ECE 445: Computer Organization and Design

MP1: The LC-3bα Processor / Active HDL Tutorial

Version 3.1

The software programs described in this document are proprietary products of Aldec, Inc. The terms and conditions governing the use of Aldec products are set forth in explicit license agreements and are subject to change. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Aldec whatsoever. Images of software programs in use are assumed to be copyrighted and may not be reproduced.

This document is for informational and instructional purposes only. The teaching staff reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult the teaching staff to determine whether any changes have been made.

This document was originally written by Dr. Ronald Barnes based on the Computer Organization and Design Course taught at University of Illinois, Urbana-Champaign. Major modifications were made by Charles Purvis, Chethan Ananth, Bhupathi Kakarlapudi and Ekawat Homsirikamol.

This turorial, version 3.1, is intended to work with Active-HDL 7.3 only.

Version 3.1 Update Note :

- Correct the diagram in 4.4.1
- More detailed table in 4.4.3
- Some correction on typo errors
- More FAQ

Table of Contents

1 Introduction

Welcome to the first ECE 445 Machine Problem! In this MP we will step through the design entry of a simple, non-pipelined processor that implements a subset of the LC-3b instruction set architecture (ISA). This subset of the LC-3b ISA is referred to as the LC-3bα ISA. This tutorial along with the material on Blackboard, (courses.gmu.edu) 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 Aldec design tool, ActiveHDL. For later MP's, you will use ActiveHDL for design entry and design simulation. Since your next MP's will require original design effort, it is **VERY IMPORTANT** for you to understand how these tools work now so that you can avoid being bogged down with tool-related problems later.

The MP1 LC-3bα processor will be discussed during the hands-on session conducted by your TA. This discussion and the principals discussed in class 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. Later MP's 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 with it yourself 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 forth chapter is the step-by-step procedure for entering the design of the processor using Active HDL. The fifth chapter covers the simulation of the design using Active HDL. The sixth chapter covers the tutorial on how to use AshIDE. The final chapter contains the items you will need to submit for a grade. Also included are several appendices that contain additional useful information at the end of this manual.

As a final note, read each and every work of the tutorial, and follow it very carefully. This is the fourth time these sets of projects have been used in ECE 445 at George Mason University and there is always the potential for errors. Most of these problems will be small typos. However, an equally likely source of student confusions with this MP will come from missing a paragraph or omitting a key step. If you think you have found a mistake in the project description or have some confusion that is not resolved by re-reading the appropriate section, please notify the TA. Take your time and be thorough in the completion of this machine problem, as you will need a functional MP1 design before working on future MP's.

1.1 Notation

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

- Bit 0 refers to the least significant bit (LSB).
- 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 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.

2 The LC-3bα Instruction Set Architecture

2.1 Overview

For this project, you will be entering the VHDL design (using ActiveHDL) of a non-pipelined implementation of the LC-3bα ISA. The LC-3bα ISA will be discussed in the ECE 445 TA hands-on sessions.

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 445 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 GPR's 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).

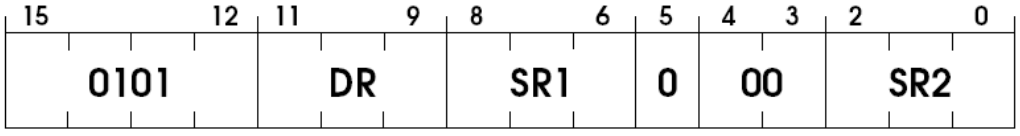
The LC-3bα program control is maintained by the Program Counter (PC). The PC is a 16-bit register that contains the address of the **next** instruction to be fetched.

2.2 Operate Instructions

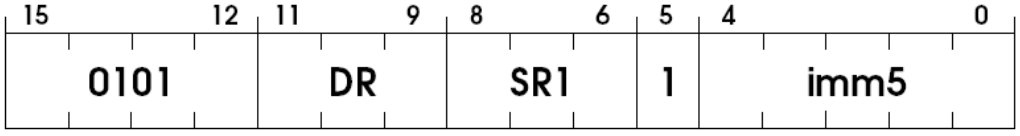
LC-3bα has two integer operate instructions: ADD and ADDI.

The ADD instruction takes a value from the general-purpose register specified by SR1 (SR stands for source register) and adds it to 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 0101.

ADD



ADDI

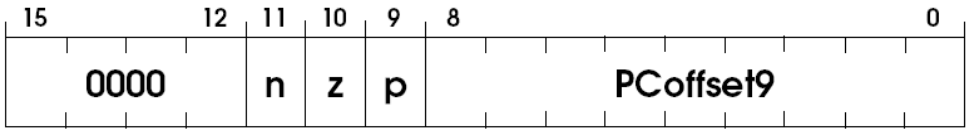


The ADDI instruction works similar to the ADD instruction; except that the second source operand (SR2) is obtained by sign extending the imm5 field to 16 bits. Then the value is added to the contents of SR1. The result of the addition is stored in DR. Note that ADD & ADDI have same opcode values and Bit [5] distinguishes between the two instructions.

2.3 Control Instructions

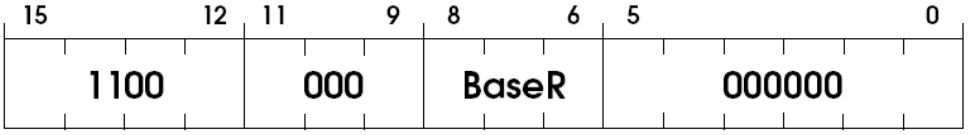
The LC-3bα branch instruction, BR, causes program control to branch to a specified address. The format of the instruction is given below. Specifically, it works as follows: if the n, z, and/or p bits in the machine code instruction are set, and the corresponding condition code is asserted, the processor 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, PC = PC + (SEXT(PCoffset9)<<1).

BR

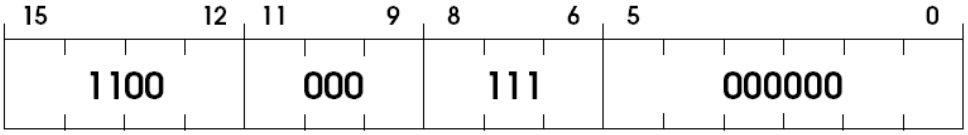


JMP is an unconditional jump which loads the program counter with the value contained in the base register specified by the bits 8:6. Similarly, RET provides the same functionality except that the base register is fixed at register 7. RET is generally used in conjunction with JSR instruction.

JMP

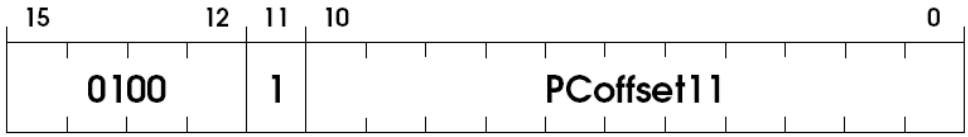


RET



JSR is another unconditional jump. The jump address is calculated by adding current PC to the sign-extended and left-shifted PCoffset11, PC = PC + (SEXT(PCoffset11)<<1). Additionally, the value of the return address, PC+2, is also stored in register 7. R7 = PC + 2.

JSR



3 Design Specification

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.

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 buses, e.g. ADDRESS, DATA, etc.) is to append to 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 memory. The processor sets the MWRITE_L signal active when it is writing to the memory. MREAD_L and MWRITE_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 STATE_H is received. Then the instruction at location 0x0000 of the main memory is executed.

3.4 Hard-Wired Controller

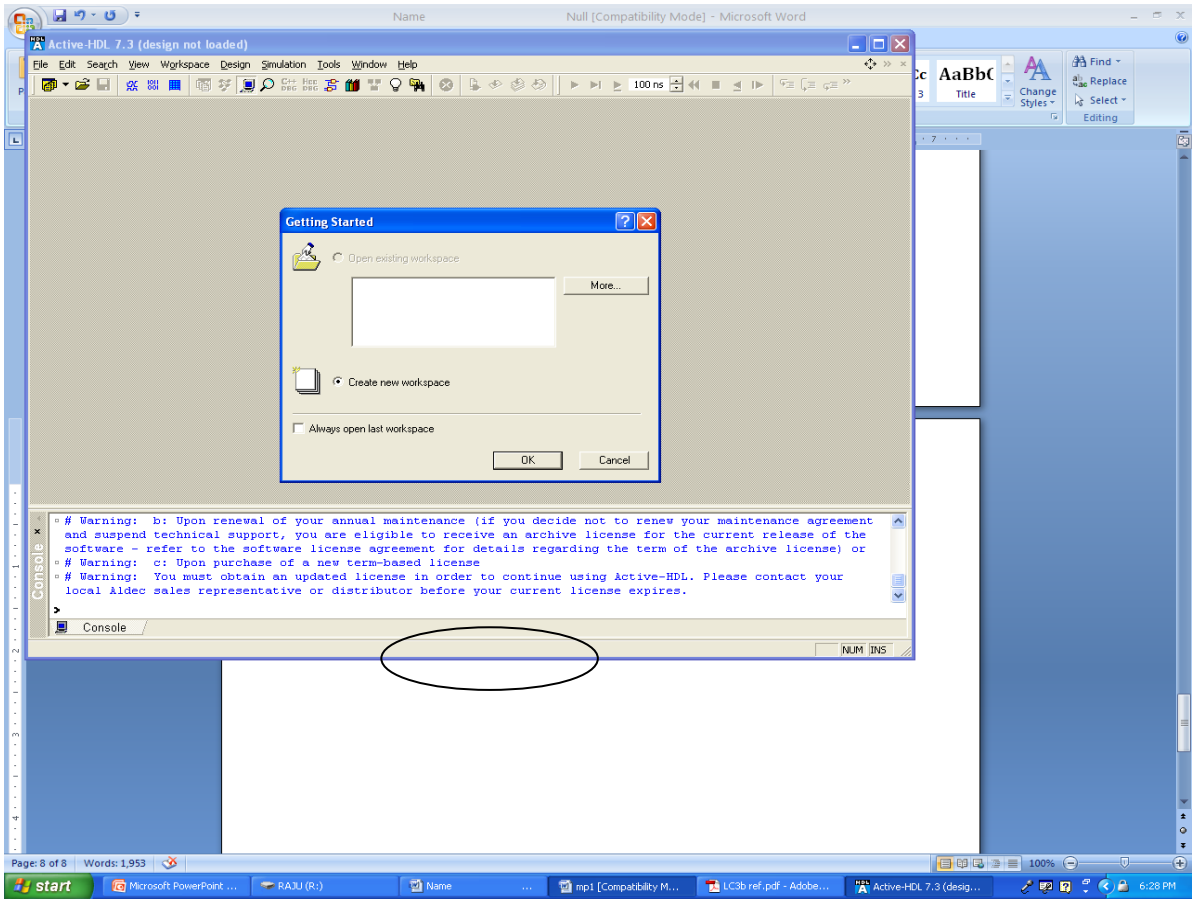
There is a sequence of states that must be executed for every instruction. Its function is to fetch and decode the current instruction.

4 Design Entry

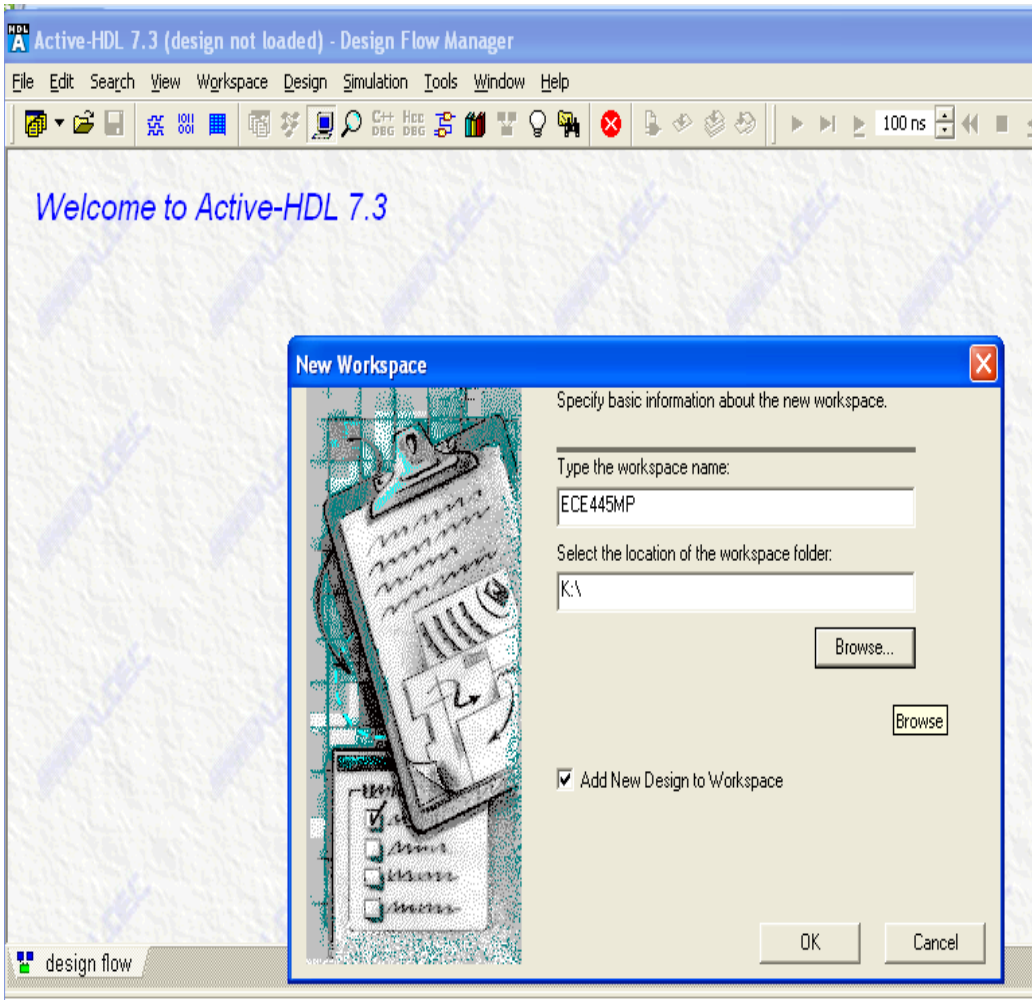
The purpose of this MP, as stated before, is to become acquainted with the LC-3bα ISA and the software tools. You will be using Active HDL from Aldec to layout the design and simulate it.

4.1 Opening a New Design in Active HDL

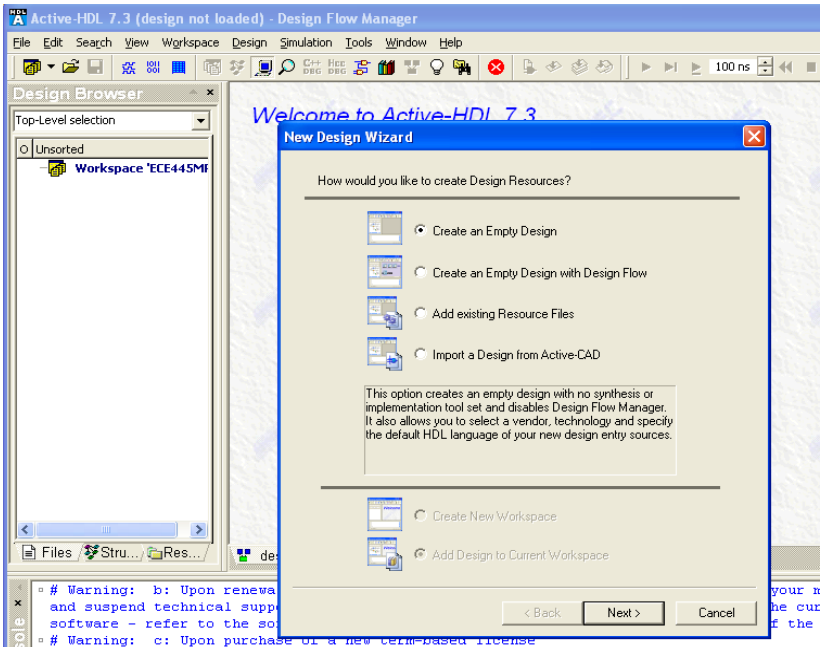
A new workspace has to be created before we start loading files on to the Active HDL tool. Clicking on the Active HDL Icon brings up the tool along with a small window which is used to create a new workspace. Click on create new workspace and click on OK.



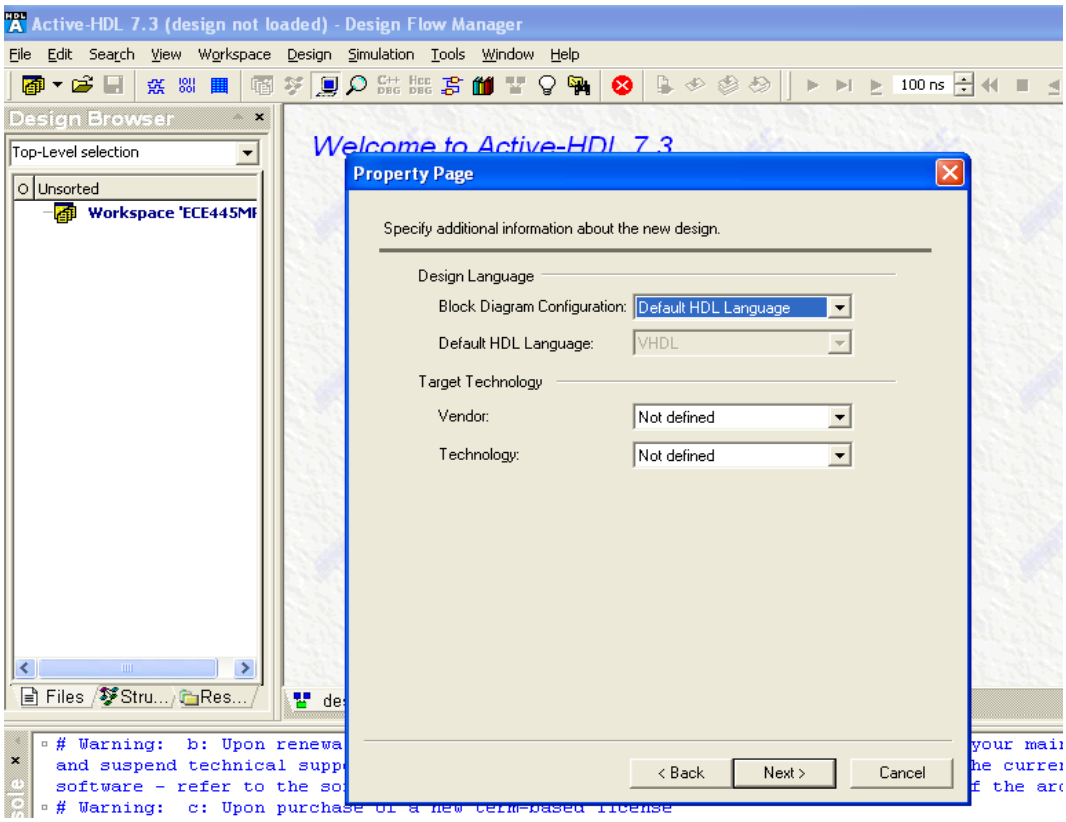
Create a new workspace by typing ECE445MP in the name field. Also check the box for "Add Design to Workspace". The location of the workspace folder can be changed to H drive as your K drive is located on a shared resource and tend to run very slow. However, if you do change your folder to H drive, **do not forget to back up** your source files before logging out. Failure to do so may result in hours of wasted time.



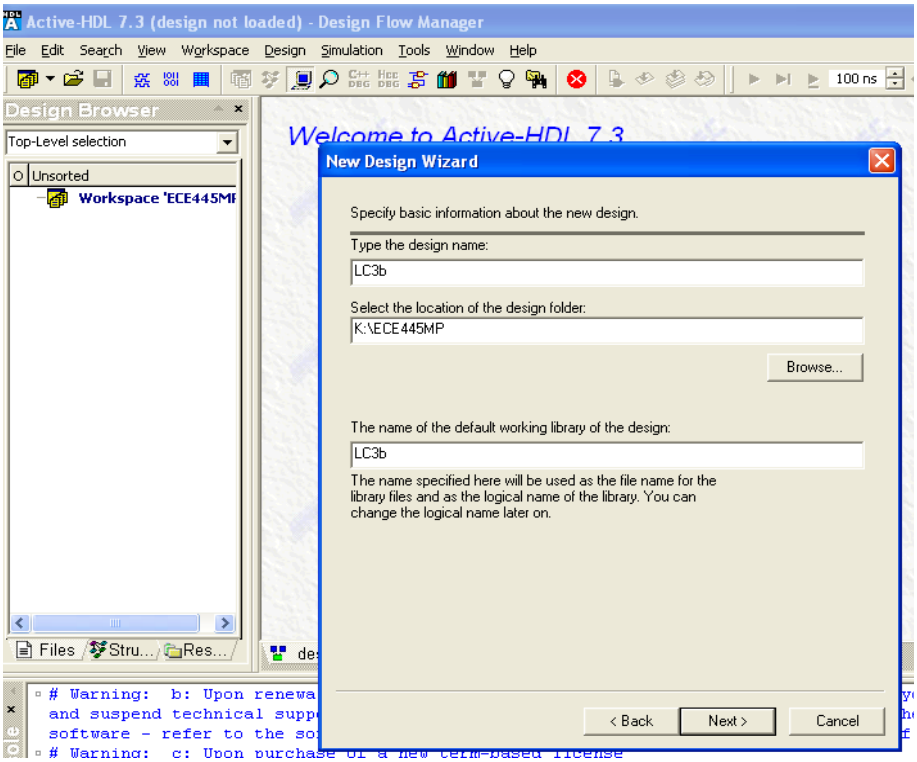
Check the box for Create an Empty Design



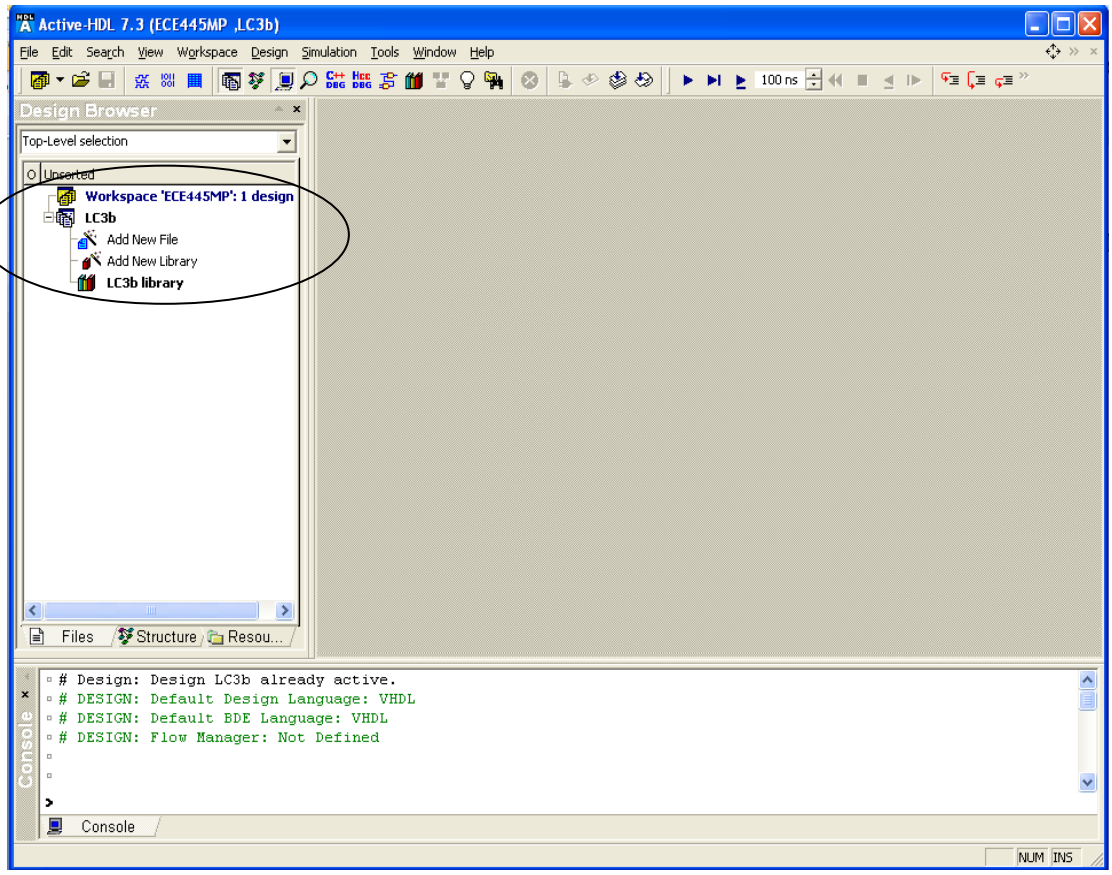
Keep the same settings as displayed in the box below and click on next.



Type the name of the design as ‘LC-3b’ as shown and click “Next” on finish.



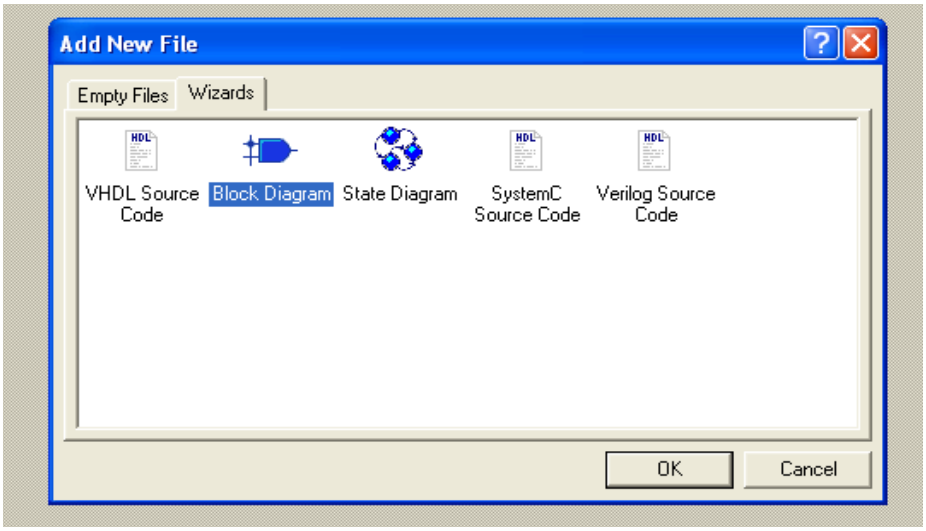
The name of the design will now appear on the left-hand side of the Active-HDL window.



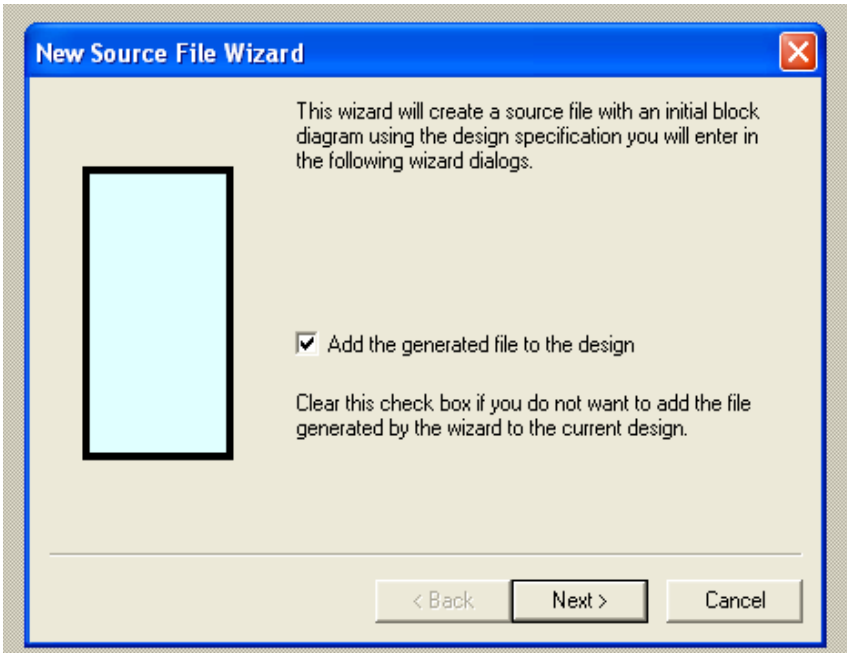
4.2 Top Level Diagram

4.2.1 Create a Block Diagram

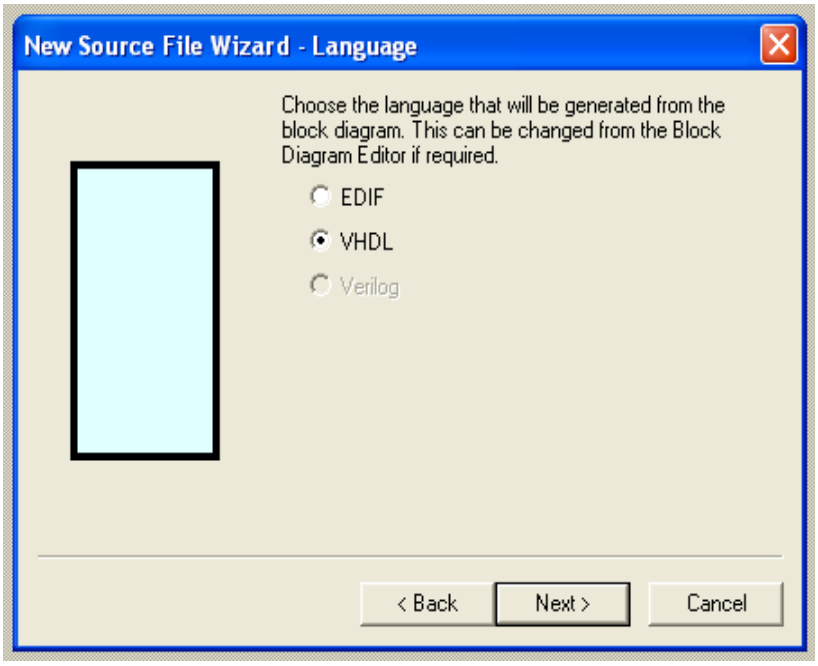
In Active-HDL, a VHDL source code can be manually typed or generated from a block diagram. To create a new block diagram, double click on **Add New File** from the Files tab on the Design Browser Select **Wizards** tab and double click **Block Diagram.** Click **Next.**



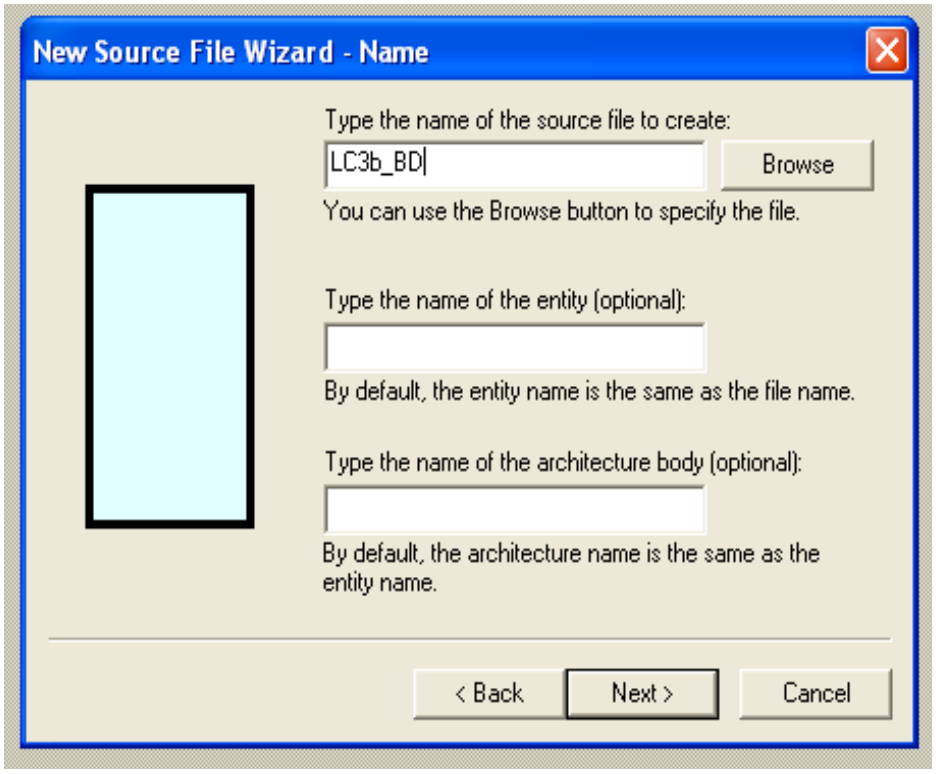
Keep the default settings as it is and click next.



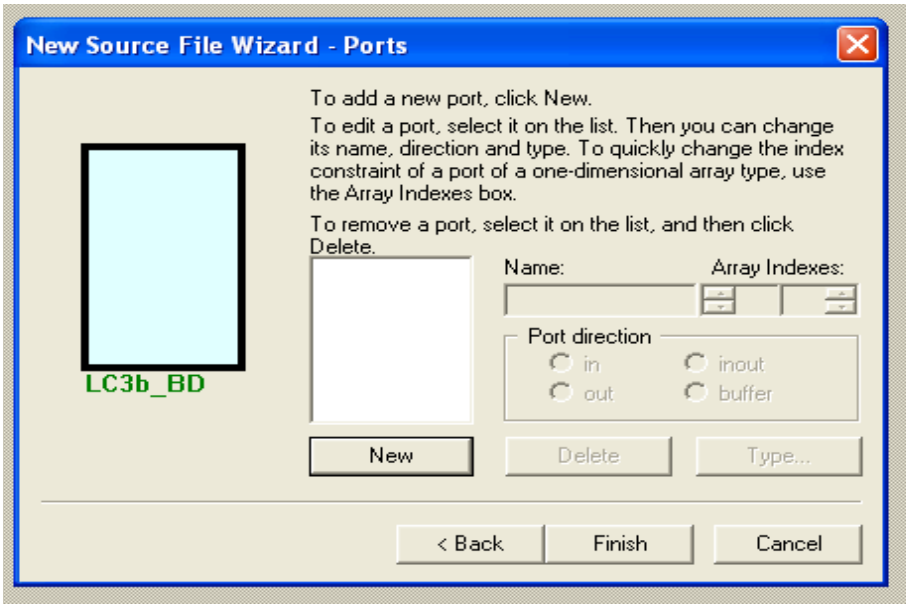
Choose the language as VHDL and click on next window.



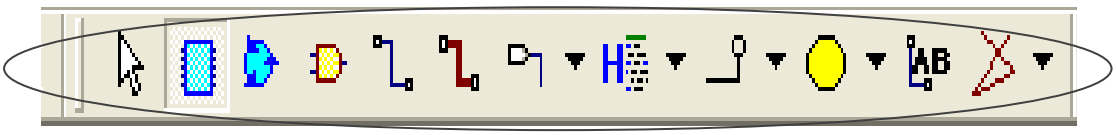
Type the name of the design as LC-3b_BD as shown (BD stands for Block Diagram).



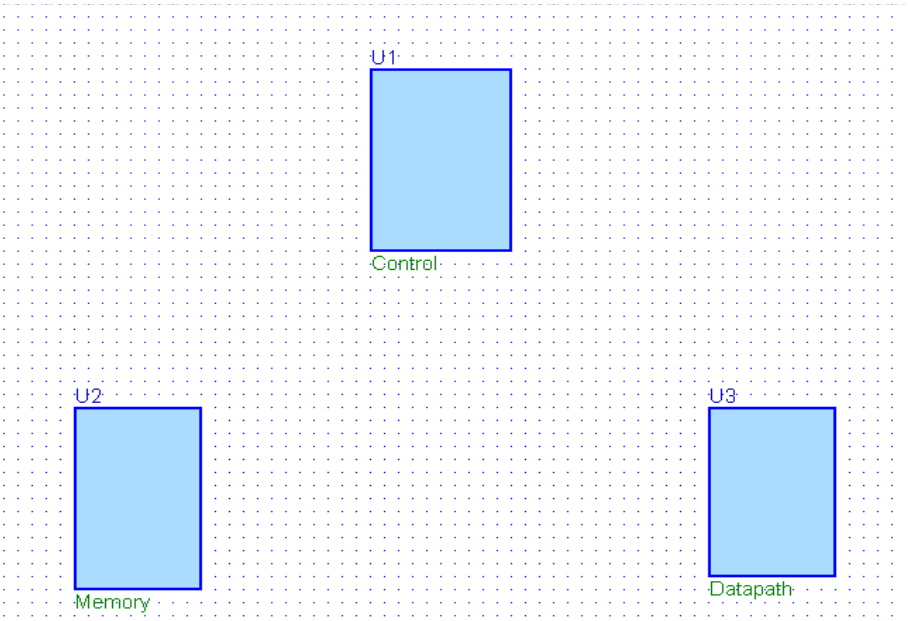
Click on finish on the following window. A blank file will be opened to create the block diagram.



The block diagram is a blank sheet except for a background grid. Create a block diagram as shown using the tools available on the toolbar shown below.



Draw three blocks, also known as Fubs, and change the names of the fubs as “Control”, “Memory” and “Datapath” by double-clicking the green labels below of each fub.



Blocks are effectively “black boxes” whose function is hidden from you in the block diagram, although you will be specifying their functions soon.

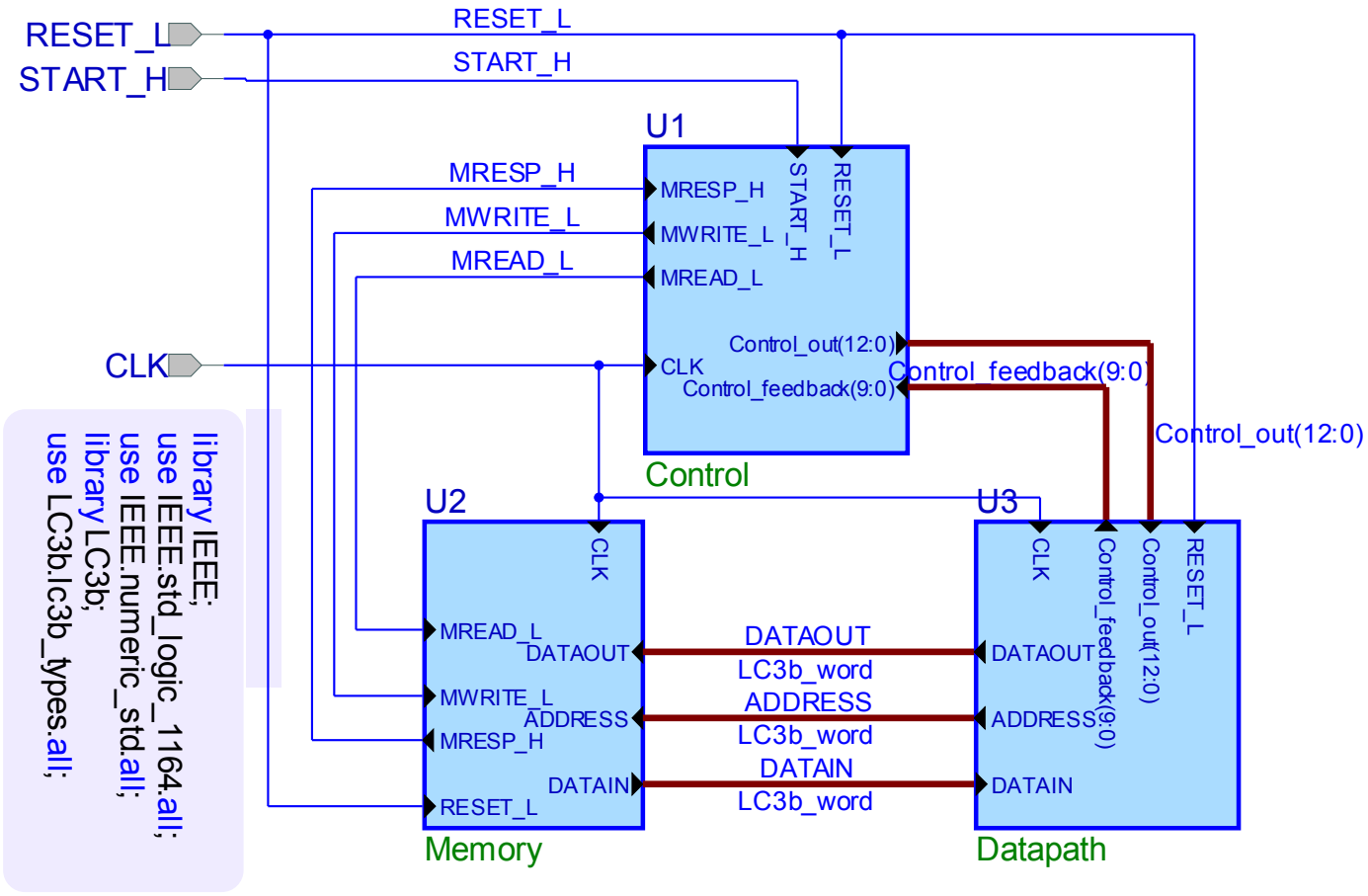
4.2.2 Add Ports and Signals to the Block Diagram

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 in the toolbar which perform equivalent functions.

In the block diagram editor perform the following steps:

1. Create two signals originating at control and ending at memory. Name these MWRITE_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 Datapath, called DATAIN.
7. Create a bus from Control to Datapath. Name the bus as Control_out(12:0)
8. Create another bus from Datapath to Control. Name the bus as Control_feedback(9:0).



To remove any port, click on its name on the list and click the **Delete** button. To create a bus, add a new port name and click the **Array indexes** arrows to specify the bus width. You can rename wires by double-clicking on them and typing a new name in the **Segment** box in the **Wire Properties** window. You can rename buses by double-clicking on them and typing a new name in the Segment box and also you can specify type of bus in the field type name in the **Bus Properties** window, example bus types, LC-3b_word, std_logic_vector etc.

As we are using user-defined types, LC-3b_word, LC-3b_reg and etc., we need to include type definitions. These definitions are included in the package described in "mp1package.vhd". We can call the package by adding

```
Library LC-3b;
USE LC-3b.LC-3b_types.all;
```

to the design unit header. The design unit header specifies the different libraries that have to be used in the design. For every block diagram, a design unit header has to be included. This is done by right-clicking on the empty space in the block diagram and selecting VHDL followed by Design Unit Header. The above statements should be entered in every block diagram.

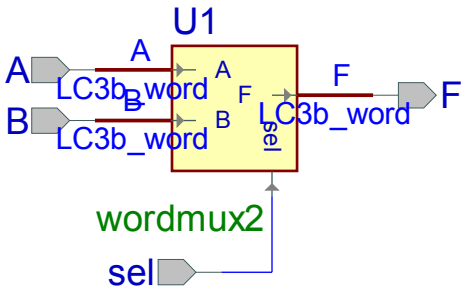
At this stage, you may compile the top level diagram, *LC-3b_bd.bde*. The generated VHDL source code can be observed by expanding the file under Design Browser. Note that it will produce errors/warnings as the components are currently unspecified.

4.2.3 **Component (Symbol) Creation:**

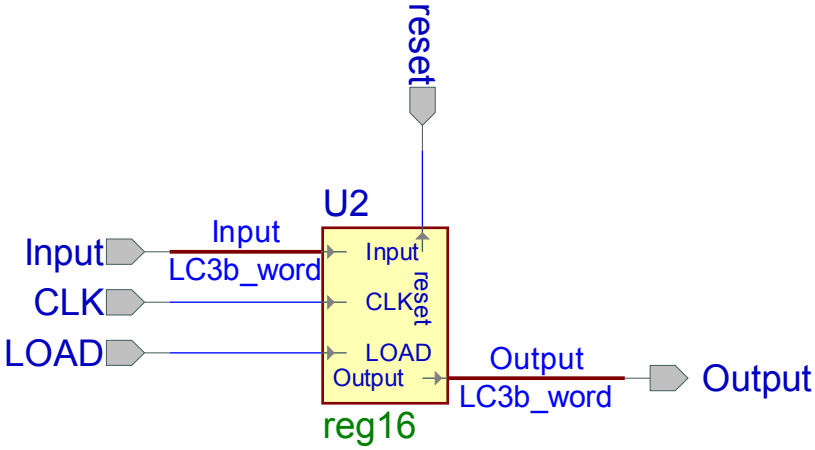
Create two components for use in the datapath. A component once created can be instantiated as many times as desired. Once a small diagram is created in the design, it can be used over and over in any block (or fub) as required by clicking on the 'Symbol' button in the tool bar and searching for the required component. The name of the component will be shown in the list and will have to be dragged and dropped to the block diagram which requires the use of the component.

Components can be created just the same way as we created the initial block diagram. Here instead of placing fubs, we can directly place the component. We will require two components (two separate block diagram files) to be used in our design – a 16 bit 2 to 1 multiplexer (name it wordmux2) and a 16 –bit register (name it Reg16). Create the components by adding a new block diagram file called *symbol*. Draw the mux as shown below and specify the input and the output port names by using the "Input terminal" button on the toolbar. The multiplexer will contain two input ports, one output port and a select line. Once the drawing is completed, double click on each bus and each terminal for A, B, and F, and under 'type name' enter the type 'LC-3b_word'.

Similarly, add another file for the 16 – bit register. The register will have an input port, an output port, a clock, a reset signal and a load signal. Check Appendix D for port types.



WordMux – 16 bit



Register -16 bit

After the both components are drawn as shown, include the vhdl code provided in appendix D in the source code by double-clicking the component and select VHDL Source Code. Do not forget to also include the design unit header with all of your library information as previously discussed.

Once the architecture and header have been specified, compile the file and check for any errors. When there are no errors in the compilation, right click on the components (the register, mux) and select the convert to symbol option. You will notice that this will change the fub's color from blue to yellow. The created components can be used in the datapath block diagram sheet by clicking on the 'Symbol' button on the toolbar and selecting the names you have specified for the two components. Also, since *symbol.bde* is not actually used for the design (it is used as a sheet to draw our components), we can exclude it from the design by right-click on the file name under Design Browser and select "Exclude from Compilation."

4.2.3.1 **Component (Symbol) Creation: Ports**

Name	Direction	Type
A	In	LC3b_word
B	In	LC3b_word
F	Out	LC3b_word
Sel	In	std_logic

WordMux2

Name	Direction	Type
load	In	std_logic
Input	In	LC3b_word
Clk	In	std_logic
RESET	In	std_logic
Output	Out	LC3b_word

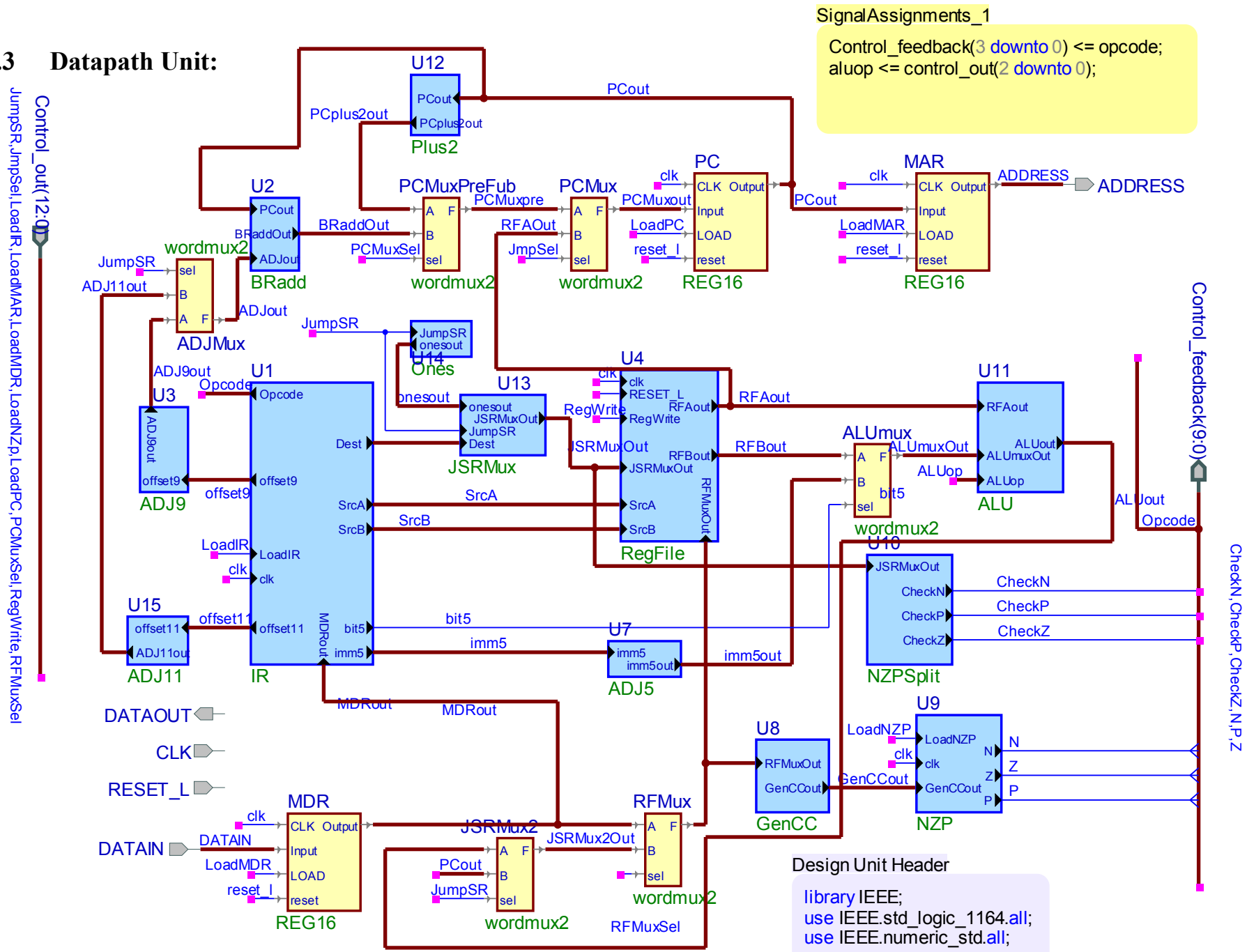
Reg16

4.2.3.2 **Component (Symbol) Creation: 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 untitled;
```

```
ARCHITECTURE untitled OF REG16 IS

Signal pre_out :LC3b_word;
begin
        process (clk, reset )
        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 untitled;
```

4.3 Datapath Unit:



The datapath unit is generated by double clicking on the datapath fub from the top level diagram and selecting "Block Diagram". The required blocks are added in the datapath as shown in the figure above. The completed block diagram should look as shown. Note that the blocks in the datapath which are in yellow are the components used and the blocks in blue are fubs in which the actual VHDL code will be included. The first datapath below is shown with input and output buses connecting to the control unit.

Wires and buses need not be directly connect to its origin. If wires or buses are using the same name, they are technically connected. This technique is especially useful for global wires such as clock and reset signals. If wires/buses have two or more output sources, you may see errors when looking at your waveform, so be **very careful** in naming them.

For signals assignment of *control_out* and *control_feedback*, you need to create buses coming out of the input/output terminals. Then select the buses and choose **Properties**. For *control_out*, name the segment as shown in the above diagram, "JumpSR, JMPSel, ... , StoreSR." Repeat the same process with *control_feedback* signal. Use the same order as specified in the diagram as this will directly affect the generated VHDL source code. You may rearrange the signals and look at the generated source code to verify.

Note that *aluop* and *opcode* signals are omitted from the segments. We have to do this because, the current version of Active-HDL is not smart enough to differentiate between user-defined types and normal signal types (std_logic and etc.). Hence, we need to create a **Signal Assignments** statement as shown in the diagram. Signal Assignments can be added by **right-click ➔ VHDL ➔ Signal Assignments**, or selecting the down arrow next to the Design Unit Header button in the toolbar and follow by Add Signal Assignments.

See tables in section 4.3.1 & 4.3.2 regarding wires & buses specification which are used in the datapath.

4.3.1 Control Signals

Control_Out(15:0)			Control_Feedback(9:0)		
Signal	**Type**	**Bit Location**	**Signal**	**Type**	**Bit Location**
JumpSR	Std_logic	12	CheckN	Std_logic	9
JMPSel	Std_logic	11	CheckP	Std_logic	8
LoadIR	Std_logic	10	CheckZ	Std_logic	7
LoadMAR	Std_logic	9	N	Std_logic	6
LoadMDR	Std_logic	8	P	Std_logic	5
LoadNZP	Std_logic	7	Z	Std_logic	4
LoadPC	Std_logic	6	Opcode	LC-3b_opcode	3 downto 0
PCMuxSel	Std_logic	5			
RegWrite	Std_logic	4			
RFMuxSel	Std_logic	3			
ALUOP	LC-3b_aluop	2 downto 0			

4.3.2 Datapath Signals

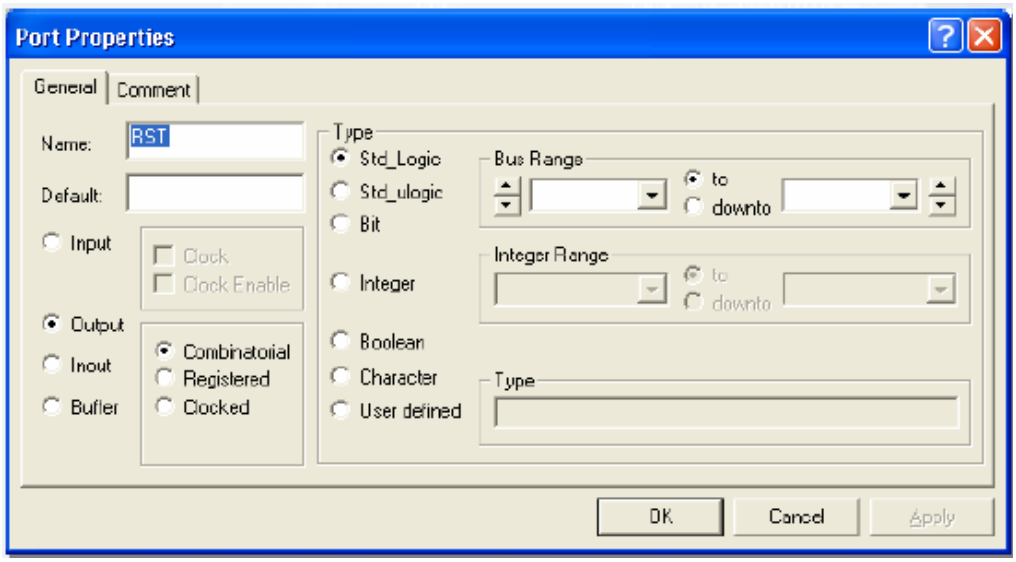
Signal Name	**Type**	**Origin Block**	**Destination Block**
Address	LC-3b_word	MAR	Port Out
ADJ11Out	LC-3b_word	ADJ11	ADJMux
ADJ6out	LC-3b_word	ADJ6	PreAluMux
ADJ9out	LC-3b_word	ADJ9	ADJMux
ADJOut	LC-3b_word	ADJMux	BRadd
AluMuxSel	Std_logic	Port In	PreAluMux
AluMuxOut	LC-3b_word	AluMux	ALU
AluOp	LC-3b_aluop	Port In	ALU
AluOut	LC-3b_word	ALU	JSRMux2
Bit5	Std_logic	IR	AluMux
BraddOut	LC-3b_word	Bradd	PCMuxPreFub
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	LC-3b_word	Port In	MDR
DATAOUT	LC-3b_word	N/A	Port Out
Dest	LC-3b_reg	IR	JSRMux
GenCCout	LC-3b_cc	GenCC	NZP
Imm5	LC-3b_imm5	IR	ADJ5
Imm5Out	LC-3b_word	ADJ5	AluMux
Signal Name	**Type**	**Origin Block**	**Destination Block**

JmpSel	Std_logic	Port In	PCMux
JSRMux2Out	LC-3b_word	JSRMux2	RFMux
JSRMuxOut	LC-3b_reg	JSRMux	RegFile, NZPSplit
JumpSR	Std_logic	Port In	ADJMux, JSRMux2, Ones
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
MDROut	LC-3b_word	MDR	IR,RFMux
N	Std_logic	NZP	Port Out
Offset11	LC-3b_offset11	IR	ADJ11
Offset9	LC-3b_offset9	IR	ADJ9
Onesout	LC-3b_reg	Ones	JSRMux
Opcode	LC-3b_opcode	IR	Port Out
P	Std_logic	NZP	Port Out
PCMuxPre	LC-3b_word	PCMuxPre	PCMux
PCMuxOut	LC-3b_word	PCMux	PC
PCOut	LC-3b_word	PC	Plus2,Bradd, JSRMux2
PCPlus2Out	LC-3b_word	Plus2	PCMuxPre
RESET_L	Std_logic	Port In	RegFile,PC
RegWrite	Std_logic	Port In	RegFile
RFAOut	LC-3b_word	RegFile	Alu,PCMux
RFBOut	LC-3b_word	RegFile	PreAluMux
RFMuxSel	Std_logic	Port In	RFMux
RFMuxOut	LC-3b_word	RFMux	RegFile,GenCC
SrcA	LC-3b_reg	IR	RegFile
SrcB	LC-3b_reg	IR	RegFile
Z	Std_logic	NZP	Port Out

4.4 Control Unit:

Setting up the properties: **Double-click** on the **Control** fub in the top level block diagram and select **State Diagram.** The Finite State Machine (FSM) editor window should open with an outline of our state machine. The FSM generator uses the Block Diagram to generate a corresponding control unit for the design.

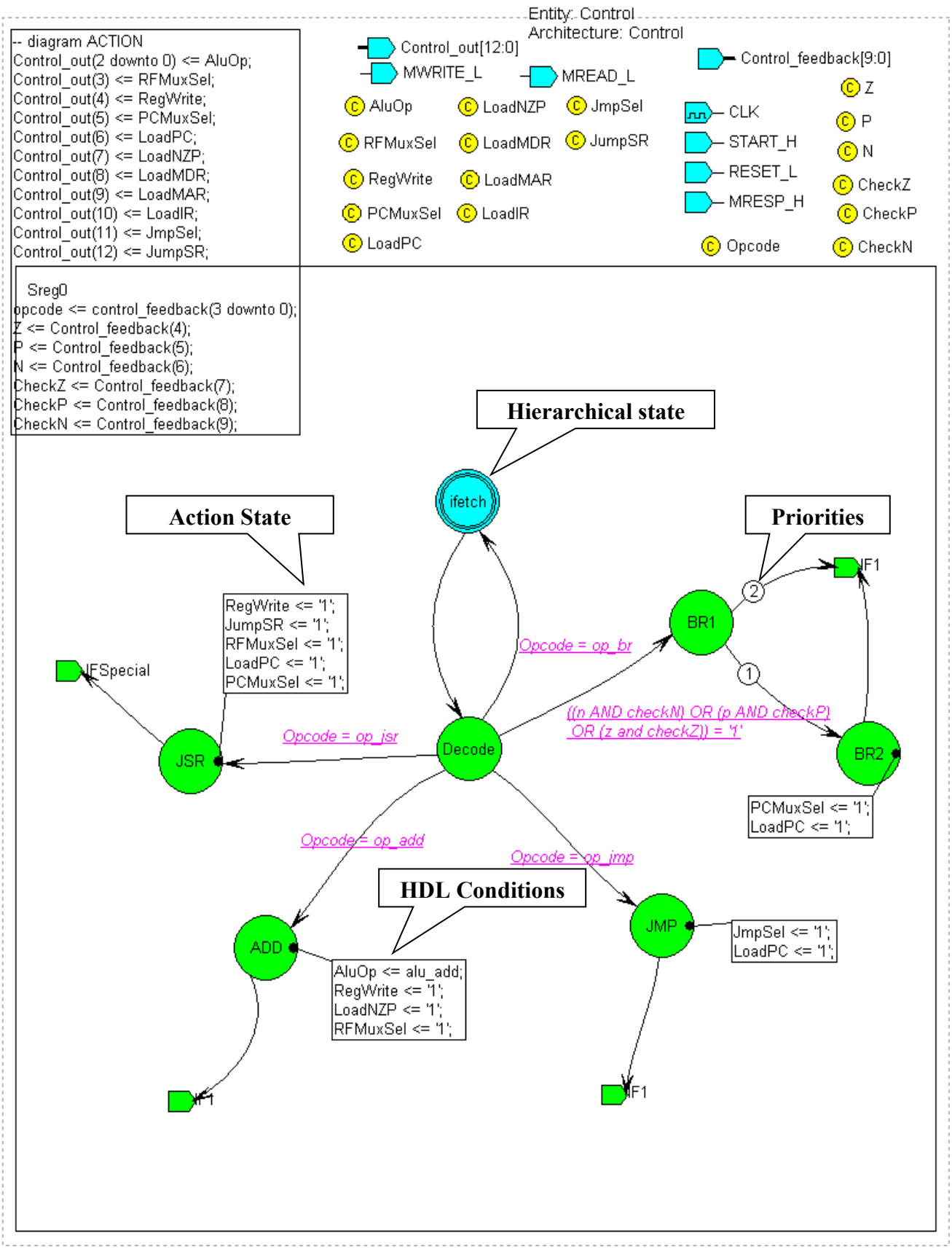
On the FSM editor window, right-click the port symbol and select the **Properties** option from the pop-up menu. You can change the port type there.



The FSM Editor is designed for behavioral descriptions of State Machines. The **Control** unit we are going to describe will be synchronous, so we must declare one of the inputs in the diagram as our machine clock. Right click on the CLK port symbol in the Control state diagram and select **Properties.** Select the **Clock** checkbox in the Port Properties window. Click **OK.**

Using the **FSM State** menu option or **State** button in the toolbar, place the states on the diagram as shown in the below FSM. Don't worry if the state names on your diagram are different from the ones in the picture, you can change them anyway. You can change a state name by right-clicking on the state, selecting **Properties** and typing a new name in the **General** tab of the **State Properties** window. If you are zoomed close enough, you can double click the old name and type the new name directly in the diagram. You can draw transitions between states by selecting **FSM ➔ Transition** from the menu or **Transition** button in the toolbar and clicking the starting state, then clicking the target state.

4.4.1 Control Unit: Top Level FSM



Elaborate reset conditions can be specified by clicking **Advanced** and typing an expression describing the reset condition in **Machine Properties**. Machine Properties can be accessed by double click on the blank space in the state diagram. You can also change the name by switching to the **General** tab and type the name of the machine in the **Name** field. To set a trap or default state, you can switch to the **Defaults** tab. These states are used in cases when illegal conditions are met.

Transition conditions can be added by selecting **FSM Condition** from the menu or **Condition** button in the toolbar, clicking the transition and typing the condition expression. Likewise, you can change Transition conditions by click at the Transition arrow and modify the expression under HDL tab. Priorities shown above can generally be ignored as we already specified the Transition's condition. They are there for instructional purpose only.

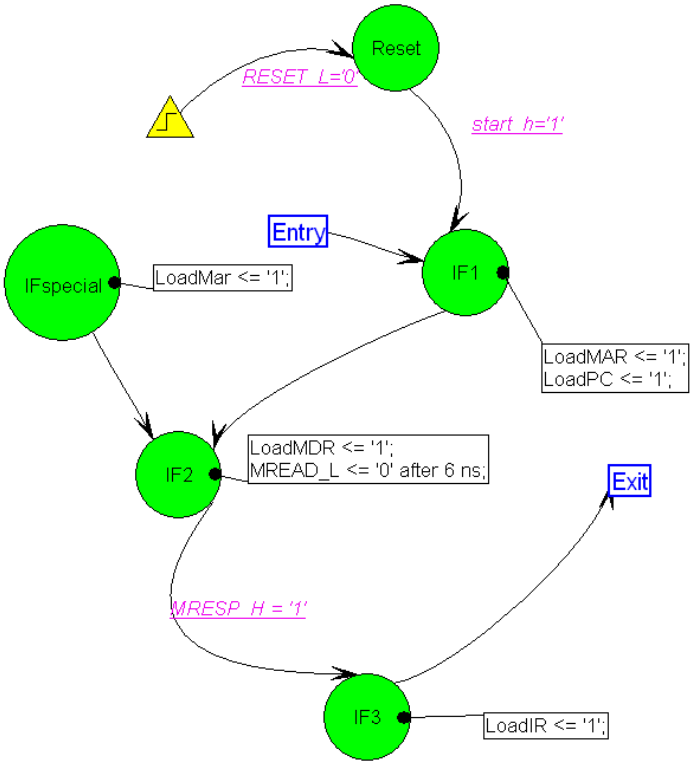
Similarly, Action State can be modified/added by click **State Action** button in the toolbar or clicking at the State directly and modify **Actions** tab. Note that actions should be placed under **State** section. If actions are placed under Entry or Exit section, it may resulted in an unintended action when you test your design.

Signals are added on the FSM by using the Menu "FSM", selecting Signals/Variables and specifying the name of the different signals as shown in the picture. Note that all of the signals are 'combinatorial' type. Include all the yellow signals as shown in the above ASM and make them as combinational signals by right clicking on signals→properties→select combinational and also include default values for all signals which are given in the control unit table below.

Signal assignments must be included by double-click at diagram actions and include all *control_out* & *control_feedback* assignments. Diagram actions can also be accessed by clicking on the "FSM" Menu then clicking on "Action", and then selecting "Diagram" and drawing a box outside the state diagram rectangle, visible on the top left corner of FSM window.

The design unit header for the state diagram can be modified by clicking on the "FSM" menu and then clicking on "Code generation settings", and then selecting the "Design Unit Header" Tab. Make the state "IFETCH" hierarchical by right-clicking on the state and clicking on 'Hierarchical State'. Edit each hierarchical state diagram. Rename the states and connect transitions between the states as shown in the figures below.

4.4.2 Control Unit: IFetch State



4.4.3 Control Unit: Signals & Defaults Table

Name	Default Value	Location	Type
JumpSR	'0'	Control_out(12)	Std_logic
JMPSel	'0'	Control_out(11)	Std_logic
LoadIR	'0'	Control_out(10)	Std_logic
LoadMAR	'0'	Control_out(9)	Std_logic
LoadMDR	'0'	Control_out(8)	Std_logic
LoadNZP	'0'	Control_out(7)	Std_logic
LoadPC	'0'	Control_out(6)	Std_logic
MRead_L	'1'	Output port	Std_logic
MWrite_L	'1'	Output port	Std_logic
PCMuxSel	'0'	Control_out(5)	Std_logic
RegWrite	'0'	Control_out(4)	Std_logic
RFMuxSel	'0'	Control_out(3)	Std_logic
ALUOP	ALU_ADD	Control_out(2 downto 0)	LC3b_aluop
START_H	N/A	Input Port	Std_logic
RESET_L	N/A	Input Port	Std_logic
MRESP_H	N/A	Input Port	Std_logic
Opcode	N/A	Control_feedback(3 downto 0)	LC3b_opcode

Name	Default Value	Location	Type
Z	N/A	Control_feeback(4)	Std_logic
P	N/A	Control_feeback(5)	Std_logic
N	N/A	Control_feeback(6)	Std_logic
CheckZ	N/A	Control_feeback(7)	Std_logic
CheckP	N/A	Control_feeback(8)	Std_logic
CheckN	N/A	Control_feeback(9)	Std_logic

Note: All input ports do not require a default value. Thus, N/A means that you can leave the default value as blank.

5 Compiling and Simulating

You have now completed the **initial phase** of the design of our LC-3bα processor. What you have now is a 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.

Thankfully, we have provided all of the HDL you require. The HDL can be found on the Blackboard page of ECE 445. Go to https://gmu.blackboard.com/ to download any necessary files. The testbench can be generated automatically by selecting the tools menu and clicking on generate test bench and selecting the appropriate entity and architecture. You do not have to make many changes in the test bench other than adding the test data and a clock period. The example test data will be posted on Blackboard shortly. The VHDL files can be included in the fubs in the datapath by right clicking on the fub and selecting push. A new pop up window will be displayed where you need to select "VHDL file" to include the VHDL code provided to you. Download the .vhd files provided and attach it to the corresponding blocks in the datapath.

5.1 Compile

The design that is created has to be compiled using the compile icon on the tool bar in Active HDL. Verify that all settings shown in your window match those in the figure. If you have error messages, check your design and resolve any errors. Most of the errors are explained in the console which gives some idea of how to fix the error. If you do find an error that should not occur, try rearranging the compilation order.

5.2 Testbench

Include the following process statement for the test bench that you generate. The test bench is generated for the main block diagram(the one which has the three fubs memory, control and datapath). This will be the stimulus that would initiate the processor. Add the following after the statement --add your stimulus here--- in the test bench generated. Use a clock period of 60 ns.

```
testing: process
  begin
        wait for 60 ns;
        reset_L <= '0';
        wait for 60 ns;
        reset_l <= '1';
        start_h <= '1';
        wait;
end process;
```

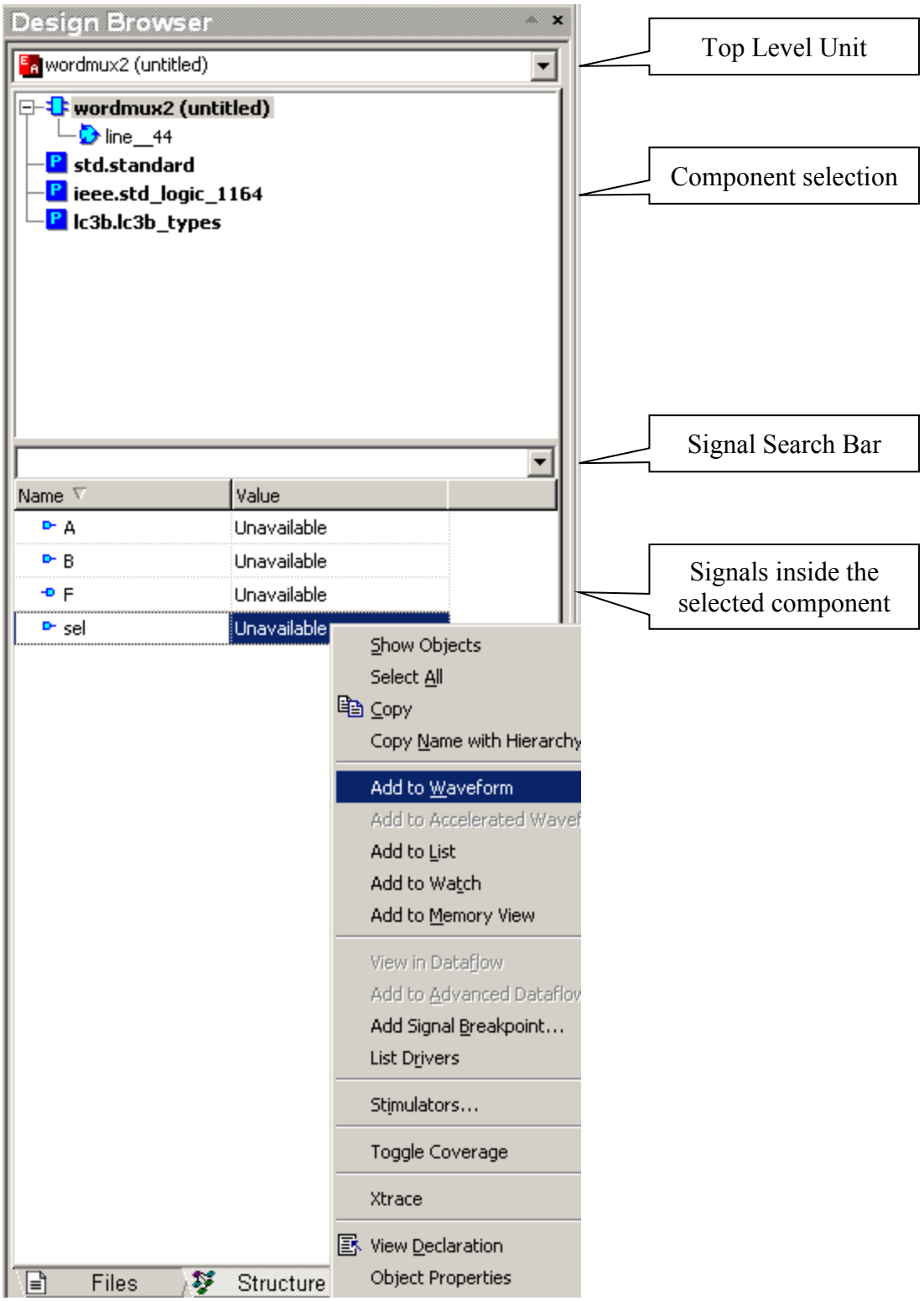
Note: Make sure your reset is set to active low.

5.3 Simulation

Once the design is compiled without errors, the simulator can be invoked by choosing the structure tab at the bottom of Design Console, and selecting your testbench as a top level unit from a menu directly below your Design Browser. Navigate to your signal by selecting the component that your signals are located. Right click on the selected signal and click on “Add to Waveform” to add the ports to the waveform. A waveform window will appear immediately on the right hand side which will contain all the signals that were added. Refer to the following picture for more detail.

Once the signals are added to the waveform, run the waveform using the “Run For” button on the tool bar. Observe all the outputs on the waveform and check if there are any deviations from the expected values. If there are any deviations, look for the output at a particular signal/port where there is a deviation and look for possible errors. The format of the signal can be changed by right clicking at the signal and click Properties.

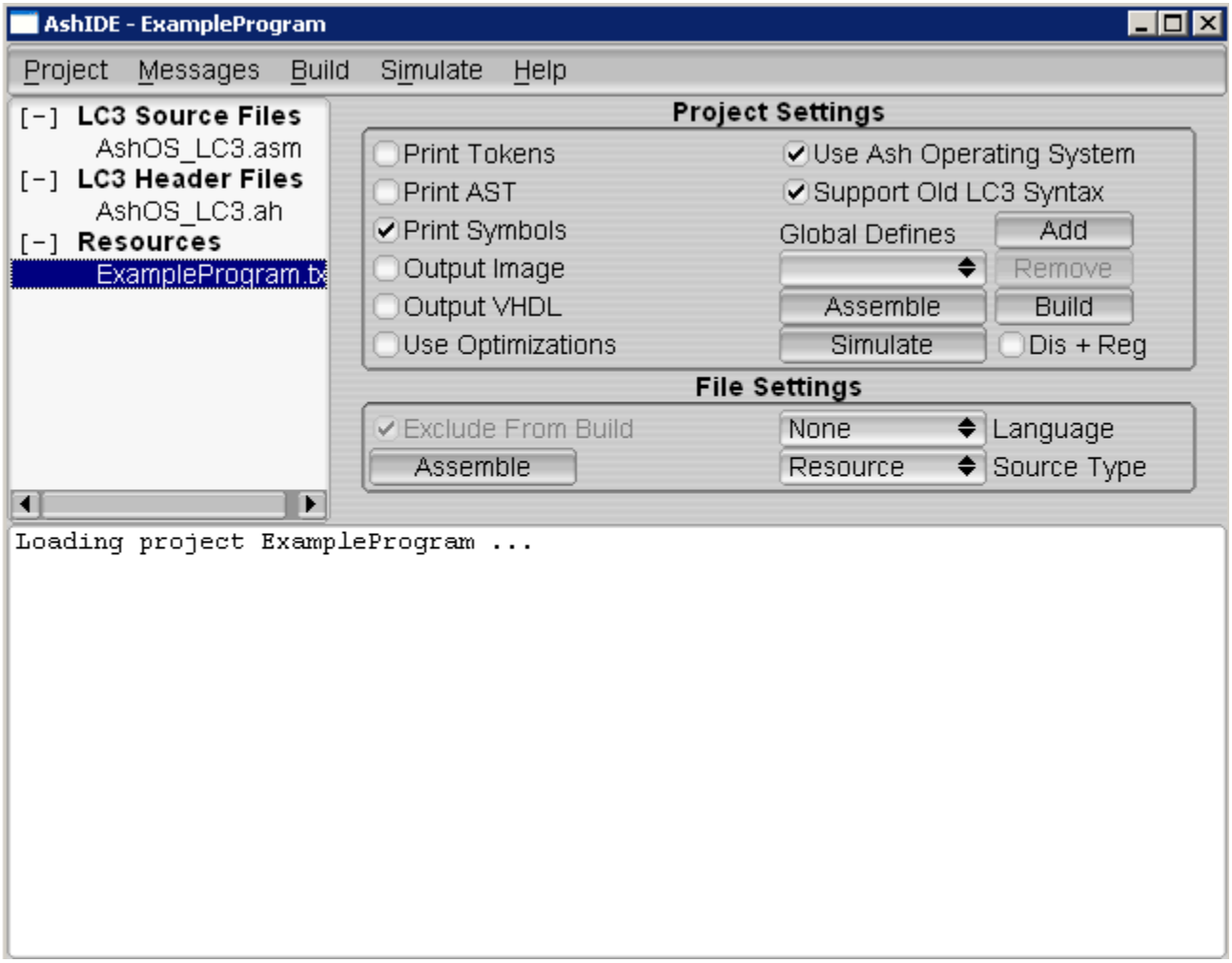
Note that you must insert your own program translated into the machine code into the memory. See Section 6 for more detail on how to create and transfer your program from assembly code to vhdl.


Design Browser
wordmux2 (untitled)
wordmux2 (untitled)
line__44
std.standard
ieee.std_logic_1164
lc3b.lc3b_types
Name
Value
A
Unavailable
B
Unavailable
F
Unavailable
sel
Unavailable
Show Objects
Select All
Copy
Copy Name with Hierarchy
Add to Waveform
Add to Accelerated Wavef
Add to List
Add to Watch
Add to Memory View
View in Dataflow
Add to Advanced Dataflow
Add Signal Breakpoint...
List Drivers
Stimulators...
Toggle Coverage
Xtrace
View Declaration
Object Properties
Files
Structure
Top Level Unit
Component selection
Signal Search Bar
Signals inside the selected component

6 AshIDE Quick Tutorial

To be able to use your created LC-3bα design, you need to insert a working program memory. Translating assembly language to machine language can be very time consuming. Luckily, there's a LC3 Integrated Development Environment (IDE) tool available. AshIDE allows you to test your assembly program for the project. It can also translates your assembly code into machine language which you can insert this into your memory file.

With that in mind, we can start using AshIDE by choosing **Project ➔ New File**. Once you have chosen your desired location and the name of your file as "ExampleProgram.txt", press **OK**. You will now see a similar window as shown below. Select your file (ExampleProgram.txt in this case) under *Resources* and uncheck all settings under **Project Settings,** and check **Dis + Reg** setting**.** Under **File Settings**, select **LC3b** for **Language** and **LC3b Source** for **Source Type**.

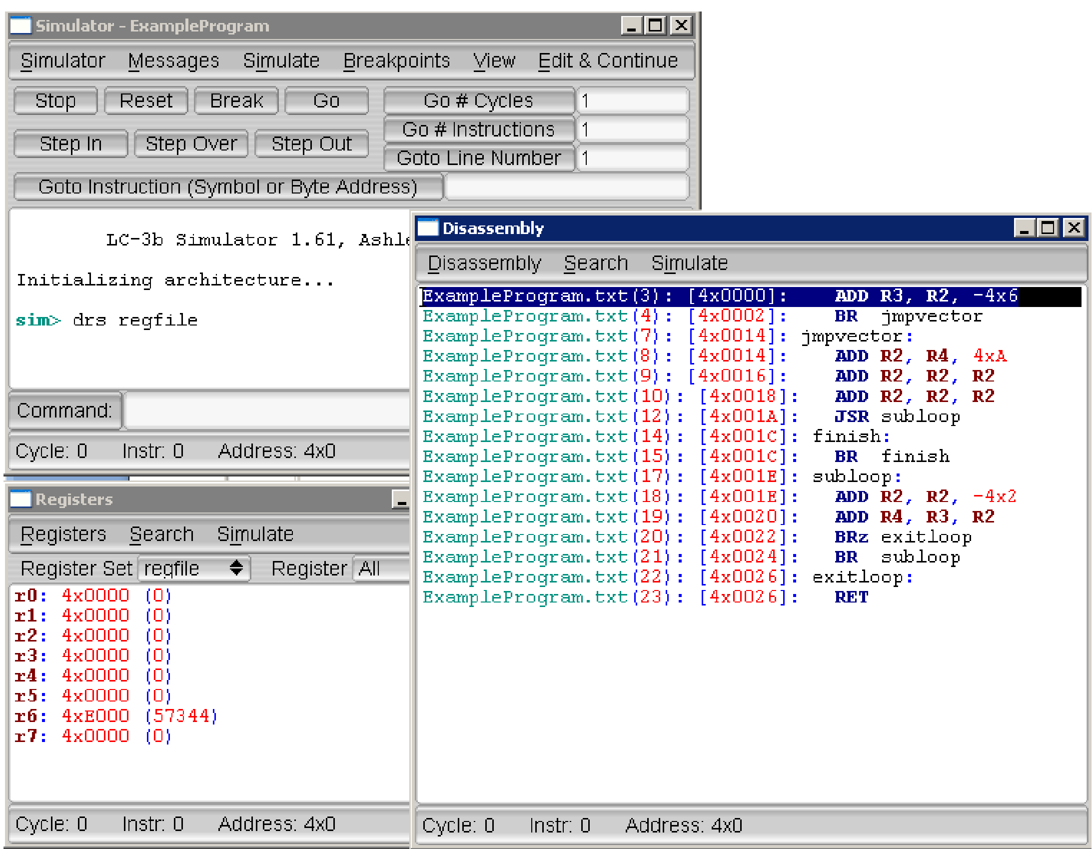


6.1 Assembly Language Simulation

Double click at the file name and insert the following code into your "ExampleProgram.txt" and press **Simulate**.

```
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
ORIGIN 0                    ; origin  specifies the first address where the program will locate
SEGMENT 0
ADD R3, R2, -6  ; R3 = -6
BR JMPVECTOR

SEGMENT 20                  ; segment tells the program that the following code will be located at memory location 20
JMPVECTOR:                  ; this is a label
ADD R2, R4, 10
ADD R2, R2, R2
ADD R2, R2, R2  ; R2 = 30

JSR SUBLOOP

FINISH:
BR FINISH

SUBLOOP:
ADD R2, R2, -2
ADD R4, R3, R2
BRZ EXITLOOP                ; branch if zero
BR SUBLOOP
EXITLOOP:
RET
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
```

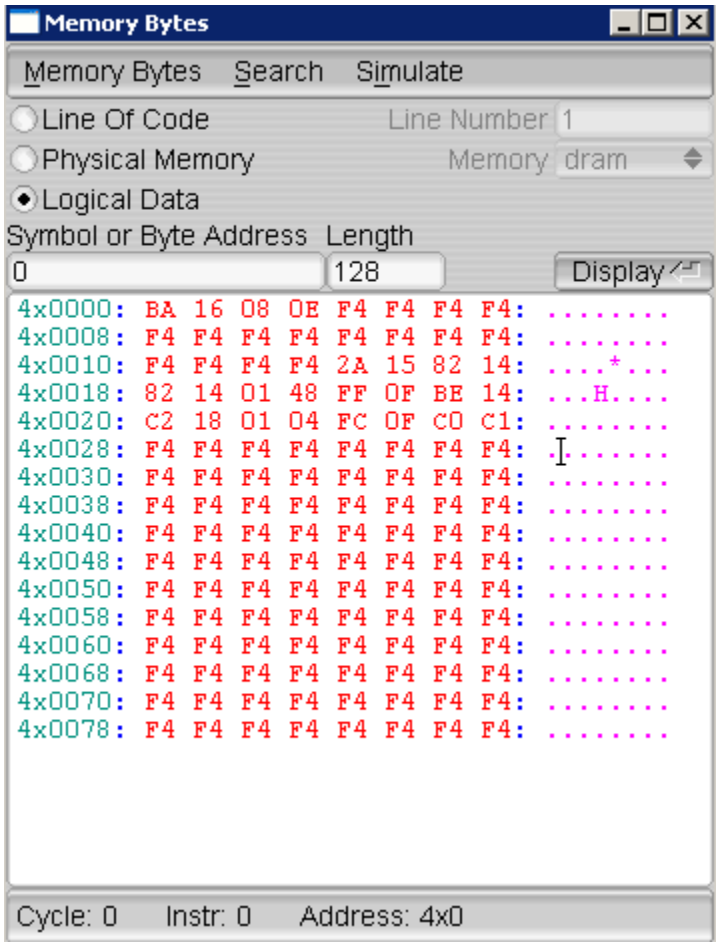
Three windows will pop up as shown below. Uninitialized registers may contain any random value, such as shown in r6. Thus, we can get rid of this by adding some value to R6 (such as *ADD R6, R7, 0*).



The **Registers** window shows your current register value. The **Disassembly** window shows your current location in your program. Pressing "GO# Cycles" in the **Simulator** window will step through the program.

6.2 Converting Machine Language Into VHDL

During simiulation we can view the corresponding machine language of our program by selecting **View ➔ Memory Bytes**. This will open a new window showing data inside the memory.



Looking at the first line, 4x0000: BA 16 08 … F4, these values correspond to the first memory location, the second, and so on. We can confirm this by translating our first LC-3b instruction, *ADD R3, R2, -6*, into machine code which corresponds to Hx16BA (See LC3b ISA for more detail). Therefore, HxBA is the first byte of our machine code and Hx16 is the second. We can directly map these values into VHDL code of our memory.vhd using the following format ➔ mem(mem addr) := x"data", e.g. mem(0) := x"BA". The code can be placed inside the process of our memory.vhd architecture.

7 Final Hand-In

You will be required to write a short program using the LC-3b language. The program should calculate 5! (five factorial). Your program must contain a sub routine. Your code must also end in an infinite loop (jumping or branching to itself). This is to prevent PC going into unspecified programming area. The tools for transforming your code into memory vectors will be found on Blackboard.

You need to hand in the following items:

1. A commented printout of your assembly program.
2. A wave trace showing the first 2000ns of your program's operation.
3. A wave trace showing from 2000ns 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 wave trace must contain the following signals:

RESET_L, START_H, clk, PCout, ADDRESS, MWRITE_L, MREAD_L, DATAIN, DATAOUT, "Machine State", and "RegFile Contents"

"Machine State refers" to the current state of your FSM (Fetch, Decode, etc).
"Regfiile Contents" refers to the contents of each register. Expand the ram signal to display the content of all registers from r0 to r7.

You may include more signals if you wish, but please don't show every signal. Make sure all signal values are shown in Hexadecimal, unless a signed/unsigned decimal is a better choice.

If you are having difficulty understanding any aspect(s) of the MP, ask the TA for assistance. Remember that material from this machine problem will be used in subsequent MPs. It is your responsibility to make sure that MP1 is complete before proceeding with future MPs.

APPENDIX

A. LC-3bα Instruction Set Description

Name	Opcode	Description
ADD	0001	DR ← SR1 + SR2
ADDI	0001	DR ← SR1 + SIGNEXT(imm5)
BR	0000	PC ← PC + (SIGNEXT(offset9)) << 1, when conditions are met
JMP	1100	PC ← SR1
RET	1100	PC ← [R7]
JSR	0100	PC ← PC + (SIGNEXT(offset11)) << 1 **AND** R7 ← PC + 2

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 avoid problems.

C. RTL

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

C.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';
C.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)
C.3 ADD, ADDI Instruction		
State	**Data**	**Control**
FETCH		
DECODE		
ADD	DR ← SR1 + SR2	ALUop ← alu_add; LoadNZP ← '1'; RFMuxSel ← '1'; RegWrite ← '1';
C.4 JMP, RET Instruction		
State	**Data**	**Control**
FETCH		
DECODE		
JMP	PC ← SR1	JmpSel ← '1'; LoadPC ← '1';
C.5 BR Instruction		
State	**Data**	**Control**
FETCH		
DECODE		
BR1	// none	
BR2	PC ← PC + SIGNEXT(offset9)<<1	PCMuxSel ← '1'; LoadPC ← '1';
C.6 JSR Instruction		
State	**Data**	**Control**
FETCH		
DECODE		
JSR	PC ← PC + SIGNEXT(offset)<<1 R7 ← PC + 2	RegWrite ← '1'; JumpSR ← '1'; RFMuxSel ← '1'; LoadPC ← '1';
IFspecial	// none	LoadMAR ← '1';

D. Commonly Used Short Cuts in Block Diagram Design

Key	Function
Shift + scroll up/down	Scroll left/right
Ctrl + scroll up/down	Zoom in/out
W	Wire
B	Bus
I	Wire Input terminal
Shift + I	Bus Input Terminal
O	Wire Output Terminal
Shift + O	Wire Output Terminal
R	Rotate
H	Invert
F	Fub
Alt + enter	Properties of the selected wire/component

E. FAQs

How can I delete a symbol?

Delete the symbol from your block diagram design. Then, double-click at your library on the design browser and remove your desired object.

How can I get rid of *Unknown Identifier* error during compilation?

Unknown identifier generallys means that your signal has not been declared properly. Check whether your signal that causes this is declared anywhere in the VHDL source code. If not, then add them in the *signal declaration area* locates between *architecture* and *begin* statement.