FLIP-FLOPS AND REGISTERS
Semiconductor Memories

• “Single-bit” Memory (Foreground)
  – Individual memory circuits that store a single bit of information and have at least the following I/O ports:
    1) data input and 2) data output
  1. “Clock-less” latch
  2. Transparent Level-Sensitive Latch
  3. Edge-triggered Flip-Flop

• “Array” Memory (Background)
  – Large memory circuit that stores many bits of information organized into multiple words accessed by an address
  1. SRAM
  2. Multi-ported SRAM
  3. DRAM
  4. Flash
  5. etc.
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 (broken)
  ```
  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 (ok)
  ```
  reg b, c;
  always @(posedge clk) begin
    c = b;  // reordered from above
    b = a;  // reordered from above
  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.
The correct solution is to use a “non-blocking assignment” written with “<=” which causes the simulator to schedule all assignments at a particular point in time and perform them all simultaneously in “simulator time”.

With the verilog below, the registers perform as normal FFs behave without a race regardless of their ordering:

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

The key differentiator is the time when the value `b` is sampled. Here it is not “blocked” by the previous assignment `b<=a`.
Single-Bit Memory Rule #1

• Rule #1 (always follow in this class): Use only edge-triggered flip-flops and never transparent (clock level-sensitive) latches

• Edge-triggered flip-flops are generally robust with regards to clocking

• Transparent latches are vulnerable to signals racing through more than one latch during a single clock pulse; solutions are generally non-trivial
  – Requiring a minimum delay between latches works but adds area and power dissipation
  – Requiring a maximum pulse width on clocks (likely << 50%) can be problematic
  – A robust but tedious solution is to use two clocks with different non-overlapping phases, and require consecutive latches to use clocks of differing phases

\[
\begin{align*}
\text{clk\_phase1} & \quad \text{\textbackslash\textbackslash} & \quad \text{clk\_phase2} \\
\text{\textbackslash\textbackslash} & \quad \text{\textbackslash\textbackslash} & \quad \text{\textbackslash\textbackslash}
\end{align*}
\]
Single-Bit Memory Rule #2

• Rule #2 (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```

Single-Bit Memory Rule #3

• Rule #3 (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
```
Single-Bit Memory Rule #4

• Rule #4 (always follow in this class): Add “#1” to give one unit of clock-to-Q delay to increase waveform readability
  – Synthesis tools have no way to implement a #1 delay so they typically give a warning saying they can not add the delay and that the tool has ignored it
  – This warning can be ignored (the only warning that can be automatically ignored!)
  – Interestingly, Quartus does not give a warning
Single-Bit Memory Rule #5

- Rule #5 (really a guideline):
  Normally do not include any logic in flip-flop declarations. You will be tempted to include functions such as resets, initializations, and counter incrementing. Try to resist but it is ok if logic is simple.

```verilog
// Flip-flop declarations
always @(posedge clock) begin
    state <= #1 state_c;
    count <= #1 count_c;
    if (reset == 1'b1 && state_pipe3 != 8'hB2) begin
        state <= #1 4'b0000;  // init state
    end
end
```
Single-Bit Memory Rule #6

• Rule #6 (really a suggestion):
  • For large designs, it is usually clearer to group related combinational logic and FFs in separate areas of the module

// Block 1
[combinational logic]
always @(posedge clock) begin
  count <= #1 count_c;
  ...
end

// Block 2
[combinational logic]
always @(posedge clock) begin
  data_r <= #1 data;
  ...
end

// Block 3
[combinational logic]
always @(posedge clock) begin
  cat <= #1 mouse;
  ...
end
Preview of 2 Clock-Related Rules

• These will be covered in more detail in the *Clock* section

1) Only clock signals may connect to flip-flop or latch clock inputs

2) Clock signals may not connect to any node other than a flip-flop or latch clock input
   – No logic gate inputs
   – No flip-flop or latch inputs other than the clock input
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{figure}
\centering
\includegraphics[width=\textwidth]{sum.png}
\caption{(combinational logic version of \textit{sum}) \textit{sum}_c \rightarrow \textit{sum} \rightarrow \textit{inputA}_r \quad \text{(registered version of \textit{inputA})}}
\end{figure}
Signal Naming Conventions

- Another possibility I have seen is to add 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., \(r_{\text{sum}}\))
Signal Naming Conventions

• 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 \texttt{pipeX}
• \texttt{inA\_pipe1} – \texttt{inA} signal in pipeline stage 1
Reset-able and Enable-able Registers

- Sometimes it is convenient or necessary to have flip-flops with special inputs like reset and enable.
- When designing flip-flops/registers, it is ok (possibly required) for there to be cases where the always block is entered, but the reg is not assigned—such as when a FF is disabled.
- No fancy code, just make it work.
- Normally use synchronous reset instead of asynchronous reset.

![Diagram of reset and enable flip-flops]
Reset-able Registers

- Two example FFs with active-high \textit{reset}
- The code also leaves out begin-end statements which is not ideal
- This code “bends” Guideline #5

```verilog
always @(posedge clk) begin
    if (reset == 1'b0)
        out <= #1 D;
    else
        out <= #1 1'b0;
end

always @(posedge clk) begin
    if (reset == 1'b1)
        out <= #1 1'b0;
    else
        out <= #1 D;
end
```
Reset-able Registers

• Example FF registers with reset
• This code bends Guideline #5

```verilog
reg out;
reg [7:0] count;
reg [23:0] rgb;

always @(posedge clk) begin
  out   <= #1 D;
  count <= #1 count_c;
  rgb   <= #1 color24;
  if (reset == 1'b1) begin
    out   <= #1 1'b0;
    count <= #1 8'b0000_0000;  // ideally put reset elsewhere
    // assume rgb does not need to be reset
  end
end
data
```
Preset-able Registers

- Example FF with reset
- This code bends Guideline #5

```verilog
always @(posedge clk) begin
    out <= #1 D;
    if (preset == 1'b1) begin
        out <= #1 1'b1;
    end
end
```
Enable-able Registers

- Example FF with reset
- This is **not** combinational logic so it is not necessary that all outputs are set in every pass through the *always* block
- This code bends Guideline #5

```verilog
classic always @(posedge clk) begin
  if (enable == 1'b1) begin
    out <= #1 D;
  end
end
```
Reset-able and Enable-able Registers

- Example FF with reset and enable (reset has priority)

<table>
<thead>
<tr>
<th>reset</th>
<th>enable</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Do nothing</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D Flip-Flop</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Reset? or Do nothing?</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Reset, Q=0</td>
</tr>
</tbody>
</table>

```verilog
always @(posedge clk) begin
    if (reset)          // highest priority
        out <= #1 1'b0;
    else if (enable)
        out <= #1 c_out;
    // ok if no assignment (out holds value)
end
```
Reset-able and Enable-able Registers

- Example FF with reset and enable (enable has priority)

<table>
<thead>
<tr>
<th>reset</th>
<th>enable</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Do nothing</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D Flip-Flop</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Reset? or Do nothing?</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Reset, Q=0</td>
</tr>
</tbody>
</table>

always @(posedge clk) begin
  if (enable) begin // highest priority
    if (reset)
      out <= #1 1'b0;
    else
      out <= #1 c_out;
  end
  // It is ok if there is no assignment to “out” in some
  // cases (out is not assigned when enable==0 in which
  // case out holds its value). Recall that this is never
  // ok for combinational logic.
end
Resets Within a Large Datapath

- Use reset-able FFs only where truly needed
  - Reset-able FFs are a little larger and higher power
  - Requires the global routing of the high-fanout reset signal

reset only these two registers, but now reset must be enabled for at least 3 clock cycles
Asynchronous Resets

• Asynchronous resets will react to any pulse, even a very short “glitch” in the middle of a clock period
• What if the reset signal comes directly from combinational logic where glitches are common?
• What if a glitch comes from capacitive coupling or inductive coupling of wires?
• How to test in a big chip?
• They utilize a different circuit than what a synchronous reset uses
• However they are needed in:
  – clock generation
  – asynchronous interface logic