

**Integrated Systems Architecture**

Lab 2

Digital arithmetic

**Prof. Guido Masera - Prof. Maurizio Martina**

Group: ISA25

Payamreza Pourreza – S299859

Ali Fakour Razeghi – S299860

Hooman Khedersolh Sedeh – S301148

19 December 2022

**Contents**

**1 Digital arithmetic and logic synthesizers …………………………………………………………………………………….2**

* 1. **Introduction ………………………………………………………………….……………………………….…………….2**

1. **Synthesis Strategies …………………………………………………………………….…….………………….…………………..3**
   1. **Synthesize with compile ……………………………………………………………………………….……………..4**
   2. **optimize registers command …………………………….…………………………………………….……………5**
   3. **compile ultra command ……………………………………….………………………………….…………………..6**
   4. **Force Design Compiler to flatten and implement as a CSA multiplier …………….…….………7**
   5. **Force Design Compiler to implement as a PPARCH multiplier ………………… …….……………..8**

**Results Summary ……………………………………………………………………………………………………….………………9**

**1 Digital arithmetic and logic synthesizers**

* 1. **Introduction**

**In this lab session, we are going to do floating point multiplication. To achieve the aims of this lab, we start with implementing the multiplication using the design ware components that can be used as CSA or PPARCH, then by R8-MBE (Radix 8 Modified Booth’s Encoder) and implement Dadda-like adder plane we can obtain a faster multiplier. Dadda tree uses a selection of full and half adders to sum the partial products in stages until two numbers are left. The design is similar to the Wallace multiplier, but the different reduction tree reduces the required number of gates and makes it slightly faster (for all operand sizes).**

**The IEEE 754 standard specifies a binary32 as having:**

**Sign bit: 1 bit**

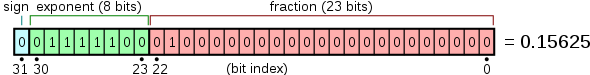
**Exponent width: 8 bits**

**Significant precision: 23 bits**

**This gives from 6 to 9 significant decimal digits precision. If a decimal string with at most 6 significant digits is converted to the IEEE 754 single-precision format, giving a normal number, and then converted back to a decimal string with the same number of digits, the final result should match the original string. If an IEEE 754 single-precision number is converted to a decimal string with at least 9 significant digits, and then converted back to single-precision representation, the final result must match the original number.**

**The sign bit determines the sign of the number, which is the sign of the significant as well. The exponent is an 8-bit unsigned integer from 0 to 255, in the biased form: an exponent value of 127 represents the actual zero. Exponents range from −126 to +127 because exponents of −127 (all 0s) and +128 (all 1s) are reserved for special numbers.**

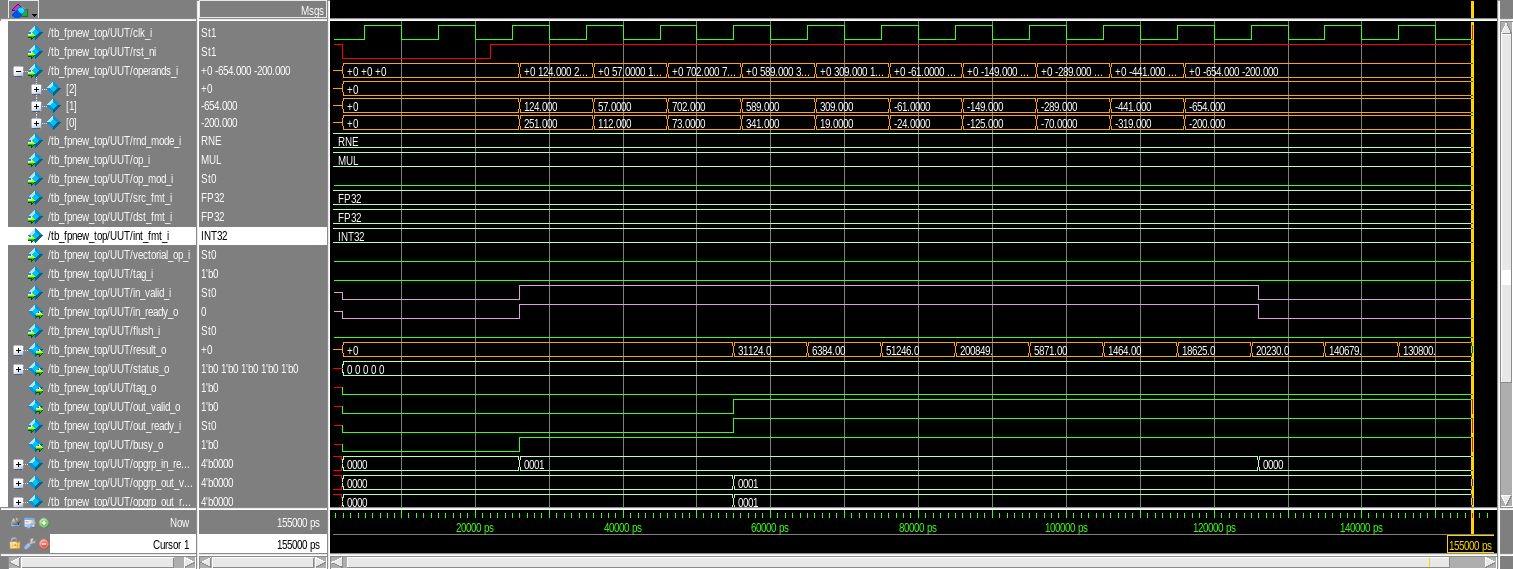
**The true significand includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point) with value 1, unless the exponent is stored with all zeros. Thus only 23 fraction bits of the significant appear in the memory format, but the total precision is 24 bits (equivalent to log10(224) ≈ 7.225 decimal digits). The bits are laid out as follows:**

****

Simulations and Logic Synthesis

1. **. Synthesis Strategies**

**Simulation and verification of the test bench:**

****

**A picture containing table

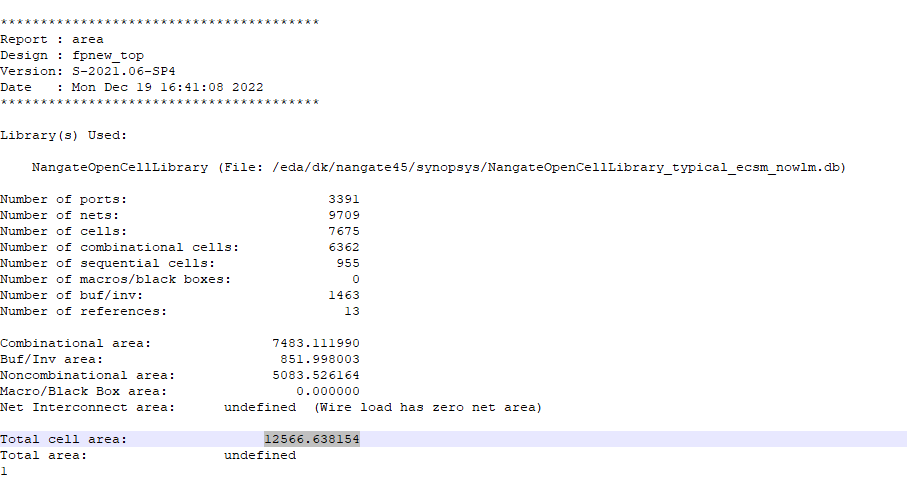
Description automatically generated**

**2.1 Synthesize with compile. Find the maximum frequency and the area.**

**Graphical user interface, text

Description automatically generated**

**2.2 Repeat the previous step issuing the optimize registers command after compile. Find the maximum frequency and the area. Verify the netlist behavior via simulation.**

****

**2.3 Repeat the previous step issuing only the compile ultra command. Find the maximum frequency and the area. Verify the netlist behavior via simulation.**

**Table

Description automatically generated**

**2.4 Force Design Compiler to flatten the hierarchy and to implement the Significands multiplier (Mantissa multiplier in fpnew\_fma.sv (3)) as a CSA multiplier. Find the maximum frequency and the area with the commands compile and optimize registers. Verify the netlist behavior via simulation.**

**Graphical user interface, text

Description automatically generated**

**Graphical user interface, text, application, email

Description automatically generated**

**2.5 Repeat the previous step by forcing the Design Compiler to implement the Significands multiplier as a PPARCH multiplier. Find the maximum frequency and the area with the commands compile and optimize registers. Verify the netlist behavior via simulation.**

**Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, email

Description automatically generated**

**Results Summary:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Slack(ns)** | **maximum frequency (MHz)** | **maximum area** |
| **compile** | 2.52 | 396.8254 | 10422.412048 |
| **optimize registers** | 1.71 | 584.7953 | 12566.638154 |
| **ultra** | 2.55 | 392.1569 | 6664.364069 |
| **CSA multiplier** | 2.37 | 421.9409 | 6918.394093 |
| **CSA multiplier (optimize registers)** | 1.69 | 591.716 | 8744.484151 |
| **PPARCH multiplier** | 2.32 | 431.0345 | 6918.394093 |
| **PPARCH multiplier (optimize registers)** | 1.68 | 595.2381 | 8716.820150 |