University of Illinois Urbana-Champaign

ECE 385 Final Project

PSLC-3: A Proof of Concept for Parallel Multiprocessing

Landon Clipp, Jack Hu

April 2017

www.GitHub.com/LandonTClipp/PSLC-3.0

Abstract

Modern computers often rely heavily on the concept of multiprocessing: partitioning some computational task and delegating multiple compute units to process each of these tasks individually. Parallel computation has the benefit of dramatically reducing the overall computation time by employing multiple compute units to the same or similar tasks. This concept is most readily apparent in the ubiquity of Graphical Processing Units (GPU) which are used in areas like video processing, 3D graphics, physical simulations etc. In this paper, an expansion of the Simple Little Computer 3's (SLC-3) instruction set architecture (ISA) is designed, along with an expansion of the physical architecture, to support the execution of parallel assembly programs. This expansion serves as a Proof of Concept for parallel processing in a shared memory space.

Contents

1 Introduction **2**
- 1.1 The SLC-3 . . . 3
- 1.2 Development Hardware and Software . . . 3
  - 1.2.1 SystemVerilog and Quartus . . . 3
  - 1.2.2 FPGA and Cyclone Development Board . . . 4

2 Design and Operation of the PSLC-3 **5**
- 2.1 Hardware . . . 5
  - 2.1.1 IO Handler . . . 7
  - 2.1.2 Data Path . . . 8
  - 2.1.3 Instruction Set Architecture . . . 9
  - 2.1.4 Instruction Set Decoder Unit . . . 12

3 Example Parallel Assembly Program **14**

4 Results **16**
- 4.1 Bug Log . . . 16
- 4.2 Simulation Waveforms . . . 16
- 4.3 Design Statistics . . . 17
- 4.4 Conclusion . . . 17

Chapter 1

Introduction

During the early days of computing, almost all programs were written in a serial-execution manner. One compute unit would be used to execute commands one after another. This form of execution proved to be extremely limiting in the case when a computation needed to be performed on some large amount of data. As a simple example, consider the case where each element in an array of length 2 million needs to be incremented by some constant number of times. If a single compute unit was performing this operation, it would have to iterate over all 2 million elements individually. If multiple compute units were employed in this operation, say 4 of them, we could subdivide the array into 4 smaller sub-arrays and then have each of the 4 compute units perform the operation on their respective sub-array. It is readily apparent that this has the potential for a theoretical performance increase of 4 times. This can be extrapolated even further: instead of 4 compute units, imagine that we have 2 million compute units. In that case, each compute unit would be assigned to one single element of that array, thus performing the operation on all 2 million elements at once.

The idea of being able to dramatically increase the performance of a program by simply delegating more and more compute units to the task seems almost too easy. In fact, this suspicion is correct. It is almost never the case that an increase in the parallelism of a program results in a 1-to-1 performance increase. This arises due to some particular difficulties when multiple units are attempting to access the same shared memory space. Regardless of how many compute units one has at their disposal, memory access remains a bottleneck to parallel computation as each unit needs to compete with other units for access to the memory. This fact sets an upper limit to the performance increase that can be gained from adding more units. The common solution to this problem is to have a large number of shared memory spaces so that each compute unit will have to compete with, for instance, at most 7 other units. In this case, the number of shared memory spaces you would need would simply be:

$$ceil(n/8)$$

where n is the number of compute units. However, the implementation in this project will only use one shared memory space.

1.1 The SLC-3

The Little Computer 3 (LC-3) architecture was designed by Yale N. Patt and Sanjay J. Patel.[1] Its main purpose was to serve as an educational tool to teach students the fundamentals of computer architecture and also how assembly programs are written. Although the LC-3 ISA is much simpler than more mature architectures like x86, it still provides a lot of the same constructs that the more advanced architectures do. There are many different variants of LC-3, one of them being the Simple Little Computer 3. The SLC-3 simply removes a large number of the supported instructions in LC-3, but retaining enough of them to perform some very simple tasks. This further simplification of an already simplified architecture is done so that students in the ECE 385 course can implement the entire circuit in the SystemVerilog Hardware Description Language (HDL) within a reasonable time frame.

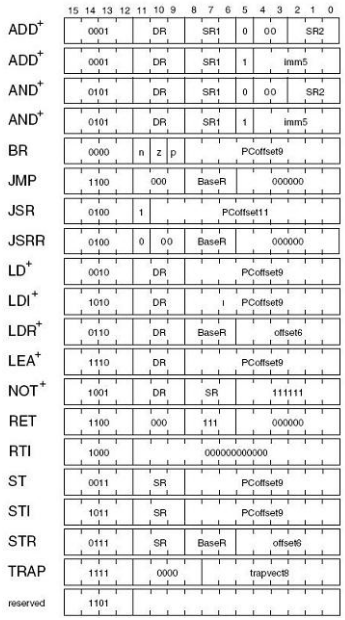

Figure 1.1: The LC-3 ISA

By the time the final project in this report was conceived, the SLC-3 had already been fully implemented so it was decided to use this circuit as a starting point. The SLC-3 has the same ISA as the LC-3 with the exception that the only instructions present are: ADD, AND, NOT, LDR, STR, JSR, JMP, BR.[1] It was possible (and allowed by course instructors) to use a full LC-3 implementation, however it was desirable to avoid using unfamiliar code and to rather use our own work that had already been thoroughly tested and confirmed to be operational.

1.2 Development Hardware and Software

Implementing a complicated circuit like an LC-3 CPU would be incredibly difficult if done with TTL chips. The process of building the circuit can be greatly expedited if done with some industry-standard programming languages and development tools. The following subsections outline the hardware and software tools used to design our circuit.

1.2.1 SystemVerilog and Quartus

SystemVerilog (SV) is a hardware description language that has seen widespread use in the circuit design industry due to its ability to facilitate a fast circuit debugging workflow (the hardware verification aspect of the language) along with the inherent benefit of abstracting circuit design into a high level language. The software suite used to write and debug the SV code in this project is Quartus II. Quartus II is an SV development suite created by Altera that includes both an SV electronic design automation (EDA) tool and an SV debugging

[1] As one exception, the LD instruction is implemented in the final PSLC-3 architecture to facilitate some useful operations in the parallel assembly program.

tool ModelSim (among many other things, but these two are what is mainly used in this project). Quartus can be downloaded for free from Altera's website.

1.2.2 FPGA and Cyclone Development Board

The Field Programmable Gate Array (FPGA) is a class of integrated circuits capable of being configured to almost any imaginable logic circuit (of course within certain design constraints). The real power of the FPGA is its ability to be quickly configured to a new circuit, a characteristic that proves to be an enormous aid not only in debugging, but in the design process itself. Before FPGAs and their associated description languages, circuit diagrams had to be crafted either manually or with the aid of some Computer Aided Design (CAD) tool. At the end of this lengthy process, the schematics were then sent to a fabrication facility that printed the integrated circuits and then shipped the order back to the circuit designers. This process proved to be lengthy, error prone and incredibly expensive. The advent of HDLs and FPGAs meant that the fabrication process could be entirely circumvented, as all that is needed is an HDL compiler and a relatively inexpensive FPGA chip or development board.

The FPGA used in this project is the Altera Cyclone IV aboard the Terasic DE2-115 development board. The development board provides a very useful interface to the FPGA chip itself which includes (but not limited to) a USB programming interface, toggle switches, push-button switches, static random access memory (SRAM), synchronous dynamic random access memory (SDRAM), flash memory, VGA controllers and many others. The FPGA on this board is connected via USB to the computer running the Quartus software.

Chapter 2

Design and Operation of the PSLC-3

2.1 Hardware

Because of the various number of considerations that have to be taken into account when designing a parallel system, a multi-core processing unit differs quite significantly than single-core units both in their internal hardware and the ancillary hardware necessary to facilitate multiple units within a shared memory space. The majority of this increased complexity arises from the fact that memory access has to be coordinated. If the memory operations were not coordinated, the compute units would step over each other when they request memory access within a few clock cycles of each other (memory access requires more than one clock cycle). This naive implementation is even potentially dangerous: if all units share the same RAM data bus, there could be more than one units driving the bus at any given time which would invariably cause damage to the hardware.

A block diagram for one kind of implementation of a single-core processing unit is shown below in figure 2.1.

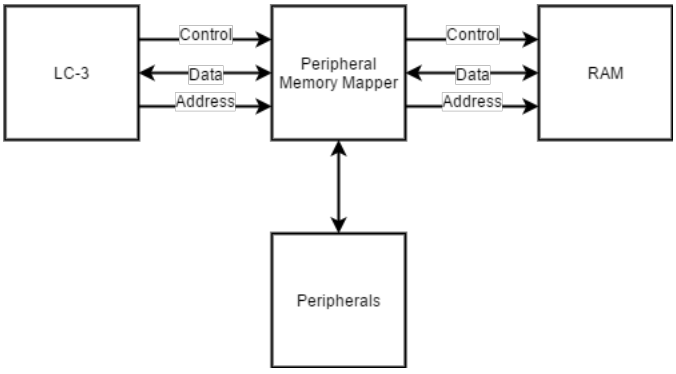

Figure 2.1: A single-core processor implementation.

In this implementation, the LC-3 CPU sends control and address signals to the RAM

through the peripheral memory mapper. In such a design, a few RAM addresses are dedicated to peripheral input and output such that the CPU can read values coming from or write values going to the peripherals simply by reading or writing to one of these predefined addresses. By interfacing the CPU with the memory mapper, access to peripherals is indistinguishable from a normal memory access operation. However, what must be noted is that the data going to or coming from the peripherals never actually resides in the RAM itself. Although appears that way to the CPU, the memory mapper actually reroutes the bidirectional data bus directly to the peripherals whenever it detects the CPU requesting one of the memory-mapped addresses. This allows the CPU to connect its data line directly to the peripherals when transferring data between them. This implementation is one of the simplest ways to interface with the outside world.

In the PSLC-3.0 design, there is a artificially similar interface to RAM (in the sense that there is some middle-man between RAM and the CPUs) but with some very major differences. PSLC-3.0 does not have memory-mapped IO, but rather a memory arbiter that handles memory access operations between multiple cores (as previously discussed). The reason for not including peripheral support is simply due to time constraints: the memory arbiter—also called the IO handler—proves to be sufficiently difficult to implement so it was decided to leave peripheral support for future versions of the PSLC-3. A high level block diagram for the parallel SLC-3 is shown below in figure 2.2.

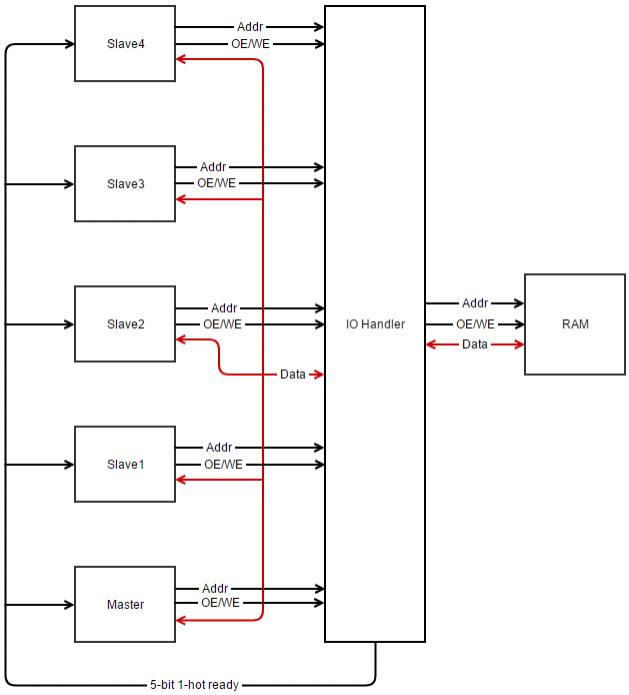

Figure 2.2: PSLC-3.0 high level block diagram.[1]

[1]This diagram should not be misinterpreted as each core sharing a data line to the IO handler. In this implementation, each core has its own separate 16-bit data line going to the IO handler.

2.1.1 IO Handler

This figure demonstrates the basic principles of how RAM is interfaced. Each of the 5 cores sends four separate signals to the IO handler: a 16 bit data line, a 16 bit address line, a 1-bit read (output enable, OE) and write (write enable, WE) line. The IO handler sends a 1-bit signal to each of the cores indicating when a memory access operation has finished, called the 1-hot ready signal. When a core wants to write data to RAM, it must populate its data line with the information to be written, populate its address line with the address to write the data, and set its WE signal high. The core will remain in this state until the IO handler sends back a ready signal to that core indicating when the data has been successfully written to RAM. The process for requesting a read operation is similar; the core sets its address line to the address it desires from memory, it sets its OE signal high, and it then waits to read data into its memory data register (MDR) until the ready signal from the IO handler is set high, an indication that the data line is populated with the requested data.

Internally, the IO handler is simply a queuing system that keeps track of memory requests, prioritizes the requests temporally (first-come-first-serve), and sends the proper signals to RAM to serve the requests to each core. This operation is controlled by a state machine. In the source directory of the accompanying GitHub page, the implementation of the IO handler can be seen in the IOhandler2.sv file. The next state logic contains a series of simple logic checks for each state that determines what the next state will be. The queuing and serving process in this state machine consists of 4 main operations:

1. **UPDATE QUEUE** — The first operation is to update the queue. The queue is a series of 5 elements each implemented as simple 3-bit registers. The value in each element in the queue is a numerical identifier corresponding to a specific core. The values chosen to represent the different cores is not terribly important, but in this implementation the binary values 1 through 4 indicates the slaves 1 through 4, and 0 indicates the master slave. These values move along the queue as it is periodically incremented (as will be seen in a moment). To update the queue, the IO handler looks at all of the current requests coming in through the OE/WE lines and places these requests at the earliest spot possible in the queue. If multiple requests happen at the same time, priority is chosen arbitrarily. These new requests are placed in a staging area, or what could be called a "queue-next" variable that will have its value placed in the queue once the INCREMENT QUEUE operation below is executed.

   One subtle problem that arose during the development of this module was that there was no way to determine if a request has already been placed in the queue without somehow keeping track of the previous state of the requests (i.e. the requests at the time the queue was updated). Without keeping track of this information, the IO handler would erroneously update the queue with requests regardless if those requests are already in the queue. This was fixed by simply adding the "readRequestPrev" and "writeRequestPrev" to the digital logic, updating these registers with the current request state each time the queue is updated. This way, if any new requests come in during any of the other states, the update state will be able to differentiate already-served (old) requests from the new requests that have yet to be queued.

2. **INCREMENT QUEUE** — This operation is rather simple. After the queue has been updated, all of the elements in the queue are assigned to the value of their respective queue-next variables. The queue-next variables, if not overwritten in the

UPDATE QUEUE stage to contain a new request, would simply contain the value of the previous element in the queue.

3. **READ/WRITE** — In the READ/WRITE state (READ and WRITE being two separate branches of state transition), the IO handler retrieves or sends data from RAM. The core that is first in the queue will have its address line sent to RAM. If the request is a READ operation, the IO handler sets the RAM OE line active and stores the result in a data register. If the request is a WRITE operation, it populates the RAM data bus with the data coming from the core, then setting the WE line active.

4. **DONE** — After the memory operation has been performed, the DONE states populate the corresponding

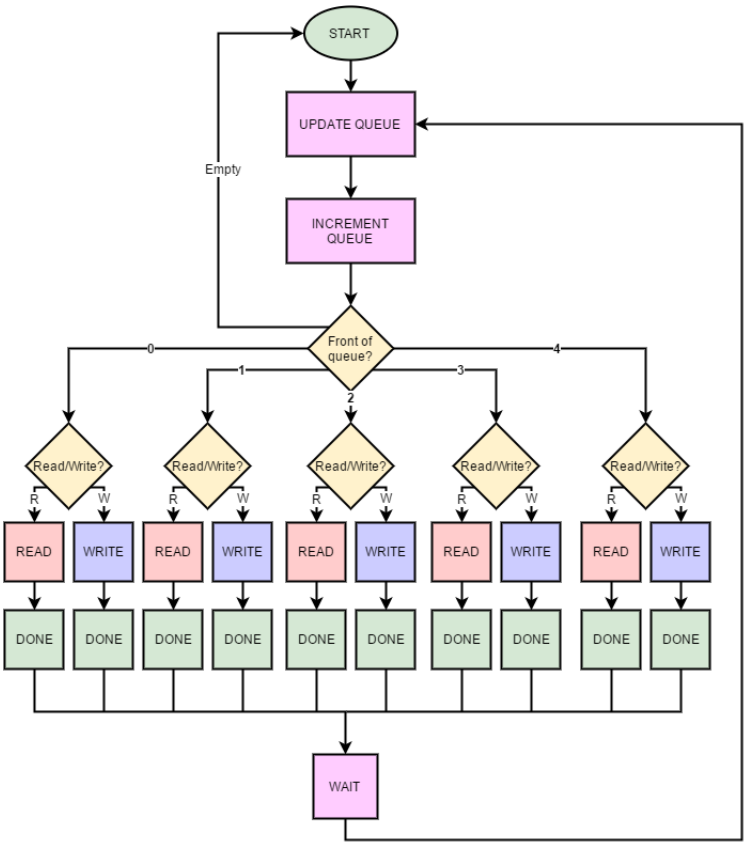

Figure 2.3: High level IO-Handler state diagram.

2.1.2 Data Path

Figure 2.4 shows the data path architecture of the SLC-3 as well as the interface between the computational hardware and the external Static Random Access Memory (SRAM). This architecture is almost identical to what the LC-3 would look like, the only differences include

the low-level details of the control unit as well as the MEM2IO and Tristate Buffer modules. The SLC-3 as implemented in the ECE 385 course (which is technically called the SLC-3.2) provides support for memory mapped peripherals, handled by the MEM2IO module. This will not be explained in detail as the SLC-3 is not the focus of this report.

The PSLC-3 architecture builds upon this design with only a few minor modifications. In order to support parallel operations within the assembly code, a few simple components needed to be added. Figure 2.5 shows the modifications done to the SLC-3 hardware. The first thing to note is that the MEM2IO module is replaced with the IO handler (discussed in previous sections). It should be noted that in this iteration, memory mapped IO is not implemented, but future versions of the PSLC-3 could build this support inside of the IO handler module.

CPU_SEL — As alluded to by figure 2.2, all parallel assembly programs are strongly suggested to be written such that the Master core mediates and controls the execution of the 4 slave nodes, much like an operating system controls the processes within it. Each core in this computer has no differences from each other either by hardware or by any queuing priority IO handler might give, but they do differ in the interconnections of their various signals. It is because of the differences in signal interconnect that only the Master core should be used to act as the executive authority in parallel execution. Each core implements within the control unit a state which will pause the execution of the core until it receives an external CPU_SEL signal as input. This CPU_SEL signal should come from the Master core. For instance, one possible scenario that the CPU_SEL signal would be useful would be to have all Slave cores independently pause execution (sending kind of signal to Master that they are in a paused state) until the Master core sends them a READY signal to begin execution. This is synchronization behavior, similar (to those who are familiar with it) to CUDA's *cudaDeviceSynchronize()* function. CPU_SEL is the 4-bit register that sends these signals to the slaves. It takes as its input the first 4 least significant bits of the Instruction Register (IR) and a CPU_SEL.LD signal from the control unit.

PC_X — PC_X is a 16-bit signal sent by the master core to interrupt the program counter of the slave nodes. This signal comes from the PC1, PC2, PC3, and PC4 16-bit registers in the master core and arrives at one of the inputs of the PCMUX. This hardware provides the capability of the master slave to provide an address for the slaves to jump to. This is useful in the case where the slave cores need to be initialized to some starting address.

READY — Every core has a single 1-bit register that will be controlled by a specific assembly instruction. This READY signal indicates to whatever hardware responsible for the core (the master slave, in the case of the slave nodes. The master slave can also send its ready signal to some outside hardware) that the core is done executing.

2.1.3 Instruction Set Architecture

Figure 1.1 show the instruction set architecture of the LC-3. One simplification of the LC-3 is the SLC-3 which only implements the ADD(i), AND(i), NOT, BR, JMP, JSR, LDR, and STR. The SLC-3 was chosen as a starting point for the PSLC-3 because it had already been implemented in SystemVerilog during an earlier ECE 385 class assignment. Although the ISA for the SLC-3 is incredibly thin, it provides enough support to do simple operations.

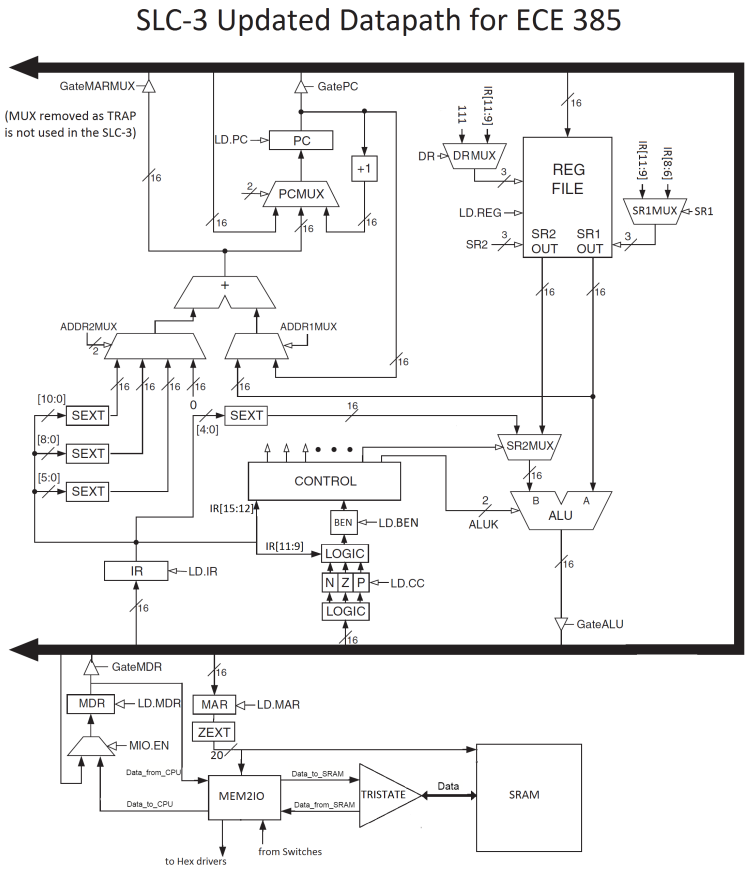

Figure 2.4: SLC-3 Data Path

The PSLC-3 includes all instructions of the SLC-3 but the PAUSE operation, whose opcode is replaced by parallel specific instructions (PSI). When a 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 used only in the compute nodes, and the rest used in both master node and the compute nodes. It is the programmers responsibility that the instructions are executed on the correct core. For example, do not execute PC_INIT on the master core because this hardware environment provides no way to initialize master to anything but a hard-wired value. This section covers 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 ECE 385 report. The PSLC-3s instructions are described below.

1. **PC_INIT** — Each core contains four 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

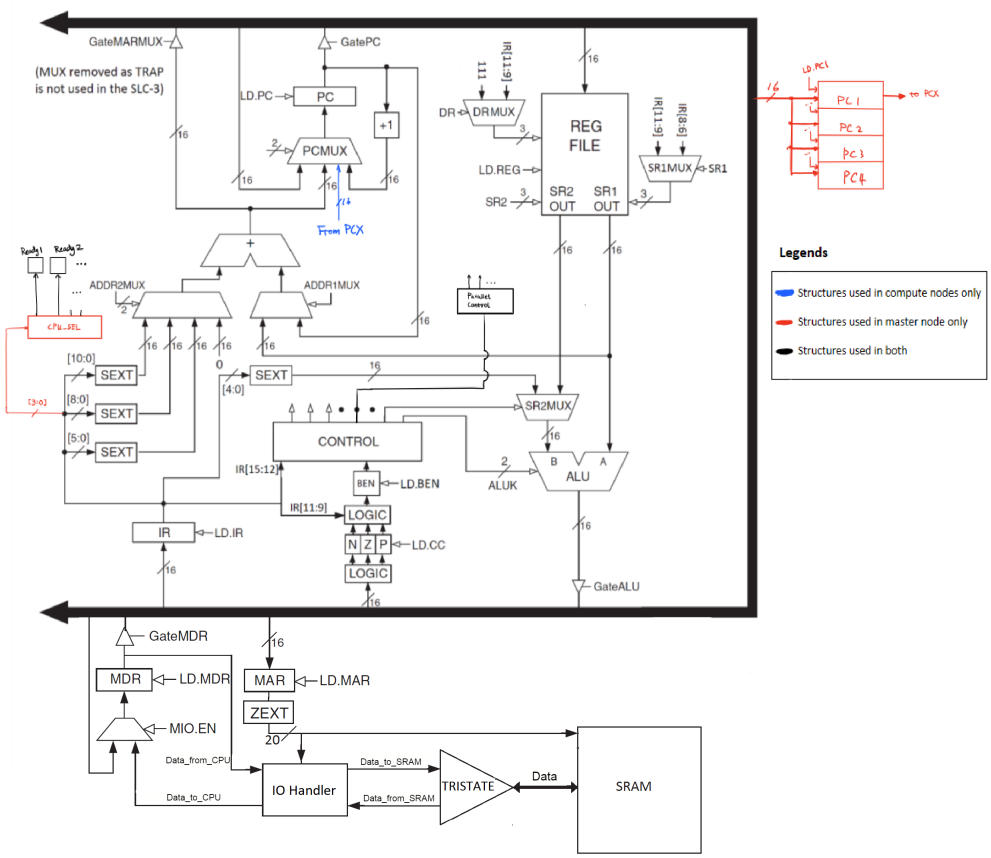

Figure 2.5: SLC-3 Data Path. NOTE!!! Change this diagram. CPU_SEL is wrong, Ready signals are not coming from CPU_SEL.

registers with the value from the specified register in the register file (REG_FILE). The MSB in PC_X corresponds to PC_1, the LSB corresponds to PC_4, similarly in between.

1101 | 000 | REG_FILE[3 bits] | 00 | PC_X [4 bits]

2. **CPU_SEL** — 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.

3. **SYNC** — 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.

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

5. **WAIT** — 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.

6. **INTR_PC** — Immediately interrupt (load) the PC with the value coming from the PC_INIT output.

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

2.1.4 Instruction Set Decoder Unit

The ISDU is a Moore finite state machine (FSM) and defines the internal state for each core. All operations that require SRAM access are split into two states due to timing constraints. The first state is to modify the address bus to the appropriate value. The second is to enable any read/write signals to SRAM. Any instructions that require memory IO access (i.e. Mem_OE, Mem_WE) are shifted one state forward to compensate for the time the signals need to pass through the memory control signal synchronizers. These synchronizers are needed for SRAM because it is an asynchronous device. Glitches in the control signals could be fatal to data stored on the memory. Since the SLC3 ISA is significantly smaller than that of the LC3's, only a handful of operations are required: ADDi, ANDi, ADD, AND, NOT, LDR, STR, JSR, JMP, BR, PSI (parallel-specific instructions).

Based on the original FETCHDECODEEXECUTE 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 uppermost 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 FETCHDECODE PSI_DECODEEXECUTE. 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 3̈6ïs 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.

Chapter 3

Example Parallel Assembly Program

The following code provides an example assembly program. This code uses a master-slave configuration of the cores with core 0 acting as the master. The master instructs all of the slaves to load their program counters with the beginning address of the slave kernel. Then, each slave will determine its rank from the hard-wired value given to it (somewhat similar to MPI's rank). Each core will then increment a single element of an array a finite number of times, effectively performing a one-to-one operation. When the cores are finished executing, they will return the READY signal back to the master, at which time the program will halt.

```
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)
18
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 a reference 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 SystemVerilog code that initializes the memory array with the correct binary information.

Chapter 4

Results

4.1 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.

4.2 Simulation Waveforms

The simulated waveform in modelsim is presented in Figure 4.1. 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 20 the beginning of execution (top waveform), this array shows all 0s. 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.

4.3 Design Statistics

Table 4.1 describes the profile of our PSLC3 implementation.

Lookup Tables	1843+1132=2975
Digital Signal Processors	0
BRAM	0
Flip-flop count	1380
Maximum Frequency	82.67 MHz
Static Power	98.77 mW
Dynamic Power	0 mW
Total Power	145.41 mW

Table 4.1: Design Statistics

4.4 Conclusion

The implementation of this parallel SLC3 architecture was successful. Each core was capable of independently executing arbitrary code within the memory space provided, and communication paradigms between the designated master and slave cores allows the programmer to write many types of assembly programs appropriate for a master-slave configuration. One of the major bottlenecks in this design was the use of the shared memory space. Although the queuing mechanism for the IO handler worked, the overall memory latency each core experienced was unreasonably high. This is an unavoidable outcome from the fact that 5 cores are now competing for the same resource, thus slowing down operations 5 times. Ever since compute cores began to outpace the performance of RAM around the 1980's and 1990's, memory has always served as a bottleneck and to this day acts as a major limiting reactant for memory-bound applications.

One of the answers to this problem is to implement a memory caching system. Most modern CPUs have multiple levels of memory cache (often 3) split amongst the cores in varying configurations. This allows the memory access to become distributed and more parallel. It is in the author's opinion that the next logical step in the evolution of the PSLC3 is to implement this caching mechanism to speed up the memory access times.

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 implementation of SMP support in an actual modern CPU, it provides a good learning platform for rudimentary parallel processing concepts.

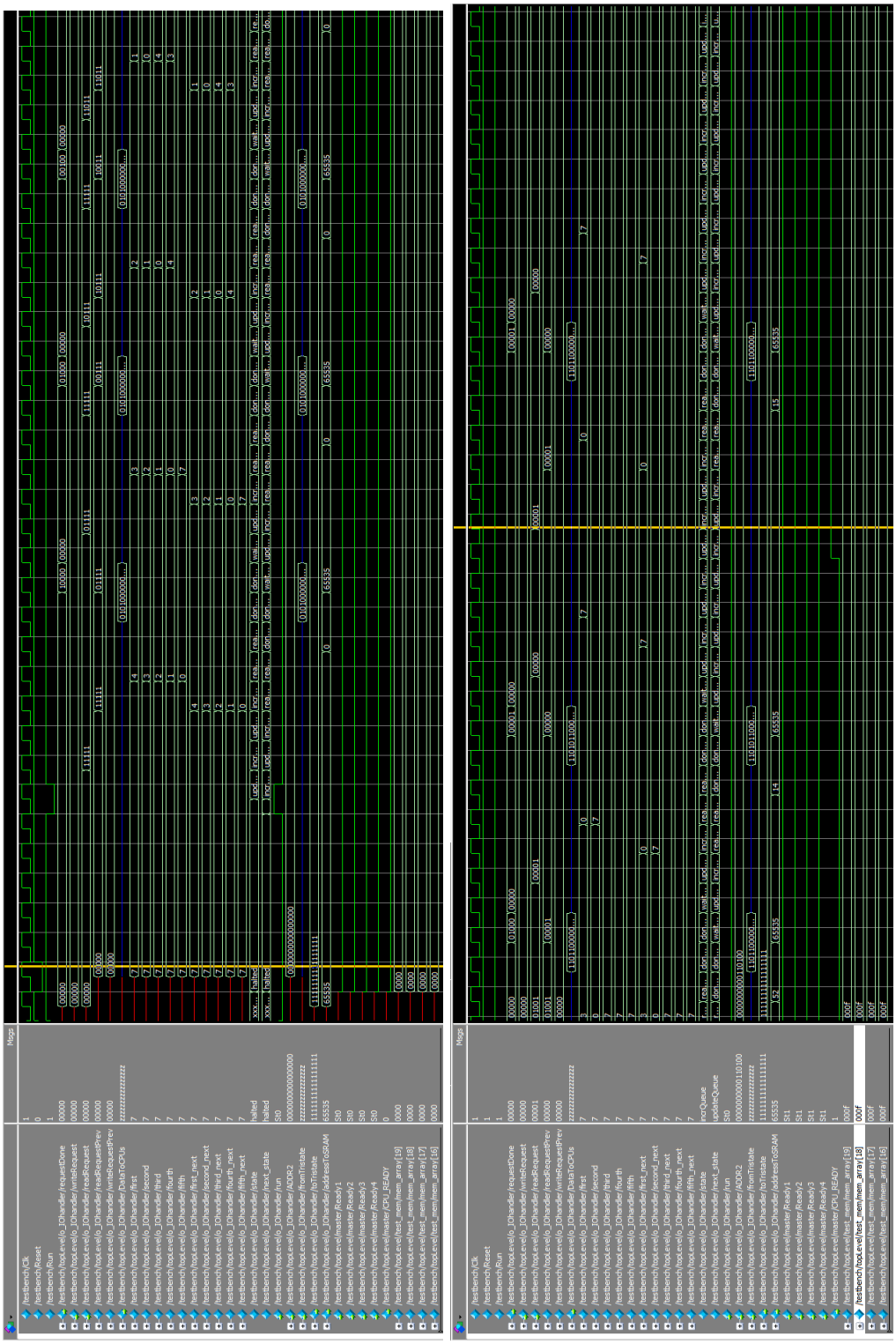

Figure 4.1: Simulation Waveform

Bibliography

[1] Patt, Yale N.; Patel, Sanjay (2003). *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. New York, NY: McGraw-Hill Higher Education. ISBN 0-07-246750-9. Retrieved 1 May 2017.