EE 469 Lab 3

Designing an ALU

Working with an SRAM, Reg File, I/O, and C Stuff Cont….

University of Washington - Department of Electrical Engineering

Matt Staniszewski - Hoon Kwon – Andrew Lawrence - James K. Peckol

---

Introduction:

In this third project, we will build upon some of the things that we have learned in the second one then add some new ideas. Our initial objective will be to specify, design, implement, and test a simplified arithmetic and logic unit – ALU. The ALU will support the arithmetic operations addition and subtraction and the logical operations shift, comparison, AND, OR, and exclusive OR. We will then integrate the ALU with the SRAM and register file subsystems.

Finally, we will continue to grow our knowledge of the C language by working with variables and their addresses. We will see that our earlier work with the SRAM carries directly to our studies of C variables and their addresses.

Prerequisites:

Familiarity with the Quartus II development environment and the Spinal Tap logic analyzer. A continued willingness to learn and to explore. No beer until the project is completed and you can turn on a several LEDs. Hey, these are all still good.

Musings:

Are cross purposes really angry or just a little upset? If C and C++ support casting integers and floats, why don't they also support casting aspersions? Can fishermen cast aspersions? How do you cast adrift? Why do the same kinds of animals make different sounds in different countries? Do they really learn to speak the local language? If a Chinese rooster comes to the U.S., does it crow in English or Chinese? Which dialect? Does it have an accent? If you told a French dog to lay down, would it understand? Why is Donald Duck a duck not a drake? Is it possible to defenestrate an elephant? What would happen if an elephant swallowed a mangle wurtzel then sneezed? If water going down a toilet goes anticlockwise south of the equator, do bottles and jars open clockwise?

Observations:

With the latest in high tech ultra child-proof containers, only the truly intelligent children will be able to open the medicine bottles and ingest what they shouldn't. Kind of makes an interesting statement, doesn't it.

Background:

For this project, you should understand and know how to implement the basic arithmetic and logical functions add, subtract, and, or, exclusive or, shift, and compare at a more detailed level. Have a working understanding of registers and how to enter, hold, and retrieve data. Understand finite state machines and how to use them as a tool for controlling the behavior of a digital system. Have a working understanding of basic methods of system I/O such as switches and LEDs.

As we noted in our previous project, most successful designers begin the design phase of a project by formulating a high-level architecture for the system they are intending to develop. We will continue with the architecture that we have discussed in class as given in the following figure. In the previous project, we completed the design and integration of memory and register file subsystems. In this project, we will incorporate the ALU.

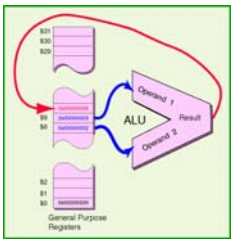

Figure 1 REG and ALU

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

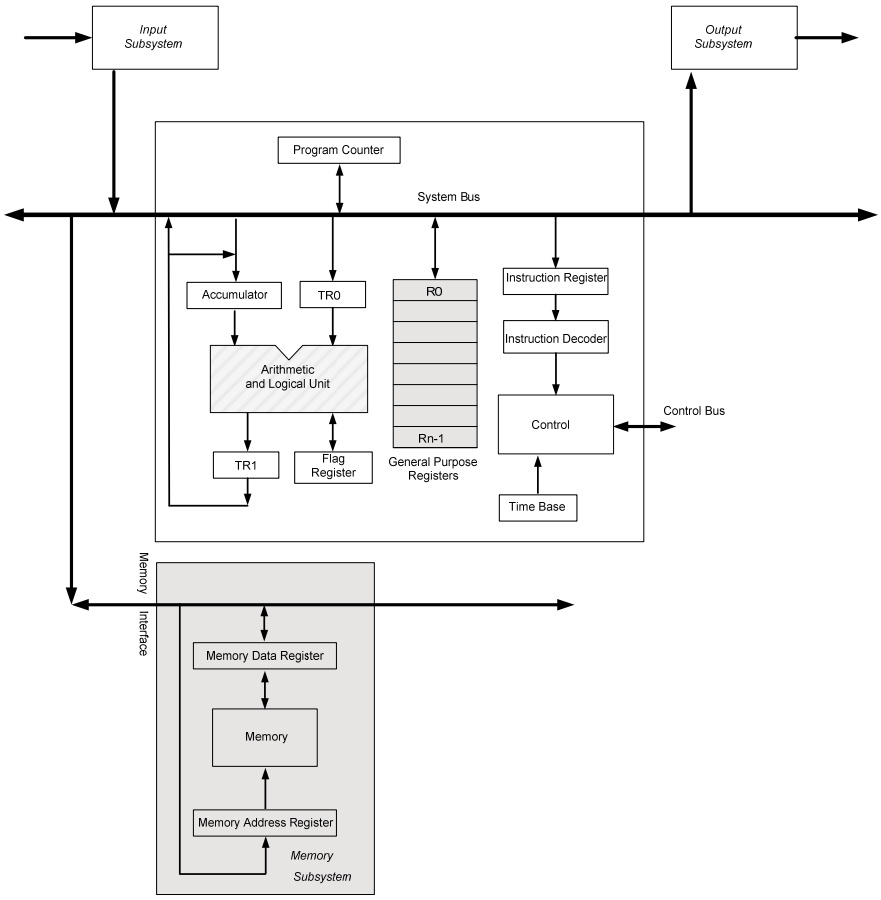

Figure 2 High Level Architecture – ALU Highlighted

Objectives:

The major objectives of this project include:

- Design, implement, and test the circuitry to support the arithmetic operations ADD and SUBTRACT.
- Design, implement, and test the circuitry to support the logical operations shift, comparison, AND, OR, exclusive OR.
- Integrate the ALU subsystem with the SRAM and register file subsystems.
- Learn and work with basic C variables and their addresses.

Designing and Building an ALU:

For the first part of the project, we will design, implement, and test a basic ALU. We will then integrate it with the memory subsystem we developed earlier. This design will only support a subset of the arithmetic and logical functionality typically found in an industrial strength ALU. A high-level block diagram for the unit is given in Figure 3.

The subsystem supports three unidirectional data busses (two in and one out), one control bus, and four status flags or bits.

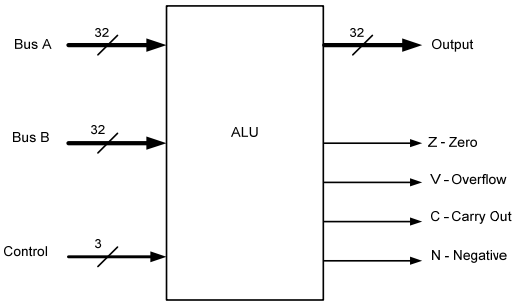

Figure 3 Basic ALU I/O

We will initially model the subsystem at the behavioural level to ensure the unit will comply with the *functional* specifications then refine the design and implement the unit at the dataflow or RTL level to ensure that we meet the operational (specifically temporal) constraints.

The ALU will support the following operations:

Arithmetic

1. Addition
2. Subtraction

Logical

1. AND
2. OR
3. XOR
4. SLT – set less than (see your text)
5. SLL –shift left logical 0..3 positions

All operations must be completed in a single clock cycle.

To satisfy the single clock cycle constraint and yet support the multibit shift operation, the shifter must be implemented as a barrel shifter. See the paper on barrel shifters on the class web page under documentation.

Subtraction must be implemented using complementary arithmetic.

The ALU control codes are specified in Table 1.

Control Code	***Function***
000	NOP
001	ADD
010	SUB
011	AND
100	OR
101	XOR
110	SLT
111	SLL

Table 1 ALU Control Codes

Requirements and Operation – Phase 1

For the first part of the project, we will verify the core functionality of our design. For this phase, we will use the switch and key combinations and hex LED displays on the DE1-SoC board given in Table 2,

SW3..SW0

Enter four hex digits as the data for operands A and B.

SW6..SW4

Specify the operation / control code.

Select Code ***SW9 SW8***		***Function***
0	0	Display/Modify A
0	1	Display/Modify B
1	X	Display Result

Table 2 Function Select

SW9 .. SW8

Specify whether operand A or B is to be entered or the result is to be displayed.

KEY[0]

Interpreted by the system as an *ENTER*. When the *ENTER* is pressed, the ALU will read the state of the input switches and respond accordingly.

KEY[1]

The RUN command, directs the ALU to perform the specified operation and display the results.

Hex3..Hex0

Display the operands or the results of the operation in hex, on the, on the DE1-SoC board.

Caution: Be aware that the key switches may bounce thereby giving an incorrect entry.

Requirements and Operation – Phase 2

For the second part of the project, we will now integrate the ALU with the memory subsystem and verify the full functionality of the combined modules. As we did earlier, we must model the control algorithms and the missing hardware pieces of the computer system that we are developing. Such a test driver must again be incorporated onto the Cyclone V FPGA and must support the following operations.

Figure 4 Static RAM

Setup

1. Ability to load instructions into SRAM.
2. Ability to load data into SRAM.

During such steps, we are playing the role of the *loader*.

Run

1. Ability to transfer a block of data from SRAM to the register file.
2. Ability to read and interpret an instruction specifying an ALU operation from SRAM. Here we are modeling the *decoder* block.
3. Ability to store the result of an ALU operation into the register file.

Use the Signal Tap logic analyzer to verify the intended operation of your subsystem.

Your test suite should verify the correct operation of each ALU operation as well as the ability to properly assert each of the status flags {*Z, V, C, N*}, *zero*, *overflow*, *carry out*, and *negative*.

Learning the C Language – The Next Steps:

We will conclude this project by introducing and working with pointer variables in the C language.

Working with C in the PC Environment and Some of the Tools

Part 1 – Getting to Know Pointers

In our last several projects on the DE1-SoC board, we have been learning about and working with the SRAM. In doing so, we wrote some data values to memory, then read them back and displayed them on the LEDs. When we wrote the values of the data variables into memory, we placed each one at some address. When we preformed a read operation at each such address, we retrieved the value that we had stored earlier. We did so, by reading from the SRAM at the addresses where we had stored our data.

Figure 5 Pointer
chestofbooks

There was nothing particularly magical (although it may have seemed so at the time) about either the write operation we performed or the read we executed to get the data back out of memory. If things worked properly, then we got back the same data values that we had stored sometime earlier.

Now that we've gained some facility in working with memory, addresses and storing and retrieving data, let's move that knowledge up to the C level (or for that matter, any other programming language). In a language such as C, when we make a declaration / definition and initialization such as:

```
int x = 9;
```

we are doing exactly what we've been doing in the last several projects. That is, we are taking, in this case, an integer and putting (or storing) it in memory as we see in the accompanying diagram.

Figure 6 Memory

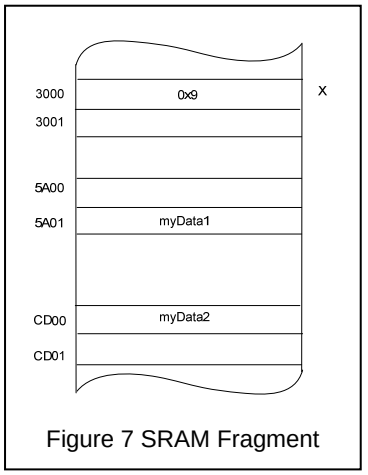

Figure 7 SRAM Fragment

The location where we store it has an address; exactly as we've been doing. Here that address is 3000. We are assigning or storing a value, in this case '9' into that memory location.

In practice, this operation will be accomplished by the efforts of the several of the software tools we've discussed earlier: the *compiler*, the *linker*, and the *loader*.

In C or C++, we can find out what that address is by using the address of operator '&'.

If we write the expression:

```
&x;
```

we can get the address of x in memory. That is, where the *loader* put the variable. In this case, this operation will return the value 3000.

If we assign that address to some other variable, we can use it at some later time to retrieve the value of the variable at that memory address or location. The variable to which we assign the address *points* to that specific location in memory. Hence, very cleverly, we call such a variable a *pointer*.

Bear in mind that each time the program is loaded into memory, the variable may be placed at a different location.

When we use a pointer variable, we distinguish it from another type of variable by writing it with a '*' preceding it.

Thus,

```
int* xPtr;
```

is the declaration of a variable that is used to contain or hold a memory address – specifically, the address of an integer.

It also has a location in memory as we see in the next figure. Here, the variable has been stored in address CD01.

Notice that unlike the value stored in memory as the value of the variable x, the value of the variable xPtr is shown as x. This is because the value for the pointer is unknown; we did not initialize it to anything. We have not given it a value.

Now, if we read each portion of the expression above from right to left we can begin to see exactly what is going on. Let's go, from the right hand side, the variable called *xPtr* (the identifier thingy) is a *pointer* (the * thingy) to an *integer* (the int or type thingy).

We could also write,

```
float* yPtr;
```

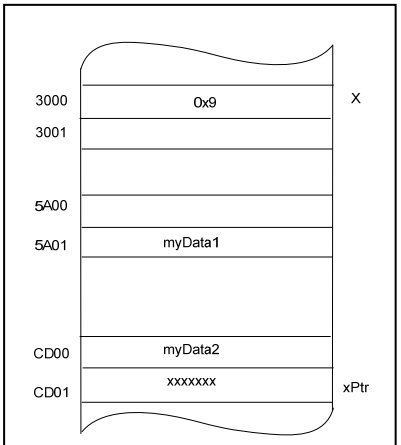

Figure 8 Pointer Storage

Once again, if we read from right to left, the variable called *yPtr* (the identifier duder) is a *pointer* (the * duder again) to a *float* (the float or type duder).

Each such pointer variable above gets assigned the address of the thing that it points to by taking the address of the thing to be referred (using the address of operator, &) to then assigning it to the pointer. Gees, what did you just say? That's confusing. Let's take another look at this.

Ok, let's assume that we first define a variable. In the first case, it's an int called x and in the second a float called y.

```
int x = 9;              //  declare a variable of type int called x and initialize it to 9
```

or

```
float y = 3.2;          //  declare a variable of type float called y and initialize it to 3.2
```

Now, let's declare some pointer variables. The first is a pointer to an int called xPtr and the second a pointer to a float called yPtr. We see xPtr in memory at location CD01 and its value is the address of x (look ahead a couple of steps). Observe that in selecting the pointer names, we add the suffix Ptr to distinguish the variable as a pointer. It just helps to make our code more readable.

```
int* xPtr ;             //  declare a variable of type pointer to integer
```

or

```
float* yPtr;            //  declare a variable of type pointer to float
```

At this point, what are the values of the pointer variables, xPtr and yPtr?

Finally, let's make the two pointers point to the two variables, x and y. We'll use the address of operator '&' to find the addresses of the two variables then make the assignments

```
xPtr = &x;          // then assign the address of x to it
```

or

```
yPtr = &y;          // then assign the address of y to it
```

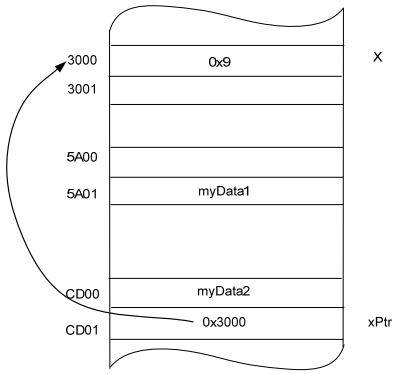

Figure 9 Pointer Dereferenced

If we wish to use the value at the memory address referred to by the pointer variable, then we perform an operation called *dereferencing*.

That proceeds as follows, as we see in the accompanying diagram for the first expression…

```
int z = *xPtr;      // go to the address in memory identified by
                    // the variable xPtr get the value at that
                    // address and assign it to the variable z
```

or

```
float w = *yPtr;    // go to the address in memory identified by
                    // the variable yPtr get the value at that
                    // address and assign it to the variable w
```

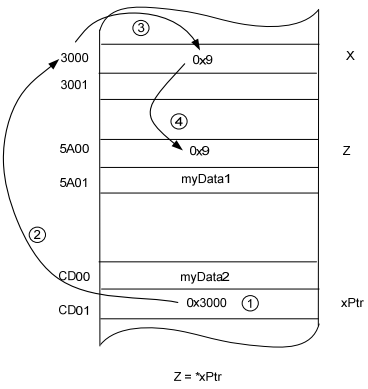

Figure 10 Value Retrieved & Assigned

These steps are exactly what we have been doing when we wrote to or read from memory in our calculator projects.

What do you think would happen if we wrote the following instead…

```
int z = *yPtr;      // go to the address in memory identified by the variable yPtr
                    // get the value at that address and assign it to the variable z
```

or

```
float w = *xPtr;    // go to the address in memory identified by the variable xPtr
                    // get the value at that address and assign it to the variable w
```

As our first step in working with pointers, let's just explore.

Declare several variables of the following types: two variables of type int, two variables of type float, and two variables of type char.

Declare the following pointer type variables: one pointer to int, one pointer to float, and one pointer to char.

Assign the address of one of the integer variables to the pointer to int. Print out the value of that integer. Repeat with the second integer. Repeat with the two floats and then the two chars.

Part 2 – Working with Pointer Variables

Now that we have a little experience with variables and addresses in C, let's put that knowledge to work.

Let's repeat the problem that we solved as the last part of an earlier project. This time, let's work at the C level and with pointers.

Once again, as a first step, declare the following variables of type integer. Initialize each to the values indicated.

A = 25

B = 16

C = 7

D = 4

E = 9

Declare one more variable, *result*, of type integer.

Next, declare and define five variables of type pointer to integer and let each refer to one of the variables.

Finally, perform the computation:

result = ((A – B)*(C+D))/E

only instead of using the variables directly, refer to each through its pointer. Of note, the * above is the multiplier operator, not a pointer thingy.

Deliverables:

A project demo showing…

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

A project report containing

1. The annotated Verilog and C source code for all applications both on the DE1-SoC 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.