EXPERIMENT NUMBER 10&11
REGISTERED ALU DESIGN

Purpose

Extend the design of the basic four bit adder to include other arithmetic and logic functions.

References

Wakerly: Section 5.10

Materials Required

Quartus II and ModelSim

Background

For this lab, you will build a fully-functional registered ALU, like the one shown in Figure 1. This ALU performs a variety of mathematic operations on inputs and stores all results in a local register, the accumulator. Mathematic operations are always performed using the value in the accumulator as one operand and the "input" value as the other operand. The operations performed by the ALU will depend on the state of the control-bits, bits 1-8 in Figure 1. For the ALU discussed here, the control bits will allow the user to (See Figure 5, 6, and 7):

a) select the 4-bit input line or the #0 as one of the ALU operands,

b) add or subtract the value at the input to or from the value in the accumulator,

c) increment or decrement the accumulator,

d) complement the input A,

e) pass the input A straight through,

f) AND or OR A and B together, and

g) shift or rotate the accumulator left or right.

The registered ALU that your TA asks you to build may have slightly different capabilities.

A selector, also known as a multiplexor, "selects" one input and mirrors its value at the output. For example, for the selector in Figure 2, if S is 0, the output will take on the value of D_0. If S is 1, it will take on the value of D_1. For the 4-input selector in Figure 3, a control input of S_1S_0=10 (a binary 2) would select input D_2, and so forth. The inputs, *D*,

and output *Y*, could be single- or multiple-bit busses. For our registered ALU, the selector will allow the user to select between an input value of 0 or the value on the 4-bit input line.

The specific operations that are implemented by an ALU depend on the functions required by the designer. Figures 4 through 6 show one possible design of an ALU, which extends the capabilities of a ripple carry adder by adding a little extra logic at the inputs. The logic is set up such that if the control input M is '1', the ALU performs a math operation (add, subtract, increment, decrement), and if M is '0', the ALU performs a logic operation (AND, OR, complement, or no-change).

The logic block FA stands for Full Adder and is a single-bit adder like you used in the 4-bit ripple-adder in an earlier lab. Let's say you wanted to extend the functions of this adder as shown in Figure 5 (assuming A and B are inputs to the ALU, and X and Y are inputs to the 4-bit adder), adding logic to allow it to decrement the value of input A by 1, to add inputs A and B, to subtract input B from input A, and to increment the value of input A by 1.

Figure 5a shows what values would have to be given at inputs X and Y of the adder to allow it to produce the desired result. For example, A-B=A+B'+1 for 2's complement numbers, where B' is the one's complement of B. So if we wanted to subtract B for 2's complement numbers, where B' is the one's complement of B. So if we wanted to subtract B from A when , we would need to present the 4-bit adder the A, B', and a Cin of 1 (giving a). The problem, then, is to put together logic which allows the 1's complement of B (B') to be presented at input Y when the control bits . The logic needed to modify the a single bit of B and present an appropriate value to the added (Y) is shown in Figures 5b-5d. This logic modifies B depending on the value of S3S2 to Decrement A, Add A+B, Subtract A-B, and Increment A.

The logic in Figure 5 makes up the Arithmetic Extender (AE) shown in Figure 4 – the little bit of extra logic added between the input B to the ALU and the input Y to the adder. Similar operations must be performed on the input A using what is called a Logic Extender (LE – Figure 4). Logic for the Logic Extender is shown in Figure 6. It allows you to perform the logic operations

The one bit of logic we haven't given you is the logic needed to modify Cin (Figure 4). Note that the table in Figure 5a shows that specific values of Cin must be presented to the adder, depending on the operation on wants to perform (the value of S_3S_2). Determine the logic needed to present the proper values of Cin for given control inputs S_3S_2, keeping track of your work in your notebook.

The last component of the registered ALU is the accumulator (Figure 7). The accumulator is a register which stores the result of any ALU operations. The accumulator can be set up so that it can be loaded with a new value, store (keep) an old value, or shift the current value left or right by one (giving your overall registered ALU one more capability). Results are stored using 4 D-flip-flops. New values are loaded into the accumulator with the falling edge of the clock (clk). For the accumulator shown in Figure 7, the accumulator keeps its old value when S_1S_0= 00, is loaded with a new value when S_1S_0=01, shifts the old value left by one when S_1S_0=10, and shift the old value right by

one when S_1S_0=11.

So let's summarize. A registered ALU consists of an arithmetic logic unit and an accumulator (Figure 1). The input to the ALU comes from and external bus (A) and the accumulator (B). Our ALU can add or subtract the inputs, add or decrement 1 from the input A, or can take the current value of the accumulator and shift it right or left by 1. The operation that is performed on any one clock cycle depends on the values of the select bits. The arithmetic operations are performed by modifying inputs to the ALU. The shift operations and load accumulator operations are performed using 1-bit selectors. Most of the logic needed to construct the registered ALU is given for you. A registered ALU like this one forms the basis of a microcomputer.

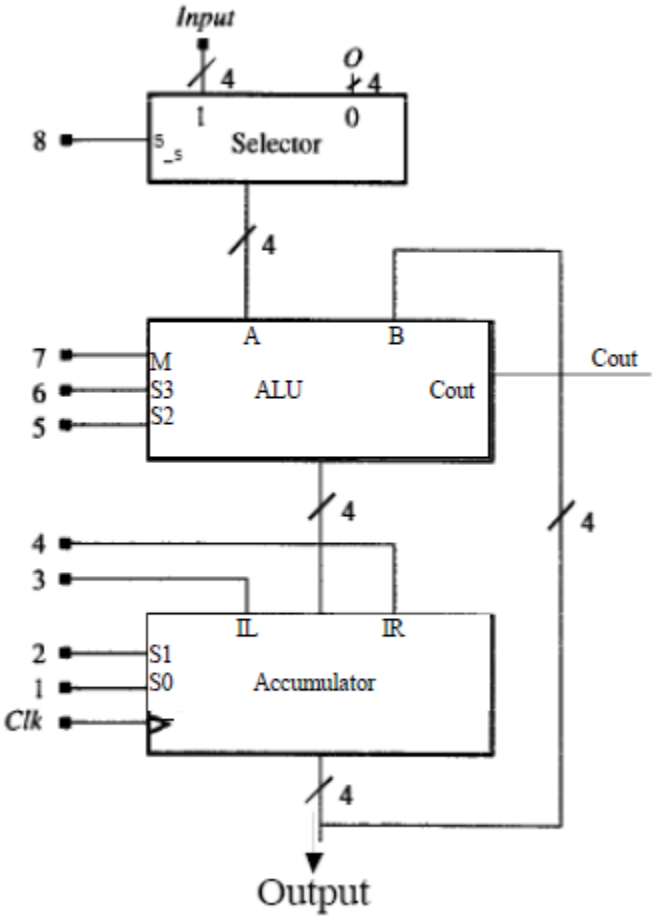


Figure 1. Top-level of registered ALU. Arithmetic operations like add, subtract, decrement, and increment are performed by the ALU. The accumulator is set up to perform shift and load operations. The accumulator serves as one of the inputs to the ALU. The accumulator can be cleared using an input selector that selects between an external input and zero.

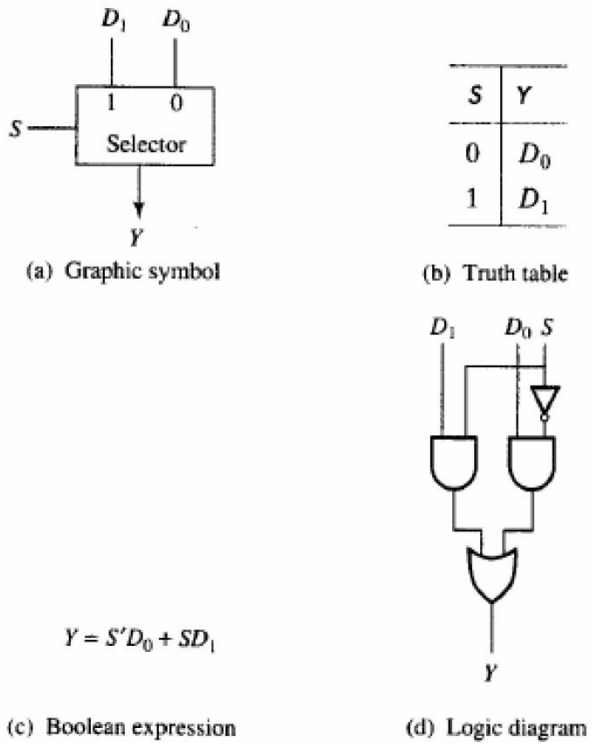


Figure 2 (a): A 1-bit slice of 4-bit selector in Figure 1.

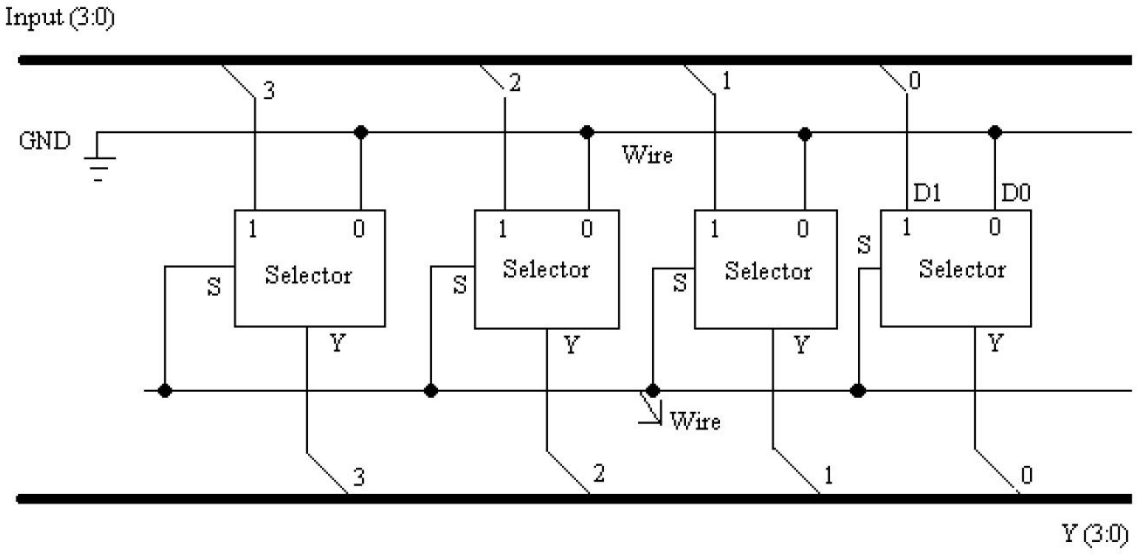


Figure 2 (b): Bank of 4 Selector shown in Fig 2 (a)

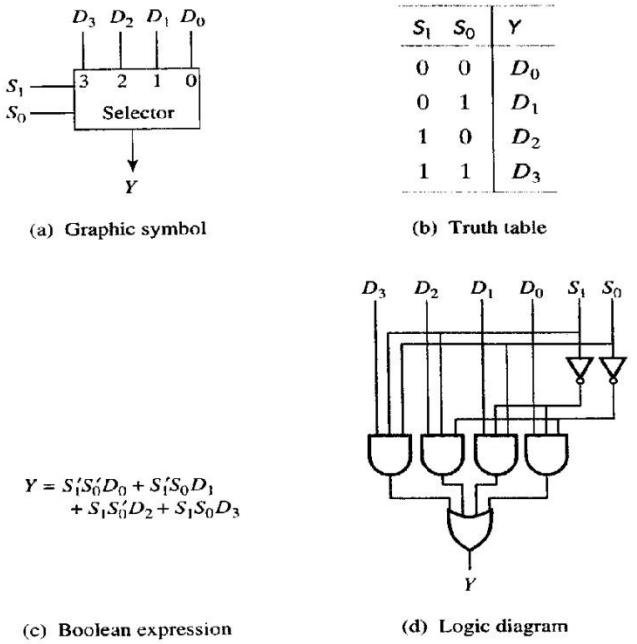


Figure 3. A 4-bit input selector.

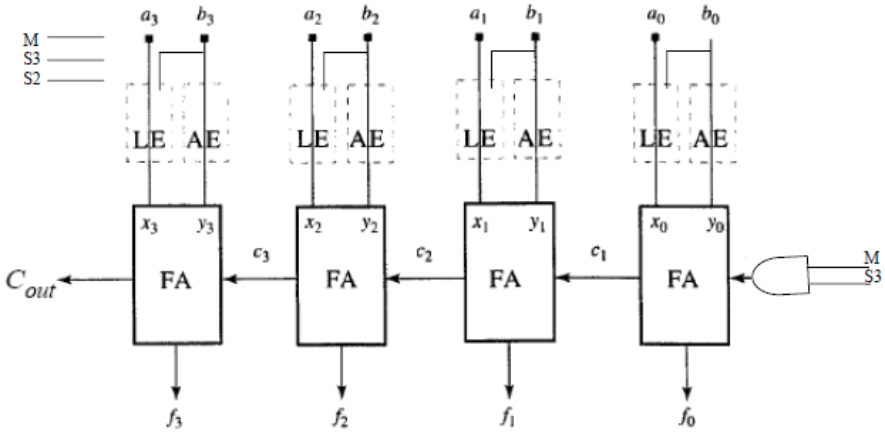


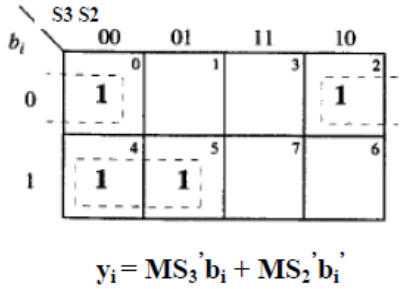
Figure 4. An arithmetic logic unit (ALU). An ALU performs many arithmetic operations, including add, subtract, increment, and decrement. This ALU is constructed from a ripple carry adder and some additional logic that modifies the inputs to the adder. Inputs are modified using a Logic Extender (LE), an Arithmetic Extender (AE), and some carry-in logic.

M	S3	S2	FUNCTION NAME	F	X	Y	C_{in}
1	0	0	Decrement	$A-1$	A	all 1's	0
1	0	1	Add	$A+B$	A	B	0
1	1	0	Subtract	$A+B'+1$	A	B'	1
1	1	1	Increment	$A+1$	A	all 0's	0

(a) Functional table

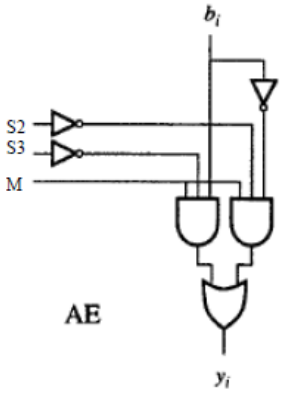
M	S3	S2	b_i	y_i
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

(b) Truth table



$y_i = MS_3'b_i + MS_2'b_i'$

(c) Map representation



(d) Logic schematic

Figure 5. Functions implemented by ALU and AE logic. The decrement, add, subtract, and increment functions can be implemented by modifying the inputs A and B. The logic needed to modify input B, the logic extender, is given in part (d).

M	S_3	S_2	FUNCTION NAME	F	X	Y	C_{in}
0	0	0	Compliment	A'	a_i	0	0
0	0	1	AND	A AND B	a_ib_i	0	0
0	1	0	Identity	A	a_i	0	0
0	1	1	OR	A OR 1	a_i+b_i	0	0

Functions

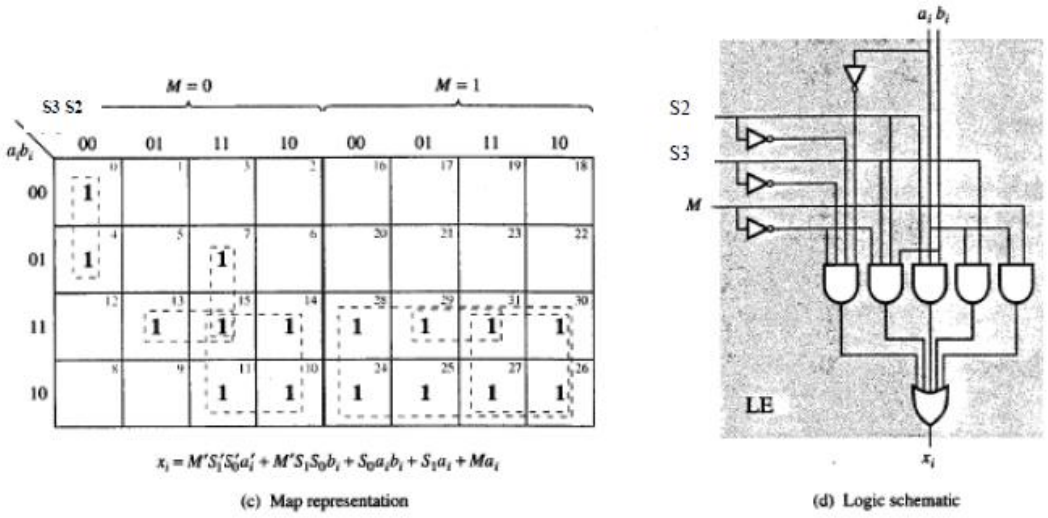
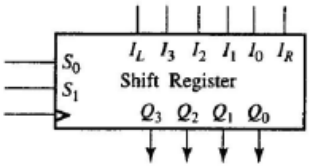


Figure 6. Derivation of logic for the Logic Extender (LE).



(a) Graphic symbol

PRESENT STATE			NEXT STATE			
S_1	S_0	OPERATION	Q_3	Q_2	Q_1	Q_0
0	0	No change	Q_3	Q_2	Q_1	Q_0
0	1	Load input	I_3	I_2	I_1	I_0
1	0	Shift left	Q_2	Q_1	Q_0	I_R
1	1	Shift right	I_L	Q_3	Q_2	Q_1

(b) Operation table

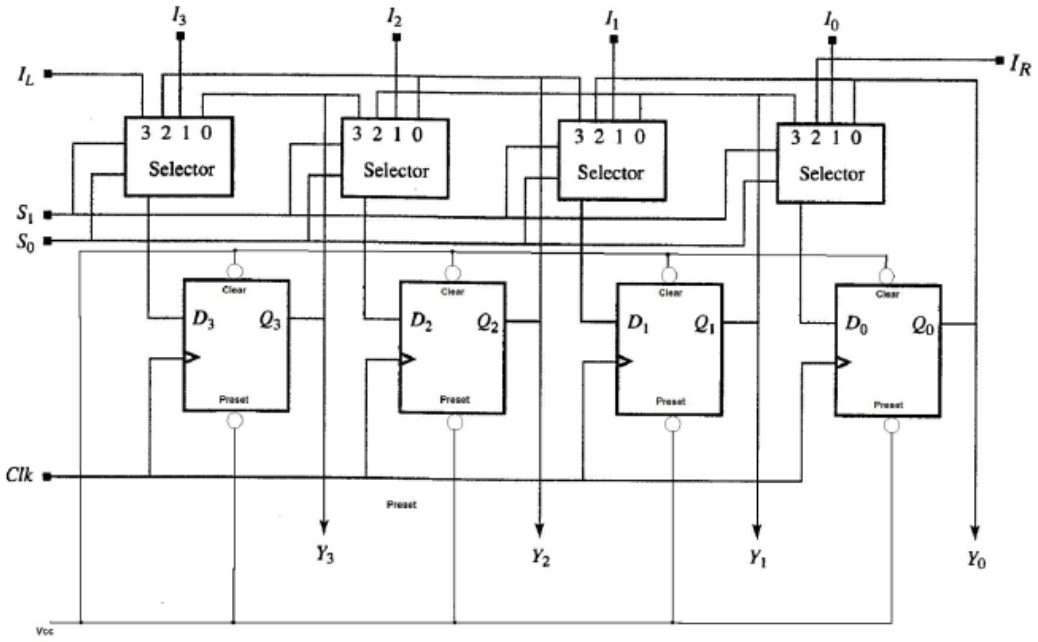


Figure 7. The accumulator. The accumulator is given the option to store its current value, load a new value, or shift its current value left or right by one using an input selector as shown.

Preliminary

Design a four bit ALU with accumulator similar to that of Figure above. Your TA will provide you with the actual specifications of the ALU or you can use those given in the background.

Develop a test plan for testing your ALU both during simulation as well as during hardware verification. Simulation is relatively simple. Supplying inputs for hardware verification may require some ingenuity on your part! Your test plan should check the most important combinations of inputs. Another way to think of this is that you should check the most important paths through the logic circuit. It is usually unreasonable to check all possible combinations of inputs. For example, our 4-bit add function would require (4-bits on A + 4-bits on B + Cin=9 bits) 2^9 =512 combinations! A more

intelligent approach would look only at the most important paths. The adder is made of 4 one-bit adders with 3 inputs: a, b, cin. If you just want to test the functionality of your circuit (should it work), you only need to show that a one-bit adder works and that the connections between the adders are good. To fully test a single one-bit adder would require 2^3 =8 input combinations:

A	*B*	*Cin*
0000	0000	0
0000	0001	0
0001	0000	0
0001	0001	0
0000	0000	1
0000	0001	1
0001	0000	1
0001	0001	1

To test the connections, you need to test the carries (c_1, c_2 , and c_3) between them. You could use:

A	*B*	*Cin*
0000	1111	0
0000	1111	1

Note that if any carry is wrong, it will show up in one of the answers for these two cases. Come up with a similar plan for the other functions of your ALU and show your test plan to your instructor before the lab.

Procedure

1. Draw your design using Quartus II.
2. Perform a functional simulation on your circuit using ModelSim to verify that it is operating correctly.
   The values assigned to each input signal are shown below:

Input signal	*Value*
D1	*1010*
S	*1*
M	*0*
S3	*1*
S2	*0*
I_R	*0*
I_L	*0*
S1	*0*
S0	*1*
Clk	*100 ps/period*

3. Download and verify the design you created and compare its performance to the simulator predicted performance.
   Use Appendix A and perform the pin assignment as the following:

Signal	*Pin*
Cout	*Green LED4*
F3	*Green LED3*
F2	*Green LED2*
F1	*Green LED1*
F0	*Green LED0*
clk	*50 MHz on-board clk*
CLPS	*SW17*
Z	*SW16*
M	*SW15*
S3	*SW14*
S2	*SW13*
S1	*SW12*
S0	*SW11*
I_L	*SW10*
I_R	*SW9*
I3	*SW3*
I2	*SW2*
I1	*SW1*
I0	*SW0*

Questions

1. What logic did you use to generate Cin?

2. Explain your testing plan in the Modelsim. Why did you pick the particular input combinations you did?

3. Was your design "successful"? Why or why not?