Introduction to Digital VLSI Design

Verilog – Logic Synthesis with Verilog HDL

Lecturer: Gil Rahav
Semester B’, EE Dept. BGU.
Freescale Semiconductors Israel
Objectives

- Define **logic synthesis** and explain the benefits of logic synthesis
- Identify Verilog HDL constructs and operators accepted in logic synthesis. Understand how the logic synthesis tool interprets these constructs
- Explain the typical design flow, using logic synthesis. Describe the components in the logic synthesis-based design flow
- Describe verification of the gate-level netlist produced by logic synthesis
- Understand techniques for writing efficient RTL description
- Describe partitioning techniques to help logic synthesis provide the optimal gate-level netlist
- Design combinational and sequential circuits, using logic synthesis
Impact of Logic Synthesis

- **Logic synthesis** always existed even in the days of schematic gate-level design, but it was always done inside the designer’s mind
  - For large design, manual conversion was prone to human error
  - The designer could never be sure that the design constraints were going to be met until the gate-level implementation was completed and tested
  - A significant portion of the design-cycle was dominated by the time taken to convert a high-level design into gates
  - The turnaround time for redesign of blocks was very high
  - What-if scenarios were hard to verify
  - Each designer would implement design blocks differently
  - If a bug was found in the final, gate-level design, this would sometimes require redesign of thousands of gates
  - Timing, area, and power dissipation in library cells are fabrication-technology specific
  - Design reuse was not possible (technology specific, hard to port, …)
The advent of computer-aided logic synthesis tools has automated the process of converting the high-level description to logic gates:

- High-level design is less prone to human error
- High-level design is done without significant concern about design constraints
- Conversion from high-level design to gates is fast
- Turnaround time for redesign of blocks is shorter
- What-if scenarios are easy to verify
- Logic synthesis tools optimize the design as a whole
- If a bug was found in the final, gate-level design, the designer goes back and changes the high-level description to eliminate the bug
- Logic synthesis tools allow technology-independent design
- Design reuse is possible for technology-independent descriptions
For the purpose of logic synthesis, designs are currently written in an HDL at a register transfer level (RTL)

- The term RTL is used for an HDL description style that utilizes a combination of dataflow and behavioral constructs

Verilog and VHDL are the two most popular HDLs used to describe the functionality at the RTL level

Logic synthesis tools take the register transfer-level HDL description and convert it to an optimized gate-level netlist

- RTL-based synthesis is currently the most popular design method
Verilog HDL Synthesis: Verilog Constructs

- In general, any construct that is used to define a cycle-by-cycle RTL description is acceptable to the logic synthesis tool.

<table>
<thead>
<tr>
<th>Construct Type</th>
<th>Keyword of Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ports</td>
<td>input, inout, output</td>
<td></td>
</tr>
<tr>
<td>parameters</td>
<td>parameter</td>
<td></td>
</tr>
<tr>
<td>module definition</td>
<td>module</td>
<td></td>
</tr>
<tr>
<td>signals &amp; variables</td>
<td>wire, reg, tri</td>
<td>Vectors are allowed</td>
</tr>
<tr>
<td>instantiation</td>
<td>module &amp; primitive instances</td>
<td>mymux m1(out, i0, i2, s); nand (out, a, b);</td>
</tr>
<tr>
<td>functions &amp; tasks</td>
<td>function, task</td>
<td>Timing constructs ignored</td>
</tr>
<tr>
<td>procedural</td>
<td>always, if, then, else, case(x/z)</td>
<td>initial is not supported</td>
</tr>
<tr>
<td>procedural blocks</td>
<td>begin, end, named blocks, disable</td>
<td>Disabling of named blocks allowed</td>
</tr>
<tr>
<td>data flow</td>
<td>assign</td>
<td>Delay information is ignored</td>
</tr>
<tr>
<td>loops</td>
<td>for, while, forever</td>
<td>while and forever loops must contain @(posedge clk) or @(negedge clk)</td>
</tr>
</tbody>
</table>

Introduction to Digital VLSI
Gil Rahav
Freescale Semiconductor Israel
Verilog HDL Synthesis: Think Hardware!!!

- Remember that we are providing a cycle-by-cycle RTL description of the circuit
- There are restrictions on the way these constructs are used for the logic synthesis tool. For example:
  - The while and forever loops must be broken by a @(posedge clock) or @(negedge clock) statement to enforce cycle-by-cycle behavior and to prevent combinational feedback
  - Logic synthesis ignores all timing delays specified by #<delay> construct. Therefore, pre- and post-synthesis Verilog simulation results may not match. The designer must use a description style that eliminates these mismatches
  - The initial construct is not supported by logic synthesis tools. Instead, the designer must use a reset mechanism to initialize the signals in the circuit
- ...
Verilog HDL Synthesis: Timing Delays

- Pre- and post-synthesis Verilog simulation results may not match
  - Logic synthesis ignores all timing delays specified by \#<delay> construct
  - In RTL simulation the outputs will not be updated on every input (in signal) change if changes happen more frequently than the delay in the logic (65 time units in the example)

- The designer must use a description style that eliminates these mismatches

```verilog
module code11 (out1, out2, in);
output out1, out2;
input in;
reg out1, out2;
always @(in)
begin
    #25 out1 = ~in;
    #40 out2 = ~in;
end
endmodule
```
Almost all operators in Verilog are allowed for logic synthesis

<table>
<thead>
<tr>
<th>Operators Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>* / + - %</td>
<td>Multiply, divide, add, subtract, modulus</td>
</tr>
<tr>
<td>Logical</td>
<td>! &amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&gt; &lt; &gt;= &lt;=</td>
<td>Greater than, Less than, Greater than or equal to, Less than or equal to</td>
</tr>
<tr>
<td>Equality</td>
<td>== !=</td>
<td>Equality, inequality</td>
</tr>
<tr>
<td>Bit-wise</td>
<td>~ &amp; ^ ~^ ~~</td>
<td>Bitwise negation, and, or, nor, xor, xnor</td>
</tr>
<tr>
<td>Reduction</td>
<td>&amp; ~&amp;</td>
<td>^ ^ ^ ~ ~</td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt; &gt;&gt;</td>
<td>Left shift, right shift</td>
</tr>
<tr>
<td>Concatenation</td>
<td>{}</td>
<td>Concatenation</td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
<td>Conditional</td>
</tr>
</tbody>
</table>

Only operators such as “===“ and “!==“ that are related to “x” and “z” are not allowed (equality with “x” and “z” does not have much meaning in logic synthesis)
**Verilog HDL Synthesis: Top-Down Design**

- **Requirements Analysis**
  - Model entire system architecturally, in Verilog or some other high-level language
- **System Partitioning**
  - Partition your design based on functionality or path length
- **Behavioral/Functional Specification**
  - Write a behavioral Verilog model for each partition as an executable bus-functional specification of the design
- **Behavioral/Functional Verification**
  - Write or generate the same models at the RTL level, using synthesizable constructs. Assemble and verify entire RTL system
- **Synthesis & Optimization**
  - Translate the functional models to gate-level netlists using synthesis and optimization tool
- **Gate Level Verification**
  - Mixed-level logic simulation allows you to verify the design at all levels
Design partitioning is another important factor for efficient logic synthesis.
- The way the designer partitions the design can greatly affect the output of the logic synthesis tool.

Various partitioning techniques can be used:
- Hierarchical partitioning
- Horizontal partitioning
- Vertical partitioning
- Parallelizing design structure
Hierarchical Design Partitioning

- Module statements create hierarchical design blocks (see part III of this course)

- Continuous assignments (assigns) and procedural blocks (always) do not create hierarchy

```vhdl
module ADR_BLK (...);
    DEC U1 (ADR, CLK, INST);
    OK U2 (ADR, CLK, AS, OK);
endmodule;
```
While writing the RTL, the designer has to decide if to keep the hierarchy of the design or to write a flat code.

- **Flattened synthesis optimization**
  - Can take longer to execute
  - Saves from calculating timing budgets between blocks (uses synthesis tools’ strengths in optimizing for timing)

- **Hierarchical synthesis optimization**
  - Speed up optimization times
  - Requires partitioning designs to optimize smaller blocks.
  - Requires management of timing budgets between blocks.
  - Simplify the synthesis process (incremental design updates, is easier to debug, multi-engineer teams)
  - Supports a mix of options for different modules
Horizontal Design Partitioning

- **Horizontal partitioning**: use bit slices to give the logic synthesis tool a smaller block to optimize
  - It reduces **complexity** of the problem and produces more **optimal** results for each block

- The downsize of **horizontal partitioning** is that global minima can often be different local minima
  - Each block is **optimized individually**, but there may be some **global redundancies** that the synthesis tool may not be able eliminate
Instead of directly designing a 16-bit ALU, design a 4-bit ALU and build the 16-bit ALU with four 4-bit ALUs.

Logic synthesis tool has to optimize only the 4-bit ALU, which is a smaller problem than optimizing the 16-bit ALU.
Vertical Design Partitioning

- **Vertical partitioning** implies that the functionality of the block is divided into smaller sub-modules.

- This is different from **horizontal partitioning**
  - In horizontal partitioning, all blocks do the same function.
  - In vertical partitioning, each block does a different function.

- For logic synthesis, it is important to create hierarchy by partitioning a large block into separate functional sub-blocks.
  - A design is best synthesized if levels of hierarchy are created and smaller blocks are synthesized individually.
  - Creating modules that contain a lot of functionality can cause logic synthesis to produce sub-optimal designs. Instead, **divide the functionality into smaller modules and instantiate those modules.**
The 4-bit ALU is a four-function ALU with functions add, subtract, shift right, and shift left.

Vertical partitioning of 4-bit ALU: each block is distinct in function.
In this technique we use more resources to produce faster design.

- We convert sequential operations into parallel operations by using more logic.

Contrast the “carry lookahead” adder (4 gate delays, more logic gates) with a “ripple carry” adder (9 gate delays, less logic gate).
Partitioning Rules for Synthesis

- No hierarchy in combinational paths
- No glue logic between blocks
- Register all outputs
- Separate designs with different goals
- Isolate state machines
- Maintain a reasonable block size
- Separate logic, pads, clocks and non-synthesizable structures
Partitioning Rules for Synthesis

- No hierarchy in combinational paths

- No glue logic between blocks

- Merge glue logic into the related combinational logic description of the lower-level architectural statements
Partitioning Rules for Synthesis

- Register all outputs

- Related combinational logic is grouped into the same block that contains the destination register for the combinational logic path

- Allows improved sequential mapping during optimization (no hierarchical boundaries between combinational and sequential logic)

- Simplifies the description of the timing interface
Partitioning Rules for Synthesis

- Separate designs with different goals

![Diagram showing critical paths and no critical paths]

- Optimization is limited because the designer cannot isolate parts of a block and optimize them solely for area or for speed
- Designer can now perform appropriate optimization techniques on each module
Design Constraints Specification

- **Design constraints** are as important as efficient HDL description in producing optimal design.

- Accurate specification of **timing**, **area**, **power**, and **environmental parameters**, such as **input drive strengths**, **output loads**, **input arrival times**, etc., are critical to produce a gate-level netlist that is optimal.

- A deviation from the correct **constraints** or omission of a constraint can lead to non-optimal designs.

- Careful attention must be given to specifying **design constrains**.
Verilog HDL Synthesis: Modeling Style

- There is a **modeling style** for each synthesis tool
  - It is possible to write Verilog descriptions for which there are no digital hardware
  - Synthesis results are sensitive to the input description

- **Goals of the modeling style**
  - Efficiency
  - Predictability
  - Synthesizability

- **Two basic guidelines for writing synthesizable Verilog descriptions:**
  - The gate-level simulation should match the functional (RTL) simulation
  - Sequential design should work independent of technology-specific propagation delays
Modeling Style Basics

- **Use meaningful names** for signals and variables
  - Names of signals and variables should be meaningful so that the code becomes self-commented and readable

- **Avoid mixing positive and negative edge-triggered flip-flops**
  - Mixing negative and positive edge-triggered flip-flops may introduce inverters and buffers into the clock tree

- **Be careful with multiple assignments to the same variable**
  - Multiple assignments to the same variable can cause undesired logic to be generated (the previous assignment might be ignored, and only the last assignment would be used)

// Two assignments to the same variable
always @(posedge clk) if(load1) q <= a1;
always @(posedge clk) if(load2) q <= a2;

- The synthesis tool infers two flip-flops with the outputs anded together to produce the q output
- The designer needs to be careful about such situation!!!
Multiply, divide, and modulo operators are very expensive to implement in terms of logic and area

- These arithmetic operators can be used to implement the desired functionality concisely and in a technology-independent manner
- On the other hand, designing custom blocks to do multiplication, division or modulo operation can take a longer time to design, and the module becomes more technology dependent

Use parentheses to optimize logic structure

```
// Translates to three adders in series
out = a + b + c+d;
/* Translates to two adders in parallel with one final adder to sum results*/
out = (a + b) + (c+d);
```

- The designer can control the final structure of logic by using parentheses to group logic
- Using parentheses also improves readability of the Verilog description
Modeling Style: Combinational Logic

- For logic to be **combinational**, the output must have only one possible value for any combination of inputs
  - There must be no timing or order dependencies
  - If the description meets this definition, it can be synthesized as a combinational logic

- There are three modeling styles that meet these requirements
  - A netlist structure of **combinational primitives** with no feedback loops
  - A **continuous assignment** statement with no feedback loops
  - A **procedural block** with an event sensitivity list consisting of all nodes to which assignments are made

- You can group the **combinational logic in a function**
  - This guarantees that the logic will be interpreted as **combinational**, eliminating the risk of generating latches in a data path
Combinational Logic

✓ **Method 1**: A netlist structure of combinational primitives with no feedback loops

```vhdl
// Method 1
or (or1, A, B);
or (or2, C, D);
and (OUT, or1, or2, E);
```

✓ **Method 2**: A continuous assignment statement with no feedback loop

```vhdl
// Method 2
assign OUT = E & (A | B) & (C | D);
```

```vhdl
module orand (OUT, A, B, C, D, E);
  input A, B, C, D, E;
  output OUT;
  // Use one of four methods
endmodule
```
Combinational Logic

Method 3: Since a function have no timing control, it is interpreted as combinational logic

```vhdl
// Method 3
function out;
input A, B, C, D, E;
output; 
Out = E & (A | B) & (C | D); 
endfunction
```

Method 4: Procedural blocks with complete event sensitivity list

```vhdl
// Method 4
reg OUT;
always @(A or B or C or D or E)
if (E) OUT = (A | B) & (C | D);
else OUT = 0;
```
If you desire combinational logic, specify all branches of a case statement, including the default branch.

If the assignments are not complete in all branches of the decision, the synthesizer adds latches to maintain the state of the circuit.
Functions in Synthesis

- **Functions always synthesize to combinational logic**

- A problem can occur when engineers make a mistake in the combinational function code and create simulation code that behaves like a latch.

```verilog
module code3a (o, a, nrst, en);
    output o;
    input a, nrst, en;
    reg o;
    always @(a or nrst or en)
        if (!nrst) o = 1'b0;
        else if (en) o = a;
    endmodule
```

```verilog
module code3b (o, a, nrst, en);
    output o;
    input a, nrst, en;
    reg o;
    always @(a or nrst or en)
        o = latch(a, nrst, en);
    function latch;
        input a, nrst, en;
        if (!nrst) latch = 1'b0;
        else if (en) o = a;
    endmodule
endmodule
```
Modeling Style: Efficient Comparison

- **Equality operators** are implemented more efficiently

  ```vhdl
  if (a <= b) @ (posedge clk) a = a + 1;
  if (a != b) @ (posedge clk) a = a + 1;
  ```

- **Avoid threshold comparison**, except where resources can be shared
  - In general, threshold comparisons are less efficiently implemented

  ```vhdl
  if (a <= b) do_1;
  if (a < b+1) do_1;
  ```

- **When there are few branches consisting of many consecutive cases**, using a “case value” is more efficient

  ```vhdl
  case (a)
  (1 < a) && (a < 11): do_A;
  (11 <= a) && (a < 18): do_B;
  (a == 20): do_C;
  default: do_D;
  endcase
  ```
**Modeling Style: Latched Logic**

- **Latched logic describes storage devices independent from clock**

  ![Latched Logic Diagram]

- **Method 1**: Using a simple feedback

  ```
  // Method 1
  wire [7:0] OUT;
  assign OUT = ENABLE ? IN : OUT;
  ```

- **Method 2**: Using an unspecified branch in `if` or `case` statement

  ```
  // Method 2
  reg [7:0] OUT;
  always @(ENABLE or IN);
  if (ENABLE) OUT = IN;
  // no else statement !!!
  ```

- **Be careful of unintentional latches!**
- **Try to avoid using latches => use FFs**
An always block, without all its conditions specified, leads to a latch.

In the example below, is a false case, the value of data must be held and the synthesizer must use a storage element.

```verilog
module latch (q, data, enable);
  input data, enable;
  output q;
  reg q;
  always @(enable or data)
    if (enable)
      q <= data;
endmodule
```

A flip-flop is inferred when the procedural block is entered into only on a single edge of the control signal (clk).

```verilog
module dffn (q, data, clk);
  input data, clk;
  output q;
  reg q;
  always @(negedge clk)
    q <= data;
endmodule
```
The output of a block is stored with a flip-flop when it is triggered by an edge-sensitive event control.

```
module mux_dff (A, B, C, D, OUT, SEL, CLK);
input [3:0] A, B, C, D;
output [3:0] OUT;
input [1:0] SEL;
input CLK;
reg [3:0] OUT;
always @(posedge CLK)
  case(SEL)
    2'b00: OUT <= A;
    2'b01: OUT <= B;
    2'b10: OUT <= C;
    2'b11: OUT <= D;
    // use default if not full case
  endcase
endmodule
```
Modeling style rules for sequential logic:

- Each always block can have only one edge of one clock (unless you are modeling asynchronous reset with the asynchronous/synchronous branch modeling style)
- Each stored variable may be assigned from only one clock-edge-triggered procedural block
The `reg` variables in a sequential block are implemented as hardware registers if they are assigned a value in one clock cycle and sampled in another.

If the sampling and assignment of the `reg` does not cross clock boundaries then the `reg` may be optimized away.

The `reg` variable does not necessarily imply a hardware register in the final synthesized output (`reg` variables do exist in RTL logic that is combinational).

If the `reg` variable is also a primary output, it will appear in the final netlist regardless of the type of logic produced.
Modeling Style: Register Examples

```verilog
two clock edges imply two storage elements and rega is not optimized away
```

### Blocking assignment.
Don’t ever use!

```verilog
module ex1reg (data, clk, out);
input data, clk;
output out;
reg out;
reg rega;
always @(posedge clk)
begin
    rega = data;
    out = rega;
end
endmodule
```

```verilog
module ex2reg (data, clk, out);
input data, clk;
output out;
reg out;
reg rega;
always @(posedge clk)
begin
    rega <= data;
    out <= rega;
end
endmodule
```

### One clock edge imply two storage elements and rega is optimized away

```verilog
module ex1reg (data, clk, out);
input data, clk;
output out;
reg out;
reg rega;
always @(posedge clk)
begin
    rega = data;
    out = rega;
end
endmodule
```
You can model reset for any type of edge or level sensitive storage device.

Asynchronous reset can be modeled in a single block, sensitive to the active clock edge and the active reset edge:

- There must be exactly one synchronous branch in the conditional statement.
- The default (else) branch is typically the synchronous one.
Modeling Style: Asynchronous Reset

- **Flip-flop**

```verilog
module dffsetclr (q, clk, reset, d);
input clk, d, reset;
output q;
reg q;
always @(posedge clk or posedge reset)
    begin
        if (reset)
            q <= 0;
        else
            q <= d;
    end
endmodule
```

- **Latch**

```verilog
module latch (q, enable, set, clr, d);
input enable, d, set, clr;
output q;
reg q;
always @(enable or set or clr or d)
    begin
        if (set)
            q <= 1;
        else if (clr)
            q <= 0;
        else if (enable)
            q <= d;
    end
endmodule
```
Modeling Style: Synchronous Reset

- Check the status of the reset signal at every clock edge
- If your target library does not contain a storage device with synchronous reset, the reset is implemented in the data path

```verilog
dmodule dffsetClr (q, clk, reset, d);
  input clk, d, reset;
  output q;
  reg q;
  always @(posedge clk)
    begin
      if (reset)
        q <= 0;
      else
        q <= d;
    end
endmodule
```
Complex operators are operations that can be recognized as high-level operations and mapped to existing cells in a vendor’s library directly:

\[ \text{out} = a \times b; \]

- Most tools know enough to map this to a multiplier
- This multiplier may exist in a special macro library that has components at a higher level of complexity than the regular cell library
- Macro libraries can include parts for design reuse such as FIFOs, adders, substractors, shift registers, counters, decoders, etc., of a various architectures
- The macro library may also contain user-defined blocks that are designed and synthesized by the user and intended or re-use
Resource sharing is the sharing of a group of logic by more than one section of RTL code

- Some synthesizers do resource sharing automatically
- You can control some resource sharing from within your RTL code
- You can force resource sharing by changing the coding style

```
always @(a or b or c or d) if (a)
    begin
        out = b + c;
    end
else
    begin
        out = b + d;
    end
```

```
temp = a ? c : d;
out = b + temp;
```
**Modeling Style: Sensitivity List**

- Inputs to a procedural block should be included in its sensitivity list
  - Some synthesis tools produce a warning when encountering an incomplete sensitivity list; others produce an error. The tools that produce a warning proceed with the assumption that you meant to have a complete list.

```verilog
/* In this example, a, b, and c are inputs to the block; a and b are conditions, and c is contained in the RHS of the procedural assignment d = c */
always @(a or b or c)
    begin
        if (a and b) d = c;
    end
```
As a result of incomplete sensitivity list (input b is not specified) the RTL and gate-level simulations will produce different results.

// In this example, a and b are inputs to the block and c is an output
always @( a ) // Incomplete sensitivity list
begin
  c = a || b;
end

---

**RTL** | **Synthesis results** | **Gate-Level**
---|---|---
A | | A
B | | B
C | | C
 Modeling Style: (Non-) Blocking Assignments

- **Use non-blocking assignments when**
  - modeling sequential logic
  - modeling latches
  - modeling both sequential and combinational logic within the same "always" block

- **Use blocking assignments when**
  - modeling combinational logic with an "always" block

- **General guidelines**
  - Do not mix blocking and non-blocking assignments in the same "always" block
  - Do not make assignments to the same variable from more than one "always" block
  - Use $strobe to display values that have been assigned using non-blocking assignments
  - Do not make assignments using #0 delays
Finite State Machines (FSM) types

- A **Mealy** machine has outputs that are a function of the present state registers and of the machine inputs.
- A **Moore** machine has output that are function of the present state only, the outputs are not directly dependent on the machine inputs.

It is always possible to model an FSM specification as either Mealy or Moore machine, the difference is the output timing:

- ✓ A Moore machine has output that settle directly after active clock edge and remain stable for the duration of the clock cycle.
- ✓ In Mealy machine changes on the input are seen a cycle earlier than in the Moore machine.
There are two distinct types of Finite State Machines (FSMs): explicit and implicit.

Implicit machines use multiple `always @(posedge clk)` statements to indicate state transitions:
- exist at a higher level of abstraction than explicit machines and in general are not synthesizable.
- In implicit FSMs, registers are created whenever data is written in one clock cycle and read in another.

Explicit machines use `case` statements to define each possible state explicitly:
- explicit machines are used in code meant for synthesis.

All FSMs must have a reset, and their state changes must be synchronous to one edge of a single clock.
Finite State Machine: Implicit vs. Explicit FSMs

**Implicit FSMs:**
- Do not need a state register
- Handle only linear state changes well
- Each state is separated by clock boundaries
- Are not handled by most synthesis tools

**Explicit FSMs:**
- Are clearer and more well-defined
- Can handle default conditions
- Handle complex (nonlinear) state changes
- You specify a state variable that define the state of the state machine
Implicit Style of FSM

From each `always` block that models sequential logic, synthesis extracts a single FSM.

If the `always` block has only one clock cycle (a degenerate FSM), it is implemented with combinational logic plus a register.

Registers are created whenever data is written in one clock cycle and read in another clock cycle for the stored variables.

If the FSM has more than one cycle, the synthesis tool generates control logic, including a state variable, and adds registers for the stored variables.
Explicit Style of FSM

- FSM can be described explicitly in a procedural block with a single clock edge and a case statement.
- A state variable that defines the state of the FSM must be specified.
- To change the current state, the value of the state variable must be changed synchronous to the clock edge.
- It is a good practice to specify a default action for conditions that normally do not occur.
- Assignments of the state variable and output signals to constant expressions are optimized efficiently.

```verilog
module exp (out, datain, clk, rst);
  output out;
  input clk, datain, rst;
  reg out, state; // state variable
  always @(posedge clk or posedge rst)
    if (rst) {state, out} <= 2'b00;
    else
      case (state) // case statement
        1'b0: begin
          out <= 1'b0;
          if (datain) state <= 1'b1;
          else state <= 1'b0;
        end
        1'b1: begin
          out <= datain;
          state <= 1'b0;
        end
        default: {state, out} = 2'b00; endcase
  endmodule
```
module exp (out, datain, clk, rst);
output out;
input clk, datain, rst;
reg out, state;  // state variable
always @(posedge clk or posedge rst)
  if (rst) {state, out} <= 2'b00;
  else
    case (state) // case statement
      1'b0: begin
        out <= 1'b0;
        if (datain) state <= 1'b1;
      end
      1'b1: begin
        out <= datain;
        state <= 1'b0;
      end
      default: {state, out} = 2'b00;
    endcase
endmodule

Explicit Style of FSM (con.)
Explicit Style of FSM (con.)

module exp (my_out, datain, clk, rst, my_in);
output my_out;
input clk, datain, rst;
input my_in;
reg out, state;  // state variable
wire my_out;
assign  my_out = out & my_in;
always @(posedge clk or posedge rst)
if (rst) {state,out} <= 2'b00; else
  case (state) // case statement
    1'b0: begin
      out <= 1'b0;
      if (datain) state <= 1'b1;
    end
    1'b1: begin
      out <= datain;
      state <= 1'b0;
    end
    default: {state, out} = 2'b00;
  endcase
endmodule
Logic synthesis is the process of converting a high-level description of the design into an optimized, gate-level representation, using the cells in the technology library.

Computer aided logic synthesis tools have greatly reduced the design cycle time and improved productivity. They allow designers to write technology-independent, high-level descriptions and produce technology-dependent, optimized, gate-level netlists. Both combinational and sequential RTL descriptions can be synthesized.

Logic synthesis tools accept high-level descriptions at the register transfer level (RTL). Thus, not all Verilog constructs are acceptable to a logic synthesis tool. We discussed the acceptable Verilog constructs and operators and their interpretation in terms of digital circuit elements.
A logic synthesis tool accepts an RTL description, design constraints, and technology library, and produces an optimized gate-level netlist. Translation, logic optimization, and technology mapping are the internal processes in a logic synthesis tool and are normally invisible to the user.

Proper Verilog coding techniques must be used to write efficient RTL descriptions, and various design trade-off must be evaluated. Guidelines for writing efficient RTL descriptions were discussed.

Design partitioning is an important technique used to break the design into smaller blocks. Smaller blocks reduce the complexity of optimization for the logic synthesis tool.

Accurate specification of design constraints is an important part of logic synthesis.