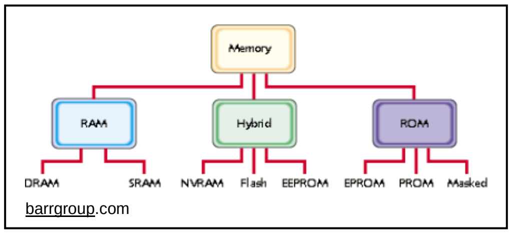
EE 469 Lab 2
Working with Memory – An SRAM and a Register File

University of Washington - Department of Electrical Engineering

James K. Peckol, Matt Staniszewski, Hoon Kwon, Andrew Lawrence

Introduction:

In this project, we will design, implement, and test three modules in a memory system: an SRAM and an SRAM interface and a register file. We will design these using the Verilog HDL, implement them on the Cyclone V FPGA, and verify each design using a test suite and the Signal Tap logic analyzer. We will then integrate the two modules and test the resulting subsystem. Ultimately, we will incorporate that subsystem into the computer that we will be designing.



barrgroup.com

We will also continue to work with the C language, introducing functions, basic IO, and look at some good design techniques.

Prerequisites:

Familiarity with Verilog modeling and the Quartus II development environment and the Signal Tap logic analyzer. A continued willingness to learn and to explore.

No beer until the project is completed, you can turn on several LEDs, remember a few things, and finish all your dinner before dessert.

Cautions, Warnings, and Questions:

Never sneak up behind a static ram in sleep mode with a pair of shears yelling 'hey sheep, time for a haircut' or you may encounter a rather upset dynamic ram in active mode.

Can you program a field programmable gate array in the lab or do you have to go out into the field? To work in the lab, do you have to buy a special version of lab programmable gate array?

If you are working with a computer and you get a stack overflow, where do the extra bits go to? Are they lying around inside your computer? Can you save them for later in case you encounter a stack underflow or do you have to use them immediately or can you solve the problem by flushing them down the pipeline and out to C? Would it be better to save them in a bit bucket in case we have a bit of a draught this summer? Life is so full of challenges and dilemmas.

Can a tristate buss go to only two states? To only one? Do they have to go to three?

Background:

For this project, you should be comfortable with arrays and how to enter and retrieve data from them. In addition, you should have a working understanding of tristate devices in support of bidirectional busses, basic registers with parallel read and write capability, decoders, and multiplexors. Finally, a solid working understanding of finite state machines and how to use them as a means to control the behavior of a digital system.

Most successful engineers begin a design by identifying the requirements, formulating a specification, then creating a high-level architecture for the system they are intending to design. Our requirements for this project are given in the following paragraphs. We will bypass the formal specification for the nonce, focus on the functionality, and begin with the RTL architecture, given in the following figure, for a computer that supports the functionality that we have discussed in class. The focus of this project is the two highlighted modules.

As the project evolves, we will revisit, and potentially modify the architecture.

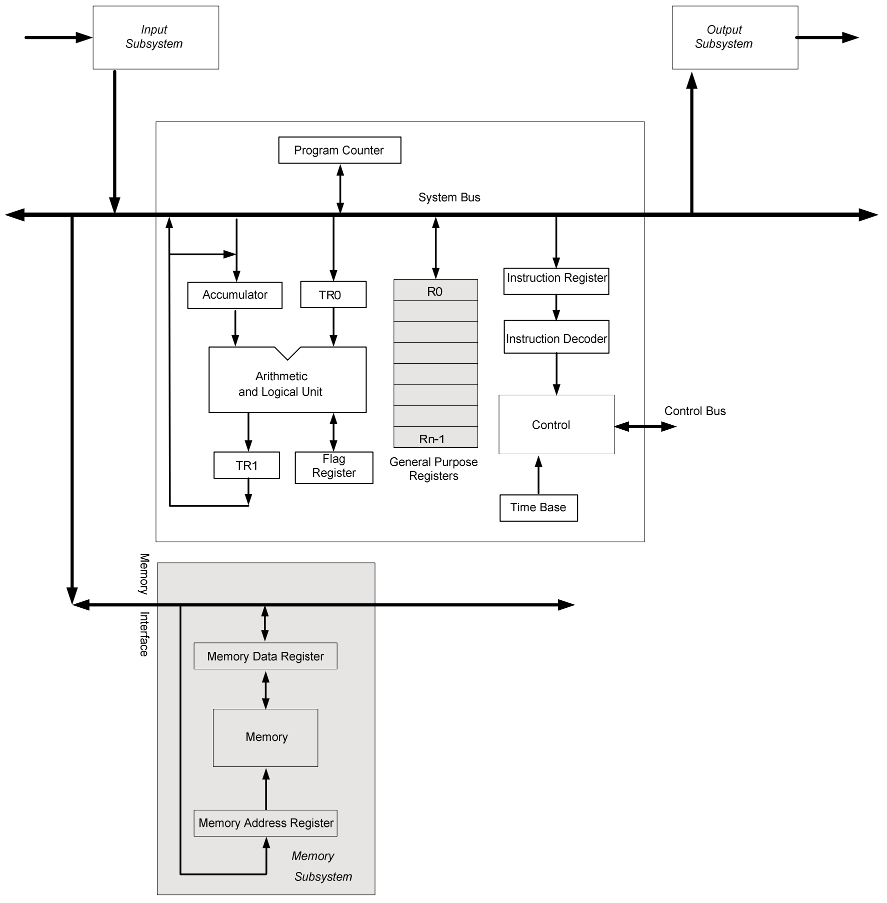


Figure 1 – The Architecture for a Basic Computer

Objectives:

The major objectives of this project include:

- Continue to learn and work with the various Verilog modeling levels.
- Build on a basic understanding of arrays to design and implement an SRAM module and a driver for the device.
- Build on a basic understanding of registers and multiplexors to design and implement a hardware module called a register file.
- Take several first steps in the process of subsystem integration and begin to learn how to model the missing pieces.
- Learn and work with basic C functions.

Working with an SRAM

For the first part of this project, we will design and build a basic static RAM or SRAM. The description of a typical SRAM and its operation are given in Appendix A of this project spec.

As the first step in this design (as we should do with most designs) we will begin with the functional definition of the desired features and capabilities.

Working from the description of the basic device given in Appendix A, design and implement an SRAM using behavioural level Verilog on the Cyclone V FPGA. The design must support the following capabilities and control signals:

1. A 2K x 16 architecture
2. Word addressable
3. A bidirectional 16 bit Data Bus
4. An 11 bit Address Bus
5. Low true Output Enable – OE
6. Read / Write control. Low true Write Enable → ~WE, High true Read → WE
7. Clock

Note that the SRAM in Figure 1 utilizes both a Memory Address Register, MAR and a Memory Data Register, MDR. Your design must include these.

To test your design, write the binary data 127..0 to the first 128 locations in the SRAM – binary data 127 should go into address 0 and binary data 0 into address 127. Then read that data back and display it on the DE1-SoC LEDs.

For the second part of this portion of the 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.

Can you design your test system so that you can choose the system clock frequency to support readable LED display or Signal Tap display without recompiling your code?

Designing and Building a Register File:

The high-level system architecture above specifies the requirement for a set of N general purpose registers. We will implement this register set as a 32 by 32 (32x32) register file. For this design, we will work at the structural Verilog level.

The general idea behind a register file is that one can read from multiple registers simultaneously and write to only one register. To be able to do so, the design must support the ability to individually address or select each register for read or write access.

Such a concept is common in file systems on a hard drive, for example. Many programs can open and read from a file at the same time; however, only one can change or write to it.

A high-level diagram for the MIPS 32 x 32 register file is illustrated in the accompanying figure. The control addresses *Reg1 Read Select* and *Reg2 Read Select* specify the registers whose values are output on the *Reg1 Read Data* and *Reg2 Read Data* buses respectively.

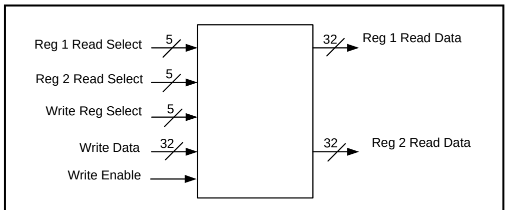


Figure 2 – A 32 x 32 Register File

The control address *Write Reg Select* specifies the target register of a write operation to the register file; when *Write Enable* is asserted and a clock occurs, the information on the *Write Data* bus register is written into the specified register.

Moving inside the device, simple block diagrams for the read and write portions are given as,

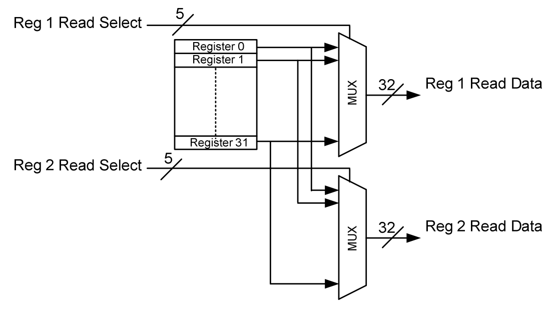


Figure 3 – A 32 x 32 Register File Read Block Diagram

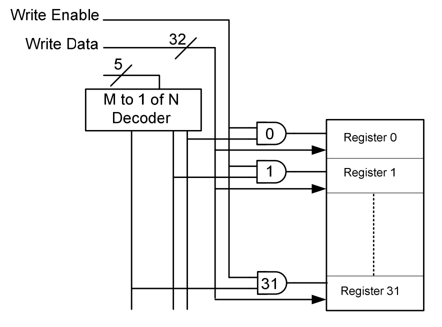


Figure 4 – A 32 x 32 Register File Write Block Diagram

Note that there is only one set of 32 registers, designated Register 0…Register 31. There is not one set for read and another set for write. Each register is built from 32 D flip-flips – one flip-flop per bit.

Schematically, in the diagrams above, the multiplexors look deceptively simple. Designing and implementing such a module can be a challenge. Think carefully about the design, explore a little, discuss it with your partners, and sketch your ideas out with pencil and paper before digging deeply. Look for the second right answer.

Remember, on the MIPS machine, Register 0 is hardwired to always output the value zero, regardless of what may or may not be written to it. Why might we want to do this? We want to include this feature in our design.

To verify the operation of the register file, write the hex values 0xFFFF000F…0xFFFF0000 to registers Register 0..Register 15 and the hex values 0x0000FFF0…0x0000FFFF to registers Register 16..Register 31.
Next, read the contents stored in the register pairs Register 0, Register 16; Register 1, Register 17….Register 15, Register 31. If KEY0 is a logical 0, successively display the least significant 8 bits from registers Register 0…Register 15 on LEDs LEDR0..LEDR7and if KEY0 is a logical 1, successively display the least significant 8 bits from registers Register 16…Register 31 on LEDs LEDR0..LEDR7.

Subsystem Integration – A First Step

We will now integrate the SRAM memory subsystem and the register file subsystem. Connect the data lines of the SRAM to the data lines of the register file so that they now share the same data bus. Take note that the bus must support the bidirectional movement of data between the SRAM and the register file.

The remainder of the computer modules are not available at this time, so, to test the functionality of the combined memory subsystem, we must model the behaviour of the missing pieces. To that end, we need to develop a test module that will allow us test the operation of the memory subsystem.

You need to test as follows,

1. Write the binary data 0..127 to the first 128 locations in the SRAM – binary data 127 should go into address 0 and binary data 0 into address 127. These data will comprise four 32 word blocks, blocks 0..3. This models data being stored from ALU operations or data blocks transferred from main memory to cache, for example. Also think about MIPS memory addresses.
2. Read the words in block 0 from the SRAM, and write each of the 32 words, in sequence, to the 32 registers in the register file. This models data transfers from memory (cache, for example) to registers for later operations.
3. Successively read and display on the DE1-SoC LEDS the contents of registers *Register 0..Register 15* as *Reg1 Read Data* and then the contents of registers *Register 16..Register 31* as *Reg2 Read Data.* Use KEY0 to select which set of registers is being displayed. This tests the ability to simultaneously read from two registers.
4. For blocks 0 and 2, write the successive values of *Reg1 Read Data* to SRAM locations 128..144 and 162..178. For blocks 1 and 3, write the successive values of *Reg2 Read Data* to SRAM locations 145..161 and 179..195. This tests the ability to transfer data from two registers into memory (cache, for example).
5. Repeat steps 2 and 3 for blocks 1..3.
6. Verify using Spinal Tap that the correct data has been written to the SRAM. This verifies that data can be written correctly to SRAM.

Learning the C Language – The Next Steps:

We will conclude this lab project by learning how to use basic functions in the C language.

Working with C in the PC Environment

Building a Program

As our next C program, using the library functions *printf()* and *scanf()*, write a function to prompt the user for the duration of a nonstop flight to London from Seattle, in hours, as a floating point number. Based upon the number of air miles from Seattle to London, compute, then write a second function print out the estimated velocity of the aircraft as a floating point number. Using the velocity you computed and an estimated head wind of 89.6 miles per hour, write a third function compute the estimated duration of the flight as a floating point number. Finally write a fourth function to print the estimated flight duration.
Use symbolic constants in your design rather than magic numbers as appropriate.
Remember, the *scanf()* function delimits input by whitespace.

The simple program lab2Functions.c on the class web page illustrates how we can use functions in C.

Deliverables:

A lab demo showing…

1. The SRAM design and working implementation that meets the specified requirements.
2. The Register File design and working implementation that meets the specified requirements.
3. A working integrated SRAM and Register File module that meets the specified requirements.
4. A working C program that meets the specified requirements.

A lab report containing…

1. The annotated Verilog and C source code for all applications both on the DE1 board and on the PC.
2. Representative screen shots showing the results of testing the various designs.
3. Representative screen shots showing the logic analyzer outputs for the various designs.
4. Answers to any questions above.
5. Other things that you deem to be important.
6. Anything that we haven't thought of.

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.

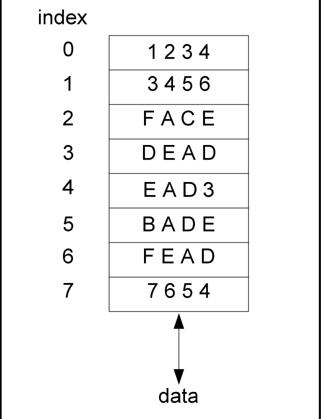


Figure 5 – A SRAM Model

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.

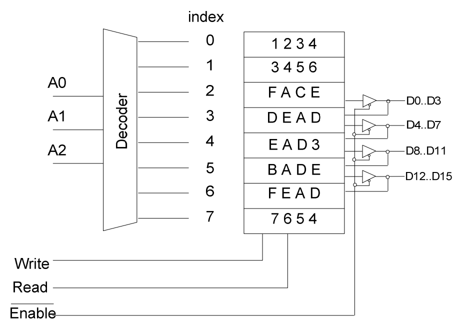


Figure 6 – A SRAM Model

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 which illustrates the major I/O signals.

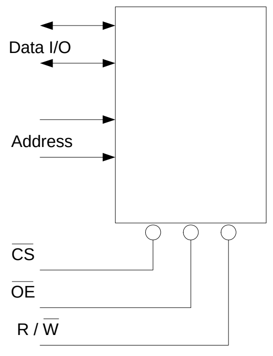
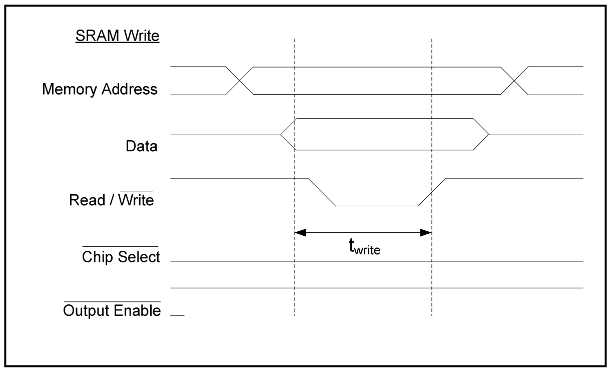


Figure 7 – SRAM Interface

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 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.



Read Operation

The read operation is performed 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.

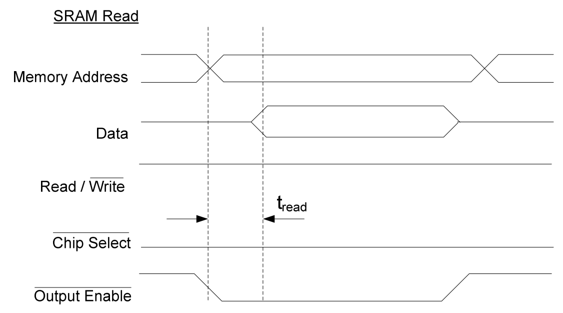


Figure 9 – Basic SRAM Read Timing Diagram