ECE 385
Spring 2020

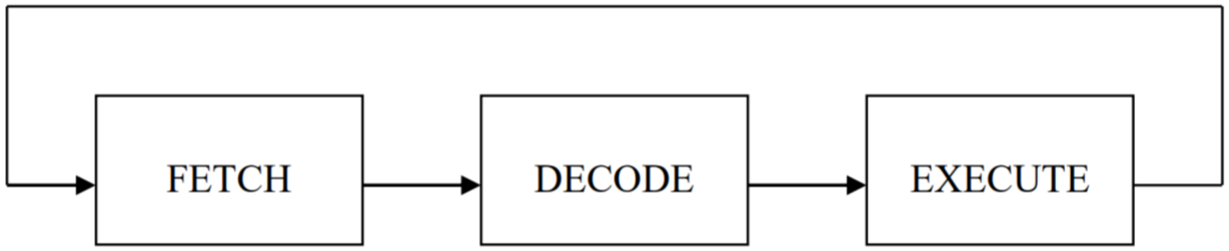
Experiment #6

Simple Computer SLC-3.2 in SystemVerilog

Ishaan Datta, Lukas Dumasius
Lab ABE/Tuesday 3pm
Jiaxuan Liu

Introduction & Summary of Operation

The purpose of the lab is to create and implement a relatively simple microprocessor in SystemVerilog. It is part of a subset of the LC-3 Instruction Set Architecture that we have used in previous classes. It is a 16-bit processor with 16-bit Program Counter (PC), 16-bit instructions, and 16-bit registers throughout. The CPU, In/Out Interface unit, and Memory Storage unit. The I/O unit communicates signals with external devices such as our switches and hex displays. The Memory Storage contains actual instructions and data for the processor along with test memory. Our processor performs three operations, namely FETCH, DECODE, and EXECUTE. An instruction is "FETCH"ed from memory, "DECODE"ed, and after the instruction is "EXECUTE"ed, the next instruction is "FETCH"ed in a cyclical manner.



The Instruction Register holds the current operation we are operating on at the moment, while the PC, also operated as a register in our implementation keeps track of the successive operation that is to be loaded next into the Instruction Register. The MAR (Memory Address Register) contains the memory address of the current location that we would like to read from or write to. The actual data itself is loaded to or from the MDR (Memory Data Register). The ISDU (Instruction Sequencing/Decoding Unit) contains the state machine that decodes each instruction based on the opcode of the data in the IR. This then sets the appropriate signals such as MUX selects and other controls that gets sent to every other part of the processor for correct execution along with identifying the actual operation to perform on the inputs to the ALU. After correct execution, the aforementioned looping of decoding instructions continues. The register file unit holds the 7 registers (R0-R7) the processor can directly access for operation inputs and outputs. These are implemented in a 8x16 register unit called module register_unit_8x16, described further in the module descriptions.

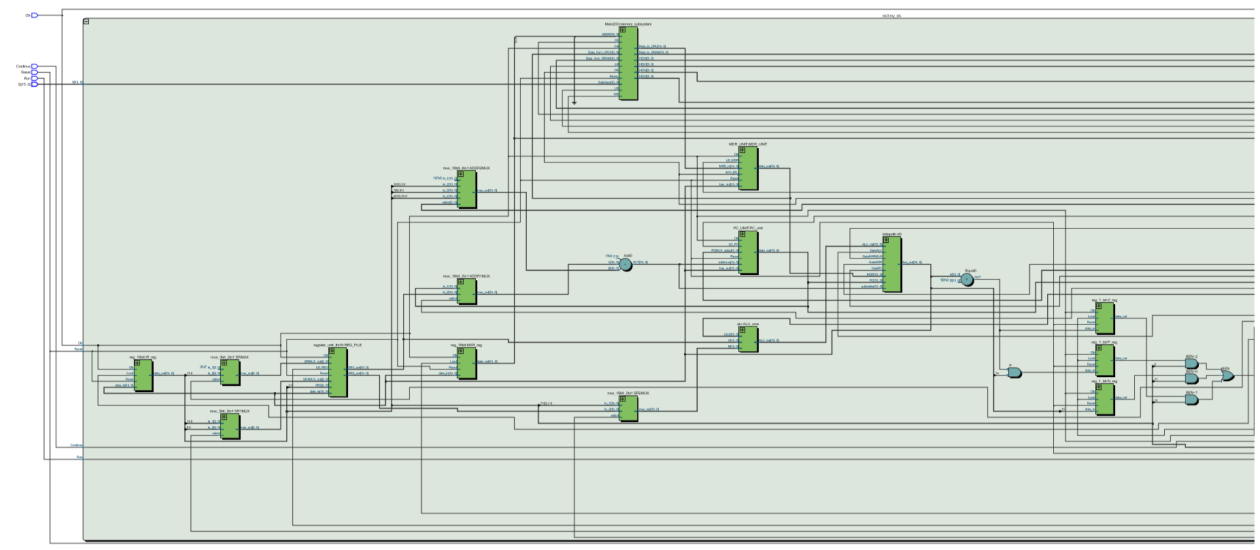
Naturally, as each separate operation/instruction process begins with placing the PC value into the MAR (MAR<-PC) which then reads the data at the address in MAR and stores it into MDR (MDR<-M[MAR]). The entire instruction is then stored into IR (IR<-MDR) for continuing the operation. After this, the PC is regularly incremented by one (PC<- PC + 1) to be ready to fetch the next operation during the next cycle (unless there is a jump/JSR that specifies a different offset). Following the FETCH stage, the opcode located in IR[15:12] is read and decoded by the FSM in the ISDU and the appropriate signals are set for the other processor components where the actual computation is carried out. The IR also contains data such as the offset or destination (DR) and source (SR) registers made available through the ISDU as these are necessary for the ALU to complete computation. For instructions that have

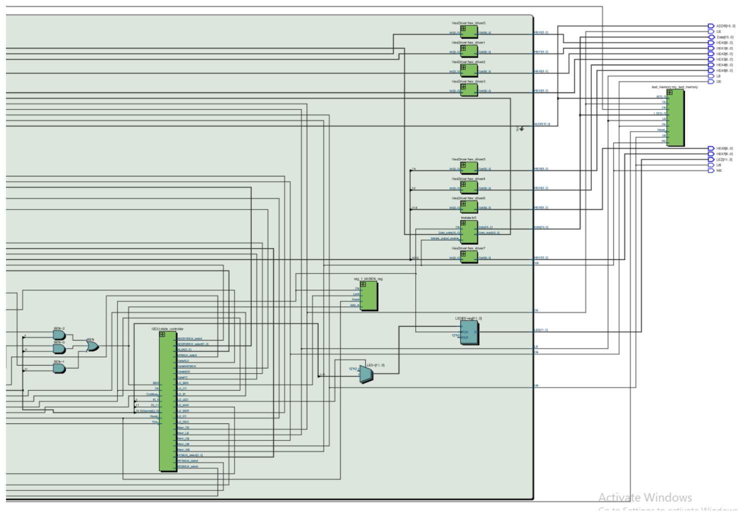
"Setcc" (Set Condition Code), we must appropriately set the values of status registers "nzp" based on the output value of the operation loaded into the register of interest in the register file. For example, if the number is negative, our reg_1_bit N_reg must go high. Notice, we implement the status registers each as a separate 1-bit register as only one needs to be marked high at a time.
Our processor has 3 input signals controlling it. When "RESET" is activated, the ISDU resets to the "halted" state and the PC is set to 0. The switches can be used to load in a specific starting point for the processor (instructions written in test_memory). Regardless, when Run is activated, we tell the processor to continue "FETCH"ing instructions one line after another and PC continues to increment by 1. If continue is activated, the processor will return to the "FETCH" stage.

The following table contains all the the operations we can perform along with their 4-bit opcodes [15:12] that define their operation for the ISDU. Operations marked with SEXT denote sign extending the 2's-complement value (of imm5, etc) from the IR as the PC, Register File, etc. only operate on 16-bit inputs.

Instruction	Instruction(15 downto 0)						Operation
ADD	0001	DR	SR1	0	00	SR2	R(DR) ← R(SR1) + R(SR2)
ADDi	0001	DR	SR	1	imm5		R(DR) ← R(SR) + SEXT(imm5)
AND	0101	DR	SR1	0	00	SR2	R(DR) ← R(SR1) AND R(SR2)
ANDi	0101	DR	SR	1	imm5		R(DR) ← R(SR) AND SEXT(imm5)
NOT	1001	DR	SR	111111			R(DR) ← NOT R(SR)
BR	0000	n z p	PCoffset9				if ((nzp AND NZP) != 0) PC ← PC + SEXT(PCoffset9)
JMP	1100	000	BaseR	000000			PC ← R(BaseR)
JSR	0100	1	PCoffset11				R(7) ← PC; PC ← PC + SEXT(PCoffset11)
LDR	0110	DR	BaseR	offset6			R(DR) ← M[R(BaseR) + SEXT(offset6)]
STR	0111	SR	BaseR	offset6			M[R(BaseR) + SEXT(offset6)] ← R(SR)
PAUSE	1101	ledVect12					LEDs ← ledVect12; Wait on Continue

Block diagram of slc3.sv





Written Description of .sv Modules

- Module: slc3

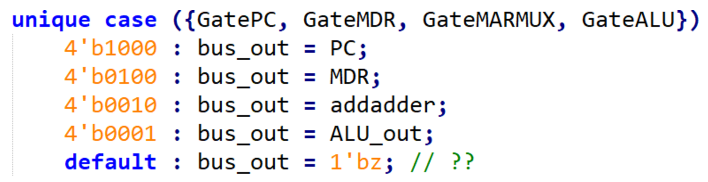
Inputs: [15:0] S, Clk, Reset, Run, Continue

Outputs: [11:0] LED, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5, [6:0] HEX6, [6:0] HEX7, CE, UB, LB, OE, WE, [19:0] ADDR

Inout wire: [15:0] Data

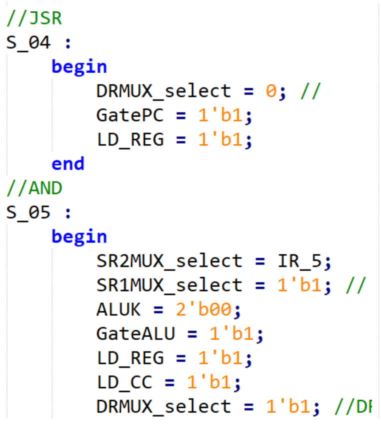
Description & Purpose: This module functions as the “pseudo” top level below the actual top level in our lab set by instructors (lab6_toplevel) and instantiates all of the other important modules for our processor. This module functions as the communication hub for all of the other modules and their control signals. Most importantly, we see that ISDU is instantiated within this module and all of the signals it controls for the other modules (Register file, all control MUXs, input/output and Hex drivers) are passed through this module (slc3.sv).

- Module: datapath
Inputs: GATEPC, GateMDR, GateALU, GateMARMUX, [15:0] PC, addadder, MDR, ALU_out
Outputs: [15:0] bus_out
Description & Purpose: The datapath is the bus of the entire processor. This is an essential component as, as its name suggests, is the datapath of the entire processor. This is a 16-bit wide “wire” that allows a central location to read/write from so that all modules can communicate. If the IR needs to grab instructions loaded from the MAR address from MDR and the ALU needs to write outputs to the REG_File as well, for example, then we need to separate these read/writes and make sure that only one module is writing to the data bus at a time. This is implemented with a unique case statement allowing only GatePC, GateMDR, GateMARMUX, or GateALU to write to the bus once at time.

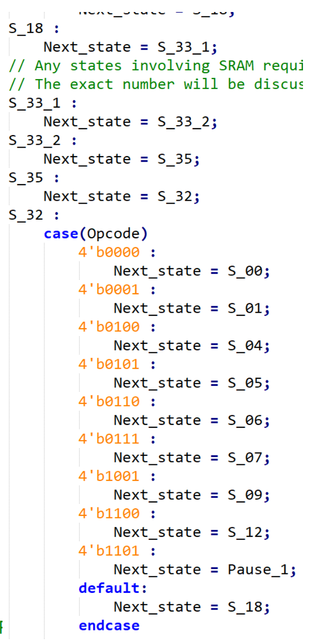


```
unique case ({GatePC, GateMDR, GateMARMUX, GateALU})
    4'b1000 : bus_out = PC;
    4'b0100 : bus_out = MDR;
    4'b0010 : bus_out = addadder;
    4'b0001 : bus_out = ALU_out;
    default : bus_out = 1'bz; // ??
```

- Module: ISDU & Further description of operation of ISDU
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, DRMUX_select, SR1MUX_select, SR2MUX_select, ADDR1MUX_select, [1:0] ADDR2MUX_select, ALUK, PCMUX_select, Mem_CE, Mem_UB, Mem_LB, Mem_OE, Mem_WE
Description & Purpose: This module is at the heart of our processor as it contains the Finite State Machine (FSM) that can handle all of the possible LC3 instructions. Firstly, all possible 26 states are enumerated and if reset is clicked, we set the current state to halted. Next, there are two major parts. The next-state logic and the current-state based signal setting. The next-state logic section defines which state we should move onto next. States that only depend on the current-state for next-state are simply set with a unique case statement on the current state. For states such as S_32 which depend on the current Opcode for figuring out next state, we also have a case statement on this Opcode (linked from the top level) which defines the next-state. Then, the last part sets MUX selects, Gates, loads, and all other control signals for other modules in the processor



```
//JSR
S_04 :
    begin
        DRMUX_select = 0; //
        GatePC = 1'b1;
        LD_REG = 1'b1;
    end
//AND
S_05 :
    begin
        SR2MUX_select = IR_5;
        SR1MUX_select = 1'b1; //
        ALUK = 2'b00;
        GateALU = 1'b1;
        LD_REG = 1'b1;
        LD_CC = 1'b1;
        DRMUX_select = 1'b1; //DI
```



```
S_18 :
    Next_state = S_33_1;
// Any states involving SRAM requi
// The exact number will be discus
S_33_1 :
    Next_state = S_33_2;
S_33_2 :
    Next_state = S_35;
S_35 :
    Next_state = S_32;
S_32 :
    case(Opcode)
        4'b0000 :
            Next_state = S_00;
        4'b0001 :
            Next_state = S_01;
        4'b0100 :
            Next_state = S_04;
        4'b0101 :
            Next_state = S_05;
        4'b0110 :
            Next_state = S_06;
        4'b0111 :
            Next_state = S_07;
        4'b1001 :
            Next_state = S_09;
        4'b1100 :
            Next_state = S_12;
        4'b1101 :
            Next_state = Pause_1;
        default:
            Next_state = S_18;
        endcase
```

depending only on a case statement on the current state.

- Module: PC_UNIT
Inputs: Clk, Reset, LD_PC, [1:0] PCMUX_select, [15:0] bus_out, [15:0] addrmux
Outputs: [15:0] data_out
Description & Purpose: This is the program counter which holds the address of the next operation to be performed. The module itself is relatively simply implemented with a reg_16bit module to hold the PC value. In our implementation, we chose to "instantiate" the PCMUX within the PC_UNIT because the PXMUX could take in the PC+1 value in input 1 (in_1), bus_out in input 2 (in_2), and addrmux value in input 3 (in_3). This MUX's control signals (PCMUX_select and PCMUX_out) are set by the ISDU and its output (PCMUX_out) is always fed into the actual PC (reg_16bit PC) which will only load the value when its select (LD_PC) is activated in the ISDU module.

- Module: alu
Inputs: [1:0] ALUK, [15:0] A, [15:0] B
Outputs: [15:0] ALU_out
Description & Purpose: The ALU serves a fundamental function to most of the operations we can perform with our processor. The control bits ALUK are set by the ISDU and depending on a case statement on ALUK, we can perform AND, NOT, ADD, and Set A. (With inputs A, B). The output of this computation unit is set to ALU_out which gets picked up by the datapath module datapath d0 and placed in the bus in the case where GateALU is "high".

- Module: Mem2IO
Inputs: Clk, Reset, [19:0] ADDR, CE, UB, LB, OE, WE, [15:0] Switches, [15:0] Data_from_CPU, [15:0] Data_from_SRAM
Outputs: [15:0] Data_to_CPU, [15:0] Data_to_SRAM, [3:0] HEX0, [3:0] HEX1, [3:0] HEX2, [3:0] HEX3
Description & Purpose: This module primarily handles all of the input/output capabilities of the board (Switches and hex displays). Because I/O for this CPU is memory-mapped, I/O devices are directly connected to memory signals with a buffer on the memory data bus. If memory access happens with I/O device (switches), then the memory (with the buffer) is deactivated and the I/O device's (switches) values/data is used instead. This is implemented in the always_comb unit on an if statement with (WE && ~OE). For writing to hex displays, this is implemented in the always_ff unit in the bottom that sets hex_data based on Data_from_CPU if there's no "Reset".

- Module: tristate
Inputs: Clk, tristate_output_enable, [N-1:0] Data_write
Outputs: [N-1:0] Data_read
inout wire [N-1:0] Data
Description & Purpose: This module is a buffer between MEM2io and SRAM, taking in a 16 bit input from MEM2io and SRAM. When the control bit tristate_output_enable is set to "high", the output, "Data", is set to the input coming from MEM2io, Data_write.

- Module: register_unit_8x16
Inputs: Clk, Reset, LD_REG, [2:0] DRMUX_out, [2:0] SR1MUX_out, [2:0] SR2, [15:0] data_in
Outputs: [15:0] SR1_out, [15:0] SR2_out
Description & Purpose: This is the essential register unit of our processor that contains the regular "scratch" registers R0-R7. Each RX register must be 16 bits and we have 8 registers, thus making the entire unit as a whole an 8x16 Register File. Because we knew going into this lab that each of the 8 individual registers could be treated as 16 bit individual registers, we set up our overall format for the register file as using the reg_16bit module to instantiate each individual register and simply did this 8 times within the register_unit_8x16 module. The module also accounts for Source Registers SR1 and SR2 by having a unique case on SR1MUX_out and SR2 which simply sets SR1_out and SR2_out based on the 3-bit wide select bits which are decoded. Also, we account for Destination Registers with an if statement on LD_REG (to decide whether we even need to load) followed by (nested-inside) a unique case on DRMUX_out which sets the load bit to the appropriate register to "high" if its corresponding binary number is outputted from DRMUX_out.

- Module: reg_16bit
Inputs: Clk, Reset, LD_REG, [2:0] DRMUX_out, [2:0] SR1MUX_out, [2:0] SR2, [15:0] data_in
Outputs: [15:0] SR1_out, [15:0] SR2_out
Description & Purpose: This is a simple, fundamental, 1-bit register implemented with an always_ff block that sets data_out to "0"s if Reset is activated and data_in is loaded to be accessible to data_out if "Load" is activated and "Reset" signal is not (Synchronous reset). The Instruction Register (IR) itself is implemented as a reg_16bit module in the top level module along with the MAR register. Most importantly, as stated in the register_unit_8x16 written module description, the individual registers R0-R7 are regular functionality 16-bit registers so they are implemented within register_unit_8x16 as reg_16bit modules (Read more: **Module: register_unit_8x16**).

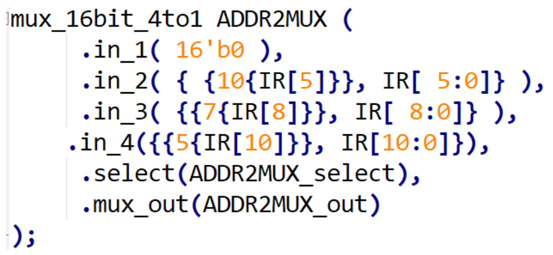
- Module: reg_1_bit
Inputs: Clk, Reset, Load, data_in
Outputs: data_out
Description & Purpose: This is a simple, fundamental, 1-bit register implemented with an always_ff block that sets data_out to "0"s if Reset is activated and data_in is loaded to be accessible to data_out if "Load" is activated and "Reset" signal is not (Synchronous reset). We used the 1-bit registers for status registers N, Z, P, and BEN the first 3 of which hold the appropriate setCC values based on the instruction being performed and the output written to reg_file, as detailed in "Summary of Operation".

- Module: mux_3bit_2to1
Inputs: [2:0] in_1. [2:0] in_2, select
Outputs: [2:0] mux_out
Description & Purpose: This is a basic 3-bit wide, 2-to-1 MUX implemented as usual with an always_comb block containing a unique case on the select bit, which sets the the output to in_1

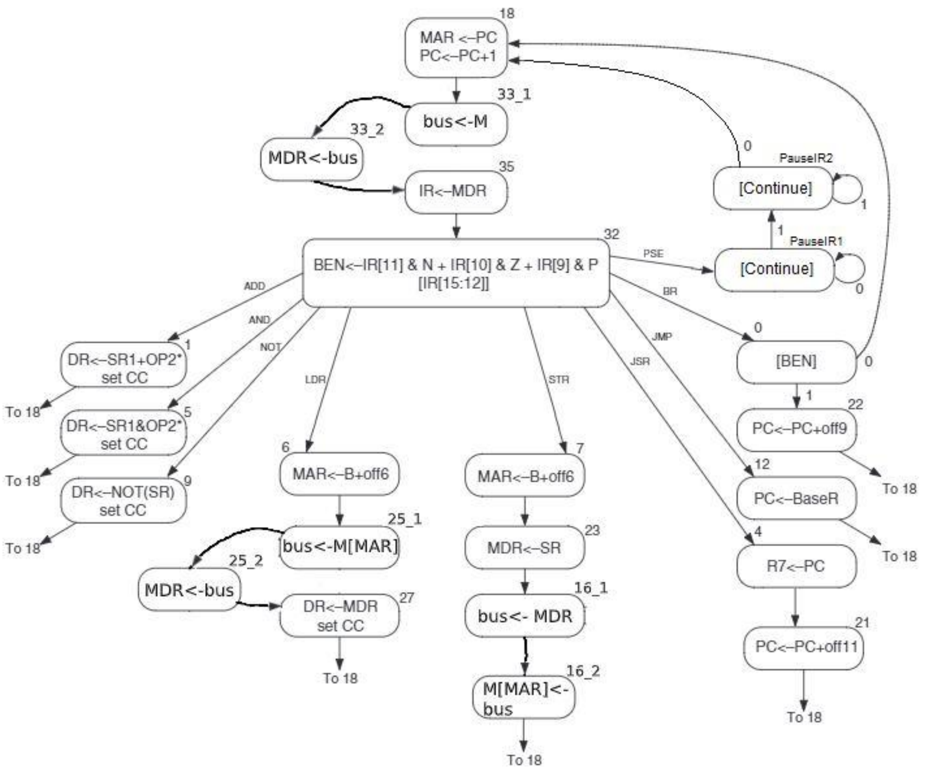
for (select==0) and in_2 for (select==1). This MUX is used to "instantiate" the DRMUX in the slc3.sv "top level" (not technically the top level, as explained in "Summary of Operation", as lab6_top_level is above slc3) along with SR1MUX as they both input 3-bit wide sources into the REG_File from 2 selection options, both of which are provided by IR.

- Module: mux_16bit_2to1
Inputs: [15:0] in_1, [15:0] in_2, select
Outputs: [15:0] mux_out
Description & Purpose: This is a basic 16-bit wide, 2-to-1 MUX implemented as usual with an always_comb block containing a unique case on the select bit, which sets the the output to in_1 for (select==0) and in_2 for (select==1). This MUX is used to "instantiate" the ADDR1MUX in the slc3.sv "top level" (not technically the top level, as explained in "Summary of Operation", as lab6_top_level is above slc3) along with SR2MUX as they both input 16-bit wide sources into the adder module and the B input to the ALU, respectively from 2 selection options each, all of which are 16-bit wide coming from SEXT(IR[4:0]), SR2_out from REG_File, SR1_out from REG_File, and and PC_out.

- Module: mux_16bit_4to1
Inputs: [15:0] in_1, [15:0] in_2, [15:0] in_3, [15:0] in_4, [1:0] select
Outputs: [15:0] mux_out
Description & Purpose: This is a basic 16-bit wide, 4-to-1 MUX implemented as usual with an always_comb block containing a unique case on the select bit, which sets the the output to in_1 for (select==00), in_2 for (select==01), in_3 for (select==10), in_4 for (select==11). This MUX is a bit unique in that it is created for only one use purpose, namely, it is used to "instantiate" the ADDR2MUX in the slc3.sv "top level" (not technically the top level, as explained in "Summary of Operation", as lab6_top_level is above slc3) as each input 1 through 4 is a 16-bit wide input. This module is also a bit unique because we use some SystemVerilog syntax that we do not see used in many of the other modules. As seen to the right, we use repeated statements of "{X{IR[Y]}}, IR[Y-1:0]" in the top level slc3 "instantiation" of ADDR2MUX.


```
mux_16bit_4to1 ADDR2MUX (
      .in_1( 16'b0 ),
      .in_2( { {10{IR[5]}}, IR[ 5:0]} ),
      .in_3( {{7{IR[8]}}, IR[ 8:0]} ),
     .in_4({{5{IR[10]}}, IR[10:0]}),
      .select(ADDR2MUX_select),
      .mux_out(ADDR2MUX_out)
);
```

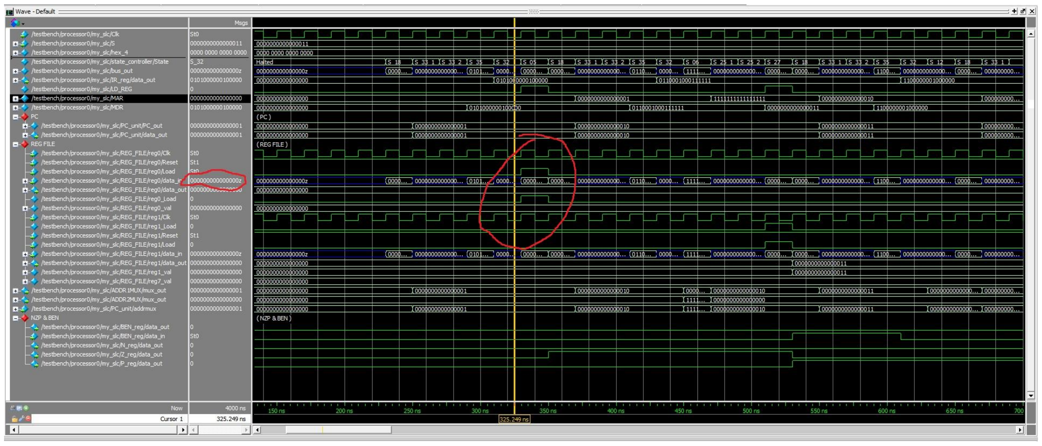
State Diagram of ISDU



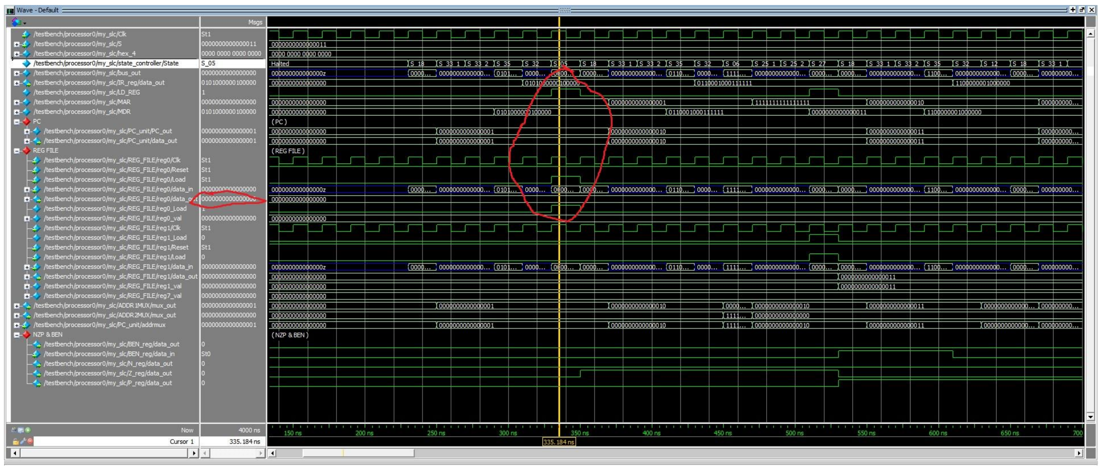
Annotated simulation waveform

AND:

Before:



After:

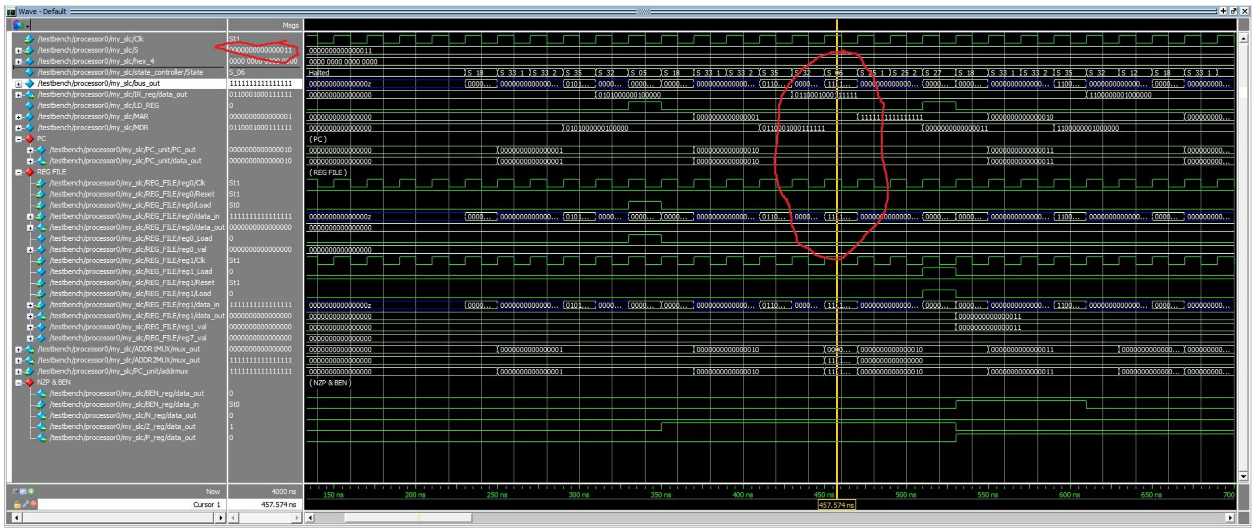


Here we see the operation for the AND instruction which stems from the i/o test where we AND the value in R0 with 16-bit 0’s. We see that before, the IR register contains the opcode 0101 in its most significant bits which corresponds to the AND instruction. The ISDU receives this opcode and sends us to state 5, which is correct. Before state 5, the value of R0 input from the bus (not loading) ends with 00z, then after the result is calculated (All 0s), R0 stores the appropriate value of 16-bit 0s after this value is passed through the bus. We have further confirmation that the result was sent through the bus correctly because the bus_out value switches from 00z (high z) to set 16-bit 0s.

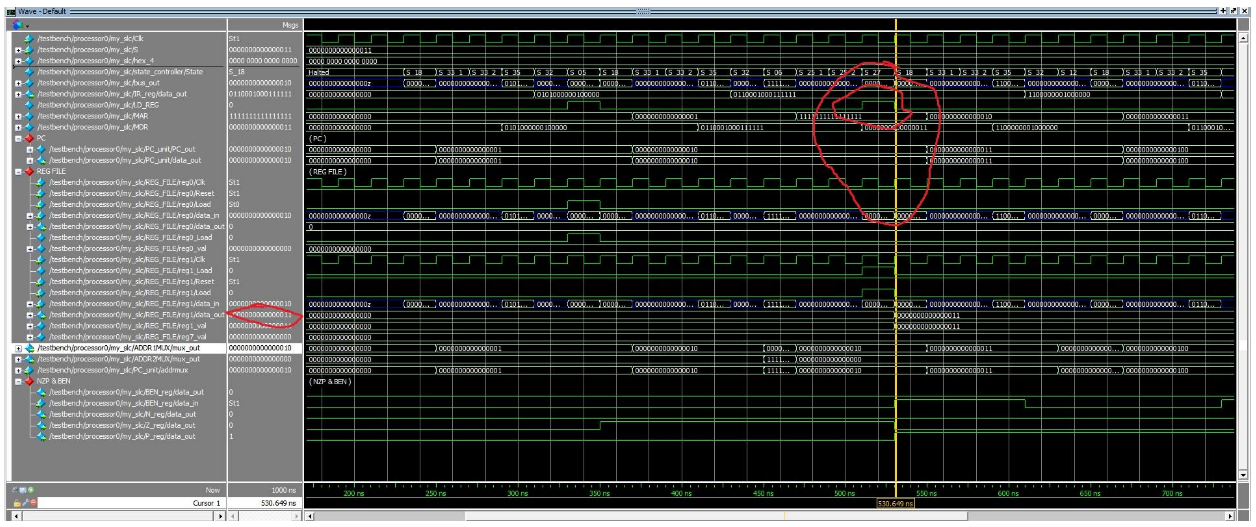
At the rising edge after the 33_2 state, MDR gets loaded with data from the BUS. The delay for this fetch operation is there to ensure proper reads from the SDRAM.

LDR:

Before:

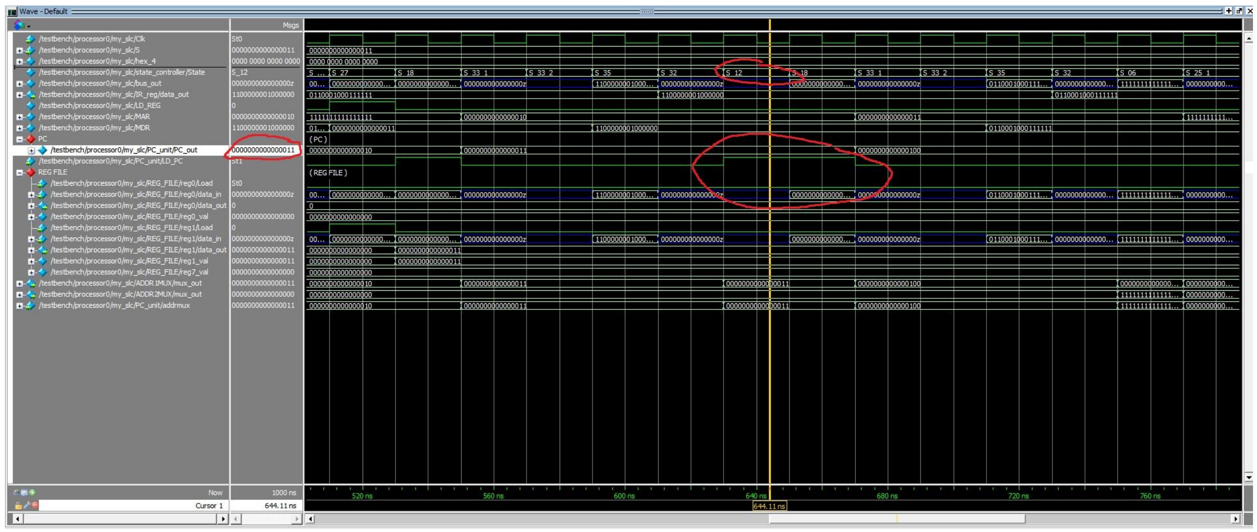


After:



Here we see the operation for the LDR instruction, also stemming from the i/o test. We pass LDR(R1, R0, inSW) which means that we will load the data available at memory location of Base Register R0 + sign extended offset of inSW into R1. We see that IR is loaded with op code 0110 in its 4 most significant bits, which corresponds to LDR. We go into state 6 to offset the Base register of R0 (already 0) with the inSW value of 11. State 25_1 and State 25_2 load the value at this memory location into MDR. We see during state 27 that LD_REG goes high as expected, and we see that the value from the bus is loaded into R1 and R1 data_out ends at 0000000000000011 and after we return to State 18.

JMP:



```
Wave - Default                                          Msgs
/testbench/processor0/my_slc/Clk                        St0
/testbench/processor0/my_slc/S                          0000000000000011
/testbench/processor0/my_slc/hex_4                      0000 0000 0000 0000
/testbench/processor0/my_slc/state_controller/State     S_12
/testbench/processor0/my_slc/bus_out                    000000000000000z
/testbench/processor0/my_slc/IR_reg/data_out            1100000001000000
/testbench/processor0/my_slc/LD_REG                     0
/testbench/processor0/my_slc/MAR                        0000000000000010
/testbench/processor0/my_slc/MDR                        1100000001000000
PC
  /testbench/processor0/my_slc/PC_unit/PC_out           0000000000000011
/testbench/processor0/my_slc/PC_unit/LD_PC              St1
REG FILE
  /testbench/processor0/my_slc/REG_FILE/reg0/Load       St0
  /testbench/processor0/my_slc/REG_FILE/reg0/data_in    000000000000000z
  /testbench/processor0/my_slc/REG_FILE/reg0/data_out   0
  /testbench/processor0/my_slc/REG_FILE/reg0_val        0000000000000000
  /testbench/processor0/my_slc/REG_FILE/reg1/Load       0
  /testbench/processor0/my_slc/REG_FILE/reg1/data_in    000000000000000z
  /testbench/processor0/my_slc/REG_FILE/reg1/data_out   0000000000000011
  /testbench/processor0/my_slc/REG_FILE/reg1_val        0000000000000011
  /testbench/processor0/my_slc/REG_FILE/reg7_val        0000000000000000
/testbench/processor0/my_slc/ADDR1MUX/mux_out           0000000000000011
/testbench/processor0/my_slc/ADDR2MUX/mux_out           0000000000000000
/testbench/processor0/my_slc/PC_unit/addrmux            0000000000000011
Now                                                     1000 ns
Cursor 1                                                644.11 ns
```

Here we see the operation for the JMP instruction, also stemming from the i/o test. We pass JMP(R1) after our previous instruction, so that means our PC should set to the value in R1 (which stayed at 0000000000000011). We see that IR is loaded with op code 1100 in its 4 most significant bits, which correctly corresponds to JMP. The ISDU sends us to State 12 which correctly sets LD_PC to high as expected, and after this PC takes on the value 0000000000000011 as expected from R1 to continue on the i/o test at the appropriate instruction.

Post-Lab Questions

LUT	543
DSP	0
Memory (BRAM)	0
Flip-Flop	279
Frequency (MHz)	66.2
Static Power (mW)	98.63
Dynamic Power (mW)	6.86
Total Power (mW)	171.88

- MEM2io is used to provide I/O functionality and its main purpose is acting as an intermediary between the SRAM and our processor implementation. Data that goes from from MAR and to and from MDR goes through MEM2io. This unit also handles inputs from the switches on the board along with outputting to Hex drivers. It can reset hex_data to all 0's if "Reset" signal is activated, or if Write Enable (WE) is activated, Data_from_CPU is written to hex_data. This is performed synchronously inside an always_ff block while the logic for "Data_to_CPU" which handles inputs from both SRAM and switches can be placed inside of an always_comb block with nested if-statements on WE and ~OE.

- The BR and JMP instructions are sometimes confused because they are closely related. BR can be considered practically as a conditional JMP. For JMP, the address in BaseR is placed in PC to change where we grab the next instruction for. This is done unconditionally. On the other hand, for BR to execute, we must satisfy and nzp condition. That is, the last instruction to setCC and send its end/output value through the bus to the Reg_File is the value that we condition on. For example, for BRn we only change PC in the case where the last value was in fact negative. Otherwise, we continue incrementing PC as normal. Also, BR does not use a BaseR as JMP does. BR works by offsetting the current PC value with a 2's complement sign-extended on PCoffset11.

- The R signal is the "memory ready" signal. In a traditional LC3 implementation, the MAR contains the address of an instruction/data to be operated on and this is read from memory to be loaded into MDR. This memory access process may take multiple clock cycles, so we only proceed and move onto next state when the ready signal is activated, telling us that the memory access it complete. Our implementation does not use the R signal, so we use multiple "sub"- states for states such as S_33 that use the memory R signal. By having state S_33_1 proceed to S_33_2 and only then to S_35, we delay by extra clock cycle to make sure the memory access is appropriately complete.

Conclusions

During the course of this lab, we ran into a few errors and small mistakes that resulted us in spending even more time debugging than we spent putting the processor together! The biggest mistake we found was that while we tried to keep track of all inputs to MUXes, their order, and which select bit combination they were associated with, when implementing our top level slc3 linked with the ISDU module, we still accidentally flipped the ordering of a few of the entries in the ISDU module which resulted in the select bits for multiple MUXes (Including SR1MUX_select, SR2MUX_select, DRMUX_select, and ADDR2MUX) to choose the wrong input. This took quite some time to track down to the ISDU after confirming the MUXes were operating as expected and we found that following a consistent system in the ISDU and MUX modules made it easiest to match up and confirm cases. We would both have the LC3 datapath open and follow a right-to-left ordering system for inputs in_1, in_2, etc on the ADDR2MUX, for example. Then, we found that our reg_16bit.sv which contained all of our registers (including the Reg_File) was not setting the Register data_out values in a consistent manner. Upon further investigation, we found that in the always_comb blocks that set the individual R0-R7 Register load values within our Reg_File module (see more: **register_unit_8x16**), we had to switch to blocking assignments which made the Reg_File function able to take on new values for registers even though we originally experimented with non-blocking assignments to try to avoid a multiple driver error.

To wrap up, this lab was the most challenging yet in terms of the complexity of the state machine and it was very easy for a simple typo (Especially in units like the ISDU, during state transitions) to make it appear to us as if another module was broken and thus much time was needed to track down problems. Generally the assignment was well put together, we would say that the most useful resource was *Patt and Patel* so make sure to point the students that way next semester as well!