Laboratory Exercise 13

Open Ended Lab

Administrivia

1. Open Ended Lab

An open-ended lab is where students are given the freedom to develop their own solution, instead of merely following the already set guidelines from a lab manual or elsewhere. The teacher gives the students an objective/purpose and not the procedure. The students would then have to come up with their own solution to fulfill the purpose.

2. Learning Outcomes

By the end of this lab you will be able to:

(a) Solve daily life problems through FPGA.

(b) Implement algorithms as hardware circuits.

(c) Propose your engineering solution to the problem.

3. Deliverable

You are required to submit

- Appropriately Commented code.
- Answer to the questions asked in the lab tasks.
- Issues in Developing the Solution and your Response in the beginning of next lab.

Laboratory Exercise 13

Open Ended Lab

Implementing Algorithms in Hardware

This is an exercise in using algorithmic state machine charts to implement algorithms as hardware circuits.

Background

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 ($A = a_{n-1}a_{n-2}...a_1a_0$). 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).

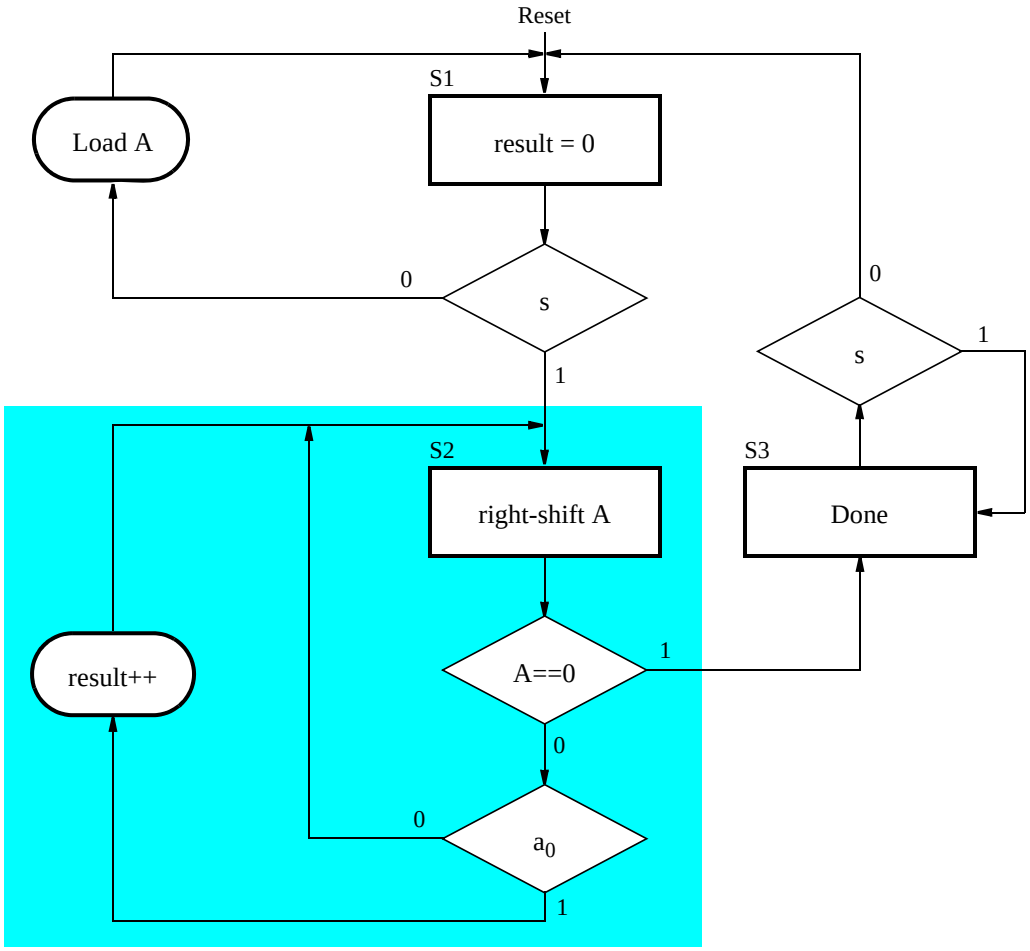

Figure 1: ASM chart for a bit counting circuit.

In this ASM chart, state $S1$ 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 $S2$, where it increments the *result* to count the number of 1's in register A. Since state $S2$ 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 $S3$, 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 $S1$ 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 $S2$. The action *right-shift A* does not happen yet, because the system is not yet in state $S2$. For each active clock cycle in state $S2$, the actions highlighted in Figure 1 take place, as follows: increment *result* if bit $a_0 = 1$, change to state $S3$ if $A = 0$ (or else remain in state $S2$), 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.

Part I

Write Verilog code to implement the bit-counting circuit using the ASM chart shown in Figure 1 on a DE-series board. Include in your Verilog 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 $LEDR_9$.

Part II

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.

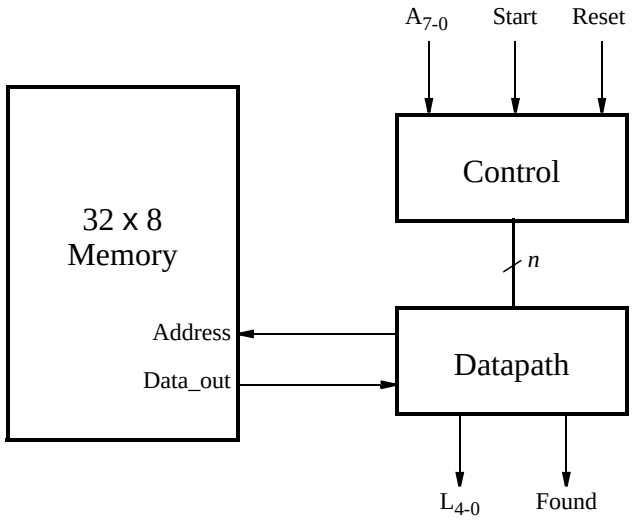

Figure 2: A block diagram for a circuit that performs a binary search.

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. This memory module has one read port and one write port, and is called a *synchronous random-access memory (synchronous RAM)*. Note that the memory module includes registers for synchronously loading addresses, input data, and the *Write* input. These registers are required due to the design of the memory resources in the Intel FPGA chip. Use the Quartus IP Catalog tool to create the memory module, by clicking on Tools > IP Catalog. In the IP Catalog window choose the *RAM: 1-PORT* module, which is found under the Basic Functions > On Chip Memory category. Select Verilog HDL as the type of output file to create, and give the file the name *memory_block.v*.

Follow through the provided sequence of dialogs to create a memory that is eight-bits wide and 32 words deep. Figures 4 and 5 show the relevant pages and how to properly configure the memory.

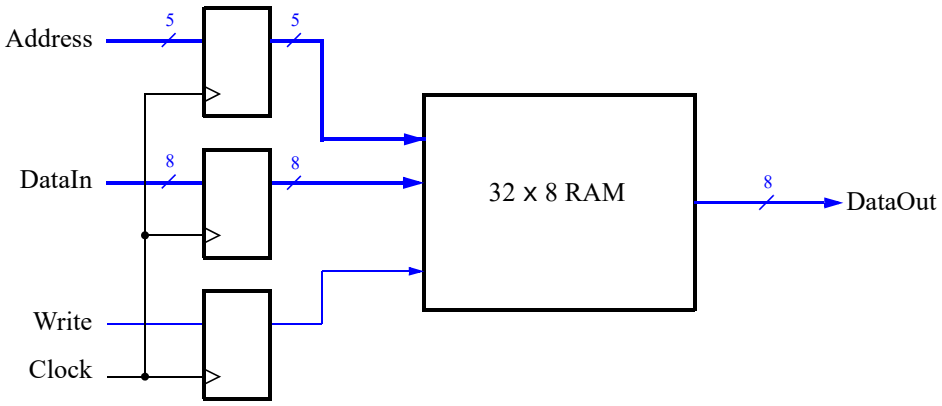

Figure 3: The 32 x 8 RAM with address register.

To place data into the memory, you need to specify *initial values* that should be stored in the memory once your circuit has been programmed into the FPGA chip. This can be done by initializing the memory using the contents of a *memory initialization file (MIF)*. The appropriate screen is illustrated in Figure 6. We have specified a file named *my_array.mif*, which then has to be created in the folder that contains the Quartus project. An example of a memory initialization file is given in Figure 7. Set the contents of your *MIF* file such that it contains a sorted collection of integers.

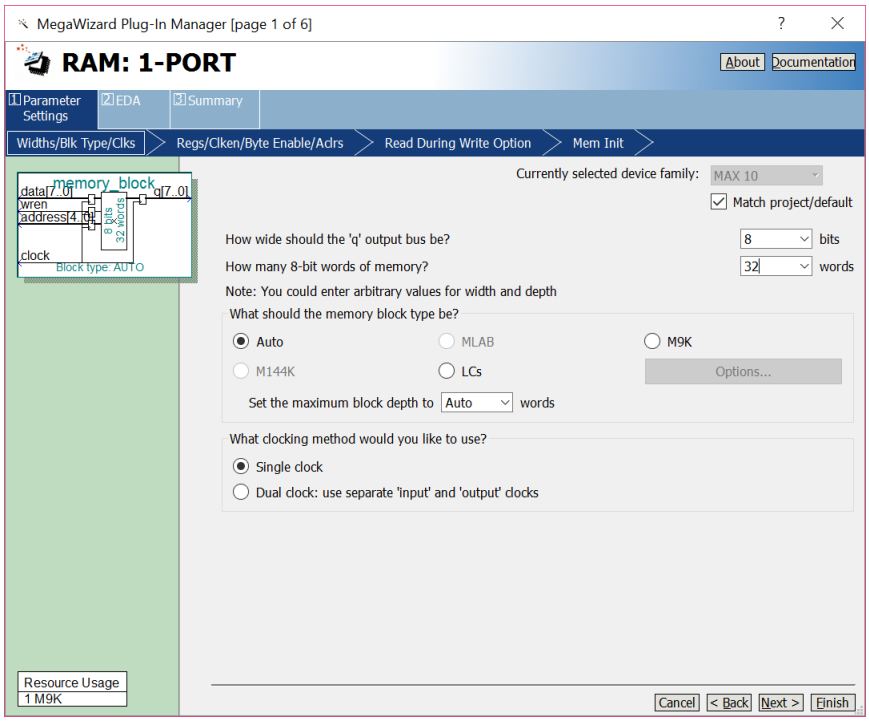

Figure 4: Specifying memory size.

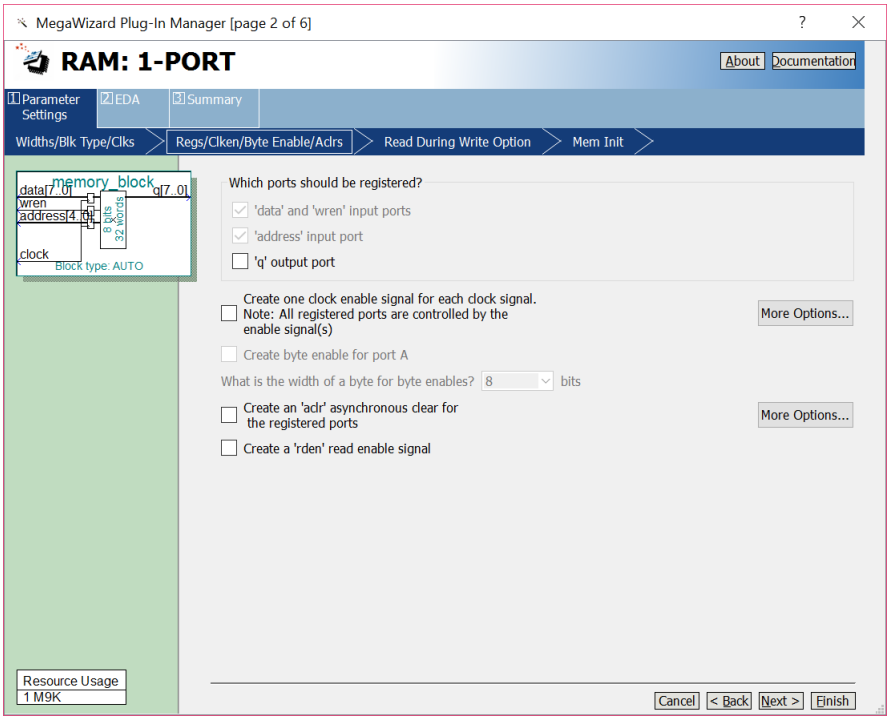

Figure 5: Specifying which memory ports are registered.

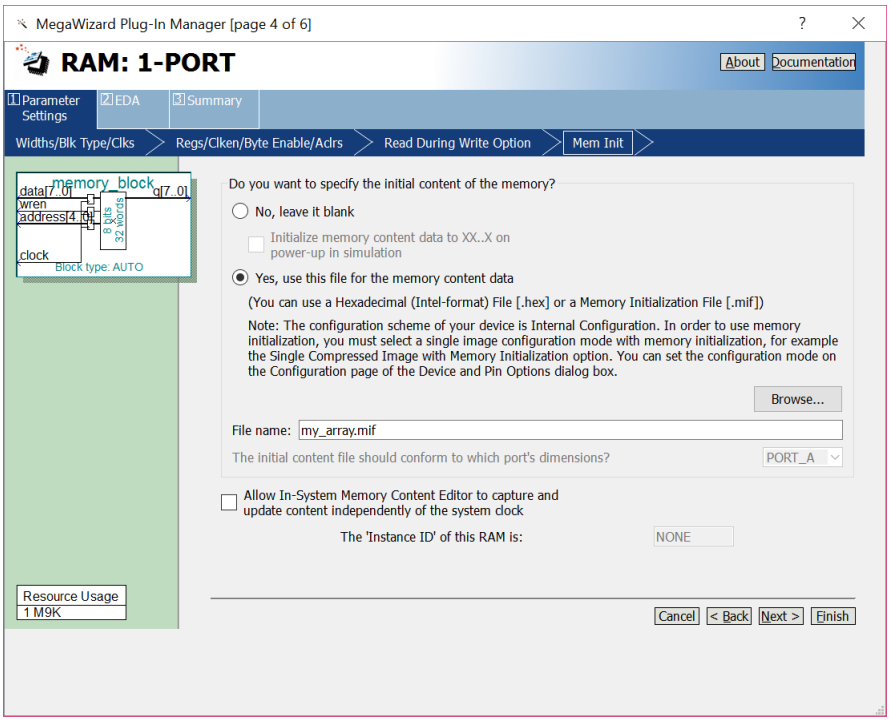

Figure 6: Specifying a memory initialization file (MIF).

```
DEPTH = 32;
WIDTH = 8;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

00 : 01;
01 : 02;
02 : 03
03 : 05;
04 : 06;
05 : 06;
06 : 07;
. . . (some lines not shown)
1E : 1F;
1F : 20;

END;
```

Figure 7: An example memory initialization file (MIF).

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.