Politecnico di Torino

III Facoltà di Ingegneria

Laboratory 2
Digital arithmetic

Master degree in Electrical Engineering

Authors: Group 21

Dilillo Nicola S284963
Moncalvo Stefano S290315
Carrano Lorenzo S281565

December 12, 2021

Contents

CHAPTER 1

Prototype

1.1 Introduction

In order to better and easier simulate the architecture, two scripts have been used: A bash script to control the simulation from high level: it is in charge to invoke the C script with the right command line argument during the proper simulation phase and to run modelsim; A C script in charge to perfrom various operations during different steps of the simulation, manipulating different files. For simplicity, a single value is assigned to both operands each simulation cycle, de facto using the multiplier as a square-evaluation circuit.

1.2 General simulation flow

The C script can accepts an additional command line parameter:

- -i corresponds to **TEST_INIT** mode: the script reads the handwrittensamples.txt file, in which desired inputs in human-like format are stored, e.g. 12.29487, -0.9872 etc, and generates two other files as output, both storing data in hexadecimal encoding: simulationinputs.hex, which contains the inputs for the Modlesim simulation expected_outputs.hex, which contains the outputs that are expected to be generated by the Modelsim simulation.
- -v corresponds to **TEST_VALIDATION** mode: the script executes a diff command between the self-generated expected_outputs.hex file and the file generated by the Modelsim simulation using a pipe to execute it in background and take back its output, then processing it in order to establish if the two files are identical, Test Successful, or different, Test Failed.

The flow of the simulation is the following:

1. the bash script executes the C program passing -i as command line parameter in order to generate the input vectors for the simulation and the file storing the expected results.
2. Modelsim is launched and the effective output file is generated
3. the C program is executed passing -v as command line parameter in order to compare the simulation results with the expected ones

1.3 C script - Key Aspects

Since IEEE754 is the standard encoding that is normally used to store floating point data inside a flash memory, to convert a floating point value in its equivalent hexadecimal encoding it's sufficient to use the **union** C data type. **union** data type allows to store multiple-encoding data variables in the same memory location, thus it is sufficient read a value from the handwrittensamples.hex file and store it as a float variable inside the previously declared memory location, and then accessing it as a 32-bit data variable, using the C standard type **uint32_t**.

```
union IEEE754conv
{
    float f;
    uint32_t ieee754Value;
};
```

The previously described operation is trivially implemented by the following procedure, that is very convenient both in terms synthax and in terms of CPU time, since what is done is just a couple of memory accesses:

```
uint32_t convFloatinIEEE754(float val)
{
        union IEEE754conv conv;
        conv.f = val;
        return conv.ieee754Value;
}
```

1.4 Simulation Results

Simulating the architecture with the procedure described above, it has been confirmed that it behaves in the expected way, since the results it produces are coincident with the expected ones generated by the C script. Tests have been made with both positive and negative numbers, with different orders of magnitudes for the modules, on both Ubuntu and Centos Linux OS. In case a new simulation has to be performed on a different Linux system, a bash script, compile.bash, is provided in order to quickly recompile binaries.

CHAPTER 2

Testbench

2.1 Design modification

Before to proceed with the simulation some modifications have been added to improve the testability:

- Add registers to inputs and a reset signal, to achieve a better behavior;
- Add a VIN and VOUT signals that have the purpose to start the operation and to say when it's over.

2.2 Simulation

The testbench has been divided in three parts.

- The first part generates inputs that will feed the multiplier, that will be the element under test (DUT). Those inputs will be taken from an input file that has been generated thanks to C prototype, like explain int the previuos chapter. The same input feed both factor and the VIN signal will be keep high until all input values have been used.
- When outputs start to be ready VOUT signal will be raised and, until this signal is high, a text file will be written in order to see which will be the produced results. Then those values will be compared with golden ones, that have been generated from C prototype script, to check the correctness of multiplier.
- The last part generates the clock that will feed the input generator, the data sink and the DUT.

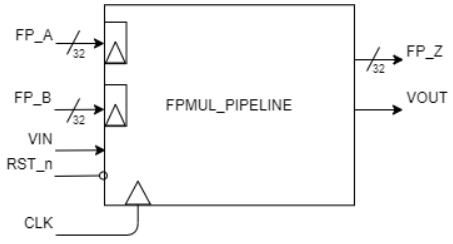

Figure 2.1:

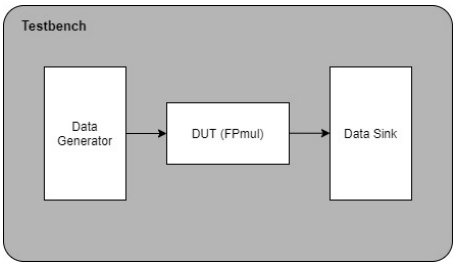

Figure 2.2:

CHAPTER 3

All Multiplier