Section 6.1.3).

Special symbols "**d0**", "**d1**", "**d2**", "**d3**", "**d4**", "**d5**", "**d6**", "**d7**", "**a0**", "**a1**", "**a2**", "**a3**", "**a4**", "**a5**", "**a6**", and "**a7**" represent the data and address registers of the M68000. Besides these special symbols, there are another three special symbols that the assembler knows about. One is "**sp**" which refers to the stack pointer, which in the M68000 is equivalent to address register a7. Another special symbol is "**sr**" which refers to the status register. The third special symbol '**.**' represents the assembly location counter. Recall that the assembly location counter points to the first word of the current instruction being assembled. Following is an example use of "**.**":

```
start movea.l #.,a0
```

Here, '**.**' refers to location `start` (i.e. the address of the `movea.l` instruction). An equivalent instruction would have been:

```
start movea.l #start,a0
```

6.1.3 Directives

The third basic construct for writing an M68000 program is an assembler directive, which is an instruction to the assembler. Some directives change the value of the assembly location counter, other directives generate machine code for data, while others assign numeric values to symbols. This section describes a few directives that you will find useful.

The **equ** directive requires a label (which must begin in the first column); some other directives (**dc** and **ds** in particular) have optional labels. If a directive does not have a label, it must leave at least one white space character at the start of the line.

Consult **Computer Organization** or a Motorola manual for a full listing of directives. (Note: the Motorola directives **opt** (options), **page** (advance to next page) and **spc** (space lines) are not supported by this assembler.)

6.1.3.1 ORG Directive

org <*expression*>

The **org** directive sets the assembly location counter to the value of <*expression*>. (Section 6.1.2.1 describes what an **expression** is.) Subsequent assembly code will be placed in memory beginning at the address specified by the **org** directive. The default value of the assembly location counter is 0 if **org** is not used. Note that **org** does not produce any data that is stored in memory, but specifies where subsequent instructions and data are going to be stored.

6.1.3.2 DC Directive

dc.<*wordlength*> <*expressionlist*>

The **dc** (declare constant) directive stores the data specified by <*expressionlist*> into successive locations in memory where the number of bytes for each location is specified by <*wordlength*>. The <*expressionlist*> is a comma-separated list of **expressions** (see Section 6.1.2.1) while <*wordlength*> is one of the letters **b**, **w**, or **l** indicating a byte, word, or long word, respectively. The assembly location counter is updated by the number of bytes that are allocated.

For example, the following

```
flag equ 30
org $200
dc.b 'A', 'B', 'C'
dc.b 0,flag,,flag+6
```

will load successive bytes beginning at address $200 as follows:

0200	41
0201	42
0202	43
0203	00
0204	30
0205	00
0206	36

The assembly location counter will be equal to 207 after the assembler has finished processing the last `dc.b` directive. An example of a word directive is:

```
org $300
dc.w $cfcc,$99
```

This will load the memory in the following fashion:

0300	cfcc
0302	0099

The **dc** directive can also be used to convert a string of **ASCII characters** into their 7-bit ASCII codes and insert them into memory at sequential byte addresses. The string is delimited by the same character at each end, where the delimiting characters may be any printing characters. For example, the following

```
org $400
dc.b ‘yes we have no bananas’
```

results in loading the memory like the following:

0400	79
0401	65
0402	73
0403	20
:	:

Note: the **dc** directive doesn’t work with all characters. Some special characters are dealt with differently. If it doesn’t work, break the **dc** statement into multiple parts and use the hexadecimal equivalent of the special characters.

6.1.3.3 DS Directive

ds.<wordlength> <number>

The **ds** directive advances the assembly location counter by the number of bytes, words, or operands indicated by <*number*>. The size of the storage units is specified by <*wordlength*>. This causes storage to be allocated, but not initialized to any particular value. An example of this directive is as follows:

```
org $500
dc.w $99
ds.w 2
dc.w $bb
```

This resulting memory load is:

0500	0099
0502	??
0504	??
0506	00bb

6.1.3.4 EQU Directive

<symbol> equ <expression>

The **equ** directive assigns the numeric value specified by <*expression*> to a <*symbol*>. (Section 6.1.2.1 explains what an **expression** is, while Section 6.1.2.4 explains what a **symbol** is.) The symbol **must** begin in the first column, otherwise the assembler will not recognize the **equ** directive and instead will print an error message. An example of a direct assignment is:

```
label0 equ $cfdc
```

6.1.3.5 EVEN Directive

.even

This directive causes the location counter to be advanced to the next even location if it is presently at an odd address.

6.2 Assembling an M68000 Assembly Language Program

Once you have written your M68000 program, you will need to assemble it, i.e. translate it into the 0's and 1's of M68000 machine code. To do this, use your favourite editor to enter your M68000 assembly-language program into a file on the *ugsparc* system. The name of the file should end in *.s*.

Assemble your program on the *ugsparc* system by executing the following command at a UNIX prompt (prompt%):

prompt% **a68 <filename>**

where <*filename*> is the name of the file containing your program. This will create another file with the same root name as the source file but with a *.srec* suffix. For example, the command

prompt% **a68 prog1.s**

will create a file called *prog1.srec* that contains the binary representation of the M68000 assembly-language program in *prog1.s.*

You can tell the assembler to a generate human-readable version of the translated machine code by using the **-l** option:

prompt% **a68 -l <filename>**

The listing is an annotated version of the original source program. Figure 10 (on page 17) shows an example of such a listing. For each M68000 instruction and each directive that specifies data storage, the listing gives its address in hexadecimal, the equivalent machine code in hexadecimal, and the original instruction or directive. This information is useful for setting breakpoints or tracepoints when you debug your programs.

By default, the listing will be printed to your screen. You can redirect the output into a file. For example, the command

prompt% **a68 -l prog1.s > prog1.l**

will create a file called *prog1.srec* as well as a file called *prog1.l.*

6.3 Executing an M68000 Assembly Language Program

Now that you've got a machine-code version of your M68000 program, you can execute it on the Ultragizmo board.

The first thing you need to do is to load the program and its data into the memory on the Ultragizmo board. First, make sure you are communicating with the monitor program on the Ultragizmo board. You should first log into the PC operating system and bring up the **CONPORT** and **DEVPORT** windows on your screen. There should be a 'Ultrag>' prompt in the **CONPORT** window. If there isn't, see Chapter 7 for instructions on how to get your PC communicating with the board. Use the **lo** command to tell the monitor to transfer a file from the PC:

Ultrag> **lo**

In the **DEVPORT** window, select menu item **transfer->send text file**. This will bring up a dialog box. In the dialog box, find the *.srec* file that has been compiled using the **a68** assembler and click the OK button. At this point you will see a few dots appearing on the screen, after which the prompt will return. A prompt will indicate that the download was successful. If you don't get a prompt, press the reset button on the Ultragizmo board and try again.

Once you have successfully downloaded your program to the Ultragizmo board, you can now execute it. This is done by typing "*go <ProgramOrigin>*" where *<ProgramOrigin>* is the address where the first instruction of your program is stored:

Ultrag> **go <ProgramOrigin>**

6.4 Putting It All Together

This section shows a single session in which you create, assemble, and execute an M68000 assembly language program called *lab1.s* where the first instruction is stored at the address $20100. (How would this be done?) Of course, you should not be doing all of this in a single seamless session at your terminal. We are showing such a session here to illustrate in one place how to assemble and execute programs on the Ultragizmo board.

It assumes that you already have three windows open on your PC: a **TERA TERM SSH** window connected to one of the *ugsparc* workstations, the **CONPORT** window, and the **DEVPORT** window.

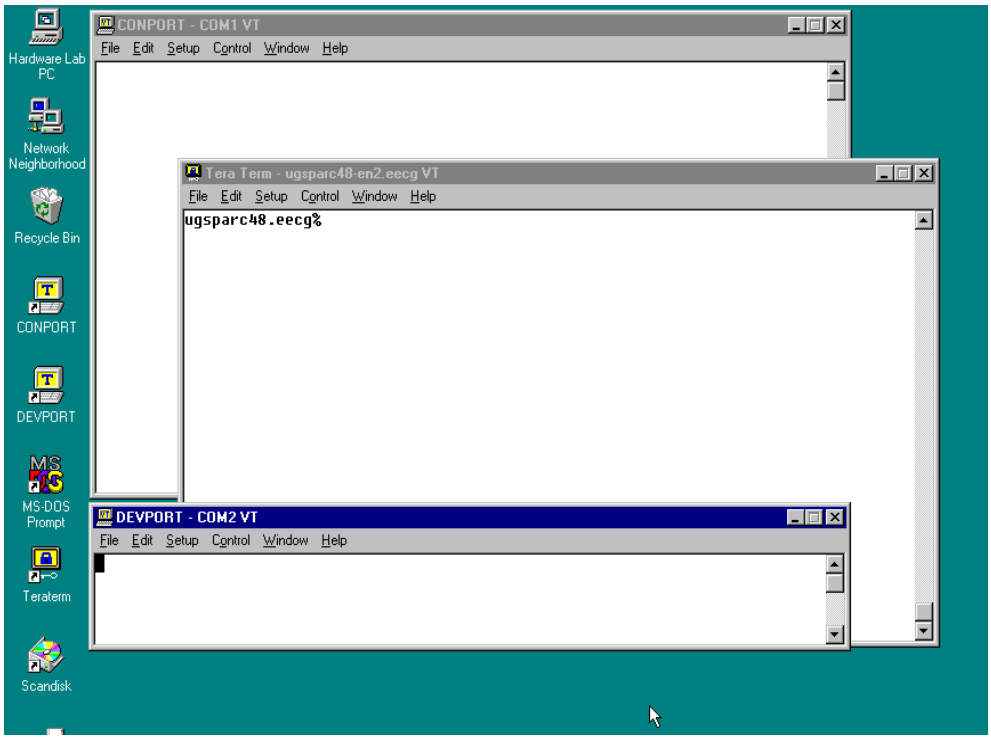


Figure 37 - *View of the Desktop*

First you should create your program on the *ugsparc* workstation through the **TERA TERM SSH** window. After *lab1.s* is created, compile the *lab1.s* using the **a68** command ("prompt%" means the prompt of the *ugsparc* workstation seen from the **TERA TERM SSH** window).

prompt% **a68 -l lab1.s > lab1.list**

This will create the *.srec* file, *lab1.srec* and the listing file, *lab1.list.*

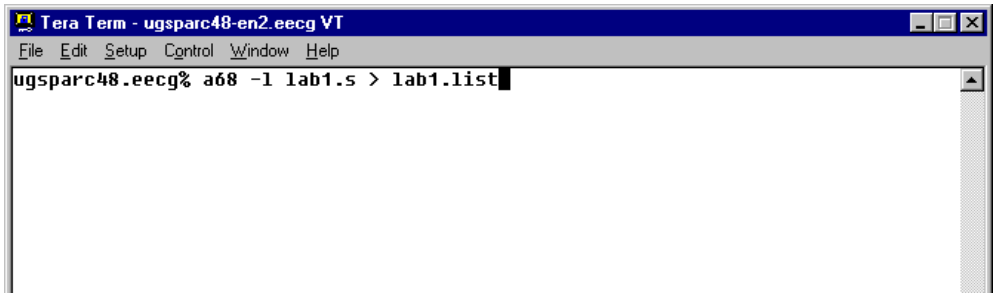


Figure 38 - *Tera Term SSH Window*

Next you should go to the **CONPORT** window, and using **lo** command to initiate the process of transferring the *.srec* file from the PC to the Ultragizmo board. (Remember that the "Ultrag>" prompt means that you are communicating with the Ultragizmo board.)

Ultrag> **lo**

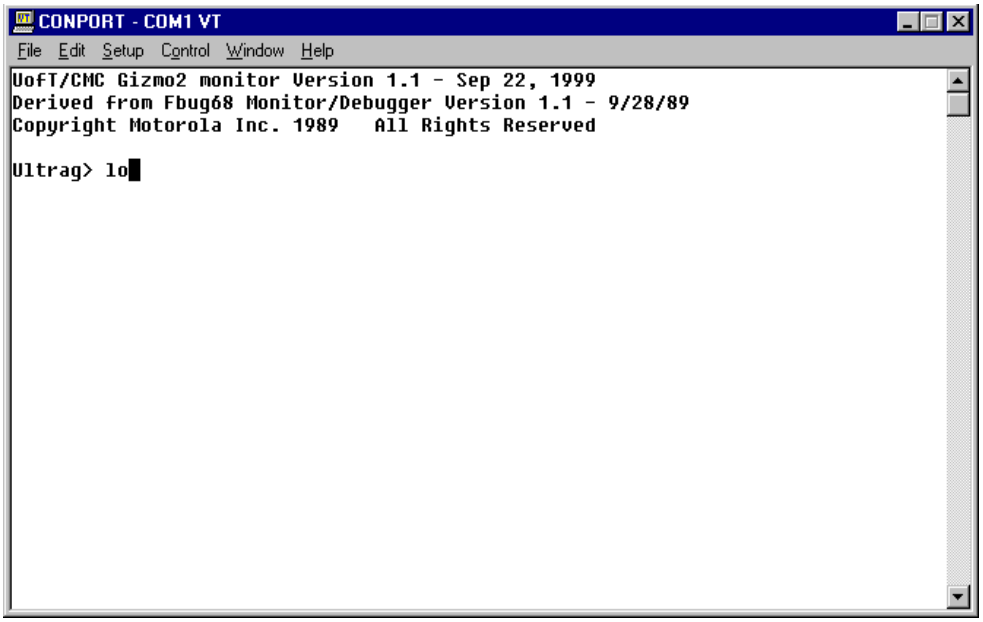


Figure 39 - *CONPORT Window*

Now the Ultragizmo board is waiting for the *.srec* file. Go to the **DEVPORT** window, select menu item **File->Send File**. Find the *lab1.srec* file in the dialog box and select **OK**.

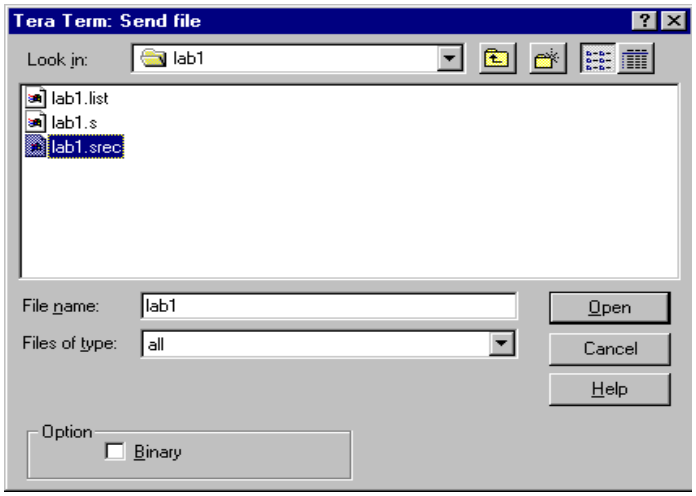


Figure 40 - *Send Text File Window in DEVPORT*

You should see some dots in the **CONPORT** window. A new "Ultrag>" prompt should also appear in the **CONPORT** window. Switch back to the **CONPORT** window and execute the assembly program by typing:

Ultrag> **go 20100**

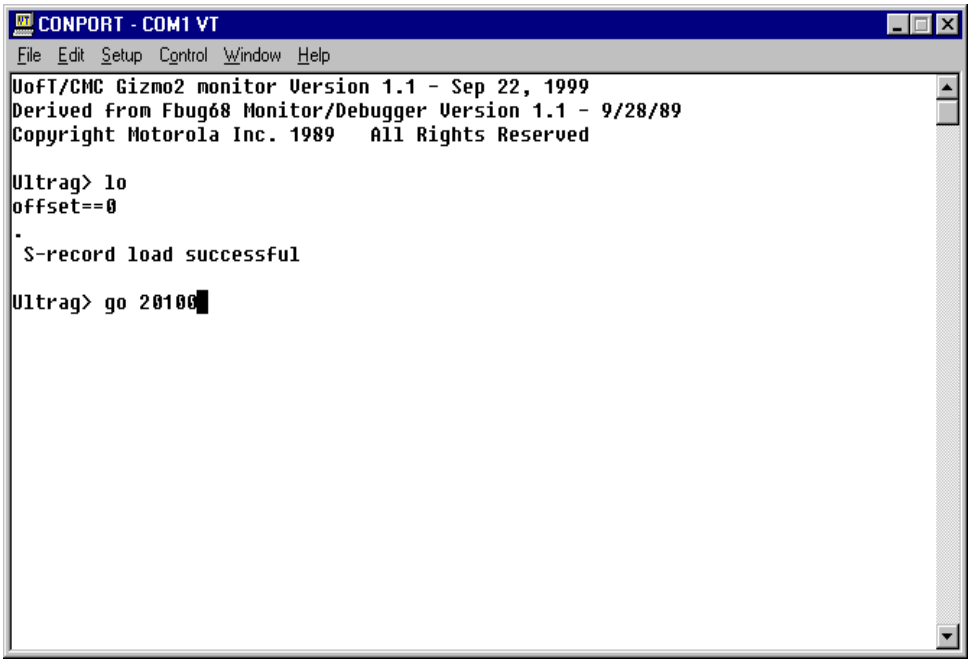


Figure 41 - *CONPORT Window after Successful Download*

6.5 Parallel Port Download

If you are executing a long M68000 assembly program, you may wish to download it onto the Ultragizmo board via the Centronics parallel port rather than through the serial interface, resulting in a faster download. In order to do this, click on the **CONPORT** window. At the 'Ultrag>' prompt, type the **lf** command:

Ultrag> **lf**

Then, run **maphome** to map your *ugsparc* account onto the *W:* drive. After that, click on *Start->Run* and type "command" in the pop-up window; this opens a DOS shell. In the DOS shell, type "W:" to change to the *W:* directory, and navigate to the directory containing your *.srec* file. Finally, at the DOS prompt, type

W:\> **copy <filename.srec> lpt1**

where <*filename.srec*> is the name of your file. Dots will appear on the screen as the file is downloading.

6.6 Using C with the Ultragizmo Board

A version of **gcc** which will produce executable code compatible with the Ultragizmo boards is installed on the *ugsparc* system. Please see Paul Chow's ECE352 website at *http://www.eecg.toronto.edu/~pc/courses/352/1999/lab1* for instructions on how to use it.

7 Using the FBUG Monitor Program

The tutorials in Section 2.2 give examples of how to use the FBUG monitor program to execute your M68000 program and to find errors which invariably occur in programs. This chapter provides a concise summary of those monitor commands as well as a description of how to start or restart the monitor program.

Note: The monitor program uses the lower 32K + 1 bytes of memory for data space and stack space. The UofT Ultragizmo board has 10Mbytes of RAM. Hence, you should write your program so that it gets stored between the memory locations $8001 and $9FFFFF to avoid accessing memory that is used for the monitor program and the stack.

7.1 Starting the Monitor Program

When the Ultragizmo board is first powered up, the monitor is communicating to your terminal through Serial Port B at 19.2K baud. You need to first start the monitor program by pressing the RESET button on the Ultragizmo board. (Figure 42 on page 92 shows the location of the RESET button on the board.) When the monitor program is running, it will display its prompt (Ultrag>) on the screen. The prompt also indicates that the monitor program is waiting for you to enter a command.

In the case of a program error, the Ultragizmo board may no longer respond to your keyboard inputs or may appear to be working incorrectly. If this happens, you need to reset the board to a known state. Resetting the board also restarts the monitor program. You do this by pressing either the RESET or the NMI (non-maskable interrupt) button on the Ultragizmo board. The RESET button is more “destructive” than the NMI button in that it resets all the board’s hardware. The advantage of this is that everything is in a known state after the RESET is pressed. The NMI button on the other hand interrupts whatever program is currently running and stores enough information such that this program can be continued later on. The NMI button is often called a *soft reset*.

Pressing NMI will first display the current contents of the M68000’s registers as well as the next instruction to be executed, and will then display the monitor prompt:

```
PC   =$00D0EA9E   SR   =$00002700   USP  =$00001B00
SSP  =$00007F88
D0   =$FFFFF7F3   D1   =$00000000   D2   =$00000120   D3   =$00000000
D4   =$00000000   D5   =$0000003F   D6   =$00000000   D7   =$00000000
A0   =$FFFFF7F3   A1   =$00002A9A   A2   =$000041E8   A3   =$00000000
A4   =$00000000   A5   =$00000000   A6   =$00007F90   A7   =$00007F88
$00D0EA9E 6A00 000C       bpl.w         $D0EAAC ?
Ultrag>
```

The first line shows the contents of the Program Counter (PC), Status Register (SR), and User Stack Pointer (USP). The second line shows the content of the System Stack Pointer (SSP). The third and fourth line show the contents of the data registers. Lines five and six show the contents of the address registers. Line seven shows the disassembled instruction at the address pointed by the program counter.

Pressing RESET will first display the monitor program version number and then display the

monitor prompt:

```
UofT/CMC Gizmo2 monitor Version 1.0 - June 26, 1995
Derived from Fbug68 Monitor/Debugger Version 1.1 - 9/28/89
Copyright Motorola Inc. 1989   All Rights Reserved
Ultrag>
```

7.2 Summary of Monitor Commands

The monitor commands are divided into several groups: program and memory inspection commands, execution control commands, and miscellaneous commands.

Most of the monitor commands take optional arguments. Table 2 shows a summary of the possible arguments.

Optional Argument	Description	Example
<*option list*>	An option delimiter (-) with options if non-default options are allowed and are being used.	-r
<*exp*>	An expression can be any numerical expression which may be evaluated using only the arithmetic + and - operators.	1000 1+3
<*addr*>	Address field is any valid expression.	2000
<*count*>	Count field is any valid expression preceded by the count delimiter (:).	:100
<*range*>	A range of memory locations denoted by either <addr>,<addr> or <addr>:<count>	0,100 0:50
<*text*>	An ASCII string of up to 255 characters preceded by the text delimiter (;).	;sample text
<*size*>	Can be either: byte (8-bit) ====> -b word(16 bit) ====> -w long(32 bit) ====> -l	 -b -w -l
<*data*>	Data can be any valid expression.	1000
<*symbol*>	Any monitor command	bf

Table 2 - *Summary of Optional Arguments to Monitor Commands*

Numerical values are hexadecimal by default. Hexadecimal, decimal, octal, and binary numbers also can be entered by preceding them with a leading $, &, @, or %, respectively.

Table 3 summarizes how to use the monitor commands. A table similar to this is printed when using the **h** (help) command. The arguments are enclosed in italicized square brackets '*[]*' to emphasize that they are optional. When you type a command with an argument, do not include the

square brackets. These commands are described in more detail in the text after the table.

Memory Inspection Commands	
Block Fill	*bf [<size>] <range> <data>*
Block Search	*bs [<size>] <range> <data>*
Memory Display	*md [<size>] <addr>* *md [<size>] <range>* *md -di <addr>*
Register Display	*rd*
Execution Control Commands	
Breakpoint	*br* *br <addr>* *br <addr>:<count>* *br -r [<addr>]* *br -r*
Continue	*co*
Go	*go [<addr>]*
Trace	*tr [<addr>][<count>]*
Miscellaneous Commands	
Data Conversion	*dc <exp>*
Help	*? [<symbol>]* *he [<symbol>]* *help [<symbol>]*
Load S-Record	*lo [<offset>]*
Load S-Record using parallel port	*lf*

Table 3 - *Summary of Monitor Commands*

7.2.1 Commands for Inspecting Memory

bf [<size>] <range> <data>

The **Block Fill** command fills the specified range of memory with the data listed. If the size option is not specified the default size used is word.

bs [<size>] <range> <data>

The **Block Search** command searches the *<range>* for an exact match of *<data>*.

md [<size>] <addr>
md [<size>] <range>

The **Memory Display** command displays the memory at the given <*addr*> or <*range*>.

md -di <addr>

The **Memory Display** command with disassemble option disassembles the memory starting at the given <*addr*>.

rd

The **Register Display** command displays the contents of all M68000 registers.

7.2.2 Commands for Controlling Execution

The execution-control commands control the execution of a program running on the Ultragizmo board. They can be used to execute instructions one at a time, to set and clear breakpoints.

You should be aware of the following when using these commands:

1. Breakpoint and single-step modes cannot be used to debug code that either prints characters to the screen or reads characters from the keyboard. Both of these modes use the same DUART as the terminal to perform the necessary I/O. Hence, any data stored in the DUART will be corrupted.
2. Similarly, be aware that if you have created and enabled an interrupt service routine to transfer data to and from the DUART, you will intercept the monitor's input and output. Hence you should not try to single-step or breakpoint a program once the interrupt is enabled, as the monitor will behave very erratically.

br

The **Breakpoint** command with no arguments lists all known breakpoints.

br <addr>

The **Breakpoint** command with the <*addr*> argument inserts a breakpoint at the given address.

br <addr> <:count>

The **Breakpoint** command with the <*addr*>*:*<*count*> argument inserts a breakpoint at the given address, however, returns to the monitor environment only after encountering the breakpoint <*count*> times.

br -r <addr>

The **Breakpoint** command with **-r** option removes a breakpoint at the given address <*addr*>.

br -r

The **Breakpoint** command with **-r** option and without arguments removes all breakpoints.

co

The **Continue** command resumes execution of the program from where it was last suspended.

go [<addr>]

The **Go** command executes the target program at the given address. If an address is not specified on the command line then the current PC value is used.

tr [<addr>] [<count>]

The **Trace** command allows the user to single-step though target code and observe the registers after executing the command line. If count is specified then the microprocessor executes <*count*> number of instructions before returning to the monitor environment. A **trace** begins from the <*addr*> listed on the command line or from the current PC if an <*addr*> is not included. The trace instruction can be continued by hitting a carriage return. To exit, a period (**.**) must be entered.

7.2.3 Miscellaneous Commands

dc <exp>

The **Data Conversion** command evaluates an input expression to determine its hexadecimal and decimal equivalent.

? [<symbol>]
he [<symbol>]
help [<symbol>]

The **Help** command allows the user to view a list of allowable commands and the syntax associated with them. Symbols used to describe the command usage can be looked up.

lo

The **Load** command waits for the host system to send an S-Record to the Ultragizmo board through the COM2 serial port.

lf

The **Load Fast** command waits for the host system to send an S-Record to the Ultragizmo board through the LPT1 parallel port.

cc

See Section 8.7 on page 130 for instructions on how to use the **Clock Configure** command to configure the programmable clock.