

Lab Assignment 3

Error Detection/Correction

Due August 22, 2013, Thursday, 4:00pm

Introduction:

This lab assignment consists of two parts. In the first part, you are required to implement an error detection/correction circuit by slightly altering the Robertson's multiplier circuit you have designed in your Lab2 assignment. We will provide you with a correctly functioning multiplier circuit for this part (well, of course, after everyone has had a chance to turn in his/her lab2) in case you do not have a fully working circuit handy from the second lab assignment. In the second part, we ask you to derive alternative adder implementations based off of the fundamental adder equations.

Part 1: Error Detection/Correction Circuit

Physical failures of digital systems can lead to errors during data transmission. Oftentimes there may be a difference between the data that is transmitted and the data that is received. In order to deal with such problems, one can use some sort of coding system when transmitting data. One such code is the *2-out-of-5* code introduced in Chapter 1 of your text book (p. 21). Although the *2-out-of-5* code is inefficient in terms of the number of bits used, the code *does* allow you to detect all single bit errors. Keep in mind that the ability to detect errors is based on the *Hamming distance* between code words. In addition to the ability of detecting errors, some codes even allow errors to be corrected as well.

In this part of the experiment, you will build an error detection/correction circuit for the coding scheme in Table 1 by using LogicWorks. You see only 10 codewords in this coding scheme, represented by 11 bits each. This is indeed a rather inefficient scheme as only 4 bits would more than suffice to represent 11 different words but don't despair; the redundancy in these codewords will deliver certain error detection and correction capabilities.

Table 1: Error detection/correction codewords

<u>Value</u>	<u>Codeword</u>	<u>Value</u>	<u>Codeword</u>
0	11001000011	5	01001110100
1	11000100101	6	00101111000
2	10100101001	7	11000010110
3	01100101010	8	10100011010
4	01010110010	9	10010011100

1.1 Theory

Invariants:

To implement the appropriate detection and correction circuit, you need to understand the properties of these codewords. These properties are typically referred to as *invariants* in coding theory; for example, the invariant of the *2-out-of-5* code we examined in your textbook is that there are exactly two 1's in any codeword (and consequently three 0's). This invariant enables the *2-out-of-5* code to detect single bit errors (and all odd number of errors), as a single flip of any bit will violate the invariant of exactly two 1's.

On the code that we show you above there are actually two *invariant* properties embedded;

1. *Total number of 1's*: each codeword consists of exactly six 0's and five 1's. This invariant alone provides a single error detection capability, in a manner similar to the *2-out-of-5* code case. Actually, a moment's reflection yields the fact that this invariant (just as in the *2-out-of-5* code) helps detect any *odd* number of errors.
2. *Sum of indices*: if we assign an index of 1 to 11 to each bit (from least significant bit to most significant) and perform a summation of indices where a 1 appears and subtract the indices where a 0 appears, we will get a total sum of -4.

Examples:

Indices:	11 10 9 8 7 6 5 4 3 2 1
Codeword 1:	1 1 0 0 0 1 0 0 1 0 1
Indices for 1s:	$1+3+6+10+11 = 31$ (Sum1)
Indices for 0s:	$2+4+5+7+8+9 = 35$ (Sum2)
Sum1-Sum2 =	-4

Indices:	11 10 9 8 7 6 5 4 3 2 1
Codeword 5:	0 1 0 0 1 1 1 0 1 0 0
Indices for 1s:	$3+5+6+7+10 = 31$ (Sum1)
Indices for 0s:	$1+2+4+8+9+11 = 35$ (Sum2)
Sum1-Sum2 =	-4

Value Embedding Property:

A moment's reflection (oh, well, maybe a bit more than a moment) should convince you that the assignment of values to codewords is not arbitrary and actually you can identify the value of the codeword by summing the appropriate bits. If you stop the accumulation for an instance, at the end of summing the least significant 5 bits, you would at that point be staring at twice the value of the codeword minus an offset of 9. Well, let's try it and see.

Examples:

Indices:	11 10 9 8 7 6 5 4 3 2 1
Codeword 7:	1 1 0 0 0 0 1 0 1 1 0
Indices for 1s in the least significant 5 bits: 2+3+5=10 (Sum1)	
Indices for 0s in the least significant 5 bits: 1+4 = 5 (Sum2)	
$((\text{Sum1}-\text{Sum2})+9)/2 = 7$ (equal to the index of the codeword)	
Indices:	11 10 9 8 7 6 5 4 3 2 1
Codeword 9:	1 0 0 1 0 0 1 1 1 0 0
Indices for 1s in the least significant 4 bits: 3+4+5 = 12(Sum1)	
Indices for 0s in the least significant 4 bits: 1+2 = 3 (Sum2)	
$((\text{Sum1}-\text{Sum2})+9)/2 = 9$ (equal to the index of the codeword)	

Calculating the Invariants/property:

Now that we have identified two invariants and the value embedding property, we need algorithms to calculate them. You will be slightly altering your Robertson's multiplier from Lab 2 in order to compute these invariants/property. Specifically, you are allowed to change the logic controlling when an operation is performed and which operation is performed.

Invariant #1: The number of 1's in a codeword can be easily calculated by the following algorithm: starting from the rightmost bit, we add a 1 for each of the 1-bits, and subtract a 1 for each of the 0-bits. It is apparent that the resulting sum will be a -1 for every codeword given above, as each codeword consists of 6 0-bits and 5 1-bits.

Invariant #2: In order to check Invariant 2, we only need a small change on the above algorithm. Instead of adding/subtracting 1 for each bit, now we need to add the index for each of the 1-bits, and subtract the index for each of the 0-bits. It is apparent that for a correct codeword, this algorithm will yield -4.

In sum, the operations performed for checking both invariants are listed as follows:

Table 2: Rules for calculating the two invariants

Invariant #1 – Total number of 1's		Invariant #2 – Sum of indices	
$C_i = 0$	Subtract 1	$C_i = 0$	Subtract i (index of C_i)
1	Add 1	1	Add i

Embedded Value: Since the embedded value is attained by summing up the indices for the least significant five bits, we can apply exactly the same set of rules for invariant 2 checking in the calculation of the embedded value. However, instead of performing 11 add/subtract operations as in calculating invariant #2, only the first 5 operations are needed while the effect of all the subsequent operations must be ignored.

Error Detection and Correction:

How can the rules described above be helpful in detecting errors? Recall that all codewords have two common invariants. For invariant #1, all codewords have a total of five '1's. Therefore, we can detect any *odd* number of errors, as this situation will change the number of 1s in the codeword. For invariant #2, an error may change the sum of indices when the rules are applied. Obviously, if a single bit flips from 0 to 1 or from 1 to 0, adding the indices of the 1's and subtracting the indices of the 0's would no longer sum to 1. Furthermore, because the indices of any two bits are distinct, any two bits of errors can also be detected by checking the *sum of indices*. It is possible to flip three bits of a codeword (for example, bits 1, 2, 3) and still get a sum of 0 for the second invariant. However, the first invariant can always detect this type of errors as it can detect any *odd* number of errors. Therefore, the two invariants together provide a capability of detecting *all errors up to 3*, implying that the code listed in Table 1 is obviously **at least** a distance-4 code. You can actually verify that the code is indeed **exactly** a distance-4 code by finding two codewords in the table that differ exactly by 4 bits.

The **value embedding property** provides us the capability of further checking if a codeword is correct or not. As the result for a correct codeword should lie in the range of $[0, 9]$, a codeword must be incorrect if it yields a result which falls outside this range.

Yet perhaps the simplest form for detecting errors is the calculation of the invariants for a codeword, and comparing the actual result with the expected one for each invariant. On the other hand, if we are interested in correcting these errors, we need to perform an additional step. Basically, a minimum *Hamming distance* of 4 between any two codewords provides the ability to correct any single errors. To achieve this goal, we thus need to understand how a single error changes the invariants.

An invariant can be used for error correction if the position of the bit that has been corrupted is preserved in the calculation of that invariant. Among the two invariants we have, which one can be used for error correction? Let's examine the *total number of 1's* first. If the j^{th} bit which should have been a 1 is flipped to a 0, the value of the first invariant would be changed to -3. Likewise, if it was flipped from a 0 to a 1, the first invariant would be changed to +1. As you can see, this change does not preserve any information of the index j . Therefore we need to turn to the *second invariant* to see if there is any chance of identifying the faulty bit.

If the j^{th} bit which should have been a 1 is flipped to a 0, the index of that bit, j , is subtracted although it should have been added to the *sum of indices*, thus resulting in the value of the second invariant being **$2j$ less** than it should have been. Similarly, if the j^{th} bit which should have been a 0 is flipped to a 1, the value of the second invariant would be **$2j$ more** than it should have been. In either case, the *sum of indices* will therefore deviate from -4 by **twice** as much as the index value. This implies that if we add 4, compute the absolute value of the resulting sum, and finally divide it by 2, we would exactly get the index (position) of the error bit.

1.2 Implementation

As you may have noticed, calculations for both invariants and the embedded value are very similar to the calculations performed by the Robertson's multiplier you implemented in Lab 2. To check for the two invariants and the embedded value, we are examining one bit at a time and performing addition or subtraction accordingly. Similarly, our multiplier of Lab 2 performed addition or subtraction depending on the individual bits of the multiplier. The difference though is that in the error detection/correction circuit, the value that should be added/subtracted is a constant **1** (for invariant #1) or the **index of the bit** being analyzed (for invariant #2 and the embedded value) rather than the value of the multiplicand. Actually there is no multiplicand

operand in this lab; error detection/correction just operates on a single word. Another difference here is that no shift operation is to be performed for the intermediate sums. This is because for multiplication, the shift of the partial product is to account for the doubling of the bit weights in a positional number system. In this coding scheme, however, the bit weights keep on increasing by 1 and not as a power of the radix as we proceed from the rightmost to the leftmost bit, thus eliminating the necessity for shift operations.

The strong similarity between the error detection/correction and multiplication operations suggests that an easy way to implement this lab assignment is to take the previous lab and make the appropriate modifications. For this reason, we will provide you a fully working Lab 2 (once all the late labs have trickled in), in case you do not have one of your own.

Circuit Overview:

Given an input word, the circuit that you will implement should check whether the two invariants and the value embedding property are satisfied. You can implement the various checks through a *two-phase algorithm*. In the first phase, your circuit should compute the number of 1's and check whether invariant #1 holds or not. The second phase examines both invariant #2 and the value embedding property, as well as corrects any single bit error.

The Control Unit from Lab 2 is still useful here, as it will direct your circuit to perform 11 operations and shifts, but your circuit needs to go through the 11 operations twice, with each run checking an invariant or a property. For this reason, we are providing you with a "Super Controller", which will attach to the inputs of your controller and simply direct it to perform its duties twice. It also has an output that lets your circuit know which phase it is in. You are asked to hook up the two controllers to implement the control unit.

The Super Controller:

Since the algorithm consists of two phases now, a new controller is needed in this lab. The basic operations within either phase are essentially the same, so rather than implementing a new controller, we add a master controller, called the *SuperController*. The *SuperController* is responsible for controlling our original controller in Lab 2 and it also keeps track of the phase information. While connecting the two control units, you should pay attention to the following:

- The SuperController will begin the error detection/correction process when its **Begin** input makes a transition from 1 to 0, which will be controlled by the user (a switch). It will start the first phase by setting its **BeginPhase** output to 0. Obviously, the Controller should take this input as its **Begin** input. The SuperController also broadcasts a signal **PhaseOne** to indicate the current phase information. In the first phase, this signal outputs a '1'.
- When the Controller raises the **End** signal for the first time, this shows the SuperController that the first phase has been completed. The super-controller has a **PhaseEnd** input to read this information. So it initializes the second phase by setting its **BeginPhase** output first to 1 and then to 0 again. In the second phase, the SuperController outputs a '0' on its **PhaseOne** output, indicating that we are currently in the second phase.
- The output signals and the cycle information within each phase are the same as Lab 2, i.e. you can refer to the table in Lab 2 to remember what happens at which cycle.
- When the Controller raises the **End** signal at the end of the second phase, the SuperController understands that this is the final time that the **End** signal will have been generated. The error detection & correction is over now and the SuperController sets its **End** output to '1' to indicate that the full operation has been completed.

Implementation Guidelines:

Beyond the Robertson's-like algorithm, each phase needs to have an additional datapath connected to the accumulator to perform some final operations to detect and correct errors.

Phase 1:

The only thing the Phase 1 datapath needs to do is to compare the accumulator with the constant numeral -1 and produce a one bit result (1 for correct, 0 for incorrect). You need to perform this comparison using only two-input logic gates (no more than 11 gates would be optimal). However, as the Phase 2 algorithm needs to reset the accumulator in order to perform its calculation, the value of the accumulator needs to be stored at the end of Phase 1. You'll need a flip-flop to hold this value. The **PhaseOne** signal from the controller can be used to control the loading operation of this flip-flop. The flip-flop value can be continually overwritten as the accumulator changes until Phase 2 is reached; the final value will be the one displayed when Phase 1 is complete. The output of this flip-flop can be compared with the numeral -1 to determine whether there are odd-bit errors in the input codeword.

Phase 2:

In Phase 2, the circuit will examine our second invariant, which can be checked in a manner similar to the check of the first invariant: we reset **A** to 0 at first and then perform the operations listed in the right part of Table 2 step by step. Our goal is to be able to correct single-bit errors, as well as detect up to 2-bit errors using this phase.

To detect errors, we only need to compare the accumulator with the constant numeral -4 at the end of Phase 2 using only basic logic gates. But what about correcting single-bit errors? As we've seen from the explanation of the invariant above, if there is a single bit error at the j^{th} bit, the content of **A** should be equal to 2^j-4 or -2^j-4 at the end of the second phase. How do we use this value to correct the corrupted bit and output a corrected codeword? There are several steps to follow: first we need to compute the index of the corrupted bit. This means that we need to add 4 to the accumulator's output, take its absolute value, and then to divide it by two. Given the position of the faulty bit, we need to perform the correction step, which can be achieved in two different ways. Each step is explained below.

Absolute Value:

There are several ways to take the absolute value of a numeral represented by a series of bit lines in LogicWorks. The method we recommend is to use an adder/subtractor to selectively compute $A - B$ or $A + B$, wherein A is always 0. If B is positive, we add, resulting in B , whereas if B is negative we subtract, resulting in $-B$, with either case thus providing a positive value. To do this variable operation we need to connect the A/S line to the sign of B . This should provide you a device that effectively takes a single input into B and outputs its absolute value. For the purpose of minimizing the hardware cost, you are encouraged to explore the possibility of reusing the adder/subtractor that is used for invariant calculation for the absolute value computation.

Divide by Two:

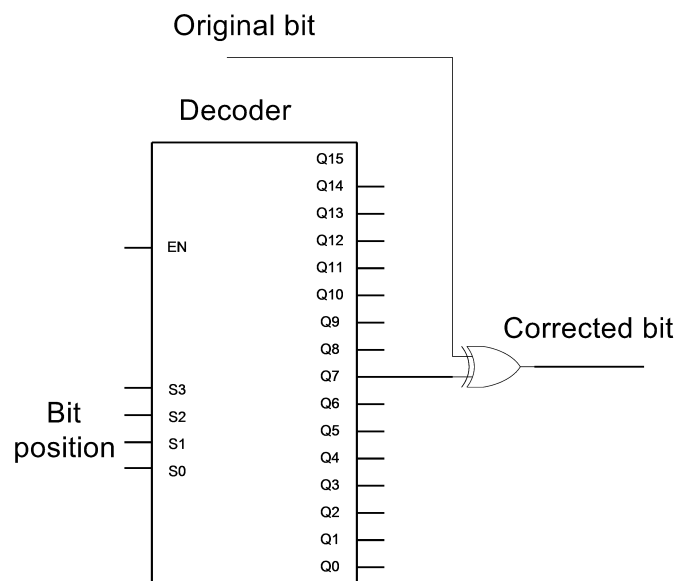
With binary numbers, multiplication and division with powers of 2 is very convenient, i.e. we add zeros to or cut off zeros from the right hand side. In this exercise, we need to divide by 2, which is essentially equivalent to cutting the LSB off. We can accomplish this by simply connecting the adder to the decoder (see correction part below) accordingly.

Correction:

At this point in the datapath, we'll have our number: the index of the corrupted bit of the input. We basically need to flip the erroneous bit and display the correct result. There are two parts to

this process. First, what we have is a binary representation of the index of the faulty bit, whereas what we need is one line per bit of input. For example, if the third bit has been corrupted, our datapath at this point would contain the value “00011”. We can use a decoder to transform this binary number into a form in which the third bit of the decoder outputs a value 1, thus helping to correct bit #3 of the codeword by flipping it.

How does this toggling-based correction work? By connecting each output pin of the decoder from Q1 to Q11 (Q0 is active when there are no errors) to one of the inputs of an XOR gate, and each bit of the input word to the other pins, we can perform conditional bit flipping on each line. This is shown in the figure below in the context of a 4-to-16-bit decoder, which is sufficient for this lab.



For a bit that does not have an error, the decoder output pin corresponding to that bit of the input will be zero. If that is the case, the original input bit will remain unchanged. If the bit was corrupted, however, the decoder will output a one on this line, and whatever value came from the input will be flipped.

The outputs of the XOR gates can now be attached to hex displays and the corrected codeword displayed.

Compute Embedded Value:

As the embedded value can be computed in the same way as we compute invariant #2, we can get this value through capturing the accumulator after the fifth summation step. This means that we need a flip-flop to store the accumulator value immediately after the fifth summation step is performed. But how do we know which step the circuit is currently performing? Well, remember that we use a counter to keep track of the index value of each bit. This index value is equivalently a record of the number of steps we have performed. Hence we can use its value to control the loading operation of the register. More concretely, the load input of the register should be activated only at the 'Load' cycle of the sixth step. In this cycle, the accumulator will possibly be updated with the summation result of the sixth step, and its original value (the result of the fifth step) can be loaded into the register.

As described in the Value Embedding property, the register value we attained needs to be divided by two for computing the embedded value. This operation can be implemented just by

dropping the least significant bit, as discussed in the *Divide by Two* part. The resulting value for a codeword should lie in the range of [0, 9].

1.3 Components

Since the Lab 2 implementation is used as a basis for this lab, we inherit most components provided in the Lab 2 library. There are four new components you may need in this lab. We provide them to you in the Lab 3 library file.

1. Lab 2 components that continue to be used in this design

You can use the 11-bit **multiplier** register in your Lab 2 implementation to store the 11-bit codeword. The **accumulator** register will still be used to store the partial sum of your error detection computation. The **Adder/Subtractor** you have developed for Lab 2 is still needed to perform addition/subtraction during error detection/correction. The **controller** in Lab 2 will be used as the datapath-controller in this design.

2. Lab2 components that will be replaced in this design

The **multiplicand** register in the design of Lab 2 will be replaced by a counter as we need to increment our indices by 1 as we traverse through each bit of our codeword. As the lowest bit of our original codeword is denoted as index 1, this counter should start counting from 1. As we need to count from 1 up to 11, a *4-bit counter* provided in the LogicWorks standard Library would more than suffice (and of course, a 3-bit one would fall just short of the requirements).

3. New components needed, either provided or available in LogicWorks library

- A **super-controller** to be used for controlling the two phase operation by indicating the start and the end of each phase. Within each phase, the operation of the circuit is actually controlled by the controller that you have used in Lab 2. You need to hook up the two controllers to enable them to operate correctly.
- A **4-to-16 decoder** to be used in the design of error correction logic
- A **11-bit comparator** to be used for comparing the calculated and the expected values.

4. New components that you need to design and insert into the circuit

- The **Error Detection** logic which should output a 1 if an error is detected in either the first or the second phase.
- The **Error Correction** logic which corrects the input word based on the value in **A** at the end of the second phase.
- The logic that captures the accumulator value immediately after the fifth summation step in Phase 2 and converts it to the embedded value.

Items to be displayed:

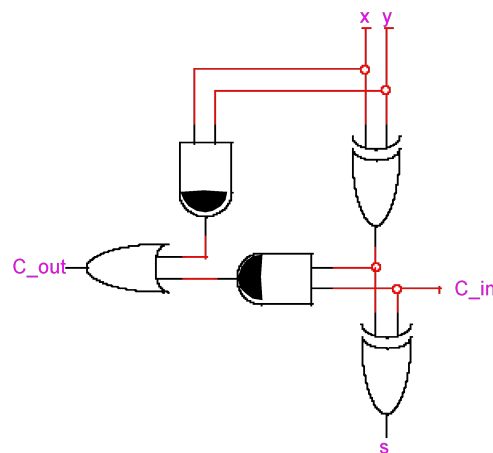
Your circuit should display the following results:

- The input word (on 2 hex keyboards);
 - Two buzzer signals showing the error detection result in each phase (on binary probes);
 - The content of **A** after the second phase (on 2 hex displays);
 - The embedded value calculated in Phase 2 (on 1 binary probe and 1 hex display);
 - The corrected code word (on 2 hex displays) according to the second phase.
-

Part 2: Adder Implementations

The objective of this part of the lab is to strengthen your understanding of the implementations of a fundamental arithmetic operation, namely, the adder. In addition to providing you expertise with the implementation of this fundamental arithmetic operation, we would like you to also develop a bit of an insight into the manipulation of Boolean expressions and for this reason we prod you along in steps to think about variant implementations of these operations while nonetheless keeping intact the structure and interconnections of the basic form of the adder implementation. The basic form of the adder implementation that you will be using as a template is the one below.

Basic adder implementation



As you remember, the function for the sum $s = x \text{ xor } y \text{ xor } C_{in}$, while the function for the carry-out $C_{out} = xy + xC_{in} + yC_{in}$. The main insight that motivates the above implementation is that the carry function can be rewritten through Boolean transformations as follows:

$C_{out} = xy + xC_{in} + yC_{in}$	Given
$C_{out} = xy + (x + y)C_{in}$	Distributivity
$C_{out} = xy + (xy + x'y + xy')C_{in}$	Definition of OR
$C_{out} = xy + xyC_{in} + (x'y + xy')C_{in}$	Distributivity
$C_{out} = xy + (x'y + xy')C_{in}$	Absorption
$C_{out} = xy + (x \text{ xor } y)C_{in}$	Definition of XOR

This enables the cannibalization of one of the XOR gates used in the sum function, which is simply the three-way xor of x , y , and C_{in} . Of course, you are probably wondering if any additional transformations can be made to the above adder to yield other more interesting cannibalizations. One can start by noticing that the sum, generated by two XOR gates, can be equivalently represented by two XNOR gates (so $s = x \text{ xnor } y \text{ xnor } C_{in} = x \text{ xor } y \text{ xor } C_{in}$). As you learned in the theory course, $a \text{ xnor } b = ab + a'b'$, and decomposing the sum into this form can open the doors up to further cannibalizations, which we ask that you explore in this portion of the lab.

One additional concept we ask you to consider stems from the observation that the positive version of gates (AND, OR, XOR) are slower than their negative forms (NAND, NOR, XNOR), as shown in Table 3. We ask that you combine this observation with the aforementioned observation about XOR and XNOR equivalence to *come up with an adder implementation that possesses exactly 6 NAND gates and 2 OR gates*. As a hint, take a look at the bolded version of the two-input xnor function:

$$a \text{ xnor } b = ab + a'b' = ((ab)'(a'b'))' = \mathbf{((ab)'(a + b))'} = (a + b)(a' + b') = (a+b)(ab)'$$

After performing your transformations for your new adder, you probably have realized that the two-input xnor form $(a+b)(ab)'$ can be helpful in *generating an adder form that utilizes 4 AND gates, 3 OR gates, and 2 inverters, and we ask that you show off your newfound discovery in this section as well*. Of course, you remember that the sum is simply $x \text{ xnor } y \text{ xnor } c_in$, so by using two instances of $(a+b)(ab)'$, you will have already used 4 AND gates, 2 OR gates, and 2 inverters to generate the sum. This means you simply need to add 1 more OR gate to generate the carry-out.

Additionally, in this section of the lab we ask that you use the structural description capabilities of Verilog. This is in contrast to your previous lab wherein you used the behavioural description capabilities which ended up automatically synthesizing the gate-level structural description. The structural Verilog that you will be using will enable you to provide directly the gate level description and furthermore the timing attributes of the gates. In order to obtain the correct timings, you must manually specify the delays for each gate by threading each gate as a module of its own and connecting the wires to each input and output port. A delay is specified by entering the time in nanoseconds preceded by a “#” sign. An example for the full adder described previously is shown below:

```
`timescale 1ns / 1ps

module FullAdder(
    input x_in,
    input y_in,
    input c_in,
    output s_out,
    output c_out
);

    wire temp0;
    wire temp1;
    wire temp2;

    xor#4.2(temp0, x_in, y_in);
    xor#4.2(s_out, temp0, c_in);
    and#2.4(temp1, temp0, c_in);
    and#2.4(temp2, x_in, y_in);
    or#2.4(c_out, temp1, temp2);

endmodule
```

When you change the inputs, it will take some time for the effect of that change to reflect in the output. This difference in time is your total delay. You will observe that typically different input-output pairs have distinct delays. Even for the same input-output pair, its delay may vary with distinct input combinations. The **maximum** delay value of each input-output pair is considered as its real delay value. Please compute the carry-to-carry delay for both of your adders, and show them in your report. Please use Table 3 below for your delay values.

Table 3: Gate List

Name	Delay (ns)	Name	Delay (ns)
2-input AND	2.4	2-input NAND	1.4
2-input OR	2.4	2-input NOR	1.4
2-input XOR	4.2	2-input XNOR	3.2
Inverter	1.0		

Deliverables:

- **LogicWorks Files:**

Part 1: As part of your submission, include a LogicWorks design file (.cct) that contains a complete error detection/correction circuit using the components provided. Be sure to test your circuit with several different kinds of inputs to detect problems.

Part 2: Please include two verilog design files (.v). One of the verilog files should contain the implementation of the adder containing 6 NAND gates and 2 OR gates, and the other should contain an implementation of the adder containing 4 AND gates, 3 OR gates, and 2 inverters.

- **Reports:**

For part 1, you should submit a report explaining how you designed the error detection/correction components of your circuit. You also need to provide explanations for the error detection and correction capabilities of your circuit. Specifically, you need to test the cases of 1-bit, 2-bit and 3-bit errors respectively, and report the behavior of each error detection/correction phase.

For part 2, you should provide the corresponding Boolean equations of your logic transformations. Additionally, you should compare the two adder implementations in terms of carry-to-carry delay; identify the carry-to-carry delay for each adder and include it in the report.