Chapter 7: Hierarchical Structural Modeling

Prof. Ming-Bo Lin

Department of Electronic Engineering

National Taiwan University of Science and Technology

Syllabus

- Objectives
- Module
  - Module definitions
  - Parameters
  - Module instantiation
  - Module parameter values
  - Hierarchical path names
- Generate statements
  - Generate-loop statement
  - Generate-conditional statement
  - Generate-case statement

Module Definitions

```
// port list style
module module_name [#(parameter_declarations)][port_list];
parameter_declarations; // if no parameter ports are used
port_declarations;
other_declaration;
statements;
endmodule

// port list declaration style
module module_name [#(parameter_declarations)][port_declarations];
parameter_declarations; // if no parameter ports are used
other_declarations;
statements;
endmodule
```

Port Declarations

- Three types
  - input
  - output
  - inout


```
module adder(x, y, c_in, sum, c_out);
input [3:0] x, y;
input c_in;
output reg [3:0] sum;
output reg c_out;
```

Syllabus

- Objectives
- Module
  - Module definitions
  - Parameters
  - Module instantiation
  - Module parameter values
  - Hierarchical path names
- Generate statements

Types of Parameters

- module parameters
  - parameter
  - localparam
- specify parameters

```
parameter SIZE = 7;
parameter WIDTH_BUSA = 24, WIDTH_BUSB = 8;
parameter signed [3:0] mux_selector = 4'b0;
```

Constants Specified Options

- `define compiler directive

  ```
  `define BUS_WIDTH  8
  ```

- Parameter

  ```
  parameter BUS_WIDTH = 8;
  ```

- localparam

  ```
  localparam BUS_WIDTH = 8;
  ```

Syllabus

- Objectives
- Module
  - Module definitions
  - Parameters
  - Module instantiation
  - Module parameter values
  - Hierarchical path names
- Generate statements

Module Instantiation

- Syntax

```
module_name [#(parameters)]
                    instance_name [range]([ports]);
module_name [#(parameters)]
                    instance_name [{,instance_name}]([ports]);
```

Port Connection Rules

- Named association

  .port_id1(port_expr1),..., .port_idn(port_exprn)

- Positional association

  port_expr1, ..., port_exprn

Syllabus

- Objectives
- Module
  - Module definitions
  - Parameters
  - Module instantiation
  - Module parameter values
  - Hierarchical path names
- Generate statements

Parameterized Modules

- An example

```
module adder_nbit(x, y, c_in, sum, c_out);
parameter N = 4;   // set default value
input [N-1:0] x, y;
input c_in;
output [N-1:0] sum;
output c_out;

assign {c_out, sum} = x + y + c_in;
endmodule
```

Overriding Parameters

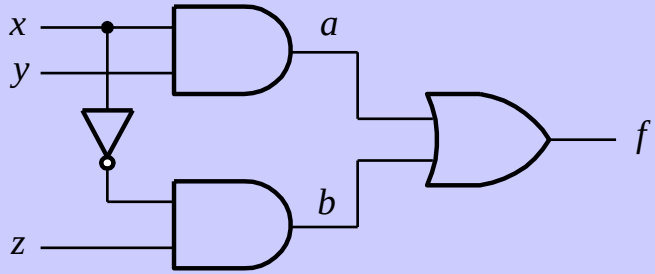
- Using module instance parameter value assignment
  --- two parameters

```
// define top level module
module …
…
hazard_static #(4, 8) example (x, y, z, f);
```

```
module hazard_static (x, y, z, f);
parameter delay1 = 2, delay2 = 5;
…
   and  #delay2 a1 (b, x, y);
   not  #delay1  n1 (a, x);
   and  #delay2 a2 (c, a, z);
   or   #delay2 o2 (f, b, c);
endmodule
```

```
hazard_static #(.delay2(4), .delay1(6))
example (x, y, z, f);
```

- parameter value assignment by name --- minimize the chance of error!



Syllabus

- Objectives
- Module
- Generate statements
  - Generate-loop statements
  - Generate-conditional statements
  - Generate-case statements

generate Block Structures

- The keywords used
  - generate and endgenerate

```
// convert Gray code into binary code
parameter SIZE = 8;
input  [SIZE-1:0] gray;
output [SIZE-1:0] bin;
genvar i;
generate for (i = 0; i < SIZE; i = i + 1) begin: bit
      assign bin[i] = ^gray[SIZE-1:i];
end endgenerate
```

```
parameter SIZE = 8;
input  [SIZE-1:0] gray;
output [SIZE-1:0] bin;
begin: bit
    assign bin[0] = ^gray[7:0];
    assign bin[1] = ^gray[7:1];
    assign bin[2] = ^gray[7:2];
    assign bin[3] = ^gray[7:3];
    assign bin[4] = ^gray[7:4];
    assign bin[5] = ^gray[7:5];
    assign bin[6] = ^gray[7:6];
    assign bin[7] = ^gray[7:7];
end
```

The generate Loop Construct

```
// convert Gray code into binary code
parameter SIZE = 8;
input  [SIZE-1:0] gray;
output [SIZE-1:0] bin;
reg    [SIZE-1:0] bin;

genvar i;
generate for (i = 0; i < SIZE; i = i + 1) begin:bit
    always @(*)
        bin[i] = ^gray[SIZE - 1: i];
end endgenerate
```

```
// convert Gray code into binary code
parameter SIZE = 8;
input  [SIZE-1:0] gray;
output [SIZE-1:0] bin;
reg    [SIZE-1:0] bin;
begin: bit
    always @(*)  bin[0] = ^gray[7: 0];
    always @(*)  bin[1] = ^gray[7: 1];
    always @(*)  bin[2] = ^gray[7: 2];
    always @(*)  bin[3] = ^gray[7: 3];
    always @(*)  bin[4] = ^gray[7: 4];
          :        :        :
    always @(*)  bin[7] = ^gray[7: 7];
end endgenerate
```

Syllabus

- Objectives
- Module
- Generate statements
  - Generate-loop statements
  - Generate-conditional statements
  - Generate-case statements

An *n*-bit Adder

```
// define a full adder at dataflow level.
module full_adder(x, y, c_in, sum, c_out);
// I/O port declarations
input  x, y, c_in;
output sum, c_out;
// Specify the function of a full adder.
   assign {c_out, sum} = x + y + c_in;
endmodule
```

An *n*-bit Adder

```
module adder_nbit(x, y, c_in, sum, c_out);
…
genvar  i;
wire  [N-2:0] c;      // internal carries declared as nets.
generate for (i = 0; i < N; i = i + 1) begin: adder
   if (i == 0)          // specify LSB
      full_adder fa (x[i], y[i], c_in, sum[i], c[i]);
   else if (i == N-1) // specify MSB
      full_adder fa (x[i], y[i], c[i-1], sum[i], c_out);
   else                  // specify other bits
      full_adder fa (x[i], y[i], c[i-1], sum[i], c[i]);
end endgenerate
```

An *n*-bit Adder

```
module adder_nbit(x, y, c_in, sum, c_out);
…
genvar  i;
wire [N-2:0] c;       // internal carries declared as nets.
generate for (i = 0; i < N; i = i + 1) begin: adder
   if (i == 0)            // specify LSB
      assign {c[i], sum[i]} =  x[i] + y[i] + c_in;
   else if (i == N-1) // specify MSB
      assign {c_out, sum[i]} =  x[i] + y[i] + c[i-1];
   else                  // specify other bits
      assign {c[i], sum[i]} =  x[i] + y[i] + c[i-1];
end endgenerate
```

An *n*-bit Adder

```
module adder_nbit(x, y, c_in, sum, c_out);
…
genvar i;
reg    [N-2:0] c;       // internal carries declared as nets.
generate for (i = 0; i < N; i = i + 1) begin: adder
  if  (i == 0)          // specify LSB
    always @(*) {c[i], sum[i]} =  x[i] + y[i] + c_in;
  else if (i == N-1)  // specify MSB
    always @(*) {c_out, sum[i]} =  x[i] + y[i] + c[i-1];
  else                  // specify other bits
    always @(*) {c[i], sum[i]} =  x[i] + y[i] + c[i-1];
end endgenerate
```

A Two's Complement Adder

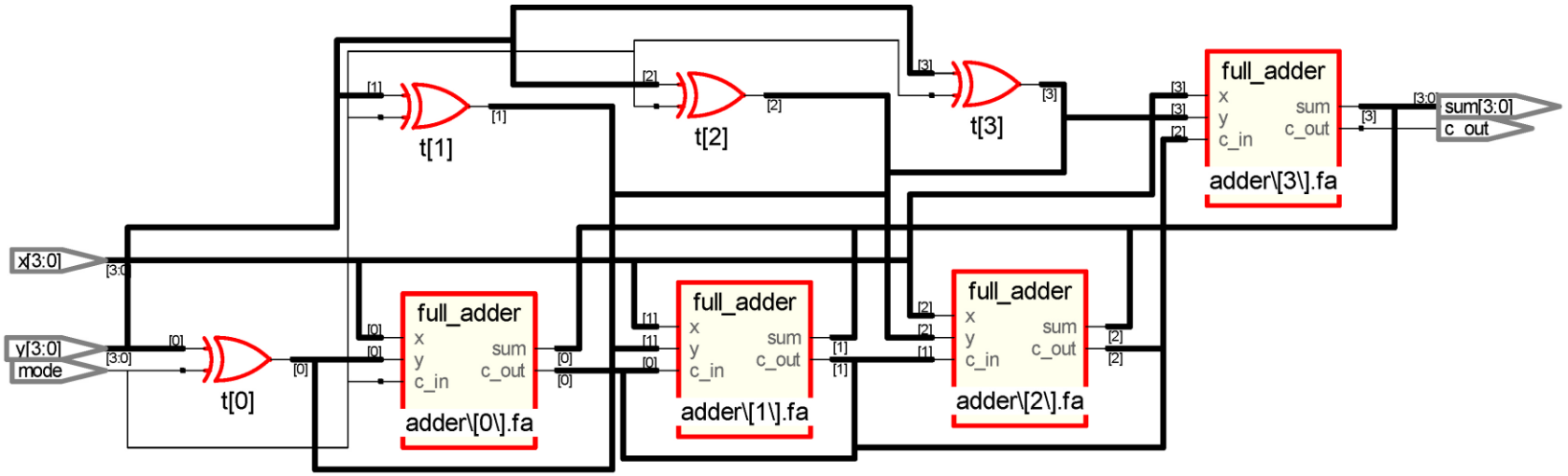
```
module twos_adder_nbit(x, y, mode, sum, c_out);
…
genvar i;
wire [N-2:0] c;   // internal carries declared as nets.
wire [N-1:0] t;    // true/ones complement outputs
generate for (i = 0; i < N; i = i + 1) begin:
// ones_complement_generator
   xor xor_ones_complement (t[i], y[i], mode);
end endgenerate
```

A Two's Complement Adder

```
generate for (i = 0; i < N; i = i + 1) begin: adder
  if (i == 0)           // specify LSB
    full_adder fa (x[i], t[i], mode, sum[i], c[i]);
  else if (i == N-1) // specify MSB
    full_adder fa (x[i], t[i], c[i-1], sum[i], c_out);
  else                  // specify other bits
    full_adder fa (x[i], t[i], c[i-1], sum[i], c[i]);
end endgenerate
```

A Two's Complement Adder

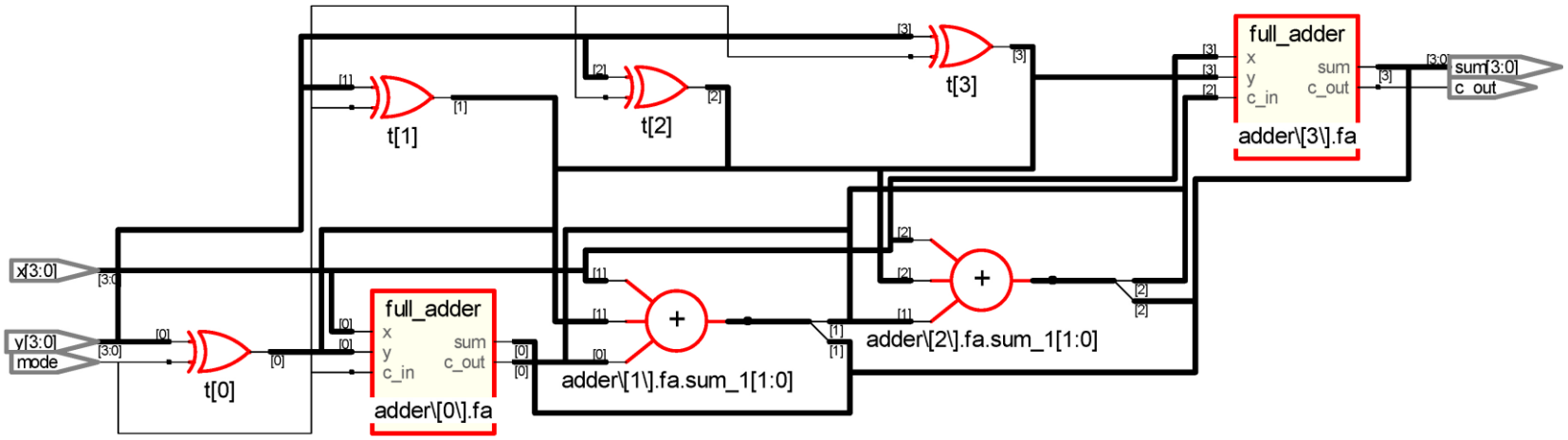
- The RTL schematic from Synplify Pro.



(a) The RTL schematic

A Two's Complement Adder

- After dissolving the second and the third bits



(b) After dissolving the second and the third full adder modules

Syllabus

- Objectives
- Module
- Generate statements
  - Generate-loop statements
  - Generate-conditional statements
  - Generate-case statements

The generate Case Construct

```
generate for (i = 0; i < N; i = i + 1) begin: adder
  case (i)
          0: assign {c[i], sum[i]} = x[i] + y[i] + c_in;
      N-1: assign {c_out, sum[i]} = x[i] + y[i] + c[i-1];
    default: assign {c[i], sum[i]} = x[i] + y[i] + c[i-1];
  endcase
end endgenerate
```

A UDP Example

```
// an example of sequential UDP instantiations
parameter N = 4;
input  clk, clear;
output [N-1:0] qout;
….
genvar  i;
generate for (i = 0; i < N; i = i + 1) begin: ripple_counter
   if (i == 0) // specify LSB
      T_FF tff (qout[i], clk, clear);
   else          // specify the rest bits
      T_FF tff (qout[i], qout[i-1], clear);
end endgenerate
```