ECE 385

Spring 2024

Experiment 5

SLC-3

Ashton Billings and Carson Conquest

Section AL1/M&W 4:00 – 4:50 p.m.

TA: Tianhao Yu

1. Introduction

a. Summary of SLC-3 basic functionality

The Simplified Little Computer 3 (SLC-3) is a 16-bit logical processor that performs operations on groups of bits according to structures specified by the instruction it is currently executing. The instructions appear in the form of 4-digit hex words and are stored in a memory block that the cpu must access using the cpu_to_io block. We (the user) specify which memory address we would like to access first by changing the 16-bit value on the switches, then hitting run. This sets the program counter to that value and the cpu will fetch, decode, and possibly execute the instruction that is stored at that address in that exact order. The program counter will increment so that the next instruction is fetched after the current instruction finishes execution, and this cycle will repeat until a pause instruction is reached.

b. How the SLC-3 differs from the LC-3

The Simplified LC-3 (SLC-3) is a version of the LC-3 processor with a reduced instruction set. We refer to this type of processor as a Reduced Instruction Set Computer (RISC) chip. This means that the overall functionality of the processor remains the same, but it is only able to execute a subset of the LC-3's instruction set. The instructions it can execute are ADD, AND, BR, JMP, JSR, LDR, NOT, STR, and PAUSE. Another key difference that we'll discuss in greater detail later is the presence of a Ready signal in the LC-3 control unit that is not present in the SLC-3 control unit. This necessitated additional states within the FSM of the SLC-3 which we'll show in the state diagram section later on.

2. Written Description and Diagrams of SLC-3

a. Summary of Operation

A 16-bit (4-hex digit) memory address is input by toggling the switches on the FPGA; the program counter is then set to the value on the switches. When the run button is pressed, the cpu fetches and decodes the instruction stored at that memory location. The program counter then increments. The cpu then uses its various modular components to perform operations on and/or direct the flow of the groups of bits specified by the instructions. The information is passed between these components via the main data bus that surrounds the cpu. When these instructions are finished, the relevant data can be stored in any one of a number of locations usually

specified by the specific bit pattern of the instruction. The cpu then moves back to the fetch state as it fetches the contents of the next memory address.

b. Function Descriptions

We'll start with the fetch and decode states, as they must occur for any one of the instructions to be executed. Fetch involves setting the program counter (PC) to the value stored on the switches that are memory mapped to xFFFF. The bridge from user input to memory is accomplished via the cpu_to_io block in the SLC-3 schematic. Input, memory, cpu, and output can be thought of as four distinct locations in space, and the aforementioned cpu_to_io block handles the flow of data between connected pairs of the locations. Once the PC is specified, the program goes to that memory address and writes the identity of the memory address into the memory address register (MAR) and the program counter is incremented by exactly one. The computer then accesses the contents of that memory address and writes them into the memory data register (MDR). The final part of fetch is that the contents of the MDR are now written into the instruction register (IR) so that decoding can commence.

Decoding involves reading the first four bits ([15:12]) of the IR as that tells the cpu which instruction it should execute. These four bits are referred to as the opcode of the instruction. The possible opcodes that are executable on the SLC-3 are 0000, 0001, 0101, 1001, 1100, 0100, 0110, 0111, and 1101. These opcodes respectively correspond to BR, ADD, AND, NOT, JMP, JSR, LDR, STR, and PAUSE.

BR (branch) involves testing bits IR[11:9] against the condition codes N, Z, and P (negative, zero, and positive respectively) using the logic block connected to the NZP register inside the cpu. *Refer to the SLC-3 state diagram for a description of the exact logic used to determine branch conditions.* Depending on if a bit of IR[11:9] matches corresponding condition code, a branch is executed via setting the PC to PC+offset9, where offset9 is determined by sign extending IR[8:0]. The program then continues running (fetching→decoding→executing) from this new memory address.

ADD is an instruction that involves (unsurprisingly) bitwise addition. There are two possible scenarios for the add instruction, determined by IR[5]. If IR[5] is 0, then the cpu will add the contents of source register 2 (SR2) to the contents of source register 1 (SR1) and store the result in the

destination register (DR). If IR[5] is 1, then the cpu will add the contents of imm5 (specified by sign extending IR[4:0]) to the contents of the source register (SR) and store the result in the destination register (DR). The bitwise addition is performed in the ALU that takes as inputs the contents of SR1 and the result of SR2MUX. The output of the ALU then goes to the DR via the main data bus.

AND uses the exact same processes and connections as ADD, except the operation being performed on the operands is a bitwise logical AND instead of bitwise addition.

NOT likewise utilizes inversion logic to bitwise invert the contents of SR and write the result into the DR using the same connections as ADD and AND. Note that even though the output of SR2MUX still gets fed into the ALU, it doesn't affect the results of the operation.

JMP utilizes connections within the cpu to write the contents of the base register (BaseR) specified by IR[8:6] into the PC, thus causing the program to "jump" to a new memory address and continue running from there.

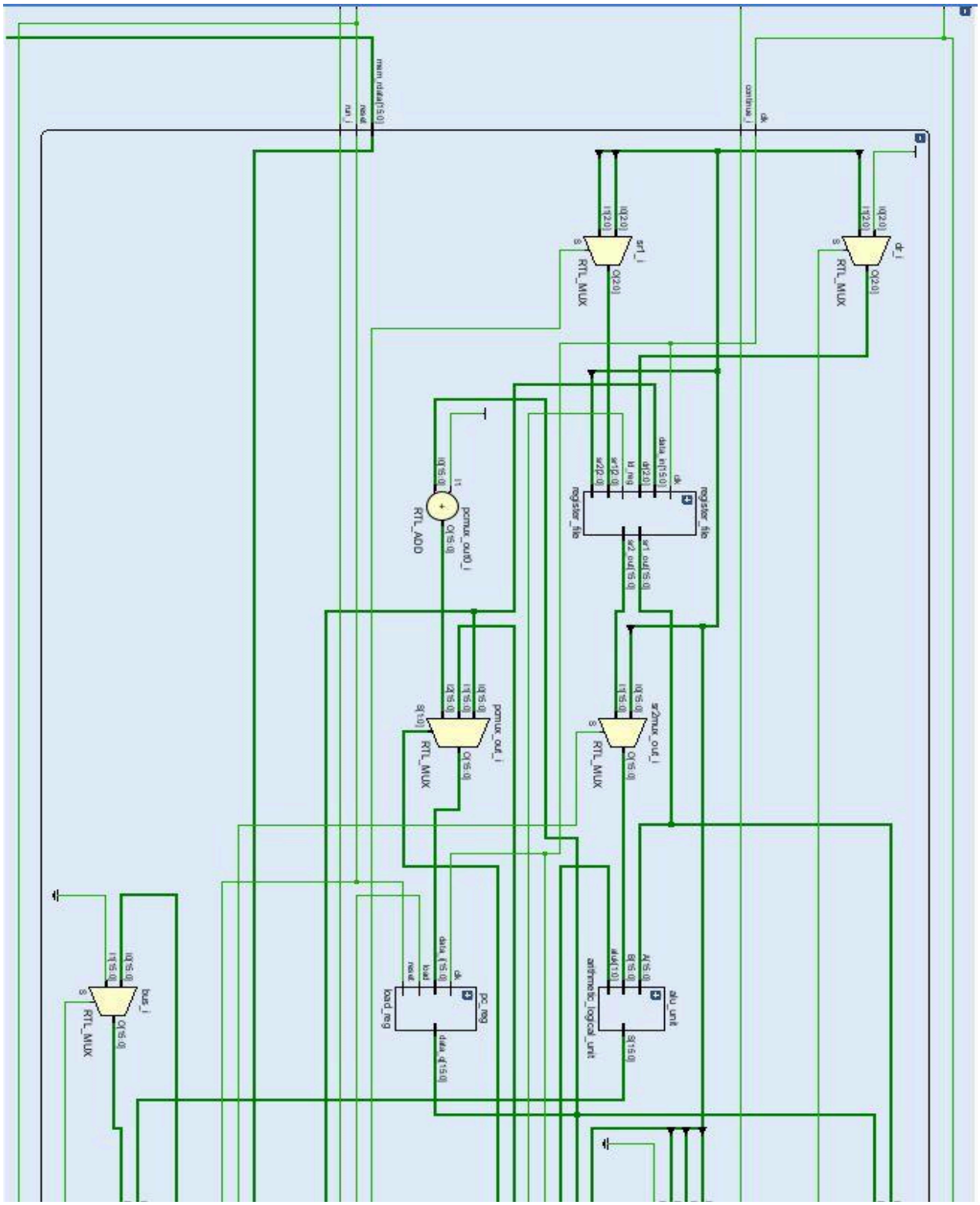
JSR is similar to JMP in that it involves writing a new value to the PC. The key differences are that the JSR instruction writes the current value of PC into R7 of the register file, and that the PC can only change by a maximum of PCoffset11 (obtained from sign extending IR[10:0]). The original PC is stored in R7 as a way to "save" that memory location in the event we want to access it later in the program.

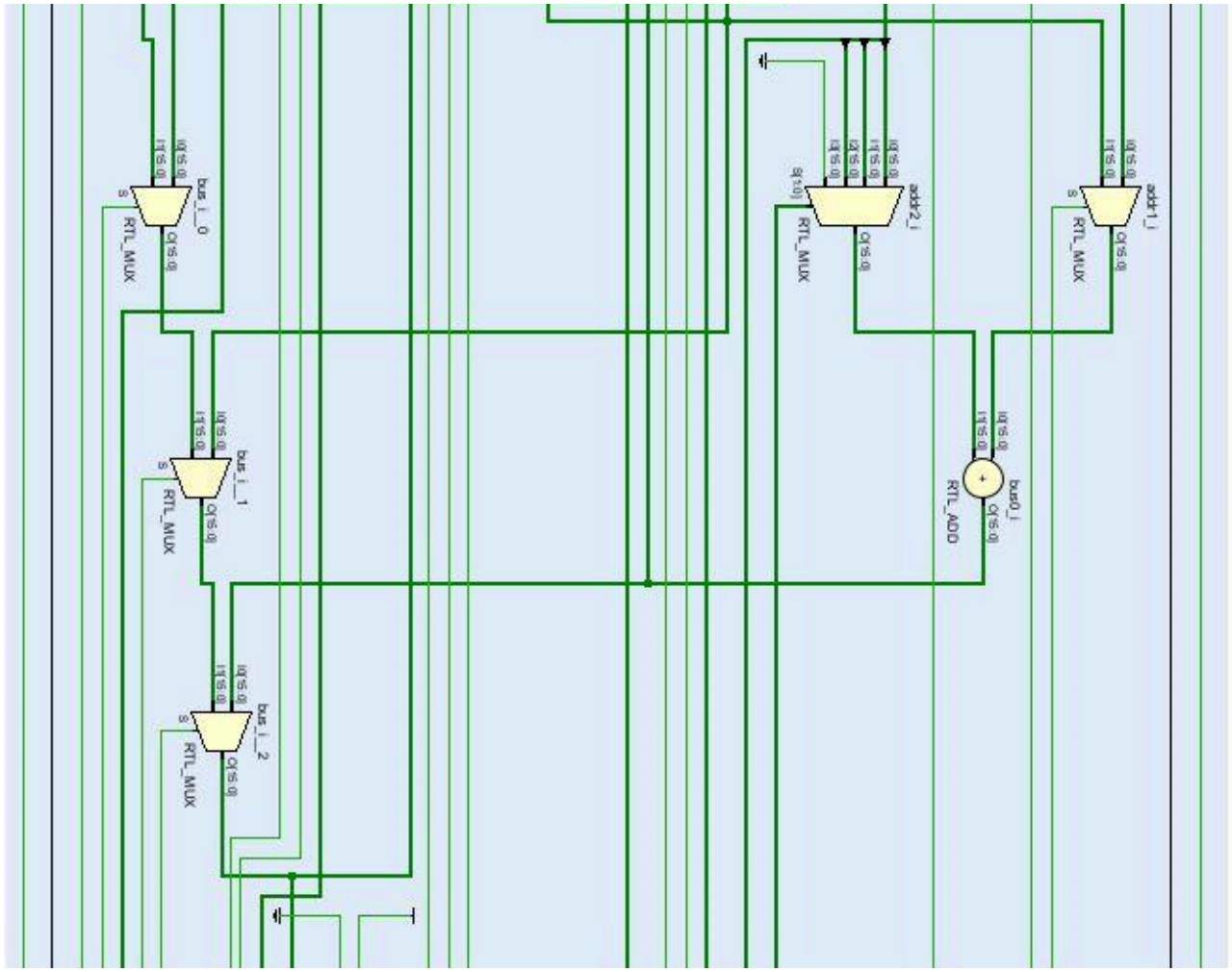
LDR involves reading the contents of the BaseR, adding to that value the value of offset6 (specified by IR[5:0]), and then accessing the contents of the memory address specified by the aforementioned addition and writing those contents into the DR.

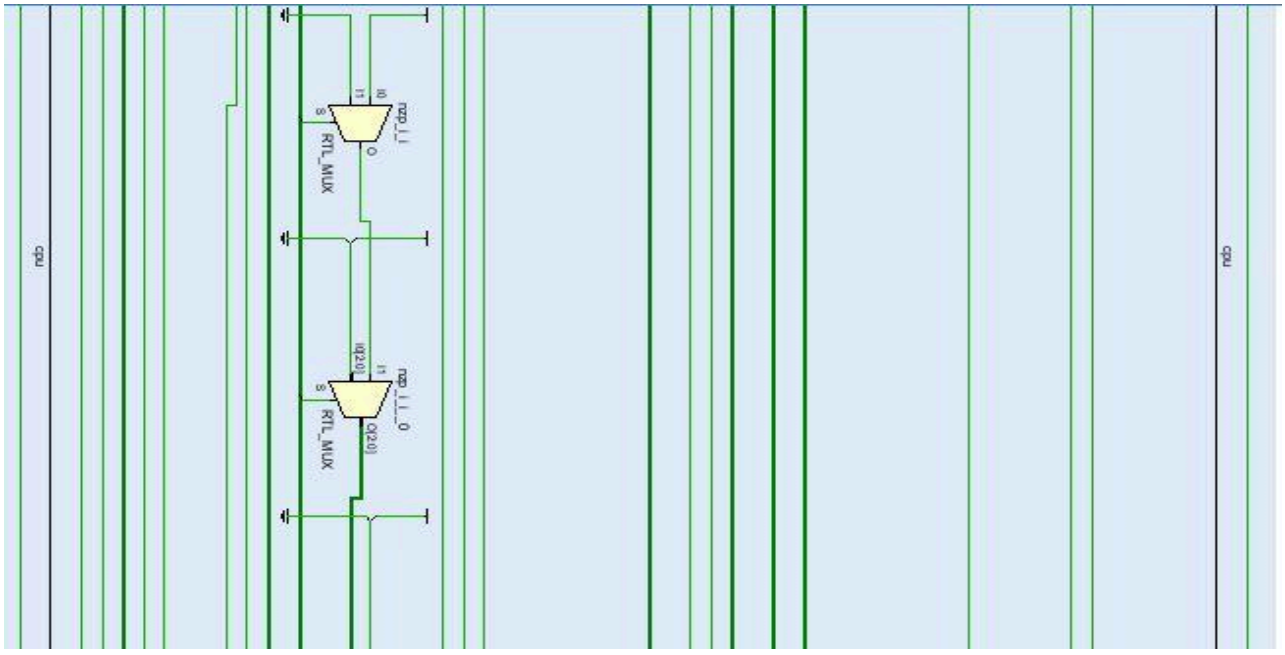
STR involves writing the contents of the SR into a memory address. The memory address that receives the data is determined by adding to the contents of the BaseR the value specified by sign extending IR[5:0].

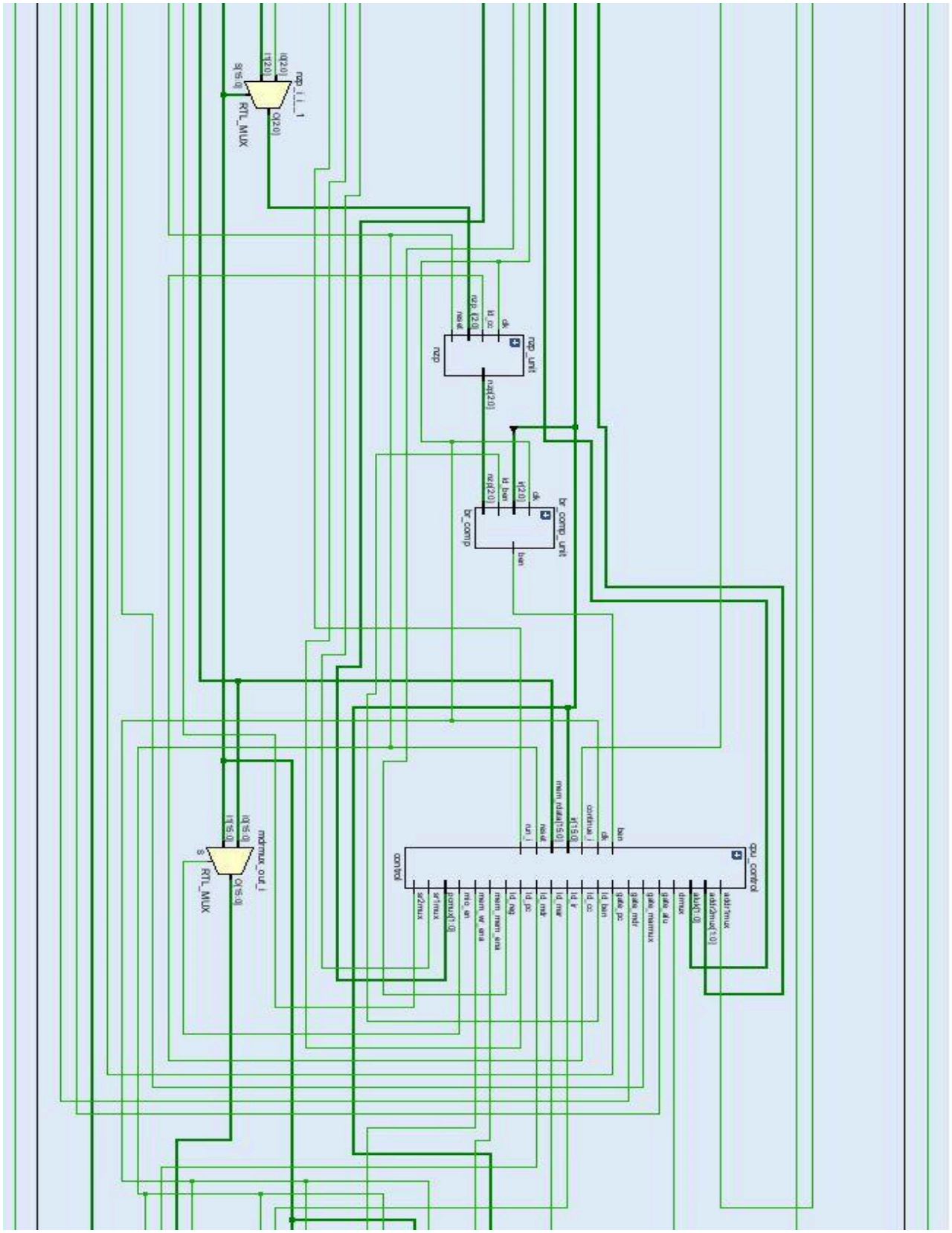
PAUSE simply stops the running of the program and activates the LEDs to match the 16-bit value currently specified by the toggle switches.

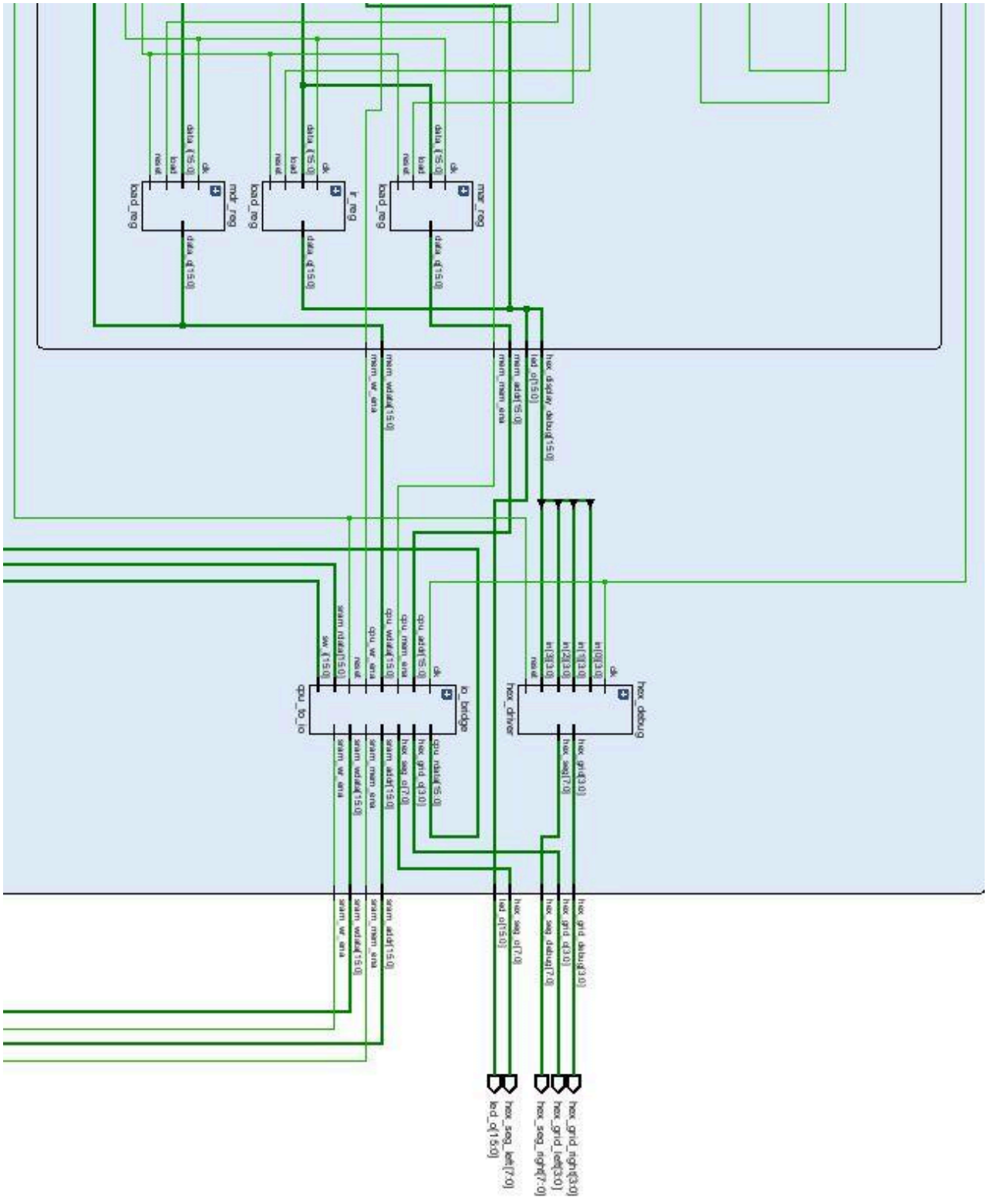
c. Block diagram of cpu.sv




addr1_i
addr2_i
bus0_i
bus_i__0
bus_i__1
bus_i__2
RTL_MUX
RTL_ADD


mux_i_i
mux_i_i_0
RTL_MUX
cpu


RTL_MUX
RTL_MUX
npc_unit
br_comp_unit
br_comp
cpu_control
control


mdr_reg
ir_reg
mar_reg
load_reg
data_d[15:0]
data_q[15:0]
clk
aclr
data[15:0]
hex_display_debug[15:0]
mem_addr[15:0]
mem_mem_ena
mem_wdata[15:0]
mem_wr_ena
cpu_addr[15:0]
cpu_mem_ena
cpu_wdata[15:0]
cpu_wr_ena
cpu_rdata[15:0]
io_bridge
cpu_to_io
sram_addr[15:0]
sram_mem_ena
sram_wdata[15:0]
sram_wr_ena
sw[15:0]
hex_seg_d[7:0]
hex_grid_d[3:0]
hex_driver
hex_debug
in0[3:0]
in1[3:0]
in2[3:0]
in3[3:0]
hex_seg[7:0]
hex_grid[3:0]
hex_seg_debug[7:0]
hex_grid_debug[3:0]
hex_seg_o[7:0]
hex_grid_o[3:0]
hex_seg_left[7:0]
hex_grid_left[3:0]
hex_seg_right[7:0]
hex_grid_right[3:0]
led_o[15:0]

d. Instruction Execution Datapaths

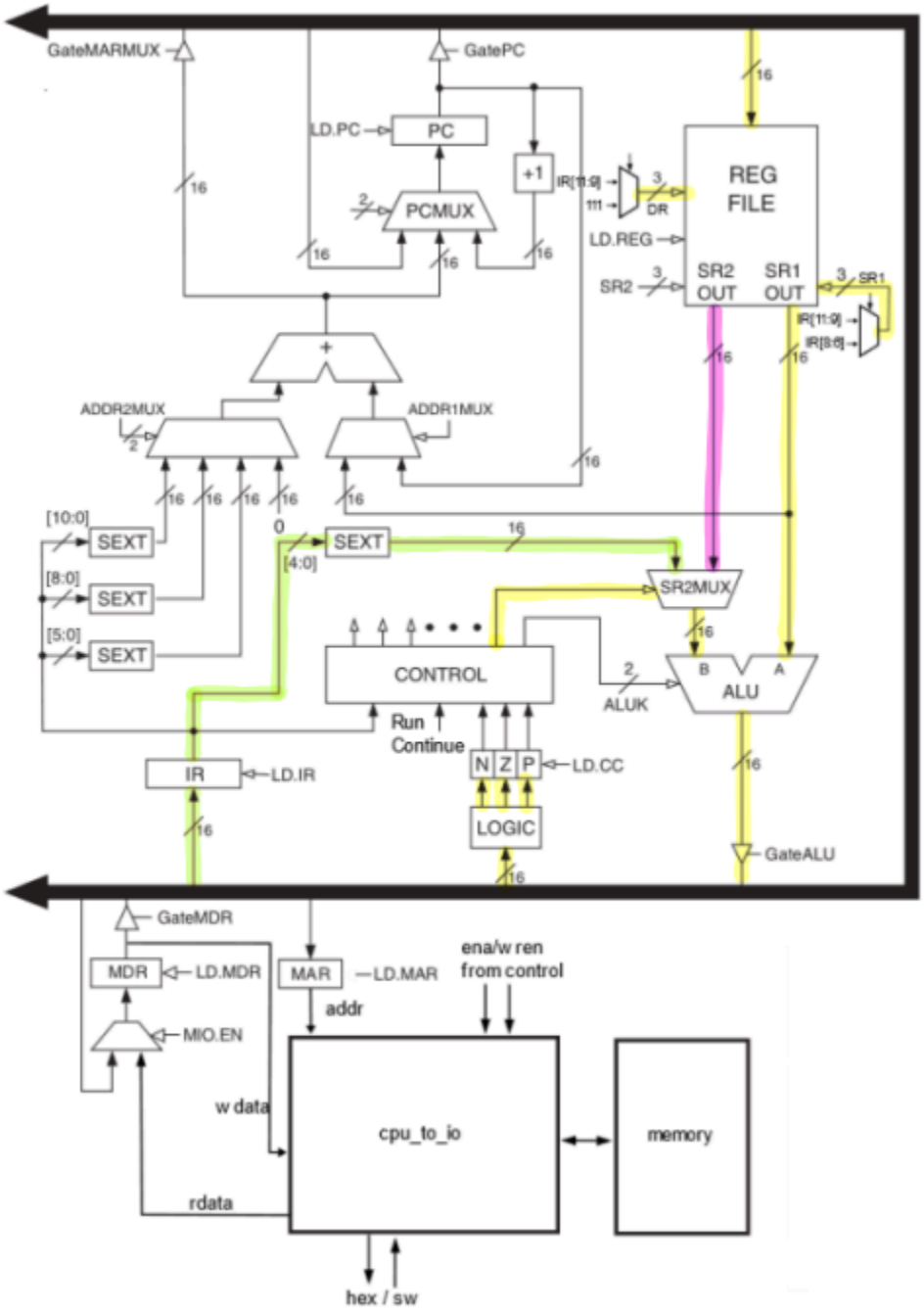


Figure 2: ADD, AND, and NOT Datapath

This is the datapath for the ADD, AND, and NOT instruction. The yellow indicates paths that will always be active, the purple represents the datapaths used if the second operand is SR2, while the green represents the datapaths used if the second operand is the sign extended [4:0] IR. Note that despite all the same

datapaths being active for NOT, it doesn't actually use the data inputted into the B input of the ALU.

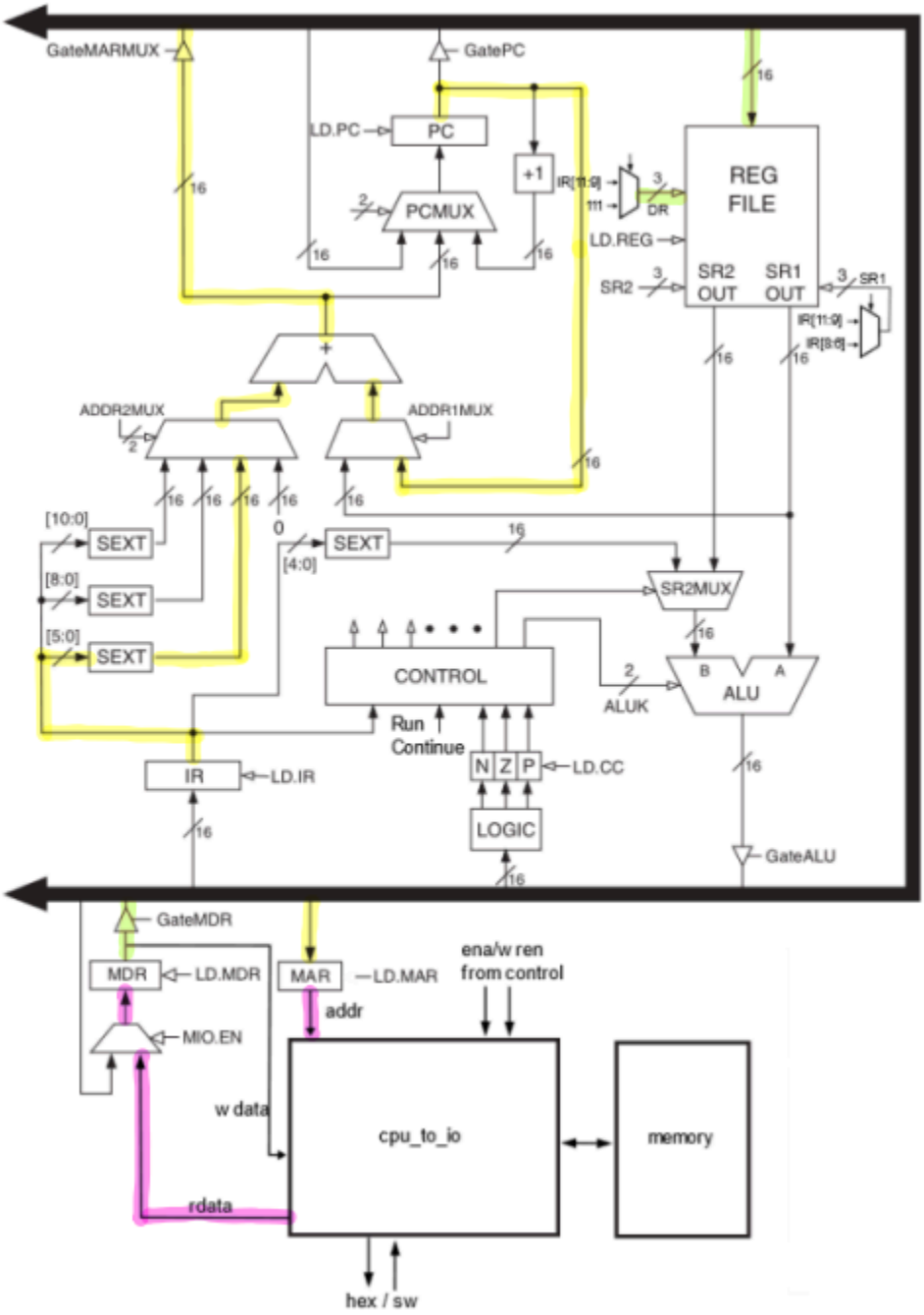


Figure 3: LDR Datapath

The yellow lines represent state 6, the magenta lines represents state 25, while the green lines represent state 27.

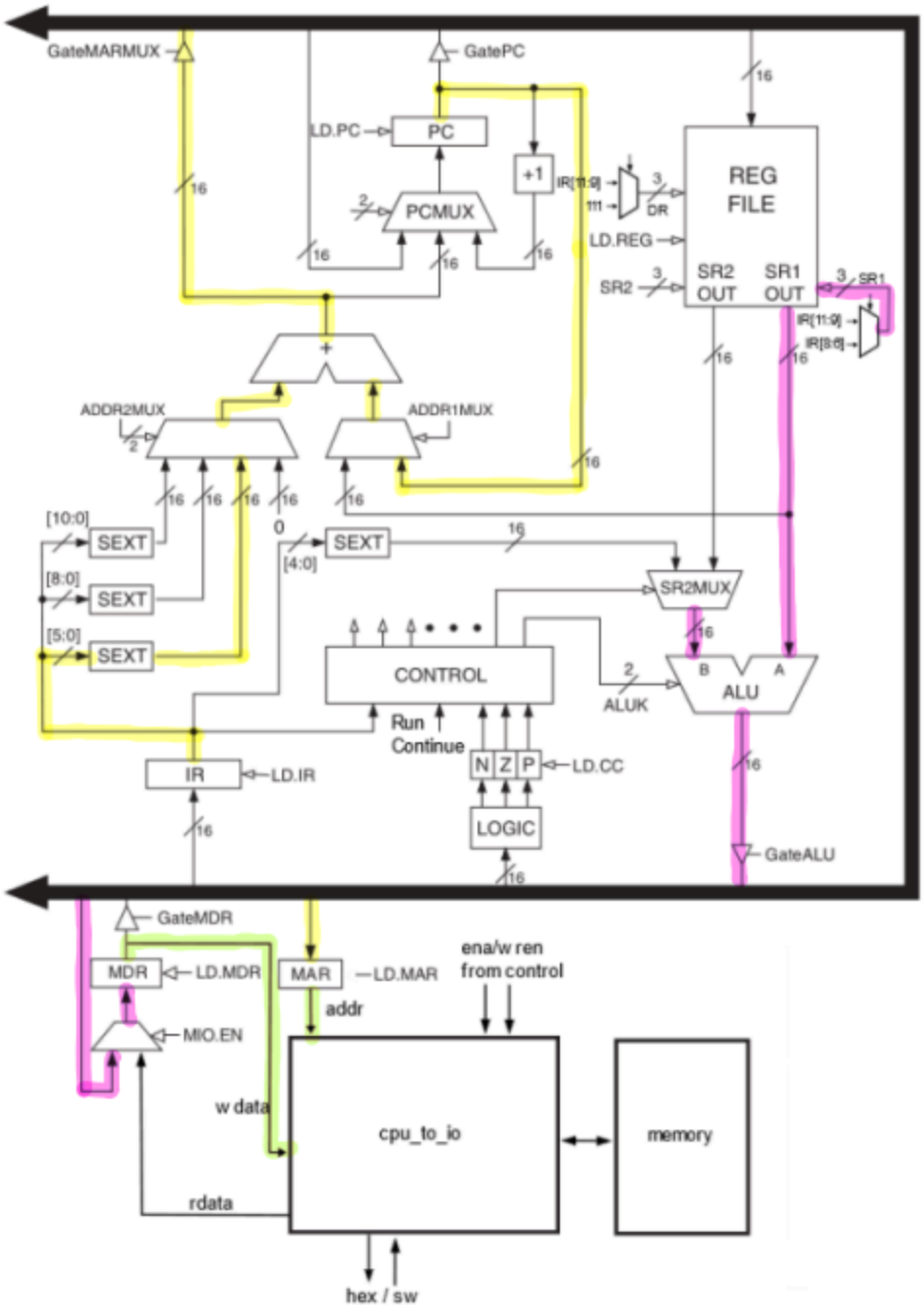


Figure 4: STR Datapath

The yellow lines represent state 7, the magenta lines represent state 23, and the green lines represent state 16.

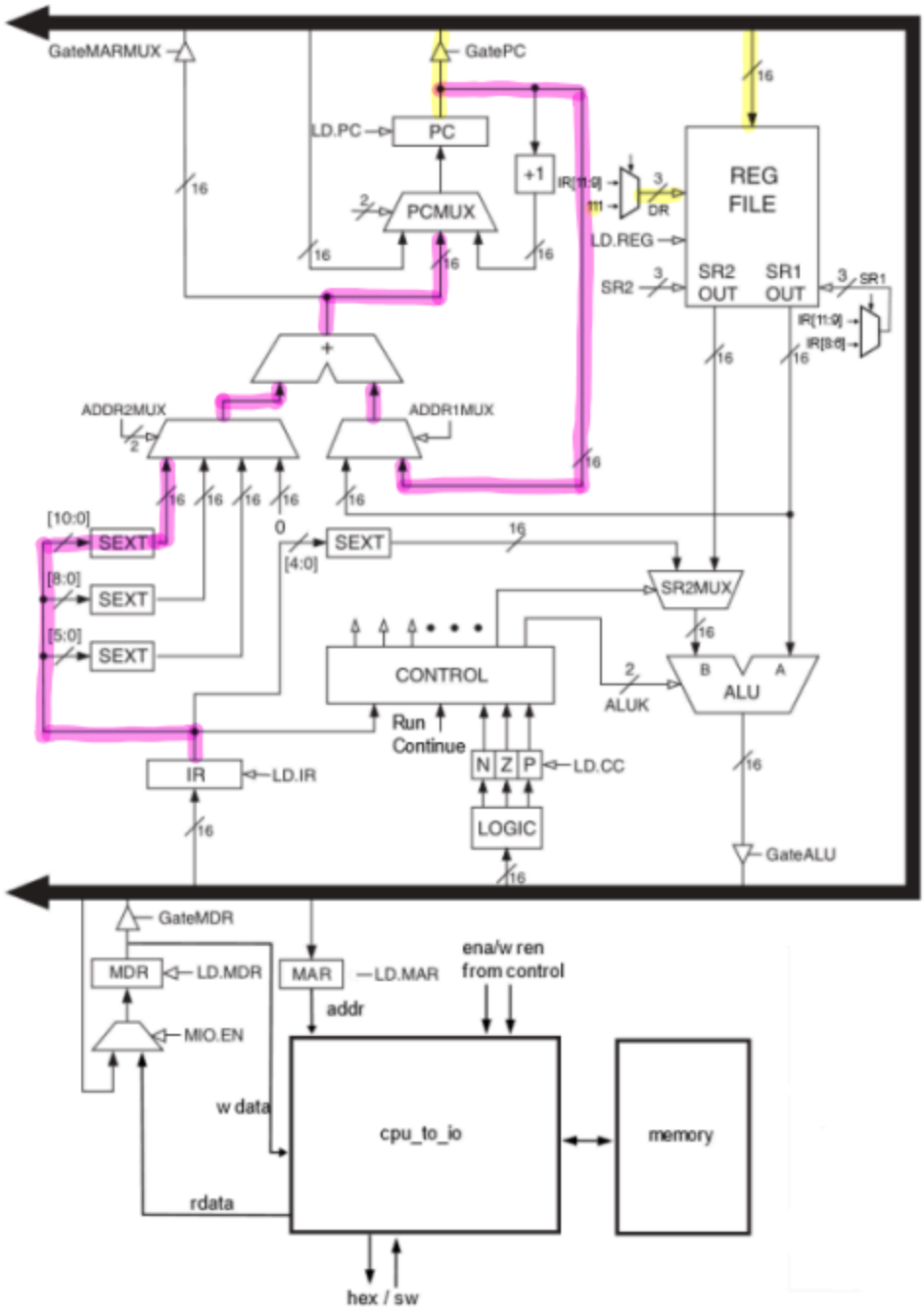


Figure 5: JSR Datapath

The yellow lines represent state 4, while the magenta lines represent state 21.

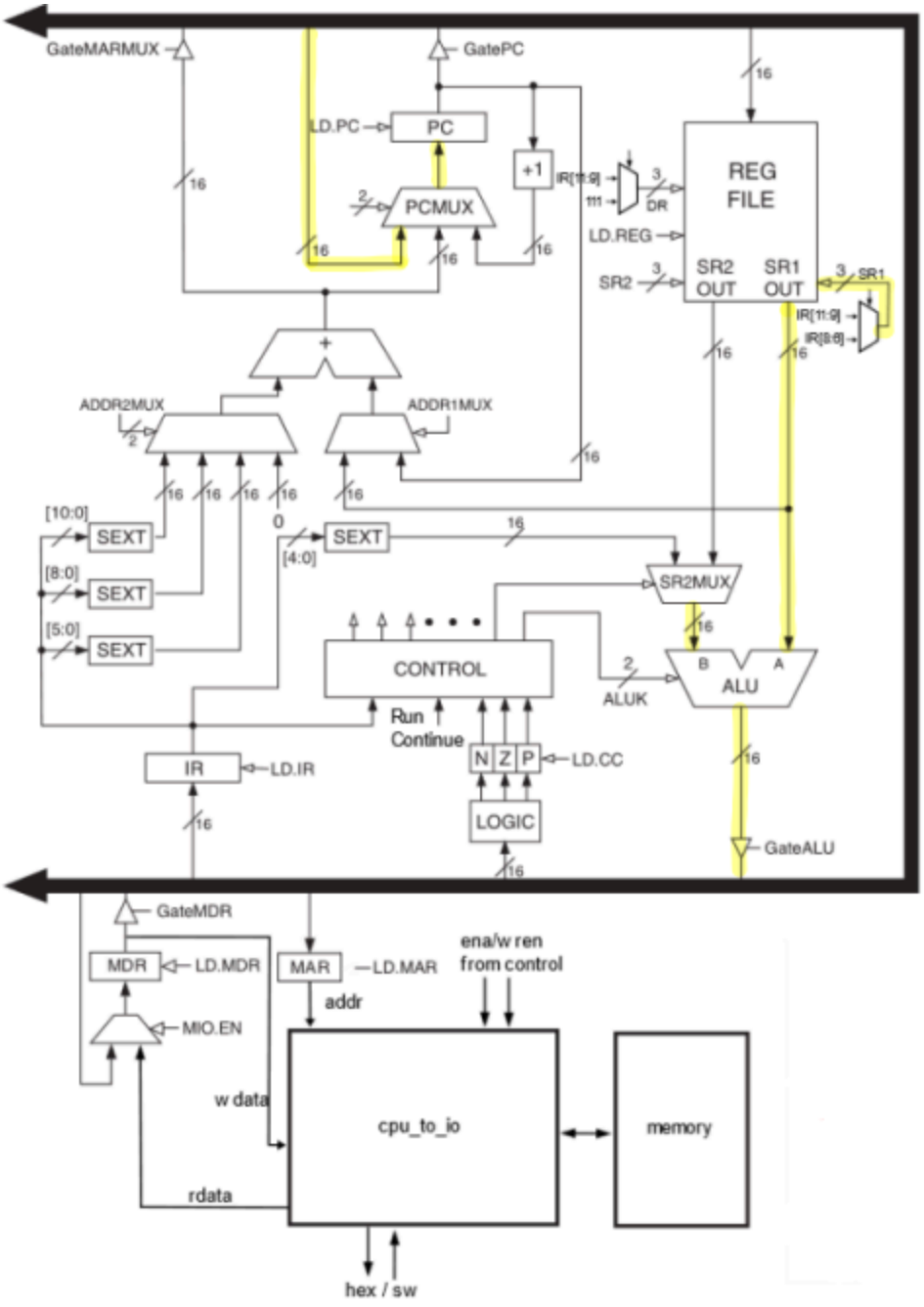


Figure 6: JMP Datapath

The yellow line represents state 12.

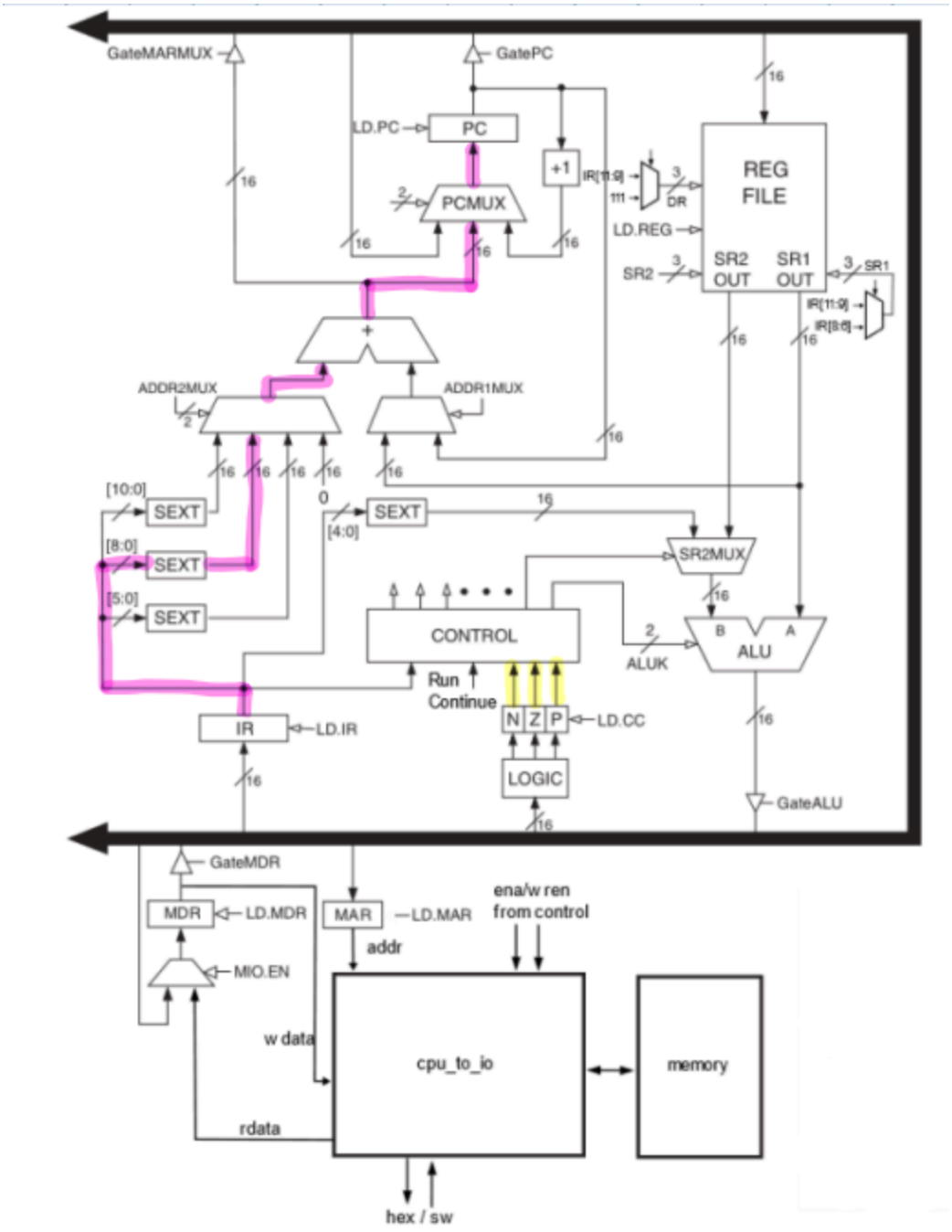


Figure 6: BR Datapath

The yellow line represent the control unit deciding if BEN is 0 or 1, state 0. If BEN is 1, state 22 will run, represented by the magenta line. If BEN is 0, then only state 0 will run.

e. Block diagram of the top-level (processor_top.sv)

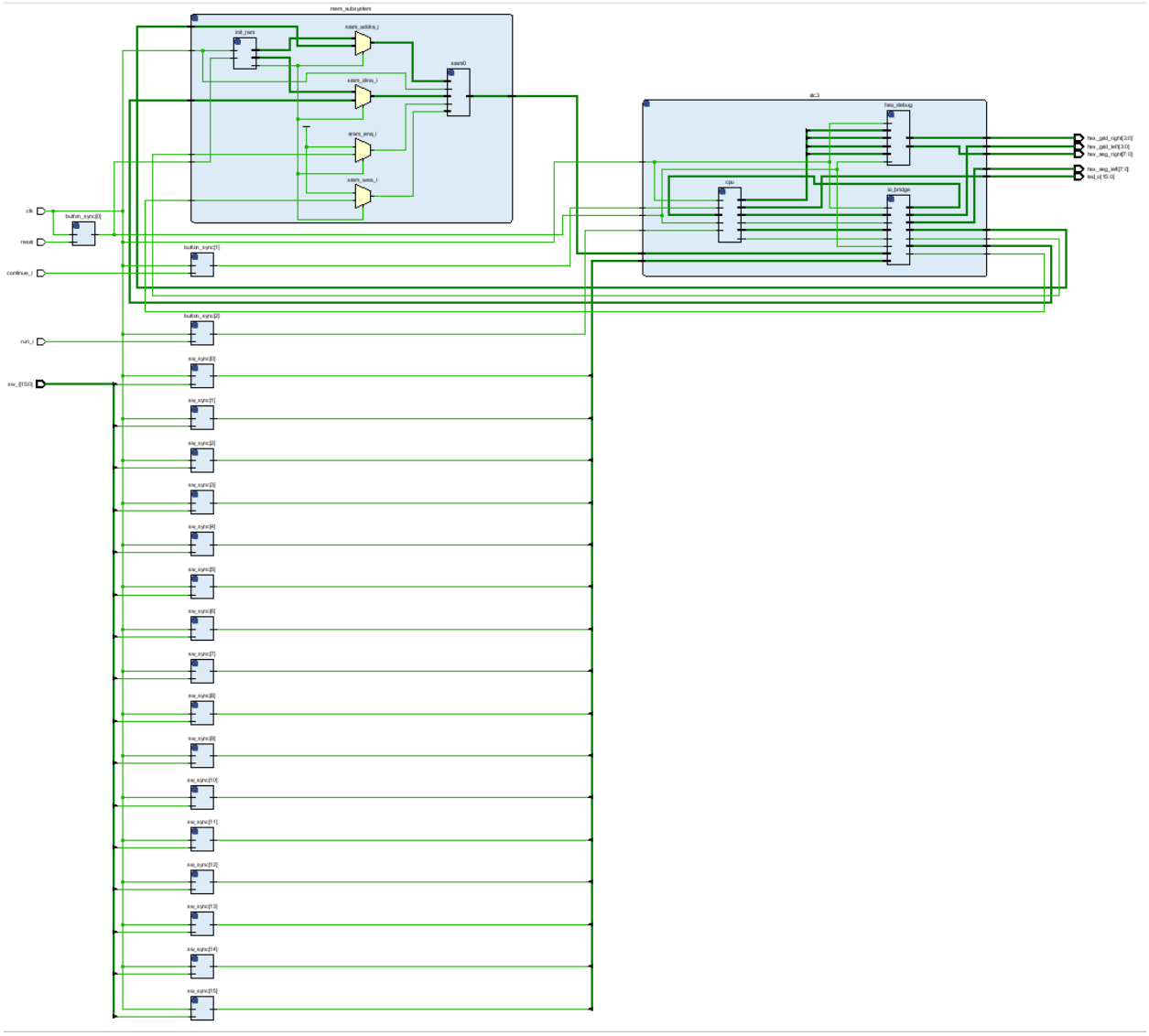


Figure 7: Processor Top Level

f. Written Description of all .sv modules

Module: processor_top.sv
Inputs: clk, reset, run_i, continue_i, [15:0] sw_i
Outputs: [15:0] led_o, [7:0] hex_seg_left, [3:0] hex_grid_left, [7:0] hex_seg_right, [3:0] hex_grid_right
Description: This module takes in our top level inputs and routes them through other modules, taking their outputs to be the top level outputs. It instantiates the sync_debounce, sync_flop, slc3, and memory modules.

Purpose: This module is our processor top level which takes in our external inputs and produces our external outputs.

Module: slc3.sv
Inputs: clk, reset, run_i, continue_i, [15:0] sw_i, [15:0] sram_rdata
Outputs: [15:0] led_o, [7:0] hex_seg_o, [3:0] hex_grid_o, [7:0] hex_seg_debug, [3:0] hex_grid_debug, [15:0] sram_wdata, [15:0] sram_addr, sram_mem_ena, sram_wr_ena
Description: This is the top level module for the slc-3 processor. It instantiates the cpu, cpu_to_io, and hex_driver module. This module is responsible for connecting the base cpu to the input-output controller cpu_to_io and to the output module hex_driver.
Purpose: This module is used to connect the base cpu to the input-output controller cpu_to_io and to the output module hex_driver.

Module: cpu.sv
Inputs: clk, reset, run_i, continue_i, [15:0] mem_rdata
Outputs: [15:0] hex_display_debug, [15:0] led_o, [15:0] mem_wdata, [15:0] mem_addr, mem_mem_ena, mem_wr_ena
Description: This is the slc-3 cpu module. It instantiates ir_reg, pc_reg, mar_reg, mdr_reg, register_file, alu_unit, br_comp_unit, nzp_unit, and control. This module connects the control unit's control signals to the other modules, as well as the other modules amongst themselves, allowing for the cpu to perform the desired instructions.
Purpose: This module is used to create the cpu we wish to implement.

Module: control.sv
Inputs: clk, reset, [15:0] ir, ben, [15:0] mem_rdata, continue_i, run_i
Outputs: ld_mar, ld_mdr, ld_ir, ld_ben, ld_cc, ld_reg, ld_pc, ld_led, gate_pc, gate_mdr, gate_alu, gate_marmux, [1:0] pcmux, drmux, sr1mux, sr2mux, addr1mux, [1:0] addr2mux, [1:0] aluk, mio_en, mem_mem_ena, mem_wr_ena
Description: This is the control unit, or FSM, of our cpu. It takes in control signals clk, reset, ben, continue_i, and run_i in order to control the control unit. It also takes the current instruction to be executed, [15:0] ir, to be used

to decide which instruction, and therefore which control signals, the control unit shall output.

This unit uses 3-always format, an always_ff to control next state transitions as well as assigning opcode to [15:12] ir, an always_comb to create the next state logic, and an always_comb to create the output logic.

As it pertains to the next state logic, the first state, s_18, only becomes the next state upon run_i high. This is what triggers our control unit to begin. It unconditionally moves through the setup states until reaching the decode state, state s_32. Here it decides on its next state based on the opcode, which was set as described above. It then moves to one of these states and follows through each state transition as shown in the state diagram section before returning to state s_18.

The output logic block works in tandem with the next state logic block, as it outputs control signals based on what the current state is. There are roughly four types of signals: load, gate, mux select, and enable. The load signals are used on all of our registers, where if the corresponding load signal is high, the register will load the value on its input. The gate signal is used for the pc, mdr, alu, and marmux. These four all share a common bus, and hence the gate signal specifies which of these modules will output to the bus at a time. Since many of our modules need to perform different tasks, often the input to these modules can vary. Hence, many of our modules have a mux on its input that can select between different inputs as desired by the cpu. As such, the mux signals are what decide what inputs each mux output which then inputs to the desired module. Finally the enable signals are specifically used to enable devices which need an enable signal. In our case, this is only our memory module which requires enable signals to read from or write to it.

Purpose: This control unit is needed as the FSM used to control our slc-3 computer.

Module: register_file.sv
Inputs: clk, reset, [2:0] dr, [2:0] sr1, [2:0] sr2, ld_reg, [15:0] data_in
Outputs: [15:0] sr1_out, [15:0] sr2_out
Description: We used behavioral HDL in the creation of our register file. An unpacked array representing eight distinct 16-bit registers is created via

internal logic; the packed dimension determines the bit width of each register, and the unpacked dimension determines the number of registers we'd like to create. The packed and unpacked dimensions are thus [15:0] and [8] respectively. Given that these registers are synchronous, an always_ff procedural block is used to determine exactly how and under which conditions the contents of any given register are to be moved and/or altered. If reset is pressed, the contents of each register are cleared in ascending order starting with R0. If the ld_reg signal is high, the destination register is loaded with the contents of data_in. We also use single-line assignments to set the outputs sr1_out and sr2_out to the contents of registers sr1 and sr2 respectively.
Purpose: Instantiating this module within the cpu creates the eight local registers R0 – R7 whose purpose is to store data locally so that the cpu can perform the various operations specified by the instructions. It's necessary to store intermediate data locally in the cpu because it is common for sequences of instructions to take more than one clock cycle, and the data would be lost if we could not store it in some kind of local memory.

Module: arithmetic_logical_unit.sv
Inputs: [1:0] aluk, [15:0] A, [15:0] B,
Outputs: [15:0] S
Description: This module takes in 16-bit inputs A and B, and performs an operation on them, giving the result as output S. The operation it performs is based on the value of the aluk input signal, performing A + B, A & B, ~A, or just passing A for aluk being 00, 01, 10, and 11, respectively.
Purpose: This module is our main module to perform most logical and arithmetic calculations needed, particularly during the ADD, AND, and NOT instruction.

Module: br_comp.sv
Inputs: clk, [2:0] nzp, [2:0] ir, ld_ben
Outputs: ben
Description: This module creates our ben, short for branch enable, signal. It does this based on comparing [2:0] ir to [2:0] nzp. If any of the places of the two are both 1, then ben = 1, else ben = 0.

Purpose: This module is specific to when the control unit is in the branch state. When in such a state, whether the branch instruction occurs or not depends on the input signal ben. This signal, ben, is what our br_comp module creates.

Module: nzp.sv
Inputs: [2:0] nzp_i, clk, reset, ld_cc
Outputs: [2:0] nzp
Description: This module is a three bit register used to store condition codes nzp. It does this by setting nzp to nzp_i if ld_cc is high. Additionally, if reset is high, then nzp is set to 3'b000. If neither are true, then nzp just holds its current value.
Purpose: This module is used to hold our nzp condition codes. It doesn't create the nzp values themselves, as the cpu module does this for it, sending the value to nzp_i.

Module: cpu_to_io.sv
Inputs: clk, reset, [15:0] cpu_addr, cpu_mem_ena, cpu_wr_ena, [15:0] cpu_wdata, [15:0] sram_data, [15:0] sw_i
Outputs: [15:0] cpu_rdata, [15:0] sram_addr, sram_mem_ena, sram_wr_ena, [3:0] hex_grid_o, [7:0] hex_seg_o
Description: This module routes the inputs and outputs to and from the cpu depending on if we wish to access the SRAM, or if we wish to access our peripherals. It does this by checking if cpu_addr is 16'hffff. If so, then we want to access peripherals, either the switches in the case of an input read, or the hex display in the case of an output read. If not, then we want to use the SRAM, and the cpu_addr is used as such to read or write from that memory address.
Purpose: The purpose of this module is to seamlessly read and write to either the memory or input-output. This module makes it such that accessing peripherals and accessing memory works in the same way, meaning no extra logic needs to be used to access such peripherals. It works on the principle of memory mapping, in which a certain memory address is used to access our peripherals, such that when we want to do so, reading and writing to that

mapped memory address will read inputs from or write outputs to the mapped peripheral.

Module: memory.sv
Inputs: clk, reset, [15:0] data, [9:0] address, ena, wren
Outputs: [15:0] readout
Description: This module decides if we are using the actual memory, instantiate_ram.sv, or the simulated memory, test_memory.sv. It does this by instantiating one or the other as the memory our slc-3 uses depending on if we are not simulating or if we are simulating.
Purpose: This module is necessary to correctly switch between both simulated and real memory depending on if we are testing in simulation, or if we are actually running the slc-3 on the Urbana board.

Module: instantiate_ram.sv
Inputs: reset, clk
Outputs: [9:0] addr, wren, [15:0] data
Description: This module creates an instance of the IP SRAM module and wires it such that we can communicate with it to read from and write to it as our slc-3 module desires.
Purpose: This is what allows us to use the onboard SRAM modules on the Urbana board during use.

Module: test_memory.sv
Inputs: clk, reset, [15:0] data, [9:0] address, ena, wren
Outputs: [15:0] readout
Description: This module creates simulated memory and loads in our memory file to be used by the slc-3.
Purpose: The purpose of this module is to simulate the onboard urbana memory module when in simulation. This is because in using the Vivado simulator, we are unable to access the onboard Urbana memory module, so we need to simulate it for the simulation.

Module: load_reg.sv
Parameters: DATA_WIDTH

Inputs: clk, reset, load, [DATA_WIDTH-1:0] data_i
Outputs: [DATA_WIDTH-1:0] data_q
Description: This is a parameterized size input/output register that loads input into the flip flop of load is high, sets the flip flop to all 0's if reset is high, or holds the current value if both are low.
Purpose: This module is used for all of our single use registers (except for the special case of nzp) such as pc, mar, mdr, and ir.

Module: hex_driver.sv
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: This module takes in 4x4 inputs and converts it to four hexadecimal values to output. It does this by mapping the 4x4 inputs to output signals hex_seg and hex_grid. Output hex_seg controls the eight anode signals of each individual LED in each 8-led figure 8 module, while hex_grid controls the shared cathode signal for said 8-leds. It sequentially changes such signals as to make four of these modules appear to be lit up at once outputting the four hexadecimal values corresponding to input [3:0] in[4].
Purpose: This module is used twice, once to display the current [15:0] ir for debugging purposes, and a second time to use as our output display when such is needed.

Module: sync_debounce.sv
Parameters: COUNTER_WIDTH
Inputs: clk, d
Outputs: q
Description: This circuit takes in input d, and only outputs q = d once 2^(COUNTER_WIDTH) occurs and on the posedge of the clock. This circuit ensures the input is debounced, and that its change is synchronized with the clock.
Purpose: This is used to sync and debounce all of our switches and buttons. In this case, this is used for sixteen switches, and three buttons.

g. Description of the operation of the control unit

Refer to the control.sv module description.

h. State Diagram of Control Unit

Shown below is the state diagram for the slc-3 we implemented. Notice how certain states are actually three sub-states that do the same thing. This is because of the three clock cycle till ready memory requirement for when we wish to access our memory.

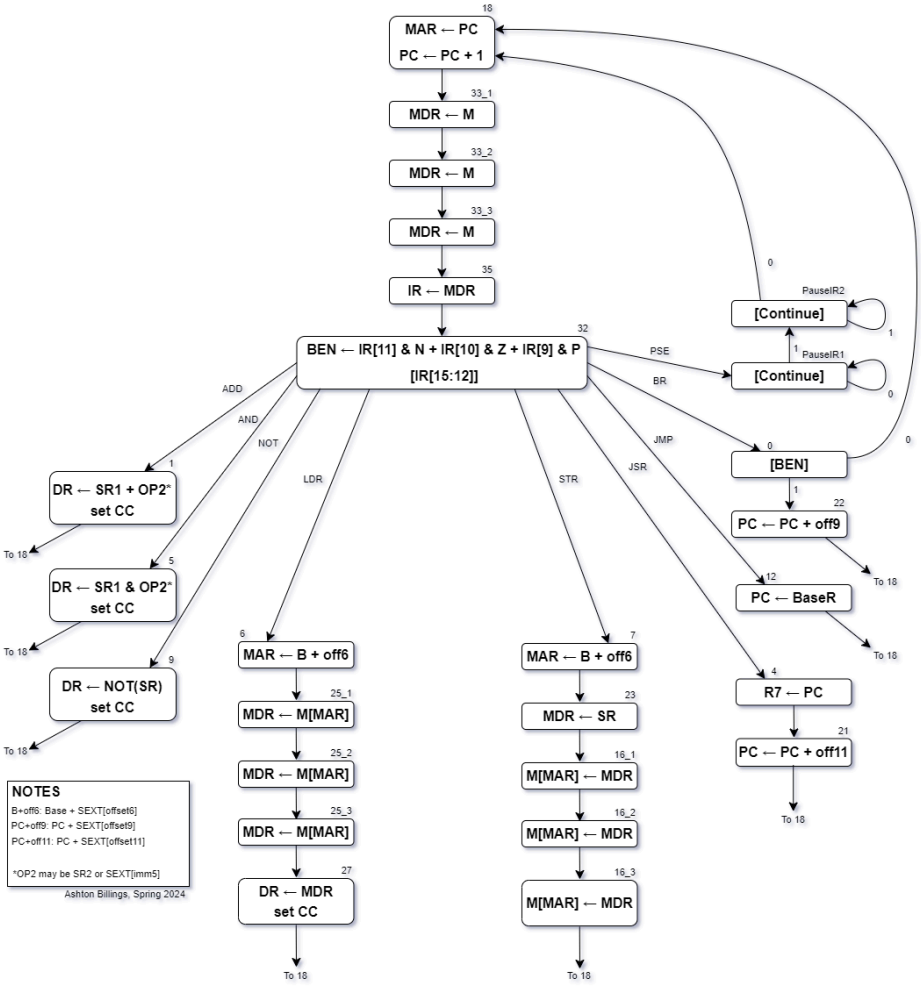


Figure 8: SLC-3 State Diagram

3. Simulations of SLC-3 Instructions

Shown below are the waveforms for each test program, highlighting the inputs and outputs, as well as relevant intermediate values. Note for the bubble sort waveform that due to its size, it had to be split into three separate images.

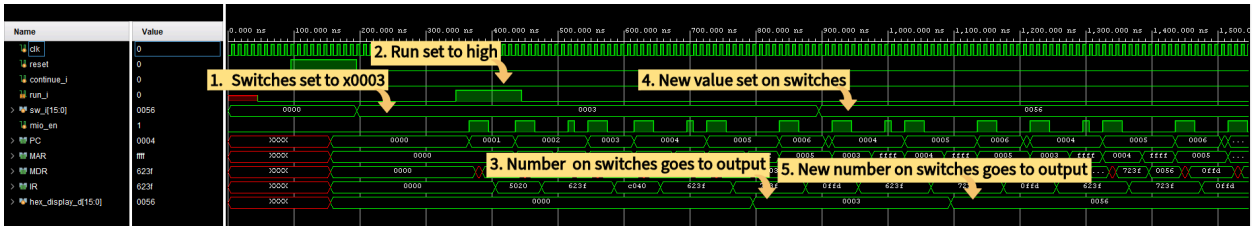


Figure 9: IO Test 1 Program Waveform

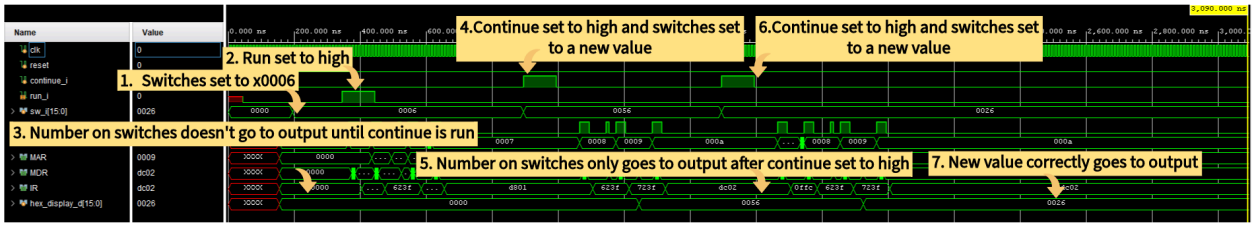


Figure 10: IO Test 2 Program Waveform

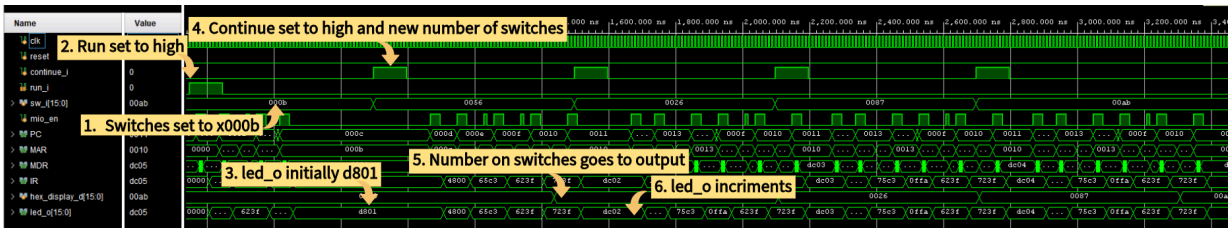


Figure 11: Self-Modifying Code Program Waveform

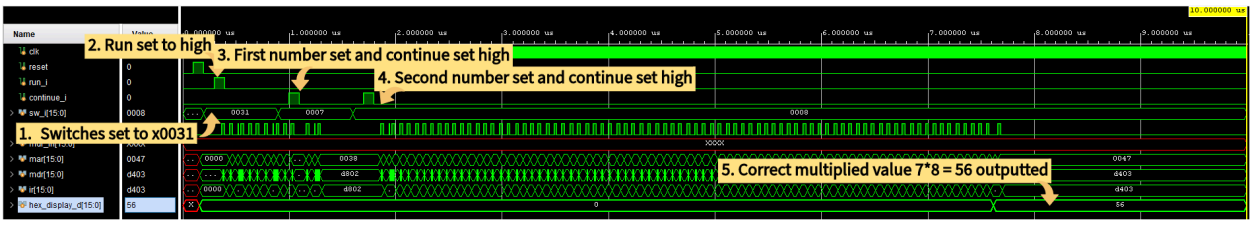


Figure 12: Multiplication Program Waveform

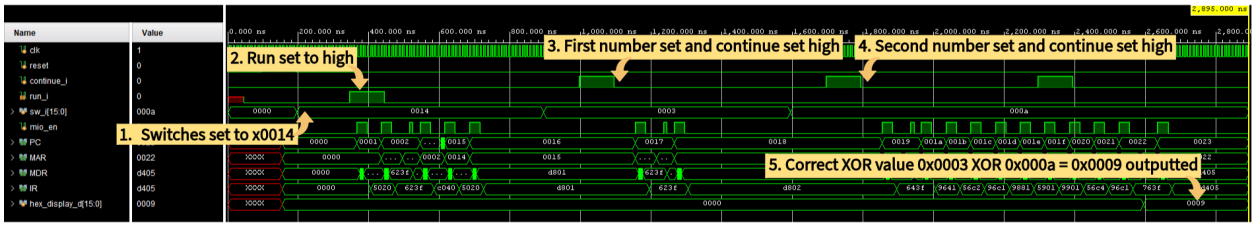


Figure 13: XOR Program Waveform

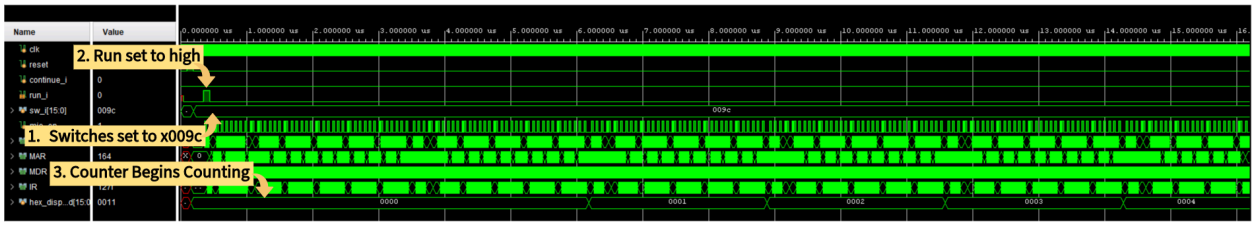


Figure 14: Counter Program Waveform

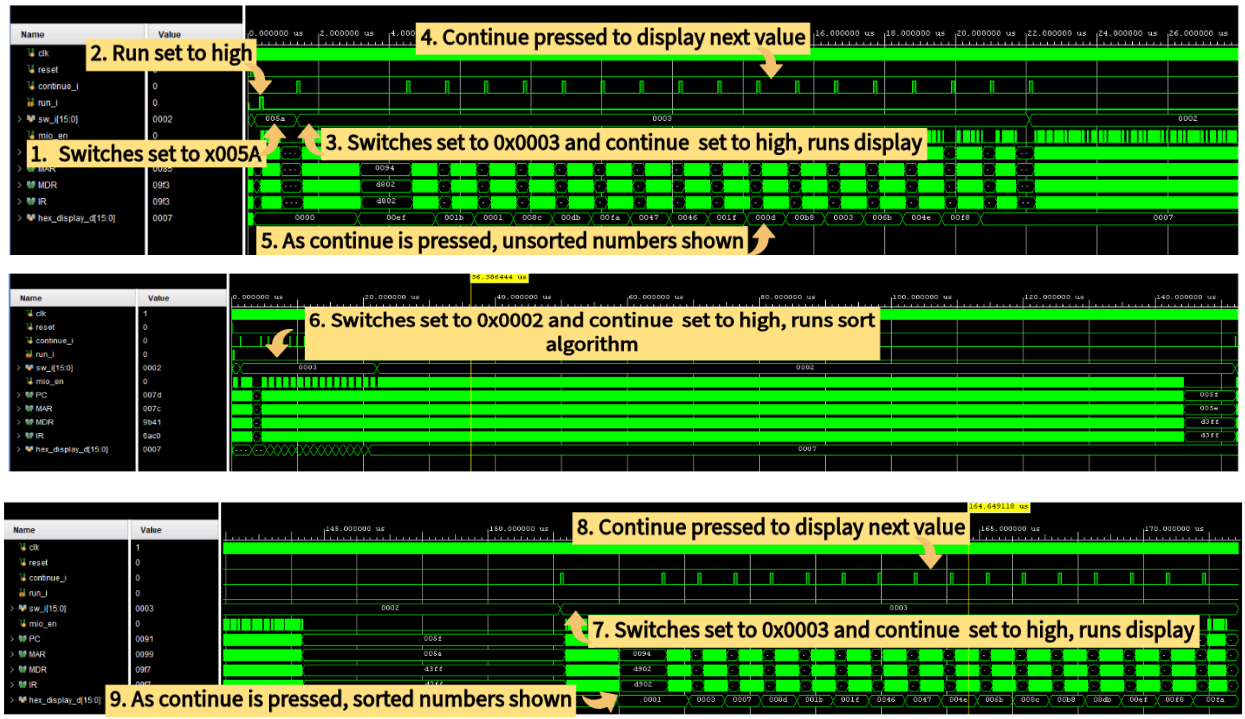


Figure 14: Bubble Sort Program Waveform

4. Post-Lab Questions

a. Fill out the Design Resources and Statistics table.

LUT	415
DSP	0
Memory (BRAM)	1
Flip-Flop	362

Latches*	0
Frequency	105.29 MHz
Static Power	0.071 W
Dynamic Power	0.010 W
Total Power	0.081 W

Document any problems you encountered and your solutions to them, and a short conclusion.

The only major problems we encountered stemmed from us leaving out a few critical logic components in our initial design. We forgot to add the BR_comp module in our first pass through; we also erroneously swapped the selector bits on essentially all our muxes. We were fortunately able to debug all these errors using simulated waveforms.

b. Answers to Post-Lab Questions

i. What is CPU_TO_IO used for, i.e. what is its main function?

The cpu_to_io module was used as an interface between the cpu block, the memory block, and the peripherals. This block allowed us to write information to and read information from memory mapped (to xFFFF) switches and hex displays.

ii. What is the difference between BR and JMP instructions?

BR is the branch instruction and JMP is the jump instruction. Jump is unconditional, meaning the program is guaranteed to ‘jump’ to whichever address is specified by the contents of the base register. Branch can be either conditional or unconditional. In the case where we want a branch instruction to be conditional, we would set bits [11:9] IR to 111. Otherwise we would choose which of the bits of nzp we want to branch off of with regards to negative, zero, or positive.

Another difference between BR and JMP is the maximum range of the movement of the program counter. JMP allows the program counter to access a greater maximum range of movement as JMP uses 11 bits, meaning it can access 2^11 memory locations, while BR uses

9 bits, meaning it can only access 2^9 memory locations. Therefore, although we can use BR for unconditional jumps, it makes more sense to use JMP in the case of greater jumps in memory location.

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 impact does this have on performance?

The R signal stands for 'Ready' and it is used as a way to signify when the memory is ready to be accessed/written to. We did not implement this signal into the control unit of the SLC-3, instead we incorporated two additional wait states for every state that is reading from/writing to memory. See the modified SLC-3 state diagram above for a visual. This impacted the performance negatively because even if the memory is read to be read from/written to, we still have to wait for the entire duration of the three total state changes.

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

Fortunately everything in our SLC-3 design worked correctly. Even though the debugging process was tedious and at times difficult due to the sheer quantity of signals that depended on other signals, the bugs themselves were not particularly difficult to eliminate from our design. The bulk of the errors stemmed from mux signals being wired in the exact opposite order they should have been, which was annoying but was also a by-product of our design.

   b. **Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it doesn't get changed.**

Including the SLC-3 datapath and state diagram in the lab document was immensely helpful. Additionally, the inclusion of the instruction set formats and their corresponding RTL lines was helpful in debugging as it allowed us to quickly confirm what certain signals should be displaying based on which instruction was currently being executed. Although there was a bit of disappointment that the control unit was done for us, its

inclusion was helpful for time's sake as if we needed to implement it ourselves it would take another weeks worth of work.