Politecnico di Torino

02GQCOQ - INTEGRATED SYSTEMS ARCHITECTURE

A.A. 2020/2021

PROF. MAURIZIO MARTINA
PROF. GUIDO MASERA

LAB2

Digital Arithmetic

Group 18

---

https://github.com/wackozz/isa/tree/main/lab2

Federica BONGO	s292395
Stefano RIZZELLO	s288013
Francesco VACCA	s279895

Contents

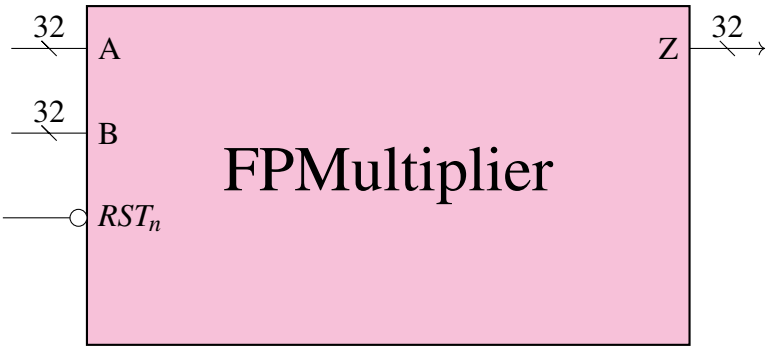

Figure 1: Multiplier top entity

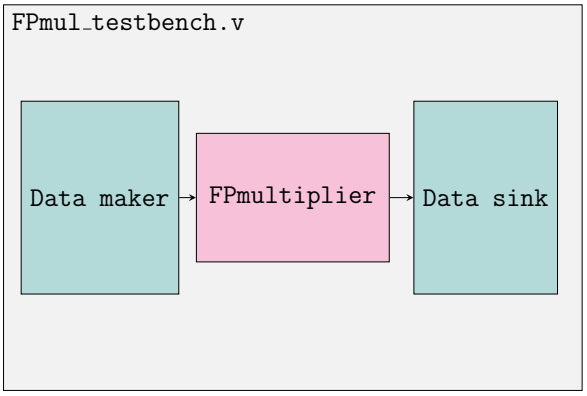

Figure 2: Testbench structure for the FPmultiplier

1 Digital arithmetic and logic synthesys

The purpose of the first part of the laboratory is to verify and synthesize an RTL for a 32 bits pipelined floating point multiplier. The top entity of the multiplier is reported in Figure 1, and is composed of four internal stages.

1.1 Verification of a pipeline multiplier

The correct behavior of the multiplier was checked through ModelSim. A testbench was created, composed of a data maker and a data sink module.

The structure of the testbench is reported in Figure 2. For every clock cycle, an hex value is taken from a text file by the data maker and fed to both A and B input. The output result Z, corresponding to the square of the input value, is printed to an external file by the data sink. The results were checked redoing the operations in C and are conform to the IEEE-754 floating point standard (single-precision).

Registers insertion As per request, two registers were added in input to the multiplier. Sampling aside, the behavior of the device remained the same.

1.2 Comparison between different implementations

The multiplier was synthesized with Synopsys' Design Compiler.

The synthesizer was initially set to flatten the hierarchy, leaving every operand to be automatically mapped. Then it was instructed to implement the significands multiplier in Stage2 with a CSA multiplier and then with a PPARCH one. Both architectures are part of the internal DesignWare library.

To automate the task of getting the minimum clock period, a custom python/tcl script was coded. Starting from $T_{ck} = 0$, the script iteratively issues a new synthesis for the RTL, increasing the clock value until a zero slack value is found.

Timing and area for the three implementations are reported in Table 1 and in Table 2 respectively.

Fine-grain pipeline and optimization The HDL description of the stage2 was modified in order to add a register at the output of the significands multiplier. Another register and some flip-flops were instantiated in order to correctly handle the timing.

The synthesis was then repeated for the three cases issuing the command `optimize_registers`. Finally, the steps were repeated issuing the command `compile_ultra` instead of `optimize_registers + compile`.

2 MBE multiplier design

The goal for the second part of the laboratory is to design an MBE[1] multiplier for unsigned data. By specifications, the adder plane has to be implemented with a Dadda tree, simplifying the sign extension in order to reduce the complexity of the device.

The purpose of the device is to be instantiated in the stage2 of the FPmultiplier to substitute the `A*B` line that describes the significands multiplier.

2.1 RTL design

Paralellism The multiplier block structure is illustrated in Figure 4. The stage2 multiplier instantiated through the `*` operand of the `numeric.std` package is a 32x32 bits one, where only bits 47 to 20 are brought in output to the next stage.

For this reason, it was chosen to design a 24x24 bits multiplier as replacement.

Encoder Input B is first padded left and right with one and two zeros respectively. Assuming $N_b = 24$, the number is then splitted in 13 triplets $(N_b/2+1)$ in the form $[b_{j+1}, b_j, b_{j-1}]$, with $j = 0, 1, ..., N_b - 1$.

[1] Modified Booth Encoding

Every encoder is fed with a triplet and A in input, and outputs a partial product p_j according to the MBE algorithm, which is reported in Equation 1.

Sign extension Because of the employed sign extension technique, it is convenient to first compute the partial product P in module inside the encoder ($Q = |P|$). Then, to account for the sign, Q is xored with the MSB of the triplet and and brought in output either as its one's complement $\overline{P}$ or its module $P = Q$. Finally, the $+b_{j+1}$ addition necessary to possibly complete the two's complement conversion is embedded in the next partial product addend.

Sum Before entering the Dadda module, every partial product P_i is shifted left of $2 \times i$ positions to account for its weight in base-4. A graphical summary of the operation is reported in Figure 3.

Dadda All the partial products are supplied in input to the Dadda module. First, the values are mapped to a 28x13 bit matrix in order to build the reverse pyramid figure. Then, the Dadda compression algorithm is applied to the matrix iteratively, until it flattens down to two 28 bits vector. The sum of those vectors give the final result of the multiplication Z.

Python scripts Given the complexity of the dadda tree, two python script has been coded to generate a netlist for a custom N-bit MBE multiplier and its relative dadda module. The graphical output of the dadda tree script, accounting for the compression algorithm management, is reported in appendix.

2.1.1 Synthesis

The synthesis of the FPmultiplier, with MBE multiplier instantiated as the significands multiplier in stage 2, has been made for all the previous cases, as illustrated in the past sections. Results are reported in Table 2 and Table 1.

$$q_j = \begin{cases} 0 & \text{for } (\overline{b_{2j} \oplus b_{2j-1}})(\overline{b_{2j+1} \oplus b_{2j}}) \\ a & \text{for } (b_{2j} \oplus b_{2j-1}) \\ 2a & \text{for } (\overline{b_{2j} \oplus b_{2j-1}})(b_{2j+1} \oplus b_{2j}) \end{cases} \tag{1}$$

$$p_j = (\overline{b_{2j+1} \oplus q_j})$$

2.2 Final remarks

The final results for all the discussed cases are reported in Table 1 and Table 2. Graphical representations summarizing the obtained data are illustrated in Figure 5 and Figure 6.

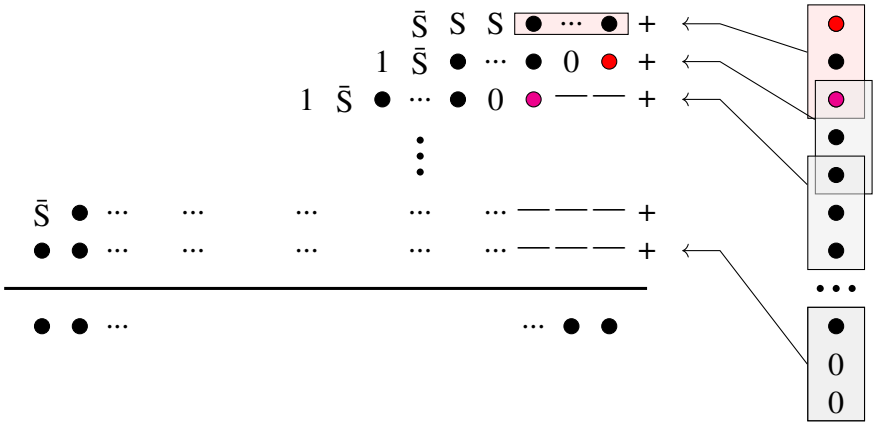

Figure 3: Scheme for sum of partial products

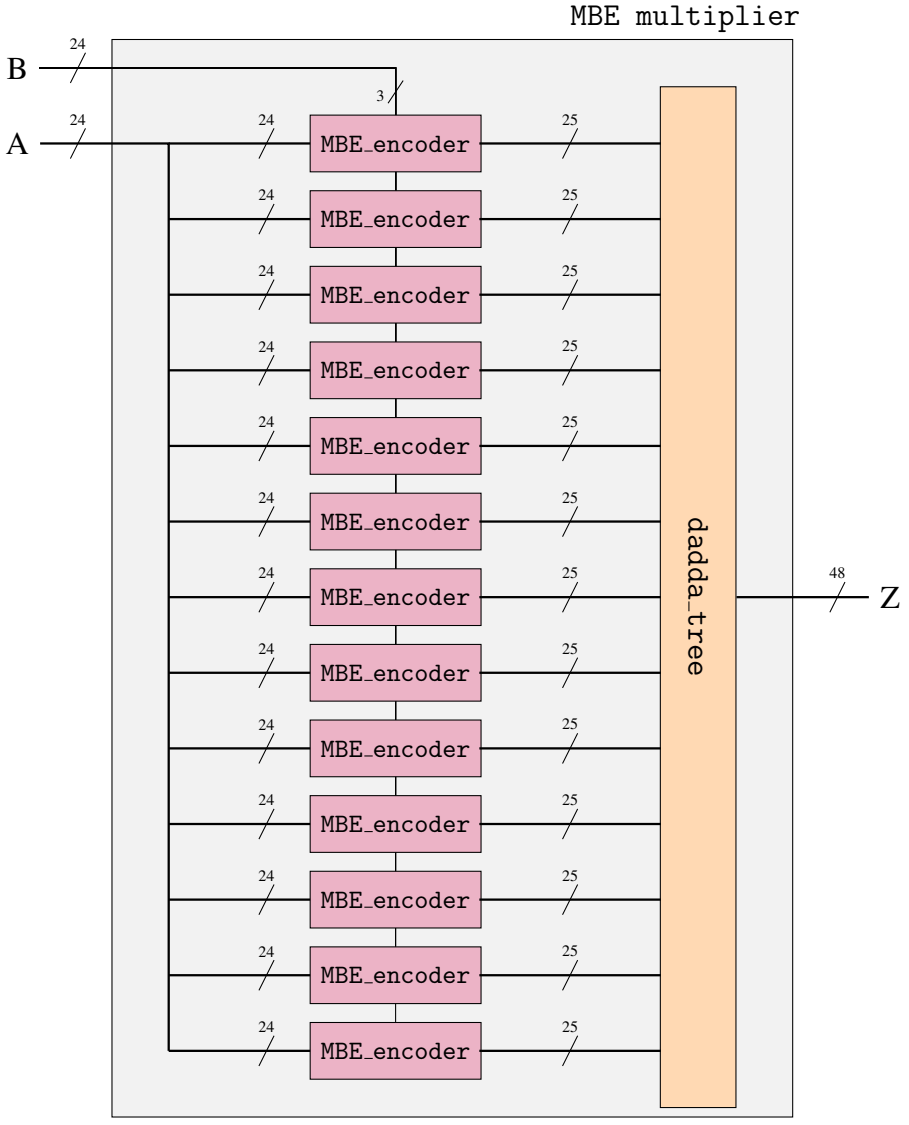

Figure 4: MBE multiplier block structure

	Auto [ns]	CSA [ns]	PPARCH [ns]	MBE [ns]
No Regs	1.58	4.29	1.56	2.74
Optimize	0.87	1.33	0.79	0.79
Compile Ultra	1.5	1.6	1.6	1.54

Table 1: Minimum clock period

	Auto [μm^2]	CSA [μm^2]	PPARCH [μm^2]	MBE [μm^2]
No Regs	4113	4899	4153	4908
Optimize	4623	6023	5266	7939
Compile Ultra	4276	4145	4145	5053

Table 2: Area reports

As showed in the figures, the PPARCH implementation of the significands multiplier is the fastest with and without registers insertion, while the automatic selection gives a better result with the compile ultra command. Regarding area, the automatic selection wins in the first two cases, while PPARCH occupies less area in the last case.

The MBE multiplier implementation performs similarly to the best in the second and last cases, even though it's the worst one by area allocation in all the three cases.

Automatic choices The custom multiplier selections can give better results than the automatic one for both optimize registers and compile ultra option. The first option gives a better result for timing, while the second is to be preferred for area saving. Overall, the results are a confirmation that the design engineer has to carefully instruct the CAD tools on design choices, and that automatic designs are not always the best ones.

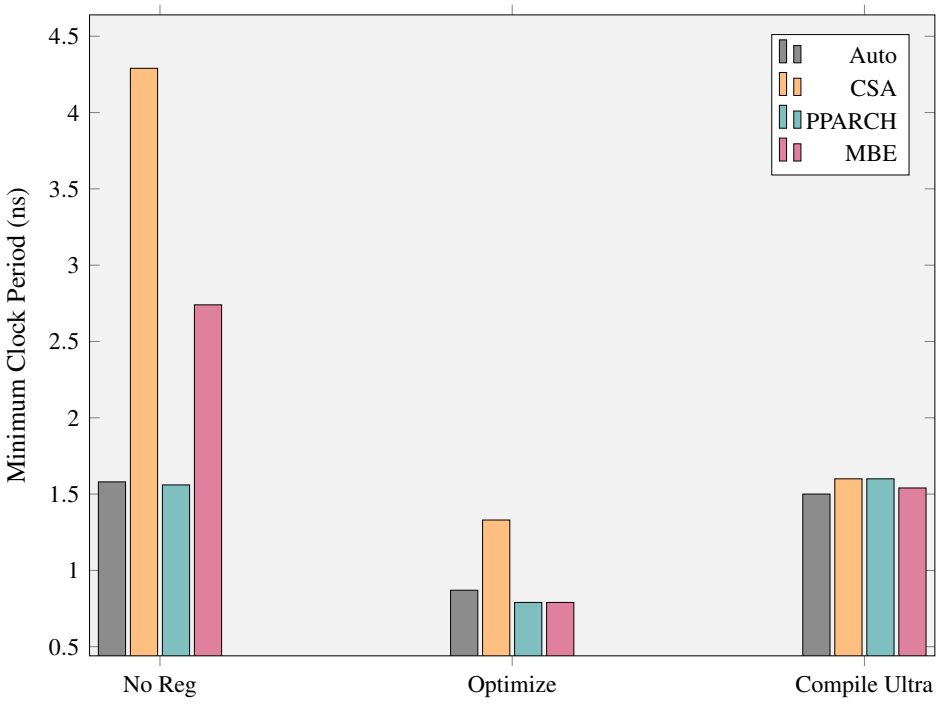

Figure 5: Timing comparison between different implementations

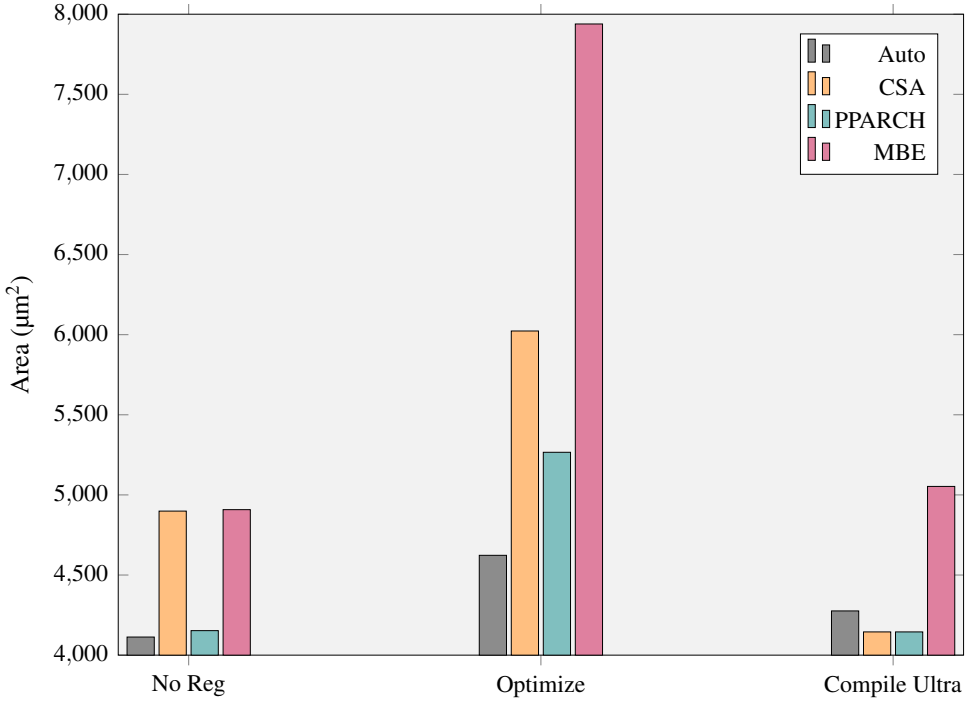

Figure 6: Area comparison between different implementations

3 Appendix

3.1 Output of Dadda Tree python script

```
--------------------------------
STEP N. : 5                 Dadda Step : 13
--------------------------------
0123456789|123456789|123456789|123456789|1234567
################################################        .0        (bits=48)
################################################        .1        (bits=48)
..############################################.        .2        (bits=45)
....########################################...        .3        (bits=41)
......####################################.....        .4        (bits=37)
........################################.......        .5        (bits=33)
..........############################.........        .6        (bits=29)
............########################...........        .7        (bits=25)
..............####################.............        .8        (bits=21)
................################...............        .9        (bits=17)
..................############.................        .10        (bits=13)
....................########...................        .11        (bits=9)
......................######.....................        .12        (bits=6)

[2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10, 11, 11, 12, 12, 13, 13,
 ↪  13, 13, 13, 13, 12, 11, 11, 10, 10, 9, 9, 8, 8, 7, 7, 6, 6, 5, 5, 4, 4, 3,
 ↪  3, 2]

--------------------------------
STEP N. : 4                 Dadda Step : 9
--------------------------------
0123456789|123456789|123456789|123456789|1234567
################################################        .0        (bits=48)
################################################        .1        (bits=48)
..############################################.        .2        (bits=45)
....########################################...        .3        (bits=41)
......####################################.....        .4        (bits=37)
........################################.......        .5        (bits=33)
..........############################.........        .6        (bits=29)
............########################...........        .7        (bits=25)
..............####################.............        .8        (bits=22)
................................................        .9        (bits=0)
................................................        .10        (bits=0)
................................................        .11        (bits=0)
................................................        .12        (bits=0)

[2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
 ↪  9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 8, 7, 7, 6, 6, 5, 5, 4, 4, 3, 3, 2]

--------------------------------
STEP N. : 3                 Dadda Step : 6
```

```
--------------------------------
0123456789|123456789|123456789|123456789|1234567
################################################        .0        (bits=48)
################################################        .1        (bits=48)
..############################################.        .2        (bits=45)
....########################################...        .3        (bits=41)
......####################################.....        .4        (bits=37)
........################################.......        .5        (bits=34)
.............................................        .6        (bits=0)
.............................................        .7        (bits=0)
.............................................        .8        (bits=0)
.............................................        .9        (bits=0)
.............................................        .10        (bits=0)
.............................................        .11        (bits=0)
.............................................        .12        (bits=0)

[2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
 ↪  6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 4, 4, 3, 3, 2]

--------------------------------
STEP N. : 2                Dadda Step : 4
--------------------------------
0123456789|123456789|123456789|123456789|1234567
################################################        .0        (bits=48)
################################################        .1        (bits=48)
..############################################.        .2        (bits=45)
....########################################..        .3        (bits=42)
.............................................        .4        (bits=0)
.............................................        .5        (bits=0)
.............................................        .6        (bits=0)
.............................................        .7        (bits=0)
.............................................        .8        (bits=0)
.............................................        .9        (bits=0)
.............................................        .10        (bits=0)
.............................................        .11        (bits=0)
.............................................        .12        (bits=0)

[2, 2, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 ↪  4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 2]

--------------------------------
STEP N. : 1                Dadda Step : 3
--------------------------------
0123456789|123456789|123456789|123456789|1234567
################################################        .0        (bits=48)
################################################        .1        (bits=48)
..############################################        .2        (bits=46)
.............................................        .3        (bits=0)
.............................................        .4        (bits=0)
.............................................        .5        (bits=0)
```

```
.............................................        .6        (bits=0)
.............................................        .7        (bits=0)
.............................................        .8        (bits=0)
.............................................        .9        (bits=0)
.............................................        .10        (bits=0)
.............................................        .11        (bits=0)
.............................................        .12        (bits=0)

[2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
↪  3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]

--------------------------------
STEP N. : 0                 Dadda Step : 2
--------------------------------
0123456789|123456789|123456789|123456789|1234567
################################################        .0        (bits=48)
################################################        .1        (bits=48)
.............................................        .2        (bits=0)
.............................................        .3        (bits=0)
.............................................        .4        (bits=0)
.............................................        .5        (bits=0)
.............................................        .6        (bits=0)
.............................................        .7        (bits=0)
.............................................        .8        (bits=0)
.............................................        .9        (bits=0)
.............................................        .10        (bits=0)
.............................................        .11        (bits=0)
.............................................        .12        (bits=0)

[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
↪  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```