Joshua Wheeler

A11362716

**Homework #1**

Problem #1:

Part A –

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pair | Multiplicand | Multiplier | Robertsons #OPs | Booths #OPs |
| 1 | 00111001 | 00011100 | 3 | 2 |
| 2 | 11111001 | 01101111 | 6 | 4 |
| 3 | 00010110 | 11001101 | 5 | 5 |
| 4 | 11010110 | 10100110 | 4 | 5 |

Part B –

If the adder is stuck at zero, then when a correction step is needed the result will be wrong. The correction step is needed when both multiplicand and multiplier are both negative or when the multiplier is negative and the multiplicand is positive. The result will be right otherwise. Pairs 1, 3 and 4 do not work since they violate the conditions to being right. Pairs 1, 3 and 4 all require a subtraction step.

Part C –

In this algorithm the multiplier is what determines whether the product will be correct or incorrect. What we want to avoid is performing a subtraction step when we should have done addition. This means the multiplier can never run into a bit pairing of 01, or else an incorrect result will be produced. Pair 3 is the only pair in the list that satisfies this conditions and thus pairs 1,2,4 & 5 will fail.

Part D –

The possible pairs are pairs 1, 3 and 4. Pair 5 does not work because there is an overflow in at least one of the partial sums. Pair two does not work because the second shift operation in the algorithm should be logical, but since sign extension is always performed we erroneously would end up doing an arithmetic shift. So pairs 2 & 5 do not work.

Part E –

All 5 pairs would display a correct product because sign extension is always performed during any shift in Booths algorithm.

Problem #2

Part A –

1-bit detection: This does have detection with 1-bit. If I change any one bit in the base-3 string, at least one of the equations will have to fail due to the variable dependencies. Changing one number will have to offset another equation that that number is also in. If it is a parity bit that is changed then it will output the wrong result in the equations as well.

2-bit detection: This also works because of the way the dependencies are set up between the variables. Changing two values would always break the equation that only has one of the two values which were changed.

3-bit detection: This level of bit detection is not possible. An example of its impossibility is by analyzing the string 0000000 and changing bit positions m1, po and p1 from a 0 to a 1. Then the equations would both check out.

For the original eqn the first string would check out as p0 = 0 = 0, p1 = 0 = 0 and p2 = 0 = 0. The modified string would also check out as p0 = 1 = 1, p1 = 1 = 1 amd p2 = 0 = 0. The equations always evaluate to true and an error would not have been detected.

Part B –

1-bit error correction is fortunately a possibility with this arrangement. If one bit changes then a unique set of parity equations would fail. Therefore each bit has a unique mapping to its corresponding parity equations that failed and therefore can be detected.

The mapping is as follows:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| M0 | M1 | M2 | P0 | M3 | P1 | P2 |  |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | Bit # |
| P0p2p1 | P0p1 | P0p2 | P0 | P1p2 | P1 | P2 | Parity Eqns |