Computer Architecture
Tutorial 4

Hardware Description Language (HDL) and Combinational Logic Circuits

Artem Burmyakov, Muhammad Fahim, Alexander Tormasov

September 24, 2020

Content of the Tutorial

- Hardware Description Languages (HDL)
- Declaration Data Types – Nets, Registers, Vectors
- Procedural Block
  - Blocking and Non-blocking statement
- Asynchronicity
- System Function – $monitor
- Example: MUX with testbench

Hardware Description Languages (HDL)

- Verilog HDL
- VHDL = VHSIC HDL = Very High Speed Integrated Circuit HDL

- Verilog HDL != VHDL
  - These are two different Languages!
  - Verilog is closer to C
  - VHDL is closer to Ada

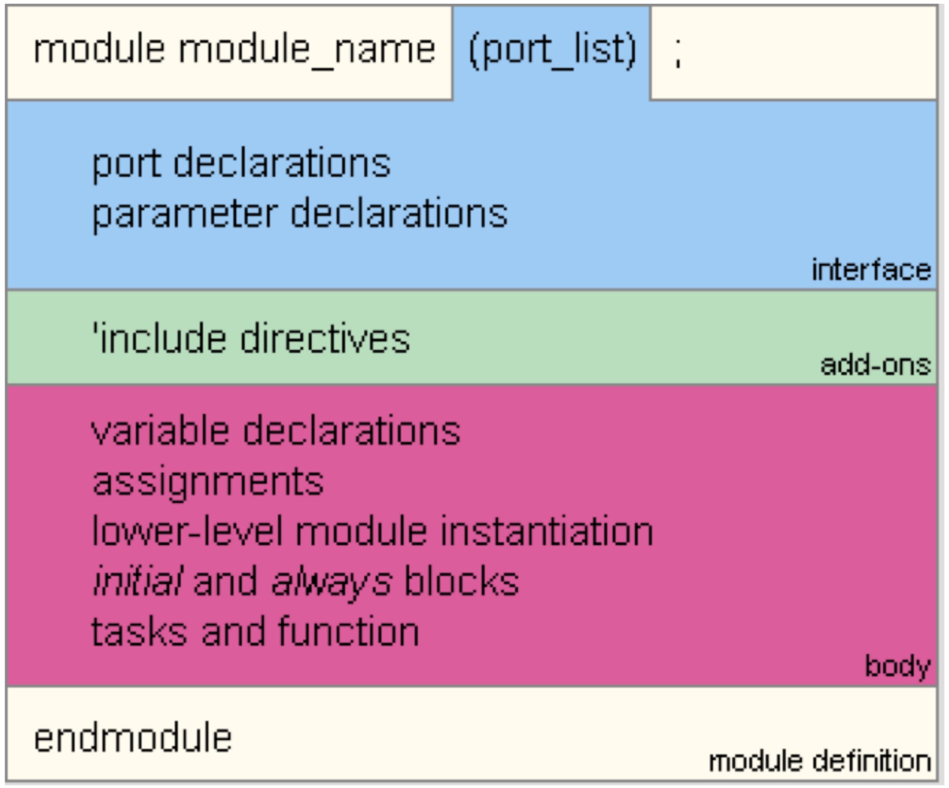
- NOTE: Our choice is: Verilog HDL

Hardware Description Languages (HDL)

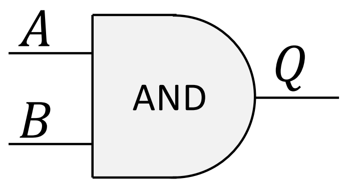
- **Module:** The main building block in Verilog and always terminates with end module
  - "**`module`**" followed by circuit name and port list

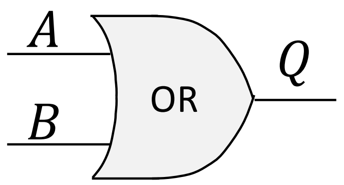
- **Comments:**
  One line comment (// ………….)
  Block Comment (/*…………….*/)
  Always use plenty of comments to make you code more readable.

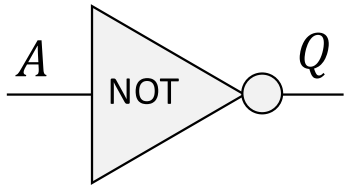
- **Case Sensitivity:** Verilog is case sensitive



Hardware Description Languages (HDL)







Verilog code

```
and(Q, A,B)      or(Q, A,B)      not(Q, A)
```

Declaration Data Types – Nets

- Used to represent connections between hardware elements
- **Keyword:** **`wire`**
- **Default:** One-bit values (unless declared as vectors)

- **Examples**

```
wire a;
wire b, c;
wire d=1’b0;
wire e=6’b001100;
```

Each bit in a net can take on one of four values:

0 – represents a logic zero
1 – represents a logic one
x – represents an unknown logic value
z – represents a high-impedance state

wires also known as “nets” or “signals”

Declaration Data Types – Registers

- Registers represent data storage elements
- Retain value until next assignment
- **Keyword:** `reg`
- **Default value:** `x`

Example

```
reg reset;
initial
begin
    reset = 1'b1;
    #100 reset=1'b0;
end
```

NOTE: this is not a hardware register

Declaration Data Types – Vectors

- Net and register data types can be declared as vectors (multiple bit widths)
- **Syntax:** `wire/reg [msb_index : lsb_index] data_id;`

Examples

```
wire a;
wire [7:0] bus;
wire [31:0] busA, busB, busC;
reg clock;
reg [0:40] virtual_addr;
```

NOTE: Access to bits or parts of a vector is possible

```
busA[7]
bus[2:0]
```

// three least-significant bits of bus

Procedural blocks

- Procedural blocks are the part of the language which represents sequential behavior.
- A module can have as many procedural blocks as necessary.
- These blocks are sequences of executable statements.
- The statements in each block are executed sequentially, but the blocks themselves are concurrent and asynchronous to other blocks.
- There are two types of procedural blocks:
  - `initial` blocks
  - `always` blocks

Procedural blocks – `initial`

- All initial blocks begin at time 0 and execute the initial statement.
- The statement may be a compound statement, this may entail executing lots of statements.
- When the initial statement finishes execution, the initial block terminates.
- If the initial statement is a compound statement, then the statement finishes after its last statement finishes.

Procedural blocks – `always`

- Always blocks also begin at time 0.

- The only difference between an always block and an initial block is that when the always statement finishes execution, it starts executing again.

- Elements in an always@ block are set/updated sequentially and in parallel, depending on the type of assignment used.

- There are two types of assignments:
  - <= (non-blocking) and
  - = (blocking).

<= (non-blocking) Assignments

- Non-blocking assignments happen in parallel

- In other words, if an always@ block contains multiple<= assignments, which are literally written in Verilog sequentially

- You should think of all of the assignments being set at exactly the same time

<= (non-blocking) Assignments – Example

```
always @( ... sensitivity list ... ) begin
B <= A;
C <= B;
D <= C;
end
```

- **Check point:** What will be the value of D?
- **Explanation:** Program specifies a circuit that reads "when the sensitivity list is satisfied, B gets A's value, C gets B's old value, and D gets C's old value." The key here is that C gets B's old value, etc

This ensures that C is not set to A.

= (blocking) Assignments

- Blocking assignments happen sequentially.
- In other words, if an **`always@`** block contains multiple =assignments.
- You should think of the assignments being set one after another

= (blocking) Assignments – Example

```
always @( ... sensitivity list ... ) begin
B = A;
C = B;
D = C;
end
```

- **Check point:** What will be the value of D?
- **Explanation:** This always@ block turns B, C, and D into A

Asynchronicity

- There may be many initial and always blocks in a module.
- All of them begin execution at time 0.
- However, there is no defined order between them.
- There is no guarantee that any statement will execute before or after any other statement which is not in the same block unless there is a time or event control to establish that relationship

System Function – `$monitor`

- Verilog provides some system functions specifically for generating input and output to help verification

```
$monitor (“format string”, parameter1, parameter2, ... );
```

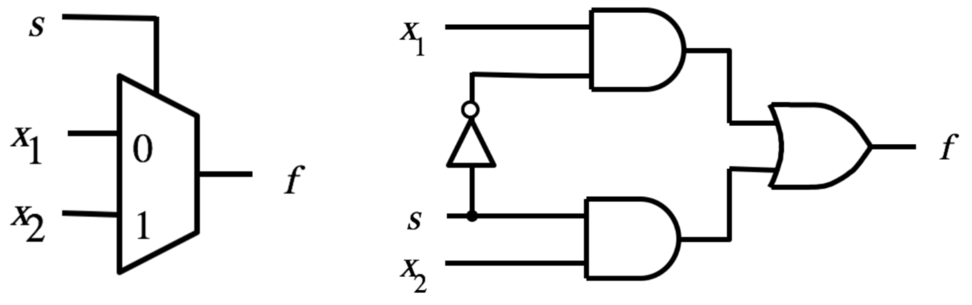
- “format string”– specifies how the parameters will be displayed:
  - **`%d`** will print the variable in decimal
  - **`%b`** will print the variable in binary
  - **`%h`** will print the variable in hexadecimal

Wire vs. Register

- **`reg`:** It can be assigned to a procedural block (a block beginning with always or initial)
- **`wire`:** It can be assigned in a continuous assignment (an assign statement) or as an output of an instantiated sub-module.
- If you plan to assign your output in sequential code, such as within an always block, declare it as a `reg`
- Otherwise, it should be a wire, which is also the default.

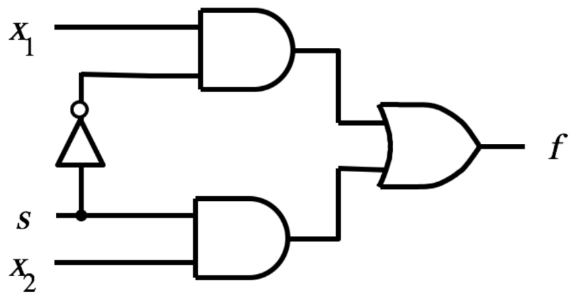
Multiplexor (MUX)

- Multiplexor is a combinational logic device.
- Designed to control the transmission of data from several sources to one output channel.
- **For Example:** 2-1 Multiplexor Schematic



$$f(S, X_1, X_2) = S' X_1 + S X_2$$

2-1 Multiplexor – Verilog HDL code

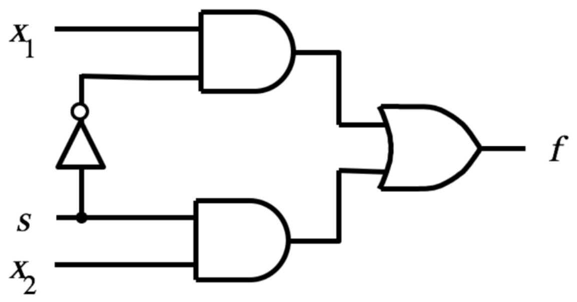


```
module mux_2_1(

    input x1,
    input x2,
    input s,
    output f

);
    assign f = ((~s) & x1) | (s & x2);

endmodule
```

using ternary operator (? :)

```
module mux_2_1_ternary
(
    input  x1,
    input  x2,
    input  s,
    output f
);

assign f = s ? x2 : x1;

endmodule
```

2-1 Multiplexor – Verilog HDL code



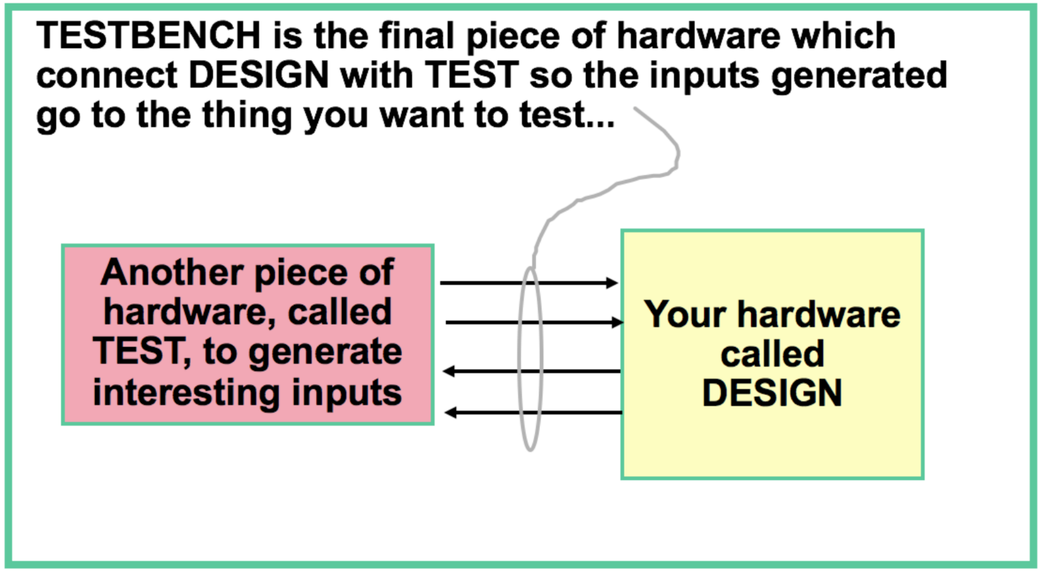
using conditional operator (case)

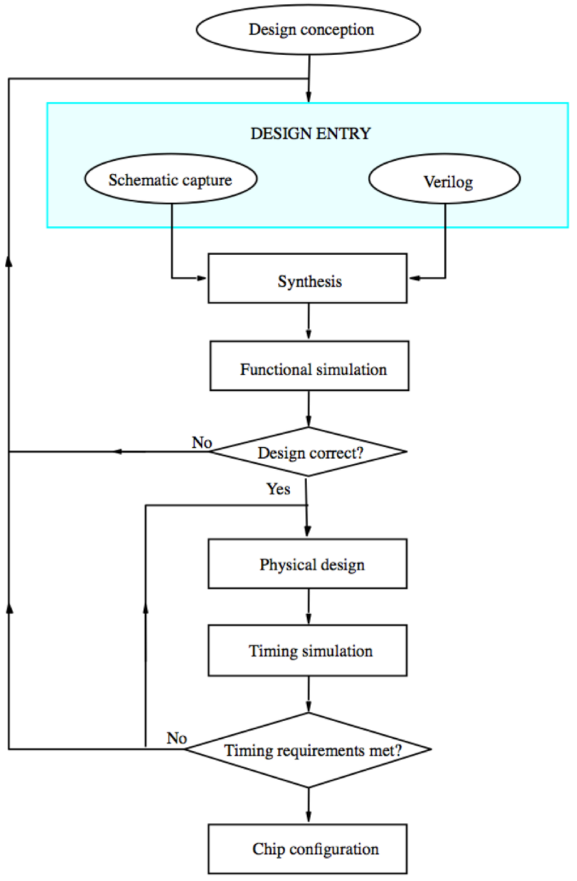
```
module mux_2_1_ternary
(
    input  x1,
    input  x2,
    input  s,
    output f
);

assign f = s ? x2 : x1;

endmodule
```

using conditional operator (if)

```
module b1_mux_2_1_if
(
    input  x1,
    input  x2,
    input  s,
    output reg f
);
    always@(*)
    begin
        if(s)
            f = x2;
        else
            f = x1;
    end

endmodule
```

Testbench


TESTBENCH is the final piece of hardware which connect DESIGN with TEST so the inputs generated go to the thing you want to test...
Another piece of hardware, called TEST, to generate interesting inputs
Your hardware called DESIGN



- Testbench is for simulation only, not for synthesis

2-1 Multiplexor – Testbench

```
module testbench;
    // input and output test signals
    reg  a;
    reg  b;
    reg  sel;
    wire y_comb;
    wire y_sel;
    wire y_if;
    wire y_case;

   // creating the instance of the module we want to test
    mux_2_1          mux (a, b, sel, y_comb);
    mux_2_1_ternary mux_2_1_ternary  (a, b, sel, y_sel);
    b1_mux_2_1_if    b1_mux_2_1_if  (a, b, sel, y_if);
    b1_mux_2_1_case b1_mux_2_1_case  (a, b, sel, y_case);
```

2-1 Multiplexor – Testbench

```
    initial
        begin
            a = 1'b0;
            b = 1'b1;
            #5;                 // pause (5 units of delay)
            sel = 1'b0;         // sel change to 0; a -> y
            #10;                // pause (10 units of delay)
            sel = 1'b1;         // sel change to 1; b -> y
            #10;
            b = 1'b0;           // b change; y changes too. sel == 1'b1
            #5;
            b = 1'b1;
            #5;
        end
    //  print signal values on every change
    initial
        $monitor("a=%b b=%b sel=%b y_comb=%b y_sel=%b y_if=%b y_
            case=%b",
                a, b, sel, y_comb, y_sel, y_if, y_case);
    initial
        $dumpvars;  //iverilog dump init
endmodule
```

Thank You ☺