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

EECS151/251A - LB, Fall 2019

# Project Specification: RISCV151

Version 4.3

## Contents

**1 Introduction** **4**
1.1 Tentative Deadlines . . . 4
1.2 General Project Tips . . . 4

**2 Checkpoints 1 & 2 - 3-stage Pipelined RISC-V CPU** **5**
2.1 Setting up your Code Repository . . . 6
2.2 Integrate Designs from Labs . . . 6
2.3 Project Skeleton Overview . . . 6
2.4 RISC-V 151 ISA . . . 7
2.4.1 CSR Instructions . . . 8
2.5 Pipelining . . . 8
2.6 Hazards . . . 8
2.7 Register File . . . 10
2.8 RAMs . . . 10
2.8.1 Initialization . . . 10
2.8.2 Endianness + Addressing . . . 10
2.8.3 Reading from RAMs . . . 11
2.8.4 Writing to RAMs . . . 11
2.9 Memory Architecture . . . 12
2.9.1 Summary of Memory Access Patterns . . . 12
2.9.2 Unaligned Memory Accesses . . . 13
2.9.3 Address Space Partitioning . . . 13
2.9.4 Memory Mapped I/O . . . 14
2.10 Testing . . . 14
2.10.1 Integration Testing . . . 15
2.11 Software Toolchain - Writing RISC-V Programs . . . 15
2.12 Assembly Tests . . . 16
2.13 RISC-V ISA Tests . . . 16
2.14 BIOS and Programming your CPU . . . 17
2.15 Target Clock Frequency . . . 18
2.16 Matrix Multiply . . . 18
2.17 How to Survive This Checkpoint . . . 19
2.17.1 How To Get Started . . . 19

2.18 Checkoff . . . 21
2.18.1 Checkpoint 1: Block Diagram . . . 21
2.18.2 Non-Checkpoint Weeks . . . 21
2.18.3 Checkpoint 2: Base RISCV151 System . . . 21
2.18.4 Checkpoints 1 & 2 Deliverables Summary . . . 22

**3 Checkpoint 3 - I/O Integration, PWM Controller, Subtractive Synthesizer 23**
3.1 I/O Integration . . . 23
3.1.1 Hookup User I/O . . . 23
3.1.2 User I/O Test Program . . . 24
3.2 PWM DAC . . . 24
3.2.1 RISC-V Core Connection . . . 25
3.2.2 Implementation . . . 26
3.2.3 Piano Program . . . 27
3.3 Subtractive Synth . . . 27
3.3.1 Numerically Controlled Oscillator (NCO) - Required . . . 29
3.3.2 Sample Buffer and CDC - Required . . . 32
3.3.3 Synth Model . . . 33
3.3.4 FPGA Testing . . . 34
3.3.5 State Variable Filter (SVF) - Optional . . . 34
3.3.6 Amplitude Envelope (ADSR) - Optional . . . 35
3.3.7 Polyphonic Synthesis - 251A Only . . . 35
3.4 Checkpoint 3 Deliverables Summary . . . 36

**4 Final Checkpoint - Optimization 37**
4.1 Clock Generation Info + Changing Clock Frequency . . . 37
4.2 Critical Path Identification . . . 38
4.2.1 Finding Actual Critical Paths . . . 38
4.3 Optimization Tips . . . 38

**5 Optimizations, Extra Credit, and Grading 40**
5.1 Grading on Optimization . . . 40
5.2 Checkpoints . . . 40
5.3 Style: Organization, Design . . . 40
5.4 Final Project Report . . . 40
5.4.1 Report Details . . . 41
5.5 Extra Credit . . . 42
5.6 Project Grading . . . 42

**6 Project Timeline 43**

**A Local Development 44**
A.1 Linux . . . 44
A.2 OSX, Windows . . . 44

**B BIOS 44**

B.1 Background . . . . . 45
B.2 Loading the BIOS . . . . . 45
B.3 Loading Your Own Programs . . . . . 45
B.4 The BIOS Program . . . . . 46
B.5 The UART . . . . . 47
B.6 Command List . . . . . 48
B.7 Adding Your Own Features . . . . . 48

# 1 Introduction

The 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 3-stage pipelined RISC-V CPU with a UART for tethering. Afterwards, you will attach the IO circuits you built in the lab to the CPU and design a subtractive audio synthesizer. Then, you will implement a simple dynamic frequency scaling algorithm to bound the power consumption and temperature of your FPGA design while achieving a performance target. Finally, you will optimize your CPU for performance (maximizing the Iron Law) and cost (FPGA resource utilization).

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

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

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

## 1.1 Tentative Deadlines

The following is a brief description of each checkpoint and approximately how many weeks will be alloted to each one. This schedule may change as the semester progresses. The current schedule is summarised at the end of the document in Section 6.

- **November 1 - Checkpoint 1 (1 week)** - Draw a schematic of your processor's datapath and pipeline stages.
- **November 22 - Checkpoint 2 (3 weeks)** - Implement your RISC-V processor core in Verilog and write tests to verify your implementation.
- **December 9 - Checkpoint 3 (2 weeks)** - Attach I/O components from lab to your processor (FIFOs, buttons, switches), general PWM controller, basic subtractive synthesizer
- **December 9 (by appointment) - Final Checkoff** - Final processor optimization and checkoff
- **December 11 - Project Report** - Final report due

## 1.2 General Project Tips

Document your project as you go. You should comment your Verilog and keep your diagrams up to date. Aside from the final project report (you will need to turn in a report documenting your

project), you can use your design documents to help the debugging process.

Finish the required features first. Attempt extra features after everything works well. **If your submitted project does not work by the final deadline, you will not get any credit for any extra credit features you have implemented.**

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

## 2 Checkpoints 1 & 2 - 3-stage Pipelined RISC-V CPU

The first checkpoint in this project is designed to guide the development of a three-stage pipelined RISC-V CPU that will be used as a base system in subsequent checkpoints.

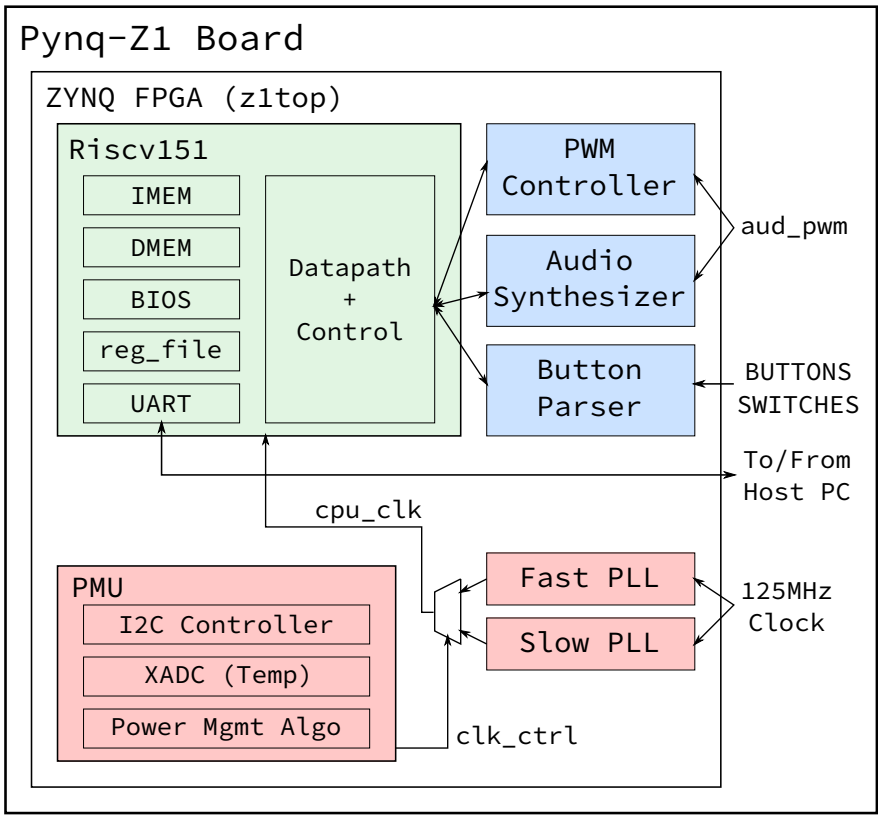



Figure 1: High-level overview of the full system

The green (RISC-V core) block on the diagram is the focus of the first and second checkpoints. The third checkpoint will add audio and IO components in blue. Finally, the fourth checkpoint will implement the power management unit in red.

## 2.1 Setting up your Code Repository

The project skeleton files are available on Github. The suggested way for initializing your repository with the skeleton files is as follows:

```
git clone git@github.com:EECS150/project_skeleton_fa19.git
cd project_skeleton_fa19
git remote add my-repo git@github.com:EECS150/fa19_teamXX.git
git push my-repo master
```

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

```
cd ..
rm -rf project_skeleton_fa19
git clone git@github.com:EECS150/fa19_teamXX.git
cd fa19_teamXX
git remote add staff git@github.com:EECS150/project_skeleton_fa19.git
```

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

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

You should check frequently for updates to the skeleton files. Update announcements will be posted to Piazza.

## 2.2 Integrate Designs from Labs

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

```
cd fa19_teamXX
cp fpga_labs_fa19/lab6/debouncer.v fa19_teamXX/hardware/src/io_circuits/.
```

**Copy these files from the labs:**

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

## 2.3 Project Skeleton Overview

- `hardware`
  - `src`
    - `z1top.v`: Top level module. The RISC-V CPU is instantiated here.

    * `PYNQ-Z1.xdc`: Constraints file. You can modify this to change pin assignments for peripherals when connecting I/O.
    * `riscv_core/Riscv151.v`: All of your CPU datapath and control should be contained in this file.
    * `riscv_core/reg_file.v`: Your register file implementation.
    * `memories/{imem, dmem, bios_mem}.v`: Synthesizable RAMs for the instruction, data, and BIOS memories.
    * `io_circuits/uart.v, uart_transmitter.v, uart_receiver.v`: Your working UART from Labs 5 and 6
  - `sim`
    * `assembly_testbench.v`: Starting point for testing your CPU. Works with the software in `assembly_tests`.
    * `echo_testbench.v`: Runs the software in `echo` on your CPU. The software implements the echo FSM from lab 5, and the testbench controls an off-chip UART to test it.
- `software`
  - `bios151v3`: The BIOS program, which allows us to interact with our CPU via the UART. You need to compile it before creating a bitstream or running a simulation.
  - `echo`: The echo program, which emulates the FSM from Lab 5 in software.
  - `assembly_tests`: Use this as a template to write assembly tests for your processor designed to run in simulation.
  - `c_example`: Use this as an example to write C programs.
  - `mmult`: This is a program to be run on the FPGA for Checkpoint 2. It generates 2 matrices and multiplies them. Then it returns a checksum to verify the correct result.

To compile `software` go into a program directory and run `make`. To build a bitstream run `make impl` in `hardware`.

### 2.4 RISC-V 151 ISA

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

### 2.4.1 CSR Instructions

You will have to implement 2 CSR instructions to support running the standard RISC-V ISA test suite. A CSR (or control status register) is some state that is stored independent of the register file and the memory. While there are $2^{12}$ possible CSR addresses, you will only use one of them (`tohost = 0x51E`). The `tohost` register is monitored by the RISC-V ISA testbench (`isa_testbench.v`), and simulation ends when a non-zero value is written to this register. A CSR value of 1 indicates success, and a value greater than 1 indicates which test failed.

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

1. `csrw tohost,x2` (short for `csrrw x0,csr,rs1` where `csr = 0x51E`)
2. `csrwi tohost,1` (short for `csrrwi x0,csr,uimm` where `csr = 0x51E`)

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

## 2.5 Pipelining

Your CPU must implement this instruction set using a 3-stage pipeline. The division of the datapath into three stages is left unspecified as it is an important design decision with significant performance implications. We recommend that you begin the design process by considering which elements of the datapath are synchronous and in what order they need to be placed. After determining the design blocks that require a clock edge, consider where to place asynchronous blocks to minimise the critical path. The RAMs we are using for the data, instruction, and BIOS 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. You must resolve data hazards by implementing forwarding whenever possible. This means that you must forward data from your data memory instead of stalling your pipeline or injecting NOPs. All data hazards can be resolved by forwarding in a three-stage pipeline.

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

1. **Read-after-write data hazards** Consider carefully how to handle instructions that depend on a preceding load instruction, as well as those that depend on a previous arithmetic instruction.
2. **Control hazards** What do you do when you encounter a branch instruction, a jal (jump and link), or jalr (jump from register and link)? You will have to choose whether to predict branches as taken or not taken by default and kill instructions that weren't supposed to execute if needed. You can begin by resolving branches by stalling the pipeline, and when your processor is functional, move to naive branch prediction.

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 |

## 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 write a testbench to verify the following:

- Register 0 is not writable, i.e. reading from register 0 always returns 0
- Registers are updated on the same cycle that a write occurs (i.e. the value read on the cycle following the rising edge of the write should be the value written).
- 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)

## 2.8 RAMs

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

### 2.8.1 Initialization

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

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

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

### 2.8.2 Endianness + Addressing

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

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

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

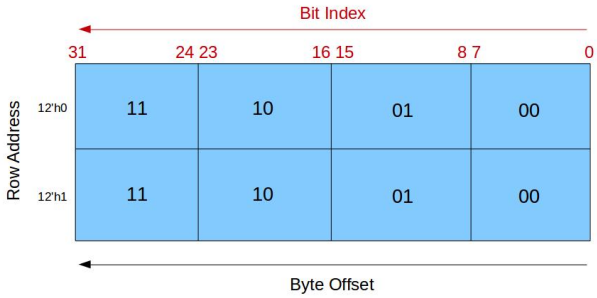



Figure 2: Block RAM organization. The labels for row address **should read 14'h0 and 14'h1.**

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

### 2.8.3 Reading from RAMs

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

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

### 2.8.4 Writing to RAMs

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

Here's an example of storing a single byte:

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

## 2.9 Memory Architecture

The standard RISC pipeline is usually depicted with separate instruction and data memories. Although this is an intuitive representation, it does not let us modify the instruction memory to run new programs. Your CPU, by the end of this checkpoint, will be able to receive compiled RISC-V binaries though the UART, store them into instruction memory, then jump to the downloaded program. To facilitate this, we will adopt a modified memory architecture shown in Figure 3.

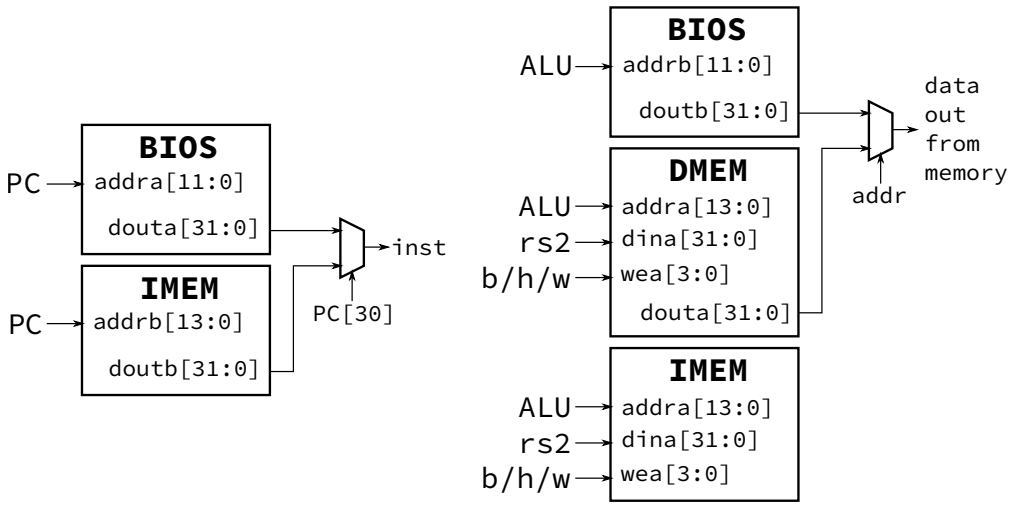



Figure 3: The Riscv151 memory architecture. There is only 1 IMEM and DMEM instance in Riscv151 but their ports are shown separately in this figure for clarity. The left half of the figure shows the instruction fetch logic and the right half shows the memory load/store logic.

### 2.9.1 Summary of Memory Access Patterns

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

The processor will begin execution from the BIOS memory, which will be initialized with the BIOS program (in `software/bios151v3`). The BIOS program should be able to read from the BIOS memory (to fetch static data and instructions), and read and write the instruction and data memories. This allows the BIOS program to receive user programs over the UART from the host PC and load them into instruction memory.

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

### 2.9.2 Unaligned Memory Accesses

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

### 2.9.3 Address Space Partitioning

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

Table 2: Memory Address Partitions

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

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

For example, a store to an address beginning with `0x3` will write to both the instruction memory and data memory, while storing to addresses beginning with `0x2` or `0x1` will write to only the instruction or data memory, respectively. For details about the BIOS and how to run programs on your CPU, see Section 2.14.

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

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

### 2.9.4 Memory Mapped I/O

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

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

Table 3 shows the memory map for this stage of the project.

Table 3: I/O Memory Map

| **Address** | **Function** | **Access** | **Data Encoding** |
|---|---|---|---|
| `32'h80000000` | UART control | Read | `{30'b0, data_out_valid, data_in_ready}` |
| `32'h80000004` | UART receiver data | Read | `{24'b0, data_out}` |
| `32'h80000008` | UART transmitter data | Write | `{24'b0, data_in}` |
| `32'h80000010` | Cycle counter | Read | Clock cycles elapsed |
| `32'h80000014` | Instruction counter | Read | Number of instructions executed |
| `32'h80000018` | Reset counters to 0 | Write | N/A |

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

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

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

## 2.10 Testing

The design specified for this project is a complex system and debugging can be very difficult without tests that increase visibility of certain areas of the design. In assigning partial credit at the end

for incomplete projects, we will look at testing as an indicator of progress. A reasonable order in which to complete your testing is as follows:

1. Test that your modules work in isolation via Verilog testbenches
2. Test the entire CPU one instruction at a time with hand-written assembly — see `assembly_testbench.v`
3. Test the CPU's memory mapped I/O — see `echo_testbench.v`
4. Run the `riscv-tests` ISA test suite

### 2.10.1 Integration Testing

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

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

## 2.11 Software Toolchain - Writing RISC-V Programs

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

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

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

There are several files:

- `start.s`: This is an assembly file that contains the start of the program. It initialises the stack pointer then jumps to the `main` label. Edit this file to move the top of the stack. Typically your stack pointer is set to the top of the data memory address space, so that the stack has enough room to grow downwards.
- `c_example.ld`: This linker script sets the base address of the program. For Checkpoint 2, this address should be in the format `0x1000xxxx` The .text segment offset is typically set to the base of the instruction memory address space.
- `c_example.elf`: Binary produced after running `make`.
  Use `riscv64-unknown-elf-objdump -Mnumeric -D c_example.elf` to view the assembly code.

- `c_example.dump`: Assembly dump of the binary.

### 2.12 Assembly Tests

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

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

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

```
_start:

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

Done: j Done
```

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

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

### 2.13 RISC-V ISA Tests

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

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

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

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

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

```
cd hardware
make isa-tests
```

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

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

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

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

## 2.14 BIOS and Programming your CPU

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

To run the BIOS:

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

   ```
   screen $SERIALTTY 115200
   ```

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

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

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

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

As an example, running `sw cafef00d 10000000` should write to the data memory and running `lw 10000000` should print the output `10000000: cafef00d`. Please also pay attention that writes

to the instruction memory (`sw ffffffff 20000000`) do not write to the data memory, i.e. `lw 10000000` still should yield `cafef00d`.

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

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

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

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

## 2.15 Target Clock Frequency

By default, the CPU clock frequency is set at 50MHz. It should be easy to meet timing at 50 MHz. Look at the reports in `hardware/build/synth/post_synth_timing_summary.rpt` and `impl/post_route_timing_summary.rpt` to see if timing is met. If you failed, the timing reports specify the critical path you should optimize.

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

## 2.16 Matrix Multiply

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

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

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

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

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

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

## 2.17 How to Survive This Checkpoint

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

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

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

### 2.17.1 How To Get Started

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

1. *Design.* You should start with a comprehensive and detailed design/schematic. Enumerate all the control signals that you will need. Be careful when designing the memory fetch stage since all the memories we use (BIOS, instruction, data, IO) are synchronous.
2. *First steps.* Implementing some modules that are easy to write and test. Write the `reg_file.v` module. Create a Verilog testbench and test the cases in Section 2.7.
3. *Control Unit + other small modules.* Implement the control unit, ALU, and any other small independent modules. Unit test them.
4. *Memory.* In the beginning, only use the BIOS memory in the instruction fetch stage and only use the data memory in the memory stage. This is enough to run assembly tests.

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

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

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

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

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

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

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

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

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

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

## 2.18 Checkoff

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

### 2.18.1 Checkpoint 1: Block Diagram

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

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

You should be able to describe in detail any smaller sub-blocks in your diagram. Though the textbook diagrams are a decent starting place, remember that they often use asynchronous-read RAMs for the instruction and data memories, and we will be using synchronous-read block RAMs. Additionally, at this point we recommend that you have completely functional UART, ALU, instruction decoder, and register file modules (see Section 2.7).

**Checkpoint 1 is due in lab no later than November 1**. 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.

### 2.18.2 Non-Checkpoint Weeks

GSIs will be in lab during the regular times to help you. Come to lab every week even if there is no checkoff deadline.

### 2.18.3 Checkpoint 2: Base RISCV151 System

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

**Checkpoint 2 materials should be committed to your project repository by November 22.**

### 2.18.4 Checkpoints 1 & 2 Deliverables Summary

| Deliverable | Due Date | Description |
|---|---|---|
| Block Diagram | November 1 | Sit down with a GSI and go over your design in detail |
| RISC-V CPU | November 22 Check in code to Github | Demonstrate that the BIOS works, you can use `hex_to_serial` to load the `echo` program, `jal` to it from the BIOS, and have that program successfully execute. Load the mmult program with `hex_to_serial`, `jal` to it, and have it execute successfully and return the benchmarking results and correct checksum. Your CPI should be under 1.2 |

# 3 Checkpoint 3 - I/O Integration, PWM Controller, Subtractive Synthesizer

In checkpoint 3 of this project you will implement a memory mapped I/O interface to user inputs and outputs (buttons, LEDs, and switches). To buffer user inputs to your processor you will integrate the FIFO you built in lab.

You will design a generic PWM controller that will function as a digital-to-analog converter (DAC) that drives the audio output jack. Finally, you will implement a simple polyphonic subtractive synthesizer with a numerically controlled oscillator, a digital filter, and an amplitude envelope.

## 3.1 I/O Integration

In lab, you built a synchronizer, debouncer and an edge detector that were used to take in various user inputs. Now, we want our processor to have access to these inputs (and the switches) and also to be able to drive outputs such as the LEDs. We will extend our memory map to give programs access to these I/Os.

When a user pushes a button on the Pynq-Z1 board, the button's signal travels through the synchroniser → debouncer → edge detector chain. The result is a single clock cycle wide pulse coming out of the edge detector that represents a single button press. If we just extended our memory map to directly include the outputs from the edge detector, the processor would have to read from those locations on every clock cycle to be sure it didn't miss any user inputs.

To fix this, we will buffer user inputs with a FIFO and let the processor consume them when it has time to do so.

### 3.1.1 Hookup User I/O

We want to give the processor access to these I/Os:

- Switches
- GPIO LEDs (the ones on the Pynq-Z1 board)
- Push-buttons

The I/O extension to the memory map is in Table 5.

On any given clock cycle, when any of the button signals pulse high, the FIFO should be written to with the status of all the button signals. The CPU should be able to read the empty signal of the FIFO, and it should be able to read out data from the FIFO with the FIFO's `rd_en` signal controlled by your memory logic.

Modify `z1top.v` and `Riscv151.v` by instantiating your FIFO, hooking up its ports to the user I/O signals, and connecting your FIFO's read interface to the RISC-V core.

Table 4: Memory Map for I/O Integration

| Address | Function | Access | Data Encoding |
|---|---|---|---|
| 32'h80000020 | GPIO FIFO Empty | Read | {31'd0, empty} |
| 32'h80000024 | GPIO FIFO Read Data | Read | {29'd0, buttons[2:0]} |
| 32'h80000028 | Switches | Read | {30'd0, SWITCHES[1:0]} |
| 32'h80000030 | GPIO LEDs | Write | {26'd0, LEDS[5:0]} |

### 3.1.2 User I/O Test Program

The `software/user_io_test` tests the FIFO and user I/O integration. After programming the FPGA, run `make` in the `user_io_test` folder, and run `hex_to_serial user_io_test.hex 30000000`. Then `screen` and `jal 10000000` from the BIOS to jump into the user I/O test program.

This program has several commands to help you debug and verify functionality:

- `read_buttons` - CPU reads from the GPIO FIFO until it is empty, decodes the button press data, and prints it out.
- `read_switches` - CPU reads the slide switches' address and prints out the state of the switches.
- `led <data>` - Writes the `<data>` (32-bits in hex) that you specify to the GPIO LEDs address. We only have 6 LEDs on the board so you can write values up to 0x3F.
- `exit` - Jump back into BIOS.

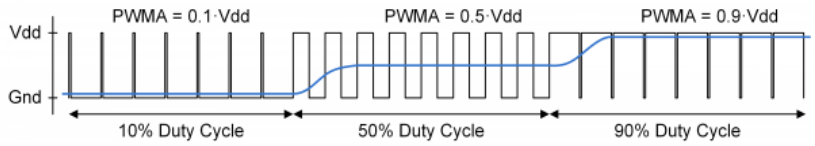
## 3.2 PWM DAC

In lab we built a `tone_generator` that took in a frequency and produced a square wave that drove the audio jack. Now, we want to produce arbitrary waveforms so we can produce different types of sound.

To do this, we will design a PWM controller to function like a Digital-Analog-Converter (DAC). Rather than feeding the `tone_generator` a "tone switch" period that describes a specific audio frequency, we will feed in a 12 bit number to a PWM controller that determines the duty cycle of the PWM output signal. The 12 bit input to the PWM controller will determine the duty cycle according to the following equation:

$$\text{Duty Cycle} = \frac{\text{Input}[11:0]}{2^{12} - 1}$$

The Pynq board has a low pass filter attached to the output of `aud_pwm` that will smooth the PWM waveform to an analog voltage:



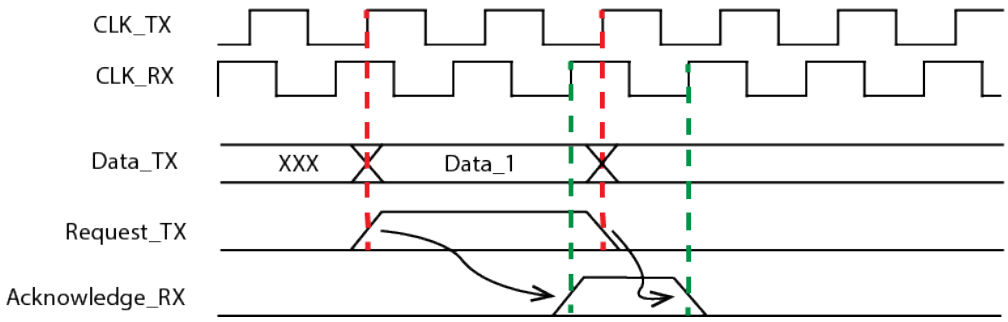

Note that there's no more notion of volume in this block (that will be handled in software). We just want this block to act as a DAC.

### 3.2.1 RISC-V Core Connection

You'll have to memory map the input of this PWM block, but there's a catch. In order to actually use this block as a proper DAC (especially for the next part), it will need to run very fast (much faster than your processor). This implies that your PWM block will be running inside a different clock domain.

We dealt with clock domain crossings before (our synchronizer took an asynchronous signal and brought it into a synchronous domain). However the single-bit synchronizer only works with 1-bit signals and not a 12-bit bus like the duty cycle input to the PWM controller. We will implement a 4 phase handshake to make sure we reliably pass data from the RISC-V core's clock domain (TX/Transmit domain) to the PWM block's clock domain (RX/Receive domain).

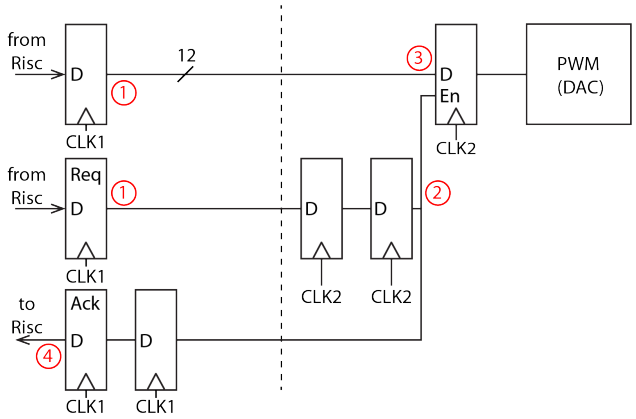
A timing diagram is shown below:




The four phase handshake uses four signals (`req_tx`, `req_rx`, `ack_rx`, and `ack_tx`). `req_rx` is the `req_tx` signal synchronized into the RX domain, and `ack_tx` is the `ack_rx` signal synchronized into the TX domain. requires the generation of two new synchronous signals, request and acknowledge, and the comparison of 4 edges.

1. Request is asserted at the same time that the data is registered in the TX domain
2. Acknowledge is asserted from the RX domain after the data has been captured in the RX domain (this could require holding the data for two clock cycles to ensure data coherency)
3. Request is de-asserted in the TX domain
4. Acknowledge is de-asserted in the RX domain and is then de-asserted in the TX domain

Implement this with 2-flop synchronizers that synchronize the request and acknowledge signals between the clock domains. A block diagram is shown below:




The numbers highlight the flow of data through the synchronizer circuit. Here is a detailed sequence of steps to send data from the TX domain to the RX domain:

1. First, the RISC-V core stores the duty cycle to the TX data register and then sets the TX request register high
2. The TX request bit is synchronized into the RX domain
3. Once the RX request bit is high, the RX data register takes in the value from the TX data register (knowing that the TX data register has been stable for 2 cycles)
4. The RX request bit is synchronized back to the TX domain as an acknowledge bit
5. This concludes a data transfer. To prepare for the next transfer, the TX request bit is de-asserted
6. This will cause the TX acknowledge bit to be de-asserted
7. The transmitting side is ready for a new data transfer

While this is slower than simply passing data through clock domains, it transmits data reliably. For this first part, your RISC-V core will handle all of the requests and acknowledges in software through a memory mapped interface detailed below.

Table 5: Memory Map for PWM Integration

| **Address** | **Function** | **Access** | **Data Encoding** |
|---|---|---|---|
| `32'h80000034` | Duty Cycle | Write | `{20'b0, duty_cycle[11:0]}` |
| `32'h80000038` | TX Request | Write | `{31'b0, tx_req}` |
| `32'h80000040` | TX Acknowledge | Read | `{31'b0, tx_ack}` |

### 3.2.2 Implementation

You can create the PWM controller in its own file. You can instantiate the PWM controller where you want.

Some code has been provided at the bottom `z1top.v` that declares a wire `pwm_out` which should connect to the output of your PWM controller. A PLL has been instantiated in `z1top.v` which produces a 150 Mhz clock (`pwm_clk_g`) used for the PWM controller. There's also a `pwm_rst` signal that can be used to reset the PWM controller.

You should first test the PWM controller using a block-level testbench. Next try creating a testbench similar to the `echo_integration_testbench` that uses the piano software from the next section. You will have to load the program into the IMEM and DMEM in simulation and set the `RESET_PC` accordingly (similar to `isa_testbench`).

### 3.2.3 Piano Program

Load the program in `software/square_piano`. After running `jal 10000000`, you can use the keyboard rows from 'c' to ',' and from 'q' to 'i' to play piano notes. You can hold down shift to generate upper and lower octaves just like in Lab 6.

This program plays square waves for a fixed note length (1/3rd of a second). The note length can be adjusted using the buttons:

- `buttons[1]` = double note length
- `buttons[2]` = halve note length
- `buttons[3]` = reset note length to 1/3rd of a second

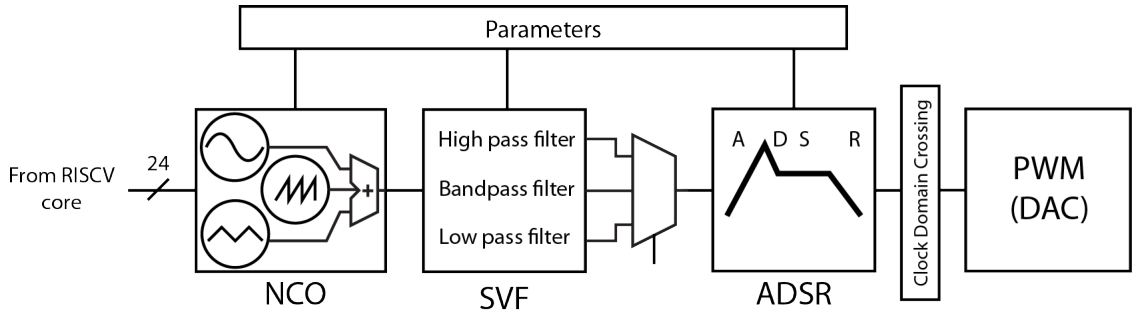
If your processor is running at a frequency other than 50 MHz, regenerate the `scale.h` file. Example below is for 100 MHz.

```
cd software/square_piano
./piano_scale_generator scale.h 100e6
```

## 3.3 Subtractive Synth

While the piano program works, it can only play a single note at a time (monophonic synthesis). More importantly, it's limited to the CPI of the processor and this in turn limits the waveforms we can send to our DAC.

How could we improve our synthesizer's throughput and implement a polyphonic synthesizer? Hardware acceleration! We can implement most of the subtractive synthesis blocks in hardware and lay the groundwork for building a rich sounding synthesizer. The main datapath that we'll be implementing is shown below:



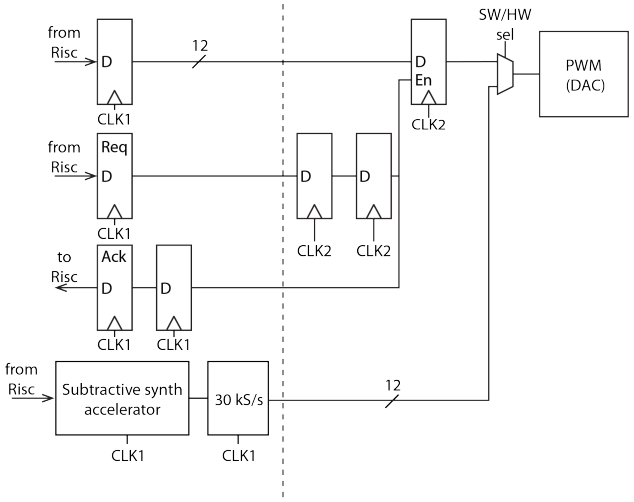

The synthesizer is composed of three blocks:

1. Numerically controlled oscillator (NCO). Produces sampled sine, square, triangle, and sawtooth waveforms of programmable frequency. - **Required**
2. State variable filter (SVF). A 2nd order digital IIR filter. - **Optional, Extra Credit**
3. Amplitude envelope (ADSR - attack, decay, sustain, release) - **Optional, Extra Credit**

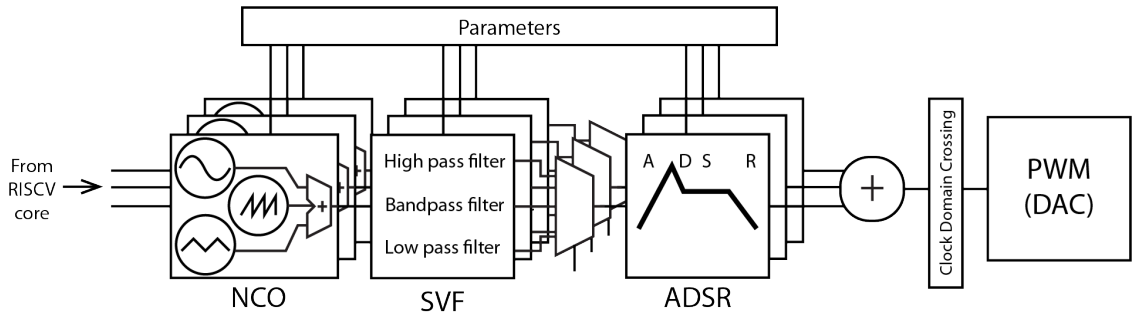
The RISC-V core controls the parameters for each of the blocks using registers that are memory mapped. A collection of parameter values designed to mimic a particular instrument is called a patch.

The RISC-V core also provides the frequency control word for the NCO, which represents the pitch of a note to play. Once a note to play is sent to the synthesizer, it will select/sum the appropriate waveforms (from the NCO), filter the samples through the SVF, and apply an ADSR envelope. The synthesizer should feed samples at the sample rate ($f_{samp}$) to the PWM DAC until the note is released.

This synthesizer is connected to the RISC-V core as shown below.




This implements a monophonic synth which can play one note at a time, but we would like to play multiple notes at a time. To accomplish this, we can replicate the monophonic synth and sum the output of the amplitude envelope blocks as shown in the figure below to produce a polyphonic synth.




**A polyphonic synth is required for 251A only**.

### 3.3.1 Numerically Controlled Oscillator (NCO) - Required

We want to generate four waveform types: sine, square, triangle, and sawtooth. We'll discuss the sine wave generation and you can generalize it to the other waveform types.

**NCO Overview** A continuous time sine wave, with a frequency $f_{sig}$, can be written as:

$$f(t) = \sin\left(2\pi f_{sig} t\right)$$

If this sine wave is sampled with sampling frequency $f_{samp}$, the resulting stream of discrete time samples is:

$$f[n] = \sin\left(2\pi f_{sig} \frac{n}{f_{samp}}\right)$$

We want to generate this stream of samples in hardware. One way to do this is to use a lookup table (LUT) and a phase accumulator (just a register and an adder). Say we have a LUT that contains sampled points for one period of a sine wave with $2^N$ entries. The entries $i, 0 \leq i < 2^N$ of this LUT are:

$$LUT[i] = \sin\left(i \frac{2\pi}{2^N}\right)$$

To find the mapping of the sample $n$ to the LUT entry $i$, we can equate the expressions inside sin():

$$\begin{aligned} i \frac{2\pi}{2^N} &= 2\pi f_{sig} \frac{n}{f_{samp}} \\ i &= \underbrace{\left(\frac{f_{sig}}{f_{samp}} 2^N\right)}_{\text{phase increment}} n \end{aligned}$$

This means that to compute sample $n+1$ we should take the LUT index used for sample $n$ and increment the index by the 'phase increment'. The phase increment (also called the 'frequency control word') is written by the RISC-V core into a MMIO register (24-bits wide).

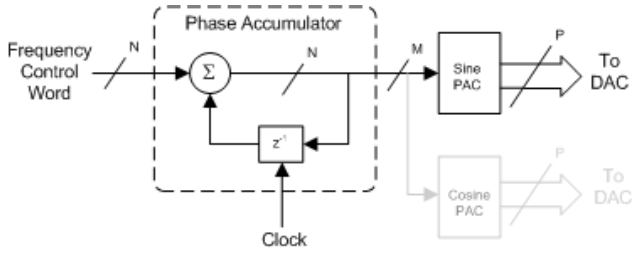



Figure 4: NCO architecture from Wikipedia

To find the frequency resolution (the minimum frequency step) we can look at what change in $f_{sig}$ causes the phase increment to increase by 1:

$$\frac{f_{sig} + \Delta_{f,min}}{f_{samp}} 2^N = \frac{f_{sig}}{f_{samp}} 2^N + 1$$

$$\Delta_{f,min} = \frac{f_{samp}}{2^N}$$

This means for a example sample frequency $f_{samp}$ of 30 kHz and $N = 24$, the frequency resolution is 0.001 Hz. We can have very precise frequency control using an NCO.

However, a $2^{24}$ entry LUT is huge and wouldn't fit on the FPGA. So, we will keep the phase accumulator $N$ (24-bits) wide, and only use the MSB $M$ bits to index the sine wave LUT. This means the LUT only contains $2^M$ entries, where $M$ is chosen based on the required phase error. We will use $M = 8$.

**Fixed Point Representation of LUT Values** The values in the sine LUT will be signed numbers representing one period of a sampled sine wave. They will be stored in a fixed point representation (16 fractional bits, 4 integer bits inclusive of 1 sign bit). This number representation will be used through the entire synthesizer.

Review this 61C handout for a quick overview of fixed point numbers.

**NCO Non-Idealities** There are three sources of inaccuracy when using an NCO:

1. Limited frequency resolution when converting a continuous frequency $f_{sig}$ to the integer frequency control word
2. Using a smaller LUT ($2^M$ entries) than the phase accumulator width ($2^N$ possible values) can accomodate
3. Fixed point quantization vs. arbitrary precision representation of the LUT values

The first source of inaccuracy can be handled by choosing a large phase accumulator bit width. The second source can be alleviated using *interpolation*. The last source can be handled by using more bits per LUT element at the cost of area.

**Interpolation** You can linearly interpolate between the $i$ and $i+1$ LUT index using the residual bits in the phase accumulator. While the MSB $M$ bits are used to index into the LUT, the LSB $N-M$ bits can be used for interpolation as a residual error. You should get linear interpolation working after building the basic NCO.

**NCO Model** We've provided a model of the NCO in `scripts/synth/NCO.py`. You can run these commands. You can replace `sine` with `square`, `triangle`, or `sawtooth`.

1. `./NCO.py --analysis` - Plot NCO samples vs ideal samples from a sine function (to play with how to choose $N$ and $M$ and the effect of interpolation)
2. `./NCO.py --sine-lut` - Dump the binary representation of the 4 types of LUTs (so they can be used by `$readmemb` in Verilog)
3. `./NCO.py --sine-plot` - Plot the sine LUT values
4. `./NCO.py --golden` - Dump the golden sample values for $f_{samp} = 30$kHz and $f_{sig} = 880$Hz

You can dump a LUT to a file like: `./NCO.py --triangle-lut > triangle.bin`.

**Waveform Summation** The NCO produces all 4 waveform types per sample. Another module (the Summer), following the NCO, sums them together after scaling each waveform's sample. The scaling is done with an arithmetic right shift amount assigned to each waveform type.

**Implementation** The NCO implementation details are up to you. Note that it is OK to use several clock cycles to generate one sample value, so pipelining is useful for making sure your critical paths are short. **We recommend** that the NCO uses a ready/valid interface to produce samples (that are later consumed by the sample buffer).

You can read in a LUT binary using `$readmemb()`:

```
reg [x:0] sine_lut [0:y];
`define STRINGIFY_SINELUT(x) `"x/src/audio/sine.bin`"
initial begin
  \$readmemh(`STRINGIFY_SINELUT(`ABS_TOP), sine_lut);
end
```

The NCO should be hooked up to the RISC-V core via MMIO (Table 6).

**Testing** You should generate golden samples from the `NCO.py` script for various signal frequencies and use a unit-level testbench to verify the NCO produces identical samples.

Table 6: Memory Map for NCO and Summer

| Address | Function | Access | Data Encoding |
|---|---|---|---|
| `32'h80000200` | Sine NCO Scale | Write | `{27'd0, sine_shift[4:0]}` |
| `32'h80000204` | Square NCO Scale | Write | `{27'd0, square_shift[4:0]}` |
| `32'h80000208` | Triangle NCO Scale | Write | `{27'd0, triangle_shift[4:0]}` |
| `32'h8000020c` | Sawtooth NCO Scale | Write | `{27'd0, sawtooth_shift[4:0]}` |
| `32'h80001000` | Frequency Control Word (Voice 1) | Write | `{8'd0, fcw[23:0]}` |

### 3.3.2 Sample Buffer and CDC - Required

The signal chain we want to implement is below (with the data format of their outputs in parentheses, FixP = signed fixed point).

NCO (4x 16/4 FixP) → Summer (16/4 FixP) → Global Gain (16/4 FixP)
→ Truncator (12-bit unsigned int) → Buffer + CDC → PWM DAC

You implemented the NCO and summer in the previous section. Now we will implement 4 other blocks:

1. **Global gain**: this is just another arithmetic right shift of the summer output
2. **Truncator**: converts a high precision signed fixed-point sample to a 12-bit unsigned integer for the PWM DAC
3. **Buffer**: buffers a continuous stream of samples and emits them at a fixed rate of $f_{samp}$
4. **CDC**: sends samples from the buffer to the PWM clock domain using a 4-phase handshake

**Global Gain and Truncator** The global gain is an arithmetic right shifter on the output of the Summer. It is used to limit the volume of the synth.

The global gain block outputs fixed point samples (with 16 fractional bits and 4 integer bits). But, the PWM DAC takes 12-bit unsigned duty cycle values. The truncator should take the 12 MSB bits from the global gain block and treat them as a signed number. Then that number should be converted into the PWM duty cycle value:

- $-2^{11}$: duty cycle[11:0] = 0
- 0: duty cycle[11:0] = 2048
- $2^{11} - 1$: duty cycle[11:0] = 4095

The global gain comes from MMIO (Table 7).

Table 7: Memory Map for Gain/Truncator

| Address | Function | Access | Data Encoding |
|---|---|---|---|
| `32'h80000104` | Global Gain Shift | Write | `{27'd0, global_gain[4:0]}` |

**Sample Buffer + CDC** The NCO continuously produces samples, but now we want to send those samples to the PWM DAC at a fixed rate $f_{samp}$. Implement a block that rate limits samples from the NCO and sends them to the PWM DAC using the 4-phase CDC handshake.

We've added a MMIO register (PWM DAC Source) (Table 8) that controls which duty cycle register feeds the PWM block. This 1-bit register should be synchronized to the PWM clock domain and controls which RX duty-cycle register is fed into the PWM block. (0 = CPU, 1 = synth)

Table 8: Memory Map for Synth CDC

| Address | Function | Access | Data Encoding |
|---|---|---|---|
| `32'h80000044` | PWM DAC Source | Write | `{31'd0, source}` |

**Synth Integration** The synth top-level should stitch together the blocks above into a signal chain. The top-level also contains control signals that the CPU can use to reset the synth (Table 9).

The CPU will play notes by performing the following sequence:

1. Configure the static parameters of the synth (sine/square/triangle/sawtooth NCO gain, global gain, PWM DAC source)
2. Write to the global synth reset
3. To play a particular note:
   (a) Write the note's FCW to `0x8000_1000`
   (b) Write to `0x8000_1004` which should begin sending samples to the PWM DAC
   (c) Once the note is released, the CPU writes to `0x8000_1008`, and you should stop sending samples to the PWM DAC
   (d) The CPU then polls for `0x8000_100c` to go high, indicating the synth has flushed out all the samples to send
   (e) The CPU writes to `0x8000_1010` to clear `note_finished` and any other synth state

You should simulate this sequence in simulation.

### 3.3.3 Synth Model

We've provided a model of all the blocks above in several Python classes:

Table 9: Memory Map for Synth Management

| Address | Function | Access | Data Encoding |
|---|---|---|---|
| `32'h80000100` | Global Synth Reset | Write | `NA` |
| `32'h80001004` | Note Start (Voice 1) | Write | `NA` |
| `32'h80001008` | Note Release (Voice 1) | Write | `NA` |
| `32'h8000100c` | Note Finished (Voice 1) | Read | `{31'd0, note_finished}` |
| `32'h80001010` | Reset (Voice 1) | Write | `NA` |

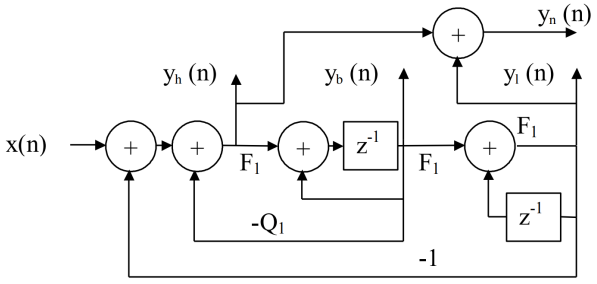
- `scripts/synth/NCO.py` - the NCO model, outputs all 4 waveform types
- `scripts/synth/Blocks.py` - models of the summer and truncator (with global gain)
- `scripts/synth/Synth.py` - model of the full mono and poly synths

You should use `Synth.py` to dump golden DAC samples for testing in simulation.

### 3.3.4 FPGA Testing

### 3.3.5 State Variable Filter (SVF) - Optional

The SVF is an infinite impulse response (IIR) filter with multiple outputs (as shown below). You will need to define this block diagram in verilog and then put the outputs into a mux that the parameters block will control. It's important to note that $Y_h$ is a high pass filtered output, $Y_b$ is the bandpass filtered output, $Y_n$ is the notch filtered output, and lastly $Y_l$ is the low pass filtered output. For further information (such as how to pick the different values for different corner frequencies) you can refer to this reference: SVF reference




In scripts/synthesizer.py, a software implementation of the synthesizer has been provided for you. It is important to keep in mind that digital implementation of IIR filters is non-trivial. If not done carefully, you may exceed the available number of bits and encounter overflow problems. In addition, multiplications must be performed by values less than one to make sure the system is bounded. Therefore, in the model provided for you, we implement "fixed point operation", where fractions are represented by binary numbers. For instance, 0.5 is "1", and 0.625 is "101" $(0.5 \times 1 + 0.25 \times 0 + 0.125 \times 1)$. Because we do not have values larger than 1, we do not need to separate integer from fraction (unless you decide to set Q to be larger than 1,in which case you

will have to handle this). Performing multiplication of fixed point numbers should be the same as performing regular multiplication. However, the sign of the number (MSB) should be taken out and computed separately. In total, you have 24 bits to spare in the SVF registers, Therefore, budget your resolution wisely to not encounter overflow problems. You may change the values of F to change the corner frequency of your filter. The filter should be able to adapt to different corner frequencies at run time.

### 3.3.6 Amplitude Envelope (ADSR) - Optional

The ADSR block can be simplified to an "ASR" block. Meaning, you just have an "attack, sustain, and release" process. This block will take the output of the SVF and modulate the amplitude such that the resulting sound will get louder, hold, and get quiet after every key press. This is done by multiplying the waveform out of the SVF filter with scalars. The ADSR should receive a "press" and a "release" signal from the CPU. You can choose the time it takes for attack to go from silent to the largest amplitude. Then the output would sustain its largest level until a "release" signal is received, and after that the sound start decaying to silence again. The python model provided for you generates a reference behavior of the ADSR envelop. You may choose the envelop to mimic any instrument you would like to.

### 3.3.7 Polyphonic Synthesis - 251A Only

To enable polyphony, instantiate multiple synths. We'll use 4 voice polyphony. Parameters that have MMIO addresses like `0x8000_01xx` or `0x8000_02xx` are shared among all synths.

Only the note-specific MMIO registers are distinct for each synth (Table 10). The 3rd nibble of the address distinguishes each synth's note registers.

A polyphonic synth model is in `scripts/synth/Synth.py`.

Table 10: Memory Map for Polyphony

| Address | Function | Access | Data Encoding |
|---|---|---|---|
| `32'h80002000` | Frequency Control Word (Voice 2) | Write | `{8'd0, fcw[23:0]}` |
| `32'h80002004` | Note Start (Voice 2) | Write | `NA` |
| `32'h80002008` | Note Release (Voice 2) | Write | `NA` |
| `32'h8000200c` | Note Finished (Voice 2) | Read | `{31'd0, note_finished}` |
| `32'h80002010` | Reset (Voice 2) | Write | `NA` |

*Coming:* software

### 3.4 Checkpoint 3 Deliverables Summary

| Deliverable | Due Date | Description |
|---|---|---|
| User I/Os + PWM Piano + Synth | December 9 | Demonstrate the working user IO test, and PWM piano programs. Describe the design of your synth and produce various sounds. |

# 4 Final Checkpoint - Optimization

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

The optimization goal for this project is to 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.

## 4.1 Clock Generation Info + Changing Clock Frequency

Open up `z1top.v`. You will notice a top level input called `CLK_125MHZ_FPGA`. It is a 125 MHz clock signal, which we will use to derive our CPU clock.

Scrolling down a little further, you will see an instantiation of `PLLE2_ADV`, which is a PLL (phase locked loop) primitive on the FPGA. This is a circuit that lets us create a new clock from an existing clock with a user-specified multiply-divide ratio.

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

$$\texttt{CLKOUT0}_f = \texttt{CLKIN1}_f \cdot \frac{\texttt{CLKFBOUT\_MULT}}{\texttt{DIVCLK\_DIVIDE} \cdot \texttt{CLKOUT0\_DIVIDE}}$$

In our case we get:

$$\texttt{CLKOUT0}_f = 125 \text{ MHz} \cdot \frac{34}{5 \cdot 17} = 50 \text{ MHz}$$

Here's a table of frequencies and ideal PLL settings to achieve them: **TODO**

Play around with the multipliers and divisors in the PLL to generate a faster (or slower) clock. You may have to consult Xilinx's documentation on the `PLL2E_ADV` primitive. (You can also use the Clocking Wizard IP generator in Vivado to generate this instantiation.) The parameters can't be set arbitrarily and there are a few caveats. The multipliers and divisors must be integers and you must fall within the device's operating frequency range - Vivado will complain if you don't. A few frequencies to try are: 60 MHz, 75 MHz, and 100 MHz.

## 4.2 Critical Path Identification

After running `make impl`, timing analysis will be performed to determine the critical path(s) of your design. The timing tools will automatically figure out the CPU clock timing constraint based on the multiply-divide ratio you used in your PLL.

The critical path can be found by looking in `build/impl/post_route_timing_summary.rpt`. Look for the paths within your CPU (`From Clock: cpu_clk, To Clock: cpu_clk`).

For each timing path look for the attribute called "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.

You can also see (And even visualise) the actual logic path that starts at the source and follows some logic in your design until it gets to the destination.

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

### 4.2.1 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 easy to meet and the tools will only attempt to optimize routing until timing is met, and will then stop.

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

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.

## 4.3 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 how many design points you explored. Keep notes of your architecture modifications in the process of optimization. Consider, but don't obsess, over area usage when optimizing (keep records though).

# 5 Optimizations, Extra Credit, and Grading

**All groups must complete the final checkoff by December 9 (by appointment).** Use the week prior to your final checkoff for code cleanup, optimizations, late checkpoints, and optional extra credit projects.

## 5.1 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 audio synthesizer. 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:

$$\text{Cost} = C_{\text{LUT}} \times \#\text{ of LUTs} + C_{\text{RAMB}} \times \#\text{of RAMBs} + C_{\text{REG}} \times \#\text{of Slice Registers}$$

where $C_{\text{LUT}}$, $C_{\text{RAMB}}$, and $C_{\text{REG}}$ 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.

## 5.2 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 6) at the end of this document.

## 5.3 Style: Organization, Design

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

## 5.4 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.

### 5.4.1 Report Details

You will turn in your project report on Gradescope by the final checkoff date. The report should be around 8 pages total with around 5 pages of text and 3 pages of figures ($\pm$ a few pages on each). 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. ($\approx$ 0.5 page)
- **High-level organization**. How is your project broken down into pieces. Block diagram level-description. We are most interested in how you broke the CPU datapath and control down into submodules, since the code for the later checkpoints will be pretty consistent across all groups. Please include an updated block diagram ($\approx$ 1 page).
- **Detailed Description of Sub-pieces**. Describe how your circuits work. Concentrate here on novel or non-standard circuits. Also, focus your attention on the parts of the design that were not supplied to you by the teaching staff. For instance, describe the details of your FIFOs, audio synthesizer, and any extra credit work. ($\approx$ 2 pages).
- **Status and Results**. What is working and what is not? At what frequency (50MHz or greater) does your design run? Do certain checkpoints work at a higher clock speed while others only run at 50 MHz? Please also provide the 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). ($\approx$ 1-2 pages).
- **Conclusions**. What have you learned from this experience? How would you do it different next time? ($\approx$ 0.5 page).
- **Division of Labor. This section is mandatory. Each team member will turn in a separate document from this part only**. The submission for this document will also be on Gradescope. How did you organize yourselves as a team. Exactly who did what? Did

both partners contribute equally? Please note your team number next to your name at the top. ($\approx$ 0.5 page).

When we grade your report, we will grade for clarity, organization, and grammar. Make sure to proofread and correct mistakes before turning it in. Submit your report to the Gradescope 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 Gradescope assignment.

### 5.5 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: Capture mic input from the Pynq's microphone and wire it to the CPU via MMIO
- RISC-V M Extension: Extend the processor with a hardware multiplier and divider

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

### 5.6 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.

# 6 Project Timeline

| Checkpoint | Deliverable | Due Date |
|---|---|---|
| 1 & 2: RISCV151 Processor | Design Review<br>In-Lab Checkoff | November 1<br>November 22 |
| 3: IOs, FIFOs, PWM Controller, Synth | In-Lab Checkoff<br>Project Interview | December 9 |
| Final Checkoff, Extra Credit, and Optimizations | In-Lab Checkoff<br>Github code submission | December 9 (by appointment) |
| Final Report | Gradescope submission | December 11 |

Table 11: EECS151 Fall 2019 Project Timeline

# A Local Development

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

## A.1 Linux

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

To install `elf2hex`:

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

## A.2 OSX, Windows

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

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

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

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

# B BIOS

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

## B.1 Background

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

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

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

## B.2 Loading the BIOS

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

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

```
screen $SERIALTTY 115200
```

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

## B.3 Loading Your Own Programs

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

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

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

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

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

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

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

## B.4 The BIOS Program

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

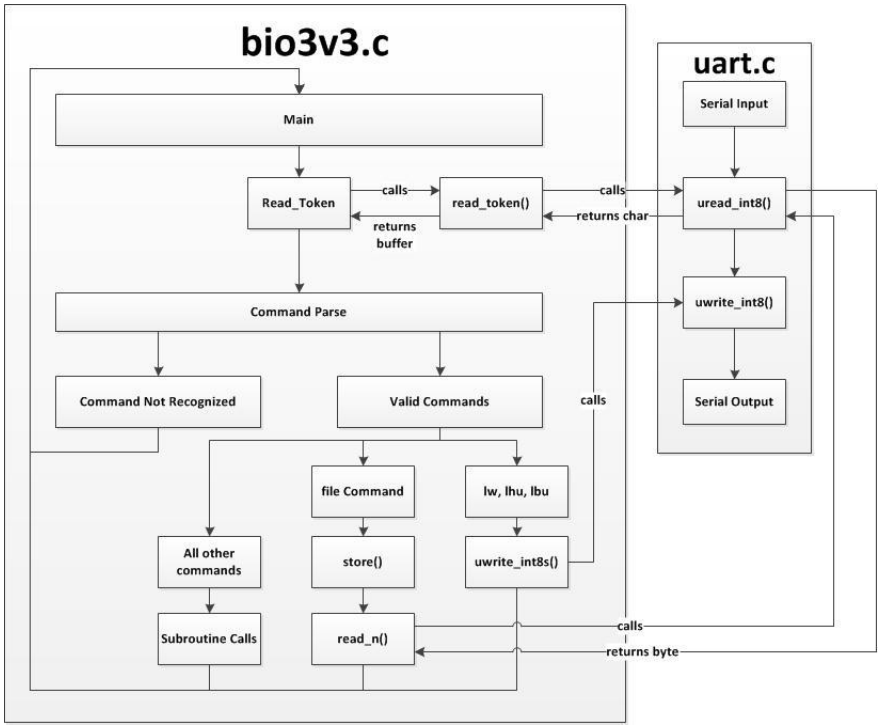



Figure 5: BIOS Execution Flow

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

## B.5 The UART

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

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

## B.6 Command List

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

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

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

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

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

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

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

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

There is another command file in the main() method that is used only when you execute `hex_to_serial`. When you execute `hex_to_serial`, your workstation will initiate a byte transfer by calling this command in the BIOS. Therefore, dont mess with this command too much as it is one of the more critical components of your BIOS.

## B.7 Adding Your Own Features

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