Lab 3

Part 1: Error Detection/Correction Circuit

– Implement an error detection/correction circuit by altering the Booth’s multiplier circuit you have designed in Lab2

Part 2: Boolean Algebra Simplifier

– Implement a Java program that simplifies Boolean Expressions utilizing the Absorption and Consensus Theorems

Error Detection/Correction Circuit

Codewords			
0	001111000110100	-0	110000111001100
1	001110100111000	-1	110001011010001
2	001101101011000	-2	110010011001001
3	001011101101000	-3	110100011000101
4	000111101110000	-4	111000011000011

Invariants of The Code

- *Total number of 1's:* each code-word consists of exactly eight 0's and seven 1's.
  - Capability of detecting any *odd* number of errors.
- *Sum of indices:* If we assign an index of 1 to 16 to each bit, add the indices where a 1 appears and subtract the indices where a 0 appears, we will get a total of 0.

  Examples:

Indices:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Codeword 2:	1	1	1	0	0	0	0	1	1	0	0	0	0	1	1

Indices for 1s: 1+2+7+8+13+14+15= 60(Sum1)
Indices for 0s: 3+4+5+6+9+10+11+12= 60 (Sum2)
Sum1-Sum2 = 0

Error Detection & Correction

- **Invariant #1**
  - Detects any *odd* number of errors
- **Invariant #2**
  - Detects any single bit errors
  - Detects any 2-bit errors
  - Pinpoints the position of the corrupted bit

Reason: *With single bit error, the resulting sum will deviate from 0 by* ***twice*** *as much as the index value*

Examples:

Indices:	16	15	14	13	**12**	11	10	9	8	7	6	5	4	3	2	1
Codeword 2:	0	1	1	0	**0**	0	1	1	1	1	0	0	0	1	1	0

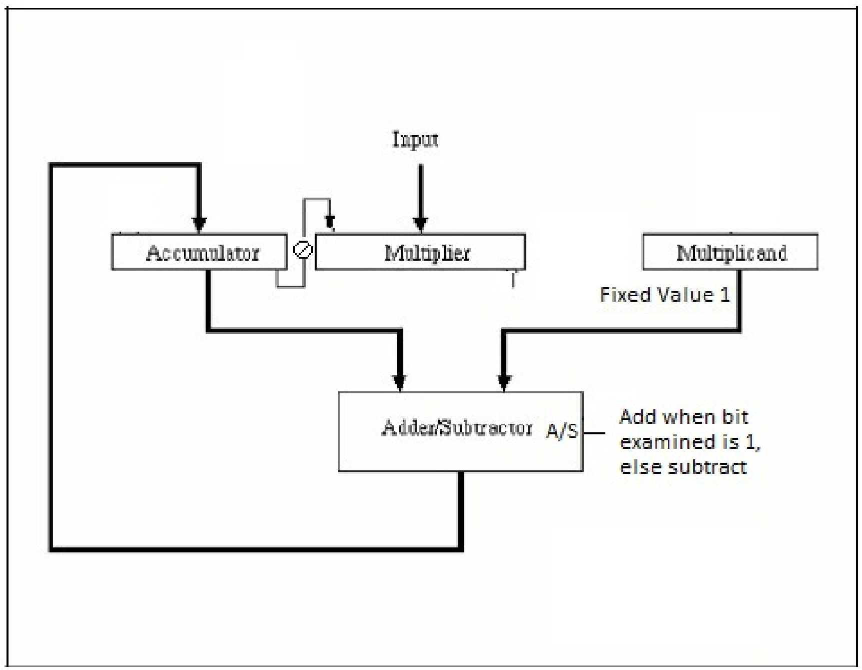
Indices for 1s: 2+3+7+8+9+10+12+14+15= 80(Sum1)

Indices for 0s: 1+4+5+6+11+13+16= 56 (Sum2)

Sum1-Sum2 = |24|/2 = 12

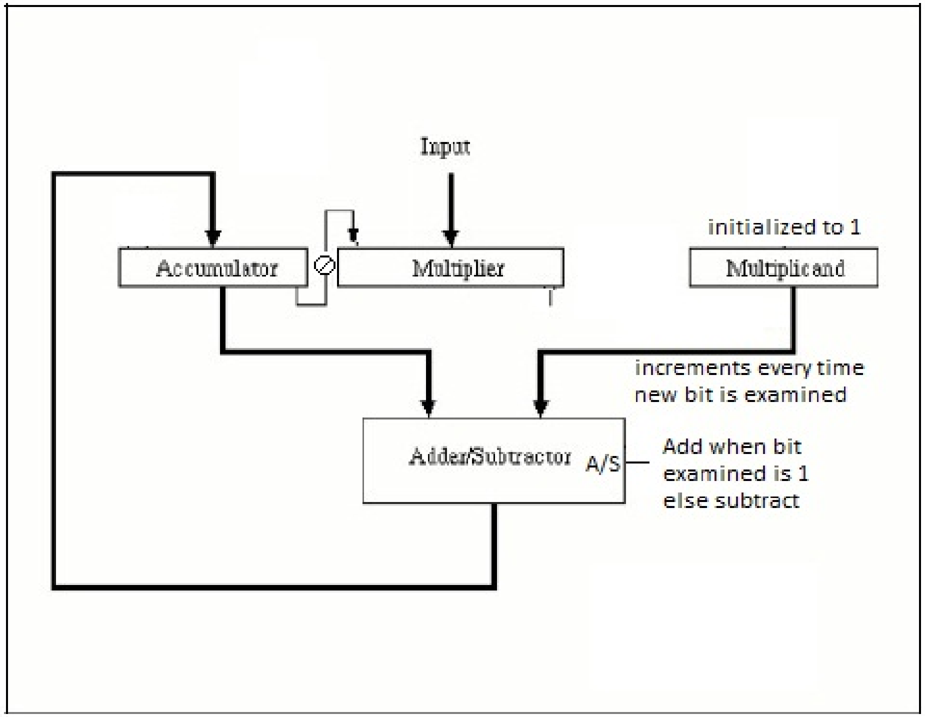
How does the multiplier fit in error detection/correction?

- Robertson's multiplier logic adds the *multiplicand* to partial sum as it examines each *multiplier* bit from right to left.



- What if *multiplicand* is set to 1?
- What if shifting within *Accumulator* and to *multiplier are disabled?*
  - The logic will compute the difference in **sum of number of ones and zeros** in the *Input.*

How does the multiplier fit in error detection/correction?



- What if *multiplicand* is set to 1 as before but now it increments every time a new bit is examined?
  - The logic will compute the difference in **sum of indices of ones and zeros** in the *Input.*

Calculation of Invariants – Booth's Algorithm

- We insert one 0 on the right hand side of the code to examine the LSB.

Indices: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 | **0**

Input: 0 0 1 1 1 1 0 0 0 1 1 0 1 0 0 | **0**

- Examining 2 bits at a time and following the table on right side, the sum would be

-3+4-5+7-10+14 = 7

- This is the **sum of number of ones** in the input!
- Does this work on a number like this ?

Indices: 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 | **0**

Input: 1 1 0 0 0 0 1 1 1 0 0 1 1 0 0 | **0**

-3+5-7+10-14 = -9?

Invariant # 1 Total number of 1s	
00	No-op
01	Add the index of left 0
10	Subtract the index of left 1
11	No-op

Calculation of Invariants – Booth's Algorithm

- So we need another 0 on the left hand side of the code to examine the MSB and close the (01….10) pairs

Indices:	**16**	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	| **0**
Input:	**0**	1	1	0	0	0	0	1	1	1	0	0	1	1	0	0	| **0**

The sum now is *-3+5-7+10-14+16* = *7* (number of ones)

Calculation of Invariants – Booth's Algorithm

- Consider a number,

  Indices: **16** 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 | **0**

  Input: **0** 0 0 1 1 1 1 0 0 0 1 1 0 1 0 0 | **0**

- Examining 2 bits at a time and following the table on right side (different than before), the sum would be

  -1-2+6-8-9+11+12+13-15-16=-9

- This is the difference in **sum of indices of ones and zeros**. Is there something else to it?

Invariant # 2 Sum of indices	
00	Subtract index of left 0
01	No-op
10	No-op
11	Add index of left 1

Calculation of Invariants – Booth's Algorithm

Example:

Indices:	**16**	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	|	**0**
Codeword -1 :	**0**	1	1	0	0	0	1	0	1	1	0	1	0	0	0	1	|	**0**

Invariant #1 Result: -1+2-5+6-7+9-10+11-14+16 = 7 (Invariant 1)
Invariant #2 Result: -3-4+8-12-13+15 = -9 ([Invariant 2]+[Invariant 1]-16)

Calculation of Embedded Value

- Each code word has a unique value embedded in it.
  - Observe least significant 7 bits.
  - Add the indices where 1 appears and subtract the indices where a 0 appears.
  - Divide by 2. The result is the value embedded for that code.

Examples:

Indices:	15 14 13 12 11 10 9 8 **7 6 5 4 3 2 1**
Codeword -2:	1 1 0 0 1 0 0 1 **1 0 0 1 0 0 1**
Indices for 1s in bits 1 to 8:	1+4+7 = 12 (Sum1)
Indices for 0s in bits 1 to 8:	2+3+5+6 = 16 (Sum2)
Sum1-Sum2	-4
Divided by 2 (right shift by 1 bit):	-4/ 2 = -2 (embedded value)

How to compute with Booth's-like algorithm?

Calculation of Embedded Value

- Calculation with Booth's-like algorithm
  - Observe least significant 7 bits.
  - Append dummy bit of 0 to bit 8 and bit 0.
  - Perform 8 steps of invariant 2 algorithm.
  - Add 5 (because there are 3 1's in least significant 7-bits, so you add 8-3 = 5 bits)
  - Divide by 2. The result is the value embedded for that code.

Examples:

Indices: 15 14 13 12 11 10 9 8 **7 6 5 4 3 2 1**
Codeword -2: 1 1 0 0 1 0 0 1 **1 0 0 1 0 0 1**
Indices **8** 7 6 5 4 3 2 1 | **0**
Least Significant 7 bits: **0** 1 0 0 1 0 0 1 | **0**
Invariant 2 calculation: -3-6 = -9
Add 4 -9 + 5 -4
Divided by 2 (right shift by 1 bit): -4/ 2 = -2 (embedded value)

Circuit Functionality

- Phase 1 (or 2): Checking number of 1’s (or sum of indices)
  - Multiplicand: bit index implemented by a counter
  - Multiplier: the input codeword
  - Accumulator: initialized to 0 in Phase 1 and 9 in Phase 2. Will contain the result obtained by checking the invariant
- Detect all errors up to 3
  - Check the value of the accumulator at the end of each phase
  - Raise a buzzer if any error is detected
- Correcting single bit error
  - Compute the absolute value of the accumulator using an absolute value evaluation logic
  - Divide the absolute value by 2 via smart connection of signal lines
  - Flip the erroneous bit using a decoder and XOR gate logic
- Phase 3:
  - Re-run the Phase 2 algorithm with an accumulator initialized to 5 and a multiplier initialized to *the corrected codeword*

Part 2: Boolean Simplification

- A Boolean Algebra: contains 6 axioms
  - Axiom 1 (Closure): Closed with respect to + and * operators
  - Axiom 2 (Identity): A*1=A; A+0=A
  - Axiom 3 (Commutativity): A*B=B*A; A+B=B+A
  - Axiom 4 (Distributivity) A*(B+C)=AB+AC; A+B*C=(A+B)*(A+C)
  - Axiom 5 (Complement): A+A'=1; A*A'=0
  - Axiom 6 (Cardinality Bound): Contains at least 2 elements, x and y, such that x != y

Some Useful Theorems

- Idempotency: A+A=A; A*A=A
- Identiy Absorption: A+1=1; A*0=0
- Involution: (A')' = A
- Associativity: (A*B)*C=A*(B*C); (A+B)+C=A+(B+C)
- DeMorgan's Law: (A+B)'=A'*B'; (A*B)'=A'+B'

Basic Definitions

- Implicant:
  - Product of one or more terms (e.g. A, AB)
- Two-level-sum-of-product expression:
  - Sum of implicants (e.g. AB + AC)
- Minterm:
  - Smallest decomposition of a minterm for a given number of variables (e.g. for 5 variables, abcde or a'bcde')

Your Two Boolean Theorems

- Absorption: AB + ABC = AB
  - Proof:
    - AB + ABC (given)
    - AB*(1+C) (distributivity)
    - AB*1 (identity absorption)
    - AB (identity)
- Consensus: AB + A'C + BC
  - Proof:
    - AB + A'C + BC (given)
    - AB + A'C + 1*BC (identity)
    - AB + A'C + (A+A')*BC (complement)
    - AB + A'C + ABC + A'BC (distributivity)
    - AB + A'C (absorption)

Representing Implicants

- Implicant encoding:
  - Represent each variable as:
    - 00 = invalid
    - 01 = uncomplemented variable
    - 10 = complemented variable
    - 11 = both
- Examples:
  - ABC' = 01 01 10
  - AC' = 01 11 10
- An implicant with '00' for any variable can be replaced with implicant with all 0's

Using The Encoding

- Absorption: an implicant that is a *subset* of another implicant can be eliminated
  - AB = 01 01 11
  - ABC = 01 01 01
- Subset definition:
  - S2 is a subset of S1 if S1 U S2 = S1
  - S2 is a subset of S1 if S1 ∩ S2 = S2
- Use *bitwise-and* for intersection operator and *bitwise-or* for union operator
  - AB U ABC = 01 01 11 | 01 01 01 = 01 01 11 = AB
  - AB ∩ ABC = 01 01 11 & 01 01 01 = 01 01 01 = ABC
- So, we know that ABC is a subset of AB and can be eliminated

Using The Encoding

- Consensus: an implicant that is decomposable into two implicants that are subsets of two other implicants can be removed
  - AB = 01 01 11
  - A'C = 10 11 01
  - BC = 11 01 01
- Intersect BC and AB to get one potential part of decomposition:
  - BC ∩ AB = 11 01 01 & 01 01 11 = 01 01 01
- Intersect BC and A'C to get another potential part of decomposition:
  - BC ∩ A'C = 11 01 01 & 10 11 01 = 10 01 01
- Verify that these two compositions can be combined
  - Implicants can be combined if they differ by 1 variable
    - e.g. abc + a'bc = (a + a')bc = bc
  - 01 01 01 and 10 01 01 differ by one variable since bitwise-xor gives a pair of 1's in a variable location, while the rest of the variable locations are 0s:
    - 01 01 01 ^ 10 01 01 = 11 00 00
- Upon verification, combine the decompositions and see if you obtain BC:
  - 01 01 01 | 10 01 01 = 11 01 01 = BC

Implementation

- Given two classes:
  - Implicant.java
    - Need to implement the *isSubset* and *isConsensus* methods
  - BooleanExpression.java
    - Need to implement the *doSimplification* and *genVerilog* methods

Implementation

- Implicant.java
  - Modifies the encoding by splitting the two bits into separate longs
    - *long myMSB*
    - *long myLSB*
  - Example: Implicant ABC
    - Encoding is 01 01 01
    - *myMSB* = 0xFFFFFFFFFFFFFFF8
    - *myLSB* = 0xFFFFFFFFFFFFFFFF
  - Organization enables easy checks for
    - Differing by single variable
    - Seeing if an implicant is invalid and can be replaced by all 0's

Implementation

- BooleanExpression.java
  - *doSimplification* method
    - Performs one nested iteration where all subsets are removed
    - Performs another nested iteration that greedily removes consensus elements, traversing from left to right
  - *genVerilog* method
    - Outputs valid verilog syntax
    - Use this to test your simplification for correctness