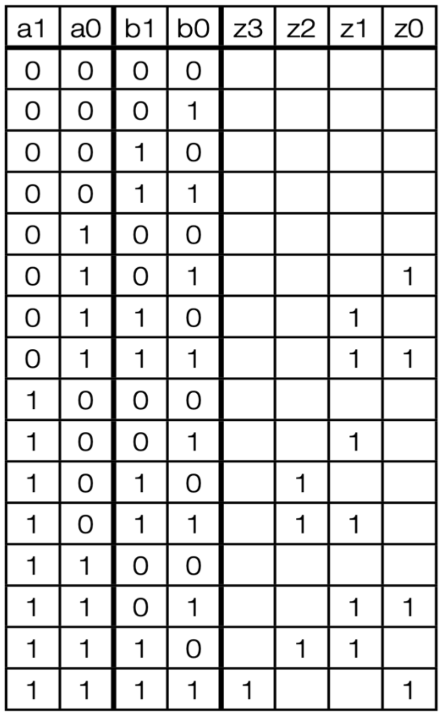
Logic Synthesis & Verification Programming Assignment 1

R12943096 林奕辰

Implementation Details in BLIF

In this section, we are asked to write a 2-bit unsigned multiplier in BLIF. Using the truth table for a 2-bit multiplier, the BLIF file can be written by assigning each output the entries in the truth table.



a1	a0	b1	b0	z3	z2	z1	z0
0	0	0	0				
0	0	0	1				
0	0	1	0				
0	0	1	1				
0	1	0	0				
0	1	0	1				1
0	1	1	0			1	
0	1	1	1			1	1
1	0	0	0				
1	0	0	1			1	
1	0	1	0		1		
1	0	1	1		1	1	
1	1	0	0				
1	1	0	1			1	1
1	1	1	0		1	1	
1	1	1	1	1			1

Figure 1: Truth table of 2-bit multiplier (Source: http://www.cs.columbia.edu/~martha/courses/3827/sp11/slides/2bit_multiplier_soln.pdf)

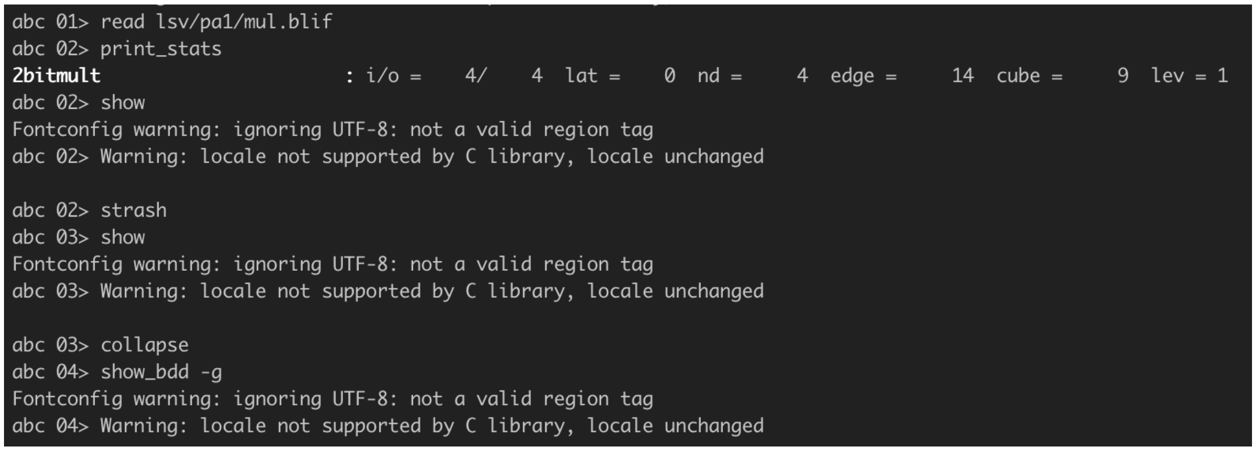
The .blif code is shown as below. Each primary output is expressed as a SOP of the primary inputs, corresponding to each entry of the truth table.

```
# mul.blif
.model 2bitmult
.inputs a1 a0 b1 b0
.outputs y3 y2 y1 y0
.names a0 b0 y0
11 1
.names a1 a0 b1 b0 y1
```

```
011- 1
10-1 1
1101 1
1110 1
.names a1 a0 b1 b0 y2
1010 1
1011 1
1110 1
.names a1 a0 b1 b0 y3
1111 1
.end
```

Using ABC (Section 2)

Next, we use ABC to visualize our circuit in various forms.



```
abc 01> read lsv/pa1/mul.blif
abc 02> print_stats
2bitmult                      : i/o =    4/    4  lat =    0  nd =     4  edge =     14  cube =     9  lev = 1
abc 02> show
Fontconfig warning: ignoring UTF-8: not a valid region tag
abc 02> Warning: locale not supported by C library, locale unchanged

abc 02> strash
abc 03> show
Fontconfig warning: ignoring UTF-8: not a valid region tag
abc 03> Warning: locale not supported by C library, locale unchanged

abc 03> collapse
abc 04> show_bdd -g
Fontconfig warning: ignoring UTF-8: not a valid region tag
abc 04> Warning: locale not supported by C library, locale unchanged
```

FIGURE 2: LIST OF COMMANDS RUN IN ABC

In Fig. 2, after the command print_stats, we can see that the structure has 4 primary inputs, 4 primary outputs, 8 internal nodes and 3 levels, which can be confirmed after the show command in Fig. 3. In addition, there are 10 cubes that comprise this logic network.

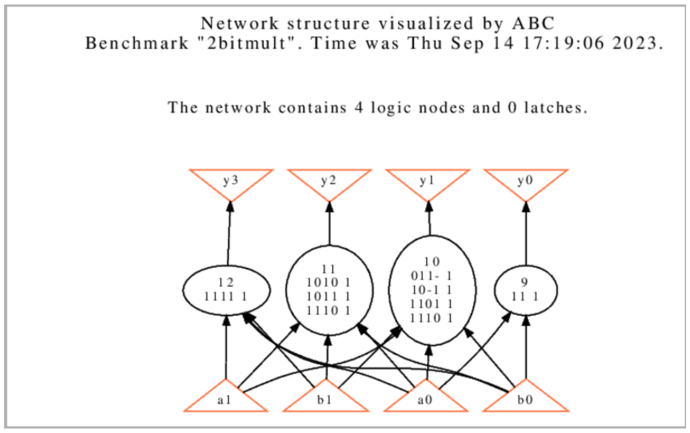

Network structure visualized by ABC
Benchmark "2bitmult". Time was Thu Sep 14 17:19:06 2023.
The network contains 4 logic nodes and 0 latches.
y3
y2
y1
y0
12
1111 1
11
1010 1
1011 1
1110 1
10
011- 1
10-1 1
1101 1
1110 1
9
11 1
a1
b1
a0
b0

FIGURE 3: INITIAL NETWORK STRUCTURE

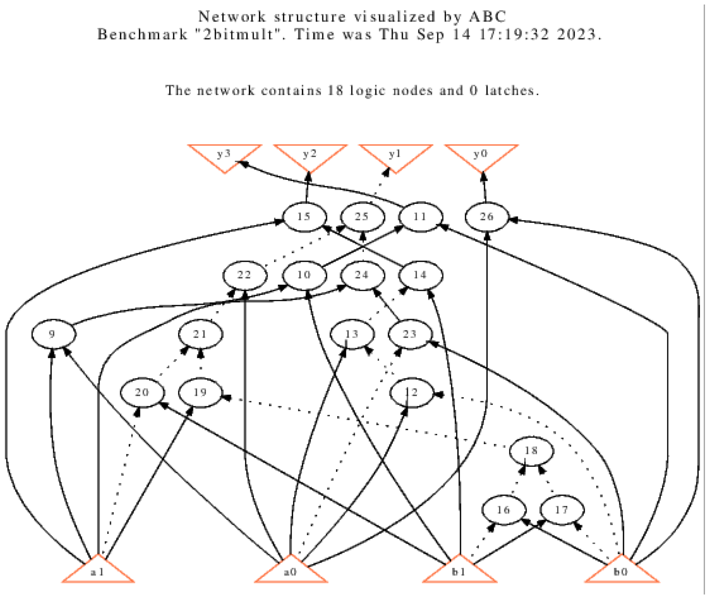

Network structure visualized by ABC
Benchmark "2bitmult". Time was Thu Sep 14 17:19:32 2023.
The network contains 18 logic nodes and 0 latches.
y3
y2
y1
y0
15
25
11
26
22
10
24
14
9
21
13
23
20
19
12
18
16
17
a1
a0
b1
b0

FIGURE 4: AIG FORMAT AFTER STRUCTURAL HASHING

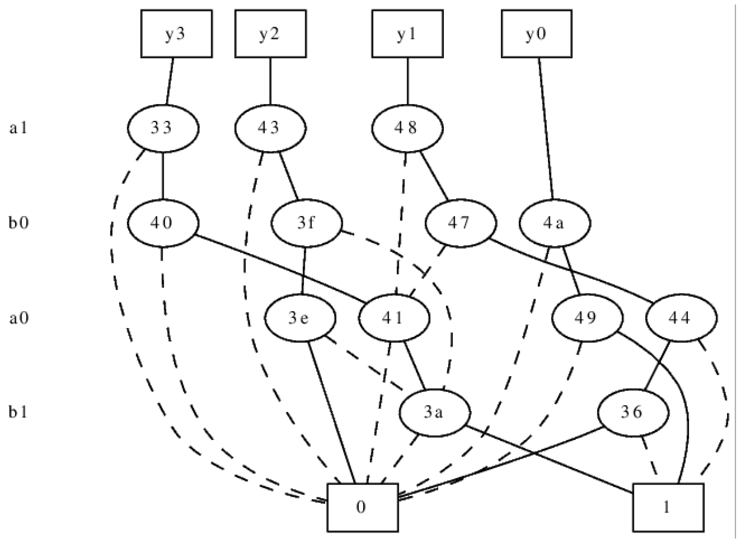


FIGURE 5: BDD STRUCTURE

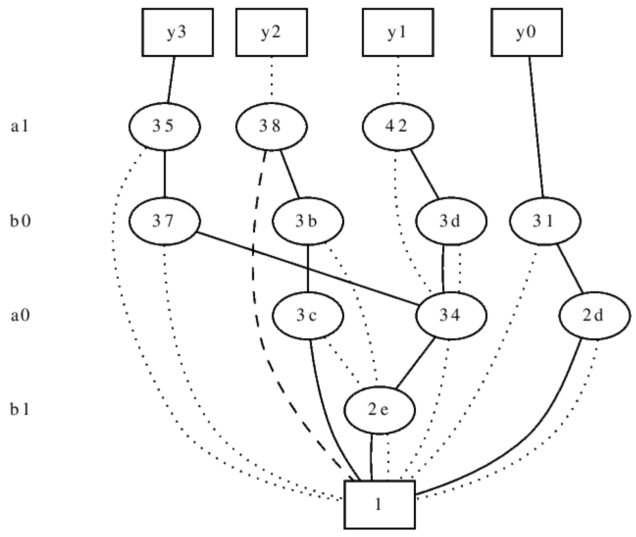


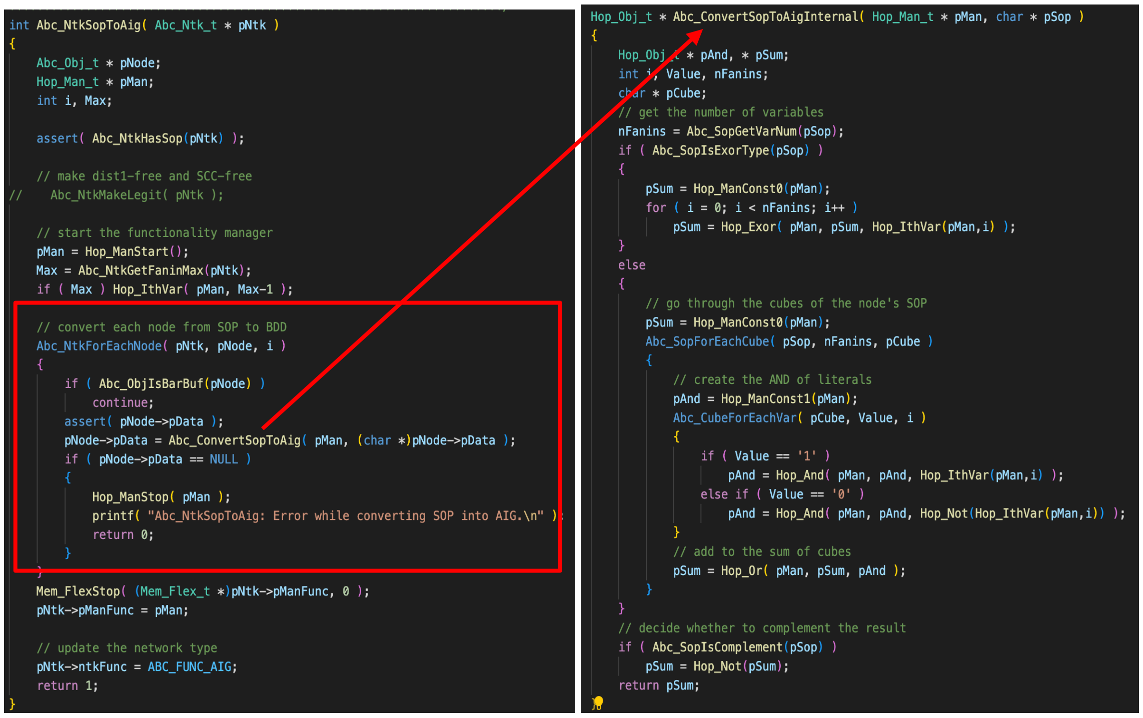
FIGURE 6: BDD STRUCTURE WITH -C OPTION

One simple property of 2-bit multipliers is that the MSB will equal 1 iff the MSB of both inputs are also 1. This can be verified by tracing a path from the constant 1 node to y3.

ABC Boolean Function Representations (Section 3)

`aig` vs. `strash`

In the beginning, when the BLIF file is read, the logic network is stored as a SOP format internally. The aig command iterates through each node and converts the SOP node into an AIG format, as shown in Fig. 7.

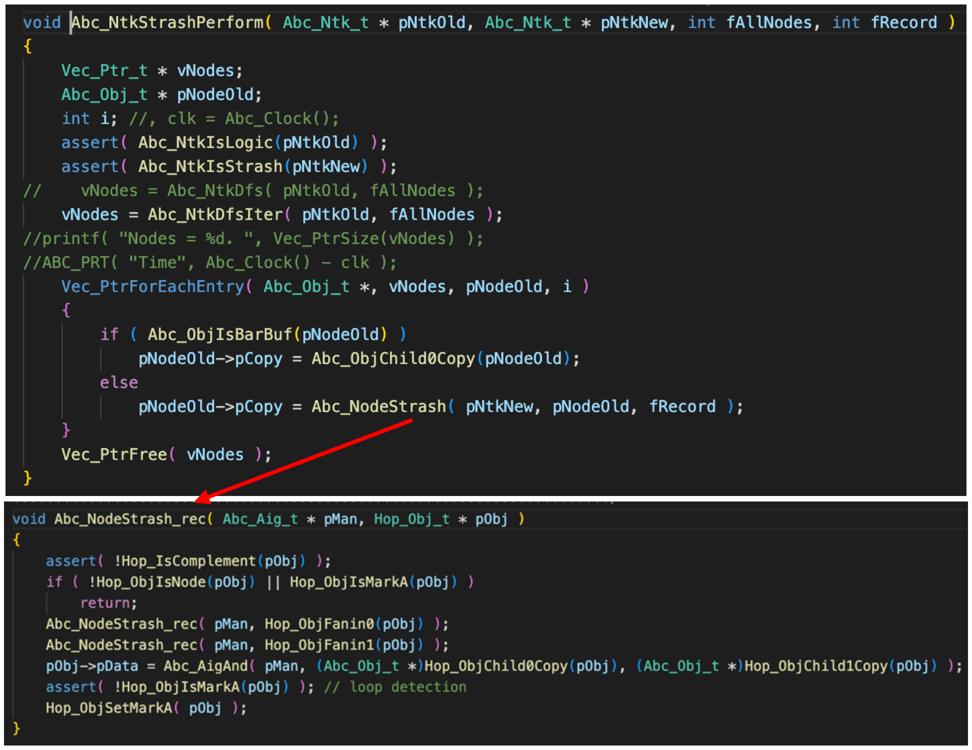


```c
int Abc_NtkSopToAig( Abc_Ntk_t * pNtk )
{
    Abc_Obj_t * pNode;
    Hop_Man_t * pMan;
    int i, Max;

    assert( Abc_NtkHasSop(pNtk) );

    // make dist1-free and SCC-free
//    Abc_NtkMakeLegit( pNtk );

    // start the functionality manager
    pMan = Hop_ManStart();
    Max = Abc_NtkGetFaninMax(pNtk);
    if ( Max ) Hop_IthVar( pMan, Max-1 );

    // convert each node from SOP to BDD
    Abc_NtkForEachNode( pNtk, pNode, i )
    {
        if ( Abc_ObjIsBarBuf(pNode) )
            continue;
        assert( pNode->pData );
        pNode->pData = Abc_ConvertSopToAig( pMan, (char *)pNode->pData );
        if ( pNode->pData == NULL )
        {
            Hop_ManStop( pMan );
            printf( "Abc_NtkSopToAig: Error while converting SOP into AIG.\n" );
            return 0;
        }
    }
    Mem_FlexStop( (Mem_Flex_t *)pNtk->pManFunc, 0 );
    pNtk->pManFunc = pMan;

    // update the network type
    pNtk->ntkFunc = ABC_FUNC_AIG;
    return 1;
}
```

```c
Hop_Obj_t * Abc_ConvertSopToAigInternal( Hop_Man_t * pMan, char * pSop )
{
    Hop_Obj_t * pAnd, * pSum;
    int i, Value, nFanins;
    char * pCube;
    // get the number of variables
    nFanins = Abc_SopGetVarNum(pSop);
    if ( Abc_SopIsExorType(pSop) )
    {
        pSum = Hop_ManConst0(pMan);
        for ( i = 0; i < nFanins; i++ )
            pSum = Hop_Exor( pMan, pSum, Hop_IthVar(pMan,i) );
    }
    else
    {
        // go through the cubes of the node's SOP
        pSum = Hop_ManConst0(pMan);
        Abc_SopForEachCube( pSop, nFanins, pCube )
        {
            // create the AND of literals
            pAnd = Hop_ManConst1(pMan);
            Abc_CubeForEachVar( pCube, Value, i )
            {
                if ( Value == '1' )
                    pAnd = Hop_And( pMan, pAnd, Hop_IthVar(pMan,i) );
                else if ( Value == '0' )
                    pAnd = Hop_And( pMan, pAnd, Hop_Not(Hop_IthVar(pMan,i)) );
            }
            // add to the sum of cubes
            pSum = Hop_Or( pMan, pSum, pAnd );
        }
    }
    // decide whether to complement the result
    if ( Abc_SopIsComplement(pSop) )
        pSum = Hop_Not(pSum);
    return pSum;
```

FIGURE 7: IMPLEMENTATION OF CONVERTING SOPS INTO AIGS IN ABCFUNC.C. NOTE HOW THE FUNCTION DOES NOT CALL ANY RESTRUCTURING OR HASHING PROCEDURES IN THE INNER LOOP.

However, this process only transforms each SOP node into AIG format without any rewriting or structural hashing. The number of nodes stays the same, only the internal data format changes. Technically, the underlying data structure is not a proper AIG at this point.



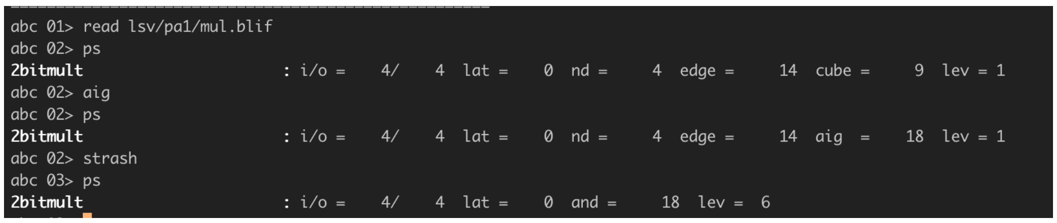
```
void Abc_NtkStrashPerform( Abc_Ntk_t * pNtkOld, Abc_Ntk_t * pNtkNew, int fAllNodes, int fRecord )
{
    Vec_Ptr_t * vNodes;
    Abc_Obj_t * pNodeOld;
    int i; //, clk = Abc_Clock();
    assert( Abc_NtkIsLogic(pNtkOld) );
    assert( Abc_NtkIsStrash(pNtkNew) );
//    vNodes = Abc_NtkDfs( pNtkOld, fAllNodes );
    vNodes = Abc_NtkDfsIter( pNtkOld, fAllNodes );
//printf( "Nodes = %d. ", Vec_PtrSize(vNodes) );
//ABC_PRT( "Time", Abc_Clock() - clk );
    Vec_PtrForEachEntry( Abc_Obj_t *, vNodes, pNodeOld, i )
    {
        if ( Abc_ObjIsBarBuf(pNodeOld) )
            pNodeOld->pCopy = Abc_ObjChild0Copy(pNodeOld);
        else
            pNodeOld->pCopy = Abc_NodeStrash( pNtkNew, pNodeOld, fRecord );
    }
    Vec_PtrFree( vNodes );
}
```

```
void Abc_NodeStrash_rec( Abc_Aig_t * pMan, Hop_Obj_t * pObj )
{
    assert( !Hop_IsComplement(pObj) );
    if ( !Hop_ObjIsNode(pObj) || Hop_ObjIsMarkA(pObj) )
        return;
    Abc_NodeStrash_rec( pMan, Hop_ObjFanin0(pObj) );
    Abc_NodeStrash_rec( pMan, Hop_ObjFanin1(pObj) );
    pObj->pData = Abc_AigAnd( pMan, (Abc_Obj_t *)Hop_ObjChild0Copy(pObj), (Abc_Obj_t *)Hop_ObjChild1Copy(pObj) );
    assert( !Hop_ObjIsMarkA(pObj) ); // loop detection
    Hop_ObjSetMarkA( pObj );
}
```

FIGURE 8: : IMPLEMENTATION OF STRASHING IN ABCSTRASH.C. THE RECURSIVE PROCEDURE STRASHES THE AIG FROM THE BOTTOM UP, STARTING FROM THE PRIMARY INPUTS AND WORKING ITS WAY UPWARDS.

When strash is called, abc iterates through each node and transforms each AIG node into a proper AIG representation, splitting nodes into 2AND gates and adding inverter edges if needed. Then, it performs structural hashing and removes redundant representations of AND gates with the same inputs.

Using the command sequence in Fig. 9, we can see that after calling aig, the number of internal nodes is still the same, and can all be put on the same single level. But after calling strash, the internal representation becomes a proper AIG, and the number of nodes increases, as well as the number of levels to accommodate for the AIG structure.



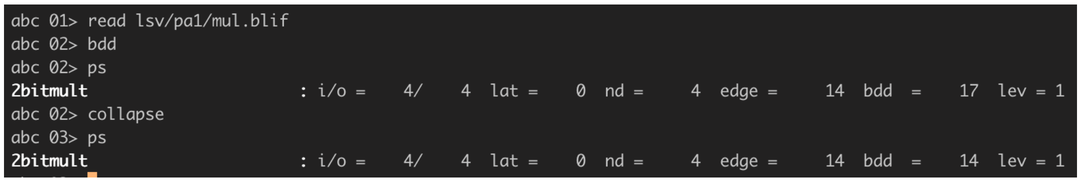
```
abc 01> read lsv/pa1/mul.blif
abc 02> ps
2bitmult                 : i/o =    4/    4  lat =    0  nd =     4  edge =     14  cube =     9  lev = 1
abc 02> aig
abc 02> ps
2bitmult                 : i/o =    4/    4  lat =    0  nd =     4  edge =     14  aig  =    18  lev = 1
abc 02> strash
abc 03> ps
2bitmult                 : i/o =    4/    4  lat =    0  and =     18  lev =  6
```

FIGURE 9: COMPARISON OF PRINT_STATS AFTER CALLING AIG AND THEN STRASH. NOTE THAT WHEN CALLING THE SHOW COMMAND AFTER THE AIG STEP, THE LOGIC STRUCTURE IS STILL SIMILAR TO THAT OF FIG. 3, WITH ONLY 4 NODES REPRESENTING THE WHOLE BOOLEAN FUNCTION.

bdd vs `collapse`

The command bdd functions in a similar manner to the aig command, in that it only converts the internal node representation into a BDD representation without actually changing the number of nodes.

Only the local functions of the nodes are altered. Additionally, the collapse command also does not change the number of nodes, only the BDD local functions of each node.



```
abc 01> read lsv/pa1/mul.blif
abc 02> bdd
abc 02> ps
2bitmult                      : i/o =    4/    4  lat =    0  nd =     4  edge =     14  bdd  =    17  lev = 1
abc 02> collapse
abc 03> ps
2bitmult                      : i/o =    4/    4  lat =    0  nd =     4  edge =     14  bdd  =    14  lev = 1
```

FIGURE 10: COMPARISON OF PRINT_STATS AFTER CALLING BDD AND THEN COLLAPSE. NOTE THAT THE NUMBER OF BDD NODES USED TO REPRESENT THE LOCAL FUNCTIONS HAS BEEN REDUCED FROM 17 TO 14.

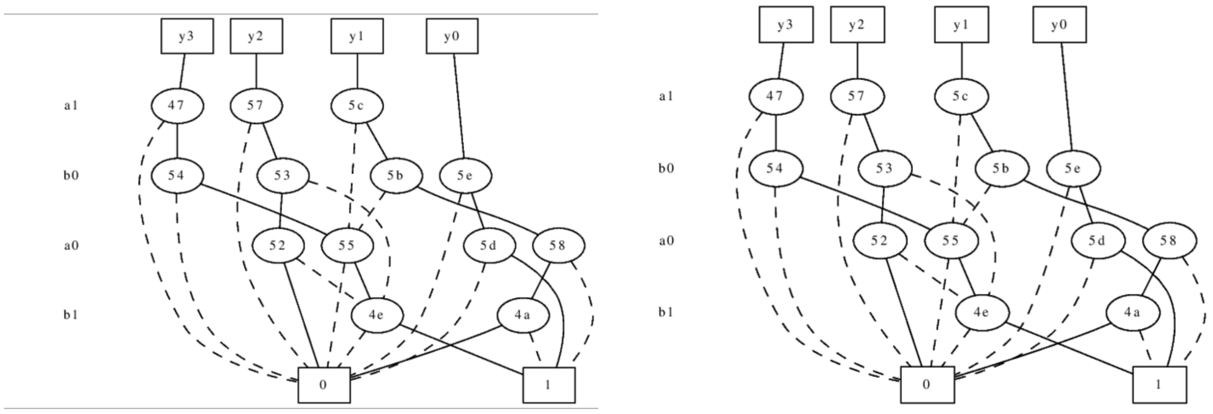


FIGURE 11: COMPARISON OF BDD STRUCTURE (SHOW_BDD -G) AFTER CALLING COMMANDS BDD AND COLLAPSE. THE TWO BDD TREES ARE STILL STRUCTURALLY IDENTICAL.

Strash-ed AIG to SOP

Initially, the logic network is represented as 4 nodes with SOPs as their inner functions. After strashing, the network is expanded, with each node representing a 2AND gate. Therefore, each node function is a 2 variable SOP.

To visualize each node function as a SOP, the function `logic` can be used, which transforms each node of the AIG (i.e. 2AND gate) into the SOP format. The results can be seen in Fig. 12.

In addition, there are other various functions that can achieve similar results. For instance, the function `multi` expands the AIG into a network of multi-input AND gates. Then, after calling the command `sop`, the corresponding SOP for each node can be seen (Fig. 13). This method reduces the number of nodes in the network and makes the representation easier to read. However, one issue in this method is that in each node, the ordering of the variables in the SOP is not preserved. For example, the XOR function represented by nodes 16 to 18 in the AIG should correspond to nodes 17 to 19 in the logic network, but the variable ordering for nodes 18 and 19 is no longer the same.

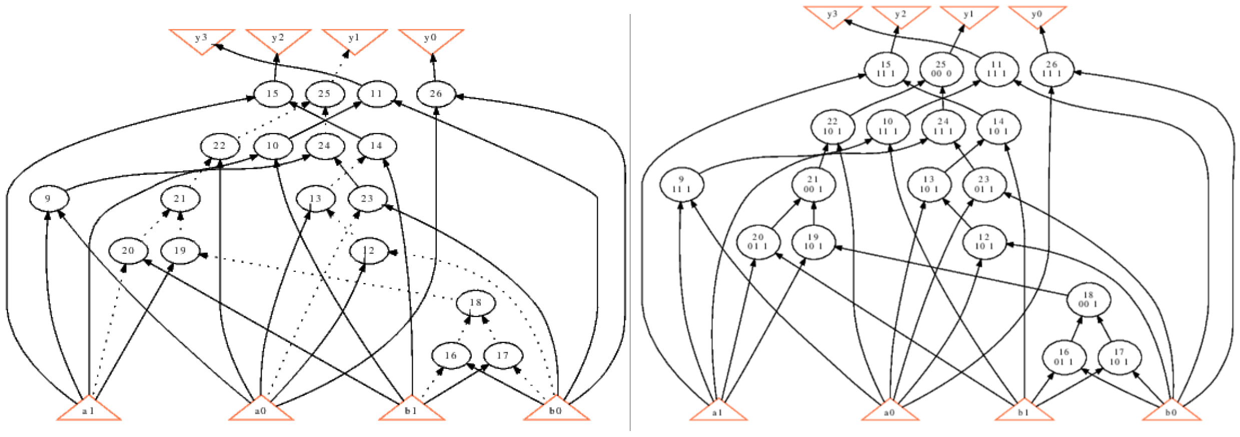


FIGURE 12: RESULTS AFTER CALLING LOGIC COMMAND ON AN AIG. EACH 2AND NODE IN THE AIG (LEFT) CORRESPONDS TO A NODE IN THE LOGIC NETWORK (RIGHT). THE SOP OF EACH NODE IS FUNCTIONALLY IDENTICAL TO THE LOGIC FUNCTION REPRESENTED BY THE AND NODE AND ITS INVERTER EDGES.

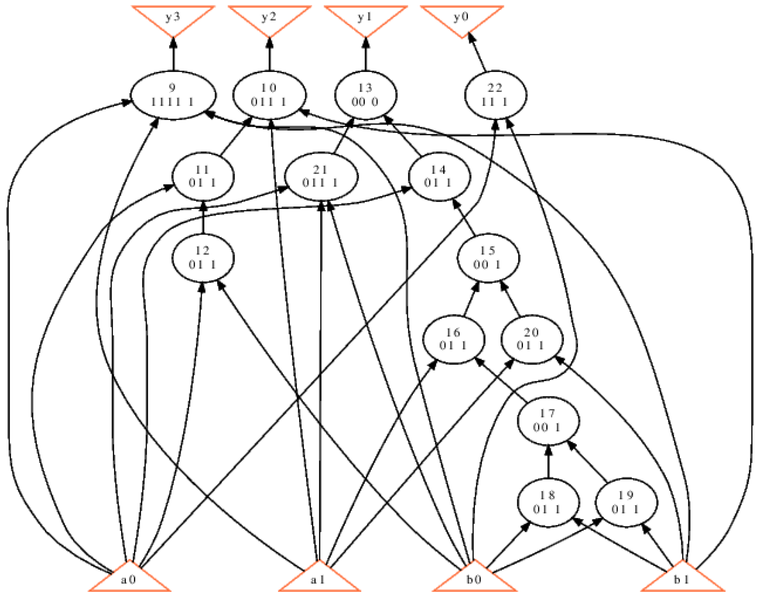


FIGURE 13: RESULTS AFTER CALLING MULTI, THEN SOP COMMANDS ON AN AIG. CERTAIN 2AND NODES ARE MERGED TO MULTI-INPUT AND GATES. CERTAIN VARIABLE ORDERINGS ARE NOT PRESERVED.