

Lab Assignment 4

Complemented Borrow Look-ahead Subtractor and Hazard-Free Design

Due August 29, 2013, 4:00 pm

Introduction:

This lab assignment consists of two parts: (1) Implementation of a *borrow look-ahead subtractor* in Verilog, and (2) Hazard analysis and implementation of hazard-free circuits in Logicworks.

Part 1: Borrow look-ahead subtractor

In this part of the lab, you will be designing a borrow look-ahead subtractor. To start, we will analyze the concept of carry look-ahead. We will then graduate to borrow look-ahead, and then we will finally discuss how to convert the general borrow look-ahead design into one that utilizes a complemented borrow.

The interesting part of a binary carry look-ahead adder is that it can in some sense precompute the carries of each bit, thus drastically speeding up the overall adder through enabling a number of operations to happen concurrently. The challenges for you will be concentrated in two parts. First, you need to think about how to design a *borrow look-ahead* logic for binary numbers based on the ideas you have learned from carry look-ahead adders. And second, you should think about how the implementation of a binary subtractor based on rippling a complemented borrow can be applied to borrow look-ahead subtractors.

Carry look-ahead binary adder

To design the correct borrow look-ahead functionality, it is instructive to first review a little bit the concepts of a carry look-ahead adder. You have learned from your textbook that a single bit **full adder** (FA) takes as inputs two operands x_i , y_i and the carry c_i , and produces as outputs the sum s_i and carry c_{i+1} . An n -bit binary adder can be implemented as a serial connection of FAs, arranged such that the output carry of each FA serves as the input carry for the next-higher significant FA. This type of adder is denoted as **ripple-carry adder**, since any change in the least significant bits c_0 , x_0 , or y_0 has to ripple through all the FAs.

One important feature of an adder is its delay, that is, the time it takes for it to output the sum and carry. For a ripple-carry adder, each FA needs to wait for the next-lower significant FA for generating the carry, thus imposing a significant limitation on the speed of this type of adder. The purpose of designing a **carry look-ahead adder** is to reduce the delay of the carry chain in a ripple-carry adder. This technique takes advantage of the fact that the major part of the expressions of each carry in the chain can be precomputed. Let's take as an example a 5-bit adder that computes $x_4x_3x_2x_1x_0 + y_4y_3y_2y_1y_0 + c_0$. We first define two functions as follows: (1) the **carry-generate** function $g_i = x_i y_i$, and (2) the **carry-propagate** function $p_i = x_i \oplus y_i$. Using these two functions, the three carries c_1 , ..., c_3 can be expressed as follows:

- $c_1 = g_0 + p_0 c_0$ (1)
- $c_2 = g_1 + p_1 c_1 = g_1 + p_1(g_0 + p_0 c_0) = g_1 + p_1 g_0 + p_1 p_0 c_0$ (2)
- $c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$ (3)

The equation c_4 and c_5 can be calculated in an identical manner. These equations show that the five carries c_1 , ..., c_5 can be computed directly from input bits $x_4x_3x_2x_1x_0$, $y_4y_3y_2y_1y_0$, and

input carry c_0 without the ripple effect! In other words, the carry value for each bit can be generated in parallel, which in turn allows the addition of each bit to be performed in parallel, thus drastically speeding up the completion delay of the overall adder!

In general, a carry look-ahead adder circuit consists of three parts:

- **Carry-generate functions and carry-propagate functions for every bit:** we know that, if the carry-generate function $g_i = x_i y_i$ is 1, then we will definitely have a carry-out without even having to consult the carry-in bit. On the other hand, the carry-propagate function $p_i = x_i \oplus y_i$ lets us know whether it is necessary to consult the carry-in bit, as when it is 1, the carry-in coming from the lower bit will be directly propagated to the carry-out.
- **A carry look-ahead (CLA) generator:** the CLA logic takes as inputs the carry-generate and carry-propagate functions of each bit, and generates the carry value for each bit in parallel using Equations (1)-(4).
- **1-bit half adders for every bit:** each adder takes as inputs the carry-propagate function p_i and the carry value c_i (generated by the CLA logic), and generates the sum $s_i = p_i \oplus c_i$.

Borrow look-ahead binary subtractor

A borrow look-ahead subtractor shares similar design principles with a carry look-ahead adder. In order to precompute the borrow information, for every bit, a **borrow-generate** function g_i and a **borrow-propagate** function p_i are needed. The borrow-generate function g_i takes as its inputs the i th bit of the minuend x , denoted by x_i , and the i th bit of the subtrahend y , denoted by y_i . When this function is 1, we definitely need to borrow from the higher bit. The borrow-propagate function p_i also takes x_i and y_i as its inputs, and lets us know whether we should consult the borrow-in bit from the lower bit. When this function is 1, the borrow-in bit will be directly propagated to the borrow-out. The borrow-generate and borrow-propagate functions of each bit are fed into a borrow look-ahead logic (BLA) which generates the borrow value for each bit of the subtraction in parallel. Finally, for each bit, a 1-bit subtractor is employed to receive its borrow value as one of its three inputs, and output the final result of the subtraction.

The borrow-generate function is defined as $g_i = x_i' y_i$. This is because, when x is 0 and y is 1, we will always borrow, regardless of the borrow-in. The borrow-propagate function is defined as $p_i = x_i' \oplus y_i = x_i \text{ xnor } y_i$, because we will otherwise propagate the borrow-in if x and y are both 0, or they are both 1. The borrow functions are defined in exactly the same way as the carry functions:

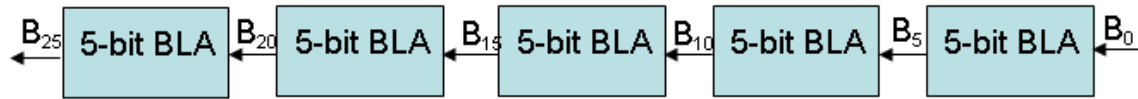
- $b_1 = g_0 + p_0 b_0$ (1)
- $b_2 = g_1 + p_1 b_1 = g_1 + p_1(g_0 + p_0 b_0) = g_1 + p_1 g_0 + p_1 p_0 b_0$ (2)
- $b_3 = g_2 + p_2 b_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 b_0$ (3)

The b_4 and b_5 bits can be computed in a similar manner as above.

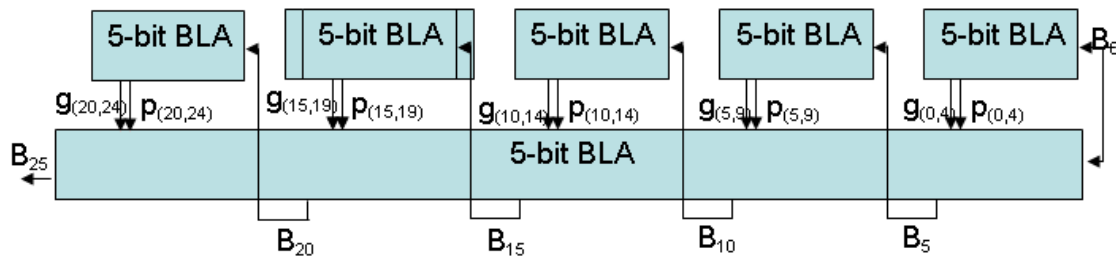
Putting it all together:

The borrow precomputation ideas enable us to theoretically derive the Boolean function for each borrow bit in a 25-bit BLA binary subtractor. However, because of fan-in and fan-out limitations, it is practically infeasible to directly implement a 25-bit BLA generator. Therefore, in this part of the lab, we will use two other methods to implement a 25-bit BLA binary subtractor. The simpler method just connects five 5-bit BLA binary subtractors in serial such that the borrow-out of each BLA subtractor serves as the borrow-in of the next-higher significant BLA subtractor. This 25-bit binary subtractor is denoted as a **single-level BLA subtractor** since it is constructed in such a way that borrows within each 5-bit subtractor are

computed by a 5-bit BLA generator based on the values of g 's and p 's, while the 5-bit borrows $b_5, b_{10}, b_{15}, b_{20}$, and b_{25} ripple through the 5-bit slices. The advantage of this method is that it requires no extra hardware besides the five 5-bit BLA subtractors. However, each 5-bit BLA subtractor still needs to wait for the next-lower significant BLA subtractor to generate the borrow, thus limiting the speed of this subtractor. A high-level schematic of this is shown below:



In contrast, the somewhat more complicated but faster method, the **two-level BLA subtractor**, uses an extra BLA generator in addition to the five 5-bit BLA subtractors in order to speed up the generation of 5-bit borrows $b_5, b_{10}, b_{15}, b_{20}, b_{25}$. A moment's reflection should show that the two-level BLA generator can be constructed in a manner similar to that shown below:



Obviously, the two designs of 25-bit BLA binary subtractor display distinct hardware complexity and timing characteristics. In this part of the lab, you are asked to implement **both** designs (of course by implementing the complemented borrow ripple throughout) and compare their timing characteristics. The detailed instruction for timing analysis will be discussed later in this writeup.

Implementation Guidelines

Now let us summarize the preceding information and try to provide you a somewhat sharper picture regarding the implementation of this design. Your implementation of the 25-bit complemented borrow subtractor should be composed of the following modules:

- **1 single-bit inverted borrow-generate and borrow-propagate subtractor**
- **1 BLA generator:** it should take as input the results of 5 generate and propagate functions and output the borrow for each single-bit subtraction as well as the generate, propagate and borrow for the 5-bit subtraction as a whole
- **1 5-bit subtractor:** this subtractor combines 5 of the single-bit modules and the BLA generator to produce a 5-bit BLA subtractor
- **2 25-bit BLA subtractors:** Both subtractors will combine 5 of the 5-bit subtractors to compute the difference. However, one will link the borrow out of one subtractor to the next in a ripple-borrow fashion, while the other will employ an additional BLA generator.

We have provided you with a 25-bit complemented borrow subtractor that uses the ripple-borrow mechanism to use as reference (and to compare with your timing results from the look-ahead implementations). You are asked to implement both the single-level and the two-level 25-bit BLA binary subtractors. Once you have finished both implementations, you are asked to furthermore compare their performance. In order to obtain the correct timings, you must manually specify the delays for each gate. We would like you to emulate the gate

timings specified in Table 2, which requires you to treat each gate as a module of its own and connect wires to each input and output port. We ask that you use structural verilog for your implementation; to specify a delay, therefore, enter the time in nanoseconds preceded by a “#” sign (examples can be found in the given ripple-borrow module). When you change the inputs, it will take some time for the effect of that change to reflect into the output. This difference in time is your total delay. Usually different input-output pairs have distinct delays. Even for the same input-output pair, its delay varies with distinct input combinations. The **maximum** delay value of each input-output pair is considered as its real delay value. After you finish your design, you need to measure the delays on the paths specified in Table 1 and fill in that table. In Table 1, x_i and y_i respectively denote the i^{th} bit of your two inputs (each input is 25 bits in total), b_i denotes the borrow-in for the i^{th} bit of your binary subtractor, and d_i denotes the i^{th} bit of your difference.

Table 1: Path delay of 25-bit borrow look-ahead subtractor

Path	Delay		Path	Delay		Path	Delay	
	Single-level	Two-level		Single-level	Two-level		Single-level	Two-level
b_0 to b_5			x_0 to b_5			y_0 to b_5		
b_0 to b_{10}			x_0 to b_{10}			y_0 to b_{10}		
b_0 to b_{15}			x_0 to b_{15}			y_0 to b_{15}		
b_0 to b_{20}			x_0 to b_{20}			y_0 to b_{20}		
b_0 to b_{25}			x_0 to b_{25}			y_0 to b_{25}		
b_0 to d_4			x_0 to d_4			y_0 to d_4		
b_0 to d_9			x_0 to d_9			y_0 to d_9		
b_0 to d_{14}			x_0 to d_{14}			y_0 to d_{14}		
b_0 to d_{19}			x_0 to d_{19}			y_0 to d_{19}		
b_0 to d_{24}			x_0 to d_{24}			y_0 to d_{24}		

Part 2: Static Hazards

This part of the lab concerns the design of static hazard free circuits. You will be returning to your familiar yet buggy environment of LogicWorks. As has been discussed in Section 8.4 of your textbook, static hazards are caused by signal transitions on two complementary signals (such as x and x'). If these two signals propagate through paths of distinct delays, they may exhibit identical values for short periods of time, possibly resulting in a signal glitch at the logic where these two signals reconverge.

Fundamentally, static hazards can be detected using K-Maps. A **static 1-hazard** occurs if there exist two adjacent **1-minterms** that are not covered by a common **product** term in a **sum-of-products** implementation. Similarly, a **static 0-hazard** occurs if there exist two adjacent **0-maxterms** that are not covered by a common **sum** term in a **product-of-sums** implementation. As you probably observed, the elimination of a static hazard is not a free lunch; it is achieved at the cost of redundant gates. Minimizing such a cost thus becomes an important goal in designing hazard-free circuits. For certain circuits whose behavior is easy to profile, it is possible to find the set of input transitions that will never occur during the regular operation of the circuit. This set of input transitions can be considered as *Transition Don't Cares*, and be utilized to minimize the redundant hardware needed in a hazard-free design. This is achieved by removing the redundant gates that only cover hazard-generating transitions in the *Transition Don't Care* set.

In this part of your lab, you are asked to design the **minimal** hazard-free implementation for three circuits. Circuit 1 and 3 must be implemented in the **sum-of-products** form while Circuit 2 must be implemented in the **product-of-sums** form. You will notice of course that Circuits 1 and 2 share the same Karnaugh Map. You have been given the following set of questions which should be of guidance to you in coming up with your minimal logic.

For Circuit 1: What extra logic is needed to avoid any static hazard, if

- the transition $1101(abcd_{current})$ to $0101(abcd_{next})$ is a *Transition Don't Care*.
- the transition $0101(abcd_{current})$ to $1101(abcd_{next})$ is a *Transition Don't Care*.

For Circuit 2: What extra logic is needed to avoid any static hazard, if

- the transition $1111(abcd_{current})$ to $1110(abcd_{next})$ is a *Transition Don't Care*.
- the transition $1110(abcd_{current})$ to $1111(abcd_{next})$ is a *Transition Don't Care*.

For Circuit 3, a set of *Transition Don't Cares* has been specified in the table on the right of its K-Map. While you are choosing the implicants (and there are multiple minimal covers as you can see, if static hazard handling was not an issue) to find the minimal logic, make use of the *Don't care transitions* to optimize the logic for the static hazard free implementation that you are asked to complete. We expect you to provide a hazard-free implementation with the least number of gates.

Karnaugh Map for Circuits 1 and 2:

cd	00	01	11	10
ab				
00	0	0	0	x
01	1	1	0	1
11	0	1	0	0
10	x	1	x	x

Karnaugh Map for Circuit 3:

cd	00	01	11	10
ab				
00	0	1	1	1
01	0	1	0	1
11	0	1	1	1
10	0	0	0	0

Transition Don't-Cares for circuit 3:

$abcd_{\text{current}} \rightarrow abcd_{\text{next}}$	
0001	$\rightarrow$ 0011
0101	$\rightarrow$ 0001
0110	$\rightarrow$ 0010
1101	$\rightarrow$ 1111
1110	$\rightarrow$ 0110
1111	$\rightarrow$ 1110

Deliverables:

- **Verilog Files:**

Part 1: Include a Verilog HDL file (.v) for each of the two 25-bit BLA subtractor circuits (with single- and two- level BLA generators), your 5-bit BLA subtractor, your 1-bit subtractor that generates your g and p values, and your BLA generator.

- **LogicWorks Files:**

Part 2: Include two distinct LogicWorks circuit files for circuits 1 and 2, each of which should contain 3 circuits: the minimal naïve (ie, hazard-prone) implementation of the circuit; the minimal hazard-free implementation of the circuit given the first transition don't-care; and the minimal hazard-free implementation of the circuit given the second transition don't care. Label each circuit accordingly.

For circuit 3, include a LogicWorks circuit file that contains the following: all possible minimal naïve implementations of the circuit; and the minimal hazard-free implementation given the transition-don't-care table (there is only one such optimal implementation).

- **Reports:**

For part 1, you should provide your measured delay data by filling in Table 1 for both implementations of your BLA subtractor circuits. (See Table 2: Gate delay list at the end of this writeup.)

For part 2, your report needs to clearly state how a transition being a *don't care* or not effects the logic needed to avoid static hazard for the Circuit 1 and 2. For circuit 3, describe which of the hazardous minimal implementations formed the basis of your final hazard-free minimal implementation, and explain how the transition don't cares effected your choice. Please present the corresponding boolean function representing the implicants present in the hazard-free circuit. Timing diagrams are an easy tool to visually observe if there is any static-hazard in your logic when the inputs transition from one value to another. For Circuit 3, if the *Transition Don't Care* table shown before is conversely used as the input transition table (*i.e.* use the input change from the value in the left column to the corresponding value in the right), we ask you to capture the timing diagram to show how the output changes when the input transitions. Some of these transitions will exhibit a glitch. Please identify which input transitions in the set that we provided you result in such static hazards, as well as which ones do not, and explain the observed behavior.

Table 2: Gate delay list

Name	Delay (ns)	Name	Delay (ns)
2-input AND	2.4	2-input NAND	1.4
3-input AND	2.8	3-input NAND	1.8
4-input AND	3.2	4-input NAND	2.2
5-input AND	3.6	5-input NAND	2.6
2-input OR	2.4	2-input NOR	1.4
3-input OR	2.8	3-input NOR	1.8
4-input OR	3.2	4-input NOR	2.2
5-input OR	3.6	5-input NOR	2.6
2-input XOR	4.2	2-input XNOR	3.2
Inverter	1.0		