Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

ECE 312

Computer Organization and Design

Fall 2001

MP1:
The LC-2α Processor / Renoir Tutorial

Version 2.0

S. Ryoo
S. J. Patel

TABLE OF CONTENTS

Chapter 1: Introduction

Welcome to the first ECE 312 Machine Problem! In this MP we will step through the design entry and simulation of a simple, non-pipelined processor that implements a portion of the LC-2 instruction set architecture. 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 Renoir and ModelSim. For MP2 and MP3, you will use Renoir for design entry and ModelSim for design simulation. It is important for you to understand how these tools work now so that during the later MPs, you can focus on the design effort without being bogged down with tool-related problems.

We will discuss the MP1 LC-2α processor extensively in class during the first several lectures. The lecture material will augment the cookbook approach taken within this tutorial. Understanding the lecture material will deepen your understanding of this tutorial, and likewise, going through this tutorial will help you understand the corresponding lectures. MP2 and MP3 will build heavily upon this MP1 design, so it is important for you to get the MP1 LC-2α processor working correctly.

In terms prior to Fall 2000, ECE 312 students used a pre-packaged tutorial from Mentor Graphics. We have decided to depart from the standard Renoir tutorial and write our own, customized to the needs of ECE 312. Your feedback on it would be greatly appreciated.

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 with it yourself before proceeding to the tutorial itself. The second chapter contains a description of the six instructions in the LC-2α 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 Renoir. 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 students had with the MP came from missing a paragraph and some key steps. Take your time, and be thorough, as you will need a functional MP1 design before working on future MPs.

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.

- The notation [address] means the contents of memory at address ‘address’. For example, if MAR = 0x12, then [MAR] would mean the contents of memory location 0x12.

- For RTL descriptions, the following operators are used:

  || means concatenation, e.g., 0||1 is 01. @ is also used in some sources.

  X^y means y consecutive Xs, 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.

Chapter 2: The iLC-2α Instruction Set Architecture

Overview

For this project, you will be entering the VHDL design (using Renoir) of a non-pipelined version of the LC-2α ISA. We will discuss the LC-2α ISA extensively in class.

The LC-2α ISA consists of six instructions selected from the full LC-2 ISA (16 instructions). The LC-2 ISA is an ISA created for instructional purposes. Because it is a relatively simple ISA, it is a natural choice for our ECE 312 projects.

All six instructions are 16 bits in length, having a format where bits [15:12] contain the opcode. The LC-2α 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.

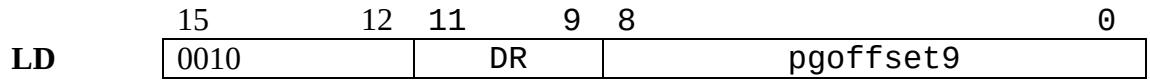
The memory of the LC-2α consists of 2^{16} locations (meaning the LC-2α has a 16 bit address space) and each location contains 16 bits (meaning that the LC-2α has 16-bit addressability). The memory is further conceptually divided into 128 regions called *pages.* Each page, therefore, contains 512 locations. More on pages later.

The LC-2α 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

Data Movement Instructions

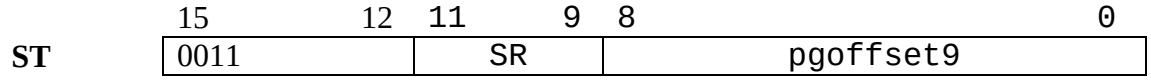
Data movement instructions are used to transfer values between the register file and the memory system. The load instruction (LD) reads a 16-bit value from the memory system and places it into a general-purpose register. The store instruction (ST) takes a value from a general-purpose register and writes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

The format of the load instruction, or *LD*, is shown below. The opcode of the *LD* instruction (bits [15:12]) is 0010. The effective address (the address of the memory location that is to be read) is the memory location specified by the 9-bit *page offset* on the same page as the LD instruction itself. The effective address is calculated by concatenating the upper 7-bits of the Program Counter (i.e., PC[15:9]) with the page-offset field in the instruction (i.e., pgoffset9).



The format of the store instruction, *ST*, is shown below. The opcode of this instruction is 0011. As with the load instruction (*LD*), the effective address is the memory location specified by the

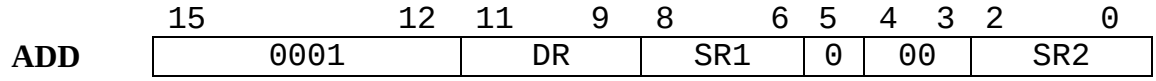
9-bit *page offset* on the same page. The effective address is formed in the same manner as that of the LD.



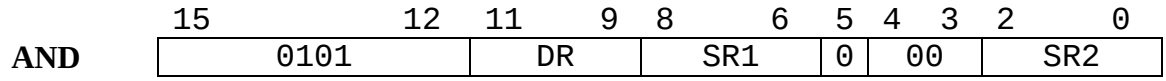
Operate Instructions

LC-2α has three integer operate instructions: ADD, AND, and NOT.

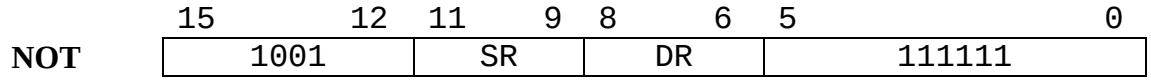
The ADD instruction takes a value from the general-purpose register specified by SR1 (SR stands for source register) and *adds* it the value from the register specified by SR2. The result is stored in the register specified by DR (DR stands for destination register). The format of the ADD instruction is shown below. The opcode for ADD is 0001.



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.

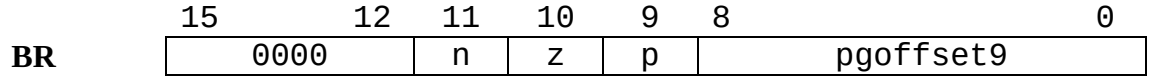


This is the format of the NOT instruction. It does a bitwise complement on the value in register SR and places the result in register DR.



Control Instruction

The LC-2α 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 next instruction to be executed is at the address specified by the page offset (pgoffset9) on the same page as the branch instruction (ie., PC[15:9]||pgoffset9)



Chapter 3: Specifications

Signals

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

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 processors to execute the instruction located at memory address 0x00000000. It must stay high for at least one clock cycle. Afterward, START_H may change state without affecting the microprocessor's operation.

CLK
: All components of the design are active on the rising edge.

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.

MWRITE_L
: Active low signal that tells memory that the address is valid and that the processor is trying to perform a memory write.

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 data paths, 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.

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 should 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 MWRITE_L signal active when it is writing to the memory. 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.

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.

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. The control unit for the design is detailed in **Appendix F**.

Chapter 4: Design

Setup

The purpose of this MP, as stated before, is to become acquainted with the basic architecture and software tools used to design the microprocessor that will be further developed and enhanced in future machine problems. You will be using Renoir™ from Mentor Graphics to layout the design and ModelSim to simulate it.

If you wish to learn more of the features in Renoir, you can go through the Renoir tutorial, which is available through Renoir itself (click on Help). This is the packet that was once sold for the course. The tutorial covers additional topics such as flow charts and truth tables.

To start using Renoir, enter your ECE 312 work directory by typing `ece312` in a terminal window in your EWS account. If you do not have an EWS account, please contact one of the TAs, and they will help you obtain an account.

Next, type *`cd mp1`* to enter your mp1 directory. Type *`pwd`*. This will tell your which work directory your work account is located; it should be of the format `/work<x>/ece312/your_login` (<x> is an integer). You will probably want to write this directory down. If you have not downloaded mp1files.tar.gz yet do so now from the MP1 web page. Once you have this file, move it to the **`mp1`** directory and uncompress it with the following command: "***gunzip -c mp1files.tar.gz | tar xf –"***

Three subdirectories with files: **`compile`**, **`vhdl`**, and **`tools`** should now be in your **`mp1`** directory.

Note: Please check the web page for any updates to both the MP1 specification and the mp1files.tar.gz file.

Type *`renoir`* at the ece312 prompt. When the wizard window pops up, click on the **Continue** button. We are going to walk through the library setup process so that you know how to do it for MP2 and MP3; thus, we do not want to select any of the other options. You may choose to skip the wizard window in the future. A log window may also pop up, which is generally not a concern.

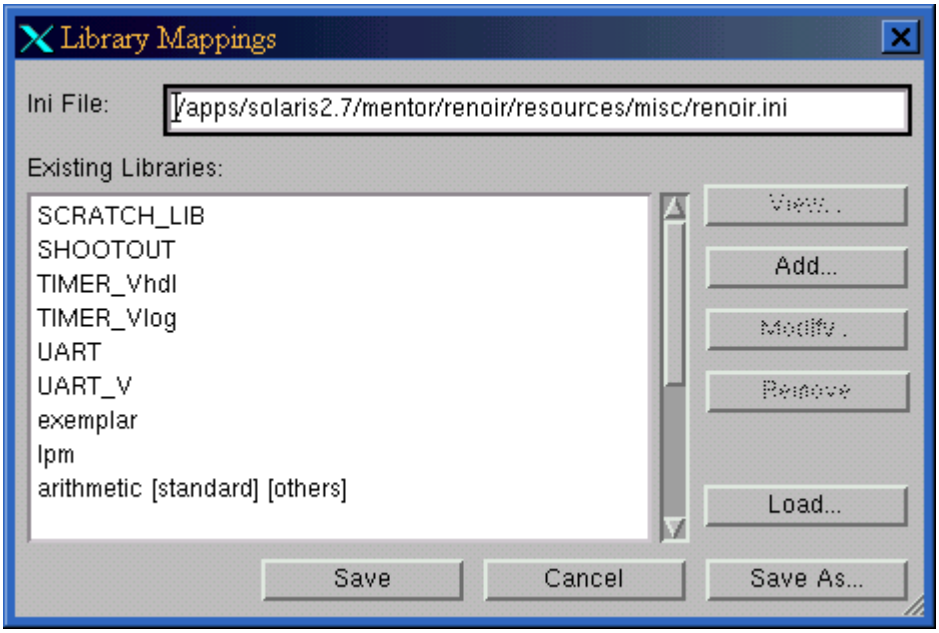
Set Library Mapping

(Paraphrased from the 1998 Renoir Tutorial.)

All designs for the course must be saved in a library directory. Before we can begin, we must establish a mapping between our design (which in this case is MP1) and a directory in which to store that design. In Renoir, this mapping is called the Library Mapping. Each of our designs in ECE312 (MP1, MP2, and MP3) will be a new library in Renoir.

First, choose **File → Close All Libraries** to close the sample libraries. To set the library mapping for MP1, choose **Options → Library Mappings** in the Design Browser window to

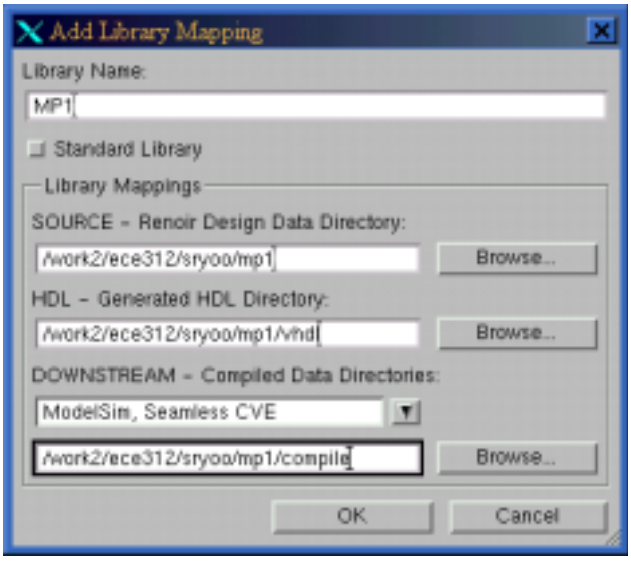
display the Library Mappings dialog box. The dialog box lists the existing libraries, mainly the standard VHDL libraries, which were loaded when **Renoir** was installed. The Library Mappings dialog box should look similar to the example below:



Note: The **View** button allows you to examine the current mappings for any of the existing libraries. You can also use the **Modify** button to change an existing library mapping or the **Remove** button to remove a library from the list.

Use the **Add** button to display the Add Library Mapping dialog box. Enter `MP1` and specify the pathname for the directory that will contain your Renoir design data. Again, this should be similar to `/work<x>/ece312/your_login/mp1`. This directory should exist already, but if it does not exist, it is created when you save a design unit to the MP1 library (provided that the parent directory exists and you have the write permissions to create a subdirectory at the specified location). If you have problems with this, consult a lab TA.

The Add Library Mapping dialog box should look similar to the example below:

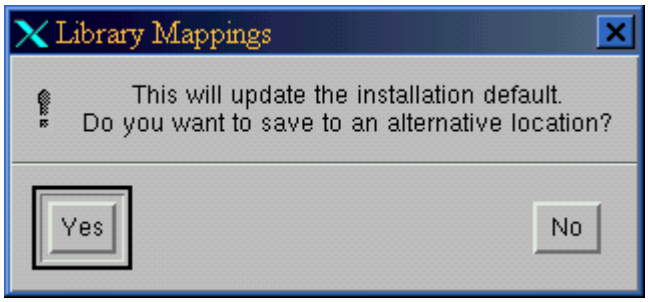


Library names and pathnames can be entered using upper, lower, or mixed case. Remember that UNIX systems are case sensitive; try to use a consistent format to simplify and speed your work.

Type *MP1* for Library Name. For the SOURCE, use the **mp1** directory in your ece312 work directory. Use the **vhdl** and **compile** subdirectories for the HDL and DOWNSTREAM directories, respectively. Choose ModelSim from the downstream tools pull-down menu, if it is not already selected. Click the **OK** button to close the Add Library dialog box and return to the Library Mapping dialog box.

Note: Since you have specified a location inside your diagram source directory for the downstream, this directory will appear as an unknown design unit in the design browser. You can browse generated HDL files below this design unit.

Your library definition must be saved in an initialization file. If you used the **QuickStart** option from the wizard, this file will have already been created and is updated when you use the **Save** button. If you try to overwrite the installation default initialization file, you are prompted to save to an alternative location and the **Save "ini" File As** dialog box is displayed as below.



The save location defaults to your user directory with the file name `renoir.ini`. Hit **Yes** to confirm your selection, then hit **Okay** to save your library mapping to your home directory.

Once it has been saved, the pathname to the current initialization file is shown in a read-only field at the top of the Library Mappings dialog box.

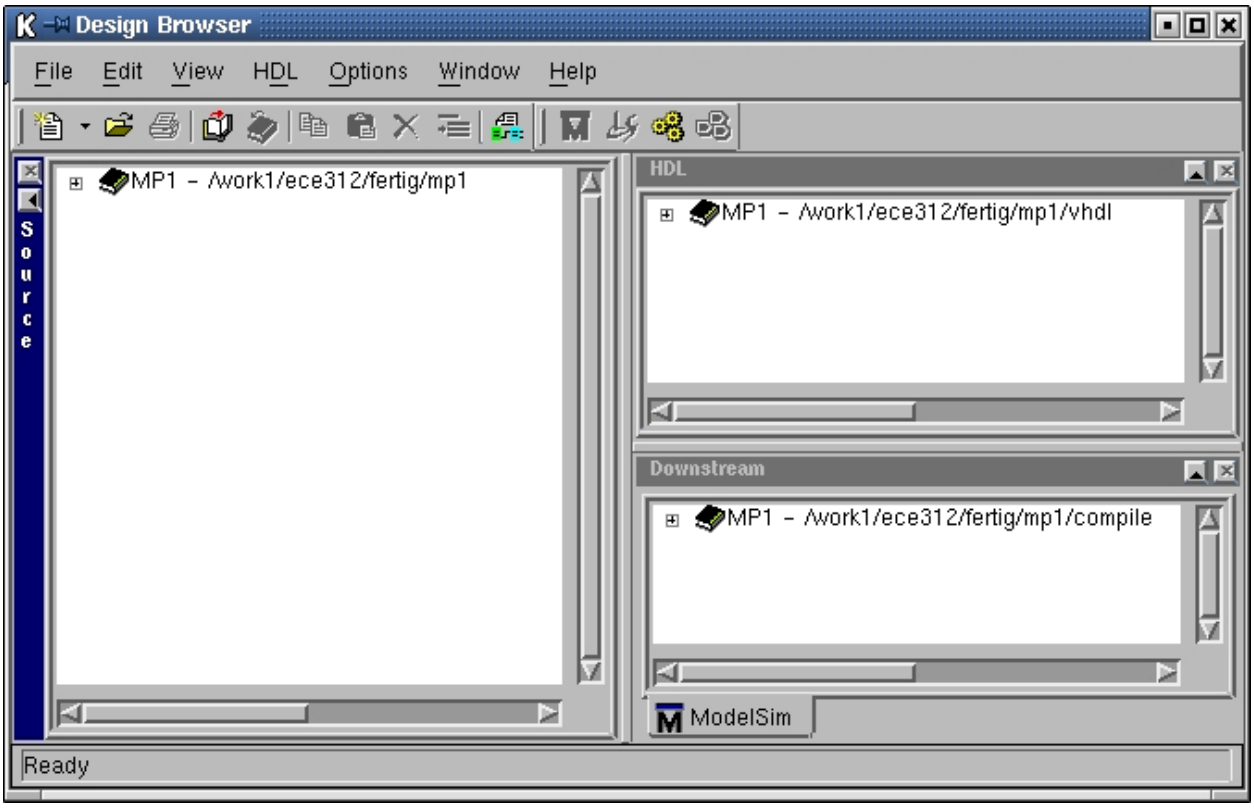
Note: You can also load an alternate initialization file using the **Load** button. This option is only available when no libraries are open.

Opening and Closing Libraries

Use the  button on the toolbar or choose **File → Open Library** from the Design Browser window to display a list of mapped libraries and choose the `MP1` library. Hit **Open**.

Note: You can use the  button on the toolbar or **File → Close Library** to close any existing libraries or choose **File → New Design Browser** open a new empty browser window.

The library should be displayed as a "folder" icon similar to the following picture:

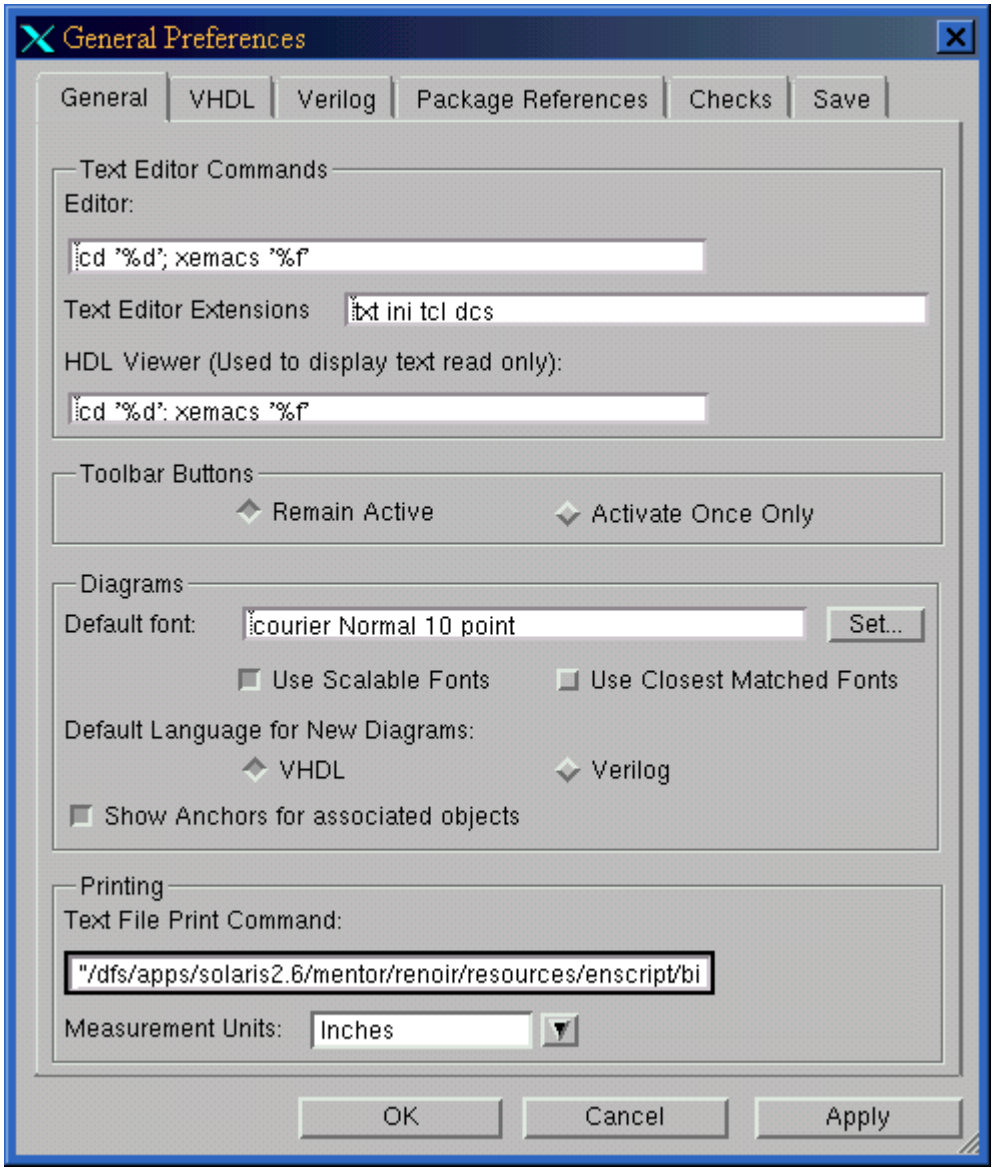


Note: A red cross is superimposed on the folder icon if the design data directory does not exist. It will be cleared and the directory created once the design data is saved to the directory.

Set General Preferences

The first time you invoke **Renoir**, the Welcome wizard usually sets the default hardware description language. Choose **Options → General Preferences** to display the General Preferences dialog box.

The General Preferences dialog box should appear similar to this:



Make sure VHDL is the default language to be used for new diagrams. All other options should be left at their initial values. However, you may want to change the following preferences:

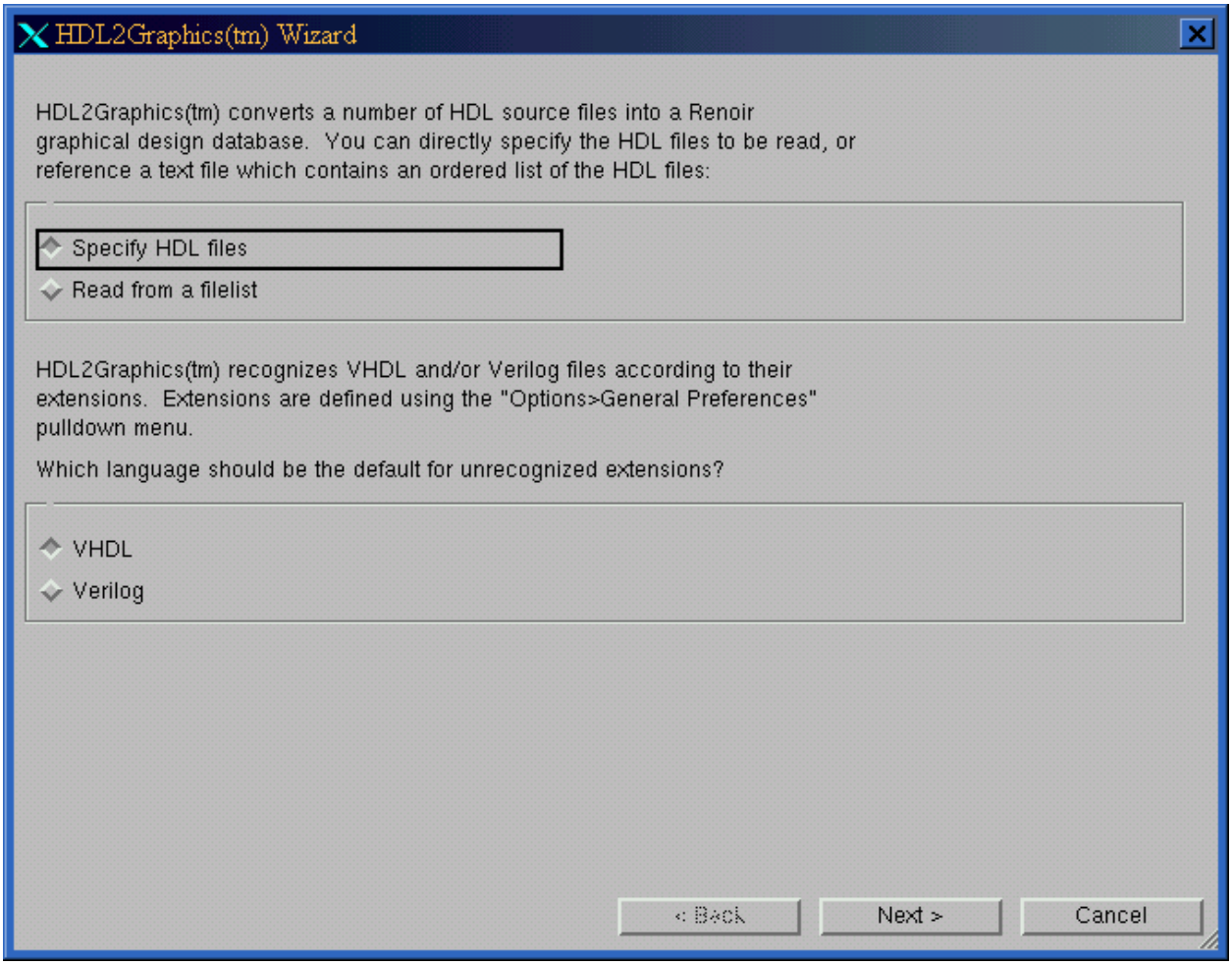
- When **Remain Active** is selected, the editor toolbar buttons auto-repeat (for example to add multiple objects of the same type). If you would prefer the buttons to have single-shot behavior, you should select **Activate Only Once**.

- You can change the **External Editor Command** used for creating and editing HDL views. The default editor is usually vi, which most students do not know. As an alternative we recommend using emacs or xemacs. See the previous screenshot for an example.

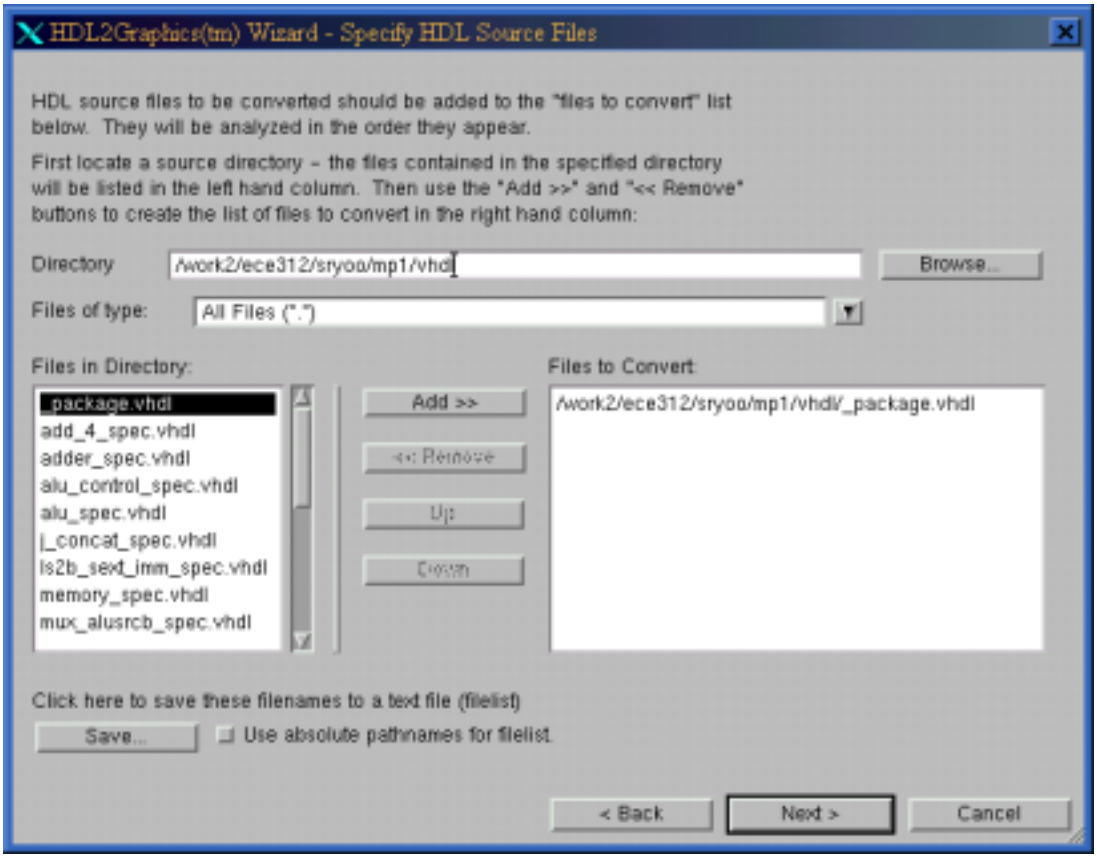
Set Up LC2_types Package

An important package that has not been mentioned to this point is the LC2_types package. This defines all of the constants that you will be using in the design. During generation, it will link to your VHDL and replace the constants with the values in the file. The _package.vhd file, which was included in the mp1files.tar.gz file, should already be in your **`vhdl`** subdirectory. This file is required to complete this section.

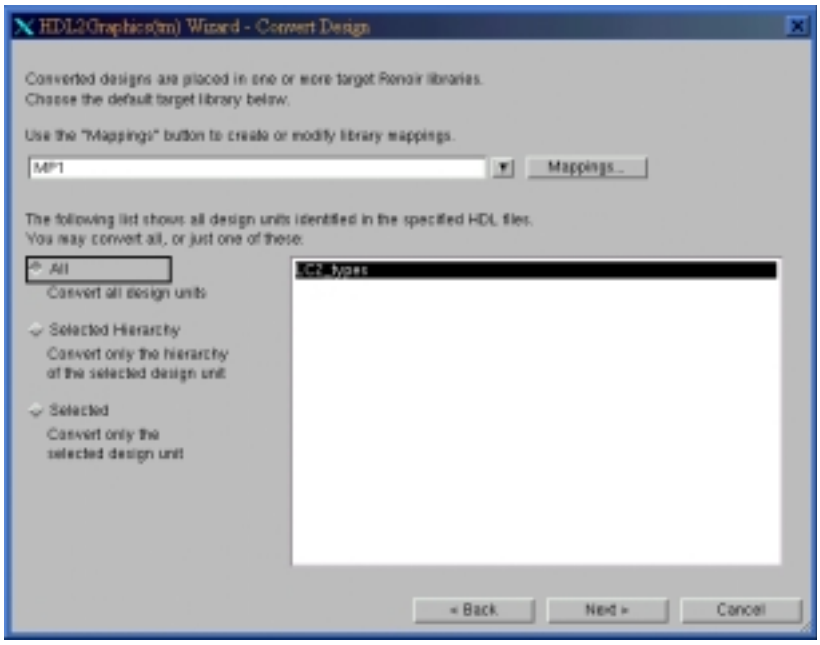
To link the file, select **File → Convert HDL to Graphics**. The HDL2Graphics(tm) Wizard dialog box will appear. Make sure that *Specify HDL files* and *VHDL* are selected as choices, then click **Next**.



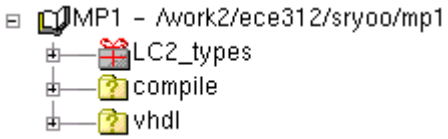
Now the dialog box contains new contents. Enter your **`vhdl`** directory for mp1, select the file `_package.vhd`, and click on the **Add** button. The dialog box should look similar to the picture below:



Click on **Next**. A log window may appear (if it has not already), which means that the program is processing the file. The contents of the window will change. Make sure MP1 is the default target library, and select *All* (LC2_types should be the only design unit). You should now have something similar to the following:



Click on **Next**, then **Finish**. Now, in the Source sub-window of your Design Browser window, you should have something similar to this:



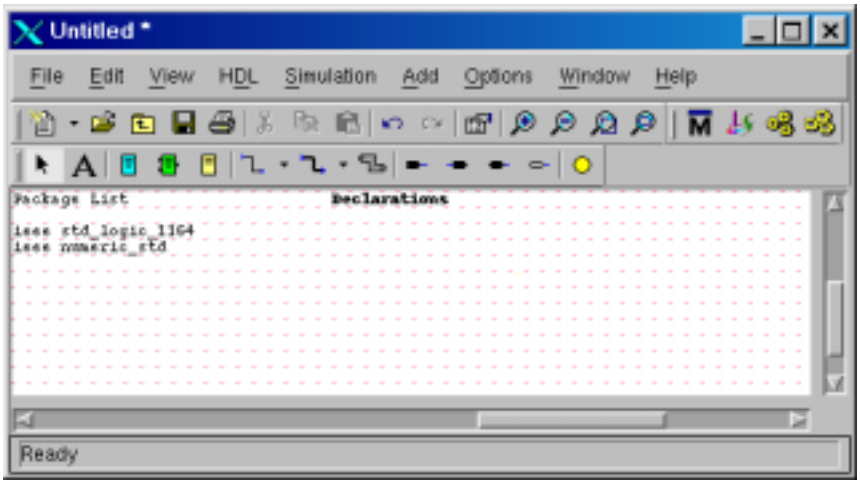
You are now ready to begin your design.

Beginning the Design

Create a Block Diagram

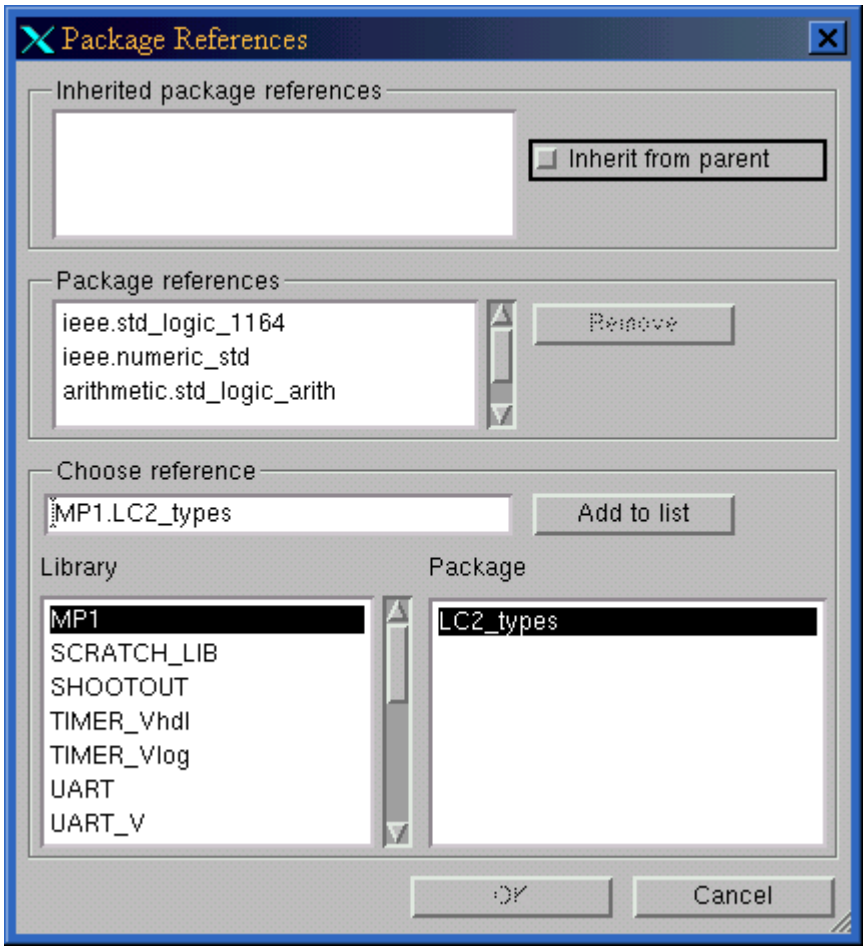
Use the  button in the design browser window and select **Block Diagram**.

A new untitled block diagram is created.



Blocks are effectively the “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 an empty text field with the label Declarations which will be used to enter local declarations that will be interpreted as architecture declarations in the generated VHDL for the block diagram.

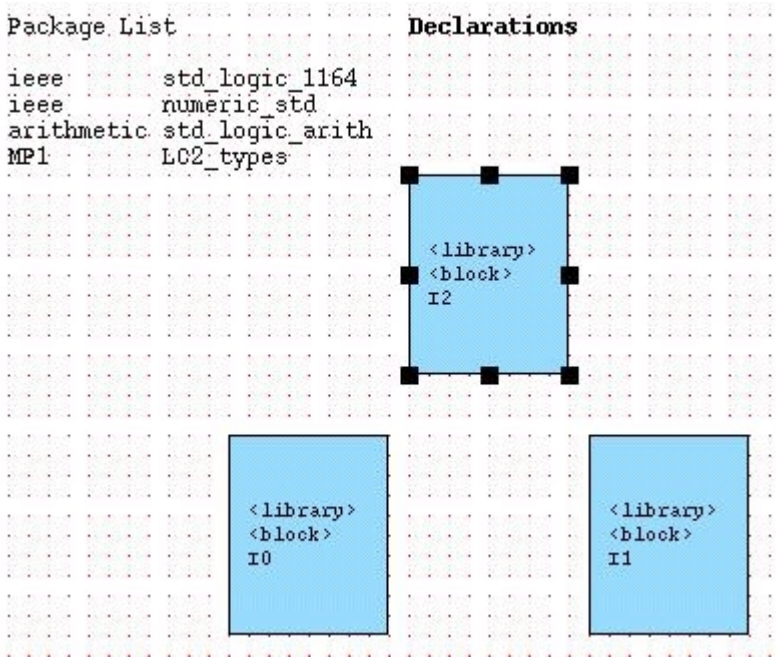
Look at your package list. You should have, at the very least, *ieee std_logic_1164* and *ieee numeric_std* present. Double-click the package list, and the following window will be displayed:



Select the MP1 library, then the LC2_types package. Click on the **Add to list** button, then click **OK**. The box should disappear and *MP1 LC2_types* should now be present in the package list. Use the same process to add the *arithmetic std_logic_arith* package, then click **OK** to make the changes.

Add and Name Blocks

Use the  button to add three blocks on your block diagram as shown in the picture below. The blocks are added with the default library <*library*>, the default name <*block*> and unique instance names (*I0*, *I1*, *I2*).

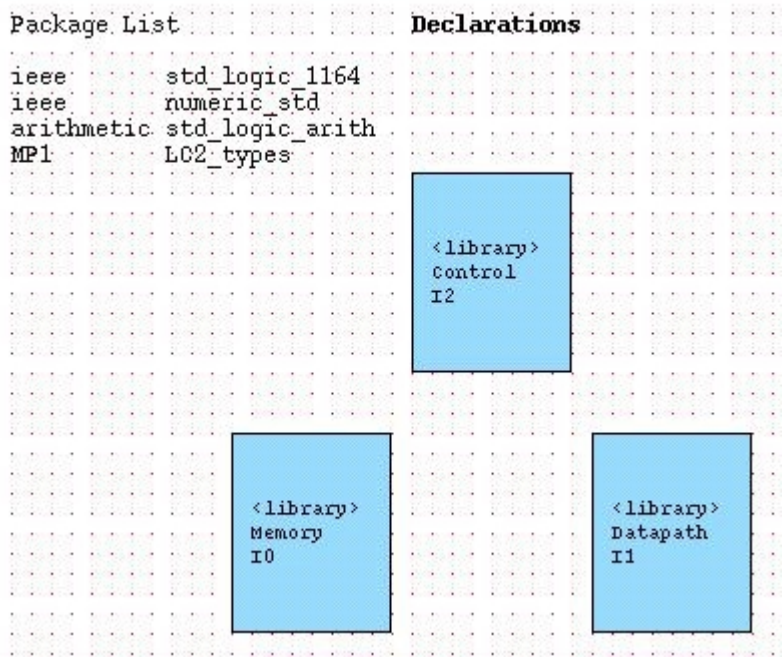


Notes: The command normally auto-repeats until you select another command or terminate the repeating command by using the right mouse button or the ESC key. As mentioned before, you can change the behavior of the toolbar buttons by setting the **Activate Once Only** preference in the General Preferences dialog box.

You can also use the Shift key with any toolbar button to toggle the repeat mode. For example, when **Remain Active** is set, Shift + adds a single block on a block diagram.

If you make a mistake when editing a diagram, you can use the button to undo your last edit and the button to redo an undo operation.

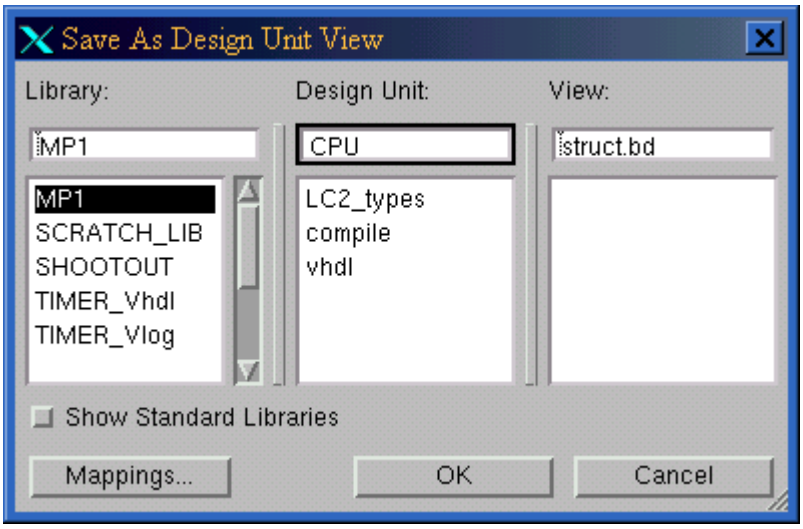
Click on the text <block> in the lower block on the left (instance *I0* in the picture) and notice the small handles that indicate that the text object is selected. Click again and notice that the text is now highlighted and can be directly overwritten. Type the name *Memory* and click outside the text to complete the edit. Repeat this procedure to change the name of block instance *I1* to *Datapath* and *I2* to *Control*. Your diagram should now look similar to the following:



Save the Block Diagram

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

Use the  button 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 *MP1* library and enter design unit *CPU*. The **Save As Design Unit View** dialog box should now look similar to the example below:



The view name can entered using any valid HDL identifier but normally defaults as follows:

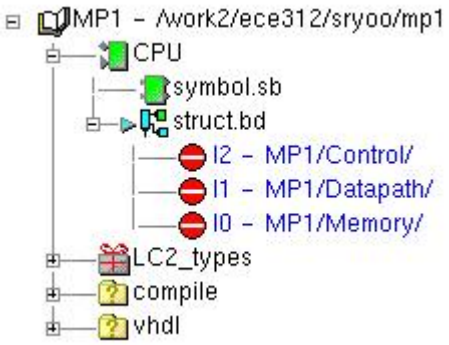
struct.bd block diagram

fsm.sm	state diagram
flow.fc	flow chart
table.tt	truth table
symbol.sb	symbol

If you omit the two-character extension it is automatically added to identify the type of diagram you are saving. The default leaf names can be changed by setting preferences. However, you should not change the extension (*.bd*, *.sm*, *.fc*, or *.tt*) or the design data file will not be recognized and cannot be reopened.

When you click the **OK** button, your diagram is saved and the window title bar is updated to show the diagram name *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 *MP1* and the design browser view is updated to display the *CPU* design unit.

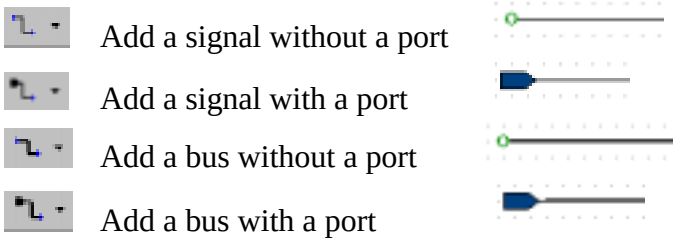
Click on the plus icon next to the CPU design unit in the Design Browser to expand the design unit. This reveals that it contains a symbol and block diagram view. Click on the ⊞ icon for the struct.bd view to display the blocks on the block diagram. The blocks are shown using the ⛔ icon to indicate that no views have been defined.

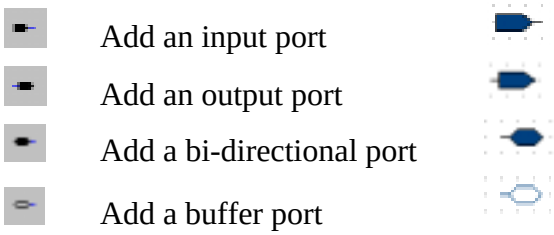


Note: A red cross superimposed on any icon in the design browser indicates that the view is not write-able. This convention is also used to show when you have read-only access. This will be of importance when you are sharing design units during group work.

Add Ports and Signals

A signal is a single wire connecting blocks. A bus is a group of wires connecting blocks. Ports are the interfaces between blocks and signals or buses. You can use the following buttons to add ports, signals and buses on a block diagram:



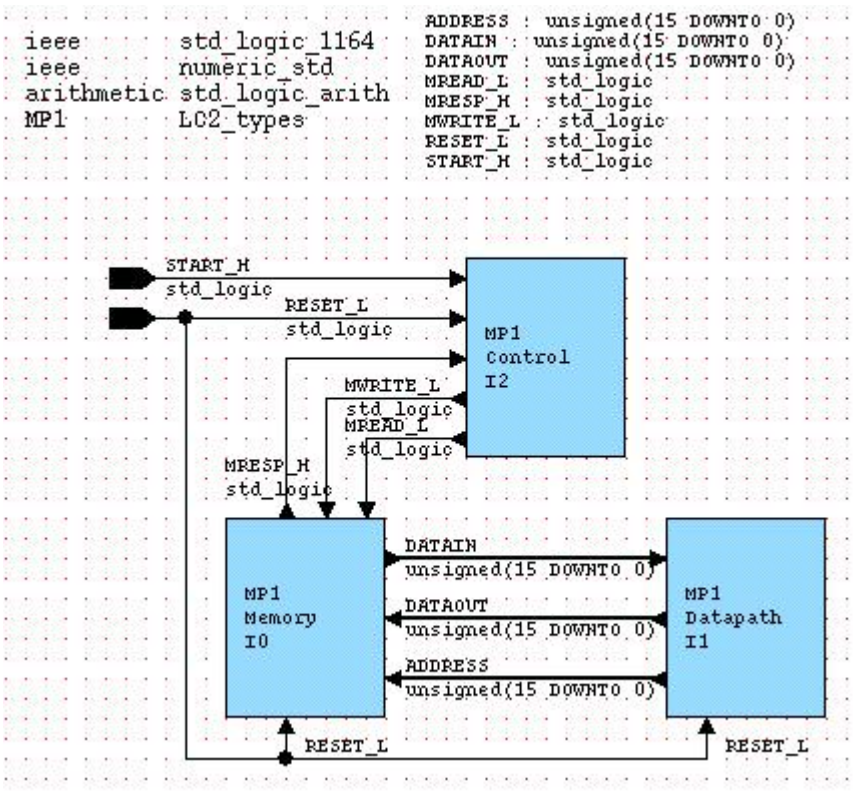


By default, signals and buses are added without ports but with a dangling net connector. However, you can use the button on the Add Signal and Add Bus buttons to change the default setting to add with a default port at unconnected end points. Note that the toolbar button changes to show the current setting.

Perform the following steps:

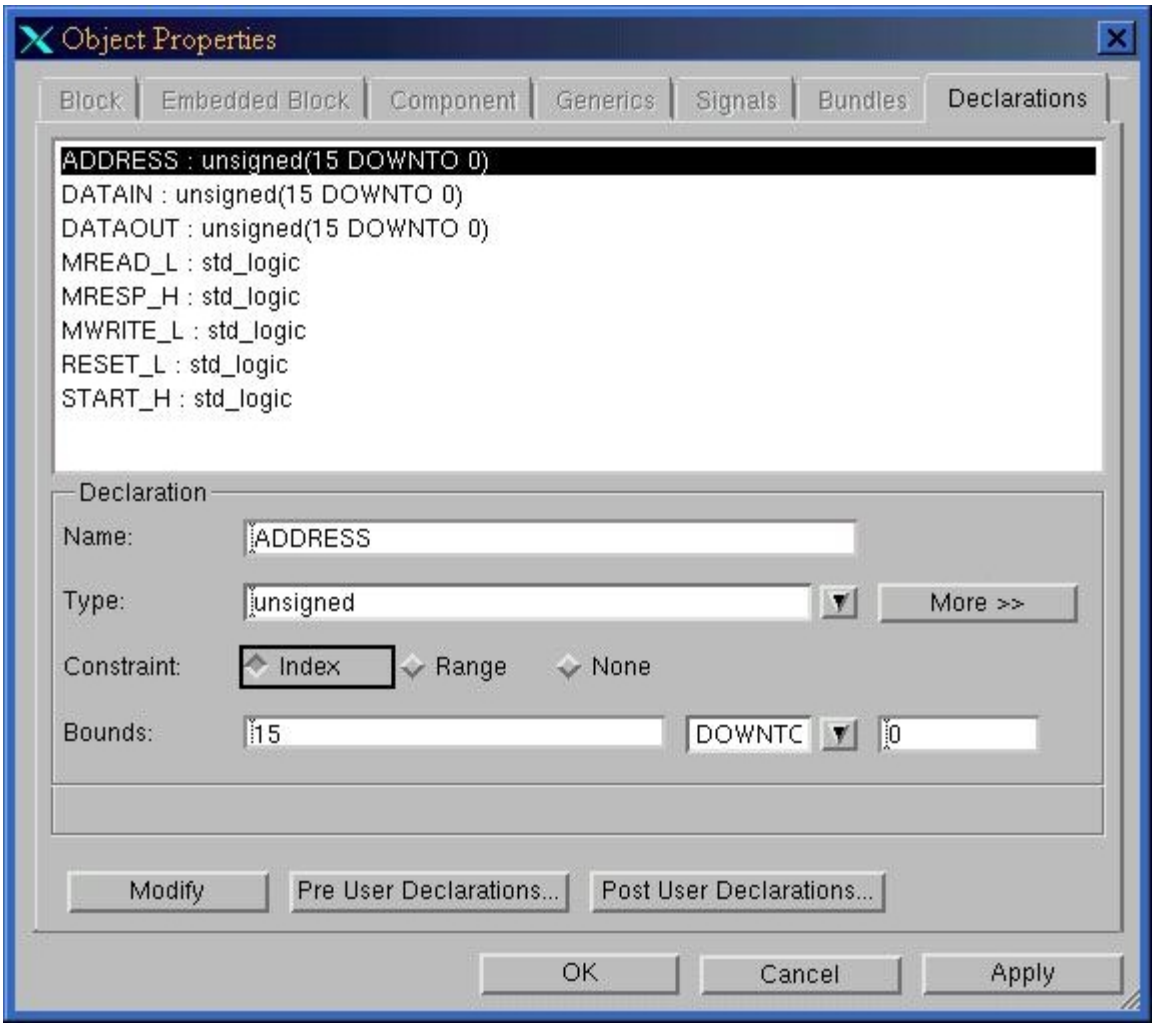
1. Connect two signals originating from Control to Memory. Name these MWRITE_L and MREAD_L.
2. Add another signal from Memory to Control, called MRESP_H.
3. Create two port-ins, with two signals from those to Control, called RESET_L and START_H.
4. Connect RESET_L to Memory and Datapath as well, by starting another signal from a point on the existing signal. Renoir should automatically rename the signal.
5. Create two buses from Datapath to Memory, called DATAOUT and ADDRESS.
6. Create another bus from Memory to the Datapath, called DATAIN.

Use the same technique you used to name the blocks to name the signals as shown below. You can also click and drag the names of the signals (or the blocks, for that matter) to a more visible or convenient place. Finally, you can right click and select **Add Route** to create more corners on the signal/bus.



Alternatively, you can use a dialog box that allows you to edit the properties for a selected object. You will need to do this now to properly change the types of the signals.

Use the  button (or choose **Edit → Object Properties**) to display the Block Diagram Object Properties dialog box. Choose the **Declarations** tab.



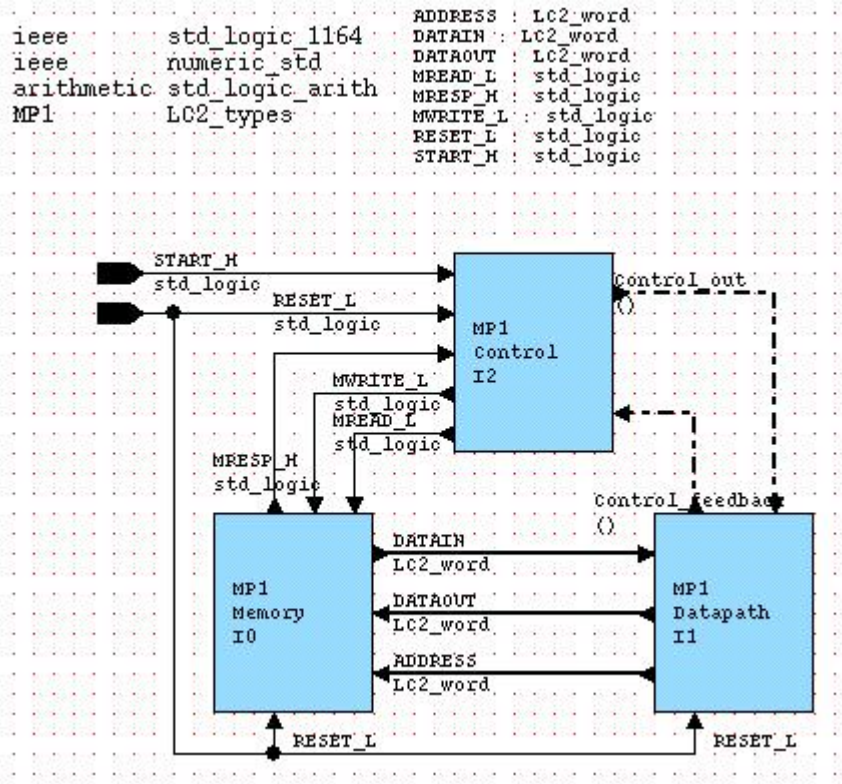
You can choose one or more existing declarations in the dialog box, enter new values for any of the declaration fields and use the **Modify** button to update the list. Note that you **MUST** press Modify for the changes to take effect.

Name the signals as shown in the diagram. If you examine the *_package.vhd* file, you will see the line "*SUBTYPE LC2_word IS std_logic_vector(15 downto 0);"*. LC2_word is the type of signal that the ADDRESS and the two DATA buses need to be. For the three buses, type "LC2_word" for the type of signal and set the constraint to none. The bounds fields should be grayed-out when you do this. Hit **Modify** to make the changes and **OK** to close the window.

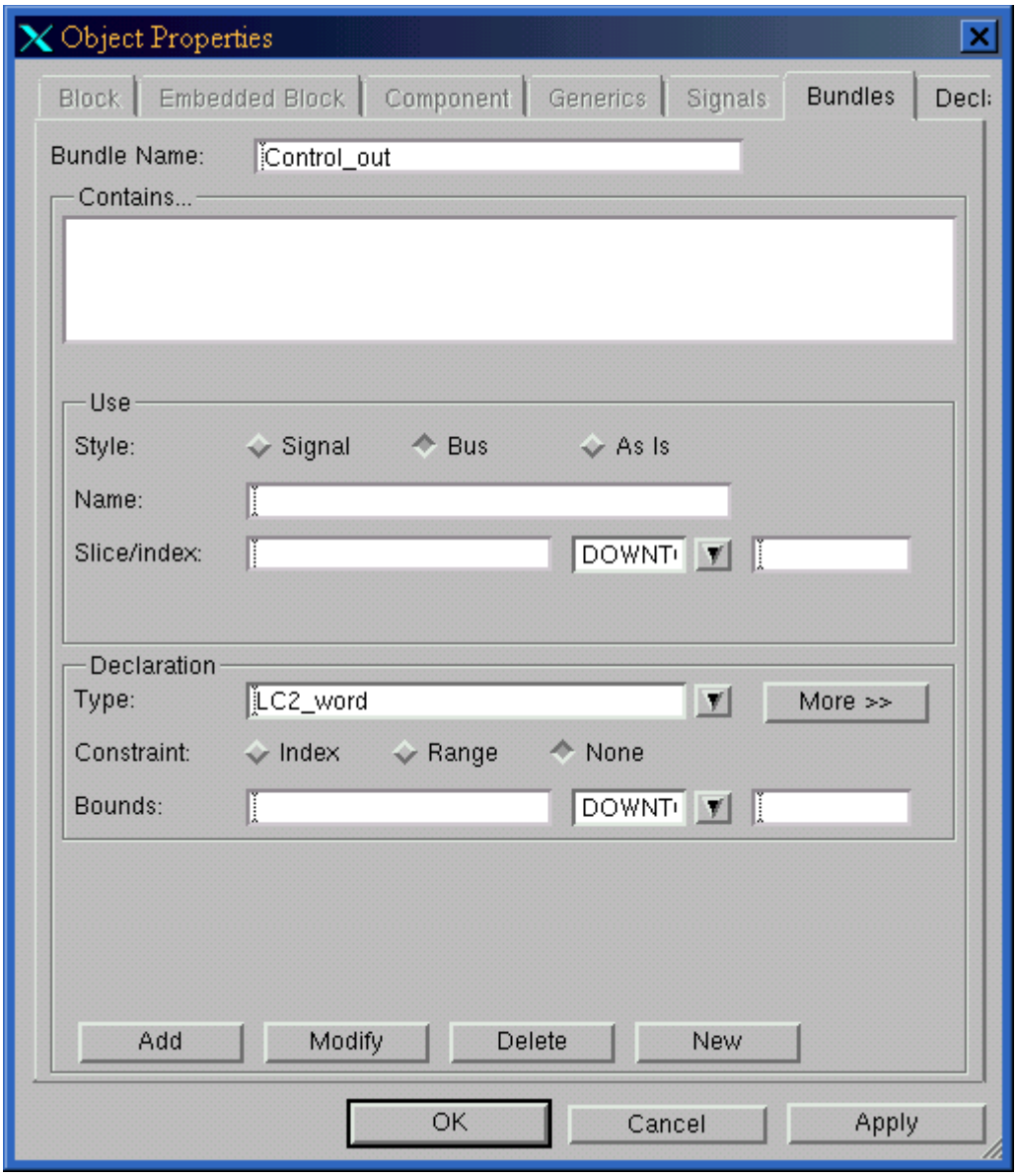
Save when you are done naming the signals. Remember: save early, save often. Renoir is known to crash at times. You have previously saved the diagram so you should not be prompted for library and design unit names. However, you have changed the names of signals connected to input and output ports, so 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.

Add a Bundle and Global Connector

Use the  button to add two bundles connecting the data path block to the control block, one in each direction. Bundles hold groups of signals or buses. These two bundles are used to control the operations performed by the data path. Name the bundle coming out of the Control block “Control_out” and the other bundle “Control_feedback” as shown in the diagram.



Double-click on the Control_out bundle to bring up the Object Properties dialog box.



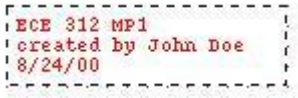
Type ALUop for the name, with type LC2_alu_op, no constraints. Click **Add** to add the signal to the bundle. A list of the remaining signals you need to add to the two bundles is contained in **Appendix D**. You will probably have to reposition components when you are done, as the signal list will be 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**.

Use the  button to add a global connector on your diagram below the bundle. Create a default input port and add a signal between the global connector and the input port. Name this signal "clk." This is a clock signal that is implicitly connected to every block on the diagram.

Add Notes to the Block Diagram

Complete the block diagram by adding text notes describing the diagram.

Use the  button to enter text mode. Click in a free space anywhere on the diagram and enter your notes in the text entry box. Put down a title, your name, and the date similar to this:



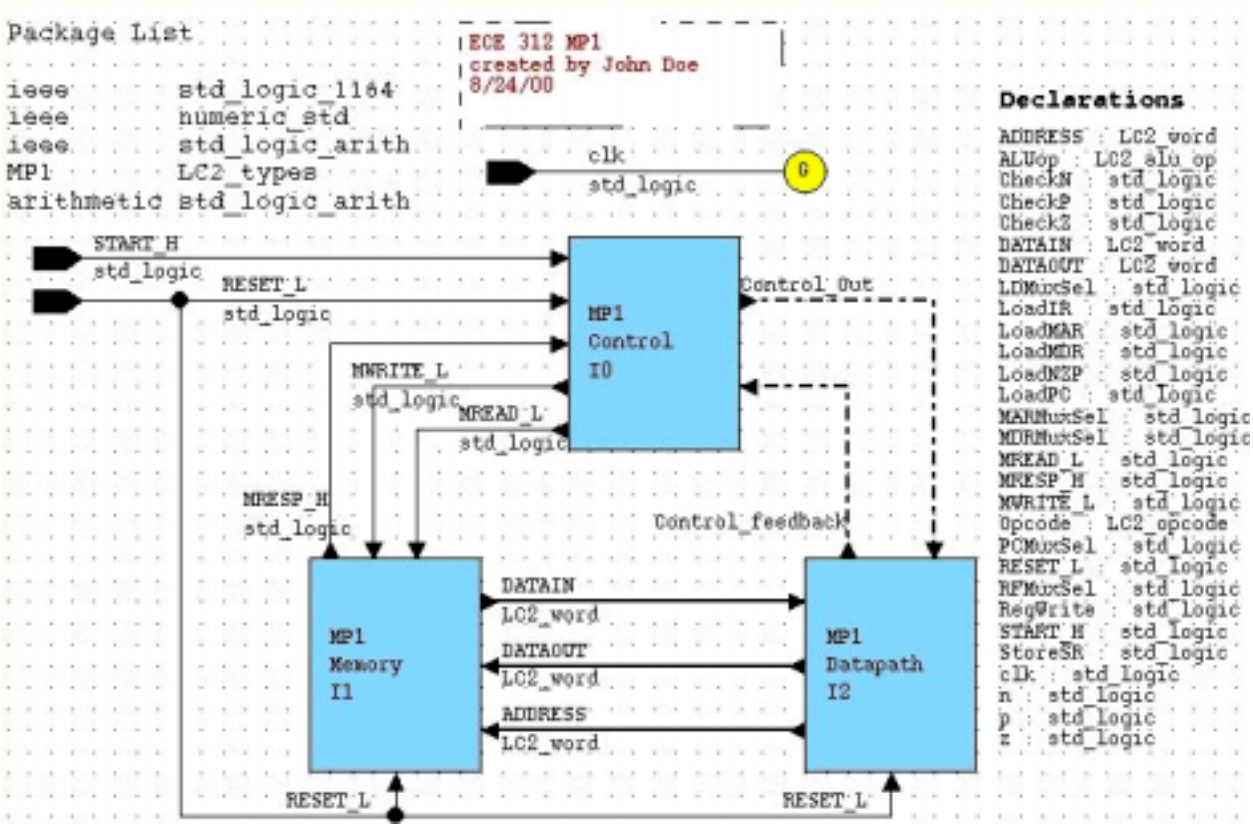
Note: You can enter free-format text including line breaks and spaces. They will be preserved on the diagram.

Click the right mouse button (or click the left mouse button outside the text entry box) to complete the text entry.

Notice that the  button is automatically selected to return to normal object select mode. You can select the notes object and use the object handles to resize and reposition it if necessary.

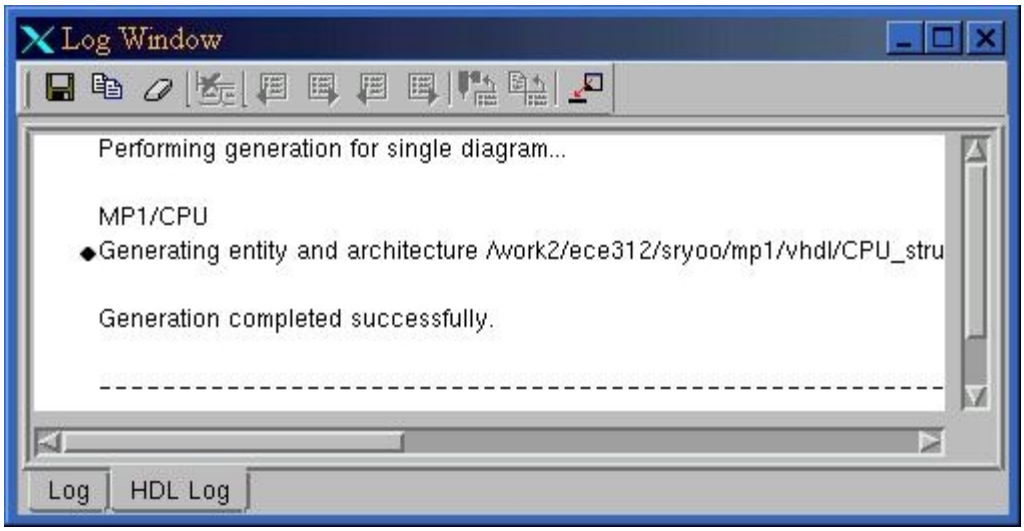
Notes: You can edit an existing notes object by clicking twice on the text. It is not necessary to explicitly enter text mode unless you want to create a new notes object.
You can also edit an existing notes object by choosing the **Comments** tab in the Block Diagram Object Properties dialog box. When notes are edited in the dialog box it is possible to enter any special characters (for example, accents or Kanji characters) that are supported on your system.

Save the block diagram. At this point, your block diagram should look similar to this:



Select **HDL → Set Generate Always** from the menu bar. Choose the **Generate** option from the HDL menu in the state diagram window. Make sure that no components are highlighted, or you will generate HDL for that component, not the CPU.

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 have problems with unrecognized types, make sure that `MP1 LC2_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 (or using the **Graphics** button when the error is selected). If you have errors, this means that something is incorrect. Go over your design to make sure you have followed all the steps correctly.

You can use the **HDL** button to display the generated HDL. If your text editor supports a line number argument, the HDL line corresponding to the error is selected automatically.

Note: You can also view the generated VHDL entity declaration and VHDL architecture body files for the active diagram by using the **View Generated HDL** option from the **HDL** menu in any editor window or when the diagram view is selected in the design browser.

Close the text editors after you have viewed the generated HDL.

Datapath

Next, you will need to design the datapath (also referred to as data path). Double-click on the Datapath block in the CPU view and create a new block diagram that is effectively the same as the one shown in **Appendix E**. Be sure to lay it out as completely as possible, and include the correct names of blocks and signals. Appendix E also contains the list of blocks and signals for you to use as a checklist. Note that *I0…* are the instance names used by the program to keep track of blocks, and you do not need to match them to the diagram.

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 shapes of the blocks, highlight the block, right-click and select **Change Shape** from the popup menu. You can then change the shape of the block to one of many standard shapes.

If you do not want to route a signal or bus very far, you can start a signal/bus in empty space and name it EXACTLY the same as the signal/bus you wish to have. 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.

You can also hide text through the popup menu. It is recommended to do this after you are done with all changes. It has been found that you can minimize errors by first placing the signal, change the name and type, and then hide the type.

As a convention, all std_logic type wires should be signals and all others should be busses.

When you are done with the Datapath diagram, save it and generate to check for syntax errors.

Complete the Block Diagram

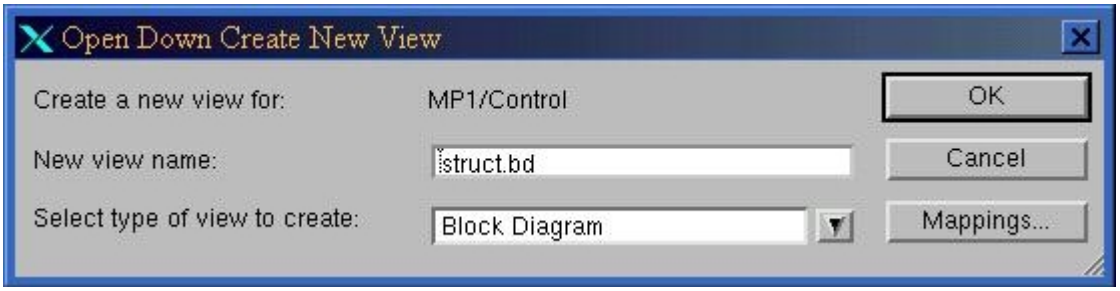
You will need to change several things before you can complete your design. We will next work on the control block, which will contain the necessary logic to control the operation of the data path to correctly implement the LC-2α ISA.

Create a State Machine (Control)

The control unit for this design will be created using a state machine. For the non-pipelined implementation of LC-2, this is an effective way of showing the actual function of the design.

Create a State Diagram

Move the cursor over the body of the *Control* block on the *CPU* block diagram, then press and release the right mouse button to select the block and display the popup menu. Choose the **Open → New View**. The Open Down Create New View dialog box is displayed:



Use the pull-down list to select the type of view you want to create. The view name defaults to `struct.bd` (for a block diagram), `flow.fc` (for a flow chart), `fsm.sm` (for a state machine), `table.tt` (for a truth table) or `untitled` (for a VHDL architecture) view. It is recommended to keep these names.

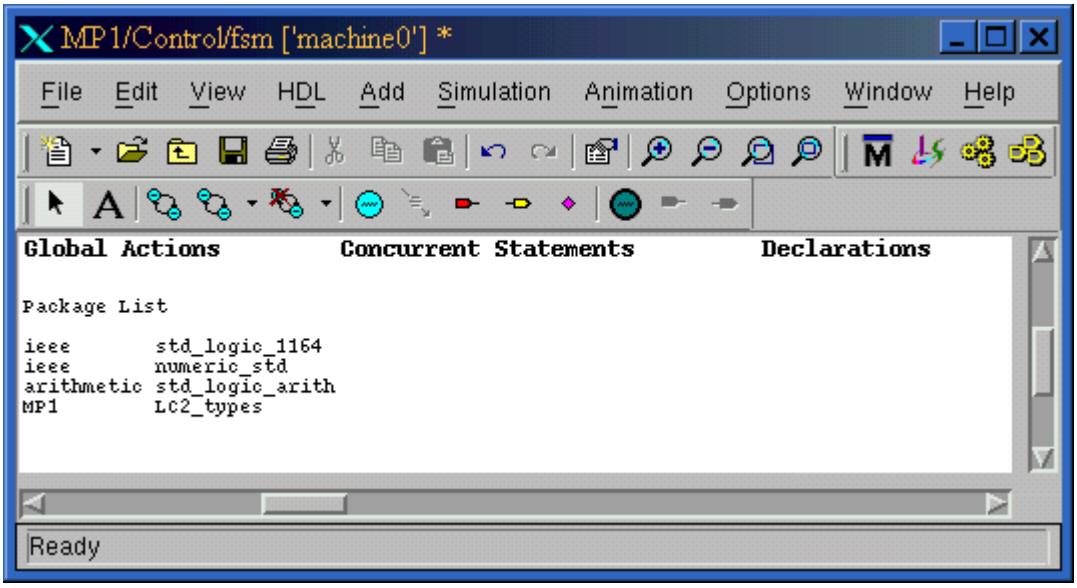
Note: You need not enter the two-character extension (`.bd, .fc, .sm or .tt`) for graphical views, as the correct extension is automatically added. However, if you do enter any other extension, the file will not be recognized.

Select **State Diagram** from the pull-down list of views you can create. Use the default view name `fsm.sm`. Click **OK** to continue.

Note: Solid handles are displayed when the body of a block (or other re-sizable object) is selected. You can display the Open Down Create New View dialog box directly by

double-clicking on the body of a block that has no views defined. However, if you click directly over a text object (such as the block name), hollow handles indicate that the text is selected and clicking again allows you to edit the text.

A new state diagram (`MP1/Control/fsm`) is created as a child view of the *Control* block:

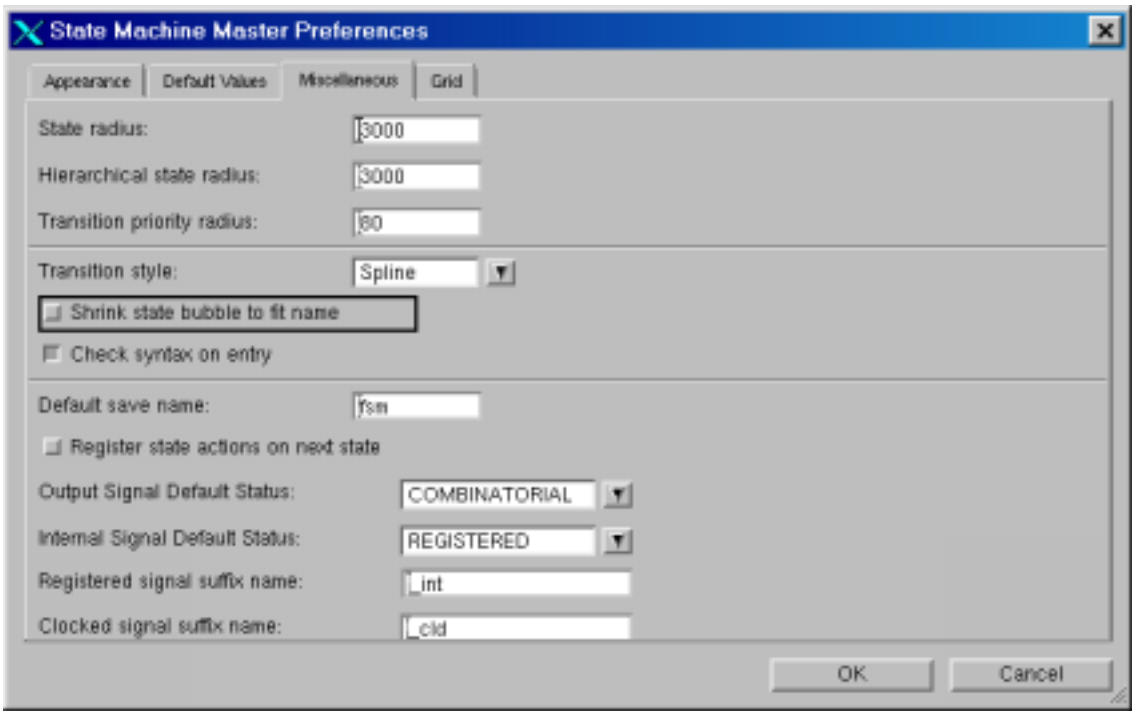


The state diagram is a blank sheet except for the default VHDL package list (which should include `MP1 LC2_types`) and labels for global actions, concurrent statements, architecture declarations and signals status.

Set State Machine Preferences

You will create a state machine view in the next procedure. It is possible to set preferences which affect the way new diagrams are drawn. In particular, for this tutorial it is recommended that you reduce the default size used the draw states on a state diagram.

In the Design Browser window, choose **Options → Edit Master Preferences → State Machine** to display the State Machine Preferences dialog box and select the **Miscellaneous** tab:



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.

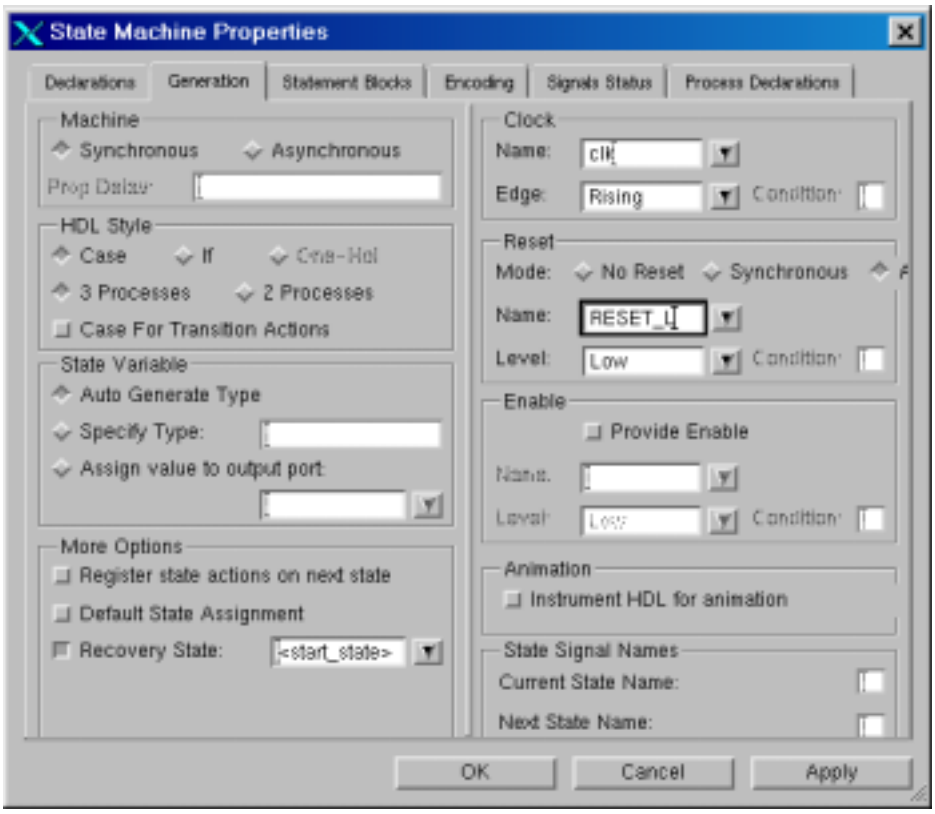
Change the state radius and hierarchical state radius values to 3000. This radius should be sufficient to enclose the state names used in this machine problem. We do not recommend changing any of the other preferences.

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

Notes: You can use the **Appearance** tab in the State Machine Preferences dialog box to change the default visual attributes used for objects on a new diagram or change the attributes for individual objects on an active diagram by choosing **Appearance** from the **Edit** menu in the editor window.
You can use the **Default Values** tab to change the default names used for objects on a state diagram and specify default values for signals, transitions and actions**.**
You can use the **Grid** tab to control whether a grid is displayed in your diagram.

Set HDL Generation Characteristics

Next, go back to the Control window and select **Edit → State Machine Properties**. A new dialog box will appear: choose the **Generation** tab. This tab sets the generation characteristics used for HDL generation.



Choose the **Synchronous** button in the Machine box.

Use the default **Case** option for HDL Style (leaving **Case for Transition Actions** unset) and **Auto Generate Type** for the State Variable.

Check that the **Recovery State** is set to <*start_state*>. (This entry automatically assigns the start state or you can explicitly enter a state name.) Leave the **Register state actions on next state** and **Default State Assignment** options unset.

Check that **Rising** is selected for the clock and enter the signal name as *clk*. Set the reset as **Asynchronous** and **Low**, entering the signal name as *RESET_L*.

Hit **OK** to set the properties. Save the state diagram.

Add States and Transitions

Use the  button to add nine (9) states on your state diagram. The states are added with default names *s0*, *s1*, etc. Notice that the first state you add is assumed to be the start state and is drawn in green with a double outline.

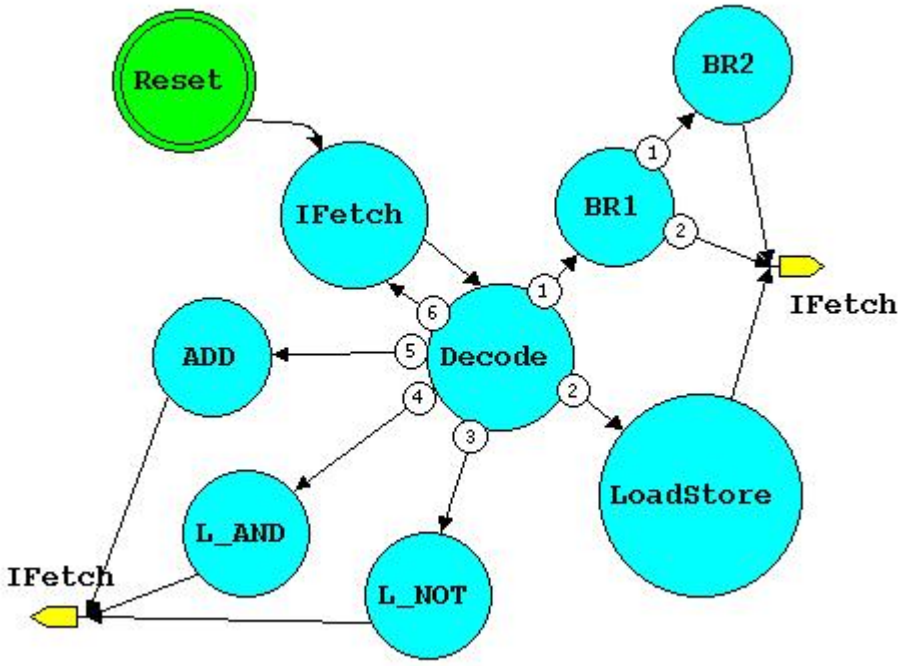
Use the  button to add transitions between the states as shown in the diagram on the next page. Click on the edge of the state you wish to start from, then click on the edge of the state you wish to go to. Notice that when you add more than one transition leaving a state, a number associated with the transition arc indicates the transition priority. The priorities for the transitions

are initially assigned in the order that you add the transitions. If you have a "default" transition, a transition taken when no other transitions' conditions are satisfied, it must have the lowest priority (highest number) of the transitions leaving that state. This is because transitions are checked by priority, 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 popup or **Edit** menu.

Next, use the  button 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 the diagram below. Name your states exactly as shown. Remember that default transitions must have the lowest priority of the transitions leaving that state.

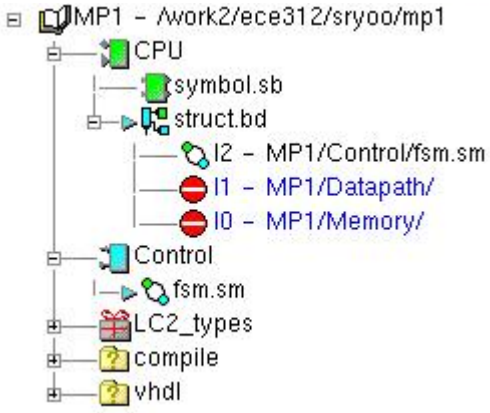


Saving the State Diagram

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 its was created. The design browser view is updated to display the *Control* design unit.

Note: You can pop the design browser window to the front by selecting it from the **Windows** menu list in an editor window.

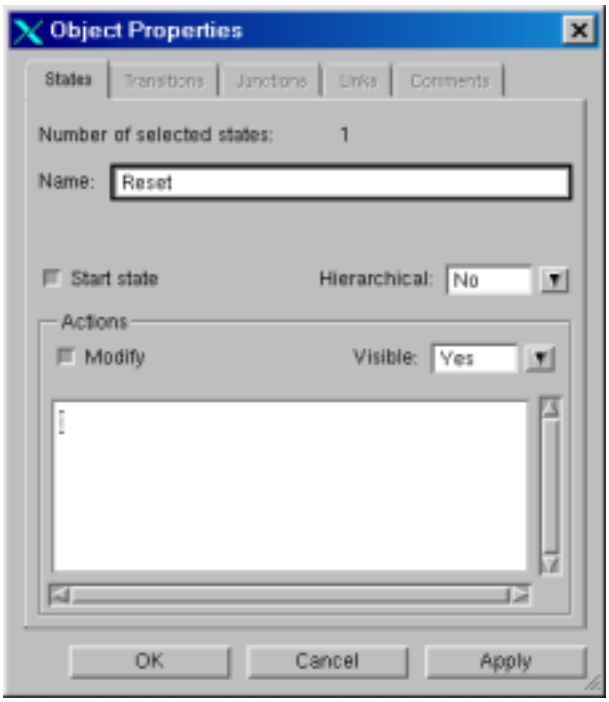
The *Control* design unit is shown as a block in the browser because its interface is defined by the connections on its parent block diagram. The *CPU* design unit is shown as a component because it has no parent block diagram as its interface is defined by a symbol. Expand the *Control* design unit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



Notice also that the icon used for instance *I2* (the Control unit) in the hierarchy for the *struct.bd* view has changed to , indicating that it is now described by a state diagram. If the hierarchy is not already displayed, click on the ⊞ icon for the *struct.bd* view.

Edit the States

Select the Reset state (green with the double outline) and use the button to display the **States** tab of the State Machine Object Properties dialog box. (Alternatively, you can display the dialog box by choosing **Edit → Object Properties**, right-clicking and selecting **Object Properties**, or double-clicking on the state.)



The **States** tab 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 start state or a hierarchical state.

Under the section Actions, you will be inputting things that the processor should be doing during that state, such as selecting multiplexer inputs or enabling register loads.

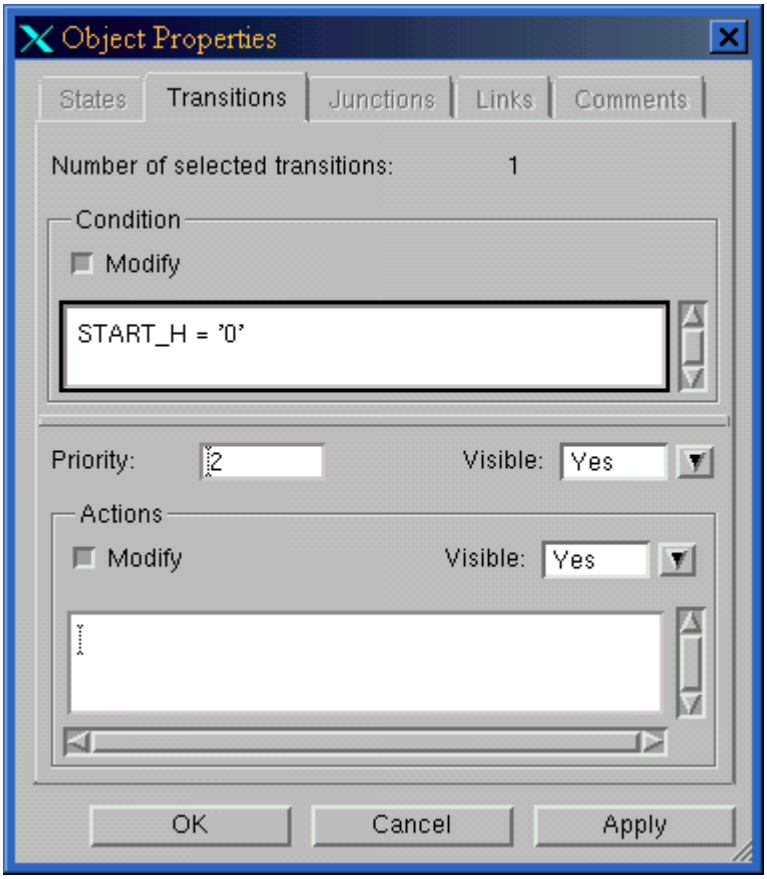
Use the dialog box to make the *IFetch* and *LoadStore* states **hierarchical**. They should be redrawn with a triple outline and a darker blue color. You will be modifying the actions of the states at a later time.

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 fit 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.

Editing Transitions

A state machine isn't of much use if the processor does not know which transition to take. Draw a new transition originating and ending at the Reset state. Double-click on the transition. It will bring up the following window:



In this screen, you can modify both conditions (when to take that transition) or actions (what to do during the branch). We will not be modifying actions for transitions in this MP, nor is it required.

For this transition, your condition should be START_H = '0'. This means that the processor should remain in the reset state as long as START_H = '0'. Note that there is no semicolon after conditions, only after actions.

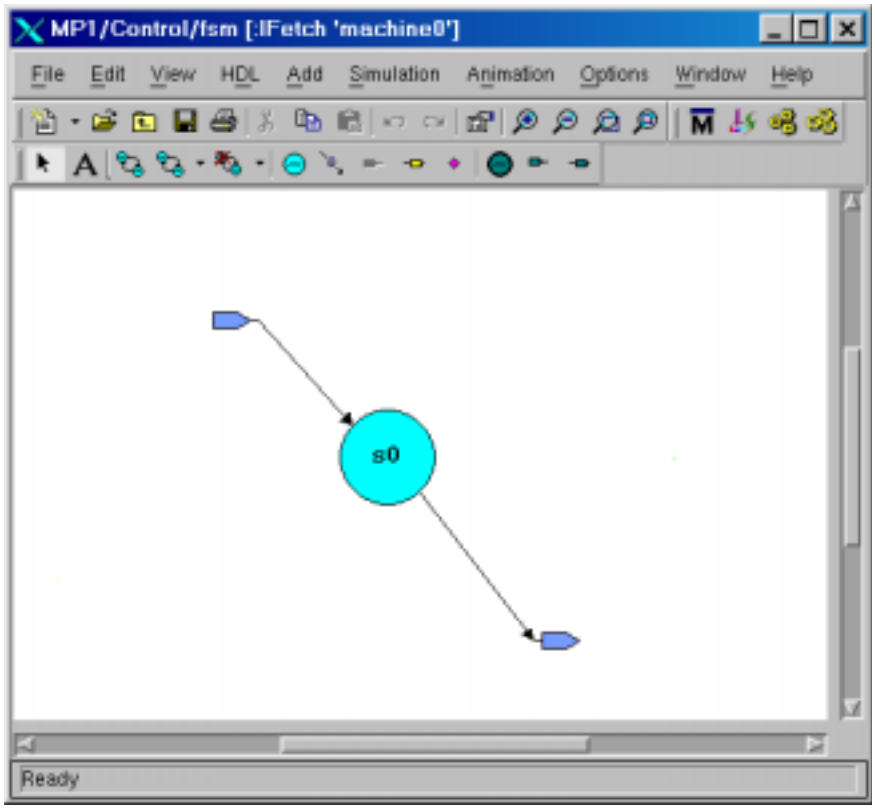
Creating a Hierarchical State

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 also select the state and choose **Edit → Change To → Hierarchical State**.

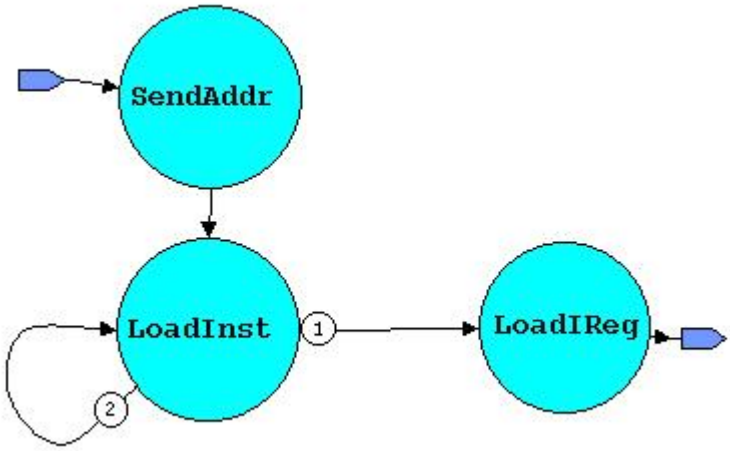
Double-click on the *IFetch* state (or use the **Open Down** option from the right-button popup menu) to create the new hierarchical child state diagram. A new child state diagram window is initialized with a single state connected to an entry point and exit point.

Notice that the child diagram is named `MP1\Control\fsm[:IFetch]` indicating that it is a partial view of the parent diagram.



Edit the Hierarchical State Diagram

Create two more states on the diagram. Rename the states and use the  button to connect transitions between the states as shown in the following picture:



Remember to save the state diagram.

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 (`MP1/Control/fsm.sm`), although the name in the title bar is `MP1/Control/fsm:IFetch`. The upshot of this is that you need to change the IFetch links to point to the SendAddr state. This is because IFetch technically is no longer a state. Otherwise, no other changes are required.

Hide Global Actions and Concurrent Statements

Use the  button (or choose **File → Open Up**) to redisplay the parent state diagram if it is not already displayed.

Double-click the left mouse button over the Global Actions or 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 unsetting the **Visible** checkbox for Global Actions and Concurrent Statements in the **Statements Blocks** tab. Hit **OK** when you are done.

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. **Appendix F** 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 states of certain signals. Default states are the states that output signals automatically have when you do not specify them in a state. This happens for each and every state. In particular you cannot have default values of '0' on active-low signals to memory. To change default values, double-click on **Signal Status** in the design view, which will bring up the State Machine Properties window. You will need to change the default values of three signals. The signals and their default values are listed in **Appendix F**.

VHDL

You have now completed the initial phase of the design of our LC-2α 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.

Input HDL

Thankfully, we have provided most of the HDL you require in mp1files.tar.gz. It was uncompressed into the `mp1/vhdl` subdirectory. All you need to do is cut and paste the code into the various data path blocks and the memory block. Alternatively, you can copy the vhdl files over each untitled.vhd file in the appropriate component directory. The untitled.vhd file will not be created unless you have already double-clicked on a component and chosen **VHDL Architecture**. This is detailed in the next section. You will need to make sure that the filenames within the VHDL files we provide match the ones you give them when you first declare them.

Not all of the VHDL has been provided. Please check the web page for information on which components you must write VHDL for.

Create a Child HDL View

Double-click over the body of the *Memory* block in the *CPU* view to display the Open Down Create New View dialog box. Select **VHDL Architecture** from the pull-down list of views. You can leave the default name as *untitled*. Hit **OK**.

A template HDL view is opened using the default text editor for your workstation. Save and close the HDL view.

Now do the same with all of the blocks in the Datapath. VHDL files have corresponding names with the blocks in the Datapath.

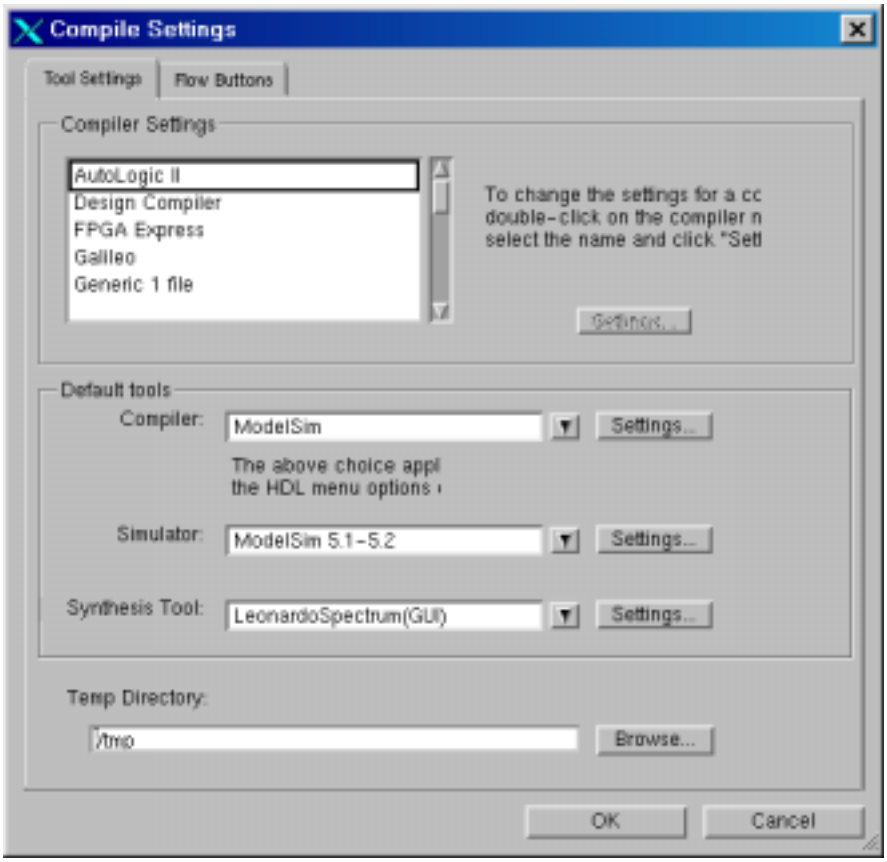
You should now be completely done with the design. To check for errors, go to the highest-level view, which is the *CPU* view, and select **HDL → Generate Through Components**. As always be sure that **HDL → Set Generate Always** is checked. Most errors are relatively well explained by the program. As before, if you have excessive difficulty isolating an error, please see a TA.

Note: Before compiling and simulation can commence, instructions in the form of memory vectors will need to be pasted into the memory block's vhdl file. You will be required to write a small amount of assembly code to perform a specific operation. Detailed information on how to accomplish this will be provided on the web page.

Chapter 5: Compiling and Simulation

Compile

First, choose **Options → Compile Settings**. A new dialog box should appear.



Set ModelSim as your compiler, ModelSim 5.1 – 5.2 as your simulator, and LeonardoSpectrum(GUI) as your synthesis tool. Your temporary directory can remain /tmp. Settings for the tools should not be changed. Click **OK** when you have completed your changes.

Before compiling any design in this and future MPs, select **HDL → Set Generate Always**. Renoir occasionally has difficulty incrementally compiling the vhdl and this option ensures that all vhdl is completely recompiled.

Select the *CPU* design unit in the design browser and choose **HDL → Generate Through Components**. The log box 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 or a teaching assistant for assistance. Continue when you can generate without error messages.

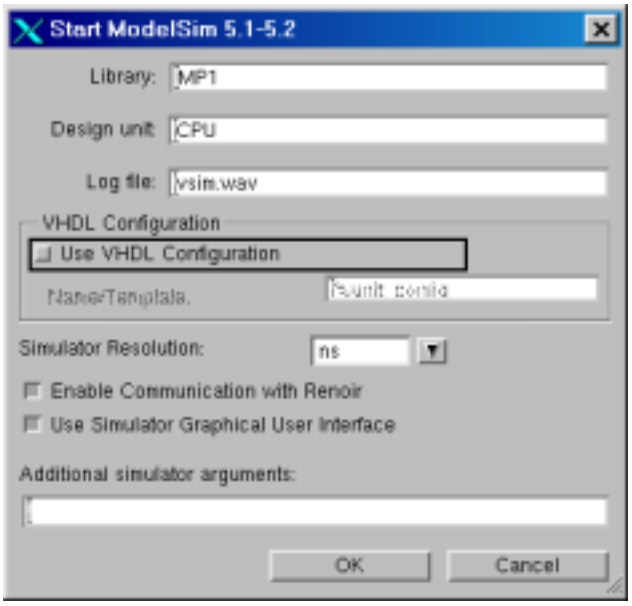
Next, make sure the *CPU* design unit is still selected, and choose **HDL → Compile Through Components**. Ignore the "Note: Cannot find generated package body LC2_types in library MP1" statement when it appears (it is due to somewhat improper syntax in the package file). Again, if you have any error messages, check your design and resolve any errors. You will need to regenerate HDL before recompiling.

Help: If nothing seems to be working, try deleting all files in the HDL window and regenerating and recompiling. Renoir's method of tracking files sometimes misses the changes you make to diagram and VHDL. Also, make sure that you've saved all of your changes. We've put a few errors into **Appendix B** to help you get started: you can list your own in there as you find them, to help you during future MPs.

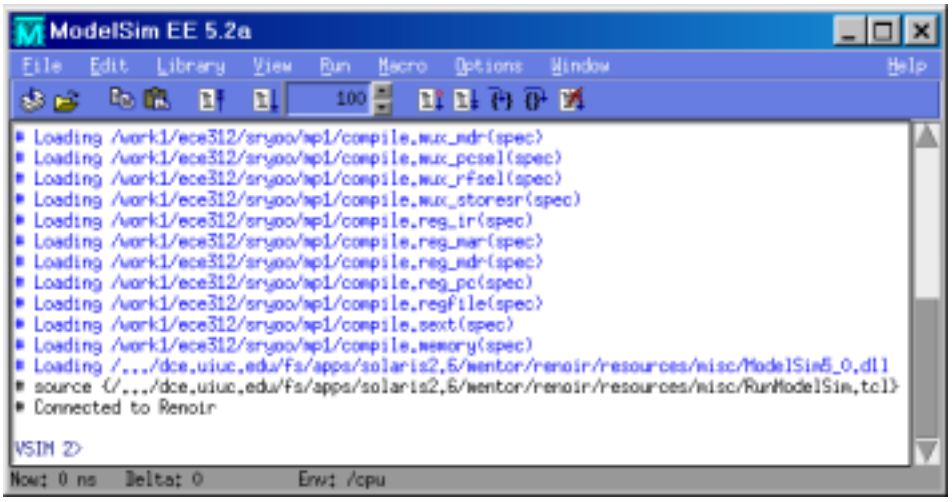
Simulation

Using ModelSim

Select the *CPU* design unit in the design browser and select **HDL → Start Simulator** to invoke the ModelSim simulator on your entire design. A new dialog box will appear:



Make sure that the MP1 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 the message:

```
Connected to Renoir
```

You are now ready to use ModelSim.

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`, where `macro_filename` is the name of the macro file.

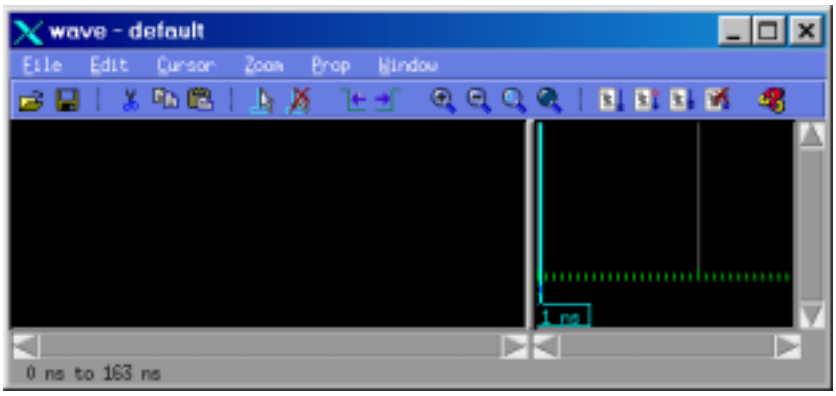
Note: You may need to change the permissions of files before executing them. This will definitely be the case in later machine problems, where you will need to write your own assembly code. If you do not know how to do this already, contact a TA.

In the ModelSim window type `do start.do`. This file has commands that establish a 30 ns clock and the initialization pattern for the processor. You will need to run this file every time you restart the simulator.

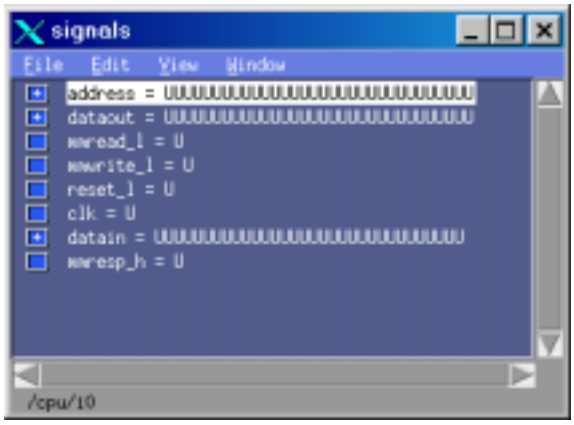
Traces and Lists

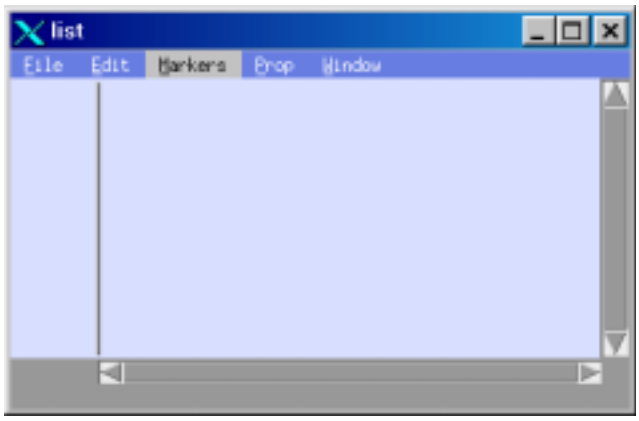
There are multiple ways of viewing the function of your design. The two methods used here are wave traces and lists.

In the ModelSim window select **View → Wave**. Alternatively, you can type `view wave` at the prompt. The new window should look like this:



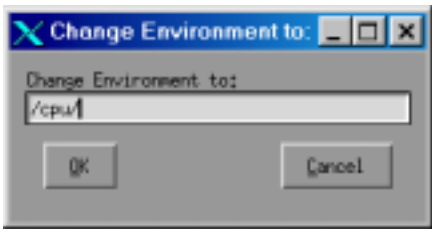
Do the same for the **Signals** and **List** options. You should have something similar to the following windows:





Drag all available signals to the wave window. In the wave window, note that the LC2_word values are displayed as 16-bit values, which can be difficult to read. Change the display to hexadecimal values by selecting the signal and selecting **Prop → Radix → Hexadecimal**. You can also change the displayed name of the signal by selecting **Prop → Signal Props**. 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 **File → Environment → Fix to Content** in the signals window. The Change Environment To dialog box will appear.



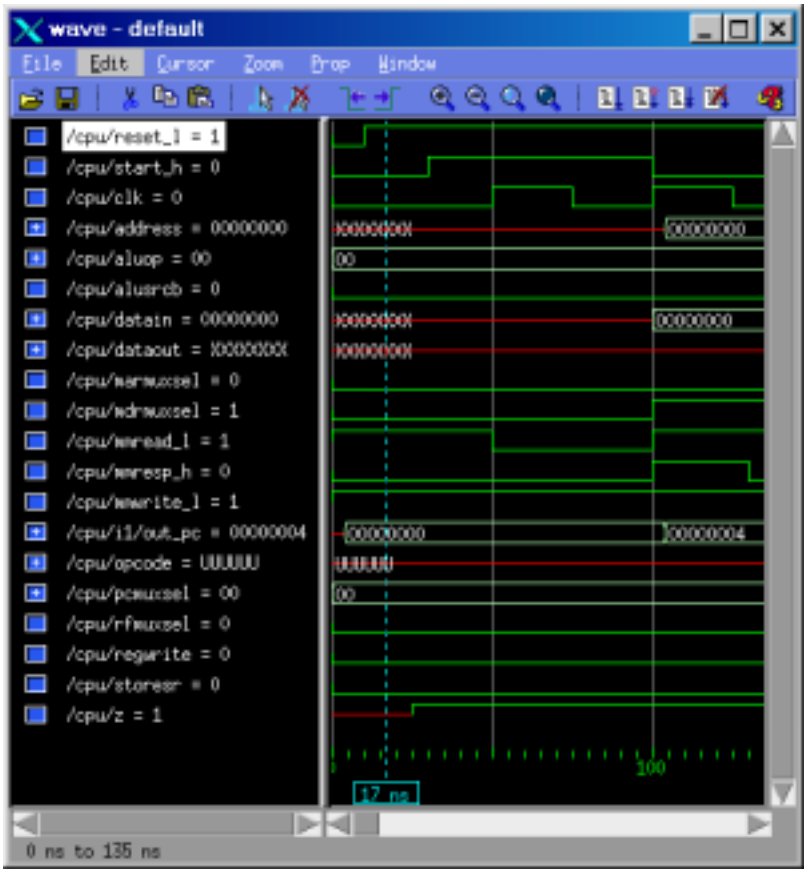
Refer to your design for the appropriate instance (I0, I1, or I2) of the Datapath block. In this case, the environment desired is `/cpu/i1`. Select **OK** and drag the signal *Out_PC* to the wave

window. You can now see the signals in the data path. You may need to go to individual units in the data path using the same method as described to obtain all the signals you want to view.

Although it is not necessary in this MP, you may want to see the variables in the design as well. If you have reviewed the VHDL in the design, variables are often used as temporary values before sending the desired values onto the signals. The easiest way to differentiate the two is that variables are used in code, while signals are the physical lines that can be viewed in block diagrams of the design. They act in different ways within designs, and cannot be directly operated together (for example, you cannot add a signal and variable). It is not likely that any issues will appear during MP1, but this is something to look out for in future MPs.

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 Format**. 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.

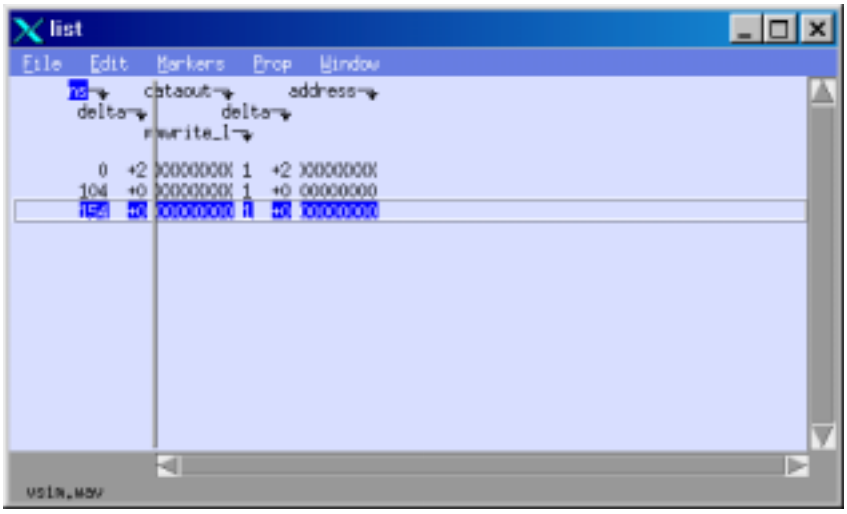
You are now ready to run a simulation of the design. In the ModelSim window type `run 2000`. This will run the design for 2000 ns. Your wave window should look similar to the following:



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, restart the simulation by typing `restart` in the ModelSim window. Optionally, the –f argument can be included with `restart` to bypass the dialog box.

We have not yet discussed the list window. Drag the address, dataout, and mwrite_l signals to the list window. Change the signal properties (either right-click or go to the **Prop** drop-down menu) so that LC2_word values are in hexadecimal. Rename the signals to save space and make them easier to read. You should have something similar to the following:



You now have all the tools necessary to observe the operation of your design.

It is likely that there are errors in the design included in this section of the tutorial. For example, you may have noticed that the data values shown in the pictures are 32 bits long (a carryover from previous semesters). Errata and corrections will be posted to the course web page and newsgroup.

Chapter 6: Final Hand-In

All hand-in requirements are listed on the web page.

If you are having difficulty understanding an aspect 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.

Appendix A: 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
ld	0010	Load the register specified by DR from the location specified by pgoffset9 on the same page as the LD instruction
not	1001	Put the bitwise complement of register SR into register DR.
st	0011	Store the word from register *rt* at the effective address

Appendix B: 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.

"LC2_word is not a valid type."
Try deleting all files in the HDL window of the Design Browser. If that does not help, try re-linking the _package.vhd file for LC2_types.

In general, if something doesn't work, do a "make clean." Delete all files in the HDL window of the Design Browser, then recompile through components. Make certain that **HDL → Set Generate Always** is enabled. Incremental compilations can cause this type of error even if the vhdl is syntactically correct.

Make sure the CPU module is selected in the Source window of the Design Browser when you generate and compile. Your design may not be simulating because you haven't generated and compiled all the components.

Appendix C: RTL

FETCH Process

State	Data	Control
SendAddr	MAR ← PC; PC ← PC + 1;	MARMuxSel <= ‘0’; LoadMAR <= ‘1’; PCMuxSel <= ‘0’; LoadPC <= ‘1’;
LoadInst	While (!Ready) {MDR ← M[MAR];}	MDRMuxSel <= ‘1’; LoadMDR <= ‘1’; MREAD_L <= ‘0’ after 6 ns;
LoadIR	IR ← MDR;	LoadIR <= ‘1’;

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)

ADD Instruction

Data	Control
FETCH	
DECODE	
DR ← A + B;	ALUop <= “00”; LDMuxSel <= ‘0’; LoadNZP <= ‘1’; RFMuxSel <= ‘1’; RegWrite <= ‘1’; StoreSR <= ‘0’;

AND Instruction

Data	Control
FETCH	
DECODE	
DR ← A & B;	ALUop <= “01”; LDMuxSel <= ‘0’; LoadNZP <= ‘1’; RFMuxSel <= ‘1’; RegWrite <= ‘1’; StoreSR <= ‘0’;

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 SendAddr
BR2	PC ← PC[15:9] || IR[8:0];	PCMuxSel <= ‘1’; LoadPC <= ‘1’;

LD Instruction

State	Data	Control
FETCH		
DECODE		
CalcAddr	MAR ← PC[15:9] || IR[8:0];	MARMuxSel <= ‘1’; LoadMAR <= ‘1’;

LD1	While (!Ready) {MDR← M[MAR];}	While (!Ready) {MDRMuxSel <= ‘1’; LoadMDR <= ‘1’; MREAD_L <= ‘0’ after 6 ns;}
LD2	DR ← MDR;	ALUop <= “11”; LDMuxSel <= ‘1’; LoadNZP <= ‘1’; RFMuxSel <= ‘0’; RegWrite <= ‘1’;

NOT Instruction

Data	Control
FETCH	
DECODE	
DR ← !A	ALUop <= “10”; LDMuxSel <= ‘0’; LoadNZP <= ‘1’; RFMuxSel <= ‘1’; RegWrite <= ‘1’; StoreSR <= ‘0’;

ST Instruction

State	Data	Control
FETCH		
DECODE		
CalcAddr	MAR ← PC[15:9] || IR[8:0];	MARMuxSel <= ‘1’; LoadMAR <= ‘1’;
ST1	MDR ← DR;	ALUOp <= “11”; MDRMuxSel <= ‘0’; LDMuxSel <= ‘1’; LoadMDR <= ‘1’; StoreSR <= ‘1’;
ST2	While (!Ready) {M[MAR]← MDR;}	While (!Ready) {MWRITE_L <= ‘0’ after 6 ns;}

Appendix D: CPU

Control_out Bundle Signals

Signal Name:	Type:
ALUop	LC2_alu_op
LoadIR	std_logic
LoadMAR	std_logic
LoadMDR	std_logic
LoadNZP	std_logic
LoadPC	std_logic
LDMuxSel	std_logic
MARMuxSel	std_logic
MDRMuxSel	std_logic
PCMuxSel	std_logic
RFMuxSel	std_logic
RegWrite	std_logic
StoreSR	std_logic

Control_feedback Bundle Signals

Signal Name:	Type:
CheckN	std_logic
CheckP	std_logic
CheckZ	std_logic
Opcode	LC2_opcode
n	std_logic
p	std_logic
z	std_logic

Appendix E: Datapath

Blocks

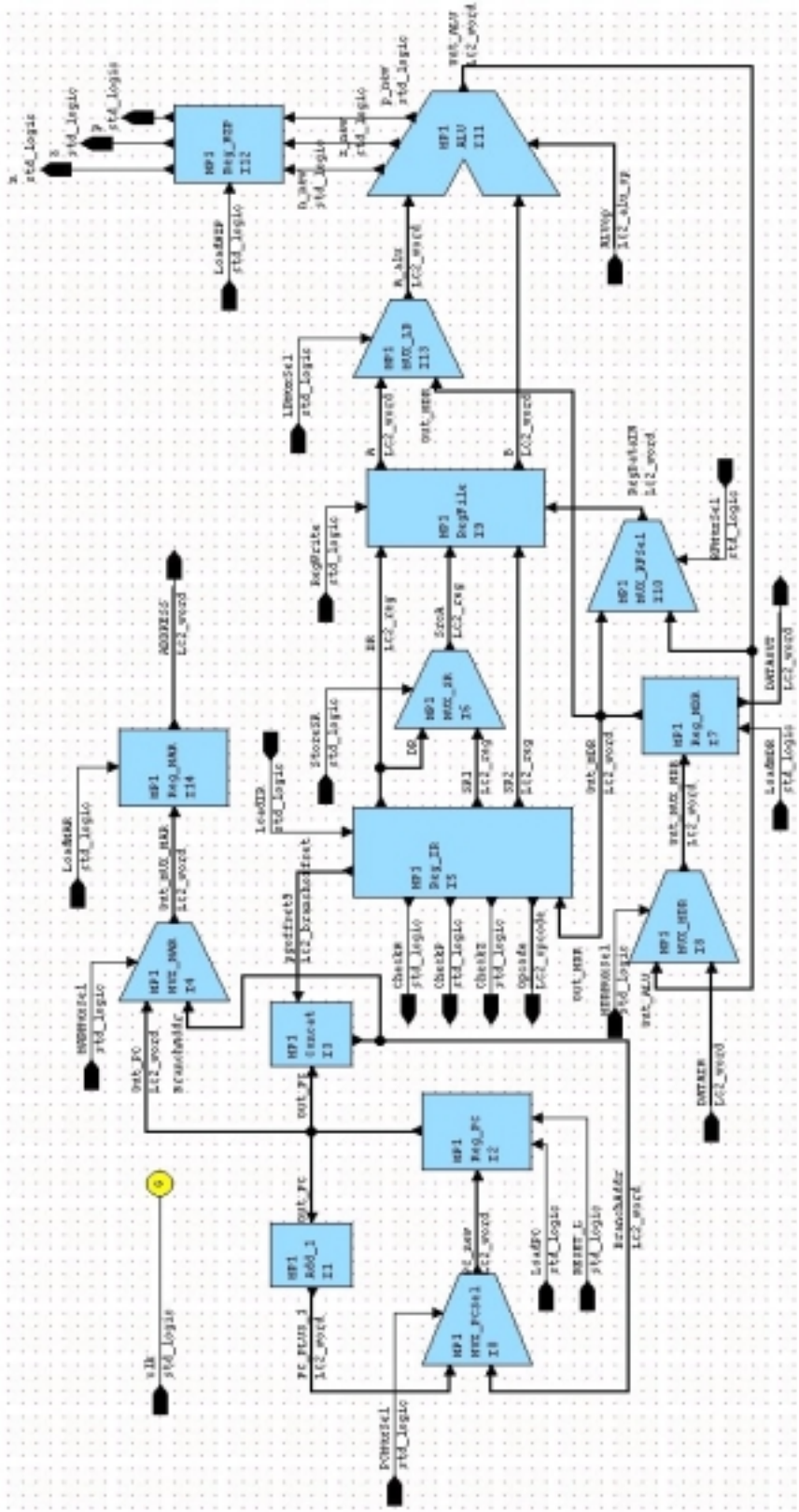
ADD_1	Reg_MDR	MUX_RFSel	RegFile
ALU	MUX_PCSel	MUX_StoreSR	REG_PC
Concat	MUX_MAR	Reg_IR	MUX_LD
Reg_MAR	MUX_MDR	Reg_NZP	

Signals

Signal Name:	**Type:**	**Origin Block:**	**Destination Block(s):**
A	LC2_word	RegFile	MUX_LD
A_alu	LC2_word	MUX_LD	ALU
ADDRESS	LC2_word	Reg_MAR	*Port Out*
ALUop	LC2_alu_op	*Port In*	ALU
B	LC2_word	RegFile	ALU
BranchAddr	LC2_word	Concat	MUX_PCSel, MUX_MAR
CheckN	std_logic	Reg_IR	*Port Out*
CheckP	std_logic	Reg_IR	*Port Out*
CheckZ	std_logic	Reg_IR	*Port Out*
clk	std_logic	*Global*	*Global*
DATAIN	LC2_word	*Port In*	MUX_MDR
DATAOUT	LC2_word	Reg_MDR	*Port Out*
DR	LC2_reg	Reg_IR	RegFile, MUX_SR
LDMuxSel	std_logic	*Port In*	MUX_LD
LoadIR	std_logic	*Port In*	Reg_IR
LoadMAR	std_logic	*Port In*	Reg_MAR
LoadMDR	std_logic	*Port In*	Reg_MDR
LoadNZP	std_logic	*Port In*	Reg_NZP
LoadPC	std_logic	*Port In*	Reg_PC
MARMuxSel	std_logic	*Port In*	MUX_MAR
MDRMuxSel	std_logic	*Port In*	MUX_MDR
N	std_logic	Reg_NZP	*Port Out*
N_new	std_logic	ALU	Reg_NZP
Opcode	LC2_opcode	Reg_IR	*Port Out*
Out_ALU	LC2_word	ALU	MUX_MDR, MUX_RFSel
Out_MDR	LC2_word	Reg_MDR	Reg_IR, MUX_RFSel, MUX_LD
Out_MUX_MAR	LC2_word	MUX_MAR	Reg_MAR
Out_MUX_MDR	LC2_word	MUX_MDR	Reg_MDR
Out_PC	LC2_word	REG_PC	ADD_1, Concat, MUX_MAR
p	std_logic	Reg_NZP	*Port Out*
p_new	std_logic	ALU	Reg_NZP
pgoffset9	LC2_branchoffset	Reg_IR	Concat

PC_plus_1	LC2_word	ADD_1	MUX_PCSel
PC_new	LC2_word	MUX_PCSel	Reg_PC
PCMuxSel	std_logic	*Port In*	MUX_PCSel
RESET_L	std_logic	*Port In*	Reg_PC
RFMuxSel	std_logic	*Port In*	MUX_RFSel
RegDataIn	LC2_word	MUX_RFSel	RegFile
RegWrite	std_logic	*Port In*	RegFile
SR1	LC2_reg	Reg_IR	MUX_SR
SR2	LC2_reg	Reg_IR	RegFile
SrcA	LC2_reg	MUX_SR	RegFile
StoreSR	std_logic	*Port In*	MUX_SR
z	std_logic	Reg_NZP	*Port Out*
z_new	std_logic	ALU	Reg_NZP

Datapath Diagram



Appendix F: Control

States

Name:	Action:
ADD	ALUop <= “00”; LoadNZP <= ‘1’; RFMuxSel <= ‘1’; RegWrite <= ‘1’; StoreSR <= ‘0’; LDMuxSel <= ‘0’;
BR1	*NONE*
BR2	PCMuxSel <= ‘1’; LoadPC <= ‘1’;
CalcAddr	MARMuxSel <= ‘1’; LoadMAR <= ‘1’;
Decode	*NONE*
IFetch	*(contains SendAddr, LoadInst, and LoadIReg states)*
L_AND	ALUop <= “01”; LoadNZP <= ‘1’; RFMuxSel <= ‘1’; RegWrite <= ‘1’; StoreSR <= ‘0’; LDMuxSel <= ‘0’;
L_NOT	ALUop <= “10”; LoadNZP <= ‘1’; RFMuxSel <= ‘1’; RegWrite <= ‘1’; StoreSR <= ‘0’; LDMuxSel <= ‘0’;
LD1	MDRMuxSel <= ‘1’; LoadMDR <= ‘1’; MREAD_L <= ‘0’ after 6 ns;
LD2	LoadNZP <= ‘1’; RFMuxSel <= ‘0’; RegWrite <= ‘1’; LDMuxSel <= ‘1’; AluOP <= “11”;
LoadIReg	LoadIR <= ‘1’;
LoadInst	MDRMuxSel <= ‘1’; LoadMDR <= ‘1’; MREAD_L <= ‘0’ after 6 ns;
LoadStore	*(contains CalcAddr, LD1, LD2, ST1, ST2)*
Reset	*NONE*
ST1	ALUop <= “11”; MDRMuxSel <= ‘0’; LoadMDR <= ‘1’; StoreSR <= ‘1’;
ST2	MWRITE_L <= ‘0’ after 6 ns;
SendAddr	MARMuxSel <= ‘0’; LoadMAR <= ‘1’; PCMuxSel <= ‘0’; LoadPC <= ‘1’;

State Transitions

Main Diagram

Origin State	Destination State/Link	Condition
ADD	SendAddr (Link)	*NONE*
BR1	BR2	((n AND CheckN) OR (p AND CheckP) OR (z AND CheckZ)) = ‘1’
BR1	SendAddr (Link)	*(lowest priority: 2)*
BR2	SendAddr (Link)	*NONE*
Decode	ADD	Opcode = op_add
Decode	BR1	Opcode = op_br
Decode	IFetch	*(lowest priority: 6)*
Decode	L_AND	Opcode = op_and
Decode	L_NOT	Opcode = op_not

Decode	LoadStore	(Opcode = op_ld) OR (Opcode = op_st)
IFetch	Decode	*NONE*
L_AND	SendAddr (Link)	*NONE*
L_NOT	SendAddr (Link)	*NONE*
LoadStore	SendAddr (Link)	*NONE*
Reset	IFetch	START_H = ‘1’
Reset	Reset	START_H = ‘0’

IFetch Hierarchical State

Origin State	**Destination State/Link**	**Condition**
Entry Point	SendAddr	*NONE*
LoadInst	LoadInst	MRESP_H = ‘0’
LoadInst	LoadIReg	MRESP_H = ‘1’
LoadIReg	*Exit Point*	*NONE*
SendAddr	LoadInst	*NONE*

LoadStore Hierarchical State

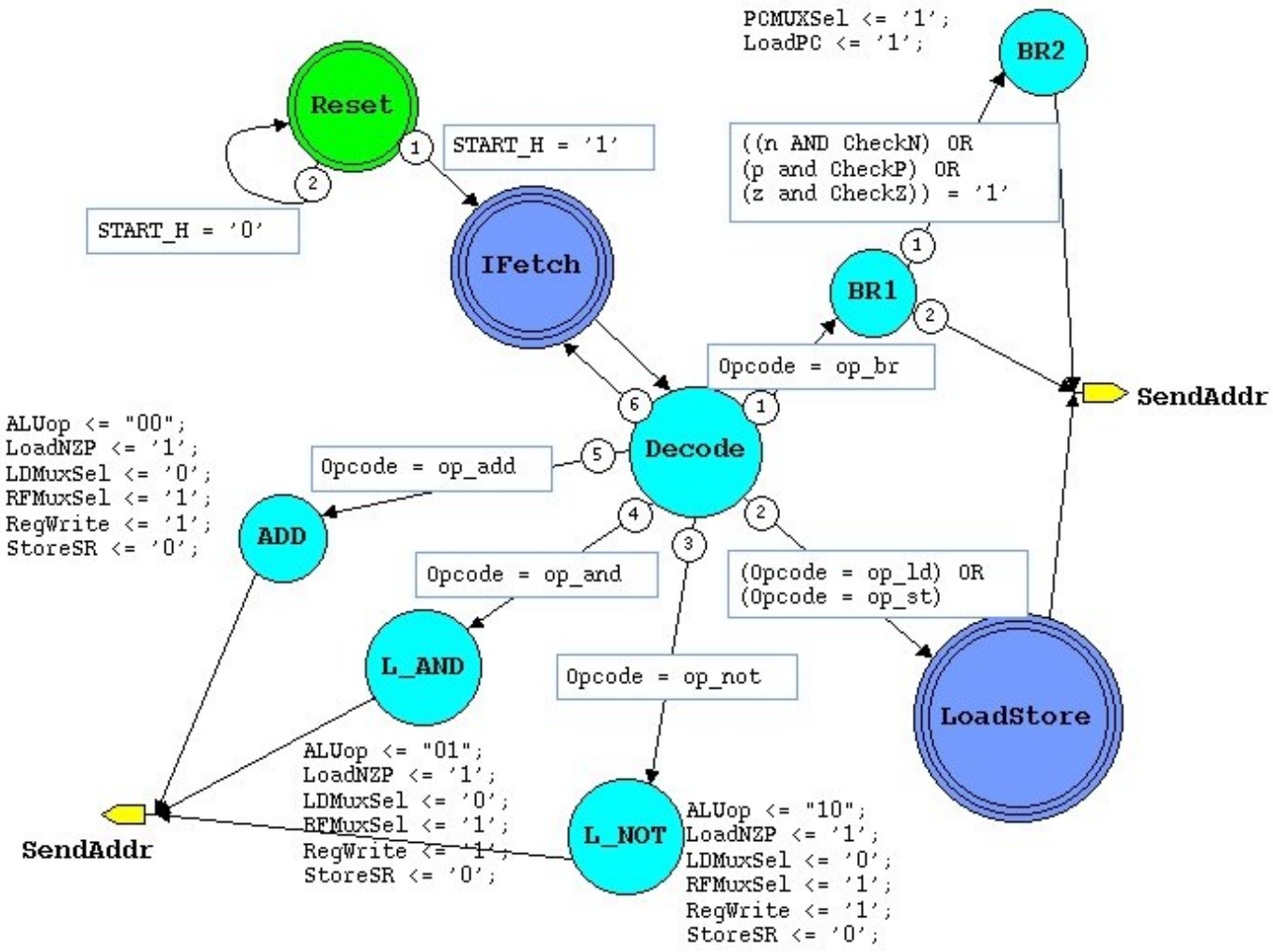
Origin State	**Destination State/Link**	**Condition**
CalcAddr	LD1	Opcode = op_ld
CalcAddr	ST1	Opcode = op_st
Entry Point	CalcAddr	*NONE*
LD1	LD1	MRESP_H = ‘0’
LD1	LD2	MRESP_H = ‘1’
LD2	*Exit Point*	*NONE*
ST1	ST2	*NONE*
ST2	*Exit Point*	MRESP_H = ‘1’
ST2	ST2	MRESP_H = ‘0’

Signals and Defaults

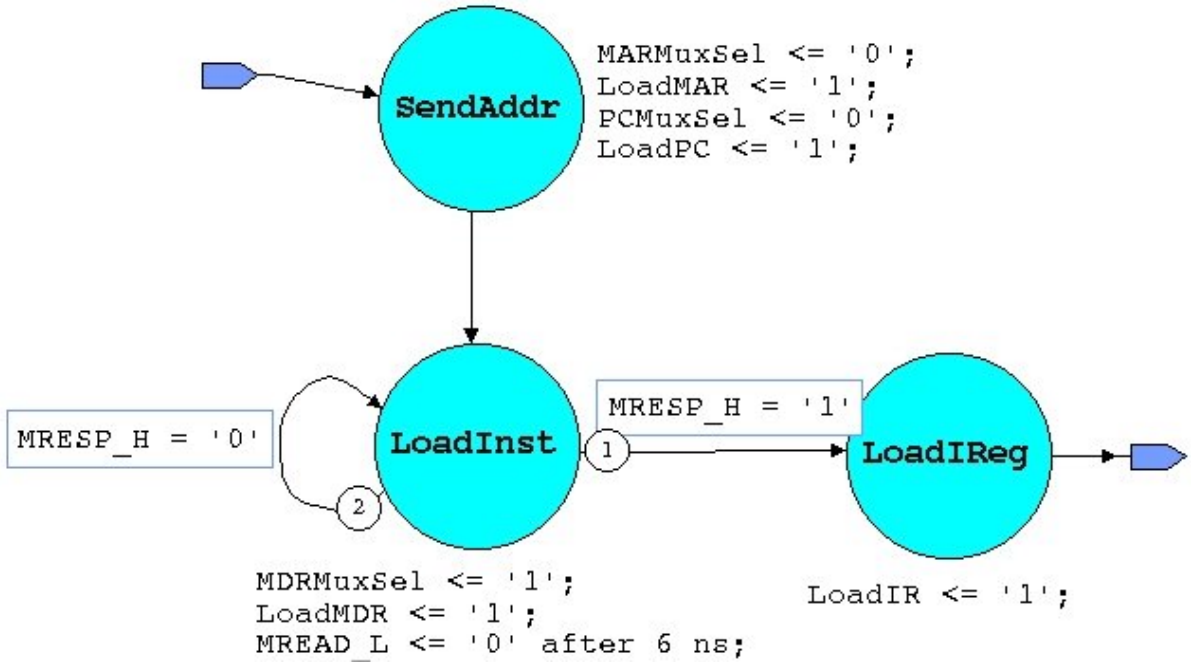
Name:	**Default Value:**
ALUop	“00”
LDMuxSel	‘0’
LoadIR	‘0’
LoadMAR	‘0’
LoadMDR	‘0’
LoadNZP	‘0’
LoadPC	‘0’
MARMuxSel	‘0’
MDRMuxSel	‘0’
MREAD_L	‘1’
MWRITE_L	‘1’
PCMuxSel	‘0’
RFMuxSel	‘0’
RegWrite	‘0’

StoreSR	‘0’

Control Diagrams



IFetch Hierarchical State Diagram



LoadStore Hierarchical State Diagram