More Verilog Constructs

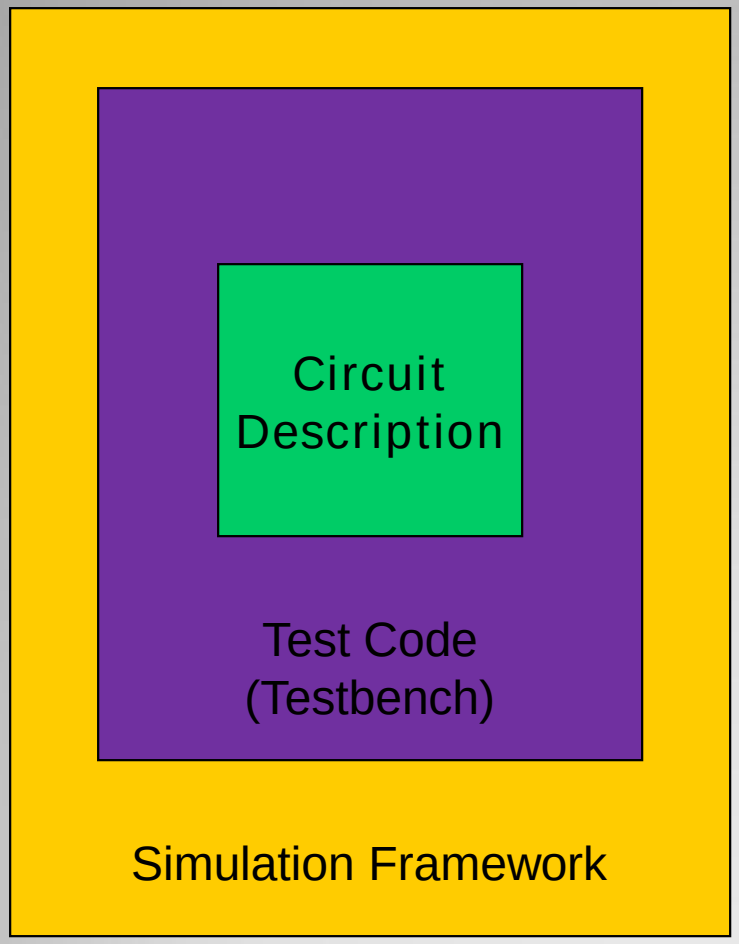
ModelSim
Parameters
Procedures

ModelSim

ModelSim

- We will use ModelSim to simulate (test) some of our circuit designs.
- We will generate test signal waveforms similar to those used in the QSim example, except we'll generate the waveforms programmatically rather than graphically.
- The programming language we'll use in ModelSim is Verilog.
- We'll need to learn just a handful of new Verilog constructs.

Elements of a Simulation



Circuit Description defines
- •Circuit inputs
- •Circuit outputs
- •Circuit Elements (gates)
- •Connections between gates

Testbench defines
- •Test signals input to circuit
- •What to do with circuit outputs

Simulation Framework
- •Provides a place for the simulation to ‘live’
- •Support services such as I/O
- •Provides support elements such as basic gate types

```
`timescale 1ns/1ns  // defines unit of time
module test_light;  // note: no I/O ports
  reg [1:0] Input;
  wire GreenLED;

  light dut( Input[0], Input[1], GreenLED ); // module
  initial    // Test stimulus
    begin
      Input = 2'h0;
      #20 Input = 2'h1;
      #20 Input = 2'h3;
      #20 Input = 2'h2;
  end

  initial
    $monitor( $stime,, Input,, GreenLED );

endmodule
```

Testbench for ‘light’ module

New Verilog Elements

- ‘timescale defines what one unit of time is.
- ‘initial’ is like ‘always’ except it runs only once at the beginning of the simulation – so it doesn’t require a sensitivity list to tell it when to run.
- #20 means “wait for 20 clock ticks before you do anything else in this procedure.” Of course 20 could be any positive integer.
- $monitor(args) prints out a line of text whenever any of its arguments changes.
- $stime stands for ‘simulation time’.

Refer to Sutherland

The ‘timescale Directive

‘timescale time_unit base / precision base
Specifies the time units and precision for delays:

- time_unit is the amount of time a delay of 1 represents. The time unit must be 1, 10, or 100
- base is the time base for each unit, ranging from seconds to femtoseconds, and must be:
  s ms us ns ps or fs
- precision and base represent how many decimal points of precision to use relative to the time units.

Example: `timescale 1 ns / 10 ps
Indicates delays are in 1 nanosecond units with
2 decimal points of precision (10 ps is .01 ns).

Note: There is no default timescale in Verilog; delays are simply relative numbers until a timescale directive declares the units and base the numbers represent.

From Sutherland’s Verilog Reference Guide

Moving ON

Consider:

```
module NANDn( I, W );
  parameter n = 8;
  input [n-1:0] I;
  output W = ~&I;
endmodule
```

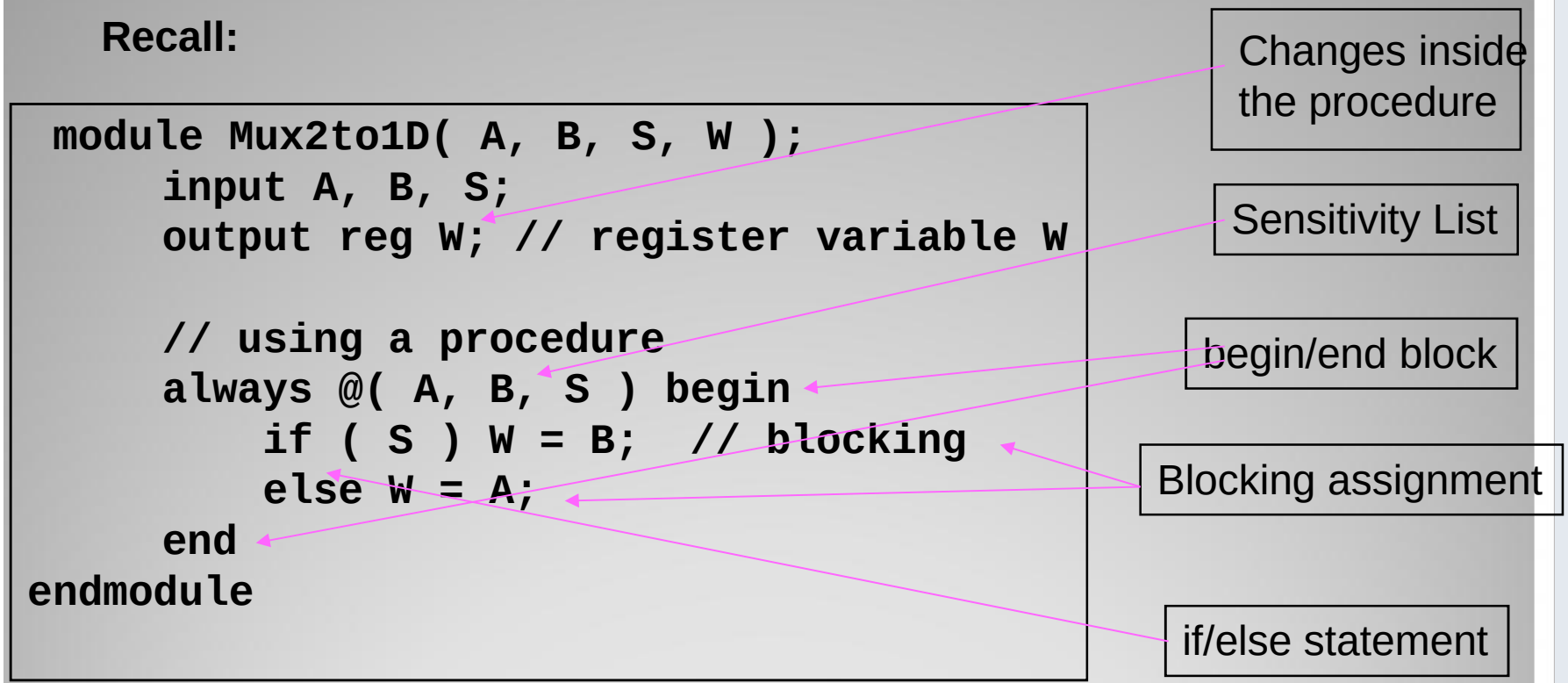
```
module NANDnTest( Z );
  output Z;

 wire [15:0] I = 16'd255;
 NANDn #(.n(16))  U1( I, Z );

endmodule
```

NANDn

Procedure Example

Recall:

```
module Mux2to1D( A, B, S, W );
    input A, B, S;
    output reg W; // register variable W

    // using a procedure
    always @( A, B, S ) begin
        if ( S ) W = B;  // blocking
        else W = A;
    end
endmodule
```



Verilog Data Types

- Net data types
  - wire : is the type we'll mostly use
  - there are about 10 more mostly dealing with 'wired' ANDs and ORs
- Variable data types
  - Used for programming storage in procedural blocks
  - reg : a variable of any bit size; unsigned unless explicitly declared as signed
  - integer : a signed 32-bit variable (we'll use these in 'for' loops)
  - genvar : used in generate blocks (otherwise, like 'integer')
  - time, real, realtime : don't synthesize

Verilog Data Types, cont.

- Other data types
  - parameter
    - we used a parameter in NANDn
  - localparam
    - like parameter, but can't be directly redefined from outside the module
    - we'll use these in state machine to give names to states.
- Refer to the Quick Reference Guide

Procedure Blocks

- initial blocks
  - Process statements one time only
  - Used in testbench design
- always blocks
  - An infinite loop
  - Use sensitivity list to determine when to execute the block
  - Use begin/end to delineate

Sensitivity Lists

- `always @( A, B, C )`
  - implies combinational logic
  - block runs whenever A or B or C changes
  - also written (A or B or C)
- `always @*`
  - implies combinational logic
  - block runs whenever any variable read within the block changes
  - new in Verilog 2001
- `always @( posedge ClkA, negedge ClkB )`
  - implies sequential logic
  - block runs if ClkA sees a positive edge or if ClkB sees a negative edge
  - a specific edge should be specified for each signal in the list
- Don’t mix combinational and sequential sensitivity lists

Procedural Assignment Statements

- `A = B;`
  - Blocking assignment
  - Expression is evaluated and assigned when the statement is encountered.
  - In `begin M=N; N=M; end` the first assignment changes M before the second one reads it.
- `A <= B;`
  - Non-blocking assignment
  - Expression is evaluated when the statement is encountered and assignment is postponed until the end of the begin/end block.
  - In `begin M<=N; N<=M; end` both assignments will be evaluated before M or N changes (swaps values).

Blocking/Non-blocking Uses

- Use blocking assignments (=) to model combinational logic.
- Use non-blocking assignments (<=) to model sequential logic.
- This helps avoid race conditions that can develop.

Procedural Programming Statements

if-statement

```
if ( expression) begin
  // statements
end
```

if/else-statement

```
if ( expression) begin
  // statements
else begin
 // statements
end
```

case-statement

```
case ( expression)
  case_item: begin … end
  case_item, case_item: begin … end
  default: begin … end
endcase
```

also: `casez` and `casex` which we'll talk about later

for-loop : be very careful

```
for (initial_assignment; expression; step_assignment) begin
  // statements
end
```

Procedural Programming Statements, cont.

- May or may not synthesize:
  - `while (expression)`
  - `repeat (number)`
  - `forever`
  - `disable group_name`

From Lab 1, Part II:

```
module Mux8( S, X, Y, M );
  input S;            // mux select line in
  input  [7:0] X, Y; // mux inputs
  output [7:0] M;    // mux output

  // the mux:
  assign M[0] = (~S & X[0]) | (S & Y[0]);
  assign M[1] = (~S & X[1]) | (S & Y[1]);
  assign M[2] = (~S & X[2]) | (S & Y[2]);
  assign M[3] = (~S & X[3]) | (S & Y[3]);
  assign M[4] = (~S & X[4]) | (S & Y[4]);
  assign M[5] = (~S & X[5]) | (S & Y[5]);
  assign M[6] = (~S & X[6]) | (S & Y[6]);
  assign M[7] = (~S & X[7]) | (S & Y[7]);

endmodule
```

Mux8 Example

```
module Mux8( S, X, Y, M );
  input S;              // mux select line in
  input  [7:0] X, Y;    // mux inputs
  output reg [7:0] M; // mux output

  integer i;
  always @ ( S, X, Y ) begin
    for ( i=0; i<8; i=i+1 ) begin
      M[i] = (~S & X[i]) | (S & Y[i]);
    end  // for-loop
  end // always

endmodule
```

Mux8, for-loop

```
module MuxN( S, X, Y, M );
  parameter N = 8;
  input S;                 // mux select line in
  input  [N-1:0] X, Y;  // mux inputs
  output reg [N-1:0] M; // mux output

  integer i;
  always @ ( S, X, Y ) begin
    for (i=0; i<N; i=i+1) begin
      M[i] = (~S & X[i]) | (S & Y[i]);
    end  // for-loop
  end // always

endmodule
```

MuxN: More Versatile

```
module MuxNTest( S, X, Y, M );
  localparam Width = 32;
  input S;                      // mux select line in
  input  [Width-1:0] X, Y;  // mux inputs
  output [Width-1:0] M; // mux output

  MuxN #(.N(Width)) U1( S, X, Y, M );

endmodule
```

MuxNTest : Sets N

Three Ones Circuit

We want to build a circuit that has eight inputs (x_7, x_6, … x_0) and produces an output of F = 1 if there are at least three consecutive ones in the input; otherwise output F = 0. So if X = 00111100 the output would be F = 1, but if X = 10101010 the output would be F = 0.

The truth table for F would have 256 entries – too long for us to deal with easily.

On the other hand, we can see that

$F = x_0x_1x_2 + x_1x_2x_3 + x_2x_3x_4 + x_3x_4x_5 + x_4x_5x_6 + x_5x_6x_7$

Write a Verilog module that computes F:

```
module ThreeOnes( X, F );
  input [7:0] X;
  output F;
…
endmodule
```

Three Ones Circuit, cont.

Now what if I said that X has 32 elements?
It wouldn't be very much fun writing out F, would it?

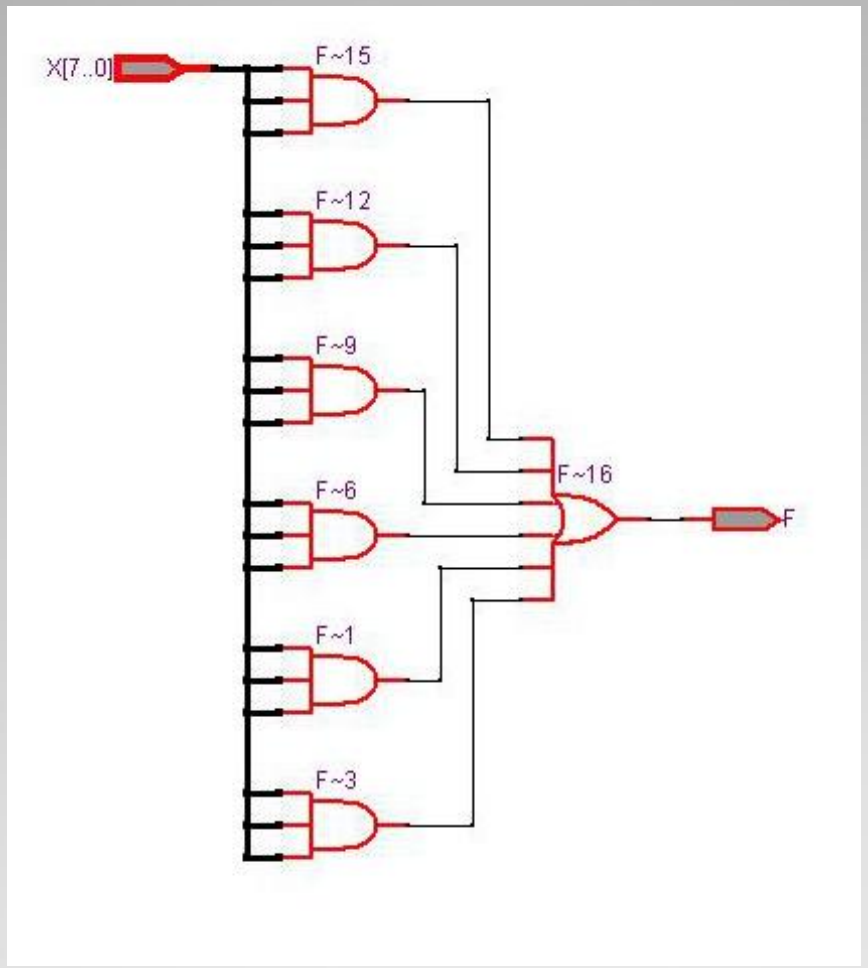
But we can use a for-loop and an if-statement to implement F.

- These statements must be in a procedure
- F must be declared as a reg (since it will be computed in a procedure
- We might as well make the size of X a parameter (N)
- The procedure is going to look something like this:

```
integer I;
always @ ( ) begin
  F = 0;    // default value
  for ( I=0; …; I=I+1 ) begin
    if ( X[I] & X[I+1] & X[I+2] ) F = 1;
  end // for-loop
end // always
```

```
module ThreeOnes( X, F );
  parameter N = 8;
  input [N-1:0] X;
  output reg F;

  integer I;

  always @* begin
    F = 0;
    for ( I=0; I<N-2; I=I+1 )
      F = F | X[I]&X[I+1]&X[I+2];
  end

  /*always @ ( X ) begin
    F = 0;
    for ( I=0; I<N-2; I=I+1 )
     if ( X[I]&X[I+1]&X[I+2] ) F = 1;
  end*/

endmodule
```

ThreeOnes.v



Three Ones Circuit (RTL)

- Test Add5To module
- Test ThreeOnes module

Using for-loop in a Testbench

Count Ones

```
module CountOnes( X, F );
  input [15:0] X;
  output reg [3:0] F;

  integer I;

  always @* begin
    F = 0;
    for ( I=0; I<16; I=I+1 )
      if ( X[I] == 1'b1 )
        F = F + 1'b1;
   end

endmodule
```

- for-loop example
- generate-loop example
- Arithmetic statements

Chapter 5 Verilog Examples

```
module adder4( carryin, x3, x2, x1, x0, y3, y2, y1, y0,
               s3, s2, s1, s0, carryout );
       input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
       output s3, s2, s1, s0, carryout;
       wire c3,c2,c1;

       fulladd stage0 ( carryin, x0, y0, s0, c1 );
       fulladd stage1 ( c1, x1, y1, s1, c2 );
       fulladd stage2 ( c2, x2, y2, s2, c3 );
       fulladd stage3 ( c3, x3, y3, s3, carryout );

endmodule

module fulladd ( Cin, x, y, s, Cout );
       input Cin, x, y;
       output s, Cout;
       assign s = x ^ y ^ Cin,
       assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

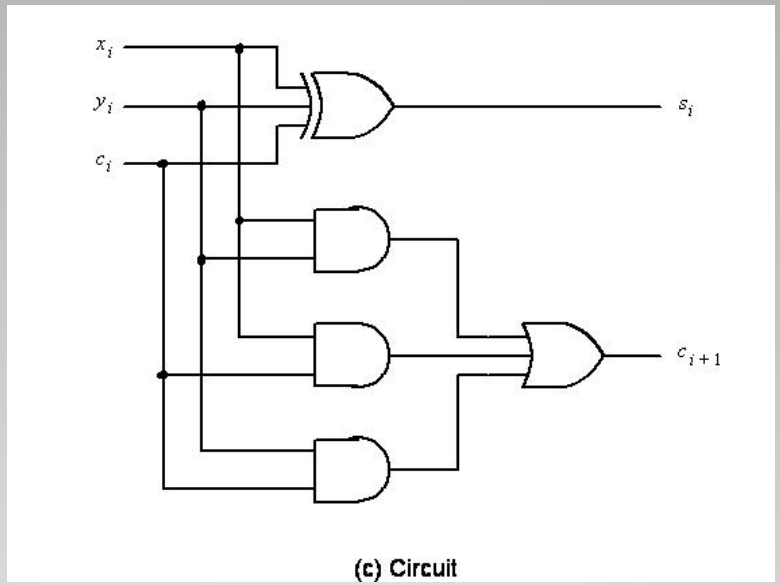
Recall: Ripple-Carry Adder

```
module adder4( carryin, x3, x2, x1, x0, y3, y2, y1, y0,
               s3, s2, s1, s0, carryout );
  input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
  output s3, s2, s1, s0, carryout;
  wire c4,c3,c2,c1,c0;
  assign c0 = carryin;
  assign carryout = c4;

  //fulladd stage0 ( c0, x0, y0, s0, c1 );
  //fulladd stage1 ( c1, x1, y1, s1, c2 );
  //fulladd stage2 ( c2, x2, y2, s2, c3 );
  //fulladd stage3 ( c3, x3, y3, s3, c4 );

  // something like:
    for (i=0; i<4; i++ ) begin
      fulladd stagei( ci, xi, yi, si, ci+1 )
    end
endmodule
```

We can't instantiate a module within a for-loop

What We Would Like to Do

Behavioral Version of Full Adder



(c) Circuit

```
module FullAdd( Cin, X, Y, S, Cout );
  input Cin, X, Y;
  output S, Cout;

  assign S = X ^ Y ^ Cin;
  assign Cout = (X & Y) | (X & Cin) | (Y & Cin);

endmodule
```

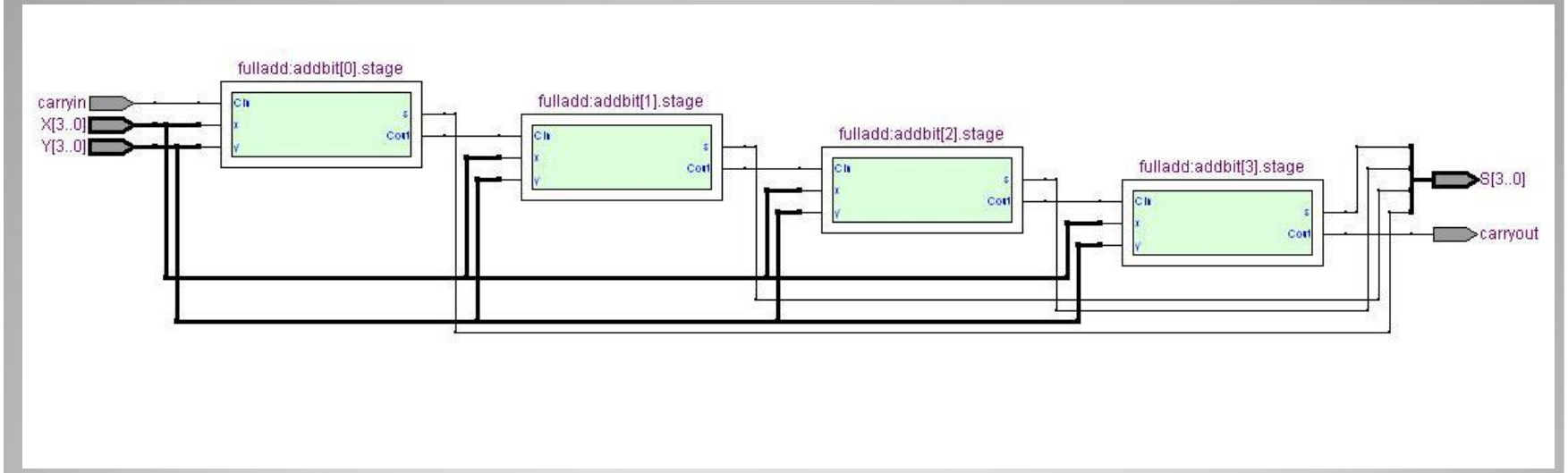
We can do this:

```
module addern (carryin, X, Y, S, carryout);
  parameter n=32;
  input carryin;
  input [n-1:0] X, Y;
  output reg [n-1:0] S; // value assigned in always block
  output reg carryout;  // value assigned in always block
  reg [n:0] C;          // notice the size of C
  integer k;            // used as for-loop index

  always @(X, Y, carryin) begin
    C[0] = carryin;  // note use of blocking assignments
    for (k = 0; k < n; k = k+1) begin
      S[k] = X[k] ^ Y[k] ^ C[k];  // we unwrap 'fulladd'
      C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
    end
      carryout = C[n];
  end  // always

endmodule
```

addern (Figure 5.28)

```
module ripple_g( carryin, X, Y, S, carryout );
  parameter n = 4;
  input carryin;
  input [n-1:0] X, Y;
  output [n-1:0] S;
  output carryout;
  wire [n:0] C;

  genvar i; // new
  assign C[0] = carryin;
  assign carryout = C[n];
  generate // kind of like a procedure block
    for ( i = 0; i <= n; i = i+1 )
    begin:addbit // notice we name the block
      fulladd stage( C[i], X[i], Y[i], S[i], C[i+1] );
    end
  endgenerate
endmodule
```

addern Using ‘generate’



What Quartus II Synthesized