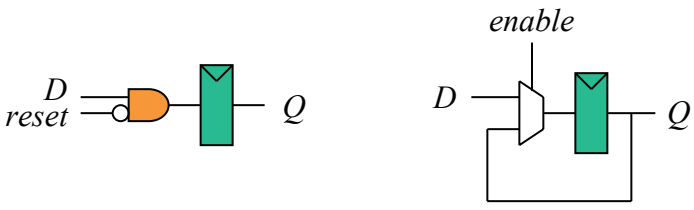
VERILOG II

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)

```
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
```

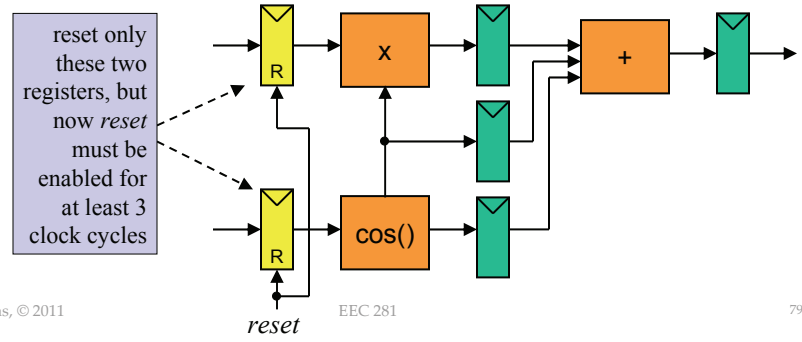

Reset-able and Enable-able Registers

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

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

Reset-able and Enable-able Registers

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



Three types of "case" statements in verilog

1) `case`
   - Normal case statement
2) `casez`
   - Allows use of wildcard "?" character for don't cares.

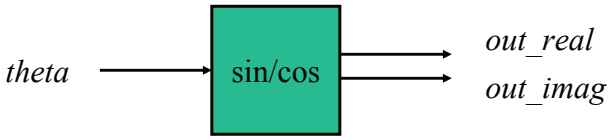
```
casez(in)
   4'b1???: out = a;
   4'b01??: out = b;
   4'b00??: out = c;
   default: out = d;
endcase
```

3) `casex`
   - Don't use it. Could use "z" or "x" logic.

- `default`
  - Normally set output to an easily-recognizable value (such as x's) in a `default` statement to make mistakes easier to spot

Hardwired Complex Functions

- Complex or "arbitrary" functions are not uncommon
- Examples
  - sin/cos
  - tangent^{-1}
  - log



Hardwired Function in Verilog using a Lookup Table

```
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
```

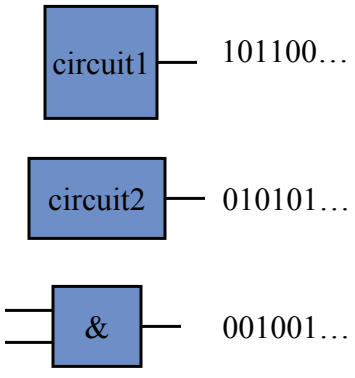
- Often best to write a matlab program to write the verilog table as plain text
  - You will need several versions to get it right
  - Easy to adapt to other specifications
- Not efficient for very large tables
- Tables with data that is less random will have smaller synthesized area

Working With Signed Values

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

Concurrency

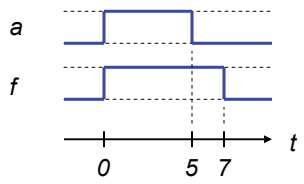
- All circuits operate independently and concurrently
  - Different from most programming paradigms



Concurrent Operation

- You should think of verilog modules as operating on independent circuits (remember *hardware* orientation).

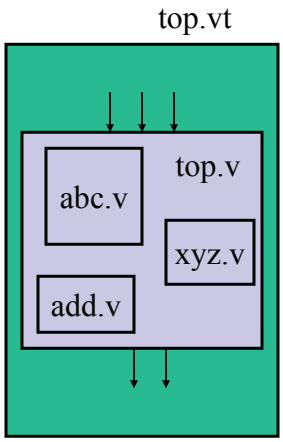
```
always begin
   a = (b&c) | d;
   #5;    // 5-unit delay
   a = ~a;
   #5;
end

always begin
   f = ~(g ^ h);
   #7;    // 7-unit delay
   f = ~f;
   #7;
end
```



TESTING

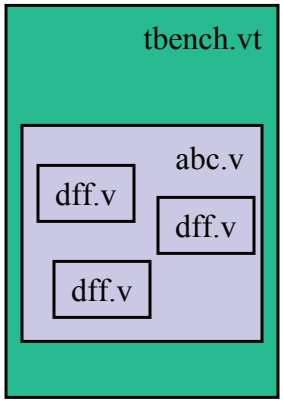
Testing

- Logic blocks (*.v)
  - Will be synthesized
  - No "#" delays except for registers
- Testing blocks (*.vt)
  - Pretty much anything goes
  - You'll need "#" delay statements
- Instantiate logic blocks inside testing blocks and drive the inputs and check outputs there



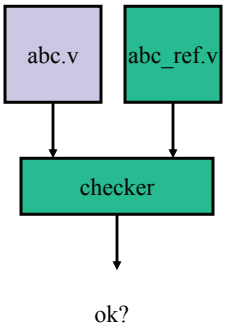
Example Given To You

- Example test bench and modules on web page under "Notes on running verilog"
- Not the most realistic partitioning (e.g., we would normally never put D FFs in their own modules), but it's a good working example to get you started



Verification

- A number of ways to verify designs
  - Eyeball text printouts
    - Quickest and easiest
  - Eyeball waveforms
    - Quick and easy
  - Write reference code and some other code to see if the two are the same. Make sure you temporarily force an error to test your setup.
    - This is the most robust and is what is required for non-trivial designs



"Bit-accurate" Verification

- As designs become more complex, verifying it is correct becomes more difficult
- Two approaches
  - A) Exhaustive testing
    - less practical or impossible
  - B) "Golden reference" approach
    - Write an easy to understand simple model in a higher-level language
      - C or matlab
      - Must be written in *a different way*
    - Designers agree this is the correct function
    - Many high-level tests run on golden reference
      - Model should be fast

"Bit-accurate" Verification

- B) "Golden reference" approach con't
  - Run a smaller number of tests but now hardware must match golden reference exactly, bit for bit
  - Easier to automate testing
  - Matlab is a natural choice for a DSP reference language
  - Matlab must now do awkward operations such as rounding and saturation that matches hardware
    - For example, `floor(in + 0.5)` for rounding