SystemVerilog Essentials for Digital Hardware Design

Intelligent Architectures: 5LIL0

March 18, 2025

Dr. Federico Corradi

Department of Electrical Engineering, Electronic Systems Group

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Outline

Goal of this lesson

Introduction

Combinational Logic Description

Sequential Logic Description

Finite State Machines

Arrays

Parameters, Testbenches & Example (Tiny NN)

Goal of today

What will I learn?

You will be able to write SystemVerilog to describe hardware for building and simulating digital circuits.

Topic overview

- Key Topics:
  - Modules
  - Combinatorial Logic
  - Sequential Logic
- Advanced Concepts
  - Finite State Machines
  - Arrays
  - Assertion and Verification

Outline

Goal of this lesson

Introduction

Combinational Logic Description

Sequential Logic Description

Finite State Machines

Arrays

Parameters, Testbenches & Example (Tiny NN)

What is SystemVerilog?

Hardware description languages

- Hardware Description Language
  - high-level behavioural modeling
  - register transfer level (RTL) behavioural modeling
  - gate and transistor level netlists
  - timing models for simulations
  - design verification and test bench development
  - ...

What is SystemVerilog?

Hardware description languages

- Hardware Description Language
  - high-level behavioural modeling
  - register transfer level (RTL) behavioural modeling
  - gate and transistor level netlists
  - timing models for simulations
  - design verification and test bench development
  - ...

Register Transfer Level

We will use System Verilog as an RTL modeling language. It is much simpler than designing with gates, still help you think and understand like a hardware designer. RTL allow for modular, reusable, and scalable designs.

A very Brief History

- **1984** Verilog was created by Phillip Moorby at Gateway Design Automation as HDL.

Phillip Moorby - Wikipedia
Interview Computer History Museum: Youtube 2016

A very Brief History

- **1984** Verilog was created by Phillip Moorby at Gateway Design Automation as HDL.
- **1990** Cadence acquired Gateway and made Verilog open to the public.

Phillip Moorby - Wikipedia
Interview Computer History Museum: Youtube 2016

A very Brief History

- **1984** Verilog was created by Phillip Moorby at Gateway Design Automation as HDL.
- **1990** Cadence acquired Gateway and made Verilog open to the public.
- **1995** Verilog was standardized as IEEE 1364-1995, gaining widespread adoption in chip design, however,
  - Low-level, lengthy, static
  - Limited support for reusability and structures
  - Limited randomization (code/testbenches)

Phillip Moorby - Wikipedia
Interview Computer History Museum: Youtube 2016

A very Brief History

- **1984** Verilog was created by Phillip Moorby at Gateway Design Automation as HDL.
- **1990** Cadence acquired Gateway and made Verilog open to the public.
- **1995** Verilog was standardized as IEEE 1364-1995, gaining widespread adoption in chip design, however,
  - Low-level, lengthy, static
  - Limited support for reusability and structures
  - Limited randomization (code/testbenches)
- **2005** SystemVerilog was standardized as IEEE 1800-2005, replacing Verilog as a high-level Hardware and Verification DL.

Phillip Moorby - Wikipedia
Interview Computer History Museum: Youtube 2016

A very Brief History

- **1984** Verilog was created by Phillip Moorby at Gateway Design Automation as HDL.
- **1990** Cadence acquired Gateway and made Verilog open to the public.
- **1995** Verilog was standardized as IEEE 1364-1995, gaining widespread adoption in chip design, however,
  - Low-level, lengthy, static
  - Limited support for reusability and structures
  - Limited randomization (code/testbenches)
- **2005** SystemVerilog was standardized as IEEE 1800-2005, replacing Verilog as a high-level Hardware and Verification DL.
- **2012** IEEE 1800-2012 introduced enhanced synthesis and UVM (Universal Verification Methodology) support.

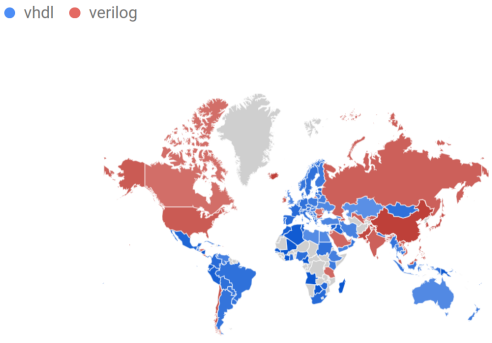
Phillip Moorby - Wikipedia
Interview Computer History Museum: Youtube 2016

A very Brief History

- **1984** Verilog was created by Phillip Moorby at Gateway Design Automation as HDL.
- **1990** Cadence acquired Gateway and made Verilog open to the public.
- **1995** Verilog was standardized as IEEE 1364-1995, gaining widespread adoption in chip design, however,
  - Low-level, lengthy, static
  - Limited support for reusability and structures
  - Limited randomization (code/testbenches)
- **2005** SystemVerilog was standardized as IEEE 1800-2005, replacing Verilog as a high-level Hardware and Verification DL.
- **2012** IEEE 1800-2012 introduced enhanced synthesis and UVM (Universal Verification Methodology) support.
- **Present** SystemVerilog remains the industry standard for design and verification of digital circuits.

Phillip Moorby - Wikipedia
Interview Computer History Museum: Youtube 2016

Hardware Description Languages Popularity (VHDL/Verilog)



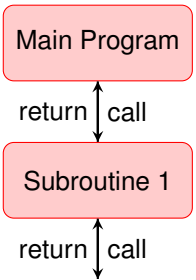
2015-2020

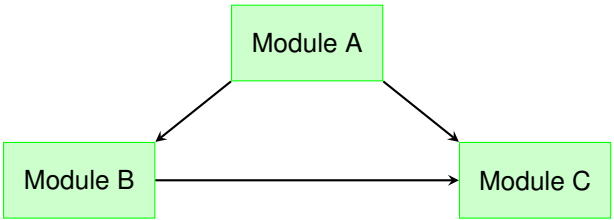
2020-2025

HDL vs Other (non-HDL) Programming Languages

HDL vs non-HDL programming

- Software (non-hdl) programs are composed of subroutines (mostly)
  - subroutines call each other
  - when in a callee, the caller's execution is paused

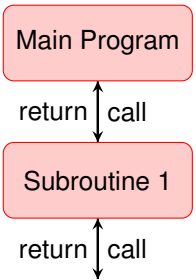


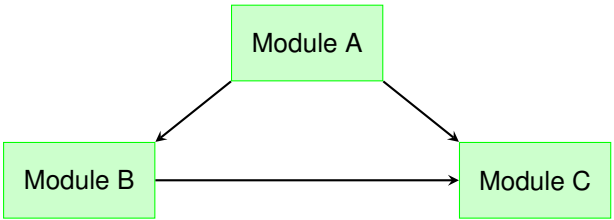


HDL vs Other (non-HDL) Programming Languages

HDL vs non-HDL programming

- Software (non-hdl) programs are composed of subroutines (mostly)
  - subroutines call each other
  - when in a callee, the caller's execution is paused
- Hardware descriptions are composed of modules (mostly)
  - a hirarchy of modules connected to each other
  - modules are active at the same time





Modules

What is a Module?

Is the basic building block. Each module defines a set of input and output ports which specify the input and output signals of the circuit. After the port definitions come any internal signal definition, finally followed by concurrent statements which specify the logic of the circuit.

```
module mymodule(a,b,c,f);
        output f
        input a,b,c;
        //description goes here
endmodule

//alternatively
module mymodule(input a,b,c, output f);
        //description goes here
endmodule
```

Modules

What is a Module?

Is the basic building block. Each module defines a set of input and output ports which specify the input and output signals of the circuit. After the port definitions come any internal signal definition, finally followed by concurrent statements which specify the logic of the circuit.

Modules

- The basic building block in SystemVerilog
  - Interfaces with outside using ports
  - Ports are either input or output (for now)

```
module mymodule(a,b,c,f);
        output f
        input a,b,c;
        //description goes here
endmodule

//alternatively
module mymodule(input a,b,c, output f);
        //description goes here
endmodule
```

Example: Multiplexer Module in SystemVerilog

```
module mux
 (
   input logic a, b, sel,
   output logic f
 );
 logic n_sel, f1, f2;

 and g1 (f1, a, n_sel);
 and g2 (f2, b, sel);
 or g3 (f, f1, f2);
 not g4 (n_sel, sel);

endmodule
```

[Short SystemVerilog Guide, Zachary Yedidia 2020]

2-1 multiplexer

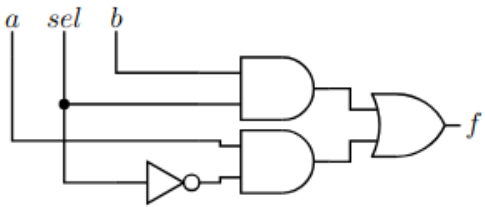
This logic circuit takes three inputs, a, b, and sel, and produces one output f. If sel is low, then $f = a$, otherwise $f = b$. We can express this using Boolean logic as:

$$f = a \cdot \neg sel + b \cdot sel$$

The Logic Data types

- logic : 0, 1, X (don't care), or Z (high impedance).

Example: Multiplexer Module in SystemVerilog




```
module mux
 (
  input logic a, b, sel,
  output logic f
 );
 logic n_sel, f1, f2;

 and g1 (f1, a, n_sel);
 and g2 (f2, b, sel);
 or g3 (f, f1, f2);
 not g4 (n_sel, sel);

endmodule
```

Instantiating Modules in SystemVerilog

Primitive Logic Gates and Module Instantiation

SystemVerilog provides built-in primitive logic gates, which are instantiated as concurrent modules. The order of instantiation does not affect circuit behavior.

```
module mymodule(a,b,c,f);
 output f;
 input a, b, c;
 //name of module to instantiate
 module_name inst_name(port_connections);
endmodule
```

Instantiating Modules in SystemVerilog

Primitive Logic Gates and Module Instantiation

SystemVerilog provides built-in primitive logic gates, which are instantiated as concurrent modules. The order of instantiation does not affect circuit behavior.

```
module mymodule(a,b,c,f);
 output f;
 input a, b, c;
 //name of module to instantiate
 module_name inst_name(port_connections);
endmodule
```

Passing Arguments

Pass by Position (less flexible): `mux mux_unit (X, Y, Z, W);`

Pass by Name (recommended): `mux mux_unit (.a(X), .b(Y), .sel(Z), .f(W));`

Vectors

We can now use instances of our 1-bit 2-1 multiplexer to create a 2-bit 2-1 multiplexer. Everything is the same except we are now passing 2-bit values.

```
module two_bit_mux
 (
  input logic [1:0] a ,b ,
  input logic sel,
  output logic [1:0] f
 );
 mux mux_1 (.f(f[0]), .a(a[0]), .b(b[0]), .sel);
 mux mux_2 (.f(f[1]), .a(a[1]), .b(b[1]), .sel);
endmodule
```

Outline

Goal of this lesson

Introduction

Combinational Logic Description

Sequential Logic Description

Finite State Machines

Arrays

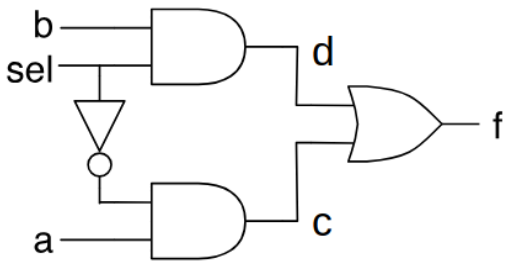
Parameters, Testbenches & Example (Tiny NN)

Continuous Assignment

Logic Behaviour

- Specify logic behaviorally by writing an expression to show how the signals are related to each other.

```
module mux2(a, b, sel, f);
  output f;
  input a, b, sel;
  logic c, d;

  assign c = a & (~sel);
  assign d = b & sel;
  assign f = c | d;

  // or alternatively
  assign f = sel ? b : a;
endmodule
```



Combinatorial Procedural Block

Can use *always_comb* procedural block to describe combinational logic using a series of sequential statements

- All *always_comb* blocks are independent and parallel to each other.

```
module mymodule(a, b, c, f);
  output f;
  input a, b, c;
  always_comb begin
    // Combinational logic
    // described
    // in C-like syntax
  end
endmodule
```

Procedural Behavioral Mux Description

```
module mux3(a, b, sel, f);
  output logic f;
  input a, b, sel;

  always_comb begin
    if (sel == 0) begin
      f = a;
    end
    else begin
      f = b;
    end
  end
endmodule
```

Note

- If we are going to *drive* f, we need to declare it as output logic.
- Important! For behavior to be combinational, every output (f) must be assigned in all possible control paths.
  - Why?

Procedural Behavioral Mux Description

```
module mux3(a, b, sel, f);
  output logic f;
  input a, b, sel;

  always_comb begin
    if (sel == 0) begin
      f = a;
    end
    else begin
      f = b;
    end
  end
endmodule
```

Note

- If we are going to *drive* f, we need to declare it as output logic.
- Important! For behavior to be combinational, every output (f) must be assigned in all possible control paths.
  - Why? Synthesis tool needs to correctly infers pure combinational logic. If a signal is not assigned in every control path, the synthesizer assumes that it should "remember" its previous value, which results in an unwanted latch instead of pure combinational logic.

Multiply-Assigned Values

```
module dosomething(...);
  ...
  always_comb begin
     b = ... something ...
  end
  always_comb begin
     b = ... something else ...
  end
endmodule
```

Note

- Both of these blocks execute concurrently
- So what is the value of b?

Multiply-Assigned Values

```
module dosomething(...);
  ...
  always_comb begin
     b = ... something ...
  end
  always_comb begin
     b = ... something else ...
  end
endmodule
```

Note

- Both of these blocks execute concurrently
- So what is the value of b? we don't know

Multi-Bit Value: example 4-bit 2-to-1 MUX

```
module mux4(a, b, sel, f);
  output logic [3:0] f;
  input [3:0] a, b;
  input sel;

  always_comb begin
    if (sel == 0) begin
      f = a;
    end
    else begin
      f = b;
    end
  end
endmodule
```

How It Works

This module implements a 4-bit 2-to-1 multiplexer (MUX):

- Takes two 4-bit inputs: *a* and *b*.
- Uses a 1-bit select signal *sel*.
- Outputs one of the two inputs based on *sel*:
  - If $sel = 0$, output $f = a$.
  - If $sel = 1$, output $f = b$.
- The *always_comb* block ensures combinational behavior.

Multi-Bit Constants and Concatenation

- Can give constants with specified number bits
  - In binary or hexadecimal
- Can concatenate with { and }

```
logic [3:0] a, b, c;
logic signed [3:0] d;
logic [7:0] e;
logic [1:0] f;
assign a = 4'b0010; // four bits, specified in binary
assign b = 4'hC; // four bits, specified in hex == 1100
assign c = 3; // == 0011
assign d = -2; // 2's complement == 1110 as bits
assign e = {a, b}; // concatenate == 0010_1100
assign f = a[2 : 1]; // two bits from middle == 01
```

Case Statements and "Don't-Cares"

```
module newmod(out, in0, in1, in2);
 input in0, in1, in2;
 output logic out;

 always_comb begin
  case ({in0, in1, in2})
   3'b000: out = 1;
   3'b001: out = 0;
   3'b010: out = 0;
   3'b011: out = x;
   3'b10x: out = 1;
  default: out = 0;
 endcase
 end
endmodule
```

Combinational logic circuit

case statement:

- Takes three 1-bit inputs: *in*0, *in*1, *in*2.
- Produces one output: *out*.
- The 3-bit input vector {in0, in1, in2} determines *out*:
  - '000 → out = 1'
  - '001, 010 → out = 0'
  - '011 → out = x' (unknown value)
  - '10x → out = 1' ('x' is a don't-care condition)
  - 'default → out = 0'
- 'x' in '10x' means any value (0 or 1).

Arithmetic Operators

- Standard arithmetic operators defined: + - * /
- Many subtleties here, so be careful:
  - four bit number + four bit number = five bit number (or..)
  - arbitrary division is difficult
  - Overflows (e.g. 4'b1000 + 4'b1000 =)
  - Use signed if you want values as 2's complement

[Wikipedia Two's complement]

```
logic signed [3:0] g, h, i;
logic signed [4:0] j;
assign g = 4'b0001; // == 1
assign h = 4'b0111; // == 7
assign i = g - h;
assign j = g - h;
//i == 4'b1010 == -6
//j == 5'b11010 == -7
```

Multiplication

- Multiply k bit number with m bit number
  - How many bits does the result have? (k+m)
- If you use fewer bits in your code
  - Gets least significant bits of the product (overflow)

```
logic signed [3:0] a, b, d;
logic signed [7:0] c;
assign a = 4'b1110; // -2
assign b = 4'b0111; // 7
assign d = a*b;
assign c = a*b;
//c = 8'b1111_0010 == -14
//d = 4'0010 == 2 (overflow)
```

Summary: Combinational and Sequential Logic in SystemVerilog

Combinational Logic (always_comb, assign)

- **Continuous Assignment**: Use `assign` for direct combinational logic.
- **`always_comb`**: Ensures automatic sensitivity list and avoids unintended latches.
- **Blocking Assignment (=)**: Used inside `always_comb` for immediate execution.
- Example:
  - `assign y = a & b;` (Continuous)
  - `always_comb begin y = a & b; end` (Procedural)

Outline

Goal of this lesson

Introduction

Combinational Logic Description

Sequential Logic Description

Finite State Machines

Arrays

Parameters, Testbenches & Example (Tiny NN)

Sequential Design

- Everything so far was purely combinational (i.e. stateless)
- What about sequential system? (flip-flops, register, finite state machines)
- New constructs
  - *always_ff*
  - @(*posedgeclk*...)
  - non-blocking assignment <=

Edge-Triggered Events

Edge-Triggered Events

- Variant of always block called *always_ff*
  - Indicates that block will be sequential logic (flip flops)
- Procedural block occurs only on a signal's edge
  - @(*posedge*...) or @(*negedge*...)

```
always_ff @(posedge clk, negedge reset_n) begin
// This procedure will be executed
// anytime clk goes from 0 to 1
// or anytime reset_n goes from 1 to 0
end
```

Flip Flop no reset

FF

q remembers (latches) what d was at the last clock edge
no reset

```
module flipflop(d, q, clk);
        input d, clk;
        output logic q;

        always_ff @(posedge clk) begin
                q <= d;
        end
endmodule
```

Flip Flop Asynchronous Reset

FF

q remembers what d was at the last clock edge
asynchronous reset

```
module flipflop_asyncr(d, q, clk, rst_n);
        input d, clk, rst_n;
        output logic q;

        always_ff @(posedge clk, negedge rst_n) begin
                if (rst_n == 0)
                        q <= 0;
                else
                        q <= d;
        end
endmodule
```

Flip Flop Synchronous Reset

FF

q remembers what d was at the last clock edge
synchronous reset

```
module flipflop_syncr(d, q, clk, rst_n);
  input d, clk, rst_n;
  output logic q;

  always_ff @(posedge clk) begin
    if (rst_n == 0)
      q <= 0;
    else
      q <= d;
  end
endmodule
```

16-Bit Flip Flop with Asynchronous Reset

```
module flipflop_asyncr(d, q, clk, rst_n);
  input [15:0] d;
  input clk, rst_n;
  output logic [15:0] q;
  always_ff @(posedge clk, negedge rst_n) begin
   if (rst_n == 0)
    q <= 0;
   else
    q <= d;
  end
endmodule
```

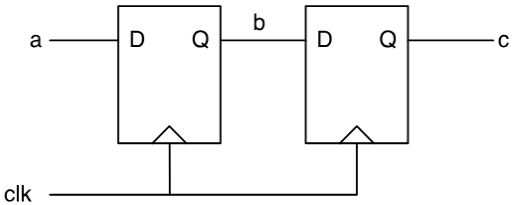
Behavior: Asynchronous Reset takes effect immediately

- On **negedge** `rst_n` (reset activated), *q* is set to '0'.
- On **posedge** `clk`, if reset is not active, *q* captures *d*.

Non-Blocking Assigments a <= b

Non-Blocking Assigment operator <=

- <= is the non-blocking assignment operator
  - All left-hand side values take new values concurrently
- This models synchronous logic!

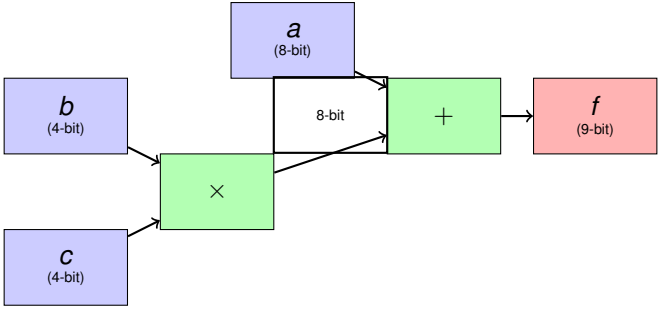


always_ff @(posedge clk) begin
b <= a;
c <= b;
end

Design example

Let's say we want to compute $f = a + b * c$

- b and c are 4-bit values, a is an 8-bit value, and f is a 9-bit result.
- First, we will build it as a combinational circuit.
- Then, we will add registers at its inputs and outputs.



Sequential Logic (always_ff, Non-Blocking Assignments)

Sequential Logic (`always_ff`, Flip-Flops)

- **Use `always_ff` for sequential logic**.
- **Non-Blocking Assignment (<=)**: Ensures all updates occur simultaneously at the clock edge.
- **Prevents Race Conditions**: <= avoids unintended dependencies.
- Example:
  - `always_ff @(posedge clk) q <= d;` (Correct)
  - `always_ff @(posedge clk) q = d;` (Incorrect – blocking used)

Non-Blocking (<=) vs. Blocking (=) Assignments

Key Differences

- **Non-blocking (<=)**: Updates occur simultaneously at the clock edge (used for flip-flops).
- **Blocking (=)**: Updates occur immediately, which can lead to incorrect behavior in sequential logic.

Correct (Non-Blocking)

```
always_ff @(posedge clk) begin
  b <= a;
  c <= b; // Uses old b value
end
```

Incorrect (Blocking)

```
always_ff @(posedge clk) begin
  b = a;
  c = b; // Uses new b value (wrong!)
end
```

Outline

Goal of this lesson

Introduction

Combinational Logic Description

Sequential Logic Description

Finite State Machines

Arrays

Parameters, Testbenches & Example (Tiny NN)

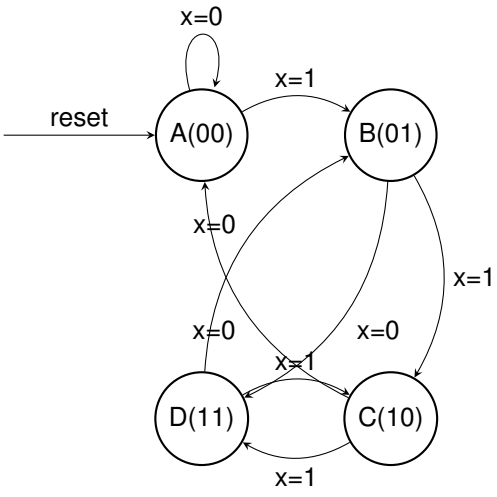
Finite State Machine (FSM)

FSM

A FSM is a mathematical model of computation that transitions between a finite number of states based on inputs and predefined rules.

Requires

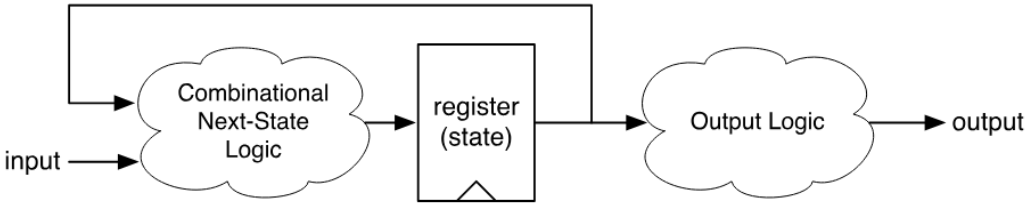
- state names
- output values
- transition values
- reset state



Each state is encoded in 2-bit, while input x is a single bit.

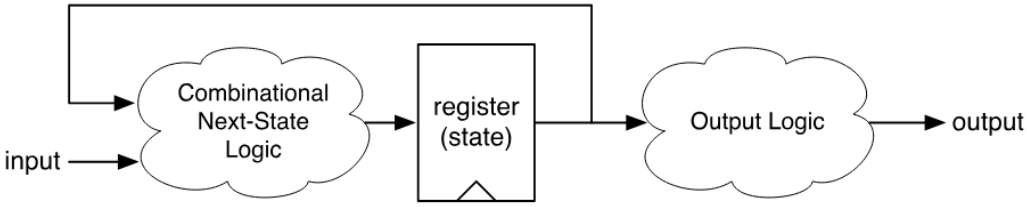
Finite State Machine (FSM)

What does an FSM look like when implemented?



Finite State Machine (FSM)

What does an FSM look like when implemented?



Combinational logic and registers: things we already know how to do!

Full FSM example (1/2)

```
module fsm(clk, rst, x, y);
  input clk, rst, x;
  output logic [1:0] y;
  enum { STATEA=2'b00, STATEB=2'b01, STATEC=2'b10,
         STATED=2'b11 } state, next_state;

  // next state logic
  always_comb begin
   case(state)
    STATEA: next_state = x ? STATEB : STATEA;
    STATEB: next_state = x ? STATEC : STATED;
    STATEC: next_state = x ? STATED : STATEA;
    STATED: next_state = x ? STATEC : STATEB;
   endcase
  end
// ... continued on next slide
```

Full FSM example (2/2)

```
// ... continued from previous slide
// register
always_ff @(posedge clk) begin
 if (rst)
  state <= STATEA;
 else
  state <= next_state;
 end
 // Output logic
 always_comb begin
  case(state)
   STATEA: y = 2'b00;
   STATEB: y = 2'b00;
   STATEC: y = 2'b11;
   STATED: y = 2'b10;
  endcase
 end
```

Outline

Goal of this lesson

Introduction

Combinational Logic Description

Sequential Logic Description

Finite State Machines

Arrays

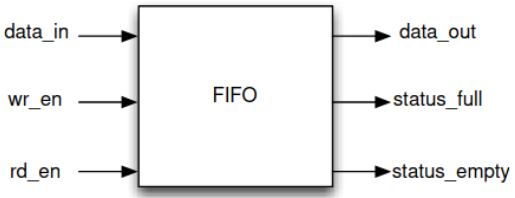
Parameters, Testbenches & Example (Tiny NN)

Arrays

```
module multidimarraytest();
logic [3:0] myarray [0:2]; // Array of 3 elements, each 4-bit wide

assign myarray[0] = 4'b0010;
assign myarray[1][3:2] = 2'b01;
assign myarray[1][1] = 1'b1;
assign myarray[1][0] = 1'b0;
assign myarray[2] = 4'hC; // Assign full value directly

initial begin
 $display("myarray[0] == %b", myarray[0]);  // element
 $display("myarray[1] == %b", myarray[1]);  // element
 $display("myarray[2] == %b", myarray[2]);  // element
 $display("myarray[1][2] == %b", myarray[1][2]);  // Single-bit access
 $display("myarray[2][1:0] == %b", myarray[2][1:0]); // Still valid
end
endmodule
```

Assertion

- Assertions are test constructs
  - Automatically validated as design is simulated
  - Written for properties that must always be true
- Makes it easier to test designs
  - Don't have to manually check for these conditions



Example FIFO queue

- Imagine you have a FIFO queue
  - When queue is full, it sets *status_full* to true
  - When queue is empty, it sets *status_empty* to true

Expected

- When *status_full* is true, *wr_en* must be false
- When *status_empty* is true, *rd_en* must be false

Assertion

```
// general form
assertion_name: assert(expression) pass_code;
else fail_code;
// example
always @(posedge clk) begin
 assert((status_full == 0) || (wr_en == 0))
 else $error("Tried to write to FIFO when full.");
end
```

Assertions

A procedural statement that checks an expression when statement is executed

Summary: SystemVerilog Essentials

Key Topics Covered

- **Introduction to SystemVerilog**
- **Modules and Instantiation**
  - Basic structure, ports, and logic gates
  - Parameterized modules for reusability
- **Combinational Logic**
  - Continuous assignments
  - *always_comb* procedural blocks
  - Case statements and "don't-care" conditions
- **Sequential Logic**
  - Flip-flops (*always_ff*, *posedge clk*)
  - Synchronous vs. asynchronous reset
  - Blocking ($=$) vs. non-blocking ($<=$) assignments

Summary: SystemVerilog Essential

Advanced Concepts

- **Finite State Machines (FSM)**
  - State transition logic and sequential design
- **Arrays**
  - Multi-bit registers and arrays
  - Synchronous and asynchronous read/write
- **Assertions for Verification**
  - Ensuring design correctness in simulation
  - Assertion-based testing for edge cases
- **Arithmetic Operations and Multiplication**
  - Handling signed numbers and overflow issues

Outline

Goal of this lesson

Introduction

Combinational Logic Description

Sequential Logic Description

Finite State Machines

Arrays

Parameters, Testbenches & Example (Tiny NN)

Module Parameters

- Parameters allow modules to be easily changed

```
module my_flipflop(d, q, clk, rst_n);
  parameter WIDTH=16;
  input [WIDTH-1:0] d;
  input clk, rst_n;
  output logic [WIDTH-1:0] q;
  ...
endmodule
```

- Instantiate and set parameter:

```
my_flipflop f0(d, q, clk, rst_n); // default
my_flipflop #(12) f0(d, q, clk, rst_n); // change parameter to 12
```

Creating Testbenches

What is a Testbench?

- A SystemVerilog module used for simulating and verifying digital designs.
- Does not have inputs or outputs, only internal signals.

Key Components of a Testbench

- **Device Under Test (DUT)**: Instantiates the circuit being tested.
- **Stimulus Generation**: Provides test input values.
- **Monitoring and Checking Outputs:** Verifies correct behavior.
- **Simulation Control**: Controls execution using ‘initial‘ blocks and ’$display()’.

Simple Neural Network: 3 Inputs, 2 Outputs (1/2)

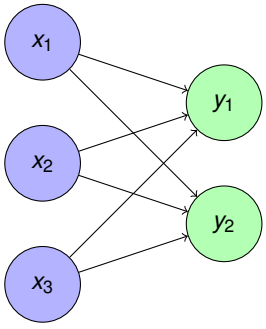
Neural Network Functionality

- Three input values: x_1, x_2, x_3.
- Two output neurons y_1, y_2.
- Each output is computed as:

$$y_1 = \text{ReLU}(w_{11}x_1 + w_{12}x_2 + w_{13}x_3)$$

$$y_2 = \text{ReLU}(w_{21}x_1 + w_{22}x_2 + w_{23}x_3)$$

- ReLU Activation: If the result is negative, output 0.



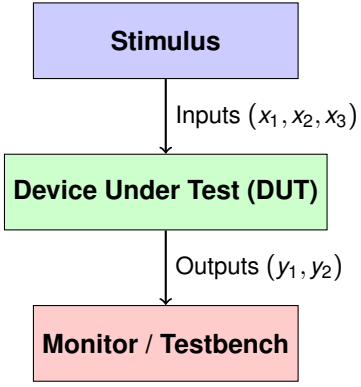
Simple Neural Network: 3 Inputs, 2 Outputs (2/2)

```
module simple_nn (
  input logic signed [7:0] x1, x2, x3,
  output logic signed [7:0] y1, y2
);
// Weights for each neuron
parameter signed [7:0] w11 = 2, w12 = -1, w13 = 3;
parameter signed [7:0] w21 = 1, w22 = 4, w23 = -2;

always_comb begin
 y1 = x1 * w11 + x2 * w12 + x3 * w13;
 y1 = (y1 < 0) ? 0 : y1; // ReLU

 y2 = x1 * w21 + x2 * w22 + x3 * w23;
 y2 = (y2 < 0) ? 0 : y2; // ReLU
end
endmodule
```

Testbench for Neural Network Module (1/2)

Testbench Overview

- Stimulus Generation: Provide different input values.
- Monitor Outputs: Check if 'y1' and 'y2' match expected values.
- Print Results: Use '$display' for debugging.



Testbench for Neural Network Module (2/2)

```
module tb_nn;
 logic signed [7:0] x1, x2, x3;
 logic signed [7:0] y1, y2;

 // Instantiate DUT (Device Under Test)
 simple_nn uut (.x1(x1), .x2(x2), .x3(x3), .y1(y1), .y2(y2));

 initial begin
  // Test Case 1
  x1 = 2; x2 = 3; x3 = -1;
  #10;
  $display("Inputs: %d %d %d -> Outputs: %d %d", x1, x2, x3, y1, y2);

  $stop; // or finish to exit the simulator
 end
endmodule
```

Simulation Execution and Expected Results

Running the Simulation

- Compile and run using a SystemVerilog simulator:

```
iverilog -g2012 -o out simple_nn.sv tb_nn.sv
vvp out
```

- The testbench applies test cases and displays the results.

Expected Output

```
Inputs: 2 3 -1 -> Outputs: 0 16
```

Key Takeaways

- Testbenches serve to verify hardware behavior.

Simulation Execution and Random Input Patterns

Stop and Finish

- $*stop* pauses simulation, allows debugging and manual continuation
- $*finish* end simulation, no debugging possible after execution
- #10 is a delay, assignments and display statements would execute instantaneously, making debugging harder.

```
initial begin
  repeat (5) begin
   x1 = $random % 256;  x2 = $random % 256; x3 = $random % 256;
   #10;
   $display("Inputs: %d %d %d -> Outputs: %d %d", x1, x2, x3, y1, y2);
  end
  $finish; // End simulation
end
```

Saving waveforms (dunpfiles & dumpvars) (1/2)

```
module tb_nn;
 logic signed [7:0] x1, x2, x3;
 logic signed [7:0] y1, y2;

 // Instantiate DUT (Device Under Test)
 simple_nn uut (.x1(x1), .x2(x2), .x3(x3), .y1(y1), .y2(y2));

 initial begin
  // Enable waveform dumping
  $dumpfile("waveform.vcd");  // Save waveform to this file
  $dumpvars(0, tb_nn);        // Dump all variables in tb_nn

  $display(" x1  x2  x3  ->  y1  y2");
  $display("-----------------------");

  //continue on next page ..
```

Saving waveforms (dunpfiles & dumpvars) (2/2)

```
    // .. continued from previous page
    x1 = 2; x2 = 3; x3 = -1;
    #10;
    $display("%3d %3d %3d -> %3d %3d", x1, x2, x3, y1, y2);
    // Test Case 2
    x1 = -2; x2 = 5; x3 = 4;
    #10;
    $display("%3d %3d %3d -> %3d %3d", x1, x2, x3, y1, y2);

   $stop;
  end
endmodule
```

File *.vcd

Can be viewed with "gtkwave waveform.vcd"

Compiling and Running the Testbench

Compile and Simulate with Icarus Verilog

- Use Icarus Verilog to compile the testbench and the neural network module.
- Run the compiled simulation to generate the waveform file.

Commands to Run

```
iverilog -g2012 -o out simple_nn.sv tb_nn_waveform.sv
vvp out
```

Viewing the Waveform with GTKWave

Step 3: Open GTKWave and Load the VCD File

- The simulation generates a Value Change Dump (VCD) file.
- Use GTKWave to visualize the waveform.

```
gtkwave waveform.vcd
```

GTKWave Interface

- Open the Signals panel on the left.
- Expand uut (Device Under Test).
- Select ‘x1‘, ‘x2‘, ‘x3‘, ‘y1‘, ‘y2‘**.
- Click "Append" to add them to the waveform view.

Understanding ‘$dumpfile‘ and ‘$dumpvars‘

Step 4: Waveform Dumping in Testbench

- $\$dumpfile(\text{"}waveform.vcd\text{"})$; $\rightarrow$ Saves waveform data to a VCD file.
- $\$dumpvars(0, tb_nn)$; $\rightarrow$ Dumps all signals in the testbench.
- You can also dump specific signals:

```
$dumpvars(1, uut.y1, uut.y2);
```

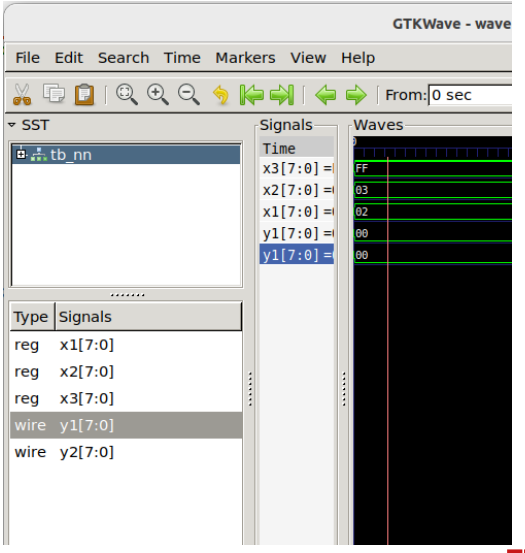
Modified Testbench Code

```
initial begin
 $dumpfile("waveform.vcd");  // Save waveform
 $dumpvars(0, tb_nn);        // Dump all signals
 x1 = 2; x2 = 3; x3 = -1; #10;
 $display("%3d %3d %3d -> %3d %3d", x1, x2, x3, y1, y2);
        ...
```

Analyzing the Waveform in GTKWave

Step 5: What You Should See

- The waveform should show:
  - Input signals: ‘x1, x2, x3‘
  - Computed outputs: ‘y1, y2‘
- Zoom in/out to analyze signal transitions.
- Ensure the expected values match the computed ‘y1‘ and ‘y2‘.



Summary: Extra Topics and Examples

Key Concepts Covered

- **Module Parameters**
  - Define flexible hardware designs.
  - Example: ‘parameter WIDTH=16;‘
- **Testbenches in SystemVerilog**
  - Stimulus generation, DUT instantiation.
  - Output monitoring using ‘$display()‘.
- **Neural Network Example**
  - Implemented 3-input, 2-output neurons with ReLU activation.
  - Verified output using testbench.

Simulation and Debugging

- **Saving and Viewing Waveforms**
  - Use ‘$dumpfile("waveform.vcd");‘ to save waveform.
  - Use ‘$dumpvars(0, *tb_nn*);‘ to track signals.
- **Running the Simulation**
  - Compile: *iverilog −g2012 −o out simple_nn.sv tb_nn.sv*
  - Execute: *vvp out*
- **Waveform Analysis in GTKWave**
  - Open VCD file: ‘gtkwave waveform.vcd‘
  - Add ‘x1, x2, x3, y1, y2‘ to waveform viewer.

Extra resources for Learning System Verilog

Learning a language is a tool, not a goal in itself!

- Quick tutorials available at ASIC-WORLD - asic-world.com
- SystemVerilog for Beginners systemverilog.io
- EDA playground Online playground
- Tool/hardware vendor specific information
  - Lots of info and examples in datasheets and whitepapers
  - see e.g. Xilinx, Altera, Lattice, Synopsys, Mentor, ...
- Best way to learn is to start with a (small) project