```vhdl
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.ALL;
use std.textio.all;

-- naming convention for architectures: an acronym created based on the name of each topology (i.e. ripple-carry adder = RCA, etc.)

entity EN_Adder is
    generic (N: natural := 64);
    port (
        A, B  : in std_logic_vector (N-1 downto 0);
        S : out std_logic_vector (N-1 downto 0);
        Cin : in std_logic;
        Cout, Ovfl : out std_logic
    );
end EN_Adder;
/*
-- Quartus Optimized Ripple Adder
architecture FastRipple of EN_Adder is

signal temp : unsigned(N downto 0);
    begin
    -- Extend A and B to N+1 bits with a leading '0'
    -- Extend Cin to N+1 bits with N zeros in the upper bits

        temp <= unsigned('0' & A) + unsigned('0' & B) + unsigned(to_unsigned(0,N) & Cin);

        S    <= std_logic_vector(temp(N-1 downto 0));
    Cout <= temp(N);
     Ovfl <= (temp(N) xor temp(N-1));
end FastRipple;*/

-- RCA : Ripple Carry Adder
architecture RCA of EN_Adder is
    signal C : std_logic_vector (N downto 0);
    signal P,G : std_logic_vector (N-1 downto 0);

    begin
    -- replace c(0) so c can be used all throughout
    c(0) <= Cin;

    -- generate and propagates block (for every bit)
    g <= a and b;
    p <= a xor b;

    -- carry network
    c(N downto 1) <= g or (p and c(N-1 downto 0));

    -- generate sum bits
    S <= p xor c(n-1 downto 0);

    -- label the final carry as cout
    Cout <= C(N);

    -- ovfl logic
    Ovfl <= (not (A(N-1) xor B(N-1))) and (A(N-1) xor S(N-1));


end RCA;

-- CSA : Conditional Sum Adder
architecture CSA of EN_Adder is
    -- define size of each recursion
    constant N_half : integer := N / 2;
    -- define intermediate signals
    signal Cout0, Cout1, Chalf : std_logic := '0';
    signal sum0, sum1 : std_logic_vector ((N_half-1) downto 0) := (others => '0');


    begin
        -- generate recursion until we reach base case of N = 2
```

```
72          recur : if N > 2 generate
73              begin
74              -- Lower half of calculations
75              CSAlow : entity work.EN_Adder(CSA)
76                  generic map (N => N_half)
77                  port map (
78                      A => A(N_half-1 downto 0),
79                      B => B(N_half-1 downto 0),
80                      S => S(N_half-1 downto 0),
81                      Cin => Cin,
82                      Cout => Chalf,
83                      Ovfl => open
84                  );
85              -- upper half caclulations assuming cin = 0
86              CSAzero : entity work.EN_Adder(CSA)
87                  generic map (N => N_half)
88                  port map (
89                      A    => A(N-1 downto N_half),
90                      B    => B(N-1 downto N_half),
91                      S    => sum0,
92                      Cin  => '0',
93                      Cout => Cout0,
94                      Ovfl => open
95                  );
96              -- -- upper half caclulations assuming cin = 1
97              CSAone : entity work.EN_Adder(CSA)
98                  generic map (N => N_half)
99                  port map (
100                     A    => A(N-1 downto N_half),
101                     B    => B(N-1 downto N_half),
102                     S    => sum1,
103                     Cin  => '1',
104                     Cout => Cout1,
105                     Ovfl => open
106                 );
107         end generate recur;
108         -- leaf case 2 bit carry-select adder
109         leaf: if N = 2 generate
110             -- define intermediate signals
111             signal g, p : std_logic;
112             begin
113                 -- generate and propagate for lower bit
114                 g <= A(0) and B(0);
115                 p <= A(0) xor B(0);
116                 S(0) <= p xor Cin;
117
118                 -- Mux input (i.e intermediate carry/ true Cin)
119                 Chalf <= g or (Cin and p);
120
121                 -- half adder with cin = 0
122                 sum0(0) <= A(1) xor B(1);
123
124                 -- half adder with cin = 1
125                 sum1(0) <= not (A(1) xor B(1));
126                 -- Cout nux inputs
127                 Cout0 <= A(1) and B(1);
128                 Cout1 <= A(1) or B(1);
129         end generate leaf;
130
131         -- sum mux
132         S((N-1)downto N_half) <= sum1 when Chalf = '1' else sum0;
133         -- cout mux
134         Cout <= Cout1 when Chalf = '1' else Cout0;
135         -- ovfl logic
136         Ovfl <= (not (A(N-1) xor B(N-1))) and (A(N-1) xor S(N-1));
137
138 end CSA;
139
140 -- LACTA : Look Ahead Carry Tree Adder
141 architecture LACTA of EN_Adder is
142     -- define intermediate signals
143     signal P, G : std_logic_vector(N-1 downto 0);
144
```

```
145     begin
146         -- propogate and generate block (for each bit)
147         P <= A xor B;
148         G <= A and B;
149 
150         -- Base case N = 4
151         leaf: if N = 4 generate
152             -- define intermediate signals
153             signal Cbits : std_logic_vector(4 downto 0); -- Carry out within leaf
154             signal Pg4, Gg4 : std_logic;
155 
156             begin
157                 -- replace Cbits(0) so c can be used all throughout
158                 Cbits(0) <= Cin;
159 
160                 -- use 4 bit look ahead carry network componenet (EN_LACGN4) to get
                    the carries
161                 U_LEAF: entity work.EN_LACG4(LACN4)
162                 generic map (N => 4)
163                 port map (
164                     Gin  => G,
165                     Pin  => P,
166                     Gout => Gg4,
167                     Pout => Pg4,
168                     Cin  => Cbits(0),
169                     C    => Cbits(3 downto 1)
170                 );
171 
172                 -- generate the sum from EN_LACGN4 outputs
173                 S    <= P xor Cbits(3 downto 0);
174 
175                 -- intermediate Cout
176                 Cout <= (Pg4 and Cin) or Gg4;
177 
178                 -- intermediate ovfl ???
179                 Ovfl <= (not (A(3) xor B(3))) and (A(3) xor S(3));
180         end generate;
181 
182         -- Generate Recursive case N > 4
183 
184         recur: if N > 4 generate
185             -- split N into quarters
186             constant Q : integer := N/4;
187 
188             -- intermediate signals
189             signal A0, A1, A2, A3 : std_logic_vector(Q-1 downto 0);
190             signal B0, B1, B2, B3 : std_logic_vector(Q-1 downto 0);
191             signal P0, P1, P2, P3 : std_logic_vector(Q-1 downto 0);
192             signal G0, G1, G2, G3 : std_logic_vector(Q-1 downto 0);
193 
194 
195 
196             -- Quarter group P/G (one bit per quarter)
197             signal PgQ : std_logic_vector(3 downto 0);
198             signal GgQ : std_logic_vector(3 downto 0);
199 
200             -- Carry fanout to quarters (Cq(0)=Cin, Cq(1..3) drive Q1..Q3)
201             signal Cq  : std_logic_vector(3 downto 0);
202 
203             -- Quarter sums
204             signal S0, S1, S2, S3 : std_logic_vector(Q-1 downto 0);
205 
206             -- Top group P/G (for overall Cout)
207             signal PgTop, GTop : std_logic;
208 
209             -- First reduction stage: 4-bit blocks inside each quarter
210             constant BLKS_PER_Q : integer := Q/4;
211             signal Pg0_blk, Gg0_blk : std_logic_vector(BLKS_PER_Q-1 downto 0);
212             signal Pg1_blk, Gg1_blk : std_logic_vector(BLKS_PER_Q-1 downto 0);
213             signal Pg2_blk, Gg2_blk : std_logic_vector(BLKS_PER_Q-1 downto 0);
214             signal Pg3_blk, Gg3_blk : std_logic_vector(BLKS_PER_Q-1 downto 0);
215         begin
216             -- Quarter slices
```

```
217             A0 <= A(Q-1 downto 0);
218             A1 <= A(2*Q-1 downto Q);
219             A2 <= A(3*Q-1 downto 2*Q);
220             A3 <= A(4*Q-1 downto 3*Q);
221
222             B0 <= B(Q-1 downto 0);
223             B1 <= B(2*Q-1 downto Q);
224             B2 <= B(3*Q-1 downto 2*Q);
225             B3 <= B(4*Q-1 downto 3*Q);
226
227             P0 <= P(Q-1 downto 0);
228             P1 <= P(2*Q-1 downto Q);
229             P2 <= P(3*Q-1 downto 2*Q);
230             P3 <= P(4*Q-1 downto 3*Q);
231
232             G0 <= G(Q-1 downto 0);
233             G1 <= G(2*Q-1 downto Q);
234             G2 <= G(3*Q-1 downto 2*Q);
235             G3 <= G(4*Q-1 downto 3*Q);
236
237             -- Build 4-bit block groups inside each quarter
238             blk_q0: for i in 0 to BLKS_PER_Q-1 generate
239               constant L : integer := 4*i;
240             begin
241                 Q0_BLK: entity work.EN_LACG4(LACN4)
242                     generic map (N => 4)
243                     port map ( Gin=>G0(L+3 downto L), Pin=>P0(L+3 downto L),
244                                Gout=>Gg0_blk(i), Pout=>Pg0_blk(i), Cin=>'0', C=>open
                               );
245             end generate;
246
247             blk_q1: for i in 0 to BLKS_PER_Q-1 generate
248                 constant L : integer := 4*i;
249             begin
250                 Q1_BLK: entity work.EN_LACG4(LACN4)
251                     generic map (N => 4)
252                     port map ( Gin=>G1(L+3 downto L), Pin=>P1(L+3 downto L),
253                                Gout=>Gg1_blk(i), Pout=>Pg1_blk(i), Cin=>'0', C=>open
                               );
254             end generate;
255
256             blk_q2: for i in 0 to BLKS_PER_Q-1 generate
257                 constant L : integer := 4*i;
258             begin
259                 Q2_BLK: entity work.EN_LACG4(LACN4)
260                     generic map (N => 4)
261                     port map ( Gin=>G2(L+3 downto L), Pin=>P2(L+3 downto L),
262                                Gout=>Gg2_blk(i), Pout=>Pg2_blk(i), Cin=>'0', C=>open
                               );
263             end generate;
264
265             blk_q3: for i in 0 to BLKS_PER_Q-1 generate
266                 constant L : integer := 4*i;
267             begin
268                 Q3_BLK: entity work.EN_LACG4(LACN4)
269                     generic map (N => 4)
270                     port map ( Gin=>G3(L+3 downto L), Pin=>P3(L+3 downto L),
271                                Gout=>Gg3_blk(i), Pout=>Pg3_blk(i), Cin=>'0', C=>open
                               );
272             end generate;
273
274         -- Combine the 4 block groups of each quarter into one group bit (Q=16)
275         combine_q16: if Q = 16 generate
276             Q0_GRP: entity work.EN_LACG4(LACN4)
277                 generic map (N => 4)
278                 port map ( Gin=>Gg0_blk, Pin=>Pg0_blk, Gout=>GgQ(0), Pout=>PgQ(0), Cin
                    =>'0', C=>open );
279             Q1_GRP: entity work.EN_LACG4(LACN4)
280                 generic map (N => 4)
281                 port map ( Gin=>Gg1_blk, Pin=>Pg1_blk, Gout=>GgQ(1), Pout=>PgQ(1), Cin
                    =>'0', C=>open );
282             Q2_GRP: entity work.EN_LACG4(LACN4)
283                 generic map (N => 4)
```

```
                port map ( Gin=>Gg2_blk, Pin=>Pg2_blk, Gout=>GgQ(2), Pout=>PgQ(2), Cin
                =>'0', C=>open );
            Q3_GRP: entity work.EN_LACG4(LACN4)
                generic map (N => 4)
                port map ( Gin=>Gg3_blk, Pin=>Pg3_blk, Gout=>GgQ(3), Pout=>PgQ(3), Cin
                =>'0', C=>open );
        end generate;

        -- Pass-through for Q=4 (each quarter already one 4-bit block)
        combine_q4: if Q = 4 generate
            PgQ(0) <= Pg0_blk(0);  GgQ(0) <= Gg0_blk(0);
            PgQ(1) <= Pg1_blk(0);  GgQ(1) <= Gg1_blk(0);
            PgQ(2) <= Pg2_blk(0);  GgQ(2) <= Gg2_blk(0);
            PgQ(3) <= Pg3_blk(0);  GgQ(3) <= Gg3_blk(0);
        end generate;

        -- Top EN_LACG4 over the 4 quarters: generates carries into Q1..Q3
        Cq(0) <= Cin;
        TOP_FANOUT: entity work.EN_LACG4(LACN4)
            generic map (N => 4)
            port map (
                Gin  => GgQ,
                Pin  => PgQ,
                Gout => GTop,
                Pout => PgTop,
                Cin  => Cq(0),
                C    => Cq(3 downto 1)  -- three carries into Q1..Q3
          );

        -- Recurse into four child adders with their proper carry-ins
        UQ0: entity work.EN_Adder(LACTA)
            generic map (N => Q)
            port map ( A=>A0, B=>B0, S=>S0, Cin=>Cq(0), Cout=>open, Ovfl=>open );

        UQ1: entity work.EN_Adder(LACTA)
            generic map (N => Q)
            port map ( A=>A1, B=>B1, S=>S1, Cin=>Cq(1), Cout=>open, Ovfl=>open );

        UQ2: entity work.EN_Adder(LACTA)
            generic map (N => Q)
            port map ( A=>A2, B=>B2, S=>S2, Cin=>Cq(2), Cout=>open, Ovfl=>open );

        UQ3: entity work.EN_Adder(LACTA)
            generic map (N => Q)
            port map ( A=>A3, B=>B3, S=>S3, Cin=>Cq(3), Cout=>open, Ovfl=>open );

        -- Stitch sums (MSB..LSB), top-level Cout, signed overflow
        S    <= S3 & S2 & S1 & S0;
        Cout <= (PgTop and Cin) or GTop;
        Ovfl <= (not (A(N-1) xor B(N-1))) and (A(N-1) xor S(N-1));
      end generate;


end LACTA;

-- BKA : Brent-Kung Adder
architecture BKA of EN_Adder is

    -- Bitwise propagate / generate
    signal P0, G0 : std_logic_vector(N-1 downto 0);

    -- intermediate signals
    signal P1, G1 : std_logic_vector(N-1 downto 0);
    signal P2, G2 : std_logic_vector(N-1 downto 0);
    signal P3, G3 : std_logic_vector(N-1 downto 0);
    signal P4, G4 : std_logic_vector(N-1 downto 0);
    signal P5, G5 : std_logic_vector(N-1 downto 0);

    -- Final p and g last stage
    signal Pf, Gf : std_logic_vector(N-1 downto 0);

    -- Carries
    signal C : std_logic_vector(N downto 0);
```

```
355
356         begin
357         -- 1) Bitwise propagate / generate
358         gen_pg : for i in 0 to N-1 generate
359             P0(i) <= A(i) xor B(i);
360             G0(i) <= A(i) and B(i);
361         end generate;
362
363         -- 2) Prefix network (Brent-Kung via iterative doubling: 1,2,4,8,16,32)
364         --    (G,P) □ (g,p) = (G or (P and g), P and p)
365
366         -- Stage 0: distance = 1
367         G1(0) <= G0(0);
368         P1(0) <= P0(0);
369         gen_s0_rest_en : if N > 1 generate
370             gen_s0_rest : for i in 1 to N-1 generate
371                 G1(i) <= G0(i) or (P0(i) and G0(i-1));
372                 P1(i) <= P0(i) and P0(i-1);
373             end generate;
374         end generate;
375
376         -- Stage 1: distance = 2
377         gen_s1_pass_en : if N >= 2 generate
378             gen_s1_pass : for i in 0 to 1 generate
379                 G2(i) <= G1(i);
380                 P2(i) <= P1(i);
381             end generate;
382         end generate;
383         gen_s1_rest_en : if N > 2 generate
384             gen_s1_rest : for i in 2 to N-1 generate
385                 G2(i) <= G1(i) or (P1(i) and G1(i-2));
386                 P2(i) <= P1(i) and P1(i-2);
387             end generate;
388         end generate;
389
390         -- Stage 2: distance = 4
391         gen_s2_pass_en : if N >= 4 generate
392             gen_s2_pass : for i in 0 to 3 generate
393                 G3(i) <= G2(i);
394                 P3(i) <= P2(i);
395             end generate;
396         end generate;
397         gen_s2_rest_en : if N > 4 generate
398             gen_s2_rest : for i in 4 to N-1 generate
399                 G3(i) <= G2(i) or (P2(i) and G2(i-4));
400                 P3(i) <= P2(i) and P2(i-4);
401             end generate;
402         end generate;
403
404         -- Stage 3: distance = 8
405         gen_s3_pass_en : if N >= 8 generate
406             gen_s3_pass : for i in 0 to 7 generate
407                 G4(i) <= G3(i);
408                 P4(i) <= P3(i);
409             end generate;
410         end generate;
411         gen_s3_rest_en : if N > 8 generate
412             gen_s3_rest : for i in 8 to N-1 generate
413                 G4(i) <= G3(i) or (P3(i) and G3(i-8));
414                 P4(i) <= P3(i) and P3(i-8);
415             end generate;
416         end generate;
417
418         -- Stage 4: distance = 16
419         gen_s4_pass_en : if N >= 16 generate
420             gen_s4_pass : for i in 0 to 15 generate
421                 G5(i) <= G4(i);
422                 P5(i) <= P4(i);
423             end generate;
424         end generate;
425         gen_s4_rest_en : if N > 16 generate
426             gen_s4_rest : for i in 16 to N-1 generate
427                 G5(i) <= G4(i) or (P4(i) and G4(i-16));
```

```
428                 P5(i) <= P4(i) and P4(i-16);
429             end generate;
430         end generate;
431 
432         -- Stage 5: distance = 32
433         gen_s5_pass_en : if N >= 32 generate
434             gen_s5_pass : for i in 0 to 31 generate
435                 Gf(i) <= G5(i);
436                 Pf(i) <= P5(i);
437             end generate;
438         end generate;
439         gen_s5_rest_en : if N > 32 generate
440             gen_s5_rest : for i in 32 to N-1 generate
441                 Gf(i) <= G5(i) or (P5(i) and G5(i-32));
442                 Pf(i) <= P5(i) and P5(i-32);
443             end generate;
444         end generate;
445 
446         -- 3) Carries / Sum / Ovfl
447         -- set C(0) as Cin for ease of computations
448         C(0) <= Cin;
449 
450         -- generate carries
451         gen_c : for i in 0 to N-1 generate
452             C(i+1) <= Gf(i) or (Pf(i) and Cin);
453         end generate;
454 
455         -- generate sums
456         gen_s : for i in 0 to N-1 generate
457             S(i) <= P0(i) xor C(i);
458         end generate;
459 
460         -- set Cout and Ovfl
461         Cout <= C(N);
462         Ovfl <= C(N) xor C(N-1);
463     end architecture BKA;
464 
465     -- Carry-Bypass Adder (CBA)
466     architecture CBA of EN_Adder is
467         -- Component declaration for the 4-bit lookahead carry generator
468         component EN_LACG4 is
469             generic (N: natural := 4);
470             port (
471                 Gin, Pin  : in std_logic_vector (N-1 downto 0);
472                 Gout, Pout  : out std_logic;
473                 Cin : in std_logic;
474                 C : out std_logic_vector (N-1 downto 1)
475             );
476         end component;
477 
478         -- Constants and signals
479         constant BLOCK_SIZE : natural := 4;
480         constant NUM_BLOCKS : natural := N / BLOCK_SIZE;
481 
482         -- Block propagate and generate signals
483         signal P_block, G_block : std_logic_vector(NUM_BLOCKS-1 downto 0);
484 
485         -- Block carry signals
486         signal block_carry_in : std_logic_vector(NUM_BLOCKS downto 0);
487         signal block_carry_out : std_logic_vector(NUM_BLOCKS-1 downto 0);
488         signal block_carry_ripple : std_logic_vector(NUM_BLOCKS-1 downto 0);
489 
490         -- Individual bit propagate and generate signals
491         signal P, G : std_logic_vector(N-1 downto 0);
492 
493         -- Internal carry signals for each bit
494         type carry_array is array (0 to NUM_BLOCKS-1) of std_logic_vector(BLOCK_SIZE-1
        downto 1);
495         signal C_internal : carry_array;
496 
497         -- Individual sum signals
498         signal S_internal : std_logic_vector(N-1 downto 0);
499 
```

```
500   begin
501       -- Initialize the first block carry input
502       block_carry_in(0) <= Cin;
503   
504       -- Generate all the propagate and generate signals for individual bits
505       GEN_PG: for i in 0 to N-1 generate
506           P(i) <= A(i) xor B(i);
507           G(i) <= A(i) and B(i);
508       end generate GEN_PG;
509   
510       -- Generate the 4-bit lookahead blocks
511       GEN_BLOCKS: for i in 0 to NUM_BLOCKS-1 generate
512           -- Instantiate the 4-bit lookahead carry generator for each block
513           LACG_INST: EN_LACG4
514               generic map (N => BLOCK_SIZE)
515               port map (
516                   Gin => G((i+1)*BLOCK_SIZE-1 downto i*BLOCK_SIZE),
517                   Pin => P((i+1)*BLOCK_SIZE-1 downto i*BLOCK_SIZE),
518                   Gout => G_block(i),
519                   Pout => P_block(i),
520                   Cin => block_carry_in(i),
521                   C => C_internal(i)
522               );
523   
524           -- Calculate the ripple carry output for this block (normal lookahead output)
525           block_carry_ripple(i) <= G_block(i) or (P_block(i) and block_carry_in(i));
526   
527           -- Skip logic: multiplexer to choose between ripple path and skip path
528           block_carry_out(i) <= block_carry_in(i) when P_block(i) = '1' else
              block_carry_ripple(i);
529   
530           -- Connect carry output to next block's carry input
531           block_carry_in(i+1) <= block_carry_out(i);
532   
533           -- Generate sum bits for this block
534           GEN_SUM_BITS: for j in 0 to BLOCK_SIZE-1 generate
535               -- For bit 0 in each block, use the block carry input
536               BIT0: if j = 0 generate
537                   S_internal(i*BLOCK_SIZE + j) <= P(i*BLOCK_SIZE + j) xor block_carry_in
                      (i);
538               end generate BIT0;
539   
540               -- For bits 1-3 in each block, use the internal carry from lookahead
541               OTHER_BITS: if j > 0 generate
542                   S_internal(i*BLOCK_SIZE + j) <= P(i*BLOCK_SIZE + j) xor C_internal(i)(
                      j);
543               end generate OTHER_BITS;
544           end generate GEN_SUM_BITS;
545       end generate GEN_BLOCKS;
546   
547       -- Connect outputs
548       S <= S_internal;
549       Cout <= block_carry_in(NUM_BLOCKS);
550   
551       -- Overflow detection for signed arithmetic
552       Ovfl <= (not (A(N-1) xor B(N-1))) and (A(N-1) xor S(N-1));
553   
554   end architecture CBA;
```