Synthesis of Combinational Logic
HDL-Based Design Flow

DESIGN ENTRY

HDL BEHAVIORAL MODEL

Functional Simulation

Timing Simulation

DESIGN VERIFICATION

PHYSICAL OPTIMIZATION AND IMPLEMENTATION

PLD: PARTITION, MAP, INTERCONNECT

FPGA: MAP MASKED GATE ARRAYS: MAP, PLACE, ROUTE

STANDARD CELLS: PLACE, ROUTE
Simulation-Based Verification

Testbench for post-synthesis design verification.

CS 4120
Logic synthesis operates on Boolean equations and produces optimized combinational logic
Logic Minimization

- Minimize the area, delay, or power

- Methods
  - two-level logic minimization
    - e.g., Espresso
  - multi-level logic minimization
    - e.g., MIS-II
      - decomposition
      - extraction
      - factoring
      - substitution
      - elimination
Transformations

- **Decomposition**
  - example
    - before decomposition, \( F = abc + abd + a’c’d’ + b’c’d’ \)
    - express \( F \) using \( X = ab \) and \( Y = c + d \)
    - after decomposition, \( F = XY + X’Y’ \)

- **Extraction**
  - example
    - before extraction, \( F = (a + b)cd + e \), \( G = (a + b)e’ \), \( H = cde \)
    - express \( F \), \( G \) and \( H \) using \( X = a + b \) and \( Y = cd \)
    - after extraction, \( F = XY + e \), \( G = Xe’ \), \( H = Ye \)

- **Factoring**
  - example
    - before factoring, \( F = ac + ad + bc + bd + e \)
    - after factoring, \( F = (a + b)(c + d) + e \)
Transformations

- Substitution
  - example
    - before substitution, $F = a + b + c$, $G = a + b$
    - after $G$ is substituted into $F$, $F = G + c$

- Elimination
  - example
    - before elimination, $F = Ga + G'b$, $G = c + d$
    - after elimination, $F = ac + ad + bc'd'$
Commonly-Supported Constructs

Module declaration
Port modes: input, output, inout
Port binding by name
Port binding by position
Parameter declaration
Connectivity nets: wire, tri, wand, wor, supply0, supply1
Register variables: reg, integer
Integer types in binary, decimal, octal, hex formats
Scalar and vector nets
Subranges of vector nets on RHS of assignment
Module and macromodule instantiation
Primitive instantiation
Continuous assignments
Shift operator
Conditional operator
Concatenation operator (including nesting)
Arithmetic, bitwise, reduction, logical, and relational operators

Procedural-continuous assignments (assign ... deassign)
Procedural block statements (begin ... end)
case, casex, casez, default
Branching: if, if ... else, if ... else ... if
disable (of procedural block)
for loops
Tasks: task ... endtask*
Functions: function ... endfunction

*No timing or event control allowed.
Unsupported Constructs

Assignment with variable used as bit select on LHS
global variables
case equality, inequality (==, !==)
defparam
event
fork... join
forever
while
wait
initial
pulldown, pullup
force... release
repeat
cmos, rcmos, rnmos, nmos, pmos, rpmos
tran, tranif0, tranif1, rtran, rtranif0, rtranif1
primitive... endprimitive
table... endtable
intra-assignment timing control
delay specifications
scalared, vectored
small, medium, large
specify, endspecify
$time
weak0, weak1, strong0, strong1, pull0, pull1
$keyword

Verilog constructs not generally supported in synthesis.
## Combinational Logic Elements

- Commonly synthesized combinational logic

<table>
<thead>
<tr>
<th>Multiplexer</th>
<th>Adder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoder</td>
<td>Subtractor</td>
</tr>
<tr>
<td>Encoder</td>
<td>ALU</td>
</tr>
<tr>
<td>Comparator</td>
<td>Multiplier</td>
</tr>
<tr>
<td>Random Logic</td>
<td>PLA Structure</td>
</tr>
<tr>
<td>Lookup Table</td>
<td>Parity Generator</td>
</tr>
</tbody>
</table>
Synthesizable Descriptions of Combinational Logic

- Netlist of Verilog primitives
- Combinational UDP
- Continuous assignment
- Behavioral statement
- Function
- Task without delay or event control
- Interconnected modules of the above
module or_nand_1 (enable, x1, x2, x3, x4, y);
input enable, x1, x2, x3, x4;
output y;
wire w1, w2, w3;
or (w1, x1, x2);
or (w2, x3, x4);
or (w3, x3, x4); // redundant
nand (y, w1, w2, w3, enable);
endmodule
Example

Describe it with primitives

Synthesis result

(d) Circuit synthesized from the generic circuit in Figure 8.20c.
Example

Describe it with primitives

Synthesis result with shared logic
UDP

```
primitive boolean_eqs (y, a, b, c);
  output  y;
  input   a, b, c;

table
  // Inputs   | Output
  // a  b  c   | y
  0  1  ?    | 1 ;
  0  0  ?    | 0 ;
  1  ?  1    | 1 ;
  1  ?  0    | 0 ;
endtable
endprimitive
```

Synthesis result
Continuous Assignment

module or_nand_2 (enable, x1, x2, x3, x4, y);
input enable, x1, x2, x3, x4;
output y;

assign y = ~(enable & (x1 | x2) & (x3 | x4));
endmodule

Synthesis result
Cyclic Behavior

- All inputs to a behavior that implements combinational logic must be included in the event control expression, either explicitly or implicitly
  - Inputs: operands on RHS of assignments + control signals whose transitions affect assignments to target register variables
- Failure to do so will produce unwanted latches
- The target of an assignment must not be an operand on RHS of any assignment
Example

module or_nand_3 (enable, x1, x2, x3, x4, y);
input    enable, x1, x2, x3, x4;
output   y;
reg      y;

always @ (enable or x1 or x2 or x3 or x4)
begin
  y = ~(enable & (x1 || x2) & (x3 || x4));
end
endmodule

Circuit synthesized from a behavioral description
Example

module and4_behav (y, x_in);
  parameter word_length = 4;
  input [word_length - 1: 0] x_in;
  output y;
  reg y;
  integer k;

always @ x_in
  begin: check_for_0
    y = 1;
    for (k = 0; k <= word_length - 1; k = k+1)
      if (x_in[k] == 0)
        begin
          y = 0;
          disable check_for_0;
        end
    end
end
endmodule
Example

```verilog
module comparator (a, b, a_gt_b, a_lt_b, a_eq_b); // Alternative algorithm
parameter size = 2;
input [size: 1] a, b;
output a_gt_b, a_lt_b, a_eq_b;
reg a_gt_b, a_lt_b, a_eq_b;
integer k;

always @ (a or b) begin: compare_loop
  for (k = size; k > 0; k = k-1) begin
    if (a[k] != b[k]) begin
      a_gt_b = a[k];
      a_lt_b = ~a[k];
      a_eq_b = 0;
      disable compare_loop;
    end
    // if
  end
  // for loop
  a_gt_b = 0;
  a_lt_b = 0;
  a_eq_b = 1;
end
  // compare_loop
endmodule
```

Synthesis result
Function or Task

- **Function:**
  - no incomplete case statements or conditionals

- **Task**
  - same restrictions as functions + no timing control constructs
Function

module or_nand_4 (enable, x1, x2, x3, x4, y);
  input enable, x1, x2, x3, x4;
  output y;
  assign y = or_nand (enable, x1, x2, x3, x4);
endmodule

function or_nand;
  input enable, x1, x2, x3, x4;
  begin
    or_nand = ~(enable & (x1 | x2) & (x3 | x4));
  end
endfunction

Synthesis result
Task

module or_nand_5 (enable, x1, x2, x3, x4, y);
    input enable, x1, x2, x3, x4;
    output y;
    reg y;
always @ (enable or x1 or x2 or x3 or x4)
    or_nand (enable, x1, x2, x3, x4, y);
task or_nand;
    input enable, x1, x2, x3, x4;
    output y;
    begin
        y = !(enable & (x1 | x2) & (x3 | x4));
    end
endtask
endmodule
Constructs to Avoid

- Multiple event controls within the same procedural block
- Named event with edge-sensitive event control expression
- Feedback loops
- Procedural continuous assignment containing event or delay control
- fork…join
- wait
- External disable statement
- Procedural loops with timing controls
- Data-dependent loops
- Tasks with timing controls
- Sequential UDPs
Simulation Efficiency

```verilog
module or_nand_6 (enable, x1, x2, x3, x4, y);
input enable, x1, x2, x3, x4;
output y; reg y;
always @ (enable)
    if (enable) assign y = ~((x1 | x2) & (x3 | x4));
    else assign y = 1;
endmodule

module or_nand_7 (enable, x1, x2, x3, x4, y);
input enable, x1, x2, x3, x4;
output y; reg y;
always @ (enable or x1 or x2 or x3 or x4)
    if (enable) y = ~((x1 | x2) & (x3 | x4));
    else y = 1;
endmodule
```

- or_nand_6 might not be acceptable to some synthesis tools, but is more efficient than or_nand_7 in simulation
Example

always @ (opcode or a or b)
case (opcode)
  3'b111: out_y = a & b;
  3'b011: out_y = a | b;
  3'b001: out_y = a ^ b;
default: out_y = 2'b0;
endcase //Preferred for synthesis.

(a)

always @ (opcode)
case (opcode)
  3'b111: assign out_y = a & b;
  3'b011: assign out_y = a | b;
  3'b001: assign out_y = a ^ b;
default: assign out_y = 2'b0;
endcase //Preferred for simulation.

(b)

Figure 8.28 Equivalent functional descriptions, but with different simulation efficiencies.

- (b) is more efficient in simulation.
- In (b), inputs a,b are implicit inputs to the event control expression.
Control Logic for Multiplexed Datapaths

- 4-channel mux described with a continuous assignment and nested conditional operator

```verilog
module syn1_mux_4bits (y, sel, a, b, c, d);
  input [3:0] a, b, c, d;
  input [1:0] sel;
  output [3:0] y;

  assign y =
    (sel == 0) ? a:
    (sel == 1) ? b:
    (sel == 2) ? c:
    (sel == 3) ? d : 4'bx;
endmodule
```
Synthesis Result
Using case

module syn2_mux_4bits (y, sel, a, b, c, d);
  input  [3:0] a, b, c, d;
  input  [1:0] sel;
  output [3:0] y;
  reg     [3:0] y;

  // Simulation efficient
  always @ (sel)
  case (sel)
    0: assign y = a;
    1: assign y = b;
    2: assign y = c;
    3: assign y = d;
    default: assign y = 4'b0;
  endcase

  endmodule

  // Simulation and synthesis friendly
  always @ (sel or a or b or c or d)
  case (sel)
    0: y = a;
    1: y = b;
    2: y = c;
    3: y = d;
    default: y = 4'b0;
  endcase
Synthesis Result
Using if

module syn3_mux_4bits (y, sel, a, b, c, d);
input [3:0] a, b, c, d;
input [1:0] sel;
output [3:0] y;
reg [3:0] y;

// Simulation efficient
always @ (sel)
  if (sel == 0) assign y = a; else
  if (sel == 1) assign y = b; else
  if (sel == 2) assign y = c; else
  if (sel == 3) assign y = d; else
    assign y = 4`bx;
endmodule

// Simulation and synthesis friendly
always @ (sel or a or b or c or d)
  if (sel == 0) y = a; else
  if (sel == 1) y = b; else
  if (sel == 2) y = c; else
  if (sel == 3) y = d; else
    else y = 4`bx;

Synthesis result is the same as the one shown on page 30.
Control Logic at Select Line

**Example 8.19** The continuous assignment in `mux_logic` has logic determining whether `sig_a` or `sig_b` is selected. The description synthesizes to the circuit shown in Figure 8.30, which has logic at the control line of the mux.

```verilog
module mux_logic (y, select, sig_G, sig_max, sig_a, sig_b);
    input select, sig_G, sig_max, sig_a, sig_b;
    output y;

    assign y = (select == 1) || (sig_G == 1) || (sig_max == 0) ? sig_a : sig_b;

endmodule
```

It is wrong. It should be a circuit having the behavior of a 2-channel mux.
Unwanted Latches

- The **case** statements and conditionals (**if**) that do not include all possible cases or conditions are called “incompletely specified”.

- Incomplete specified **case** statements and conditionals may lead to the synthesis of unwanted latches.

- Example:

```verilog
module mux_latch (y_out, sel_a, sel_b, data_a, data_b);
input sel_a, sel_b, data_a, data_b;
output y_out;
reg y_out;

always @ (sel_a or sel_b or data_a or data_b)
    case ({sel_a, sel_b})
      2'b10: y_out = data_a;
      2'b01: y_out = data_b;
    endcase
endmodule
```
Generic Synthesis Result

When \((\text{sel}_a, \text{sel}_b) = 00, 11\), \(y_{\text{out}}\) keeps the old value.
Synthesis from Library Cells

mux2_a

data_a
data_b

latnb_a

sel_a

xor2_a

esdpupd_

y_out

enable

reset

CS 4120
Bad Examples Using case

```verilog
class module alu_incomplete1a (alu_out, data_a, data_b, enable, opcode);
input [2:0] opcode;
input [3:0] data_a, data_b;
input enable;
output reg [3:0] alu_out; // Note: scalar
assign alu_out = (enable == 1) ? alu_reg : 4'b0;
always @(opcode)
  case (opcode)
    3'b001: alu_reg = data_a | data_b;
    3'b010: alu_reg = data_a ^ data_b;
    3'b110: alu_reg = ~data_b;
  endcase
endmodule

module alu_incomplete2a (alu_out, data_a, data_b, enable, opcode);
input [2:0] opcode;
input [3:0] data_a, data_b;
input enable;
output alu_out;
reg [3:0] alu_reg;
assign alu_out = (enable == 1) ? alu_reg : 4'b0;
always @(opcode or data_a or data_b)
  case (opcode)
    3'b001: alu_reg = data_a | data_b;
    3'b010: alu_reg = data_a ^ data_b;
    3'b110: alu_reg = ~data_b;
  endcase
endmodule
```
Synthesis result
Good example using case

```verilog
module alu_complete (alu_out, data_a, data_b, enable, opcode);
    input [2:0]     opcode;
    input [3:0]     data_a, data_b;
    input           enable;
    output          alu_out;
    reg             [3:0]     alu_reg;

    assign alu_out = (enable == 1) ? alu_reg : 4'b0;

    always @ (opcode or data_a or data_b)
        case (opcode)
            3'b001:   alu_reg = data_a | data_b;
            3'b010:   alu_reg = data_a ^ data_b;
            3'b110:   alu_reg = ~data_b;
            default:  alu_reg = 4'b0;
        endcase
    endmodule
```
Bad example using if

```verilog
module incomplete_and (y, a1, a2);
    input    a1, a2;
    output   y;
    reg      y;

    always @ (a1 or a2)
        begin
            if ({a2, a1} == 2'b11) y = 1; else
            if ({a2, a1} == 2'b01) y = 0; else
            if ({a2, a1} == 2'b10) y = 0;
        end
    endmodule
```
Priority Structure

If the case items of a `case` statement are mutually exclusive, synthesis tool will treat them having equal priority, and synthesize a `mux` rather than priority structure.

An `if` statement will also synthesize to a `mux` when branching conditions are mutually exclusive.

```verilog
text
module mux_4pri (y, a, b, c, d, sel_a, sel_b, sel_c);
    input a, b, c, d, sel_a, sel_b, sel_c;
    output y;
    reg y;

always @ (sel_a or sel_b or sel_c or a or b or c or d)
    begin // Not mutually exclusive
        if (sel_a == 1) y = a; else
        if (sel_b == 0) y = b; else
        if (sel_c == 1) y = c; else
            y = d;
    end
endmodule
```

Treatment of Default Conditions

- Explicit assignments of x or z will be treated as “don’t cares” by a synthesis tool.
- This may lead a mismatch between the results obtained by simulating the Verilog code and the synthesized circuit.
Example - 8:3 Encoder (without Priority)

```verilog
module encoder (Data, Code);
    input [7:0] Data;
    output [2:0] Code;
    reg [2:0] Code;

    always @ (Data)
    begin
        if (Data == 8'b00000001) Code = 0; else
        if (Data == 8'b00000010) Code = 1; else
        if (Data == 8'b00000100) Code = 2; else
        if (Data == 8'b00001000) Code = 3; else
        if (Data == 8'b00010000) Code = 4; else
        if (Data == 8'b00100000) Code = 5; else
        if (Data == 8'b01000000) Code = 6; else
        if (Data == 8'b10000000) Code = 7; else Code = 3'bx;
    end

    // Alternative description is given below
```

```verilog
always @ (Data)
    case (Data)
        8'b00000001 : Code = 0;
        8'b00000010 : Code = 1;
        8'b00000100 : Code = 2;
        8'b00001000 : Code = 3;
        8'b00010000 : Code = 4;
        8'b00100000 : Code = 5;
        8'b01000000 : Code = 6;
        8'b10000000 : Code = 7;
        default        : Code = 3'bx;
    endcase
endmodule
```
Synthesis Result
**Example - 8:3 Priority Encoder**

```verilog
module priority (Data, Code, valid_data);

input [7:0] Data;
output [2:0] Code;
output valid_data;
reg [2:0] Code;
assign valid_data = |Data; // reduction or

always @ (Data)
begin
  if (Data[7]) Code = 7; else
  if (Data[6]) Code = 6; else
  if (Data[5]) Code = 5; else
  if (Data[4]) Code = 4; else
  if (Data[3]) Code = 3; else
  if (Data[2]) Code = 2; else
  if (Data[1]) Code = 1; else
    if (Data[0]) Code = 0; else
      Code = 3'bx;
end

/*// Alternative description is given below

always @ (Data)
case (Data)
  8'b1xxxxxx : Code = 7;
  8'b01xxxxxx : Code = 6;
  8'b001xxxxx : Code = 5;
  8'b0001xxx  : Code = 4;
  8'b00001xx  : Code = 3;
  8'b000001x  : Code = 2;
  8'b0000001x : Code = 1;
  8'b00000001 : Code = 0;
  default : Code = 3'bx;
endcase
*/
endmodule
Synthesis Result
Example - 3:8 Decoder

```verilog
module decoder (Code, Data);
    output [7:0] Data;
    input [2:0] Code;
    reg [7:0] Data;

    always @(Code)
    begin
        if (Code == 0) Data = 8'b00000001; else
        if (Code == 1) Data = 8'b00000010; else
        if (Code == 2) Data = 8'b00000100; else
        if (Code == 3) Data = 8'b00001000; else
        if (Code == 4) Data = 8'b00010000; else
        if (Code == 5) Data = 8'b01000000; else
        if (Code == 6) Data = 8'b10000000; else
        Data = 8'bX;
    end

    // Alternative description is given below
    always @(Code)
    begin
        case (Code)
            0 : Data = 8'b00000001;
            1 : Data = 8'b00000010;
            2 : Data = 8'b00000100;
            3 : Data = 8'b00001000;
            4 : Data = 8'b00010000;
            5 : Data = 8'b00100000;
            6 : Data = 8'b01000000;
            7 : Data = 8'b10000000;
            default: Data = 8'bX;
        endcase
    end
endmodule
```
Synthesis Result
Technology Mapping

A technology library contains cells implemented with certain technology.
Technology mapping converts a generic Boolean circuit to physical gates in a technology library.
Example

module badd_4 (Sum, C_out, A, B, C_in);

output [3:0] Sum;
output C_out;
input [3:0] A, B;
input C_in;

assign {C_out, Sum} = A + B + C_in;
endmodule

Use "+" operator

Use full adder cells
Another Result

Use 5-bit adders

(b)

CS 4120
Resource Sharing

Example 8.29 The use of parentheses in the description in res_share forces the synthesis tool to multiplex the datapaths and produce the circuit shown in Figure 8.39.

```verilog
module res_share (y_out, sel, data_a, data_b, accum);
  input [3:0] data_a, data_b, accum;
  input sel;
  output [4:0] y_out;

  assign y_out = data_a + (sel ? accum : data_b);
endmodule
```

Want to use 1 adder, instead of 2.
Synthesis Result

Implementation of a datapath forced to share resources.
module stuff_to_bus1 (data_to_bus, bus_enabled, clk);
    input bus_enabled, clk;
    output [31:0] data_to_bus;
    reg [31:0] ckt_output_to_bus;
    assign data_to_bus = (bus_enabled) ? ckt_output_to_bus : 32'b0;

    // Description of core circuit goes here to drive ckt_output_to_bus
endmodule
Bidirectional Buses

```verilog
module stuff_to_bus2 (data_to_from_bus, clk, send_data, rcv_data);
    input clk, send_data, rcv_data;
    inout [31:0] data_to_from_bus;
    reg [31:0] reg_to_bus;
    wire [31:0] data_to_from_bus, inbound_data;

    assign inbound_data = (rcv_data) ? data_to_from_bus : 32'bz;
    assign data_to_from_bus = (send_data) ? reg_to_bus : data_to_from_bus;

    // Behavior using inbound_data go here
endmodule
```

---

![Diagram of bidirectional circuit](image)
Bus Loading

module stuff_to_bus3 (data_to_from_bus, enab_a, enab_b, clk, rcv_data);
  input enab_a, enab_b, rcv_data, clk;
  inout [31:0] data_to_from_bus;
  reg [31:0] reg_A_to_bus, reg_B_to_bus;

  assign data_to_from_bus = (enab_a) ? reg_A_to_bus :
                             (enab_b) ? reg_B_to_bus : 32'b0;
endmodule // Not showing other logic.

Multiplexed bus drivers.
Tri-State Outputs and Don’t Cares

```verilog
module alu_with_z1 (alu_out, data_a, data_b, enable, opcode);
    input [2:0] opcode;
    input [3:0] data_a, data_b;
    input enable;
    output alu_out;     // scalar for illustration
    reg [3:0] alu_reg;

    assign alu_out = (enable == 1) ? alu_reg : 4'b0;

    always @ (opcode or data_a or data_b)
        case (opcode)
            3'b001: alu_reg = data_a | data_b;
            3'b010: alu_reg = data_a ^ data_b;
            3'b110: alu_reg = ~data_b;
            default: alu_reg = 4'b0;
            // alu_with_z2 has default: alu_reg = 4'bx;
        endcase
    endmodule
```

Don’t cares
Synthesis Results

with don’t cares

(a)

(b)

CS 4120