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

EECS151/251A - LB, Fall 2017

Project Specification: RISCV151

Version 2.1

Contents

1 Introduction

The primary goal of this project is to familiarize EECS151/251A students with the methods and tools of digital design. In teams of 2, you will design and implement a complete 3-stage pipelined version of a RISC-V CPU. In addition to this, you will create a audio (AC97) controller that can stream audio samples to the headphone jack on the FPGA board. A functional implementation will be your primary goal. To better expose you to real design decisions and tradeoffs, however, we are requiring that you optimize your design for cost (FPGA resource utilization) and performance (maximizing the Iron Law).

You will use Verilog to implement this system, targeting the Xilinx XUPv5 platform (based around the Virtex 5 LX110T FPGA on the ML505 evaluation board). 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 our 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 potentially take significant time if you have not thought out your design prior to trying implementation. After you have built a working implementation, the next step will be optimizing it for area (cost, resource use) on the target FPGA. You will be expected to produce a relatively minimal circuit, implementing the required functionality, given a clock fixed at a certain frequency. At the end of this second phase (optimization, post implementation), you should have a greater understanding for the development process of digital hardware.

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 Philosophy

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

You should consider the GSIs a source of direction and clarification, but it is up to you to produce a fully functional design targeting the XUPv5 boards. We will attempt to help, when possible, but ultimately the burden of designing and debugging your solution lies on you.

There is the opportunity to extend our framework with additional functionality for extra credit. This is described in Section 4.5.

1.2 Tentative Deadlines

The following is a brief description of each checkpoint and approximately how many weeks will be alloted to each one. This schedule may change as the semester progresses.

- **Thursday, October 19** - Finish Lab 5 (AC97 Controller) and begin Lab 6 (UART Piano).
- **Thursday, October 19** - Checkoffs for Lab 6.
- **Thursday, October 26 - Checkpoint 0.5 (1 week)** - Design a high level schematic of your processor's datapath and pipeline stages.
- **Thursday, November 9 - Checkpoint 1 (2 weeks)** - Implement your RISC-V processor core in Verilog and write tests to verify the accuracy of your implementation.
- **Thursday, November 16 - Checkpoint 2 (1 week)** - Implement the audio (AC97) controller. Map user inputs into the processor address space and design a standard and clock-crossing FIFO.
- **Thursday, November 30 - Checkpoint 3 (2 weeks)** - Implement checkpoint 3 functionality (I2C master, video controller, TBA).
- **Friday, December 8 - Final Checkoff** - Final processor optimization and checkoff. Project report due.

1.3 General Project Tips

Make sure to use top-down design methodologies in this project. We begin by taking the problem of designing a basic computer system, modularizing it into distinct parts, and then refining those parts into manageable checkpoints. You should take this scheme one step further; we have given you each checkpoint, so break each into smaller and manageable pieces. If you follow this guideline, and our interface specifications, you should be able to split the project up between you and your partner.

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

- **Design** your modules well, make sure you understand what you want before you begin to code.
- **Code** exactly what you designed; do not try to add features without redesigning.
- **Simulate** thoroughly; writing a good testbench is as much a part of creating a module as actually coding it.
- **Debug** completely; anything which can go wrong with your implementation will.

Document your project thoroughly, as you go. You should never forget to comment your Verilog and to keep your diagrams up to date. Aside from the final project report (you will need to turn in a report documenting your project), you can use your design documents to help the debugging process. Finish the required features first. Attempt extra features after everything works well. **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 Pipelined RISC-V CPU - Checkpoint 1

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.

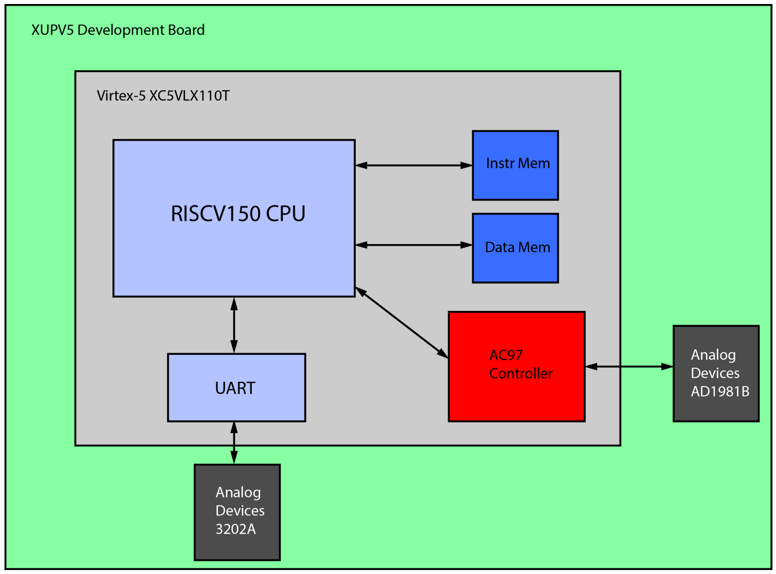

Figure 1: A high-level overview of the final system

The blue blocks on the diagram are the focus of the first checkpoint. Next, you will integrate the AC97 controller, shown in red. The dark gray blocks are the physical ICs on the board that your design on the FPGA will interact with.

2.1 Project Setup

The skeleton files for the project will be available through a git repository provided by the staff. The suggested way for initializing your repository with the skeleton files is as follows:

```
git clone git@github.com:EECS150/fa17_project_skeleton.git
cd fa17_project_skeleton
git remote add my-repo git@github.com:EECS150/fa17_teamXX.git
git push my-repo master
```

This will make a single commit to your repository with the base files, we suggest you then do the following:

```
cd ..
rm -rf fa17_project_skeleton
git clone git@github.com:EECS150/fa17_teamXX.git
cd fa17_teamXX
git remote add staff git@github.com:EECS150/fa17_project_skeleton.git
```

These commands will delete the skeleton repository you cloned, clone your repository that now has a single commit, and add a remote repository named `staff` that points to the skeleton files to allow easy future merges of staff updates.

Note that if you havent emailed your GSI with your group information (names and Github logins) you will not have a Git repository to save your work in so do that ASAP.

2.2 Integrate Designs from Lab

Here are some commands to add a modules from a previous lab to your git repository.

```
cd fa17_project_skeleton/hardware/src/audio/.
cp ~/labs_fa17/lab5/src/ac97_controller.v .
git add ac97_controller.v
git commit -m "Adding AC97 controller from lab"
git push
```

The `git add` command tells your local git repository to add the file `ac97_controller.v` to the list of tracked files or stage it for the next commit. The `git commit` command tells your local repository to actually commit any of the files you added and create a new change set with the changes in the files you added. Finally the `git push` command sends your new local commit to the remote repository so that you or your partner can pull it later.

Going forward, any update to project files will be made through the `fa17_project_skeleton` repository. If you have followed the setup described here, then you can just do a `git pull staff master`, and the latest changes will be fetched and automatically merged in.

Here are the files you should copy over from previous labs:

```
cp labs_fa17/lab5/src/debouncer.v fa17_project_skeleton/hardware/src/io_circuits/.
cp labs_fa17/lab5/src/synchronizer.v fa17_project_skeleton/hardware/src/io_circuits/.
cp labs_fa17/lab5/src/edge_detector.v fa17_project_skeleton/hardware/src/io_circuits/.
cp labs_fa17/lab5/src/rotary_decoder.v fa17_project_skeleton/hardware/src/io_circuits/.

cp labs_fa17/lab6/src/fifo.v fa17_project_skeleton/hardware/src/io_circuits/.
cp labs_fa17/lab6/src/async_fifo.v fa17_project_skeleton/hardware/src/io_circuits/.
cp labs_fa17/lab6/src/uart.v fa17_project_skeleton/hardware/src/io_circuits/.
cp labs_fa17/lab6/src/uart_receiver.v fa17_project_skeleton/hardware/src/io_circuits/.
cp labs_fa17/lab6/src/uart_transmitter.v fa17_project_skeleton/hardware/src/io_circuits/.
```

We have provided an organization structure for the `hardware/src` folder, but you are free to change it as you please. We only ask that you leave the `testbenches` and `memories` folder in place to accommodate future staff repo merges.

2.3 Relevant Files and Scripts

To only **synthesize** your design, go to the `hardware/` directory and run `make synth`. To **build** your entire design run `make`. To **program** the FPGA, run `make impact` from this same directory.

To **simulate** your design, go to the `hardware/` directory and run `make sim`. To run a specific testbench, you can run `make CASES=tests/fifo_testbench.do` in the `hardware/sim` directory.

The following are located in the `hardware/src` directory:

- `ml505top.v`: Top level file. Your Riscv151 CPU module is instantiated here. You should not need to modify this, but looking at this file can be helpful for understand what's going on.
- `ml505top.ucf`: Constraints file. This specifies constraints for the synthesis tools. You should not need to modify this.

In `hardware/src/riscv_core`:

- `Riscv151.v`: All of your CPU datapath and control should be contained in this file.
- `reg_file.v`: Your register file implementation

In `hardware/src/memories`:

- `imem_blk_ram`: Contains the block RAM you will use for your instruction memory. Make sure to copy over the bios151v3.coe file and run the build script to generate the Verilog file.
- `dmem_blk_ram`: Contains the block RAM you will use for your data memory. Make sure to copy over the bios151v3.coe file and run the build script to generate the Verilog file.
- `bios_mem`: Contains the block RAM you will use for your BIOS memory. Make sure to copy over the bios151v3.coe file and run the build script to generate the Verilog file.

In `hardware/src/io_circuits`:

- `uart.v, uart_transmitter.v, uart_receiver.v`: Your working solution from Lab 4

In `hardware/src/testbenches`:

- `echo_testbench.v`: Basic testbench for your CPU. Use this as an example and create others. It implements the echo FSM from lab 4 using software executed by your processor.

The following are located in `software/` directory. To compile the C code, go into one of the directories and run `make`:

- `bios151v3`: This directory contains the BIOS, which will allow us to interact with our CPU via the serial UART. Make sure to compile it and copy over the `.coe` file to the two block ram directories.
- `echo`: This directory contains the software necessary to run the echo program, which behaves exactly like in Lab 4.
- `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 1. It generates 2 6x6 matrices and multiplies them. Then it returns a checksum to verify the correct result.
- `ac97_basic_test`: This is a program to be run on the FPGA for Checkpoint 2.

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. This processor will not support floating point, coprocessor, memory fence, and several other instructions that are of little utility for this project but would greatly complicate the design. For the specific details of each instruction, refer to sections 2.2 through 2.6 in the RISC-V Instruction Set Manual. You may also find the RISC-V green card helpful when implementing your CPU.

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

RV32I Base Instruction Set

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

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 block RAMs that we will be using for the data and instruction memories are both synchronous read and write.

2.6 Hazards

As you have learned in lecture, pipelines create hazards. Your design will have to resolve both control and data hazards. This is a very common source of bugs, so think through and test this carefully. You must resolve data hazards by implementing forwarding whenever possible. This means that you must forward data from your data memory and not stall your pipeline instead. 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 computation 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 should begin by resolving branches by stalling the pipeline, and when your processor is functional, move to naive branch prediction.

2.7 Register File

Your register file should have two asynchronous-read ports and one synchronous-write port (positive edge). To test your register file, you should verify the following:

- Register 0 is not writable, i.e. reading from register 0 always returns 0
- Other registers are updated on the same cycle that a write occurs (i.e. the value read on the cycle following the positive edge of the write should be the new value).
- The write enable signal to the register file controls whether a write occurs (`we` is active high, meaning you only write when `we` is high)
- Reads should be asynchronous (the value at the output one simulation timestep (#1) after feeding in an input address should be the value stored in that register)

After you build your design, look for warnings in the report (`make report`) about the register file. Occasionally, the tools infer a block RAM rather than distributed (slice) RAM. This will not show up in simulation, but it will cause synchronous reads on hardware. To fix this, you can add a flag to your register file:

```
(* ram_style = "distributed" *) reg myReg...
```

For more information on how to infer various structures on the FPGA, see Xilinx Synthesis and Simulation Design Guide.

2.8 Block RAMs

In this project, we will be using generated block RAM modules to implement memory structures for the processor.

2.8.1 Initialization

Inside of `hardware/src/memories/imem_blk_ram, dmem_blk_ram, bios_mem` there are three skeleton files:

- `*.xco`: This file contains configuration information used by coregen to build the memory. The only attribute you may need to change is the `coe_file` on line 46. To initialize the memories with a particular program for synthesis, set this field to point to the desired `.coe` file and re-generate the memories. **This does not initialize the memory for simulation, it is for FPGA implementation only**

  Tip: copy the `.coe` file you want to use into the directory where the `.xco` file resides.

- `build`: Running `./build` generates the memory based on the configuration information. Run this if you change the parameters in the `.xco` file. You must run this if you decide to use a different `.coe` file.

- `clean`: Run `./clean` to delete the files created when you generate the memories. Do not run this from the GUI or any other directory!

The skeleton files contain two programs that you will likely want to initialize your memories with: `bios151v3` and `echo`. The bios is significantly more complicated, so while debugging, you may want to stick with `echo` until it works on the hardware. As previously mentioned, you **must** compile one of the software applications, copy the `.coe` file into the block ram directories, and re-run the build scripts. If you don't copy over the `.coe` file, the build scripts will fail to generate the block ram cores and you will get errors when trying to synthesize your processor. The `.coe` file is called a coefficients file and it describes the initial contents of a memory

2.8.2 Usage in Simulation

To make the block RAMs work in simulation, build them using any `.coe` file you desire. **However, in simulation, the initial contents of the memory aren't specified by the .coe file, but rather a .mif file.** A `.mif` file is generated by running `make` in any software directory.

In simulation, the memories are initialized with a `.mif` file. The software toolchain generates these for you; look at the `hardware/sim/test/echo_testbench.do` file for examples of how to use these.

2.8.3 Endianness + Addressing

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

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

For us, the bottom 16 bits of the addresses computed by the CPU are relevant for block RAM access. The top 14 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).

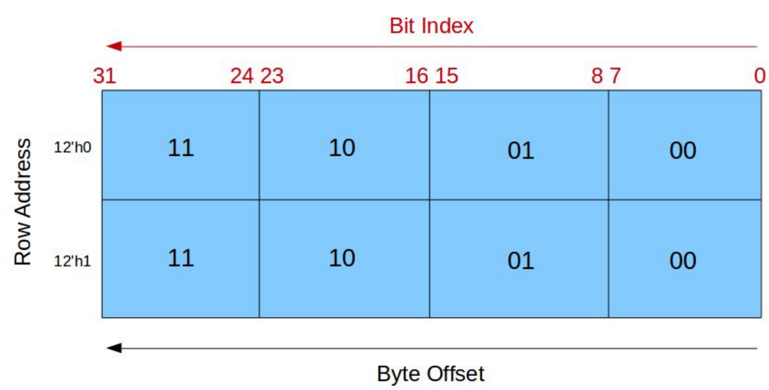

Figure 2: Block RAM organization. The labels for row address **should read 14'h0 and 14'h1.**

Figure 2 illustrates the 14-bit word addresses and the two bit byte offsets. Observe that the RAM is **little-endian**, i.e. the most significant byte is at the most significant memory address (offset '11').

2.8.4 Reading from Block RAMs

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

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

2.8.5 Writing to Block RAMs

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

Here is an example how storing a byte should work: Assuming you want to write the byte `a4` to address `0x10000002`. That is a write to the third byte of the first word of data memory. Accordingly, the write enable bits should be set to `we = {4'b0100}` and the data should be `dina = {32'hxxa4xxxx}` where x is used in the meaning of dont 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 instruction memory to run new programs. Your CPU, by the end of this checkpoint, will be able to receive compiled RISC-V binaries though a serial interface (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 3:

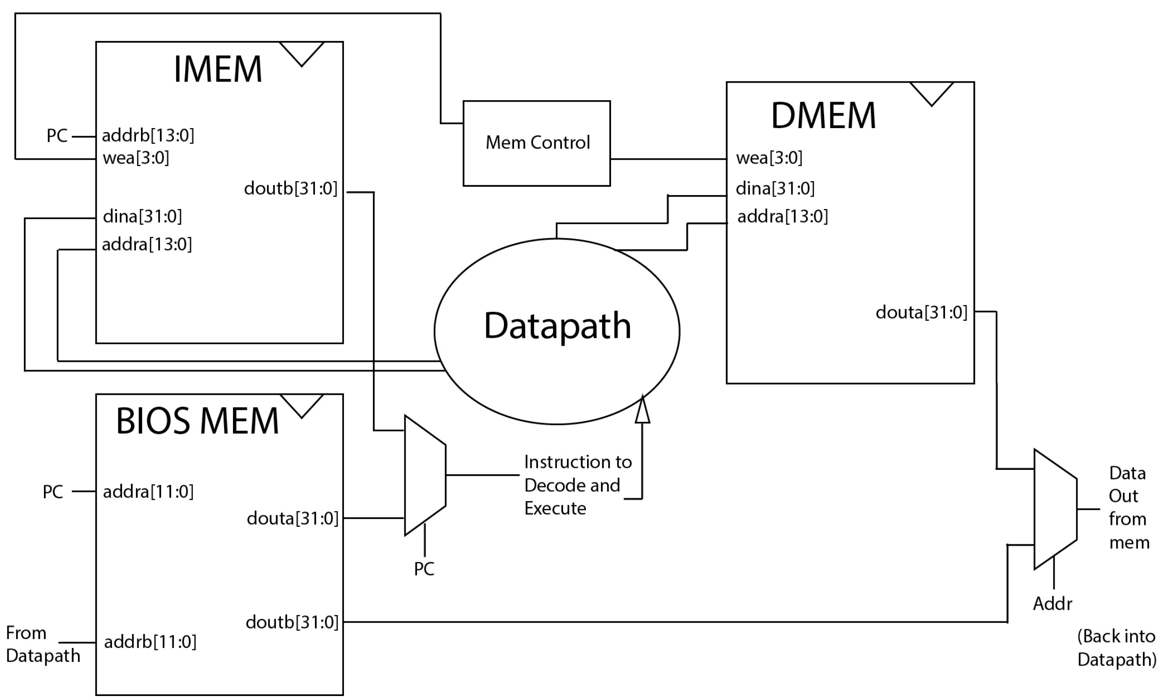

Figure 3: Memory Architecture

See Section 2.8 for details about initializing and using the block RAMs. To simplify things for later, you are **required** to instantiate these block RAMs in the top level of `src/riscv_core/Riscv151.v`.

2.9.1 Summary of Memory Access Patterns

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

Your processor will begin execution from the BIOS memory, which will be instantiated with a BIOS program we wrote using a `.coe` file. The BIOS program will have the ability to read from the BIOS memory (both static data and instructions), and to read and write to and from instruction and data memory. This allows the BIOS program to receive user programs over the serial line (UART) from your workstation and load them into instruction memory. You can then instruct the BIOS program to jump to an instruction memory address, which will then begin 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 do not need to deal with unaligned memory accesses if they would require multiple block RAM accesses. You can ignore instructions that request an unaligned access or you can zero out the byte offset.

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 main categories: data memory read and writes, instruction memory writes, BIOS memory reads, and memory-mapped I/O. This will be encoded in the top nibble of the memory address generated in load and store operations, as shown in Table 2. In other words, the target device of a load or store instruction is dependent on the address. For this checkpoint, the reset signal should reset the PC to the start 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]
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 only 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 2.13.

Please note that a given address maybe refers 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.

This memory map is designed to allow a few important features:

- **Initialization**: The top nibble of the PC should start at `0x4` upon reset. This lets us press the reset button to jump into BIOS memory execution.
- **Reprogrammable**: When running from the BIOS, the instruction memory can be written to. Our BIOS program listens to the UART and if it detects the transmission of a binary, it will receive it and store it to the instruction memory. **When downloading a program to the CPU, store it to an address beginning with 0x3 for coherence between the memories.**

2.9.4 Memory Mapped I/O

At this stage in the project the only way to interact with your CPU is through the serial interface. The UART from Lab 4 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 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, DataOutValid, DataInReady}
32'h80000004	UART receiver data	Read	{24'b0, DataOut}
32'h80000008	UART transmitter data	Write	{24'b0, DataIn}
32'h80000010	Cycle counter	Read	Total number of cycles
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. Essentially, you will want to set the output valid/ready/data signals from the CPU based on the calculated load/store address, and mux in the appropriate valid/ready/data signals to be written back to the register file. Keep in mind that your UART's ready-valid interface is synchronous, so you will need to set the appropriate control signals for a write or read before the rising edge on which the operation should execute. 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.

The cycle counter should be incremented every cycle, and the instruction counter should be incremented for every instruction that is run (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. Although we will not require or grade testing efforts, we expect that teams utilizing the testing tools will be able to complete checkpoints faster. Furthermore, in assigning partial credit at the end for incomplete projects, we will look at testing as an indicator of progress. We strongly encourage that you follow the suggestions here for testing. 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 an assembly program — see `assembly_testbench.v`
3. Test the CPU's memory mapped I/O — see `echo_testbench.v`

2.10.1 Simulation

You learned how to write Verilog testbenches and simulate your Verilog modules in the labs. You should use what you learned, to write testbenches for all of your sub-modules. As you design the modules that you will use in your CPU, you should be thinking about how you can write testbenches for these modules. For example, you may want to create a module that handles all of the branching logic.

To run new simulations, you should write new testbenches and put them in your `hardware/src/testbenches` directory. Then you should create new `.do` files in the `hardware/sim/tests` directory. The staff have provided you with `echo_testbench.v` and the corresponding `.do` file. When you run `make sim` in the `hardware` directory, all of the `.do` files in the `sim/tests` directory will run. If you only want to run one test, run `make CASES=tests/echo_testbench.do` in the `hardware/sim` folder.

Just like in the labs, you can debug your logic by looking at waveforms. After running your simulations run the `viewwave` script in the `hardware/sim` directory, passing in the `.wlf` file as an argument:

```
./viewwave results/echo_testbench.wlf
```

2.10.2 Modifying the .do Scripts

In `echo_testbench.do` you will see lines that look like this:

```
file copy -force ../../../software/echo/echo.mif imem_blk_ram.mif
add wave echo_testbench/*
add wave echo_testbench/CPU/*
```

The first line initializes the contents of `imem_blk_ram` to whatever values you have in `echo.mif`. The `.mif` files are generated when you compile software from the `software` folder, and they describe the initial values in a block RAM for ModelSim. You can change the filepath to point to a different `.mif` file, and you can specify different memories you want to initialize. The second line tells Modelsim to collect data for all the signals in the top level of `echo_testbench`. The third line tells it to also get data for all signals **inside** the CPU, assuming you instantiated your CPU inside `echo_testnench.v` and called it `CPU`. You could also add a line like:

```
add wave echo_testbench/CPU/mySubModule/*
```

which would allow you to see all signals inside a submodule called `mySubModule` in the CPU. Note that the name `mySubModule` should correspond to the unique name which you instantiated that submodule with (i.e. if you instantiated a UART called `myUART`, you want to use `myUART`). In this way, you can add all of the signals you want to inspect to your waveform viewer for debugging.

2.10.3 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 pieces of the toolchain are given below:

- `riscv-gcc`: gcc for RISC-V, compiles C code to RISC-V binaries.
- `riscv-as`: RISC-V assembler, compiles assembly code to RISC-V binaries.
- `riscv-objdump`: Displays contents of RISC-V binaries in a readable format

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

There are several files in the example project, each with a specific purpose:

- `start.s`: This is an assembly file that contains the start of the program. It initializes the stack pointer then jumps to the `main` label. Edit this file to move the top of the stack. You will have to move the top of the stack to give it some room to grow downwards.
- `c_example.ld`: This linker script sets the base address of the program. For checkpoint 1, this address should be in the format `0x1000xxxx`
- `c_example.elf`: Binary produced after running `make`. Use `riscv-objdump -D c_example.elf` to view the assembly code.
- `c_example.asmdump`
- `c_example.mif`: Produced by the toolchain. Use this to initialize the block RAMs in ModelSim.
- `c_example.coe`: Produced by the toolchain. Use this to initialize the block RAMs during generation (coregen).

2.12 Assembly Tests

This section describes the contents of `software/assembly_tests`. You can test individual instructions in simulation with a program similar to the following example in `assembly_tests/start.s`:

```
_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
```

This testbench works as follows: During the execution of the program you should load different values to the flag register (default: x20). The principle of the Verilog testbench is to wait for the flag register to assume the flag value and afterwards check the registers that hold the results of our operation. If the testbench does not print any output and exits with `<EOF>` it indicates the test timed out and that the `li` instruction does not work properly or that you jumped over this instruction. Note that `li` is a psuedo-instruction that compiles down to a combination of `addi` and `lui`.

Note that RISC-V assembly syntax is slightly different than MIPS, which many of you may be used to. In particular, register names do not all start with the $, and all registers are referenced as x0. . . 31.

Follow the directions from Section 2.8 and Section 2.11 to assemble your test program and run it in simulation. You will also need to write a Verilog testbench that instantiates the CPU and perhaps has helpful `$display` statements to help you debug your CPU. An example is provided for this program in `hardware/src/testbenches/assembly_testbench.v` and the respective `.do` file is in `hardware/sim/tests/assembly_testbench.do`.

Writing tests that can self-verify with either a pass or fail, rather than ones that require you to open up the wave viewer to manually verify, are preferred, as they will make it easier for you to come back and run these tests later.

2.12.1 RISC-V Assembly Syntax

To write load or store instructions:

```
li x1, 0x10000000
li x2, 0xdeadbeef
sw x2, 0(x1)     # Store Reg[x2] at Mem[Reg[x1] + 0]
sw x2, 4(x1)     # Store Reg[x2] at Mem[Reg[x1] + 4]
lw x3, 4(x1)     # Load Mem[Reg[x1] + 4] to x3
```

To write branch or jump instructions:

```
loop: addi x1, x1, 1
j loop                  # Unconditionally jump to 'loop' label
jal x3, loop            # Jump to 'loop' and link x3
beq x2, x0, loop        # Branch to 'loop' if Reg[x2] == 0
```

2.13 BIOS and Programming your CPU

We have provided a `BIOS` program in `software/bios151v3` that allows you to interact with your CPU and bootstrap into other programs over the serial interface. This compiled C program is basically just an infinite loop that reads from the serial port, checks if the input string matches a known control sequence, and then performs the action. For more detailed information on the BIOS, check out this supplement.

To use this, do the following steps:

1. Verify that the stack pointer and .text segment offset are set properly in `start.s` and `bios151v3.ld`. See Protips for details.
2. Compile the program with `make` in the `software/bios151v3` directory
3. Rebuild your instruction, data, and BIOS memories with the generated `.coe` file by copying the `.coe` file into the `hardware/src/memories/{imem, dmem}_blk_ram,bios_mem` directories, and then re-running the `build` script
4. Build your CPU and impact it to the board
5. Press the CPU_RESET button to reset your CPU
6. As in lab 4, use screen to access the serial port: `screen $SERIALTTY 115200`

Please remember to shut down 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 create 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 also allows you to load programs to the CPU. Close screen using ctrl-a shift-k, and execute in the terminal:

```
coe_to_serial <coe_file> <address>
```

This stores the `.coe` file at the specified hexadecimal address. In order to write into both the data and instruction memories, remember to set the top nibble to 0x3 (i.e. `coe_to_serial echo.coe 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 have set the set the base address to `0x10006000` (see 2.16). Therefore, when loading the `mmult` program to the FPGA you should place it into the memory that it starts aligned with the base address: `coe_to_serial mmult.coe 30006000`. Then, you can start in in your screen session by using `jal 10006000`.

2.14 Target Clock Frequency

By default, the minimum clock period is set at 50MHz. With a good design, you should be able to meet this constraint, though it may take some tweaking. At the end of the `make` run, pay attention and make sure it says that you met all timing constraints. If you failed, the timing reports in `make report` can give you a good idea of where your critical path is so you can attempt to 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 (up to 100Mhz) for full credit. Details on how to build your FPGA design for a different system clock will come later.

2.15 Git

You should check often for updates to the skeleton files. Update announcements will be posted to Piazza and emailed to all students. As previously stated, you can pull them into your repository, assuming you have correctly followed the configuration instructions, by issuing this command from a directory in your repository:

```
git pull staff master
```

2.16 Matrix Multiply

For us to check the behavior of your processor we have provided a program called mmult (in `software/mmult/`) which performs matrix multiplication. You should be able to load it into your processor in the same manner 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 the Memory Mapped IO section. 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) needs to accommodate three 64×64 matrices as well as additional space for temporary results. It should be set to `0x10006000`.

The .text segment offset (set in mmult.ld) needs to accommodate the full set of instructions and static data in the mmult binary. It should be set to `0x10006000`.

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, and ideally should be under 1.15. 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 Tips

In previous iterations of this project, students have struggled with the following issues:

- **Off by one errors**. These occur in many different forms, but are usually the result of not thinking carefully about the timing of your design. It is important to understand the difference between synchronous and asynchronous elements. The synchronous elements in your design include the UART, Block RAMs for data and instruction memory, registers, as well as the register file (synchronous write only).
- **Memory mapped I/O**. As the name implies, 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 in the memory stage. The CPU itself should not check the valid and ready signals; this check is handled in software.

- **Byte/halfword/word and endianness**. Read the block RAM section 2.8.3 carefully, and ask questions if you are confused at all.
- **Incorrect control signals**. A comprehensive assembly test program will help you systematically squash bugs caused by incorrect control signals.
- **Mismatched bus widths**. It is a fairly common error to instantiate a wire or reg with the wrong bus width. If you hook up a 1 bit wire to a driver that is 32 bits, it will still be syntactically correct, but it probably wont work. Pay attention to the synthesis warnings, as they will advise you if you have mismatched bus widths.
- **Incorrect hazard logic**. Make sure you write carefully crafted tests which will stress test the forwarding behavior.
- **ALU inputs**. Check to make sure that the inputs you are feeding into the A and B inputs of your ALU reconcile the way you coded your ALU. Remember that A and B are not symmetric inputs, and you need to feed specific datapath elements to each for correct operation.
- **PC Width**. Check to make sure that the width of your PC accommodates the top nibble which contains the memory partitioning info for a particular address.
- **JALR**. The JALR instruction is commonly used, but will be especially stressed in the mmult program. Make sure your implementation is robust and can handle forwarding data dependencies.
- **Reset Logic**. Make sure that when the reset signal is asserted that all your pipeline registers are cleared so that no erroneous writes occur. Also check any register resets internal to your submodules.
- **Stack pointer and .text segment offset**. Make sure that your stack pointer is set to near the top of the data memory address space, so that the stack has enough room to grow downwards. Also verify that the .text segment offset (in the .ld files) is set properly to give the code and static data enough room as well.

2.18 How to Survive This Checkpoint

The key to this checkpoint will be to start early and work on your design incrementally. This project is not something that can be done with an all nighter and we can almost guarantee that you will not finish if you start two or three days before the due date. The key to this checkpoint will be to draw up a very detailed and organized block diagram and thoroughly understand all parts of the specification. Groups that have been successful in the past usually have unit test cases that thoroughly test every module and progressively larger integration tests. We recommend for your final integration test of the whole system that you write individual programs that thoroughly test the behavior of each instruction. The final BIOS program that you will be required to run is several 1000 lines of assembly and will be nearly impossible to use for debugging by just looking at the Modelsim waveforms.

We also encourage groups to work together and bounce ideas off of each other. 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. We highly recommend getting adequate sleep during the weeks of this checkpoint. We realize there are not windows or clocks in the lab so its very easy to get carried away and work into the early morning in the lab. If you find yourself spinning your wheels, its probably time to go home and sleep a bit before trying again.

2.19 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:

1. **Design.** You should start with a comprehensive and detailed design/schematic. We suggest that you think carefully about all the functionality and instructions your processor needs to support and enumerate all the control signals that you will need. Be especially careful when designing the memory fetch stage of your pipeline as all the memories we use (BIOS, inst, data, IO) are synchronous.
2. **First steps.** You should get started by implementing some modules that are straightforward to write and test. We suggest you get started by writing `reg_file.v`, for which there has been a template provided in the project skeleton. Once you finish writing the regfile, test it comprehensively by writing a Verilog testbench. Look at the Register File section for details on what the test should verify.
3. **Control Unit + other small modules.** Next try implementing your control unit, the ALU, and any other small independent modules that you identified in your design. Make sure you unit test these aggressively, so that you verify their correctness and get used to writing Verilog testbenches.
4. **Memory.** Create your memory controller and other auxiliary structures. Only add the BIOS memory in the instruction fetch stage and only add the data memory block RAM in the memory stage of your pipeline. This will keep things simple in order to test the base functionality of your processor.
5. **Connect stages and pipeline.** Now you should have all of the modules ready to connect them together and pipeline them by inserting registers between the stages. 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.** Now 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 your control instructions work properly with assembly tests. You can insert explicit NOP instructions in your tests to get around data dependencies.

7. **Implement data forwarding for data hazards.** Add forwarding muxes to the proper place in your datapath and forward the outputs of the ALU and memory stage. Implement a hazard unit that can detect data dependencies and set the control signals for the forwarding muxes accordingly. 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. Make sure you 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. It is crucial to write tests to stress this portion of the processor; we suggest writing tests that first write instructions to the instruction memory, and then jump (using jalr) to instruction memory to see 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. This part can be tricky, ask a TA for a full explanation of how a program would communicate with the UART. We have provided you with the `echo_testbench` which performs a test of the UART. You should extend this testbench with more comprehensive tests, as many bugs can be traced to a faulty UART integration.

12. **Run the BIOS.** If everything so far has gone well, you can try making the CPU with instantiating the BIOS memory with the BIOS program. Impact the CPU on the board and verify that the BIOS performs as expected. As a precursor to this step, you might try to make the CPU with instantiating the BIOS memory with the echo program, since it is a smaller and easier to analyze program.

13. **Run matrix multiply.** As a final step to check your implementation, you should be able to load the `mmult` program with the `coe_to_serial` utility, and run `mmult` on the FPGA. Verify that it returns the correct checksum.

14. **Check CPI.** Now that your processor is complete as far as functionality goes, 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. With this step complete, you are ready for the next checkpoint.

2.20 Checkoff

The checkoff for this specification 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.

2.20.1 Checkpoint 0.5: 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. You should also be able to describe in detail any smaller sub-blocks in your diagram. If working electronically, you can use a schematic capture program, Logisim, or anything that can produce a diagram that is easily modifiable. Though the textbook diagrams are a decent starting place, please remember that they use asynchronous-read memories for the instruction and data memories, and we will be using synchronous-read block RAMs. Additionally, at this point we recommend that you have a completely functional UART, ALU, ALU decoder, and register file modules (see 2.7), though we will not be checking this.

Checkpoint 0.5 is due in lab no later than 8:00 PM Thursday, October 26 You are required to go over your design with a GSI during lab. Be prepared to talk generally about how you came up with your design and defend your design decisions.

2.20.2 Non-Checkpoint Weeks

In labs, you probably found that you spent significantly more time debugging and verifying your design than actually writing Verilog. Though your skills are continually improving, this project involves a complex system and as such, bugs are inevitable. Design verification can take more than twice as long as writing the initial implementation. Given this, we recommend that you have completed your first stab at writing the Verilog and associated module testbenches for your processor by the end of this week.

2.20.3 Checkpoint 1: 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, storing a program (echo and mmult) over the serial interface, and successfully jumping to and executing the program.

Checkpoint 1 materials should be committed to your project repository by 8:00 PM Thursday, November 9.

2.20.4 Checkpoint 1 Deliverables Summary

Deliverable	Due Date	Description
Block Diagram	Thursday, October 26 @ 8:00 PM	Sit down with a GSI and go over your design in detail
RISC-V CPU	Thursday, November 9 @ 8:00 PM Check in code to Github	Demonstrate that the BIOS works, you can use `coe_to_serial` to load the echo program, `jal` to it from the BIOS, and have that program successfully execute. Load the mmult program with `coe_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 Final Checkpoint - Optimization

This optimization checkpoint is lumped with the final checkpoint and the checkoff will occur at the same time. 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 general optimization goal for this project is to achieve maximal performance on the `mmult` program, as defined by the 'Iron Law' of Processor Performance.

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

Your goal is to minimize the execution time of `mmult`. 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.

3.1 Clock Generation Info + Changing Clock Frequency

Open up `ml505top.v`. You will notice a top level input called `USER_CLK`. This signal comes from a crystal on the ML505 board and it comes into our FPGA design. It is a 100 Mhz clock signal, which we will use to derive our CPU clock.

Scrolling down a little further, you will see an instantiation of `PLL_BASE`, which is a PLL (phase locked loop) primitive on the FPGA. This is a circuit that lets us create a new clock from a known clock with a user-specified multiply-divide ratio.

The `CLKIN` input clock of the PLL. is driven by the 100 Mhz `user_clk_g` (buffered `USER_CLK`). The PLL multiplies the frequency of this input clock by the `CLKFBOUT_MULT` parameter, which is set to 6. Thus, internally, the PLL creates a 600 Mhz clock. Then, this multiplied clock is divided by the `CLKOUT0_DIVIDE` parameter. In our case, this parameter is set to `600_000_000 / CPU_CLOCK_FREQ` and `CPU_CLOCK_FREQ = 50_000_000`. Thus our divide parameter is 12. Finally, the multiplied and divided clock ($\times 6 \div 12 = \frac{1}{2}$) shows up at the `CLKOUT0` output clock of the PLL, which is connected to `cpu_clk`. The `cpu_clk` is buffered and `cpu_clk_g` is used in our CPU and other modules.

Take a look at the `CPU_CLOCK_FREQ` parameter at the top of `ml505top`. This sets the clock frequency we want to synthesize from the 100 Mhz `USER_CLK` to be used in the rest of our design. You can alter this parameter to change the CPU clock frequency, but it can't be set arbitrarily, and there are a few caveats. You can only set this value to an integer divisor of 600 Mhz, unless you change the multiplication parameter in the PLL. A few frequencies to try are: 60 Mhz, 75 Mhz, and 100 Mhz. You can also try frequencies in the middle and even adjust the multiply parameter for more variety.

3.2 Critical Path Identification

Begin by pulling the latest skeleton files from the staff repository: `git pull staff master`. After running `make`, your FPGA design will be placed and routed, and timing analysis will be performed to determine the critical path(s) of your design. The timing tools will automatically figure out the CPU clock timing constraint based on the multiply-divide ratio you used in your PLL.

To see the critical path run `make report`, and click on `Post-PAR Static Timing Report` in the list on the left. You are interested in the timing paths for `cpu_clk_g` which is the clock used by your CPU and the rest of your design.

Perform a CTRL+F on `TS_cpu_clk` and you will come upon a section like this in your report:

```
Timing constraint: TS_cpu_clk = PERIOD TIMEGRP "cpu_clk" TS_USER_CLK / 0.5 HIGH
50%;
For more information, see Period Analysis in the Timing Closure User Guide
 ↪  (UG612).

73245958 paths analyzed, 6219 endpoints analyzed, 0 failing endpoints
0 timing errors detected. (0 setup errors, 0 hold errors, 0 component switching
 ↪  limit errors)
Minimum period is  19.897ns.
```

This section indicates the start of the timing report for the CPU clock. It will tell you how the timing constraint for the `cpu_clk` was derived from the `TS_USER_CLK` constraint via the PLL multiply-divide ratio, and it will tell you if there are any timing (setup or hold) errors.

As you scroll down, you will find a list of 100 timing paths that have the smallest timing slack. They will look something like this:

```
Slack (setup path): 0.103ns
Source: CPU/dataMem/...
Destination: CPU/instrMem/...
Requirement:          20.000ns
Data Path Delay:      19.352ns (Levels of Logic = 12)
Source Clock:         cpu_clk_g rising at 0.000ns
Destination Clock:    cpu_clk_g rising at 20.000ns

Location               Delay type         Delay(ns)  Physical Resource
Logical Resource(s)
----------------------------------------------------  -------------------
RAMB36_X0Y26.DOADOU0   Trcko_DOWA            2.180   CPU/dataMem
SLICE_X11Y110.D1       net (fanout=1)        2.302   CPU/dataMem/ram_douta
SLICE_X11Y110.D        Tilo                  0.094   CPU/Data_In_dmem<25>
SLICE_X11Y110.C6       net (fanout=1)        0.139   CPU/Data_In_dmem<25>
SLICE_X11Y110.C        Tilo                  0.094   CPU/dpath/Data_In<25>1
SLICE_X23Y106.B2       net (fanout=3)        1.291   CPU/dpath/Data_In<25>
...
```

```
SLICE_X29Y100.B2        net (fanout=69)       1.677   CPU/dpath/ALU_rd2_E<1>
SLICE_X29Y100.B         Tilo                  0.094   CPU/dpath/ALUCompute/Out
SLICE_X28Y97.D1         net (fanout=9)        1.064   CPU/dpath/ALUCompute/Out
SLICE_X28Y97.D          Tilo                  0.094   CPU/dpath/ALUCompute/Out
...
SLICE_X23Y92.D          Tilo                  0.094   CPU/dpath/next_PC_F<10>
SLICE_X23Y92.C6         net (fanout=9)        0.154   CPU/BiosAddr<10>
SLICE_X23Y92.C          Tilo                  0.094   CPU/dpath/next_PC_F<10>
RAMB36_X2Y15.ADDRBL11   net (fanout=32)       3.593   CPU/InstrAddr<10>
RAMB36_X2Y15.CLKBWRCLKL Trcck_ADDRB           0.347   CPU/instrMem
----------------------------------------------------  ---------------------------
Total 19.352ns (3.953ns logic, 15.399ns route) (20.4% logic, 79.6% route)
```

The first attribute for a timing path is the slack. Slack describes how much extra time the combinational delay of the path has before the rising edge of the receiving clock. It 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.

You will then see the source and destination of the path which you can usually map to a net in your design. Afterwards, comes the actual logic path that starts at the source and follows some logic in your design until it gets to the destination. In the example above, you can make out that the logic path represents data memory forwarding into the ALU, which is then used for a branch/jal/jalr next PC calculation.

There are 3 common delay types that you will encounter during optimization. Most of the `Trc*` delays are RAM delays that represent either Clk-to-q delays or setup time constraints. `Tilo` delays are combinational delays through LUTs. `net` delays are routing delays. If you want details on a specific delay type, check the Virtex 5 Datasheet starting from page 40.

`net` delays include a fanout attribute. You will likely want to minimize fanout of a given net along a timing path in order to reduce routing delay. You will notice that as a percentage of total delay, routing dominates over combinational logic delay. As you continue optimization, you can reach the point where the routing delay percentage of total delay will be roughly one-half.

3.2.1 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 an easy timing constraint for the tools to meet for most CPU designs and thus, the tools will only attempt to optimize routing until timing is met, and will then stop. The critical paths you see in the report may not be the 'actual' critical paths since the tools haven't been pushed to the limit.

We recommend that you begin optimization by increasing the clock frequency slowly and running `make` until the routing tool fails to meet timing. At this point, you know that the tools tried as hard as they could and just missed timing, so then the critical paths you see in the report are the 'actual' ones you need to work on.

As an aside, 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. Thus, you will get 'false' critical paths, that aren't necessarily where you should spend your time when optimizing.

3.3 Optimization Tips

As you work on achieving a higher clock speed, you will likely notice that the routing tool (PAR) is quite temperamental. You may find that your design might meet timing for a given clock speed, but after making a small, insignificant design change, the tool fails to meet timing. This is because PAR uses a random seed as a starting point in its algorithm. Sometimes it is a 'good' seed and yields an optimal result, but a small design change may cause the same seed to become 'bad' for that design and it yields a sub-optimal result.

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, add a branch predictor, 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.

You will be graded based on the best `mmult` performance you were able to achieve, as well as your documentation/reasoning for your architecture modifications in the process of optimization. You need to also take into consideration area usage when optimizing, so be sure to keep records as you optimize.

4 Final Checkpoint: Optimizations, Extra Credit, and Grading

All groups must complete the final checkoff by Friday, December 8. Use the week prior to your final checkoff for code cleanup, optimizations, late checkpoints, and optional extra credit projects.

4.1 Grading on Optimization

To receive full credit, you must demonstrate a working CPU at an optimized clock frequency (above 50Mhz) that has a working BIOS, can load and execute programs (both echo and mmult), can receive, process, and send to user I/O, and has a working AC97 controller. Additionally, you will be graded on total FPGA resource utilization, with the best designs using as few resources as possible. 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.

Credit for your area optimizations will be calculated using a cost function. At a high level, the cost function will look like:

$$\text{Cost} = \text{C}_{\text{LUT}} \times \#\text{ofLUTs} + \text{C}_{\text{RAMB}} \times \#\text{ofRAMBs} + \text{C}_{\text{REG}} \times \#\text{ofSliceRegisters}$$

where C_{LUT}, C_{RAMB}, and C_{REG} are constant value weights that will be decided upon based on how much each resource that you use should cost. As part of your final grade we will evaluate the cost of your design based on this metric. Keep in mind that cost is only one very small component of your project grade. Correct functionality is far more important.

4.2 Checkpoints

We have divided the project up into checkpoints so that you (and the staff) can pace your progress. The due dates are indicated at the end of each checkpoint section, as well as in the **Project Timeline** section at the end of this document. During the week each checkpoint is due, you will be required to get your implementation checked off by the GSI in the lab section you are enrolled in.

4.3 Style: Organization, Design

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

4.4 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 (not make excuses for why something didn't work). 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.

4.4.1 Report Details

You will turn in your project report on bCourses by the final checkoff date. The report should be around 8 pages total with around 5 pages of text and 3 pages of figures (± a few pages on each). 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 (≈ 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. For instance, describe the details of your AC97 controller, FIFOs, DVI controller, and any extra credit work. (≈ 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 number of LUTs and SLICE registers used by your design, which can be found by running `make report`. 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). (≈ 1-2 pages).
- **Conclusions**. What have you learned from this experience? How would you do it different next time? (≈ 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 bCourses. 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. (≈ 0.5 page).

When we grade your report, we will grade for clarity, organization, and grammar. Make sure to proofread and correct mistakes before turning it in. Submit your report to the bCourses assignment. Only one partner needs to submit the shared report, while each individual will need to submit the division of labor report to a separate bCourses assignment.

4.5 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.

- Branch Predictor: Implement a two bit (or more complicated) branch predictor with a branch history table (BHT) to replace the naive 'always taken' predictor used in the project
- 5-Stage Pipeline: Add more pipeline stages and push the clock frequency past 100MHz
- Audio Recording: Enable capturing mic input from the AC97 controller (for undergrads)
- RISC-V M Extension: Extend the processor with a hardware multiplier and divider
- 3 (or more) bit color: Increase the size of the framebuffer to have control of the RGB content of each pixel
- Dynamic Resolution: Allow the processor to control the output resolution of the DVI controller at runtime

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

4.6 Project Grading

80% Functionality at project due date. Your design will be subjected to a comprehensive test suite and your score will reflect how many of the tests your implementation passes.

5% Optimization at project due date. This grade is a function of the resources used by your implementation. 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. 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.

5 Project Timeline

Checkpoint	Deliverable	Due Date
1: RISCV151 Processor	Design Review In-Lab Checkoff	Thursday, October 26 @ 8:00 PM Thursday, November 9 @ 8:00 PM
2: IO, FIFOs, AC97 Controller	In-Lab Checkoff	Thursday, November 16 @ 8:00 PM
3: TBD	In-Lab Checkoff	Thursday, November 30 @ 8:00 PM
Final Checkoff, Extra Credit, and Optimizations	In-lab Checkoff Github code submission	Friday, December 8 @ TBD (by appointment)
Final Report	bCourses submission	Friday, December 8 @ 11:59pm

Table 4: EECS151 Fall 2017 Project Timeline