UNIVERSITY OF MASSACHUSETTS AMHERST

Development of Equation Parser and Network Visualizer

ECE667 COURSE PROJECT

MUHAMMAD NOMAN ASHRAF

ID: 26262403

Contents

1. INTRODUCTION

It has been shown that any arithmetic circuit can be expressed as a network of half adders, full adders and inverters[1]. A half adder with inputs a and b is represented as

a + b = 2C + S .... eq. 1

Similarly, a full adder with additional cin input can be represented as,

a + b + cin = 2C + S

Logic gates can be expressed by half adders due to the fact that XOR(a,b) is equal to S in eq. 1. Similarly, AND(a,b) is equal to C in eq.1. OR(a,b) can be derived by two half adders.

OR(a,b) = d: a + b =2C + S

C + S =2e + d

However, it can be shown that e = 0 resulting in just C + S =d.

Also, inverters can be expressed as,

INV(a) = b : a + b = 1

Currently, research is being done for verifying bit level arithmetic circuits based on the mathematical model of the circuit derived from the above half adder and full adder expressions. These algebraic equations are used as constraints for solving the equations using linear programming solver. The experimental setup requires deriving adder equations from design netlist and then transforming these equations into a specific format defined by a tool, named ARCHI, currently being used for solving the constraints (through GLPK). The output of ARCHI is a "residual expression" if any consisting of some of the signals used in the design.

1.1 Project Goal

The goal of the project was to develop a tool to automatically perform this transformation from equations to the ARCHI format. Additionally, the tool was required to generate a image file using standard DOT program [2] to visualize the half adders and full adders connected together representing the algebraic form of the mathematical design. It was also desirable to mark the signals that participated in the "residual expression".

2. Experimental Setup

Fig. 1 shows the block diagram of the above discussed experimental setup. In the diagram, the block labeled EQ2ARCHI has been developed. Equations are provided to the tool, which transforms it into archi format for passing to ARCHI program and creates a DOT file for visualizing the design in adder representation which is passed to the DOT program. (DOT program is a tool to create images according to the specification defined in a dot file[2]). The output of the Archi program is fed back to the visualizer

part of the tool, from which the tool extracts the residual expression to mark the inputs and outputs that are present in the residual expression.

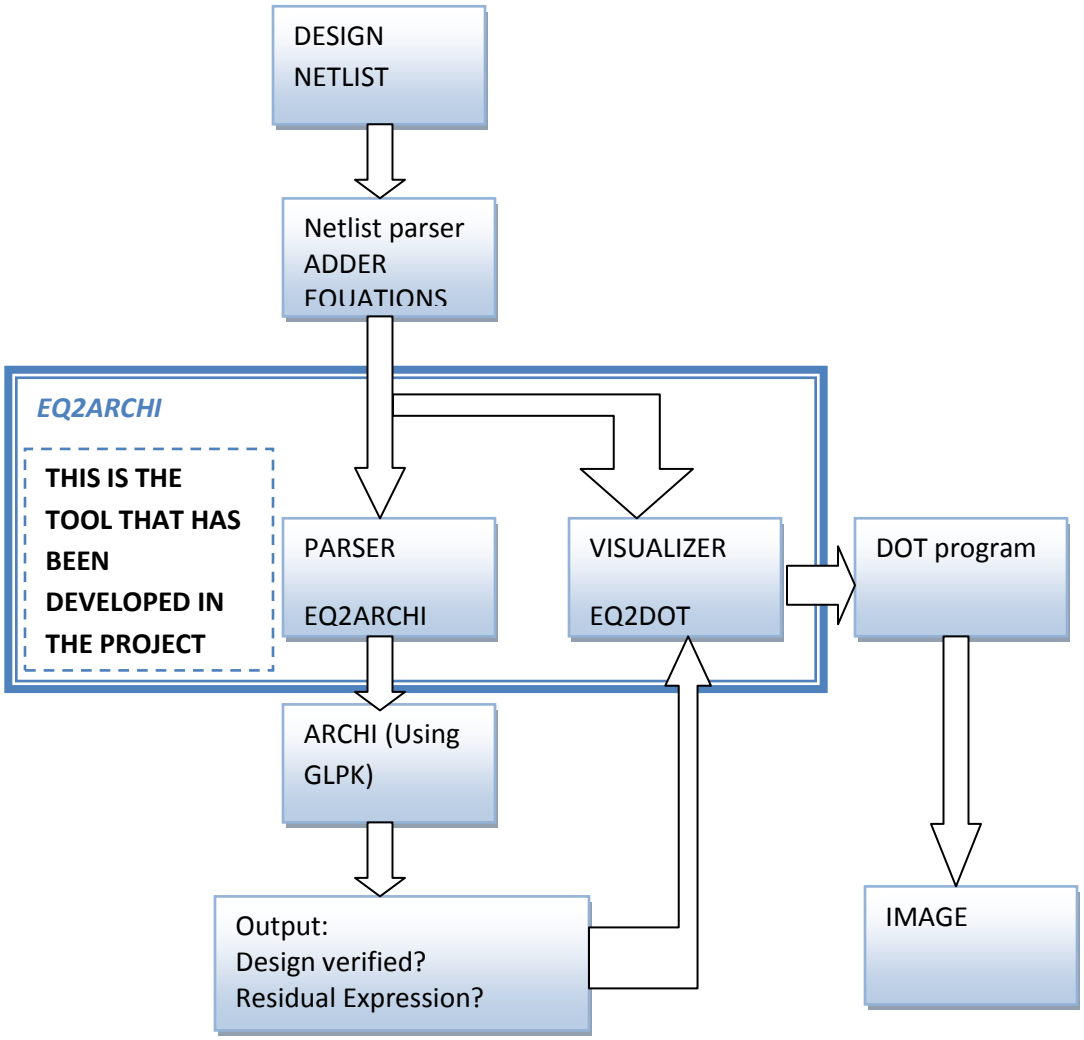


Fig. 1: Block diagram of experimental setup

3. TOOL REQUIREMENTS

3.1 Equation file format (Input requirement)

The verilog parser outputs an equation file containing adder equations. The file contains information about the primary input, output names and primary input, output signatures as comments(#) in the beginning of the file. The rest of the lines contains equations for the logic gates with the following constraints on the format of the equations:

HA: ***a + b = 2*C + S (must have * for multiplication)***
FA: ***a + b + cin = 2*C + S***
Buffer: ***a = b***
Inverter: ***a = (1-b) so that a=input and b=a' is output***
OR gate: ***two HA equations (a+b=2*C+S; S+C=d)***

Note that for XOR and AND gates HA or FA equations are used with dummy variable inserted if either XOR or AND of the inputs are not in the design.

The comments specifying the input, output names and signatures consist of the following format:

#PI-names: a, b, d_2,AIN[0], AIN[1], ... (arrays must be written in expanded form)
#PI-sig: 2*a + 3*b -3*d_2, etc.
#PO-names: d, g, s12, ... (ordered, with LSB on the right)
#PO-sig: 4*d + 2*g + s12, ...

The tool needs to read the equations and transform it into the Archi file format which is described below.

3.2 ARCHI file format (Output requirement)

ARCHI program expects a text file according to following format. The tool needs to produce the information required by the Archi program from the equations and input signature obtained from equation file.

The first line has 6 integers:

n npi nfs npo nps m

where
n: total number of variables (i.e. signals) in the problem
npi: number of primary inputs,
nfs: number of free internal signals
npo: number of primary outputs
nps: number of signals in the (given) partial signature
m: number of constraints in the architecture

The second line contains **indices of primary inputs** in a comma separated list as below:
1,2,3,4,5....
The third line contains **indices of internal signals** in a comma separated list as below:

6,7,8,9,10,11....

The fourth line contains **indices of primary outputs** in a comma seperated list as below:
12,13,14,15....

The fifth line contains **indices of signals** in the given signature in a comma seperated list as below:
1,2,3,4,5,12,13,14,15...

The sixth line contains **coefficients of signals** in the order specified in the aboveline
128,64,...,128,64,...,-128,-64....

The next **m lines** are built as follows:
N var1 coeff1 var2 coeff2 ... varN coeffN = cst

Where N is the number of variables in the constraint
var1 is the index of the first variable, coeff1 is its coefficient
...
= cst is the right hand side.

The remaining n lines contains names of the variables(names are limited to 10 characters)
name1
name2
...
namen

3.3 Output file format (Input requirement)

Although there are a lot of information in the output file, the tool eq2archi only requires the residual expression from the output file. It needs to scan the whole file to find the expression which is formatted in the file as follows by the ARCHI program:

....
Residual Expression: 2a + 3c - d .....
.....

The tool scans this line and finds out the variables a,c,d .... and marks these variables in the generated image.

3.4 DOT file format (Output requirement)

In order to draw half adders and full adders, following format is required in the dot file[3].

FULL ADDER with inputs (I1,I2,I3) outputs (c,d)
add0 [shape=record,pos= "4,18!",label =" { { <I1> | <I2> | <I3> }| {<c> | <d> - sum - }}"];

HALF ADDER with inputs (I1,I2) outputs (c,d)
add1 [shape=record,pos= "4,18!",label =" { { <I1> | <I2> }| {<c> | <d> - sum - }}"];

For connecting output of one adder to input of the other adder, following is the defined format:
add0:c:s -> add1:I3 [label="n49"];
where c output of adder 'add0' is connected to input I3 of adder 'add1'. This edge gets labeled as 'n49'.

To force multiple adders to be drawn at one level (in one row), following format is used:
{rank=same;"sum[0]";"sum[1]";"sum[2]";"sum[3]";"sum[4]";"sum[5]";"sum[6]";"sum[7]";"cout";}

3.5 VISUALIZER REQUIREMENT

The visualizer is required to put all the primary inputs on the first level (row) in the dot diagram. And all the primary outputs are required to be placed at the lowest level(bottom row). Since, there may be some free internal signals (mainly due to the dummy variables added in the equations) which don't participate in the primary outputs, these signals should terminate with a node at the next level and should not be taken down to the primary output level.

Output signals of adders become inputs of other adders(internal signals), an adder was required to be placed in a level as soon as all the inputs become available.

The signals that form residual expression obtained from output file generated by ARCHI, was required to be marked on the network.

4. TOOL DESIGN DETAILS

4.1 Development Language

The tool has been developed in C. Since, the program is meant to be part of a CAD tool which will be performing processing intensive task when verifying large bits arithmetic circuits which involves thousands of signals, parser or a visualizer tool based on scripting languages will be slow. Scripting languages are not compiled and so they lack a number of optimizations that may have been ignored by the developer which compiled applications do possess. Secondly, they are also slow to execute. And the fact is that many scripting languages like PERL [4] are written in C, thus they only provide an easier interface to the developer by adding an additional layer which results in execution overhead.

Lex and yacc [5] or other helper tools based programs are always efficient and suitable when the input files are handwritten and do not have properly defined format. However, for properly defined format and specially when the inputs file are being generated with another software tool (like verilog/netlist parser as in our case) it is better to avoid lexical analyzers because they add additional checks which may have been guaranteed by the format and so may not require checks, effecting performance/execution time on large designs.

And usually lexical analyzers or scripting based tools for performing these tasks are used not due to performance gain but only because they are easier to be developed. C program requires more involved development but contains optimizations and results in better performance.

4.2 Architecture

4.2.1 PARSER:

The parser performs equations to archi transformation. Each signal is assigned a integer value and all the occurrences of that signal is replaced by the integer. Since, searching integer values using signaal names is a time consuming task, a HASH table has been used to accelerate searching by storing signal names and it's integer value in a key-value pair.

Removing the underlying details and requirements, the simplified algorithm for performing this task is as follows:

ReadEQN:

- Read a equation
- extract inputs, outputs, co-efficients for both and constant
- FOR all inputs,outputs
{
- Check for Integer value for the signal name in HASH table
- If integer value found
-do nothing
-else
{
- assign integer value to the signal name
- store integer value for the signal name in Hash Table
- increment integer value by 1
}
}
CALL WriteEQN

WriteEQN:

- write equation as constraint in the file using the integer values instead of signal names

Listing 1: Algorithm for Generating ARCHI input file

4.2.2 VISUALIZER

The requirements of the visualized network discussed above requires a number of structures to be linked, maintained and used properly. Although each equation results in a half adder or a full adder and it appears to be apparently simple to draw adder for each equation read, it turns out to be not as simple for the following reasons:

1. It is required to draw adders in ASAP manner depending upon the inputs availability. If an adder is drawn for each equation without any analysis of inputs availability, the dot program will place it anywhere in the image.

2. Classification of all the adders whose inputs become available at a specific level.

In order to fulfill the requirements, dynamic programming technique has been used. The algorithm first reads the equations, fills a data structure termed "adder" using the information in the equation, this data structure is used to draw the adders at the later stage.

```
struct adder
{
        int NumDep;
        int d[3];
        int level;
        int count;
        int offset;
        int visited;
        int totalInputs;
        char totalOutputs;
        char inp[3][100];
        char strcout[100];
        char sum[100];
        char cdone;
        char sdone;
        char edgeTypeSum;
        char edgeTypeCout;
};
```

Note: The fields in bold are referenced in the text.

While processing the equation, it maintains another data structure which provides the link to the adder which generates the current outputs in a HASH table referenced by these outputs/signals. This information will be used at the later stage for evaluating where the inputs to an adder is coming from.

The "adder" data structure contains a level pointer which is stored with the level information when the level is calculated. The level at which the adder is placed depends upon the level at which adders providing inputs to this adder is placed. The maximum level of these is used for calculating the level of the current adder. Level of primary inputs is always one. Level of adders using primary inputs have the next level. After these data structures are built, the next stage of the program is to calculate levels of all the adders recursively. Inputs of each adder are analyzed from the hash table. A half adder may result in recursive analysis of two more adders and recursively it may result in analysis of other adders on which the adders depend until an adder is reached with only primary inputs whose level is defined to be 1. In the return path all these adders will be assigned levels until the initial adder is reached. Since, Adders that are assigned levels are not re-evaluated so it does not become O(N^2) problem and remains to be just O(N) problem. Listing 2 illustrates this algorithm.

FOREACH Adder
CALL calcLevel

Procedure
calcLevel:
Mark Adder as visited

/ this is interesting - it may not be obvious but it is done to avoid infinite recursion in case of an incorrect design with adder outputs being fed to an adder generating input for this adder. Although arithmetic*

```
circuits will not have this feedback, but an incorrect design may have this and to avoid program hanging
and still managing to show the adders on the image at correct level */


if Adder level defined
{
/* No need to recalculate the level, it was defined during evaluation of one of it's children' adders */
return Adder->Level
}

if (Number of adders this adder is depending on is 0)
calculatedLevel = 1   // level of primary inputs -> actually it is being incremented below to 2
else
{

calculatedLevel = -1   //initialize

FOREACH depending adder [i=0,1,...]
{
        if depending adder level is defined
                candidateLevel [i] = depending adder level
        else
        {
                if depending adder was not visited in the past
                        CALL calcLevel for depending adder
        }

        if calculatedLevel IS LESS THAN candidateLevel [i]
                update calculatedLevel to candidateLevel [i]
}
}

OK previous level calculated - increment it by 1

calculatedLevel = calculatedLevel + 1

if MAX_LEVEL < calculatedLevel
        update MAX_LEVEL = calculatedLevel

return calculatedLevel
```

Listing 2: Algorithm for calculating level of each adder (ASAP - inputs availability)

In order to avoid high requirement of searching when drawing the adders according to levels as all adders with the same level, needs to be searched from the huge "adder" table, another list of data structure is maintained termed as "level" which tells which adders constitute this level. During level calculation, total number of levels are recorded. During the drawing phase, a dynamic allocation is done for the total number of levels for the "level" data structure. Adders table is then iterated and total number of adders on each level is recorded along with the adders constituting that level. Then the "level" table is used to draw adders at the specific level eliminating any need of exhaustive searching of adders.

```
struct _level
{
        int width;
        int width2;
        int * adder;
};

typedef struct _level level;
```

Note: The field in bold is referenced in the text.

For drawing edges connecting outputs of one adders to inputs of the other or primary inputs to inputs of adders etc, the information is obtained from the HASH table which contains information that which adder produces a specific output. The edges are drawn as dotted if the signal is involved in residual expression. This edge type field is present in the "adder" structure and is updated after the ARCHI generates the output file which is then analyzed for residual expression.

Note that algorithms for dot file generation, residual expression marking has not been presented here. The source code of the tool can be observed if further explanation is needed.

4.3 Internal program flow:

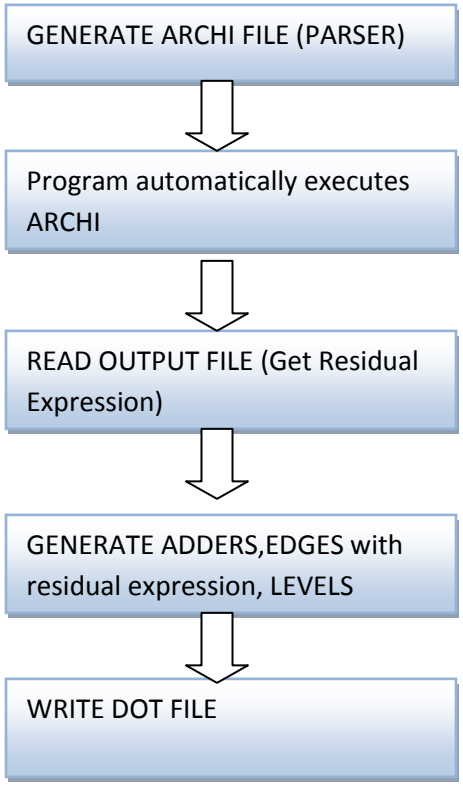


Fig. 2. Internal program flow

5. PROGRAM USAGE

The program requires ARCHI application named "archi-sig" to be present in the same directory where executable "eq2archi" is present. The tool uses config.txt file which must also be present in the same directory.

For help just enter:

./eq2archi

For only generating archi file (PARSER mode)

./eq2archi 1 <equation_file> <archi_file>

i.e.

<equation_file> - the file that contains equations. Note that this file must be present (Since it is input file)

< archi _file> - the output file for ARCHI. Note that if a file of same name already exists, it will be overwritten(updated) otherwise the file will be created.

For only generating dot file (VISUALIZER mode)

./eq2archi 2 <equation_file> <output_dot_file>

i.e.

<equation_file> - the file that contains equations. Note that this file must be present (Since it is input file)

<output_dot_file> - the dot file. Note that if a file of same name already exists, it will be overwritten(updated) otherwise the file will be created.

For combined mode (Both PARSER and VISUALIZER)

./eq2archi 3 <equation_file> <archi_file> <output_dot_file>

i.e.

<equation_file> - the file that contains equations. Note that this file must be present (Since it is input file)

< archi _file> - the output file for ARCHI. Note that if a file of same name already exists, it will be overwritten(updated) otherwise the file will be created.

<output_dot_file> - the dot file. Note that if a file of same name already exists, it will be overwritten(updated) otherwise the file will be created.

5.1 Using ARCHI Input file:

./archi-sig <archi_file> > <output_file>

results in output of ARCHI in <output_file>

5.2 Using DOT file:

dot -Tpng <output_dot_file> -o <output.png>

results in the network diagram in <output.png>

6. Compiling Source Code

Requires glib development package[6] and gcc compilers installed on the linux machine. Usually present by default.

gcc -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -c main.c

gcc -o eq2archi ./main.o -lglib-2.0

7. EXAMPLES:

7.1 8-bit Ripple carry adder

7.1.1Verilog/Netlist file

```
module rca8 ( a, b, cin, sum, cout );
  input [7:0] a;
  input [7:0] b;
  output [7:0] sum;
  input cin;
  output cout;
  wire   n49 , n47 ,
         n45 , n43 ,
         n41 , n39 ,
         n37 ;

  FAX1 U24  ( .A(a[0]), .B(b[0]), .C(cin), .YC(n49 ), .YS(sum[0]) );
  FAX1 U19  ( .A(a[1]), .B(b[1]), .C(n49 ), .YC(n47 ), .YS(sum[1]) );
  FAX1 U17  ( .A(a[2]), .B(b[2]), .C(n47 ),          .YC(n45 ),
.YS(sum[2]) );
  FAX1 U10  ( .A(a[3]), .B(b[3]), .C(n45 ),          .YC(n43 ),
.YS(sum[3]) );
  FAX1 U8  ( .A(a[4]), .B(b[4]), .C(n43 ),          .YC(n41 ),
.YS(sum[4]) );
```

```
  FAX1 U7  ( .A(a[5]), .B(b[5]), .C(n41 ),          .YC(n39 ),
.YS(sum[5]) );
  FAX1 U5  ( .A(a[6]), .B(b[6]), .C(n39 ),          .YC(n37 ),
.YS(sum[6]) );
  FAX1 U4  ( .A(a[7]), .B(b[7]), .C(n37 ),          .YC(cout),
.YS(sum[7]) );
endmodule
```

7.1.2 STEP 1 - NETLIST PARSER + HAND CODING of IO signals and signature

7.1.2.1 contents of rca8.eq

```
#PI-names:
a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],b[0],b[1],b[2],b[3],b[4],b[5],
b[6],b[7], cin
#PO-names: sum[0],sum[1],sum[2],sum[3],sum[4],sum[5],sum[6],sum[7],
cout
#PI-sig:
a[0]+2*a[1]+4*a[2]+8*a[3]+16*a[4]+32*a[5]+64*a[6]+128*a[7]+b[0]+2*b[1]
+4*b[2]+8*b[3]+16*b[4]+32*b[5]+64*b[6]+128*b[7]+ cin
#PO-sig:
sum[0]+2*sum[1]+4*sum[2]+8*sum[3]+16*sum[4]+32*sum[5]+64*sum[6]+128*su
m[7]+256*cout
a[0]+b[0]+cin=2*n49+sum[0]
a[1]+b[1]+n49=2*n47+sum[1]
a[2]+b[2]+n47=2*n45+sum[2]
a[3]+b[3]+n45=2*n43+sum[3]
a[4]+b[4]+n43=2*n41+sum[4]
a[5]+b[5]+n41=2*n39+sum[5]
a[6]+b[6]+n39=2*n37+sum[6]
a[7]+b[7]+n37=2*cout+sum[7]
```

7.1.3 STEP2: Command Line: ./eq2archi 3 rca8.eq rca8.archi rca8.dot

7.1.3.1 Standard output says:

archi file generated
No residual expression
dot file generated

7.1.3.2 contents of rca8.archi

```
33 17 7 9 26 8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
27 28 29 30 31 32 33
18 19 20 21 22 23 24 25 26
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
-1 -2 -4 -8 -16 -32 -64 -128 -1 -2 -4 -8 -16 -32 -64 -128 -1 1 2 4 8
16 32 64 128 256
```

```
5 1 1 9 1 17 1 27 -2 18 -1 = 0
5 2 1 10 1 27 1 28 -2 19 -1 = 0
5 3 1 11 1 28 1 29 -2 20 -1 = 0
5 4 1 12 1 29 1 30 -2 21 -1 = 0
5 5 1 13 1 30 1 31 -2 22 -1 = 0
5 6 1 14 1 31 1 32 -2 23 -1 = 0
5 7 1 15 1 32 1 33 -2 24 -1 = 0
5 8 1 16 1 33 1 26 -2 25 -1 = 0
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
b[0]
b[1]
b[2]
b[3]
b[4]
b[5]
b[6]
b[7]
cin
sum[0]
sum[1]
sum[2]
sum[3]
sum[4]
sum[5]
sum[6]
sum[7]
cout
n49
n47
n45
n43
n41
n39
n37
```

7.1.4 STEP 3: Command Line: dot -Tpng rca8.dot -o rca8.png

Note that there is no residual expression in the output of ARCHI.

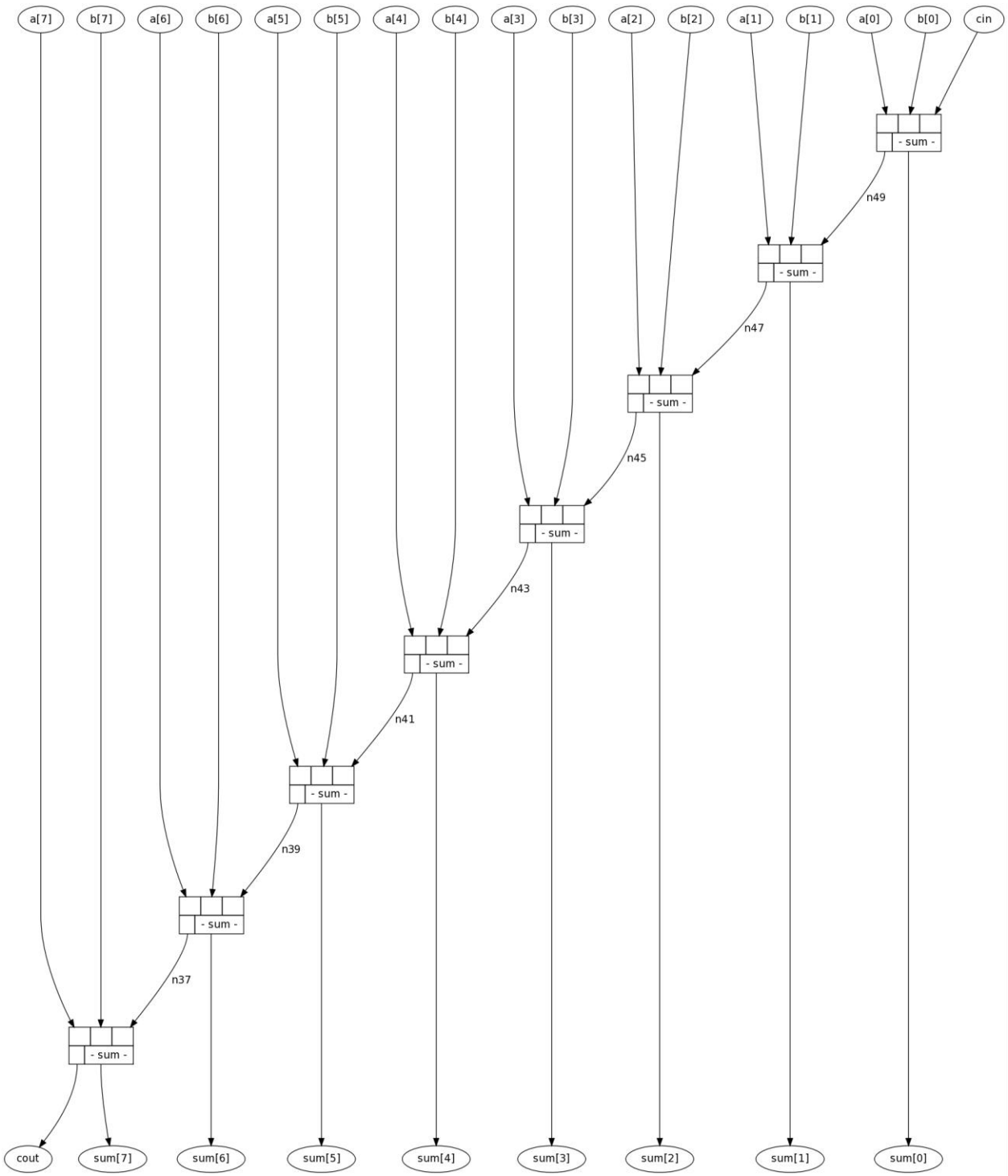


Fig. 3 rca8.png

7.2 4-bit Parallel prefix adder

7.2.1 Contents of equation file:

```
# PI-names: a[0], a[1], a[2], a[3], b[0], b[1], b[2], b[3], cin
# PI-sig: a[0] + b[0] + 2*a[1]+2*b[1] +
4*a[2]+4*b[2]+8*a[3]+8*b[3]+cin
# PO-names: sum[0], sum[1],sum[2],sum[3], c3
# PO-sig:   sum[0]+2* sum[1]+4*sum[2]+8*sum[3]+16*c3

b[0]+a[0]=2*dc7+n7
b[0]+cin+a[0]=2*n6+ds9
b[1]+a[1]=2*n13+n5
b[2]+a[2]=2*n11+n3
b[3]+a[3]=2*n9+n1
cin+n7=2*dc3+sum[0]
dc4+ds0=c3
dc5+ds3=n2
dc6+ds6=n4
n1+n2=2*n8+sum[3]
n1+n7=2*n15+ds12
n10+n11=2*dc5+ds3
n12+n13=2*dc6+ds6
n3+n15=2*n14+ds11
n3+n4=2*n10+sum[2]
n5+n14=2*P3+ds10
n5+n6=2*n12+sum[1]
n8+n9=2*dc4+ds0
```

7.2.2 Command Line: ./eq2archi 3 pp.eq pp.archi pp.dot

7.2.2.1. Standard output says:

archi file generated
residual_expression -2*dc7+2*n6-4*n13-8*n11-16*n9-2*dc3+16*dc4+16*ds0+8*n2+4*n4-16*n8-8*n10-4*n12
dot file generated

7.2.2.2 Contents of pp.archi

```
42 9 28 5 14 18
1 2 3 4 5 6 7 8 9
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
38 39 40 41 42
10 11 12 13 14
1 5 2 6 3 7 4 8 9 10 11 12 13 14
-1 -1 -2 -2 -4 -4 -8 -8 -1 1 2 4 8 16
4 5 1 1 1 15 -2 16 -1 = 0
5 5 1 9 1 1 1 17 -2 18 -1 = 0
4 6 1 2 1 19 -2 20 -1 = 0
```

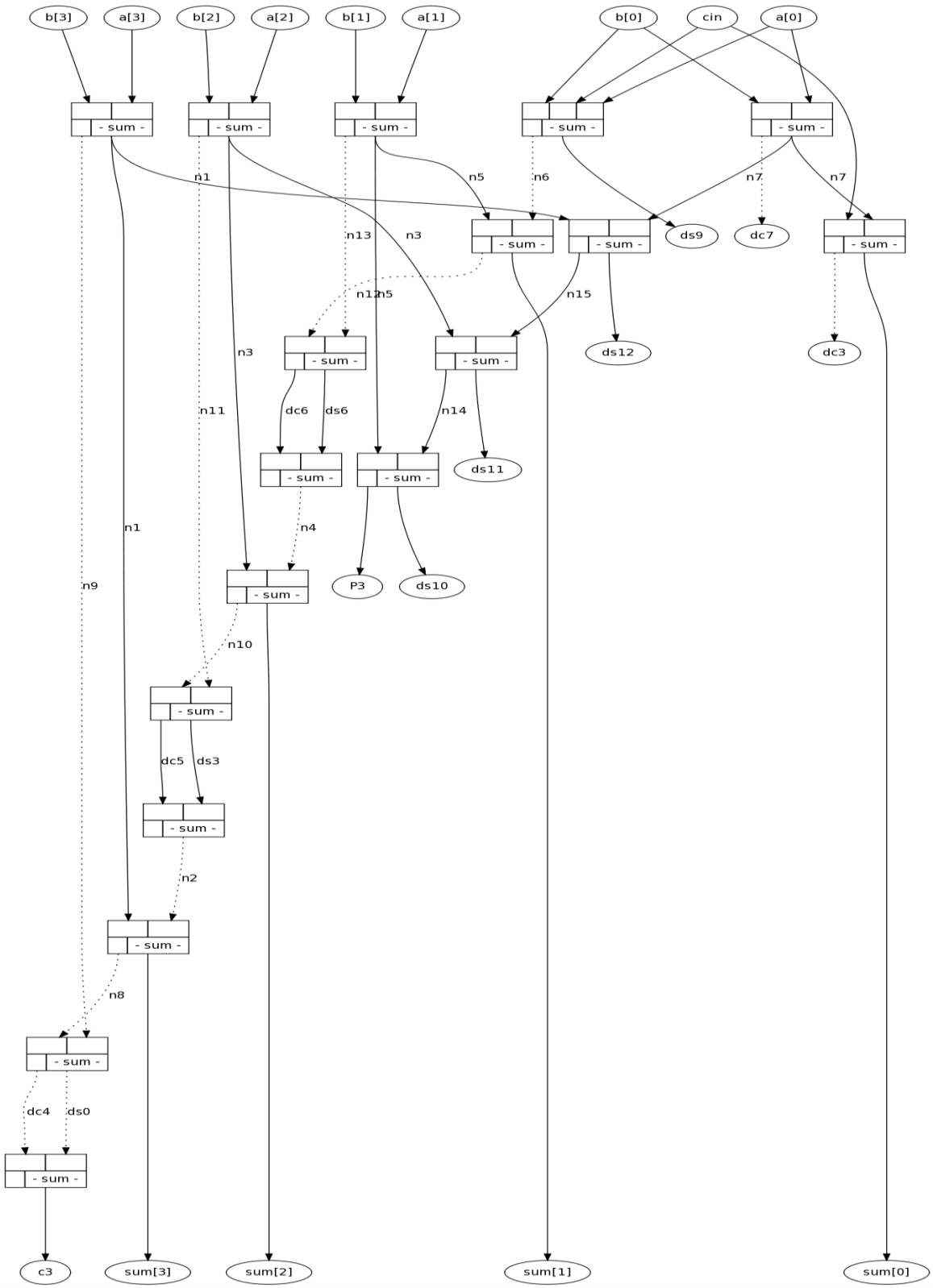
```
4 7 1 3 1 21 -2 22 -1 = 0
4 8 1 4 1 23 -2 24 -1 = 0
4 9 1 16 1 25 -2 10 -1 = 0
3 26 1 27 1 14 -1 = 0
3 28 1 29 1 30 -1 = 0
3 31 1 32 1 33 -1 = 0
4 24 1 30 1 34 -2 13 -1 = 0
4 24 1 16 1 35 -2 36 -1 = 0
4 37 1 21 1 28 -2 29 -1 = 0
4 38 1 19 1 31 -2 32 -1 = 0
4 22 1 35 1 39 -2 40 -1 = 0
4 22 1 33 1 37 -2 12 -1 = 0
4 20 1 39 1 41 -2 42 -1 = 0
4 20 1 17 1 38 -2 11 -1 = 0
4 34 1 23 1 26 -2 27 -1 = 0
a[0]
a[1]
a[2]
a[3]
b[0]
b[1]
b[2]
b[3]
cin
sum[0]
sum[1]
sum[2]
sum[3]
c3
dc7
n7
n6
ds9
n13
n5
n11
n3
n9
n1
dc3
dc4
ds0
dc5
ds3
n2
dc6
ds6
n4
n8
n15
ds12
n10
```

```
n12
n14
ds11
P3
ds10
```

7.2.3. Command Line: dot -Tpng pp.dot -o pp.png

residual_expression -2*dc7+2*n6-4*n13-8*n11-16*n9-2*dc3+16*dc4+16*ds0+8*n2+4*n4-16*n8-8*n10-4*n12

Note that the signals in the residual expression are marked using dotted edges in the network diagram.


b[3]
a[3]
b[2]
a[2]
b[1]
a[1]
b[0]
cin
a[0]
- sum -
n1
n5
n6
n7
n7
n13
n3
ds9
dc7
n12
n5
n15
n3
ds12
dc3
n11
dc6
ds6
n14
ds11
n1
n4
n9
P3
ds10
n10
dc5
ds3
n2
n8
dc4
ds0
c3
sum[3]
sum[2]
sum[1]
sum[0]

8. OPEN PROBLEM

The input and output signature (example shown below) in the equation file which is generated by the Netlist parser is not currently automated and user is required to enter it in the file.

```
#PI-sig:
a[0]+2*a[1]+4*a[2]+8*a[3]+16*a[4]+32*a[5]+64*a[6]+128*a[7]+b[0]+2*b[1]
+4*b[2]+8*b[3]+16*b[4]+32*b[5]+64*b[6]+128*b[7]+ cin
#PO-sig:
sum[0]+2*sum[1]+4*sum[2]+8*sum[3]+16*sum[4]+32*sum[5]+64*sum[6]+128*su
m[7]+256*cout
```

9. FUTURE WORK

A set of constraints can be applied to all the equations like:

x = 0 or 1 so that every occurrence of x can be replaced by a constant 0 or 1 in the equations

x - y = 0 or 1 so that every occurrence of y is replaced by x in the equations

x + y = 1 so that every occurrence of y is replaced by 1-x in the equations

It will result in lesser constraints that are fed to solver.

10. CONCLUSION

The tool was successfully developed and integrated with the ARCHI program and successfully generated the network diagram according to the requirements using DOT.

11. REFERENCES

[1] Student presentation, "*Verification of Arithmetic Circuits, Algebraic Approach", http://www.ecs.umass.edu/ece/labs/vlsicad/ece667/ece667-presentations.html*

[2] http://www.graphviz.org/

[3] *"Drawing graphs with DOT" , http://www.graphviz.org/Documentation/dotguide.pdf*

[4] http://www.perl.org/

[5] http://dinosaur.compilertools.net/

[6] http://www.gtk.org/