Spring 2018, EC 513 Computer Architecture

Adaptive and Secure Computing Systems (ASCS) Laboratory
Department of Electrical and Computer Engineering, Boston University
Prof. **Michel A. Kinsy**

http://ascslab.org/courses/ec513/index.html

Class Project: BRISC-V© Extensions

Assigned
March 28th 2018

Milestones
a. Project Proposal
b. Mid-project (Phase I) Report
c. Project Presentations and Final Project Report

Due Dates
a. April 5th, 2018
b. April 19th, 2018
c. May 1st, 2018

I. Introduction

RISC-V instruction set is recently proposed by a group of researchers at EECS Department of University of California, Berkeley. Main purposes of proposing RISC-V ISA is summarized below:

- To have a completely free-access ISA for both academic and industrial activities.
- To support different variations in processor design including 1) processor widths of 32, and 64 bits, 2) single, multi and many core designs, 3) FPGA and ASIC implementations.
- To have a core set of base integer instructions that can be extended with other categories of instructions, allowing architects to include only the needed features.
- To support both user and supervisor modes of working for the processor.
- To support variable-length instructions.
- To support custom instructions based on specific tasks the processor is intended to run in specific fields.

The RISC-V instruction set has been used by researchers to test architectures relating to memory and cache sub-systems, power and performance improvement, among others. There are also several open-source CPU designs, including the 64-bit Berkeley Out of Order Machine (BOOM), 64-bit Rocket, five 32-bit Sodor CPUs from Berkeley, picorv32 by Clifford Wolf, scr1 from Syntacore, and our own open-source RISC-V processor implementation, called BRISC-V©. The parameterized BRISC-V© implementation, developed at the Adaptive and Secure Computing Systems laboratory (ASCS Lab) of Boston University, uses the RV32I version of the ISA.

II. Project Overview

In lecture, we have covered the key and time-tested concepts in computer architecture: pipelining, complex pipelining (Superscalar, Out-of-Order Execution, VLIW, Vector, Hardware Multi-threading, Branch Prediction, Speculative Execution, Caching, Memory Virtualization,

Multi-core, etc.). All these concepts exploit one or more of these parallelism modalities: Instruction Level Parallelism (ILP), Data Level Parallelism (DLP) and Task Level Parallelism (TLP) to improve the cycle-per-instruction (CPI), a/the core processor performance measurement metric. In this project, you will select one of these concepts, implement it, and optimize it for hands-on architecture design trade-offs experience.

III. Design Base

To start off your design, we are providing you with the BRISC-V© single cycle processor base.

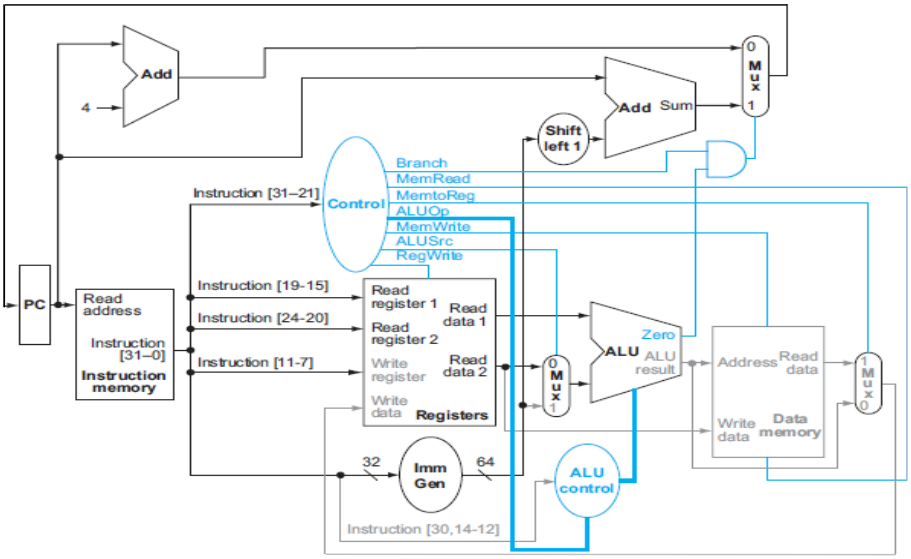


Figure 1: Canonical view of a single-cycle, non-pipeline of the BRISC-V© CPU.

BRISC-V© is part of the Heracles© [1]design platform, an open-source, functional, parameterized, synthesizable research and teaching tool for architectural exploration and hardware-software co-design.

The provided BRISC-V© single cycle processor base comprises the soft hardware (HDL) modules and their testbeds, and application compiler. It is designed with a high degree of modularity to support fast exploration of different architectural features and memory system organizations. It is a component-based framework with parameterized interfaces and strong emphasis on module reusability. The compiler toolchain is used to map C based applications onto the processor.

IV. Project Outline

Phase I: Pipelining the base version of the BRISC-V© processor and add a multi-level direct-mapped cache to the design.

Task 1: Pipelining the base version of the BRISC-V© processor

- You can select the number of pipeline stages (ideally between 2 to 10). In your report, explain clearly why you select a given number of stages.

[1] http://ascslab.org/research/heracles/index.html

- We observed in lecture that the main difficulty in pipelining is pipeline hazards, that is, data dependencies between instructions in the pipeline. We discussed two possibilities for dealing with pipeline hazards: stalling and data forwarding. Stalling means that we stop issuing instructions when we detect that the next instruction that we will issue depends on the result of in-flight instructions. An alternative to stalling is data forwarding. Rather than waiting for the instruction to complete and commit data to the register file, we forward the data from an earlier instruction in the pipeline directly to later instructions. You can elect to implement stall or bypass. Again, make sure to highlight the design choice in your report.

Task 2: Multi-level direct-mapped cache design

- You can implement two or three levels of caches (L1, L2 and Main Memory or L1, L2, L3 and Main Memory). You should have two L1 caches (one for instructions and one for data), but single inclusive L2 or L3.
- Describe your caching policies: Write-back or Write-Through and Write Allocate or No Write Allocate.
- You do not have to implement any cache coherence protocol.
- For this phase of the project, you are not required to have it working with the processor. But you should have a testbed implemented to test it.
- You can also use the Heracles© design base for inspiration on how you may want to implement your caches.

Phase II: Implement one complex architecture feature to the design in Phase I.
Task 1: The memory system with the caches should be integrated and tested with the processor.
Task 2: Implement the additional feature.
Task 3: Evaluate your processor design performance (e.g, CPI, cache miss rates).

Project Timeline

1. Apr 5: Project Proposals due.
2. Apr 19: Mid-project (Phase I) reports due
3. May 1: Project Presentations and Final project report due.

Deliverables

One group member uploads a .zip file titled [group name] _project.zip (example: brisc_bros_project.zip) to the Blackboard assignment posting. For both the mid-project and final submissions, per group, you will submit (1) Verilog source code, (2) Verilog testbeds for each new or modified module, and (3) 2-3 page reports.

V. BRISC-V© Base

The BRISC-V© provided base is split into hardware and software directories. In the hardware folder there are two sub folders named ***testbeds*** and ***src***. The testbed directory contains an individual testbed for each module of the BRISC-V© processor. The ***src*** sub-folder contains the Verilog files for each module of BRISC-V©.

In the software directory there are three folders called ***applications***, ***compiler-scripts*** and ***riscv-compiler***. There is also a file named compile.sh which will generate all the binary files in the binaries directory, from the C programs in ***applications/src/***, using the ***riscv-compiler***. Editing ***compile.sh*** can allow for not provided program binaries to be generated. In the applications folder there are two sub folders, ***binaries*** and ***src***. In the ***src*** folder there are 12 sample C programs and in the binaries folder there are sample program's .asm, .dump, .mem, .s, and .vmh versions. In compiler-scripts there are perl scripts used to arrange the BRISC-V© program kernel. The folder ***riscv-compiler*** contains two folders named ***bin*** and ***libexec***. Libexec contains a number of sub-folders which are empty while bin contains the RISC-V gcc tools used for generating binary files to compile and generate binaries for the RISC-V rv32 ISA.

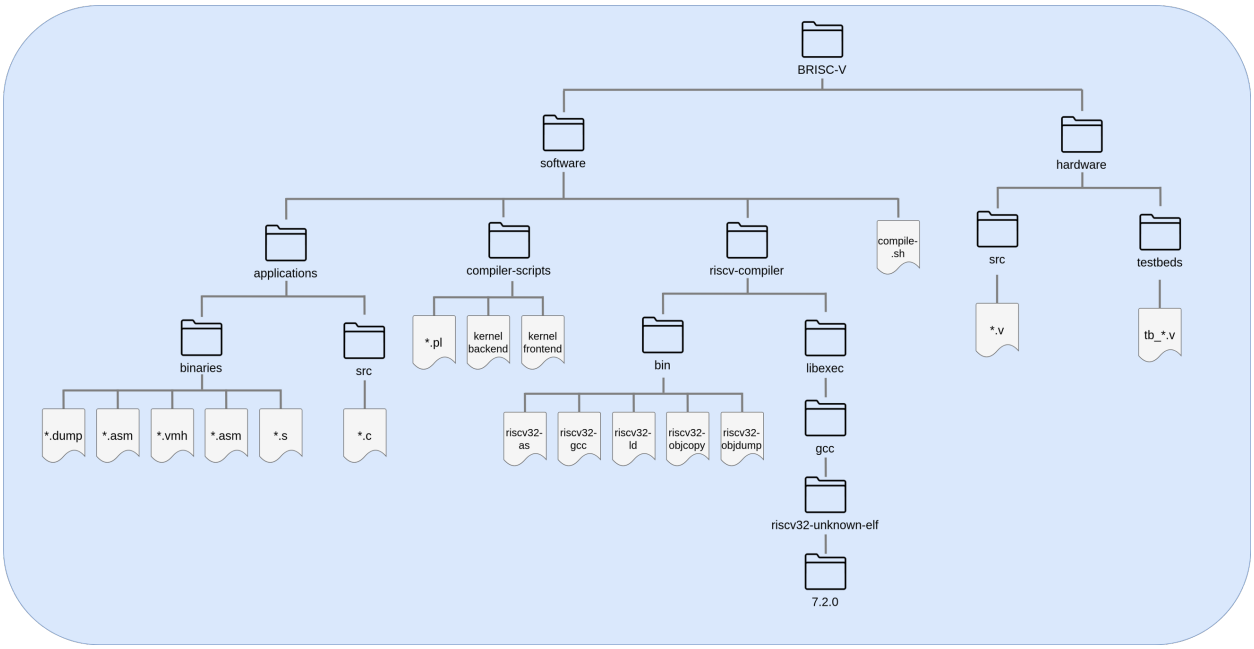


Figure 2:Illustrative schematic describing the organizations of the hardware and software folders.

Program compilation flow

To assist in developing software for the BRISC-V© processor, it is accompanied with a GCC RISC-V cross-compiler. The figure above depicts the software flow for compiling a C program into the compatible BRISC-V© instruction code that can be executed on the processor. The compilation process consists of a series of seven steps.

1. First, the user invokes ***riscv32-unknown-elf-gcc*** to translate the C code into assembly language (*e.g., ./riscv32-unknown-elf-gcc -S fibonacci.c*).
2. In step 2, the assembly code is then run through the linker to set up the stack pointer and return value registers (e.g., *./link.pl fibonacci.s*). Its output is a .asm file.
3. In step 3, the user compiles the assembly file into an object file using the cross-compiler. This is accomplished by executing ***riscv32-unknown-elf-as*** on the .asm file (e.g., *./riscv32-unknown-elf-as fibonacci.asm –o fibonacci.o*).
4. In this step, all the jump addresses are properly linked with *./riscv32-unknown-elf-ld -N -Ttext 0x0004 --unresolved-symbols=ignore-all fibonacci.o –ofibonacci.*

5. In step 5, the object file is disassembled using the ***riscv32-unknown-elf-objdump*** command (e.g., *./ riscv32-unknown-elf-objdump fibonacci.o*). Its output is a .dump file.
6. In step 6, the constructor script is called to transform the dump file into a Verilog memory .vmh file format (e.g., *./riscv32-unknown-elf-objcopy fibonacci.dump*).
7. Finally, a second constructor script is called to transform the dump file into another Verilog memory .mem file format (e.g., ./dump2vmh fibonacci.dump). Different Verilog simulations or FPGA synthesis tools use different formats, i.e., .vmh or .mem. They contain the same data. Programs/Applications that have some initial values/data stored in memory will also have a data file generated for them (e.g., data_fibonacci.vmh/mem).

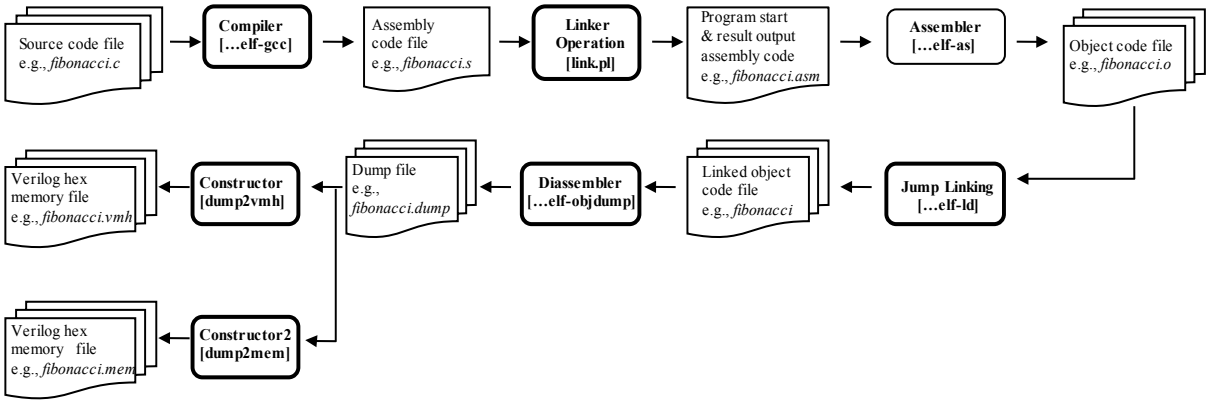


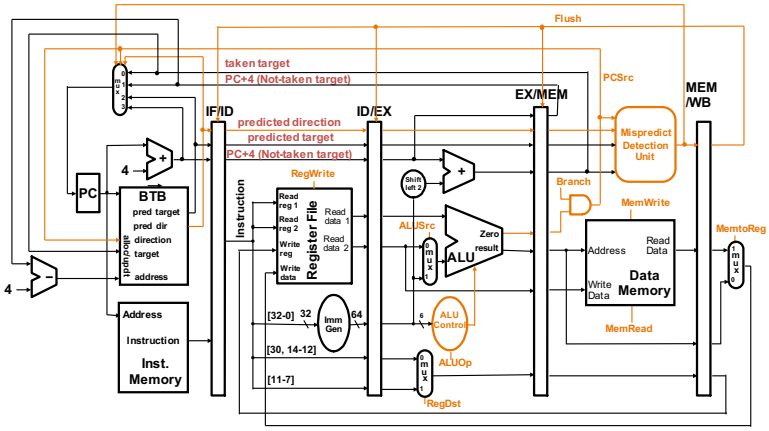
Figure 3: Application compilation toolchain

For script-based compilation, if you run ***./compile.sh***, it will take a set of predefined C applications/programs in the ***application/src*** folder and compile all of them. If you would like to compile your own application (e.g., *albert_s_ beautiful_code.c*) with your own stack pointer size (*albert_s_ stack*, a decimal number), you can execute ***./compile.sh albert_s_ beautiful_code.c albert_s_ stack***. (e.g., *./compile.sh foo.c 128*).

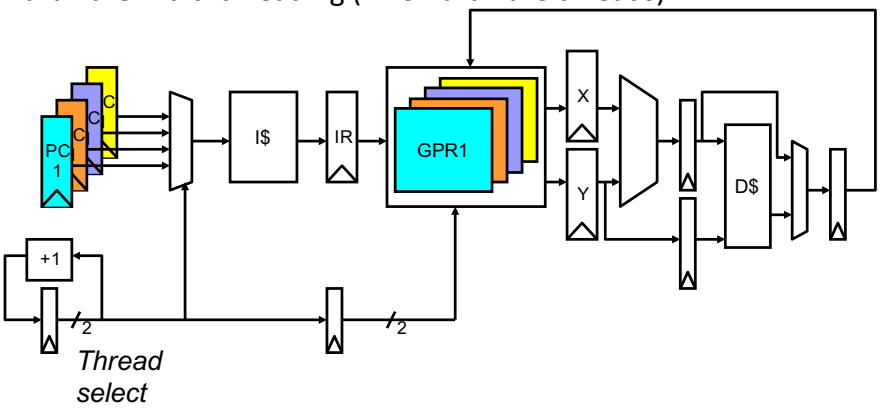
Project Ideas

These are just suggested project ideas. Please feel free to be creative.

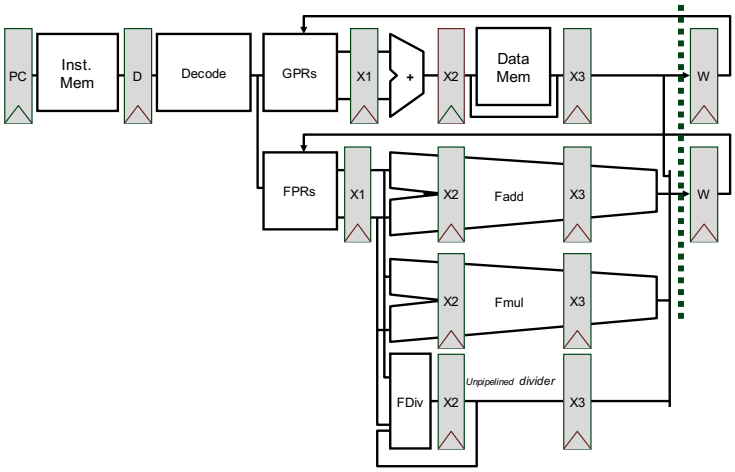
1. Branch Predictor



2. Hardware Multi-threading (Two hardware threads)



3. Memory Virtualization (TLB Implementation)
4. Out-of-Order Execution
5. Dynamic Register Renaming Unit
6. Vector Extension
7. Superscalar (Add additional execution units to the processor, e.g., a multiplication unit with the correct logic modification)



8. Very long instruction word (VLIW) – two or three instructions bundles
9. Special co-processor (e.g., a Neural Network Accelerator director connected to the processor

General Notes: Although you can develop processor design on any machine, we suggest you use the PHO 307 machines that we provided, and strongly suggest that you test your design on it. Do not use the VM, just the native computer.

Setting up:

To obtain the materials for the project, download the file at:

`https://bit.ly/2GhJIww`

a. Compilation

You can extract the project by running:

```
$ tar –xfz ec513-project.tar.gz
```

Now you have the project base containing both the BRISC-V© processor written in Verilog (located in **`ec513-project/hardware`**) and the cross-compiler (located in **`ec513-project/software`**) that takes your C code and provides you with raw memory files to run on the bare processor. Along with the compiler, you will find some simple applications (located in **`ec513-project/software/applications`**), and associated scripts.

Once you have extracted your project, you compile the applications for BRISC-V©:

```
$ cd ec513-project/software
$ ./compile.sh
```

If everything went correctly, you should see a message:

```
COMPILATION SUCCESSFUL!
```

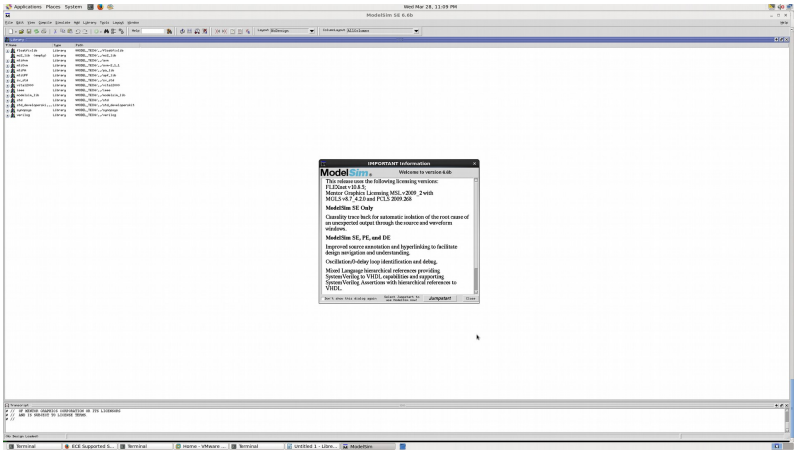
In the **`software/applications/binaries/`** directory you should be able to find ***.vmh** and ***.mem** that you will use in your simulations.

b. Simulation

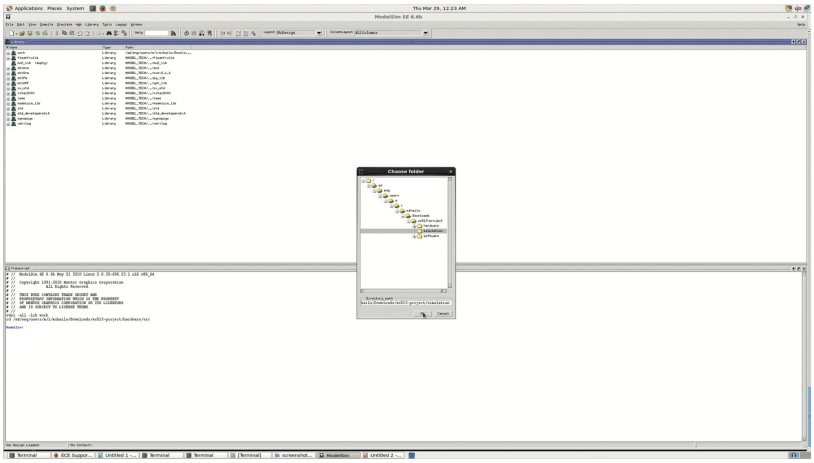
Step 1: Start **ModelSim** simulation environment: To run **ModelSim**, type the following in the terminal:

```
$ /ad/eng/opt/mentor/modelsim/modeltech/bin/vsim
```

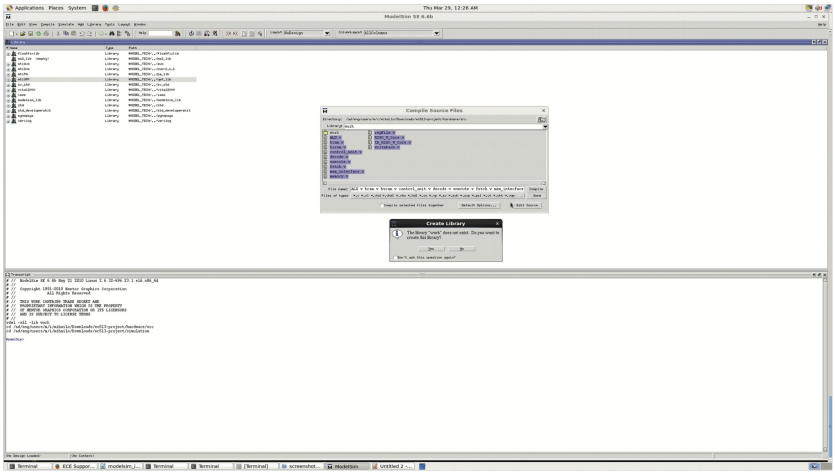
A new window should appear, and it should display a popup. You can safely close the popup.



Step 2: Setting up the design library in **ModelSim**:
In the main menu, click on the file menu item and on the change directory sub item. A file explorer will appear and you will want to navigate to **your_project_location/ec513-project/simulation/**.

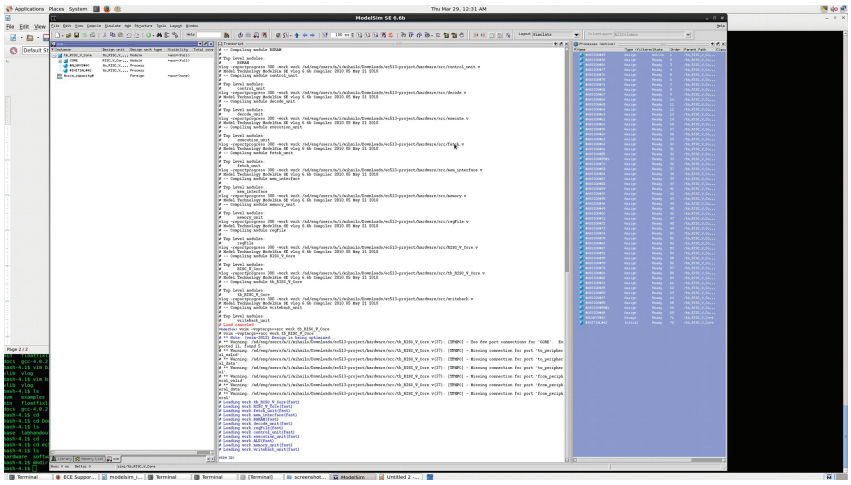


Step 3: Compiling Verilog sources: In the main menu, click on compile, and compile. A popup should open, and you should navigate to your Verilog sources in **ec513-project/hardware/src/**. Select all of the verilog sources, and click compile.

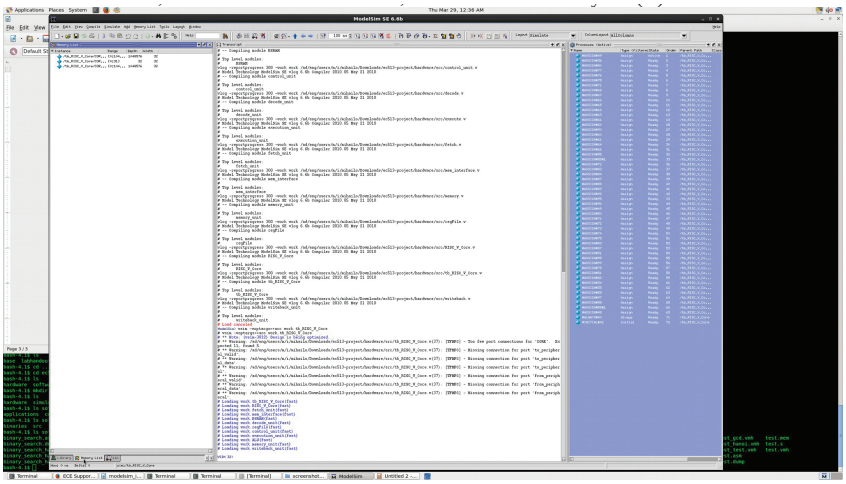


You may think that the compile button did not work, but a “create library” popup should have appeared, possibly behind the “compile source files” popup. When prompted on whether you want to create a library, click yes. Click compile, and then done.

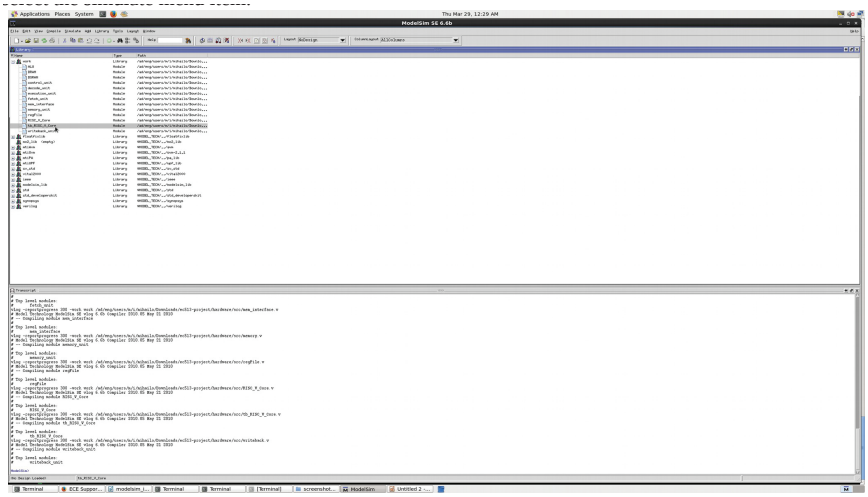
Step 4: Load simulation: In the library pane, you should see multiple libraries. Expand the first one – work, and you should see all your Verilog files. Right click on the testbench **tb_RISC_V_Core**, and select the simulate menu item.



After clicking on simulate, you should see this window:

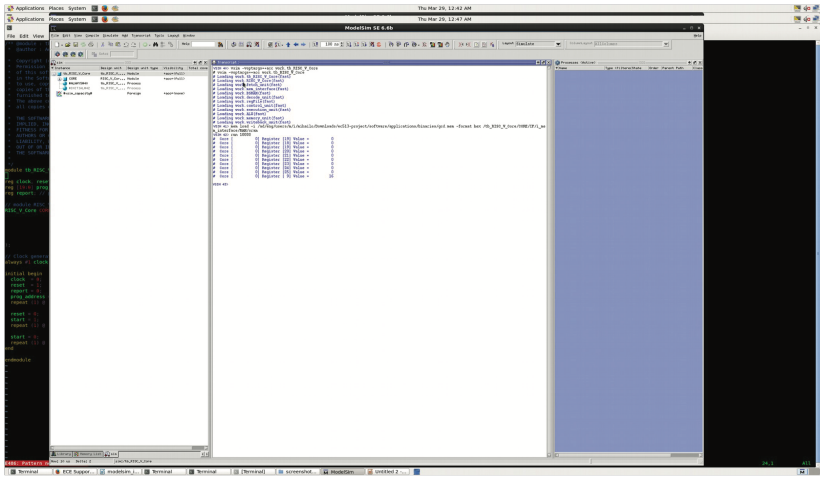


Step 5: Load the binaries. Before we can run our processor, we need to populate instruction memory with our binaries, and for some applications, the data memory as well. Assuming you have run the compile script, in the **ec513-project/software/applications/binaries** directory you should find some *.mem files.
In the new window, click on the View menu item, and select the “memory list (w)” sub-menu item.



You will see three different memories: the fetch stage instruction memory, the decode stage reigster file, and the data memory. We want to populate the first one, **IF/i_mem_interface/RAM/sram**. If you right click on the first item and select “view contents”, a pane will open on the right-hand side. Notice that it is currently undefined – it is filled with x-es. Right click on the new pane, and select “Import Data Patterns”. A popup will appear. In the “Load Type” segment select “File only”, and in the “File format” segment select “Verilog hex”. Next, click on browse, and navigate to **ec513-project/software/applications/binaries/**, and from there select the right binary, for example gcd.mem. Click OK, and in the memory pane you should see the contents of your .mem file in the instruction memory.

Step 6: Run the simulation. In the Transcript pane, go ahead and type “run 10000”. This command will run the simulation for 10000 ns.



We see that the register 9 contains the value 16, which is the greatest common divisor of 64 and 48.

End Note: Make sure to manage your time properly and distribute the workload fairly.