

ECE 411: Computer Organization and Design

MP 0: The RV32I Processor / Altera Quartus Tutorial

Version 0.0.0

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 RV32Iα instruction set architecture	6
2.1	Overview	6
2.2	Memory instructions	6
2.3	Arithmetic instructions	7
2.4	Control instruction	7
2.5	U-type Instructions	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	25
A	Loading programs into your design	26
B	RTL	27
B.1	FETCH process	27
B.2	DECODE process	27
B.3	SLTI instruction	27
B.4	SLTIU instruction	27
B.5	SRAI instruction	27

B.6	other immediate instructions	28
B.7	BR instruction	28
B.8	LW instruction	28
B.9	SW instruction	28
B.10	AUIPC	29
B.11	LUI	29
C	CPU	30
D	Control	31
D.1	Signals and defaults	31
D.2	Control diagram	31
E	Datapath	34
E.1	Signals	34
E.2	Datapath diagram	35
F	Components	36
F.1	Ports for mux2	36
F.2	Parameters for mux2	36
F.3	SystemVerilog module for mux2	36
G	Block diagram based design	37
G.1	Add and name new blocks	37
G.2	Save the block diagram	38
G.3	Add ports and signals	38
G.4	Add finite state machine	38
G.5	Complete datapath and finite state machine	38
G.6	Generate HDL code from the block diagram	39

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 RV32I instruction set architecture (ISA). We will refer to this subset of the RV32I ISA as the RV32I 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 *19 instructions* in the RV32I 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-a2 ~]\$).

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

2 The RV32I α instruction set architecture

2.1 Overview

For this project, you will be entering the SystemVerilog design of a non-pipelined implementation of the RV32I α instruction set architecture. The RV32I α ISA consists of 19 instructions selected from the full RV32I ISA specifically for this MP. Because RV32I is a relatively simple ISA, it is a natural choice for our ECE 411 projects. The RISC-V specification was created to be a free and open alternative to other popular ISAs and includes a 64 bit variant (and plans for 128 bit) and many extensions for atomic operations, floating point arithmetic, compressed instructions, etc.

All 19 instructions are 32 bits in length, having a format where bits [6:0] contain the opcode. The RV32I 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 32 bits in length, and there are 31 GPRs total, as well as the register x0 which is hardwired as constant 0.

The memory space of the RV32I α consists of 2^{32} locations (meaning the RV32I α has a 32-bit address space) and each location contains 8 bits (meaning that the RV32I α has byte addressability). Due to the limitations of Modelsim, we will only be able to utilize a fraction of this 4GB memory space.

The RV32I α program control is maintained by the Program Counter (PC). The PC is a 32-bit register that contains the address of the current instruction being executed.

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12:10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

Figure 1: RV32I instruction formats

2.2 Memory instructions

Data movement instructions are used to transfer values between the register file and the memory system. The load instruction (LW) reads a 32-bit value from the memory system and places it into a general-purpose register. The store instruction (SW) takes a value from a general-purpose register and writes it into the memory system.

The format of the load instruction, or LW, is shown below. The opcode of the LW instruction bits [6:0] is 0000011. The effective address (the address of the memory location that is to be read) is specified by the rs1 and imm[11:0] fields. The effective address is calculated by adding the contents of the rs1 to the sign-extended imm[11:0] field.

imm[11:0]	rs1	010	rd	0000011	LW
-----------	-----	-----	----	---------	----

The format of the store instruction, SW, is shown below. The opcode of this instruction is 0100011. As with the load instruction (LW), the effective address is the memory location specified by the rs1 and imm[11:0]. The effective address is formed in the same manner as that of the LW except that offset bits imm[4:0] come from the rd part of the instruction instead of the rs2 portion. This is to ensure that the signals for selecting which register index to read from or write to are not dependent on what the instruction opcode is.

imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
-----------	-----	-----	-----	----------	---------	----

2.3 Arithmetic instructions

RV32Ia has nine register-immediate integer instructions: ADDI, SLLI, SLTI, SLTIU, XORI, SRLI, SRAI, ORI, and ANDI. These instructions represent addition, logical left shift, set less than (signed) comparison, set less than unsigned comparison, bitwise exclusive disjunction, logical right shift, arithmetic right shift, bitwise disjunction, and bitwise conjunction, respectively. The encoding format for these instructions is shown below. Note that SRLI and SRAI share the same funct3 code, so you must look at the funct7 portion of the instruction to determine which is which. SLTI and SLTIU will write a value of 1 or 0 to rd depending on if the comparison is true or false, respectively. Each instruction operates on rs1 and the I-type immediate. For comparison and shift, rs1 represents the left side of the operator and the immediate represents the right side of the operator (the shift amount).

imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

2.4 Control instruction

The RV32Ia branch instructions, BEQ, BNE, BLT, BGE, BLTU, BGEU, cause program control to branch to a specified address if the relationship between the first and second operand is equal, not equal, less (signed), greater-or-equal (signed), less (unsigned), or greater-or-equal (unsigned), respectively. When the branch is taken, the address of the next instruction to be executed is calculated by adding the current PC value to the B-type immediate.

imm[12:10:5]	rs2	rs1	000	imm[4:1:11]	1100011	BEQ
imm[12:10:5]	rs2	rs1	001	imm[4:1:11]	1100011	BNE
imm[12:10:5]	rs2	rs1	100	imm[4:1:11]	1100011	BLT
imm[12:10:5]	rs2	rs1	101	imm[4:1:11]	1100011	BGE
imm[12:10:5]	rs2	rs1	110	imm[4:1:11]	1100011	BLTU
imm[12:10:5]	rs2	rs1	111	imm[4:1:11]	1100011	BGEU

2.5 U-type Instructions

The load upper immediate instruction, LUI, puts a 20 bit immediate into the most significant bits of the destination register, leaving the rest as zeros. Combined with ADDI, you can place any arbitrary 32 bit value into a RISCv register. The add upper immediate PC instruction, AUIPC, adds a 20 bit immediate (also padded with 12 zeros in the least significant bits) to the PC and saves that value in the destination register.

imm[31:12]		rd	0110111	LUI
imm[31:12]		rd	0010111	AUIPC

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[31:0]`

Memory is accessed using this 32-bit signal

`mem_rdata[31:0]`

32-bit data bus for receiving data from memory

`mem_wdata[31:0]`

32-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[3:0]`

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 RV32I 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 RV32Ia 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** → **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** → **System Tools** → **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/riscv_mp0/ece411_given.tar.gz \
$ | tar -xzv
```

Two directories will be extracted: *mp0/*, *testcode/*.

- *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)
 - *rv32i_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

Separately, */class/ece411/software* in the EWS filesystem contains software that will be used throughout the semester

- *scripts/rv_load_memory.sh*: script to generate *memory.lst* file from *.asm* test code for use in testbench memory.
- *riscv-tools/bin/riscv32-unknown-elf-**: a GCC-style toolchain for RV32. *as* is the assembler, *ld* is the linker, *gcc* will assemble and link and of course it can even compile C code.
- *scripts/rreference.sh*: script to aid in renaming projects by renaming files and replacing textual references.
- *README*: details of the executables are found here

To begin work on the MP, set up your environment and open Quartus (version 13.1).

```
$ ECE411_SOFTWARE=/class/ece411/software
$ export PATH=$PATH:$ECE411_SOFTWARE/riscv-tools/bin:$ECE411_SOFTWARE/bin
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ECE411_SOFTWARE/lib64:
  $ECE411_SOFTWARE/riscv-tools/lib
```

```
$ export PYTHONPATH=$PYTHONPATH:$ECE411_SOFTWARE/python2.7/site-packages
$ module load altera/13.1
$ 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 run these commands every time, add the lines to your `~/.bashrc` file. You will need to logout and login again for changes to take effect.

4.1 Beginning the design

Some components for the RV32I 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 E.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 F. 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 = 32)
(
    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 32 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 a 4 input mux which will become the input to the register file.

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 `cmpmux`.

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

Listing 4: Internal signals for `cmpmux` in *datapath.sv*

```
| rv32i_word rs2_out, i_imm, cmpmux_out;
```

Note that `rv32i_word` is defined in *rv32i_types.sv*. `cmpmux_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 cmpmux_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 cmpmux
| (
|     .sel(cmpmux_sel),
|     .a(rs2_out),
|     .b(pc_out),
|     .f(cmpmux_out)
| );
```

Here, we instantiated a two-input mux called `cmpmux` and connected the mux select signal to `cmpmux_sel`, the inputs to `rs1_out` and `pc_out`, and the output to `cmpmux_out`. The default mux width of 32 bits is exactly what we need but see the listing below for an example of instantiating a module with non-default parameters.

Listing 7: `cmpmux` instantiation with correct width parameter

```
| mux2 #(.width(32)) cmpmux
| (
|     .sel(cmpmux_sel),
|     .a(rs2_out),
```

```

        .b(pc_out),
        .f(cmpmux_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 #(32) cmpmux
(
    cmpmux_sel,
    rs2_out,
    pc_out,
    cmpmux_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 E](#) 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 rv32i_types::*; /* Import types defined in rv32i_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 AUIPC instruction. The first line simply imports the types that are defined in *rv32i_types.sv*. Next, the input and output ports need to be specified.

Listing 10: Controller port declaration

```

module control
(
    input clk,

    /* Datapath controls */
    input rv32i_opcode opcode,
    output logic load_pc,
    output logic load_ir,
    output logic load_regfile,
    output logic alumux1_sel,
    output logic [1:0] alumux2_sel,
    output alu_ops aluop,
    /* et cetera */

    /* Memory signals */
    input mem_resp,
    output logic mem_read,
    output logic mem_write,
    output rv32i_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 instruction¹ (see Appendices B and D).

Listing 11: Additional states for AUIPC instruction.

```

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

```

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

Listing 12: Additional state actions for AUIPC.

```

always_comb
begin : state_actions
    /* Default assignments */
    load_pc = 1'b0;
    load_ir = 1'b0;
    load_regfile = 1'b0;
    alumux1_sel = 1'b0;
    alumux2_sel = 2'b00;

```

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

```

        //in many cases, aluop will be the same as funct3, so just typecast
        it
        aluop = alu_ops'(funct3);
        mem_read = 1'b0;
        mem_write = 1'b0;
        mem_byte_enable = 4'b1111;
        pcmux_sel = 0;
        /* et cetera (see Appendix E) */

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

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

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

        decode: /* Do nothing */;

        s_auiopc: begin
            /* DR <= PC + u_imm */
            load_regfile = 1;

            //PC is the first input to the ALU
            alumux1_sel = 1;

            //the u-type immediate is the second input to the ALU
            alumux2_sel = 1;

            //in the case of auiopc, funct3 is some random bits so we
            //must explicitly set the aluop
            aluop = alu_add;

            /* PC <= PC + 4 */
            load_pc = 1;
        end

        default: /* Do nothing */;

    endcase
end

```

Inside the always block, the default assignments are made to the control signals. Then, the control signals are driven based on what the current state is.

After the state actions are in place, we move on to the next state logic.

Listing 13: The next state logic.

```
always_comb
begin : next_state_logic
    next_state = state;
    case(state)
        fetch1: next_state = fetch2;
        fetch2: if (mem_resp) next_state = fetch3;
        fetch3: next_state = decode;

        decode: begin
            case(opcode)
                op_auihc: next_state = s_auihc;
                default: $display("Unknown opcode");
            endcase
        end

        default: next_state = fetch1;
    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 14: 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 AUIHC instruction. Finish the controller for all other instructions by following Appendices [B](#), [C](#), and [D](#), 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 *magic_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 *magic_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.

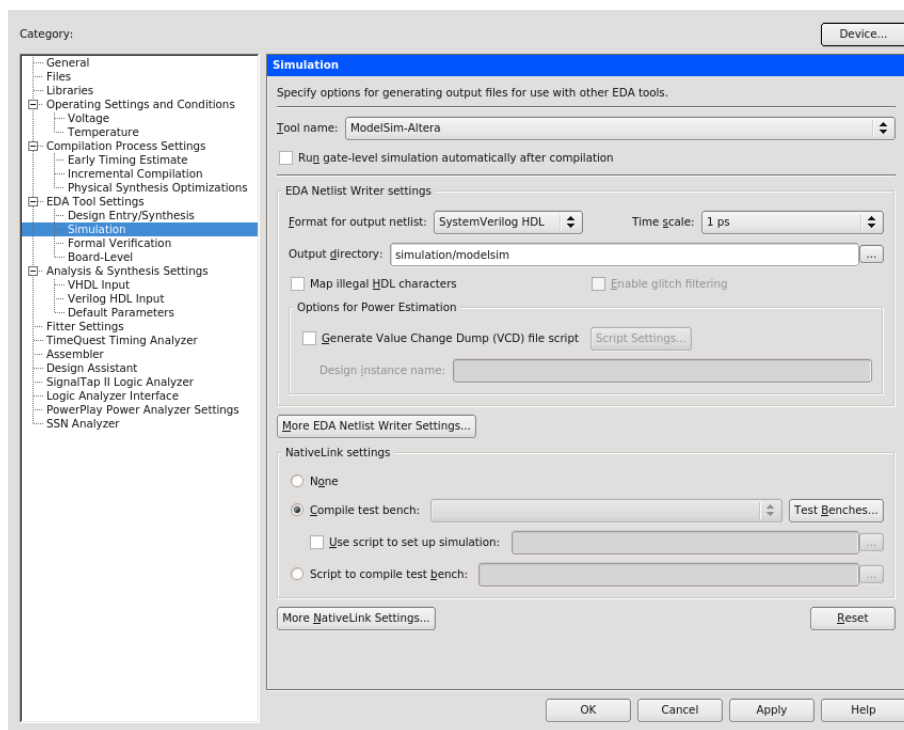


Figure 2: Simulation options

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 *magic_memory.sv*. Click **OK** several times to save the settings.

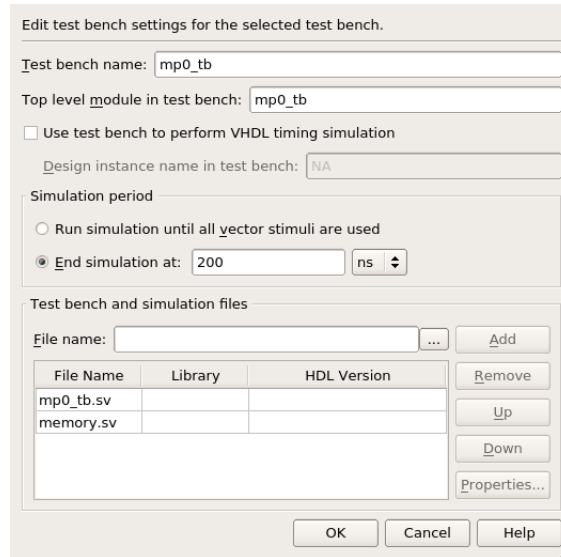


Figure 3: Test bench options

5.1.1 Testbench memory initialization

To test the design, we will load the memory with a RISC-V 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 4: 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

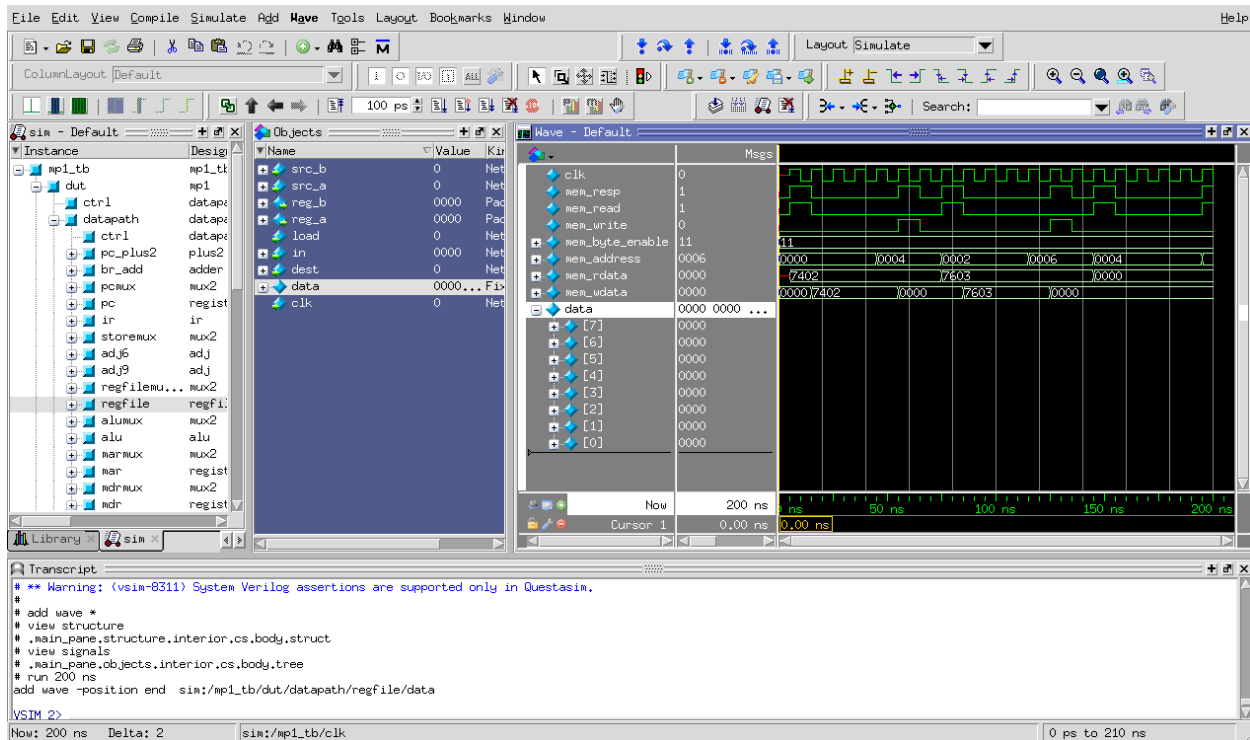


Figure 5: The wave trace window

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 RV32ISimulator 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 RISC-V 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).

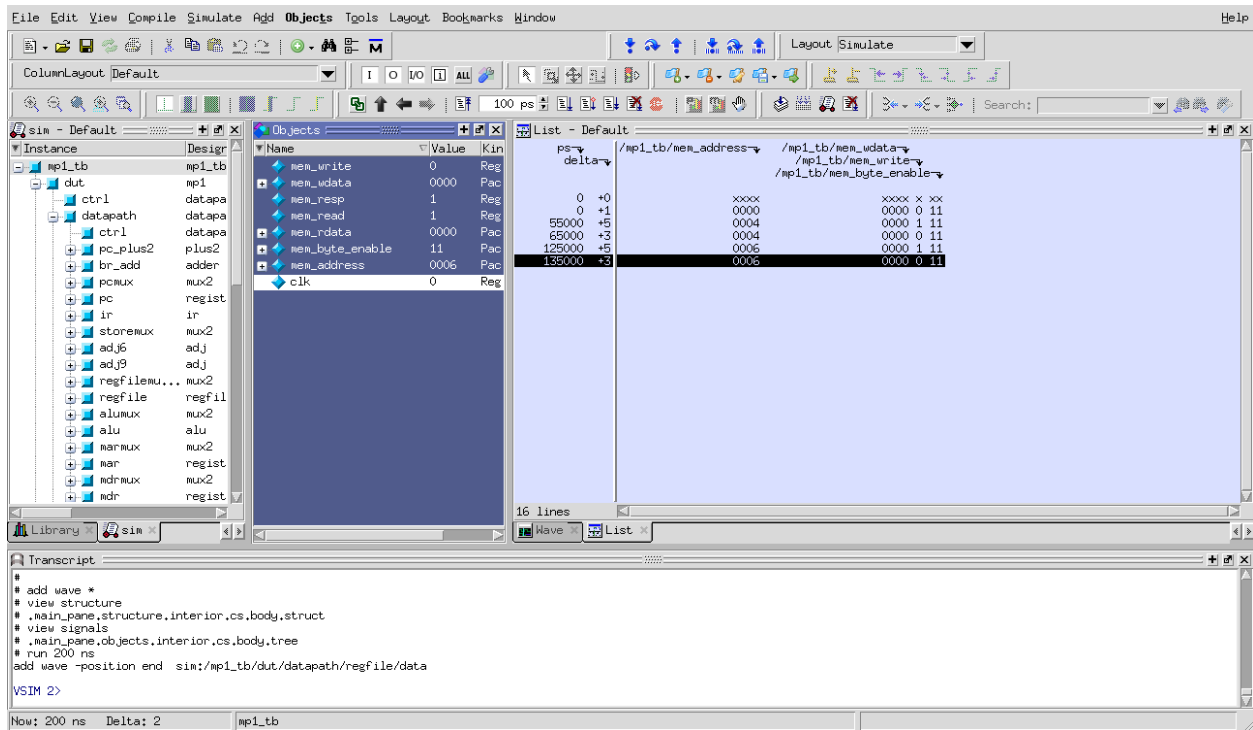


Figure 6: The lists window

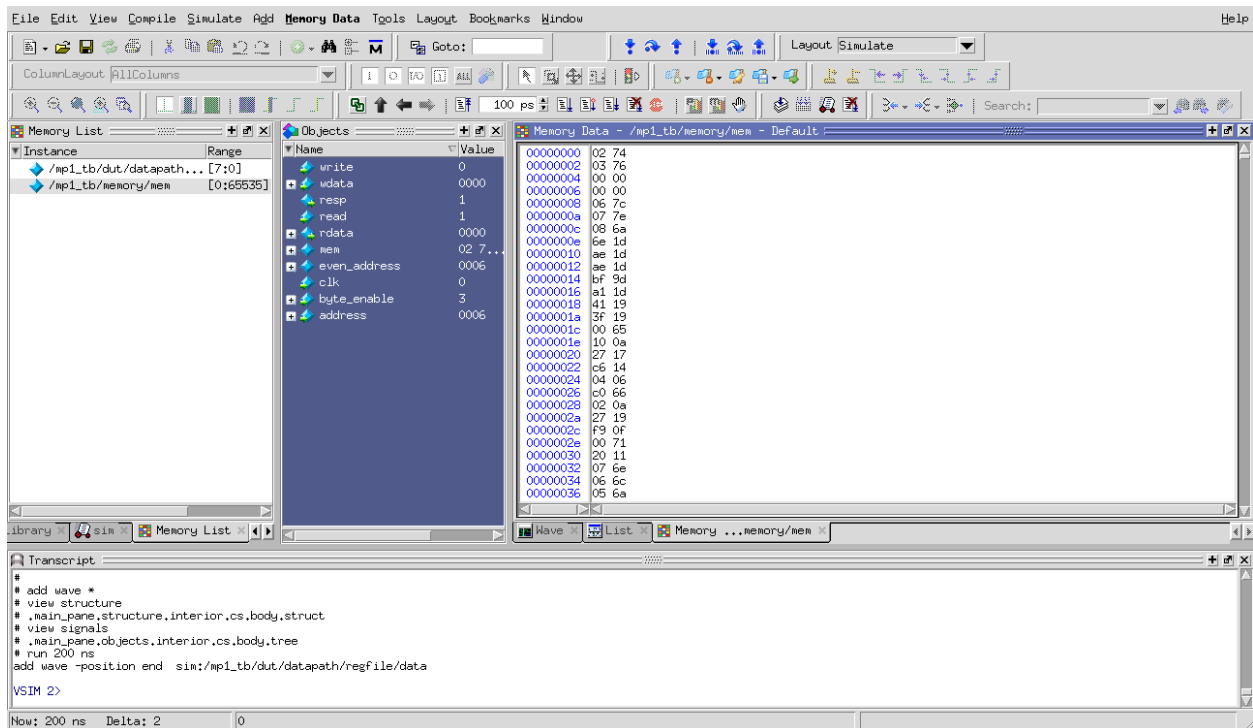


Figure 7: The memory lists window

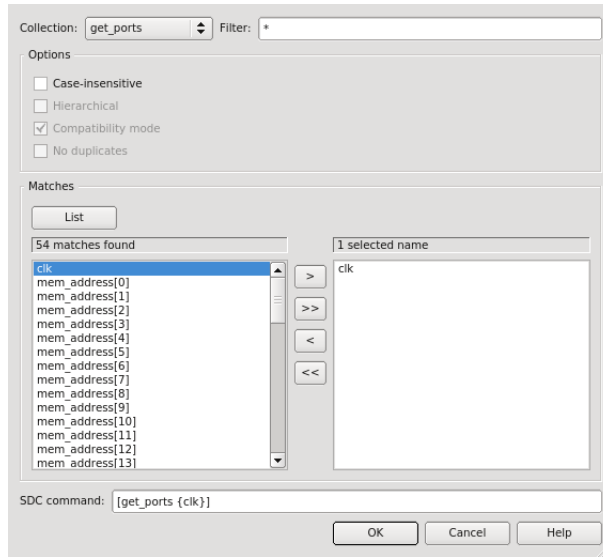


Figure 9: Selecting clock to constrain

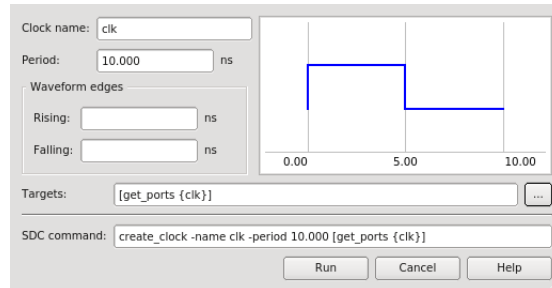


Figure 10: Specifying clock constraints

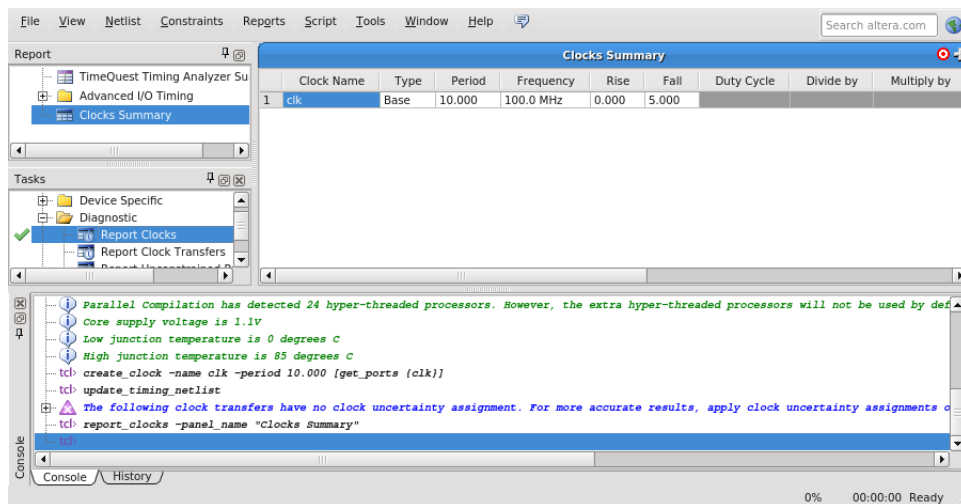


Figure 11: The clock report

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.

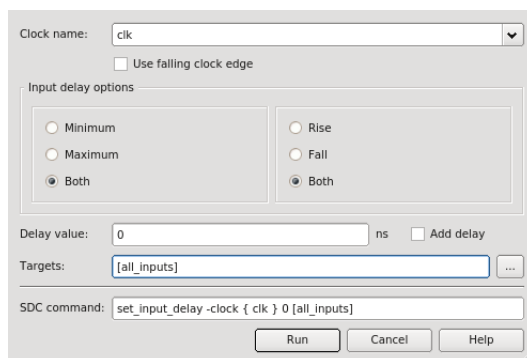


Figure 12: Specifying input constraints

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]**.

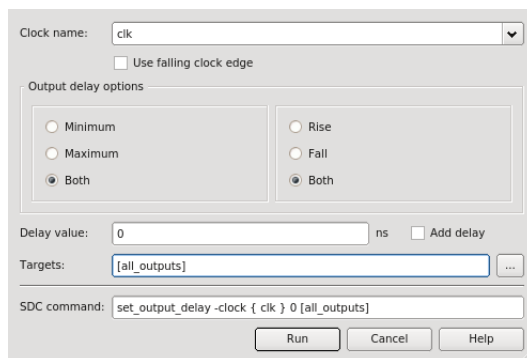


Figure 13: Specifying output constraints

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.

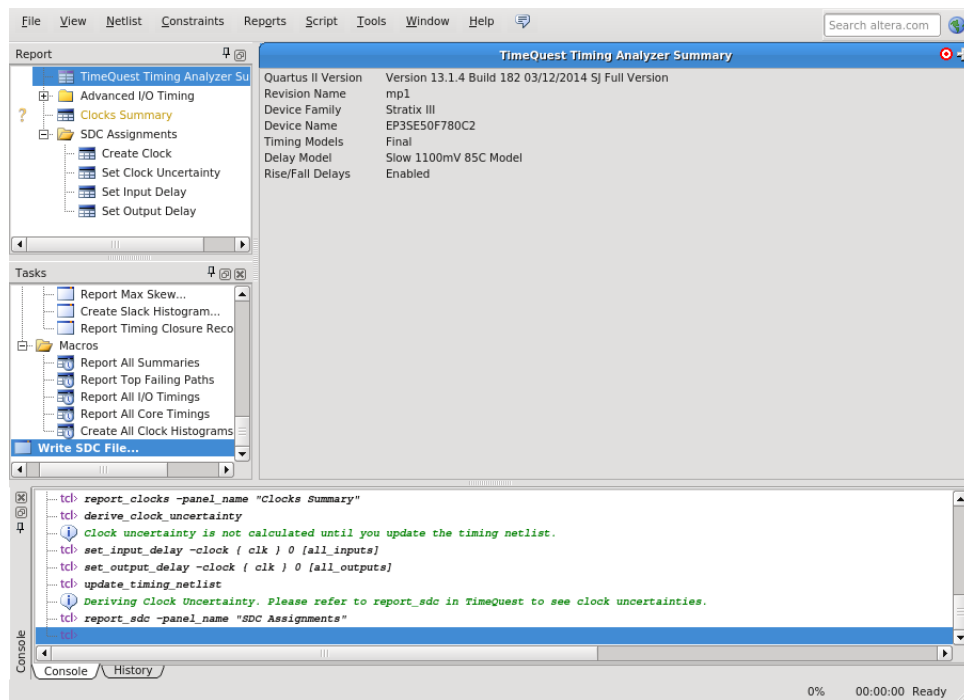


Figure 14: Writing the SDC file

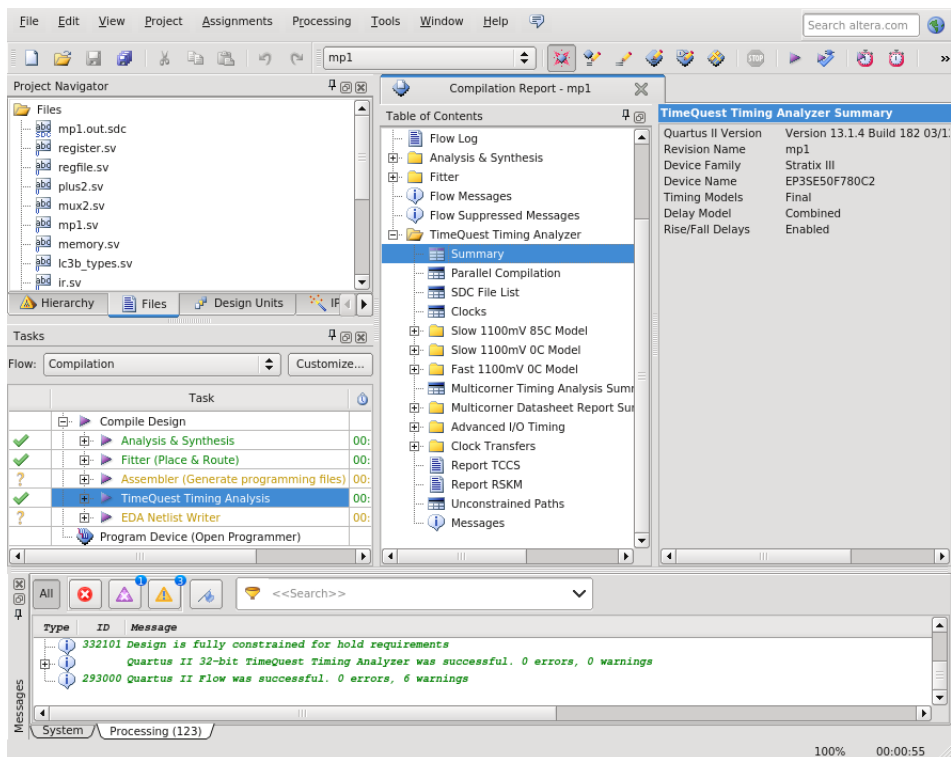


Figure 15: The timing analysis summary

7 Final hand-in

Write a short program using the RISC-V assembly language to calculate 5! (five factorial). Your program *must be iterative*. You must use load instructions to initialize registers. Reference the sample program located at `test-code/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 must include comments in this assembly program and upload it in PDF format to Compass before the deadline. Comments should be more than an explanation of what a particular instruction does.

All SystemVerilog files in your design that were not provided to you must be committed and *and pushed* to Gitlab before the deadline. Any provided code that you were asked to modify (datapath and control modules) must also be committed and pushed to Gitlab before the deadline. The autograder will use the original versions of all provided files that you were not asked to modify (ALU, IR, etc.) so you will lose points if your design relies on changes you made to those files. You must also commit and push your `mp0.out.sdc` file before the deadline as you will be graded on its correctness. All these files must be contained within a directory named `mp0` which must be contained in the root directory of your assigned ECE411 Gitlab repository. These constraints are extremely important so please don't hesitate to ask a TA if you need help understanding distributed version control.

The autograder will test your design in two ways. First it will run many small tests that each target a very minimal amount of functionality but together they should cover nearly all functionality. This is the best way for the autograder to give you as much partial credit as possible for small bugs. The second method of testing will be a larger test code that will test that your design can successfully run larger sequences of instructions. No partial credit will be given for this larger test code but it will not test corner cases as thoroughly as the targeted tests.

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
Targeted Tests	7	35
Larger Testcode	4	20
Timing report	3	5
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 *rv_load_memory.sh* script located in the */class/ece411/software/scripts* directory can be used to do this.

The *rv_load_memory.sh* script takes a RISCv assembly file as input, assembles it into a RISCv 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=/class/ece411/software/riscv-tools/bin/riscv32-unknown-elf-
gcc
OBJCOPY=/class/ece411/software/riscv-tools/bin/riscv32-unknown-elf-
objcopy
OBJDUMP=/class/ece411/software/riscv-tools/bin/riscv32-unknown-elf-
objdump
ADDRESSABILITY=1
```

To execute *rv_load_memory.sh*, you need to supply the name of a RISCv assembly file and, optionally, the location to write *memory.lst*.

```
$ ./rv_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 *rv_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/
$ ./rv_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 RTL

B.1 FETCH process

State	Data	Control
fetch1	$MAR \leftarrow PC$	$load_mar \leftarrow 1$;
fetch2	while ($mem_resp == 0$) $MDR \leftarrow M[MAR]$;	$load_mdr \leftarrow 1$; $mem_read \leftarrow 1$;
fetch3	$IR \leftarrow MDR$;	$load_ir \leftarrow 1$;

B.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)

B.3 SLTI instruction

State	Data	Control
FETCH		
DECODE		
s_imm	$rd \leftarrow rs1 \oplus i_imm$; $PC \leftarrow PC + 4$;	$load_regfile \leftarrow 1$; $load_pc \leftarrow 1$; $cmpop \leftarrow blt$; $regfilemux_sel \leftarrow 1$; $cmpmux_sel \leftarrow 1$

B.4 SLTIU instruction

State	Data	Control
FETCH		
DECODE		
s_imm	$rd \leftarrow rs1 \oplus i_imm$; $PC \leftarrow PC + 4$;	$load_regfile \leftarrow 1$; $load_pc \leftarrow 1$; $cmpop \leftarrow bltu$; $regfilemux_sel \leftarrow 1$; $cmpmux_sel \leftarrow 1$

B.5 SRAI instruction

State	Data	Control
FETCH		
DECODE		
s_imm	$rd \leftarrow rs1 \oplus i_imm$; $PC \leftarrow PC + 4$;	$load_regfile \leftarrow 1$; $load_pc \leftarrow 1$; $alu_op \leftarrow alu_sra$;

B.6 other immediate instructions

State	Data	Control
FETCH		
DECODE		
s_imm	$rd \leftarrow rs1 \oplus i_imm$; $PC \leftarrow PC + 4$;	$load_regfile \leftarrow 1$; $load_pc \leftarrow 1$; $alu_op \leftarrow funct3$;

B.7 BR instruction

State	Data	Control
FETCH		
DECODE		
br	$PC \leftarrow PC + (br_en ? b_imm : 4)$;	$pcmux_sel \leftarrow br_en$; $load_pc \leftarrow 1$; $alumus1_sel \leftarrow 1$; $alumus2_sel \leftarrow 2$; $alu_op \leftarrow alu_add$

B.8 LW instruction

State	Data	Control
FETCH		
DECODE		
calc_addr	$MAR \leftarrow rs1 + i_imm$;	$aluop \leftarrow alu_add$; $load_mar \leftarrow 1$; $marmux_sel \leftarrow 1$;
ldr1	while ($mem_resp == 0$) $MDR \leftarrow M[MAR]$;	$load_mdr \leftarrow 1$; $mem_read \leftarrow 1$;
ldr2	$rd \leftarrow MDR$; $PC \leftarrow PC + 4$;	$regfilemux_sel \leftarrow 3$; $load_regfile \leftarrow 1$; $load_pc \leftarrow 1$;

B.9 SW instruction

State	Data	Control
FETCH		
DECODE		
calc_addr	$MAR \leftarrow rs1 + s_imm$; $data_out \leftarrow rs2$	$alumus2_sel \leftarrow 3$; $aluop \leftarrow alu_add$; $load_mar \leftarrow 1$; $load_data_out \leftarrow 1$; $marmux_sel \leftarrow 1$;
str1	while ($mem_resp == 0$) $M[MAR] \leftarrow data_out$;	$mem_write \leftarrow 1$;
str2	$PC \leftarrow PC + 4$;	$load_pc \leftarrow 1$;

B.10 AUIPC

State	Data	Control
FETCH		
DECODE		
s_auipc	$rd \leftarrow pc + u_imm; PC \leftarrow PC + 4;$	$alumux1_sel \leftarrow 1; alumux2_sel \leftarrow 1;$ $load_regfile \leftarrow 1; load_pc \leftarrow 1;$ $alu_op \leftarrow alu_add;$

B.11 LUI

State	Data	Control
FETCH		
DECODE		
s_lui	$rd \leftarrow u_imm; PC \leftarrow PC + 4;$	$load_regfile \leftarrow 1; load_pc \leftarrow 1;$ $regfilemux_sel \leftarrow 2;$

C CPU

(a) Control to datapath		(b) Datapath to control	
Name	Type	Name	Type
load_pc	logic	opcode	rv32i_opcode
load_ir	logic	funct3	logic [2:0]
load_regfile	logic	funct7	logic [6:0]
load_mar	logic	br_en	logic
load_mdr	logic		
load_data_out	logic		
pcmux_sel	logic		
cmpop	branch_funct3_t		
alumux1_sel	logic		
alumux2_sel	logic [1:0]		
regfilemux_sel	logic [1:0]		
marmux_sel	logic		
cmpmux_sel	logic		
aluop	alu_ops		
(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 [3:0]		
(e) Datapath to memory		(f) Memory to datapath	
Name	Type	Name	Type
mem_address	rv32i_word	mem_rdata	rv32i_word
mem_wdata	rv32i_word		

Table 1: CPU connections

D Control

D.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_data_out	1'b0
pcmux_sel	1'b0
cmpop	funct3
alumux1_sel	1'b0
alumux2_sel	2'b0
regfilemux_sel	2'b0
marmux_sel	1'b0
cmpmux_sel	1'b0
aluop	funct3
mem_read	1'b0
mem_write	1'b0
mem_byte_enable	4'b1111

D.2 Control diagram

See [Appendix B](#) for control state actions.

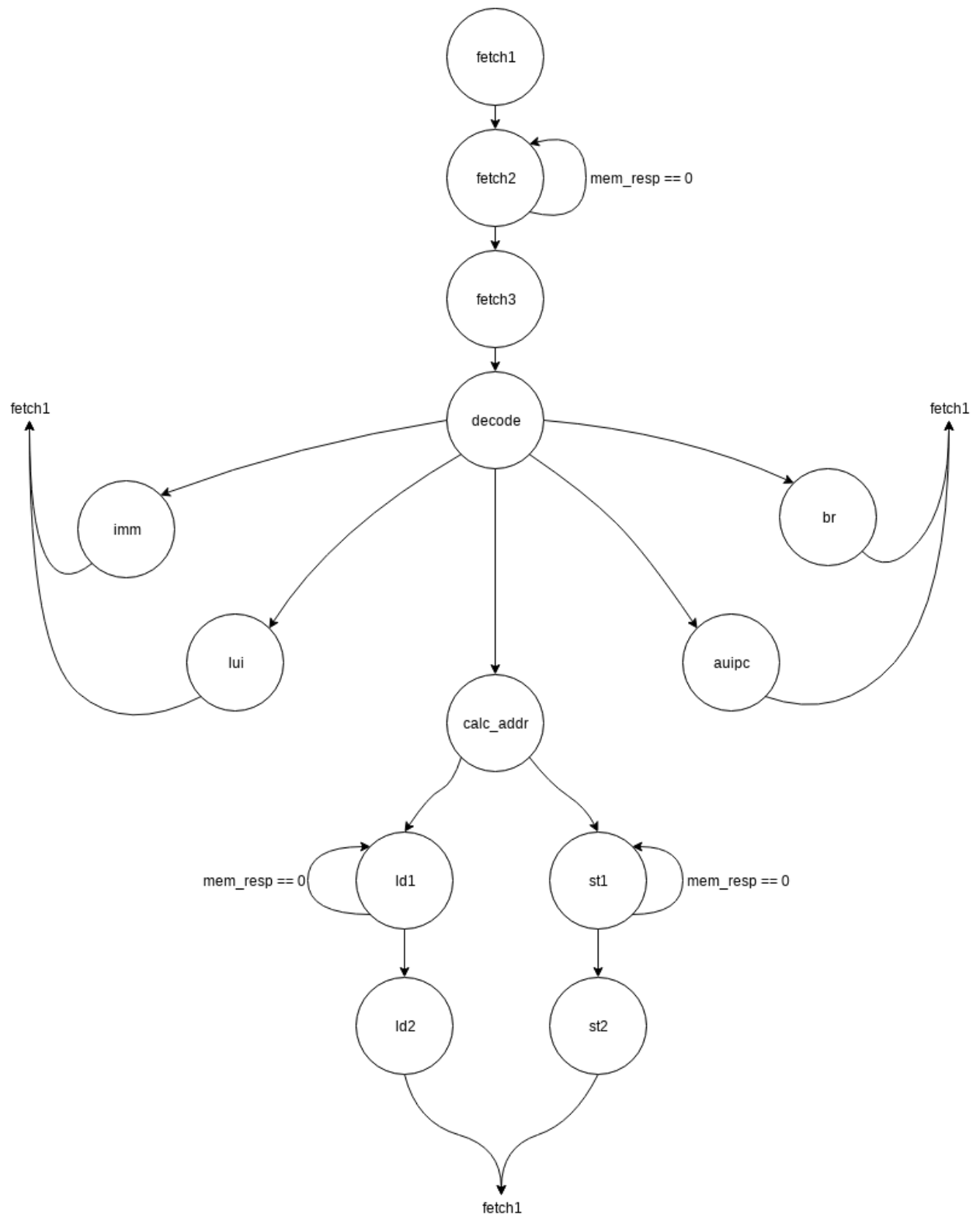


Figure 16: The RV32Iα control state diagram

E Datapath

E.1 Signals

Name	Type	Origin	Destination
clk	logic	input port	PC, IR, regfile, MAR, MDR, mem_data_out
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_data_out	logic	control	mem_data_out
pcmux_sel	logic	control	pcmux
alumux1_sel	logic	control	alumux1
alumux2_sel	logic [1:0]	control	alumux2
regfilemux_sel	logic [1:0]	control	regfilemux
marmux_sel	logic	control	marmux
cmpmux_sel	logic	control	cmpmux
aluop	alu_ops	control	ALU
rs1	rv32i_reg	IR	regfile
rs2	rv32i_reg	IR	regfile
rd	rv32i_reg	IR	regfile
rs1_out	rv32i_word	regfile	alumux1, CMP
rs2_out	rv32i_word	regfile	cmpmux, mem_data_out
i_imm	rv32i_word	IR	alumux2, cmpmux
u_imm	rv32i_word	IR	alumux2, regfilemux
b_imm	rv32i_word	IR	alumux2
s_imm	rv32i_word	IR	alumux2
pcmux_out	rv32i_word	pcmux	PC
alumux1_out	rv32i_word	alumux	ALU
alumux2_out	rv32i_word	alumux	ALU
regfilemux_out	rv32i_word	regfilemux	regfile
marmux_out	rv32i_word	marmux	MAR
cmpmux_out	rv32i_word	cmpmux	CMP
alu_out	rv32i_word	ALU	regfilemux, marmux, pcmux
pc_out	rv32i_word	PC	pc_plus4, alumux1, marmux
pc_plus4_out	rv32i_word	pc_plus4	pc_mux
mdrreg_out	rv32i_word	MDR	regfilemux, IR
mem_address	rv32i_word	MAR	output port
mem_wdata	rv32i_word	mem_data_out	output port
mem_rdata	rv32i_word	input port	MDR
opcode	rv32i_opcode	IR	control
cmpop	branch_funct3_t	control	CMP
funct3	logic [2:0]	IR	control
funct7	logic [6:0]	IR	control
br_en	logic	cmp 34	control, regfilemux

E.2 Datapath diagram

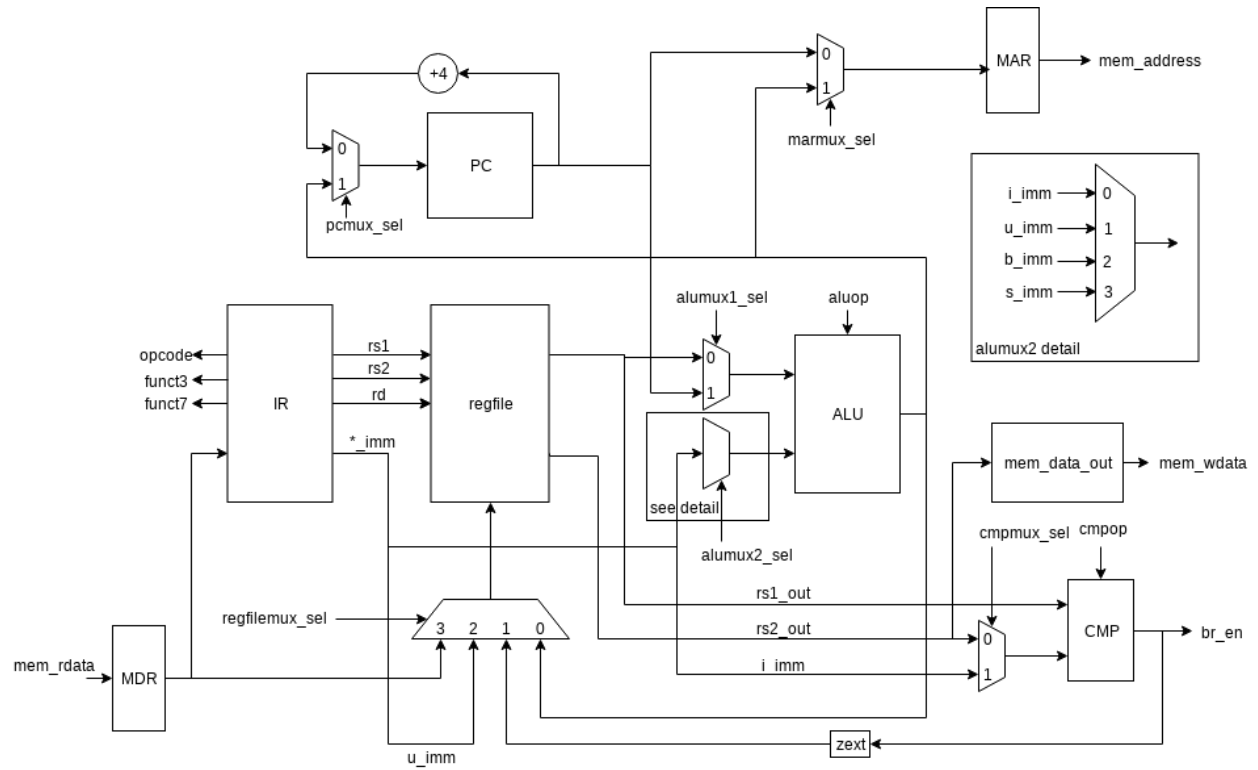


Figure 17: The RV32Iα datapath diagram

F Components

F.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]

F.2 Parameters for mux2

Name	Default value
width	32

F.3 SystemVerilog module for mux2

Listing 15: The mux2 module

```
module mux2 #(parameter width = 32)
(
    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
```

G 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.

G.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 F, 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 18.

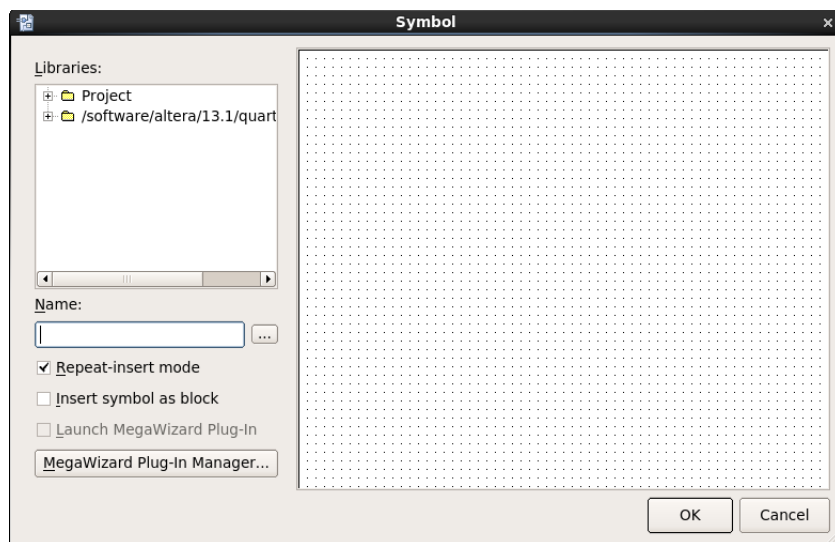


Figure 18: Symbol tool

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.




G.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.

G.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 [31 : 0]. However, the bus name is in the format of address [31 . . 0] in the .bdf block diagram. Make sure you don't mix these two formats.

G.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 19. By default, it will result in empty symbol.

G.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 19 and 20. 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.

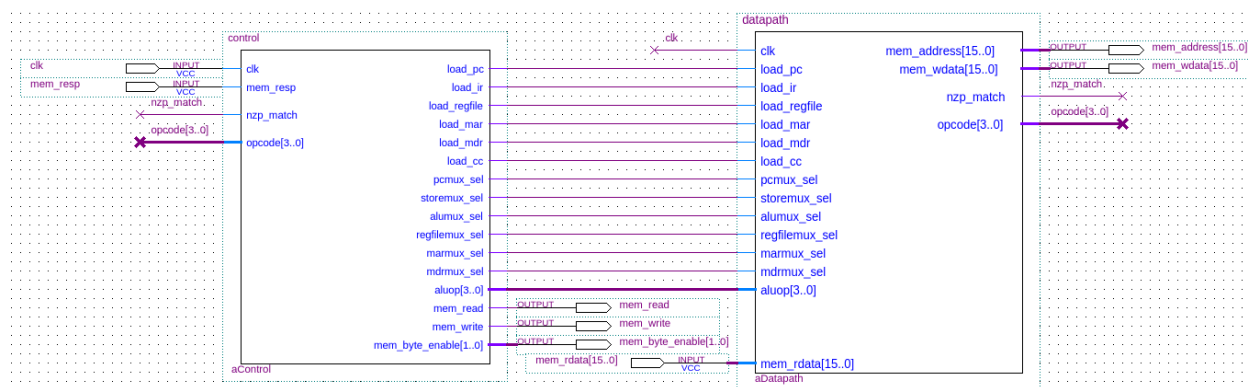


Figure 19: Completed mp0 block diagram *mp0.bdf*.

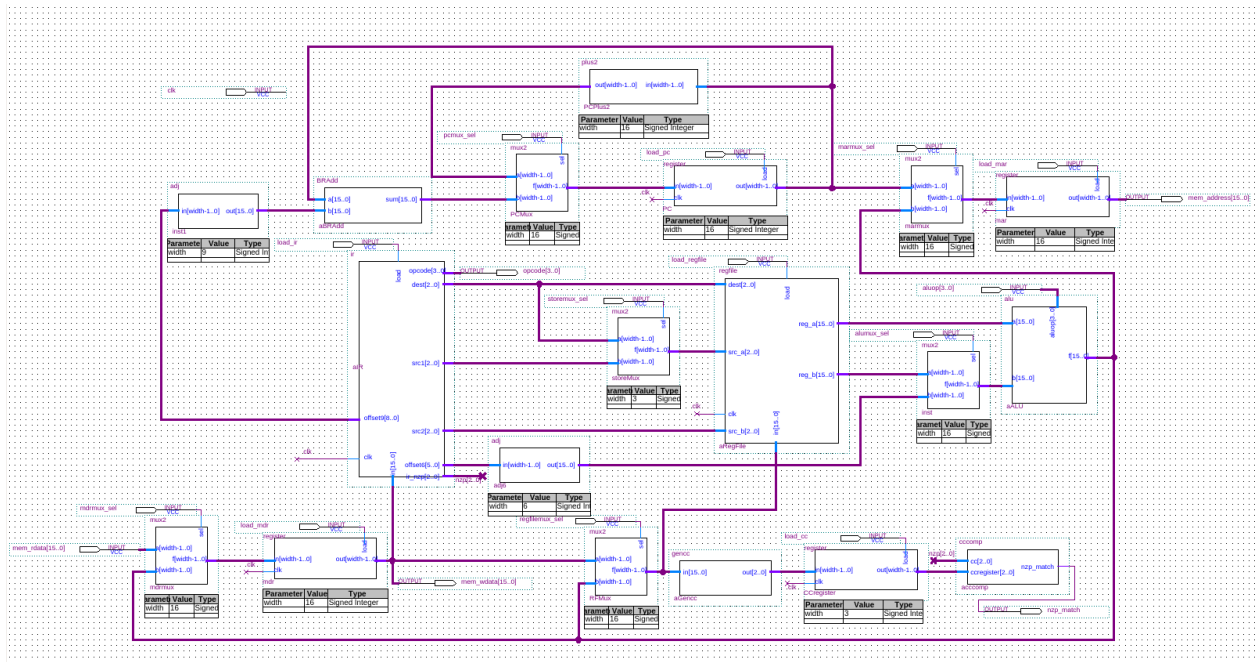


Figure 20: Completed datapath block diagram *datapath.bdf*.

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.

G.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.