Synthesizeable Code with Loops

• Today we'll discuss a few constructs which involve a concern of **Control** Loops and **Data** (dependency) Loops

• We'll first concern ourselves with combinatorial behavior and then allow additional flexibility with sequential
Think of each statement as a node on a graph with the edges denoting dependencies. Nodes can be producers and consumers of values. A graph with loops cannot be directly resolved as a combinatorial circuit.

The inputs not generated from within the code are also nodes – they represent an assignment elsewhere.

\[
\begin{align*}
y_1 &= a + b; \\
y_2 &= y_1 + c; \\
y_3 &= y_1 + y_2 \times y_2;
\end{align*}
\]
Branches can be thought of as multiplexors that depend on the result of a condition. A new variable based on the condition evaluation may be introduced to make this clear.

\[
y_1 = a + b; \\
\text{if (}y_1 < 0\text{)} \\
y_2 = 0; \\
\text{else} \\
y_2 = y_1;
\]

\[
y_1 = a + b; \\
\text{flag} = (y_1 < 0); \\
y_2 = \text{flag?0:y1};
\]
To achieve the status of single assignment code, every variable may only be assigned once.

We may need to convert code to an equivalent single-assignment code to understand its underlying structure. To do this introduce additional variables when variables are assignment more than once.

\[
\begin{align*}
y &= a + b; \\
s &= y + c; \\
y &= y + s \cdot s;
\end{align*}
\]

\[
\begin{align*}
y_1 &= a + b; \\
s &= y_1 + c; \\
y_2 &= y_1 + s \cdot s;
\end{align*}
\]
Synthesis: Feedback (data dependency loops)

```verilog
always @ (a,b) begin
    y = 0;
    y = y&a;
    y = y&b;
end

always @ (a,y) begin
    y = ~(y&a);
end

always @ (a,b,yA,yB) begin
    yA = ~(yB&a);
    yB = ~(yA&b);
end

always @ (posedge clk) begin
    y = ~(y&a);
end
```

No feedback after substitutions

Feedback

Feedback

This clearly does not attempt to describe combinatorial hardware it is edge-triggered describing sequential hardware.
Registered logic (mix comb and seq.) should be separated to understand the dependencies. New variables may be introduced to denote the difference in signals before and after a register.

```verilog
always @(posedge clk) begin
  if (counter == CNT_MAX)
    counter <= 0;
  else
    counter <= counter + 1;
end
```

Feedback across clock cycles is OK.

```verilog
always @(posedge clk) begin
  counter <= counter_comb;
end

always @ (*) begin
  flag = counter == CNT_MAX;
  counter_comb = flag?0:counter+1;
end
```

No feedback in comb. part.
Synthesis: Non-blocking statements

- Not part of original Verilog, added to accommodate RTL code
- RHS is evaluated and result is schedule to be assigned to LHS at end of timestep, eliminating race conditions
  - Multiple scheduled assignments to same variable will cause only the last one to be used
  - Resulting behavior is synthesized
always @ (posedge clk) begin
  dir <= a > b;
  if (dir)
    counter <= counter + 1;
  else
    counter <= counter - 1;
end
Synthesis: Loops

• Often you can't describe a combinatorial circuit with non-static loops. Below the condition that is checked before every iteration is dependent on assignments within the body of the loop.

• Furthermore, the multiple data movements are problematic

```c
//”while loop”
temp=datain;
count =0;
for (index=0; |temp; index=index+1)
begin
  if temp[0]==1 (count=count +1)
temp>>1;
end
```
Synthesis: Loops

• Should rewrite to have static loop count and no implied data movement:

```plaintext
//”while loop”
count = 0;
for (index = 0; index < 8; index = index + 1) begin
  if temp[index] == 1 (count = count + 1);
end
```
• Note you may be able describe a **sequential** circuit with non-static loops

```plaintext
//"while loop" with iterations sync. to clock
count = 0;
for(index=0; |temp; index=index+1) begin
  @(posedge clk);
  if temp[0] == 1 (count=count + 1)
  temp>>=1;
end
```
The keyword **disable** may be used to implement a “break” from a loop. Consider this not yet covered and avoid for now.
Synthesis: Registers and Pipeling

For edge-triggered begin-end blocks, values produced from blocking statements with consumers outside the block or in subsequent executions of the block are going to be implicitly implement registers, creating a registered and non-registered version of the signal.

Our required practice is to code the producer instead using non-blocking assignments.

Values from blocking statements consumed only within the same execution pass of the procedural block are ok and are implemented as combinatorial signals.
Code to compare input value with its previous value:

```verilog
always @ (posedge clk) begin
    prev_bit = current_bit;
    current_bit = input_bit;
    match = current_bit ^ prev_bit;
end
```

- prev_bit not a register,
- current bit is a register
- match is a register* (*assuming it is used elsewhere)
← Violates our coding practice

```verilog
always @ (posedge clk) begin
    current_bit = input_bit;
    prev_bit = current_bit;
    match = current_bit ^ prev_bit;
end
```

- prev_bit not a register,
- current bit is not a register
- match is a register*
- Incorrect behavior

```verilog
always @ (posedge clk) begin
    prev_bit <= current_bit;
    current_bit = input_bit;
    match <= current_bit ^ prev_bit;
end
```

- prev_bit is a register,
- current bit is not register
- match is a register* and depends on prev_bit from previous trigger
Thought question, if you didn't what match to be a register what would you have to change?
Typical to employ multi-cycle operations to reduce hardware through resource sharing (reuse of hardware in different clock cycles) and reduce the critical path lengths.

```verilog
always @ (posedge clk)
begin
    temp=a*b;
    @(posedge clk)
    y=temp*c;
end
```

Another in class example:
For Generate

- Uses a special indexing variable. Use for repetitive instantiations

```verilog
genvar index;
generate
for (index=0; index < 8; index=index+1)
    begin: gen_code_label
        BUFR BUFR_inst (  
            .O(clk_o(index)), // Clock buffer output  
            .CE(ce), // Clock enable input  
            .CLR(clear), // Clock buffer reset input  
            .I(clk_i(index)) // Clock buffer input  
        );
    end
endgenerate
```

- In class example adder
For Generate

- Uses a special indexing variable. Use for repetitive instantiations

```verbatim
genvar index;
generate
for (index=0; index < 8; index=index+1)
    begin: gen_code_label
        adder adder_inst (
            .cin(c[index]),
            .a(a[index]),
            .b(b[index]),
            .cout(c[index+1]),
            .y(u[index])
        );
    end
endgenerate
```
Concluding Points

- Combinatorial dependency loops cannot be synthesized
- Static for loops can be synthesized by being unrolled
- Dynamic for loops may not be synthesizable
- Dynamic for loops with timing control may be synthesized as a “multicycle operation” or a state machine.
- We'll want to formalize multi-cycle operations as state machines.