

Module 32 - Lab A: Virtual Prototyping Using VHDL

Fully Functional VHDL Modeling Tutorial using the i860XP and VME Bus Models

Copyright 1997-1999 SCRA

All rights reserved. This information is copyrighted by the SCRA, through its Advanced Technology Institute (ATI), and may only be used for non-commercial educational purposes. Any other use of this information without the express written permission of the ATI is prohibited. Certain parts of this work belong to other copyright holders and are used with their permission. All information contained, may be duplicated for non-commercial educational use only provided this copyright notice and the copyright acknowledgements herein are included. No warranty of any kind is provided or implied, nor is any liability accepted regardless of use.

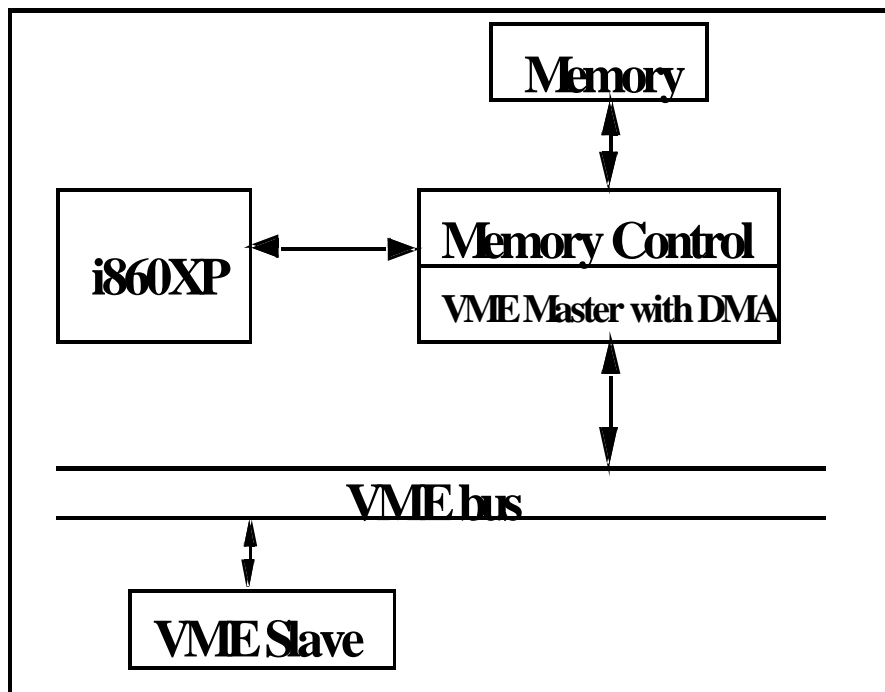
The United States Government holds “Unlimited Rights” in all data contained herein under Contract F33615-94-C-1457. Such data may be liberally reproduced and disseminated by the Government, in whole or in part, without restriction except as follows: Certain parts of this work to other copyright holders and are used with their permission; This information contained herein may be duplicated only for non-commercial educational use. Any vehicle, in which part or all of this data is incorporated into, shall carry this notice .

See the RASSP Disclaimer file for additional RASSP Disclaimer, Warranty and Limitation of Liability Information concerning the material, VHDL code and software developed under the RASSP programs or incorporated in RASSP material.

1. Overview

In this lab experiment, you will create a single processor embedded system design with a local memory, memory controller, and VME bus master and slave interfaces. VHDL models of the system elements will be compiled into libraries using the makefile contained in the top directory of your path. Three libraries will be generated. The first will contain the processor model, local memory, and its memory controller, the second, the VME master and slave interfaces, and the third contains a system level model to tie all the component elements together. The component models were developed at the fully behavioral (sometimes referred to as fully functional) level of abstraction. The figure below illustrates the component interconnection.

Figure 1: Laboratory design example.



The system will be used to explore the advantages of developing a virtual prototype at this level of design abstraction.

2. What you will learn

- 2.1. How to build libraries for component VHDL elements
- 2.2. How to build an embedded system design using library component elements
- 2.3. How to run software on the virtual prototype of the embedded system
- 2.4. The necessity for model interoperability
- 2.5. The types of errors that can be detected using this level of abstraction for a virtual prototype
- 2.6. The limitations of using a virtual prototype at this level of abstraction

3. Create the directory structure and component libraries

- 3.1. Create a working directory in your home directory with the name "FBM" and go to that directory to start working.

```
UNIX>> mkdir FBM
```

```
UNIX>> cd FBM
```

- 3.2. Copy the fully behavioral laboratory files to the newly created directory from the CD-ROM. For UNIX systems, use the *cp* command to copy the lab files into the directory just created, i.e. "FBM". This document assumes the use of Mentor on UNIX systems.
- 3.3. Setup your VHDL environment correctly so that you have access to the simulator's executables on the UNIX system. For Mentor, the environment variable MGC_HOME must point to the top-level directory of the Mentor tools.
- 3.4. Using the *tar* utility in UNIX, copy the zipped file containing the component models, m32_lab_a.tar, to your home directory. untar the file, and look at the directory structure. It should have the form shown in the Figure 2. If this is not available on your machine, then copy the files directly from the CD-ROM.

```
<home_dir> >> tar -xvf m32_lab_a.tar
```

- 3.5. The files in the *<design_files>* directory contain the entity/architecture pairs for the components of the system. The *<pkg_files>* directory contains the supporting packages needed by the design file entities and architectures. To compile the VHDL files for this example into separate libraries, the following order of compilation is required for each of the component elements.

- A. The files and compilation order for the i860XP model used in this example are:

1. datatype.hdl
2. BitMath_pkg.hdl
3. fp_pkg.hdl
4. memory_fcts.hdl
5. trap_reset_handler.hdl
6. instruction_set.hdl
7. process_handling.hdl
8. i860.hdl
9. clk_reset_gen.hdl
10. memory.hdl
11. memory_control.hdl

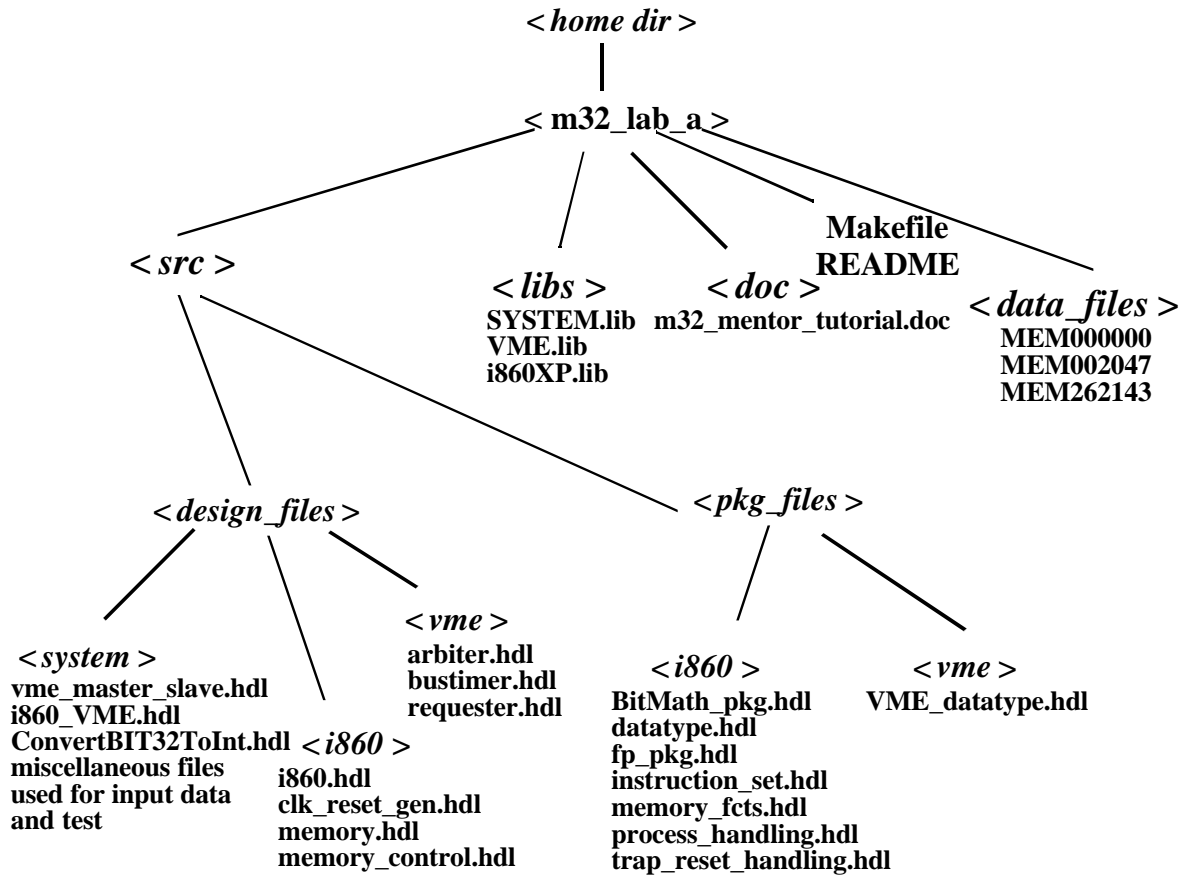


Figure 2: Laboratory File Structure

B. The files and compilation order for the VME model used in this example are:

1. VME_datatype.hdl
2. arbiter.hdl
3. bustimer.hdl
4. requester.hdl

C. The files and compilation order for the system level model is the following:

1. ConvertBIT32ToInt.hdl
2. vme_master_slave.hdl
3. i860_VME.hdl

The file *i860_VME.hdl* is the top-level configuration and is used to tie all the component models together. For quick compilation, use the *makefile* provided to build and compile all files into three libraries, *i860XP.lib*, *VME.lib*, and *SYSTEM.lib*. From the */m32_lab_a* directory, check for the QuickHDL environment variables being set before running the entire “make” process.

```
UNIX>> cd m32_lab_a
```

```
m32_lab_a>> make check
```

If the above command line operation returns "Proceed with compilation", then make all the files using the following.

```
m32_lab_a>> make all
```

After executing this command, three libraries will exist in the */m32_lab_a/libs* directory with the names, *i860XP.lib*, *VME.lib*, and *SYSTEM.lib*. As these libraries are being generated, proceed to the next section to acquaint yourself with the format for placing code and data into the memory model of the processor.

4. Running code on the i860XP processor

- 4.1. Two files exist, "VMEtestMEM1" and "interruptMEM", in the directory */m32_lab_a/src/design_files/system* containing code segments in binary and assembly language formats. Change to this directory and open the two files using your favorite editor (emacs is used in this writeup) and notice the instruction format. The instruction set word for the i860XP RISC processor is 32 bits wide.

```
UNIX>> cd ./src/design_files/system
/system>> emacs VMEtestMEM1 &
/system>> emacs interruptMEM &
```

Iconify the file VMEtestMEM1 for later viewing. Two additional files, MEM002047 and MEM262143, located in the */m32_lab_a/data_files* directory are used as the memory files for the design and contain code and data formatted for the processor. These files are automatically loaded into the VHDL memory model when an address, presented to the memory, falls within the specified range of the file. The contents of MEMnnnnnn are the integer equivalent of the binary numbers found in "VMEtestMEM1" and "interruptMEM". The file "interruptMEM" is used to handle interrupts and reset. This code is inserted at the reset location in memory, which corresponds to physical address 0xFFFFF00. Opening MEM262143 and going to line 4035 shows how (integer format) and where (0xFFFFF00) the code is placed in memory.

```
UNIX>> cd ../../../../
m32_lab_a>> emacs ./data_files/MEM262143 &
m32_lab_a>> goto line 4035 (meta-x goto-line 4035)
```

The first two lines of the "MEMnnnnnn" files contain the beginning and end location (assuming 32-bit word location) in memory where the contents of this file are placed.

```
editor>> Goto line 1 (meta-x goto-line 1)
```

- 4.2. "MEM002047" is the memory file containing the application or test code. After reset, the code at the reset location executes a branch to the start of this file. Figure 3 describes the application that will be used throughout the lab. The file "VMEtestMEM1" contains the application code.

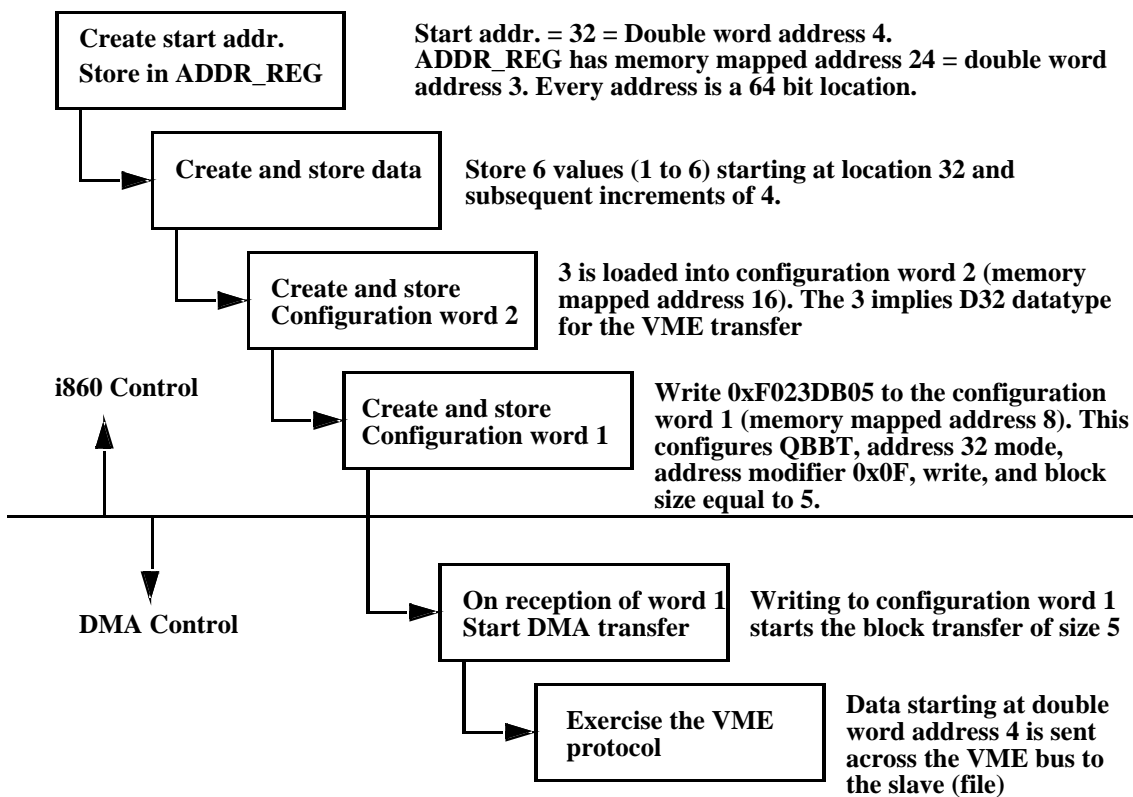


Figure 3: Program Flow of i860XP Application Code

For this example, the application code generates a sequence of numbers from 1 to 6 and writes the values to the processor's local memory. It also configures the memory controller by writing to memory mapped addresses 0x01, 0x02, and 0x03. These addresses are register locations in the memory controller for configuring the DMA and VME protocol. They specify the data transfer from the processor's local memory through the VME bus to the slave device. The layout of the three memory mapped registers are shown in Figure 4. The next step is to run the code and observe the results. From `/m32_lab_a` do the following.

```
m32_lab_a>> make simulate
```

At this time, the QuickHDL simulator interface window will appear on the screen. Select the following options to load into the simulator.

```
QuickHDL>> select 'ps' for resolution
```

```
QuickHDL>> select 'i860_VME' entity from work library
```

```
QuickHDL>> select 'structural' architecture
```

Click on the "load" button and the simulator will load the files from the library. To observe the results of simulation, the correct signals must be selected for viewing.

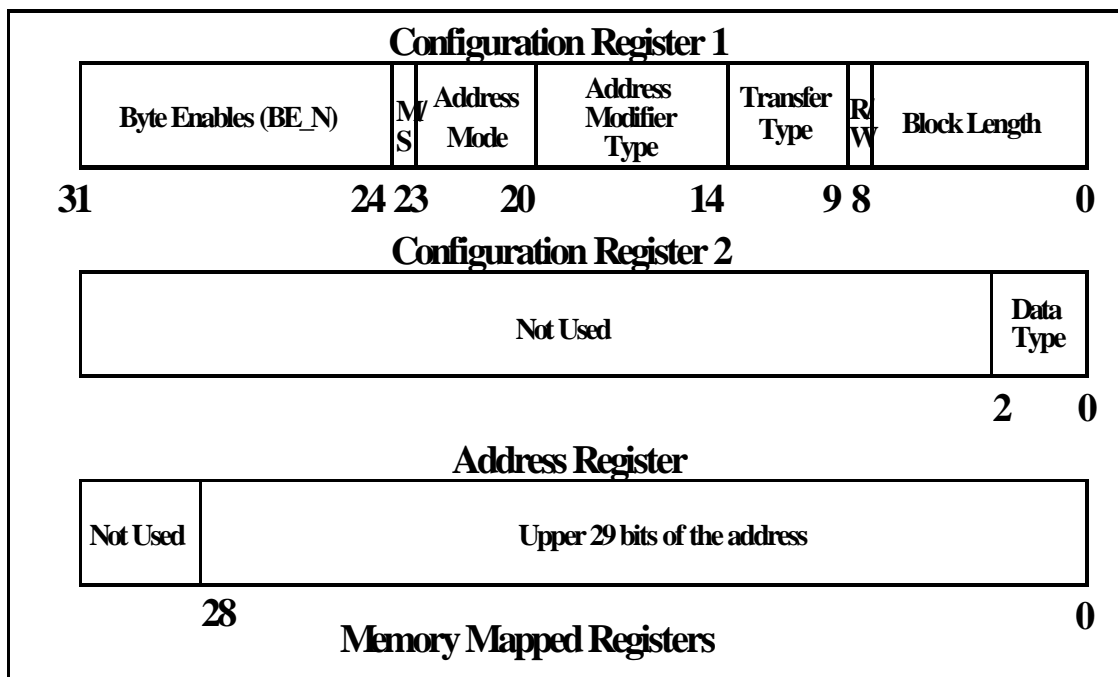


Figure 4: Layout of the Systems Memory Mapped Configuration Registers

From the view menu,

```
QuickHDL>> select View->Wave
```

The wave viewer appears. The signals to be viewed must now be loaded. A file in the *src/design_files/system* directory called "wave.do" contains the signals. To load it, execute the following from the QuickHDL file menu,

```
QuickHDL>> select File->Execute command file
```

```
QuickHDL>> select the file "wave.do" from FILE listing which is  
contained in the /src/design_files/system directory
```

```
QuickHDL>> click on the Execute button
```

The signals will now appear in the Wave window in the left-hand column. Run the simulation for 3000000 picoseconds. In the QuickHDL command line window do the following.

```
QHSIM>> run 3000000 ps
```

Observe the wave plot. From the Wave menu do the following,

```
Wave>> choose Zoom->Full Size
```

Zoom in on the segment where an instruction cache line fill takes place.

```
Wave>> choose Zoom->Range...
```

```
Wave>> Start = 300000 ps
```

```
Wave>> Stop = 450000 ps
```

In this output plot, the address lines start at 0x1FFFFFFE0 (reset address) and begin loading the instructions (64 bits on data bus => 2 instructions). By clicking on the falling edge of the "strobe" signal (in the wave window) and also on the location where the instructions are being placed on the data bus, we can measure the load time of memory. The difference is currently set to 10 ns. This is a generic parameter passed to the model and its usage will be explored later. Go back to a full zoom.

```
Wave>> choose Zoom->Full Size
```

We now look at the region where the data is being generated and sent to the memory. To compare what is on the screen with what is in the test file, open "VMEtestMEM1" if it is not already open.

```
Wave>> choose Zoom->Range...
```

```
Wave>> Start = 1100000 ps
```

```
Wave>> Stop = 1400000 ps
```

Click on the */address* line in the *Wave* window where the address has the value equal to 00000003. This is the address of the memory mapped ADDR_REG mentioned previously. It corresponds to the first "st.l" instruction in the VMEtestMEM1 file. The register R8 contains the value 4, the start address of the data to be stored. No memory strobe is generated in this case because the memory-mapped register is the target. Immediately following this write, we store the value '1' to memory. This is the second "st.l" instruction found in the file and corresponds to address 4 in the *Wave* window. Click on this location and observe the value on the data lines (0x0000000000000001). By continuing in this manner, observe on subsequent (click on the slide bar at the bottom of the window to move from screen to screen) *Wave* screens all the values being written to memory. The first *Wave* screen also contains another instruction cache fill which occurs on the second **adds_i 1,r2,r2** instruction in the "VMEtestMEM1" file. This is the ninth instruction in the file and since all cache fills are 8 instructions long, the ninth will result in a cache miss during the initial pass through the code. After observing all the data being transferred to memory, go back to full zoom.

```
Wave>> choose Zoom->Full Size
```

The last item to illustrate in detail is the transfer of data from the memory over the VME bus using the DMA in the memory controller. Do a zoom in the following region,

```
Wave>> choose Zoom->Range...
```

```
Wave>> Start = 2000000 ps
```

```
Wave>> Stop = 2400000 ps
```

Click on the first transaction on the address lines. This corresponds to the final "st.l" instruction in the file "VMEtestMEM1". The instruction writes to configuration register 1 and loads it with the value on the lower 32 bits of the data lines (0xF023DB05). This write also initiates the DMA transfer over the VME bus (at time 2012500 ps).

Before the transfer occurs, the DMA must obtain access to the address bus using the HOLD/HLDA arbitration of the processor. The controller sets the HOLD signal (at time 2037500 ps) and the processor, if all its bus transactions are complete, will give the bus to the controller by asserting HLDA (at time 2050000 ps).

The VME protocol is then initiated by requesting access to the bus (DWB set to '1' at 2060000 ps). The bus arbiter and requester determine if the bus is available and issue a bus grant (DGB set to '0' at 2090000 ps) to the controller. When the controller is granted the bus, it starts driving the address lines to read data from memory (at time 2110000 ps). Data is placed on the data bus after the memory access time, which is set to 10 ns in this case (at time 2140000 ps). The master notifies the slave that data is on the bus by setting the data strobe lines (DS0 and DS1) active (low)(at time 2150000 ps). The slave acknowledges the reception of data by asserting the DTACK line active low (at time 2160000 ps).

The cycle repeats itself until all the data specified by the block length is transferred. Observe the five data values being transferred across the bus on the DATA_32 lines. At the end of the transfer, the device releases the bus and the HOLD is de-asserted. The processor is then able to access the address lines and perform another cache line fill operation.

5. Model interoperability at the full behavioral level of abstraction

5.1. The section will illustrate the need for model interoperability. First, a system level entity is required to tie all the components together. The file *i860_VME.hdl* contains the structural-level system model where all the components are instantiated and connected through port maps. To induce an interoperability error, edit the file *vme_master_slave.hdl* in the */src/design_files/system* directory. Change directory to *src /design_files/system* and do the following.

```
UNIX>> cd ./src/design_files/system
/system>> emacs vme_master_slave.hdl &
editor>> search for the port signal DGB (ctrl-s DGB)
editor>> change its signal type from STD_LOGIC to BIT
editor>> save the file and keep it open (ctrl-x, ctrl-s)
```

From the */m32_lab_a* directory, compile the changes by typing the following,

```
UNIX>> cd ../../../../
```

```
m32_lab_a>> make system
```

The resulting compilation results in two errors of mismatched types. These errors are a result of one model assuming STD_LOGIC as its input and the other model using a BIT output. This illustrates the need for interoperable models among library components. The memory controller entity was expecting the DGB input in STD_LOGIC format while the library entity *vme_master_slave* required BIT. For all library components to work together using a common method of specification, the IEEE standard 1164 was adopted as the method to achieve interoperability for logic values. The 1164 standard defines the STD_LOGIC nine-value system with its corresponding resolution functions. All models following this standard will be interoperable at the pin level.

Undo the error that was created by editing the file and replace STD_LOGIC with BIT and compile the system again.

```
editor>> change the signal type on DGB from BIT to
STD_LOGIC
```

```
editor>> save the file (ctrl-x, ctrl-s)
```

```
m32_lab_a>> make system
```

6. Effects of memory speed and controller design on the performance of the code execution

6.1. In this section, we will vary the access time of the local memory to show the memory speed effects on the execution time of the code. This requires a flexible controller with adjustable wait states to account for slower memories. The memory is currently set for a 10 ns access time and the memory controller is configured with 0 wait states. From the current *Wave* window, note that the time when the bus grant becomes inactive is 2370000 ps. This is the baseline for which the next two measurements will be compared. Open the *i860_VME.hdl* file to modify the correct generic parameters.

```
/system>> emacs ./src/design_files/system/
i860_VME.hdl &
```

```
editor>> search for 'CHANGE_HERE', the section of code where
the components are instantiated (ctrl-s CHANGE_HERE)
```

The first code segment to modify is in the memory instantiation. Change the memory load time to 30 ns.

```
editor>> change 10 to 30 for the generic MEM_LOAD_TIME
```

```
editor>> search for the next occurrence of 'CHANGE_HERE'
```

This is the memory controller instantiation. Change the memory load time to 30 ns and the number of wait states to 1.

```
editor>> change 10 to 30 for the generic MEM_LOAD_TIME
editor>> change 0 to 1 for the generic MEM_WAIT_STATES
editor>> search for the next occurrence of 'CHANGE_HERE'
```

This is the VME master instantiation. Change the memory load time to 30 ns.

```
editor>> change 10 to 30 for the generic MEM_LOAD_TIME
editor>> search for the next occurrence of 'CHANGE_HERE'
```

This is the VME slave instantiation. Change the memory load time to 30 ns

```
editor>> change 10 to 30 for the generic MEM_LOAD_TIME
editor>> save the file (ctrl-x, ctrl-s)
```

Run the simulation and observe the effects of these changes on the completion time of the code. First, compile the system file because of the changes to the *i860_VME.hdl* file. In the */m32_lab_a* directory do,

```
m32_lab_a>> make system
```

In the simulator, restart the simulation.

```
QuickHDL>> choose FILE->Restart Design from the menu
QuickHDL>> push the button Restart and keep all settings
```

From the command line in the simulator, run the simulation by typing the following,

```
QHSIM>> run 4000000 ps
```

Observe the time the DGB line goes high after transferring the data. It occurs at 3245000 ps as compared to the previous 2370000 ps, an increase of 875000 ps or 37%. The total transfer time increases because of the slower memory resulting in an increased time for which the VME bus resource is locked.

Repeat the above steps again for memory that has a 60 ns load time and set the memory controller to 2 wait states. Run the simulation for 4500000 ps.

```
editor>> change the file as above with 60 ns memory access
time and 2 wait states in the controller
m32_lab_a>> make system
QuickHDL>> choose FILE->Restart Design from the menu
QHSIM>> run 4500000 ps
```

Observe the time the DGB line goes high after transferring the data. It occurs at 3995000 ps as compared to the original 2370000 ps, an increase of 1625000 ps or 69%.

Return to the original settings of 10 ns for the memory access time and 0 wait states for the controller and compile the system file.

```
editor>> change the file to its original state
m32_lab_a>> make system
```

7. Types of errors found by modeling at the full behavioral level of abstraction

In this section, we will explore the types of errors that can be detected in a design using fully behavioral models of the components.

We will address four common types of errors found in systems design and how this level of abstraction can detect them. These include bus protocol, bus operation, chip-to-chip interface, and software touching hardware errors. Additional errors can be found but will not be described in this section. These include, but are not limited to, functional errors where an ASIC performs an arithmetic function and produces incorrect results, control logic errors where signal polarities are reversed or inputs are not driven, and data handling errors which include corrupted bits or dropped bits.

7.1. Bus protocol errors

In this example, we introduce an error in the VME protocol by having the slave not return the data transfer acknowledge (DTACK) signal. Open the file *vme_master_slave.hdl* and search for DTACK_ERROR.

```
/system>> emacs ./src/design_files/system/vme_master
_slave.hdl &
editor>> search for the string 'DTACK_ERROR' (ctrl-s
DTACK_ERROR)
editor>> comment out the line following it, i.e. change
the line to -- DTACK <= '0' after TPD;
editor>> save the file (ctrl-x, ctrl-s)
```

Compile the file by typing the following in the */m32_lab_a* directory.

```
m32_lab_a>> make system
```

Load the design changes into the simulator environment, change the observable *Wave* window signals, and run the simulation again to obtain the results of the change.

```
QuickHDL>> choose FILE->Restart Design from the menu
```

In the Wave window, select edit and delete all the signals on the screen.

```
Wave>> choose Edit->Delete All
```

In the QuickHDL window load the wave3.do file.

```
QuickHDL>> select File->Execute command file
QuickHDL>> choose wave3.do and execute (wave3.do can be found in
the /src/design_files/system directory)
```

On the command line of the QuickVHDL window run the simulation for 4200000 ps.

```
QHSIM>> run 4200000 ps
```

Look at the range from 2000000 ps to 3300000 ps in the Wave window.

```
Wave>> choose Zoom->Range...
```

```
Wave>> Start = 2000000 ps
```

```
Wave>> Stop = 3300000 ps
```

Notice the data strobe lines (DS0 and DS1) were set (active low) at time 2150000 ps. The slave should respond with a data acknowledge (DTACK low) but it never occurred in this case. The end result was a time-out error as indicated by the BERR line (3160000 ps) going low 1010000 ps later. The bus timer logic in the file *bus_timer.hdl* set the BERR line when it was observed that the acknowledge signal had not occurred after 1 microsecond. This resulted in a bus protocol error and the data transfer was not completed.

This value is set using a generic in the file *i860_VME.hdl*. This VHDL file is at the highest level of the VHDL hierarchy (i.e. in the configuration). Open the file *i860_VME.hdl* and change this parameter using the steps below.

```
/system>> emacs ./src/design_files/system/i860_VME .hdl &  
editor>> search for BT_CHANGE (ctrl-s BT_CHANGE)
```

This is the instantiation of the bus timer entity. Change the BTO value from 1 to 2. The units are in microseconds and represent the delay time before a time-out can occur.

```
editor>> change BTO => 1 to BTO => 2
```

```
editor>> save the file (ctrl-x, ctrl-s)
```

Compile the files that were changed and simulate again by following the steps below,

```
m32_lab_a>> make system
```

```
QuickHDL>> choose FILE->Restart Design from the menu
```

```
QuickHDL>> push the button Restart and keep all settings
```

On the command line of the QuickHDL window run the simulation for 4200000 ps.

```
QHSIM>> run 4200000 ps
```

Look at the range from 2000000 ps to 4200000 ps in the Wave window.

```
Wave>> choose Zoom->Range...
```

```
Wave>> Start = 2000000 ps
```

```
Wave>> Stop = 4200000 ps
```

Observe the new location of the BERR signal going low. The difference between the time when the data strobe lines are active and when a bus time-out occurs is now 2 microseconds as expected. Undo all the previous changes to the files *vme_master_slave.hdl* and *i860_VME.hdl*.

```
editor>> search for 'DTACK_ERROR' in the first file and  
uncomment the line DTACK <= '0' after TPD;
```

```
editor>> search for 'BT_CHANGE' in i860_VME.hdl and  
change the 2 back to 1 on the BTO generic
```

Compile the system to verify the changes were done correctly before moving on to the next example.

```
m32_lab_a>> make system
```

7.2. Bus contention errors

In this section, an example of bus contention will be discussed where multiple components require the address bus. In the system model, both the DMA and the i860XP processor can drive the address lines at any given time, therefore, the need for arbitration is required. In this example, an error is inserted into the arbitration section of the code and the results are observed. Edit the files *vme_master_slave.hdl* and *memory_control.hdl*. The first file is in the *src/design_files/system* directory and the second in the *src/design_files/i860* directory. Starting with *memory_control.hdl* do the following,

```
editor>> emacs ./src/design_files/i860/memory_
control.hdl &
editor>> search for 'HOLD_ARBITRATION', it is located in
two places. Comment out the line immediately following
this search marker
```

Now do the same for the file *vme_master_slave.hdl*.

```
editor>> emacs ./src/design_files/system/vme_master_ slave.hdl
&
editor>> search for 'HOLD_ARBITRATION', it is located in
one place. Comment out the line immediately following
this search marker
editor>> save both files
```

Exit the simulator and recompile the system.

```
QuickHDL>> select File->Quit command file
m32_lab_a>> make all
m32_lab_a>> make simulate
QuickHDL>> select 'ps' for resolution
QuickHDL>> select 'i860_VME' entity from work library
QuickHDL>> select 'structural' architecture
```

In the Wave window, from the edit menu, delete all the signals on the screen so we can load the original wave file, *wave.do*.

```
QuickHDL>> select File->Execute command file
QuickHDL>> choose /src/design_files/system/wave.do and execute
```

Load the changed system into the simulator environment.

```
QHSIM>> run 3000000 ps
```

Look at the range from 2000000 ps to 2400000 ps in the Wave window.

```
Wave>> choose Zoom->Range...
```

```
Wave>> Start = 2000000 ps
```

```
Wave>> Stop = 2400000 ps
```

Notice HOLD was not asserted when the write to address 0x01 was completed. This causes the DMA to attempt to access memory at the same time as the cache fill is occurring. This condition can be seen by observing the 'X' values on the address lines, implying multiple drivers are not resolved to a known value. Data is being read from memory and being sent across the VME to the slave, but it is incorrect. This error is corrected using the proper hold arbitration for access to the address lines to memory. Go back to the two files and remove the errors (comment lines) that were inserted. Compile the corrected version to verify the changes were correctly inserted before moving to the next exercise.

```
editor>> emacs ./src/design_files/i860/memory_
control.hdl &
editor>> search for 'HOLD_ARBITRATION', it is located in
two places. Uncomment the line immediately following this
search marker
```

Now do the same for the file *vme_master_slave.hdl*.

```
editor>> emacs ./src/design_files/system/vme_master_ slave.hdl
&
editor>> search for 'HOLD_ARBITRATION', it is located in
one place. Uncomment out the line immediately following
this search marker
QuickHDL>> select File->Quit command file
m32_lab_a>> make all
m32_lab_a>> make simulate
QuickHDL>> select 'ps' for resolution
QuickHDL>> select 'i860_VME' entity from work library
QuickHDL>> select 'structural' architecture
```

In the Wave window, from the edit menu, delete all the signals on the screen so we can load the original wave file, *wave.do*.

```
QuickHDL>> select File->Execute command file
```

QuickHDL>> choose **/src/design_files/system/wave.do** and **execute**

Load the changed system into the simulator environment.

QHSIM>> run 3000000 ps

Look at the range from 2000000 ps to 2400000 ps in the Wave window.

Wave>> choose **Zoom->Range...**

Wave>> Start = **2000000 ps**

Wave>> Stop = **2400000 ps**

7.3. Chip-to-chip interface errors

In this test, we are going to observe the effects of a chip-to-chip interface problem where the memory controller is assuming a fast memory access time (10 ns) while the actual memory is slower (30 ns).

For this example, open the file *i860_VME.hdl* and search for **CHANGE_HERE**.

```
/system>> emacs ./src/design_files/system/i860_VME.hdl &
/system>> search for the first occurrence of
'CHANGE_HERE' (ctrl-s CHANGE_HERE)
```

This will be the instantiation of the memory component. Change the memory load time generic from 10 ns to 30 ns.

```
editor>> Search for the next two occurrences of
'CHANGE_HERE', change the value of the MEM_LOAD_TIME
generic from 10 ns to 30 ns and save the file (ctrl-x,
ctrl-s)
```

Compile the system.

m32_lab_a>> make system

Load the changed system into the simulator environment.

QuickHDL>> choose **FILE->Restart Design** from the menu

QuickHDL>> push the button **Restart** and keep all settings

On the command line of the QuickHDL window, run the simulation for 450000 ps.

QHSIM>> run 450000 ps

Notice the warnings of the form 'illegal bit detected' -- not '1' or '0' in the QuickHDL window. This implies that data is being read from the data bus when the values are in an unknown state. Look at the simulation results using the full range.

Wave>> choose **Zoom->Full Size** from the Wave window menu

Looking at the data lines at time 350000 ps, we see they all contain the value 'Z', implying no data on the bus at the time when the controller expected it. This is what caused the warning messages to appear in the QuickHDL command line window. The BRDY_N signal is sampled

on the rising edge of the clock and if it is low, the data is ready to be input to the processor. The controller set the BRDY_N signal low with the assumption that the data is on the bus, however, it did not arrive until 17.5 ns later, at time 367500 ps. The data was also corrupted with some 'X' and 'Z' values because the memory does not know whether to react to the first or second strobe.

Undo the change made at the beginning of this section by doing the following.

```
editor>> Search for 'CHANGE_HERE' in i860_VME.hdl and
change the three locations where the MEM_LOAD_TIME generic
was changed. Change it from 30 ns to 10 ns again and save
the file
m32_lab_a>> make system
```

7.4. Errors where software is addressing hardware

In this section, we will examine the common type of error found when there is a misinterpretation between hardware and software engineers. This occurs when code is written with the assumption that the address that is being written is the one the hardware engineers designed it to be. To illustrate, we use another set of code contained in the file "VMEtestMEM2". The difference between the files can be observed by typing the following in the */src/design_files/system* directory.

```
/system>> diff VMEtestMEM1 VMEtestMEM2
```

The address being set for the ADDR_REG is byte address 32 instead of the correct value of 24.

We need to make the new memory file. Open the file *ConverBIT32ToInt.hdl* contained in the */src/design_files/system* directory and search for the string "VMEtestMEM1".

```
/system>> emacs ./src/design_files/system/
ConvertBIT32ToInt.hdl &
editor>> search for the string 'VEMtestMEM1', change the
string to 'VMEtestMEM2' and save the file
```

To generate the new memory file we must compile the changed file and run the simulator to create the data. This can be done by typing the following in the */m32_lab_a* directory.

```
m32_lab_a>> make mem_file
```

A prompt of the form **QHSIM>** will appear which is the command line access for simulating a design. To generate the data, type the following at the command line prompt.

```
QHSIM> run 0 ns
QHSIM> quit
```

A new memory file, "MEM002047" is created in the */data_files* directory. We can run the system simulation again using the new file as input and observe the results.

Load the changed system into the simulator environment.

QuickHDL>> choose **FILE->Restart Design** from the menu

QuickHDL>> push the button **Restart** and keep all settings

On the command line of the QuickHDL window, run the simulation for 3000000 ps.

QHSIM>> run 3000000 ps

Look at the range from 2000000 ps to 2400000 ps in the Wave window.

Wave>> choose **Zoom->Range...**

Wave>> Start = **2000000 ps**

Wave>> Stop = **2400000 ps**

In this case, the desired data is no longer crossing the VME data lines. Looking at "addr_word", the signal at the bottom of the display, which is passed the contents of the ADDR_REG, it contains its reset value of all zeros. This implies we never wrote our intended value of double word 4 to that address.

Undo the previous changes by typing the following,

```
/system>> emacs ./src/design_files/system/
ConvertBIT32ToInt.hdl &
editor>> search for the string 'VEMtestMEM2', change the
string to 'VMEtestMEM1' and save the file
m32_lab_a>> make mem_file
QHSIM> run 0 ns
QHSIM> quit
```

Reload the design and simulate to verify the correct performance.

QuickHDL>> choose **FILE->Restart Design** from the menu

QuickHDL>> push the button **Restart** and keep all settings

On the command line of the QuickHDL window run the simulation for 3000000 ps.

QHSIM>> run 3000000 ps

We have now completed the section on the types of errors commonly found in connecting components together in the prototyping stage of design. This completes the virtual prototyping lab at the fully behavioral level in the design process. It emphasizes the types of information that can be gleaned from using models of processors running code in a system cosimulation environment in order to help achieve first-time design success.