DATE: 12/8/2024
TO: Prof. Patrick Farrell
FROM: Joaquin Calilao - IN33216
SUBJECT: CMPE 316 - Project #4

---

1 Overview

Project 4 is the culmination of the processor implementation on the Xilinx FPGA board. For this project, students were tasked with creating control logic via a finite state machine. Additionally, data memory via an SRAM block was added to the CPU architecture. All components were to be wrapped under a Verilog top file and the output of the processor is connected to **led_driver.v**. The following section goes into more detail regarding design choices made for the control logic as well as how each unit of the CPU was interfaced with one another in the final implementation.

2 Description of Design

The project 4 design consists of 4 total modules SystemVerilog modules: **program_counter.sv, reg_file.sv, alu.sv, fsm.sv**. Additionally, led_driver.v module for the 7-segment display, and 2 instances of block SRAM were included: one for instruction memory and one for data memory. All modules/IP blocks (excluding **led_driver.v**) were wrapped under a **cputop.v** file. A **cputop_tb.sv** testbench file was created in order to ensure that all CPU operations could be performed as expected.

Please note that another test module called **ins_mem_test.sv** was programmed. This module simulated the function of the instruction memory, and could be easily programmed with the appropriate instructions. During testing, the cputop.sv file would use a “generate” block to create an instance of the ins_mem_test module, rather than the actual custom IP block for the instruction memory. The “generate” block is controlled by a parameter that can switch between these two modules on the fly, depending on the version of the CPU being tested (for simulation or for synthesis).

Below is a matrix that describes the hardware interconnect based on the cputop.v file for clearer interpretation of the block diagram in Section 3.

This space is intentionally left blank.

Table 1: cputop.v Hardware Interconnect Description

Signal name	Bit Width	Direction	Board Pin	Registered	Description
clk	1	input	CLK100MHZ	no	Clock signal @ 100MHz
rst	1	input	BTNR [M17]	no	Reset button signal (posedge triggered)
SW	16	input		yes	Signals that allow the user to toggle between reading data from the IO register and the SRAM Data. SW[15] controls which module to read from, and SW[7:0] controls which address from SRAM Data memory to read from. SW[14:8] are left open in the final design.
data_o	8	output		yes	Data bus that comes out of the cputop (outputs either IO Register data or SRAM Data) and goes into the led_driver.v module to be displayed on the FPGA post-implementation

***NOTE: All other output ports are unused in the top-level cputop.v block and are not included in the table above.** These ports are specifically left on the module due to an issue in which the synthesis tool optimizes away lower level cells. Additional output ports are implemented to prevent this from happening. Additionally, these output ports essentially acts as built-in ILA, allowing the end-user to see the different states that the FSM goes through, different control signals that the FSM sends to other modules (ex: incrementing the PC), and the different data/address buses going between logic and memory elements. **They serve no other functional purpose at the top level aside from testing.**

2.3 fsm.sv: Control Logic Design Overview

The logic of fsm.sv is broken into 3 procedural statements for easy interpretation: 2 sequential blocks and 1 combinatorial. The first sequential statement handles registering all inputs, outputs, and **several internal control flags (See Section 2.3.1)** of the Finite State Machine (FSM), as well as clearing out those respective registers during a posedge reset. The second sequential statement is responsible for driving a clock cycle counter that is internal to the FSM.

Finally, the combinatorial block handles the logic for determining the behavior of the Mealy State Machine. A total of 9 states were created. A table below describes the function of each state and the control signals that directly affect the behavior of the FSM. Furthermore, a matrix that describes the hardware interconnect of the fsm.sv module is also provided.

Table 1: FSM States Description

State name	Description
idle	stays idle for 10 cycles (CPU is inactive)
fetch	allows new instruction at the input to propagate through
decode	decodes the instruction/determines operation (sets control flags to control state flow)
execute	allows ALU to compute value from current operands and send the result back
loadfsm	loads data from ALU to output ports and other sources to be written back to the register file
write	FSM pulses write enable signals to the appropriate memory units based on current op
advance	sends inc signal to the PC to increment the address in instruction memory
advance2	sends either JUMP, JUMPC, CALL, or RET based on current op or condition (JUMPC)
terminate	prevents incrementing of PC and loops on this state forever (end of execution, must reset)

Table 2: fsm.sv Hardware Interconnect Description

Signal name	Bit Width	Direction	Board Pin	Registered	Description
clk	1	input	CLK100MHZ	no	Clock signal @ 100MHz
rst	1	input	BTNR [M17]	no	Reset button signal
instruction	8	input	N/A	yes	Instruction data bus that is connected directly to instruction memory. Used to fetch next instruction or jump/call address
ci, zi, ni	1	input	N/A	yes	Carry, zero, and negative flags taken from the ALU after an arithmetic operation
wr_data_alu	8	input	N/A	yes	Data bus taken from the result of the ALU
wr_data_mem	8	input	N/A	yes	Data bus taken from the data memory after a RD operation (forwards this incoming data back to the register file)
io_sram_sel	1	input	SW15	yes	The IO to SW[15] of the top level is directly fed into the FSM and is used to drive an internal mux switching between the SRAM output and IO Register
sram_addr_SW	8	input	SW[7:0]	yes	SW[7:0] which select which address to access in SRAM Data
rr_addr_get	8	input	N/A	yes	The written-back address taken from the register file rr. FSM forwards this address to the SRAM during program execution.
sram_addr_sel	8	output	N/A	yes	The port used to forward the address to the SRAM Data. During the program, this address comes from

					Rr in the register file. Afterward, driven by SW[7:0]
wr_en_reg_file	1	output	N/A	yes	Control signal to trigger a write to the register file when appropriate
wr_en_data_mem	1	output	N/A	yes	Control signal to trigger a write to the SRAM Data memory when appropriate
wr_addr	2	output	N/A	yes	2-bit address that controls which register to be accessed (for read/write purposes)
wr_data	8	output	N/A	yes	8-bit data that is to be written into the register file
opcode	8	output	N/A	yes	Forwards the full 8-bit instruction to the ALU when an arithmetic operation is decoded
rda	2	output	N/A	yes	Goes into the register file, controls which register data gets at one of the outputs of the register file
rdb	2	output	N/A	yes	Goes into the register file, controls which register data gets at one of the outputs of the register file
inc	1	output	N/A	yes	Increment signal that is connected to the program counter’s inc input. Controls increments.
jump	1	output	N/A	yes	Jump signal that is connected to the program counter’s jump input. Controls jumps.
call	1	output	N/A	yes	Call signal that is connected to the program counter’s call input. Controls calls
ret	1	output	N/A	yes	Return signal that is connected to the program counter’s ret input. Controls returns.

***NOTE:** **All other output ports are unused in the top-level cputop.v block and thus are not included in the table above.** Again, they are not included in this table since they serve no functional purpose for the FSM besides acting as a built-in ILA during simulation. They are commented out in the testbench file.

2.3.1 fsm.sv: Internal Control Flags

There are a couple internal control flags used in the design that directly affect the behavior of the FSM. They are listed below:

LD_flag
ALU_flag
WR_flag
WRIO_flag
RD_flag
JUMP_flag
JUMPC_flag
CALL_flag
RET_flag

Each flag is triggered by their respective instruction opcode during the decode state. For example, if an ADD operation is to be performed, the ALU_flag is set HIGH. By using these flags, the FSM is able to determine the order of the states it must traverse in order to successfully complete the instruction.

Notice that in table 3, all other states are directly affected by the internal control flags, whilst decode takes the instruction as an input and sets the appropriate flag.

This space is intentionally left blank.

Table 3: Mealy Machine Behavior Description

Input control signal trigger conditions	**curr_state**	**next_state**	**output**
All control flags == 0	**idle**	**fetch**	inc, jump, call, ret = 0, All wr_en = 0,
LD_flag || WR_flag || WRIO_flag || RD_flag || JUMP_flag || JUMPC_flag || CALL_flag || RET_flag == 1	**idle**	**loadfsm**	inc, jump, call, ret = 0, All wr_en = 0,
No specific trigger signal required (some control flags can be on or off, it does not matter for this state)	**fetch**	**decode**	No output changed from previous state
Arithmetic instruction at the input is decoded (ADD, SUB, MULT, etc.)	**decode**	**execute**	opcode = instruction[7:0], rda = instruction[3:2], rbd = instruction[1:0], wr_addr = instruction[3:2]
LD, JUMP, JUMPC, CALL, RET, NOOP instruction at the input is decoded	**decode**	**advance**	opcode = instruction[7:0], wr_addr = instruction[3:2] (LD)
WR, WRIO, RD instruction at the input is decoded	**decode**	**idle**	opcode = instruction[7:0], rda = instruction[3:2] (WR), wr_addr = instruction[3:2] (RD), rbd = instruction[1:0]
BRK instruction at the input is decoded	**decode**	**terminate**	No output changed from previous state
LD_flag || ALU_flag || WR_flag || WRIO_flag || RD_flag == 1	**loadfsm**	**write**	Wr_data = instruction[7:0], inc = 0
All control flags == 0	**loadfsm**	**advance**	inc = 0
JUMP_flag || JUMPC_flag || CALL_flag || RET_flag == 1	**loadfsm**	**advance2**	jump = 1 or call = 1 or ret = 1 (depends on instruction)
No specific trigger signal required	**execute**	**loadfsm**	No output changed from previous state

No specific trigger signal required	**write**	**advance**	wr_en_data_mem = 1 or Wr_en_reg_file =1 (depends on instruction),
No specific trigger signal required	**advance**	**idle**	inc = 1
No specific trigger signal required	**advance2**	**idle**	Inc = 0, Jump = 0, Call = 0, Ret = 0,
BRK instruction at the input is decoded	**terminate**	**terminate**	No output changed from previous state

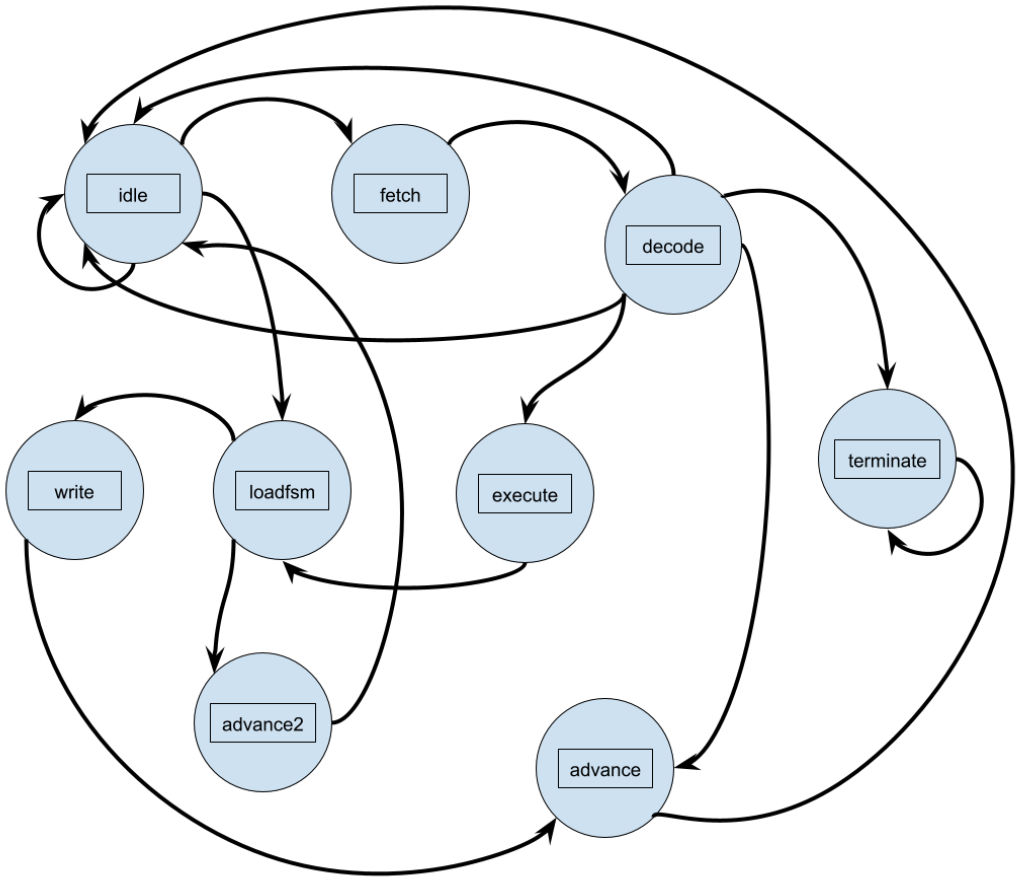

Figure 1: Flow Diagram of Mealy FSM (Please refer to Table 3 for input and curr_state conditions that affect state flow)

3 Testing Procedure and Results

3.1 Block Diagram and Simulation

In order to synthesize and implement all units onto the FPGA hardware, a block diagram was created, which was given an HDL wrapper. Below is the full block diagram including all appropriate port names to be utilized on the FPGA.

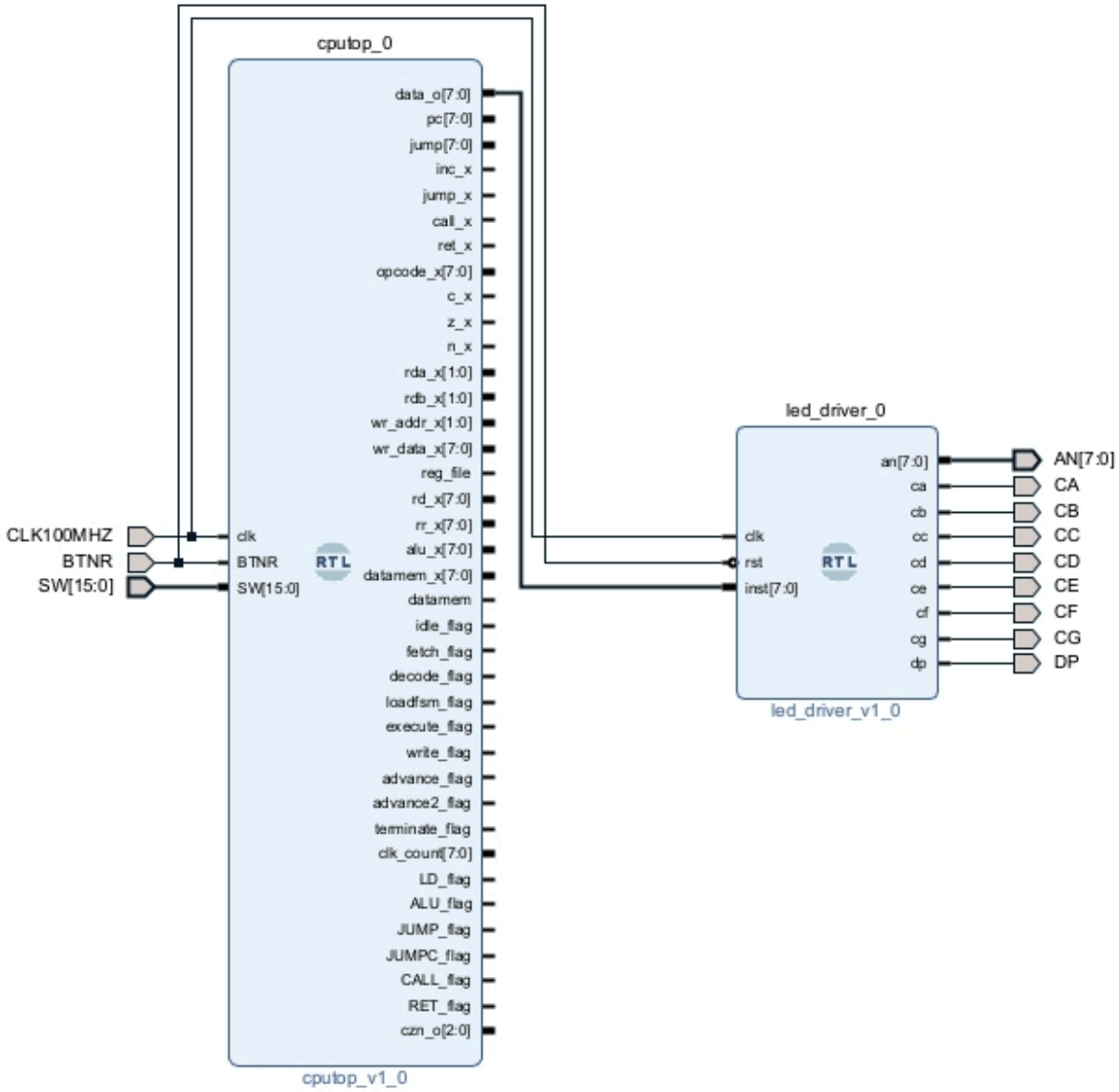

Figure 2: Block Diagram of Project 4; Top Level

Additionally, below are the RTL-Level Block Diagrams for cputop.v. Technically, these are post-synthesis diagrams, but for the sake of context, they are included here.

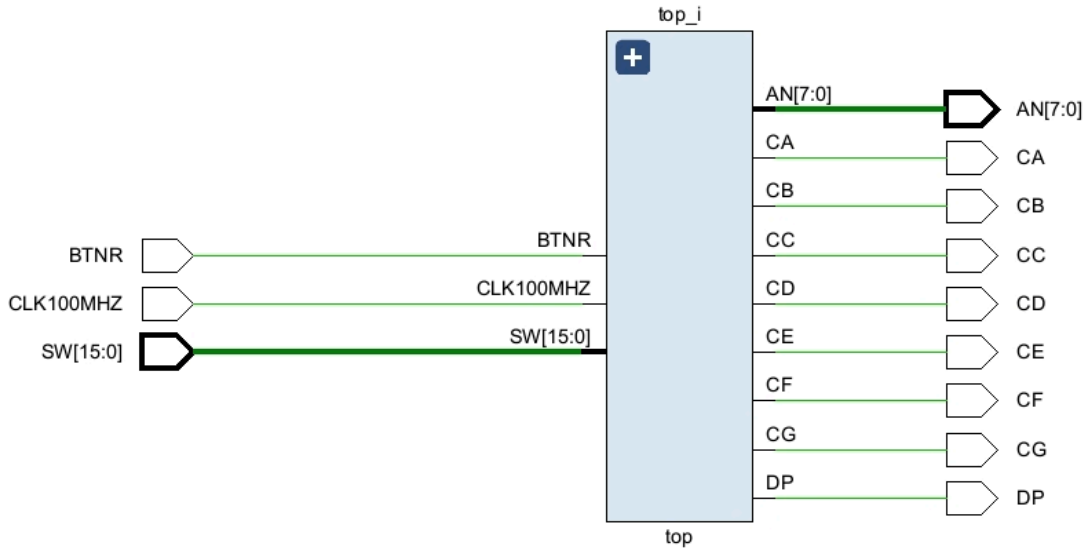

Figure 3: RTL Block Diagram of Project 4; Top Level

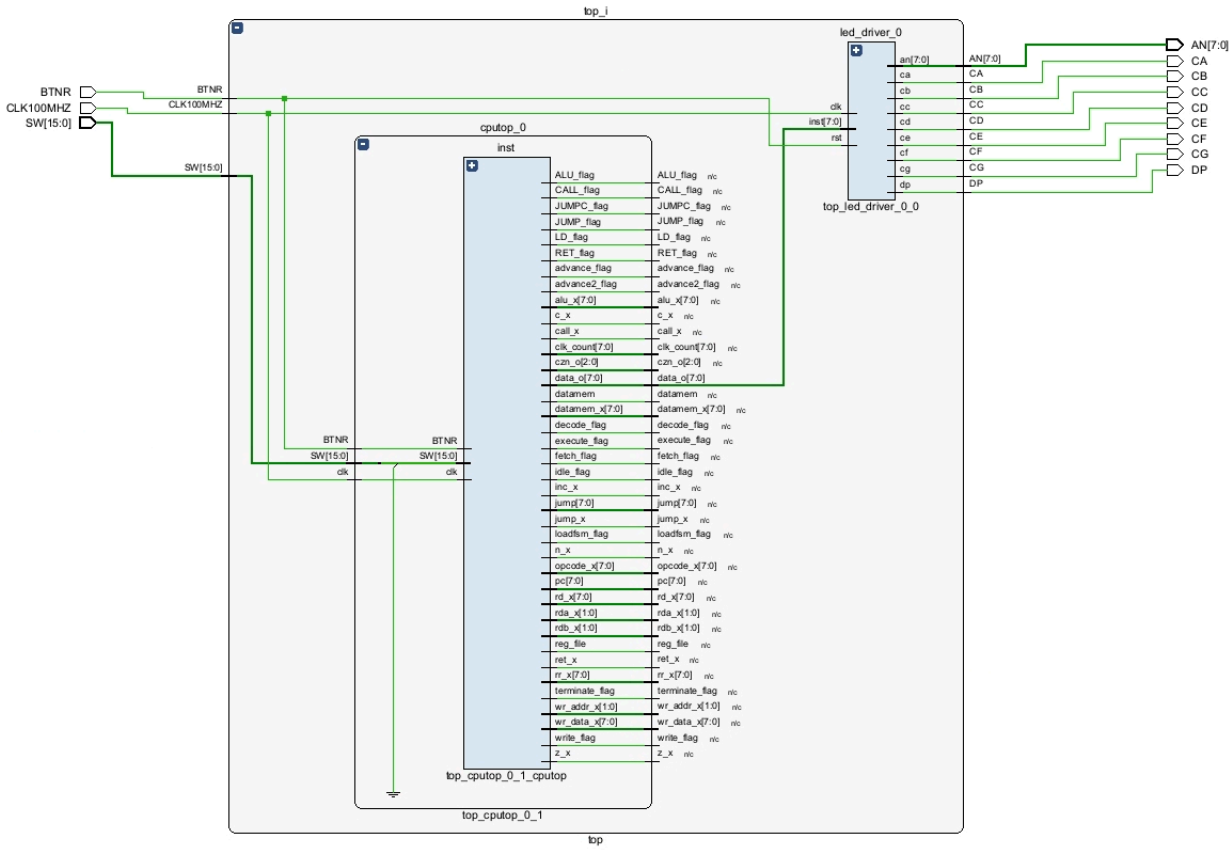

Figure 4: RTL Block Diagram of Project 4; Expanded

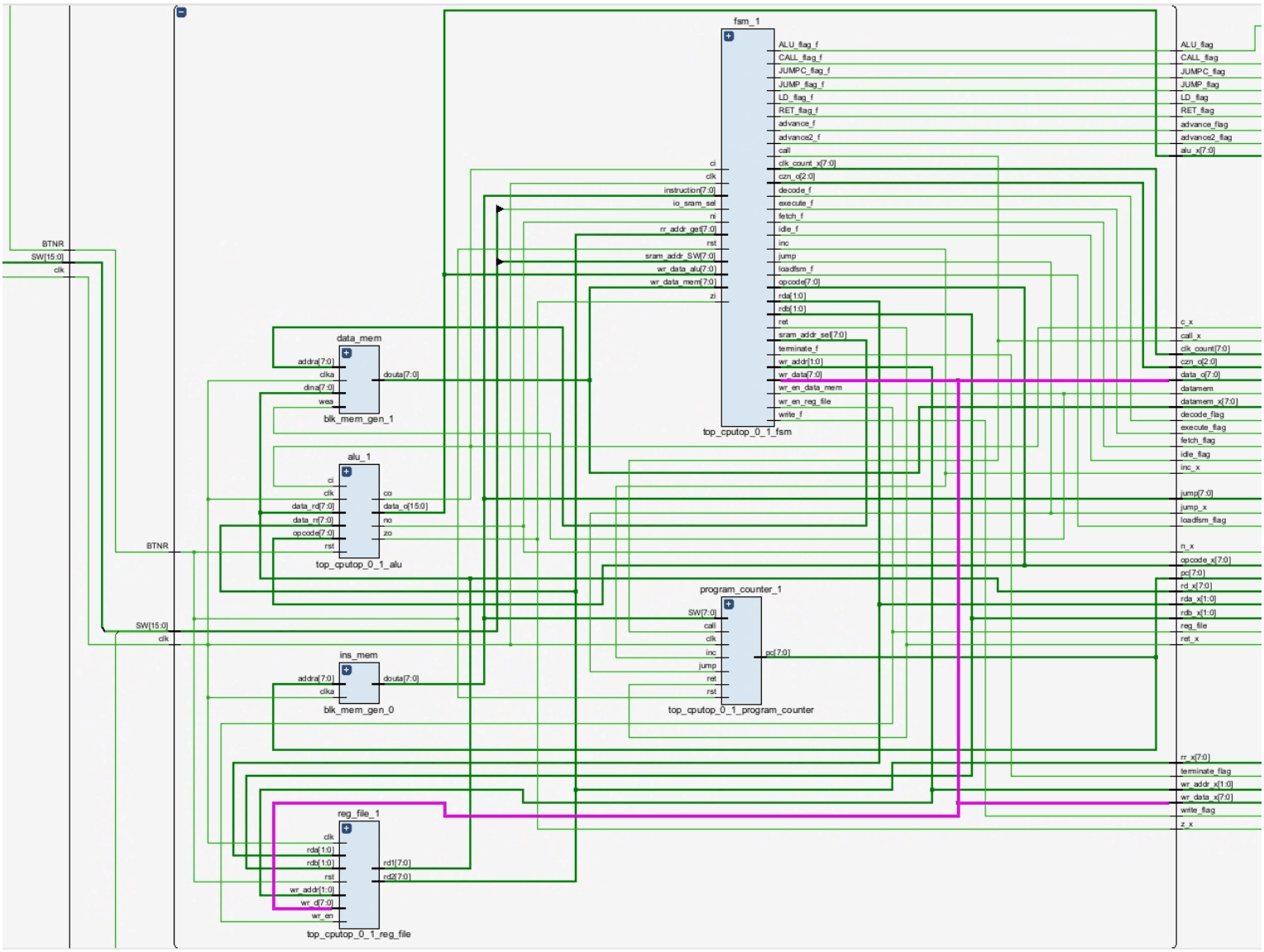

Figure 6: Internal RTL Block Diagram of cputop.v (port/wire labels included)

Note that the data_o signal is highlighted in magenta. This is the only output port that is connected to led_driver.v. That is, data_o is the only output of the cputop.v module that is utilized at the top level design (Please refer to higher level diagrams above for the full path of all signals from cputop.v).

The cputop.v module testbench (**cputop_tb.sv**) is written to execute all instructions in instruction memory twice: initially, and then once more after a reset to the CPU. The testbench then self-checks that the contents of the IO register and all SRAM Data Addresses are correct after the program execution, by comparing to known values. Below are screenshots of the results, with an accompanying explanation of what is being checked.

SW[15] = Control between reading IO Register and SRAM Data
SW[7:0] = Selecting which SRAM Data address to read from

Ex: Read from IO register => SW = 16'b0000000000000000 = 16'h0000
Ex: Read from SRAM, Address x03 => SW = 16'b1000000000000011 = 16'h8003

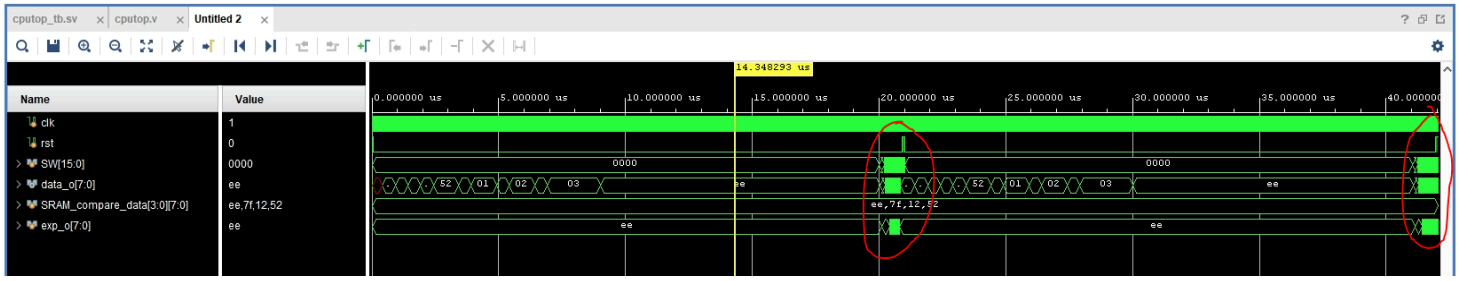

Figure 7: Simulation view of cputop_tb.sv (Full Waveform)

This is a demonstration of the CPU running for the full time. A red circle is drawn on the waveform, demarcating the end of a single run-through of the instructions, followed by a reset. Likewise, it can be seen that 2 runs are performed in order to test that the CPU can restart and still write the correct data in memory.

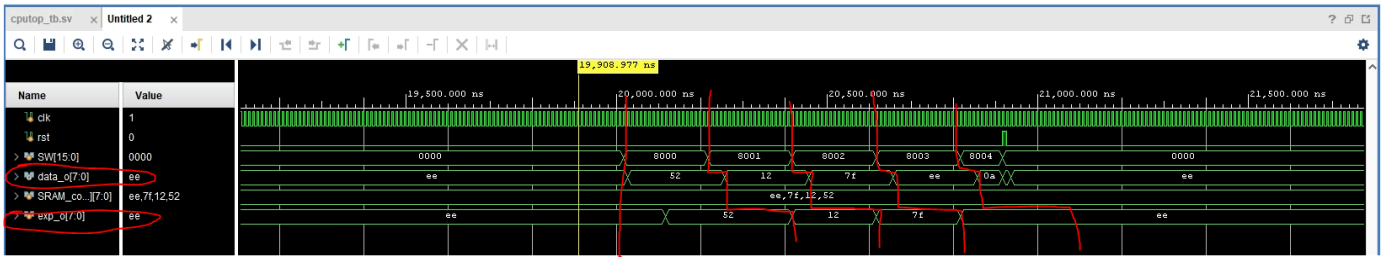

Figure 8: Simulation view of cputop_tb.sv (Zoomed in, first run-through)

When zooming into the end of the simulation run time, the testbench checks if the correct data in the IO register and SRAM Data memory is written in, and that both memories can be accessed by switching SW[15] (that change happens at t = 20000ns). Recall at SW[15] = 0 means data_o is read from the IO Register and SW[15] = 1 reads from the SRAM data.

Prior to the switch, the value of xEE is being read from the IO register which is expected. When SW[15] => 1, the data from data_o is read from the SRAM address as x52, which is the expected value of SRAM Data at address 0. The rest of the values for the SRAM Data that were written to, are also evaluated.

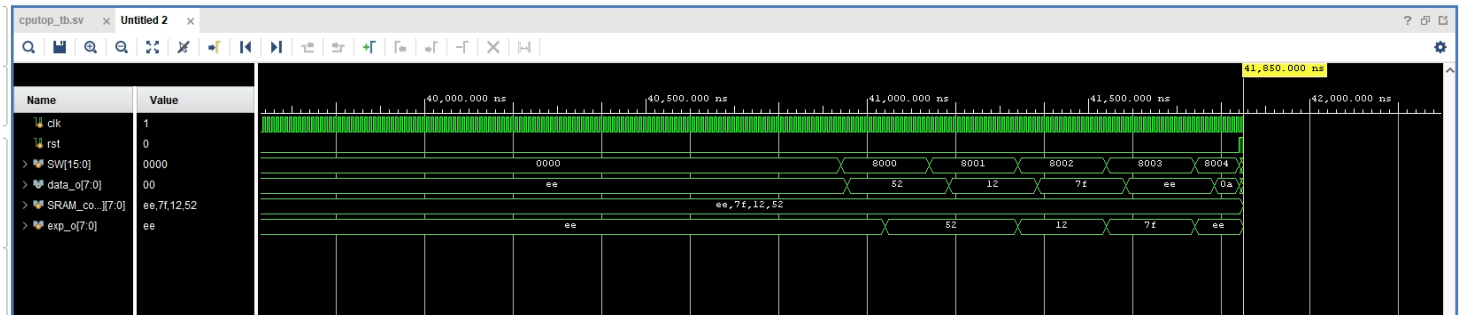

Figure 9: Simulation view of cputop_tb.sv (Zoomed in, second run through)

Similar to the first run-through, the IO Register and SRAM data contents are checked and compared with expected values. This also ensures that the instructions in the CPU are replicable, without any data interfering from previous runs.

```
run all
IO Register output SUCCESS=> Data OUT: 11101110
SRAM Data output SUCCESSS @ 20215000 => Data @ Address: 00: 52
SRAM Data output SUCCESSS @ 20415000 => Data @ Address: 01: 12
SRAM Data output SUCCESSS @ 20615000 => Data @ Address: 02: 7f
SRAM Data output SUCCESSS @ 20815000 => Data @ Address: 03: ee
resetting CPU @ t=20915000
IO Register output SUCCESS=> Data OUT: 11101110
SRAM Data output SUCCESSS @ 41230000 => Data @ Address: 00: 52
SRAM Data output SUCCESSS @ 41430000 => Data @ Address: 01: 12
SRAM Data output SUCCESSS @ 41630000 => Data @ Address: 02: 7f
SRAM Data output SUCCESSS @ 41830000 => Data @ Address: 03: ee
resetting CPU @ t=41930000
$finish called at time : 42030 ns : File "C:/Users/lolph/Desktop/cmpe316projects/project_4_cpu/project_4_cpu.srcs/sim_1/new/cputop_tb.sv" Line 202
```

Figure 10: TCL Console output of cputop_tb.sv

This space is intentionally left blank.

3.2 Synthesis and Implementation

"Generate Bitstream" was run to synthesize and implement the program. **Unfortunately, the final implementation of the CPU could not be realized in hardware.** A video demonstration shows that the data memory was not being written to after programming the FPGA and all SRAM addresses / IO Register contents were still 8'b0.

In simulation, it was seen that the processor works as intended, all instructions/opcodes executed as expected. All critical warnings and errors were quelled during this phase of development. The final design raises no critical warnings besides those related to the constraint file and .coe files.

Due to limited time, the project was submitted with a successful simulation and synthesis, but unrealized FPGA functionality. **I speculate that this was a result of timing issues when the program would attempt to write to the register file, and then to the SRAM or potentially even an issue with the synthesis tool, although unlikely.** It was possible that the setup time for the data being written to the register file was inadequate, and thus the data did not write properly in reality, but appeared fine in simulation. However, **a fix could not be procured in time.**

The WNS reported by Vivado was > 2.688. While not infringing on the rule that WNS must be >= 0, it *might have been improved as a possible solution to the hardware problem*. Attempts were made to create a new project and copy over all the Verilog modules, but that proved unsuccessful as well.

Tcl Console | Messages | Log | Reports | **Design Runs** | IP Status

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP	Start	Elap
synth_1 (active)	**constrs_1**	**synth_design Complete!**												**0**	**0**	**0**	**0**	**0**	**12/9/24, 11:26 PM**	**00:**
impl_1	constrs_1	write_bitstream Complete!	2.688	0.000	0.106	0.000		0.000	0.128	0	88 Warn			362	241	1	0	0	12/9/24, 11:29 PM	00:
Out-of-Context Module Runs																				
blk_mem_gen_0_synth_1	blk_mem_gen_0	synth_design Complete!												0	0	0.5	0	0	12/6/24, 9:56 PM	00:
top		Submodule Runs Complete																	12/9/24, 10:08 PM	01:
blk_mem_gen_1		Using cached IP results																		

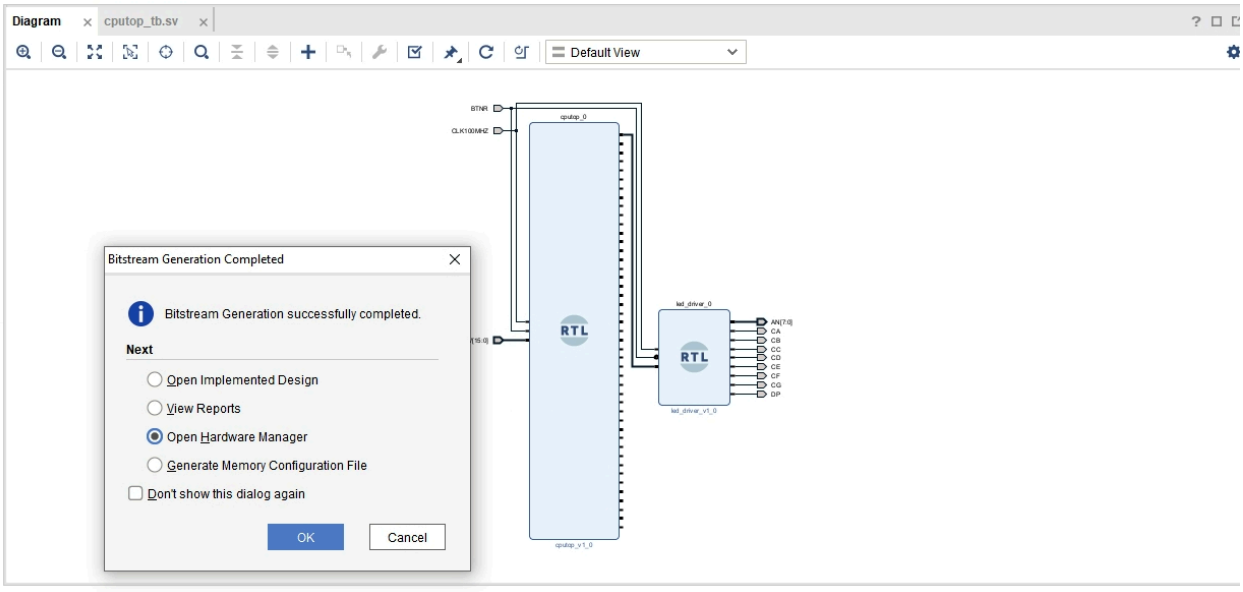

Figure 11: Generate Bitstream Success

For each test, a “$display()” prompt is sent to the TCL Console to inform the designer if any and all tests were successful. The simulation will automatically pause via “$stop” should an error occur to allow for debugging and analysis.

A compressed .zip file containing the project directory is included with the project submission.

Note that files for led_driver.v, .xdc, .coe are not included, as the path for those files were not local to the directory.

A PDF of this report is also submitted.