EECS151/251A
Fall 2022
Final Project Specification

RISCV151

Version 1.22

TA: Yikuan Chen, Simon Guo, Jennifer Zhou, Paul Kwon, Ella Schwarz, Raghav Gupta

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

Contents

1 Introduction

The goal of this project is to familiarize EECS151/251A students with the methods and tools of digital design. Working alone or in a team of two, you will design and implement a 3-stage pipelined RISC-V CPU with a UART for tethering and a simple branch predictor.

Finally, you will optimize your CPU for performance (maximizing the Iron Law) and cost (FPGA resource utilization).

You will use Verilog to implement this system, targeting the Xilinx PYNQ platform (a PYNQ-Z1 development board with a Zynq 7000-series FPGA). The project will give you experience designing with RTL descriptions, resolving hazards in a simple pipeline, building interfaces, and teach you how to approach system-level optimization.

In tackling these challenges, your first step will be to map the high level specification to a design which can be translated into a hardware implementation. After that, you will produce and debug that implementation. These first steps can take significant time if you have not thought out your design prior to trying implementation.

As in previous semesters, your EECS151/251A project is probably the largest project you have faced so far here at Berkeley. Good time management and good design organization is critical to your success.

1.1 Tentative Deadlines for All Sections

The following is a brief description of each checkpoint and approximately how many weeks will be alloted to each one. Note that this schedule is tentative and is subjected to change as the semester progresses.

- **Nov 4, 2022 - Checkpoint 1 (2 weeks)** - Draw a schematic of your processor's datapath and pipeline stages, and provide a brief write-up of your answers to the questions in 2.12.1. In addition, push all of your IO-circuit Verilog modules that you have implemented in the labs to your assigned GitHub repository under `hardware/src/io_circuits` (see 2.2). Also commit your design documents (block diagram + write-up) to `docs`.
- **Nov 18, 2022 - Checkpoint 2 (2 weeks)** - Implement a fully functional RISC-V processor core in Verilog. Your processor core should be able to run the **mmult** demo successfully.
- **Dec 02, 2021 - Checkpoint 3 (2 weeks)** - Implement a branch predictor in Verilog.
- **Dec 09, 2022 - Final Checkoff + Demo** - Final processor optimization and checkoff
- **Dec 12, 2022 - Project Report** - Final report due.

1.2 General Project Tips

Document your project as you go. You should comment your Verilog and 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. **If your submitted project does not work by the final deadline, you will not get any credit for any extra credit features you have implemented.**

This project, as has been done in past semesters, will be divided into checkpoints. The following sections will specify the objectives for each checkpoint.

2 Checkpoints 1 & 2 - Three-stage Pipelined RISC-V CPU

The first checkpoint 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 in subsequent checkpoints.

2.1 Setting up your Code Repository

The project skeleton files are available on GitHub. Your (private) project repo will be created by GSIs and assigned to your group. Its name will be in the format of **"fa22_fpga_teamXX.git"**. The suggested way for initializing your repository with the skeleton files is as follows:

```
git clone https://github.com/EECS150/fpga_project_skeleton_fa22
cd fpga_project_skeleton_fa22
git submodule init
git submodule update
git remote add my_repo_name https://github.com/EECS150/fa22_fpga_teamXX
git push my_repo_name master
```

Then reclone your repo and add the skeleton repo as a remote:

```
cd ..
rm -rf fpga_project_skeleton_fa22
git clone https://github.com/EECS150/fa22_fpga_teamXX
cd fa22_fpga_teamXX
git remote add staff https://github.com/EECS150/fpga_project_skeleton_fa22
```

Note: The above instructions are for HTTPS authentication. If you are running into HTTPS authentication errors, you can use SSH authentication by replacing the above Git repo URLs with the following:

```
git@github.com:EECS150/fpga_project_skeleton_fa22.git
git@github.com:EECS150/fa22_fpga_teamXX.git
```

To pull project updates from the skeleton repo, run `git pull staff master`.

To get a team repo, fill the Google form with your team information (names, GitHub logins). Only one person in a team is required to fill the form.

You should check frequently for updates to the skeleton files. Whenever you resume your work on the project, it is highly suggested that you do git pull from the skeleton repo to get the latest update. Update announcements will be posted to Piazza.

2.2 Integrate Designs from Labs

You should copy some modules you designed from the labs. We suggest you keep these with the provided source files in `hardware/src/io_circuits` (overwriting any provided skeletons).

Copy these files from the labs:

```
debouncer.v
synchronizer.v
```

```
edge_detector.v
fifo.v
uart_transmitter.v
```

2.3   Project Skeleton Overview

- hardware

  - src

    * z1top.v: Top level module. The RISC-V CPU is instantiated here.

    * riscv_core/cpu.v: All of your CPU datapath and control should be contained in this file.

    * io_circuits: Your IO circuits from previous lab exercises.

    * riscv_core/opcode.vh: Constant definitions for various RISC-V opcodes and funct codes.

  - sim

    * cpu_tb.v: Starting point for testing your CPU. The testbench checks if your CPU can execute all the RV32I instructions (including CSR ones) correctly, and can handle some simple hazards. You should make sure that your CPU implementation passes this testbench before moving on.

    * asm_tb.v: The testbench works with the software in software/assembly_tests.

    * isa_tb.v: The testbench works with the RISC-V ISA test suite in software/riscv-isa-tests. The testbench only runs one test at a time. To run multiple tests, use the script we provide. There is a total of 38 ISA tests in the test suite.

    * c_tests_tb.v: This testbench verifies the correct execution of the software in software/c_tests. There are 6 C tests provided.

    * echo_tb.v: The testbench works with the software in software/echo. The CPU reads a character sent from the serial rx line and echoes it back to the serial tx line.

    * uart_parse_tb.v: This testbench verifies a few tricky functions from the BIOS in isolation using the software in software/uart_parse.

    * bios_testbench.v: This testbench simulates the execution of the BIOS program. It checks if your CPU can execute the instructions stored in the BIOS memory. The testbench also emulates user input sent over the serial rx line, and checks the BIOS message output obtained from the serial tx line.

- software

  - bios: The BIOS program, which allows us to interact with our CPU via the UART. You need to compile it before creating a bitstream or running a simulation.

  - echo: The echo program, which emulates the echo test of Lab 5 in software.

- **asm**: Use this as a template to write assembly tests for your processor designed to run in simulation.

- **c_tests**: Use these as examples to write C programs for testing.

- **riscv-isa-tests**: A comprehensive test suite for your CPU. Available after doing **git submodule** (see **??**).

- **mmult**: This is a program to be run on the FPGA for Checkpoint 2. It generates 2 matrices and multiplies them. Then it returns a checksum to verify the correct result.

To compile **software** go into a program directory and run **make**. To build a bitstream run **make write-bitstream** in **hardware**.

2.4 RISC-V 151 ISA

Table 1 contains all of the instructions your processor is responsible for supporting. It contains most of the instructions specified in the RV32I Base Instruction set, and allows us to maintain a relatively simple design while still being able to have a C compiler and write interesting programs to run on the processor. For the specific details of each instruction, refer to sections 2.2 through 2.6 in the RISC-V Instruction Set Manual.

2.4.1 CSR Instructions

You will have to implement 2 CSR instructions to support running the standard RISC-V ISA test suite. 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 RISC-V ISA testbench (**isa_testbench.v**), and simulation ends when a non-zero value is written to this register. A CSR value of 1 indicates success, and a value greater than 1 indicates which test failed.

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

1. **csrw tohost,x2** (short for **csrrw x0,csr,rs1** where **csr** = 0x51E)

2. **csrwi tohost,1** (short for **csrrwi x0,csr,uimm** where **csr** = 0x51E)

csrw will write the value from **rs1** into the addressed CSR. **csrwi** will write the immediate (stored in the rs1 field in the instruction) into the addressed CSR. Note that you do not need to write to **rd** (writing to x0 does nothing), since the CSR instructions are only used in simulation.

2.5 Pipelining

Your CPU must implement this instruction set using a 3-stage pipeline. The division of the datapath into three stages is left unspecified as it is an important design decision with significant performance implications. We recommend that you begin the design process by considering which elements of the datapath are synchronous and in what order they need to be placed. After determining the design blocks that require a clock edge, consider where to place asynchronous blocks to minimize the critical path. The RAMs we are using for the data, instruction, and BIOS memories are both **synchronous** read and **synchronous** write.

Table 1: RISC-V ISA

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

RV32I Base Instruction Set

imm[31:12]			rd	0110111	LUI
imm[31:12]			rd	0010111	AUIPC
imm[20|10:1|11|19:12]			rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
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
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
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
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

RV32/RV64 *Zicsr* Standard Extension

csr	rs1	001	rd	1110011	CSRRW
csr	uimm	101	rd	1110011	CSRRWI

8

2.6 Hazards

As you have learned in lecture, pipelines create hazards. Your design will have to resolve both control and data hazards. You must resolve data hazards by implementing forwarding whenever possible. This means that you must forward data from your data memory instead of stalling your pipeline or injecting NOPs. All data hazards can be resolved by forwarding in a three-stage pipeline.

You'll have to deal with the following types of hazards:

1. **Read-after-write data hazards** Consider carefully how to handle instructions that depend on a preceding load instruction, as well as those that depend on a previous arithmetic instruction.

2. **Control hazards** What do you do when you encounter a branch instruction, a jal (jump and link), or jalr (jump from register and link)? You will have to choose whether to predict branches as taken or not taken by default and kill instructions that weren't supposed to execute if needed. You can begin by resolving branches by stalling the pipeline, and when your processor is functional, move to naive branch prediction.

2.7 Register File

We have provided a register file module for you in `EECS151.v`: `ASYNC_RAM_1W2R`. The register file has two asynchronous-read ports and one synchronous-write port (positive edge). In addition, you should ensure that register 0 is not writable in your own logic, i.e. reading from register 0 always returns 0.

2.8 RAMs

In this project, we will be using some memory blocks defined in `EECS151.v` to implement memories for the processor. As you may recall in previous lab exercises, the memory blocks can be either synthesized to Block RAMs or LUTRAMs on FPGA. For the project, our memory blocks will be mapped to Block RAMs. Therefore, read and write to memory are **synchronous**.

2.8.1 Initialization

For synthesis, the BIOS memory is initialized with the contents of the BIOS program, and the other memories are zeroed out.

For simulation, the provided testbenches initialize the BIOS memory with a program specified by the testbench (see `sim/assembly_testbench.v`).

2.8.2 Endianness + Addressing

The instruction and data RAMs have 16384 32-bit rows, as such, they accept 14 bit addresses. The RAMs are **word-addressed**; this means that every unique 14 bit address refers to one 32-bit row (word) of memory.

However, the memory addressing scheme of RISC-V is **byte-addressed**. This means that every unique 32 bit address the processor computes (in the ALU) points to one 8-bit byte of memory.

We consider the bottom 16 bits of the computed address (from the ALU) when accessing the RAMs. The top 14 bits are the word address (for indexing into one row of the block RAM), and the bottom two are the byte offset (for indexing to a particular byte in a 32 bit row).

Figure 1: Block RAM organization. The labels for row address **should read 14'h0 and 14'h1.**

Figure 1 illustrates the 14-bit word addresses and the two bit byte offsets. Observe that the RAM organization is **little-endian**, i.e. the most significant byte is at the most significant memory address (offset '11').

2.8.3 Reading from RAMs

Since the RAMs have 32-bit rows, you can only read data out of the RAM 32-bits at a time. This is an issue when executing an `lh` or `lb` instruction, as there is no way to indicate which 8 or 16 of the 32 bits you want to read out.

Therefore, you will have to shift and mask the output of the RAM to select the appropriate portion of the 32-bits you read out. For example, if you want to execute a `lbu` on a byte address ending in `2'b10`, you will only want bits `[23:16]` of the 32 bits that you read out of the RAM (thus storing `{24'b0, output[23:16]}` to a register).

2.8.4 Writing to RAMs

To take care of `sb` and `sh`, note that the `we` input to the instruction and data memories is 4 bits wide. These 4 bits are a byte mask telling the RAM which of the 4 bytes to actually write to. If `we={4'b1111}`, then all 32 bits passed into the RAM would be written to the address given.

Here's an example of storing a single byte:

- Write the byte `0xa4` to address `0x10000002` (byte offset = 2)
- Set `we = {4'b0100}`
- Set `din = {32'hxx_a4_xx_xx}` (x means don't care)

2.9 Memory Architecture

The standard RISC pipeline is usually depicted with separate instruction and data memories. Although this is an intuitive representation, it does not let us modify the instruction memory to run new programs. Your CPU, by the end of this checkpoint, will be able to receive compiled RISC-V binaries though the UART, store them into instruction memory, then jump to the downloaded program. To facilitate this, we will adopt a modified memory architecture shown in Figure 4.
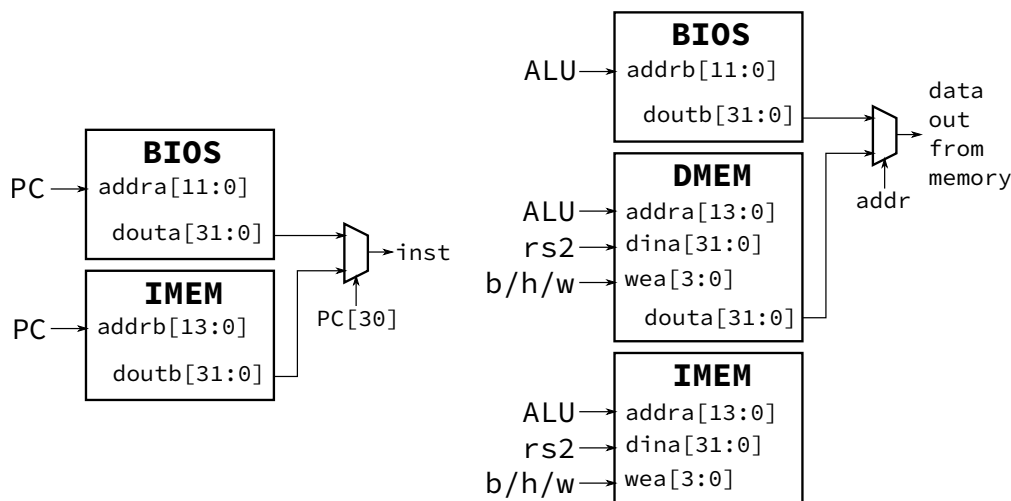

Figure 2: The Riscv151 memory architecture. There is only 1 IMEM and DMEM instance in Riscv151 but their ports are shown separately in this figure for clarity. The left half of the figure shows the instruction fetch logic and the right half shows the memory load/store logic.

2.9.1 Summary of Memory Access Patterns

The memory architecture will consist of three RAMs (instruction, data, and BIOS). The RAMs are memory resources (block RAMs) contained within the FPGA chip, and no external (off-chip, DRAM) memory will be used for this project.

The processor will begin execution from the BIOS memory, which will be initialized with the BIOS program (in `software/bios`). The BIOS program should be able to read from the BIOS memory (to fetch static data and instructions), and read and write the instruction and data memories. This allows the BIOS program to receive user programs over the UART from the host PC and load them into instruction memory.

You can then instruct the BIOS program to jump to an instruction memory address, which begins execution of the program that you loaded. At any time, you can press the reset button on the board to return your processor to the BIOS program.

2.9.2 Unaligned Memory Accesses

In the official RISC-V specification, unaligned loads and stores are supported. However, in your project, you can ignore instructions that request an unaligned access. Assume that the compiler will never generate unaligned accesses.

2.9.3 Address Space Partitioning

Your CPU will need to be able to access multiple sources for data as well as control the destination of store instructions. In order to do this, we will partition the 32-bit address space into four regions: data memory read and writes, instruction memory writes, BIOS memory reads, and memory-mapped I/O. This will be encoded in the top nibble (4 bits) of the memory address generated in load and store operations, as shown in Table 2. In other words, the target memory/device of a load or store instruction is dependent on the address. The reset signal should reset the PC to the value defined by the parameter `RESET_PC` which is by default the base of BIOS memory (`0x40000000`).

Table 2: Memory Address Partitions

Address[31:28]	Address Type	Device	Access	Notes
4'b00x1	Data	Data Memory	Read/Write	
4'b0001	PC	Instruction Memory	Read-only	
4'b001x	Data	Instruction Memory	Write-Only	Only if PC[30] == 1'b1
4'b0100	PC	BIOS Memory	Read-only	
4'b0100	Data	BIOS Memory	Read-only	
4'b1000	Data	I/O	Read/Write	

Each partition specified in Table 2 should be enabled based on its associated bit in the address encoding. This allows operations to be applied to multiple devices simultaneously, which will be used to maintain memory consistency between the data and instruction memory.

For example, a store to an address beginning with `0x3` will write to both the instruction memory and data memory, while storing to addresses beginning with `0x2` or `0x1` will write to only the instruction or data memory, respectively. For details about the BIOS and how to run programs on your CPU, see Section **??**.

Please note that a given address could refer to a different memory depending on which address type it is. For example the address `0x10000000` refers to the data memory when it is a data address while a program counter value of `0x10000000` refers to the instruction memory.

The note in the table above (referencing PC[30]), specifies that you can only write to instruction memory if you are currently executing in BIOS memory. This prevents programs from being self-modifying, which would drastically complicate your processor.

2.9.4 Memory Mapped I/O

At this stage in the project the only way to interact with your CPU is through the UART. The UART from Lab 5 accomplishes the low-level task of sending and receiving bits from the serial lines, but you will need a way for your CPU to send and receive bytes to and from the UART. To accomplish this, we will use memory-mapped I/O, a technique in which registers of I/O devices are assigned memory addresses. This enables load and store instructions to access the I/O devices as if they were memory.

To determine CPI (cycles per instruction) for a given program, the I/O memory map is also used to include instruction and cycle counters.

Table 4 shows the memory map for this stage of the project.

Table 3: I/O Memory Map

Address	Function	Access	Data Encoding
32'h80000000	UART control	Read	{30'b0, uart_rx_data_out_valid, uart_tx_data_in_ready}
32'h80000004	UART receiver data	Read	{24'b0, uart_rx_data_out}
32'h80000008	UART transmitter data	Write	{24'b0, uart_tx_data_in}
32'h80000010	Cycle counter	Read	Clock cycles elapsed
32'h80000014	Instruction counter	Read	Number of instructions executed
32'h80000018	Reset counters to 0	Write	N/A
32'h8000001c	Total branch instruction counter	Read	Number of branch instructions encounted (Checkpoint 3)
32'h80000020	Correct branch prediction counter	Read	Number of branches successfully predicted (Checkpoint 3)

You will need to determine how to translate the memory map into the proper ready-valid handshake signals for the UART. Your UART should respond to `sw, sh, and sb` for the transmitter data address, and should also respond to `lw, lh, lb, lhu, and lbu` for the receiver data and control addresses.

You should treat I/O such as the UART just as you would treat the data memory. This means that you should assert the equivalent write enable (i.e. valid) and data signals at the end of the execute stage, and read in data during the memory stage. The CPU itself should not check the `uart_rx_data_out_valid` and `uart_tx_data_in_ready` signals; this check is handled in software. The CPU needs to drive `uart_rx_data_out_ready` and `uart_tx_data_in_valid` correctly.

The cycle counter should be incremented every cycle, and the instruction counter should be incremented for every instruction that is committed (you should not count bubbles injected into the pipeline or instructions run during a branch mispredict). From these counts, the CPI of the processor can be determined for a given benchmark program.

2.10   Testing

The design specified for this project is a complex system and debugging can be very difficult without tests that increase visibility of certain areas of the design. In assigning partial credit at the end for incomplete projects, we will look at testing as an indicator of progress. A reasonable order in which to complete your testing is as follows:

1. Test that your modules work in isolation via Verilog testbenches that you write yourself

2. Test that your CPU pipeline works with the `sim/cpu_tb.v`

3. Test the entire CPU one instruction at a time with hand-written assembly — see `sim/asm_tb.v`

4. Run the `riscv-tests` ISA test suite (`make isa-tests`)

5. Some extra tests with other software C program, such as `c_tests` and `uart_parse`. They could help reveal more bugs – see `c_tests_tb.v` and `uart_parse_tb.v`

6. Test the CPU's memory mapped I/O — see `echo_tb.v`

7. Test the CPU's memory mapped I/O with BIOS software program — see `bios_tb.v`

For more information on testing, please see the README at `hardware/README.md`.

2.11 How to Succeed in This Checkpoint

Start early and work on your design incrementally. Draw up a very detailed and organised block diagram and keep it up to date as you begin writing Verilog. Unit test independent modules such as the control unit, ALU, and regfile. Write thorough and complex assembly tests by hand, and don't solely rely on the RISC-V ISA test suite. The final BIOS program is several 1000 lines of assembly and will be nearly impossible to debug by just looking at the waveform.

The most valuable asset for this checkpoint will not be your GSIs but will be your fellow peers who you can compare notes with and discuss design aspects with in detail. However, do NOT under any circumstances share source code.

Once you're tired, go home and *sleep*. When you come back you will know how to solve your problem.

2.11.1 How to Get Started

It might seem overwhelming to implement all the functionality that your processor must support. The best way to implement your processor is in small increments, checking the correctness of your processor at each step along the way. Here is a guide that should help you plan out Checkpoint 1 and 2:

1. *Design.* You should start with a comprehensive and detailed design/schematic. Enumerate all the control signals that you will need. Be careful when designing the memory fetch stage since all the memories we use (BIOS, instruction, data, IO) are synchronous.

2. *First steps.* Implementing some modules that are easy to write and test.

3. *Control Unit + other small modules.* Implement the control unit, ALU, and any other small independent modules. Unit test them.

4. *Memory.* In the beginning, only use the BIOS memory in the instruction fetch stage and only use the data memory in the memory stage. This is enough to run assembly tests.

5. *Connect stages and pipeline.* Connect your modules together and pipeline them. At this point, you should be able to run integration tests using assembly tests for most R and I type instructions.

6. *Implement handling of control hazards.* Insert bubbles into your pipeline to resolve control hazards associated with JAL, JALR, and branch instructions. Don't worry about data hazard handling for now. Test that control instructions work properly with assembly tests.

7. *Implement data forwarding for data hazards.* Add forwarding muxes and forward the outputs of the ALU and memory stage. Remember that you might have to forward to ALU input A, ALU input B, and data to write to memory. Test forwarding aggressively; most of your bugs will come from incomplete or faulty forwarding logic. Test forwarding from memory and from the ALU, and with control instructions.

8. *Add BIOS memory reads.* Add the BIOS memory block RAM to the memory stage to be able to load data from the BIOS memory. Write assembly tests that contain some static data stored in the BIOS memory and verify that you can read that data.

9. *Add Inst memory writes and reads.* Add the instruction memory block RAM to the memory stage to be able to write data to it when executing inside the BIOS memory. Also add the instruction memory block RAM to the instruction fetch stage to be able to read instructions from the inst memory. Write tests that first write instructions to the instruction memory, and then jump (using jalr) to instruction memory to see that the right instructions are executed.

10. *Run Riscv151_testbench.* The testbench verifies if your Riscv151 is able to read the RV32I instructions from instruction memory block RAM, execute, and write data to either the Register File or data memory block RAM.

11. *Run isa_testbench.* The testbench works with the RISCV ISA tests. This comprehensive test suites verifies the functionality of your processor.

12. *Run software_testbench.* The testbench works with the software programs under `software` using the CSR check mechanism as similar to the `isa_testbench`. Try testing with all the supported software programs since they could expose more hazard bugs.

13. *Add instruction and cycle counters.* Begin to add the memory mapped IO components, by first adding the cycle and instruction counters. These are just 2 32-bit registers that your CPU should update on every cycle and every instruction respectively. Write tests to verify that your counters can be reset with a `sw` instruction, and can be read from using a `lw` instruction.

14. *Integrate UART.* Add the UART to the memory stage, in parallel with the data, instruction, and BIOS memories. Detect when an instruction is accessing the UART and route the data to the UART accordingly. Make sure that you are setting the UART ready/valid control signals properly as you are feeding or retrieving data from it. We have provided you with the `echo_testbench` which performs a test of the UART. In addition, also test with `c_testbench` and `bios_testbench`.

15. *Run the BIOS.* If everything so far has gone well, program the FPGA. Verify that the BIOS performs as expected. As a precursor to this step, you might try to build a bitstream with the BIOS memory initialized with the echo program.

16. *Run matrix multiply.* Load the `mmult` program with the `hex_to_serial` utility (located under `scripts/`), and run `mmult` on the FPGA. Verify that it returns the correct checksum.

17. *Check CPI.* Compute the CPI when running the `mmult` program. If you achieve a CPI 1.2 or smaller, that is acceptable, but if your CPI is larger than that, you should think of ways to reduce it.

2.12   Checkoff

The checkoff is divided into two stages: block diagram/design and implementation. The second part will require significantly more time and effort than the first one. As such, completing the block diagram in time for the design review is crucial to your success in this project.

2.12.1 Checkpoint 1

Block Diagram

The first checkpoint requires a detailed block diagram of your datapath. The diagram should have a greater level of detail than a high level RISC datapath diagram. You may complete this electronically or by hand.

If working by hand, we recommend working in pencil and combining several sheets of paper for a larger workspace. If doing it electronically, you can use Inkscape, Google Drawings, draw.io or any program you want.

You should be able to describe in detail any smaller sub-blocks in your diagram. **Though the diagrams from textbooks/lecture notes are a decent starting place, remember that they often use asynchronous-read RAMs for the instruction and data memories, and we will be using synchronous-read block RAMs.**

Additionally, you will be asked to provide short answers to the following questions based on how you structure your block diagram. The questions are intended to make you consider all possible cases that might happen when your processor execute instructions, such as data or control hazards. It might be a good idea to take a moment to think of the questions first, then draw your diagram to address them.

Questions

1. How many stages is the datapath you've drawn? (i.e. How many cycles does it take to execute 1 instruction?)

2. How do you handle ALU → ALU hazards?

   addi x1, x2, 100

   addi x2, x1, 100

3. How do you handle ALU → MEM hazards?

   addi x1, x2, 100

   sw x1, 0(x3)

4. How do you handle MEM → ALU hazards?

   lw x1, 0(x3)

   addi x1, x1, 100

5. How do you handle MEM → MEM hazards?

   lw x1, 0(x2)

   sw x1, 4(x2)

   also consider:

   lw x1, 0(x2)

   sw x3, 0(x1)

6. Do you need special handling for 2 cycle apart hazards?

   addi x1, x2, 100

   nop

   addi x1, x1, 100

7. How do you handle branch control hazards? (What is the mispredict latency, what prediction scheme are you using, are you just injecting NOPs until the branch is resolved, what about data hazards in the branch?)

8. How do you handle jump control hazards? Consider jal and jalr separately. What optimizations can be made to special-case handle jal?

9. What is the most likely critical path in your design?

10. Where do the UART modules, instruction, and cycle counters go? How are you going to drive `uart_tx_data_in_valid` and `uart_rx_data_out_ready` (give logic expressions)?

11. What is the role of the CSR register? Where does it go?

12. When do we read from BIOS for instructions? When do we read from IMem for instructions? How do we switch from BIOS address space to IMem address space? In which case can we write to IMem, and why do we need to write to IMem? How do we know if a memory instruction is intended for DMem or any IO device?

Commit your block diagram and your writeup to your team repository under `fa22_fpga_teamXX/docs` by Nov 4, 2022. Please also remember to push your working IO circuits to your GitHub repository.

2.12.2 Checkpoint 2: Base RISCV151 System

This checkpoint requires a fully functioning three stage RISC-V CPU as described in this specification. Checkoff will consist of a demonstration of the BIOS functionality, loading a program (`echo` and `mmult`) over the UART, and successfully jumping to and executing the program.

Additionally, please find the maximum achievable frequency of your CPU implementation. To do so, lower the `CPU_CLOCK_PERIOD` (starting at 20, with a step size of 1) in `hardware/src/z1top.v` until the Implementation fails to meet timing. Please report the critical path in your implementation.

Checkpoint 2 materials should be committed to your project repository by Nov 18, 2022.

2.12.3 Checkpoints 1 & 2 Deliverables Summary

Deliverable	Due Date (for all sections)	Description
Block Diagram, RISC-V ISA Questions, IO code	Nov 4, 2022	Push your block diagram, your write-up, and IO code to your GitHub repository. In-lab Checkoff: Sit down with a GSI and go over your design in detail.
RISC-V CPU, Fmax and Crit. path	Nov 18, 2022	Check in code to GitHub. In-lab Checkoff: Demonstrate that the BIOS works, you can use `hex_to_serial` to load the `echo` program, `jal` to it from the BIOS, and have that program successfully execute. Load the mmult program with `hex_to_serial`, `jal` to it, and have it execute successfully and return the benchmarking results and correct checksum. Your CPI should not be greater than 1.2

3   Checkpoint 3 - Branch Predictor

In our current datapath design, we handle data hazards by forwarding path, but we do not handle control hazards well. This results in high CPI as our naive predictor (always predict non-taken) often mispredict and we need to stall or flush the pipeline upon such misses.

In Checkpoint 3, we want to explore the idea of a **Branch Predictor**. This particular predictor should predict the direction of branch (whether it is taken or not), not the address of branch target. There are many ways to implement branch predictors, but for consistency of this checkpoint please follow the scheme we describe below as you need to pass our testbench. You can build a better and more sophisticated branch predictor in Checkpoint 4 Optimization.

3.1   Branch History Table Overview

Branch History Table (BHT) or Branch-Prediction Buffer (BPB) is a form of dynamic branch prediction, which allows our prediction to adapt to program behavior.

To do so, we need to build the branch predictor that can

- **Guess:** When the branch instruction is in the first stage of processor, make a prediction whether to take the branch based on the past history.

- **Check:** When the branch instruction reaches the second stage of processor (where branch result is resolved), check if your prediction is correct and update to make better prediction next time.

One way to build such a system is by building a **cache** whose entries that is represented by a **saturating counter**.

2-bit Saturating Counter is a state machine that correspond to a particular branch instruction (based on its address) to show whether the branch was recently taken or not. We can use that information to make a prediction as past is usually a good indicator of the future. We increment the counter on branch taken and decrement it on branch not taken. Instead of a single bit that represents taken or not taken recently, we make it more robust by using two bits and less sensitive to anomaly branch behaviors. We can take the top bit as the prediction. For example, if the branch behavior is mostly not taken and the state is currently in strong not taken, even the branch is taken once, it will still be weakly not taken and prediction remains not taken.
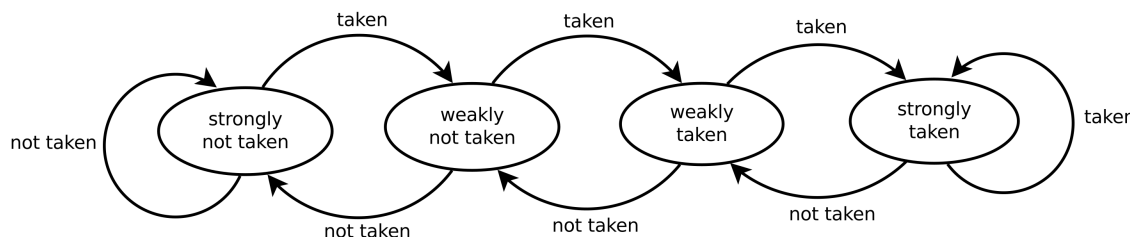

Figure 3: State Machine for 2-bit Saturating Counter.

19

BHT / BPB as a Cache. The branch history table and branch prediction table can be thought of a cache or a buffer. We map each branch instruction by the lower portion of the address into entries of the cache (index into cache by lower address bits). Each cache line contains the saturating counter value that represents history of the instruction. During the guess stage, we check if the cache contains entries of this branch instruction by checking the tag and valid bit. If it is a hit, we read the entry and use the top bit as prediction.

During the check stage, we read the entry of the cache again to see if behavior of corresponding branch instruction has been recorded. If the entry exists (cache hit) we read and update the current saturating counter based on whether the branch is taken or not, and write back to the cache with updated saturating counter value. If the entry did not exist before, we write a new line in the cache with saturating counter value based on whether the branch was taken or not.
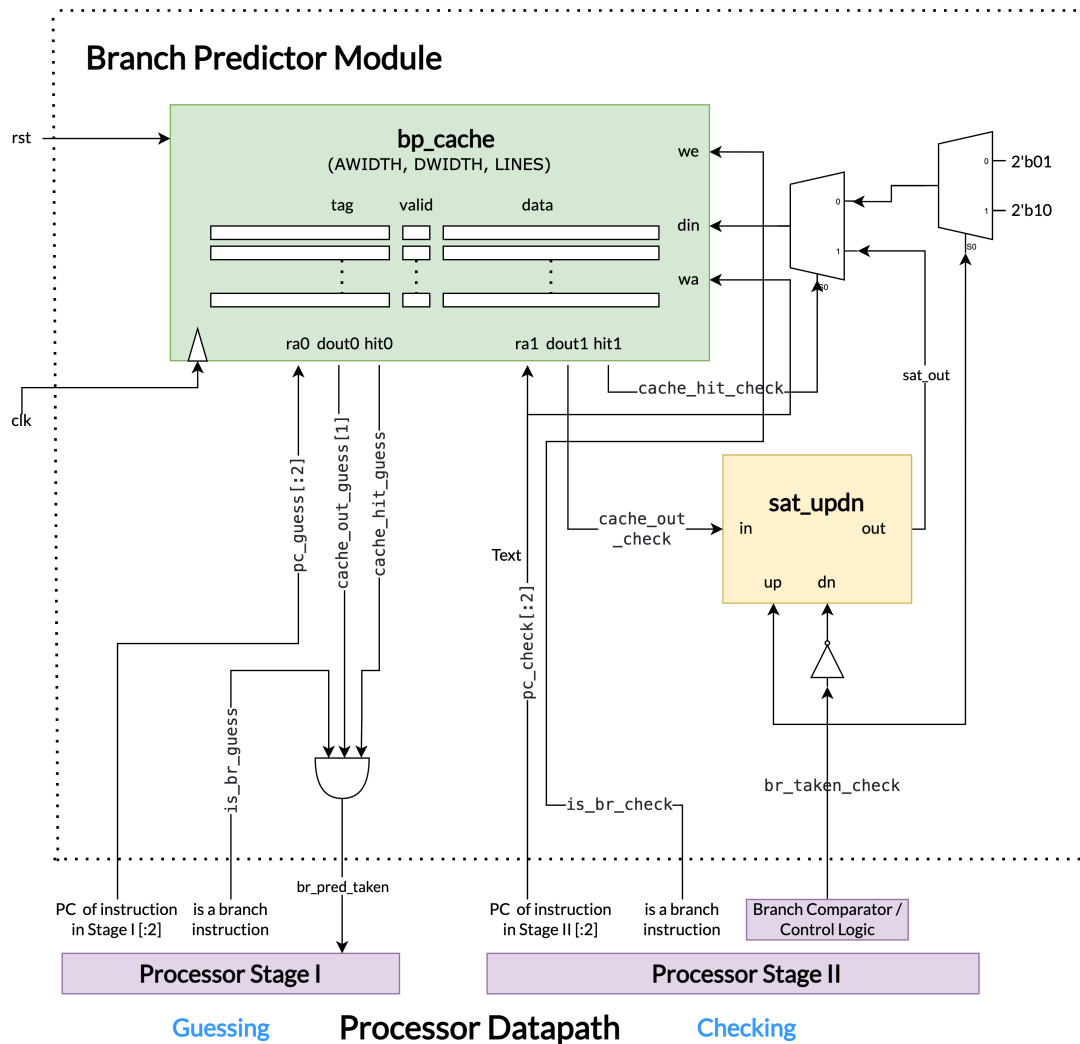

Figure 4: Structure of the Branch Predictor Module; see how the cache, saturating counter composes the branch predictor and how it interacts with signals from Stage I and II of the processor datapath.

This formulation treats the BHT just as a regular cache, but with entry that represents a saturating counter which can be used for branch prediction. The diagram below illustrates the module interface our skeleton code provide. We separate the module into a cache and a saturating counter. The cache supports 2 asynchronous reads as we can simultaneously have two branch instructions in flight in Stage I and II. The cache supports 1 synchronous write to update its entry.

3.2 Guidelines and requirements

Please pull the skeleton again before starting this checkpoint. We have added skeleton code in hardware/src/riscv_core/branch_prediction as well as test benches and modifications to other parts of the datapath and software to support adding the branch predictor. Before you start this, make sure you have Checkpoint 2 datapath implemented with naive branch predictor and have mechanism to recover from branch mispredict, as we will solely be replacing naive predictor with a more accurate one.

1. Within hardware/src/riscv_core/branch_prediction, we have prepared branch_predictor.v which is the top level branch predictor. It uses components bp_cache.v and sat_updn.v which for both you would implement. You do not need to modify branch_predictor.v. It is important to understand how they all interface with each other, and how you can instantiate and connect the branch_predictor in your datapath.

2. Implement a 2-bit **saturating counter** in sat_updn.v. Note this should be a purely combination circuit that takes in existing counter value and whether to increment/decrement and compute the new counter value. We do not provide a test bench but we recommend you write one.

3. Implement the **cache** in bp_cache.v, with 2 asynchronous read ports and 1 synchronous write port. Each cache line has tag, valid bit, and fields to store the data. The cache should be parameterizable by address width, data width, and number of cache lines.

    (a) EECS151 Student should implement a **direct-mapped** cache.

    (b) EECS251A Student should implement either 1) a direct-map and a 2-way set associative cache. 2) a configurable N-way set associative cache.

    We will not be providing a test bench for the cache. However, you are **required** to design a test bench for your cache that covers representative cases such as read miss, read hit, write, eviction, etc (251 students to write test regarding the associativity). You will need to **explain** your cache testbench to a TA upon checkoff.

4. With both saturating counter and cache implemented, you have completed the branch predictor module. We have provided a **testbench** in hardware/sim/branch_predictor_tb.v. This test bench tests your prediction under a series of branches. You branch predictor must **satisfy** the behavior of this test bench for this checkpoint!

    Note for 251A students: The final test case in branch_predictor_tb tests cache hit/miss after cache line replacement **assuming a direct-mapped cache**. You will need to update this test case to deal with a 2-way set associative or a configurable N-way set associative cache.

5. Connect the branch predictor module to the rest of your CPU datapath, with inputs and

output in appropriate stages of your CPU (this will vary based on your design). Make sure you still pass all the checkpoint 2 tests so it is still functionally correct.

6. To track performance of your branch predictor, we ask you to add two counters, one for **total** number of branch instructions the CPU encountered, and one for the number of **correct** predictions. These counters should be mapped as **Memory Mapped IO** to address 0x8000_001c and 0x8000_0020 respectively; they can also be reset similar to cycle and instruction counters. See 2.9.4 Memory Mapped I/O section for updated mapping. With these stats, you can calculate the **branch prediction accuracy**. The mmult program has been modified to print those results at the end as well, make sure you recompile it after pulling the changes.

   We have also provided a testbench in hardware/sim/mmio_counter_tb.v, which will run a small set of instructions and print out the MMIO counter values. You may find it helpful to debug branch prediction and MMIO counters here in simulation before testing it on the FPGA. Feel free to add additional test cases.

7. We connected SWITCH[0] to enable bp_enable. Once you successfully uploaded the new design to CPU, run mmult with the switch off and on will show the result with branch prediction off / on. The checksum should remain the same but you should see improved performance with branch prediction enabled.

3.3 Checkoff

Checkpoint 3 materials should be committed to your project repository by Dec 02, 2021.

- Pass test bench in hardware/sim/branch_predictor_tb.v

- Explain your cache test bench design and how that covers all the representative scenarios for your cache to the TA.

- Run mmult with branch prediction disabled and enabled by toggling the switch. The program must still be functionally correct with the correct checksum. For both settings, record the CPI, the amount of total branch predictions, the amount of correct branch predictions, and calculate branch prediction accuracy. CPI and branch prediction accuracy must be **strictly better** with Checkpoint 3 branch predictor pass this checkpoint.

4 Checkpoint 4 - Optimization

Checkpoint 4 is an optimization checkpoint lumped with the final checkoff. This part of the project is designed to give students freedom to implement the optimizations of their choosing to improve the performance of their processor.

The optimization goal for this project is to minimize the **execution time** on the mmult program, as defined by the 'Iron Law' of Processor Performance.

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

The number of instructions is fixed, but you have freedom to change the CPI and the CPU clock frequency. Often you will find that you will have to sacrifice CPI to achieve a higher clock frequency, but there also will exist opportunities to improve one or both of the variables without compromises.

4.1 Grading on Optimization: Frequency vs. CPI

You must demonstrate that your processor has a working BIOS and can load and execute **mmult**.

Full credit will be awarded if you're able to evaluate different design trade-off points (at least three) between frequency and CPI of **mmult** (especially if you have implemented some interesting optimization for CPI and increase the frequency further would degrade the performance instead of helping).

When exploring design trade-offs, also consider the total FPGA resource utilization. You should note the number of LUTs, Block RAMs, FFs, and DSP Blocks that you use for each design point. You can find these numbers in `hardware/build/impl/post_place_utilization.rpt`.

Also note that your final optimized design does not need to be strictly three-stage pipeline. Extra credit will be awarded based on additional optimizations listed in the extra credit section, please check with a GSI ahead of time if you are expanding to include these. If you have other ideas please check with a GSI to see if it can be awarded extra credit.

4.2 Clock Generation Info + Changing Clock Frequency

Open up `z1top.v`. There's top level input called `CLK_125MHZ_FPGA`. It's a 125 MHz clock signal, which is used to derive the CPU clock.

Scrolling down, there's an instantiation of `clocks` (`clocks.v`), which is a wrapper module of PLL (phase locked loop) primitives on the FPGA. This is a circuit that can create a new clock from an existing clock with a user-specified multiply-divide ratio.

The `CLKIN1` (line 47) input clock of the PLL is driven by the 125 MHz `CLK_125MHZ_FPGA`. The frequency of `CLKOUT0` (line 39) is calculated as:

$$\texttt{CLKOUT0\_}f = \texttt{CLKIN1\_}f \times \frac{\texttt{CPU\_CLK\_CLKFPOUT\_MULT}}{\texttt{CPU\_CLK\_DIVCLK\_DIVIDE} \times \texttt{CPU\_CLK\_CLKOUT\_DIVIDE}}$$

Based on the default parameter values, we get the following CPU clock frequency:

$$\texttt{CLKOUT0\_}f = 125 \text{ MHz} \times \frac{34}{5 \times 17} = 50 \text{ MHz}$$

To change the target CPU clock frequency, you must do **both** of the following:

1. Change the parameters (`CPU_CLK_CLKFBOUT_MULT`, `CPU_CLK_DIVCLK_DIVIDE`, `CPU_CLK_CLKOUT_DIVIDE`) in z1top according to the table below

2. Change the `CPU_CLOCK_FREQ` parameter in z1top to match the PLL parameters

Table 4: PLL Settings

Frequency	DIVCLK_DIVIDE	CLKFBOUT_MULT	CLKOUT_DIVIDE
50 MHz	5	34	17
60 MHz	5	36	15
65 MHz	5	39	15
70 MHz	5	42	15
75 MHz	5	33	11
80 MHz	5	48	15
85 MHz	5	34	10
90 MHz	5	36	10
95 MHz	5	38	10
100 MHz	5	36	9

4.3 Critical Path Identification

After running `make impl`, timing analysis will be performed to determine the critical path(s) of your design. The timing tools will automatically figure out the CPU's clock timing constraint based on the PLL parameters you set in `z1top.v`.

The critical path can be found by looking in

`hardware/build/impl/post_route_timing_summary.rpt`.

Look for the paths within your CPU. There are 2 types of paths: max delay paths (setup) and min delay paths (hold).

For each timing path look for the attribute called "slack". Slack describes how much extra time the combinational delay of the path has before/after the rising edge of the receiving clock.

For a **max delay path**, the slack is a **setup** time attribute. Positive slack means that this timing path resolves and settles **before** the rising edge of the clock, and negative slack indicates a setup time violation.

For a **min delay path**, the slack is a **hold** time attribute. Positive slack means that this timing path resolves and settles **after** the rising edge of the clock, and negative slack indicates a hold time violation.

There are some common delay types that you will encounter. `LUT` delays are combinational delays through a LUT. `net` delays are from wiring delays. They come with a fanout attribute which you should aim to minimize. Notice that your logic paths are usually dominated by routing delay; as you optimize, you should reach the point where the routing and LUT delays are about equal portions of the total path delay.

4.3.1 Schematic View

To visualize the path, you can open Vivado (`make vivado`), and open a DCP (Design Checkpoint) file (File → Checkpoint → Open). The DCP is in `build/impl/z1top_routed.dcp`.

Re-run timing analysis with Reports → Timing → Report Timing Summary. Use the default options and click OK. Navigate (on the bottom left) to Intra-Clock Paths → `cpu_clk` → Setup (or Hold).

You can double-click any path to see the logic elements along it, or you can right-click and select Schematic to see a schematic view of the path.

The paths in post-PAR timing report may be hard to decipher since Vivado does some optimization to move/merge registers and logic across module boundaries. It may help to look at the post-synth DCP in `build/synth/z1top.dcp`. You can also use the keep_hierarchy attribute to prevent Vivado from moving registers and logic across module boundaries (although this may degrade QoR).

```
// in z1top.v
(* keep_hierarchy="yes" *) cpu #( ) cpu ( );
```

4.3.2 Finding Actual Critical Paths

When you first check the timing report with a 50 MHz clock, you might not see your 'actual' critical path. 50 MHz is easy to meet and the tools will only attempt to optimize routing until timing is met, and will then stop.

You should increase the clock frequency slowly and rerun `make impl` until you fail to meet timing. At this point, the critical paths you see in the report are the 'actual' ones you need to work on.

Don't try to increase the clock speed up all the way to 100 MHz initially, since that will cause the routing tool to give up even before it tried anything.

4.4 Optimization Tips

As you optimize your design, you will want to try running `mmult` on your newly optimized designs as you go along. You don't want to make a lot of changes to your processor, get a better clock speed, and then find out you broke something along the way.

You will find that sacrificing CPI for a better clock speed is a good bet to make in some cases, but will worsen performance in others. You should keep a record of all the different optimizations you tried and the effect they had on CPI and minimum clock period; this will be useful for the final report when you have to justify your optimization and architecture decisions.

There is no limit to what you can do in this section. The only restriction is that you have to run the original, unmodified `mmult` program so that the number of instructions remain fixed. You can add as many pipeline stages as you want, stall as much or as little as desired, or perform any other optimizations. If you decide to do a more advanced optimization (like a 5 stage pipeline), ask the staff to see if you can use it as extra credit in addition to the optimization.

Keep notes of your architecture modifications in the process of optimization. Consider, but don't obsess, over area usage when optimizing (keep records though).

4.5   Checkoff

Refer to **4.1**. You will run your new implementation on the FPGA again and will be graded based on the best `mmult` performance you were able to achieve, but *more critically* on how many design points you explored.

5   Grading and Extra Credit

All groups must complete the final checkoff by Dec 09, 2022. If you are unable to make the deadline for any of the checkpoints, it is still in your best interest to complete the design late, as you can still receive most of the credit if you get a working design by the final checkoff.

5.1   Checkpoints

We have divided the project up into checkpoints so that you (and the staff) can pace your progress.

5.2   Style: Organization, Design

Your code should be modular, well documented, and consistently styled. Projects with incomprehensible code will upset the graders.

5.3   Final Project Report

Upon completing the project, you will be required to submit a report detailing the progress of your EECS151/251A project. The report should document your final circuit at a high level, and describe the design process that led you to your implementation. We expect you to document and justify any tradeoffs you have made throughout the semester, as well as any pitfalls and lessons learned. Additionally, you will document any optimizations made to your system, the system's performance in terms of area (resource use), clock period, and CPI, and other information that sets your project apart from other submissions.

The staff emphasizes the importance of the project report because it is the product you are able to take with you after completing the course. All of your hard work should reflect in the project report. Employers may (and have) ask to examine your EECS151/251A project report during interviews. Put effort into this document and be proud of the results. You may consider the report to be your medal for surviving EECS151/251A.

5.3.1   Report Details

You will turn in your project report PDF file on Gradescope by **Dec 12, 2022, 11:59PM**. The report should be around 8 pages total with around 5 pages of text and 3 pages of figures ($\pm$ a few pages on each), though this is not a strict limit. Ideally you should mix the text and figures together.

Here is a suggested outline and page breakdown for your report. You do not need to strictly follow this outline, it is here just to give you an idea of what we will be looking for.

- **Project Functional Description and Design Requirements**. Describe the design objectives of your project. You don't need to go into details about the RISC-V ISA, but you need to describe the high-level design parameters (pipeline structure, memory hierarchy, etc.) for this version of the RISC-V. (≈ 0.5 page)

- **High-level organization**. How is your project broken down into pieces. Block diagram level-description. We are most interested in how you broke the CPU datapath and control

down into submodules, since the code for the later checkpoints will be pretty consistent across all groups. Please include an updated block diagram ($\approx$ 1 page).

- **Detailed Description of Sub-pieces**. Describe how your circuits work. Concentrate here on novel or non-standard circuits. Also, focus your attention on the parts of the design that were not supplied to you by the teaching staff. ($\approx$ 2 pages).

- **Status and Results**. What is working and what is not? At what frequency (50MHz or greater) does your design run? Do certain checkpoints work at a higher clock speed while others only run at 50 MHz? Please also provide the area utilization. Also include the CPI and minimum clock period of running `mmult` for the various optimizations you made to your processor. This section is particularly important for non-working designs (to help us assign partial credit). ($\approx$ 1-2 pages).

- **Conclusions**. What have you learned from this experience? How would you do it different next time? ($\approx$ 0.5 page).

- **Division of Labor. This section is mandatory. Each team member will turn in a separate document from this part only**. The submission for this document will also be on Gradescope. How did you organize yourselves as a team. Exactly who did what? Did both partners contribute equally? Please note your team number next to your name at the top. ($\approx$ 0.5 page).

When we grade your report, we will grade for clarity, organization, and grammar. Both team members need to submit the Final Report assignment (same report content, but with different writeup for division of labor) to Gradescope. **We require your final report to be typeset using tools like LaTeX, or Markdown, or Google Docs/MS Word/Apple Pages etc., but the file that you turn in must be a single PDF file.**

5.4 Extra Credit

Teams that have completed the base set of requirements are eligible to receive extra credit worth up to 10% of the project grade by adding extra functionality and demonstrating it at the time of the final checkoff.

The following are suggested projects that may or may not be feasible in one week.

- Improve Branch Predictor: Beyond our 2-bit saturating based BHT, you can improve the Branch Predictor from Checkpoint 3. You can come up with improved scheme, such as incorporating making the cache set associative (required for 251A students), incorporating global history, adding a Branch Target Buffer, etc. Whatever you choose to do, you must improve your CPI from checkpoint 3 to qualify for extra credit.

- 5-Stage Pipeline: Add more pipeline stages and push the clock frequency past 100MHz. Note that on some deeper pipelines, the `host_to_fpga` task does not wait enough clock cycles for the CPU to process a character before sending the next character over uart. You may want to bump up the wait cycles, e.g., 500 cycles, to avoid this (though it does slow down the testbench).

- RISC-V M Extension: Extend the processor with a hardware multiplier and divider

- Everything 100MHz or beyond: Push the frequency of the full `z1top` to 100MHz or better.

When the time is right, if you are interested in implementing any of these, see the staff for more details.

5.5 Project Grading

70% Functionality at project due date. You will demonstrate the functionality of your processor during the final interview.

15% Optimization at final project due date. This score is contingent on implementing all the required functionality. An incomplete project will receive a zero in this category.

5% Checkpoint functionality. You are graded on functionality for each completed checkpoint at the checkpoint deadline. The total of these scores makes up 5% of your project grade. The weight of each checkpoint's score may vary.

10% Final report and style demonstrated throughout the project.

Not included in the above tabulations are point assignments for extra credit as discussed above. Extra credit is discussed below:

Up to 10% Additional functionality. Credit based on additional functionality will be qualified on a case by case basis. Students interested in expanding the functionality of their project must meet with a GSI well ahead of time to be qualified for extra credit. Point value will be decided by the course staff on a case by case basis, and will depend on the complexity of your proposal, the creativity of your idea, and relevance to the material taught.

Appendices

Appendix A   Local Development

You can build the project on your laptop but there are a few dependencies to install. In addition to Vivado and Icarus Verilog, you need a RISC-V GCC cross compiler and an `elf2hex` utility.

A.1   Linux

A system package provides the RISC-V GCC toolchain (Ubuntu): `sudo apt install gcc-riscv64-linux-gnu`. There are packages for other distros too.

To install `elf2hex`:

```
git clone git@github.com:sifive/elf2hex.git
cd elf2hex
autoreconf -i
./configure --target=riscv64-linux-gnu
make
vim elf2hex # Edit line 7 to remove 'unknown'
sudo make install
```

A.2   OSX, Windows

Download SiFive's GNU Embedded Toolchain from here. See the 'Prebuilt RISC-V GCC Toolchain and Emulator' section.

After downloading and extracting the tarball, add the `bin` folder to your `PATH`. For Windows, make sure you can execute `riscv64-unknown-elf-gcc -v` in a Cygwin terminal. Do the same for OSX, using the regular terminal.

For Windows, re-run the Cygwin installer and install the packages `git, python3, python2, autoconf, automake, libtool`. See this StackOverflow question if you need help selecting the exact packages to install.

Clone the `elf2hex` repo `git clone git@github.com:sifive/elf2hex`. Follow the instructions in the elf2hex repo README to build it from git. You should be able to run `riscv64-unknown-elf-elf2hex` in a terminal.

Appendix B   BIOS

This section was written by Vincent Lee, Ian Juch, and Albert Magyar.

B.1   Background

For the first checkpoint we have provided you a BIOS written in C that your processor is instantiated with. BIOS stands for Basic Input/Output System and forms the bare bones of the CPU system on initial boot up. The primary function of the BIOS is to locate, and initialize the system and peripheral devices essential to the PC operation such as memories, hard drives, and the CPU cores.

Once these systems are online, the BIOS locates a boot loader that initializes the operating system loading process and passes control to it. For our project, we do not have to worry about loading the BIOS since the FPGA eliminates that problem for us. Furthermore, we will not deal too much with boot loaders, peripheral initialization, and device drivers as that is beyond the scope of this class. The BIOS for our project will simply allow you to get a taste of how the software and hardware layers come together.

The reason why we instantiate the memory with the BIOS is to avoid the problem of bootstrapping the memory which is required on most computer systems today. Throughout the next few checkpoints we will be adding new memory mapped hardware that our BIOS will interface with. This document is intended to explain the BIOS for checkpoint 1 and how it interfaces with the hardware. In addition, this document will provide you pointers if you wish to modify the BIOS at any point in the project.

B.2   Loading the BIOS

For the first checkpoint, the BIOS is loaded into the Instruction memory when you first build it. As shown in the Checkpoint 1 specification, this is made possible by instantiating your instruction memory to the BIOS file by building the block RAM with the `bios151v3.hex` file. If you want to instantiate a modified BIOS you will have to change this .hex file in your block RAM directory and rebuild your design and the memory.

To do this, simply cd to the `software/bios151v3` directory and make the .hex file by running "make". This should generate the .hex file using the compiler tailored to our ISA. The block RAM will be instantiated with the contents of the .hex file. When you get your design to synthesize and program the board, open up screen using the same command from Lab 5:

`screen $SERIALTTY 115200`

or

`screen /dev/ttyUSB0 115200`

Once you are in `screen`, if you CPU design is working correctly you should be able to hit Enter and a carrot prompt `'>'` will show up on the screen. If this doesn't work, try hitting the reset button on the FPGA which is the center compass switch and hit enter. If you can't get the BIOS carrot to come up, then your design is not working and you will have to fix it.

B.3 Loading Your Own Programs

The BIOS that we provide you is written so that you can actually load your own programs for testing purposes and benchmarking. Once you instantiate your BIOS block RAM with the `bios151v3.hex` file and synthesize your design, you can transfer your own program files over the serial line.

To load you own programs into the memory, you need to first have the .hex file for the program compiled. You can do this by copying the software directory of one of our C programs folders in /software directory and editing the files. You can write your own MIPS program by writing test code to the .s file or write your own c code by modifying the .c file. Once you have the .hex file for your program, impact your board with your design and run:

```
hex_to_serial <file name> <target address>
```

The `<file name>` field corresponds to the .hex file that you are to uploading to the instruction memory. The `<target address>` field corresponds to the location in memory you want to write your program to.

Once you have uploaded the file, you can fire up screen and run the command:

```
jal <target hex address>
```

Where the `<target hex address>` is where you stored the location of the hex file over serial. Note that our design does not implement memory protection so try to avoid storing your program over your BIOS memory. Also note that the instruction memory size for the first checkpoint is limited in address size so large programs may fail to load. The jal command will change the PC to where your program is stored in the instruction memory.

B.4 The BIOS Program

The BIOS itself is a fairly simple program and composes of a glorified infinite loop that waits for user input. If you open the `bios151v3.c` file, you will see that the main method composes of a large for loop that prints a prompt and gets user input by calling the `read_token` method. If at any time your program execution or BIOS hangs or behaves unexpected, you can hit the reset button on your board to reset the program execution to the main method. The `read_token` method continuously polls the UART for user input from the keyboard until it sees the character specified by ds. In the case of the BIOS, the termination character `read_token` is called with is the 0xd character which corresponds to Enter. The `read_token` method will then return the values that it received from the user. Note that there is no backspace option so if you make a mistake you will have to wait until the next command to fix it.
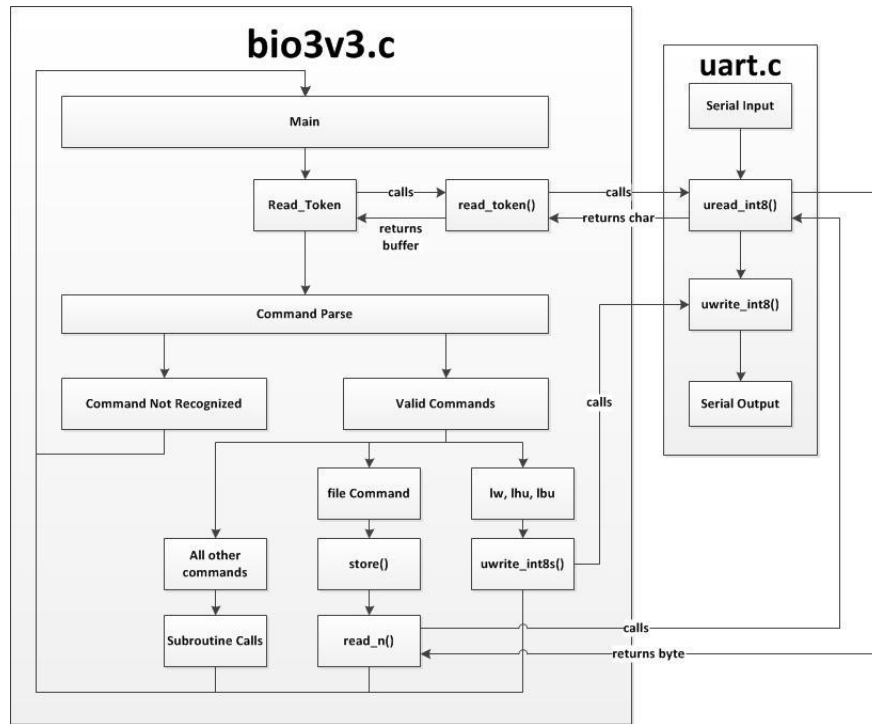
Figure 5: BIOS Execution Flow

The buffer returned from the `read_token` method with the user input is then parsed by comparing the returned buffer against commands that the BIOS recognizes. If the BIOS parses a command successfully it will execute the appropriate subroutine or commands. Otherwise it will tell you that the command you input is not recognized. If you want to add commands to the BIOS at any time in the project, you will have to add to the comparisons that follow after the `read_token` subroutine in the BIOS.

B.5 The UART

You will notice that some of the BIOS execution calls will call subroutines in the uart.c file which takes care of the transmission and reception of byte over the serial line. The uart.c file contains three subroutines. The first subroutine, `uwrite_int8` executes a UART transmission for a single byte by writing to the output data register. The second subroutine `uwrite_int8s` allows you to process an array of type `int8_t` or chars and send them over the serial line. The third routine `uread_int8` polls the UART for valid data and reads a byte from the serial line.

In essence, these three routines are operating the UART on your design from a software view using the memory mapped I/O. Therefore, in order for the software to operate the memory map correctly, the `uart.c` module must store and load from the correct addresses as defined by out memory map. You will find the necessary memory map addresses in the uart.h file that conforms to the design specification.

B.6 Command List

The following commands are built into the BIOS that we provide for you. All values are interpreted in hexadecimal and do not require any radix prefix (ex. "0x"). Note that there is not backspace command.

`jal <hexadecimal address>` - Moves program execution to the specified address

`lw <hexadecimal address>` - Displays word at specified address to screen

`lhu <hexadecimal address>` - Displays half at specified address to screen

`lbu <hexadecimal address>` - Displays byte at specified address to screen

`sw <value> <hexadecimal address>` - Stores specified word to address in memory

`sh <value> <hexadecimal address>` - Stores specified half to address in memory

`sb <value> <hexadecimal address>` - Stores specified byte to address in memory

There is another command file in the main() method that is used only when you execute `hex_to_serial`. When you execute `hex_to_serial`, your workstation will initiate a byte transfer by calling this command in the BIOS. Therefore, don't mess with this command too much as it is one of the more critical components of your BIOS.

B.7 Adding Your Own Features

Feel free to modify the BIOS code if you want to add your own features during the project for fun or to make your life easier. If you do choose to modify the BIOS, make sure to preserve essential functionality such as the I/O and the ability to store programs. In order to add features, you can either add to the code in the `bios151v3.c` file or create your own c source and header files. Note that you do not have access to standard c libraries so you will have to add them yourself if you need additional library functionality.