VERILOG II
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.
- No fancy code, just make it work.
- Normally use synchronous reset instead of asynchronous reset (easier to test).
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>

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
Reset-able and Enable-able Registers

- Use reset-able FFs only where 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

reset
Three types of “case” statements in verilog

1) case
   - Normal case statement

2) casez
   - Allows use of wildcard “?” character for don’t cares.

```verilog
casez(in)
  4'b1???: out = a;
  4'b01??: out = b;
  4'b00??: out = c;
  default: out = d;
endcase
```

3) casex
   - Don’t use it for 281. It can use “z” or “x” logic.
   • default
      - It may be wise to set the output to a special value even if you expect the state
        machine will never reach these default values
        • Setting unused states to x should allow the synthesis tool to simplify logic
        • Setting unused states to an easily-recognizable value (such as x’s) could make
          mistakes easier to spot
Hardwired Complex Functions

• Complex or “arbitrary” functions are not uncommon
• Examples
  – sin, cos, tan
  – tangent\(^{-1}\)
  – log
  – e\(^x\)
  – A/D converter correction values
  – RF mixer bias values

\[ \theta \rightarrow \sin/\cos \rightarrow \text{out_real} \rightarrow \text{out_imag} \]
Hardwired Complex Functions

- Two main approaches to implement complex functions
  - High-precision numerical calculations
    - Almost certainly requires many clock cycles per calculation
      - 1–2 bits per clock cycle is common. In some cases, more bits/cycle are possible by adding hardware
      - Can regain *throughput* by parallel implementations
      - However *latency* is unavoidable and may be less desirable
    - Ex: CORDIC, polynomial expansions, etc.
  - Lookup tables
    - ROM array memory
      - “Real” memory with address decoder, wordlines, bitlines, sense amplifiers, etc.
      - Frequently available as macros from the standard cell vendor
      - Could be mask-defined at manufacture, one-time programmable with fuses or anti-fuses, or off-chip Flash
    - Synthesized from standard cell logic
Hardwired Function in Verilog using a Lookup Table

- ROM array memories
  - Generally perform better with very large tables since ROM cells are among the densest of all CMOS structures

- Lookup tables
  - Generally perform better with data that is less random (in an entropy information-theory sense)
  - Less random data results in simpler and smaller logic equations

```verilog
always @(input) begin
  case (input)
    4'b0000: begin real=3'b100; imag=3'b001; end
    4'b0001: begin real=3'b000; imag=3'b101; end
    4'b0010: begin real=3'b110; imag=3'b011; end
    ...
    default: begin real=3'bxxx; imag=3'bxxx; end
  endcase
end
```
Hardwired Function in Verilog using a Lookup Table

• It is usually best to write a matlab program to print the verilog table as plain text
  – You will need several versions to get it right so rapid generation is a huge help
  – matlab has rock-solid common functions, rounding, etc.
  – matlab has superb plotting capabilities for checking all sorts of characteristics such as bias, frequency response, etc.
  – Easy to adapt to other specifications such as binary word width, number format, etc.
  – Print everything between “case” and “endcase” then copy & paste the output into your verilog file
  – Helpful commands:
    fprintf(...\n, x, out_real, out_imag);
    num2bin
Lookup Tables with Cascaded Functions

- In many cases, computation is expressed or can be transformed into cascaded functions.
- Example: The angle of a rectangular 2D vector = $\tan^{-1}(y/x)$
- A straightforward implementation using lookup tables would use a table for division followed by a table for $\tan^{-1}()$
- A better implementation would merge the cascaded functions into a single $\tan^{-1}(y/x)$ function implemented with a single memory:
  - Assuming the intermediate result $y/x$ is not needed elsewhere.
  - In both cases, the input address is the concatenated $\text{address} = \{x, y\}$ or $\{y, x\}$; in fact, the bits can be mixed arbitrarily although the two forms shown certainly are the clearest.
Working With Signed Values

- IEEE Verilog 1364-2001 allows wires and regs to be declared signed which makes signed arithmetic much easier.
- Unfortunately, it will be many years before all CAD tools fully support this feature.
- Therefore, in EEC 281, write all *.v hardware verilog code without declaring wires or regs signed. Anything is ok in your test code.
- In practice, it would be wise to run test cases on the particular mix of CAD tools used in a design before using signed regs or wires.