ECE 411: Computer Organization and Design

MP1: The LC-3bα Processor / FPGA Advantage Tutorial

Version 3.2

Table of Figures

1. Introduction

Welcome to the first ECE 411 Machine Problem! In this MP we will step through the design entry and simulation of a simple, non-pipelined processor that implements a subset of the LC-3b instruction set architecture (ISA). We will refer to this subset of the LC-3b ISA as the LC-3bα ISA. This tutorial (along with material on the course web page) contains the entire design - essentially you will follow the step-by-step directions to enter it.

The primary objective of this exercise is to give you a better understanding of the important features of the Mentor Graphics design tools FPGA Advantage and ModelSim. For later MPs, you will use FPGA Advantage for design entry and ModelSim for design simulation. Since your next MPs will require original design effort, it is important for you to understand how these tools work now so that you can avoid being bogged down with tool-related problems later.

We will discuss the MP1 LC-3bα processor extensively in class during the first several lectures. Understanding the lecture material will deepen your understanding of this tutorial, and likewise, going through this tutorial will help you understand the corresponding lectures. Later MPs will build heavily upon this design, so it is very important for you to get the MP1 LC-3bα processor working correctly.

The remainder of this chapter describes some notation that you will encounter throughout this tutorial. Most of this notation should not be new to you; however, it will be worthwhile for you to reacquaint yourself with it before proceeding to the tutorial itself. The second chapter contains a description of the six instructions in the LC-3bα instruction set. The third chapter contains a high-level view of the design. The fourth chapter is the step-by-step procedure for entering the design of the processor using FPGA Advantage. The fifth chapter covers the simulation of the design using ModelSim. The final chapter contains the items you will need to submit for a grade. Also included are several appendices that contain additional useful information.

As a final note, **read each and every word of the tutorial**, and follow it very carefully. There may be some small errors and typos. However, most problems that past students have had with this MP came from missing a paragraph and omitting some key steps. Take your time and be thorough, as you will need a functional MP1 design before working on future MPs.

1.1. Notation

The numbering and notation conventions used in this tutorial are described below:

- Bit 0 refers to the LEAST significant bit.
- Numbers beginning with **0x** are hexadecimal. In order to avoid ambiguity between decimal numbers and binary numbers, occasionally decimal will be prefixed with a **0#** in situations where the base of the number is not clear.

- [address] means the contents of memory at location `address'. For example, if MAR = 0x12, then [MAR] would mean the contents of memory location 0x12.
- For RTL descriptions, the following operators are used:
  - X^y means y consecutive X's, e.g., 0^3 is 000.
  - field[x:y] identifies a bit field consisting of bits x through y of a larger binary pattern. For example, X[15:12] identifies a field consisting of bits 15, 14, 13, and 12 from the value X.
- A macro instruction (or simply instruction) means an assembly-level or ISA level instruction.
- Commands to be typed at the terminal are shown in a grey box as follows:

```
[netid@linux6 ~] command
```

Do not type the portion in square brackets (`[netid@linux6 ~]`); this is called the 'prompt' and is automatically displayed by the shell (usually `/bin/bash`).

2. The LC-3bα Instruction Set Architecture

2.1. Overview

For this project, you will be entering the VHDL design (using FPGA Advantage) of a non-pipelined implementation of the LC-3bα ISA. We will discuss the LC-3bα ISA extensively in class.

The LC-3bα ISA consists of six instructions selected from the full LC-3b ISA (19 instructions). The LC-3b ISA is an ISA created for instructional purposes. Because it is a relatively simple ISA, it is a natural choice for our ECE 411 projects.

All six instructions are 16 bits in length, having a format where bits [15:12] contain the opcode. The LC-3bα ISA is a **Load-Store ISA**, meaning data values must be brought into the General-Purpose Register File before they can be operated upon. Each general-purpose register (GPR) is 16 bits in length, and there are 8 GPRs total.

The memory of the LC-3bα consists of 2^{16} locations (meaning the LC-3bα has a 16-bit address space) and each location contains 8 bits (meaning that the LC-3bα has byte addressability).

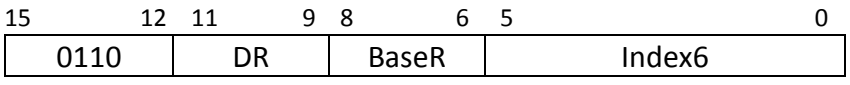
The LC-3bα program control is maintained by the Program Counter (PC). The PC is a 16-bit register that contains the address of the **next** instruction to be fetched

2.2. Data Movement Instructions

Data movement instructions are used to transfer values between the register file and the memory system. The load instruction (LDR) reads a 16-bit value from the memory system and places it into a general-purpose register. The store instruction (STR) takes a value from a general-purpose register and writes it into the memory system.

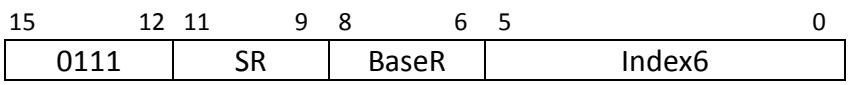
The format of the load instruction, or LDR, is shown below. The opcode of the LDR instruction bits [15:12]) is 0110. The effective address (the address of the memory location that is to be read) is specified by the BaseR and index6 fields. The effective address is calculated by adding the contents of the BaseR to the zero-extended and left-shifted-by-one index6 field.

LDR



The format of the store instruction, STR, is shown below. The opcode of this instruction is 0111. As with the load instruction (LDR), the effective address is the memory location specified by the BaseR and index6. The effective address is formed in the same manner as that of the LDR.

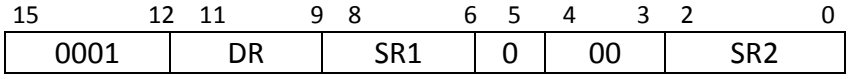
STR



2.3. Operate Instructions

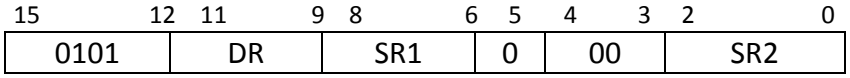
LC-3bα has three integer operate instructions: ADD, AND, and NOT.
The ADD instruction takes a value from the general-purpose register specified by SR1 (SR stands for source register) and adds it the value from the register specified by SR2. The result is stored in the register specified by DR (DR stands for destination register). The format of the ADD instruction is shown below. The opcode for ADD is 0001.

ADD



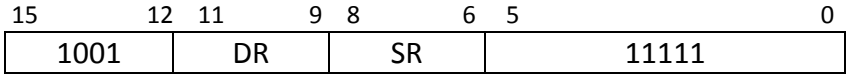
The AND instruction works similar to the ADD instruction, except the operation performed is a bitwise AND of the two source registers. The opcode for AND is 0101 and its format is shown below.

AND



This is the format of the NOT instruction. It does a bitwise complement on the value in register SR and places the result in register DR.

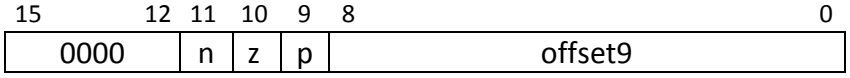
NOT



2.4. Control Instruction

The LC-3bα branch instruction, BR, causes program control to branch to a specified address. The format of the instruction is given below. Specifically, it works as follows: if the n, z, and/or p bits in the instruction are set, and the corresponding condition code is asserted, the processor will take the branch. When the branch is taken, the address of the next instruction to be executed is calculated by adding the incremented PC value to the sign-extended and left-shifted offset9 field.

BR



3. Design Specifications

3.1. Signals

The microprocessor communicates with the outside world through an address bus, a data bus, and five control signals, as well as a clock.

3.1.1. External inputs

`RESET_L`
: Active low signal that puts the processor in the initial state. Once in this state, only a START_H signal will cause the processor to leave the initial state.

`START_H`
: Active high signal that causes the processor to execute the instruction located at memory address 0x0000. It must stay high for at least one clock cycle. Afterward, START_H may change state without affecting the microprocessor's operation.

`CLK`
: A clock signal. All components of the design are active on the rising edge.

3.1.2. Memory Subsystem Signals

`ADDRESS(15:0)`
: Memory is accessed using this 16-bit signal.

`DATAIN(15:0)`
: 16-bit data bus receiving data from memory.

`DATAOUT(15:0)`
: 16-bit data bus sending data to memory.

`MREAD_L`
: Active low signal that tells memory that the address is valid and that the processor is trying to perform a memory read.

`MWRITEL_L`
: Active low signal that tells memory that the address is valid and that the processor is trying to perform a memory write to the low byte location.

`MWRITEH_L`
: Active low signal that tells memory that the address is valid and that the processor is trying to perform a memory write to the high byte location.

`MRESP_H`
: Active high signal generated by memory indicating that the memory has finished the requested operation.

The convention that is used for all signal names (except buses, e.g. `ADDRESS`, `DATA`, etc.) is to append an underscore and polarity to the end of the signal name. For example, `XYZ_H` indicates that the signal XYZ is active when high.

3.2. Bus Control Logic

The memory system is asynchronous, meaning that the processor waits for the memory to respond to a request before completing the access cycle. In order to meet this constraint,

inputs to the memory subsystem must be held constant until the memory subsystem responds. In addition, outputs from the memory subsystem should be latched if necessary.

The processor sets the MREAD_L control signal active (low) when it needs to read data from the memory. The processor sets the MWRITEL_L and MWRITEH_L signals active when it is writing to the memory. MREAD_L and MWRITEL_L or MWRITEH_L must never be active at the same time! The memory activates the MRESP_H signal when it has completed the read or write request. We assume the memory response will always occur so the processor never has an infinite wait.

3.3. Reset Logic

It is necessary to be able to reset the processor (put the processor in a known state). An external signal, RESET_L, can put the microprocessor to a known state by clearing the PC register and sending your controller into a reset state. The controller will remain in the reset state until START_H is received. Then the instruction at location 0x0000 of the main memory is executed.

3.4. Hard-Wired Controller

There is a sequence of states that must be executed for every instruction. Its function is to fetch and decode the current instruction.

4. Design Entry

4.1. Setup

If you do not have an EWS account, please contact one of the TAs, and they will help you obtain an account.

The purpose of this MP, as stated before, is to become acquainted with the LC-3bα ISA and with the software tools. You will be using FPGA Advantage from Mentor Graphics to layout the design and ModelSim to simulate it.

If you wish to learn more about the features in FPGA Advantage, you can go through the FPGA Advantage tutorial, which is available through FPGA Advantage itself (click on Help). The tutorial covers additional topics not covered here.

To start using FPGA Advantage, first make sure you are in your EWS home directory. Type the following in a terminal window (Applications > System Tools > Terminal) on an EWS Linux machine (Everitt Lab 252 recommended):

```
[netid@linux6 ~] cd ~
```

Create a directory for ece411 work:

```
[netid@linux6 ~] mkdir ece411
```

Change into your newly created directory:

```
[netid@linux6 ~] cd ece411
```

Print the current directory as a sanity check; the output should appear as below:

```
[netid@linux6 ece411] pwd
/home/<netid>/ece411
```

Download the given files from the course webpage and extract them (the 'wget' command can be typed on a single line):

```
[netid@linux6 ece411] wget
courses.engr.illinois.edu/ece411/mp/mp1/ece411_given.tar
[netid@linux6 ece411] tar –xvf ece411_given.tar
```

Three folders will be extracted: *bin*, *ece411_given*, and *testcode*. */bin* contains executables that will be used throughout the semester; */testcode* is where you will put your test code to debug your design, and */ece411_given* contains the main design files for use in FPGA Advantage.

Finally, make a copy the *ece411_given* folder to begin mp1:

```
[netid@linux6 ece411] cp –r ece411_given mp1
[netid@linux6 ece411] ls
bin    ece411_given   ece411_given.tar   mp1   testcode
```

Open FPGA Advantage:

```
[netid@linux6 ece411] fpgadv
```

Once FPGA Advantage has opened, open the project file we have provided you by selecting **File → Open**. Then select the file named *ece411.hdp* in the *mp1* directory. Make sure you do not accidentally select *ece411_mp3.hdp*.

4.2. Setting Default Libraries

In the Design Manager, select **Options → VHDL**, then select the **Default Packages** tab. Double click in the **Default Package References** text field and enter the following:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.NUMERIC_STD.all;

LIBRARY ece411;
USE ece411.LC3b_types.all;
```

This will set the default libraries for every file you create in FPGA Advantage. MAKE SURE your default packages are what is listed above before creating any new blocks or components. Many students have spent hours searching for errors caused by having incorrect packages.

4.3. Beginning the Design

4.3.1. Opening the CPU Block Diagram

In the Design Manager, double click on the *mp1_CPU* under **Design Unit** to open the CPU block diagram. The block diagram is a blank sheet except for a background grid, a default package list, and a comment block.

4.3.2. Add and Name New Blocks

In your block diagram window, select **Add → Block**, then add three blocks on your block diagram, placing them roughly as shown in Figure 1. The blocks are added with the default library < library >, the default name < block > and unique instance names (U_0, U_1, and U_2). The block instance names must be unique.

Click on the text “<block>" in the lower block on the left and notice the small handles that indicate that the text object is selected. Click again and notice that the text is now highlighted and can be directly overwritten. Change the default names of the blocks to Control, Memory, and Datapath.

Once you have changed the three block names, we will go ahead and edit the unique instance names of the blocks. Although FPGA Advantage provides us with default names (U_0, U_1, etc.) that are unique, it is very useful to enter your own unique name to assist in debugging later. Double click on the instance name of the Memory block and give it the name *DRAM* and click outside the text to complete the edit. Repeat this procedure to change the instance name for Datapath to *theDatapath* and the instance name for Control to *ControlUnit*. Your diagram should now look similar to **Figure 1**.

Note: A command normally auto-repeats until you select another command or terminate the repeating command by using the right mouse button or the ESC key. You can change the behavior of the toolbar buttons by setting the Activate Once Only preference in the General Preferences dialog box. You can also use the Shift key with any toolbar button to toggle the repeat mode.

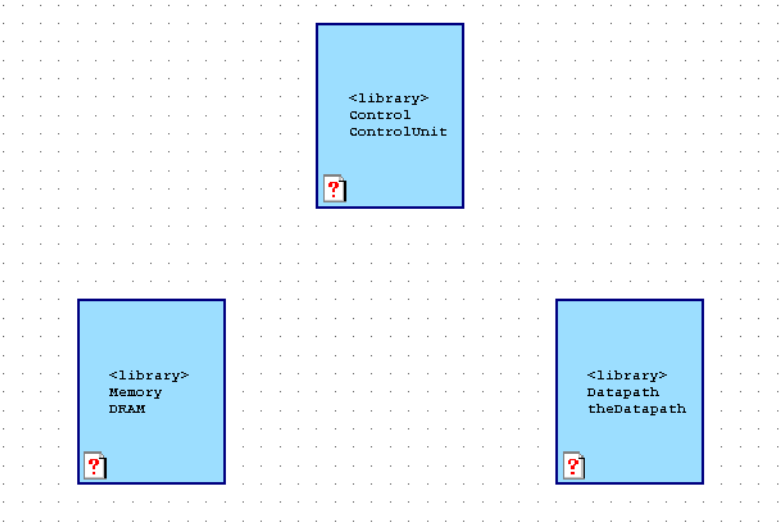


Figure 1: The new blocks are named

4.3.3. Save the Block Diagram

Note the asterisk (*) character in the title bar of the block diagram editor window. This indicates that the diagram has been edited since it was last saved.

Select **File → Save t**o save the block diagram. Notice that the * character has been cleared in the block diagram header. Click on the plus icon next to the *mp1_CPU* design unit in the Design Manager to expand the design unit. This reveals that it contains a symbol and block diagram view.

Note: A red cross superimposed on any icon in the Design Manager indicates that the view is not writable. This convention is also used to show when you have read-only access. This will be of importance when you are sharing design units during group work.

4.3.4. Add Ports and Signals

A signal is a single wire connecting blocks and is drawn as a thin line. A bus is a group of wires connecting blocks, and is drawn as a thicker line than a signal. Ports are the interfaces between blocks and signals or buses. All three types of items can be added by selecting the appropriate commands under the Add menu, or there are buttons in the toolbar which perform equivalent functions. Signal/bus names can be changed by double-clicking the existing name.

In the *mp1_CPU* block diagram editor, perform the following steps:

1) Create three signals originating at Control and ending at Memory. Name these MWRITEL_L, MWRITEH_L and MREAD_L.
2) Add another signal from Memory to Control. Name the signal MRESP_H.

3) Create two port-ins. Connect one signal from each port-in to the Control block. Name the signals RESET_L and START_H.

4) Connect the RESET_L signal to Memory and Datapath as well, by starting another signal from a point on the existing signal. HDL Designer should automatically rename the new signals.

5) Create two buses from Datapath to Memory. Name the buses DATAOUT and ADDRESS.

6) Create another bus from Memory to the Datapath, called DATAIN.

Signals and buses should look similar to those found in Figure 4 when the above steps are completed. You can drag the names of the signals/buses to more visible or convenient places, if you wish. Finally, you can right click and select **Add Route** to create more corners on the signal/bus if needed.

Note: In FPGA Advantage, the distinction between signals and busses is purely cosmetic: busses appear thicker when drawn. There is no functional difference between a signal and a bus: they are completely interchangeable.

Now that you have created and named new signals, it is time to set some additional properties, such as bus width and signal type.

By right clicking on any of these objects and selecting Object Properties you can display the “BD Object Properties" window. From here you can modify any of the Declaration fields to specify the type of a signal or bus.

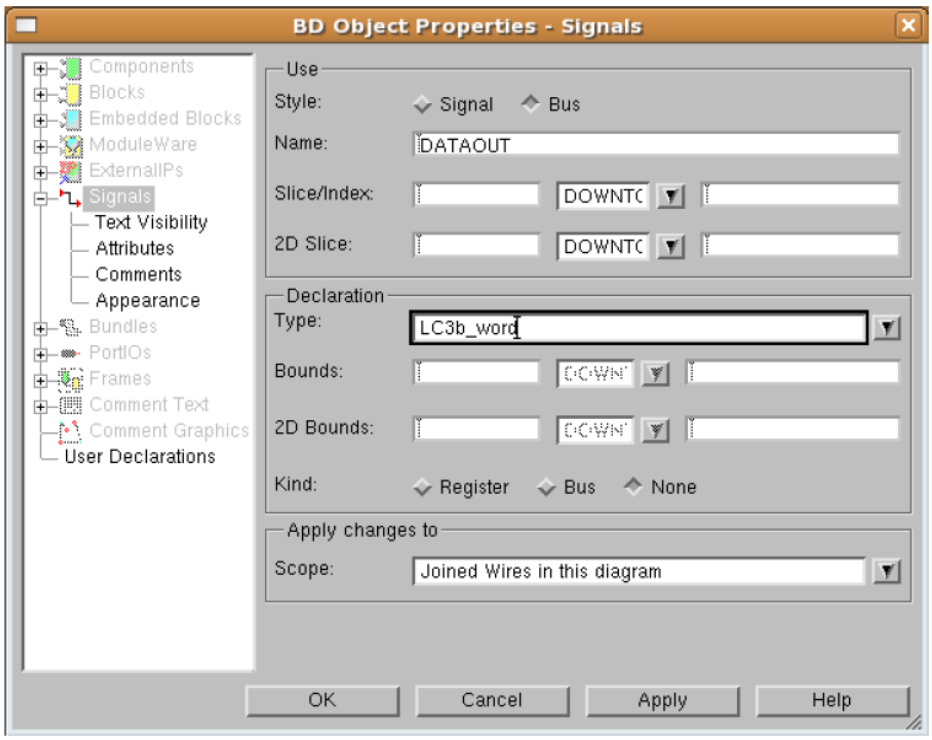


Figure 2: The block diagram Object Properties dialog

If you examine the ece411package.vhd file, you will see the line "`SUBTYPE LC3b_word IS std_logic_vector(15 downto 0);`". LC3b_word is the type of signal that the ADDRESS and the two DATA buses need to be. For each of the three buses, type "`LC3b_word`" for the Type of signal and delete any values within the "Bounds" fields. The window should now look similar to **Figure 2**. Hit **OK** after you edit each signal.

BEWARE: None of your signals should be of bare *std_logic_vector* types. If the tool suggests converting a *signal* to *std_logic_vector*, it means you didn't do step 4.2 correctly. Please consult a TA.

Save when you are done naming the signals. Remember: save early, save often. FPGA Advantage is known to crash at times. You have previously saved the diagram so you should not be prompted for library and design unit names. However, you have changed the names of signals connected to input and output ports, so the block diagram may be inconsistent with the symbol that was automatically created by the previous save. You may be prompted whether to update the symbol. If so, click the Yes button to confirm.

4.3.5. Add a Bundle

Select **Add → Bundle** then add two bundles connecting the datapath block to the control block, one in each direction. Bundles hold groups of signals or buses. The operation of the contained signals/buses is not affected by being in a bundle, but the block diagram looks cleaner. These two bundles are used to control the operations performed by the datapath.

Name the bundle coming out of the Control block “Control_out" and the other bundle “Control_feedback" as shown in **Figure 4**. Ignore the clock component for the time being.

Double-click on the Control out bundle to bring up the BD Object Properties dialog box. You will need to add signals and buses to this bundle. To add an item to a bundle, click the Create Signal button and enter its name and declaration info in the dialog box. For example, you need a bus called “ALUop." Enter “ALUop" in the name field, with Type “LC3b_aluop". Make sure Style is set to *Bus*. Click **OK** to add the signal to the bundle. The dialog should now resemble **Figure 3**. A list of the all the signals you need to add to the two bundles can be found in **Section F.1**. You will probably have to reposition components when you are done, as the signal list will be much longer than before. Also, hide the second line of text for the bundles (containing all the signals) so that it doesn't scale your printouts smaller. Select the text and right-click on it to choose **Hide Text**.

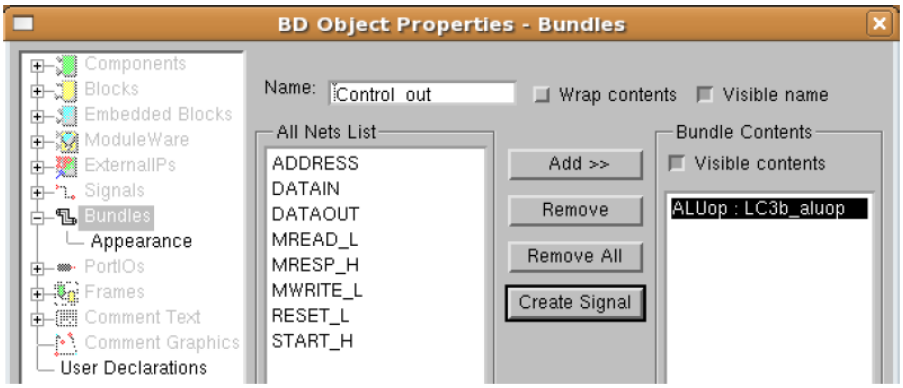


Figure 3: The bundle Object Properties dialog

4.3.6. Add Notes to the Block Diagram

Complete the block diagram by adding text notes describing the diagram. You should use the template provided on your block diagram for these notes. Fields in the template can be edited by double-clicking them. Save the block diagram. At this point, your block diagram should look similar to **Figure 4**.

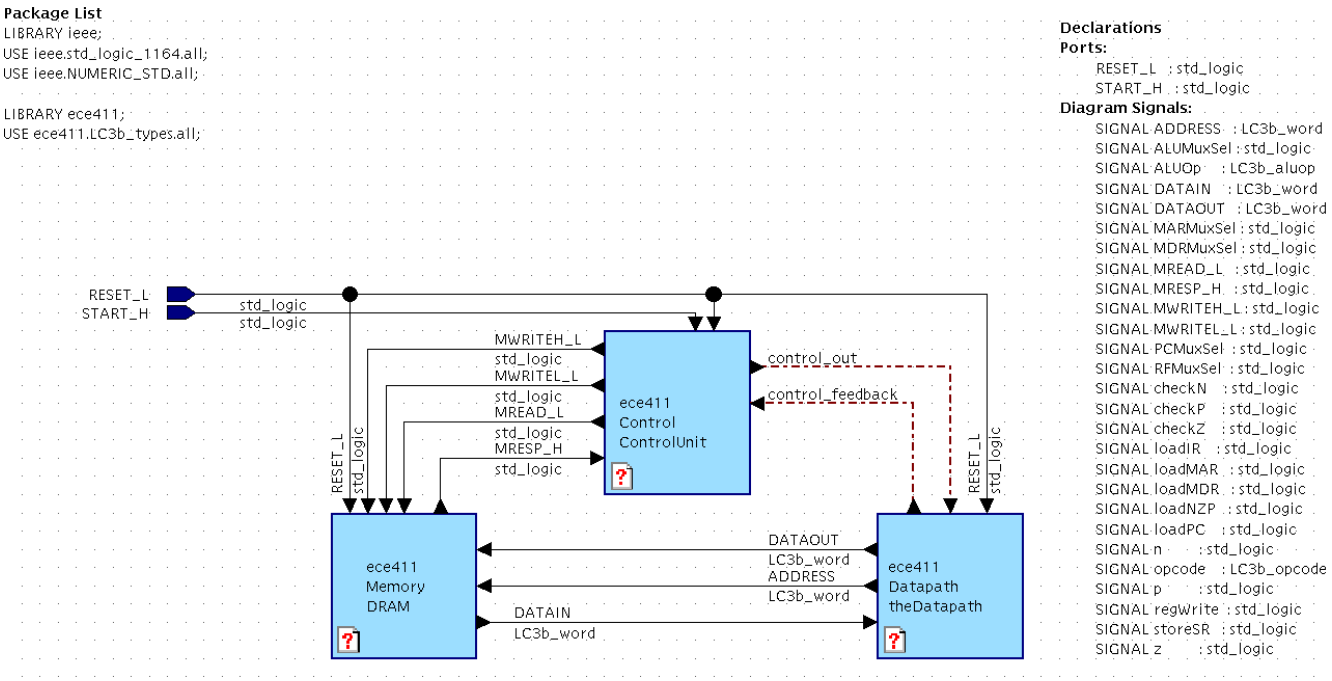


Figure 4: The completed CPU block diagram

Select **Tasks → Generate → Run Single** from the menu bar. You must make sure that no components are highlighted before executing this command, or you will generate HDL for only that component, not the CPU. If you accidentally had any component selected, unselect in and run the command again.

The progress of the HDL generation is monitored in a log window that includes any errors and warnings issued during generation. There should be (hopefully) no errors or warnings.

If you are successful, the log window should resemble **Figure 5**. If you have problems with unrecognized types, make sure that `ece411.LC3b_types` is included in the package list.

If there are any errors, you can display the corresponding graphics by double-clicking on the error message in the log window. If you have errors, this means that something is incorrect. Go over your design to make sure you have followed all the steps correctly.

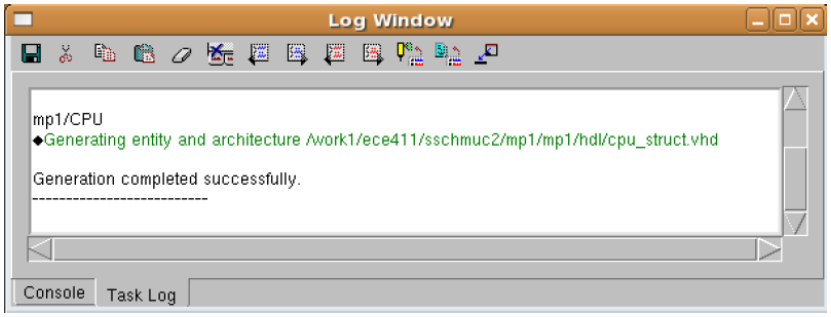


Figure 5: The log window after successful generation

4.3.7. Creating New Block Diagrams

Please reference **Section B** in the appendix for information on creating a block diagram.

4.4. Component Creation

Next, we will be creating several components for use in the Datapath. A component, once created, can be instantiated as many times as desired. For you C++ programmers, a good analogy might be a class declaration.

We'll be creating three components for this part of the MP: a 16-bit 2-to-1 multiplexer, a 16-bit register, and a clock generator. You create a new component by selecting **File → New → Design Content**, choosing the “Graphical View" category and the “Interface" file type, and clicking “Finish."

You can do this from almost any window; for now, the Design Manager might be a good place. In the window that appears, select “Symbol" from the “Structure Navigator" on the right side. You should see a window with a green block. This block will be the symbol that will represent your component each time it is instantiated. Change the shape of the block by right-clicking and selecting **Autoshapes. . .** from the popup menu. Choose “Mux" as the shape. Next, make sure to examine the Package List to verify the presence of the LC3b_types package.

Now would be a good time to save the component. Select File → Save. Select “ece411" as the Library, and enter “WordMux2" as the Design Unit. Leave other fields unedited. After you have saved, you'll notice that the symbol is updated with the library name and Design Name you chose.

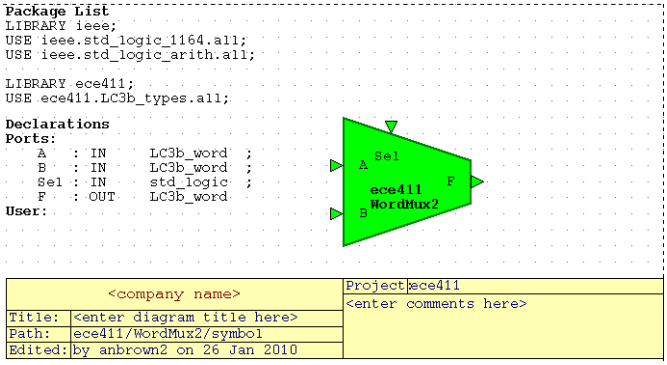


Figure 6: The diagram for the WordMux2 component

Add two input ports to the left side of the symbol, one input port to the top of the symbol, and one output port on the right side of the symbol. (The toolbar buttons for the ports are green triangles.) Double-clicking a port enters the “Interface" window that lets you change the port's name and type. The left ports are named “`A`" and “`B`". The top port is named “`Sel`" and the right port is named “`F`". The ports on the left and right sides should all be of type “`LC3b_word`" (no constraints), and the top port should be of type “`std_logic`". At this point your diagram should resemble **Figure 6**. Remember will reuse this component throughout your design, so make sure to create a clean looking symbol that is not too big or cluttered with text.

You have now created a symbol, but you have not yet defined how the component behaves. We will define the component's behavior in VHDL. To do this, double-click the symbol. A Dialog Box will appear titled “Open Down Create New View". Select “VHDL File/Combined" as the type of view to create and hit Next. In the Architecture: field, type “untitled". When you click Finish, a VHDL template should open up in the text editor you selected in **Section K.1**. If you have already created a VHDL file and you are attempting to edit it, you may receive a popup window asking how to open the file. You will need to select Text View from the drop down menu in place of Default View. Notice that the ports you set up earlier are listed in the template, as well as the LC3b_types library. Unless you are very sure of what you are doing, you shouldn't edit any part of the template above the start of the ARCHITECTURE section.

The part of the template you should edit begins with the line

```
ARCHITECTURE untitled OF WordMux2 IS
```

You should edit this to match the text contained in **Section H.2**. After editing, save and close the editor.

Now you have gone through the process of creating a component. Create another component in the ece411 library and name it Reg16. Its symbol may remain just a box. Add the ports listed in **Section H.1**, and edit its VHDL to match that in **Section H.2**. A suggested layout for the symbol is shown in **Figure 7**. Note that in this figure, the clk input has been marked as “Clocked", and the reset port has been marked as “Not". These options can be located by right-clicking the ports and modifying the “Clock" or “Not" menu options. Setting either of these options affects only the appearance, not the functionality.

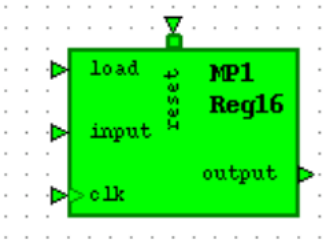


Figure 7: Suggested symbol for the Reg16 component

Finally, create another component in the ece411 library and name it CLKgen. Its symbol may remain just a box as Reg16. *CLKgen* will have only 1 port of type INOUT name *clk.* Once you have created the symbol such as the layout in **Figure 8**, double click it to enter vhdl code just as you did before with Mux16 and Reg16. This time, copy the entire contents from the /given/mp1/given_CLKgen.vhd. Note that line *clk : INOUT std_logic := '0'* sets an initial value to the variable clk. Omitting this initial value will result in an undefined clock in your design.

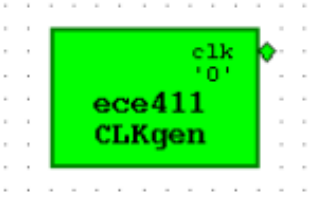


Figure 8: Suggested symbol for the CLKgen component

4.4.1. Add CLKgen to your CPU

Now that we have a basic CPU diagram, we need to add out CLKgen component to the top level design so every underlying component or block runs off the same clock. Open your *mp1_cpu* block diagram, then select **Add → Component**. In the Component Browser window, drag the *CLKgen* component onto your *mp1_cpu* diagram.

Next, select **Add → Global Connector** to add a global connector on your diagram. A global connector will implicitly connect the signal attached to it to every block on your diagram. Add a signal between the global connector and the output of your *CLKgen*. Name this signal *clk* if it does not default to that. This is your global clock that your Memory, Control, and Datapath will all use. Your CPU should look like **Figure 9**.

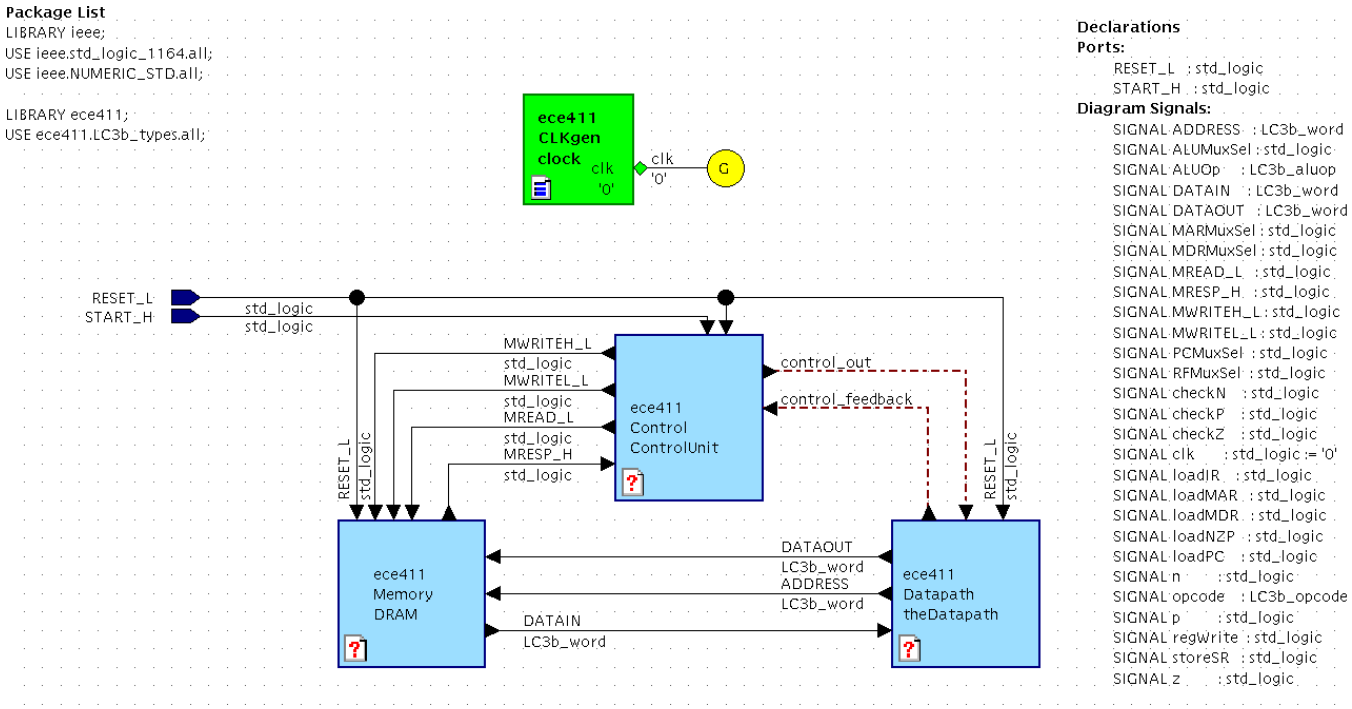


Figure 9: CPU Block Diagram with Clock Generator

4.4.2. Datapath

Next, you will need to design the Datapath (sometimes referred to as “data path"). Double-click on the Datapath block in the MP1_CPU view to create a new block diagram. The 'Open Down Create New View' window will pop up. Select 'Block Diagram' under 'Graphical Views' and click Next. Leave the default view name unchanged (struct.bd) and click Finish. The design window for the Datapath will open. You should create new blocks, components, and wires to make the datapath be effectively the same as the one described in **Section I** and shown in **Figure 22**. The datapath displayed in this document has been laid out to be compact and fit nicely on a page. Your layout need not be exactly identical, though that may help when you are double-checking that you entered everything. Be sure to include the correct names of blocks and signals! **Section I** also contains lists of blocks and signals for you to use as a checklist. Note that U_0, etc. are the unique instance names used by the program to keep track of blocks; these should be changed to descriptive names that reference what the block actual does . Finally, see **Section 4.6** for information on VHDL to be used for the blocks on the datapath diagram.

For example, you might want to call the Reg16 component whose function is to hold the PC value “PC" and the WordMux2 component that feeds into that Reg16 “PCMux".

Blocks in green are instances of the components you should have created in **Section 4.3.7**.

The yellow block (upper left corner) is an Embedded Block. It is used to join the DATAOUT and MDRout buses. While any type of VHDL statements can be contained in embedded blocks, we recommend you only use them to join wires/buses having different names but the same type.

You can change the size of the blocks by clicking once on the block, then using the solid handles to resize. Similarly, you can reposition the titles of the blocks. To edit the shape of a block, right-click the block and select **Shape → Autoshapes. . .** from the pop-up menu. You can then select from a list of many standard shapes. If you do not want to route a signal or bus over a very long distance, you can start a signal/bus in empty space and name it EXACTLY the same as the signal/bus you wish to connect to. The program will ask you if this is what you want to do. Click **OK**. You can also set it so that the program automatically associates it with the other signal.

As a convention, all std_logic type wires should be signals and all others should be buses. You may hide the Type information on signals. (You can hide text using the right-click pop-up menu.) It is recommended to not hide signal types until after you are done with all changes.

When you are done with the Datapath diagram, save it and generate to check for syntax errors. VHDL to describe each blue block will be created in Section 4.6.

4.5. Create a State Machine (Control)

The control unit for this design will be created using a Moore state machine. For the non-pipelined implementation of LC-3b, this is an effective way of showing the actual function of the design.

4.5.1. Create a State Diagram

Double-click the *Control* block on the *mp1_CPU* block diagram. The Open Down Create New View dialog box is displayed.

Select “State Diagram" from the list of Graphical views given and click Next. Keep the default view name **fsm.sm** and click Finish. A new state diagram (**ece411/Control/fsm**) is then created as a child view of the *Control* block. The state diagram is a blank sheet except for the default VHDL package list (which should include **ece411.LC3b_types.all**), labels for global actions, concurrent statements, architecture declarations and signals status. The clock and reset signals should also appear graphically. If the package list is incomplete, edit it to add the required packages.

4.5.2. Set HDL Generation Characteristics

In the Control state diagram window select **Diagram → State Machine Properties. . .** .
A new dialog box will appear; select “Generation" from the left column. Choose the *Synchronous* button in the Machine box and click **OK** to exit.

Now double-click on the yellow icon corresponding to the clock signal. This will bring up the “SM Object Properties" window for the clock signal. Verify that the Name field is set to clk and that the Edge field is set to **Rising**. Next double-click on the icon corresponding to the reset signal. Verify that the reset mode is set to “Asynchronous," that the Reset name is **RESET_L** and that the Reset Level is **Low**. Your state machine should resemble **Figure 10**. Save the state diagram.

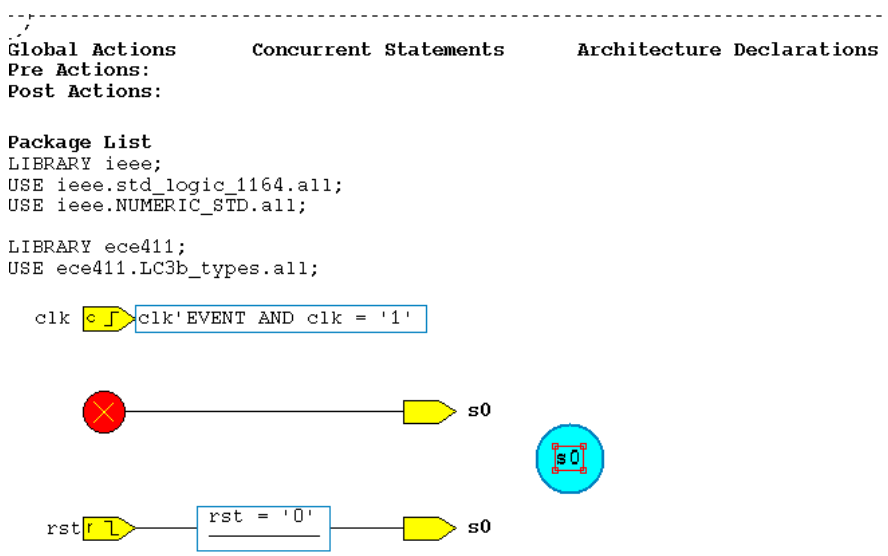


Figure 10: Basic State Machine Settings

4.5.3. Add States and Transitions

Select **Add → State** then add eight (8) more states on your state diagram in addition to the start state s0 which was created by default. The states are added with default names s1, s2, etc.

Select **Add → Transition** to add transitions between the states as shown in **Figure 11**. When adding transitions, first click on the state you wish to start from, then click on the edge of the state you wish to go to. Notice that when you add more than one transition leaving a state, a number associated with the transition arc indicates the transition priority. The priorities for the transitions are initially assigned in the order that you add the transitions. If you want a “default" transition (a transition taken when no other transitions' conditions are satisfied) it must have the lowest priority (highest number) of all the transitions leaving that state. This is because transitions are checked by increasing priority number, and if a transition has no condition, it is always taken. We will discuss editing transitions in a few pages.

Note: If you add a transition in the wrong direction, you can easily change its direction by choosing **Reverse Direction** from the pop-up or **Diagram** menu.

Next, select **Add → Link** to create a link to the IFetch state. You will need to return to the IFetch state after executing an instruction, so that you can fetch the next instruction from

memory. Here, you will need links to IFetch from all the states except Reset, IFetch, and Decode. An example is shown below. You can rotate the link by right clicking on the link and selecting **Rotate**.

Your state diagram should now look similar to **Figure 11**, although state colors may be a different color than yellow. Name your states exactly as shown. Remember that default transitions must have the lowest priority (highest number, in this case 6) of the transitions leaving that state.

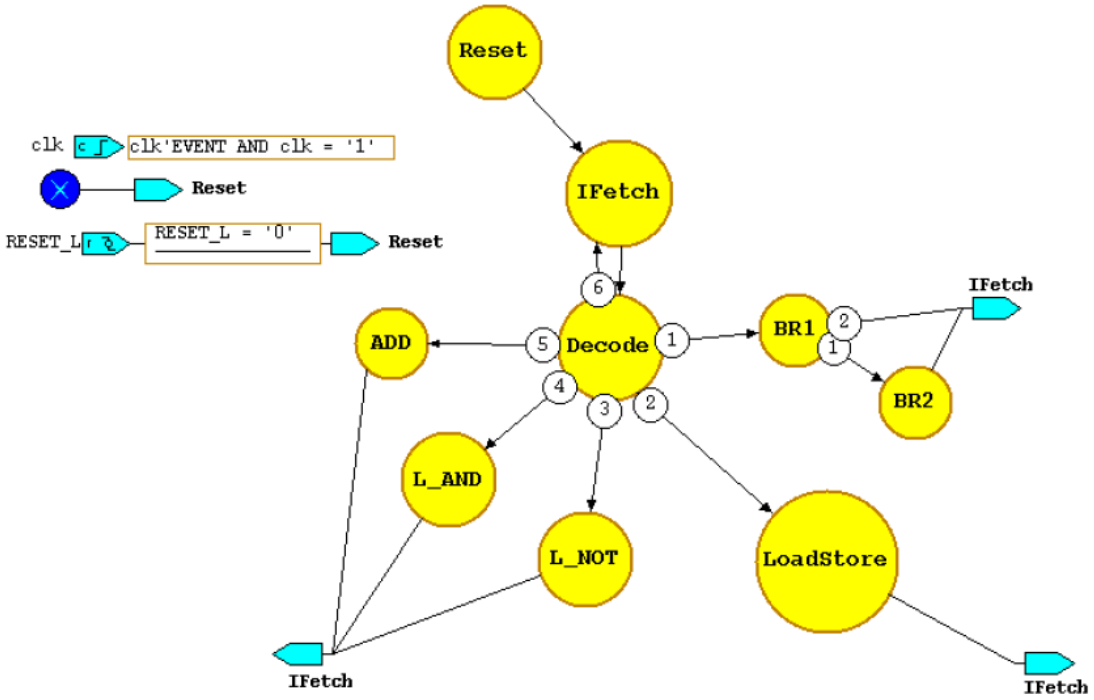


Figure 11: New states drawn in the state machine

4.5.4. Saving the State Diagram

Save the state diagram. The state diagram was created as a child view from its parent block diagram and is saved using the library, design unit and view names specified when it was created. The Design Manager view is updated to display the Control design unit.

Note: You can pop the Design Manager window to the front by selecting it from the Window menu list in an editor window.

The Control design unit is shown as a block in the browser because its interface is defined by the connections on its parent block diagram. The mp1_CPU design unit is shown as a component because it has no parent block diagram as its interface is defined by a symbol. Expand the Control design unit to reveal that it contains a state diagram view.

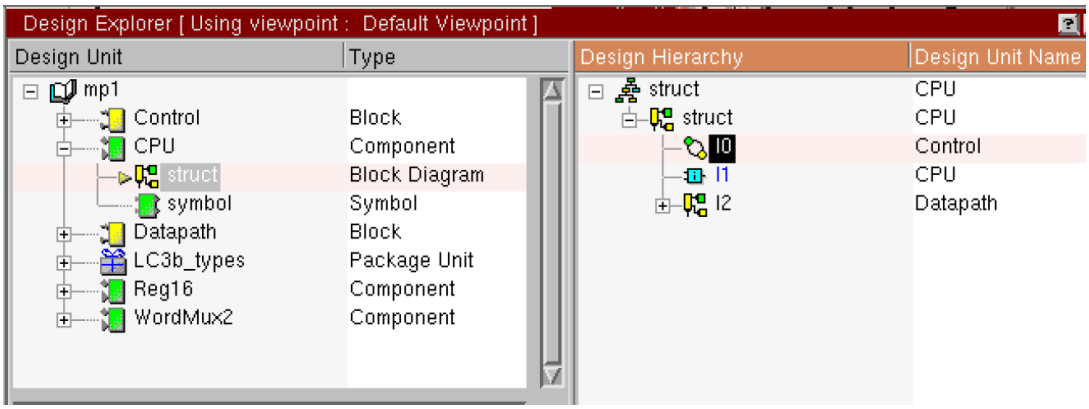


Figure 12: The Design Manager after creating the state machine

Notice also that the icon used for the Control unit in the hierarchy for the struct view has changed, indicating that it is now described by a state diagram. If the hierarchy is not already displayed, right-click on struct and select “Show Hierarchy".

4.5.5. Edit the States

Right-click the IFetch state and select **Object Properties. . .** to display the “States" section of the State Machine Object Properties dialog box. This window allows you to enter a name and actions text for one or more selected states on a state diagram. You can also change the visibility of state actions and (when a single state is selected) change the state to a hierarchical state. Under the *State Actions* tab, you can enter things that the processor should be doing during that state, such as selecting multiplexer inputs or enabling register loads.

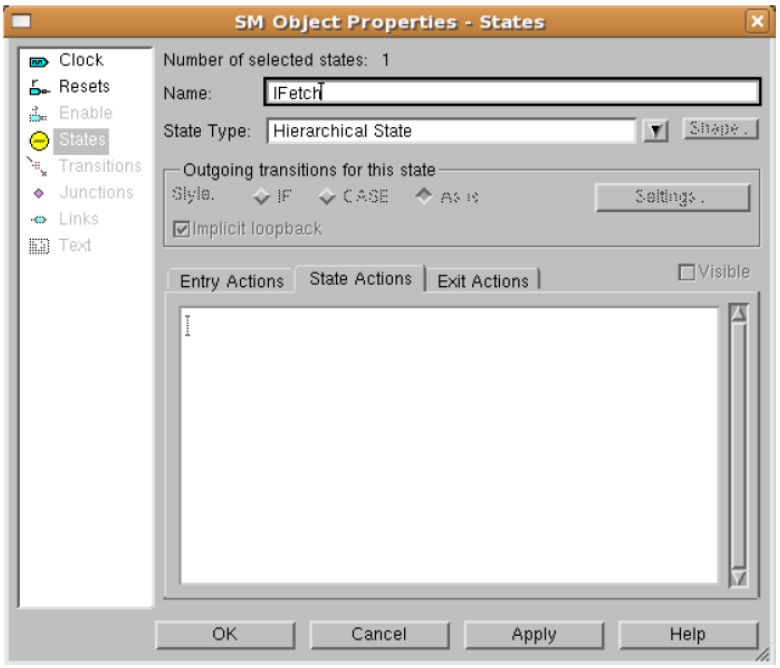


Figure 13: The Object Properties dialog for a state

Use the dialog box to make the *IFetch* and *LoadStore* states hierarchical. They should be redrawn with a triple outline and a darker blue color. You will be modifying the actions of the states at a later time. (This option is found under the “State Type" drop-down list.)

Notes before we continue:

1) You can also edit the state name or actions by direct text editing on the diagram.
2) If the new name is larger than the state, it auto-resizes to _t the new name. You can also resize a state by selecting the state and dragging one of its resize handles, but you cannot make it smaller than the enclosed name.
3) The syntax for state actions is automatically checked and any errors reported on entry.
4) You can enter any valid VHDL statements, which must be terminated by a semicolon (;) character. Line breaks and spaces can be used for clarity and will be preserved on the diagram.
5) Single bits must have single quotes around them, i.e. `0'. Multiple-bit patterns must have double quotes, i.e. “00".
6) You can add route points while routing a transition by clicking at several points between states to create a smooth arc.

4.5.6. Editing Transitions

A state machine isn't of much use if the processor does not know which transition to take or when to take it. Double-click on the transition from RESET to IFetch. It will bring up a window like that in **Figure 14**.

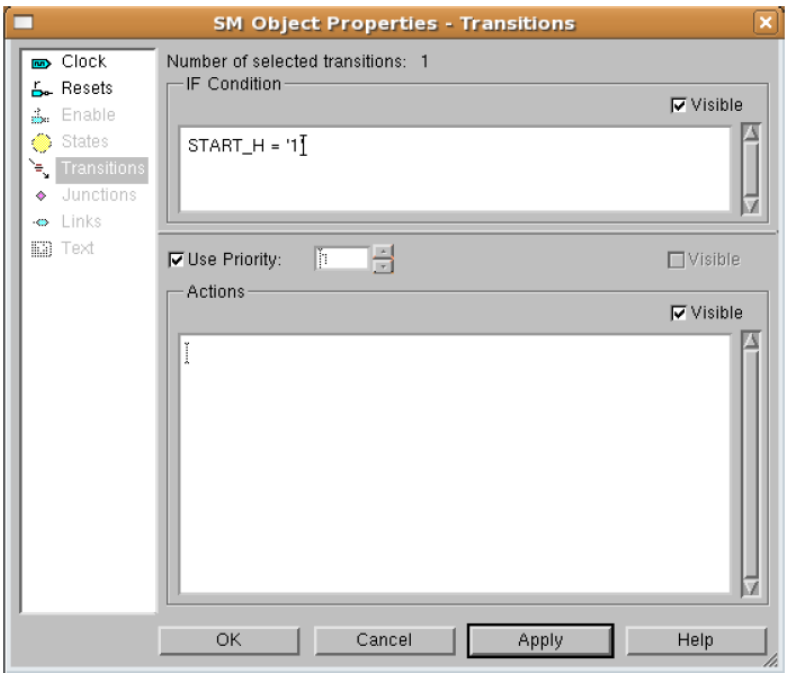


Figure 14: A transition Object Properties dialog

In this dialog, you can modify both conditions (when to take that transition) and actions (what to do during the branch). *We will not be modifying actions for transitions in this class.*

For this transition, your condition should be `START_H = '1'`. This means that the processor should remain in the reset state until the `START_H` signal is high. Note that there is no semicolon after conditions, only after actions.

4.5.7. Creating a Hierarchical State

Hierarchical states are used to further subdivide a state diagram into smaller pieces, making it easier to read and understand. They are not real states, but contain a series of states that save space and help compartmentalize concepts. Before continuing, save your changes to the state diagram.

You should have previously made the *IFetch* state hierarchical. You can select the state and choose **Diagram → Change To → Hierarchical State** if you haven't already made the change.

Double-click on the *IFetch* state to create the new hierarchical child state diagram. A new child state diagram window is initialized with a single state connected to an entry point and exit point. Notice that the child diagram is named similar to MP1/Control/fsm[:IFetch] indicating that it is a partial view of the parent diagram.

4.5.8. Edit the Hierarchical State Diagram

Create two more states on the diagram. Rename the states and connect transitions between the states as shown in **Section J.1**. Remember to save the state diagram when you are finished!

Although it is displayed as a separate window, a child hierarchical diagram is a partial view of a state machine and the parent and child diagrams are saved as a single design unit view (**ece411/Control/fsm.sm**), although the name in the title bar is **ece411/Control/fsm:IFetch**.

4.5.9. Hide Global Actions and Concurrent Statements

Choose **File → Open → Open Up** to redisplay the main Control state diagram if it is not already displayed. Double-click the “Concurrent Statements" label in the window to display the “Statement Blocks" tab of the State Machine Properties dialog box.

There are no global actions or concurrent statements in this design. What you should do (to reduce clutter) is hide the labels for these objects on the state diagram by clearing the “Visible" checkboxes for the three items on this tab. Hit **OK** when you are done.

4.5.10. Complete the State Machine

Now that you have placed and named the states for the state machine, you will need to complete it. If you try generating the state machine, errors will pop up since no conditions have been set to the transitions in the diagram. **Section J** contains the information you need to complete the control unit.

For actions, as mentioned before, you will need to include terminating semi-colons. This is because VHDL requires terminating semi-colons.

You will also need to edit the default values of certain signals. A default value is the value that an output signal automatically has when you do not explicitly assign it a value in a state. This happens for each and every state. It is particularly important to not have default values of `0' on the active-low signals to Memory! To change default values, double-click on **Signal Status** in the design view, which will bring up the Signals Table. You will need to change the default values of four of the signals. The signals and their default values are listed in **Section G.1**.

4.6. VHDL

You have now completed the initial phase of the design of our LC-3bα processor. What you have in the computer is a very rough layout of the basic structure of the processor. However, it is far from complete. You will need to input a considerable amount of modeling code to get your design to a functional state.

4.6.1. Input HDL

Thankfully, we have provided most of the HDL you require in the ece411_given.tar file. These files are located in the folder ece411_given/given/mp1. All you will need to do is copy and paste the code into the various datapath blocks and the memory block as described below. However, you will need to write your own HDL code for the blocks *ADJ9*, *StoreMux*, and *BRAdd* using the provided code as a guide.

Double-click over the body of the Memory block in the MP1_CPU view to display the Open Down Create New View dialog box. Select 'Combined' under VHDL files from the list and click **Next**. You should type 'untitled' in the 'Architecture' field and click **Finish**.

A template HDL view is opened using the default text editor for your workstation. You need to modify this template by adding the VHDL we have provided for you. The VHDL for the memory block is named `given_memory.vhd`. In the template, the ARCHITECTURE section should be replaced by the contents of the provided file. Save and close the HDL view.

Now do the same with all of the blocks in the Datapath. Our provided VHDL files have names which similarly correspond with the blocks in the Datapath.

You should now be completely done with the design. To check for errors, select the highest-level view in the Design Manager, which is the *MP1_CPU* box, and select **Task → Generate → Run Through Components**. Many errors are relatively well explained by the program. As always, if you have excessive difficulty isolating an error, please consult the FAQ, check the webboard, or see a TA.

5. Compiling and Simulation

5.1. Compile

Select the *mp1_CPU* design unit in the Design Manager and choose **Task → Generate → Run Through Components**. The log window will appear and generate HDL for all components of your design. If you have error messages, check your design and resolve any errors before regenerating. Most errors are explained in the log window, and give some idea of how to fix the error. If you cannot find the error quickly, ask your fellow classmates, check the FAQ, or ask a TA for assistance. Continue when you can generate without error messages.

Next, make sure the *mp1_CPU* design unit is still selected, and choose **Task → Modelsim-flow → Run Through Components**. Again, if you have any error messages check your design and resolve any errors. You will need to regenerate HDL before recompiling.

Note: FPGA Advantage's method of tracking files sometimes misses the changes you make to diagrams and VHDL. You may need to exit the simulator and restart it to resolve changes not being detected. Also, we've put a few errors into **Section D** to help you get started: you can list your own in there as you find them, to help you during future MPs.

5.2. Simulation

5.2.1. Using ModelSim

Once the design is compiled without errors, the simulator is invoked. A new dialog box will appear as shown in **Figure 15**.

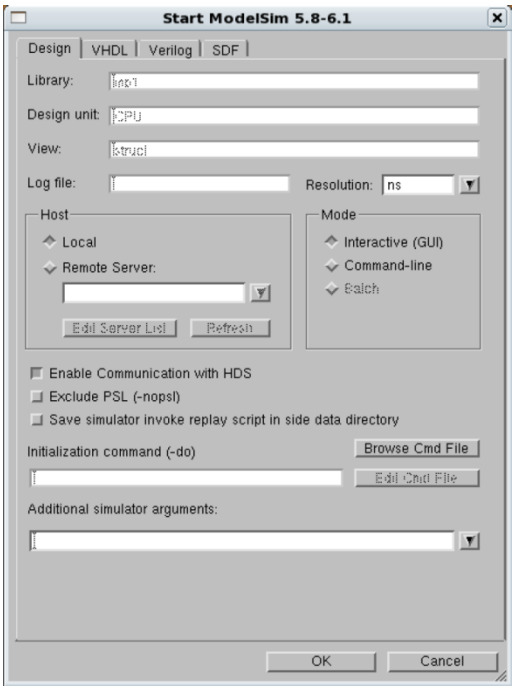


Figure 15: The Start ModelSim window

Verify that the ece411 library is designated and the resolution of the simulator is set at nanoseconds. Click **OK** when you are ready. The log menu will appear and invoke ModelSim. ModelSim will appear in its own window and issue loading messages for all of your components, ending with a prompt like "`VSIM 1>`". You are now ready to use ModelSim.

5.2.2. Changing the View to Hexadecimal

When printing out waveforms and lists, you will need all your signals to be displayed in hexadecimal. To set ModelSim to always display your signals in hexadecimal, select **Simulate → Runtime Options...** Under **Default Radix**, select **Hexadecimal**.

5.2.3. Change to a Fixed Width Font

ModelSim's default font is not fixed width, which makes it default to compare and read out signals. To change your default font, select **Tools → Edit Preferences…** Then, under the **Window List** toolbar, select **Wave Windows**. Within the Font selection, click Choose and select ***courier*** as your **Font**, ***regular*** as your **Font style**, and ***9*** as your **Size**.

5.2.4. Force Files and Other Macros

Force files are used in ModelSim to help set parameters for and debug designs. They are one of several macros (files containing multiple instructions) that can be of use during simulation. To execute macros, copy the macro into your *compile* directory, and type **do macro_filename** in the ModelSim window, where **macro_filename** is the name of the macro file.

5.2.5. Wave Traces and Lists

There are multiple ways of viewing the functionality of your design. The two methods we will be using in ece411 are wave traces and lists.

5.2.5.1. Wave Traces

In the ModelSim window select **File → New → Window → Wave**. Alternatively, you can type `view wave` at the prompt.

Drag the "datapath" item from the ModelSim window to the wave window. In the wave window, note that the LC3b_word values are displayed as 16-bit values, which can be difficult to read. If you have not set your signals to display in hexadecimal above, you can change them temporarily by selecting (in the wave window) **Edit → Select All** then **Format → Radix → Hexadecimal**. This will make the diagram easier to read. You can also reorder signals by dragging them to new locations on the diagram.

You will need to obtain access to some of the other signals in the design. To do this, select the appropriate structure within the left panel of ModelSim. The "signals" window should update to show the signals available in that structure.

In order to save time when you reenter the program, you can save the signal list displayed in the wave window. To do this, select **File → Save. . .** . The default filename is `wave.do`: save this. In the future, you can type `do wave.do` in the ModelSim window and the signals will appear in the wave window.

5.2.5.2. Lists

In addition to wave traces, we will also use lists within ece411 to easily examine what is being written to our memory system. In the ModelSim window select **File → New → Window → List**.

Now, drag the `address`, `dataout`, `mwritel_l`, and `mwriteh_l` signals to the list window. Change the signal properties (select the signal name then pick **View → Properties. . .**) so that all values are in hexadecimal if necessary.

`address` is the location *where* are writing to memory, `dataout` is *what* we write to memory, and `mwriteh_l` and `mwritel_l` is *when* we write to memory. By default, each time a signal in the list window changes, it generates a new entry in the list. For some signals, you may not want a new line every time its value changes. In this case, we only want our list to generate entries when are actually writing to our memory (when `mwriteh_l` or `mwritel_l`) transitions from inactive[1] to active[0]). There, we only want to trigger entries to be added to our list when `mwriteh_l` or `mwritel_l` changes. To accomplish this, select the `address` and `dataout` signals, choose **View → Properties. . .** , and select "Does not trigger line" under the **Trigger** box. You should have something similar to **Figure 16**. Note that you will have two mwrite signals, rather than a single mwrite signal.

5.2.6. Testing Your Design

With you waveform and lists created, you are now ready to run a simulation of the design. You will need to load the provided LC3b assembly file (mp1test.asm in your /testcode folder) into your design. Please see **Appendix A** for information on loading a program into your design. You will need a 50 ns clock cycle specified in your .do file. Once you have the correct .do file in your compile directory, type `do mp1test.do` in the ModelSim command prompt. This file is located in your compile directory (it was provided in the ece411_given.tar) and it has commands that establish a 50 ns clock and the initialization pattern for the processor. You will need to run this file every time you start or restart the simulator. If you receive an error such as "Object '/mp1_cpu/dram/vhdl_memory/mem(0)' not found', please check the instance names of your blocks. See the FAQ in **Section 8** for more details.

In the ModelSim window type `run 2000`. This will run the design for 2000 ns. Your wave window should look similar to **Figure 17**. Please note that this figure may or may not contain any correct wave values. You should not use its values as a reference to whether your design is working or not.

You can ignore the majority of the warnings in the ModelSim window. Most of these are due to undefined values that the simulation program attempts to perform operations on. If you encounter problems with the processor not initializing properly or you would like to add signals to your waveform, you can restart the simulation by typing `restart -f` in the ModelSim window. You must then rerun the `*.do` and `run` commands. For the handin for this MP however, please follow **Section 6**.

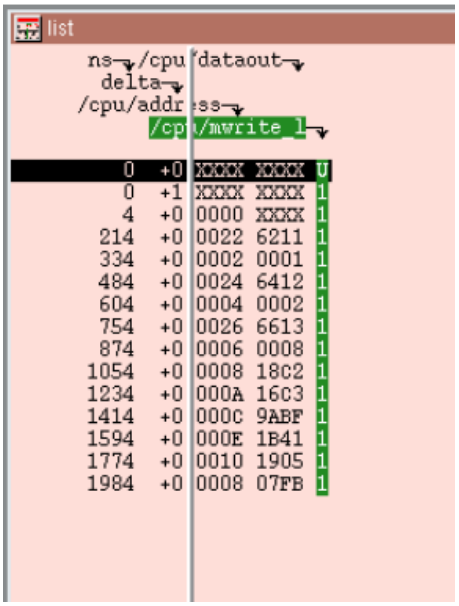


Figure 16: The list window

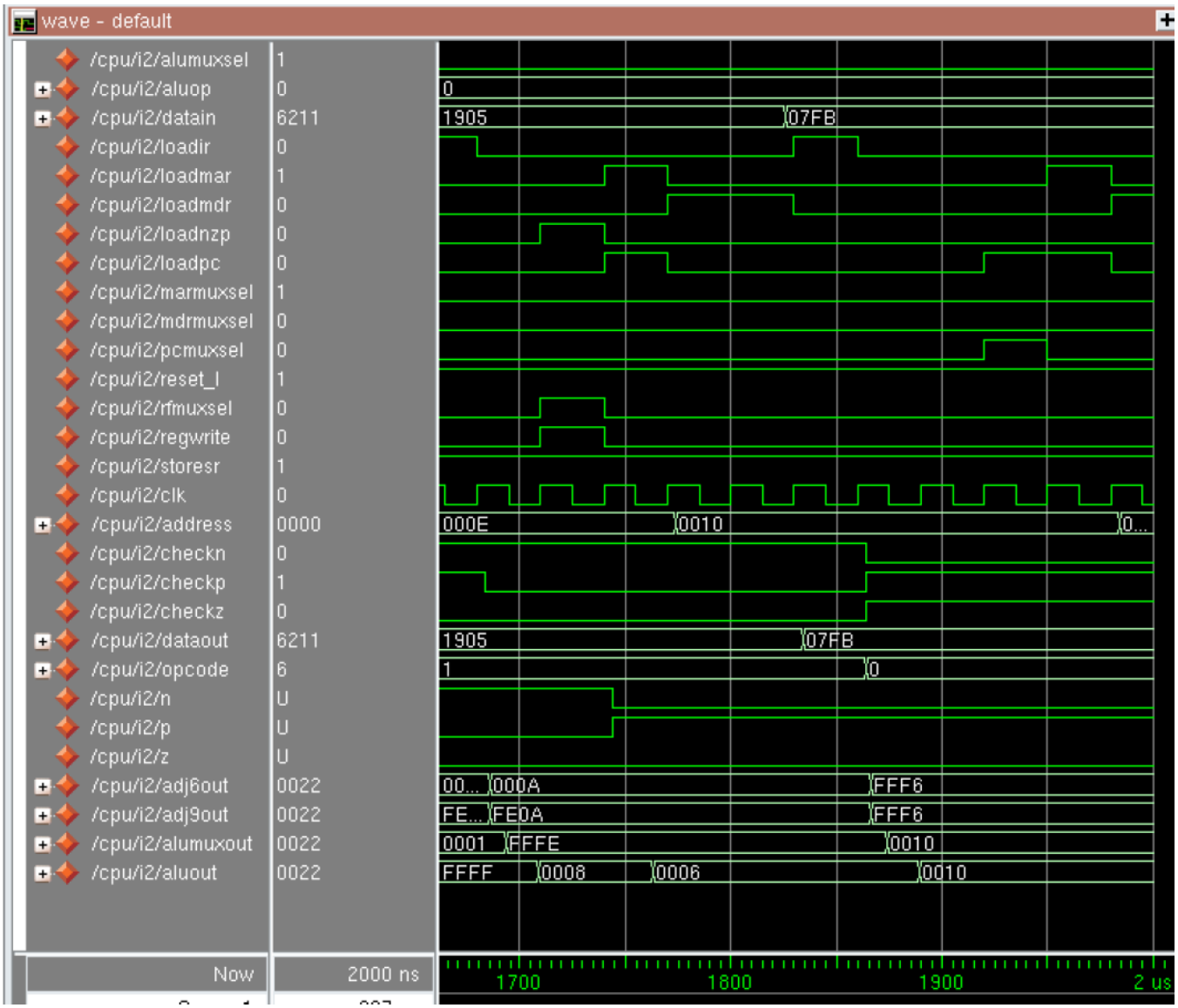


Figure 17: The wave window after running for 2000 ns

6. Final Hand-In

You will be required to write a short program using the LC-3b assembly language. The program should calculate 5! (five factorial). Your program must be iterative. It does not need to account for negative numbers or 0, although it can if you like. **Note that your datapath does not yet contain support for immediate adds**, so you must use load instructions to initialize registers. Reference the sample program located in the *given* directory of the *ece411_given.tar* for assistance with example instructions you can use. The factorial program should be easily expanded to calculate 6!, 7!, 8!, etc by changing only one variable. It does not have to handle 0! or negative factorials. Like the sample program, your code must end in an infinite loop. This will make simulation a lot easier. Please see **Appendix A** for a description of how to load a program into your processor.

You need to hand in the following items to the ECE411 Drop Box in the basement of Everitt (or a TA's mailbox if the drop box sign has gone missing):

1) A **commented** printout of your assembly program. Comments should be more than an explanation of what a particular instruction does.
2) A wave trace showing the first 1000ns of your program's operation. Make sure the signals below are present on your wave with data. If you add a signal after running a simulation, it will not show data.
3) A wave trace showing from 1000ns before your infinite loop until the first iteration of the infinite loop (The PC must be shown to repeat). Highlight the register which contains the value of five factorial.

Notes:
Each wavetrace must contain the following signals in the order listed below:
`RESET_L, START_H, clk, PCout, ADDRESS, MWRITEH_L, MWRITEL_L MREAD_L, DATAIN, DATAOUT, RegFile Contents.`

"RegFile Contents" refers to the contents of each register. Please expand this signal so that each register is shown on its own line. Make sure all signal values are shown in hex! Changing to hex is covered earlier in this handout. If you are having difficulty understanding any aspect(s) of the MP, ask a TA for assistance.

Remember that material from this machine problem will be used in subsequent MPs and may appear on exams. It is your responsibility to make sure that MP1 is complete and functioning correctly before proceeding with future MPs.

WARNING: Be sure the wave trace actually has values on it, not just signal names. If you add signals to the wave after simulating you must re-run the simulation in order for the new values to be displayed.

7. Grading Rubric

Code: 12 points (40%)

Up to 8 points for correctness

Up to 4 points for good commenting (i.e. define what registers map to what values, etc.)

First 1000ns: 3 pts (10%)

-1 for not showing all 1000ns (or showing too much)

-1 not including all required signals (see MP1 pdf)

-3 if missing completely

Final 1000ns: 15 pts (50%)

-1 final answer not highlighted / circled

-1 not including all required signals

-1 for not showing 1000ns (or for unreadable signal values)

-6 infinite loop is not shown

-6 result is not correct (120 or 0x78)

-15 if missing completely

8. MP1 Frequently Asked Questions

- **I am having a problem where Mentor Graphics will not display text on the blocks. It just shows all the characters as vertical lines. It will allow you to type, but it just creates more vertical lines for each character.**
  This issue affects only a small portion of students. Zooming in or out usually solves this problem.

- **When should I select the type of a wire to be 'signal' and when should the type be 'bus'? Does this affect my design?**
  Selecting whether a wire is a signal or a bus only affects the visual representation of the wire within your design. Signals will be thin lines, while buses will be thicker lines; however, this does not affect the performance of your design at all.

- **Should my new logic block be created as a 'block' or a 'component'?**
  Blocks are used for logic that you will only use for that one specific instance, whereas components can be reused at will throughout your design. Blocks can be converted to components at any time, but components cannot be converted back into blacks.
  However, since you will reuse components from MP1 to MP2 to MP3, we recommend creating everything as a component. If you need two slightly different instances of a component, you can always make a copy of the component and make changes to the copy.

- **I am trying to simulate the program that I've written using the do file generated from doify.sh, but I get an error that says:**
  **** Error: (vsim-4008) Object '/home/engr/xxx/ece411/mp1/hdl/DRAM_untitled.vhd/mem(0)' not found.**
  You probably specified the path wrong when you ran doify.sh. When you open up ModelSim, there should be a hierarchical list of your components on the lefthand side like this:

  mp1_cpu
  -->dram
  ---->vhdl_memory
  -->control
  -->datapath

  It also might say "i0","i1","i2" if you did not rename your components. Anyways, this is the path you are looking for, so it should be something like below, depending on how you named your components of course:

  ```
  #> doify.sh myvhdlfile.vhd 30 /mp1_cpu/dram/vhdl_memory
  ```

A. Loading Programs Into Your Design

To load a program into your design, you need to generate a .do force file that contains a memory initialization file and place it in the compile directory of your design. We will use the doify.sh script located in your /bin folder to do this. Note that the doify.sh script references LC3bAssembler, so please make sure that executable is also located in /bin.

A.1. doify.sh

The doify.sh script automatically assembles your LC3b code, converts the generated .vhd file it into a .do file, and copies that file to your compile directory. This allows you to run multiple programs without the need to recompile your design. Using the doify.sh script located in the /bin folder, run the program as follows (type on a single line):

```
[netid@linux6 ece411/bin] ./doify.sh [.asm file] [memory
path[,second memory path]] [compile path]
```

For example, if you have a .asm file named test.asm, a memory path of '/mp1_cpu/dram/vhdl_memory', and a compile path at '$HOME/ece411/mp1/compile' run:

```
[netid@linux6 ece411/bin] doify.sh ../testcode/test.asm
/mp1_cpu/dram/vhdl_memory $HOME/ece411/mp1/compile
```

Note: the path to your memory will depend on the interface names specific to your design. *It is **not** a path in the filesystem!* Both the memory path and the compile path can be hardcoded by changing the default variables in the *doify.sh* file. Please read the comments of the script or ask a TA for assistance.

Now, when running ModelSim you can run your program as follows:

do wave.do	(open the wave file; you must create the wave file first)
restart –f	(restart any previous simulations)
do test.do	(loading the program and initialize your processor)
run 1000	(run the design for 1000 ns)

Note: these last three commands can be combined together to save typing as:

restart –f; do test.do; run 1000

B. Creating a New Block Diagram

B.1. Create a Block Diagram

In the Design Manager, select **File → New → Design Content**. The Design Content Creation Wizard will appear (Shown in **Figure 18**). Select the **Graphical View** category and the **Block Diagram** file type. Then click “Finish."A new untitled block diagram is created. Blocks are effectively “black boxes" whose function is hidden from you in the block diagram, although you will be specifying their functions soon. The block diagram is a blank sheet except for a background grid, a default package list (which you will need to edit) and a comment block.

Double-click the Package List (in the upper-left hand of the diagram, a list of libraries being used), and a Package List window will open. Here you can edit the package lists for a specific file. Click **OK** to commit the changes.

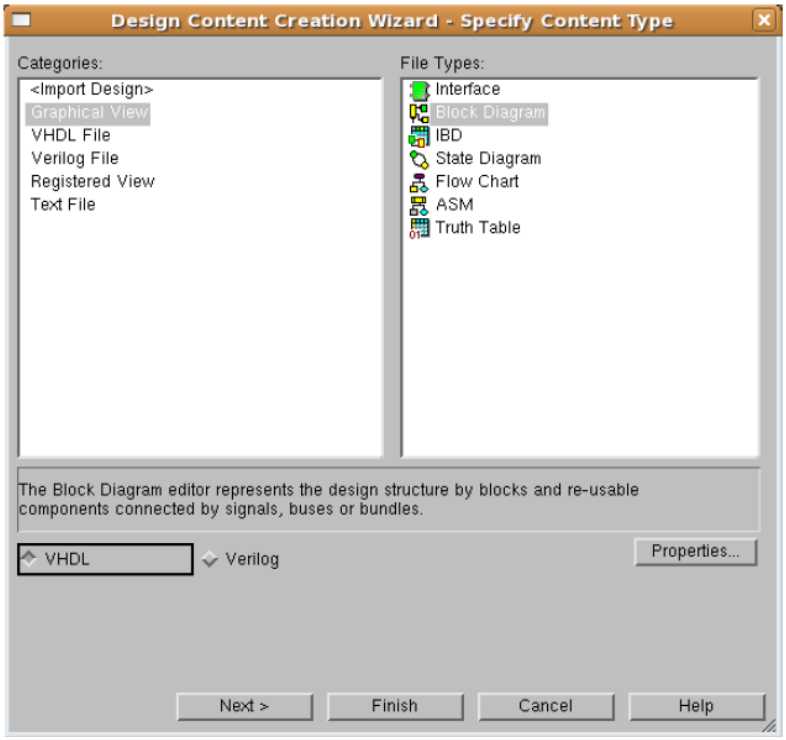


Figure 18: Design Content Creation Wizard - Making a New Block Diagram

B.2. Saving the Block Diagram

Note the asterisk (*) character in the title bar of the block diagram editor window. This indicates that the diagram has been edited since it was last saved.

Select **File → Save** to save the block diagram. The Save As Design Unit View dialog box is displayed which allows you to choose from the currently mapped libraries and specify the design unit and design unit view names. Choose the *ece411* library and enter the block

name in design unit. Do not change the default view name. The dialog box should now look similar to **Figure 19**.

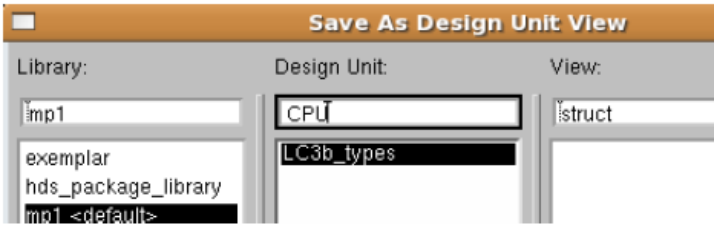


Figure 19: The "Save As Design Unit View" dialog box

When you click the **OK** button, your diagram is saved and the window title bar is updated to show the diagram name ece411/mp1_CPU/struct. Notice that the * character has been cleared in the block diagram header, the library name used on the blocks in your diagram has been updated to ece411, and the Design Manager view is updated to display the mp1_CPU design unit.

Click on the plus icon next to the CPU design unit in the Design Manager to expand the design unit. This reveals that it contains a symbol and block diagram view. Click on the plus icon for the struct view to display the blocks on the block diagram.

Note: A red cross superimposed on any icon in the Design Manager indicates that the view is not writeable. This convention is also used to show when you have read-only access. This will be of importance when you are sharing design units during group work.

C. Instruction Set Description

Name	Opcode	Description
add	0001	Put the sum of registers SR1 and SR2 into register DR
and	0101	Put the logical AND of registers SR1 and SR2 into register DR
br	0000	Conditionally branch if a matching condition is present
ldr	0110	Load register DR from the location specified by BaseR + offset9
not	1001	Put the bitwise complement of register SR into register DR
str	0111	Store the word from register ST at the effective address.

D. Sample Errata/ Debugging Help

"<Name> is not a signal type/state/block." This often happens when items of different types (state, signal, block, etc.) have the same name. Sometimes the generator catches these types of errors, but there are times when such errors fall through the cracks. No items should have the same name as any other item, and names are NOT case sensitive. This should not be a problem

now, but it will come up during MP2 and MP3. Find a consistent naming scheme you like and stick to it to avoid problems.

E. RTL

Note: Assignment of default values to signals is not shown in these tables.

E.1. FETCH Process

State	Data	Control
IF1	MAR ← PC; PC ← PC + 2;	LoadMAR ← '1'; LoadPC ← '1'
IF2	While (MRESP_H = '0') {MDR ← M[MAR];}	LoadMDR ← '1'; MREAD_L ← '0' after 6 ns;
IF3	IR ← MDR;	LoadIR ← '1';

E.2. DECODE Process

State	Data	Control
DECODE	// NONE	// NONE (note that although there is no code here, realistically speaking an instruction needs time to be decoded so that the processor knows which branch to take

E.3. ADD Instruction

State	Data	Control
FETCH		
DECODE		
ADD	DR ← A + B;	ALUop ← alu_add; LoadNZP ← '1'; RFMuxSel ← '1'; RegWrite ← '1';

E.4. AND Instruction

State	Data	Control
FETCH		
DECODE		
L_AND	DR ← A & B;	ALUop ← alu_and; LoadNZP ← '1'; RFMuxSel ← '1'; RegWrite ← '1';

E.5. NOT Instruction

State	Data	Control
FETCH		
DECODE		
L_NOT	DR ← NOT(A);	ALUop ← alu_not; LoadNZP ← '1'; RFMuxSel ← '1'; RegWrite ← '1';

E.6. BR Instruction

State	Data	Control
FETCH		
DECODE		
BR1	// NONE	Goto BR2 if ((n AND CheckN) OR (p AND CheckP) OR (z AND CheckZ)) else goto IF1
BR2	PC ← PC + ADJ9(IR[8:0]);	PCMuxSel ← '1'; LoadPC ← '1';

E.7. LDR Instruction

State	Data	Control
FETCH		
DECODE		
CalcAddr	MAR ← A + ADJ6(IR[5:0]);	MARMuxSel ← '1'; LoadMAR ← '1'; ALUMuxSel ← '1'; ALUop ← alu_add;
LD1	While (MRESP_H = '0') { MDR ← M[MAR]; }	While (MRESP_H='0') { LoadMDR ← '1'; MREAD_L ← '0' after 6 ns; }
LD2	DR ← MDR;	LoadNZP ← '1'; RegWrite ← '1';

E.8. STR Instruction

State	Data	Control
FETCH		
DECODE		
CalcAddr	MAR ← A + ADJ6(IR[5:0]);	MARMuxSel ← '1'; LoadMAR ← '1'; ALUMuxSel ← '1'; ALUop ← alu_add;
ST1	MDR ← SR;	ALUOp ← alu_pass; MDRMuxSel ← '1'; LoadMDR ← '1'; StoreSR ← '0';
ST2	While (MRESP_H = '0') { M[MAR]← MDR; }	While (MRESP_H='0') { MWRITEL_L ← '0' after 6 ns; MWRITEH_L ← '0' after 6 ns; }

F. CPU

F.1. Bundle contents

Signal_Name	Type
ALUMuxSel	std_logic
ALUop	LC3b_aluop
LoadIR	std_logic
LoadMAR	std_logic
LoadMDR	std_logic
LoadNZP	std_logic
LoadPC	std_logic
MARMuxSel	std_logic
MDRMuxSel	std_logic
PCMuxSel	std_logic
RFMuxSel	std_logic
RegWrite	std_logic
StoreSR	std_logic

Table 1: Control_out Signals

Signal_Name	Type
n	std_logic
z	Std_logic
p	std_logic
Opcode	LC3b_opcode
CheckN	std_logic
CheckZ	std_logic
CheckP	std_logic

Table 2: Control_feedback Signals

G. Control

G.1. Signals and Defaults

Name	Default Value
ALUMuxSel	‘0’
ALUop	“000”
LoadIR	‘0’
LoadMAR	‘0’
LoadMDR	‘0’
LoadNZP	‘0’
LoadPC	‘0’
MARMuxSel	‘0’
MDRMuxSel	‘0’
MREAD_L	‘1’
MWRITEL_L	‘1’
MWRITEH_L	‘1’
PCMuxSel	‘0’
RFMuxSel	‘0’
RegWrite	‘0’
StoreSR	‘1’

Table 3: Control Unit Signals

H. Components

H.1. Ports

Name	Direction	Type
A	IN	LC3b_word
B	IN	LC3b_word
F	OUT	LC3b_word
Sel	IN	std_logic

Table 4: WordMux2 ports

Name	Direction	Type
load	IN	std_logic
Input	IN	LC3b_word
clk	IN	std_logic
RESET	IN	std_logic
Output	OUT	LC3b_word

Table 5: Reg16 ports

H.2. VHDL Architectures

```
ARCHITECTURE untitled OF WordMux2 IS
BEGIN
 PROCESS (A, B, Sel)
  variable state : LC3b_word;
 BEGIN
  case Sel is
   when '0' =>
    state := A;
   when '1' =>
    state := B;
   when others =>
    state := (OTHERS => 'X');
  end case;
  F <= state after delay_MUX2;
 END PROCESS;
END ARCHITECTURE untitled;
```

Figure 20: WordMux2 VHDL

```
ARCHITECTURE untitled OF Reg16 IS
 signal pre_out : LC3b_word;
BEGIN
 PROCESS (clk, RESET, Input)
 BEGIN
  if RESET = '0' then
   pre_out <= (others => '0');
  elsif clk'event and clk = '1' then
   if (load = '1') then
    pre_out <= Input;
   end if;
  end if;
 end process;

 Output <= pre_out after delay_reg;
END ARCHITECTURE untitled;
```

Figure 21: Reg16 VHDL

I. Datapath

I.1. Signals

Signal Name	Type	Origin Block	Destination Block(s)
ADDRESS	LC3b_word	MAR	Port Out
ADJ6out	LC3b_word	ADJ6	ALUMux
ADJ9out	LC3b_word	ADJ9	Bradd
ALUMuxSel	std_logic	Port In	ALUMux
ALUMuxout	LC3b_word	ALUMux	ALU
ALUop	LC3b_aluop	Port In	ALU
ALUout	LC3b_word	ALU	RFMux, MDRMux, MARMux
Braddout	LC3b_word	BRadd	PCMux
CheckN	std_logic	NZPSplit	Port Out
CheckP	std_logic	NZPSplit	Port Out
CheckZ	std_logic	NZPSplit	Port Out
clk	std_logic	Port In	Global
DATAIN	LC3b_word	Port In	MDRMux
DATAOUT	LC3b_word	MDR	Port Out
Dest	LC3b_reg	IR	StoreMux, RegFile
GenCCout	LC3b_cc	GenCC	NZP
index6	LC3b_index6	IR	ADJ6
LoadIR	std_logic	Port In	IR
LoadMAR	std_logic	Port In	MAR
LoadMDR	std_logic	Port In	MDR
LoadNZP	std_logic	Port In	NZP
LoadPC	std_logic	Port In	PC
MARMuxSel	std_logic	Port In	MARMux
MARMuxout	LC3b_word	MARMux	MAR
MDRMuxSel	std_logic	Port In	MDRMux
MDRMuxout	LC3b_word	MDRMux	MDR
MDRout	LC3b_word	MDR	IR, RFMux
n	std_logic	NZP	Port Out
offset9	LC3b_offset9	IR	ADJ9
Opcode	LC3b_opcode	IR	Port Out
p	std_logic	NZP	Port Out
PCMuxSel	std_logic	Port In	PCMux
PCMuxout	LC3b_word	PCMux	PC
PCPlus2out	LC3b_word	Plus2	PCMux
PCout	LC3b_word	PC	Plus2, MARMux, BRadd
RESET_L	std_logic	Port In	RegFile, PC
RFAout	LC3b_word	RegFile	ALU
RFBout	LC3b_word	RegFile	ALUMux
RFMuxSel	std_logic	Port In	RFMux
RFMuxout	LC3b_word	RFMux	RegFile, GenCC
RegWrite	std_logic	Port In	RegFile
SrcA	LC3b_reg	IR	StoreMux
SrcB	LC3b_reg	IR	RegFile
StoreMuxout	LC3b_reg	StoreMux	RegFile
StoreSR	std_logic	Port In	StoreMux
z	std_logic	NZP	Port Out

Table 6: Datapath Signals

I.2. Datapath Diagram

The completed datapath is shown in **Figure 22**. (Also available online)

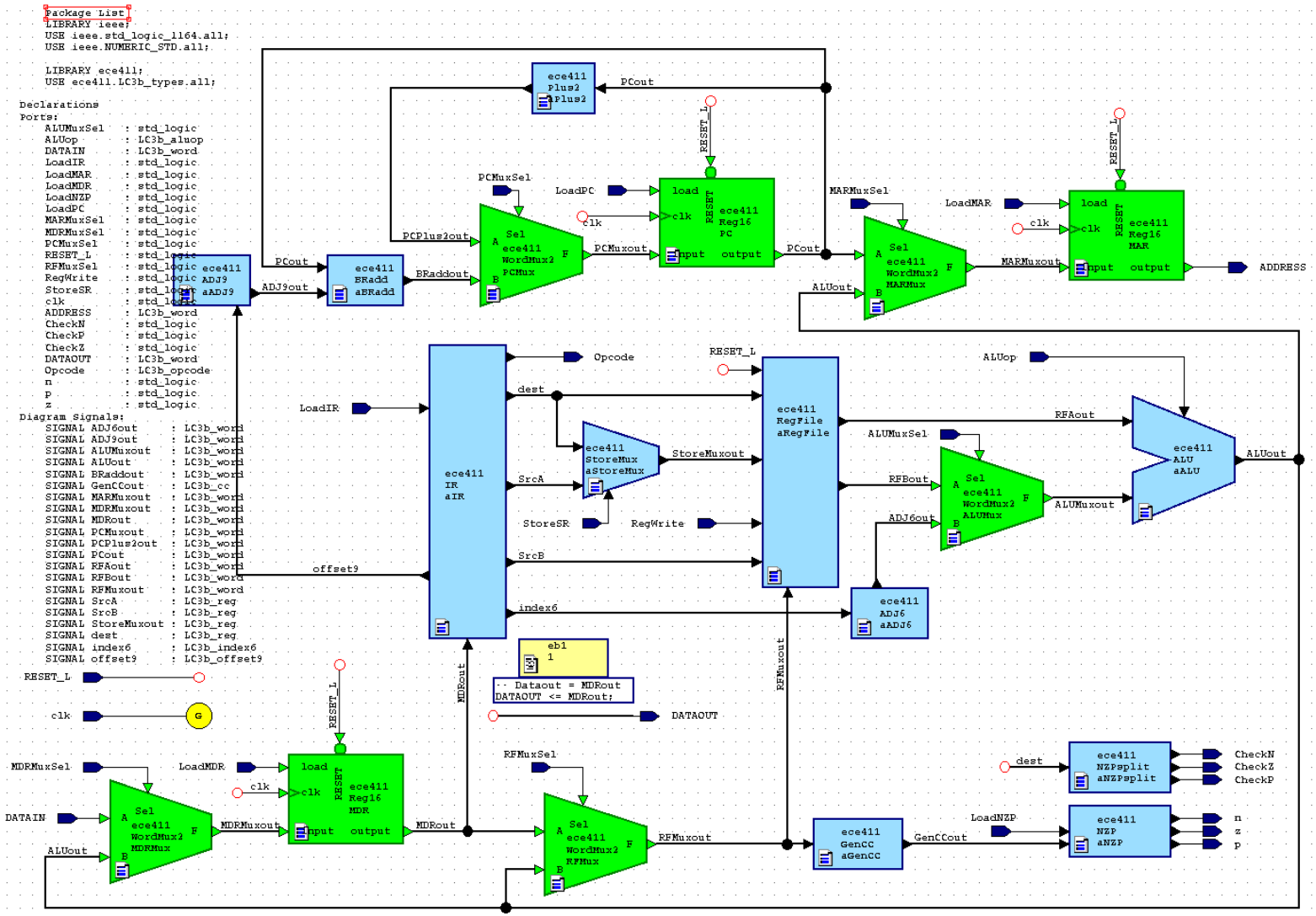

Package List
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.NUMERIC_STD.all;
LIBRARY ece411;
USE ece411.LC3b_types.all;
Declarations
Ports:
ALUMuxSel : std_logic
ALUop : LC3b_aluop
DATAIN : LC3b_word
LoadIR : std_logic
LoadMAR : std_logic
LoadMDR : std_logic
LoadNZP : std_logic
LoadPC : std_logic
MARMuxSel : std_logic
MDRMuxSel : std_logic
PCMuxSel : std_logic
RESET_L : std_logic
RFMuxSel : std_logic
RegWrite : std_logic
StoreSR : std_logic
clk : std_logic
ADDRESS : LC3b_word
CheckN : std_logic
CheckP : std_logic
CheckZ : std_logic
DATAOUT : LC3b_word
Opcode : LC3b_opcode
n : std_logic
p : std_logic
z : std_logic
Diagram Signals:
SIGNAL ADJ6out : LC3b_word
SIGNAL ADJ9out : LC3b_word
SIGNAL ALUMuxout : LC3b_word
SIGNAL ALUout : LC3b_word
SIGNAL BRaddout : LC3b_word
SIGNAL GenCCout : LC3b_cc
SIGNAL MARMuxout : LC3b_word
SIGNAL MDRMuxout : LC3b_word
SIGNAL MDRout : LC3b_word
SIGNAL PCMuxout : LC3b_word
SIGNAL PCPlus2out : LC3b_word
SIGNAL PCout : LC3b_word
SIGNAL RFAout : LC3b_word
SIGNAL RFBout : LC3b_word
SIGNAL RFMuxout : LC3b_word
SIGNAL SrcA : LC3b_reg
SIGNAL SrcB : LC3b_reg
SIGNAL StoreMuxout : LC3b_reg
SIGNAL dest : LC3b_reg
SIGNAL index6 : LC3b_index6
SIGNAL offset9 : LC3b_offset9
ece411 Plus2 aPlus2
PCout
RESET_L
PCMuxSel
LoadPC
clk
load
RESET
ece411 Reg16 PC
input
output
PCPlus2out
Sel
A
B
F
ece411 WordMux2 PCMux
PCMuxout
MARMuxSel
LoadMAR
ece411 Reg16 MAR
MARMuxout
ADDRESS
ALUout
ece411 ADJ9 aADJ9
ADJ9out
ece411 BRadd aBRadd
BRaddout
LoadIR
ece411 IR aIR
Opcode
dest
SrcA
SrcB
StoreSR
RegWrite
offset9
index6
ece411 StoreMux aStoreMux
StoreMuxout
ece411 RegFile aRegFile
ALUop
RFAout
RFBout
ALUMuxSel
ece411 WordMux2 ALUMux
ALUMuxout
ece411 ALU aALU
ALUout
ADJ6out
ece411 ADJ6 aADJ6
eb1
1
-- Dataout = MDRout
DATAOUT <= MDRout;
DATAOUT
MDRout
RFMuxout
RESET_L
clk
G
MDRMuxSel
LoadMDR
DATAIN
ALUout
ece411 WordMux2 MDRMux
MDRMuxout
ece411 Reg16 MDR
RFMuxSel
ece411 WordMux2 RFMux
RFMuxout
ece411 GenCC aGenCC
GenCCout
LoadNZP
dest
ece411 NZPsplit aNZPsplit
CheckN
CheckZ
CheckP
ece411 NZP aNZP
n
z
p

Figure 22: The Completed Datapath Block Diagram

J. Control Diagrams

The top-level Control diagram is shown **Figure 23**.

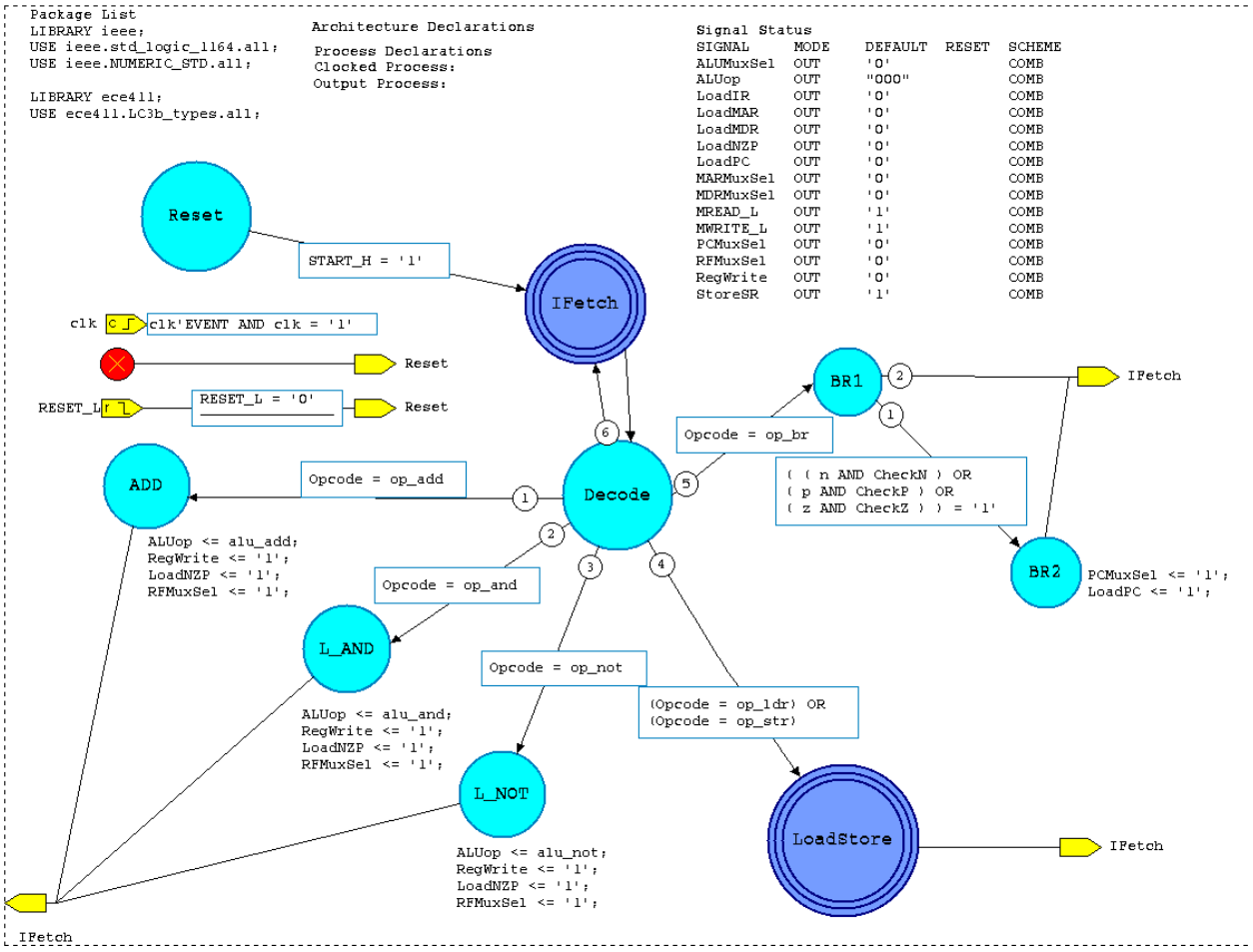


Figure 23: The top-level Control State Machine Diagram

J.1. IFetch Hierarchical State Diagram

The IFetch State Diagram is shown in **Figure 24.**

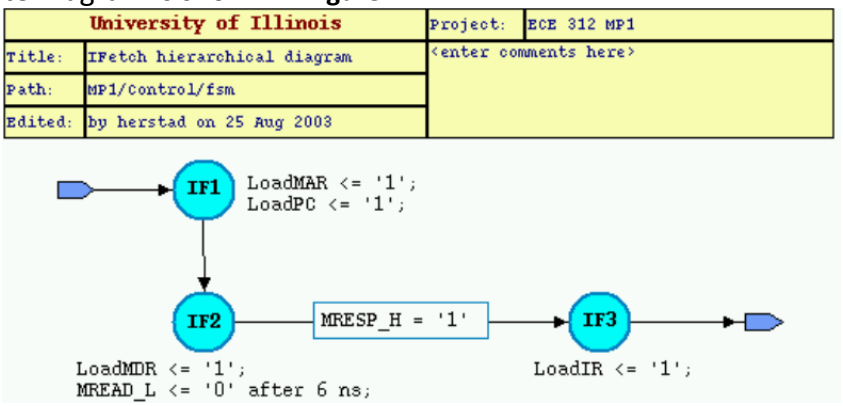


Figure 24: The IFetch sub-diagram of the Control State Machine

J.2. LoadStore Hierarchical State Diagram

The LoadStore State Diagram is shown in **Figure 25**.

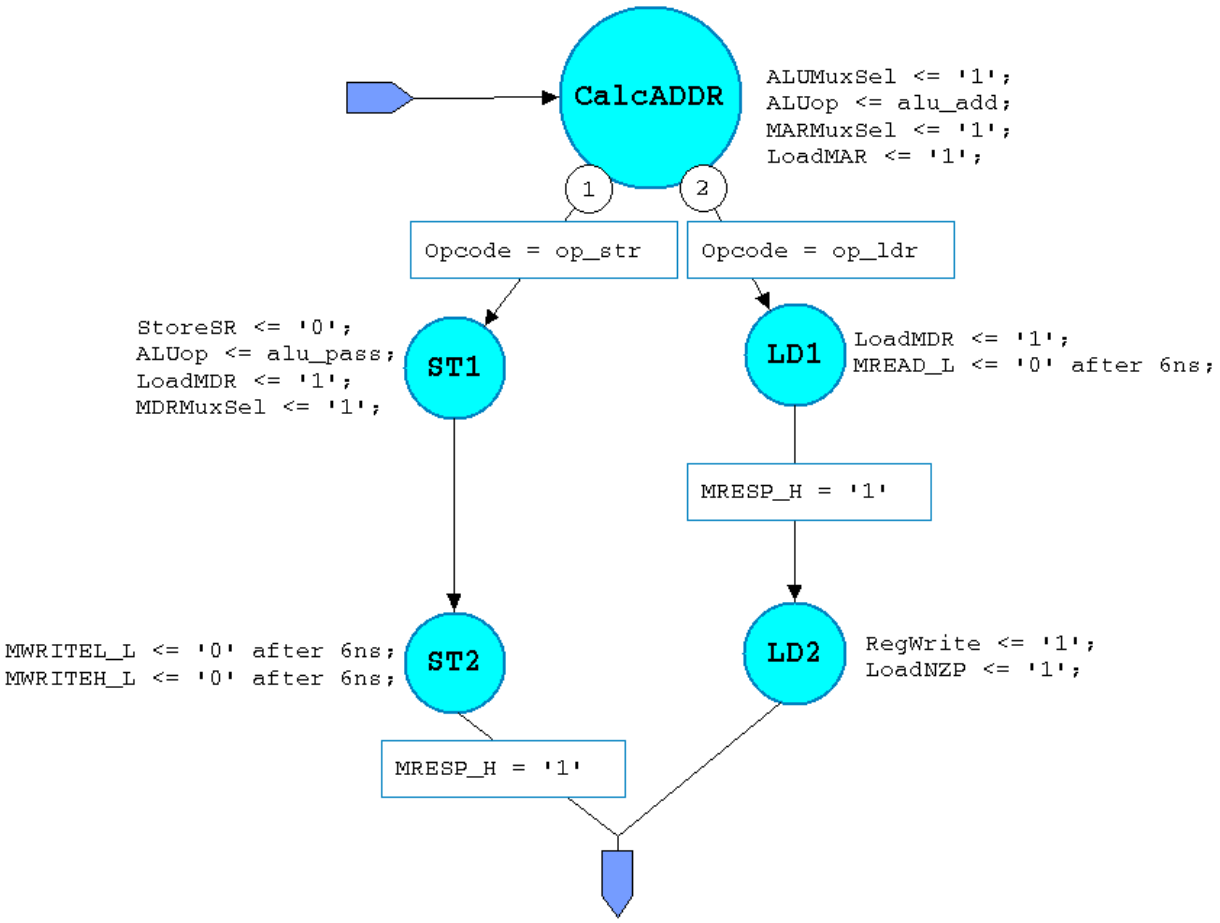


Figure 25: The LoadStore sub-diagram of the Control State Machine

K. Setting up Preferences

K.1. Set Program Preferences

From the Design Manager, choose **Options → Main. . .** to display the Main Settings dialog box. Relevant options for each tab in this box are described below. When all values have been set, click the **OK** button to return to the Design Manager.

“General" tab

When **Remain Active** is selected, the editor toolbar buttons auto-repeat (for example to add multiple objects of the same type). If you would prefer the buttons to have single-shot behavior, you should select **Activate Once Only**. Make sure VHDL is selected as the default language to be used for new diagrams. (This should already be set by default.) All other options should be left at their initial values.

“Text" tab

Here you can change which program is used for editing VHDL files. We recommend using DesignPad.

From the Design Manager, choose **Options → Master Preferences → Block Diagram**. Under “Display Settings," click “VHDL Signal Display" in the left-hand column. Make the options match those shown in **Figure 26**, then return to the Design Manager.

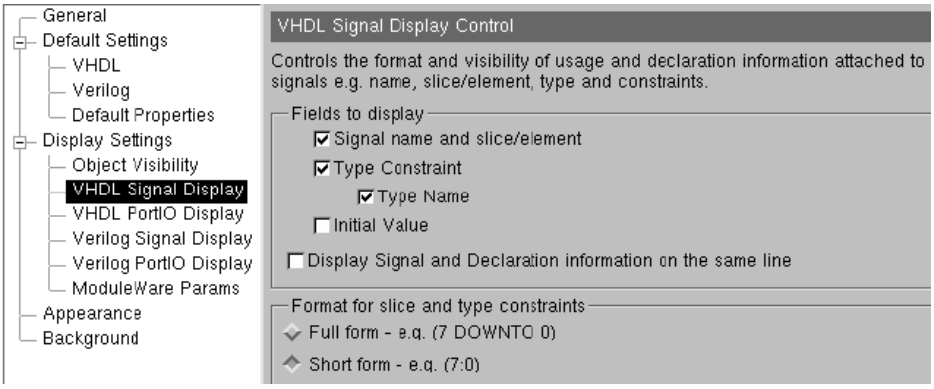


Figure 26: VHDL Signal Display Control

From the Design Manager, choose **Options → Master Preferences → State Machine. . .** Select “General" from the left-hand column.

You can set the minimum radius for states, hierarchical states and the transition priority object. The states will auto-size if the state name is larger than can be enclosed by the miscellaneous radius. However, the minimum size is overridden if you check **Shrink state bubble to fit name**, which allows the states to shrink below this size if the state names are short.

Set the “state radius" and “hierarchical state radius" values to 3000. This radius should be sufficient to enclose the state names used in this machine problem. We do not recommend changing any of the other preferences. Your window should look like **Figure 27**.

Use the **OK** button to set the changed preference. If it warns you that you are changing the master preference template, click **Yes**.

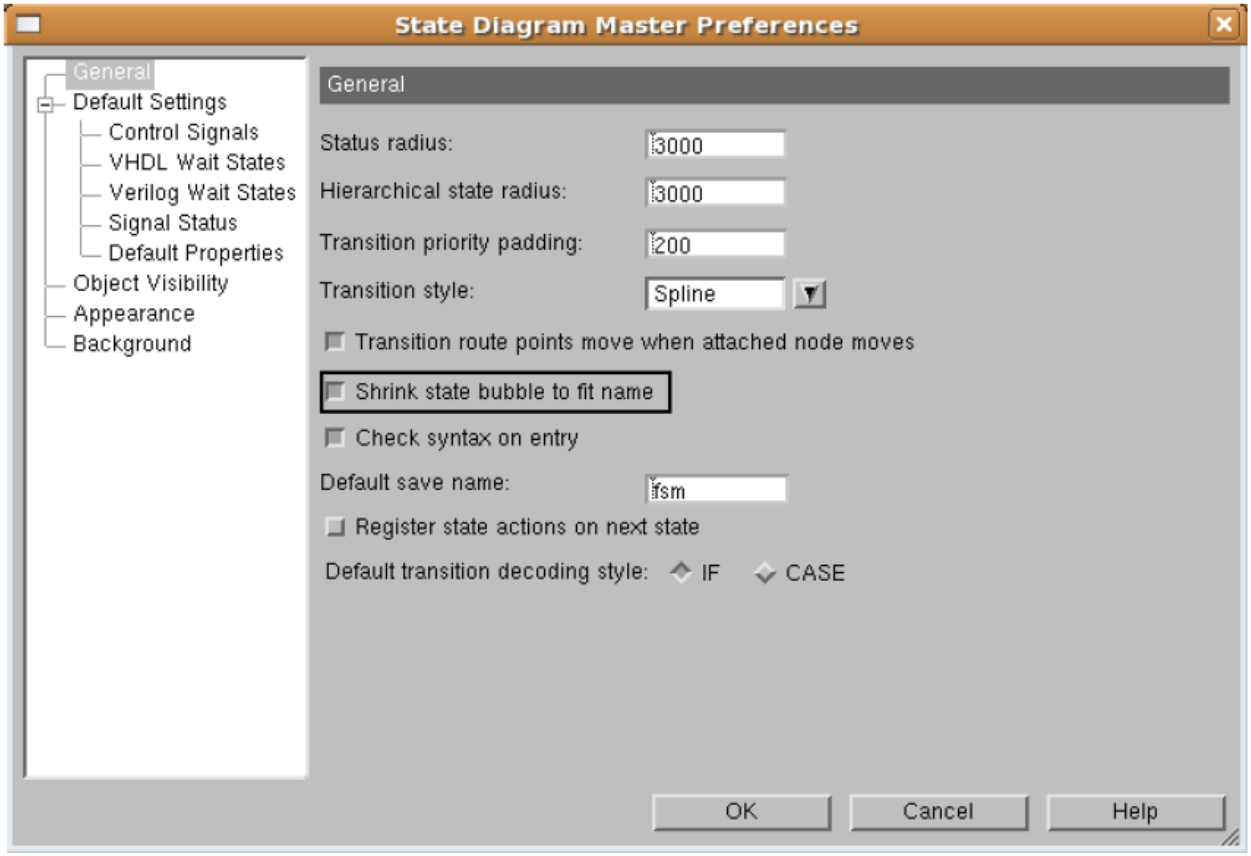


Figure 27: The master state machine preferences dialog