FPGA Design Basics: LUT-Based Structure and Coding Style

Objectives

The objective of this lab is to introduce the students to the concept of hardware utilization and the effects of coding style in hardware descriptive languages (HDL), such as Verilog or VHDL. By the end of this lab, students should be able to:

- Understand the basic coding skills of Intel FPGA Tools
- Understand the Shannon's expansion theorem and Davio expansion theorem

Preliminaries

In this lab, you will be using both the Intel Quartus® Prime. You should be familiar with the Altera tools from previous classes. Review the procedure on how to create a new project and how to compile a design on Intel Quartus® Prime.

Procedure

LUT-structure (Intel FPGA)

The smallest unit of logic in the Cyclone 4 architecture, the LE, is compact and provides advanced features with efficient logic utilization. Each LE features:

- A four-input look-up table (LUT), which is a function generator that can implement any function of four variables
- A programmable register
- A carry chain connection
- A register chain connection
- The ability to drive all types of interconnects: local, row, column, register chain, and direct link interconnects
- Support for register packing
- Support for register feedback

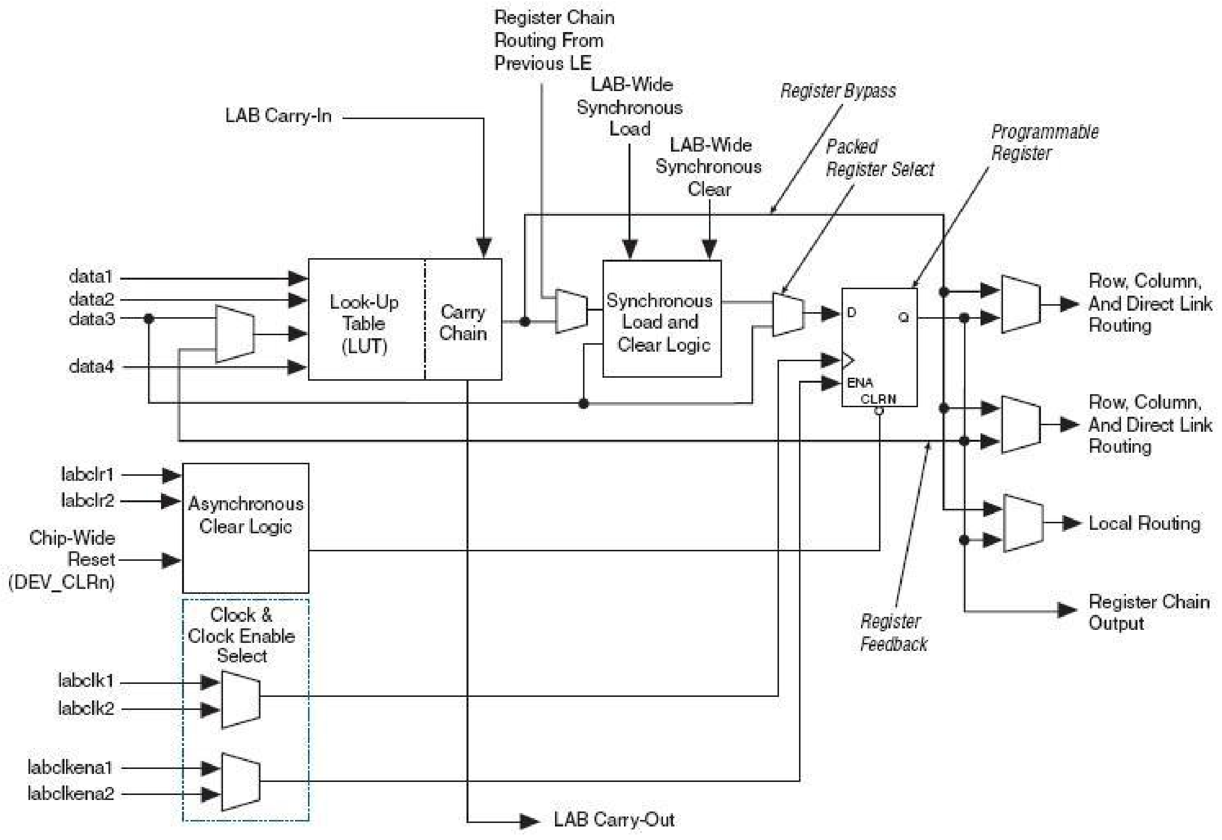


Figure 1: Cyclone II LE

LE Operating Modes (Intel FPGA)

The Cyclone II LE operates in one of the following modes:

- Normal mode
- Arithmetic mode

Each mode uses LE resources differently. In each mode, six available inputs to the LE—the four data inputs from the Logic Array Block (LAB) local interconnect, the LAB carry-in from the previous carry-chain LAB, and the register chain connection—are directed to different destinations to implement the desired logic function. LAB-wide signals provide clock, asynchronous clear, synchronous clear, synchronous load, and clock enable control for the register. These LAB-wide signals are available in all LE modes.

The Quartus® II software, in conjunction with parameterized functions such as library of parameterized modules (LPM) functions, automatically chooses the appropriate mode for common functions such as counters, adders, subtractors, and arithmetic functions. If required, you can also create special-purpose functions that specify which LE operating mode to use for optimal performance.

Normal Mode

The normal mode is suitable for general logic applications and combinational functions. In normal mode, four data inputs from the LAB local interconnect are inputs to a four-input LUT. The Quartus Compiler automatically selects the carry-in or the data3 signal as one of the inputs to the LUT. LEs in normal mode support packed registers and register feedback.

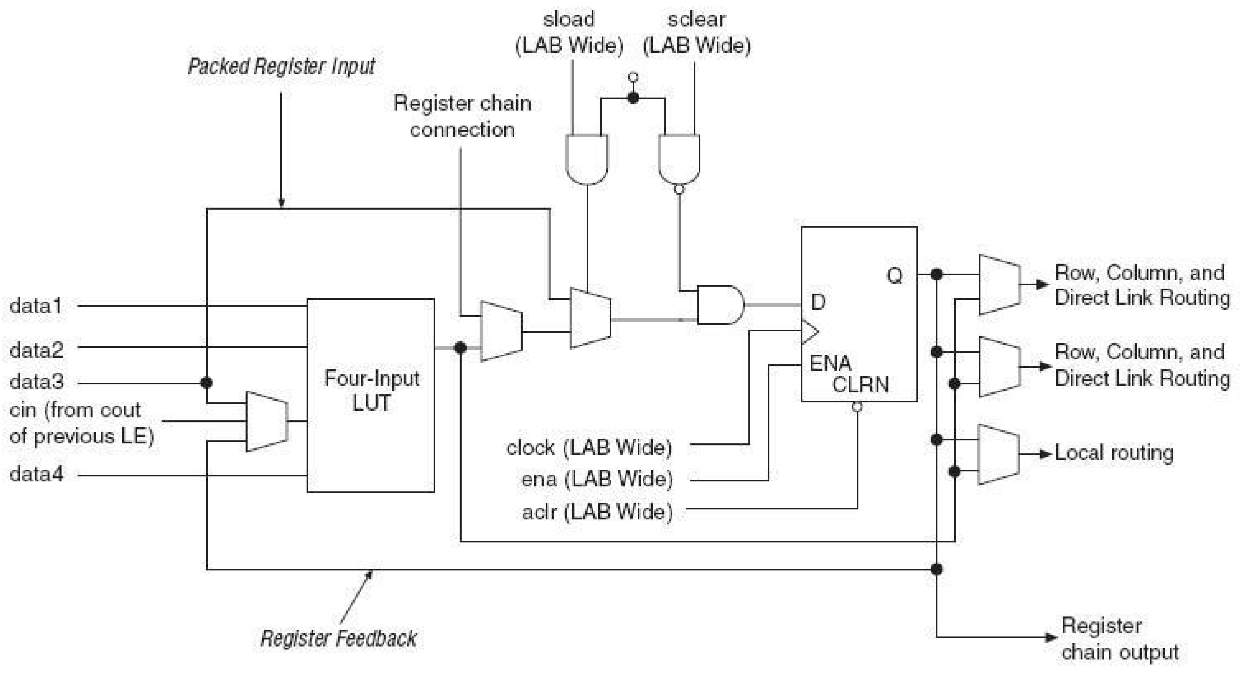


Figure 2: LE in Normal Mode

Arithmetic Mode

The arithmetic mode is ideal for implementing adders, counters, accumulators, and comparators. An LE in arithmetic mode implements a 2-bit full adder and basic carry chain. LEs in arithmetic mode can drive out registered and unregistered versions of the LUT output. Register feedback and register packing are supported when LEs are used in arithmetic mode.

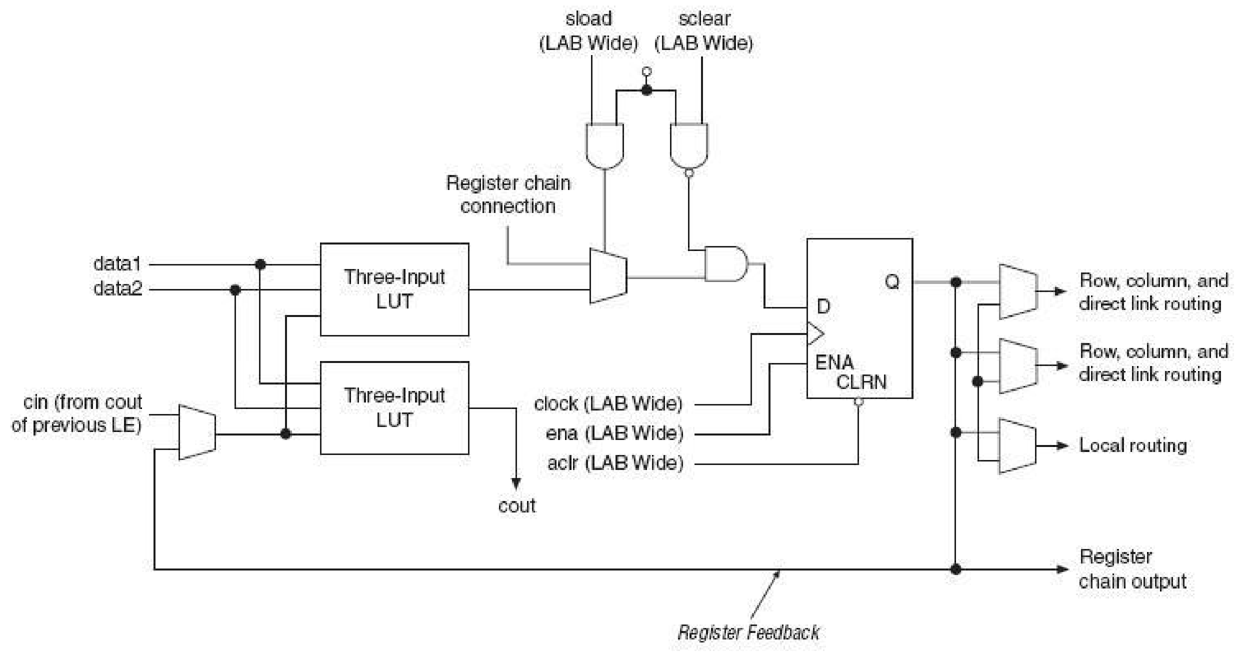


Figure 3: LE in Arithmetic Mode

The Quartus Compiler automatically creates carry chain logic during design processing, or you can create it manually during design entry. Parameterized functions such as LPM functions automatically take advantage of carry chains for the appropriate functions.

The Quartus Compiler creates carry chains longer than 16 LEs by automatically linking LABs in the same column. For enhanced fitting, a long carry chain runs vertically, which allows fast horizontal connections to M4K memory blocks or embedded multipliers through direct link interconnect. For example, if a design has a long carry chain in a LAB column next to a column of M4K memory blocks, any LE output can feed an adjacent M4K memory block through the direct link interconnect. Whereas if the carry chains ran horizontally, any LAB not next to the column of M4K memory blocks would use other row or column interconnects to drive a M4K memory block. A carry chain continues as far as a full column.

Coding Style (Intel FPGA)

Multiplier

To infer multiplier functions, synthesis tools look for multipliers and convert them to LPM_MULT or ALTMULT_ADD IP Core, or may map them directly to device atoms. For devices with DSP blocks, the software can implement the function in a DSP block instead of logic, depending on device utilization. The Quartus® II Fitter can also place input and output registers in DSP blocks (that is, perform register packing) to improve performance and area utilization.

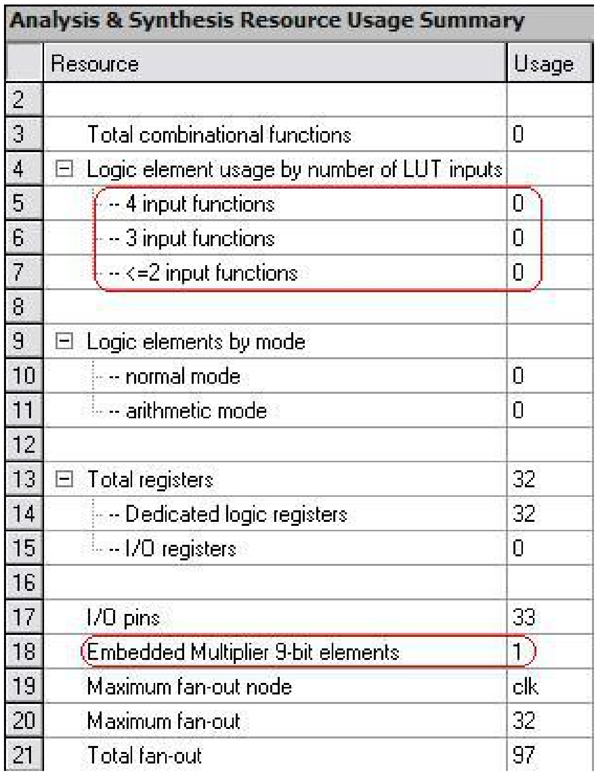
Example 1:

```
module signed_mult (out, clk, a, b);
  output [15:0] out;
  input clk;
  input signed [7:0] a;
  input signed [7:0] b;

  reg signed [7:0] a_reg;
  reg signed [7:0] b_reg;
  reg signed [15:0] out;
  wire signed [15:0] mult_out;

  assign mult_out = a_reg * b_reg;
  always @ (posedge clk)
  begin
    a_reg <= a;
    b_reg <= b;
    out <= mult_out;
  end
endmodule
```

The synthesis result is shown in Figure 4. Note that in the resource usage summary, no LUTs are spent in the registered multiplier.



Analysis & Synthesis Resource Usage Summary

	Resource	Usage
2		
3	Total combinational functions	0
4	⊟ Logic element usage by number of LUT inputs	
5	-- 4 input functions	0
6	-- 3 input functions	0
7	-- <=2 input functions	0
8		
9	⊟ Logic elements by mode	
10	-- normal mode	0
11	-- arithmetic mode	0
12		
13	⊟ Total registers	32
14	-- Dedicated logic registers	32
15	-- I/O registers	0
16		
17	I/O pins	33
18	Embedded Multiplier 9-bit elements	1
19	Maximum fan-out node	clk
20	Maximum fan-out	32
21	Total fan-out	97

Figure 4: Resource usage summary

Read Only Memory

To infer ROM functions, synthesis tools detect sets of registers and logic that can be replaced with the ALTSYNCRAM or LPM_ROM IP Core, depending on the target device family, and only for device families that have dedicated memory blocks.

ROMs are inferred when a `CASE` statement exists in which a value is set to a constant for every choice in the `CASE` statement. Because small ROMs typically achieve the best performance when they are implemented using the registers in regular logic, each ROM function must meet a minimum size requirement to be inferred and placed into memory.

Example 2:

```
module sync_rom (clock, address, data_out);
  input clock;
  input [3:0] address;
  output [5:0] data_out;

  reg [5:0] data_out;

  always @ (posedge clock)
  begin
    case (address)
      4'b0000: data_out <= 6'b101111;
      4'b0001: data_out <= 6'b110110;
      // You should finish the missing code at here.
      // Hint: address is from 4'b0000 to 4'b1111,
      // and the data_out can be arbitrary.
      4'b1110: data_out <= 6'b000001;
      4'b1111: data_out <= 6'b101010;
    endcase
  end
endmodule
```

Multiplexer Types

This section addresses how multiplexers are created from various types of HDL code. `CASE` statements, `IF` statements, and state machines are all common sources of multiplexer logic in designs. These HDL structures create different types of multiplexers, including binary multiplexers, selector multiplexers, and priority multiplexers. Understanding how multiplexers are created from HDL code, and how they might be implemented during synthesis, is the first step toward optimizing multiplexer structures for best results.

Binary Multiplexers

Binary multiplexers select inputs based on binary-encoded selection bits. Example 3/4 shows Verilog HDL code for two ways to describe a simple 4:1 binary multiplexer.

Example 3:

```
module bin_mux (sel, a, b, c, d, z);
  input [1:0] sel;
  input a, b, c, d;
  output z;

  always @ *
  begin
    case (sel)
      2'b00: z <= a;
      2'b01: z <= b;
      2'b10: z <= c;
      2'b11: z <= d;
    endcase
  end
```

Stratix series devices starting with the Stratix II device family feature 6-input look up tables (LUTs) which are perfectly suited for 4:1 multiplexer building blocks (4 data and 2 select inputs). The extended input mode facilitates implementing 8:1 blocks, and the fractured mode handles residual 2:1 multiplexer pairs. For device families using 4-input LUTs, such as the Cyclone series and Stratix devices, the 4:1 binary multiplexer is efficiently implemented by using two 4-input LUTs. Larger binary multiplexers are decomposed by the synthesis tool into 4:1 multiplexer blocks, possibly with a residual 2:1 multiplexer at the head.

Selector Multiplexer

Selector multiplexers have a separate select line for each data input. The select lines for the multiplexer are one-hot encoded. Example 4 shows a simple Verilog HDL code example describing a one-hot selector multiplexer.

Example 4:

```
module sel_mux (sel, a, b, c, d, z);
  input [3:0] sel;
  input a, b, c, d;
  output z;

  always @ *
  begin
    case (sel)
      4'b0001: z <= a;
      4'b0010: z <= b;
      4'b0100: z <= c;
      4'b1000: z <= d;
      Default: z <= 1'bx;
    endcase
  end
```

Selector multiplexers are commonly built as a tree of AND and OR gates. An N-input selector multiplexer of this structure is slightly less efficient in implementation than a binary multiplexer. However, in many cases the select signal is the output of a decoder, in which case Quartus Synthesis will try to combine the selector and decoder into a binary multiplexer.

Priority Multiplexers

In priority multiplexers, the select logic implies a priority. The options to select the correct item must be checked in a specific order based on signal priority. These structures commonly are created from `IF`, `ELSE`, `WHEN`, `SELECT`, and `?:` statements in VHDL or Verilog HDL. The example Verilog code in Example 5 probably results in the schematic implementation illustrated in Figure 5.

Example 5:

```
module priority_mux (cond1, cond2, cond3, a, b, c, d, z);
  input cond1, cond2, cond3;
  input a, b, c, d;
  output z;

  always @ *
  begin
    if(cond1)
      z <= a;
    else if(cond2)
      z <= b;
    else if(cond3)
      z <= c;
    else
      z <= d;
  end
endmodule
```

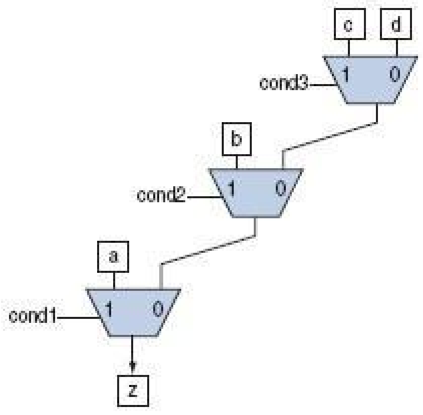


Figure 5: Priority Multiplexer Implementation of an IF Statement

Hint: Select the following device: **Cyclone IV, EP4CE115F29C7**.

Lab Problems (Altera FPGA)

1. (1)Explain the synthesis difference between using case statements and if/else statements.
2. (3)Describe in your own words how your design can be optimized for performance, and how it can be optimized for (minimum) size/area.
3. (2)The VHDL and Verilog code for parity_conventional will be provided along with a template for parity_davio. Based on the conventional implementation, complete the HDL file for the Davio expansion (only required to do one of VHDL or Verilog). Refer to the Davio expansion theorem in your class ppt file.
4. (3)Verify that the Davio expansion code is functionally equivalent to the conventional method. (Use Modelsim/Questasim to prove correctness). Include screenshots that show your verification. *Hint: make sure you actually show/prove that your design works.*
5. (1)Try changing the optimization strategy (Area/Power/Performance/Balanced) and explain your results (Assignments→Settings→Compiler Settings). What does the synthesis tool adjust to get the best 'area' or 'performance' or 'power'? How does the Davio expansion compare to the conventional method in terms of synthesis (explain your result)?

Deliverables

Hand in a short report of the lab problems on Blackboard. Include your Davio expansion code and your testbench.