Lecture 21 -- Synthesis 1

Outline

- Introduction to Synthesis -- Concepts
- Writing Synthesizable Verilog
- Synthesizable State Machines

What is Synthesis?

- Last time, we defined synthesis tools as any tools that transforms a design from one format to another
- Most common use of “synthesis” describes tools that transform designs from hardware description languages (VHDL/Verilog) into designs that can be fabricated
- This is commonly done through the use of a standard cell library, which can be viewed as one input to the synthesis tool
- Tool takes design, standard cell library, and generates an implementation of the design using only the cells in the library

Synthesis

Two-step process

1. Synthesis tool takes HDL description of design and standard-cell library, generates *netlist* of standard cells and connections between them
   - Can provide *timing* requirements to tool as well, and it will try to generate a netlist that meets those requirements

2. Place and route tool finds locations for each standard cell in the netlist, and creates physical wires to implement the connections

Writing Synthesizable Verilog/VHDL

- Verilog is a very powerful language, even containing some of the object-oriented features found in C++/Java
- However, synthesis tools will only do a good job on a somewhat-structural subset of the language
- Golden rule: If you can't envision the hardware behind your Verilog, the synthesizer won't be able to either
- Tips:
  - Synthesizer ignores delays. Your design needs to be race-free and count only on latches for timing
  - Synthesizer won't implement initial blocks. You need to provide a reset mechanism
  - Define signal widths explicitly, particularly in assignments

Verilog Tips 2

- RTL specification should be as close as possible to desired structure
- Meaningful names should be used for signals and variables
- Mixing of positive and negative edge-triggered flip-flops should be avoided
  - ◇ Result in buffers and inverters in clock tree, which contribute to clock skew
- Avoid multiple assignments to the same variable
- Define `if-else` and `case` statements explicitly
  - ◇ Latches may be inferred if all conditions are not specified.

Translation of Verilog Constructs

- `assign` statements are translated to equivalent combinational logic
  - ◇ assign out = (a&b)|c;
  - ◇ translated to a 2-input and gate feeding a 2-input or gate
- `if-else` statement translated to a mux
- `case` statement translated to a mux
- `for` loops used to build cascaded combinational logic

```
c = c_in;
for (i = 0; i <= 7; i = i + 1)
   {c, sum[i]} = a[i] + b[i] + c; // 8-bit ripple adder
c_out = c;
```

Translation of Verilog Constructs

- `always` statement used to infer sequential and combinational logic
  - ◇ Positive edge-triggered flip-flop inferred

```
always @(posedge clk)
   q = d;
```

  - ◇ Level-sensitive latch inferred

```
always @(clk or d)
   if (clk)
      q = d;
```

  - ◇ Combinational logic inferred

```
always @(a or b or c_in)
   {c_out,sum} = a + b + c_in;
```

Translation of Verilog Constructs

- Functions synthesize to combinational blocks

```
function [4:0] fulladd;
input [3:0] a, b;
input c_in;
   begin
      fulladd = a + b + c_in;
   end
endfunction
```

Example of Inferred Latches

```
module wrong (out, in1, in2);
   output out;
   reg    out;
   input  in1, in2;

   always @(in1 or in2)
     case ({in1,in2})
       2'b00 : out = 0;
       2'b01 : out = 1;
       2'b10 : out = 1;

     endcase
endmodule
```

```
module right (out, in1, in2);
   output out;
   reg    out;
   input  in1, in2;

   always @(in1 or in2)
     case ({in1,in2})
       2'b00 : out=0;
       2'b01 : out=1;
       2'b10 : out=1;
       default: out=x;
     endcase
endmodule
```

- Module *wrong* infers one latch. Why?

Optimization

- Like compiling programs, generating circuitry that implements a given line of HDL code is relatively straightforward
  - Most of the work lies in making the generated circuitry fast/small
- Once the synthesizer has generated an initial netlist for a design, it applies optimizations
  - Logic minimizations
  - Transformations of multi-input gates into few-input gates and vice versa
  - Buffer insertion
  - Logic re-ordering for timing

Finite State Machines

- Two classes of finite state machine:
  - Moore machine: outputs depend only on current state, next state depends on current state and inputs
  - Mealy machine: outputs and next state depend on both current state and inputs
- Theory result: Mealy and Moore machines are *equivalent*, meaning that you can implement any desired FSM with either a Mealy or Moore machine
  - Equivalency ignores timing effects
  - Moore machines generally have more states than an equivalent Mealy machine

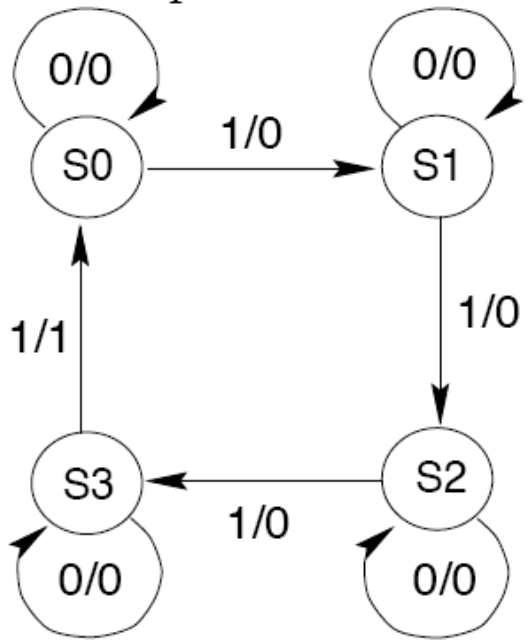
Implementing Finite State Machines

- I generally recommend the use of Moore-type machines
  - More states, but
  - Less timing worries --
    - outputs only change in response to clock edges
    - timing problems in the FSM can be solved by making clock cycle longer
    - No potential for oscillations

- Often useful to break an FSM down into three pieces
  - State vector: latches that hold the current state
  - Logic that computes the next state
  - Logic that computes the outputs of the FSM

Mealy Machine Example

- Consider the following Mealy machine (input/output):
  - ◇ Mealy machine: *output* depends on *present state* and *input*



Mealy Machine Example

- Module header and synchronous *present_state* logic:

```
module fsm (clk, reset, in, out);
   input  clk, reset, in;
   output out;
   reg out;
   reg[1:0] present_state, next_state;
   parameter[1:0] S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;

   always @(posedge clk)
      begin
         if (reset == 1) present_state = S0;
         else present_state = next_state;
      end
```

Mealy Machine Example

- Continued: *next_state* logic

```
always @(present_state or in)
  begin
    case(present_state)
      S0: if (in == 1) next_state = S1;
          else next_state = S0;
      S1: if (in == 1) next_state = S2;
          else next_state = S1;
      S2: if (in == 1) next_state = S3;
          else next_state = S2;
      S3: if (in == 1) next_state = S0;
          else next_state = S3;
      default: next_state = S0;
    endcase
  end
```

Mealy Machine Example

- Continued: output decoding

```
  always @(present_state or next_state)
    begin
      if ((present_state == S3) && (next_state == S0))
         out=1;
      else out=0;
    end
endmodule
```

Providing Hints to the Synthesis Tool

- Select the state encoding scheme (i.e., enumerate states)

```
module fsm_enum (clk, reset, in, out);
  input  clk, reset, in;
  output out;
  reg out;
  reg[1:0] /* synopsys enum STATE_TYPE */ present_state;
  reg[1:0] /* synopsys enum STATE_TYPE */ next_state;
  parameter[1:0] // synopsys enum STATE_TYPE
          S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;

  //synopsys state_vector present_state
  always @(posedge clk)
    begin
      if (reset == 1) present_state = S0;
      else present_state = next_state;
   ...
```