# EECS 151 Final Project Report

**Scott Shao, Wade Burns, Group 16**

## 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

The goal of this project is for EECS151/251A students to become familiar with the methods and tools of digital design, and the implementation of that goal is a custom designed three-stage pipelined RISC-V CPU with a UART that supports a selected set of the 32-bit RISC-V ISA.

The three stages of the RISC-V CPU are Fetch, Execution, and Writeback. In general, instructions are executed as follows: during the Fetch stage, an instruction is fetched from IMEM (instruction memory); during the Execution stage, the values of the registers specified in the instruction are executed (with whatever operation was specified) to get a data output result from ALU (this data can be used as the result of a calculation or a memory address); during the Writeback stage, the data is stored to or loaded from the DMEM (data memory) and the value is possibly written back to a register in reg_file. There are three memories in this implementation: read-only bios memory (BIOS mem), instructions memory (IMEM), and data memory (DMEM). The BIOS mem has two synchronous read port; the DMEM has two synchronous read ports and one synchronous write port; the IMEM has one synchronous read port and one synchronous write port. The register sub-module (reg_file) has two asynchronous read ports and one synchronous write port.

The CPU in this project is implemented onto a Pynq-Z1 FPGA board and supports communication between the CPU and the host PC using a UART interface in the CPU. All the I/Os in this project are memory mapped to an I/O memory and can be accessed using store and load instructions with associated addresses. In addition to the RISC-V CPU itself, the project also contains a PWM controller, an audio synthesizer, a button parser, and a build-in PMU.

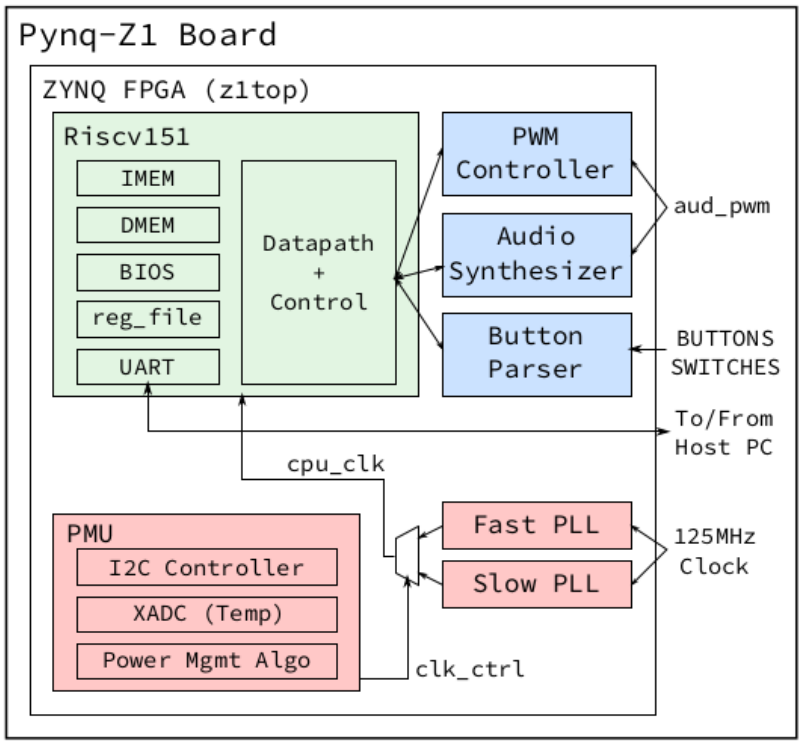



Figure 1: High-level overview of the full system.

## High-level Organization

During the Fetch stage, the program counter (PC) is selected from a five-mux (five input multiplexer) with inputs from ALU, PC + 4, PC, RESET_PC, and JalPCGen (PC generator for the jump and link operation). The PC is then fed into both BIOS mem and IMEM for synchronous instruction fetch and the output is then selected using a two-mux with PC[30] bit as control. The output from the two-mux is then fed into another two-mux to handle instruction stall when the branch prediction is incorrect. The JalPCGen sub-module in the Fetch stage calculates the output PC for JAL instruction using current PC and instruction to avoid stalling.

During the Execute stage, in the case of register-register operations, two values, one from each of the two registers rs1 and rs2, are read from RegFile asynchronously according to the current instruction from instruction register, and the data are then fed into two four-mux to handle data forwarding. The two forwarding mux supports ALU-ALU forwarding, memory-ALU forwarding, and immediate-ALU forwarding. The ALU then takes in two inputs from two two-mux's to select either PC or rs1 for the first input and immediate or rs2 for the second input. The ALU submodule supports add, subtraction, shift left logical, set less than, set less than unsigned, bit-wise xor, bit-wise or, bit-wise and, shift right logical, and shift right arithmetic. The ALU also outputs the least significant two bits of the value calculated to store as the store offset (for byte and half-word write operations). The store sub-module takes in the offset from ALU and rs2 to calculate the four bit mask and shifted data for memory write. The immediate select two-mux chooses between immediate output from immediate generation and rs1 for the CSR (control status register, used in testing/de-bugging) in the Writeback stage. The immediate generation sub-module takes in the instruction and outputs the appropriate immediate based on the immediate type which is decided by the control logic. The Execute stage also contains the branch comparator. The branch comparator takes in the rs1 and rs2, and branch type control to determine whether the branch is taken. A branch is defaulted to not taken; if the branch comparator outputs that the branch should be taken,

the next instruction will be replaced with a nop (an operation that changes nothing) at stall two-mux to flush out the wrong instruction.

During the Writeback stage, the external I/O devices such as UART, button parser, and LED's are mapped to a memory mapped I/O memory(IO mem) such that the I/O operations will look like memory store and load for the CPU. The IO mem also contains the cycle counter and the instruction counter. These counters share the same access pattern as any other memory mapped I/Os and will output to the load sub-module when the associated memory address is accessed. All the memory ports will synchronously have a value at their input every clock cycle and the data access are controlled using write and read enable control signals. The BIOS mem takes in the output from ALU as a memory address to be read and outputs the value stored in the address. The DMEM takes the output from ALU as an address to be read from or written to and the output from the store sub-module as the write data and mask. If the instruction is load, the mask from store sub-module will be all zero to ensure no data will be stored to memory when the memory control is high. The IMEM also takes in the output from ALU as the write address and output from store sub-module as the write data and mask. The IO mem takes in the output from ALU as the address, output from store as the write data and the mask, and the current instruction for the instruction counter. The IO mem also takes in the output from mapped I/O devices and outputs to mapped I/O devices directly for I/O operations. All the memories output data synchronously and the appropriate data is selected using a three-mux with the data address[31:30] top two bits as the control. The correct data output is then fed into the load sub-module along with the bottom two bits of the ALU output address as the mask. The load sub-module then outputs the correct data based on the mask and the load instruction type. Finally, a four-mux selects the correct data type for the register writeback. This four-mux takes in the immediate, ALU, load sub-module, as well as PC + 4. If the register writeback is enabled, the correct data output from the four-mux is then written to the RegFile synchronously with the writeback address from the current instruction. The CSR sub-module is also in the Writeback stage. The CSR sub-module will handle the debug instructions CSRRW and CSRRWI. The CSR sub-module takes in the current instructions and immediate. When the instruction is a CSR type, the immediate select two-mux in the Execute stage will select between the immediate and the rs1 for the CSRRWI and CSRRW instruction, respectively.

The control module takes in values from different stages of the pipelines to create a control logic that is utilizes both combinatorial and sequential logic. The control logic is also staged using registers to follow the instructions at different stages. The inputs of the control logic are clock, reset, and buttons from CPU, the instruction in Fetch stage, the instruction in Execute stage, PC in Fetch stage, the address calculated from ALU in Execute stage, and output from branch comparator in Execute stage. All control logic outputs are calculated using combinatorial logic. All instructions take one clock cycle except for the branch and JALR instruction. The control logic assumes the branch is not taken and stall for one cycle if the branch is taken, and the JALR instruction always takes two clock cycles to complete, because it has to wait for the address output from the ALU in the Execute stage. The control logic also controls the reading and writing access to the memory mapped I/O just like all the other memories with read and write enable signals. A separate IO mem sub-module is used for the actual reading and writing control logic using sequential logic similar to the bios and data memory.

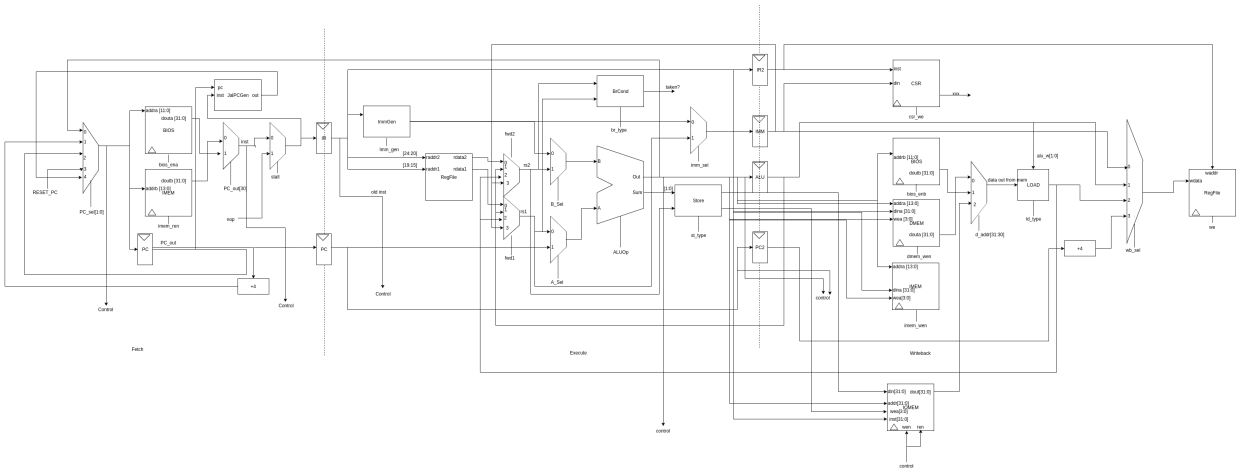


Figure 2: Three-stage RISC-V CPU datapath.

## 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. (≈ 2 pages).

- Multiplexers: All multipliers including two-mux, four-mux, and five-mux are implemented using one always @(*) block and case statement controlled by control signal from the control logic.
- Flip flop: All flip flops are implemented as sequential logic with one always @(posedge clk) block.
- JalPCGen: JAL instruction program counter generator uses one combinatorial logic to calculate the UJ type immediate.
- ImmGen: Immediate generator uses one combinatorial logic and case statement to calculate the R, I, S, B, U, J, and CSRI immediate and is controlled by the output from control logic.
- BrCond: Branch comparator uses one combinatorial logic and case statement controlled by the output from control logic to calculate whether the branch should be taken for BEQ, BNE, BLT, BGE, BLTU, and BGEU instruction and send the result to the control logic.
- ALU: Arithmetic logic unit uses one combinatorial logic and case statement controlled by the output from control logic to calculate the output for ADD, SUB, SLL, SLT, SLTU, XOR, OR, AND, SRL, and SRA operations. ALU also outputs the last two bits of the calculated value as the mask for store.
- Store: Store uses one combinatorial logic and cases statement controlled by the output from control logic to calculate the four bit mask for the memory access and shifts the output from ALU according the the mask.
- CSR: The CSR sub-module uses one sequential logic to write to the CSR register. When the write enable control is high and the rd address equals 12'h51e, the input is write to the CSR register. Since the input is already selected by the immediate select two-mux to give the correct data for CSRRW and CSRRWI, there is no need to check in the CSR sub-module.
- I/O memory: The I/O memory sub-module is the interface between all memory mapped I/O devices and the CPU. The I/O memory uses sequential logic and control signals from control logic to read from I/O devices and write to I/O devices. I/O memory also contains cycle counter that interments every clock cycle and instruction counter that interments every clock cycle when the instruction is not a nop. When a load instruction is performed on a memory mapped I/O devices, the data from the I/O devices are output to the data out port of the I/O

memory at the rising edge of the clock. At every rising edge of the clock, if the instruction is a store instruction to a I/O device, the data from the store sub-module will store to the I/O device directly byte-wise using mask.

- Load: Load uses one combinatorial logic, cases statement controlled by the output from control logic, and mask to shift and extend the data for LB, LH, LW, LBU, and LHU instructions.
- Bios memory: Bios memory has two synchronous read ports. The data will output from the bios memory at the rising edge of the clock if the controlling signal is high from the control logic. When the system is synthesized and program to the FPGA, the bios hex file will write to the bios memory directly since the bios memory is a read only memory and cannot be changed after program to the FPGA.
- Instruction memory: Instruction memory has one synchronous read port and one synchronous write port. There is no control for the instruction read, at every rising edge of the clock the instruction will output from the memory. When the write enable control is high, the data will write to the instruction memory byte-wise using mask at rising edge of the clock cycle.
- Data memory: Data memory has one synchronous read port and one synchronous write port. Similar to instruction memory, there is no control for the data memory read so at every rising edge of the clock cycle data will output from the data memory. When the write enable is high from control logic, the data will write to the data memory byte-wise using mask at the rising edge of the clock cycle.
- RegFile: Registers File has two asynchronous read ports and one synchronous write port. The two outputs are wires with no control, and at every rising edge of the clock cycle the data will write to the register at write address if writeback control is high from the control logic.
- FIFO: The FIFO buffer has one asynchronous read port and one synchronous write port. The FIFO uses a circular buffer with one write pointer and one read pointer. There is also a status count that increments when write is enabled, read is not enabled, and status count does not equals to the depth of the FIFO. The status count decrements when the read is enabled, write is not enabled, and the status count does not equals to zero. This FIFO is full when status count equals to the depth of FIFO and empty when the status count equals zero. FIFO uses one sequential logic to write to the buffer at the rising edge of the clock when write enable is high and uses one combinatorial logic to read from the FIFO when the read enable is high. This is to account for the difference in FIFO access pattern between write and read.
- pwm_handshake: This is actually a general 4 phase handshake module which through a ready-valid interface, or rather here, a request-acknowledge interface, allows for communicating reliably with two modules that run on different clocks. The received request signal from the received clock time domain is translated synchronously to the transmit clock time domain through two flip-flops. The data from the receive domain is allowed to be transmitted to to the transmit domain once the transmit data register receives the synchronized request bit. At the same time the request bit is sent back to to the receive domain through back to back flip flops to in turn become an acknowledge bit signal.
- NCO: The Number Controlled Oscillator takes in an N-bit frequency control word which is also called a “phase increment” (stored and recieved in the MMIO) because it represents the index increment needed to access the next wave sample stored in a particular wave LUT (Look Up Table). The NCO has access to four of these LUTs, one each for sine, square, triangle, and sawtooth waves.  Ideally each LUT would have 2^N entries or samples for better resolution, but since N here is 24, LUTs of this size are not feasible on our FPGA boards so instead they are of size M=8 and only the top most signifcant bits of the current phase increment are used to index a LUT. Each entry in the wave LUTs are Q16.4 signed fixed point numbers, meaning there are 16 integer bits and 4 decimal bits. Since we have sacrificed resolution for space and speed there will be appreciable errors between samples. To interpolate between samples a residual is used, which is the bottom 16 (N-M) bits of the current phase index. The difference between the current sample and the next is multiplied by this residual and added to the current sample for output to the summer. All four wave samples are output at once.
- Summer: The summer takes the four current wave samples from the NCO as well as attenuation factors for each (stored in the MMIO) and scales them by their appropriate factors. The 4 scaled waves are then summed together into one Q16.4 bit signal and outputs to the global gain module. The global gain module further attenuates the combined waves if specified. Both functions are achieved by properly arithmetic right shifting the fixed point wave number.

- Truncator: The truncator module maps the Q16.4 bit signed composite wave signal to a 12 bit unsigned number for use as a PWM (Pulse Width Modulated) number to control the audio output (if selected). A buffer and another 4 phase handshake is needed to orchestrate the streaming input from the NCO and the time domain shift to the PWM control module, which ultimately outputs the audio. The PWM number is the duty cycle of the square wave output. setting the duty cycle effectively sets the voltage level to a ratio of VDD, the power supply voltage, so the output can be set to any arbitrary value in between (within the resolution) and unique continuous analog wave-forms can be created.

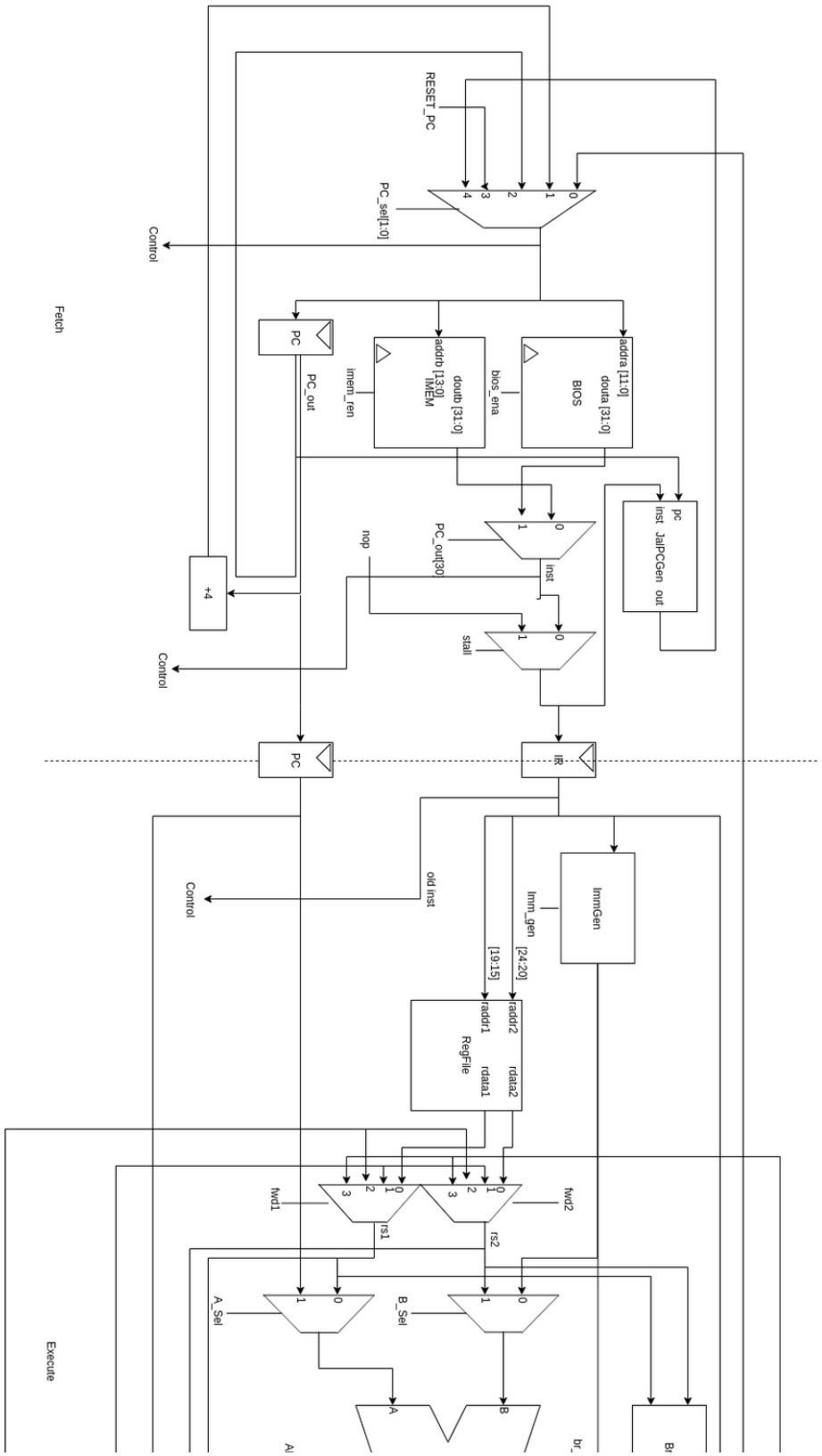
## Status and Results

Unfortunately, we were not able to finish all the system design at the end of the project due date. Our team spent weeks to debug the datapath and control in order to make the system works on a FPGA board. This left us minimal time to design and implement the I/O, piano, subtractive synthesizer, and optimization. The system passed all isa-tests and was able to run reliably on a FPGA board. The bios was working and the echo, mmult, and user_io_test program were able to load onto the FPGA and ran reliably. We are still working on debugging the square_piano program. So far the piano program works if the square_piano.c program set PWM_DUTY_CYCLE to a constant rather than a variable. We also have a design for the subtractive synthesizer. Due to the piano not working and the time constraints we were not be able to incorporate our design into the overall system. Overall the design can run at 50 MHz. The user specified timing constraints can be met at up to 80 MHz. From the device usage statistics report, at 80 MHz, the design used 14 lut1, 148 lut3, 1956 lut_as_logic, 0 lut_as_memory, 1545 slice_registers, 814 lut_in_front_of_the_register_us_unused, 1172 register_driven_from_outside_the_slice, 693 slicel, 913 slice, 220 slicem, 358 lut_in_front_of_the_register. At 50 MHz, mmult program reported cycle count of 00e5026d and instruction count of 00c8c26f. The CPI is 1.1407.

## Conclusions

Although we didn't finish the whole project by the end of the due date, this is a really good and ambitious project and we would love to continue working on it after the finals to finish the design and optimization. One of the most important lessons we learned throughout this project was the importance of testing. At the beginning of the project we were ahead of the schedule, but soon falling behind at checkpoint two. The weeks of long debugging hours before a working design on the FPGA board due to inferred latches errors and small timing bugs taught us the importance of testing. Although we have also wrote testbenches for sub-modules like ALU, pwm handshake, and UART, we relied heavily on the staff provided testbench before checkpoint two. We were not used to writing testbench in verilog and were having difficulties wiring testbenches at the beginning. If we had knew what we know now we would invest a lot more time in designing and implementing robust testbenches for our design. We ended up spending more time trying to find the bug using waveform produced from bad testbench than learning and writing a good testbench to produce a more meaningful waveform for debugging. One of the major challenge we faced before checkpoint two was error produced from inferred latches. We didn't learn enough from the labs about the evil of the inferred latches due to it was never an issue for us in the lab, but during the project the inferred latches error really slowed us down. We spent weeks trying to figure out the bugs and turns out it was due to a single inferred latch. It is a big price to pay to learn to avoid the inferred latches but we are glad we learned it the hard way. This project also showed us the potential of RISC-V ISA. It is amazing that it is possible for a undergraduate class project to be able to implement a simple CPU. We had a lot of fun discussing various ways of unconventional datapath and performance optimization, however, due to time constraints and other classes' project, we were not be able to design and implement everything. This is a pretty ambitious project that requires a good workload and time balance, we would use agile and iterative design principles next time to keep up with the tight schedule and pinpoint the part we need to focus on early on in the project.

# Appendix


RESET_PC
PC_sel[1:0]
Control
Fetch
PC
PC_out
imem_ren
addrb [13:0]
doutb [31:0]
MEM
bios_ena
addra [11:0]
douta [31:0]
BIOS
pc
inst
JalPCGen
out
PC_out[30]
nop
inst
+4
stall
Control
PC
IR
old inst
Control
Imm_gen
[24:20]
[19:15]
ImmGen
raddr1
raddr2
RegFile
rdata1
rdata2
fwd1
fwd2
rs1
rs2
A_Sel
B_Sel
Execute
A
B

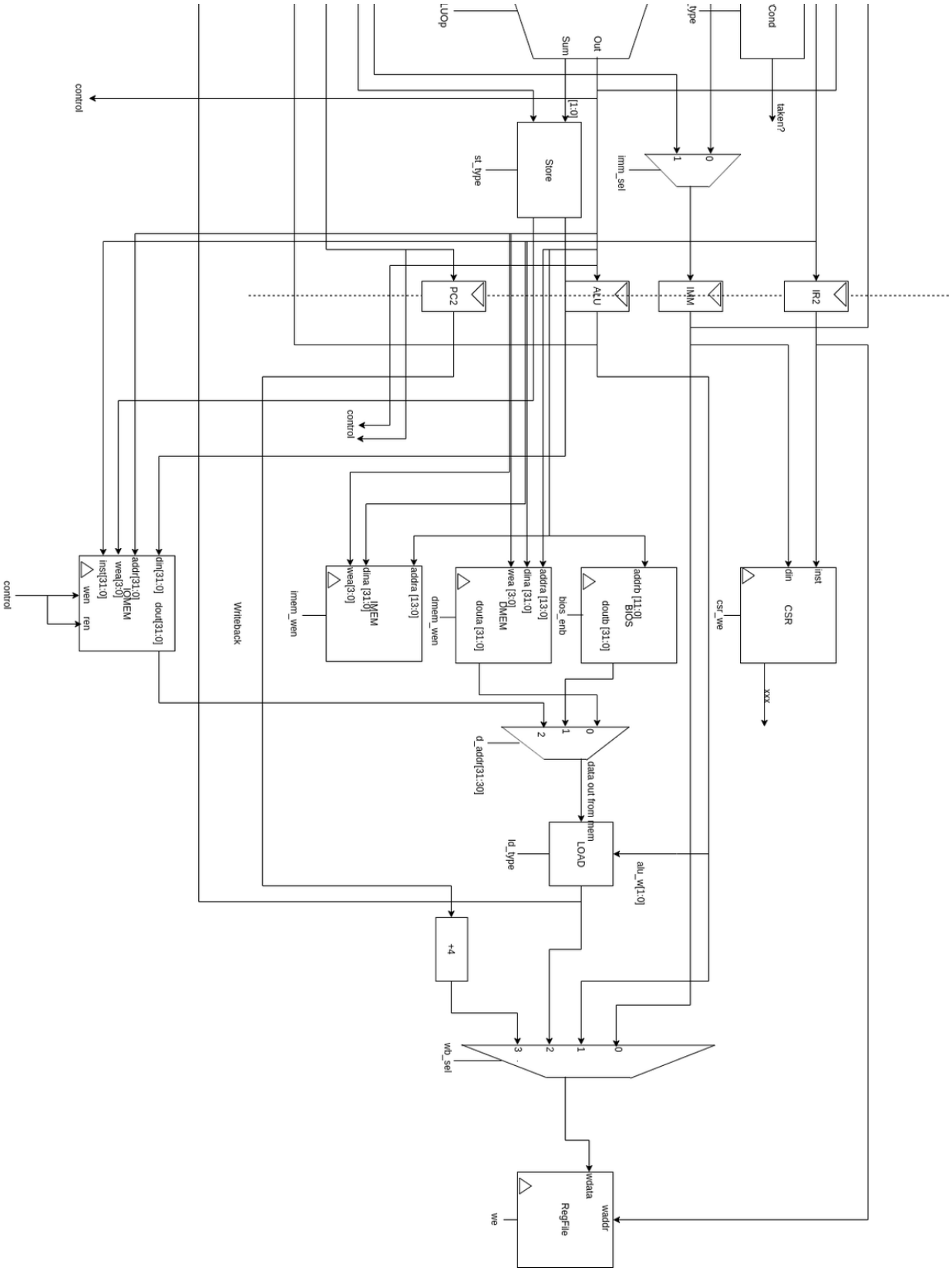



Figure 1: RISC-V CPU datapath.