RISC-V Single Cycle CPU

Matthew Lund

November 1 2023

ECE 505 Computer Architecture

Prof. Xinming Huang

Table of Contents

Introduction

In this project, I was tasked to develop a hardware design for a single cycle processor using the RISC-V instruction set architecture in Verilog in the Vivado ISE to design and simulate our processor. The design I developed was required to implement these instructions: **ADD, ADDI, SW, LW, ADD, SUB, AND, OR, MUL, SLLI.**

In each stage of development, I was provided test programs and developed testbenches to validate the design and ensure measured performance was up to standard. Through the report, provided and written assembly code that was translated into machine code using Venus (https://venus.kvakil.me/) along with a set of waveforms generated by the processor will be shown and given an explanation. All machine code generated by the Venus tool is copied into the instruction memory of the processor.

Processor Structure

Single cycle CPUs are designed with the sole intention of running each instruction in only one clock cycle as the name states. Below is an example of the architecture of a RISC-V Single Cycle CPU.

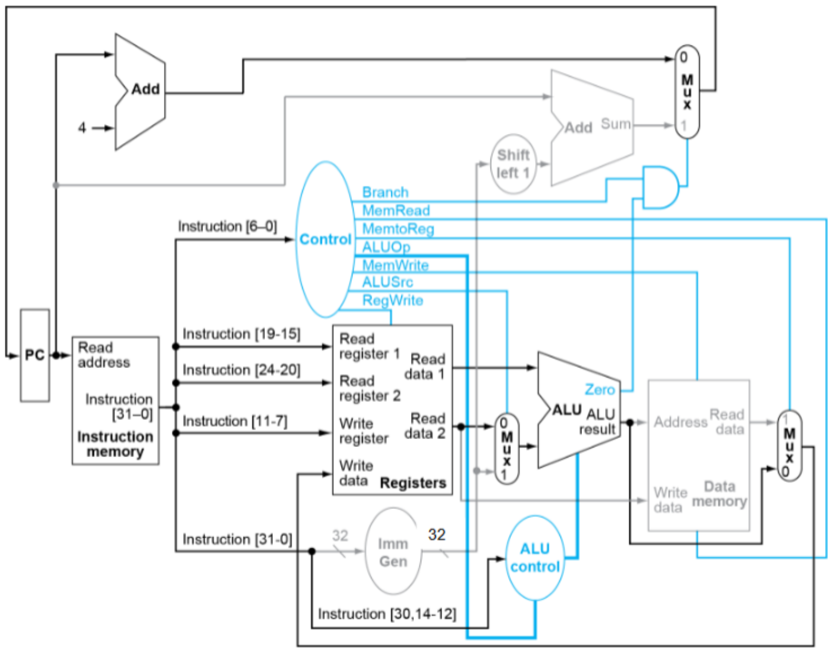


Figure 1: Example Single-Cycle CPU Datapath an R-Type Instruction

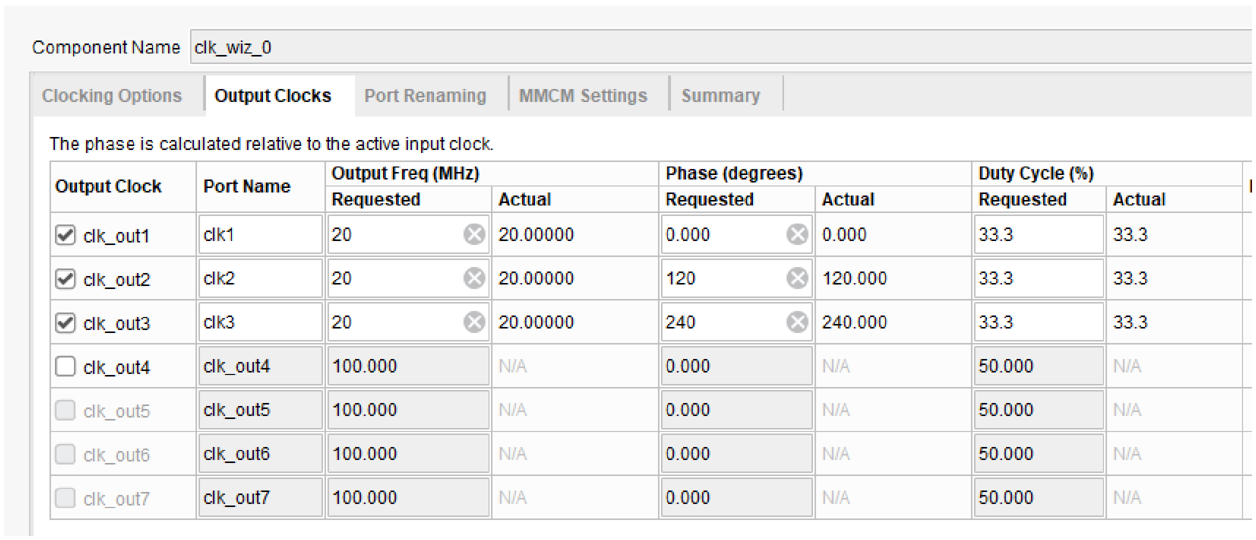
The processor designed in this project had some design requirements, with those being

1. Implement the RISC-V instructions in the previous section
2. Run at a **minimum** of 20MHz (Clock Frequency)
3. Utilize the IP catalog for the data memory and ensure stability with it.

Each subsection will describe how I implemented each module and the interconnections between them.

Clock Wizard IP

With using the IP catalog in Vivado, I was required to have the clock run at 20MHz, while allowing the processor to have **at least** half a cycle to fetch and execute instructions, as well as the rest to access data memory and update the register. To solve this issue I decided to create from a 20 MHz input clock, **three** output clocks, each phased 120 degrees from each other, as shown in figure 2.



Component Name clk_wiz_0

Clocking Options | Output Clocks | Port Renaming | MMCM Settings | Summary

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)	
		Requested	Actual	Requested	Actual	Requested	Actual
✓ clk_out1	clk1	20	20.00000	0.000	0.000	33.3	33.3
✓ clk_out2	clk2	20	20.00000	120	120.000	33.3	33.3
✓ clk_out3	clk3	20	20.00000	240	240.000	33.3	33.3
☐ clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000	N/A
☐ clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000	N/A
☐ clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000	N/A
☐ clk_out7	clk_out7	100.000	N/A	0.000	N/A	50.000	N/A

Figure 2: Clock IP Wizard Configuration

Within this configuration, the first output clock controls the program counter (PC), the second output clock phased at 120 degrees is used in the block RAM to access memory, and the last output clock is phased at 240 degrees,controlling the write back to the registers.

Block Ram IP (Data Memory)

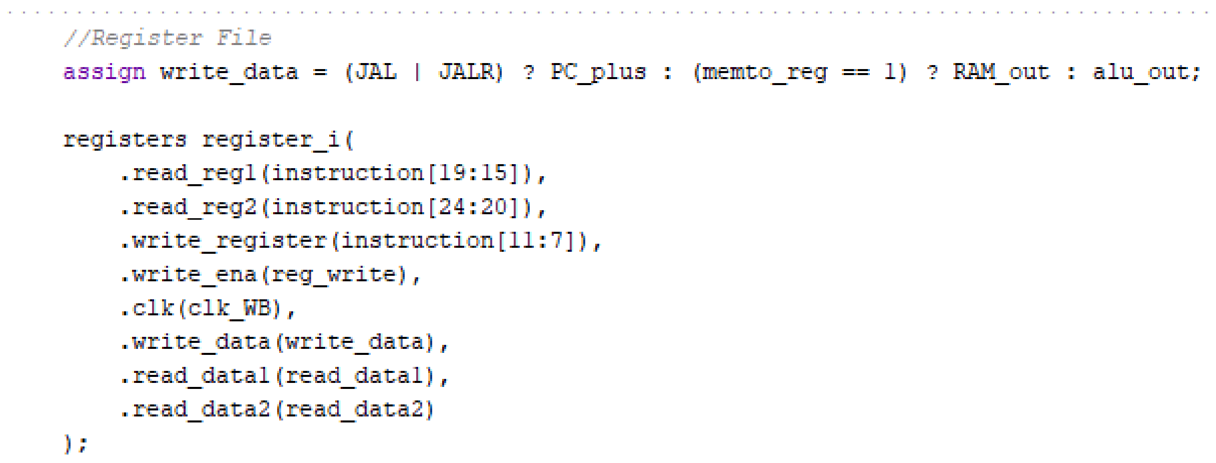
As recommended in the requirements, the Block Ram IP is used as a storage for data memory rather than creating one from scratch. Within this block, the data memory size was made to be 256 words (1024 bits) and 32 bits deep as the RISCV32 library uses 32-bit values. The data RAM uses the clock phased at the **falling edge** of the program counter's clock as shown in Figure 2 in order to allow time to fetch the instructions. The IP is configured as shown in Figure 3.

Program Counter

The program counter is used to keep track of what instruction is being/will be executed and is contained within the top-level module of the design. It is controlled by the first phased clock, It is only updated after the lock signal from the clock IP is pulled high to avoid any stability issues with the clock. The program counter should run as long as the instruction is not a HALT, where in which the program counter will stop changing. Other signals, such as branch, jump-and-link (JAL) and jump-and-link-register (JALR) will also change where the program counter ends up, besides the traditional PC + 4, as shown in Figure 4.

Registers

For the registers module, the read and write reg busses are determined from sections of the instruction (Figure 5) that the PC directs to. Clocking-wise, the third phase clock controls when writing back to the registers happens. The data written is being controlled by if either J-type flag is raised, or if the operation is writing memory to a register (Load Word), or the output of the ALU module otherwise. The write enable signal is triggered on whether or not the register write signal from the control module is raised high.



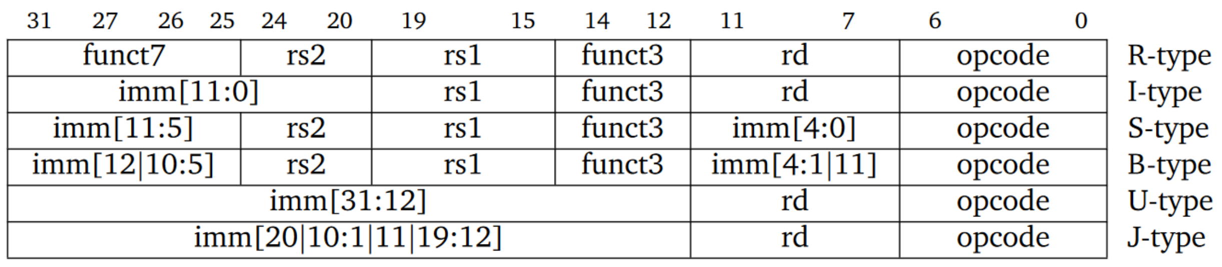
```
//Register File
assign write_data = (JAL | JALR) ? PC_plus : (memto_reg == 1) ? RAM_out : alu_out;

registers register_i(
    .read_reg1(instruction[19:15]),
    .read_reg2(instruction[24:20]),
    .write_register(instruction[11:7]),
    .write_ena(reg_write),
    .clk(clk_WB),
    .write_data(write_data),
    .read_data1(read_data1),
    .read_data2(read_data2)
);
```

Figure 5: Register Module Instantiation in Top Level Module

Within the module itself, a 32-by-32 2D array is generated to create the 32, 32-bit registers needed for the ISA to properly work. All of the registers are initialized to 0 at the start and the outputs of the read data busses are assigned based on where the read_reg inputs points to in the register file array. Whenever the write-back clock is high, if the write enable flag is raised, the register pointed to in the array by the write_register bus is assigned to the value of write_data, while the location of X0 is kept assigned to 0.

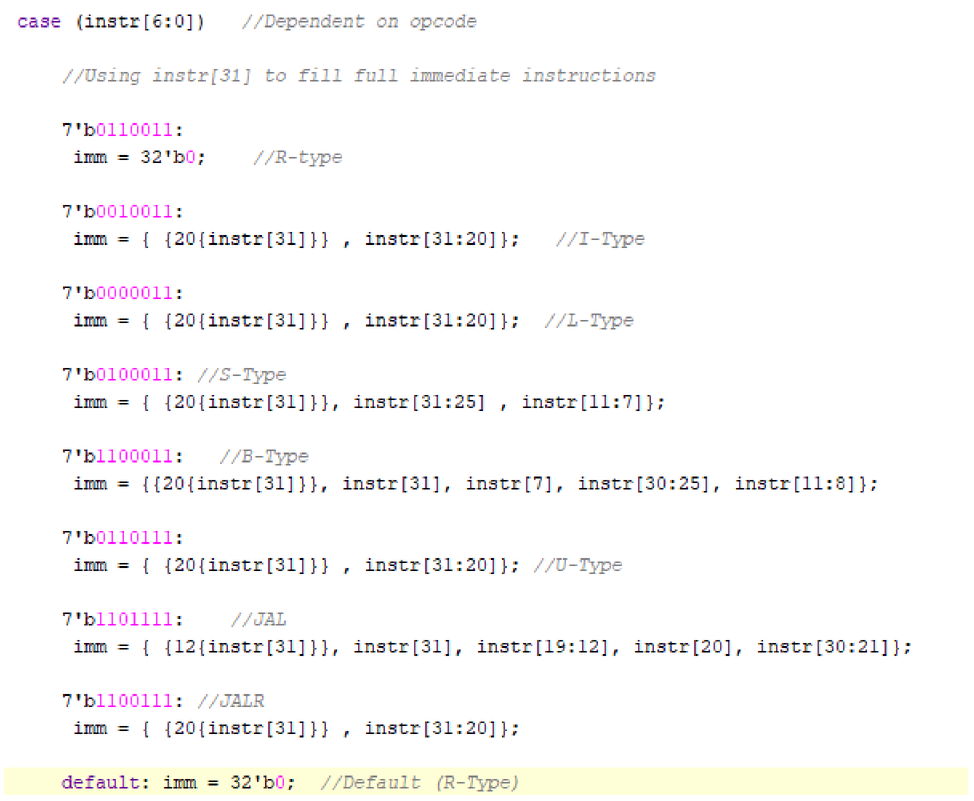
Immediate Generation

Within each instruction, a section of the instruction is part of the immediate. Within the immediate generation module, we are looking specifically at the last 7 bits of the instruction (the opcode) to determine what the immediate value should be, with using Figure 6 as a reference guide for bit concatenation in Figure 7. The most-significant bit (MSB) of the instruction input is used to fill the beginning of the immediate function wherever it is necessary.



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

Figure 6: Core Instruction Formats Table

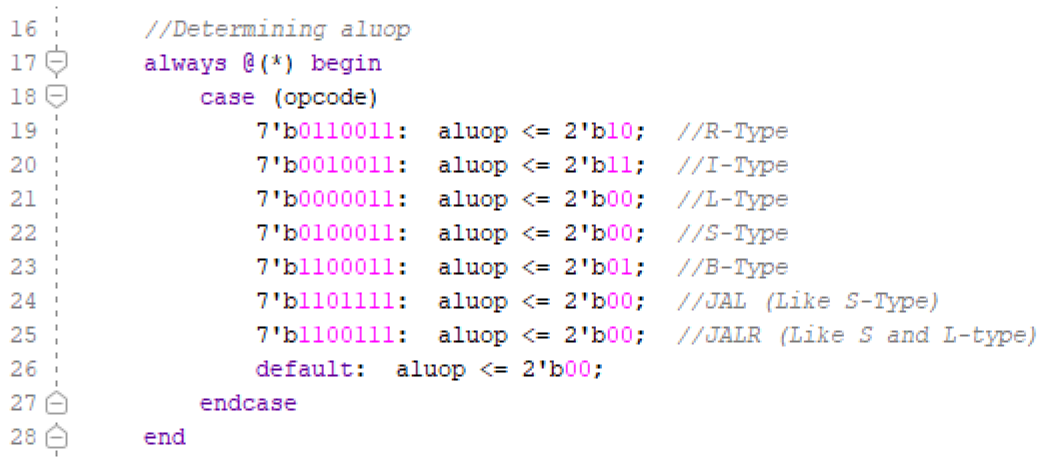


```
case (instr[6:0])    //Dependent on opcode

    //Using instr[31] to fill full immediate instructions

    7'b0110011:
     imm = 32'b0;     //R-type

    7'b0010011:
     imm = { {20{instr[31]}} , instr[31:20]};    //I-Type

    7'b0000011:
     imm = { {20{instr[31]}} , instr[31:20]};   //L-Type

    7'b0100011: //S-Type
     imm = { {20{instr[31]}}, instr[31:25] , instr[11:7]};

    7'b1100011:    //B-Type
     imm = {{20{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8]};

    7'b0110111:
     imm = { {20{instr[31]}} , instr[31:20]}; //U-Type

    7'b1101111:    //JAL
     imm = { {12{instr[31]}}, instr[31], instr[19:12], instr[20], instr[30:21]};

    7'b1100111: //JALR
     imm = { {20{instr[31]}} , instr[31:20]};

    default: imm = 32'b0;   //Default (R-Type)
```

Figure 7: Immediate Assignment based on Opcode in Verilog Module

Control Module

The control module can be thought of as the “brain” of our design; it determines what the ALU control module tells the ALU module what instruction to perform, alerts the program counter of a branch of J-type instruction, if the second operand of an ALU operation is coming from the immediate, or if a register write function is going to be performed.the 7 least significant bits from the instructions are used to determine what type of operation is being performed, as well as bits [14:12] in differentiating between a branch if equal (BEQ) and branch if not equal (BNE) operation. Flags for operations that involve memory read, memory to register, or memory write are also controlled here. Case statements like the ones shown in Figure 6 are used to determine the ALU operation code to be sent to the ALU control module, based on the type of instruction.

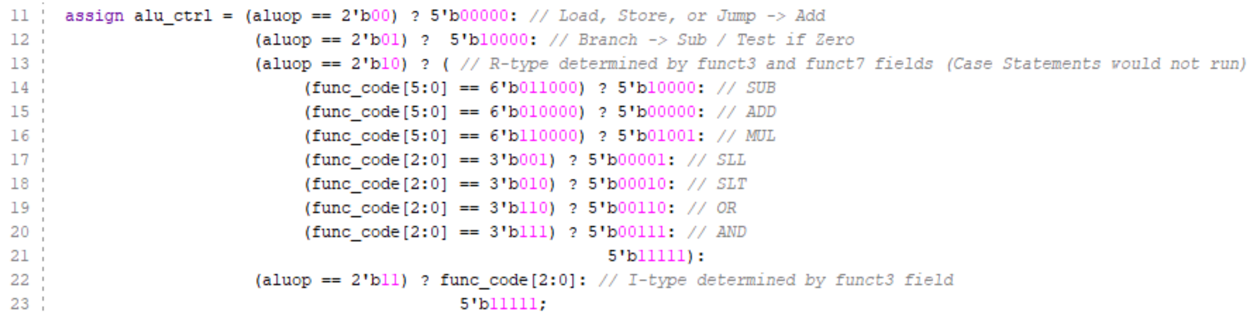


```verilog
//Determining aluop
always @(*) begin
    case (opcode)
        7'b0110011:  aluop <= 2'b10;  //R-Type
        7'b0010011:  aluop <= 2'b11;  //I-Type
        7'b0000011:  aluop <= 2'b00;  //L-Type
        7'b0100011:  aluop <= 2'b00;  //S-Type
        7'b1100011:  aluop <= 2'b01;  //B-Type
        7'b1101111:  aluop <= 2'b00;  //JAL (Like S-Type)
        7'b1100111:  aluop <= 2'b00;  //JALR (Like S and L-type)
        default:  aluop <= 2'b00;
    endcase
end
```

Figure 6: Determination of ALU Operation in Control Module

ALU Control

From the ALU operation output of the control module as well as a concatenation of bits from the instruction module that create funct7 ({instruction[25], instruction[5], instruction[30], instruction[14:12]}), a five-bit output can be generated to tell the ALU module what kind of operation to perform. Load store, and jump operations are translated into add operations within the ALU, branching instructions are subtraction operations with a Zero flag that is fed into the control module to recognize a working branch, R-type instructions are determined by the funct7 and funct3 fields, while I-type instructions are determined by the funct3 field (instruction[14:12]), as shown in Figure 7.

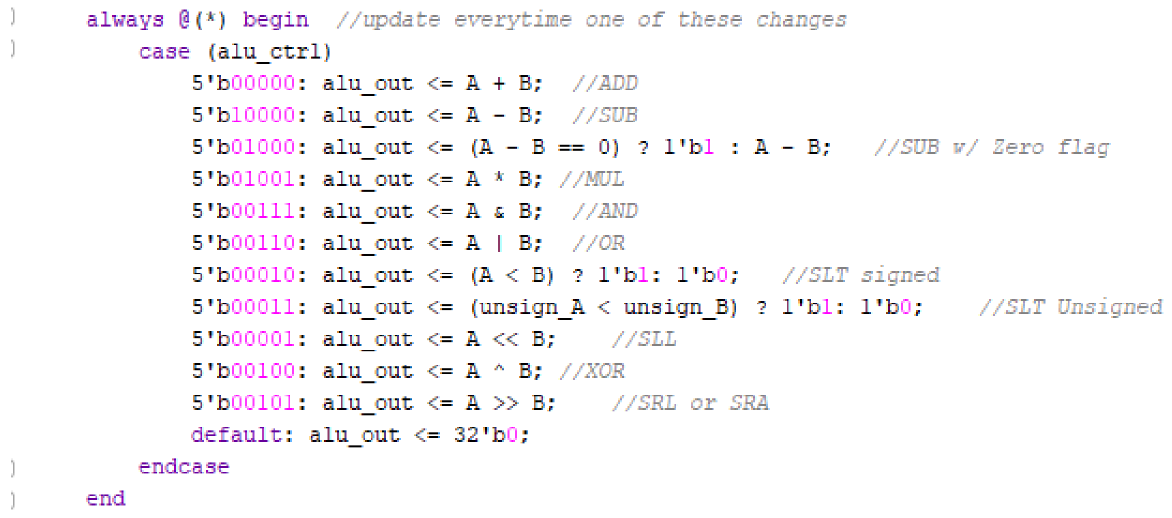


```verilog
assign alu_ctrl = (aluop == 2'b00) ? 5'b00000: // Load, Store, or Jump -> Add
                  (aluop == 2'b01) ?  5'b10000: // Branch -> Sub / Test if Zero
                  (aluop == 2'b10) ? ( // R-type determined by funct3 and funct7 fields (Case Statements would not run)
                      (func_code[5:0] == 6'b011000) ? 5'b10000: // SUB
                      (func_code[5:0] == 6'b010000) ? 5'b00000: // ADD
                      (func_code[5:0] == 6'b110000) ? 5'b01001: // MUL
                      (func_code[2:0] == 3'b001) ? 5'b00001: // SLL
                      (func_code[2:0] == 3'b010) ? 5'b00010: // SLT
                      (func_code[2:0] == 3'b110) ? 5'b00110: // OR
                      (func_code[2:0] == 3'b111) ? 5'b00111: // AND
                                                    5'b11111):
                  (aluop == 2'b11) ? func_code[2:0]: // I-type determined by funct3 field
                                      5'b11111;
```

Figure 7: Determination of ALU Control Module Output based on ALU Operation and Funct7 and Funct3

ALU

The ALU module handles all of the arithmetic operations of the processor. From the outputs of the ALU control module, the ALU module can perform the correct operations on the two operands, with the first, A, being determined by the read_data1 output of the register module and the second operand, B, being either the output of the immediate, or read_data2 from the register module, dependent on whether or not the ALUSrc flag is raised high or low respectively. A and B are signed inputs, so wires for unsigned operations are created that take the values of A and B. From there, a set of case statements can be made dependent on alu_ctrl to determine the output of the ALU module. The zero flag used in the control module is also assigned here and is pulled high whenever the alu output is 0, or if the subtracting with zero flag operation occurs and assigns alu_out a value of 1. The full list of operations that are performed in this design can be seen in Figure 8.



```
always @(*) begin  //update everytime one of these changes
    case (alu_ctrl)
        5'b00000: alu_out <= A + B;  //ADD
        5'b10000: alu_out <= A - B;  //SUB
        5'b01000: alu_out <= (A - B == 0) ? 1'b1 : A - B;   //SUB w/ Zero flag
        5'b01001: alu_out <= A * B; //MUL
        5'b00111: alu_out <= A & B;  //AND
        5'b00110: alu_out <= A | B;  //OR
        5'b00010: alu_out <= (A < B) ? 1'b1: 1'b0;   //SLT signed
        5'b00011: alu_out <= (unsign_A < unsign_B) ? 1'b1: 1'b0;    //SLT Unsigned
        5'b00001: alu_out <= A << B;    //SLL
        5'b00100: alu_out <= A ^ B; //XOR
        5'b00101: alu_out <= A >> B;    //SRL or SRA
        default: alu_out <= 32'b0;
    endcase
end
```

Figure 8: List of ALU Operations Performed in Module

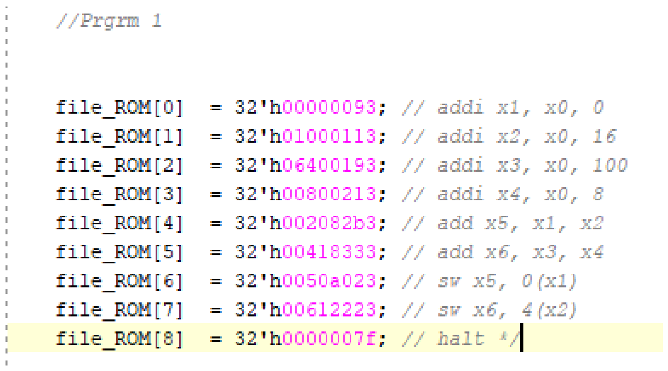
Instruction Read-Only-Memory (ROM)

Within the top-level, the address in the instruction memory is determined by the program counter divided by 4 (PC = 4 -> address = 1, etc.) , with the output instruction being determined inside of the instruction memory module, as the ROM component will not be updated until the positive edge of the clock. Instead, each set of programs is written in machine code as a 32-bit register inside a 32-by-32 2D array, similar to the register module. As each program uses the same 2D array, only one set of instructions can be uncommented at a time for use while in simulation. The set of assembly codes in the instruction memory module is shown in the figures in the following section.

Programs and Waveforms

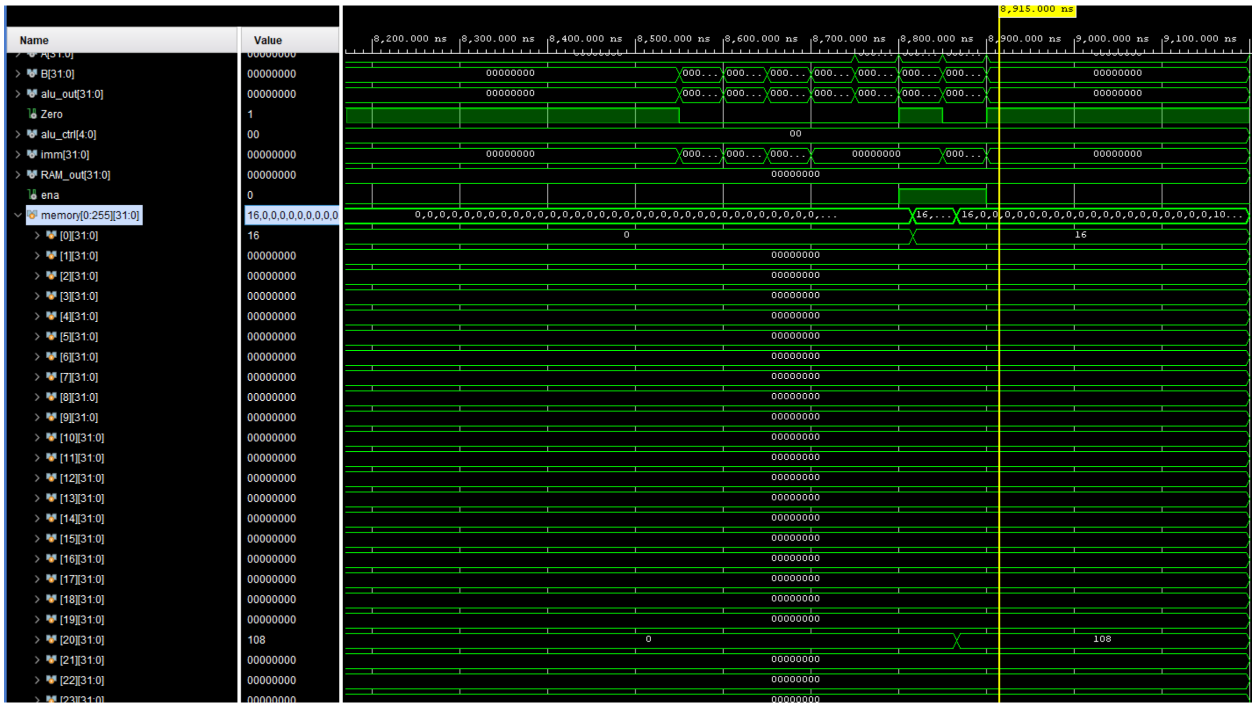
Program 1: Add, Addi, SW

Within this first given program, the X1, X2, X3, and X4 registers are initialized to 0, 16, 100, and 8 respectively. From there, X5 becomes the sum of X1 and X2, while X6 is the sum of X3 and X4. These sums (16 and 108 respectively) are then stored at 2 separate memory locations. The machine code with assembly commented and the simulated waveform in Figures A and B show the processor successfully running these programs.



```
//Prgrm 1


file_ROM[0]  = 32'h00000093; // addi x1, x0, 0
file_ROM[1]  = 32'h01000113; // addi x2, x0, 16
file_ROM[2]  = 32'h06400193; // addi x3, x0, 100
file_ROM[3]  = 32'h00800213; // addi x4, x0, 8
file_ROM[4]  = 32'h002082b3; // add x5, x1, x2
file_ROM[5]  = 32'h00418333; // add x6, x3, x4
file_ROM[6]  = 32'h0050a023; // sw x5, 0(x1)
file_ROM[7]  = 32'h00612223; // sw x6, 4(x2)
file_ROM[8]  = 32'h0000007f; // halt */
```

Figure 9: Program 1 Machine Code in Instruction Memory Module



Name	Value
B[31:0]	00000000
alu_out[31:0]	00000000
Zero	1
alu_ctrl[4:0]	00
imm[31:0]	00000000
RAM_out[31:0]	00000000
ena	0
memory[0:255][31:0]	16,0,0,0,0,0,0,0,0,0
[0][31:0]	16
[1][31:0]	00000000
[2][31:0]	00000000
[3][31:0]	00000000
[4][31:0]	00000000
[5][31:0]	00000000
[6][31:0]	00000000
[7][31:0]	00000000
[8][31:0]	00000000
[9][31:0]	00000000
[10][31:0]	00000000
[11][31:0]	00000000
[12][31:0]	00000000
[13][31:0]	00000000
[14][31:0]	00000000
[15][31:0]	00000000
[16][31:0]	00000000
[17][31:0]	00000000
[18][31:0]	00000000
[19][31:0]	00000000
[20][31:0]	108
[21][31:0]	00000000
[22][31:0]	00000000

Figure 10: Program 1 Simulation Waveform Memory Dump

When comparing our simulation results in Figure 10 to the ones from the supporting material's waveform in Figure 11, we can see that the memory dump is the exact same, with Figure 10 showing the values in decimal and in hexadecimal in Figure 11.

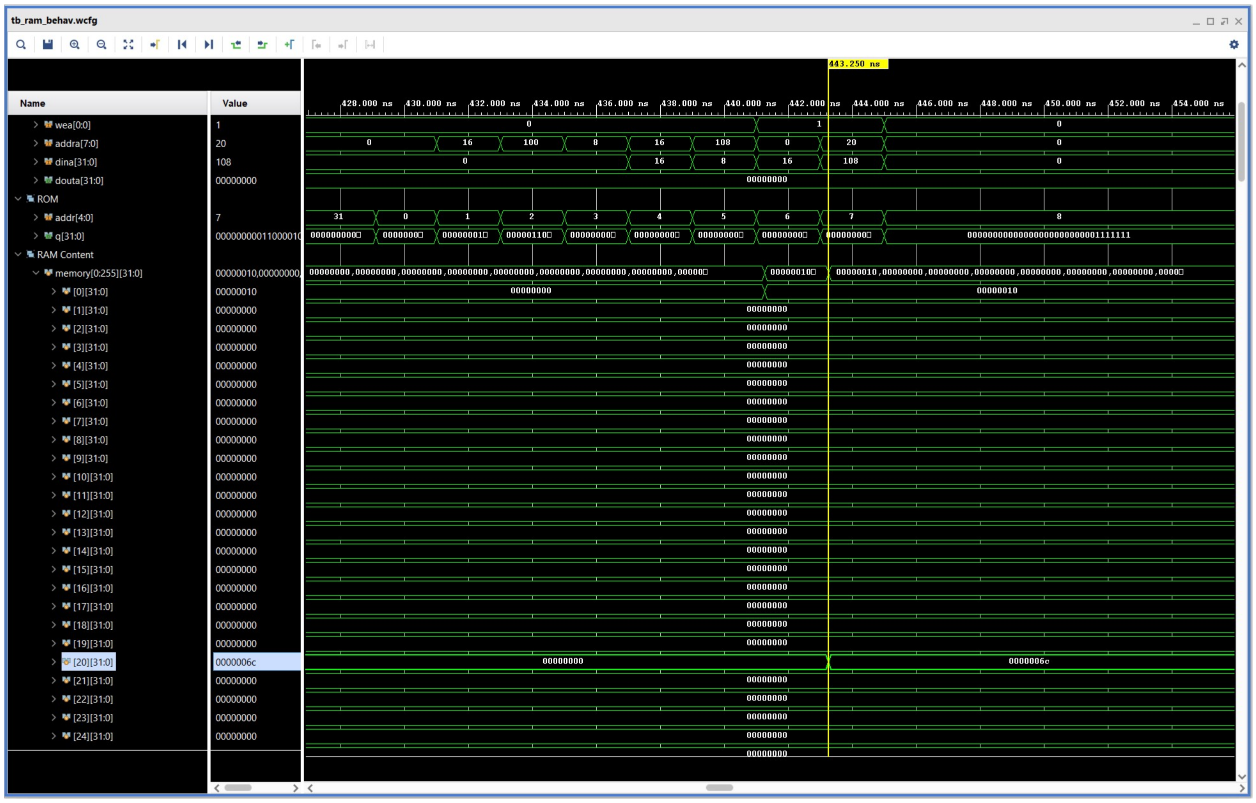
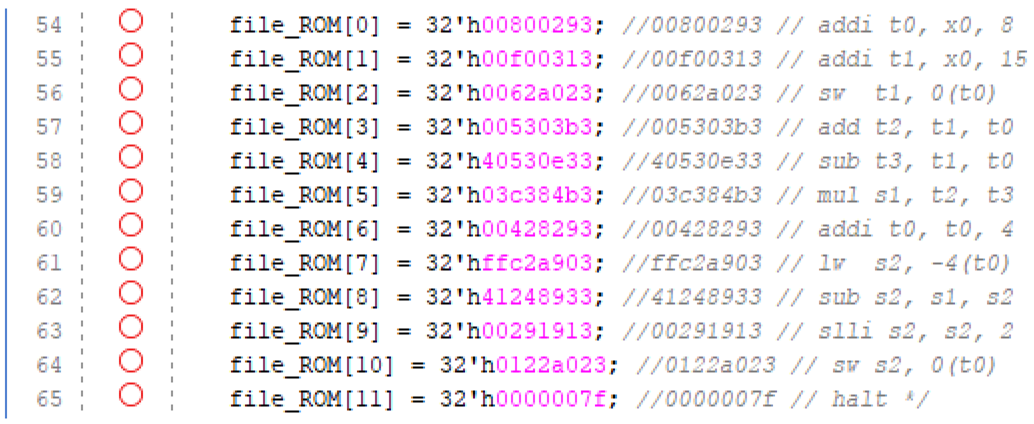


Figure 11: Supporting Material Memory Dump Program 1

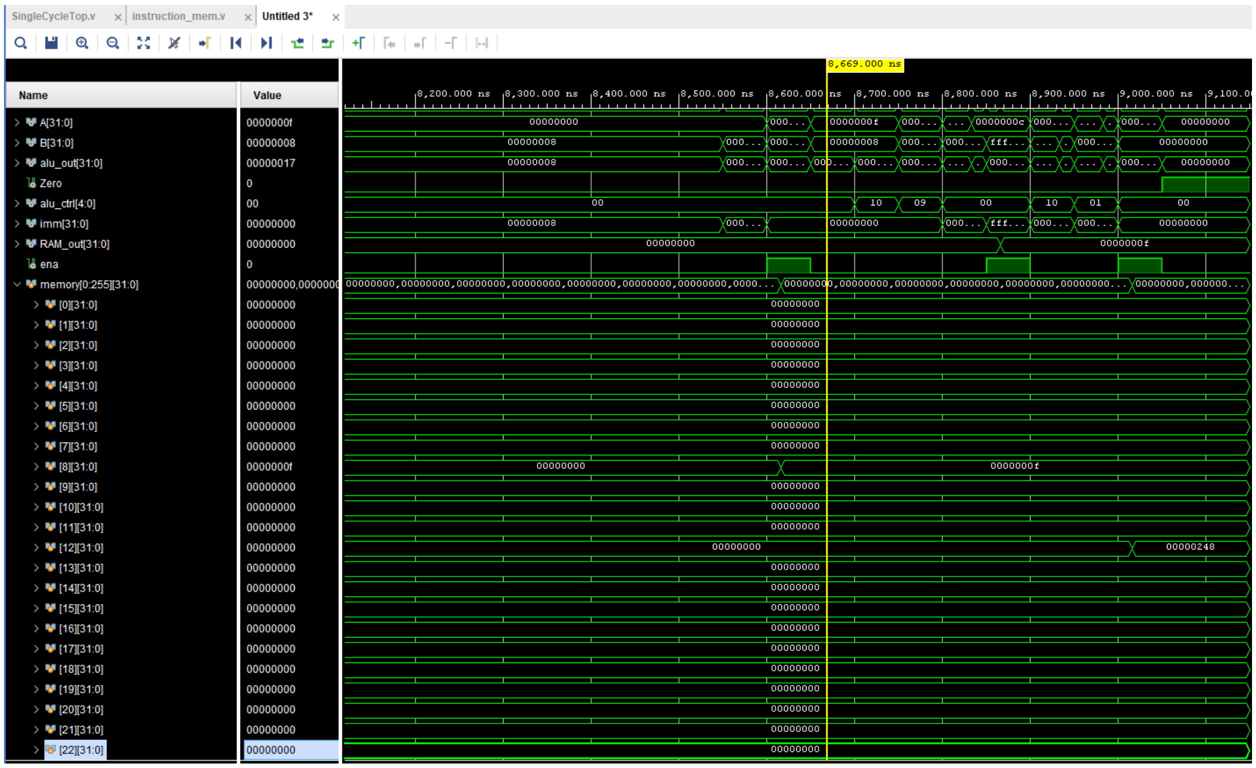
Program 2: Mul, SW, LW, SLLI

Within this set of instructions, T0 and T1 are assigned the respective values of 8 and 15, with t1 being stored in the memory initially. From there T2 is the sum of T0 and T1, and T3 is the difference between T1 and T0. The T2 and T3 registers are then multiplied together to result in S1. T0 is then incremented, and a value is loaded into S2. S2 is then calculated as the difference between S1 and itself. After that, S2 is shifted left by 2 and stored at the end of the program. Figures 12 and 13 show the assembly code and simulation waveform. Figure 14 shows the supporting material's memory dump of the same code



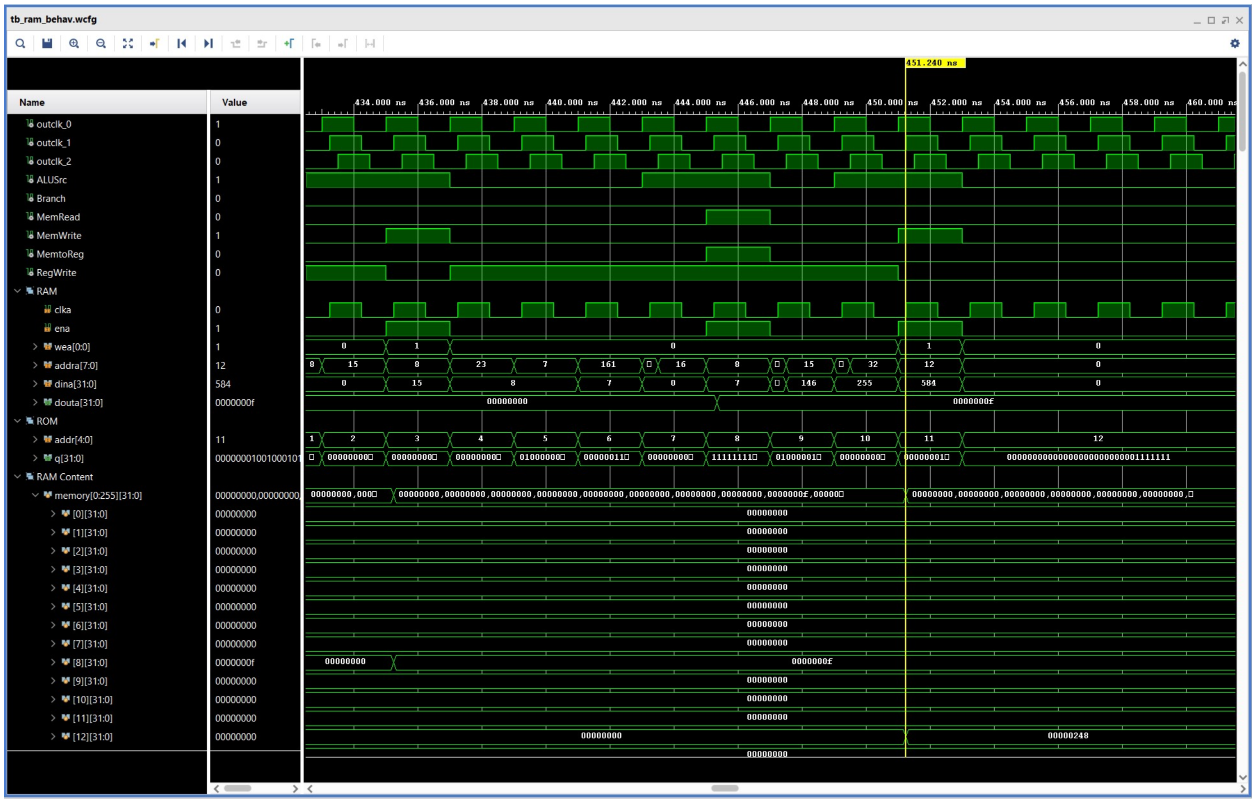
```
    file_ROM[0] = 32'h00800293; //00800293 // addi t0, x0, 8
    file_ROM[1] = 32'h00f00313; //00f00313 // addi t1, x0, 15
    file_ROM[2] = 32'h0062a023; //0062a023 // sw  t1, 0(t0)
    file_ROM[3] = 32'h005303b3; //005303b3 // add t2, t1, t0
    file_ROM[4] = 32'h40530e33; //40530e33 // sub t3, t1, t0
    file_ROM[5] = 32'h03c384b3; //03c384b3 // mul s1, t2, t3
    file_ROM[6] = 32'h00428293; //00428293 // addi t0, t0, 4
    file_ROM[7] = 32'hffc2a903; //ffc2a903 // lw  s2, -4(t0)
    file_ROM[8] = 32'h41248933; //41248933 // sub s2, s1, s2
    file_ROM[9] = 32'h00291913; //00291913 // slli s2, s2, 2
    file_ROM[10] = 32'h0122a023; //0122a023 // sw s2, 0(t0)
    file_ROM[11] = 32'h0000007f; //0000007f // halt */
```

Figure 12: Program 2 Machine Code in Instruction Memory Module



Name	Value
A[31:0]	0000000f
B[31:0]	00000008
alu_out[31:0]	00000017
Zero	0
alu_ctrl[4:0]	00
imm[31:0]	00000000
RAM_out[31:0]	00000000
ena	0
memory[0:255][31:0]	0000000,000000
[0][31:0]	00000000
[1][31:0]	00000000
[2][31:0]	00000000
[3][31:0]	00000000
[4][31:0]	00000000
[5][31:0]	00000000
[6][31:0]	00000000
[7][31:0]	00000000
[8][31:0]	0000000f
[9][31:0]	00000000
[10][31:0]	00000000
[11][31:0]	00000000
[12][31:0]	00000000
[13][31:0]	00000000
[14][31:0]	00000000
[15][31:0]	00000000
[16][31:0]	00000000
[17][31:0]	00000000
[18][31:0]	00000000
[19][31:0]	00000000
[20][31:0]	00000000
[21][31:0]	00000000
[22][31:0]	00000000

Figure 13: Program 2 Memory Waveform



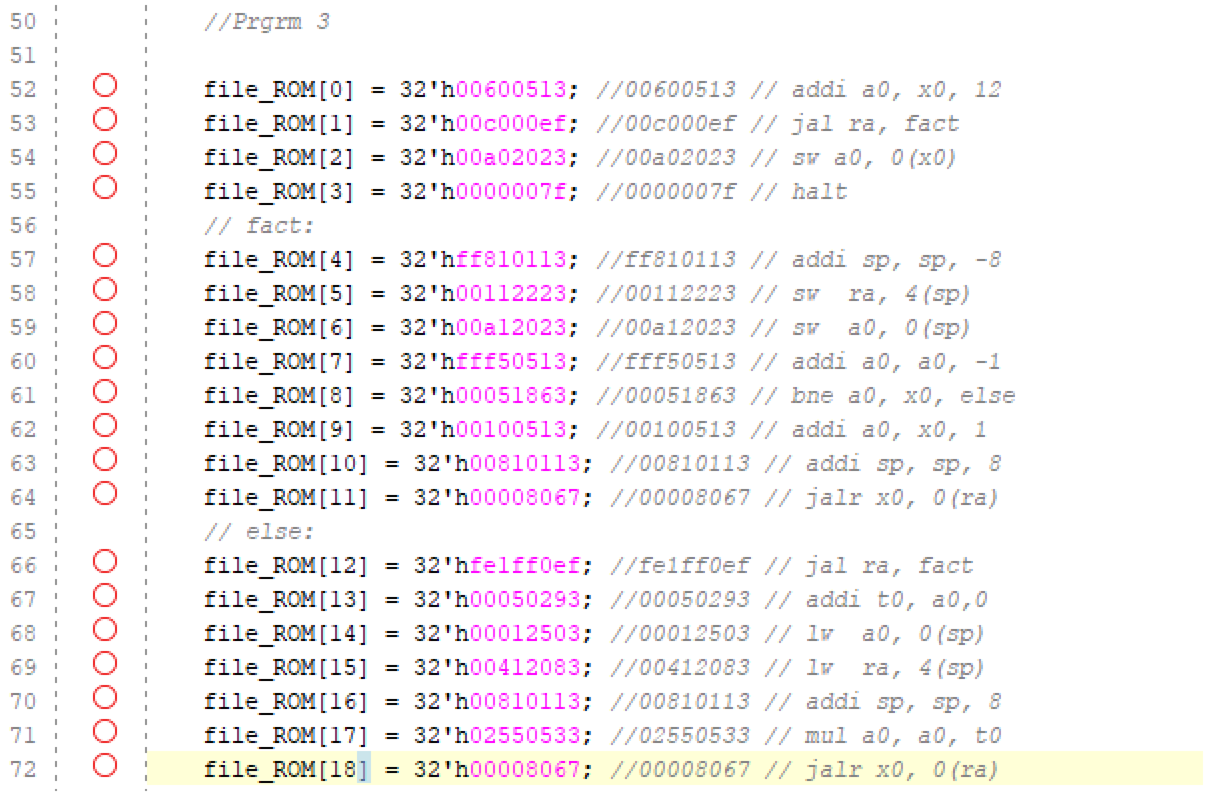
tb_ram_behav.wcfg

Name	Value
outclk_0	1
outclk_1	0
outclk_2	0
ALUSrc	1
Branch	0
MemRead	0
MemWrite	1
MemtoReg	0
RegWrite	0
RAM	
clka	0
ena	1
wea[0:0]	1
addra[7:0]	12
dina[31:0]	584
douta[31:0]	0000000f
ROM	
addr[4:0]	11
q[31:0]	00000001001000101
RAM Content	
memory[0:255][31:0]	00000000,00000000,
[0][31:0]	00000000
[1][31:0]	00000000
[2][31:0]	00000000
[3][31:0]	00000000
[4][31:0]	00000000
[5][31:0]	00000000
[6][31:0]	00000000
[7][31:0]	00000000
[8][31:0]	0000000f
[9][31:0]	00000000
[10][31:0]	00000000
[11][31:0]	00000000
[12][31:0]	00000000

Figure 14: Supporting Material Memory Dump Program 2

Program 3 : Factorial

The next program runs the factorial code. For this section, we were given the assembly code and used the Venus tool described in the introduction to translate into the machine code as shown in Figure 15. Figures 16 and 17



```
//Prgrm 3

file_ROM[0] = 32'h00600513; //00600513 // addi a0, x0, 12
file_ROM[1] = 32'h00c000ef; //00c000ef // jal ra, fact
file_ROM[2] = 32'h00a02023; //00a02023 // sw a0, 0(x0)
file_ROM[3] = 32'h0000007f; //0000007f // halt
// fact:
file_ROM[4] = 32'hff810113; //ff810113 // addi sp, sp, -8
file_ROM[5] = 32'h00112223; //00112223 // sw  ra, 4(sp)
file_ROM[6] = 32'h00a12023; //00a12023 // sw  a0, 0(sp)
file_ROM[7] = 32'hfff50513; //fff50513 // addi a0, a0, -1
file_ROM[8] = 32'h00051863; //00051863 // bne a0, x0, else
file_ROM[9] = 32'h00100513; //00100513 // addi a0, x0, 1
file_ROM[10] = 32'h00810113; //00810113 // addi sp, sp, 8
file_ROM[11] = 32'h00008067; //00008067 // jalr x0, 0(ra)
// else:
file_ROM[12] = 32'hfe1ff0ef; //fe1ff0ef // jal ra, fact
file_ROM[13] = 32'h00050293; //00050293 // addi t0, a0,0
file_ROM[14] = 32'h00012503; //00012503 // lw  a0, 0(sp)
file_ROM[15] = 32'h00412083; //00412083 // lw  ra, 4(sp)
file_ROM[16] = 32'h00810113; //00810113 // addi sp, sp, 8
file_ROM[17] = 32'h02550533; //02550533 // mul a0, a0, t0
file_ROM[18] = 32'h00008067; //00008067 // jalr x0, 0(ra)
```

Figure 15: Machine and Assembly Code Program 3

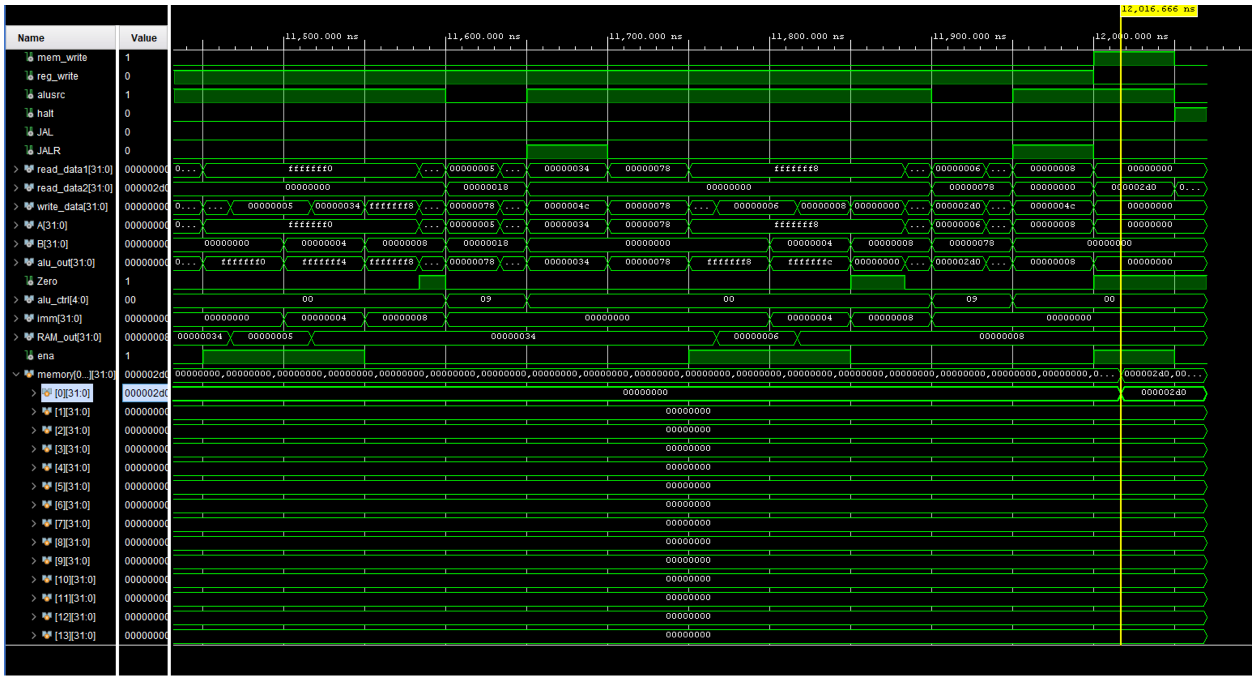


Figure 16: Simulation Waveform Memory Dump Program 3

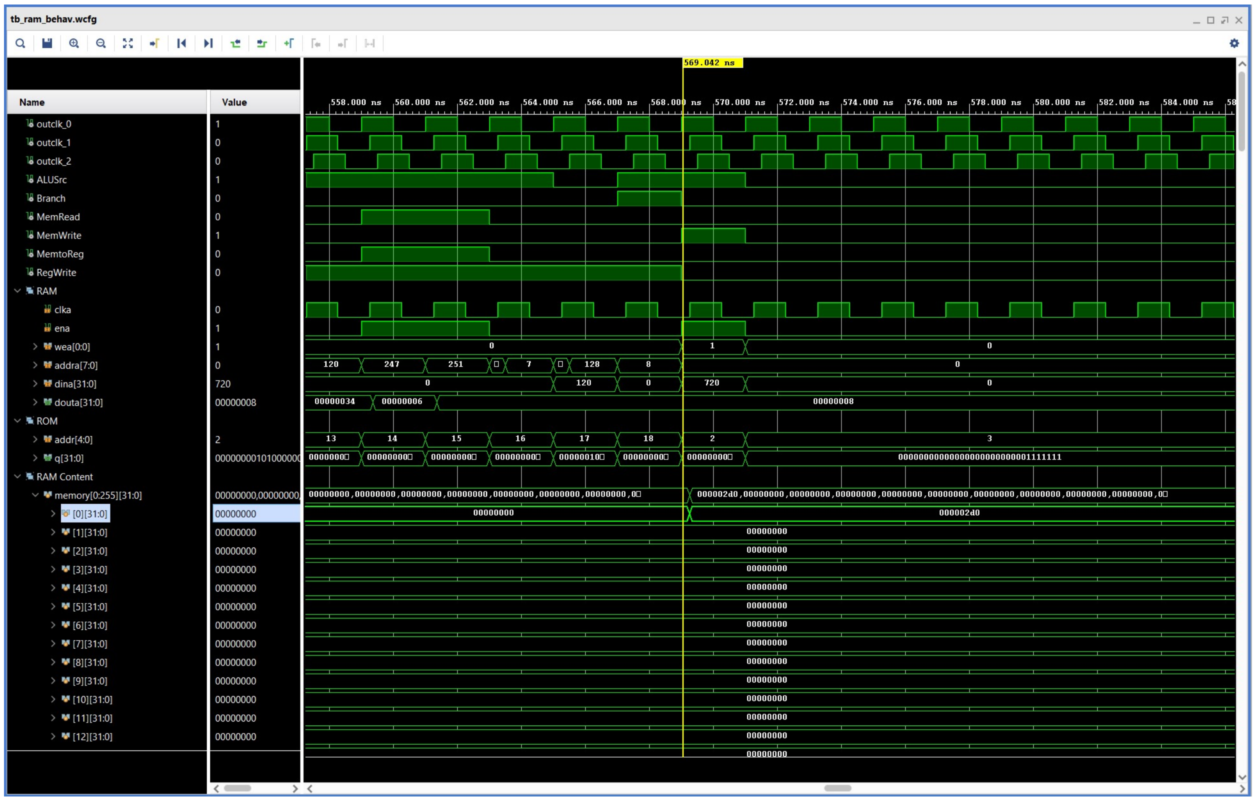
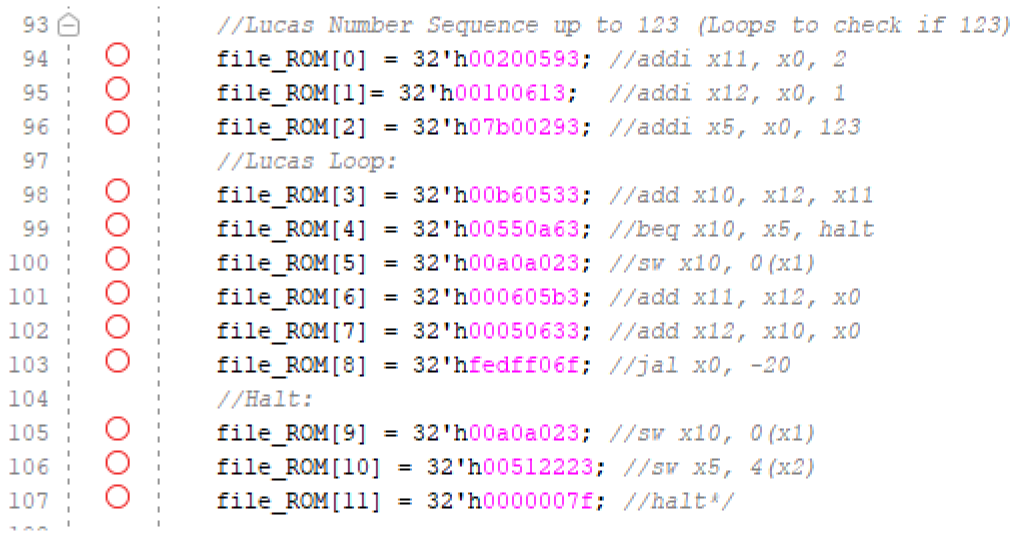


Figure 17: Supporting Material Memory Dump Program 3

Program 4/Custom Program: Lucas Sequence Loop

This program was created by myself to implement the Lucas Number Sequence loop. The Lucas sequence is similar to the Fibonacci sequence except that the first 2 values are swapped (first is 2 and second is 1, where Fibonacci is 1 -> 2). In my custom program, I have X10 be the sum of the two numbers in the Lucas sequence and test to see if it equals 123, if not X11 and X12 are updated to the next pair in the sequence, and the loop continues until X10 equals 123, as can be read in the created machine language and assembly code in Figure 18. When this condition occurs, the values of X10 and X5 are stored in the memory, as shown in Figure 19.



```
//Lucas Number Sequence up to 123 (Loops to check if 123)
file_ROM[0] = 32'h00200593; //addi x11, x0, 2
file_ROM[1]= 32'h00100613;  //addi x12, x0, 1
file_ROM[2] = 32'h07b00293; //addi x5, x0, 123
//Lucas Loop:
file_ROM[3] = 32'h00b60533; //add x10, x12, x11
file_ROM[4] = 32'h00550a63; //beq x10, x5, halt
file_ROM[5] = 32'h00a0a023; //sw x10, 0(x1)
file_ROM[6] = 32'h000605b3; //add x11, x12, x0
file_ROM[7] = 32'h00050633; //add x12, x10, x0
file_ROM[8] = 32'hfedff06f; //jal x0, -20
//Halt:
file_ROM[9] = 32'h00a0a023; //sw x10, 0(x1)
file_ROM[10] = 32'h00512223; //sw x5, 4(x2)
file_ROM[11] = 32'h0000007f; //halt*/
```

Figure 18: Lucas Number Sequence Machine Code

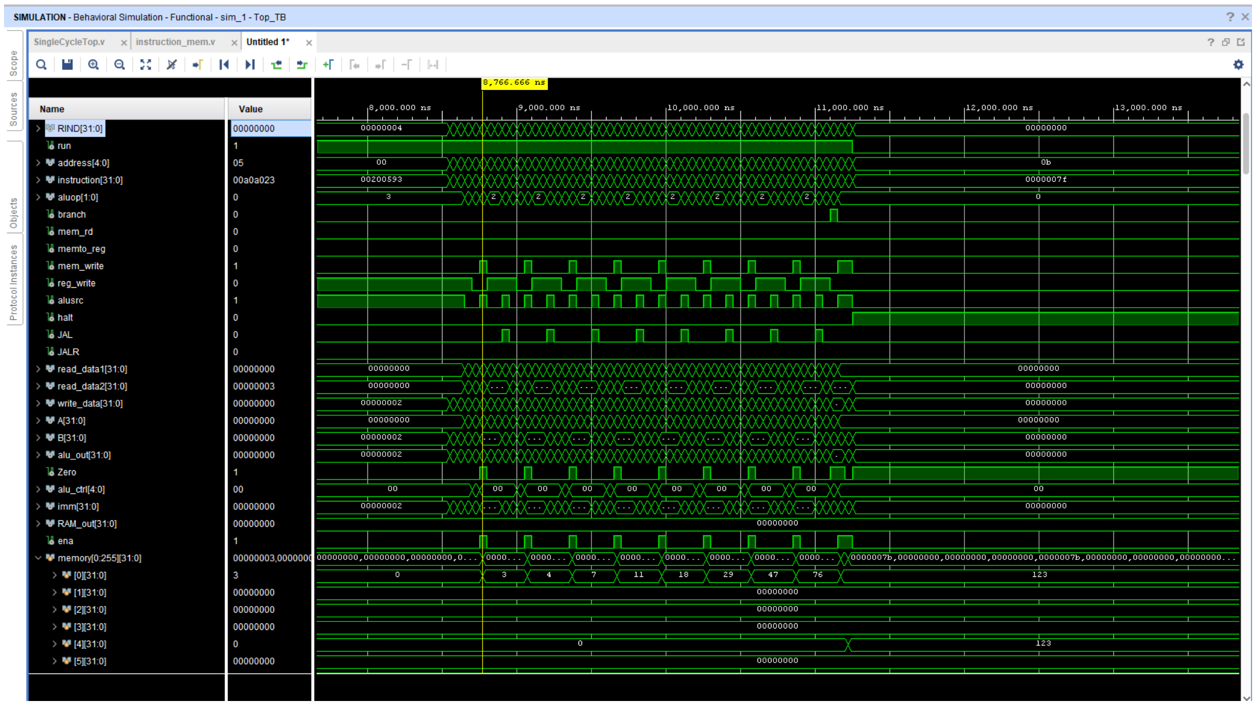
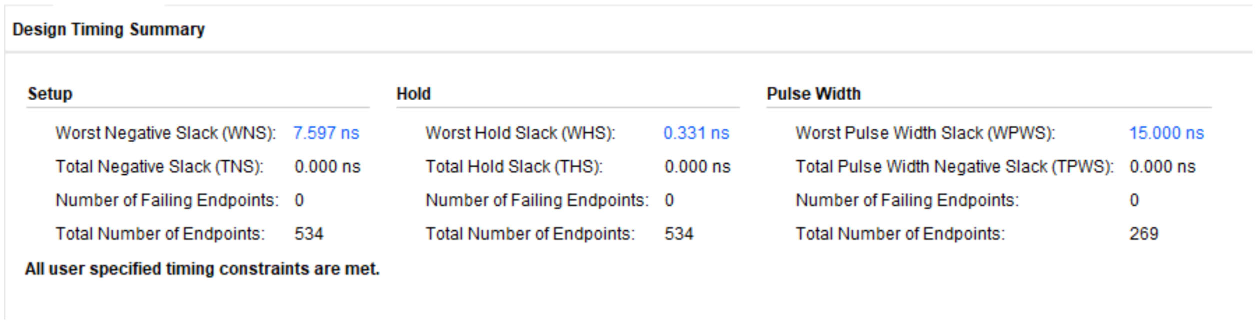


Figure 19: Lucas Number Sequence Memory Waveform

Synthesis & Design Utilization

Timing Results

From the Timing Summary from the synthesized project (Figure X), we can see that the worst negative slack (WNS) is around 7.6 ns and that our timing constraints are met. The maximum frequency can be calculated by taking the WNS, subtracting it from the period of our clock and dividing 1 by that (1/(*T-WNS*)). By doing so with a period of 50 ns (period for a freq of 20 MHz), we get a maximum frequency of 23.58 MHz.



Design Timing Summary

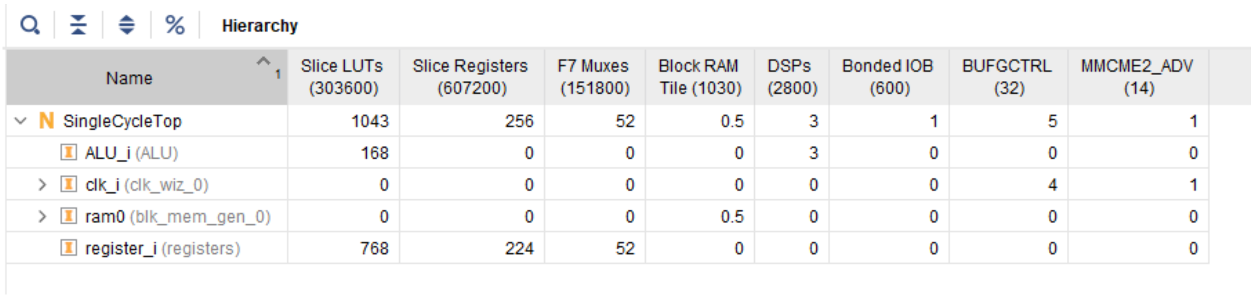
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7.597 ns	Worst Hold Slack (WHS):	0.331 ns	Worst Pulse Width Slack (WPWS):	15.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	534	Total Number of Endpoints:	534	Total Number of Endpoints:	269

All user specified timing constraints are met.

Figure 20: Vivado Synthesis Design Timing Summary

Resource Utilization

From the Resource Utilization Summary from Vivado shown in Figure Y, we can see that when the design was targeted to a Virtex-7 FPGA, the design uses a total of 1043 LUTs, 256 Slice Registers, 52 Muxes, 3 DSPs, 1 MMCM for the clock, and half of a block RAM tile for the block RAM IP. Most of the LUTs were used in the register and ALU modules and the Slice registers and Muxes used mainly by the register module.



Hierarchy

Name	Slice LUTs (303600)	Slice Registers (607200)	F7 Muxes (151800)	Block RAM Tile (1030)	DSPs (2800)	Bonded IOB (600)	BUFGCTRL (32)	MMCME2_ADV (14)
SingleCycleTop	1043	256	52	0.5	3	1	5	1
ALU_i (ALU)	168	0	0	0	3	0	0	0
clk_i (clk_wiz_0)	0	0	0	0	0	0	4	1
ram0 (blk_mem_gen_0)	0	0	0	0.5	0	0	0	0
register_i (registers)	768	224	52	0	0	0	0	0

Figure 21: Vivado Synthesis Resource Utilization Summary

Conclusion

With the set of performance metrics to hit and a limited library of RISC-V operations given, I was able to successfully learn and implement a single cycle CPU using Vivado. The project helps to provide a glimpse of what the world of processor development looks like, solidifying the material taught in the lectures with this hands-on approach. Through a rigorous set of test programs, I was able to display the functionality of my processor, while exceeding the 20MHz minimum clock speed, performing each instruction and operation properly.