Integrated Systems Architectures

Lab 2: digital arithmetic

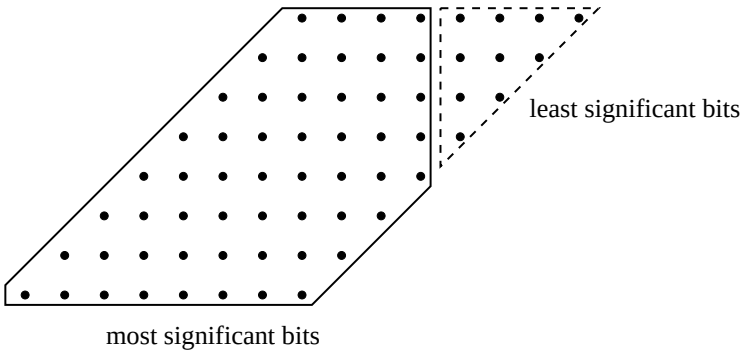

Figure 1

The aim of this lab is to deal with digital arithmetic issues.

1 Digital arithmetic and logic synthesizers

1.1 Introduction and background

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 Design Ware (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. To see the complete Design Compiler documentation you have to source the initialization script:
`source /software/scripts/init_synopsys`
and then type *sold*, which stands for Synopsys On-Line Documentation.

As detailed in the documentation, Design Ware contains among the others

- a parametric adder (*DW01_add*), which can be implemented as ripple-carry (rpl), carry-look-ahead (cla) or parallel-prefix (pparch);
- a parametric multiplier (*DW02_mult*), which can be implemented as carry-save (csa) or parallel-prefix (pparch).

For further details see
/software/synopsys/sold_current/doc/online/dw_ip/doc/dwf/intro.pdf.

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. You can get the name of each cell from the report generated by the `report_resources` command. If you want to specify one architecture for all the adders you can use the `find` command.

example Specify one architecture (e.g. ripple-carry) for adder cell `add_124`:
`set_implementation DW01_add/rpl add_124`

example Specify one architecture (e.g. ripple-carry) for all the adders:
`set_implementation DW01_add/rpl [find cell *add_*]`

1.1.1 Dealing with hierarchy

If your desing has some hierarchical organization you have to discover the hierarchy to customize the `set_implementation` command. This can be done by checking the `report_resources` report. Sometimes this can be difficult so you can first flatten the hierarchy and then force Design Compiler to use a DW component.

example Flatten the hierarchy and specify one architecture (e.g. ripple-carry) for all the adders:

```
ungroup -all -flatten
set_implementation DW01_add/rpl [find cell *add_*]
```

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 force Design Compiler to re-compile the design performing retiming by using the `optimize_registers` command. During this operation input/output registers might be moved by the tool, so increasing the input/output delay. You can keep input/output registers fixed by using the `set_dont_touch` clause. In order to make clear which registers are input/output and which not, it is recommended to use good naming conventions, e.g. use the suffix *_in_reg* for input registers and *_out_reg* for output registers. In this case you can keep input/output registers fixed by issuing:

```
compile
set_dont_touch *_in_reg
set_dont_touch *_out_reg
optimize_registers
```

1.3 Optimization

Several optimization tools are automatically enabled by using the *ultra* mode in Design Compiler. To enable *ultra* mode you have to add:

```
set_ultra_optimization true
```

before issuing the `analyze` command. Then, instead of using the `compile` command use `compile_ultra`.

1.4 Assignment

Use the VHDL of the last architecture you developed for lab 1 and try the following. **Note: do not instantiate Design Ware components in your design, use Design Compiler commands to infer them instead.**

1. Force Design Compiler to use Design Ware adders and multipliers for all the cells of your design (**Note: use the `compile` command**). Show the obtained results trying all the possible combinations of DW adders and multipliers and forcing Design Compiler to achieve the maximum clock frequency. You have to show the results by getting the relevant information from the reports generated by the `report_resources`, `report_timing` and `report_area` commands. **Verify the correct behaviour of the whole design through simulation (as in lab 1).**
2. Improve the performance of your design by adding pipeline registers to the multipliers. **Note: do not force Design Compiler to infer a specific DW arithmetic block**, leave it free to choose the best solution with the `compile_ultra` command. Use the `optimize_registers` command to exploit retiming and find the maximum clock frequency. **Verify the correct behaviour of the whole design through simulation (as in lab 1). Note: FIR filters do not give any particular problem even**

with deeply-pipelined multipliers. On the other hand, pipelining in the auto-regressive part of IIR filters may need adoption of look-ahead techniques.

3. Challenge: can you do better than Design Compiler ? Design a Modified-Boot-Encoding (MBE) based multiplier for 2's complement data exploiting Roorda's approach [1]. The adder plane must rely on a Dadda-tree. Then use the new multiplier, referred to as *version 1* instead of the behavioural operator "*". **Note: Verify via simulation the correct behaviour of both multiplier and whole filter circuit and show the results.**

4. Idea: you can do better by reducing the precision. Modify (if needed) your digital filter such that only the most significant part of the multiplication is used (see example in Fig. 1). Then, remove all the adders related to the 6 least significant bits. This solutions will be referred to as *version 2*. Show via simulation the error you obtain on the result.

5. For the two versions of the multiplier defined above: synthesize the complete filter architecture with Synopsys Design Compiler, find the maximum clock frequency and compare/discuss the obtained results.

6. Further development: describe in VHDL the approximate compressors proposed in [2] and propose a good blend of correct and approximated compressors for your MBE-Dadda multiplier (*version 3*). You can find a copy of each paper on "Portale della didattica".

   (a) Prepare a testbench and simulate the design. The testbench should highlight the specific behaviour of the circuit. **Note: verify the correct behaviour of the circuit via extensive simulations.**

   (b) Synthesize the circuit with Synopsys Design Compiler. Find the maximum frequency and the corresponding area.

   (c) Modify your digital filter (lab 1) to include instances of the arithmetic circuit you designed. **Note: verify the correct behaviour of the whole architecture via extensive simulations.** Finally, synthesize the complete architecture with Synopsys Design Compiler, find the maximum clock frequency and compare/discuss the obtained results.

References

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

[2] P. Yin, C. Wang, W. Liu, E. E. Swartzlander Jr., and F. Lombardi. Designs of approximate floating-point multipliers with variable accuracy for error-tolerant applications. *Springer Journal of Signal Processing Systems*, 2018.