FP-Gen Floating Point Generator

User Guide

Jing Pu
2012/3/19

Contents

Chapter 1 Introduction

FP-Gen is a floating point multiply-adder generator. The goal of this generator is to study how to build a very flexible floating point module as well as its validation environment, the synthesis flow, and the optimization tool. This goal is achieved by using Gensis2[1] and other software tools.

By using Genesis2, we can define many customized parameters in the FP-Gen, such as fraction width, booth type[2] and different types of multiply tree, and later generate the RTL codes automatically once these parameters are given values. Genesis also helps us build the verification environment as well as the gold model in a parameterized way. Besides, we also integrate IBM's Floating-Point Test Generator[3] in our verification flow to strengthen the test coverage. In the synthesis phase, we embed designer intent[4] by using manually relative placement in order to improve the results of the completely tool placing and routing. The optimization is another key point in FP-Gen. Due to the fact that there are a lot of changeable parameters in the design, the optimization tool can automatically optimize these parameters when some constrains are given, such as the energy and the delay.

The primary goal of this user guide is to explain how to use FP-Gen framework to generate the RTL codes, verify and synthesis the design, and optimize by giving certain constrains.

Chapter 2 Design Overview

In FP-Gen, we actually implement three floating point modules, i.e. floating point multiplier, fused multiply-add, and cascade multiply-add. In this chapter, we will do a brief overview of the logic design of the three floating point modules.

2.1 Floating Point Multiplier

The hierarchical structure of the floating point multiplier[5] is shown in the figure 1. Unlike adders, which have many architectural variants, multiplier architecture is quite fixed. The mantissa parts of inputs to the multiplier go to the partial-product generation block (booth encoders) and the partial-products are aligned in columns and are reduced by a particular CSA tree column reduction implementation. Besides the flexibility of the bit width of floating point numbers, we, however, implement several other architectural switches. One is that we add a user increment switch to enable increment signals alongside the inputs, which allows forwarding results before rounding and rounding the previous operation in the multiplier. This feature reduces the effective latency of FP operations. We also implement a collection of possible architecture choices, including booth type (1, 2 and 3), reduction tree type (array, OS[6], ZM[7], and Wallace[8]). You can generate any combinations of above using Genesis by giving proper configuration.

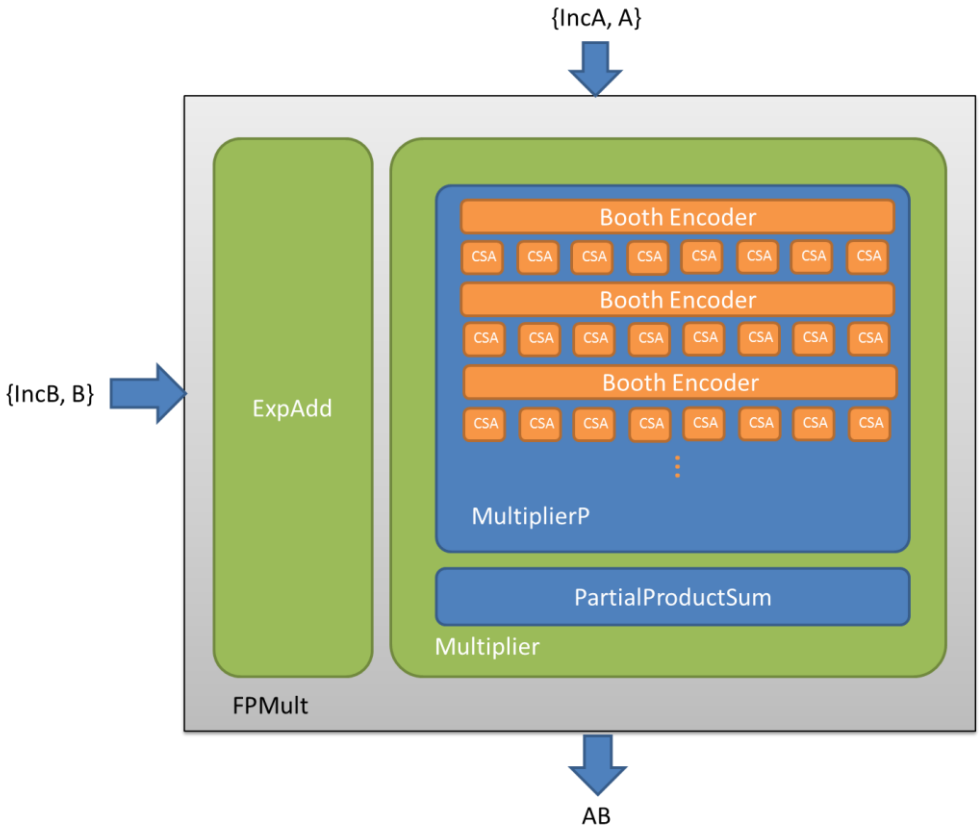

Figure 1 Structure of Floating Point Multiplier

2.2 Fused Multiply-Add

The structure of FMA[9] is illustrated in the figure 2. The exponents of A, B and C go into FMA_Exp block and the FMA_Exp produces the exponent of the final result. However, the mantissa parts of inputs go to the FMA_Man block, in which ManA and ManB get multiplied first and then added by a properly shifted ManC, and finally FMA_Man gives the final result of ManZ. The two blocks, FMA_Man and FMA_Exp, will talk to each other, sending shift amount signals to each other based on their inputs. The Rounding logic can generate an increment of Z depending on the round mode and the extra bits of Z. As you can see, we also leave an option to enable the increment signals of inputs and output.

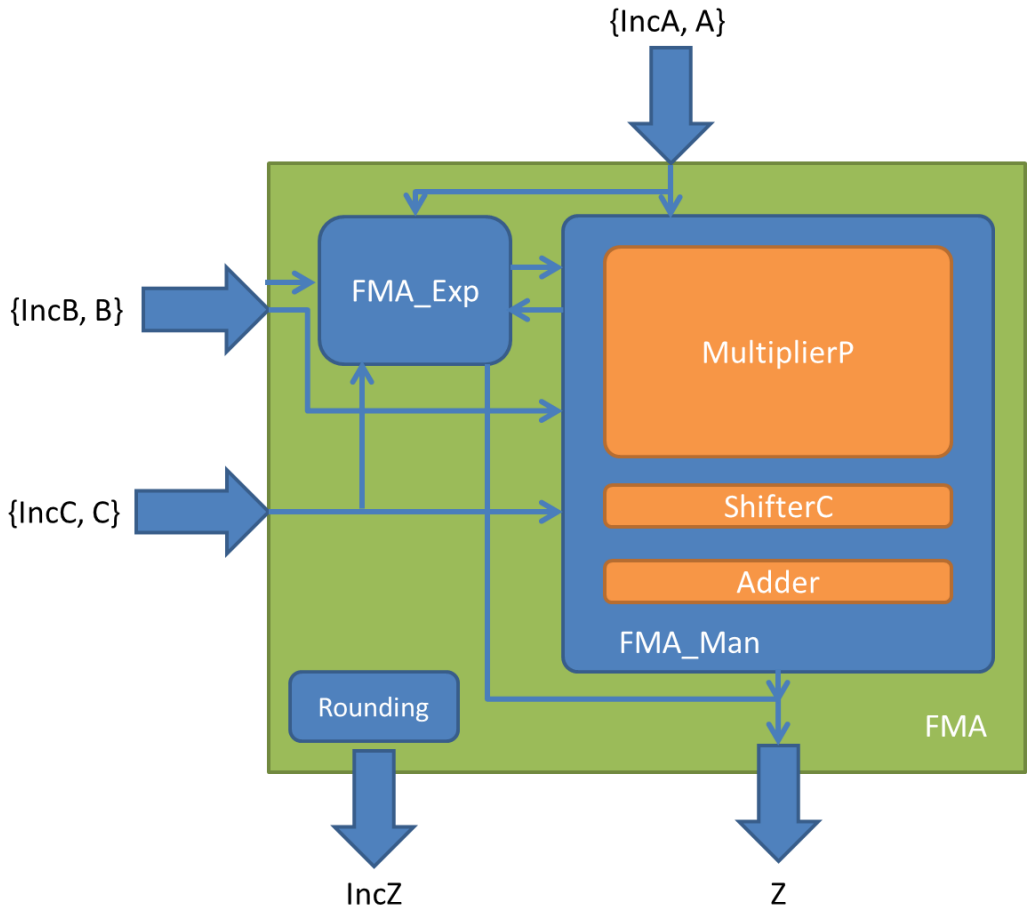

Figure 2 Structure of Fused Multiply Adder

2.3 Cascade Multiply-Add

The structure of CMA is simpler, just floating point multiply followed by floating point add, which is shown in figure 3. The floating point inputs, A and B, together with their increment signals, go into a floating point multiplier module, and the product is added by C in the floating point adder module.

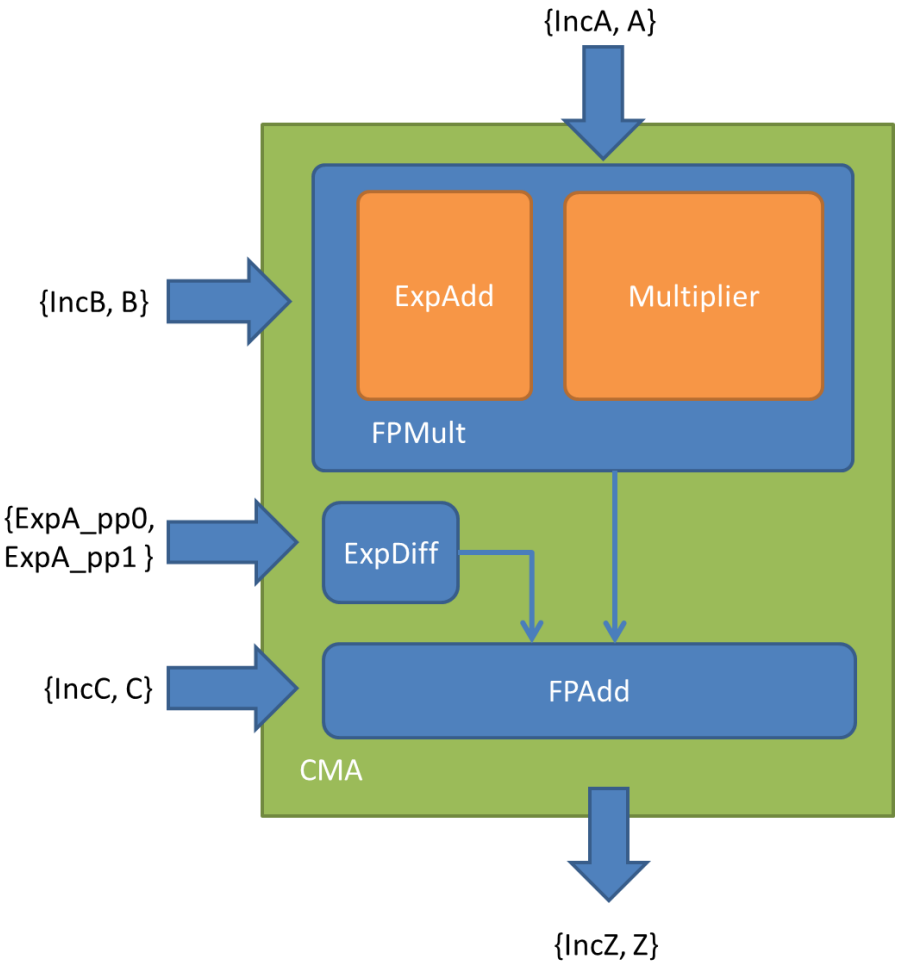

Figure 3 Structure of Cascade Multiply Adder

Chapter 3 Getting Started

This chapter introduces the workflow, file structure and make file of FP-Gen.

3.1 Workflow

FP-Gen basically uses Genesis2.pl to generate RTL codes and use VSC and Design Compiler[10] and IC Compiler[11] to verify and synthesis the design. The following figure illustrates the workflow of FP-Gen.

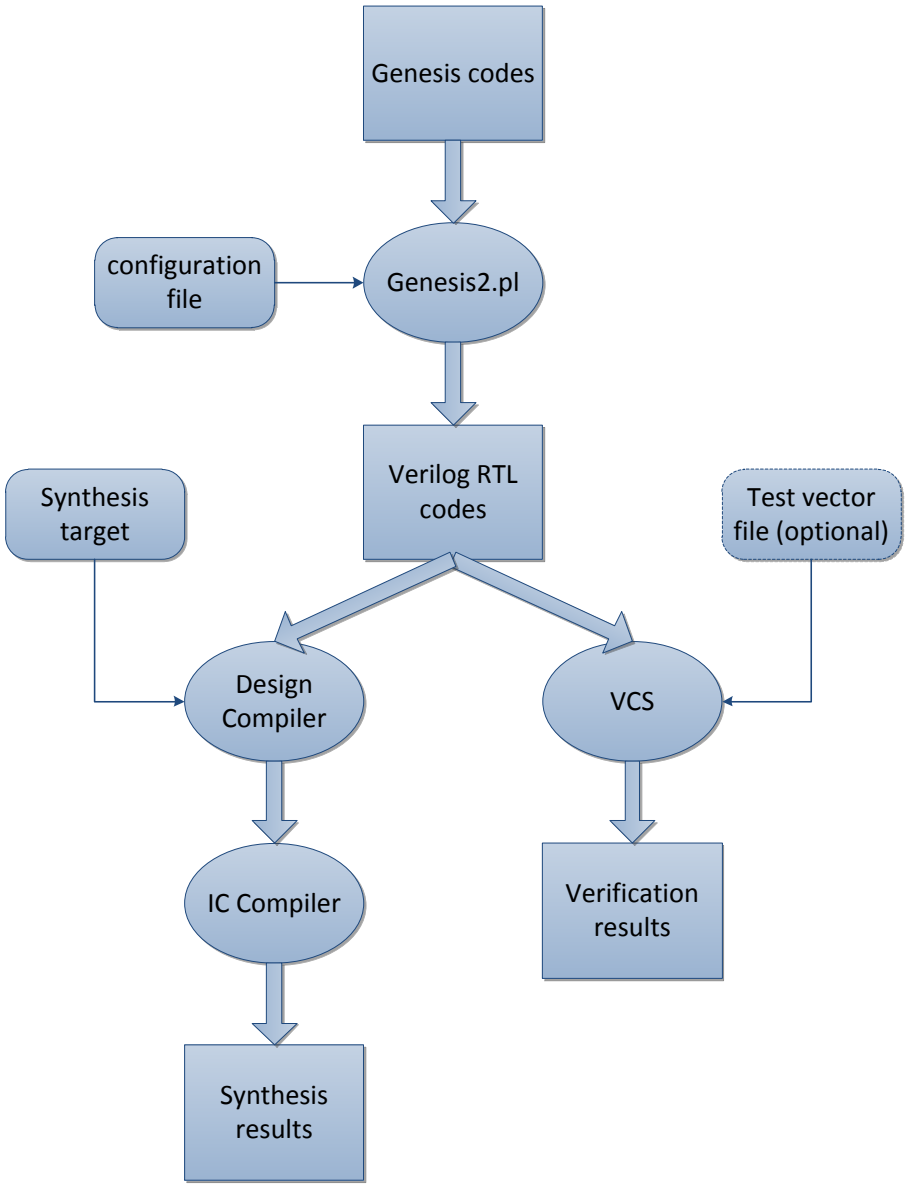

Figure 4 FP-Gen workflow

3.2 File Structure

- Make file – Makefile

  The main make file specifies the workflow of using Genesis to generate Verilog RTL codes and run verification framework.

- Genesis RTL codes – rtl/*.vp

- Genesis verification codes – verif/*.vp

- Genesis configuration files – SysCfgs/*.xml

  Genesis configuration files are a bunch of pre-stored xml files. Each defines a particular design, such as a booth 2 Wallace tree double precision FMA, and works as an input configuration of Genesis phase.

- Default configuration file – empty.xml

  The default configuration file is actually an empty xml file, and Genesis will then take the default values for parameters specified in Genesis codes.

- Synthesis tcl scripts – synthesis/*.tcl

  Synthesis tcl scripts include the dc script, the icc script and the common header script.

- Synthesis target script – synthesis/run_*.sh

  Synthesis target script defines several delay constrains for a particular run of synthesis.

- Misc. scripts – scripts/*

  Miscellaneous scripts include environment setup scripts, test vector converter script, auto synthesis and plotting scripts, optimizer scripts and etc.

3.3 Makefile

The main makefile describes the dependency of the different tasks in the workflow and some key options as well. To run the makefile, enter the following command:

```
Make [targets] [options] [RUN=+<runtime_arg[=value]>]
```

The following three tables describe possible FP-Gen targets, options and runtime arguments respectively.

Table 1 FP-Gen Makefile Targets

Target Name	**Description**
clean	Cleans up the directory
parse	Parses the Genesis codes and generate .pm files
gen	Generates Verilog codes
comp/all	Compiles Verilog code for verification
run	Runs simulation

run_wave	Runs simulation with a wave output
run_dc	Runs Design Compiler
run_icc	Runs IC compiler

Table 2 FP-Gen Makefile Key Options

Option Name	**Description**	**Default Value**
TOP_MODULE	Top module for running	top, which is the top module for Multiplier
MOD_NAME	Module name	MultiplierP
GENESIS_CFG_XML	Genesis input configuration file	empty.xml, which is an empty file
GENESIS_HIERARCHY	Genesis output configuration file	$(MOD_NAME).xml
VT	The threshold voltage of the technology for synthesis	N/A
Voltage	The drive voltage for synthesis	N/A
target_delay	The target delay of the design for dc and icc	N/A

Table 3 PF-Gen Makefile Runtime Arguments

Runtime Argument	**Description**	**Default Value**
Wave	Runs simulation with a wave output	No wave
Seed	Random seed for simulation	12345
NumTrans	Number of transactions of the simulation	10
File	Uses input test vector file instead of randomly generating test number	Randomly generating test number

3.4 Run Multiple Synthesis in Batch Mode

We also provide an easy way to run multiple synthesis jobs and plot delay-energy graphs in batch mode. To use it, enter following commands:

a. To set up the environment, use

```
source ./scripts/setup.csh  # for tcsh shell
source ./scripts/setup.bash # for bash shell
```

b. To compile .vp file and to run synthesis jobs, use:

```
./scripts/run.pl [-c] [-j=<num>] -x=<XMLFILE> -s=<SYNTHSCRIPT>
```

Example:

```
./scripts/run.pl -c -x=Designware_16_syn.xml -s=run_lvt_16.sh
-x=Designware_32_syn.xml -s=run_lvt_32.sh  -x=Designware_64_syn.xml
-s=run_lvt_64.sh -x=Designware_128_syn.xml -s=run_lvt_128.sh
-x=WlcBth2Sqr_16_syn.xml -s=run_lvt_16.sh -x=WlcBth3Sqr_16_syn.xml
-s=run_lvt_16.sh -x=WlcBth2Sqr_32_syn.xml -s=run_lvt_32.sh
-x=WlcBth3Sqr_32_syn.xml -s=run_lvt_32.sh -x=WlcBth2Sqr_64_syn.xml
-s=run_lvt_64.sh -x=WlcBth3Sqr_64_syn.xml -s=run_lvt_64.sh
-x=WlcBth2Sqr_128_syn.xml -s=run_lvt_128.sh -x=WlcBth3Sqr_128_syn.xml
-s=run_lvt_128.sh
```

Description:

Compile and run Genesis .vp file for each <XMLFILE> in ./work directory, and then run synthesis with target delay script <SYNTHSCRIPT>.

Options:

-c, --cluster: whether you are on a cluster

-j, --jobs: the maximum jobs to run on the cluster. The default is 50.

-x, --xml: input xml config file in ./SysCfgs folder to run genesis

-s, --synth: target delay script in ./synthesis folder to run synthesis

NOTE: input option <XMLFILE> must be followed by its <SYNTHSCRIPT>, and more than one pairs of <XMLFILE> and <SYNTHSCRIPT> can be set, but the numbers of input <XMLFILE> and <SYNTHSCRIPT> must be the same.

c. To collect the synthesis results, use:

```
./scripts/collectResults.pl [-f=<FOLDER>] -x=<XMLFILE>
```

Example:

```
./scripts/collectResults.pl -x=Designware_16_syn.xml
-x=Designware_32_syn.xml -x=Designware_64_syn.xml
-x=Designware_128_syn.xml -x=WlcBth2Sqr_16_syn.xml
-x=WlcBth3Sqr_16_syn.xml -x=WlcBth2Sqr_32_syn.xml
-x=WlcBth3Sqr_32_syn.xml -x=WlcBth2Sqr_64_syn.xml
-x=WlcBth3Sqr_64_syn.xml -x=WlcBth2Sqr_128_syn.xml
-x=WlcBth3Sqr_128_syn.xml
```

Description:

Collect results for each <XMLFILE>, the output csv files will be saved in <FOLDER>.

Options:

-f, --folder: define the folder to save csv files. The default is ./results

d. To plot the graphs, use:

```
./scripts/graph.pl [-f=<FOLDER>] [-n] [--Vdd=<VDD>] [--Vth=<VTH>]
-c=<CSVFILE> -d=<DESIGNWARE_CSV>
```

Example:

```
./scripts/graph.pl -c=WlcBth2Sqr_64_syn.csv
-c=WlcBth3Sqr_64_syn.csv -d=Designware_64_syn.csv
./scripts/graph.pl -n -c=WlcBth2Sqr_64_syn.csv
-c=WlcBth3Sqr_64_syn.csv -d=Designware_64_syn.csv
./scripts/graph.pl -f=MultP_results -c=WlcBth2Sqr_64_syn.csv
-c=WlcBth3Sqr_64_syn.csv -d=Designware_64_syn.csv
```

Description:

Generate a MATLAB script to plot a graph of several result csv files. Then run MATLAB, generate the graph and save a pdf version in <FOLDER>. --nodisplay option is available for non-GUI environment.

Options:

-f, --folder: define the folder to get csv files and save pdf file. The default is ./results

-n, --nodisplay: do not launch MATLAB GUI

--Vdd: the Vdd we use to plot the graph. The default is 1.0.

--Vth: the Vth we use to plot the graph. The default is lvt.

Chapter 4 Validation Framework

The FP-Gen validation framework supports both randomly generated tests and file input tests, which are integrated with IBM's FPgen test generator tool.

4.1 Randomly Generated Tests

FP-Gen can randomly generate floating point numbers using the FPTransaction module and take them as the inputs of the floating point modules, and then verify the result using the corresponding testbench and gold model. We currently support the verifications of floating point multiplier and FMA.

To verify FP multiplier, enter the following command:

```
make clean run [GENESIS_CFG_XML=xmlfile] RUN="+NumTrans=1000"
```

To verify FMA, enter the following command:

```
make clean run TOP_MODULE=top_FMA [GENESIS_CFG_XML=xmlfile] RUN="+NumTrans=1000"
```

To know more about the options and runtime arguments of the makefile, please see table 2 and table 3.

The FPTransaction module is a parameterized module, which means we can change its parameters to change the random pattern of generating floating point numbers. The following table describes the parameters in the FPTransaction.

Table 4 The Parameters of FPTransaction

Parameters	**Description**	**Default**
SignIsPos_w	Distribution weight for positive values (range 0 to 100)	50
Inc_w	Distribution weight for an increment (range is 0..100)	50
Zero_w	Distribution weight for Zero pattern	2
Denorm100_w	Distribution weight for Denorm with 100..0 pattern	2
DenormFFF_w	Distribution weight for Denorm with FFF..F pattern	2
Denorm001_w	Distribution weight for Denorm with 000..01 pattern	4
DenormRnd_w	Distribution weight for Denorm random (non-zero) pattern	10
QuietNaN_w	Distribution weight for QuietNaN pattern	10
SignalingNaN_w	Distribution weight for SignalingNaN pattern	10
Min_w	Distribution weight for Min pattern	10

Max_w	Distribution weight for Max pattern	10
Inf_w	Distribution weight for Inf pattern	10
One_w	Distribution weight for One pattern	10
PointOneOneOne_w	Distribution weight for PointOneOneOne (0.111111111) pattern	10
EzAndSml_w	Distribution weight for an easy and small number	0
Random_w	Distribution weight for Random (excluding Zero, Inf, Denorm and NaN) pattern	10

Note: the probability distribution of a pattern (except positive sign and increment) is the ratio of its weight to the sum of the weights of all the patterns (except positive sign and increment).

4.2 File Input Tests

The workflow for file input tests is to create an IBM' FPgen test generator input file to define a certain coverage model, then to use FP-gen to generate test vector files and to convert them into proper format, and finally to use runtime argument to take the input file and run the validation.

4.2.1 IBM's FPgen Test Generator

FPgen is IBM's test generator dedicated to floating point verification. Users can place data constraints on the input and output operands of floating point instructions in order to build customized coverage models as well as cross-product of these coverage models. The tool then can solve the constrains and output certain sets of input and output floating point numbers, which meet all the constrains.

To use the FPgen tool, enter following command:

a. Setup environment

   FPgen is already installed on /cad, so use following load module command to setup the environment:

```
source /cad/modules/init_modules.csh
module avail
module load FPGen
```

b. Run FPgen, use:

```
fpgen [<FPgen options>] [<fpdef-file>]
```

   The <fpdef-file> argument specifies which fpdef file should be used to guide the generation. The file should define the coverage model you want to verify and it should also have a '.fpdef' extension. For more detail about how to write .fpdef file please refer to IBM's *Floating-Point Test Generator User Guide*. The following table describes some key options for fpgen.

Table 5 Key Options for FPgen

Param	Description	Default value
D <design name>	Runs fpgen for specified processor	
S <seed>	The seed to use.	Random seed
C <cluster-size> -i <cluster-index>	Clusters the tasks to groups, each having at most <cluster-size> instructions, and runs cluster no. <cluster-index>	cluster-size = number of tasks in the file, cluster-index = 0
o <dir>	Places the fpres, fplog, and temporary files in directory <dir>.	Current directory
m <model-name>	Model override Overrides the floating point model constraint.	All the tasks defined in the fpdef file The floating point models as defined in the fpdef file
P <precision>	Precision filter Only the tasks from the fpdef that include the mnemoniccs of the required precision will be solved.	All the tasks defined in the fpdef file
M <mnemonic>	Mnemonic filter Only the tasks from the fpdef that include the required mnemonic will be solved. This option can appear more than once in the command line.	All the tasks defined in the fpdef file.
-Z <total time>	Limit on total time of generation for the fpdef file	
h	Prints a help file for FPgen.	

4.2.2 Integration with FP-Gen Validation Environment

After we get the output file of FPgen, we should first convert the format. To do this, enter following command:

```
Scripts/converter.pl [<fpres-file>]
```

The <fpres-file> argument specifies which fpres file should be converted. The file should have a '.fpres' extension, and the output file of the converter will have a '.txt' extension.

Then to run verification of these test vectors, enter following command:

```
make clean run TOP_MODULE=top_FMA [GENESIS_CFG_XML=xmlfile] RUN="+File=<txt-file>"
```

The <txt-file> argument specifies which txt file should be the input of the verification.

Chapter 5 Optimization Tool

To be done.

References

[1] O. Shacham, “Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogeneous Compute Platforms,” Ph.D. dissertation.

[2] A. Booth, “A Signed Binary Multiplication Technique”, *Q J Mechanics Appl Math*, vol. 4, no. 2, pp. 236-240, 1951.

[3] https://www.research.ibm.com/haifa/projects/verification/fpgen/

[4] https://www-vlsi.stanford.edu/mediawiki/index.php/LaTeX:CG/EmbeddingDesignIntentInAnFMA

[5] G. Colon-Bonet, and P. Winterrowd Jr, “Multiplier Evolution: A Family of Multiplier VLSI Implementations”, *The Computer Journal*, vol. 51, no. 5, pp. 585-594, 2008.

[6] Z.-J. A. Mou and F. Jutand, “’overturned-stairs’ adder trees and multiplier design,” *IEEE Trans. Computers*, 1992.

[7] D. Zuras and W. H. McAllister, “Balanced delays trees and combinatorial division in VLSI,” *IEEE journal of solid-state circuits*, vol. SC-21, pp. 814–819, 1986.

[8] Wallace, “A suggestion for a fast multiplier,” *IEEETC: IEEE Transactions on Electronic Computers*, vol. 13, pp. 13–17, 1964.

[9] E. Quinnell, E.E. Swartzlander,C. Lemonds, “Floating-Point Fused Multiply-Add Architectures”, *Asilomar Conference on Signals, Systems and Computers, 2007. ACSSC 2007.*

[10] http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/default.aspx

[11] http://www.synopsys.com/tools/implementation/physicalimplementation/pages/iccompiler.aspx