VLSI (VERY LARGE SCALE INTEGRATON)

A report submitted in partial fulfillment of the
requirement for the award of the degree of

Bachelor of Technology
in
Electronics And Communication Engineering

by

PULIPAKA VENKATA JAGADEESH KUMAR (Y22EC141)

Department of Electronics & Communication Engineering
R.V.R. & J.C. COLLEGE OF ENGINEERING
(Autonomous)
Approved by AICTE :: Affiliated to Acharya Nagarjuna University
Chowdavaram, Guntur - 522019, Andhra Pradesh, India

2024

Department of
Electronics And Communication Engineering

CERTIFICATE

This is to certify that the report of **EC353 Summer Internship** entitled **"VLSI (VERY LARGE SCALE INTEGRATON)"** that is being submitted by **"PULIPAKA VENKATA JAGADEESH KUMAR (Y22EC141), "** in partial fulfillment for the award of the Degree of **Bachelor of Technology** in **Electronics And Communication Engineering** to the R.V.R. & J.C. College of Engineering is a record of bonafide work carried out by him under my supervision.

Date:

Signature of Coordinator

Dr.D. Eswara Chaitanya M.Tech., Ph.D

Associate Professor in ECE

Signature of HOD

Dr.T.Ranga Babu M.Tech., Ph.D

Professor & Head

SkillDzire

CERTIFICATE OF INTERNSHIP

This is to Certify that Mr./Ms

Pulipaka Venkata Jagadeesh Kumar

Enrolled in the **Electronics and Communication Engineering - Y22EC141**

From College **R.V.R. & J.C.College of Engineering**

of university **Acharya Nagarjuna University**

has Successfully Completed short-term Internship programme titled

VLSI

under SkillDzire for 2 Months.Organized By **SkillDzire** in collaboration with **Andhra Pradesh State Council of Higher Education**.

Certificate ID:
SDST-09848

Issued On:
1-Jul-2024

Approved By AICTE

Authorized Signature

Abstract

Since I have a keen interest in knowing new things especially related to the area of VLSI, I selected a topic related to this field.

As I want to enhance my career in VLSI design, which is also my core subject, the research made by me to complete this internship will prove to be very helpful.

The main objective of my internship on the topic “RTL DESIGN, VERILOG AND FPGA programming” is to study the depth knowledge about the behavior and designing of different digital circuits. With the use of XILINX software, designing practical electronic devices has become much easier than before.

I would like to clearly mention that, my internship purely involves the basic concepts of RTL DESIGN and their designing using XILINX

I express my sincere thanks to Dr. T. Ranga Babu sir, Head of the Department of Electronics and Communication Engineering, R.V.R J.C College of Engineering, Guntur for his encouragement and support to carry out this summer internship successfully.

Table of contents

1

INTRODUCTION TO VLSI

Very-large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. VLSI began in the 1970s when complex semiconductor and communication technologies were being developed. The microprocessor is a VLSI device. Before the introduction of VLSI technology, most ICs had a limited set of functions they could perform. An electronic circuit might consist of a CPU, ROM, RAM, and other glue logic. VLSI lets IC designers add all of these into one chip. DEVELOPMENTS :

The first semiconductor chips held two transistors each. Subsequent advances added more transistors, and as a consequence, more individual functions or systems were integrated over time. The first integrated circuits held only a few devices, perhaps as many as ten diodes, transistors, resistors, and capacitors, making it possible to fabricate one or more logic gates on a single device. Now known retrospectively as small-scale integration (SSI), improvements in technique led to devices with hundreds of logic gates, known as medium-scale integration (MSI). Further improvements led to large-scale integration (LSI), i.e., systems with at least a thousand logic gates. Current technology has moved far past this

mark, and today's microprocessors have many millions of gates and billions of individual transistors.

At one time, there was an effort to name and calibrate various levels of large-scale integration above VLSI. Terms like ultra-large-scale integration (ULSI) were used. But the huge number of gates and transistors available on common devices has rendered such fine distinctions moot. Terms suggesting greater than VLSI levels of integration are no longer in widespread use.

As of early 2008, billion-transistor processors are commercially available. This became more commonplace as semiconductor fabrication advanced from the then-current generation of 65 nm processes. Current designs, unlike the earliest devices, use extensive design automation and automated logic synthesis to lay out the transistors, enabling higher levels of complexity in the resulting logic functionality. Certain high-performance logic blocks, like the SRAM (static random-access memory) cell, are still designed by hand to ensure the highest efficiency.

1.1 HARDWARE DESCRIPTION LANGUAGE (HDL)

In electronics, a hardware description language (HDL) is a specialized computer language used to program the structure, design, and operation of electronic circuits, and most commonly, digital logic circuits.

A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis, simulation, and simulated testing of an electronic circuit. It also allows for the compilation of an HDL program into a lower-level specification of physical electronic components, such as the set of masks used to create an integrated circuit.

A hardware description language looks much like a programming language such as C; it is a textual description consisting of expressions, statements, and control structures. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time. HDLs form an integral part of electronic design automation (EDA) systems, especially for complex circuits, such as microprocessors.

1.2 INTRODUCTION TO VERILOG

The Verilog HDL is an IEEE standard - number 1364. The first version of the IEEE standard for Verilog was published in 1995. A revised version was published in 2001; this is the version used by most Verilog users. The IEEE Verilog standard document is known as the Language Reference Manual (LRM). This is the complete authoritative definition of the Verilog HDL.

A further revision of the Verilog standard was published in 2005, though it has little extra compared to the 2001 standard. System Verilog is a huge set of extensions to Verilog, and was first published as an IEEE standard in 2005. See the appropriate section for more details about System Verilog.

IEEE Std 1364 also defines the Programming Language Interface (PLI). This is a collection of software routines which permit a bidirectional interface between Verilog and other languages (usually C).

Note that VHDL is not an abbreviation for Verilog HDL—Verilog and VHDL are two different HDLs. They have more similarities than differences, however.

Verilog was started initially as a proprietary hardware modeling language by Gateway Design Automation Inc. around 1984. It is rumored that the original language was designed by taking features from the most popular HDL language of the time, called Hilo, as well as from traditional computer languages such as C. At that time, Verilog was not standardized and the language modified itself in almost all the revisions that came out between 1984 to 1990. Verilog simulators were first used beginning in 1985 and were extended substantially through 1987. The implementation was the Verilog simulator sold by Gateway.

The first major extension was Verilog-XL, which added a few features and implemented the infamous ”XL algorithm” which was a very efficient method for doing gate-level simulation. This was around late 1990. Cadence Design Systems, whose primary product at that time included thin-film process simulators, decided to acquire Gateway Automation Systems. Along with other Gateway products, Cadence became the owner of the Verilog language and continued to market it

1.3 VHDL/VERILOG compared and contrasted

This section compares and contrasts individual aspects of the two languages; they are listed in alphabetical order.

Capability

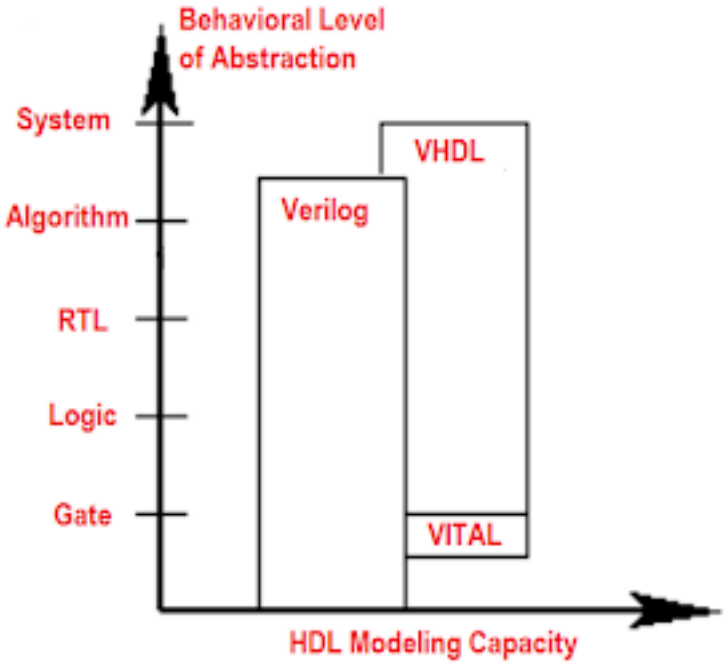
Hardware structure can be modeled equally effectively in both VHDL and Verilog. When modeling abstract hardware, the capability of VHDL can sometimes only be achieved in Verilog when using the PLI. The choice of which to use is not therefore based solely on technical capability but on:

personal preferences

EDA tool availability

commercial, business and marketing issues

The modeling constructs of VHDL and Verilog cover a slightly different spectrum across the levels of behavioral abstraction; see below figure.



1.4 COMPILATION

VHDL. Multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired. However, it is good design practice to keep each

design unit in its own system file in which case separate compilation should not be an issue.

Verilog. The Verilog language is still rooted in its native interpretative mode. Compilation is a means of speeding up simulation, but has not changed the original nature of the language. As a result care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation.

1.5 Data types

VHDL. A multitude of language or user defined data types can be used. This may mean dedicated conversion functions are needed to convert objects from one type to another. The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types. This will make models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code. VHDL may be preferred because it allows a multitude of language or user defined data types to be used.

Verilog. Compared to VHDL, Verilog data types are very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user. There are net data types, for example wire, and a register data type called reg. A model with a signal whose type is one of the net data types has a corresponding electrical wire in the implied modeled circuit. Objects, that is signals, of type reg hold their value over simulation delta cycles and should not be confused with the modeling of a hardware register. Verilog may be preferred because of its simplicity.

1.6 DESIGN REUSABILITY

VHDL. Procedures and functions may be placed in a package so that they are available to any design-unit that wishes to use them.

Verilog. There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module. To make functions and procedures generally

accessible from different module statements the functions and procedures must be placed in a separate system file and included using the include compiler directive.

1.7 EASIEST TO LEARN

Starting with zero knowledge of either language, Verilog is probably the easiest to grasp and understand. This assumes the Verilog compiler directive language for simulation and the PLI language is not included. If these languages are included they can be looked upon as two additional languages that need to be learned. VHDL may seem less intuitive at first for two primary reasons. First, it is very strongly typed; a feature that makes it robust and powerful for the advanced user after a longer learning phase. Second, there are many ways to model the same circuit, especially those with large hierarchical structures.

2

REGISTER-TRANSFER LEVEL (RTL) and FPGA

2.1 REGISTER-TRANSFER LEVEL (RTL)

Register-Transfer Level (RTL) is a design abstraction used in digital circuit design, which describes a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers and the logical operations performed on those signals.

2.1.1 RTL DESIGN

1. **RTL Code:**

   Written in HDLs like Verilog or VHDL.

   Defines the behavior of the digital system at the level of registers and the data transfers between them.

2. **Components of RTL Design:**

   Registers: Storage elements that hold the state of the digital system.

   Combinational Logic: Performs operations on the data stored in the registers.

   Clock Signals: Control the timing of data transfers between registers.

3. **SYNCHRONOUS DESIGN:**

   All operations are synchronized with a clock signal.

   Ensures predictable timing and behavior.

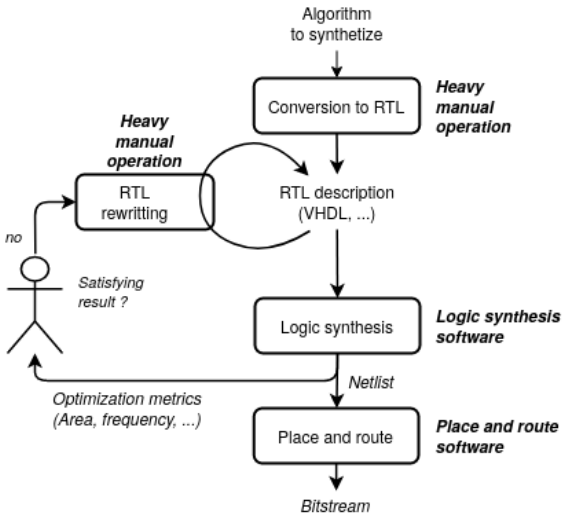
4. **STATE MACHINES:**

   Used to model sequential logic.

   Consists of states, transitions, and actions.

2.2 RTL DESIGN FLOW

1. Design Entry: Writing the RTL code using Verilog or VHDL.
2. Simulation: Testing the design to ensure it behaves as expected.
3. Synthesis: Converting the RTL code into a gate-level netlist.
4. Place and Route: Mapping the netlist onto the FPGA's physical resources.
5. Programming: Loading the design onto the FPGA.
6. Validation: Verifying the design in hardware.



2.3 FPGA BASICS

FPGAs (Field Programmable Gate Arrays) are integrated circuits that can be programmed to perform a wide range of digital functions1 . Here are some key concepts:

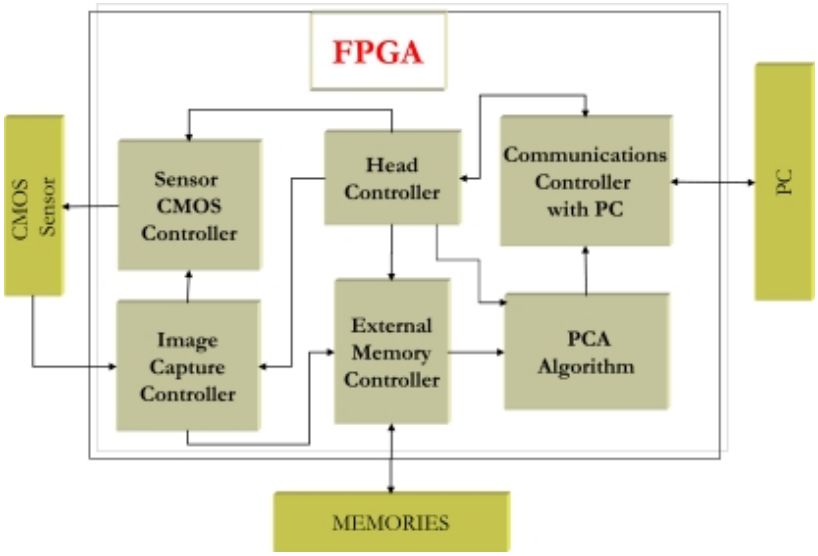
1. **Architecture:** FPGAs consist of an array of configurable logic blocks (CLBs) connected via programmable interconnects2 . CLBs include logic elements like Look-Up Tables (LUTs), flip-flops, and multiplexers2

2. **Reprogrammability:** Unlike ASICs (Application-Specific Integrated Circuits), FPGAs can be reprogrammed in the field to accommodate design changes or new applications1 .

3. **Applications:** FPGAs are used in various applications, including digital signal processing, communication systems, and rapid prototyping

4. **Programming:** FPGAs are programmed using HDLs like Verilog or VHDL, and tools like Xilinx Vivado or Intel Quartus.

   Advantages: FPGAs offer flexibility, high performance, and the ability to reconfigure for different tasks



2.4 KEY COMPONENTS OF A FPGA

1. **Configurable Logic Blocks (CLBs):** These are the basic building blocks of an FPGA, consisting of logic elements like Look-Up Tables (LUTs), flip-flops, and multiplexers2 .
2. **Programmable Interconnects:** These allow the CLBs to be connected in various ways to implement different logic functions1 .
3. **I/O Blocks:** These provide the interface between the FPGA and the outside world, allowing it to communicate with other devices

2.5 ADVANTAGES OF FPGA

1. **Flexibility:** FPGAs can be reprogrammed to perform different tasks, making them adaptable to changing requirements.
2. **Performance:** They can execute complex computations in parallel, leading to high performance.
3. **Rapid Prototyping:** FPGAs are ideal for testing and developing new designs quickly.

2.6 APPLICATIONS OF FPGA

1. **Digital Signal Processing (DSP):** Used in audio, video, and communication systems.
2. **Embedded Systems:** Used in automotive, aerospace, and industrial control systems.
3. **Data Centers:** Used for high-performance computing tasks.
4. **Networking:** Used in routers, switches, and other networking equipment.

2.7 PROGRAMMING FPGAs

FPGAs are typically programmed using Hardware Description Languages (HDLs) like Verilog or VHDL . Tools like Xilinx Vivado or Intel Quartus are used to design, simulate, and

program the FPGA.

Programming an FPGA is an engaging process that allows you to customize the hardware to perform specific tasks. Here's a rundown of the key steps involved:

1. **1. Design Specification** Define what you want your FPGA to do. This could be a simple task like blinking an LED, or something more complex like a digital signal processor.

2. **2. Design Entry** HDL Coding: Write your design using a Hardware Description Language (HDL) like Verilog or VHDL. This is where you describe the functionality of your circuit.

3. **3. Simulation** Simulate your design to verify that it behaves as expected. This helps you catch and fix any errors before moving to hardware.

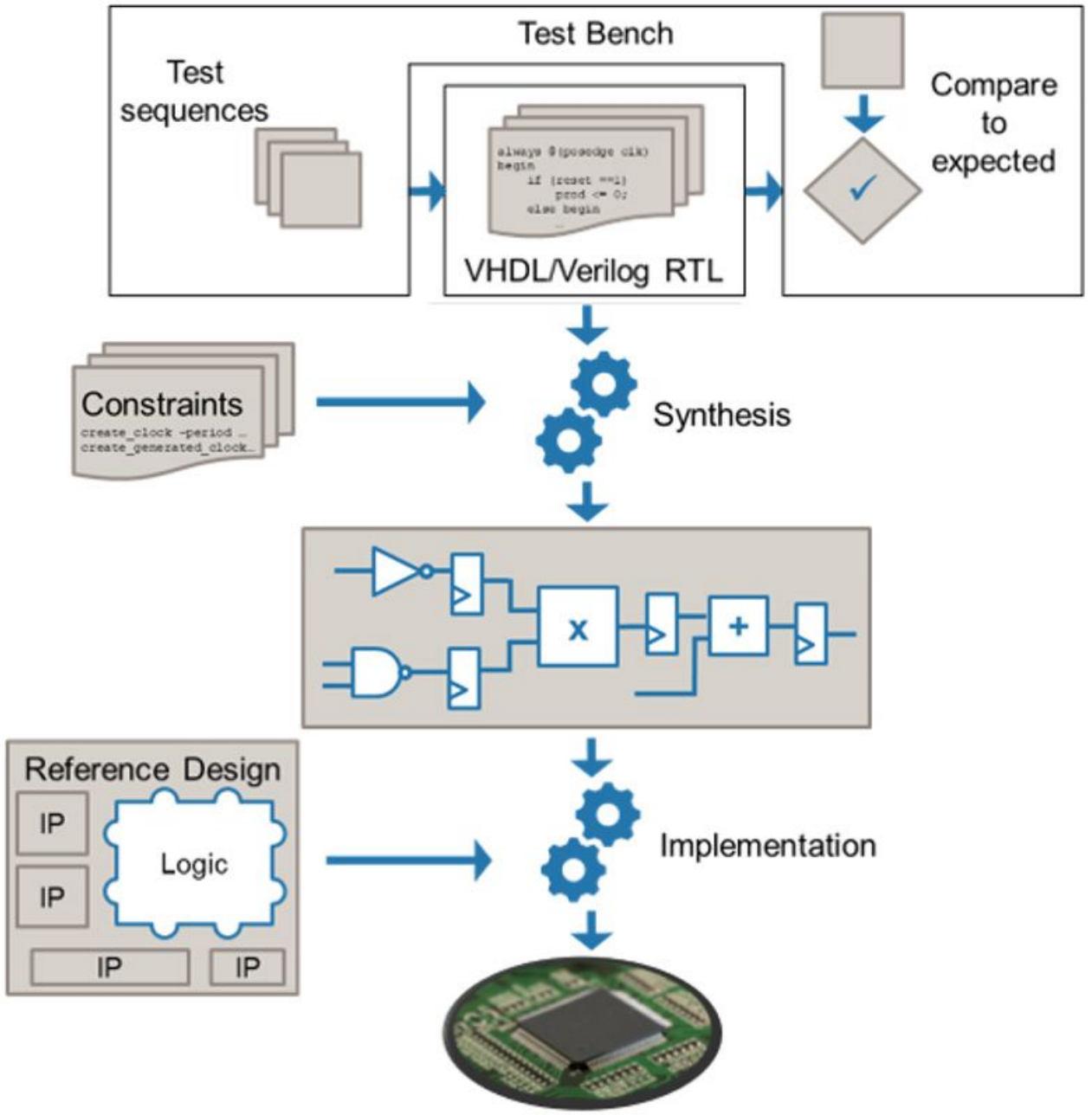
4. **4. Synthesis** Convert your HDL code into a gate-level netlist using synthesis tools provided by FPGA vendors like Xilinx Vivado. This process translates your high-level design into a hardware implementation.

5. **5. Implementation** Place and Route: This step involves placing the synthesized logic onto the FPGA and routing the connections between them. The tools will optimize the design for performance and resource usage.

   Bitstream Generation: Create the bitstream file that will be used to program the FPGA.

6. **6. Programming** Load the bitstream file onto the FPGA using programming tools like Xilinx Vivado. This step configures the FPGA to perform your specified task.

7. **7. Testing and Validation** Test your FPGA in the real world to ensure it performs as expected. Debug any issues that arise and iterate on your design if necessary.


Test Bench
Test sequences
always @(posedge clk)
begin
if (reset ==1)
prod <= 0;
else begin
VHDL/Verilog RTL
Compare to expected
Constraints
create_clock -period ...
create_generated_clock...
Synthesis
x
+
Reference Design
IP
IP
Logic
IP
IP
Implementation

3

XILINX VIVADO

3.1 XILINX

Xilinx, Inc. was an American technology and semiconductor company that primarily supplied programmable logic devices. The company is renowned for inventing the first commercially viable field-programmable gate array (FPGA). It also pioneered the first fabless manufacturing model. Xilinx was founded in Silicon Valley in 1984 and is headquartered in San Jose, United States. The company also has offices in Longmont, United States; Dublin, Ireland; Singapore; Hyderabad, India; Beijing, China; Shanghai, China; Brisbane, Australia, Tokyo, Japan and Yerevan, Armenia.[12][13]

According to Bill Carter, former CTO and current[when?] fellow at Xilinx, the choice of the name Xilinx refers to the chemical symbol for silicon Si. The ”linx” represents programmable links that connect programmable logic blocks together. The ’X’s at each end represent the programmable logic blocks.

Xilinx sold a broad range of field programmable gate arrays (FPGAs), and complex

programmable logic devices (CPLDs), design tools, intellectual property, and reference designs. Xilinx customers represent just over half of the entire programmable logic market, at 51 percent. Altera (now subsidiary of Intel) is Xilinx's strongest competitor with 34 percent of the market. Other key players in this market are Actel (now subsidiary of Microsemi) and Lattice Semiconductor

Xilinx was co-founded by Ross Freeman, Bernard Vonderschmitt, and James V Barnett II in 1984. The company went public on the Nasdaq in 1990.[7][8] In October 2020, AMD announced its acquisition of Xilinx, which was completed on February 14, 2022, through an all-stock transaction valued at approximately 60 billion.[9][10] Xilinx remained a wholly owned subsidiary of AMD until the brand was phased out in June 2023, with Xilinx's product lines now branded under AMD.

3.2 VIVADO

Vivado is Xilinx's design suite for FPGAs and SoCs, providing an integrated workflow for creating digital designs. It supports multiple design entry methods including Verilog, VHDL, and high-level synthesis languages. The suite includes powerful tools for synthesis, place-and-route, and bitstream generation.

Vivado Design Suite is a software suite for synthesis and analysis of hardware description language (HDL) designs, superseding Xilinx ISE with additional features for system on a chip development and high-level synthesis. Vivado represents a ground-up rewrite and

re-thinking of the entire design flow (compared to ISE).

Like the later versions of ISE, Vivado includes the in-built logic simulator.[11] Vivado also introduces high-level synthesis, with a toolchain that converts C code into programmable logic.

Replacing the 15 year old ISE with Vivado Design Suite took 1000 man-years and cost US 200 million.

3.3 KEY FEATURES

Vivado was introduced in April 2012, and is an integrated design environment (IDE) with system-to-IC level tools built on a shared scalable data model and a common debug environment. Vivado includes electronic system level (ESL) design tools for synthesizing and verifying C-based algorithmic IP; standards based packaging of both algorithmic and RTL IP for reuse; standards based IP stitching and systems integration of all types of system building blocks; and the verification of blocks and systems. A free version WebPACK Edition of Vivado provides designers with a limited version of the design environment.

1. Design Entry: Flexible entry methods to suit various design styles.
2. Simulation: Robust simulation tools to verify functionality.
3. Synthesis: Converts high-level code to gate-level netlists.
4. Place and Route: Optimizes resource allocation and performance.
5. IP Integration: Easy integration of pre-verified IP blocks.
6. Power Analysis: Tools to estimate and optimize power consumption.
7. Dynamic Function Exchange: Supports partial reconfiguration for flexibility.
8. Verification: Comprehensive verification tools for design validation.
9. Documentation and Support: Extensive resources to assist designers

3.4 COMPONENTS

The Vivado High-Level Synthesis compiler enables C, C++ and SystemC programs to be directly targeted into Xilinx devices without the need to manually create RTL. Vivado HLS

is widely reviewed to increase developer productivity, and is confirmed to support C++ classes, templates, functions and operator overloading. Vivado 2014.1 introduced support for automatically converting OpenCL kernels to IP for Xilinx devices. OpenCL kernels are programs that execute across various CPU, GPU and FPGA platforms.

The Vivado Simulator is a component of the Vivado Design Suite. It is a compiled-language simulator that supports mixed-language, Tcl scripts, encrypted IP and enhanced verification.

The Vivado IP Integrator allows engineers to quickly integrate and configure IP from the large Xilinx IP library. The Integrator is also tuned for MathWorks Simulink designs built with Xilinx's System Generator and Vivado High-Level Synthesis.

The Vivado Tcl Store is a scripting system for developing add-ons to Vivado, and can be used to add and modify Vivado's capabilities.[19] Tcl is the scripting language on which Vivado itself is based. All of Vivado's underlying functions can be invoked and controlled via Tcl scripts.

4

REQUIREMENTS

4.1 SOFTWARE REQUIREMENTS

1. **Vivado Design Suite:** Ensure you have Vivado 2020.1 or later installed1 . This suite includes all necessary tools for design entry, synthesis, implementation, and verification .

2. **Operating System:** Vivado supports Windows and Linux2 . Linux is often preferred for better performance .

3. **Supported HDLs:** Familiarity with Verilog, VHDL, or SystemC for design entry .

4. **Simulation Tools:** Tools like ModelSim or Vivado Simulator for verifying your designs 1 .

5. **IP Cores:** Access to Xilinx IP cores if needed for your design

4.2 HARDWARE REQUIREMENTS

1. **Processor:** A powerful CPU, such as an AMD Ryzen Threadripper or Intel i9, is recommended for faster design processing .
2. **Memory:** At least 16GB of RAM, though 32GB or more is ideal for handling large designs .
3. **Storage:** Fast storage solutions like NVMe SSDs for quicker file access and project handling .
4. **Graphics Card:** A dedicated GPU can help with rendering and simulation tasks .
5. **FPGA Board:** A compatible Xilinx FPGA board, such as the Zynq-7000 series or Kintex-7 series, depending on your project requirements .
6. **JTAG/USB Cable:** For programming and debugging the FPGA board

5

PROJECT IMPLEMENTATION

5.1 MULTIFUNCTION RESIDUE ARCHITECTURES

Multifunction Residue Architectures (MRAs) are advanced digital systems that leverage the Residue Number System (RNS) for efficient computation. These architectures are particularly useful in applications requiring high-speed arithmetic operations, such as digital signal processing (DSP) and cryptography.

5.2 Key Concepts of Multifunction Residue Architectures

1. **Residue Number System (RNS):** RNS is a non-weighted number system that represents integers as a set of residues with respect to a set of pairwise coprime moduli. This allows for parallel processing of arithmetic operations, leading to high-speed computation.

2. **Modular Arithmetic:** Operations in RNS are performed modulo a set of chosen

moduli. This modular arithmetic simplifies complex operations and reduces the likelihood of overflow.

3. **Parallelism:** One of the main advantages of RNS is its inherent parallelism. Multiple operations can be performed simultaneously, significantly speeding up computation.

4. **Modular Reduction:** After performing operations in RNS, the results need to be converted back to the conventional number system. This process is known as modular reduction and is a critical part of MRA design.

5.3 Implementing MRAs in Vivado

1. **Design Entry:** Write the RTL code using Verilog or VHDL to define the RNS modules and functional units.

2. **Simulation:** Simulate the RTL code to verify its functionality and ensure correct implementation of RNS operations.

3. **Synthesis:** Use Vivado to synthesize the RTL code, converting it into a gate-level representation.

4. **Place and Route:** Map the synthesized design onto the FPGA's physical resources, optimizing for performance and resource utilization.

5. **Programming:** Load the generated bitstream onto the FPGA and test the entire system.

6

PROGRAM

Let's create a more detailed example for a multifunction residue architecture in Vivado using Verilog. This example will perform addition, subtraction, and multiplication using the Residue Number System (RNS).

6.1 TOP MODULE:

Integration: The top module integrates all the submodules, like the RNS core and shared memory. It acts as a wrapper that connects all the components together.

Interfacing: It provides a single interface for all inputs and outputs, simplifying the overall design and making it easier to manage and test.

Hierarchy Management: Having a top module helps in creating a clear design hierarchy, making it easier to understand and debug the system.

6.2 SHARED MEMORY MODULE :

The shared memory module acts as a common storage space that can be accessed by multiple cores or functional units. This is crucial in a multifunction system where different operations might need to read from or write to a shared resource.

6.3 RNS CODE MODULE

```
module RNS_Core(
    input wire [3:0] a,
    input wire [3:0] b,
    input wire [1:0] operation, // 00: addition, 01: subtraction, 10: multi
    output reg [3:0] result
);

// Operation encoding
localparam ADD = 2'b00;
localparam SUB = 2'b01;
localparam MUL = 2'b10;

always @(*) begin
    case (operation)
        ADD: result = (a + b) % 16; // Modulo for residue
        SUB: result = (a - b) % 16;
        MUL: result = (a * b) % 16;
        default: result = 4'b0000;
    endcase
end

endmodule
```

the RNS (Residue Number System) core module is used for performing arithmetic operations (addition, subtraction, and multiplication) on numbers represented in a residue number system. Here's a breakdown:

Residue Number System: RNS represents numbers as a set of residues with respect to a set of pairwise coprime moduli. This allows for parallel processing of arithmetic operations, making them faster and more efficient in certain applications.

Arithmetic Operations: The RNS core module handles three basic operations:

Addition: result = (a + b)

Subtraction: result = (a - b)

6.4 SHARED MEMORY MODULE

```
module SharedMemory(
    input wire clk,
    input wire [3:0] address,
    input wire [3:0] data_in,
    output reg [3:0] data_out,
    input wire write_enable
);

reg [3:0] memory [0:15]; // 16x4-bit memory

always @(posedge clk) begin
    if (write_enable)
        memory[address] ≤ data_in;
    else
        data_out ≤ memory[address];
end

endmodule
```

Data Storage: The shared memory module acts as a common storage space that can be accessed by multiple cores or functional units. This is crucial in a multifunction system where different operations might need to read from or write to a shared resource.

Data Synchronization: It ensures that data is consistently available to all parts of the system, helping manage the flow of data between different components.

Modularity: By separating memory management into its own module, the design becomes more modular and easier to modify or expand. For example, if you need to change the memory architecture, you can do so without altering the core logic.

These modules help keep the design organized, modular, and scalable, making it easier to implement and debug complex digital systems. Want to dive deeper into any part of the system?

6.5 TOP MODULE

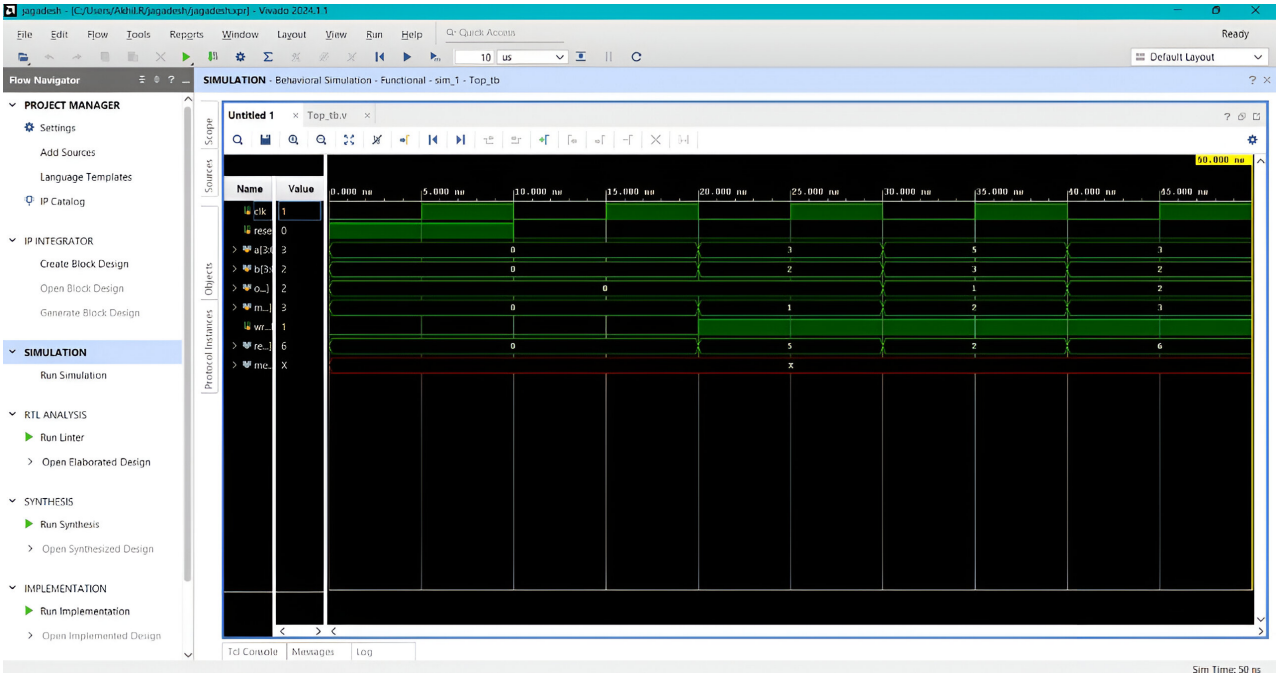
```
module Top(
    input wire clk,
    input wire reset,
    input wire [3:0] a,
    input wire [3:0] b,
    input wire [1:0] operation,
    input wire [3:0] mem_address,
    input wire write_enable,
    output wire [3:0] result,
    output wire [3:0] mem_data_out
);

// Instantiate RNS Core
RNS_Core rns_core (
    .a(a),
    .b(b),
    .operation(operation),
    .result(result)
);

// Instantiate Shared Memory
SharedMemory shared_memory (
    .clk(clk),
    .address(mem_address),
    .data_in(result),
    .data_out(mem_data_out),
    .write_enable(write_enable)
);

endmodule
```

6.6 TESTBENCH

```
module Top_tb;

    reg clk;
    reg reset;
    reg [3:0] a;
    reg [3:0] b;
    reg [1:0] operation;
    reg [3:0] mem_address;
    reg write_enable;
    wire [3:0] result;
    wire [3:0] mem_data_out;

    // Instantiate the Top module
    Top uut (
        .clk(clk),
        .reset(reset),
        .a(a),
        .b(b),
        .operation(operation),
        .mem_address(mem_address),
        .write_enable(write_enable),
        .result(result),
        .mem_data_out(mem_data_out)
    );

    initial begin
        // Initialize inputs
        clk = 0;
        reset = 1;
        a = 4'b0000;
        b = 4'b0000;
        operation = 2'b00;
        mem_address = 4'b0000;
        write_enable = 0;

        // Reset the design
        #10;
        reset = 0;
        #10;

        // Test addition
        a = 4'b0011;
        b = 4'b0010;
        operation = 2'b00; // Add
        write_enable = 1;
        mem_address = 4'b0001;
        #10;

        // Test subtraction
        a = 4'b0101;
        b = 4'b0011;
        operation = 2'b01; // Subtract
        write_enable = 1;
        mem_address = 4'b0010;
        #10;

        // Test multiplication
        a = 4'b0011;
        b = 4'b0010;
        operation = 2'b10; // Multiply
        write_enable = 1;
        mem_address = 4'b0011;
        #10;

        $finish;
    end

    always #5 clk = ~clk; // Clock generator

endmodule
```

7

RESULT/OUTPUT



it seems like the signals are changing values at expected intervals. To ensure this is the correct output for the provided code, let's break down what we should see:

1. **Clock (clk):** The clock signal should toggle at a regular interval. The typical fre-

quency for the clock in the testbench is set to change every 5 time units.

2. **Reset (reset):** This signal should initially be high (active) to reset the system and then transition to low (inactive) to allow normal operation.

3. **Inputs (a and b):** These should change values as defined in the testbench code, reflecting different test cases for arithmetic operations.

4. **Operation (operation):** This signal determines the arithmetic operation being performed:

   00 for addition.

   01 for subtraction.

   10 for multiplication.

5. **Result (result):** The output should reflect the result of the arithmetic operation performed by the RNS core. For example:

   When a = 3, b = 2, and operation = 00, result should be 5.

   When a = 5, b = 3, and operation = 01, result should be 2.

   When a = 3, b = 2, and operation = 10, result should be 6.

6. **Memory Address :** This should indicate the address in the shared memory where the result is stored.

7. **Write Enable :** This signal should be high when writing the result to shared memory.

8. **Memory Data Out:** This should show the value read from the shared memory address.

8

CONCLUSION

”On doing this internship on RTL design, Verilog, and FPGA programming, I came across many interesting facts and figures about electronic circuits. While doing simulation of these digital circuits, I realized the importance of software analysis before constructing our required devices. If we construct our devices before performing software analysis, it would be cumbersome and tough to use accurately measured devices.”

9

REFERENCE

---

1. **WIKIPEDIA:** For detailed knowledge of characteristics, advantages, disadvantages of various electronic devices and to know about xilinx and vivado.

2. **TO DOWNLOAD VIVADO** http://www.xilinx.com/

3. **LIST OF VLSI SAMPLE PROJECTS** 1000projects.org/projects/vlsi-projects