EE 371, Lab 4

Implementing Algorithms in Hardware

Lab Objectives

In this lab you will use algorithmic state machine charts to implement algorithms as hardware circuits.

Introduction

Algorithmic State Machine (ASM) charts are a design tool that allow the specification of digital systems in a form similar to a flow chart. An example of an ASM chart is shown in Figure 1. It represents a circuit that counts the number of bits set to 1 in an n-bit input A ($A=a_{n-1}a_{n-2}\cdots a_1a_0$).

The rectangular boxes in this diagram represent the *states* of the digital system, and actions specified inside of a state box occur on each active clock edge in this state. Transitions between states are specified by arrows. The diamonds in the ASM chart represent conditional tests, and the ovals represent actions taken only if the corresponding conditions are either true (on an arrow labeled 1) or false (on an arrow labeled 0).

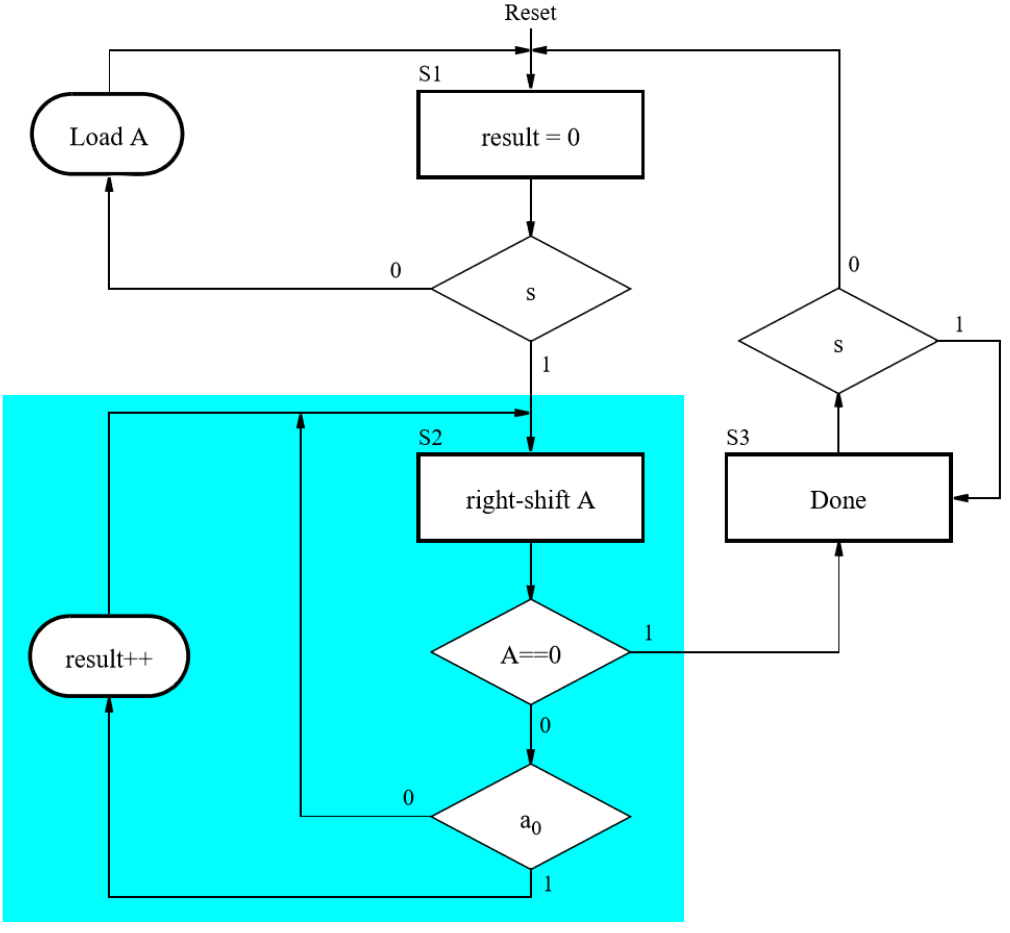

Figure 1. ASM chart for a bit counting circuit

In this ASM chart, state *S*1 is the initial state. In this state the *result* is initialized to 0, and data is loaded into a register *A*, until a start signal, *s*, is asserted. The ASM chart then transitions to state *S*2, where it increments the *result* to count the number of 1's in register *A*. Since state *S*2 specifies a shifting operation, then *A* should be implemented as a shift register. Also, since the *result* is incremented, then this variable should be implemented as a counter. When register *A* contains 0 the ASM chart transitions to state *S*3, where it sets an output *Done* =1 and waits for the signal *s* to be deasserted.

A key distinction between ASM charts and flow charts is a concept known as *implied timing*. The implied timing specifies that all actions associated with a given state take place only when the system is in that state when an active clock edge occurs. For example, when the system is in state *S*1 and the start signal *s* becomes 1, then the next active clock edge performs the following actions: initializes *result* to 0, and transitions to state *S*2. The action *right-shift A* does not happen yet, because the system is not yet in state *S*2. For each active clock cycle in state *S*2, the actions highlighted in Figure 1 take place, as follows: increment *result* if bit a_0=1, change to state *S*3 if *A*=0 (or else remain in state *S*2), and shift *A* to the right.

The implementation of the bit counting circuit includes the counter to store the *result* and the shift register *A*, as well as a finite state machine. The FSM is often referred to as the *control* circuit, and the other components as the *datapath* circuit.

Task 1

Write SystemVerilog code to implement the bit-counting circuit using the ASM chart shown in Figure 1 on the DE1-SoC board. Include in your code the datapath components needed, and make an FSM for the control circuit. The inputs to your circuit should consist of an 8-bit input connected to slide switches SW_{7-0}, a synchronous reset connected to KEY_0, and a start signal (*s*) connected to switch SW_9. Use the 50 MHz clock signal provided on the board as the clock input for your circuit. Be sure to synchronize the *s* signal to the clock. Display the number of 1s counted in the input data on the 7-segment display HEX0, and signal that the algorithm is finished by lighting up $LEDR_9$.

Task 2

We wish to implement a binary search algorithm, which searches through an array to locate an 8-bit value *A* specified via switches SW_{7-0}. A block diagram for the circuit is shown in Figure 2.

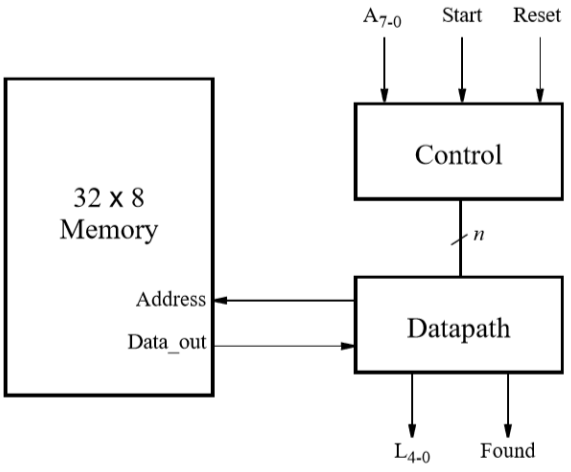

Figure 2. A block diagram for a circuit that performs a binary search.

The binary search algorithm works on a sorted array. Rather than comparing each value in the array to the one being sought, we first look at the middle element and compare the sought value to the middle element. If the middle element has a greater value, then we know that the element we seek must be in the first half of the array. Otherwise, the value we seek must be in the other half of the array. By applying this approach recursively, we can locate the sought element in only a few steps.

In this circuit, the array is stored in a memory module that is implemented inside the FPGA chip. A diagram of the memory module that we need to create is depicted in Figure 3. In a similar fashion to the first task of lab 2, create a memory that is eight-bits wide and 32 words deep.

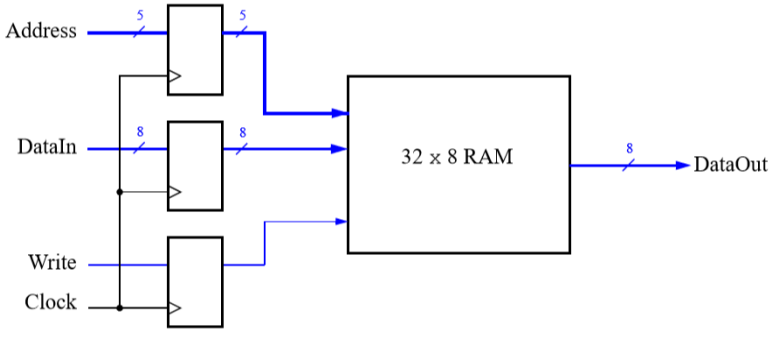

Figure 3. The 32 x 8 RAM with address register

To place data into the memory, initialize the memory using the contents of a *memory initialization file (MIF)* and call it *my_array.mif*, which then has to be created in the folder that contains the Quartus project. Set the contents of your *MIF* file such that it contains a sorted collection of integers. Your circuit should produce a 5-bit output *L*, which specifies the address in the memory where the number *A* is located. In addition, a signal *Found* should be set high to indicate that the number *A* was found in the memory, and set low otherwise.

Perform the following steps:

1. Create an ASM chart for the binary search algorithm. Keep in mind that the memory has registers on its input ports. Assume that the array has a fixed size of 32 elements.

2. Implement the FSM and the datapath for your circuit.

3. Connect your FSM and datapath to the memory block as indicated in Figure 2.
4. Include in your project the necessary pin assignments to implement your circuit on your DE-series board. Use switch SW_9 to drive the *Start* input, use $SW_{7...0}$ to specify the value *A*, use KEY_0 for *Resetn*, and use the board's 50 MHz clock signal as the *Clock* input (be sure to synchronize the *Start* input to the clock). Display the address of the data *A*, if found, on 7-segment displays *HEX*1 and *HEX*0, as a hexadecimal number. Finally, use $LEDR_9$ for the *Found* signal.
5. Create a file called *my_array.mif* and fill it with an ordered set of 32 eight-bit integer numbers.
6. Compile your design, and then download and test it.
7. Use the SignalTap II functionality of Quartus to verify the contents of your RAM module. Find the Intel SignalTap II tutorial on canvas (SignalTap.pdf) to get started. For your RAM module, you'll probably want to probe the data loaded via *my_array.mif*.
8. Demonstrate to your TA.

Lab Demonstration and Submission Requirements

- Submit a lab report that includes the procedures and results obtained in the lab. Suggestions on what to include follow:
  - Abstract: A brief overview of the entire report
  - Introduction: What the lab is (specifications or background info, etc.
  - Procedure/Results/Analysis:
    - Steps to complete the lab
    - Description of each module (explain how they work, point out code, etc.)
    - Simulations of each module with comments on important things to notice about them.
    - Demonstarte how to use SignalTapII verify the contents of the requried modules.
    - Overall description of what the finished product was/how it behaved
    - Discussions of any problems had while completing the lab, or unsolved errors
  - Conclusion: A final summary, reflection on the lab or what was learned, etc.
- Include any hurdles or challenges (if any) that you faced in finishing this lab and how you overcome them.
- Submit the SystemVerilog code for all tasks and include screenshots for the ModelSim waveforms of all modules.
- Submit the Flow Summary (produced during compilation) of compiling your system.
- In your report, include the number of hours (estimated) it took to complete this lab, including reading, planning, design, coding, debugging, testing, etc. Everything related to the lab (in total).
- Submit your report and programs to Canvas. No hard copies. Submit a pdf file as well as a compressed folder of your source files.