Digital Logic
Laboratory Exercise 1

Switches, Lights, and Multiplexers

The purpose of this exercise is to learn how to connect simple input and output devices to an FPGA chip and implement a circuit that uses these devices. We will use the switches on the DE-series boards as inputs to the circuit. We will use light emitting diodes (LEDs) and 7-segment displays as output devices.

Part I

The DE10-Lite, DE0-CV, and DE1-SoC boards provide ten switches and lights, called SW_{9-0} and $LEDR_{9-0}$. Similarly, the DE2-115 provides eighteen switches and lights. The switches can be used to provide inputs, and the lights can be used as output devices. Figure 1 shows a simple Verilog module that uses ten switches and shows their states on the LEDs. Since there are multiple switches and lights it is convenient to represent them as vectors in the Verilog code, as shown. We have used a single assignment statement for all *LEDR* outputs, which is equivalent to the individual assignments:

```
. . .
assign LEDR[2] = SW[2];
assign LEDR[1] = SW[1];
assign LEDR[0] = SW[0];
```

The DE-series boards have hardwired connections between its FPGA chip and the switches and lights. To use the switches and lights it is necessary to include in your Quartus® project the correct pin assignments, which are given in your board's user manual. For example, the DE1-SoC manual specifies that SW_0 is connected to the FPGA pin *AB12* and $LEDR_0$ is connected to pin *V16*. A good way to make the required pin assignments is to import into the Quartus software the pin assignment file for your board, which is provided on the Boards section of FPGAcademic.org web site. The procedure for making pin assignments is described in the tutorial *Quartus Introduction using Verilog Design*, which is also available from FPGAcademic.org.

It is important to realize that the pin assignments in the file are useful only if the pin names that appear in this file are exactly the same as the port names used in your Verilog module. For example, if the pin assignment file uses the names *SW*[0], . . ., *SW*[9] and *LEDR*[0], . . ., *LEDR*[9], then these are the names that must be used for input and output ports in the Verilog code, as we have done in Figure 1.

```
// Module that connects ten switches and lights
module  part1 (SW, LEDR);
    input [9:0] SW;          // slide switches
    output [9:0] LEDR;       // red LEDs

    assign LEDR = SW;
endmodule
```

Figure 1: Verilog code that uses ten switches and lights.

Perform the following steps to implement a circuit corresponding to the code in Figure 1 on the DE-series boards.

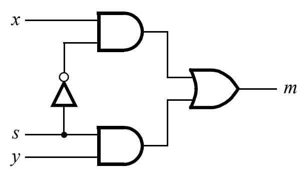
1. Create a new Quartus project for your circuit. Select the target chip that corresponds to your DE-series board. Refer to Table 1 for a list of devices.
2. Create a Verilog module for the code in Figure 1 and include it in your project.
3. Include in your project the required pin assignments for your DE-series board, as discussed above. Compile the project.
4. Download the compiled circuit into the FPGA chip by using the Quartus Programmer tool (the procedure for using the Programmer tool is described in the tutorial *Quartus Introduction*). Test the functionality of the circuit by toggling the switches and observing the LEDs.

Board	**Device Name**
DE10-Lite	MAX® 10 10M50DAF484C6GES
DE0-CV	Cyclone® V 5CEBA4F23C7
DE1-SoC	Cyclone® V SoC 5CSEMA5F31C6
DE2-115	Cyclone® IVE EP4CE115F29C7

Table 1: DE-series FPGA device names

Part II

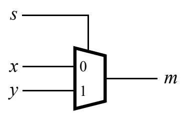
Figure 2a shows a sum-of-products circuit that implements a 2-to-1 *multiplexer* with a select input s. If $s = 0$ the multiplexer's output m is equal to the input x, and if $s = 1$ the output is equal to y. Part b of the figure gives a truth table for this multiplexer, and part c shows its circuit symbol.



a) Circuit

s	m
0	x
1	y

b) Truth table



c) Symbol

Figure 2: A 2-to-1 multiplexer.

The multiplexer can be described by the following Verilog statement:

assign m = (∼s & x) | (s & y);

You are to write a Verilog module that includes four assignment statements like the one shown above to describe the circuit given in Figure 3a. This circuit has two four-bit inputs, X and Y, and produces the four-bit output M. If $s = 0$ then $M = X$, while if $s = 1$ then $M = Y$. We refer to this circuit as a four-bit wide 2-to-1 multiplexer. It has the circuit symbol shown in Figure 3b, in which X, Y, and M are depicted as four-bit wires.

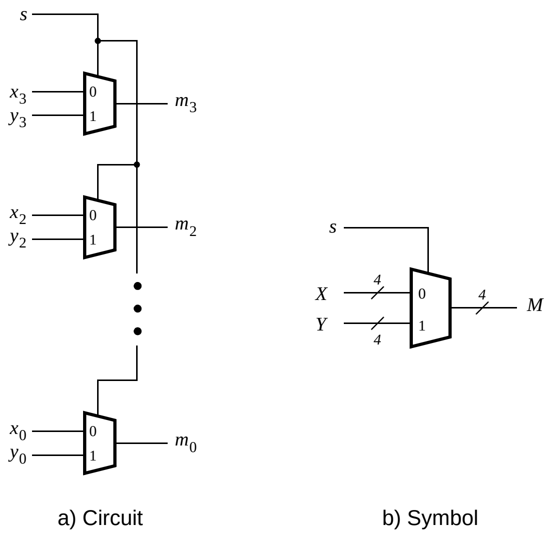


Figure 3: A four-bit wide 2-to-1 multiplexer.

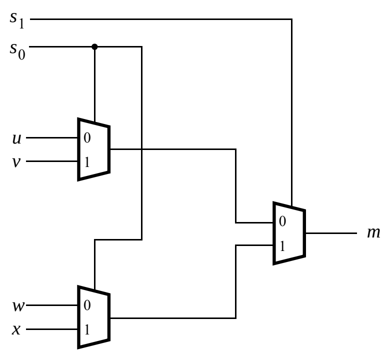
Perform the steps listed below.

1. Create a new Quartus project for your circuit.
2. Include your Verilog file for the four-bit wide 2-to-1 multiplexer in your project. Use switch *SW*$_9$ as the s input, switches *SW*$_{3-0}$ as the X input and *SW*$_{7-4}$ as the Y input. Display the value of the input s on *LEDR*$_9$, connect the output M to *LEDR*$_{3-0}$, and connect the unused LEDR lights to the constant value 0.
3. Include in your project the required pin assignments for your DE-series board. As discussed in Part I, these assignments ensure that the ports of your Verilog code will use the pins on the FPGA chip that are connected to the *SW* switches and *LEDR* lights.
4. Compile the project, and then download the resulting circuit into the FPGA chip. Test the functionality of the four-bit wide 2-to-1 multiplexer by toggling the switches and observing the LEDs.

Part III

In Figure 2 we showed a 2-to-1 multiplexer that selects between the two inputs *x* and *y*. For this part consider a circuit in which the output *m* has to be selected from four inputs u, v, w, and x. Part a of Figure 4 shows how we can build the required 4-to-1 multiplexer by using three 2-to-1 multiplexers. The circuit uses a 2-bit select input $s_1 s_0$ and implements the truth table shown in Figure 4b. A circuit symbol for this multiplexer is given in part c of the figure.

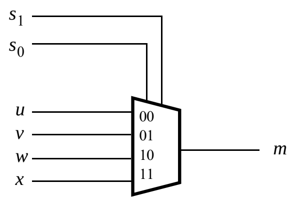
Recall from Figure 3 that a four-bit wide 2-to-1 multiplexer can be built by using four instances of a 2-to-1 multiplexer. Figure 5 applies this concept to define a two-bit wide 4-to-1 multiplexer. It contains two instances of the circuit in Figure 4a.



a) Circuit

s_1 s_0	m
0 0	u
0 1	v
1 0	w
1 1	x

b) Truth table



c) Symbol

Figure 4: A 4-to-1 multiplexer.

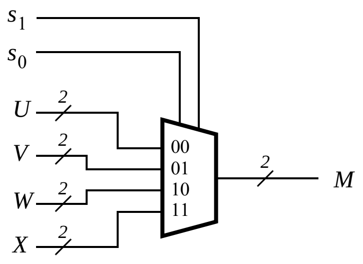


Figure 5: A two-bit wide 4-to-1 multiplexer.

Perform the following steps to implement the two-bit wide 4-to-1 multiplexer.

1. Create a new Quartus project for your circuit.
2. Create a Verilog module for the two-bit wide 4-to-1 multiplexer. Connect its select inputs to switches SW_{9-8}, and use switches SW_{7-0} to provide the four 2-bit inputs U to X. Connect the output M to the red lights $LEDR_{1-0}$.
3. Include in your project the required pin assignments for your DE-series board. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the two-bit wide 4-to-1 multiplexer by toggling the switches and observing the LEDs. Ensure that each of the inputs U to X can be properly selected as the output M.

Part IV

The objective of this part is to display a character on a 7-segment display. The specific character displayed depends on a two-bit input. Figure 6 shows a *7-segment decoder* module that has the two-bit input c_1c_0. This decoder produces seven outputs that are used to display a character on a 7-segment display. Table 2 lists the characters that should be displayed for each valuation of c_1c_0 for your DE-series board. Note that in some cases the 'blank' character is selected for code 11.

The seven segments in the display are identified by the indices 0 to 6 shown in the figure. Each segment is illuminated by driving it to the logic value 0. You are to write a Verilog module that implements logic functions to activate each of the seven segments. Use only simple Verilog **assign** statements in your code to specify each logic function using a Boolean expression.

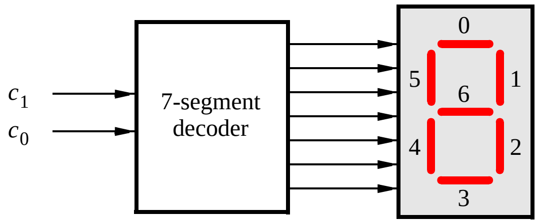


Figure 6: A 7-segment decoder.

c_1c_0	DE10-Lite	DE0-CV	DE1-SoC	DE2-115
00	d	d	d	d
01	E	E	E	E
10	1	0	1	2
11	0			

Table 2: Character codes for the DE-series boards.

Perform the following steps:

1. Create a new Quartus project for your circuit.
2. Create a Verilog module for the 7-segment decoder. Connect the c_1c_0 inputs to switches SW_{1-0}, and connect the outputs of the decoder to the *HEX0* display on your DE-series board. The segments in this display are called $HEX0_0$, $HEX0_1$, ..., $HEX0_6$, corresponding to Figure 6. You should declare the 7-bit port

   output [0:6] HEX0;

   in your Verilog code so that the names of these outputs match the corresponding names in your board's user manual and pin assignment file.
3. After making the required pin assignments, compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling the SW_{1-0} switches and observing the 7-segment display.

Part V

Consider the circuit shown in Figure 7. It uses a two-bit wide 4-to-1 multiplexer to enable the selection of four characters that are displayed on a 7-segment display. Using the 7-segment decoder from Part IV this circuit can display the characters d, E, 0, 1, 2, or 'blank' depending on your DE-series board. The character codes are set according to Table 2 by using the switches SW_{7-0}, and a specific character is selected for display by setting the switches SW_{9-8}.

An outline of the Verilog code that represents this circuit is provided in Figure 8. Note that we have used the circuits from Parts III and IV as subcircuits in this code. You are to extend the code in Figure 8 so that it uses four 7-segment displays rather than just one. You will need to use four instances of each of the subcircuits. The purpose of your circuit is to display any word on the four 7-segment displays that is composed of the characters in Table 2, and be able to rotate this word in a circular fashion across the displays when the switches SW_{9-8} are toggled. As an example, if the displayed word is dE10, then your circuit should produce the output patterns illustrated in Table 3.

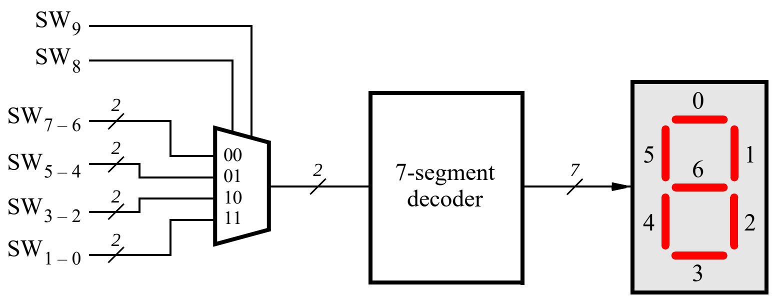


Figure 7: A circuit that can select and display one of four characters.

SW_{9-8}	Characters			
00	d	E	1	0
01	E	1	0	d
10	1	0	d	E
11	0	d	E	1

Table 3: Rotating the word dE10 on four displays.

Perform the following steps.

1. Create a new Quartus project for your circuit.
2. Include your Verilog module in the Quartus project. Connect the switches SW_{9-8} to the select inputs of each of the four instances of the two-bit wide 4-to-1 multiplexers. Also connect SW_{7-0} to each instance of the multiplexers as required to produce the patterns of characters shown in Table 2. Connect the SW switches to the red lights LEDR, and connect the outputs of the four multiplexers to the 7-segment displays *HEX3*, *HEX2*, *HEX1*, and *HEX0*.
3. Include the required pin assignments for your DE-series board for all switches, LEDs, and 7-segment displays. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by setting the proper character codes on the switches SW_{7-0} and then toggling SW_{9-8} to observe the rotation of the characters.

```
module  part5 (SW, LEDR, HEX0);
    input [9:0] SW;                        // slide switches
    output [9:0] LEDR;                     // red lights
    output [0:6] HEX0;                     // 7-seg display

    wire [1:0] M0;

    mux_2bit_4to1 U0 (SW[9:8], SW[7:6], SW[5:4], SW[3:2], SW[1:0], M0);
    char_7seg H0 (M0, HEX0);
    . . .
endmodule

// implements a 2-bit wide 4-to-1 multiplexer
module mux_2bit_3to1 (S, U, V, W, X, M);
    input [1:0] S, U, V, W, X;
    output [1:0] M;
    . . . code not shown

endmodule

// implements a 7-segment decoder for d, E, 1 and 0
module char_7seg (C, Display);
    input [1:0] C;           // input code
    output [0:6] Display;    // output 7-seg code
    . . . code not shown

endmodule
```

Figure 8: Verilog code for the circuit in Figure 7.

.

Part VI

Extend your design from Part V so that is uses all 7-segment displays on your DE-series board. Your circuit needs to display a three- or four-letter word, corresponding to Table 2, using 'blank' characters for unused displays. Implement rotation of this word from right-to-left as indicated in Table 4 and Table 5. To do this, you will need to connect 6-to-1 multiplexers to each of six 7-segment display decoders for the DE10-Lite, DE0-CV and DE1-SoC. Note that for the DE10-Lite you will need to use 3-bit codes for your characters, because five characters are needed when including the 'blank' character (your 7-segment decoder will have to use 3-bit codes, and you will need to use 3-bit wide 6-to-1 multiplexers). For the DE2-115, you will need to connect 8-to-1 multiplexers to each of the eight 7-segment display decoders. You will need to use three select lines for each of the multiplexers: connect the select lines to switches SW_{9-7}. In your Verilog code connect constants to the 6-to-1 (or 8-to-1) multiplexers that select each character, because there are not enough SW switches.

SW_{9-7}	Character pattern					
000			d	E	1	0
001		d	E	1	0	
010	d	E	1	0		
011	E	1	0			d
100	1	0			d	E
101	0			d	E	1

Table 4: Rotating the word dE10 on six displays.

SW_{9-7}	Character pattern							
000						d	E	2
001					d	E	2	
010				d	E	2		
011			d	E	2			
100		d	E	2				
101	d	E	2					
110	E	2						d
111	2						d	E

Table 5: Rotating the word dE2 on eight displays.

Perform the following steps:

1. Create a new Quartus project for your circuit.
2. Include your Verilog module in the Quartus project. Connect the switches SW_{9-7} to the select inputs of each instance of the multiplexers in your circuit. Connect constants in your Verilog code to the multiplexers as required to produce the patterns of characters shown in Table 4 or Table 5 depending on your DE-series board. Connect the outputs of your multiplexers to the 7-segment displays *HEX5*, . . ., *HEX0* of the DE10-Lite, DE0-CV and DE1-SoC or *HEX7*, . . ., *HEX0* for the DE2-115.
3. Include the required pin assignments for your DE-series board for all switches, LEDs, and 7-segment displays. Compile the project.
4. Download the compiled circuit into the FPGA chip. Test the functionality of the circuit by toggling SW_{9-7} to observe the rotation of the characters.

Digital Logic
Laboratory Exercise 2

Numbers and Displays

This is an exercise in designing combinational circuits that can perform binary-to-decimal number conversion and binary-coded-decimal (BCD) addition.

Part I

We wish to display on the 7-segment displays *HEX1* and *HEX0* the values set by the switches SW_{7-0}. Let the values denoted by SW_{7-4} and SW_{3-0} be displayed on *HEX1* and *HEX0*, respectively. Your circuit should be able to display the digits from 0 to 9, and should treat the valuations 1010 to 1111 as don't-cares.

1. Create a new project which will be used to implement the desired circuit on your Intel® FPGA DE-series board. The intent of this exercise is to manually derive the logic functions needed for the 7-segment displays. Therefore, you should use only simple Verilog **assign** statements in your code and specify each logic function as a Boolean expression.
2. Write a Verilog file that provides the necessary functionality. Include this file in your project and assign the pins on the FPGA to connect to the switches and 7-segment displays. Make sure to include the necessary pin assignments.
3. Compile the project and download the compiled circuit into the FPGA chip.
4. Test the functionality of your design by toggling the switches and observing the displays.

Part II

You are to design a circuit that converts a four-bit binary number $V = v_3v_2v_1v_0$ into its two-digit decimal equivalent $D = d_1d_0$. Table 1 shows the required output values. A partial design of this circuit is given in Figure 1. It includes a comparator that checks when the value of V is greater than 9, and uses the output of this comparator in the control of the 7-segment displays. You are to complete the design of this circuit.

$v_3v_2v_1v_0$	d_1	d_0
0000	0	0
0001	0	1
0010	0	2
...	...	...
1001	0	9
1010	1	0
1011	1	1
1100	1	2
1101	1	3
1110	1	4
1111	1	5

Table 1: Binary-to-decimal conversion values.

The output z for the comparator circuit can be specified using a single Boolean expression, with the four inputs V_{3-0}. Design this Boolean expression by making a truth table that shows the valuations of the inputs V_{3-0} for which z has to be 1.

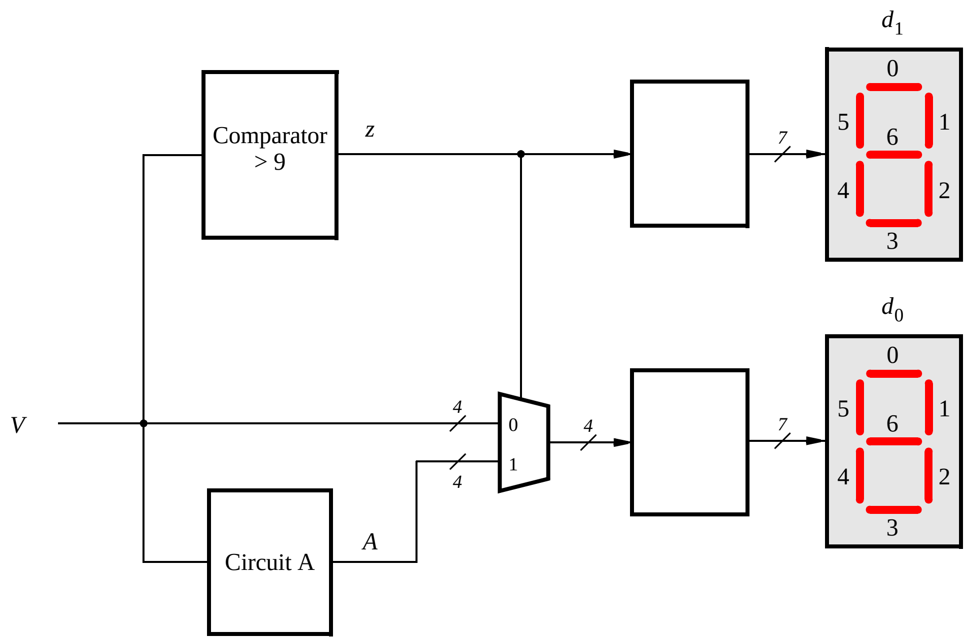


Figure 1: Partial design of the binary-to-decimal conversion circuit.

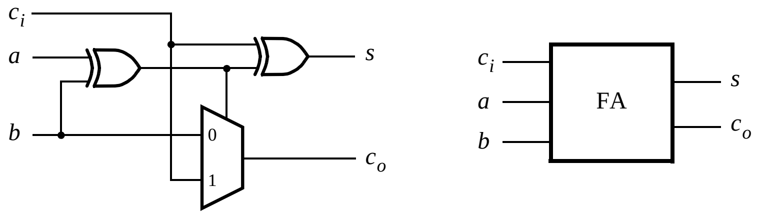
Notice that the circuit in Figure 1 includes a 4-bit wide 2-to-1 multiplexer (a similar multiplexer was described as part of Laboratory Exercise 1). The purpose of this multiplexer is to drive digit d_0 with the value of V when $z = 0$, and the value of A when $z = 1$. To design circuit A consider the following. For the input values $V \leq 9$, the circuit A does not matter, because the multiplexer in Figure 1 just selects V in these cases. But for the input values $V > 9$, the multiplexer will select A. Thus, A has to provide output values that properly implement Table 1 when $V > 9$. You need to design circuit A so that the input $V = 1010$ gives an output $A = 0000$, the input $V = 1011$ gives the output $A = 0001$, . . ., and the input $V = 1111$ gives the output $A = 0101$. Design circuit A by making a truth table with the inputs V_{3-0} and the outputs A_{3-0}.

Perform the following steps:

1. Write Verilog code to implement your design. The code should have the 4-bit input SW_{3-0}, which should be used to provide the binary number V, and the two 7-bit outputs *HEX1* and *HEX0*, to show the values of decimal digits d_1 and d_0. The intent of this exercise is to use simple Verilog **assign** statements to specify the required logic functions using Boolean expressions. Your Verilog code should not include any **if-else**, **case**, or similar statements.
2. Make a Quartus® project for your Verilog module.
3. Compile the circuit and use functional simulation to verify the correct operation of your comparator, multiplexers, and circuit A.
4. Download the circuit into an FPGA board. Test the circuit by trying all possible values of V and observing the output displays.

Part III

Figure 2a shows a circuit for a *full adder*, which has the inputs a, b, and c_i, and produces the outputs s and c_o. Parts b and c of the figure show a circuit symbol and truth table for the full adder, which produces the two-bit binary sum $c_o s = a + b + c_i$. Figure 2d shows how four instances of this full adder module can be used to design a circuit that adds two four-bit numbers. This type of circuit is usually called a *ripple-carry* adder, because of the way that the carry signals are passed from one full adder to the next. Write Verilog code that implements this circuit, as described below.

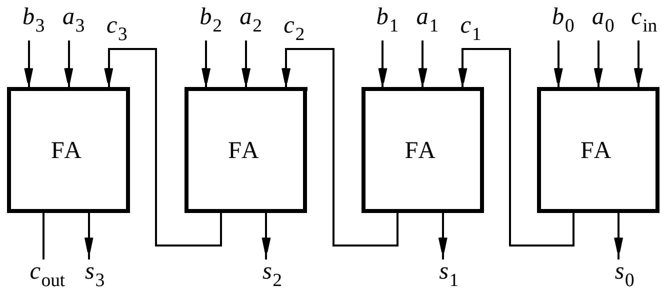


a) Full adder circuit

b) Full adder symbol

b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

c) Full adder truth table



d) Four-bit ripple-carry adder circuit

Figure 2: A ripple-carry adder circuit.

1. Create a new Quartus project for the adder circuit. Write a Verilog module for the full adder subcircuit and write a top-level Verilog module that instantiates four instances of this full adder.
2. Use switches SW_{7-4} and SW_{3-0} to represent the inputs A and B, respectively. Use SW_8 for the carry-in c_{in} of the adder. Connect the outputs of the adder, c_{out} and S, to the red lights LEDR.
3. Include the necessary pin assignments for your DE-series board, compile the circuit, and download it into the FPGA chip.
4. Test your circuit by trying different values for numbers A, B, and c_{in}.

Part IV

In part II we discussed the conversion of binary numbers into decimal digits. For this part you are to design a circuit that has two decimal digits, X and Y, as inputs. Each decimal digit is represented as a 4-bit number. In technical literature this is referred to as the *binary coded decimal* (BCD) representation.
You are to design a circuit that adds the two BCD digits. The inputs to your circuit are the numbers X and Y, plus a carry-in, c_{in}. When these inputs are added, the result will be a 5-bit binary number. But this result is to be displayed on 7-segment displays as a two-digit BCD sum S_1S_0. For a sum equal to zero you would display $S_1S_0 = 00$, for a sum of one $S_1S_0 = 01$, for nine $S_1S_0 = 09$, for ten $S_1S_0 = 10$, and so on. Note that the

inputs X and Y are assumed to be decimal digits, which means that the largest sum that needs to be handled by this circuit is $S_1S_0 = 9 + 9 + 1 = 19$.
Perform the steps given below.

1. Create a new Quartus project for your BCD adder. You should use the four-bit adder circuit from part III to produce a four-bit sum and carry-out for the operation X + Y.

   A good way to work out the design of your circuit is to first make it handle only sums $(X + Y) \leq 15$. With these values, your circuit from Part II can be used to convert the 4-bit sum into the two decimal digits S_1S_0. Then, once this is working, modify your design to handle values of $15 < (X + Y) \leq 19$. One way to do this is to still use your circuit from Part II, but to modify its outputs before attaching them to the 7-segment display to make the necessary adjustments when the sum from the adder exceeds 15.

   Write your Verilog code using simple **assign** statements to specify the required logic functions–do not use other types of Verilog statements such as **if-else** or **case** statements for this part of the exercise.

2. Use switches SW_{7-4} and SW_{3-0} for the inputs X and Y, respectively, and use SW_8 for the carry-in. Connect the four-bit sum and carry-out produced by the operation X + Y to the red lights LEDR. Display the BCD values of X and Y on the 7-segment displays *HEX5* and *HEX3*, and display the result S_1S_0 on *HEX1* and *HEX0*.

3. Since your circuit handles only BCD digits, check for the cases when the input X or Y is greater than nine. If this occurs, indicate an error by turning on the red light *LEDR*$_9$.

4. Include the necessary pin assignments for your DE-series board, compile the circuit, and download it into the FPGA chip.

5. Test your circuit by trying different values for numbers X, Y, and c_{in}.

Part V

In part IV you created Verilog code for a BCD adder. A different approach for describing the adder in Verilog code is to specify an algorithm like the one represented by the following pseudo-code:

```
1   $T_0 = A + B + c_0$
2   if ($T_0 > 9$) then
3       $Z_0 = 10$;
4       $c_1 = 1$;
5   else
6       $Z_0 = 0$;
7       $c_1 = 0$;
8   end if
9   $S_0 = T_0 - Z_0$
10  $S_1 = c_1$
```

It is reasonably straightforward to see what circuit could be used to implement this pseudo-code. Lines 1 and 9 represent adders, lines 2-8 correspond to multiplexers, and testing for the condition $T_0 > 9$ requires comparators. You are to write Verilog code that corresponds to this pseudo-code. Note that you can perform addition operations in your Verilog code instead of the subtraction shown in line 9. The intent of this part of the exercise is to examine the effects of relying more on the Verilog compiler to design the circuit by using **if-else** statements along with the Verilog $>$ and $+$ operators. Perform the following steps:

1. Create a new Quartus project for your Verilog code. Use switches SW_{7-4} and SW_{3-0} for the inputs A and B, respectively, and use SW_8 for the carry-in. The value of A should be displayed on the 7-segment display *HEX5*, while B should be on *HEX3*. Display the BCD sum, S_1S_0, on *HEX1* and *HEX0*.

2. Use the Quartus RTL Viewer tool to examine the circuit produced by compiling your Verilog code. Compare the circuit to the one you designed in Part IV.
3. Download your circuit onto your DE-series board and test it by trying different values for numbers A and B.

Part VI

Design a combinational circuit that converts a 6-bit binary number into a 2-digit decimal number represented in the BCD form. Use switches SW_{5-0} to input the binary number and 7-segment displays *HEX1* and *HEX0* to display the decimal number. Implement your circuit on a DE-series board and demonstrate its functionality.

Digital Logic
Laboratory Exercise 3

Latches, Flip-flops, and Registers

The purpose of this exercise is to investigate latches, flip-flops, and registers.

Part I

Intel® FPGAs include flip-flops that are available for implementing a user's circuit. We will show how to make use of these flip-flops in Part IV of this exercise. But first we will show how storage elements can be created in an FPGA without using its dedicated flip-flops.
Figure 1 depicts a gated RS latch circuit. Two styles of Verilog code that can be used to describe this circuit are given in Figures 2 and 3. Figure 2 specifies the latch by instantiating logic gates, and Figure 3 uses logic expressions to create the same circuit. If this latch is implemented in an FPGA that has 4-input lookup tables (LUTs), then only one lookup table is needed, as shown in Figure 4a.

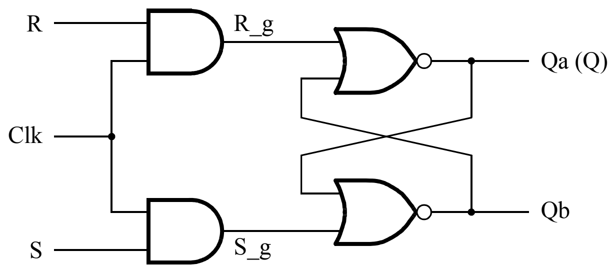


Figure 1: A gated RS latch circuit.

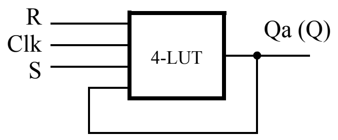
```
// A gated RS latch
module part1 (Clk, R, S, Q);
   input Clk, R, S;
   output Q;

   wire R_g, S_g, Qa, Qb /* synthesis keep */ ;

   and (R_g, R, Clk);
   and (S_g, S, Clk);
   nor (Qa, R_g, Qb);
   nor (Qb, S_g, Qa);

   assign Q = Qa;

endmodule
```

Figure 2: Specifying an RS latch by instantiating logic gates.

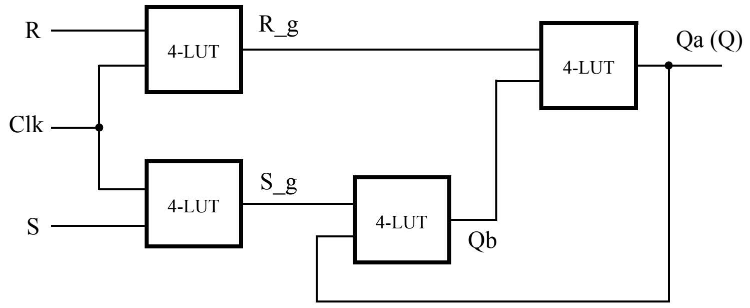
```
// A gated RS latch
module part1 (Clk, R, S, Q);
    input Clk, R, S;
    output Q;

    wire R_g, S_g, Qa, Qb /* synthesis keep */ ;

    assign R_g = R & Clk;
    assign S_g = S & Clk;
    assign Qa = ~(R_g | Qb);
    assign Qb = ~(S_g | Qa);

    assign Q = Qa;

endmodule
```

Figure 3: Specifying an RS latch by using Boolean expressions.

Although the latch can be correctly realized in one 4-input LUT, this implementation does not allow its internal signals, such as R_g and S_g, to be observed, because they are not provided as outputs from the LUT. To preserve these internal signals in the implemented circuit, it is necessary to include a *compiler directive* in the code. In Figures 2 and 3 the directive /* synthesis keep */ is included to instruct the Quartus® compiler to use separate logic elements for each of the signals R_g, S_g, Qa, and Qb. Compiling the code produces the circuit with four 4-LUTs depicted in Figure 4b.



(a) Using one 4-input lookup table for the RS latch.



(b) Using four 4-input lookup tables for the RS latch.

Figure 4: Implementation of the RS latch from Figure 1.

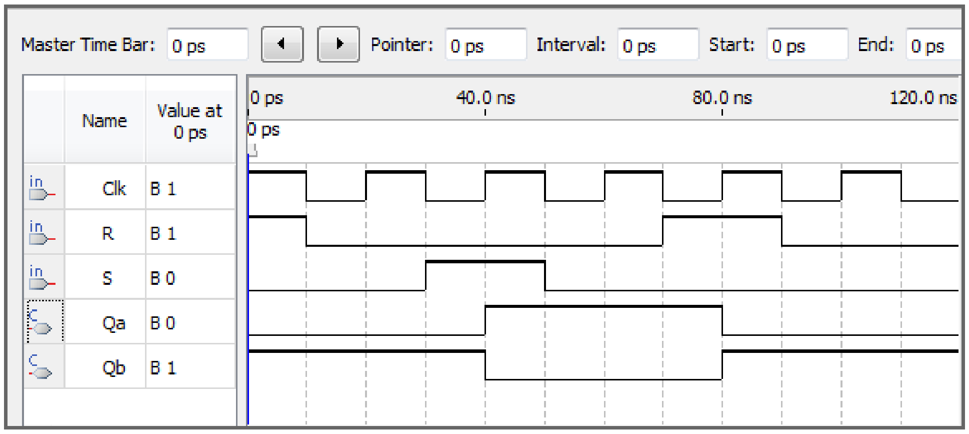
Create a Quartus project for the RS latch circuit as follows:

1. Create a new Quartus project for your DE-series board.

2. Generate a Verilog file for the RS latch. Use the code in either Figure 2 or Figure 3 (both versions of the code should produce the same circuit) and include it in the project.

3. Compile the code. Use the Quartus RTL Viewer tool to examine the gate-level circuit produced from the code, and use the Technology Map Viewer tool to verify that the latch is implemented as shown in Figure 4*b*.

4. Simulate the behavior of your Verilog code by using the simulation feature provided in the Modelsim software. Use the testbench provided in the laboratory materials to drive the signals for your simulation. The procedure for using Modelsim for simulation is described in the tutorial *Using the ModelSim-Intel FPGA Simulator*. An example of a vector waveform file is displayed in Figure 5. The waveforms in the figure begin by setting $Clk = 1$ and $R = 1$, which allows the simulation tool to initialize all of the signals inside of the latch to known values.



Name	Value at 0 ps
Clk	B 1
R	B 1
S	B 0
Qa	B 0
Qb	B 1

Figure 5: Simulation waveforms for the RS latch.

Part II

Figure 6 shows the circuit for a gated D latch.

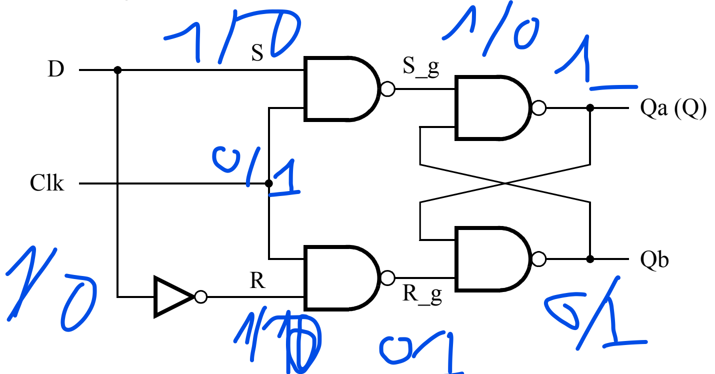


Figure 6: Circuit for a gated D latch.

Perform the following steps:

1. Create a new Quartus project. Generate a Verilog file using the style of code in Figure 3 for the gated D latch. Use the /* synthesis keep */ directive to ensure that separate logic elements are used to implement the signals R, S_g, R_g, Qa, and Qb.

2. Compile your project and then use the Technology Map Viewer tool to examine the implemented circuit.

3. Verify that the latch works properly for all input conditions by using functional simulation. Examine the timing characteristics of the circuit by using timing simulation.
4. Create a new Quartus project which will be used for implementation of the gated D latch on your DE-series board. This project should consist of a top-level module that contains the appropriate input and output ports (pins) for your board. Instantiate your latch in this top-level module. Use switch SW_0 to drive the D input of the latch, and use SW_1 as the *Clk* input. Connect the Q output to $LEDR_0$.
5. Include the required pin assignments and then compile your project and download the compiled circuit onto your DE-series board.
6. Test the functionality of your circuit by toggling the D and *Clk* switches and observing the Q output.

Part III

Figure 7 shows the circuit for a master-slave D flip-flop.

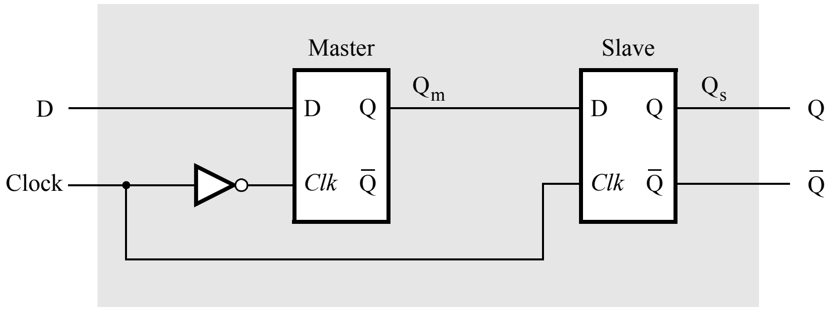


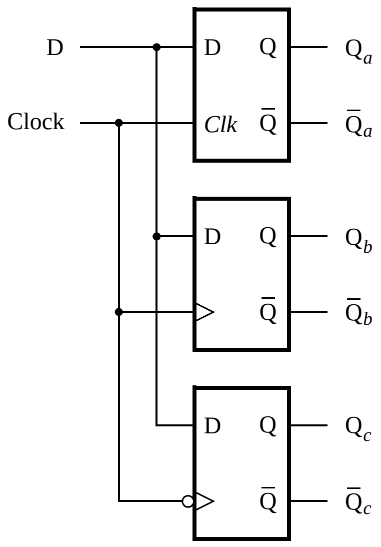
Figure 7: Circuit for a master-slave D flip-flop.

Perform the following:

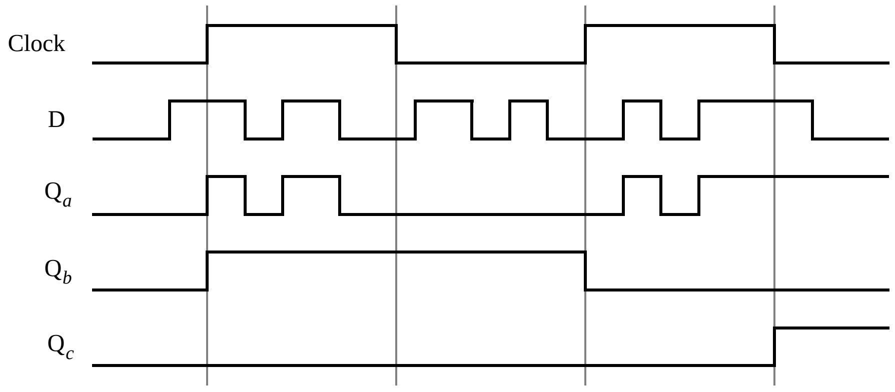
1. Create a new Quartus project. Generate a Verilog file that instantiates two copies of your gated D latch module from Part II to implement the master-slave flip-flop.
2. Include in your project the appropriate input and output ports for your DE-series board. Use switch SW_0 to drive the D input of the flip-flop, and use SW_1 as the *Clock* input. Connect the Q output to $LEDR_0$.
3. Include the required pin assignments and then compile your project.
4. Use the Technology Map Viewer to examine the D flip-flop circuit, and use simulation to verify its correct operation.
5. Download the circuit onto your DE-series board and test its functionality by toggling the D and *Clock* switches and observing the Q output.

Part IV

Figure 8 shows a circuit with three different storage elements: a gated D latch, a positive-edge triggered D flip-flop, and a negative-edge triggered D flip-flop.



(a) Circuit



(b) Timing diagram

Figure 8: Circuit and waveforms for Part IV.

Implement and simulate this circuit using the Quartus software as follows:

1. Create a new Quartus project.
2. Write a Verilog file that instantiates the three storage elements. For this part you should no longer use the /* synthesis keep */ directive from Parts I to III. Figure 9 gives a behavioral style of Verilog code that specifies the gated D latch in Figure 6. This latch can be implemented in one 4-input lookup table. Use a similar style of code to specify the flip-flops in Figure 8.
3. Compile your code and use the Technology Map Viewer to examine the implemented circuit. Verify that the latch uses one lookup table and that the flip-flops are implemented using the flip-flops provided in the target FPGA.

4. Use Modelsim to simulate the circuit you created. Use the included testbench file to specify the inputs D and *Clock* as indicated in Figure 8. Make sure that the testbench correctly instantiates the module you created and run the simulation to observe the different behavior of the three storage elements.

```
module D_latch  (D, Clk, Q);
   input D, Clk;
   output reg Q;

   always @ (D, Clk)
      if (Clk)
         Q = D;
endmodule
```

Figure 9: A behavioral style of Verilog code that specifies a gated D latch.

Part V

We wish to display the hexadecimal value of an 8-bit number A on the two 7-segment displays $HEX3-2$. We also wish to display the hex value of an 8-bit number B on the two 7-segment displays $HEX1-0$. The values of A and B are inputs to the circuit which are provided by means of switches SW_{7-0}. To input the values of A and B, first set the switches to the desired value of A, store these switch values in a register, and then change the switches to the desired value of B. Finally, use an adder to generate the arithmetic sum $S = A + B$, and display this sum on the 7-segment displays $HEX5-4$. Show the carry-out produced by the adder on LEDR[0].

1. Create a new Quartus project which will be used to implement the desired circuit on your DE-series board.
2. Write a Verilog file that provides the necessary functionality. Use KEY_0 as an active-low asynchronous reset, and use KEY_1 as a clock input.
3. Include the necessary pin assignments for the pushbutton switches and 7-segment displays, and then compile the circuit.
4. Download the circuit onto your DE-series board and test its functionality by toggling the switches and observing the output displays.

Digital Logic
Laboratory Exercise 4

Counters

The purpose of this exercise is to build and use counters. The designed circuits are to be implemented on an Intel® FPGA DE10-Lite, DE0-CV, DE1-SoC, or DE2-115 Board.

Students are expected to have a basic understanding of counters and sufficient familiarity with the Verilog hardware description language to implement various types of latches and flip-flops.

Part I

Consider the circuit in Figure 1. It is a 4-bit synchronous counter which uses four T-type flip-flops. The counter increments its value on each positive edge of the clock signal if the *Enable* signal is high. The counter is reset to 0 on the next positive clock edge if the synchronous *Clear* input is low. You are to implement an 8-bit counter of this type.

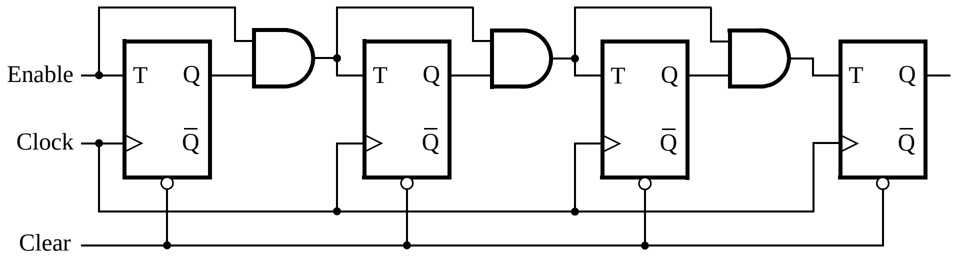


Figure 1: A 4-bit counter.

1. Write a Verilog file that defines an 8-bit counter by using the structure depicted in Figure 1. Your code should include a T flip-flop module that is instantiated eight times to create the counter. Compile the circuit. How many logic elements (LEs) are used to implement your circuit?
2. Simulate your circuit to verify its correctness.
3. Augment your Verilog file to use the pushbutton *KEY*$_0$ as the *Clock* input and switches SW_1 and SW_0 as *Enable* and *Clear* inputs, and 7-segment displays *HEX1-0* to display the hexadecimal count as your circuit operates. Make the necessary pin assignments needed to implement the circuit on your DE-series board, and compile the circuit.
4. Download your circuit into the FPGA chip and test its functionality by operating the switches.
5. Implement a four-bit version of your circuit and use the Quartus® RTL Viewer to see how the Quartus software synthesized the circuit. What are the differences in comparison with Figure 1?

Part II

Another way to specify a counter is by using a register and adding 1 to its value. This can be accomplished using the following Verilog statement:

$$Q <= Q + 1;$$

Compile a 16-bit version of this counter and determine the number of LEs needed. Use the RTL Viewer to see the structure of this implementation and comment on the differences with the design from Part I. Implement the counter on your DE-series board, using the displays *HEX3-0* to show the counter value.

Part III

Design and implement a circuit that successively flashes digits 0 through 9 on the 7-segment display *HEX0*. Each digit should be displayed for about one second. Use a counter to determine the one-second intervals. The counter should be incremented by the 50-MHz clock signal provided on the DE-series boards. Do not derive any other clock signals in your design–make sure that all flip-flops in your circuit are clocked directly by the 50-MHz clock signal. A partial design of the required circuit is shown in Figure 2. The figure shows how a large bit-width counter can be used to produce an enable signal for a smaller counter. The rate at which the smaller counter increments can be controlled by choosing an appropriate number of bits in the larger counter.

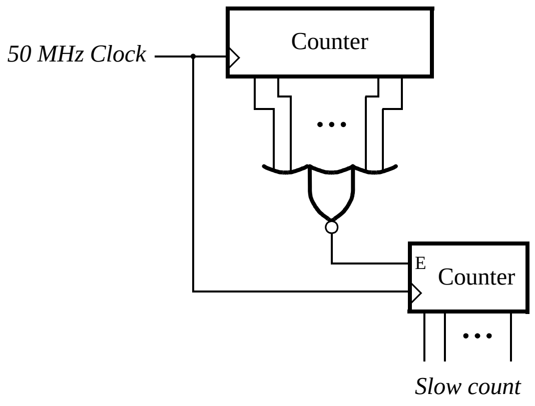


Figure 2: Making a slow counter.

Part IV

Design and implement a circuit that displays a word on four 7-segment displays $HEX3-0$. The word to be displayed for your DE-series board is given in Table 1. Make the letters rotate from right to left in intervals of about one second. The rotating pattern for the DE10-Lite is given in Table 2. If you are using the DE0-CV, DE1-SoC, or DE2-115, use the word given in Table 1. There are many ways to design the required circuit. One solution is to re-use the Verilog code designed in Laboratory Exercise 1, Part V. Using that code, the main change needed is to replace the two switches that are used to select the characters being rotated on the displays with a 2-bit counter that increments at one-second intervals.

Board	Word
DE10-Lite	dE10
DE0-CV	dE0
DE1-SoC	dE1
DE2-115	dE2

Table 1: DE-series boards and corresponding word to display

Count	Characters			
00	d	E	1	0
01	E	1	0	d
10	1	0	d	E
11	0	d	E	1

Table 2: Rotating the word dE10 on four displays.

Part V

Augment your circuit from Part IV so that it can rotate the word over all of the 7-segment displays on your DE-series board. The shifting pattern for the DE10-Lite is shown in Table 3.

Count	Character pattern					
000			d	E	1	0
001		d	E	1	0	
010	d	E	1	0		
011	E	1	0			d
100	1	0			d	E
101	0			d	E	1

Table 3: Rotating the word dE10 on six displays.

Digital Logic
Laboratory Exercise 5

Timers and Real-time Clock

The purpose of this exercise is to study the use of clocks in timed circuits. The designed circuits are to be implemented on an Intel® FPGA DE10-Lite, DE0-CV, DE1-SoC, or DE2-115 board.

Background

In the Verilog hardware description language we can describe a variable-size counter by using a parameter declaration. An example of an *n*-bit counter is shown in Figure 1.

```
module  counter (Clock, Reset_n, Q);
    parameter n = 4;

    input Clock, Reset_n;
    output [n-1:0] Q;
    reg [n-1:0] Q;

    always @(posedge Clock or negedge Reset_n)
    begin
        if (!Reset_n)
            Q <= 1'd0;
        else
            Q <= Q + 1'b1;
    end
endmodule
```

Figure 1: A Verilog description of an *n*-bit counter.

The parameter *n* specifies the number of bits in the counter. When instantiating this counter in another Verilog module, a particular value of the parameter *n* can be specified by using a **defparam** statement. For example, an 8-bit counter can be specified as:

```
counter eight_bit (Clock, Reset_n, Q);
    defparam eight_bit.n = 8;
```

By using parameters we can instantiate counters of different sizes in a logic circuit, without having to create a new module for each counter.

Part I

Create a modulo-k counter by modifying the design of an 8-bit counter to contain an additional parameter. The counter should count from 0 to $k-1$. When the counter reaches the value $k-1$, then the next counter value should be 0. Include an output from the counter called *rollover* and set this output to 1 in the clock cycle where the count value is equal to $k-1$.
Perform the following steps:

1. Create a new Quartus® project which will be used to implement the desired circuit on your DE-series board.

2. Write a Verilog file that specifies the circuit for $k = 20$, and an appropriate value of n. Your circuit should use pushbutton KEY_0 as an asynchronous reset and KEY_1 as a manual clock input. The contents of the counter should be displayed on the red lights *LEDR*. Also display the *rollover* signal on one of the LEDR lights.
3. Include the Verilog file in your project and compile the circuit.
4. Simulate the designed circuit to verify its functionality.
5. Make the necessary pin assignments needed to implement the circuit on your DE-series board, and compile the circuit.
6. Verify that your circuit works correctly by observing the lights.

Part II

Using your modulo-counter from Part I as a subcircuit, implement a 3-digit BCD counter (hint: use multiple counters, not just one). Display the contents of the counter on the 7-segment displays, *HEX2−0*. Connect all of the counters in your circuit to the 50-MHz clock signal on your DE-series board, and make the BCD counter increment at one-second intervals. Use the pushbutton switch KEY_0 to reset the BCD counter to 0.

Part III

Design and implement a circuit on your DE-series board that acts as a real-time clock. It should display the minutes (from 0 to 59) on $HEX5 - 4$, the seconds (from 0 to 59) on $HEX3 - 2$, and hundredths of a second (from 0 to 99) on $HEX1 - 0$. Use the switches SW_{7-0} to preset the minute part of the time displayed by the clock when KEY_1 is pressed. Stop the clock whenever KEY_0 is being pressed and continue the clock when KEY_0 is released.

Part IV

An early method of telegraph communication was based on the Morse code. This code uses patterns of short and long pulses to represent a message. Each letter is represented as a sequence of dots (a short pulse), and dashes (a long pulse). For example, the first eight letters of the alphabet have the following representation:

A	● —
B	— ● ● ●
C	— ● — ●
D	— ● ●
E	●
F	● ● — ●
G	— — ●
H	● ● ● ●

Design and implement a circuit that takes as input one of the first eight letters of the alphabet and displays the Morse code for it on a red LED. Your circuit should use switches SW_{2-0} and pushbuttons KEY_{1-0} as inputs. When a user presses KEY_1, the circuit should display the Morse code for a letter specified by SW_{2-0} (000 for A, 001 for B, etc.), using 0.5-second pulses to represent dots, and 1.5-second pulses to represent dashes. Pushbutton KEY_0 should function as an asynchronous reset. A high-level schematic diagram of the circuit is shown in Figure 2.

Hint: Use a counter to generate 0.5-second pulses, and another counter to keep the $LEDR_0$ light on for either 0.5 or 1.5 seconds.

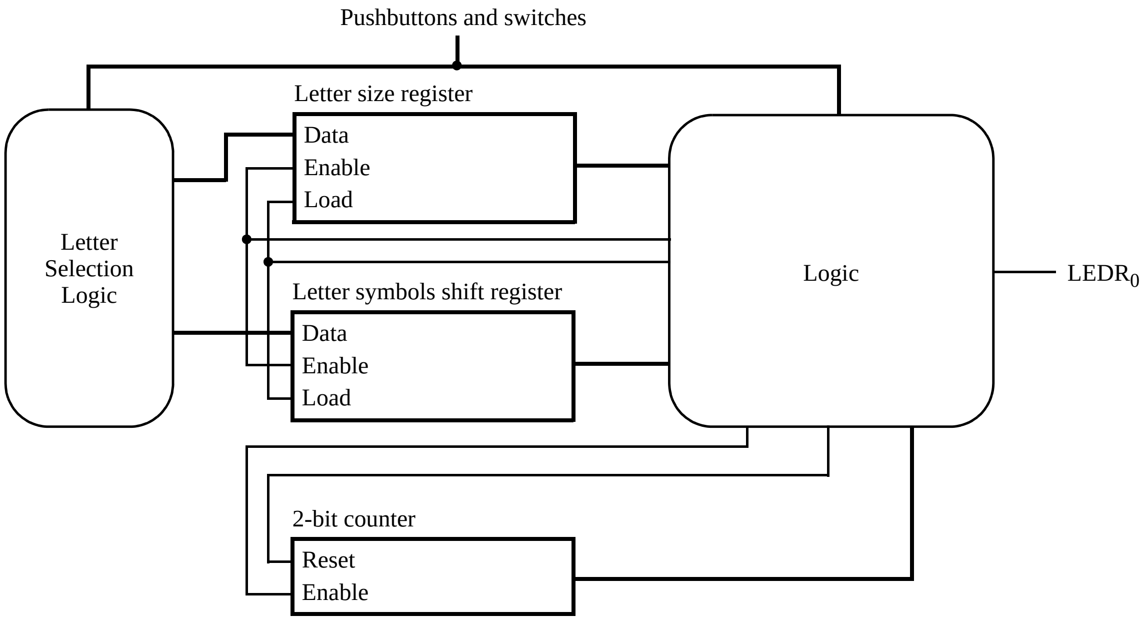


Figure 2: High-level schematic diagram of the circuit for part IV.

Digital Logic
Laboratory Exercise 6

Adders, Subtractors, and Multipliers

The purpose of this exercise is to examine arithmetic circuits that add, subtract, and multiply numbers. Each circuit will be described in Verilog and implemented on an Intel® FPGA DE10-Lite, DE0-CV, DE1-SoC, or DE2-115 board.

Part I

Consider again the four-bit ripple-carry adder circuit used in lab exercise 2; its diagram is reproduced in Figure 1.

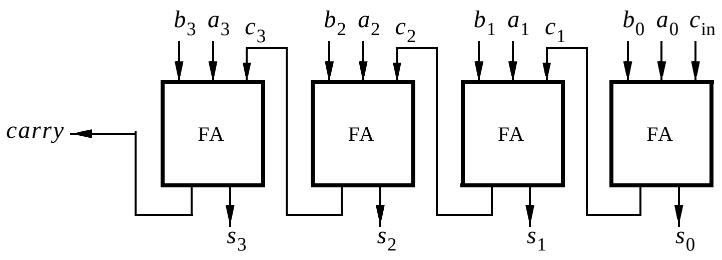


Figure 1: A four-bit ripple carry adder.

This circuit can be implemented using a '+' sign in Verilog. For example, the following code fragment adds *n*-bit numbers A and B to produce outputs sum and $carry$:

```
wire [n-1:0] sum;
wire carry;
. . .
assign {carry, sum} = A + B;
```

Use this construct to implement a circuit shown in Figure 2. This circuit, which is often called an *accumulator*, is used to add the value of an input A to itself repeatedly. The circuit includes a carry out from the adder, as well as an *overflow* output signal. If the input A is considered as a 2's-complement number, then *overflow* should be set to 1 in the case where the output *sum* produced does not represent a correct 2's-complement result.

Perform the following steps:

1. Create a new Quartus® project. Write Verilog code that describes the circuit in Figure 2.

2. Connect input A to switches SW_{7-0}, use KEY_0 as an active-low asynchronous reset, and use KEY_1 as a manual clock input. The sum from the adder should be displayed on the red lights $LEDR_{7-0}$, the registered carry signal should be displayed on $LEDR_8$, and the registered *overflow* signal should be displayed on $LEDR_9$. Show the registered values of *A* and *S* as hexadecimal numbers on the 7-segment displays HEX3 − 2 and HEX1 − 0.

3. Make the necessary pin assignments needed to implement the circuit on your DE-series board, and compile the circuit.

4. Use timing simulation to verify the correct operation of the circuit. Once the simulation works properly, download the circuit onto your DE-series board and test it by using different values of A. Be sure to check that the *overflow* output works correctly.

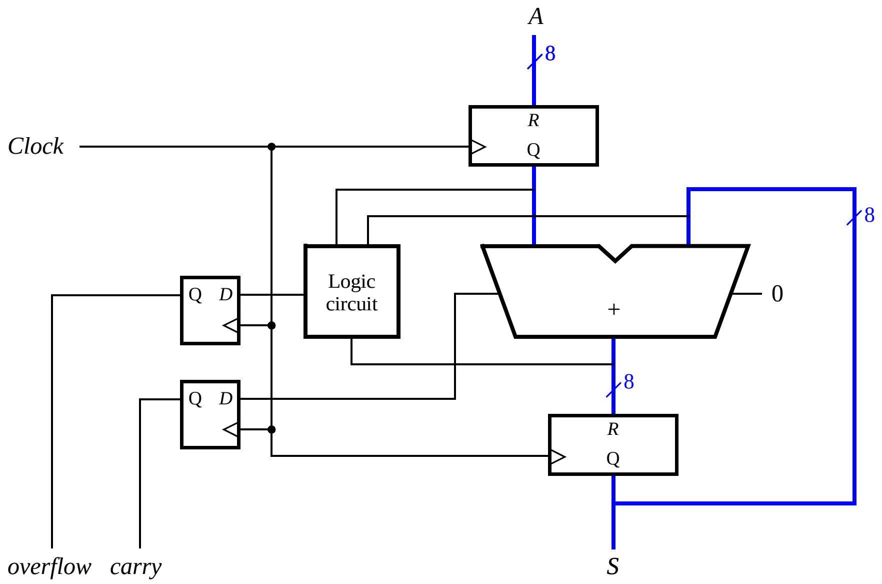


Figure 2: An eight-bit accumulator circuit.

Part II

Extend the circuit from Part I to be able to both add and subtract numbers. To do so, introduce an *add_sub* input to your circuit. When *add_sub* is 1, your circuit should subtract A from S, and when *add_sub* is 0 your circuit should add A to S as in Part I.

Part III

Figure 3a gives an example of paper-and-pencil multiplication $P = A \times B$, where $A = 11$ and $B = 12$.

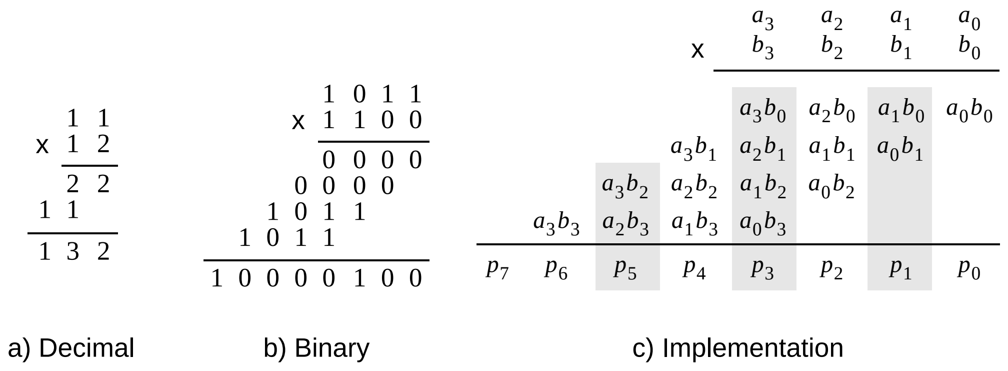


Figure 3: Multiplication of binary numbers.

We compute $P = A \times B$ as an addition of summands. The first summand is equal to A times the ones digit of B. The second summand is A times the tens digit of B, shifted one position to the left. We add the two summands to form the product $P = 132$.

Part b of the figure shows the same example using four-bit binary numbers. To compute $P = A \times B$, we first form summands by multiplying A by each digit of B. Since each digit of B is either 1 or 0, the summands are either shifted versions of A or 0000. Figure 3c shows how each summand can be formed by using the Boolean AND operation of A with the appropriate bit of B.

A four-bit circuit that implements $P = A \times B$ is illustrated in Figure 4. Because of its regular structure, this type of multiplier circuit is called an *array multiplier*. The shaded areas correspond to the shaded columns in Figure 3c. In each row of the multiplier AND gates are used to produce the summands, and full adder modules are used to generate the required sums.

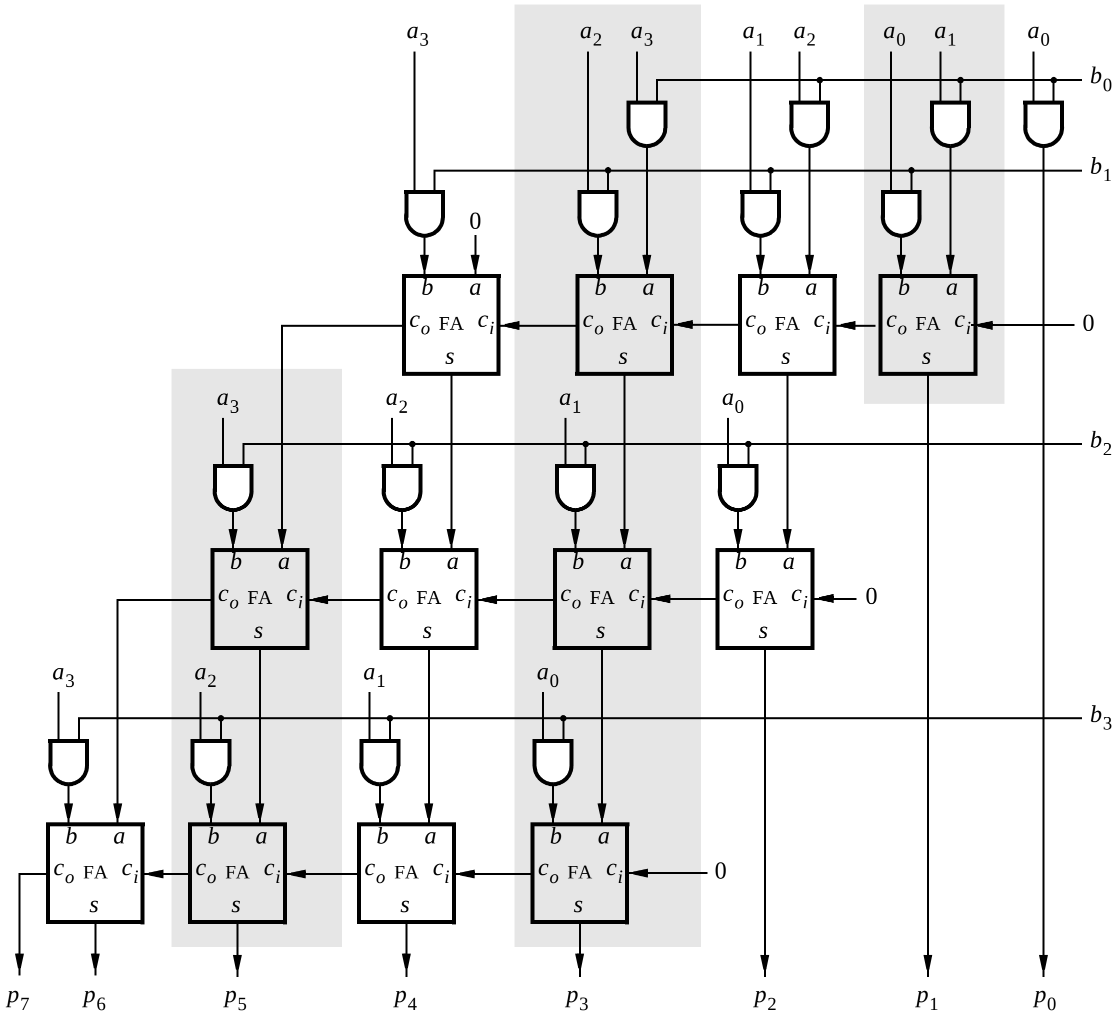


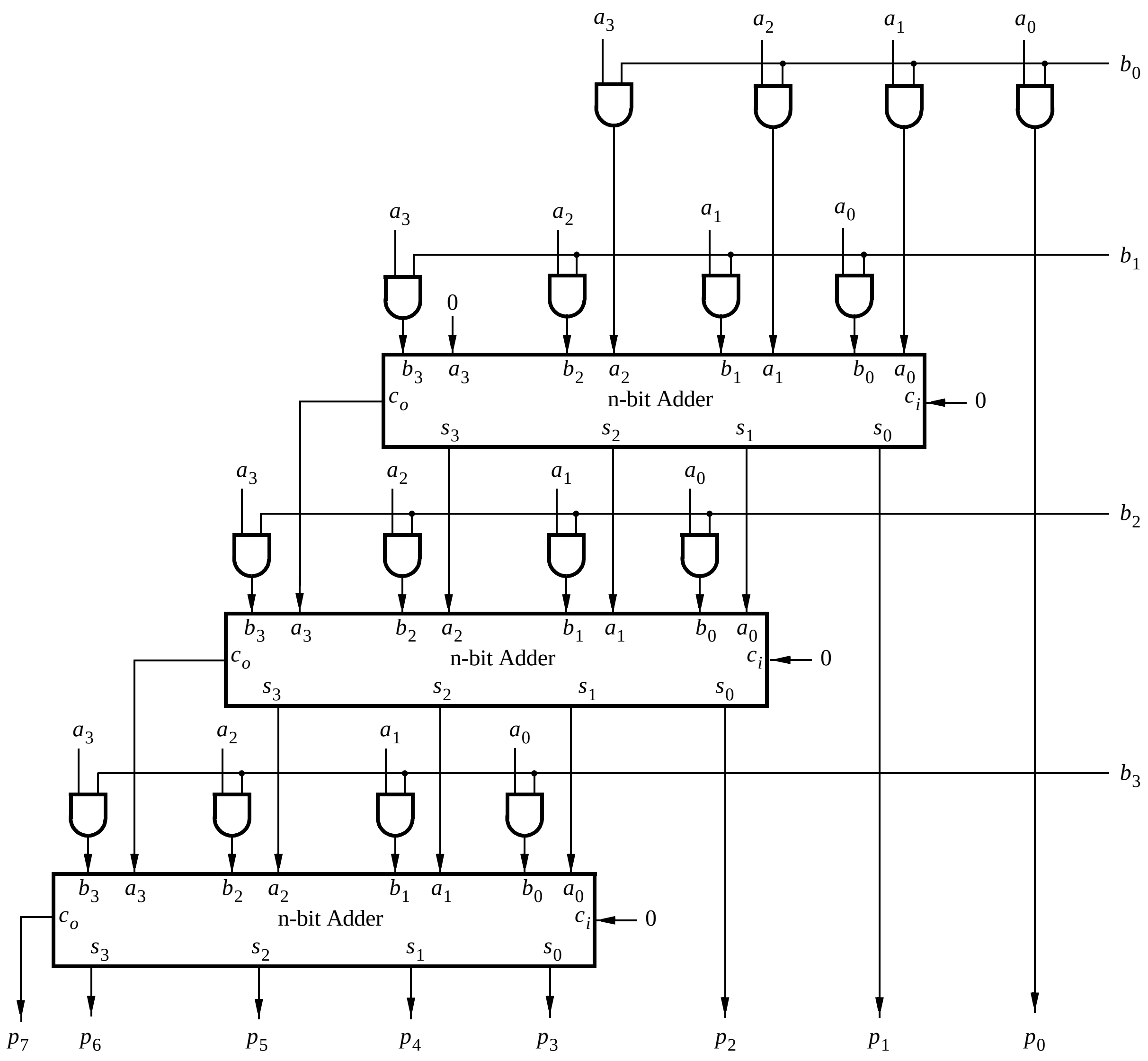
Figure 4: An array multiplier circuit.

Perform the following steps to implement the array multiplier circuit:

1. Create a new Quartus project.
2. Generate the required Verilog file. Use switches SW_{7-4} to represent the number A and switches SW_{3-0} to represent B. The hexadecimal values of A and B are to be displayed on the 7-segment displays *HEX*2 and *HEX*0, respectively. The result $P = A \times B$ is to be displayed on *HEX*5 − 4.
3. Make the necessary pin assignments needed to implement the circuit on your DE-series board, and compile the circuit.
4. Use simulation to verify your design.
5. Download your circuit onto your DE-series board and test its functionality.

Part IV

In Part III, an array multiplier was implemented using full adder modules. At a higher level, a row of full adders functions as an n-bit adder and the array multiplier circuit can be represented as shown in Figure 5.



a_3 a_2 a_1 a_0
b_0
b_1
b_2
b_3
0
n-bit Adder
b_3 a_3 b_2 a_2 b_1 a_1 b_0 a_0
c_o c_i
s_3 s_2 s_1 s_0
p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

Figure 5: An array multiplier implemented using n-bit adders.

Each n-bit adder adds a shifted version of A for a given row and the *partial product* of the row above. Abstracting the multiplier circuit as a sequence of additions allows us to build larger multipliers. The multiplier should consist of n-bit adders arranged in a structure shown in Figure 5. Use this approach to implement an 8 x 8 multiplier circuit with registered inputs and outputs, as shown in Figure 6.

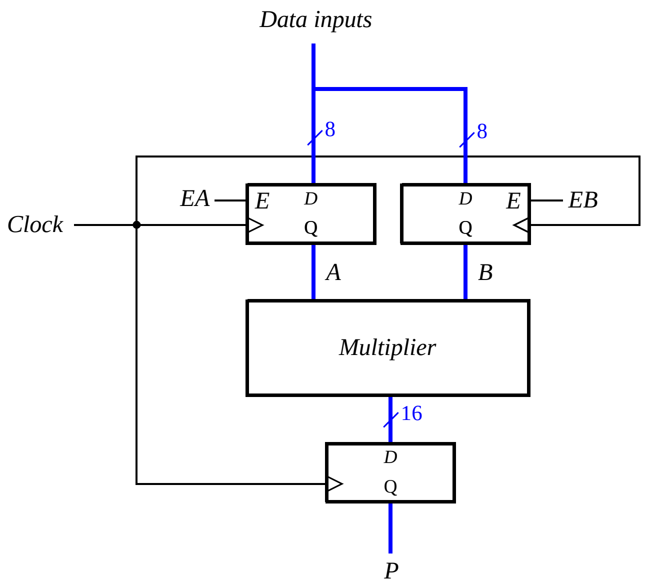


Figure 6: A registered multiplier circuit.

Perform the following steps:

1. Create a new Quartus project and write the required Verilog file.
2. Use switches SW_{7-0} to provide the data inputs to the circuit. Use SW_9 as the enable signal EA for register A, and use SW_8 as the enable for register B. When $SW_9 = 1$ display the contents of register A on the red lights LEDR, and display the contents of register B on these lights when $SW_8 = 1$. Use KEY_0 as a synchronous reset input, and use KEY_1 as a manual clock signal. Show the product $P = A \times B$ as a hexadecimal number on the 7-segment displays *HEX3-0*.
3. Make the necessary pin assignments needed to implement the circuit on your DE-series board, and compile the circuit.
4. Test the functionality of your design by inputting various data values and observing the generated products.

Part V

Part IV showed how to implement multiplication $A \times B$ as a sequence of additions, by accumulating the shifted versions of A one row at a time. Another way to implement this circuit is to perform addition using an adder tree. An adder tree is a method of adding several numbers together in a parallel fashion. This idea is illustrated in Figure 7. In the figure, numbers A, B, C, D, E, F, G, and H are added together in parallel. The addition $A + B$ happens simultaneously with $C + D$, $E + F$ and $G + H$. The result of these operations are then added in parallel again, until the final sum P is computed.

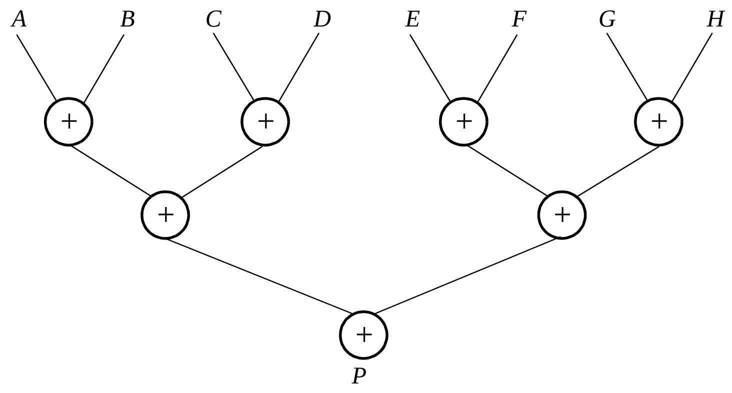


Figure 7: An example of adding 8 numbers using an adder tree.

In this part you are to implement an 8 x 8 multiplier circuit by using the adder-tree approach. Inputs A and B, as well as the output P should be registered as in Part IV.

Digital Logic
Laboratory Exercise 7

Finite State Machines

This is an exercise in using finite state machines.

Part I

We wish to implement a finite state machine (FSM) that recognizes two specific sequences of applied input symbols, namely four consecutive 1s or four consecutive 0s. There is an input w and an output z. Whenever $w = 1$ or $w = 0$ for four consecutive clock pulses the value of z has to be 1; otherwise, $z = 0$. Overlapping sequences are allowed, so that if $w = 1$ for five consecutive clock pulses the output z will be equal to 1 after the fourth and fifth pulses. Figure 1 illustrates the required relationship between w and z.

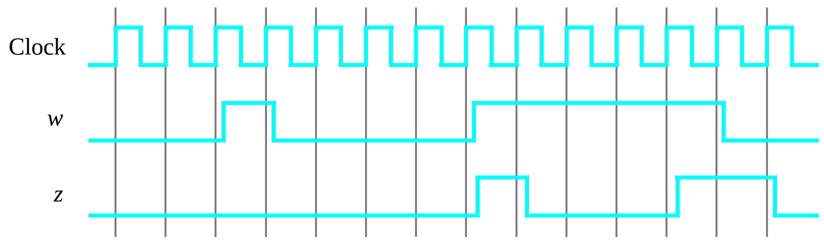


Figure 1: Required timing for the output z.

A state diagram for this FSM is shown in Figure 2. For this part you are to manually derive an FSM circuit that implements this state diagram, including the logic expressions that feed each of the state flip-flops. To implement the FSM use nine state flip-flops called $y_8, \ldots, y_0$ and the one-hot state assignment given in Table 1.

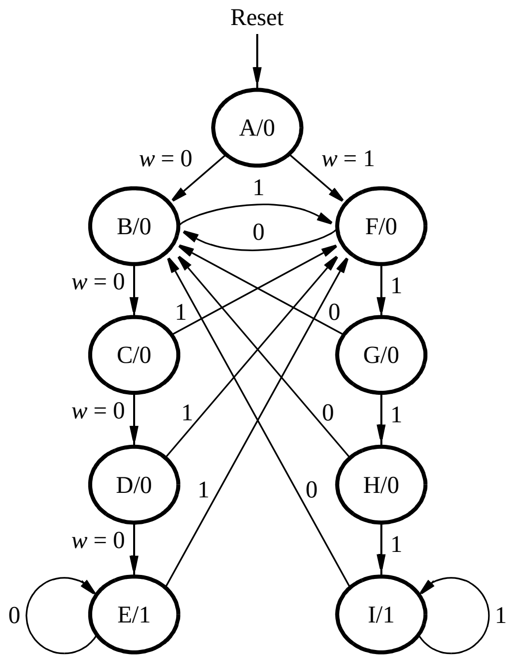


Figure 2: A state diagram for the FSM.

Name	State Code $y_8 y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0$
A	000000001
B	000000010
C	000000100
D	000001000
E	000010000
F	000100000
G	001000000
H	010000000
I	100000000

Table 1: One-hot codes for the FSM.

Design and implement your circuit on your DE-series board as follows:

1. Create a new Quartus® project for the FSM circuit.
2. Write a Verilog file that instantiates the nine flip-flops in the circuit and which specifies the logic expressions that drive the flip-flop input ports. Use only simple **assign** statements in your Verilog code to specify the logic feeding the flip-flops. Note that the one-hot code enables you to derive these expressions by inspection.

   Use the toggle switch *SW*$_0$ as an active-low synchronous reset input for the FSM, use *SW*$_1$ as the *w* input, and the pushbutton *KEY*$_0$ as the clock input which is applied manually. Use the red light *LEDR*$_9$ as the output z, and assign the state flip-flop outputs to the red lights *LEDR*$_8$ to *LEDR*$_0$.
3. Include the Verilog file in your project, and assign the pins on the FPGA to connect to the switches and the LEDs.

4. Simulate the behavior of your circuit.
5. Once you are confident that the circuit works properly as a result of your simulation, download the circuit into the FPGA chip. Test the functionality of your design by applying the input sequences and observing the output LEDs. Make sure that the FSM properly transitions between states as displayed on the red LEDs, and that it produces the correct output values on $LEDR_9$.
6. Finally, consider a modification of the one-hot code given in Table 1. It is often desirable to set all flip-flop outputs to the value 0 in the reset state.

   Table 2 shows a modified one-hot state assignment in which the reset state, *A*, uses all 0s. This is accomplished by inverting the state variable y_0. Create a modified version of your Verilog code that implements this state assignment. (*Hint*: you should need to make very few changes to the logic expressions in your circuit to implement the modified state assignment.)
7. Compile your new circuit and test it.

Name	State Code $y_8y_7y_6y_5y_4y_3y_2y_1y_0$
A	000000000
B	000000011
C	000000101
D	000001001
E	000010001
F	000100001
G	001000001
H	010000001
I	100000001

Table 2: Modified one-hot codes for the FSM.

Part II

For this part you are to write another style of Verilog code for the FSM in Figure 2. In this version of the code you should not manually derive the logic expressions needed for each state flip-flop. Instead, describe the state table for the FSM by using a Verilog **case** statement in an **always** block, and use another **always** block to instantiate the state flip-flops. You can use a third **always** block or simple assignment statements to specify the output z. To implement the FSM, use four state flip-flops $y_3, \ldots, y_0$ and binary codes, as shown in Table 3.

Name	State Code $y_3y_2y_1y_0$
A	0000
B	0001
C	0010
D	0011
E	0100
F	0101
G	0110
H	0111
I	1000

Table 3: Binary codes for the FSM.

A suggested skeleton of the Verilog code is given in Figure 3.

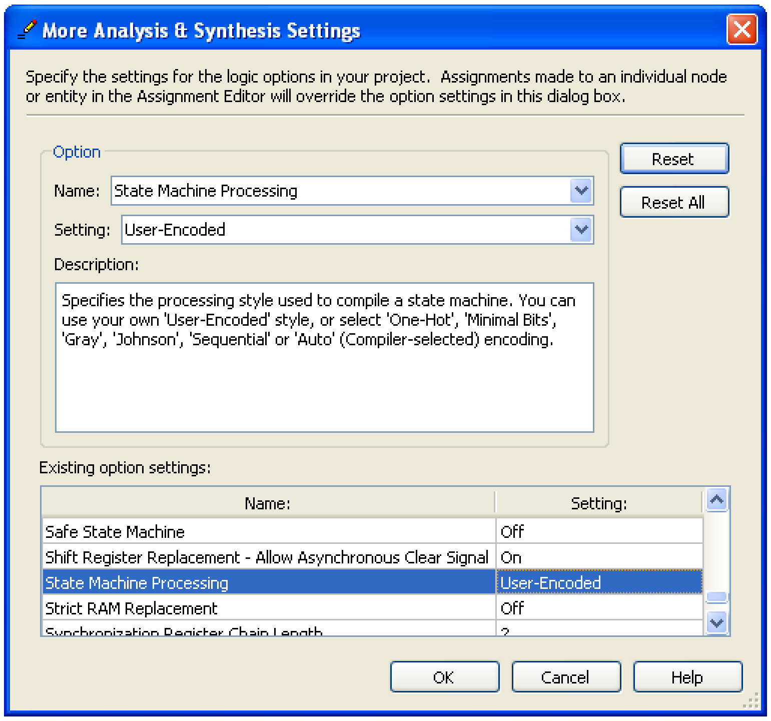
```
module part2 ( . . . );
    . . . define input and output ports

    . . . define signals
    reg [3:0] y_Q, Y_D;     // y_Q represents current state, Y_D represents next state
    parameter A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100,
        F = 4'b0101, G = 4'b0110, H = 4'b0111, I = 4'b1000;

    always @(w, y_Q)
    begin: state_table
        case (y_Q)
            A: if (!w) Y_D = B;
                else Y_D = F;
            . . . remainder of state table
            default: Y_D = 4'bxxxx;
        endcase
    end // state_table

    always @(posedge Clock)
    begin: state_FFs
        . . .
    end // state_FFS

    . . . assignments for output z and the LEDs
endmodule
```

Figure 3: Skeleton Verilog code for the FSM.

Implement your circuit as follows.

1. Create a new project for the FSM.
2. Include in the project your Verilog file that uses the style of code in Figure 3. Use the same switches, pushbuttons, and lights that were used in Part I.
3. Before compiling your code it is necessary to explicitly tell the Synthesis tool in Quartus that you wish to have the finite state machine implemented using the state assignment specified in your Verilog code. If you do not explicitly give this setting to Quartus, the Synthesis tool will automatically use a state assignment of its own choosing, and it will ignore the state codes specified in your Verilog code. To make this setting, choose Assignments > Settings in Quartus, and click on the Compiler Settings item on the left side of the window, then click on the Advanced Settings (Synthesis) button. As indicated in Figure 4, change the parameter State Machine Processing to the setting User-Encoded.
4. Compile your project. To examine the circuit produced by Quartus open the RTL Viewer tool. Double-click on the box shown in the circuit that represents the finite state machine, and determine whether the state diagram that it shows properly corresponds to the one in Figure 2. To see the state codes used for your FSM, open the Compilation Report, select the Analysis and Synthesis section of the report, and click on State Machines.
5. Download the circuit into the FPGA chip and test its functionality.

6. In step 3 you instructed the Quartus Synthesis tool to use the state assignment given in your Verilog code. To see the result of removing this setting, open again the Quartus settings window by choosing Assignments > Settings, and click on the Compiler Settings item on the left side of the window, then click on the Advanced Settings (Synthesis) button. Change the setting for State Machine Processing from User-Encoded to One-Hot. Recompile the circuit and then open the report file, select the Analysis and Synthesis section of the report, and click on State Machines. Compare the state codes shown to those given in Table 2, and discuss any differences that you observe.



Specify the settings for the logic options in your project. Assignments made to an individual node or entity in the Assignment Editor will override the option settings in this dialog box.

Option

Name: State Machine Processing

Setting: User-Encoded

Description:

Specifies the processing style used to compile a state machine. You can use your own 'User-Encoded' style, or select 'One-Hot', 'Minimal Bits', 'Gray', 'Johnson', 'Sequential' or 'Auto' (Compiler-selected) encoding.

Existing option settings:

Name:	Setting:
Safe State Machine	Off
Shift Register Replacement - Allow Asynchronous Clear Signal	On
State Machine Processing	User-Encoded
Strict RAM Replacement	Off
Synchronization Register Chain Length	2

Figure 4: Specifying the state assignment method in Quartus.

Part III

The sequence detector can be implemented in a straightforward manner using shift registers, instead of using the more formal approach described above. Create Verilog code that instantiates two 4-bit shift registers; one is for recognizing a sequence of four 0s, and the other for four 1s. Include the appropriate logic expressions in your design to produce the output z. Make a Quartus project for your design and implement the circuit on your DE-series board. Use the switches and LEDs on the board in a similar way as you did for Parts I and II and observe the behavior of your shift registers and the output z. Answer the following question: could you use just one 4-bit shift register, rather than two? Explain your answer.

Part IV

In this part of the exercise you are to implement a Morse-code encoder using an FSM. The Morse code uses patterns of short and long pulses to represent a message. Each letter is represented as a sequence of dots (a short pulse), and dashes (a long pulse). For example, the first eight letters of the alphabet have the following representation:

A	● —
B	— ● ● ●
C	— ● — ●
D	— ● ●
E	●
F	● ● — ●
G	— — ●
H	● ● ● ●

Design and implement a Morse-code encoder circuit using an FSM. Your circuit should take as input one of the first eight letters of the alphabet and display the Morse code for it on a red LED. Use switches SW_{2-0} and pushbuttons KEY_{1-0} as inputs. When a user presses KEY_1, the circuit should display the Morse code for a letter specified by SW_{2-0} (000 for A, 001 for B, etc.), using 0.5-second pulses to represent dots, and 1.5-second pulses to represent dashes. Pushbutton KEY_0 should function as an asynchronous reset.
A high-level schematic diagram of a possible circuit for the Morse-code encoder is shown in Figure 5.

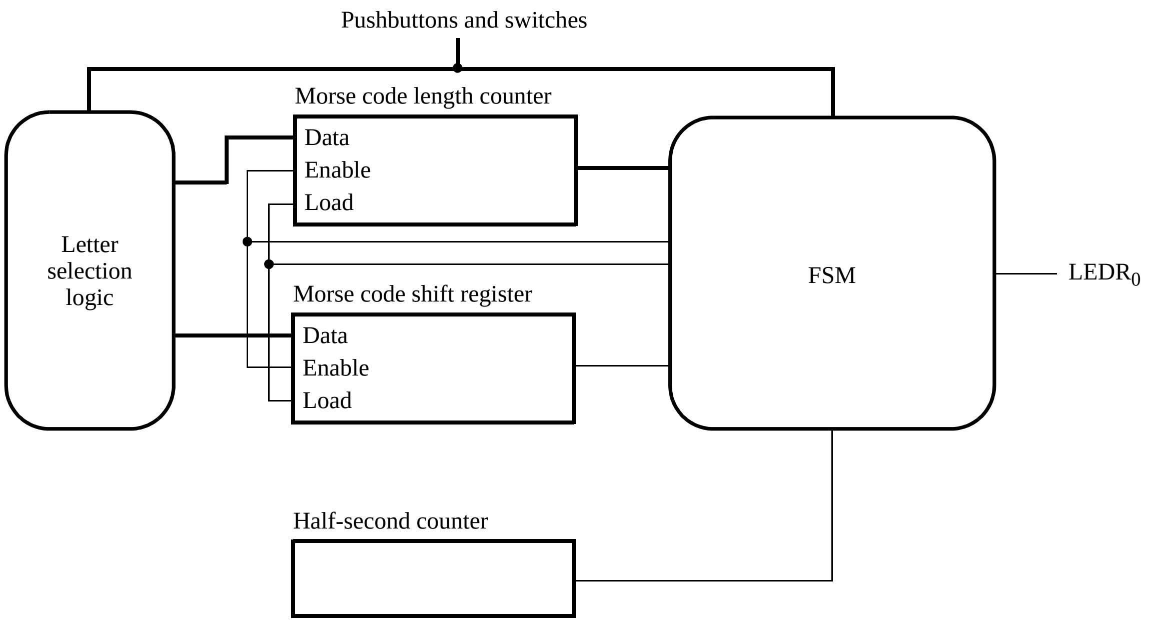


Figure 5: High-level schematic diagram of the circuit for Part IV.

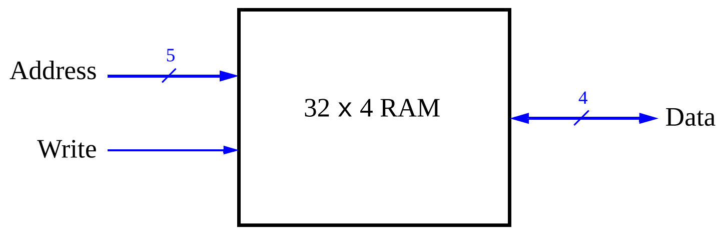
Digital Logic
Laboratory Exercise 8

Memory Blocks

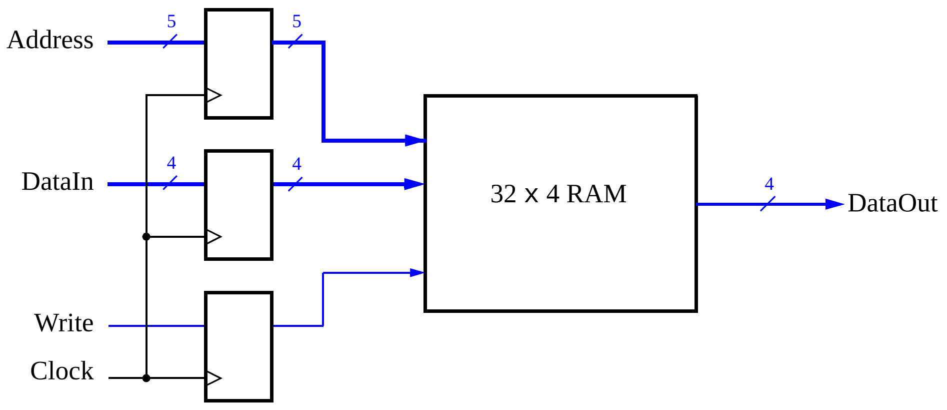
In computer systems it is necessary to provide a substantial amount of memory. If a system is implemented using FPGA technology it is possible to provide some amount of memory by using the memory resources that exist in the FPGA device. In this exercise we will examine the general issues involved in implementing such memory.

A diagram of the random access memory (RAM) module that we will implement is shown in Figure 1*a*. It contains 32 four-bit words (rows), which are accessed using a five-bit *address* port, a four-bit *data* port, and a *write* control input.

The FPGAs that are included on the Intel® FPGA DE10-Lite, DE0-CV, DE1-SoC, and DE2-115 boards provide dedicated memory resources. The MAX® 10 FPGA on the DE10-Lite, and Cyclone® IV FPGA on the DE2-115 contain dedicated memory resources called *M9K blocks*. The Cyclone V FPGA on the DE0-CV and DE1-SoC boards have *M10K blocks*. Each M9K block contains 9216 memory bits, while each M10K block contains 10240 memory bits. Both M9K and M10k blocks can be configured to implement memories of various sizes. A common term used to specify the size of a memory is its *aspect ratio*, which gives the *depth* in words and the *width* in bits (depth x width). In this exercise we will use an aspect ratio that is four bits wide, and we will use only the first 32 words in the memory. Although the M9K and M10K blocks support many other modes of operation, we will not discuss them here.



(*a*) RAM organization



(b) RAM implementation

Figure 1: A 32 x 4 RAM module.

There are two important features of the M9K and M10K blocks that have to be mentioned. First, they includes registers that can be used to synchronize all of the input and output signals to a clock input. The registers on the input ports must always be used, and the registers on the output ports are optional. Second, the blocks have separate ports for data being written to the memory and data being read from the memory. Given these requirements, we will implement the modified 32 x 4 RAM module shown in Figure 1*b*. It includes registers for the *address*, *data input*, and *write* ports, and uses a separate unregistered *data output* port.

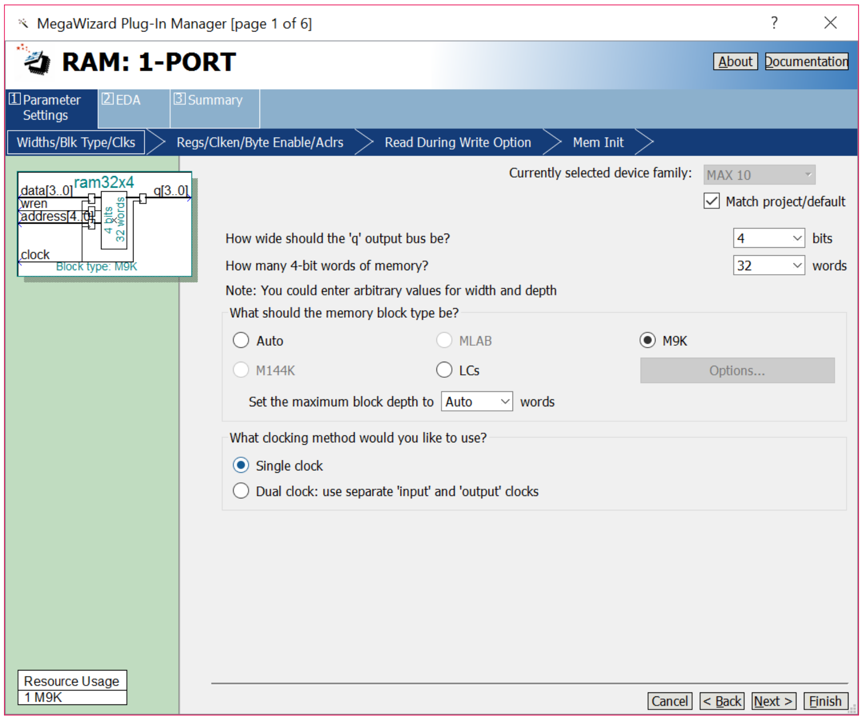
Part I

Commonly used logic structures, such as adders, registers, counters and memories, can be implemented in an FPGA chip by using prebuilt modules that are provided in *libraries*. In this exercise we will use such a module to implement the memory shown in Figure 1*b*.

1. Create a new Quartus® project to implement the memory module.

2. To open the IP Catalog in the Quartus software click on Tools > IP Catalog. In the IP Catalog window choose the *RAM: 1-PORT* module, which is found under the Basic Functions > On Chip Memory category. Select Verilog HDL as the type of output file to create, give the file the name *ram32x4.v*, and click OK. As shown in Figure 2 specify a memory size of 32 four-bit words. Select M9K if your DE-series board has a MAX 10 or Cyclone IV FPGA, otherwise select M10K. Also on this screen accept the default setting to use a single clock for the memory's registers, and then advance to the page shown in Figure 3. On this page *deselect* the setting called 'q' output port under the category Which ports should be registered?. This setting creates a RAM module that matches the structure in Figure 1*b*, with registered input ports and unregistered output ports. Accept defaults for the rest of the settings in the Wizard, and click the Finish button to exit from this tool. Examine the *ram32x4.v* Verilog file which defines the following subcircuit:

module ram32x4 (**input** [4:0] address, **input** clock, **input** [3:0] data, **input** wren, **output** [3:0] q);



Currently selected device family: MAX 10

[✓] Match project/default

How wide should the 'q' output bus be? 4 bits

How many 4-bit words of memory? 32 words

Note: You could enter arbitrary values for width and depth

What should the memory block type be?

Auto MLAB [● M9K] M144K LCs

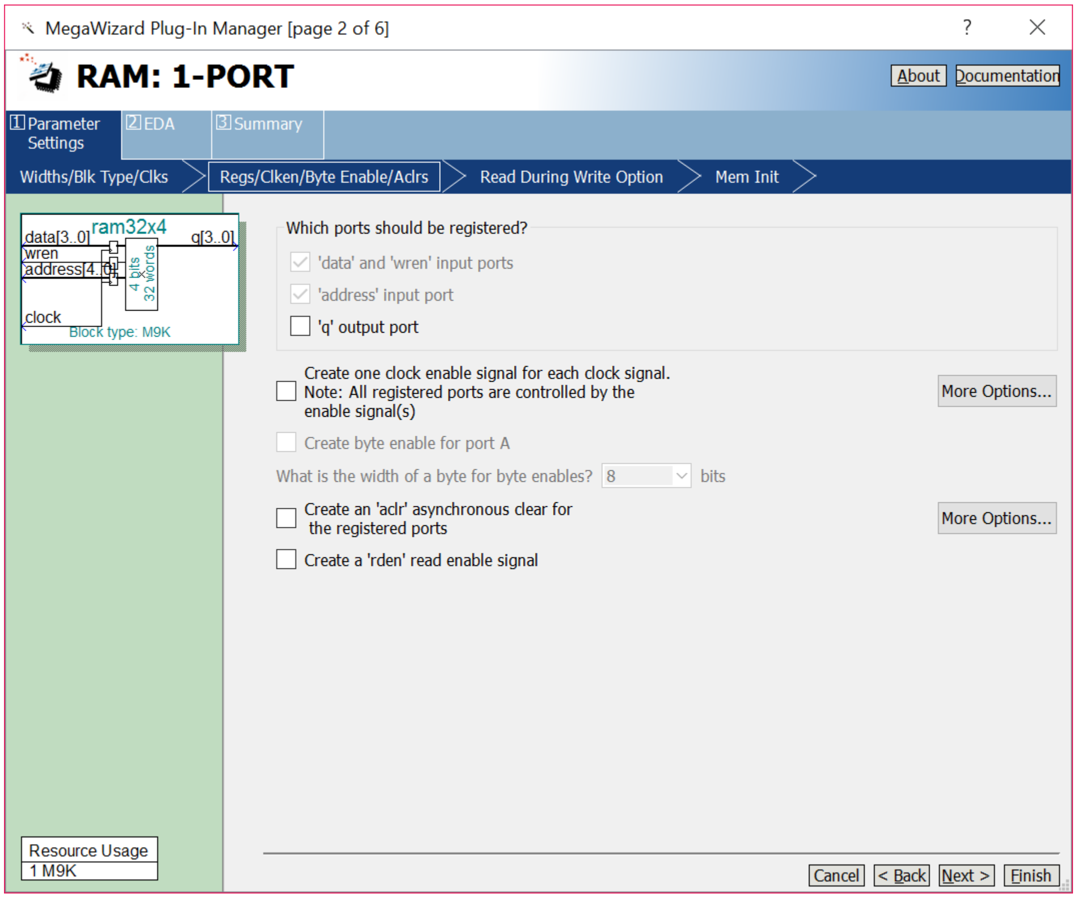
Set the maximum block depth to Auto words

What clocking method would you like to use?

[● Single clock]

Dual clock: use separate 'input' and 'output' clocks

Figure 2: Configuring the size of the memory module.


MegaWizard Plug-In Manager [page 2 of 6]

RAM: 1-PORT

About | Documentation

1 Parameter Settings | 2 EDA | 3 Summary

Widths/Blk Type/Clks > Regs/Clken/Byte Enable/Adrs > Read During Write Option > Mem Init

Which ports should be registered?

- [✓] 'data' and 'wren' input ports
- [✓] 'address' input port
- [ ] 'q' output port

- [ ] Create one clock enable signal for each clock signal. Note: All registered ports are controlled by the enable signal(s) — More Options...
- [ ] Create byte enable for port A

What is the width of a byte for byte enables? 8 bits

- [ ] Create an 'aclr' asynchronous clear for the registered ports — More Options...
- [ ] Create a 'rden' read enable signal

Resource Usage
1 M9K

Cancel | < Back | Next > | Finish

Figure 3: Configuring input and output ports.

3. Instantiate this subcircuit in a top-level Verilog file that includes appropriate input and output signals for the memory ports given in Figure 1*b*. Compile the circuit. Observe in the Compilation Report that the Quartus Compiler uses 128 bits in one of the FPGA memory blocks to implement the RAM circuit.

4. Simulate the behavior of your circuit using Modelsim and ensure that you can read and write data in the memory. Use the included testbench file as a baseline for your simulation inputs. An example simulation output is given in Figure 4.

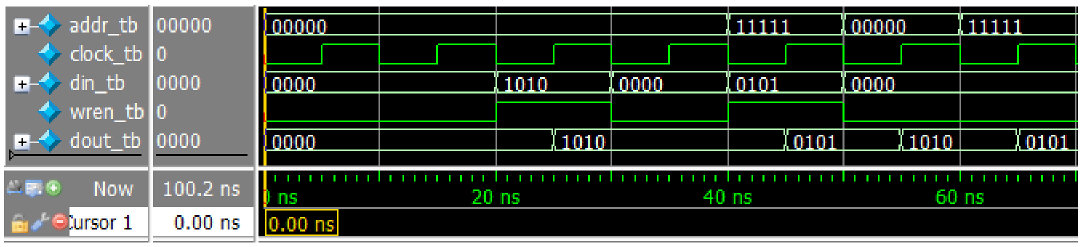


Figure 4: An example of simulation output.

Part II

Now, we want to realize the memory circuit in the FPGA on your DE-series board, and use slide switches to load some data into the created memory. We also want to display the contents of the RAM on the 7-segment displays.

1. Make a new Quartus project which will be used to implement the desired circuit on your DE-series board.
2. Create another Verilog file that instantiates the *ram32x4* module and that includes the required input and output pins on your DE-series board. Use slide switches SW_{3-0} to provide input data for the RAM, and use switches switches SW_{8-4} to specify the address. Use SW_9 as the *Write* signal and use KEY_0 as the *Clock* input. Show the address value on the 7-segment displays $HEX5-4$, show the data being input to the memory on $HEX2$, and show the data read out of the memory on $HEX0$.
3. Test your circuit and make sure that data can be stored into the memory at various locations.

Part III

Instead of creating a memory module subcircuit by using the IP Catalog, we can implement the required memory by specifying its structure in Verilog code. In a Verilog-specified design it is possible to define the memory as a multidimensional array. A 32 x 4 array, which has 32 words with 4 bits per word, can be declared by the statement

```
reg [3:0]  memory_array [31:0];
```

In the Cyclone series of FPGAs, such an array can be implemented either by using the flip-flops that each logic element contains or, more efficiently, by using the built-in memory blocks. The Quartus Help provides other examples of Verilog code that show how memory can be specified (search in the Help for "Inferred memory").

Perform the following steps:

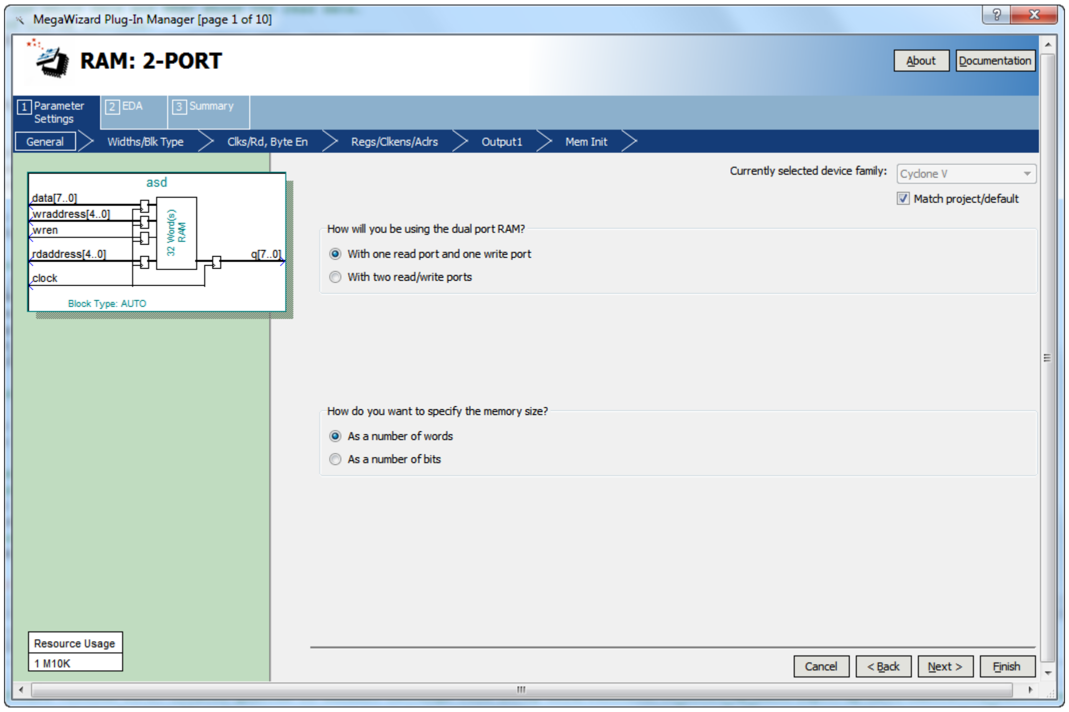
1. Create a new project which will be used to implement the desired circuit on your DE-series board.
2. Write a Verilog file that provides the necessary functionality, including the ability to load the RAM and read its contents as was done in Part II.
3. Assign the pins on the FPGA to connect to the switches and the 7-segment displays.
4. Compile the circuit and download it into the FPGA chip.
5. Test the functionality of your design by applying some inputs and observing the output.

Part IV

The SRAM block in Figure 1 has a single port that provides the address for both read and write operations. For this part you will create a different type of memory module, in which there is one port for supplying the address for a read operation, and a separate port that gives the address for a write operation. Perform the following steps.

1. Create a new Quartus project for your circuit. To generate the desired memory module open the IP Catalog and select the *RAM: 2-PORT* module in the Basic Functions > On Chip Memory category. As shown in Figure 5, choose With one read port and one write port in the category called How will you be using the dual port ram?

   Configure the memory size, clocking method, and registered ports the same way as Part II. As shown in Figure 6 select I do not care (The outputs will be undefined) for Mixed Port Read-During-Write for Single Input Clock RAM. This setting specifies that it does not matter whether the memory outputs the new data being written, or the old data previously stored, in the case that the write and read addresses are the same during a write operation.


MegaWizard Plug-In Manager [page 1 of 10]

RAM: 2-PORT

Currently selected device family: Cyclone V

[✓] Match project/default

How will you be using the dual port RAM?

[● With one read port and one write port]

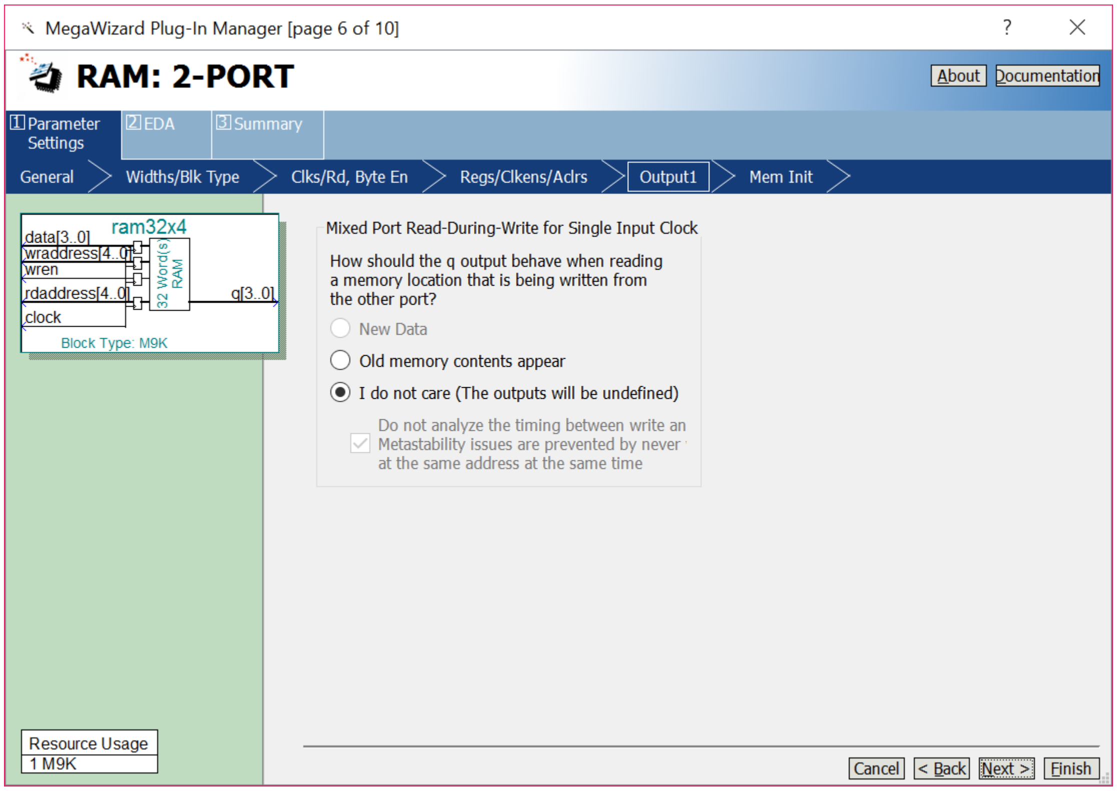
○ With two read/write ports

How do you want to specify the memory size?

[● As a number of words]

○ As a number of bits

Figure 5: Configuring the two input ports of the RAM.


MegaWizard Plug-In Manager [page 6 of 10]

RAM: 2-PORT

Mixed Port Read-During-Write for Single Input Clock

How should the q output behave when reading a memory location that is being written from the other port?

○ New Data

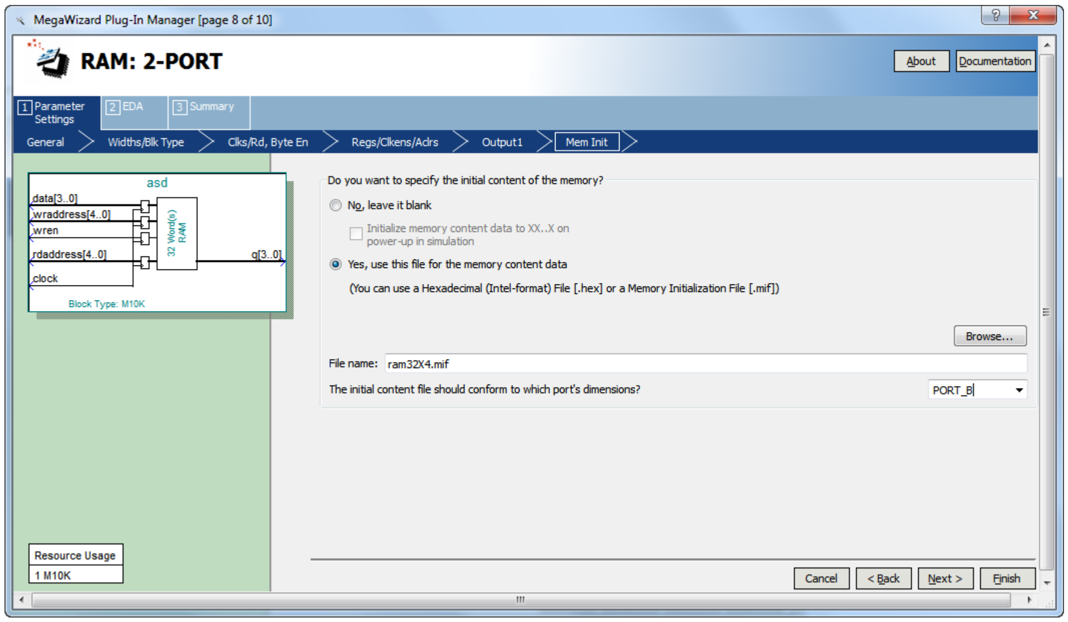
○ Old memory contents appear

[● I do not care (The outputs will be undefined)]

[✓] Do not analyze the timing between write an Metastability issues are prevented by never at the same address at the same time

Figure 6: Configuring the output of the RAM when reading and writing to the same address.

Figure 7 shows how the memory words can be initialized to specific values. It makes use of a feature that allows the memory module to be loaded with data when the circuit is programmed into the FPGA chip. As shown in the figure, choose the setting Yes, use this file for the memory content data, and specify the filename *ram32x4.mif*. An example of a *MIF* file is provided in Figure 8. You can also learn about the format of a *memory initialization file* (MIF) by using the Quartus Help. You will need to create a MIF file like the one in Figure 8 to test your circuit. Finish the Wizard and then examine the generated memory module in the file *ram32x4.v*.



Do you want to specify the initial content of the memory?

No, leave it blank

Initialize memory content data to XX..X on power-up in simulation

[✓ Yes, use this file for the memory content data]

(You can use a Hexadecimal (Intel-format) File [.hex] or a Memory Initialization File [.mif])

File name: ram32X4.mif

The initial content file should conform to which port's dimensions? PORT_B

Figure 7: Specifying a memory initialization file (MIF).

```
DEPTH = 32;
WIDTH = 4;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

0 : 0000;
1 : 0001;
2 : 0010;
3 : 0011;
. . . (some lines not shown)
1E : 1110;
1F : 1111;

END;
```

Figure 8: An example memory initialization file (MIF).

2. Write a Verilog file that instantiates your dual-port memory. To see the RAM contents, add to your design a capability to display the content of each four-bit word (in hexadecimal format) on the 7-segment display

*HEX*0. Use a counter as a read address, and scroll through the memory locations by displaying each word for about one second. As each word is being displayed, show its address (in hex format) on the 7-segment displays *HEX*$3-2$. Use the 50 MHz clock, *CLOCK_50*, and use KEY_0 as a reset input. For the write address and corresponding data use switches SW_{8-4} and SW_{3-0}. Show the write address on *HEX*$5-4$ and show the write data on *HEX*1. Make sure that you properly synchronize the slide switch inputs to the 50 MHz clock signal.

3. Test your circuit and verify that the initial contents of the memory match your *ram32x4.mif* file. Make sure that you can independently write data to any address by using the slide switches.

Laboratory Exercise 9

A Simple Processor

Figure 1 shows a digital system that contains a number of 16-bit registers, a multiplexer, an adder/subtracter, and a control unit (finite state machine). Information is input to this system via the 16-bit *DIN* input, which is loaded into the *IR* register. Data can be transferred through the 16-bit wide multiplexer from one register in the system to another, such as from register *IR* into one of the *general purpose* registers $r0, \ldots, r7$. The multiplexer's output is called *Buswires* in the figure because the term *bus* is often used for wiring that allows data to be transferred from one location in a system to another. The FSM controls the *Select* lines of the multiplexer, which allows any of its inputs to be transferred to any register that is connected to the bus wires.

The system can perform different operations in each clock cycle, as governed by the FSM. It determines when particular data is placed onto the bus wires and controls which of the registers is to be loaded with this data. For example, if the FSM selects $r0$ as the output of the bus multiplexer and also asserts A_{in}, then the contents of register $r0$ will be loaded on the next active clock edge into register *A*.

Addition or subtraction of signed numbers is performed by using the multiplexer to first place one 16-bit number onto the bus wires, and then loading this number into register *A*. Once this is done, a second 16-bit number is placed onto the bus, the adder/subtracter performs the required operation, and the result is loaded into register *G*. The data in *G* can then be transferred via the multiplexer to one of the other registers, as required.

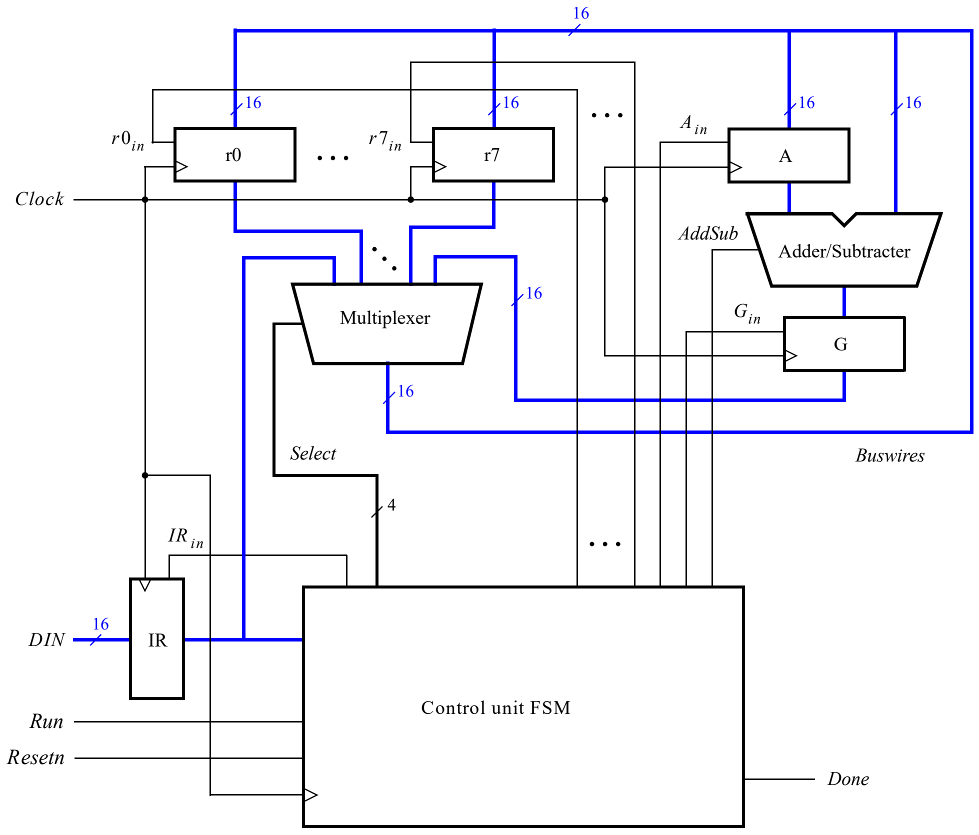


Figure 1: A digital system.

A system like the one in Figure 1 is often called a *processor*. It executes operations specified in the form of *instructions*. Table 1 lists the instructions that this processor supports. The left column shows the name of an instruction and its operands. The meaning of the syntax $rX \leftarrow Op2$ is that the second operand, *Op2*, is loaded into register *rX*. The operand *Op2* can be either a register, *rY*, or *immediate data*, #*D*.

Instruction		Function performed
mv	*rX*, *Op*2	$rX \leftarrow Op2$
mvt	*rX*, #*D*	$rX_{15-8} \leftarrow D_{7-0}$
add	*rX*, *Op*2	$rX \leftarrow rX + Op2$
sub	*rX*, *Op*2	$rX \leftarrow rX - Op2$

Table 1: Instructions performed in the processor.

Instructions are loaded from the external input *DIN*, and stored into the *IR* register, using the connection indicated in Figure 1. Each instruction is *encoded* using a 16-bit format. If *Op*2 specifies a register, then the instruction encoding is `III0XXX000000YYY`, where `III` specifies the instruction, `XXX` gives the *rX* register, and `YYY` gives the *rY* register. If *Op*2 specifies immediate data #*D*, then the encoding is `III1XXXDDDDDDDDD`, where the field `DDDDDDDDD` represents a nine-bit *signed* (2's complement) value. Although only two bits are needed to encode our four instructions, we are using three bits because other instructions will be added to the processor later. Assume that `III` = 000 for the *mv* instruction, 001 for *mvt*, 010 for *add*, and 011 for *sub*.

The *mv* instruction (*move*) copies the contents of one register into another, using the syntax `mv rX, rY`. It can also be used to initialize a register with immediate data, as in `mv rX, #D`. Since the data *D* is represented inside the encoded instruction using only nine bits, the processor has to *sign-extend* the data, as in $D_8D_8D_8D_8D_8D_8D_8D_{8-0}$, before loading it into register *rX*. The *mvt* instruction (*move top*) is used to initialize the most-significant byte of a register. The instruction `mvt rX, #D` loads the 16-bit value $\texttt{D}_{7-0}\texttt{00000000}$ into *rX*. As an example, to load register $r0$ with the value `0xFF00`, you would use the instruction `mvt r0, #0xFF`. The instruction `add rX, rY` produces the sum $rX + rY$ and loads the result into *rX*. The instruction `add rX, #D` produces the sum $rX + D$, where *D* is sign-extended to 16 bits, and saves the result in *rX*. The *sub* instruction generates either $rX - rY$, or $rX - \#D$ and loads the result into *rX*.

Some instructions, such as an *add* or *sub*, take a few clock cycles to complete, because multiple transfers have to be performed across the bus. The finite state machine in the processor "steps through" such instructions, asserting the control signals needed in successive clock cycles until the instruction has completed. The processor starts executing the instruction on the *DIN* input when the *Run* signal is asserted and the processor asserts the *Done* output when the instruction is finished. Table 2 indicates the control signals from Figure 1 that have to be asserted in each time step to implement the instructions in Table 1. The only control signal asserted in time step T_0, for all instructions, is IR_{in}. The meaning of *Select rY* or *IR* in the table is that the multiplexer selects and puts onto the bus (*BusWires*) either the contents of register *rY* or the immediate data in *IR*, depending on the value of *Op*2. For the *mv* instruction, when *IR* is selected the multiplexer outputs `DDDDDDDDD` sign-extended to 16 bits, and for *mvt* the multiplexer outputs `DDDDDDDD00000000`. Only signals from Figure 1 that have to be asserted in each time step are listed in Table 1; all other signals are not asserted. The meaning of *AddSub* in step T_2 of the *sub* instruction is that this signal is set to 1, and this setting causes the adder/subtracter unit to perform subtraction using 2's-complement arithmetic.

The processor in Figure 1 can perform various tasks by using a sequence of instructions. For example, the sequence below loads the number 28 into register $r0$ and then calculates, in register $r1$, the 2's complement value -28.

```
mv     r0, #28        // original number = 28
mvt    r1, #0xFF
add    r1, #0xFF      // r1 = 0xFFFF
sub    r1, r0         // r1 = 1's-complement of r0
add    r1, #1         // r1 = 2's-complement of r0 = -28
```

This sequence of instructions produces the same result as the single instruction `mv r1,#-28`.

	T_0	T_1	T_2	T_3
mv	*IR*$_{in}$	*Select rY* or *IR*, *rX*$_{in}$, *Done*		
mvt	*IR*$_{in}$	*Select IR*, *rX*$_{in}$, *Done*		
add	*IR*$_{in}$	*Select rX*, A_{in}	*Select rY* or *IR*, G_{in}	*Select G*, *rX*$_{in}$, *Done*
sub	*IR*$_{in}$	*Select rX*, A_{in}	*Select rY* or *IR*, *AddSub*, G_{in}	*Select G*, *rX*$_{in}$, *Done*

Table 2: Control signals asserted in each instruction/time step.

Part I

Implement the processor shown in Figure 1 using Verilog code, as follows:

1. Make a new folder for this part of the exercise. Part of the Verilog code for the processor is shown in parts a to c of Figure 2, and a more complete version of the code is provided with this exercise, in a file named *proc.v*. You can modify this code to suit your own coding style if desired—the provided code is just a suggested solution. Fill in the missing parts of the Verilog code to complete the design of the processor.

```
module proc(DIN, Resetn, Clock, Run, Done);
    input [15:0] DIN;
    input Resetn, Clock, Run;
    output Done;
    ... declare variables

    assign III = IR[15:13];
    assign IMM = IR[12];
    assign rX = IR[11:9];
    assign rY = IR[2:0];
    dec3to8 decX (rX_in, rX, R_in); // produce r0 - r7 register enables

    parameter T0 = 3'b000, T1 = 3'b001, T2 = 3'b010, T3 = 3'b011;
    // Control FSM state table
    always @(Tstep_Q, Run, Done)
        case (Tstep_Q)
            T0: // data is loaded into IR in this time step
            if (~Run) Tstep_D = T0;
            else Tstep_D = T1;
            T1: ...
            ...
      endcase
```

Figure 2: Skeleton Verilog code for the processor. (Part a)

```
parameter mv = 3'b000, mvt = 3'b001, add = 3'b010, sub = 3'b011;
// selectors for the BusWires multiplexer
parameter _R0 = 4'b0000, _R1 = 4'b0001, _R2 = 4'b0010,
    _R3 = 4'0011, ..., _R7 = 4'b0111, _G = 4'b1000,
    _IR8_IR8_0 /* signed-extended immediate data */ = 4'b1001,
    _IR7_0_0   /* immediate data << 8 */ = 4'b1010;

// control FSM outputs
always @(*) begin
    rX_in = 1'b0; Done = 1'b0; ... // default values for variables
    case (Tstep_Q)
        T0: // store DIN into IR
            IR_in = 1'b1;
        T1: // define signals in time step T1
            case (III)
                mv: begin
                    if (!Imm) Select = rY;            // mv rX, rY
                    else Select = _IR8_IR8_0;         // mv rX, #D
                    rX_in = 1'b1;                     // enable rX
                    Done = 1'b1;
                end
                mvt: // mvt rX, #D
                ...
            endcase
        T2: // define signals in time step T2
            case (III)
                ...
            endcase
        T3: // define signals in time step T3
            case (III)
                ...
            endcase
        default: ;
    endcase
end

// Control FSM flip-flops
always @(posedge Clock, negedge Resetn)
    if (!Resetn)
        ...

regn reg_0 (BusWires, Resetn, R_in[0], Clock, r0);
regn reg_1 (BusWires, Resetn, R_in[1], Clock, r1);
...
regn reg_7 (BusWires, Resetn, R_in[7], Clock, r7);

... instantiate other registers and the adder/subtracter unit
```

Figure 2: Skeleton Verilog code for the processor. (Part *b*)

```
    // define the internal processor bus
    always @(*)
        case (Select)
            _R0: BusWires = r0;
            _R1: BusWires = r1;
            ...
            _R7: BusWires = r7;
            _G: BusWires = G;
            _IR8_IR8_0: BusWires = {{7{IR[8]}}, IR[8:0]};
            _IR7_0_0: BusWires = {IR[7:0], 8'b00000000};
            default: BusWires = 16'bxxxxxxxxxxxxxxxx;
        endcase
endmodule

module dec3to8(E, W, Y);
    input E; // enable
    input [2:0] W;
    output [0:7] Y;
    reg [0:7] Y;

    always @(*)
        if (E == 0)
            Y = 8'b00000000;
        else
            case (W)
                3'b000: Y = 8'b10000000;
                3'b001: Y = 8'b01000000;
                3'b010: Y = 8'b00100000;
                3'b011: Y = 8'b00010000;
                3'b100: Y = 8'b00001000;
                3'b101: Y = 8'b00000100;
                3'b110: Y = 8'b00000010;
                3'b111: Y = 8'b00000001;
            endcase
endmodule
```

Figure 2: Skeleton Verilog code for the processor. (Part *c*)

2. Set up the required subfolder and files so that your Verilog code can be compiled and simulated using the Questa or ModelSim Simulator to verify that your processor works properly. An example simulation result for a correctly-designed circuit is given in Figure 3. It shows the value `0x101C` being loaded into *IR* from *DIN* at time 30 ns. This pattern represents the instruction `mv r0,#28`, where the immediate value $D = 28$ (`0x1C`) is loaded into $r0$ on the clock edge at 50 ns. The simulation results then show the instruction `mvt r1,#0xFF` at 70 ns, `add r0,#0xFF` starting at 110 ns, and `sub r1,r0` starting at 190 ns.

   You should perform a thorough simulation of your processor. A sample Verilog testbench file, *testbench.v*, execution script, *testbench.tcl*, and waveform file, *wave.do* are provided along with this exercise.

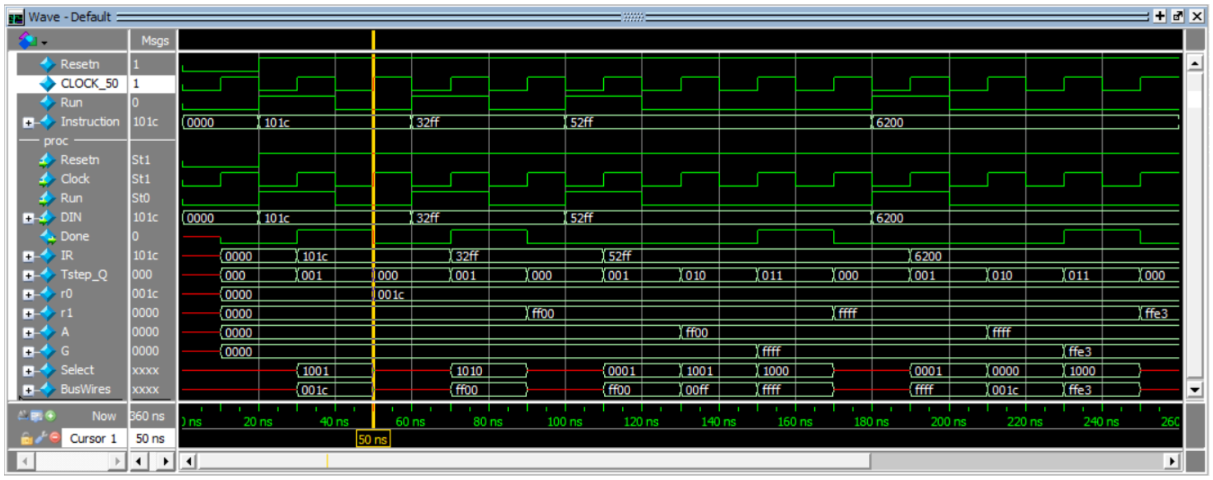


Figure 3: Simulation results for the processor.

Part II

In this part we will design the circuit illustrated in Figure 4, in which a memory module and counter are connected to the processor. The counter is used to read the contents of successive locations in the memory, and this data is provided to the processor as a stream of instructions. To simplify the design and testing of this circuit we have used separate clock signals, *PClock* and *MClock*, for the processor and memory.

You are to implement the circuit in Figure 4 in a suitable FPGA (for example, the DE1-SoC, DE10-Standard, or DE10-Lite) board. As part of the process for designing and debugging the circuit, you must use the Questa or ModelSim simulator to test your Verilog code's functionality. Also, you can use the DESim tool as a way of observing how your circuit will behave on an FPGA board, even if you do not have access to a physical board (for example, at home). You are encouraged to make use of the DESim tool to help in the design and debug of your Verilog code. This tool is available from `FPGAcademy.org`. To make it easy to use the DESim tool, the design files for this part of the exercise include all required DESim setup files. Perform the following steps:

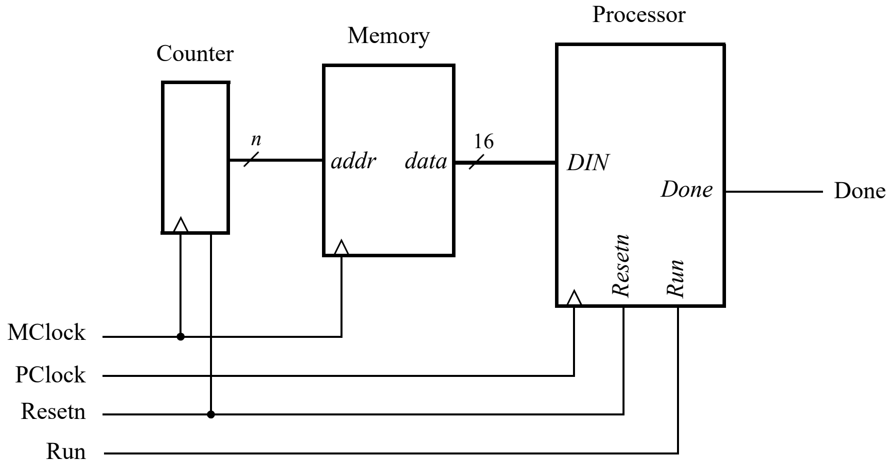


Figure 4: Connecting the processor to a memory module and counter.

1. A sample top-level Verilog file that instantiates the processor, memory module, and counter is shown in Figure 5. This code is provided, along with this exercise, in a file named *part2.v*. It is the top-level Verilog file for this part of the exercise. The code instantiates a memory module called *inst_mem*. A diagram of

this memory module is depicted in Figure 6. Since this module has only a read port, and no write port, it is called a *synchronous read-only memory (synchronous ROM)*. The memory module includes a register for synchronously loading addresses. This register is required due to the design of the memory resources in the Intel FPGA chip.

The synchronous ROM is defined in a Verilog source-code file named *inst_mem.v*, which is provided along with this exercise. You can use the provided file, or you can create one yourself (if you want to see how this is done) by using the Quartus Prime software. The instructions for creating the *inst_mem.v* file using the Quartus software are given below. If you do not wish to perform these steps, and just want to make use of the provided file, then skip to item 3, below.

```
 module part2 (KEY, SW, LEDR);
    input [1:0] KEY;
    input [9:0] SW;
    output [9:0] LEDR;

    wire Done, Resetn, PClock, MClock, Run;
    wire [15:0] DIN;
    wire [4:0] pc;
    assign Resetn = SW[0];
    assign MClock = KEY[0];
    assign PClock = KEY[1];
    assign Run = SW[9];
    proc U1 (DIN, Resetn, PClock, Run, Done);
    assign LEDR[9] = Done;
    inst_mem U2 (pc, MClock, DIN);
    count5 U3 (Resetn, MClock, pc);
endmodule

module count5 (Resetn, Clock, Q);
    input Resetn, Clock;
    output reg [4:0] Q;

    always @ (posedge Clock, negedge Resetn)
        if (Resetn == 0)
            Q <= 5'b00000;
        else
            Q <= Q + 1'b1;
endmodule
```

Figure 5: Verilog code for the top-level module.

2. A Quartus project file is provided along with this part of the exercise. Use the Quartus software to open this project, which is called *part2.qpf*. The top-level file in this Quartus project is *part2.v*. Use the Quartus IP Catalog tool to create the memory module, by clicking on Tools > IP Catalog in the Quartus software. In the IP Catalog window choose the *ROM: 1-PORT* module, which is found under the Basic Functions > On Chip Memory category. Select Verilog HDL as the type of output file to create, and give the file the name *inst_mem.v*.

   Follow through the provided dialogue to create a memory that has one 16-bit wide read data port and is 32 words deep. Figures 7 and 8 show the relevant pages and how to properly configure the memory. In Figure 8, under `Which ports should be registered?`, ensure that the `'q' output port` is *not* selected. To place processor instructions into the memory, you need to specify *initial values* that should be stored in the memory when your circuit is programmed into the FPGA chip. This can be done

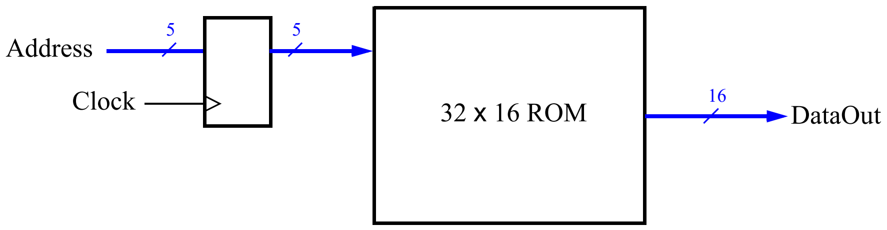
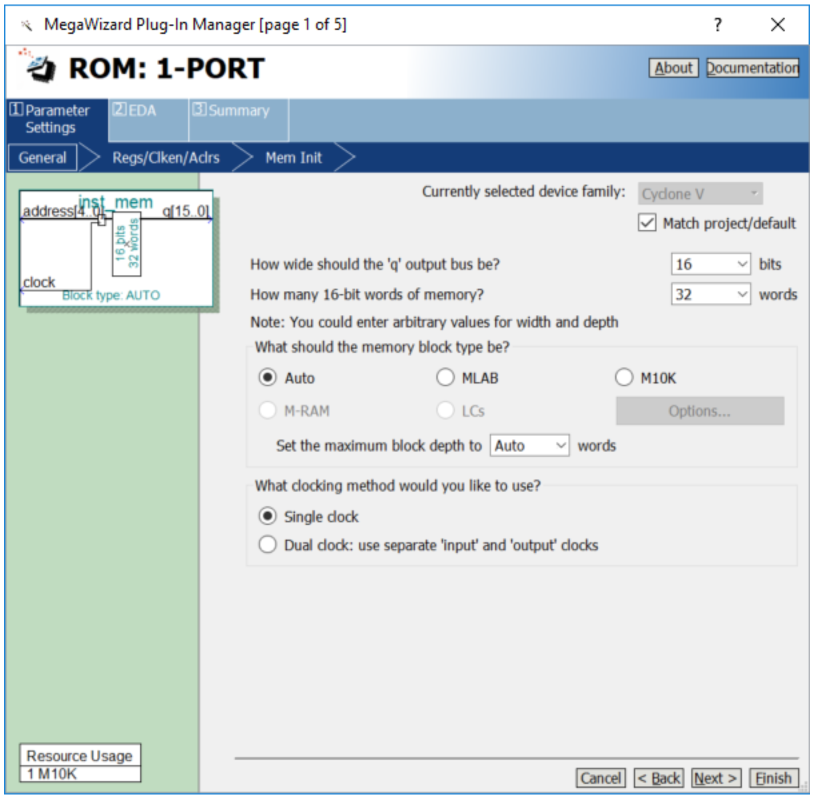


Figure 6: The 32 x 16 ROM with address register.

by initializing the memory using the contents of a *memory initialization file (MIF)*. The appropriate screen is illustrated in Figure 9. We have specified a file named *inst_mem.mif*, which then has to be created in the folder that contains the Quartus project. Clicking `Next` two more times will advance to the `Summary` screen, which lists the names of files that will be created for the memory IP. You should select *only* the Verilog file *inst_mem.v*. Make sure that none of the other types of files are selected, and then click `Finish`.



Currently selected device family: Cyclone V

☑ Match project/default

How wide should the 'q' output bus be? 16 bits

How many 16-bit words of memory? 32 words

Note: You could enter arbitrary values for width and depth

What should the memory block type be?

● Auto ○ MLAB ○ M10K

○ M-RAM ○ LCs

Set the maximum block depth to Auto words

What clocking method would you like to use?

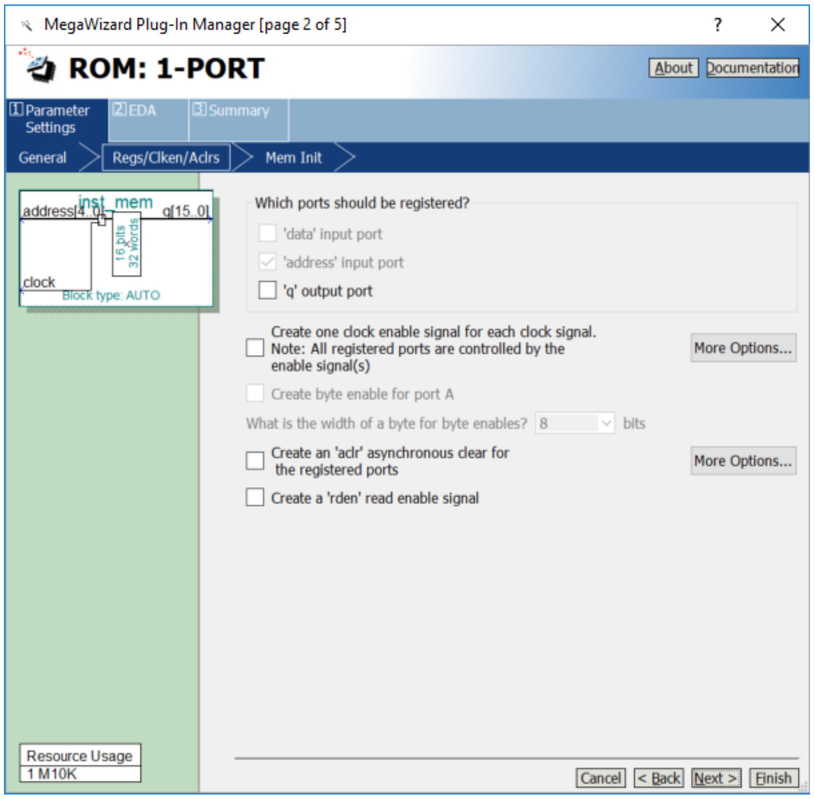
● Single clock

○ Dual clock: use separate 'input' and 'output' clocks

Figure 7: Specifying memory size.

3. As described above, you need to provide a memory initialization file (MIF) called *inst_mem.mif* to specify the contents of the ROM. An example of a memory initialization file is given in Figure 10. Note that comments (% . . . %) are included in this file as a way of documenting the meaning of the provided instructions. Set the contents of your `MIF` file such that it provides enough processor instructions to test your circuit.

4. The Verilog code in Figure 5 includes the appropriate port names for implementation of the design on an FPGA board, like the DE1-SoC. The switch SW_9 drives the processor's *Run* input, SW_0 is connected to *Resetn*, KEY_0 to *MClock*, and KEY_1 to *PClock*. The Run signal is displayed on $LEDR_9$ and *Done* is connected to $LEDR_0$.


MegaWizard Plug-In Manager [page 2 of 5]

ROM: 1-PORT

Parameter Settings | EDA | Summary

General > Regs/Clken/Aclrs > Mem Init

Which ports should be registered?

- [ ] 'data' input port
- [✓] 'address' input port
- [ ] 'q' output port

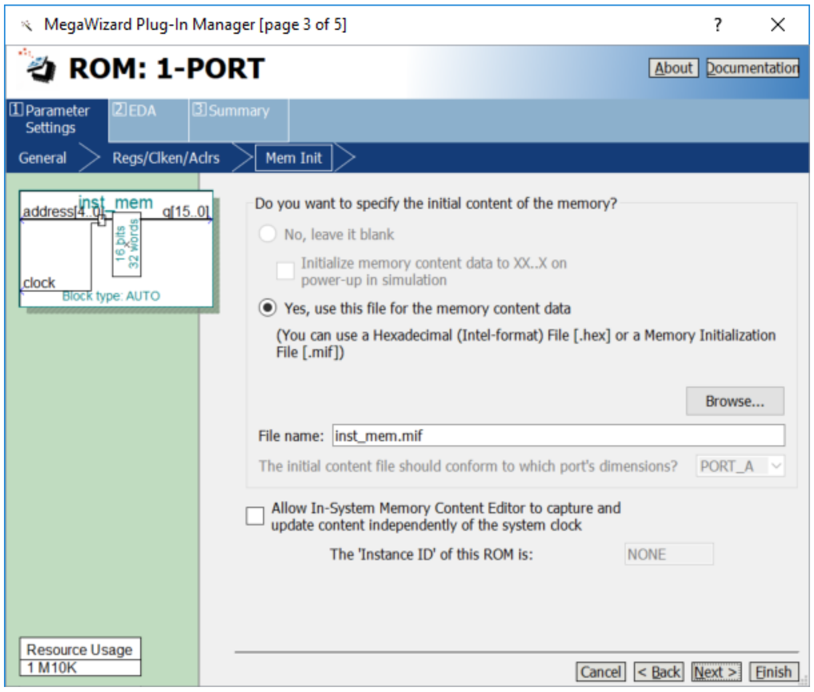
- [ ] Create one clock enable signal for each clock signal. Note: All registered ports are controlled by the enable signal(s)
- [ ] Create byte enable for port A

What is the width of a byte for byte enables? 8 bits

- [ ] Create an 'aclr' asynchronous clear for the registered ports
- [ ] Create a 'rden' read enable signal

Resource Usage
1 M10K

Figure 8: Specifying which memory ports are registered.


MegaWizard Plug-In Manager [page 3 of 5]

ROM: 1-PORT

Parameter Settings | EDA | Summary

General > Regs/Clken/Aclrs > Mem Init

Do you want to specify the initial content of the memory?

- ( ) No, leave it blank
  - [ ] Initialize memory content data to XX..X on power-up in simulation
- (●) Yes, use this file for the memory content data

(You can use a Hexadecimal (Intel-format) File [.hex] or a Memory Initialization File [.mif])

File name: inst_mem.mif

The initial content file should conform to which port's dimensions? PORT_A

- [ ] Allow In-System Memory Content Editor to capture and update content independently of the system clock

The 'Instance ID' of this ROM is: NONE

Resource Usage
1 M10K

Figure 9: Specifying a memory initialization file (MIF).

5. Use the Questa or ModelSim Simulator to test your Verilog code. Ensure that instructions are read properly out of the ROM and executed by the processor. An example of simulation results with the MIF file from Figure 10 is shown in Figure 11. The corresponding Simulator setup files are provided along with this exercise.

6. Once your simulations show a properly-working circuit, you should implement your design on a DE1-SoC (or similar) board. Use the Quartus project files that are provided along with this exercise to compile your Verilog code with the Quartus Prime software, and then download the resulting circuit to the board.

   Demonstrate the functionality of your circuit by toggling the switches on the board and observing the LEDs. Since the circuit's clock inputs are controlled by pushbutton switches, it is possible to step through the execution of instructions and observe the behavior of the circuit.

```
DEPTH = 32;
WIDTH = 16;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN
00 : 0001000000011100;     % mv  r0, #28      %
01 : 0011001011111111;     % mvt r1, #0xFF    %
02 : 0101001011111111;     % add r1, #0xFF    %
03 : 0110001000000000;     % sub r1, r0       %
04 : 0101001000000001;     % add r1, #1       %
05 : 0000000000000000;
. . . (some lines not shown)
1F : 0000000000000000;
END;
```

Figure 10: An example memory initialization file (MIF).

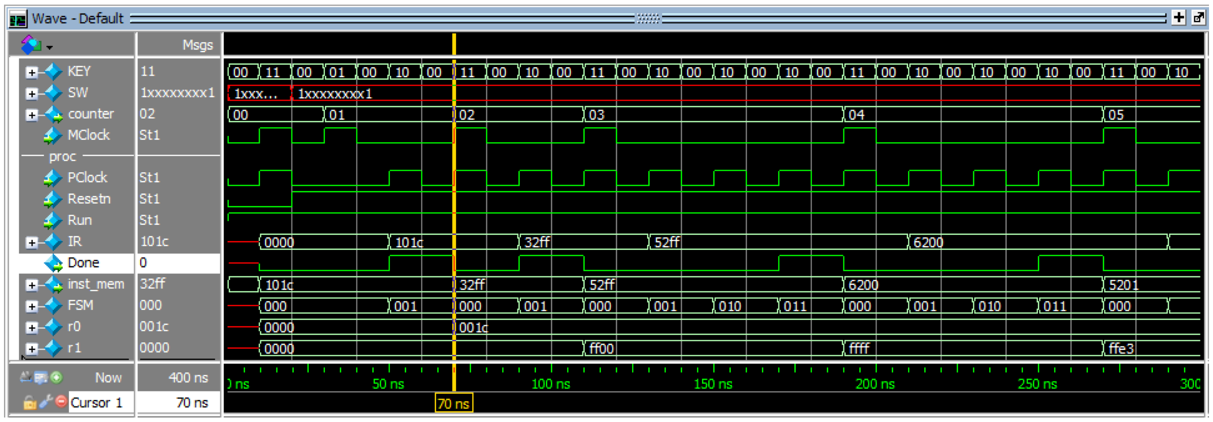


Figure 11: An example simulation output using the MIF in Figure 10.

Enhanced Processor

You can enhance your processor so that the counter in Figure 4 is no longer needed, and so that the processor has the ability to perform read and write operations using memory or other devices. These enhancements involve adding new instructions to the processor, as well as other capabilities—they are discussed in the next lab exercise.

Laboratory Exercise 10

An Enhanced Processor

In Laboratory Exercise 9 we described a simple processor. In Part I of that exercise the processor itself was designed, and in Part II the processor was connected to an external counter and a memory unit. This exercise describes subsequent parts of the processor design. The numbering of figures and tables in this exercise are continued from those in Parts I and II of the preceding lab exercise.

In this exercise we will extend the capability of the processor so that the external counter is no longer needed, and so that the processor can perform read and write operations using memory or other devices. A schematic of the enhanced processor is given in Figure 12. In this figure registers $r0$ to $r6$ are the same as in Figure 1 of Lab 9, but register $r7$ has been changed to a counter. This counter is used to provide the addresses in the memory from which the processor's instructions are read; in the preceding lab exercise, a counter external to the processor was used for this purpose. We will refer to $r7$ as the processor's *program counter* (*pc*), because this terminology is common for real processors available in the industry. When the processor is reset, *pc* is set to address 0. At the start of each instruction (in time step T_0) the value of *pc* is used as an address to read an instruction from the memory. The instruction returned from the memory is stored into the *IR* register and the *pc* is automatically incremented to point to the next instruction.

The processor's control unit increments *pc* by using the *pc_incr* signal, which is just an enable on this counter. It is also possible to load an arbitrary address into *pc* by having the processor execute an instruction in which the destination register is specified as *pc*. In this case the control unit uses pc_{in} to perform a *load* of the counter. Thus, the processor can execute instructions at any address in the memory, as opposed to only being able to execute instructions that are stored at successive addresses. The current contents of *pc*, which always has the address of the *next* instruction to be executed, can be copied into another register if needed by using a *mv* instruction.

The enhanced processor will have four new instructions, which are listed in Table 3. The *ld* (load) instruction *reads* data into register *rX* from the external memory address specified in register *rY*. Thus, the syntax [*rY*] means that the contents of register *rY* are used as an *external address*. The *st* (store) instruction *writes* the data contained in register *rX* into the memory address found in *rY*. The *and* instruction is similar to the *add* and *sub* instructions that were introduced in Lab 9. This instruction extends the adder/subtracter unit in the processor into an *arithmetic logic unit*. Besides performing addition and subtraction, it has the ability to generate a bit-wise logical AND (&) of the destination register *rX* with the second operand *Op2*. As discussed in Lab 9, the operand *Op2* can be either another register *rY*, or immediate data *#D*.

The *b{cond}* instruction in Table 3 is used to cause a processor *branch*, which means to change the program counter (*pc*) to the address of a specific instruction. The *cond* part of the branch instruction is optional and represents a *condition*. The instruction loads the address *Label* into *pc* only if the specified condition evaluates to true. An example of a condition is *eq*, which stands for *equal* (to zero). The instruction `beq Label` will load the address *Label* into *pc* if the last result produced by the arithmetic logic unit, which is stored in register G, was 0. The 3-bit register F shown in Figure 12 is required for the *b{cond}* instruction, which is discussed in more detail in Part V.

Operation	Function performed
ld rX, [*rY*]	*rX* ← [*rY*]
st rX, [*rY*]	[*rY*] ← *rX*
and rX, *Op2*	*rX* ← *rX* & *Op2*
b{cond} Label	if (*cond*), *pc* ← *Label*

Table 3: New instructions in the enhanced processor.

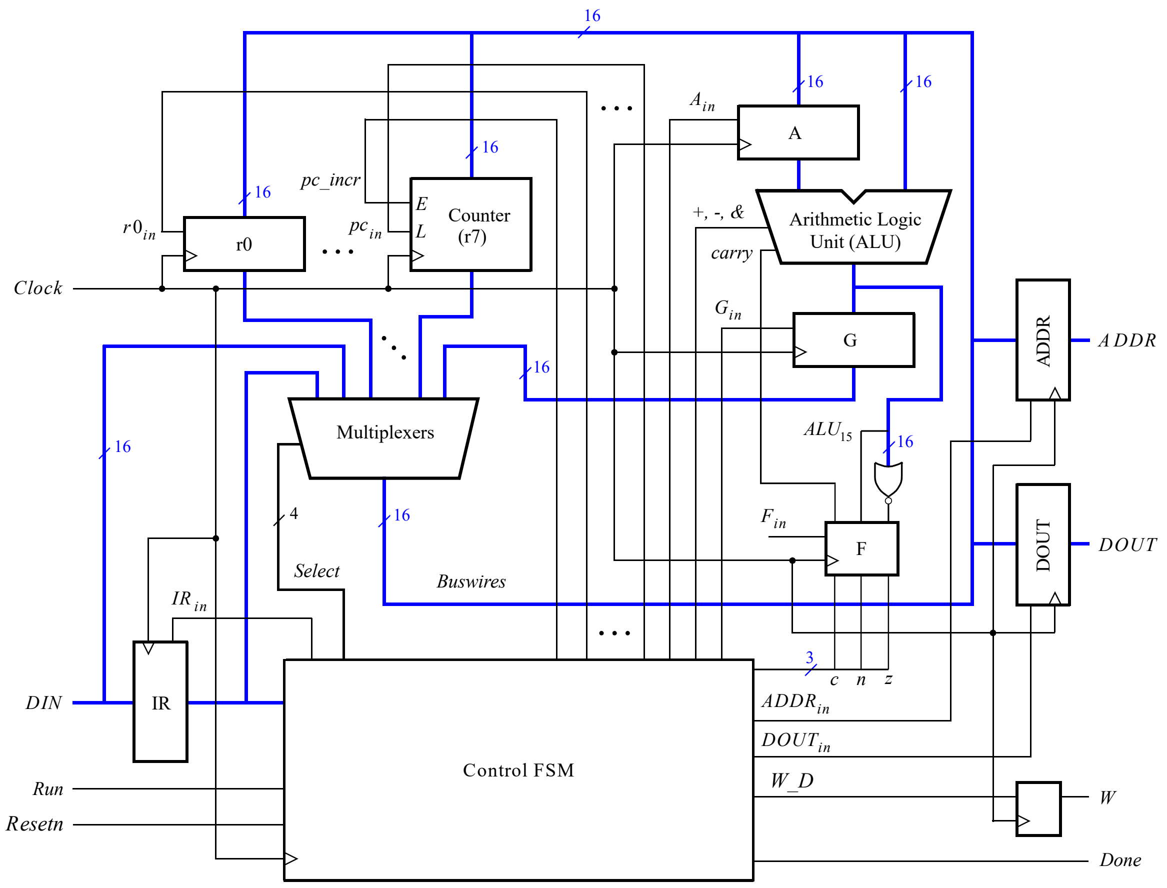


Figure 12: An enhanced version of the processor.

Recall from Lab 9 that instructions are encoded using a 16-bit format. For instructions that specify *Op2* as a register the encoding is `III0XXX000000YYY`, and if *Op2* is an immediate constant the format is `III1XXXDDDDDDDDD`. You should use these same encodings for this exercise. Assume that `III` = 100 for the *ld* instruction, 101 for *st*, and 110 for *and*. The encoding for *b{cond}* is discussed in Part V of this exercise.

Figure 12 shows two registers in the processor that are used for data transfers. The *ADDR* register is used to send addresses to an external device, such as a memory module, and the *DOUT* register is used by the processor to provide data that is to be stored outside of the processor. One use of the *ADDR* register is for reading, or *fetching*, instructions from memory; when the processor wants to fetch an instruction, the contents of *pc* are transferred across the bus and loaded into *ADDR*. This address is provided to the memory.

In addition to fetching instructions, the processor can read data at any address by using the *ADDR* register. Both data and instructions are read into the processor on the *DIN* input port. The processor can write data for storage at an external address by placing this address into the *ADDR* register, placing the data to be stored into the *DOUT* register, and asserting the output of the *W* (*Write*) flip-flop to 1.

Connecting the Processor to External Devices

Figure 13 illustrates how the enhanced processor can be connected to memory and other devices. The memory unit in the figure is 16-bits wide and 256-words deep. A diagram of this memory is given in Figure 14. It supports both read and write operations and therefore has both address and data inputs, as well as a write-enable input. As depicted in Figure 14, the memory has a clock input that is used to store the address, data, and write enable inputs into registers. This type of memory unit is called a *synchronous* static random access memory (SSRAM).

Figure 13 also includes a 9-bit output port (register) that can be used to store data from the processor. In the figure this output port is connected to a set of LEDs, like those available on the DE1-SoC, and similar, FPGA boards. To allow the processor to select either the memory unit or output port when performing a write operation, the circuit includes *address decoding*, which is done using NOR gates and AND gates. Let the processor's address lines be referred to as ADDR = $A_{15}A_{14}\cdots A_1A_0$. If the upper address lines $A_{15}A_{14}A_{13}A_{12} = 0000$, then the memory unit can be written. Figure 13 shows n *lower* address lines connected from the processor to the memory; since the memory has 256 words, then $n = 8$ and the memory's *address* port is driven by the processor address lines $A_7 \ldots A_0$. For addresses in which $A_{15}A_{14}A_{13}A_{12} = 0001$, the data written by the processor is loaded into the output port connected to *LEDs* in Figure 13.

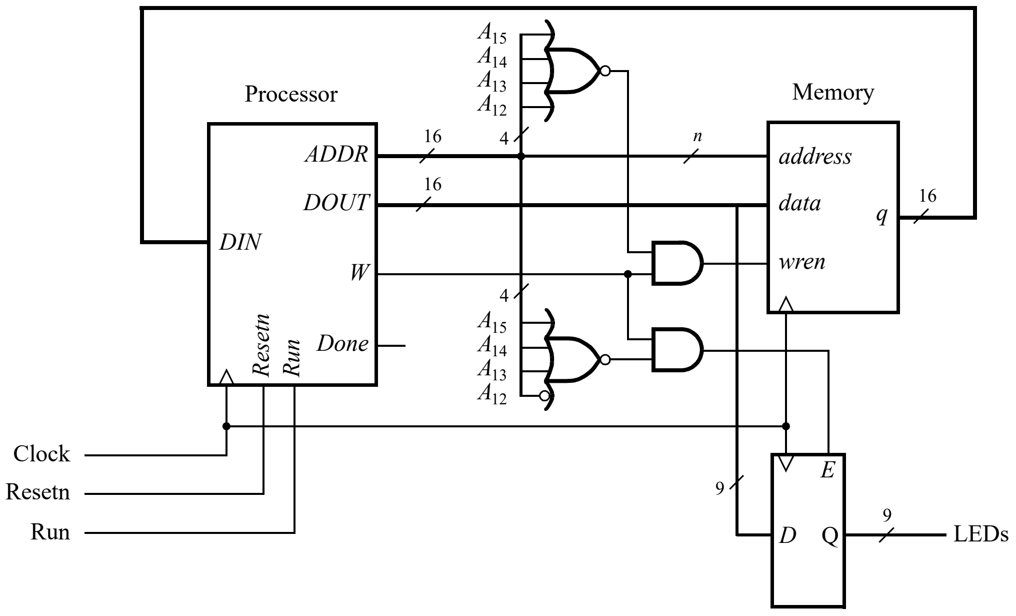


Figure 13: Connecting the enhanced processor to a memory unit and output register.

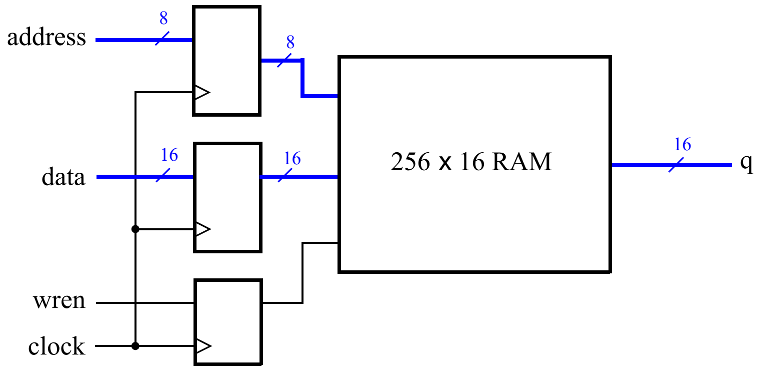


Figure 14: The synchronous SRAM unit.

Part III

Figure 15 gives Verilog code for a top-level file that you can use for this part of the exercise. The input and output ports for this module are chosen so that it can be implemented on a DE1-SoC, or similar, board. The Verilog code corresponds to the circuit in Figure 13, plus an additional input port that is connected to switches $SW_8 \ldots SW_0$. This input port can be read by the processor at addresses in which $A_{15} \ldots A_{12} = 0011$. (Switch SW_9 is not a part of the input port, because it is dedicated for use as the processor's *Run* input.) To support reading from both the SW input port and the memory unit, the top-level circuit includes a multiplexer that feeds the processor's *DIN* input. This multiplexer is described by using an `if-else` statement inside the `always` block in Figure 15.

The code in Figure 15 is provided with this exercise, along with a few other source-code files: *flipflop.v*, *inst_mem.v*, *inst_mem.mif*, and (part of) *proc.v*. The *inst_mem.v* source-code file was created by using the Quartus IP Catalog to instantiate a `RAM:1-PORT` memory module. It has a 16-bit wide read/write data port and is 256-words deep, corresponding to Figure 14.

The Verilog code in the *proc.v* file implements register *r7* as a program counter, as discussed above, and includes a number of changes that are needed to support the new *ld*, *st*, *and*, and *b{cond}* instructions. In this part you are to augment this Verilog code to complete the implementation of the *ld* and *st* instructions, as well as the *and* instruction. You do not need to work on the *b{cond}* instruction for this part.

```
module part3 (KEY, SW, CLOCK_50, LEDR);
   input [0:0] KEY;
   input [9:0] SW;
   input CLOCK_50;
   output [9:0] LEDR;

   wire [15:0] DOUT, ADDR;
   wire Done, W;
   reg [15:0] DIN;
   wire inst_mem_cs, SW_cs, LED_reg_cs;
   wire [15:0] inst_mem_q;
   wire [8:0] LED_reg, SW_reg;      // LED[9] and SW[9] are used for Run

   proc U3 (DIN, KEY[0], CLOCK_50, SW[9], DOUT, ADDR, W, Done);

   assign inst_mem_cs = (ADDR[15:12] == 4'h0);
   assign LED_reg_cs = (ADDR[15:12] == 4'h1);
   assign SW_cs = (ADDR[15:12] == 4'h3);
   inst_mem U4 (ADDR[7:0], CLOCK_50, DOUT, inst_mem_cs & W, inst_mem_q);

   always @ (*)                     // input multiplexer
      if (inst_mem_cs == 1'b1)
         DIN = inst_mem_q;
      else if (SW_cs == 1'b1)
         DIN = {7'b0000000, SW_reg};
      else
         DIN = 16'bxxxxxxxxxxxxxxxx;

   regn #(.n(9)) U5 (DOUT[8:0], Resetn, LED_reg_cs & W, CLOCK_50, LED_reg);
   assign LEDR[8:0] = LED_reg;
   assign LEDR[9] = SW[9];

   regn #(.n(9)) U7 (SW[8:0], Resetn, 1'b1, CLOCK_50, SW_reg); // Run = SW[9]
endmodule
```

Figure 15: Verilog code for the top-level file.

Perform the following:

1. Augment the code provided in *proc.v* so that the enhanced processor reads each of its instructions from the external memory, using the program counter to provide the memory address. Also, implement the *ld*, *st*, and *and* instructions. The control FSM requires six time steps for the enhanced processor, as indicated in Table 4. The first three steps are needed to *fetch* an instruction into the processor from memory. In step T_0 the program counter is copied into the *ADDR* register, so that this address will be provided to the memory. This action is accomplished by placing the program counter onto the *Buswires*, and asserting $ADDR_{in}$. Also, *pc_incr* is asserted so that the program counter will be incremented to the address in memory of the next instruction. Since the memory has a synchronous interface, as shown in Figure 14, the processor must use time-step T_1 to wait for the memory to respond. Then, the IR_{in} signal can be asserted in step T_2, so that in step T_3 the *IR* register will hold the machine code of the instruction to be executed.

   We can compare the time steps in the enhanced processor to those of the simple processor from Laboratory Exercise 9. In the enhanced processor, step T_2 serves the same function as step T_0 in the simple processor. Thus, in Table 4 the control signals asserted in steps T_2 to T_5 for the *mv*, *mvt*, *add*, and *sub* instructions are the same as those used in time steps T_0 to T_3 for the simple processor. The *and* instruction uses the same control signals as for *add* and *sub*, with one difference—for *and*, the control signal *ALU_and* is asserted in step T_4, which causes the ALU to perform the logical AND operation.

   The last two lines in Table 4 show the timing needed for *ld* and *st*. In both instructions the contents of register *rY* is transferred in step T_3 to the *ADDR* register. For *ld* the processor uses T_4 to wait for the memory to respond with data, and then step T_5 causes this data to be loaded into register *rX*. For *st*, the data in *rX* to be written to the memory is transferred to register *DOUT* in step T_4, and *W_D* (see Figure 12) is asserted to set the *W* (*write*) signal for the memory.

	T_0	T_1	T_2	T_3	T_4	T_5
mv	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rY* or *IR*, rX_{in}, *Done*		
mvt	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select IR*, rX_{in}, *Done*		
add	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rX*, A_{in}	*Select rY* or *IR*, G_{in}	*Select G*, rX_{in}, *Done*
sub	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rX*, A_{in}	*Select rY* or *IR*, *AddSub*, G_{in}	*Select G*, rX_{in}, *Done*
and	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rX*, A_{in}	*Select rY* or *IR*, *ALU_and*, G_{in}	*Select G*, rX_{in}, *Done*
ld	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rY*, $ADDR_{in}$		*Select DIN*, rX_{in}, *Done*
st	*Select pc*, $ADDR_{in}$, *pc_incr*		IR_{in}	*Select rY*, $ADDR_{in}$	*Select rX*, $DOUT_{in}$, W_D, *Done*	

Table 4: Control signals asserted in each instruction/time step.

   Test your Verilog code by using the Questa or ModelSim Simulator. Sample setup files for the Simulator, including a testbench, are provided along with the other design files for this exercise. The sample testbench first resets the processor system and then asserts the *Run* switch, SW_9, to 1. A sample program to test your processor is also provided, in a file called *inst_mem.mif*. This file represents the assembly-language program shown in Figure 16, which tests the *ld* and *st* instructions by reading the values of the SW switches and writing these values to the LEDs, in an endless loop. At the beginning of a simulation, the Simulator loads the contents of the file *inst_mem.mif* into the *inst_mem* memory module, so that the program can be executed by the processor. Examine the signals inside your processor, as well as the external LEDR values, as the program executes within the simulation.

An *Assembler* software tool, called *sbasm.py*, is provided for use with your processor. The Assembler is written in Python and is included along with the design files for this exercise. To use this Python script you need to have Python (version 3) installed on your computer. The Assembler includes a README file that explains how to install and use it. The *sbasm.py* Assembler can generate machine code for all of the processor's instructions. The provided file *inst_mem.mif* was created by using *sbasm.py* to *assemble* the program in Figure 16. As the figure indicates, you can define symbolic constants in your code by using the .define *directive*, and you can use labels to refers to lines of code, such as `MAIN`. Comments are specified in the code by using `//`. The Assembler ignores anything on a line following `//`.

```
.define LED_ADDRESS 0x10
.define SW_ADDRESS 0x30

// Read SW switches and display on LEDs
         mvt   r3, #LED_ADDRESS  // point to LED port
         mvt   r4, #SW_ADDRESS   // point to SW port
MAIN:    ld    r0, [r4]          // read SW values
         st    r0, [r3]          // light up LEDs
         mv    pc, #MAIN
```

Figure 16: Assembly-language program that uses *ld* and *st* instructions.

An example simulation result for a correctly-designed circuit is given in Figure 17. It shows the execution of the first four instructions in Figure 16.

2. Once your simulation results look correct, you can then implement your Verilog code on an FPGA board, such as the DE1-SoC. However, as we mentioned in Lab Exercise 9, as an interim step you may wish to first simulate your processor by using the graphical user interface (GUI) provided by the DESim tool—it gives a convenient way of observing the behaviour of programs running on your processor that use the lights, switches, and other features of the board. You are encouraged to make use of DESim, as it is a convenient way of debugging issues, especially when you do not have access to a physical board.

   The setup files that are needed to use DESim for this part of the exercise are provided along with its design files. When using DESim, the memory module in your design will be initialized with the contents of the *inst_mem.mif* file, so that the program in the memory can be executed by your processor. Once you start the simulation within the DESim GUI make sure to reset the circuit by clicking on the `Push Button` that corresponds to $\texttt{KEY}_0$, and assert the *Run* signal to 1 by setting the `Switch` that corresponds to $\texttt{SW}_9$. Toggle the values of the `Switches` $\texttt{SW}_{8-0}$ in the DESim GUI and observe the `LEDs`.

3. To implement your Verilog code using an actual (as opposed to *simulated*, with DESim) DE1-SoC board, you have to use the Quartus Prime software. A sample Quartus project file, *part3.qpf*, and Quartus settings file, *part3.qsf*, are provided with the exercise. After compiling your code with the Quartus software, download the resulting circuit to the board. Toggle the SW switches and observe the LEDs to test your circuit.

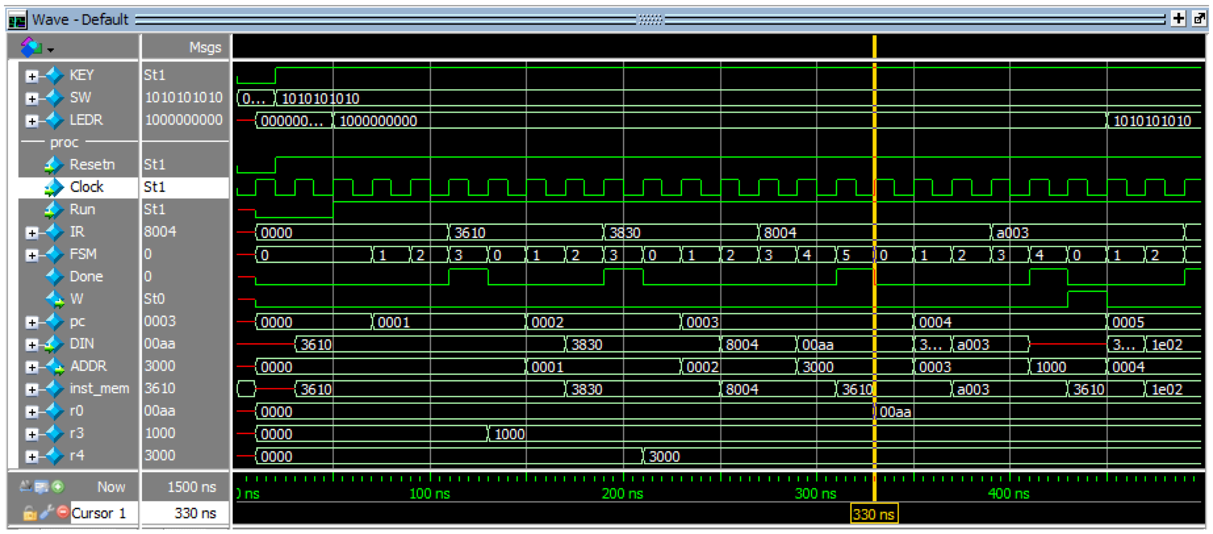


Figure 17: Simulation results for the processor.

Part IV

In this part you are to create a new Verilog module that represents an output port called *seg7*. It will allow your processor to write data to each of the six 7-segment displays that are available on a DE1-SoC, or similar, board. The *seg7* module will include six write-only seven-bit registers, one for each display. Each register should directly drive the segment lights for one seven-segment display, so that the processor can write characters onto the displays.

Perform the following:

1. A top-level file is provided for this part called *part4.v*. The top-level module has output ports for connecting to each of the 7-segment displays. For each display, segment 0 is on the top of the display, and then segments 1 to 5 are assigned in a clockwise fashion, with segment 6 being in the middle of the display.

   The *part4.v* Verilog code includes address decoding for the new *seg7* module, so that processor addresses in which $A_{15}A_{14}A_{13}A_{12} = 0010$ select this module. The intent is that address `0x2000` should write to the register that controls display *HEX0*, `0x2001` should select the register for *HEX1*, and so on. For example, if your processor writes `0` to address `0x2000`, then the *seg7* module should turn off all of the segment-lights in the *HEX0* display; writing `0x7f` should turn on all of the lights in this display.

2. You are to complete the partially-written Verilog code in the file *seg7.v*, so that it contains the required six registers—one for each 7-segment display.

3. You can compile and test your Verilog code by using the Simulator setup files that are provided for this part of the exercise. An *inst_mem.mif* file is also provided that corresponds to the assembly-language program shown in Figure 18. This program works as follows: it reads the *SW* switch port and lights up a seven-segment display corresponding to the value read on SW_{2-0}. For example, if $SW_{2-0} = 000$, then the digit 0 is shown on *HEX0*. If $SW_{2-0} = 001$, then the digit 1 is displayed on *HEX1*, and so on, up to the digit 5 which would be shown on *HEX5* if $SW_{2-0} = 101$.

4. Once your simulation results look correct, you can then implement your Verilog code on an FPGA board. As we discussed in Part III, above, you are encouraged to make use of the DESim tool for testing your processor and programs (project set-up files for DESim are included with this exercise). When using your processor, remember to reset the circuit and set $Run = 1$, and then toggle the values of the `Switches` and observe the `Seven-segment Displays`. When compiling your Verilog code with Quartus Prime, you may wish to make use of the sample Quartus project file, *part4.qpf*, and Quartus settings file, *part4.qsf*, that are included along with this exercise.

```
.define HEX_ADDRESS 0x20
.define SW_ADDRESS 0x30

// This program shows the digits 543210 on the HEX displays. Each digit has to
// be selected by using the SW switches.
MAIN:   mvt     r2, #HEX_ADDRESS      // point to HEX port
        mv      r3, #DATA             // used to get 7-segment display pattern

        mvt     r4, #SW_ADDRESS       // point to SW port
        ld      r0, [r4]              // read switches
        and     r0, #0x7              // use only SW2-0
        add     r2, r0                // point to correct HEX display
        add     r3, r0                // point to correct 7-segment pattern

        ld      r0, [r3]              // load the 7-segment pattern
        st      r0, [r2]              // light up HEX display

        mv      pc, #MAIN

DATA:   .word   0b00111111            // '0'
        .word   0b00000110            // '1'
        .word   0b01011011            // '2'
        .word   0b01001111            // '3'
        .word   0b01100110            // '4'
        .word   0b01101101            // '5'
```

Figure 18: Assembly-language program that tests the seven-segment displays.

Part V

In this part you are to enhance your processor so that it implements the *b{cond}* instruction. The *conditions* supported by the processor are called *eq*, *ne*, *cc*, *cs*, *pl*, and *mi*, which means that the variations of the branch instruction are *b*, *beq*, *bne*, *bcc*, and so on. The *b* instruction *always* branches. For example, `b MAIN` loads the address of the label `MAIN` into the program counter. The meanings of the conditional versions are explained below.

The instruction `beq LABEL` means *branch if equal* (to zero). It performs a branch (sets *pc* = `LABEL`) if the most recent result of an instruction executed using the arithmetic logic unit (ALU), which is stored in register G, was 0. Similarly, *bne* means *branch if not equal* (to zero). It performs a branch only if the contents of G are not equal to 0. The instruction *bcc* stands for *branch if carry clear*. It branches if the last add/subtract operation did *not* produce a carry-out. The opposite branch condition, *bcs*, *branch if carry set*, branches if the most recent add/sub generated a carry-out. The conditions *bpl* and *bmi* allow a branch to be taken when the value in register G is a positive or negative (2's complement) value, respectively.

To support the conditional branch instructions, you should create three *condition-code flags*, called z, n, and c in your processor. The z flag should have the value 1 when the ALU generates a result of zero; otherwise z should be 0. The n flag should be 1 when the ALU generates a result that is negative, meaning that the most-significant bit (the *sign* bit) of the result is 1; otherwise n should be set to 0. Finally, the c flag should reflect the carry-out from the ALU; this flag should be 1 when an *add* instruction generates a carry-out, or when a *sub* operation does *not* generate a borrow. Figure 12 indicates how you can implement the flags as the outputs of a three-bit register, named F. The z flag is controlled by a NOR gate that is used to check when the output of the ALU is zero, the n flag is connected to the sign-bit from the ALU's output, and a carry-out from the ALU drives the c flag. These flags are connected to the FSM controller, which should examine the flags in the appropriate clock cycles when executing a *b{cond}* instruction.

The branch instructions are encoded similarly to the *mvt* instruction introduced in Lab Exercise 9, which has the format `III1XXXDDDDDDDDD`, with `III` = 001. The format of *b{cond}* is `III0XXXDDDDDDDDD`, where `III` = 001, `XXX` gives the condition, and `DDDDDDDDD` specifies an *offset*. The condition `XXX` is encoded as *none* (always branch) = 000, *eq* = 001, *ne* = 010, *cc* = 011, *cs* = 100, *pl* = 101, and *mi* = 110.

The *offset* `DDDDDDDDD` is the 2's-complement value needed to reach the target `LABEL` relative to the current contents of the *pc* register. This offset assumes that the *pc* has already been incremented after fetching the *b{cond}* instruction from memory. For example, the instruction `HERE: b HERE` would be encoded as `0010000111111111`, where the offset is the 2's-complement value -1.

Perform the following:

1. Enhance your processor so that it implements the condition-code flags *z*, *c*, and *n* and supports the *b{cond}* instruction. Table 5 indicates the control signal timing that can be used for this instruction. Step T_3 copies the contents of the *pc* into register A, in the ALU. This step also checks whether or not the branch should be *taken*, based upon the *condition*. If the *condition* is not true, indicated in Table 5 using the syntax (!*cond*), then the *Done* signal is asserted to abort the branch instruction. But if the *condition* is satisfied, then the finite state machine will continue to step T_4. This step places the branch offset, which is in the instruction register (*IR*), onto the *Buswires* so that the ALU can add it to the value of the *pc* that was previously copied into register A. Finally, step T_5 transfers the computed branch-target address to the *pc*, so that the branch will be taken.

	T_0	T_1	T_2	T_3	T_4	T_5
b{cond}	*Select pc*, *ADDR*$_{in}$, *pc_incr*		*IR*$_{in}$	*Select pc*, A_{in}, if (!*cond*) *Done*	*Select IR*, G_{in}	*Select G*, *pc*$_{in}$, *Done*

Table 5: Control signals asserted for *b{cond}* in each time step.

   To help with testing and debugging of your processor, setup files for a Simulator are provided, including a testbench. It simulates your processor instantiated in the top-level file *part5.v*, which is the same as the one from Part IV. An example *inst_mem.mif* file is also provided, which corresponds to the program in Figure 19. This program is quite short, which makes it suitable for visual inspection of the waveforms produced by a Simulator. The program uses a sequence of instructions that tests the various conditional branches. If the program reaches the line of code labelled `DEAD`, then at least one instruction has not worked properly.

   An example of simulation output for a correctly-working processor is given in Figure 20. It shows the processor executing instructions near the start of the code in Figure 19. The instruction that is completed at simulation time 510 ns is `sub r0, #1` (`0x7001`). As shown in the figure, this instruction causes the zero flag, *z*, to become 1. The next instruction loaded into *IR*, at time 570 ns, is `bne 0x1` (`0x25fe`). This instruction does not take the branch, because $z = 1$. Finally, the instruction loaded at 650 ns is `beq 0x5` (`0x2201`), which does take the branch.

2. Once your simulation indicates a correctly-functioning processor, you can implement it on a DE1-SoC, or similar, board. As discussed above in Parts III and IV you can use the DESim tool as a convenient way of testing and debugging your design using a simulation of a board, and you can use Quartus Prime to compile and download your circuit to a physical board. The required project set-up files for both DESim and Quartus are included along with this exercise. To test your processor, you can use the assembly-language program displayed in Figure 21. It provides code that tests for the correct operation of instructions supported by the enhanced processor. If all of the tests pass, then the program shows the word PASSEd on the `Seven-segment Displays`. It also shows a binary value on the LEDs that represents the number of successful tests performed. If any test fails, then the program shows the word FAILEd on the `Seven-segment Displays` and places on the `LEDs` the address in the memory of the instruction that caused the failure. Assemble the program, which is provided in a file called *sitbooboosit.s*, by using the *sbasm.py* assembler. Store the output produced by *sbasm.py* in the file *inst_mem.mif*.

If you compile your processor system using the DESim tool, it uses the current contents of the *inst_mem.mif* file to initialize the memory. When simulating your processor make sure to reset it by clicking on the `Push Button` that corresponds to KEY$_0$. Then, set the *Run* signal to 1 by setting the `Switch` that corresponds to SW$_9$. If the *sitbooboosit* program displays FAILEd on the `Seven-segment Displays`, then you can identify the offending instruction by cross-referencing the LED pattern with the corresponding address in the file *inst_mem.mif*.

```
MAIN:   mv     r0, #2
LOOP:   sub    r0, #1          // subtract to test bne
        bne    LOOP
        beq    T1              // r0 == 0, test beq
        mv     pc, #DEAD
T1:     mvt    r0, #0xFF
        add    r0, #0xFF       // r0 = 0xFFFF
        bcc    T2              // carry = 0, test bcc
        mv     pc, #DEAD
T2:     add    r0, #1
        bcs    T3              // carry = 1, test bcs
        mv     pc, #DEAD
T3:     bpl    T4
        mv     pc, #DEAD
T4:     add    r0, #-1
        bmi    T5
        mv     pc, #DEAD
T5:     b      MAIN
DEAD:   mv     pc, #DEAD
```

Figure 19: Assembly-language program that uses various branches.

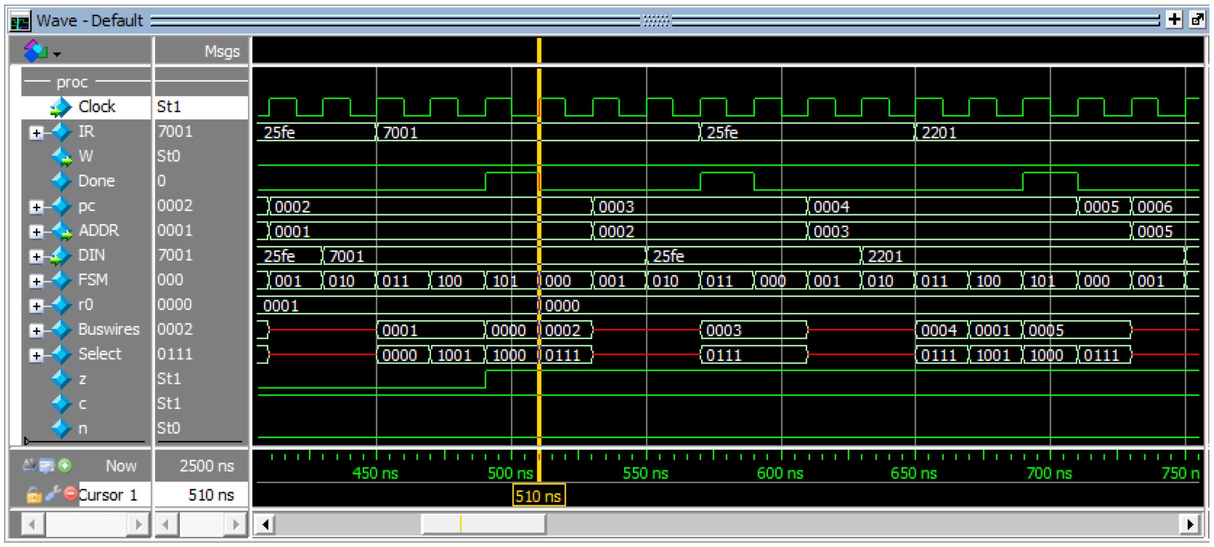


Figure 20: Simulation results for the processor.

```
.define LED_ADDRESS 0x10
.define HEX_ADDRESS 0x20

// shows on HEX displays either PASSEd or FAILEd
             mv     r2, #0       // counts each successful test
             mv     r6, #T1      // save address of next test
             sub    r0, r0       // set the z flag
// test bne and beq
T1:          bne    FAIL         // should not take the branch!
             mv     r6, #C1      // save address of next test
C1:          beq    C2           // should take the branch
             mv     pc, #FAIL    // Argh!

C2:          add    r2, #2       // count the last two successful tests
             mv     r6, #T2      // save address of next test
// test bne and beq
T2:          bne    S1           // should take the branch!
             mv     pc, #FAIL
S1:          mv     r6, #C3      // save address of next test
C3:          beq    FAIL         // should not take the branch
             add    r2, #2       // count the last two successful tests
             mv     r6, #T3      // save address of next test
             mv     r3, #-1      // r3 = 0xFFFF
             add    r3, #1       // set the c flag
// test bcc and bcs
T3:          bcc    FAIL         // should not take the branch!
             mv     r6, #C4      // save address of next test
C4:          bcs    C5           // should take the branch
             mv     pc, #FAIL    // Argh!
C5:          add    r2, #2       // count the last two successful tests
             mv     r6, #T4
             mv     r3, #0
             add    r3, r3       // clear carry flag
// test bcc and bcs
T4:          bcc    S2           // should take the branch!
             mv     pc, #FAIL
S2:          mv     r6, #C6      // save address of next test
C6:          bcs    FAIL         // should not take the branch!
             add    r2, #2       // count the last two successes
             mv     r6, #T5      // save address of next test
             mv     r3, #0
             add    r3, #-1
// test bpl and bmi
T5:          bpl    FAIL         // should not take the branch!
             mv     r6, #C7      // save address of next test
C7:          bmi    C8           // should take the branch
             mv     pc, #FAIL    // Argh!
C8:          add    r2, #2       // count the last two successful tests
             mv     r6, #T6
             mv     r3, #0
             add    r3, r3       // clear negative flag
// test bpl and bmi
T6:          bpl    S3           // should take the branch!
             mv     pc, #FAIL
```

Figure 21: Assembly-language program that tests various instructions. (Part *a*)

```
S3:          mv     r6, #C9     // save address of next test
C9:          bmi    FAIL        // should not take the branch!
             add    r2, #2      // count the last two successes
// finally, test ld and st from/to memory
             mv     r6, #T7          // save address of next test
             mv     r4, #_LDTEST
             ld     r4, [r4]
             mv     r3, #0x0A5
             sub    r3, r4
T7:          bne    FAIL             // should not take the branch!
             add    r2, #1           // incr success count

             mv     r6, #T8          // save address of next test
             mv     r3, #0x0A5
             mv     r4, #_STTEST
             st     r3, [r4]
             ld     r4, [r4]
             sub    r3, r4
T8:          bne    FAIL             // should not take the branch!
             add    r2, #1           // incr success count

             mv     pc, #PASS

// Loop over the six HEX displays
FAIL:        mvt    r3, #LED_ADDRESS
             st     r6, [r3]         // show failed test address on LEDs
             mv     r5, #_FAIL
             mv     pc, #PRINT
PASS:        mvt    r3, #LED_ADDRESS
             st     r2, [r3]         // show success count on LEDs
             mv     r5, #_PASS

PRINT:       mvt    r4, #HEX_ADDRESS // address of HEX0
             // We would normally use a loop counting down from 6 with
             // conditional branching, but in this testing code we can't
             // assume that branching even works!
             ld     r3, [r5]         // get letter
             st     r3, [r4]         // send to HEX display
             add    r5, #1           // ++increment character pointer
             add    r4, #1           // point to next HEX display
             ld     r3, [r5]         // get letter
             st     r3, [r4]         // send to HEX display
             add    r5, #1           // ++increment character pointer
             add    r4, #1           // point to next HEX display
             ld     r3, [r5]         // get letter
             st     r3, [r4]         // send to HEX display
             add    r5, #1           // ++increment character pointer
             add    r4, #1           // point to next HEX display
```

Figure 21: Assembly-language program that tests various instructions. (Part *b*)

```
                ld      r3, [r5]            // get letter
                st      r3, [r4]            // send to HEX display
                add     r5, #1              // ++increment character pointer
                add     r4, #1              // point to next HEX display
                ld      r3, [r5]            // get letter
                st      r3, [r4]            // send to HEX display
                add     r5, #1              // ++increment character pointer
                add     r4, #1              // point to next HEX display
                ld      r3, [r5]            // get letter
                st      r3, [r4]            // send to HEX display
                add     r5, #1              // ++increment character pointer
                add     r4, #1              // point to next HEX display

HERE:           mv      pc, #HERE

_PASS:          .word   0b0000000001011110     // d
                .word   0b0000000001111001     // E
                .word   0b0000000001101101     // S
                .word   0b0000000001101101     // S
                .word   0b0000000001110111     // A
                .word   0b0000000001110011     // P

_FAIL:          .word   0b0000000001011110     // d
                .word   0b0000000001111001     // E
                .word   0b0000000000111000     // L
                .word   0b0000000000110000     // I
                .word   0b0000000001110111     // A
                .word   0b0000000001110001     // F

_LDTEST: .word   0x0A5
_STTEST: .word   0x05A
```

Figure 21: Assembly-language program that tests various instructions. (Part *c*)

3. When compiling your design using the Quartus Prime software, it is possible to make use of an updated *inst_mem.mif* file without completely recompiling your Verilog code for the processor system. You can execute the Quartus command `Processing` > `Update Memory Initialization File` to include a new *inst_mem.mif* file in your Quartus project. Then, select the Quartus command `Processing` > `Start` > `Start Assembler` to produce a new programming *bitstream* for your FPGA board. Finally, use the Quartus Programmer to download the new bitstream onto your board. If the *Run* signal is asserted, your processor should execute the new program.

Part VI

Write an assembly-language program that displays a binary counter on the LED port. Initialize the counter to 0, and then increment the counter by one in an endless loop. You should be able to control the speed at which the counter is incremented by using nested delay loops; the inner loop should have a fixed delay, and the outer loop should be controlled by the SW switch settings. Changing the settings of the SW switches should cause the counter to increment more slowly/quickly on the LEDs.

Assemble your program by using the *sbasm.py* assembler, and then run it on your processor. If you are using the DESim tool, then (re)-compiling your Verilog code will initialize the processor's memory with the current contents of the *inst_mem.mif*, as mentioned before. If you are using Quartus Prime to run your processor on an FPGA board, then follow the procedure described in Step 3., above, to incorporate an updated *MIF* file that can be downloaded with your processor system onto the board.

Part VII

Augment your assembly-language program from Part VI so that counter values are displayed on the seven-segment display port rather than on the LED port. You should display the counter values as decimal numbers from 0 to 65535. The speed of counting should be controllable using the SW switches in the same way as for Part VI. As part of your solution you may want to make use of the code shown in Figure 22. This code provides a subroutine, *DIV10*, that divides the number in register $r0$ by 10, returning the quotient in $r1$ and the remainder in $r0$. Dividing by 10 is a useful operation when performing binary-to-decimal conversion. A skeleton of the required code for this part is shown in Figure 23. Since the enhanced processor does not provide a method for calling and returning from a subroutine, the code in Figures 22 and 23 uses an ad hoc method, in which register $r6$ is used to compute a return address for the *DIV10* subroutine.

As described previously, assemble your code by using the *sbasm.py* assembler tool, and then execute the new program on your processor system.

```
// subroutine DIV10
//      This subroutine divides the number in r0 by 10
//      The algorithm subtracts 10 from r0 until r0 < 10, and keeps count in r1
//      This subroutine also changes r2
//      input: r0
//      returns: quotient Q in r1, remainder R in r0
DIV10:
            mv     r1, #0                 // init Q
DLOOP:      mv     r2, #9                 // check if r0 is < 10  yet
            sub    r2, r0
            bcs    RETDIV                 // if so, then return

INC:        add    r1, #1                 // but if not, then increment Q
            sub    r0, #10                // r0 -= 10
            b      DLOOP                  // continue loop
RETDIV:
            add    r6, #1                 // adjust the return address
            mv     pc, r6                 // return results
```

Figure 22: A subroutine that divides by 10

```
.define HEX_ADDRESS 0x20
.define SW_ADDRESS 0x30

// This program shows a decimal counter on the HEX displays
MAIN:   mv      r6, pc              // return address for subroutine
        mv      pc, #BLANK          // call subroutine to blank the HEX displays
        mv      r0, #0              // initialize counter
LOOP:   mvt     r3, #HEX_ADDRESS    // point to HEX port
        ...
        ... use a loop to extract and display each digit
        ...

// Delay loop for controlling the rate at which the HEX displays are updated
        ...
        ... read from SW switches, and use a nested delay loop
        ...
        add     r0, #1              // counter += 1
        bcc     LOO                 // continue until counter overflows

        b       MAIN

// subroutine DIV10
        ...
        ... code not shown here
        ...
        add     r6, #1              // adjust the return address
        mv      pc, r6              // return results

// subroutine BLANK
        ...
        ... code not shown here
        ...
        add     r6, #1
        mv      pc, r6              // return from subroutine

DATA:   .word   0b00111111          // '0'
        ....
```

Figure 23: Skeleton code for displaying decimal digits.

Laboratory Exercise 11

A More Enhanced Processor

In Lab Exercise 10 you made enhancements to the processor from Lab 9, by including a program counter, memory interface, and the `ld`, `st`, `and`, and `b{cond}` instructions. This exercise involves further extensions to the processor design. The numbering of figures and tables in this document are continued from those in Parts I to VII of Lab Exercises 9 and 10.

For this exercise you will augment the processor architecture so that it supports subroutines and stacks, and also provides shift and rotate operations. All of the processor registers will be the same as in Lab 10, except that register $r5$ will be changed into an *up/down counter*, as illustrated in Figure 24. This figure shows only the processor registers $r4, \ldots, r7$ (*pc*) and their connections to the *Buswires* multiplexer; refer to Lab 10 to see a more complete schematic of the processor.

In assembly-language code register $r5$ can be referred to as the *stack pointer* register, *sp*. It is used as an *address* for pushing and popping data on the stack. Since it is an up/down counter, the *sp* can easily be *decremented* before a register is *pushed* onto the stack, and *incremented* after a register has been *popped* off of the stack. The processor's control unit decrements *sp* by using the *sp_decr* signal shown in Figure 24, and increments this register by using the *sp_incr* signal. These signals are just the *up/down* control inputs for the counter. Arbitrary data can also be loaded into register $r5$ (*sp*) in the same way as in Lab 10, by using the $r5_{in}$ signal.

The processor will have eight new instructions, which are listed in Table 5. The `push rX` instruction is used to store the contents of a register, *rX*, onto the stack. This instruction first decrements the *sp* (register $r5$), and then stores *rX* into memory at the address in *sp*. The `pop rX` instruction is used to load data into a register *rX* from memory at the address in *sp*. After loading this data, *sp* is then incremented.

The branch instruction, `b{cond}`, was introduced in Lab 10. This exercise defines a new type of branch instruction, `bl Label`, which is used for *subroutine linkage*. This *branch with link* instruction first copies the address of the program counter (which will already have been incremented to point to the *next* instruction after the `bl`), into register $r6$. Then, the `bl` instruction sets the program counter to the address of the subroutine, `Label`. In assembly-language code register $r6$ can be referred to as the *link register*, *lr*. To effect a *return*, a subroutine can use the instruction `mv  pc, lr`.

Operation	Function performed
push rX	*sp* ← *sp* −1, [*sp*] ← *rX*
pop rX	*rX* ← [*sp*], *sp* ← *sp* +1
bl Label	$r6$ ← *pc*, *pc* ← *Label*
cmp rX, Op2	performs *rX* − *Op2*, sets flags
lsl rX, Op2	*rX* ← *rX* << *Op2*
lsr rX, Op2	*rX* ← *rX* >> *Op2*
asr rX, Op2	*rX* ← *rX* >>> *Op2*
ror rX, Op2	*rX* ← *rX* <<>> *Op2*

Table 5: New instructions.

The `cmp` instruction is similar to the `sub` instruction that was introduced in Lab 9. This instruction performs the operation *rX* − *Op2*, but only affects the flags. The `cmp` instruction does not modify register *rX*.

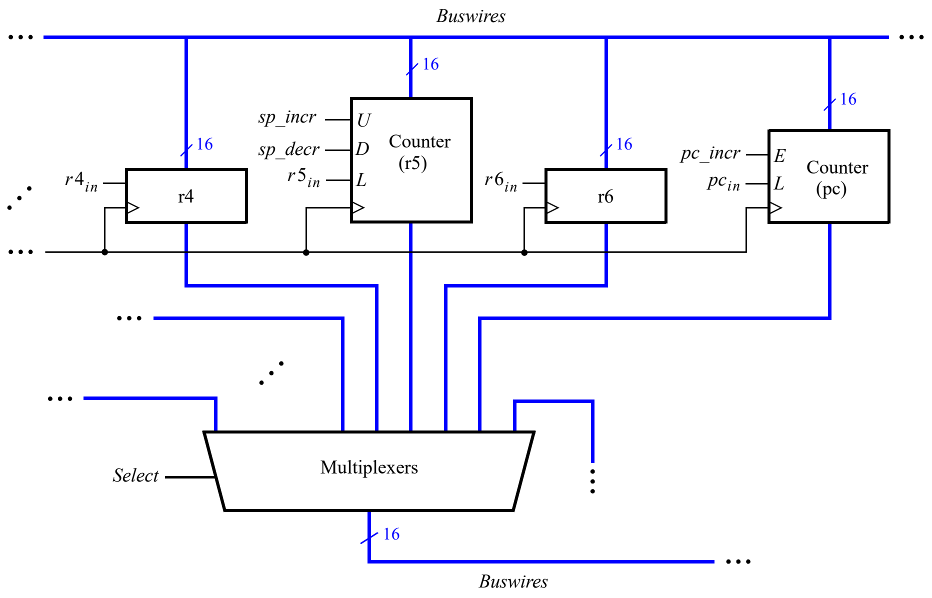


Figure 24: The stack pointer register.

Finally, the `lsl`, `lsr`, `asr`, and `ror` instructions extend the ALU in the processor to provide *shift* and *rotate* capability. The `lsl` instruction performs a logical-shift-left operation (<<). It shifts the contents of register *rX* to the left by the amount specified in *Op2*. The effect of this instruction is to perform *multiplication* by *powers of two*. The maximum possible shift amount is 15. It can be given in the form of immediate data, #*D*, or in (the four least-significant bits of) another register, *rY*. The result produce by the `lsl` instruction affects all of the processor's condition-code flags *z*, *n*, and *c*. The last bit shifted-left out of *rX* determines the value of the *c* flag.

The `lsr` instruction performs a logical-shift-right operation (>>). This means that the contents of register *rX* are shifted to the right by the amount specified in *Op2*, and each bit *shifted-in* has the value 0. The effect of this instruction is to perform an *unsigned division* by powers of two. The shift amount is specified in *Op2* in the same way as described previously for the `lsl` instruction. The `lsr` instruction affects all of the processor's condition-code flags *z*, *n*, and *c*, but the effect on the *c* flag is *undefined*.

The `asr` instruction performs an arithmetic-shift-right operation (>>>). This means that the contents of register *rX* are shifted to the right by the amount specified in *Op2*, and each bit *shifted-in* replicates the *sign-bit* of *rX*. The effect of this instruction is to perform a *signed division* by powers of two. The shift amount is specified in *Op2* in the same way as described previously. The `asr` instruction affects all of the processor's condition-code flags *z*, *n*, and *c*, but the effect on the *c* flag is *undefined*.

The `ror` instruction performs a rotate-right operation (<<>>). It shifts the contents of register *rX* to the right in a circular fashion, so that each bit shifted out of the least-significant-bit of *rX* is shifted into the most-significant bit. The shift amount is specified in *Op2* in the same way as described previously. The `ror` instruction affects all of the processor's condition-code flags *z*, *n*, and *c*, but the effect on the *c* flag is *undefined*.

Instruction Encodings

Recall from Labs 9 and 10 that instructions are encoded using a 16-bit format. For instructions that have two operands, when *Op2* is a register the encoding is `III0XXX000000YYY`, and when *Op2* is an immediate #*D* constant the format is `III1XXXDDDDDDDDD`. The `ld` and `st` instructions are encoded as `1000XXX000000YYY` and `1010XXX000000YYY`, respectively. You should encode the `pop` instruction similarly to `ld`, with the encoding `1001XXX000000101`. Also, encode `push` similarly to `st`, using the encoding `1011XXX000000101`. Notice that for both `push` and `pop` the `YYY` field is hard-coded to correspond to the stack pointer register, *r*5.

Recall from Lab 10 that the `b{cond}` instruction uses the `XXX` field to encode a *condition*, where `XXX` = 000 (*none*), 001 (*eq*), 010 (*ne*), and so on. Implement the `bl` instruction by using the previously-unassigned code `XXX` = 111.

You should implement the `cmp` instruction similarly to the `add`, `sub`, and `and` instructions. Use the previously-unassigned code `III = 111`; if *Op2* is a register, then `cmp` is encoded as `1110XXX000000YYY`, and if *Op2* is *#D*, then `cmp` is encoded as `1111XXXDDDDDDDDD`. For the shift/rotate instructions you should also use the code `III = 111`, as follows. When *Op2* is a register, encode these instructions as `1110XXX10SS00YYY`, and when *Op2* is *#D* encode them as `1110XXX11SS0DDDD`. In these encodings `SS` specifies the type of shift/rotate, where `SS` = 00 (`lsl`), `01` (`lsr`), `10` (`asr`), or `11` (`ror`). Note that the instruction `cmp rX,rY` and the various shift/rotate instructions share the most-significant digits of their encodings (bits 15-9), which are `1110XXX`. However, for this `cmp` instruction the next six digits (bits 8-3) are `000000`, whereas for the shift/rotate instructions these bits are never all zeros. To differentiate between `cmp rX,rY` and the shift/rotate instructions it is sufficient to examine the digit in bit-position 8.

Barrel Shifter

To implement the required shift and rotate operations for the `lsl`, `lsr`, `asr`, and `ror` instructions, you need to add a 16-bit *barrel shifter* to the processor's ALU. Register *A* should serve as the data input for the barrel shifter and the shift amount should be provided by *Op2*. The FSM has to control the ALU such that its output comes from the barrel shifter when needed, and the FSM has to control the barrel shifter so that it produces the required type of shift, or rotate, operation. Example Verilog code for a barrel shifter is provided in Figure 25. You should augment your ALU using (a modified version of) the code given inside the always block in this module.

```
module barrel (shift_type, shift, data_in, data_out);
    input wire [1:0] shift_type;
    input wire [3:0] shift;
    input wire [15:0] data_in;
    output reg [15:0] data_out;

    parameter lsl = 2'b00, lsr = 2'b01, asr = 2'b10, ror = 2'b11;

    always @(*)
        if (shift_type == lsl)
            data_out = data_in << shift;
        else if (shift_type == lsr)
            data_out = data_in >> shift;
        else if (shift_type == asr)
            data_out = {{16{data_in[15]}},data_in} >> shift;     // sign extend
        else // ror
            data_out = (data_in >> shift) | (data_in << (16 - shift));
endmodule
```

Figure 25: Verilog code for a barrel shifter.

Finite State Machine Timing

To implement each of the new instructions, you will need to augment the finite state machine for your processor. Table 6 indicates how the required signals may be asserted in each time step to implement the instructions in Table 5. Following the style used in Labs 9 and 10, in this table *Select pc* means "put the program counter onto the *Buswires*," *Select #D* means "put the sign-extended immediate data that is in the instruction register (*IR*) onto the *Buswires*," *W_D* means "assert the input to the flip-flop that provides the *write* signal for the memory," and *do_shift* means "set the control signal on the ALU such that its output will be provided by the barrel shifter."

	T_0	T_1	T_2	T_3	T_4	T_5
push	*Select pc*, *ADDR$_{in}$*, *pc_incr*	*Wait*	*IR$_{in}$*	*sp_decr*	*Select rY*, *ADDR$_{in}$*	*Select rX*, *DOUT$_{in}$*, *W_D*, *Done*
pop	*Select pc*, *ADDR$_{in}$*, *pc_incr*	*Wait*	*IR$_{in}$*	*Select rY*, *ADDR$_{in}$*, *sp_incr*	*Wait*	*Select DIN*, *rX$_{in}$*, *Done*
bl	*Select pc*, *ADDR$_{in}$*, *pc_incr*	*Wait*	*IR$_{in}$*	*Select pc*, *A$_{in}$*, *r6$_{in}$*	*Select #D*, *G$_{in}$*	*Select G*, *pc$_{in}$*, *Done*
cmp	*Select pc*, *ADDR$_{in}$*, *pc_incr*	*Wait*	*IR$_{in}$*	*Select rX*, *A$_{in}$*	*Select rY* or *#D*, *AddSub*, *F$_{in}$*, *Done*	
lsl, lsr asr, ror	*Select pc*, *ADDR$_{in}$*, *pc_incr*	*Wait*	*IR$_{in}$*	*Select rX*, *A$_{in}$*	*Select rY* or *#D*, *do_shift*, *G$_{in}$*, *F$_{in}$*	*Select G*, *rX$_{in}$*, *Done*

Table 6: Control signals asserted in each instruction/time step.

Part VIII

You should connect your processor to a memory and I/O devices in the same way as for Lab 10, including the instruction memory, LED, SW, and seg7 (HEX5-0) I/O devices. The design files for this exercise include a suitable top-level file for your use, called *part8.v*, and a new *inst_mem.v* file for the instruction memory. In this design the instruction memory has been increased from the previous size of 256 words to 4K words. Thus, the processor is connected to the memory using 12 address lines, rather than eight. Other than this change, *part8.v* is the same as the top-level file provided in Part V of Lab 10 (*part5.v*).

To assemble code for your processor, you can use the *sbasm.py* assembler. It supports all of the instructions in the processor, including `push`, `pop`, `bl`, `cmp`, `lsl`, `lsr`, `asr`, and `ror`. The Assembler assumes by default that your machine code will not require more than 256 words—to use all of the new 4K memory you have to include the directive

```
DEPTH 4096
```

at the start of your assembly-language program. This directive will cause *sbasm.py* to produce a *memory initialization file* (MIF) that supports up to 4K words of machine code.

Perform the following:

1. First, extend your processor (from Part V of Lab 10) to provide support for subroutines, by implementing the `push`, `pop`, and `bl` instructions. Make sure to change register $r5$ into a counter that has the *up*, *down*, and *load* controls shown in Figure 24. Test your Verilog code by using the Questa or ModelSim Simulator. Sample setup files for the Simulator, including a testbench, are provided along with the design files for this exercise. The sample testbench first resets the processor system and then asserts the *Run* switch, *SW*$_9$, to 1. A simple example of assembly language code that can be used to test your subroutine support is given in Figure 26. The first line of code initializes the stack pointer, *sp*, to the value $1000_{16} = 4096_{10}$, which places the stack at the bottom of the 4K memory module. The next line of code in Figure 26 uses a syntax, *=D*, that is supported by the *sbasm.py* assembler for initializing a register with a 16-bit value. The instruction

   ```
   mv    r4, =0x0F0F
   ```

   is implemented by the assembler using the *two* instructions

   ```
   mvt   r4, #0x0F
   add   r4, #0x0F
   ```

   This *=D* syntax can be used as a convenient way of initializing a register to any 16-bit value.

```
START:  mvt    sp, #0x10       // sp = 0x1000 = 4096
        mv     r4, =0x0F0F
        push   r4
        bl     SUBR
        pop    r4
END:    b      END

SUBR:   sub    r4, r4
        mv     pc, lr
```

Figure 26: An assembly-language program to test subroutine support.

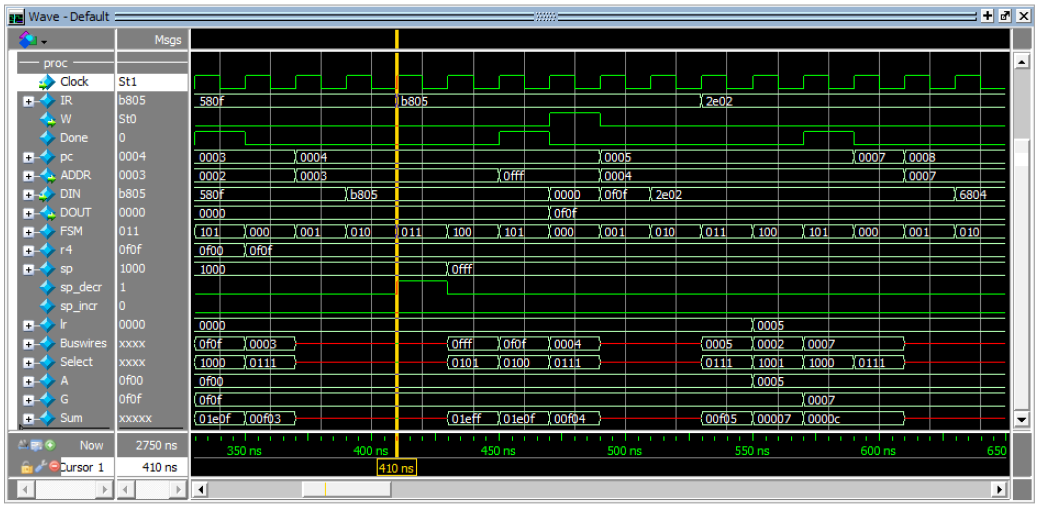


Figure 27: Simulation results for code in Figure 26. (Part *a*)

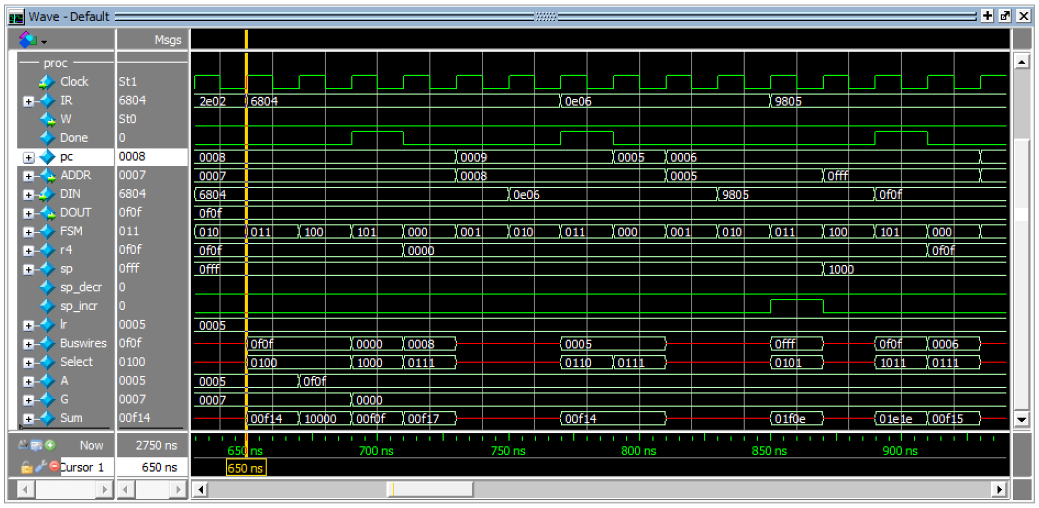


Figure 27: Simulation results for code in Figure 26. (Part *b*)

An example of simulation results produced by executing the code in Figure 26 is displayed in Figure 27.

In part (a) of the figure the first two lines of code (three instructions) in the program have already been executed, so that the stack pointer *sp* = 0x1000 and register $r4$ = 0x0F0F. At 410 *ns* in simulation time the processor loads the instruction at address 3, which is `push r4` (`0xb805`). As shown in the simulation results the signal *sp_decr* is asserted to decrement *sp* to `0xFFF`, and then the contents of register $r4$ are written to the memory. At 530 ns in simulation time the instruction `bl SUBR` (`0x2E02`) is loaded into *IR*, from address 4. First, this instruction sets the link register *lr* to the value 5 (the subroutine return address), and then sets *pc* = 7, which is the address of `SUBR`.

Figure 27b continues the simulation results from Part (a). The first instruction of the `SUBR` subroutine, `sub r4,r4` (`0x6804`), is loaded into *IR* at 650 *ns*. As shown in the simulation, this instruction results in $r4$ = 0. Then, the subroutine return instruction `mv pc,lr` (`0x0E06`) is executed to return control to the address in *lr*, which is 5. The instruction at address 5 is `pop r4` (`0x9805`). It first reads from the memory at the address in *sp*, which is 0xFFF, and then asserts the *incr_sp* signal, resulting in *sp* = 0x1000. Finally, the data read from the memory is used to restore the value $r4$ = 0x0F0F.

2. Next, you should add the `cmp` instruction, which is similar to `sub`, as well as the shift and rotate instructions. Augment your ALU to include the barrel-shifter capability illustrated in Figure 25. Simulation results for a correctly-designed processor, executing code in Figure 28, are displayed in Figure 29. In Part (a) of the figure the first three instructions in the code have already been executed, so that register $r0$ = 4 and $r4$ = 0x0F0F. At 350 *ns* in simulation time the processor fetches the instruction at address 3, which is `lsl r4, #1`. In time step T_3 of this instruction (which is indicated as `011` in the waveform labeled *FSM*) register $r4$ is placed onto *Buswires* so that it can be copied into register A, in the ALU. Then, in time step T_4 the immediate data, which is in *IR* and specifies the shift amount, is placed onto *Buswires*. The *do_shift* signal is asserted, so that the ALU's *Sum* output is driven by the barrel shifter. It uses bits 3 - 0 from *Buswires* as the shift amount for the `lsl` instruction. The barrel shifter generates the result *Sum* = 0x1E1E, which is loaded into $r4$ at the end of the instruction.

   The next instruction executed in Figure 29a is `lsr r4, #1`. It reverses the previous `lsl` operation, resulting in $r4$ = 0x0F0F. At 590 *ns* in simulation time, the `lsl r4, r0` instruction is executed. Steps $T_0 - T_3$ of this instruction appear in part (a) of Figure 29, and the remaining time steps are shown in Figure 29b. Observe in time step T_4 that register $r0$ is placed onto *Buswires*, because the shift amount (4) is contained in this register. This `lsl` instruction results in $r4$ = 0xF0F0. The final two instructions in the simulation are `asr r4, #1`, which produces $r4$ = 0xF878, and `ror r4, r0`, which results in $r4$ = 0x8F87.

```
START:  mv     r0, #4
        mv     r4, =0x0F0F

        lsl    r4, #1         // lsl with Op2 = #D
        lsr    r4, #1         // lsr with Op2 = #D
        lsl    r4, r0         // lsl with Op2 = rY
        asr    r4, #1         // asr with Op2 = #D
        ror    r4, r0         // ror with Op2 = rY

END:    b      END
```

Figure 28: A program to test shift and rotate instructions.

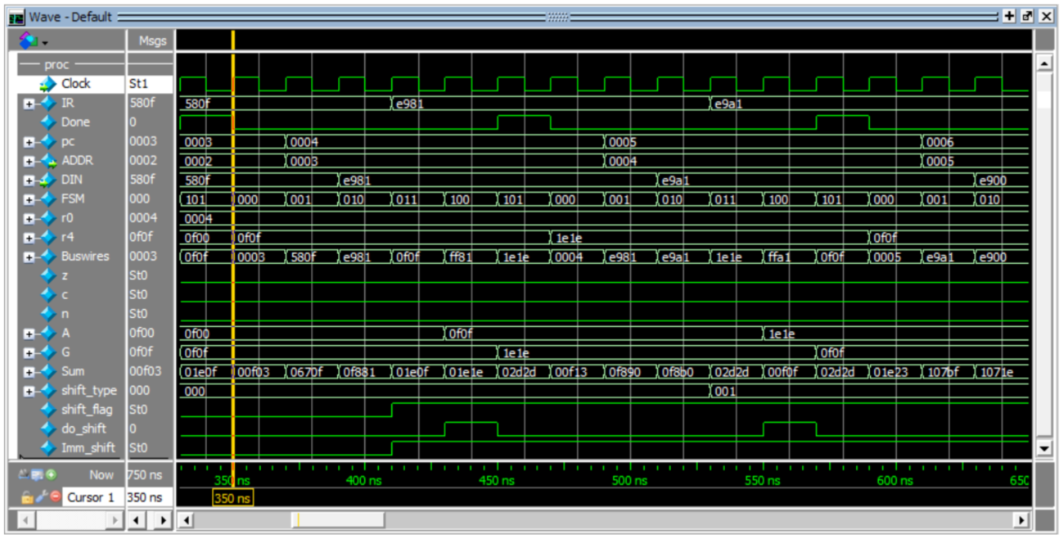


Figure 29: Simulation results for code in Figure 28. (Part a)

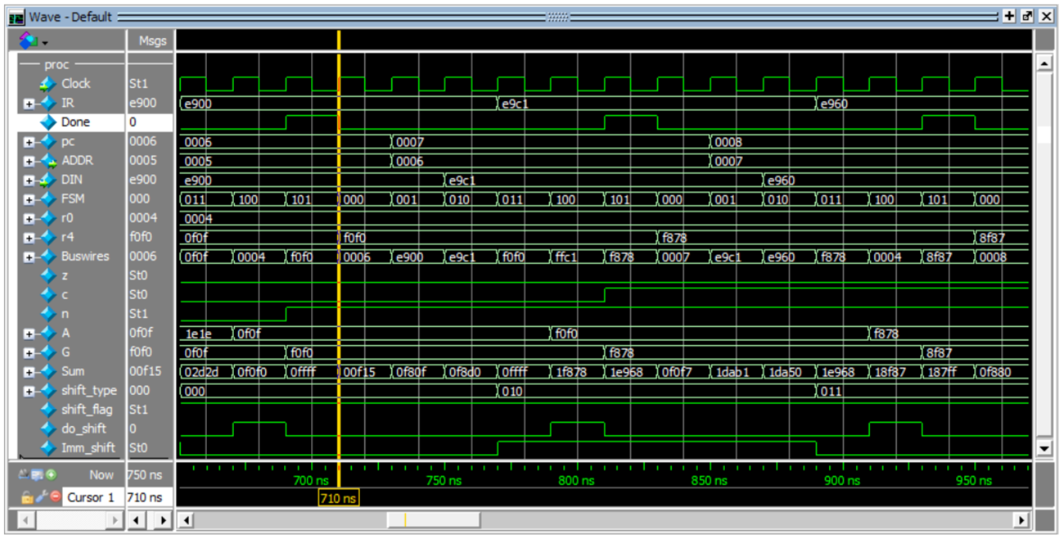


Figure 29: Simulation results for code in Figure 28. (Part b)

3. An example of a subroutine, called `REG`, that you may find useful is given in Figure 30. This subroutine is passed one parameter, in register $r0$. The purpose of the subroutine is to display the contents of this register, in hexadecimal, on HEX3-0. This code utilizes the `push`, `pop`, `cmp`, and `lsr` instructions, and also uses the *lr* to return from the subroutine.

   A main program that calls the `REG` subroutine is provided in Figure 31. This program tests the various shift and rotate operations, as selected by the `SW` switches. The type of shift/rotate is chosen by setting the switches $\texttt{SW}_{6-5}$ = 00 (`lsl`), 01 (`lsr`), 10 (`asr`), or 11 (`ror`). The shift amount is chosen by setting $\texttt{SW}_{3-0}$. The pattern shifted, $r0$ = 0xF0F0, is loaded at the start of the program. This pattern is reloaded into $r0$ if a shift operation results in $r0$ = 0x0000 or $r0$ = 0xFFFF.

An assembly-language source-code file, called *shift_test.s*, which includes the code in Figures 30 and 31 is provided as part of the design files for this exercise. Assemble this code using *sbasm.py* and ensure that it works with your processor. As mentioned in Labs 9 and 10, you may want to make use of the DESim tool while developing and debugging your processor. A video demonstration of the program in Figure 31 running on a correctly-working processor using the *DESim* tool can be found at the URL:

https://youtu.be/0k5GPGg_Vto

4. Write some assembly-language code of your choosing that demonstrates the operations supported by your processor. You should make use of various I/O devices that are available to your processor, such as the LEDR lights, SW switches, and HEX displays. In general, try to conceive of a program that does something interesting and challenging. You should be able to demonstrate your code working properly on a DE1-SoC, or similar board, but you may want to make use of *DESim* while developing/debugging your code.

```
.define HEX_ADDRESS 0x2000

// subroutine that displays register r0 (in hex) on HEX3-0
REG:    push  r1
        push  r2
        push  r3

        mv    r2, =HEX_ADDRESS  // point to HEX0

        mv    r3, #0            // used to shift digits
DIGIT:  mv    r1, r0            // the register to be displayed
        lsr   r1, r3            // isolate digit
        and   r1, #0xF          // "     "  "  "
        add   r1, #SEG7         // point to the codes
        ld    r1, [r1]          // get the digit code
        st    r1, [r2]
        add   r2, #1            // point to next HEX display
        add   r3, #4            // for shifting to the next digit
        cmp   r3, #16           // done all digits?
        bne   DIGIT

        pop   r3
        pop   r2
        pop   r1
        mv    pc, lr

SEG7:   .word 0b00111111       // '0'
        .word 0b00000110       // '1'
        .word 0b01011011       // '2'
        .word 0b01001111       // '3'
        .word 0b01100110       // '4'
        .word 0b01101101       // '5'
        .word 0b01111101       // '6'
        .word 0b00000111       // '7'
        .word 0b01111111       // '8'
        .word 0b01100111       // '9'
        .word 0b01110111       // 'A' 1110111
        .word 0b01111100       // 'b' 1111100
        .word 0b00111001       // 'C' 0111001
        .word 0b01011110       // 'd' 1011110
        .word 0b01111001       // 'E' 1111001
        .word 0b01110001       // 'F' 1110001
```

Figure 30: A useful subroutine.

```
DEPTH 4096
.define LED_ADDRESS 0x1000
.define SW_ADDRESS  0x3000

START: mv   sp, =0x1000        // initialize sp

MAIN:  mv   r0, =0x9010
       bl   REG                // display r0 on HEX3-0
       bl   DELAY
LOOP:  mv   r1, =SW_ADDRESS
       ld   r1, [r1]
       mv   r2, =LED_ADDRESS
       st   r1, [r2]
       mv   r2, r1
       lsr  r2, #5             // get shift type (SW bits 6:5)

       cmp  r2, #0b00
       bne  LSR
       lsl  r0, r1
       b    CONT
LSR:   cmp  r2, #0b01
       bne  ASR
       lsr  r0, r1
       b    CONT
ASR:   cmp  r2, #0b10
       bne  ROR
       asr  r0, r1
       b    CONT
ROR:   ror  r0, r1

CONT:  bl   REG
       bl   DELAY

       cmp  r0, #0
       beq  MAIN
       cmp  r0, #-1
       beq  MAIN

END:   b    LOOP

// Causes a delay that works well when using DESim. For an actual
// DE1-SoC board, use a longer delay!
DELAY: push r1
       mvt  r1, #0x04        // r2 <- 2^10 = 1024
WAIT:  sub  r1, #1
       bne  WAIT
       pop  r1
       mv   pc, lr
```

Figure 31: A program that test shift/rotate operations.