```vhdl
----------
----------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.
----------
-- Library required for the "append" utility function
use work.Utils.all;
-- TB ENTITY declaration
ENTITY TBCSAN IS
GENERIC (N:NATURAL := 64);
END ENTITY TBCSAN;
-- TB ARCHITECTURE Declaration
ARCHITECTURE TestCSAN OF TBCSAN IS
-- Constants
-- Test Vector File
CONSTANT TestVectorFile : STRING := "Adder00.tvs";
-- Pre-Stimulation Delay
CONSTANT PreStimTime: TIME := 1ns;
-- Time to wait after stimulation for outputs to settle. This should be arbitrary: it's a topology so delays are nonexistent
CONSTANT PostStimTime: TIME := 10ns;
-- Newline character
CONSTANT NL : string(1 to 1) := (1 => CHARACTER'VAL(10));
-- Test Record Declaration
TYPE TestVectorOp IS RECORD
-- Inputs
inX: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
inY: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
inC: STD_LOGIC;
-- Outputs
outS: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
outC: STD_LOGIC;
outOvfl: STD_LOGIC;
END RECORD TestVectorOp;
-- Files
FILE InputFile: TEXT OPEN READ_MODE IS "./TestVectors/" & TestVectorFile;
-- Signals
-- Testbench Signals
```

```vhdl
SIGNAL TBX: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
SIGNAL TBY: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
SIGNAL TBC: STD_LOGIC;
-- DUT Signals
SIGNAL DUT_S: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
SIGNAL DUT_Cout: STD_LOGIC;
SIGNAL DUT_Ovfl: STD_LOGIC;
BEGIN
-- Actual DUT Instance. This would be RCAN in the TBRCAN.vhd file
DUT: ENTITY WORK.CSAN(LogicFuncCSAN) -- DUT: ENTITY WORK.RCAN(Baseline) in TBRCAN.vhd
GENERIC MAP (N => N)
PORT MAP (
X => TBX,
Y => TBY,
Cin => TBC,
S => DUT_S,
Cout => DUT_Cout,
Ovfl => DUT_Ovfl
);
main:
PROCESS
-- Line we are currently processing
VARIABLE CurrentLine: LINE;
VARIABLE TranscriptLine: LINE;
-- The TestVector
VARIABLE TV: TestVectorOp;
-- Index
VARIABLE MeasurementIndex : INTEGER := 1;
-- For holding each segment of the line to set the TV
VARIABLE TempHex : STRING(1 to 16);
-- Reason blurb in case of failure
VARIABLE ReasonStr : STRING(1 to 2048);
-- Position index used in the append function
VARIABLE p : NATURAL := 1;
-- For Cin, Cout, Ovfl
VARIABLE TempBit : STD_LOGIC;
-- Used in between each readline from the test vector file due to the spacing requirement in the
test vector specifications
VARIABLE TempChar : CHARACTER;
-- To keep track of whether the DUT passed the current Test Vector
VARIABLE TVPassed : BOOLEAN;
BEGIN
-- Loop until the end of the test vector file
WHILE NOT ENDFILE(InputFile) LOOP
ReasonStr := (OTHERS => ' ');
p := 1;
TVPassed := TRUE;
```

```
READLINE(InputFile, CurrentLine);
READ(CurrentLine, TempHex);
TV.inX := hex_to_slv(TempHex, 64);
READ(CurrentLine, TempChar);
READ(CurrentLine, TempHex);
TV.inY := hex_to_slv(TempHex, 64);
READ(CurrentLine, TempChar);
READ(CurrentLine, TempBit);
TV.inC := TempBit;
READ(CurrentLine, TempChar);
READ(CurrentLine, TempHex);
TV.outS := hex_to_slv(TempHex, 64);
READ(CurrentLine, TempChar);
READ(CurrentLine, TempBit);
TV.outC := TempBit;
READ(CurrentLine, TempChar);
READ(CurrentLine, TempBit);
TV.outOvfl := TempBit;
-- Now, to actually care about the DUT.
-- Apply 'X' to all input bits, hold for PreStimTime
TBX <= (OTHERS => 'X');
TBY <= (OTHERS => 'X');
TBC <= 'X';
WAIT FOR PreStimTime;
-- Apply stimuli until outputs are stable
TBX <= TV.inX;
TBY <= TV.inY;
TBC <= TV.inC;
WAIT FOR PostStimTime;
-- Verify correct results, append message to reason blurb on failure
IF DUT_S /= TV.outS THEN
TVPassed := FALSE;
work.Utils.append(ReasonStr, p, "-|Sum Mismatch|-");
END IF;
IF DUT_Cout /= TV.outC THEN
TVPassed := FALSE;
work.Utils.append(ReasonStr, p, "-|Cout Mismatch|-");
END IF;
IF DUT_Ovfl /= TV.outOvfl THEN
TVPassed := FALSE;
work.Utils.append(ReasonStr, p, "-|Ovfl Mismatch|-");
END IF;
-- One-liner describing the test vector result
IF TVPassed = FALSE THEN
REPORT  "Measurement #" & INTEGER'IMAGE(MeasurementIndex) & " Failed." &
        " Reason:" & ReasonStr &
        " Stimulus:" &
```

```
     " [A: " & slv_to_hex(TV.inX) &
     " B: " & slv_to_hex(TV.inY) &
     " Cin: " & INTEGER'IMAGE(conv_integer(TV.inC)) & "]" &
     " Expected Outputs:" &
     " [S: " & slv_to_hex(TV.outS) &
     " Cout: " & INTEGER'IMAGE(conv_integer(TV.outC)) &
     " Ovfl: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) & "]" &
     " Actual Outputs:" &
     " [S: " & slv_to_hex(DUT_S) &
     " Cout: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) &
     " Ovfl: " & INTEGER'IMAGE(conv_integer(DUT_Ovfl)) & "]";
ELSE
REPORT  "Measurement #" & INTEGER'IMAGE(MeasurementIndex) & " Passed." &
     " Stimulus:" &
     " [A: " & slv_to_hex(TV.inX) &
     " B: " & slv_to_hex(TV.inY) &
     " Cin: " & INTEGER'IMAGE(conv_integer(TV.inC)) & "]" &
     " Expected Outputs:" &
     " [S: " & slv_to_hex(TV.outS) &
     " Cout: " & INTEGER'IMAGE(conv_integer(TV.outC)) &
     " Ovfl: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) & "]" &
     " Actual Outputs:" &
     " [S: " & slv_to_hex(DUT_S) &
     " Cout: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) &
     " Ovfl: " & INTEGER'IMAGE(conv_integer(DUT_Ovfl)) & "]";
END IF;
-- Increment measurement index for next measurement
MeasurementIndex := MeasurementIndex + 1;
END LOOP;
-- Finishing wait due to this process not containing an initializer list. Without this we would loop
the process infinitely
WAIT;
END PROCESS main;
END TestCSAN;
----------
----------End of TBCSAN.vhd
----------


----------
----------Start of TBRCAN.vhd.
----------
-- Library required for the "append" utility function
use work.Utils.all;
-- TB ENTITY declaration
ENTITY TBRCAN IS
GENERIC (N:NATURAL := 64);
END ENTITY TBRCAN;
```

```vhdl
-- TB ARCHITECTURE Declaration
ARCHITECTURE TestRCAN OF TBRCAN IS
-- Constants
-- Test Vector File
CONSTANT TestVectorFile : STRING := "Adder00.tvs";
-- Pre-Stimulation Delay
CONSTANT PreStimTime: TIME := 1ns;
-- Time to wait after stimulation for outputs to settle. This should be arbitrary: it's a topology so delays are nonexistent
CONSTANT PostStimTime: TIME := 10ns;
-- Newline character
CONSTANT NL : string(1 to 1) := (1 => CHARACTER'VAL(10));
-- Test Record Declaration
TYPE TestVectorOp IS RECORD
-- Inputs
inX: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
inY: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
inC: STD_LOGIC;
-- Outputs
outS: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
outC: STD_LOGIC;
outOvfl: STD_LOGIC;
END RECORD TestVectorOp;
-- Files
FILE InputFile: TEXT OPEN READ_MODE IS "./TestVectors/" & TestVectorFile;
-- Signals
-- Testbench Signals
SIGNAL TBX: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
SIGNAL TBY: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
SIGNAL TBC: STD_LOGIC;
-- DUT Signals
SIGNAL DUT_S: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
SIGNAL DUT_Cout: STD_LOGIC;
SIGNAL DUT_Ovfl: STD_LOGIC;
BEGIN
-- Actual DUT Instance. This would be RCAN in the TBRCAN.vhd file
DUT: ENTITY WORK.RCAN(Baseline) -- DUT: ENTITY WORK.RCAN(Baseline) in TBRCAN.vhd
GENERIC MAP (N => N)
PORT MAP (
X => TBX,
Y => TBY,
Cin => TBC,
S => DUT_S,
Cout => DUT_Cout,
Ovfl => DUT_Ovfl
);
main:
```

```
PROCESS
-- Line we are currently processing
VARIABLE CurrentLine: LINE;
VARIABLE TranscriptLine: LINE;
-- The TestVector
VARIABLE TV: TestVectorOp;
-- Index
VARIABLE MeasurementIndex : INTEGER := 1;
-- For holding each segment of the line to set the TV
VARIABLE TempHex : STRING(1 to 16);
-- Reason blurb in case of failure
VARIABLE ReasonStr : STRING(1 to 2048);
-- Position index used in the append function
VARIABLE p : NATURAL := 1;
-- For Cin, Cout, Ovfl
VARIABLE TempBit : STD_LOGIC;
-- Used in between each readline from the test vector file due to the spacing requirement in the test vector specifications
VARIABLE TempChar : CHARACTER;
-- To keep track of whether the DUT passed the current Test Vector
VARIABLE TVPassed : BOOLEAN;
BEGIN
-- Loop until the end of the test vector file
WHILE NOT ENDFILE(InputFile) LOOP
ReasonStr := (OTHERS => ' ');
p := 1;
TVPassed := TRUE;
READLINE(InputFile, CurrentLine);
READ(CurrentLine, TempHex);
TV.inX := hex_to_slv(TempHex, 64);
READ(CurrentLine, TempChar);
READ(CurrentLine, TempHex);
TV.inY := hex_to_slv(TempHex, 64);
READ(CurrentLine, TempChar);
READ(CurrentLine, TempBit);
TV.inC := TempBit;
READ(CurrentLine, TempChar);
READ(CurrentLine, TempHex);
TV.outS := hex_to_slv(TempHex, 64);
READ(CurrentLine, TempChar);
READ(CurrentLine, TempBit);
TV.outC := TempBit;
READ(CurrentLine, TempChar);
READ(CurrentLine, TempBit);
TV.outOvfl := TempBit;
-- Now, to actually care about the DUT.
-- Apply 'X' to all input bits, hold for PreStimTime
```

```
TBX <= (OTHERS => 'X');
TBY <= (OTHERS => 'X');
TBC <= 'X';
WAIT FOR PreStimTime;
-- Apply stimuli until outputs are stable
TBX <= TV.inX;
TBY <= TV.inY;
TBC <= TV.inC;
WAIT FOR PostStimTime;
-- Verify correct results, append message to reason blurb on failure
IF DUT_S /= TV.outS THEN
TVPassed := FALSE;
work.Utils.append(ReasonStr, p, "-|Sum Mismatch|-");
END IF;
IF DUT_Cout /= TV.outC THEN
TVPassed := FALSE;
work.Utils.append(ReasonStr, p, "-|Cout Mismatch|-");
END IF;
IF DUT_Ovfl /= TV.outOvfl THEN
TVPassed := FALSE;
work.Utils.append(ReasonStr, p, "-|Ovfl Mismatch|-");
END IF;
-- One-liner describing the test vector result
IF TVPassed = FALSE THEN
REPORT  "Measurement #" & INTEGER'IMAGE(MeasurementIndex) & " Failed." &
      " Reason:" & ReasonStr &
      " Stimulus:" &
      " [A: " & slv_to_hex(TV.inX) &
      " B: " & slv_to_hex(TV.inY) &
      " Cin: " & INTEGER'IMAGE(conv_integer(TV.inC)) & "]" &
      " Expected Outputs:" &
      " [S: " & slv_to_hex(TV.outS) &
      " Cout: " & INTEGER'IMAGE(conv_integer(TV.outC)) &
      " Ovfl: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) & "]" &
      " Actual Outputs:" &
      " [S: " & slv_to_hex(DUT_S) &
      " Cout: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) &
      " Ovfl: " & INTEGER'IMAGE(conv_integer(DUT_Ovfl)) & "]";
ELSE
REPORT  "Measurement #" & INTEGER'IMAGE(MeasurementIndex) & " Passed." &
      " Stimulus:" &
      " [A: " & slv_to_hex(TV.inX) &
      " B: " & slv_to_hex(TV.inY) &
      " Cin: " & INTEGER'IMAGE(conv_integer(TV.inC)) & "]" &
      " Expected Outputs:" &
      " [S: " & slv_to_hex(TV.outS) &
      " Cout: " & INTEGER'IMAGE(conv_integer(TV.outC)) &
      " Ovfl: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) & "]" &
      " Actual Outputs:" &
      " [S: " & slv_to_hex(DUT_S) &
      " Cout: " & INTEGER'IMAGE(conv_integer(TV.outOvfl)) &
      " Ovfl: " & INTEGER'IMAGE(conv_integer(DUT_Ovfl)) & "]";
END IF;
-- Increment measurement index for next measurement
MeasurementIndex := MeasurementIndex + 1;
END LOOP;
-- Finishing wait due to this process not containing an initializer list. Without this we would loop the process infinitely
WAIT;
END PROCESS main;
END TestRCAN;
----------
----------End of TBRCAN.vhd
----------
```

```
----------
----------Start of CSAN.vhd
----------
-- CSAN entity:
-- N: Number of bits of operands (and sum)
-- C: Recursion base case adder. For instance, if N=64, and C=4, recursive principle of CSA will be applied until N=4.
-- X: Operand 1 (A)
-- Y: Operand 2 (B)
-- Cin: Carry-in
-- S: Sum
-- Cout: Carry-out
-- Ovfl: True if overflow occurred in the addition of signed operands
ENTITY CSAN IS
GENERIC (N: NATURAL := 64;
         C: NATURAL := 1);
PORT (
      X: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      Y: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      S: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      Cin: IN STD_LOGIC;
      Cout: OUT STD_LOGIC;
      Ovfl: OUT STD_LOGIC);
END CSAN;
ARCHITECTURE LogicFuncCSAN OF CSAN IS
-- signal decs
-- Carries of the upper and lower adders on left half
SIGNAL CwC0, CwC1: STD_LOGIC;
```

```
-- Sums of the upper and lower adders on left half, as well as sum of right half
SIGNAL SwC0, SwC1, SR: STD_LOGIC_VECTOR((N-1)/2 DOWNTO 0);
-- Overflow of upper and lower adders on left half (Not currently in use)
SIGNAL Ovfl0, Ovfl1: STD_LOGIC;
-- Intermediate signals, feeding into the N/2 + 1 bit MUX. Concatenation of carryout of left half adders and their sums
SIGNAL Int0, Int1: STD_LOGIC_VECTOR((N+1)/2 DOWNTO 0);
-- Result of N/2 + 1 bit MUX. Select bit is carry from right half, inputs are the intermediate signals
SIGNAL MuxResult: STD_LOGIC_VECTOR((N+1)/2 DOWNTO 0);
-- Temporary Carryout, Overflow signals. (tempO not currently in use)
SIGNAL tempC, tempO: STD_LOGIC;
BEGIN
-- C equals 1 for usage of standard full adders in base case. For non-leaf nodes, divide into left and right halves
gen: IF N > C GENERATE
-- Left half adder with Cin = 0
  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 adder with Cin = 1
  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);
-- Intermediate (Concatenated Cout, Sums) signals of leftmost adders
  Int0 <= CWC0 & SwC0;
  Int1 <= CWC1 & SwC1;
-- N/2 + 1 bit, 2 channel MUX to select which carry, sum will be the output for the left half.
-- If we think of this as a divide and conquer algorithm, this is exactly the merging step.
  mux:
     ENTITY work.Mux2cNb
     GENERIC MAP(N => N/2 + 1, C => 1)
     PORT MAP(x1 => Int0, x2 => Int1, s => tempC, y => MuxResult);
     -- Ultimate Cout
     Cout <= MuxResult((N+1)/2);
     -- Upper half of sum bits
     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
  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
        );
        -- Lower half of sum bits
        S((N-1)/2 DOWNTO 0) <= SR;
-- 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).
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;

-- Recursion base case: just use a full adder
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;
END LogicFuncCSAN;
----------
----------End of CSAN.vhd
----------

----------
----------Start of RCAN.vhd
----------Note: FullAddr entities are logically Identical to FA entities.
----------
-- Take the FullAddr entity to make 64 full adders into a ripple adder
-- RCAN entity:
-- N: Number of bits of operands (and sum)
-- C: Recursion base case adder. For instance, if N=64, and C=4, recursive principle of CSA will be applied until N=4.
-- X: Operand 1 (A)
-- Y: Operand 2 (B)
-- Cin: Carry-in
-- S: Sum
-- Cout: Carry-out
-- Ovfl: True if overflow occurred in the addition of signed operands
Entity RCAN is
```

```vhdl
generic(
  N : natural := 64
);
port (
      X, Y  : in std_logic_vector(N-1 downto 0);
      Cin   : in std_logic;
      S     : out std_logic_vector(N-1 downto 0);
      Cout    : out std_logic;
      Ovfl  : out std_logic
);
end RCAN;
-- Baseline Adder
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;
-- Quartus FastRipple:
architecture FastRipple of RCAN is
begin
PROCESS(X, Y)
-- Temporary holding variable for the sum
VARIABLE tempS : STD_LOGIC_VECTOR(N DOWNTO 0);
BEGIN
tempS := std_logic_vector((resize(unsigned(X), N+1) + resize(unsigned(Y),N+1) + resize( ( unsigned( std_logic_vector'('0' & Cin)) ),N+1 )   ) );
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))));
S <= tempS(N-1 DOWNTO 0);
Cout <= tempS(N);
END PROCESS;
end architecture;
----------
----------End of RCAN.vhd
```

```
----------

----------
----------Start of FA.vhd
----------Note: Logically identical to FullAddr.vhd
----------
-- FA entity:
-- C: For a full adder, this is 1. If an N-bit ripple carry adder is desired, set C to N.
-- X: Operand 1 (A)
-- Y: Operand 2 (B)
-- Cin: Carry-in
-- S: Sum
-- Cout: Carry-out
-- Ovfl: True if overflow occurred in the addition of signed operands
GENERIC (
C: NATURAL := 1
);
PORT(
        X: IN STD_LOGIC_VECTOR(C-1 DOWNTO 0);
        Y: IN STD_LOGIC_VECTOR(C-1 DOWNTO 0);
        S: OUT STD_LOGIC_VECTOR(C-1 DOWNTO 0);
        Cin: IN STD_LOGIC;
        Cout: OUT STD_LOGIC;
        Ovfl: OUT STD_LOGIC);
END FA;
ARCHITECTURE LogicFuncFA OF FA IS
-- Basic ripple adder implementation
SIGNAL Couts: STD_LOGIC_VECTOR(C DOWNTO 0);
BEGIN
Couts(0) <=  Cin;
gen1: FOR i IN 0 TO C-1 GENERATE
   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
                   '0';
   S(i) <= Couts(i) XOR X(i) XOR Y(i);
END GENERATE gen1;
Cout <= Couts(C);
-- Ovfl logic here is Cn xor Cn-1 because we have access to all carries. Requires less gates.
Ovfl <= Couts(C) XOR Couts(C-1);
END LogicFuncFA;
----------
----------End of FA.vhd
----------

----------
----------Start of FullAddr.vhd
----------
```

```
-- Entity for a 1-bit adder so that the RTL viewer makes it look nice (and not all gates)
-- FullAddr entity:
-- X: Operand 1 (A)
-- Y: Operand 2 (B)
-- Cin: Carry-in
-- S: Sum
-- Cout: Carry-out
-- Ovfl: In a 1-bit full adder, this is not very important, so it is omitted.
Entity FullAddr is
port (
      X  : in std_logic;
      Y  : in std_logic;
      Cin   : in std_logic;
      S      : out std_logic;
      Cout     : out std_logic;
);
end FullAddr;
architecture behavioural of FullAddr is
begin
  S <= (X xor Y)    xor Cin;
  Cout <= (X and Y) or ((X or Y) and Cin);
end architecture;
----------
----------End of FullAddr.vhd
----------


----------
----------Start of Mux2cNb.vhd
----------
-- 2-channel, N-bit MUX
-- Mux2cNb entity:
-- N: Number of bits in inputs and output
-- C: Base case mux (should be 1, for 2-channel 1 bit MUX)
-- x1: operand 1 (N bits, selected when s = 0)
-- x2: operand 2 (N bits, selected when s = 0)
-- s: select bit
-- y: (N bits) output
ENTITY Mux2cNb IS
GENERIC (N: NATURAL := 2;
                C: NATURAL := 1);
PORT (x1, x2: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
      s: IN STD_LOGIC;
      y: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0)
      );
END Mux2cNb;

ARCHITECTURE LogicFuncMux2cNb OF Mux2cNb IS
```

```
BEGIN
-- Will generate N side by side 2-channel 1 bit muxes to create the 2-channel n bit mux
gen: FOR i IN 0 TO N-1 GENERATE
WITH s SELECT
y(i) <= x1(i) WHEN '0',
        x2(i) WHEN others;

END GENERATE gen;
END LogicFuncMux2cNb;
----------
----------End of Mux2cNb.vhd
----------

----------
----------Start of Mux2c1b.vhd
----------
-- 2-channel, 1-bit MUX
-- Standard 2-channel 1 bit mux. equation: y = s'x1 + sx2
-- Mux2c1b entity:
-- x1: operand 1 (1 bit, selected when s = 0)
-- x2: operand 2 (1 bit, selected when s = 0)
-- s: select bit
-- y: output (1 bit)
ENTITY Mux2c1b IS
PORT (
x1, x2 : IN STD_LOGIC;
s: IN STD_LOGIC;
y: OUT STD_LOGIC
);
END Mux2c1b;
ARCHITECTURE LogicFunc OF Mux2c1b IS
BEGIN
WITH s SELECT
  y <=  x1 WHEN '0',
        x2 WHEN others;

END LogicFunc;
----------
----------End of Mux2c1b.vhd
----------

----------
----------Start of Utils.vhd
----------Note: This provides utility functions across the workspace
----------
PACKAGE Utils IS
FUNCTION hex_to_slv(Hex : IN STRING; WIDTH : IN NATURAL) RETURN STD_LOGIC_VECTOR;
```

```vhdl
FUNCTION slv_to_hex(X: IN STD_LOGIC_VECTOR) RETURN STRING;
PROCEDURE append(Buf: INOUT STRING;p: INOUT NATURAL;S: IN STRING);
END PACKAGE;
PACKAGE BODY Utils IS
-- Helper function to convert a hex string to a std logic vector
FUNCTION hex_to_slv(Hex : IN STRING; WIDTH: IN NATURAL) RETURN STD_LOGIC_VECTOR IS
  VARIABLE Result    : STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0) := (OTHERS => '0');
  VARIABLE V       : UNSIGNED(WIDTH-1 DOWNTO 0) := (OTHERS => '0');
BEGIN
-- Iterate through all the elements in the hexadecimal string
  FOR i IN 1 TO Hex'LENGTH LOOP
      V := V SLL 4;
       -- Put the chunks into the variable V 4 bits by 4 bits
      CASE Hex(i) IS
          WHEN '0' => V := V OR to_unsigned(0, WIDTH);
          WHEN '1' => V := V OR to_unsigned(1, WIDTH);
          WHEN '2' => V := V OR to_unsigned(2, WIDTH);
          WHEN '3' => V := V OR to_unsigned(3, WIDTH);
          WHEN '4' => V := V OR to_unsigned(4, WIDTH);
          WHEN '5' => V := V OR to_unsigned(5, WIDTH);
          WHEN '6' => V := V OR to_unsigned(6, WIDTH);
          WHEN '7' => V := V OR to_unsigned(7, WIDTH);
          WHEN '8' => V := V OR to_unsigned(8, WIDTH);
          WHEN '9' => V := V OR to_unsigned(9, WIDTH);
          WHEN 'A' | 'a' => V := V OR to_unsigned(10, WIDTH);
          WHEN 'B' | 'b' => V := V OR to_unsigned(11, WIDTH);
          WHEN 'C' | 'c' => V := V OR to_unsigned(12, WIDTH);
          WHEN 'D' | 'd' => V := V or to_unsigned(13, WIDTH);
          WHEN 'E' | 'e' => V := V OR to_unsigned(14, WIDTH);
          WHEN 'F' | 'f' => V := V OR to_unsigned(15, WIDTH);
          WHEN OTHERS => NULL;
      END CASE;
  END LOOP;
  -- Cast V to an slv and return it
  Result := STD_LOGIC_VECTOR(V);
  RETURN Result;
END FUNCTION;
-- Helper function to convert a std logic vector to a hexadecimal string
FUNCTION slv_to_hex(X: IN std_logic_vector) RETURN STRING IS
  CONSTANT HexChars : string := "0123456789ABCDEF";
  VARIABLE Hstr : string(1 TO X'LENGTH / 4);
  VARIABLE Chunk  : unsigned(3 DOWNTO 0);
BEGIN
   FOR i IN 0 TO X'LENGTH/4 - 1 LOOP
          -- Get 4 bits at a time. 'HIGH is the high end of the slv
          Chunk := unsigned(X(X'HIGH-i*4 DOWNTO X'HIGH-i*4-3));
          -- +1 since VHDL string index starts at 1
          Hstr(i+1) := HexChars(to_integer(Chunk)+1);
    END LOOP;
RETURN Hstr;
END FUNCTION;
-- Helper procedure to append a string to a buffer
-- Needs to be a procedure, because it needs to modify the Buf in place
PROCEDURE append(Buf: INOUT STRING; p: INOUT NATURAL; S: IN STRING) IS
  VARIABLE lenS: NATURAL := S'LENGTH;
  CONSTANT BUF_LOW: NATURAL := 1;
  -- Can be changed around but hopefully this is enough space
  CONSTANT BUF_HIGH: NATURAL := 512;
BEGIN
  -- If the string fits within the buffer
  IF p+lenS-1 <= BUF_HIGH THEN
      Buf(p TO p+lenS-1) := S;
      p := p + lenS;
  ELSE
  -- If only part of the string fits
    Buf(p TO BUF_HIGH) := S(1 TO BUF_HIGH-p+1);
    p := BUF_HIGH+1;
  END IF;
END PROCEDURE;
END PACKAGE BODY;
----------
----------End of Utils.vhd
----------

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