TEST YOUR KNOWLEDGE – FlexKR

Last updated: **[May 10, 2022]**

Contents

1.0 Introduction

1.1 Prerequisites

- Completion of Introduction to Intel FPGAs and Quartus Software course
    - Quartus 18.1
        - You will need 3 files. These are in the individual files tab on the download page. Do not download the .tar file! Download the following(You may need to accept T&C and/or login with your Intel account:
            - ModelSim-Intel® FPGA Edition (includes Starter Edition)
            - Intel® Quartus® Prime (includes Nios® II EDS)
            - Intel® MAX® 10 Device Support
        - Double click the Quartus executable to install all 3 files. It will select ModelSim and the Max10 device during install automatically from your downloads folder.
    - Please complete the following labs prior to attempting this one:
        - OUWINTRO
        - Knight Rider ROM
- Basic knowledge of digital logic design, state machines and Verilog HDL.

1.2 Reference Documents

Table 1-1. Reference Documents

The starting file for this project is a working knight_rider project from part 3 of the Introduction to Intel FPGAs and Quartus Prime Software course. This .zip file has pin assignments set for a DE10-Lite board.

If you have a different FPGA board (such as the DE1-SoC), you will need to do project set up and pin assignments **based on previous projects**. You can also reassign the Pins manually using the Pin Planner and the appropriate schematic from the downloadable CD-ROM on terasIC's website.

Document	Document No./Location
FlexKR Project File Name	FlexKR.qar

2.0 Lab Guide

Now that you have completed the Knight Rider and Knight Rider ROM labs, this lab should really test your knowledge of digital logic design. This lab is designed to instruct you on memory initialization, memory mapping, finite state automata and logic design principles.

This lab is a sequencer for a row of LEDs. As in the previous Knight Rider labs, all 10 LEDs are to be utilized on the DE10-Lite board. The design uses the 6 seven segment displays to display the ROM address, clock frequency (via a clock divider variable), and the sequence number. All three outputs each use 2 seven segment displays. There are 2 push buttons to control the sequence up/down commands. There are 3 of the 10 switches utilized. Switches 0-1 are the frequency up/down inputs and switch 2 is a system reset. Please see Part 3 for diagrams. //change to keys and change code to a switch for push_buttons to control both freq and seq selection

2.1 What Is Provided?

You are provided with a file FlexKR.sv, a System Verilog file, which contains multiple modules already defined for you. These are the modules and their intended usage:

- FlexKR(…);
    - This module is out top-level module. It is designed to initialize all the other modules provided, including the stub for the state machine you will be implementing as your contribution to this project.
    - Sensitivity List:
        - clk_50 – the 50MHz clock from the system
        - reset – wired to switch[9]. Used as a system reset
        - [3:0] push_button – [0-1] are wired to the 2 push buttons on the board and [2-3] are wired to switch[0-1]. The push buttons control the up/down of the sequences stored in memory. The 2 switches are the up and down for the CLOCK_DIVIDER constant that you modified in a previous Knight Rider lab.
        - ROM_addr_seven_seg_2_to_0,
          ROM_addr_seven_seg_2_to_1,
          freq_seven_seg_ones,
          freq_seven_seg_tens,
          seq_selection_seven_seg_ones,
          seq_selection_seven_seg_tens – these are the seven segment displays on the board. The order is:
            - [0-1] – seq_selection_*
            - [2-3] – freq_seven_*
            - [4-5] – ROM-addr_*

        - [9:0] LED – These are the 10 LED on the board. They will blink when the state machine is correctly implemented.
- function_table(…);
    - This is the module that needs to be implemented. This will be explained further in the lab manual.
    - Sensitivity List:
        - clk – 50MHz clock input.
        - reset – reset signal
        - next – a signal that is passed by the driver module to indicate if the next sequence was detected from user input.
        - previous – a signal from the driver to indicate the previous sequence has been requested.
        - init – the initial bit to indicate the need to read from the ROM and store the data in a register array.
        - data_out_rom – contains the LED 10-bit sequence as bits [11-2]. The 2 least significant bits are the [1] End of Sequence bit and [0] End of ROM bit.
        - done – EOF flag bit for state machine to move from read mode to user input mode.
        - start_addr – the start address for the current sequence.
        - end_addr – the end address for the current sequence.
        - address – sent to the ROM to get data back in data_out_rom. Also the next address. *The address will most likely be the next address and not the address matching the data in data_out_rom. The data take 2 clock cycles to come back to the function table once we update address. You can see this in a simulation.
        - current_sequence – used to pass address to module to convert from binary to decimal, then to the seven-segment display.
- divide_clock(…);
    - This module slows the clock to and allows the user to modify the COUNTER_SIZE variable with the push buttons.
    - Sensitivity List:
        - fast_clock – 50MHz clock input.
        - [1:0] debounce_out – debounced signals from the switches on the board.
        - slow_clock – the reduced speed clock.
        - freq_adj – the value used for the seven-segment display.
- Seven_segment(…);
    - This module converts the 4-bit decimal number to the output of a seven-segment display.
    - Sensitivity List:
        - led_bcd – the 4-bit input from the output of the binary to decimal module. This is the number that will appear on the seven-segment display.

        - led_out – signal out to the seven-segment display. Wired to the outputs in the top module that are connected to the appropriate pins.
- LED_driver(…);
    - slow_clock – used to control the speed of the LED sequence.
    - reset – reset signal.
    - enable – bit to allow LEDs to turn on.
    - start_addr – the start address of the current sequence.
    - end_add– the end of the current sequence.
    - data_from_ROM – the data of the current sequence including the 10 bits describing the LED sequence, but also the End of Sequence and End of ROM bits.
    - LED – used to store the current sequence to be displayed on the LEDs.
    - addr – current sequence address.
- add3(…);
    - This module exists as a subcomponent of eight_bit_binary_to_decimal module. It is used to help convert binary numbers to decimal by adding 3 to the bit count > 4.
    - Sensitivity List:
        - in – number going into adder
        - out – number + 3
- eight_bit_binary_to_decimal(…);
    - This module will convert an 8-bit binary number to a decimal representation to be sent to the seven-segment display.
    - Sensitivity List:
        - binary – the binary representation of the number
        - decimal_ones – the ones place of the decimal form of the number.
        - decimal_tens – the tens place of the decimal representation of the number.
        - decimal_hundreds – the hundreds place of the decimal form of the number.
- driver(…);
    - This module is a state machine that direct the "traffic" of user input from the push buttons, switches and reset signals.
    - Sensitivity List:
        - clk – the clock signal. Uses the 50MHz clock signal.
        - reset – the reset signal.
        - done – a bit to signify if the function_table has completed reading data from ROM and stored the addresses in a register array.
        - next_seq – signal to change to the next sequence from one of the push buttons that have been debounced.
        - previous_seq – signal to go back to a previous sequence.
        - faster – increase in the LED sequencing speed.
        - slower – decrease in the LED sequencing speed.

        - LED_enable – signal to allow the LEDs to turn on or remain off. LEDs do not need to be on during the function_table stage of startup.
        - init – active if the function_table is reading for the first time from the ROM.
        - next_signal – to the function_table module.
        - prev_signal – to the function_table module.
        - faster_signal – to the function_table module.
        - slower_signal – to the function_table module.

- debouncer(…);
    - This module will divide the clock speed by using a shift register. It also has a pulse detector to allow a switch to only be read once.
    - Sensitivity List:
        - noisy_switch – the input signal from the hardware.
        - clk_in – the 50MHz clock
        - debounced_pulse – the denoised and pulse detected signal to be used throughout the circuit.
- ROM(…);
    - This module was generated by Quartus IP using a 2-Port ROM. It allows for data to be read at multiple speeds. We will use it to read data using both the 50MHz clock and the slow_clock generated by the divide_clock module.
    - Sensitivity List:
        - address_a – an address in the ROM to get data from.
        - address_b – another port to get ROM data from.
        - clock_a – this clock is used to access and read data from all *_a signals.
        - clock_b – this clock is used to access and read data from all *_b signals.
        - q_a – the data at address_a.
        - q_b – the data at address_b.

You are also provided with "ROM.mif", a memory initialization file. This is like the one you created in the Knight Rider ROM Lab. The image below is an example of a generic ROM memory initialization file that happens to hold a familiar sequence. However, the structure of the file is identical to the one in the lab, so pay close attention to it's implementation.

This File has a 2-digit memory address in hexadecimal format and 12-bits of binary data. The 10 most significant bits are the sequence the LEDs will display for that memory location. The next bit is the End of Sequence bit used to flag the last memory address in the current sequence. The least significant bit is End of ROM which should stop the sequencer from going to a higher memory address if one exists since there is no initialized data past this point.

Note: To add comments to a Memory Initialization File use "—".

```
-- Copyright (C) 2022  Intel Corporation. All rights reserved.
-- Your use of Intel Corporation's design tools, logic functions
-- and other software and tools, and any partner logic
-- functions, and any output files from any of the foregoing
-- (including device programming or simulation files), and any
-- associated documentation or information are expressly subject
-- to the terms and conditions of the Intel Program License
-- Subscription Agreement, the Intel Quartus Prime License Agreement,
-- the Intel FPGA IP License Agreement, or other applicable license
-- agreement, including, without limitation, that your use is for
-- the sole purpose of programming logic devices manufactured by
-- Intel and sold by Intel or its authorized distributors.  Please
-- refer to the applicable agreement for further details, at
-- https://fpgasoftware.intel.com/eula.

-- input logic or output logic instead of wire and reg
-- If the name is the same for the net and the port you can use .* format
-- Enumerated types for FSMs


DEPTH = 256;                  -- The size of memory in words
WIDTH = 12;                   -- The size of data in bits
ADDRESS_RADIX = HEX;          -- The radix for address values
DATA_RADIX = BIN;             -- The radix for data values
CONTENT                       -- start of (address : data pairs)
BEGIN

00 : 000000000100;            -- memory address : data
01 : 000000001000;
02 : 000000010000;
03 : 000000100000;
04 : 000001000000;
05 : 000010000000;
06 : 000100000000;
07 : 001000000000;
08 : 010000000000;
09 : 100000000000;
0A : 010000000000;
0B : 001000000000;
0C : 000100000000;
0D : 000010000000;
0E : 000001000000;
0F : 000000100000;
10 : 000000010000;
11 : 000000001011; -- <- The LSB is the flag for end of ROM. The 2nd to LSB
                   --    is the flag for End of Sequence. The 10 MSB are the
                   --    LED sequence.
END;
```

2.2 What is Needed?

This lab requires you to design a state machine, called function_table. Please read the rest of the manual before attempting to complete this project to ensure you have all necessary information.

The first part of the state machine is to populate a 2-dimensional array of register memory. In this register we need to know which sequence we are on, the start address for that sequence, the end address for the sequence and if it is the last sequence in the ROM.

Once that register array is filled the state machine moves into a user/operator mode. In this mode the machine waits for the user to hit a switch or button to increment or decrement the sequence to be displayed on the board.

The state machine will send signals to the driver module through the prewritten outputs in the function_table module. You will need to go through the code and understand what is happening to be able to design this portion of the lab. Please do not skip this step. Please continue to the diagrams for further explanation.

You will need to use ModelSim to debug your state machine and ensure it is functioning properly. You will also need to write your own test benches as each design is too different to give you one.

You should be able to use the 2 push_buttons (key0 and key1) to speed up or slow down the rate at which the LEDs are changed. This is already done for you. There are 3 switched being used on the DE10-Lite for this project. Switch0 and switch1 are up/dwon for the sequence the FPGA is showing respectively. The switch in postion 2 is a system reset.

IT IS REQUIRED TO HAVE THE RESET SWITCH IN THE HIGH POSITION FOR AT LEAST 3 SECONDS OR THE PROGRAM WILL NOT WORK WHEN YOU RUN IT ON THE BOARD!!! This is because registers do not have an initialized vaule and the init flag will never be set to allow your state machine to begin.

Here are some helpful hints:

You should always start by drawing a picture. Please see the below diagrams to help you further understand the problem you are presented. Feel free to design this in any way that works. We would love to see a demo of your project and the state machine solution you created.

You can create a new template of a Mealy or a Moore state machine by:

- Go to File/New/System Verilog HDL File
  - Click OK
- Go to File/Save As
  - Give it a name (i.e. “function_table”).

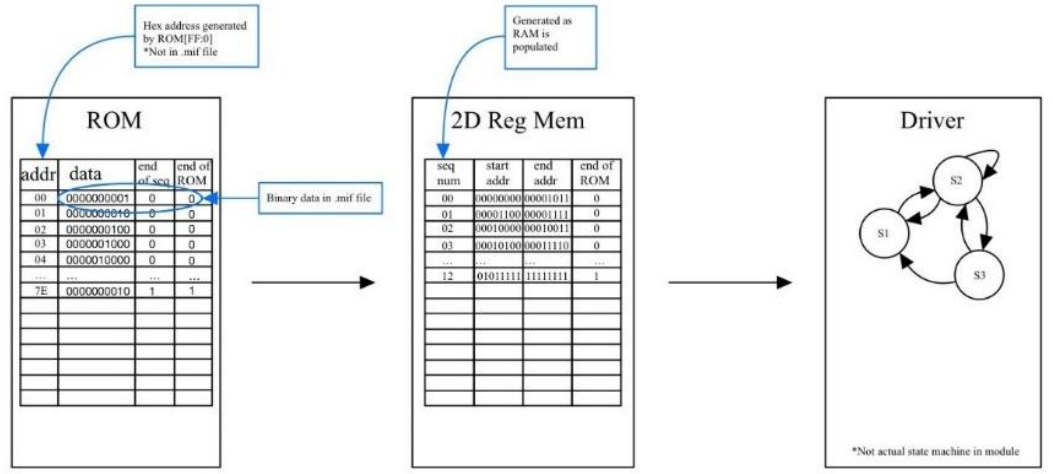
    - Choose an appropriate location, usually the top level of your project folder, or createt a folder to store source code in called "src" and choose this location.
    - Click Save once you are in the directory you would like to save the document.
- Right click anywhere in the document window.
    - Choose "Inset Template…"
    - A window will pop up and select the greater than symbol next to "System Verilog"
    - Click on the greater than symbol next to "Constructs".
    - Click on the greater than symbol next to "State Machines".
    - Select "State Machine With Enumerated Types".
    - Click Insert and then Close and it will insert a template for a 4-state state machine into your System Verilog file. You will need to heavily modify this to get it to work and may need more or less states. This will just help you to get started if you choose to use it. Feel free to make your own too.

If you are not familiar with Mealy and/or Moore state machines, please watch this informative video.

If you are using a DE10-Lite, please set it to allow memory initialization with a file as in the Knight Rider ROM lab or you will get no data from the ROM.

Make sure to debug this program. We have included a test bench that we used for the solution. If you do not use ModelSim, you will have an impossible time debugging. Reach out for help from someone with these skills if you get stuck. You will get stuck. It is ok.

Here is a diagram with an explanation how the design works.



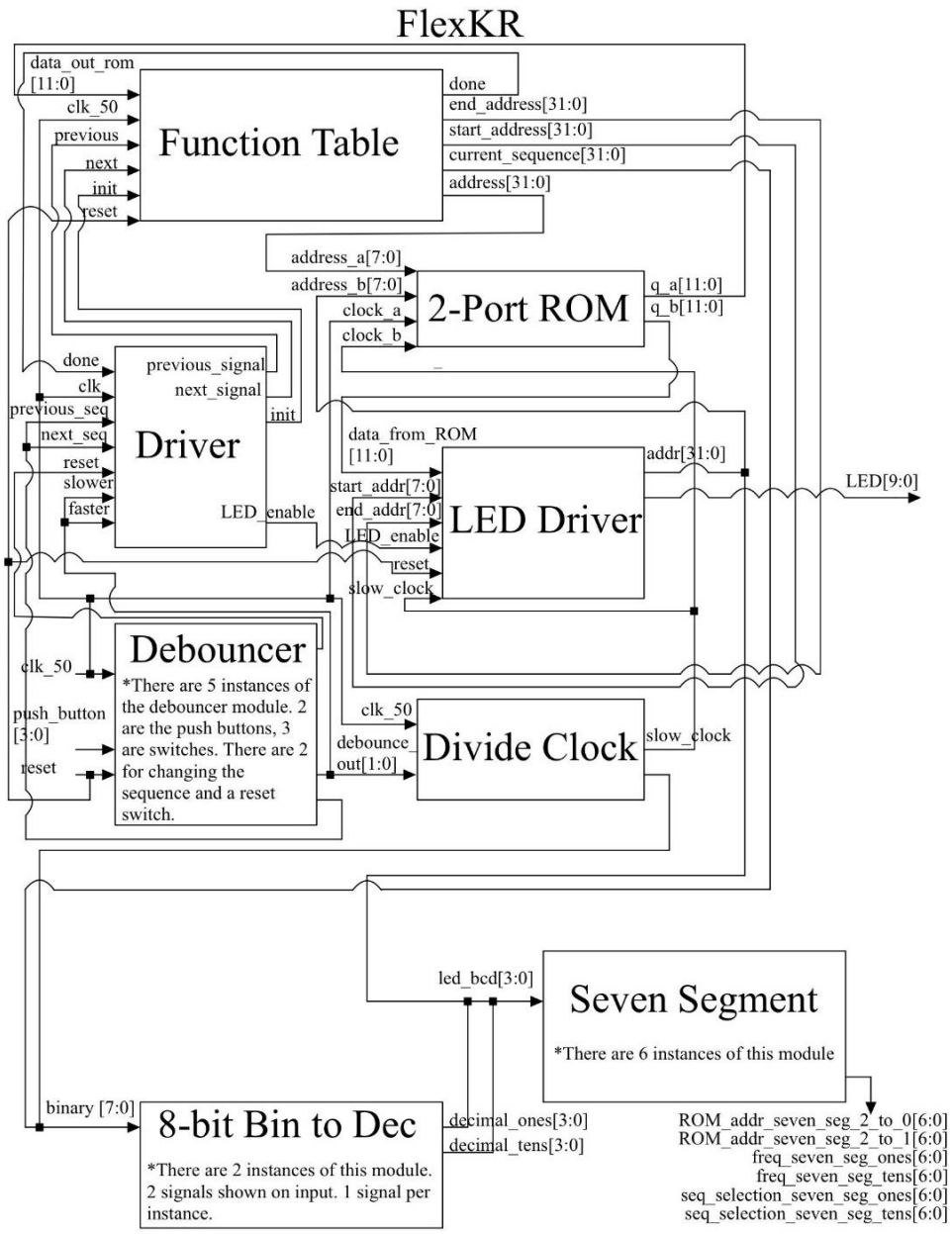
The ROM stores the 12-bits of information containing the LED instance for that address in a sequence along with a bit for end of the current sequence and end of ROM. ROM address starts at 0x00.

This is a RAM for all intents and purposes. This register array will populate each address as the function table state machine reads each sequence data to see if it is the first or last address in that sequence. It will also set a flag to signal last sequence reached.

Once the RAM is populated, the Driver module will be sent a done signal to begin driving the rest of the functions of the circuit.

Once the function table is complete, synthesize your design and use the Programmer in Quartus as you have in previous labs. You should see the first sequence of your ROM being displayed across the LEDs and the HEX displays should show the memory address, the sequence, and the frequency.

3.0 Top Level Schematic



4.0 Document Revision History

Date	Author	Comments
3/23/2022	Kevin Drake	Original release for beta testing.
5/10/2022	Kevin Drake	Updated lab instructions for clarity. Removed System Verilog requirement.