HARDWARE DESCRIPTION LANGUAGE FOR DIGITAL DESIGN

數位設計硬體描述語言

Hierarchical Structure Modeling

Materials partly adapted from "Digital System Designs and Practices Using Verilog HDL and FPGAs," M.B. Lin.

NCKU EE
LPHP Lab

OUTLINE

- Verilog Basics
- Structural Modeling
- Dataflow Modeling
- Behavioral Modeling
- Tasks and Functions
- Hierarchical Structure Modeling

HIERARCHICAL STRUCTURE MODELING

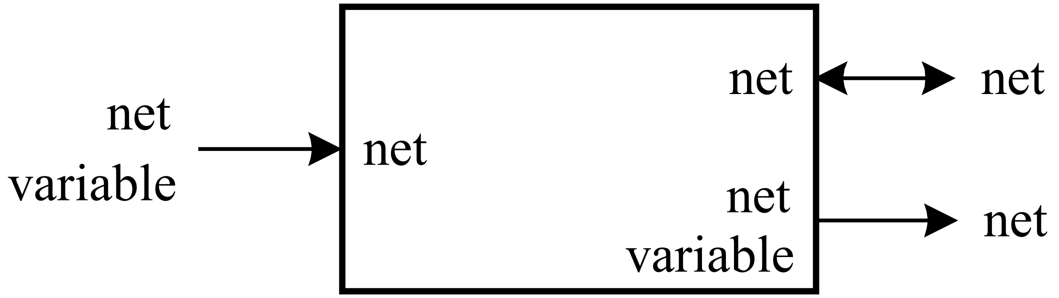
MODULE DEFINITIONS

```
// 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;
```

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;
  ```

PARAMETER PORTS

```
module module_name
#(parameter SIZE = 7,
  parameter WIDTH_BUSA = 24, WIDTH_BUSB = 8,
  parameter signed [3:0] mux_selector = 4’b0
)
(port list or port list declarations)
...
endmodule
```

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

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
```

MODULE PARAMETERS VALUES

- Ways to change module parameters values
  - defparam statement
  - module instance parameter value assignment

OVERRIDING PARAMETERS

- Using the defparam statement

```
module counter_nbits (clock, clear, qout);
parameter N = 4;  // define counter size
…
always @(negedge clock or posedge clear)
begin          // qout <= (qout + 1) % 2^n
     if (clear) qout <= {N{1'b0}};
     else          qout <= (qout + 1) ;
End
```

```
// define top level module
…
output [3:0] qout4b;
output [7:0] qout8b;
// instantiate two counter modules
defparam cnt_4b.N = 4, cnt_8b.N = 8;
counter_nbits cnt_4b (clock, clear, qout4b);
counter_nbits cnt_8b (clock, clear, qout8b);
```

OVERRIDING PARAMETERS

- Using module instance parameter value assignment--- one parameter

```
module counter_nbits (clock, clear, qout);
parameter N = 4;  // define counter size
…
always @(negedge clock or posedge clear)
begin                // qout <= (qout + 1) % 2^n;
    if (clear) qout <= {N{1'b0}};
    else       qout <= (qout + 1) ;
end
```

```
// define top level module
…
output [3:0] qout4b;
output [7:0] qout8b;
// instantiate two counter modules
counter_nbits #(4) cnt_4b (clock, clear, qout4b);
counter_nbits #(8) cnt_8b (clock, clear, qout8b);
```

OVERRIDING PARAMETERS

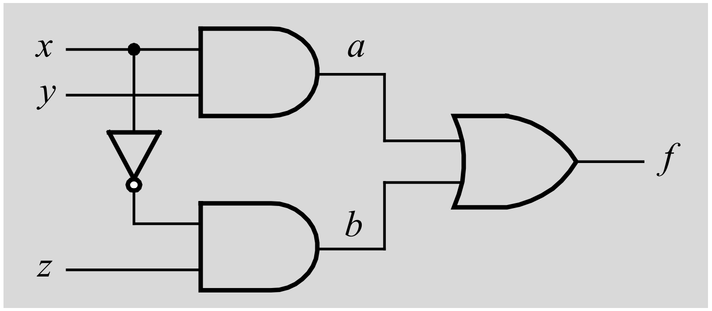
- Using module instance parameter value assignment
  --- two parameters

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

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

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

```
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
```



HIERARCHICAL PATH NAMES

- An identifier can be defined within
  - Modules
  - Tasks
  - Functions
  - Named blocks (See Section 7.1.3)
- Hierarchical path names

```
4bit_adder             // top level --- 4bit_adder
4bit_adder.fa_1        // fa_1 within 4bit_adder
4bit_adder.fa_1.ha_1             // ha_1 within fa_1
4bit_adder.fa_1.ha_1.xor1     // xor1 within ha_1
4bit_adder.fa_1.ha_1.xor1.S // net s within xor1
```

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
```

NCKU EE
LPHP Lab

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
```

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
```

NCKU EE
LPHP Lab

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

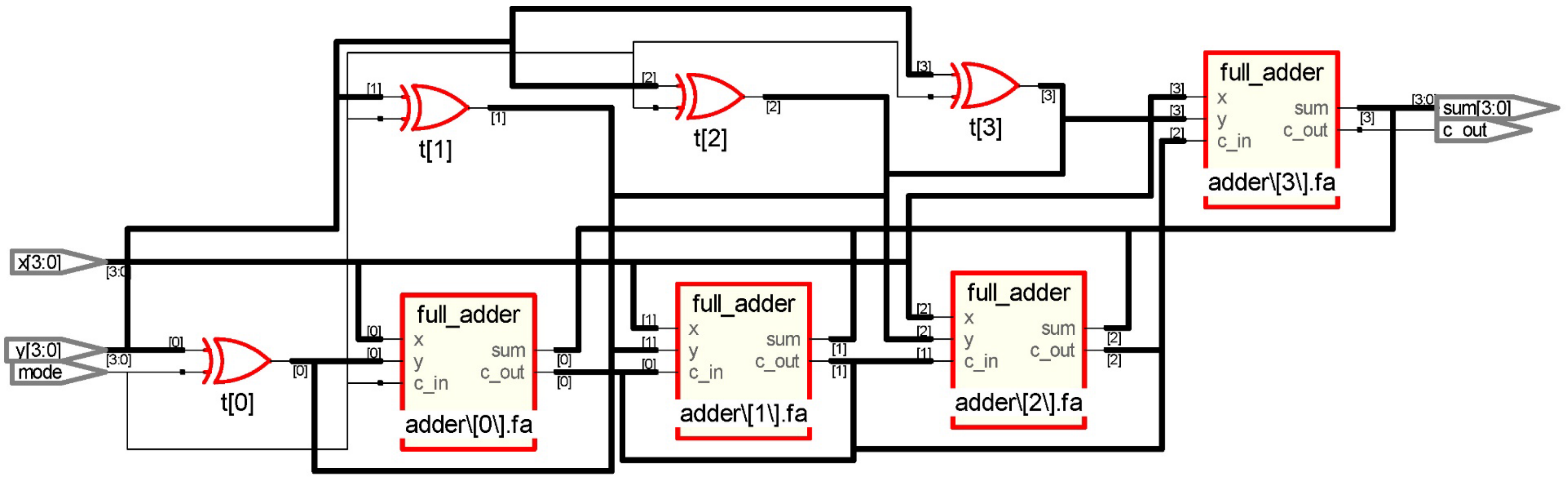
```
module twos_adder_nbit(x, y, mode, sum, c_out);
…
genvar i;
wire [N-2:0] c;   // internal carries declared as nets.
wire [N-1:0] t;    // true/ones complement outputs
generate for (i = 0; i < N; i = i + 1) begin:
// ones_complement_generator
   xor xor_ones_complement (t[i], y[i], mode);
end endgenerate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

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

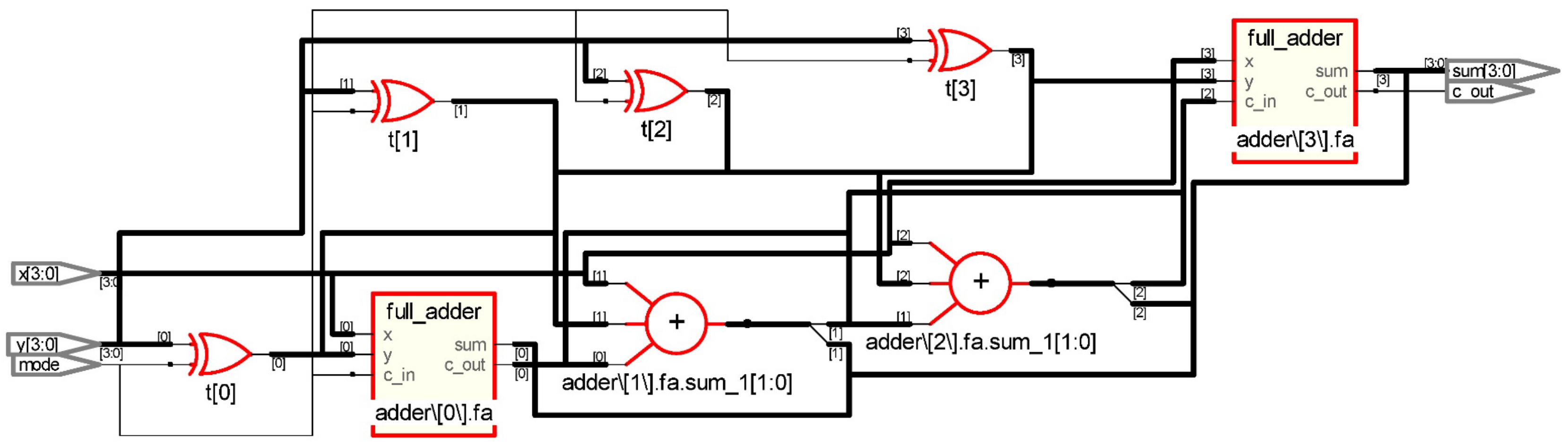
- The RTL schematic from Synplify Pro.



(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

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
```