VLSI System Design

Synthesis - Concepts and Applications

李昆忠
國立成功大學電機系

06-2757575 X 62371
kjlee@mail.ncku.edu.tw
Synthesis and Optimization

- Behavioral synthesis
- RTL Synthesis
- Logic optimization
- Structure-to-layout synthesis
- Placement and Routing
- Synthesizable Verilog codes
- Practical synthesis considerations with Synopsys
Behavioral Synthesis

Behavioral description \[\rightarrow\] logic (gate level)

or

Behavioral description \[\rightarrow\] RTL

Behavioral description examples

- \(c = a + b\)

\[
\begin{array}{ccc}
  a & b & c \\
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 0 \\
\end{array}
\]

- Events, tasks
- Algorithms
- Architectures
RTL Synthesis

RTL HDL → Logic (gate-level)

RTL HDL captures the following attributes

• Control flow using if - then - else or case
• Iteration
• Hierarchy structure
• Word widths, bit vectors and bit fields
• Sequential versus parallel operations
• Register specification and allocation
• Arithmetic, logic and comparison operations

— Synopsys Verilog compiler is basically for RTL synthesis.
Logic Synthesis

• Convert an RTL or gate-level description of the circuit under design to an the optimized gate-level description, based on
  – A standard cell library, also known as technology library
  – Design constraints such as timing, area, and power

• Logic Synthesis = Translation + Optimization
  – Technology independent optimization by algebraic or Boolean techniques.
  – Technology-mapping — map the technology-independent description to specific library standard cells, FPGA or other logic gates.
Typical Flow of Optimization

- Network organization, redundancy removal, converting to two-level description, etc..

Summary of critical synthesis tasks:
- Detect and eliminate redundant logic
- Detect combinational feedback loops
- Exploit don’t care conditions
- Detect unused states
- Detect and collapse equivalent states
- Make state assignments
- Synthesize optimal, multilevel realizations of logic subject to constraints in a physical technology
  - Two-level minimization — optimized SOP or POS
  - Algebraic decomposition — multiple-level synthesis
  - Iterative
Design Constraint

- Timing constraint—The circuit must meet certain timing requirements.
- Area constraint—The area of the final layout must not exceed a limit.
- Power—The power dissipation in the circuit must not exceed a threshold.
HDL/Design Compiler Interface

- Synopsys
- design_vision
- dc_shell
- command
- GUI

HDL/Design Compilers
RTL Design Flow (I)

1. Write an RTL description in the Verilog HDL.
   - The description is used by the Synopsys® Verilog HDL compiler and the Verilog-XL Simulator.

2. Provide a stimulus as the testbench.

3. Use the stimulus from Step 2 to simulate the RTL description and verify the output data.

4. Synthesize the RTL description with the Synopsys® Verilog HDL compiler to create an internal representation of the design.

5. Use Synopsys® Design Vision to produce an optimized gate-level description based on the target library.
6. Use Synopsys® Design Vision to output a gate-level Verilog description using the library components as leaf-level cells of the design.

7. Use the stimulus from Step 2 to simulate the gate-level description to gather the output data.

8. Compare both the outputs of original (RTL) and gate-level descriptions to verify that the synthesized circuit is correct.
Y-chart

Structural Description
- Processor
- Memory
- Peripheral interface
- Registers, ALUs, etc.
- Logic netlist, schematic

Behavioral Description
- Algorithm
- Dataflow/RTL
- Boolean algebra

View
- Hierarchical modules and primitive instantiations
- Cell geometry
- Photomask
- Layout
- Database

Physical Description

Procedural assignment
- Nonblocking assignment
- Continuous assignment
## Synthesis Terminology

<table>
<thead>
<tr>
<th>Level</th>
<th>Graphical Representation</th>
<th>HDL, Graphical specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partioning, Pipelining</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Behavioral</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling, Allocation</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RTL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>State Machines, Logic</td>
<td></td>
<td>HDL, Bubble diagrams</td>
</tr>
<tr>
<td>and RTL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-level, 2-level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimization</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Logic</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Technology mapping/translation</td>
<td></td>
<td>Logic equations, HDL, Gates</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td></td>
<td>CIF/GDSII</td>
</tr>
<tr>
<td>Place and Route</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synthesis-Tool Organization

- Verilog Description
  - TRANSLATION ENGINE
    - Two-Level Logic Function
  - OPTIMIZATION ENGINE
    - Optimized Multilevel Logic Functions
  - MAPPING ENGINE
    - Technology Implementation

- Technology Libraries
Synthesis Flow

1. RTL Description
2. Translation
3. Un-optimized intermediate representation
4. Logic Optimization
5. Technology Mapping and Optimization
6. Optimized Gate-Level representation
7. Library of available gates, and leaf-level cells (technology library)

Design Constraints
Structure to Layout Synthesis

- Usually logic gate layout
- Major problems
  - Placement
  - Routing
Structure-to-layout Synthesis

Placement — placing modules to minimize area or cycle time
  - Min-cut algorithm
  - Kernighan-Lin algorithm
  - Simulating annealing
  - ...

Routing — taking a module placement and a list of connections and connects the modules with wires.
  - Channel routing
  - Maze routing
  - Switch box routing
  - ...
**Placement**

1. **Pairwise interchange**
   - Divide components into two partitions
   - Exchange a pair of nodes, one from each partition, to see whether better result is obtained. If yes, do the exchange.
   - Iteration until no more improvement

2. **Min-Cut**
   - Divide components into two partitions
   - Exchange a pair of nodes, one from each partition, to see whether better result is obtained. If yes, do the exchange.
   - Iteration until no more improvement
Placement (cont.)

3. Kernighan-Lin algorithm
   • Similar to Min-Cut, but exchange a set of nodes. (Selects the pairs of components to exchange and allows multiple exchanges to occur before deciding to accept the sequence of exchanges)

4. Simulated annealing
   • Analogue to the annealing of metal
     — heat to high temp $\rightarrow$ atoms move fast and randomly
     — cool $\rightarrow$ atoms tend to fall into proper places
   • An initial placement is selected, then each component is allowed to move.
   • Acceptances of improved moves and worsen moves are given different probabilities.
   • Increase the acceptance prob. for improved moves and decrease the acceptance prob. for worsen moves during “cooling” process to gradually achieve the best solution or next-to-best solution.
Lee’s algorithm—greedy and optimum.

• Algorithm used for finding a path between any two vertices on a planar rectangular grid.
• The exploration phase of Lee’s algorithm is an improved version of the breadth-first search.
A greedy and optimum solution

1. Order the nets
2. track = 0 ;
3. while (more nets){
   track++; 
   search the nets in order, if any one can fit in the current track then fit and remove it ;
}

Ex:

```
1  2  3  4  5  6  7  8
```

```
1  2  3  4  5  6  7  8
```

```
1  2  3  4  5  6  7  8
```

```
1  2  3  4  5  6  7  8
```
Synthesizable Verilog Codes

- Synopsys can’t accept all kinds of Verilog constructs.
- Synopsys can only accept a subset of Verilog syntax and this subset is called “Synthesizable Verilog Code.”
- Understanding how to write synthesis-friendly Verilog models is the key to automated design methods.
- The same situation also exists in VHDL.
Unsupported Verilog Constructs (1/5)

Definitions and Declarations

- `primitive` definition
- `real` declaration
- `time` declaration
- `event` declaration
- `triand`, `trior`, `tri1`, `tri0`, and `trireg` net types
- ranges and arrays for `integers`
- `specify` and `endspecify`
Unsupported Verilog Constructs (2/5)

• Statements (1/2)
  – `defparam`
  – `initial`
  – `repeat` (they are not supported because equivalent functionality can be constructed by other constructs)
  – `delay` (delay-based timing control, intra-assignment timing control)
  – `event` (named event control)
  – `wait` (level-sensitive timing control)
  – `forever`
Unsupported Verilog Constructs (3/5)

• Statements (2/2)
  – **fork/join** (helpful in modeling complex waveform in testbenches and abstract models of behavior)
  – **assign/deassign** for registers (commonly used for controlled periods of time)
  – **force/release** for registers and nets (typically used in the interactive debugging process)
    – * Recommended to be used in stimulus or as debug statement.
  – Assignment statements with a variable used as a bit-select on the left side of the equal sign.
Unsupported Verilog Constructs (4/5)

• Operators
  – case equality (===) and case inequality (!==)
  – division and modulus operators for variables

• Miscellaneous constructs
  – hierarchical names within a module
  – ‘ifdef, ‘endif, and ‘else compiler directives
Unsupported Verilog Constructs (5/5)

• Switch-level Constructs
  – nmos, pmos, cmos
  – rnmos, rpmos, rcmos
  – tran, tranif0, trnaif1
  – rtran, rtranif0, rtranif1
  – pullup, pulldown
Language Constructs commonly supported by Synthesis Tools (I)

- Module declaration
- Port name: **input**, **output**, and **inout**
- Port binding by name
- Port binding by position
- Local parameter declaration
- Connectivity nets: **wire**, **tri**, **wand**, **wor**, **supply0**, and **supply1**
- Register variables: **reg**, **integer**
- integer types in binary, decimal, octal, hex format
- Scalar and vector nets
- Sub_range of vector nets on RHS of assignment
- Module and macro_module instantiation
- Primitive instantiation
Language Constructs commonly supported by Synthesis Tools (II)

- Continuous assignments
- Shift operator
- Conditional operator
- Concatenation operator (including nesting)
- Arithmetic, bitwise, reduction, logical and relational operators
- Procedure block assignments (`begin … end`)
- `case`, `casex`, `casez`, `default`
- Branching: `if`, `if … else`, `if … else … if`
- `disable` (of procedure block)
- `for` loop
- Tasks: `task … endtask` (no timing or event control)
- Functions: `function … endfunction`
Practical considerations

- Non-blocking & blocking statements
- Resource sharing
- Conditional operator
- If, case, for statements
- Parenthesis – multi-level circuit
- Minimizing registers
- Don’t care inference
- Sharing complex operators
- Finite state machine
- Latches & Flip-flops
- Synopsys compiler directives
- Some Synthesis tips
Blocking and non-Blocking (1/2)

- Use non_blocking assignments within sequential always block.

```verilog
always @(posedge clock) begin
    x <= a;
    y <= x;
    z <= y;
end
```

```verilog
always @(posedge clock) begin
    x = a;
    y = x;
    z = y;
end
```
Blocking and non-Blocking (2/2)

- Use blocking assignments within combinational always block

```verilog
always @(a or b or x) begin
    x = a & b;
    y = x | b;
    x = a;
end
```

expect

```verilog
always @(a or b or x) begin
    x <= a & b;
    y <= x | b;
    x <= a;
end
```

may not expect

```verilog
always @(a or b or x) begin
    x <= a & b;
    y <= x | b;
    x <= a;
end
```
Resource Sharing

- Operations can be shared if they lie in the same always block – One adder is enough for the following example.

```verilog
always @(a or b or c or sel) begin
    if (sel)
        z = a + b;
    else
        z = c + d;
end
```
Conditional Operator

- The value assigned to LHS is the one that results in TRUE from the expression evaluation
- This can be used to model multiplexer
- Can be nested

```vhdl
assign out = (sel == 2'b00) ? in1 :
            (sel == 2'b01) ? in2 :
            (sel == 2'b10) ? in3 :
            (sel == 2'b11) ? in4 :
            1'bx;
```
Combinational Always Block

- Sensitivity list must be specified completely, otherwise synthesis may mismatch with simulation

always @(a or b or c)
f = a&b|c;

always @(a or b)
f = a&b|c;

Warning: Variable ‘c’ is being read
in routine train line 6 in file but does not occur in the timing control of the block which begins there. (HDL-180)
If Statement (1/3)

- Provide for more complex conditional actions, each condition expression controls a multiplexer
- Legal only in function & always construct
- Syntax

```plaintext
if (expr)
    begin
    ... statements ...
    end
else
    begin
    ... statements ...
    end
```
If Statement (2/3)

- What’s the difference between these two coding styles?

```verilog
module mult_if(a, b, c, d, e, sel, z);
input a, b, c, d, e;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or e or sel)
begin
    z = e;
    if(sel[0]) z = a;
    if(sel[1]) z = b;
    if(sel[2]) z = c;
    if(sel[3]) z = d;
end
endmodule
```

```verilog
module single_if(a, b, c, d, e, sel, z);
input a, b, c, d, e;
input [3:0] sel;
output z;
reg z;
always @(a or b or c or d or e or sel)
begin
    z = e;
    if(sel[3]) z = d;
    else if(sel[2]) z = c;
    else if(sel[1]) z = b;
    else if(sel[0]) z = a;
end
endmodule
```
If Statement (3/3)

mult_if

single_if
Case Statement (1/8)

• Legal only in the function & always construct
• Syntax

```
  case (expr)
    case_item1: begin
      ... statements ...
    end
    case_item2: begin
      ... statements ...
    end
    default: begin
      ... statements ...
    end
  endcase
```
Case Statement (2/8)

• A case statement is called a full case if all possible branches are specified

```verilog
always @(bcd) begin
    case (bcd)
        4'd0: out = 3'b001;
        4'd1: out = 3'b010;
        4'd2: out = 3'b100;
        default: out = 3'bxxx;
    endcase
end
```
Case Statement (3/8)

- If a case statement is not a full case, it will infer a *latch* to retain its previous value (as inferred by the semantic).

```verilog
case (bcd)
    4’d0: out = 3’b001;
    4’d1: out = 3’b010;
    4’d2: out = 3’b100;
endcase
```

```verilog
always @(bcd) begin
    case (bcd)
        4’d0: out = 3’b001;
        4’d1: out = 3’b010;
        4’d2: out = 3’b100;
    endcase
end
```
Case Statement (4/8)

• If you do not specify all possible branches, but you know the other branches will never occur, you can use “//synopsys full_case” directive to specify full case

always @(bcd) begin
    case (bcd) //synopsys full_case
        4'd0:out=3'b001;
        4'd1:out=3'b010;
        4'd2:out=3'b100;
    endcase
end
Case Statement (5/8)

- **Note:** the second case item does not modify `reg2`, causing it to be inferred as a latch (to retain last value).

```vhdl
case (cntr_sig) // synopsys full_case
2'b00 : begin
    reg1 = 0 ;
    reg2 = v_field ;
end
2'b01 : reg1 = v_field ; /* latch will be inferred for reg2*/
2'b10 : begin
    reg1 = v_field ;
    reg2 = 0 ;
end
endcase
```
Case Statement (6/8)

Two possible ways we can assign a default value to next_state

1. 
   ```
   out = 3'b000 ; // this is called unconditional assignment
   case (condition)
   ...
   endcase
   ```

2. 
   ```
   case (condition)
   ...
   default : out = 3'b000 ; // out=0 for all other cases
   endcase
   ```
Case Statement (7/8)

- You can declare a case statement as parallel case with the “//synopsys parallel_case” directive.

```verilog
always @(u or v or w or x or y or z) begin
  case (2'b11) //synopsys parallel_case
    u: out=10'b0000000001;
    v: out=10'b0000000010;
    w: out=10'b0000000100;
    x: out=10'b0000001000;
    y: out=10'b0000010000;
    z: out=10'b0000100000;
    default: out=10'b0000000000;
  endcase
end
```
Case Statement (8/8)

- If HDL Compiler can’t determine those case branches are parallel, its synthesized hardware will include a priority decoder.

```vhdl
always @(u or v or w or x or y or z) begin
  case (2'b11)
  u: out = 10'b0000000001;
  v: out = 10'b0000000010;
  w: out = 10'b0000000100;
  x: out = 10'b0000001000;
  y: out = 10'b0000010000;
  z: out = 10'b0000100000;
  default: out = 10'b0000000000;
  endcase
end
```
### Priority / Non-Priority Circuit

<table>
<thead>
<tr>
<th>Priority encoded</th>
<th>No priority encoded</th>
</tr>
</thead>
</table>
| always @(X or Y or Z) begin  
  if (Z) value = result4;  
  else if (Y) value = result3;  
  else if (X) value = result2;  
  else value = result1;  
end |
| always @(X or Y or Z)  
  if (Z) value = result4;  
  else if (Y) value = result3;  
  else if (X) value = result2;  
  else value = result1;  
endcase |

//synopsys parallel_case

Case 1b1:  
X: value = result2;  
Y: value = result3;  
Z: value = result4;  
default: value = result1;  
endcase
For Loop

- Provide a shorthand way of writing a series of statements.
- Loop index variables must be integer type.
- Step, start & end value must be constant.
- In synthesis, for loops are "unrolled", and then synthesized.

- Example

```verilog
always @(a or b) begin
    for( k=0; k<=3; k=k+1 ) begin
        out[k]=a[k]^b[k];
        c=(a[k]|b[k])&c;
    end
end
```

- Examples

```verilog
out[0] = a[0]^b[0];
out[1] = a[1]^b[1];
```
Merging Cascaded Adders with a Carry

- HDL compiler replaces two adders with a simple adder that has a carry input. To infer an adder with a carry input, set the variable `hdlin_use_cin` to true (the default is false).

```verilog
class module carry(sum, a, b, cin);
  output sum;
  input a, b, cin;
  assign sum = cin + a + b;
endmodule
```

dc_shell > hdlin_use_cin = true
Use Parentheses Properly

Restructured expression tree with subexpression preserved

Restructured according to the data arrival time & the dependency relationship

\[ \text{out} = ((a+(b+c))+d+e)+f; \]

\[ \text{out} = ((a+(b+c))+(d+e))+f; \]
Coding Skill -- Operator in if (1/2)

• We assume that signal “A” is the latest arrival signal.

Before_improved
module cond_oper(A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D; // A is latest arriving
output [N-1:0] Z;
reg [N-1:0] Z;
always @(A or B or C or D)
begin
  if (A + B < 24)
    Z <= C;
  else
    Z <= D;
end
endmodule

Improved
module cond_oper_improved (A, B, C, D, Z);
parameter N = 8;
input [N-1:0] A, B, C, D; // A is latest arriving
output [N-1:0] Z;
reg [N-1:0] Z;
always @(A or B or C or D)
begin
  if (A < 24 - B)
    Z <= C;
  else
    Z <= D;
end
endmodule
Coding Skill -- Operator in if (2/2)

- In this example, not only latency reduced, but also area reduced.

Before improved   Improved
Writing Efficient Circuit Description

• Restructure a design that makes repeated use of several large components, to minimize the number of instantiations.

• In a design that needs some, but not all, of its variables or signals stored during operation, minimize the number of latches or flip-flops required.

• Consider collapsing hierarchy for more-efficient synthesis for small circuit, but use hierarchy for large circuit.
Minimizing Registers (1/2)

- In an **always** block that is triggered by a clock edge, every variable that has a value assigned has its value held in a flip-flop.

```verilog
always @(posedge clock) begin
    if (reset)
        count = 0; // reg [2:0] count;
    else
        count = count + 1;
    and_bits = & count;
    or_bits = | count;
    xor_bits = ^ count;
end
endmodule
```

- Use six flip-flops; however, `and_bits`, `or_bits`, and `xor_bits` depend solely on the value of `count`. 
Minimizing Registers (2/2)

- Assign the outputs from within an asynchronous `always` block to avoid implying extra registers.

```vhdl
always @(posedge clock) begin //synchronous
    if (reset)
        count = 0;
    else
        count = count + 1;
end
always @(count) begin //asynchronous
    and_bits = & count;
    or_bits = | count;
    xor_bits = ^ count;
end
```
Separating Sequential and combinational Ckts

- Make a separate *always* block within edge trigger (synchronous) and no edge trigger (asynchronous)

```verbatim
always @(posedge clk or negedge reset) // state vector flip-flops (sequential)
    if (!reset)
        current_state = 0;
    else
        current_state = next_state;
always @(in1 or in2 or current_state) // output and state vector decode (combinational)
    case (current_state)
        0: begin
            next_state = 1;
            out = 1'b0;
            end
        1: begin
            next_state = 1'b2;
            out = in1;
            end
        2: ....
    endcase
```
Don’t Care Inference

In some cases, using don’t care values as default assignments can cause these problems:

- Don’t care values create a greater potential for mismatches between simulation and synthesis.
- Defaults for variables can hide mistakes in the Verilog code.
- To a simulator, a don’t care is a distinct value, different from a 1 or a 0.
- In synthesis, a don’t care becomes a 0 or a 1 (and hardware is built that treats the don’t care value as either a 0 or a 1).

The safest way to use don’t care values is to

- Assign don’t care values only to output ports.
- Make sure that the design never reads those don’t care values.
Sharing Complex Operators (1/2)

- To share most operators
  - Noncomputable array index
  - Function call
- Example of sharing noncomputable array index:

```verilog
always @ (a or i or j or c) begin
  z = 0; y = 0;  //input [7:0] a;
  if (c) begin
    z = a[i];
  end
  else begin
    y = a[j];
  end
end
```
always @(a or i or j or c) begin
  if(c) begin
    index = i;
    end
  else begin
    index = j;
    end
  temp = a[index];
  z = 0; y = 0;
  if(c) begin
    z = temp;
    end
  else begin
    y = temp;
    end
end
Implicit Finite State Machine

- You can describe a FSM implicitly without defining a state register.
- Each clock represents a transition to another state.

```verbatim
always begin
    @(posedge clk)
    total <= data;
    @(posedge clk)
    total <= total + data;
    @(posedge clk)
    total <= total + data;
end
```

Diagram of state transitions:

```
S0  -----> S1 -----> S2 -----> S0
```
Finite State Machine Directive

- //synopsys enum enum_name
  - Use with Verilog parameter state to specify state machine encoding and where they are used.

- //synopsys state_vector vector_name
  - Indicate which variable is chosen as a state vector.

/* Define states and encodings */
parameter [2:0] // synopsys enum code
  Grant_A=3'b001, Wait_A=3'b011, Timeout_A1=3'b111,
  Grant_B=3'b010, Wait_B=3'b110, Timeout_B1=3'b101;
reg [2:0] /* synopsys enum code */ present_State, next_State;

//synopsys state_vector present_State
reg ACKA, ACKB, TIMESTART;
always @ (REQA or REQB or TIMEUP or reset or present_State)
begin
FSM Optimizing Steps

• Extracting the FSM logic
  - Recognizing, extracting and translating the FSM portion into an internal data structure (SYNOPSYS state table format).

• Optimizing the state table
  - Minimize the number of states.
  - Assign the best state encodings (state assignment) w.r.t. QOR (quality of result).
  - Minimize the combinational logic of the FSM.
State Encoding

- Design Compiler tool can use one of four state-encoding styles
  - one-hot
    - has only one bit 1 position and other bit positions are 0
    - many flip-flops required
    - simple combinational logic and fastest machine
  - binary or gray
    - encoding styles assign state code to ordered states, based on a binary or gray numbering sequence
    - require the least number of flip-flops – $\log_2(\text{number of states})$
  - auto
    - the best to reduce the complexity of the combinational logic of the FSM while using a minimum number of encoding bits
Synthesis of Combinational Logic

- Synthesizable combinational logic can be described by
  - a netlist of structural primitives
    - A design that is expressed as a netlist of primitives should be synthesized to remove any redundant logic before mapping the design into a technology.
  - a set of continuous assignment statement
    - The expression that assigns value to a net variable in a continuous assignment statement will be translated by the synthesis tool to an equivalent Boolean equation.
  - a level-sensitive cyclic behavior (e.g., always @(a or b))
    - It will be synthesized to combinational logic if it assigns a value to each output for every possible value of its inputs.
Synthesis of Sequential Logic with Latches

- Latches are synthesized in two ways: intentionally and accidentally.
  - **Accident:** If the output of combinational logic is not completely specified for all cases of inputs, then a latch will be inferred.
  - **Intention:** A synthesis tool infers the need for a latch when a register variable in a level-sensitive behavior is assigned value in some threads of activity, but not in others (e.g., an incomplete if statement in a behavior)
Synthesis of Sequential Logic with Flip-Flops

- Flip-flops are synthesized only from edge-sensitive cyclic behaviors, but not every register variable that is assigned value in an edge-sensitive block is synthesized to a flip-flop.
- A register variable in an edge-sensitive block will be synthesized as a flip-flop if
  - it is referenced outside the scope of the block,
  - it is referenced within the block before it is assigned value, or
  - it is assigned value in only some of the branches of the activity within the block.
- For an incomplete conditional statement (i.e., an if … else statement or a case statement), if the block is edge-sensitive, it will be synthesized to a logic that implements a “clock enable”
- If the event-control expression is sensitive to the edge of more than one signal, an if statement must be the first statement in the block.
D Flip-Flop With Asynchronous Set

- When inferring a D flip-flop with an asynchronous set or reset, include edge expressions for the clock and the asynchronous signals in the sensitivity list of the always block.

```verilog
module dff_async_set (DATA, CLK, SET, Q);
    input DATA, CLK, SET;
    output Q;
    reg Q;
    always @(posedge CLK or negedge SET)
        if (~SET)
            Q <= 1'b1;
        else
            Q <= DATA;
endmodule
```
D Flip-Flop With Synchronous Set

- When inferring a D flip-flop with an synchronous set or reset, include only the clock in the sensitivity list of the always block.

```verilog
module dff_sync_set (DATA, CLK, SET, Q);
    input DATA, CLK, SET;
    output Q;
    reg Q;
    //synopsys sync_set_reset "SET"
    always @(posedge CLK)
        if (SET)
            Q <= 1'b1;
        else
            Q <= DATA;
endmodule
```
D Flip-Flop With Asynchronous Set and Reset

• Use the one_hot directive to prevent priority encoding of the set and reset signals.
D Flip-Flop with Syn. and Asyn. Load Controls

- D flip-flops can have asynchronous or synchronous controls.

```verilog
module dff_a_s_load (ALOAD, SLOAD, ADATA, SDATA, CLK, Q);
input ALOAD, ADATA, SLOAD, SDATA, CLK;
output Q;
reg Q;
always @ (posedge CLK or posedge ALOAD)
    if (ALOAD)
        Q <= ADATA;
    else if (SLOAD)
        Q <= SDATA;
endmodule
```
Multiple Flip-Flops With Asyn. and Syn. Controls

• If a signal is synchronous in one block but asynchronous in another block,

```vhdl
//synopsys sync_set_reset "RESET"
always @(posedge CLK)
begin : infer_sync
  if (~RESET)
    Q1 <= 1'b0;
  else if (SLOAD)
    Q1 <= DATA1;  // note: else hold Q
end
always @(posedge CLK or negedge RESET)
begin: infer_async
  if (~RESET)
    Q2 <= 1'b0;
  else if (SLOAD)
    Q2 <= DATA2;
end
```
Limitations of D Flip-Flop Inference

- An if statement must occur at the top level of the always block.

```verbatim
always @(posedge clk or posedge reset)
    begin
        q = d;
        if (reset)
            . . .
    end

Error: The statements in this 'always' block are outside the scope of the synthesis policy (%s). Only an 'if' statement is allowed at the top level in this 'always' block. (ELAB-302)
```
Synopsys HDL Compiler Directive

- `//synopsys translate_on` & `//synopsys translate_off` control the HDL Compiler translation of Verilog code off & on.

```verilog
module trivial(a, b, f);
input a, b;
output f;
assign f = a & b;
//synopsys translate_off
initial $monitor(a, b, f);
//synopsys translate_on
endmodule
```
Some Synthesis Tip (1/4)

- An assignment to $x$ in a case or an if statement will be treated as a don't-care condition in synthesis.
- A synthesis tool treats `casex` and `casez` statements as `case` statement.
- If a conditional operator assigns the value $z$ to the right-hand side expression of a continuous assignment in a level-sensitive block, the statement will be synthesized to a three-state device driven by combinational logic.
- Use parentheses to control operator grouping and reduce the size of a circuit.
- A feedback-free netlist of combinational primitives will be synthesized into latch-free combinational logic.
Some Synthesis Tip (2/4)

- A set of feedback-free continuous assignments will be synthesized into latch-free combinational logic.
- A continuous assignment using a conditional operator with feedback will be synthesized into a latch.
- A Verilog description of combinational logic must assign value to the outputs for all possible values of the inputs.
- An if statement in a level-sensitive block will be synthesized to a latch if the statement assigns value to a register variable in some, but not all, branches (i.e., the statement is incomplete).
Some Synthesis Tip (3/4)

- A variable that is referenced within an edge-sensitive block before it is assigned value in the block will be synthesized as the output of a flip-flop. (e.g., nonblocking assignment)

- A variable that is assigned value in a block before it is referenced within the block, but is not referenced outside the block, will be eliminated by the synthesis process. (e.g., iteration constant)

- A variable that is assigned value by an edge-sensitive block and is referenced outside the block will be synthesized as the output of a flip-flop. (e.g., declared as an output port)
Some Synthesis Tip (4/4)

- Use two cyclic blocks to describe an explicit state machine: level-sensitive block to describe the combinational logic for the next state and outputs and an edge-sensitive block to synchronize the state transitions.
- Use the procedural assignment operator (=) in the level-sensitive cyclic blocks describing the combinational logic of a finite-state machine.
- Use the nonblocking assignment operator (<=) in the edge-sensitive cyclic blocks describing the state transitions of a finite-state machine and the register transfers of the datapath of a sequential machine.