EE 3755

Verilog Handout #5

```
/// LSU EE 3755 -- Fall 2001 -- Computer Organization
//
/// Verilog Notes 5 -- Integer Addition and Subtraction

/// Contents
//
// Representation of Integers in Computers
// Binary Addition and Subtraction
// Construction of ALUs
// Carry Look Ahead Adders
// Flat Carry Look-Ahead Adder
// Two-Level Carry Look-Ahead Adder
// Real-World Integer Adders (Under Construction)

/// References
//
// :P:   Palnitkar, "Verilog HDL"
// :Q:   Qualis, "Verilog HDL Quick Reference Card Revision 1.0"
// :H:   Hyde, "Handbook on Verilog HDL"
// :LRM: IEEE, Verilog Language Reference Manual  (Hawaii Section Numbering)
// :PH:  Patterson & Hennessy, "Computer Organization & Design"


////////////////////////////////////////////////////////////////////////////////
/// Representation of Integers in Computers

// :PH: 4.1, 4.2

// Difference between real-world and computer integers:
//
//  Real-world integers don't have a maximum or minimum.
//  Computer integers do because they are represented by a fixed
//    number of bits. (With a few exceptions.)
//
//  Keep this in mind for overflow and negative numbers.

 /// Two's Complement Representation
//
// Since the use of two's complement is nearly universal, "signed integer"
// is usually understood to mean "two's complement signed integer."
//
// Let i be a positive integer.  The /n-bit two's complement representation/
// of -i is obtained by:
//
//   Start with the n-bit binary representation of i.
//   Invert the bits in i.
//   Add 1.
//
//   In Verilog or C: minus i =  ~i + 1;
//

 /// Two's Complement Sign Bit
//
// A two's complement representation is negative if and only if the
// most-significant bit is 1.
//
// Not surprisingly, the MSB is called the /sign bit/.

 /// Example:  Find 4-bit representation of -5.
//
//   Start:  +5 = 0101
//   Invert:      1010
//   Add 1:       1011
//   So -5 = 1011

 /// Range of Representation:
//
// (Smallest and largest representable numbers.)
```

```
//
// Unsigned n-bit range:
//   Minimum: 0
//   Maximum: (2^n)-1   Two-to-the-n minus 1.
//   Example: 4-bit: Minimum 0, maximum 15.
//
// Signed 2's complement n-bit range:
//   Minimum: -2^{n-1}
//   Maximum: 2^{n-1}-1
//   Example: 4-bit:  Minimum -8, maximum 7.

// Remember These 4-bit Signed Numbers:
//    0:  0000
//   -1:  1111
//   -2:  1110
//   -7:  1001
//   -8:  1000


////////////////////////////////////////////////////////////////////////////////
/// Binary Addition and Subtraction

// :PH: 4.3 and classroom lectures.

// If necessary, review binary addition.
// Notes here discuss overflow.

 /// Overflow Definition
//
// Describes an arithmetic operation in which the result cannot
// be represented.  The result might be too large or too small.

 /// Overflow in Unsigned Addition
//
// There is overflow in unsigned addition of two n-bit numbers
// if there is a carry out (from the MSB).

// Example:
//
// Addition of 4-bit unsigned representations of 10 and 7:
//
//  1010
// +0111
// -----
// 10001
// C
//
// The digit over the "C" is the carry out.  Since it is 1 the
// addition overflowed.

 /// Overflow Detection
//
// There has been overflow in the addition of two n-bit two's complement
// numbers when the sign of the two operands are the same and the sign
// of the sum is different.
//
// This implies that there can not be overflow when the signs of the
// two operands are different.
//
// A carry out does not necessarily indicate overflow.


// :Example:
//
//  6:  0110   Sign of first operand: 0 (positive)
// -2:  1110   Sign of second operand: 1 (negative)
//     -----
//  4: 10100   Sign of sum: 0 (positive), sum correct
```

```
//     CS
//
// The digit over the C is the carry out.  Since this is signed
// addition, ignore it.
//
// The sign of the two operands are different, so overflow is not possible.

// :Example:
//
//  6:  0110   Sign of first operand: 0 (positive)
//  2:  0010   Sign of second operand: 0 (positive)
//     -----
// -8: 01000   Sign of sum: 1 (negative), addition overflowed so sum wrong.
//     CS
//
// The digit over the C is the carry out.  In this case there is no
// carry out, which is irrelevant since carry out is ignored.
//
// The sign of the two operands are the same, 0, and is different than
// the sign of the sum, 1, so there is overflow.

// :Example:
//
// -6:  1010   Sign of first operand: 1 (negative)
// -2:  1110   Sign of second operand: 1 (negative)
//     -----
// -8: 11000   Sign of sum: 1 (negative), no overflow, sum correct.
//     CS
//
// The digit over the C is the carry out.
//
// The sign of the two operands are the same, 1, and is the same as
// the sign of the sum, 1, so there is no overflow.



////////////////////////////////////////////////////////////////////////////////
/// Construction of ALUs

// :PH: 4.5

// /ALU/: The primary device in a computer for performing integer
// arithmetic and logical operations.  [There may be other /functional units/
// that perform additional integer and also floating-point operations.]

// Goal: Design ALU that can share as much hardware as possible.
//
// First ALU below does not share much hardware.
// In the second ALU below adder is shared for addition, subtraction, and
//   less than.


 /// First ALU
//
// Operations: and (0), or (1), add (2)
//   Numbers refer to "op" (operation) input.

// Step 1:
// Define a slice module that can perform any of the operations
// (and, or, add) on two one-bit inputs and a carry.
//
// Step 2:
// Connect the slices to construct the entire ALU.


// :Example:
//
// Slice for first alu.  See :PH: Figure 4.14
```

```
module alu_slice(result,cout,a,b,cin,op);
   input a, b, cin;
   input [1:0] op;
   output      result, cout;

   parameter   op_and = 0;
   parameter   op_or  = 1;
   parameter   op_add = 2;

   bfa_implicit bfa(sum,cout,a,b,cin);

   assign result =
          op == op_and ? a & b :
          op == op_or  ? a | b :
          op == op_add ? sum : 0;

endmodule


// :Example:
//
// ALU using slice above.  See :PH: Figure 4.15.

module alu(result,a,b,cin,op);
   input [31:0] a, b;
   input        cin;
   input [1:0]  op;
   output [31:0] result;

   wire [31:0]   cout;

   alu_slice slice0(result[0],cout[0],a[0],b[0],1'b0,op);
   alu_slice slice1(result[1],cout[1],a[1],b[1],cout[0],op);
   alu_slice slice2(result[2],cout[2],a[2],b[2],cout[1],op);
   alu_slice slice3(result[3],cout[3],a[3],b[3],cout[2],op);
   alu_slice slice4(result[4],cout[4],a[4],b[4],cout[3],op);
   alu_slice slice5(result[5],cout[5],a[5],b[5],cout[4],op);
   alu_slice slice6(result[6],cout[6],a[6],b[6],cout[5],op);
   alu_slice slice7(result[7],cout[7],a[7],b[7],cout[6],op);
   alu_slice slice8(result[8],cout[8],a[8],b[8],cout[7],op);
   alu_slice slice9(result[9],cout[9],a[9],b[9],cout[8],op);
   alu_slice slice10(result[10],cout[10],a[10],b[10],cout[9],op);
   alu_slice slice11(result[11],cout[11],a[11],b[11],cout[10],op);
   alu_slice slice12(result[12],cout[12],a[12],b[12],cout[11],op);
   alu_slice slice13(result[13],cout[13],a[13],b[13],cout[12],op);
   alu_slice slice14(result[14],cout[14],a[14],b[14],cout[13],op);
   alu_slice slice15(result[15],cout[15],a[15],b[15],cout[14],op);
   alu_slice slice16(result[16],cout[16],a[16],b[16],cout[15],op);
   alu_slice slice17(result[17],cout[17],a[17],b[17],cout[16],op);
   alu_slice slice18(result[18],cout[18],a[18],b[18],cout[17],op);
   alu_slice slice19(result[19],cout[19],a[19],b[19],cout[18],op);
   alu_slice slice20(result[20],cout[20],a[20],b[20],cout[19],op);
   alu_slice slice21(result[21],cout[21],a[21],b[21],cout[20],op);
   alu_slice slice22(result[22],cout[22],a[22],b[22],cout[21],op);
   alu_slice slice23(result[23],cout[23],a[23],b[23],cout[22],op);
   alu_slice slice24(result[24],cout[24],a[24],b[24],cout[23],op);
   alu_slice slice25(result[25],cout[25],a[25],b[25],cout[24],op);
   alu_slice slice26(result[26],cout[26],a[26],b[26],cout[25],op);
   alu_slice slice27(result[27],cout[27],a[27],b[27],cout[26],op);
   alu_slice slice28(result[28],cout[28],a[28],b[28],cout[27],op);
   alu_slice slice29(result[29],cout[29],a[29],b[29],cout[28],op);
   alu_slice slice30(result[30],cout[30],a[30],b[30],cout[29],op);
   alu_slice slice31(result[31],cout[31],a[31],b[31],cout[30],op);

endmodule
```

```
/// Second ALU
//
// This ALU can perform additional operations: subtract and less than.
// Both subtract and less than use the adder's hardware.


// :Example:
//
// Slice for second ALU.  See :PH: Figure 4.16.  Input less is
// generated by MSB slice, alu_slice2MSB for bit zero and 0 for all
// the others.

module alu_slice2(result,cout,a,b,cin,op,binvert,less);
   input a, b, cin, binvert, less;
   input [1:0] op;
   output      result, cout;

   parameter   sop_and = 0;
   parameter   sop_or  = 1;
   parameter   sop_add = 2;
   parameter   sop_slt = 3;

   wire        b2 = binvert ? ~b : b;
   wire        sum;

   bfa_implicit bfa(sum,cout,a,b2,cin);

   assign result =
          op == sop_and ? a & b :
          op == sop_or  ? a | b :
          op == sop_add ? sum :
          less; // else, op == sop_slt

endmodule


// :Example:
//
// Bit slice for bit 31.  This also generates a set and overflow
// output.

module alu_slice2MSB(result,set,overflow,a,b,cin,op,binvert,less);
   input a, b, cin, binvert, less;
   input set, overflow;
   input [1:0] op;
   output      result;

   parameter   sop_and = 0;
   parameter   sop_or  = 1;
   parameter   sop_add = 2;
   parameter   sop_slt = 3;

   wire        b2 = binvert ? ~b : b;
   wire        sum;

   assign overflow =
          op != sop_add ? 1'b0 :
          a != b2       ? 1'b0 :
          a == sum      ? 1'b0 : 1'b1;

   assign set = a != b2 ? sum : a;

   bfa_implicit bfa(sum,cout,a,b2,cin);

   assign result =
          op == sop_and ? a & b :
          op == sop_or  ? a | b :
          op == sop_add ? sum :
```

```
        less; // else, op == sop_slt

endmodule


// :Example:
//
// Second ALU.  The same module is used for bits 0-30, bit 31
// is computed by a special module that also determines overflow
// and whether a < b.

module alu2(result,overflow,a,b,alu_op);
   input [31:0] a, b;
   input [2:0]  alu_op;
   output [31:0] result;
   output        overflow;

   wire [31:0]   cout;

   parameter     op_and = 0;
   parameter     op_or  = 1;
   parameter     op_slt = 2;
   parameter     op_add = 3;
   parameter     op_sub = 4;

   parameter     sop_and = 0;
   parameter     sop_or  = 1;
   parameter     sop_add = 2;
   parameter     sop_slt = 3;

   wire [1:0] op =
              alu_op == op_and ? sop_and :
              alu_op == op_or  ? sop_or  :
              alu_op == op_slt ? sop_slt :
              alu_op == op_add ? sop_add :
              alu_op == op_sub ? sop_add : 0;

   wire       binvert = alu_op == op_sub || alu_op == op_slt;

   wire       set;  // MSB of sum.

   alu_slice2 slice0(result[0],cout[0],a[0],b[0],binvert,op,binvert,set);
   alu_slice2 slice1(result[1],cout[1],a[1],b[1],cout[0],op,binvert,1'b0);
   alu_slice2 slice2(result[2],cout[2],a[2],b[2],cout[1],op,binvert,1'b0);
   alu_slice2 slice3(result[3],cout[3],a[3],b[3],cout[2],op,binvert,1'b0);
   alu_slice2 slice4(result[4],cout[4],a[4],b[4],cout[3],op,binvert,1'b0);
   alu_slice2 slice5(result[5],cout[5],a[5],b[5],cout[4],op,binvert,1'b0);
   alu_slice2 slice6(result[6],cout[6],a[6],b[6],cout[5],op,binvert,1'b0);
   alu_slice2 slice7(result[7],cout[7],a[7],b[7],cout[6],op,binvert,1'b0);
   alu_slice2 slice8(result[8],cout[8],a[8],b[8],cout[7],op,binvert,1'b0);
   alu_slice2 slice9(result[9],cout[9],a[9],b[9],cout[8],op,binvert,1'b0);
   alu_slice2 slice10(result[10],cout[10],a[10],b[10],cout[9],op,binvert,1'b0);
   alu_slice2 slice11(result[11],cout[11],a[11],b[11],cout[10],op,binvert,1'b0);
   alu_slice2 slice12(result[12],cout[12],a[12],b[12],cout[11],op,binvert,1'b0);
   alu_slice2 slice13(result[13],cout[13],a[13],b[13],cout[12],op,binvert,1'b0);
   alu_slice2 slice14(result[14],cout[14],a[14],b[14],cout[13],op,binvert,1'b0);
   alu_slice2 slice15(result[15],cout[15],a[15],b[15],cout[14],op,binvert,1'b0);
   alu_slice2 slice16(result[16],cout[16],a[16],b[16],cout[15],op,binvert,1'b0);
   alu_slice2 slice17(result[17],cout[17],a[17],b[17],cout[16],op,binvert,1'b0);
   alu_slice2 slice18(result[18],cout[18],a[18],b[18],cout[17],op,binvert,1'b0);
   alu_slice2 slice19(result[19],cout[19],a[19],b[19],cout[18],op,binvert,1'b0);
   alu_slice2 slice20(result[20],cout[20],a[20],b[20],cout[19],op,binvert,1'b0);
   alu_slice2 slice21(result[21],cout[21],a[21],b[21],cout[20],op,binvert,1'b0);
   alu_slice2 slice22(result[22],cout[22],a[22],b[22],cout[21],op,binvert,1'b0);
   alu_slice2 slice23(result[23],cout[23],a[23],b[23],cout[22],op,binvert,1'b0);
   alu_slice2 slice24(result[24],cout[24],a[24],b[24],cout[23],op,binvert,1'b0);
   alu_slice2 slice25(result[25],cout[25],a[25],b[25],cout[24],op,binvert,1'b0);
   alu_slice2 slice26(result[26],cout[26],a[26],b[26],cout[25],op,binvert,1'b0);
```

ALU

slice of ALU

```
   alu_slice2 slice27(result[27],cout[27],a[27],b[27],cout[26],op,binvert,1'b0);
   alu_slice2 slice28(result[28],cout[28],a[28],b[28],cout[27],op,binvert,1'b0);
   alu_slice2 slice29(result[29],cout[29],a[29],b[29],cout[28],op,binvert,1'b0);
   alu_slice2 slice30(result[30],cout[30],a[30],b[30],cout[29],op,binvert,1'b0);
   alu_slice2MSB slice31(result[31],set,overflow,a[31],b[31],
                         cout[30],op,binvert, 1'b0);

endmodule


// :Example:
//
// Testbench for second ALU.  Material for this testbench not covered.

module test_alu();

   wire [31:0] result;
   wire        overflow;
   reg [31:0]  a, b;
   reg [2:0]   alu_op;

   alu2 alu(result,overflow,a,b,alu_op);

   integer     i, shadow_result, shadow_overflow, as, bs;

   initial begin

      alu_op = alu.op_sub;

      for(i=0; i<10000; i=i+1) begin

         as = $random;
         a = as;
         bs = $random;
         b = bs;

         alu_op = ( $random & 32'h7fffffff ) % 5;

         case( alu_op )
           alu.op_and:
             begin
                shadow_result = as & bs;
                shadow_overflow = 0;
             end
           alu.op_or:
             begin
                shadow_result = as | bs;
                shadow_overflow = 0;
             end
           alu.op_add:
             begin
                shadow_result = as + bs;
                shadow_overflow = as > 0 && bs > 0 && shadow_result < 0
                                  || as < 0 && bs < 0 && shadow_result > 0;
             end
           alu.op_sub:
             begin
                shadow_result = as - bs;
                shadow_overflow = as > 0 && bs < 0 && shadow_result < 0
                                  || as < 0 && bs > 0 && shadow_result > 0;
             end
           alu.op_slt:
             begin
                shadow_result = as < bs;
                shadow_overflow = 0;
             end
         endcase
```

```
         #1;

         if( shadow_result != result ) begin
            $display("Wrong result op %d,  %x op %x = %x != %x",
                     alu_op, a, b, shadow_result, result);
            $stop;
         end
         if( shadow_overflow != overflow ) begin
            $display("Wrong overflow op %d,  %x op %x;  %d != %d",
                     alu_op, a, b, shadow_overflow, overflow);
            $stop;
         end
      end
      $display("Done with test.");
   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Carry Look Ahead Adders

// :PH: 4.5 (In middle of section 4.5.)

 /// Motivation
//
// Ripple adders take to long:
//   Delay of n-bit adder is 2n+1 gate delays.
//   Modern processors need 1-cycle 64-bit adders,
//      that's 129 gate delays in one cycle. A ripple adder is too slow.

 /// Carry Look-Ahead Adder Properties
//
// More expensive but much faster.
// Cost varies by configuration.
//
// Flat (One-level) Carry Look-Ahead Adder
//   Delay is six gate delays, regardless of size. (Counting xor as 3 gates.)
//   Cost is proportional to square of number of bits.
//   Requires large fan-in gates, which may not be practical.
//
// Multiple Level Carry Look-Ahead Adders.  (Second level of abstraction.)
//   Delay typically proportional to square root number of bits or lower.
//   Cost lower than flat CLA.
//   Designed to fit technology constraints and capabilities (fan-in, etc.)


////////////////////////////////////////////////////////////////////////////////
/// Flat Carry Look-Ahead Adder

// :PH: 4.5 (In middle of section 4.5.)

 /// Flat Carry Look-Ahead Adder Structure
//
// At each slice (pair of input bits and output bit) create a
//   /generate/ and /propagate/ signal to be used by more-significant
//   bits.
//
// At each slice use generate and propagate signals of less-significant
//   bits to produce a sign bit.
//
// See examples below.


// :Example:
//
// A slice for a CLA.  Like a BFA, its inputs are one bit
// of each operand, a and b, and a carry in, cin.  Its outputs
```

```
// are one bit of the sum and, instead of a cout, a generate
// and propagate signal.

module cla_slice(sum,g,p,a,b,cin);
   input a, b;
   input cin; // Determined using other generate and propagates.
   output sum;
   output g; // Generate
   output p; // Propagate

   /// Generate
   //
   // If a and b are 1 a carry will be generated here.
   //
   assign g = a & b;

   /// Propagate
   //
   // If a or b are 1 a carry in to this digit position will
   // "propagate through" to a carry out.  (There is no actual
   // carry out signal.)
   //
   assign p = a | b;

   /// Sum
   //
   // This is the usual formula for the sum.
   //
   assign sum =
          ~a & ~b &  cin |
          ~a &  b & ~cin |
           a & ~b & ~cin |
           a &  b &  cin;

endmodule


// :Example:
//
// A 5-bit CLA using five cla slices.  The generate and propagate
// signals are used to produce the carry in signals.

module cla_5(sum,a,b);
   input [4:0] a, b;
   output [5:0] sum;

   wire [4:0]   g, p, carry;

   /// Logic for Carry In Signals
   //
   // General idea:  for there to be a carry at bit x either:
   //   either the carry is generated at bit x-1
   //   or the carry is generated at bit x-2 and prop. through x-1
   //   or the carry is generated at bit x-3 and prop. through x-1 and x-1
   //   ...
   //   or the carry is generated at bit 0 and prop. through 1 and ... and x-1

   assign       carry[0] = 1'b0;

   assign       carry[1] = g[0];

   assign       carry[2] = g[0] & p[1]
                         | g[1];

   assign       carry[3] = g[0] & p[1] & p[2]
                         | g[1] & p[2]
                         | g[2];
```

```
   assign       carry[4] = g[0] & p[1] & p[2] & p[3]
                         | g[1] & p[2] & p[3]
                         | g[2] & p[3]
                         | g[3];

   assign sum[5] = g[0] & p[1] & p[2] & p[3] & p[4]
                 | g[1] & p[2] & p[3] & p[4]
                 | g[2] & p[3] & p[4]
                 | g[3] & p[4]
                 | g[4];

   cla_slice s0(sum[0],g[0],p[0],a[0],b[0],carry[0]);
   cla_slice s1(sum[1],g[1],p[1],a[1],b[1],carry[1]);
   cla_slice s2(sum[2],g[2],p[2],a[2],b[2],carry[2]);
   cla_slice s3(sum[3],g[3],p[3],a[3],b[3],carry[3]);
   cla_slice s4(sum[4],g[4],p[4],a[4],b[4],carry[4]);

endmodule

// :Example:
//
// Testbench for CLA.  The testbench includes material not covered
// in the course.

module testcla();

   wire [5:0] sum;
   reg [5:0] shadow_sum;
   reg [4:0]  a, b;
   integer    i;

   cla_5 c(sum, a, b);

   initial begin

      for(i=0; i<1024; i=i+1) begin

         a = i[4:0];
         b = i[9:5];
         shadow_sum = a + b;
         #1;
         if( sum != shadow_sum ) begin
            $display("Wrong sum: %h + %h = %h != %h\n",
                     a, b, shadow_sum, sum);
            $stop;
         end
      end

      $display("Tests completed.");

   end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Two-Level Carry Look-Ahead Adder

// :PH: 4.5 (In middle of section 4.5.)  Also see Figure 4.24

// Goal: Avoid large fan-ins of flat CLA.
//
// Idea:
//   Divide adder into multiple adder blocks.
//   Each adder block is itself a CLA.
//   Adder block produces "super" generate and propagate signals.
//   Use these to generate cin for adder blocks.
```

```
// Below a 12-bit two-level CLA is constructed using three 4-bit adder
// blocks.
//
// For comparison, these are followed by a flat 12-bit CLA, a 12-bit
// ripple adder, and a testbench.


// :Example:
//
// A four-bit adder block.  This uses the same cla_slice as the
// earlier CLA.

module cla_4_block(sum,gout,pout,a,b,cin);
   input [3:0]  a, b;
   input        cin;
   output [3:0] sum;
   output       gout, pout;

   wire [3:0]   carry, p, g;

   assign carry[0] = cin;

   assign carry[1] = cin  & p[0]
                   | g[0];

   assign carry[2] = cin  & p[0] & p[1]
                   | g[0] & p[1]
                   | g[1];

   assign carry[3] = cin  & p[0] & p[1] & p[2]
                   | g[0] & p[1] & p[2]
                   | g[1] & p[2]
                   | g[2];

   assign gout = g[0] & p[1] & p[2] & p[3]
               | g[1] & p[2] & p[3]
               | g[2] & p[3]
               | g[3];

   assign pout = p[0] & p[1] & p[2] & p[3];

   cla_slice s0(sum[0],g[0],p[0],a[0],b[0],carry[0]);
   cla_slice s1(sum[1],g[1],p[1],a[1],b[1],carry[1]);
   cla_slice s2(sum[2],g[2],p[2],a[2],b[2],carry[2]);
   cla_slice s3(sum[3],g[3],p[3],a[3],b[3],carry[3]);

endmodule


// :Example:
//
// The 12-bit two-level CLA.

module cla_12_two_level(sum,a,b);
   input [11:0]  a, b;
   output [12:0] sum;

   wire [2:0]  P, G, carry;

   assign      carry[0] = 1'b0;

   assign      carry[1] = G[0];

   assign      carry[2] = G[0] & P[1]
                        | G[1];

   assign      sum[12] = G[0] & P[1] & P[2]
```

```
                    | G[1] & P[2]
                    | G[2];

   cla_4_block ad0(sum[3:0],  G[0], P[0], a[3:0],  b[3:0],  carry[0]);
   cla_4_block ad1(sum[7:4],  G[1], P[1], a[7:4],  b[7:4],  carry[1]);
   cla_4_block ad2(sum[11:8], G[2], P[2], a[11:8], b[11:8], carry[2]);

endmodule


// :Example:
//
// A 12-bit flat CLA.  For comparison with the two-level adder.

module cla_12_flat(sum,a,b);
   input [11:0] a, b;
   output [12:0] sum;

   wire [11:0]   g, p, carry;

   assign       carry[0] = 1'b0;

   assign       carry[1] = 1'b0;

   assign       carry[2] = g[0] & p[1]
                | g[1];

   assign       carry[3] = g[0] & p[1] & p[2]
                | g[1] & p[2]
                | g[2];

   assign       carry[4] = g[0] & p[1] & p[2] & p[3]
                | g[1] & p[2] & p[3]
                | g[2] & p[3]
                | g[3];

   assign       carry[5] = g[0] & p[1] & p[2] & p[3] & p[4]
                | g[1] & p[2] & p[3] & p[4]
                | g[2] & p[3] & p[4]
                | g[3] & p[4]
                | g[4];

   assign       carry[6] = g[0] & p[1] & p[2] & p[3] & p[4] & p[5]
                | g[1] & p[2] & p[3] & p[4] & p[5]
                | g[2] & p[3] & p[4] & p[5]
                | g[3] & p[4] & p[5]
                | g[4] & p[5]
                | g[5];

   assign       carry[7] = g[0] & p[1] & p[2] & p[3] & p[4] & p[5] & p[6]
                | g[1] & p[2] & p[3] & p[4] & p[5] & p[6]
                | g[2] & p[3] & p[4] & p[5] & p[6]
                | g[3] & p[4] & p[5] & p[6]
                | g[4] & p[5] & p[6]
                | g[5] & p[6]
                | g[6];

   assign       carry[8] = g[0] & p[1] & p[2] & p[3] & p[4] & p[5] & p[6] & p[7]
                | g[1] & p[2] & p[3] & p[4] & p[5] & p[6] & p[7]
                | g[2] & p[3] & p[4] & p[5] & p[6] & p[7]
                | g[3] & p[4] & p[5] & p[6] & p[7]
                | g[4] & p[5] & p[6] & p[7]
                | g[5] & p[6] & p[7]
                | g[6] & p[7]
                | g[7];

   assign       carry[9] = g[0] & p[1] & p[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8]
                | g[1] & p[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8]
```

```
                | g[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8]
                | g[3] & p[4] & p[5] & p[6] & p[7] & p[8]
                | g[4] & p[5] & p[6] & p[7] & p[8]
                | g[5] & p[6] & p[7] & p[8]
                | g[6] & p[7] & p[8]
                | g[7] & p[8]
                | g[8];

   assign       carry[10] = g[0] & p[1] & p[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8] &
                | g[1] & p[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8] & p[9]
                | g[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8] & p[9]
                | g[3] & p[4] & p[5] & p[6] & p[7] & p[8] & p[9]
                | g[4] & p[5] & p[6] & p[7] & p[8] & p[9]
                | g[5] & p[6] & p[7] & p[8] & p[9]
                | g[6] & p[7] & p[8] & p[9]
                | g[7] & p[8] & p[9]
                | g[8] & p[9]
                | g[9];

   assign       carry[11] = g[0] & p[1] & p[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8] &
                | g[1] & p[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8] & p[9] & p[10]
                | g[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8] & p[9] & p[10]
                | g[3] & p[4] & p[5] & p[6] & p[7] & p[8] & p[9] & p[10]
                | g[4] & p[5] & p[6] & p[7] & p[8] & p[9] & p[10]
                | g[5] & p[6] & p[7] & p[8] & p[9] & p[10]
                | g[6] & p[7] & p[8] & p[9] & p[10]
                | g[7] & p[8] & p[9] & p[10]
                | g[8] & p[9] & p[10]
                | g[9] & p[10]
                | g[10];

   assign       sum[12] = g[0] & p[1] & p[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8] & p
                | g[1] & p[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8] & p[9] & p[10] & p
                | g[2] & p[3] & p[4] & p[5] & p[6] & p[7] & p[8] & p[9] & p[10] & p[11]
                | g[3] & p[4] & p[5] & p[6] & p[7] & p[8] & p[9] & p[10] & p[11]
                | g[4] & p[5] & p[6] & p[7] & p[8] & p[9] & p[10] & p[11]
                | g[5] & p[6] & p[7] & p[8] & p[9] & p[10] & p[11]
                | g[6] & p[7] & p[8] & p[9] & p[10] & p[11]
                | g[7] & p[8] & p[9] & p[10] & p[11]
                | g[8] & p[9] & p[10] & p[11]
                | g[9] & p[10] & p[11]
                | g[10] & p[11]
                | g[11];

   cla_slice s0(sum[0],g[0],p[0],a[0],b[0],carry[0]);
   cla_slice s1(sum[1],g[1],p[1],a[1],b[1],carry[1]);
   cla_slice s2(sum[2],g[2],p[2],a[2],b[2],carry[2]);
   cla_slice s3(sum[3],g[3],p[3],a[3],b[3],carry[3]);
   cla_slice s4(sum[4],g[4],p[4],a[4],b[4],carry[4]);
   cla_slice s5(sum[5],g[5],p[5],a[5],b[5],carry[5]);
   cla_slice s6(sum[6],g[6],p[6],a[6],b[6],carry[6]);
   cla_slice s7(sum[7],g[7],p[7],a[7],b[7],carry[7]);
   cla_slice s8(sum[8],g[8],p[8],a[8],b[8],carry[8]);
   cla_slice s9(sum[9],g[9],p[9],a[9],b[9],carry[9]);
   cla_slice s10(sum[10],g[10],p[10],a[10],b[10],carry[10]);
   cla_slice s11(sum[11],g[11],p[11],a[11],b[11],carry[11]);

endmodule


// :Example:
//
// A BFA, for use in 12-bit ripple adder.

module bfa_implicit(sum,cout,a,b,cin);
   input a,b,cin;
   output sum,cout;
```

```
   assign sum =
          ~a & ~b &  cin |
          ~a &  b & ~cin |
           a & ~b & ~cin |
           a &  b &  cin;

   assign cout = a & b | b & cin | a & cin;

endmodule


// :Example:
//
// Twelve-bit ripple adder, for comparison.

module ripple_12(sum,a,b);
   input [11:0] a, b;
   output [12:0] sum;

   wire [11:0]  carry;

   bfa_implicit bfa0(sum[0],carry[0],a[0],b[0],1'b0);
   bfa_implicit bfa1(sum[1],carry[1],a[1],b[1],carry[0]);
   bfa_implicit bfa2(sum[2],carry[2],a[2],b[2],carry[1]);
   bfa_implicit bfa3(sum[3],carry[3],a[3],b[3],carry[2]);
   bfa_implicit bfa4(sum[4],carry[4],a[4],b[4],carry[3]);
   bfa_implicit bfa5(sum[5],carry[5],a[5],b[5],carry[4]);
   bfa_implicit bfa6(sum[6],carry[6],a[6],b[6],carry[5]);
   bfa_implicit bfa7(sum[7],carry[7],a[7],b[7],carry[6]);
   bfa_implicit bfa8(sum[8],carry[8],a[8],b[8],carry[7]);
   bfa_implicit bfa9(sum[9],carry[9],a[9],b[9],carry[8]);
   bfa_implicit bfa10(sum[10],carry[10],a[10],b[10],carry[9]);
   bfa_implicit bfa11(sum[11],sum[12],a[11],b[11],carry[10]);

endmodule


// :Example:
//
// Testbench.  Runs all three adders.  Material for testbench not covered.

module testcla2();
   wire [12:0] sum_r, sum_2, sum_f;
   reg [12:0] shadow_sum;
   reg [11:0]  a, b;
   integer    i;

   cla_12_flat c1(sum_f,a,b);
   cla_12_two_level c2(sum_2,a,b);
   ripple_12 c3(sum_r,a,b);


   initial begin

      for(i=0; i<4194304; i=i+1) begin

         a = i[11:0];
         b = i[21:12];
         shadow_sum = a + b;
         #1;
         if( sum_2 != shadow_sum || sum_2 != sum_r || sum_2 != sum_f ) begin
            $display("Wrong sum: %h + %h = %h != %h or %h or %h\n",
                     a, b, shadow_sum, sum_2, sum_f, sum_r);
            $stop;
         end
      end

      $display("Tests completed.");
```

```
    end

endmodule


////////////////////////////////////////////////////////////////////////////////
/// Real-World Integer Adders

// Under construction.
```