

Lab Assignment 6

Combinational and Sequential Adder Circuits

Due December 14, Sunday, 4:30pm

Introduction

One important concept in digital logic design that you have undoubtedly dealt with already is that of a circuit's latency. Latency is the time the circuit takes to provide an output when a new set of inputs are presented. In a combinational circuit this is simply the propagation delay of the longest path from input to output. In a sequential circuit this is the number of clock cycles the circuit takes to produce the result multiplied by the clock period.

In this lab you will explore different implementations of a very common operation, i.e. addition, and how the various implementations affect the latency of the circuit. You will discover how the latency of the combinational components in a sequential circuit influences the choice of clock period used and what happens to the circuit's timing and outputs if this value is not properly specified.

This lab consists of two sections: (1) Implementation in *LogicWorks* of (i) a 25-bit ripple adder, (ii) a 25-bit *carry look-ahead adder*, and (iii) a 25-bit *two-level carry look-ahead adder*. (2) Implementation in *Verilog* of a 25-bit *synchronous carry look-ahead adder*.

Section 1 - LogicWorks

Ripple Adder

You have learned from class that a single bit full adder (FA) takes three inputs, two operands (A, B) and the carry-in (C_{in}) and produces two outputs, the sum (S) and carry-out (C_{out}). You have studied a handful of different implementations of a single-bit adder which have various latencies but for this lab we ask that you use the following implementation.

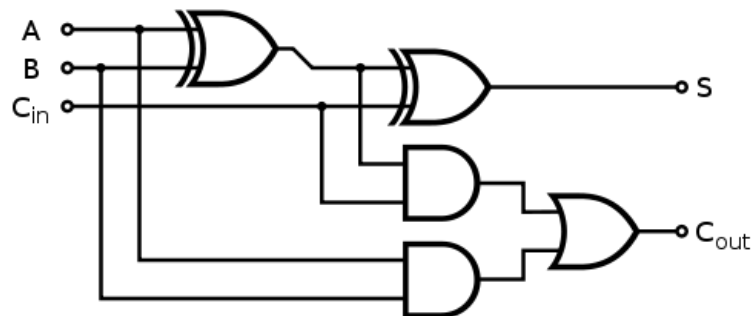


Figure 1: Single-bit FA Circuit

An n-bit binary adder can be implemented as a serial connection of FAs, arranged such that the carry-out of each FA serves as the carry-in for the next-higher significant FA. This type of adder is denoted as a ripple adder, since any change in the least significant bits c_0 , x_0 , or y_0 has to ripple through all the FAs.

For this lab you will implement a 25-bit adder using the ripple method described above.

Carry Look-Ahead Adder

The carry look-ahead (CLA) adder in some sense precomputes the carries of each bit, thus drastically speeding up the overall adder by enabling a number of operations to happen in parallel.

For a ripple 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 adder as this is the critical path and in fact still remains the critical path even in a CLA adder. This technique takes advantage of the fact that the computation for all of the carry values in the chain can be precomputed in parallel and be available in a much shorter time than the ripple method. Let's take as an example a 4-bit adder that computes $x_3x_2x_1x_0 + y_3y_2y_1y_0 + c_0$. We first define two functions as follows: (a) the carry-generate function $g_i = x_iy_i$, and (b) the carry-propagate function $p_i = x_i \oplus y_i$. Using these two functions, the four carries $c_1 \dots c_4$ can be expressed as follows:

$$c_1 = g_0 + p_0c_0 \quad (1)$$

$$c_2 = g_1 + p_1c_1 = g_1 + p_1(g_0 + p_0c_0) = g_1 + p_1g_0 + p_1p_0c_0 \quad (2)$$

$$c_3 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 \quad (3)$$

$$c_4 = g_3 + p_3c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0 \quad (4)$$

These four equations show that the four carries $c_1 \dots c_4$ can be computed directly from input bits $x_3x_2x_1x_0$, $y_3y_2y_1y_0$, and carry-in c_0 without incurring a ripple. In other words, the carry value for each bit can be generated simultaneously, which in turn allows the addition of each bit to be performed in parallel, thus drastically reducing the latency of the adder circuit! Figure 2 shows how these CLA generators can be daisy chained together to make larger adders but in doing so the circuit still suffers a similar, yet attenuated, delay as the ripple adder. The carry delay of c_4 which is needed in order to enable the next CLA generator to begin computation is still the critical path and use of larger fan-in gates causes the delay of this path (c_0 to c_4) to be greater than necessary. In order to speed up the generation of c_4 , the equation for c_4 can be transformed to:

$$c_4 = G_{0,3} + P_{0,3}c_0 \quad (5)$$

where $G_{0,3}$ and $P_{0,3}$ are defined in a form that can be precomputed as:

$$G_{0,3} = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 \quad (6)$$

$$P_{0,3} = p_3p_2p_1p_0 \quad (7)$$

This transformation speeds up the generation of c_4 by reducing the maximum fan-in requirements from 5-inputs down to 4-inputs for a 4-bit CLA. In the case of your implementation, a similar transformation will obviate the need for 6-input gates, enabling you to implement your design with gates of up to 5 inputs.

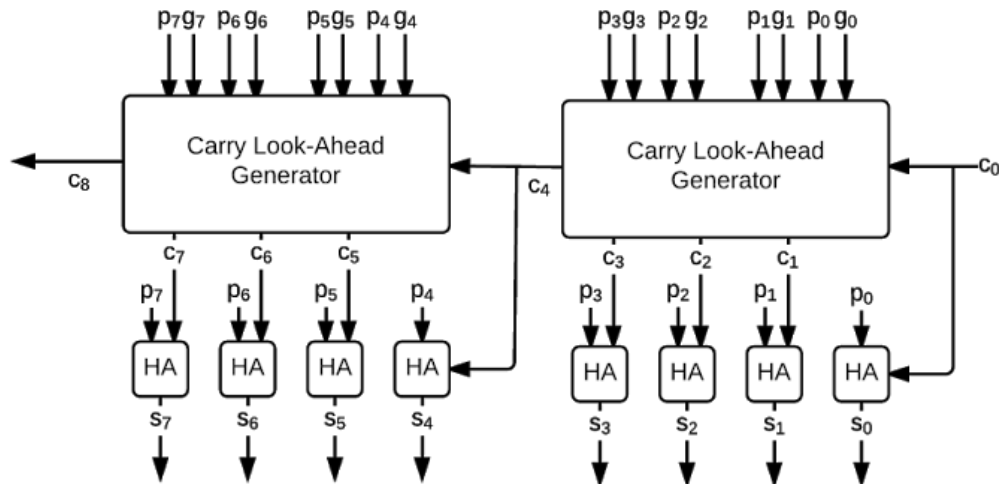


Figure 2: Example 8-bit CLA Adder

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

1. Carry-generate functions and carry-propagate functions for each bit
 - a. The carry-generate function $g_i = x_i y_i$ determines whether the two input values x_i and y_i **will generate** a carry to a higher bit.
 - b. The carry-propagate function $p_i = x_i \oplus y_i$ determines whether the two input values **can propagate** a carry coming from a lower bit to a higher bit.
2. A CLA Generator which 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).
3. Single-bit Half Adders (HA) for each bit, taking as inputs the carry-generate function p_i and the carry value c_i (provided by the CLA Generator), producing the sum $s_i = p_i \oplus c_i$.

The carry precomputation ideas enable us to derive the Boolean functions that can generate each carry bit essentially in parallel in a 25-bit CLA adder. However, because of fan-in and fan-out limitations, it is practically infeasible to directly implement a 25-bit CLA generator. Therefore, in this section of the lab, we will use two other methods to implement a

25-bit CLA adder. The simpler method just connects five 5-bit CLA adders in serial so that the carry-out of each CLA adder serves as the carry-in of the next-higher significant CLA adder. This 25-bit adder is denoted as a single-level CLA adder since it is constructed in such a way that carries within each 5-bit adder are computed by a 5-bit CLA generator based on the values of g 's and p 's, while the 5-bit carries c_5 , c_{10} , c_{15} , c_{20} , and c_{25} ripple through the 5-bit slices. An alternate approach that avoids even this attenuated ripple is discussed next.

2-Level Carry Look-Ahead Adder

The advantage of the single level CLA adder is that it requires a moderate amount of extra hardware (five 5-bit CLA generators replace the 25 single-bit adders) over what a ripple adder requires. However, each 5-bit CLA generator still needs to wait for the next-lower significant CLA generator to provide the carry-in despite our efforts on reducing this delay, thus limiting the speed of the adder. In contrast, the somewhat more complicated but faster, two-level CLA adder, uses an extra CLA generator in addition to the five 5-bit CLA adders in order to speed up the generation of c_5 , c_{10} , c_{15} , c_{20} , and c_{25} carries by computing them all simultaneously in the second level CLA generator.

To demonstrate this better and explain how the data flows, let's extend the previous example with the 4-bit CLA generators to a 16-bit adder and include a second level CLA generator. You can see in Figure 3 that the two operands still enter the first level CLA generators, which in turn produces the G 's and P 's that the second level operates on. These G 's and P 's are the same ones we derived earlier to speed up the generation of c_4 but we now generalize them to $G_{(i,i+3)}$ and $P_{(i,i+3)}$ and provide these bottom CLA generators outputs as inputs to the top level CLA generator. The second level CLA generator takes these G 's and P 's as inputs and produces carries c_4 , c_8 , c_{12} , and c_{16} all essentially simultaneously and provides them back to the first level CLA generators. From there the operation continues as if it were a single-level 16-bit CLA adder except now all 4 CLA blocks can operate essentially in parallel.

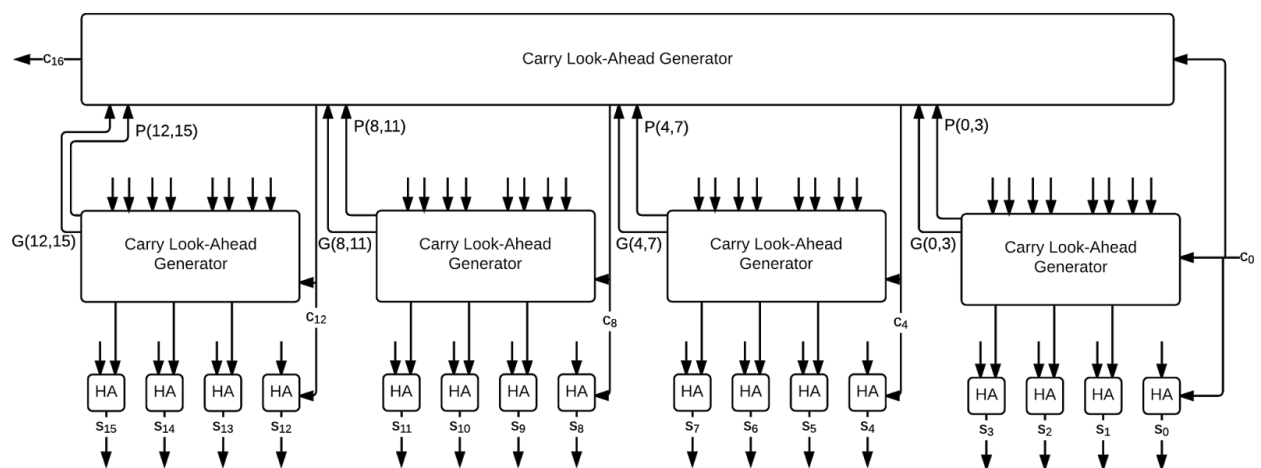


Figure 3: Example 16-bit 2-Level CLA Adder

Formulas (6) and (7) in their general form (which serve as the inputs to the second-level 4-bit CLA generator) become:

$$G_{i,i+3} = g_{i+3} + p_{i+3}g_{i+2} + p_{i+3}p_{i+2}g_{i+1} + p_{i+3}p_{i+2}p_{i+1}g_i \quad (8)$$

$$P_{i,i+3} = p_{i+3}p_{i+2}p_{i+1}p_i \quad (9)$$

The second-level 4-bit CLA generator produces c_i through c_4 by replacing g_i and p_i with $G_{i,i+3}$ and $P_{i,i+3}$ in formulas (1) through (4)

$$c_1 = G_{0,3} + P_{0,3}c_0 \quad (10)$$

$$c_2 = G_{4,7} + P_{4,7}G_{0,3} + P_{4,7}P_{0,3}c_0 \quad (11)$$

$$c_3 = G_{8,11} + P_{8,11}G_{4,7} + P_{8,11}P_{4,7}G_{0,3} + P_{8,11}P_{4,7}P_{0,3}c_0 \quad (12)$$

$$c_4 = G_{12,15} + P_{12,15}G_{8,11} + P_{12,15}P_{8,11}G_{4,7} + P_{12,15}P_{8,11}P_{4,7}G_{0,3} + P_{12,15}P_{8,11}P_{4,7}P_{0,3}c_0 \quad (13)$$

LogicWorks Combinational Adder Implementation Guidelines

Obviously, the three designs for the 25-bit CLA adders display distinct hardware complexity and timing characteristics. In this section of the lab, you are asked to implement all three of the designs and compare their timing characteristics.

Let us summarize the preceding information and try to provide you a somewhat sharper picture regarding the implementation of this design.

- **25 single-bit full adders:** used to implement the 25-bit ripple adder circuit.
- **25 carry-generate and carry-propagate functions:** you need AND and XOR gates to implement $g_i = x_i y_i$ and $p_i = x_i \oplus y_i$, respectively.
- **25 single-bit half adders:** each adder takes as inputs the carry-generate function g_i and the carry value c_i (provided by the CLA generators), and produces the sum $s_i = p_i \oplus c_i$.
- **5 (or 6) 5-bit CLA generators:** A total of five CLA generators are needed to implement a single-level 25-bit CLA adder, while an additional CLA generator is needed to implement a two-level 25-bit CLA adder.

A similar transformation that we showed for the 4-bit CLA generator will need to be effected in the case of your 5-bit CLA generators to carry c_5 . We leave the generation of the c_5 equation for the 5-bit CLA block and the analogous transformation to ensure the speedy generation of c_5 for you to figure out.

Once you have finished all three implementations, you are asked to compare their performance. To achieve this goal, you need to first set the delay of the gates according that you will be using to the delay values specified in Table 1. You can set the delay value of any gate by using the “*Simulation Params*” option in the “*Simulation*” menu of LogicWorks.

Name	Delay (ns)	Name	Delay (ns)
2-input AND	24	2-input OR	24
3-input AND	28	3-input OR	28
4-input AND	32	4-input OR	32
5-input AND	36	5-input OR	36
2-input XOR	42	Inverter	10

Table 1: Gate delay list

You can find the total delay of the circuit by reading the *timing display* in LogicWorks. When you change the inputs, it will take some time for the effect of that change to be reflected at the output. This difference in time is your implementation's latency. Usually different input-output pairs have distinct delays. Determining the set of input combinations and the order in which to apply them is known as test pattern generation and is beyond the scope of this lab; therefore, we have provided you (see Table 3 on last page) with an ordered set of test inputs that will exercise all the paths in your circuit. Run all of these test patterns **in the order given** and note the delays for each input-output pair after each new test pattern. You should only report the longest delay value that you observe during this testing process. The given values in Table 2 are to guide you in design by specifying what your circuit delays should be at each of these points. In Table 2, x_i and y_i respectively denote the i^{th} bit of your two 25-bit inputs, c_i denotes the carry-in for the i^{th} bit of your binary adder, and s_i denotes the i^{th} bit of your results.

Path	Delay (ns) Ripple, CLA, 2-Lvl	Path	Delay (ns) Ripple, CLA, 2-Lvl	Path	Delay (ns) Ripple, CLA, 2-Lvl
c_0 to c_5	192 / 48 / 48	x_0 (or y_0) to c_5	234 / 126 / 126	x_4 (or y_4) to c_5	90 / 138 / 138
c_0 to c_{10}		x_0 (or y_0) to c_{10}		x_4 (or y_4) to c_{10}	
c_0 to c_{15}		x_0 (or y_0) to c_{15}		x_4 (or y_4) to c_{15}	
c_0 to c_{20}		x_0 (or y_0) to c_{20}		x_4 (or y_4) to c_{20}	
c_0 to c_{25}		x_0 (or y_0) to c_{25}		x_4 (or y_4) to c_{25}	

Table 2: Path Delays of 25-bit Adder Circuits

Your design will be evaluated based on both correctness and elegance. You should always try to minimize the total number of gates. You should minimize the number of gates used and the whole design of your adders should be completed within 200 gates per design.

Given this significant number of gates needed in your implementation, it is almost impossible to put all the gates in one circuit. Therefore, in this lab assignment you are asked to package at least the FA and CLA generator into a component¹ so that you can reuse it multiple times. You can furthermore package the 5-bit CLA adder into another component. In

¹ The way of creating components in *LogicWorks* is presented in your *LogicWorks* tutorial.

this way, the highest level of your schematic would be composed of five 5-bit CLA adders and one additional CLA generator for the two-level 25-bit CLA adder circuit.

Section 2 - Verilog

In this section of the lab you will implement two 25-bit synchronous CLA adders in Verilog. The first adder will consist of one single-bit synchronous adder that is clocked 25 times to calculate the sum. The FA will be implemented as a Mealy Finite State Machine (FSM) consisting of 2 states, 2 inputs, 1 output and an asynchronous reset.

Your goal is to design an adder circuit with the shortest latency by reducing the clock cycle period to as small as possible while still yielding correct results. Note that the circuit should complete the entire 25-bit calculation in exactly 25 cycles meaning that the latency is actually 25 times your implementation's clock period. This type of design is often referred to as resource constrained, as we are constraining you to no more than 1 sequential full adder and asking that under this constraint you optimize your circuit for the lowest achievable latency.

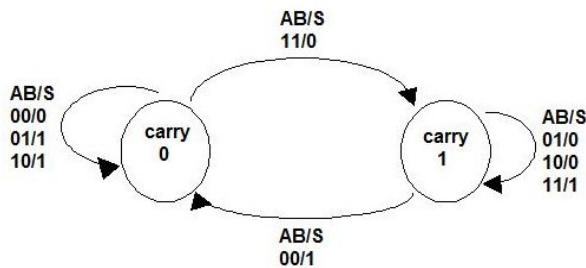


Figure 4: Single-bit FA State Diagram

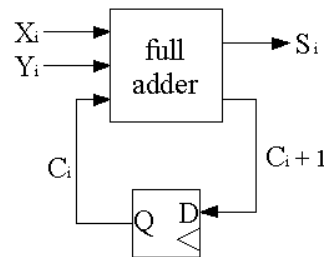


Figure 5: Single-bit Synchronous Adder

After you have completed the single-bit implementation, your next step will be to improve upon this design by extending the synchronous FA to take as input two 5-bit numbers and produce a 5 bit sum output. In the 5-bit FA FSM the next state logic will be the value of carry c_{i+5} instead of c_{i+1} . You must specify both of these in gate primitives only, but use **no behavioral Verilog**. Again your goal will be to reduce the latency of this circuit by designing the circuit to have the shortest clock cycle as possible. In this implementation the entire calculation should finish in exactly 5 cycles; therefore, the latency of this implementation will be 5 times the clock period.

Verilog Sequential Adder Implementation Guidelines

Use the provided Verilog files (*tb_adder25bit1FA.v*, *tb_adder25bit5FA.v*, *adder-25bit1FA.v*, *adder-25bit5FA.v*, *1fa.v*, and *5fa.v*) as guidelines for implementing each component and building the adder circuit and test bench. In the files **fa.v*, a template for an FSM that counts from 0 to 3 is provided and should be used as a guide for implementing your FA. In these files there are statements that specify delays between calculations; you should follow the same semantics when designing all of your modules and use the same values for each gate type as specified in Table 1. Remember when using two-level (or greater) Boolean logic statements that you must use the correct delay value for the entire statement. Not doing so could be reflected in our evaluation of your circuit, especially if we set the delay to the

Deliverables:

Section 2: Submit the Verilog files of both of your implementations including any additional modules you may have used in your solutions.

Test Cases:

[illegible]

Table 3: Ordered Test Pattern Set