**ECE 385**

Spring 2021

Experiment #5

**Simple Computer SLC-3.2 in SystemVerilog**

Nathan Moskal and Daniel Marks

ABF – Friday

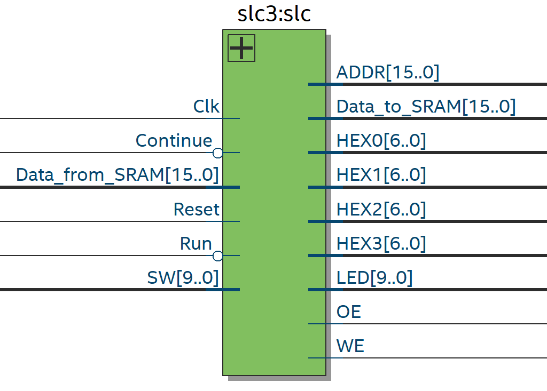
Abigail Wezelis & Harris Mohamed

**Part 1:** **Introduction:** In this lab, we built the SLC-3, a simple “computer”, in SystemVerilog. The computer operates in an FSM cycle of Fetch 🡪 Decode 🡪 Execute 🡪 Fetch 🡪 … In each cycle, the computer fetches an instruction from memory, decodes it with the Instruction Sequencer/Decoder (ISDU) Unit, then executes the instruction. The computer then recurses through the cycle continuously. The computer contains several modules that make up its structure: the aforementioned ISDU, the Program Counter (PC), Instruction Register (IR), Memory Address Register (MAR), Memory Data Register (MDR), Status Register (nzp), an 8x16-bit general data register file, and a 16-bit Arithmetic Logic Unit (ALU).

**Part 2: Written Description and Diagrams of SLC-3:**

1. **Further Summary of Operation:** The SLC-3 computer can perform several operations: ADD\*, AND\*, NOT, BR (Branch), JMP (Jump), JSR (Jump to Subroutine), LDR (Load Register), STR (Store Register), and PSE (Pause). ADD and AND have asterisks because they also have alternate forms ADDi and ANDi, which use immediate data from the command rather than register data for part of the operation. Which operation the computer performs is determined by the “opcode” of the instruction fetched from memory. In a FETCH cycle, the PC is stored into MAR.Then, the memory at MAR’s held address is stored to MDR, MDR’s value is passed to the IR, and PC is incremented by 1. In DECODE, the data from IR is passed to the ISDU. During the EXECUTE cycle, the ISDU uses the instruction from DECODE to perform an operation, and then writes the result to a register or memory location. Of the listed operations above, ADD, AND, NOT, and LDR set the Status Register. What that means is that after each of these operations, depending on whether the result is Negative, Zero, or Positive, the Status Register stores a 1 in either n, z, or p. When the system tries to execute a BR (Branch) statement, it will only branch if the Status Register’s value matches the statement. Branch commands are formatted as “0000nzpOffset[0:8]”, so 0000111x… would Branch if the result was negative, zero, or positive (so always). 0000101x… branches on nonzero values, etc. Other commands are simpler, as Branch is the only command that uses the Status Register. ADD, AND, and NOT take two register arguments, and output the result to a third register. ADDi and ANDi perform the same operation but with the immediate value from the command as the second operand. JSR stores the current Program Counter value into Register 7 and adds the 11-bit PCoffset to the PC. JMP is simpler: it copies an address from a register to the PC. Unlike Branch, which takes conditions, JSR and JMP will jump regardless of the status registers. JMP is often called on register 7 in order to return from a subroutine but can be used on any register to jump the PC. LDR and STR use the FPGA board’s memory to load data from (and store data to) the 8x16 register file. LDR stores into its destination register (DR) the data from the memory address at (BaseR + SEXT[[1]](#footnote-1)(offset6)), while STR stores the source register’s (SR) data to the address at (BaseR + SEXT(offset6)). In the LC-3 (not SLC) ISA, anytime something uses memory, that step stays until R is true, and R is a control signal that indicates the memory is ready for read/write operation. This control signal is not present in the SLC-3, so instead each operation that uses the memory instead waits at the read/write state for several cycles to hopefully ensure the data is utilized correctly. To answer the post-lab question on the topic, the implication this has for synchronization is that there is a possibility of an invalid read or write, which would have an unknown result. The data taken or stored from/to the memory can have an unpredictable value. The final operation is Pause, which takes a 12-bit LED vector as an argument after 1101 in the 16-bit instruction. Pause turns the first 3 Hex LEDs on the FPGA board to the value of ledVect12 and then waits until a continue signal is called.

**c. Block Diagram of SLC3.sv**

 Diagram, schematic

Description automatically generated I/O Pins

**e. Written Description of all .sv modules**

Module: Reg\_1.sv

Inputs: Clk, Reset, Load, D

Outputs: Data\_Out

Description: This is a single bit register. If Reset is high, then Data\_Out is set to zero. If Load is high, Data\_Out is set to D.

Purpose: To hold a single bit until another is loaded or it is reset

Module: Reg\_16.sv

Inputs: Clk, Reset, Load, [15:0] D

Outputs: [15:0] Data\_Out

Description: This is a 16-bit register which behaves similarly to Reg\_1, except that it holds 16 bits.

Purpose: To hold 16 bits of data until more is loaded or it is reset.

Module: Reg\_File.sv

Inputs: Clk, Reset, [15:0] D, [2:0] DR, [2:0] SR1, [2:0] SR2, LD\_REG

Outputs: [15:0] SR1\_OUT, [15:0] SR2\_OUT

Description: This is an 8x16-bit register, formed from 8 Reg\_16’s. Registers are numbered 0-7, and otherwise it too behaves like a scaled up version of Reg\_1, as Reg\_16 does.

Purpose: To hold 8 16-bit pieces of data until they are replaced or reset.

Module: full\_adder.sv

Inputs: A, B, Cin

Outputs: S, Cout

Description: This is a 1-bit full adder. It takes binary inputs A, B, and Cin, and outputs A + B to S, with Cin and Cout being the carry in and out of that operation.

Purpose: To add two 1-bit binary numbers with a carry bit.

Module: CLA.sv

Inputs: [3:0] A, [3:0] B, Cin

Outputs: [3:0] S, Cout, P, G

Description: This is a 4-bit Carry-Lookahead adder, meaning it uses propagate and generate bits to calculate carry ins and outs for the individual full\_adder.sv’s used in its construction, of which there are 4. These local P and G bits are used to calculate the CLA’s overall P and G output, which are used to string multiple CLA’s together.

Purpose: To add two 4-bit operands to a carry in bit, and be usable in a larger lookahead adder.

Module: adder\_16.sv

Inputs: [15:0] A, [15:0] B, Cin

Outputs: [15:0] S, Cout  
Description: This is a 4x4 hierarchical 16-bit adder, and it uses four 4-bit Carry Lookahead Adders (CLA.sv)’s to add two 16-bit operands. P and G bits from each CLA are used to calculate carry ins for each CLA, as well as Cout for the overall adder\_16 module.

Purpose: To add two 16-bit operands with a carry bit.

Module: ALU.sv

Inputs: [1:0] ALUK, [15:0] A, [15:0] B, Cin

Outputs: [15:0] out

Description: The ALU performs one of 4 operations on the inputs A and B depending on the value of ALUK. For 00, out will equal the output of A + B from an adder\_16.sv module with a carry in of 0. For 01, out will be A & B. For 10, out will be ~A. For 11, A is passed through and out will simply equal A.

Purpose: To perform logical operations on 2 16-bit operands.

Module: HexDriver.sv

Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: HexDriver converts a 4 bit binary number input into a 7 bit output that displays that number when passed into the Hex LEDs on the FPGA board.

Purpose: To turn binary inputs into LED-ready Hex outputs

Module: datapath.sv

Inputs: Reset, Clk, LD\_MAR, LD\_MDR, LD\_IR, LD\_BEN, LD\_CC, LD\_REG, LD\_PC, LD\_LED, GatePC, GateMDR, GateALU, GateMARMUX, [1:0] PCMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, [1:0] ADDR2MUX, [1:0] ALUK, MIO\_EN, [15:0] MDR\_In

Outputs: [15:0] MAR, [15:0] MDR, [15:0] PC, [15:0] IR, BEN, [9:0] LED

Description: datapath.sv maintains the SLC-3’s registers, such as the nzp Status Registers, the PC register, IR, MDR, MAR, and the 8x16 general register file. Datapath also contains one ALU and one adder\_16. The input signals are used to control loading of the registers and the MUXes connected to them. ALUK controls the ALU, MIO\_EN controls the MDR’s MUX, and MDR\_In is the input to MDR.

Purpose: To maintain registers used in the SLC-3, as well as one ALU and Adder.

Module: slc3.sv

Inputs: [9:0] SW, Clk, Reset, Run, Continue, [15:0] Data\_from\_SRAM

Outputs: [9:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [15:0] ADDR, [15:0] Data\_to\_SRAM, OE, WE

Description: slc3.sv is the top-level entity in Lab5, and contains the entire SLC-3 as well as the other modules needed to operate it such as RAM and and Button drivers. Within the SLC-3, there is one ISDU, one datapath, one MEM2IO, and 4 HexDrivers. All of the corresponding inputs and outputs of these sub-modules are connected, which is why the slc3 has a smaller number of outputs than its components. As stated previously, the SLC-3 operates on a cycle of FETCH – DECODE – EXECUTE.

Purpose: To connect all parts of the SLC-3 computer together that they may function.

Module: ISDU.sv

Inputs: Clk, Reset, Run, Continue, [3:0] Opcode, IR\_5, IR\_11, BEN

Outputs: LD\_MAR, LD\_MDR, LD\_IR, LD\_BEN, LD\_CC, LD\_REG, LD\_PC, LD\_LED, GatePC, GateMDR, GateALU, GateMARMUX, [1:0] PCMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, [1:0] ADDR2MUX, [1:0] ALUK, Mem\_OE, Mem\_WE

Description: The ISDU is the Instruction Sequencer and Decoder, meaning it receives instructions in the form of the [3:0] Opcode, uses a decoder to get the starting state from the Opcode, and then uses the FSM it contains to execute that decoded instruction. Based on the current state of the FSM, different combinations of the ISDU’s output control signals will be set to high, which will control the other parts of the SLC-3. The diagram for the state machine is below.

Purpose: To Sequence and Decode Instructions fetched from memory.

**g. State Diagram of ISDU**

Diagram

Description automatically generated

**Part 3: Simulations of SLC-3 Instructions:**

b. Annotations for the above simulations should include start of the test program, any user input (for example, entering the numbers in the multiplier test), and reading the expected result.

**Part 4: Post-Lab Questions:**

**a.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| LUT | DSP | BRAM | Flip-F | Freq | StatP | DynaP | TotP |
| 1656 | 0 | 958464 | 1973 | 96.36mHz | 89.98mW | 8.21mW | 107.02mW |

**b.** MEM2IO serves as the middle man between the SRAM and LC-3. If its inputted address is xFFFF and WE is high, the MEM2IO will write to the LEDs on the FPGA board. If a Load is performed at xFFFF, MEM2IO will load the data from the FPGA board switches. For any other address, it will operate on the SRAM, and Data\_to\_CPU will be set to Data\_from\_SRAM, and Data\_to\_SRAM will be set to Data\_from\_CPU.

**c.** The differences between BR and JMP lie in their conditionality and where they move the PC. JMP is unconditional, so when a JMP statement is reached, the PC will move to the address specified by the register listed in the JMP statement. BR is conditional: only if its nzp value ANDed with the NZP register values are nonzero (meaning that one of the three parameters match) will the BR statement execute, and then it increments the PC by an amount equal to the offset specified, rather than jump to a certain address.

**Part 5: Conclusion:**

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.

b. This lab procedure was fine, I would have liked a little bit more guidance on what is expected to be included for Part 3’s simulation waveforms.

1. Sign Extended [↑](#footnote-ref-1)