ECE 2031 Spring 2020 Project: SRAM

Project Overview

Create an I/O peripheral for SCOMP that will enable read/write access to the SRAM chip on the DE2 board. The device will allow future 2031 students to use that memory during their projects.

Device Requirements

- Enable 16-bit read and write access between SCOMP and the DE2 board SRAM.
- Operate as an I/O device with any address(es) 0x10-0x1F.
- Implement as a VHDL device, generating a Quartus symbol for use in the top-level design file

General Comments

Simulation is a crucial part of the design process. Creating a physical prototype can cost hundreds of thousands of dollars, so being confident that it will work before prototyping is desirable. In areas like space flight, the first time some systems can be physically tested involves the possibility for loss of life. Even in the low-stakes environment of this project, we would like to be confident that the design we choose to carry forward to future semesters will work, before we spend time integrating it with the standard project files.

So even though this semester you won't be able to physically test your device, learning how to be confident that your device works through simulation is an important skill that this semester's project will highlight more than in most semesters.

Design Demonstration

At the end of the semester, you will present your results in a video conference with the course instructors. You should

- Present an overview of your device's internal design.
- Explain how a user interfaces with your device via assembly code.
- Show simulations of your device correctly responding to SCOMP INs and OUTs and correctly controlling the SRAM signals.

Provided Project Files

You are given a Quartus project that contains SCOMP, the IO address decoder, an example SRAM interface, and an SRAM emulator that will make simulations easier to use. The top-level design file is greatly simplified from the one that you used in the lab. Hardware devices like switches, LEDs, and the timer, which would be difficult to simulate and of questionable value in producing a good project, have been eliminated. This should make it much easier for you to focus on the devices of importance.

SLOW_SRAM.BDF

There is still an example SRAM interface in the project file, called SLOW_SRAM.BDF. It serves the same purpose as it did when the project started. It is something that you are supposed to delete and replace with an improved I/O device that meets the project requirements (see the three bulleted items under "Device Requirements" above).

Simulation Waveform

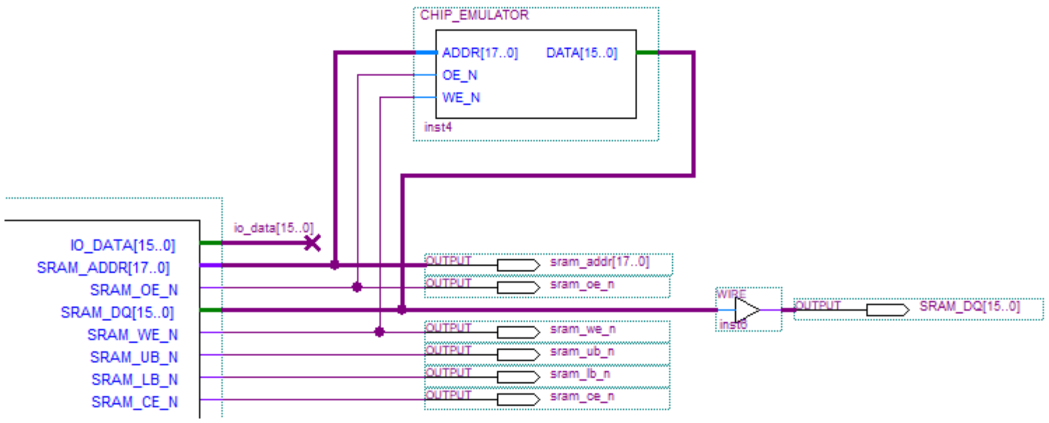
A simulation is provided that contains signals that will be of interest to you. You are free to modify it, but it will be a good starting point.

The simulation file runs for 1 ms. To save yourself some time, if you do not need to simulate for that long, you can limit the simulation duration in the simulator tool.

SRAM Emulator

The SRAM emulator (CHIP_EMULATOR) connects to the signals that would connect to the DE2 board's SRAM chip, and behaves similarly to how the SRAM would behave. Specifically:

- The emulator has 32 memory locations (addresses 0-31) that can be written to using the same sequence of control signals as the actual SRAM chip.
- The emulator will output data under the same conditions as the actual SRAM chip. If accessing addresses 0-31, the emulator will provide the data that has been written to that address. For other addresses, the emulator will provide the one's complement of the high 16 bits of address as the data. You should think of this as having 32 addresses where you can fully test reads and writes, and being able to test reads from the rest of the memory space because you know what is "stored" at those addresses.



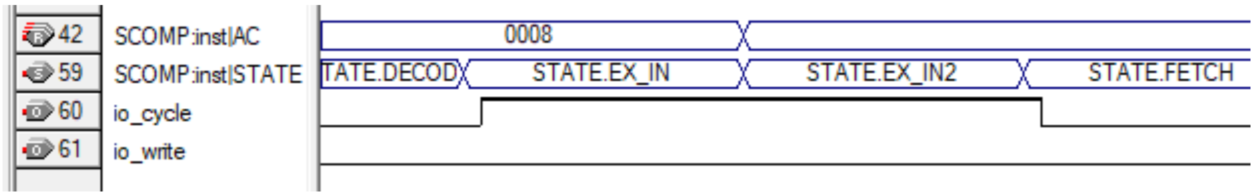
SCOMP I/O Specification

The SCOMP used in the 2031 project has been slightly modified from labs 7/8. Most relevant to you, the I/O behavior has been made more explicit in its timing to allow a greater variety of I/O devices. All of the changes described here are in SCOMP.VHD (not in any of the other project devices), and you should not expend any of your project time changing them. These changes are improvements, for your benefit.

I/O Reads (IN)

The following sequence occurs during an IN instruction. Each step occurs on subsequent SCOMP clocks.

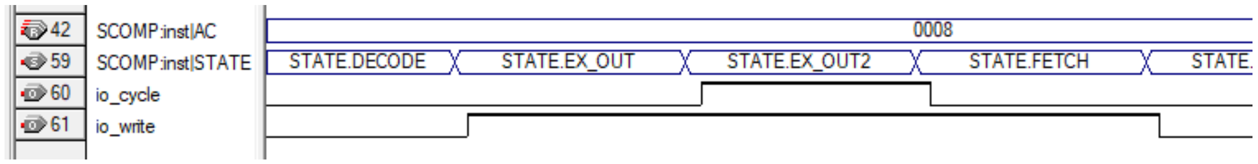
1. SCOMP sets IO_ADDR
2. SCOMP asserts IO_CYCLE
3. SCOMP latches the data on IO_DATA to its AC. The peripheral must be driving IO_DATA by this time.
4. SCOMP deasserts IO_CYCLE



I/O Writes (OUT)

The following sequence occurs during an OUT instruction. Each step occurs on subsequent SCOMP clocks.

1. SCOMP sets IO_ADDR
2. SCOMP asserts IO_WRITE and begins driving IO_DATA
3. SCOMP asserts IO_CYCLE. If the peripheral registers the data from SCOMP, it can latch IO_DATA on the rising edge of IO_CYCLE.
4. SCOMP deasserts IO_CYCLE. Since IO_DATA is driven by SCOMP for the entire time that IO_CYCLE is asserted, it is safe for a peripheral to use IO_CYCLE as a latch enable.
5. SCOMP deasserts IO_WRITE and stops driving IO_DATA.



Note: peripheral devices that respond to both IN and OUT should use NOT(IO_WRITE)·IO_CYCLE to control driving IO_DATA. Since IO_WRITE is asserted before IO_CYCLE during an OUT, that will ensure that the peripheral never drives the bus during an OUT.