

ECE 385

Spring 2017

Final Project

PSLC-3:

**A Proof of Concept for Symmetric
Multiprocessing**

Jack Hu & Landon Clipp

yhu49 & clipp2

Section ABC

Introduction

In lab 6 we implemented the SLC-3 with SystemVerilog to apply our CPU architecture knowledge learned from ECE 120. Although the SLC-3 was a great educational CPU architecture, it is nowhere like a modern CPU in many ways. While the latest consumer CPUs offer up to eight cores to perform 64 bit operations simultaneously, the original SLC-3 and LC-3 are only single core, meaning that they must perform operations serially at all time. This does not matter for most of the common computational tasks, since most tasks only occupy one core at any given time, but what if more performance is needed? The most straight forward answer is just to crank up the maximum operational frequency to increase the clock speed. In real world applications however, this is not always possible. A higher clock speed dissipates significantly more heat and consumes more power, making it hardly the preferred solution. Another answer to this dilemma is to develop a multicore platform on top of the SLC-3/LC-3 architecture and extend the existing instruction set architecture to support parallel programming. This solution is particularly efficient when performing easily separable tasks, such as writing to a large set of arrays. After sorting out our ideas, we have developed a multicore version of SLC-3, we call it Parallel SLC-3. PSLC-3 has five SLC-3 CPUs, four of which are compute nodes that perform the computational tasks, and one is the master node that distributes instructions as assembly code dictates. Because the vast similarities between SLC-3 and PSLC-3 in the lower level, this report will not describe much of the basic SLC-3 features, but to focus on features unique to PSLC-3.

Description of the Operation of the Circuit

As far as a programmer is concerned, minimal consideration is needed to change the code for SLC-3 to fit our architecture for PSLC-3. A lot of work spent in modifying the architecture involves minimizing changes we make to the current infrastructure. More specifically, the main structure for each of the five cores is very similar to the original SLC-3. The biggest change is a secondary decode state dedicated to parallel specific instructions (PSI), which replaces the PAUSE

(1101) state from lab 6.1; consequently, the programmer must write some code using these instructions to initialize parallel tasks.

To achieve functionalities described above and to simplify the parallel structure, we need a mechanism to synchronize memory I/O operations. This mechanism is expected to prevent race conditions between different cores by using a queue structure. In addition, an SRAM controller with a tristate buffer, similar to that from lab 6 with a MUX was added for routing data for different cores. The high-level block diagram for the interconnections is presented below.

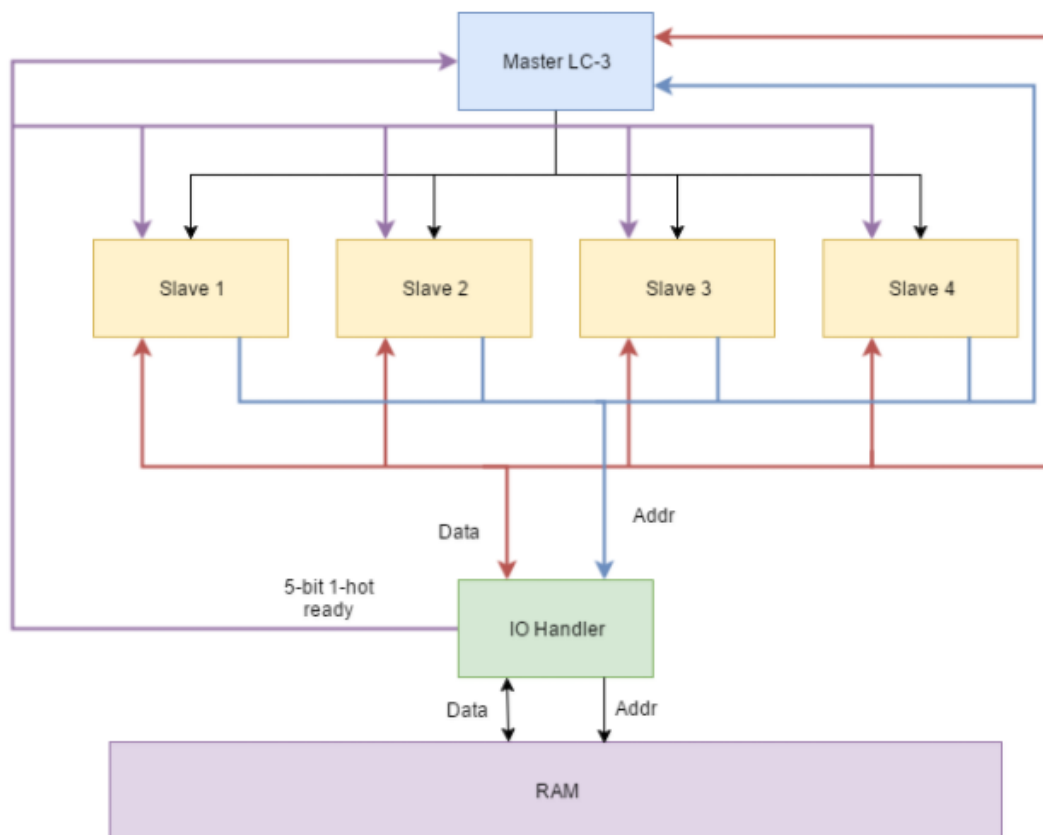


Figure 1 Highly Level Block Diagram

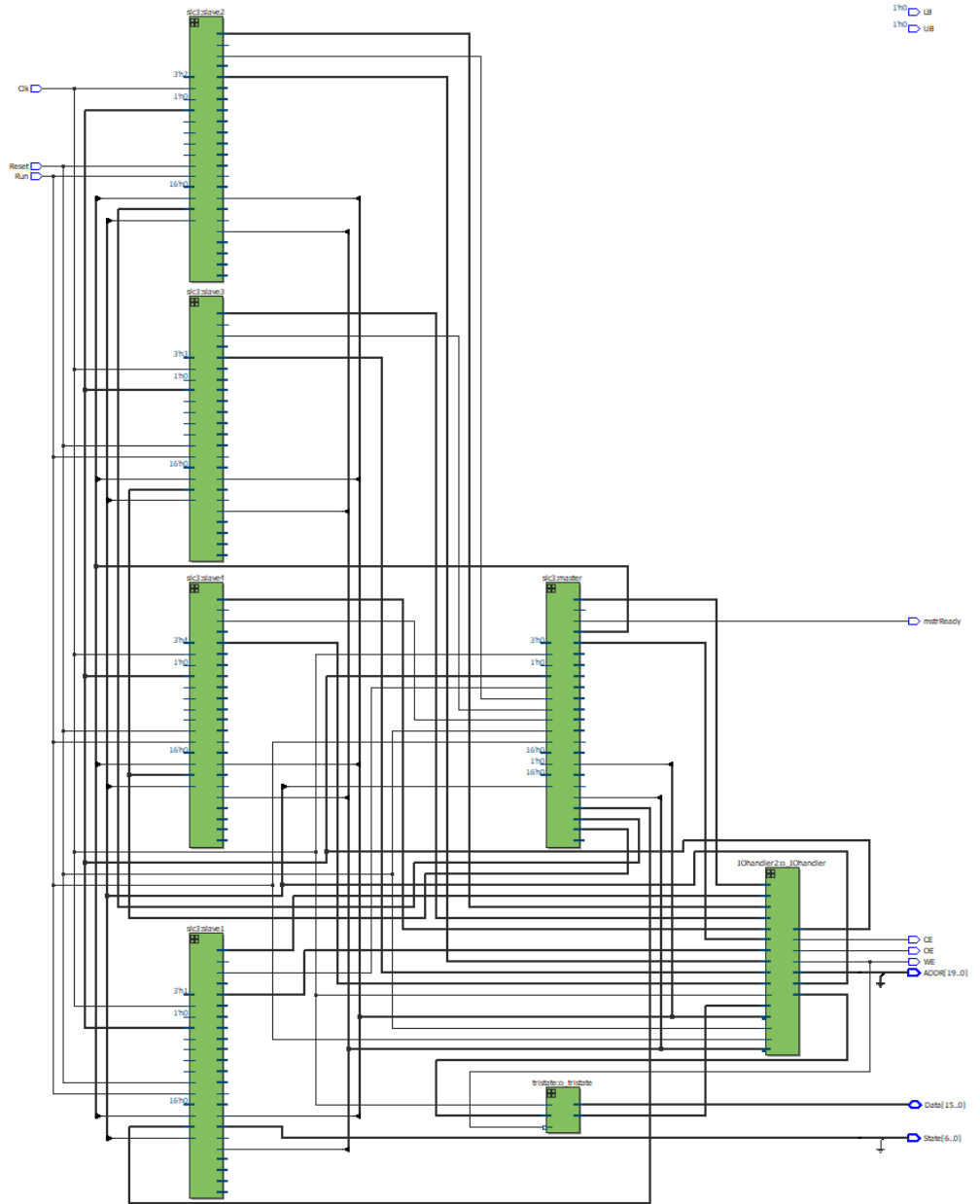


Figure 2 Top level RTL Viewed in Quartus

To accommodate the parallel operations, a number of hardware modifications were made. One thing to note is that all five CPUs share the same hardware architecture to simplify the design process, so some of the components necessary in the compute nodes might be redundant in the master nodes and vice versa. Four dedicated registers PC_1, PC_2, PC_3, and PC_4 are used in the master CPU and are connected to their respective CPUs (e.g. PC_1 to PCMUX in CPU1, PC_2 to PCMUX in CPU2). This is the most convenient implementation because in the original SLC-3,

the PCMUX is a 4-to-1 MUX with only three input signals, so the new control signal for routing PC_X to the correct PC only requires minimum modifications to the control signals.

To record the status of which CPUs are selected to perform a certain task, a CPU_SEL register is embedded in the master CPU. This is a 4-bit register that corresponds to the status of the four compute nodes (e.g. “1101” means all but CPU3 is selected). To complete this structure, we added READY registers to all five CPUs. READY is a 1-bit register that shows if the CPU is ready to perform tasks when requested. READY_0 is for the master CPU, READY_1 through 4 are for the compute nodes. The status of READY registers is controlled manually via READY and WAIT instructions, which will be elaborated in the next section.

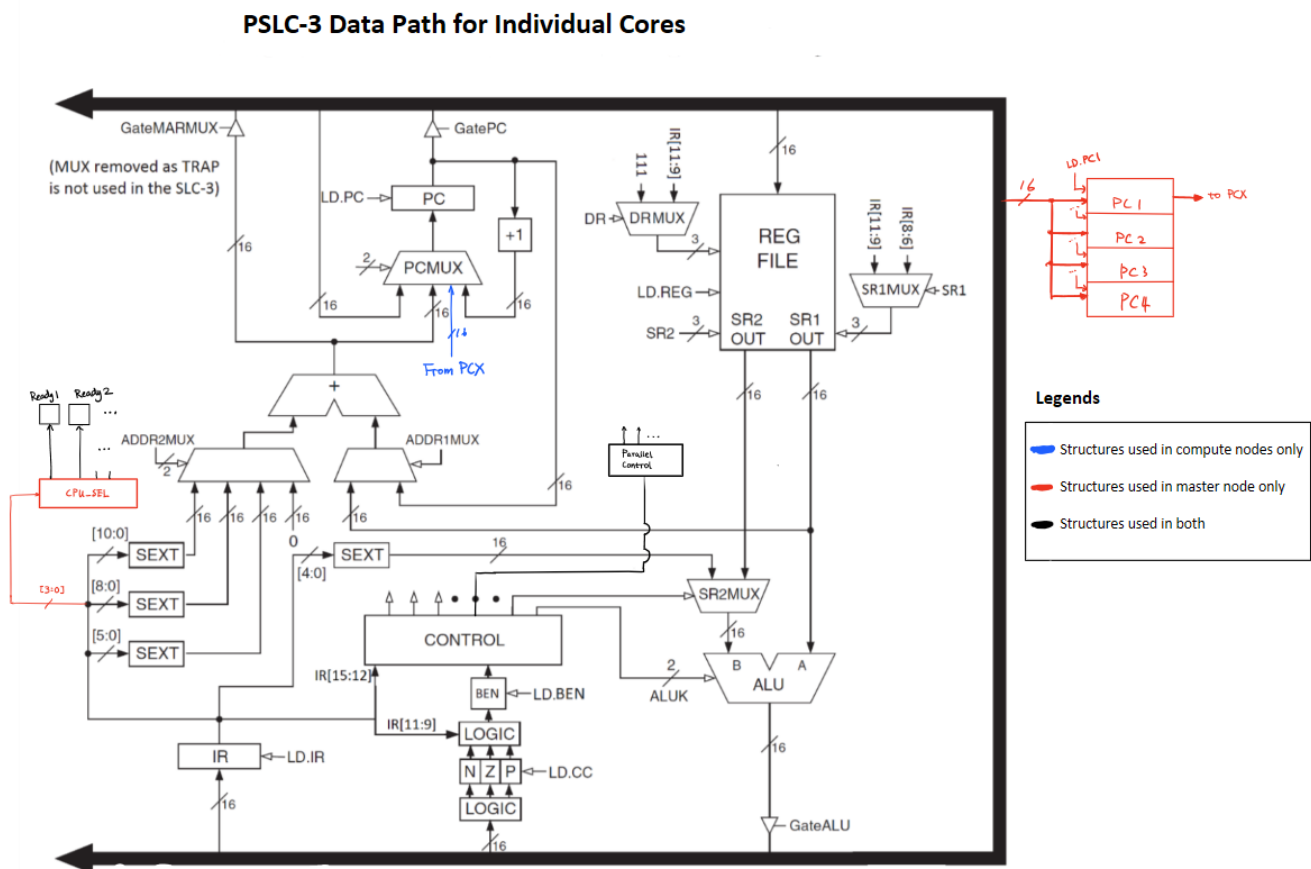


Figure 3 PSLC-3 General Datapath

ISA Description

The PSLC-3 includes all instructions but the PAUSE operation, which is replaced by parallel specific instructions (PSI). When PSI is parsed in state 18 (Decode State), it is then routed to state 36, which is the dedicated decode state for parallel instructions. These instructions are used to initiate the parallel tasks, some of which are used only in the master node, some are used only in the compute nodes, the rest are used in both master node and the compute nodes. It is the programmer's responsibility that the instructions are executed on the correct core. For example, do NOT execute PC_INIT while toggling a slave. This section cover all instructions added to support parallel operations, but does not include any original SLC-3 instructions, which are described in detail in the lab 6 report. The PSLC-3's instructions are described below.

PC_INIT (Master, state 37): 1101 | 000 | Register[3 bits] | 00 | PC_X [4 bits]

Each core contains 4 16-bit PC_X registers. The purpose of these registers is to send a starting address to the slave cores (which would grab the value coming from these registers by the INTR_PC instruction). This instruction should only be used by the master for the initialization of the slaves. PC_INIT loads the PC_X registers with the value from the specified register in the register file. The MSB corresponds to PC_1, the LSB corresponds to PC_4.

$$PC_X \leftarrow Register_X$$

CPU_SEL (Master, state 38): 1101 | 001 | 00000 | IMM4

Each core contains a single 4-bit CPU_SEL register. This register signals to the slaves to begin execution. This instruction should be used in conjunction with the WAIT signal in the slaves. Note that the slaves should never use the CPU_SEL signal because they are not responsible for the execution of any sub-cores. CPU_SEL loads the 4-bit CPU_SEL register with the 4-bit immediate value. The MSB corresponds to CPU_1, the LSB corresponds to CPU_4.

$$CPU_SEL \leftarrow IMM4$$

SYNC (Master, state 39) : 1101 | 011 | 00000 | CPU_X [4 bits]

The master core can pause execution of its program until the specified slaves send a READY signal. The CPU_X value is a 4 bit value to indicate which READY signals are desired from which slaves, where the most significant bit corresponds to CPU_1 (a.k.a slave 1) and the least significant bit corresponds to CPU_4, and likewise in between. The logic should be like below.

```
If ( CPU_X & (Ready4 <<3) & (Ready3<<2) & (Ready2<<1) & Ready1) != CPU_X {  
    Loop  
}  
Else go to S_18
```

READY (Master and slave, state 40) : 1101 | 011 | 000000000

Set the 1-bit CPU_READY flip flop to 1. This instruction can be used in conjunction with the SYNC instruction.

$CPU_READY \leftarrow 1$

WAIT (Slave, state 41): 1101 | 100 | 000000000

Pause execution of program until CPU_SEL is set high. Then set CPU_READY to 0. The purpose of this instruction is to pause the slave cores until the master gives them the CPU_SEL signal. This can also be used to pause the master node indefinitely once the entire program is finished.

$CPU_READY \leftarrow 0$

INTR_PC (Slave, state 42): 1101 | 101 | 000000000

Immediately interrupt (load) the PC with the value coming from the PC_INIT output.

$PC \leftarrow PC_X$ (coming from Master)

BR_CPUID (Master and slave, 43): 1101 | 110 | CPUID [3 bits] | PCoffset6

Each core has a 3-bit identifier hard-wired as input. BR_CPUID branches to the PC relative offset *PCoffset6* if the core's identifier matches the CPUID given by the instruction, else go to state 18.

ISDU Description

Based on the original **FETCH→DECODE→EXECUTE** instruction cycle, the PSLC-3 adopts a very similar approach. However, in order to keep things clean and easy to implement, we must add the necessary instructions without making significant changes to the state machine. If we had strictly followed the LC-3 ISA syntax, we would have added the instructions in the four upper-most bits, except all combinations of the upper-four bits are occupied in either LC-3 or SLC-3. It is highly unfavorable to expand the opcode to 5-bit from 4-bit because doing so involves rewriting the entire ISA and significant modifications to the hardware infrastructure. Therefore, we made a design choice to add a 3-bit secondary decode stage to accommodate the parallel instructions. This way we can add all the instructions we need while retaining the existing ISA.

Although there are opening slots for opcodes in the SLC-3 ISA, we wanted our design to be friendly to future development, so it is ideal that we also leave the opcodes for LC-3 alone. Recall that back in lab 6.1, we implemented a PAUSE state using opcode 1101 to view the current status of computation via the LEDs. Now that the SLC-3 is fully implemented, the PAUSE is rendered useless and ready to be used for the secondary decode stage (although we have implemented another PAUSE state in a different flavor, which will be explained later in this section). We name all the subsequent instructions originated from this secondary decode stage Parallel Specific Instructions (PSI) and the secondary decode stage `PSI_DECODE`

As a result, the instruction cycle for PSLC-3 is **FETCH→DECODE→PSI_DECODE→EXECUTE**. We enter the `PSI_DECODE` stage by calling opcode 1101, then parse `IR[11:9]` which are the three bits immediately lower than the regular opcodes. A diagram is shown below, the transition marked “36” is the `PSI_DECODE`. The expanded state machine for parallel instructions generally takes care of two things: initializing the PC for each core and preventing conflict between different cores. More specifically, state 37 sets up PC values for the compute nodes in the master node, then in state 41 compute nodes load their PC with the designated values from the master node by sending proper control signals.

To avoid race condition between cores, we implemented a policy to handle requests very much like a generic Linux spinlock. In state 38, CPUs are selected to perform task as the programmer desires, then the `READY_X` registers’ internal state is controlled by state 40 and state

41. The name for state 41(WAIT) might be somewhat misleading, since this instruction does not immediately pause execution of assembly program; state 39 (SYNC) actually pauses the program by comparing the contents in READY_X (4*1-bit) registers against CPUX(1*4-bit) register to determine if this state is looped. The programmer controls under what circumstances should the execution pause by passing proper CPUX value.

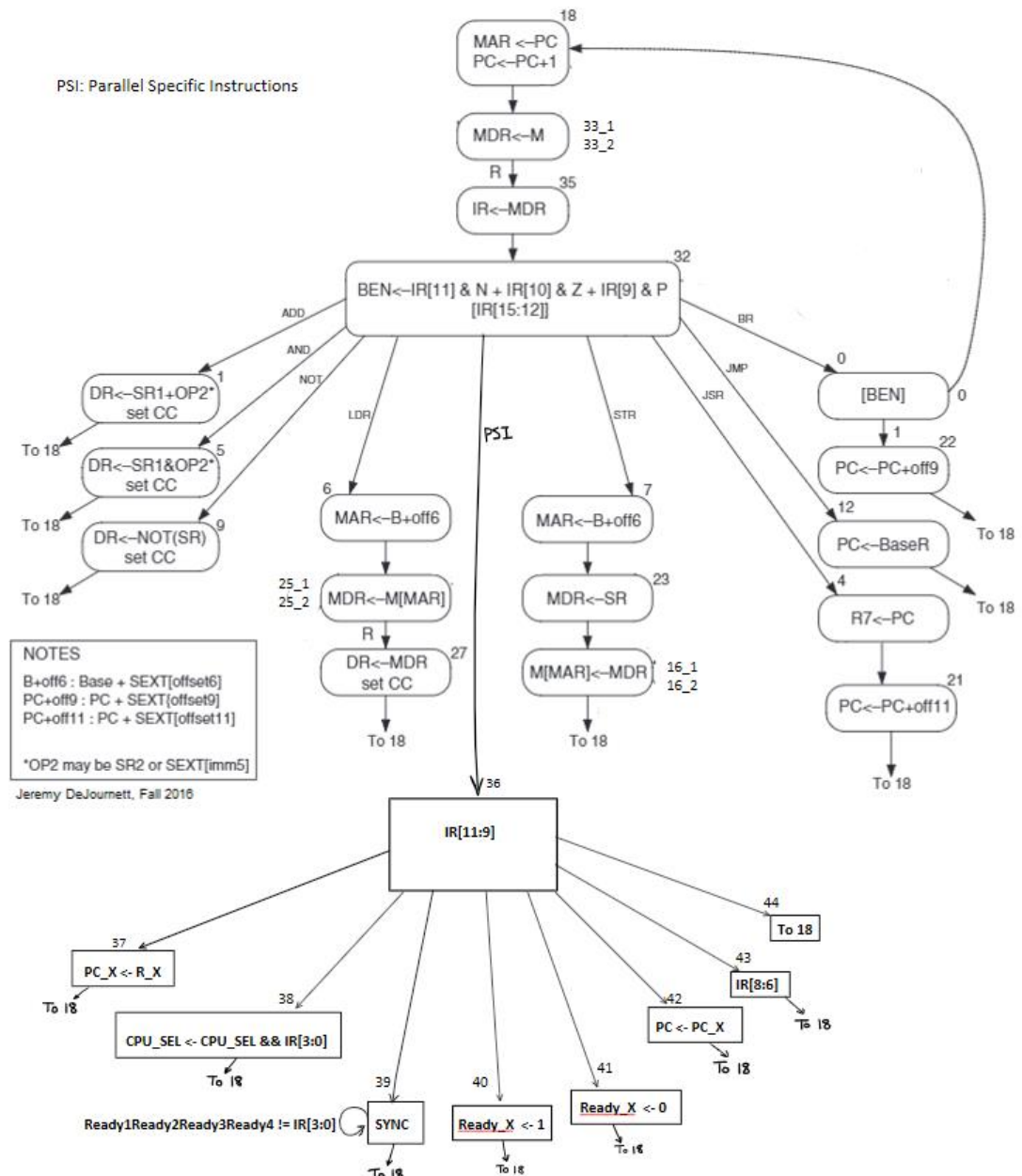
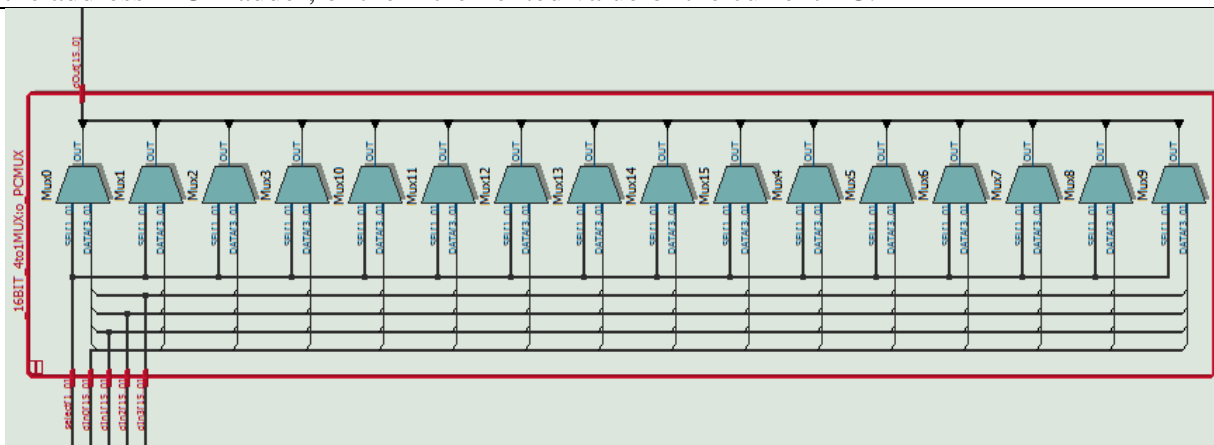


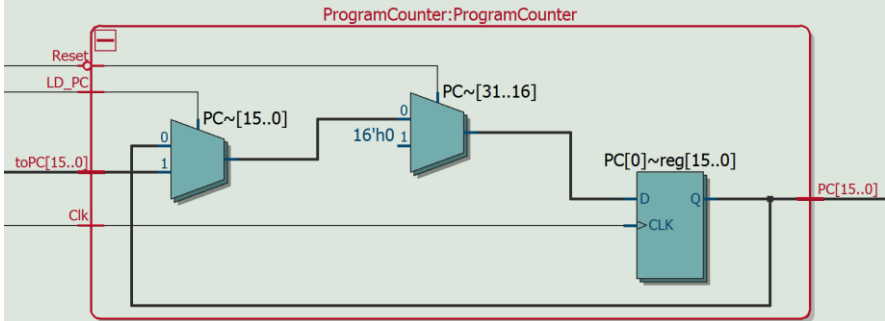
Figure 4 PSLC-3 State Diagram

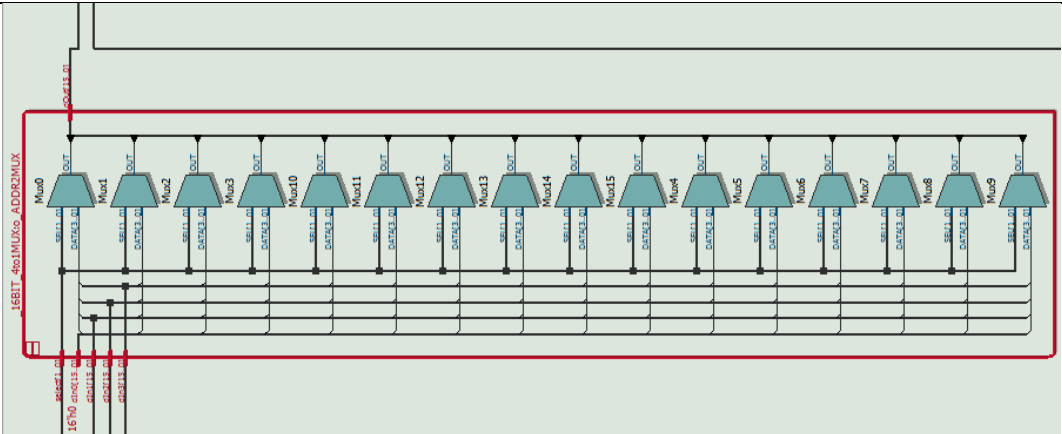
Module Description

Name: IOhandler.sv
Inputs: input logic clk,, input logic reset, input logic [4:0] writeRequest, input logic [4:0] readRequest, input logic [15:0] ADDR0, input logic [15:0] DATA0, input logic [15:0] ADDR1, input logic [15:0] DATA1, input logic [15:0] ADDR2, input logic [15:0] DATA2, input logic [15:0] ADDR3, input logic [15:0] DATA3, input logic [15:0] ADDR4, input logic [15:0] DATA4, input logic [15:0] fromTristate,
Outputs: output logic [15:0] toTristate, output logic [15:0] addressToSRAM, output logic [15:0] DataToCPUs, output logic SRAM_WE, output logic SRAM_RE, output logic [4:0] requestDone
Description: Implements a queueing mechanism that manages memory access operations between cores to avoid any collision. There is no particular settings for priority levels, memory access is first come first served. If multiple memory access requests arrive in the same clock cycle, priority is arbitrarily chosen. In addition, an SRAM controller is implemented in this module to assign data lines to the correct core via MUXes.

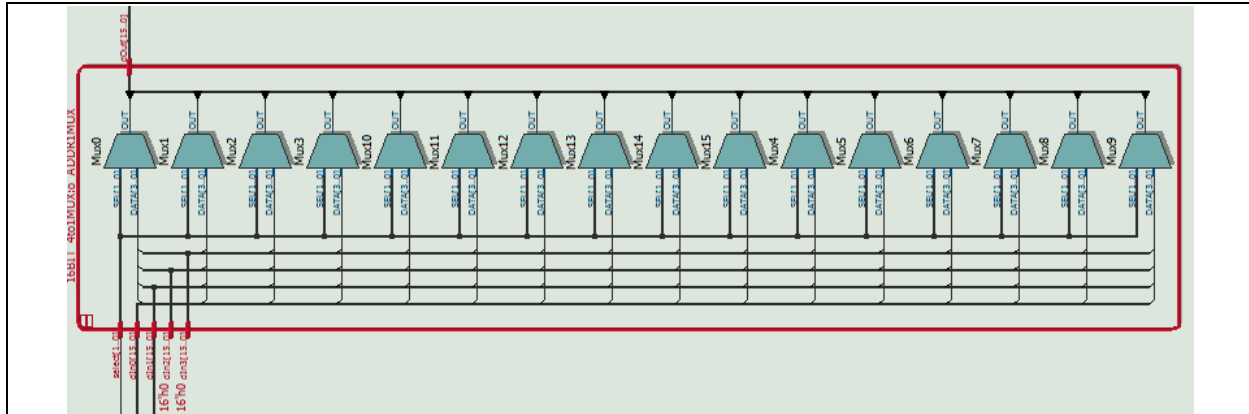
Module: PCMUX (16-bit 4-1 Multiplexer)
Inputs: busLine, PC+1, ADDRMUX Adder, PCselect
Outputs: fromPCMUX
Description: The PC MUX selects whether to send to the PC register the value from the bus, the address MUX adder, or the incremented value of the current PC.



Module: PC (16-bit register)
Inputs: fromPCMUX, Clk, Reset, LoadPC
Outputs: fromPC
Description: The program counter keeps track of the next instruction to be executed.

Module: MAR MUX (a.k.a address adder, 16-bit adder)
Inputs: fromADDR1MUX, fromADDR2MUX
Outputs: fromMARMUX
Description: The MAR MUX, or address adder, adds the ADDR1 MUX and ADDR2 MUX for memory access operations. The output of this module can be sent either to MAR or to the PC.
<i>Diagram not available. Simple 16-bit adder.</i>

Module: ADDR2MUX (16-bit 4-1 multiplexer)
Inputs: Ground, SEXT(IR[10:0]), SEXT(IR[8:0]), SEXT(IR[5:0]), select
Outputs: fromADDR2MUX
Description: This module selects IR[10:0], IR[8:0], IR[5:0], or 16-bit 0's depending on where the immediate value in IR is stored. The immediate values would be given in an ADDi, ANDi, or JSR instruction.


Module: ADDR2MUX (16-bit 4-1 multiplexer implemented as 2-1)
Inputs: fromPC, fromSR1, select
Outputs: fromADDR2MUX
Description: The ADDR2 multiplexer chooses between the PC or source register 1 (which comes from the register file).

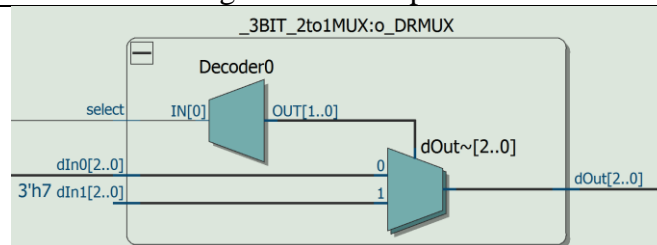


Module: DRMUX (3-bit 2-1 MUX)

Inputs: fromIR[11:9], 111, DRMUX select

Outputs: fromDRMUX

Description: The DR MUX selects between the register number given by fromIR[11:9] or register 7. Register 7 is used to store PC during the JSR instruction. The output is sent to the register file to select the destination register where input will be stored.

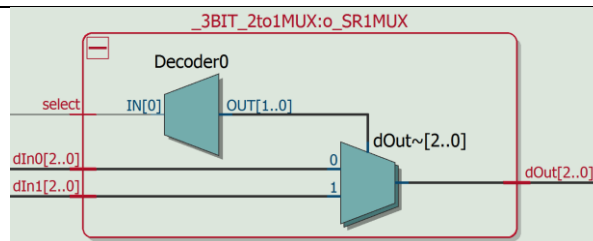


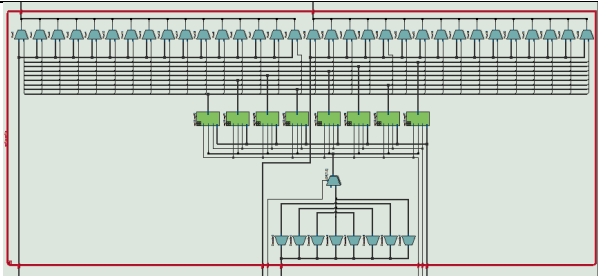
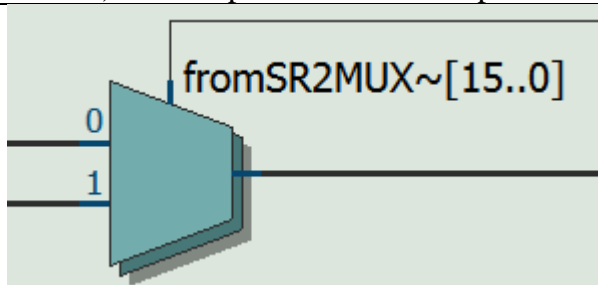
Module: SR1MUX (3-bit 2-1 MUX)

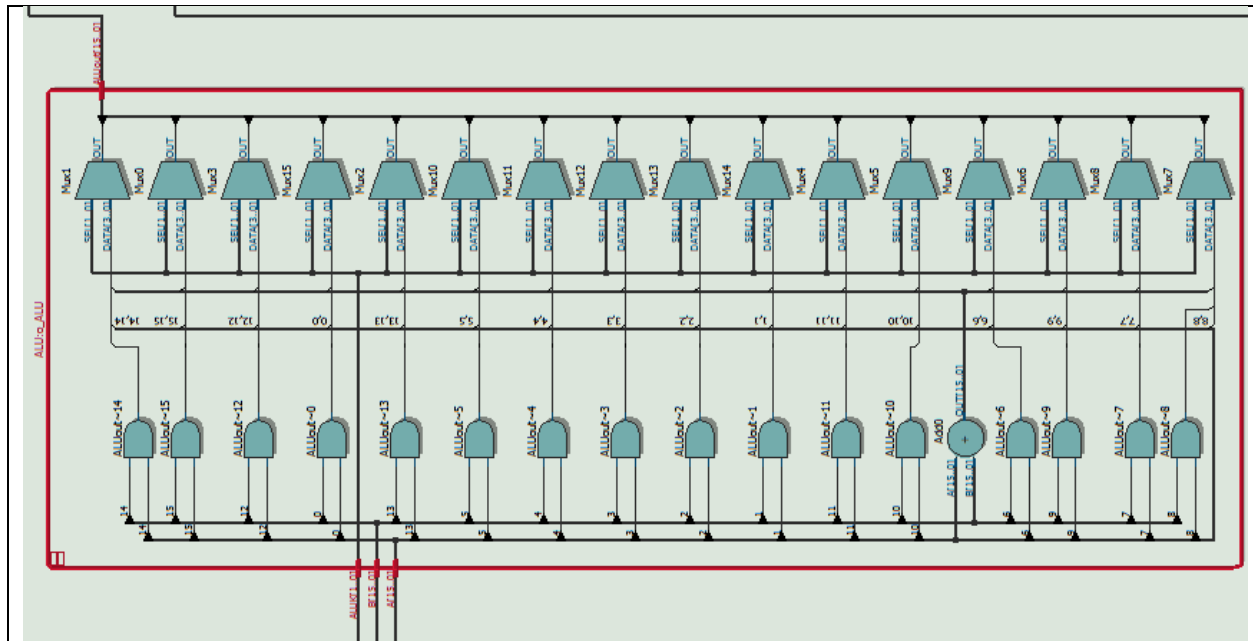
Inputs: fromIR[11:9], fromIR[8:6], SR1MUX select

Outputs: fromSR1MUX

Description: The SR1 MUX selects where to grab from IR the source register 1 value. The location of the SR1 value in the instruction changes depending on the instruction. The output of this module is fed into the register file to select which register to output to the register file's SR1 output.



Module: Register File	
Inputs: busLine, fromDRMUX (for DR select), fromSR1MUX (for SR1 select), SR2select (comes from IR[2:0]), LD.REG, Clk, Reset	
Outputs: fromSR2, fromSR1	
Description: The register file contains 8 16-bit registers numbered R0 through R7. It takes 16-bit input from the bus and will write to the register given by fromDRMUX if LD.REG is high. It will always output the register given by fromSR1MUX to fromSR2 and will output the register given by SR2select to fromSR2. This module also clears all data from all registers if reset is active.	
	
Module: SR2MUX (16-bit 2-1 MUX)	
Inputs: SEXT(IR[4:0]), fromSR2, SR2MUX select	
Outputs: fromSR2MUX	
Description: The SR2MUX chooses between an immediate value from IR or the value coming from SR2 (from the register file). The output is sent to one input of the arithmetic logic unit	
	
Module: Arithmetic Logic Unit (ALU)	
Inputs: fromSR2MUX, fromSR1, ALUK (chooses which operation to perform on the two numbers)	
Outputs: fromALU	
Description: The ALU performs the arithmetic or logic operations on the two numbers provided to it. Which operation to perform is provided by the ALUK signal. The operations that the values of ALUK denote is an arbitrary choice, but must obviously remain consistent between the ALU and the control logic in the ISDU.	

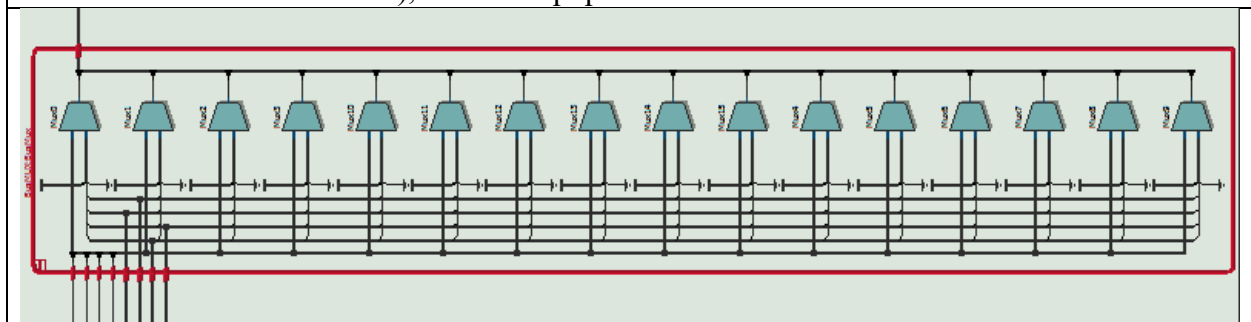


Module: Bus (16-bit multiplexer)

Inputs: GatePC, GateMDR, GateALU, GateMARMUX, fromMARMUX, fromPC, fromALU, fromMDR

Outputs: busLine

Description: Because the bus must be implemented on an FPGA instead of a physical wire, it needs to be implemented as a multiplexer. The 1-bit signals GatePC, GateMDR, GateALU, GateMARMUX act as the select lines to the MUX. Since there are 4 select lines, this is technically a 16-1 MUX, however the address lines are one-hot because only one module should be populating the bus at any given time. This means that we have a 4-1 MUX with one-hot select encoding. This module has been written such that if more than one gate is open at any time (an event that should never occur), the bus is populated with all 0's.



Module: Tristate Buffer
Inputs: Clk, ~WriteEnable, DataToSRAM, Data
Outputs: DataFromSRAM, Data (NOTE: Data is the bidirectional connection to SRAM)
Description: The SRAM uses a bidirectional in/out wire to handle SRAM input and output. This module provides an interface to the SRAM's bidirectional wire via two separate <i>DataToSRAM</i> (as input to this module, coming from the CPU) and <i>DataFromSRAM</i> (as output, coming from SRAM) lines. If the CPU is requesting a write operation to SRAM, the tristate buffer will populate the data line with <i>DataToSRAM</i> . Otherwise, the buffer will “drive” the data line with high impedance to allow reading data coming from SRAM.

Module: NZP logic
Inputs: busLine
Outputs: NZPlogic (combinational/asynchronous)
Description: The NZP logic is a combinational circuit that continuously outputs the sign (negative, zero, positive) of whatever value is on the bus. The 3 bits coming from this module are by definition mutually exclusive, i.e. one-hot; a number cannot simultaneously satisfy more than one of these signs. What follows is the logic that was used to generate these signals. <pre> isNeg = busLine[15] ? 1'b1 : 1'b0; isPos = ~isNeg & ~isZero; case (busLine) 16'b0000000000000000: isZero = 1; default : isZero = 0; endcase </pre>

Module: NZP (3 flip-flops)
Inputs: NZPlogic, LD.CC
Outputs: fromN, fromZ, fromP
Description: The NZP flip flops are loaded with the 3-bit signal coming from NZPlogic when the condition code load signal is set active.

Module: Branch logic (combinational)
Inputs: NZPff, IR[11:9]
Outputs: shouldBranch
Description: The branch logic takes as input the NZP flip flops as well as the 3-bit code coming from IR that denotes the branching condition. The output is a 1-bit signal that denotes whether the NZP coming from IR matches at least one NZP coming from the flip flops. The logic for shouldBranch is: $\text{shouldBranch} = (\text{fromIR}[11] \ \& \ \text{fromN}) \mid (\text{fromIR}[10] \ \& \ \text{fromZ}) \mid (\text{fromIR}[9] \ \& \ \text{fromP});$

Module: Branch enable (1-bit flip flop)
Inputs: shouldBranch, LD.BEN
Outputs: BEN
Description: The branch enable flip flop is loaded with the value shouldBranch when LD.BEN is set active.

Module: ISDU
Inputs: Clk, Reset, Run, Continue, Opcode[3:0], IR_5, IR_11, BEN, fromIR[15:0], CPUID [2:0], Ready1, Ready2, Ready3, Ready4, CPUX[3:0], fromCPU_SEL, memReady
Outputs: LD_MAR, LD_MDR, LD_IR, LD_BEN, LD_CC, LD_REG, LD_PC, LD_LED, LD_PC1, LD_PC2, LD_PC3, LD_PC4, LC_CPU_SEL, LD_SET_READY, SET_READY_DATA, GatePC,, GateMDR, GateALU, GateMARMUX, PCMUX [1:0], DRMUX, SR1MUX, SR2MUX, ADDR1MUX, ADDR2MUX [1:0], ALUK [1:0], Mem_CE, Mem_UB, Mem_LB, Mem_OE, Mem_WE
Description: Defines all the states and specifies the next state logic based on the state diagram. Splits states that require SRAM access into two states due to timing constraints; moves instructions that require memory I/O access (i.e., Mem_OE, Mem_WE) one state forward to compensate the time it needs to pass through the memory control signal synchronizers. Synchronizers are needed for SRAM because the SRAM is asynchronous. Glitches in the control signals could be fatal to data stored on the memory. Since the SLC3 ISA is significantly smaller than that of LC3's, only a handful of operations are required: ADD(i), AND(i), ADD, AND, NOT, LDR, STR, JSR, JMP, BR, PSI (includes all parallel operations instructions).

Module: MDR Mux
Inputs: busLine, fromMem2IO
Outputs: fromMDRMUX
Description: The MDR MUX is capable of sending either the bus or the data coming from Mem2IO to MDR. It is a simple 16-bit 2 to 1 multiplexer.

Module: MAR (memory address register)
Inputs: Clk, Reset, LD.MAR, busLine
Outputs: fromMAR
Description: The MAR is a 16-bit register that stores the memory address used to operate RAM.

Module: MDR (memory data register)
Inputs: Clk, Reset, LD.MDR, busLine
Outputs: fromMDR
Description: The MDR is a 16-bit register that stores the data coming from RAM or the data to be written to RAM.

Module: IR (instruction register)
Inputs: Clk, Reset, LD.IR, busLine
Outputs: fromIR
Description: The IR contains the current instruction to be executed by the CPU.

Sample Assembly Program

The following assembly program is what was used to test the PSLC-3.

```

ANDi R0,R0,0
BRnzp 3
0x0009          ; master address
0x0014          ; kernel address
0x0010          ; pointer to the array to increment
BR_CPUID 0,3    ; skip master CPU to line 9
LD R0,-4        ; if a slave, load beginning of kernel
LD R6,-4        ; load array pointer to R6
JMP R0          ; branch to beginning of kernel
SYNC 1111       ;          MASTER: beginning of master program
CPU_SEL 1111    ; instruct the slaves to begin
BRnzp 0
CPU_SEL 0000    ; Remove the "run" signal to prevent immediate continuation
SYNC 1111       ; wait until the slaves are done
READY          ; send ready signal to top-level
WAIT           ; program is done
0x0000         ; array[0]
0x0000         ; array[1]
0x0000         ; array[2]
0x0000         ; array[3]
READY          ;          KERNEL: set ready signal to 1
WAIT           ; wait until master gives the go-ahead
ANDi R1,R1,0    ; clear R1 for all slaves
ADDi R1,R1,15   ; store 15 in all slaves (number of times to increment)
BR_CPUID 1,3
BR_CPUID 2,4
BR_CPUID 3,5
BR_CPUID 4,6
LDR R0,R6,0     ; Load the value to increment into R0 (CPU1)
BRnzp 5
LDR R0,R6,1     ; Load the value to increment into R0 (CPU2)
BRnzp 3
LDR R0,R6,2     ; Load the value to increment into R0 (CPU3)

```

```

BRnzp 1
LDR R0,R6,3      ; Load the value to increment into R0 (CPU4)
ADDi R0,R0,1     ; Increment the value
BR_CPUID 1,3
BR_CPUID 2,4
BR_CPUID 3,5
BR_CPUID 4,6
STR R0,R6,0      ; store value back to array (CPU1)
BRnzp 5
STR R0,R6,1      ; store value back to array (CPU2)
BRnzp 3
STR R0,R6,2      ; store value back to array (CPU3)
BRnzp 1
STR R0,R6,3      ; store value back to array (CPU4)
NOT R0,R0        ; Do 2's compliment
ADDi R0,R0,1     ; 2's compliment
ADD R0,R0,R1     ; see if we are done incrementing
BRp -27          ; If we're not done, continue on
READY
WAIT

```

The PSLC-3 ISA is accompanied by an assembler program that parses the assembly text and converts it into a Big-endian binary file. This binary file can then be loaded directly onto the SRAM on the DE-2 development board. This assembler is very necessary when writing assembly code because the alternative would be explicitly writing out each bit of every instruction, a tedious and error-prone task. In addition to this, the assembler that was written creates an ASCII text file that you can copy-paste into the test_mem module to initialize the simulated memory with the corresponding machine instructions. This text file is simple SV code that initializes the memory array with the correct binary information.

Bug Log

One of the most difficult parts of this project was the IO handler. The main difficulty was making sure that all of the signals to SRAM were coordinated properly with the requests coming from the cores. The main bug that kept coming up was improper writes to SRAM. Often when a core would request a write operation, the data for a previously adjacent read operation would still be populating the bus line due to being registered in the tri-state output, thus writing the wrong data to the SRAM. This was remedied by spacing out all of the read and write operations by two clock cycles so that there was no garbage data in the registers at the time of a write.

Surprisingly, modifying the ISDU of the cores was not that difficult to achieve. Since the majority of the LC-3 architecture was already present, adding the extra few states to implement our parallel instructions was not difficult.

Another difficulty in this lab deals with making the IO handler sufficiently fast with carrying out memory requests. Our first iteration of the IO handler had no state machine, but it was able to process requests very quickly. This proved to be a problem however because it would sometimes process them too quickly. Because the IO handler sends IO to the tristate buffer, which then sends data to the SRAM, performing operations on the memory requires two clock cycles, the first clock cycle to populate the data register in the IO handler itself, and another clock cycle to populate the registers in the tristate buffer. Initially, the IO handler only allowed one clock cycle for memory access. This was remedied by adding a state machine in order to control exactly how many clock cycles each internal operation took.

Simulation Waveforms

The simulated waveform in modelsim is presented below. This waveform shows the signals for the array in memory that will contain incremented values. A sample assembly program was loaded onto SRAM that increments a 4 element array, each slave node in the PSLC-3.0 being responsible for one of those elements. This array is simulated by the “mem_array” variables. At

the beginning of execution (top waveform), this array shows all 0's. When the program is done executing, the array now shows all 0xf indicating that each core incremented its respective element 15 times.

Both waveforms also show the READY signals of each core. The top waveform shows that the ready signals for each core are set 0. Once each of the slaves are done executing, they set their READY bits high (as per the code written in assembly). These ready signals are sent directly to the master node, where it then sets its READY signal high once all the slaves are ready. These variables can be seen by *Ready1*, *Ready2*, *Ready3*, *Ready4* (which correspond to the slaves) and *CPU_READY* (which corresponds to the master).

In addition to these, the queueing system in the IOhandler module can be seen. The variables *first*, *second*, *third*, *fourth* and *fifth* show where in the queue each IO request resides.



Design Statistics

LUT	1843+1132=2975
DSP	0
BRAM	0
Flip-flop	1380
Frequency	82.67 MHz
Static Power	98.77 mW
Dynamic Power	0 mW
Total Power	145.41 mW

Conclusion

The LC-3 is an excellent educational computer architecture because it is complete, yet it exhibits some degrees of flexibility for modifications. The PSLC-3 takes full advantage of that by deriving a new set of parallel instructions that coordinate multiple CPU cores when computing large tasks. Even though this proof of concept is far from the actual implementation of SMP support in an actual modern CPU, it provides a good learning platform for rudimentary parallel processing concepts.