Announcements

- Homework #1 due today
  - Drop box in 240 Cory

- Homework #2 out tonight
  - Due next Thurs.
Example: Variable Counter

Parameterized Version
Generate Loop
Permits variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop.

```
// Gray-code to binary-code converter
module gray2bin1 (bin, gray);
    parameter SIZE = 8;
    output [SIZE-1:0] bin;
    input [SIZE-1:0] gray;
    genvar i;
    generate for (i=0; i<SIZE; i=i+1)
    begin
        bit assign bin[i] = ^gray[SIZE-1:i];
    end endgenerate
endmodule
```

Generate if-else-if based on an expression that is deterministic at the time the design is synthesized.

Generate case: selecting case expression must be deterministic at the time the design is synthesized.

Logic Synthesis
Hierarchically define structure and/or behavior of circuit.

HDL Specification

Simulation
Functional verification.

Synthesis
Maps specification to resources of implementation platform (FPGA for us).
Logic Synthesis

• Verilog and VHDL started out as simulation languages, but quickly people wrote programs to automatically convert Verilog code into low-level circuit descriptions (netlists).

Verilog HDL → Synthesis Tool → circuit netlist

• Synthesis converts Verilog (or other HDL) descriptions to implementation technology specific primitives:
  - For FPGAs: LUTs, flip-flops, and RAM blocks
  - For ASICs: standard cell gate and flip-flop libraries, and memory blocks.

Why Logic Synthesis?
1. Automatically manages many details of the design process:
   → Fewer bugs
   → Improved productivity
2. Abstracts the design data (HDL description) from any particular implementation technology.
   - Designs can be re-synthesized targeting different chip technologies. Ex: first implement in FPGA then later in ASIC.
3. In some cases, leads to a more optimal design than could be achieved by manual means [ex: logic optimization]

Why Not Logic Synthesis?
1. May lead to non-optimal designs in some cases.
Main Logic Synthesis Steps

**Parsing and Syntax Check**
Load in HDL file, run macro preprocessor for `define, `include, etc..

**Design Elaboration**
Compute parameter expressions, process generates, create instances, connect ports.

**Inference and Library Substitution**
Recognize and insert special blocks (memory, flip-flops, arithmetic structures, ...)

**Logic Expansion**
Expand combinational logic to primitive Boolean representation.

**Logic Optimization**
Apply Boolean algebra and heuristics to simplify and optimize under constraints.

**Partition, Place & Route**
CL to LUTs, map memory and state elements to chip, assign physical locations, route connections.

Operators and Synthesis

- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, ...
  - Unsigned 2s complement
  - Model carry: target is one-bit wider than source
  - Watch out for *, %, and /
- Relational operators generate comparators
- Shifts by constant amount are just wire connections
  - No logic involved
- Variable shift amounts a whole different story --- shifter
- Conditional expression generates logic or MUX
Simple Example

```verilog
module foo (A, B, s0, s1, F);
  input [3:0] A;
  input [3:0] B;
  input s0, s1;
  output [3:0] F;
  reg F;
  always @ (*)
    if (!s0 & s1 || s0) F=A; else F=B;
endmodule
```

Some More Interesting Examples

```verilog
module mux4to1 (out, a, b, c, d, sel);
  output out;
  input a, b, c, d;
  input [1:0] sel;
  reg out;
  always @(sel or a or b or c or d)
    begin
      case (sel)
        2'd0: out = a;
        2'd1: out = b;
        2'd3: out = d;
        endcase
    end
endmodule
```
Fix (Rule #1 for CL Always Blocks)

To avoid synthesizing a latch in this case, add the missing select line:

\[ 2'd2: \text{out} = c; \]

Or, in general, use the “default” case:

\[ \text{default: out} = \text{foo}; \]

If you don’t care about the assignment in a case (for instance you know that it will never come up) then you can assign the value “x” to the variable. Example:

\[ \text{default: out} = 1'bx; \]

The x is treated as a “don’t care” for synthesis and will simplify the logic.

Be careful when assigning x (don’t care). If this case were to come up, then the synthesized circuit and simulation may differ.

Another Example

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

    always @(in1) begin
        out = in1 & in2;
    end
endmodule
```
Incomplete Triggers

Leaving out an input trigger usually results in latch generation for the missing trigger.

```
module and_gate (out, in1, in2);
    input in1, in2;
    output out;
    reg out;
    always @(in1) begin
        out = in1 & in2;
    end
endmodule
```

Easy way to avoid incomplete triggers for combinational logic is with: `always @*`

Procedural Assignments

Verilog has two types of assignments within `always` blocks:

- **Blocking** procedural assignment “=”
  - In simulation the RHS is executed and the assignment is completed before the next statement is executed.

- **Non-blocking** procedural assignment “<=”
  - In simulation the RHS is executed and all assignment take place at the same time (end of the current time step - not clock cycle).
**Synthesized Procedural Assignments**

\[
\text{always @ (posedge clk) begin}
\begin{align*}
a &= \text{in}; \\
b &= a;
\end{align*}
\text{end}
\]

\[
\text{always @ (posedge clk) begin}
\begin{align*}
a &= \text{in}; \\
b &= a;
\end{align*}
\text{end}
\]

**Procedural Assignments**

The sequential semantics of the blocking assignment allows variables to be multiply assigned within a single always block. Unexpected behavior can result from mixing these assignments in a single block. Standard rules:

i. Use blocking assignments to model combinational logic within an always block ("=").

ii. Use non-blocking assignments to implement sequential logic ("<=").

iii. Do not mix blocking and non-blocking assignments in the same always block.

iv. Do not make assignments to the same variable from more than one always block.
FSM CL block rewritten

```verilog
always @(*) begin
    next_state = IDLE;
    out = 1'b0;
    case (state)
        IDLE : begin
            out = 1'b0;
            if (in == 1'b1) next_state = S0;
            else next_state = IDLE;
        end
        S0 : begin
            out = 1'b0;
            if (in == 1'b1) next_state = S1;
            else next_state = IDLE;
        end
        S1 : begin
            out = 1'b1;
            if (in == 1'b1) next_state = S1;
        end
        default: begin
            next_state = IDLE;
            out = 1'b0;
        end
    endcase
endmodule
```

Nominal values: used unless specified below.
Within case only need to specify exceptions to the nominal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.

Encoder Example

Nested IF-ELSE might lead to "priority logic"
Example: 4-to-2 encoder

```verilog
always @(x)
begin : encode
    if (x == 4'b0001) y = 2'b00;
    else if (x == 4'b0010) y = 2'b01;
    else if (x == 4'b0100) y = 2'b10;
    else if (x == 4'b1000) y = 2'b11;
    else y = 2'bxx;
end
```

This style of cascaded logic may adversely affect the performance of the circuit.
Encoder Example (cont.)

To avoid "priority logic" use the case construct:

```verbatim
always @(x)
  begin : encode
    case (x)
      4'b0001: y = 2'b00;
      4'b0010: y = 2'b01;
      4'b0100: y = 2'b10;
      4'b1000: y = 2'b11;
      default: y = 2'bxx;
    endcase
  end
end
```

All cases are matched in parallel.

Encoder Example (cont.)

A similar simplification would be applied to the if-else version also.

This circuit would be simplified during synthesis to take advantage of constant values as follows and other Boolean equalities: