Embedded Memories and an External SRAM Interface

Objective

To understand how to build and simulate a digital system containing embedded memory blocks, such as read-only memory (ROM), single-port random access memory (RAM), and dual-port RAM. To gain experience with design verification using SystemVerilog testbenches and the ModelSim simulation environment. To design and implement a self-test engine for an external static RAM (SRAM) memory.

Preparation

- Revise the first three labs
- Get familiarized with the source code and the in-lab experiments for this lab

In-lab experiments

- Experiment 1
- Experiment 2
- Experiment 3
- Experiment 4

Evaluation

Take-home exercises (due 16 hours before the next lab is scheduled):

- Exercise 1 (has a weight of 2% of your final grade)
- Exercise 2 (has a weight of 2% of your final grade)

Experiment 1

The objective of this experiment is to get you familiarized with SystemVerilog for design verification.

Hardware description languages (HDLs), e.g., VHDL, Verilog, SystemVerilog, are employed for both describing hardware and verifying it. As shown in Figure 1, the aim of verification is to control the inputs of a design (using *drivers*) and to observe its internal states and outputs (using *monitors*) in order to ensure the correct behavior before the design is implemented. There are (System)Verilog language constructs which (as shown in the testbench for **experiment 1**) are used **exclusively** for verification. Note, most of these language constructs do not have an equivalent hardware circuit, as they are used primarily to drive and monitor the design behavior through simulation. They are used in special verification modules called “testbenches”. A properly designed testbench will ensure that the design will work correctly as soon as it is implemented, e.g., programmed into a field-programmable gate array (FPGA) device.

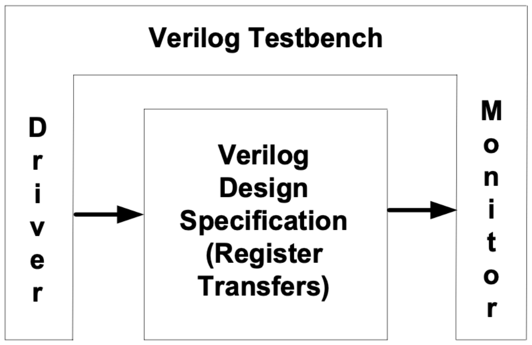


Figure 1 – Design verification using testbenches

Verifying the design through simulation can be done at different levels of design abstraction. The (System)Verilog source code can be simulated “fast” by using timing control and sensitivity lists (part of the language features for verification). This is called behavioral simulation and it is commonly at least one order of magnitude faster in terms of runtime than timing simulation. Timing simulation works on the implemented netlist and it uses accurate timing information, which is back-annotated from the technology library. Most of the iterative debugging is done by updating the register-transfer level (RTL) source code and running behavioral simulations. Only when the confidence is high about the design correctness, one should perform timing simulation and/or download the compiled bitstream to the FPGA.

Most of the essential language constructs for simulation will be learnt incrementally, as you progress through the labs and the project. It is worth mentioning that, unlike for the synthesizable RTL code, for driving purposes, in a testbench file one can update the same input in one or multiple `initial` and/or `always` blocks (nonetheless this should be done in a judicious manner to avoid race conditions). The timing of an update, can be controlled using, for example, `#`, which depending on its usage can delay the next evaluation and/or update by a user-specified time. For monitoring purposes, both in the `initial` and `always` blocks you can use the built-in *$write* or *$display* tasks that have similar behavior to *printf* in C/C++ (note, a new line is implied for *$display*). Custom tasks can also be defined and included in `initial` and `always` blocks. The timing can also be controlled either by checking if a condition is true and, if not, wait until it occurs (using `wait`) or by waiting for an event, e.g., a signal transition, to occur (using `@`).

The reference design from this experiment implements the same functionality as the reference design from **experiment 2** from **lab 1**, i.e., it finds the index of the most significant switch that is *on* (from switches 9 down to 0). You have to perform the following tasks in the lab for this experiment:

- expand the testbench (from the **tb** subfolder) to confirm the correctness of the design, by writing the code for the monitors for green LEDs and seven-segment displays, as well as by writing the code to emulate additional toggles on input switches
- change the design (from the **rtl** subfolder) to find also the least significant switch that is *off* (from switches 9 down to 0) and display its index on the leftmost (most significant) seven-segment display, and (in binary) on the green LEDs from position 7 down to 4; note, if all the switches are *on* then assume that the least significant switch that is *off* has index `4'hF`; verify that your design modifications are correct by updating the testbench (**note**: you should **not** use waves for this experiment)

Experiment 2

The aim of this experiment is to understand the functionality of embedded RAM blocks.

Programmable logic devices contain embedded memory blocks that can be configured as ROMs or single-port RAMs or dual-port RAMs (or DP-RAMs). They are necessary for avoiding storing large amounts of state information in on-chip registers. They are suitable for implementing the first level of caches in memory systems. The difference between having a RAM on-chip vs. off-chip lies in the fast memory access time on-chip. Furthermore, having multiple memory blocks facilitates concurrent access of data in all the embedded memories, thus avoiding accessing a shared resource off-chip (this avoids memory access bottlenecks that can slow down the performance of computer systems).

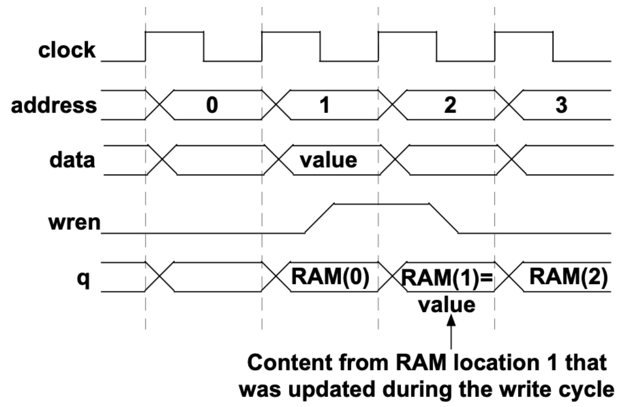

clock
address
0
1
2
3
data
value
wren
q
RAM(0)
RAM(1)= value
RAM(2)
Content from RAM location 1 that was updated during the write cycle

Figure 2 – The memory access waveforms for single-port RAMs

As shown in Figure 2, for single-port RAMs the write enable signal (*wren*) decides whether the memory cycle is a READ or a WRITE. For the given RAM configuration, after the address is applied to the inputs, the output data will be available after the edge of the clock cycle. If *wren* is high then the data available on the input port will be written to the memory location determined by the address bits on the next edge of the clock cycle.

To better understand the benefits of embedded RAMs, this experiment and the next one show how values from two arrays of 512 elements, and 8 bits per element, can be read, added and subtracted, and subsequently these results stored back to the embedded RAMs. In Figure 3 an implementation with 2 single-port RAMs is given. The single-port RAMs share the same address register and write enable. The data is first read from a location determined by the address register and in the following clock cycle the address register is kept the same while the addition and the subtraction results are written back to the embedded RAMs by activating the write enable signal.

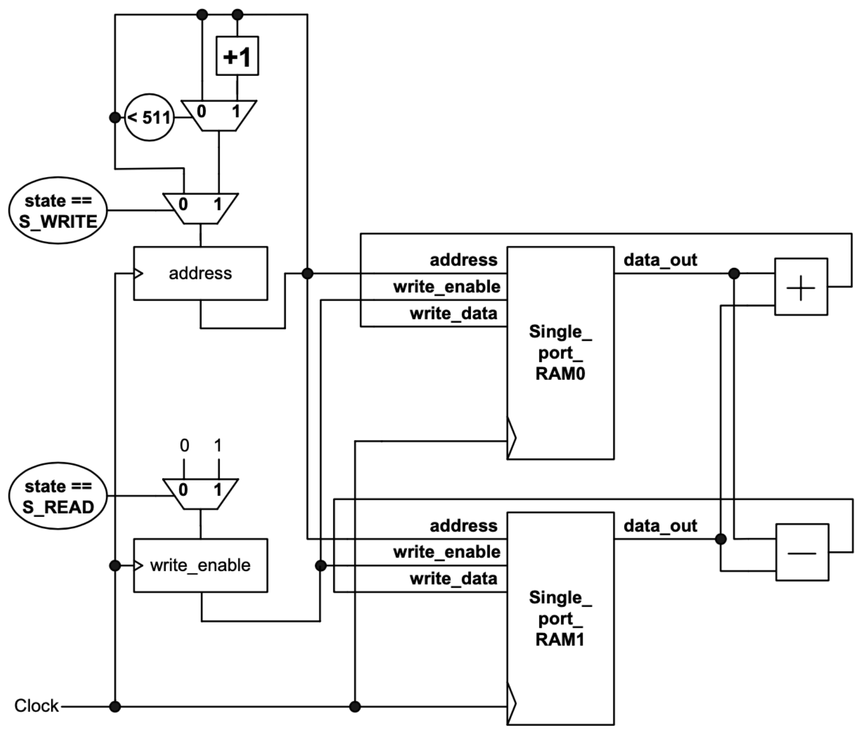


Figure 3 – Circuit from **experiment 2** implemented using 2 single-port RAMs

You have to perform the following tasks in the lab for this experiment:

- verify the correctness of the design through simulation; note, the 8-bit values that are read from the embedded memories have been initialized using MIF files available in the rtl subfolder; note also, at the end of the simulation, the content of the embedded memories will be dumped in .mem files in the sim subfolder; it is also worth mentioning that in the case of arithmetic overflows, no correction is applied, i.e., the 8 least significant bits of the result are stored in the memory

Experiment 3

The aim of this experiment is to help you understand the key benefits of dual-port RAMs (DP-RAMs).

The basic DP-RAMs support reading on both ports or reading on one port and writing on the other port. For true DP-RAMs, data can be written on both ports. Note however if a write to the same location is attempted on both ports at the same clock cycle a hazard will occur, which may push the system into meta-stability (i.e., the value from the respective address will be undefined).

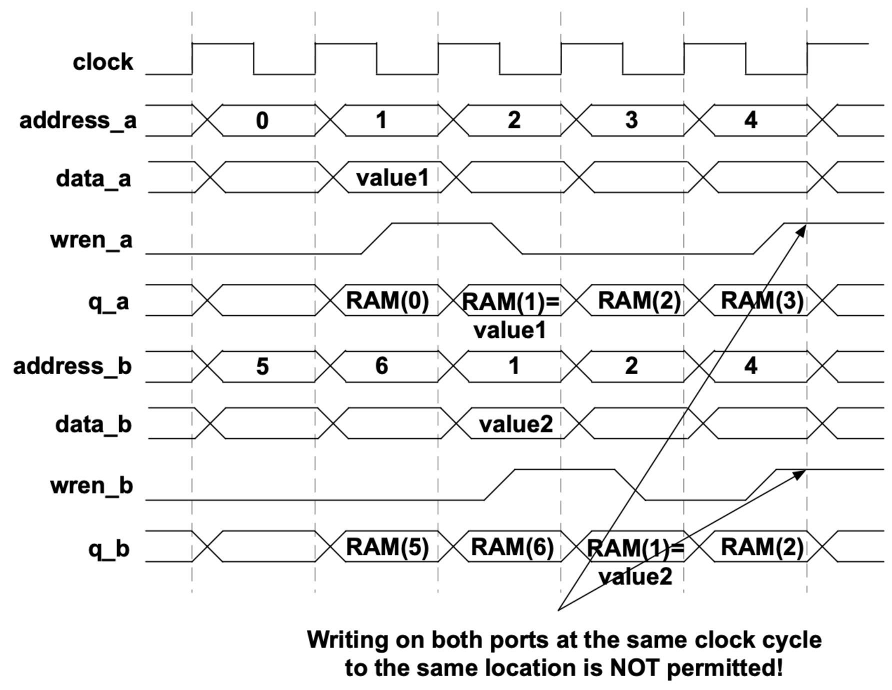


Figure 4 – The memory access waveforms for dual-port RAMs

This computation for the array calculations from **experiment 2** can be accelerated by using two dual-port RAMs of the same capacity. There will be two distinct address registers (one for reading and one for writing). Unlike the case where the single-port RAM is employed, the read address register is incremented every clock cycle. This register will be hooked up to a read port while the write address register (delayed by one clock cycle) is connected to a write port. Because (by design construction) the read address and write address registers will not conflict, a new result can be written back to the memory every single clock cycle.

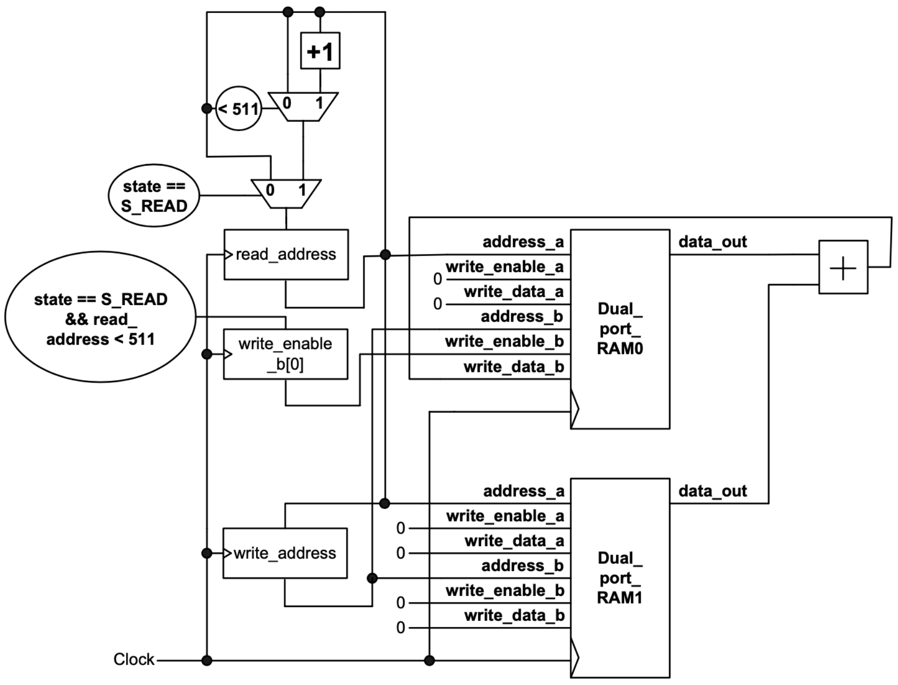


Figure 5 – Circuit with the same behavior as the circuit from **experiment 2** implemented using 2 dual-port RAMs (to be completed)

You have to perform the following tasks in the lab for this experiment:

- in the code from folder **experiment 3**, the addition of the values read from the two RAMs is stored in the top RAM; extend the dual-port implementation (shown above) in such way that the subtraction of the values read from the two RAMs is stored in the bottom RAM
- understand the speed-up facilitated by dual-port RAMs over single-port RAMs

Experiment 4

The objective of this experiment is to introduce the external SRAM and to show how it can be used and modeled.

First, it is important to note that although the capacity of the SRAM device on the DE2-115 board is 2 MB (accessed by 20 address lines and 16 data lines), for the rest of this course we will use the external SRAM in the following mode: 18 address lines and 16 data lines (organization of 256k locations with 16 bits per location). This is because the source code was originally developed for a previous version of the DE2 board … and, not surprisingly, there were a total of 18 address lines for the respective SRAM device. Nonetheless, it is worth noting that working with less memory brings hurdles that are common for many form-factor constrained systems and that make some design decisions more intellectually challenging. Furthermore, working with smaller data sets stored in the external memory is indirectly advantageous in terms of clock cycles that need to be simulated, without avoiding the more challenging verification challenges that provide the basis for an improved learning experience.

The SRAM controller implemented for the external SRAM device has the memory access cycles shown in Figure 6. Note, although the latency for both reading and writing is 2 clock cycles, it does have a throughput of one result per clock cycle, i.e., in any clock cycle we can initiate a read or a write that completes two clock cycles later. Note also, the write enable for the external SRAM is **active low**.

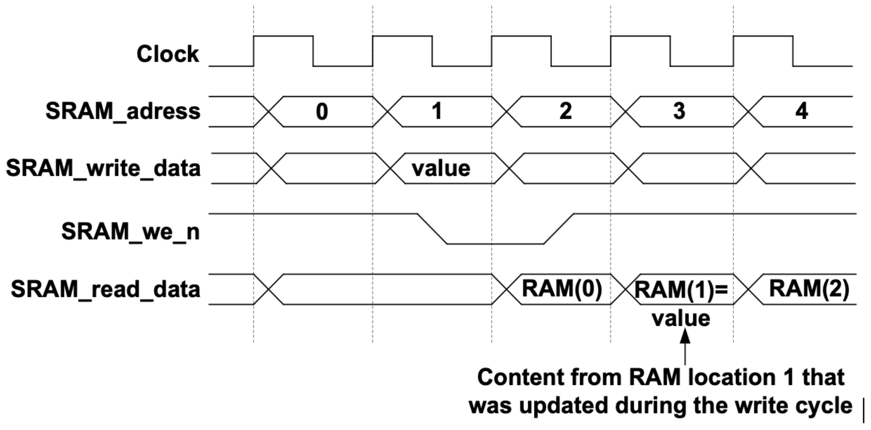
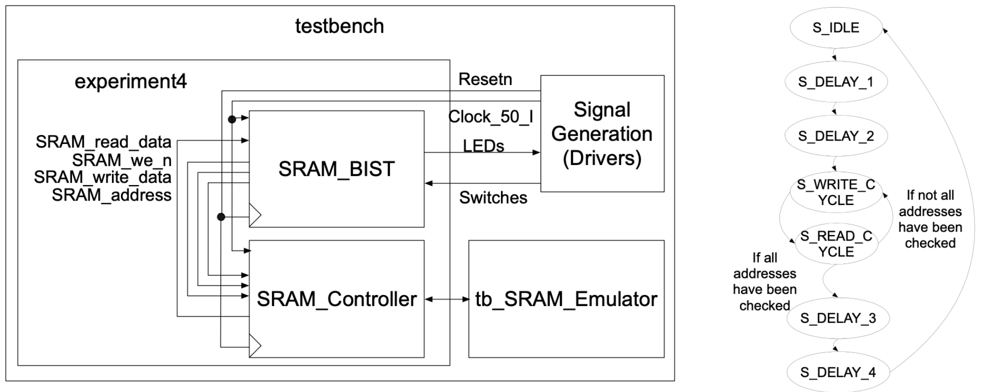


Figure 6 – Memory access waveforms to the external SRAM controller

In the **experiment 4** folder, a simple built-in self-test (BIST) engine is designed for the SRAM device. BIST is an enabling technology that is essential for characterization, test and diagnosis of high-density memory devices. The circuit architecture and its testbench are shown in Figure 7(a), while the state machine for a reference BIST engine is given in Figure 7(b). The BIST engine employs a counter that iterates through the entire memory space of the SRAM and it reads back the value that has just been written. In the case of a mismatch an error flag will be set. The key point of this experiment is the use of an SRAM emulator for simulation purposes. Because the SRAM is an external reactive device (i.e., it responds to the requests given by our circuit) it is essential that its behavior is accurately modeled in the verification testbenches.



(a) Design architecture and testbench (b) BIST state machine

Figure 7 – Circuit to support the SRAM BIST engine (including its testbench and state machine)

You have to perform the following tasks in the lab for this experiment:

- understand how an emulator module can be used for simulating an external device, such the external SRAM from this experiment
- in the given source code the memory BIST is done by reading back the value which has just been written in the SRAM; modify the code in such way that all the values are written in the SRAM in a *burst* mode (the data value equals the lowest 16 bits of the address); then all the values from the memory should be read back and compared against the expected values generated on-chip; prove that your design works first through simulation

Exercise 1

Modify the design from **experiment 3** as follows.

In the first and second DP-RAMs, there are two arrays (**W** and **X**, respectively) whose elements are 8-bit signed integers. Each of these arrays has 512 elements (initialized the same way as in the lab, i.e., using memory initialization files). Design the circuit that computes two arrays, **Y** and **Z**, as specified below:

- for the top half of the memory, i.e., *k* is between 0 and 255 (inclusive), **Y**[*k*] and **Z**[*k*] are defined as follows. If **W**[*k*] is negative then **Y**[*k*] = **X**[*k*] - 1 and **Z**[*k*] = **W**[*k*] + 1; otherwise **Y**[*k*] = **W**[*k*] - 1 and **Z**[*k*] = **X**[*k*] + 1;

- for the bottom half of the memory, i.e., *k* is between 256 and 511 (inclusive), **Y**[*k*] and **Z**[*k*] are defined as follows. If **X**[*k*] is positive (assume zero is also a positive number) then **Y**[*k*] = **W**[*k*] + **X**[*k*] and **Z**[*k*] = **W**[*k*] - **X**[*k*]; otherwise **Y**[*k*] = **X**[*k*] - **W**[*k*] and **Z**[*k*] = **X**[*k*] + **W**[*k*];

Each element **Y**[*k*] should overwrite the corresponding element **W**[*k*] in the first DP-RAM (for every *k* from 0 to 511); likewise, each **Z**[*k*] should overwrite the **X**[*k*] in location *k* in the second DP-RAM. As for the in-lab experiment, if the arithmetic overflow occurs as a direct consequence of the additions/subtractions, it is unnecessary to detect it, i.e., keep the 8 least significant bits of the result. The above calculations should be implemented in as few clock cycles as the two DP-RAMs can facilitate.

For this exercise only, in your report, you **MUST** discuss your resource usage in terms of registers. You should relate your estimate to the register count from the compilation report in Quartus. You should also inspect the critical path either in the Timing Analyzer menu, as shown in the videos on circuit implementation and timing from **lab 3**, or by checking the `.sta.rpt` file from the `syn/output_files` sub-folder, which contains the same info as displayed in the Timing Analyzer menu. Based on your specific design structure, you should also provide your best possible interpretation of the critical path in your design.

œ**Note**: A multi-bit signal whose value is to be interpreted by a circuit as a signed number, as is the case for this exercise, can be tested if it is positive or negative in multiple ways, with the following two options suggested.

- check its most significant bit: if it is one, then the number is negative; otherwise, it is positive;

- by default, a multi-bit signal declared using the **logic** type in SystemVerilog is interpreted as an unsigned number; for example, if an 8-bit signal is declared as **logic** *[7:0] my_signal;* and it is assigned value `8'hFF` then condition *my_signal < 0* would evaluate to `FALSE` because `255` is greater than `0`; however, you can use the ***$signed*** system function to interpret the multi-bit signal as a signed number and therefore condition ***$signed****(my_signal) < 0* would evaluate to `TRUE` because `-1` is less than `0`;

Submit your sources, and in your report, write approximately half a page (but not more than a full page) that describes your reasoning. Your sources should follow the directory structure from the in-lab experiments (already set up for you in the `exercise1` folder); note, your report (in `.pdf`, `.txt` or `.md` format) should be included in the `exercise1/doc` sub-folder.

Your submission is due 16 hours before your next lab session. Late submissions will be penalized.

Exercise 2

Modify the built-in self-test (BIST) engine from **experiment 4** as follows. To verify all the 2^{18} (or 256k) locations of the external SRAM, two sessions of writes and reads will be performed: the first one for the 2^{17} (or 128k) **even** locations and the second one for the 128k **odd** locations, as explained below.

In the first session, the 128k **even** locations will be verified by first writing the ***bitwise complement*** of the 16 least significant bits of the address. For example, in location 00000, write value FFFF; in location 00002, write value FFFD; and so on, in the last even location 3FFFE, write value 0001. While writing the data during the first session, the address lines must change in *increasing* order. Then, the same 128k even locations will be read to verify their content. When reading the data during the first session, the address lines must change in the same direction as when writing, i.e., the *increasing* order.

After the first session, in the second session, the BIST engine will start by writing the bitwise complement of the 16 least significant bits to the 128k **odd** locations. Unlike the addressing order for the even locations, when writing data to the odd locations, the address lines must change in *decreasing* order. For example, start from the last location in the memory 3FFFF and write value 0000 into it; in location 3FFFD, write value 0002; and so on, in the first odd location 00001 (but the last to be written), write value FFFE. Then, the 128k odd locations will be read and compared against the expected values to verify their content. When reading during the second session, the address lines must change in the same direction as when writing, i.e., the *decreasing* order.

It is important to note that in each of the two sessions, every location must be written exactly once and checked exactly once.

Submit your sources, and in your report, write approximately half a page (but not more than a full page) that describes your reasoning. Your sources should follow the directory structure from the in-lab experiments (already set up for you in the `exercise2` folder); note, your report (in `.pdf`, `.txt` or `.md` format) should be included in the `exercise2/doc` sub-folder. Note also that although this lab is focused on simulation, your design must still pass compilation in Quartus before you simulate it and write the report.

Your submission is due 16 hours before your next lab session. Late submissions will be penalized.