Politecnico di Torino

III Facoltà di Ingegneria

Digital Arithmetic and Logic Sythesis

Master degree in Electriconic Engineering

Integrated Systems Architectures

Authors: Group 34

github repository link :

https://github.com/alessionaclerio22/Integrated-Systems-Architecture-Labs

Simone Di Blasi, Stefano Floridia, Alessio Naclerio

Contents

CHAPTER 1

Floating Point Multiplier Synthesis

The main goal of this laboratory is to exploit Synopsys Design Compiler to compare the results obtained by different synthesis performed on the floating point multiplier. This is a pipelined component that is included in the "Floating Point Adder and Multiplier" project, which has been downloaded from Portale della Didattica.

1.1 Multiplier Simulation

First of all, a testbench has been created in order to test the correctness of the multiplier. Hence, the work environment has been organized in three main folders, *src* containing all the source *.vhd* files, *tb* with all testbench-related files and *sim* for the simulation with *ModelSim*. Then, *sim* has been subdivided in different subfolders. In order to perform the simulation, *clk_gen.vhd*, *data_maker.vhd* and *tb_fpmul.v*" in folder *tb* have been used. They respectively generate a 10ns clock signal, feed the multiplier with the values provided in *fp_samples.hex* and connect the DUT to the testbench. Then, in subfolder *sim_init*, the script *compile.sh* has been run in order to compile all needed files. At last, the simulation has been carried out using *ModelSim* and a *pdf* file has been generated with the resulting waveforms. The results have been compared with the one in *fp_prod.hex* in order to check their correctness.

The same flow has been adopted for the next point of the assignment, where it is asked to add a register to the inputs of the multiplier. After properly modifying *fpmul_pipeline.vhd*, the previous steps have been followed changing the simulation folder to *sim_reg*. At last, a *pdf* file has been created showing the waveforms and the obtained results have been checked against the ones contained in *fp_prod.hex*.

1.2 Synthesis

Using the floating point pipelined multiplier with the additional registers at the inputs, various synthesis have been carried out. As a matter of fact, the aim of this part of the laboratory is to force Synopsys Design Compiler to use a specific architecture for the behavioural multiplication in *fpmul_stage2_struct.vhd* and compare the different results in terms of maximum frequency and area. In order to do this, a folder called *syn* has been created. As requested by the assignment, three synthesis have been performed respectively asking to:

- force Synopsys Design Compiler to flatten the hierarchy;

- force Synopsys Design Compiler to flatten the hierarchy and implement the multiplier in Stage2 as a CSA multiplier;
- force Synopsys Design Compiler to flatten the hierarchy and implement the multiplier in Stage2 as a PPARCH multiplier.

In order to fulfill these requirements, the subfolders *syn_first*, *syn_CSA* and *syn_PPARCH* have been created inside *syn*, each related to the correspondent synthesis. Three similar *tcl* scripts have been exploited to speed up the Synopsys Design Compiler flow, *syn_first.tcl*, *syn_CSA.tcl* and *syn_PPARCH.tcl*. They all contain the same commands in order to set up the synthesis and to flatten the hierarchy ($ungroup - all - unflatten$). However, $syn_CSA.tcl$ and *syn_PPARCH.tcl* force an implementation with the command *set_implementation*, while *syn_first.tcl* does not. In order to obtain the maximum frequency, an iterative approach has been adopted setting the clock to 0ns as first step and then adding the retrieved slack until *report_timing* gives a null slack. Three files have been generated for each synthesis containing the outcome of the commands *report_resources*, *report_area* and *report_timing*. The obtained values have been summarized in Table.

1.3 Fine-grain Pipelining and Optimization

In this part of the laboratory, it is asked to modify the Stage2 structure by adding a register at the output of the significand multliplier and all the required ones to preserve the right pipeline behaviour. In order to do this, a file called *fpmul_stage2_struct_modified.vhd* has been created. It contains the Stage2 architecture modified as requested by the assignment. It has been simulated using the same testbench files mentioned before, changing the simulation environment to folder *sim_modified*. In this folder, the script *compile.sh* has been used to compile the needed files, this time substituting *fpmul_stage2_struct.vhd* with *fpmul_stage2_struct_modified.vhd*. A *pdf* file has been generated to check the correctness of the waveforms and the results have been compared to the ones in *fp_prod.hex*.

Using the modified structure for the floating point multiplier, two additional synthesis have been carried out. The *syn* subfolders *syn_modified* and *syn_modified_ultra* have been used for this purpose. The main goal of this part is to investigate the results obtained by the commands *optimize_registers* after *compile* and *compile_ultra* instead of the combination of the previous two. The synthesis have been perfomed using *tcl* scripts issuing the above mentioned operations. An iterative approach has been adopted in order to obtain the maximum frequency, as described in the previous section. Moreover, *report_area* and *report_timing* have been generated in each subfolder. The results concerning maximum frequency and area are summarized in Table.

CHAPTER 2

Modified Booth Encoding Multiplier

2.1 MBE Multiplier Design

In order to complete the set of comparisons among different multiplier's implementations, a modified-Booth-encoding multiplier for unsigned data has been designed. In the folder *MBE_multiplier*, three subfolders have been created, *src*, *tb* and *sim*. In order to follow the specifications given by the assignment, partial products have been generated without using any adders or subtracters, the adder plane has been implemented relying on a Dadda tree and sign extension bits have been simplified as proposed in the file *sign_extension_booth_multiplier_Stanford.pdf*. Once the simulation has been carried out to check its correctness, the MBE multiplier has been exploited in the Stage2 of the floating point multiplier, instead of the behavioural one "$*$".

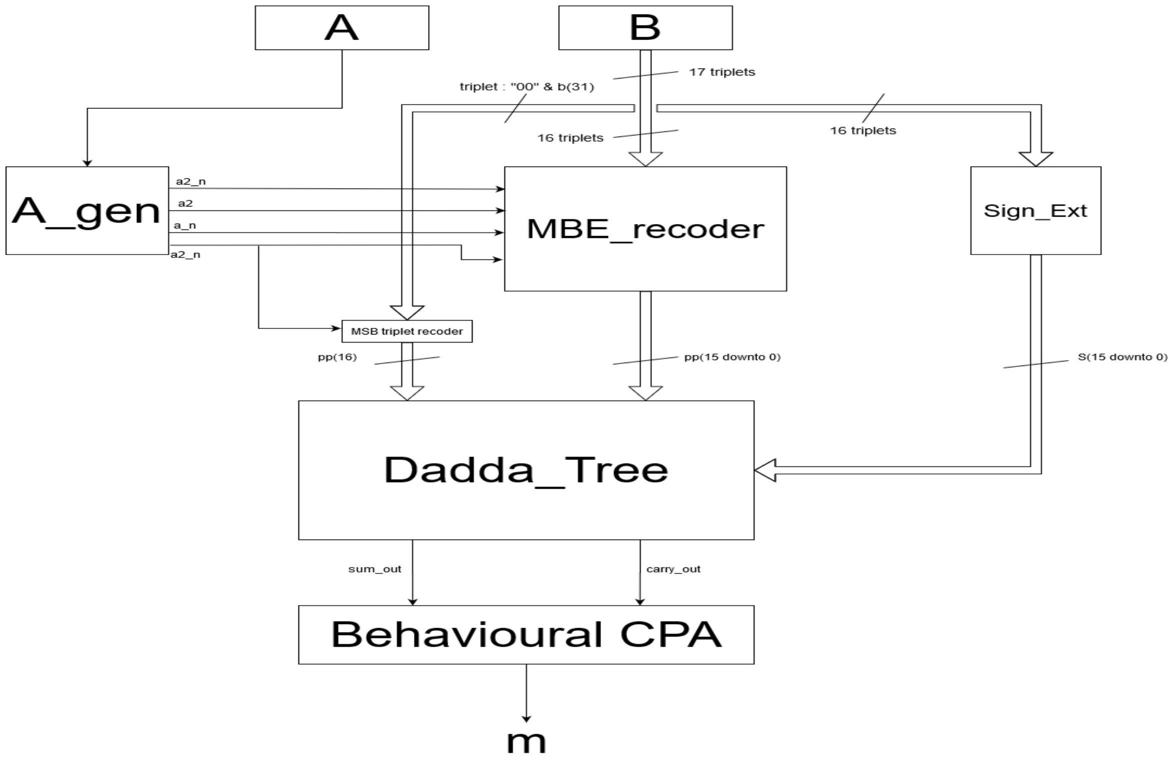

Figure 2.1: Modified Booth encoding multiplier block diagram

As depicted in figure, the main blocks used in this design are:

- *MBE_recoder*: based on the input triplet coming from the operand b, it provides the correct partial product at the output, according to the table shown in Figure 2.2;
- *A_gen*: it generates the values that will be fed to the $MBE_recoder$. It takes a on 32 bits as input and it provides four 33-bit outputs, a (setting '0' as MSB), $2a$, 1's complement a and 1's complement $2a$;
- *Sign_Ext*: it selects the value for S as indicated in the file *sign_extension_booth_multiplier_Stanford.pdf*. If the partial product is negative, S will be equal to '1' in order to add 1 to the toggled version of a or $2a$ and generate the correct 2's complement value. If the partial product is positive, S will be '0' in order to clear the sign extension bits;
- *Dadda_Tree*: it performs the addition on the 17 sign-extended partial products as stated by the Dadda algorithm. Starting from the initial "Staircase" form, partial products have been reorganized in a "V-shape" structure, where each column contains equally-weighted bits. Also S bits have been properly assigned. Since 17 partial products are present, 6 reduction operations have to be performed before having the final 2 operands on which a behavioural addition has been perfomed. At each level, the minimum number of HA and FA have been assigned in order to reach the next level's maximum heigth. As described by comments in file "Dadda_Tree.vhd", a three-dimensional array has been exploited to have 7 different "V-shape" matrices, each corresponding to a reduction level. For each of this levels, the Dadda Algorithm has been applied, until reaching the final two partial products;

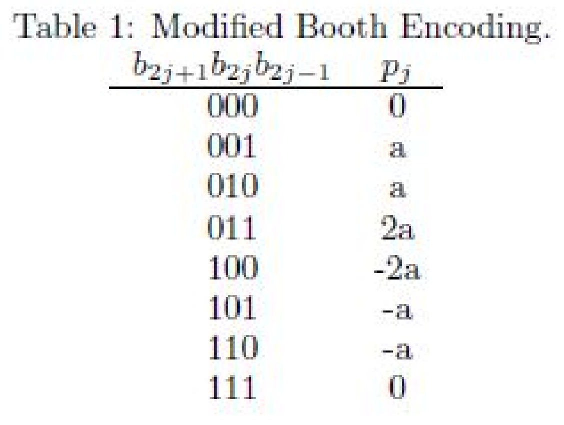

Table 1: Modified Booth Encoding.

$b_{2j+1}b_{2j}b_{2j-1}$	p_j
000	0
001	a
010	a
011	2a
100	-2a
101	-a
110	-a
111	0

Figure 2.2: Recoding Table

All these components have been used in the top entity "Dadda_Multiplier", connected as depicted in figure. As mentioned before, a behavioural adder "+" has been exploited in order to add the final reduction level partial products.

2.2 MBE Multiplier Simulation

A simulation of the designed multiplier has been carried out using *tb_Dadda_Mul.vhd* in subfolder *tb*. This testbench exploits a 32-bits LFSR generating the multiplier's inputs. The same values are given to a behavioural unsigned multiplier in order to check the correctness. If the two outputs are equal, a "check signal" is equal to one. The simulationn has been performed in the subfolder *sim* using the script *compile.sh* to compile the needed files. It has been run for in order to verify the correct behaviour of the component and a *pdf* file containing the resulting waveforms has been saved.

Then, the a new file called $fpmul\_stage2\_struct\_MBE.vhd$ has been created starting from $fpmul\_stage2\_struct.vhd$ by replacing the behavioural multiplier with the MBE one. A new subfolder named sim_MBE has been created in folder *sim* and the simulation has been run with the help of *compile.sh*. The resulting values have been checked and a *pdf* file has been generated with the obtained waveforms.

2.3 MBE Multiplier Synthesis and Final Comparison

Once the correctness has been verified, a subfolder called syn_MBE has been created in *syn* and set as work environment for the synthesis. It has been perfomed using a *tcl* script and an iterative approach has been adopted in order to obtain the maximum frequency, as described in the previous paragraph. Moreover, an area report and a timing report have been produced in order to check the results concerning maximum frequency and area, which are summarized in Table along with all the others.

Name	t_{min}[ns]	f_{max}[MHz]	area[µm^2]
syn_first	1.56	641.025641	4047.721967
syn_CSA	4.28	233.644859	3712.828004
syn_PPARCH	1.56	641.025641	4097.197964
syn_modified	0.87	1149.42528	5572.433932
syn_modified_ultra	1.50	666.666666	4207.321944
syn_MBE	4.10	243.902439	6905.625925

Tab 1: Comparisons Table.

2.3.1 Final Overall Comparisons

Regarding the maximum frequency, the best result has been obtained with the synthesis using the command $optimize\_registers$, which applies a proper retiming. The same clock has been retrieved for the first synthesis and the one forcing a PPARCH implementation. This is due to the fact that a pparch architecture has been used by both of them, with the only difference in the radix used. In the first one a radix-4 has been adopted, while in the second a radix-8. The worst one in terms of timing is the CSA one, with 4.28nsecond. As shown in figure, the best one in terms of area occupation is the CSA, while the worst is the MBE implementation. As before, the first synthesis and the PPARCH one are very similar, with the latter a little bit greater than the first. As regards the *compile_ultra* results, it has a lower area occupation than the $optimize\_registers$ version. However, it also shows a greater clock period leading to a lower maximum frequency.