RISC-V on FPGA in SystemVerilog

Spring/Summer 2022

Introduction:

Our implementation of RISC-V on the DE-10 Lite FPGA includes 31 of the base instructions of the RV32I Base Integer instruction set. The processor fetches a 32-bit instruction stored on on-chip memory and located by the program counter. It is then decoded and the proper operations are executed and the result is stored in a register or memory.

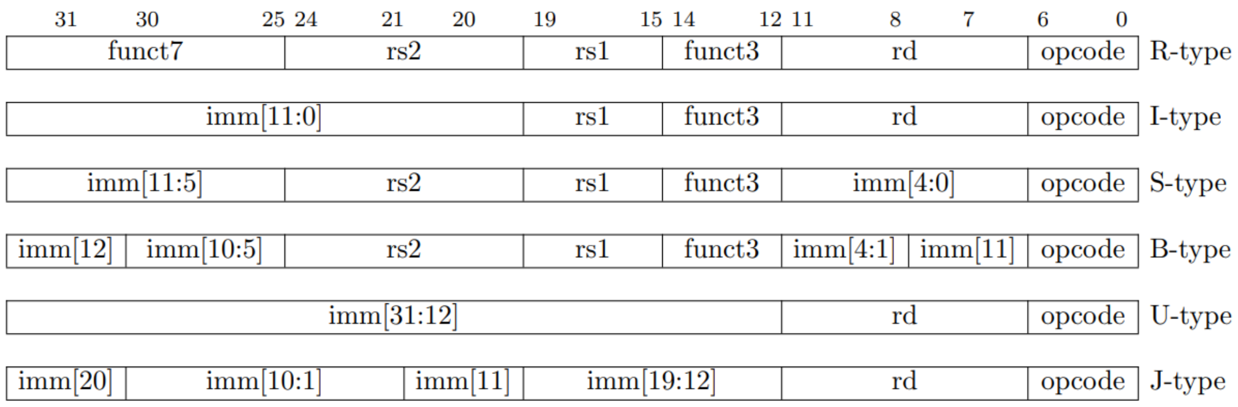
We used many elements of lab 5 to guide our way through the implementation of RISC-V as they were both risc ISAs. We incorporated the same system to read and write to memory as we use MAR, MDR, IR, and PC registers to access instructions and write to memory. There were many things that we had to change as RV32I utilizes 32-bit instructions and uses far more instructions than the SLC3 that we implemented. RISC-V also features byte addressable memory rather than word addressable memory. This allows flexibility to read and write half words and bytes to memory rather than always needing to write the full 32 bits to memory every time.

Our goal for this project was to get the base instruction set of RV32I to work on the FPGA so we could write and test our own code on the board.

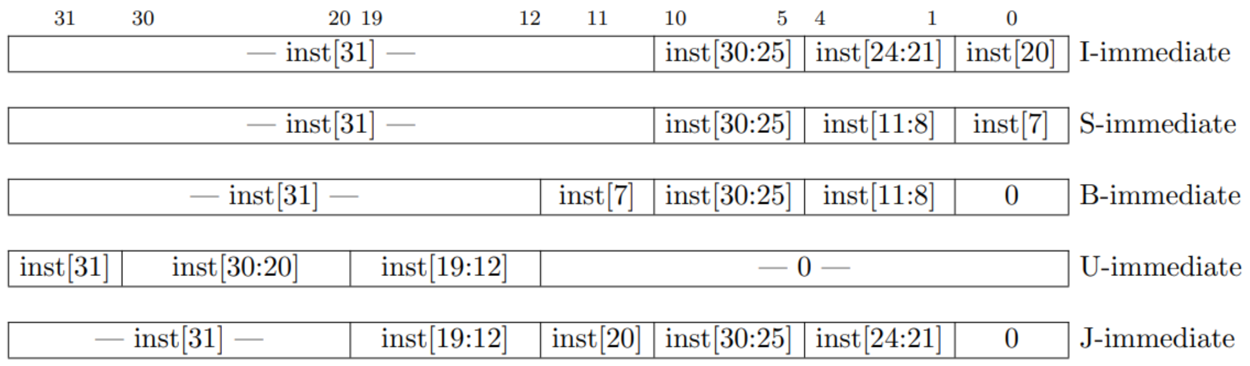
Written Descriptions and Diagram of RISC-V

a. Summary of Operation

We designed a Reduced Instruction Set Computer with a 32-bit processor, program counter, instructions, and registers with SystemVerilog. The RV32I implementation has 32-bit user address space that is byte-addressable. The RV32I processor begins with fetching an instruction from memory starting at address zero. The instruction is loaded into the Instruction Register and then the Program Counter is incremented by four. The 32-bit opcode is decoded and the instruction is executed. The implementation follows a fetch-decode-execute cycle like lab five.

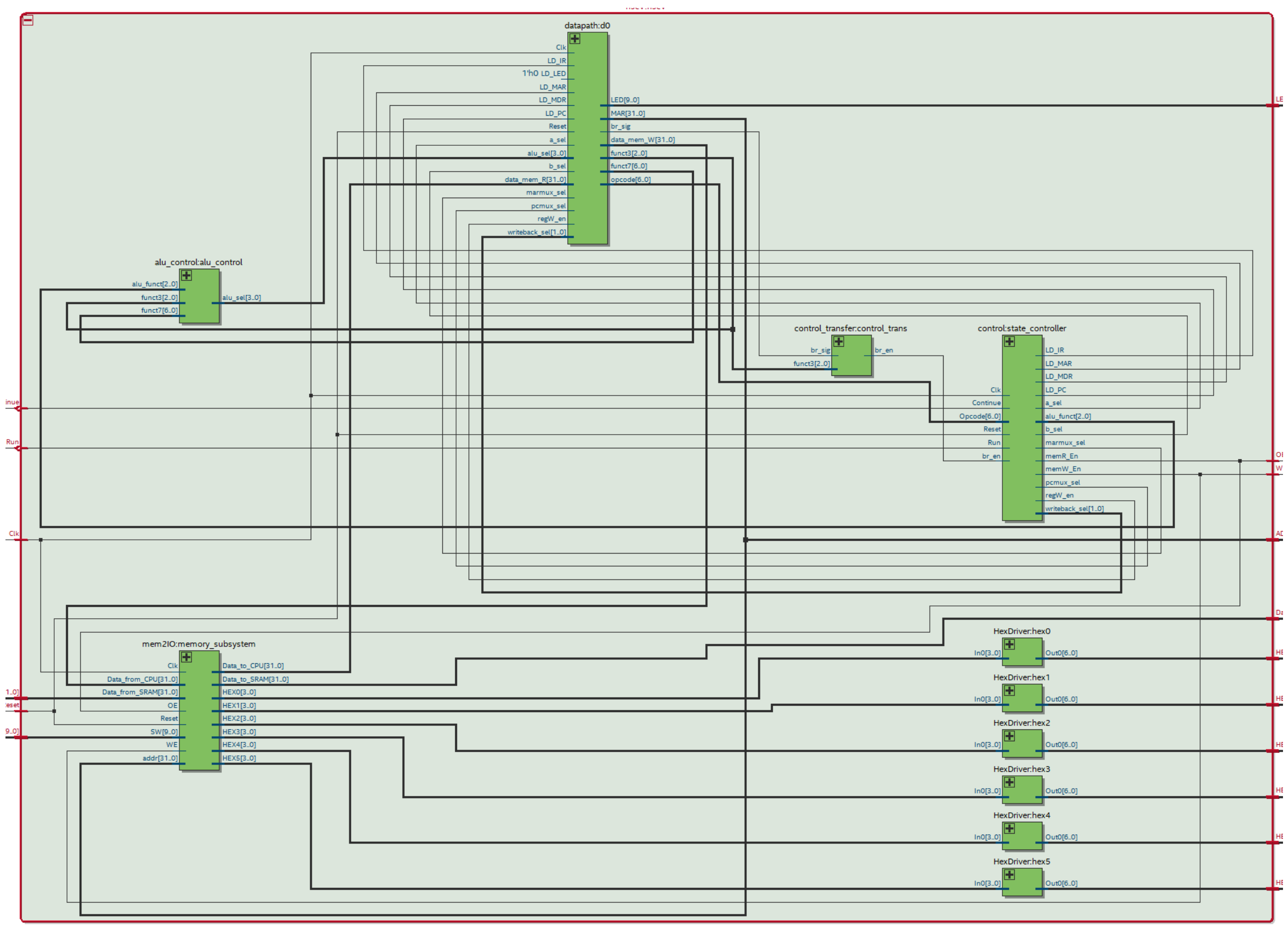


The instructions are formatted into four core instruction types. They are R-type, I-type, S-type, and U-type. There are two more instruction formats, B-type and J-type, for immediate formatting. The RISC-V instructions are aligned to have the opcode in the first 7-bits and destination register in the next 5-bits, unless it is a S-type instruction (store).



The ADD and ADDI instructions will add two registers or one register and a sign-extended 12-bit immediate. Their formattings are R-type and I-type, respectively. The SLTI/SLTIU instruction will return a value of 1 if rs1 is less than the sign-extended immediate or return a 0 otherwise. SLTIU will do the same comparison, but treat the immediates as unsigned values. AND and ANDI instructions will execute a logical AND on two registers or on a register and an immediate. OR and ORI instructions will do the same with a logical OR and XOR and XORI will do the same with a logical XOR operation. LUI and AUIPC are the two U-type instructions. LUI will place an immediate value of 20 bits filled with zeros in the lower 12 bits to the destination register. AUIPC does the same operation on the immediate value, but adds the immediate as an offset value to the Program Counter. The SUB function will perform a subtraction on two registers (RS2-RS1). SLT/SLTU are the register counterparts of SLTI/SLTIU and will do a less than comparison on two registers and write 1 to rd if true and 0 if false. SLL, SRL, and SRA are all shift instructions that shift the lower 5 bits of RS2 by the value held in RS1. SLL performs a logical left shift, SRL performs a logical right shift, and SRA performs an arithmetic right shift. JAL and JALR are jump instructions. JAL has a J-type formatting and adds a signed offset value in multiples of 2 bytes to the Program Counter. JALR has an I-type formatting and adds a signed 12-bit immediate value to RS1 to the Program Counter. The new address is stored into the destination register. BEQ, BNE, BLT/BLTU, and BGE/BGEU are all branch instructions that compare two registers and take the branch if the comparison is true. The comparisons are if they are equal, not equal, less than/less than unsigned, and greater than/greater than unsigned, respectively. Our implementation has Store and Load instructions for 32-bit values. LD is encoded as an I-type and stores a 32-bit value from memory into the destination register. ST is encoded as an S-type and stores a 32-bit value in RS2 to memory.

b. Block Diagram (riscV.sv)



c. Written Description of all .sv modules

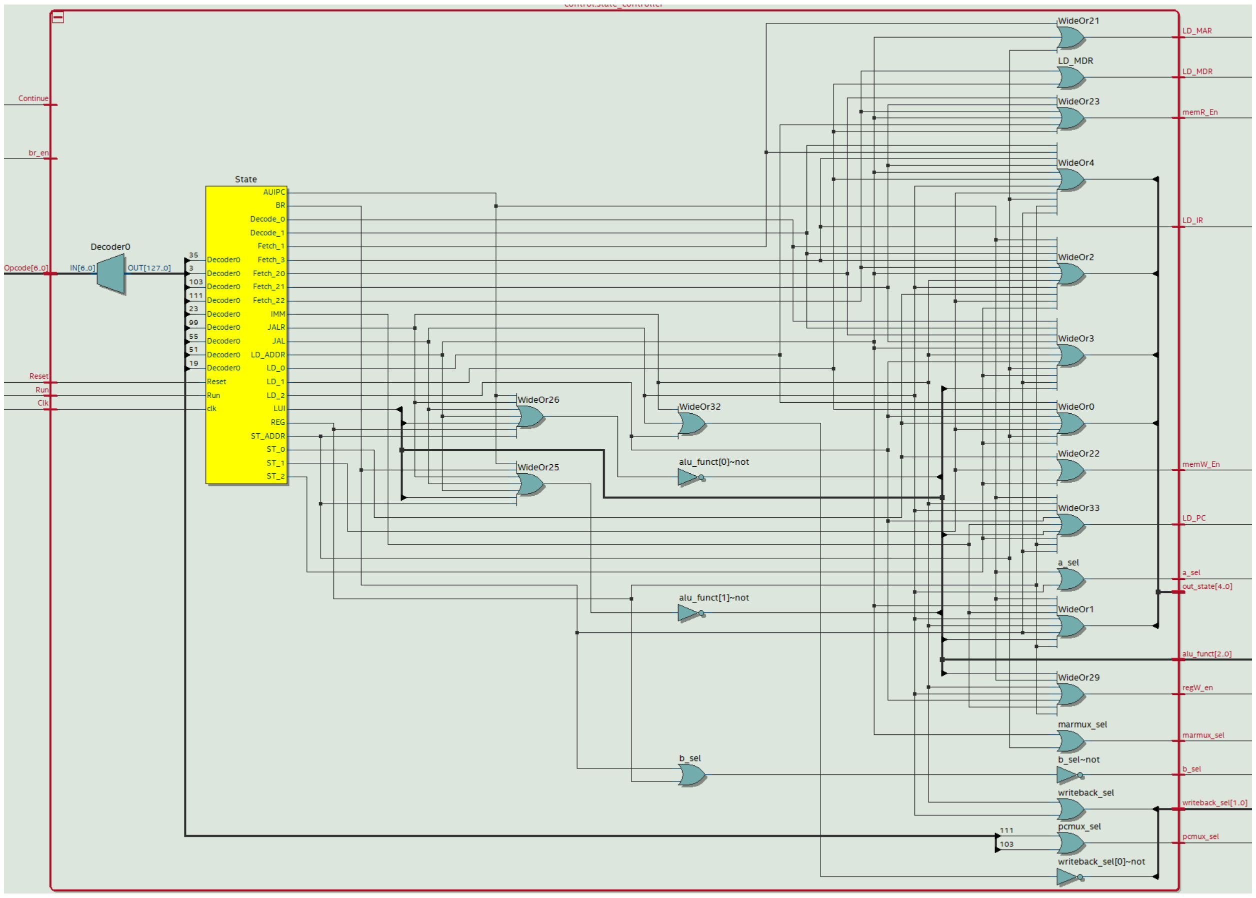
Module: riscV.sv
Inputs: [9:0] SW, Clk, Reset, Run, Continue, [31:0] Data_from_SRAM
Outputs: OE, WE, [9:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [31:0] ADDR, [31:0] Data_to_SRAM
Description: This module sets up the connections to the datapath, mem2IO, Hexdrivers, alu_control, control_transfer, and control.
Purpose: This module creates connections amongst the datapath, mem2IO, Hexdrivers, and the control modules. This module is like a high-level module making sure all the lower level modules are connected to the datapath.

Module: control.sv
Inputs: Clk, Reset, Run, Continue, [6:0] Opcode

Outputs: br_en, LD_IR, LD_MDR, LD_MAR, marmux_sel, [2:0] alu_funct, a_sel, b_sel, regW_en, [1:0] writeback_sel, memW_En, memR_En, pcmux_sel, LD_PC, [4:0] out_state

Description: This module instantiates the state machine needed for our RISC-V implementation. The state machine follows a fetch-decode-execute cycle that branches to different states depending on the opcode. The states for the instructions are split by I-type, R-type, upper-immediate, branch, jump, load, and store functions. Memory read and write operations, like fetch, have additional states for memory retrieval time. All the states return to the fetch state.

Purpose: This module is necessary to set the control signals for each state and each instruction. It creates the state machine needed for RISC-V implementation.

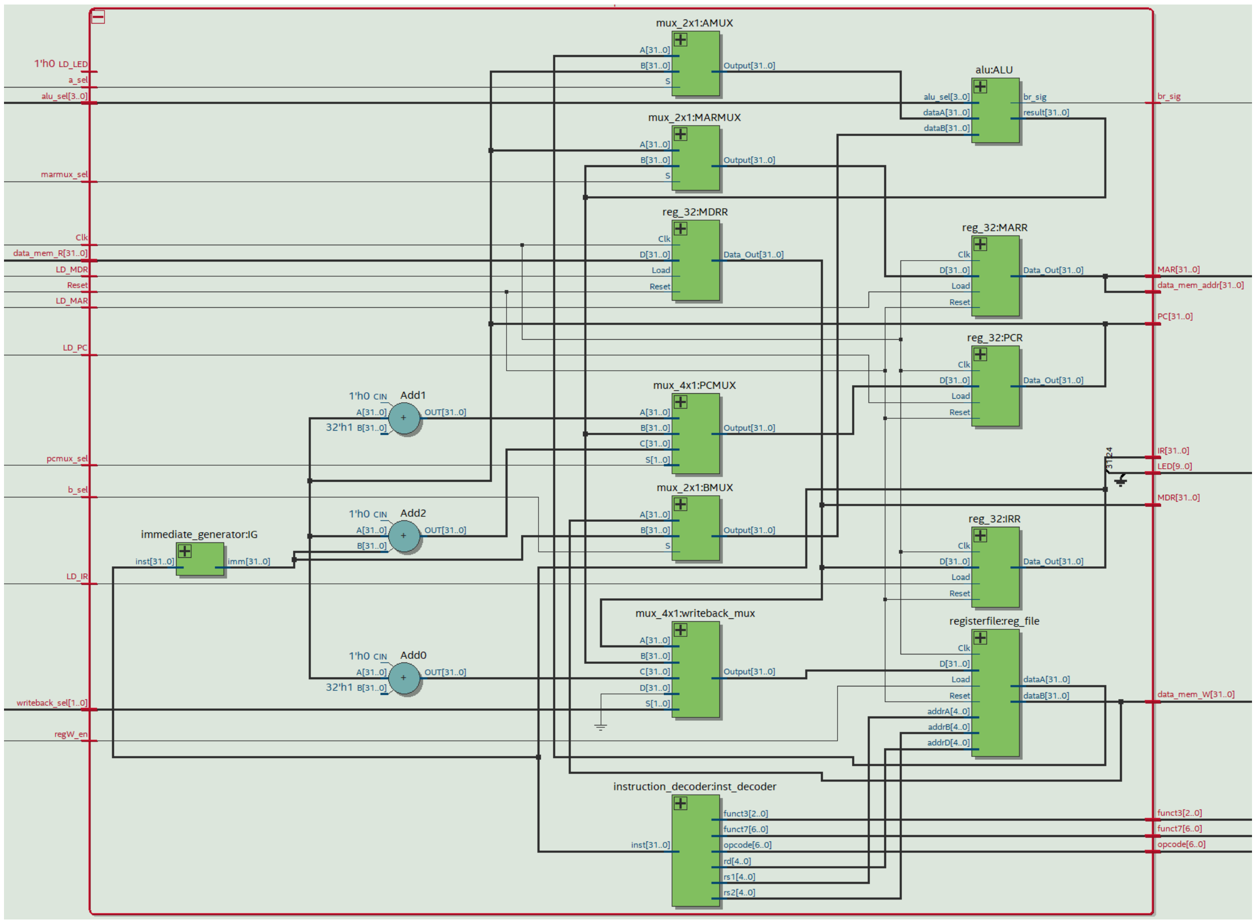


Module: datapath.sv

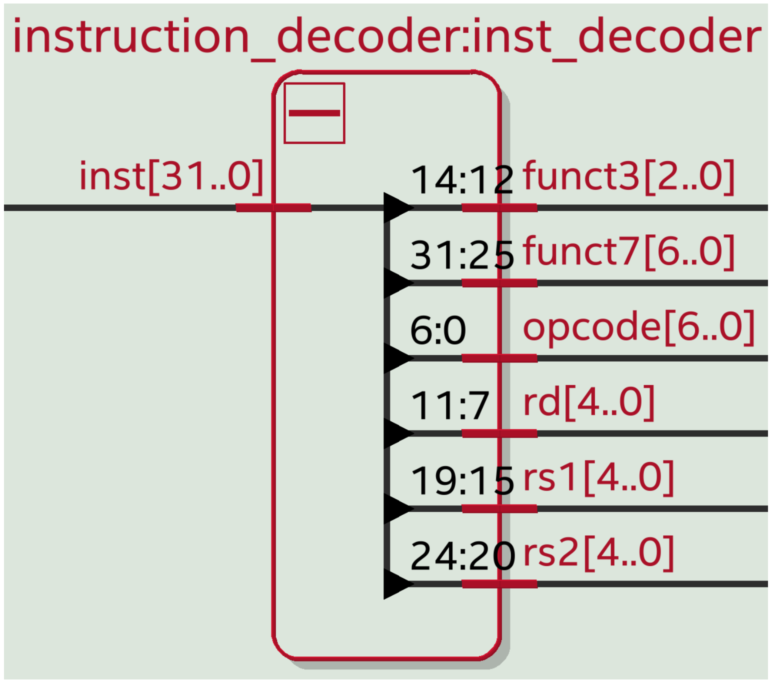
Inputs: Clk, Reset, LD_MAR, LD_MDR, LD_IR, LD_PC, LD_LED, a_sel, b_sel, marmux_sel, pcmux_sel, [1:0] writeback_sel, [3:0] alu_sel, [31:0] data_mem_R, [6:0]

Outputs: opcode, [2:0] funct3, [6:0] funct7, br_sig, [31:0]: MAR, MDR, IR, PC, data_mem_W, data_mem_addr, [9:0] LED

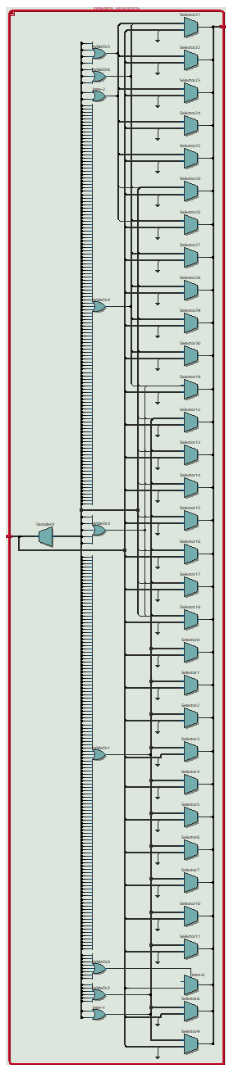
Description: This module sets up the muxes, registers, alu, immediate generator, and instruction decoder used in RISC-V. It also has a lot of local variables for connections to the riscV module..
Purpose: This module implements the data path of all the logic components and registers in the RISC-V. It uses the control signals set in the control module to set the registers and select values for the logic components to the correct values and load the corresponding bits into the logic in each state.



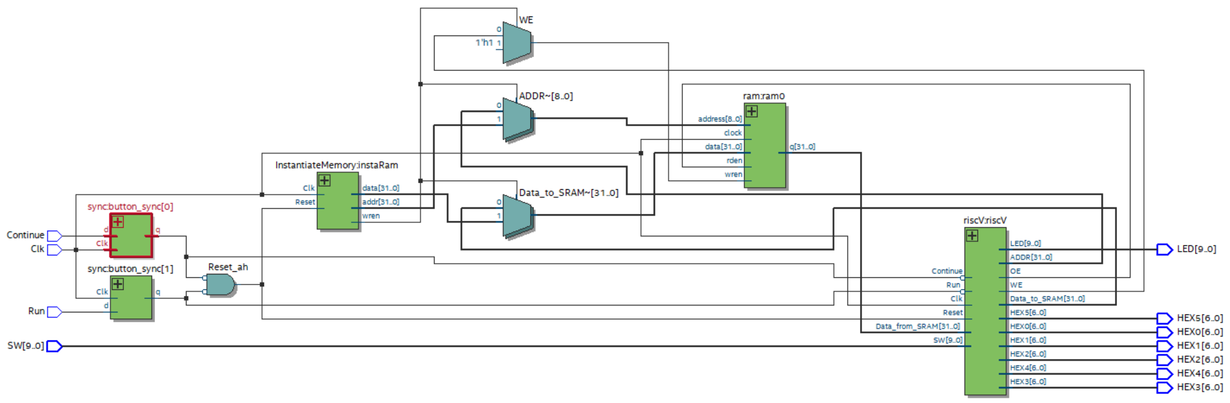
Module: instruction_decoder.sv
Inputs: [31:0] inst
Outputs: [6:0] opcode, [2:0] funct3, [6:0] funct7, [4:0]: rd, rs1, rs2
Description: Splits the 32-bit instruction and labels them. The instruction is divided by the opcode, destination register, function 3, function 7, and register 1 & 2.
Purpose: This module allows easier and cleaner implementation when specific parts of the instruction are needed for operation execution and calculation.



Module: immediate_generator.sv
Input: [31:0] inst
Output: [31:0] imm
Description: The immediate is sorted and formatted in a certain order depending on the opcode.
Purpose: In RISC-V, the immediates are formatted differently depending on the instruction type and this is important mainly for sign extension purposes.

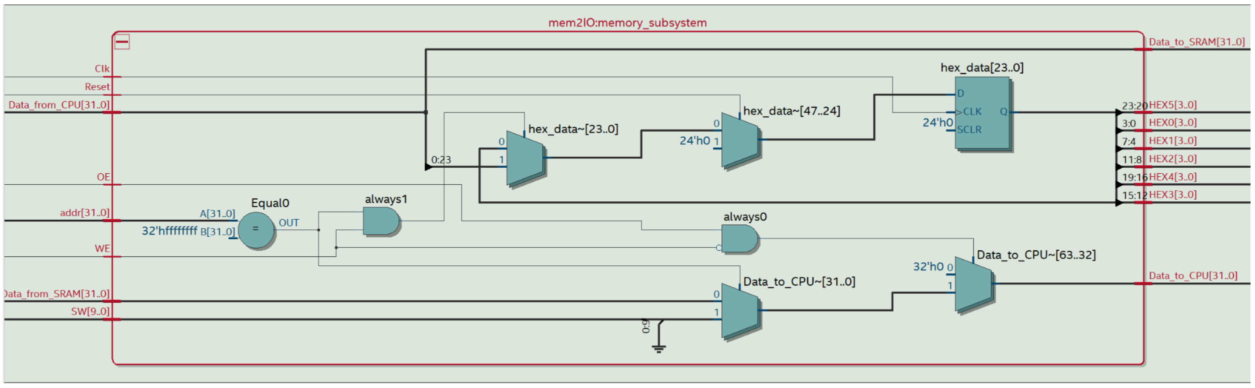


Module: riscV_sramtop.sv
Inputs: [9:0] SW, Clk, Run, Continue
Outputs: [9:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5
Description: This is the top level module used for testing using the on-chip memory and for synthesis.
Purpose: We need this module for FPGA implementation and instantiate the ram for on-chip memory.

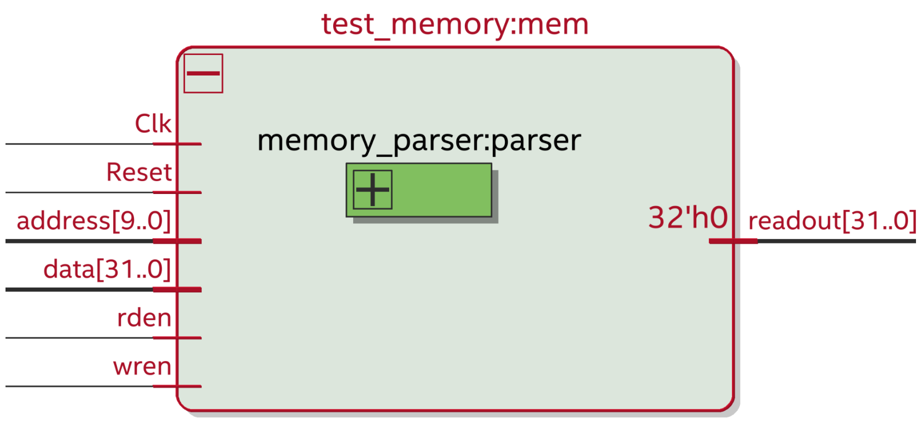


Module: riscv_testtop.sv
Inputs: [9:0] SW, Clk, Run, Continue
Outputs: [9:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5
Description: This is the top level module used for testing using the memory created in test_memory.
Purpose: This module allows testing and simulation using ModelSim. A test ram is instantiated for a substitution for physical memory on the FPGA.

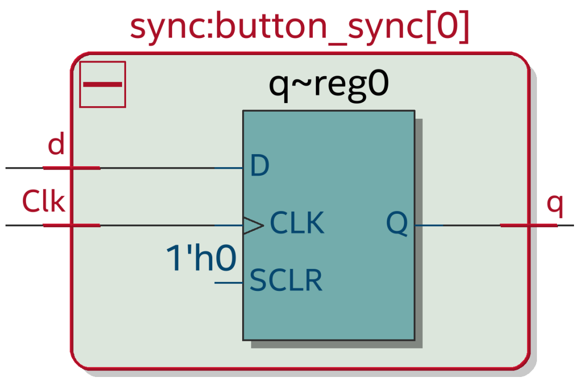
Module: mem2IO.sv
Inputs: Clk, Reset, [31:0] addr, OE, WE, [9:0] Switches, [31:0] Data_from_CPU, [31:0] Data_from_SRAM
Outputs: [31:0] Data_to_CPU, [31:0] Data_to_SRAM, [3:0] HEX0, [3:0] HEX1, [3:0] HEX2, [3:0] HEX3, [3:0] HEX4, [3:0] HEX5
Description: This module will read from the switches when the address is xFFFF and will read from SDRAM otherwise. It will also set the HEX values for the output.
Purpose: This module allows the FPGA to read and write to SDRAM and read from the switches.



Module: test_memory.sv
Inputs: Reset, Clk, [31:0] data, [9:0] address, rden, wren
Outputs: [31:0] readout
Description: This instantiates memory that is used to be a substitution for the on-Chip memory.
Purpose: This module is needed so that Modelsim can be used for simulation and is a replacement for the on-Chip memory.

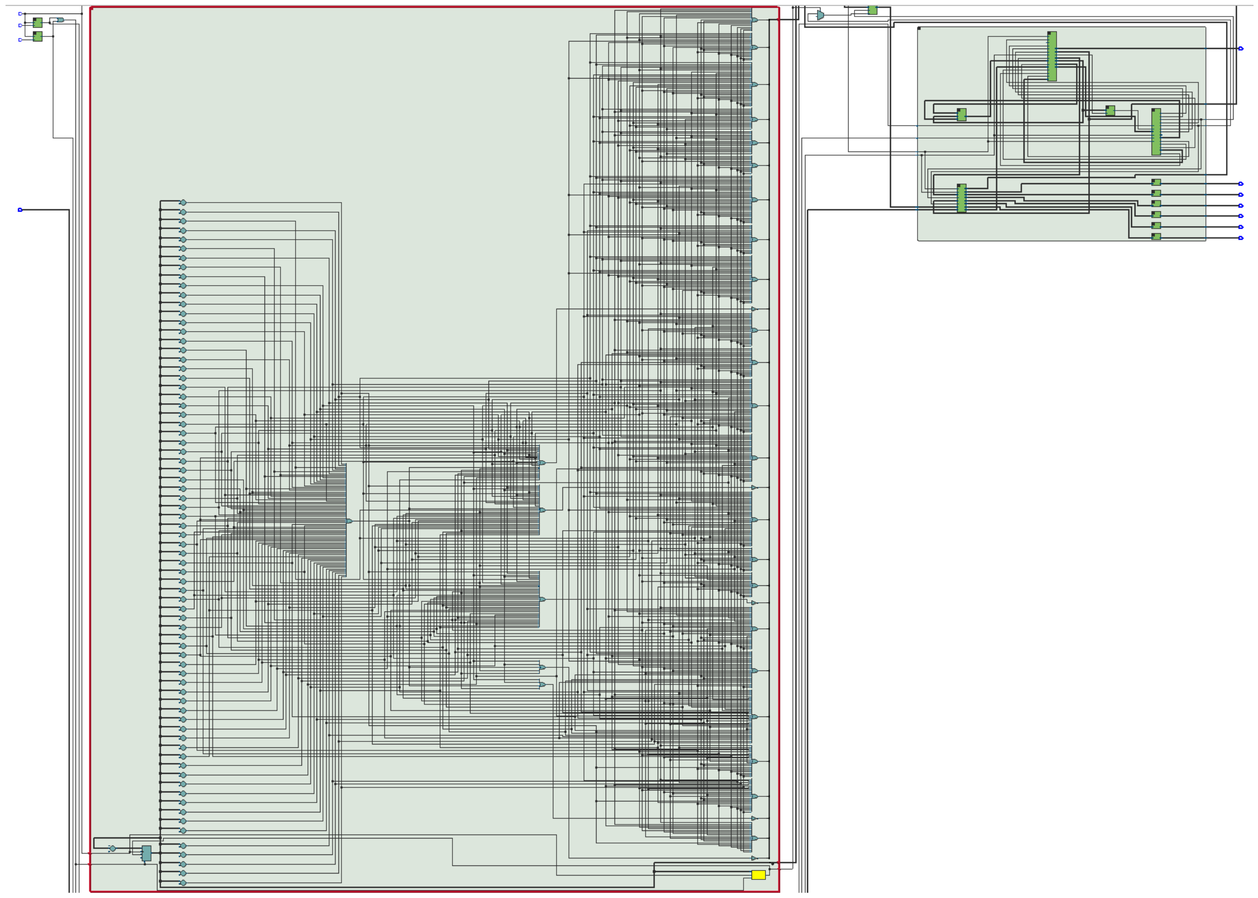


Module: synchronizers.sv
Inputs: Clk, d
Outputs: q
Description: This module synchronizes the signals with the positive edge of the clock
Purpose: This module is needed to make sure signals are all received at the correct times and that asynchronous signals are sent to the FPGA.



Module: Instantiateram.sv
Inputs: Reset, Clk
Outputs: [31:0] addr, wren, [31:0] data

Description: This module instantiates the on-Chip memory and contains all the instructions for the tests we need to run.
Purpose: This instantiates the memory needed for the processor and allows for us to test our instructions and implementation on the FPGA.

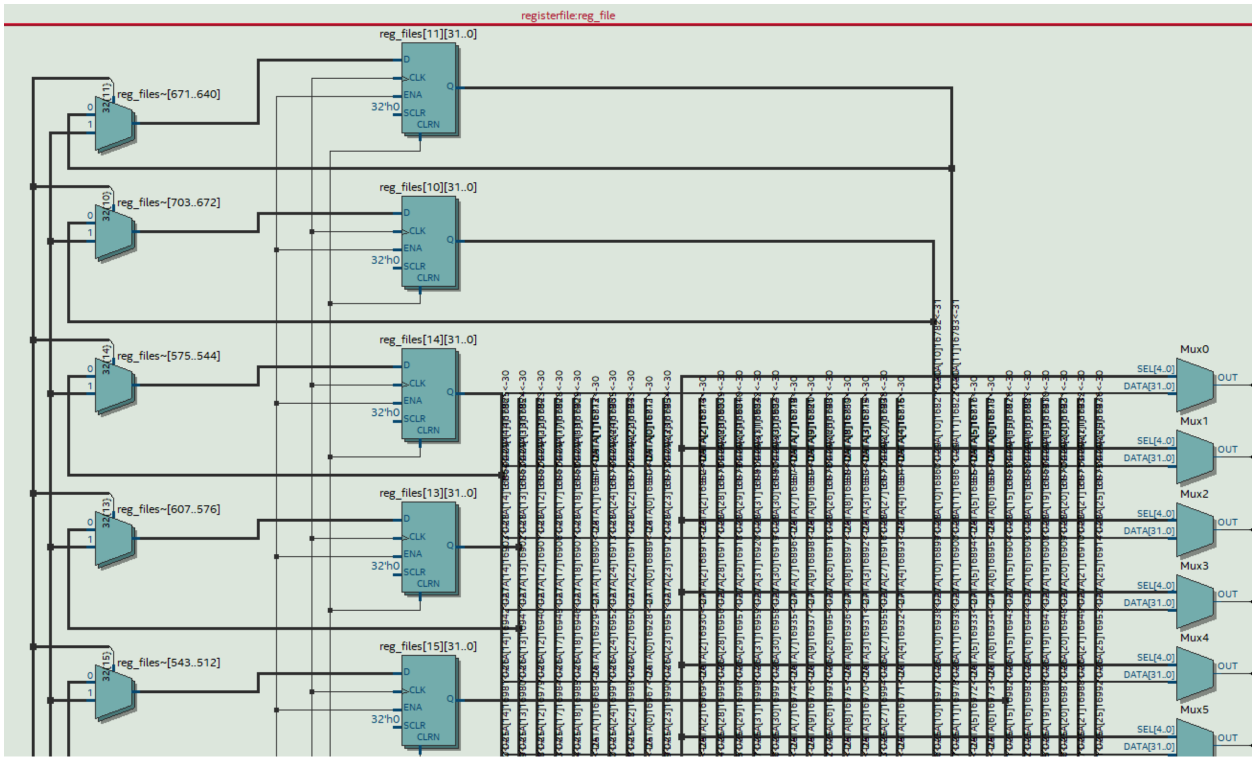


Module: memory_contents.sv
Description: This module creates an array for 255 instructions of 32-bit size.
Purpose: This implements instructions for testing into the memory for simulation. This is the simulation version of Instantiateram.

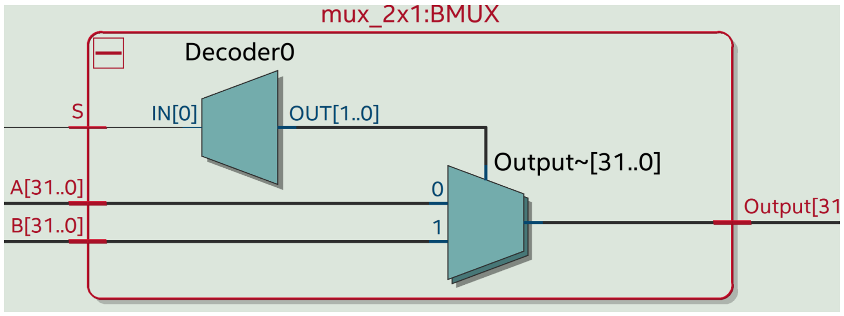
Module: RISCV_Opcode_Constants.sv
Description: Definitions of opcode and function encoded by bits.
Purpose: Allows us to write test code into the memory easily.

Module: registerfile.sv
Inputs: Clk, Reset, Load, [31:0] D, [4:0] addrD, [4:0] addrA, [4:0] addrB
Outputs: [31:0] dataA, [31:0] dataB
Description: This module acts as the register file and will load and reset 32 registers.

Purpose: This module will select and write or read from a register given the inputs. It also instantiates the 32 registers that are used in RISC-V.

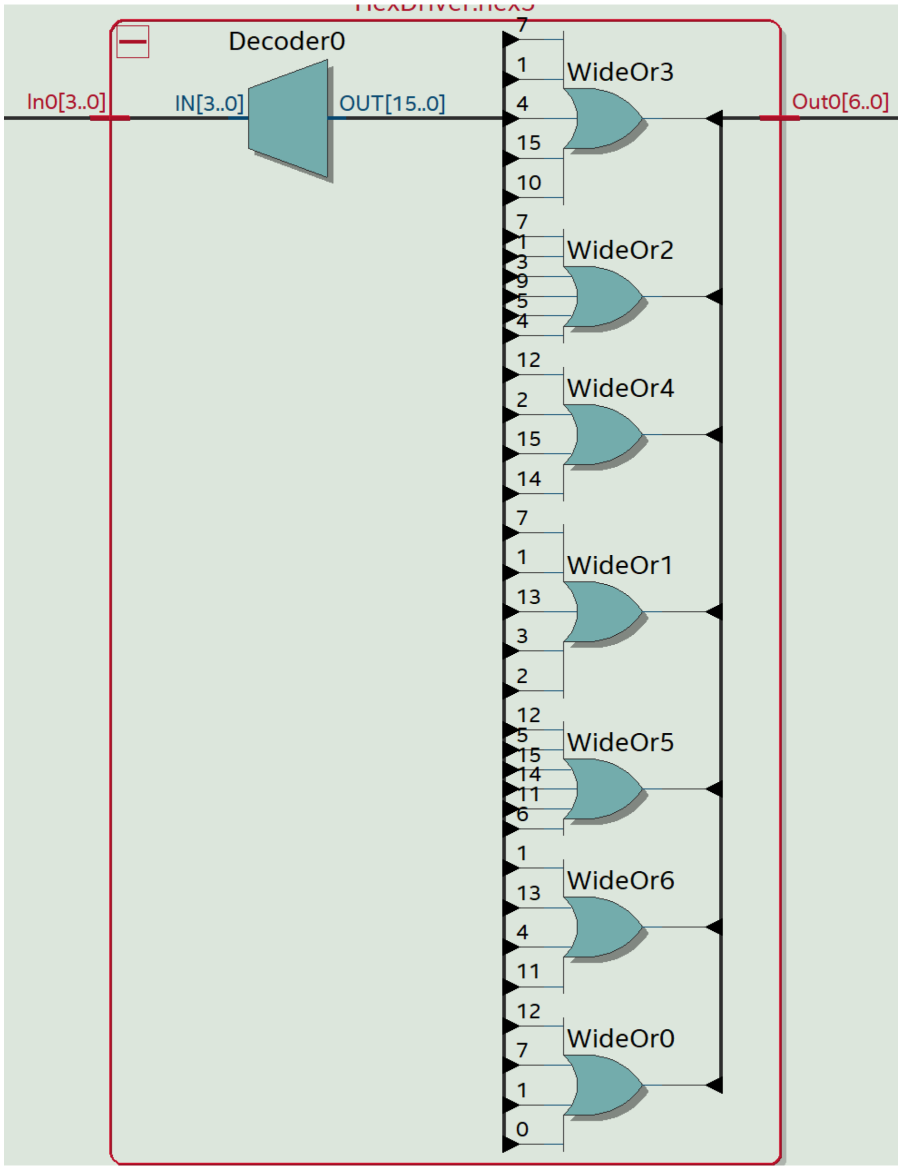


Module: mux_2x1.sv
Inputs: S, [31:0] A, [31:0] B
Outputs: [31:0] Output
Description: Instantiates a mux with two inputs and one output.
Purpose: This module is needed for mux inputs for the ALU and a mux for MAR.



Module: mux_4x1.sv
Inputs: S, [31:0] A, [31:0] B, [31:0] C, [31:0] D
Outputs: [31:0] Output
Description: Instantiates a mux with four inputs and one output.
Purpose: This module is needed for writeback signal select and to select how to update the PC.

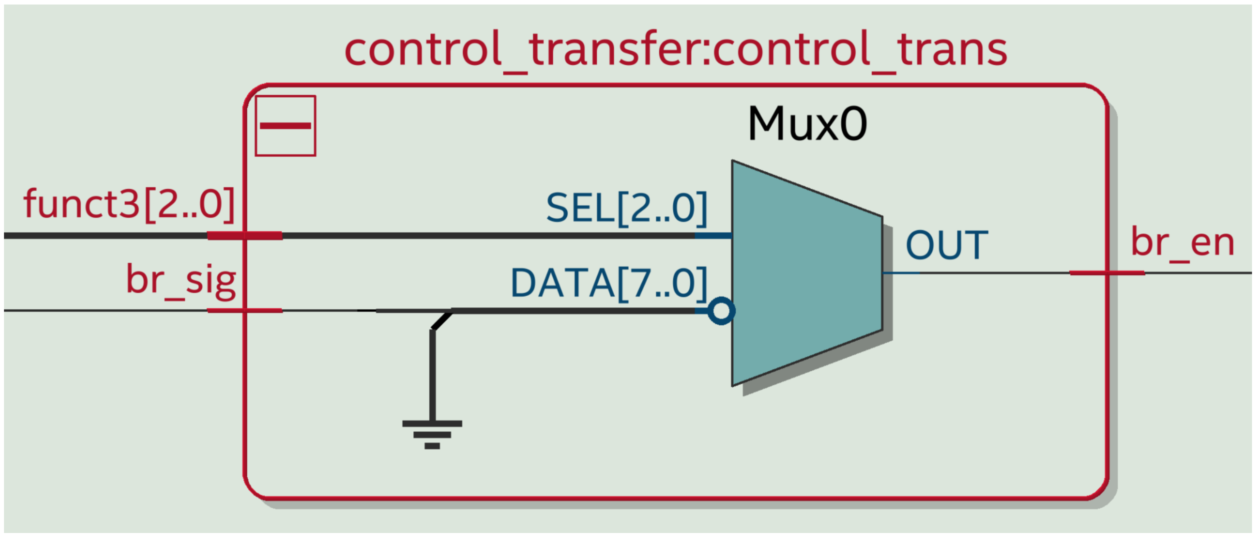
Module: HexDriver.sv
Input: [3:0] In0
Output: [6:0] Out0
Description: This module has the hardcoded conversions of the 4-bit binary numbers into hexadecimal values to be displayed.
Purpose: This module is utilized to display the right hex values on the FPGA.



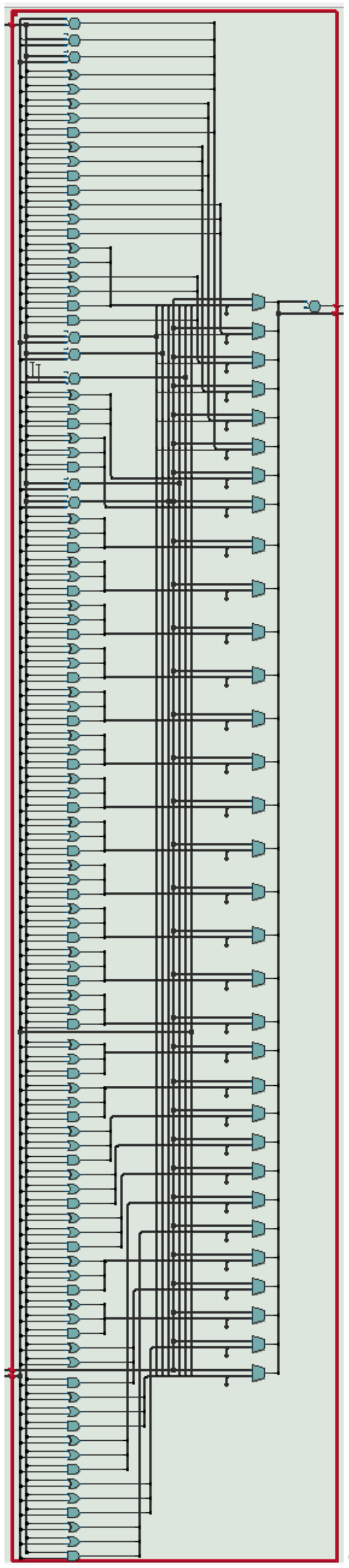
Module: definitions.sv
Description: It defines all the bit codes for operations, functions, and selects for control.
Purpose: The definitions make the code clearer and understandable.

Module: control_transfer.sv

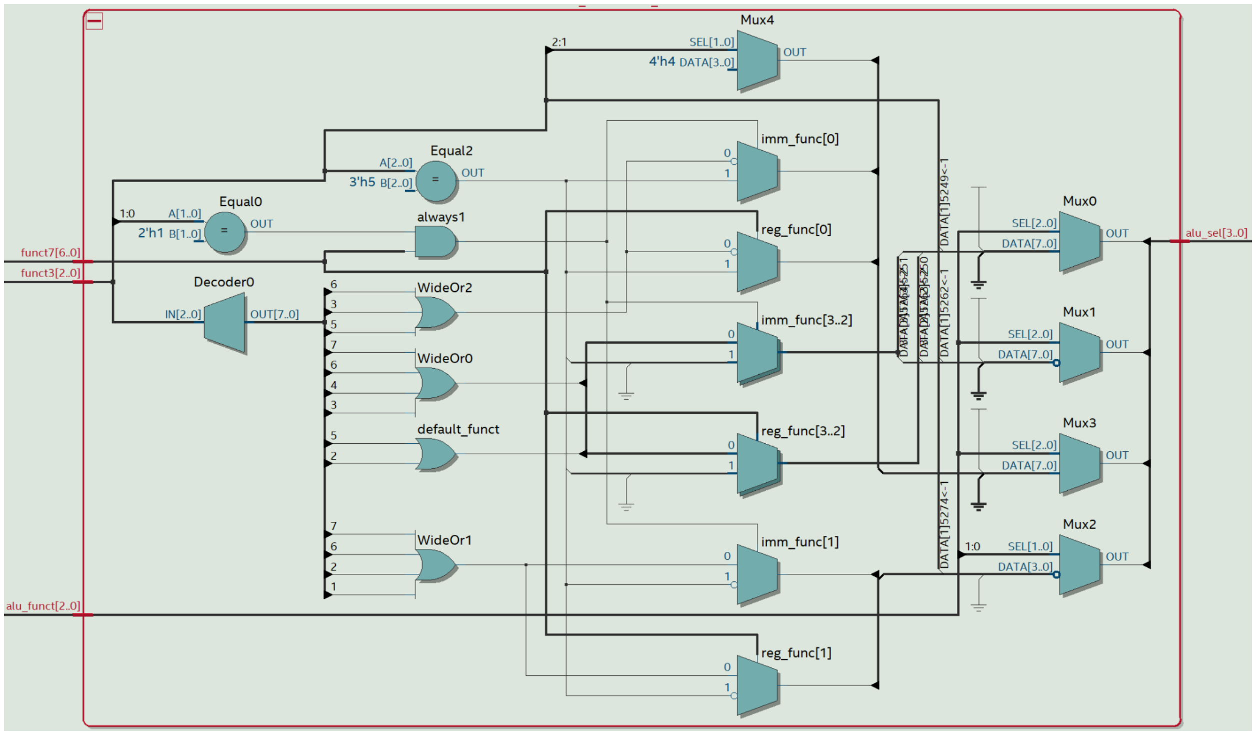
Inputs: br_sig, [2:0] funct3
Output: br_en
Description: It sets the branch enable signal for the control depending on the ALU output.
Purpose: This module makes sure that the branch enable is being set properly and the execution of the instructions branches correctly.



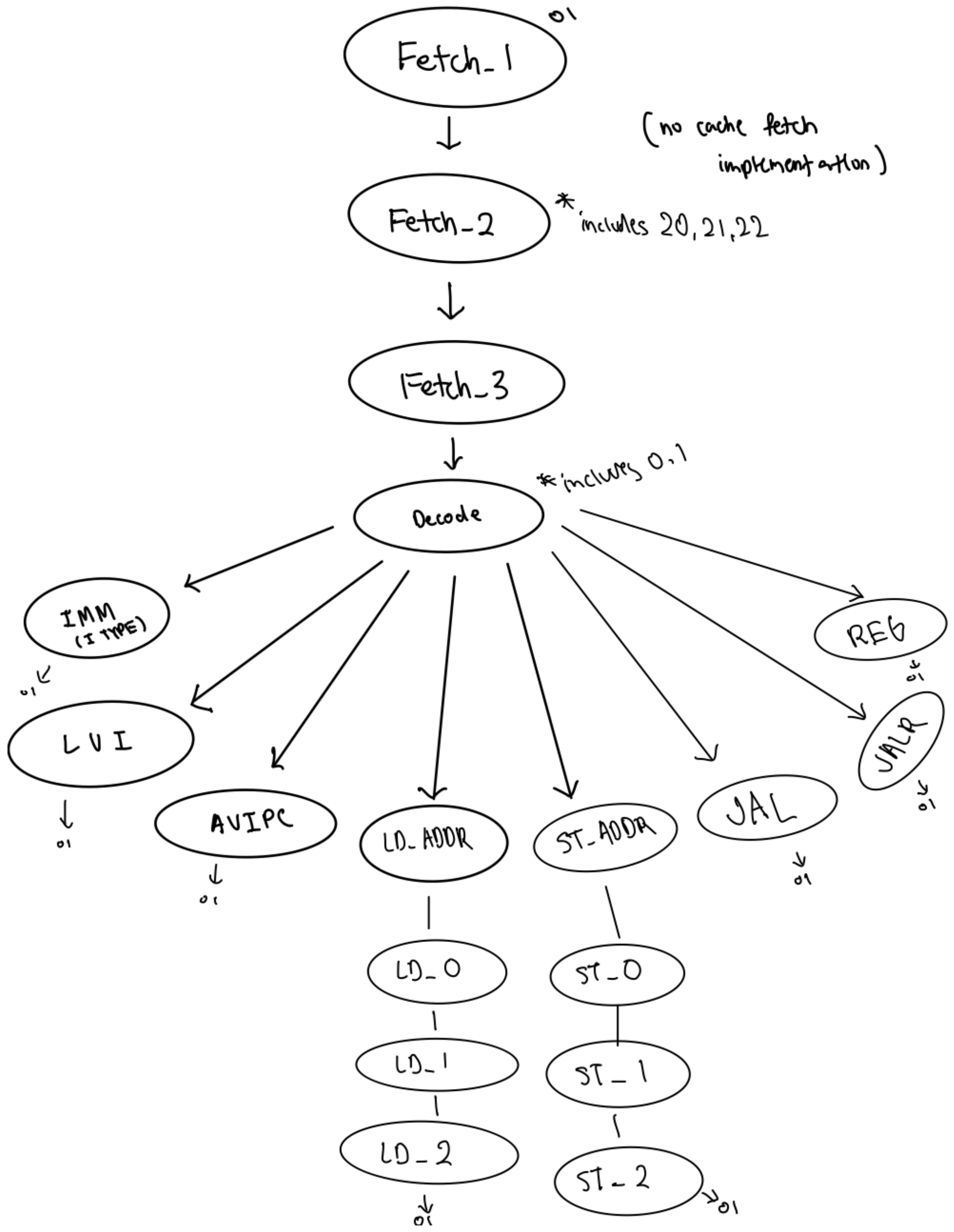
Module: alu.sv
Inputs: [3:0] alu_sel, [31:0]: dataA, dataB
Outputs: [31:0] result, br_sig
Description: This is the arithmetic logic unit for our RISC-V implementation. It has all the logical operations needed to execute the RV32I instructions. The output defaults zero and outputs the result for the branch signal.
Purpose: This module is needed for any logical operation or comparison for RISC-V instruction set.


(alu)

Module: alu_control.sv
Inputs: [2:0]: alu_funct, funct3, [6:0] funct7
Output: [3:0] alu_sel
Description: This module sets the select for the alu depending on the function set by the control unit.
Purpose: This module decodes the bits in function to set the right ALU select value for to choose the right logical operation and comparison executed in the ALU.



d. Control/ISDU state diagram

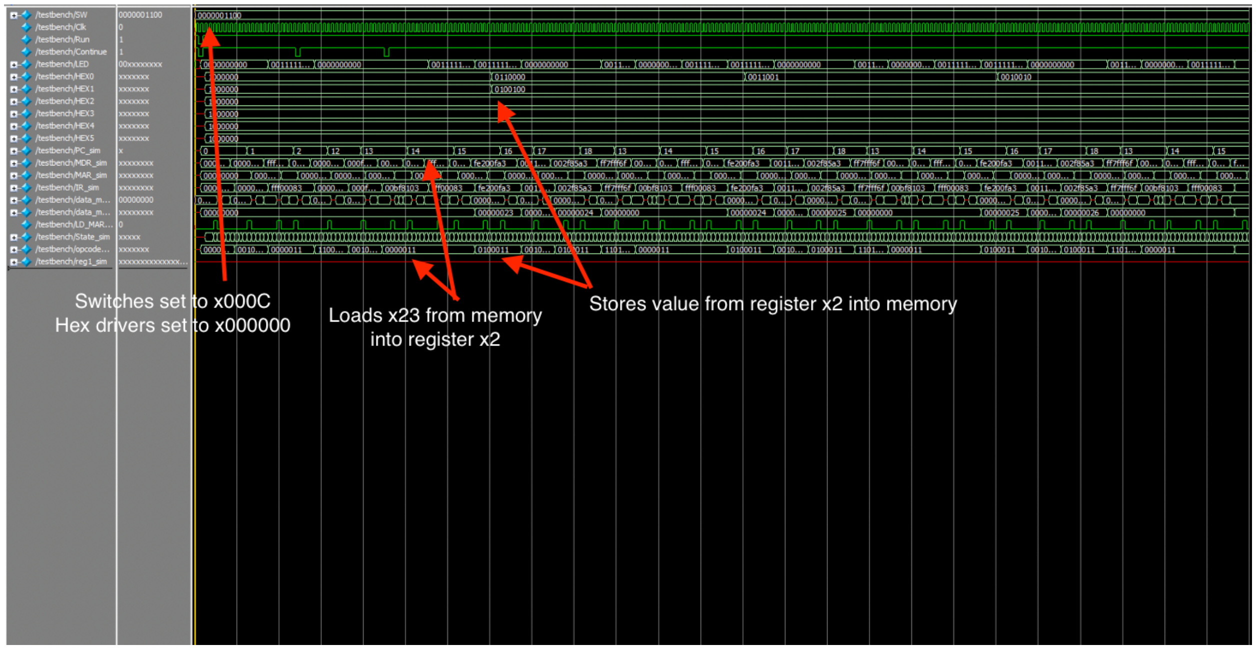


2. Simulations of Test Cases

a. Shift test

Description:

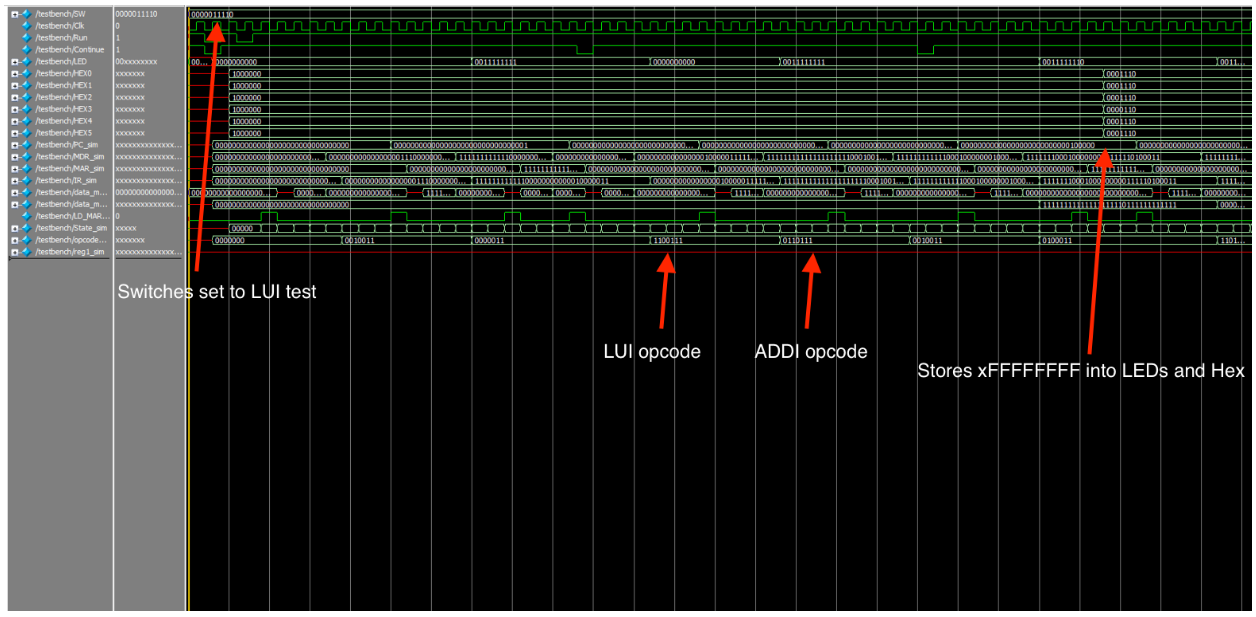
Instructions: LD, ADDI, SUB, SLTI, SLLI, SLTUI, OR, ST, JAL



b. LUI test

Description: test LUI and should output xFFFFFF in hex

Instructions: LUI, ORI, ST, JAL



3. Design Resources and Statistics Table

LUT	**3858**
DSP	**0**
MEMORY	**0**
FLIP-FLOP	**16,384**
Frequency	**59.66 MHz**
Static Power	**89.94 mW**
Dynamic Power	**0 mW**
Total Power	**98.7 mW**

Conclusion

We were able to adapt the structure from Lab 5 to create our RISC-V implementation on the FPGA. We were mostly successful in creating the RV32I instruction set architecture missing a few instructions. We created general LD and ST functions for 32-bits words and were unable to implement the Fence instruction. Creating the instruction set was a challenge and we ran into plenty of bugs. To fix, clean, and test, we implemented a test memory to test with ModelSim and then later a separate entity top level module for physical testing on the FPGA and its hex displays. In the end, we were able to finish with a working Single Cycle Instruction processor. Implementing RISC-V allowed us to more firmly understand the design of a processor. We gained a firm grasp on the workings of a CPU, functions and design of memory, and input/output interface with the FPGA through this project. Overall, it was a difficult, but rewarding experience.