ECE 411: Computer Organization and Design

MP 1: The RV32I Processor / Altera Quartus Tutorial

Version 0.0.0

Contents

1 Introduction

Welcome to the first ECE 411 RISC-V 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). 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 verify it using dynamic simulation.

The primary objective of this exercise is to give you a better understanding of multicycle micro-architectures, and of the RISC-V 32i ISA. Additionally, you will learn how to use Altera Quartus software to synthesize your design for an FPGA, and continue using ModelSim to verify your design. 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 brief description of the relevant instructions in the RV32I 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, as well as what is due for Check Point 1, and what is due for the Final handin. Also included are several appendices that contain additional useful information.

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

1.1 Notation

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

- Bit 0 refers to the *least* significant bit.
- Numbers beginning with `0x` are hexadecimal.
- `[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 instruction set architecture

2.1 Overview

For this project, you will be entering the SystemVerilog design of a non-pipelined implementation of the RV32I instruction set architecture. Because RV32I is a relatively simple load-store ISA with a robust toolchain published under GPL, it is a natural choice for our ECE 411 projects. The RISCV 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. For this MP, you will implement all of the RV32I instructions with the exception of the **FENCE***, **ECALL**, **EBREAK**, and **CSRR*** instructions.

Instructions are fixed width and 32 bits in length, having a format where bits `[6:0]` contain the opcode. The RV32I 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 consists of 2^{32} locations (meaning the RV32I has a 32-bit address space) and each location contains 8 bits (meaning that the RV32I 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 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.

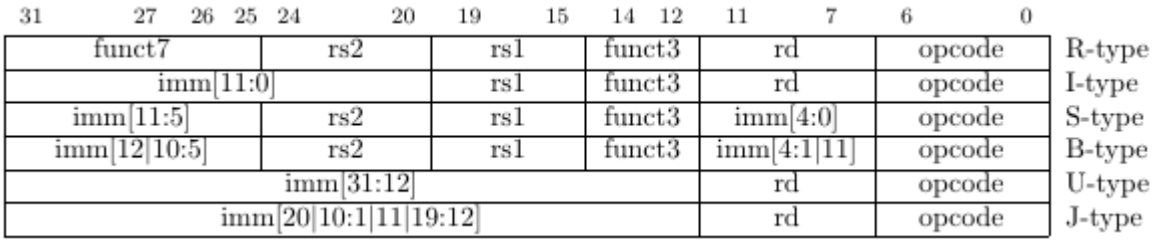


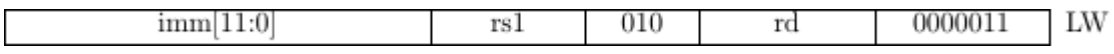
Figure 1: RV32I instruction formats

Detailed descriptions of all RV32-I instructions may be found in the RISC-V Instruction Set Manual. Detailed encodings for each instruction can be found in this table.

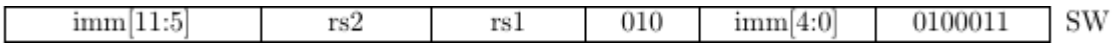
2.2 Memory Instructions

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.



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.



2.3 Arithmetic Instructions

RV32I 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

Additionally, RV32I has several register-register integer computational instructions.

2.4 Control instruction

The RV32I 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

Additionally, RV32I supports two unconditional branching instructions which are used to create call and return type operations, as well as implement function pointers. These are the `JSR` and `JSRR` instructions.

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, etc.

`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, 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 RV32I 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 tutorials, which is available through Quartus itself (click on **Help**). These tutorials may cover additional topics not covered here.

To run Quartus from an EWS Linux machine, run

```
$  module load altera/18.1-std && quartus
```

To work remotely, use the '-X' option over ssh to enable X-forwarding.

To get the provided base code for MP1, from your ece411 MP directory, run

```
$  git fetch release
$  git merge --allow-unrelated-histories release/mp1 -m "Merging MP1"
```

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/rereference.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:

```
$  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/18.1-std && quartus &
```

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, or source your bashrc.

In Quartus, use the "New Project Wizard" to create your MP1 project. Use your git repository's MP1 directory as the project working directory. Name the project 'mp1'. Create an empty project, and add all of the SystemVerilog files from the *mp1/hdl* directory. Under "Family, Device and Board Settings", select the Stratix IV EP4SGX530NF45C4ESas your target device. Under 'EDA Tool Settings", select 'Modelsim-Altera' as your Simulation tool, and SystemVerilog HDL as the Format. Ensure that 'Run gate-level simulation...' is NOT checked.

4.1 Beginning the design

Some components for the RV32I 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. 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 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 1: 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
```

4.1.2 Connect the datapath and controller

The *mp1.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. Finish the controller for all instructions by following the design in Appendices B, C, and D. You will have to figure out the design for several of the instructions, including the register-register integer computational instructions. After adding an instruction, try compiling your design and testing the newly added instruction.

5 Analysis and functional verification

After the design has been entered, you will perform RTL simulation to verify the correctness of the design. We recommend that you test your design after adding each instruction.

The main hvl file to use in simulation is *mp1/hvl/top.sv*. This file does several things:

- it instantiates your MP1 design as the DUT;
- it instantiates one of two testbenches which provide input stiumulus to the DUT;
- it instanties an interface between itself, the testbench, the DUT, and memory, and generates a clock
- it provides several halting conditions for your simulation;
- it instantiates a 'riscv_formal_monitor_rv32i', which monitors the output as well as some of the internal state of the DUT and reports an error when the DUT outputs an incorrect value or enters an incorrect state.

Two different testbenches are provided. To choose which one to instantiate in *mp1/hvl/top.sv*, set the `TESTBENCH` macro to either `SRC` or `RAND`.

The `SRC` testbench drives the DUT by loading a program binary into memory, and executing the program. This testbench should largely remain unchanged, instead modify the tests by modifying the compiled program. We suggest using this testbench to execute simulations which use large amounts of branches and jumps.

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 Quartus project. See Appendix A for instructions on compiling RISC-V programs and loading them into memory.

The `RAND` testbench drives the DUT by executing a sequence of randomly generated instructions. This testbench can and should be modified, as we have only provided the code to test immediate arithmetic instructions. We suggest extending this testbench to support simulation of randomly generated register-register instructions, and load-store instructions.

5.1 Testbench memory initialization

See Appendix A for how to load an assembly program into the design. Use the instructions to load the given test code in *mp1/testcode/riscv_mp1testcode.s*.

5.2 Adding Testbench in Quartus

Under **Assignments →Settings...** add a new testbench with the following settings:

Test bench name: **top**
Top level module in test bench: **top**
Simulation Period: **Run Simulation until all vector stimuli are used**

Under the **Test bench and simulation files** section, add all of the files in the *hvl* directory. Click **OK** several times to save the settings.

5.3 RTL simulation

5.3.1 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/18.1/modelsim_ase/linuxaloem*. Now, upon initiating ModelSim simulation from within Quartus, Quartus will

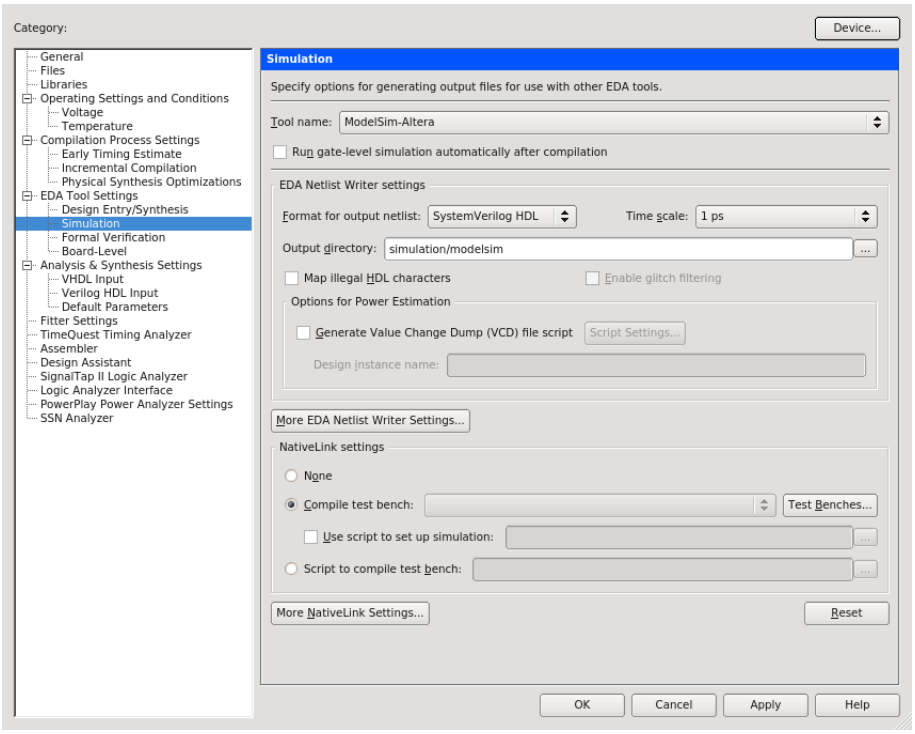


Figure 2: Simulation options

generate a Tcl script in the *simulation/modelsim* directory. Upon launching the ModelSim GUI, this Tcl script is executed.

You can, of course, execute this Tcl script from the ModelSim shell as in MP0. **We recommend that you focus your testbench efforts on creating useful text output from ModelSim, and use the waveform viewer as just another tool for debugging, not as your main verification tool**.

5.3.2 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.

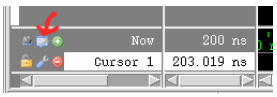


Figure 3: Grid and timeline options

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

5.3.2.1 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., **top →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
```

5.3.2.2 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 view 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 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**.

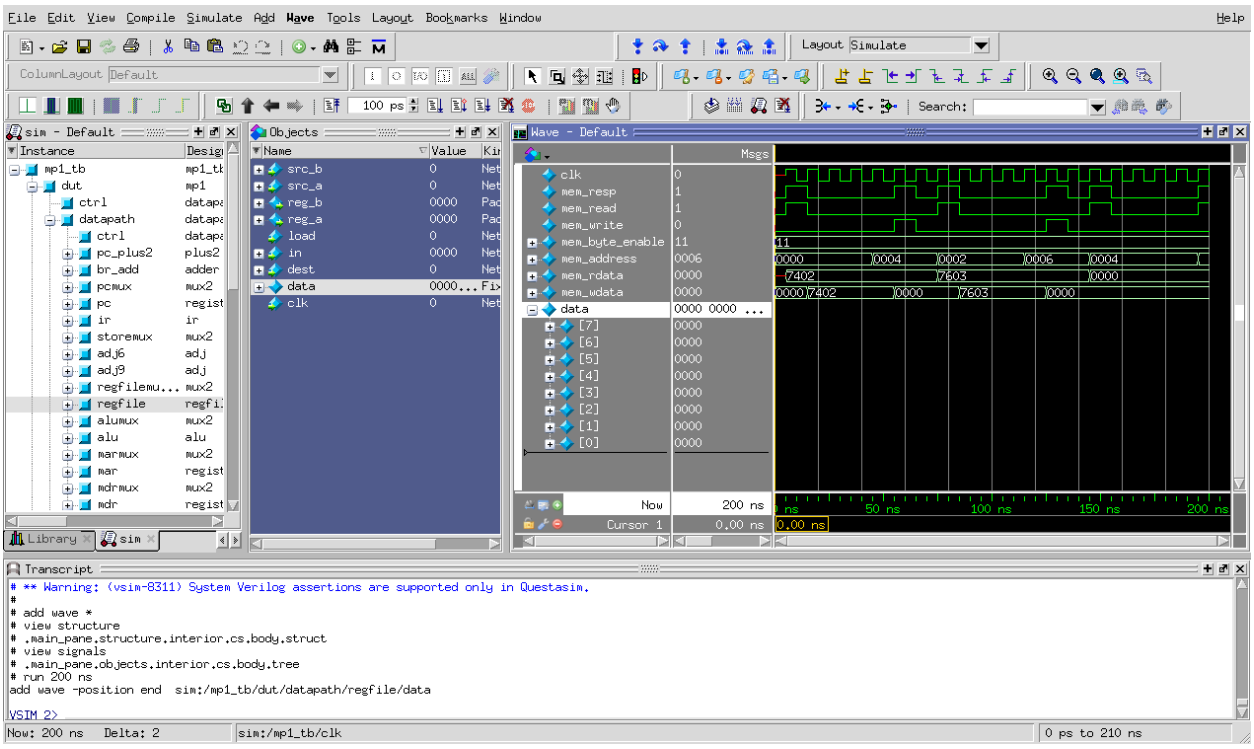


Figure 4: The wave trace window

5.3.2.3 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.

5.3.3 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 RISCV 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).

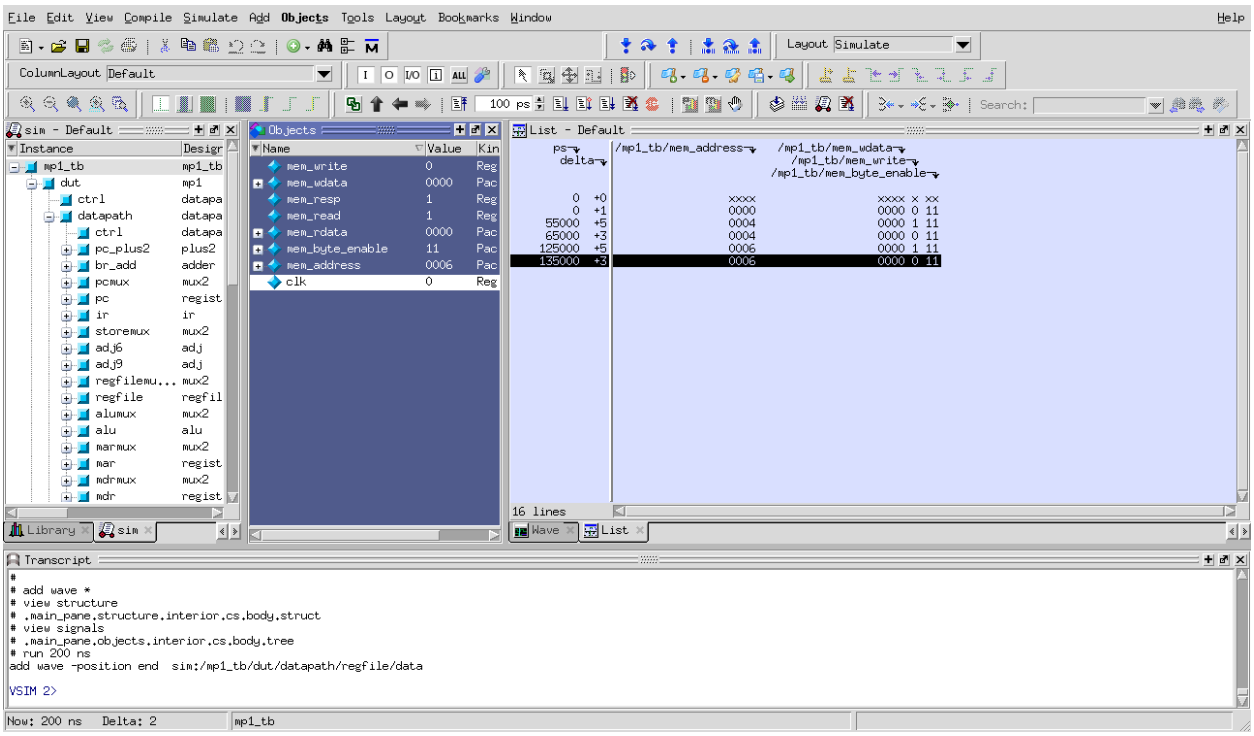


Figure 5: The lists window

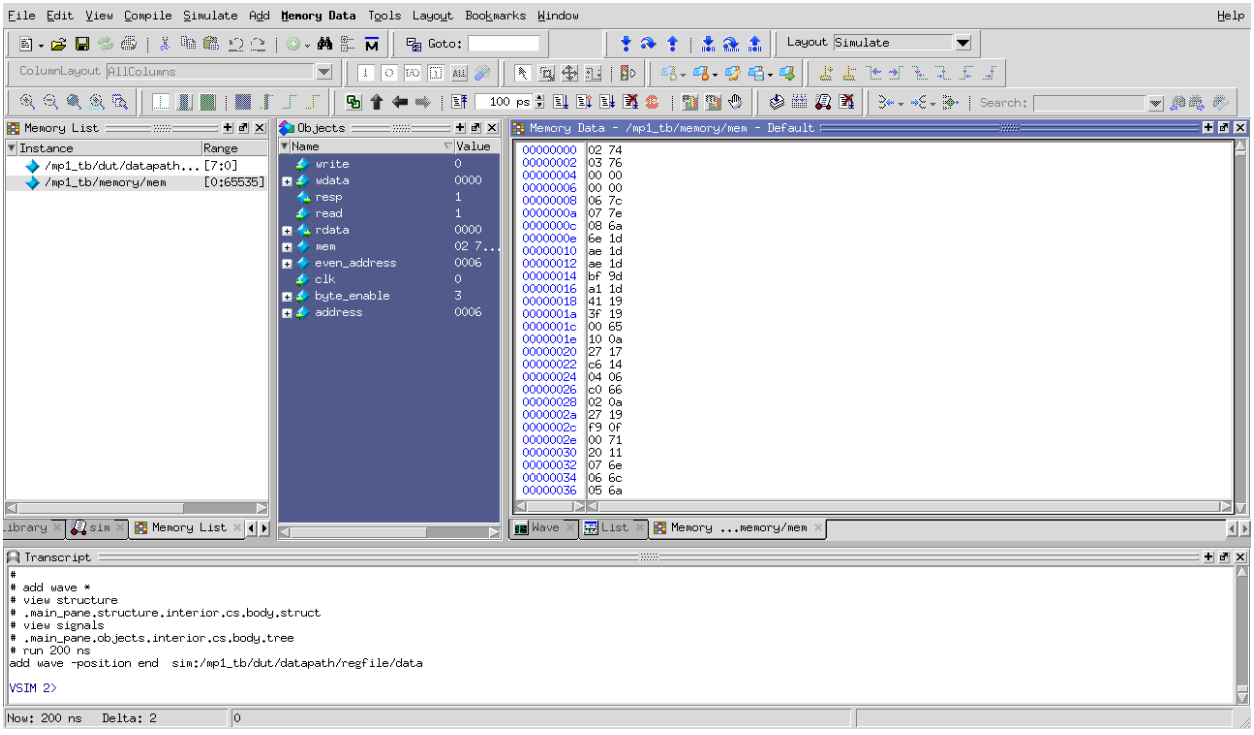


Figure 6: The memory lists window

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.

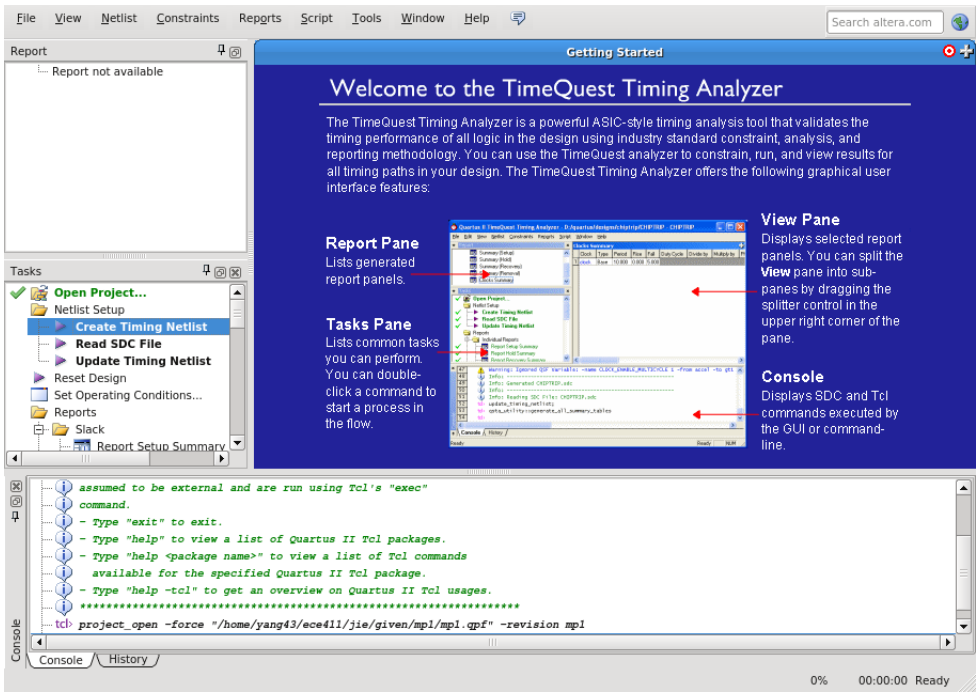


Figure 7: The TimeQuest Timing Analyzer

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.

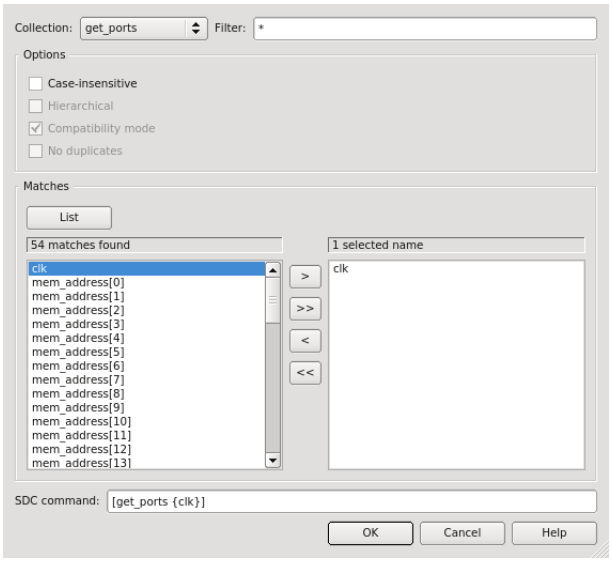


Figure 8: Selecting clock to constrain

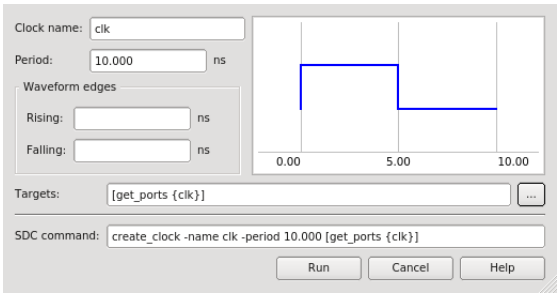


Figure 9: Specifying clock constraints

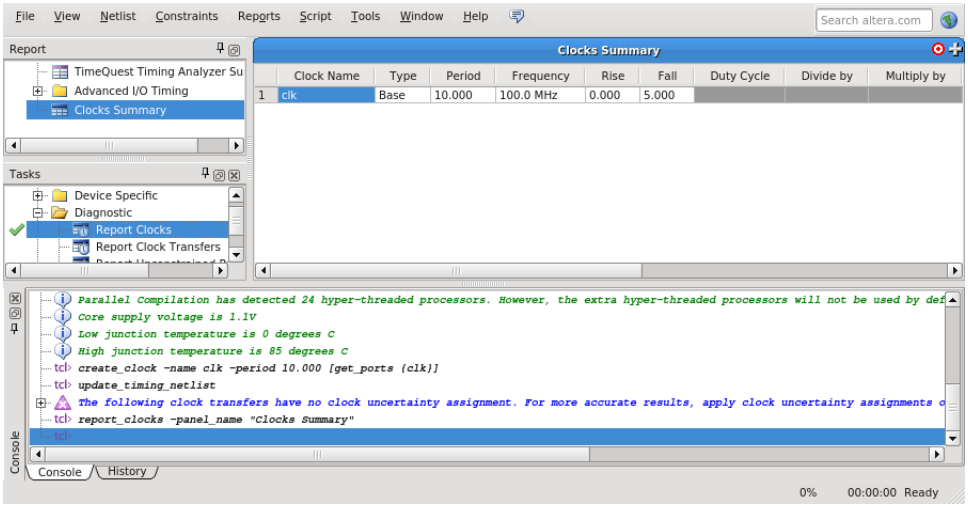


Figure 10: The clock report

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.

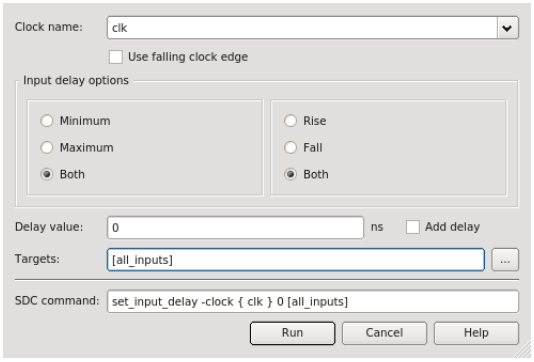


Figure 11: Specifying input constraints

Select **Contraints →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]**.

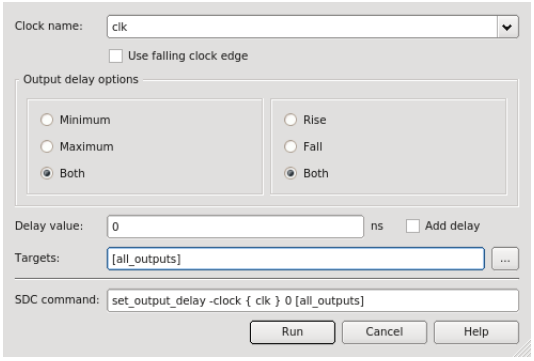


Figure 12: Specifying output constraints

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 *mp1.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.

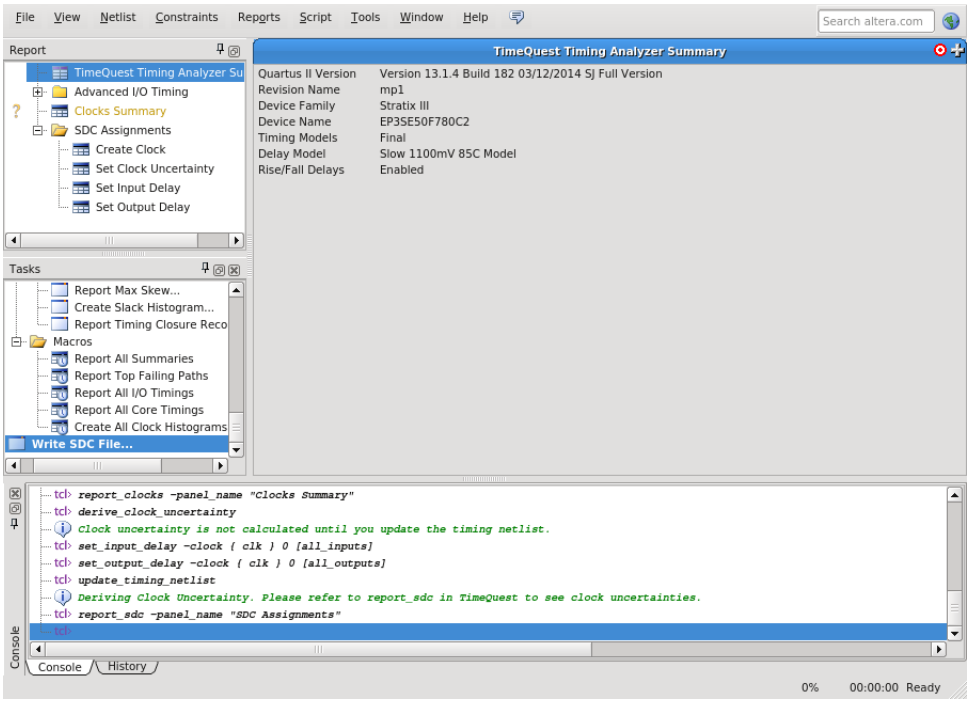


Figure 13: Writing the SDC file

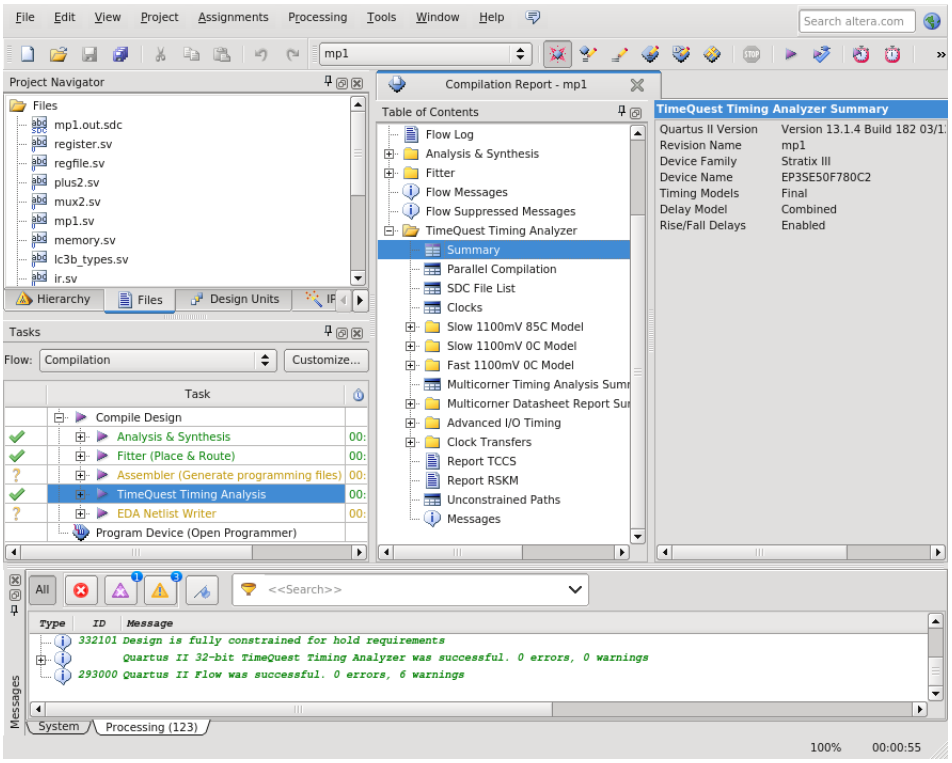


Figure 14: The timing analysis summary

7 Hand Ins

7.1 Check Point 1

For CP1, you must submit a design with

- all arithmetic computational instructions (immediate and register);
- Load Word and Store Word memory instructions;
- all conditional branch instructions (not JSR, JSRR);
- Both U-type instructions (LUI, AUIPC).

Additionally, you must submit a short RISC-V assembly program, *mp1/testcode/factorial.s*. Your program *must be iterative*. You must use load instructions to initialize registers. Reference the sample program located at *testcode/riscv_mp1test.s* 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 (which are imaginary, anyway). 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.

7.2 Final

The final handin requires you to complete the design by adding all missing instructions (with the exception of the instructions listed as not implemented in Section 2.1).

7.3 Autograder

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.

Additionally, **certain tests may be witheld from you until the CP1 and Final due-dates**. This means that you should not treat earlier autograding runs as your verification effort. **You must verify your own design**.

Since generating a timing report requires significantly more compilation effort than compiling for simulation, the autograder will only do grade timing at the deadlines and 24 hours prior to the deadlines.

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

8 Grading rubric

Item	**Pts**	**%**
Code	6	30
Targeted Tests	7	35
Larger Testcode	4	20
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 *mp1/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
ECE411DIR={path to your ECE411 git repo}
DEFAULT_TARGET=$ECE411DIR/mp1/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 *./mp1/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 *./mp1/testcode/my-test.s* and place the result in the default target path.

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

If successful, you should see a message similar to

```
Assembled ./mp1/testcode/my-test.s and wrote memory contents to
./mp1/simulation/modelsim/memory.lst.
```

B RTL

The tables in this section cover RTL for most of the controller states needed for CP1.

B.1 FETCH process

State	Data	Control
fetch1	MAR←PC	load_mar←1;
fetch2	while (mem_resp == 0) MDR←M[MAR];	load_mdr←1; mem_read←1;
fetch3	IR←MDR;	load_ir←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←rs1_out ⊕ i_imm; PC←PC + 4;	load_regfile←1; load_pc←1; cmpop←blt; regfilemux_sel←1; cmpmux_sel←1; rs1_addr←rs1

B.4 SLTIU instruction

State	Data	Control
FETCH		
DECODE		
s_imm	rd←rs1_out ⊕ i_imm; PC←PC + 4;	load_regfile←1; load_pc←1; cmpop←bltu; regfilemux_sel←1; cmpmux_sel←1; rs1_addr←rs1

B.5 SRAI instruction

State	Data	Control
FETCH		
DECODE		
s_imm	rd←rs1_out ⊕ i_imm; PC←PC + 4;	load_regfile←1; load_pc←1; aluop←alu_sra; rs1_addr←rs1

B.6 other immediate instructions

State	Data	Control
FETCH		
DECODE		
s_imm	rd←rs1_out ⊕ i_imm; PC←PC + 4;	load_regfile←1; load_pc←1; aluop←funct3; rs1_addr←rs1

B.7 BR instruction

State	Data	Control
FETCH		
DECODE		
br	PC←PC + (br_en ? b_imm : 4);	pcmux_sel←br_en; load_pc←1; alumux1_sel←1; alumux2_sel←2; aluop←alu_add; rs1_addr←rs1; rs2_addr←rs2

B.8 LW instruction

State	Data	Control
FETCH		
DECODE		
calc_addr	MAR←rs1_out + i_imm;	aluop←alu_add; load_mar←1; marmux_sel←1;
ldr1	while (mem_resp == 0) MDR←M[MAR];	load_mdr←1; mem_read←1;
ldr2	rd←MDR; PC←PC + 4;	regfilemux_sel←3; load_regfile←1; load_pc←1; rs1_addr←rs1

B.9 SW instruction

State	Data	Control
FETCH		
DECODE		
calc_addr	MAR←rs1_out + s_imm; data_out←rs2_out	alumux2_sel←3; aluop←alu_add; load_mar←1; load_data_out←1; marmux_sel←1;
str1	while (mem_resp == 0) M[MAR]←data_out;	mem_write←1;
str2	PC←PC + 4;	load_pc←1; rs1_addr←rs1; rs2_addr←rs2

B.10 AUIPC

State	Data	Control
FETCH		
DECODE		
s_auipc	rd←pc + u_imm; PC←PC + 4;	alumux1_sel←1; alumux2_sel←1; load_regfile←1; load_pc←1; aluop←alu_add;

B.11 LUI

State	Data	Control
FETCH		
DECODE		
s_lui	rd←u_imm; PC←PC + 4;	load_regfile←1; load_pc←1; regfilemux_sel←2; rs1_addr←rs1

C CPU

(a) Control to datapath

Name	Type
load_pc	logic
load_ir	logic
load_regfile	logic
load_mar	logic
load_mdr	logic
load_data_out	logic
pcmux_sel	pcmux::pcmux_sel_t
cmpop	branch_funct3_t
alumux1_sel	alumux::alumux1_sel_t
alumux2_sel	alumux::alumux2_sel_t
regfilemux_sel	regfilemux::regfilemux_sel_t
marmux_sel	marmux::marmux_sel_t
cmpmux_sel	cmpmux::cmpmux_sel_t
aluop	alu_ops

(b) Datapath to control

Name	Type
opcode	rv32i_opcode
funct3	logic [2:0]
funct7	logic [6:0]
br_en	logic
rs1	logic [4:0]
rs2	logic [4:0]

(c) Control to memory

Name	Type
mem_read	logic
mem_write	logic
mem_byte_enable	logic [3:0]

(d) Memory to control

Name	Type
mem_resp	logic

(e) Datapath to memory

Name	Type
mem_address	rv32i_word
mem_wdata	rv32i_word

(f) Memory to datapath

Name	Type
mem_rdata	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	pcmux::pc_plus4
cmpop	funct3
alumux1_sel	alumux::rs1_out
alumux2_sel	alumux::i_imm
regfilemux_sel	regfilemux::alu_out
marmux_sel	marmux::pc_out
cmpmux_sel	cmpmux::rs2_out
aluop	funct3
mem_read	1'b0
mem_write	1'b0
mem_byte_enable	4'b1111
rs1_addr	5'b0
rs2_addr	5'b0

D.2 Control diagram

See Appendix B for control state actions.

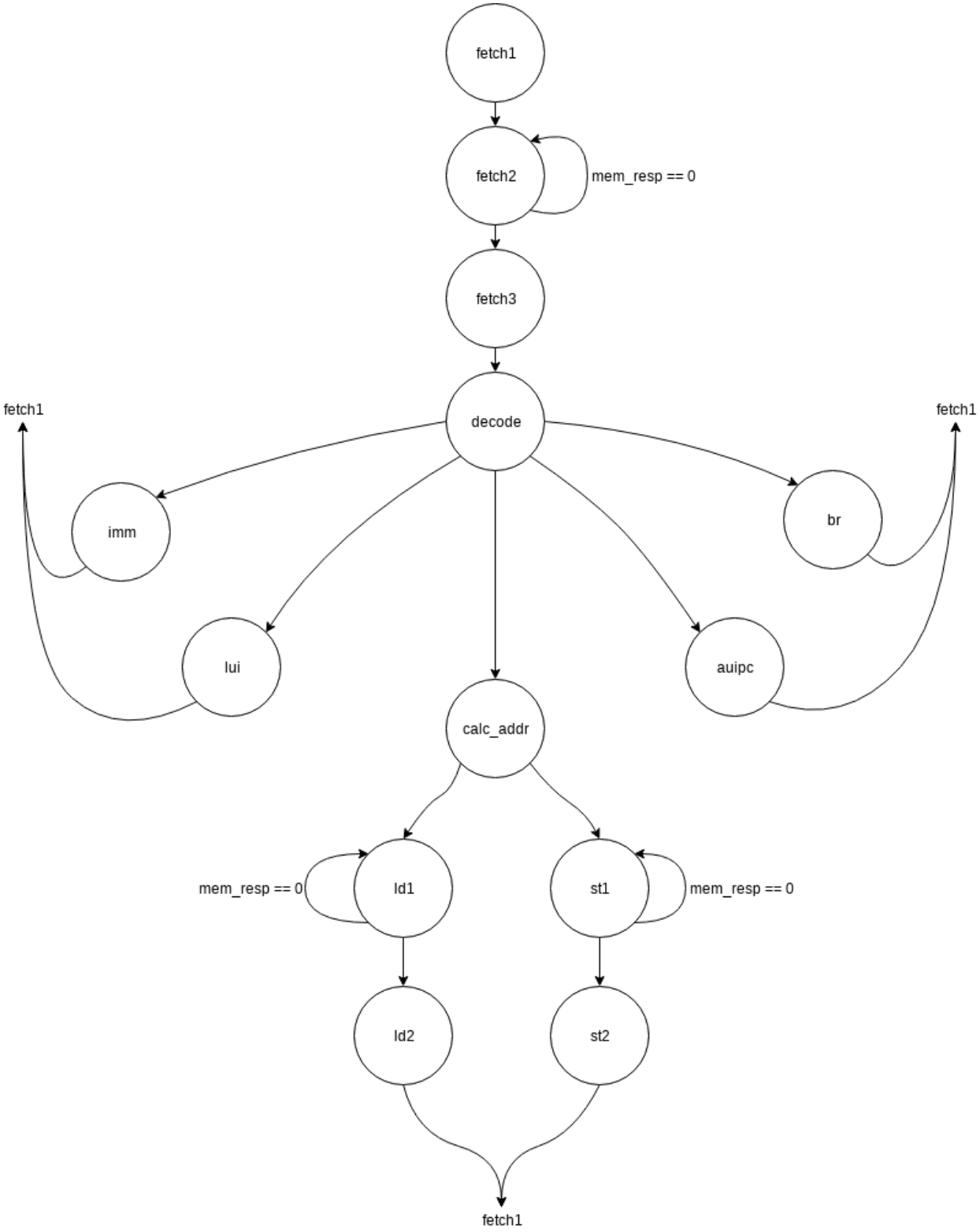


Figure 15: The RV32I control state diagram — sufficient for most of CP1

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	pcmux_sel_t	control	pcmux
alumux1_sel	alumux1_sel_t	control	alumux1
alumux2_sel	alumux2_sel_t	control	alumux2
regfilemux_sel	regfilemux_sel_t	control	regfilemux
marmux_sel	marmux_sel_t	control	marmux
cmpmux_sel	logic	control	cmpmux
aluop	alu_ops	control	ALU
rs1	rv32i_reg	IR	regfile, control
rs2	rv32i_reg	IR	regfile, control
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	alumux1	ALU
alumux2_out	rv32i_word	alumux2	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	control, regfilemux

E.2 Datapath diagram

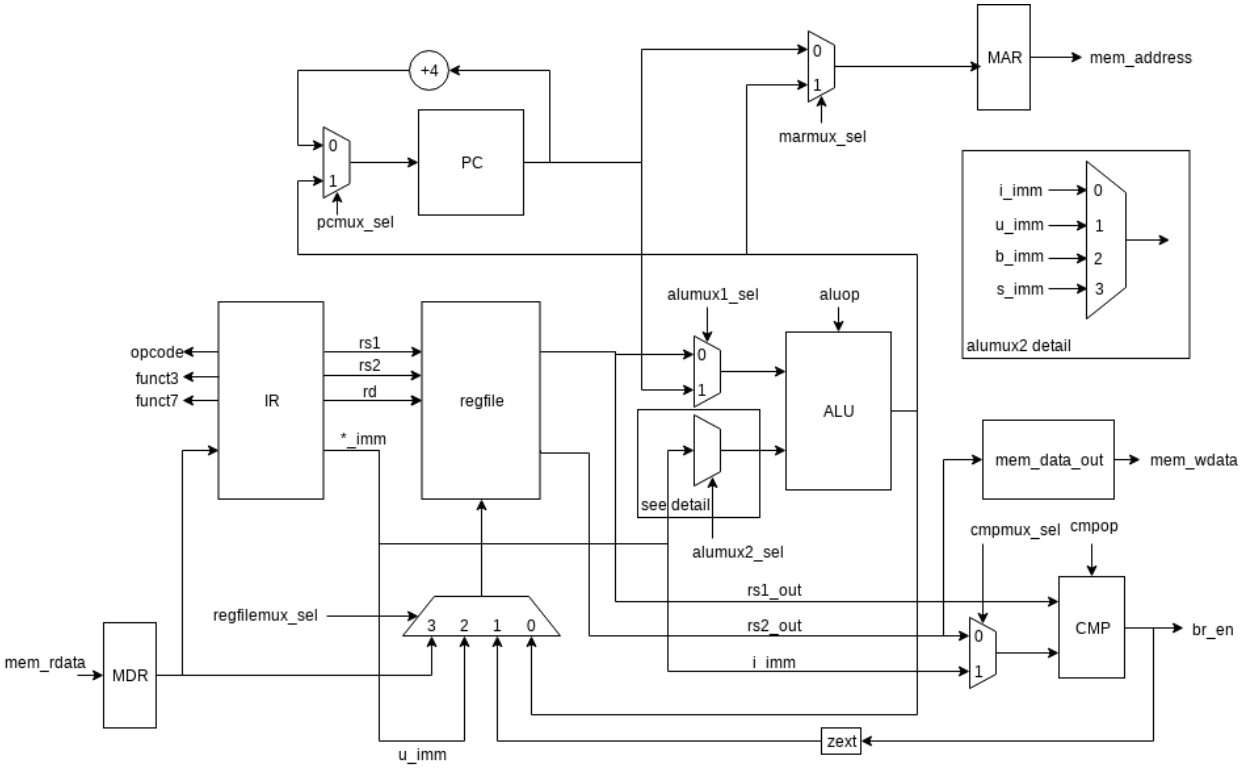


Figure 16: The RV32I datapath diagram — sufficient for most of CP1