

Timing- and Power-Driven ALU Design Training Using Spreadsheet-Based Arithmetic Exploration

Per Larsson-Edefors

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
e-mail: perla@chalmers.se

Kjell Jeppson

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden
e-mail: jeppson@chalmers.se

Abstract—We describe master-level design training that combines ALU design exercises based on commercial synthesis tools and arithmetic explorations based on spreadsheets. Despite its limited complexity, the ALU has a few important properties that make it suitable for our training; 1) the ALU subcircuits are diverse and contain both short and long timing paths, 2) timing-driven design is called for, since the ALU is a performance bottleneck, and 3) the ALU is continuously used, making power dissipation an important design parameter. After enforcing strict timing constraints during synthesis of the ALU, the students need to reconsider how to implement the arithmetic block, which initially is too slow. Here, performing arithmetic explorations inside an innovative spreadsheet environment helps to visualize circuit implementation tradeoffs. The final phase in the design training focuses on power analysis and demonstrates that the choice of timing constraint impacts power dissipation.

I. INTRODUCTION

The implementation of an electronic system under constraints on timing, power dissipation, area, etc. typically involves several different tasks which need to be carefully planned and prioritized. As the implementation flow reaches a certain stage it may suddenly, despite the planning, become clear that the timing goal is not met. While the designer with the help of the Electronic Design Automation (EDA) tools may be able to resolve this timing problem by simply resizing the gates inside the critical circuit, the problem can be more challenging. For example, the designer may have to replace a slow block with one that has a different, inherently faster microarchitecture, and this may force the designer to make a detour and revisit early implementation phases to efficiently reconcile the required changes. In this example, the designer is becoming agonizingly aware of a common problem known as timing closure.

If timing were the only important implementation parameter the designer could consistently use fast logic and circuit solutions throughout the implementation. This is, however, not the case. In general, fast circuits and microarchitectures dissipate more power than slow ones. Since power dissipation is important, the designer needs to implement the electronic system with the slowest and least complex circuits that still meet the system's timing goal. The need for power-efficient

The authors want to thank the students who have given constructive feedback on the ALU design exercises during 2007-2013.

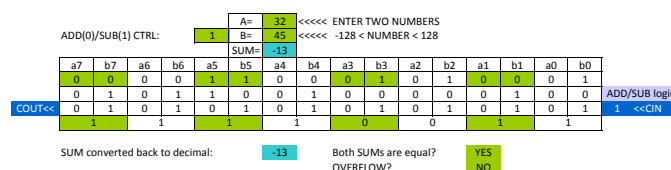
circuits makes the implementation procedure complex since the designer continuously has to respect two conflicting requirements: Using small and simple circuits for power efficiency, or large and complex circuits for timing efficiency.

The university training of design engineers for electronics and electronic system needs to address not only the circuit and logic structures that deliver performance and power efficiency, but also methods for design and exploration. This paper describes a series of design exercises, which target the design, implementation and verification of a processor Arithmetic Logic Unit (ALU), using state-of-the-art EDA tools, explicitly considering methodology for timing- and power-driven design.

II. SPREADSHEET-BASED DESIGN EXPLORATION

The ALU design exercises (Sec. III) are preceded by a set of adder design exercises in which students learn adder basics. These adder design exercises are performed within the framework of the *Introduction to Integrated Circuit Design (IICD)* course. Here, an exploration environment is set up in Excel, in which students can test different arithmetic implementations in the context of an 8-bit datapath.

The spreadsheet format has two properties that makes it a good learning tool: First, the cell organization, in which each cell can be made to represent a standard cell in the layout, and the row height, which represents the standard-cell pitch, give a feel for the size of the adder design in the CMOS technology of choice. Second, the inherent logical AND/OR functions give the student a chance to instantly check the correctness of the design and the principles behind it [1].



ADD(SUB) CTRL:																A=	32	<<<<< ENTER TWO NUMBERS	
																B=	45	<<<<< -128 < NUMBER < 128	
																SUM=	-13		
a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0				
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	ADD/SUB logic			
0	1	0	1	1	0	0	1	0	0	0	0	0	1	0	0	<<<<<CIN			
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1 <<<<<CIN			
SUM converted back to decimal:																-13	Both SUMs are equal?		
																	OVERFLOW?		
																	YES		
																	NO		

Fig. 1. Ripple-carry adder spreadsheet design template.

The 8-bit ripple-carry adder design template shown in Fig. 1 is given to the students to implement the ADD/SUB logic, the ripple-carry logic and the SUM logic. As an example, the ADD/SUB logic for bit B_0 can be written as

$= \text{IF}(\text{SUB}; \text{NOT}(B0); B0) * 1$, where SUB is the field to the right of ADD(0)/SUB(1) CTRL in Fig. 1. Assume two decimal numbers A and B that are to be added or subtracted. A and B are converted to binary numbers and placed in the A7:0 and B7:0 positions, after which the spreadsheet adder performs an addition or subtraction. The result from the spreadsheet adder is compared to $A + B$ (if SUB = 0) or $A - B$ (if SUB = 1). If equal, the adder design is correct for the chosen numbers and the student is given a green go-ahead YES.

An early exercise performed in the IICD course is one where the students design and implement an 8-bit ripple-carry adder by using the full-adder cell found in the reference 65-nm CMOS cell library. The full-adder functionality is added to the spreadsheet using its built-in logic functions and, by means of the click-and-drag function, eight instances of the full-adder cell will be made to form a ripple-carry Iterative Logic Array (ILA). Ideas for how to implement the logic expressions of a full adder can be found in most textbooks (Fig. 2).

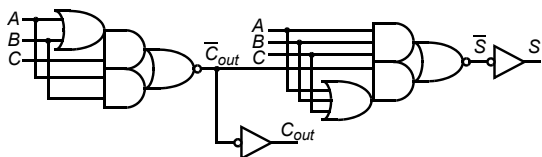


Fig. 2. Full adder for ripple-carry operation (reproduced from [2]).

Parallel to this design exercise, there is a separate set of lab exercises in the IICD course where the emphasis is on circuit design aspects of adders; including the design of pull-up and pull-down MOSFET networks to implement carry functionality, and the introduction to commercial circuit-level EDA tools for schematic capture, layout, design rule checking, and layout-vs-schematic verification. The understanding of schematic and layout issues and their impact on performance and power dissipation is pivotal for the ensuing ALU design exercises, which take place at the standard-cell level.

III. ALU DESIGN EXERCISES

We will describe the flow of the ALU design exercises arranged in our computer lab, inside the *Methods for Electronic System Design and Verification* course. In terms of the spreadsheet-based design exploration initially described in Sec. II, there is a focus on the design of the adder circuit as this can be implemented in very different ways, demonstrating to the students the virtue of making design explorations.

A. Initial Processor ALU Specification

Since it constitutes the core of nearly all electronic systems, a processor ALU is an interesting block to consider when teaching embedded electronics. There are three reasons as to why designing an ALU is a suitable challenge in the context of this paper: First, since the ALU is a likely performance bottleneck of a processor pipeline, timing considerations are critically important. Second, since the ALU is active on all clock cycles, power dissipation considerations too are critically important. Third, although a block of limited complexity, the

ALU has a relatively heterogeneous collection of functions (arithmetic, logic, shift, and multiplexing), which makes both timing and power considerations quite complex.

The following are the initial specifications for the ALU that the students are assigned to develop:

- 1) The following OP codes are to be used in order to comply with a 32-bit MIPS-like processor [3]:

```
0000: add A+B (signed)
0001: add A+B (unsigned)
0010: sub A-B (signed)
0011: sub A-B (unsigned)
0100: bitwise AND
0101: bitwise OR
0110: bitwise NOR
0111: bitwise XOR
1000: shift left
1010: shift right (logical)
1011: shift right (arithmetical, signed)
1110: SLT (Set on Less Than)
1111: SLTU (Set on Less Than Unsigned)
```

- 2) The interface to the processor pipeline in which the ALU is integrated is strictly defined as:

```
entity ALU is
  port(
    clk      : in std_logic;
    reset    : in std_logic;
    Ain      : in std_logic_vector(31 downto 0);
    Bin      : in std_logic_vector(31 downto 0);
    OPin     : in std_logic_vector(3 downto 0);
    Outs     : out std_logic_vector(31 downto 0);
  end ALU;
```

- 3) To simplify the synthesis phase, the ALU should have registers on both the input and the output.
- 4) Initially the students should implement the arithmetic circuit using a ripple-carry adder (for reasons that later will become obvious).

B. ALU and Test Bench Coding

The students initially develop an ALU block schematic from the specifications in Sec. III-A. The visual representation of a block schematic, such as the one in Fig. 3, is important to guide the VHDL coding. Without this proper preparation, our experience is that the resulting VHDL code, due to being written in an improvised manner, tends to be of poor quality.

When the ALU code has been developed, a test bench is developed. Based on test vectors for Ain , Bin , and $OPin$ stored in files, the test bench allows the ALU to be instantiated and input vectors to be applied, after which the results of the ALU operations are compared to reference vectors for $Outs$.

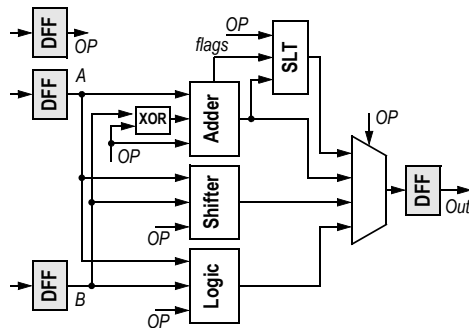


Fig. 3. ALU block schematic.

C. Timing-Driven Synthesis

Once the ALU has been functionally verified, using logic simulation for a set of diverse test vectors obtained using tools from our research frameworks [4], the students begin synthesizing the ALU to a CMOS cell library. The overall timing goal of the ALU depends on the technology node chosen; for a 65-nm process technology, a 1-ns timing constraint can be reasonable for the synthesis phase.

It soon turns out that it is impossible for the ripple-carry adder to sustain an ALU performance around 1 GHz. As mentioned in Sec. II, the students have studied adder structures in the previous IICD course. Thus, the fact that the delay of a ripple-carry adder depends linearly on the input word length should not come as a surprise. At this stage in the exercises, the students study the resulting netlist via the GUI of the synthesis tool and identify the logic depth for the different ALU subcircuits. This step reinforces that the ripple-carry adder structure is very bad from timing point of view.

To fulfil the timing goal of the design exercises, the students now need to revisit their ALU code and redesign the adder component. They are encouraged to consider prefix adders, such as the logarithmic-depth adders that were briefly introduced in the previous IICD course.

D. Timing-Driven Adder Considerations

Now, let us return to the spreadsheet-based design exploration of Sec. II, but with the focus moved from functionality to performance in terms of timing. What information is needed for the student to estimate the carry propagation delay along the worst case critical path, including identifying this critical path? A good start would be a unit-delay model, since only very few logic cells are used; but how should the unit delay be estimated? As students learn more about circuit design, the delay model can be gradually improved as concepts like input capacitance, output driving capability, parasitic capacitances, sizing, etc., are introduced. The unit-delay model evolves into the linear RC-delay model, the slope-dependent linear delay model, and on to nonlinear delay models and possibly also current source models stored in look-up tables.

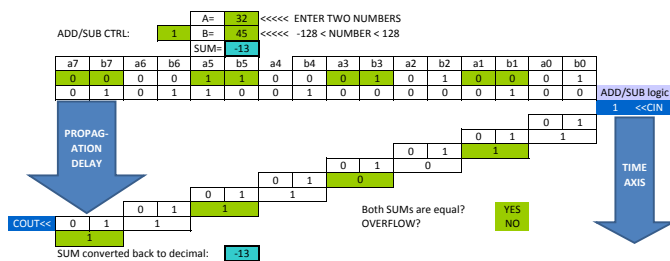


Fig. 4. Adder design template with time axis.

What can be done in the spreadsheet environment to clarify timing aspects is to include a timing axis. We then abandon the layout view of the spreadsheet template, and prepare for an understanding of the more advanced tree-adder PG networks [2]. An example of how a time axis can be included to illustrate the linear ripple-carry delay is shown in Fig. 4.

Once timing has been considered and included in the spreadsheet, students can start exploring the adder design space by playing around with parameters like sizing. Beyond sizing, *what if* they could simplify the ripple-carry cell; would that make the adder faster? What if the input data to the ripple-cell could be prepared in a way such that less complex, faster ripple-cells could be used? What if an input setup layer could be introduced in a such way that the ripple-cell delay is halved? The original 8 unit delays for the carry to ripple from the input to the output would then be reduced to a 5-unit delay, the setup delay plus 8 half-unit ripple delays. For a 32-bit adder, the reduction would be even more dramatic, from 32 unit delays to 17 unit delays.

From this starting point, the inputs to the carry logic can be prefixed by use of bit-propagate and bit-generate setup logic. Together with the dot operator implementation of the carry cell, not only are the correct bit sums produced but the adder microarchitecture is also extended with the block propagate and block generate facility.

After this design exploration exercise, where different approaches for optimizing the ripple-carry adder have been evaluated, time is now appropriate for the student to reflect over the reasons why we chose the ripple-carry approach in the first place. What if there are other solutions where the delay does not increase linearly with the input word length N , but slower? Of course, the initiated student is aware of the fact that for binary tree solutions the delay grows as $\log_2 N$. As an example, an 8-bit AND gate can be built as a 3-stage binary tree of AND2 gates, a solution that can be used for implementing the 8-bit zero detect logic, or the 8-bit equal condition when two words A and B are equal.

Actually, the same logarithmic structure is useful for the barrel shifter of the ALU. For a SHIFT7 operation, the bits do not have to ripple through seven layers of shift/no_shift muxes, but three layers of SHIFT4, SHIFT2, and SHIFT1 muxes are enough as illustrated in Fig. 5.

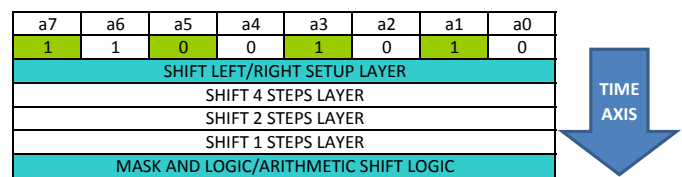


Fig. 5. Basic barrel-shifter microarchitecture with time axis.

Whether the adder can be implemented as a binary tree now depends on the availability of an idempotent and associative logic cell, where bits can be treated two and two in parallel, to generate the appropriate signals, and where the data from less significant bits can be included after the operation, instead of being available before. As it turns out, the already considered dot operator has these properties suitable for implementing a tree adder. This opens up the world of prefix tree-adders [5], like the Kogge-Stone, Han-Carlson, Sklansky, Ladner-Fischer, and Brent-Kung adders, to the student.

The students are encouraged to try the Sklansky adder

structure in their first prefix-tree adder design. As they return to the design exercises, they discover that the synthesis of the Sklansky adder means that the timing goal can be accomplished and, thus, this step closes the timing-driven synthesis.

E. Power-Driven Synthesis

Due to its long delay, the ripple-carry adder had to be discarded in the timing-driven design in Sec. III-C and Sec. III-D. But this simple adder does have some advantages and in the implementation phase when power dissipation is addressed, the students are to find this out.

Power dissipation in CMOS circuits has been introduced in the previous IICD course. The well-known switching power function is written as

$$P_{sw} = \sum_i f V_{DD}^2 (\alpha_i C_i), \quad (1)$$

in which we can find system parameters like clock rate f and supply voltage V_{DD} . The switching activity α and the switched capacitance C are given for each node i of the circuit. In these exercises, the design exploration will assume that 1) the supply voltage is constant (nominal V_{DD} is 1.2 V for this cell library), 2) $\alpha = 0.2$ for all primary inputs (A_{in} , B_{in} , and OP_{in}), and 3) the system is operated at the lower performance corner of the evaluated range, that is, 200 MHz.

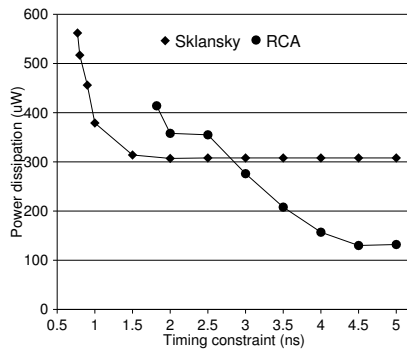


Fig. 6. Adder power dissipation as function of ALU timing constraint. 0.77 ns and 1.82 ns corresponds to the maximal performance for the Sklansky and the ripple-carry ALU, respectively. Below these constraints, the synthesis tool produces netlists with negative slack.

What is varied in this investigation is the timing constraint for synthesis; from the most relaxed one 5 ns (200 MHz) down to the constraint that corresponds to the maximal speed of each ALU type. Fig. 6 shows the result of an exploration done for two different ALUs; one based on the ripple-carry adder (RCA) and one based on the Sklansky adder. To clearly bring out the power trend for the adders, the timing constraint is applied on the whole ALU, while the power dissipation is for the adder circuit only.

The figure shows that the power dissipation of the simple ripple-carry adder is relatively low when the ALU timing constraint is relaxed. As we make the constraint stricter, the power increases rapidly because the long logic depth must be compensated for by increased gate drive strengths and costly

logic reorganizations, which lead to an increase in switched capacitance. For timing constraints slightly below 2 ns, the ripple-carry adder even dissipates more power than the more complex Sklansky adder. As it approaches its operational limit, the ripple-carry adder clearly is not a good alternative.

The students are also encouraged to make a graph showing the area for different timing constraints. As is shown in Fig. 7, the area and power dissipation trends are similar. However, since the switching activities on individual nodes downstream from the primary inputs depend strongly on the logic gates used, the heuristic algorithms employed in the timing-driven logic reorganization can impact power and area data points somewhat differently.

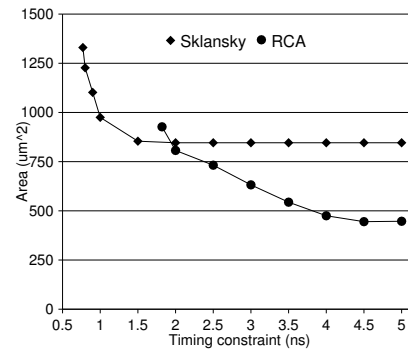


Fig. 7. Adder area as function of ALU timing constraint.

IV. CONCLUSION

We have described a design training concept in which we combine commercial EDA tools with spreadsheet-based design exploration. As the students work with the ALU design they come to the point when they need to make a decision whether to use a *simple and slow* or a *large and fast* adder for the arithmetic circuit of the ALU. At this stage, they can use a spreadsheet-based adder template (known from a previous course) to explore tradeoffs such as speed versus area. The outcomes of the exploration and the synthesis are similar; for strict performance requirements, fast circuits have to be used. However, the power analysis done at the end of the design exercises demonstrates that designing electronic systems means considering more parameters than timing: Slow circuits are attractive in systems with lower performance requirements.

REFERENCES

- [1] K. Jeppson and P. Larsson-Edefors, "Exploring Prefix-Tree Adders Using Excel Spreadsheets: Setting Up an Explorative Learning Environment," in *Proc. 2013 IEEE Int. Conf. on Microelectronic Systems Education (MSE)*, Jun. 2013, pp. 48–51.
- [2] N. Weste and D. Harris, *Integrated Circuit Design*, 4th ed. Pearson Education Inc., 2011.
- [3] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design, The Hardware/Software Interface*, 2nd ed. Morgan Kaufman Publishers Inc., 1998.
- [4] M. Sjalander and P. Larsson-Edefors, "FlexCore: Implementing an Exposed Datapath Processor," in *Proc. IEEE Int. Conf. on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XIII)*, Jul. 2013, pp. 306–313.
- [5] S. Knowles, "A Family of Adders," in *Proc. 15th IEEE Symp. on Computer Arithmetic*, 2001, pp. 277–281.