|  |
| --- |
|  |
| FP-Gen Floating Point Generator |
| User Guide |

|  |
| --- |
| Jing Pu  2012/3/19 |

Contents

[Chapter 1 Introduction 1](#_Toc320276887)

[Chapter 2 Design Overview 2](#_Toc320276888)

[2.1 Floating Point Multiplier 2](#_Toc320276889)

[2.2 Fused Multiply-Add 3](#_Toc320276890)

[2.3 Cascade Multiply-Add 4](#_Toc320276891)

[Chapter 3 Getting Started 5](#_Toc320276892)

[3.1 Workflow 5](#_Toc320276893)

[3.2 File Structure 6](#_Toc320276894)

[3.3 Makefile 6](#_Toc320276895)

[3.4 Run Multiple Synthesis in Batch Mode 7](#_Toc320276896)

[Chapter 4 Validation Framework 10](#_Toc320276897)

[4.1 Randomly Generated Tests 10](#_Toc320276898)

[4.2 File Input Tests 11](#_Toc320276899)

[4.2.1 IBM’s FPgen Test Generator 11](#_Toc320276900)

[4.2.2 Integration with FP-Gen Validation Environment 13](#_Toc320276901)

[Chapter 5 Optimization Tool 14](#_Toc320276902)

[References 15](#_Toc320276903)

# 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]](#endnote-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]](#endnote-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]](#endnote-3) in our verification flow to strengthen the test coverage. In the synthesis phase, we embed designer intent[[4]](#endnote-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]](#endnote-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]](#endnote-6), ZM[[7]](#endnote-7), and Wallace[[8]](#endnote-8)). You can generate any combinations of above using Genesis by giving proper configuration.

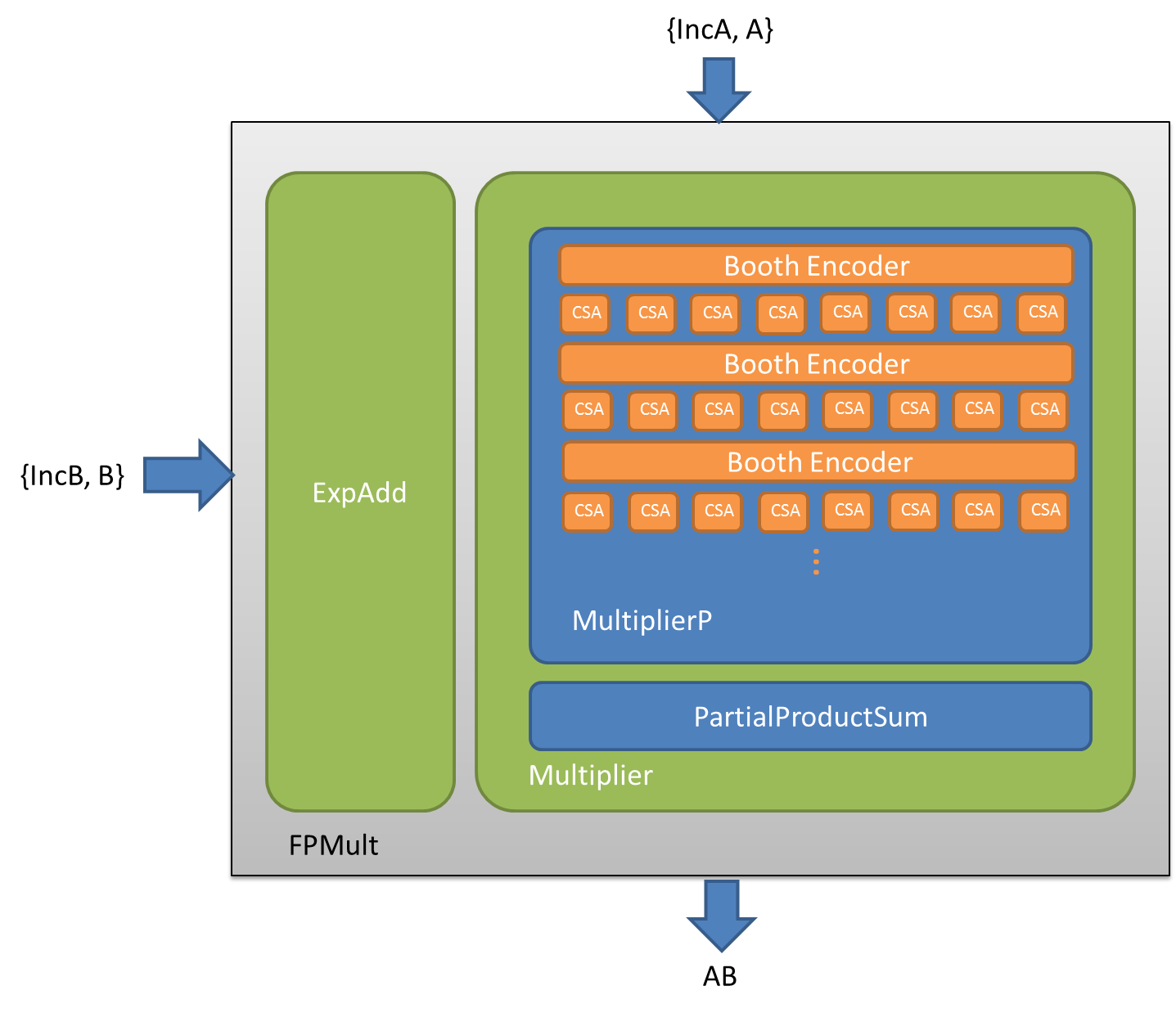


Figure Structure of Floating Point Multiplier

## 2.2 Fused Multiply-Add

The structure of FMA[[9]](#endnote-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.

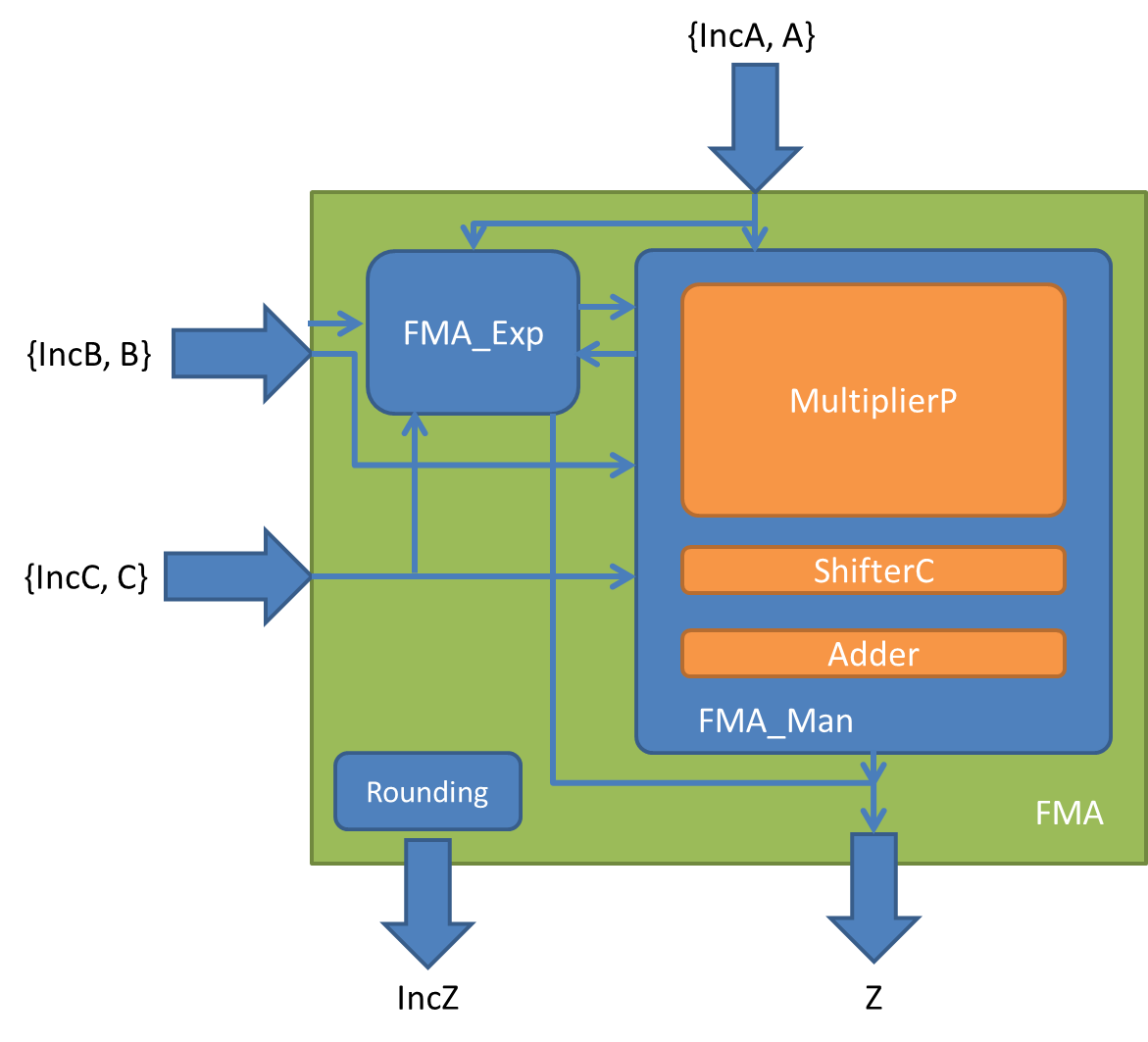


Figure Structure of Fused Multiply Adder

## 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.

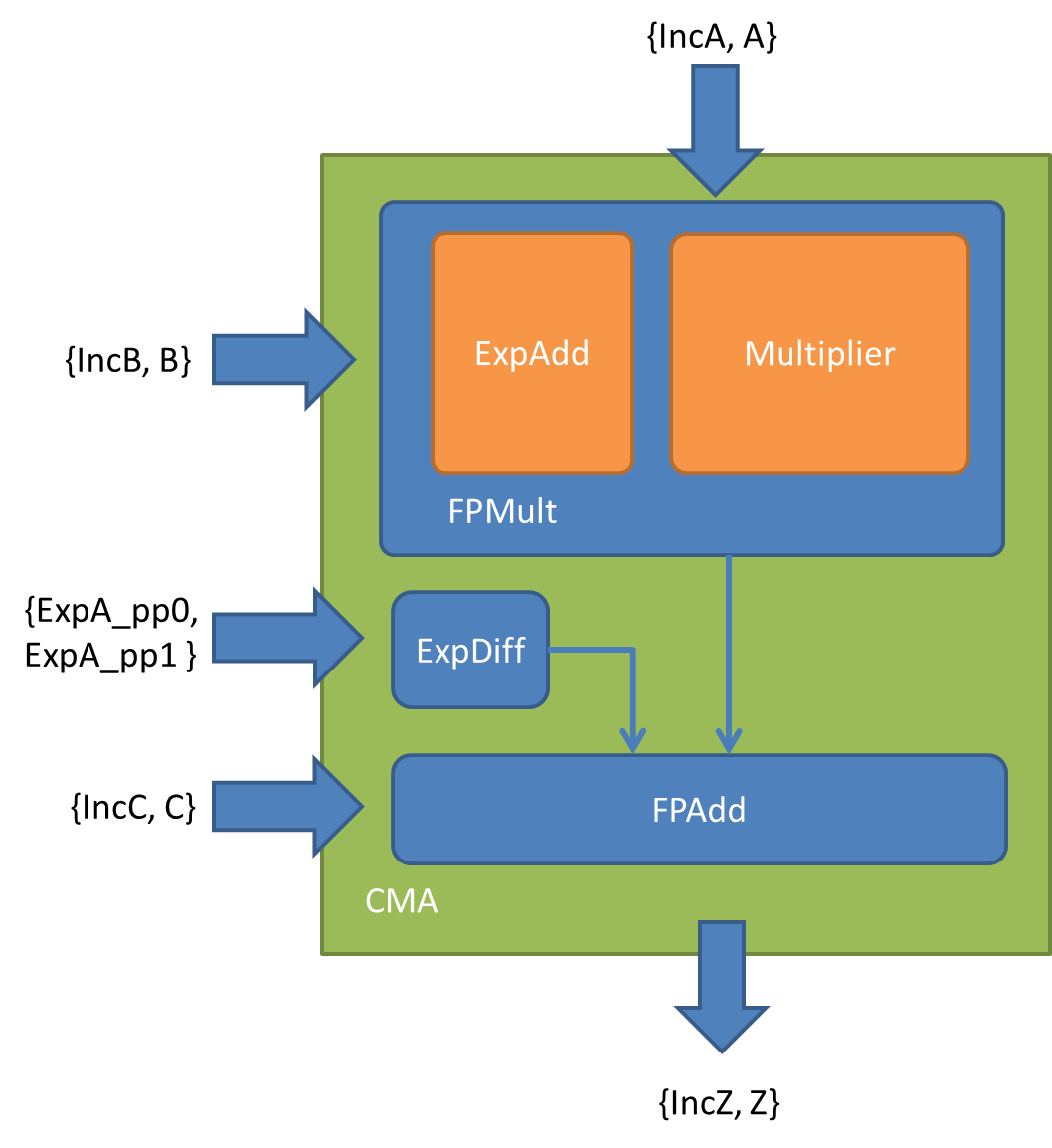


Figure Structure of Cascade Multiply Adder

# 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]](#endnote-10) and IC Compiler[[11]](#endnote-11) to verify and synthesis the design. The following figure illustrates the workflow of FP-Gen.



Figure 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 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 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 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 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:

1. 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

1. 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 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. [↑](#endnote-ref-1)
2. A. Booth, “A Signed Binary Multiplication Technique”, Q J Mechanics Appl Math, vol. 4, no. 2, pp. 236-240, 1951. [↑](#endnote-ref-2)
3. <https://www.research.ibm.com/haifa/projects/verification/fpgen/> [↑](#endnote-ref-3)
4. <https://www-vlsi.stanford.edu/mediawiki/index.php/LaTeX:CG/EmbeddingDesignIntentInAnFMA> [↑](#endnote-ref-4)
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. [↑](#endnote-ref-5)
6. Z.-J. A. Mou and F. Jutand, “’overturned-stairs’ adder trees and multiplier design,” *IEEE Trans. Computers*, 1992. [↑](#endnote-ref-6)
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. [↑](#endnote-ref-7)
8. Wallace, “A suggestion for a fast multiplier,” *IEEETC: IEEE Transactions on Electronic Computers*, vol. 13, pp. 13–17, 1964. [↑](#endnote-ref-8)
9. E. Quinnell, E.E. Swartzlander,C. Lemonds, “Floating-Point Fused Multiply-Add Architectures”, *Asilomar Conference on Signals, Systems and Computers, 2007. ACSSC 2007.*  [↑](#endnote-ref-9)
10. <http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/default.aspx> [↑](#endnote-ref-10)
11. <http://www.synopsys.com/tools/implementation/physicalimplementation/pages/iccompiler.aspx> [↑](#endnote-ref-11)