Formal Verification of Arithmetic FPGA Circuits

ECE 667 COURSE PROJECT

Mohammad Aftab Usmani
Aftaab Mohammed

Contents

1. INTRODUCTION

The escalating cost, time, and risk associated with custom integrated circuit (IC) fabrication has driven increased field programmable gate array (FPGA) usage across electronics applications. FPGAs are larger, faster, and more power-efficient than ever, and bring a number of capabilities unavailable in custom silicon design, such as field updates, multi-function devices, and simplified prototyping, making them an attractive option. The latest FPGAs can support designs with more than 20 million equivalent gates, plus processor platforms and a range of communications, digital signal processing (DSP), and other functional blocks. These devices are a far cry from the simple programmable chips of yesteryear, where a designer could quickly load a few thousand gates of logic into an FPGA and immediately see them run. Today's devices require a comprehensive verification strategy every bit as exhaustive as that for an ASIC.

The work presented in this report presents the solution to verification of Arithmetic circuits synthesized on FPGAs. It addresses the verification problem at an algebraic level, treating an arithmetic circuit and its specification (if known) as a properly constructed algebraic system. The proposed technique solves the verification problem by function extraction, i.e., by deriving arithmetic function computed by the circuit from its low-level circuit implementation. The method can be used to verify the extracted function against the given specification (if known), or as a reverse engineering tool, to learn the function performed by the circuit. If the circuit is implemented incorrectly, the verifier will give the signature showing where the error has occurred.

1.1 PROJECT GOAL

The goal of this project is to build a Verifier for arithmetic circuits implemented on FPGAs. The requirement for this tool was to take the synthesized output of the FPGA software and convert the Boolean equations into algebraic equations and generate the output signature from the Verilog output file.

2. EXPERIMENTAL SETUP

Fig. 1 shows the block diagram of the experimental setup. In the diagram, the block labeled VO2EQN has been developed. The HDL code of the arithmetic circuit under test is synthesized using Altera QUARTUS software. Synthesized Verilog output files are then fed to the parser tool which is written partly in C++ and partly in python. The parser generates symbolic equation for Boolean formulas and then this file is fed to a substitution tool. PetBoss is the substitution tool, which generates the input signature from the output signature generated using parser.

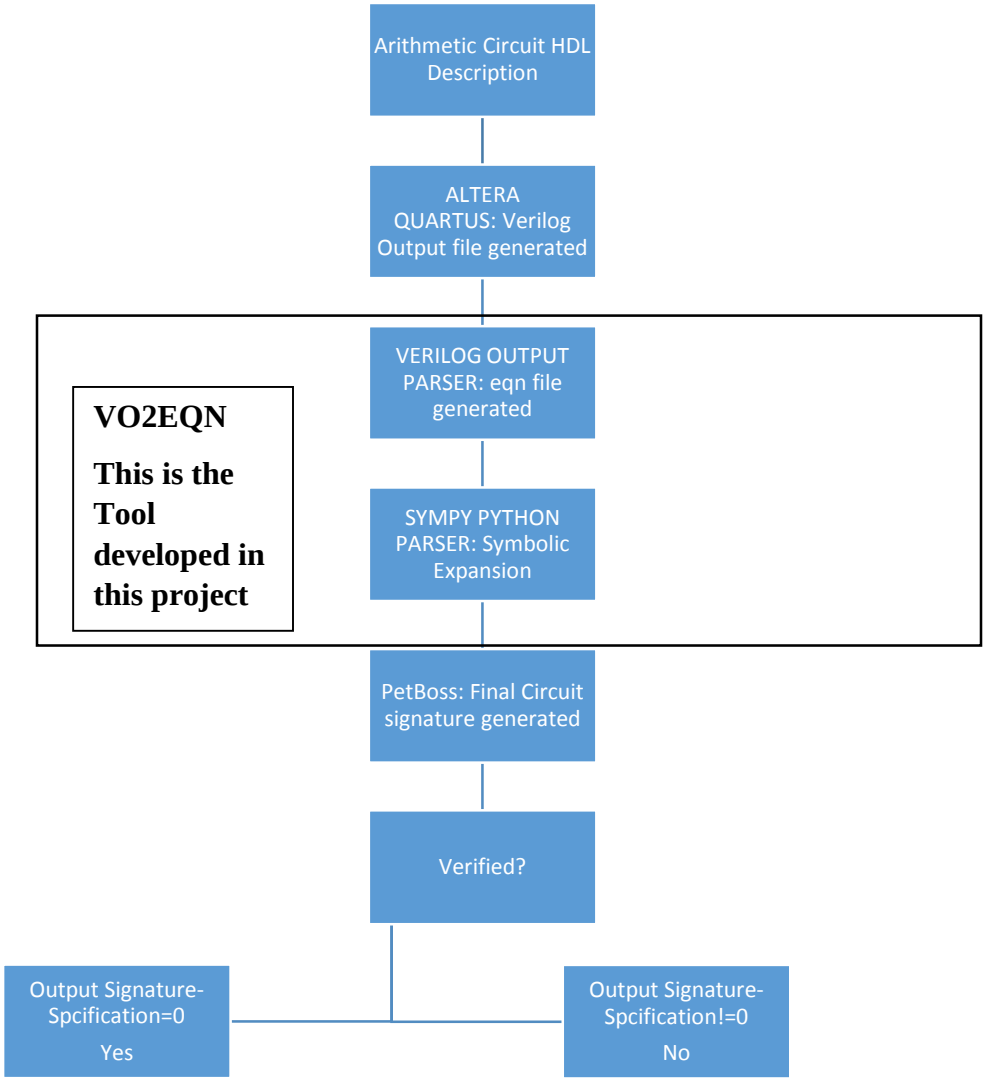

Fig. 1: Block diagram of experimental setup

3. TOOL REQUIREMENTS

The tools used for this project are:

1) Altera QUARTUS: This tool can perform different functionalities with respect to FPGAs. But in our project, we use it for synthesizing FPGA circuits written in HDL description and generating Verilog output file.

2) SYMPY: It is a python library for symbolic mathematics [2]. We have used this library for expanding the intermediate equations which were obtained using parser.

3) PETBOSS: The input to this tool is an algebraic equation file generated using the parser, it substitutes all the equations in the final output signature and generates an input signature comparing it with the golden signature of that circuit.

4.TOOL DESIGN DETAILS

4.1 DEVELOPMENT LANGUAGE

For the most part of the program, C++ was used to code the tool and in the later parts python was used as well. C++ was used to parse the Verilog output file and obtain unexpanded algebraic equations which could then be fed in to SYMPY library which can only be used with python. There was also code written in C++ to parse the XRF file generated during synthesis, we do so to generate all the variables for the circuit required by SYMPY library.

4.2 ARCHITECTURE

4.2.1 PARSER

The basic idea behind the parser was to generate algebraic equations which can be accepted by the PETBOSS tool i.e. the output of the parser tool shouldn't have the symbols or characters which cannot be analyzed by it. So to start with, we took the Verilog output file search for the equations which are necessary and sufficient and write them to an interim (virtual) file. After getting all the equations required we try to find the characters, symbols, spaces, brackets which cannot be inputted to PETBOSS and eliminate them.

4.2.2 SYMBOLIC REPLACER

Other thing to worry about the equations is to convert the Boolean symbols like #, $, !, & to their respective algebraic forms. For example, # out here represents OR function in Boolean, $ is XOR, ! is complement and & means AND function. These are replaced by their respective algebraic equations:

NOT : $\neg a = 1 - a$;

AND : $a \wedge b = a \cdot b$;

OR : $a \vee b = a + b - a \cdot b$;

XOR : $a \oplus b = a + b - 2 \cdot a \cdot b$;

The idea is adapted from DAC conference paper [1]. These equations are written in an interim file. The next part of the code is to parse the XRF file to generate all the variables required for SYMPY library to process the interim equations generated by the first part of the parser and expand them respectively.

4.3 INTERNAL PROGRAM FLOW

To summarize, the Verilog output file inputs in to parser written in C++ language, which generates intermediate equations with all the symbolic replacements. These equations are inserted in to end part of parser i.e. SYMPY library. The second part of parser takes in XRF input file and generates an intermediate file of all variables which are then again inserted in to SYMPY library which generates the main algebraic equation file.

5. PROGRAM USAGE

The program requires SYMPY module to be installed in the python packager. This can be installed by running the following command in the terminal:

pip install sympy

To run the main program, the script vo2eqn.sh has to started. This is done in terminal by using the command ***sh.*** The script takes two inputs:

1. The name of the Verilog output produced by Quartus without extension
2. The number of output bits produced by the arithmetic circuit.

sh vo2eqn.sh filename number_of_bits

The output equation file that will be produced will have the name **final.eqn**.

6. COMPILING SOURCE CODE

Requires SYMPY development package [2] for python and g++ compilers installed on the linux machine. Usually present by default. Both the files are written under C++11 standard and this flag must be passed during compilation.

g++ -std=c++11 main.cpp -o parser

g++ -std=c++11 main_2.cpp -o parser2

7. EXAMPLE:

7.1. 4-bit Multiplier Bit Circuit

7.1.1. The QUARTUS synthesis Verilog output file:

For convenience only the relevant part of the vo file is presented.

```
// \z~1_combout  = (\b[1]~input_o  & (\a[0]~input_o  $ (((\a[1]~input_o  & \b[0]~input_o ))))) # (!\b[1]~input_o  & (\a[1]~input_o  & ((\b[0]~input_o ))))
// \mult_stage1_cout0~combout  = (\b[1]~input_o  & (\a[1]~input_o  & (\a[0]~input_o  & \b[0]~input_o )))
```

```
// \mult_stage1_sum1~combout  = (\a[2]~input_o  & (\b[0]~input_o  $ (((\a[1]~input_o  & \b[1]~input_o ))))) #
(!\a[2]~input_o  & (\a[1]~input_o  & (\b[1]~input_o )))
// \z~2_combout  = \mult_stage1_cout0~combout  $ (\mult_stage1_sum1~combout  $ (((\a[0]~input_o  &
\b[2]~input_o ))))
// \mult_stage2_cout0~0_combout  = (\mult_stage1_cout0~combout  & ((\mult_stage1_sum1~combout ) #
((\a[0]~input_o  & \b[2]~input_o )))) # (!\mult_stage1_cout0~combout  & (\a[0]~input_o  & (\b[2]~input_o  &
\mult_stage1_sum1~combout )))
// Equation(s):
// \mult_stage1_sum2~combout  = (\b[1]~input_o  & (\a[2]~input_o  $ (((\a[3]~input_o  & \b[0]~input_o ))))) #
(!\b[1]~input_o  & (((\a[3]~input_o  & \b[0]~input_o ))))
// \mult_stage2_sum1~0_combout  = \mult_stage1_cout1~combout  $ (\mult_stage1_sum2~combout  $
(((\b[2]~input_o  & \a[1]~input_o ))))
// Equation(s):
// \mult_stage3_cout0~0_combout  = (\mult_stage2_cout0~0_combout  & ((\mult_stage2_sum1~0_combout ) #
((\b[3]~input_o  & \a[0]~input_o )))) # (!\mult_stage2_cout0~0_combout  & (\b[3]~input_o  & (\a[0]~input_o  &
\mult_stage2_sum1~0_combout )))
// \mult_stage2_cout1~0_combout  = (\mult_stage1_cout1~combout  & ((\mult_stage1_sum2~combout ) #
((\b[2]~input_o  & \a[1]~input_o )))) # (!\mult_stage1_cout1~combout  & (\b[2]~input_o  & (\a[1]~input_o  &
\mult_stage1_sum2~combout )))
// \mult_stage2_sum2~4_combout  = (\a[3]~input_o  & (\b[1]~input_o  & ((!\b[0]~input_o ) # (!\a[2]~input_o )))) #
(!\a[3]~input_o  & (((\a[2]~input_o ))))
// \mult_stage2_sum2~5_combout  = (\a[3]~input_o  & (\mult_stage2_sum2~4_combout  $ (((\b[2]~input_o  &
\a[2]~input_o ))))) # (!\a[3]~input_o  & (\mult_stage2_sum2~4_combout  & (\b[2]~input_o  & \a[2]~input_o )))
// \mult_stage3_sum1~combout  = \mult_stage2_cout1~0_combout  $ (\mult_stage2_sum2~5_combout  $
(((\b[3]~input_o  & \a[1]~input_o ))))
// \z~4_combout  = \mult_stage3_cout0~0_combout  $ (\mult_stage3_sum1~combout )
// \mult_stage2_cout2~9_combout  = (\b[1]~input_o  & (\mult_stage2_cout2~3_combout  & (\a[3]~input_o  &
\a[2]~input_o )))
// \mult_stage3_cout1~0_combout  = (\mult_stage2_cout1~0_combout  & ((\mult_stage2_sum2~5_combout ) #
((\b[3]~input_o  & \a[1]~input_o )))) # (!\mult_stage2_cout1~0_combout  & (\b[3]~input_o  & (\a[1]~input_o  &
\mult_stage2_sum2~5_combout )))
// \mult_stage3_xor2~combout  = (\a[3]~input_o  & (\b[2]~input_o  $ (((\b[3]~input_o  & \a[2]~input_o ))))) #
(!\a[3]~input_o  & (\b[3]~input_o  & ((\a[2]~input_o ))))
// \mult_laststage_cout0~combout  = (\mult_stage3_cout0~0_combout  & \mult_stage3_sum1~combout )
// Equation(s):
// \mult_stage3_and11~combout  = (\b[3]~input_o  & \a[2]~input_o )
// \mult_stage3_cout2~0_combout  = (\mult_stage2_cout2~9_combout  & ((\mult_stage3_and11~combout ) #
((\a[3]~input_o  & \b[2]~input_o )))) # (!\mult_stage2_cout2~9_combout  & (\a[3]~input_o  & (\b[2]~input_o  &
\mult_stage3_and11~combout )))
// \mult_laststage_cout1~0_combout  = (\mult_stage3_cout1~0_combout  & ((\mult_laststage_cout0~combout ) #
(\mult_stage2_cout2~9_combout  $ (\mult_stage3_xor2~combout )))) # (!\mult_stage3_cout1~0_combout  &
(\mult_laststage_cout0~combout  &
// \z~6_combout  = \mult_stage3_cout2~0_combout  $ (\mult_laststage_cout1~0_combout  $ (((\a[3]~input_o  &
\b[3]~input_o ))))
// \mult_laststage_cout2~0_combout  = (\mult_stage3_cout2~0_combout  & ((\mult_laststage_cout1~0_combout ) #
((\a[3]~input_o  & \b[3]~input_o )))) # (!\mult_stage3_cout2~0_combout  & (\a[3]~input_o  & (\b[3]~input_o  &
\mult_laststage_cout1~0_combout )))
```

7.1.2. Step2: Command Line: sh vo2eqn.sh muliplier1.vo 8:

Output final.eqn file has the following algebraic equations:

z7=mult_laststage_cout20
z0=a0*b0
z1=-2*a0*a1*b0*b1 + a0*b1 + a1*b0

mult_stage1_cout0=a0*a1*b0*b1
mult_stage1_sum1=-2*a1*a2*b0*b1 + a1*b1 + a2*b0
z2=4*a0*b2*mult_stage1_cout0*mult_stage1_sum1 - 2*a0*b2*mult_stage1_cout0 -
2*a0*b2*mult_stage1_sum1 + a0*b2 - 2*mult_stage1_cout0*mult_stage1_sum1 +
mult_stage1_cout0 + mult_stage1_sum1
mult_stage2_cout00=-2*a0*b2*mult_stage1_cout0*mult_stage1_sum1 +
a0*b2*mult_stage1_cout0 + a0*b2*mult_stage1_sum1 +
mult_stage1_cout0*mult_stage1_sum1
mult_stage1_cout1=a1*a2*b0*b1
mult_stage1_sum2=-2*a2*a3*b0*b1 + a2*b1 + a3*b0
mult_stage2_sum10=4*a1*b2*mult_stage1_cout1*mult_stage1_sum2 -
2*a1*b2*mult_stage1_cout1 - 2*a1*b2*mult_stage1_sum2 + a1*b2 -
2*mult_stage1_cout1*mult_stage1_sum2 + mult_stage1_cout1 + mult_stage1_sum2
z3=4*a0*b3*mult_stage2_cout00*mult_stage2_sum10 - 2*a0*b3*mult_stage2_cout00 -
2*a0*b3*mult_stage2_sum10 + a0*b3 - 2*mult_stage2_cout00*mult_stage2_sum10 +
mult_stage2_cout00 + mult_stage2_sum10
mult_stage3_cout00=-2*a0*b3*mult_stage2_cout00*mult_stage2_sum10 +
a0*b3*mult_stage2_cout00 + a0*b3*mult_stage2_sum10 +
mult_stage2_cout00*mult_stage2_sum10
mult_stage2_cout10=-2*a1*b2*mult_stage1_cout1*mult_stage1_sum2 +
a1*b2*mult_stage1_cout1 + a1*b2*mult_stage1_sum2 +
mult_stage1_cout1*mult_stage1_sum2
mult_stage2_sum24=-a2*a3*b0*b1 - a2*a3 + a2 + a3*b1
mult_stage2_sum25=-3*a2*a3*b2*mult_stage2_sum24 + a2*a3*b2 +
a2*b2*mult_stage2_sum24 + a3*mult_stage2_sum24
mult_stage3_sum1=4*a1*b3*mult_stage2_cout10*mult_stage2_sum25 -
2*a1*b3*mult_stage2_cout10 - 2*a1*b3*mult_stage2_sum25 + a1*b3 -
2*mult_stage2_cout10*mult_stage2_sum25 + mult_stage2_cout10 + mult_stage2_sum25
z4=-2*mult_stage3_cout00*mult_stage3_sum1 + mult_stage3_cout00 + mult_stage3_sum1
mult_stage2_cout23=-b0*b2 + b0 + b2
mult_stage2_cout29=a2*a3*b1*mult_stage2_cout23
mult_stage3_cout10=-2*a1*b3*mult_stage2_cout10*mult_stage2_sum25 +
a1*b3*mult_stage2_cout10 + a1*b3*mult_stage2_sum25 +
mult_stage2_cout10*mult_stage2_sum25
mult_stage3_xor2=-2*a2*a3*b2*b3 + a2*b3 + a3*b2
mult_laststage_cout0=mult_stage3_cout00*mult_stage3_sum1
z5=-8*mult_laststage_cout0*mult_stage2_cout29*mult_stage3_cout10*mult_stage3_xor2 +
4*mult_laststage_cout0*mult_stage2_cout29*mult_stage3_cout10 +
4*mult_laststage_cout0*mult_stage2_cout29*mult_stage3_xor2 -
2*mult_laststage_cout0*mult_stage2_cout29 +
4*mult_laststage_cout0*mult_stage3_cout10*mult_stage3_xor2 -
2*mult_laststage_cout0*mult_stage3_cout10 - 2*mult_laststage_cout0*mult_stage3_xor2 +
mult_laststage_cout0 + 4*mult_stage2_cout29*mult_stage3_cout10*mult_stage3_xor2 -
2*mult_stage2_cout29*mult_stage3_cout10 - 2*mult_stage2_cout29*mult_stage3_xor2 +
mult_stage2_cout29 - 2*mult_stage3_cout10*mult_stage3_xor2 + mult_stage3_cout10 +
mult_stage3_xor2

mult_stage3_and11=a2*b3
mult_stage3_cout20=-2*a3*b2*mult_stage2_cout29*mult_stage3_and11 +
a3*b2*mult_stage2_cout29 + a3*b2*mult_stage3_and11 +
mult_stage2_cout29*mult_stage3_and11
mult_laststage_cout10=4*mult_laststage_cout0*mult_stage2_cout29*mult_stage3_cout10*mult
_stage3_xor2 - 2*mult_laststage_cout0*mult_stage2_cout29*mult_stage3_cout10 -
2*mult_laststage_cout0*mult_stage2_cout29*mult_stage3_xor2 +
mult_laststage_cout0*mult_stage2_cout29 -
2*mult_laststage_cout0*mult_stage3_cout10*mult_stage3_xor2 +
mult_laststage_cout0*mult_stage3_cout10 + mult_laststage_cout0*mult_stage3_xor2 -
2*mult_stage2_cout29*mult_stage3_cout10*mult_stage3_xor2 +
mult_stage2_cout29*mult_stage3_cout10 + mult_stage3_cout10*mult_stage3_xor2
z6=4*a3*b3*mult_laststage_cout10*mult_stage3_cout20 - 2*a3*b3*mult_laststage_cout10 -
2*a3*b3*mult_stage3_cout20 + a3*b3 - 2*mult_laststage_cout10*mult_stage3_cout20 +
mult_laststage_cout10 + mult_stage3_cout20
mult_laststage_cout20=-2*a3*b3*mult_laststage_cout10*mult_stage3_cout20 +
a3*b3*mult_laststage_cout10 + a3*b3*mult_stage3_cout20 +
mult_laststage_cout10*mult_stage3_cout20

#Output Signature:
z0*2^0+z1*2^1+z2*2^2+z3*2^3+z4*2^4+z5*2^5+z6*2^6+z7*2^7

7.1.3. Step3: command line: ./PetBoss.exe -l <final.eqn

The output of the PETBOSS symbolic replacer is:

a0*b0+2*a0*b1+2*a1*b0+4*a0*b2+4*a1*b1+4*a2*b0+8*a0*b3+8*a1*b2+8*a2*b1+8*a3*b0
+16*a1*b3+16*a2*b2+16*a3*b1+32*a2*b3+32*a3*b2+64*a3*b3

which is the correct output signature for a 4-bit multiplier.

8. RESULTS

The following table lists the results obtained with various arithmetic circuits of different complexity. The table shows both the time elapsed by the VO2EQN parser tool and the Symbolic replacement tool PETBOSS.

Circuit Under Consideration	Time required by VO2EQN Tool	Time required by PETBOSS tool
4-bit Multiplier	1.8sec	0.5sec
Adder 64-bits	5sec	0.5sec
Adder 256-bits	16.5sec	1.14sec
CSA Multiplier 64-bits	40sec	26sec
Mult3_128	32 minutes	Out of memory

Table 1. Runtimes obtained for various Arithmetic circuits

9. FUTURE WORK

One of the things that can be done is forcing the synthesizer tool to use bigger black boxes for adder or multiplier circuits where applicable. This would result in reduction in the number of Boolean equations and the signature for the black box can be generated directly from its input and output because the functionality of standard block is known.

One other thing that could be done is removing the use of Python symbolic library for expression expansion and doing it directly in C++. This would result in a considerable increase in the performance of the tool.

Also, we can find a method to levelize the order of equations so that the Symbolic replacer tool petBoss doesn't run out of memory and we can use blind substitution in place of substitution with levelization.

10. CONCLUSION

The project successfully developed a parser tool VO2EQN which took the synthesized Verilog output of the QUARTUS tool and converted the Boolean equations in to algebraic equations. The execution time was quite small for the conversion. Overall the tool performance is satisfactory.

11. REFERENCES

[1]. M. Ciesielski, C. Yu, W. Brown, D. Liu and A. Rossi, "Verification of gate-level arithmetic circuits by function extraction," *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, San Francisco, CA, 2015, pp. 1-6. doi: 10.1145/2744769.2744925

[2]. SymPy Development Team (2016). SymPy: Python library for symbolic mathematics URL http://www.sympy.org.

Appendix

Link to source code and Verilog output files:

Source code + Quartus Synthesized Arithmetic circuits