University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

EECS 151/251A, Spring 2020
Brian Zimmer, Nathan Narevsky, and John Wright
Modified by Arya Reais-Parsi and Cem Yalcin (2019), Tan Nguyen (2020)

Project Specification
EECS 151/251A RISC-V Processor Design

Contents

1 Introduction

The primary goal of this project is to familiarize students with the methods and tools of digital design. In order to make the project both interesting and useful, we will guide you through the implementation of a CPU that is intended to be integrated on a modern SOC. Working alone or in teams of 2, you will be designing a simple 3-stage CPU that implements the RISC-V ISA, developed here at UC Berkeley. If you work in a team, you both must work on the project together (i.e. you are not allowed to divide up the work), and you will both receive the same grade.

Your first and most important goal is to write a functional implementation of your processor. To better expose you to real design decisions, you will also be tasked with improving the performance of your processor. You will be required to meet a minimum performance to be specified later in the project.

You will use Verilog HDL to implement this system. You will be provided with some testbenches to verify your design, but you will be responsible for creating additional testbenches to exercise your entire design. Your target implementation technology will be the ASAP7 7nm Educational PDK, a predictive model technology used for instruction. The project will give you experience designing synthesizeable RTL (Register Transfer Level) code, resolving hazards in a simple pipeline, building interfaces, and approaching system-level optimization.

Your first step will be to map our high level specification to a design which can be translated into a hardware implementation. You will then generate and debug that implementation in Verilog. These steps may take significant time if you do not put effort into your system architecture before attempting implementation. After you have built a working design, you will be optimizing it for speed in the 7nm technology that we have been using this semester.

1.1 RISC-V

The final project for this class will be a VLSI implementation of a RISC-V (pronounced *risk-five*) CPU. RISC-V is a new instruction set architecture (ISA) developed here at UC Berkeley. It was originally developed for computer architecture research and education purposes, but recently there has been a push towards commercialization and industry adoption. For the purposes of this lab, you don't need to delve too deeply into the details of RISC-V. However, it may be good to familiarize yourself with it, as this will be at the core of your final project. Check out the official Instruction Set Manual and explore `http://riscv.org` for more information.

- Read through sections 2.2 and 2.3 starting on page 11 in the RISC-V Instruction Set Manual to understand how the different types of instructions are encoded.
- Read through sections 2.4, 2.5, and 2.6 starting on page 13 in the Instruction Set Manual and think about how each of the instructions will use the ALU.

You do not need to read 2.7 or 2.8, as you will not be implementing those instructions in the project.

1.2 Project phases

Your project will consist of two different phases: front-end and back-end. Within each phase, you will have multiple checkpoints that will ensure you are making consistent progress. These checkpoints will contribute (although not significantly) to your final grade. You are free to make design changes after they have been checked off if they will help subsequent phases or improve QOR.

In the first phase (front-end), you will design and implement a 3-stage RISC-V processor in Verilog, and run simulations to test for functionality. At this point, you will only have a functional description of your processor that is independent of technology (there are no standard cells yet). You have about 5 weeks to complete the first phase, but you are highly encouraged to try to finish each checkpoint early, as each checkpoint will be released before the due date of the ongoing one. Everything will take much longer than you expect, and finishing early gives you more time to improve your QOR (Quality of Results, e.g. clock period).

In the second phase (back-end), you will implement your front-end design in the ASAP7 7nm kit using the VLSI tools you used in lab. When you have finished phase 2, you will have a design that could actually be fabricated if this were a real process. You will have about 2 weeks to complete the second phase after its release.

1.3 Philosophy

This document is meant to describe a high-level specification for the project and its associated support hardware. You can also use it to help lay out a plan for completing the project. As with any design you will encounter in the professional world, we are merely providing a framework within which your project must fit.

You should consider the GSI(s) a source of direction and clarification, but it is up to you to produce a fully functional design, as well as a physical implementation. We will attempt to help, when possible, but ultimately the burden of designing and debugging your solution lies on you.

1.4 General Project Tips

Be sure to use top-down design methodologies in this project. We began by taking the problem of designing a basic computer system, modularizing it into distinct parts, and then refining those parts into manageable checkpoints. You should take this scheme one step further; we have given you each checkpoint, so break each into smaller, manageable pieces.

As with many engineering disciplines, digital design has a normal development cycle. In the norm, after modularizing your design, your strategy should roughly resemble the following steps:

Design your modules well, make sure you understand what you want before you begin to code.

Code exactly what you designed; do not try to add features without redesigning.

Simulate thoroughly; writing a good testbench is as much a part of creating a module as actually coding it.

Debug completely; *anything which can go wrong with your implementation will.*

Document your project thoroughly as you go. Your design review documents will help, but you should never forget to comment your Verilog and to keep your diagrams up to date. Aside from the final project report (you will need to turn in a report documenting your project), you can use your design documents to help the debugging process. Finish the required features first. Attempt extra features after everything works well.

This project is divided into checkpoints. Each checkpoint will be due 1 to 2 weeks after its release, but the next checkpoint will be released early. Use this to your advantage- try to get ahead so that you have additional time to debug. Your TA will clarify the specific timeline for your semester.

The most important goal is to design a functional processor- this alone is 50-60% of the final grade, and you must have it **working completely** to receive *any* credit for performance.

2 Front-end design (Phase 1)

The first phase in this project is designed to guide the development of a three-stage pipelined RISC-V CPU that will be used as a base system for your back-end implementation.

Phase 1 will last for 5 weeks and has weekly checkpoints.

- Checkpoint 1: ALU design and pipeline diagram (due End of 03/16-03/22, 2020)
- Checkpoint 2: Core implementation (due End of 04/06-04/12, 2020)
- Checkpoint 3: Core + memory system implementation (due End of 04/20-04/26, 2020)

2.1 Project Setup

The skeleton files for the project will be delivered as a git repository provided by the staff. You should clone this repository as follows. It is highly recommended to familiarize yourself with git and use it to manage your development.

```
% git clone /home/ff/eecs151/labs/project_skeleton /path/to/my/project
```

As soon as you start your project, you must post your group information as a private note on Piazza. Please provide each group member's name, student ID number, and instructional account name for the group members (e.g. eecs151-aa). Please do this even if you are working alone, as these git repos will be used for part of the final checkoff. Once it is setup you will be given a team number, and you will be given a repo hosted on the servers for version control for the project. You should be able to add the remote host of "geecs151:teamXX" where "XX" is the team number that you are assigned. An example working flow to be able to pull from the skeleton as well as push/pull with your team repository is shown below:

```
% git clone /home/ff/eecs151/labs/project_skeleton /path/to/my/project
% git remote add myOrigin geecs151:teamXX
```

Then to pull changes from the skeleton, you would need to type:

```
% git pull origin master
```

To pull changes from your team repository you would type:

```
% git pull myOrigin master
```

And to push changes to your team repository, you would usually want to pull first (above) and then type:

```
% git push myOrigin master
```

3 Checkpoint #1: ALU design and pipeline diagram

The ALU that we will implement in this lab is for a RISC-V instruction set architecture. Pay close attention to the design patterns and how the ALU is intended to function in the context of the RISC-V processor. In particular it is important to note the separation of the datapath and control used in this system which we will explore more here.

The specific instructions that your ALU must support are shown in the tables below. The branch condition should **not** be calculated in the ALU. Depending on your CPU implementation, your ALU may or may not need to do anything for branch, jump, load, and store instructions (i.e., it can just output 0).

3.1 Making a pipeline diagram

The first step in this project is to make a pipeline diagram of your processor, as described in lecture. You only need to make a diagram of the datapath (not the control). Each stage should be clearly separated with a vertical line, and flip-flops will form the boundary between stages. It is a good idea to name signals depend on what stage they are in (eg. s1_killf, s2_rd0). Also, it is a good idea to separately name the input/output (D/Q) of a flip flop (eg. s0_next_pc, s1_pc). Draw your diagram in a drawing program, because you will need to keep it up-to-date as you build your processor. It helps to print out scratch copies while you are debugging your processor and to keep your drawings revision-controlled with git. Once you have finished your initial datapath design, you will implement the main building block in the datapath—the ALU.

3.2 ALU functional specification

Given specifications about what the ALU should do, you will create an ALU in Verilog and write a test harness to test the ALU.

The encoding of each instruction is shown in the table below. There is a detailed functional description of each of the instructions in Section 2.4 (starting on page 13) of the Instruction Set Manual. Pay close attention to the functional description of each instruction as there are some subtleties.

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	SB-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	UJ-type

RV32I Base Instruction Set

imm[31:12]			rd	0110111	LUI rd,imm
imm[31:12]			rd	0010111	AUIPC rd,imm
imm[20|10:1|11|19:12]			rd	1101111	JAL rd,imm
imm[11:0]		rs1	000	rd	1100111	JALR rd,rs1,imm
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ rs1,rs2,imm
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE rs1,rs2,imm
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT rs1,rs2,imm
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE rs1,rs2,imm
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU rs1,rs2,imm
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU rs1,rs2,imm
imm[11:0]		rs1	000	rd	0000011	LB rd,rs1,imm
imm[11:0]		rs1	001	rd	0000011	LH rd,rs1,imm
imm[11:0]		rs1	010	rd	0000011	LW rd,rs1,imm
imm[11:0]		rs1	100	rd	0000011	LBU rd,rs1,imm
imm[11:0]		rs1	101	rd	0000011	LHU rd,rs1,imm
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB rs1,rs2,imm
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH rs1,rs2,imm
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW rs1,rs2,imm
imm[11:0]		rs1	000	rd	0010011	ADDI rd,rs1,imm
imm[11:0]		rs1	010	rd	0010011	SLTI rd,rs1,imm
imm[11:0]		rs1	011	rd	0010011	SLTIU rd,rs1,imm
imm[11:0]		rs1	100	rd	0010011	XORI rd,rs1,imm
imm[11:0]		rs1	110	rd	0010011	ORI rd,rs1,imm
imm[11:0]		rs1	111	rd	0010011	ANDI rd,rs1,imm
0000000	shamt	rs1	001	rd	0010011	SLLI rd,rs1,shamt
0000000	shamt	rs1	101	rd	0010011	SRLI rd,rs1,shamt
0100000	shamt	rs1	101	rd	0010011	SRAI rd,rs1,shamt
0000000	rs2	rs1	000	rd	0110011	ADD rd,rs1,rs2
0100000	rs2	rs1	000	rd	0110011	SUB rd,rs1,rs2
0000000	rs2	rs1	001	rd	0110011	SLL rd,rs1,rs2
0000000	rs2	rs1	010	rd	0110011	SLT rd,rs1,rs2
0000000	rs2	rs1	011	rd	0110011	SLTU rd,rs1,rs2
0000000	rs2	rs1	100	rd	0110011	XOR rd,rs1,rs2
0000000	rs2	rs1	101	rd	0110011	SRL rd,rs1,rs2
0100000	rs2	rs1	101	rd	0110011	SRA rd,rs1,rs2
0000000	rs2	rs1	110	rd	0110011	OR rd,rs1,rs2
0000000	rs2	rs1	111	rd	0110011	AND rd,rs1,rs2
imm[11:0]		rs1	001	rd	1110011	CSRRW rd,rs1,imm
imm[11:0]		rs1	101	rd	1110011	CSRRWI rd,rs1,imm

3.3 Lab Files

We have provided a skeleton directory structure to help you get started.

Inside, you should see a `src` folder, as well as a `tests` folder. Similar to the setting of previous labs, there are also some YAML files for using HAMMER to run simulation, synthesis, and place-and-route. The `src` folder contains all of the verilog modules for this phase, and the `tests` folder contains some RISC-V test binaries for your processor.

3.4 Testing the Design

Before writing any of modules, you will first write the tests so that once you've written the modules you'll be able to test them immediately. This is effectively Test-driven Development (TDD). Writing tests first is good practice- it forces you to write thorough tests, and ensures that tests will exist when you need to rapidly iterate through module design tweaks. Thorough understanding of the expected functionality is key to writing good tests (or RTL). You will be expected to write unit tests for any modules that you design and implement and write integration tests. Unit tests will verify the functionality of individual modules against your specification. Integration tests verify that all the modules work as a system once you connect them together.

3.4.1 Verilog Testbench

One way of testing Verilog code is with testbench Verilog files. The outline of a test bench file has been provided for you in `ALUTestbench.v`. There are several key components to this file:

- `` `timescale 1ns / 1ps `` - This specifies, in order, the reference time unit and the precision. This example sets the unit delay in the simulation to 1ns (i.e. #1 = 1ns) and the precision to 1ps (i.e. the finest delay you can set is #0.001 = 1ps).
- The clock is generated by the code below. Since the ALU is only combinational logic, this is not necessary, but it will be a helpful reference once you have sequential elements.
  - The `initial` block sets the clock to 0 at the beginning of the simulation. You should be sure to only change your stimulus when the clock is falling, since the data is captured on the rising edge. Otherwise, it will not only be difficult to debug your design, but it will also cause hold time violations when you run gate level simulation.
  - You must use an always block without a sensitivity list (the @ part of an always statement) to cause the clock to run automatically.

```
parameter Halfcycle = 5; //half period is 5ns
localparam Cycle = 2*Halfcycle;
reg Clock;
// Clock Signal generation:
initial Clock = 0;
always #(Halfcycle) Clock = ~Clock;
```

- `task checkOutput;` - this task contains Verilog code that you would otherwise have to copy paste many times. Note that it is **not** the same thing as a function (as Verilog also has functions).

- `{$random} & 31'h7FFFFFFF` - `$random` generates a pseudorandom 32-bit integer. A bitwise AND will mask the result for smaller bit widths.

For these two modules, the inputs and outputs that you care about are `opcode`, `funct`, `add_rshift_type`, `A`, `B` and `Out`. To test your design thoroughly, you should work through every possible `opcode`, `funct`, and `add_rshift_type` that you care about, and verify that the correct `Out` is generated from the `A` and `B` that you pass in.

The test bench generates random values for `A` and `B` and computes `REFout = A + B`. It also contains calls to `checkOutput` for load and store instructions, for which the ALU should perform addition. It will be up to you to write tests for the remaining combinations of `opcode`, `funct`, and `add_rshift_type` to test your other instructions.

Remember to restrict `A` and `B` to reasonable values (e.g. masking them, or making sure that they are not zero) if necessary to guarantee that a function is sufficiently tested. Please also write tests where the inputs and the output are hard-coded. These should be corner cases that you want to be certain are stressed during testing.

3.4.2 Test Vector Testbench

An alternative way of testing is to use a test vector, which is a series of bit arrays that map to the inputs and outputs of your module. The inputs can be all applied at once if you are testing a combinational logic block or applied over time for a sequential logic block (e.g. an FSM).

You will write a Verilog testbench that takes the parts of the bit array that correspond to the inputs of the module, feeds those to the module, and compares the output of the module with the output bits of the bit array. The bit vector should be formatted as follows:

```
[106:100] = opcode
[99:97] = funct
[96] = add_rshift_type
[95:64] = A
[63:32] = B
[31:0] = REFout
```

Open up the skeleton provided to you in `ALUTestVectorTestbench.v`. You need to complete the module by making use of `$readmemb` to read in the test vector file (named `testvectors.input`), writing some assign statements to assign the parts of the test vectors to registers, and writing a for loop to iterate over the test vectors.

The syntax for a for loop can be found in `ALUTestbench.v`. `$readmemb` takes as its arguments a filename and a reg vector, e.g.:

```
reg [5:0] bar [0:20];
$readmemb(foo.input, bar);
```

3.4.3 Writing Test Vectors

Additionally, you will also have to generate actual test vectors to use in your test bench. A test vector can either be generated in Verilog (like how we generated `A`, `B` using the random number generator and iterated over the possible opcodes and functs), or using a scripting language like python. Since we have

already written a Verilog test bench for our ALU and decoder, we will tackle writing a few test vectors by hand, then use a script to generate test vectors more quickly.

Test vectors are of the format specified above, with the 7 `opcode` bits occupying the left-most bits. Open up the file `tests/testvectors.input` and add test vectors for the following instructions to the end (i.e. manually type the 107 zeros and ones required for each test vector): `SLT`, `SLTU`, `SRA`, and `SRL`.

In the same directory, we've also provided a test vector generator written in Python, which is a popular language used for scripting. We used this generator to generate the test vectors provided to you. If you're curious, you can read the next paragraph and poke around in the file. If not, feel free to skip ahead to the next section.

The script `ALUTestGen.py` is located in `tests`. Run it so that it generates a test vector file in the `tests` folder. Keep in mind that this script makes a couple assumptions that aren't necessary and may differ from your implementation:

- Jump, branch, load and store instructions will use the ALU to compute the target address.
- For all shift instructions, `A` is shifted by `B`. In other words, `B` is the shift amount.
- For the `LUI` instruction, the value to load into the register is fed in through the `B` input.

You can either match these assumptions or modify the script to fit with your implementation. All the methods to generate test vectors are located in the two Python dictionaries `opcodes` and `functs`. The lambda methods contained (separated by commas) are respectively: the function that the operation should perform, a function to restrict the `A` input to a particular range, and a function to restrict the `B` input to a particular range.

If you modify the Python script, run the generator to make new test vectors. This will overwrite the file, so if you want to save your handwritten test vectors, rename the file before running the script, then append them once the file has been generated.

```
% python3 ALUTestGen.py
```

This will write the test vector into the file `testvectors.input`. Use this file as the target test vector file when loading the test vectors with `$readmemb`.

3.5 Writing Verilog Modules

For this exercise, we've provided the module interfaces for you. They are logically divided into a control (`ALUdec.v`) and a datapath (`ALU.v`). The datapath contains the functional units while control contains the necessary logic to drive the datapath. You will be responsible for implementing these two modules. Descriptions of the inputs and outputs of the modules can be found in the first few lines of each file. The ALU should take an `ALUop` and its two inputs `A` and `B`, and provide an output dependent on the `ALUop`. The operations that it needs to support are outlined in the Functional Specification. Don't worry about sign extensions–they should take place outside of the ALU. The ALU decoder uses the `opcode`, `funct`, and `add_rshift_type` to determine the `ALUop` that the ALU should execute. The `funct` input corresponds to the `funct3` field from the ISA encoding table. The `add_rshift_type` input is used to distinguish between `ADD/SUB`, `SRA/SRL`, and `SRAI/SRLI`; you will notice that each of these pairs has the same `opcode` and `funct3`, but differ in the `funct7` field.

You will find the case statement useful, which has the following syntax:

```
always@(*) begin
    case(foo)
        3'b000: // something happens here
        3'b001: // something else happens here
        3'b010, 3'b011: // you can have more than
                        // one case do the same thing
        default: // everything else
    endcase
end
```

To make your job easier, we have provided two Verilog header files: `Opcode.vh` and `ALUop.vh`. They provide, respectively, macros for the opcodes and functs in the ISA and macros for the different ALU operations. You should feel free to change `ALUop.vh` to optimize the `ALUop` encoding, but if you change `Opcode.vh`, you will break the test bench skeleton provided to you. You can use these macros by placing a backtick in front of the macro name, e.g.:

```
case(opcode)
`OPC_STORE:
```

is the equivalent of:

```
case(opcode)
7'b0100011:
```

3.6 Running the Simulation

Open the file `sim-rtl.yml`, set the testbench's name to be ALUTestbench

```
tb_name: &TB_NAME "ALUTestbench"
```

By typing `make sim-rtl` you will run the ALU simulation.

You may change the testbench's name to ALUTestVectorTestbench to use the test vector testbench.

Once you have a working design, you should see the following output when you run either of the given testbenches:

```
# ALL TESTS PASSED!
```

3.7 Viewing Waveforms

As in the previous labs, you should use DVE to view waveforms.

1. List of the modules involved in the test bench. You can select one of these to have its signals show up in the object window.
2. Object window - this lists all the wires and regs in your module. You can add signals to the waveform view by selecting them, right-clicking, and doing Add → To Wave → Selected Signals.
3. Waveform viewer - The signals that you add from the object window show up here. You can navigate the waves by searching for specific values, or going forward or backward one transition at a time.

As an example of how to use the waveform viewer, suppose you get the following output when you run `ALUTestbench`:

```
# FAIL: Incorrect result for opcode 0110011, funct: 101:, add_rshift_type: 1
#       A: 0x92153524, B: 0xffffde81, DUTout: 0x490a9a92, REFout: 0xc90a9a92
```

The `$display()` statement actually already tells you everything you need to know to fix your bug, but you'll find that this is not always the case. For example, if you have an FSM and you need to look at multiple time steps, the waveform viewer presents the data in a much neater format. If your design had more than one clock domain, it would also be nearly impossible to tell what was going on with only `$display()` statements.

Add all the signals from `ALUTestbench` to the waveform viewer and you see the following window: The two highlighted boxes contain the tools for navigation and zoom. You can hover over the icons to find out more about what each of them do. You can find the location (time) in the waveform viewer where the test bench failed by searching for the value of `DUTout` output by the `$display()` statement above (in this case, `0x490a9a92`:

1. Selecting `DUTout`
2. Clicking `Edit > Wave Signal Search > Search for Signal Value > 0x490a9a92`

Now you can examine all the other signal values at this time. Compare the `DUTout` and `REFout` values at this time, and you should see that they are similar but not quite the same. From the `opcode`, `funct`, and `add_rshift_type`, you know that this is supposed to be an SRA instruction, but it looks like your ALU performed a SRL instead. However, you wrote

```
Out = A >>> B[4:0];
```

That looks like it should work, but it doesn't! It turns out you need to tell Verilog to treat `B` as a signed number for SRA to work as you wish. You change the line to say:

```
Out = $signed(A) >>> B[4:0];
```

After making this change, you run the tests again and cross your fingers. Hopefully, you will see the line:

```
# ALL TESTS PASSED!
```

If not, you will need to debug your module until all test from the test vector file and the hard-coded test cases pass.

3.8 Checkpoint #1: Simple test program

Checkoff due: End of 03/16-03/22, 2020

Congratulations! You've started the design of your datapath by drawing a pipeline diagram, and written and thoroughly tested a key component in your processor. You should now be well-versed in testing Verilog modules. Please answer the following questions to be checked off by a TA:

1. Show your pipeline diagram, and explain when writes and reads occur in the register file and memory relative to the pipeline stages.
2. Show your working ALU test bench files to your TA and explain your hard-coded cases. You should also be able to show that the tests for the test vectors generated by the python script and your hard-coded test vectors both work.
3. In ALUTestbench, the inputs to the ALU were generated randomly. When would it be preferable to perform an exhaustive test rather than a random test?
4. What bugs, if any, did your test bench help you catch?
5. For one of your bugs, come up with a short assembly program that would have failed had you not caught the bug. In the event that you had no bugs and wrote perfect code the first time, come up with an assembly program to stress the SRA bug mentioned in the above section.

4 Checkpoint #2: Fully functioning core

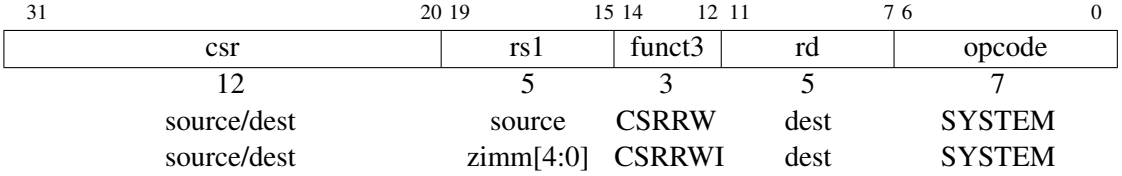
4.1 Additional Instructions

In order to run the testbenches, there are a few new instructions that need to be added for help in debugging/creating testbenches. Read through section 6.2 in the RISC-V specification. A CSR (or control status register) is some state that is stored independent of the register file and the memory. While there are 2^{12} possible CSR addresses, you will only use one of them (`tohost = 0x51E`). The `tohost` register is monitored by the test harness, and simulation ends when a value is written to this register. A value of 1 indicates success, a value greater than 1 gives clues as to the location of the failure.

There are 2 CSR related instructions that you will need to implement:

1. `csrw tohost,t2` (short for `csrrw x0,csr,rs1` where `csr = 0x51E`)
2. `csrwi tohost,1` (short for `csrrwi x0,csr,zimm` where `csr = 0x51E`)

`csrw` will write the value from register in rs1. `csrwi` will write the immediate (stored in rs1) to the addressed csr. Note that you do not need to write to rd (writing to x0 does nothing).



4.2 Details

Your job is to implement the core of the 3-stage RISC-V CPU.

4.3 File Structure

Implement the datapath and control logic for your RISC-V processor in the file `Riscv151.v` that is provided. Make sure that the inputs and outputs remain the same, since this module connects to the memory system for system-level testing. If you look at `riscv_test_harness.v` you can see a testbench that is provided.

4.4 Running the Test

This testbench will load a program into the instruction memory, and will then run until the exit code register has been set. There is also a timeout to make sure that the simulation does not run forever. This will also tell you whether or not your testbench is passing the test. You should only be running this test suite after you have eliminated some of the bugs using single instruction tests described below.

4.5 Running assembly tests

We have provided a suite of assembly tests to help you debug all of the instructions you need to estimate. To run all of them:

```
make sim-rtl test_asm=all
```

This will generate .out files in the `asm_output/` directory, and summarize which tests passed and failed. You can also run single asm test with the following command:

```
make sim-rtl test_asm=rv32ui-p-simple.out
```

If you would like to generate waveforms for a single test:

```
make sim-rtl test_asm=rv32ui-p-simple.vpd
```

, where 'simple' gets replaced with any of the available tests defined in the `Makefile`.

You can read the assembly code of the programs by looking at the dump file. Comments in the code will help you understand what is happening.

```
cd tests/asm/
vim rv32ui-p-addi.dump
```

Last, you can see the hex code that is loaded directly into the memory by looking at the hex file.

```
cd tests/asm/
vim rv32ui-p-addi.hex
```

4.6 Checkpoint #2 Deliverables

Checkoff due: End of 04/06-04/12, 2020

Congratulations! You've started the design of your datapath by implementing your pipeline diagram, and written and thoroughly tested a key component in your processor and should now be well-versed in testing Verilog modules. Please answer the following questions to be checked off by a TA.

1. Show that all of the assembly tests pass
2. Show your final pipeline diagram, updated to match the code.

5 Checkpoint #3: CPU + Cache

A processor operates on data in memory. Memory can hold billions of bits, which can either be instructions or data. In a VLSI design, it is a *very bad idea* to store this many bits close to the processor. The chip area required would be *huge* - consider how many DRAM chips your PC has, and that DRAM cells are much smaller than SRAM cells (which can actually be implemented in the same CMOS process). Moreover, the entire processor would have to slow down to accommodate delays in the large memory array. Instead, caches are used to create the illusion of a large memory with low latency.

Your task is to implement a (relatively) simple cache for your RISC-V processor, based on some predefined SRAM macros (memory arrays) and the interface specified below.

5.1 Cache overview

When you request data at a given address, the cache will see if it is stored locally. If it is (cache hit), it is returned immediately. Otherwise if it is not found (cache miss), the cache fetches the bits from the main memory.

Caches store data in "ways." A way is a logical element which contains valid bits, tag bits, and data. The simplest type of cache is direct-mapped (a 1-way cache). A cache stores data in larger units (lines) than single words. In each way, a given address may only occupy a single location, determined by the lowest bits of the cache line address. The remaining address bits are called the "tag" and are stored so that we can check if a given cache line belongs to a given address. The valid bit indicates which lines contain valid data.

Multi-way caches allow more flexibility in what data is stored in the cache, since there are multiple locations for a line to occupy (the number of ways). For this reason, a "replacement policy" is needed. This is used to decide which way's data to evict when fetching new data. For this project you may use any policy you wish, but pseudo-random is recommended.

5.2 Guidelines and requirements

You have been given the interface of a cache (`Cache.v`) and your next task is to implement the cache. **EECS151 students** should build a direct-mapped cache, and **EECS251 students** are required to implement a cache that either:

1. is configurable to be either direct-mapped or at least 2-way set associative; or

2. is set-associative with configurable associativity.

You are welcome to implement a more performant cache if you desire.

Your cache should be at least 512 bytes; if you wish to increase the size, implement the 512 bytes cache first and upgrade later.

Use the SRAMs that are available in
`/home/ff/eecs151/hammer/src/hammer-vlsi/technology/asap7/sram_compiler/memories/behavioral/sram_behav_models.v`
for your data and tag arrays.

The pin descriptions for these SRAMs are as follows:

`A`	address
`CE`	clock edge
`OEB`	output enable bar (tie this to 0)
`WEB`	write enable bar (1 is a read, 0 is a write)
`CSB`	chip select bar (tie this to 0)
`BYTEMASK`	write byte mask
`I`	write data
`O`	read data

You should use cache lines that are 512 bits (16 words) for this project. The memory interface is 128 bits, meaning that you will require multiple (4) cycles to perform memory transactions.

Below find a description of each signal in `Cache.v`:

`clk`	clock
`reset`	reset
`cpu_req_valid`	The CPU is requesting a memory transaction
`cpu_req_ready`	The cache is ready for a CPU memory transaction
`cpu_req_addr`	The address of the CPU memory transaction
`cpu_req_data`	The write data for a CPU memory write (ignored on reads)
`cpu_req_write`	The 4-bit write mask for a CPU memory transaction (each bit corresponds to the byte address within the word). `4'b0000` indicates a read.
`cpu_resp_valid`	The cache has output valid data to the CPU after a memory read
`cpu_resp_data`	The data requested by the CPU
`mem_req_valid`	The cache is requesting a memory transaction to main memory
`mem_req_ready`	Main memory is ready for the cache to provide a memory address
`mem_req_addr`	The address of the main memory transaction from the cache. Note that this address is narrower than the CPU byte address since main memory has wider data.
`mem_req_rw`	1 if the main memory transaction is a write; 0 for a read.
`mem_req_data_valid`	The cache is providing write data to main memory.
`mem_req_data_ready`	Main memory is ready for the cache to provide write data.
`mem_req_data_bits`	Data to write to main memory from the cache (128 bits/4 words).
`mem_req_data_mask`	Byte-level write mask to main memory. May be 16'hFFFF for a full write.
`mem_resp_valid`	The main memory response data is valid.
`mem_resp_data`	Main memory response data to the cache (128 bits/4 words).

To design your cache, start by outlining where the SRAMs should go. You should include an SRAM per way for data, and a separate SRAM per way for the tags. Depending on your implementation, you may want to implement the valid bits in flip flops or as part of the tag SRAM.

Next you should develop a state machine that covers all the events that your cache needs to handle for both hits and misses. You can do it without an explicit state machine, but you will suffer. Keep in mind you will need to write any valid data back to main memory before you start refilling the cache (you can use a write-back or a write-through policy). Both of these transactions will take multiple cycles.

5.3 Changes to the flow for this checkpoint

You should now be able to pass the `bmark` test suite. The test suite includes many C programs that do various things to test your processor and cache implementation. You can observe the number of cycles that each `bmark` test takes to run by opening `bmark_output/*.out` and taking note of the number on the last line. The `make sim-rtl test_bmark=all` target will also print this number for you.

To run a specific benchmark (e.g., cachetest), run

```
make sim-rtl test_bmark=cachetest.out
```

After completing your cache, run the tests with both the cache included and with the fake memory (`no_cache_mem`) included. To use `no_cache_mem` be sure to have `+define+no_cache_mem` in the `simOptions` variable in the `sim-rtl.yml` file (line 16- this is the default). To use your cache, uncomment `+define+no_cache_mem`. Take note of the cycle counts for both- you should see the cycle counts increase when you use the cache.

5.4 Checkpoint #3 Deliverables

Checkoff due: End of 04/20-04/26, 2020

1. Show that all of the assembly tests and bmark tests pass using the cache
2. Show the block diagram of your cache
3. What was the difference in the cycle count for the final test with the perfect memory and the cache?
4. Show your final pipeline diagram, updated to match the code

6 Back-end Design (Phase 2)

7 Checkpoint #4: Synthesis & PAR

Available Wednesday, April 8, 2020. *Checkoff due: Friday, May 8, 2020.*