Politecnico di Torino

1859

Integrated Systems Architecture

A.Y. 2025/2026

Laboratory 2

Digital arithmetic and logic synthesis strategies

Delivery instructions

A template for the homework report is available on the *Portale della Didattica*. Students are required to follow the **provided structure** and ensure that the report is **self-contained**, **complete**, **clear**, and **accurate**. **Diagrams**, **graphs**, and **figures** that facilitate the understanding of the design must be included in the report.

Each project prepared for this course must be submitted by uploading it to the *Portale della Didattica*, section *Elaborati*. Each member of the group must submit the same deliverable.

The deliverable must consist of a `.zip` archive containing all the **source files**, the **scripts**, and the **main report**. The `.zip` must be called `isa<nn>_lab2.zip`, where `<nn>` represents the group number (for example, group 8 should name the file `isa08_lab2.zip`).

Work folders, temporary files, and intermediate files must not be included. Only the **input** and **output files** are required, together with **clear instructions** on how to reproduce the reported results using the uploaded material.

Delivery deadline: December 21 at 23:59

1 Introduction

The aim of this lab is to deal with digital arithmetic issues. This is done by simulating and synthesizing a Floating Point Unit (FPU) under different conditions.

As you could appreciate during the first lab, modern logic synthesizers (such as Synopsys Design Compiler) handle the behavioural description of adders and multipliers by directly inferring the component in the netlist.

For Design Compiler, this is possible due to the availability of DesignWare Library (DW), which is basically a collection of ready-to-use blocks. In particular, the `report_resources` command shows the arithmetic resources employed in your design and the corresponding architecture.

As detailed in the documentation, DW contains among the others a parametric multiplier (*DW02_mult*), which can be implemented as carry-save (csa) or parallel-prefix (pparch).

You can force Design Compiler to use a specific architecture for each element (adder, multiplier, ...) in your design by using the `set_implementation` command. After specifying the clock constraints, before issuing the `compile` command, you can specify the implementation of each cell.

1.1 Dealing with hierarchy

If your design has some hierarchical organization, you have to discover the hierarchy to customize the `set_implementation` command.

To ease this task you can first flatten the hierarchy and then force Design Compiler to use the DW component you want.

example Flatten the hierarchy and specify the architecture (e.g. csa) for all the multipliers:

```
ungroup -all -flatten
set_implementation DW02_mult/csa [find cell *mult*]
```

Then, you can issue the `compile` command and check the result with `report_resources`.

1.2 Pipelining

Instead of writing by hand a pipelined multiplier or adder, you can exploit Design Compiler capability of optimizing register position (retiming). A simple example is the design of a pipelined multiplier. You can describe the multiplier with the behavioural operator "*" and place a chain of registers at the output. Then, you can synthesize the design with the `compile` command and then force Design Compiler to re-compile the design performing retiming by using the `optimize_registers` command.

1.3 Optimization

Several optimization tools are automatically enabled by using the *ultra* mode in Design Compiler. The *ultra* mode is automatically enabled by issuing the `compile_ultra` command instead of `compile`.

2 FPU model

Download from the *Portale della didattica* the "Floating Point Unit lite" (cvfpu_lite) project. This project is a simplified version of the *cvfpu* project developed by the OpenHW Group[1]. It is an FPU, implemented in SystemVerilog, fully compliant with the IEEE 754-2008 standard[2]. Furthermore, it supports different floating-point formats, including the brain floating-point 16-bit (bfloat16) one[3].

Unpack the project. The package contains the source files (`src`) and the testbench files (`tb`). As you can see, you have two top files for the testbench: *tb_fpnew_top_rtl.sv* and *tb_fpnew_top_net.sv.* The reason is that user-defined types at the top module interface in the RTL description are translated into standard arrays in the netlist.

The text file prj.txt gives you the hierarchy order of the files (source and testbench), so it can be useful to derive scripts for both simulation and synthesis.

[1] `https://github.com/openhwgroup/cvfpu`
[2] `https://en.wikipedia.org/wiki/IEEE_754`
[3] `https://en.wikipedia.org/wiki/Bfloat16_floating-point_format`

2.1 Simulation

Verify the model by testing it with a simple testbench to perform at least 10 different bfloat16 floating-point multiplications.

You can extend the testbench already provided in cvfpu_lite project. The package also includes a simple C++ program (`fp16_num.cpp`) that computes the product of two bfloat16 floating-point numbers and prints the multiplicand, the multiplier, and the result, also showing their binary representation in compliance with the Brain Floating Point format.

Note: the program relies on the *flexfloat* library[4], which requires *CMake 3.1 or higher* and *GCC 7.1 or higher*. Thus, it cannot be built on the server. Nevertheless, you have the sources and a statically compiled executable (`fp16_num`) that can be executed directly on the server (e.g., `./fp16_num 7.0 25.0`).

During compilation, you may observe warnings concerning potential conflicts with `always_comb` and `always_latch` variables. These warnings can be ignored, as additional checks are performed later when invoking `vsim`.

Assignment

1. Select at least 10 different bfloat16 floating-point multiplications and compute them using the executable `fp16_num`. The obtained results will serve as reference test values for the FPU.
2. Verify the model by running a simple testbench that performs the selected 10 (or more) multiplications, and include a snapshot of the simulation results.

2.2 Synthesis strategies

You need to perform the following synthesis:

1. Synthesize with `compile`. Find the maximum frequency and the area. Verify the netlist behaviour via simulation.
2. Repeat the previous step issuing the `optimize_registers` command after `compile`. Find the maximum frequency and the area. Verify the netlist behaviour via simulation.
3. Repeat the previous step issuing only the `compile_ultra` command. Find the maximum frequency and the area. Verify the netlist behaviour via simulation.
4. Force Design Compiler to flatten the hierarchy and to implement the Significands multiplier (Mantissa multiplier in fpnew_fma.sv[5]) as a CSA multiplier. Find the maximum frequency and the area with the commands `compile` and `optimize_registers`. Verify the netlist behaviour via simulation.
5. Repeat the previous step by forcing 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 behaviour via simulation.

Note: the maximum frequency corresponds to the minimum period with slack met and equal to zero.

Assignment

1. For all five syntheses, include a snapshot of the netlist simulation result.
2. Build a table summarizing the results in terms of period, frequency and area.
3. Explain and comments the differences between the synthesis strategies.
4. Add as an appendix in your report the full text of the `report_timing` and `report_area` commands. For the CSA and PPARCH architecture add the full text of the `report_resources` command as well.

[4] `https://github.com/oprecomp/flexfloat`

[5] In the hierarchy of the design you find the mantissa multiplier in: *gen_operation_group[0]* $\rightarrow$ *i_opgroup_block* $\rightarrow$ *gen_parallel_slices[2]* $\rightarrow$ *active_format* $\rightarrow$ *i_fmt_slice* $\rightarrow$ *gen_num_lanes[0]* $\rightarrow$ *active_lane* $\rightarrow$ *lane_instance* $\rightarrow$ *i_fma*.

3 R4-MBE multiplier implementation

Design in SystemVerilog a Radix-4 Modified Booth Encoder (R4-MBE) multiplier for unsigned data and use it as the Mantissa multiplier in `fpnew_fma.sv`.

Partial products must be generated without adders or subtracters (see Appendix A). The adder plane must rely on a Wallace tree (see Appendix B). Sign extension bits in the Wallace tree must be simplified to reduce the number of half adders and full adders [1].

Once the design is completed, you must:

1. Simulate the multiplier first as a stand-alone component, and then include it in the FPU as the Mantissa multiplier.
2. Synthesize the FPU (including your multiplier) using `compile_ultra`. Determine the maximum frequency and the area. Verify the netlist behaviour via simulation.

Assignment

1. Report the block diagram of the R4-MBE.
2. Report the Wallace tree.
3. Report a snapshot of the simulation of the R4-MBE multiplier as a stand-alone component.
4. Report a snapshot of the simulation of the R4-MBE multiplier included in the FPU as the Mantissa multiplier.
5. Report the period, the maximum frequency and the area of the modified FPU.
6. Compare and explain the differences with the syntheses performed in Section 2.
7. Add as an appendix in your report the full text of the `report_timing` and `report_area` commands.

Appendix

A The Modified Booth Encoder multiplier

A.1 Introduction

In order to explain how the Modified Booth Encoder (MBE) multiplier works, it is convenient to start from the Radix-2 multipliers. Let $\mathbf{a}$ and $\mathbf{b}$ be two unsigned values, each represented with n bits:

$$\mathbf{a} = \sum_{i=0}^{n-1} a_i 2^i, \tag{1}$$

$$\mathbf{b} = \sum_{j=0}^{n-1} b_j 2^j. \tag{2}$$

Let $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$, then $\mathbf{c}$ is represented with $2n$ bits. Stemming from (1) and (2) we can write $\mathbf{c}$ as:

$$\mathbf{c} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j 2^{i+j}. \tag{3}$$

As it can be observed, the Radix-2 solution produces partial products which are in the form $p_j = a \cdot b_j$ or

$$p_j = \begin{cases} 0 & \text{if } \overline{b}_j \\ a & \text{if } b_j \end{cases} \tag{4}$$

where $a = [a_{n-1}a_{n-2}\dots a_1 a_0]$.

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

Table 1: Modified Booth Encoding

A.2 Modified Booth Enconding (MBE)

MBE is an extension of the Radix-2 approach; namely, instead of considering the multiplier on a bit-by-bit basis, more bits are analyzed simultaneously. Usually, MBE is a Radix-4 approach; namely, it produces half partial products with respect to the Radix-2 solution. This is achieved by dividing the multiplier in 3 bit slices (with $b_{-1} = 0$), where two consecutive slices feature a 1-bit overlap.

If n is odd, the multiplier must be sign extended to have "complete" triplets of bits. Then, each triplet of bits is exploited to encode the multiplicand according to Table 1. As a consequence, the expression describing partial products, which can be derived from direct inspection of Table 1, is more complex in MBE than in Radix-2 solutions, namely $p_j = (b_{2j+1} \oplus q_j) + b_{2j+1}$, where

$$q_j = \begin{cases} 0 & \textbf{if } \left(\overline{b_{2j} \oplus b_{2j-1}}\right)\left(\overline{b_{2j+1} \oplus b_{2j}}\right) \\ a & \textbf{if } b_{2j} \oplus b_{2j-1} \\ 2a & \textbf{if } \left(\overline{b_{2j} \oplus b_{2j-1}}\right)\left(b_{2j+1} \oplus b_{2j}\right) \end{cases} \tag{5}$$

A.3 Adding partial products

A general scheme showing how partial products are added in an MBE-based multiplier is depicted in Figure 1.

The coverage of the dots can be made with any suited structure, including Wallace tree, Dadda tree, etc. As it can be observed, sign extension is needed to correctly add partial products. Unfortunately, this requires adders to properly cover all the dots.

A simple and effective technique to reduce the number of adders required for covering partial product sign extension dots is presented in [2] and summarized in `Bewick_AppendixA.pdf`, which is available on *Portale della didattica.*

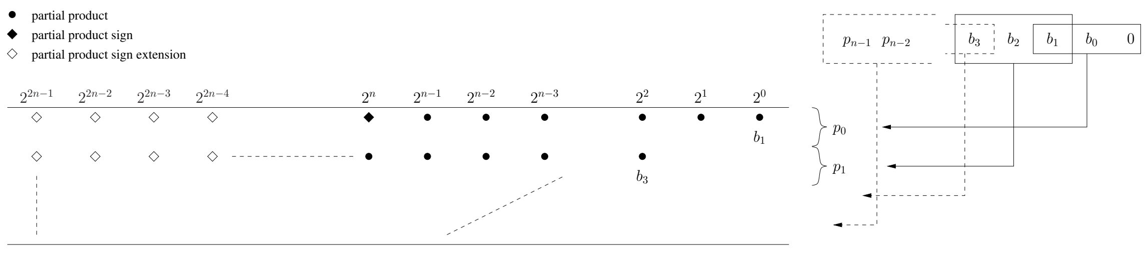

Figure 1: General scheme of a parallel MBE-based multiplier

B About Wallace and Dadda trees

Wallace and Dadda trees are famous structures for adding multiple operands. In both cases, the idea is to exploit the so-called partial addition by columns. In the case of multipliers, the terms to be added together come from the partial product generation. The first step to design Wallace or Dadda trees is to align data so that partial terms with the same weight are in the same column.

Then, partial addition by columns resorts to counters, namely elements which are able to count the number of ones in each column. Simple circuits for counters are Half-Adders (HAs) and Full-Adders (FAs). An HA (FA) compresses two (three) bits with weight k to one bit of weight k and one bit of weight $k+1$. As it can be observed, FAs allow for a compression ratio of at most 3/2. As a consequence, starting from a structure with n operands, several levels of HA/FA are needed to compress up to two operands. Finally, when the tree has reduced the operands to two operands, they are added together with a fast two-operand adder.

Let us concentrate on the reduction tree. The number of required levels L and the maximum number of elements at each level can be obtained as follows:

- The maximum number of elements at the last level (before the fast adder) is two:

$$l_0 = 2. \tag{6}$$

- The maximum number of elements at level one is:

$$l_1 = \left\lfloor \frac{3}{2} l_0 \right\rfloor = 3. \tag{7}$$

- Thus, in general at level j the maximum number of elements is:

$$l_j = \left\lfloor \frac{3}{2} l_{j-1} \right\rfloor . \tag{8}$$

- The number of required levels can be obtained by iterating (8) and the stopping condition is $j \geq n$.

As an example consider an 8×8 bit multiplier. Partial product alignment is shown with dot notation in the left upper part of Figure 2.

The maximum number of operands to be added together in a column is $n = 8$. Through (8) we obtain the sequence:

$$l_0 = 2 \quad l_1 = 3 \quad l_2 = 4 \quad l_3 = 6 \quad l_4 = 9. \tag{9}$$

As a consequence, we need five levels to compress eight operands to two operands by the means of HAs and FAs. The maximum number of operands at each level is given in (9).

Thus, given this structure, Wallace and Dadda give the rules to allocate HAs and FAs at each level. Wallace allocation is As-Soon-As-Possible (ASAP), meaning that at each level we divide dots in three-row-wide stripes and we fill each stripe with the maximum possible amount of FAs and HAs. Stripes made of only one or two operands are forwarded to the next level. The complete tree for $n = 8$ is shown in the left part of Figure 2.

Dadda allocation is As-Late-As-Possible (ALAP), meaning that at each level j we allocate the minimum number of FAs and HAs such that the maximum number of operands at the next level $(j-1)$ is the one in (9). Thus, as shown in the top right part of Figure 2, at level $j = 4$ we have eight operands, and at level $j = 3$ we must have no more than six operands ($l_3 = 6$). As a consequence, the first six columns do not require compression. The seventh column has seven elements, thus, with one HA, we compress it to six elements. The eighth column has eight elements, but the HA in column seven added one element. Thus, we have to compress nine elements to six. One FA compresses three elements to one (reduction by 2) and one HA compresses two elements to one (reduction by 1). Similarly, the ninth column has seven elements but the eighth column added two elements (one from the FA and one from the HA). As a consequence, the ninth column must compress nine elements to six. This can be done with one FA and one HA. The same idea is extended to the other columns and to the other levels up to $l_0 = 2$, as shown in the left part of Figure 2.

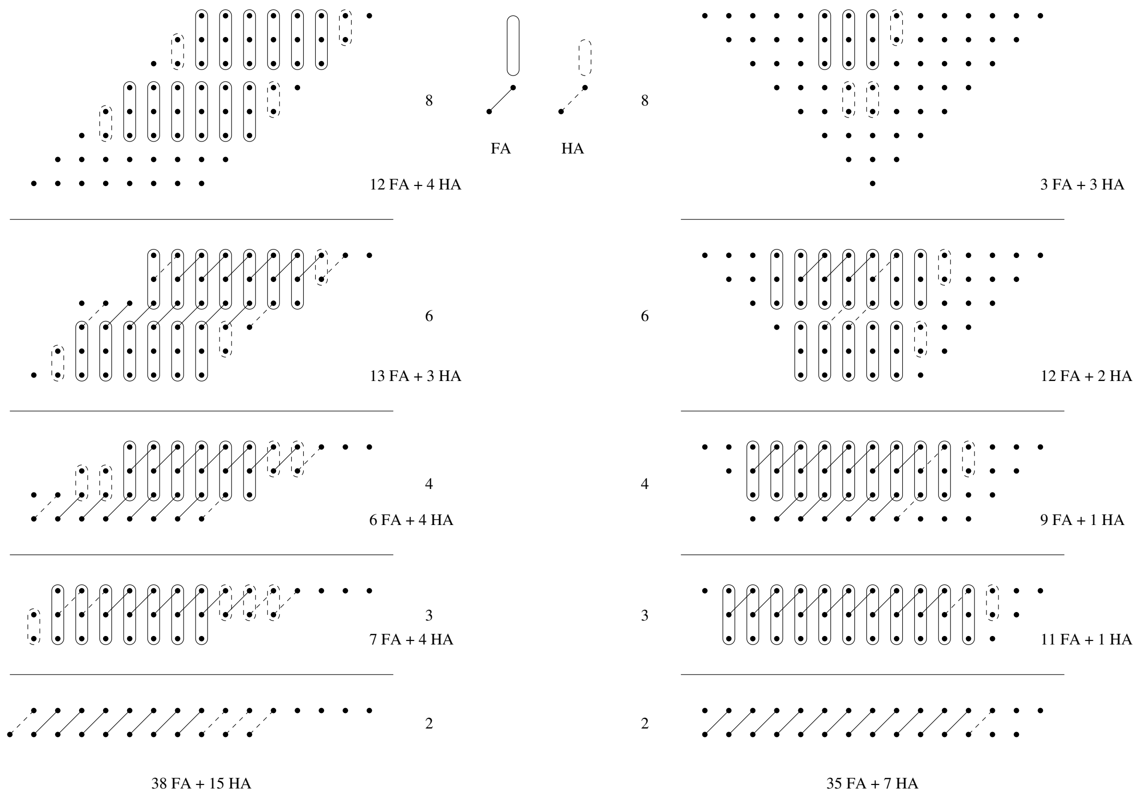

Figure 2: Wallace and Dadda trees for an 8×8 bit multiplier.

References

[1] G.W. Bewick. *Fast multiplication: algorithms and implementation - Appendix A - Sign Extension in Booth Multiplier*. PhD thesis, Stanford, 1994.

[2] M. Roorda. Method to reduce the sign bit extension in a multiplier that uses the modified booth algorithm. *Electronics Letters*, 1986.