TABLE OF CONTENTS

1.0 Introduction

The objective of this project is to design, synthesize and evaluate by testing out four adding circuits. Each design will be compared for performance and hardware cost to support decision making by shareholders, future investors and managers.

1.1 Background

Addition is one of the most frequently used operations in an arithmetic logic unit (ALU) of a CPU. Performance is limited by computing large numbers in large systems such as 64 bit computers. Improving the speed of circuits will prevent clock latency due to large operands. Thus, our focus for these circuits is speed for large operands. A 1-bit full adder entity should be implemented as a purely combinational circuit using signal assignments to compute the summation, carry out, and overflow. Four topologies are derived from this entity and evaluated in terms of timing and resource usage (LEs/ALMs) on FPGA devices.

1.2 Purpose Statement

This report compares the four candidates that meet the Design Under Test (DUT) specifications. The design principle, topology, and implementation will be outlined for each candidate, and their timing performance and hardware costs will be provided from verification results. The goal is to identify the trade-offs between designs and determine the most suitable candidate.

1.3 Experimental Procedure

The functional verification process that we adopted for this project involved creating a diverse set of test vectors, located in the Adder00.tvs file under ./Simulation/TestVectors. The first 23 test vectors in this file were created to ensure correct operation of each output bit individually, and the last 29 are randomly generated test vectors created by the program genTVDP1-1 program located in the same directory. All test vectors were verified for accuracy by the verifyTVDP1-1 program, also in the same directory. After creating this set of test vectors, we also created the .do scripts CSANfv.do and RCANfv.do, which streamline the functional verification process, allowing for recompilation, execution of tests, and verbose, single-string output logging to timestamped transcript files located in the ./Simulation/OutputFiles directory, for both topologies, all with the execution of a single do command. The timestamped transcripts contain a series of lines, each detailing all important information regarding the outcome of a test vector. Below are some examples of transcript lines generated by these .do scripts:

```
#
# ** Note: Measurement #1 Passed. Stimulus: [A: 0AAAAAAAAAAAAAAA B: 0555555555555555 Cin: 1] Expected Outputs: [S: 1000000000000000 Cout: 0 Ovfl: 0] Actual Outputs: [S: 1000000000000000 Cout: 0 Ovfl: 0]
#    Time: 11 ns  Iteration: 0  Instance: /tbcsan
# ** Note: Measurement #2 Passed. Stimulus: [A: 0000000000000000 B: 0000000000000000 Cin: 0] Expected Outputs: [S: 0000000000000000 Cout: 0 Ovfl: 0] Actual Outputs: [S: 0000000000000000 Cout: 0 Ovfl: 0]
#    Time: 22 ns  Iteration: 0  Instance: /tbcsan
# ** Note: Measurement #3 Passed. Stimulus: [A: FFFFFFFFFFFFFFFF B: 0000000000000000 Cin: 0] Expected Outputs: [S: FFFFFFFFFFFFFFFF Cout: 0 Ovfl: 0] Actual Outputs: [S: FFFFFFFFFFFFFFFF Cout: 0 Ovfl: 0]
```

Figure 1.3.1: Three Successful Tests: Measurement #1, #2, #3

```
#    Time: 539 ns  Iteration: 0  Instance: /tbcsan
# ** Note: Measurement #50 Passed. Stimulus: [A: B5E131DC0DD0E676 B: 699967DBD6954B50 Cin: 1] Expected Outputs: [S: 1F7A99B7E46631C7 Cout: 1 Ovfl: 0] Actual Outputs: [S:
1F7A99B7E46631C7 Cout: 0 Ovfl: 0]
#    Time: 550 ns  Iteration: 0  Instance: /tbcsan
# ** Note: Measurement #51 Passed. Stimulus: [A: 260A30FB7A405734 B: DE58C54DDA785F7A Cin: 1] Expected Outputs: [S: 0462F64954B8B6AF Cout: 1 Ovfl: 0] Actual Outputs: [S:
0462F64954B8B6AF Cout: 0 Ovfl: 0]
#    Time: 561 ns  Iteration: 0  Instance: /tbcsan
# ** Note: Measurement #52 Failed. Reason:-|Sum-Mismatch|- Stimulus: [A: 27134568A6A8FD69 B: 8FA4C4C3B7BC4D95 Cin: 1] Expected Outputs: [S: B6B80A2C5E654AF1 Cout: 0 Ovfl: 0] Actual
Outputs: [S: B6B80A2C5E654AFF Cout: 0 Ovfl: 0]
```

Figure 1.3.2: An Unsuccessful Test on Measurement #52

It is important to note that for each of these test vectors, an appropriate pre-stimulation time, and post-stimulation time, is waited for in between every measurement to ensure the stability of the input and output signals. In this case, we have chosen a pre-stimulation time of 1ns and a post-stimulation time of 10ns. Something to realize is that, as we are only testing topologies, propagation delay is nonexistent, as the topologies are still theoretical and not physically implemented. That means that the signals are assumed to settle instantly within the simulation model itself, so these times serve only to ensure that the simulator correctly registers the inputs and outputs without any glitches or race conditions, rather than modelling real hardware delays.

2.0 Design Candidates

Each candidate is to be constructed from a 1-bit full adder entity and evaluated as a topology/FPGA implementation pair. Two topologies are considered: a conditional sum adder network (CSAN) and a ripple carry adder network (RCAN), both with overflow detection. Each topology will be synthesized on two FPGA families: a Cyclone IV and an Arria II. The baseline to compare against in terms of speed and resource utilization is the RCAN synthesized on a Cyclone IV.

2.1. Design Candidate 1: Ripple Carry Adder - Baseline (Cyclone IV)

The full adder entity uses a structural implementation with direct signal assignments. The baseline architecture directly instantiates one full adder entity for the base case (N=1). For N > 1, additional adders are generated for bits 1 through N−2, with a final adder for bit N−1. A signal array of carries propagates carry values between bits. Overflow is detected by comparing the carry-in and carry-out of the most significant bit. If they differ, an overflow has occurred, indicating that the result exceeds the representable range for the given bit width.

Content Rectangle 2.1.1: VHDL code of the architecture of a RCAN:

```
architecture Baseline of RCAN is

signal carries : std_logic_vector(N-1 downto 0);
signal tempC : std_logic;

begin
--            S[0]:
              U0 : entity work.FullAddr(behavioural) port map(X => X(0), Y => Y(0),
Cin => Cin, S => S(0), Cout => carries(0));
--            S[62:1]:
              gen0 : for i in 1 to N-2 generate
                     U1 : entity work.FullAddr(behavioural) port map(X => X(i), Y =>
Y(i), Cin => carries(i-1), S => S(i), Cout => carries(i));
              end generate gen0;
--            S[63]:
              U3 : entity work.FullAddr(behavioural) port map(X => X(N-1), Y =>
Y(N-1), Cin => carries(N-2), S => S(N-1), Cout => tempC);
              Cout <= tempC; --Carry out
              Ovfl <= tempC XOR carries(N-2); -- Overflow
end architecture;
```

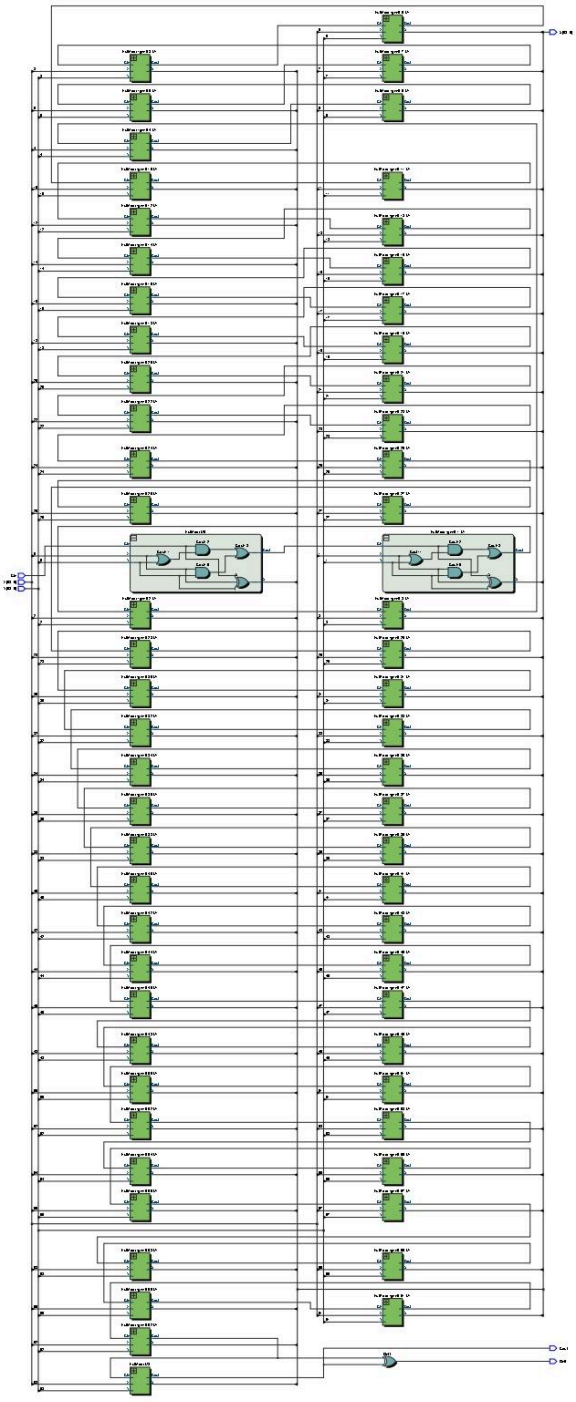
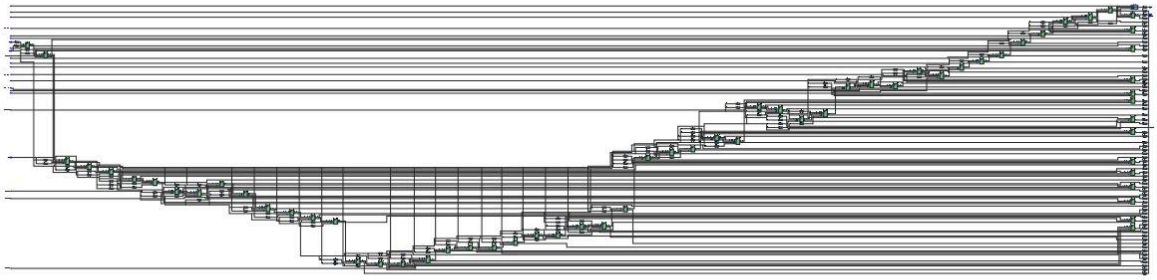


Figure 2.1.2: RTL View of the Baseline, which shows each 1-bit full adder entity as a green box, and carries propagating through each adder.



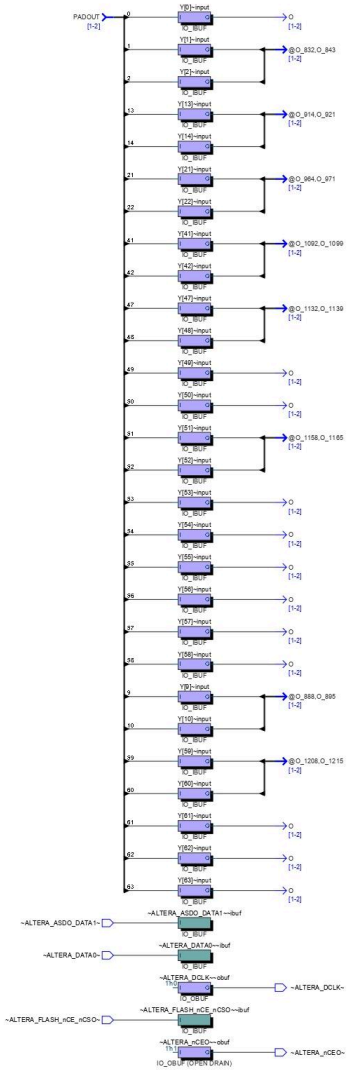

PADOUT
[1-2]
Y[0]~input
IO_BUF
O
[1-2]
Y[1]~input
@O_832.O_843
Y[2]~input
Y[13]~input
@O_914.O_921
Y[14]~input
Y[21]~input
@O_964.O_971
Y[22]~input
Y[41]~input
@O_1092.O_1099
Y[42]~input
Y[47]~input
@O_1132.O_1139
Y[48]~input
Y[49]~input
Y[50]~input
Y[51]~input
@O_1158.O_1165
Y[52]~input
Y[53]~input
Y[54]~input
Y[55]~input
Y[56]~input
Y[57]~input
Y[58]~input
Y[9]~input
@O_888.O_895
Y[10]~input
Y[59]~input
@O_1208.O_1215
Y[60]~input
Y[61]~input
Y[62]~input
Y[63]~input
~ALTERA_ASDO_DATA1~
~ALTERA_ASDO_DATA1~~ibuf
~ALTERA_DATA0~
~ALTERA_DATA0~~ibuf
~ALTERA_DCLK~~obuf
IO_OBUF
~ALTERA_DCLK~
~ALTERA_FLASH_nCE_nCSO~
~ALTERA_FLASH_nCE_nCSO~~ibuf
~ALTERA_nCEO~~obuf
~ALTERA_nCEO~
IO_OBUF (OPEN DRAIN)

Figure 2.1.3: Technology Viewer View of Baseline on a Cyclone IV FPGA. Each logic element (LE) represents part of the full-adder structure, including sum and carry logic. The figure highlights the sequential propagation of the carry chain across the LEs, which forms the critical path and dominates the worst-case timing. This visual representation confirms how the synthesized VHDL design is physically implemented on the FPGA.

2.2 Design Candidate 2: Conditional Sum Adder (Cyclone IV)

The conditional sum adder network (CSAN) is based on the carry-select design principle. Unlike a RCAN where a carry bit propagates throughout the entire circuit, a CSAN assumes both a carry bit of 0 and 1 and computes the summation in parallel for both outcomes.

The following implementation uses a recursive tree structure. An N-bit operation is divided into two halves. The summation of the bottom N/2 bits takes the actual value of the carry in. The upper N/2 bits are computed in parallel; one with an assumed carry-in of 0 and the other with a carry-in of 1. The decomposition is applied recursively by splitting an N bit number into N/2 bits until a base case is reached (ie. a 1 bit full adder). A multiplexer then selects which upper half summation to be used, based on the value of the carry out from the lower half.

Content Rectangle 2.2.1: Generates of architecture of CSAN that recursively splits an N bit number into 2 and computes their sum:

```
gen: IF N > C GENERATE
       left_half_upper:
              ENTITY work.CSAN(LogicFuncCSAN)
              GENERIC MAP (N => N/2)
              PORT MAP(X=>X(N-1 DOWNTO (N+1)/2), Y=>Y(N-1 DOWNTO (N+1)/2), S=>SwC0,
Cin => '0', Cout => CwC0);--, Ovfl => Ovfl0);

       left_half_lower:
              ENTITY work.CSAN(LogicFuncCSAN)
              GENERIC MAP (N => N/2)
              PORT MAP(X=>X(N-1 DOWNTO (N+1)/2), Y=>Y(N-1 DOWNTO (N+1)/2), S=>SwC1,
Cin => '1', Cout => CwC1);--, Ovfl => Ovfl1);

       Int0 <= CWC0 & SwC0;
       Int1 <= CWC1 & SwC1;

       mux:
              ENTITY work.Mux2cNb
              GENERIC MAP(N => N/2 + 1, C => 1)
              PORT MAP(x1 =>      Int0, x2 => Int1, s => tempC, y => MuxResult);
              Cout <= MuxResult((N+1)/2);
              S(N-1 DOWNTO (N+1)/2) <= MuxResult((N-1)/2 DOWNTO 0);
       --muxOvfl:
```

```
            --ENTITY work.Mux2c1b
            --PORT MAP(x1 => Ovfl0, x2 => Ovfl1, s => tempC, y => Ovfl);

      right_half:
            ENTITY work.CSAN(LogicFuncCSAN)
            GENERIC MAP (N => N/2)
            PORT MAP(
            X=>X((N-1)/2 DOWNTO 0),
            Y=>Y((N-1)/2 DOWNTO 0),
            S=>SR,
            Cin => Cin,
            Cout => tempC
            );
            S((N-1)/2 DOWNTO 0) <= SR;
Ovfl <= (X(N-1) AND Y(N-1) AND (NOT MuxResult((N-1)/2))) OR (NOT (X(N-1)) AND (NOT
Y(N-1)) AND (MuxResult((N-1)/2)));
END GENERATE gen;
genbase: IF N <= C GENERATE
      -- Use base full adder
      base_case_fa:
            ENTITY work.FA(LogicFuncFA)
            GENERIC MAP (C => N)
            PORT MAP(X=>X(C-1 DOWNTO 0), Y=>Y(C-1 DOWNTO 0), S=>S(C-1 DOWNTO 0),
Cin => Cin, Cout => Cout, Ovfl => Ovfl);
END GENERATE genbase;
```

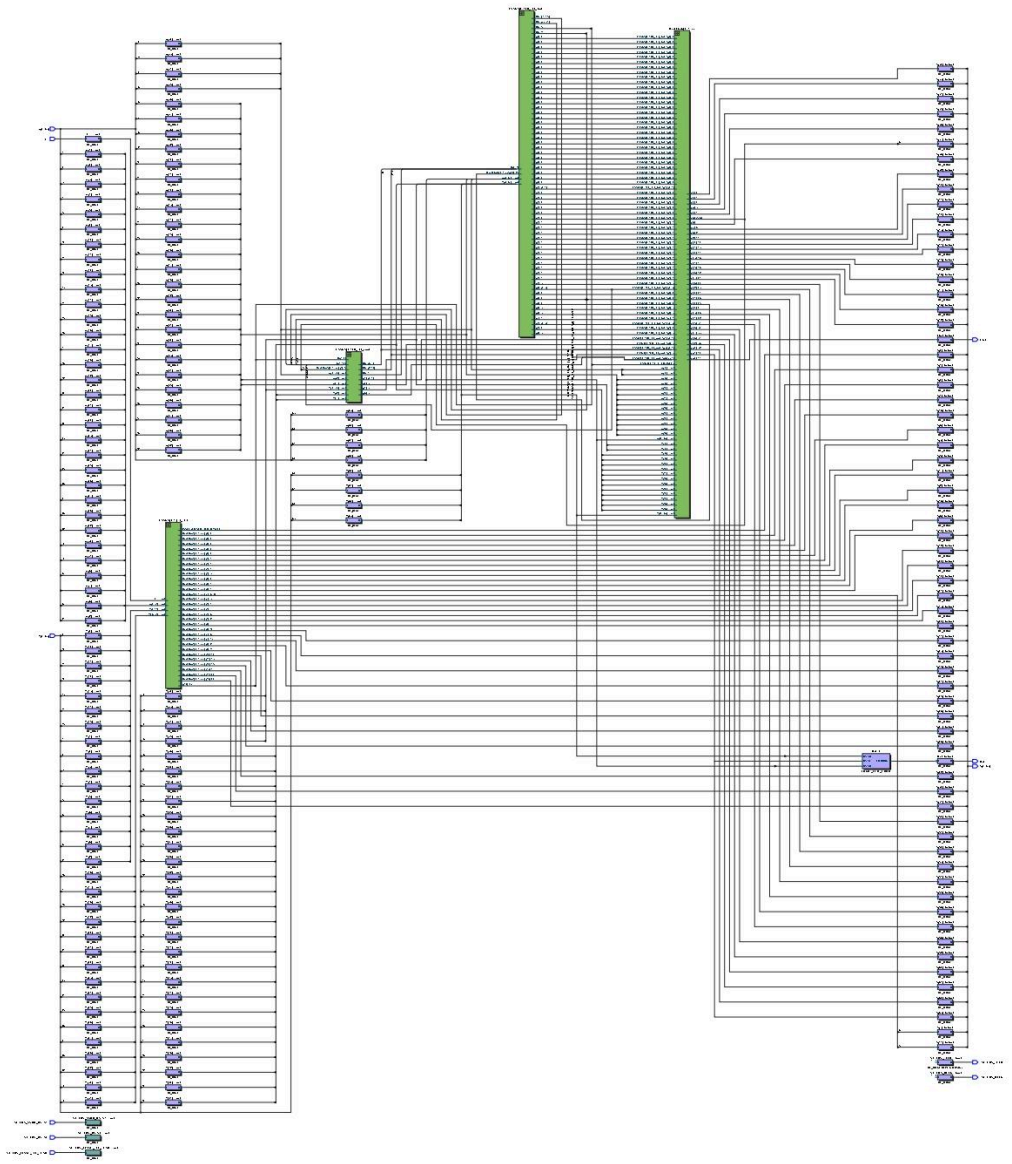


Figure 2.2.3: Technology Viewer View of CSAN Candidate on a Cyclone IV FPGA. Each logic element (LE) used by the recursive N/2-bit adders and multiplexers is displayed, illustrating the placement and interconnection of combinational blocks. The purple boxes show the parallel computation of sums for both carry-in values and the routing of carry signals through the multiplexer selection logic.

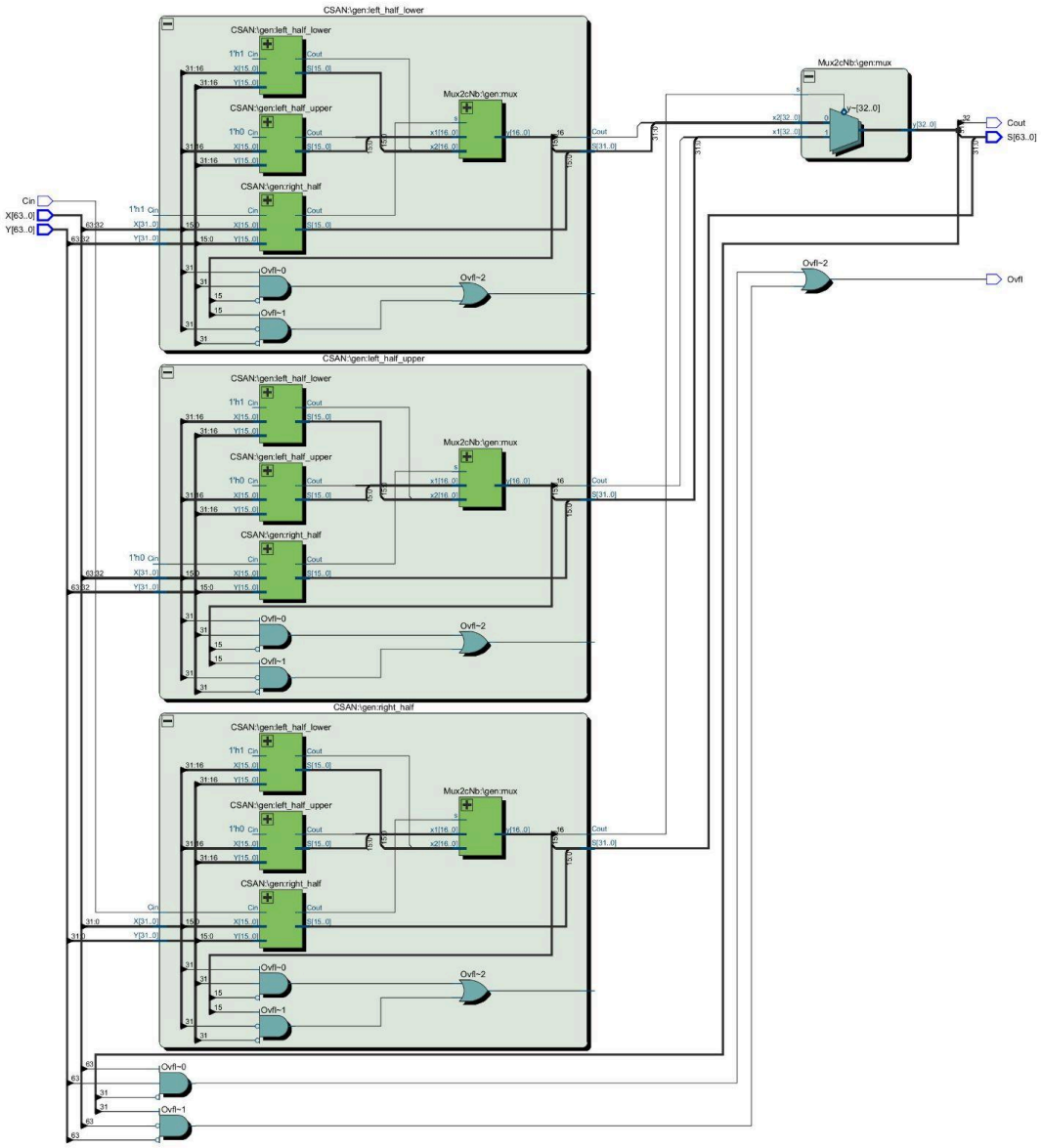
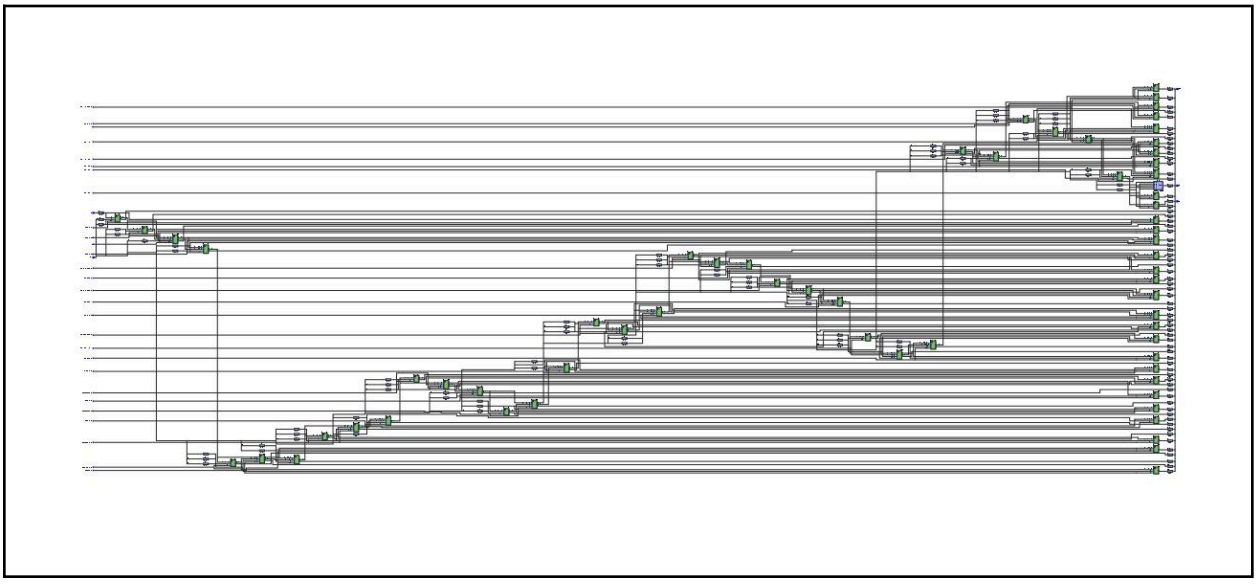


Figure 2.2.4: RTL Viewer of CSAN Candidate, which shows the N-bit number being recursively split into N/2-bit full adders (green boxes) and a multiplexer that selects the appropriate upper-half sum based on the carry-out from the lower-half computation

2.3 Design Candidate 3: Ripple Carry Adder - Baseline (Arria II)

Design candidate 3 uses the same topology as the baseline candidate. The 1-bit full adder is instantiated N times to form an N-bit RCAN with the same overflow detected logic. This candidate is instead synthesized on an Arria II FPGA. On a Cyclone IV, Quartus maps the RCAN topology to logic elements (LEs). However, an Arria II instead uses Adaptive logic modules (ALMs) which use 8-bit adaptive look-up tables (LUTs) rather than 4-bit LUTs and D-flip flops.



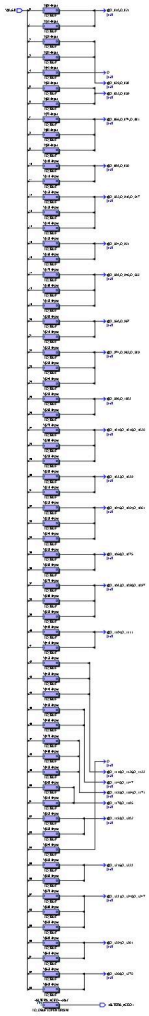


Figure 2.3.1: Technology Viewer View of the RCAN on an Arria II FPGA. Adaptive Logic Modules (ALMs) are used instead of Cyclone IV LEs. The figure illustrates how each ALM implements multiple logic functions and flip-flops, optimizing both resource utilization and carry-chain performance. Compared to the Cyclone IV implementation, the Arria II mapping demonstrates a shorter critical path and more compact logic placement, contributing to the improved timing performance

2.4 Design Candidate 4: Conditional Sum Adder (Arria II)

Design Candidate 4 implements the same Conditional Sum Adder (CSAN) topology and RTL description used in Design Candidate 2. The hierarchical structure, VHDL code, and overall architecture remain unchanged. This candidate will instead be implemented on an Arria II as a similar way to design candidate 3.

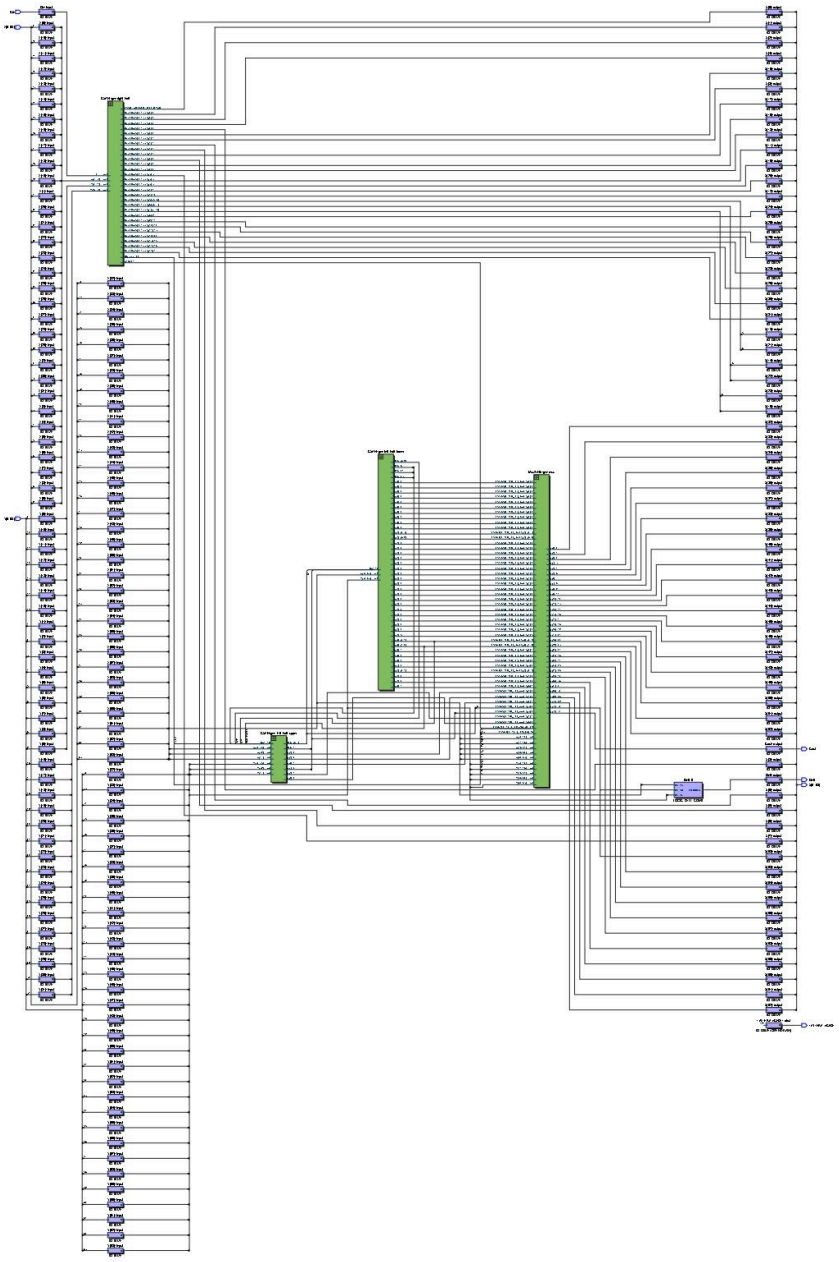


Figure 2.4.1: Technology Viewer View of the CSAN on an Arria 2 FPGA. The adaptive logic modules (ALMs) used for each 1-bit full adder are shown, along with the carry-chain connections that propagate signals sequentially through the adder. The viewer shows the physical placement of combinational logic and the routing paths, providing info on resource utilization and critical paths

3.0 - Summary of Results

The purpose of this section is to compare the remaining design candidates in terms of cost/resource utilization and performance. The required FPGA resources are evaluated by examining the number of Logic Elements (LEs) and Adaptive Logic Modules (ALMs) used by each implementation. Performance is assessed using timing analysis results, including maximum propagation delay.

3.1 Performance Comparison

The adder topologies are purely combinational and contain no clocked elements. The timing analyzer in Quartus can compute worst case timing through every input to every output. The worst case propagation is assumed to be from Cin to Cout.

3.1.1: Baseline device

The baseline device consists of a 64-bit ripple carry adder implemented on a Cyclone IV FPGA. The critical path is formed by the sequential propagation of the carry signal through all 64 full-adder stages. As a result, the propagation delay increases linearly with operand width. Timing analysis indicates that the maximum combinational delay is dominated by the carry chain, limiting the achievable operating frequency. While the topology minimizes logic resource usage, it exhibits the poorest timing performance among the evaluated candidates.

3.1.2: Design Candidates 2,3,4:

The CSAN demonstrates improved performance compared to the ripple-carry baseline due to reduced carry propagation depth. By separating the adder into blocks and computing results in parallel for both possible carry inputs, the critical path is shortened to the delay of a single block plus multiplexer selection logic. This results in a reduced combinational delay relative to the RCAN topology.

When implemented on the Arria II device, both RCAN and CSAN exhibit improved performance compared to their implementation on a Cyclone IV. This improvement is attributed to the enhanced adaptive logic module (ALM) architecture, improved routing resources, and optimized carry-chain structures of the Arria II FPGA family.

Table 3.1.3: Performance of each design candidate based on its worst case propagation delays

	FPGA:	**Worst case propagation delay (in ns)**
(Baseline) - RCAN (Cyclone IV)	Cyclone IV	58.677 ns
CSAN (Cyclone IV)	Cyclone IV	44.863 ns
RCAN (Arria II)	Arria II	44.942 ns
CSAN (Arria II)	Arria II	31.499 ns

3.2 Cost Comparison

The cost is defined as a FPGA resource complexity, using Cyclone IV logic elements (LEs) as the standard unit. Actual Cyclone costs were taken directly from the Quartus fit summary as Total logic elements. For the Arria II, the Quartus compilation reports combinational ALUTs, so ALMs were estimated as [ALUTs/2]. The Arria II cost was then converted to Cyclone-equivalent LEs using the chosen conversion factor k = 1.5. Finally, all costs were normalized by dividing by the Cyclone baseline ripple cost (155 LEs) to enable direct comparison between design candidates.

3.2.0: Baseline Design:

The baseline RCAN on Cyclone IV serves as the reference for normalization. Its measured resource usage provides a benchmark to evaluate other designs.

3.2.1: Design Candidates 2,3,4:

The FPGA resource utilization of the three remaining adder design candidates was also obtained from the Quartus compilation reports. Resource usage was evaluated in terms of LEs for the Cyclone IV device and ALMs for the Arria II device.

3.2.2: Comparing Cyclone IV's LEs and Arria II's ALMs

Since an ALM provides greater functional flexibility and can implement multiple logic functions that would require several Cyclone IV LEs, the cost estimation method is to compare device utilization for identical synthesized designs and derive an equivalence ratio.

Table 3.2.3: Table of costs for comparison between design candidates

	Predicted number of logic elements (LE)/ ALMs	**Actual number of LEs/ALMs used**	**Cost of each**	**Normalized cost (compared to Baseline RCAN On a Cyclone IV)**
(Baseline) RCAN (Cyclone IV)	64 LEs	155 LEs	155	1.00
CSAN (Cyclone IV)	1758 LEs	291 LEs	291	1.88
RCAN (Arria II)	43 ALMs	133 ALUTS = 67 ALMs	100.5	0.65
CSAN (Arria II)	1172 ALMs	209 ALUTS = 105 ALMs	157.5	1.02

3.3 Concluding Statement

In this project, four design candidates were synthesized and analyzed across two FPGAs: RCAN (baseline) and CSAN on Cyclone IV and on Arria II. The evaluation focused on timing performance and FPGA resource cost, including LUTs, ALMs, and worst-case propagation delays. On Cyclone IV devices, RCAN is simple and area-efficient, while CSAN reduces latency at the expense of extra LUTs. On an Arria II, RCAN and CSAN benefit from faster propagation and higher frequency. CSAN achieves the best timing, though resource usage is proportionally higher. The choice between RCAN and CSAN depends on whether low latency or minimal FPGA resource usage is the design priority. Normalization against the baseline confirms that the Arria II implementations provide a clear speed advantage, while the Cyclone IV designs illustrate tradeoffs between complexity and timing.

4.0 Appendix

4.1 VHDL Listing - DP1-VHDL-Listing-G12-350-1261.pdf

The full PDF document containing all VHDL sourcecode, with coloured syntax highlighting and line numbers, is located in the ./Documentation directory, named:
“DP1-VHDL-Listing-G12-350-1261.pdf”.
In it contains the complete vhdl source code for this project.