Verilog for Sequential Circuits

Design of Digital Circuits 2016
Srdjan Capkun
Frank K. Gürkaynak

http://www.syssec.ethz.ch/education/Digitaltechnik_16
What will we learn?

- Short summary of Verilog Basics
- Sequential Logic in Verilog
- Using Sequential Constructs for Combinational Design
- Finite State Machines
Summary: Defining a module

- A module is the main building block in Verilog

- We first need to declare:
  - Name of the module
  - Types of its connections (input, output)
  - Names of its connections
Summary: Defining a module

```verilog
module example (a, b, c, y);
    input a;
    input b;
    input c;
    output y;

    // here comes the circuit description

endmodule
```
Summary: What if we have busses?

- You can also define multi-bit busses.
  - [range_start : range_end]

```vhdl
input  [31:0] a;  // a[31], a[30] .. a[0]
output [15:8] b1; // b1[15], b1[14] .. b1[8]
output [7:0]  b2; // b2[7], b2[6] .. b1[0]
input   clk;
```
Structural HDL Example

Short Instantiation

```verilog
module top (A, SEL, C, Y);
    input A, SEL, C;
    output Y;
    wire n1;

    // alternative
    small i_first (A, SEL, n1);

    /* Shorter instantiation, 
       pin order very important */

    // any pin order, safer choice
    small i2 (.B(C),
               .Y(Y),
               .A(n1));

endmodule
```

```verilog
module small (A, B, Y);
    input A;
    input B;
    output Y;

    // description of small
endmodule
```
module gates(input [3:0] a, b,
            output [3:0] y1, y2, y3, y4, y5);

/* Five different two-input logic
gates acting on 4 bit busses */

assign y1 = a & b;     // AND
assign y2 = a | b;     // OR
assign y3 = a ^ b;     // XOR
assign y4 = ~(a & b);  // NAND
assign y5 = ~(a | b);   // NOR

endmodule
Summary: Conditional Assignment

- `? :` is also called a ternary operator because it operates on 3 inputs:
  - `s`  
  - `d1`  
  - `d0`.

module mux2(input [3:0] d0, d1,
            input    s,
            output [3:0] y);

assign y = s ? d1 : d0;
// if (s) then y=d1 else y=d0;
endmodule
Summary: How to Express numbers?

\[ N'Bxx \]

\[ 8'b0000_0001 \]

- \textbf{(N) Number of bits}
  - Expresses how many bits will be used to store the value

- \textbf{(B) Base}
  - Can be b (binary), h (hexadecimal), d (decimal), o (octal)

- \textbf{(xx) Number}
  - The value expressed in base, apart from numbers it can also have X and Z as values.
  - Underscore _ can be used to improve readability
## Summary: Verilog Number Representation

<table>
<thead>
<tr>
<th>Verilog</th>
<th>Stored Number</th>
<th>Verilog</th>
<th>Stored Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>4'b1001</td>
<td>1001</td>
<td>4'd5</td>
<td>0101</td>
</tr>
<tr>
<td>8'b1001</td>
<td>0000 1001</td>
<td>12'hFA3</td>
<td>1111 1010 0011</td>
</tr>
<tr>
<td>8'b0000_1001</td>
<td>0000 1001</td>
<td>8'o12</td>
<td>00 001 010</td>
</tr>
<tr>
<td>8'bX0X1zZ1</td>
<td>XX0X 1ZZ1</td>
<td>4'h7</td>
<td>0111</td>
</tr>
<tr>
<td>'b01</td>
<td>0000 .. 0001</td>
<td>12'h0</td>
<td>0000 0000 0000</td>
</tr>
</tbody>
</table>
## Precedence of Operations in Verilog

<table>
<thead>
<tr>
<th>Highest</th>
<th>~</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*, /, %</td>
<td>mult, div, mod</td>
</tr>
<tr>
<td></td>
<td>+, -</td>
<td>add, sub</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;, &gt;&gt;</td>
<td>shift</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;&lt;, &gt;&gt;&gt;</td>
<td>arithmetic shift</td>
</tr>
<tr>
<td></td>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>comparison</td>
</tr>
<tr>
<td></td>
<td>==, !=</td>
<td>equal, not equal</td>
</tr>
<tr>
<td></td>
<td>&amp;, ~&amp;</td>
<td>AND, NAND</td>
</tr>
<tr>
<td></td>
<td>^, ~^</td>
<td>XOR, XNOR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>, ~</td>
</tr>
<tr>
<td>Lowest</td>
<td>?:</td>
<td>ternary operator</td>
</tr>
</tbody>
</table>
Sequential Logic in Verilog

- Define blocks that have memory
  - Flip-Flops, Latches, Finite State Machines

- Sequential Logic is triggered by a ‘CLOCK’ event
  - Latches are sensitive to level of the signal
  - Flip-flops are sensitive to the transitioning of clock

- Combinational constructs are not sufficient
  - We need new constructs:
    - always
    - initial
always Statement, Defining Processes

always @ (sensitivity list)
statement;

- Whenever the event in the sensitivity list occurs, the statement is executed
Example: D Flip-Flop

```verilog
module flop(input clk,
            input [3:0] d,
            output reg [3:0] q);

    always @ (posedge clk)
    q <= d;       // pronounced “q gets d”

endmodule
```
Example: D Flip-Flop

```
module flop(input clk,
            input [3:0] d,
            output reg [3:0] q);

always @(posedge clk)
    q <= d; // pronounced “q gets d”

endmodule
```

- The posedge defines a rising edge (transition from 0 to 1).
- This process will trigger only if the clk signal rises.
- Once the clk signal rises: the value of d will be copied to q
Example: D Flip-Flop

```verilog
module flop(input clk,
            input [3:0] d,
            output reg [3:0] q);

  always @(posedge clk)
    q <= d; // pronounced “q gets d”

endmodule
```

- ‘assign’ statement is not used within always block
- The <= describes a ‘non-blocking’ assignment
  - We will see the difference between ‘blocking assignment’ and ‘non-blocking’ assignment in a while
Example: D Flip-Flop

```verilog
module flop(input clk, 
            input [3:0] d, 
            output reg [3:0] q);

always @ (posedge clk) 
  q <= d; // pronounced “q gets d”

endmodule
```

- Assigned variables need to be declared as `reg`
- The name `reg` does not necessarily mean that the value is a register. (It could be, it does not have to be).
- We will see examples later
D Flip-Flop with Asynchronous Reset

```verilog
module flop_ar (input clk, input reset, input [3:0] d, output reg [3:0] q);

always @(posedge clk, negedge reset)
begin
    if (reset == '0') q <= 0; // when reset
    else q <= d; // when clk
end
endmodule
```

- In this example: two events can trigger the process:
  - A rising edge on clk
  - A falling edge on reset
D Flip-Flop with Asynchronous Reset

```verilog
module flop_ar (input clk,
                input reset,
                input [3:0] d,
                output reg [3:0] q);

always @(posedge clk, negedge reset)
begin
    if (reset == '0') q <= 0;  // when reset
    else q <= d;              // when clk
end
endmodule
```

- For longer statements a `begin` `end` pair can be used
  - In this example it was not necessary

- The always block is *highlighted*
D Flip-Flop with Asynchronous Reset

```verilog
module flop_ar (input clk,
               input reset,
               input [3:0] d,
               output reg [3:0] q);

always @ (posedge clk, negedge reset)
begin
    if (reset == '0') q <= 0;  // when reset
    else
        q <= d;  // when clk
end
endmodule
```

- **First reset is checked, if reset is 0, q is set to 0.**
  - This is an ‘asynchronous’ reset as the reset does not care what happens with the clock

- **If there is no reset then normal assignment is made**
D Flip-Flop with **Synchronous Reset**

```verilog
module flop_sr (input clk, input reset, input [3:0] d, output reg [3:0] q);

always @(posedge clk)
begin
  if (reset == '0') q <= 0;  // when reset
  else                q <= d;  // when clk
end
endmodule
```

- **The process is only sensitive to clock**
  - Reset *only happens* when the *clock rises*. This is a ‘synchronous’ reset

- **A small change, has a large impact on the outcome**
D Flip-Flop with Enable and Reset

```verilog
module flop_ar (input clk, input reset, input en, input [3:0] d, output reg [3:0] q);

always @ (posedge clk, negedge reset)
begin
    if (reset == '0') q <= 0; // when reset
    else if (en) q <= d; // when en AND clk
end
endmodule
```

- A flip-flop with enable and reset
  - Note that the en signal is not in the sensitivity list

- Only when “clk is rising” AND “en is 1” data is stored
Example: D Latch

```verilog
module latch (input clk,
              input [3:0] d,
              output reg [3:0] q);

always @ (clk, d)
  if (clk) q <= d; // latch is transparent when // clock is 1

endmodule
```

Diagram showing the D latch with inputs d[3:0], clk, and output q[3:0].
Summary: Sequential Statements so far

- Sequential statements are within an ‘always’ block
- The sequential block is triggered with a change in the sensitivity list
- Signals assigned within an always must be declared as reg
- We use <= for (non-blocking) assignments and do not use ‘assign’ within the always block.
Summary: Basics of always Statements

module example (input clk,
    input [3:0] d,
    output reg [3:0] q);

wire [3:0] normal;       // standard wire
reg [3:0] special;       // assigned in always

always @(posedge clk)
special <= d;           // first FF array

assign normal = ~ special; // simple assignment

always @(posedge clk)
q <= normal;            // second FF array
endmodule

- You can have many always blocks
Summary: Basics of always Statements

module example (input clk,
               input [3:0] d,
               output reg [3:0] q);

wire [3:0] normal; // standard wire
reg [3:0] special;  // assigned in always

always @(posedge clk)
special <= d;       // first FF array

assign normal = ~ special; // simple assignment

always @(posedge clk)
q <= normal;        // second FF array
endmodule

Assignments are different within always blocks
Why does an always Statement Memorize?

module flop (input clk,
            input [3:0] d,
            output reg [3:0] q);

    always @ (posedge clk)
    begin
        q <= d;  // when clk rises copy d to q
    end
endmodule

- This statement describes what happens to signal q
- ... but what happens when clock is not rising?
Why does an always Statement Memorize?

This statement describes what happens to signal q

... but what happens when clock is not rising?

The value of q is preserved (memorized)
Why does an always Statement Memorize?

```verilog
module comb (input inv,
             input [3:0] data,
            output reg [3:0] result);

always @ (inv, data) // trigger with inv, data
  if (inv) result <= ~data; // result is inverted data
  else result <= data; // result is data

endmodule
```

- This statement describes what happens to signal result
  - When `inv` is 1, result is `~data`
  - What happens when `inv` is `not 1`?
Why does an always Statement Memorize?

module comb (input inv,
            input [3:0] data,
            output reg [3:0] result);

always @ (inv, data) // trigger with inv, data
if (inv) result <= ~data; // result is inverted data
else result <= data; // result is data

endmodule

- **This statement describes what happens to signal result**
  - When inv is 1, result is \(~data\)
  - When inv is not 1, result is data

- **Circuit is combinational (no memory)**
  - The output (result) is defined for all possible inputs (inv data)
always Blocks for Combinational Circuits

- If the statements define the signals completely, nothing is memorized, block becomes combinational.
  - Care must be taken, it is easy to make mistakes and unintentionally describe memorizing elements (latches).

- Always blocks allow powerful statements
  - if .. then .. else
  - case

- Use always blocks only if it makes your job easier
Always Statement is not Always Practical...

Both statements describe the same multiplexer

In this case, the always block is more work
module sevensegment (input [3:0] data, 
output reg [6:0] segments);
always @ ( * )               // * is short for all signals
  case (data)                // case statement
    4'd0: segments = 7'b111_1110;   // when data is 0
    4'd1: segments = 7'b011_0000;   // when data is 1
    4'd2: segments = 7'b110_1101;
    4'd3: segments = 7'b111_1001;
    4'd4: segments = 7'b011_0011;
    4'd5: segments = 7'b101_1011;
    // etc etc
    default: segments = 7'b000_0000;   // required
  endcase
endmodule
The case Statement

- Like `if .. then .. else` can only be used in always blocks

- The result is combinational only if the output is defined for all cases
  - Did we mention this before?

- Always use a `default` case to make sure you did not forget a case (which would infer a latch)

- Use `casez` statement to be able to check for don’t cares
  - See book page 202, example 4.28
Non-blocking and Blocking Statements

**Non-blocking**

```verilog
always @ (a)
begin
    a <= 2'b01;
    b <= a;
    // all assignments are made here
    // b is not (yet) 2'b01
end
```

- Values are assigned at the end of the block.
- All assignments are made in parallel, process flow is not-blocked.

**Blocking**

```verilog
always @ (a)
begin
    a = 2'b01;
    // a is 2'b01
    b = a;
    // b is now 2'b01 as well
end
```

- Value is assigned immediately.
- Process waits until the first assignment is complete, it **blocks** progress.
Why use (Non)-Blocking Statements

- There are technical reasons why both are required
  - It is out of the scope of this course to discuss these

- Blocking statements allow sequential descriptions
  - More like a programming language

- If the sensitivity list is correct, blocks with non-blocking statements will always evaluate to the same result
  - It may require some additional iterations
Example: Blocking Statements

- Assume all inputs are initially ‘0’

```verilog
always @ (*)
begin
    p    = a ^ b ;        // p    = 0
    g    = a & b ;        // g    = 0
    s    = p ^ cin ;      // s    = 0
    cout = g | (p & cin) ; // cout = 0
end
```
Example: Blocking Statements

- Now a changes to ‘1’

```vhdl
always @ ( * )
begin
  p    = a ^ b ;      // p    = 1
  g    = a & b ;      // g    = 0
  s    = p ^ cin ;    // s    = 1
  cout = g | (p & cin) ; // cout = 0
end
```

- The process triggers

- All values are updated in order

- At the end, s = 1
Same Example: Non-Blocking Statements

- Assume all inputs are initially ‘0’

```verbatim
always @( *)
begin
    p <= a ^ b ;   // p = 0
    g <= a & b ;   // g = 0
    s <= p ^ cin ; // s = 0
    cout <= g | (p & cin) ; // cout = 0
end
```
Same Example: Non-Blocking Statements

- Now `a` changes to ‘1’

```plaintext
always @ ( * )
begin
  p  <= a ^ b ;  // p  = 1
  g  <= a & b ;  // g  = 0
  s  <= p ^ cin ;  // s  = 0
  cout <= g | (p & cin) ; // cout = 0
end
```

- The process triggers

- All assignments are concurrent

- When `s` is being assigned, `p` is still `0`, result is still `0`
Same Example: Non-Blocking Statements

- After the first iteration $p$ has changed to ‘1’ as well

```verilog
always @( * )
begin
    p <= a ^ b ;  // p = 1
    g <= a & b ;  // g = 0
    s <= p ^ cin ;  // s = 1
    cout <= g | (p & cin) ;  // cout = 0
end
```

- Since there is a change in $p$, process triggers again

- This time $s$ is calculated with $p=1$

- The result is correct after the second iteration
Rules for Signal Assignment

- Use `always @(posedge clk)` and non-blocking assignments (`<=`) to model synchronous sequential logic.
  ```verilog
  always @(posedge clk)
  q <= d; // nonblocking
  ```

- Use continuous assignments (`assign ...`) to model simple combinational logic.
  ```verilog
  assign y = a & b;
  ```
Rules for Signal Assignment (cont)

- Use `always @ (*)` and blocking assignments (=) to model more complicated combinational logic where the `always` statement is helpful.

- Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.
Finite State Machines (FSMs)

- Each FSM consists of three separate parts:
  - next state logic
  - state register
  - output logic
FSM Example: Divide by 3
FSM in Verilog, Definitions

module divideby3FSM (input clk, input reset, output q);

reg [1:0] state, nextstate;

parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;

- We define state and nextstate as 2-bit reg
- The parameter descriptions are optional, it makes reading easier
This part defines the state register (memorizing process)

- Sensitive to only clk, reset
- In this example reset is active when ‘1’
FSM in Verilog, Next State Calculation

// next state logic
always @ (*)
    case (state)
        S0:    nextstate = S1;
        S1:    nextstate = S2;
        S2:    nextstate = S0;
        default: nextstate = S0;
    endcase

- Based on the value of state we determine the value of nextstate

- An always .. case statement is used for simplicity.
FSM in Verilog, Output Assignments

// output logic
assign q = (state == S0);

- In this example, output depends only on state
  - Moore type FSM

- We used a simple combinational assign
module divideby3FSM (input clk, input reset, output q);

  reg [1:0] state, nextstate;

  parameter S0 = 2'b00;
  parameter S1 = 2'b01;
  parameter S2 = 2'b10;

  always @(posedge clk, posedge reset) // state register
    if (reset) state <= S0;
    else state <= nextstate;
  always @(*) // next state logic
    case (state)
      S0:  nextstate = S1;
      S1:  nextstate = S2;
      S2:  nextstate = S0;
      default: nextstate = S0;
    endcase
  assign q = (state == S0); // output logic
endmodule
What Did We Learn?

- Basics of Defining Sequential Circuits in Verilog

- **Always statement**
  - Is needed for defining memorizing elements (flip-flops, latches)
  - Can also be used to define combinational circuits

- **Blocking vs Non-blocking statements**
  - `=` assigns the value immediately
  - `<=` assigns the value at the end of the block

- **Writing FSMs**
  - Next state calculation
  - Determining outputs
  - State assignment