HDL Modeling

HDL Modeling
Combinational Circuits

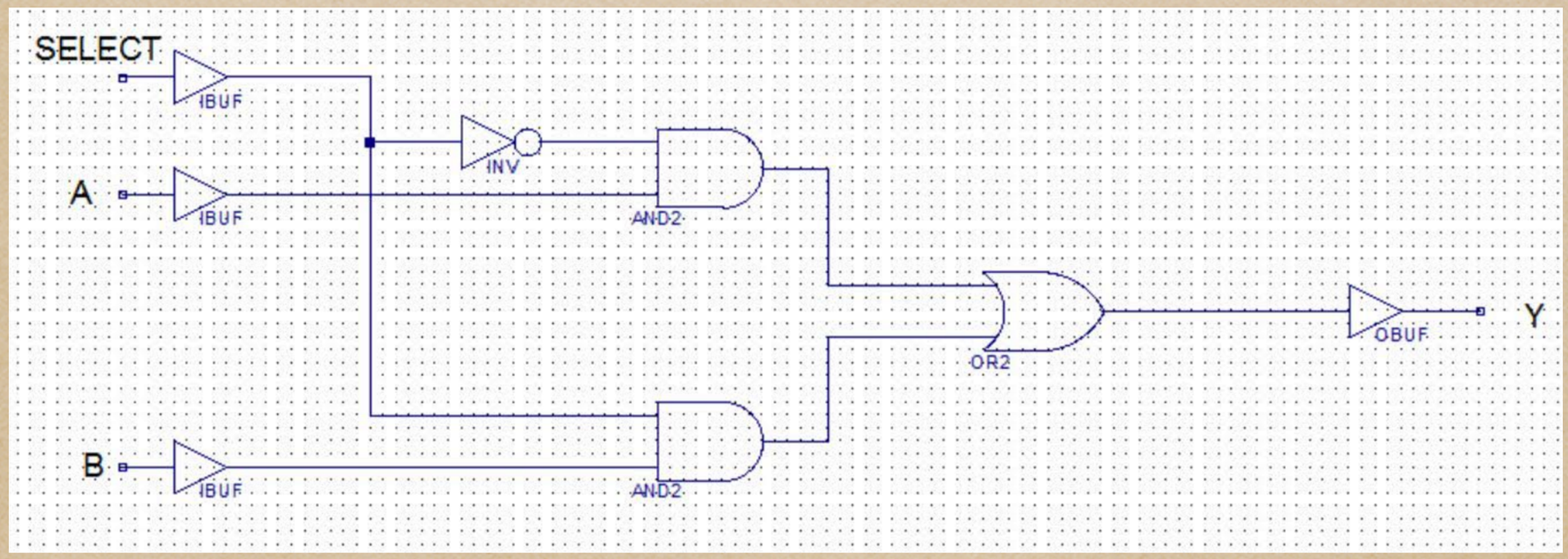
CSULB CECS
John Tramel

Alternate Descriptions of Combinational Logic

- Gate-level modeling using instantiations of predefined and user-defined primitive gates
  - Gate-level (structural)modeling describes a circuit by specifying its gates ad how they are connected with each other
- Dataflow modeling using continuous assignment statements with the keyword **assign**
  - Dataflow modeling is used mostly for describing the Boolean equations of combinational logic
- Behavioral modeling using procedural assignment statements with the keyword **always**
  - Behavioral modeling is used to describe combinational and sequential circuits at a higher level of abstraction

Gate Level Modeling

- Circuits here are specified by the logic gates that comprise the design and their interconnections
- Gate level modeling provides a textual description of what was found in a schematic diagram
- There are basic gates available as predefined primitives: **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not & buf**
- All are declared with lowercase keywords and are n-bit input devices (except **not/buf)**



Asynchronous (Combinational) Logic Design

- Three techniques for modeling asynchronous logic
  - 1 Continuous assignments
    - Used to define relationships (expressions)
    - Contained within the body of the module
    - Not contained within an *always block*
    - Useful for small and compact circuits
    - Data type of left-hand side always a *wire*
  - 2 Procedural block (*always*)
    - All inputs included in sensitivity list (*) (no edge qualifiers)
    - Functionality defined using expressions and operators
    - Expressions evaluated and output updated when any input changes
    - Useful for more complex circuit descriptions
    - Syntax includes powerful procedural structures
    - Data type of left-hand side is always a register (*reg or integer)*

Asynchronous (Combinational) Logic Design - 2

- Three techniques for modeling asynchronous logic
  - 3 Function
    - Similar to its use in software allowing partitioning into manageable parts
    - Defined within a module but not within an always block
    - Defines a relationship whose value will be updated when inputs switch
    - No timing delays allowed (strictly combinational)
    - No references to tasks
    - Functions should be general in order to enhance readability/reusability
    - Use functions instead of repeating the same code sequence

Unwanted Latch Inference

- Potential danger when using procedural blocks to define asynchronous logic
- When designing a latch may be inferred when not desired by omitting assignments for all cases
- Additional latches increase design size and change functionality
- Classic latch inference is achieved by defining an if-else structure without the else clause

```
if (en == 1'b1)
    q = d;
```

Expected Latch

```
case (var)
   2'b00: y = a + b;
   2'b01: y = a - b;
   2'b10: y = a * b;
endcase
```

Unexpected Latch

Synchronous Logic Design

- Synchronous logic is defined as a logic block of a defined functionality whose outputs will be captured in D flip-flops
- There may be a combinational construct within the block but the fact the outputs are registered classifies the block as synchronous
- All synchronous logic should use the following template

```
always @(posedge clock, posedge rst)
    if (rst == 1'b1)
        register <= INITIAL_STATE;
    else
        register <= ASSIGNED_STATE;
```

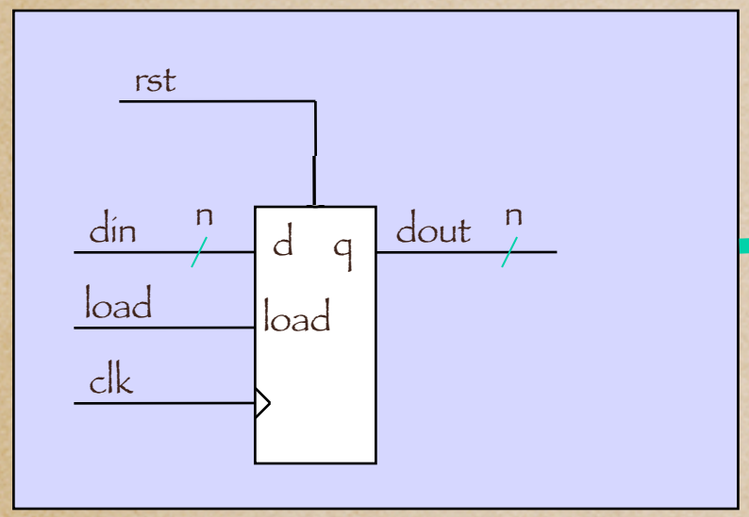
- Active edge of clock may be "posedge" or "negedge" but must be consistent throughout the block - we will use "posedge"
- Resets may be high or low active but we will be using HIGH ACTIVE for Xilinx FPGA designs

More Synchronous Logic Design

- Remember that synchronous RTL design results in the following:

- Our designs will have a common clock and a common reset. Data set in the first flop will be able to be sampled at the second flop on the next clock.
- Proper design techniques will assist in the debug of the resultant circuits once you have programmed the FPGA
- Partition your design by functionality - don't try and do too much in a single file. Remember to design before you implement RTL

Register Transfer Logic (RTL)



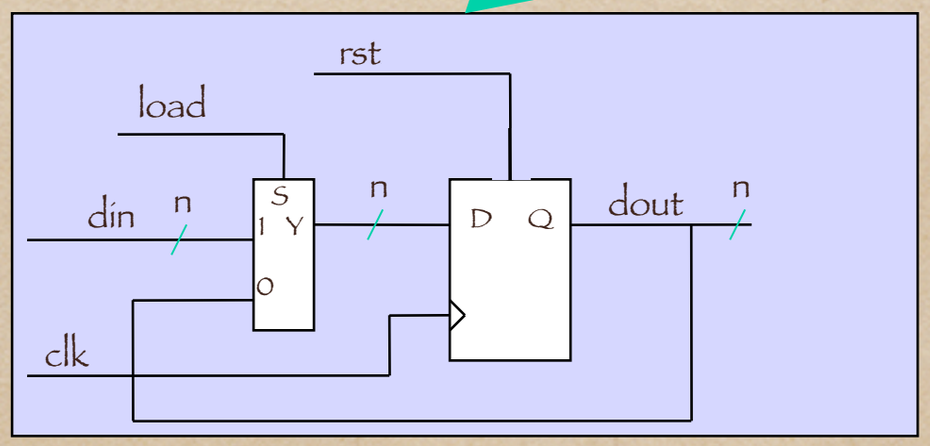
We Think

```
module register (clk, rst , load, din, dout);
input clk, rst , load;
input [n-1:0] din;
output [n-1:0] dout;

reg [n-1:0] dout;

always @(posedge clk or posedge  rst )
   if ( rst )
      dout <= 1'b0;
   else if (load)
      dout <= din;
endmodule
```

We Write



We Get

Variables and Identifiers

- Variable Declaration
  - Two basic data types used when modeling RTL
    - Procedural blocks (always) use **reg**
    - Continuous assignments use **wire**
    - There are other data types available but these will cover 99.9% of cases
  - Default data type in Verilog is a scalar wire – you must declare busses
- Verilog Identifiers
  - Module names, instance names, parameters, variables, block names, task names, function names are all identifiers
  - Verilog is case sensitive but do not rely on this (meaning don't have Signal, signal, SIGNAL)
  - Identifiers start with a letter and may contain letters, numbers and the underscore "_" - so numbers can be 8'b0010_1010
  - Use descriptive identifiers but avoid making them too long

Identifiers Continued

- Verilog Identifiers
  - Do not use Verilog or VHDL keywords
  - Remember Verilog is “case sensitive” so keywords are lower-case
  - Follow a consistent convention
    - Clocks begin with “clk” - clk30, or clk_30
    - Resets begin with “rst” - rst_n, or rstb
  - Designate low active with “_n” or “b” or similar
  - Designate chip I/O with I, O or IO - clk30_I
  - Embed the “_” to enhance readability - out_strobe (or OutStrobe)

Verilog Operators

- Verilog has a large number of operators supporting modeling
- Use the correct operators for your end product: RTL or verification
- Do not mix logical and bit-wise operators, use appropriately

Operator	Description	Usage Recommendation
~	Bit-wise negation	RTL modeling: Bit manipulation
!	Logical negation	RTL modeling: Bit manipulation
* / %	Multiply, Divide, Modulus	RTL modeling: Bit manipulation
+ -	Addition, Subtraction	RTL modeling: Bit manipulation
<< >>	Shift left, Shift right	RTL modeling: Bit manipulation
< <=	Less than, Less than or equal	RTL modeling: Decision making
> >=	Greater than, Greater than or equal	RTL modeling: Decision making
== !=	Logical equality, Logical inequality	RTL modeling: Decision making
=== !==	Case equality, Case inequality	Testbench (verification): Decision making
& ~&	Bit-wise AND,/NAND / Reduction	RTL modeling: Bit manipulation
^ ~^	Bit-wise XOR,/XNOR/ Reduction	RTL modeling: Bit manipulation
|	Bit-wise OR/Reduction	RTL modeling: Bit manipulation
&&	Logical AND	RTL modeling: Bit manipulation
||	Logical OR	RTL modeling: Decision making
?:	Conditional	RTL modeling: Bit manipulation

Numeric Values and Assignments

- Numeric Values
  - Default base is decimal (other bases – hex, octal & binary)
  - Numeric values defined as bits'base value – 16'habcd
    - # of bits is optional and for documentation purposes only (except for concatenation)
- Procedural Assignments (used in procedural blocks – always/initial
  - **Always use "=" (blocking) for combinational logic blocks**
  - **Always use "<=" (non-blocking) for synchronous blocks**
- NEVER mix blocking and non-blocking assignments in same block
- NEVER assign the same variable in more than one procedural block

Case Statements

- Case statements are commonly used for data selection
- Verilog has three variations: case, casex* and casez – * don't use
- casez allows use of "don't cares" in comparison
- Case statement compares case variable to the table entries – 1st match
- The casez allows a table entry of "?" to serve as "don't care"
- No 'break'

Data Value	case	casez
0	Matches: 0	Matches: 0
1	Matches: 1	Matches: 1
X	No match	Matches: X
Z	No match	Matches: 0, 1, X & Z

```
always @(opt or in1 or in2 or in3)
   casez (opt)
      2'b0?: out_d = in1;
      2'b10: out_d = in2;
      2'b11: out_d = in3;
    default: out_d = 'bx;
   endcase
```

“if - else” versus “case” constructs

- Case statements define parallel case structures where no path has priority over another
- These can be synthesized into fast full parallel multiplexers
- If the choice between the two is only style then case is preferred
- The “else” branch will always be associated with the most recent “if” as long as the association is not forced by begin-end pairs
- if statements generate priority muxes where the first options switch faster

```
begin
   if (a == 2'b00) out_val = input_fast; else
   if (a == 2'b01) out_val = input_slower; else
   if (a == 2'b10) out_val = input_slowerer; else
   if (a == 2'b11) out_val = input_slowest;
end
```

Loops

- Verilog supports “for” loops. The use of “for” loops is encouraged where their use enhances the readability of the source code.
- Care should be taken to ensure the width of the index variable supports the application

Module Definition

- When modules are defined the port list should be sorted by inputs, outputs and inouts
- It is a good technique to alphabetize each group for fast tracing
- When declaring inputs, outputs and inouts
  - It is good to define one port per line
  - Include a good comment covering the function of the port

Module Instantiation

- Always use “explicit” or “named” mapping when instantiating modules or library elements - “implicit” mapping should never be used
- The instance name should be the same as the module name wherever possible - multiple instances of one module requires iteration

BAD

```
shift_reg shift_reg (clk, rst_n, data_in, shift, data_out);
```

GOOD

```
shift_reg shift_reg
(
.clk(clk),
.data_in(data_in),
.rst_n(rst_n),
.shift(shift),
.data_out(data_out)
);
```

File Naming Conventions

- There should always be just one Verilog module per file
- The name of the file should correspond exactly to the name of the module within the file
- All Verilog RTL source files should have the extension ".v"
- Test benches should have the convention "file_tb.v"

Comments

- Comments should explain the logic blocks within the module (for yourself and for others)
- The code organization should dictate where comments are inserted
- Procedural blocks should be preceded with a detailed comment
- Do not flood the design with comments so as to make the code unreadable
- Single line comment "//" recommended over multi-line "/* … */"

File Creation

- When you create your files keep in mind that the editors will ‘word wrap’ if you exceed a particular width (80?)
- The end result is that when you print out your source code it becomes nearly unreadable
- Be conscious of this when creating your code so that the printed file retains the formatting
- Format you code to enhance readability. Use a common indentation scheme
- Avoid the use of tabs when creating your code. Different tools interpret tabs differently and the end result may not correspond with your intention
- When printing you code use a ‘proportional font’ to maintain a uniformity in the presentation of your code (columns align with proportional fonts)
- Avoid ‘over-nesting’ of procedural constructs

Module Header

- Please insert this header at the start of every file you create and update the information within '<…>'

```
//*************************************************************//
//  This document contains information proprietary to the      //
//  CSULB student that created the file - any reuse without    //
//  adequate approval and documentation is prohibited          //
//                                                             //
//  Class: <class that you are submitting this work for>       //
//  Project name: <name of the project you are working on>     //
//  File name: <name of this file>                             //
//                                                             //
//  Created by <yourname> on <date created>                    //
//  Copyright © 2016 <yourname>. All rights reserved.          //
//                                                             //
//  Abstract: <description of the purpose of this module>      //
//  Edit history: <keep track of changes to the file>          //
//                                                             //
//  In submitting this file for class work at CSULB            //
//  I am confirming that this is my work and the work          //
//  of no one else.                                            //
//                                                             //
//  In the event other code sources are utilized I will        //
//  document which portion of code and who is the author       //
//                                                             //
// In submitting this code I acknowledge that plagiarism       //
// in student project work is subject to dismissal from the class //
//*************************************************************//
```

Timing Delays

- The Verilog RTL should always be written without timing delays
- All delay characteristics will be supplied by the target library (this means no #5 in a design)
- Xilinx expects a **`timescale 1ns/10ps** before each module (1 per file)

Include Files

- The use of `include files for including Verilog code is discouraged
- If a common set of parameters is defined and reused then these may be included in each of the files

Conclusion

- Please note that adherence to these recommended guidelines will be a consideration when work is submitted for review.