ÇUKUROVA ÜNİVERSİTESİ

Department of Computer Engineering

CEN 263 Digital Design

Autumn 2024

Lecture 4

Week 4 Outlines

- Combinational building blocks
  - Multiplexers
  - Decoders
  - Delays

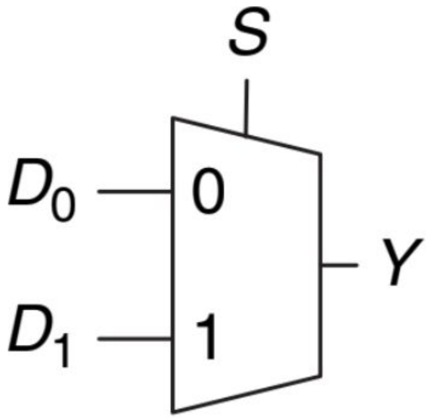
- Introduction to Verilog & SystemVerilog

Combinational building blocks

What is Multiplexers?

Combinational building blocks

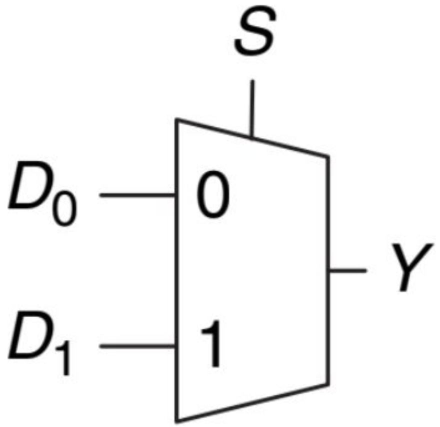
Multiplexers



A **Multiplexer** (MUX) is a digital device used to select one input from multiple data inputs and forward it to a single output line. The selection of the input is controlled by selection lines or control signals. Typically, a multiplexer has 2^n inputs and n selection lines. For example, a 4-to-1 MUX has 4 inputs and 2 selection lines to choose between them.

Combinational building blocks

Multiplexers



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

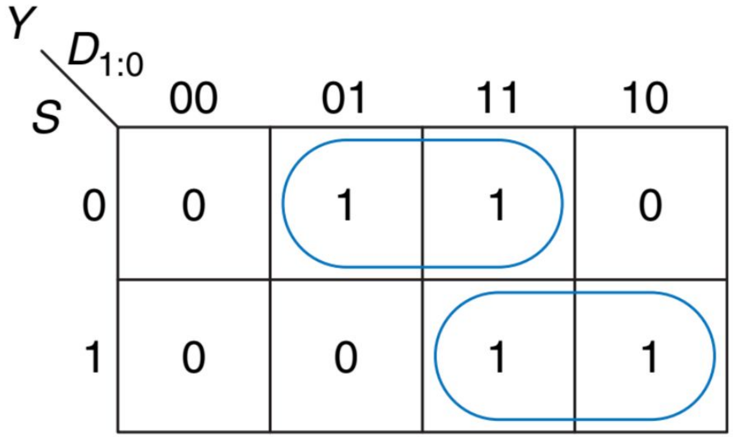
Figure 2.54 2:1 multiplexer symbol and truth table

Combinational building blocks

Multiplexers

S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

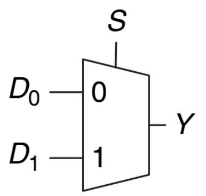
Figure 2.54 2:1 multiplexer symbol and truth table



$$Y = D_0\overline{S} + D_1 S$$

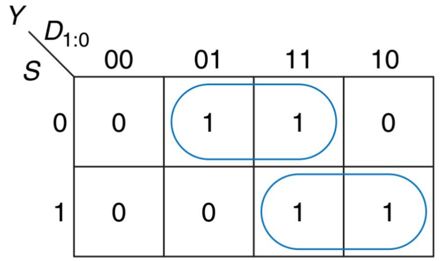
Combinational building blocks

Multiplexers



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2.54 **2:1 multiplexer symbol and truth table**



$$\begin{array}{c|cccc} S \backslash D_{1:0} & 00 & 01 & 11 & 10 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{array}$$

$Y = D_0\overline{S} + D_1S$

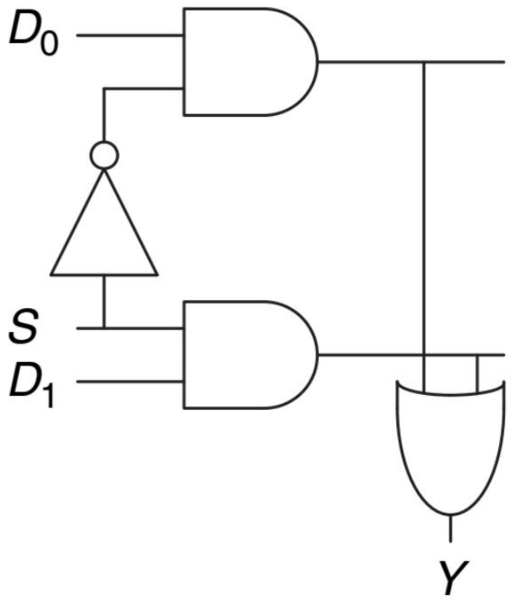
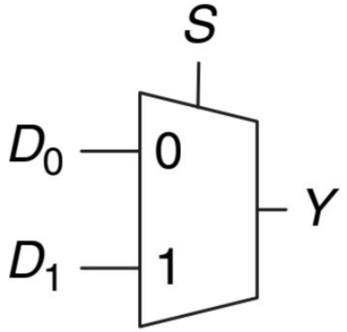


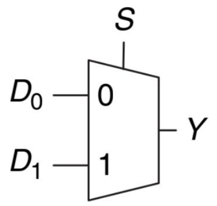
Figure 2.55 **2:1 multiplexer implementation using two-level logic**

Combinational building blocks

What You Can Do with a Multiplexer?



Combinational building blocks



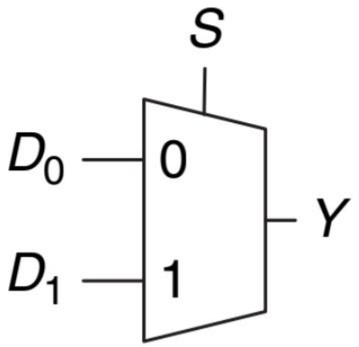
What You Can Do with a Multiplexer?

Multiplexers are highly versatile and can be used in a variety of digital applications:

- **Data Routing**: Direct data from one of several inputs to a single output line.
- **Signal Selection**: Select signals from multiple sources, such as sensors or communication channels, for processing.
- **Function Implementation**: Construct simple combinational logic functions and implement truth tables using multiplexers.
- **Microprocessor Systems**: Used in microprocessors for selecting memory locations, I/O lines, or registers based on certain control signals.
- **Switching Networks**: Used in telecommunications to allow multiple signals to share a single line.

Combinational building blocks

What You Can Do with a Multiplexer?



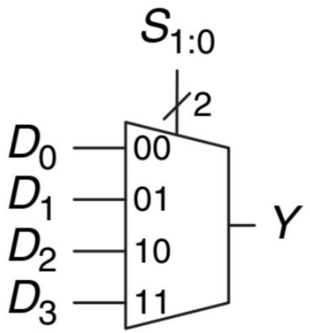


Figure 2.57 **4:1 multiplexer**

Combinational building blocks

Why Multiplexers are important?

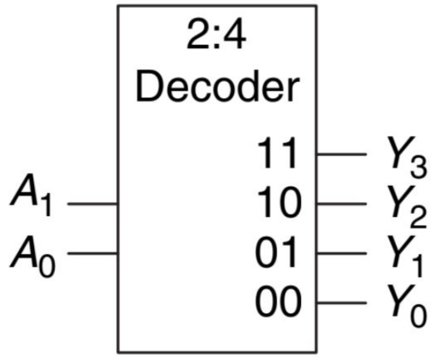
- **Space and Cost-Efficient**: Multiplexers allow a reduction in the number of wires and connections, saving space and minimizing hardware complexity.
- **Scalability**: They make it easier to design large and complex systems by modularly managing multiple inputs.
- **Flexibility in Design**: Multiplexers provide a way to dynamically control which signal is processed or forwarded, which is essential for systems needing rapid switching between inputs.
- **Foundational Component in Digital Systems**: Multiplexers are basic components in designing various digital systems, such as computers, routers, and communication systems, where efficient data routing is essential.

Combinational building blocks

What is Decoders?

Combinational building blocks

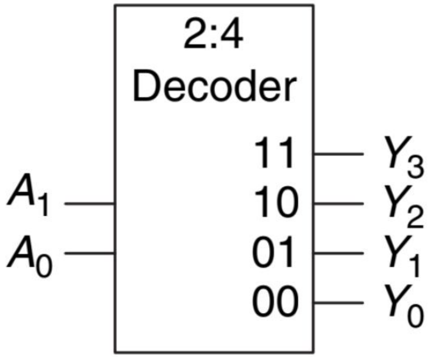
Decoders



A **Decoder** is a digital logic circuit that takes a binary input and activates a specific output line based on that input. It essentially "decodes" binary input into a single active output line, making it the opposite of a multiplexer in some ways. For example, a 2-to-4 decoder has two input lines and four output lines, with only one output being active (high) at a time based on the binary value of the inputs.

Combinational building blocks

Decoders

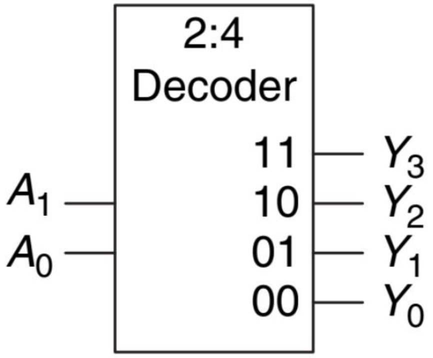


A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figure 2.63 2:4 decoder

Combinational building blocks

Decoders



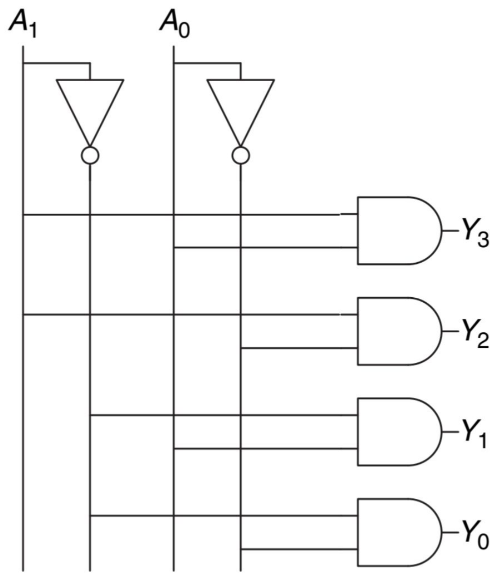


Figure 2.64 **2:4 decoder implementation**

Combinational building blocks

Applications of Decoders

Decoders are highly useful in digital electronics and are often employed in applications where a specific response is needed based on binary input values:

- **Memory Address Decoding**: Decoders are used to select specific memory locations in RAM or ROM based on the address provided by a microprocessor.
- **Binary to One-Hot Encoding**: Convert binary values to "one-hot" encoding, where only one line is high, which is useful for selecting specific outputs in digital circuits.
- **Demultiplexing**: Used in demultiplexing operations to direct a single input to one of multiple outputs.
- **Display Systems**: Decoders can control displays (like seven-segment displays) by decoding binary numbers into display codes.
- **Microprocessor Control**: Used in control units to activate specific control signals based on the instruction being executed.
- **Logic Implementation**: Can simplify the implementation of combinational logic functions by generating specific outputs based on input values.

Combinational building blocks

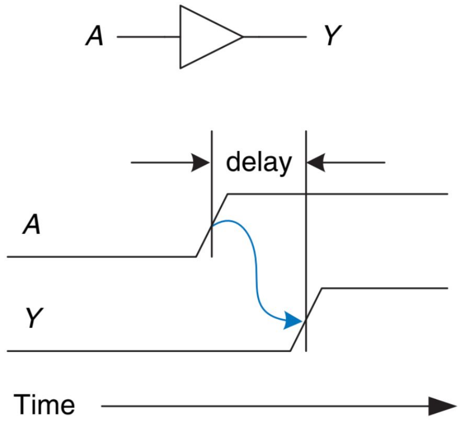
Why Decoders are important?

- **Efficient Control in Digital Systems**: Decoders are crucial in systems requiring precise control of components based on binary input, such as activating specific memory addresses or I/O ports.
- **Minimizing Complexity**: Decoders reduce wiring complexity in digital systems by ensuring that only one line is activated per binary input, reducing the number of connections needed.
- **Foundation for More Complex Circuits**: Decoders are foundational in the design of larger digital systems and can be combined with other logic components like multiplexers for more advanced functionalities.
- **Essential in Computing**: They play a significant role in computing hardware, especially in memory management, data processing, and control units of CPUs.

Combinational building blocks

Timing

Propagation Delay and **Contamination Delay** are key concepts in digital circuit design, particularly in timing analysis, where they affect the performance, stability, and reliability of a circuit.



Combinational building blocks

Timing

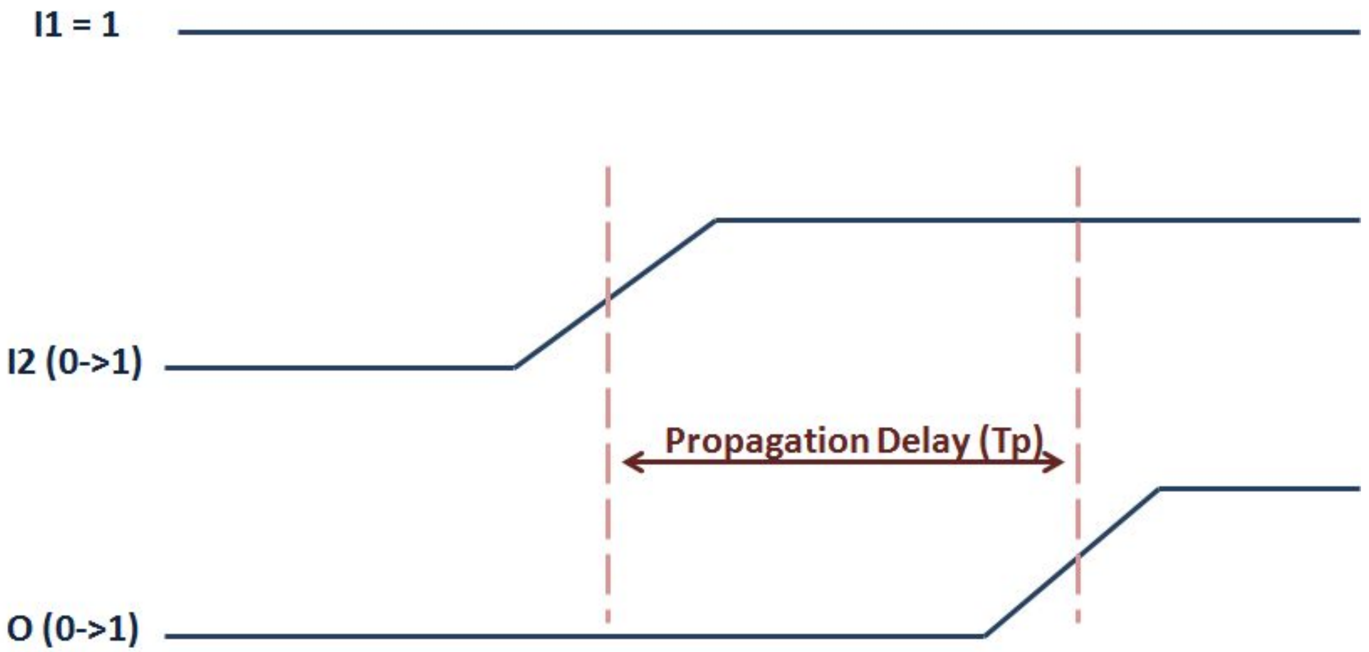
Propagation Delay (t_{pd})

Propagation delay, often denoted as t_{pd}, is the time it takes for a change in the input of a digital circuit to be reflected at the output. Specifically, it is measured from when an input signal makes a transition (e.g., from low to high or high to low) until the output signal reaches a stable state.

- **Definition**: The maximum time it takes for a change in input to produce a valid, stable output.
- **Measurement**: Propagation delay is typically measured from the 50% point of the input transition to the 50% point of the output transition.

Combinational building blocks

Timing



Combinational building blocks

Timing

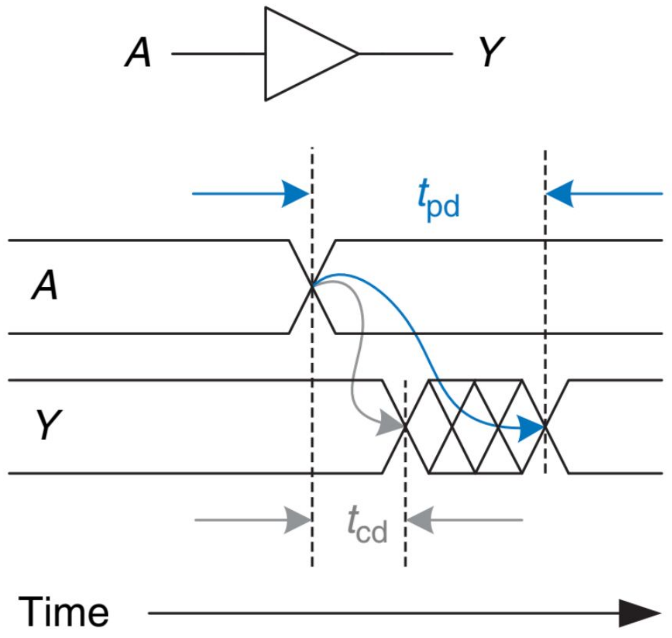
Contamination Delay (t_{cd})

Contamination delay, or t_{cd}, represents the minimum time after an input change before the output may start to change. It's essentially the earliest moment the output might begin responding to an input change.

- **Definition**: The minimum time it takes for the effects of an input change to begin propagating through to the output.
- **Measurement**: Contamination delay is measured from the 50% point of the input transition to the point where the output signal starts moving towards a new value.

Combinational building blocks

Timing



Combinational building blocks

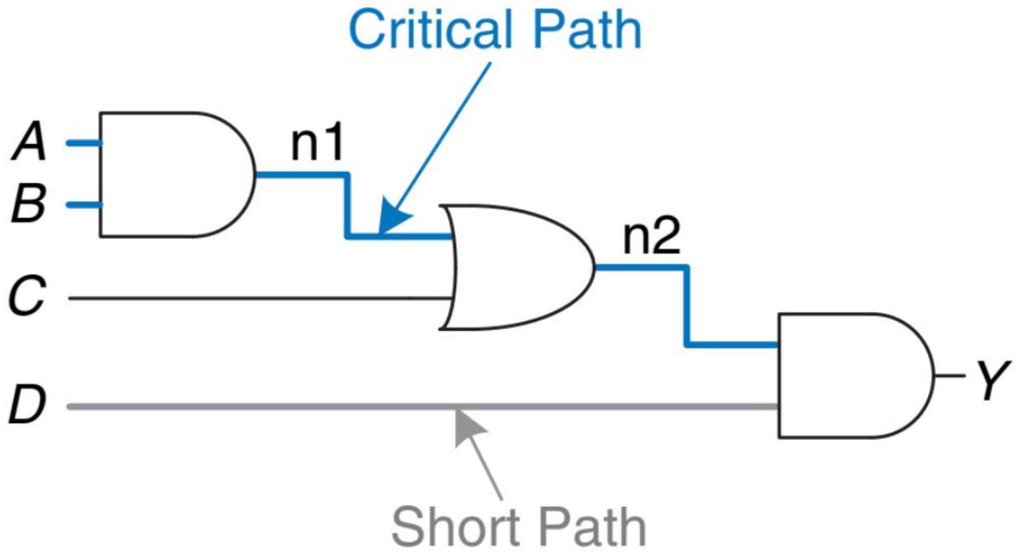
Timing

Key Differences

- **Propagation Delay** is the maximum delay from input to output, ensuring a fully stable output.
- **Contamination Delay** is the minimum delay from input to output, indicating the earliest possible change in the output.

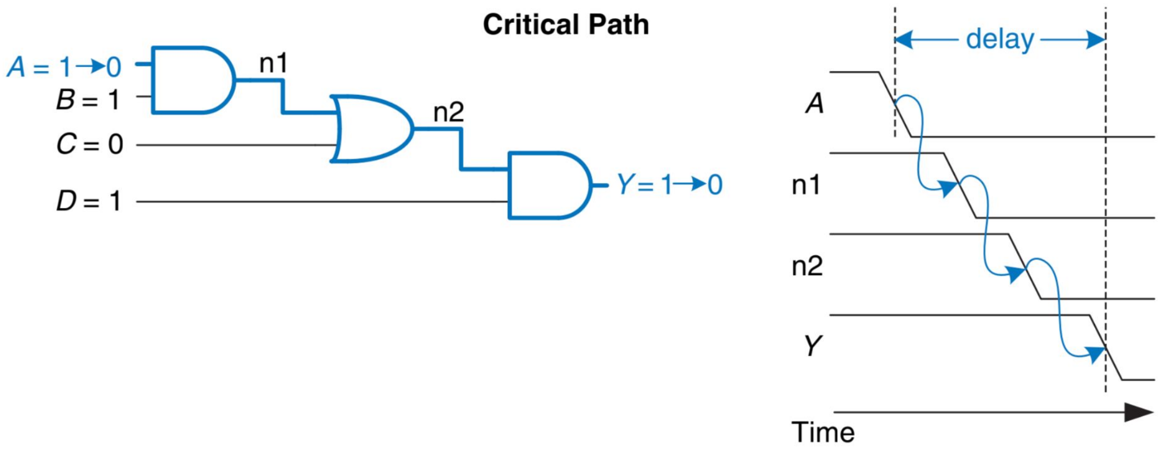
Combinational building blocks

Timing

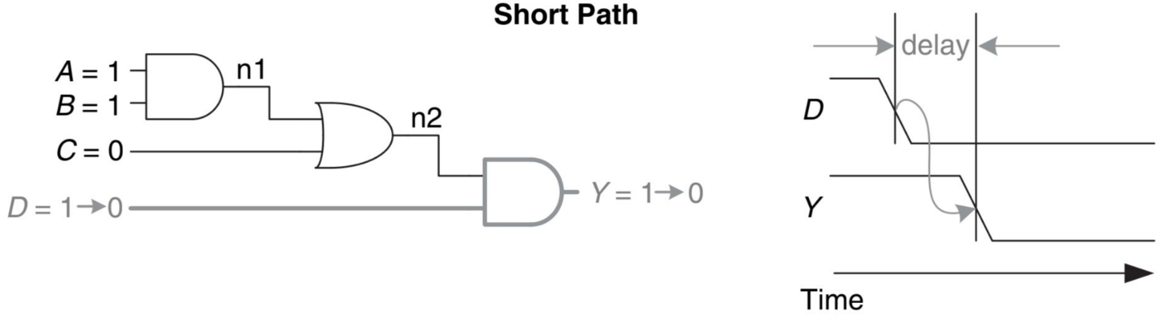


Combinational building blocks

Critical Path



Short Path



Combinational building blocks

Applications of Propagation and Contamination Delay

These delays are important in timing analysis, which helps in ensuring that digital circuits perform as expected, especially in synchronous systems:

- **Clock Cycle Determination**: Propagation delay is a key factor in determining the minimum clock cycle time in synchronous circuits. The clock period must be long enough to accommodate the maximum propagation delay.
- **Hold Time Constraints**: Contamination delay is critical in ensuring hold time constraints are met, which are the minimum times that input signals need to stay stable after the clock edge.
- **Signal Stability**: Understanding both delays helps in managing signal stability, ensuring that transitions do not lead to glitches or incorrect data being latched.
- **Designing Pipelines**: In pipelined architectures, knowing these delays helps balance stages, ensuring data moves smoothly without causing bottlenecks.

Combinational building blocks

Importance of Propagation and Contamination Delay in Digital Circuit Design

- **Ensuring Reliable Timing**: Accurate timing analysis with propagation and contamination delays ensures that signals are latched correctly, avoiding errors due to timing mismatches.
- **Preventing Glitches**: By accounting for contamination delays, designers can prevent glitching in outputs, where signals might briefly change to an unintended value.
- **Performance Optimization**: Propagation delay directly impacts the maximum speed at which a circuit can operate, so optimizing these delays allows for faster, more efficient designs.
- **Synchronization**: Proper synchronization in multi-stage designs (like pipelines and flip-flops) depends on managing these delays, ensuring data moves smoothly through each stage.

Take a look at Glitches

(not included in exams)

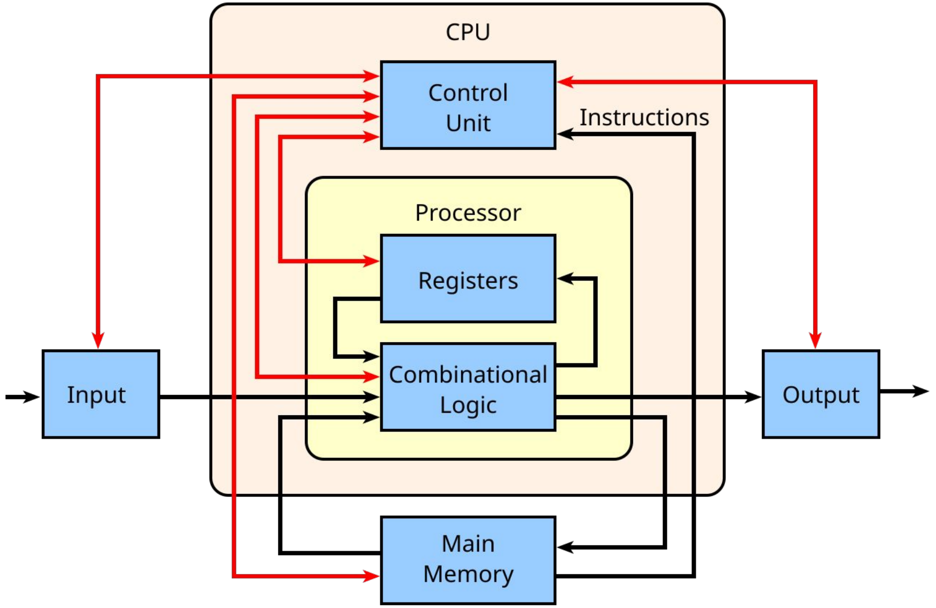
End of Chapter 2

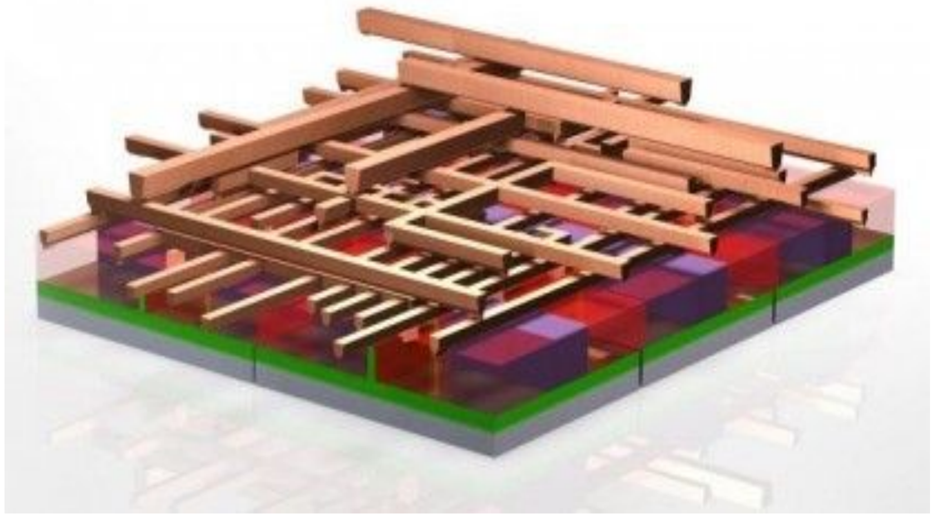
Verilog & SystemVerilog

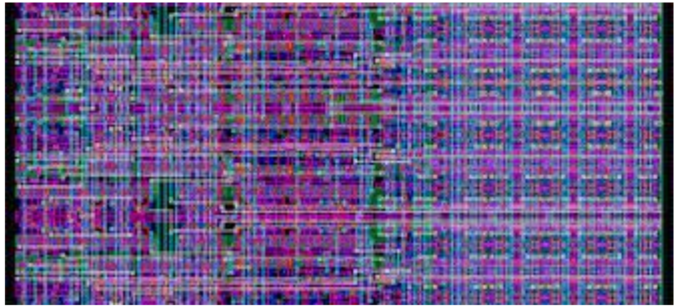
Introduction

- What is *Hardware Description Language (HDL)*?
- Why we go for SystemVerilog?
- What is verification?
- How to verify a design?
- Why SystemVerilog for verification?
- Design features

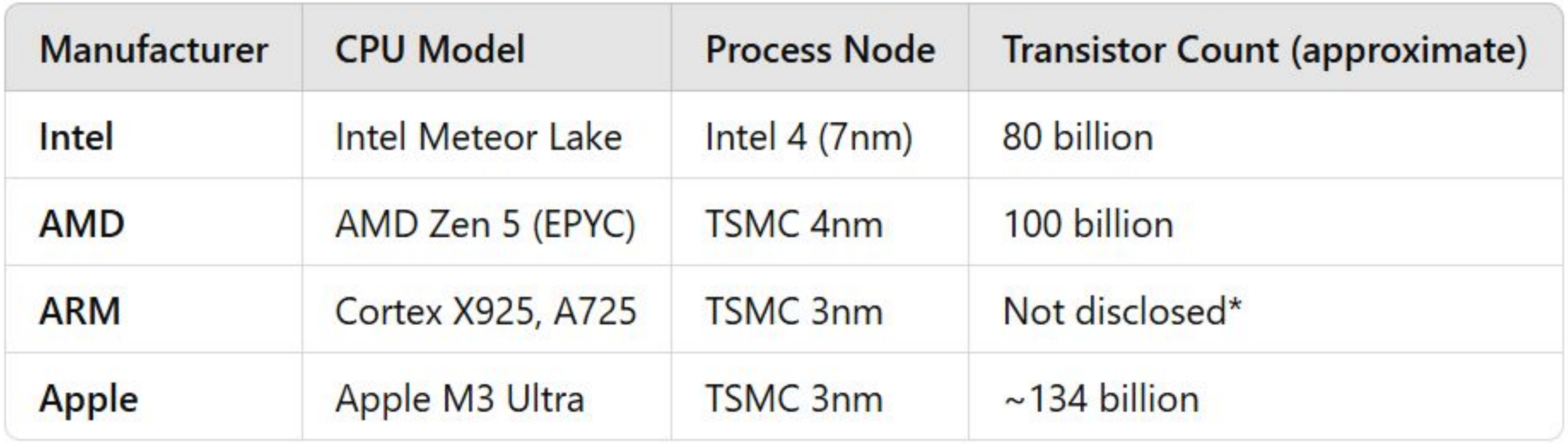
What is Hardware Description Language (HDL)?







What is Hardware Description Language (HDL)?



Manufacturer	CPU Model	Process Node	Transistor Count (approximate)
Intel	Intel Meteor Lake	Intel 4 (7nm)	80 billion
AMD	AMD Zen 5 (EPYC)	TSMC 4nm	100 billion
ARM	Cortex X925, A725	TSMC 3nm	Not disclosed*
Apple	Apple M3 Ultra	TSMC 3nm	~134 billion

Can I draw 80 billion transistors by hand?

What is Hardware Description Language (HDL)?

- A **Hardware Description Language (HDL)** is a specialized programming language used to describe the structure, design, and operation of electronic circuits, particularly digital logic circuits such as microprocessors, **FPGAs (Field-Programmable Gate Arrays)**, and **ASICs (Application-Specific Integrated Circuits)**.

- Unlike general-purpose programming languages that describe software algorithms, **HDLs describe the actual physical connections and logic gates** within digital hardware.

What is Hardware Description Language (HDL)?

Or, in short, hardware description languages have been developed to describe hardware in software and save engineering time.

What is Hardware Description Language (HDL)?

Key Features of HDLs:

Concurrent Execution: HDLs, like VHDL (VHSIC Hardware Description Language), Verilog, and SystemVerilog are designed for **concurrent execution of code, which matches the parallel nature of digital circuits**. This is different from typical sequential execution in software programming.

What is Hardware Description Language (HDL)?

Key Features of HDLs:

Simulation and Synthesis: HDLs allow designers to simulate digital circuits to verify their behavior before hardware is built. They can also be synthesized, meaning they can be converted into a netlist—a description of the components and their connections—ready for fabrication onto a chip.

What is Hardware Description Language (HDL)?

Key Features of HDLs:

Modular Design: HDLs facilitate modular design, enabling complex systems to be built from smaller, reusable components. This makes it easier to manage, test, and modify large digital systems.

What is Hardware Description Language (HDL)?

Popular HDLs:

Verilog: A straightforward and widely-used HDL known for ease of use, Verilog is highly effective for both behavioral and structural design, especially in digital logic and system-on-chip (SoC) applications.

What is Hardware Description Language (HDL)?

Popular HDLs:

VHDL (VHSIC Hardware Description Language): Known for its strong typing and modularity, VHDL is often used in industries that require rigorous, reliable designs, such as aerospace and defense. Its verbose syntax helps in creating complex, high-integrity systems.

aselsan roketsan

What is Hardware Description Language (HDL)?

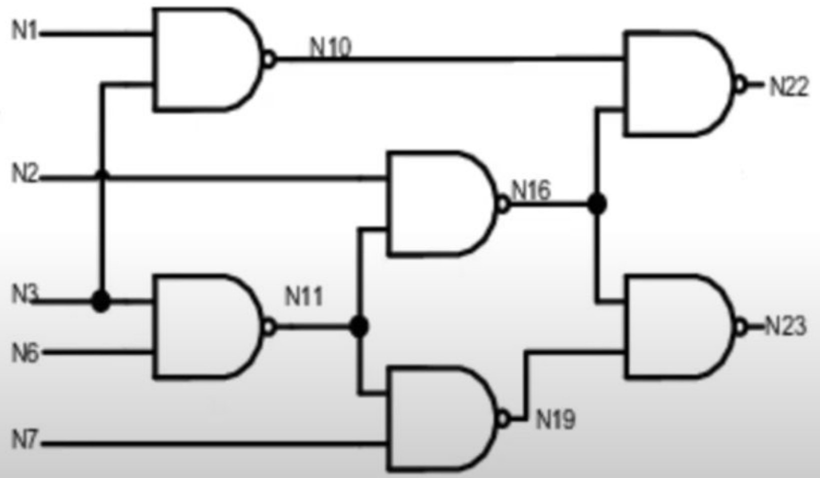
Popular HDLs:

SystemVerilog: An enhanced version of Verilog, SystemVerilog is used **not only for hardware design but also extensively for hardware verification**. It introduces **object-oriented programming (OOP)** and advanced verification features, including assertions and randomization, which make it highly effective in building and testing complex digital systems.

intel® AMD NVIDIA ARM

What is Hardware Description Language (HDL)?

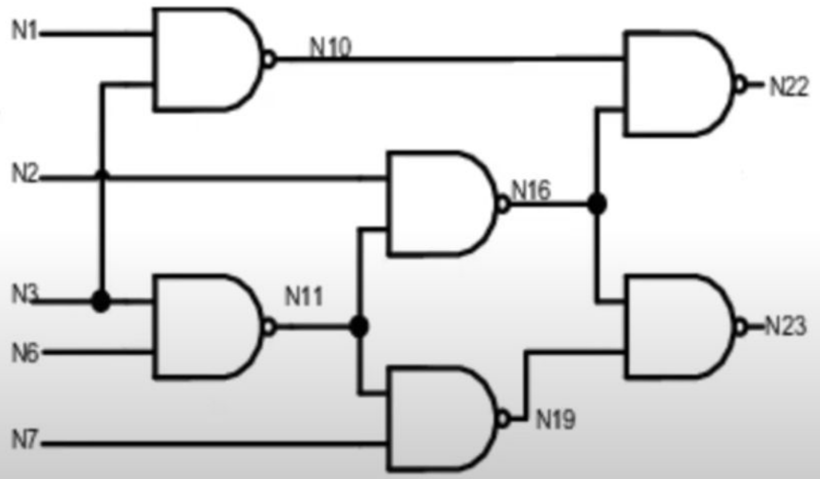
An example of a logic gate design with Verilog



```
module c17 (N1,N2,N3,N6,N7,N22,N23);
input N1,N2,N3,N6,N7;
output N22,N23;
wire N10,N11,N16,N19;
nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5 (N22, N10, N16);
nand NAND2_6 (N23, N16, N19);
endmodule
```

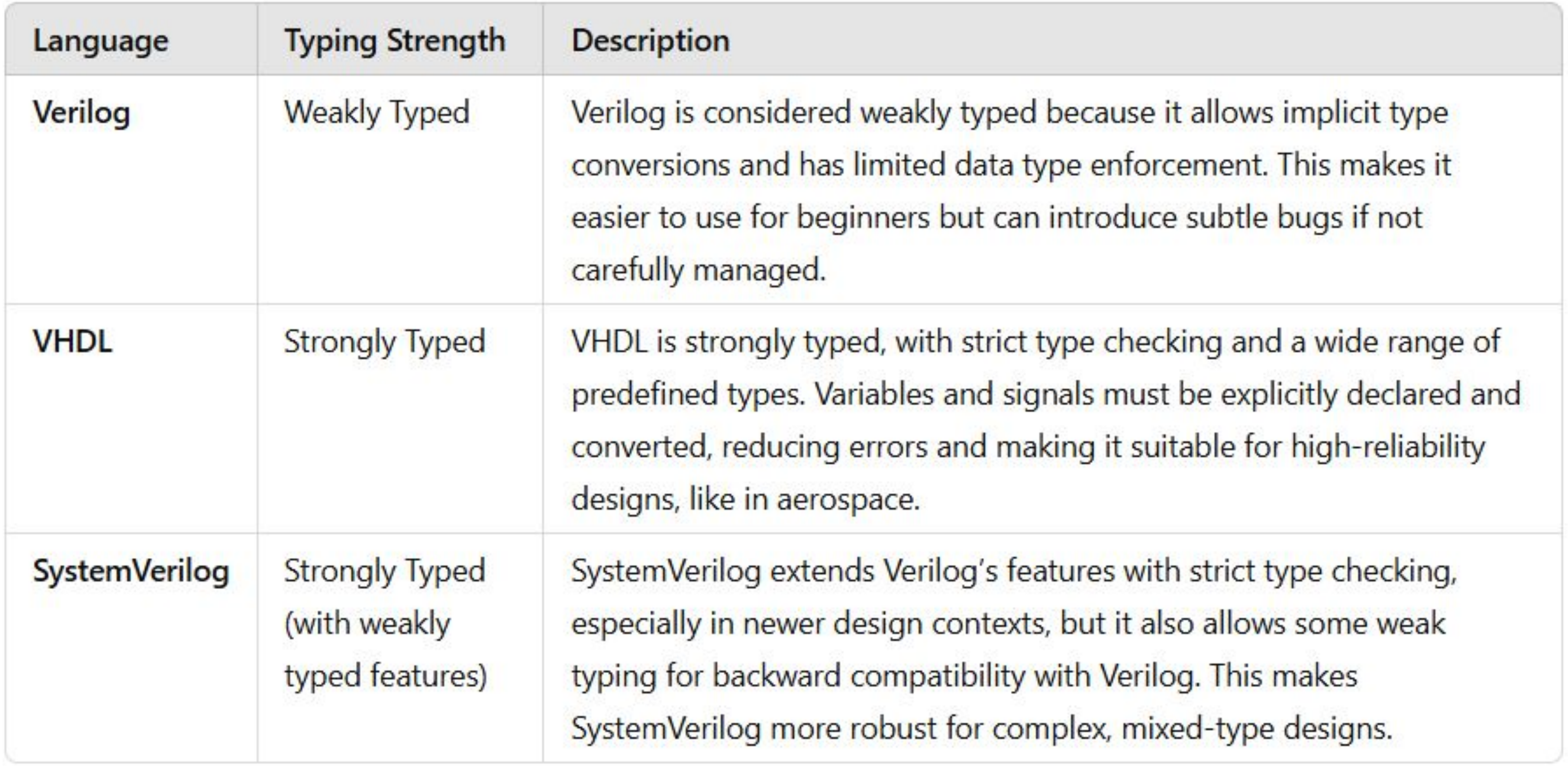
What is Hardware Description Language (HDL)?

Verification?



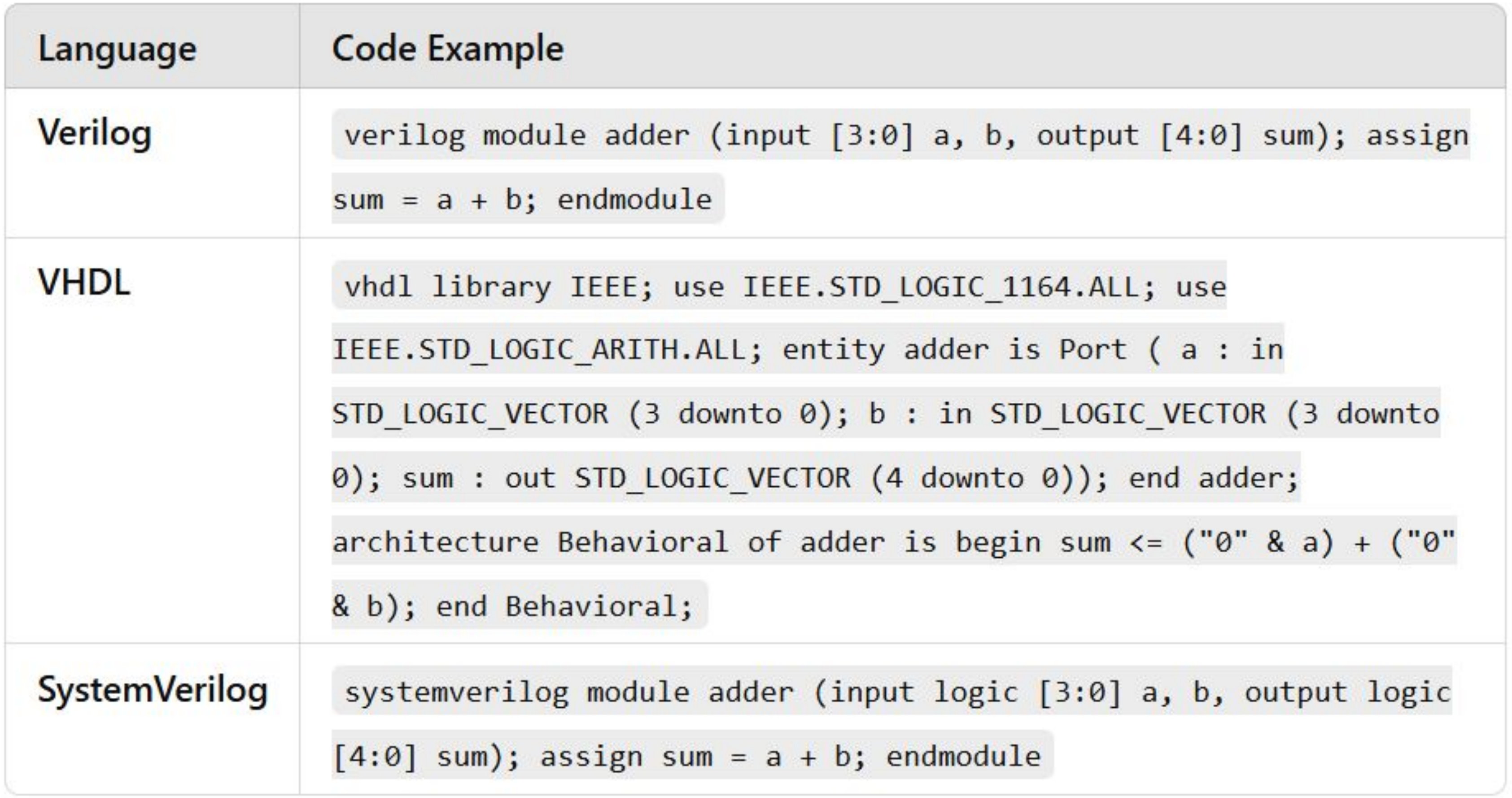
```
module c17 (N1,N2,N3,N6,N7,N22,N23);
input N1,N2,N3,N6,N7;
output N22,N23;
wire N10,N11,N16,N19;
nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5 (N22, N10, N16);
nand NAND2_6 (N23, N16, N19);
endmodule
```

What is Hardware Description Language (HDL)?



Language	Typing Strength	Description
Verilog	Weakly Typed	Verilog is considered weakly typed because it allows implicit type conversions and has limited data type enforcement. This makes it easier to use for beginners but can introduce subtle bugs if not carefully managed.
VHDL	Strongly Typed	VHDL is strongly typed, with strict type checking and a wide range of predefined types. Variables and signals must be explicitly declared and converted, reducing errors and making it suitable for high-reliability designs, like in aerospace.
SystemVerilog	Strongly Typed (with weakly typed features)	SystemVerilog extends Verilog's features with strict type checking, especially in newer design contexts, but it also allows some weak typing for backward compatibility with Verilog. This makes SystemVerilog more robust for complex, mixed-type designs.

A basic 4-bit adder



Language	Code Example
Verilog	`verilog module adder (input [3:0] a, b, output [4:0] sum); assign sum = a + b; endmodule`
VHDL	`vhdl library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL; entity adder is Port (a : in STD_LOGIC_VECTOR (3 downto 0); b : in STD_LOGIC_VECTOR (3 downto 0); sum : out STD_LOGIC_VECTOR (4 downto 0)); end adder; architecture Behavioral of adder is begin sum <= ("0" & a) + ("0" & b); end Behavioral;`
SystemVerilog	`systemverilog module adder (input logic [3:0] a, b, output logic [4:0] sum); assign sum = a + b; endmodule`

A basic 4-bit adder

- **Verilog**: This module defines a 4-bit adder using the `assign` statement to perform addition. It is straightforward and uses basic syntax, suitable for simple logic.
- **VHDL**: The VHDL example includes a library and package imports for standard logic types. VHDL requires explicit definition of the entity (interface) and architecture (implementation) blocks, making it more verbose but with stricter typing. The sum is calculated by concatenating a leading zero to each input vector to handle carry-over bits.
- **SystemVerilog**: SystemVerilog extends Verilog syntax, adding more robust data type options (like `logic`). This example is similar to Verilog but uses `logic` to define variables, which offers better type checking and debugging.

A basic 4-bit adder

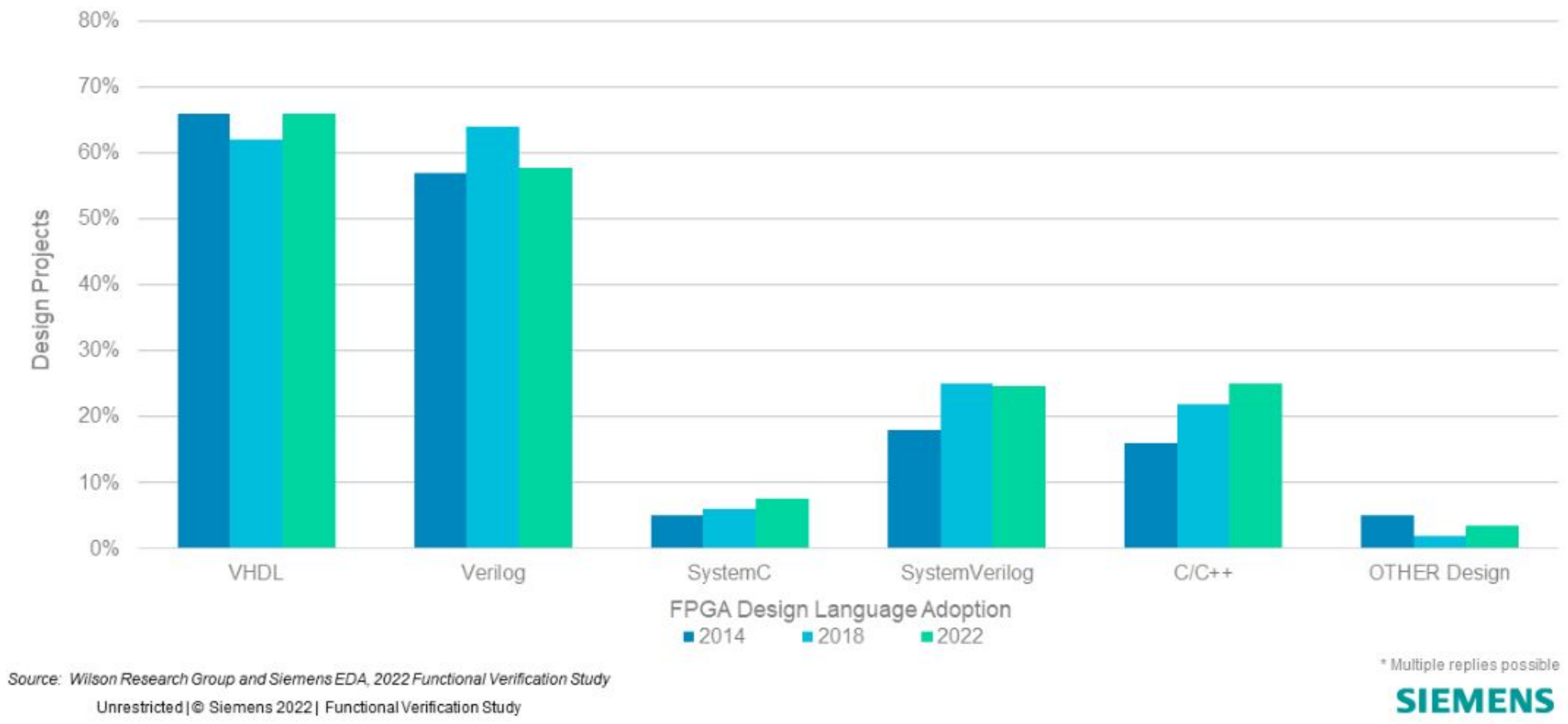
- **P.S.** The reason for using a 5-bit output for a 4-bit addition operation is to account for the possibility of a **carry-out** from the most significant bit.

- When two 4-bit numbers are added, the maximum possible sum is

  1111(15) + 1111(15) = 11110 (30)

  in binary, which requires 5 bits to represent (`11110`).

- If we only used a 4-bit output, any sum greater than `1111` would be truncated, resulting in an incorrect result due to overflow.



Source: Wilson Research Group and Siemens EDA, 2022 Functional Verification Study

Unrestricted | © Siemens 2022 | Functional Verification Study

Figure 6-1. Trends in languages used for FPGA design

Source: Prologue: The 2022 Wilson Research Group Functional Verification Study

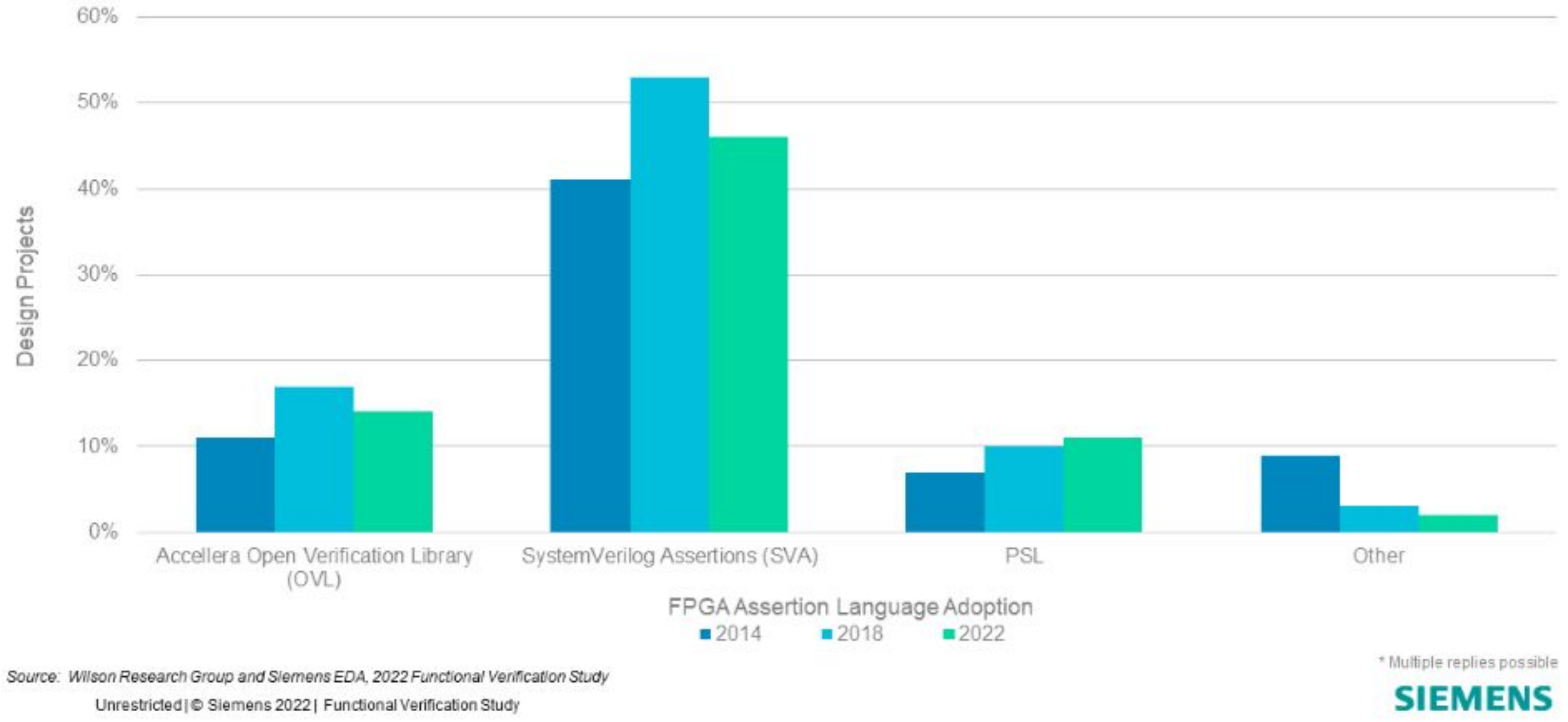


Figure 6-4. FPGA methodology adoption trends

[Source: Prologue: The 2022 Wilson Research Group Functional Verification Study]()

ASIC design language adoption (DUT)

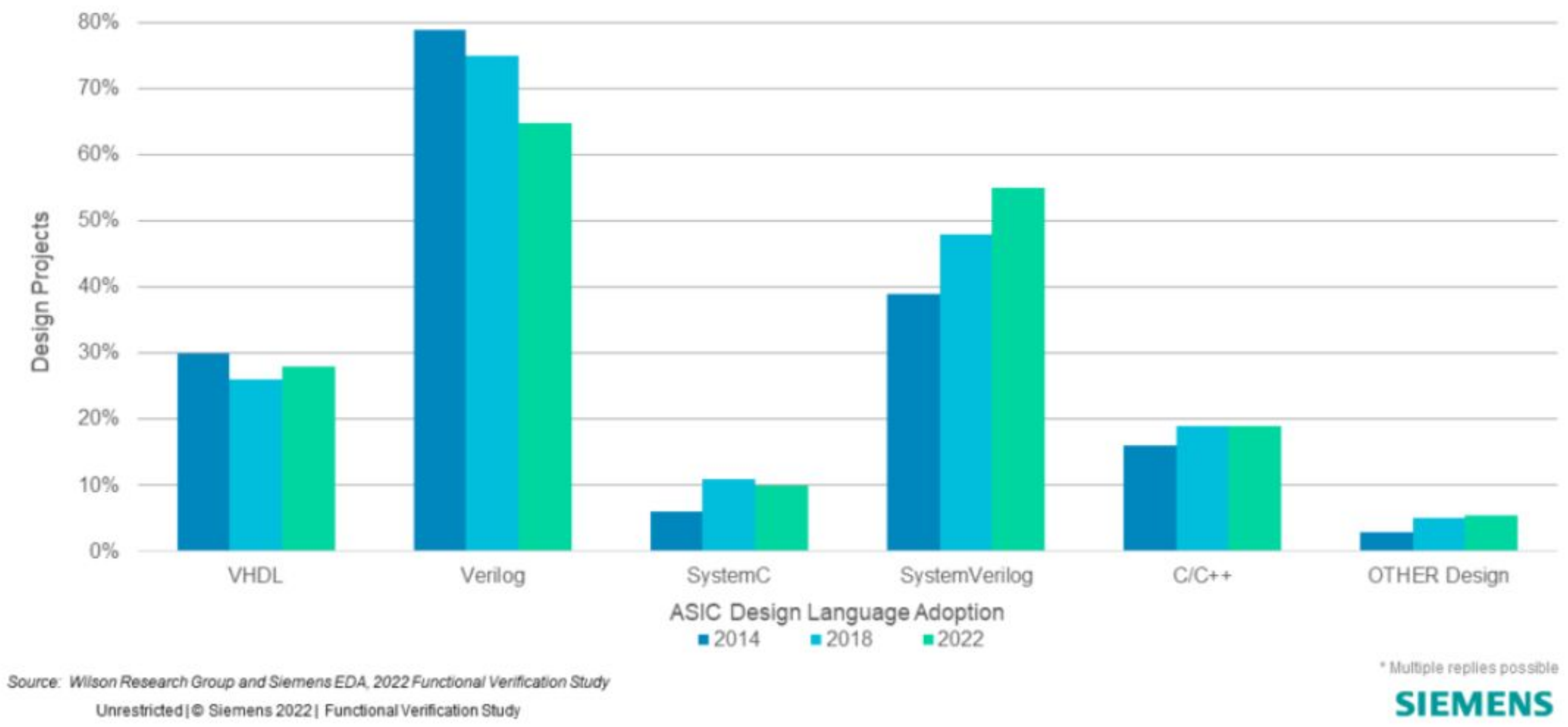
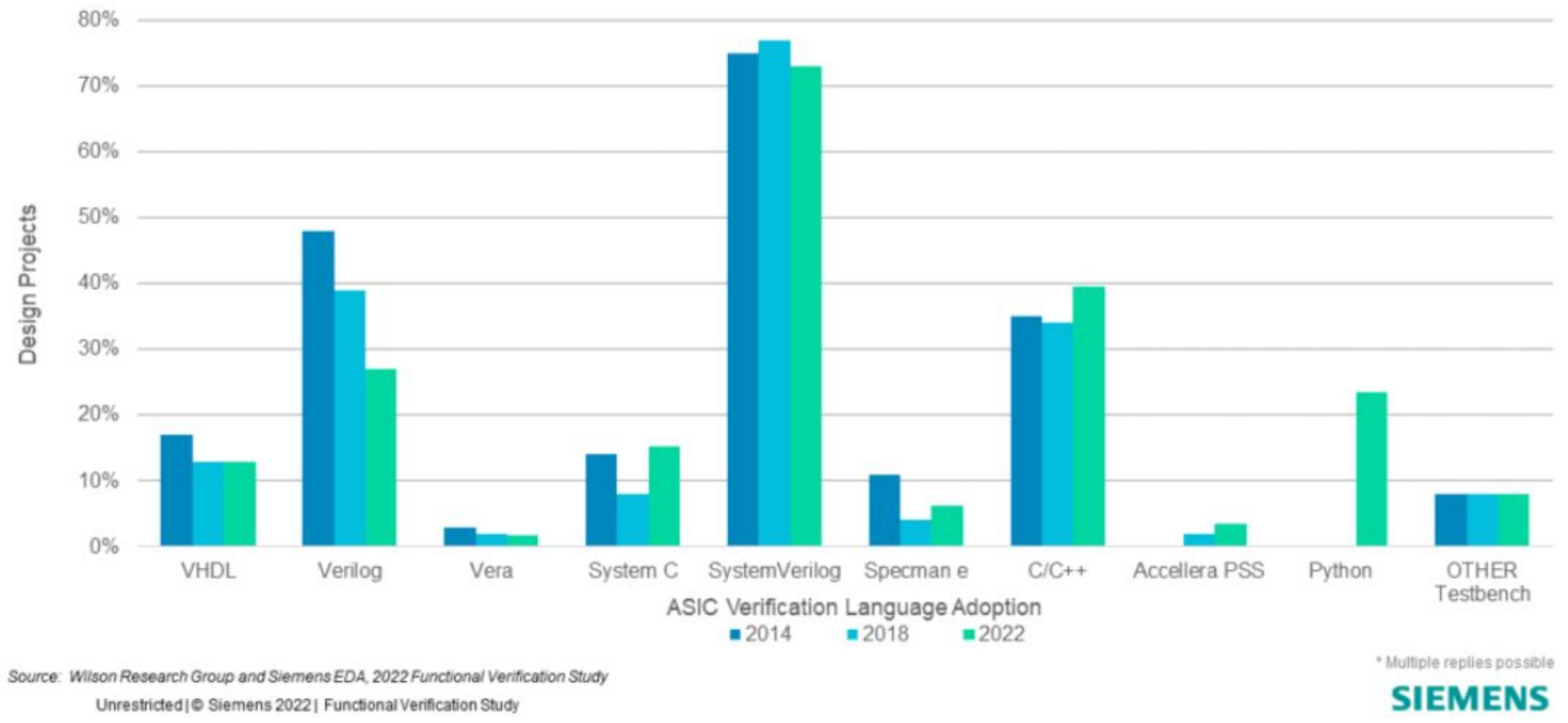


Figure 10-1. IC/ASIC Languages Used for RTL Design

Source: Prologue: The 2022 Wilson Research Group Functional Verification Study



Source: Prologue: The 2022 Wilson Research Group Functional Verification Study

Break!

How to run example SV code or files?

- Follow the SystemVerilog Tutorials (Youtube)

- Go to EDA Playground website

Introduction

- What is *Hardware Description Language (HDL)*?
- Why we go for SystemVerilog?
- What is verification?
- How to verify a design?
- Why SystemVerilog for verification?
- **Design features**

Verilog HDL

Dimension	Description
Origin	Developed in 1984 by Gateway Design Automation.
Primary Use	Digital circuit design, especially in industry.
Syntax Style	C-like syntax, concise and procedural.
Type System	Weakly typed, less strict on data type declarations.
Abstraction Levels	Gate level and RTL (Register Transfer Level).
Data Types	Limited to basic types (integer, real, bit vectors).
Concurrency	Implicit concurrency based on signal assignments.
Modeling Features	Basic constructs, lacks some high-level features.
Verification Support	Limited verification features.
Standardization	IEEE 1364 (2005, latest standard).
Tool Support	Supported by most synthesis and simulation tools.

SystemVerilog

Dimension	Description
Origin	Developed in 2005 as an extension of Verilog by Accellera.
Primary Use	Used for both design and verification, especially in industry.
Syntax Style	Extends Verilog syntax with additional features for modeling.
Type System	Strongly typed, with enhanced data types over Verilog.
Abstraction Levels	RTL to higher-level abstractions, supports testbench modeling.
Data Types	Advanced data types (enums, structs, unions, classes).
Concurrency	Adds process control like classes, interfaces, and assertions.
Modeling Features	Enhanced modeling features (interfaces, classes, constraints).
Verification Support	Comprehensive verification support (Assertions, UVM).
Standardization	IEEE 1800 (2017, latest standard).
Tool Support	Widely supported, especially for verification environments.

As a summary

What is SystemVerilog?

- A hardware description and verification language.
- Extends Verilog for design and testbenches.
- Supports object-oriented programming and assertions.

Why use SystemVerilog for Digital Circuits?

- Enables efficient simulation and verification of digital designs.
- Powerful data types and constructs for modeling real-world behavior.

Design Features

The feature-set of SystemVerilog can be divided into two distinct roles:

- SystemVerilog for register-transfer level (RTL) design is an extension of Verilog-2005; all features of that language are available in SystemVerilog. Therefore, Verilog is a subset of SystemVerilog.

- SystemVerilog for verification uses extensive object-oriented programming techniques and is more closely related to Java than Verilog.

Design Features - Data lifetime

There are two types of data lifetime specified in SystemVerilog: static and automatic (local variable).

Automatic variables are created the moment program execution comes to the scope of the variable.

Static variables are created at the start of the program's execution and keep the same value during the entire program's lifespan, unless assigned a new value during execution.

Design Features - Data lifetime

Any variable that is declared inside a task or function without specifying type will be considered automatic.

To specify that a variable is static place the "`static`" keyword in the declaration before the type, e.g., "`static int x;`".

SystemVerilog Data Types

Built-in Data Types:

- **reg:** Stores binary values.
- **wire:** Represents connections between components.
- **integer:** 32-bit signed integer.
- **real:** Floating-point values.

User-Defined Types:

- **typedef:** Create custom data types.
- **enum:** Represents a set of named values (useful for state machines).

SystemVerilog Data Types

Built-in Data Types:

`reg` (Register):

- **Used in Verilog**: Introduced in Verilog for modeling storage elements like flip-flops and registers.
- **Behavior**: Holds a value over time (retains its value until changed).
- **Usage**: **Primarily for sequential logic** where values need to be stored.

SystemVerilog Data Types

Built-in Data Types:

Example code for SystemVerilog:

```
reg [3:0] data; // 4-bit register

always @(posedge clk) begin

  data <= next_data; // Stores next_data on clock edge

end
```

SystemVerilog Data Types

Built-in Data Types:

Example code for SystemVerilog:

```
module reg_example(input logic clk, input logic reset, input logic [3:0] next_data, output reg [3:0] data);

  always @(posedge clk or posedge reset) begin

    if (reset)

      data <= 4'b0000;   // Reset condition

    else

      data <= next_data; // Store next_data on clock edge

  end

endmodule
```

SystemVerilog Data Types

Built-in Data Types:

`logic` (SystemVerilog):

- **Introduced in SystemVerilog**: A more versatile data type that replaces `reg` and `wire`.
- **Behavior**: Holds a value and is used for both combinational and sequential logic.
- **Usage**: Recommended for **both combinational and sequential logic**. Supports 4-state logic (`0`, `1`, `x`, `z`).

SystemVerilog Data Types

Built-in Data Types:

Example code for SystemVerilog:

```
logic [3:0] data; // 4-bit logic variable
always @(posedge clk) begin
  data <= next_data; // Stores next_data on clock edge
end
```

SystemVerilog Data Types

Built-in Data Types:

Example code for SystemVerilog:

```
module logic_example(input logic clk, input logic reset, input logic [3:0]
next_data, output logic [3:0] data);

  always @(posedge clk or posedge reset) begin

    if (reset)

      data <= 4'b0000;   // Reset condition

    else

      data <= next_data; // Store next_data on clock edge

  end

endmodule
```

SystemVerilog Data Types

Built-in Data Types:

Key Differences:

- `reg` is specific to storing values in **sequential** logic.
- `logic` can be used for both **combinational** and **sequential** logic.
- `logic` is the preferred type in SystemVerilog as it is more flexible and easier to use.

When to Use:

- **`reg`**: For legacy Verilog code or when modeling storage elements in older designs.
- **`logic`**: For modern SystemVerilog designs and more flexible usage in both sequential and combinational circuits.

SystemVerilog Data Types

Types of Registers & Variables

- **reg vs wire:**
  - **reg** stores data and can hold values over time (like a memory cell).
  - **wire** is used to connect different components (like a bus).
- **Example:**
  - reg [7:0] data; // 8-bit register
  - wire enable; // Connection line

SystemVerilog Data Types

Daily Life Example: Traffic Light System

- **Problem:**
  - Model a traffic light using basic digital circuits.
  - States: Green, Yellow, Red.

SystemVerilog Data Types

Enums

Enumerated data types (`enums`) allow numeric quantities to be assigned meaningful names. Variables declared to be of enumerated type cannot be assigned to variables of a different enumerated type without casting. This is not true of parameters, which were the preferred implementation technique for enumerated quantities in Verilog-2005

SystemVerilog Data Types

Enum example:

```
typedef enum logic [2:0] {

  RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW

} color_t;



color_t   my_color = GREEN;



initial $display("The color is %s", my_color.name());
```

SystemVerilog Data Types

Enums

```
typedef enum logic [2:0] {...}color_t;:
```

- `typedef`: This keyword is used to create a new **user-defined data type**.

- `enum`: An **enumerated type** defines a set of named values. In this case, the set of values will represent different colors.

SystemVerilog Data Types

Enums

```
typedef enum logic [2:0] {...}color_t;:
```

- `logic [2:0]`: This defines the **bit width** of the enum values. The `[2:0]` means that each value in the enum will be represented by a 3-bit binary value. So, each color will be assigned a 3-bit number starting from `0`.

- The values will be assigned automatically, starting from `RED = 3'b000`, `GREEN = 3'b001`, `BLUE = 3'b010`, and so on.

SystemVerilog Data Types

Enums

```
typedef enum logic [2:0] {

  RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW} color_t;
```

The Values of Enum:

```
RED = 3'b000

GREEN = 3'b001

BLUE = 3'b010

CYAN = 3'b011

MAGENTA = 3'b100

YELLOW = 3'b101
```

SystemVerilog Data Types

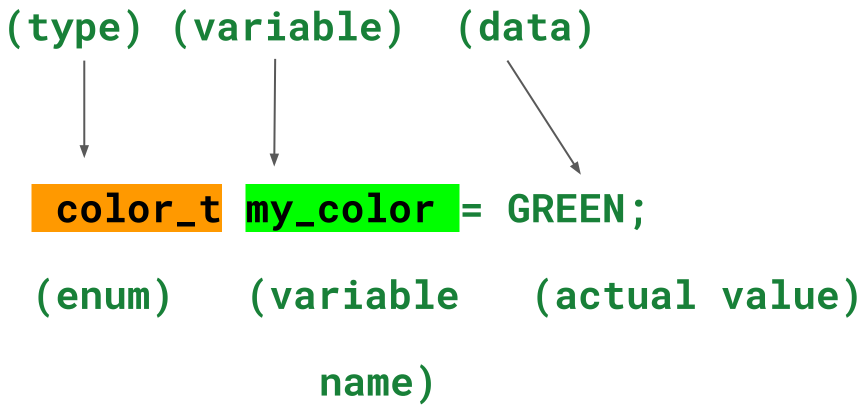
Enums

```
typedef enum logic [2:0] {...}color_t;
```

`color_t`: This is the **name of the new type** (an enum type) that we just defined. It allows us to create variables of this type, which will hold one of the color values.

SystemVerilog Data Types - **Enums**

```
typedef enum logic [2:0] {...}color_t;
```



```
color_t my_color = GREEN;
```

This creates a variable `my_color` of type `color_t`, which is the **enum type** we just defined.

- It initializes `my_color` with the value `GREEN`. Since `GREEN` is the second item in the enum, it will hold the value `3'b001`.

SystemVerilog Data Types

Enums

```
initial $display("The color is %s", my_color.name());
```

The `initial` keyword in SystemVerilog is used to define **initial blocks** in the design. An **initial block** executes **only once at the beginning of the simulation**, at **time 0**, before any other operations in the simulation.

This is useful for setting initial values, performing setup tasks, or displaying information when the simulation starts.

SystemVerilog Data Types

Enums

```
initial $display("The color is %s", my_color.name());
```

You **can** use the `initial` keyword **with a block**

```
initial begin
    // use for comments
    // Statements to execute at time 0
end
```

or you can use **without a block** when you are executing a single statement, such as `$display`. In SystemVerilog, if you only have one statement to execute, you don't need to use the `begin` and `end` keywords to define a block.

SystemVerilog Data Types - **Enums**

```
typedef enum logic [2:0] {RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW}color_t;

color_t   my_color = GREEN;

initial $display("The color is %s", my_color.name());
```

```
    typedef enum logic [2:0] {RED, GREEN, BLUE, CYAN, MAGENTA,
YELLOW}color_t;

    color_t my_color;

    initial begin
        my_color = GREEN;  // assign GREEN
        $display("The color is %s", my_color.name());
    end
```

SystemVerilog Data Types

Enums

```
initial $display("The color is %s", my_color.name());
```

- `$display` is a system task in Verilog/SystemVerilog used to print output to the console.
- `my_color.name()`: This is a **SystemVerilog feature** that allows you to access the **name** of the enum value stored in `my_color`. In this case, it will return the string `"GREEN"`, which corresponds to the value of `my_color` (which is `3'b001`).
- The `%s` format specifier is used to print the **string representation** of the enum value.

SystemVerilog Data Types - **Enums - same code**

```
typedef enum logic [2:0] {RED, GREEN, BLUE, CYAN, MAGENTA, YELLOW}color_t;

color_t   my_color = GREEN;

initial $display("The color is %s", my_color.name());
```

```
    typedef enum logic [2:0] {RED, GREEN, BLUE, CYAN, MAGENTA,
YELLOW}color_t;

    color_t my_color;

    initial begin
        my_color = GREEN;  // assign GREEN
        $display("The color is %s", my_color.name());
    end
```

SystemVerilog Data Types

Daily Life Example: Traffic Light System (we were here..)

- **Problem:**
  - Model a traffic light using basic digital circuits.
  - States: Green, Yellow, Red.

SystemVerilog Data Types

Daily Life Example: Traffic Light System

- **Problem:**
  - Model a traffic light using basic digital circuits.
  - States: Green, Yellow, Red.

- **SystemVerilog Code Structure:**
  - Define states using enum type.
  - Use reg to hold the current state and wire for transitions.

SystemVerilog Data Types

Daily Life Example: Traffic Light System

- **Problem:** Model a traffic light using basic digital circuits.
  - States: Green, Yellow, Red.
- **SystemVerilog Code Structure:** Define states using enum type.
  - Use reg to hold the current state and wire for transitions.

Daily Life Analogy:

- A traffic light changes **states** based on time.
- A car (signal) moves through intersections based on these states.

SystemVerilog Data Types

Basic Digital Circuits – Example

- **AND Gate:**
  - **Truth Table:** Output is 1 only when both inputs are 1.

SystemVerilog Code Example:

```
module AND_Gate(input wire A, input wire B, output wire C);
  assign C = A & B;
endmodule
```

Daily Life Analogy: Imagine a switch that requires two buttons to be pressed at the same time for the light to turn on.

SystemVerilog Data Types

Basic Digital Circuits – Example

`assign` keyword:

The `assign` keyword in SystemVerilog is used to define continuous assignments. Continuous assignments are constantly evaluated, meaning the value of `C` will always reflect the current values of `A` and `B` **without the need for clocking or triggering events.**

SystemVerilog Data Types

Basic Digital Circuits – Example

`assign` keyword:

Can I use assign keyword in always block?

SystemVerilog Data Types

Basic Digital Circuits – Example

`assign` keyword:

You cannot use the `assign` keyword inside an `always` block in SystemVerilog or Verilog. The `assign` keyword is specifically used for continuous assignments, which are evaluated and updated automatically whenever the values of the right-hand side expressions change. Continuous assignments are meant for combinational logic and are typically written outside of procedural blocks, like `always` blocks.

SystemVerilog Data Types

Basic Digital Circuits – Example

```
module example_assign (input logic A, input logic B,output logic C);
    assign C = A & B; // Continuous assignment
endmodule
```

```
module example_always (input logic A,input logic B,input logic clk,output logic C);
    always_ff @(posedge clk) begin
        C <= A & B; // Non-blocking assignment in sequential logic
    end
endmodule
```

SystemVerilog Data Types

Basic Digital Circuits – OR Gate

- **OR Gate:**
    - **Truth Table:** Output is 1 if at least one input is 1.

SystemVerilog Code Example:

```
module OR_Gate(input wire A, input wire B, output wire C);
  assign C = A | B;
endmodule
```

Daily Life Analogy: Think of a light that turns on if either of two motion detectors detects movement.

End of Week 4!