# EECS 151 FPGA Lab: Final Project Report

Keyi Hu and Jessie Mindel (Team 24)

May 9, 2022

## 1 Design Requirements

In this project, we designed and implemented a three-stage pipelined 32-bit RISC-V CPU with a UART for tethering, and synthesized it for deployment to a Xilinx PYNQ-Z1 FPGA board. It supports the full RV32I Base Instruction Set, as well as two commands from the Zicsr Standard Extension (`csrrw` and `csrrwi` to one preset CSR register, 0x51E). The goal of this project was to gain experience in creating effective, elegant digital designs, and to optimize the performance of this CPU, measured by the number of cycles per instruction (CPI), by maximizing the Iron Law and minimizing resource utilization to reduce cost.

### 1.1 Pipelining

The number of pipeline stages is a crucial design decision in creating an efficient CPU. We opted for a three-stage pipeline for its simplicity and the ease and efficiency with which it seemed to handle control and data hazards. We discuss our forwarding logic in Section 3.4, and handle all control hazards by stalling for a single cycle (Section 3.5).

### 1.2 Memory Hierarchy and Memory-Mapped I/O

The CPU's 32-bit address space is partitioned into four main sections, described below. The first four bits of an address indicate to which partition it points.

- **Data memory (DMEM):** Stores programs' data. Any program can read and write to it.
- **Instruction memory (IMEM):** Stores the instructions for the current program. Only the BIOS program can write new instructions.
- **Basic input/output system (BIOS):** Read-only memory storing the BIOS program, responsible for retrieving new instructions over a UART serial connection and inserting them into IMEM. The program runs in a loop until a command is sent to jump (move the program counter, or PC) to an IMEM address.
- **Memory-mapped I/O (MMIO):** Addresses pointing to peripherals and specialized registers, separated into readable and writable addresses. Our system has six: UART control data (`rx_out_valid`, `tx_in_ready`) (read), UART receiver data (read), UART transmitter data (write), a counter for the number of cycles elapsed (read), a counter for the number of instructions executed (read), and an input for resetting the aforementioned counters (write). Here, the full address, not just the prefix, determines the specific register or peripheral to which to route the load or store instruction.

## 2 High-Level Organization

To support such a large project and maintain organization, a detailed design is necessary. We used a block diagram to represent the RISC-V CPU.

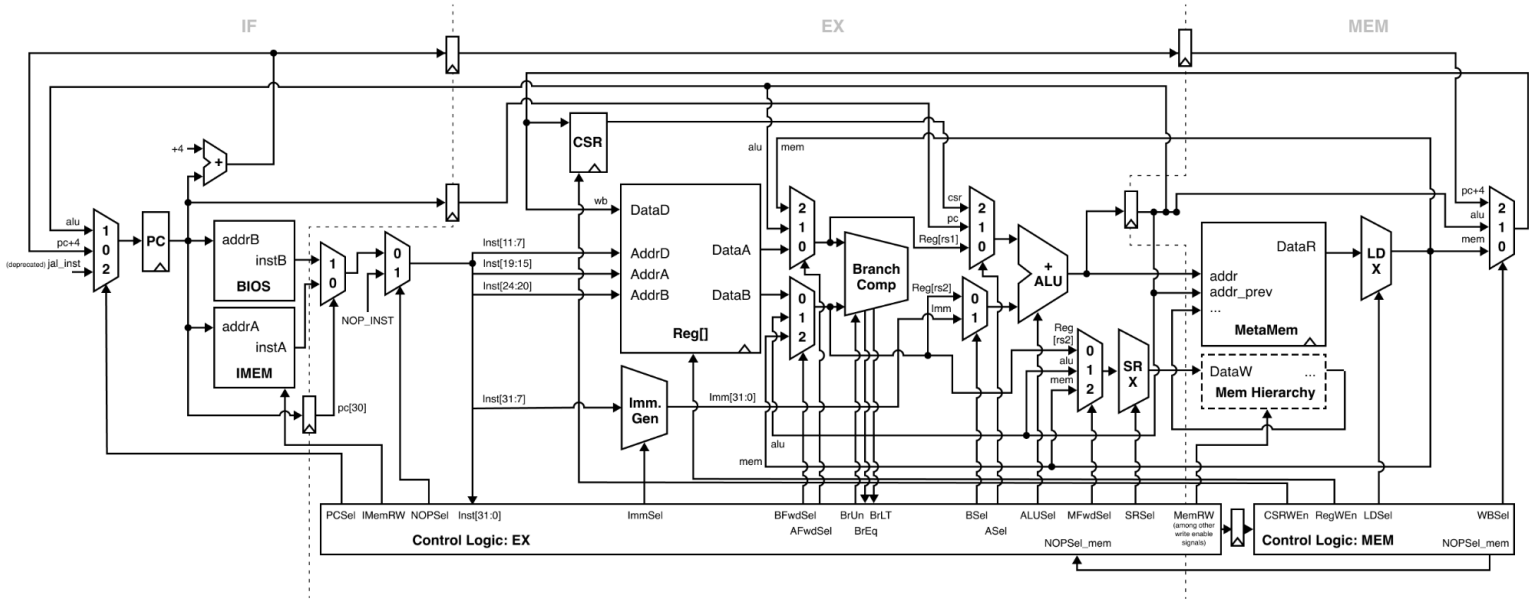



**Figure 2:** Block diagram depicting all but the memory hierarchy (see Figure 3).

Figure 2 represents our design of a three stage pipeline. The three stages are as follows: IF (fetch instruction) stage, EX (execute), and MEM (memory and writeback) stage. These were chosen to resemble combinations of stages commonly used in five-stage RISC-V pipelines. We use registers to separate the stages, except with memory (memory modules have registers in their own modules, so there is no register marked in the block diagram). The block diagram has basic modules for the datapath such as PC registers, ALU, immediate generator, memory block, regfile, etc. The control logic is also separated into three stages by registers to ensure that the correct control signals (i.e., corresponding to the correct instruction) are being used per stage.

The first stage is IF (instruction fetch) stage. In the first stage, we will need to select the correct address so that we can load an instruction from either IMEM or BIOS memory, depending on the prefix of the current PC, and pass it to the next stage.

The second stage is EX (execution) stage. We should have an instruction passed in from the previous stage so that they can fetch values from correlated registers in the register file. It will also generate the necessary immediate. In this stage, values will be going through all numeric operations, and will pass the result to the MEM stage through pipelined registers. Some

writeback will also occur: the EX stage is responsible for writeback to the PC during branch and jump instructions.

The last stage is the MEM stage. The ALU result passed in is used to determine which memory address to which to read or write, and the data we would like to write is determined by Reg[rs2] or by forwarding. In this stage, the datapath will write back to the addressed memory or register. We use an LD X module to mask data loaded from memory for load instructions with different bit widths (e.g., lw vs. lb).

Both the first stage (IF) and last stage (MEM) involve interaction with memory. However, the first stage only reads instructions from BIOS and IMEM memory. In the last stage, MEM will take the responsibility for communication to all memory. To distinguish the memory blocks across the full memory hierarchy, we partition them using the mapping discussed in Section 1.2. To abstract this logic, we built a module called MetaMem, depicted in Figure 3, which we discuss in Section 3.1.

Finally, we added forwarding logic and muxes with control logic to resolve any data hazards and control hazards; we discuss these in Sections 3.4 and 3.5.

# 3 Sub-Pieces and Design Choices

A large majority of the modules in our design are standard: the immediate generator, ALU, branch comparator, load mask (LD X), register file, memories, registers, and muxes are common components. In the sections that follow, we explain the non-standard modules we developed and any logic connecting them.

### 3.1 Wrapping the Memory Hierarchy: MetaMem

Given the somewhat large number of memory modules included in our memory hierarchy, we created a wrapper to simplify write (store) permissions and read (load) outputs. `MetaMem` takes as inputs the current address from which to read or write `addr`, the previous cycle's address from which to read or write `addr_prev`, and the output of each memory module (DMEM, IMEM, BIOS, UART's RX out, RX valid, and TX ready, the cycle counter register, and the instruction counter register). It uses these inputs to calculate two selectors, `MapRSel` (from which memory module it's reading, if any; calculated using `addr_prev` in the MEM stage) and `MapWSel` (to which memory module it's writing, if any; calculated using `addr` in the EX stage), which it uses to determine which input to send to `DataR`. It also passes these selector values along as outputs for use in read and write permissions logic. Finally, `MetaMem` is also responsible for truncating `addr` into the appropriate addresses for DMEM, IMEM, and BIOS, whose address spaces are not 32-bit.

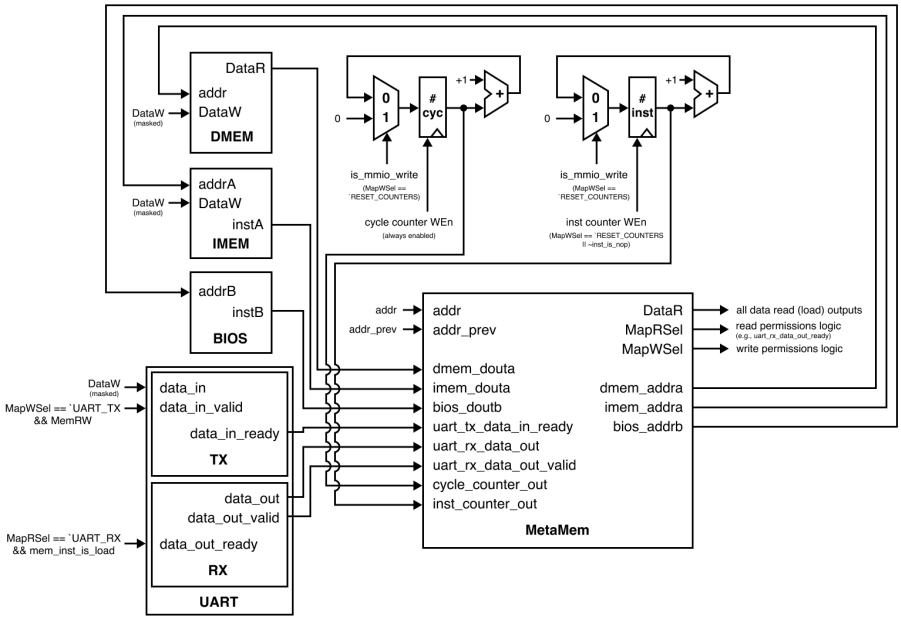



**Figure 3:** MetaMem and MMIO Logic

Write operations are handled independently, since different memory modules store values of different widths. `DataW`, once processed (see Section 3.2), is sent into DMEM and IMEM's `DataW` input, and its first 8 bits are sent to UART TX's `data_in` input.

### 3.2 Shifting and Masking Memory Inputs: Store Mask (SR X)

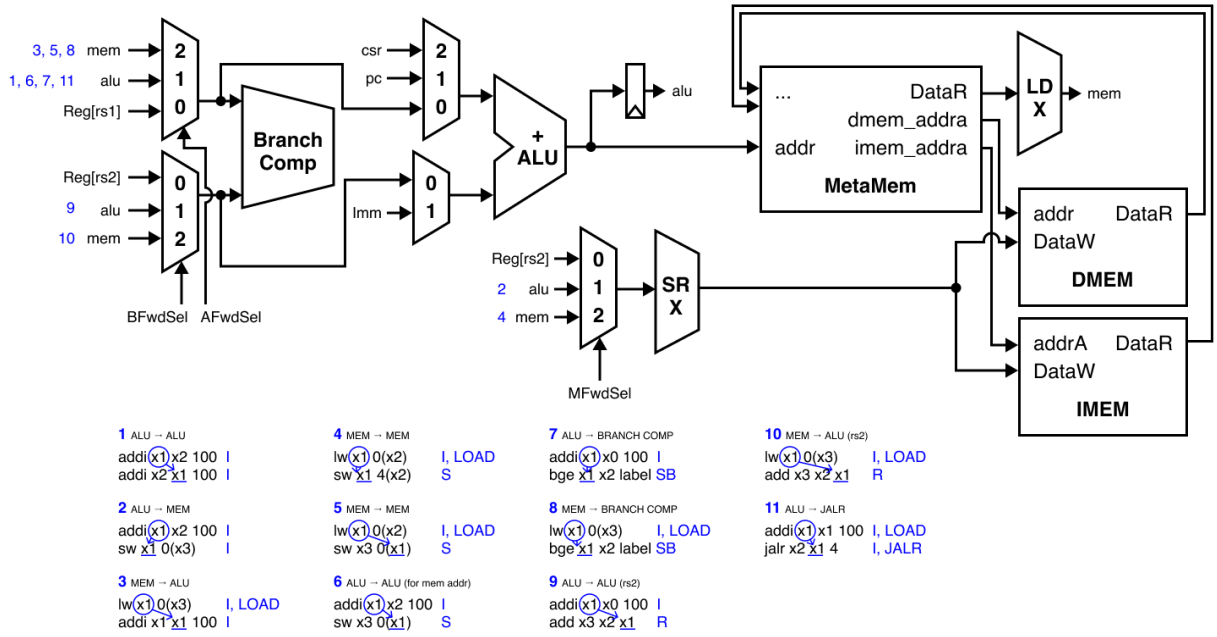
Since RAMs have 32-bit rows, in order to load or store a byte or halfword rather than a full 32-bit word, we need to mask and shift the data. The load mask (LD X) module already performs

this for load operations. For store operations, we created a new module, store mask (SR X), which first shifts and masks the data to store based on the current operation using a selector from `Control Logic EX`, `SRSel`, before sending it to DMEM and IMEM's `DataW`. We do not allow misaligned halfword operations (i.e., only a 0- or 2-bit offset is valid in `lh`, `lhu`, or `sh`).

### 3.3 Memory-Mapped I/O

Since MetaMem partitions the 32-bit address space of the memory hierarchy, it also handles memory-mapped I/O; it manages both permissions and read operations for each peripheral. Figure 3 depicts how each peripheral (UART, the instruction count register, and the cycle count register) is connected to MetaMem.

### 3.4 Forwarding Control Logic: EX




Our forwarding logic occurs in the EX stage, using the pipelined ALU output from the MEM stage and the MEM's masked output. We use three muxes to forward data: one feeding into the ALU and branch comparator's first inputs, one feeding into the ALU and branch comparator's second inputs, and one feeding into MetaMem's DataW. In order to determine whether to forward data, our control logic first looks for overlap between rd and either rs1 or rs2 (for commands that have them, i.e., store and branch do not use rd; jal and U do not use rs1; and only R, S, and SB instructions use rs2). If such an overlap exists, then we determine from where to forward (`alu` or `mem`, pictured above), and to where to forward (Reg[rs1], Reg[rs2], or DataW), based on whether the current instruction (i.e., currently in EX) is a store instruction, and whether the previous instruction (i.e., currently in MEM) is a load instruction. In the figure above, we present exhaustive cases for forwarding, annotating a simplified display of our design.

### 3.5 Writeback Control Logic: EX and MEM

Many memory modules need to be written to at different stages of the pipeline. Because DMEM, IMEM, and BIOS all have registers built in, it takes one cycle to read from or write to each. As such, these modules receive inputs in the EX stage and produce outputs in the MEM stage, which means that their write permissions must be determined during EX. The same is true for UART's TX input, and the cycle and instruction count registers. Each of these modules has a write enable value and/or byte mask:

- **DMEM** (mask)**:** If the address's prefix is in DMEM's partition for a store instruction, create a mask based on whether the instruction is sb, sh, or sw, and its offset. 0 otherwise.
- **IMEM** (mask)**:** Same as DMEM, but if the prefix is in IMEM's partition.
- **BIOS:** Writing is never allowed.
- **TX input valid (and write enabled)** (bool)**:** The address is exactly the correct MMIO address, and the instruction is a store instruction of any kind.
- **Cycle counter** (bool)**:** Always enabled; see Figure 3.
- **Instruction counter** (bool)**:** The instruction is not a no-op, or the reset counters MMIO address is being written to; see Figure 3.

The register file and CSR register, on the other hand, receive data during the MEM stage (rather than the EX stage), and as such, their write enable control logic is in MEM (as is `WBSel`, for the same reason). This logic is more straightforward (and produces booleans rather than byte masks):

- **Regfile:** The instruction is R-, I-, U-, or UJ-type, and rd is not x0.
- **CSR:** The instruction is either csrrw or csrrwi.

### 3.6 Control Hazards: NOPSel in EX and MEM

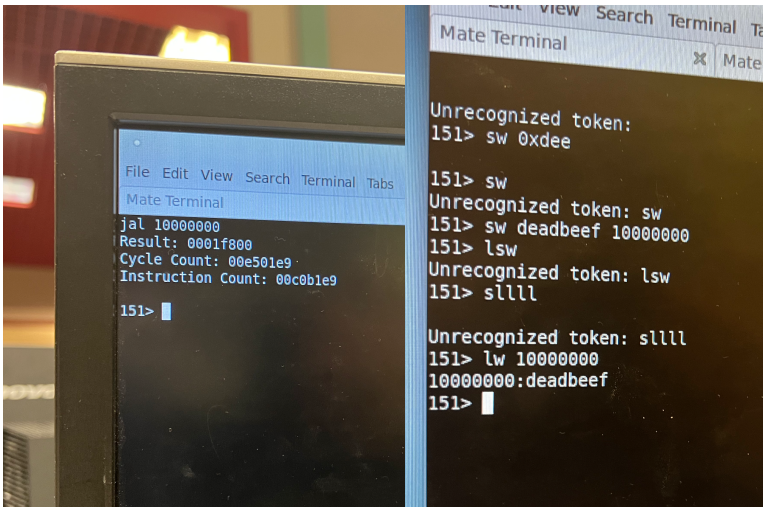
A control hazard is determined to occur during the MEM stage if the instruction in the MEM stage is a jal or jalr instruction, or a branch instruction that has been determined to be taken by pipelined `BrEq` and `BrLT` values. This is stored in the value `NOPSel_mem`, which Control Logic EX receives directly without register-based delay, and outputs through the value `NOPSel`. If `NOPSel` goes high, using a mux, rather than the instruction from IMEM or BIOS being used at the start of the EX stage, a hardcoded no-op instruction is used. This bubbles through the MEM stage in the following cycle. Since the ALU output from the EX stage gets written back to the PC register, by the time one cycle has passed, the correct next instruction is already ready. As such, we only need to stall for one cycle.

### 3.7 Other Control Logic

Notably, `PCSel`, which controls which value is written to the PC register, is in the EX stage, since PC register values are written back from branch and jump instructions during the EX stage.

Early in the design process, we thought we had come upon a potential optimization: if we had determined the instruction during the IF stage, then we could immediately determine whether it were a jump instruction, and if so, calculate and immediately set the next PC to the correct value, bypassing control hazards for jump instructions. However, this was founded on a misunderstanding; we did not initially realize that reading the instruction took one cycle, and as such, that we could not determine whether it was a jump instruction until the EX stage. As such, any use we once had for Control Logic IF is now gone; it remains in our codebase as a vestigial module.

# 4 Status and Results




## 4.1 Functionality

Our implementation successfully passes all provided tests in simulation (`Riscv151_testbench`, `assembly_testbench`, `isa_testbench` on all instructions, `c_testbench`, `software_testbench` on all programs supplied, `echo_testbench`, and `bios_testbench`), as confirmed during Checkoff 3, as well as a few that we wrote while exploring edge cases. The only piece of functionality which requires further debugging is the MMIO address for resetting the instruction and cycle counters; the counters do not seem to reset correctly after that address is written to. The BIOS program successfully runs in hardware on the Pynq-Z1; shown above is a successful execution of `mmult` and a successful store and load to DMEM.

## 4.2 Performance on `mmult`

As shown above, `mmult` ran in 15008233 cycles, and executed 12628457 instructions. This yields a CPI of 1.188, lower than the desired benchmark of 1.2 cycles per instruction. The clock has a period of 20 ns, running at a frequency of 50 MHz.

Due to time constraints and a Vivado board manager bug that prevented us from synthesizing, we did not perform any optimization, and could not determine the critical period and area usage of our CPU. However, had we begun to optimize the CPU, we might have started with the following approaches:

- Determine the critical path, and change the clock period accordingly. Iteratively test the CPU in hardware to ensure that it still functions at the new clock speed, until it fails to function, at which point, we would have found our minimum clock period.
- Simplify the forwarding logic. It's possible that our use of three muxes was unnecessary; the input DataW and output of Reg[rs2] might be mergeable into one mux, since store instructions only use immediates in ALU operations (rather than both rs1 and rs2).
    - We have also heard from peers that removing the forwarding logic altogether greatly simplifies the hardware complexity. We might attempt this and add stalls for data hazards.
    - Other elements of the design could easily be similarly polished: some wires and entire modules have become redundant as we have iterated and debugged.
- Build a branch prediction module to lessen the chance of control hazards.

## 5 Conclusions

It has been challenging but extremely rewarding to design and implement a RISC-V CPU nearly from scratch. Given the project's large scope, we learned to develop an effective design early on in order to break the project down into smaller pieces like submodules. Whenever we identified a mistake, we could quickly differentiate between an error in our Verilog implementation or in our design, and could always return to our design to maintain organization or revisit a prior idea. By keeping our design up-to-date with our code and our thinking, we were able to continually focus on the details and while still having clarity about the big-picture structure.

We also learned to use unit testing on individual modules before integrating everything together at the top level. Unit tests can help ensure the behavior of our submodule, especially if we're rigorous about edge cases, such that we can be certain that a bug is occurring at the top level and not in a module once we run tests on the full system. Spending a substantial amount of time debugging in simulation was helpful in developing a new perspective on system design, too: whereas debugging software more often requires linearly stepping through a set of logical statement, debugging hardware using waveforms requires a more parallel mode of analysis and thinking (interesting parallels can be drawn to some visual software languages, e.g., the music- and graphics-oriented language MaxMSP). When working on hardware, software, and other engineered or artistic systems in the future, this perspective will continue to be helpful.

At times, the process of uncovering new ideas or a better understanding of the project surprised us. We came to realizations about additional pipeline stages, the placement of control logic, and more, both while revisiting our design and while debugging. The process felt fundamentally exploratory. This sits squarely between two very different pieces of advice we have received from two different fields: from peers in this course, the legendary ability to design something incredible, "turn off your brain," and implement it; and in product design, to iterate constantly, and to fail sometimes intentionally. We find the interplay between these to be interesting, and saw both as helpful during this project; we look forward to continuing to find a balance between these approaches in future work.

Teamwork has been extremely important in this project. We have coordinated well with each other. However, we could have scheduled our time better and communicated more frequently and earlier about the project's checkpoints. Especially given these setbacks, we're grateful to the teaching team for their support and dedication all throughout.

Working on this project has made digital design feel much more accessible, even if extremely difficult, and we look forward to continuing to explore this field and/or apply the experience we have gained to other systems we hope to develop. Having this project as a playground for future exploration is an exciting premise.