ENGN4213/6213 Digital Systems and Microprocessors

Semester 1, 2023

Week 3, Lecture A:

Verilog (extension)

Australian National University

Activities (Week-3)

- Lab-2: Implementing designs onto FPGA and Essential sequential designs
  - *Implementation of the Full Adder*
  - *Overflow counter*
  - *Clock division using an overflow counter*
  - *Flexible clock division and heartbeat generators*
- Tutorial-2: Verilog
  - *Structural vs Behavioral*
  - *Blocking vs. Non-blocking*
  - *Verilog examples*

Recap of Week-2 Lecture-B

Introducing Verilog

Verilog basics:

Verilog is a hardware description language (HDL) used for designing the behavior (what they do) and structure (what they are made of) of digital circuits.

- Verilog structure – Module declaration
- Verilog Syntax
  - Data values: `0, 1 , x , z`
  - Data types: `wire, reg, input, output, inout, parameter`
  - Number Specification: `<size>'<base format><number>`
  - Operators: *Arithmetic, Relational, Bit-wise, Logical, Reduction, Shift, Conditional*
- Behavioral Design
  - `always` procedural blocks
  - Blocking and non-blocking assignments

What's in this lecture

Verilog (extension)

- Procedural statements in behavioural design
  - `if-else`
  - `case`
  - Looping statements: `for`, `while`, `repeat`, `forever`
- Hierarchical Design
  - Module Instantiation and parameterization
- Simulation: Test Benches, `initial` block
- Coding rules for synthesis
- Verilog examples of common circuits

Resources

- Pre-recorded videos lectures
  - VL4: Introducing Verilog
  - VL5: Verilog (continued)
- Wakerly (5th edition)
  - Chapter 5: Verilog HDL
  - Chapter 6 to 8: Verilog code examples of combinational circuits
  - Chapter 10: Verilog code examples of latches and flip-flops
  - Chapter 11: Verilog code examples of counters and shift registers
- ENGN3213 old Reading Brick from 2008 - Chapter 5 (Introduction to Verilog)

Recap of Week-2 Lecture-B

wire, reg

wire

- represent physical wires (nets) used for interconnecting components, typically in *structural* Verilog
- used to design *combinational* logic
- must be driven by continuous assignment statement using `assign` keyword or by connecting to module ports with instantiation

reg

- represent variables that hold state in *behavioral* Verilog
- used to design both *combinational* and *sequential* logic
- must be assigned inside procedural blocks (`always`)

```
module mux2x1(a,b,sel,out_w,out_r);

  input wire a, b, sel;
  output wire out_w;
  output reg out_r;

  // structural design
  assign out_w = sel ? b : a;

 // behavioral design
  always @(*) begin
      if (sel) out_r = b;
      else out_r = a;
  end

endmodule
```

➢ Input and inout ports are of type `wire`.

➢ Output can be `wire` (to connect other components) or `reg` (if we need to make the output readable).

➢ Port types defaults to `wire`

Recap of Week-2 Lecture-B

Behavioural Design: `always` block

- Verilog supports behavioral designs (<u>both sequential and combinational</u>) through blocks of procedural code
- Keyword `always` introduces these blocks.

```
always @(sensitivity list) begin
        … <procedural statements> …
end
```

  - Sensitivity list is a list of signals can be named explicitly or a special one-fits-all using (*).
    - Combinational logic is typically defined using an **`always @(*)`** block.
    - Sequential logic can be defined using:

      `always @ (posedge` *`<signal_name>`*`)` or **`always @ (negedge`** *`<signal_name>`*`)`
  - Statements following `always` are executed whenever one of the signals in the sensitivity list changes value.
    - Left hand terms in assignments inside an always block are always of type `reg`.
    - Procedural statements execute sequentially, but `always` block itself executes concurrently with other concurrent statements in the same module.
    - Procedural statements are written in a style similar to software programming languages

Recap of Week-2 Lecture-B

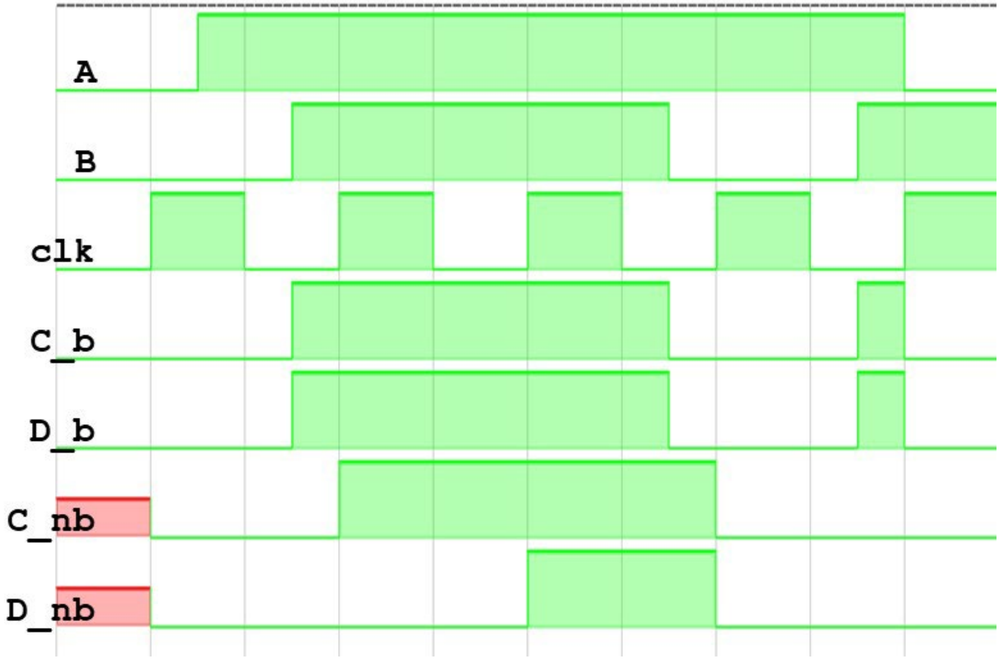
Behavioural Design: Blocking and Non-blocking assignments

- Blocking assignments (" = ") : right-hand side expression is evaluated and assigned to the left-hand side variable immediately, blocking the execution of subsequent procedural statements in the same `always` block
    - Typically, used to model combinational logic with procedural blocks of type `always @(*)`
- Non-blocking assignments (" <= "): right-hand side expression is evaluated but the value is not assigned to the left-hand side variable until all other assignments in the same `always` block have been evaluated.
    - "old" value remains current until the whole block has been evaluated
    - Typically, used to model sequential logic and concurrent data transfers with procedural blocks of type `always @(posedge clk ..)`
- It is recommended that blocking and nonblocking assignments not be mixed in the same procedural (`always)` block.

Recap of Week-2 Lecture-B

Behavioural Design: Blocking and Non-blocking assignments

```
module block_vs_nonblock(
    input wire A,B,clk,
    output reg C_b,D_b,C_nb,D_nb
);
    always @(*) begin
    //combinational block
        C_b=A&B;
        D_b=C_b&B;
    end
    always @(posedge clk) begin
    //sequential block
        C_nb<=A&B;
        D_nb<=C_nb&B;
    end
endmodule
```



Recap of Week-2 Lecture-B

Structural vs. Behavioural

HDL is used for designing the behavior (what they do) and structure (what they are made of) of digital circuits.

Structural	Behavioural
creates a design by explicitly defining its components and their interconnections (bottom-up approach).	creates a design by specifying its functionality without explicitly defining its components (top-down approach).
Typically based on `wire` data types, `assign` statements, gates, and module instantiations.	Typically based on `reg` data types, procedural blocks (`always`) with constructs like `if…else`, `case` etc.
Statements executed concurrently i.e., order doesn't matter	Blocking assignments are executed in order, whereas non-blocking assignments are executed concurrently
Provides a clear picture of the hierarchy and connectivity	Provides a high-level, abstract view of the functionality
Allows for precise timing information and control.	Supports timing modeling but may be less precise.
More opportunities for optimization and efficiency	May be less efficient due to higher-level descriptions
Requires a deeper understanding of the underlying hardware and circuit design.	Allows rapid prototyping and algorithm development.

Behavioral Design

if-else

- `if-else` is used to design a circuit with an event occurring only if a particular condition is verified, otherwise some other event will be executed
- Syntax:

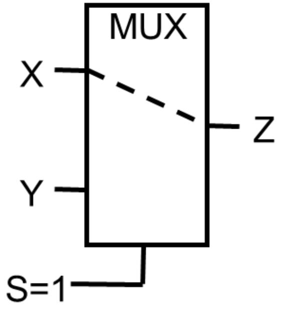
```
if (condition) begin
        <statement events if condition true>;
end
else begin
        <statement events if condition false>;
end
```

- `begin … end` constructs are used if there are multiple statements within following `if` or `else`
- `if-else` statements can be nested

Behavioral Design

if-else

```
module mux2x1(
   input wire X, Y, S;
   output reg Z);

   always @(*) begin
      if(S==1) Z=X;
      else Z=Y;
   end

endmodule
```



```
module example(
      input wire a, b,
      output reg out);
      always @(*) begin
         if (a) begin
               out = 1'b1;
         end else if (b) begin
               out = 1'b0;
         end else begin
               out = 1'bZ;
         end
      end
endmodule
```

a	b	Out
1	x	1
0	1	0
0	0	z

Behavioral Design

case

- Similar to `if-else` statements, but used to test multiple conditions on the same variable.
- Syntax:

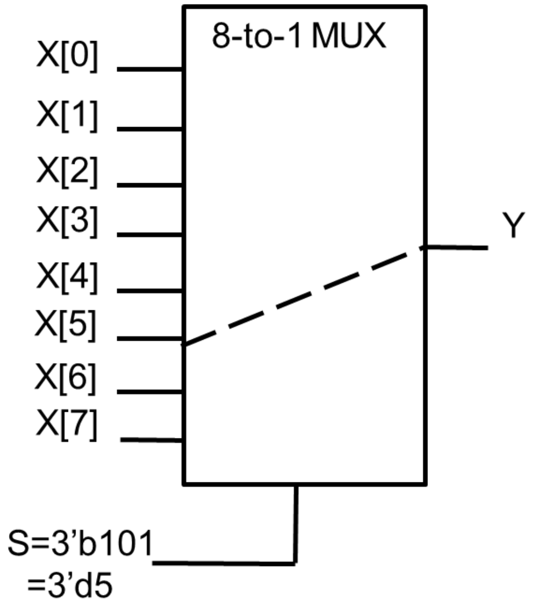
```
case (case_variable)
  choice#1  : <statement#1>;
  choice#2  : <statement#2>;
  …
  default : <statement if none of the above occur>;
endcase
```

- Finds the first one of the choices that matches the `case_variable`'s value, and executes the corresponding procedural statement
- `begin … end` constructs are used if there are multiple statements for any one case
- *Full* `case`: listed choices are all-inclusive (including default case) to avoid inferred latches
- *Parallel* `case`: listed choices are mutually-exclusive for faster and less expensive synthesis

Behavioral Design

case

Example: 8-to-1 Multiplexer



```
module eight_to_1_MUX (
     input wire [7:0] X,
     input wire [2:0] sel,
     output reg Y);
  always @(*) begin
     case(sel)
          3'd0 :Y = X[0];
          3'd1 :Y = X[1];
          3'd2 :Y = X[2];
          3'd3 :Y = X[3];
          3'd4 :Y = X[4];
          3'd5 :Y = X[5];
          3'd6 :Y = X[6];
          3'd7 :Y = X[7];
          default : Y=1'bx;
     endcase
   end
endmodule
```

Behavioral Design

`if-else` **vs.** `case`

Example: 4-to-1 Multiplexer

```
always @(*) begin
    if(S==2'b00) Z=X[0];
    else if(S==2'b01) Z=X[1];
    else if(S==2'b10) Z=X[2];
    else if(S==2'b11) Z=X[3];
    else Z=X[0];
end
```

Implemented using four 2:1 multiplexers

```
always @(*) begin
  case(S)
    2'b00 : Z = X[0];
    2'b01 : Z = X[1];
    2'b10 : Z = X[2];
    2'b11 : Z = X[3];
    default : Z=X[0];
  endcase
end
```

Implemented using a single 4:1 multiplexer. So, more efficient in terms of hardware synthesis.

Behavioral Design

Inferred Latches

If you use `if-else` and `case` statements in combinational designs, it is important to ensure that signals are assigned a value in every possible conditions i.e., **Ensure** *full* `case` statement or that every `if` statement has an `else` clause.

- Inferred latches: When a signal is not assigned a value in every possible path through a behavioral block, the synthesizer may create a latch to hold the previous value of the signal.
- Inferred latches are generally undesirable because they can lead to unpredictable behavior, wastage of hardware resources, and can cause timing and power issues.

```
always @(*) begin
     // If reset is high, reset data_out to 0
     if (reset) begin data_out <= 8'h00; end
     // If enable is high, assign data_in to data_out
     else if (enable) begin data_out <= data_in; end
     // If neither reset nor enable is high, assign a default value to data_out
     else begin data_out <= 8'h00; end // without default statement: Inferred latch issue
end
```

Behavioral Design

Looping statements: `for`, `while`, `repeat`, `forever`

- Carry out an operation repeatedly as long as
  - **`for`**: a looping index meets the requirements
  - **`while`**: a particular logical expression remains true
  - **`repeat`**: a counter is not exhausted
  - **`forever`**: the circuit is running (used commonly in test benches)
- Syntax:

```
for (index=initial; expression; index=next)
   //index is typically integer variable
   begin <procedural statements> end
```

```
while (logical_expression)
   begin <procedural statements> end
```

```
repeat (integer_expression)
    begin <procedural statements> end
```

```
forever
   begin <procedural statements> end
```

Behavioral Design

Looping statements: `for`, `while`, `repeat`, `forever`

Example: 8-bit comparator module using a `for` loop. *(not an efficient design, just to illustrate working of `for` loop)*

```
module Vrcomp (
    input [7:0] X, Y,
    output reg gt);      // gt will be 1 if X > Y

    integer i;

    always @ (X, Y) begin
        gt = 0; // starts out as 'not greater'
        for ( i=0 ; i<=7 ; i=i+1 ) begin
            if (X[i] & ~Y[i]) gt = 1;
            else if (~X[i] & Y[i]) gt = 0;
            else gt=0;
        end
    end
endmodule
```

Behavioral Design

Looping statements: `for`, `while`, `repeat`, `forever`

Looping statements are dangerous:

In source file code, we will **limit their use** or avoid them altogether! This is an important practical difference with computer programming, where loops are very common.

- They can create *non-synthesizable* designs, particularly when the number of repetitions in the loop is unknown a priori.
- `repeat`, `while`, `forever` cannot be used to synthesize combinational logic, only for sequential designs with timing control
- They often introduce feedback
- Designs may blow out in terms of hardware space requirements (not particularly useful for hardware designs.)
- Somewhat useful in writing test benches: `forever #5 clk=~clk;`

```
//Non-synthesizable!
…
always @(*) begin
  while (c < d) begin
          c=c+2;
          a=a+1;
  end
end
..
```

Hierarchical Design

Hierarchical design

Structure of a Verilog file

Verilog 1995 standard

```
module <module name>(
    // All IO names are defined here
 <port names>
 );

 // We can optionally declare parameters here
 parameter <parameter_name>=<default_value>

 // Complete the definition of the module IOs here
 <direction> <data_type> <size> <port_name>

 // Behavioral/Structural statements

endmodule
```

```
module mux2x1(a,b,sel,out);
  input a, b, sel;
  output out;
  assign out = sel ? b : a;
endmodule
```

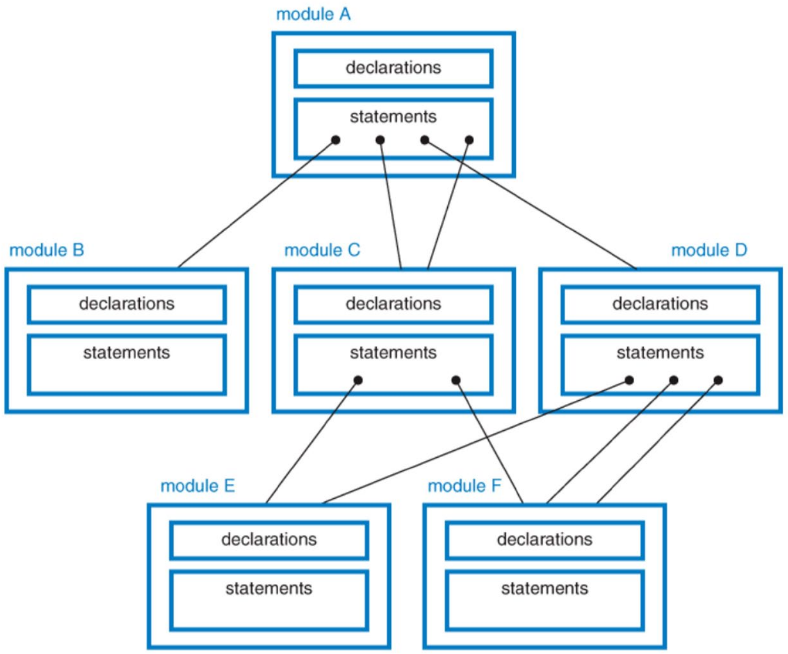
Verilog 2001 standard

```
module #(
    // We can optionally declare parameters here
    parameter <parameter_name>=<default_value>
    )
    <module_name>(
    // All IO are defined here
     <direction> <data_type> <size> <port_name>
    );

    // Behavioral/Structural statements

endmodule
```

```
module mux2x1(
  input a, b, sel
  output out
);
  assign out = sel ? b : a;
endmodule
```

Hierarchical design

- Basic unit of design and programming in Verilog is a module: a text file containing declarations and statement.
- The hardware is said to be modeled by a single module or by a collection of modules working together.
- In a hierarchical design, higher-level modules can ***instantiate*** (call) lower-level modules
  - A higher-level module may use a lower-level module multiple times, and multiple top-level modules may use the same lower-level one.
  - Facilitates the *re-use of code* for modules performing similar operations
  - Makes the overall design easier to read and understand



Hierarchical design

Module Instantiation

- Let's refer to the *higher-level calling module* as the *parent* and the *lower-level called module* as the *child*. Let `<c…>` identify the child signals and `<p…>` identify the parent signals
- Instantiation syntax:

$$\texttt{ChildModule\_name}\ \textit{instance\_identifier}\ \underbrace{\texttt{(.c1(p1), .c2(p2), …)}}_{Port-association\ list}\texttt{;}$$

  - `instance_identifier` should be unique within a parent module
  - Port-association list: Each child module signal is listed preceded by a period (.) and followed by the associated parent module signal in brackets.
    - Paired signals need not have the same name
- When the physical hardware is created during synthesis, each instance of a module is a separate piece of hardware that has the inputs and outputs specified in its instantiation
  - All instance statements execute concurrently in a parent module
  - Each synthesized instance of a child module operates independently of other instances

Hierarchical design

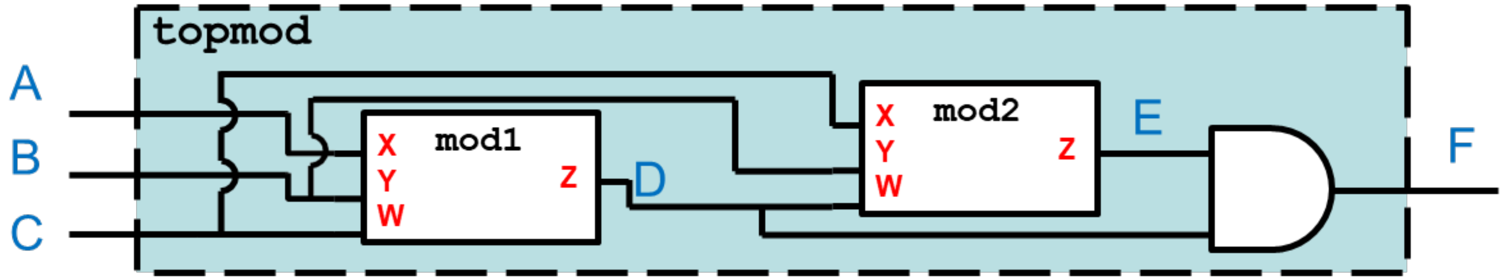
Module Instantiation

Child module

```
module childmod  (
    input wire X, Y, W,
    output reg Z);
    always @(*)
         Z = (X & Y)|(X & W);
endmodule
```

Parent module

```
module topmod (
    input wire A, B, C,
    output wire F);
    wire D, E;
    childmod mod1(.X(A),.Y(B),.W(C),.Z(D));
    childmod mod2(.X(C),.Y(B),.W(D),.Z(E));
    assign F = D & E;
endmodule
```



Hierarchical design

Module Instantiation: Parameterization

- Modules can be parameterized to handle inputs and outputs of any width
- In the child module, keyword `parameter` defines a constant that can be used throughout the module. The value inside this child module declaration is the default value of the parameter.

```
module <ChildModule_name> #(parameter <par_name>=<default_value>, …)
            (// port declarations);
                    // behavioral/structural) statements
endmodule
```

- When instantiating the child module inside a parent module, the syntax is

```
ChildModule_name #(.par_name(par_value),…)
                    instance_identifier (.c1(p1), .c2(p2), … );
```

  - parameter value inside the `#()` placeholder will override the default parameter value for that particular instance only.
  - If child module has just one parameter, syntax can be

```
ChildModule_name #(par_value) instance_identifier (.c1(p1), .c2(p2), … );
```

Hierarchical design

Module Instantiation: Parameterization

Example 1:

Child module

```
module childmodule
      #(parameter BUS_WIDTH=8,
        parameter DATA_WIDTH=16)
      (input wire [BUS_WIDTH-1:0] addr,
       output reg [DATA_WIDTH -1:0] Y
      // other port declarations
      );

      // Behavioral/structural statements

endmodule
```

Parent module

```
module topmodule
      (
      input wire [15:0] data_in,
      output reg [31:0] data_out
      // other port declarations
      );

      // Behavioral/structural statements

      // Module instantiation
      childmodule #(.BUS_WIDTH(16), .DATA_WIDTH(32)) child1 (.addr(data_in),
                                                            .Y(data_out),…);

      // Behavioral/structural statements

endmodule
```

Hierarchical design

Module Instantiation: Parameterization

Example 2:

Child module:

A variable-width adder (default width 2 bits)

```
module cooladder #(parameter WID=2)(
   input wire cin,
   input wire [WID-1:0] X,
   input wire [WID-1:0] Y,
   output reg cout,
   output reg [WID-1:0] Z
   );
   reg sum [WID:0];
   always @(*) begin
          sum = X+Y+cin;
          Z=sum[WID-1:0];
          cout=sum[WID];
   end
endmodule
```

Examples of instantiations:

```
//2-bit adder (default WID)
cooladder add2 (.cin(p1),.X(p2),
.Y(p3),.cout(p4),.Z(p5));

//4-bit adder (WID=4)
cooladder #(4) add4 (.cin(p1),
.X(p2),.Y(p3),.cout(p4),.Z(p5));

//8-bit adder (WID=8)
cooladder #(8) add8 (.cin(p1),
.X(p2),.Y(p3),.cout(p4),.Z(p5));
```

Hierarchical design

Module Instantiation: Parameterization

Child module:

A variable-width adder (default width 2 bits)

```
module cooladder #(parameter WID=2)(
   input wire cin,
   input wire [WID-1:0] X,
   input wire [WID-1:0] Y,
   output reg cout,
   output reg [WID-1:0] Z
   );

   reg sum [WID:0];

   always @(*) begin
          sum = X+Y+cin;
          Z=sum[WID-1:0];
          cout=sum[WID];
   end
endmodule
```

This instantiation has issues: *module instantiation should be outside of procedural blocks*

```
module smartadder(
      input wire Cin,
      input wire [3:0] x, y,
      output wire Cout,
      output wire [3:0] z);

    always @(*)begin
      if (X < 3 and Y < 3)
          cooladder add2
          (.cin(Cin),.X(x),.Y(y),
          .cout(Cout), .Z(z));
      else
          cooladder #(4) add4
          (.cin(Cin),.X(x),.Y(y),
          .cout(Cout), .Z(z));
    end
endmodule
```

Simulation

Simulation

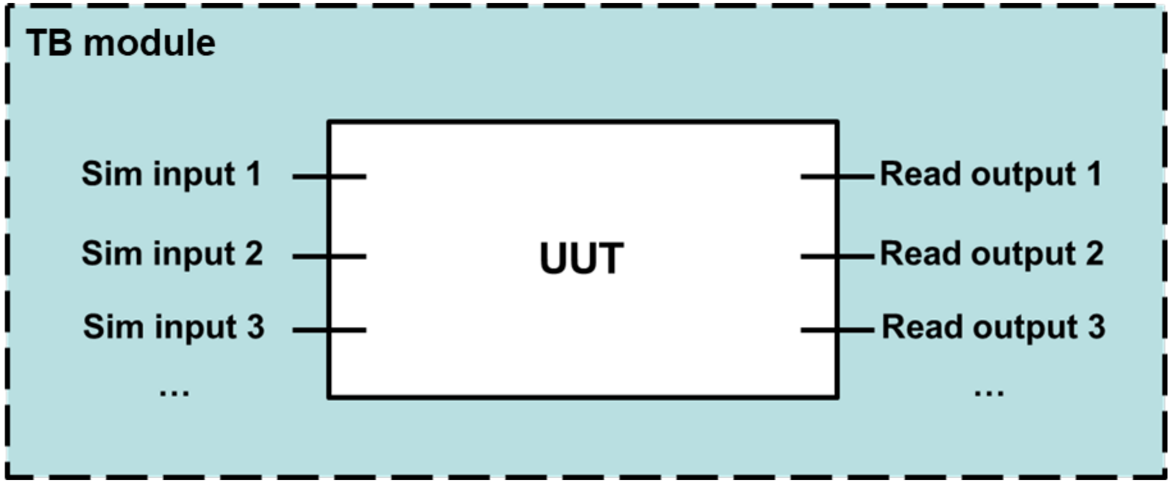
Simulator is used to observe the operation of a Verilog model for error correction and optimizing performance before committing to hardware synthesis.

- Simulator operation begins at a simulation time of zero
  - At this time, the simulator initializes all signals to a default value of "`x`".
  - It also initializes any signals or variables for which initial values have been declared explicitly
- Next, the *simulator begins the execution of all the concurrent statements* in the design based on their time-based event list and a sensitivity matrix
  - When an event is processed, the simulator updates the affected signals and variables and schedules any new events that may result from the change
  - Simulator continues to advance the simulation time and process events until the simulation is complete, as determined by the designer or a predetermined stopping condition.
- Generates reports and outputs that summarize the behavior of the design over time, such as waveform traces or timing diagrams.

Simulation

Test Bench

- A test bench is a top module with no external inputs or outputs.
- It instantiates the module under test.
- It provides a description of test signals which will be fed to the inputs of the design Unit Under Test (UUT) and lists the test outputs which will be read



Simulation

Test Bench: Common Components

- Time Scale specifies the time unit and time precision.

  Syntax: `` `timescale time-unit / time-precision ``

  - `time-unit`: indicates the new default units that will be associated with all time values
  - `time-precision`: specifies the time granularity with which the simulator will operate

- Time Delay specifies the delay in units of the timescale before a statement is executed

  Syntax: `#(delay_steps);`

  - With a time-unit of 10 ns, the expression #2.3 will have a delay of 23.0 ns.

```
…
forever begin
   #1 sysclk=1;     /* Create a clock source (sysclk) for simulation purposes. The signal sysclk switches
   #1 sysclk=0;     value between 0 and 1 at every 10 ns with a time-unit of 10 ns */
end
…
```

- `$display(<string>)`: used to print a text-based output in the terminal console as the simulation is running. (format of the string is given in C-language-like syntax)
- `$finish;` forces a simulation to terminate

Simulation

Test Bench: `initial` block

- Procedural block (like the always block) used in test benches for simulation purposes. It is generally not synthesizable (there are exceptions).
- Typically used to initialize variables and specify signal waveforms during simulation.
- Syntax:

```
        initial begin
                … <procedural statements> …
         end
```

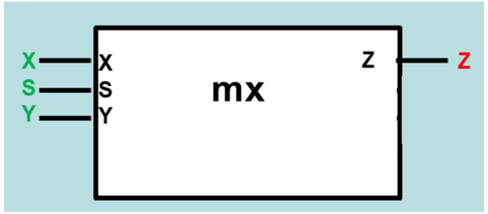
  - Unlike always, it has no associated sensitivity list.
- `initial` block is executed at the beginning of the simulation (at time 0)
- Test benches can have multiple initial blocks

Simulation

Test Bench: Typical structure

Essential test bench structure:

```
`timescale 1ns/1ps        //specifies simulation time steps (default 1ns)

module TB_mux1;            //Testbench is a normal module, just with no i/o

   reg X;
   reg S;                  //Inputs meant for the UUT
   reg Y;                  /* Note that X,Y,S are of type reg since they are to be used in
                           initial (procedural) blocks to describe the test conditions */
   wire Z;                  //Connected to output from the UUT: note it is of type wire

   mux mx(.X(X), .S(S), .Y(Y), .Z(Z)); //instantiation of module under test

   initial begin
         #1X  =  1'b0;     //procedural block (initial) to specify signal changes
         #1Y  =  1'b1;
         #1S  = 1'b1;
         ...
         #1Y  = 1'b0;
         #1S  = 1'b0;
   end
endmodule
```



Coding Rules

Coding Rules for synthesis

General rules:

- `wire` data types is used to connect components in structural Verilog
  - **`wire` must be driven using `assign` keyword** or by connecting to module ports
- `reg` data types are used to hold values in behavioral Verilog
  - **`reg` must be assigned inside procedural (`always`) blocks**
- **Input** and inout ports **must be type `wire`**.
- Output can be `wire` or `reg`.
- Port types defaults to wire
- Specify the bit width of all numbers you use.
  - It's good practice and avoids involuntary hardware wastage
  - Example:

```
a[3:0] = b[3:0] + 8; // is 8 a 32-bit integer (default)?
a[3:0] = b[3:0] + 4'd8; // better
```

Coding Rules for synthesis

General rules:

- No `initial` blocks
- No delay specifications
- Do not make assignments to the same variable in multiple always blocks or multiple assign statements: multiple driver error.
- Avoid looping statements whenever possible
  - Think that the synthesizer will work only if it can "unroll the loop" or if the number of repetitions are known in prior.
- In conditional statements, prefer `case` to nested `if-else` if there are multiple cases to prevent the synthesis of slow gate chains in hardware.
- Do not mix blocking and nonblocking assignments in the same always block.

Coding Rules for synthesis

Combinational logic:

- Value assignments inside the design module must be done with **all inputs on the RHS**, and all **outputs on the LHS**
- No feedback loops permitted
- Structural: Use `wires` and **continuous assignment** `assign` for simple combinational logic
- Behavioral:
  - Use a **level sensitivity** list **`always @(*)`** for procedural blocks
  - Use **only blocking assignments** "**=**" for procedural assignments
- `if-else` and `case` statements must ensure that signals are assigned a value in every possible conditions to avoid inferred latches.
  - full `case` statement: listed choices are all-inclusive including default case
  - every `if` statement has an `else` clause.

Coding Rules for synthesis

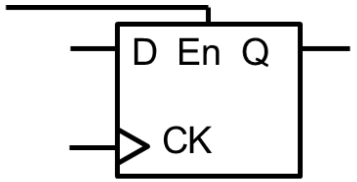
Sequential logic:

- Behavioral design:
  - Use **an edge sensitivity list** with a single clock entry, e.g., `always @(posedge sysclk)`
    - clock triggers the execution of procedural blocks, but the clock does not appear inside the block
  - **Use only non-blocking assignments** "**<=**“ for procedural statements
- Any **external signals must be synchronized** to avoid metastability
- If statements do not need to be complete
- Feedback loops are fine

Common Circuits

Common Circuits

D Flip-flop



With enable (En)

```
module Dff_En(
     input wire Clk,
     input wire D, En
     output reg Q);
       always @(posedge Clk)
         if(En)
           Q <= D;
endmodule
```

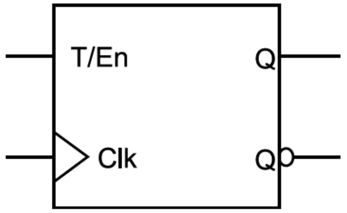
Without enable

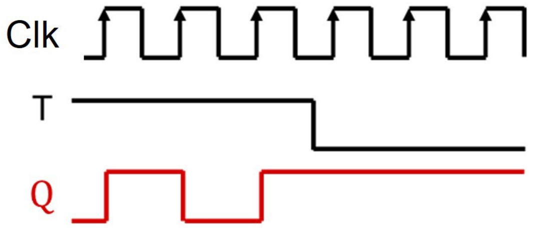
```
module Dff_noEn(
     input wire Clk,
     input wire D,
     output reg Q);
       always @(posedge Clk)
           Q <= D;
endmodule
```

Common Circuits

T Flip-flop

```
module T_ff(
      input wire Clk,
      input wire T
      output reg Q);
         always @(posedge Clk)
           if(T)
              Q <= ~Q;
endmodule
```

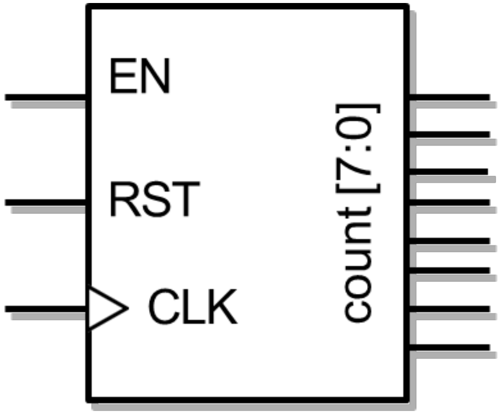




Common Circuits

8-bit up-counter

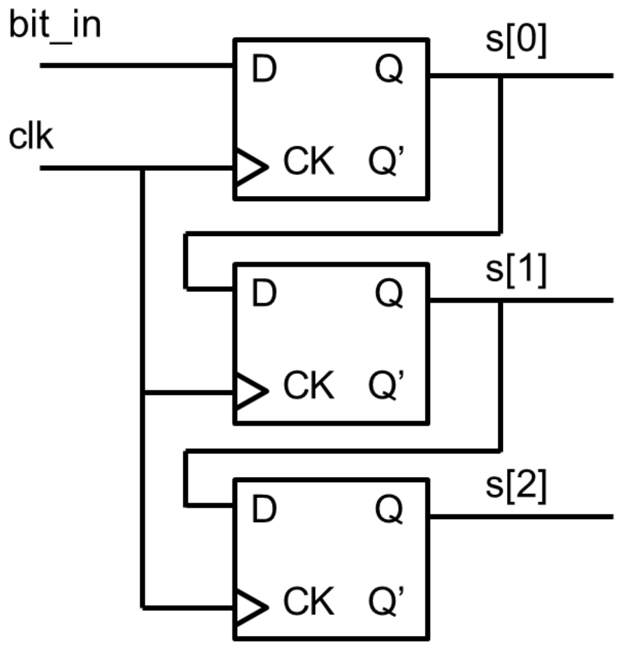
```
module counter8(
  input wire CLK, RST, EN,
  output reg [7:0] count);

    always @(posedge CLK) begin
         if(RST) count <= 8’h00;
         else if(EN) count <= count + 1’b1;
    end

endmodule
```



Common Circuits

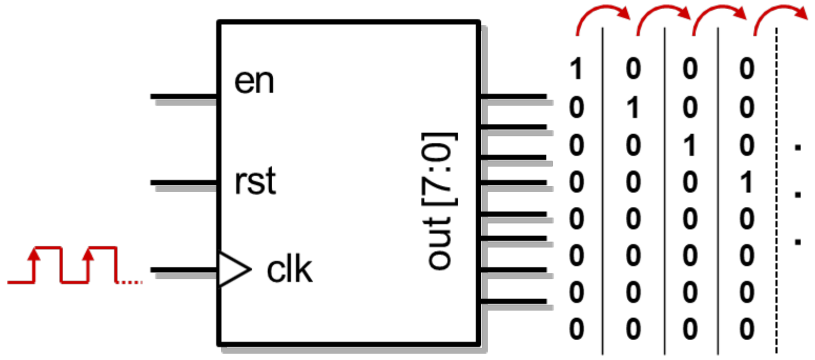
3-bit serial-in parallel-out (SIPO) shift register

```
module siporeg(
  input wire clk, reset, bit_in,
  output reg [2:0] s);
    always @(posedge clk) begin
        if(reset) s <= 3'b000;
        else begin
            s[0] <= bit_in;
            s[1] <= s[0];
            s[2] <= s[1];
        end
    end
endmodule
```



Common Circuits

8-bit “one-hot” counter



```
module one_hot_cnt8(
   input wire clk, rst, en,
   output reg [7:0] out);

      always @(posedge clk) begin
            if(rst) out <= 8’b00000001;
            else if(en) begin
            // {}: concatenation operator combines two or more operands to form a larger vector
                  out <= {out[6],out[5],out[4],out[3],out[2],out[1],out[0],out[7]};
            end
      end
endmodule
```

Test your understanding

Q1

Mis-matched buses:

```
wire a;
assign a=2'b10;
```

A. Syntax Error

B. MSB will drop

C. LSB will drop

Q1

Mis-matched buses:

```
wire a;
assign a=2'b10;
```

A. Syntax Error

B. MSB will drop

C. LSB will drop

Q2

Find the issues

```
module module1 (
 input wire X,S,Y,
 output wire Z);
 always @(*) begin
      if(S==1) Z=X;
      else Z=Y;
  end
endmodule
```

A. input

B. output

C. always

D. if

E. else

F. other

Q2

Find the issues

```
module module1 (
 input wire X,S,Y,
 output wire Z);
 always @(*) begin
      if(S==1) Z=X;
      else Z=Y;
  end
endmodule
```

A. input

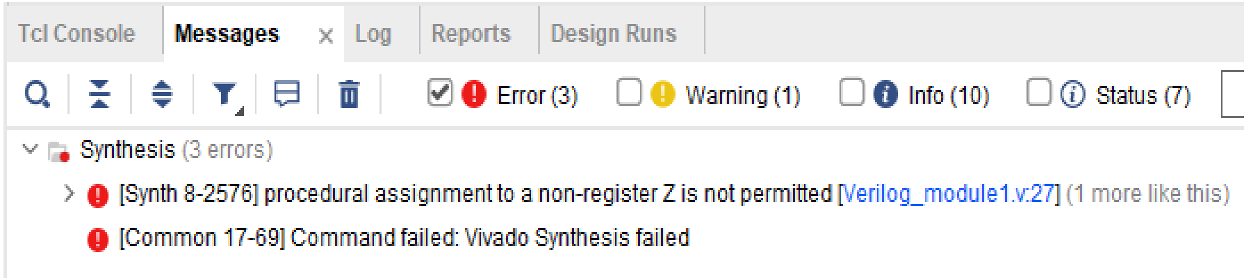
B. output

C. always

D. if

E. else

F. other



```
Synthesis (3 errors)
  [Synth 8-2576] procedural assignment to a non-register Z is not permitted [Verilog_module1.v:27] (1 more like this)
  [Common 17-69] Command failed: Vivado Synthesis failed
```

Q3

Find the issues

```
module module2 (
 input reg X,
 input wire clk,
 output wire Z);

 always @(posedge clk) begin
       assign Z<=X;
  end

endmodule
```

A. Input reg X

B. Input wire clk

C. Output wire Z

D. always sensitivity list

E. assign

F. other

Q3

Find the issues

```
module module2 (
 input reg X,
 input wire clk,
 output wire Z);
 always @(posedge clk) begin
      assign Z<=X;
  end
endmodule
```

A. Input reg X

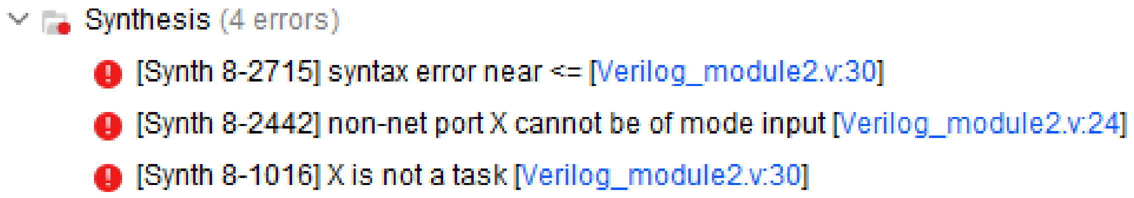
B. Input wire clk

C. Output wire Z

D. always sensitivity list

E. assign

F. other



```
Synthesis (4 errors)
    [Synth 8-2715] syntax error near <= [Verilog_module2.v:30]
    [Synth 8-2442] non-net port X cannot be of mode input [Verilog_module2.v:24]
    [Synth 8-1016] X is not a task [Verilog_module2.v:30]
```

Q4

Find the issues

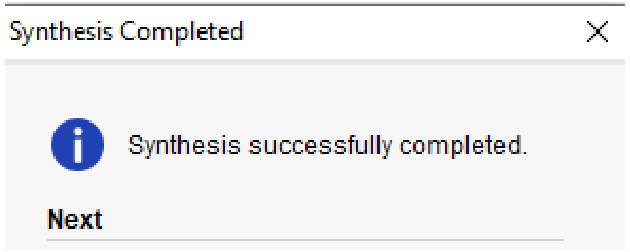
```
module module3 (
 input wire X,S,Y,
 output reg Z);
 always @(posedge S) begin
      if(S) Z<=X;
      else Z<=Y;
  end
endmodule
```

Q4

Find the issues

```
module module3 (
 input wire X,S,Y,
 output reg Z);

 always @(posedge S) begin
       if(S) Z<=X;
       else Z<=Y;
  end

endmodule
```

No error in synthesis, but behaviour might not be what we expect


Synthesis Completed

Synthesis successfully completed.

Next

always block executes at the positive edge of the clock, but what is the level of S at this edge? At the transition? 0? 1?

Q5

Find the issues

```
module module4 (
 input wire A,B,C,
 output reg Z);

 module3 inst
     (.X(A),.S(B),.Y(C),.Z(Z))

endmodule
```

A. Input

B. output

C. instantiation

D. other

```
module module3 (
 input wire X,S,Y,
 output reg Z);

 always @(posedge S) begin
      if(S) Z<=X;
      else Z<=Y;
  end

endmodule
```

Q5

Find the issues

```
module module4 (
 input wire A,B,C,
 output reg Z);

 module3 inst
     (.X(A),.S(B),.Y(C),.Z(Z))

endmodule
```

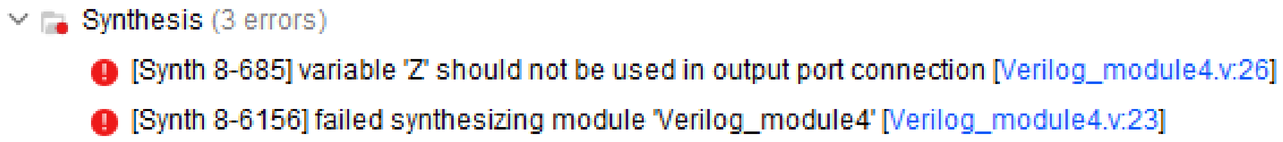
A. Input

B. output

C. instantiation

D. other

```
module module3 (
 input wire X,S,Y,
 output reg Z);

 always @(posedge S) begin
      if(S) Z<=X;
      else Z<=Y;
  end

endmodule
```



```
Synthesis (3 errors)
  [Synth 8-685] variable 'Z' should not be used in output port connection [Verilog_module4.v:26]
  [Synth 8-6156] failed synthesizing module 'Verilog_module4' [Verilog_module4.v:23]
```

Q6

Find the issues

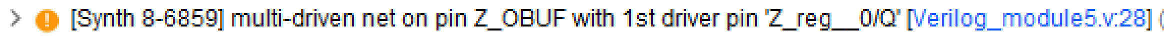
```
module module5 (
  input wire [1:0] X,
  input wire S1, S2, clk, clk2,
  output reg Z);
    always @(posedge clk) begin
      if(S1==1) Z=X[0];
    end
    always @(posedge clk2) begin
      if(S2==1) Z=X[1];
    end
endmodule
```

Q6 Find the issues

```
module module5 (
  input wire [1:0] X,
  input wire S1, S2, clk, clk2,
  output reg Z);
    always @(posedge clk) begin
      if(S1==1) Z=X[0];
    end
    always @(posedge clk2) begin
      if(S2==1) Z=X[1];
    end
endmodule
```

Critical warnings during synthesis

- Multiple assignments to Z
- Should not use blocking assignment in sequential block



```
[Synth 8-6859] multi-driven net on pin Z_OBUF with 1st driver pin 'Z_reg__0/Q' [Verilog_module5.v:28] (
```

Q7

Find the issues

```
module module6 (
  input wire [1:0] X,
  input wire [1:0] S1,
  input wire [1:0] S2,
  output reg Z);

    always @(*) begin
      if(S1==2’b10) Z=X[0];
        case(S2) begin
         2’b00: Z=X[1];
         2’b01: Z= X[1]+1;
        endcase
    end

endmodule
```

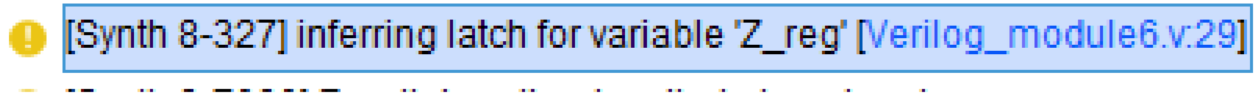
Q7

Find the issues

```
module module6 (
  input wire [1:0] X,
  input wire [1:0] S1,
  input wire [1:0] S2,
  output reg Z);
    always @(*) begin
      if(S1==2’b10) Z=X[0];
        case(S2) begin
         2’b00: Z=X[1];
         2’b01: Z= X[1]+1;
        endcase
    end
endmodule
```

Critical warnings during synthesis due to inferred latch issue

- No else clause for if
- No default clause for case



```
[Synth 8-327] inferring latch for variable 'Z_reg' [Verilog_module6.v:29]
```

Concluding Remarks

- Covered sufficient concepts and features of Verilog to carry out moderately complex designs
    - Refer the resources for more complex and advanced features which we have not discussed.
- Write your own code and study already written examples (plenty of them in your textbook).
- The very essential things to remember:
    - Verilog can do both simulation and synthesis, and some constructs which work in simulations do not work in synthesis
    - There are general rules on how to do sequential and combinational designs. Although they are not absolutes, you should be able to avoid common mistakes if you follow them.

THANK YOU

Acknowledgement

These presentation slides are the modified version of slides prepared by Dr. Jihui (Aimee) Zhang based on the original version by Dr. Nicolo Malagutti

Contact:

Amy Bastine
School of Engineering
ANU College of Engineering, Computing and Cybernetics
The Australian National University

Office: B147, Brian Anderson Building (Bldg. 115)
Email: amy.bastine@anu.edu.au