Multicycle UnPipelined RISC-V Processor

Tane Koh, Eddy Pan, Darian Jimenez

For this project, we implemented and tested our RISC-V Processor.

The processor takes 6 cycles to complete all instructions, giving us a throughput of 1 instruction to 6 cycles, and a latency of 6 cycles to complete 1 instruction.

High Level Design:

We architected our processor with the following modular design:

Register File module: register_file.sv

- We implemented a 32-register by 32-bit register file typically used in RISC-V processors. It takes a clock signal, write enable, 2 read addresses, destination register, and input data register as inputs, and outputs 2 register values. The read operations within the register file are combinational, while the write operations are clocked (occurs on the positive clock edge) and only occur when the write enable register is on. It's specifically configured so read-before-write operations are possible (reading old register values before writing new values).

ALU module: `alu.sv`

- We implemented a 32-bit ALU that takes in two inputs and an alu operation type input, and outputs the result of the operation as well as flags for zero, less than signed, and less than unsigned. The ALU is capable of doing the ADD, SUB, AND, OR, XOR, SLL, SRL, and SRA operations. The ALU is combinational so that the output changes immediately with the inputs. It's structured around a case statement so that operations depend on the alu operation type input.
- The ALU is used for program counter calculations, figuring out the next instruction memory address, for arithmetic operations in instructions, to calculate addresses for load/store instructions, and to evaluate branch conditions (using the flags).

Immediate Generator module: imm_gen.sv

- We implemented an immediate generator module responsible for sign-extending the immediate values from the instructions. This works as part of the decode state that prepares components for ALU operations. It takes the

instructions as input and outputs the sign-extended immediate value. The module is combinational so outputs change as soon as the new instruction is read. The generator works by reading the instruction to determine the opcode (instruction type), extracting the immediate bits from the instruction, and then extending them depending on the instruction type.

Memory module memory.sv

- This module was adapted from the iceBlinkPico memory module, we reused the Data Memory (DMEM) and Instruction Memory (IMEM) from the original implementation.

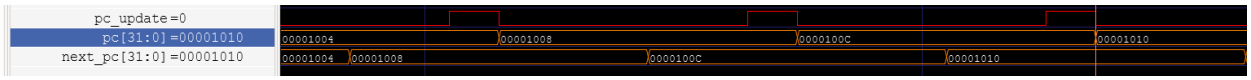
Top module top.sv

- The top module ties together all our modules. Initiates alu, register, memory and imm_gen modules
- The **Program Counter (PC)** is the same thing as the instruction memory address, responsible for determining the instruction that the processor follows. In this processor, the next iteration of the program counter is calculating within the states, depending on whether it is just advancing normally, or advancing to a specified value through a jump or branch instruction.
- We also implement an **ALU** input multiplexer within the top module to enable the ALU to be used for calculating the next PC value (using PC + 4), do ALU operations on register values (branch, r-type, etc.), or use immediate values for calculations (i-type, load, store instructions).
- We implemented a six-stage Finite State Machine with the following states: IDLE, FETCH, DECODE, EXECUTE, MEMORY, and WRITEBACK. The IDLE state is used as a reset state to initialize control signals, and leads to FETCH.
    - The **FETCH** state is responsible for reading the proper instruction from instruction memory, and then calculating the next address to read the next cycle's instruction. To do this, it uses the same ALU that is used for R-Type operations and instructions. It then leads to the DECODE state.
    - In the **DECODE** state, the processor decodes the instruction fields and prepares the immediate values (which are combinational). For branch and jump instructions, it also calculates the branch/jump target addresses. It also sends the values to the register values, making the output from the register values available for the next state, the EXECUTE state.
    - The **EXECUTE** state is where the bulk of the ALU operations or address calculations are completed, where the ALU op type is configured and the resulting bit is stored as the data memory address. This operation is

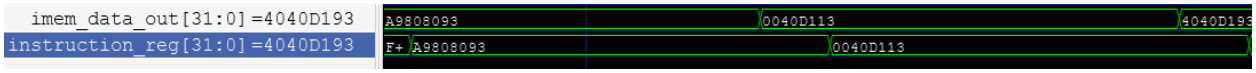
combinational as the ALU is combinational. This leads to the next state, the MEMORY state.
    - In the **MEMORY** state, during load, store, or branch instructions, the processor accesses the memory module, either to read/write values (load/store instructions), or to check address conditions for branch and jump instructions. This then leads to the final state, the WRITEBACK state.
    - The **WRITEBACK** state is where the processor writes results back to the register file, whether that's ALU operation results, memory data, or address results. Additionally, this triggers the program counter update, so during the next FETCH state, it is reading from the proper instruction memory address.
- Manages RGB LED based on the opcode of the current instruction

How does our simulation demonstrate proper processor operation and RISC-V implementation?

Progressing through the different FSM states.



Progression of PC iteration. For non branch/jump instructions, the program counter advances 4 more each time.



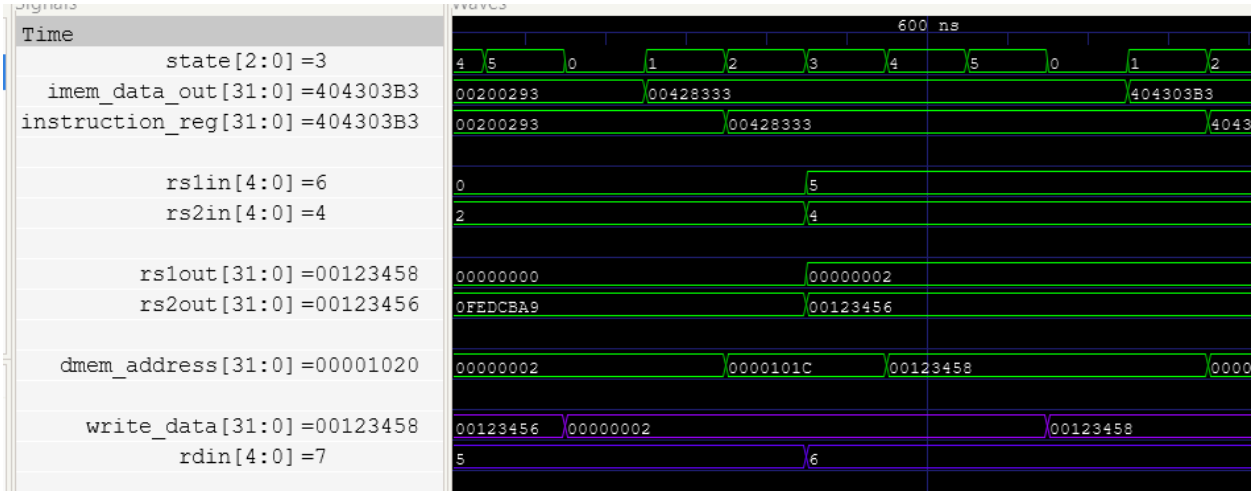
The instruction register is read from the instruction memory (output) with a 1-state delay for it to read it from fetch to decode state.

Instruction Types:

R-Type Instructions

For R-type instructions, the processor operates normally in the fetch state, and during the decode state, selects values from two registers based on the portion of the instruction register that determines it. The funct3 and funct7 values and the eventual destination register are isolated from the instruction register. In the execute state, the processor sets the multiplexer so the two ALU input values are both the

two registers assigned. Additionally, depending on the funct3 value, different ALU operator types are assigned for the ALU operation. During the memory state, all that happens is the pc_update flag is updated so pc will update in the next state. Finally, during the writeback state, the result of the ALU operation (or the resulting flags) are written to the destination register.

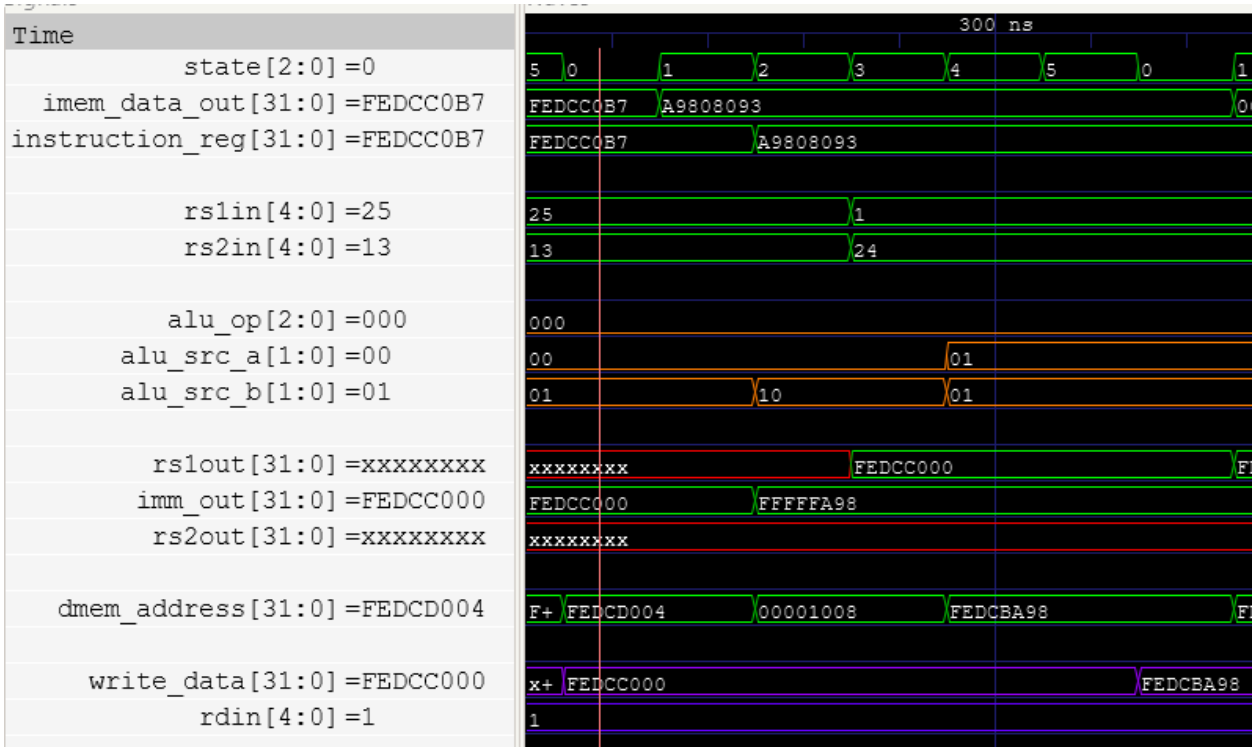


As we can see in the simulation, for the instruction add x6, x5, x4, the instruction register (00428333) is declared in the first state, the registers to operate on (x5 and x4) and to write on (x6) are declared in the second, and instantaneously, the values of the registers are found (combinational structure), and after the third state, the result of the ALU operation is found, and after the writeback state, written to the correct register.

In summary, R-Type Instructions:

- **ADD**: Adds the values in rs1 and rs2 and stores the result in rd.
- **SUB**: Subtracts the value in rs2 from rs1 and stores the result in rd.
- **SLT**: Sets rd to 1 if rs1 is less than rs2 (signed), otherwise sets rd to 0.
- **SLTU**: Sets rd to 1 if rs1 is less than rs2 (unsigned), otherwise sets rd to 0.
- **AND**: Performs a bitwise AND between rs1 and rs2, storing the result in rd.
- **OR**: Performs a bitwise OR between rs1 and rs2, storing the result in rd.
- **XOR**: Performs a bitwise XOR between rs1 and rs2, storing the result in rd.
- **SLL**: Performs a logical left shift on the value in rs1 by the amount specified in the lower 5 bits of rs2, storing the result in rd.
- **SRL**: Performs a logical right shift on rs1 by the amount in rs2.
- **SRA**: Performs an arithmetic right shift on rs1 by the amount in rs2

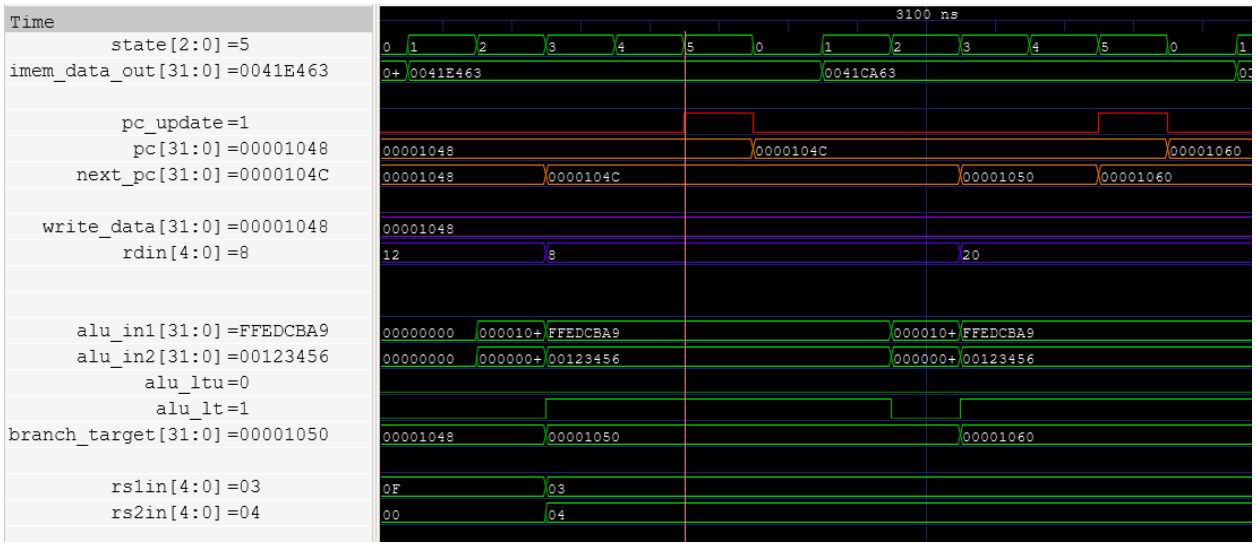
I-Type Instructions: Immediate and load



It operates the same way during the fetch and decode stages, although there is no rs2 value to assert (since I-type instructions use the immediate values). During the execute state, the processor sets the ALU source muxes so that the ALU will be pulling from RS1out (determined by alu_src_a) and the immediate generator (determined by alu_src_b). Then, the ALU performs the ADD operation, and the result given in dmem_address is written to the register specified in the decode stage. For the instruction addi x1, x1, 0xA98, the destination register (x1) and first source register (x1) are given, as well as the immediate value (xA98), so a simple add operation is performed, giving a new value to be written to the x1 register.

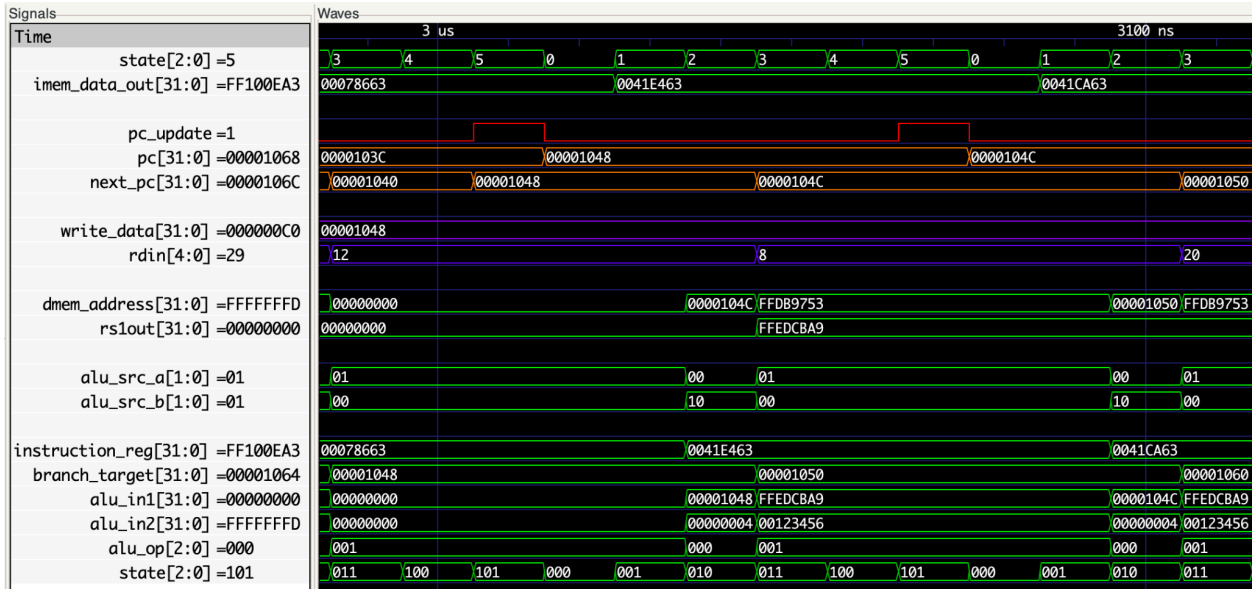
B-Type Instructions: Conditional branch

The fetch stage proceeds as normal for the branch instructions. During the decode stage, the processor takes the registers to compare and sets the mux for the ALU to be set to compare. Additionally, the branch target for the operation is set using the immediate values. Within the execute state, the ALU operation will complete, setting the flags for zero, less than, and less than unsigned comparisons. During the memory state, the processor operates through case statements to see if it should move to the branch target, or just advance to the next instruction. There is no writeback involved in the branch instructions.



For the two instructions bltu x3, x4, 8 and blt x3, x4, 20, we can see the progression through the states for this. It is comparing the x3 and x4 registers, as seen on the bottom, after the decode state, and the alu inputs reflect the values of those registers. The less than unsigned flag does not get raised for the first instruction, and so pc just goes to the next instruction (pc+4) instead of the calculated branch target. For the second instruction, the less than flag does get raised, and so the next pc is calculated to the branch target, and then the processor jumps to that instruction.

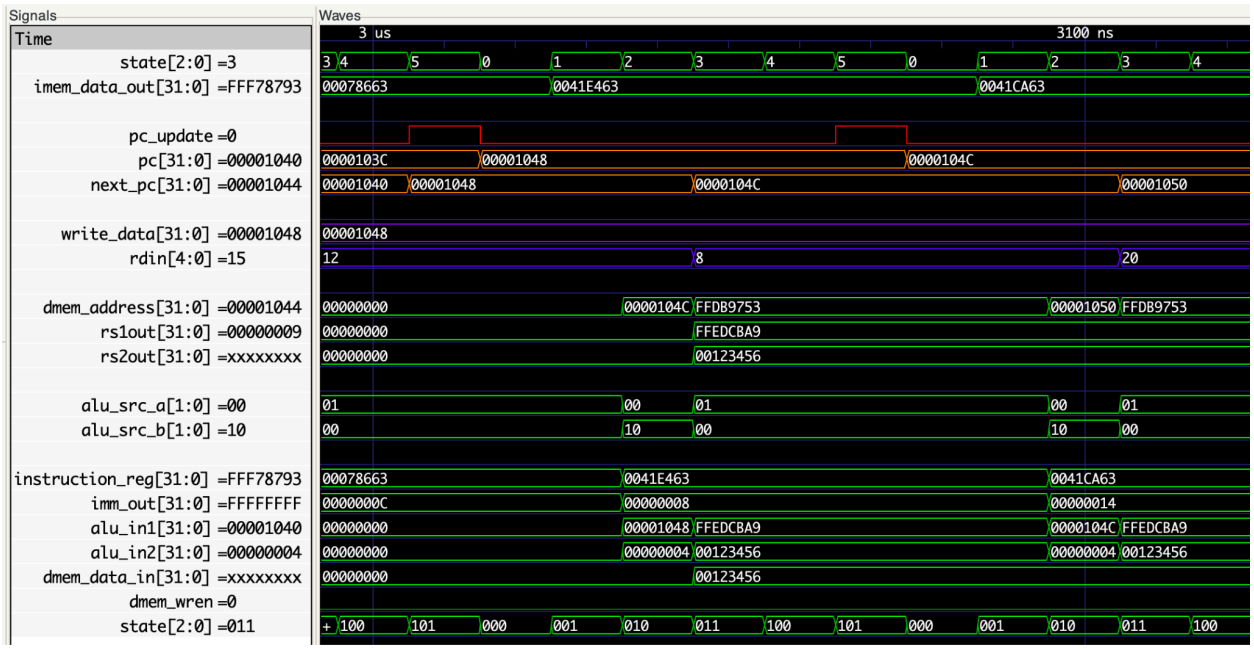
J-Type Instructions: Unconditional jump



For JAL the decode stage computes the jump target using the J-type immediate and computes PC+4 as the return address. In the memory stage, it sets next_pc to the jump target, and in writeback, PC+4 is written to rd.

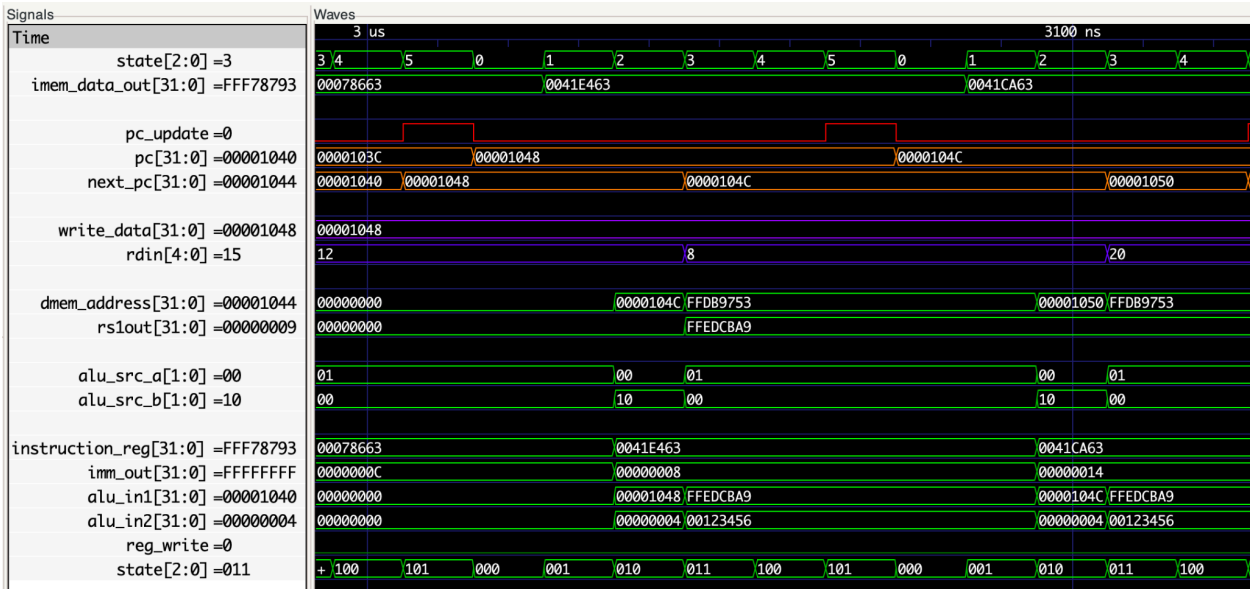
For JALR, execute computes the rs1 plus imm as the jump target. The stored PC+4 is then written back to rd. No data memory access happens here.

S-Type Instructions: Store



Here we extract rs1, rs2, and store the immediate value during the decode state. Execute calculates the effective address as rs1 plus imm. In the memory stage, dmem_wren is asserted and rs2out is written to memory at the computed address. Write back only affects the PC (stores don't write to the register file).

U-Type Instructions: Long immediate

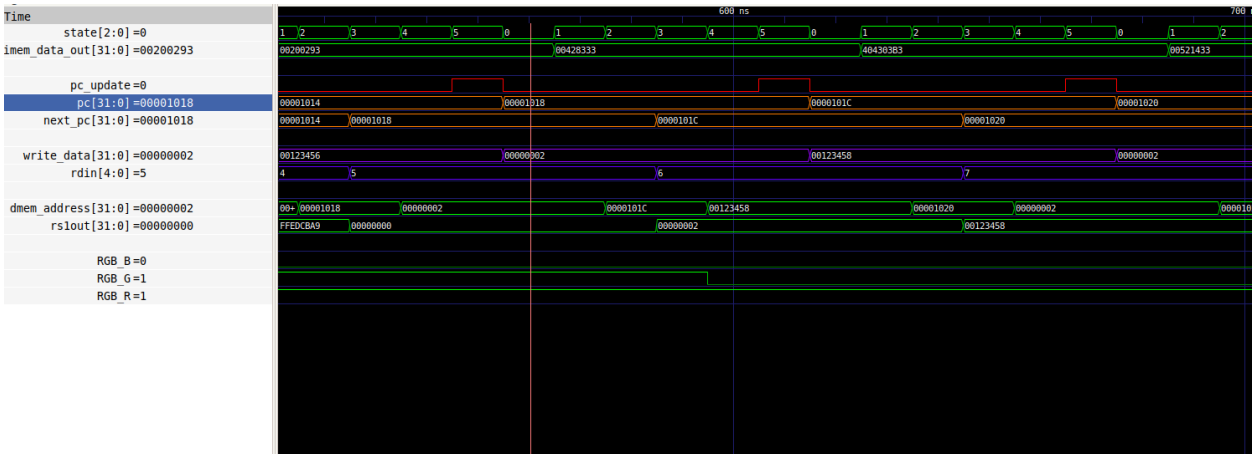


For U-Type instructions, we use a 20 bit upper intermediate and shift it left by 12 bits. LUI writes this value to the destination register in the writeback stage, and AUIPC adds the intermediate to the current PC in the execute state and writes the result during the writeback state.

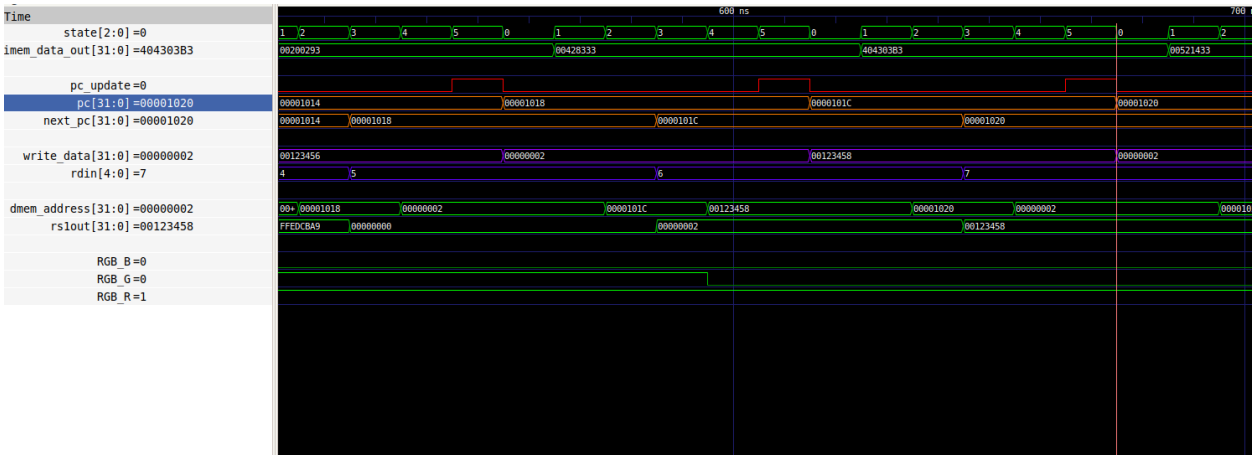
LED Behavior:

LED behavior is implemented in the memory module and is based on the memory-mapped LED register at 0xFFFFFFFC. The memory module takes in the opcode of the current instruction, and enters an always_ff block on the positive edge of the clock to write the output to red, blue, and green channels that update RGB_R, RGB_B, and RGB_G.

- R-type (ADD, SUB, AND, OR, etc.) → Red (255, 0, 0)
- I-type ALU (ADDI, ANDI, ORI, etc.) → Yellow (255, 255, 0)
- I-type Load (LB, LH, LW, etc.) → Cyan (0, 255, 255)
- S-type Store (SB, SH, SW) → Magenta (255, 0, 255)
- B-type Branch (BEQ, BNE, BLT, etc.) → Green (0, 255, 0)
- J-type JAL → Blue (0, 0, 255)
- I-type JALR → Purple (128, 0, 255)
- U-type LUI → Orange (255, 165, 0)
- U-type AUIPC → Dark Orange (255, 128, 0)
- DEFAULT→ Off (0, 0, 0)



I-Type transition into R-Type:
00200293 == addi x5, x0, 2 # pc = 0x14, x5 = 0x00000002
00428333 == add x6, x5, x4 # pc = 0x18, x6 = 0x00123458



Transitions from R and G high (Yellow) to just R high (Red).

Github Link:

https://github.com/darianjimenez/MP4