EECS151/251A
Spring 2024
Final Project Specification

RISCV151

Version 2.21

TA: Dhruv Vaish, Daniel Endraws, Rohit Kanagal

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. After that, you will optimize your CPU to achieve a higher value of the form of merits.

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. Note that this schedule is tentative and is subjected to change as the semester progresses.

- **March 22 - Checkpoint 1** - Draw a schematic of your processor's datapath and pipeline stages, and provide a brief write-up of your answers to the questions. Also commit your design documents (block diagram + write-up) to `docs`.
- **April 5 - Checkpoint 2(a)** - Implement most of the RISC-V processor datapth in Verilog. Your processor core should will be able to run a subset of the basic ISA tests (TBD).
- **April 13 - Checkpoint 2(b)** - Implement a fully functional RISC-V processor core in Verilog. Your processor core should be able to run the **mmult** demo successfully, including all other testbenches.
- **RRR Week - Final Checkoff** - Processor optimization and checkoff. Optional accelerator block.
- **Finals Week - Project Report** - Project report due.

1.2 Tentative Grading Rubric

50% Functionality at the final checkoff. You need to show your design passes all testbenches and executes `mmult` correctly on the FPGA board.

35% Optimization at the final checkoff. The quality of your design will be evaluated according to the figure of merit. This score is contingent on implementing the correct functionality. A malfunctioning design will receive a zero in this category.

5% Checkpoints. The checkpoints are set to guide you to finish your design on time. To encourage you to follow the timing, checking off at each checkpoint on time makes up 5% of your project grade in total. The weight of each checkpoint's score may vary.

10% Project report. The final report summarizing your project.

1.3 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. Optimize your design after everything works well. You should fully utilize the version control system (Git) to maintain a functioning design while making changes incrementally. **If your design does not work at the final checkoff, you will not get any credit for any optimization you did.**

2 Setup

2.1 Creating Your Repository

Create a repository for your team by accepting the GitHub Classroom assignment (https://classroom.github.com/a/
the link will also be posted on Ed. The first member needs to create a team when accepting the assignment, where naming of the group is up to you. The other member should join the team created by the first member.

Your repository is initially empty, Import the project skeleton as follows:

```
git clone git@github.com:EECS151-sp23/fpga_project_sp24-<your team name>.git
cd fpga-project-<your team name>
git remote add skeleton https://github.com/EECS150/fpga_project_sp24.git
git pull skeleton main
git push -u origin main
```

To pull project updates from the skeleton repository (you will be notifed through Ed), run the following:

```
git pull skeleton main
git push
```

while you can pull updates made by the other team member simply by `git pull` after the other did `git push`. Resolve merge conflict carefully if it happens.

Remember to push your changes to the remote repository frequently. **We are not responsible for any loss of your data.** No extensions will be given for that reason.

2.2 Integrating Designs from Labs

You should copy some modules you designed from the labs, overwriting the provided skeleton files in `hardware/src/io_circuits`.

Files to copy from the labs:

```
debouncer.v
synchronizer.v
edge_detector.v
uart_transmitter.v
```

2.3 Project Skeleton Overview

- `hardware`
  - `src`
    - `z1top.v`: Top level module. Your RISC-V CPU is instantiated here.
    - `z1top.xdc`: Constraint file for the top level module.
    - `EECS151.v`: The EECS151 register library. Some bugs have been fixed.

    * `clk_wiz.v`: Generates a clock signal for your CPU.
    * `io_circuits`: IO circuits from previous lab exercises.
    * `riscv_core/cpu.v`: Your RISC-V CPU design goes here. Some modules are provided. Don't change the names of provided modules and signals.
    * `riscv_core/opcode.vh`: Constant definitions for RISC-V opcodes and funct codes.
- `sim`
    * `asm_tb.v`: The testbench works with the software in `software/asm`. The hex file will be written to the instruction/data memories.
    * `cpu_tb.v`: The testbench checks if your CPU can execute all the RV32I instructions (including CSR instructions) correctly, and can handle some simple hazards. This testbench directly edits the contents of the register file and the instruction/data memories for tests.
    * `isa_tb.v`: The testbench works with the RISC-V ISA test suite in `software/riscv-isa-tests`. We use 38 tests from the test suite. The testbench only runs one test at a time. To run all tests, run `make isa-tests`. The hex file will be written to the instruction/data memories.
    * `c_tests_tb.v`: The testbench verifies the correct execution of the software in `software/c_tests`. There are 6 tests provided. The hex file will be written to the instruction/data memories.
    * `uart_parse_tb.v`: The testbench works with the software in `software/uart_parse`. It performs a simple write/read using the serial rx/tx lines. The hex file will be written to the instruction/data memories.
    * `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. The hex file will be written to the instruction/data memories.
    * `mmio_counter_tb.v`: The testbench runs a small set of instructions and print out the memory mapped I/O counter values. This testbench directly edits the contents of the register file and the instruction/data memories for tests.
    * `bios_tb.v`: The testbench simulates the execution of the BIOS program in `software/bios`. 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.
    * `small_tb.v`: The testbench can be used to estimate the CPI of your design. It works with a smaller version of benchmark because the original benchmark is too large to do simulation. It reports the result checksum, cycle count and instruction count, and compares the checksum with a reference value. The program is loaded to the instruction/data memories without using BIOS, while the UART interface is used to get the result checksum and counter values.

    * `mem_path.vh`: Specifies the location of register file and memories. Some testbenches refer to this file to edit/check their values.

  - `scripts`:

    * `run_all_sims.py`: Runs all testbenches above other than `mmio_counter_tb.v` and `small_tb.v`.

    * `hex_to_serial.py`: Sends a hex file to the FPGA through the serial rx line.

    * `run_fpga.py`: Calls `hex_to_serial` and executes the program on the FPGA. It also prints out the result checksum and counter values.

    * `get_fmax.py`: Finds the CPU frequency from timing summary report.

    * `get_cost.py`: Calculates the cost of design from resource utilization report. The cost is a weighted sum of element counts. If your design uses elements that do not have cost assigned, please let us know. Note that your design must not have latches (`LDCE` and `LDPE`).

    * `get_cpi.py`: Runs simulation or FPGA to get the geomean CPI for the benchmark programs. Simulation uses a smaller version of each benchmark, so CPI obtained in simulation is just an estimate. The final grade is based on the real CPI when running the original benchmarks on the FPGA.

    * `fom.py`: Calculates the figre of merits. By default, it reads the newest set of reports for cost and fmax, runs simulation for CPI, and displays the estimated FOM. Change parameters (commandline flags) to specify the report location or to calculate the real FOM through execution on the FPGA.

    * `*.tcl`: Scripts for Xilinx Vivado.

  - `Makefile`: Makefile.

  - `README.md`: Explains make commands.

  - `stubs, sim_models`: Other modules from Xilinx library.

- `software`

  - `Makefile, Makefrag`: Makefiles.

  - `151_library`: Files needed to compile software for our RISC-V CPU.

  - `asm`: Template of assembly tests. Modify this for a particular test you want to perform.

  - `riscv-isa-tests`: Tests from the RISC-V ISA test suite will be compiled here.

  - `c_tests`: Example C programs for tests. You may add a new one.

  - `uart_parse`: Simple tests using UART ports.

  - `echo`: The echo program, refer to the explanation for `echo_tb.v` above.

  - `bios`: The BIOS program, which allows us to interact with our CPU via the UART.

  - `benchmark`:
    - `mmult`: This is a benchmark program to be run on the FPGA board. It generates 2 matrices and multiplies them. Then it returns a checksum to verify the correct result.
    - `bdd`: This is another benchmark program. It constructs binary decision diagrams for adder trees.
  - `small`: This directory contains a smaller version of each benchmark.
- `spec`: This specification document is located in this directory.
- `docs`: Documentation of your design goes to this directory.

3 Checkpoint 1: Block Diagram of RISC-V CPU

This checkpoint is designed to guide the development of a three-stage pipelined RISC-V CPU described in Section 4.

The second checkpoint will require significantly more time and effort than the first one. As such, completing the first checkpoint (block diagram) early is crucial to your success in this project.

Commit your block diagram and write-up to your team repository under `docs`.

3.1 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, but we strongly recommend writing it electrically. You can use Adobe Illustrator, Inkscape, Google Drawings, draw.io or any program you want. If working by hand, we recommend working in pencil and combining several sheets of paper for a larger workspace.

You should create 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. You should be able to describe in detail any smaller sub-blocks in your diagram.

Although 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. Therefore, we have one stage after IMEM, one stage after DMEM (write-back to register file at the end of this stage), and another stage in-between. Note that the stages before IMEM (e.g. program counter) are not included in the stage count. The reason behind is that Block RAMs in FPGA have a long clk-to-q delay and a small setup time.

3.2 Questions

Besides the block diagram, you will be asked to provide short answers to the following questions based on how you structure your block diagram. Write up your answers in any format. 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.

1. How many stages is the datapath you've drawn? (i.e. How many cycles does it take to execute one 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 prediction scheme are you using, or are you just injecting NOPs until the branch is resolved? If any prediction is done, what is the mispredict latency? What about data hazards in the branch?)
8. How do you handle jump control hazards? Consider jal and jalr separately.
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`?
11. What is the role of the CSR register? Where does it go?
12. When do we read from the BIOS memory 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?

4 RISC-V CPU Design Specification

4.1 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.

4.1.1 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 = 12'h51E`). That means you only need to instantiate one 32-bit register. The `tohost` register is monitored by the testbench, 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 = 12'h51E`)
2. `csrwi tohost,1` (short for `csrrwi x0,csr,uimm` where `csr = 12'h51E`)

`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).

4.2 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.

4.3 Hazards

As you have learned in lecture, pipelines create hazards. Your design will have to resolve both control and data hazards. It is up to you how to resolve the hazards. One way is stalling your pipeline and injecting bubbles (NOPs). Alternative way is implementing forwarding, which means forwarding data from your write-back stage.

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

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

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, jal (jump and link), or jalr (jump from register and link)? It is not a good idea to forward the branch result directly to the bios/instruction memories, as it will make a long critical path that limits the maximum frequency. You can resolve branches by stalling the pipeline or by a naive branch prediction. In the naive branch prediction, branches are predicted as either always taken or always not taken, and instructions are canceld when the prediction was wrong.

4.4 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.

4.5 RAMs

In this project, we are 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**.

4.5.1 Initialization

For simulation, the provided testbenches initialize the BIOS memory or instruction/data memory with a program specified by the testbench. The program counter will be initialized accordingly.

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

4.5.2 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).

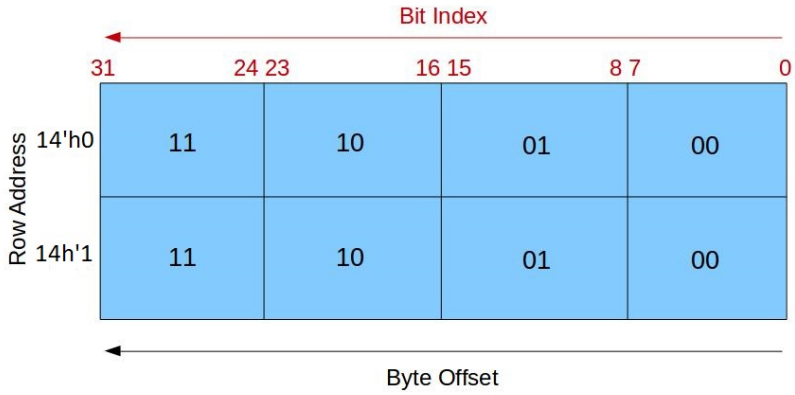

Figure 1: Block RAM organization.

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').

4.5.3 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).

4.5.4 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 `8'ha4` to address `32'h10000002` (byte offset = 2)
- Set `we = {4'b0100}`
- Set `din = {32'hxx_a4_xx_xx}` (`x` means don't care)

4.5.5 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.

4.6 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. Remember to assign their enebles properly in your logic.

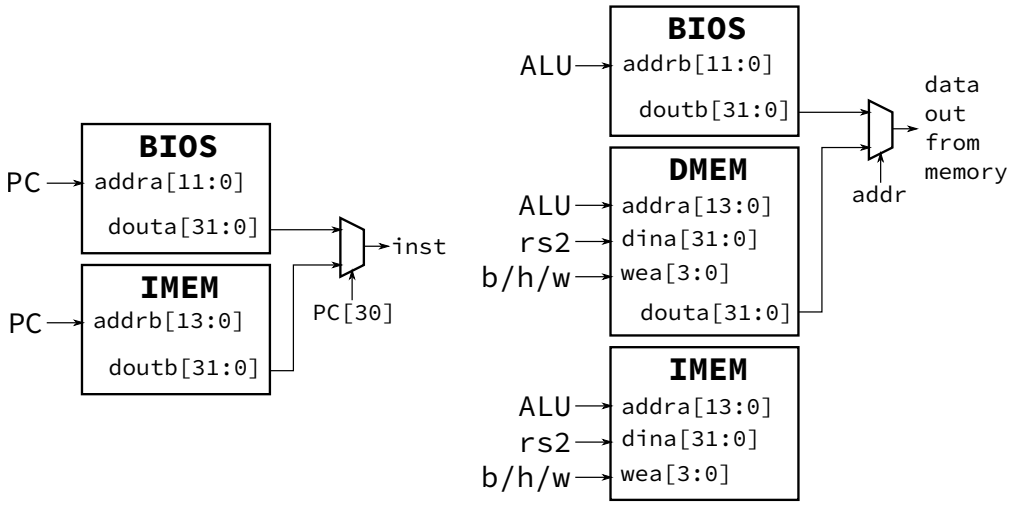

Figure 2: The Riscv151 memory architecture. There is only 1 IMEM and DMEM instance in your CPU but their ports are shown separately in this figure for clarity. The left half of the figure shows the instruction fetch logic and the right half shows the memory load/store logic.

4.6.1 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 (the right

most button) on the board to return your processor to the BIOS program.

4.6.2 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. According to this partitioning, 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 (`32'h40000000`).

The target memory/device is also dependent on the address type (PC or DATA). For example the address beginning with `4'h1` refers to the data memory when it is a data address, while it refers to the instruction memory when it is a program counter value.

Table 2: Memory Address Partitions

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

Here are some examples. When we are loading instructions, we are using a PC value as an address, and the instruction memory is read when it starts with `4'h1`, while the BIOS memory is read when it starts with `4'h4`. On the other hand, when we are loading data by load instructions, the address type is Data, and the data memory is read when it starts with `4'h1`, while the BIOS memory is read when it starts with `4'h4`.

One tricky thing is that a store to an address beginning with `4'h3` will write to both the instruction memory and the data memory, while storing to addresses beginning with `4'h2` or `4'h1` will write to either the instruction memory or the data memory, respectively.

4.6.3 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, 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

You should treat I/O such as the UART just as you would treat the data memory. The software checks the `uart_rx_data_out_valid` and `uart_tx_data_in_ready` signals by a load from `32'h80000000`, and proceeds to a load from `32'h80000004` or a store to `32'h80000008` if the corresponding valid or ready signal is 1. Then your datapath will fetch `uart_rx_data_out` or update `uart_tx_data_in`, while asserting `uart_rx_data_out_ready` or `uart_tx_data_in_valid`, respectively.

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.

Appendices

Appendix A BIOS

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

A.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.

A.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 `bios.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/bios` 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. If you can't get the BIOS carrot to come up, then your design is not working and you will have to fix it.

A.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 `bios.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 RISC-V 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:

```
./script/hex_to_serial.py <file name>
```

The `<file name>` field corresponds to the .hex file that you are to uploading to the instruction memory.

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

```
run
```

Note that the instruction memory size is limited in address size so large programs may fail to load.

A.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 `bios.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, the rightmost 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.

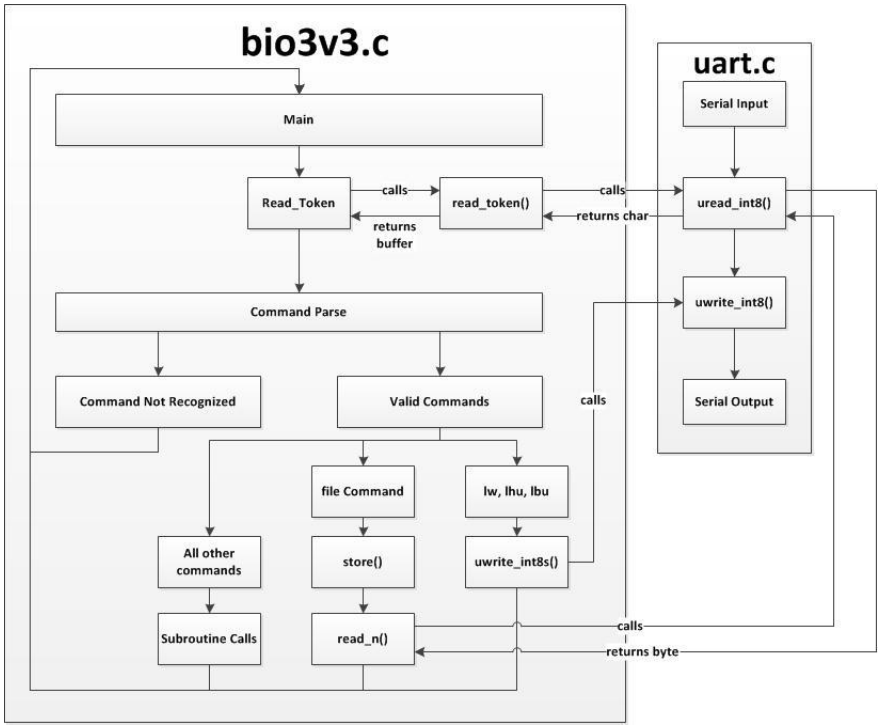

Figure 3: BIOS Execution Flow

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.

A.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.

A.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

`run` - Alias of `jal 10000000`

There is another command `file` in the main() method that is used only when you execute `hex_to_serial.py`. When you execute `hex_to_serial.py`, 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.

A.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 `bios.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.