ECE385 Spring 2023 – Lab 3 Report

Ziyuan Chen, Weijie Liang | ziyuanc3, weijiel4

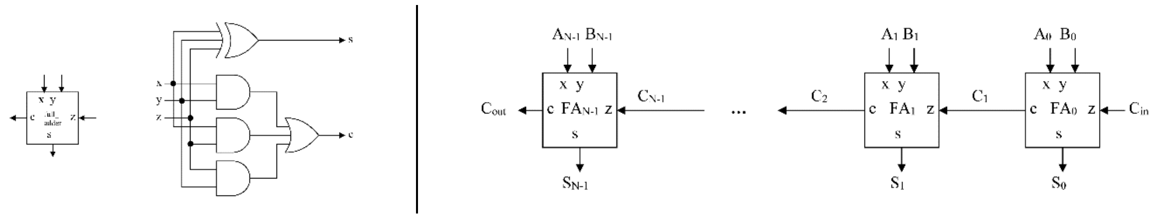
Specific colors are used to highlight the Register, MUX, FSM modules,
the `Din[9:0]`, `Load`, `Run`, `Reset`, and `CLK` signals throughout the document.

Introduction

The three adders in this lab take two 16-bit numbers and add them together. With the help of periphery modules, the user can input an operand `Din` and choose whether to accumulate the number into the register or overwrite the register with this initial value. An independent FSM ensures the circuit performs exactly one operation for each button press.

The Ripple Carry Adder (CRA)

This is the most intuitive variant of a multi-bit adder. Each unit in a CRA – often referred to as the *full adder* – gets the carry-bit `Cin` from its previous unit and computes the sum `S` and carry-bit `Cout` for the next unit. 16 full adders are serially connected to propagate the carry bit and sum up two 16-bit numbers. Adders in the more significant bits must wait for the carry bit to "ripple" through the circuit, resulting in computational inefficiency.

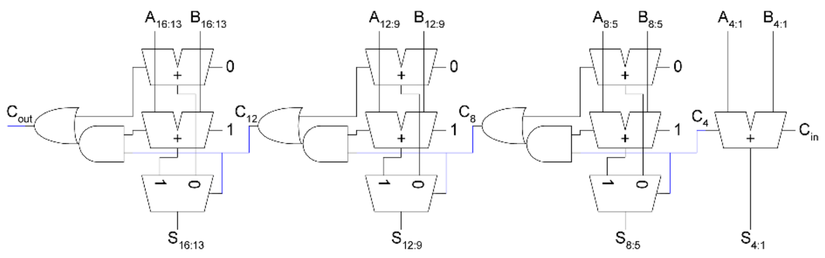


The Carry Select Adder (CSA)

The CSA adopts a hierarchical structure comprising smaller adder units, each 4 bits wide in our design choice. The diagram shows that each CSA unit is effectively two CRAs plus some glue logic, just as in our code. They calculate the sum segments based on `A` and `B` segments combined with a *hypothetical* previous carry bit (`0` or `1`). Once the previous carry bit is determined and propagated to the next CSA unit, a 2-to-1 MUX can immediately select between the two cases. Note that the first CSA unit can be simplified into a 4-bit CRA since the previous carry bit can only be `0`.

As a metaphor, each CSA unit is prepared with two pre-calculated “parallel universes,” and the “wave function” can instantly “collapse” once the prerequisite information is ready. The following timeline illustrates this acceleration technique. (The CRA takes T = 16)

- T = 1, `S[0]` determined, possibilities of `S[4]`, `S[8]`, `S[12]` ready
- T = 2, `S[1]` determined, possibilities of `S[5]`, `S[9]`, `S[13]` ready
- T = 3, `S[2]` determined, possibilities of `S[6]`, `S[10]`, `S[14]` ready
- T = 4, `S[3]` determined, possibilities of `S[7]`, `S[11]`, `S[15]` ready, C[4] ready
  - Compared to the CRA where `S[3:0]` is also ready at T = 4, the CSA has additionally prepared two possibilities of `S[15:4]`.
- T = 5, `C[4]` propagated, `S[7:4]` determined, `C[8]` ready
- T = 6, `C[8]` propagated, `S[11:8]` determined, `C[12]` ready
- T = 7, `C[12]` propagated, `S[15:12]` determined, `Cout` ready, `S[15:0]` ready
  - MUXes with multi-bit input/output are effectively parallel. Over 120% faster!



The Carry Lookahead Adder (CLA)

The CLA has a different methodology to accelerate the propagation of carry bits. For each pair of bits from `A` and `B` and the previous carry bit, unlike the full adder that computes the next carry bit, a CLA cell (1-bit adder) calculates the `Propagate` and `Generate` bits – information for other cells to “look ahead” and get the carry bit early. This notion can be somehow counterintuitive, but consider the following truth table for a full adder:

A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0

A	B	Cin	Cout	S
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

In the red-shaded cells, `Cout` = `Cin` and the carry bit is "Propagated" if *exactly one of* `A` and `B` is `1` (`P` = `A XOR B`). In the green-shaded cells, `Cout` = `1` and the carry bit is "Generated" if *both* `A` and `B` are 1 (`Cout` = `A AND B`). The power of `P` and `G` lies in the feature that they can be computed independently of previous bits, providing a shortcut to higher carry bits.

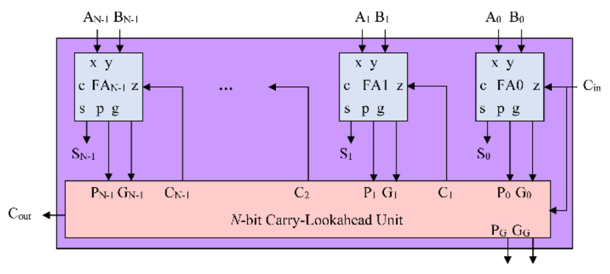
Correspondingly, the "previous" carry bit for a CLA cell does *not* come from the previous cell. Instead, this "team effort" of carry lookahead is concerted by a dedicated Carry Lookahead Unit (CLU), which accepts *n* `P`s, *n* `G`s, previous `Cin`, and produces (*n*–1) carry bits back to the cells along with `Cout`. Consider this example in which the second carry bit originates from `Cin`, the zeroth, or the first carry bit in the three SOP terms.

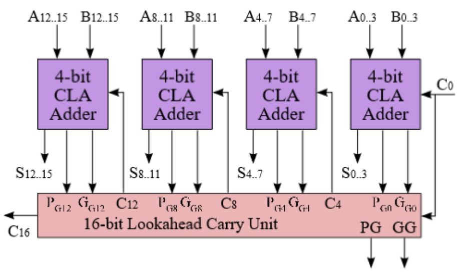
$$C_2 = C_{in} \cdot P_0 \cdot P_1 + G_0 \cdot P_1 + G_1$$

These logical equations, although convenient, can be long and require extremely large gates for higher carry bits. Therefore, we sacrifice some speed for plausibility and introduce some seriality. The 16-bit CLA has a similar 4×4 hierarchy, in which each 4-bit group propagates and generates carry bits as efficiently as possible, and the carry bits are traditionally rippled *between* units – or we can do better. We adopt a larger CLU to "wrap up" the lookahead information from each group, enabling the CLA to be, in theory, infinitely nested (e.g., 4×4×4). This introduces the final concept of Group Propagate `Pg` and Group Generate `Gg`:

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

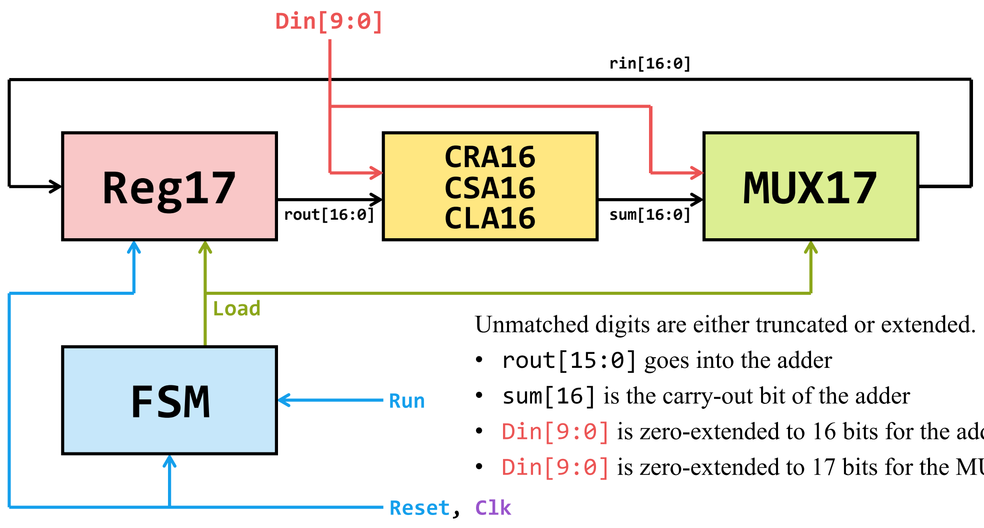
A carry bit is propagated through the group if and only if it propagates through every CLA cell, and is generated in the group if and only if it is generated in any of the four cells and propagated through the rest of the cells (corresponding to the four SOP terms). The 4-bit CLUs are responsible for computing these two extra output values.





$A_{12..15}$ $B_{12..15}$ $A_{8..11}$ $B_{8..11}$ $A_{4..7}$ $B_{4..7}$ $A_{0..3}$ $B_{0..3}$
4-bit CLA Adder
4-bit CLA Adder
4-bit CLA Adder
4-bit CLA Adder
C_0
$S_{12..15}$ $S_{8..11}$ $S_{4..7}$ $S_{0..3}$
P_{G12} G_{G12} C_{12} P_{G8} G_{G8} C_8 P_{G4} G_{G4} C_4 P_{G0} G_{G0}
C_{16}
16-bit Lookahead Carry Unit
PG GG

SystemVerilog Modules



Unmatched digits are either truncated or extended.

- rout[15:0] goes into the adder
- sum[16] is the carry-out bit of the adder
- Din[9:0] is zero-extended to 16 bits for the adder
- Din[9:0] is zero-extended to 17 bits for the MUX

Reg17	**Inputs:**	rin[16:0], Load, Reset, Clk
	Outputs:	rout[16:0]
	Description:	At the positive edges of Clk, rout is loaded with rin when Load is set and cleared when Reset is set.
	Purpose:	Register unit that holds the accumulated result.
C*A16	**Inputs:**	rout[15:0], Din_ext[15:0]
	Outputs:	sum[15:0], sum[16] (carry-out bit)
	Description:	Performs parallel addition and outputs the result to sum.
	Purpose:	Algorithmic interface to the register and MUX.
MUX17	**Inputs:**	sum[16:0], Din_ext[16:0], Load
	Outputs:	rin[16:0]
	Description:	rin is loaded with sum when Load is set and Din_ext when Load is clear.
	Purpose:	Routing unit that updates the register value.
CRA1	**Inputs:**	A, B, Ci
	Outputs:	S, Co
	Purpose:	Standard 1-bit full adder.

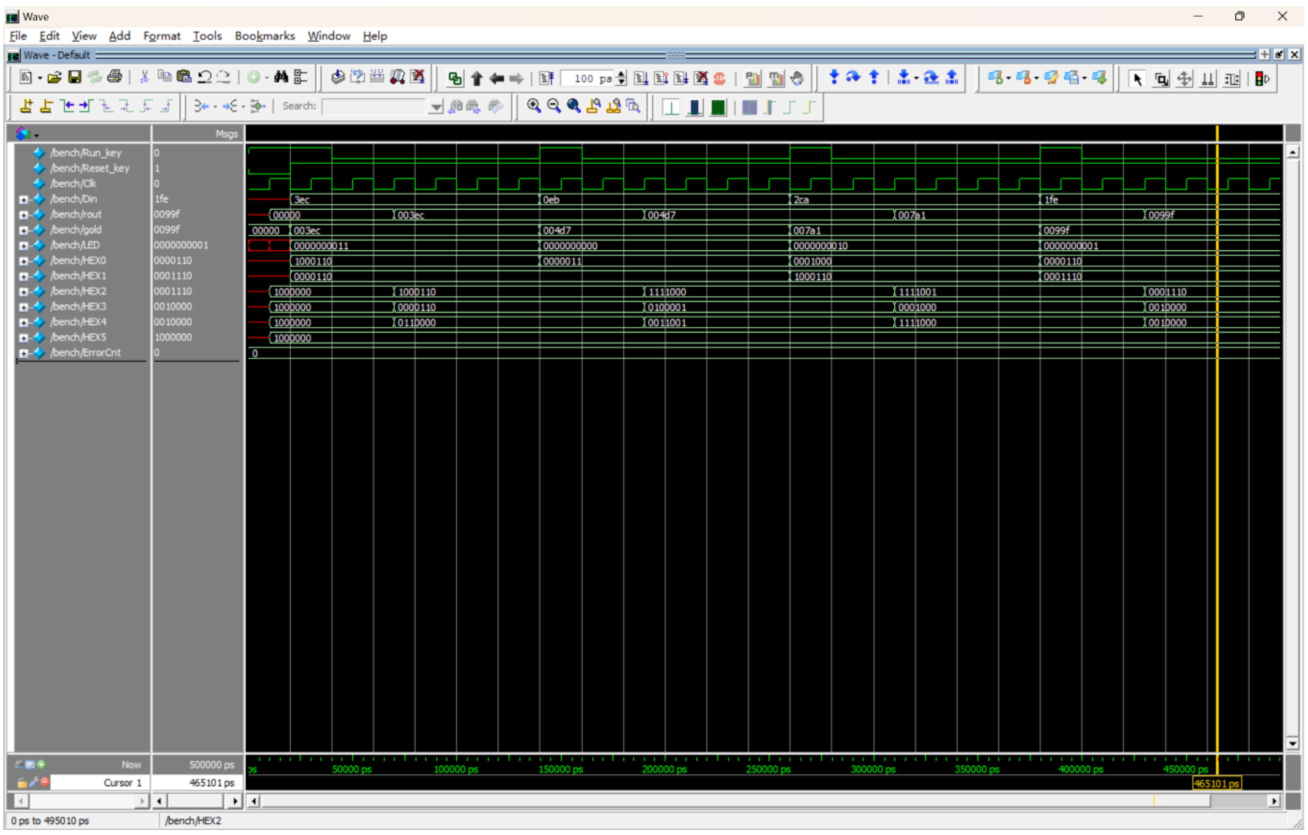
`CLA1`	**Inputs:**	`A, B, Ci`
	Outputs:	`S, P, G`
	Description:	Computes the "intermediates" `P` and `G` into `CLU4`.
	Purpose:	The "lookahead variant" of a 1-bit full adder.
`CRA4`	**Inputs:**	`A[3:0], B[3:0], Ci`
	Outputs:	`S[3:0], Co`
	Purpose:	4-bit Carry Ripple Adder module.
`CSA4`	**Inputs:**	`A[3:0], B[3:0], Ci`
	Outputs:	`S[3:0], Co`
	Description:	Inherits two `CRA4`s that concurrently compute two versions of `S`, which are then fed into a MUX for selection according to the carry-out bit `Ci` of the former `CSA4` unit.
	Purpose:	4-bit Carry Select Adder module.
`CLA4`	**Inputs:**	`A[3:0], B[3:0], Ci`
	Outputs:	`S[3:0], Pg, Gg`
	Description:	Wraps four `CLA1`s and a `CLU` together. Computes the collective `Pg` and `Gg` for hierarchical lookahead.
	Purpose:	4-bit Carry Lookahead Adder module.
`CLU4`	**Inputs:**	`P[3:0], G[3:0], Ci`
	Outputs:	`C1, C2, C3, Co, Pg, Gg`
	Description:	Implements the logic for generating four carry-in bits from the "intermediates." It also computes grouped `Pg` and `Gg`.
	Purpose:	4-bit Carry Lookahead Unit.
`FSM`	**Inputs:**	`Run, Reset, Clk`
	Outputs:	`Load`
	Description:	`Load` is set for precisely one `Clk` cycle after each rising edge of `Run`. `Reset` forcefully resets the FSM.
	Purpose:	Control unit that manages updates of the register.

Design Tradeoffs

CRA uses the fewest logic gates and is the simplest model, but its time complexity is $O(n)$, where n is the number of input bits. It must wait for the carry bits to ripple.

CSA has time complexity $O(n)$, but it operates faster than CRA *by a constant factor* because we can decrease the delay by using nearly twice the number of gates as CRA (which introduces the tradeoff). If n bits are divided into k groups, the number of delay levels is $\frac{n}{k}$.

CLA adopts a complex logic and requires the largest area; in return, the time complexity *in each hierarchy* is $O(1)$ because we can figure out the carry bits in constant time using `P` and `G`. A hierarchical design helps us use fewer gates at the cost of some efficiency.



	Msgs
/bench/Run_key	0
/bench/Reset_key	1
/bench/Clk	0
/bench/Din	1fe
/bench/rout	0099f
/bench/gold	0099f
/bench/LED	0000000001
/bench/HEX0	0000110
/bench/HEX1	0001110
/bench/HEX2	0001110
/bench/HEX3	0010000
/bench/HEX4	0010000
/bench/HEX5	1000000
/bench/ErrorCnt	0

ModelSim trace *that shows the accumulation of four numbers: 10'h3EC, 10'h0EB, 10'h2CA, and 10'h1FE.*
IMPORTANT: *Running this simulation requires* `rout` *to be defined as output in the top-level* `Adder`.

Annotations
(0x3EC, 0x4D7, 0x7A1, 0x99F) is the prefix sum of the array (0x3EC, 0x0EB, 0x2CA, 0x1FE).
Adder initialized at 10ns. Inputs are ready at 20ns, 140ns, 260ns, and 380ns.
Computation cycles begin at 40ns, 160ns, 280ns, and 400ns.
Computation cycles finish at 70ns, 190ns, 310ns, and 430ns.

Post-Lab Problems

- Is the 4×4 hierarchy ideal for CSA?

  It may not be ideal – 8×2 or 2×8 hierarchies may also work well.

  - 4×4 hierarchy: 4 levels of CRA delay + 3 levels of 2-to-1 MUX delay
  - 8×2 hierarchy: 8 levels of CRA delay + 1 level of 2-to-1 MUX delay
  - 2×8 hierarchy: 2 levels of CRA delay + 7 levels of 2-to-1 MUX delay

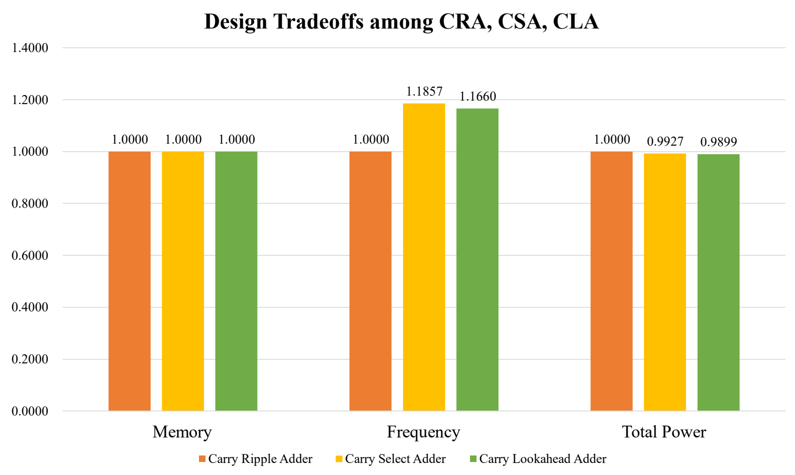
  The sum of delays has properties similar to the function of $f(x) = x + \frac{1}{x}$. If we assume each CRA delay is *a* and each MUX delay is *b*, then we design the hierarchy to minimize

$$f(x) = ax + \frac{16b}{x}$$

. At $x = 4\sqrt{\frac{b}{a}}$, $f(x)_{min} = 8\sqrt{ab}$. Determining *a* and *b* requires analyses of physical circuits.

- Design Resources and Statistics

	CRA	CSA	CLA
LUT	588	595	598
DSP	0	0	0
Flip-Flop	719	720	720
Memory (BRAM)	9,216	9,216	9,216
Frequency/MHz	81.19	96.27	94.67
Static Power/mW	90.00	89.99	89.99
Dynamic Power/mW	6.77	6.05	5.72
Total Power/mW	110.75	109.94	109.63



The three adders are identical in their memory usage, but the CRA operates at an approx. 15% lower frequency than the other two, corresponding to the “rippling” nature of carry bits. Nonetheless, it appears strange that we observe very little difference in power consumption – this is possibly related to the warning from Quartus that power estimation has “low confidence” since the “user provided insufficient toggle rate data.” Typically, we would expect the CLA to consume the most power since it adopts the most complex design with the most elements.

Critical Path Analysis *(Refer to our GitHub repository for clear images)*

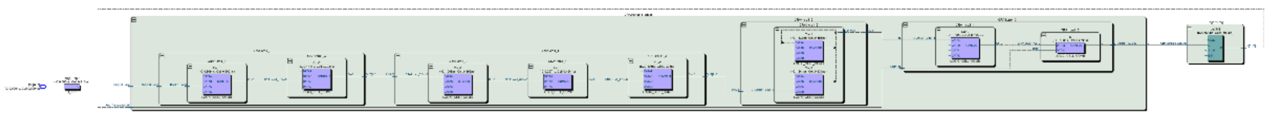
Besides the 0.5ns delay at rising `Clk` edges on the two buttons, we set the following I/O false paths to exclude noise from unrelated, purely combinational paths:

```
set_false_path -from [get_ports {Din*}] -to [get_ports {LED*}]
set_false_path -from [get_ports {Din*}] -to [get_ports {HEX*}]
```

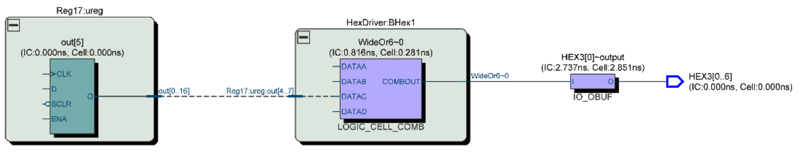
	Slack/Setup	Hold	Recovery	Removal	PWMin	Skew	Delay
CRA	7.683	0.262	17.126	2.181	9.552	3.665	15.995
CSA	**9.613**	0.252	17.149	2.164	9.547	-3.495	**6.892**
CLA	9.437	0.244	17.412	1.957	9.540	3.392	13.366

Focusing on the lowest slack value, we document the critical path in each design:

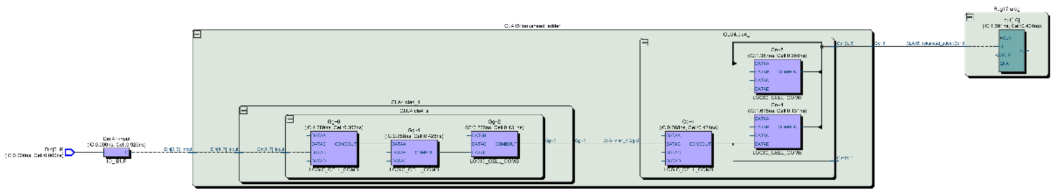
CRA	Din[1]	1@0	3@2	4@0	5@0	7@2	10@0	10@1	13@0	14@S	Reg[14]
IC	0.000	**4.809**	0.441	1.008	0.297	0.403	**1.641**	**1.336**	**1.389**	0.686	0.000
Cell	0.916	0.303	0.303	0.369	0.424	0.303	0.131	0.355	0.359	0.433	0.089



CSA	Reg[5]	HDB1@0	HEX3[0]
IC	0.000	0.816	**2.737**
Cell	0.000	0.281	2.851



CLA	Din[4]	S1@G0	S1@G1	S1@G2	L@C1	L@C4	L@C5	Reg[16]
IC	0.000	**4.389**	0.288	0.236	0.668	**1.618**	**1.367**	**1.691**
Cell	0.929	0.303	0.428	0.131	0.424	0.131	0.359	0.404



Notes on the abbreviations:
“7@2” is short for “cra4_1:cra1_3@Co~2”, “13@0” is short for “cra4_3:cra1_1@Co~0”,
“14@S” is short for “cra4_3:cra1_2@S”, “HDB1@0” is short for “HexDriver:BHex1@WideOr6~0”,
“S1@G1” is short for “cla4_1:clu4_s@Gg~1”, “L@C5” is short for “clu4_l@Co~5”, etc.

Corresponding to the 18.6% improvement in frequency (compared to CRA) in the Statistics section, the **CSA** has such an efficient computation route between the switches and the registers that this delay is less significant than the register-HEX route (with a stunning delay of 6.9 and slack rate 9.6). The switch-register pairs are not even in the top 10 lowest routes!

The **CLA** also has a lower delay than CRA, but this is mainly due to a *shorter data path* instead of a *lower delay in each element* – like the CRA, it takes longer for elements on the two ends to produce valid value, and a delay of 1.7 falls on the register alone!

Conclusion

In this lab, we design 3 kinds of 16-bit adders, namely CRA, CLA, and CSA. A subtle bug in our testbench is that the circuit only accepts 10-digit operands (from `SW`s). The lab manual is clear, but the CLA may be simplified: the 4-bit CLAs introduces too much overhead compared to the small scale of our design. Instead, the 16-bit CLU can simply take `A` and `B`, compute `Pg`s, `Gg`s, and $C_{4,8,12}$, then output the carry bits to the four 4-bit **CRA**s.

Also, the following code must appear in the SDC file to constrain all inputs/outputs:

```
create_clock -name {altera_reserved_tck} -period 100.000 -waveform { 0.000 50.000 } [get_ports {altera_reserved_tck}]
set_input_delay -add_delay -rise -clock [get_clocks {Clk}]  0.000 [get_ports {altera_reserved_tdi}]
set_input_delay -add_delay -rise -clock [get_clocks {Clk}]  0.000 [get_ports {altera_reserved_tms}]
set_output_delay -add_delay -rise -clock [get_clocks {Clk}]  0.000 [get_ports {altera_reserved_tdo}]
set_clock_groups -asynchronous -group [get_clocks {altera_reserved_tck}]
```