ESE 382 Digital System Design Using VHDL and PLDs

Spring 20, Ken Short
revised: February 15, 2020 9:19 am

Laboratory 3: Dataflow Style Combinational Design Using Boolean Equations - Gray Code to Binary Code Conversion

This laboratory is to be performed the week starting Feb. 16th.

Prerequisite Reading

1. Chapter 3 and Sections 4.1 and 4.2 of the text.
2. ispGAL22V10 Data Sheet.

Purpose

One purpose of this laboratory is to reinforce your understanding of the VHDL/PLD design flow steps introduced in Laboratories 1 and 2. Another purpose is to have you write your own first dataflow style architectures for design entities. This laboratory will also demonstrate the synthesizer's ability to simplify Boolean equations.

You will create your own VHDL design descriptions (using dataflow architectures) for a combinational circuit decoder. You will perform functional simulations of your design descriptions and timing simulations of the VHDL timing models produced by the place-and-route tool. Finally, you will program each design into an ispGAL22V10C-10LJ SPLD and verify its operation in a test circuit.

Any combinational function can be represented by a truth table. Once a truth table has been created for a combinational function, a Boolean expression can be written for each output. These expressions can be written in a canonical sum-of-products (CSOP) form, as the logical sum of the function's minterms. This is a straight forward, though not concise, way of expressing any combinational function. Alternatively, the Boolean expression for each output can be written in canonical product-of-sums (CPOS) form, as the logical product of the function's maxterms.

Hopefully, you remember minterms and maxterms from your introductory digital design course. If not, you should review those concepts.

Since the synthesizer will minimize either CSOP or CPOS Boolean functions, the synthesized logic corresponds to either the simplified sum-of-products or simplified product-of-sums form for the functions.

VHDL provides logical operators **and**, **or**, **nand**, **nor**, **xor**, **xnor**, and **not**. These operators can be applied to boolean, bit, and, if the package std_logic_1164 is used, std_logic data types. The examples of dataflow architectures that we have studied so far all used a form of concurrent signal assignment statement that is analogous to a Boolean equation. This is the form of concurrent signal assignment statement you must use for this laboratory's designs. Recall that in VHDL, the

operator **not** has the highest precedence. All the other operators have the same precedence. Therefore, parentheses must be used to enforce any other desired precedence.

Using a truth table and writing either a CSOP or CPOS expression for each output is, for some combinational functions, an inefficient approach. However, the advantages of this approach are that it works for any combinational function and is straightforward, even though possibly tedious. In contrast, if there is a simple logical or mathematical relationship between an output and the inputs of a combinational function, then that relationship likely provides a more efficient way to write the design description in VHDL than the approach being used in this particular laboratory.

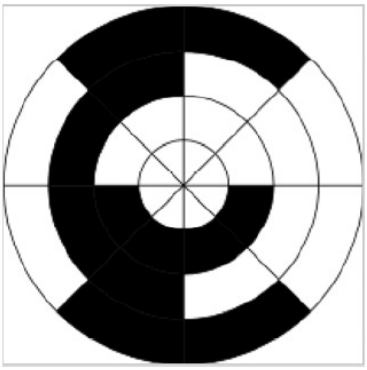
Gray Code and Absolute Encoders

Gray code is a binary code named after its designer Frank Gray. In his writings about this code, he referred to it as reflected binary code. So, both names are used for this code.

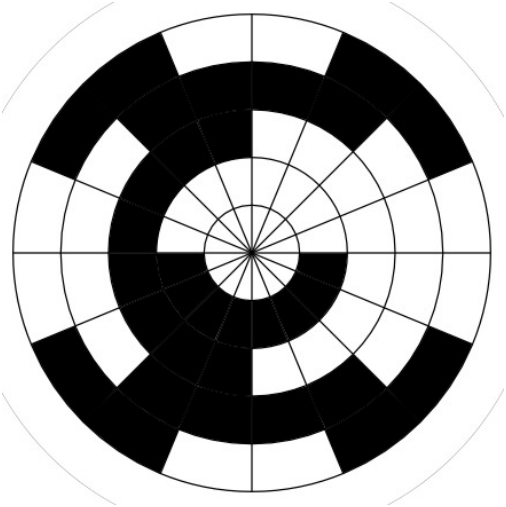
The objective in creating the Gray code was to have a binary numerical code that changed in a single bit when going from the code for one numeric value to the code for an adjacent numeric value. Use of such a code in an electromechanical rotary encoder results in encoder outputs that are immune from spurious errors when reading the code wheel. Gray code is used in many more areas today, including digital communications.

Rotary encoders with Gray outputs are also called absolute encoders, because their outputs tell the absolute position of the encoder's shaft. The rotational position is known by simply reading the code. This eliminates the need to return to the origin at startup as is required with serial rotary encoders.

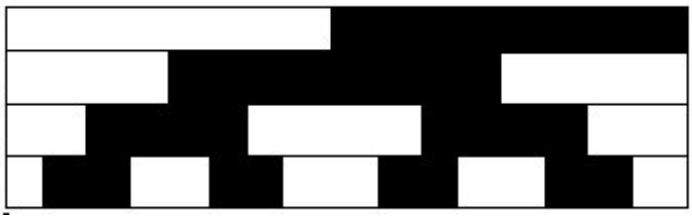
An encoding wheel for a 3-bit rotary encoder is shown in the following figure. The outer three rings provide the 3-bit binary code. The outer ring is the least-significant bit. Assume that the white areas correspond to 0s and the black areas to 1s. Also assume there are sensors just above the positive *x* axis on each of the outer three rings. The code corresponding to the angular position of the wheel, positioned as shown in the figure, would be 000. If the wheel is turned 45 degrees clockwise, the code would be 001. If the wheel is turned 45 degrees counter-clockwise, the code would be 100. Only one bit changed between these adjacent codes. The complete sequence of codes for a 360 degree clockwise rotation would be: 000, 001, 011, 010, 110, 111, 101, 100, 000. Notice that only one bit changes between any two adjacent codes.



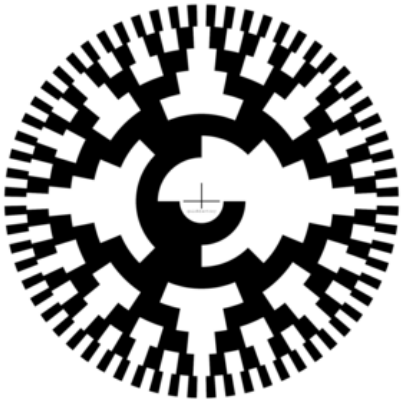
From the 3-bit code we can determine the angular position of a shaft connected to the code wheel to within 45 degrees. With an *n*-bit Gray code we could determine angular position to within 360/ 2**n degrees. For example, a 9 bit code would give us a resolution of 0.7 degrees.
A 4-bit version of the Gray code encoding wheel would look like this:



The same techniques can be used for encoding linear position. The pattern for a 4-bit linear Gray Code encoder is:



The following pattern is for an 8-bit rotary encoder.



A commercial implementation of an 8-bit absolute encoder with Gray code output (Omron E6CP-A) is shown in the following photo.

If you have an encoder with a Gray code output and you want to arithmetically process the output value, it is usually preferable to convert the Gray code output to natural binary before processing. This can be done by a Gray-to-binary encoder, which is a combinational circuit.

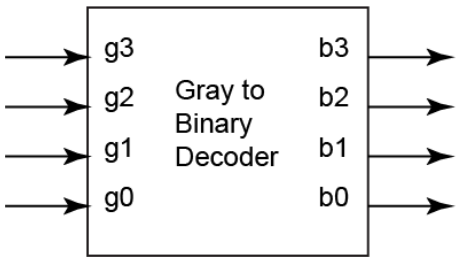
The following table shows a 4-bit Gray code and its corresponding binary code.

Table 1: Gray to Binary

g3	g2	g1	g0	b3	b2	b1	b0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	1	0	0	1	0
0	0	1	0	0	0	1	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
0	1	0	1	0	1	1	0
0	1	0	0	0	1	1	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	1	1	0	1	0
1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	0
1	0	0	0	1	1	1	1

Design Tasks

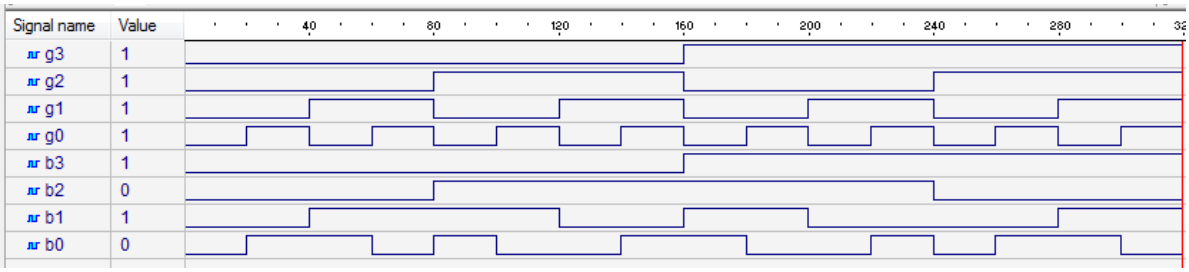
A system is needed that converts a 4-bit Gray code to a 4-bit binary code. The scalar inputs that represent the Gray code are `g3`, `g2`, `g1`, `g0`, where `g3` is the most-significant-bit (msb). The scalar outputs that represent the binary code are `b3`, `b2`, `b1`, `b0`, where `b3` is the msb.



Design Task 1: Canonical Sum-of-Products Approach
From the truth table, write (on paper) the canonical sum-of-products (CSOP) expression for each output of the 4-bit Gray to binary code converter. Do not try to simplify your CSOP Boolean expressions by hand or try to write simplified SOP expressions directly from the truth table. Leave that task to the synthesizer, it will automatically simplify your CSOP expressions for you. And, unlike us, it doesn't make mistakes nor get tired nor frustrated while doing the simplifications.

Using Aldec Active-HDL, create a new workspace named `gray_to_binary`. In this workspace create a design named `gray_bin_csop`.

Using these CSOP expressions, write a dataflow style design description for the system in the form of concurrent signal assignment statements using Boolean expressions. Name the entity `gray_bin` and name its architecture `csop`. Use the testbench `gray_bin_tb` provided to verify your design. Compile and functionally simulate your design. Since the testbench is a non-self-checking testbench, you must use the waveform editor to verify your design's outputs. This is somewhat complicated by the fact that testbench generates the stimulus values applied to the UUT's Gray code inputs using a binary counting sequence and the truth table Gray code inputs are written in the Gray code counting sequence. This means that when you verify your waveforms, the Gray code inputs are applied in a different order than their appearance in the truth table when the truth table is read from top to bottom. This demonstrates why having a self-checking testbench is advantageous. Imagine if you were verifying a 10-bit Gray-to-binary encoder with 1024 rows in its truth table.



Design Task 2: Canonical Product-of-Sums Approach
From the previous truth table write (on paper) the canonical product-of-sums (CPOS) expression for each output of the design entity `gray_bin`.

Create a new design named `gray_bin_cpos` in the existing workspace.

Using the CPOS expressions write a dataflow style design description for the Gray code to binary code converter. Keep the entity name `gray_bin`, but name the architecture `cpos`. Using the testbench from Task 1, compile and functionally simulate this design.

Your design description source file listings for Tasks 1 and 2 are required as part of your prelab that is due when you enter the laboratory.

Laboratory Tasks

Laboratory Task 1: Canonical Sum-of-Products Approach
Using Aldec Active-HDL, create a new workspace named `gray_to_binary`. In this workspace create a design named `gray_bin_csop`. Import your design description source file for Design Task 1 and the provided testbench.

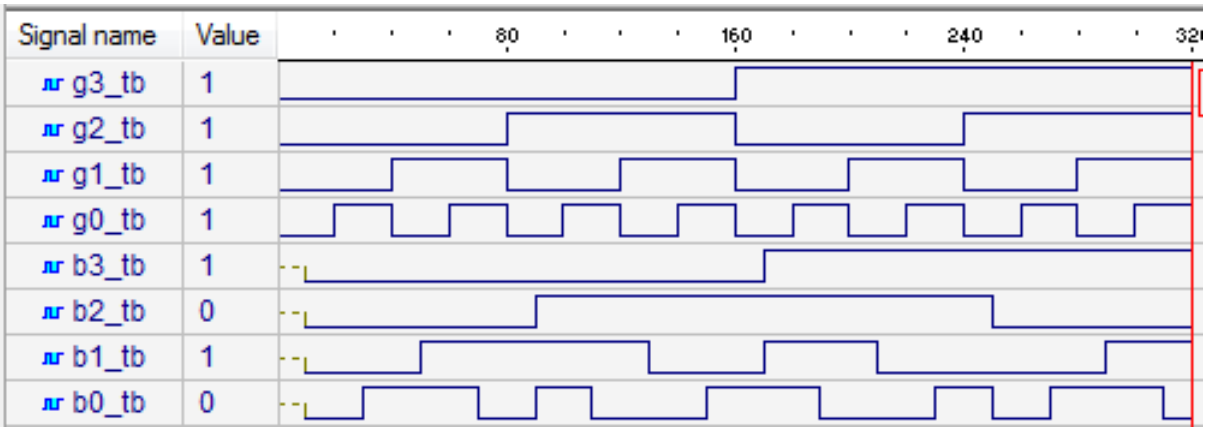
Using the pin assignments given to you in the laboratory, add the appropriate pin assignment attribute statements to your design description. Compile your design description and the testbench. Functionally simulate your design using the testbench. If your outputs are not correct, modify your design description until they are.

Have a TA verify and sign whether the waveforms from your functional simulation meet the design specification.

Using Synplify, synthesize the logic for a Lattice ispGAL22V10C-10LJ target. After synthesis is complete, from the **HDL Analyst** menu select **RTL,** then **Hierarchical View.** Print this diagram of the synthesized logic. Next from the **HDL Analyst** menu select **Technology,** then **Flattened to Gates View.** Print this diagram of the synthesized logic.

Using ispLEVER, fit the synthesized logic to a Lattice ispGAL22V10C-10LJ. In ispLEVER, print the pre-fit equations, post-fit equations, and the chip report.

Perform a timing simulation of the VHDL timing model generated by ispLEVER.



Have a TA verify and sign whether the waveforms from your timing simulation are correct.

Program an ispGAL22V10C-10LJ. Place your programmed ispGAL22V10C-10LJ in the test circuit and verify its functional performance against what was predicted by your simulations.

Have a TA verify and sign whether your programmed PLD meets the design specification.

Laboratory Task 2: Canonical Product-of-Sums Approach
Create a new design named `gray_bin_cpos` in the existing workspace. Import your design description source file for Design Task 2 and the provided testbench to this design. Repeat the process in Laboratory Task 1 for this design.

Questions
1. Create a Karnaugh map for output `b1`. Use this Karnaugh map to determine the **simplified** SOP expressions for `b1`. Repeat this process to find the **simplified** POS expression for `b1`.

2. Compare the pre-fit equations and post-fit equations from ispLEVER with your simplified SOP and POS expressions obtained from your Karnaugh maps. Are they equivalent? If not, how do they differ?

notes
a. In the equations in ispLEVER, the # symbol represents OR, the & symbol represents AND, and the ! symbol represents NOT.
b. The pre-fit and post-fit equations listings give equations for both the normal and complement of each function. This is done because it is sometimes easier (requires fewer product terms) to implement the complement of a function and then configure the PLD to complement the output of the OR gate creating the function than it is to implement the normal form of the function.

3. From the chip report determine exactly which equations were actually implemented in the PLD for each design.

4. Compare the "Technology Flattened to gates view" schematics for the two designs. Are the two schematics identical? If not, are they logically the same?

5. From the HDL Analyst "Technology Flattened to gates view" schematic for your CSOP design write the Boolean equations for outputs `b3` and `b2` and `b1` as implemented by the synthesizer.