ECE385

Fall 2022

Experiment 3

Introduction to System Verilog, FPGA, CAD, and 16-bit Adders

Lai Shu Xian
Section NL
TA: Nick Lu

Objective

The purpose of this lab is to understand trade-offs and optimizations that can come with different design approaches. We were expected to design a simple arithmetic adder but with different approaches (the carry ripple adder, the carry lookahead adder, and the carry select adder) to speed up its computation time, as well as considering hardware usage. After completing the design, this lab also introduces us to use post-analysis tools to observe and compare the trade-offs between different designs.

Introduction

In terms of abstraction, all three adders that we design in this lab have the same capability – to compute the sum of two 16-bit binary number. They differ however, in the method that how they generate the carry out when addition is performed. The ripple adder generates the carry out by progressively adding the bits right to left (like how most elementary kids do addition). The lookahead adder generates the carry out by performing additional logic with two additional outputs: Propagate and Generate. The carry select adder performs twice the number of its required addition to consider all possibilities of carry out and then uses some simple logic and muxes to choose the correct output.

Adders

a. Ripple carry adder
    a. The design of a ripple carry adder is a **serial combination** of the simplest arithmetic adder unit. The basic block (the simplest arithmetic adder unit) takes in 3 inputs: a single bit from one input, a single bit from another input, and a carry in. and the output are the sum of these three inputs and a carry out is generated per input conditions. These basic blocks are then chained together by connecting a carry out as another block's carry in. This chaining extends indefinitely for as many bits as the inputs are made up of.

Block diagram of a ripple carry adder:

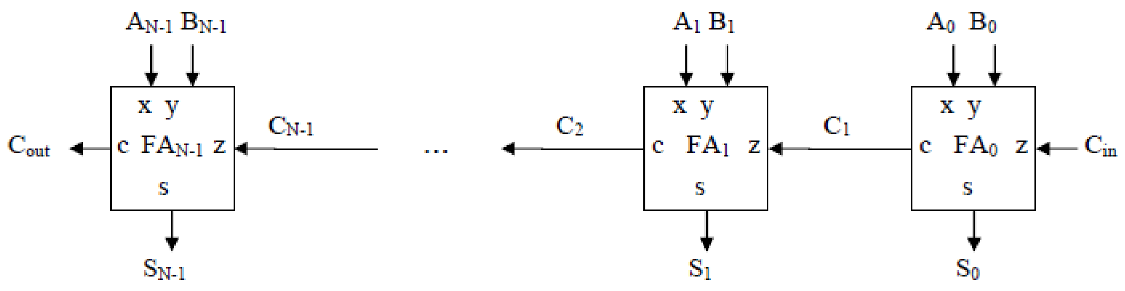

Figure 1 shows the block diagram for a carry ripple adder.

b. Carry Lookahead Adder (CLA)

<u>Written Description</u>

The lookahead adder architecture is made up of the <u>arithmetic adder unit</u> (similar to the basic block for carry ripple adder) and the <u>carry lookahead generator</u>. One difference from the basic arithmetic adder unit in carry ripple adder, the ALU in carry lookahead adder performs additional logic on the inputs to generate **two extra output signals**: Propagate and Generate.

Another aspect of the architecture is the design for CLA is **hierarchical**. We combine a 4-bit CLA and generate two other outputs: which we call the Propagate and Generate groups. We duplicate four of these 4-bit CLA and feed their Propagate and Generate groups into yet another carry lookahead generator (hence the term hierarchical).

Propagate, Generate signals

Propagate(P) and generate(G) are the signals that give CLA the boost in performance because these signals enable the sum of each individual pair of bits to be calculated in parallel without waiting for previous carry forward.

The G signal comes from by observing the fact regardless of any other parts of a binary add, an addition on two bits will generate a carry, hence

$$G = A\ AND\ B, where\ A\ and\ B\ represent\ any\ two\ binary\ bits\ being\ added$$

The P signal comes from the considering the situation where "if the previous added two bits passed in a carry forward, **and** the current addition **(simply due to the carry forward)** will itself produced another carry forward", hence the term propagate.

$$P = A\ OR\ B$$

If we then extend the P and G logic indefinitely across all bits, all the carry forward for each operation can be determined by only the first carry in, and all the respective P and G, using the logic

$$\begin{aligned} CarryForward_0 &= C_{in} \\ CarryForward_1 &= C_{in}P_0 + G_0 \\ CarryForward_2 &= C_{in}P_1P_0 + G_0P_1 + G_1 \\ CarryForward_3 &= C_{in}P_2P_1P_0 + G_0P_1P_2 + G_1P_2 + G_2 \end{aligned}$$

And this pattern can be easily extended with a correct understanding of P and G. And since A & B can be load in parallel and all carry forwards can be obtained irrespective of each other, hence we showed each addition on a pair of bits can be done in parallel.

Hierarchical Design

In terms of building the circuit however, we do not want to simply extend the pattern of getting all the carry forwards as the SOP expression can grow with the number of bits and force unnecessary hardware. By the reasoning of how P and G come about, we can use multiple P and G's to generate a bundle logic for $P_{group}\ and\ G_{group}$. For example if we want to bundle 4 P and 4 G, we can get:

$$\begin{aligned} P_{group} &= P_0P_1P_2P_3 \\ G_{group} &= G_3 + G_2P_3 + G_1P_3P_2 + G_0P_3P_2P_1 \end{aligned}$$

And if we zoom out further from the bundle of $P_{group}\ and\ G_{group}$, then we can realize multiple $P_{group}\ and\ G_{group}$s can be used to generate the carry out between the adder blocks that produced each $P_{group}\ and\ G_{group}$. And because the reasoning P and G do not change from its simplest version where it is used to generate the carry forwards, we can reach a hierarchical design that reuses that logic again.

The hierarchy structure of our 4x4 adder comes from combining 4 individual lookahead adders to form a CLA block (which itself outputs $P_{group}\ and\ G_{group}$). We then have 4 of such CLA blocks and use their $P_{group}\ and\ G_{group}$ to produce carry forwards for those blocks in between.

Block Diagrams (CLA)

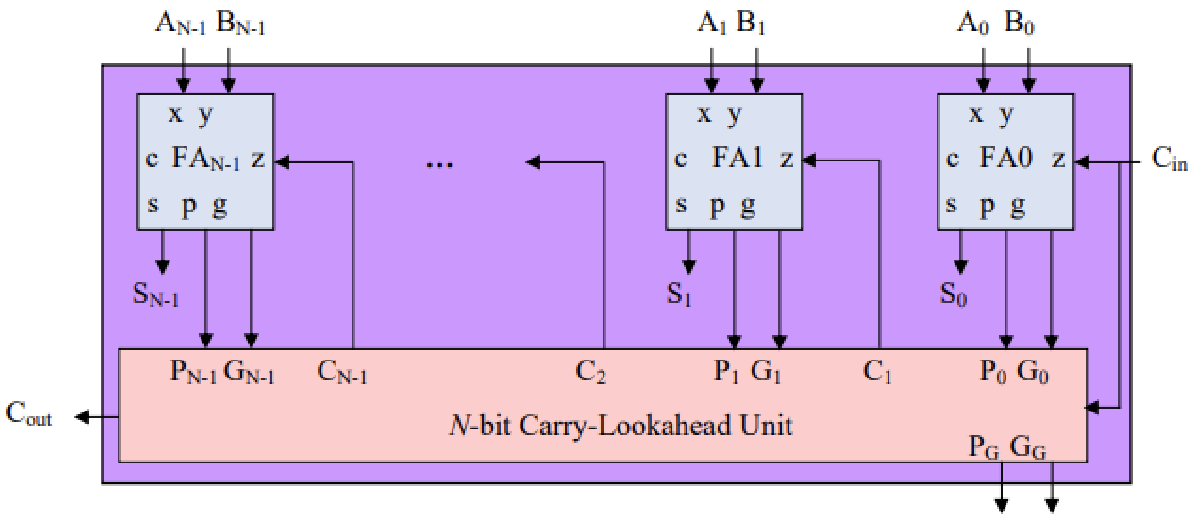

Figure 2 shows a single bundle of 4-bit CLA block

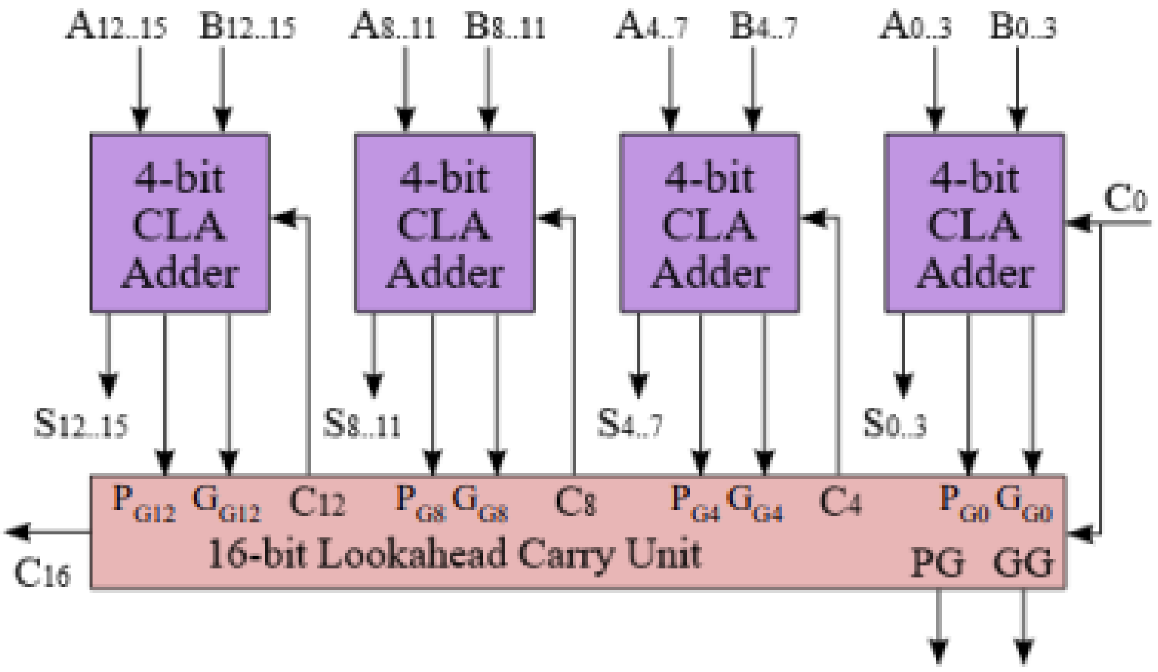

Figure 3 shows the design of figure 2 being replicated 4 times, and their aggregate P and G reuse the carry unit to connect the figure 2 designs together.

c. Carry Select Adder (CSA)

CSA is more of a brute-force attempt to counter the effect of a ripple adder while still maintaining a linear design flow. The operating philosophy behind CSA is to simply calculate each pair's sum by considering the possibility of having and not having a carry forward (by having the two full adders with their carry in inputs to be different). Therefore, it gains some speed up but with the cost of twice the total number of CRAs used (since it considers two possibilities for each add). Muxes are then used to determine which of the two sums should be the final output and some combinational logic is used to determine the carry forward in between the CSA.

CSA achieves its speed up by having the two possible sums readily calculated, waiting only for the carry forward bit from its previous neighbour. Although still taking a serial design and carry forwards are still dependent on previous neighbours, CSA's advantage mainly comes from the fact that when it ripples through to reach the next neighbour (it no longer needs to ripple through the full adder), less computation time is used up since those sums and outputs (for glue logic) are readily available.

Block Diagram

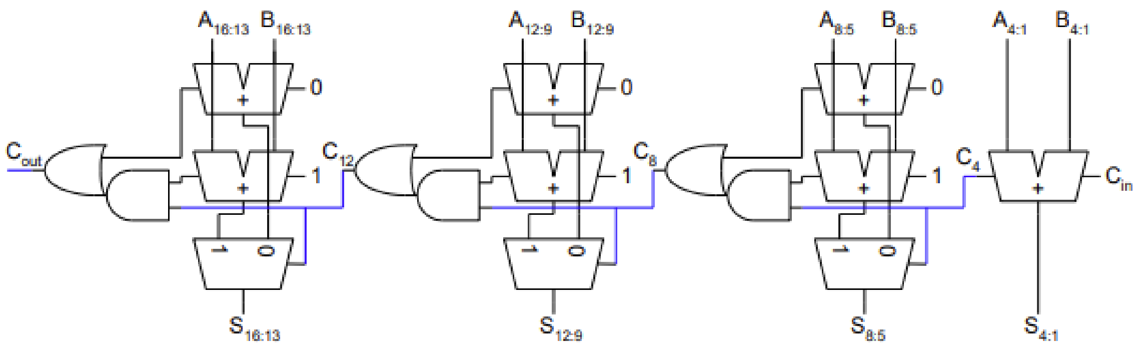

Figure 4 shows the block diagram for a 16-bit adder.

d. Written Description of all .SV modules

Module: adder2.sv Inputs: Clk, Reset_Clear, Run_accumulate, [9:0] SW Outputs: [9:0] LED, [6:0] for Hex0-5	Description: Takes in Clock signal as global clock for register loads. Reset_Clear and Run_accumulate are button presses on the FPGA board, whereas SW is switches input on FPGA. SW will be used to set what we want to add but only to a max of 10 bits because of limited switches. The outputs of the sum (or what we want to load, depending on the buttons pressed) will be displayed on the six segments of FPGA display segments. LED[9] will turn on to indicate overflow in addition. Reset_Clear clears all registers while Run_accumulate adds the current sum to the switches value. Purpose: Acts as a top level design entity that wraps all the modules together to form a 16-bit adder to load into FPGA board.
Module: single_ripperadder.sv Inputs: A, B, Cin Outputs: S, CarryOut	Description: This is a single-bit full adder that performs the addition of A and B with a provided carry in Cin. The output S will be the sum of A + B and CarryOut signifies whether a carry forward is generated from the addition. Purpose: This module is the most basic block of two-input addition.
Module: fourbit_ripple.sv Inputs: [3:0] A, [3:0] B, Cin Outputs: [3:0] S, Cout	Description: This takes 4 single-bit full adder and chain them together by subsequent Cout into the next Cin to form a 4-bit ripple adder. The adder will always perform A + B (this is S) and have the carry forward to be Cout.

	Purpose: Combines the single bit adder to form a bulk of 4, so that can be reused to form 16-bit adder instead of calling single-bit 16 times over.
Module: ripple_adder.sv Inputs: [15:0] A, B, cin Outputs: [15:0] S, cout	Description: This takes two 16-bit inputs, A and B with a 1 bit carry in and perform an addition of A + B. S is the output of the addition and cout signifies whether there is overflow. Purpose: Performs the addition of A + B in a compact, straightforward design.
Module: lookahead_adder.sv Inputs: [15:0] A, [15:0] B , Cin Outputs: [15:0] S, cout	Description: This takes two 16-bit inputs A, B and sum them together. Cin is the first carry in for the least significant bit of A and B. S is the output of the addition and cout signifies whether there is an overflow. Purpose: To perform a 16-bit addition on two operands in parallel manner.
Module: fourbit_CLA.sv Inputs: [3:0] A, [3:0] B, Cin, Outputs: PG, GG	Description: This takes in two 4-bit numbers A and B and perform addition on them. Cin is the carry in for the addition of the least significant bit of A and B. Each bit of A and of B will parallelly generate their own propagate and generate signals. All of these propagates and generates will produce output PG and GG. Purpose: Performs a 4-bit addition in parallel while generating PG and GG which is useful when we can combine multiple PG and GG to be the output that connect multiple units of this module.
Module: fourbitCLA_carryunit Inputs: [3:0] Propagate, [3:0] Generate, Cin Outputs: [3:0] CarryOut	Description: Takes in the 4 propagate signals and 4 generate signals, together with a carry in (Cin) to generate 4 carry forwards (CarryOut[]) for adder units that are connected. Purpose: Instead of relying on previous carry ins and their sum with add operands to get next carry forward, we use the logic behind propagate and generate and 1 single carry in to generate all carry forwards for a 4 bit adder. This module also is reused when we design a hierarchy CLA adder by inputting the aggregate Propagate and Generate into this module.
Module: singleCLA.sv Inputs: [3:0] A, [3:0] B, [3:0] Cin Outputs: [3:0] S	Description: Performs 4-bit addition on A and B with all their carry forward provided readily as inputs. S is the output of the individual bits concatenated and represent the sum of A + B Purpose: Performs addition A + B into S with carry ins that can load in parallel.
Module: select_adder.sv Inputs: [15:0] A, [15:0] B, cin, Outputs: [15:0] S, cout	Description: Takes in 16-bit input A and B and a given carry-in to perform addition to produce 16-bit output S. cout signifies if there is an overflow.

	Purpose: Perform 16-bit addition using the CSA concept.
Module: CSA_select.sv Inputs: cout1, cout2, prev_cout Outputs: next_cout	Description: Glue logic for CSA by taking in carry outs from 1. previous addition 2. current addition with no carry in 3. current addition with carry in to generate the next carry forward. Purpose: To wait for the previous carry forward with ready inputs and generate the carry in for the next addition.
Module: CSA_mux.sv Inputs: [3:0] adder0, [3:0] adder1, select Outputs: [3:0] s	Description: Functions like a mux to use the select signal to choose between two possible sums. If select 0 outputs adder0, if select 1 outputs adder1. Purpose: Serves as a selector select the correct option for CSA.
Module: reg_17.sv Inputs: Clk, Reset, Load, [16:0] Din Outputs: [16:0] Data_Out	Description: Positive edge triggered 17-bit register with asynchronous reset and synchronous load. When Load is high, data is loaded from Din into the register on Clk high edge. Purpose: Module is used to create the registers that store operands A and B in the adder circuit.
Module: control.sv Inputs:Clk, Reset, Run Outputs: Run_0	Description: A state machine that acts as a self-looping whenever the Run button is pressed but has a asynchronous Reset input that makes the state machine halt until Run is hit again. Purpose: Allows the register to load once, and not during the full duration of button press.
Module: router.sv Inputs: R, [15:0] Ain, [16:0] Bin Outputs: [16:0] Q_out	Description: A 17-bit multiplexer gets its select from input R. When R is high, B is parallelly produced as the 17-bit output, otherwise, A is the parallel output. Purpose: Acts as a mux that puts either sum of A+B into register or puts only B
Module: hexdriver.sv Inputs: [3:0] In0 Outputs: [6:0] Out0	Description: Takes in a 4-bit binary number and outputs its corresponding hexadecimal value on a 7-segment LED display setup using switch case statements. Purpose: To light up the correct LED segments to show a 4 bit binary number as a single hex number on the LED display segment.
Testbench.sv Inputs: N/A Outputs: N/A	Purpose: a self-written test file to be run on ModelSim to assist in debugging and verification process of design.

e. Area, performance, and complexity trade-offs between adders.

	Area	Performance	Complexity
Carry ripple adder	Uses the least area because of the straightforward design and no optimization logic.	Performs least optimally because each addition is dependent on the entire chain prior to it (waiting for the carry forward). If we are adding two n-bit numbers, the delay grows linearly with n.	Simplest and most straightforward to design as the only logic involves is the addition of two bits being replicated n times for a n-length addition. There is no glue logic at all between blocks, just directly chaining them together.
Carry lookahead adder	Consumes the most area to achieve speed-up. Design is area-consuming because of multiple additional signals being generated, the Propagate and Generate for each bit, as well as the propagate and generate groups. The additional layers between the hierarchies also demands more hardware to perform.	Has the best performance amongst these three adders as the addition performed on each bit can be done independent of one another.	Involves a more complicated logic design process and complex architecture structure. To achieve full parallel addition, we have to aggregate signals and combine them and use a hierarchical structure.
Carry select adder	Consumes more area than CRA but less than CLA. More than CRA because twice the addition is being performed but only one extra signal for bitwise addition.	Performs marginally better than CRA but not as fast as CLA because the carry forward is still being rippled. Is slightly faster than CRA because all the additions are partially completed in parallel and all of them directly produce an output when the carry forward arrives.	Slightly more complicated than CRA but still mainly repetitive. The additional logic is also straightforward with the use of a mux and a two-level combinational logic.

f. Performance Analysis

	Carry Ripple Adder	Carry Select Adder	Carry lookahead adder
Memory (BRAM)	0	0	0
Frequency	66.12MHz	68.21MHz	68.89MHz
Total Power	98.71mW	104.21mW	105.12mW

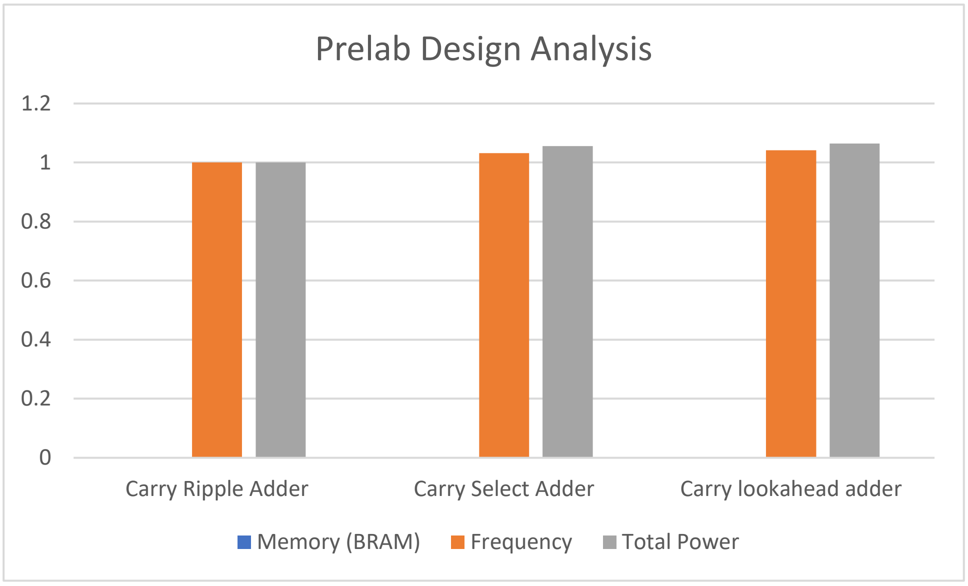

Figure 5 shows the bar chart comparison for the three different adders.

g. Annotated simulation trace

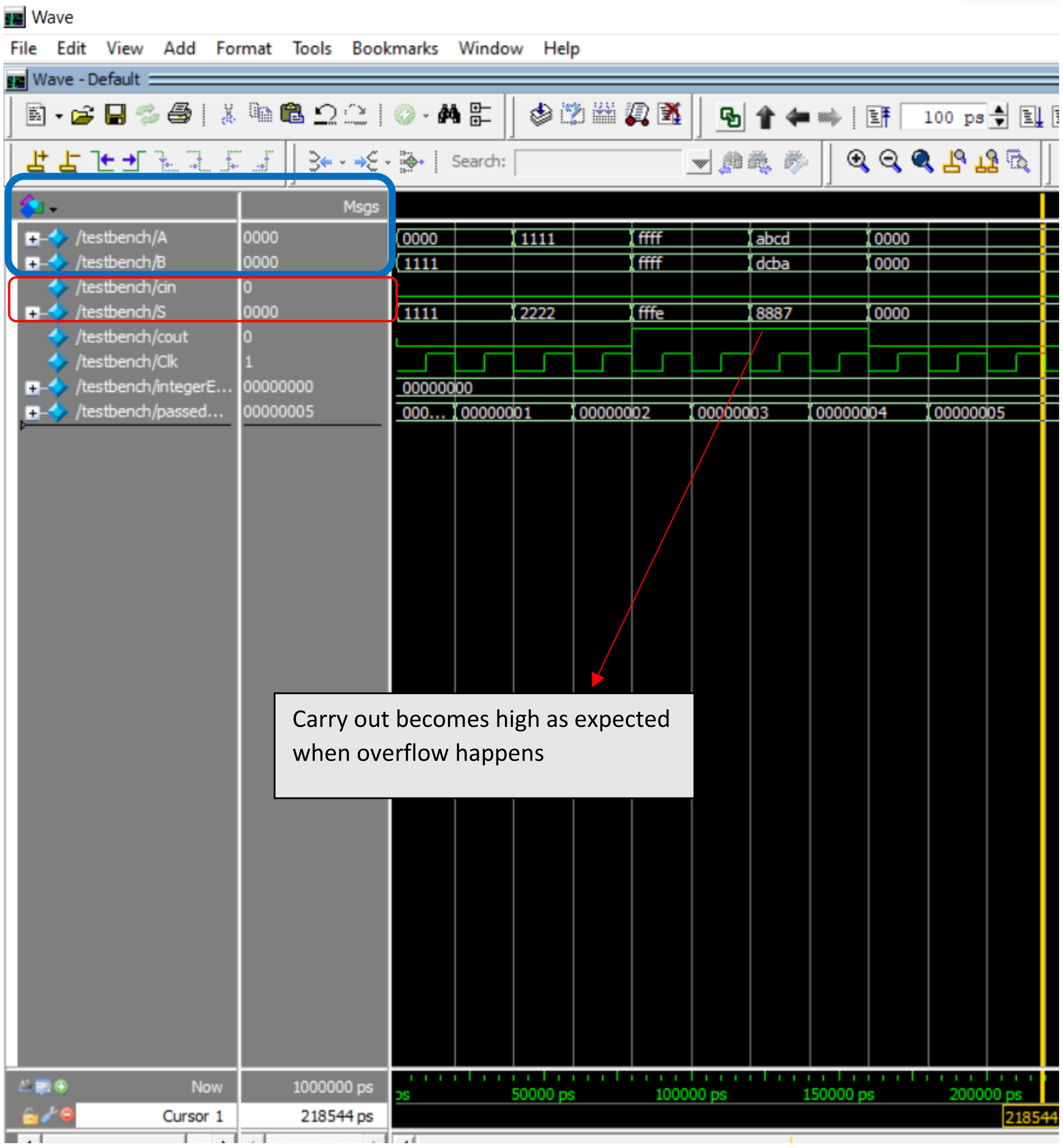

Figure 6 shows the annotated waveform for simulation on one of the adders (CRA).

Description of simulation trace: The blue box shows our input signals (which in this experiment are the two sixteen-bit number to be added together), A and B represented in hexadecimal format. The red box shows the output for the adder, S representing the sum, and cout representing whether there is a 1-bit overflow.

In the simulation, 5 test cases were ran and we can read the signal column by column, left to right. The test cases that I conducted were:

x0000 + x1111 => x1111

x1111 + x1111 => x2222

xffff + xffff => xfffe (overflowed)

xabcd + xdcba => x8887 (overflowed)

x0000 + x0000 => x0000

All outputs match with expected result. The signals at the bottom are just for sanity check, they are respectively the Clock signal, the number of test cases failed, and the number of test cases passed and we can see that we passed all 5 test cases.

Answers to Post-Lab Questions

1. Is 4x4 the ideal hierarchy design? Possible ways to improve.

The information we can try to find out is whether combinational logic with 'longer' terms is faster than combinational logic with multiple expressions but shorter terms.

For example, in our 4x4 design, we used the logic below to generate the aggregate Propagate and Generate signals. But we can achieve the same parallelism with smaller granularity by further breaking down the structure.

$$P_G = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$G_G = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

For instance, we can only have the aggregate Propagate and Generate signals to only combine two propagates and two generates with the following logic:

$$P_G = P_0 \cdot P_1$$
$$G_G = G_1 + G_0 P_1$$

We can experiment that on an 8x2 hierarchy design and see if we gain any improvements based on design analysis after running it.

2.

	CRA	CSA	CLA
LUT	76	79	87
DSP	0	0	0
Memory (BRAM)	0	0	0
Flip-flop	20	20	20
Frequency	66.12MHz	68.21MHz	68.89MHz
Static Power	89.84mW	89.97mW	89.97mW
Dynamic Power	0.00mW	1.35mW	1.40mW
Total Power	98.71mW	104.21mW	105.12mW

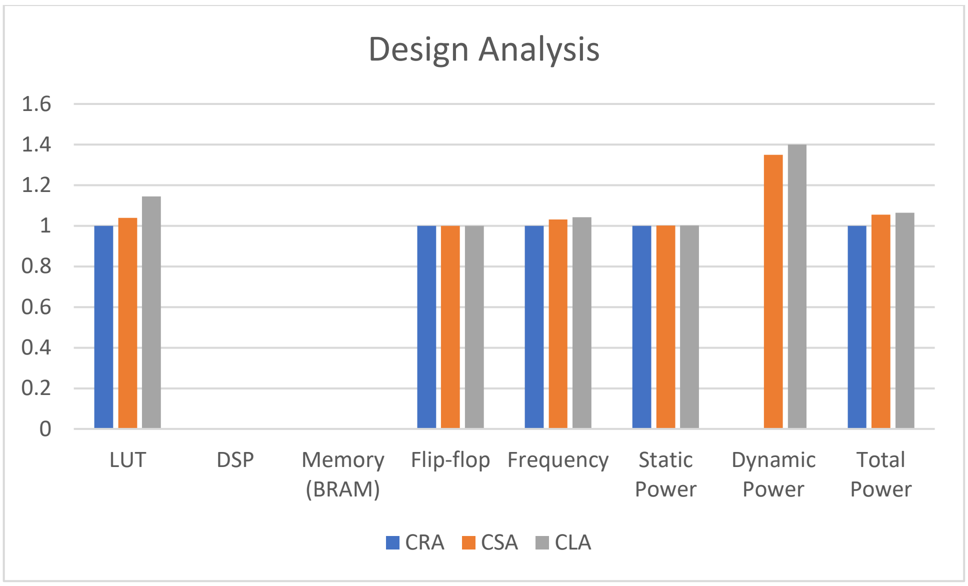

Figure 7 shows the resource breakdown plotting after normalizing based on CRA.

The resource breakdown comparison makes sense because we can see that as the design complexity goes up (highest being CLA and lowest being CRA), we gain performance speed up (CLA has a higher operating frequency than CRA) in the cost of power (CLA also has highest power). The numbers comply with the theoretical design as we use up the least LUT for the most compact CRA and most LUT for CLA.

Conclusion

The foundational concept of this lab is not rocket science – it's just addition with using different methodologies and design approaches. As such, there was no significant logic bugs, and if there was, they mainly arose because of copy-pasting code without careful inspection. And, since most of the logic expressions were readily and directly provided, one can mostly produce careless bugs. The most challenging part mainly comes from understanding the hierarchical design of the CLA and coding out the hardware design code, as well as the conceptual understanding of how CSA performs faster than CRA but not as fast as CLA.

The lab wasn't unnecessarily difficult, but I thought the explanation of how we should perform an addition was quite a pain. The lab has been repeatedly done over many semesters, if our feedback really matters, they should already have.