VHDL Circuit Design: From Universal Gates to Hierarchical Systems

This report details the design and implementation of various digital circuits using VHDL (Very High Speed Integrated Circuit Hardware Description Language), focusing on fundamental logic gates constructed from universal NAND gates, hierarchical multiplexer design, and a 5-input majority circuit. The discussion emphasizes VHDL modeling styles, component reuse, and adherence to best practices for synthesizable hardware.

Introduction to VHDL Design Principles

VHDL serves as a powerful Hardware Description Language, enabling the abstraction and design of complex digital circuits. Its utility is rooted in its ability to describe hardware at various levels of abstraction, facilitating a structured and efficient design process. VHDL offers three primary modeling styles: Dataflow, Behavioral, and Structural, each distinguished by the type of concurrent statements employed within its architecture.[1]

The **Dataflow style** describes how data propagates and transforms through a system, utilizing concurrent signal assignment statements. This approach directly implies a corresponding gate-level implementation, making it suitable for designs where the flow of data is explicit.[1] In contrast, the **Behavioral style** describes a system's functionality algorithmically, often employing process statements that contain sequential statements. This is considered the most abstract style, as it does not directly dictate a specific gate-level realization.[1] Finally, the **Structural style** describes an entity as an interconnection of various components. This method is particularly effective for decomposing intricate systems into smaller, reusable modules.[1]

The choice among these modeling styles is not merely a stylistic preference; it significantly influences how synthesis tools interpret the VHDL code. Behavioral descriptions, while offering flexibility, may lead to unexpected hardware if not meticulously constrained, such as the unintended inference of latches when all possible conditions are not explicitly handled. Conversely, dataflow and structural styles provide more direct control over the resulting gate-level implementation. This decision represents a critical balance between design complexity, code readability, and the desired hardware outcome.

A cornerstone of modern digital design is the principle of **hierarchical design and**

component reuse. This paradigm allows complex projects to be systematically broken down into simpler, more manageable sub-designs, thereby simplifying the overall design process and reducing complexity.[1] The ability to reuse already implemented and verified modules, rather than redesigning them from scratch, significantly accelerates design cycles and enhances reliability.[2] This modular approach, often referred to as "divide and conquer," is directly analogous to strategies employed in software engineering, but with distinct implications for hardware. It enables parallel development by different engineering teams and promotes the reuse of intellectual property (IP) blocks, which are pre-verified design components. The instantiation of proven building blocks, such as a 2-to-1 MUX to construct a 4-to-1 MUX, is a testament to this principle, underscoring its role in managing complexity in contemporary ASIC and FPGA development.

I. Universal Logic: Implementing Basic Gates with NAND Gates

The NAND gate holds a unique and fundamental position in digital electronics due to its "universal" property. This characteristic implies that any Boolean function, no matter how complex, can be constructed using only NAND gates.[5] This functional completeness allows for the implementation of all other basic logic functions, including NOT, AND, OR, XOR, NOR, and XNOR, solely through various configurations of NAND gates.[7]

The requirement to implement basic gates using only NAND gates and port mapping necessitates a structural VHDL approach. This involves defining a foundational nand_gate entity as a reusable component and then instantiating it multiple times to construct more complex logic functions. This methodology powerfully illustrates how even the most fundamental logic operations can be abstracted into modular, reusable components. By defining the nand_gate once, designers can then build functions like AND, OR, or XOR by merely describing the interconnections of these nand_gate instances, thereby reinforcing a hierarchical design philosophy from the ground up.

A. NOT Gate from NAND

A NOT gate, also known as an inverter, produces an output that is the logical complement of its input. When using a NAND gate, this functionality is achieved by simply connecting both inputs of a 2-input NAND gate together.[5] If the single input is '0', the NAND gate outputs '1' (NOT 0). If the input is '1', the NAND gate outputs '0' (NOT 1).

Truth Table for NOT Gate (from NAND)

Input A	Output Q
0	1
1	0

VHDL Structural Code for NOT Gate (using nand_gate component)

VHDL

```
-- nand_gate component (assumed to be defined elsewhere, e.g., behaviorally)
-- entity nand_gate is
--   port (i1, i2 : in std_logic; o1 : out std_logic);
-- end nand_gate;
-- architecture Behavioral of nand_gate is
-- begin
--   o1 <= not (i1 and i2);
-- end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity not_gate_from_nand is
    port (
        A : in std_logic;
        Q : out std_logic
    );
end not_gate_from_nand;

architecture Structural of not_gate_from_nand is
    component nand_gate
        port (i1, i2 : in std_logic; o1 : out std_logic);
    end component;
begin
    -- Instantiate a NAND gate with both inputs tied to A
    U1_NOT : nand_gate
        port map (
```

```
        i1 => A,
        i2 => A,
        o1 => Q
    );
end Structural;
```

B. AND Gate from NAND

An AND gate produces a high output only when all its inputs are high. To construct an AND gate using only NAND gates, the output of a standard NAND gate is inverted. This is achieved by taking the output of a NAND gate (A NAND B) and feeding it into another NAND gate configured as a NOT gate (by tying its inputs together).[7] The Boolean expression for this configuration is Q = (A NAND B) NAND (A NAND B).

Truth Table for AND Gate (from NAND)

Input A	Input B	Output Q
0	0	0
0	1	0
1	0	0
1	1	1

VHDL Structural Code for AND Gate (using nand_gate component)

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and_gate_from_nand is
    port (
        A, B : in std_logic;
        Q    : out std_logic
```

```
    );
end and_gate_from_nand;

architecture Structural of and_gate_from_nand is
    component nand_gate
        port (i1, i2 : in std_logic; o1 : out std_logic);
    end component;
    signal s_nand_out : std_logic; -- Internal signal for the output of the first NAND
begin
    -- First NAND gate: performs A NAND B
    U1_NAND : nand_gate
        port map (
            i1 => A,
            i2 => B,
            o1 => s_nand_out
        );

    -- Second NAND gate: inverts the output of the first NAND (s_nand_out NAND s_nand_out)
    U2_INVERTER : nand_gate
        port map (
            i1 => s_nand_out,
            i2 => s_nand_out,
            o1 => Q
        );
end Structural;
```

C. OR Gate from NAND

An OR gate produces a high output if at least one of its inputs is high. The implementation of an OR gate using NAND gates leverages De Morgan's theorem. Specifically, A + B is equivalent to (A' · B')'. Applying this with NAND gates, the inputs A and B are first inverted using two separate NAND gates (each configured as a NOT gate). The outputs of these two inverters (A' and B') are then fed into a third NAND gate. The output of this third NAND gate will be (A' NAND B'), which simplifies to A + B.[6] The Boolean expression is Q = (A NAND A) NAND (B NAND B).

Truth Table for OR Gate (from NAND)

Input A	Input B	Output Q

0	0	0
0	1	1
1	0	1
1	1	1

VHDL Structural Code for OR Gate (using nand_gate component)

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity or_gate_from_nand is
    port (
        A, B : in std_logic;
        Q    : out std_logic
    );
end or_gate_from_nand;

architecture Structural of or_gate_from_nand is
    component nand_gate
        port (i1, i2 : in std_logic; o1 : out std_logic);
    end component;
    signal s_not_A : std_logic; -- Internal signal for NOT A
    signal s_not_B : std_logic; -- Internal signal for NOT B
begin
    -- First NAND gate: NOT A (A NAND A)
    U1_NOT_A : nand_gate
        port map (
            i1 => A,
            i2 => A,
            o1 => s_not_A
        );
```

```
    -- Second NAND gate: NOT B (B NAND B)
    U2_NOT_B : nand_gate
        port map (
            i1 => B,
            i2 => B,
            o1 => s_not_B
        );

    -- Third NAND gate: (NOT A) NAND (NOT B) which is equivalent to A OR B
    U3_OR : nand_gate
        port map (
            i1 => s_not_A,
            i2 => s_not_B,
            o1 => Q
        );
end Structural;
```

D. XOR Gate from NAND

An XOR gate, or Exclusive OR gate, produces a true output only when its two inputs differ (one is true and the other is false).[9] Implementing an XOR gate using only NAND gates is a classic demonstration of NAND gate universality. A common configuration uses four NAND gates.[7] The Boolean expression for the 4-gate version is Q = NAND.[7] An alternative 5-gate version exists, though the 4-gate version is generally preferred for its efficiency.[7]

Truth Table for XOR Gate (from NAND)

Input A	Input B	Output Q
0	0	0
0	1	1
1	0	1
1	1	0

VHDL Structural Code for XOR Gate (using nand_gate component)

VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity xor_gate_from_nand is
    port (
        A, B : in std_logic;
        Q    : out std_logic
    );
end xor_gate_from_nand;

architecture Structural of xor_gate_from_nand is
    component nand_gate
        port (i1, i2 : in std_logic; o1 : out std_logic);
    end component;
    signal s_nand1_out : std_logic; -- Output of first NAND (A NAND B)
    signal s_nand2_out : std_logic; -- Output of second NAND (A NAND s_nand1_out)
    signal s_nand3_out : std_logic; -- Output of third NAND (B NAND s_nand1_out)
begin
    -- First NAND gate: (A NAND B)
    U1_NAND_AB : nand_gate
        port map (
            i1 => A,
            i2 => B,
            o1 => s_nand1_out
        );

    -- Second NAND gate: A NAND (A NAND B)
    U2_NAND_A_NAND1 : nand_gate
        port map (
            i1 => A,
            i2 => s_nand1_out,
            o1 => s_nand2_out
```

```
    );

  -- Third NAND gate: B NAND (A NAND B)
  U3_NAND_B_NAND1 : nand_gate
    port map (
      i1 => B,
      i2 => s_nand1_out,
      o1 => s_nand3_out
    );

  -- Fourth NAND gate: (Output of U2) NAND (Output of U3)
  U4_FINAL_NAND : nand_gate
    port map (
      i1 => s_nand2_out,
      i2 => s_nand3_out,
      o1 => Q
    );
end Structural;
```

While NAND gates are theoretically universal, implementing all basic gates solely with them often results in a higher gate count and increased propagation delay compared to using native AND, OR, or XOR gates. For instance, an XOR gate constructed from four NAND gates incurs a propagation delay three times that of a single NAND gate.[7] This exemplifies a fundamental trade-off in digital design: the theoretical elegance of universality versus the practical considerations of area, power consumption, and speed. In real-world FPGA and ASIC design, engineers typically leverage direct VHDL operators (and, or, xor) or instantiate optimized library cells, allowing synthesis tools to map these to the most efficient underlying hardware structures. The exercise of building gates exclusively from NAND gates is invaluable for understanding fundamental logic synthesis principles but is less common for direct implementation in modern HDL design.

II. Multiplexer Design and Hierarchical Structural Modeling

Multiplexers (MUXes) are essential combinational circuits used to select one of several input signals and route it to a single output line, based on the value of a set of selection inputs.[2] This section details the design of a 2-to-1 MUX and subsequently demonstrates how to construct a 4-to-1 MUX hierarchically by reusing the 2-to-1 MUX as a component, showcasing structural modeling with port mapping.

A. 2-to-1 Multiplexer (MUX)

A 2-to-1 MUX features two data inputs (typically labeled X and Y), one select input (S), and a single output (Z). Its functionality is straightforward: if the select input S is '0', the output Z takes the value of input X; if S is '1', the output Z takes the value of input Y.[3]

Truth Table for 2-to-1 MUX

Select (S)	Data Input (X)	Data Input (Y)	Output (Z)
0	X	Y	X
1	X	Y	Y

The 2-to-1 MUX is most effectively implemented using a behavioral VHDL style. This approach, typically involving an if-then-else statement within a process, clearly describes the selection logic and is widely adopted for such components due to its conciseness and direct representation of the desired behavior.[3]

VHDL Behavioral Code for 2-to-1 MUX

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2_1 is
    port (
        X, Y : in std_logic;
        S    : in std_logic;
        Z    : out std_logic
    );
end mux2_1;

architecture Behavioral of mux2_1 is
begin
```

```
    process (X, Y, S) -- Sensitivity list includes all inputs that can affect the output
    begin
        if (S = '0') then
            Z <= X;
        else
            Z <= Y;
        end if;
    end process;
end Behavioral;
```

B. 4-to-1 Multiplexer (MUX) using 2-to-1 MUX Components

A 4-to-1 MUX selects one of four data inputs (A, B, C, D) based on two select lines (S1, S0). This can be efficiently constructed using a hierarchical approach by instantiating three 2-to-1 MUX components: two in the first stage and one in the second stage.[3] The select line S0 controls the first stage of MUXes, while S1 controls the final MUX, which selects between the intermediate outputs of the first stage.

Truth Table for 4-to-1 MUX

Select (S1)	Select (S0)	Data Input (A)	Data Input (B)	Data Input (C)	Data Input (D)	Output (Z)
0	0	A	B	C	D	A
0	1	A	B	C	D	B
1	0	A	B	C	D	C
1	1	A	B	C	D	D

VHDL Component Instantiation and Port Mapping

In VHDL, incorporating one design entity into another traditionally involved a component declaration, which defined the interface of the sub-module before its instantiation.[1] However, with the introduction of VHDL-93, direct entity instantiation became the recommended practice. This method allows direct referencing of an entity using entity work.entity_name without the need for a redundant component declaration, unless integrating mixed-language designs (e.g., a Verilog module into VHDL).[4] This evolution in VHDL best practices significantly reduces maintenance

overhead, as changes to a sub-module's interface no longer require cascading updates across multiple declaration points, thereby improving the robustness and maintainability of large, hierarchical VHDL projects.

Port mapping is the process of connecting the input/output ports of a component instance to the actual signals in the main module. Two primary methods exist:

- **Positional Port Map:** Maps formal ports to actual signals based on their order in the port list.[1] While concise, this method can be prone to errors if the port order of the component changes.
- **Nominal Port Map:** Explicitly assigns formal parameters with actual parameters using the => operator (e.g., formal_port => actual_signal).[3] This method is generally preferred for its enhanced readability and robustness against changes in component port order. For instance, port map(A => x, B => y, S0 => s, m1 => z) clearly indicates which actual signal connects to which formal port.

Internal signals are crucial for connecting the outputs of the first stage of 2-to-1 MUXes to the inputs of the final MUX. For the 4-to-1 MUX, two internal signals (e.g., temp1, temp2 or m1, m2) are used to carry the intermediate selected data.[3]

Full VHDL Structural Code for 4-to-1 MUX (using mux2_1 component)

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux4_1 is
    port (
        A, B, C, D : in std_logic;   -- Four data inputs
        S0, S1     : in std_logic;   -- Two select inputs
        Z          : out std_logic   -- Single output
    );
end mux4_1;

architecture Structural of mux4_1 is
    -- Declare the 2-to-1 MUX as a component (for direct entity instantiation, this component declaration can often be omitted in modern VHDL if mux2_1 is compiled in 'work' library)
```

```
    component mux2_1
        port (X, Y : in std_logic;
              S    : in std_logic;
              Z    : out std_logic);
    end component;

    -- Internal signals to connect the outputs of the first stage MUXes to the inputs of the final MUX
    signal temp1, temp2 : std_logic;
begin
    -- First stage MUXes
    -- MUX U1: Selects between A and B based on S0, output to temp1
    U1_MUX2_1 : entity work.mux2_1 -- Direct entity instantiation
        port map (
            X => A,
            Y => B,
            S => S0,
            Z => temp1
        );

    -- MUX U2: Selects between C and D based on S0, output to temp2
    U2_MUX2_1 : entity work.mux2_1 -- Direct entity instantiation
        port map (
            X => C,
            Y => D,
            S => S0,
            Z => temp2
        );

    -- Second stage MUX
    -- MUX U3: Selects between temp1 and temp2 based on S1, output to Z
    U3_MUX2_1 : entity work.mux2_1 -- Direct entity instantiation
        port map (
            X => temp1,
            Y => temp2,
            S => S1,
            Z => Z
        );
end Structural;
```

The implementation of the 4-to-1 MUX using 2-to-1 MUXes exemplifies a common and effective "mixed-style" modeling approach in complex designs.[1] While the top-level 4-to-1 MUX is described structurally by connecting instances of the 2-to-1 MUX, the 2-to-1 MUX itself is concisely and clearly described behaviorally. This blend leverages the strengths of each style: behavioral for abstract functional description of simpler components and structural for clear interconnection and hierarchical organization of these components. This approach is a practical application of top-down design, where the functionality is defined at lower levels of the hierarchy, and the overall structure is defined at higher levels.

III. Designing the 5-Input Majority Circuit

A 5-input Majority circuit is a combinational logic circuit that produces a logic '1' output if three or more of its five inputs are simultaneously at a logic '1'.[13] This type of circuit is common in fault-tolerant systems or voting logic.

A. Logic Definition and Boolean Expression

For five inputs (A, B, C, D, E), the derivation of the minimal Sum-of-Products (SOP) Boolean expression involves considering all combinations where three, four, or five inputs are high. Manually deriving this expression for five variables typically involves complex K-maps, sometimes referred to as "super K-maps" or "compressed K-maps," due to the total of 2^5 = 32 possible input combinations.[13] The process involves systematically identifying product terms that result in a '1' output and then minimizing them.

The resulting SOP expression for a 5-input majority circuit is:
Y = (A·B·C) + (A·B·D) + (A·C·D) + (B·C·D) + (A·B·E) + (A·D·E) + (A·C·E) + (B·C·E) + (C·D·E) + (B·D·E).13

Key Input Combinations for 5-Input Majority Circuit

A	B	C	D	E	Output (Y)	(Number of High Inputs)
0	0	0	0	0	0	0
0	0	1	0	1	0	2
0	1	1	0	1	1	3

1	1	1	0	0	1	3
1	1	1	1	0	1	4
1	1	1	1	1	1	5

The derivation of the Boolean expression for a 5-input majority circuit highlights the practical limitations of manual logic minimization techniques like K-maps for circuits with a large number of inputs. While theoretically feasible, a 5-variable K-map is cumbersome, and for even more inputs, it becomes intractable. This underscores a fundamental advantage of Hardware Description Languages (HDLs) like VHDL. Instead of manually minimizing and drawing gate-level schematics, designers can describe the circuit's behavior or dataflow—for instance, by directly expressing the SOP equation—and rely on sophisticated synthesis tools to perform the complex logic minimization and gate mapping. This level of abstraction is crucial for designing modern, high-complexity digital systems efficiently.

B. VHDL Implementation

Given the complexity and numerous product terms in the derived SOP expression, a direct structural implementation using individual AND and OR gates would be highly verbose, difficult to read, and prone to errors. Therefore, a **dataflow** or **behavioral** style is most appropriate for implementing this circuit in VHDL.[1] These styles allow for the direct translation of the Boolean expression into VHDL using logical operators (and, or), letting the synthesis tool handle the underlying gate-level realization.

The VHDL code will utilize a concurrent signal assignment statement (dataflow style) to implement the SOP expression. This approach is concise and directly reflects the combinational nature of the circuit.

VHDL Dataflow Code for 5-Input Majority Circuit

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```vhdl
entity majority_of_five is
    port (
        A, B, C, D, E : in std_logic; -- Five inputs
        Y             : out std_logic  -- Single output
    );
end majority_of_five;

architecture Dataflow of majority_of_five is
begin
    -- Output Y is '1' if 3 or more inputs are '1'
    Y <= (A and B and C) or
         (A and B and D) or
         (A and C and D) or
         (B and C and D) or
         (A and B and E) or
         (A and D and E) or
         (A and C and E) or
         (B and C and E) or
         (C and D and E) or
         (B and D and E);
end Dataflow;
```

It is important to note that while the VHDL implementation for the majority circuit uses a dataflow style (a direct expression), the synthesis tool will implicitly create a structural implementation using logic gates. This illustrates a critical concept in hardware design: VHDL allows designers to describe hardware at a higher level of abstraction, but the ultimate output of the synthesis process is a netlist of interconnected gates, which is a structural representation. The designer provides the functional description, and the sophisticated electronic design automation (EDA) tools handle the low-level structural realization. This demonstrates the power of modern EDA tools in bridging the gap between high-level design intent and the physical hardware implementation.

IV. VHDL Coding Best Practices for Synthesizable Designs

Effective VHDL coding extends beyond mere functional correctness; it encompasses practices that ensure synthesizable, readable, and maintainable designs. Adherence to these guidelines is crucial for successful digital circuit development, especially in complex projects.

A. General Structure and Modularity

A **top-down design** approach is highly recommended, advocating for the separation of behavioral RTL (for leaf-level logic inference and sub-blocks) and structural code (for higher-level blocks) into their respective architectures.[15] Mixing styles within a single descriptive file is generally discouraged to enhance clarity and improve synthesis results. This hierarchical organization ensures that the VHDL code represents a clear hierarchy of structural elements, leading to a well-defined design.

The **entity declaration** defines the external interface, or ports, of a design unit. Entities should possess unique, descriptive names, typically in lowercase with underscores to separate words.[15] The **architecture body** describes the internal implementation of an entity, which can be behavioral, dataflow, or structural. Architecture names often resemble their associated entity (e.g., entity_arch or entity_rtl).[15]

B. Port Mapping and Component Instantiation

For VHDL-only designs (VHDL-93 and later), **direct entity instantiation** using entity work.entity_name is the recommended practice. This approach eliminates the need for redundant component declarations, which were a source of maintenance issues in older VHDL versions.[4] However, **component declarations** remain necessary for specific use cases, such as when instantiating a Verilog module within a VHDL design, to provide interface information to the tools.[4]

Explicit port mapping (nominal mapping) is strongly preferred over positional mapping. Explicitly declaring port mappings using formal_port => actual_signal significantly improves readability and makes future modifications easier, as it is robust against changes in the component's port order.[3] Furthermore, adopting a **one-line-per-signal** convention for signal assignments and port mappings enhances readability, particularly for components with numerous ports.[15]

C. Signal and Variable Usage

Signals in VHDL represent physical wires or registers in the hardware. They hold current and future values and can be assigned concurrently (outside a process) or sequentially (inside a process).[16] Key assignment rules for signals include: ideally, a signal should be assigned within only one process. In unclocked (combinatorial) processes, multiple assignments to the same signal should be avoided. In clocked processes, assignments should be confined to reset or clock edge statements.[15] The **sensitivity list** of a combinatorial process must include all signals that are read within that process to ensure correct behavior. For clocked processes, typically only the

clock and reset signals are required.[15] A general guideline is to use **active high** signals whenever possible, as this improves consistency and understanding.[15]

Variables, unlike signals, are local to a process or function. Their values change immediately upon assignment, and they are not necessarily synthesized as physical wires.[15] Variable assignments are order-dependent, similar to sequential programming languages. VHDL is a **strongly typed language**, meaning explicit type casting is often required when converting between different data types (e.g., using functions from ieee.numeric_std for conversions between std_logic_vector and integer).[15]

D. Naming Conventions and Comments

Consistent **naming conventions** are paramount for fostering design reuse and making code easier to read, understand, and debug for all designers involved in a project. Names for entities, architectures, processes, functions, and signals should be descriptive, lowercase, and use underscores to separate words.[15] It is advisable to avoid _in and _out suffixes for port names to prevent confusion, instead using _i for internal signals that represent ports (e.g., output_i for an output port).[15] Specific suffixes like _addr for addresses and _clk for clocks also contribute to clarity.[15]

Comments are mandatory for maintaining reusable code, as VHDL syntax alone cannot fully convey the design rationale behind signal assignments. **Comment headers** for entities, processes, and functions should clearly describe their responsibility and behavior.[15] Within processes and functions, comments for key statements are important to explain the reasoning, though not every single line requires a comment.[15]

E. Readability and Robustness

Consistent indentation is crucial for code readability and maintainability. Code blocks delineated by begin/end keywords should be properly indented.[15] When using **conditional statements** (if-then-else or case), designers should avoid deep nesting and strive to create exclusive cases. Crucially, behavior for all possible conditions must be explicitly defined (using else or when others) to prevent unintended latch inference or unpredictable synthesis results.[15]

The emphasis on these best practices highlights that VHDL is not merely for simulation; its primary role in digital design is hardware synthesis. Many guidelines, such as explicit sensitivity lists, avoiding multiple assignments to the same signal, and fully defining conditional cases, are directly driven by the need to produce predictable and efficient hardware. Violations can lead to inferred latches (undesirable memory

elements in combinatorial logic), non-synthesizable code, or inefficient gate structures. This underscores that VHDL coding fundamentally differs from software programming; every line implies a hardware structure, and understanding this implication is paramount for a digital design engineer.

Furthermore, many VHDL best practices, particularly those concerning naming conventions, commenting, and indentation, are not aimed at improving synthesis results but at enhancing the human readability and maintainability of the code. In complex projects, multiple engineers will interact with and maintain the same VHDL codebase over extended periods. Consistent style, clear comments, and descriptive names drastically reduce the cognitive load, accelerate debugging efforts, and facilitate design reuse. Poorly documented or inconsistently styled VHDL can be as detrimental as functionally incorrect code in a collaborative engineering environment, emphasizing that effective hardware design is as much about collaborative engineering as it is about technical correctness.

Conclusion

This report has explored various facets of digital circuit design using VHDL, demonstrating the versatility and power of this Hardware Description Language. The journey began with the fundamental concept of universal gates, specifically the NAND gate, illustrating how all basic logic functions (NOT, AND, OR, XOR) can be constructed solely from NAND gates through structural modeling and precise port mapping. This exercise underscored the theoretical completeness of universal gates while also highlighting the practical trade-offs in terms of gate count and propagation delay in real-world implementations.

The report then progressed to the design of multiplexers, showcasing a hierarchical approach where a 4-to-1 MUX was built by instantiating and interconnecting smaller 2-to-1 MUX components. This section emphasized the critical role of structural modeling and component instantiation in managing design complexity and promoting reusability. The discussion also covered the evolution of VHDL instantiation practices, from traditional component declarations to the more efficient direct entity instantiation, underscoring advancements aimed at improving design maintainability. The practicality of mixed-style modeling, where lower-level components are described behaviorally and integrated structurally, was also demonstrated.

Finally, the design of a 5-input majority circuit highlighted the advantages of VHDL in handling complex combinational logic. The derivation of its Boolean expression illustrated the scalability challenges of manual logic minimization for high-input

circuits, thereby reinforcing the necessity of HDLs and sophisticated synthesis tools that can translate high-level descriptions into optimized gate-level structures.

In summary, proficiency in VHDL, coupled with a deep understanding of underlying digital logic principles and adherence to established coding best practices, is essential for designing robust, efficient, and maintainable hardware systems. The principles of modularity, reusability, and hierarchical design are not merely theoretical constructs but fundamental strategies that enable engineers to tackle the increasing complexity of modern digital circuits, from basic gates to intricate systems.

Works cited

1. VHDL Modelling Styles: Behavioral, Dataflow, Structural – Buzztech, accessed on June 1, 2025, https://buzztech.in/vhdl-modelling-styles-behavioral-dataflow-structural/
2. Understanding VHDL - Digilent Reference, accessed on June 1, 2025, https://digilent.com/reference/programmable-logic/guides/getting-started-with-vhdl
3. VHDL Component and Port Map Tutorial - Invent Logics, accessed on June 1, 2025, https://allaboutfpga.com/vhdl-component-port-map-tutorial/
4. VHDL Component Declaration? - Hardware Coder, accessed on June 1, 2025, https://hardwarecoder.com/qa/190/vhdl-component-declaration
5. NOT, AND, OR Gates Using NAND Gates : 4 Steps (with Pictures ..., accessed on June 1, 2025, https://www.instructables.com/NOT-AND-OR-Gates-From-NAND-Gates/
6. Implementation of OR Gate from NAND Gate - GeeksforGeeks, accessed on June 1, 2025, https://www.geeksforgeeks.org/implementation-of-or-gate-from-nand-gate/
7. NAND logic - Wikipedia, accessed on June 1, 2025, https://en.wikipedia.org/wiki/NAND_logic
8. Realization of logic functions with the help of universal gates NAND and NOR Gate, accessed on June 1, 2025, https://de-iitr.vlabs.ac.in/exp/realization-of-logic-functions/theory.html
9. XOR gate - Wikipedia, accessed on June 1, 2025, https://en.wikipedia.org/wiki/XOR_gate
10. Creating XOR Gate Using NAND Gate - Shiksha Online, accessed on June 1, 2025, https://www.shiksha.com/online-courses/articles/xor-gate-using-nand-gate/
11. VHDL 4 to 1 MUX (Multiplexer) - Invent Logics, accessed on June 1, 2025, https://allaboutfpga.com/vhdl-4-to-1-mux-multiplexer/
12. vhdl-tutorial/structural/mux4x1.vhd at main - GitHub, accessed on June 1, 2025, https://github.com/ARC-Lab-UF/vhdl-tutorial/blob/main/structural/mux4x1.vhd
13. Project 3: Majority of Five - Digilent Reference, accessed on June 1, 2025, https://digilent.com/reference/learn/courses/digital-projects/simple-combo-circuit-design/start

14. Project 3.1: Majority of 5 : 19 Steps - Instructables, accessed on June 1, 2025, https://www.instructables.com/Project-31-Majority-of-5/
15. VHDL Coding Style Guidelines, accessed on June 1, 2025, https://webdocs.cs.ualberta.ca/~amaral/courses/329/labs/VHDL_Guideline.html
16. VHDL – Very High Speed Integrated Circuit Hardware Description ..., accessed on June 1, 2025, https://www.geeksforgeeks.org/vhdl-very-high-speed-integrated-circuit-hardware-description-language/