Writing Verilog HDL code for Synthesis

Logic synthesis is the process of converting a high-level description of design into an optimized gate-level representation.

Logic synthesis uses a standard cell library which have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adder, muxes, memory, and flip-flops.

Standard cells put together are called technology library. Normally the technology library is known by the transistor size (0.18u, 90nm).

What is logic synthesis?

A circuit description is written in Hardware Description Language (HDL) such as Verilog. The designer should first understand the architectural description.

Life before HDL (Logic synthesis)

High-level design is less prone to human error because designs are described at a higher level of abstraction.

High-level design is done without significant concern about design constraints.

Conversion from high-level design to gates is done by synthesis tools, using various algorithms to optimize the design as a whole.

This removes the problem with varied designer styles for the different blocks in the design and suboptimal designs.

Logic synthesis tools allow technology independent design. Design reuse is possible for technology-independent descriptions.
Constructs Not Supported in Synthesis

<table>
<thead>
<tr>
<th>Construct Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>Used only in test benches.</td>
</tr>
<tr>
<td>events</td>
<td>Events make more sense for syncing test bench components.</td>
</tr>
<tr>
<td>real</td>
<td>Real data type not supported.</td>
</tr>
<tr>
<td>time</td>
<td>Time data type not supported.</td>
</tr>
<tr>
<td>force and release</td>
<td>Force and release of data types not supported.</td>
</tr>
<tr>
<td>assign and deassign</td>
<td>Assign and deassign of reg data types is not supported. But assign on wire data type is supported.</td>
</tr>
<tr>
<td>fork join</td>
<td>Use nonblocking assignments to get same effect.</td>
</tr>
<tr>
<td>primitives</td>
<td>Only gate level primitives are supported.</td>
</tr>
<tr>
<td>table</td>
<td>UDP and tables are not supported.</td>
</tr>
</tbody>
</table>

PROCEDURAL CONTINUOUS ASSIGNMENT (PCA)

PURPOSE: (1) Model combinational logic (assign), (2) model asynchronous override of synchronous activity (assign ... deassign), and (3) support test signal insertion (force ... release).

EXAMPLE:

```verilog
module mux4_PCA (a, b, c, d, select, y_out);
  input a, b, c, d;
  input [1:0] select;
  output y_out;
  reg y_out;
  always @ (select)
    if (select == 0) assign y_out = a; else
    if (select == 1) assign y_out = b; else
    if (select == 2) assign y_out = c; else
    if (select == 3) assign y_out = d; else assign y_out = 1'bx;
endmodule
```

```
module Flop_PCA (preset, clear, q, qbar, clock, data);
  input preset, clear, clock, data;
  output q, qbar;
  reg q;
  assign qbar = ~q;
  always @(negedge clock)
    q = data;
  always @ (clear or preset)
    begin
      if (!clear) assign q = 0;
      else if (!preset) assign q = 1;
      else deassign q;
    end
endmodule
```

The binding created by a procedural continuous assignment remains effective until a subsequent binding overrides it, or until a "deassign" is executed.

```
force sig_a = 1;
force sig_b = 1;
force sig_c = 0;
sig_in1 = 0;
#5 sig_in1 = 1;
#5 sig_in1 = 0;
// Insert code to conduct tests
release sig_a;
release sig_b;
release sig_c;
```
Another common type of code, where one reg variable is driven from more than one always block. It will surely cause synthesis error.

Example of Non-Synthesizable Verilog construct.

Any code that contains the above constructs shown in slide 4 are not synthesizable, but within synthesizable constructs, bad coding could cause synthesis issues. Such as a flip-flop with both posedge of clock and negedge of clock in sensitivity list.

Delays

This code is useful only for simulation purpose.

Synthesis tool normally ignores such constructs, and just assumes that there is no #10 in above statement, thus treating above code as

\[ a = b; \]

Delays

\[ a = \#10 b; \]

Synthesis tool normally ignores such constructs, and just assumes that there is no \#10 in above statement, thus treating above code as

\[ a = b; \]

Multilexers

// 1. using an always
always@(a or b or sel)
if (sel == 1'h1)
c = a;
else
c = b;

// 2. using the ternary operator
wire c = sel ? a : b;
Priority Decoder using a case statement

```verbatim
// 3. using the case statement
always @(a or b or sel)
case (sel)
  1'b1: c = a;
  1'b0: c = b;
endcase
```

1. Both case and if statements result in priority structures.
2. The order of the variables determines the priority

Priority Decoder using an if/else statement

```verbatim
// 2. using an if statement
always @(sl or a or b or c)
if (sel == 2'b11)
  d = a;
else if (sel == 2'b10)
  d = b;
else
  d = c;
```
Priority Decoder using an if/else statement

```vhdl
// using a synthesis directive
always @ (a1 or a or b or c)
case (sel) // parallel_case
  2'b11: d = a;
  2'b10: d = b;
default: d = c;
endcase
```

Bus Enabling

```vhdl
// A1. using a wire
wire [3:0] d = ({enable} & c);
// A2. using a reg
reg [3:0] d;
always @ (c or enable)
  d = c & {enable};
```

Parallel Priority Decoders Using a Synthesis Directive

```vhdl
// B1. using a wire
wire [2:0] e = [a[1],b[3:2]];
// B2. using a reg
reg [2:0] e;
always @ (a or b)
  e = [a[1],b[3:2]];
```

Bus Concatenation

```vhdl
a[3]
a[2]
a[1] e[2]
a[0] e[1]
b[3] e[0]
b[2]
b[1] b[0]
```
Bus Replication

```verilog
// bus replication
wire [1:0] a;
wire [3:0] b;
assign b = {2{a}};
```

Comparators

```verilog
// 1. using a wire
wire d;
assign d = (a == c);

// 2. using a reg
reg d;
always @(a or c)
d = (a == c);
```

D Type Flip Flops

```verilog
// 1. positive edge triggered D flip flop
always @ (posedge clock)
q <= d;
```

Use non-blocking assignments (<=) in clocked procedures.

```verilog
// 2. negative edge triggered D flip flop
always @ (negedge clock)
q <= d;
```
Resettable D Type Flip Flops

// 1. synchronously resettable D flip flop
always @ (posedge clock)
if (reset)
    q <= 1'b0;
else
    q <= d;

// 2. asynchronously resettable D flip flop
// (active high async reset)
always @ (posedge clock orposedge reset)
if (reset)
    q <= 1'b0;
else
    q <= d;

Resettable D Type Flip Flops

// 3. asynchronously resettable D flip flop
// (active low reset)
always @ (posedge clock or negedge reset)
if (~reset)
    q <= 1'b0;
else
    q <= d;

Data Enabled and Clock Gated Flip Flops

// 1. data enabled flip flop
always @ (posedge clock)
if (enable)
    q <= d;
Data Enabled and Clock Gated Flip Flops

```verilog
// 2. D flip flop with gated clock
wire gclk = (clock && enable);
always @ (posedge gclk)
  q <= d;
```

Enable signal must be glitch free.

---

Tri-state Drivers

```verilog
// 1. using a reg
reg y;
always @ (d or enable)
  if (enable)
    y = d;
  else
    y = 1'bz;

// 2. using a wire
wire y;
assign y = enable ? d : 1'bz;

// 3. using a primitive
buf1 (y,d,enable);
```

---

Counter

3 bit asynchronously resettable counter which counts 0, 1, 2, 3, 4, 5.

```verilog
// 3 bit asynchronously resettable
// partial range counter
always @ (posedge clock or posedge reset)
  if (reset)
    count <= 3'b0;
  else
    if (count == 3'b101)
      count <= 3'b0;
    else
      count <= count + 3'b001;
```

---
**Counter**

3 bit asynchronously resettable counter which counts 0, 1, 2, 3, 4, 5,

```verilog
module counter (clock, reset, data_in, data_out);
input clock;
input reset;
input [3:0] data_in;
output [3:0] data_out;
reg [3:0] data_out;
reg [3:0] shift_reg_1;
reg [3:0] shift_reg_2;
reg [3:0] shift_reg_3;

always @ (posedge clock)
if (reset)
begin
    shift_reg_1 <= data_in;
    shift_reg_2 <= shift_reg_1;
    shift_reg_3 <= shift_reg_2;
    data_out <= shift_reg_3;
end

endmodule
```

---

**Enabled Shift Register**

```verilog
module enabled_shift_reg (clock, enable, data_in, data_out);
input clock;
input enable;
input [3:0] data_in;
output [3:0] data_out;
reg [3:0] data_out;
reg [3:0] shift_reg_1;
reg [3:0] shift_reg_2;
reg [3:0] shift_reg_3;

always @ (posedge clock)
if (enable)
begin
    shift_reg_1 <= data_in;
    shift_reg_2 <= shift_reg_1;
    shift_reg_3 <= shift_reg_2;
    data_out <= shift_reg_3;
end

endmodule
```

---

**Unsigned Adders and Multipliers**

Note that the * and + and - signs give you unsigned arithmetic.

```verilog
wire [5:0] c = a + b;
wire [11:0] e = c * d;
```
Coding Guidelines

• Use non-blocking assignments (<=) in clocked procedures.
  
  always @ (posedge clock)
  q <= d;

• Use blocking assignments (=) in combinational procedures.
  
  always @ (a or b or sl)
  if (sl)
    d = a;
  else
    d = b;

• Make sure that the event lists are complete.
  
  always @ (a or b) // this event list is missing signal sl
  if (sl)
    d = a;
  else
    d = b;

• Comment code properly.
  
  // example of bad comments
  // add a and b together
  always @ (a or b)
    c = a + b;

  // Good commenting
  // 8 bit unsigned adder for data signals 'a' and 'b'
  // output is sent to UART2
  always @ (a or b)
    c = a + b;

• Always completely specify literals.
  
  always @ (c)
  if (c == 4' b1010)
    a = 2' bxx;
  else
    a = 2' b10;

• Use named port mapping when instantiating.

  state_machine ul (  
    .sm_in   (in1),
    .sm_clock (clk),
    .reset   (reset),
    .sm_out  (data_mux)  
  );

  see another example on next slide

• Don’t make the code any more complicated than it needs to be.

  module proj2TB (  
    rst,  
    start,  
    inputA, //multiplier  
    inputB, //multiplicand  
    load_multi,  
    result,  
    done,  
    count
  );

  module proj2SYS ();

  //Project 2 Testbench
  //John Schell
  //ECE5242

  module proj2SYS ();

  //Project System
  //John Schell
  //ECE5242

  module proj2SYS ();

  wire [7:0] inputA, inputB;
  wire [15:0] result;
  wire [16:0] product;
  wire [7:0] mcand_in;
  wire [3:0] count;

  assign result = product[16:1];
  //don't need booths bit

  proj2TB TB(  
    .rst(rst),
    .start(start),
    .inputA(inputA), //multiplier  
    .inputB(inputB), //multiplicand  
    .load_multi(load_multi),
    .result(result),
    .done(dones),
    .count(count);  
  );
Write a 100% synthesizable Verilog code for this state machine.

```verilog
module state_machine (clock, reset, start, transmit, wait, stop, ack, offline, online);

    // parameter declarations
    parameter pIDLE = 2'b10; // state labels and state encoding
    parameter pRUN = 2'b01;
    parameter pPAUSE = 2'b00;
    parameter pFINISH = 2'b11;

    // IO declaration section
    input clock;
    input reset;
    input start, transmit, wait, stop;
    output ack, offline, online;

    // internal variables declaration section
    reg [1:0] state, next_state;
    reg ack, offline, online;

    // clocked procedure with synchronous reset
    always @ (posedge clock)
    if (reset) // reset strategy
        state <= pIDLE;
    else
        state <= next_state;

    // combinational procedure with case statement and output logic
    always @ (start or transmit or stop or wait or state)
    begin
        next_state = state; // default assignment to state and output variables
        ack = 1'b0;
        offline = 1'b0;
        online = 1'b0;
        case (state)
            pIDLE:
                begin
                    offline = 1'b1;
                    online = 1'b0;
                    if (start)
                        next_state = pRUN;
                    end
            pRUN:
                begin
                    if (wait)
                        next_state = pPAUSE;
                    if (stop)
                        next_state = pFINISH; // this has priority over the wait transition
                    end
            pPAUSE:
                begin
                    ack = 1'b1;
                    if (transmit)
                        next_state = pRUN;
                    if (stop)
                        next_state = pFINISH;
                    end
            pFINISH:
                begin
                    ack = 1'b0;
                    endstate
                end
        endcase
    end
endmodule
```