Evan Apinis
ECE 5724 Homework 1
2/4/24

1. Introduction

For this assignment, we were tasked with creating a selection sort algorithm based on a simple comparison-based sorting algorithm. The sorting algorithm uses a memory buffer containing the elements to be sorted and the physical memory space gets broken into two virtual spaces. One is the elements that have already been sorted by smallest to largest, and the other is the elements that still need to be sorted. The selection sort algorithm will iterate through every unsorted element to determine the smallest element before swapping it with the first element in the unsorted memory space. This process will continue to loop until the remaining unsorted memory space becomes zero, at which point all elements will be sorted. In this assignment, we were to implement the selection sort algorithm and meet the following four goals.

- Show Verilog description of the components of the datapath and then enclose them in the complete datapath.
- Show Verilog description of the controller according to the provided FSM, using Huffman modeling style.
- In a top-level Sorter Circuit architecture, put the datapath and controller of the circuit together.
- Generate an appropriate testbench to test your design (Control the duration of running a testbench, using “$stop or $finish”).

2. SSC Controller

The SSC controller implements a state machine and two 8-bit counters to control the multiplexers and memory buffer found in the datapath and wrapper modules respectively. The state machine was implemented using the Huffman model, which used combinational logic to determine the next state and current outputs. Then clocked on the rising edge a synchronous logic block is used to transition the state machine to its next state. The state machine stays in its Idle state until a start pulse is asserted externally to the ssc wrapper. The state machine then remains in the Start state until a reset signal is applied or the start pulse ends and goes low. When the start pulse returns low the state machine transitions into its Init state. This is where the sorting algorithm begins. A pair of counters is used to keep track of how many elements in the buffer have been sorted and when indexed to address in the buffer for reading new data. The state machine will remain in a find minimum state to index through the entirety of the unsorted buffer to identify the smallest element. The ssc_controller.sv module contains only the state machine and the counters used to control the addressing of the memory buffer. It does not

contain any additional logic pertaining to the actual comparison of data, these modules are found in the datapath. The Visio model used to create the controller can be seen in Figure 1.

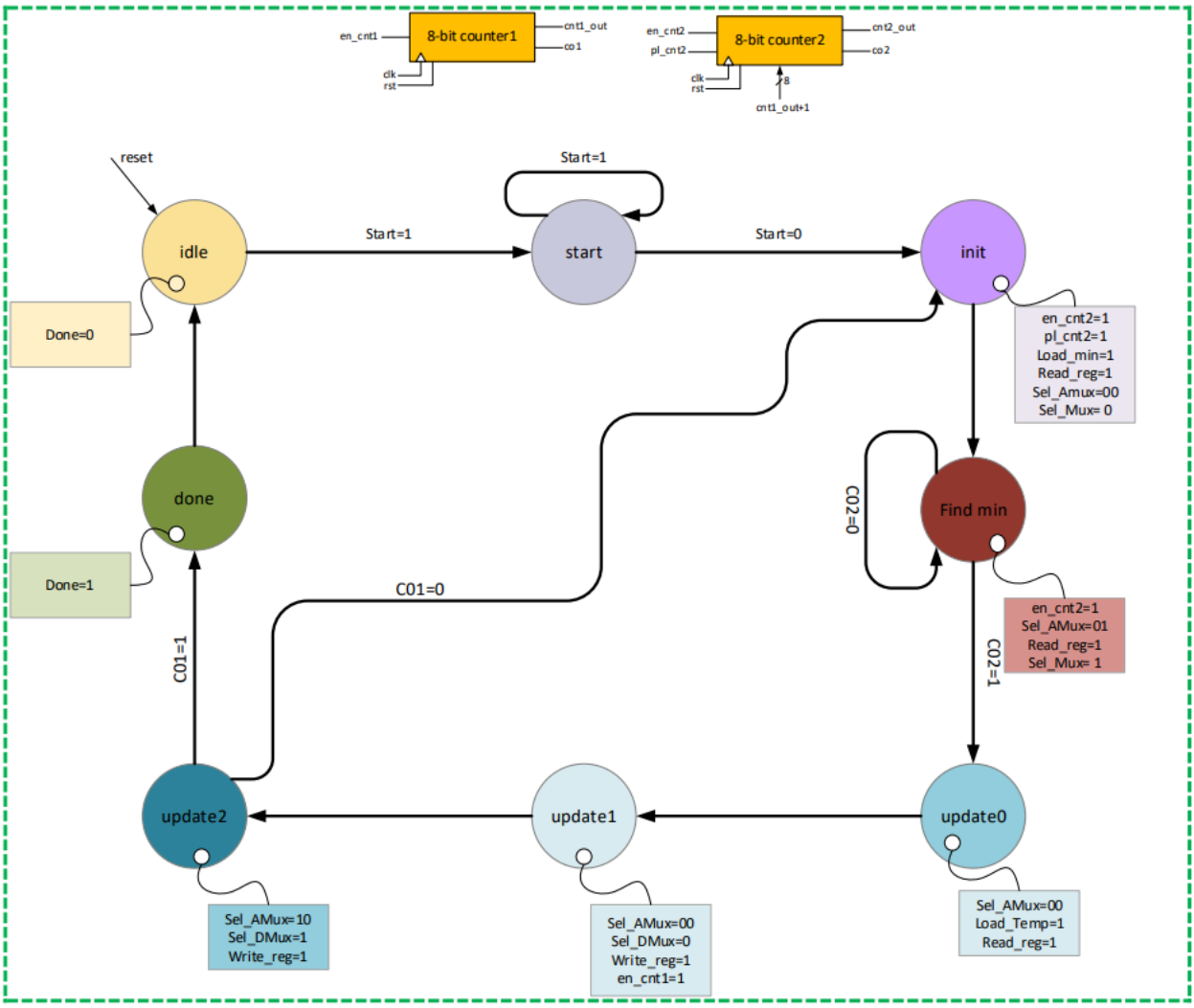

Figure 1. Selection Sort Controller State Machine Diagram and Design.

3. SSC Datapath

The SSC Datapath or ssc_datapath.sv module contains several smaller module instantiations and is very similar to the Visio drawing provided in the assignment details. The only modifications are the fact that the memory buffer has been removed and instantiated a level higher in the wrapper to make its internal ports more visible to the wrapper's inputs and outputs for testbench verification. The datapath, otherwise employs all of the logic seen in the Visio drawing. It instantiates three multiplexers for the address mux, data mux, and counter mux. In the module logic, there is the implementation of the three registers for holding the minimum address and value as well as the temporary data to be relocated in the memory buffer. A comparator is also

implemented in the datapath to determine if the read data value is larger or smaller than the current minimum value. It is in the ssc_datapath module where you will see all of the physical verilog modules and logic that is performing the comparisons and moving memory buffer data around the ssc module.

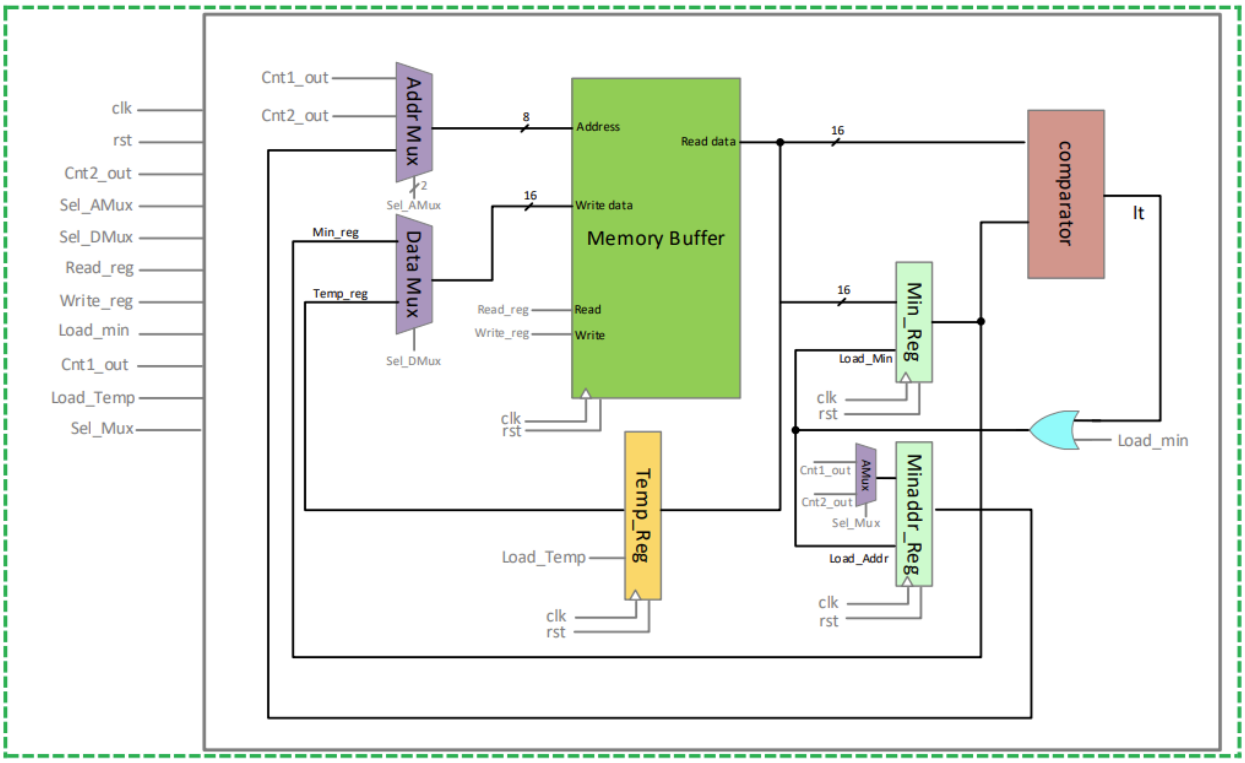

Figure 2. Provided Datapath for Selection Sort Algorithm

4. SSC Wrapper

The SSC Wrapper or ssc_wrapper.sv module is the top-level module of the entire design. As such it contains the ssc_datapath.sv and ssc_controller.sv modules. It also contains the mem_buffer.sv module which is the memory buffer containing the 16-bit numbers to be sorted. This buffer was included in the wrapper for easier access to its ports through the top-level inputs and outputs to allow for debugging and verification by the testbench. With the memory buffer in the top-level, a group of assign statements are implemented in the wrapper to allow for the use of debug inputs for testbench verification. Apart from this, the wrapper contains only the three instantiations and their internal net connections.

5. Simulation & Verification

A testbench was created to verify the sorting algorithm by reading and comparing the contents in the memory buffer after the controller asserts the done signal. Additional input and outputs were

created in the top-level ssc_wrapper module to allow for debugging the design in the testbench. These included a debug enable flag which was used as a selection for a pair of multiplexers. These multiplexers determined the address and read enable nets that interfaced with the memory buffer. This allowed for manually indexing through the buffer following the completion of the sorting algorithm. By default, without the debug enable input set high the design uses its traditional datapath. The last additional output for the design was the Read_Data net which was made an output to be visible in the testbench for verification. Included with the debug enable flag were the associated debug signals to give control over the memory buffer to the testbench for verification purposes. All of these nets can be found on the top level of the design in the ssc_wrapper.sv module with the name Debug_xxxx.

5.1. Mid-Execution Reset

This test was executed to evaluate the functionality of the state machine and how it reacts to a reset in the middle of its sorting state. In doing so, we verify that the Verilog description has been properly written for reset conditions. Not only should the state machine move back to the Idle state but the registers in the datapath and counters in the controller need to be properly reset to avoid writing new data into the memory buffer. It can be seen that three words were sorted before the reset signal was applied. The values in addresses 0x00 through 0x02 were arranged prior to the application of the reset signal. Following the reset, the state machine begins where it started once more and begins sorting the buffer from the starting address of 0x00 after a start signal is applied. Based upon the state machine leaving the first three sorted items untouched it can be determined that all registers, counters, and outputs were properly reset. If this were not the case we would expect to see conflicting writes to memory before beginning to sort the remainder of the buffer from address 0x03 onwards.

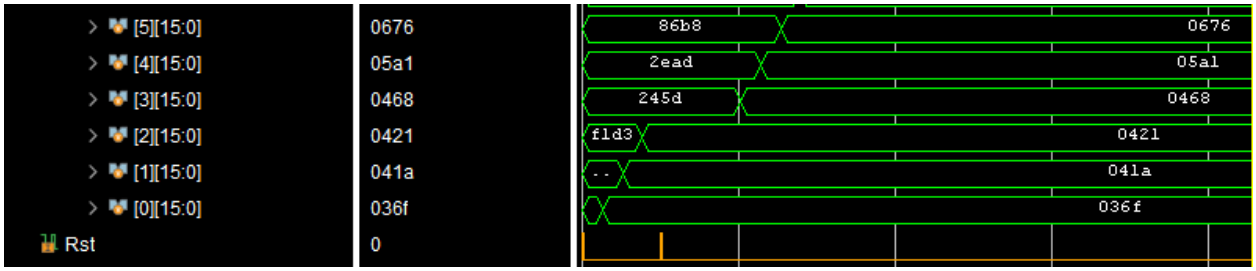

Figure 3. Waveform Results From Applying Reset in Execution

Once the state machine finished iterating through the memory buffer the testbench read the contents of the buffer to test for the correct sorting order. This was a second test of the design for the first run but it was a part of the fourth test that will be described later. It was done to verify that the algorithm worked correctly and was able to re-arrange the contents in the memory buffer from smallest to largest. The results of the run indicate that for this given test the contents were sorted properly.

```
Current 62562 Previous 62737 OK 1
Current 62737 Previous 62796 OK 1
Current 62796 Previous 62948 OK 1
Current 62948 Previous 62960 OK 1
Current 62960 Previous 63046 OK 1
Current 63046 Previous 63153 OK 1
Current 63153 Previous 63274 OK 1
Current 63274 Previous 63767 OK 1
Current 63767 Previous 63769 OK 1
Current 63769 Previous 63837 OK 1
Current 63837 Previous 64134 OK 1
Current 64134 Previous 64345 OK 1
Current 64345 Previous 64701 OK 1
Current 64701 Previous 64828 OK 1
RUN 1 PASSES
```

Figure 4. Testbench Results From Testing Reset.

5.2. Repeated Sorts (Elements Already In Proper Order)

This test was performed after the state machine made it to its Done state, indicating that it finished executing the sorting algorithm. In this test, we apply a start signal once the state machine reaches its Idle state to sort through the memory buffer again. In this instance, the memory buffer is ideally in the correct order after the first sort, but by sorting it once more we can show that counter 1 and counter 2 are enabled properly by the state machine. Additionally, it shows that the minimum address and data registers are functioning correctly. If the counters were not being incremented correctly we could have seen the first index being missed and the contents become out of order over time. This is due to the controller design with counter 2 being loaded with the sum of the value in counter 1 and the numerical value of 1. This will verify the corner case where the item to be sorted is already in the correct spot because the counter 2 value, or memory buffer address, should begin at the lowest data value left unsorted, and therefore be stored in the minimum registers. From the results of the test, we see that the results of the second sorting are identical to the first. This means that the items arranged in the buffer after the first run remained untouched and did not become out of order from the first run, verifying that items already in the correct order will not be affected.

```
Current 62796 Previous 62948 OK 1
Current 62948 Previous 62960 OK 1
Current 62960 Previous 63046 OK 1
Current 63046 Previous 63153 OK 1
Current 63153 Previous 63274 OK 1
Current 63274 Previous 63767 OK 1
Current 63767 Previous 63769 OK 1
Current 63769 Previous 63837 OK 1
Current 63837 Previous 64134 OK 1
Current 64134 Previous 64345 OK 1
Current 64345 Previous 64701 OK 1
Current 64701 Previous 64828 OK 1
RUN 2 PASSES
```

Figure 5. Testbench Results After Sorting Arranged Elements

5.3. Simultaneous Read and Write

One of the design principles for a single port memory buffer is the idea that you cannot perform reads and writes simultaneously. This is because both actions share the same data address and the data that is read from the buffer in this situation is unpredictable and potentially unstable. As such the buffer should only perform either a read or a write during a single clock cycle, but not both. In this test, the design of the memory buffer was examined by attempting to perform a read and write at the same time. The memory buffer was written to write data into the buffer with an enable signal only if the read enable signal is not asserted. The same behavior was implemented with the read port using the read enable signal. If both the read and write enables are asserted the buffer should not change any contents in the buffer and the read data port would output its previous contents.

```
This is Run 3: Testing Read and Write at the Same Time
Address 7f Read_Data 8835 Write_Data 0000 Read 1 Write 0
Address 80 Read_Data 8883 Write_Data 0000 Read 1 Write 0
Address 7f Read_Data 8883 Write_Data ffff Read 1 Write 1
Address 7f Read_Data 8835 Write_Data ffff Read 1 Write 0
RUN 3 PASSES
```

Figure 6. Testbench Results of Simultaneous Reads and Writes to Memory Buffer.

The simulation results show that the data stored in the buffer at address 0x7f was 0x8835 after the state machine finished sorting. The data stored at address 0x80 was 0x8883. In the following clock cycle a read and write was attempted at address 0x7f. The write data was intended to be 0xffff for debug purposes. Given the simultaneous read and write it can be seen that the read data was 0x8883, or the data that was previously output by the buffer. In the following clock cycle where a single read was performed at the same address, it can be seen that the data read was the

original 0x8835 and not the write data. This behavior was by design and verifies the functionality of the memory buffer in the design of the sorting algorithm.

5.4. Correct Order of Sorting

The final test was a culmination of the first two tests and it was to verify that the memory buffer was properly sorted at the assertion of the done signal. In doing this we are able to see that the entirety of the system works properly under standard conditions. This includes the controller, memory buffer, and all other subcomponents of the datapath. To complete this test the start signal was applied and the test bench would wait until the assertion of the done signal by the controller. This indicates that the state machine was finished sorting the contents of the buffer. To verify that the items are in the correct order the debug ports added to the wrapper were used to interface with the memory buffer directly. Through this, the testbench will read the entire buffer by looping through it. During this loop, the testbench will read the memory contents from the current address and the previous address. Then it will take a comparison and verify that the current address value is larger than the value stored in the previous address. For this reason, the loop begins at address 0x01 to access the value stored at this address and address 0x00. During each loop, if the value stored is larger than the previous one it will perform a logical and between the run value and the result of the greater than operation. The value is then assigned back to the run value to determine if every single comparison operation was performed correctly. From the display statements created in the testbench, it can be seen that all four runs passed, indicating that the testbench as a whole, also passed.

```
This is Test 4: Verifying Previous Runs to Determine if Buffer is Sorted Properly
RUN 1 PASSES
RUN 2 PASSES
RUN 3 PASSES
RUN 4 PASSES
TESTBENCH PASSES
```

Figure 7. Final Testbench Result Verifying Correctly Sorted Elements and Subset Simulations.