A

Examples

This appendix presents five examples that demonstrate basic concepts of Synopsys HDL Compiler:

Count Zeros—Combinational Version

Using this circuit is one possible solution to a design problem. Given an 8-bit value, the circuit must determine two things:

- The presence of a value containing exactly one sequence of zeros
- The number of zeros in the sequence (if any)

The circuit must complete this computation in a single clock cycle. The input to the circuit is an 8-bit value, and the two outputs the circuit produces are the number of zeros found and an error indication.

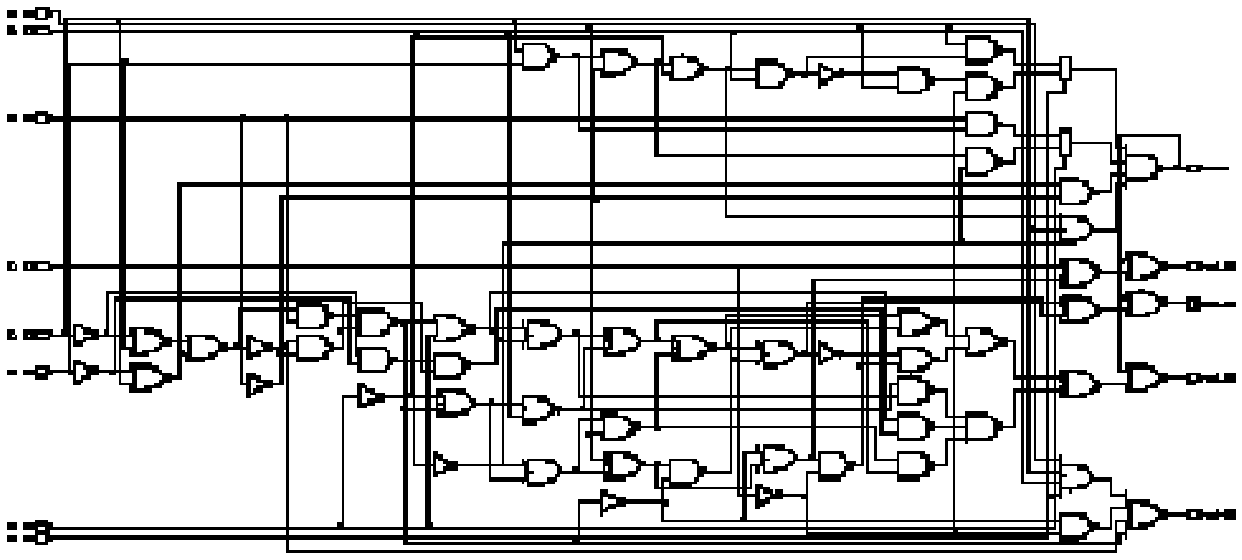
A valid value contains only one series of zeros. If more than one series of zeros appears, the value is invalid. A value consisting of all ones is a valid value. If a value is invalid, the count of zeros is set to zero. For example,

- The value 00000000 is valid, and the count is eight zeros.
- The value 11000111 is valid, and the count is three zeros.
- The value 00111110 is invalid.

A Verilog description and a schematic of the circuit are shown in Example A-1.

Example A-1 Count Zeros—Combinational

```
module count_zeros(in, out, error);
   input  [7:0] in;
   output [3:0] out;
   output error;
   function legal;
   input [7:0] x;
   reg seenZero, seenTrailing;
   integer i;
   begin : _legal_block
      legal = 1; seenZero = 0; seenTrailing = 0;
      for ( i=0; i <= 7; i=i+1 )
         if ( seenTrailing && (x[i] == 1’b0) ) begin
            legal = 0;
            disable _legal_block;
            end
         else if ( seenZero && (x[i] == 1’b1) )
            seenTrailing = 1;
         else if ( x[i] == 1’b0 )
            seenZero = 1;
      end
   endfunction

   function [3:0] zeros;
   input [7:0] x;
   reg   [3:0] count;
   integer i;

   begin
      count = 0;
      for ( i=0; i <= 7; i=i+1 )
         if ( x[i] == 1’b0 ) count = count + 1;
         zeros = count;
      end
   endfunction
   wire is_legal = legal(in);
   assign error = ! is_legal;
   assign out   = is_legal ? zeros(in) : 1’b0;
endmodule
```



This example shows two Verilog functions: legal and zeros. The function legal determines if the value is valid. It returns a 1-bit value: either 1 for a valid value or 0 for an invalid value. The function zeros cycles through all bits of the value, counts the number of zeros, and returns the appropriate value. The two functions are controlled by continuous assignment statements at the bottom of the module definition. This example shows a combinational (parallel) approach to counting zeros; the next example shows a sequential (serial) approach.

Count Zeros—Sequential Version

Example A-2 shows a sequential (clocked) solution to the “count zeros” design problem. The circuit specification is slightly different from the specification in the combinational solution. The circuit now accepts the 8-bit string serially, 1 bit per clock cycle, using the data and clk inputs. The other two inputs are

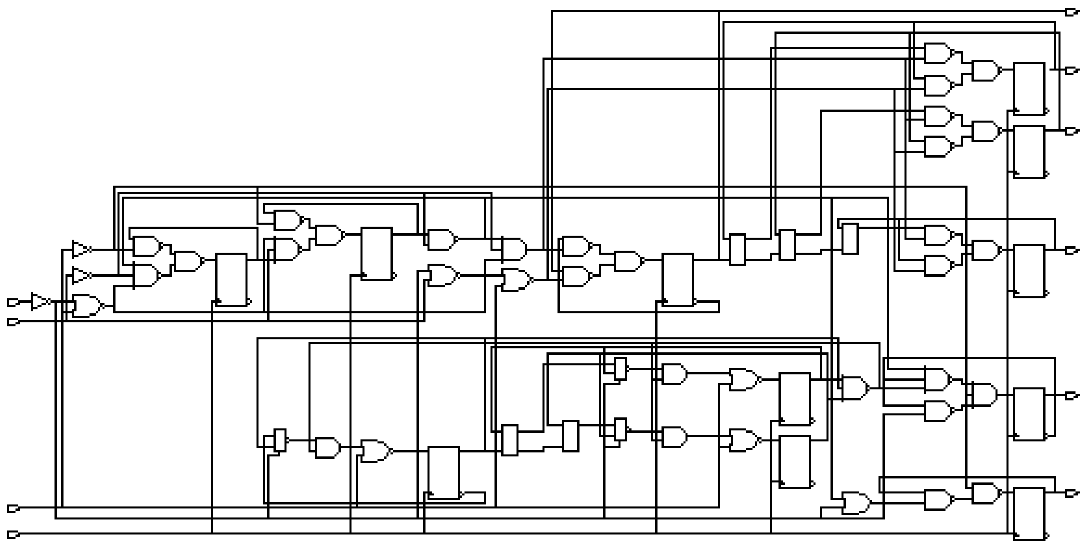
- `reset`, which resets the circuit
- `read`, which causes the circuit to begin accepting data

The circuit’s three outputs are

- `is_legal`, which is true if the data is a valid value
- `data_ready`, which is true at the first invalid bit or when all 8 bits have been processed
- `zeros`, which is the number of zeros if `is_legal` is true

Example A-2 Count Zeros—Sequential Version

```
module count_zeros(data,reset,read,clk,zeros,is_legal,
                   data_ready);

    parameter TRUE=1, FALSE=0;

    input  data, reset, read, clk;
    output is_legal, data_ready;
    output [3:0] zeros;
    reg  [3:0] zeros;

    reg is_legal, data_ready;
    reg seenZero, new_seenZero;
    reg seenTrailing, new_seenTrailing;
    reg new_is_legal;
    reg new_data_ready;
    reg [3:0] new_zeros;
    reg [2:0] bits_seen, new_bits_seen;

always @ ( data or reset or read or is_legal
           or data_ready or seenTrailing or
            seenZero or zeros or bits_seen ) begin
        if ( reset ) begin
            new_data_ready   = FALSE;
            new_is_legal     = TRUE;
            new_seenZero     = FALSE;
            new_seenTrailing = FALSE;
            new_zeros        = 0;
            new_bits_seen    = 0;
        end
        else begin
            new_is_legal     = is_legal;
            new_seenZero     = seenZero;
            new_seenTrailing = seenTrailing;
            new_zeros        = zeros;
            new_bits_seen    = bits_seen;
            new_data_ready   = data_ready;
             if ( read ) begin
               if ( seenTrailing  && (data == 0) )
                  begin
                  new_is_legal   = FALSE;
                  new_zeros      = 0;
                  new_data_ready = TRUE;
                  end
               else if ( seenZero && (data == 1’b1) )
                  new_seenTrailing = TRUE;
               else if ( data == 1’b0 ) begin
                  new_seenZero = TRUE;
                  new_zeros = zeros + 1;
                  end

     if ( bits_seen == 7 )
                  new_data_ready = TRUE;
               else
                  new_bits_seen = bits_seen+1;
            end
        end
    end

always @ ( posedge clk) begin
     zeros = new_zeros;
     bits_seen = new_bits_seen;
     seenZero = new_seenZero;
     seenTrailing = new_seenTrailing;
     is_legal = new_is_legal;
     data_ready = new_data_ready;
end
endmodule
```



Drink Machine—State Machine Version

The next design is a vending control unit for a soft drink vending machine. The circuit reads signals from a coin-input unit and sends outputs to a change-dispensing unit and a drink-dispensing unit.

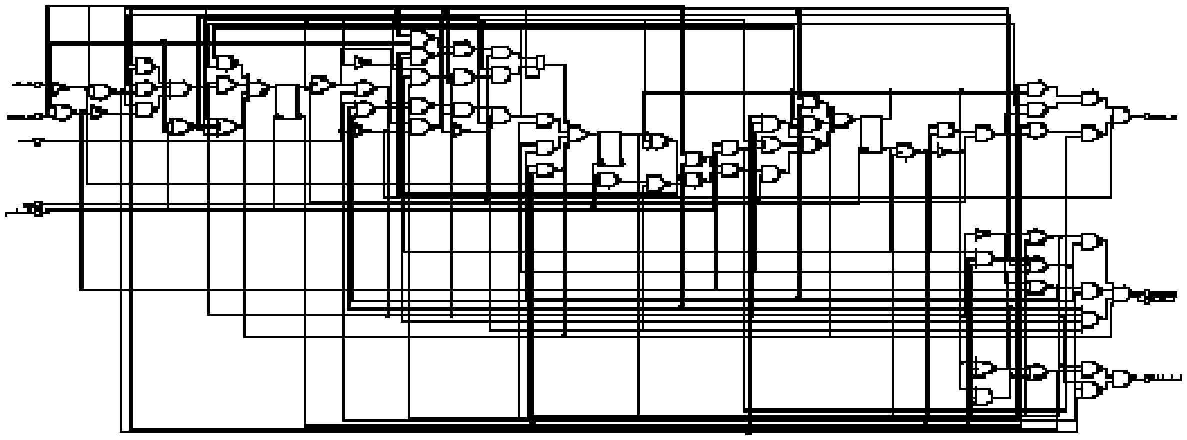
Input signals from the coin-input unit are `nickel_in` (nickel deposited), `dime_in` (dime deposited), and `quarter_in` (quarter deposited).

Outputs to the vending control unit are `collect` (collect coins), to the coin-input unit; `nickel_out` (nickel change) and `dime_out` (dime change), to the change-dispensing unit; and `dispense` (dispense drink), to the drink-dispensing unit.

The price of a drink is 35 cents. The Verilog description for this design, shown in Example A-3, uses a state machine description style. The description includes the `state_vector` directive, which enables Design Compiler to extract an equivalent state machine.

Example A-3 Drink Machine—State Machine Version

```
‘define vend_a_drink {D,dispense,collect} = {IDLE,2’b11}

module drink_machine(nickel_in, dime_in, quarter_in,
                     collect, nickel_out, dime_out,
                     dispense, reset, clk) ;
   parameter IDLE=0,FIVE=1,TEN=2,TWENTY_FIVE=3,
             FIFTEEN=4,THIRTY=5,TWENTY=6,OWE_DIME=7;

   input  nickel_in, dime_in, quarter_in, reset, clk;
   output collect, nickel_out, dime_out, dispense;

   reg collect, nickel_out, dime_out, dispense;
   reg [2:0] D, Q; /* state */
// synopsys state_vector Q

always @ ( nickel_in or dime_in or quarter_in or reset )
      begin
         nickel_out = 0;
         dime_out   = 0;
         dispense   = 0;
         collect    = 0;

         if ( reset ) D = IDLE;
         else begin
            D = Q;

            case ( Q )
            IDLE:
               if (nickel_in)       D = FIVE;
               else if (dime_in)    D = TEN;
               else if (quarter_in) D = TWENTY_FIVE;
            FIVE:
               if(nickel_in)        D = TEN;
               else if (dime_in)    D = FIFTEEN;
               else if (quarter_in) D = THIRTY;
            TEN:
               if (nickel_in)       D = FIFTEEN;
               else if (dime_in)    D = TWENTY;
               else if (quarter_in) ‘vend_a_drink;
            TWENTY_FIVE:
               if( nickel_in)       D = THIRTY;
               else if (dime_in)    ‘vend_a_drink;
               else if (quarter_in) begin

                  ‘vend_a_drink;
                   nickel_out = 1;
                   dime_out = 1;
               end

            FIFTEEN:
               if (nickel_in)       D = TWENTY;
               else if (dime_in)    D = TWENTY_FIVE;
               else if (quarter_in) begin
                   ‘vend_a_drink;
                   nickel_out = 1;
               end

            THIRTY:
               if (nickel_in)       ‘vend_a_drink;
               else if (dime_in)    begin
                   ‘vend_a_drink;
                   nickel_out = 1;
               end
               else if (quarter_in) begin
                   ‘vend_a_drink;
                   dime_out = 1;
                   D = OWE_DIME;
               end

            TWENTY:
               if (nickel_in)       D = TWENTY_FIVE;
               else if (dime_in)    D = THIRTY;
               else if (quarter_in) begin
                   ‘vend_a_drink;
                   dime_out = 1;
               end

            OWE_DIME:
               begin
                   dime_out = 1;
                   D = IDLE;
               end
            endcase
    end
end

always @ (posedge clk ) begin
     Q = D;
end
endmodule
```



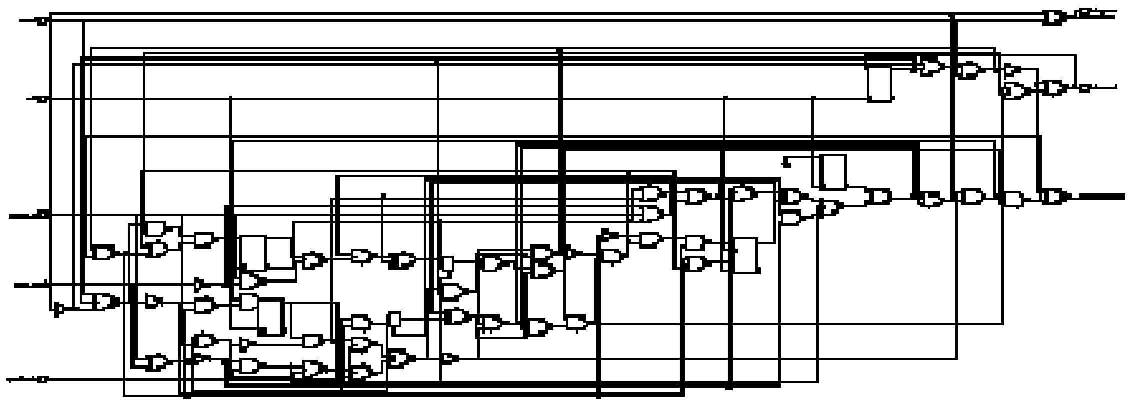
Drink Machine—Count Nickels Version

Example A-4 uses the same design parameters as Example A-3 with the same input and output signals. In this version, a counter counts the number of nickels deposited. This counter is incremented by one if the deposit is a nickel, by two if it’s a dime, and by five if it’s a quarter.

Example A-4 Drink Machine—Count Nickels Version

```
module drink_machine(nickel_in,dime_in,quarter_in,collect,
       nickel_out,dime_out,dispense,reset,clk);

        input nickel_in, dime_in, quarter_in, reset, clk;
        output nickel_out, dime_out, collect, dispense;

        reg nickel_out, dime_out, dispense, collect;
        reg [3:0] nickel_count, temp_nickel_count;
        reg temp_return_change, return_change;

             always @ ( nickel_in or dime_in or quarter_in or
             collect or temp_nickel_count or
             reset or nickel_count or return_change) begin
                   nickel_out = 0;
                   dime_out   = 0;
                   dispense   = 0;
                   collect    = 0;
                   temp_nickel_count = 0;
                   temp_return_change = 0;

                   // Check whether money has come in
                   if (! reset) begin
                        temp_nickel_count = nickel_count;
                        if (nickel_in)
                          temp_nickel_count = temp_nickel_count + 1;
                        else if (dime_in)
                          temp_nickel_count = temp_nickel_count + 2;
                        else if (quarter_in)
                          temp_nickel_count = temp_nickel_count + 5;

                   // correct amount deposited?
                   if (temp_nickel_count >= 7) begin
                        temp_nickel_count = temp_nickel_count - 7;
                        dispense = 1;
                        collect = 1;
                   end
```

```
                    // return change
                    if (return_change || collect) begin
                          if (temp_nickel_count >= 2) begin
                            dime_out = 1;
                            temp_nickel_count = temp_nickel_count - 2;
                            temp_return_change = 1;
                          end

                          if (temp_nickel_count == 1) begin
                            nickel_out = 1;
                            temp_nickel_count = temp_nickel_count - 1;
                          end
                    end
              end
        end
        always @ (posedge clk ) begin
              nickel_count = temp_nickel_count;
              return_change = temp_return_change;
        end
endmodule
```



Carry-Lookahead Adder

Figure A-1 on page A-17 and Example A-5 on page A-18 show how to build a 32-bit carry-lookahead adder. The adder is built by partitioning of the 32-bit input into eight slices of 4 bits each. The PG module computes propagate and generate values for each of the eight slices.

Propagate (output P from PG) is 1 for a bit position if that position propagates a carry from the next-lower position to the next-higher position. Generate (output G) is 1 for a bit position if that position generates a carry to the next-higher position, regardless of the carry-in from the next-lower position.

The carry-lookahead logic reads the carry-in, propagate, and generate information computed from the inputs. It computes the carry value for each bit position. This logic makes the addition operation an XOR of the inputs and the carry values.

The following list shows the order in which the carry values are computed by a three-level tree of 4-bit carry-lookahead blocks (illustrated in Figure A-1):

1. The first level of the tree computes the 32 carry values and the 8 group propagate and generate values. Each of the first-level group propagate and generate values tells if that 4-bit slice propagates and generates carry values from the next-lower group to the next-higher. The first-level lookahead blocks read the group carry computed at the second level.

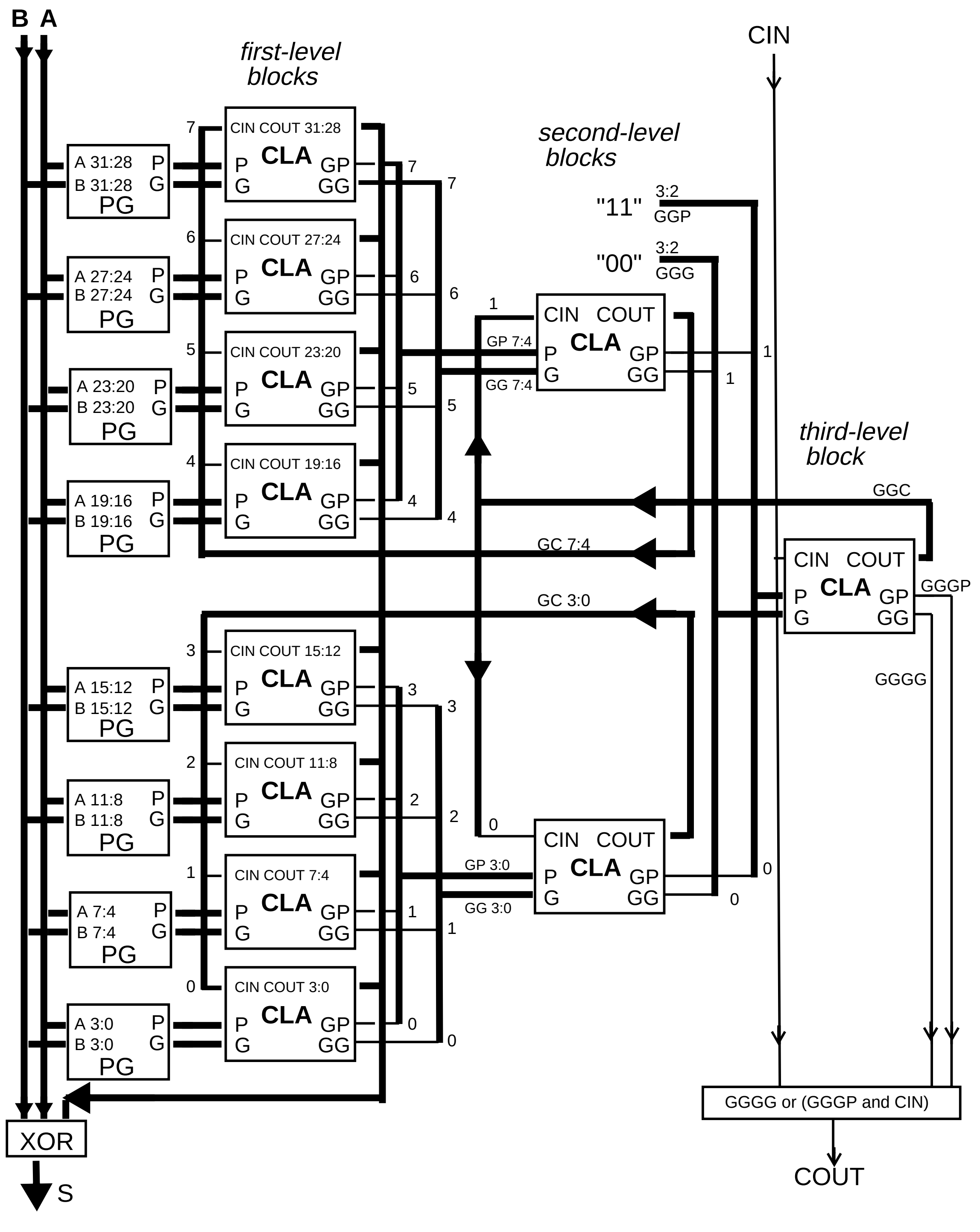
2. At the second level of the tree, the lookahead blocks read the group propagate and generate information from the four first-level blocks and then compute their own group propagate and generate information. They also read group carry information computed at the third level to compute the carries for each of the third-level blocks.

3. At the third level of the tree, the third-level block reads the propagate and generate information of the second level to compute a propagate and generate value for the entire adder. It also reads the external carry to compute each second-level carry. The carry-out for the adder is 1 if the third-level generate is 1 or if the third-level propagate is 1 and the external carry is 1.

   The third-level carry-lookahead block can process four second-level blocks. Because there are only two second-level blocks in Figure A-1, the high-order 2 bits of the computed carry are ignored, the high-order 2 bits of the generate input to the third-level are set to 00 (zero), and the propagate high-order bits are set to 11. This causes the unused portion to propagate carries but not to generate them.

Figure A-1 shows the three levels of a block diagram of the 32-bit carry-lookahead adder. Example A-5 shows the code for the adder.

Figure A-1 Carry-Lookahead Adder Block Diagram



Example A-5 Carry-Lookahead Adder

```
‘define word_size 32
‘define word [‘word_size-1:0]

‘define n 4
‘define slice [‘n-1:0]

‘define s0 (1*‘n)-1:0*‘n
‘define s1 (2*‘n)-1:1*‘n
‘define s2 (3*‘n)-1:2*‘n
‘define s3 (4*‘n)-1:3*‘n
‘define s4 (5*‘n)-1:4*‘n
‘define s5 (6*‘n)-1:5*‘n
‘define s6 (7*‘n)-1:6*‘n
‘define s7 (8*‘n)-1:7*‘n

module cla32_4(a, b, cin, s, cout);
input ‘word a, b;
input cin;
output ‘word s;
output cout;

 wire [7:0] gg, gp, gc; // Group generate, propagate,
                                   // carry
 wire [3:0] ggg, ggp, ggc;// Second-level gen., prop.
 wire gggg, gggp; // Third-level gen., prop.

 bitslice i0(a[‘s0], b[‘s0], gc[0], s[‘s0], gp[0], gg[0]);
 bitslice i1(a[‘s1], b[‘s1], gc[1], s[‘s1], gp[1], gg[1]);
 bitslice i2(a[‘s2], b[‘s2], gc[2], s[‘s2], gp[2], gg[2]);
 bitslice i3(a[‘s3], b[‘s3], gc[3], s[‘s3], gp[3], gg[3]);

 bitslice i4(a[‘s4], b[‘s4], gc[4], s[‘s4], gp[4], gg[4]);
 bitslice i5(a[‘s5], b[‘s5], gc[5], s[‘s5], gp[5], gg[5]);
 bitslice i6(a[‘s6], b[‘s6], gc[6], s[‘s6], gp[6], gg[6]);
 bitslice i7(a[‘s7], b[‘s7], gc[7], s[‘s7], gp[7], gg[7]);

 cla c0(gp[3:0], gg[3:0], ggc[0], gc[3:0], ggp[0], ggg[0]);
 cla c1(gp[7:4], gg[7:4], ggc[1], gc[7:4], ggp[1], ggg[1]);

 assign ggp[3:2] = 2’b11;
 assign ggg[3:2] = 2’b00;
 cla c2(ggp, ggg, cin, ggc, gggp, gggg);
 assign cout = gggg | (gggp & cin);
endmodule

// Compute sum and group outputs from a, b, cin
```

```
module bitslice(a, b, cin, s, gp, gg);
input ‘slice a, b;
input cin;
output ‘slice s;
output gp, gg;

 wire ‘slice p, g, c;
 pg i1(a, b, p, g);
 cla i2(p, g, cin, c, gp, gg);
 sum i3(a, b, c, s);
endmodule

// compute propagate and generate from input bits

module pg(a, b, p, g);
input ‘slice a, b;
output ‘slice p, g;

 assign p = a | b;
 assign g = a & b;
endmodule

// compute sum from the input bits and the carries

module sum(a, b, c, s);
input ‘slice a, b, c;
output ‘slice s;

 wire ‘slice t = a ^ b;
 assign s = t ^ c;
endmodule

// n-bit carry-lookahead block

module cla(p, g, cin, c, gp, gg);
input ‘slice p, g;// propagate and generate bits
input cin;    // carry in
output ‘slice c; // carry produced for each bit
output gp, gg; // group generate and group propagate

 function [99:0] do_cla;
 input ‘slice p, g;
 input cin;

 begin : label
 integer i;
 reg gp, gg;
 reg ‘slice c;
 gp = p[0];
 gg = g[0];
 c[0] = cin;
 for(i = 1; i < ‘n; i = i+1) begin
        gp = gp & p[i];
        gg = (gg & p[i]) | g[i];
        c[i] = (c[i-1] & p[i-1]) | g[i-1];
 end
 do_cla = {c, gp, gg};
 end
 endfunction

 assign {c, gp, gg} = do_cla(p, g, cin);
endmodule
```