Synthesis of Combinational Logic

In theory, synthesis tools *automatically* create an optimal gate-level realization of a design from a high level HDL description.

In reality, the results depend on the skill of the designer using the tool, and his/her knowledge of how the HDL infers logic from language constructs.

It is important to adopt a vendor-specific style that is amenable to synthesis. Otherwise synthesis may fail or produce unexpected results.

It is important to know which constructs to avoid, and to be able to predict what the outcome will be for a particular construct.

A Verilog description consisting only of a netlist of combinational primitives *without feedback* can always be synthesized.

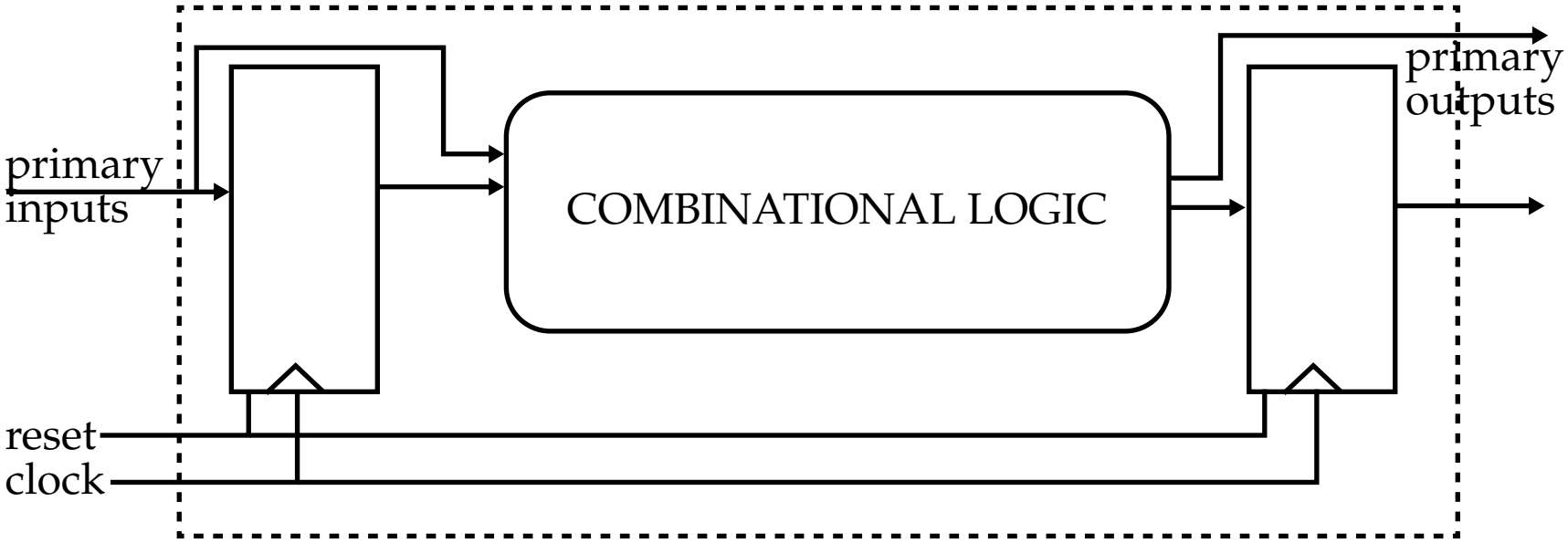
Some general rules:

- All storage elements should be controlled by an external clock and possibly a reset line.

Synthesis of Combinational Logic

Some general rules:

- The combinational part should be driven by primary inputs (through the ports) or internal storage elements.



- Avoid referencing the same variable in more than one **always** behavior. This can cause races, resulting in differences in pre-synthesized and post-synthesized behavior.

- Use only synchronous, resettable FFs in the design.

Synthesis of Combinational Logic

Some general rules:

- Timing constructs are ignored.
- Avoid code expressions that perform **explicit** Boolean operations on the logic values "x" and "z".

Commonly Supported Verilog Constructs

- Module declaration
- Port modes: **input**, **output** and **inout**
- Port binding by name or position
- Parameter declaration
- Connectivity nets: **wire**, **tri**, **wand**, **wor**, **supply0** and **supply1**
- Register variables: **reg** and **integer**
- Integer types in binary, decimal, octal and hex formats
- Scalar and vector nets
- Subranges of vector nets on RHS of assignment
- Module and primitive instantiation
- Continuous assignment
- Shift operator

Commonly Supported Verilog Constructs

- Conditional and concatenation operators
- Arithmetic, bitwise, reduction, logical and relational operators
- Procedural-continuous assignments (**assign** ... **deassign**)
- Procedural block statements (**begin** and **end**)
- **case**, **casex**, **casez**, **default**
- Branching: **if**, **if ... else**,
- **disable** (of procedural block)
- **for** loops
- Tasks: **task ... endtask**
- Functions: **function ... endfunction**

Synthesis tools do not support constructs used for transistor/switch level descriptions of behavior.

Constructs to avoid:

- Assignment with a variable used as bit select on LHS
- Global variables
- case equality, inequality (===, !===)

Constructs To Avoid

- **defparam**
- **event** and **fork ... join**
- **forever, while, repeat** and **wait**
- **initial**
- **pulldown, pullup**
- **force ... release**
- **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**

Styles For Synthesis of Combinational Logic

In general, do not include any aspects that attempt to model a specific technology.

Although Verilog supports feedback loops, e.g., cross-couple **nand** gates, they are to be avoided for synthesis.

If the rules are violated, synthesis will fail or produce a *sequential* circuit!

Commonly synthesized combinational logic:

- Multiplexer
- Encoder/Decoder
- Comparator
- Random Logic
- Lookup Table
- Adder/Subtractor/Multiplier
- ALU
- PLA Structure
- Parity Generator

Styles For Synthesis of Combinational Logic

The following descriptive styles can be used:

- Netlist of Verilog primitives
- Combinational UDP
- Continuous assignment
- Behavioral statement
- Function
- Task without delay or event control
- Interconnected modules of the above

Synthesis should be used even if the design is expressed as a netlist of primitives because it will remove redundant logic.

Redundant logic causes serious problems for test tools.

Continuous assignment statically bind an expression to a net variable.

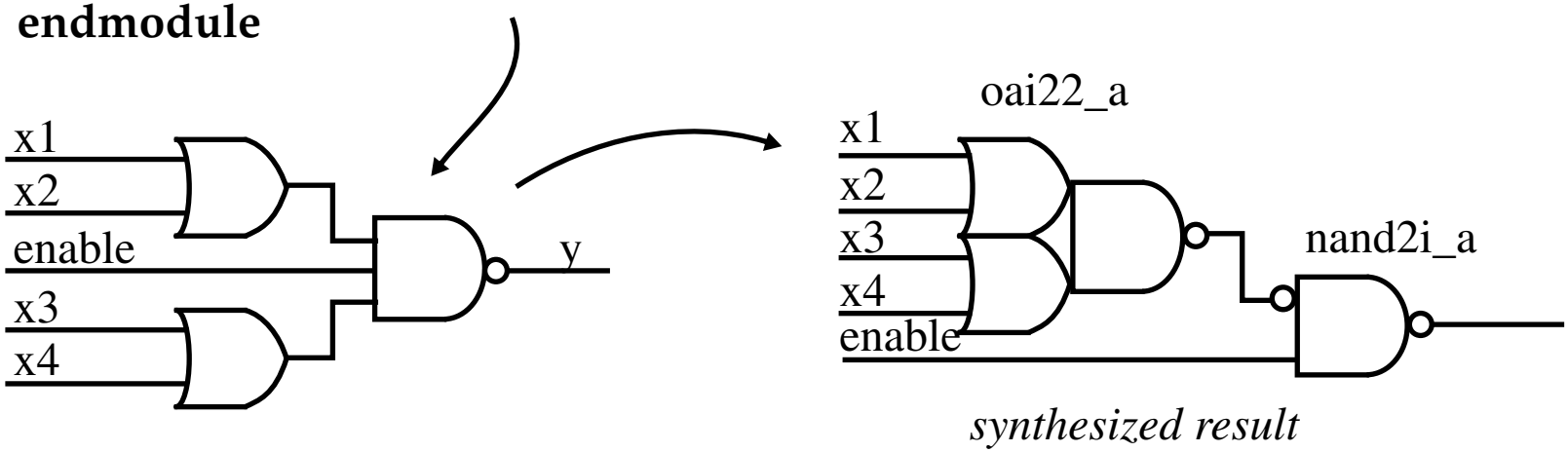
Synthesis tools translate continuous assignment statements into a set of equivalent Boolean equations.

Styles For Synthesis of Combinational Logic

For example:

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

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



synthesized result

Combinational Synthesis from a Cyclic Behavior

Key here is to make sure the behavior assigns value to the output **under all events** that affect the RHS expression of the assignment.

Failure to do so will produce unwanted *latches!!!*

Combinational Synthesis from a Cyclic Behavior

All inputs to a behavior designed to implement combinational logic *must* be included in the event control expression, either explicitly or implicitly.

assign dynamically binds and *implicitly* adds to the event control.

Also, any *control signals* whose transitions affect the assignments to the target register variables in the behavior are considered to be inputs to the behavior.

Any operands that appear on the RHS of any procedural or procedural-continuous assignment *must not* appear on the LHS of an expression.

Otherwise, the behavior has *implicit* feedback and will *not* synthesize into combinational logic.

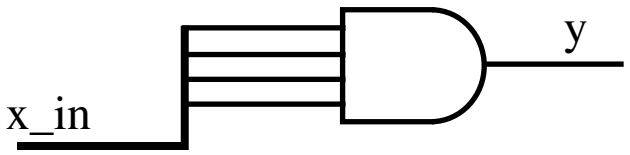
```
module or_nand_2 (enable, x1, x2, x3, x4, y);
   input enable, x1, x2, x3, x4;
   output y;
   reg y;                          // Does NOT imply a hardware register
   always @( enable or x1 or x2 or x3 or x4 )     // Synthesis is identical
      begin                                       // to previous circuit
         y = ~(enable & (x1 | x2) & (x3 | x4));
      end
endmodule
```

Combinational Synthesis from a Cyclic Behavior

```
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;                                          // This guarantees that y is assigned
         for (k = 0; k <= word_length -1; k = k + 1)     // to within the behavior
            if ( x_in[k] == 0 )
            begin
               y = 0;
               disable check_for_0;
            end
      end
endmodule
```

// Synthesis result shows no sign of the for loop

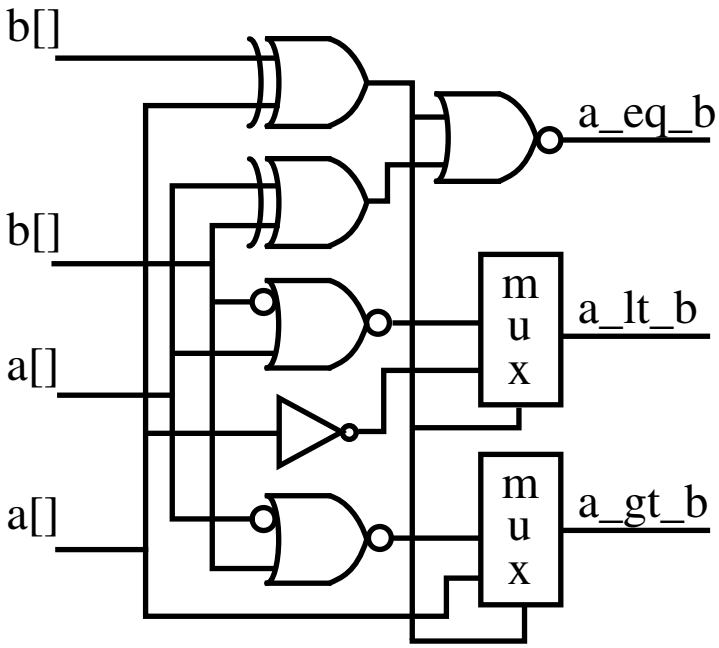


Combinational Synthesis from a Cyclic Behavior

```
module comparator( a, b, a_gt_b, a_lt_b, a_eq_b);
   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)
            if ( a[k] != b[k] )
               begin
                  a_gt_b = a[k];
                  a_lt_b = ~a[k];
                  a_eq_b = 0;
                  disable compare_loop;
               end
         a_gt_b = 0;
         a_lt_b = 0;
         a_eq_b = 1;
      end                  // compare_loop
endmodule
```



Implicit Event Control

This appears to violate the previous "general rule" that the event control expression must contain all the inputs of a behavior.

```
module or_nand_6 (enable, x1, x2, x3, x4, y);
   input enable, x1, x2, x3, x4, y;
   output y;
   reg y;                          // This synthesizes to the same circuit shown
   always @(enable)                // earlier but is more efficient w.r.t. simulation
      if (enable)
         assign y = ~((x1 | x2) & (x3 | x4));
      else
         assign y = 1;             // deassign implies sequential behavior
endmodule
```

Avoid for combinational synthesis: all of these imply a sequential behavior:

- Multiple event controls with the same procedural block
- Named event with edge-sensitive event control expression
- Procedural loops w/ timing controls, feedback and data-dependent loops
- Procedural continuous assignment containing event or delay control
- Parallel threads of activity: **fork ... join** and suspended activity: **wait**
- External disable statement, tasks w/ timing controls and sequential UDPs

Unexpected and Unwanted Latches

The golden rule is combinational logic *must* specify the value of the output for all values of the input.

If violated, the description implies that the output should *retain its residual value* under the unspecified conditions.

Be sure that **case** and conditional **if** stmts are complete (an incomplete conditional operator, i.e., **? ... :**, causes a syntax error).

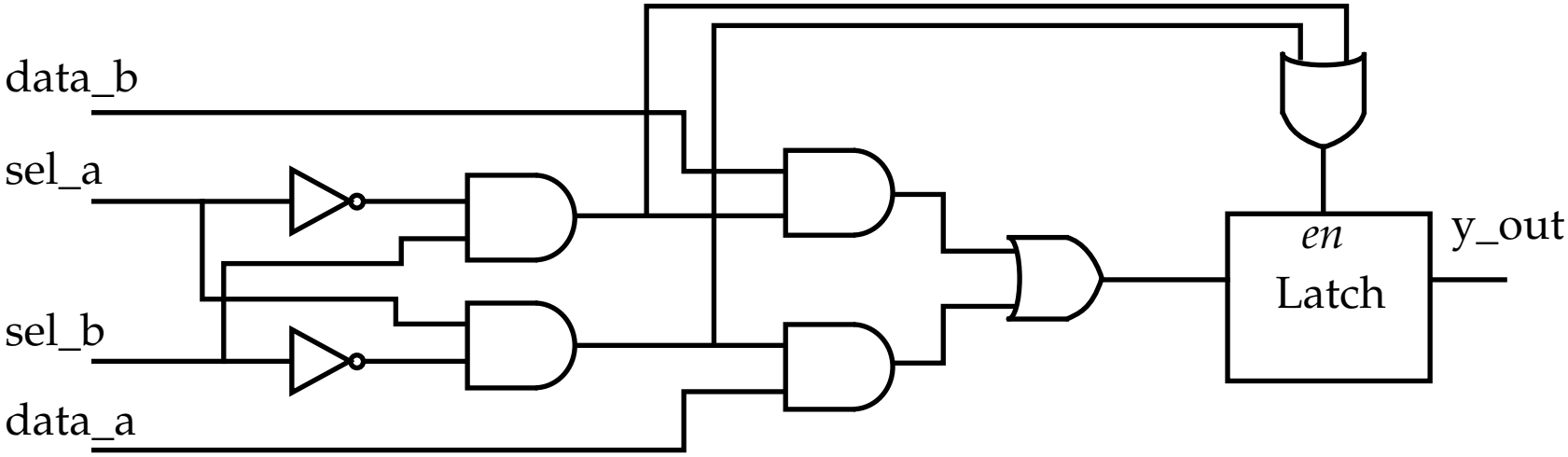
```
module mux_latch( y_out, sel_a, sel_b, data_a, data_b);
   input sel_a, sel_b, data_a, data_b;
   output y_out;
   reg y_out;

   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
```

A latch is added and *enabled* by the conditions handled.

Unexpected and Unwanted Latches

One possible synthesis result:



Text gives other examples that show when this happens.

Synthesis of Priority Structures

The **case** and **if** stmts implicitly assign priority, i.e., first block has higher priority than the second, etc.

Synthesis tool will determine if the case items are mutually exclusive, and if so, will give them equality priority and use a MUX.

Synthesis of Priority Structures

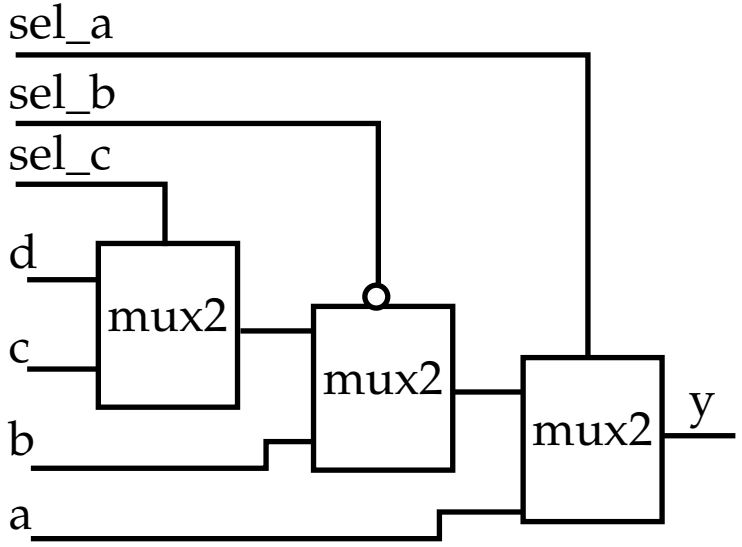
Otherwise, a priority structure is synthesized.

For the **if** stmt, a MUX is used when the branching is specified by mutually exclusive conditions.

```
module mux_4pri( y, a, b, c, 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
         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
```

```
// Priority because sel_a decodes
// a independently of sel_b or
// sel_c
```



Synthesis of Priority Structures

In contrast:

```
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
endmodule
```

//Default conditions
// are treated as *don't*
// *cares* in synthesis

This code is optimized and synthesized as three 4-input OR gates, each one receiving one of Data[7:0] signals and generating a bit of Code[2:0]

Synthesis of Priority Structures

For an 8:3 *priority encoder*, an implied priority needs to be present.

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

   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
endmodule
```

OR

```
casex (Data)
   8'b1xxxxxxx Code = 7; else
   8'b01xxxxxx Code = 6; else
   8'b001xxxxx Code = 5; else
   8'b0001xxxx Code = 4; else
   8'b00001xxx Code = 3; else
   8'b000001xx Code = 2; else
   8'b0000001x Code = 1 else
   8'b00000001 Code = 0; else
   default Code = 3'bx;
endcase
```

The synthesized circuit, here, is much more complex, so care should be taken to make sure conditions are *mutually exclusive* when priority is not needed.

Other Topics Covered in the Text

Sharing datapath resources.

Specifying bi-directional buses

Specifying three-state outputs