Tutorial – 1

Intro to Processor Architecture

- Kaamya Dasika
- Krrish Goenka
- Siddarth Gottumukkula
- Vedant Tejas
- M P Samartha
- Vedant Pahariya

*Order of Names is decided by Mario Kart Race

Overview

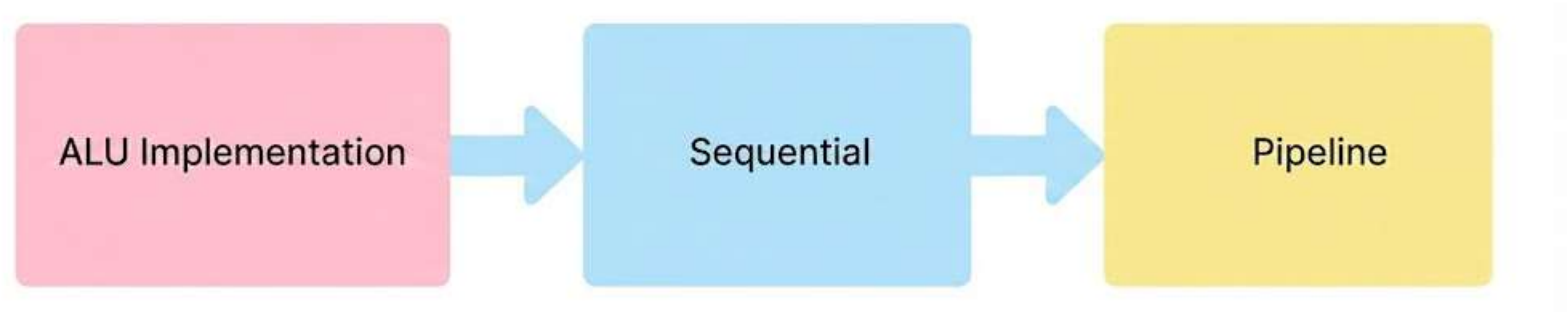
- Course Description
- Project Outline/Github
- AI usage Policy/Moss Check
- Assignment/Project
- Verilog Revision
- Project Motivation

Course Description

Type of Evaluation	Weightage (in %)
Quiz-1	10
End Sem Exam	30
Project	60

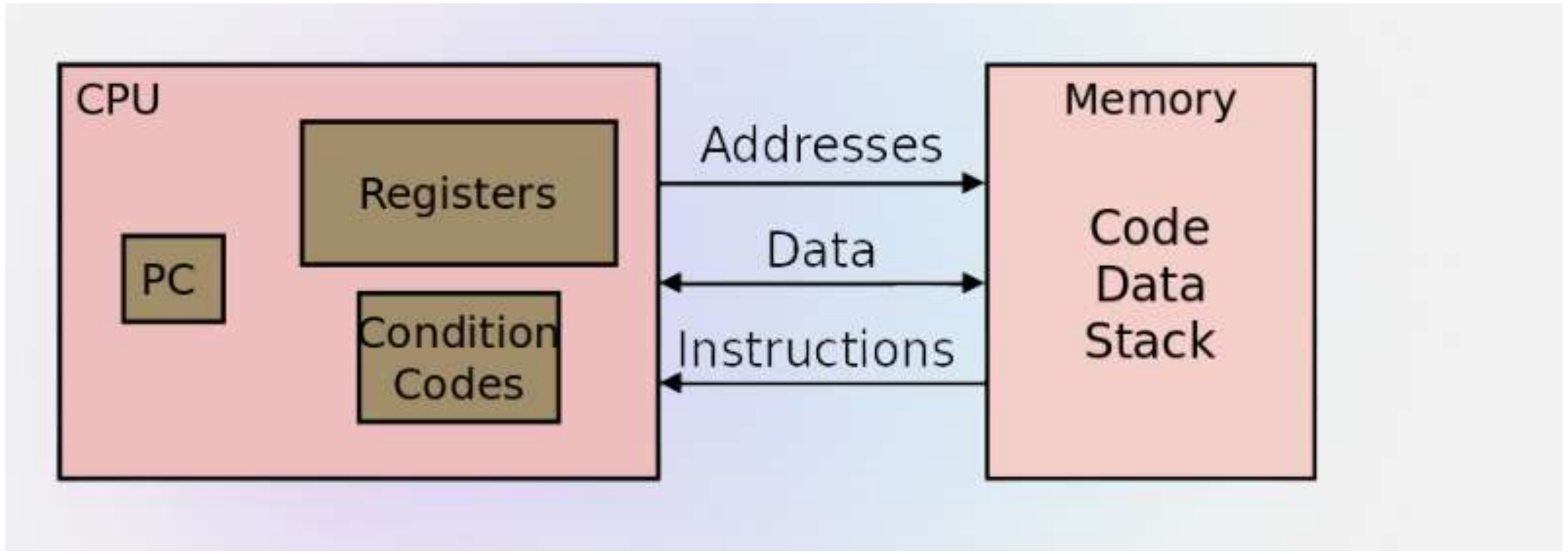
Project Heavy Course!!!

Project Outline



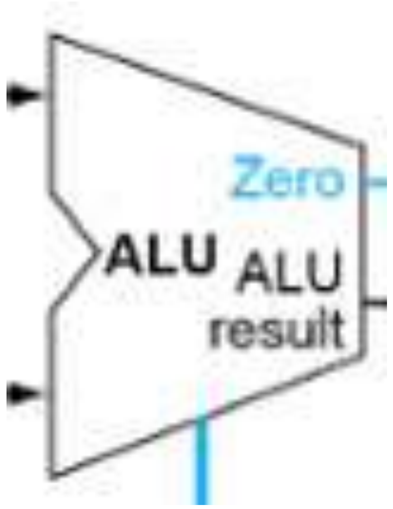
All this is going to be implemented in the Verilog

Processor & Memory

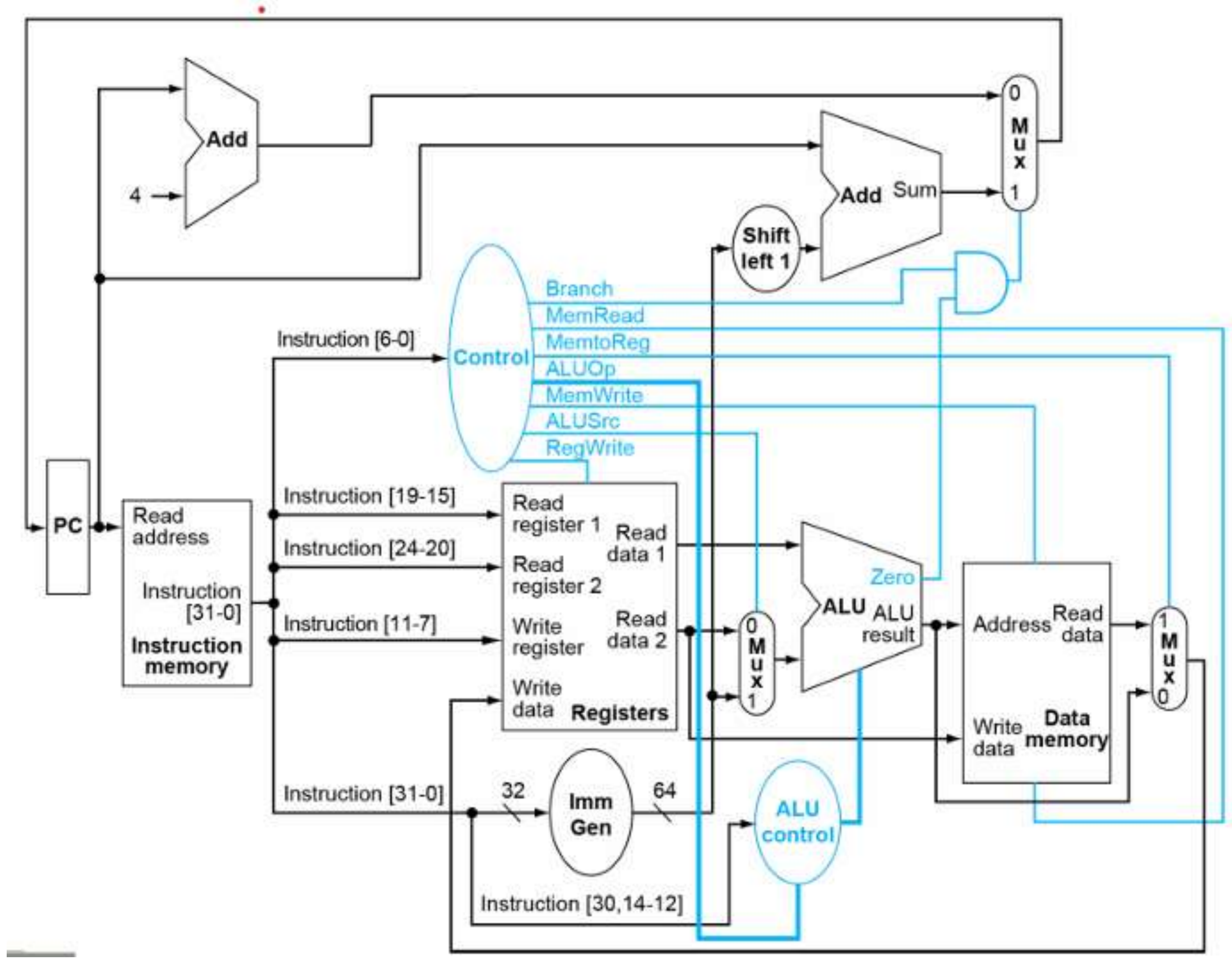


Von-Neumann Architecture

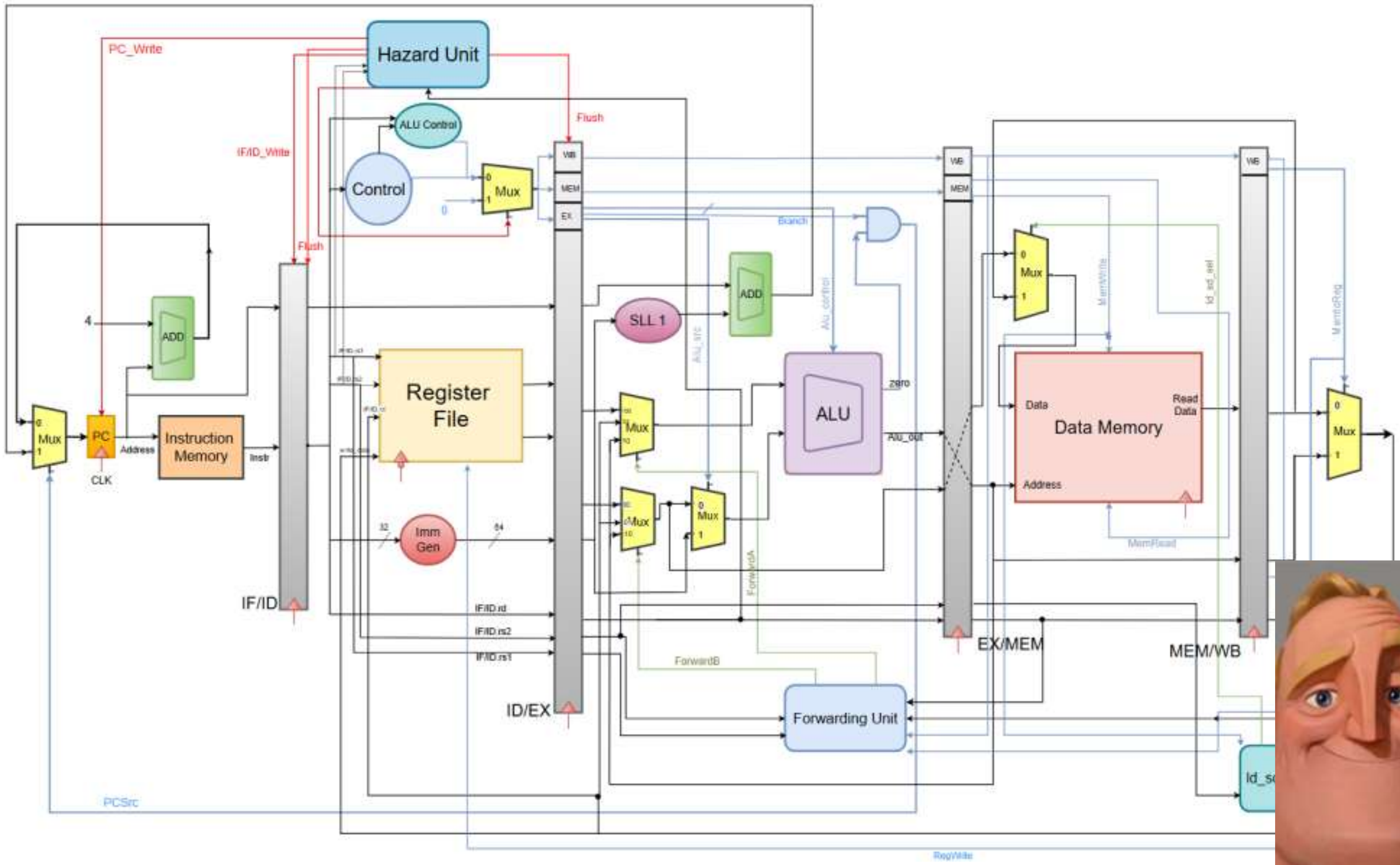
Phase-1



Sequential Design



Pipelined Processor



Verilog Setup

- **Ubuntu / Debian Based system**

```
sudo apt update
sudo apt install iverilog
sudo apt install gtkwave
```

- **Windows**

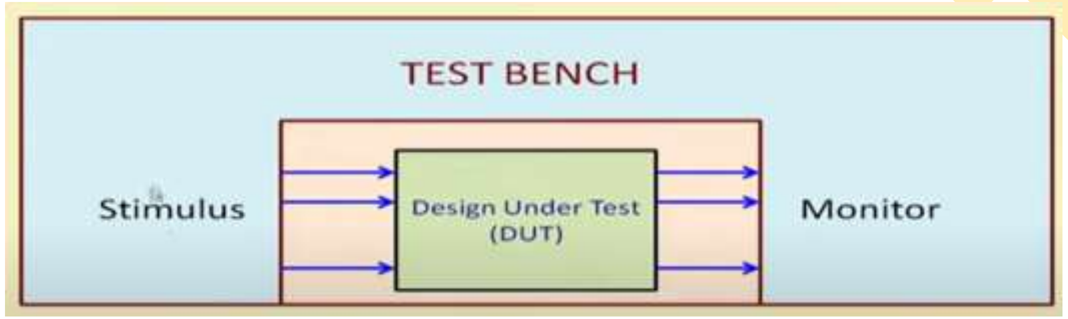
Head to https://bleyer.org/icarus/ and download the latest source file . Make sure to select the gtkwave option when it installs . U r done !!

Just to confirm the installation , enter `iverilog` in the command prompt and review the details .

Short recap on Verilog

- Verilog is a **hardware description** language.

- Testbench consists of the testing infrastructure and the test cases for testing code.



- GTKwave is a **waveform visualizer tool** that allows to examine the input and output signal stated across the simulation period.

- **Commands:**

➢ iverilog -o output example.v example_test.v

➢ vvp output

Continued…

- In Verilog, the basic unit of hardware is a module. A module **cannot be defined within other modules** but can be instantiated.

Basic Structure:

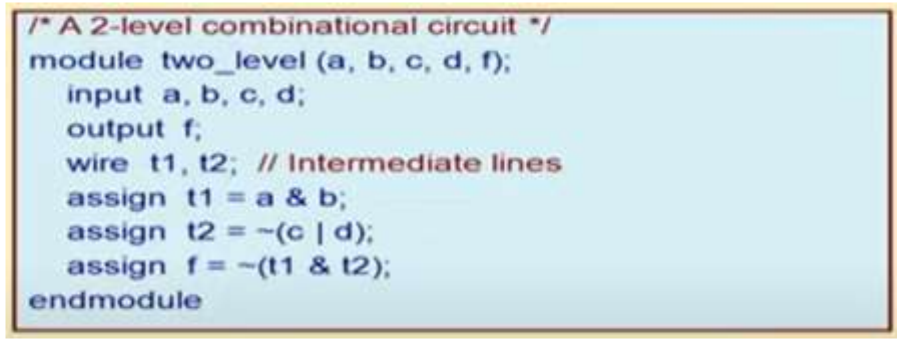
module module_name(list_of_ports);

Input/output declaration;

Local net declarations;

Parallel statements;

Endmodule



```
/* A 2-level combinational circuit */
module two_level (a, b, c, d, f);
   input a, b, c, d;
   output f;
   wire t1, t2; // Intermediate lines
   assign t1 = a & b;
   assign t2 = ~(c | d);
   assign f = ~(t1 & t2);
endmodule
```

Behavioral vs Structural

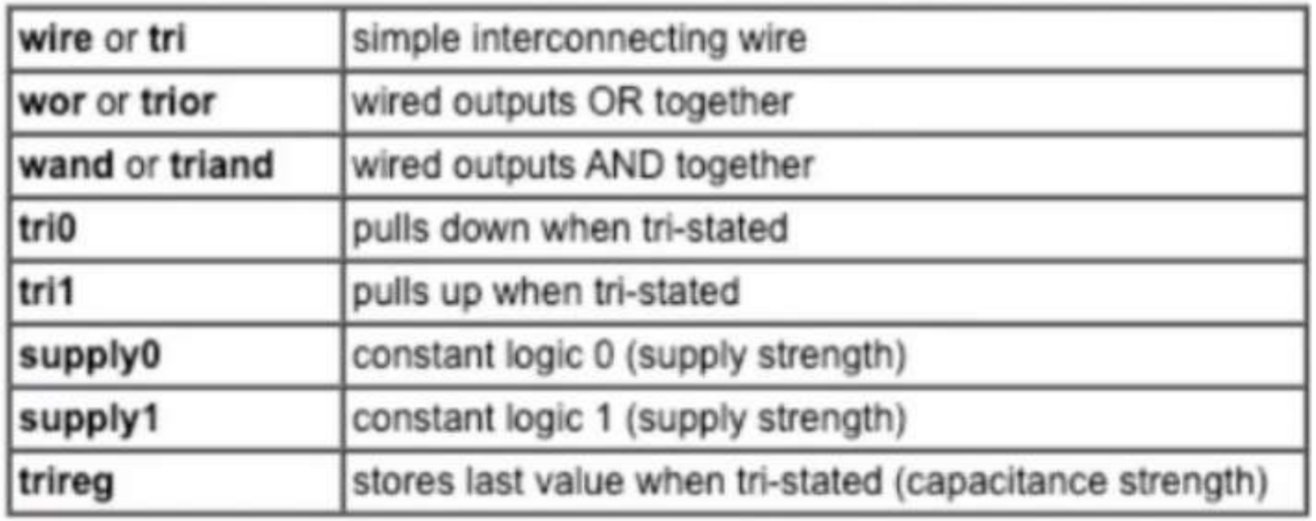
In **structural** verilog code, you describe the hardware In terms of primitives such as **gates**. Low level

In **behavioral** verilog code, you defined the behavior of the module in the **form of a conditional statements, loops, functions .** High level .

Data types in Verilog

1. Net

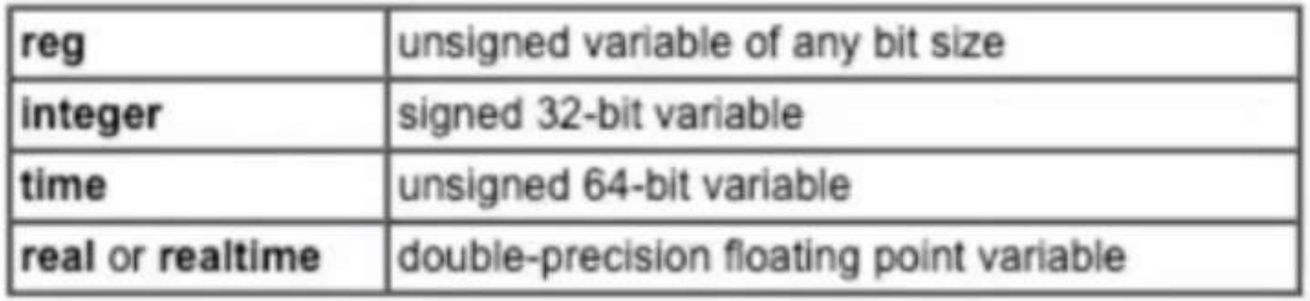
- Nets are **continuously driven,** cannot be used to store a value.
- Used to model connections between continuous assignments and instantiations.
- Default value of net data type is **Z (High impedance value)**
- **wire** is one of the most commonly used net types.



wire or **tri**	simple interconnecting wire
wor or **trior**	wired outputs OR together
wand or **triand**	wired outputs AND together
tri0	pulls down when tri-stated
tri1	pulls up when tri-stated
supply0	constant logic 0 (supply strength)
supply1	constant logic 1 (supply strength)
trireg	stores last value when tri-stated (capacitance strength)

2. Reg

- Retains the last value assigned to it, often used to r**epresent a storage element**, but sometimes it can translate to combinational circuits as well.
- It can be assigned a value in synchronism with clk signal or even otherwise.
- Default value of a "reg" data type is x (Unknown logic state)
- Often used in **procedural assignments.**



reg	unsigned variable of any bit size
integer	signed 32-bit variable
time	unsigned 64-bit variable
real or **realtime**	double-precision floating point variable

Vectors

"Net" or "Reg" type variables can be **declared as vectors of multiple bitwidths**.

Vectors are declared by specifying a range [range1:range2], where **range1 is MSB and range2 is LSB.**

Part of vectors can be addressed and used in an expression.

Example:

```
reg [31:0] IR;
reg [5:0] opcode;
Opcode = IR[31:26];

reg [63:0] register_banks[15:0];   // 16  64-bit registers
```

assign statement

- Type

```
wire y;
assign y = a & b;
```

- The assign statement in Verilog is used for continuous assignment, meaning it continuously drives the value of a net whenever the right-hand side (RHS) expression changes.

- LHS (Left-Hand Side): The LHS of the assign statement must be a net type (e.g., wire), because the assign statement is used to model continuous assignments, which require a net type to carry signals.

- RHS (Right-Hand Side): The RHS can be either a net type (e.g., wire) or a reg type (e.g., reg). However, note that reg is typically used for variables that store values in procedural blocks (always), not for continuous assignments.

- In short , in this example, y will continuously be updated to reflect the AND of a and b.

• Connectivity during instantiation

a. **Positional association**

The parameters of the module being instantiated are listed in **same order as in original description.**

b. **Explicit(or Named) association**

Syntax: .port_name(signal_name)

The parameters of the module are listed in any order.

```
-----------------------------------------------------------------------------------------

module example(A,B,C,D,E,F,Y);

…

endmodule

----------------------------------------------------------------------------
  --

module testbench;

reg X1, X2, X3, X4, X5, X6;  wire OUT;

…

example DUT(.OUT(Y), .X1(A), .X2(B), .X3(C), .X4(D), .X5(E), .X6(F))

endmodule
```

Parameters: A constant with a given value. If the size is not specified, it is taken to be 32-bit.

```
parameter IPA = 25;  // Default is 32-bit, IPA = 25
parameter LO = 2'b01;  // 2-bit value (binary 01)
```

Primitive Gates:

Verilog provides built-in logic gates such as **and, or, nor, xor**, etc., which can be instantiated in your designs to create complex logic functions.

The `timescale derivative

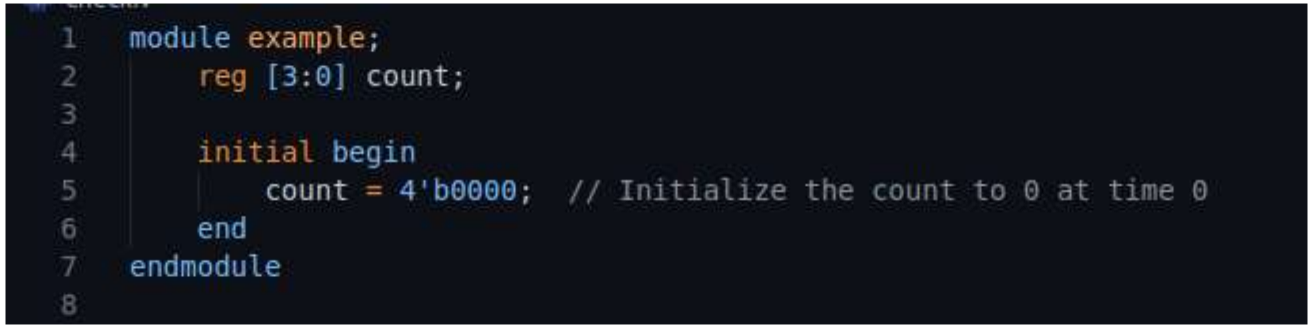
Often used in simulation, delay values in one module need to be specified in terms of some time unit.

SYNTAX : `timescale<reference-time-unit> /<time – precision>

EXAMPLE : `timescale 10ns/1 ns // Time unit is 10ns, precision is 1ns (like all time operations will be rounded off to 1ns)

Procedural Assignment

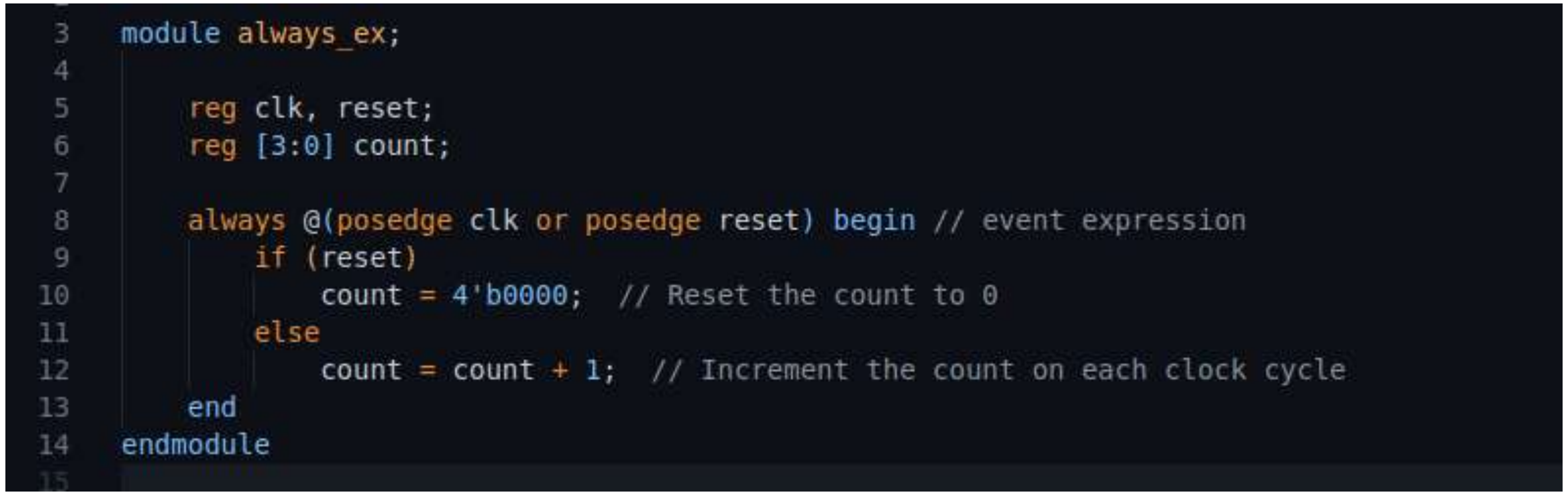
- Procedural assignments are used to specify actions that happen **over time** during simulation .
- 1. **initial** block
- **Purpose**: The initial block is used to define actions that **occur only once at time 0** (the start of simulation). After the block executes, it **does not repeat.**
- **Execution**: All initial blocks start executing at the same time (at time 0) and will run **concurrently in parallel**.
- **Typical Use**: The initial block is useful for setting initial values or for defining the initial conditions of your design, such as initializing registers or wires.



```
module example;
    reg [3:0] count;

    initial begin
        count = 4'b0000;  // Initialize the count to 0 at time 0
    end
endmodule

```

Procedural Assignment

- **2. always** block
- **Purpose**: The always block is used to define processes that need to be **executed repeatedly** in response to changes in signals or events. This block will repeat indefinitely as long as the condition specified in the event **expression is met**.
- **Event Expression**: The event expression determines the condition under which the block executes. Whenever the signals in the event expression change, the always block is triggered.



```
module always_ex;

    reg clk, reset;
    reg [3:0] count;

    always @(posedge clk or posedge reset) begin // event expression
        if (reset)
            count = 4'b0000;  // Reset the count to 0
        else
            count = count + 1;  // Increment the count on each clock cycle
    end
endmodule
```

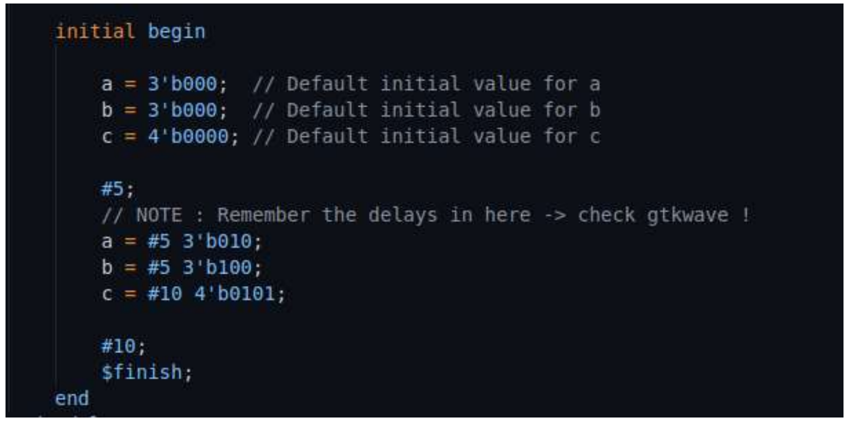
Blocking vs Non-blocking statement {very imp !}

- Blocking Assignments (Sequential Execution) :
- - Blocking assignments are **executed in the order** they appear in the code. The next statement cannot execute until the current one finishes.
- - Syntax: **=**

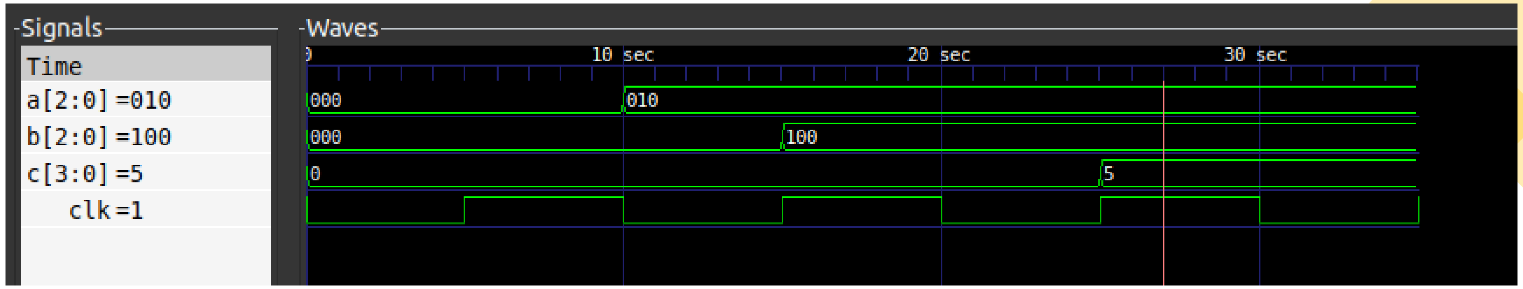
- Non-Blocking Assignments (Concurrent Execution)
- - Non-blocking assignments allow the next statement to execute immediately, without waiting for the current one to finish. This simulates concurrent operations t**hat occur at the same time in hardware.**
- - Syntax: **<=**

Blocking vs Non-blocking statement {Example}

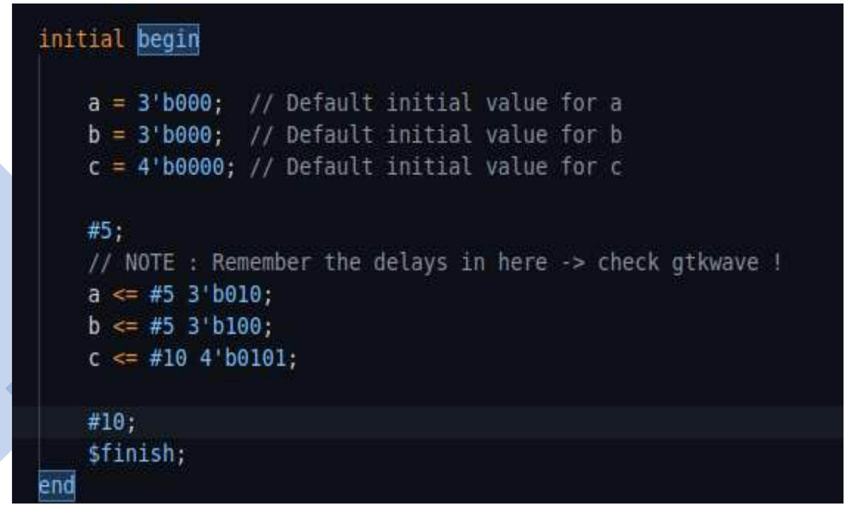
- 1. Blocking



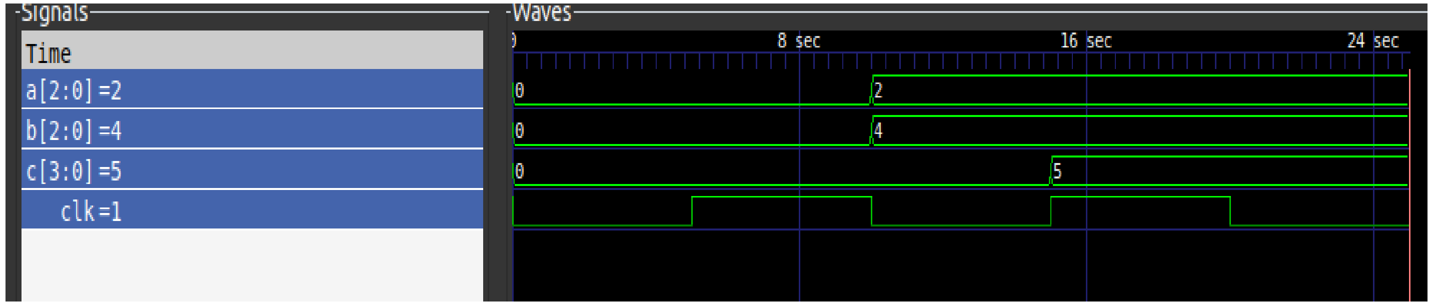
```
initial begin

    a = 3'b000;  // Default initial value for a
    b = 3'b000;  // Default initial value for b
    c = 4'b0000; // Default initial value for c

    #5;
    // NOTE : Remember the delays in here -> check gtkwave !
    a = #5 3'b010;
    b = #5 3'b100;
    c = #10 4'b0101;

    #10;
    $finish;
end
```



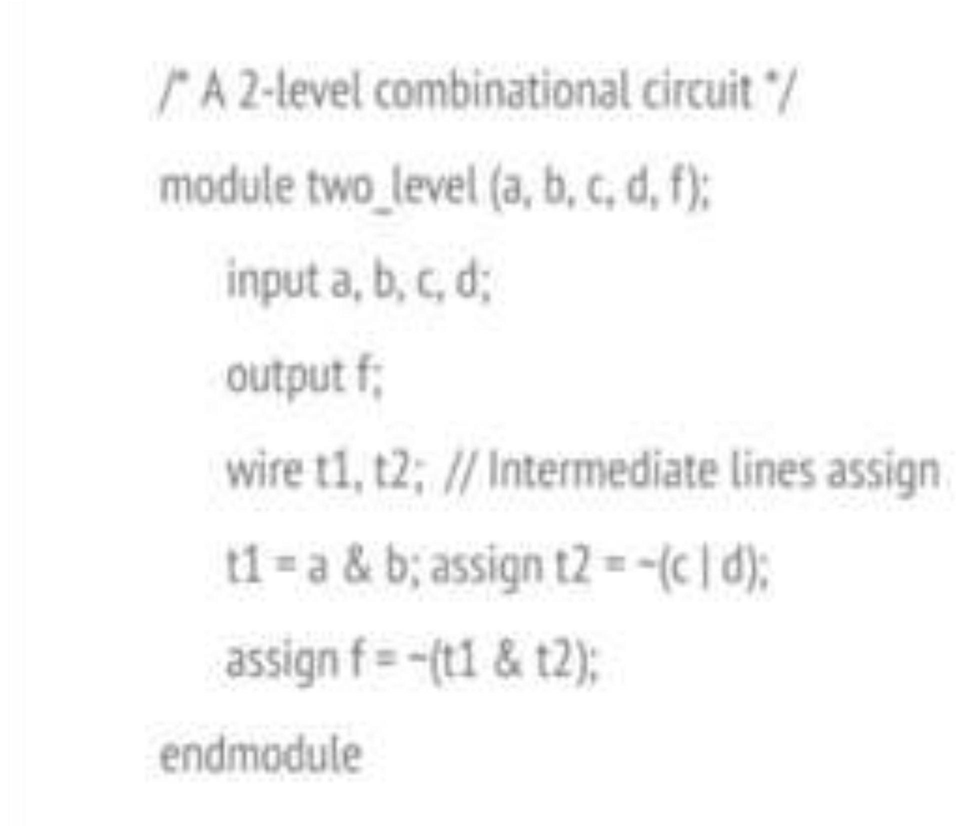
2. Non blocking



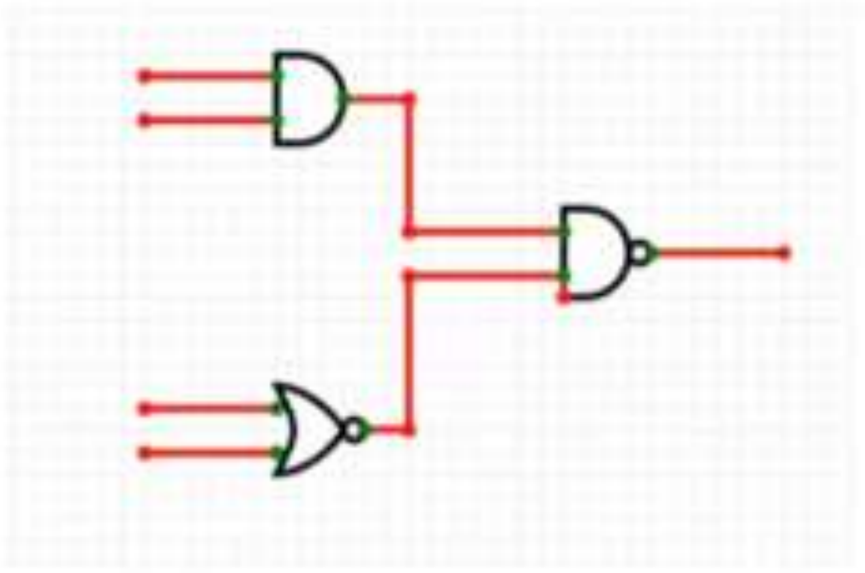
```
initial begin

    a = 3'b000;  // Default initial value for a
    b = 3'b000;  // Default initial value for b
    c = 4'b0000; // Default initial value for c

    #5;
    // NOTE : Remember the delays in here -> check gtkwave !
    a <= #5 3'b010;
    b <= #5 3'b100;
    c <= #10 4'b0101;

    #10;
    $finish;
end
```



Combinational Circuits



```
/* A 2-level combinational circuit */
module two_level (a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2;  // Intermediate lines assign
    t1 = a & b; assign t2 = ~(c | d);
    assign f = ~(t1 & t2);
endmodule
```

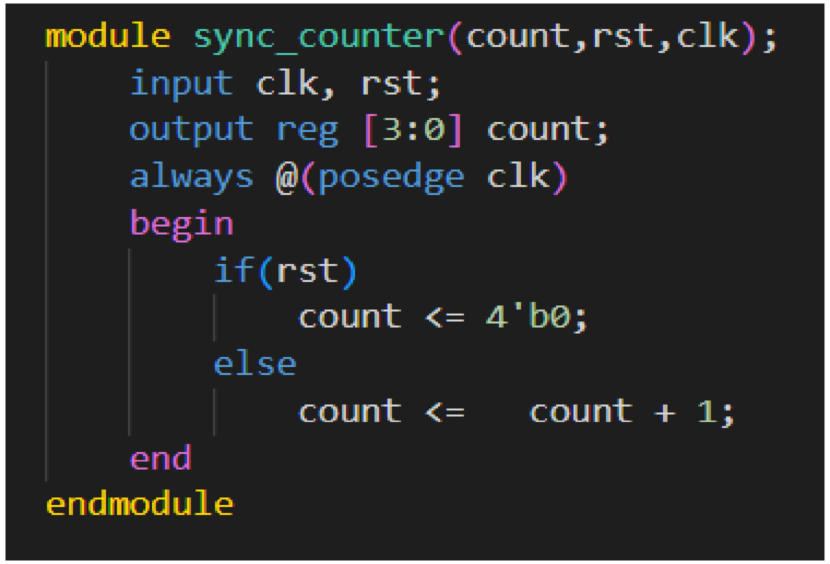


Inputs for the gates are a,b,c,d respectively.

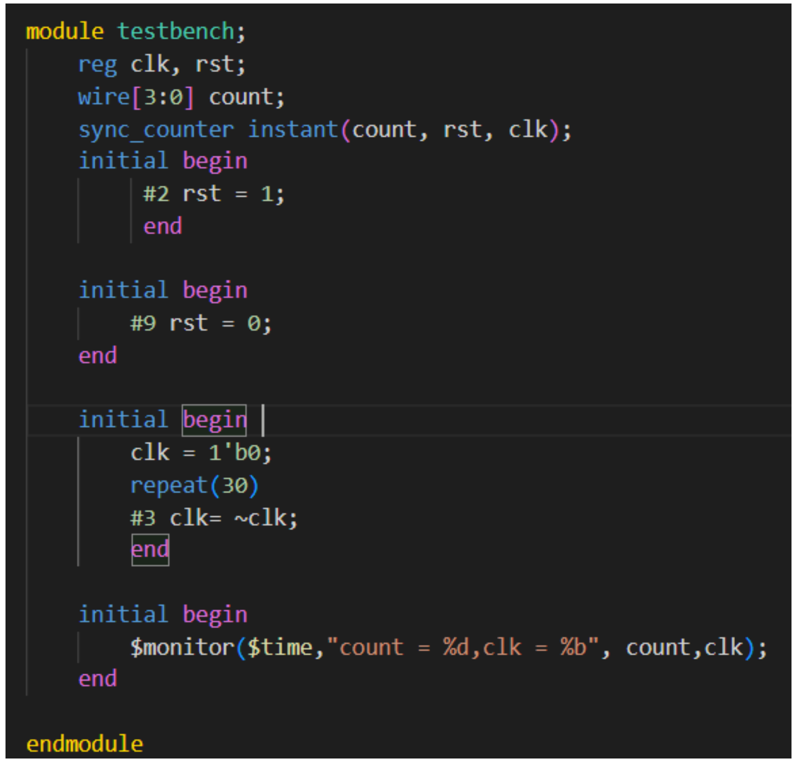
Sequential Circuits

- Many circuits of practical importance are required to have some sort of **memory element**
- This could be to store the output, store the input, feedback purposes etc.
- Combinational circuits cannot store any information and output varies every time the input is changed.
- This is why we need sequential circuits, which can be defined as "combinational circuits with feedback". The output of these circuits thus depends on the **present as well as the past state inputs.**

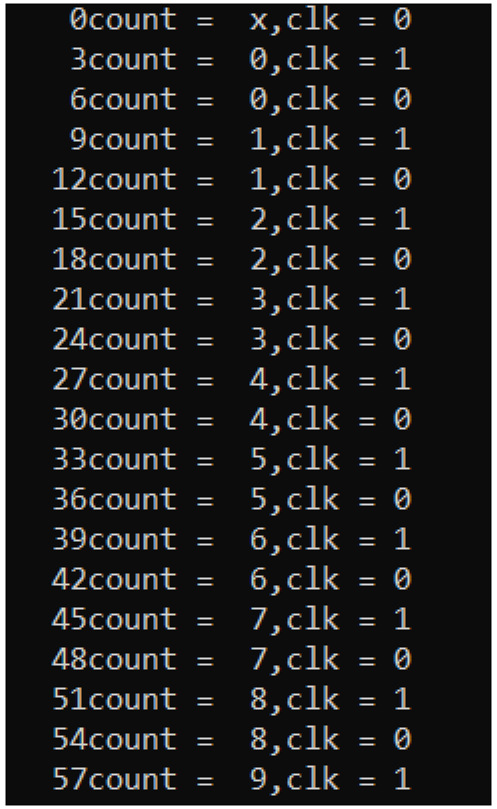
- **Synchronous Counter**



```
module sync_counter(count,rst,clk);
    input clk, rst;
    output reg [3:0] count;
    always @(posedge clk)
    begin
        if(rst)
            count <= 4'b0;
        else
            count <=   count + 1;
    end
endmodule
```



```
module testbench;
    reg clk, rst;
    wire[3:0] count;
    sync_counter instant(count, rst, clk);
    initial begin
        #2 rst = 1;
        end

    initial begin
        #9 rst = 0;
    end

    initial begin |
        clk = 1'b0;
        repeat(30)
        #3 clk= ~clk;
        end

    initial begin
        $monitor($time,"count = %d,clk = %b", count,clk);
    end

endmodule
```



```
   0count =  x,clk = 0
   3count =  0,clk = 1
   6count =  0,clk = 0
   9count =  1,clk = 1
  12count =  1,clk = 0
  15count =  2,clk = 1
  18count =  2,clk = 0
  21count =  3,clk = 1
  24count =  3,clk = 0
  27count =  4,clk = 1
  30count =  4,clk = 0
  33count =  5,clk = 1
  36count =  5,clk = 0
  39count =  6,clk = 1
  42count =  6,clk = 0
  45count =  7,clk = 1
  48count =  7,clk = 0
  51count =  8,clk = 1
  54count =  8,clk = 0
  57count =  9,clk = 1
```

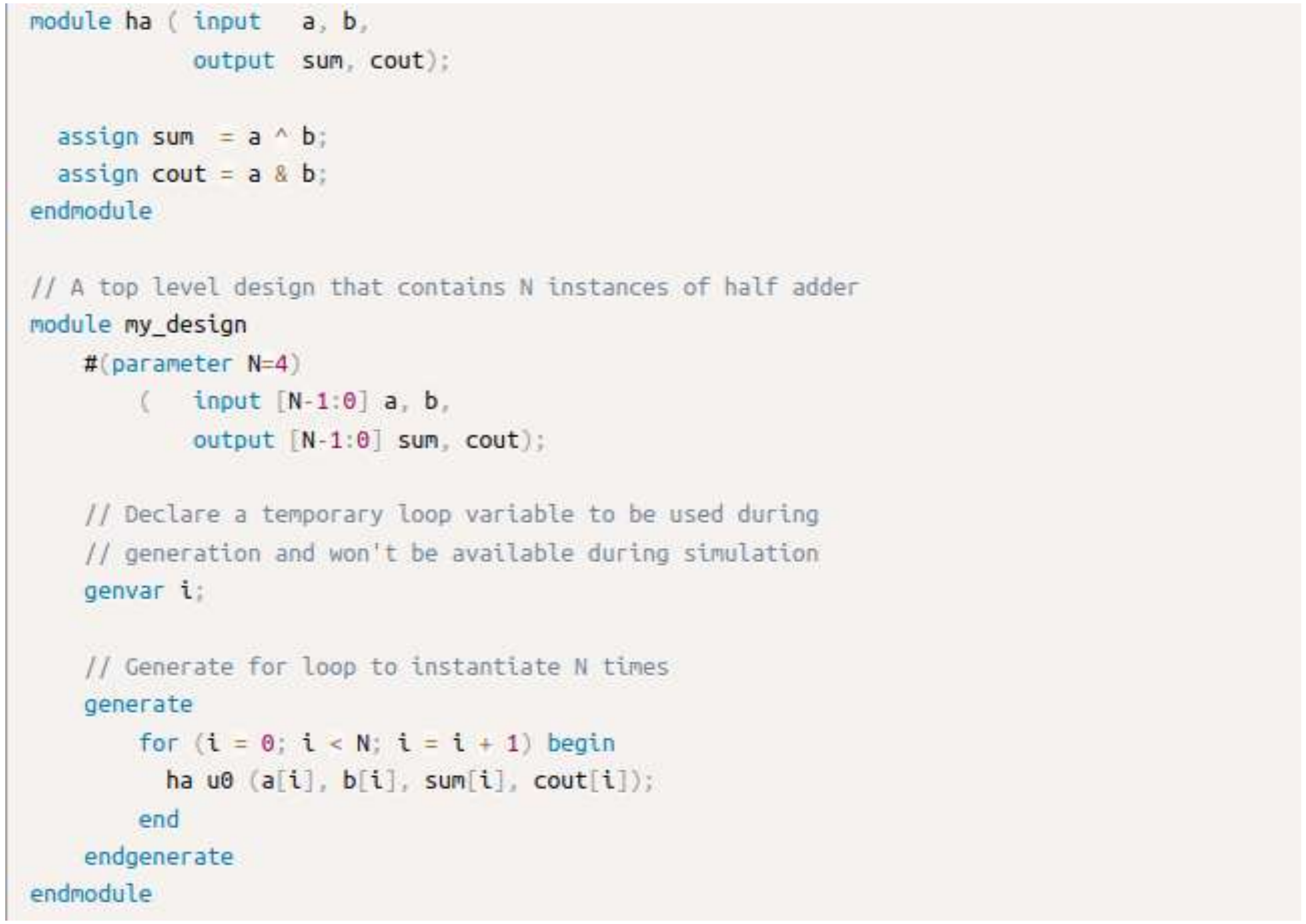
Q. Can you guess why the count value changes every 6s (don’t worry about the units) and not every 3s ?

Verilog Generate block

A generate block allows to multiply module instances or perform conditional instantiation of any module. It provides the ability for the design to be built based on Verilog parameters. These statements are particularly convenient when the same operation or module instance needs to be repeated multiple times or if certain code has to be conditionally included based on given Verilog parameters.
Three types of generate statements:

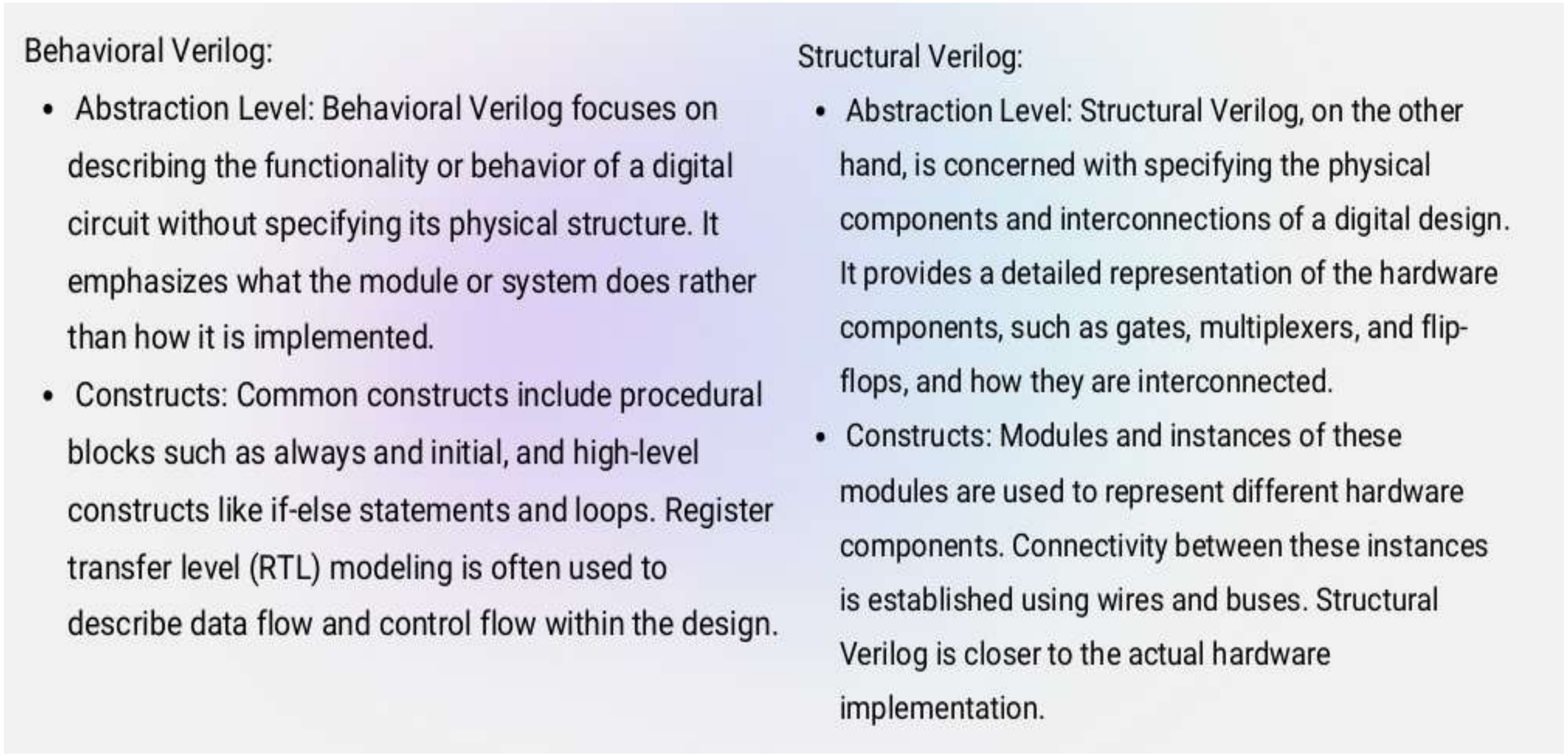
1. Generate for loop
2. Generate if else
3. Generate case

Example. A half adder will be instantiated N times in another top level design module called my_design using a generate for loop construct. The loop variable has to be declared using the keyword genvar which tells the tool that this variable is to be specifically used during elaboration of the generate block.



```
module ha ( input   a, b,
            output  sum, cout);

  assign sum  = a ^ b;
  assign cout = a & b;
endmodule

// A top level design that contains N instances of half adder
module my_design
    #(parameter N=4)
        (   input [N-1:0] a, b,
            output [N-1:0] sum, cout);

    // Declare a temporary loop variable to be used during
    // generation and won't be available during simulation
    genvar i;

    // Generate for loop to instantiate N times
    generate
        for (i = 0; i < N; i = i + 1) begin
          ha u0 (a[i], b[i], sum[i], cout[i]);
        end
    endgenerate
endmodule
```

Behavioural vs Structural Code



Behavioral Verilog:

- Abstraction Level: Behavioral Verilog focuses on describing the functionality or behavior of a digital circuit without specifying its physical structure. It emphasizes what the module or system does rather than how it is implemented.
- Constructs: Common constructs include procedural blocks such as always and initial, and high-level constructs like if-else statements and loops. Register transfer level (RTL) modeling is often used to describe data flow and control flow within the design.

Structural Verilog:

- Abstraction Level: Structural Verilog, on the other hand, is concerned with specifying the physical components and interconnections of a digital design. It provides a detailed representation of the hardware components, such as gates, multiplexers, and flip-flops, and how they are interconnected.
- Constructs: Modules and instances of these modules are used to represent different hardware components. Connectivity between these instances is established using wires and buses. Structural Verilog is closer to the actual hardware implementation.

Some further instructions

- Assignment has to be done individually !! (only project will be done in teams)
- Github Classrooms will be set up
- You can expect the first assign to be released in a week and it would cover the basic implementation of ALU.
- AI Usage Policy: You will need to attach the Chat Links for code generation with the Submission. We encourage you do the Assignment without any AI assistance.
- Scripts will be used to evaluate the Assignments/Projects so carefully follow the input/output guidelines attached in problem statement doc.

Do not indulge in any malpractices or plagiarism. Any such malpractice will have serious consequences.

Any Questions ?