VERILOG: FLIP-FLOPS AND REGISTERS
Instantiating Flip-Flops/Registers

• One way to build a FF/register (do not use this)
  reg a;
  always @(posedge clk) begin
    a = a_c;
  end

• The “=” is a “blocking assignment” which causes
  the simulator to “block” on an assignment until
  the operation is completed, then it moves to the
  next statement

• It makes a race condition possible
  reg b, c;
  always @(posedge clk) begin
    b = a;
    c = b;
  end
  – In this case, a races to c in one cycle!
Instantiating Flip-Flops/ Registers

- One solution would be to simply reorder the reg assignment commands to eliminate the race

- *a* races to *c* in one cycle
  ```
  reg b, c;
  always @(posedge clk) begin
      b = a;
      c = b;
  end
  ```

- *a* progresses to *b* and *b* to *c* in one cycle each as expected
  ```
  reg b, c;
  always @(posedge clk) begin
      c = b;
      b = a;
  end
  ```

- Of course this is a terrible solution and you should never use it. Besides being very tedious and prone to errors, it will not work with loops.
Instantiating Flip-Flops/Registers

- The best solution is to use a “non-blocking assignment” written with a “<=” which causes the simulator to schedule all assignments at a particular point in time and perform them all simultaneously.
- With the verilog below, the registers perform as normal FFs behave without a race.

```verilog
reg b, c;
always @(posedge clk) begin
    b <= a;
    c <= b;
end
```

![Verilog Code Diagram]
Verilog Register Assignments

• Rule #1 (always follow in this class):
  For **combinational logic always blocks**, always use **blocking assignments** ("=")

  // OR gate
  always @(a or b) begin
    c = a | b;
  end
Verilog Register Assignments

• Rule #2 (always follow in this class):
  For flip-flop (register) *always blocks*, always use *non-blocking*
  assignments ("<=")

  ```verilog
  always @(posedge clk) begin
    sum <= #1 c_sum;
    r_product <= #1 product;
  end
  ```

• Also add "#1" to give one unit of clock-to-Q delay to increase
  waveform readability
  - This does, however, produce a warning during synthesis,
    but it can be ignored (the only warning that can be automatically ignored!)
• Rule #3 (really a guideline):
  Normally do not include any logic in flip-flop declarations even if you are tempted to include functions such as resets, initializations, or counter incrementing. Try to resist but it is ok occasionally.

    // Flip-flop declarations
    always @(posedge clock) begin
        state    <= #1 state_c;
        count    <= #1 count_c;
        if (reset == 1'b1) begin
            state <= #1 4'b0000;
        end
    end
Verilog Register Assignments

• Rule #4 (really a suggestion): Normally group all related register declarations together in one *always* block. Do this only for the purpose of increasing readability.

```verilog
// Flip-flop declarations
always @(posedge clock) begin
    state    <= #1 state_c;
    count    <= #1 count_c;
    data_r   <= #1 data;
    cat      <= #1 mouse;
end
```
Signal Naming Conventions

- It is helpful to have conventions for signal names
  - Easier for others to understand your code
  - Easier for YOU to understand your code
- Add a suffix to signal names to indicate they are from an earlier pipeline stage or a later pipeline stage
- \_*c – input to a register \ (e.g., \textit{sum\_c})
- \_*r – output of a register \ (e.g., \textit{sum\_r})

\begin{center}
\begin{tikzpicture}
\node at (0,0) (sum_c) {sum\_c};
\node at (2,0) (sum) {sum};
\draw (sum_c) -- (sum);
\node at (0,-1) (sum) {sum};
\node at (2,-1) (sum_r) {sum\_r};
\draw (sum) -- (sum_r);
\end{tikzpicture}
\end{center}
Another possibility I have seen is to make it a prefix but this has the possibly-negative feature that associated signals are not adjacent when sorted alphabetically

- $c_*$ – input to a register (e.g., $c_{\text{sum}}$)
- $r_*$ – output of a register (e.g., $r_{\text{sum}}$)
If a signal is pipelined across multiple pipe stages, it is probably a good idea to indicate that in the signal’s name—for example with a suffix such as \_pipeX.

\textit{inA} – signal in pipeline stage 1