Verilog
For Computer Design

CS/ECE 552
Karu Sankaralingam

Based on slides from
Derek Hower (UW-Madison), Andy Phelphs (UW-Madison) and
Prof. Milo Martin(University of Pennsylvania)
Overview

• Why Verilog?
  High-level description of Verilog

• Verilog Syntax
  • Primitives
  • Number Representation
  • Modules and instances
  • Wire and Reg Variables
  • Operators
  • Miscellaneous
    • Parameters, Pre-processor, case statements, Common errors, system tasks

• Sequential logic

• Test bench structure

• Case study, Verilog tools and Demo
Why Verilog and why not Manual design?

State of Art Design

Do you want to design this Processor manually?
How To Represent Hardware?

- If you’re going to design a computer, you need to write down the design so that:
  - You can read it again later
  - Someone else can read and understand it
  - It can be simulated and verified
  - Even software people may read it!
  - It can be synthesized into specific gates
  - It can be built and shipped and make money($$$)
Ways to represent hardware:

- Draw schematics
  - Hand-drawn (Seriously?)
  - Machine-drawn

- Write a netlist – ASCII representation of Interconnect of a schematic
  - Z52BH I1234 (N123, N234, N4567);

- Write primitive Boolean equations
  - AAA = abc DEF + ABC def

- Use a Hardware Description Language (HDL)
  - assign overflow = c31 ^ c32;
Hardware Description Languages (HDLs)

- Textual representation of a digital logic design
  - Can represent specific gates, like a netlist, or more abstract logic

- HDLs are not “programming languages”
  - No, really. Even if they look like it, they are not.
  - For many people, a difficult conceptual leap

- Similar development chain
  - Compiler: source code • assembly code • binary machine code
  - Synthesis tool: HDL source • gate-level specification • hardware
What is an HDL? – “Think hardware”

if(x != 0) vs. if((x <= -1) || (x >= 1))
What hardware is generated here?

module counter(clk,rst_n,cnt);

input clk,rst_n;
output [3:0] cnt;

reg [3:0] cnt;

always @(posedge clk) begin
  if (~rst_n)
    cnt = 4'b0000;
  else
    cnt = cnt+1;
end
endmodule

• It looks like a programming language
• It is **NOT** a programming language
  ✓ It is always critical to recall you are describing hardware
  ✓ This codes primary purpose is to generate hardware
  ✓ The hardware this code describes (a counter) can be simulated on a computer. In this secondary use of the language it does act more like a programming language.
Why an HDL is not a Programming Language

- In a program, we start at the beginning (e.g. "main"), and we proceed sequentially through the code as directed
- The program represents an algorithm, a step-by-step sequence of actions to solve some problem
  
  ```c
  for (i = 0; i<10; i=i+1) {
      if (newPattern == oldPattern[i]) match = i;
  }
  ```

- Hardware is all active at once; there is no starting point
Why Use an HDL?

• Enables Larger Designs
  - More abstracted than schematics, allows larger designs.
    ✓ Register Transfer Level Description
    ✓ Wide data paths (16, 32, or 64 bits wide) can be abstracted to a single vector
    ✓ Synthesis tool does the bulk of the tedious repetitive work
    ✓ Work at transistor/gate level for large designs: cumbersome

• Explore larger solution space
  ✓ Synthesis options can help optimize (power, area, speed)
  ✓ Synthesis options and coding styles can help examine tradeoffs
    - Speed | Power | area
Why use an HDL? (continued)

• Easy to write and edit
• Compact
• Don’t have to follow a maze of lines
• Easy to analyze with various tools

Why not to use an HDL

• You still need to visualize the flow of logic
• A schematic can be a work of art
  • But often isn’t! (My first Processor example 😊)
Other Important HDL Features

- Are highly portable (text)
- Are self-documenting (when commented well)
- Describe multiple levels of abstraction
- Represent parallelism
- Provides many descriptive styles
  - Structural
  - Register Transfer Level (RTL)
  - Behavioral
- Serve as input for synthesis tools
Starting with an example...

module fulladd (input A, B, Cin, 
              output sum, Cout );

assign sum = A ^ B ^ Cin;
assign Cout = (A & B)| (A & Cin)| (B & Cin);
endmodule
Pitfalls of trying to “program” in Verilog

- If you program sequentially, the synthesizer may add a lot of hardware to try to do what you say
  - In last example, need a priority encoder
- If you program in parallel (multiple “always” blocks), you can get non-deterministic execution – Race Condition
  - Which “always” happens first?
- You create lots of state that you didn’t intend
  - if (x == 1) out = 0;
  - if (y == 1) out = 1; // else out retains previous state? R-S latch!
- You don’t realize how much hardware you’re specifying
  - x = x + 1 can be a LOT of hardware
- Slight changes may suddenly make your code “blow up”
  - A chip that previously fit suddenly is too large or slow
Two Roles of HDL and Related Tools

- **#1: Specifying digital logic**
  - Specify the logic that appears in final design
  - Either
    - Translated automatically (called *synthesis*) or
    - Optimized manually (automatically checked for equivalence)

- **#2: Simulating and testing a design**
  - High-speed simulation is crucial for large designs
  - Many HDL *interpreters* optimized for speed
  - Testbench: code to test design, but not part of final design
Module Styles

• Modules can be specified different ways
  Structural – connect primitives and modules
  Dataflow – use continuous assignments
  Behavioral – use initial and always blocks

• A single module can use more than one of the above 3 coding styles!

What are the differences?
HDL Constructs

- **Structural** constructs specify actual hardware structures
  - Low-level, direct correspondence to hardware
    - Primitive gates (e.g., and, or, not)
    - Hierarchical structures via modules
  - Analogous to programming software in assembly

- **RTL/Dataflow** constructs specify an operation on bits
  - High-level, more abstract
    - Specified via equations, e.g., out = (a & b) | c

- **Behavioral** – Describes behavior of the circuit
  - Always, initial blocks, procedural assignments

- **Not all behavioral constructs are synthesizable**
  - We’ve already talked about the pitfalls of trying to “program”
  - But even some combinational logic won’t synthesize well
  - out = a % b  // modulo operation – what does this synthesize to?
  - We will not use: / % > >= < <= >> <<
module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;
wire N1, N2, N3;
and A0 (N1, V1, V2),
    A1 (N2, V2, V3),
    A2 (N3, V3, V1);
or Or0 (major, N1, N2, N3);
endmodule
RTL/Dataflow Example

Continuous Assignment Statement

module majority (major, V1, V2, V3) ;

output major ;
input V1, V2, V3 ;

assign major = V1 & V2 | V2 & V3 | V1 & V3;
endmodule
Behavioral Example

module majority (major, V1, V2, V3);  

    output reg major;  
    input V1, V2, V3;  

    always @(V1, V2, V3) begin  
        if (V1 && V2 || V2 && V3 || V1 && V3) major = 1;  
        else major = 0;  
    end  

endmodule
Overview

- Why Verilog?
  High-level description of Verilog

- Verilog Syntax
  - Primitives
  - Number Representation
  - Modules and instances
  - Wire and Reg Variables
  - Operators
  - Miscellaneous
    - Parameters, Pre-processor, case statements, Common errors, system tasks

- Sequential logic

- Test bench structure

- Case study, Verilog tools and Demo
Recall: Two Types of Digital Circuits

- **Combinational Logic**
  - Logic without state variables
  - Examples: adders, multiplexers, decoders, encoders
  - No clock involved
  - Not edge-triggered
  - All “inputs” (RHS nets/variables) are triggers

- **Sequential Logic (Details explained later)**
  - Logic with state variables
  - State variables: latches, flip-flops, registers, memories
  - Clocked - Edge-triggered by clock signal
  - State machines, multi-cycle arithmetic, processors
  - Only clock (and possibly reset) appear in trigger list
  - Can include combinational logic that feeds a FF or register
Verilog Structural Primitives

No declaration; can only be instantiated
Imp * - All output ports appear in list before any input ports
Optional drive strength, delay, name of instance
Example: and N25 (Z, A, B, C); //instance name
Example: and #10 (Z, A, B, X); // delay
(X, C, D, E); //delay
/* Usually better to provide instance name for debugging. */

Example: or N30 (SET, Q1, AB, N5),
    N41 (N25, ABC, R1);
Example: and #10 N33(Z, A, B, X); // name + delay
Number Representation

Format: <size><base_format><number>

Examples:

6'b010_111 gives 010111
8'b0110 gives 00000110
8'b1110 gives 00001110
4'bx01 gives xx01
16'H3AB gives 0000001110101011
24 gives 0...0011000
5'O36 gives 11100
16'Hx gives xxxxxxxxxxxxxxxx
8'hz gives zzzzzzzz
Connections – Module Instantiations

• By position association
  
  module 2_to_4_decode (A, E_n, D);
  4_to_16_decode DX (X[3:2], W_n, word);
  A = X[3:2], E_n = W_n, D = word

• By name association (this is supposed to be used in HW s and Projects)
  
  module 2_to_4_decode (A, E_n, D);
  C_2_4_decoder_with_enable DX (.E_n(W_n), .A(X[3:2]), .D(word));
  A = X[3:2], E_n = W_n, D = word
Hierarchical Verilog Example

- Build up more complex modules using simpler modules
- Example: 4-bit wide mux from four 1-bit muxes
  - Again, just “drawing” boxes and wires

```verilog
module mux2to1_4(
    input [3:0] A,
    input [3:0] B,
    input Sel,
    output [3:0] O
);

    mux2to1 mux0 (Sel, A[0], B[0], O[0]);
    mux2to1 mux1 (Sel, A[1], B[1], O[1]);
    mux2to1 mux2 (Sel, A[2], B[2], O[2]);
    mux2to1 mux3 (Sel, A[3], B[3], O[3]);

endmodule
```
Variables

- Nets (Also called as wires)
  - Used for structural connectivity

- Registers
  - Abstraction of storage (May or may not be real physical storage)

- Properties of Both
  - Informally called signals
  - May be either scalar (one bit) or vector (multiple bits)
module mux2to1(
    input S, A, B,
    output O);
wire S_, AnS_, BnS;
not (S_, S);
and (AnS_, A, S_);
and (BnS, B, S);
or (O, AnS_, BnS);
endmodule
Net (wire) Examples

- Wire vectors:
  ```
  wire [7:0] W1;     // 8 bits, w1[7] is MSB
  ```
  • Also called “buses”

- Operations
  - Bit select: \( W1[3] \)
  - Range select: \( W1[3:2] \)
  - Concatenate:
  ```
  vec = {x, y, z};
  {carry, sum} = vec[0:1];
  ```
  • e.g., swap high and low-order bytes of 16-bit vector
  ```
  wire [15:0] w1, w2;
  assign w2 = {w1[7:0], w1[15:8]}
  ```
Wire and Vector Assignment

- Wire assignment: “continuous assignment”
  - Connect combinational logic block or other wire to wire input
  - **Order of statements not important to Verilog**, executed totally in parallel
  - But order of statements can be important to clarity of thought!
  - When right-hand-side changes, it immediately flows through to left
  - Designated by the keyword `assign`

```verilog
wire c;
assign c = a | b;
wire c = a | b;     // same thing
```
Register Assignment

• A register may be assigned value only within:
  ✓ a procedural statement
  ✓ a user-defined sequential primitive
  ✓ a task, or
  ✓ a function.

• A reg object may never be assigned value by:
  ✓ a primitive gate output
  ✓ or a continuous assignment

Examples:
  
  reg a, b, c;
  reg [15:0] counter, shift_reg;
  reg [8:4] flops;
When to use wire and when reg!

**Wire**
- Module declaration = Inputs(Yes), Outputs (Yes)
- Module instantiation = Connect input and output ports
- Must be driven by something, cannot store values
- Only legal type on left side of an assign statement
- Not allowed on left side of = or <= in an always@ block
- Most of the times combinational logic

**Reg**
- Module instantiation = Input port (Yes), Output Port (No)
- Module declaration = Inputs(No), Outputs (Yes)
- Only legal type on left side of = or <= in an always@ block
- Only legal type on left side of initial block (test bench)
- Not Allowed on left side of an assign statement
- Used for both sequential and combinational logic

CS/ECE 552, Spring 2008
Operators

• Operators similar to C or Java
• On wires:
  • & (and), | (or), ~ (not), ^ (xor)
• On vectors:
  • &, |, ~, ^ (bit-wise operation on all wires in vector)
    • E.g., assign vec1 = vec2 & vec3;
  • & |, ^ (reduction on the vector)
    • E.g., assign wire1 = | vec1;
  • Even ==, != (comparisons)

Can be arbitrarily nested: (a & ~b) | c
Conditional Operator

• Verilog supports the ?: conditional operator
  • Just like in C
  • But much more common in Verilog

• Examples:

  ```verilog
  assign out = S ? B : A;
  ```

  ```verilog
  assign out = sel == 2'b00 ? a :
              sel == 2'b01 ? b :
              sel == 2'b10 ? c :
              sel == 2'b11 ? d : 1'b0;
  ```

• What do these do?
Overview

- Why Verilog?
  High-level description of Verilog

- Verilog Syntax
  - Primitives
  - Number Representation
  - Modules and instances
  - Wire and Reg Variables
  - Operators
  - Miscellaneous
    - Parameters, Pre-processor, case statements, Common errors, system tasks

- Sequential logic

- Test bench structure

- Case study, Verilog tools and Demo
Parameters

- Allow per-instantiation module parameters
  - Use “parameter” statement
- modname #(10, 20, 30) instname(in1, out1);
- Example:

```verbatim
defmodule mux2to1_N(Sel, A, B, O);
  parameter N = 1
  input [N-1:0] A;
  input [N-1:0] B;
  input Sel;
  output [N-1:0] O;
  mux2to1 mux0[N-1:0] (Sel, A, B, O);
endmodule

Mux2to1_N #(4) mux1 (S, in1, in2, out)
```
Verilog Pre-Processor

- Like the C pre-processor
  - But uses ` (back-tick) instead of #
  - Constants: `define
    - No parameterized macros
    - Use ` before expanding constant macro
      `define letter_A 8’h41
      wire w = `letter_A;
  - Conditional compilation: `ifdef, `endif
  - File inclusion: `include

- Parameter vs `define
  - Parameter only for “per instance” constants
  - `define for “global” constants
Common Errors

- Tools are from a less gentle time
  - More like C, less like Java
  - Assume that you mean what you say

- Common errors:
  - Not assigning a wire a value
  - Assigning a wire a value more than once

- Avoid names such as:
  - clock, power, pwr, ground, gnd, vdd, vcc, init, reset
  - Some of these are “special” and will silently cause errors
  - We will use “clk” and “rst”, but only for their intended uses
Verilog in Project / Homework

- Use the primitive modules and other basic modules given in course webpage for your ‘design’
- Follow the Verilog rules only for Design
- You are free to use your own test bench
- Only use the specified Verilog Keywords, allowed operators
- Go through the usage examples

- Ask TA’s if you are experiencing any difficulty in following these guidelines.
Non-binary Hardware Values

- A hardware signal can have four values
  - 0, 1
  - x: don’t know, don’t care
  - z: high-impedance (no current flowing)

- Two meanings of “x”
  - Simulator indicating an unknown state
  - Or: You telling synthesis tool you don’t care
    - Synthesis tool makes the most convenient circuit (fast, small)
    - Use with care, leads to synthesis dependent operation

- Uses for “z”
  - Tri-state devices drive a zero, one, or nothing (z)
  - Many tri-states drive the same wire, all but one must be “z”
    - Example: multiplexer
Case Statements

case (<expr>)
   <match-constant1>:<stmt>
   <match-constant2>:<stmt>
   <match-constant3>,<match-constant4>:<stmt>
   default:  <stmt>
endcase
Case Statements

• Useful to make big muxes
• Very useful for “next-state” logic
• But they are easy to abuse
• If you don’t set a value, it retains its previous state
  • Which is a latch!
• We will allow case statements, but with some severe restrictions:
  • Every value is set in every case
  • Every possible combination of select inputs must be covered
  • Each case lives in its own “always” block, sensitive to changes in all of its input signals
  • This is our only use of “always” blocks
Different types of Case statements

Verilog has three types of case statements: 
- **case**, **casex**, and **casez**
  - Performs bitwise match of expression and case item
    - Both **must** have same bitwidth to match!
  - **case**
    - Can detect **x** and **z**! (good for testbenches)
  - **casez**
    - Uses **z** and **?** as “don’t care” bits in case items and expression
  - **casex**
    - Uses **x**, **z**, and **?** as “don’t care” bits in case items and expression
always @*
  casex ({goBack, currentState, inputA, inputB})
    6'b1_???_?_? : begin out = 0; newState = 3'b000; err=0; end
    6'b0_000_0_? : begin out = 0; newState = 3'b000; err=0; end
    6'b0_000_1_? : begin out = 0; newState = 3'b001; err=0; end
    6'b0_001_1_? : begin out = 0; newState = 3'b001; err=0; end
    6'b0_001_0_0 : begin out = 0; newState = 3'b010; err=0; end
    6'b0_001_0_1 : begin out = 0; newState = 3'b010; err=0; end
    6'b0_010_?_0 : begin out = 0; newState = 3'b010; err=0; end
    6'b0_010_?_1 : begin out = 0; newState = 3'b011; err=0; end
    6'b0_011_?_1 : begin out = 0; newState = 3'b011; err=0; end
    6'b0_011_?_0 : begin out = 0; newState = 3'b100; err=0; end
    6'b0_100_?_? : begin out = 1; newState = 3'b000; err=0; end
    6'b0_101_?_? : begin out = 0; newState = 3'b000; err=1; end
    6'b0_110_?_? : begin out = 0; newState = 3'b000; err=1; end
    6'b0_111_?_? : begin out = 0; newState = 3'b000; err=1; end
  default: begin out = 0; newState = 3'b000; err=1; end
endcase
What happens if it’s wrong?

Here are our rules:

- A case statement should always have a default
- Hitting this default is an error
- Every module has an “err” output
- Can be used for other checks, like illegal inputs
- OR together all “err” signals -- bring “err” all the way to top
- Our clock/reset module will print a message if err == 1
System tasks

- Start with `$`
- For output:
  
  ```plaintext
  $display(<fmtstring><,signal>*)
  $fdisplay(<fhandle>,<fmtstring><,signal>*)
  ```
  
- Signal printf/fprintf
  
- ```plaintext
  $monitor(<fmtstring><,signal>*)
  ```
  
- Non-procedural printf, prints out when a signal changes
  
- ```plaintext
  $dumpvars(1<,signal>*)
  ```
  
- Similar to monitor
- VCD format for waveform viewing (gtkwave)
- Output is in dumpfile.vcd
More System Tasks

$\texttt{time}$
- Simulator’s internal clock (64-bit unsigned)
- Can be used as both integer and auto-formatted string

$\texttt{finish}$
- Terminate simulation

$\texttt{stop}$
- Pause simulation and debug

$\texttt{readmemh}(\texttt{<fname>},\texttt{<mem>},\texttt{<start>},\texttt{<end>});$  
$\texttt{writememh}(\texttt{<fname>},\texttt{<mem>},\texttt{<start>},\texttt{<end>});$  
- Load contents of ASCII file to memory array (and vice versa)  
- Parameters $\texttt{<start>},\texttt{<end>}$ are optional  
- Useful for loading initial images, dumping final images
Overview

- **Why Verilog?**
  High-level description of Verilog
- **Verilog Syntax**
  - Primitives
  - Number Representation
  - Modules and instances
  - Wire and Reg Variables
  - Operators
  - Miscellaneous
    - Parameters, Pre-processor, case statements, Common errors, system tasks
- **Sequential logic**
- **Test bench structure**
- **Case study, Verilog tools and Demo**
Sequential Logic in Verilog

Use the Homework Modules provided-
Instantiate the dff module given for all FFs

module dff
(q, d, clk, rst);

output q;
input d;
input clk;
input rst;
reg state;

assign #(1) q = state;
always @(posedge clk) begin
state = rst? 0 : d;
end
endmodule
Designing Sequential Logic

• CS/ECE 552 design rule: separate combinational logic from sequential state elements in lowest-level modules
  • Not enforced by Verilog, but a very good idea
  • Possible exceptions: counters, shift registers
• We’ll give you a 1-bit flip-flop module (see previous slide)
  • Edge-triggered, not a latch
  • Use it to build n-bit register, registers with “load” inputs, etc.
• Example use: state machine
Clocks Signals

- Clocks signals are not normal signals

- Travel on dedicated “clock” wires
  - Reach all parts of the chip
  - Special “low-skew” routing

- Ramifications:
  - Never do logic operations on the clocks
  - If you want to add a “write enable” to a flip-flop:
    - Use a mux to route the old value back into it
    - Do not just “and” the write-enable signal with the clock!

- Messing with the clock can cause errors
  - Often can only be found using timing simulation
Overview

• Why Verilog?
  High-level description of Verilog

• Verilog Syntax
  • Primitives
  • Number Representation
  • Modules and instances
  • Wire and Reg Variables
  • Operators
  • Miscellaneous
    • Parameters, Pre-processor, case statements, Common errors, system tasks

• Sequential logic

• Test bench structure

• Case study, Verilog tools and Demo
Verilog Simulation using Modelsim

- Testbench
- Setting up the mentor environment
- Using modelsim (simple example: 4 bit register)
  - Fixing compile errors
  - Debugging functional errors (with waveforms)
- Shortcut! Use wsrun.pl
- Vcheck – check for illegal constructs
- Pattern/Sequence detector
Testbench – variant 1

And “visually” inspect the outputs
Testbench – variant 2

Stimulus

Design

Outputs

Expected Outputs

Pass / Fail

Visual inspection not required!
Sophisticated Test Environment
(To be used in the course project)

Inputs

- Design
- Functional model written in other languages (e.g., C)

Pass / Fail
Setting up the mentor environment

- Edit `.bashrc` or `.bashrc.local`
- Create mentor directory in home area, copy over `.location` and edit it
- Find detailed instructions at:
  - Sidebar of Course home page -> “Tools” -> “Getting started with Mentor”
Using modelsim (simple example: 4 bit register)

• Interfaces:
• What we are going to build:
  ```
  module reg_4bit(out, in, wr_en, clk, rst)
  ```
• What we have to start with:
  ```
  module dff (q, d, clk, rst);
  ```
Using modelsim
(simple example: 4 bit register)
Shortcut! Use wsrun.pl

- `wsrun.pl -wave reg4bit_bench *.v`

- Find detailed instructions at:
  - Sidebar of Course home page -> “Command-line Simulation”
Vcheck – check for illegal constructs

- Limited subset of verilog constructs allowed in CS552
- Restriction is only for the “Design”
- Testbench can use any valid verilog syntax
- 3 ways to run the checks. Find detailed instructions at:
  - Sidebar of Course home page -> “Tools” -> “Verilog Rules Check”
Pattern/Sequence detector

Pattern: 1101
Pattern/Sequence detector

- Use binary numbers to encode state
- Current state: 4 bit binary number: Q2 Q1 Q0
- Next state: 4 bit binary number: Q2\textsubscript{n} Q1\textsubscript{n} Q0\textsubscript{n}
Pattern/Sequence detector

<table>
<thead>
<tr>
<th>Q2</th>
<th>Q1</th>
<th>Q0</th>
<th>InA</th>
<th>Q2_n</th>
<th>Q1_n</th>
<th>Q0_n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- \( \text{Out} = f(Q_2, Q_1, Q_0) \)
- \( Q_{2n} = f(Q_2, Q_1, Q_0, \text{InA}) \)
- \( Q_{1n} = f(Q_2, Q_1, Q_0, \text{InA}) \)
- \( Q_{0n} = f(Q_2, Q_1, Q_0, \text{InA}) \)

- \( \text{Out} = Q_2 \)
- \( Q_{2n} = Q_2' Q_1 Q_0 \text{InA} \)
- \( Q_{1n} = Q_2' Q_1' Q_0 \text{InA} + Q_2' Q_1 Q_0' \text{InA}' + Q_2' Q_1 Q_0' \text{InA} + Q_2 Q_1' Q_0' \text{InA} \)
- \( Q_{0n} = Q_2' Q_1' Q_0' \text{InA} + Q_2' Q_1 Q_0' \text{InA}' \)
A sequence detector is sequential logic

Design Rule: separate combinational logic from sequential state elements in lowest-level modules

We will give you a 1-bit flip-flop module to hold state and a clock/reset generator
- See the course web site
• Two ways to implement the “Combinational logic” block.

  1) Implement the state equations using verilog bitwise logical operators

  2) Use a case statement to specify the state transitions