
-----Start of DP1-VHDL-Listing-G12-350-1261.pdf
-----Notes for this listing:
-----1. Unnecessary, and commented out code has been removed for this vhdl listing to improve conciseness
-----2. VHDL files will be listed in the order of high-level entities to low-level entities to place importance on high level entities

-----Start of TBCSAN.vhd.
-----Note: Identical to TBRCAN.vhd other than the entity declaration, and CSAN->RCAN,
LogicFuncCSAN->Baseline.

1. -- Library required for the "append" utility function
2. **use** work.Utils.all;
3. -- TB ENTITY declaration
4. **ENTITY** TBCSAN **IS**
5. **GENERIC** (N:NATURAL := 64);
6. **END ENTITY** TBCSAN;
7. -- TB ARCHITECTURE Declaration
8. **ARCHITECTURE** TestCSAN **OF** TBCSAN **IS**
9. -- Constants
10. -- Test Vector File
11. **CONSTANT** TestVectorFile : STRING := "Adder00.tvs";
12. -- Pre-Stimulation Delay
13. **CONSTANT** PreStimTime: TIME := 1ns;
14. -- Time to wait after stimulation for outputs to settle. This should be arbitrary: it's a topology so delays are nonexistent
15. **CONSTANT** PostStimTime: TIME := 10ns;
16. -- Newline character
17. **CONSTANT** NL : string(1 to 1) := (1 => CHARACTER'VAL(10));
18. -- Test Record Declaration
19. **TYPE** TestVectorOp **IS RECORD**
20. -- Inputs
21. inX: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
22. inY: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
23. inC: STD_LOGIC;
24. -- Outputs
25. outS: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
26. outC: STD_LOGIC;
27. outOvfl: STD_LOGIC;
28. **END RECORD** TestVectorOp;
29. -- Files
30. **FILE** InputFile: TEXT **OPEN READ_MODE IS** "./TestVectors/" & TestVectorFile;
31. -- Signals
32. -- Testbench Signals

```

33. SIGNAL TBX: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
34. SIGNAL TBY: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
35. SIGNAL TBC: STD_LOGIC;
36. -- DUT Signals
37. SIGNAL DUT_S: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
38. SIGNAL DUT_Cout: STD_LOGIC;
39. SIGNAL DUT_Ovfl: STD_LOGIC;
40. BEGIN
41. -- Actual DUT Instance. This would be RCAN in the TBRCAN.vhd file
42. DUT: ENTITY WORK.CSAN(LogicFuncCSAN) -- DUT: ENTITY WORK.RCAN(Baseline) in TBRCAN.vhd
43. GENERIC MAP (N => N)
44. PORT MAP (
45. X => TBX,
46. Y => TBY,
47. Cin => TBC,
48. S => DUT_S,
49. Cout => DUT_Cout,
50. Ovfl => DUT_Ovfl
51. );
52. main:
53. PROCESS
54. -- Line we are currently processing
55. VARIABLE CurrentLine: LINE;
56. VARIABLE TranscriptLine: LINE;
57. -- The TestVector
58. VARIABLE TV: TestVectorOp;
59. -- Index
60. VARIABLE MeasurementIndex : INTEGER := 1;
61. -- For holding each segment of the line to set the TV
62. VARIABLE TempHex : STRING(1 to 16);
63. -- Reason blurb in case of failure
64. VARIABLE ReasonStr : STRING(1 to 512);
65. -- Position index used in the append function
66. VARIABLE p : NATURAL := 1;
67. -- For Cin, Cout, Ovfl
68. VARIABLE TempBit : STD_LOGIC;
69. -- Used in between each readline from the test vector file due to the spacing requirement in the
test vector specifications
70. VARIABLE TempChar : CHARACTER;
71. -- To keep track of whether the DUT passed the current Test Vector
72. VARIABLE TVPassed : BOOLEAN;
73. BEGIN
74. -- Loop until the end of the test vector file
75. WHILE NOT ENDFILE(InputFile) LOOP
76. ReasonStr := (OTHERS => ' ');
77. p := 1;
78. TVPassed := TRUE;

```

```

79. READLINE(InputFile, CurrentLine);
80. READ(CurrentLine, TempHex);
81. TV.inX := hex_to_slv(TempHex, 64);
82. READ(CurrentLine, TempChar);
83. READ(CurrentLine, TempHex);
84. TV.inY := hex_to_slv(TempHex, 64);
85. READ(CurrentLine, TempChar);
86. READ(CurrentLine, TempBit);
87. TV.inC := TempBit;
88. READ(CurrentLine, TempChar);
89. READ(CurrentLine, TempHex);
90. TV.outS := hex_to_slv(TempHex, 64);
91. READ(CurrentLine, TempChar);
92. READ(CurrentLine, TempBit);
93. TV.outC := TempBit;
94. READ(CurrentLine, TempChar);
95. READ(CurrentLine, TempBit);
96. TV.outOvfl := TempBit;
97. -- Now, to actually care about the DUT.
98. -- Apply 'X' to all input bits, hold for PreStimTime
99. TBX <= (OTHERS => 'X');
100. TBY <= (OTHERS => 'X');
101. TBC <= 'X';
102. WAIT FOR PreStimTime;
103. -- Apply stimuli until outputs are stable
104. TBX <= TV.inX;
105. TBY <= TV.inY;
106. TBC <= TV.inC;
107. WAIT FOR PostStimTime;
108. -- Verify correct results, append message to reason blurb on failure
109. IF DUT_S /= TV.outS THEN
110. TVPassed := FALSE;
111. work_Utils.append(ReasonStr, p, "-|Sum Mismatch|");
112. END IF;
113. IF DUT_Cout /= TV.outC THEN
114. TVPassed := FALSE;
115. work_Utils.append(ReasonStr, p, "-|Cout Mismatch|");
116. END IF;
117. IF DUT_Ovfl /= TV.outOvfl THEN
118. TVPassed := FALSE;
119. work_Utils.append(ReasonStr, p, "-|Ovfl Mismatch|");
120. END IF;
121. -- One-liner describing the test vector result
122. IF TVPassed = FALSE THEN
123. REPORT "Measurement #" & INTEGER'IMAGE(MeasurementIndex) & " Failed." &
124.      " Reason:" & ReasonStr(1 to p-1) &
125.      " Stimulus:" &

```

```

126.    " [A: " & slv_to_hex(TV.inX) &
127.    " B: " & slv_to_hex(TV.inY) &
128.    " Cin: " & INTEGER'IMAGE(conv_integer(TV.inC)) & "]" &
129.    " Expected Outputs:" &
130.    " [S: " & slv_to_hex(TV.outS) &
131.    " Cout: " & INTEGER'IMAGE(conv_integer(TV.outC)) &
132.    " Ovfl: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) & "]" &
133.    " Actual Outputs:" &
134.    " [S: " & slv_to_hex(DUT_S) &
135.    " Cout: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) &
136.    " Ovfl: " & INTEGER'IMAGE(conv_integer(DUT_Ovfl)) & "]";
137. ELSE
138. REPORT "Measurement #" & INTEGER'IMAGE(MeasurementIndex) & " Passed." &
139.    " Stimulus:" &
140.    " [A: " & slv_to_hex(TV.inX) &
141.    " B: " & slv_to_hex(TV.inY) &
142.    " Cin: " & INTEGER'IMAGE(conv_integer(TV.inC)) & "]" &
143.    " Expected Outputs:" &
144.    " [S: " & slv_to_hex(TV.outS) &
145.    " Cout: " & INTEGER'IMAGE(conv_integer(TV.outC)) &
146.    " Ovfl: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) & "]" &
147.    " Actual Outputs:" &
148.    " [S: " & slv_to_hex(DUT_S) &
149.    " Cout: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) &
150.    " Ovfl: " & INTEGER'IMAGE(conv_integer(DUT_Ovfl)) & "]";
151. END IF;
152. -- Increment measurement index for next measurement
153. MeasurementIndex := MeasurementIndex + 1;
154. END LOOP;
155. -- Finishing wait due to this process not containing an initializer list. Without this we would loop
the process infinitely
156. WAIT;
157. END PROCESS main;
158. END TestCSAN;
-----  

-----End of TBCSAN.vhd  

-----
```

-----Start of TBRCAN.vhd.

1. -- Library required for the "append" utility function
2. use work_Utils.all;
3. -- TB ENTITY declaration
4. ENTITY TBRCAN IS
5. GENERIC (N:NATURAL := 64);
6. END ENTITY TBRCAN;

```

7. -- TB ARCHITECTURE Declaration
8. ARCHITECTURE TestRCAN OF TBRCAN IS
9. -- Constants
10. -- Test Vector File
11. CONSTANT TestVectorFile : STRING := "Adder00.tvs";
12. -- Pre-Stimulation Delay
13. CONSTANT PreStimTime: TIME := 1ns;
14. -- Time to wait after stimulation for outputs to settle. This should be arbitrary: it's a topology so
delays are nonexistent
15. CONSTANT PostStimTime: TIME := 10ns;
16. -- Newline character
17. CONSTANT NL : string(1 to 1) := (1 => CHARACTER'VAL(10));
18. -- Test Record Declaration
19. TYPE TestVectorOp IS RECORD
20. -- Inputs
21. inX: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
22. inY: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
23. inC: STD_LOGIC;
24. -- Outputs
25. outS: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
26. outC: STD_LOGIC;
27. outOvfl: STD_LOGIC;
28. END RECORD TestVectorOp;
29. -- Files
30. FILE InputFile: TEXT OPEN READ_MODE IS "./TestVectors/" & TestVectorFile;
31. -- Signals
32. -- Testbench Signals
33. SIGNAL TBX: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
34. SIGNAL TBY: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
35. SIGNAL TBC: STD_LOGIC;
36. -- DUT Signals
37. SIGNAL DUT_S: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
38. SIGNAL DUT_Cout: STD_LOGIC;
39. SIGNAL DUT_Ovfl: STD_LOGIC;
40. BEGIN
41. -- Actual DUT Instance. This would be RCAN in the TBRCAN.vhd file
42. DUT: ENTITY WORK.RCAN(Baseline) -- DUT: ENTITY WORK.RCAN(Baseline) in TBRCAN.vhd
43. GENERIC MAP (N => N)
44. PORT MAP (
45. X => TBX,
46. Y => TBY,
47. Cin => TBC,
48. S => DUT_S,
49. Cout => DUT_Cout,
50. Ovfl => DUT_Ovfl
51. );
52. main:

```

53. **PROCESS**
 54. -- Line we are currently processing
 55. **VARIABLE** CurrentLine: LINE;
 56. **VARIABLE** TranscriptLine: LINE;
 57. -- The TestVector
 58. **VARIABLE** TV: TestVectorOp;
 59. -- Index
 60. **VARIABLE** MeasurementIndex : INTEGER := 1;
 61. -- For holding each segment of the line to set the TV
 62. **VARIABLE** TempHex : STRING(1 to 16);
 63. -- Reason blurb in case of failure
 64. **VARIABLE** ReasonStr : STRING(1 to 512);
 65. -- Position index used in the append function
 66. **VARIABLE** p : NATURAL := 1;
 67. -- For Cin, Cout, Ovfl
 68. **VARIABLE** TempBit : STD_LOGIC;
 69. -- Used in between each readline from the test vector file due to the spacing requirement in the test vector specifications
 70. **VARIABLE** TempChar : CHARACTER;
 71. -- To keep track of whether the DUT passed the current Test Vector
 72. **VARIABLE** TVPassed : BOOLEAN;
 73. **BEGIN**
 74. -- Loop until the end of the test vector file
 75. **WHILE NOT ENDFILE**(InputFile) **LOOP**
 76. ReasonStr := (OTHERS => ' ');\n 77. p := 1;
 78. TVPassed := TRUE;
 79. **READLINE**(InputFile, CurrentLine);
 80. **READ**(CurrentLine, TempHex);
 81. TV.inX := hex_to_slv(TempHex, 64);
 82. **READ**(CurrentLine, TempChar);
 83. **READ**(CurrentLine, TempHex);
 84. TV.inY := hex_to_slv(TempHex, 64);
 85. **READ**(CurrentLine, TempChar);
 86. **READ**(CurrentLine, TempBit);
 87. TV.inC := TempBit;
 88. **READ**(CurrentLine, TempChar);
 89. **READ**(CurrentLine, TempHex);
 90. TV.outS := hex_to_slv(TempHex, 64);
 91. **READ**(CurrentLine, TempChar);
 92. **READ**(CurrentLine, TempBit);
 93. TV.outC := TempBit;
 94. **READ**(CurrentLine, TempChar);
 95. **READ**(CurrentLine, TempBit);
 96. TV.outOvfl := TempBit;
 97. -- Now, to actually care about the DUT.
 98. -- Apply 'X' to all input bits, hold for PreStimTime

```

99. TBX <= (OTHERS => 'X');
100. TBY <= (OTHERS => 'X');
101. TBC <= 'X';
102. WAIT FOR PreStimTime;
103. -- Apply stimuli until outputs are stable
104. TBX <= TV.inX;
105. TBY <= TV.inY;
106. TBC <= TV.inC;
107. WAIT FOR PostStimTime;
108. -- Verify correct results, append message to reason blurb on failure
109. IF DUT_S /= TV.outS THEN
110. TVPassed := FALSE;
111. work_Utils.append(ReasonStr, p, "-| Sum Mismatch |-");
112. END IF;
113. IF DUT_Cout /= TV.outC THEN
114. TVPassed := FALSE;
115. work_Utils.append(ReasonStr, p, "-| Cout Mismatch |-");
116. END IF;
117. IF DUT_Ovfl /= TV.outOvfl THEN
118. TVPassed := FALSE;
119. work_Utils.append(ReasonStr, p, "-| Ovfl Mismatch |-");
120. END IF;
121. -- One-liner describing the test vector result
122. IF TVPassed = FALSE THEN
123. REPORT "Measurement #" & INTEGER'IMAGE(MeasurementIndex) & " Failed." &
124.   " Reason:" & ReasonStr(1 to p-1) &
125.   " Stimulus:" &
126.   " [A: " & slv_to_hex(TV.inX) &
127.   " B: " & slv_to_hex(TV.inY) &
128.   " Cin: " & INTEGER'IMAGE(conv_integer(TV.inC)) & "]" &
129.   " Expected Outputs:" &
130.   " [S: " & slv_to_hex(TV.outS) &
131.   " Cout: " & INTEGER'IMAGE(conv_integer(TV.outC)) &
132.   " Ovfl: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) & "]" &
133.   " Actual Outputs:" &
134.   " [S: " & slv_to_hex(DUT_S) &
135.   " Cout: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) &
136.   " Ovfl: " & INTEGER'IMAGE(conv_integer(DUT_Ovfl)) & "]";
137. ELSE
138. REPORT "Measurement #" & INTEGER'IMAGE(MeasurementIndex) & " Passed." &
139.   " Stimulus:" &
140.   " [A: " & slv_to_hex(TV.inX) &
141.   " B: " & slv_to_hex(TV.inY) &
142.   " Cin: " & INTEGER'IMAGE(conv_integer(TV.inC)) & "]" &
143.   " Expected Outputs:" &
144.   " [S: " & slv_to_hex(TV.outS) &
145.   " Cout: " & INTEGER'IMAGE(conv_integer(TV.outC)) &

```

```

146. " Ovfl: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) & "]" &
147. " Actual Outputs:" &
148. " [S: " & slv_to_hex(DUT_S) &
149. " Cout: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) &
150. " Ovfl: " & INTEGER'IMAGE(conv_integer(DUT_Ovfl)) & "]";
151. END IF;
152. -- Increment measurement index for next measurement
153. MeasurementIndex := MeasurementIndex + 1;
154. END LOOP;
155. -- Finishing wait due to this process not containing an initializer list. Without this we would loop
the process infinitely
156. WAIT;
157. END PROCESS main;
158. END TestRCAN;
-----  

-----End of TBRCAN.vhd  

-----  


```

```

-----  

-----Start of CSAN.vhd  

-----  

1. -- CSAN entity:  

2. -- N: Number of bits of operands (and sum)  

3. -- C: Recursion base case adder. For instance, if N=64, and C=4, recursive principle of CSA will be
applied until N=4.  

4. -- X: Operand 1 (A)  

5. -- Y: Operand 2 (B)  

6. -- Cin: Carry-in  

7. -- S: Sum  

8. -- Cout: Carry-out  

9. -- Ovfl: True if overflow occurred in the addition of signed operands  

10. ENTITY CSAN IS  

11. GENERIC (N: NATURAL := 64;  

12.     C: NATURAL := 1);  

13. PORT (  

14.     X: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);  

15.     Y: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);  

16.     S: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);  

17.     Cin: IN STD_LOGIC;  

18.     Cout: OUT STD_LOGIC;  

19.     Ovfl: OUT STD_LOGIC);  

20. END CSAN;  

21. ARCHITECTURE LogicFuncCSAN OF CSAN IS  

22. -- signal decs  

23. -- Carries of the upper and lower adders on left half  

24. SIGNAL CwC0, CwC1: STD_LOGIC;
```

```

25. -- Sums of the upper and lower adders on left half, as well as sum of right half
26. SIGNAL SwC0, SwC1, SR: STD_LOGIC_VECTOR((N-1)/2 DOWNTO 0);
27. -- Overflow of upper and lower adders on left half (Not currently in use)
28. SIGNAL Ovfl0, Ovfl1: STD_LOGIC;
29. -- Intermediate signals, feeding into the N/2 + 1 bit MUX. Concatenation of carryout of left half
   adders and their sums
30. SIGNAL Int0, Int1: STD_LOGIC_VECTOR((N+1)/2 DOWNTO 0);
31. -- Result of N/2 + 1 bit MUX. Select bit is carry from right half, inputs are the intermediate signals
32. SIGNAL MuxResult: STD_LOGIC_VECTOR((N+1)/2 DOWNTO 0);
33. -- Temporary Carryout, Overflow signals. (tempO not currently in use)
34. SIGNAL tempC, tempO: STD_LOGIC;
35. BEGIN
36. -- C equals 1 for usage of standard full adders in base case. For non-leaf nodes, divide into left
   and right halves
37. gen: IF N > C GENERATE
38. -- Left half adder with Cin = 0
39.   left_half_upper:
40.     ENTITY work.CSAN(LogicFuncCSAN)
41.     GENERIC MAP (N => N/2)
42.     PORT MAP(X=>X(N-1 DOWNT0 (N+1)/2, Y=>Y(N-1 DOWNT0 (N+1)/2, S=>SwC0, Cin => '0',
   Cout => CwC0);-- Ovfl => Ovfl0);
43. -- Left half adder with Cin = 1
44.   left_half_lower:
45.     ENTITY work.CSAN(LogicFuncCSAN)
46.     GENERIC MAP (N => N/2)
47.     PORT MAP(X=>X(N-1 DOWNT0 (N+1)/2, Y=>Y(N-1 DOWNT0 (N+1)/2, S=>SwC1, Cin => '1',
   Cout => CwC1);-- Ovfl => Ovfl1);
48. -- Intermediate (Concatenated Cout, Sums) signals of leftmost adders
49.   Int0 <= CwC0 & SwC0;
50.   Int1 <= CwC1 & SwC1;
51. -- N/2 + 1 bit, 2 channel MUX to select which carry, sum will be the output for the left half.
52. -- If we think of this as a divide and conquer algorithm, this is exactly the merging step.
53. mux:
54.   ENTITY work.Mux2cNb
55.   GENERIC MAP(N => N/2 + 1, C => 1)
56.   PORT MAP(x1 => Int0, x2 => Int1, s => tempC, y => MuxResult);
57.   -- Ultimate Cout
58.   Cout <= MuxResult((N+1)/2);
59.   -- Upper half of sum bits
60.   S(N-1 DOWNT0 (N+1)/2) <= MuxResult((N-1)/2 DOWNT0 0);
61. --muxOvfl:
62.   --ENTITY work.Mux2c1b
63.   --PORT MAP(x1 => Ovfl0, x2 => Ovfl1, s => tempC, y => Ovfl);
64.
65. -- Right half
66. right_half:
67.   ENTITY work.CSAN(LogicFuncCSAN)

```

```

68.  GENERIC MAP (N => N/2)
69.  PORT MAP(
70.    X=>X((N-1)/2 DOWNTO 0),
71.    Y=>Y((N-1)/2 DOWNTO 0),
72.    S=>SR,
73.    Cin => Cin,
74.    Cout => tempC
75.  );
76.  -- Lower half of sum bits
77.  S((N-1)/2 DOWNTO 0) <= SR;
78. -- Ovfl logic: if X,Y are positive (MSB 0), and S is negative (MSB 1), then overflow has occurred.
Likewise, overflow is true if X,Y are negative (MSB 1) and S is positive (MSB 0).
79. 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)));
80.
81. END GENERATE gen;
82.
83. -- Recursion base case: just use a full adder
84. genbase: IF N <= C GENERATE
85.   -- Use base full adder
86.   base_case_fa:
87.     ENTITY work.FA(LogicFuncFA)
88.     GENERIC MAP (C => N)
89.     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);
90. END GENERATE genbase;
91. END LogicFuncCSAN;
-----
```

-----End of CSAN.vhd

-----Start of RCAN.vhd

-----Note: FullAddr entities are logically Identical to FA entities.

1. -- Take the FullAddr entity to make 64 full adders into a ripple adder
2. -- RCAN entity:
3. -- N: Number of bits of operands (and sum)
4. -- C: Recursion base case adder. For instance, if N=64, and C=4, recursive principle of CSA will be applied until N=4.
5. -- X: Operand 1 (A)
6. -- Y: Operand 2 (B)
7. -- Cin: Carry-in
8. -- S: Sum
9. -- Cout: Carry-out
10. -- Ovfl: True if overflow occurred in the addition of signed operands
11. Entity RCAN is

```

12. generic(
13.   N : natural := 64
14. );
15. port (
16.   X, Y : in std_logic_vector(N-1 downto 0);
17.   Cin : in std_logic;
18.   S : out std_logic_vector(N-1 downto 0);
19.   Cout : out std_logic;
20.   Ovfl : out std_logic
21. );
22. end RCAN;
23. -- Baseline Adder
24. architecture Baseline of RCAN is
25. signal carries : std_logic_vector(N-1 downto 0);
26. signal tempC : std_logic;
27. begin
28. -- S[0]:
29.   U0 : entity work.FullAddr(behavioural) port map(X => X(0), Y => Y(0), Cin => Cin, S => S(0),
Cout => carries(0));
30. -- S[62:1]:
31.   gen0 : for i in 1 to N-2 generate
32.     U1 : entity work.FullAddr(behavioural) port map(X => X(i), Y => Y(i), Cin => carries(i-1), S
=> S(i), Cout => carries(i));
33.   end generate gen0;
34. -- S[63]:
35.   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);
36.   Cout <= tempC; --Carry out
37.   Ovfl <= tempC XOR carries(N-2); -- Overflow
38. end architecture;
39. -- Quartus FastRipple:
40. architecture FastRipple of RCAN is
41. begin
42. PROCESS(X, Y)
43. -- Temporary holding variable for the sum
44. VARIABLE tempS : STD_LOGIC_VECTOR(N DOWNTO 0);
45. BEGIN
46. tempS := std_logic_vector((resize(unsigned(X), N+1) + resize(unsigned(Y),N+1) + resize( (
unsigned( std_logic_vector'(0' & Cin)) ),N+1 ) ));
47. Ovfl <= (X(N-1) AND Y(N-1) AND (NOT tempS((N-1)))) OR (NOT (X(N-1)) AND (NOT Y(N-1)) AND
(tempS((N-1)))); 
48. S <= tempS(N-1 DOWNTO 0);
49. Cout <= tempS(N);
50. END PROCESS;
51. end architecture;
-----
-----End of RCAN.vhd

```

```

-----
-----Start of FA.vhd
-----Note: Logically identical to FullAddr.vhd
-----
1. -- FA entity:
2. -- C: For a full adder, this is 1. If an N-bit ripple carry adder is desired, set C to N.
3. -- X: Operand 1 (A)
4. -- Y: Operand 2 (B)
5. -- Cin: Carry-in
6. -- S: Sum
7. -- Cout: Carry-out
8. -- Ovfl: True if overflow occurred in the addition of signed operands
9. GENERIC (
10. C: NATURAL := 1
11. );
12. PORT(
13.   X: IN STD_LOGIC_VECTOR(C-1 DOWNTO 0);
14.   Y: IN STD_LOGIC_VECTOR(C-1 DOWNTO 0);
15.   S: OUT STD_LOGIC_VECTOR(C-1 DOWNTO 0);
16.   Cin: IN STD_LOGIC;
17.   Cout: OUT STD_LOGIC;
18.   Ovfl: OUT STD_LOGIC);
19. END FA;
20. ARCHITECTURE LogicFuncFA OF FA IS
21. -- Basic ripple adder implementation
22. SIGNAL Couts: STD_LOGIC_VECTOR(C DOWNTO 0);
23. BEGIN
24. Couts(0) <= Cin;
25. gen1: FOR i IN 0 TO C-1 GENERATE
26.   Couts(i+1) <= '1' WHEN ((Couts(i) = '1' AND X(i) = '1') OR (Couts(i) = '1' AND Y(i) = '1') OR (X(i) = '1' AND Y(i) = '1')) ELSE
27.     '0';
28.   S(i) <= Couts(i) XOR X(i) XOR Y(i);
29. END GENERATE gen1;
30. Cout <= Couts(C);
31. -- Ovfl logic here is Cn xor Cn-1 because we have access to all carries. Requires less gates.
32. Ovfl <= Couts(C) XOR Couts(C-1);
33. END LogicFuncFA;

```

-----End of FA.vhd

-----Start of FullAddr.vhd

```

1. -- Entity for a 1-bit adder so that the RTL viewer makes it look nice (and not all gates)
2. -- FullAddr entity:
3. -- X: Operand 1 (A)
4. -- Y: Operand 2 (B)
5. -- Cin: Carry-in
6. -- S: Sum
7. -- Cout: Carry-out
8. -- Ovfl: In a 1-bit full adder, this is not very important, so it is omitted.
9. Entity FullAddr is
10. port (
11.   X : in std_logic;
12.   Y : in std_logic;
13.   Cin : in std_logic;
14.   S : out std_logic;
15.   Cout : out std_logic;
16. );
17. end FullAddr;
18. architecture behavioural of FullAddr is
19. begin
20.   S <= (X xor Y) xor Cin;
21.   Cout <= (X and Y) or ((X or Y) and Cin);
22. end architecture;
-----  

-----End of FullAddr.vhd  

-----  

-----  

-----Start of Mux2cNb.vhd  

-----  

1. -- 2-channel, N-bit MUX
2. -- Mux2cNb entity:
3. -- N: Number of bits in inputs and output
4. -- C: Base case mux (should be 1, for 2-channel 1 bit MUX)
5. -- x1: operand 1 (N bits, selected when s = 0)
6. -- x2: operand 2 (N bits, selected when s = 0)
7. -- s: select bit
8. -- y: (N bits) output
9. ENTITY Mux2cNb IS
10. GENERIC (N: NATURAL := 2;
11.           C: NATURAL := 1);
12. PORT (x1, x2: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
13.         s: IN STD_LOGIC;
14.         y: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0)
15.       );
16. END Mux2cNb;
17.  

18. ARCHITECTURE LogicFuncMux2cNb OF Mux2cNb IS

```

```

19. BEGIN
20. -- Will generate N side by side 2-channel 1 bit muxes to create the 2-channel n bit mux
21. gen: FOR i IN 0 TO N-1 GENERATE
22. WITH s SELECT
23. y(i) <= x1(i) WHEN '0',
24.       x2(i) WHEN others;
25.
26. END GENERATE gen;
27. END LogicFuncMux2cNb;

-----
-----End of Mux2cNb.vhd
-----


-----
-----Start of Mux2c1b.vhd
-----

1. -- 2-channel, 1-bit MUX
2. -- Standard 2-channel 1 bit mux. equation: y = s'x1 + sx2
3. -- Mux2c1b entity:
4. -- x1: operand 1 (1 bit, selected when s = 0)
5. -- x2: operand 2 (1 bit, selected when s = 0)
6. -- s: select bit
7. -- y: output (1 bit)
8. ENTITY Mux2c1b IS
9. PORT (
10. x1, x2: IN STD_LOGIC;
11. s: IN STD_LOGIC;
12. y: OUT STD_LOGIC
13. );
14. END Mux2c1b;
15. ARCHITECTURE LogicFunc OF Mux2c1b IS
16. BEGIN
17. WITH s SELECT
18.   y <= x1 WHEN '0',
19.     x2 WHEN others;
20.
21. END LogicFunc;

-----
-----End of Mux2c1b.vhd
-----


-----
-----Start of Utils.vhd
-----Note: This provides utility functions across the workspace
-----

1. PACKAGE Utils IS
2. FUNCTION hex_to_slv(Hex : IN STRING; WIDTH : IN NATURAL) RETURN STD_LOGIC_VECTOR;

```

```

3. FUNCTION slv_to_hex(X: IN STD_LOGIC_VECTOR) RETURN STRING;
4. PROCEDURE append(Buf: INOUT STRING;p: INOUT NATURAL;S: IN STRING);
5. END PACKAGE;
6. PACKAGE BODY Utils IS
7. -- Helper function to convert a hex string to a std logic vector
8. FUNCTION hex_to_slv(Hex : IN STRING; WIDTH: IN NATURAL) RETURN STD_LOGIC_VECTOR IS
9. VARIABLE Result  : STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0) := (OTHERS => '0');
10. VARIABLE V     : UNSIGNED(WIDTH-1 DOWNTO 0) := (OTHERS => '0');
11. BEGIN
12. -- Iterate through all the elements in the hexadecimal string
13. FOR i IN 1 TO Hex'LENGTH LOOP
14.   V := V SLL 4;
15.   -- Put the chunks into the variable V 4 bits by 4 bits
16.   CASE Hex(i) IS
17.     WHEN '0' => V := V OR to_unsigned(0, WIDTH);
18.     WHEN '1' => V := V OR to_unsigned(1, WIDTH);
19.     WHEN '2' => V := V OR to_unsigned(2, WIDTH);
20.     WHEN '3' => V := V OR to_unsigned(3, WIDTH);
21.     WHEN '4' => V := V OR to_unsigned(4, WIDTH);
22.     WHEN '5' => V := V OR to_unsigned(5, WIDTH);
23.     WHEN '6' => V := V OR to_unsigned(6, WIDTH);
24.     WHEN '7' => V := V OR to_unsigned(7, WIDTH);
25.     WHEN '8' => V := V OR to_unsigned(8, WIDTH);
26.     WHEN '9' => V := V OR to_unsigned(9, WIDTH);
27.     WHEN 'A' | 'a' => V := V OR to_unsigned(10, WIDTH);
28.     WHEN 'B' | 'b' => V := V OR to_unsigned(11, WIDTH);
29.     WHEN 'C' | 'c' => V := V OR to_unsigned(12, WIDTH);
30.     WHEN 'D' | 'd' => V := V OR to_unsigned(13, WIDTH);
31.     WHEN 'E' | 'e' => V := V OR to_unsigned(14, WIDTH);
32.     WHEN 'F' | 'f' => V := V OR to_unsigned(15, WIDTH);
33.     WHEN OTHERS => NULL;
34.   END CASE;
35. END LOOP;
36. -- Cast V to an slv and return it
37. Result := STD_LOGIC_VECTOR(V);
38. RETURN Result;
39. END FUNCTION;
40. -- Helper function to convert a std logic vector to a hexadecimal string
41. FUNCTION slv_to_hex(X: IN std_logic_vector) RETURN STRING IS
42. CONSTANT HexChars : string := "0123456789ABCDEF";
43. VARIABLE Hstr : string(1 TO X'LENGTH / 4);
44. VARIABLE Chunk : unsigned(3 DOWNTO 0);
45. BEGIN
46.   FOR i IN 0 TO X'LENGTH/4 - 1 LOOP
47.     -- Get 4 bits at a time. 'HIGH' is the high end of the slv
48.     Chunk := unsigned(X(X'HIGH-i*4 DOWNTO X'HIGH-i*4-3));
49.     -- +1 since VHDL string index starts at 1

```

```

50.      Hstr(i+1) := HexChars(to_integer(Chunk)+1);
51.  END LOOP;
52. RETURN Hstr;
53. END FUNCTION;
54. -- Helper procedure to append a string to a buffer
55. -- Needs to be a procedure, because it needs to modify the Buf in place
56. PROCEDURE append(Buf: INOUT STRING; p: INOUT NATURAL; S: IN STRING) IS
57.   VARIABLE lenS: NATURAL := S'LENGTH;
58.   CONSTANT BUF_LOW: NATURAL := 1;
59.   -- Can be changed around but hopefully this is enough space
60.   CONSTANT BUF_HIGH: NATURAL := 512;
61. BEGIN
62.   -- If the string fits within the buffer
63.   IF p+lenS-1 <= BUF_HIGH THEN
64.     Buf(p TO p+lenS-1) := S;
65.     p := p + lenS;
66.   ELSE
67.     -- If only part of the string fits
68.     Buf(p TO BUF_HIGH) := S(1 TO BUF_HIGH-p+1);
69.     p := BUF_HIGH+1;
70.   END IF;
71. END PROCEDURE;
72. END PACKAGE BODY;

```

-----End of Utils.vhd

-----End of DP1-VHDL-Listing-G12-350-1261.pdf
