Digital Design Review

Lecture 1

Overview

- Digital design flow
- Synchronous sequential logic circuits vs combinational logic circuits
- Verilog recap
  - Language features and syntax
  - Registers vs Latches
  - Coding style
- The Verilog Simulator
- Design Example

The Digital Design Flow (FPGA)

- Required features and behaviour

- Plan to meet spec
- Create block level diagram

- Write Verilog code
- Incorporate IP
- Write constraints

- Convert behavioural description to circuit
- Maps to FPGA resources
- **Output**: Netlist

- Map to specific locations of FPGA
- Place and route according to constraints
- **Output**: Bitstream

How would this differ for ASIC?

Digital Logic Types

- Combinational/ Asychronous logic:
  - Outputs of the circuit rely only on current inputs
- Sequential logic:
  - Outputs of the circuit can depend on current inputs and history of inputs
  - Uses feedabck
- Synchronous logic:
  - Controlled by a clock signal, changes in circuit are aligned to a clock edge

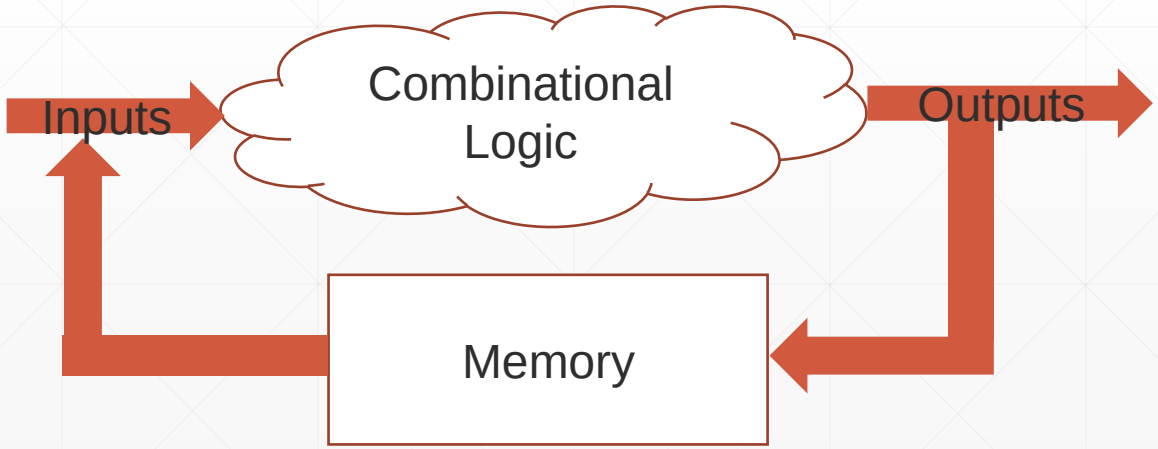
Digital Logic Types

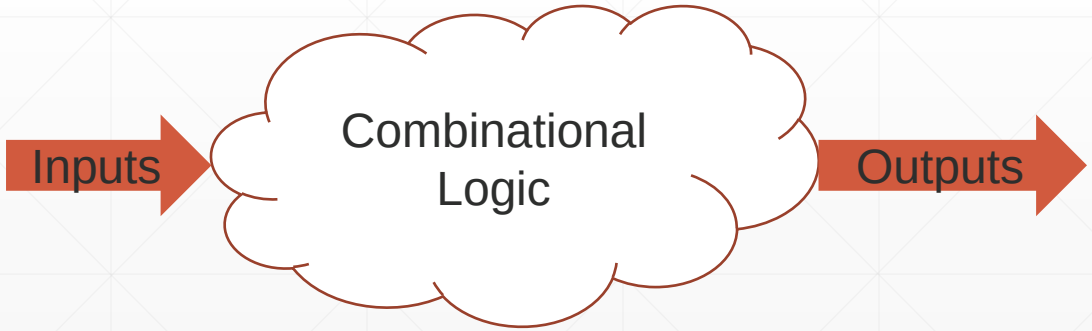
Example components of: Synchronous Sequential

- Finite State Machine (FSM)
- First In First Out (FIFO) memory

Example components of: Combinational

- OR gate
- Buffers
- Adder





- Module definition & instantiation
- Data types and declarations
  - Implicit declaration
- Operators
  - Logic
  - Arithmetic
  - Assignment
- Synchronous always block
- Combinational always block
- Conditional syntax
  - if, case, ternary statement
- Avoiding latches

Verilog Recap

Primarily focusing on Verilog-2001 syntax.

All Verilog-2001 can be compiled as SystemVerilog.

Some SystemVerilog features will be mentioned, but **SV-only code syntax is not examinable**

Module Definition and Instantiation

Module definition

```
module module_name (
    input               a,
    input       [1:0] b,
    output              c,
    output reg          d
);
endmodule
```

Module instantiation

```
module_name instance_name (
    .a(sig1),
    .b(sig2),
    .c(sig3),
    .d(sig4)
);
```

Data Types

- Standard types: `wire`, `reg`, `integer`
  - Systemverilog adds `logic` type
- 4 state types for synthesizable types: ‘0’, ‘1’, ‘x’, ‘z’

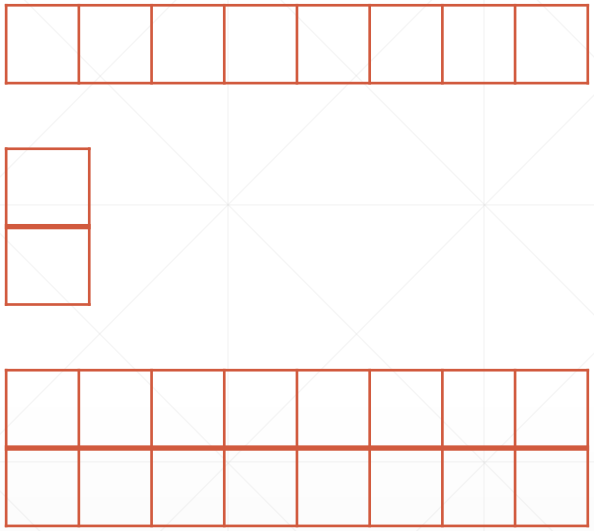
- **What does it imply if a signal has a value of z?**
  - 1: Value is too large to display
  - 2: Negative value assigned to unsigned data type
  - 3: The signal is disconnected
  - 4: The signal value is unknown

Data Types

- Many synthesizable types… `wire`, `reg`,
  - Systemverilog adds `logic` type
- 4 state types for synthesizable types: ‘0’, ‘1’, ‘x’, ‘z’
- Other types primarily used for parameters and/or simulation
  - int, real
- Implicit declaration type is one bit wide
  - Verilog won’t warn you if you assign without a declaration!
  - `assign my_undeclared_signal = 4’b1110;`
- Literal format is <bit width>’<base><value>
  - `4’b111, 4’d15, 4’hf, 4’o17`

Data Types – Vectors and Arrays

- Vector definition, SV “packed” dimension
  - `reg [7:0] a;`
- Array definition, SV “unpacked” dimension
  - `wire b [1:0];`
- Can have multiple dimension arrays:
  - `wire [7:0] c [1:0];`
  - `reg [7:0] d [3:0] [1:0];`
- Index value is arbitrary
  - `reg [6:-1] a;`
- Index definition is pretty flexible…
  - `reg [7:0] d [0:3] [2];`
- Be consistent!



Operators

- Two versions of AND, OR, NOT:
  - Logical operators &&, ||, !
  - Bit-wise operators &, |, ~
  - Equality ==
  - Using the wrong type can be difficult to debug… Choose wisely!
- Arithmetic operators
  - +, -, *, /, **
- Shift operators
  - Unsigned >>, <<,
  - Signed >>>, <<<
- Assignment operators: =, <= …

Synchronous Logic
Verilog Definition of a Flip-Flop

```
reg [3:0] count_ff, count_nxt;
always @ (posedge clk) begin
    if (reset) begin
        count_ff <= 'b0;
    end else begin
        count_ff <= count_nxt;
    end
end
```

1. reg type
2. Clocked always block
3. Reset behaviour
4. D to Q assignment
5. Only <= used (NBA)

Synchronous Logic
With Asynchronous Reset

```
reg [3:0] count_ff, count_nxt;

always @ (posedge clk, negedge reset_n)
    if (!reset_n) begin
        count_ff <= 'b0;
    end else begin
        count_ff <= count_nxt;
    end
end
```

1. reg type
2. Clocked always block
3. Reset behaviour
4. D to Q assignment
5. Only <= used (NBA)

Quick Poll: Is this reset 1) active high or 2) active low?

Strict Coding Rules for Synchronous Logic

- Every register to be inferred uses
  - _nxt for combinatorial next value
  - _ff to denote registered value
- Clocked Process
  - Registered value _ff only updated in clocked always block
  - No other logic in clocked block (besides reset or enable)
  - Clocked block uses non-blocking (<=)
  - Register input _nxt is assigned by combinational process…

Combinational Logic

- Continuous assignments

```
wire up;
assign up = 1’b1;
assign count_nxt = up? count_ff + 1’b1 : count_ff – 1’b1;
```

- Procedural assignments

```
reg [3:0] count_ff, count_nxt;
always @(*) begin
    if (up) count_nxt = count_ff + 4’b1;
end
```

- What if up is 1’b0?

Asynchronous Sequential Logic… Latches

- What is a latch?
- Inferred by synthesis if combinatorial signal not defined in all cases
- Why is inferring a latch a problem?

Avoiding Latches: Golden Rule

- *To synthesize combinational logic using an always block, **all** variables must be assigned under **all** conditions.*
- Every combinatorial always block should have default assignments

Avoiding latches

- What is wrong here?

```
wire up, down;
always @(*) begin
    if (up) begin
        count_nxt = count_ff + 4'b1;
    end else if (down) begin
        count_nxt = count_ff - 4'b1;
    end
end
```

Incorrect, DO NOT COPY

Avoiding latches

- Avoid by completing if statement:

```
wire up, down;
always @(*) begin
    if (up) begin
        count_nxt = count_ff + 4'b1;
    end else if (down) begin
        count_nxt = count_ff - 4'b1;
    end else begin // final else to avoid latch
        count_nxt = count_ff;
    end
end
```

Avoiding latches

- Avoid by using default case item in case statements:

```
always @(*) begin
    case ({down, up})
       2’b01: count_nxt = count_ff + 4’b1;
       2’b10: count_nxt = count_ff - 4’b1;
     default: count_nxt = count_ff;
    endcase
end
```

Avoiding latches

- Avoid by using default assignments:

```
always @(*) begin
    count_nxt = count_ff;
    if (up)   count_nxt = count_ff + 4'b1;
    if (down) count_nxt = count_ff - 4'b1;
end
```

Verilog is a Hardware Description Language

REMEMBER: How descriptions are interpreted is subjective.

```
reg [3:0] count_ff, count_nxt;

always @ (posedge clk) begin
    if (reset) begin
        count_ff <= ‘b0;
    end else begin
        count_ff <= count_nxt;
    end
end

 always @ (*) begin
    if (enable) begin
        count_nxt = count_ff + 4’d1;
    end else begin
        count_nxt = count_nxt;
    end
end
```

Think about your resources when designing logic

A standard D-flop will exist in almost any library.

```
always @ (posedge clk or posedge reset) begin
    if (reset) begin
        reg_ff <= ‘b0;
    end else begin
        reg_ff <= reg_in;
    end
end
```

But not all configurations will exist. For example Xilinx FPGA registers can do either async set or async reset but not both at the same time.

```
always @ (posedge clk or posedge set or posedge reset) begin
    if (reset) begin
        reg_ff <= 1‘b0;
    end else if(set) begin
        reg_ff <= 1’b1;
    end else begin
        reg_ff <= reg_in;
    end
end
```

Strict Coding Rules

- Separate the combinatorial process(es)
  - All _nxt values assigned in always @(*) block, never assign _ff types here
  - Only use blocking assignments (=)
- A variable can only be assigned value by single process
  - What happens if multiple always @(*) blocks assign to one variable?
- All variables given value in all paths in code, avoid latches

Strict Coding Rules

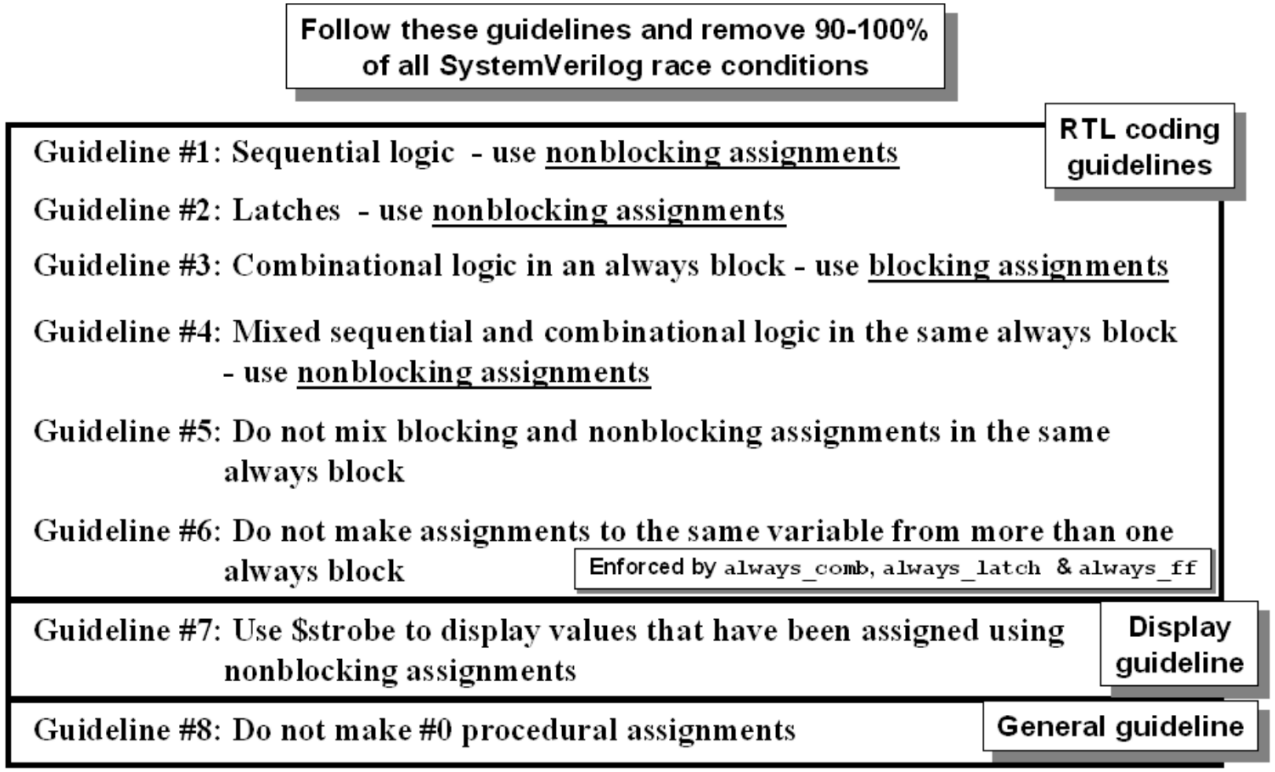


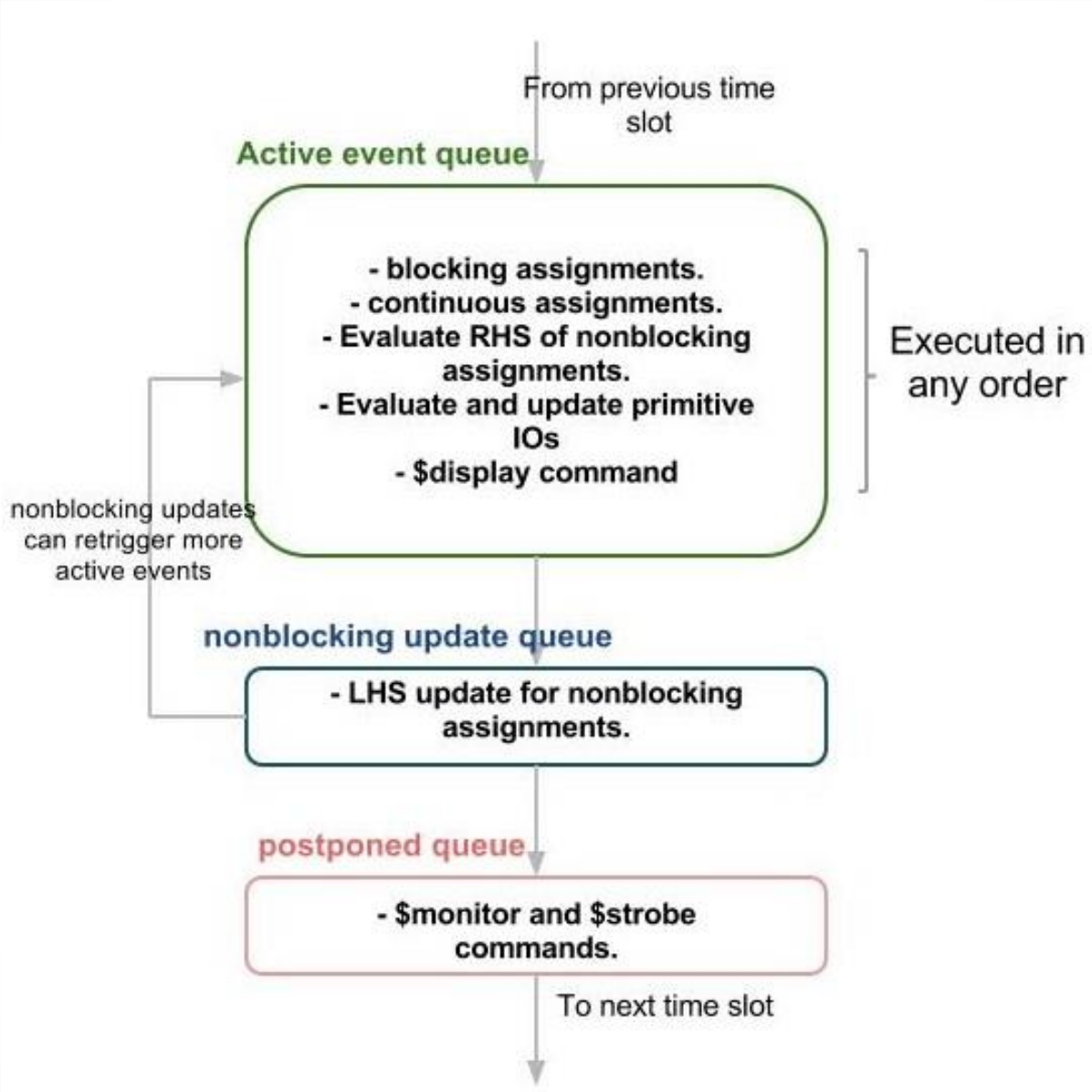
Figure 3 - Sunburst Design's - 8 Coding Guidelines to Avoid Verilog Race Conditions

Strict Coding Rules

- These apply to describing hardware-synthesisable Verilog only
- Will see non-synthesisable code constructs for testbench use only…
- In the case of non-synthesisable code these rules do not all apply.

Other Useful Verilog Constructs…

- Constants: localparam, parameter
- generate
- function & task
- for loop, while loop
- New with SV (useful but NOT required for this course):
  - enum & typedef, struct, interface, always_ff & always_comb



Verilog Simulator

Source for diagram **here**

Further reading **here**

Non-Blocking vs Blocking
Which should be used to describe shift register?

Option 1:
Non-Blocking Assignment

```
shift_reg[0] <= in;

shift_reg[1] <= shift_reg[0];

shift_reg[2] <= shift_reg[1];
```

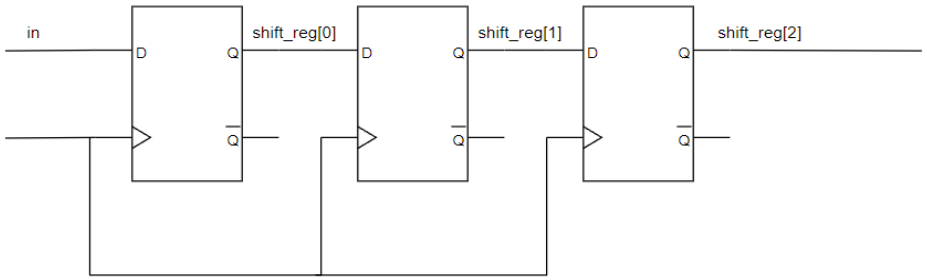
Option 2:
Blocking Assignment

```
shift_reg[0] = in;

shift_reg[1] = shift_reg[0];

shift_reg[2] = shift_reg[1];
```

NBA vs BA

Shift register using nonblocking assignment

```
always@(posedge clk)
  begin
     if(reset) begin
       shift_reg[2:0] <= 3'd0;
     end else begin
        shift_reg[0] <= in;
        shift_reg[1] <= shift_reg[0];
        shift_reg[2] <= shift_reg[1];
     end
  end
```



Shift register using blocking assignment

```
always@(posedge clk)
  begin
       if(reset) begin
          shift_reg[2:0] = 3'd0;
       end else begin
          shift_reg[0] = in;
          shift_reg[1] = shift_reg[0];
          shift_reg[2] = shift_reg[1];
     end
  end
```