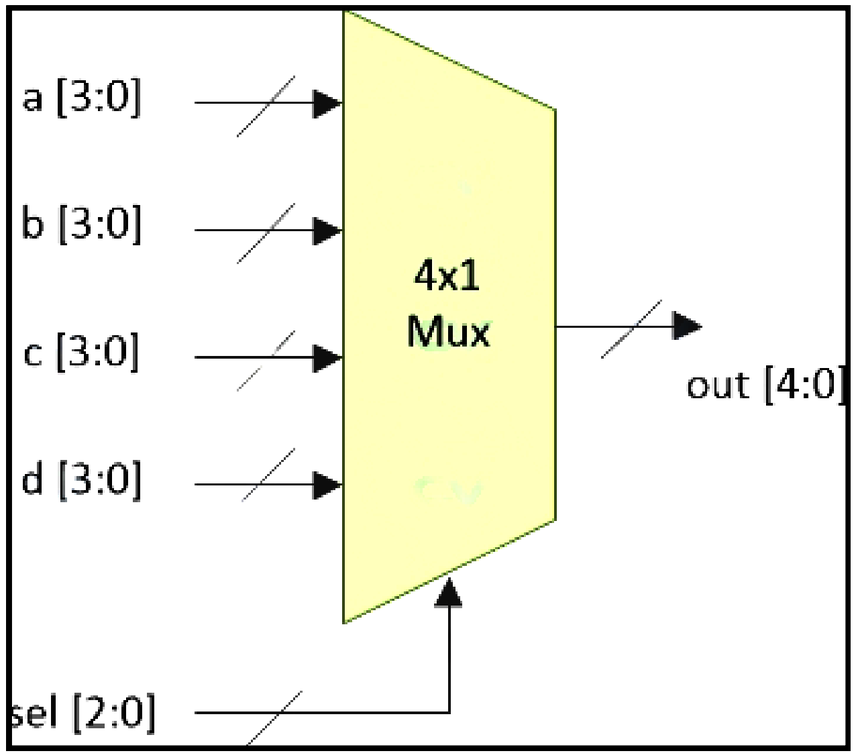
INTRODUCTION TO HDL

by Salvatore Ruocco
IoT Security course
a.y. 2024/2025

LIST OF CONTENTS

LESSON GOAL

Write a 4x1 multiplexer in Verilog




```
module mux_4x1_case (
    input [3:0] a,
    input [3:0] b,
    input [3:0] c,
    input [3:0] d,
    input [1:0] sel,
    output [3:0] out
);

    always @ (a or b or c or d or sel) begin
        case (sel)
            2'b00 : out <= a;
            2'b01 : out <= b;
            2'b10 : out <= c;
            2'b11 : out <= d;
        endcase
    end
endmodule
```

ABOUT ME 👋

Hi, I'm **Salvatore Ruocco**, I'm just a student in the master of cybersecurity course like you.

ABOUT ME 👋

Hi, I'm **Salvatore Ruocco**, I'm just a student in the master of cybersecurity course like you.

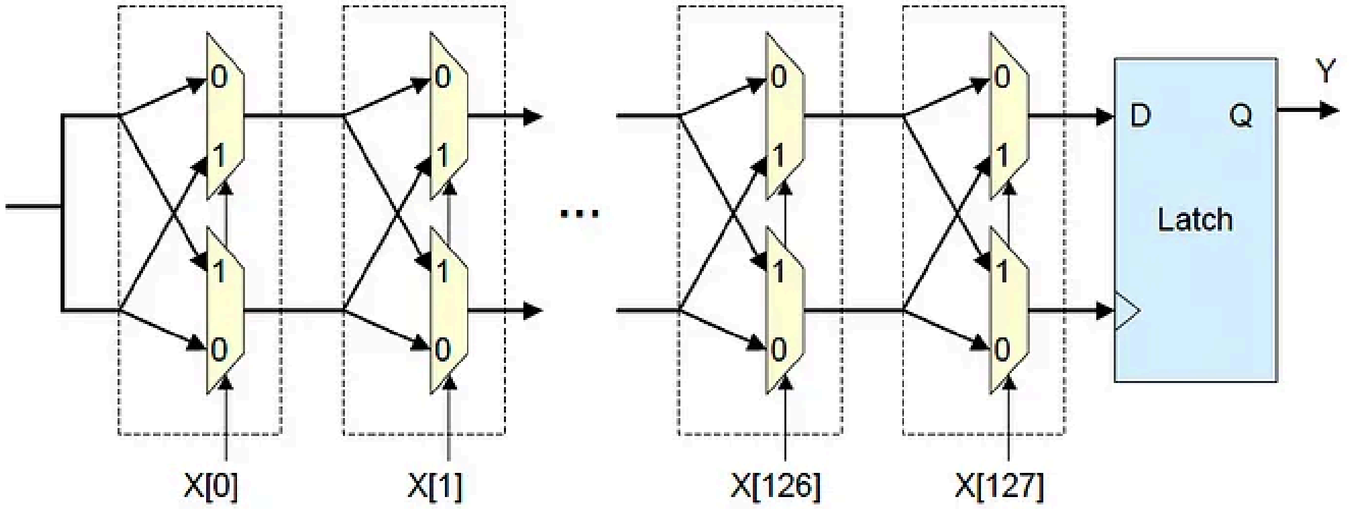
In less than a month I will be discussing my bachelor's thesis on

ARBITER PUF IMPLEMENTATION ON FPGA

ARBITER PUF

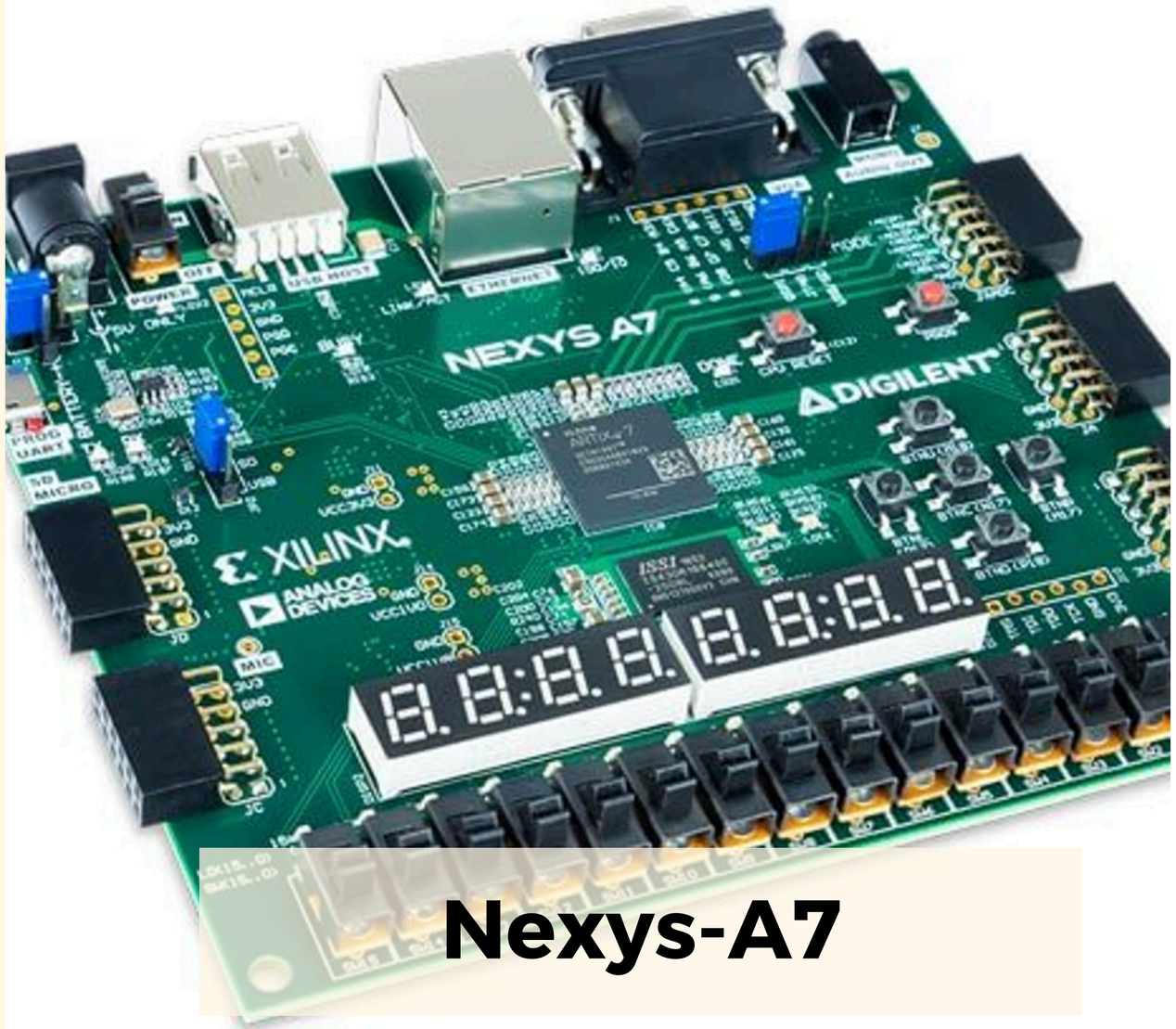
The concept of Physical Unclonable Functions (PUF) is vital in the fields of hardware security and cryptography due to their ability to generate unique, device-specific identifiers that are nearly impossible to clone or replicate.

Unlike traditionalmethods that rely on stored keys or software-based algorithms, PUFs **exploits the inherent physical variations present in each hardware component variations introduced naturally during the manufacturing process.**



Field
Programmable
Gate
Array

An FPGA is a configurable integrated circuit.
With **Field Programmable** we mean that the user can reprogram the chip after it has been used, so we can say that we are programming ‘pieces of silicon’.



Nexys-A7

Why learn FPGA in IoT Security

01

02

03

Why learn FPGA in IoT Security

01

Velocity

FPGAs can process and relay strange behaviors with low latency.

02

03

Why learn FPGA in IoT Security

01

Velocity

FPGAs can process and relay strange behaviors with low latency.

02

Cryptography

Some cryptographic algorithms such as AES, RSA and others can be implemented directly in hardware.

03

Why learn FPGA in IoT Security

01

Velocity

FPGAs can process and relay strange behaviors with low latency.

02

Cryptography

Some cryptographic algorithms such as AES, RSA and others can be implemented directly in hardware.

03

Specific uses

FPGAs allow hardware circuits to be created for specific uses, such as firewalls, key management or authentication.

HOW PROGRAM IT?

HDL

Hardware
Description
Language

VHDL

It is the primary HDL used for digital circuit design, it was developed in the 1980s by the U.S. Department of Defense.

always them...

HOW PROGRAM IT?

HDL

Hardware
Description
Language

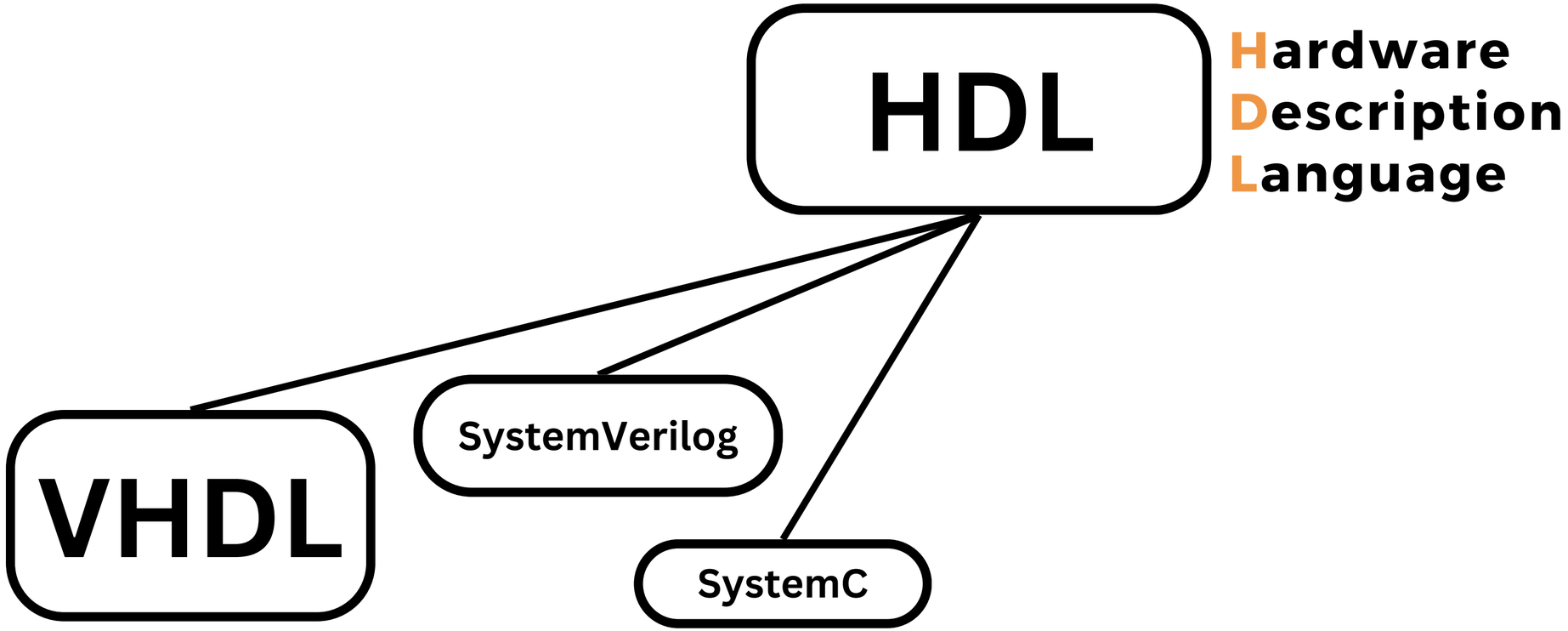
VHDL

SystemVerilog

It is the primary HDL used for digital circuit design, it was developed in the 1980s by the U.S. Department of Defense.

always them...

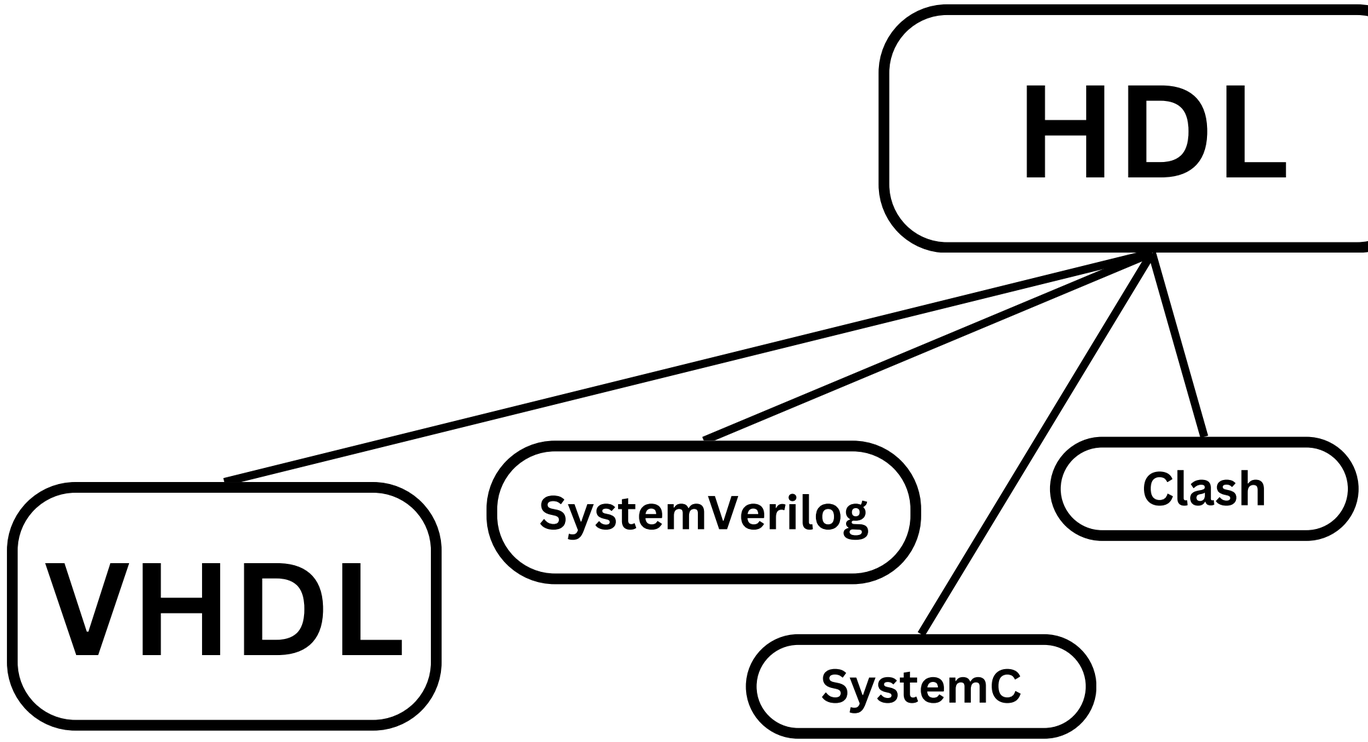
HOW PROGRAM IT?



It is the primary HDL used for digital circuit design, it was developed in the 1980s by the U.S. Department of Defense.

always them...

HOW PROGRAM IT?

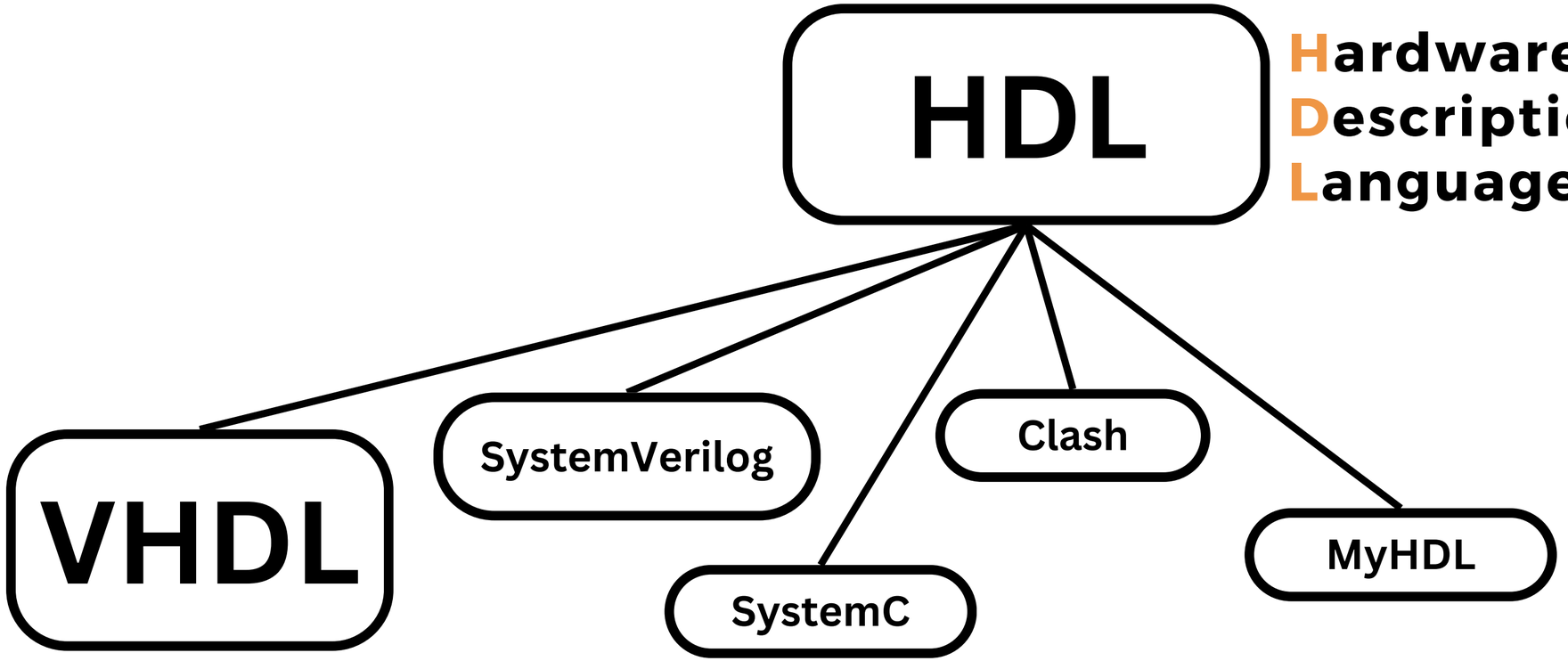


Hardware
Description
Language

It is the primary HDL used for digital circuit design, it was developed in the 1980s by the U.S. Department of Defense.

always them...

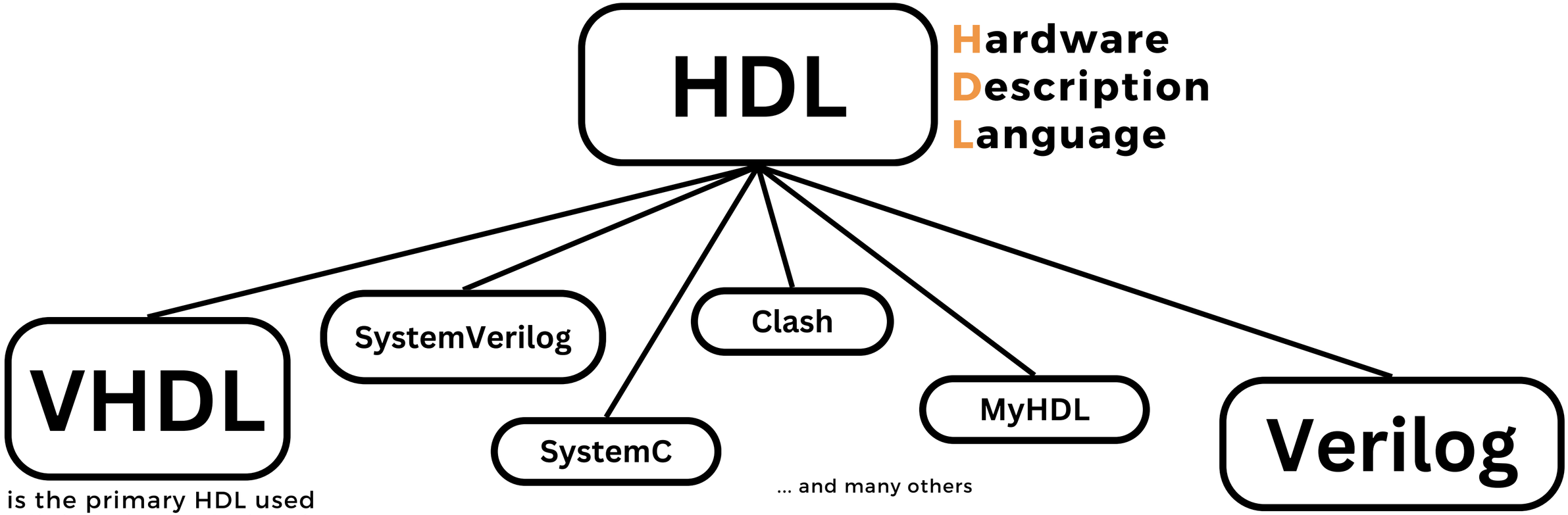
HOW PROGRAM IT?



It is the primary HDL used for digital circuit design, it was developed in the 1980s by the U.S. Department of Defense.

always them...

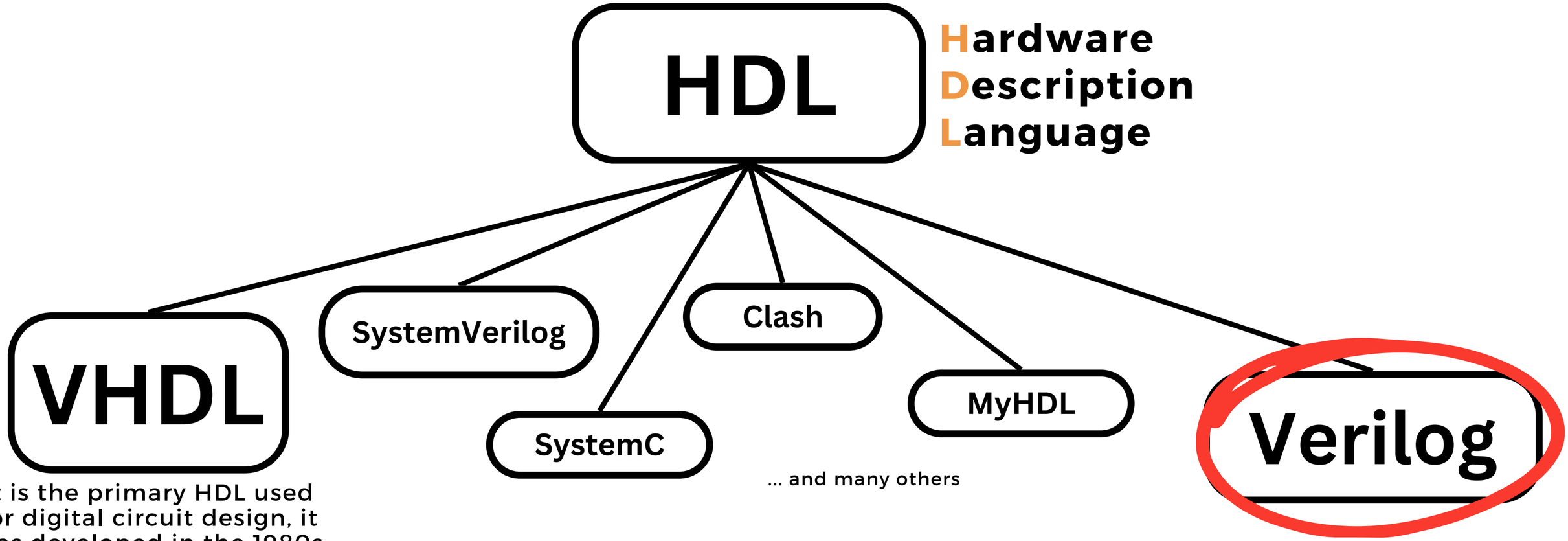
HOW PROGRAM IT?



It is the primary HDL used for digital circuit design, it was developed in the 1980s by the U.S. Department of Defense.

always them...

HOW PROGRAM IT?



It is the primary HDL used for digital circuit design, it was developed in the 1980s by the U.S. Department of Defense.

always them...

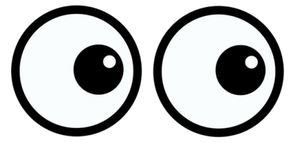
... and many others

I choose it for my thesis project

WHY VERILOG?

- High-level abstract descriptions to write hardware circuits
- Simple syntax that is similar to C

But HDLs have **many differences** with the high-level languages (C, C++, Java, ...) we often use.



HDL VS HIGH-LEVEL LANGUAGES

Main differences between languages used to describe hardware circuits and the most common high-level languages

01 Sequence of instructions

02 No loops

HDL VS HIGH-LEVEL LANGUAGES

Main differences between languages used to describe hardware circuits and the most common high-level languages

01 Sequence of instructions

In the languages we all know, instructions are executed one line at a time, from top to bottom, in procedural way.
FPGAs don't work in the same way: in HDL we describe logic circuits, such as logic gates (AND, OR, NOT), multiplexers, Flip-Flops and circuits to perform arithmetic operation (half adder, full adder).
The programming paradigm is not sequential but **concurrent**: so, multiple parts of the circuit can operate simultaneously.

HDL VS HIGH-LEVEL LANGUAGES

Main differences between languages used to describe hardware circuits and the most common high-level languages

02 No loops

Yes, there are **for** and **while** but not as we understand them in a time sequence sense, but they are used **to replicate hardware.**

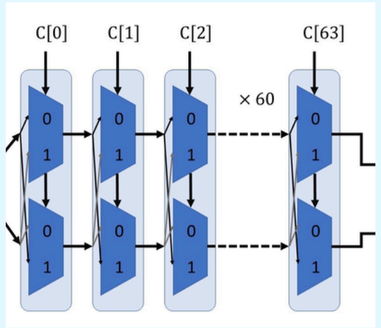
HDL VS HIGH-LEVEL LANGUAGES

Main differences between languages used to describe hardware circuits and the most common high-level languages

02 No loops

Yes, there are **for** and **while** but not as we understand them in a time sequence sense, but they are used **to replicate hardware.**

I used this approach to generate the **128 blocks of multiplexers** for the arbiter PUF.



C

```c
int sum = 0;
for (int i = 0; i < 3; i++) {
    sum += array[i];
}
```

This is simple script in C for summing the elements of an array

C

```c
int sum = 0;
for (int i = 0; i < 3; i++) {
    sum += array[i];
}
```

This is simple script in C for summing the elements of an array

Verilog

This loop doesn't sum values one at a time, but describes a **combinatorial circuit that sums all array elements simultaneously**, using parallel logic; thus each iteration of the loop constitutes a part of the circuit that contributes to the sum.

```
sum = 0;
for(i = 0; i < 3; i = i + 1) begin
    sum <= sum + array[i];
end;
```

Procedural loop in Verilog

This is an example of how a procedural loop.

```
module example_module;
    reg [7:0] array [0:3];
    integer i, sum;

    initial begin
        for (i = 0; i < 3; i = i + 1) begin
            sum = sum + array[i];
        end
    end
endmodule
```

Procedural loop in Verilog

This is an example of how a procedural loop.

```
module example_module;
    reg [7:0] array [0:3];
    integer i, sum;

    initial begin
        for (i = 0; i < 3; i = i + 1) begin
            sum = sum + array[i];
        end
    end
endmodule
```

But

This code is not synthesizable, it cannot be run in hardware, but only on a software simulator since there is the **initial block** used mainly to ‘mock up’ variables for use in testbench.

Always block

Unlike a loop, an always block doesn`t repeat in time in a sequential manner, but **runs in parallel with the rest of the circuit** every time an event specified among the block`s input parameters occurs.

```
always @(posedge clk) begin // Executed at the positive edge of clk 0->1
    y = a & b;
end
```

Module template

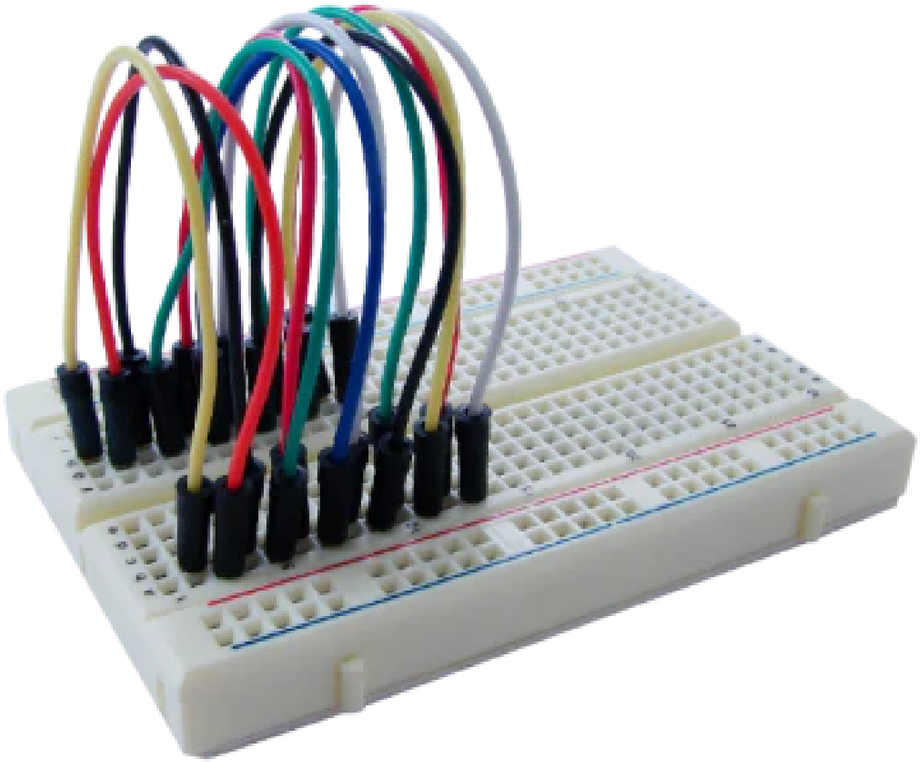
```
module [module_name] (
    [list_of_input_ports]
    [list_of_output_ports]
);

    [declaration_of_other_variables]

    [other_module_instantiations_if_required]

    [behavioral_code_for_this_module]
endmodule
```

Data types in Verilog

Nets and variables are the two main groups of data type

Nets

Nets are used to **connect between hardware entities** like logic gates and hence do not store any value on its own.
There are different types of nets each with different characteristics,
but the most popular and widely used net in Verilog is of type **wire**.
The **wire** is similar to the electrical wire that is used to connect two components on a breadboard.



Data types in Verilog

Nets and variables are the two main groups of data type

Nets

Nets are used to **connect between hardware entities** like logic gates and hence do not store any value on its own.
There are different types of nets each with different characteristics,
but the most popular and widely used net in Verilog is of type **wire**.
The **wire** is similar to the electrical wire that is used to connect two components on a breadboard.

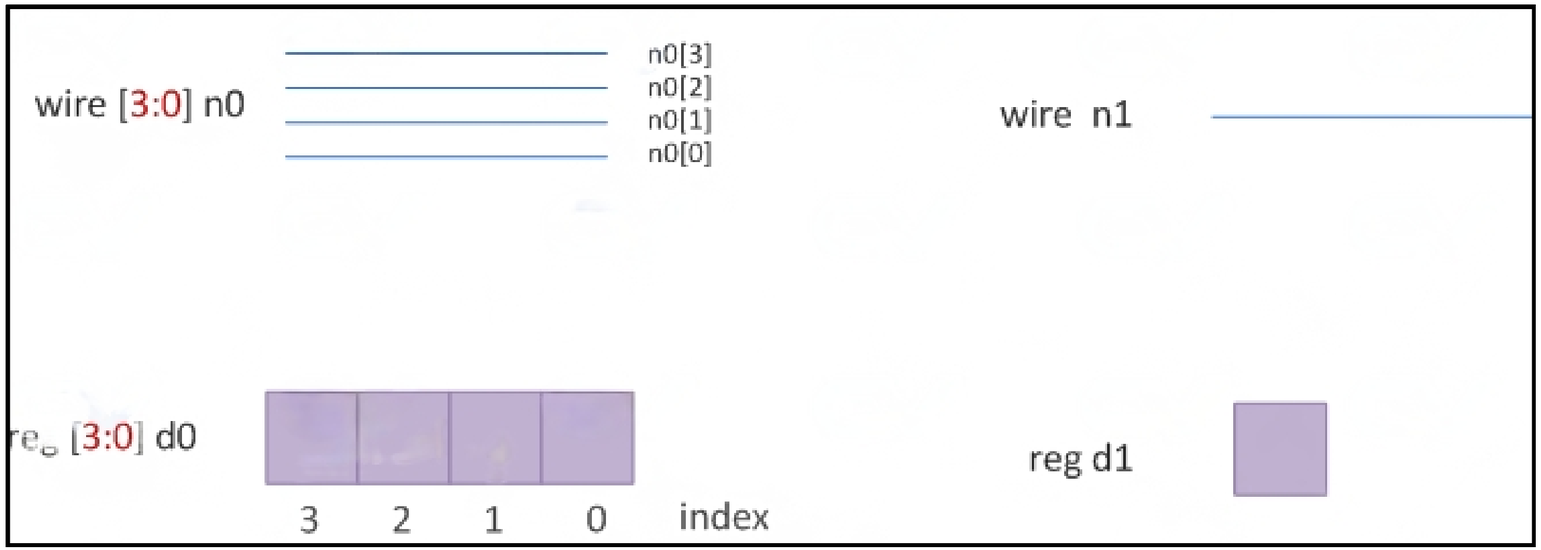
Variables

A variable is an abstraction of a **data storage element** and can hold values.
reg in Verilog is a variables and it is usually represented in hardware by a flip-flop.

Scalar and vector

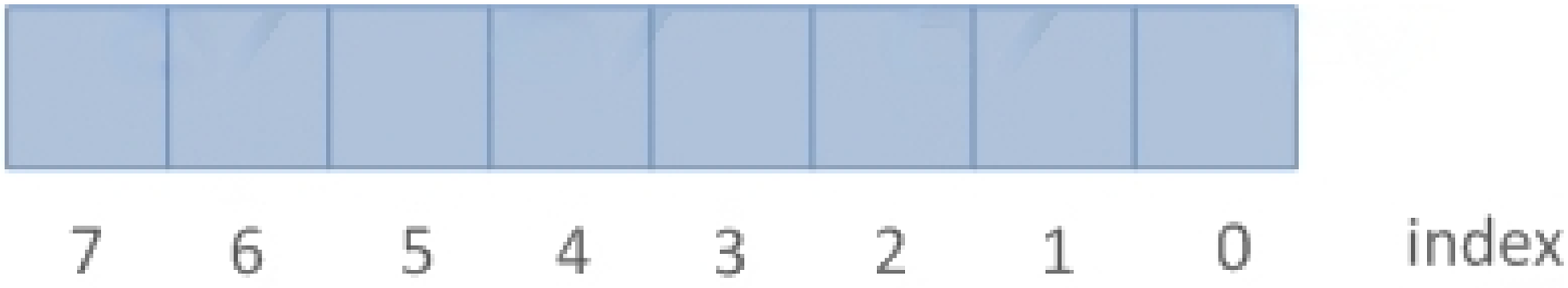
A **wire** or **reg** declaration without a range specification
is considered 1-bit wide and is a **scalar**.
If a range is specified, then the net or reg becomes a multi-bit entity known as a **vector**.

```
wire        single_bit;     // scalar
wire [7:0]  wire_8_bit;     // 8-bit vector net
reg         parity;         // single bit scalar variable
reg  [31:0] addr;           // 32 bit vector variable
```



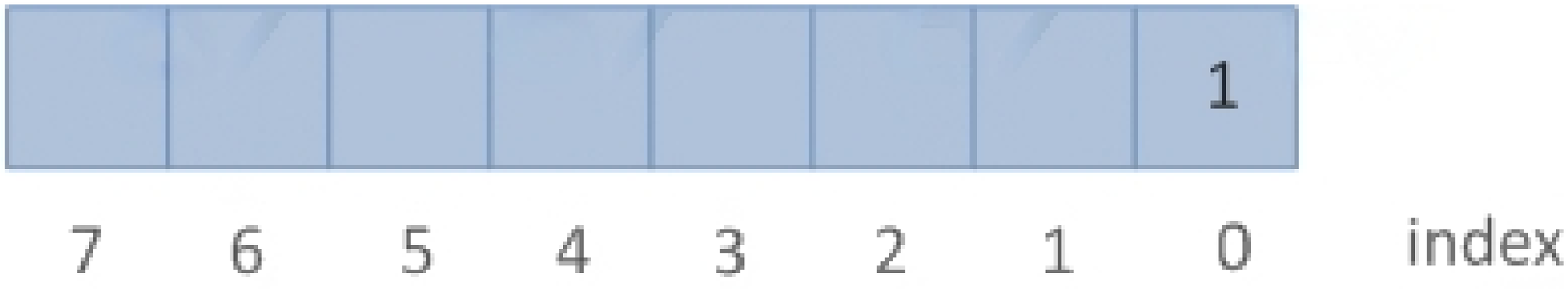
Bit selection

```
1  reg [7:0]       addr;          // 8-bit reg variable [7, 6, 5, 4, 3, 2, 1, 0]
```



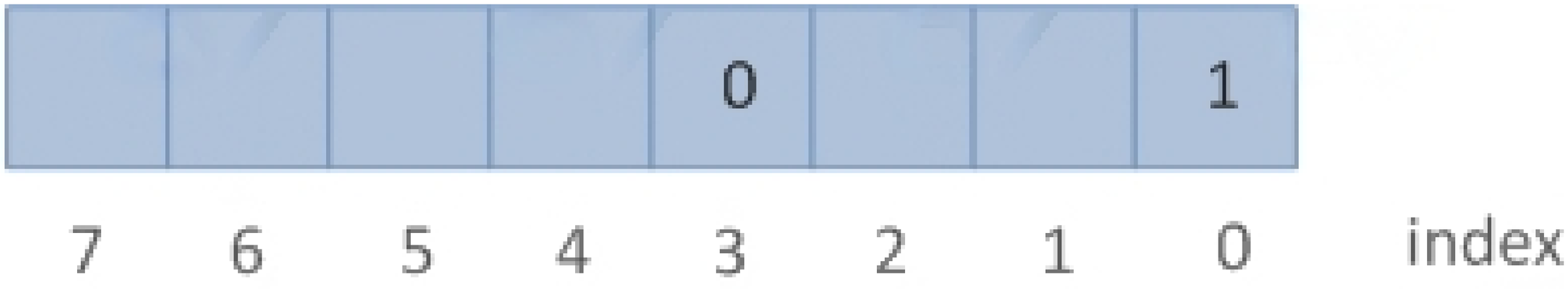
Bit selection

```
1  reg [7:0]       addr;          // 8-bit reg variable [7, 6, 5, 4, 3, 2, 1, 0]
2
3  addr [0] = 1;                  // assign 1 to bit 0 of addr
```



Bit selection

```
1  reg [7:0]       addr;          // 8-bit reg variable [7, 6, 5, 4, 3, 2, 1, 0]
2
3  addr [0] = 1;                  // assign 1 to bit 0 of addr
4  addr [3] = 0;                  // assign 0 to bit 3 of addr
```



Ports

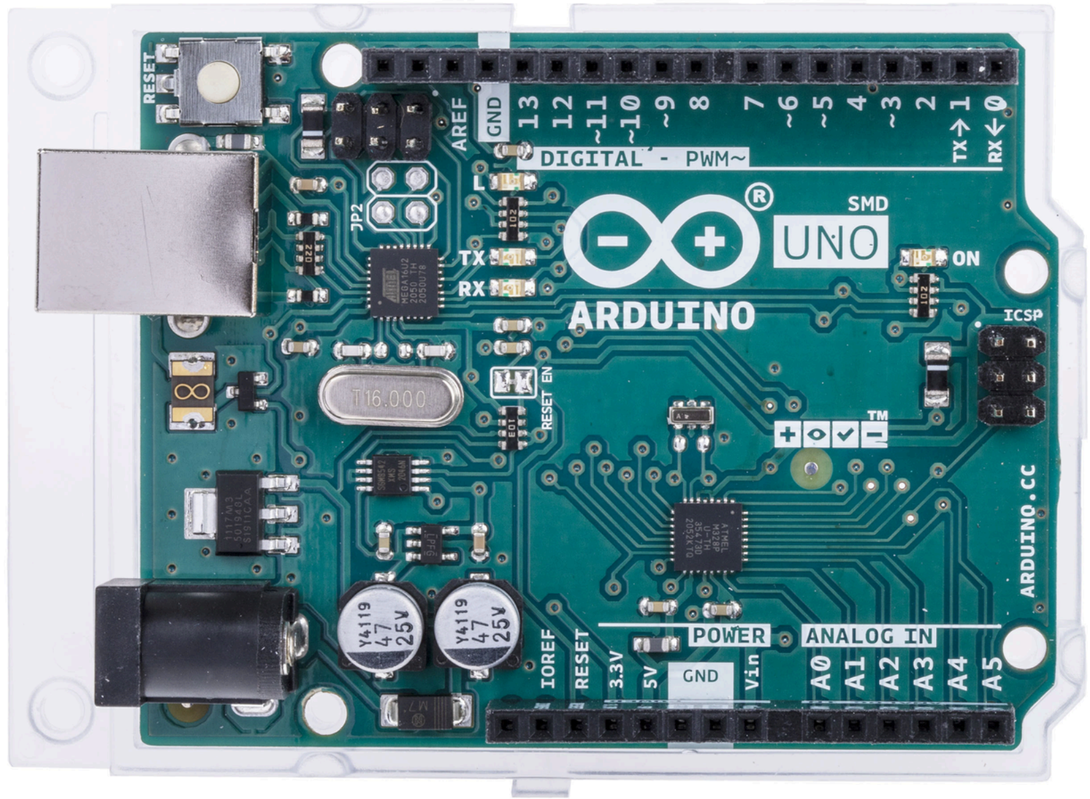
Ports are a set of signals that act as **inputs** and **outputs** to a particular module and are **the primary way of communicating with device**.

Ports

Ports are a set of signals that act as **inputs** and **outputs** to a particular module and are **the primary way of communicating with device**.

Think for example at

Arduino

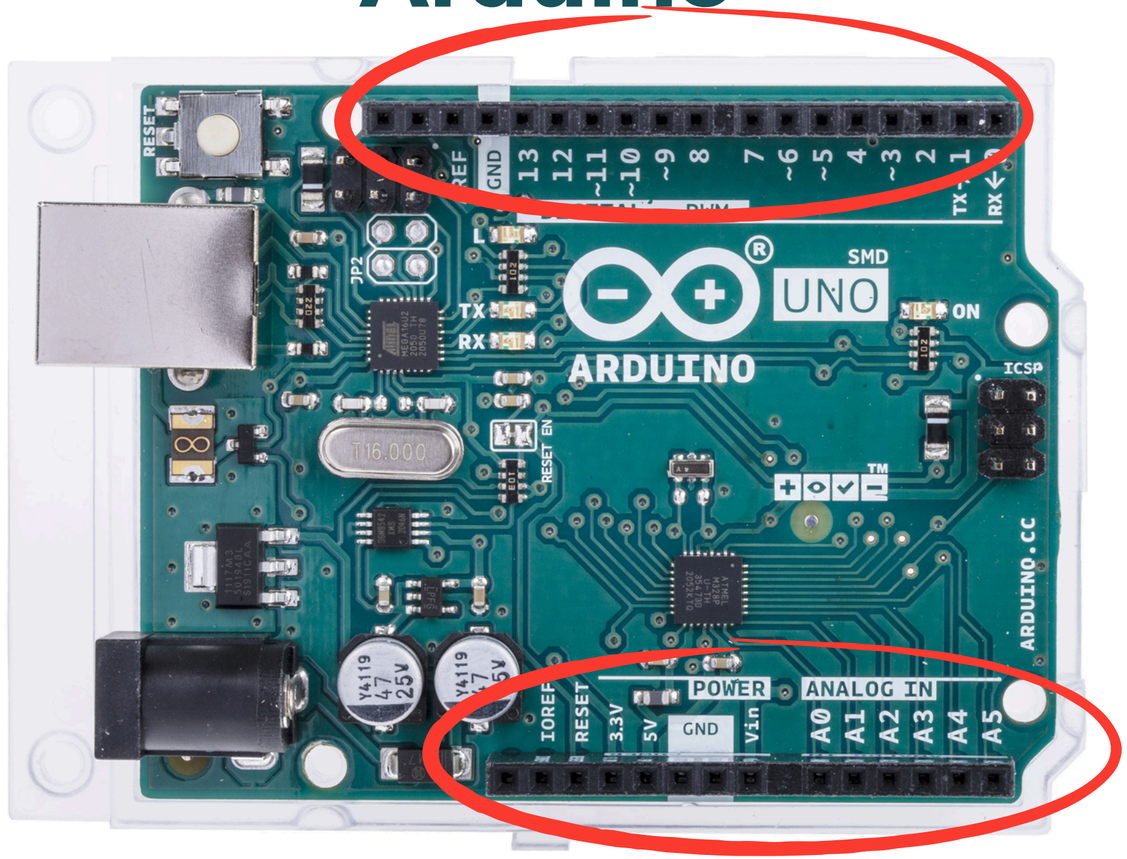


Ports

Ports are a set of signals that act as **inputs** and **outputs** to a particular module and are **the primary way of communicating with device**.

Think for example at

Arduino



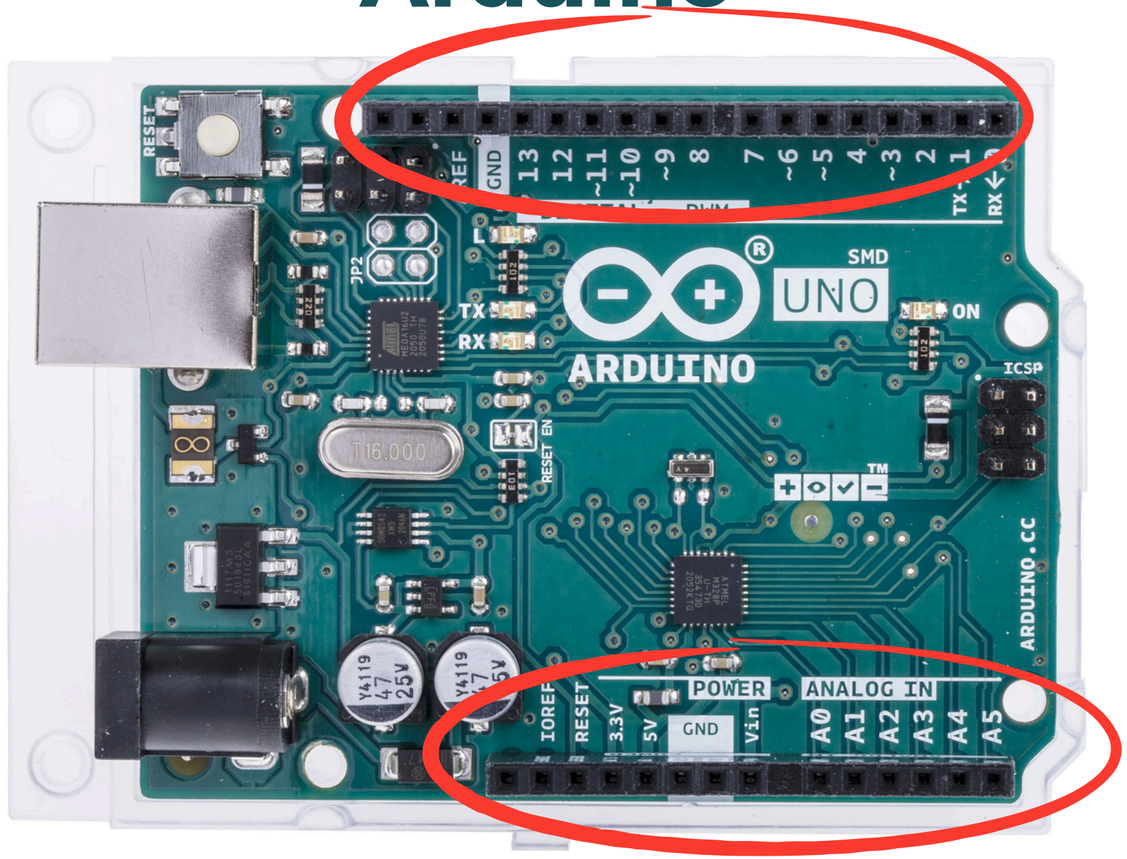
Ports are like Arduino pins and they are used by the design to send and receive signals from the outside world.

Ports

Ports are a set of signals that act as **inputs** and **outputs** to a particular module and are **the primary way of communicating with device**.

Think for example at

Arduino



Ports are like Arduino pins and they are used by the design to send and receive signals from the outside world.

There are **three types** of port:

- input
- output
- inout

Number format

To write a number in verilog we must follow the following syntax:

[size]'[base_format][number]

Number format

To write a number in verilog we must follow the following syntax:

[size]'[base_format][number]

size is written in decimal to specify the number of bits in the number.

Number format

To write a number in verilog we must follow the following syntax:

[size]'[base_format][number]

size is written in decimal to specify the number of bits in the number.

base_format can be:

- decimal `d`
- binary `b`
- octal `o`
- hexadecimal `h`

Number format

To write a number in verilog we must follow the following syntax:

[size]'[base_format][number]

size is written in decimal to specify the number of bits in the number.

base_format can be:

- decimal `d`
- binary `b`
- octal `o`
- hexadecimal `h`

number is simply the number written in the chosen base format

Number format

To write a number in verilog we must follow the following syntax:

[size]'[base_format][number]

size is written in decimal to specify the number of bits in the number.

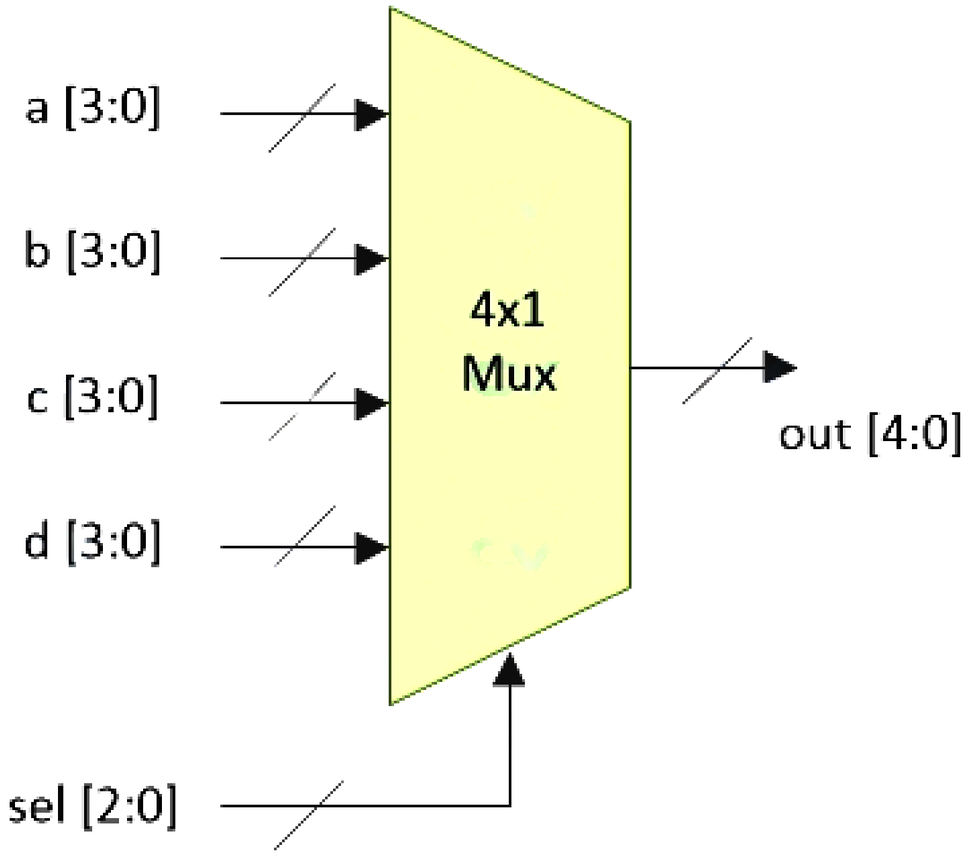
base_format can be:

- decimal `d`
- binary `b`
- octal `o`
- hexadecimal `h`

number is simply the number written in the chosen base format

```
1  3'b010;     // size is 3, base format is binary ('b), and the number is 010
   (indicates value 2 in binary)
2  3'd2;       // size is 3, base format is decimal ('d) and the number is 2
   (specified in decimals)
3  8'h70;      // size is 8, base format is hexadecimal ('h) and the number is
   0x70 (in hex) to represent decimal 112
```

MULTIPLEXER 4x1



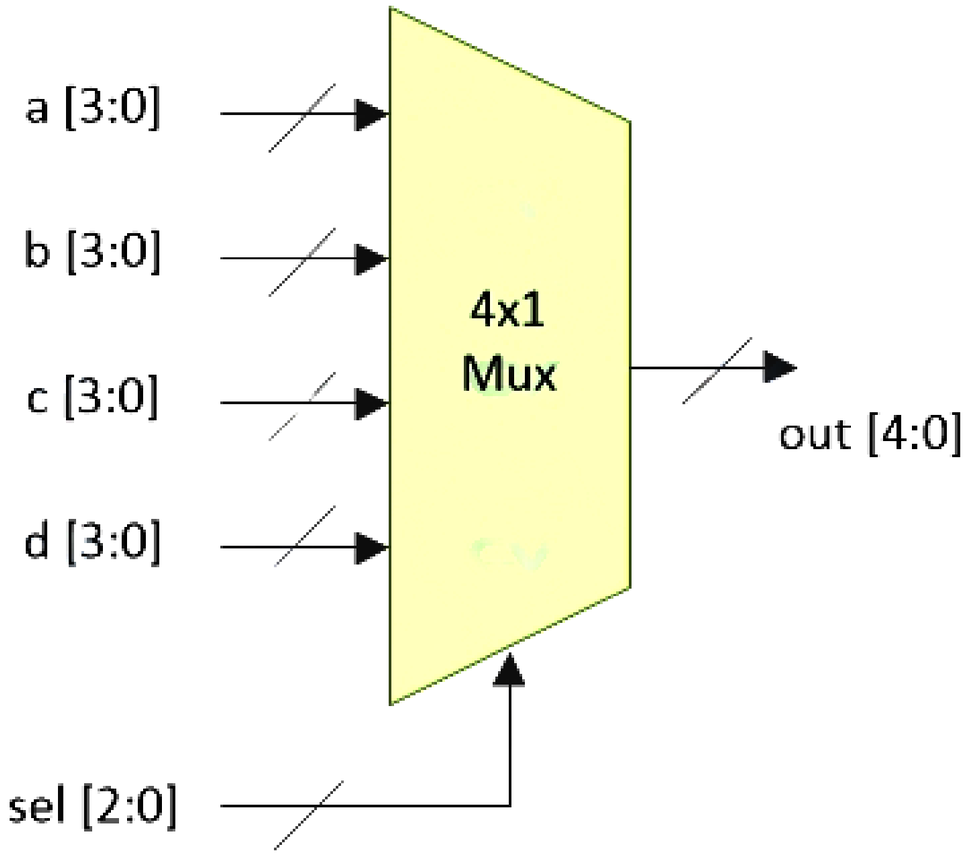
A multiplexer is a digital element that transfers data from one of the N inputs to the output **based on the select signal.**

sel is a 2-bit input and can have four values. Each value on the select line will **allow one of the four inputs to be sent to output pin out**.

A 4x1 multiplexer can be implemented in Verilog in multiple ways and now we'll see two of the most common ways:

- Using a **case** statement
- Using an **assign** statement

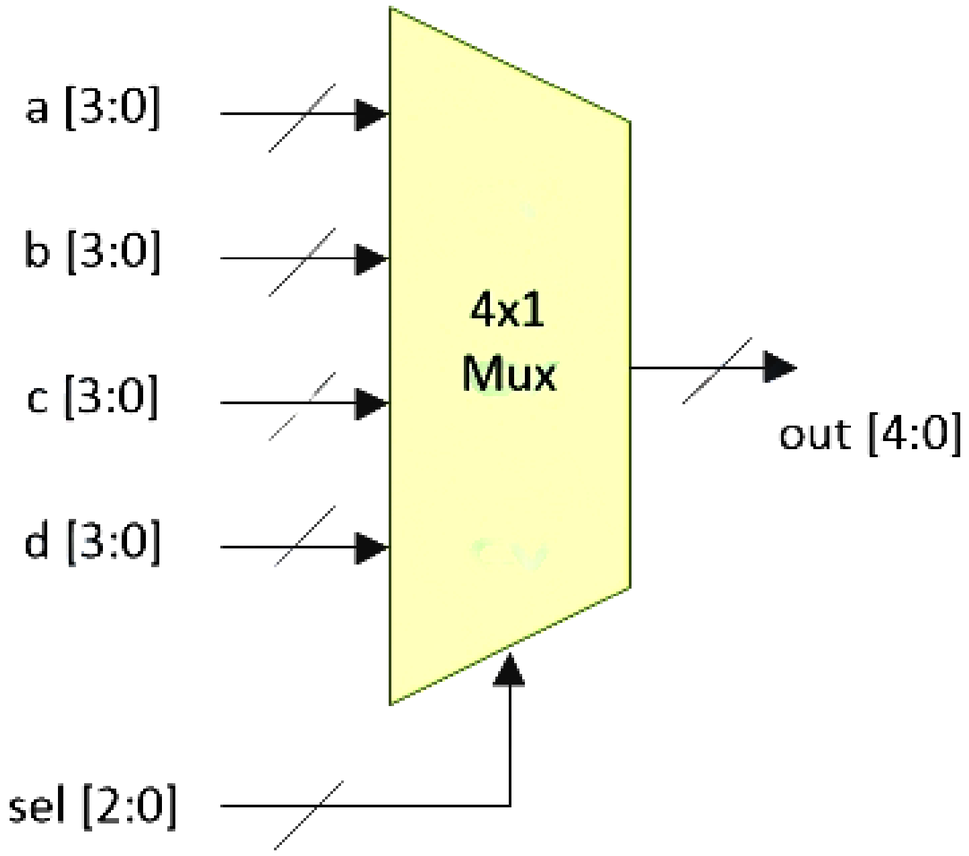
MULTIPLEXER 4x1



Case statement

```
1  module mux_4x1_case (
2      input [3:0] a,
3      input [3:0] b,
4      input [3:0] c,
5      input [3:0] d,
6      input [1:0] sel,
7      output [3:0] out
8  );
9
10     always @ (a or b or c or d or sel) begin
11         case (sel)
12             2'b00 : out <= a;
13             2'b01 : out <= b;
14             2'b10 : out <= c;
15             2'b11 : out <= d;
16         endcase
17     end
18 endmodule
```

MULTIPLEXER 4x1



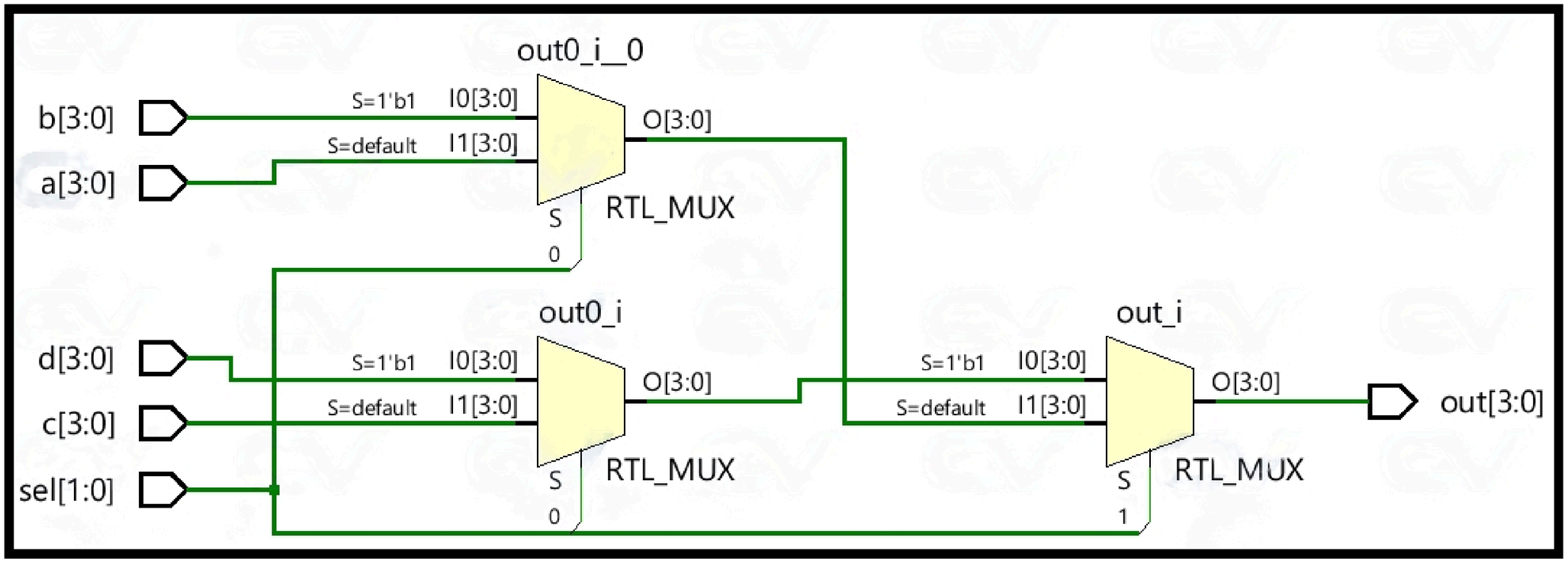
Assign statement

```
1  module mux_4x1_assign (
2      input [3:0] a,
3      input [3:0] b,
4      input [3:0] c,
5      input [3:0] d,
6      input [1:0] sel,
7      output [3:0] out
8  );
9
10     assign out = sel[1] ? (sel[0] ? d : c) : (sel[0] ? b : a);
11 endmodule
```

We write less code, but the reading is less clear

MULTIPLEXER 4x1

Hardware



In any case, both types of implementation get synthesized into the **same hardware**.

As you can see, in hardware there are only 2x1 multiplexers, other types of multiplexers are obtained by c**oncatenating them.**

That’s all

Thank you for your attention!