THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE
Department of Electrical and Computer Engineering

CPE 324 Advanced Logic Design Laboratory

Laboratory Assignment #6
Memories

(12% of Final Grade)

Purpose

The purpose of this laboratory is to give each student the opportunity to further expand his/her knowledge of synchronous circuit design, adding the use of memory and data structures on top of the experience from previous labs. Memories provide important features for digital circuits; they can store packets for communication systems, they can store data samples for test or sensor systems, they can store instructions and data for CPUs, they can store keys and values for high speed data searching applications, etc. In this lab the student will perform simulation and FPGA-based synthesis and verification of two different memory structures. The first structure is a read-only memory (ROM), which will be loaded with the first 36 values of the Fibonacci sequence. The second structure is a first-in/first-out buffer using random access memory (RAM). These structures will be tied together and produce an interactive sequence on the DE10-Lite or DE2-115 board.

Design Problem

This lab will be conducted initially via simulation, and then ported to the DE2-115 or DE10-Lite board for demonstration. The lab will be broken up into 3 phases, with the first phase intended to generate a ROM-based table and verify its functionality in simulation. The second phase will add a RAM-based FIFO exercise to pass the ROM sequence data from one clock domain to another, and will be developed and debugged in simulation. The third phase will be to port this onto the DE2 or DE10 board to demonstrate its function. The demonstration will incorporate button presses to read the next data from the FIFO, and the 7-segment LED display will show the current read data value from the FIFO.

Background

Memories are indispensable building blocks for digital circuits, and can be categorized by a number of uses, features, and characteristics. For example, a dynamic RAM (DRAM) is an enormous array of MOS capacitors, each of which stores a bit as a voltage level (VCC means 1, GND means 0). These caps need to be refreshed periodically to maintain the stored voltage level, but are otherwise very fast, small, and inexpensive, so they are used in virtually every modern computer application. Static RAM (SDRAM) uses a DQ Flip Flop to store every bit, which occupies perhaps 20x-50x as much die area of a MOS capacitor, but reliably maintains state without refresh and can be built inside of circuits that contain general CMOS combo/sequential logic. In this lab, we will focus entirely on memories that rely on flip-flops, though we will explore the use of certain FPGA building blocks where possible.

A RAM allows us to randomly access its stored contents in any order (hence Random). An example of a data structure that doesn't allow random access is a shift register. A ROM in an FPGA provides a data structure that is known and constant at compile-time, nevertheless it can perform useful logical functions. Take for example error correction codes (https://en.wikipedia.org/wiki/BCH_code); in certain communication protocols, a transmission header is sent with 51 bits of data, 12 bits of BCH error correction parity, and 1 additional bit for XOR parity. When decoding, the 51 data bits of BCH code go through a specific set of XOR gates to see whether they match the 12-bit code, and any mismatches between the 12 parity bits that were computed and the 12 parity bits that were received will perfectly identify any cases of 1 or 2 bit errors received. There are (63 choose 2) + (63 choose 1) + (63 choose 0) different values among those 12 bits that will be used to indicate where the (2, 1, or 0) errors are, if any. That means there are 2,017 special values among the 4,096 possible that are considered correctable, and it can all be pre-computed and loaded into a 4,096 deep x 12-bit wide ROM that reads out the index of the 0, 1, or 2 bits that need to be inverted (corrected) to get the transmitted message.

Memory Building Blocks

As explained in the CPE 322 lectures, FPGAs typically include a number of embedded memory blocks in addition to normal flip-flops that exist all around the repeated grid of configurable logic blocks. These dedicated embedded memories range from a few hundred bits, to tens of thousands, to hundreds of thousands of bits. The larger RAMs, typically referred to as "Block RAM" are usually provided as true dual-port memory to capture as many use-cases as possible. A dual-port memory can simultaneously read or write from two independent and asynchronous ports, with the caveat that accessing an address location on one port while the other port is writing to the same address will produce uncertain results. Both ports can freely read the same address simultaneously though. A schematic symbol for a true dual-port RAM is shown in Figure 1.

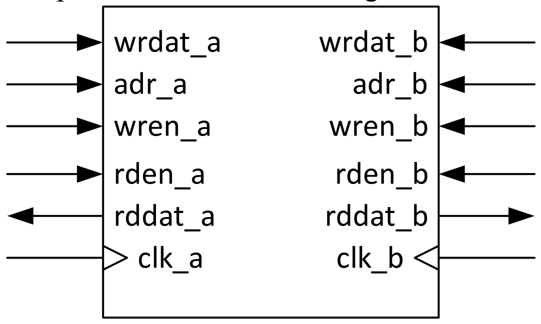

Figure 1: True Dual Port Block RAM Schematic Symbol

Other notable behaviors/features of Block RAMs are as follows:

- All writes are synchronous; set wren_* high to write wrdat_* to the address specified by adr_* (where * represents either port a or b).
- All reads are synchronous and have a 1-cycle latency; set adr_* to the desired address and set rden_* high, and on the following clock cycle the rddat_* output will contain the data from the specified address (again, * is port a or b)
- Either port a or port b can be used for read & write, or can tie wren_* to 0 to be read-only, or can tie rden_* to 0 to be write-only. A FIFO is a good example of a useful configuration with one write-only port and a separate read-only port.
- The Block RAM contents can be written to any specified value during configuration as part of the configuration bit stream loading sequence
- The Block RAM contents cannot be cleared by a single reset; to clear out the contents (or to set them all to specific values), an FSM must run an initialization sequence to write each address individually
- Note: for a FIFO memory buffer, typically the initial contents are don't care, because the FIFO is initially empty, and won't be read until the first data is written

FIFO Micro-architecture

The block diagram of the FIFO the student must implement is shown in Figure 2 below. At its center is the dual-port Block RAM, with port A tied off as a write-only port on the WrClk clock domain, and port B tied off as a read-only port on the RdClk clock domain. The write data and read data connect directly to the RAM, as do the write-enable and read-enable pins (note that the read-enable to read-data latency is a cycle due to the Block RAM itself). The write address and read address of the RAM are each connected to a binary up-counter with enable, which is to say that each counter increments by 1 when the write enable / read enable is high, otherwise it keeps its current value.

A few important notes about the FIFO's read pointer and write pointer:

- For a FIFO with DEPTH number of entries, one can find the number of bits able to express DEPTH by the Verilog system function $clog2(<value>), which provides the ceil($\log_2$(<value>)) function
- If DEPTH is a power-of-2 number (e.g. 4, 8, 16, 32, etc.) the full indication needs an extra pointer bit to distinguish between when 0 entries have been written compared to when DEPTH entries have been written. For example, if DEPTH=8, then the read and write pointers need to be 4 bits so that the write pointer (next address to write) can be 8 addresses greater than the read pointer (next address to read) without the values equaling each other. In the case, the pointers are 4 bits wide but the RAM address only needs to be 3 bits wide (8 unique entries)
- The write pointer is sent to the read clock domain to inform the read side when data is available in the FIFO; when the pointers are equal the read side is told the FIFO is empty (else it has data).
- The read pointer is sent to the write clock domain to inform the write side when the FIFO is too full to allow further writes; when the pointer difference is equal to the FIFO DEPTH parameter, the FIFO is full and cannot accept another write (else it can accept another write enable).
- The Gray Coder logic converts the binary pointer from one clock domain to the other. The Gray Coded pointer only has one bit value changing per clock cycle. The Gray Coded value is first clocked by a flip-flop in the source domain. The Verilog for this function is provided as follows
- A pair of synchronizing flip-flops brings the Gray Coded pointer into the destination domain. This may be handled in the meta.v modules from previous labs.
- The Gray Decoder logic converts the Gray Coded pointer back to binary for the destination domain so it can be compared to the domain's own pointer to produce the Empty (read-side) or Full (write-side) status.
- The FIFO implements failsafe options to disallow writes when Full and to disallow reads when Empty. The WrEn input is AND-ed with a not-Full status, and the RdEn input is AND-ed with a not-Empty status to invoke a change to the pointer states.

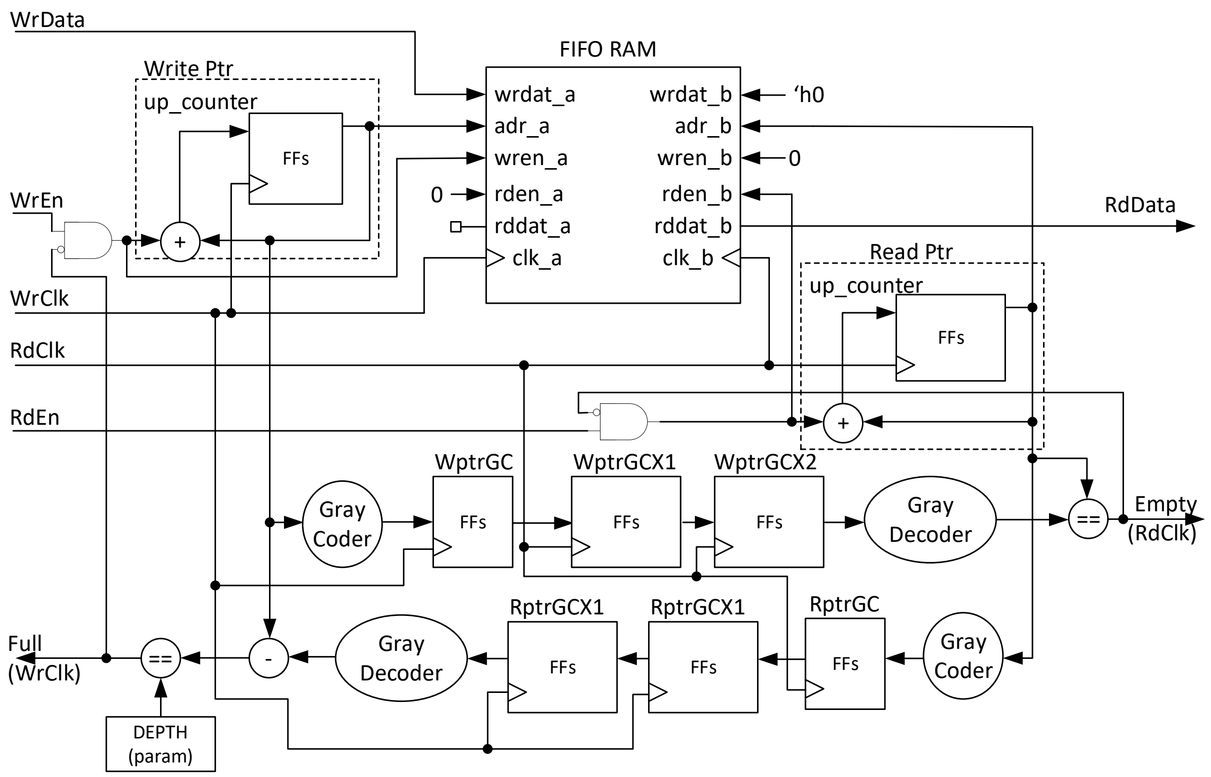

Figure 2: FIFO Block Diagram

Top Level Design

The top-level for Phase 3 of this lab must be adapted from the student's prior Lab 4 top-level file (or simply from the original top-level provided for Lab 4). This includes the familiar display_driver.v module, as well as meta.v modules for synchronizing the switches and buttons. A PLL is incorporated to generate a 66.667 MHz clock so that the write and read side of the FIFO operate asynchronously with respect to each other. The student must generate this PLL (or re-customize the syspll from Lab 4) to generate 66.667 MHz. The KEY[0] button is connected to the PLL areset input, and the PLL's "locked" status output is inverted, synchronized, and used as a reset for the 50 MHz clock and 66.667 MHz clock domains.

The Top-level design must also include the ROM and FIFO, with the FIFO's read-enable connected to the KEY[1] button and reset connected to the KEY[0] button. The top-level module also creates a ROM address counter (using the up_counter module) outside of the FIFO that continually increments and reads out of the ROM until the FIFO is full. Upon reset, the ROM's read counter will reset to 0 and the FIFO will set its write and read pointers to 0. After reset, the ROM's contents are read out continuously until the FIFO is full. The FIFO will read out the contents at the current read pointer, which will be displayed as a hexadecimal value on the 7-segment LED display. Upon a press of KEY[1], the read-enable signal to the FIFO will get a single high pulse and move on to the next FIFO entry.

A pair of LEDs (LEDR[1:0]) display debug status for the FIFO empty and PLL locked signals.

TOP LEVEL

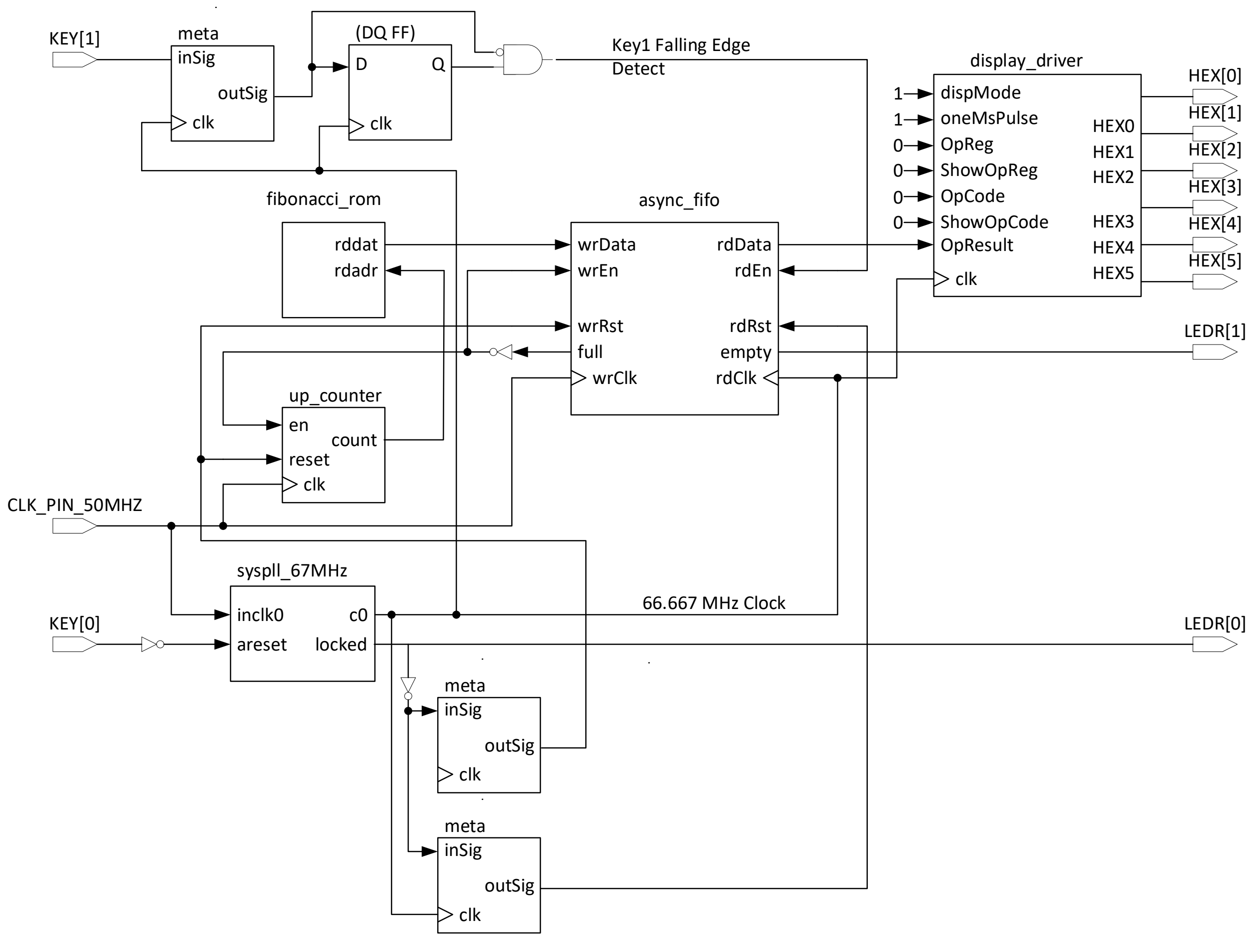

Figure 3: Top-Level Design Block Diagram

Lab Instructions

Phase 1

1. Begin by opening the file named fibonacci_rom.v
2. Within the fibonacci_rom.v file, there is an existing declaration for a 24-bit wide, 36-deep ROM, with 0 clock cycles of delay from the rdadr[5:0] input to the rddata[23:0] output. However, the student will recognize that the ROM is not initialized with contents. Using a for loop within the "initial" procedural block, set the contents of the ROM's 36 addresses (from 0 to 35) to the first 36 values in the Fibonacci sequence. The sequence is defined as follows; with the first 2 entries of the sequence both set to 1, for every subsequent entry Fibonacci_seq[i] = Fibonacci_seq[i-1] + Fibonacci_seq[i-2].
3. Once the ROM initialization routine is established, create a brand-new Verilog module named up_counter.v with a parameter called WIDTH that sets the width of the counter, and a parameter TERM_CNT that sets the value at which the counter automatically wraps to 0. The up_counter.v module will have as inputs **clk**, **reset**, and **en**, all of which are 1 bit wide, and will have a single output signal called **count**[WIDTH-1:0]. When the synchronous **reset** signal is high on a rising edge of clk, the value of the **count** output will be set to 0. When **reset** is low, for every cycle the **en** input is high, the count output will increment by 1 upon

the subsequent rising edge, rolling over at 2**(WIDTH)-1 to 0. The **count** signal can be declared as an "output reg" so that it can be set within the main always @(posedge clk) procedural block, or it may be declared as simply "output" and assigned from an internal reg.

4. Once the up_counter and ROM are complete, the student will compile the tb_fibonacci_rom.v testbench file along with fibonacci_rom.v and up_counter.v within Modelsim by creating a work library, and running the "vlog …v" command. Upon successful compilation, the student will run "vsim tb_fibonacci_rom" and add all the signals from tb_fibonacci_rom and its subordinate modules to the Wave window. Once the waveform window shows all pertinent signals, the student will run the simulation for 1 usec (i.e. "run 1 us"). A screenshot of the simulation showing various simulated values of the ROM will indicate successful completion of Phase 1.

Phase 2

1. The design of the FIFO commences with a provided Verilog model of a simple dual-port memory, which aides the Quartus tool's inference of a Block RAM. By "inference", it is meant that the FPGA synthesis tool will identify a RAM as coded in Verilog and map it directly to dedicated memory resources instead of FFs and LUTs. The sdp_ram_infer.v module provides this code – please open and read it, noticing that it roughly matches the example covered in the CPE 322 lecture. (Note that true dual-port asynchronous Block RAMs are difficult to describe via Verilog in a way that the FPGA synthesis tools will accurately map them to a Block RAM component, so this lab will avoid that headache).
2. Using Figure 2 to show the FIFO's microarchitecture, and using the sdp_ram_infer, up_counter, gray_coder, and gray_decoder modules, the student must design and code the async_fifo.v module. The async_fifo.v module I/O shell and parameters are provided to the student, with the signals defined/described as listed above, and with parameters WIDTH to determine the data width of each FIFO entry, and DEPTH to determine the number of entries in the FIFO. **NOTE:** wrRst and rdRst are not drawn in the block diagram, but are necessary to add to reset the write pointer and read pointer, respectively.
3. Using Modelsim and the provided tb_async_fifo.v module, the student must simulate his/her implementation of the async_fifo.v module, debugging until it is determined that expected behavior is achieved. The testbench performs a simple function to write until full, then read until empty, toggling back-and-forth between those two phases of operation. The testbench does not include the Fibonacci ROM, so the data being transferred through the FIFO is a simple incrementing pattern.

Phase 3

1. The FIFO is to be integrated into the provided top-level file, with integrated display_driver connected to the FIFO read data. KEY[0] shall be connected to the PLL's arst line, the KEY[1] pushbutton is connected through a pulse generator to connect to the read-enable of the FIFO. See the provided block diagram for a complete list of required modules and the interconnect between them.
2. The PLL must be customized by the student to generate a 66.667 MHz clock from the 50 MHz on-board oscillator. The student must report on the PLL configuration in the post-lab questions. To access the PLL configuration, please open the syspll component in the project hierarchy to view its configuration.
3. The student will build the FPGA, verify that timing is met at 50 MHz and 66.667 MHz. The student must also enable SignalTap on either the 50 MHz clock (to debug the FIFO write side) or on the 66.667 MHz clock (to debug the FIFO read side).
    a. If everything works without a hitch, placing debug signals on the FIFO write side is recommended, which should include the ROM read address (from up_counter), the ROM read data, the FIFO full indication, FIFO wren signal, the FIFO's internal wrPtr and rdPtrGCX2 signals. The selected clock for sampling should be the 50 MHz

oscillator pin (wrClk).

   b. If the student needs to debug the read-side to achieve proper functionality, please add the FIFO rdEn, rdPtr, rdData, Empty, wrPtrGCX2 signals among any others that are useful for debug. The selected clock for sampling must be the 66.667 MHz PLL output (c0 PLL pin, rdClk).

4. Upon successful completion of a build, the student must record the (at most 5-minutes) demonstration video, showing button press behavior along with the 7-segment hex display. Remark about the design of the FIFO component from the block diagram and its basic building blocks.
5. Two common mistakes that can eliminate proper inference of Block RAM are to encode it with zero latency on the read-side, or to reset the contents. In this phase, the student will modify the sdp_ram_infer.v to include a fan-out of a reset input signal to set the RAM contents to 0 in a single cycle, and report the impact of coding it this way. Then the student will eliminate the 1-cycle latency from rdAdr to rdData and report on the impact of coding it this way.
   a. For the reset function, use the rdRst input and place the condition in the always @(posedge wrClk) procedural block, and use a for-loop to increment from addresses 0 to DEPTH-1, setting each entry to 0.
   b. For the 0-cycle read latency, comment out the always @(posedge rdClk) procedural block and assign the output value of rdDat directly from rom[rdAdr] via assign statement.
6. In case a design needs to initialize a RAM's contents upon reset, it is a good practice to create an "init" state to increment through the entire address range of the memory

Post Lab Questions:

1. Run through the entire 36 entries of the Fibonacci sequence stored in the ROM and report the numbers on the lab report. What is the next value of the sequence? Will it fit in a 24-bit binary number space of the 6-digit hex display?
2. For generating the 66.667 MHz clock for the read-side of the FIFO:
   a. What frequency does the VCO oscillate at?
   b. What is the clock division ratio on the 50 MHz input clock?
   c. What is the divider for the VCO feedback path?
   d. The answers to 1b and 1c above must lead to the same frequency for the PLL to lock – do they? What frequency is it?
   e. What is the division ratio from the VCO to the 66.667 MHz output clock?
3. The top-level module has two meta.v blocks drawn at the ~locked output of the system PLL. They are connected to the same inSig, so why are there two different ones? Where do the outputs of these blocks lead?
4. What is the reported device utilization with the original sdp_ram_infer.v block? This must include the number of logic elements, flip-flops, and dedicated RAM blocks and/or bits.
   a. In Phase 3, after recording the device utilization numbers, the student is asked to modify the sdp_ram_infer.v file to include a reset to set the RAM contents to 0 for every address. What effect does this have on device utilization numbers? Does it have an adverse impact on meeting timing?
   b. The other modification to sdp_ram_infer (besides resetting all contents) is to remove the 1-cycle latency between the RAM data array storage and the read-data output. Again in this scenario, report and remark on the ability for Quartus to map the RAM into the large, efficient Block RAMs, and whether device utilization and timing were negatively impacted.
5. Try the following depths for the async_fifo block, and report on device utilization numbers. For the case where the depth of the FIFO is not a power-of-two, simulate it to ensure it works properly and then demonstrate that it performs the same as any other (power-of-two) option.

a. 32
b. 128
c. 10
d. 512
e. 2048