University of Illinois at Urbana-Champaign

ECE385
Digital Systems Laboratory

Lab5: Simple Computer SLC-3.2 in SystemVerlog

TA: Abigail Wezelis, Harris Mohamed

Yan Miao, Guangxun Zhai
yanmiao2, gzhai5

March 2021

Contents

1 Introduction

In this lab, we designed a simple microprocessor SLC-3 using the hardware language SystemVerilog. It is a simplified version of LC-3, which includes a 16-bit Program Counter(PC), 16-bit instructions, and 16-bit registers and several MUXs. With the ISA fully implemented, we can us the simple processor to carry out tasks like multiply, xor and bubble sort.

2 Written Description and Diagrams of SLC-3

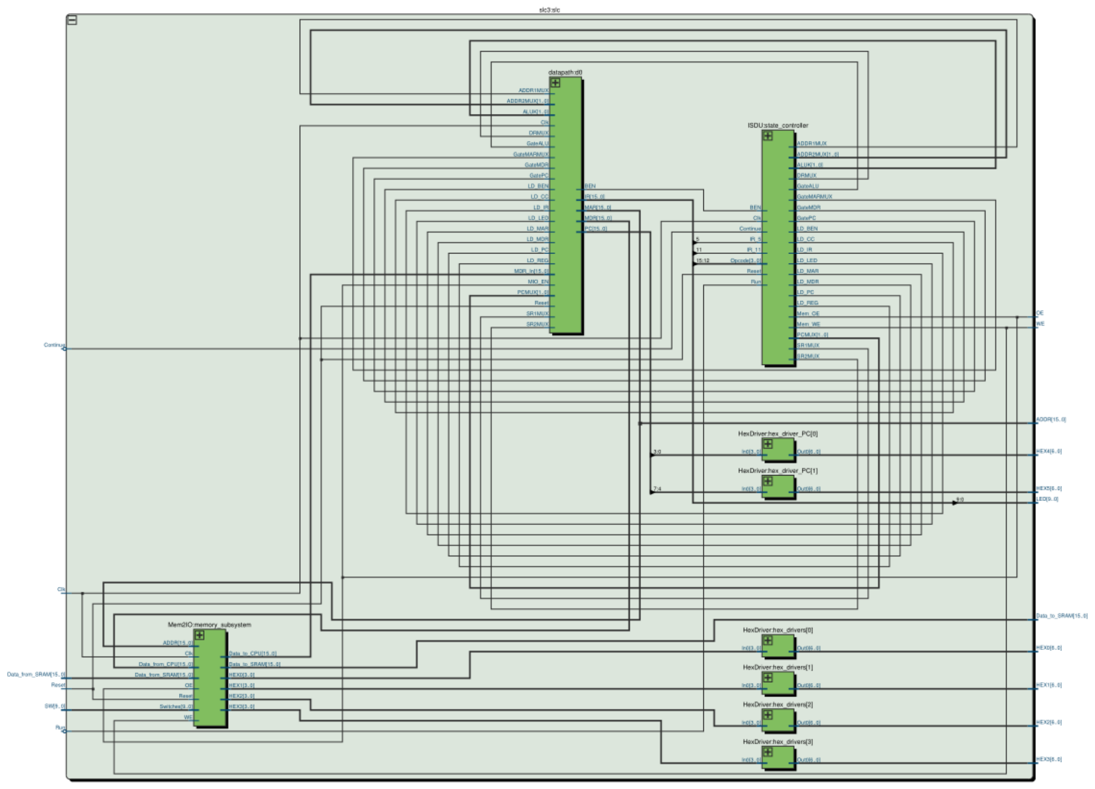
2.1 Summary of Operation

While operating on the DE10 board, we set switches to the starting address of our desired program; with the help of the provided first three lines of programs, we can jump to that starting address(set by switches) and execute our program. If we look inside our design, we have already been provided with the interface between the CPU and the memory(MEM2IO). The processor will then fetch the instruction from the memory. After receiving the instruction, then we decode the instruction into operand and opcode; we also set the conditional code according. Next, we execute the operation corresponding to the opcode with the provided operand.

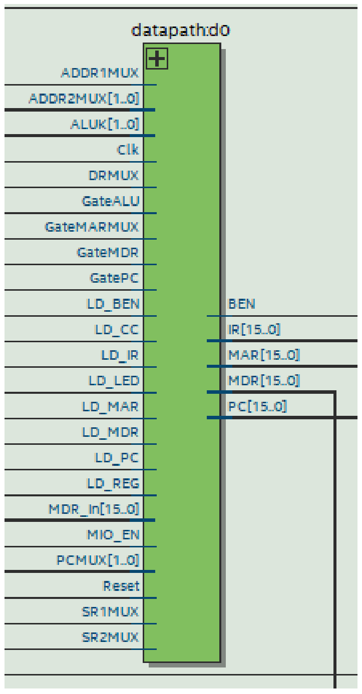
2.2 How the SLC-3 performs its functions

The fetch operation is just simply getting the instructions from the memory and store them in the IR. The decode operation is to differentiate various instructions by their opcode(first 4 bits), and make the whole operation go to the specific each next-state. Execute includes the basic calculations like ADD or AND. We call "fetch -> decode -> execute" a cycle, and whenever we finish a cycle, we jump back to the fetch operation of the next loop.

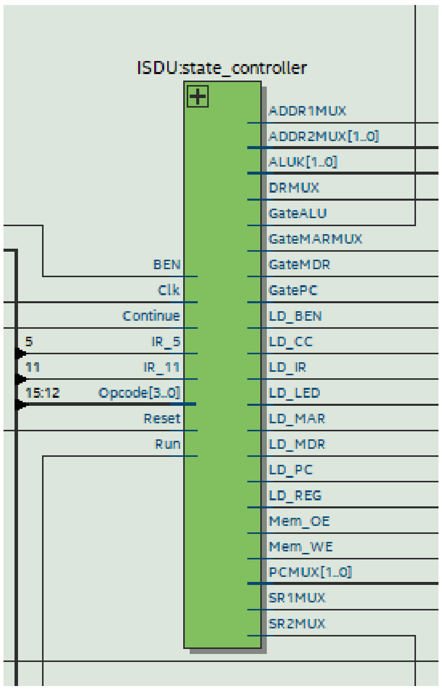
2.3 Block Diagram of slc3.sv



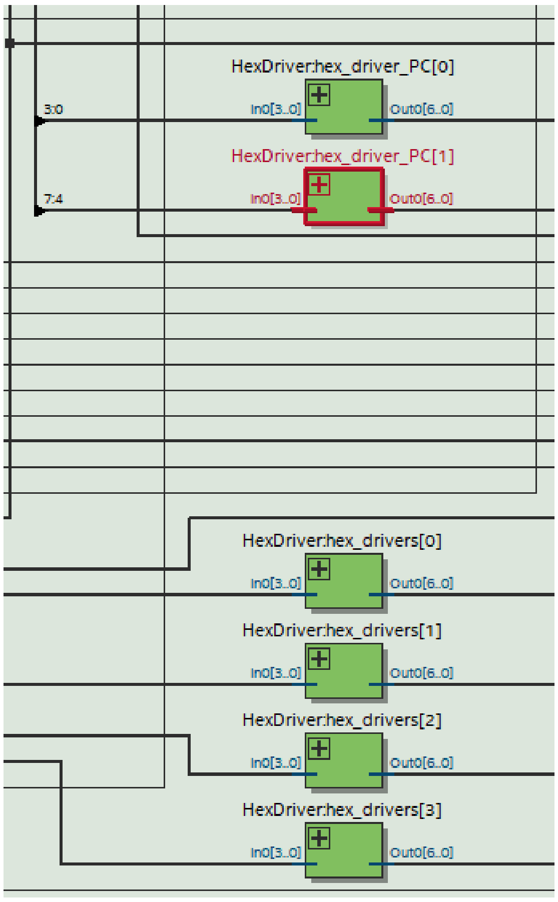
2.4 Block Diagram Inside slc3.sv



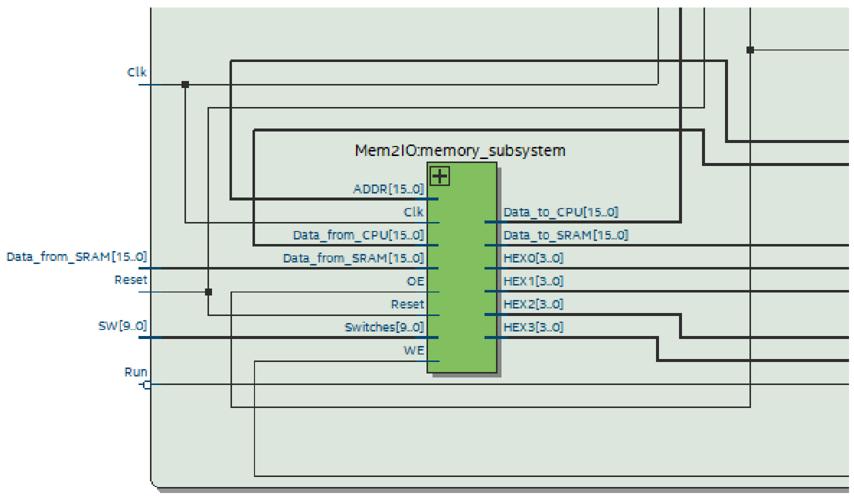
Datapath



ISDU



HexDriver



Mem2IO

2.5 Written Description of all .sv modules

1. **Module: reg_file**

   - Inputs: Clk, Reset, DRMUX_OUT, SR1MUX_OUT, SR2, LD_REG, Data_In
   - Outputs: SR1_OUT, SR2_OUT
   - Description: Reg_file has 8 registers in total, each of them can store a 16-bit value. We can read and write to the registers in the reg _file. To write to a register, we need to enable LD_REG and specifiy which register to write to using DRMUX_OUT, the data being written into the register will be provided by Data_In. To read from a register, we need to specify SR1MUX and SR2 to indicate which register's value we are currently reading from.
   - Purpose: Reading and writing from/to memory takes a long time, while the registers offers a quick R/W operation(even faster than a cache). Reg_file improve efficiency and save runtime of the programs.

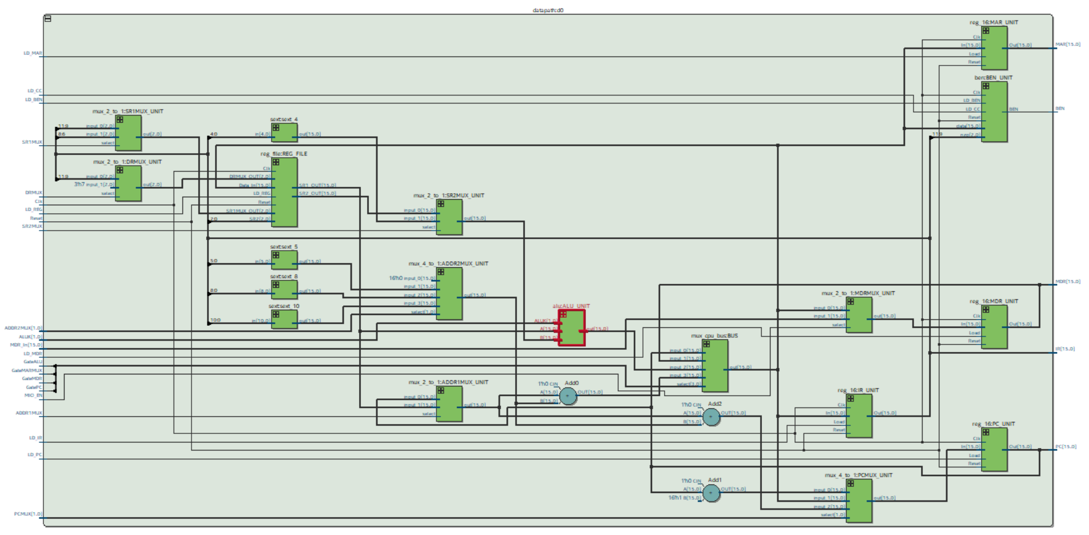
2. **Module: reg_16**

   - Inputs: Clk, Reset, Load, In
   - Outputs: Out
   - Description: This module serves as a 16-bit register. It preserves the value unless Load signal is enable and a new value is provided by In signal.
   - Purpose: In this lab, PC, IR, MDR, MAR are all 16 bits and need a place to store the value.

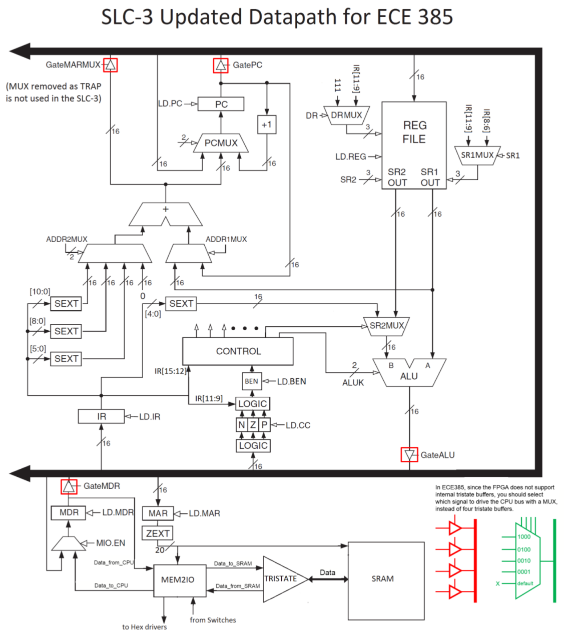
3. **Module: mux**
    - Inputs: input_0, input_1, select, width
    - Outputs: output
    - Description: This is a typical mux module that has been done many times in the previous lab. The mux depends on the select bit to control which input becomes the final output. Width is a parameter specified by the caller function to indicate how many bits the inputs should have.
    - Purpose: In this lab, PCMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, ADDR2MUX, and BUS are all the structures of MUX, so they will depend on this mux file to do their functions.
4. **Module: HexDriver**
    - Inputs: [3:0] In0
    - Outputs: [6:0] Out0
    - Description:Each hex value has its unique corresponding 7-bit binary value to display on the board screen.
    - Purpose: Hex module is used to output the hex value on screen.
5. **Module: ext**
    - Inputs: [N:0] in, N: parameter provided by caller
    - Outputs: [15:0] out
    - Description: This module can extend the bits of 11, 9, 6, 5(depending on the parameter N) into 16 bits by two algorithms. Sign extension will extend the bits based on the most significant bit and zero extension is just adding zero in front of all the bits to make the length become 16.
    - Purpose: The ADDR2MUX requires 16-bit inputs while we can only offer bits that are less than 16 if we don’t have the extension methods. Thanks to this module, we now can easily drag a part of IR and extend it to 16 bit so that we can do other operations.

6. **Module: datapath**

- Inputs: Clk, Reset, LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, GatePC, GateMDR, GateALU, GateMARMUX, SR2MUX, ADDR1MUX, DRMUX, SR1MUX, MIO_EN, PCMUX, ADDR2MUX, ALUK, MDR_In
- Outputs: MAR, MDR, IR, PC, BEN
- Description: This module serves as the bridge to link with different modules and makes them into a whole design. It also has various load inputs that can control each register whether to load in the data.
- Purpose: to connect each small modules and fulfil the operations we are going to make.



Datapath RTL



Ideal Datapth

7. **Module: ben**
    - Inputs: Clk, Reset, data, nzp, LD_BEN, LD_CC
    - Outputs: BEN
    - Description: This module compares the nzp value got from IR with the sign of the data got from the bus. When the condition is matched, output BEN will be high.
    - Purpose: Some operations in our design requires to setCC.
8. **Module: alu**
    - Inputs: A, B, ALUK
    - Outputs: out
    - Description: This module will depend on the value given by ALUK and do four different operations (A + B, A AND B, NOT A, A) and output the result.
    - Purpose: This used for doing the basic 4 operations that are required by the SLC-3.
9. **test_memory**
    - Inputs: Reset, Clk, data, address, rden, wren
    - Outputs: readout
    - Description: This memory has similar behavior to the on-Chip memory on the MAX10 board. This file is for simulations only. In simulation, this memory is guaranteed to work at least as well as the actual memory.
    - Purpose: This serves as an interface with memory contents to do the simulation.
10. **Module: slc3_testtop**
    - Inputs: SW, Clk, Run, Continue, LED
    - Outputs: HEX0, HEX1, HEX2, HEX3, HEX4, HEX5
    - Description: This one is the top-level file for the simulation. We added HEX4 and HEX5 to display the PC value, which makes us easier to debug.
    - Purpose: The top-file determines the pin names in the pin assignments, and uses this pins to connect with the slc3.file to run the design.

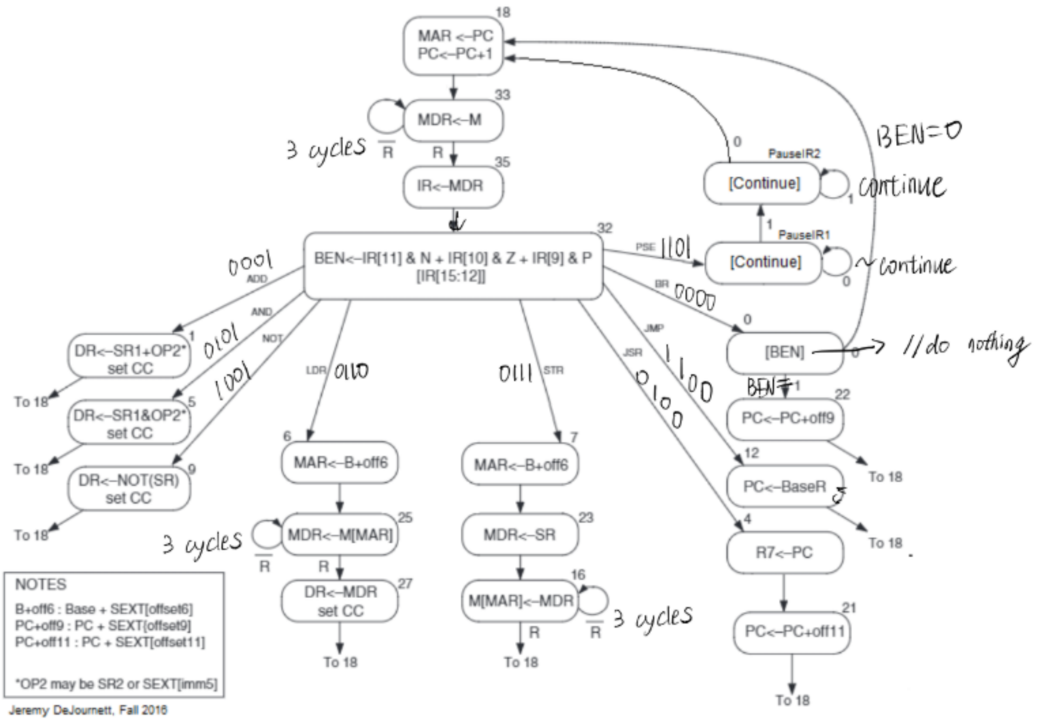
11. **Module: slc3_sramtop**
    - Inputs: SW, Clk, Run, Continue, LED
    - Outputs: HEX0, HEX1, HEX2, HEX3, HEX4, HEX5
    - Description: This one is the top-level file for the board. We added HEX4 and HEX5 to display the PC value, which makes us easier to debug.
    - Purpose: The top-file determines the pin names in the pin assignments, and uses this pins to connect with the slc3.file to run the design.
12. **Module: slc3**
    - Inputs: SW, Clk, Reset, Run, Continue, LED, Data_from_SRAM
    - Outputs: OE, WE, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, ADDR, Data_to_SRAM
    - Description: This one is also like a top-file since it basically define all the functions run by SLC-3.
    - Purpose: It process the data from the memory and head to the state table for performing all the SLC-3 operations.
13. **Module: Mem2IO**
    - Inputs: Clk, Reset, ADDR, OE, WE, Switches, Data_from_CPU, Data_from_SRAM
    - Outputs: Data_to_CPU, Data_to_SRAM, HEX0, HEX1, HEX2, HEX3
    - Description: MEM2IO module serves an interface for our SLC-3 Processor to communicate with main memory or external I/O devices. It manages all I/O with the DE10-Lite physical I/O devices, namely, the switches and 7-segment displays as well as R/W operation with SRAM.
    - Purpose: It serves as an interface with the CPU and the on-chip physical memory.
14. **Module: ISDU**
    - Inputs: Clk, Reset, Run, Continue, 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, PCMUX, DRMUX, SR1MUX, SR2MUX, ADDR1MUX, ADDR2MUX, ALUK, Mem_OE, Mem_WE
    - Description: This module is a finite state machine. It separates the whole fetch, decode, execute into several states and do different operations in each state. More importantly, it assigns the control signals to the datpath depending on the current state.
    - Purpose: It is the skeleton of the design that it tells the sequence of finishing each operations.

15. **Module: Instantiateram**
    - Inputs: Reset, Clk
    - Outputs: ADDR, wren, data
    - Description: The module helps instantiate the on-chip memory into our project folder. It "hard-coded" all the memory in mem_contents and write into our on-chip memory one-by-one.
    - Purpose: so that we can also run the program on DE10 board.

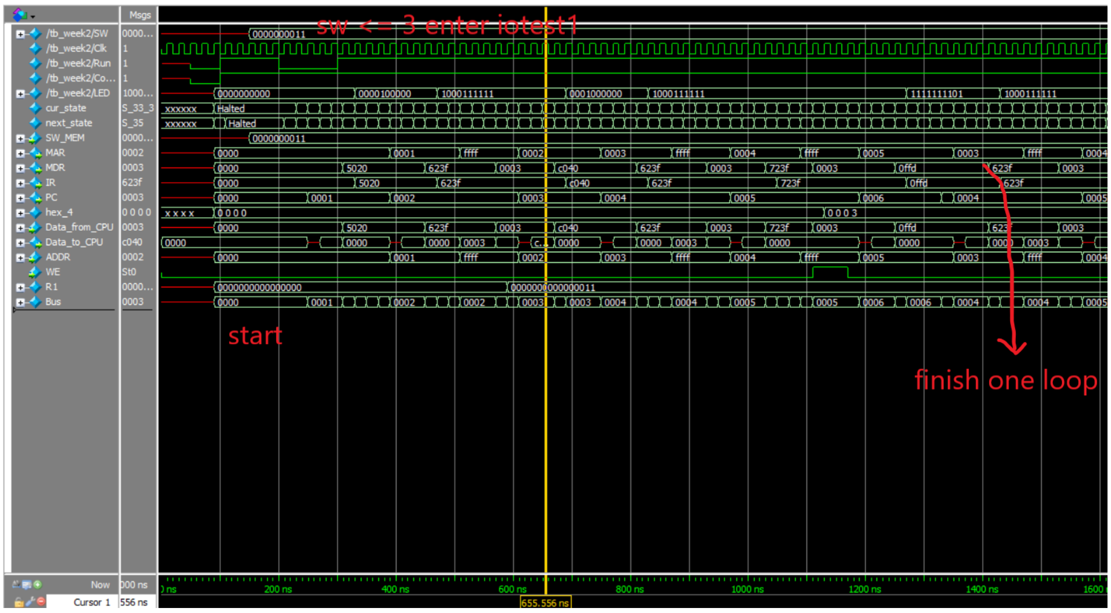
2.6 Description of the operation of the ISDU

ISDU is a state machine, and it includes all the operation in Fetch, Decode and Execute. ISDU controls the values of all the "selects" (LD.PC, SR1, DR, ADDR2MUX, and so forth) in the SLC-3 datapath. When we set those "select" bit into a specific value in every state, the data will do the operation that is suggested in this state. If we use the ISDU file to connect all the states together, we got a design that could do all the operations that we want.

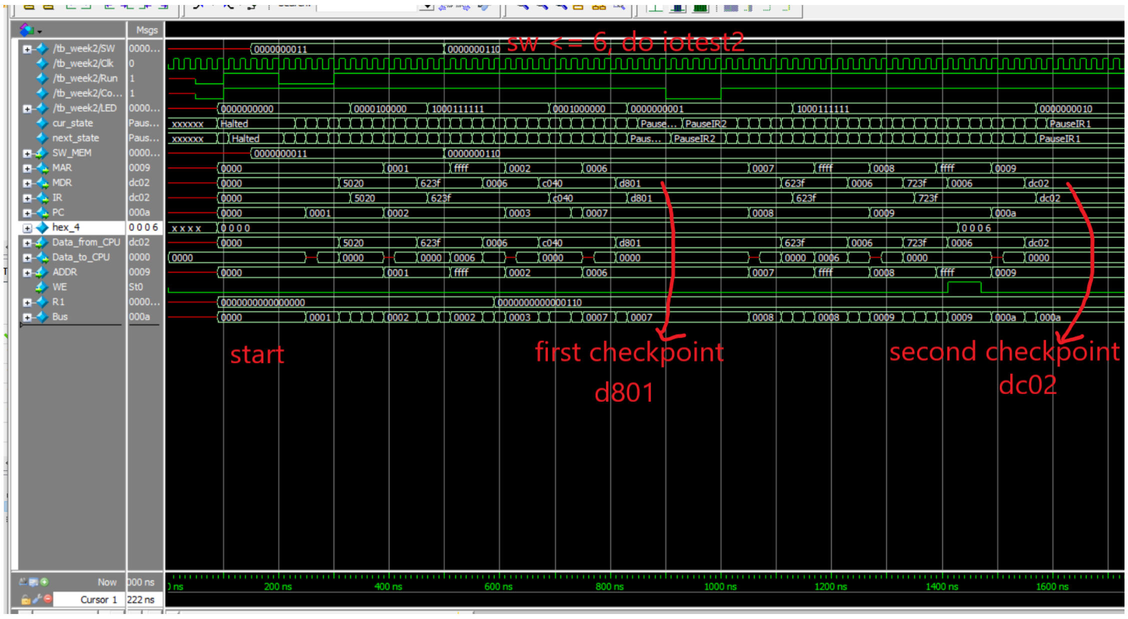
2.7 State Diagram of ISDU



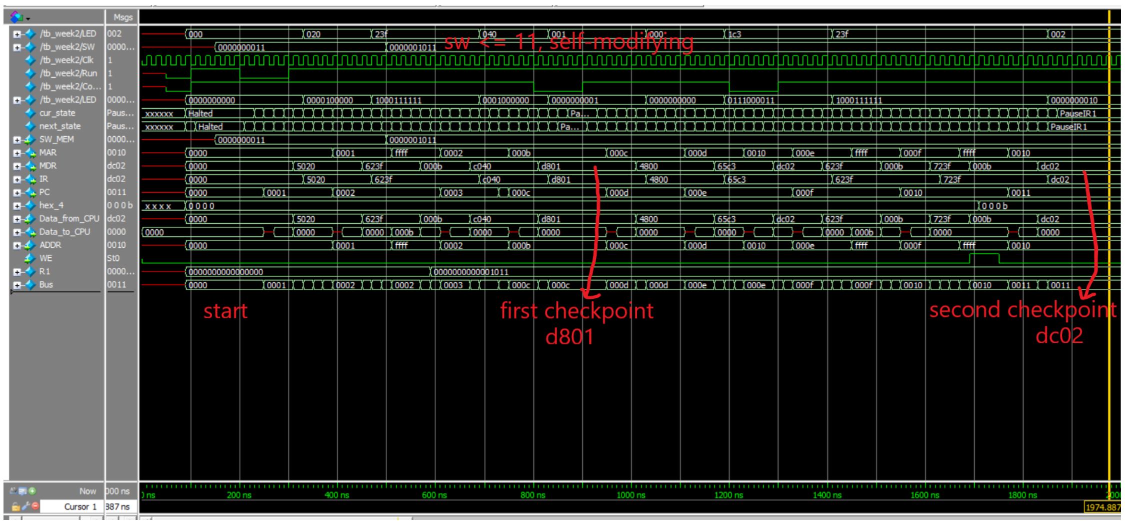
3 Simulations of SLC-3 Instructions



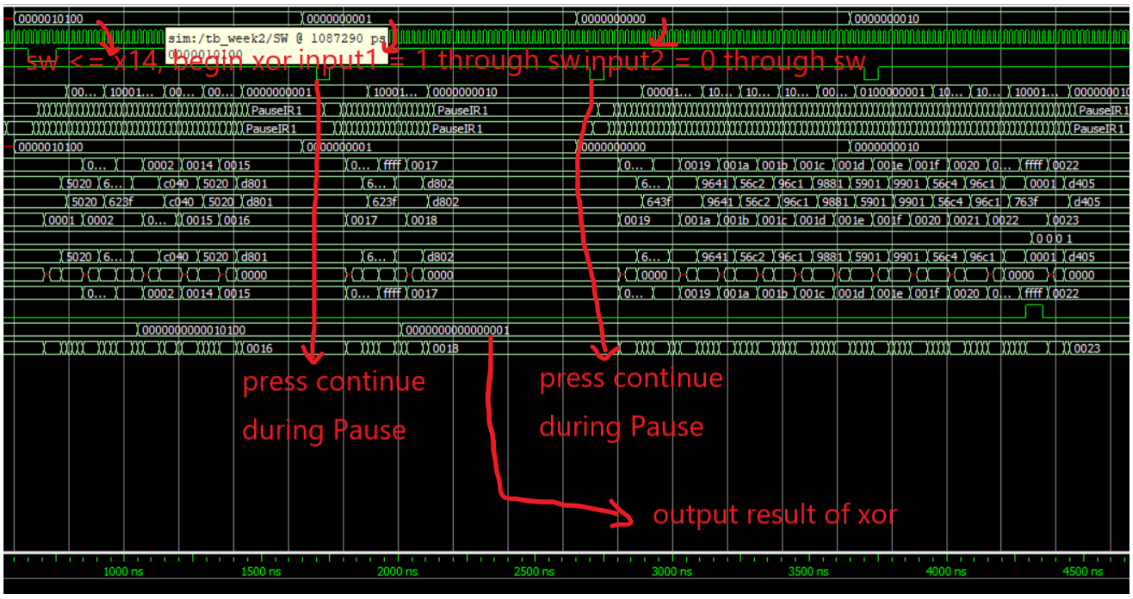
I/O Test 1



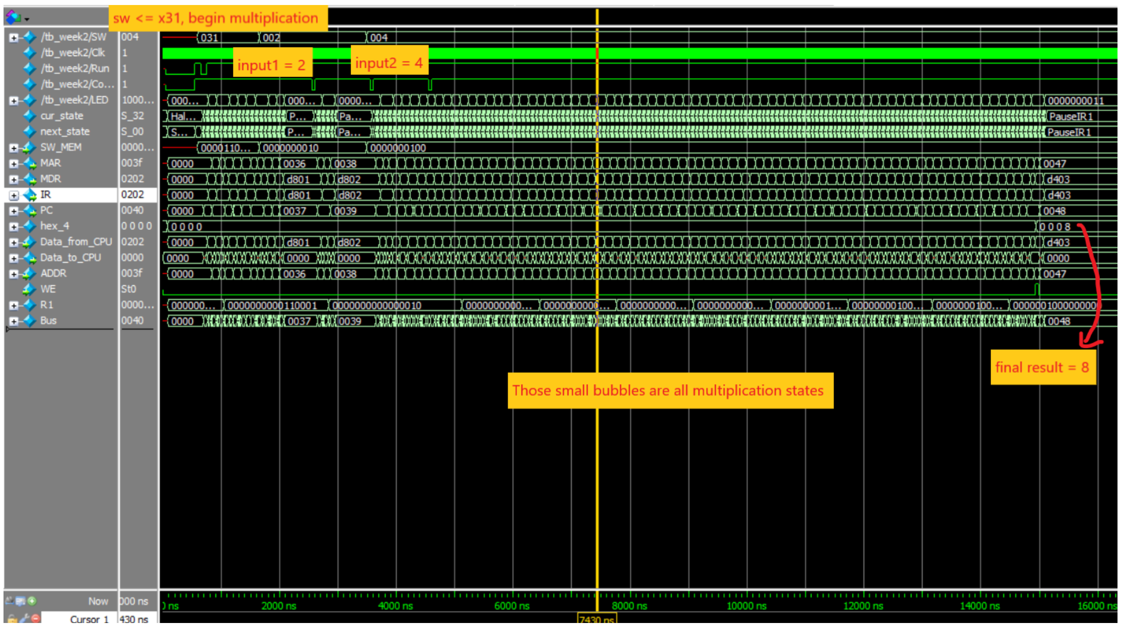
I/O Test 2



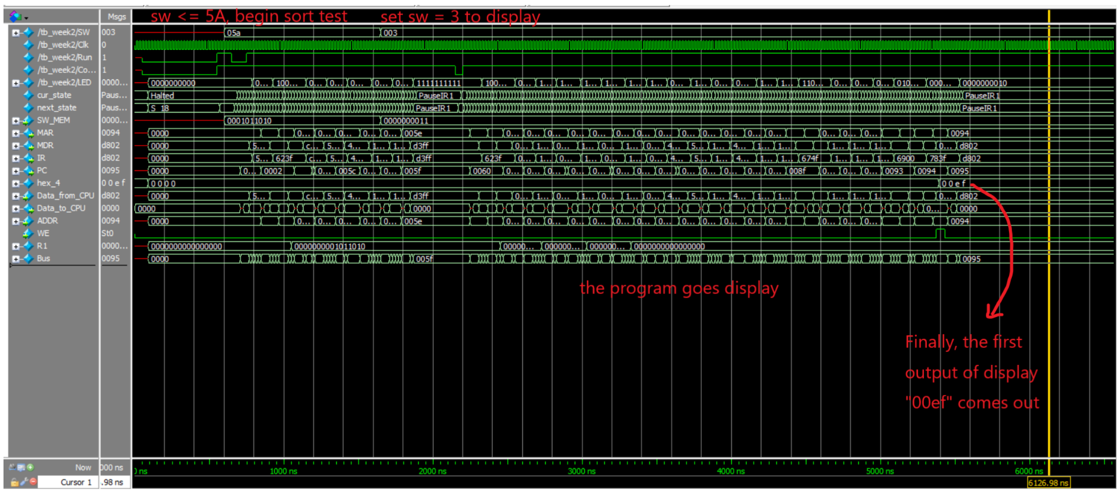
Self-Modifying Code



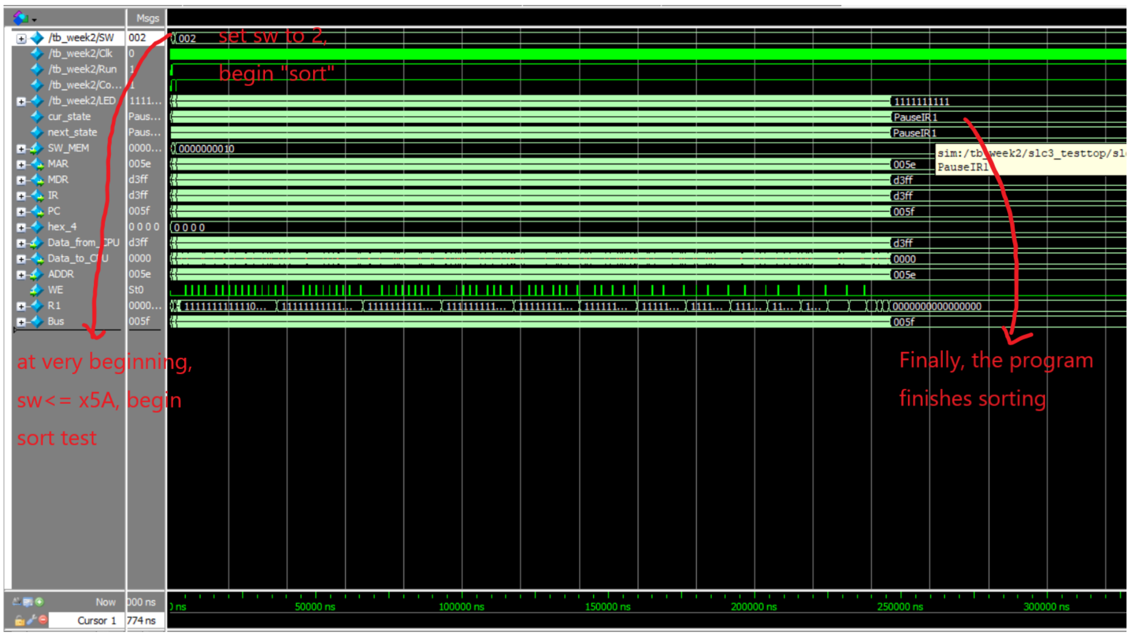
XOR


sw <= x31, begin multiplication
input1 = 2
input2 = 4
Those small bubbles are all multiplication states
final result = 8

Multiplier



Sort (only display mode)



Sort (only sort mode)

4 Post-Lab Questions

(a) **Refer to the Design Resources and Statistics in IQT and complete the following design statistics table.**

LUT	2381
DSP	0
Memory (BRAM)	0
Flip-Flop	1840
Frequency	83.35MHz
Static Power	90.03mW
Dynamic Power	16.69mW
Total Power	118.86mW

(b) **Document any problems you encountered and your solutions to them, and a short conclusion.**

We have encountered 2 major bugs. The first one is we thought it only takes 2 cycles for our CPU to read from SRAM, so the DE10 board never works(only displaying 0000 all times) while the simulation works fine. This bug leads us to try to learn signalTap to analyze the PC, MAR, MAR, IR values when continue is pressed, then we figured out that we didn't wait for enough cycles so that the programs continue to execute with the wrong PC value(data hasn't been fully loaded into our processor). We find the fix by adding one more state to read from memory, and we find that fix even before the professor announced that fix during lectures.

The second huge bug we have it we initially implemented BEN incorrectly. This caused our program not to be able to loop back to the start(BRnzp) and can execute only once. We fix this bug by rewriting our BEN logic and using a 2-always method to achieve synchronization.

(c) **What is MEM2IO used for, i.e. what is its main function?**

MEM2IO module serves as an interface for our SLC-3 Processor to communicate with main memory or external I/O devices(Switches and HEX displays). In the provided MEM2IO.sv, OE(Output Enable) and WE(Write Enable) is to control whether we are sending data to or from the CPU.

When we want to load data into our CPU, OE will be enabled and if the address is FFFF, zero-extended switch values will be read into the CPU; for all other addresses, memory contents will be read into the CPU.

When we want to transmit data out of our CPU, WE will be enabled and if the address is FFFF, hex drivers will display the data transmitted; for all other addresses, data will be written into the memory.

(d) **What is the difference between BR and JMP instructions?**

Firstly, BR is a conditional jump while JMP is an unconditional jump. So we have to judge the sign before doing BR instruction. More precisely, BR wants to satisfy the sign condition provided in its [11:9] bits, and will only do the jump operation for a distance provided in its offset numbers. JMP does not have this prior condition for jumping, and the destination address of its jumping is specified by the base register, which is also a bit different from BR.

(e) **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?**

In Patt and Patel's design, the R signal(or Ready Signal) is used to indicate whether data is still being read from memory or the read data is ready for use. If R is not asserted, which means we need to wait; R will be asserted when the processor finished reading from memory. This is necessary, because reading from memory is much slower than reading from registers/cache and usually takes more than 1 cycle.

However, in our SRAM, we don't have an R signal. Therefore, we compensate for the lack of the R signal with 2 extra cycles. Since in our case, all data fetching from memory is guaranteed to finish in 3 cycles, so we just make 2 more fetching states to achieve the same effect of the R signals.

Using extra cycles means we don't have to deal with synchronization issues of R signals. Because in Patt and Patel's design, R signals is an synchronized signal which may be asserted anytime when data is ready(and most likely NOT on a rising clock edge), which might need lead to undefined states if synchronization constraint is not met. However, since we do not have an R signal in our design, and use 2 extra states to achieve the same functionality, we do not have this synchronization concern because the state transition always happens on the rising clock edge, no matter how many extra states we have.

5 Extra Credit

In ECE411, I have learned how to calculate CPI(cycles per instructions) in single-cycle and multi-cycle processors. I applied the same principles in the below calculations. To get the MIPS(million instruction per second), I divide cycle time by CPI. Our cycle time is defined as 50MHz in the sdc file.

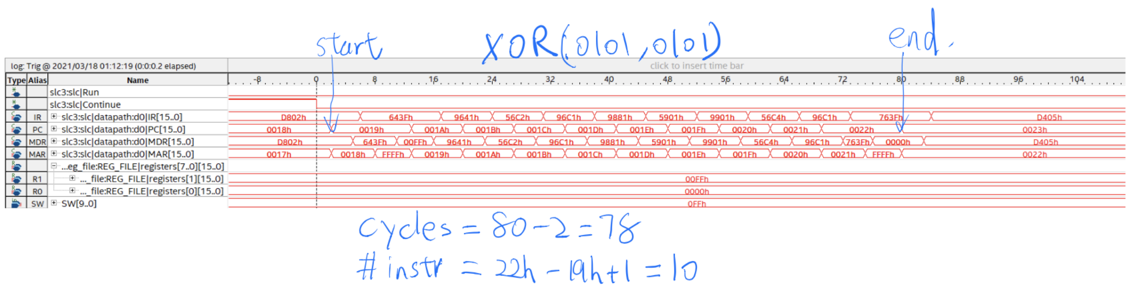
$$MIPS(Million\_Instruction/s) = \frac{Cycle\_Time(Million\_Cycle/second)}{CPI(cycle/instruction)} \tag{1}$$

5.1 SignalTap Traces

Since we set timestamp in Signal Tap to sample unit rather than time unit. We can directly get how many cycles per instruction takes from the signaltap tracer. (E.g. if instruction X at PC=0x20 ranges from 12 to 19 in the time stamp, then we know this instruction takes 19-12 = 7 cycles to finish.)

5.1.1 XOR

In this experiment, we recorded the traces of 1010 XOR 1010 right after the second value has been entered.

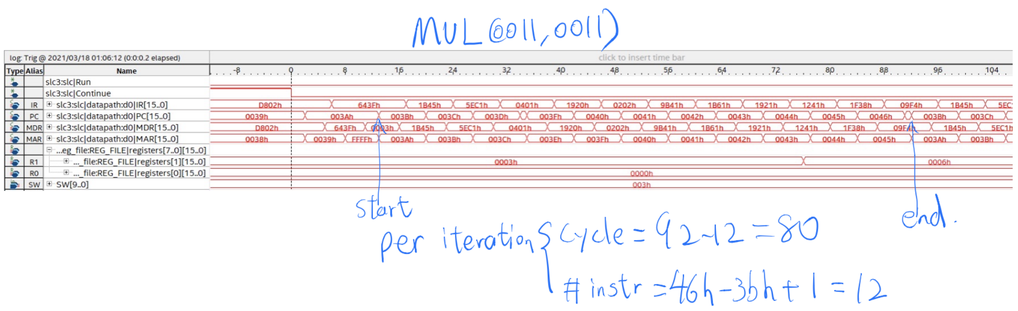


The first real instruction(after the PAUSE for the 2nd argument) is at 0x19, and the last instruction of XOR(before the PAUSE to display output) is 0x22. Therefore, there are 0x22-0x19+1=**10 instructions** in total. The first instruction starts at the 2nd clock and the last instruction ends at the 80th clock, which means there are 80-2= **78 cycles** in total.

Applying the above formula, **CPI** = 78/10, **MIPS** = 50/CPI ≈ 6.41

5.1.2 Multiplier

In this experiment, we recorded the traces of 0011 * 0011 right after the second value has been entered. **Since multiplier involves several iterations, in order to efficiently calculate the average MIPS for Multiplier, I decided to only record the number of instruction and number of cycles in one iteration, it might be less accurate than carrying out 10000 iterations, but it's simpler to record the SignalTap Trace, show the data in the report and it should be good enough for an estimation.**

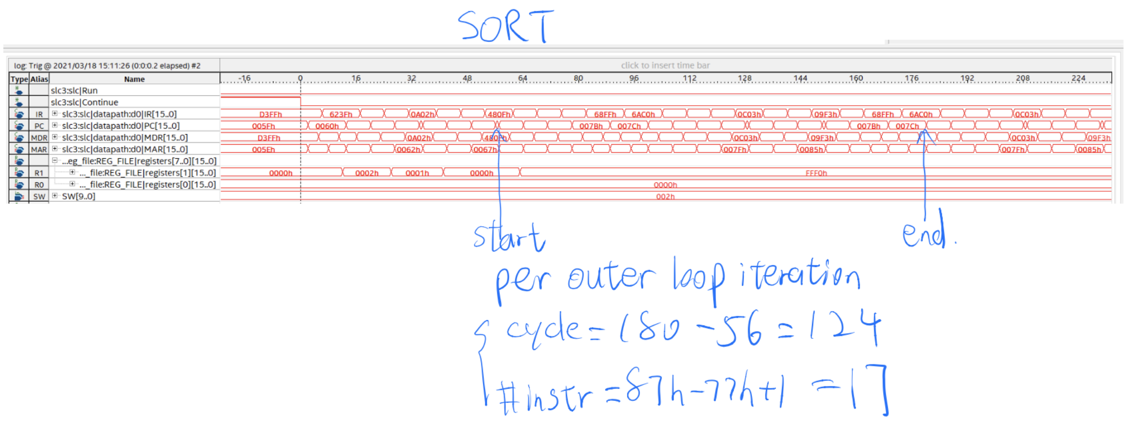


The first instruction in the iteration is at 0x3B, and the last instruction of the iteration is 0x46. Therefore, there are 0x46-0x3B+1=**12 instructions** in each iteration. The first instruction starts at the 13th clock and the last instruction ends at the 93rd clock, which means there are 93-13 = **80 cycles** in total.

Applying the above formula, **CPI** = 80/12, **MIPS** = 50/CPI = 7.5

5.1.3 Sort

In this experiment, we recorded the traces of sorting right we select the sorting subroutine. **Since Sorting involves several iterations, in order to efficiently calculate the average MIPS for this program, I decided to only record the number of instruction and number of cycles in one OUTER LOOP iteration, it might be less accurate than carrying out 10000 iterations, but it's simpler to record the SignalTap Trace, show the data in the report and it should be good enough for an estimation.**


SORT
log: Trig @ 2021/03/18 15:11:26 (0:0:0.2 elapsed) #2
Type
Alias
Name
slc3:slc|Run
slc3:slc|Continue
IR slc3:slc|datapath:d0|IR[15..0]
PC slc3:slc|datapath:d0|PC[15..0]
MDR slc3:slc|datapath:d0|MDR[15..0]
MAR slc3:slc|datapath:d0|MAR[15..0]
...eg_file:REG_FILE|registers[7..0][15..0]
R1 ..._file:REG_FILE|registers[1][15..0]
R0 ..._file:REG_FILE|registers[0][15..0]
SW SW[9..0]
start
end.
per outer loop iteration
cycle = 180 − 56 = 124
#instr = 87h − 77h + 1 = 17

The first instruction in the outer loop is at 0x77, and the last instruction in the outer loop is 0x87. Therefore, there are 0x87-0x77+1=**17 instructions** in each outer loop iteration. The first instruction starts at the 56th clock and the last instruction ends at the 180th clock, which means there are 180-56 = **124 cycles** in total.

Applying the above formula, **CPI** = 124/17, **MIPS** = 50/CPI = 6.855

5.2 MIPS Analysis

The different MIPS for different programs actually makes sense because each program use different instructions, and different instructions take a different number of cycles to finished(E.g. In our SLC-3 design, ADD takes 7 cycles to finish, while LDR takes 11 cycles since they need extra states to wait for memory data). And it's obvious that the programs that have more LDR and STR will have relatively lower MIPS.

The **overall estimated MIPS** for our CPU (ASSUMING XOR, Multiply and Sort have equal weights – each has a 33% chances of being called) will be (6.41 + 7.5 + 6.855) / 3 ≈ **6.922**

6 Conclusion

6.1 Functionality of Our Design

Our SLC-3 design is fully functional and we got all points during the demo in Week1 and Week2. All provided programs(xor, multiply, sort...) can be executed with correct behavior, details of which can be found in the above Section 3. Both simulation with Quartus and actual operation on the DE10 board works perfectly. Moreover, we also explore the option of verifying our design using Signal Tap.

6.2 Suggestions to Lab-Manual and Given Document

We surprisingly found that we are delivered the lab content ahead of our actual doing lab time, which helps a lot in understanding the whole image of our labs. However, we found the pin assignment provided for lab6 week1 is really confusing. We spent loads of time correcting that.