Optimised Implementation of Logic Functions

A python script to implement an optimization technique based on cubical representation

Atishay Jain - 16110024 - atishay.jain@iitgn.ac.in
Debanuj Nayak - 16110048 - debanuj.nayak@iitgn.ac.in
Kunal Verma - 16110084 - kunal.verma@iitgn.ac.in
Rahul Yadav - 16110128 - rahul.yadav@iitgn.ac.in
Shreyas Singh - 16110150 - shreyas.singh@iitgn.ac.in

Abstract

The aim of the project is to implement a Boolean logic minimization algorithm than can take multi bit expressions as inputs and generate a structural verilog code of the minimised expression as an output. The code is written in python and does not require any external dependencies to be run.

Introduction

Karnaugh maps provide a systematic way of manually deriving minimum-cost implementations of simple logic functions, but they become impractical for functions of many variables. That is why they are not used directly as algorithms for CAD tools.

This report describes a python based script that takes multi bit input expressions to create the optimized verilog structural code. The script essentially does a 2 level minimisation for each bit of the output. It uses an algorithm described in the Fundamentals of Digital Logic with Verilog Design by Brown and Vranesic that maps all variables to the vertices of a n-dimensional hypercube. The script uses an adapted minimization technique (that uses cubical representation) described by Willard Quine and Edward McCluskey (the Quine-McCluskey method).

.

The python script mainly consists of three sections

- **Accepting multi bit inputs :** Multi bit inputs are accepted and the addition and multiplication expressions for them are generated.
- **Minimisation Algorithm :** These expressions are fed into the minimisation algorithm to generate minimized expressions for addition and multiplication.
- **Verilog Output File Generation :** These minimized expressions are then converted into a verilog output file which can be used.

Procedure of Implementation

1. Accepting Multi Bit Inputs

The python script accepts inputs as any general expression with user defined variables in form of string, where the size of every variable is specified. The script then asks the user to enter the expression specifying the operation to apply on the multi bit inputs.The script generates expressions for minimisation.

The expression involving the multi bit inputs can be a combination of addition and multiplication. Thus the script contains special functions that computes the respective expressions whenever two such multi bit inputs are added or multiplied.

The adder function mimics the way a ripple carry adder adds two numbers. It repeatedly calls the full_add function which is the script's equivalent of a basic full adder unit.

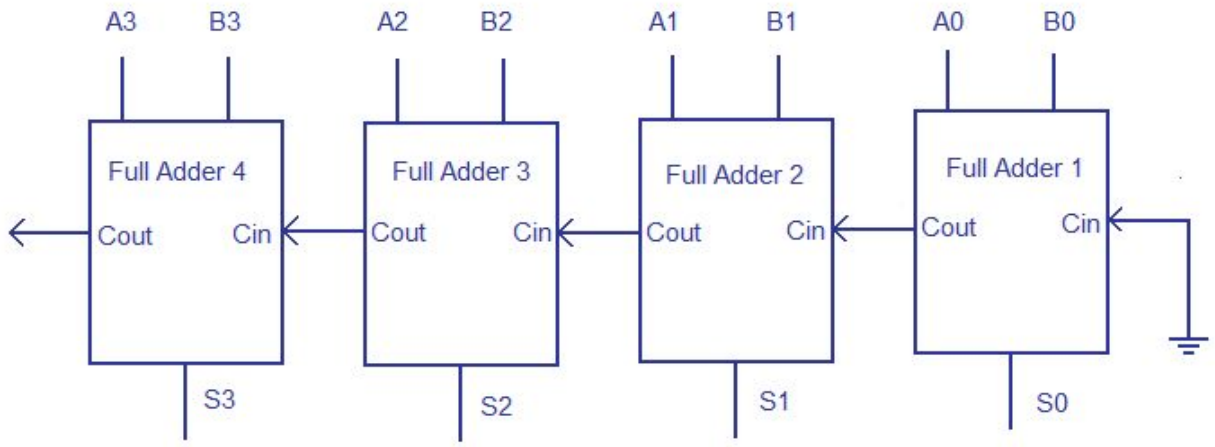


Fig - Ripple carry adder circuit

Here the whole ripple carry adder is the adder function and the individual full adders are the repeated calling of the full_add function.

The multiplier function on the other hands works by calling the adder function multiple times inside it. The multiplier function mimics the basic structure of the multiplier circuit given below. As shown, in each step of the circuit, the ripple carry adder is present.

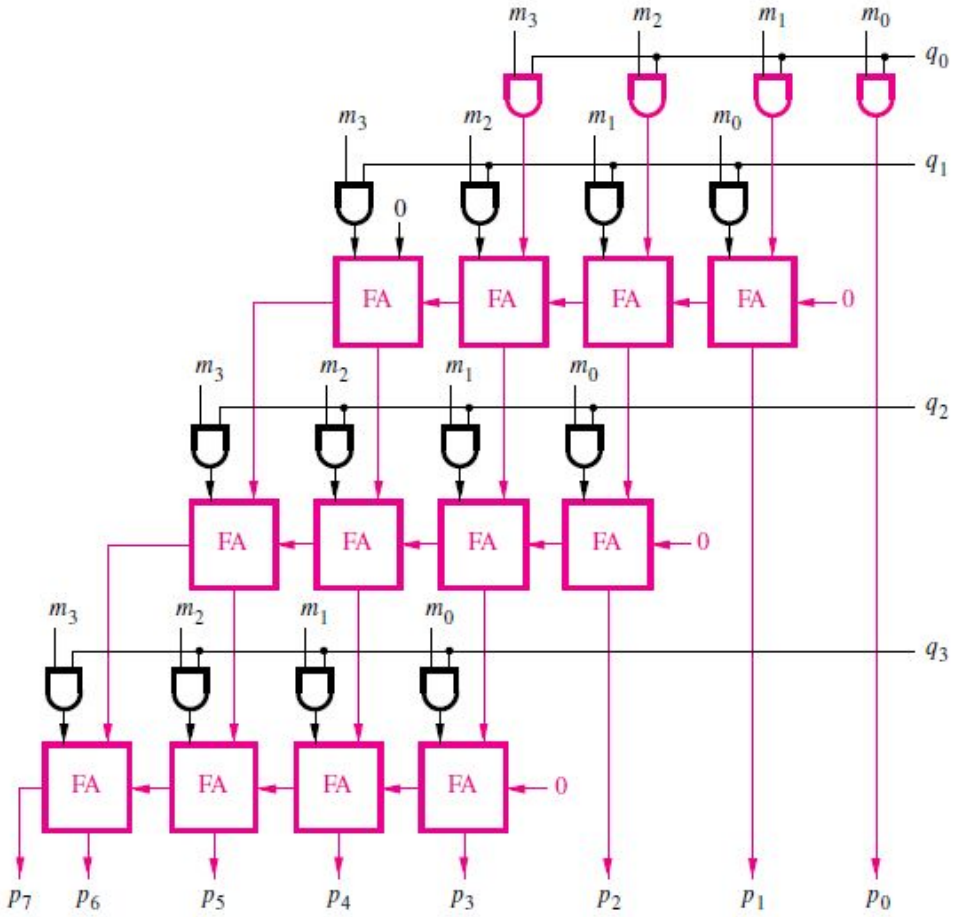


Fig - Multiplier Circuit

Apart from these, the script also contains basic functions mynot , myor , myand , myxor which are used to represent the basic NOT , OR , AND and XOR gates.

2. The Minimisation Algorithm

1. Let $\mathbf{C^0 = ON \cup DC}$ be the initial cover of function f and its don't-care conditions.
2. Find all prime implicants of C^0 using the ∗-operation and let P be this set of prime implicants.
3. Find the essential prime implicants using the #-operation. A prime implicant p^i is essential if $\mathbf{p^i \# (P - p^i) \# DC = \emptyset}$. If the essential prime implicants cover all vertices of the ON-set, then these implicants form the minimum-cost cover.
4. Delete any nonessential p^i that is more expensive (i.e., a smaller cube) than some other prime implicant p^j if $\mathbf{p^i \# DC \# p^j = \emptyset}$.

5. Choose the lowest-cost prime implicants to cover the remaining vertices of the ON-set. Use the branching algorithm on the prime implicants of equal cost and retain the cover with the lowest cost.

Finding the Prime Implicants

The ∗-operation provides a simple way of deriving a new cube by combining two cubes that differ in the value of only one variable. If A = A1A2 … An and B = B1B2 … Bn be two cubes that are implicants of an n-variable function. Thus each coordinate Ai and Bi is specified as having the value 0, 1, or x. There are two distinct steps in the ∗-operation.

First, the ∗-operation is evaluated for each pair Ai and Bi , in coordinates i = 1, 2, … , n, according to the table.

A_i \ B_i	0	1	x
0	0	ø	0
1	ø	1	1
x	0	1	x

$A_i * B_i$

Then based on the results of the previous steps the following set of rules are applied to determine the final result:

1. C = ø if Ai ∗ Bi = ø for more than one i.
2. Otherwise, Ci = Ai ∗ Bi when Ai ∗ Bi = ø, and Ci = x for the coordinate where Ai ∗ Bi = ø.

To find the prime implicants of a function f, we denote its cover as C^k and c^i and c^j be any two cubes in C^k . Then applying the ∗-operation to all pairs of cubes in C^k we will create G^{k+1} which will be the set of newly generated cubes. Hence $G^{k+1} = c^i * c^j$.

Now we will form a new cover of f, $C^{k+1} = C^k \cup G^{k+1}$ − redundant cubes, where the redundant cubes are those cubes which are already included in other cubes. We continue this process until $C^{k+1} = C^k$, then the cubes in the cover are the prime implicants of f.

Note: For an n-variable function, it is necessary to repeat the step at most n times.

Finding Essential Prime Implicants

All essential prime implicants must be included in the minimal cover.To find the essential prime implicants, we define an operation that determines a part of a cube (implicant) that is not covered by another cube. For this we define the #-operation**.** Let A = A1A2 · · · An and B = B1B2 · · · Bn be two cubes (implicants) of an n variable function.The sharp operation

A_i \ B_i	0	1	x
0	ε	ø	ε
1	ø	ε	ε
x	1	0	ε

$A_i \# B_i$

A#B leaves as a result “that part of A that is not covered by B.”

The #-operation is defined as follows:

- C = A#B, such that
- C = A if Ai#Bi = ø for some i.
- C = ø if Ai#Bi = ε for all i.
- Otherwise, C = $\bigcup_i$(A1, A2, … , ~Bi, … , An) , where the union is for all i for which Ai = x and Bi != x.

Let P be the set of all prime implicants of a given function f and p^i denote one prime implicant in the set P and let DC denote the don’t-care vertices for f. Then p^i is an essential prime implicant if and only if

p^i # (P − p^i) # DC = ø

This meaning is applied successively to each prime implicant in P.

Removing the Expensive Non-Essential Prime Implicants

Now that all the essential prime implicants have been found, these will surely be present in the minimal cover. Now from the rest of the non essential prime implicants , the more expensive prime implicants must be removed.

A particular non essential prime implicant p^i is more expensive than some other prime implicant p^j and must be removed from the minimal cover, if

p^i # DC # p^j = ø

Generating the minimal cover

If the remaining prime implicants are of equal cost, we apply a branching algorithm. In this algorithm, we choose one of the remaining prime implicants (in our case we chose the first prime implicant in the list). Now two cases arise:

1. The implicant is part of the final cover
2. The implicant is not part of the final cover

In both these cases we again apply the above algorithm of finding the essential prime implicants from the remaining prime implicants ,removing expensive non-essential prime implicants and applying the branching algorithm again if required. Each case leads to the formation of a cover.

After all the possible covers are obtained, the least expensive cover is chosen to be the final minimised cover.

The following flowchart helps in visualising the process.

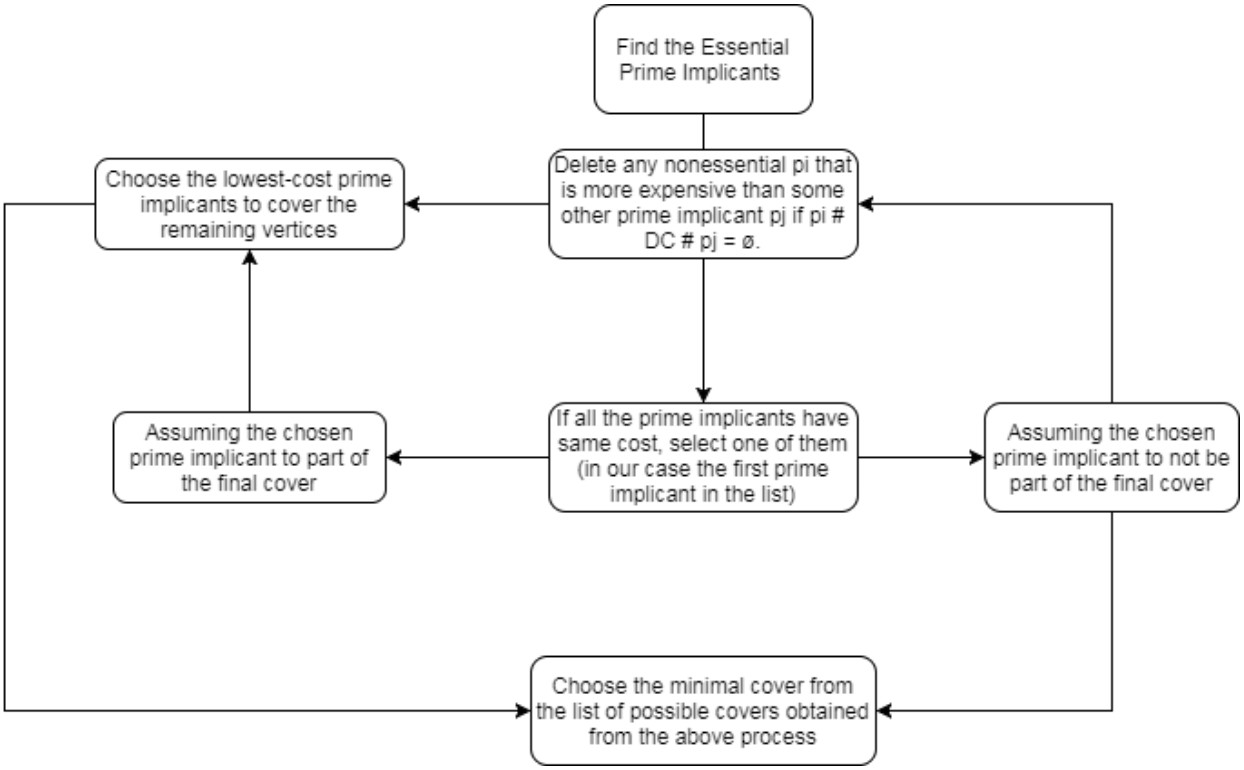


Fig - Flowchart explaining the branching algorithm

3. Output file Generation

The generated output expression from the algorithm is taken as an input in this step.
Using that expression a text file with the main module of the verilog code is generated. This module consists of calls to other modules that are also generated.
The verilog code generated follows a set of rules in taking variable names and other parameters. As we follow structural coding during verilog implementation, this allows us to specify the actual representation.

Result and Discussion

For testing the project, we took the example AB + BC + AC, where A,B,C are 2 bit buses. The expression is read such that AB means A is multiplied with B, and A + B means A is added to B.

To test the example, we used the code generated by our script to create a verilog file and we created a test bench to test the same. For the values inputted into the test bench, we obtained the expected results, thus showing the correctness of our project.
For checking whether the logic was minimised, we compared the number of LUTs used by our code with the number of LUTs used by Xilinx. The design report showed that our code used lesser number of LUTs.
The code we used for comparison was a behavioral code which is expected to be optimised by Xilinx itself.

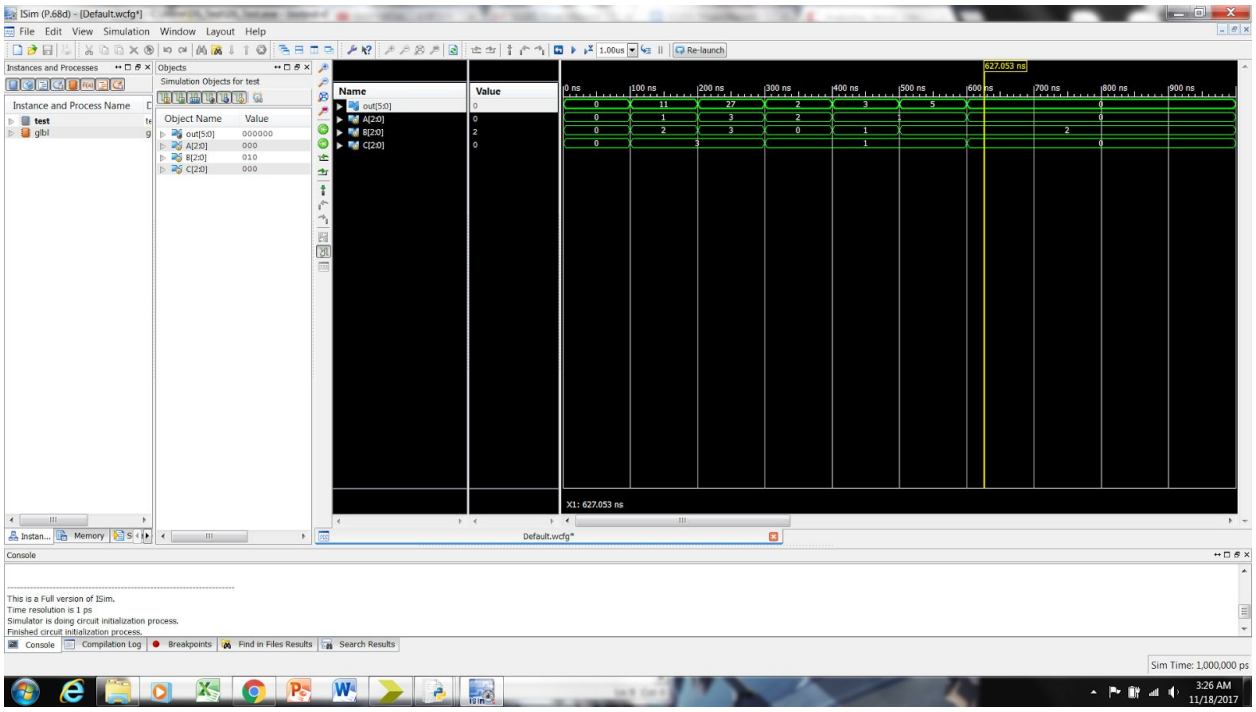


Fig - Test Bench for the given example

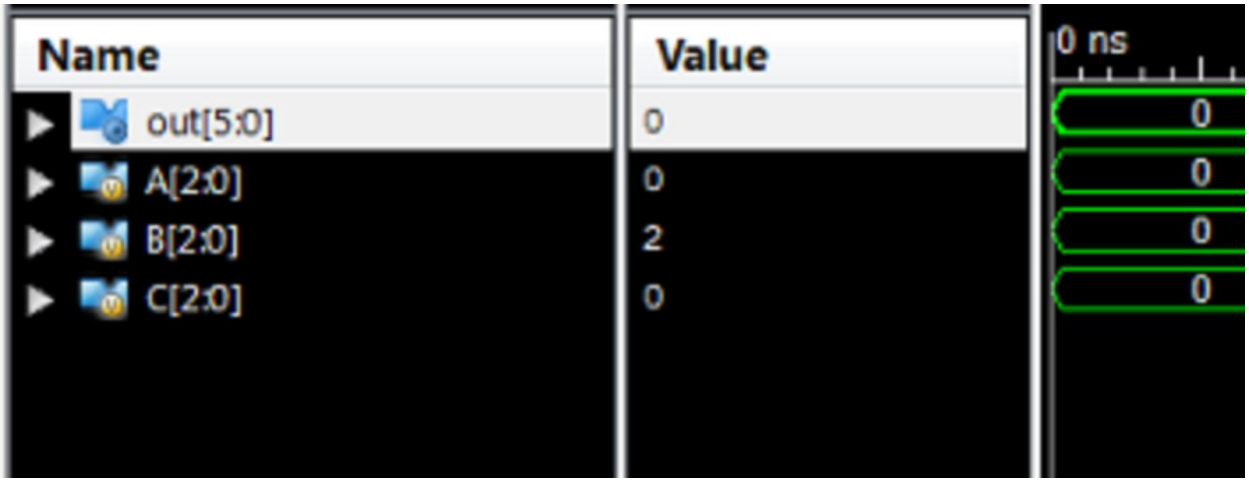


Fig - Zoomed in test bench (1)

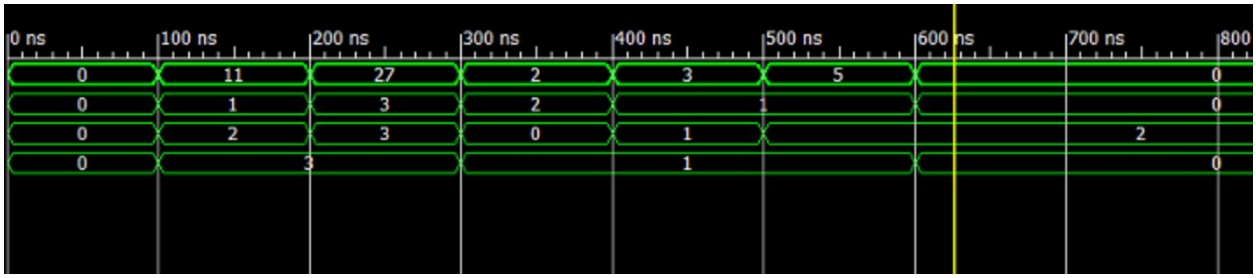


Fig - Zoomed in test bench (2)

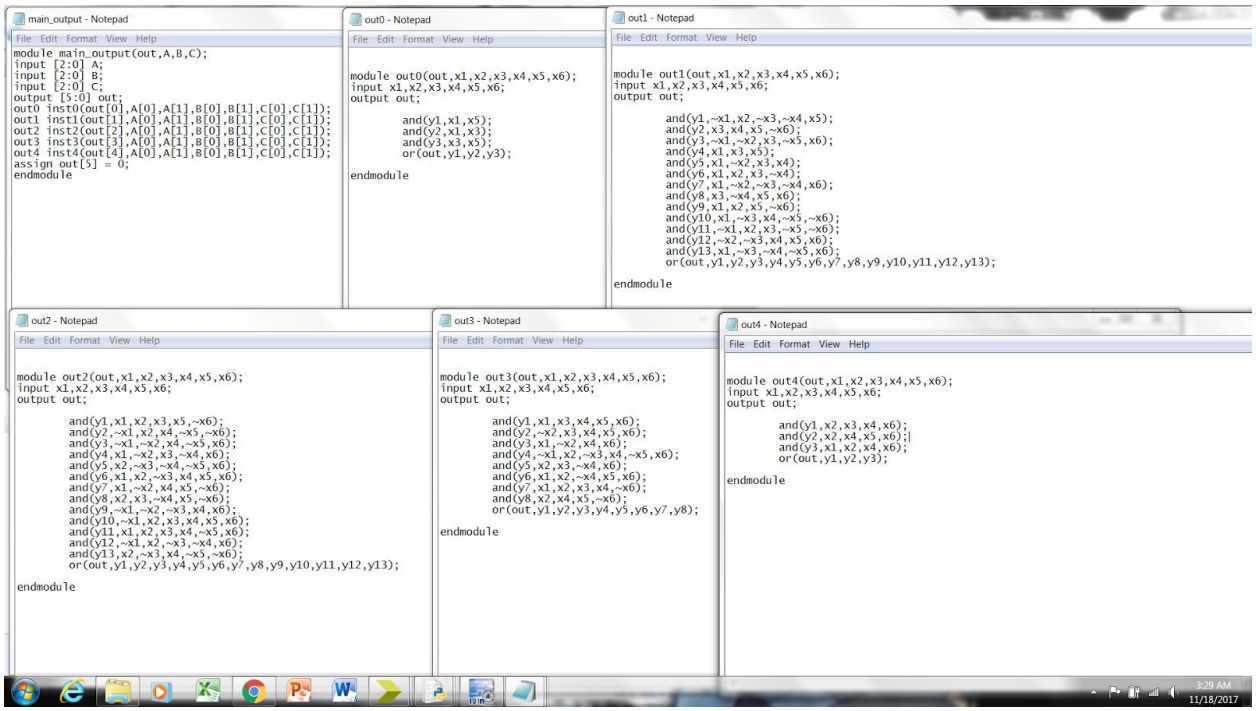


Fig - The generated text files containing the verilog code

Fig - Stage 1.1 of execution of code for example - Expression Generation

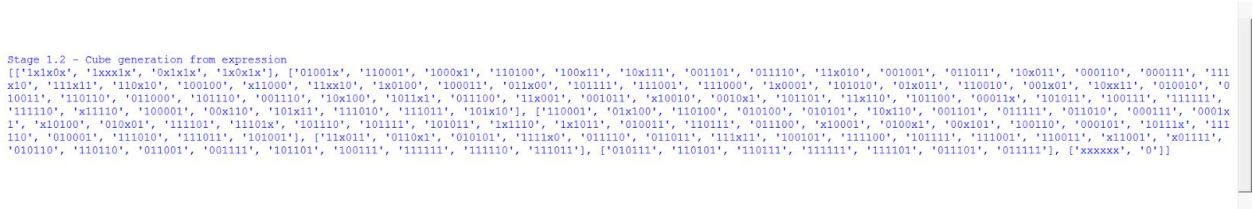


Fig - Stage 1.2 of execution of code for example - Cube Generation

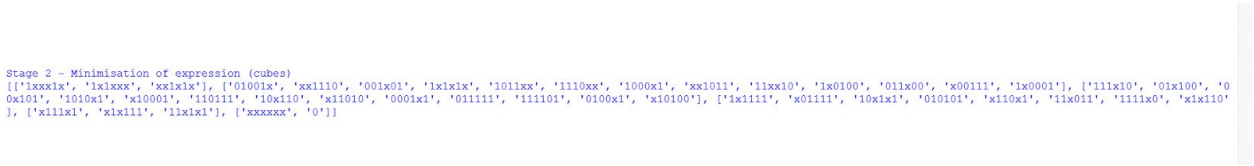
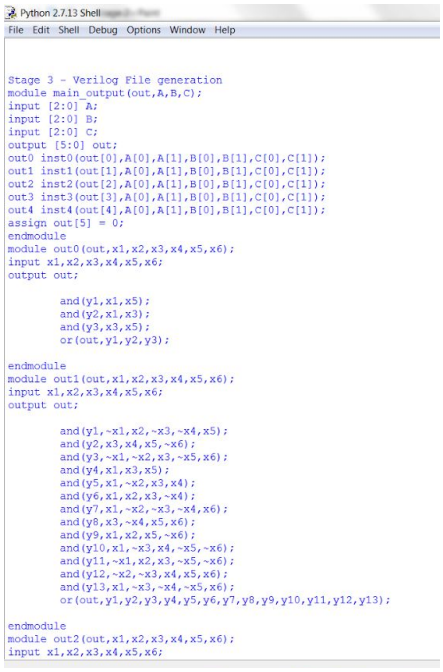


Fig - Stage 2 of execution of code for example - Minimised Expression Generation



```
Python 2.7.13 Shell
File  Edit  Shell  Debug  Options  Window  Help


Stage 3 - Verilog File generation
module main_output(out,A,B,C);
input [2:0] A;
input [2:0] B;
input [2:0] C;
output [5:0] out;
out0 inst0(out[0],A[0],A[1],B[0],B[1],C[0],C[1]);
out1 inst1(out[1],A[0],A[1],B[0],B[1],C[0],C[1]);
out2 inst2(out[2],A[0],A[1],B[0],B[1],C[0],C[1]);
out3 inst3(out[3],A[0],A[1],B[0],B[1],C[0],C[1]);
out4 inst4(out[4],A[0],A[1],B[0],B[1],C[0],C[1]);
assign out[5] = 0;
endmodule
module out0(out,x1,x2,x3,x4,x5,x6);
input x1,x2,x3,x4,x5,x6;
output out;

        and(y1,x1,x5);
        and(y2,x1,x3);
        and(y3,x3,x5);
        or(out,y1,y2,y3);

endmodule
module out1(out,x1,x2,x3,x4,x5,x6);
input x1,x2,x3,x4,x5,x6;
output out;

        and(y1,~x1,x2,~x3,~x4,x5);
        and(y2,x3,x4,x5,~x6);
        and(y3,~x1,~x2,x3,~x5,x6);
        and(y4,x1,x3,x5);
        and(y5,x1,~x2,x3,x4);
        and(y6,x1,x2,x3,~x4);
        and(y7,x1,~x2,~x3,~x4,x6);
        and(y8,x3,~x4,x5,x6);
        and(y9,x1,x2,x5,~x6);
        and(y10,x1,~x3,x4,~x5,~x6);
        and(y11,~x1,x2,x3,~x5,~x6);
        and(y12,~x2,~x3,x4,x5,x6);
        and(y13,x1,~x3,~x4,~x5,x6);
        or(out,y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13);

endmodule
module out2(out,x1,x2,x3,x4,x5,x6);
input x1,x2,x3,x4,x5,x6;
```

Fig - Stage 3 of execution of code for example - Verilog Code Generation

main_output Project Status			
Project File:	DS_Test.xise	**Parser Errors:**	No Errors
Module Name:	main_output	**Implementation State:**	Mapped
Target Device:	xc3s100e-5cp132	• **Errors:**	No Errors
Product Version:	ISE 14.6	• **Warnings:**	6 Warnings (6 new)
Design Goal:	Balanced	• **Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	
Environment:	System Settings	• **Final Timing Score:**	

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	15	1,920	1%	
Number of occupied Slices	8	960	1%	
Number of Slices containing only related logic	8	8	100%	
Number of Slices containing unrelated logic	0	8	0%	
Total Number of 4 input LUTs	15	1,920	1%	
Number of bonded IOBs	12	83	14%	
Average Fanout of Non-Clock Nets	4.22			

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Sat Nov 18 03:25:54 2017	0	6 Warnings (6 new)	0
Translation Report	Current	Sat Nov 18 03:26:08 2017	0	0	0
Map Report	Current	Sat Nov 18 03:26:45 2017	0	0	4 Infos (4 new)
Place and Route Report					
Power Report					
Post-PAR Static Timing Report					
Bitgen Report					

Fig - The design report of the code generated by us

xil Project Status (11/18/2017 - 04:38:23)			
Project File:	DS_Test.xise	**Parser Errors:**	No Errors
Module Name:	xil	**Implementation State:**	Mapped
Target Device:	xc3s100e-5cp132	**• Errors:**	No Errors
Product Version:	ISE 14.6	**• Warnings:**	No Warnings
Design Goal:	Balanced	**• Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	
Environment:	System Settings	**• Final Timing Score:**	

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	25	1,920	1%	
Number of occupied Slices	14	960	1%	
Number of Slices containing only related logic	14	14	100%	
Number of Slices containing unrelated logic	0	14	0%	
Total Number of 4 input LUTs	26	1,920	1%	
Number used as logic	25			
Number used as a route-thru	1			
Number of bonded IOBs	11	83	13%	
Average Fanout of Non-Clock Nets	3.26			

Detailed Reports						[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**	
Synthesis Report	Current	Sat Nov 18 04:37:54 2017	0	0	0	
Translation Report	Current	Sat Nov 18 04:38:01 2017	0	0	0	
Map Report	Current	Sat Nov 18 04:38:21 2017	0	0	2 Infos (0 new)	
Place and Route Report						
Power Report						
Post-PAR Static Timing Report						

Fig - The design report of the behavioural code

Conclusion

We choose to this project as we wanted to learn more about the synthesis of boolean expressions and how tools such a Espresso optimize the boolean expressions given to them. We saw that it was a challenging project in itself, but still had a lot of room for advancement at later stages.
The script though correctly generates all terms, it takes quite a lot of time to generate them. The main difficulty is that the number of cubes that must be considered in the process can be extremely large. Future improvements to the script are definitely possible in terms of handling the expressions (deriving heuristics techniques that produce good results in reasonable time) and other improvements in the algorithm (as the book mentions that this method is not entirely suited for CAD implementation). Our algorithm can handle don't cares, but we did not make provisions for it in the script. Handling don't cares is another future improvement. Other extensions to the project include minimizations of LUTs and Finite State Machines and also do multi level synthesis.
Another point to note is that our methods of proving correctness are really primitive. Coming up with better ways of proving it and improving the consumer usability would help the staff very much.

Appendix of Function Descriptions

<ds_algo_input.py>

algo_input
Converts expressions into cubes.

<ds_algorithm.py>

binary
Converts a number into its binary form.

star_out
Calculate the star function output of two sets.

find_pi
Takes in the initial cover and and returns the set of prime implicants.

remov_redun
Removes redundancy from the list of prime implicants.

sharp_out
Calculates the sharp function output.

essential_pm
Calculates the set of essential prime implicants.

part4
Solves part 4 of the minimisation algorithm.

rec
Helps in recursively applying the part5 function.

part5
Solves part 5 of the minimisation algorithm.

algo

Helps in integrating the minimisation algorithm in other parts of the program.

<ds_mainInput.py>

main_input
Takes the input from the user.

<ds_output.py>

get_and
Generates AND statement in verilog output file.

get_or
Generates OR statement in verilog output file.

make_func
Generates verilog output file for a particular module.

out
Generates verilog output file for all the modules for a multi bit output.

<ds_parser.py>

myxor
Calculates the xor of two expressions.

myand
Calculates the and of two expressions.

mynot
Calculates the not of an expression.

myor
Calculates the or of two expressions.

<ds_add_mul.py>

full_add
Mimics the functioning of a full adder circuit.

adder
Mimics the functioning of a ripple carry adder.

multiplier
Mimics the functioning of a multiplier circuit.

Acknowledgment

We would like to thank Prof. Joycee for giving us this opportunity to implement an optimization technique in Python. We would also like to thank Mr Vishwanath for his constant support and guidance.

Reference

1. Brown, Stephen, and Zvonko G. Vranesic. *Fundamentals of Digital Logic with Verilog Design*. McGraw-Hill, 2014.