

Numbering Systems

Introduction

Verilog HDL modeling language allows numbers being represented in several radix systems. The underlying circuit processes the number in binary, however, input into and output from such circuits is typically done using decimal numbers. In this lab you will learn various representations and methods for converting numbers from one representation into another. *Please refer to the PlanAhead tutorial on how to use the PlanAhead tool for creating projects and verifying digital circuits.*

Objectives

After completing this lab, you will be able to:

- Define a number in various radix
- Design combinatorial circuits capable of converting data represented in one radix into another
- Design combinatorial circuits to perform simple addition operation
- Learn a technique to improve addition speed

Number Representations

Part 1

In Verilog HDL a signal can have the following four basic values:

- 0 : logic-0 or false
- 1 : logic-1 or true
- x : unknown
- z : high-impedance (tri-state)

A z value at the input of a gate or in expression is usually interpreted as an x. In general, Verilog HDL is case-sensitive, but when the values are represented they are case-insensitive.

There are three types of constants in Verilog HDL: (i) integer, (ii) real, and (iii) string. An underscore character (`_`) can be used in an integer or real constant to improve readability. It cannot appear as the first or last character.

Integer numbers can be written in (i) simple decimal or (ii) base format. An integer in simple decimal form is specified as a sequence of digits with an optional + or a -. For example,

```
15
-32
```

where 15 can be represented in binary as 01111 in 5-bit format and -32 can be represented in binary as 100000 in 6-bit format. The simple format representation may result in 32-bit hardware.

A number can be represented in the base format with syntax:

```
[size] 'base value
```

where the *size* specifies the number of bits, *base* is one of o or O (for octal), b or B (for binary), d or D (for decimal), h or H (for hexadecimal), and *value* is a sequence of digits which are valid for the specified base. The *value* must be unsigned. For example,

```
wire [4:0] 5'O37           5-bit octal representation
reg [3:0] 4'B1x_01        4-bit binary
wire [3:0] 4'd-4          illegal, value cannot be negative
wire [11:0] 7'Hx          7-bit x extended to xxxxxxx
```

If the size specified is larger than the value for the specified constant, the number is padded to the left with 0's except for the case where the left most bit is x or z then the padding is done with x or z. If the size specified is smaller than the extra leftmost bits are ignored. If the size is not specified then it will use 32-bit data.

1-1. Define a 4-bit number in the model and display it on the right most 7-segment display.

- 1-1-1. Open PlanAhead and create a blank project called lab2_1_1.
- 1-1-2. Create and add the Verilog module that defines a 4-bit number in binary format and displays the same on the right most 7-segment display. You may use the model developed in 4-2 of Lab 1 assuming that the number you define will be between 0 and 9.
- 1-1-3. Create and add the UCF file to the project.
- 1-1-4. Synthesize and implement the design.
- 1-1-5. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.

Binary Codes

Part 2

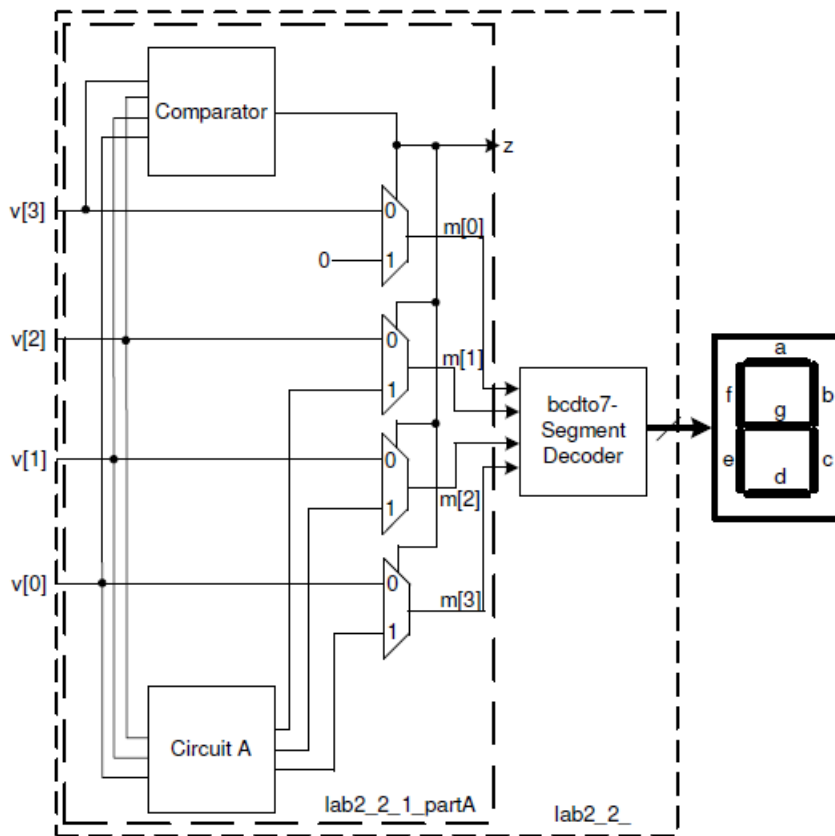
Although most processors compute data in binary form, the input-output is generally done in some coded form. Normally, we exchange information in decimal. Hence decimal numbers must be coded in terms of binary representations. In the simplest form of binary code, each decimal digit is replaced by its binary equivalent. This representation is called Binary Coded Decimal (BCD) or 8-4-2-1 (to indicate weight of each bit position). Because there are only ten decimal digits, 1010 through 1111 are not valid BCD. Table below shows some of the widely used binary codes for decimal digits. These codes are designed and used for communication reliability and error detection.

Decimal Digits	BCD (8-4-2-1)	6-3-1-1	Excess-3	2-out-of-5	Gray code
0	0000	0000	0011	00011	0000
1	0001	0001	0100	00101	0001
2	0010	0011	0101	00110	0011
3	0011	0100	0110	01001	0010
4	0100	0101	0111	01010	0110
5	0101	0111	1000	01100	1110
6	0110	1000	1001	10001	1010
7	0111	1001	1010	10010	1011
8	1000	1011	1011	10100	1001
9	1001	1100	1100	11000	1000

2-1. Output 4-bit binary input onto one LED (most significant bit) and the right most 7-segment display (least significant digit) after converting them into two-digit decimal equivalent (BCD). Use only dataflow modeling.

Design a circuit that converts a 4-bit binary number v into its 2-digit decimal equivalent, z and m . Since valid input range is between 0 and 15, the most significant digit can be displayed using one LED. Table below shows the required output values. A block diagram of the design is given in figure below. 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. Hint: The $m3$ would be zero whenever V is greater than binary 1001.

v[3:0]	z	m[3:0]
0000	0	0000
0001	0	0001
0010	0	0010
0011	0	0011
0100	0	0100
0101	0	0101
0110	0	0110
0111	0	0111
1000	0	1000
1001	0	1001
1010	1	0000
1011	1	0001
1100	1	0010
1101	1	0011
1110	1	0100
1111	1	0101



2-1-1. Open PlanAhead and create a blank project lab2_2_1.

2-1-2. Create and add the Verilog module (name it as lab2_2_1_partA) with v[3:0] input, and z and m[3:0] output, instantiating comparator_dataflow, lab2_circuitA_dataflow, and mux_2to1 and connecting them as necessary.

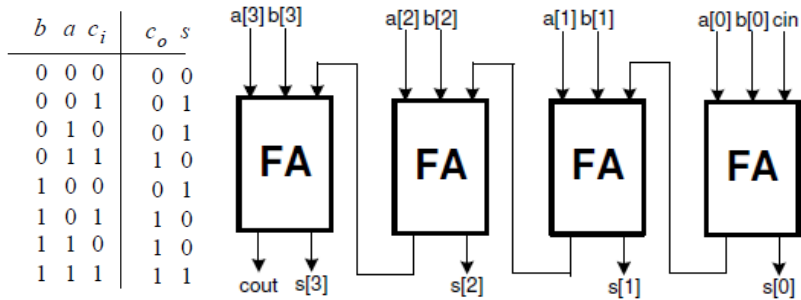
- 2-1-3. Model comparator_dataflow and lab2_circuitA_dataflow functionality using the dataflow constructs.
 - 2-1-4. Add the provided testbench (lab2_2_1_partA_tb.v) to the project.
 - 2-1-5. Simulate the design for 200 ns and verify the design works.
 - 2-1-6. Extend the design by creating the top-level module (lab2_2_1) to have bcdto7segment_dataflow decoder (that you developed in Lab 1) and provide one 7-bit output *seg0* instead of *m*.
 - 2-1-7. Create and add the UCF file to the project. Assign *v* input to **SW3-SW0**, *z* to **LED0**, and *seg0* to **SEG0** (the right most 7-segment display).
 - 2-1-8. Synthesize the design.
 - 2-1-9. Implement the design.
 - 2-1-10. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.
- 2-2. Model a 2-out-of-5 binary code and display a 4-bit binary coded decimal input number onto five LEDs. Use dataflow modeling.**

- 2-2-1. Open PlanAhead and create a blank project lab2_2_2.
- 2-2-2. Create and add a hierarchical design with 4-bit input (*x[3:0]*) and 5-bit output (*y[4:0]*). Use dataflow modeling statements only.
- 2-2-3. Create and add the UCF file to the project. Assign **SW3 -SW0** to *x* and **LED4 to LED0** to *y*.
- 2-2-4. Synthesize and implement the design.
- 2-2-5. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.

Performing Addition

Part 3

When two one-bit numbers are added, they may produce two bits output. For example, $1 + 1 = 10$ (all in binary). When you add three one-bit numbers the result will still be two bits, e.g. $1 + 1 + 1 = 11$. This simple operation can be viewed as adding two bits with carry in from the lower significant operation, resulting into sum and carry out- the left bit is carry out and the right bit is sum. Figure below shows a 4-bit adder. Since the carry is rippled from least significant bit position (cin) to the most significant position (cout), such adder is called ripple carry adder.



3-1. Create a 4-bit ripple carry adder using dataflow modeling.

3-1-1. Open PlanAhead and create a blank project lab2_3_1.

3-1-2. Create and add the Verilog module named fulladder_dataflow with three inputs (a , b , cin) and two outputs (s and $cout$) using dataflow modeling. All inputs and outputs should be one-bit wide.

3-1-3. Add the provided testbench (fulladder_dataflow_tb.v) to the project.

3-1-4. Simulate the design for 80 ns and verify that the design works

3-1-5. Create and add the Verilog module to the project with three inputs (a , b , cin) and two outputs ($cout$ and s) instantiating full adder (FA) four times and connecting them as necessary. The a , b , and s should be a 4-bit vector and cin and $cout$ should each be one-bit wide.

3-1-6. Create and add the UCF file, assigning a to **SW7-SW4**, b to **SW3-SW0**, s to **LED3-LED0**, cin to **BTNU**, and $cout$ to **LED7**.

3-1-7. Synthesize and implement the design.

3-1-8. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.

3-2. Modify the design of 3-1 by treating the 4-bit input as BCD, performing addition, generating result in BCD, and displaying it on LED0 and right most 7-segment display. Use switches to input two 4-bit BCD input and BTNU for carry-in. Reuse models developed in 2-1 and 3-1 as needed. Use dataflow modeling.

3-2-1. Open PlanAhead and create a blank project lab2_3_2.

3-2-2. Modify the project of 3-1 as necessary to perform the required function and outputting the result on LED0 and the right most 7-segment display.

3-2-3. Modify the UCF file

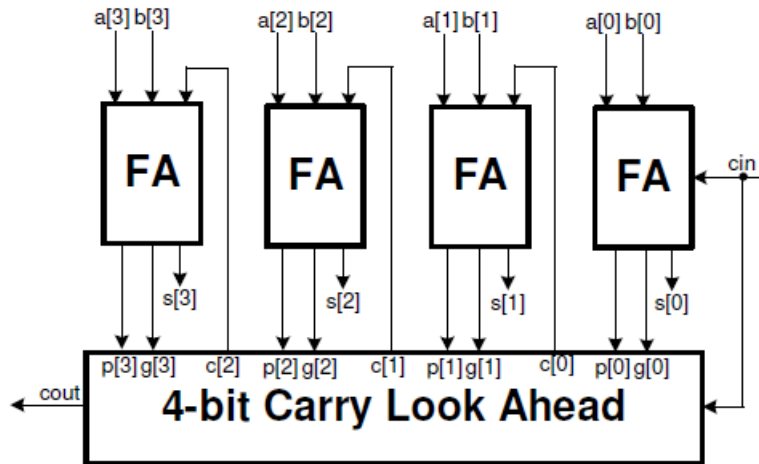
3-2-4. Synthesize and implement the design.

3-2-5. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.

Improving Addition Speed

Part 4

The ripple-carry adders take a longer time to compute when two large numbers (e.g. 8, 16, 32 bits) are added. To reduce the computation time, another structure, called carry-lookahead adders (see figure below), can be used. It works by creating two signals (P and G) for each bit position, based on whether a carry is propagated through a less significant bit position (at least one input is a '1'), generated in that bit position (both inputs are '1'), or killed in that bit position (both inputs are '0'). After P and G are generated the carries for every bit position are created.



Where $P_i = A_i + B_i$ and $G_i = A_i B_i$. Within Carry Look Ahead unit, $C_{i+1} = G_i + P_i C_i$. The speed-up is achieved through the fact that C_i being generated at the same time irrespective of i^{th} position.

4-1. Create a carry-look ahead adder circuit by modifying the project of 3-1 and using dataflow modeling.

4-1-1. Open PlanAhead and open the project you had created in 3-1.

4-1-2. Modify the project of 3-1 as necessary to perform the addition of two four-bit numbers using the carry look-ahead structure and outputting the result LEDs. Provide carry-in through BTNU. Hint: You will need to modify FA to output P_i and G_i , and then create and add another module CLA to perform the carry look ahead function (takes c_0 and p, g_i ($i=0$ to 3) and outputs c_4 and p, g).

4-1-3. Modify the UCF file to provide input b through **SW3-SW0**, a through **SW7-SW4**, cin through **BTNU**. Output $cout$ through **LED7** and sum through **LED3-LED0**.

4-1-4. Synthesize and implement the design.

4-1-5. Generate the bitstream, download it into the Nexys3 board, and verify the functionality.

Conclusion

In this lab, you learned how to define numbers in various radix systems. You also designed various number conversion circuits using dataflow modeling. You also learned a technique of improving addition speed.