## Introduction

SLC-3 processor is a computational unit with a 16-bit wide data width and 16-bit addressability. SLC-3 can perform basic arithmetic (addition) and logical operations (bitwise AND, NOT). SLC-3 is also integrated with a memory unit to store and load values as well as sequence of instructions (mini-programs such as bubble sorting a list of values). On top of other things (such as the ALU, the general-purpose registers etc.), SLC-3 also has a branching unit that enables conditional checks within its sequence of instructions.

SLC-3 operates on instruction codes, which are passed into the control unit (the 'brain' of SLC-3 architecture). The control unit sends out control signals based on the instruction code to the rest of the data path. The inputs and outputs of SLC-3 are memory-mapped. They come from the switches, LED and display segments of the FPGA board. Switches determines which value PC is set to, LEDs serves as visual cue for user clarity and display segments can be used to display either the outputs (of an operation) or the PC, or the IR per the user's needs.

## Written Description and Diagrams of SLC-3

1. Summary of operations
2. How SLC-3 performs its functions

On a high level, SLC-3 must complete 3 sub-processes to complete a full operation. The 3 subprocesses are respectively – fetch, decode and execute.

<u>The three high level sub-processes (Fetch, Decode, Execute)</u>

The first subprocess is "Fetch", which represents fetching an instruction code from a PC-specified memory location into the instruction register. In terms of the memory architecture of the SLC-3, this means PC is loaded into the MAR register, where PC serves as a pointer to which we want to start our program. After PC is loaded into MAR, PC itself is incremented by 1.

To retrieve the memory pointed by MAR (if MAR is a valid address greater than 0 and less than 0xFFFF), we must first instantiate SRAM on the physical FPGA board. For this lab, the memory contents within the SRAM are readily provided to us. The retrieved memory is loaded into the MDR register. This value stored in MDR register will then be passed into the instruction register. This marks the end of the fetching instruction.

In terms of assembly-like pseudocode, "Fetch" does the following:

```
MAR ←PC; PC ← PC+1

MDR ← M(MAR)

IR ← MDR
```

The second subprocess is "Decode", which is breaking down the instruction value in the instruction register into its opcode and operands.

There is no generic pseudocode to generalize the operations in the Decode phase, since there are many different operations, and they assign parameters within the IR different from each other.

The third subprocess is "Execute", which is the actual performing of specific instruction opcode (the leading 4 bits in the IR). Depending on the instructions, there could be multiple states or just a single state of execution. The specifics of how each hardware should perform is dependent on the control signals produced by the control unit (the finite state machine) of SLC-3.

3. Block diagram of slc3.sv

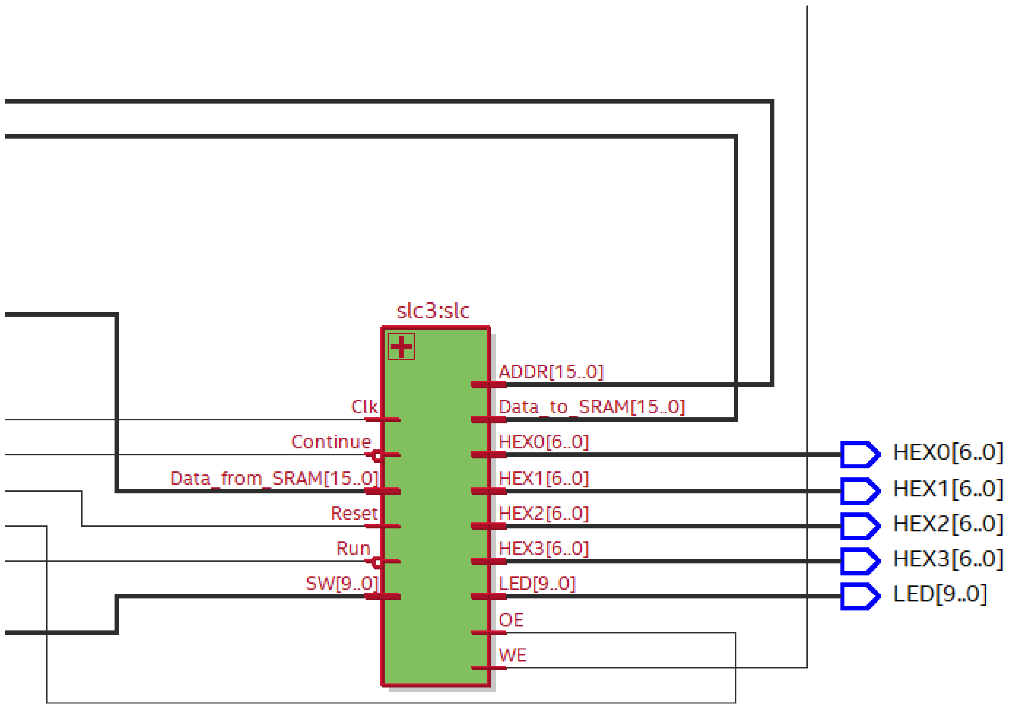
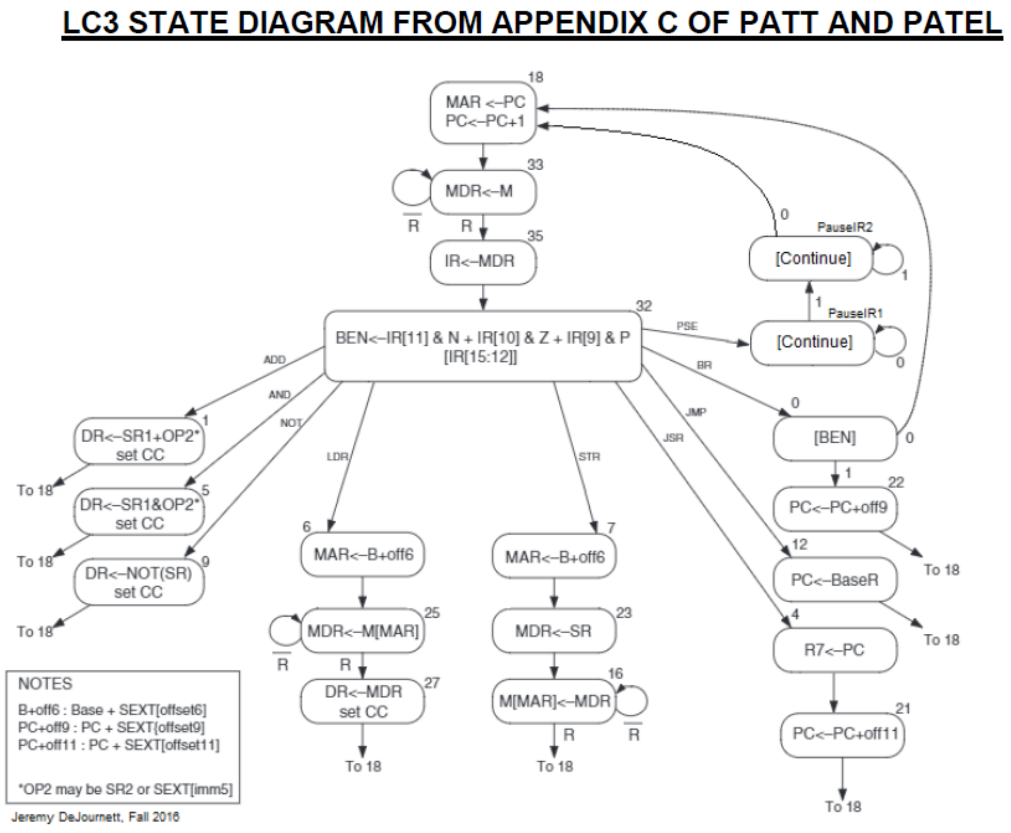


*Figure shows the block diagram of slc3.sv using Quartus Netlist RTL Viewer.*

State diagram for SLC-3 control unit:



5. Written description of all System Verilog modules

| | |
|---|---|
| Module: Gate_decoder.sv<br><br>Inputs: GateMARMUX, GateALU, GatePC, GateMDR<br>Outputs: GateSelect[1:0] | Description: Takes in 4 signals that represents which register/path is driving the data bus in SLC3. The 4 output signals get decoded into a 2-bit output signal. The 2-bit output signal is used to differentiate out of the 4 possible paths that can drive the data bus.<br><br>Purpose: The outputs of these decoders will serve as the selects for a 4-to-1 select mux to determine which output path drives the data bus. Mainly serve as a subcomponent to replace the lack of tristate buffer on the FPGA board. |
| Module: test_memory.sv<br><br>Inputs: Reset, Clk, [15:0] Data, [9:0] address, rden, wren<br>Outputs: [15:0] readout | Description:<br>Takes in a data input stream, a address to be read/written, as well as control signals read enable and write enable. Depending on the control signals, perform reads or writes correspondingly to the memory. Readout will be the content read |

| | |
|---|---|
| | from memory when a read is desired and read enable is HIGH.<br><br>Purpose: Acts as the SRAM for SLC-3 when running on simulation. Synthesized as blank during actual FPGA board usage. |
| Module: slc3_testtop.sv<br><br>Inputs: [9:0] SW, Clk, Run, Continue<br><br>Outputs: [9:0] LED, [6:0] Hex0-3 | Description: Takes in user input such as Switches, Run and Continue to control the execution flow of the SLC-3. Clk serves as the global clock signal for the SLC-3. Outputs depending on the program execution is displayed accordingly on the LEDs, and Hex displays.<br><br>Purpose: Top level module for SLC-3 to be compiled and run for simulation purposes. Encloses the slc3.sv module as well as the test_memory.sv but does not instantiate real SRAM on the board. |
| Module: slc3_sramtop.sv<br><br>Inputs: [9:0] SW, Clk, Run, Continue<br><br>Outputs: [9:0] LED, [6:0] Hex0-3 | Description: Takes in user input such as Switches, Run and Continue to control the execution flow of the SLC-3. Clk serves as the global clock signal for the SLC-3. Outputs depending on the program execution is displayed accordingly on the LEDs, and Hex displays.<br><br>Purpose: Mainly serve the same purpose as slc3_testtop.sv but this works for actual synthesizing onto the board. Because of interacting with the board, this module includes some other modules that setup actual SRAM on the board. |
| Module: SLC3_2.sv<br><br>Inputs/Outputs: N/A | Description: -<br><br>Purpose: |
| Module: slc3.sv<br><br>Inputs: [9:0] Sw, Clk, Reset, Run, Continue, [15:0] Data_from_SRAM<br><br>Outputs: [9:0] LED, OE, WE, [6:0] Hex0-3, [15:0] Addr, [15:0] Data_to_SRAM | Description: Takes in the input from the upper level module (sramtop or testtop.sv) but now directly use them onto operating the SLC-3 processor. Also takes in Data_from_SRAM from the outer module to be used within the memory path of SLC3.<br><br>The outputs are fed from here onto the outer level modules, which then connect to the actual FPGA board. |

| | |
|---|---|
| | Purpose: Serves as the module for the actual SLC-3 architecture without considering whether actual SRAM needed to be instantiated (isolates difference between simulation and FPGA board). |
| Module: memory_contents.sv<br><br>Inputs/Outputs: N/A | Description: -<br><br>Purpose: Contains the instruction sequences on each memory location. Instruction sequences combine to make mini programs to serve as test cases when user jump to the correct PC. |
| Module: Mem2IO.sv<br><br>Inputs: Clk, Reset, [15:0] Addr, OE, WE, [9:0] Switches, [15:0] Data_from_CPU, Data_from_SRAM<br><br>Outputs: [15:0] Data_to_CPU, [15:0]Data_to_SRAM, [3:0] Hex0-3 | Description:<br>Takes in two control signals OE and WE, a memory address ADDR, as well as Data_from_CPU and Data_from_SRAM.<br><br>Transfer accessed memory between the SRAM, CPU as well as the input switches and output displays.<br><br>Purpose: Serves as a bridge for memory mapping to both the SRAM (physical memory) as well as the memory mapped I/O. The bridge main purpose of existence is because the memory-mapped IO architecture for the switches and hex displays requires that the memory address from CPU can be referring to the physical memory but also can refer to the IO devices. |
| Module: ISDU.sv<br><br>Inputs: Clk, Reset, Run, Continue, [3:0] Opcode, IR5, IR11, BEN,<br><br>Outputs:<br>LD_MAR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED<br>GatePC, GateMDR, GateALU, GateMarMux<br>PCMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, ADDR2MUX, ALUK, MemOE, MemWE | Description:<br>Takes in the global clock and input button triggers (user input) and decoded first four bits of the instruction that represents the opcode. IR5, IR11 and distinguishable bits in the Instruction manual for SLC3. BEN is the branching conditional Boolean.<br><br>Based on the opcode, sets up the executional flow of the instruction and its required control signals to be fed into the corresponding hardware modules.<br><br>Purpose:<br>Acts as a finite state machine for the SLC-3 processor to enclose the entire fetch-decode-execute operation of the SLC-3. Acts as the control path that determines the SLC-3 behavior for all components. |
| Module: InstantiateRAM.sv | Description: |

| | |
|---|---|
| Inputs: Reset, Clk<br><br>Outputs: [15:0] Addr, wren, data | Takes in the control signal write enable, an input data stream, and an address to be written to and write the input data stream to SRAM if wren allows.<br><br>Purpose:<br>Initializes the SRAM for the physical on-chip memory when running the SLC-3 on the FPGA board. |
| Module: PC_register.sv<br><br>Inputs: Clk, Reset, LD_PC, [15:0] Data_in<br><br>Outputs: [15:0] Data_Out, Next_PC | Description:<br>Register that takes in a clock and an asynchronous reset signal. Load signal comes from ISDU to determine whether to load the register.<br><br>Purpose: |
| Module: datapath.sv<br><br>Inputs: Clk, Reset, Run, LD_IR, LD_MAR, LD_PC, LD_MDR, LD_CC, LD_BEN, LD_REG, MIO_EN, [15:0]data_in, [15:0] Data_to_CPU, [1:0] pcmux_select, sr1mux_select, drmux_select, [1:0] addr2mux_select, addr1mux_select, sr2mux_select, [1:0] ALUK_select,<br><br>Outputs:<br>BEN_Out, [15:0] IR_Out, MAR_Out,<br>[15:0] MARMUX_Out, ALU_Out, PC_Out, MDR_Out | Description:<br>Takes in the different control signals from ISDU and feed them into different modules including Muxes, registers etc. to determine their outputs/behavior.<br><br>Produces 4 outputs on the 4 different paths that will later be determined by the control gate signals which will drive the data path.<br><br>Purpose:<br>Serves as the data bus that transfers the data between the data unit and the memory unit. Encloses the mini components that performs the ALU, the branching logic and other internals of the SLC3. |
| Module: fourtoone_mux.sv<br><br>Inputs: [1:0] select, [15:0] input0-3<br><br>Outputs: [15:0] muxout | Description:<br>Based on the select, produce the corresponding 16-bit wide output based on combinational logic<br><br>Purpose:<br>Acts like any four-to-one mux, is used in multiple instances for different purposes in the SLC3 structure. |
| Module: MDR_register.sv<br><br>Inputs: Clk, Reset, LD_MDR, MIO_EN, [15:0] Data_in, Data_to_CPU | Description:<br>Register that takes in a clock and an asynchronous reset signal. Load signal comes from ISDU to determine whether to load the register. |

| | |
|---|---|
| Outputs: [15:0] Data_Out | Purpose:<br>Represents the memory retrieved register that stores the value retrieved from memory mapped IO or from physical RAM. |
| Module: HexDriver.sv<br><br>Inputs: [3:0] In0<br>Outputs: [6:0] Out0 | Description: Takes in a 4-bit binary number and outputs its corresponding hexadecimal value on a 7-segment LED display setup using switch case statements.<br><br>Purpose: To light up the correct LED segments to show a 4-bit binary number as a single hex number on the LED display segment. |
| Module: testbench.sv<br><br>Inputs:<br>N/A<br><br>Outputs:<br>N/A | Description:<br>N/A<br><br>Purpose:<br>a self-written test file to be run on ModelSim to assist in debugging and verification process of design. |
| Module: twotoone_mux.sv<br><br>Inputs:select, [2:0] input0-1<br><br>Outputs: [2:0] muxout | Description:<br>Acts like any 2-to-1 mux with a 3-bit wide inputs and outputs.<br><br>Purpose:<br>Reused multiple times across the SLC-3 structure but mainly in the register unit portion of the structure. |
| Module: register_unit.sv<br><br>Inputs: Clk, LD_REG, Reset,<br>[2:0] sr2in, [2:0] sr1in, [2:0] drmux,<br>[15:0] Data_In<br><br>Outputs: [15:0] sr1out, [15:0] sr2out | Description:<br>A packed module that encloses the eight general purpose registers. Takes in the control signals to determine which registers to load into (and whether load should be done), and which register to be written.<br><br>Purpose:<br>Represent the register file for the SLC-3 structure, has a packed array within the module to represent the eight general purpose registers used in SLC-3. |
| Module: twotoone_mux16.sv<br><br>Inputs: select, [15:0] input0-1<br><br>Outputs: [15:0] output_mux | Description:<br>Serves like any two-to-one mux but with a 16-bit wide data_out.<br><br>Purpose:<br>Used for ADDR1 Mux |

| | |
|---|---|
| Module: ALU.sv<br><br>Inputs: [1:0] ALUK, [15:0] sr1_in, [15:0] sr2_in<br><br>Outputs:<br>[15:0] ALU_Out | Description:<br>Takes in two 16 bit inputs from the register file (or some sign extended from the IR) and perform simple arithmetic/logic operation on the two inputs. Output is passed onto the data bus whenever gateALU is high.<br><br>Purpose:<br>Serves as the ALU unit to perform ADD, AND, Not and PASS instruction in SLC3. |
| Module: SEXT.sv<br><br>Inputs: [4:0] slice_base, slice5p1, [2:0] slice6p3, [1:0] slice9p2<br><br>Outputs: [15:0] sext16 | Description:<br>Takes in different lengths of input bit strings and sign extend them to 16 bits.<br><br>Purpose:<br>Used in multiple occasions in the SLC-3 architecture because different arguments take different lengths for sign extend to 16 bits i.e. some are sign extend 4-bit string to 16, some are 6-bit and so on. This generic function handles all cases. |
| Module: BranchLogic.sv<br><br>Inputs:<br>[15:0] data_in<br><br>Outputs: n, z, p | Description:<br>Given an input data stream that is 16-bit wide, determine whether the number is positive, negative or 0.<br><br>Purpose:<br>Serves as condition checking for branch purposes. |
| Module: nzp_reg.sv<br><br>Inputs:<br>Clk, Reset, LD_CC, n_in, z_in, p_in<br><br>Outputs:<br>N, z, p | Description:<br>Register that takes in a clock and an asynchronous reset signal. Load signal comes from ISDU to determine whether to load the register.<br><br>Purpose:<br>Register that acts as a signifier to the control unit whenever the n,z,p flags are set to be relevant in the instruction flow. |
| Module: instr_branch.sv<br><br>Inputs: [2:0] IR_br, n_in, z_in, p_in<br><br>Outputs: branch_bool | Description:<br>Extract the information from the IR register bits to compare with the nzp flag to determine whether branching is legal.<br><br>Purpose:<br>Compute the Boolean logic for branching. |
| Module: ben_reg.sv | Description: |

| | |
|---|---|
| Inputs: Clk, Reset, ben_in, LD_BEN<br><br>Outputs: ben_out | Register that takes in a clock and an asynchronous reset signal. Load signal comes from ISDU to determine whether to load the register.<br><br>Purpose:<br>Register that is relevant when branching is enabled. If enabled, the signal is relevant to the control unit; else, the output of this register does not have relevance. |

**Simulations of SLC-3 Instructions**

a. Simulate the completion of all 6 test programs, I/O Test 1, I/O Test 2, Self-Modifying Code, XOR, Multiplier and Sort.

***Important notes for all simulations description:

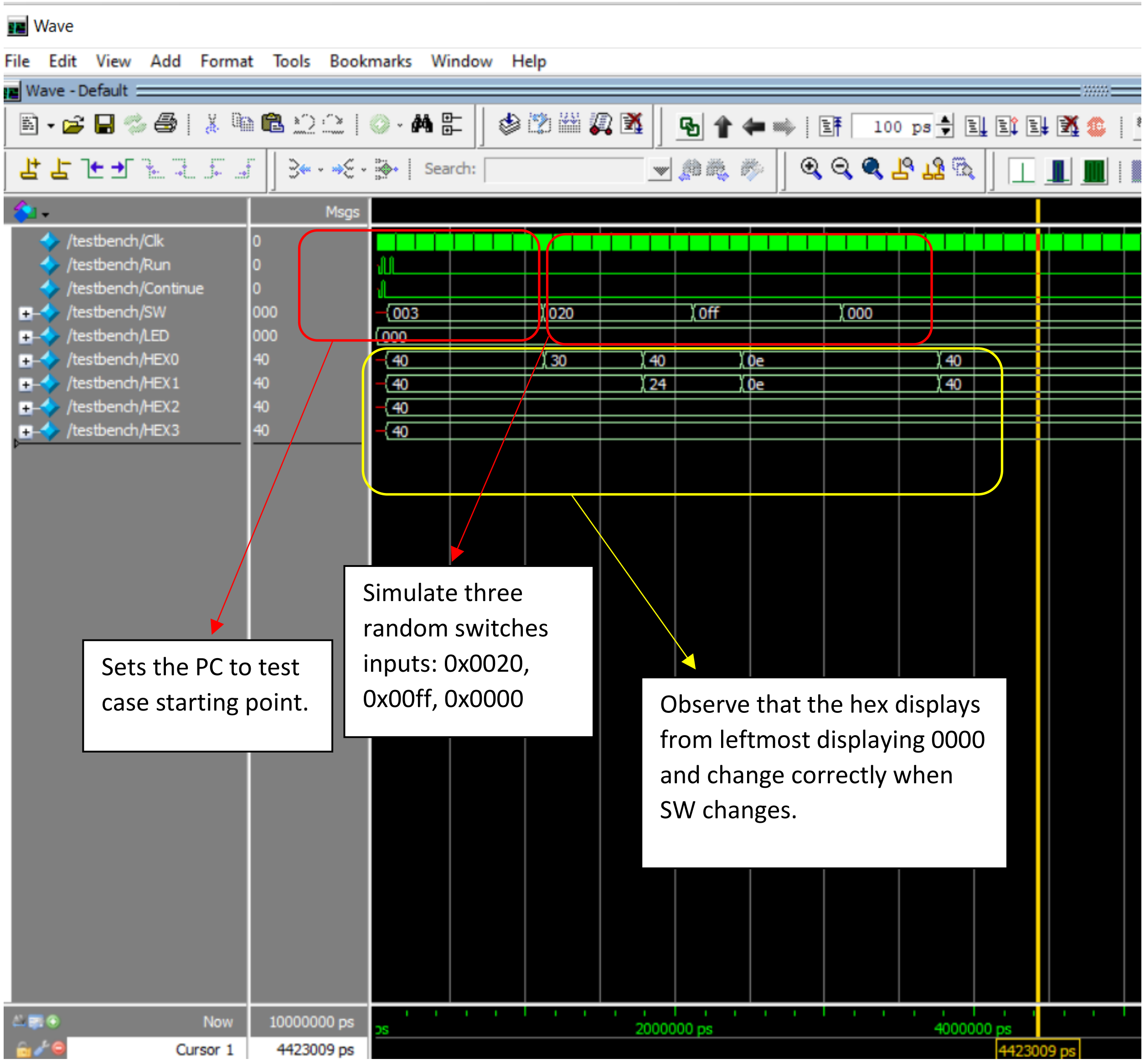
- Many of the outputs used to verify test cases' correctness are on the hex displays, so it is important to understand what each hex display code physically represents.

| Hex display code (hexadecimal value) | Integer displayed | Hex display code (hexadecimal value) | Integer displayed |
|---|---|---|---|
| 0x40 | 0 | 0x00 | 8 |
| 0x79 | 1 | 0x10 | 9 |
| 0x24 | 2 | 0x08 | A |
| 0x30 | 3 | 0x03 | b |
| 0x19 | 4 | 0x46 | C |
| 0x12 | 5 | 0x21 | d |
| 0x02 | 6 | 0x06 | E |
| 0x78 | 7 | 0x0e | F |

- Each test case is annotated on a separate page for clarity purposes.

<u>Test 1 (starting PC = 0x0003)</u>

Expected behaviour is for hex displays to change accordingly to user input on switches.



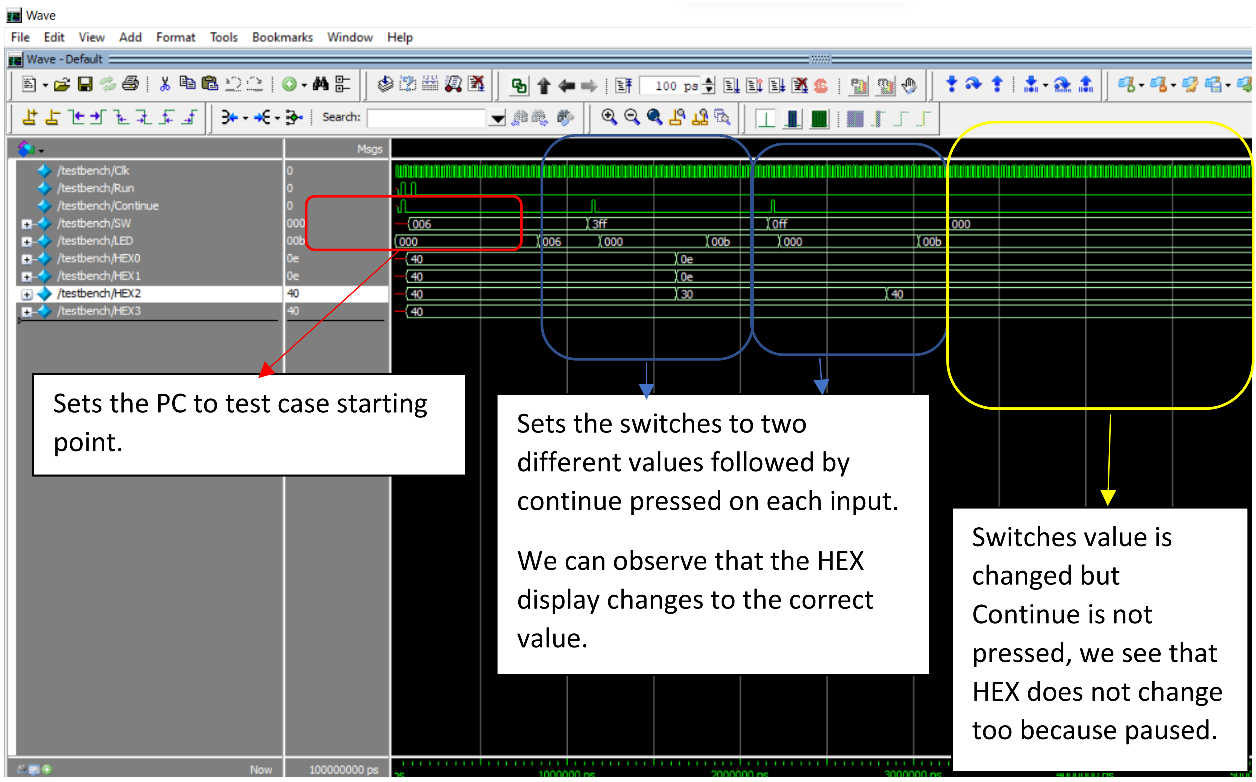
Sets the PC to test case starting point.

Simulate three random switches inputs: 0x0020, 0x00ff, 0x0000

Observe that the hex displays from leftmost displaying 0000 and change correctly when SW changes.

Test 2 (starting PC = 0x0006)

Expected behaviour is for hex displays to change accordingly to user input on switches, but this time hex displays only change when “Continue” is pressed. If “continue” is not pressed, indefinite pause is expected.



Sets the PC to test case starting point.

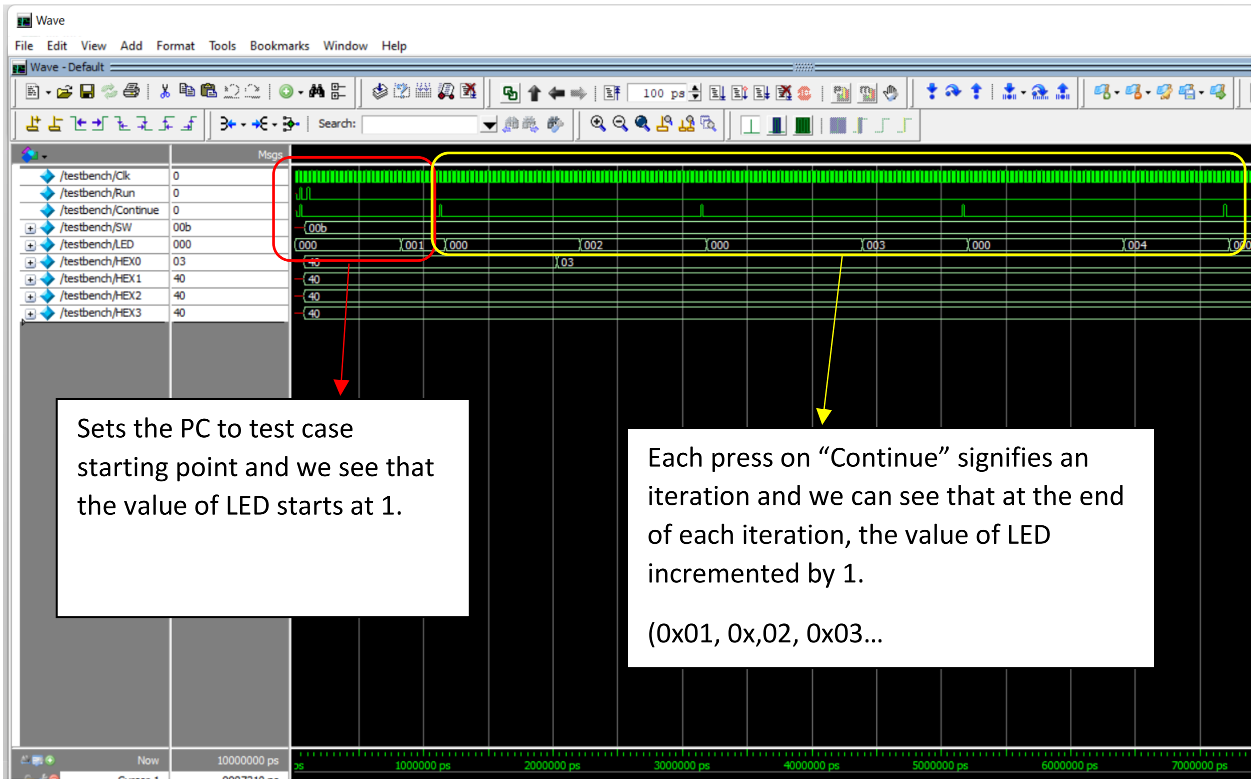
Sets the switches to two different values followed by continue pressed on each input.

We can observe that the HEX display changes to the correct value.

Switches value is changed but Continue is not pressed, we see that HEX does not change too because paused.

Test 3 (starting PC = 0x000B)

Expected behaviour is for program to be executing in a loop. With each press of "Continue", the value displayed on the LEDs should increment by 1.



```
/testbench/Clk       0
/testbench/Run       0
/testbench/Continue  0
/testbench/SW        00b
/testbench/LED       000
/testbench/HEX0      03
/testbench/HEX1      40
/testbench/HEX2      40
/testbench/HEX3      40
```

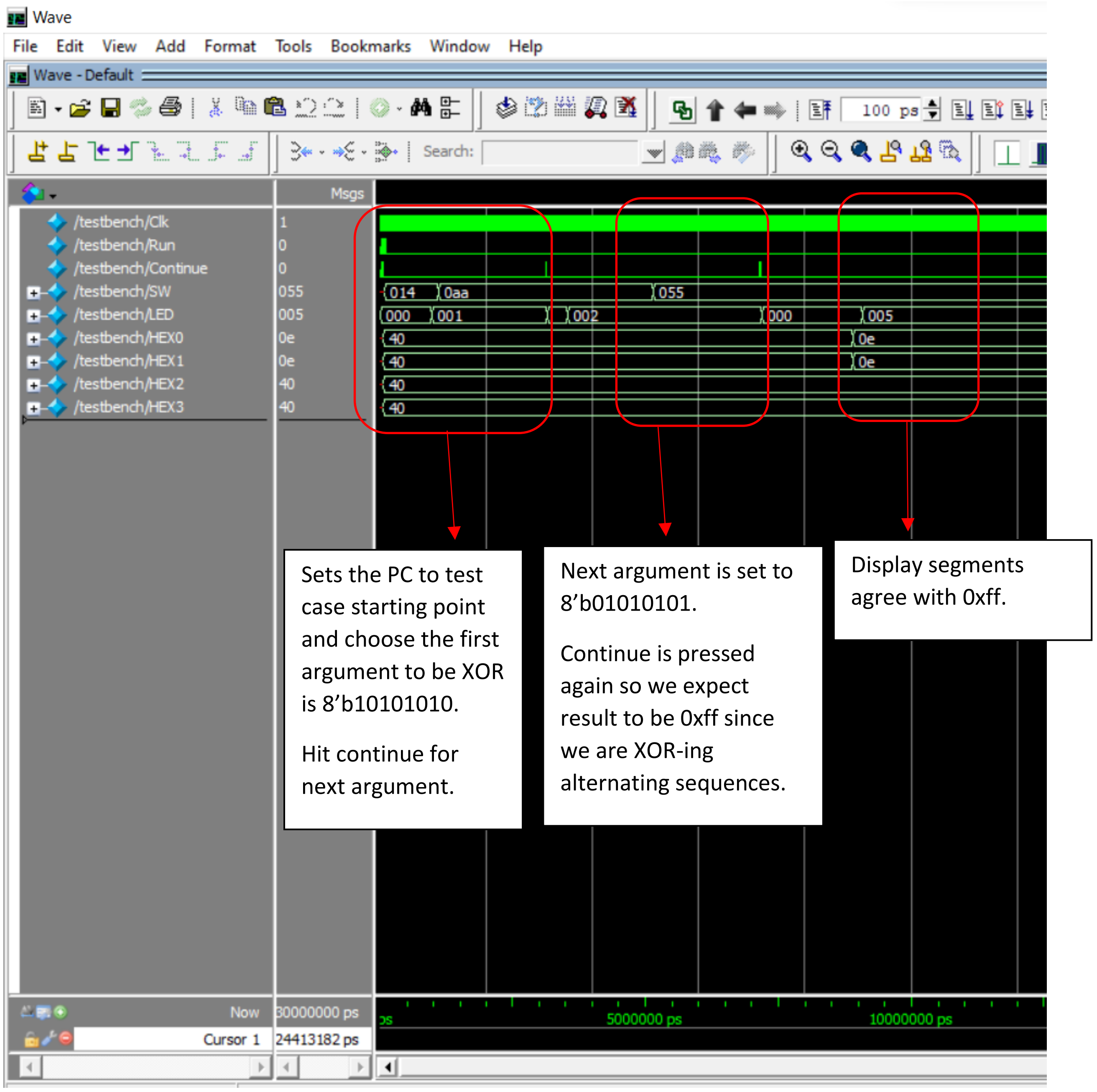
Sets the PC to test case starting point and we see that the value of LED starts at 1.

Each press on "Continue" signifies an iteration and we can see that at the end of each iteration, the value of LED incremented by 1.

(0x01, 0x,02, 0x03…

Test 4 XOR (starting PC = 0x0014)

Expected behaviour is for program to be executing in a loop. We input two 8-bit numbers and XOR them together. Output is displayed in hexadecimal display segments.



```
/testbench/Clk        1
/testbench/Run        0
/testbench/Continue   0
/testbench/SW         055   014 0aa 055
/testbench/LED        005   000 001 002 000 005
/testbench/HEX0       0e    40 0e
/testbench/HEX1       0e    40 0e
/testbench/HEX2       40    40
/testbench/HEX3       40    40
Now       30000000 ps
Cursor 1  24413182 ps
```

Sets the PC to test case starting point and choose the first argument to be XOR is 8'b10101010.

Hit continue for next argument.

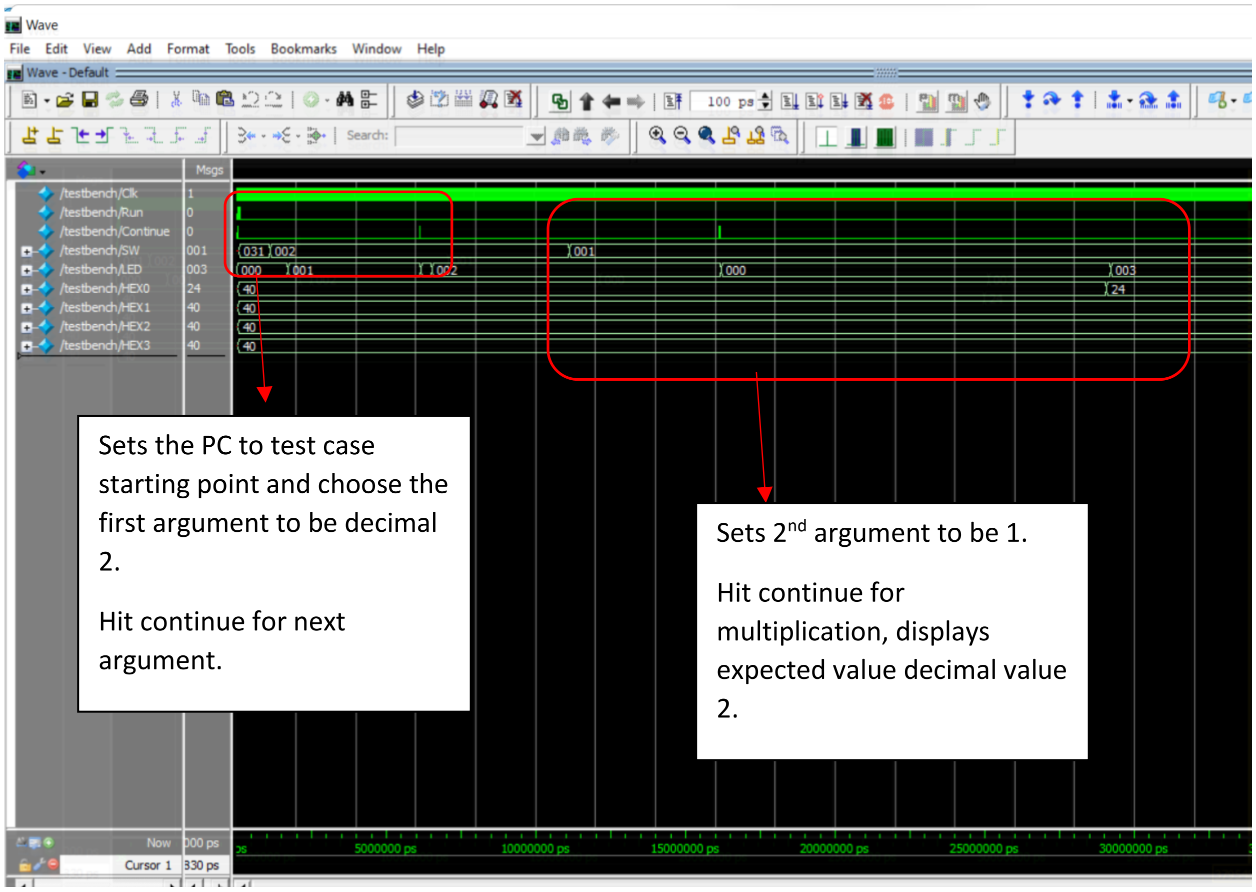
Next argument is set to 8'b01010101.

Continue is pressed again so we expect result to be 0xff since we are XOR-ing alternating sequences.

Display segments agree with 0xff.

## Test 5 Multiplication (starting PC = 0x0031)

Like XOR but perform multiplication on the two operands.



Sets the PC to test case starting point and choose the first argument to be decimal 2.

Hit continue for next argument.

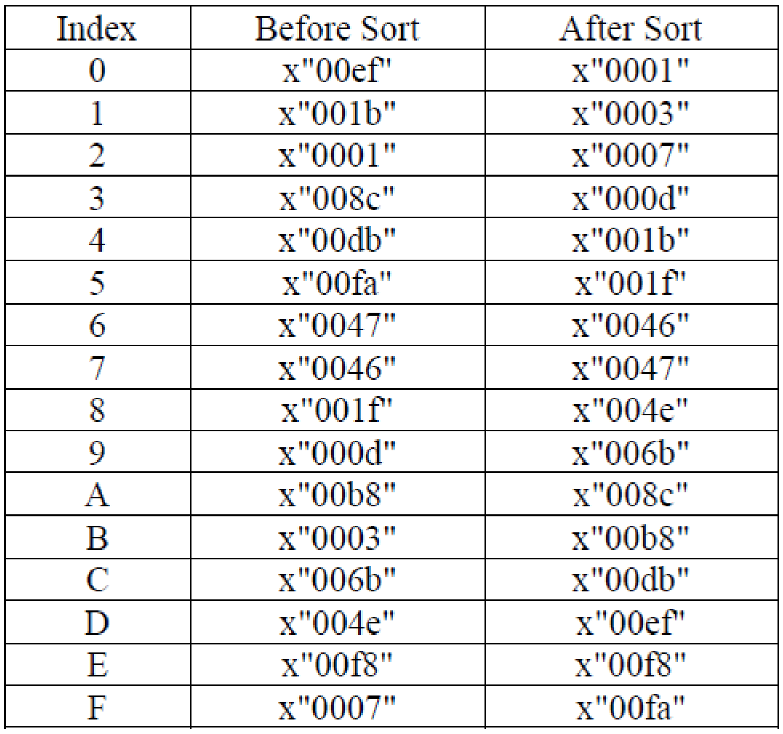
Sets 2nd argument to be 1.

Hit continue for multiplication, displays expected value decimal value 2.

Test 6: Sort (starting PC = 0x005A)

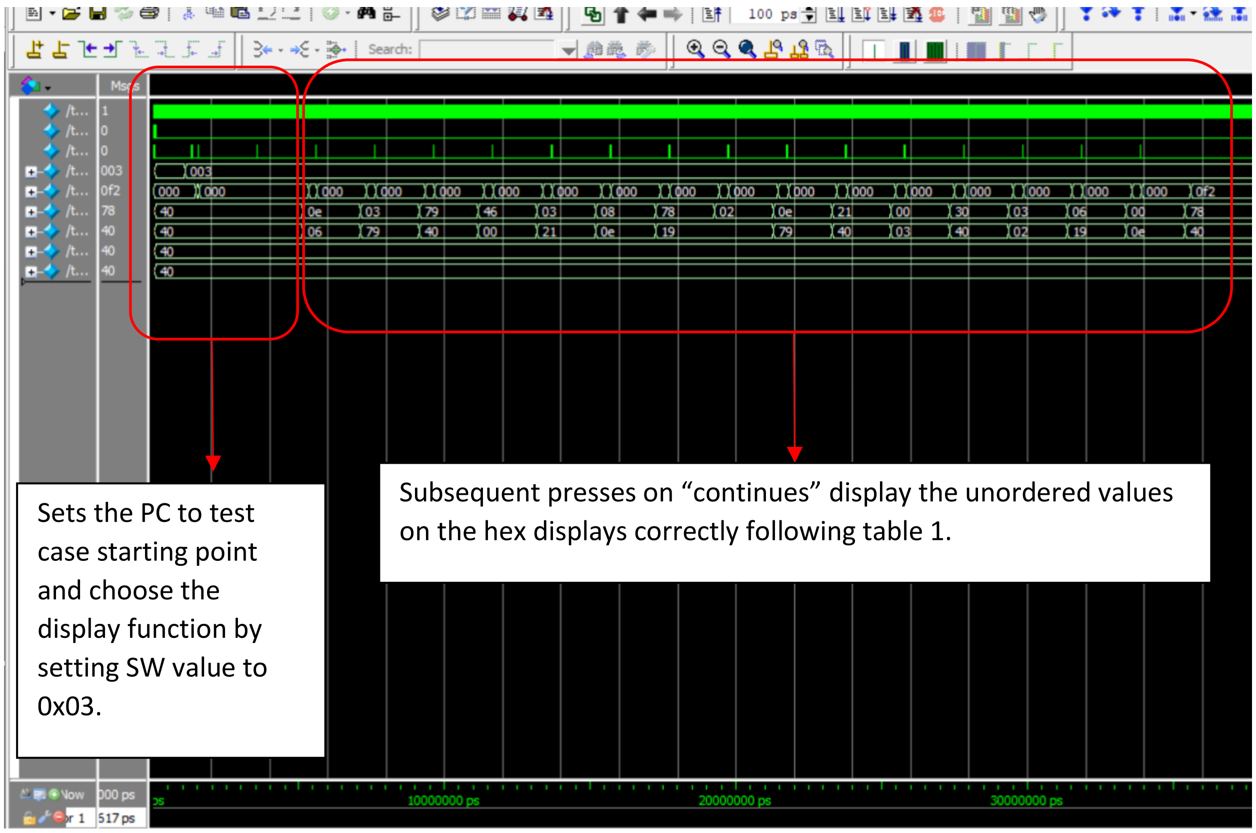
This test is organized into four parts. The first part is a menu, containing function calls to the other three parts that are executed based on input. The menu contains a single pause instruction,

Entering x0001 will call the “data entry” function, entering x0002 will call the “sort” function, and entering x0003 will call the “display” function. Any other value will simply cause the menu to loop back to the start without doing anything. We test by running display, sort then display again and compare with the expected sorted result. A list of 16 unordered numbers is provided to us as below:



| Index | Before Sort | After Sort |
|---|---|---|
| 0 | x"00ef" | x"0001" |
| 1 | x"001b" | x"0003" |
| 2 | x"0001" | x"0007" |
| 3 | x"008c" | x"000d" |
| 4 | x"00db" | x"001b" |
| 5 | x"00fa" | x"001f" |
| 6 | x"0047" | x"0046" |
| 7 | x"0046" | x"0047" |
| 8 | x"001f" | x"004e" |
| 9 | x"000d" | x"006b" |
| A | x"00b8" | x"008c" |
| B | x"0003" | x"00b8" |
| C | x"006b" | x"00db" |
| D | x"004e" | x"00ef" |
| E | x"00f8" | x"00f8" |
| F | x"0007" | x"00fa" |

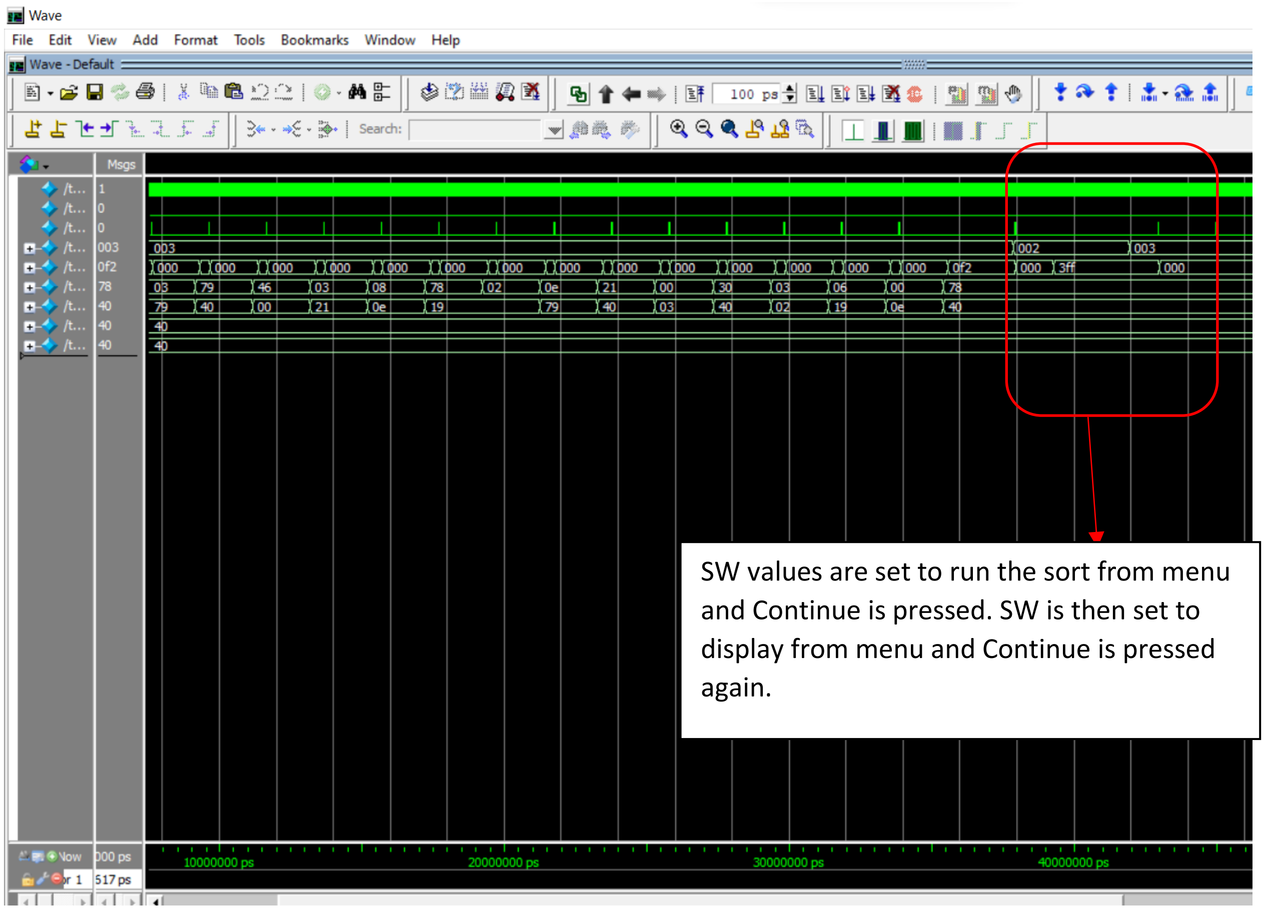
Table 1: Sample list values before and after sorting



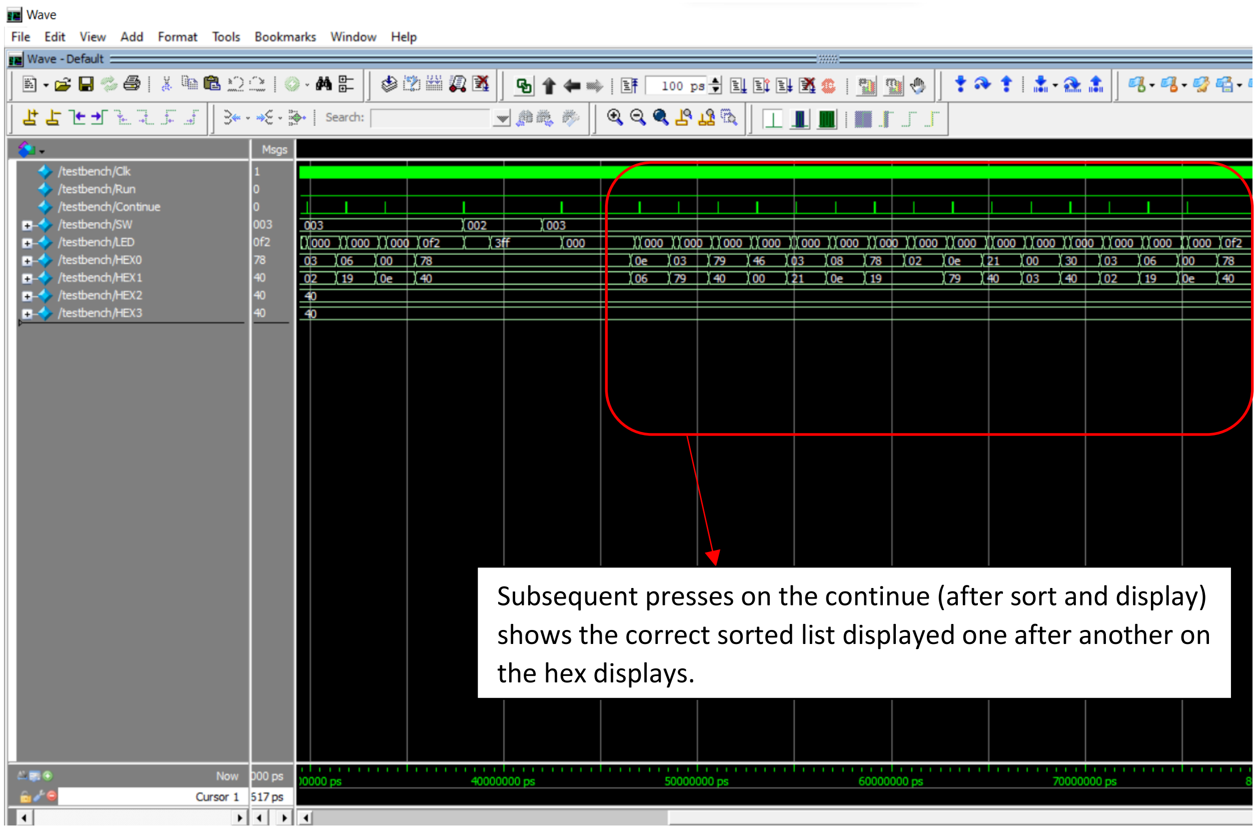
Sets the PC to test case starting point and choose the display function by setting SW value to 0x03.

Subsequent presses on "continues" display the unordered values on the hex displays correctly following table 1.

Note: Please see last snapshot of simulation to see the naming of the signals. Most importantly, four signals from bottom up represent Hex3 – Hex0 respectively.



SW values are set to run the sort from menu and Continue is pressed. SW is then set to display from menu and Continue is pressed again.



Subsequent presses on the continue (after sort and display) shows the correct sorted list displayed one after another on the hex displays.

**Design Resource and Statistics**

| | |
|---|---|
| LUT = numbers of *combinational with no register* and *combinational with register* | 801 |
| DSP | 0 |
| Memory (BRAM) = Total block memory implementation bits | 18432 |
| Flip-flop | 259 |
| Frequency | 68.52 MHz |
| Static Power | 90.00 mW |
| Dynamic Power | 10.53 mW |
| Total Power | 111.31 mW |

**Bug log**

| | |
|---|---|
| Description:<br>Some outputs of the muxes that I use output "no cares" at instances where they should give determinate output, causing wrong behavior of the processor. | Resolve:<br>Removed unnecessary always_comb blocks within the muxes that assign no cares to the output values.<br><br>I thought that once I assigned all possible output-combinations for a mux, the extra always_comb will not matter. |
| Description:<br>Failing tests whenever branching logic is required. | Resolve:<br>Corrected a careless bug where I compared the wrong flags for n, z, p values. I compared the n_flag to the bit that corresponds to p and vice versa. |

**Post-lab Questions**

1. What is MEM2IO used for, what's its main function?

   The input output devices we used in this lab: the switches and the hexadecimal displays are memory-mapped IO. Because these IO devices are memory mapped, certain memory addresses correspond to writing to/reading from the external hardware devices. If the addresses involved in read/write is within the IO devices range, then MEM2IO will serve as the interface to transfer data to/from the external devices to/from the data bus.

2. What is the difference between the JMP and BR instruction?

JMP executes unconditional branching. In other words, JMP directly takes the value that is being stored in the base register and set PC to be equal to that value. No validation checking is required.

BR checks the Boolean logic of the nzp flags. The user program can determine which flags are required for validation and if any of those flags satisfy the validation, then only the "branch" will be executed.

3. What is the purpose of the R signal in Patt and Patel? How do we compensate for the lack of the signal in our design? What implications does this have for synchronization?

The purpose of the R signal is to account for potential slow memory access (read/write) to the CPU. In our design, we did not use such R signal. Instead, we force the state machine to go through additional dummy states (multiple same states across several clock cycles) to ensure the correctness of memory accesses.

With regard for synchronization, this will extend several clock cycles of memory read/write operations.

**Conclusion**

The functionality of my design works as intended and perfectly executes all the instructions of the SLC-3 processor and pass all the test cases. I don’t think this lab is unnecessarily difficult, it can in fact be harder. I honestly think our feedback doesn’t matter. Regardless of what we write here, the labs will just be what they would be next semester and so on. As such, my only feedback is remove the question in the conclusion section that asks about our feedback.