ECE 429, Fall 2012
Final Project: Design and Synthesis of Multi-Operand Adders

Dr. Erdal Oruklu

11/20/2012
Version 1.2

Project duration: *4 weeks, 11/05 – 12/07*
Final Report Due: *12/07 (Friday) Midnight Chicago time*

1 Design Objective

This project introduces VLSI design concepts including datapath circuit design, standard cell based design flow, and design validation and verification through construction of fast multi-operand adder architectures in Verilog, to be synthesized using commercial EDA tools from Synopsys and Cadence Design Systems. The intent is to show, first, how to construct a nontrivial adder hardware; second, how to make design trade-offs, e.g. performance, cost, and design time, through architectural exploration; and third, how the EDA tools transform the design implementation from higher abstraction levels, e.g. Verilog, to lower abstraction levels, e.g. layout, through cell-based design flow, what the differences are among the implementations and how to verify their correctness at the different abstraction levels.

In this project, you will be exploring multi-operand adder architectures with two different implementation methods. Specifically, you will apply redundant arithmetic operations based on *Carry Save Adders (CSA)*. You will be able observe the carry chain delay elimination provided by the CSA. An 8-bit carry-ripple adder design and its stimulus file is provided. You can use this adder for conventional adder design (i.e., apply 2-operand addition repeatedly). All the designs should be synthesized through the cell-based flow. A pre-defined performance constraint must be met after the synthesis and the correctness of the design at different abstraction levels should be verified.

This project is a group project and can be done with a single partner. For local and remote students, use of *Blackboard Collaborate software* is highly encouraged. You will be able to communicate and work on the project with your partner remotely. Please see the tutorial for this desktop sharing software on the blackboard.

A single report is required from each group. However, each student should declare their role and contribution to the project on the cover page and sign it. Signature is MANDATORY.

It is important to note that COPYING without proper CITATION, and extensive COPY from other materials including but not limited to project instructions and textbooks, will be treated as PLAGIARISM and called for DISCIPLINARY ACTION. **NEVER share your reports with others**.

2 Multi Operand Addition

Multi-operand addition is used in several algorithms including multiplication, recurrences, transforms and filters. Main approach for multi-operand addition is the use of redundant digit set adders which perform *reduction by row* on the bit array. The objective of having output in redundant representation is to reduce addition time by reducing the length of maximum carry propagation.

Reduction by row is implemented using [p:2] adders which reduce p bit-vectors to 2 bit-vectors (see Figure 1). [3:2] is a regular full-adder and also known as carry-save adder. [4:2] adder can be constructed from [3:2] adders and it can add two carry-save inputs. [5:2] and [7:2] take even more rows as inputs and reduces them to two output rows. Figure 2 shows an example with [3:2] adders a.k.a CSA. You can see that there is no delay propagation from one digit to another (Carry is saved for later). [4:2] adder block can be constructed from CSA as shown in Figure 3.

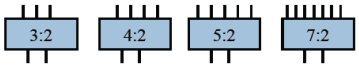

Figure 1: [p:2] Compressor/Adder blocks

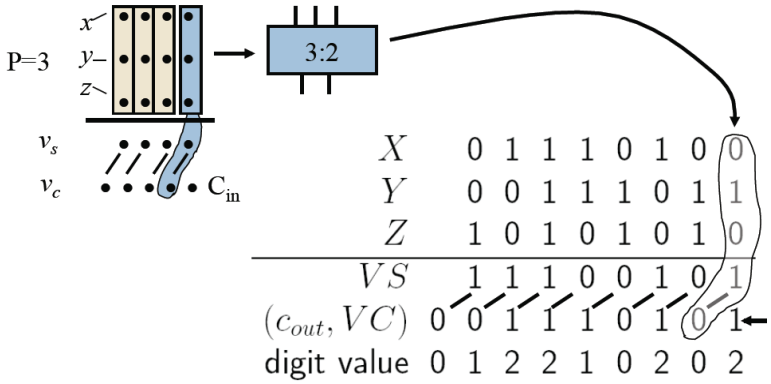

Figure 2: [3:2] Compressor/Adder example

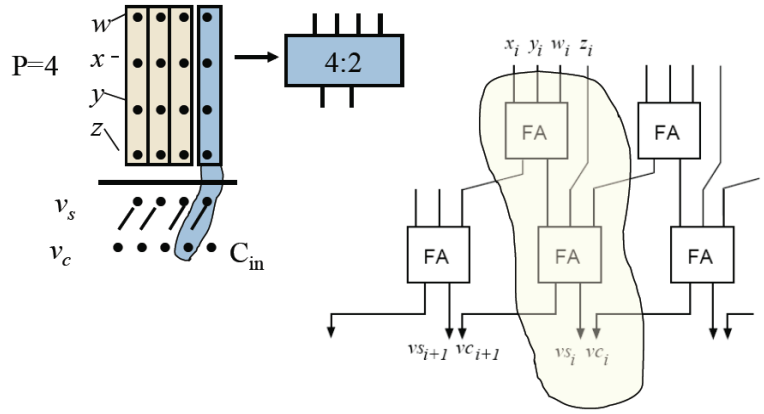

Figure 3: [4:2] Compressor/Adder block

Figure 4 illustrates the addition operation. Note that even though it looks like carry is propagated, the C_{out} form each [4:2] cell is computed directly from A and B inputs.

Figure 4: [4:2] Compressor Adder example

2.1 Word-length Extension

Consider the case in which each m operands is represented by an n-bit vector. Bit-array is then an n *by* m rectangular array and the sum bit-vector has $n + p$ bits with $p = log_2 m$. To perform the addition, the range of operands have to be extended (0-fill for unsigned) to $n+p$ bits (see Figure 5).

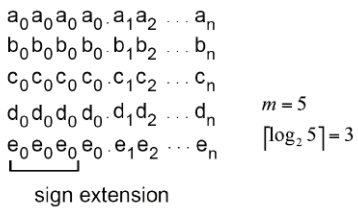

Figure 5: Word-length extension

2.2 Implementation

Reduction by Rows can be implemented in two different ways: Linear or Adder Tree architectures.

2.2.1 Linear

If $[p:2]$ adders are used, for an addition of m operands, the array consists of $[(m-2)/(p-2)]$ adders since the first adder receives p operands and the rest receives $p-2$. Delay is equal to $[(m-2)/(p-2)]t_{[p:2]}$.

Figure 6 shows a block diagram for linear implementation with [3:2] compressors. CPA at the last stage is the carry propagation adder which can be any conventional adder such as a carry ripple adder.

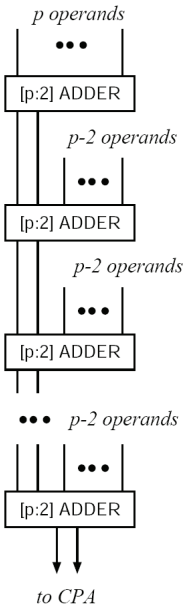

Figure 6: Linear implementation with [3:2] adders

Figure 7 shows a 4-bit 4-operand implementation using linear CSA arrays. Please pay attention to the carry bits (which have twice the weight value) that are connected to higher digit location (to the left).

2.2.2 Adder Tree

Array of adders can be organized as a tree which has fewer levels than the linear array. The number of adders required for the tree is the same as that for the linear array and equal to $[(m-2)/(p-2)]$. Figure 8 shows an example for 8-operand addition with tree architecture using [4:2] adders.

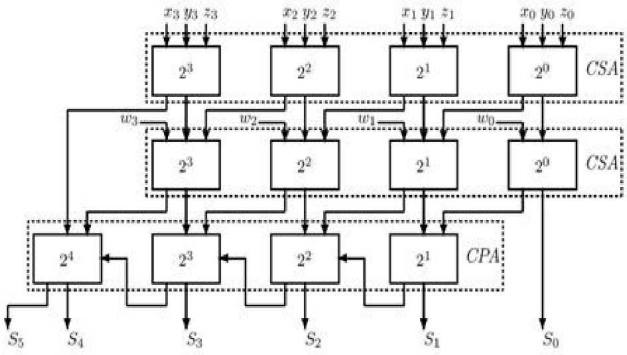

Figure 7: 4-bit 4-operand addition with CSA

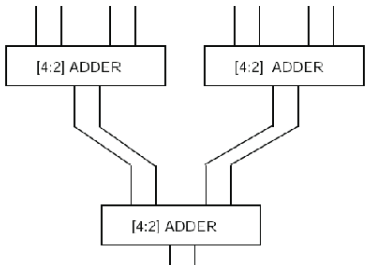

Figure 8: 8-operand addition with tree structure and [4:2] adders

3 Project Deliverables

The specific requirement of the project is the design of a **8-operand adder**. Each initial operand is assumed to be 4-bit unsigned number (i.e., [0-15]). You need to implement and compare **three** different architectures:

1. Conventional implementation using regular full adders. Simply repeat and accumulate carry ripple addition for every two-operand addition (no redundant digit set).

2. Use LINEAR implementation with CSA ([3 : 2]) adders. You need to use 7 [3:2] adders.

3. Use TREE implementation with [4:2] adders. You need to use 3 [4:2] adders. For design of the [4:2] block, refer to Figure 3.

Hint: As described earlier, although input operands are 4-bit wide, due to multiple accumulation steps, you need to extend your operand size by 3 bits ($log_2 8 = 3$). Hence use 7-bit wordlength in your design.

The key objective is to determine *which implementation is the fastest and has the fewest area*. Find and document timing and area synthesis results for each of the three implementations.

3.1 Automatic Synthesis of Adder Architectures

We follow the standard cell based design flow to synthesize the CPU. Please refer to the online tutorial (Tutorial IV) for detailed information. All adder implementations need to be done using *structural Verilog*. Pay attention to the circuit characteristics before and after P&R. Is the timing specification met? Determine the critical path delay of each synthesized design.

In order to generate area report after P&R, type 'reportGateCount -limit 0' in the encounter shell. The results will be in μm^2. For power results, type 'report_power' in the encounter shell. The results will be in mW.

3.2 Functional Validation

We must ensure that there is no bug in the adder design before synthesis. We validate the functionality of the adders by running testing programs. You will create a stimulus file similar to the provided test file for carry ripple adder (`adder8test.v`)and you should validate that the output is as expected for each implementation. Make sure to include reasonable number of input patterns for the testbench.

3.3 Timing Specification

For logic synthesis with Synopsys Design Compiler, please set the desired clock frequency to be 250Mhz. Synopsys Design Compiler will automatically pass the specification to Cadence Encounter for place & route.

3.4 Equivalence Checking

Please follow the instructions in Tutorial IV to verify the correctness of the synthesized results by equivalence checking, namely `adder8.v` and `final.v` should be equivalent.

4 Bonus Design Problem

As an option, you can also implement a 10-operand adder with [5:2] adder/compressor. The [5:2] adder block is shown in Figure 9. Use *Tree* architecture to implement this addition. Same conditions apply; that is the input operands are 4-bit unsigned. Implementation will be done using structural Verilog code only. You need to provide a testbench and verify the functionality of the 10-operand adder. The bonus part will earn you up to 20 more points added to overall project grade.

5 Project Evaluation

The final project will contribute to 14% of the course grade, plus the bonus which contributes an additional 2%.

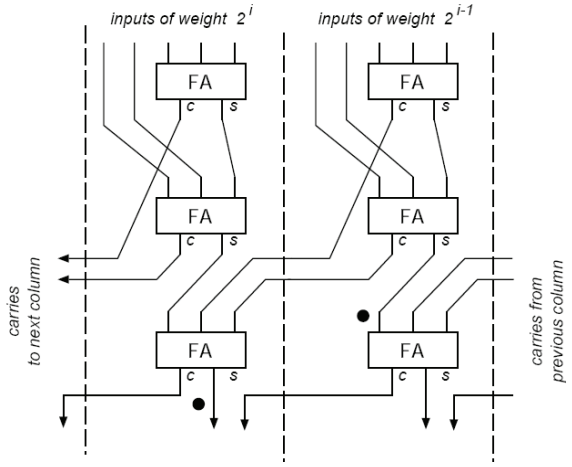

Figure 9: [5:2] adder block and operation

Results and Design	Final Report	Bonus Problem
70	70	20

5.1 Final Report

You are required to submit a stand-alone report to summarize the whole final project. This report should include the necessary results and analysis as required in the previous sections, and be readable without reference to other materials, including but not limited to the project instructions, the textbook, and the progress report.

The report should be limited to 20 pages at most and be legible when printing on letter-size papers. Please use common sense to format your report, e.g. report with small fonts/figures will not be graded. Your codes/screen shots/results can be attached as the appendix which will not count toward the 20-pages limit. However, you should discuss them within the 20-pages limit.

All the writings, results, codes, and screen shots should be by yourself. COPY without proper CITATION, and extensive COPY from other materials including but not limited to project instructions and textbooks, will be treated as PLAGIARISM and called for DISCIPLINARY ACTION. You should clearly separate your contribution from existing works, e.g. to separate your implementation from the implementation that is already given. **NEVER share your reports with others**.

The following sections are recommended as an effective way to organize your writing in your reports.

- **Abstraction**
  In less than 100 words, briefly discuss your contributions in the project.

- **Introduction**
  Summarize the motivation of the project. Highlight the engineering and design challenges in this project and the methods to overcome these challenges.

- **Background**
  Give concise descriptions of the adder architectures. Note that you should cite various references properly, e.g. the project instructions and the textbook.

- **Architectural Exploration of Adders**
  Discuss the trade-offs among the adders and then motivate your choice of adder architecture and parameters. Explain your implementation in plain English, possibly with pieces of Verilog code.

- **Bonus Design**
  Discuss your solution to the bonus design problem if you decide to solve it.

- **Functional Validation and verification**
  Discuss the approaches taken to validate and verify your designs, e.g. RTL simulations with the test bench and the equivalence checking between the two adder designs. Justify your claims of correctness with simulation results or equivalence checking reports.

- **Synthesis Results**
  Discuss the synthesis flow including RTL, post-synthesis, place & route, etc. Explain the differences of the designs at various design stages. **Compare your designs in terms of performance and cost.** Create charts/tables to tabulate the performance results and scaling in terms delay and area. Make sure you reach a conclusion with respect to best adder architecture.

- **Conclusion and Future Work**
  Summarize your contributions and discuss possible future works.

- **Appendix**
  Verilog code/screen shots/results listing.

- **References**