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

EECS151/251A - LB, Spring 2020

Project Specification: RISCV151

Version 4.4

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. Afterwards, you will attach the IO circuits you built in the lab to the CPU, and design a 2D Convolutional filter accelerator for image processing applications. 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

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. The checkpoints are mostly for you to work progressively towards to goal of building a fully-functional RISC-V core. Do not expect to start working on the project last minute.

- **Mar 22, 2020 - Checkpoint 1 (1 week)** - Submit a report to Gradescope with the following items: a schematic of your processor's datapath and pipeline stages, and your answers to the checkpoint 1 questions. The TA will give you feedback if applicable. In addition, push all of your IO-circuit Verilog modules that you have implemented in the labs to your assigned Github repository: `debouncer`, `edge_detector`, `synchronizer`, `fifo`, `uart_receiver`, `uart_transmitter`, `display_controller`.
- **April 5, 2020 - Checkpoint 1.1 (2 week)** - Go over your RISC-V microarchitecture with the TA (via Zoom). You should have a working ALU module and a complete instructional decoder by this point, and some basic datapath and control logic. This is mainly for the TA to keep track of your current progress.
- **April 12, 2020 - Checkpoint 2 (1 weeks)** - Implement a fully functional RISC-V processor core in Verilog. Your processor core should pass the assembly tests.
- **April 26, 2020 - Checkpoint 3 (2 weeks)** - Attach I/O components from lab to your processor (FIFOs, buttons, switches, display), and 2D Convolutional filter. If time permits, interfacing with off-chip DRAMs using AXI4 bus.

- **May 8 (by appointment) - Final Checkoff + Demo** - Final processor optimization and checkoff
- **May 11 - 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 - 3-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. The suggested way for initializing your repository with the skeleton files is as follows:

```
git clone git@github.com:EECS150/project_skeleton_sp20.git
cd project_skeleton_sp20
git remote add my-repo git@github.com:EECS150/sp20_teamXX.git
git push my-repo master
```

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

```
cd ..
rm -rf project_skeleton_sp20
git clone git@github.com:EECS150/sp20_teamXX.git
cd sp20_teamXX
git remote add staff git@github.com:EECS150/project_skeleton_sp20.git
```

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

To get a team repo, fill one line in the Google spreadsheet posted on Piazza with your team information (names, Github logins).

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 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` (overwriting any provided skeletons).

```
cd sp20_teamXX
cp fpga_labs_sp20/lab6/debouncer.v sp20_teamXX/hardware/src/io_circuits/.
```

Copy these files from the labs:

```
lab6/debouncer.v
lab6/synchronizer.v
lab6/edge_detector.v
lab6/fifo.v
lab6/uart_receiver.v
lab6/uart_transmitter.v
lab6/display_controller.v
```

2.3 Project Skeleton Overview

- `hardware`
  - `src`
    - `z1top.v`: Top level module. The RISC-V CPU is instantiated here.
    - `riscv_core/Riscv151.v`: All of your CPU datapath and control should be contained in this file.
    - `io_circuits`: Your IO circuits from previous lab exercises.
    - `EECS151.v`: Our EECS151-SP20 library file of register and memory modules. **You are expected to use these modules for your sequential logic**.
  - `sim`
    - `assembly_testbench.v`: Starting point for testing your CPU. Works with the software in `assembly_tests`.
    - `echo_testbench.v`: Runs the software in `echo` on your CPU. The software implements the echo FSM from lab 5, and the testbench controls an off-chip UART to test it.
- `software`
  - `bios151v3`: 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 FSM from Lab 5 in software.
- `assembly_tests`: Use this as a template to write assembly tests for your processor designed to run in simulation.
- `c_example`: Use this as an example to write C programs.
- `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 impl` 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

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

the critical path. The RAMs we are using for the data, instruction, and BIOS memories are both **synchronous** read and **synchronous** write.

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`: `REGFILE_1W2R`. The register file has two asynchronous-read ports and one synchronous-write port (positive edge).

2.8 RAMs

In this project, we will be using inferred block RAMs to implement memories for the processor.

2.8.1 Initialization

The Verilog `$readmemb(filename, path to 2D reg, start addr, end addr)` and `$readmemh()` system tasks can be used to initialize a 2D reg with a text file containing the desired contents of the memory (in binary or hex respectively). These system tasks are placed inside an `initial` block and point to a particular 2D reg instance to initialize. If a 2D reg isn't initialized it is filled with Xs.

For synthesis, the BIOS memory is initialized with the contents of the BIOS program, and the other memories are left uninitialized (zeroed out)
(see `src/memories/{imem, dmem, bios_mem}.v`).

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).

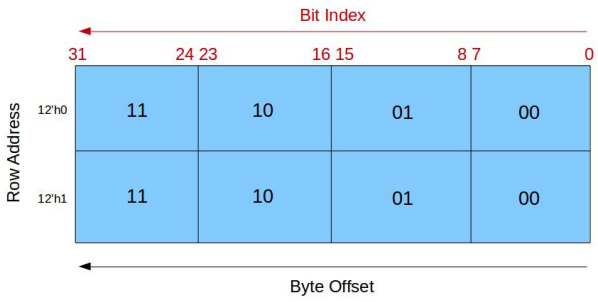

Figure 1: Block RAM organization. The labels for row address **should read 14'h0 and 14'h1.**

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 2.

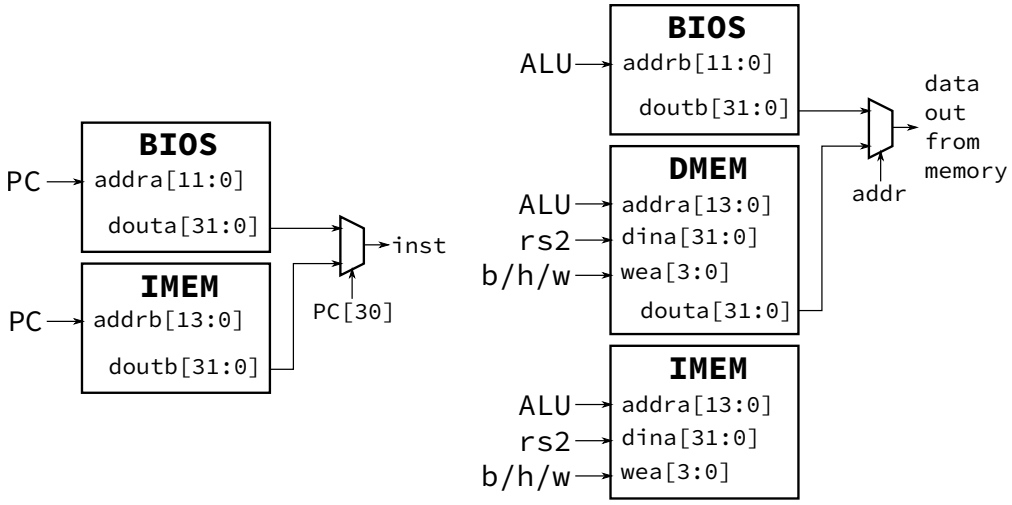

Figure 2: The Riscv151 memory architecture. There is only 1 IMEM and DMEM instance in Riscv151 but their ports are shown separately in this figure for clarity. The left half of the figure shows the instruction fetch logic and the right half shows the memory load/store logic.

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/bios151v3`). 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 2.14.

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 3 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, data_out_valid, data_in_ready}`
`32'h80000004`	UART receiver data	Read	`{24'b0, data_out}`
`32'h80000008`	UART transmitter data	Write	`{24'b0, 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

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 `data_out_valid` and `data_in_ready` signals; this check is handled in software. The CPU needs to drive `data_out_ready` and `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
2. Test the entire CPU one instruction at a time with hand-written assembly — see `assembly_testbench.v`
3. Test the CPU's memory mapped I/O — see `echo_testbench.v`
4. Run the `riscv-tests` ISA test suite

2.10.1 Integration Testing

Once you are confident that the individual components of your processor are working in isolation, you will want to test the entire processor as a whole. The easiest way to do this is to write an assembly program that tests all of the instructions in your ISA. A skeleton is provided for you in `software/assembly_tests`. See Section 2.12 for details.

Once you have verified that all the instructions in the ISA are working correctly, you may also want to verify that the memory mapped I/O and instruction/data memory reading/writing work with a similar assembly program.

2.11 Software Toolchain - Writing RISC-V Programs

A GCC RISC-V toolchain has been built and installed in the eecs151 home directory; these binaries will run on any of the c125m machines in the 125 Cory lab. The most relevant programs in the toolchain are:

- `riscv64-unknown-elf-gcc`: GCC for RISC-V, compiles C code to RISC-V binaries.
- `riscv64-unknown-elf-as`: RISC-V assembler, compiles assembly code to RISC-V binaries.
- `riscv64-unknown-elf-objdump`: Dumps RISC-V binaries as readable assembly code.

Look at the `software/c_example` folder for an example of a C program.

There are several files:

- `start.s`: This is an assembly file that contains the start of the program. It initialises the stack pointer then jumps to the `main` label. Edit this file to move the top of the stack. Typically your stack pointer is set to the top of the data memory address space, so that the stack has enough room to grow downwards.
- `c_example.ld`: This linker script sets the base address of the program. For Checkpoint 2, this address should be in the format `0x1000xxxx` The .text segment offset is typically set to the base of the instruction memory address space.

- `c_example.elf`: Binary produced after running `make`.
  Use `riscv64-unknown-elf-objdump -Mnumeric -D c_example.elf` to view the assembly code.
- `c_example.dump`: Assembly dump of the binary.

2.12 Assembly Tests

Hand written assembly tests are in `software/assembly_tests/start.s` and the corresponding testbench is in `hardware/sim/assembly_testbench.v`. To run the test, run:
`make sim/assembly_testbench.fst` (iverilog) or `make sim/assembly_testbench.vpd` (VCS).

If you want to forcibly re-run the test even though you didn't change the Verilog, run:
`make -B sim/assembly_testbench.fst`

`start.s` contains assembly that's compiled and loaded into the BIOS RAM by the testbench.

```
_start:

# Test ADD
li x10, 100          # Load argument 1 (rs1)
li x11, 200          # Load argument 2 (rs2)
add x1, x10, x11     # Execute the instruction being tested
li x20, 1            # Set the flag register to stop execution and inspect the
↪  result register
                     # Now we check that x1 contains 300 in the testbench

Done: j Done
```

The `assembly_testbench` toggles the clock one cycle at time and waits for register `x20` to be written with a particular value (in the above example: 1). Once `x20` contains 1, the testbench inspects the value in `x1` and checks it is 300, which indicates your processor correctly executed the add instruction.

If the testbench times out it means `x20` never became 1, so the processor got stuck somewhere or `x20` was written with another value.

2.13 RISC-V ISA Tests

You will need the CSR instructions to work before you can use this test suite, and you should have confidence in your hand-written assembly tests. Test the CSR instructions using hand assembly tests.

To run the ISA tests, first pull the latest skeleton changes:

```
git pull staff master
git submodule update --init --recursive
```

Modify line 4 of `sim/isa_testbench.v` to point to the tohost CSR register in your CPU. Then run:

```
cd software/riscv-isa-tests
make
cd hardware
make isa-tests
```

The output of each test (a .log file and a .fst waveform) is stored in `hardware/sim/isa`. To re-run a particular ISA test (e.g. `add`) run `make sim/isa/add.fst`.

To check what tests passed run `cat sim/isa/*.log | grep -i pass`. To check for failures run `cat sim/isa/*.log | grep -i fail`.

You can expect the `fence_i` test to fail, but the rest should pass. If you're failing other tests, debug using the test assembly file in
`software/riscv-isa-tests/riscv-tests/isa/rv32ui` or the generated assembly dump.

The `RESET_PC` parameter is used in `isa_testbench` to start the test in the IMEM instead of the BIOS. Make sure you have used it in `Riscv151.v`.

2.14 BIOS and Programming your CPU

We have provided a BIOS program in `software/bios151v3` that allows you to interact with your CPU and download other programs over UART. The BIOS is just an infinite loop that reads from the UART, checks if the input string matches a known control sequence, and then performs an associated action. For detailed information on the BIOS, see Appendix B.

To run the BIOS:

1. Verify that the stack pointer and .text segment offset are set properly in `start.s` and `bios151v3.ld`
2. Compile the program with `make` in the `software/bios151v3` directory
3. Verify the `bios_mem.v` module is initialized with the BIOS hex file
4. Build a bitstream and program the FPGA
5. Use screen to access the serial port:

   ```
   screen $SERIALTTY 115200
   ```

6. Press the reset button to make the CPU PC go to the start of BIOS memory

Close screen using `Ctrl-a Shift-k`, or other students won't be able to use the serial port! If you can't access the serial port you can run `killscreen` to kill all screen sessions.

If all goes well, you should see a `151 >` prompt after pressing return. The following commands are available:

- `jal <address>`: Jump to address (hex).

- `sw, sb, sh <data> <address>`: Store data (hex) to address (hex).
- `lw, lbu, lhu <address>`: Prints the data at the address (hex).

As an example, running `sw cafef00d 10000000` should write to the data memory and running `lw 10000000` should print the output `10000000: cafef00d`. Please also pay attention that writes to the instruction memory (`sw ffffffff 20000000`) do not write to the data memory, i.e. `lw 10000000` still should yield `cafef00d`.

In addition to the command interface, the BIOS allows you to load programs to the CPU. *With screen closed*, run:

```
hex_to_serial <hex_file> <address>
```

This stores the `.hex` file at the specified hex address. In order to write into both the data and instruction memories, **remember to set the top nibble to 0x3** (i.e. `hex_to_serial echo.hex 30000000`, assuming the `.ld` file sets the base address to `0x10000000`).

You also need to ensure that the stack and base address are set properly (See Section 2.11). For example, before making the `mmult` program you should set the set the base address to `0x10000000` (see 2.16). Therefore, when loading the `mmult` program you should load it at the base address: `hex_to_serial mmult.hex 30000000`. Then, you can jump to the loaded `mmult` program in in your screen session by using `jal 10000000`.

2.15 Target Clock Frequency

By default, the CPU clock frequency is set at 50MHz. It should be easy to meet timing at 50 MHz. Look at the timing report to see if timing is met. If you failed, the timing reports specify the critical path you should optimize.

For this checkpoint, we will allow you to demonstrate the CPU working at 50 MHz, but for the final checkoff at the end of the semester, you will need to optimize for a higher clock speed ($\geq$ 100MHz) for full credit. Details on how to build your FPGA design with a different clock frequency will come later.

2.16 Matrix Multiply

To check the correctness and performance of your processor we have provided a benchmark in `software/mmult/` which performs matrix multiplication. You should be able to load it into your processor in the same way as loading the echo program.

This program computes $S = AB$, where A and B are 64×64 matrices. The program will print a checksum and the counters discussed in Section 2.9.4. The correct checksum is `0001f800`. If you do not get this, there is likely a problem in your CPU with one of the instructions that is used by the BIOS but not mmult.

The matrix multiply program requires that the stack pointer and the offset of the .text segment be set properly, otherwise the program will not execute properly.

The stack pointer (set in `start.s`) should start near the top of DMEM to avoid corrupting the program instructions and data. It should be set to `0x1000fff0` and the stack grows downwards.

The .text segment offset (set in `mmult.ld`) needs to accommodate the full set of instructions and static data (three 64×64 matrices) in the mmult binary. It should be set to the base of DMEM: `0x10000000`.

The program will also output the values of your instruction and cycle counters (in hex). These can be used to calculate the CPI for this program. Your target CPI should be under 1.2. If your CPI exceeds this value, you will need to modify your datapath and pipeline to reduce the number of bubbles inserted for resolving control hazards (since they are the only source of extra latency in our processor). This might involve performing naive branch prediction or moving the jalr address calculation to an earlier stage.

2.17 How to Survive 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.

2.17.1 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. Write the `reg_file.v` module. Create a Verilog testbench and test the cases in Section 2.7.
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. *Add 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.

11. *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.

12. *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.

13. *Run matrix multiply.* Load the `mmult` program with the `hex_to_serial` utility, and run `mmult` on the FPGA. Verify that it returns the correct checksum.

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

2.18 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.

2.18.1 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 textbook diagrams 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. 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.

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. Can you handle jal writeback data forwarding?

   jal x1, label

   label: addi x2, x1, 4

10. How do you resolve JALR? How many NOPs are injected?
11. What is the most likely critical path in your design?
12. Where do the UART and cycle counters go? How are you going to drive data_in_valid (give a logic expression)?
13. Where does the BIOS mem go? How about the IMEM (in the memory stage)?

Submit your block diagram and answers to Gradescope by Mar 22, 2020. If you are working in team, only one of you is required to upload the report. Please also remember to push your working IO circuits to your Github repository: the TA will test your circuits. The TA will also provide feedback for your block diagram proposal if applicable.

2.18.2 Checkpoint 1.1: Revised RISC-V microarchitecture, Basic Datapath + Control

This checkpoint is for TA to keep track of your current progress, and to ensure that your implementation is on the right track. The TA will go over your design with you. You are expected to have a fully-functioning ALU module and a complete Instructional Decoder. Although highly encouraged, it is not required to pass any assembly test for this checkpoint.

2.18.3 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.

Checkpoint 2 materials should be committed to your project repository by April 12, 2020.

2.18.4 Checkpoints 1 & 2 Deliverables Summary

Deliverable	Due Date	Description
Block Diagram, RISC-V ISA Questions, IO code	Mar 22, 2020	Submit your diagram and answers to Gradescope. Push your IO code to your Github repository.
Revised Block Diagram, Working ALU, Instructional Decoder, Basic datapath and control	April 5, 2020	Check in your code to Github. Go over your RISC-V microarchitecture with TA
RISC-V CPU	April 12, 2020 Check in code to Github	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 be under 1.2

3 Checkpoint 3 - Hardware-accelerated 2D Convolutional Filter - TBA

4 Final Checkpoint - Optimization - TBA

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.

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 impact to the board, open up screen using the same command from Lab 5:

```
screen $SERIALTTY 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 doesnt work, try hitting the reset button on the FPGA which is the center compass switch and hit enter. If you cant 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.

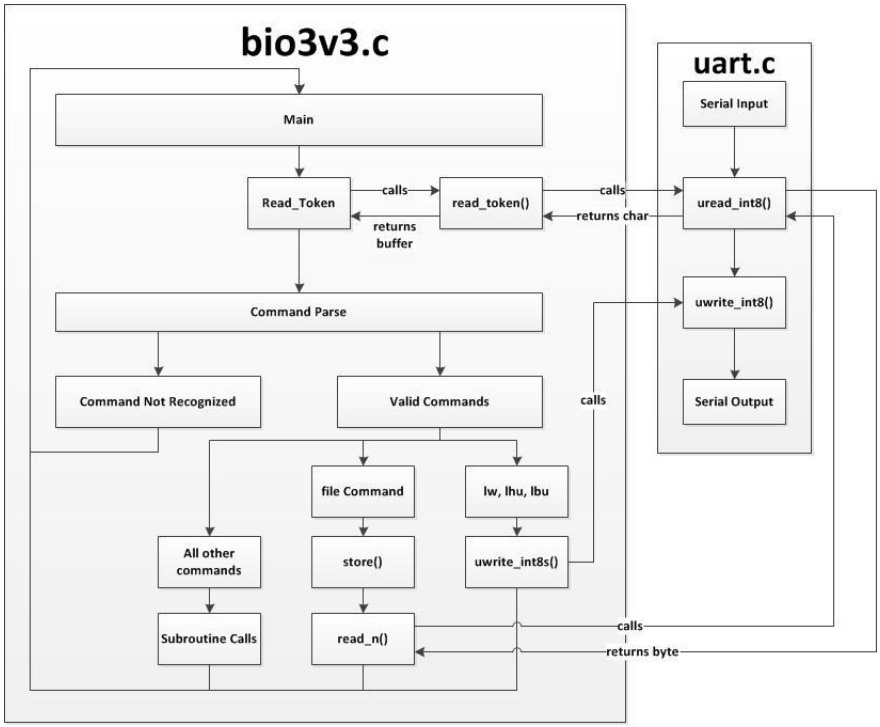

Figure 3: BIOS Execution Flow

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, dont 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.