

ECE 411: Computer Organization and Design

MP 0: The LC-3b α Processor / Altera Quartus Tutorial

Version 2.0.1

The software programs described in this document are confidential and proprietary products of Altera Corporation and Mentor Graphics Corporation or its licensors. The terms and conditions governing the sale and licensing of Altera and Mentor Graphics products are set forth in written agreements between Altera, Mentor Graphics and its customers. 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 Altera and Mentor Graphics whatsoever. Images of software programs in use are assumed to be copyright and may not be reproduced.

This document is for informational and instructional purposes only. The ECE 411 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.

Contents

1	Introduction	5
1.1	Notation	5
2	The LC-3bα instruction set architecture	6
2.1	Overview	6
2.2	Data movement instructions	6
2.3	Operate instructions	6
2.4	Control instruction	7
3	Design specifications	8
3.1	Signals	8
3.1.1	Top level signals	8
3.2	Bus control logic	8
3.3	Controller	8
4	Design entry	9
4.1	Beginning the design	10
4.1.1	Add a new component	10
4.1.2	Instantiate components	11
4.1.3	Create the controller	12
4.1.4	Connect the datapath and controller	15
5	Analysis and functional verification	16
5.1	Testbench creation	16
5.1.1	Testbench memory initialization	17
5.2	RTL simulation	17
5.2.1	Verify EDA tool settings	17
5.2.2	Run RTL simulation	17
5.2.2.1	Wave traces	18
5.2.2.2	Lists	18
5.2.2.3	Memory lists	19
5.2.3	Testing your design	19
6	Timing analysis	21
6.1	Set constraints	21
6.1.1	Set clock constraint	21
6.1.2	Set input and output constraints	22
6.2	Write SDC file	23
6.3	Run Timing Analysis	23
7	Final hand-in	25
8	Grading rubric	26
A	Loading programs into your design	27
B	Instruction set description	28
C	RTL	29
C.1	FETCH process	29
C.2	DECODE process	29
C.3	ADD instruction	29
C.4	AND instruction	29
C.5	NOT instruction	30

C.6	BR instruction	30
C.7	LDR instruction	30
C.8	STR instruction	30
D	CPU	31
E	Control	32
E.1	Signals and defaults	32
E.2	Control diagram	32
F	Datapath	35
F.1	Signals	35
F.2	Datapath diagram	36
G	Components	37
G.1	Ports for mux2	37
G.2	Parameters for mux2	37
G.3	SystemVerilog module for mux2	37
H	Example timing analysis report	38
J	Block diagram based design	40
J.1	Add and name new blocks	40
J.2	Save the block diagram	41
J.3	Add ports and signals	41
J.4	Add finite state machine	41
J.5	Complete datapath and finite state machine	41
J.6	Generate HDL code from the block diagram	42

1 Introduction

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

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

The remainder of this section describes some notation that you will encounter throughout this tutorial. Most of this notation should not be new to you; however, it will be worthwhile for you to reacquaint yourself with it before proceeding to the tutorial itself. Section 2 contains a description of the *six instructions* in the LC-3b α instruction set. Section 3 contains a high-level view of the design. Section 4 is the step-by-step procedure for entering the design of the processor using Altera Quartus. Section 5 covers the simulation of the design using ModelSim. Section 7 contains the items you will need to submit for a grade. Also included are several appendices that contain additional useful information.

As a final note, *read each and every word of the tutorial* and follow it very carefully. There may be some small errors and typos. However, most problems that past students have had with this MP came from missing a paragraph and omitting some key steps. Take your time and be thorough, as you will need a functional MP 0 design before working on future MPs.

1.1 Notation

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

- Bit 0 refers to the *least* significant bit.
- Numbers beginning with 0x are hexadecimal.
- [address] means the contents of memory at location address. For example, if MAR = 0x12, then [MAR] would mean the contents of memory location 0x12.
- For RTL descriptions, pattern[x:y] identifies a bit field consisting of bits x through y of a larger binary pattern. For example, X[15:12] identifies a field consisting of bits 15, 14, 13, and 12 from the value X.
- A macro instruction (or simply instruction) means an assembly-level or ISA level instruction.
- Commands to be typed at the terminal are shown as follows:

```
$ command
```

Do not type the dollar sign; this represents the prompt displayed by the shell (e.g., [netid@linux-v1 ~]\$).

- Filenames are shown in *italics*.
- Signal names are shown in *fixed width*.
- Actions to take in the GUI are shown in **bold**.

2 The LC-3b α instruction set architecture

2.1 Overview

For this project, you will be entering the SystemVerilog design of a non-pipelined implementation of the LC-3b α instruction set architecture. The LC-3b α ISA consists of six instructions selected from the full LC-3b ISA (19 instructions). The LC-3b ISA is an ISA created for instructional purposes. Because it is a relatively simple ISA, it is a natural choice for our ECE 411 projects.

All six instructions are 16 bits in length, having a format where bits [15:12] contain the opcode. The LC-3b α 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.

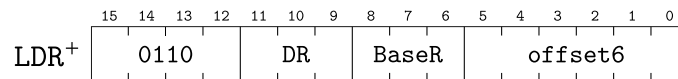
The memory of the LC-3b α consists of 2^{16} locations (meaning the LC-3b α has a 16-bit address space) and each location contains 8 bits (meaning that the LC-3b α has byte addressability).

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

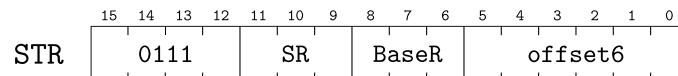
2.2 Data movement instructions

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

The format of the load instruction, or LDR, is shown below. The opcode of the LDR instruction bits [15:12]) is 0110. The effective address (the address of the memory location that is to be read) is specified by the BaseR and offset6 fields. The effective address is calculated by adding the contents of the BaseR to the sign-extended and left-shifted-by-one offset6 field.



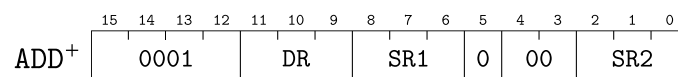
The format of the store instruction, STR, is shown below. The opcode of this instruction is 0111. As with the load instruction (LDR), the effective address is the memory location specified by the BaseR and offset6. The effective address is formed in the same manner as that of the LDR.



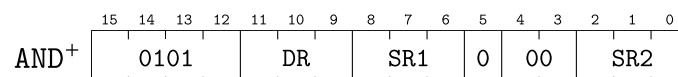
2.3 Operate instructions

LC-3b α has three integer operate instructions: ADD, AND, and NOT.

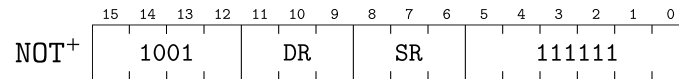
The ADD instruction takes a value from the general-purpose register specified by SR1 (SR stands for source register) and adds it the value from the register specified by SR2. The result is stored in the register specified by DR (DR stands for destination register). The opcode for ADD is 0001 and its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



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.

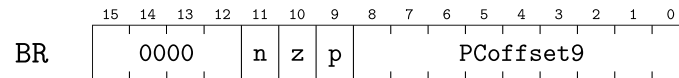


The NOT instruction performs a bitwise complement on the value in register SR and places the result in register DR. The opcode for NOT is 1001 and its format is shown below.



2.4 Control instruction

The LC-3b α branch instruction, BR, causes program control to branch to a specified address. The format of the instruction is given below. Specifically, it works as follows: if the n, z, and/or p bits in the instruction are set, and the corresponding condition code is asserted, the processor will take the branch. When the branch is taken, the address of the next instruction to be executed is calculated by adding the incremented PC value to the sign-extended and left-shifted PCoffset9 field.



3 Design specifications

3.1 Signals

The microprocessor communicates with the outside world (e.g., the memory) through an address bus, read and write data buses, four memory control signals, and a clock.

3.1.1 Top level signals

`clk`

A clock signal, all components of the design are active on the rising edge

`mem_address[15:0]`

Memory is accessed using this 16-bit signal

`mem_rdata[15:0]`

16-bit data bus for receiving data from memory

`mem_wdata[15:0]`

16-bit data bus for sending data to memory

`mem_read`

Active high signal that tells memory that the address is valid and that the processor is trying to perform a memory read.

`mem_write`

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

`mem_byte_enable[1:0]`

A mask describing which byte(s) of memory should be written on a memory write. If the MSB is high, the high byte location will be written. If the LSB is high, the low byte location will be written. If both are high, both locations will be written.

`mem_resp`

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

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 `mem_read` control signal active (high) when it needs to read data from the memory. The processor sets the `mem_write` signal active when it is writing to the memory (and sets the `mem_byte_enable` mask appropriately). `mem_read` and `mem_write` must never be active at the same time! The memory activates `mem_resp` 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 Controller

There is a sequence of states that must be executed for every instruction. The controller contains the logic that governs the movement between states and the actions in each state. In the LC-3b α , each instruction will pass through the fetch and decode states, and once decoded, pass through any states appropriate to the particular instruction.

4 Design entry

Note: If you do not have an EWS account, please contact one of the TAs and he or she will help you obtain an account.

The purpose of this MP, as stated before, is to become acquainted with the LC-3b α ISA and with the software tools. You will be using Quartus II from Altera to lay out designs and ModelSim to simulate them for the remainder of the semester, so it is important that you understand how to use the tools.

Note: If you wish to learn more about the features in Quartus, you can go through the Quartus tutorial, which is available through Quartus itself (click on **Help** \rightarrow **Getting Started Tutorial**). The tutorial may cover additional topics not covered here.

To start using Quartus, first make sure you are in your EWS home directory. Type the following in a terminal window (**Applications** \rightarrow **System Tools** \rightarrow **Terminal**) on an EWS Linux machine (e.g., ECEB 2022 or Grainger 57):

```
$ cd ~
```

Create a directory for ECE 411 work. All paths referenced in this document will refer to this directory as the root.

```
$ mkdir ece411
```

Change into your newly created directory.

```
$ cd ece411
```

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

```
$ pwd      # example output: /home/<netid>/ece411
```

Download the given files from the course website and extract them.

```
$ curl -L courses.engr.illinois.edu/ece411/mp/mp0/ece411_given.tar.xz \  
$      | tar -xJv
```

Three directories will be extracted: *bin/*, *mp0/*, and *testcode/*.

- *bin/* contains executables that will be used throughout the semester
 - *load_memory.sh*: script to generate *memory.lst* file from *.asm* test code for use in testbench memory
 - *LC3bAssembler*: LC-3b assembler used to simulate the assembly test code in terminal
 - *LC3bIDE*: LC-3b GUI simulator used to run the assembly test code to view the register values, etc.
 - *LC3bSimulator*: LC-3b command line simulator used to run the assembly test code to view the register values, etc.
 - *rereference.sh*: script to aid in renaming projects by renaming files and replacing textual references.
 - *README*: details of the executables are found here
- *mp0/* contains a set of files to get you started on MP 0
 - Quartus II project and settings files (*mp0.qpf*, *mp0.qsf*)
 - SystemVerilog design files (**.sv*)
 - *lc3b_types.sv*: a package that defines useful types and enums for the project
- *testcode/* is where you will place test code (either given or created by yourself) to simulate the design

To begin work on the MP, open Quartus (32-bit version).

```
$ module load altera  
$ quartus &
```

To open the project in Quartus,

1. Click on **File** → **Open Project** (not **Open**)
2. Navigate to **given** → **mp0** and select **mp0.qpf**.

Note: If you don't want to type `module load altera` every time, add the line to your `~/.modulerc` file. You may need to logout and login again for changes to take effect.

```
$ cp -iv /srv/adm/modules/init/modulerc ~/.modulerc
$ echo "module load altera" >> ~/.modulerc
```

4.1 Beginning the design

Some components for the LC-3b α have been provided for you. You will create several missing components, connect them together to form the datapath, and implement a controller to sequence the machine.

You can view the provided files by clicking the **Files** tab at the bottom of the Project Navigator. Take a look at Appendix F.2 to get a feel for what components are provided and what components need to be created.

Open up the datapath by double-clicking **datapath.sv** in the **Files** tab. The given *datapath.sv* file contains a couple of already instantiated components and a partial port declaration. You will need to create and instantiate additional components and declare additional ports to complete the design.

4.1.1 Add a new component

Begin the design by creating a two-input mux. Click **File** → **New** and create a new SystemVerilog HDL File.

In the editor that opens, paste the code from Appendix G. We will walk through the code below. If you have not done SystemVerilog design before, it may be helpful to review the SystemVerilog resources before or in parallel with the explanations below.

Listing 1: The mux2 port declaration

```
module mux2 #(parameter width = 16)
(
    input sel,
    input [width-1:0] a, b,
    output logic [width-1:0] f
);
```

The first section declares the two-input mux module, mux2, and its input and output ports. A parameter is used to specify the width of the mux with the default width being 16 bits. The output signal f is multiplexed from signals a and b using the select signal sel. Unless specified, the type of input and output signals is wire. The logic type is specified for f so that it can be driven from the always block. The difference between wire and logic can be subtle, see the Verilog resources for more information. The select signal is 1 bit wide while the width of a, b, and f are determined by the width parameter.

Listing 2: The mux2 definition

```
always_comb
begin
    if (sel == 0)
        f = a;
    else
        f = b;
end
```

The next section specifies the internal workings on the two-input mux. The always_comb block specifies a section of code that will always be executed and will synthesize as combinational logic. The keywords always_ff

and `always_latch` tell the synthesis tools that you intend to generate flip-flops or latches, respectively. For an `always_ff` block, a sensitivity list needs to be provided to specify when the block will execute (see *register.sv* for a usage example).

Listing 3: The mux2 module end

```
|     endmodule : mux2
```

The final statement specifies the end of the module. The colon and following label are optional, but if given, must match the name of the module. Save the file as *mux2.sv*.

Now, create the rest of the missing components: an adder to compute the branch target address and an nzp comparator to compute the branch enable signal.

4.1.2 Instantiate components

Once the components are created, you need to instantiate the components in the datapath (*datapath.sv*). If you haven't done SystemVerilog design before, we'll walk through the process by instantiating the storemux.

Before instantiating the storemux, the internal signals that it is connecting to need to be declared.

Listing 4: Internal signals for storemux in *datapath.sv*

```
|     lc3b_reg sr1;
|     lc3b_reg dest;
|     lc3b_reg storemux_out;
```

Note that `lc3b_reg` is defined in *lc3b_types.sv*. `storemux_sel` needs to come from the control unit, so we will add it to the existing port declaration for the datapath.

Listing 5: Additional signal for datapath port declaration

```
|     module datapath
|     (
|         /* control signals */
|         input storemux_sel
|     );
```

If you're familiar with object oriented programming, instantiating a component is similar to instantiating an object. To instantiate a component, we need to provide the type, a name, and a port connection list.

Listing 6: An instantiation of the mux2 module

```
|     mux2 storemux
|     (
|         .sel(storemux_sel),
|         .a(sr1),
|         .b(dest),
|         .f(storemux_out)
|     );
```

Here, we instantiated a two-input mux called `storemux` and connected the mux select signal to `storemux_sel`, the inputs to `sr1` and `dest`, and the output to `storemux_out`. Because the default width of the two-input mux is 16, we need to specify the correct width using a parameter map between the component type and name.

Listing 7: storemux instantiation with correct width parameter

```
|     mux2 #(.width(3)) storemux
|     (
|         .sel(storemux_sel),
```

```

        .a(sr1),
        .b(dest),
        .f(storemux_out)
    );

```

For the port map, we connected the ports with explicitly named connections. Another way to connect the ports is using positional mapping, where the connections are made based on the order they are defined in the module declaration. Positional mapping also works for parameters.

Listing 8: Alternative instantiation using positional mapping

```

mux2 #(3) storemux
(
    storemux_sel,
    sr1,
    dest,
    storemux_out
);

```

Named mapping is usually preferred over positional mapping because it can make errors more apparent.

Now, instantiate the rest of the components and connect them with the appropriate signals. Use [Appendix F](#) as a guide for finishing the datapath layout.

4.1.3 Create the controller

Next, we create the controller for the processor as a state machine in SystemVerilog. A skeleton controller is given in *control.sv* which you can use to follow along in this section. The basic structure for a state machine can be written in the following manner:

Listing 9: Basic state machine structure

```

import lc3b_types::*; /* Import types defined in lc3b_types.sv */

module control
(
    /* Input and output port declarations */
);

enum int unsigned {
    /* List of states */
} state, next_states;

always_comb
begin : state_actions
    /* Default output assignments */
    /* Actions for each state */
end

always_comb
begin : next_state_logic
    /* Next state information and conditions (if any)
     * for transitioning between states */
end

always_ff @(posedge clk)

```

```

begin: next_state_assignment
    /* Assignment of next state on clock edge */
end

endmodule : control

```

We'll walk through the code for the controller while adding the functionality for the ADD instruction. The first line simply imports the types that are defined in *lc3b_types.sv*. Next, the input and output ports need to be specified.

Listing 10: Controller port declaration

```

module control
(
    input clk,

    /* Datapath controls */
    input lc3b_opcode opcode,
    output logic load_pc,
    output logic load_ir,
    output logic load_regfile,
    output lc3b_aluop aluop,
    /* et cetera */

    /* Memory signals */
    input mem_resp,
    output logic mem_read,
    output logic mem_write,
    output lc3b_mem_wmask mem_byte_enable
);

```

The state and next_state variables are of an enumerated type that contains the names of all the states. Add the states that will be needed for the ADD instruction¹ (see Appendices C and E).

Listing 11: Additional states for ADD instruction.

```

enum int unsigned {
    fetch1,
    fetch2,
    fetch3,
    decode,
    s_add
} state, next_state;

```

In the following always block, assign the default values and state actions.

Listing 12: Additional state actions for ADD.

```

always_comb
begin : state_actions
    /* Default assignments */
    load_pc = 1'b0;
    load_ir = 1'b0;
    load_regfile = 1'b0;
    aluop = alu_add;
    mem_read = 1'b0;
    mem_write = 1'b0;

```

¹We prepend "s_" to state names to avoid conflicts with SystemVerilog keywords

```

mem_byte_enable = 2'b11;
/* et cetera (see Appendix E) */

case(state)
  fetch1: begin
    /* MAR <= PC */
    marmux_sel = 1;
    load_mar = 1;

    /* PC <= PC + 2 */
    pcmux_sel = 0;
    load_pc = 1;
  end

  fetch2: begin
    /* Read memory */
    mem_read = 1;
    mdrmux_sel = 1;
    load_mdr = 1;
  end

  fetch3: begin
    /* Load IR */
    load_ir = 1;
  end

  decode: /* Do nothing */;

  s_add: begin
    /* DR <= SRA + SRB */
    aluop = alu_add;
    load_regfile = 1;
    regfilemux_sel = 0;
    load_cc = 1;
  end

  default: /* Do nothing */;

endcase
end

```

In the case statement, we specify the state transitions for each state. If a state is not listed, then the next state will be `fetch1`. The `next_state = state` line, in conjunction with the transition information from the `fetch2` state, implies that we will stay in the same state until the memory has responded. After the next state logic is in place, all that is left is to implement the next state assignment.

Listing 13: The next state assignment.

```

always_ff @(posedge clk)
begin : next_state_assignment
  state <= next_state;
end

```

The `@(posedge clk)` means that the `always_ff` block will execute on the positive edge of the clock.

4.1.4 Connect the datapath and controller

The *mp0.sv* file contains is the top-level module. The hierarchy of the project can be viewed under the **Hierarchy** tab. You need to connect the datapath and controller you just finished. To do this, follow a similar method as you did to connect components within the datapath. Declare the relevant internal signals and instantiate (and connect) the two modules.

Try compiling your design by selecting **Processing** → **Start Compilation** from the toolbar. Fix any errors you may have. You have now designed a processor with an ADD instruction. Finish the controller for all other instructions by following Appendices [C](#), [D](#), and [E](#), then move on to testing.

5 Analysis and functional verification

After the design has been entered, you will perform RTL simulation to verify the correctness of the design. For the simulation, the design will be hooked up to a testbench containing a generated clock signal and a model of the memory provided in *memory.sv*.

5.1 Testbench creation

Open *mp0_tb.sv* in Quartus (**File** → **Open**) and familiarize yourself with the contents. This file will be the top level of the testbench (mp0 is still the top level of the design). The testbench template contains a clock generator and two instantiated modules: your mp0 design (the design under test, or DUT) and a model of the memory.

The memory model is provided as a behavioral SystemVerilog file *memory.sv*. The model reads memory contents from the *memory.lst* file in the *simulation/modelsim/* directory of your project.

To configure Quartus so that the testbench is loaded when you simulate your design, select **Assignments** → **Settings...** from the Quartus menu bar and select **Simulation** under **EDA Tool Settings** in the left side panel. Under NativeLink settings, select **Compile test bench** and then **Test Benches...** on the right side.

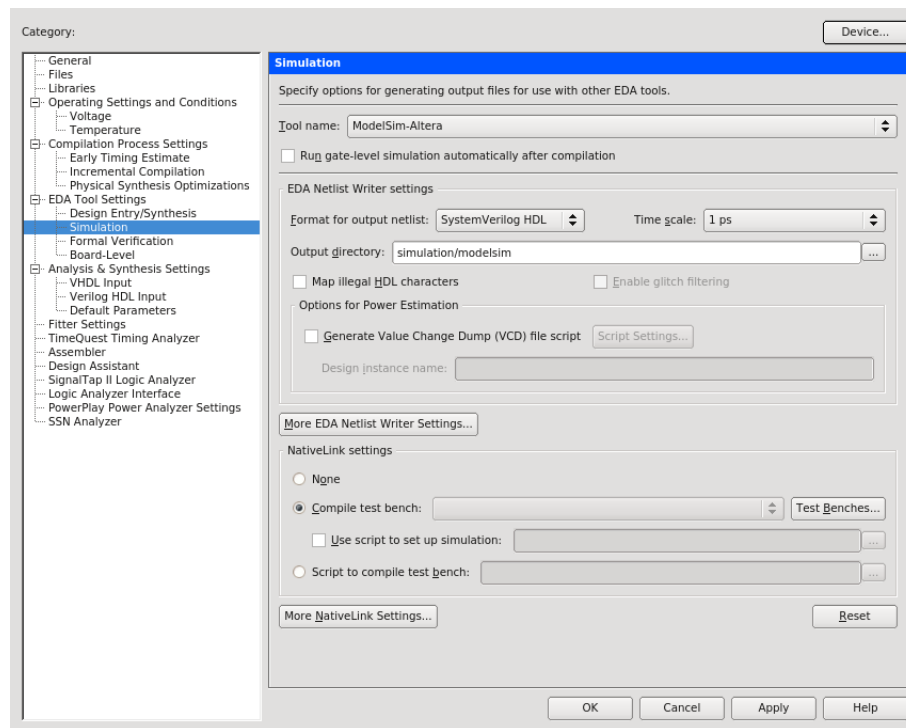


Figure 1: Simulation options

Click **New...** to create a new testbench with the following settings:

Test bench name: **mp0_tb**

Top level module in test bench: **mp0_tb**

End simulation at: **200 ns**

Under the **Test bench and simulation files** section, add the files *mp0_tb.sv* and *memory.sv*. Click **OK** several times to save the settings.

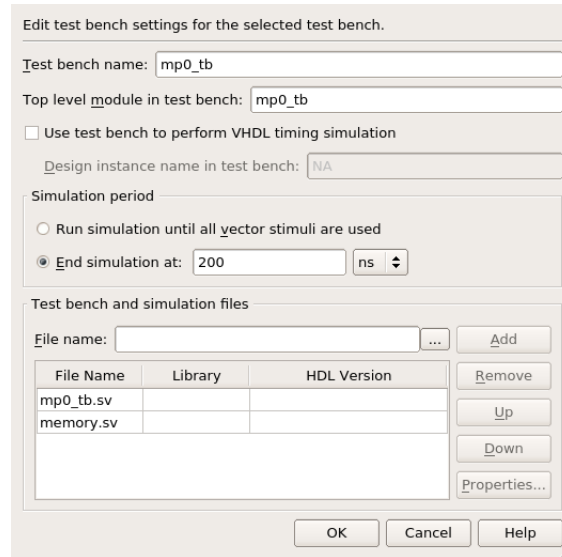


Figure 2: Test bench options

5.1.1 Testbench memory initialization

To test the design, we will load the memory with an LC-3b program. See Appendix A for how to load an assembly program into the design. Use the instructions to load the given test code in *testcode/mp0test.asm*.

5.2 RTL simulation

5.2.1 Verify EDA tool settings

Under **Assignments** → **Settings...** select **EDA Tool Settings** on the left side pane. Make sure that **ModelSim-Altera** is selected as the simulation tool with the format **SystemVerilog HDL** then click OK. Also, under **Tools** → **Options...** select **EDA Tool Options** and make sure the path to the ModelSim-Altera binary is */software/altera/13.1/modelsim_ase/linuxaloem*.

5.2.2 Run RTL simulation

Select **Tools** → **Run Simulation Tool** → **RTL Simulation**. Modelsim should open up and simulate the testbench for a short time. Status and error messages are displayed in the transcript pane at the bottom of the window. A prompt in the same pane allows you to enter commands for Modelsim. Before continuing with RTL simulation, we will first set some user interface options.

Set the default radix

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

Change to a fixed width font

To change your default font, select **Tools** → **Edit Preferences...** Then, under the **Window List** section, select **Wave Windows**. Within the **Font** section, click **treeFont** in the left pane and then click **Choose...** Select your favorite fixed width font (e.g., fixed, Consolas, Courier New, etc), set a comfortable size and click **OK** until you return to the main Modelsim window.

Set timeline time unit to ns

Select the **Wave → Wave Preferences...** Then, open the **Grid & Timeline** tab and under the **Timeline Configuration** section, change the time units to ns. Click **OK** to save the changes. If you don't see the **Wave** menu, click in the wave window first. Instead of the **Wave** menu, you can also click the blue icon near the bottom left of the wave window.



Figure 3: Grid and timeline options

There are multiple ways of viewing the functionality of your design, we introduce a few options here.

5.2.2.1 Wave traces

If the wave pane is not open already, select **View → Wave** to open it. To add signals to the wave, drag them from the structure and objects panes on the left side to the wave pane. For now, find the register file in your design (e.g., **mp0_tb → dut → datapath → regfile**) and drag the data object (from the object pane) to the wave pane. You can also do it by right clicking on the signal and select **Add Wave** or using the shortcut **Ctrl+W**. Expand the newly created node by clicking the + sign to reveal the individual registers.

At the prompt in the transcript window, type the following to restart the simulation and then run it for a specified amount of time.

```
> restart -f
> run 20000ns
```

Note that you can combine commands on the same line by separating them with a semicolon, which will look like this.

```
> restart -f; run 20000ns
```

After running the commands, you should see the wave window being populated with signal values. If you set the default radix correctly above, the values should be displayed in hexadecimal. You can change the radix of individual signals by right clicking the name of the signal and choosing a radix in the context menu.

To add additional signals to the wave, simply drag them from structure and objects panes on the left. You can reorder signals by dragging their names in the wave pane. Signals can also be grouped or colored for easy viewing via the right-click context menu (**Group...** or **Properties...**).

Once you are satisfied with the layout of the wave window, you can save the layout for future use by selecting **File → Save Format...** and specifying a location and name (the default name is wave.do). This will save the wave format as a Modelsim macro file. Next time you open Modelsim, type the following to run the macro file.

```
> do wave.do
```

5.2.2.2 Lists

Lists give a textual representation of signals over time and can be used to view signal values at certain events. To open the list pane, select **View → List** or type **list** at the prompt. Signals can be added by dragging and dropping into the list pane. Drag the **mem_address**, **mem_wdata**, **mem_write**, and **mem_byte_enable** signals to the list window. Change the signal properties (select the signal name then select **View → Properties...**) so that all values are in the appropriate radix if necessary.

By default, each time a signal in the list window changes, it generates a new entry in the list. For some signals, you may not want a new line every time its value changes. In this case, we only want our list to generate entries

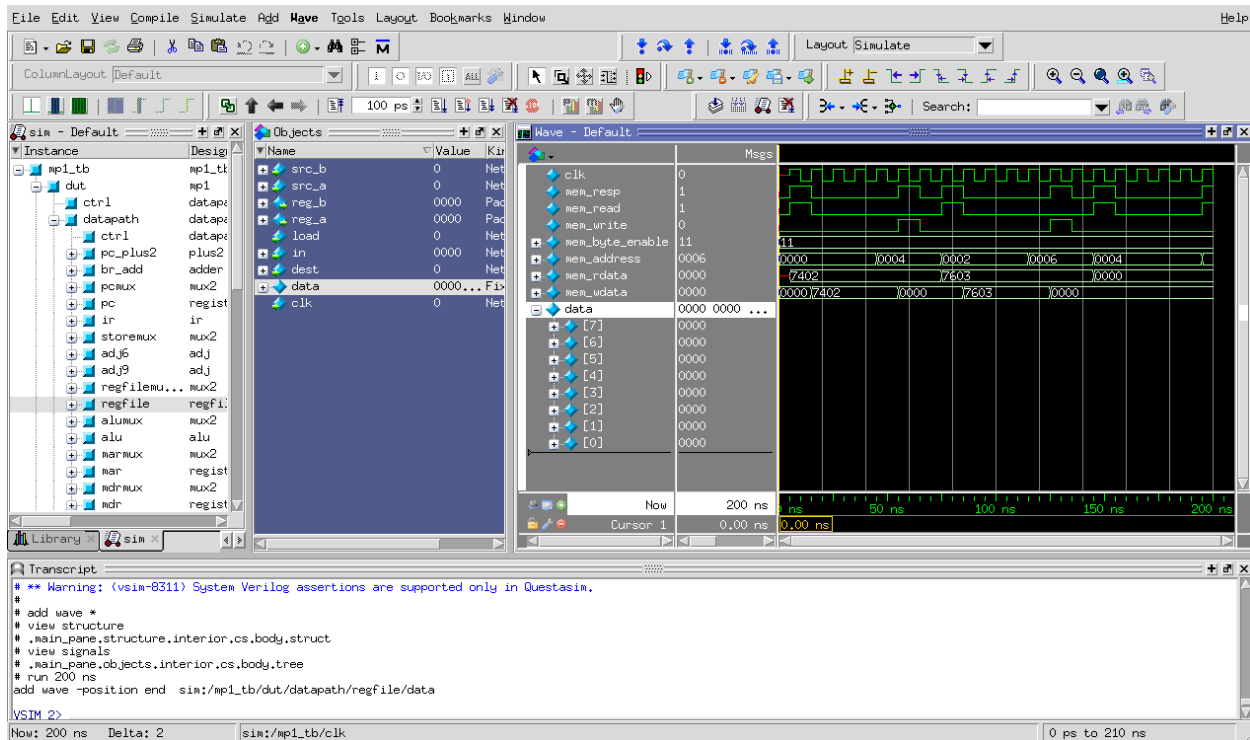


Figure 4: The wave trace window

when we are actually writing to our memory (when mem_write becomes active). Therefore, we only want to trigger entries to be added to our list when mem_write changes. To accomplish this, select the mem_address, mem_wdata, and mem_byte_enable signals, choose **View → Properties...**, and select **Does not trigger line**.

5.2.2.3 Memory lists

Memory lists allow us to view the contents of memory at the current point in the simulation. To see the memory list, select **View → Memory List** or type `view memory` at the prompt. Double click the memory that you want to view to show its contents. For now, choose the memory from the testbench. A new pane will open with the memory contents. To make the memory contents easier to read, right click in the memory pane and select properties, then change the address and data radix to **hexadecimal** and under **Line Wrap** choose to display 2 (or your favorite number) words per line.

5.2.3 Testing your design

With the above tools, you should be able to verify the functionality of your design. You can use the LC3bIDE to run any test code to determine the correct behavior for the code and see if the operation of your design matches the expected behavior. You should write your own test code in LC-3b assembly to test corner cases that might occur in your design and load it into memory as described in Appendix A.

In Modelsim, you can restart the current simulation by typing `restart -f` and run the simulation by typing `run 2000ns` (or a time interval of your choosing).

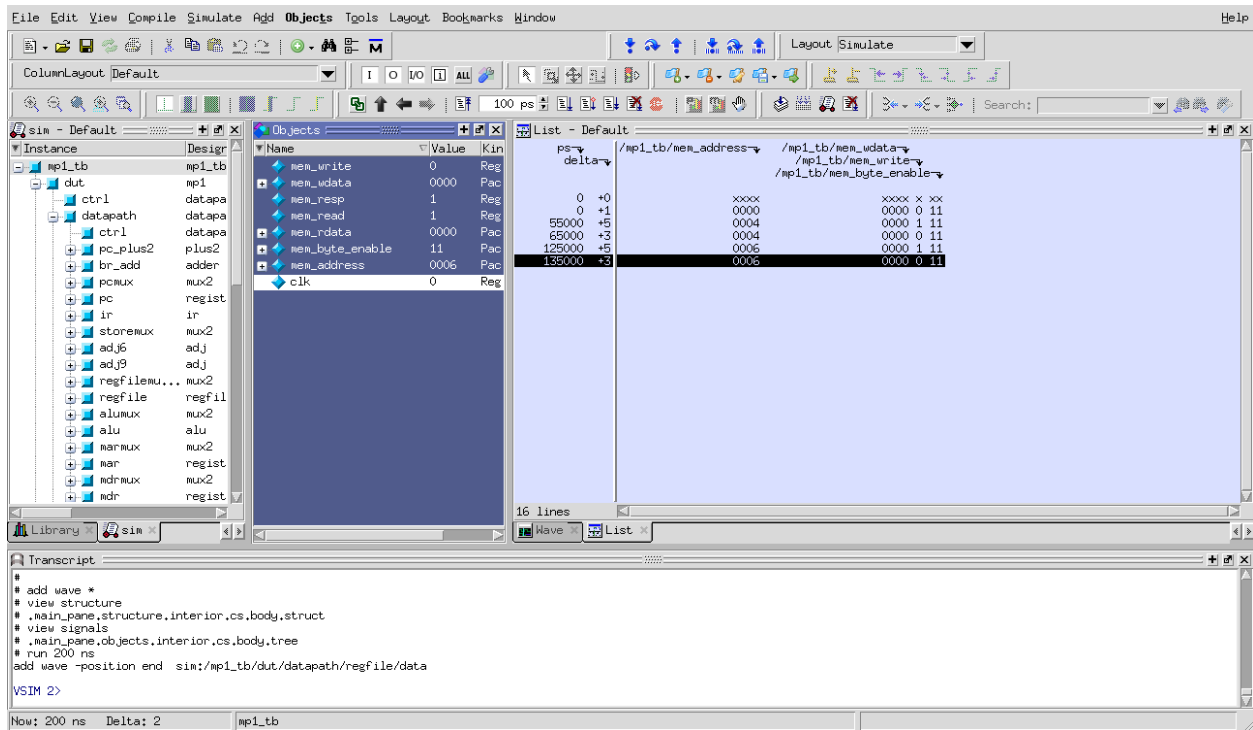


Figure 5: The lists window

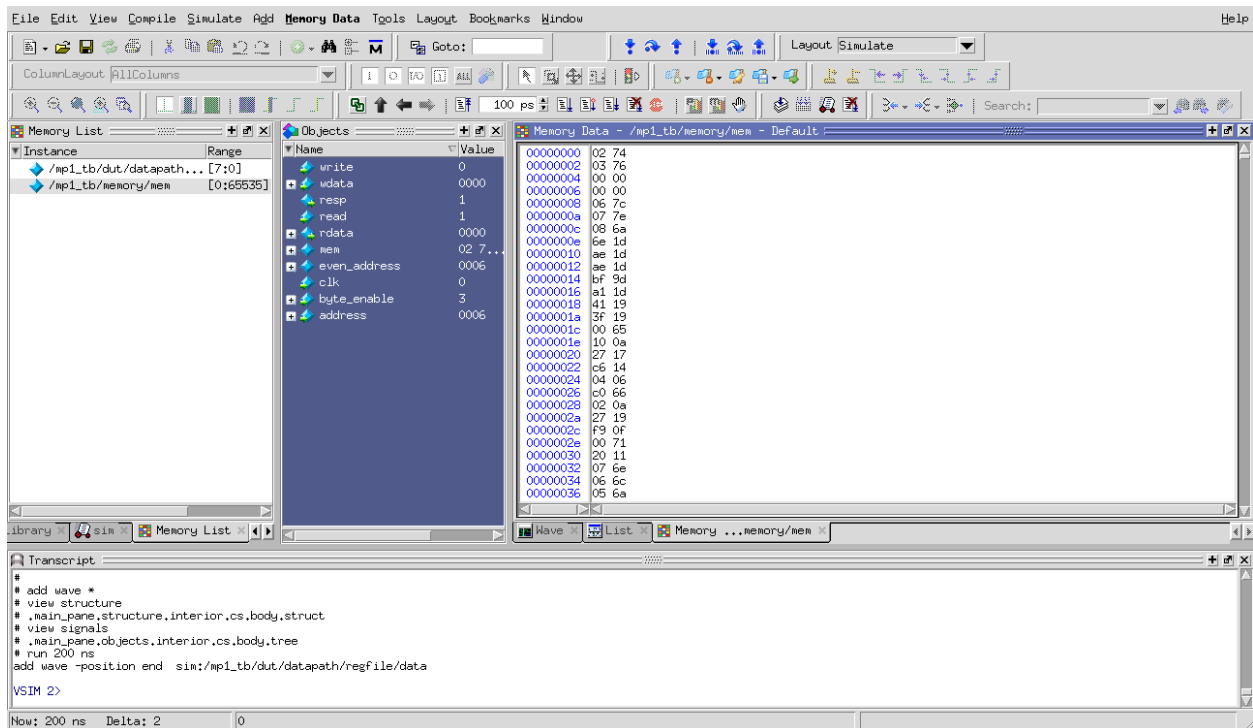


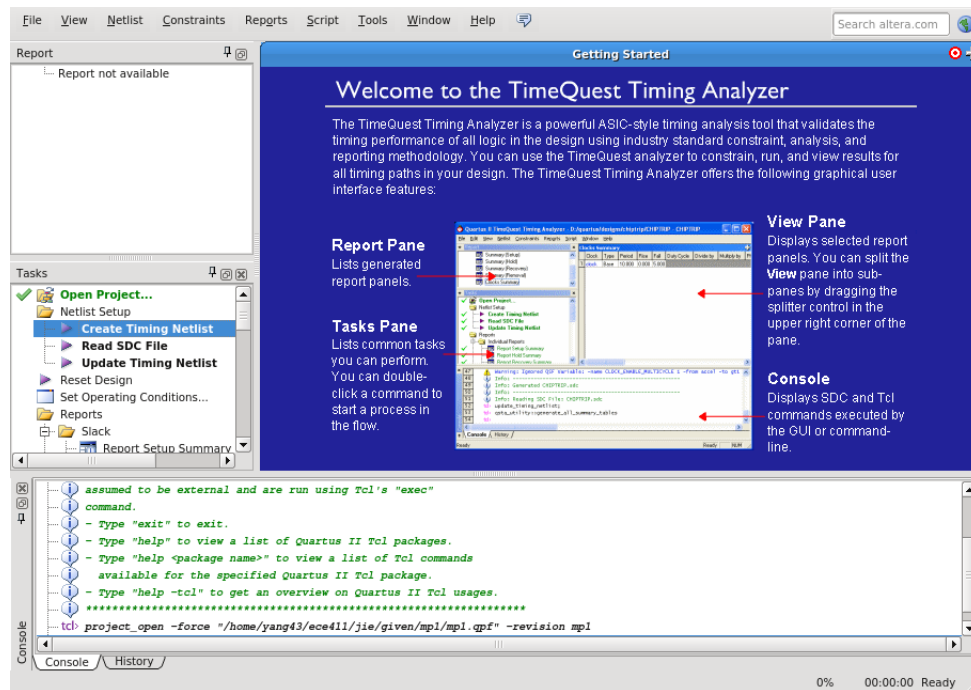
Figure 6: The memory lists window

6 Timing analysis

Once the design is functionally correct, we need to make sure that timing requirements are met with respect to a given clock frequency. For this MP, the target frequency is *100MHz* (10ns period).

To begin the timing analysis, first compile your design by selecting **Processing** → **Start Compilation** (or press Ctrl+L). If you take a look at the compilation report under **TimeQuest Timing Analyzer**, you should see a lot of failures due to Quartus assuming your target frequency is 1GHz by default. Note: the failures will show up as list items whose names are red.

Open up the TimeQuest Timing Analyzer by selecting **Tools** → **TimeQuest Timing Analyzer**. Double click **Create Timing Netlist** in the Tasks pane on the left to generate a timing netlist for analysis.



6.1 Set constraints

6.1.1 Set clock constraint

Select **Constraints** → **Create Clock...** from the menu bar and specify a clock with 10ns period. For **Targets**, click the **ellipses** to the right, then click **List** to get a list of ports.

Select **clk** and add it to the list on the right side, then click **OK**. Note the SDC command field at the bottom of the Create Clock window. This command shows what constraint is being specified. Here you can type a command directly instead of navigating through the GUI. For now, click **Run** to create the constraint.

To verify that your clock was created correctly, scroll down in the Tasks pane and double click **Report Clocks** under **Diagnostics** to generate a clock summary.

It should show that clk is constrained to operate at 100 MHz. In the process, you should get a warning about clock uncertainty. To do this, select **Constraints** → **Derive Clock Uncertainty...** and click **Run**. The clock uncertainty is not calculated until you update the timing netlist.

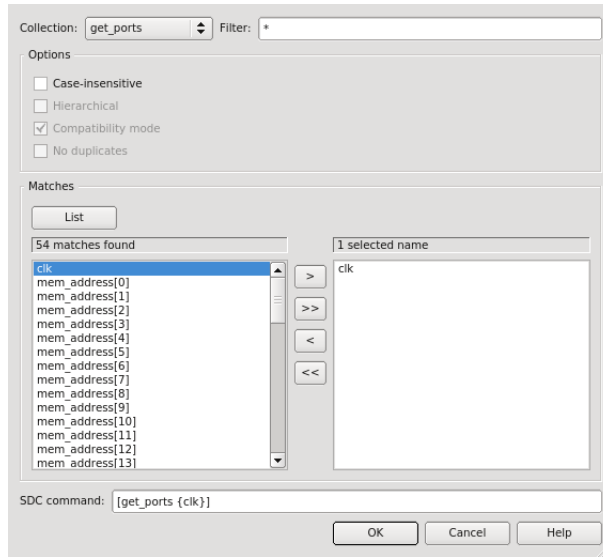


Figure 8: Selecting clock to constrain

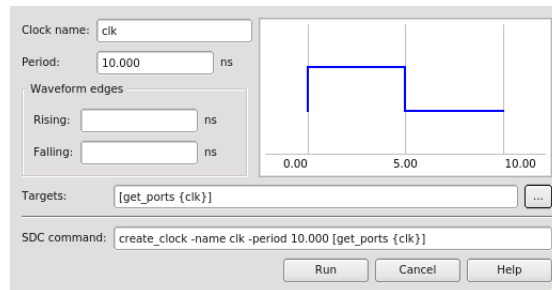


Figure 9: Specifying clock constraints

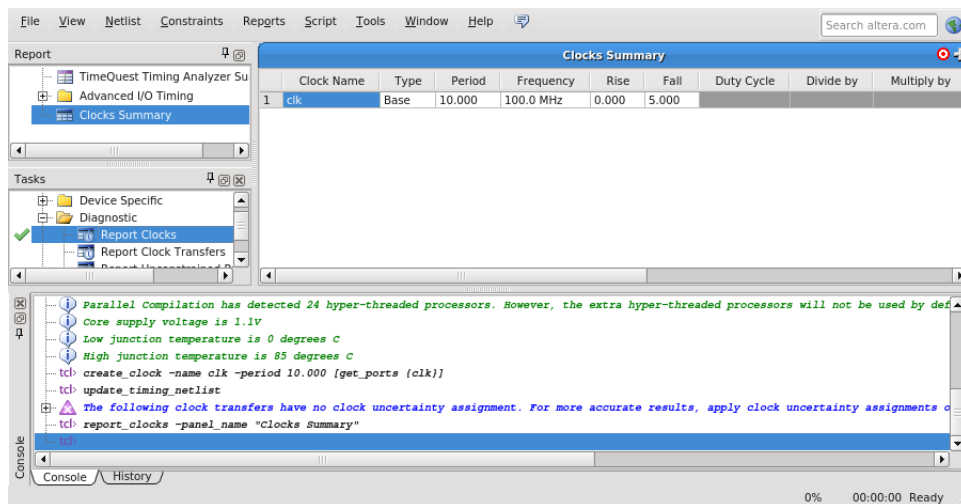


Figure 10: The clock report

6.1.2 Set input and output constraints

In addition to the clock constraint, input and output constraints to the top level ports must also be set. For simplicity, we will set all the input and output delays to zero. Select **Constraints** → **Set Input Delay...** and in the dialog

set Clock name to **clk**, set Delay value to **0**, under Targets type **[all_inputs]**, and click Run.

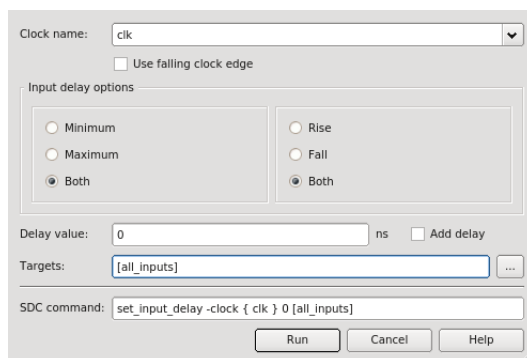


Figure 11: Specifying input constraints

Select **Constraints** → **Set Output Delay...** to set the output delays, the settings are the same as for input delays, except **[all_inputs]** is replaced with **[all_outputs]**.

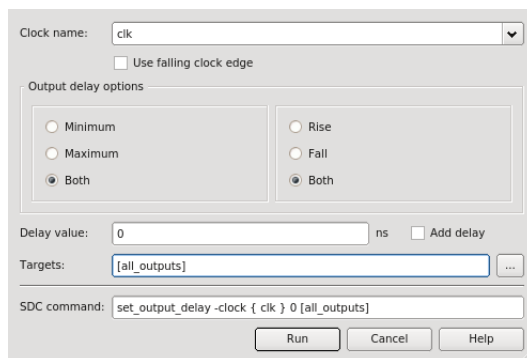


Figure 12: Specifying output constraints

6.2 Write SDC file

After setting all constraints, double click **Update Timing Netlist** in the Tasks pane. Now save the SDC (Synopsys Design Constraints) file by double clicking **Write SDC File...** in the Tasks pane (you need to scroll all the way down in the pane), specify the SDC file name and then click OK. The SDC file contains the commands that we specified above. To edit the constraints (e.g., to change the clock period or to constrain additional input/output ports), you can either use the GUI (like above) or edit the SDC file directly.

After the SDC File is written, it needs to be added to the project. Exit TimeQuest and select **Project** → **Add/Remove Files in Project...** in the main Quartus window. Choose the SDC file (by default it is named *mp0.out.sdc*) and add it to the project (make sure to look for "All Files" instead of only "Design Files" in the select file dialog).

6.3 Run Timing Analysis

After adding the SDC file to the project, run timing analysis again by double clicking **TimeQuest Timing Analysis** in the Tasks pane (alternatively you can run the full compilation via **Processing** → **Start Compilation**). If all goes well, the Compilation Report should indicate that no timing constraints were violated.

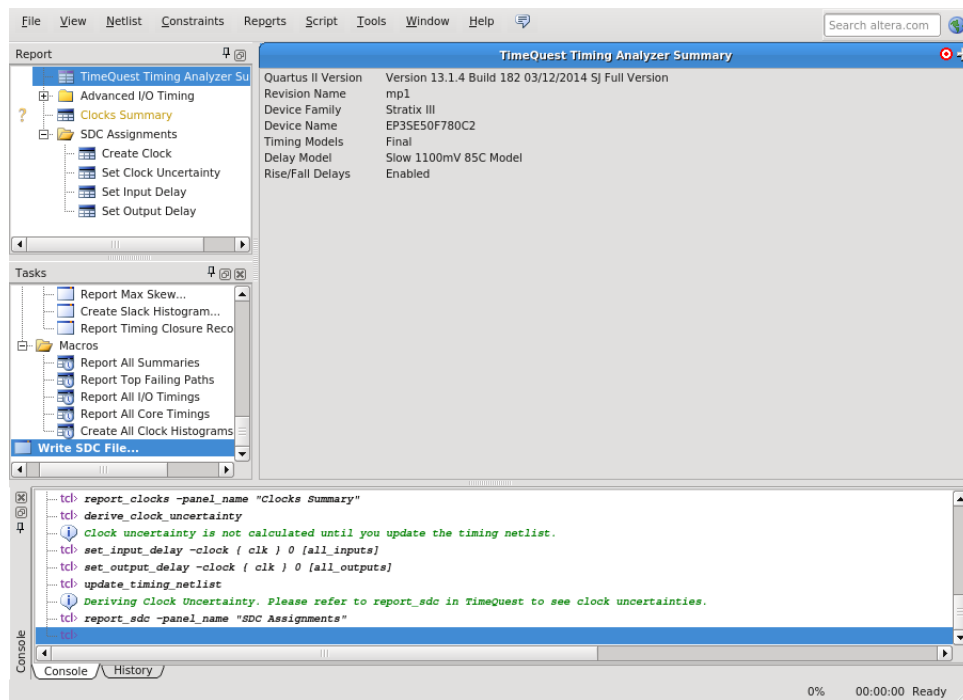


Figure 13: Writing the SDC file

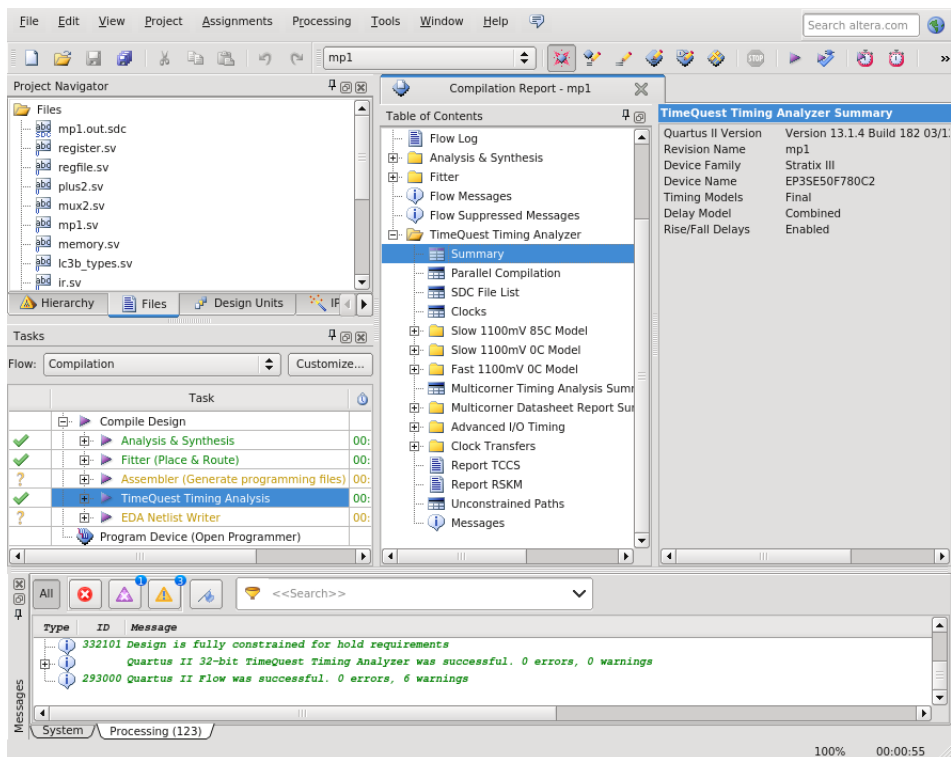


Figure 14: The timing analysis summary

7 Final hand-in

Write a short program using the LC-3b assembly language to calculate 5! (five factorial). Your program *must be iterative*. Note that your datapath does not yet contain support for immediate adds, so you must use load instructions to initialize registers. Reference the sample program located at `testcode/mp0test.asm` for assistance with the example instructions you can use. The factorial program should be flexible to calculate any other integer factorials like 4!, 6!, 7!, etc. by changing only one variable. It does not have to handle 0! or negative factorials. Like the sample program, your code must end in an infinite loop. This will make simulation a lot easier. Please see Appendix A for a description of how to load a program into your processor.

You need to hand in the following items to the ECE 411 dropbox on the third floor of ECEB at the completion of the assignment:

- A commented printout of your assembly program. Comments should be more than an explanation of what a particular instruction does.
- A wave trace showing the first 1000ns of your program's operation. Make sure the signals below are present on your wave with data in the order we have listed. If you add a signal after running a simulation, it will not show data. The wavetrace must contain the following signals (in order):

1. clk
2. pc_out
3. mem_address
4. mem_read
5. mem_rdata
6. mem_write
7. mem_byte_enable
8. mem_wdata
9. register file contents (expanded)

"Register file contents" refers to the contents of each register. Please expand this signal so that each register is shown on its own line. Make sure all signal values are shown in hexadecimal! Changing to hex is covered in Section 5.2.2.

Note: To print waveform in white background, click the wave window and choose **File → Print Postscript...**, then choose the range (e.g. From 0 ns to 1000 ns) according to the submission requirement and modify the file name (.ps). Print the created .ps file. Do NOT take the screenshot of the waveform which will make the fonts not so legible as well as leave the background black wasting ink.

Warning: Be sure that the wave traces actually have values on them, not just signal names. If you add signals to a wave after simulating, you must re-run the simulation in order for the new values to be displayed.

- A wave trace showing the final 1000ns including the first iteration of the infinite loop (The PC must be shown to repeat). Highlight the register which contains the value of five factorial. The wave trace must include the same signals as above 2 (first 1000 ns wave trace). Also, all other procedures are identical to 2 (initial wave trace).
- A printout of your timing analysis report containing the sections listed below. The report can be found in the `output_files/mp0.sta.rpt` file after running the TimeQuest Timing Analysis task (Ctrl+Shift+T). Note that the report tends to be very long. You may print only the requested sections. If you submit the entire report, you must highlight the requested sections.

The timing analysis report should contain the following sections (as listed in the report's table of contents):

1. TimeQuest Timing Analyzer Summary
2. Clocks

3. Slow 1100mV 85C Model Fmax Summary

4. TimeQuest Timing Analyzer Messages (make sure that no messages are being suppressed)

An example of the requested sections is given in Appendix H. If you are having difficulty understanding any aspect(s) of the MP, ask a TA for assistance. Material from this machine problem will be used in subsequent MPs and may appear on exams. It is your responsibility to make sure that MP0 is complete and functioning correctly before proceeding with future MPs.

8 Grading rubric

Item	Pts	%
Code	6	30
First 1000ns	2	10
Final 1000ns	9	45
Timing report	3	15
Total	20	100

A Loading programs into your design

To load a program into your design, you need to generate a memory initialization file, *memory.lst*, that is placed into the simulation directory *mp0/simulation/modelsim/* (this directory may need to be created if modelsim hasn't been run yet). The *load_memory.sh* script located in the *bin/* directory can be used to do this.

The *load_memory.sh* script takes an LC-3b assembly file as input, assembles it into an LC-3b object file, and converts the object file into a suitable format for initializing the testbench memory. The script assumes that your project directory structure is set up according to the instructions in this document. If not, you'll need to edit the paths for the memory initialization file and assembler at the top of the script. The default settings are shown below.

```
# Settings
DEFAULT_TARGET=$HOME/ece411/mp0/simulation/modelsim/memory.lst
ASSEMBLER=$HOME/ece411/bin/LC3bAssembler
ADDRESSABILITY=1
```

To execute *load_memory.sh*, you need to supply the name of an LC-3b assembly file and, optionally, the location to write *memory.lst*.

```
$ ./load_memory.sh <asm-file> [memory-file]
```

By default, the script places the output at *~/ece411/mp0/simulation/modelsim/memory.lst*. Note that you should specify the path to *load_memory.sh* if you're not already in the *bin/* directory.

For example, suppose we want to generate a memory initialization file from the program *~/ece411/testcode/my-test.asm* and place the result in the default target path.

```
$ cd ~/ece411/bin/
$ ./load_memory.sh ~/ece411/testcode/my-test.asm
```

If successful, you should see a message similar to

```
Assembled ~/ece411/testcode/my-test.asm and wrote memory contents to
~/ece411/mp0/simulation/modelsim/memory.lst.
```

B Instruction set description

Name	Opcode	Description
ADD	0001	Put the sum of registers SR1 and SR2 into register DR
AND	0101	Put the logical AND of registers SR1 and SR2 into register DR
BR	0000	Conditionally branch if a matching condition is present
LDR	0110	Load register DR from the location specified by BaseR + offset6
NOT	1001	Put the bitwise complement of register SR into register DR
STR	0111	Store the word from register SR at the memory location BaseR + offset6

C RTL

C.1 FETCH process

State	Data	Control
fetch1	MAR←PC; PC←PC + 2;	marmux_sel←1; load_mar←1; pcmux_sel←0; load_pc←1;
fetch2	while (mem_resp == 0) MDR←M[MAR];	mdrmux_sel←1; load_mdr←1; mem_read←1;
fetch3	IR←MDR;	load_ir←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 and there is code in the next_state logic)

C.3 ADD instruction

State	Data	Control
FETCH		
DECODE		
s_add	DR←A + B;	aluop←alu_add; load_regfile←1; load_cc←1;

C.4 AND instruction

State	Data	Control
FETCH		
DECODE		
s_and	DR←A & B;	aluop←alu_and; load_regfile←1; load_cc←1;

C.5 NOT instruction

State	Data	Control
FETCH		
DECODE		
s_not	DR ← NOT(A);	aluop ← alu_not; load_regfile ← 1; load_cc ← 1;

C.6 BR instruction

State	Data	Control
FETCH		
DECODE		
br	// NONE	// NONE
br_taken	PC ← PC + SEXT(IR[8:0] « 1);	pcmux_sel ← 1; load_pc ← 1;

C.7 LDR instruction

State	Data	Control
FETCH		
DECODE		
calc_addr	MAR ← A + SEXT(IR[5:0] « 1);	alumux_sel ← 1; aluop ← alu_add; load_mar ← 1;
ldr1	MDR ← M[MAR];	mdrmuxsel ← 1; load_mdr ← 1; mem_read ← 1;
ldr2	DR ← MDR;	regfilemux_sel ← 1; load_regfile ← 1; load_cc ← 1;

C.8 STR instruction

State	Data	Control
FETCH		
DECODE		
calc_addr	MAR ← A + SEXT(IR[5:0] « 1);	alumux_sel ← 1; aluop ← alu_add; load_mar ← 1;
str1	MDR ← SR;	storemux_sel ← 1; aluop ← alu_pass; load_mdr ← 1;
str2	M[MAR] ← MDR;	mem_write ← 1;

D CPU

(a) Control to datapath		(b) Datapath to control	
Name	Type	Name	Type
load_pc	logic	opcode	lc3b_opcode
load_ir	logic	branch_enable	logic
load_regfile	logic		
load_mar	logic		
load_mdr	logic		
load_cc	logic		
pcmux_sel	logic		
storemux_sel	logic		
alumux_sel	logic		
regfilemux_sel	logic		
marmux_sel	logic		
mdrmux_sel	logic		
aluop	lc3b_aluop		
(c) Control to memory		(d) Memory to control	
Name	Type	Name	Type
mem_read	logic	mem_resp	logic
mem_write	logic		
mem_byte_enable	logic [1:0]		
(e) Datapath to memory		(f) Memory to datapath	
Name	Type	Name	Type
mem_address	lc3b_word	mem_rdata	lc3b_word
mem_wdata	lc3b_word		

Table 1: CPU connections

E Control

E.1 Signals and defaults

Name	Default value
load_pc	1'b0
load_ir	1'b0
load_regfile	1'b0
load_mar	1'b0
load_mdr	1'b0
load_cc	1'b0
pcmux_sel	1'b0
storemux_sel	1'b0
alumux_sel	1'b0
regfilemux_sel	1'b0
marmux_sel	1'b0
mdrmux_sel	1'b0
aluop	alu_add
mem_read	1'b0
mem_write	1'b0
mem_byte_enable	2'b11

E.2 Control diagram

See Appendix C for control state actions.

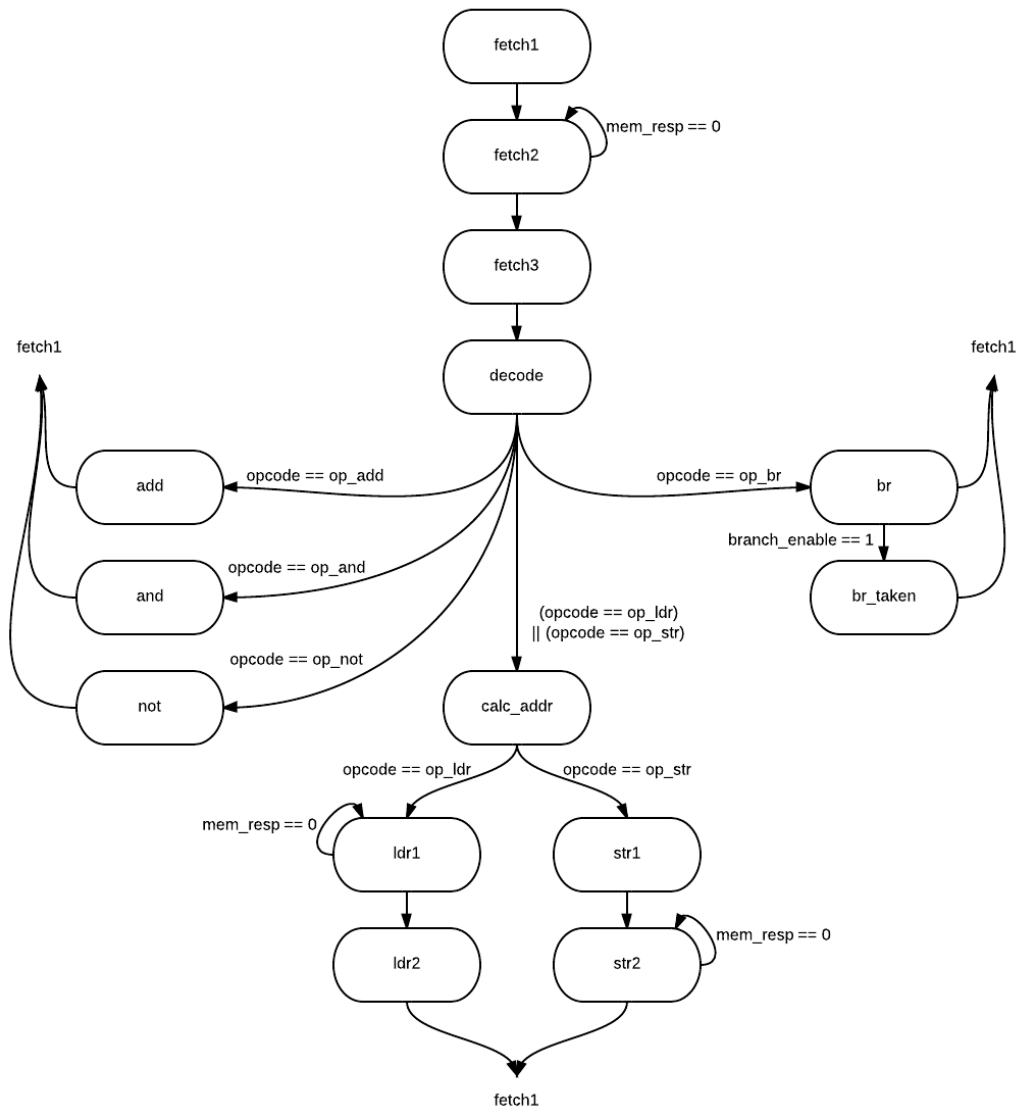


Figure 15: The LC-3b control state diagram

F Datapath

F.1 Signals

Name	Type	Origin	Destination
clk	logic	input port	PC, IR, REGFILE, MAR, MDR, CC
load_pc	logic	control	PC
load_ir	logic	control	IR
load_regfile	logic	control	regfile
load_mar	logic	control	MAR
load_mdr	logic	control	MDR
load_cc	logic	control	CC
pcmux_sel	logic	control	pcmux
storemux_sel	logic	control	storemux
alumux_sel	logic	control	alumux
regfilemux_sel	logic	control	regfilemux
marmux_sel	logic	control	marmux
mdrmux_sel	logic	control	mdrmux
aluop	lc3b_aluop	control	ALU
sr1	lc3b_reg	IR	storemux
sr2	lc3b_reg	IR	regfile
dest	lc3b_reg	IR	storemux, regfile, cccomp
storemux_out	lc3b_reg	storemux	regfile
sr1_out	lc3b_word	regfile	ALU
sr2_out	lc3b_word	regfile	alumux
offset6	lc3b_offset6	IR	adj6
offset9	lc3b_offset9	IR	adj9
adj6_out	lc3b_word	adj6	alumux
adj9_out	lc3b_word	adj9	br_add
pcmux_out	lc3b_word	pcmux	PC
alumux_out	lc3b_word	alumux	ALU
regfilemux_out	lc3b_word	regfilemux	regfile, gencc
marmux_out	lc3b_word	marmux	MAR
mdrmux_out	lc3b_word	mdrmux	MDR
alu_out	lc3b_word	ALU	regfilemux, marmux, mdrmux
pc_out	lc3b_word	PC	pc_plus2, br_add, marmux
br_add_out	lc3b_word	br_add	pc_mux
pc_plus2_out	lc3b_word	pc_plus2	pc_mux
mem_address	lc3b_word	MAR	output port
mem_wdata	lc3b_word	MDR	output port, regfilemux, IR
mem_rdata	lc3b_word	input port	mdrmux
opcode	lc3b_opcode	IR	control
branch_enable	logic	cccomp	control
gencc_out	lc3b_nzp	gencc	CC
cc_out	lc3b_nzp	CC	cccomp

E.2 Datapath diagram

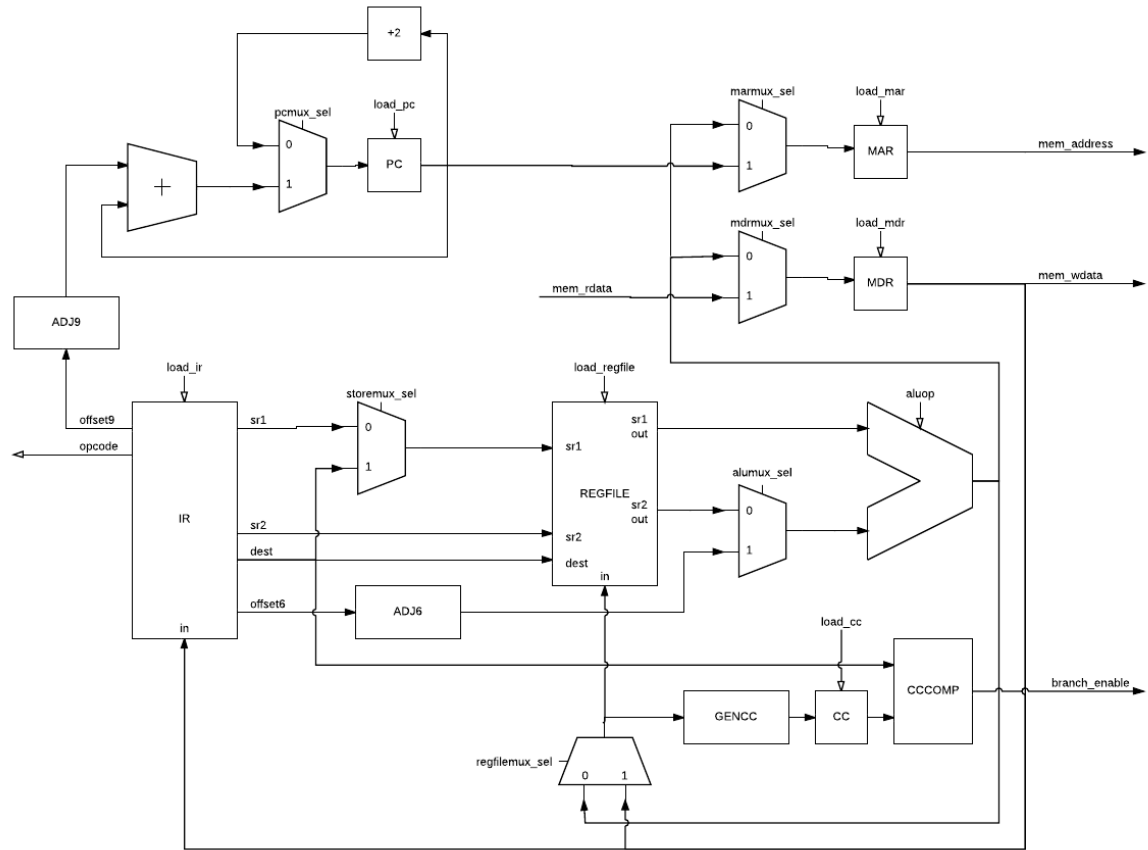


Figure 16: The LC-3b datapath diagram

G Components

G.1 Ports for mux2

Name	Direction	Type
sel	input	logic
a	input	logic [width-1:0]
b	input	logic [width-1:0]
f	output	logic [width-1:0]

G.2 Parameters for mux2

Name	Default value
width	16

G.3 SystemVerilog module for mux2

Listing 14: The mux2 module

```
module mux2 #(parameter width = 16)
(
    input sel,
    input [width-1:0] a, b,
    output logic [width-1:0] f
);

always_comb
begin
    if (sel == 0)
        f = a;
    else
        f = b;
end

endmodule : mux2
```

H Example timing analysis report

Listing 15: Example timing analysis report

```
+-----+
; TimeQuest Timing Analyzer Summary ;
+-----+
; Quartus II Version ; Version 13.1.4 Build 182 03/12/2014 SJ Full Version ;
; Revision Name      ; mp0 ;
; Device Family      ; Stratix III ;
; Device Name        ; EP3SE50F780C2 ;
; Timing Models      ; Final ;
; Delay Model        ; Combined ;
; Rise/Fall Delays   ; Enabled ;
+-----+

+-----+
; Clocks ;
+-----+
; Clock Name ; Type ; Period ; Frequency ; Rise ; Fall ; [...] ; Targets ;
+-----+
; clk      ; Base ; 10.000 ; 100.0 MHz ; 0.000 ; 5.000 ; ; { clk } ;
+-----+

+-----+
; Slow 1100mV 85C Model Fmax Summary ;
+-----+
; Fmax      ; Restricted Fmax ; Clock Name ; Note ;
+-----+
; 142.71 MHz ; 142.71 MHz ; clk ; ;
+-----+

+-----+
; TimeQuest Timing Analyzer Messages ;
+-----+
Info: *****
Info: Running Quartus II 32-Bit TimeQuest Timing Analyzer
      Info: Version 13.1.4 Build 182 03/12/2014 SJ Full Version
      Info: Processing started: Mon Aug 25 00:00:00 2014
Info: Command: quartus_sta mp0 -c mp0
Info: qsta_default_script.tcl version: #1
Info (20030): Parallel compilation is enabled and will use 4 of the 4 processors detected
Info (21077): Core supply voltage is 1.1V
Info (21077): Low junction temperature is 0 degrees C
Info (21077): High junction temperature is 85 degrees C
Info (332104): Reading SDC File: 'mp0ref.sdc'
Info: Found TIMEQUEST_REPORT_SCRIPT_INCLUDE_DEFAULT_ANALYSIS = ON
Info: Analyzing Slow 1100mV 85C Model
Info (332146): Worst-case setup slack is 2.993
      Info (332119): Slack      End Point TNS Clock
      Info (332119): =====

```

```

Info (332119): 2.993 0.000 clk
Info (332146): Worst-case hold slack is 0.308
Info (332119): Slack End Point TNS Clock
Info (332119): =====
Info (332119): 0.308 0.000 clk
Info (332140): No Recovery paths to report
Info (332146): Worst-case minimum pulse width slack is 4.373
Info (332119): Slack End Point TNS Clock
Info (332119): =====
Info (332119): 4.373 0.000 clk
Info: Analyzing Slow 1100mV OC Model
Info (332146): Worst-case setup slack is 3.381
Info (332119): Slack End Point TNS Clock
Info (332119): =====
Info (332119): 3.381 0.000 clk
Info (332146): Worst-case hold slack is 0.286
Info (332119): Slack End Point TNS Clock
Info (332119): =====
Info (332119): 0.286 0.000 clk
Info (332140): No Recovery paths to report
Info (332140): No Removal paths to report
Info (332146): Worst-case minimum pulse width slack is 4.373
Info (332119): Slack End Point TNS Clock
Info (332119): =====
Info (332119): 4.373 0.000 clk
Info: Analyzing Fast 1100mV OC Model
Info (332146): Worst-case setup slack is 5.146
Info (332119): Slack End Point TNS Clock
Info (332119): =====
Info (332119): 5.146 0.000 clk
Info (332146): Worst-case hold slack is 0.189
Info (332119): Slack End Point TNS Clock
Info (332119): =====
Info (332119): 0.189 0.000 clk
Info (332140): No Recovery paths to report
Info (332140): No Removal paths to report
Info (332146): Worst-case minimum pulse width slack is 4.654
Info (332119): Slack End Point TNS Clock
Info (332119): =====
Info (332119): 4.654 0.000 clk
Info (332101): Design is fully constrained for setup requirements
Info (332101): Design is fully constrained for hold requirements
Info: Quartus II 64-Bit TimeQuest Timing Analyzer was successful. 0 errors, 0 warnings
Info: Peak virtual memory: 732 megabytes
Info: Processing ended: Mon Aug 25 00:00:07 2014
Info: Elapsed time: 00:00:07
Info: Total CPU time (on all processors): 00:00:02

```

J Block diagram based design

In some situations, you might find it helpful to design using the graphical tools that Quartus provides. While we do not recommend this approach, we have provided some instructions below should you choose to go this route. You must still generate Verilog code for handing if you choose to design using the GUI tools.

To start the block diagram based design, first download a set of given files that give a foundation for the block diagram based design.


```
$ cd ~/ece411/
$ curl -L courses.engr.illinois.edu/ece411/mp/mp0/mp0_block.tar.xz \
$ | tar -xJv
```

The above command should create the `~/ece411/mp0_block/` directory which contains the files needed. next, open the project `mp0_block/mp0.qpf` in Quartus.

J.1 Add and name new blocks

In the section, you will learn how to add a component to the design. First, copy the code for `mux2` in Appendix G, and save it as `mux2.sv`. Then inside the new created SystemVerilog file select **File** → **Create/Update** → **Create Symbol Files for Current File**. A Compilation tab will pop up and tell you if the symbol is created successfully or not. If it complains about some error, you need to fix the error. Note that you can follow the same procedure for `.bdf` files to create symbol for upper hierarchy.

If it succeeded, `.bsf` will be created in the design directory. In case of (re)creating symbol from `.bdf` (block diagram) file, a window will pop-up to ask how you want to save the symbol file name when the symbol file will automatically be updated from `.sv` file.

Then, go to `datapath.bdf`, and click **Symbol Tool** or symbol . It will pop-up a window like in Figure 17.

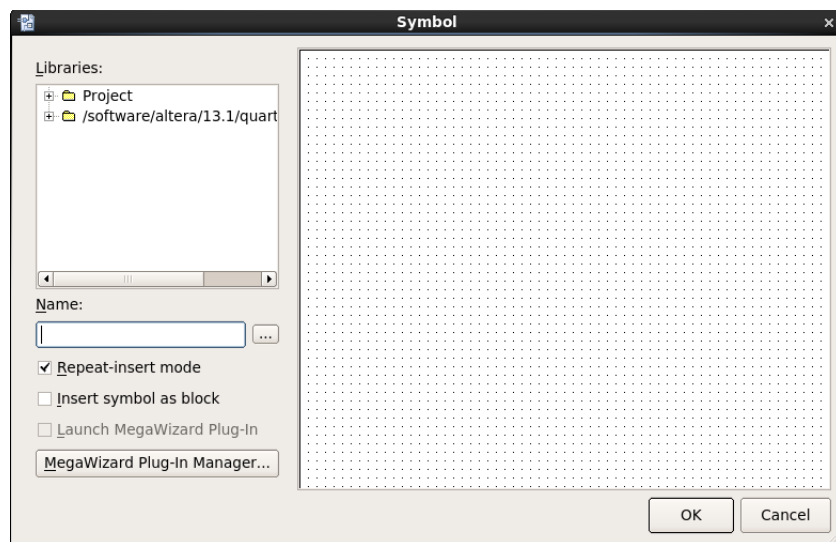


Figure 17: Symbol tool

On the left, click the + sign of **Project** to expand, then choose `mux2`, and click **OK**. Place the block in the appropriate position in the datapath. You can edit the block interface by right clicking on the symbol, and selecting **Edit Selected Symbol**.

The default name for the instances of symbols are `inst`, `inst1`, `inst2`... The instance names *must be unique*. You can edit the name of the newly created instance of symbol like this. Click on the text "inst" in the lower block on the

left and notice the small handles that indicate that the text object is selected. Click again and notice that the text is now highlighted and can be directly overwritten. Change the default names of the blocks to Control, Memory, and Datapath as appropriate.




J.2 Save the block diagram

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

Select **File** → **Save** to save the block diagram. Notice that the * character has been cleared in the block diagram header.

J.3 Add ports and signals

A signal is a single wire connecting blocks and is drawn as a thin line. A bus is a group of wires connecting blocks, and is drawn as a thicker line than a signal. Ports are the interfaces between blocks and signals or buses. Signal/bus names can be changed by double-clicking the existing name.

You can use the **Pin Tool** or  to add input and output ports to the block diagram. You can use the **Node Tool** or  and **Bus Tool** or  to add and connect ports.

Note: In SystemVerilog, the bus name is in the format of address [15:0]. However, the bus name is in the format of address [15..0] in the .bdf block diagram. Make sure you don't mix these two formats.

J.4 Add finite state machine

Copy the code in Section 4.1.3, and save it as *control.sv*. Then create a symbol for the *control.sv*. Afterwards, add the symbol to *mp0.bdf*. Note that you will have to declare input and output ports to match the block diagram in the Figure 18. By default, it will result in empty symbol.

J.5 Complete datapath and finite state machine

Now, you can modify the *control.sv*, and *datapath.bdf* to complete the design. A complete design will look like Figures 18 and 19. Note that there are two different ways to connect two ports:

1. Directly connect the ports with the wires/bus (e.g. *load_pc*, *load_ir* signals)
2. Make an open-end wire connection on both ports and name the wires identical (e.g. *nzp_match*, *opcode[3..0]* signals). The open-end wire will have 'x' at the end of the connection.

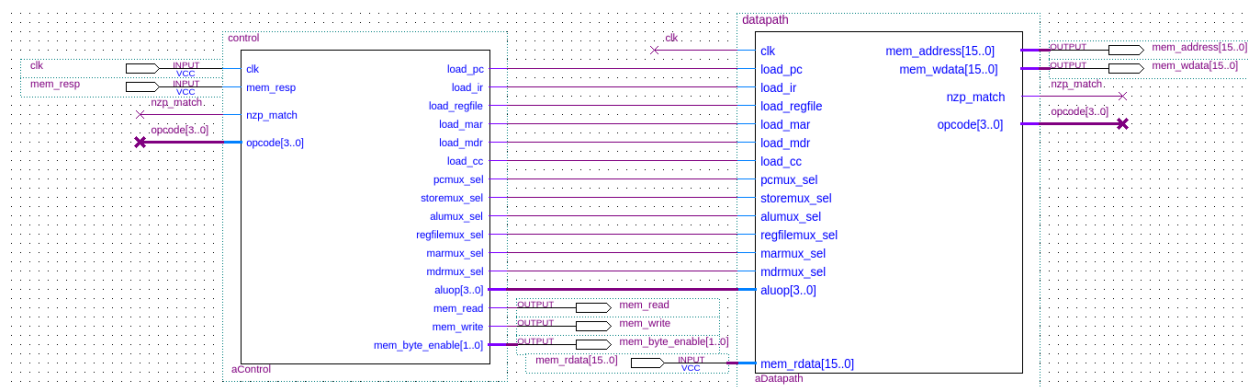


Figure 18: Completed mp0 block diagram *mp0.bdf*.

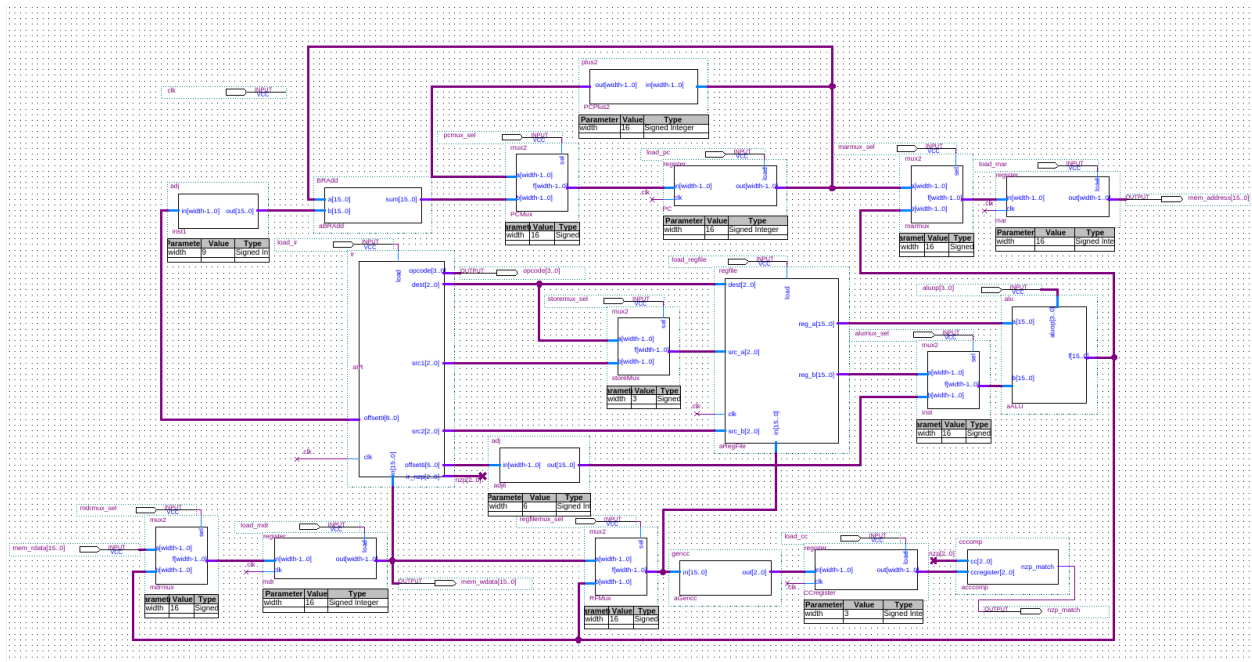


Figure 19: Completed datapath block diagram *datapath.bdf*.

If you modify the source code for the symbol without updating the interface, i.e., the input and output ports, then you don't need to update the symbol. If you have updated the interface, you will need to update symbol. To update the symbol, you can **right click** on the symbol, and select **update symbol or block**. You will have three choices:

1. Selected symbol(s) or block(s)
2. All occurrences of selected symbol(s) or block(s)
3. All symbols or blocks in the file.

Choose the appropriate one to meet your need. After the update, the symbol will look different, and some of the previous connected signals will be disconnected or moved. You will need to manually (re)connect them. This repetitive procedure is one of the reasons that we do not recommend the block diagram design. Since updating symbol interface happens very frequently, we need to do the work again and again.

J.6 Generate HDL code from the block diagram

To run the simulation, you will need to generate the verilog code (Quartus doesn't support SystemVerilog code generation) from the block diagram. To generate the verilog code, in *mp0.bdf*, select **File** → **Create/Update** → **Create HDL Design File from Current File**. Then choose **Verilog HDL**, and click **OK**. Do the same thing for *datapath.bdf* to generate *datapath.v*.

Afterward, you will need to remove *mp0.bdf* and *datapath.bdf* from the project, and add *mp0.v* and *datapath.v* to the project. To do this, in the **Project Navigator**, select the **File** tab, **right click** on **Files**, choose **Add/Remove Files in Project**, and then add and remove the appropriate files.