Verilog HDL Review

October 3, 2023
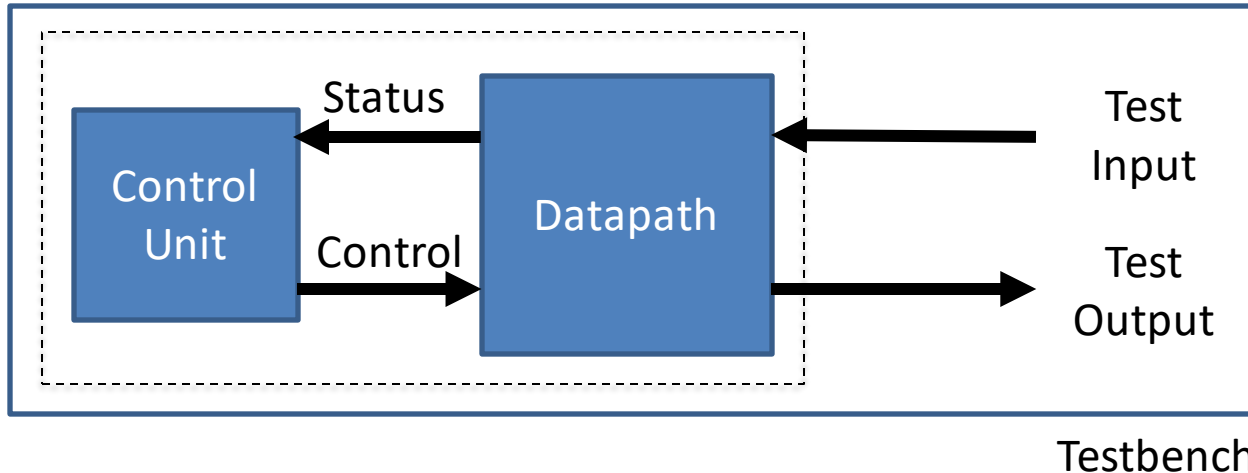
Ahmet Can Mert

ahmet.mert@iaik.tugraz.at

Hardware Description Language (HDL): Overview of a Digital System

- Datapath
    - Performs data processing
- Control Unit (Finite State Machine)
    - Generates control signals to control the datapath
- Testbench
    - Used to verify the functional correctness of the design (for simulation)


Testbench

Hardware Description Language (HDL): Definition

- It is **NOT** a programming language.

- It is used to describe any digital circuit.
  - i.e., you can describe circuit elements and connections between them.

- Many languages available for RTL Modeling: VHDL, Verilog, SystemVerilog
  - **Verilog** is simple and similar to C
  - Verilog has more than half of the world digital design market
  - Many free resources are available:
    - http://www.asic-world.com/verilog/veritut.html
    - https://www.chipverify.com/verilog/

Hardware Description Language (HDL): Logic and Memory

- Register Transfer Level: An abstract level used to describe the operation of synchronous digital circuits.
  - Logic Functions (computation)
    - Any combinatorial computation
  - Memory (update)
    - Flip-Flop: edge sensitive
    - Latch: level sensitive (WE WILL NOT USE)
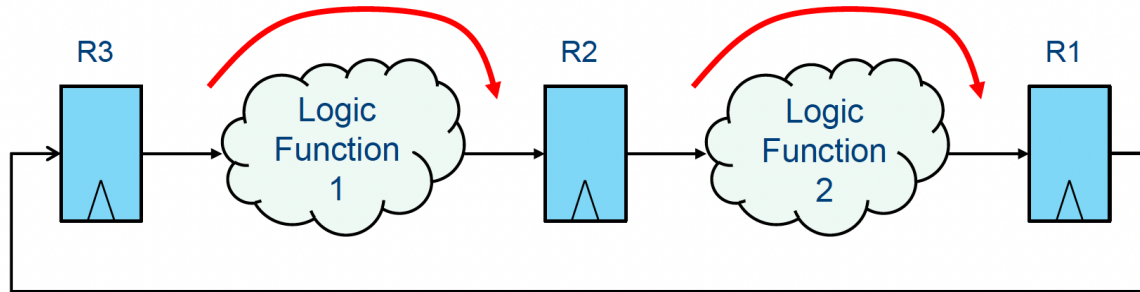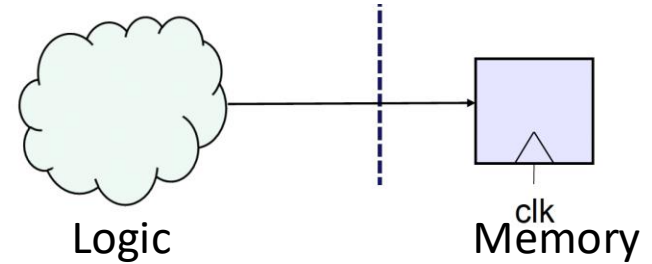
Verilog Operators

- Logical, arithmetic and conditional operators

Syntax	Operation
~	Bit-wise negation
&	AND
!&	NAND
|	OR
~|	NOR
^	XOR
^~ or ~^	XNOR

```
i.e.,
c = ~a;
c = a & b;
```

Syntax	Operation
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulo
<<	Left shift
>>	Right shift

```
i.e.,
c = a + b;
c = a >> 2;
```

Syntax	Operation
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

```
i.e.,
c = (a==b) ? 1 : 0;
```

Verilog Operators

- Operator precedence is important.

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	logical bit-wise reduction reduction reduction reduction reduction reduction
+ -	unary (sign) plus unary (sign) minus	arithmetic arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
* / %	multiply divide modulus	arithmetic arithmetic arithmetic
+ -	binary plus binary minus	arithmetic arithmetic
<< >>	shift left shift right	shift shift
> >= < <=	greater than greater than or equal to less than less than or equal to	relational relational relational relational
== !=	logical equality logical inequality	equality equality
=== !==	case equality case inequality	equality equality
&	bit-wise AND	bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	bit-wise bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

* Table from: https://class.ece.uw.edu/cadta/verilog/operators.html

Verilog Operators

- Operator precedence is important.

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	logical bit-wise reduction reduction reduction reduction reduction reduction
+ -	unary (sign) plus unary (sign) minus	arithmetic arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
* / %	multiply divide modulus	arithmetic arithmetic arithmetic
+ -	binary plus binary minus	arithmetic arithmetic
<< >>	shift left shift right	shift shift
> >= < <=	greater than greater than or equal to less than less than or equal to	relational relational relational relational
== !=	logical equality logical inequality	equality equality
=== !==	case equality case inequality	equality equality
&	bit-wise AND	bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	bit-wise bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

$$c0 = a + b << 2;$$

$$a = 4, \quad b = 1$$
$$c0 = (5 << 2) = 20$$

* **Table from:** https://class.ece.uw.edu/cadta/verilog/operators.html

Verilog Operators

- Operator precedence is important.

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	logical equality	equality
!=	logical inequality	equality
===	case equality	equality
!==	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
^~ or ~^	bit-wise XNOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Table from: https://class.ece.uw.edu/cadta/verilog/operators.html

$$c0 = a + b << 2;$$

$$a = 4, \ b = 1$$
$$c0 = (5 << 2) = 20$$

$$c1 = a + (b << 2);$$

$$a = 4, \ b = 1$$
$$c1 = 4 + (1<<2) = 8$$

Verilog Operators - Example

- Using + operator to design an adder
  - 4-bit inputs and 5-bit output

Carry	Sum

= A + B

- { } operator is used to concatenate signals
  - `Carry` is 1-bit
  - `Sum` is 4-bit

  {Carry,Sum} = A + B;

- {{}} operator is used to repeat a signal
  - Repeating `Carry[0]` bit four times

  {Carry[0],Carry[0],Carry[0],Carry[0]} --> {4{Carry[0]}}

Language Element - Literals

- Literals are constant numbers (in binary, octal, decimal and hexadecimal).
- Literals as represented as:

    `<size>`'**`<signed><radix>`**`value`

Language Element - Literals

- Literals are constant numbers (in binary, octal, decimal and hexadecimal).
- Literals as represented as:

  <size>'<signed><radix>value

  - e.g., `A = 16'd12987;`
    - `16` indicates the bit size of the signal
    - `d` indicates decimal representation is used.
      - `b` or `B` -> binary
      - `o` or `O` -> octal
      - `d` or `D` -> decimal
      - `h` or `H` -> hexadecimal
    - No `s` after ' shows it is unsigned
  - e.g., `B = 20;`
    - If bit size, sign and radix are not specified, default representation is 32-bit unsigned decimal

Language Element - Literals

- Literals are constant numbers (in binary, octal, decimal and hexadecimal).
- Literals as represented as:

    <size>'<signed><radix>value

  - e.g., `A = 16'd12987;`
    - `16` indicates the bit size of the signal
    - `d` indicates decimal representation is used.
      - `b` or `B` -> binary
      - `o` or `O` -> octal
      - `d` or `D` -> decimal
      - `h` or `H` -> hexadecimal
    - No `s` after ' shows it is unsigned
  - e.g., `B = 20;`
    - If bit size, sign and radix are not specified, default representation is 32-bit unsigned decimal.

Language Elements – Data Types

- Bus definition
    - n-bit data type declaration
        - `reg [n-1:0] a;`
        - `wire [n-1:0] a;`
    - Part selection:

```
reg [31:0] a,b;
wire [16:0] sum;
assign sum = a[15:0]+ b[15:0];
```

Language Elements – Data Types

- Bus definition
  - n-bit data type declaration
    - `reg [n-1:0] a;`
    - `wire [n-1:0] a;`
  - Part selection:

```
reg [31:0] a,b;
wire [16:0] sum;
assign sum = a[15:0]+ b[15:0];
```

- Verilog is case-sensitive
  - `reg [3:0] Rega, RegA;`

- Net/Variable names cannot start with a number
  - `reg [3:0] 2num;` ✗
  - `reg [3:0] num2;`

Language Elements – Module and ports

- Verilog module declaration starts with `module` and ends with `endmodule`.

```
module module_name (<port list>);

// Module content

endmodule
```

- Module ports (by default, ports are considered as type `wire`):
  - `input`
  - `output`
  - `inout`

Language Elements – Module and ports

- Example:

```
module add_unit (a,b,c);

input [3:0] a,b;
output[4:0] c;

assign c = a+b;

endmodule
```

```
module add_unit (input [3:0] a,b,
                 output[4:0] c);

assign c = a+b;

endmodule
```

Language Elements – Statements

- Statements are used to drive nets
    - There are two different methods to define Statements:
        - assign
            - Combinational (Blocking: =)
        - always
            - Combinational (Blocking: =)
            - Sequential (Non-blocking: <=)

Language Elements – `assign` Statement

- It is used to drive `output` and `wire` types. It is used to define combinational circuit parts.
- Order of `assign` statements is not important.
- When a variable at the RHS of assign statement changes, LHS is re-evaluated.
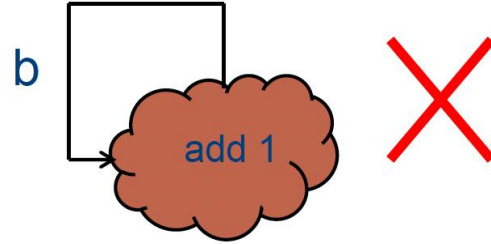
Language Elements – `assign` Statement

- It is used to drive `output` and `wire` types. It is used to define combinational circuit parts.
- Order of `assign` statements is not important.
- When a variable at the RHS of assign statement changes, LHS is re-evaluated.

```verilog
module Module_1 (A, B, C, D, E);

    input [3:0]  A, B, C;
    output[11:0] D, E;

    wire [4:0] t1, t2, t3;

    assign t1 = A + B;
    assign t2 = A - B;
    assign t3 = (C << 1);
    assign D = (t1 * t2) + t3;
    assign E = A * C;

endmodule
```

Language Elements – `assign` Statement

- It is used to drive `output` and `wire` types. It is used to define combinational circuit parts.
- Order of `assign` statements is not important.
- When a variable at the RHS of assign statement changes, LHS is re-evaluated.

```
module Module_1 (A, B, C, D, E);

    input [3:0]  A, B, C;
    output[11:0] D, E;

    wire [4:0] t1, t2, t3;

    assign t1 = A + B;
    assign t2 = A - B;
    assign t3 = (C << 1);
    assign D = (t1 * t2) + t3;
    assign E = A * C;

endmodule
```

- When A changes, the new values of $t1$, $t2$ and E are computed concurrently

Language Elements – `assign` Statement

- It is used to drive `output` and `wire` types. It is used to define combinational circuit parts.
- Order of `assign` statements is not important.
- When a variable at the RHS of assign statement changes, LHS is re-evaluated.

```
module Module_1 (A, B, C, D, E);

    input [3:0]  A, B, C;
    output[11:0] D, E;

    wire [4:0] t1, t2, t3;

    assign t1 = A + B;
    assign t2 = A - B;
    assign t3 = (C << 1);
    assign D = (t1 * t2) + t3;
    assign E = A * C;

endmodule
```

- When A changes, the new values of t1, t2 and E are computed concurrently

- Since t1 and t2 are updated, D is re-evaluated

Language Elements – `assign` Statement

- It is used to drive `output` and `wire` types. It is used to define combinational circuit parts.
- Order of `assign` statements is not important.
- When a variable at the RHS of assign statement changes, LHS is re-evaluated.

```
module Module_1 (A, B, C, D, E);

    input [3:0]  A, B, C;
    output[11:0] D, E;

    wire [4:0] t1, t2, t3;

    assign t1 = A + B;
    assign t2 = A - B;
    assign t3 = (C << 1);
    assign D = (t1 * t2) + t3;
    assign E = A * C;

endmodule
```

- When `A` changes, the new values of `t1`, `t2` and `E` are computed concurrently

- Since `t1` and `t2` are updated, `D` is re-evaluated

- `D` does not update any net

Language Elements – `assign` Statement

- No combinatorial loops

```
wire [7:0] b;
assign b = b + 1;
```

- No combinatorial loops between signals in a clock cycle

Language Elements – `always` Statement

- It is used to drive `reg` types. It is used to define both combinational and sequential parts.
- A **sensitivity list** is defined for each `always` block.
  - It has signals that trigger the execution of the logic defined in `always` block

- Syntax:

```
always @(sensitivity list)
begin
    <your logic>
end
```

Clock-sensitive synchronous design

```
always @(posedge clk)
begin
    <your logic>
end
```

Combinational design

```
always @(*)
begin
    <your logic>
end
```

Language Elements - Conditional Assignments

- Three ways to do conditional assignment.
- Method1: if/else if/else

```
always @ (*)
begin
    if(S==1'b0)
        Y = I0;
    else
        Y = I1;
end
```


- Method2: case/endcase

```
always @ (*)
begin
    case(S)
    1'b0: Y = I0;
    1'b1: Y = I1;
    endcase
end
```

- Method3:

```
always @ (*)
begin
    Y =(S) ? I1 : I0;
end
```

Language Elements - Conditional Assignments

- Three ways to do conditional assignment.
- Method1: if/else if/else

```verilog
always @ (*)
begin
    if(S==1'b0)
        Y = I0;
    else
        Y = I1;
end
```

For combinational circuits, never use incomplete conditional assignments!

- Method2: case/endcase

```verilog
always @ (*)
begin
    case(S)
    1'b0: Y = I0;
    1'b1: Y = I1;
    endcase
end
```

- Method3:

```verilog
always @ (*)
begin
    Y =(S) ? I1 : I0;
end
```

A Sample Design: Full Adder

- module/endmodule is used to define the design
- A unique name must be given to each design in a project

```
module Full_Adder




endmodule
```
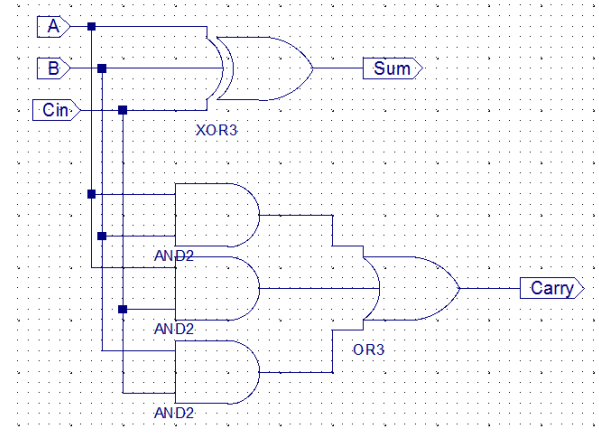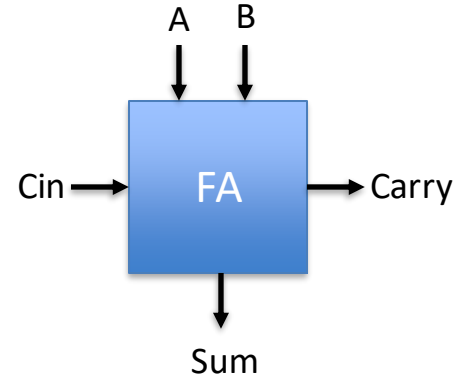
A Sample Design: Full Adder

- All I/Os must be defined in argument list. Order of the list is not important
- The polarity of the ports (input or output) must be defined at the beginning.

```
module Full_Adder (A, B, Cin, Sum, Carry);

    input A, B, Cin;
    output Sum, Carry;




endmodule
```
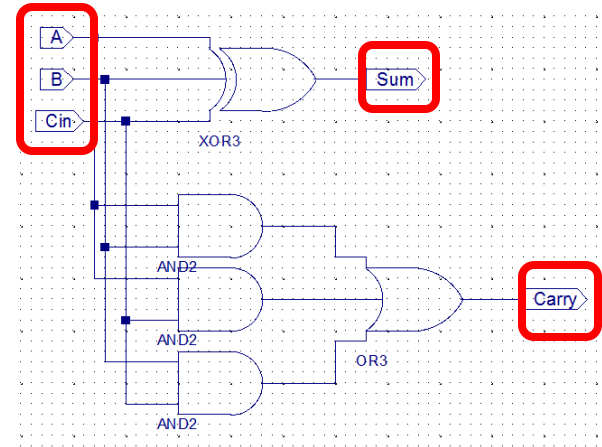
A Sample Design: Full Adder

- There may be some interconnections between gates
- Gates are connected with nets which are defined as `wire`

```verilog
module Full_Adder (A, B, Cin, Sum, Carry);

    input A, B, Cin;
    output Sum, Carry;

    wire w1, w2, w3;




endmodule
```
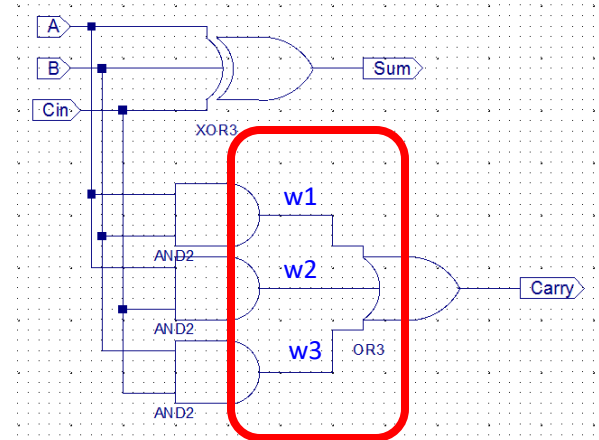
A Sample Design: Full Adder

- After the module is created and all I/Os and nets are defined, the interconnections may be defined.

```verilog
module Full_Adder (A, B, Cin, Sum, Carry);

    input A, B, Cin;
    output Sum, Carry;

    wire w1, w2, w3;

    assign w1 = A & B;
    assign w2 = A & Cin;
    assign w3 = B & Cin;


endmodule
```
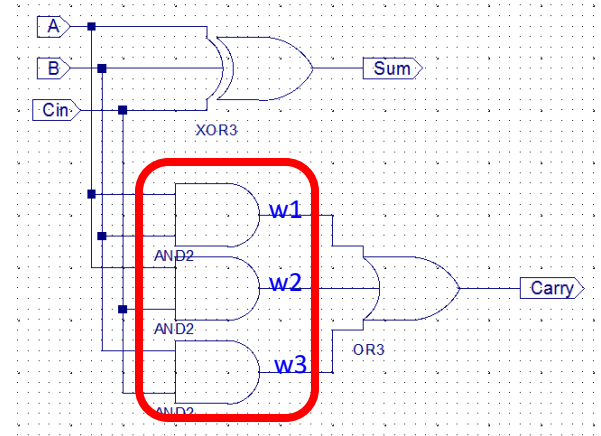
A Sample Design: Full Adder

- After the module is created and all I/Os and nets are defined, the interconnections may be defined.

```verilog
module Full_Adder (A, B, Cin, Sum, Carry);

    input A, B, Cin;
    output Sum, Carry;

    wire w1, w2, w3;

    assign w1 = A & B;
    assign w2 = A & Cin;
    assign w3 = B & Cin;
    assign Carry = w1 | w2| w3;
    assign Sum = A ^ B ^ Cin;

endmodule
```
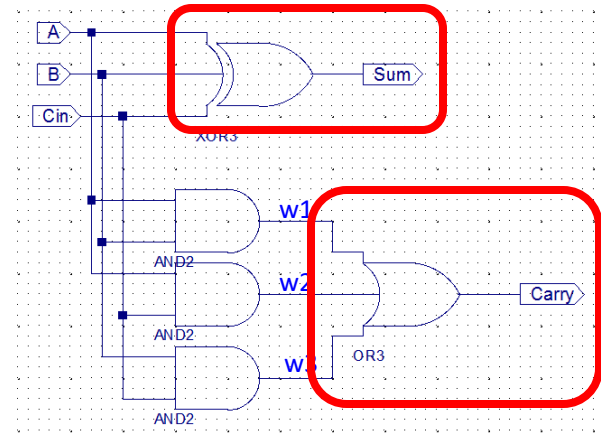
A Sample Design: Full Adder

- All interconnections do not have to be defined seperately.
- // (line comment) or /* */ (block comment) may be used to add comments.

```verilog
module Full_Adder (A, B, Cin, Sum, Carry);

    input A, B, Cin; //inputs
    output Sum, Carry; /*outputs*/

    assign Carry = (A & B)|(A & Cin)|(B & Cin);
    assign Sum = A ^ B ^ Cin;

endmodule
```
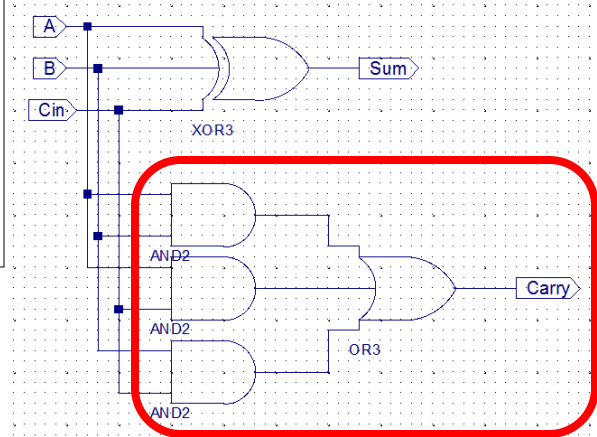
A Sample Design: 3-bit Ripple Carry Adder

- Hierarchical Design
  - A module may be used as a sub-module of another module.
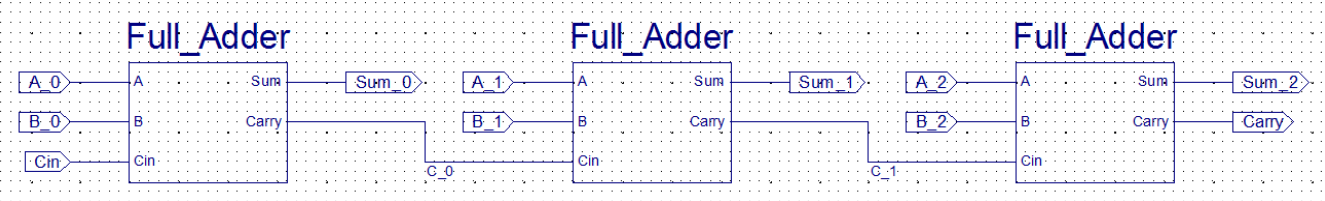
A Sample Design: 3-bit Ripple Carry Adder

- Hierarchical Design
  - A module may be used as a sub-module of another module.
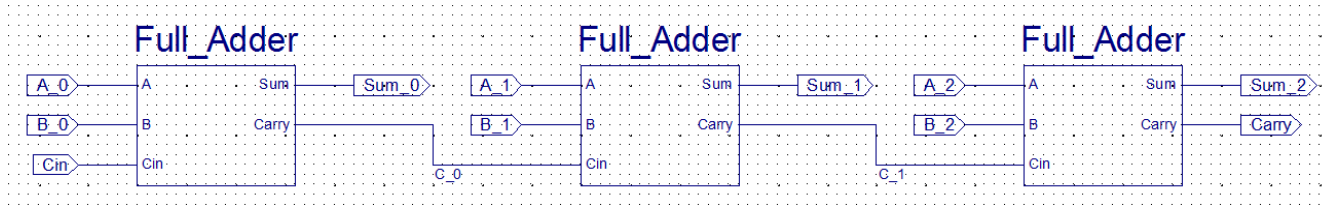

```verilog
module RCA3 (A, B, Cin, S, Carry);
    input [2:0] A, B;
    input Cin;
    output [2:0] S;
    output Carry;
    wire C_0, C_1;

    Full_Adder FA0 (A[0], B[0], Cin, S[0], C_0);
    Full_Adder FA1 (.A(A[1]), .B(B[1]), .Cin(C_0), .S(S[1]), .Carry(C_1));
    Full_Adder FA2 (.S(S[2]), .B(B[2]), .Cin(C_1), .Carry(Carry), .A(A[2]));
endmodule
```

A Sample Design: 3-bit Ripple Carry Adder

- Module Instantiation
  - Firstly, the name of module, which is instantiated, is specified.
  - Then, a unique name is given to each module.

```
Full_Adder FA0 (<ports>);
```

A Sample Design: 3-bit Ripple Carry Adder

- Module Instantiation
  - Firstly, the name of module, which is instantiated, is specified.
  - Then, a unique name is given to each module.

    ```
    Full_Adder FA0 (<ports>);
    ```

  - Finally, I/O connections of the module are defined. There are two methods:

A Sample Design: 3-bit Ripple Carry Adder

- Module Instantiation
  - Firstly, the name of module, which is instantiated, is specified.
  - Then, a unique name is given to each module.

    ```
    Full_Adder FA0 (<ports>);
    ```

  - Finally, I/O connections of the module are defined. There are two methods:
    - Method1: Signal names are written inside the parenthesis. Signals have to be written in the same order of submodule argument list.

      ```
      Full_Adder FA0 (A[0], B[0], Cin, S[0], C_0);
      ```

A Sample Design: 3-bit Ripple Carry Adder

- Module Instantiation
  - Firstly, the name of module, which is instantiated, is specified.
  - Then, a unique name is given to each module.

    ```
    Full_Adder FA0 (<ports>);
    ```

  - Finally, I/O connections of the module are defined. There are two methods:
    - Method1: Signal names are written inside the parenthesis. Signals have to be written in the same order of submodule port list.

      ```
      Full_Adder FA0 (A[0], B[0], Cin, S[0], C_0);
      ```

    - Method2: Signals and ports are connected explicitly. Order of the signals is not important in this method.

      ```
      Full_Adder FA0 (.A(A[0]), .B(B[0]), .Cin(Cin), .S(S[0]),
      .Carry(C_0));
      ```

Language Element – Generate Block

- A generate block is used to instantiate a module multiple times
    - It must be coded in a module

```
genvar i;

generate
    for(i=0; i<N; i=i+1)
    begin
        <module instantiation>
    end
endgenerate
```

Language Element – Generate Block

- C

```
for(int i=0; i<4; i++) {
    s = Full_Adder(…);
}
```

- Verilog

```
genvar i;
generate
    for(i=0; i<4; i=i+1) begin
        Full_Adder fa(…);
    end
endgenerate
```
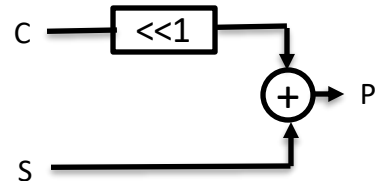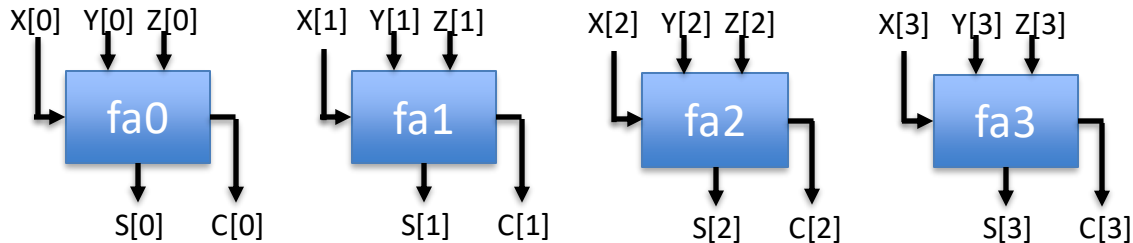
Language Element – Generate Block

- Example: 4-bit Carry Save Adder
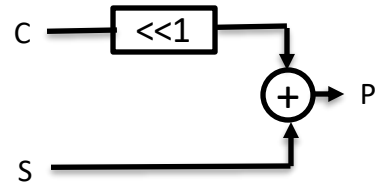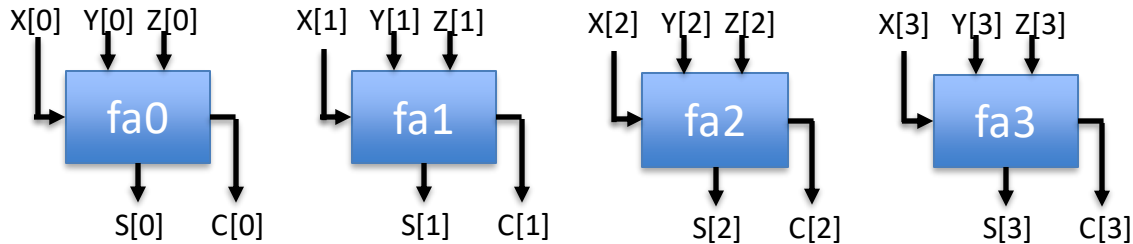
Language Element – Generate Block

- Example: 4-bit Carry Save Adder

```verilog
module CSA4 (X, Y, Z, P);
    input [3:0] X, Y, Z;
    output[5:0] P;
    wire [3:0] C, S;

    genvar i;
    generate
        for(i=0; i<4; i=i+1) begin
            Full_Adder fa(X[i], Y[i], Z[i], S[i], C[i]);
        end
    endgenerate

    assign P = S + (C << 1);
endmodule
```

Language Element – Parameter

- Parameters are constants that allow a module to be re-used with different specifications

```
parameter PARAMETER_NAME = <value>;
```

Language Element – Parameter

- Parameters are constants that allow a module to be re-used with different specifications

```
parameter PARAMETER_NAME = <value>;
```

- Example:

```
parameter N = 8;

wire [N-1:0] a,b;
wire [N:0] c;

assign c = a+b;
```

Language Element – Parameter

- Example: Parameterized module

```verilog
module CSA #(parameter N=4) (X, Y, Z, P);
    input [N-1:0] X, Y, Z;
    output[N+1:0] P;
    wire [N-1:0] C, S;

    genvar i;
    generate
        for(i=0; i<N; i=i+1) begin
            Full_Adder fa(X[i], Y[i], Z[i], S[i], C[i]);
        end
    endgenerate

    assign P = S + (C << 1);
endmodule
```

Language Element – Parameter

- Example: Parameterized module

```verilog
module CSA #(parameter N=4) (X, Y, Z, P);
    input [N-1:0] X, Y, Z;
    output[N+1:0] P;
    wire [N-1:0] C, S;

    genvar i;
    generate
        for(i=0; i<N; i=i+1) begin
            Full_Adder fa(X[i], Y[i], Z[i], S[i], C[i]);
        end
    endgenerate

    assign P = S + (C << 1);
endmodule
```

- How to instantiate a parameterized module?

```verilog
CSA #(.N(8)) unit(X,Y,Z,P);
```

Combinational Design vs Sequential Design

- Combinational design
  - Logic computation
- Sequential design
  - Logic computation + Memory element

Combinatorial

Example1:
```
wire a,b,c;
assign a = b+c;
```

Example2:
```
wire b,c;
reg a;
always @(*)
   a <= b+c;
```

Not a flip flop!

Sequential

```
wire b,c;
reg a;
always @(posedge clk)
   a <= b+c;
```

Flip flop

Sequential Design

- Sequential circuits have memory elements and logic computation
  - Flip-flops + Combinatorial part
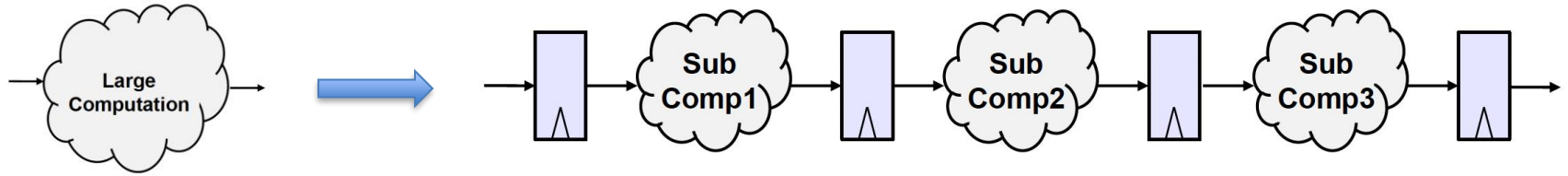
Sequential Design

- Sequential circuits have memory elements and logic computation
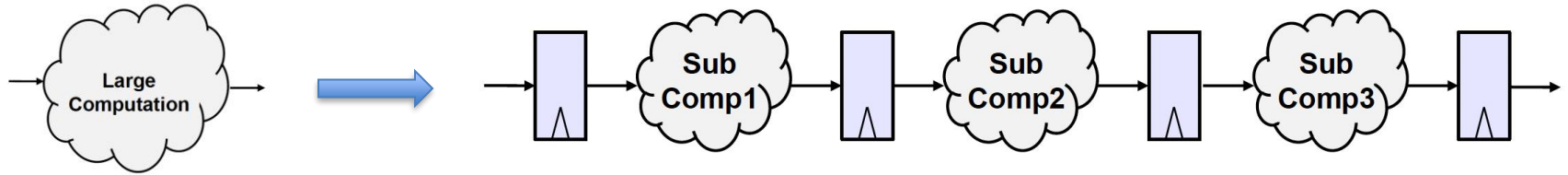 - Flip-flops + Combinatorial part
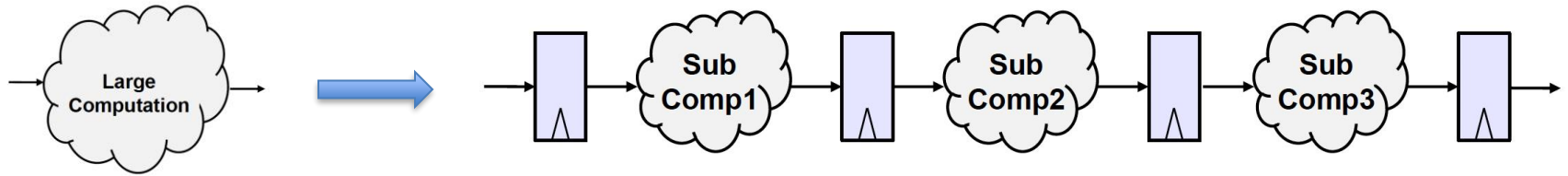
- Flip-flop outputs change (updated) at only edge of trigger signal
  - Clock
    - Positive clock edge (posedge)
    - Negative clock edge (negedge)
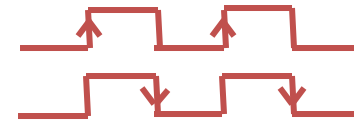
Sequential Design

- Sequential circuits have memory elements and logic computation
 - Flip-flops + Combinatorial part

- Flip-flop outputs change (updated) at only edge of trigger signal
  - Clock
    - `Positive clock edge (posedge)`
    - `Negative clock edge (negedge)`
  - Reset (optional)
    - Dependent to clock (**synchronous**)
    - Independent from clock (**asynchronous**)
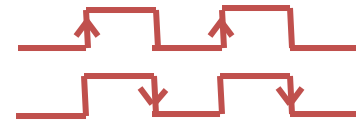
Sequential Design – Flip-Flops

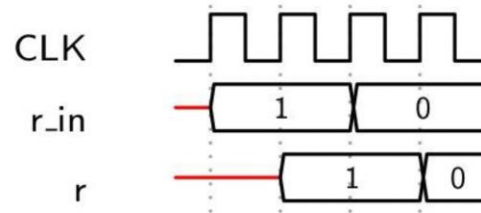- Result is only available after clock's posedge/negedge transition

```verilog
always @ (posedge CLK)
begin
    r <= r_in;
end
```
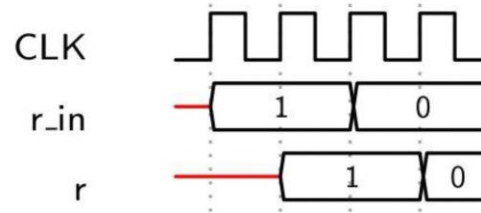
Sequential Design – Flip-Flops

- Result is only available after clock's posedge/negedge transition

```verilog
always @ (posedge CLK)
begin
    r <= r_in;
end
```


D flip-flop with synchronous reset
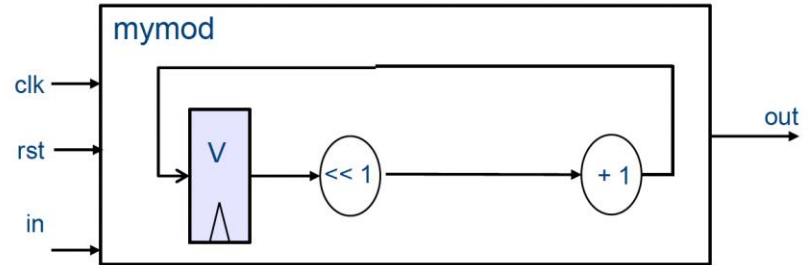
```verilog
always @ (posedge CLK)
begin
    if(RST)
        r <= 0;
    else
        r <= r_in;
end
```

D flip-flop with asynchronous reset

```verilog
always @ (posedge CLK or posedge RST)
begin
    if(RST)
        r <= 0;
    else
        r <= r_in;
end
```

Sequential Design – Reset

- Some sequential elements require a reset signal to initialize the circuit with a known state/value
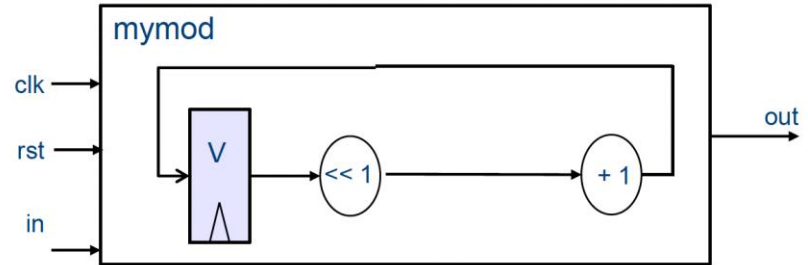
Sequential Design – Reset

- Some sequential elements require a reset signal to initialize the circuit with a known state/value

```
module mymod(clk, rst, in, out);
    input clk, rst;
    input [7:0] in;
    output [7:0] out;
    reg[7:0] v;




endmodule
```
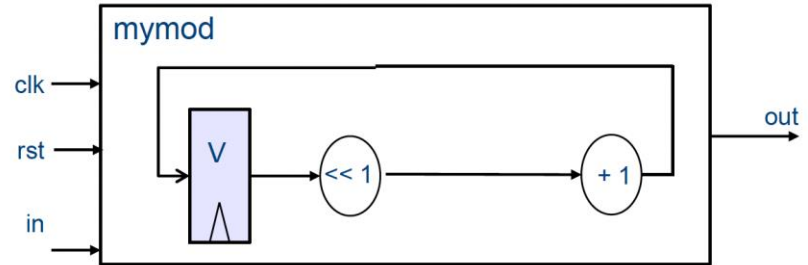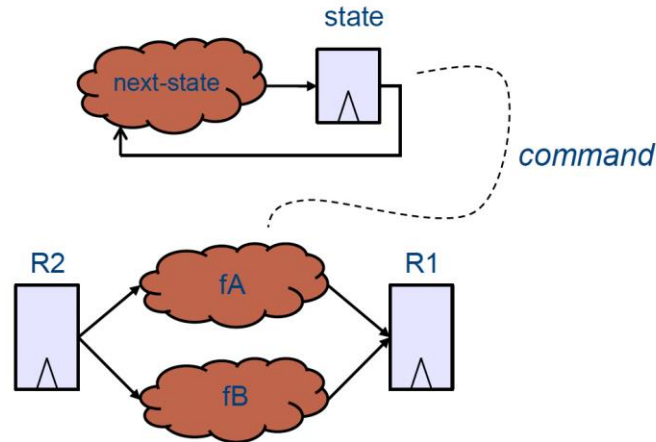
Sequential Design – Reset

- Some sequential elements require a reset signal to initialize the circuit with a known state/value

```verilog
module mymod(clk, rst, in, out);
    input clk, rst;
    input [7:0] in;
    output [7:0] out;
    reg[7:0] v;

    always @(posedge clk)
    begin
        if (rst)
            v <= in;
        else
            v <= (v<<1) + 1;
    end

    assign out = v;

endmodule
```

Control Unit (FSM) with Datapath

- **Basic idea:** Control Unit and datapath exist as separate circuits.

- **Control Unit:**
  - Controls the data flow
  - An easy way to make a control unit: Finite State Machine (FSM)
- **Datapath:**
  - Performs data processing operations

Design with FSM and Datapath Example – A pattern detection circuit

- A pattern detection circuit
  - A circuit takes 1-bit input and outputs "1" when the last 3-bits that it takes are "110". Otherwise, it outputs "0".
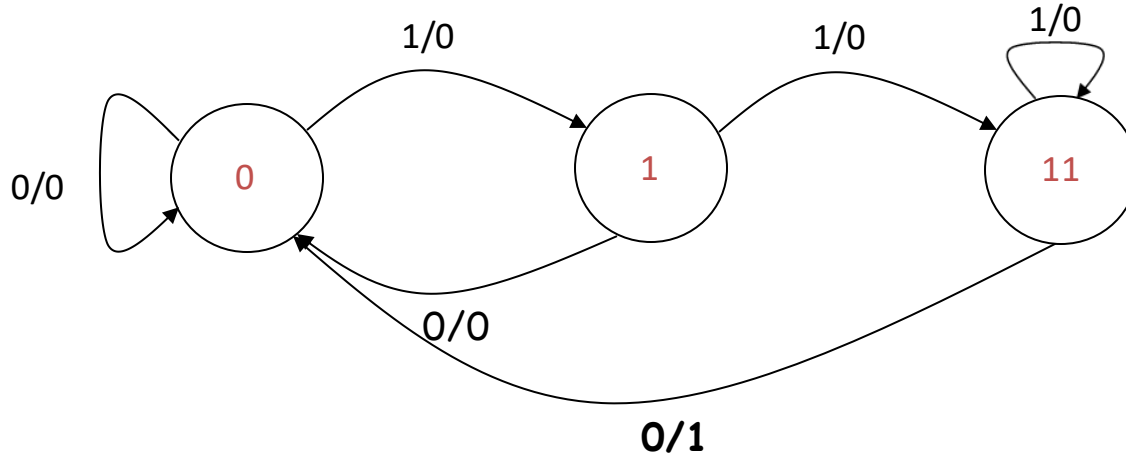
Design with FSM and Datapath Example – A pattern detection circuit

- A pattern detection circuit
  - A circuit takes 1-bit input and outputs "1" when the last 3-bits that it takes are "110". Otherwise, it outputs "0".

Design with FSM and Datapath Example – A pattern detection circuit

```verilog
module PD(input  clk, reset, bit_i,
         output bit_o);

reg [1:0] next_state;
reg [1:0] curr_state;
reg bit_o;

parameter ST_0  = 2'd0,
parameter ST_1  = 2'd1;
parameter ST_11 = 2'd2;

//State register
always@(posedge clk)
begin
   if(reset)
      curr_state <= ST_0;
   else
      curr_state <= next_state;
end
```

Design with FSM and Datapath Example – A pattern detection circuit

```verilog
module PD(input  clk, reset, bit_i,
         output bit_o);

reg [1:0] next_state;
reg [1:0] curr_state;
reg bit_o;

parameter ST_0  = 2'd0,
parameter ST_1  = 2'd1;
parameter ST_11 = 2'd2;

//State register
always@(posedge clk)
begin
   if(reset)
      curr_state <= ST_0;
   else
      curr_state <= next_state;
end
```
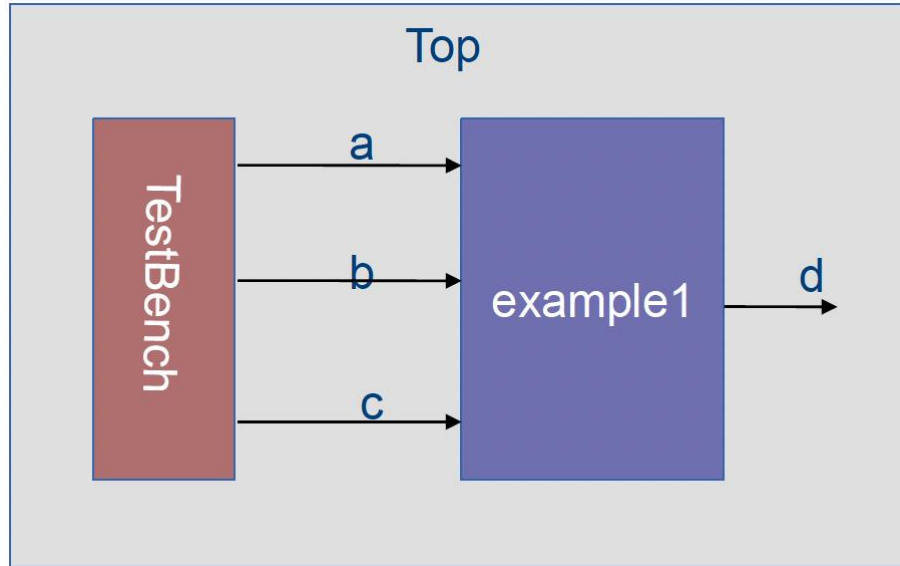
```verilog
//Next state logic
always@(*) begin
     case (curr_state)
      ST_0 : next_state = (bit_i == 1) ? ST_1  : ST_0;
      ST_1 : next_state = (bit_i == 1) ? ST_11 : ST_0;
     ST_11: next_state = (bit_i == 1) ? ST_11 : ST_0;
      default: next_state = ST_0;
end

// output logic
always@(posedge clk) begin
    if(reset)
       bit_o <= 0;
    else
       bit_o <= (curr_state == ST_11 && bit_i == 0) ? 1 : 0;
end

endmodule
```

Verilog Testbench

- Used to simulate design and test its functional correctness.
- Simulation is much faster than testing/debugging on actual hardware.

Verilog Testbench

- How to generate a testbench for your combinatorial design module?

    1. Create a new module for testbench (tb)
    2. Create a `reg` for each input of your design in tb
    3. Create a `wire` for each output of your design in tb
    4. Create clock (if your design has a clock)
    5. Instantiate your design in tb
    6. Connect `reg`s and `wire`s to your design in tb
    7. Give inputs to your input
    8. Observe/verify outputs

Verilog Testbench

- How to generate a testbench for your combinatorial design module?

    1. Create a new module for testbench (tb)
    2. Create a `reg` for each input of your design in tb
    3. Create a `wire` for each output of your design in tb
    4. Create clock (if your design has a clock)
    5. Instantiate your design in tb
    6. Connect `reg`s and `wire`s to your design in tb
    7. Give inputs to your input
    8. Observe/verify outputs

- Let's look at the pattern detector example.

```
module PD(input  clk, reset, bit_i,
          output bit_o);
```

Verilog Testbench – Steps for writing testbench

1. Create a new module for testbench (tb)

```
`timescale 1ns/1ps

module PD_tb();




...
```

```
...




endmodule
```

Verilog Testbench – Steps for writing testbench

2. Create a `reg` for each input of your design in tb

```verilog
`timescale 1ns/1ps

module PD_tb();

reg clk, reset, bit_i;



...
```

```verilog
...




endmodule
```

Verilog Testbench – Steps for writing testbench

3. Create a `wire` for each output of your design in tb

```verilog
`timescale 1ns/1ps

module PD_tb();

reg clk, reset, bit_i;
wire bit_o;



...
```

```verilog
...





















endmodule
```

Verilog Testbench – Steps for writing testbench

4. Create a `clock`

```verilog
`timescale 1ns/1ps

module PD_tb();

reg clk, reset, bit_i;
wire bit_o;

always #5 clk = ~clk;

...
```

```verilog
...




endmodule
```

Verilog Testbench – Steps for writing testbench

5+6. Instantiate your design in tb + Connect `regs` and `wires` to your design in tb

```verilog
`timescale 1ns/1ps

module PD_tb();

reg clk, reset, bit_i;
wire bit_o;

always #5 clk = ~clk;

PD dut(clk,reset,bit_i,bit_o);

...
```

```verilog
...




endmodule
```

Verilog Testbench – Steps for writing testbench

7+8. Give inputs to your design and observe outputs

```verilog
`timescale 1ns/1ps

module PD_tb();

reg clk, reset, bit_i;
wire bit_o;

always #5 clk = ~clk;

PD dut(clk,reset,bit_i,bit_o);

...
```

```verilog
...
initial begin
    // initialize all to 0
    clk=0; reset=1; bit_i=0;
    #20; // wait for 20 ns
    reset=0;
    #10; // wait for 10 ns
    bit_i=1; #20;
    bit_i=0; #20;
end

endmodule
```

Verilog Testbench – Steps for writing testbench

7+8. Give inputs to your design and observe outputs

```verilog
`timescale 1ns/1ps

module PD_tb();
```



```verilog
PD dut(clk,reset,bit_i,bit_o);

...
```

```verilog
...
initial begin
    // initialize all to 0
    clk=0; reset=1; bit_i=0;
    #20; // wait for 20 ns
    reset=0;
    #10; // wait for 10 ns
    bit_i=1; #20;
    bit_i=0; #20;
end

endmodule
```
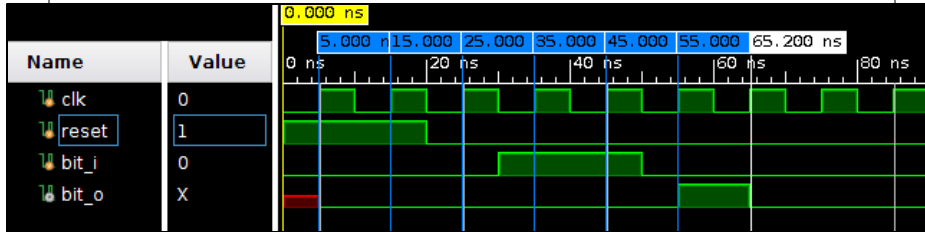
Common Mistakes/Bad Practices – Latches

- Latches easily cause timing problems:
    - In simulation: latches give correct results.,
    - On hardware: they almost always cause wrong results.
    - The tool throws warning when detecting latches in your design.

Example 1:

Latches

```
reg b;
always @(*)
begin
  if (condition)
    b <= b_in1;
end;
```

Not Solved!

```
reg b;
always @(*)
begin
  if (condition)
    b <= b_in1;
  else
      b <= b;
end;
```

Common Mistakes/Bad Practices – Latches

- Latches easily cause timing problems:
  - In simulation: latches give correct results.,
  - On hardware: they almost always cause wrong results.
  - The tool throws warning when detecting latches in your design.

Example 1:

Latches

```
reg b;
always @(*)
begin
  if (condition)
    b <= b_in1;
end;
```

Solved

```
reg b;
always @(*)
begin
  if (condition)
    b <= b_in1;
  else
    b <= 0;
end;
```

Common Mistakes/Bad Practices – Latches

- Latches easily cause timing problems:
    - In simulation: latches give correct results.,
    - On hardware: they almost always cause wrong results.
    - The tool throws warning when detecting latches in your design.

Example 2:

Latches

```
reg a;
always @(*)
begin
  case (condition)
    0: a <= a_in;
  endcase;
end;
```

Fixed

```
reg a;
always @(*)
begin
  case (condition)
    0: a <= a_in;
    default: a <= 0;
  endcase;
end;
```

Common Mistakes/Bad Practices – Multi-driven Nets

- Multi-driven nets

```verilog
reg state;
reg [7:0] a,b;

always @(posedge clk)
begin
  if (state==0)
    a <= 1;
  else
    a <= 2;    ←
  end;
end

always @(posedge clk)
begin
  if (state==0)
    b <= 1;
  else
    b <= 2;
    a <= 1;    ←
  end;
end;
```

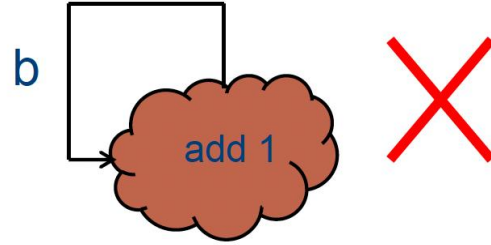Tip: Multiple always blocks simplifies your design.

Be careful!

Never assign the same "reg" in two different always blocks.

Why? Always blocks run in parallel.

Common Mistakes/Bad Practices – Combinatorial Loops

- Combinatorial loops

wire [7:0] b;
assign b = b + 1;

b

add 1

- No combinatorial loops between signals in a clock cycle

Common Mistakes/Bad Practices – Mixed Control Unit and Datapath

- Never use the same always block for control unit and datapath

<div align="center">BAD</div>

```
reg state;
reg [7:0] R1, R2;

always @(posedge clk) begin
  state <= state ^ 1;
  if (state==0)
    R1 <= R2 + 1;
  else
    R1 <= R2 << 2;
end
```

<div align="center">GOOD</div>

```
reg state;
reg[7:0] r;

always @(*) begin
  if (state==0)
    R1 <= R2 + 1;
  else
    R1 <= R2 << 2;
end

always @(posedge clk)
begin
  state <= state ^ 1;
end;
```

- Advantages:
  - Easier to maintain and read code
  - Likely to lead to better critical path
  - Easier for tool to synthesize