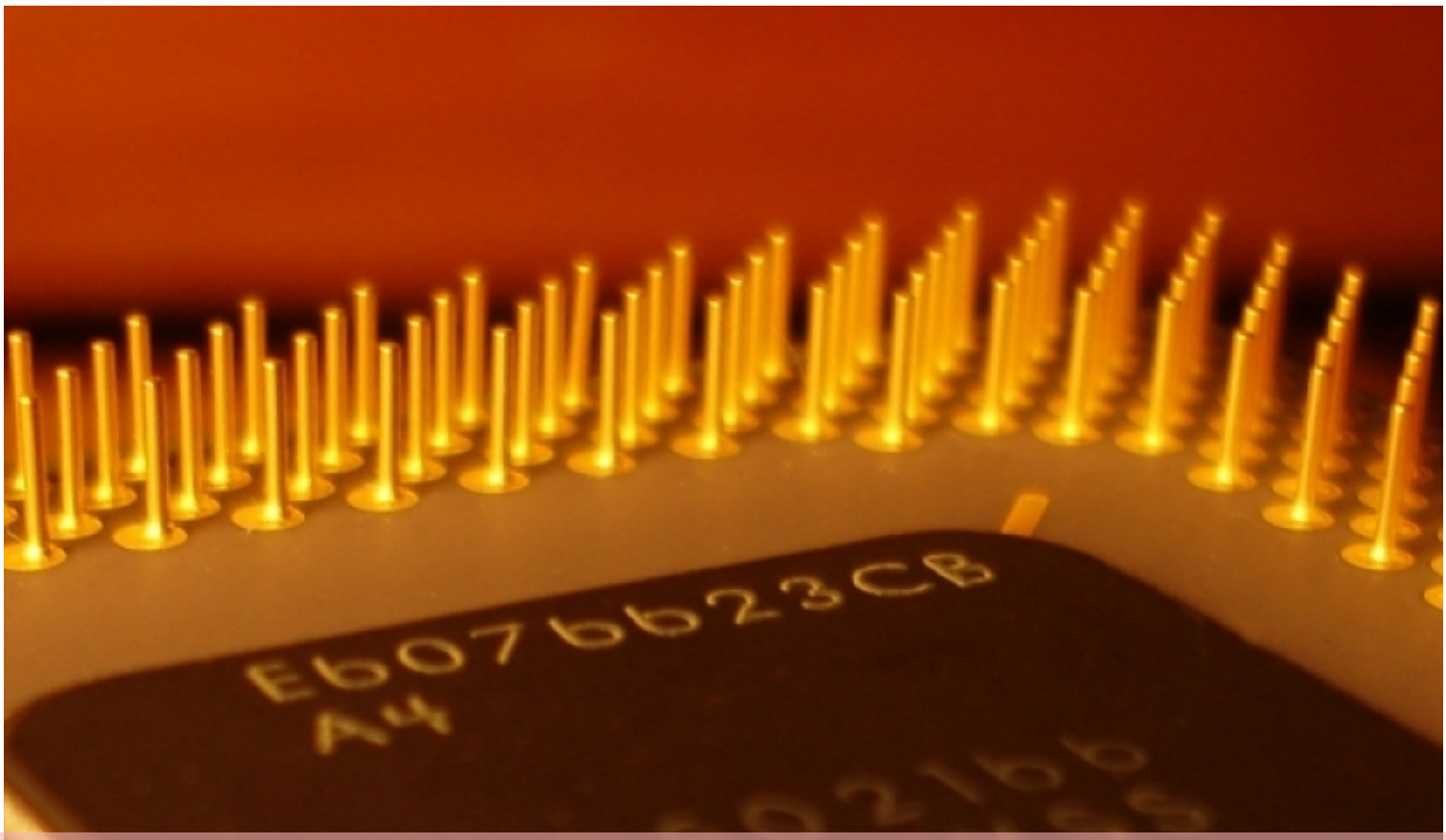


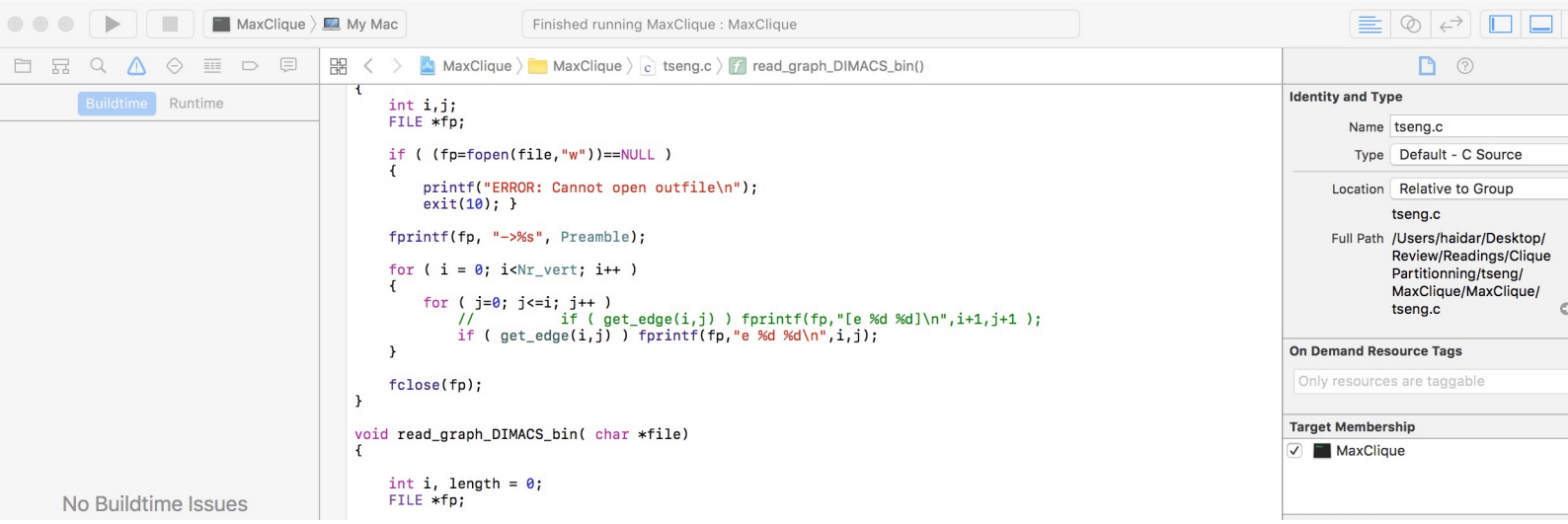
Hands-On Computer Organization Lab

With C and Verilog

Haidar M. Harmanani

HARMANANI.GITHUB.IO

First printing, March 2016



Contents

1. Combinational Logic Using Quartus Prime

"I am now about to set seriously to work upon preparing for the press an account of my theory of Logic and Probabilities which in its present state I look upon as the most valuable if not the only valuable contribution that I have made or am likely to make to Science and the thing by which I would desire if at all to be remembered hereafter"

—George Boole

An adder is a circuit whose output is the binary sum of its inputs, and are essential part of a digital system. A full adder has three inputs (A, B, C_{in}) and two outputs (Sum, C_{out}). Inputs A and B represent bits of the two binary numbers that are being added, and Sum represents a bit of the resulting sum. C_{in} and C_{out} are the carry-in and the carry-out, respectively. The truth table along with the corresponding circuit is shown Figure 1.1.

A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

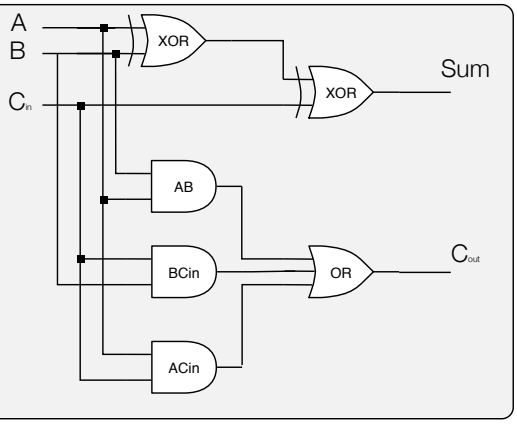


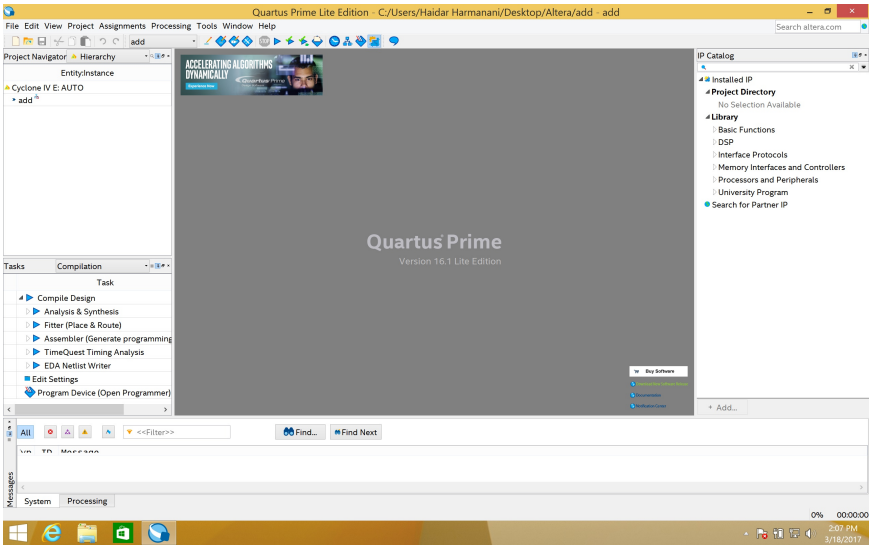
Figure 1.1: Binary Adder: Truth Table and Corresponding Logic Circuit.

Lab 1.1 Adder Circuit.. Derive the logic equations for the two outputs (Sum, C_{out}) as shown in Figure 1.1. Draw a schematic diagram for the one-bit adder using `Quartus Prime`. Use two *XORs*, three *ANDs*, and one *OR* gate.

Start a new project using `File > New Project Wizard`. Specify the name of the file and the directory.

2. Select the `Cyclone IV E` device as the target device family and select the device called EP4CE115F29C7N which is the FPGA used on `Altera's DE2-115` board.
3. Create a new circuit using the Schematic editor: `File > New` and then select `Block Diagram/Schematic File`. Put a checkmark in the box `Add file to current project`
4. Import the logic gates. Double-click on the blank space in the `Graphic Editor` window, or click on the icon in the toolbar that looks like an `AND` gate. A pop-up box will appear.
5. Expand the hierarchy in the Libraries box. First expand libraries, then expand the library primitives, followed by expanding the library logic which comprises the logic gates. Select the needed gates.
6. Wire up the one-bit adder according to the schematic diagram you developed in part (2)
7. Simulate the one-bit adder and prove that it works.

■



Lab 1.2 Simulation Using University Program VWF.. Before simulation you must perform *Analysis and Synthesis*. Once done, proceed as follows:

Go to `File` → `New` → `Verification/Debugging Files` → `University Program VWF`[a]

2. When the `Simulation Waveform Editor` window appears click `Edit` → `Insert`
3. When the `Insert Node or Bus` window appears, click on `Node Finder`
4. When the `Node Finder` window appears, click on List.
5. Select the node names and hit the > button to move them to selected nodes. Hit `OK`.
6. Hit OK to close `Insert Node or Bus`
7. You should get a window that shows the inputs as `0`, and the outputs as undefined (since we have not run the simulation).

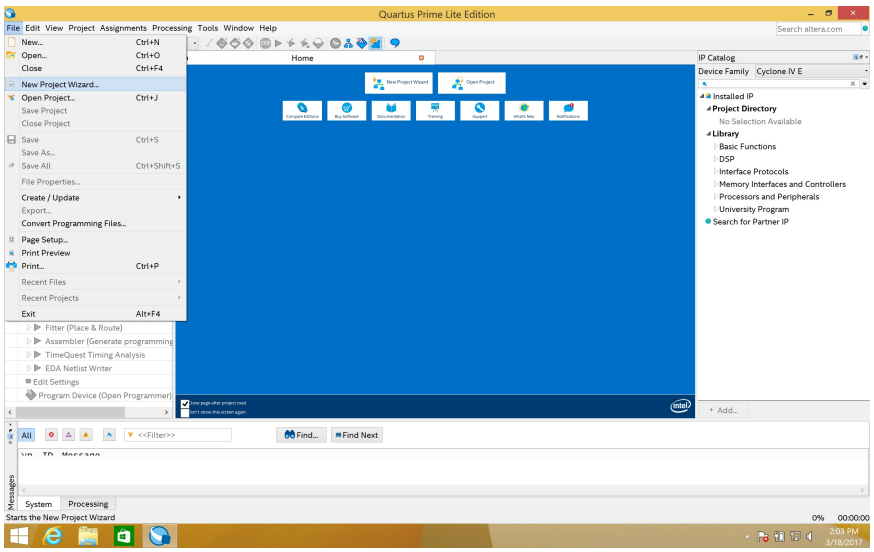


Figure 1.2: Step 1: Create a Project

8. We first define the inputs. Select `input 1` on the left side of the page and then choose `Overwrite clock` from the menu above the timing diagram. Choose a 10 ns period and 50% duty cycle clock. Hit `OK`.
9. Now do the same for `input 2` but make it a 20 ns period (half as fast as before).
10. This leaves a lot of oscillations of the inputs, so go to `Edit` → `End Time` and set the end time to 80 ns.
11. Go to `Simulation` → `Run Functional Simulation`. The output should look like below. Recall that we are adding three values. The resulting sum is `input 1` and carry out is `input 2`. Are the results as expected?

■

[a] VWF = vector waveform file

1.1 The Arithmetic Logic Unit (ALU)

The Arithmetic and Logic Unit (ALU), introduced in Lab 2, is a basic building block in every modern processor. In Lab 2 we tackled the design of the ALU using *C* and *Python*. In this Lab, we will tackle the design and simulation of a one-bit slice of the design shown in Figure 1.9 using logic gates. We will also explore the design of the ALU using the `Quartus lpm` library.

1.1.1 Operations

The ALU can perform the following arithmetic and logical operations: `Add`, `Subtract`, `AND`, and `OR`. Since the focus of this lab is combinational logic, we will ignore all sequential elements, that is registers `A`, `B`, `Ovf`, `Z`, and `Result` as these will be introduced in the next lab.

1.1.2 Combinational Building Blocks

The ALU has two combinational units: an *adder/subtractor* and a *multiplexer*. The *add/sub* unit can add or subtract two numbers based on the `OpCode` selector which determines the type of operation the ALU is executing. The *multiplexer* is a building block that is used to select one output from `n` different inputs. The ALU has two multiplexers. The first multiplexer, *mux1*, selects either the `AND`

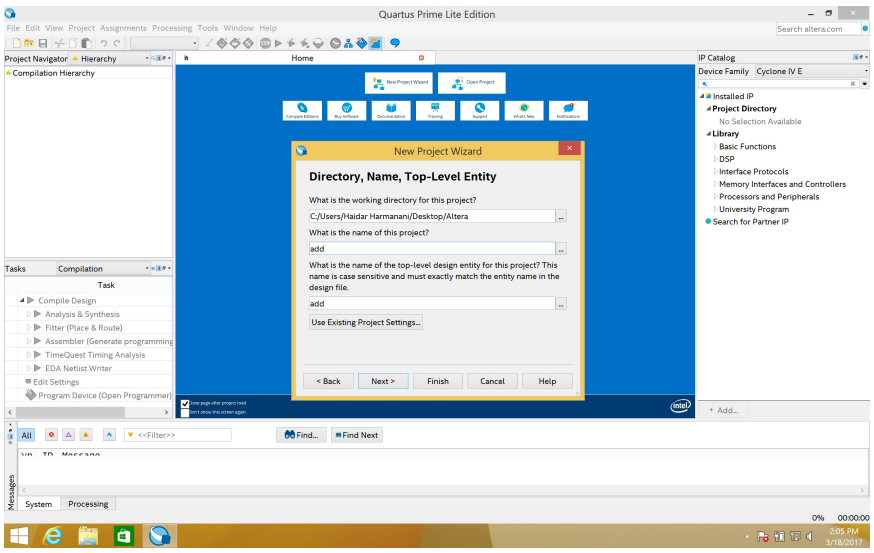


Figure 1.3: Step 1: Create a Project

or the *OR* operation. The second multiplexer, *mux2* selects either the output of the *Add/Sub* unit or the output of the *mux1*. The output is saved in the `result` register.

Figure 1.11*a* 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.

Exercise 1.1 Design the *add/sub* and the *multiplexer* units at the logic level. Use *Intel Quartus Prime* to simulate the design of both units and show that they work. ■

1.2 ALU Verilog Implementation

`Verilog` is a Hardware Description Language (HDL) that is universally being used to model, simulate and synthesize hardware. Verilog has a C-like syntax, and is widely used in the CAD industry. `Verilog` can be used in order to create *structural* as well as *behavioral* models of the circuits. structural modeling focus on the structural composition of components. Behavioral modeling uses high-level language constructs in order to model circuits, in a similar fashion to C and Java. Behavioral models are also used in order to create test benches that automatically generate inputs to and verify outputs from circuit models. Figure 1.10 shows a simple adder using Verilog.

1.3 Library of Parameterized Modules (LPM)

The LPM library is a technology-independent library of logic functions that are parameterized to achieve scalability and adaptability. Quartus has *parameterized modules* or *functions* that offer architecture-independent design entry for all `Intel Quartus Prime` supported devices. The `Quartus Prime` software suite includes built-in compilation support for `LPM` functions used in schematic, AHDL, VHDL, Verilog, and EDIF input files.

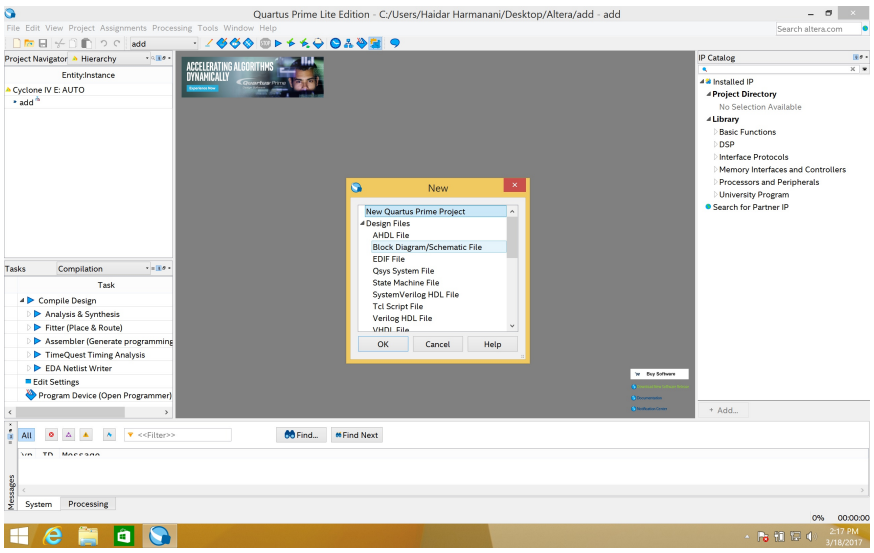


Figure 1.4: Step 2: Create Schematic File

lpm_add_sub

The first lpm module that we will be using is the `lpm_add_sub`, shown in Figure 1.12. The `lpm_add_sub` generates adder, subtractor, and dynamically configurable adder/subtractor functions. The module supports data width of 1–256 bits as well as signed and unsigned data representation. It also supports carry-in, carry-out, synchronous clear, overflow output ports, constants additions and subtraction (only one data bus), clock enable input ports, and pipelining with configurable output latency. The `lpm_add_sub` can be parametrized either graphically, by clicking on the parameters table (Figure 1.13), or directly bu modifying the code. The Verilog prototype for a one bit adder/subtractor using `lpm_add_sub` looks is shown in Figure 1.15.

The `LPM_ADD_SUB IP` core has the following parameters:

1. `LPM_WIDTH`: Specifies the widths of the dataa[], datab[], and result[] ports.
2. `LPM_DIRECTION`: Values are ADD, SUB, and UNUSED. If omitted, the default value is DEFAULT, which directs the parameter to take its value from the `add_sub` port. The `add_sub` port cannot be used if `LPM_DIRECTION` is used. Intel recommends that you use the `LPM_DIRECTION` parameter to specify the operation of the `LPM_ADD_SUB` function, rather than assigning a constant to the `add_sub` port.
3. `LPM_REPRESENTATION`: Values are SIGNED and UNSIGNED. If omitted, the default value is SIGNED.
4. `LPM_PIPELINE`: Specifies the number of latency clock cycles associated with the result[] output. A value of zero (0) indicates that no latency exists, and that a purely combinational function will be instantiated. If omitted, the default value is 0 (non-pipelined).
5. `LPM_HINT`: Allow users to specify Intel-specific parameters in design files.
6. `ONE_INPUT_IS_CONSTANT`: Values are YES, NO, and UNUSED. If omitted, the default value is NO.
7. `MAXIMIZE_SPEED`: You must use the `LPM_HINT` parameter to specify the `MAXIMIZE_SPEED` parameter. Value is between 0 and 10. If used, the Intel Quartus Prime software attempts to optimize a specific instance of the `LPM_ADD_SUB` function for speed rather than routability, and overrides the setting of the Optimization Technique logic option. If `MAXIMIZE_SPEED` is unused, the value of the Optimization Technique option is used instead. If the setting

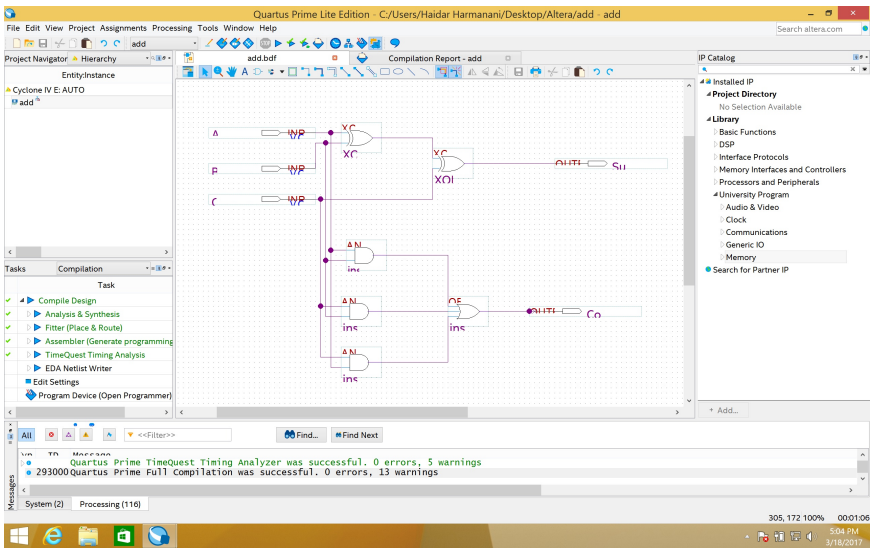


Figure 1.5: Step 2: Create Schematic File

for `MAXIMIZE_SPEED` is 6 or higher, the Compiler optimizes the `LPM_ADD_SUB` IP core for higher speed using carry chains; if the setting is 5 or less, the Compiler implements the design without carry chains. This parameter must be specified for Cyclone, Stratix, and Stratix GX devices only when the `add_sub` port is not used.

8. The `clock` is only used for pipelining as it provides input for a pipelined operation. For `LPM_PIPELINE` values other than 0 (default), the clock port `clken` must be enabled. Finally, `aclr` is an asynchronous clear for pipelined usage.

lpm_mux

The second `lpm` module that we will be using is the `lpm_mux`, shown in Figure 1.14. The `lpm_mux` is a parameterized multiplexer. The module pipelining, and can be parametrized either graphically, by clicking on the parameters table, or directly by modifying the code.

The `LPM_MUX` IP core has the following parameters:

1. `LPM_WIDTH`: Specifies the width of the data[][] and result[] ports.
2. `LPM_SIZE`: Specifies the number of input buses to the multiplexer. The `LPM_SIZE` $\leq 2^{LPM\_WIDTHS}$.
3. `LPM_WIDTHS`: Specifies the width of the sel[] input port.
4. `LPM_PIPELINE`: Specifies the number of latency clock cycles associated with the result[] output. A value of zero (0) indicates that no latency exists, and that a purely combinational function will be instantiated. If omitted, the default value is 0 (non-pipelined).
5. The `clock` is only used for pipelining as it provides input for a pipelined operation. For `LPM_PIPELINE` values other than 0 (default), the clock port `clken` must be enabled. Finally, `aclr` is an asynchronous clear for pipelined usage.

1.4 Connecting Multi-Bit Designs

In digital circuits, adding two four-bit values `A` and `B` requires the use of multiple bits, and are typically labeled in a manner that is similar to arrays in software. Thus, adding two four-bit quantities and storing the result in a four-bit output is illustrated as follows: `Sum[3..0]` $\leftarrow$ `A[3..0] + B[3..0]`. Multiple bits are grouped together and transported through a bus of the

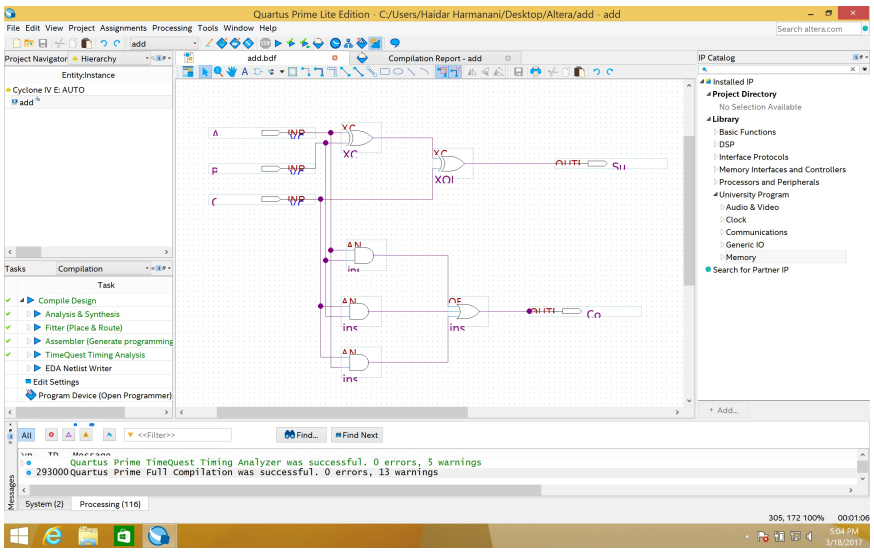


Figure 1.6: Step 3: Create Simulation File: File → New → Verification/Debugging Files → University Program VWF

appropriate size. In Quartus Prime, a bus is selected using the bus tool  . Inputs should be labeled using a vector notation as shown Figure 1.16.

Quartus supports generic design modules. Such designs appear to have one single input which can be parametrized to include multiple inputs. This includes devices such as `lpm_and`, `lpm_or`, and `lpm_mux` to give few examples. The inputs of such modules are two dimensional arrays: one refers to the input line while the other to the number of bits. For example, to access the first input of the AND gate in Figure 1.17, one may use `data[0][3..0]`.

Lab 1.3 LPM Adder/Subtractor Design. Design a 4-bit adder/subtractor using the lpm component `lpm_add_sub`. Simulate the design and demonstrate that it works. ■

Lab 1.4 LPM Mux Design. Design a 4-bit multiplexor using the lpm component `lpm_mux`. Simulate the design and demonstrate that it works. ■

Lab 1.5 ALU Design. Create the complete ALU design by adding the necessary registers. Design a 4-bit register as well as a one bit register using the lpm component `lpm_ff`. Redesign the ALU by adding the appropriate registers. ■

1.5 Problems

Problem 1.1 Derive the truth table of the circuit below. What function does it implement?

Problem 1.2 Each of the four members of a "Jukebox Jury" is provided with a push button; the button is pressed if the record played is a "hit" and is not pressed if the record played is a "miss." Design a digital circuit that lights a lamp when a majority of the jury thinks the record is a hit.

Problem 1.3 Prove that: $(X+Y).(\overline{X}+Z) = X.Z+\overline{X}.Y$

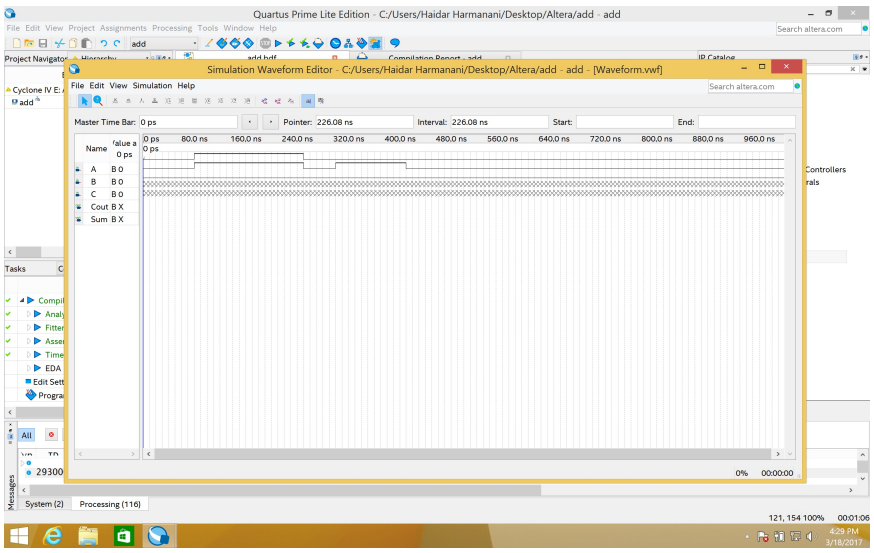


Figure 1.7: Step 3: Create Simulation File: 2) File → New → Verification/Debugging Files → University Program VWF

Problem 1.4 Derive the truth table of the circuit below. What function does it implement?

Problem 1.5 Design a circuit for a car alarm buzzer that goes on if the ignition is on, the gear is engaged, and the driver's seat is occupied but the seat belt is not fastened.

Problem 1.6 Design a logic circuit that adds two digits in modulo-4 arithmetic. There are 4 digits only in modulo-4, namely, 0, 1,2, and 3. For any two digits, x and y, in modulo-4 arithmetic, addition is defined by: $x +_4 y = (x +_{10} y) \bmod 4$

Problem 1.7 Use *Karnaugh* maps in order to find a minimal sum-of-products representation of the following logic functions:

1. $f(A,B,C,D) = \sum m(0,1,2,3,4,5,6,8,10,11,12,14,15)$
2. $f(A,B,C,D) = \sum m(0,2,10,11,12,14)$
3. $f(A,B,C,D) = \sum m(1,5,6,7,11,12,13,15)$

Problem 1.8 In the yard tower of a railroad yard, a controller must select the route of freight cars entering a section of the yard from point A as shown in Figure 1.19. Depending on the position of the switches, a car can arrive at any of four destinations. Other cars may enter from points B or C. Design a circuit that will receive as inputs signals s_1 to s_5, indicating the positions of the corresponding switches, and will light a lamp, D_0 to D_4, showing which destination the car from A will reach. For these cases when cars enter from B or C (S_2 or S_5 in the 0 position), all output lamps should light, indicating that a car from A cannot reach its destination safely.

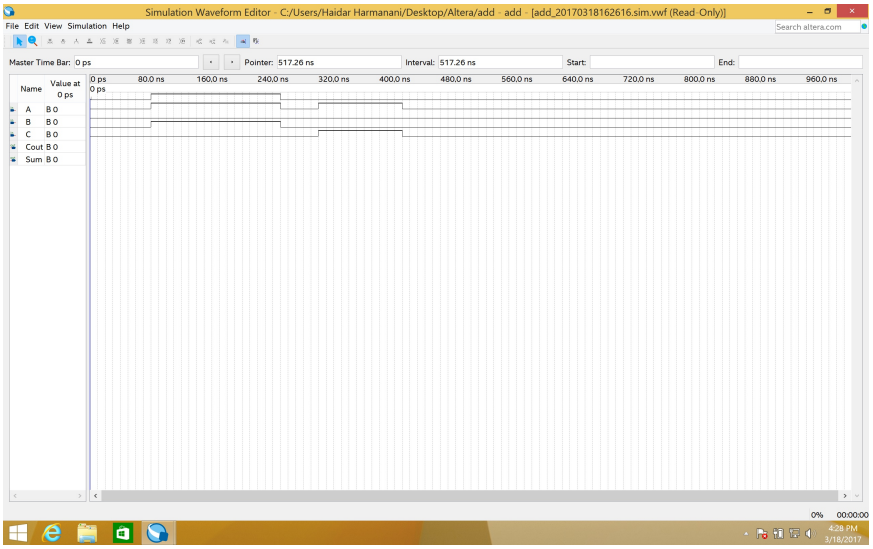


Figure 1.8: Step 3: Simulate and Verify

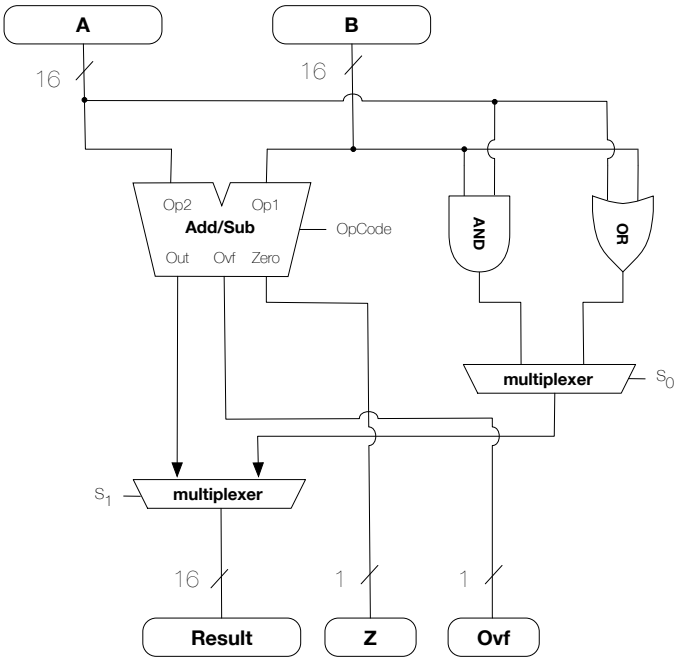


Figure 1.9: ALU Block Diagram

```
module adder(A, B, CI, SUM);
input  [3:0] A;
input  [3:0] B;
input  CI;
output [3:0] SUM;

assign SUM = A + B + CI;
endmodule
```

Figure 1.10: Behavioral Verilog Model for an Unsigned 4-bit Adder with Carry-In

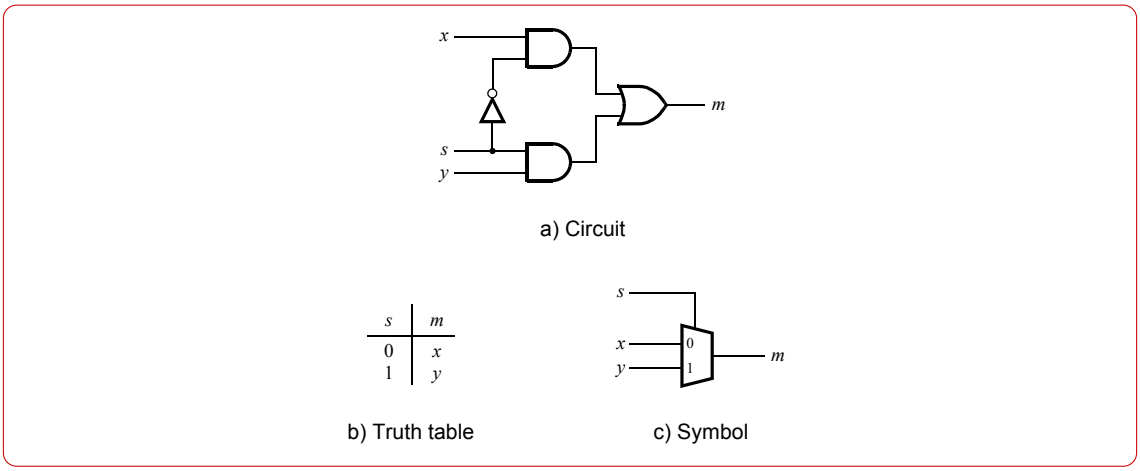


Figure 1.11: A 2-to-1 multiplexer.

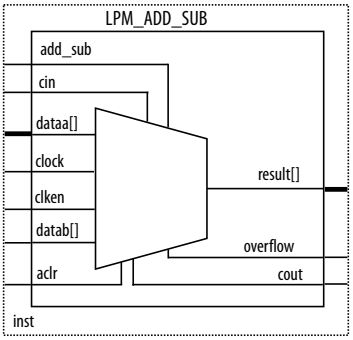


Figure 1.12: `lpm_add_sub` Function

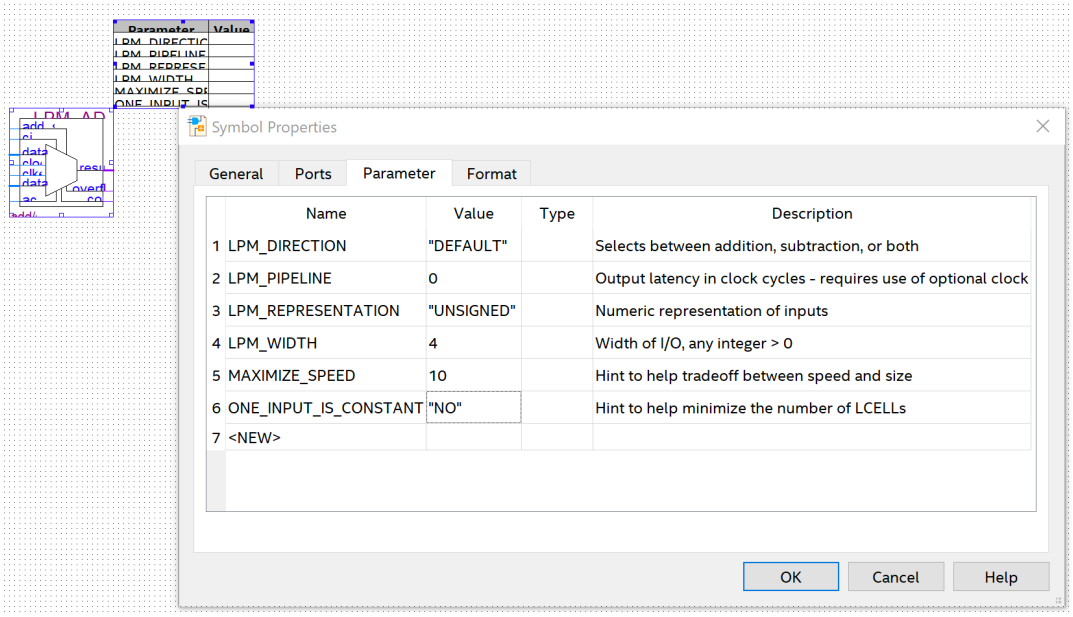


Figure 1.13: Parametrizing the `lpm_add_sub` Function

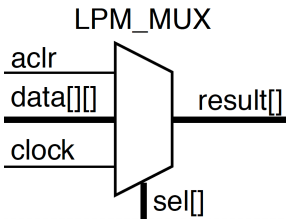


Figure 1.14: lpm_mux Function

```
module lpm_add_sub (result, cout, overflow, add_sub, cin, dataa, datab,
                                                    clock, clken, aclr);
parameter lpm_type = "lpm_add_sub";
parameter lpm_width = 1;
parameter lpm_direction = "UNUSED";
parameter lpm_representation = "SIGNED";
parameter lpm_pipeline = 0;
parameter lpm_hint = "UNUSED";
input [lpm_width-1:0] dataa, datab;
input add_sub, cin;
input clock;
input clken;
input aclr;
output [lpm_width-1:0] result;
output cout, overflow;
endmodule
```

Figure 1.15: `lpm_add_sub` Verilog HDL Prototype

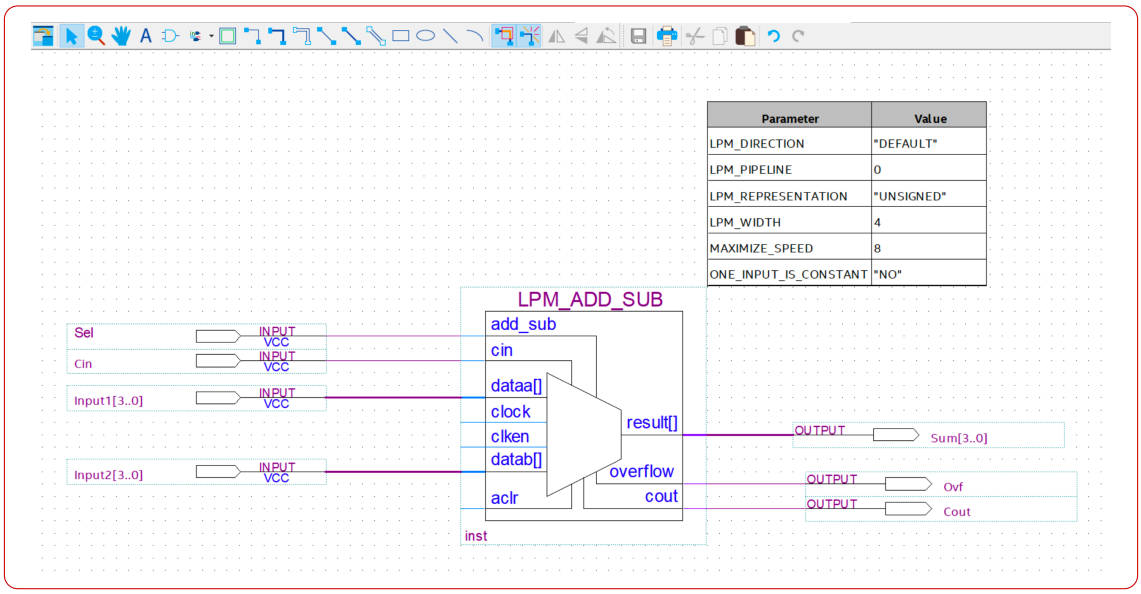


Figure 1.16: Using the `lpm_add_sub` Function

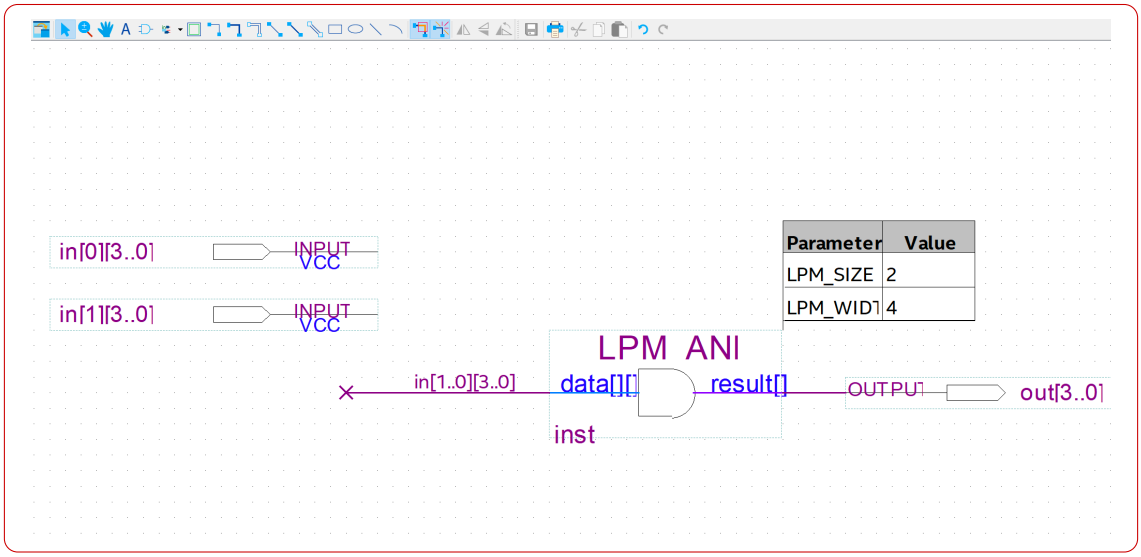
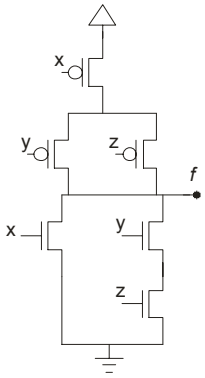
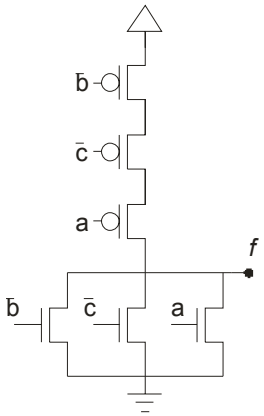


Figure 1.17: Connecting the `lpm_and` Function

1. File menu ⇒ New ⇒ Block Diagram /Schematic File
2. Select your megafunction from the library.
3. Connect your components using wires for single bits, and buses for multiple bits.
4. connect your inputs and outputs to input and output ports.
5. Buses and wires can be either connected directly or by using name association. In other words, wires that have the same names are automatically connected by Quartus, even if they do not appear to be connected on the screen.

Figure 1.18: Create `lp` Functions Using Quartus Prime Schematic Editor





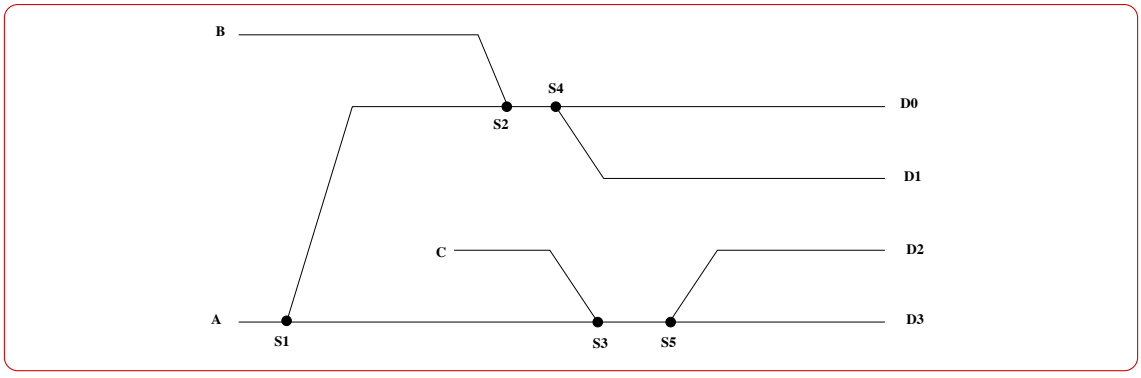


Figure 1.19: Railroad Yard