Hardware Description Languages

Index

Computer-Aided Design (CAD)

- Finding an efficient set of logic gates is **labour-intensive** and **error-prone**
- Designers became more productive by working at a **higher level of abstraction**:
  - specify only the **logical function**
  - let **CAD tools** generate the optimized gates

- **Hardware Description Languages (HDLs)**
  - describe **what** the circuit does
  - not **how** a processor executes instruction (not a programming language)
  - **VHDL** and **Verilog** – industry-standard HDLs
- Cannot be “executed” like a program, but can be **simulated** to verify behaviour
- After verification, can be **synthesized** into gates for implementation

- Targets:
  - **FPGA** – reconfigurable hardware for prototyping and experimentation
  - **ASIC** – custom chip for a specific function
- Tools:
  - Proprietary: **Xilinx Vivado**
  - Open-source: GHDL (simulation), Yosys (synthesis)

Levels of Abstraction

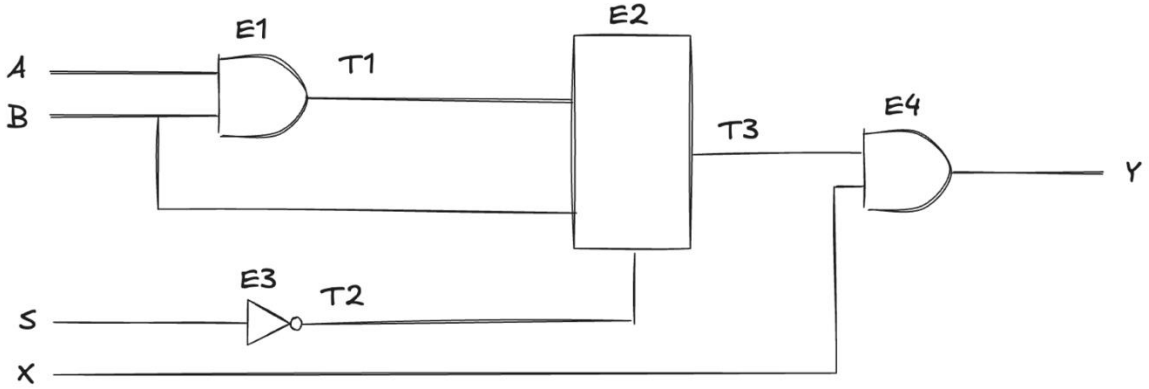
- HDLs allow the description of circuits at **different levels of abstraction**
  - each level exposes only the **necessary details** to manage complexity effectively
  - higher levels focus on **functionality** and **verification**
  - lower levels provide **control** over **timing**, **area**, and **implementation details**.

- Designers can move across levels to **design**, **test**, and **optimize** systems efficiently

- **Examples of abstraction levels:**
  - *Behavioural level* → describe what the circuit does
  - *RTL (Register Transfer Level)* → describe data transfers
  - *Structural (Gate) level* → describe interconnections between logic gates

Structural Level (gate-level)

- At the **lowest level of abstraction**, a circuit can be seen as a **graph**:
  - **Nodes** → logic elements (gates, multiplexers, subcircuits)
  - **Edges** → signal connections between them



- **Explicitly defines circuit elements** and their interconnections
- Keeps **functional behaviour implicit** (emerges from the structure)

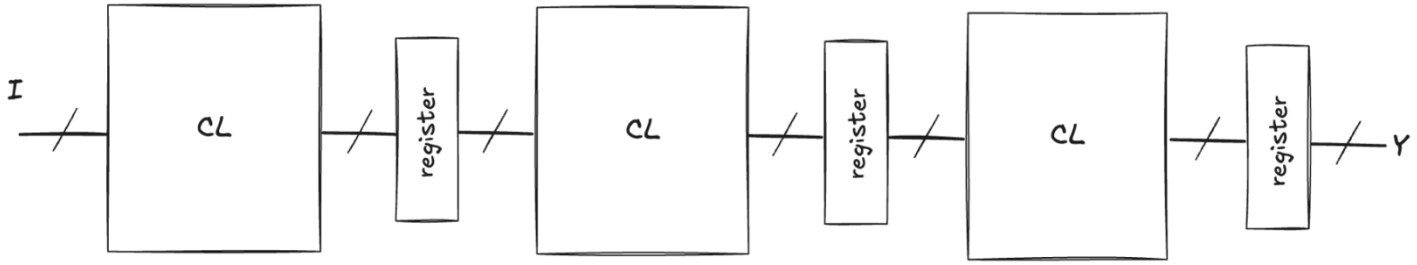
- Typical information includes:
  - names and types of **primary inputs/outputs** (A, B, S, X, Y)
  - types of **logic elements** (AND, NOT, etc.)
  - **instance names** of components (E1, E2, E3, E4)
  - **Internal signals** (T1, T2, T3)
  - **Connections** between signals and component ports

- Such a description forms a **netlist:** the **closest model to the final physical implementation** of the circuit

RTL level (data-flow)

- At a **higher level of abstraction**, circuits are described at the **Register Transfer Level (RTL)**
  - **how data is transformed and transferred** through the circuit

- Two key types of elements:
  - **combinational networks** → transform data using logic or arithmetic expressions
  - **memory elements** (registers) → store intermediate results between computations



- Each **subcomponent** fits the same concept
- Focus shifts from **physical structure** to **functional behaviour and data flow**

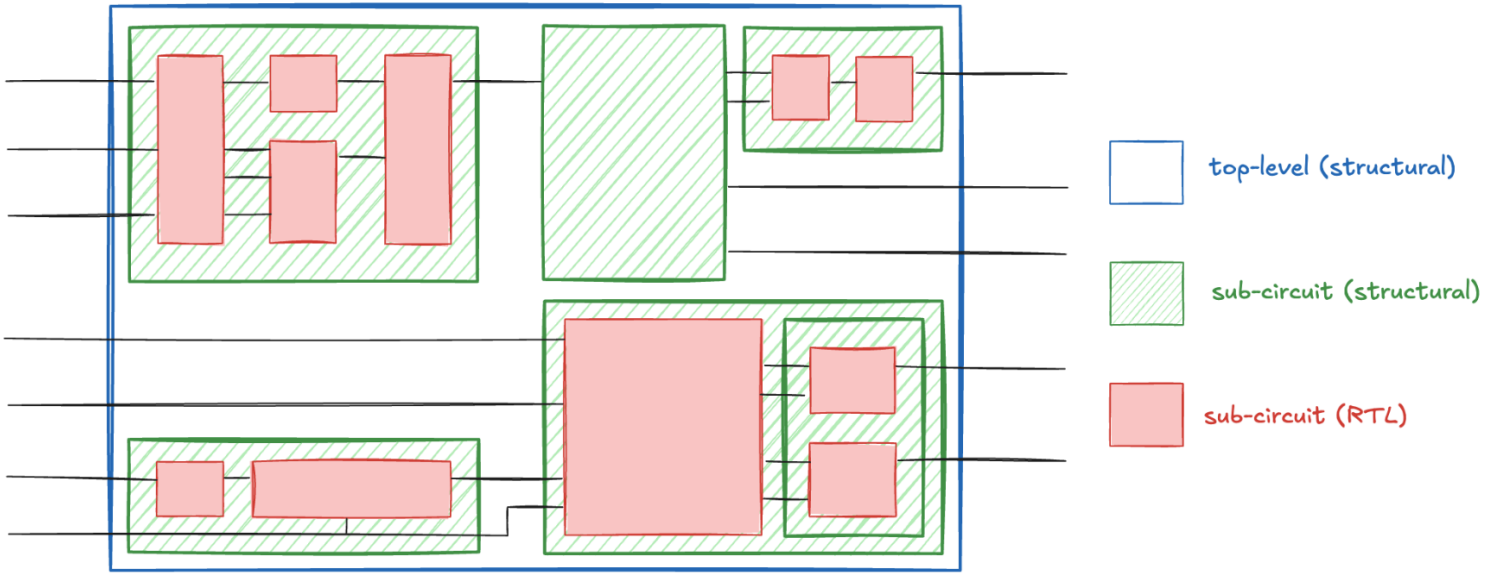
- Comparison:
  - Structural model → describes **how** the circuit is built
  - **RTL model** → describes **what** transformations occur and **where** data is stored

Algorithmic level (behavioural)

- Represents the **highest level of abstraction** in hardware design
  - describes **what the circuit does** in terms of **algorithms** (**behaviors**), not how it is built
  - the **structure** (registers, data paths, timing) is **not explicitly visible**
- The **synthesis tool** determines:
  - how to **allocate hardware resources**
  - according to designer **constraints** (e.g., max area)
- Offers the **greatest design flexibility but** requires more **sophisticated tools** for synthesis
- Behavioural-level descriptions are not covered in these notes

Specification

- As designs grow in complexity, they are organized as **hierarchical structures** of interconnected **subcircuits**



- Each subcircuit performs a **well-defined function** and should remain **modular** and **independent**
- Each submodule has its own ports, which connect either to primary I/O or to **internal signals**
- Smaller blocks are combined to form higher-level modules, until the **complete system** is defined (**top-level module**)

Structural for higher levels, RTL for leaf modules

- Although RTL is a higher abstraction level than gate-level descriptions: top-level modules are about integration, not computation

- **Structural descriptions suit higher levels**
  - specify **how modules connect**, not **how they operate internally**

- **RTL descriptions suit leaf modules**
  - they define **how data is processed** and **how registers interact**

- This separation promotes **modularity**, **readability**, and **reusability** in complex designs

Entities (1)

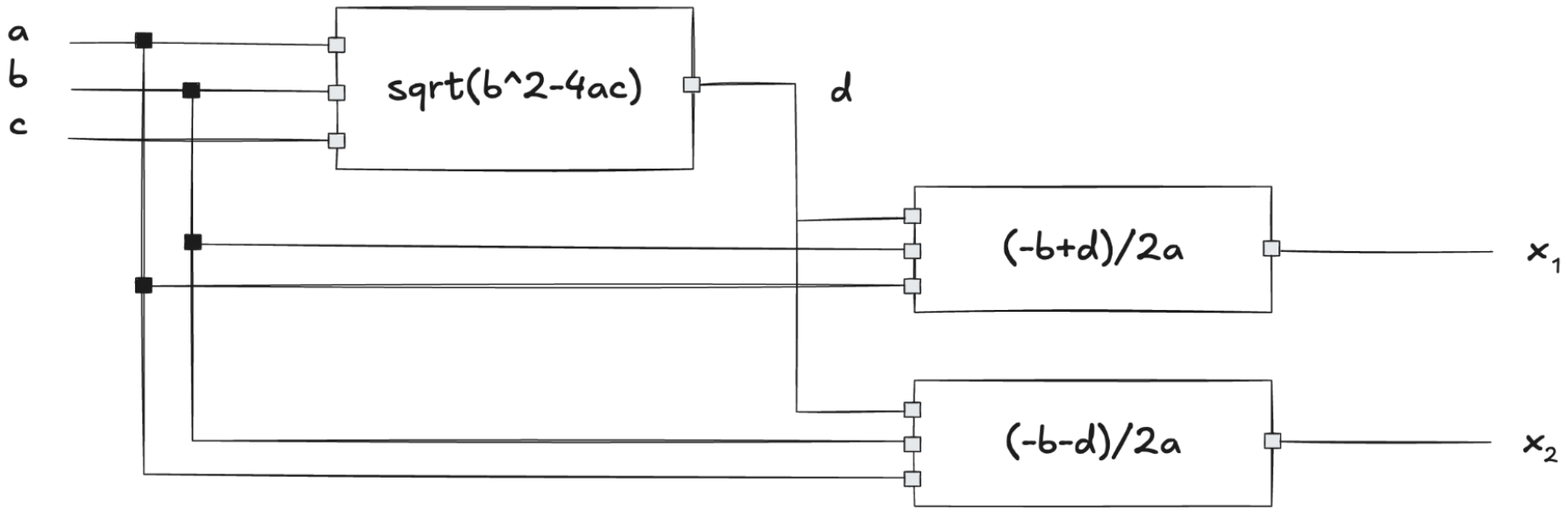
- Well-defined **functional units**
  - designed to **isolate a specific function** of the overall system
  - provides a **structured view** of the design
  - breaks down a very complex problem (developing an entire system) into a set of smaller, more manageable subproblems

- Example: a system that calculates the roots of a quadratic equation given the three coefficients a, b, and c

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$
$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

- One possible way to break down this problem:
  - 1 - compute the square root of the discriminant $d = \sqrt{b^2 - 4ac}$
  - 2- compute the first solution $x_1 = \frac{-b + d}{2a}$
  - 3 - compute the second solution $x_2 = \frac{-b - d}{2a}$

Entities (2)



- This decomposition is based on **three modules**:
  - one for computing the discriminant
  - two for computing the solutions from d, a, and b

- These modules are **relatively complex** and **sufficiently distinct** from each other that they are **unlikely to share resources** or be easily optimized together

Entities (3)

- We can further refine by observing that the solutions can also be derived by distributing the denominator:

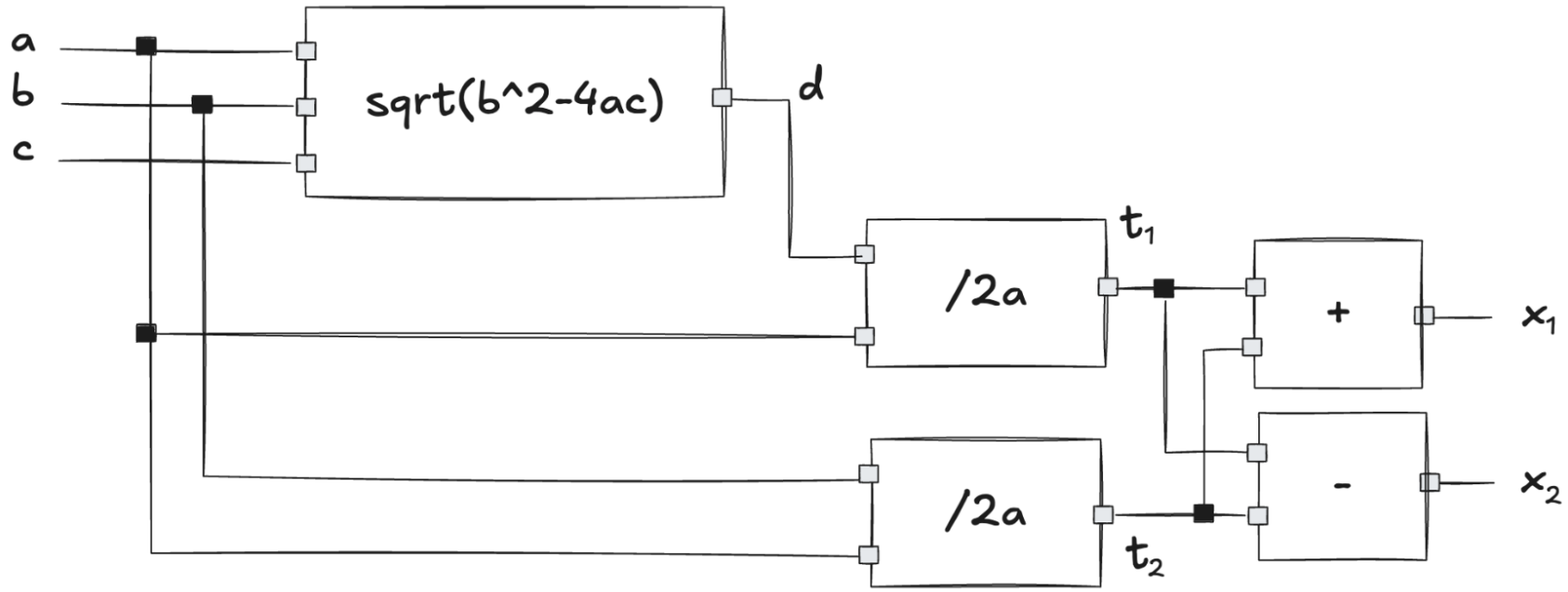
$$x_1 = \frac{-b}{2a} + \frac{d}{2a}$$

$$x_2 = \frac{-b}{2a} - \frac{d}{2a}$$

- This new decomposition highlights five distinct operations
  - 1. Compute the square root of the discriminant: $d = \sqrt{b^2 - 4ac}$
  - 2. Compute the first term of the solutions: $t_1 = \frac{-b}{2a}$
  - 3. Compute the second term of the solutions: $t_2 = \frac{d}{2a}$
  - 4. Compute the first solution: $x_1 = t_1 + t_2$
  - 5. Compute the second solution: $x_2 = t_1 - t_2$

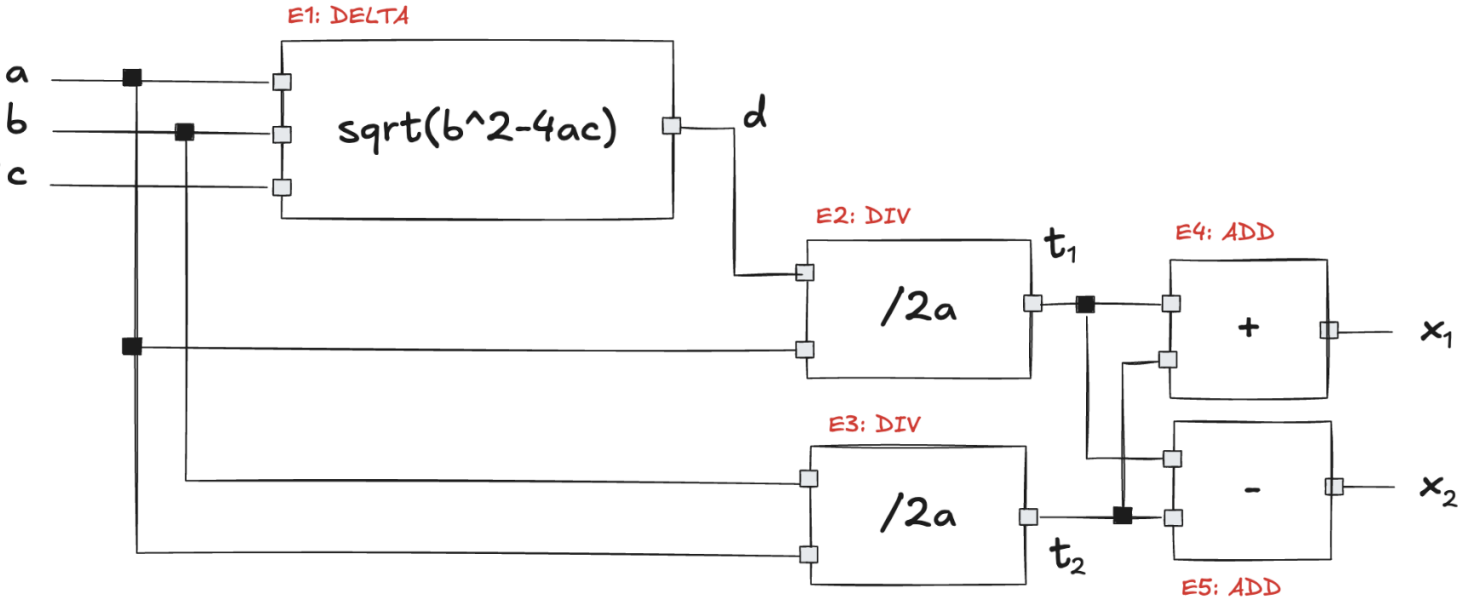
Entities (4)

- The division by 2a is performed twice: the same division block can be reused
- The module that performs addition or subtraction of can be used for two steps



Entities (5)

- **Exploit the reuse** of modules in the design:
  - **module:** a unique entity consisting of an interface and its defined behaviour
  - **Instance:** a specific use of a module within the circuit's construction



- Three modules (DELTA, DIV and SUM) and a five instances (E1, E2, E3, E4, E5)
- Each module is described by **separating** its two complementary aspects:
  - **Interface:** defines the input and output signals and specifies how the block connects to other blocks (**entity construct**)
  - **Behavior:** describes how the inputs are processed to produce the outputs (**architecture construct**)

Entity construct (1)

- The interface of each module is described by an **entity declaration**:
  - **name of the module**, must be unique within a design
  - **list of port declarations**, the signals used as inputs and outputs of the entity
  - **set of parameters** (called **generics**) used to adjust its properties

```
entity entity_name is
  [generic( generic_list );]
  [port( port_list );]
end entity_name;
```

- Each **port declaration** is structured according to the following syntax:

```
port_name[,port_name,...]: {in|out|inout} port_type;
```

  - **name**: used in the architecture to refer to the connected signal
  - **type**: indicate the kind of data that can be transmitted through that port (e.g. a single bit, a vector of bits, etc.)
  - **direction**:
    - **in** for input ports, can **only be read** (may appear only on the right-hand side of an assignment or in a conditional expression
    - **out** for output ports, can **only be written** may appear only on the left-hand side of an assignment)
    - **inout** for bidirectional port, can be both **read and written**
- Similarly, the definition of a generic follows the syntax:

```
generic_name[,generic_name,...]: generic_type;
```

  - no direction and can have complex types (floating-point numbers, strings)

Entity construct (2)

- Example: declaration of an AND gate
  - two input signals (A and B), one bit wide
  - one output signal (Y), one bit wide
  - we specify a delay parameter

- Generics: describing a component **parametrically**
  - Configurable components
  - for **simulation** (e.g the delay)
  - for **synthesis**

- As in software development, it is good practice to **enrich the specification with comments**
  - clarify key aspects
  - introduced using **--** to the end of the line

```
entity and_gate is
  generic (delay: real);
  port (A, B: in  bit;
        Y : out bit);
end and_gate;
```

```
entity and_gate is

    -- Generics
    generic(
        delay: real -- The delay in ns
    );

    -- Ports
    port(
        -- Inputs
        A: in bit; -- First operand
        B: in bit; -- Second operand

        -- Outputs
        Y: out bit; -- Output
    );

end and_gate;
```

Architecture construct (1)

- Describe the behaviour of an entity:
  - **list of declarations**: elements used to implement the behavior (constants, signals, user-defined types, and components), visible only within that specific architecture
  - **begin** and **end** keywords containing the **implementation section**

```
architecture architecture_name of entity_name is
    [declarations]
begin
    [implementation]
end architecture_name;
```

- The functionality of a circuit is therefore **distributed across the architecture declarations** of the various entities that make up the system's modules

- Example:
  - the implementation of the and gate

```
architecture first of and_gate is
begin
    Y <= A and B;
end first;
```

Architecture construct (2)

- Each architecture is **associated with exactly one entity**.
- However, it is possible to **define multiple alternative architectures for the same entity**

- Example:
  - a second architecture for the and gate

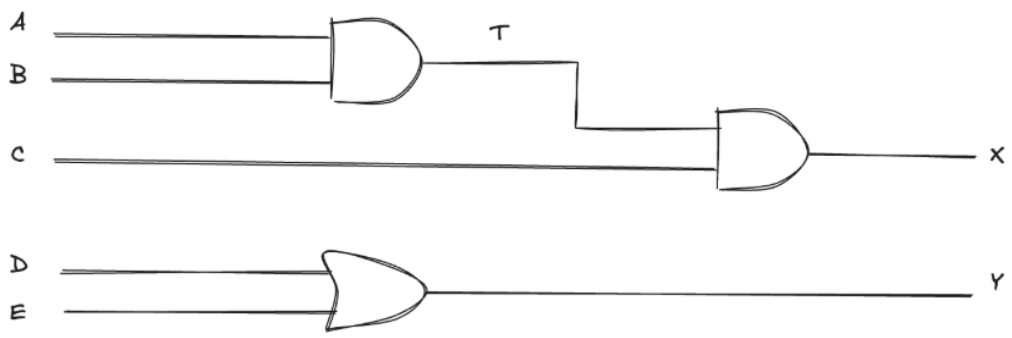
```
architecture second of and_gate is
    signal P: bit;
begin
    P <= A and B;
    Y <= P;
end second;
```

- In projects of **moderate complexity**, having multiple architectures is usually of limited benefit. However, in **larger and complex designs**, this approach can be very useful. For example: to maintain separate architectures for different synthesis strategies

Concurrency (1)

- The implementation section consists of a set of statements
  - **Assignments** (y <= a and b;)
  - **Conditional assignments** (e.g., y <= a when sel='1' else b;)
  - **Selection constructs** (e.g., case op is ... end case;)
  - **Instantiations** (e.g., U1: and2 port map(a, b, y);)
- In a typical programming language like C, these statements are **executed sequentially**
- In hardware description languages, these statements **are executed concurrently**, their results are **computed simultaneously**
- This distinction is **crucial**, as it reflects the fact that **all elements of a real circuit process their inputs inherently in parallel**

```
architecture par_one of dummy is
    signal T: bit;
begin
    T <= A and B;
    X <= T and C;
    Y <= D or E;
end par_one;
```



Concurrency (2)

- Logic gates are **physical hardware components**, they **continuously and simultaneously produce an output** based on their current inputs, they **don't wait their turn**

- In the example:
  - Y, X and T are **always available** and are **continuously updated**
  - any change to inputs (A, B, C, D, E) will **propagate through the circuit**, updating T, X Y according to the data dependencies
  - the two AND gates and the OR gate **process their inputs in parallel**

- **The order of the expressions can be rearranged without changing the behaviour of the resulting circuit**.
  - the following architecture is therefore completely equivalent to the previous one:

```
architecture par_two of dummy is
    signal T: bit;
begin
    Y <= D or E;
    X <= T and C;
    T <= A and B;
end par_two;
```

Types

- VHDL is **flexible** and allows to define **many different kinds of data types**
  - bits, bit vectors, integers, floating-point numbers, enumerated types, records, and even user-defined types
  - **great for simulation** (to test a wide range of scenarios and write models that can look more like software)
  - not to **move from simulation to synthesis** (when design is turned into hardware

- The hardware must **physically implement** whatever types we've used:
  - **some types map naturally to hardware** (bit)
  - other types (floating point or complex user-defined records) **do not have a straightforward or efficient hardware representation** or may not be supported at all

- In practice, designers usually **stick to a smaller synthesizable subset of data types**, since synthesis tools know how to translate these into real hardware

bit type

- The simplest type available and **represents a binary value** that can only take the logical values 0 and 1
- The operators defined for this type include
  - **assignment operators**
  - **comparison operators**
  - **logical operators**

```
X <= A and B;
Y <= '1';
Z <= A xor (B and not C);
```

bit_vector type

- We often have **many signals that are related**
  - binary number, lines of a bus, data lines connecting two components
  - we can group them together under one name

- **Bit vector**

```
signal_name: bit_vector( index1 {to|downto} index2 );
```

  - **composite signal** made up of (index2 − index1 + 1) bit
  - **ordering** by **to** or **downto** defines which bit is the ****most significant bit****

```
signal A : bit_vector(0 to 7);     -- A(0) is LSB, A(7) is MSB
signal B : bit_vector(7 downto 0); -- B(7) is MSB, B(0) is LSB
```

  - to **access a specific element** in the vector, write its index in parentheses:

```
A(5) -- the element 5 of vector A
```

Integer type

- Represent **numbers** using **32 bits**, useful to describe **counters**, **loop indices**, or **arithmetic operations** that go beyond simple bits
- Keep in mind that we are **describing hardware**, not **just software** and we need to be careful
- Should they be treated as **signed** or **unsigned**?
  - by default, they're **unsigned**
  - negative numbers should be made explicit
- Declaring a signal as an integer means the synthesis tool sees a 32-bit, not just a small counter or value
  - just to count from 0 to 7, the tool will build a 32-bit adder and a bank of 32 flip-flops
  - **wasting area** and **power** when only 3 bits would do

IEEE types

- The **bit** and **bit_vector** types have some limitations
  - do not allow to specify **don't care conditions**
  - or **high-impedance states**

- **Standard IEEE library** defines additional types

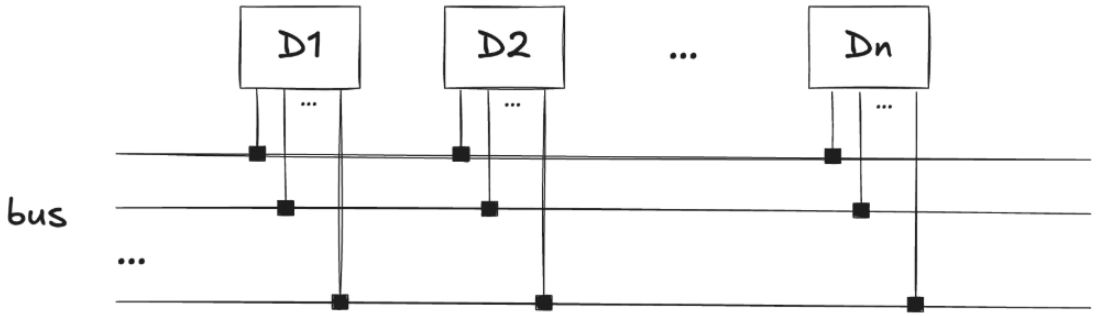
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

signal A : STD_LOGIC;
signal B : STD_LOGIC_VECTOR(7 downto 0);
```

  - extend basic logic with a **9-value system** as described below
    - 0 : logical value 0
    - 1 : logical value 1
    - Z' : high impedance
    - X : unknown value (could be 0 or 1)
    - U : uninitialized (no value has ever been assigned)
    - W : weak unknown signal (cannot be clearly interpreted as 0 or 1)
    - L : weak signal, interpreted as 0
    - H : weak signal, interpreted as 1
    - - : don't care condition

Resolved types

- In well-designed combinational logic, we should **not directly connect multiple outputs to the same wire**
  - **contention** and **physically unsafe**
- In real hardware, sometimes we allow **multiple drivers on a shared bus**



- A way to determine what the **actual signal** if the drivers conflict
- A **resolved type** uses a **resolution function** to combine multiple drivers into a single final value according to **predefined rules**
  - e.g. H overrides L, conflicts result in X
  - **std_logic** and **std_logic_vector** are resolved types
- An **unresolved types** (like bit) cannot handle multiple drivers

User-defined types

- A **new type**
  - characteristics must be explicitly specified
  - any operator needed to work with this type should be specified

- VHDL is flexible and offers **several mechanisms** for creating custom types
- However, it is usually sufficient to consider just two of these mechanisms

  - **Subtyping**: involves defining a new type that is equivalent to an existing type but with **a restricted range** of valid values

    ```
    subtype new_type_name is type_name range val1 to val
    ```

    - useful for integers, to ensure a binary representation with minimum number of bits
    - not recommended (**rely on the synthesis tool**) compared to explicitly declaring a std_logic_vector or bit_vector (**hardware representation clear**)

  - **Enumeration**: listing all the symbolic values it can assume

    ```
    type new_type_name is ( val0, val1, ..., valN );
    ```

    - can makes logic more readable and self-explanatory when you select operations in your design

Slice

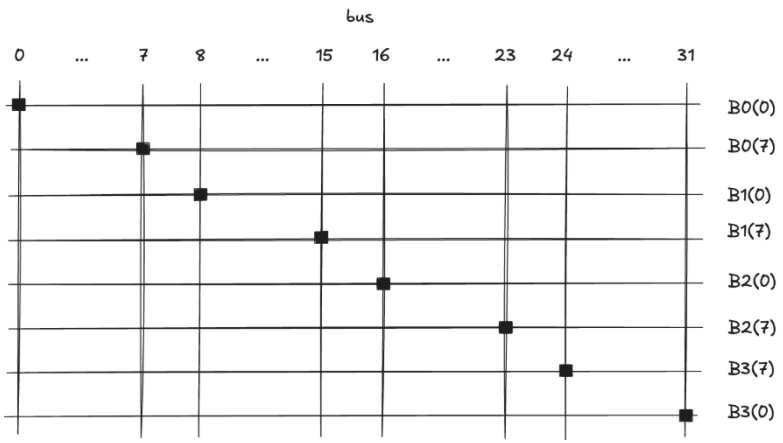
- Essential operations to **extract parts of a vector**

  `signal_name( index1 {to|downto} index2 )`

  - index1 and index2 must be valid indices for the vector signal name
  - must respect the ordering defined by the to or downto clause
  - circuit perspective: a **subset of the lines** within a composite signal

- Often used to **extract certain lines from a bus under a single, more meaningful name**
  - example: a bus that carries an entire word (32 bits) splitted into the individual bytes (8 bits each)

```
architeture rtl of dummy is
    signal BUS: std_logic_vector(0 to 31);
    signal B0, B1, B2, B3: std_logic_vector(0 to 7);
begin
    ...
    B0 <= BUS(0 to 7);
    B1 <= BUS(8 to 15);
    B2 <= BUS(16 to 23);
    B3 <= BUS(31 downto 24);
    ...
end rtl;
```



Concatenate

- The **opposite** of slice: it **combines multiple signals** under a single name.
  - operator **&**

- Examples:
  - reassemble a new bus bus with swapped order

```
BUS2 <= B3(7 downto 0) & B2 & B1 & B0;
```

  - group independent signals under a single name

```
if ( A = '1' and B = '0' and C = '1' ) then ...
```

```
signal temp: std_logic_vector(0 to 2);
temp <= A & B & C;
if ( temp = "101" ) then ...
```

Logical Expressions

- The operators available for constructing logical expressions are:

Logical Operator	VHDL Operator	Example
AND	and	A and B
OR	or	A or B
NOT	not	not A
XOR	xor	A xor B

- Logical expressions describe how signals should be combined to produce other signals, and it is implemented using the **signal assignment operator <=**

$$F = A\,\overline{C} + D\,(\overline{A} + \overline{B}\,C)$$

```
F <= A and (not C) or D and ( (not A) or ((not B) and C) );
```

- Logical expressions belong to the style of specification

Truth Tables (1)

- We can express truth tables using logical expressions and **conditional assignments:**

```
signal_name <= const_1 when cond_1 else
               const_2 when cond_2 else
               ...
               const_N when cond_N else
               const_default;
```

- Example:

A	B	F
0	0	1
0	1	0
1	0	1
1	1	1

```
F <= '1' when A='0' and B='0' else
     '0' when A='0' and B='1' else
     '1' when A='1' and B='0' else
     '1' when A='1' and B='1';
```

Truth Tables (2)

- Conditional assignment expressions **must explicitly cover all possible cases**
- If signals were of type std_logic, the assignment must take on **nine possible values** for each variable, which is impractical
- A common solution is to **explicitly specify the main conditions** and then **provide a default value** to cover any cases not explicitly listed:

```
F <= '1' when A='0' and B='0' else
     '0' when A='0' and B='1' else
     '1' when A='1' and B='0' else
     '1';
```

- By following this strategy, it's possible to simplify by **considering only the on-set or the off-set**:

```
F <= '0' when A='0' and B='1' else
     '1';
```

  - not always the best solution in terms of clarity
  - only applicable to completely specified functions

Truth Tables (3)

- To support also **don't care**, we need to use signals of type std_logic

A	B	F
0	0	1
0	1	0
1	0	-
1	1	1

```
F <= '1' when A='0' and B='0' else
     '0' when A='0' and B='1' else
     '-' when A='1' and B='0' else
     '1' when A='1' and B='1';
```

```
F <= '1' when A='0' and B='0' else
     '0' when A='0' and B='1v else
     '-' when A='1' and B='0' else
     '1';
```

```
F <= '0' when A='0' and B='1' else
     '-' when A='1' and B='0' else
     '1';
```

Truth Tables (3)

- VHDL provides an **alternative construct for conditional assignment**

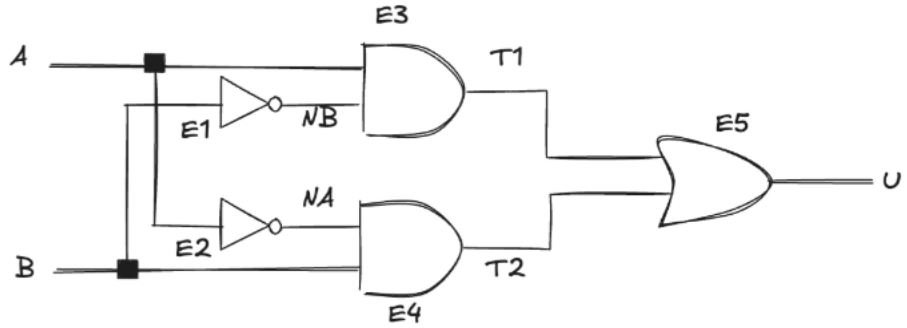
```
with signal_test select
    signal_name <= const_1 when case_1,
                   const_2 when case_2,
                   ...
                   const_N when case_N,
                   const_default when others;
```

  - more concise comparison expressions

```
temp <= A & B;
with temp select
    F <= '1' when "00",
         '0' when "01",
         '-' when "10",
         '1' when "11",
         '-' when others;
```

Structural Description (1)

- What we have covered so far is sufficient for specifying modules of **low or moderate complexity**
- Not well-suited for designing **highly complex circuits**
- As in many other areas of science and engineering, a good approach to solving a complex problem is to **break it down into simpler subproblems** and then combine their solutions.
  - identifying the system's basic functions
  - designing components that implement these functions*
  - then interconnecting these modules appropriately to build the complete system
- Simple example: implement XOR gate using only basic gates OR, AND, NOT



$$A \oplus B = (A\,\overline{B}) + (\overline{A}\,B)$$

Structural Description (2)

- We first create the basic components:

```
-- The NOT gate
entity NOT_GATE is
    port( X: in std_logic;
          Z: out std_logic );
end NOT_GATE;

architecture rtl of NOT_GATE is
begin
    Z <= not X;
end rtl;
```

```
-- The 2-input AND gate
entity AND2_GATE is
    port( X: in std_logic;
          Y: in std_logic;
          Z: out std_logic );
end AND2_GATE;

architecture rtl of AND2_GATE is
begin
    Z <= X and Y;
end rtl;
```

```
-- The 2-input OR gate
entity OR2_GATE is
    port( X: in std_logic;
          Y: in std_logic;
          Z: out std_logic );
end OR2_GATE;

architecture rtl of OR2_GATE is
begin
    Z <= X or Y;
end rtl;
```

- We can solve the problem **by connecting these components appropriately.** We start by defining the entity for the XOR gate, which will have two inputs and one output:

```
entity XOR2_GATE is
    port( A: in std_logic;
          B: in std_logic;
          U: out std_logic );
end XOR2_GATE;
```

Structural Description (3)

- To avoid any misunderstanding: there is a **fundamental difference** between the operator "and" the "AND2_GATE" component we defined
  - the operator is built into the language and **can only be used inside expressions**
  - AND2_GATE is a generic component that **cannot be used in an expression, it must be instantiated**, which means inserting it into the design specification and connecting it properly to other components using signals

- Considering this point, the declarative part of the architecture **must specify the names and types of any components it uses**

```
component component_name is
    [generic( generic_list );]
    port( port_list );
end component;
```

  - it is identical to its corresponding entity declaration, except that it uses the **keyword component** instead of entity

Structural Description (4)

```
architeture structural of XOR2_GATE is

    -- The NOT gate
    component NOT_GATE is
        port( X: in std_logic;
              Z: out std_logic );
    end component;

    -- The 2-input AND gate
    component AND2_GATE is
        port( X: in std_logic;
              Y: in std_logic;
              Z: out std_logic );
    end component;

    -- The 2-input OR gate
    component OR2_GATE is
        port( X: in std_logic;
              Y: in std_logic;
              Z: out std_logic );
    end component;

    -- Internal signals
    signal NA: std_logic;
    signal NB: std_logic;
    signal T1: std_logic;
    signal T2: std_logic;

begin
    ...
end structural;
```

Structural Description (5)

- We have declared everything needed to proceed with **building the circuit**
- we need to **instantiate the components**

```
instance_name: component_name
    [generic map( generic_assignment_list );]
    port map( port_assignment_list );
```

  - **assign a name** to each instance (**unique within the architecture**
    - **no connection** to the name of the component being instantiated
  - specifies **how ports of the instance are connected to the signals in the architecture**

```
port map( port_1 => signal_1, ..., port_N => signal_N );
```

- At this point, we can complete the body of the XOR architecture by simply listing all the instances that make it up:

```
...
begin
    U1: NOT_GATE
        port map( X => B, Z => NB );
    U2: NOT_GATE
        port map( X => A, Z => NA );
    U3: AND2_GATE
        port map( X => A, Y => NB, Z => T1 );
    U4: AND2_GATE
        port map( X => NA, y => b, z => T2 );
    U5: OR2_GATE
        port map( X => T1, Y => T2, Z => U );
end structural;
```

Process (1)

- A **composed concurrent statement**
  - like a signal assignment or a conditional assignment
  - groups a set of sequential statements **executed in order**
  - the process itself is **evaluated concurrently** with other statements

```
[process_name]: process ( sensitivity_list )
    [declarations]
begin
    [body]
end process;
```

- A (optional) **name**, to improve readability and clarity)
- A (optional) declarative region
- A **statement body**
- A **sensitivity list**
  - signals that can **trigger the execution**
  - the process is evaluated only when an "event" occurs on one or more signals in the sensitivity list

Process (2)

- To clarify the concept, let's consider a general concurrent statement

  $$Z <= (X + Y)\ T$$

  - it automatically update Z whenever there is a change (**event**) on any of the signals X, Y, or T
  - hardware this statement continuously monitors these signals in parallel

- To describe the same logic inside a process, we need to make sure the process **knows which signals should trigger** its execution:

```
sample: process(X, Y, T)
begin
    Z <= (X or Y) and T;
end process;
```

- The **sequential constructs** that can be used **inside the body of a process** are the same as those found in **most programming languages**:
  - expressions and assignments, conditional statements, and loops
  - loops require **care**, they can lead to issues when synthesizing into hardware

Testbench (1)

- A special module used to **verify the behaviour of another module** (device under test **DUT**)
  - generates and applies input signals to the DUT
  - observes the outputs to ensure they match the **expected results**
  - the sets of input values and their corresponding expected outputs are called **test vectors**
- Example:

$$Y = (\overline{A}\,\overline{B}\,\overline{C}) + (A\,\overline{B}\,\overline{C}) + (A\,\overline{B}\,C$$

```
entity sillyfunction is
    port(A, B, C: in STD_LOGIC;
         Y: out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
    y <= (not A and not B and not C) or
         (A and not B and not C) or
         (A and not B and C);
end;
```

Testbench (2)

```
entity testbench is
end;

architecture sim of testbench is
    component sillyfunction
        port(A, B, C: in STD_LOGIC;
             Y: out STD_LOGIC);
    end component;

    signal A, B, C, Y: STD_LOGIC;

begin
    -- instantiate device under test
    DUT: sillyfunction port map(B, B, C, Y);

    -- apply inputs one at a time checking results
    process begin
        A <= '0'; B <= '0'; C <= '0'; wait for 10 ns;
        assert Y = '1' report "000 failed.";

        ...

        C <= '1'; wait for 10 ns;
        assert Y = '0' report "111 failed.";

        wait;
    end process;
end;
```

Testbench (3)

- Testbenches are **simulated** in the same way as other HDL modules; however, they are **not synthesizable** and are **used purely for verification purposes**