EE 371 Project 3

Building and Working with a Simple Microprocessor….

University of Washington - Department of Electrical Engineering

James K. Peckol

Introduction:

In this third project, we will work with some new tools and concepts as we design and incorporate a NIOS II microprocessor onto our gate array as the controller of our habitat system. Building on the core hardware, we will add support for general purpose I/O to enable information exchange between the processor and functionality both within and external to the FPGA. Our first piece of functionality will be a memory subsystem; our second will be the interlock subsystem designed earlier. With the processor implemented, operating, and integrated with the memory and interlock subsystems, we will put our knowledge of C to work by developing several simple programs, running them on our processor to generate commands initially to our memory subsystem and ultimately to the interlock subsystem.

Prerequisites:

Familiarity with the Quartus development environment and the *Signal Tap* (nope still ain't *Spinal Tap*) logic analyzer will definitely help. A continued willingness to learn and to explore. No beer until the project is completed and you can turn on a several LEDs using switches on the processor and print the requisite *hello world* and toggle a few lines as you run several C programs on our new microprocessor. Hey, these are all still good.

Cautions and Warnings:

Now that we are working with a microprocessor, always make certain that the cables connecting the PC to the DE1-SoC board are not twisted and don't have any knots. If they are twisted or tangled, the compiled C instructions might get reversed as they are downloaded into the target and your program will run backwards. Also, try to keep your DE1-SoC board lower than your PC thus making it easier for the electrons to get to your board and making the data transfer much faster. However, this will not affect the speed at which your program runs. Also be careful that you do not get the data moving at too high of a rate otherwise it may continue on past your target system. If that does occur, try adding code to indicate that the system is to perform no operation (NOPs) in critical places in the stream. These can be lost with little damage.

Background:

Have some knowledge of the C language and ability to develop simple C programs. Have a working understanding of basic methods of system I/O such as switches and LEDs.

Objectives:

The major objectives of this project include:

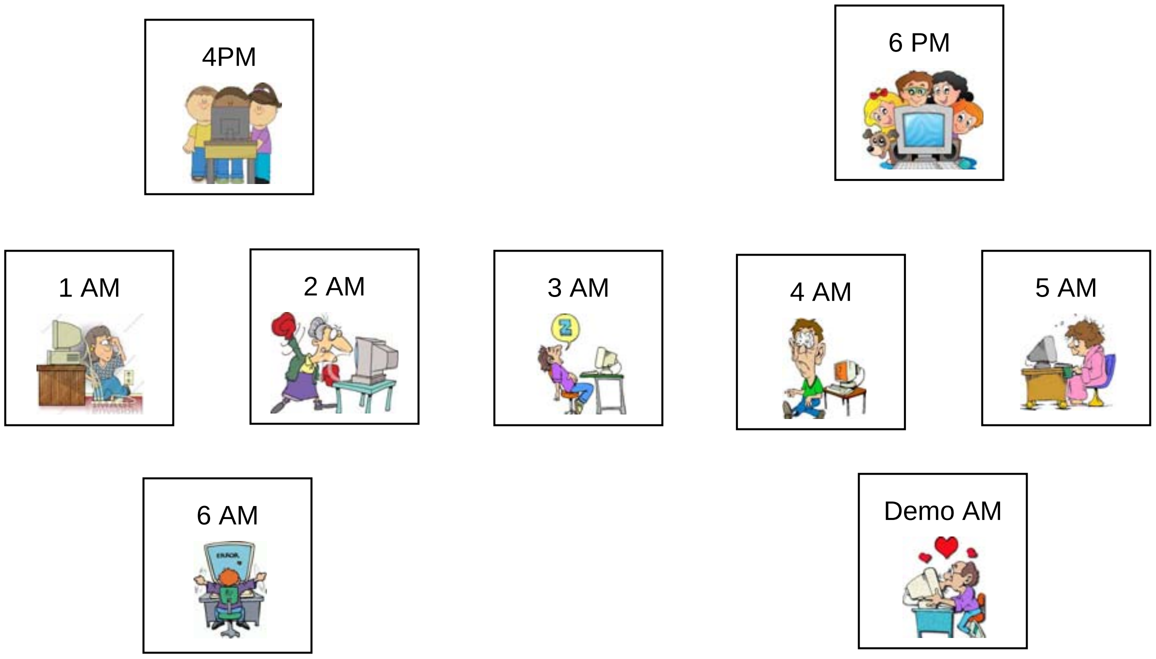
- Design, implement, and test a simple microprocessor on the DE1-SoC board FPGA.
- Design, implement, and test a fundamental I/O interface to the microprocessor.

- Build on a basic understanding to arrays to design and implement an SRAM and associated software driver.
- Integrate the interlock management subsystem into the habitat system and use the microprocessor to control it.
- Use our knowledge of the C language to develop and execute some simple C programs to put our new microprocessor to work.
- Build a foundation for the final project.

Designing and Building a Microprocessor and C Application:

For the first part of the project, we will work with a new Quartus tool called Qsys to design and implement a simple microprocessor based upon the Altera NIOS II core. We will then use the Eclipse based NIOS II IDE to develop several C applications that will run on that processor.

Basic Microprocessor Development Cycle



Building a Basic Microprocessor

For this first part of the project, we will focus on designing and implementing the core microprocessor system. Our primary tool will be the Altera Qsys system integration tool that is intended to support the design of digital hardware systems containing a variety of different digital components.

To facilitate getting started with this tool, we have placed several relevant documents on the class web page, under ***documentation/NIOS, Tutorials, UsersGuides, Handbooks***. Note that the Tutorials have nothing to do with King Tut or his tutor, although it may have had roots in Greek mythology.

Note: You do not have to download any additional Altera software for this project and you do not need the Altera Monitor Program.

Find the document *NIOSii_hardware_tutorial.pdf*. Read through it, then follow the instructions, step by step, to build and download a NIOS II based system to the DE1-SoC board. Pay attention to the note above this paragraph.

We have the following points of modification and clarification for the tutorial.

Page 1-9: Start the Quartus system and create a new project. Rather than working with a bsd file as the top-level module, we will begin with an empty Verilog file. Add the file to the project and ensure that the name of the project and the top-level Verilog module are the same. It is not necessary to use the names suggested by the tutorial.

Proceed with the tutorial.

Page 1-11: ***Specify Target FPGA and Clock Settings***: the ***Clock Settings*** and ***Project Settings*** tabs are in the main window

Page 1-14: step 1, Select ***Embedded Processors***.

Page 1-23: steps 1-4, ***Generate the Qsys System***

This step generates or creates a Verilog file for your NIOS II processor. If this step is successful, select the ***Generate*** tab again then ***HDL Example***. The example is a template for the interface to your processor. You will need this to specify the I/O in your top-level module and when you assign pins for your system.

Page 1-24: ***Integrate the Qsys System into the Quartus II Project***

a. Create an instance of the NIOS II processor in your top-level module.
b. Add your processor's IP file to your project.

   Browse to your working directory and find the file:
   <workingFilesDirectory>/<myProcessorName>/synthesis/<myProcessorName>.qip
c. Specify your top-level module I/O. Use the information from the ***HDL Example*** template above to identify your module I/O.
d. Compile your project.
e. Assign pins to your project
f. Compile again and program the Cyclone V.
g. Skip ahead to page 1-32.

Developing the Software

Once you have successfully built the NIOS II processor, assigned the pins, and downloaded it to the DE1-SoC board, it's time to do the software part.

Page 1-32: ***Develop Software Using the NIOS II SBT for Eclipse***

Follow the steps to create the firmware and software package that you then run on the microprocessor that you have created and built on the Cyclone V FPGA,

You have now achieved a fundamental goals of electrical engineering. You have managed to flash an LED….in fact many of them.

If you ever get the message that the IDE can't find or identify your target system, elaborate the error and scroll to the far right hand side of that page to the 'refresh connection' button. Select that and all should be good. If not, there is another problem of some kind.

Developing Additional Applications

When you have the first application (*Count Binary*) successfully running, you are ready to explore the Qsys system further.

Adding General Purpose I/O

1. Using the tutorial: *Introduction_to_Qsys_Tool.pdf* from the class web page, work through the tutorial to develop the *lights and switches* system and application. This project illustrates how to develop a basic GPIO (General Purpose I/O) input and output system for the processor. We will use this interface in this and the next project.

Adding Textual I/O

2. From the same environment, and using the NIOS II processor, create a new application. This time, select the template, *hello world small*.
3. Complete the *hello world small* project just as you did the *Count Binary* project. You will now have two systems and two applications.

Adding Textual I/O Again

4. When that is working, make the following modifications to the program:
    a. After printing *hello world…* to the console, the program waits for an input from the user. When the user enters the letter 'G' from the console, the system enters an infinite loop enabling you to control the LEDs from the switches as you did in the *lights and switches* project. You will need to use the 'alt' form of stdio for this project.
    b. In the infinite loop, the system will read the state of the switches. If SW0 is a '1' the system will complement the bits corresponding to SW1..SW7 and then

illuminate the corresponding LEDs. Otherwise, it will illuminate the corresponding LEDs based upon the state of the switches.

Designing and Integrating a Memory Subsystem

For this part of the project, we will build a simple SRAM and corresponding driver interface. The description of typical SRAM and its operation are given in Appendix A. As the first step in this design (as we should do with most designs) we will begin with the data sheet(s) for the part(s) we will be using.

While the description of a typical device may give us a high level understanding of the behaviour of a component, the data sheet gives us a description of and specifications for an actual part. These are what we generally use during the design process.

In this case, we will be designing an SRAM that is based upon the 16K bit CY7C128A architecture and which is organized as 2K x 8. The data sheet is on the class web page in the documentation directory. However, we will not actually be using that specific part. We are using it as a simple reference.

High Level Requirements

Overview

Working from and following the Cypress CY7C128A datasheet, we will design, implement, and integrate an SRAM subsystem as an external peripheral to the NIOS II microprocessor.

Requirements

The required system additions are as follows:

- Working from the specifications and timing diagrams in the CY7C128A datasheet, design a behavioural level Verilog model SRAM and associated driver.
- Design and build the interface between the NIOS II microprocessor and the SRAM.

  Review what you did in the *lights and switches* project.
- To test the interface, the driver should write the binary data 127..0 to the first 128 locations in the SRAM, then read that data back and display it on the LEDs.
- For the second part of this project, use the Signal Tap logic analyzer to display the addresses that you are writing/reading to/from and the data that you are writing/reading to/from each address.

Integrating a Peripheral Subsystem

For the final part of the project, we will incorporate the interlock management subsystem that we designed in the previous project.

High Level Requirements

Overview

The interlock management subsystem designed earlier will now be integrated into the larger habitat system. Based upon the designs done for the *lights and switches* system and application and the *hello world small* project, requests received from the bathysphere will be interpreted and displayed on the system console. Responses to such requests will direct the interlock management subsystem as appropriate.

Requirements

The required system additions are as follows:

- Requests from the operators of the bathysphere will be received as inputs to the NIOS II microprocessor that is controlling the habitat management system. Such requests will be interpreted and displayed on the system command console.
- Commands will be entered on the system command console in response to incoming requests from the bathysphere operators.
- Console commands will interpreted by the habitat management system which will then generate and send the necessary control signals to the interlock management subsystem to affect the appropriate response.

Deliverables:

A lab demo showing…

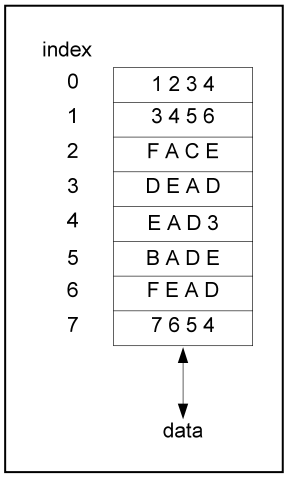
1. The *countBinary* design and working implementation that meets the specified requirements.
2. The *lights and switches* design and working implementation that meets the specified requirements.
3. The *hello world small* design and working implementation that prints the ‘hello world’ message to the console.
4. The *hello world small* design and working implementation extended to support, console I/O and the modified functionality of the *lights and switches* program as specified in the requirements.
5. An integrated and working NIOS II - memory subsystem design that meets all of the specified requirements.
6. A working interlock management subsystem and command console integrated into the NIOS II based habitat management system.

A lab report that complies with the *Lab Format* specification given on the *Workload and Grading* page of the class web page. Ensure that your report contains:

1. An abstract, introduction, general description of your hardware and software design, discussion of any problems that you had with the development of the project, a summary, and a conclusion.
2. The annotated Verilog and C source code for all applications both on the DE1-SoC board and on the NIOS II processor.
3. Answers to any questions above.
4. Other things that you deem to be important.
5. Anything that we haven't thought of.
6. Signature page, signed and stating that the report and its contents are solely your team's work and cites outside references for work that is not your own.

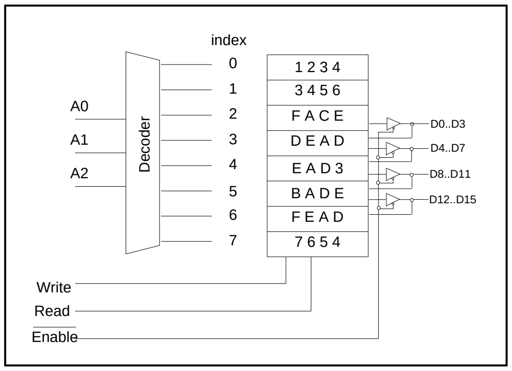
Appendix A SRAM Fundamentals

As a first level model of memory, we can view the device as an array. A value can be assigned to a location in an array and the value of a piece of data that has been stored there can be read. For an array, we identify where the data is stored by an index number. The diagram in the accompanying figure illustrates a simple array with eight entries. For each index that is accessed, the corresponding stored value appears on the output. Conversely, if one provides an index and an input value, the data will be stored at the corresponding indexed location.



One could seamlessly carry this mathematical model into a physical implementation. In doing so, however, one finds rather quickly that several difficulties arise. First, using one index value for each entry will very quickly lead to a substantial number of input signals. That problem can be solved by encoding the index value as a binary number that is defined as an *address*. The binary encoded address can now easily be decoded into the corresponding index value. In the mathematical model, a *read access* at a specified index automatically returns the stored value and a *write access* stores a new value.

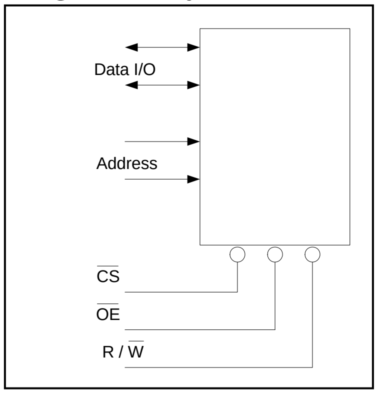
The physical model requires a bit more work. One must *control* the *data* lines going into and out of the memory. One must also *control* when the *read* and *write* operations take place. The diagram in the adjacent figure illustrates how such capabilities might be added to the array model.



The three-to-one-of-eight decoder converts the incoming address patterns into the equivalent in index numbers. The read and write signals perform the associated operations. Prior to writing, the output drivers for the memory must be placed into the high impedance state so that there is no conflict with the incoming data words. For this simple model, the data is entered into or read from the memory as a sixteen bit word.

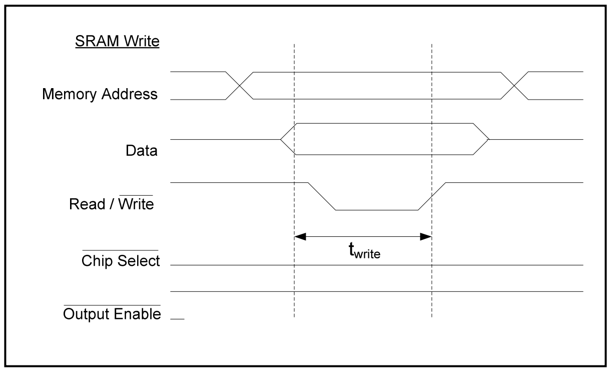
Thus, we see that a memory interface generally requires three categories of signals, *address*, *data*, and *control*. *Address* signals are inputs to the memory, *data* can be either an input or an output, and the *control* signals are generally inputs. All of the different memory types require both address and data signals. They differ in the number and the nature of the necessary control signals.

A high-level interface to the SRAM is given in the next figure that illustrates the major I/O signals.



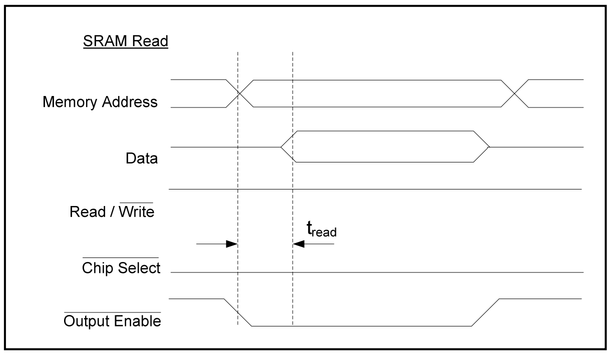
Write Operation

A value is written into the memory as illustrated in the following timing diagram. The control signal *Chip Select* is placed into the logical 0 state to select the device and thereby enable reading from or writing to the device. The control signal *Output Enable* is placed into the logical 1 state to disable the output drivers so as to allow data to be written into the device. The *address* lines are set to the value of the address to which the data is to be written, the desired data values are placed on the *data* lines, then, the Read/~Write control line is strobed low then high to perform the write operation, just like we do on a flip-flop. The assumption here is that the write take place on the rising edge of the Read/~Write line.


SRAM Write
Memory Address
Data
Read / $\overline{\text{Write}}$
t_{write}
$\overline{\text{Chip Select}}$
$\overline{\text{Output Enable}}$

Read Operation

The read operation is preformed in a similar manner as we see in the next timing diagram. Once again, the control signal *Chip Select* is placed into the logical 0 state to select the device and thereby enable reading from or writing to the device. The control signal *Output Enable* is now placed into the logical 0 state to enable the output drivers to send data out of the device. The *address* lines are set to the value of the address from which the data is to be read. The Read/~Write control line placed into the logical 1 state to enable the read operation. After a propagation delay, the data that had been stored at the specified address appears on the *data* lines.


SRAM Read
Memory Address
Data
Read / $\overline{\text{Write}}$
t_{read}
$\overline{\text{Chip Select}}$
$\overline{\text{Output Enable}}$