FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 3

Rechnerarchitektur

Ashwin Varkey

A SystemC-Based Implementation of an IEEE 754 Pipelined Floating-Point Unit with Zynq FPGA Verification

Master's Thesis in Computational Engineering

10. June 2025

Please cite as:
Ashwin Varkey, "A SystemC-Based Implementation of an IEEE 754 Pipelined Floating-Point Unit with Zynq FPGA Verification," Master's Thesis, University of Erlangen, Dept. of Computer Science, June 2025, University of Erlangen, Dept. of Computer Science, June 2025.

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Rechnerarchitektur

Martensstr. 3 · 91058 Erlangen · Germany

www3.cs.fau.de

A SystemC-Based Implementation of an IEEE 754 Pipelined Floating-Point Unit with Zynq FPGA Verification

Master's Thesis in Computational Engineering

vorgelegt von

Ashwin Varkey

geb. am 30. January 1998
in India

angefertigt am

Lehrstuhl für Informatik 3
Rechnerarchitektur

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Prof. Dr.-Ing. Dietmar Fey**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. Dietmar Fey**
Beginn der Arbeit:	**10. December 2024**
Abgabe der Arbeit:	**10. June 2025**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Ashwin Varkey)
Erlangen, 10. June 2025

Table of Contents

Abstract

Floating-point arithmetic units are a fundamental component in modern processor architectures, and RISC-V open standard approach to floating point extensions provides a simplified model for implementing IEEE 754 compliant operations in a pipeline structure. The IEEE-754 standard ensures consistency and computational accuracy defines the specific formats and operations that hardware implementations must follow for proper floating-point computation.

This research presents a floating-point processor that implements standard arithmetic units-adder, multiplier, subtractor and divider within a five-stage pipeline structure. The fetch and decode components for the floating-point unit were developed using the rv32f extensions of RISC-V Instruction Set Architecture (ISA). The initial stage includes SystemC modeling followed by Register Level Transfer (RTL) design which comes from synthesis and simulation before the development of a prototype that will be implemented in Field Programmable Gate Array (FPGA) board.

To ensure the design worked correctly, a custom testbench was created with a wide range of inputs and edge cases covering all the arithmetic limits. The floating-point units received improvements through instruction level simulation results. Detailed waveform analysis revealed instruction control behaviors and facilitated solving the execution anomalies. The RISC-V ISA simulator Spike served as a cross-verification tool to execute test benches that originated from the RISC-V GCC toolchain and was compared with our floating-point unit results.

This methodology allows quick development cycles without affecting performance providing a template that joins high level synthesis (HLS) with simulator based verification before hardware deployment.

Chapter 1

Introduction

System on Chip design complexity has surged requiring the need for advanced methodologies like high level synthesis (HLS) which facilitate faster design through high-level programming languages such as C/C++ and SystemC [1]. While HLS has proven effective for application-specific accelerators, its application to general-purpose processors has been limited due to challenges in handling control-dominated logic. Floating point units are important for processor architecture development enabling precise representation and manipulation of real numbers which advance computing performance [2].

The IEEE 754 standard is used to establish consistency and computational accuracy with specific formats and operations that hardware implementations must adhere to for reliable floating-point computations [3]. The design and implementation of custom-floating point units present several challenges, taking inspiration from RISC-V pipeline ensuring compliance with IEEE 754, managing pipeline hazards and optimizing for the FPGA board. Addressing these obstacles requires a comprehensive approach that spans high-level architectural modelling, detailed RTL implementation and thorough verification methodology.

As computational demands continue to increase particularly in embedded computing environments, the efficient hardware implementation of floating-point operations is important. While regular computer processors have built-in components for handling decimal calculations, they are designed to be all-purpose rather than specialized for specific tasks [4]. If you are working on applications with strict speed, power or resource requirements, custom floating-point designs on reconfigurable hardware like FPGAs can offer major benefits by optimizing the architecture and fine tuning the processing pipeline.

In programming terms, FPGA plays the role of personal computer while High Level Synthesis Tool acts like the C compiler. It gives access to the FPGA through high level language. Zynq XC7Z020 from Xilinx is a SOC containing several com-

ponents: processors, memories, USB and Ethernet interfaces whose programmable part contains 6650 Configurable Logic Blocks (CLB). [3]

The increasing use of heterogenous computing boards starts a new revolution for customised floating point processor with the gap between high level algorithm and hardware implementation being addressed sufficiently. [5]

1.1 Motivation and Problem Statement

Despite advances in processor architecture and digital design methodologies several challenges persist in efficient implementation of floating-point units [6]:

- Conventional floating-point implementation often struggles to achieve high accuracy, low latency and efficient resource utilization.
- The detailed implementation of IEEE 754 compliant operations, especially division and special case handling, introduces a complexity that must be carefully managed to ensure correctness.
- Transitioning from high-level programming languages to hardware description languages (HDLs) requires adapting to fundamentally different design paradigms and hardware-specific constructs.
- Comprehensive verification of floating-point operation requires sufficient test coverage across normal and special cases requiring proper verification methodologies.
- Implementing floating point units on specific FPGA platform introduces additional considerations like available pinouts, clock frequency limits and integration with existing board design.

1.2 Research Objectives

This research aims to achieve the following objectives:

- Design and implement a five-stage pipeline for IEEE-754 standard floating point operations including addition,subtraction,multiplication and division
- Develop a comprehensive SystemC model that allows us to explore different architectural options, analyze accuracy and functionality before moving to detailed RTL implementation phase
- Develop a practical verification that confirms computational accuracy across normal operations, special cases and boundary conditions for all floating-point units

- Implement floating point unit on Xilinx Zynq using Vivado with careful resource utilization and integration with existing system components

1.3 Development Environment and Tools

The floating-point unit implementation leveraged a comprehensive development environment spanning:

1.3.1 High-Level Modeling Environment

- **SystemC Library:** Version 2.3.3, providing transaction-level modeling constructs and simulation kernel
- **C++ Compiler:** GCC 9.3.0 with C++14 standard support for compilation and linking
- **Build System:** CMake 3.16, managing cross-platform build configuration and dependency resolution

1.3.2 RTL Design and Synthesis Tools

- **Xilinx Vivado 2019.2:** Primary tool for RTL synthesis, implementation, and FPGA targeting
- **Intel Compiler for SystemC (ICSC):** High-level synthesis tool that translates synthesizable SystemC code into SystemVerilog.

1.3.3 Verification and Analysis Tools

- **GTKWave 3.3.104:** Waveform visualization and signal analysis for debugging
- **RISC-V GNU Toolchain:** Cross-compiler and assembler for generating test programs
- **Spike RISC-V ISA Simulator:** Reference implementation for golden model comparison

1.3.4 Development Infrastructure

- **Documentation:** LaTeX with TeXstudio for technical documentation
- **Hardware Platform:** Xilinx Zynq XC7Z020 evaluation board for implementation validation

1.4 Literature Review

The literature review sets the theoretical foundation for this thesis topic and mentions concepts for floating point arithmetic, pipelined architecture, and FPGA implementation. The search uses peer reviewed journals, conference papers, technical standards (IEEE 754), and reference texts published in the past two decades. Academic databases like IEEE Xplore, ACM Digital Library, Science Direct, and Google Scholar, employing keywords related to floating-point hardware, pipelined processors, SystemC modeling, and FPGA implementation. Floating point representation allows computers to work with real numbers, addressing limitations of fixed arithmetic where a fractional is used for storing a certain fixed number of digits. The IEEE 754 standard was established in 1985 and revised twice in 2008 and 2019 to account for changes and improvements suggested by scholars.Recent work by Jaiswal and So examines implementation changes specific to floating point addition, proposes a method that minimizes the critical path through optimized normalization and rounding [7]. Similarly, Dinechin and Pasca address multiplication optimization algorithm through specialized significand multipliers and exception handling logic [8]. SystemC is a very important programming language for hardware modelling and simulation. Grotker et al. explains the SystemC modeling approach, describing its ability to represent hardware to the lowest level of digital circuits [9]. For floating point units, Reshadi et al.'s work describes both functional and timing behavior [10]. High level synthesis (HLS) has picked up a lot of significance in hardware design. Goossens studies the path from algorithmic to hardware implementation, helping developers to feature processor parts at high abstraction level [11]. HLS struggles with optimising peak performance and resource efficiency despite making designing easier. The solution Goossens suggests is manual debugging, which is a hybrid approach with HLS for fast prototyping and targets RTL level fine tuning [11]. The Xilinx Zynq platform gives significant advantages and challenges for floating point implementations with both programming and embedded logic being involved. Crocket et al. gives a proper description of Zynq architecture, describing communication mechanisms between processing system (PS) and programming logic (PL) parts [12]. Comprehensive verification of the floating point hardware will require a testbench of basic, special and boundary operations/testcases. Verma et al. give a simple way to floating point operations that cover underflow and overflow with an emphasis on randomised testing [13]. The literature review exposes several gaps and opportunities to look into floating point processor design and implementation. While a lot of research explains individual floating point designs in high level programming languages like Python and Verilog/VHDL, integrated methodologies that use SystemC modeling and then FPGA implementation remain limited.

Chapter 2

Architectural Design

2.1 Development Flow Stages

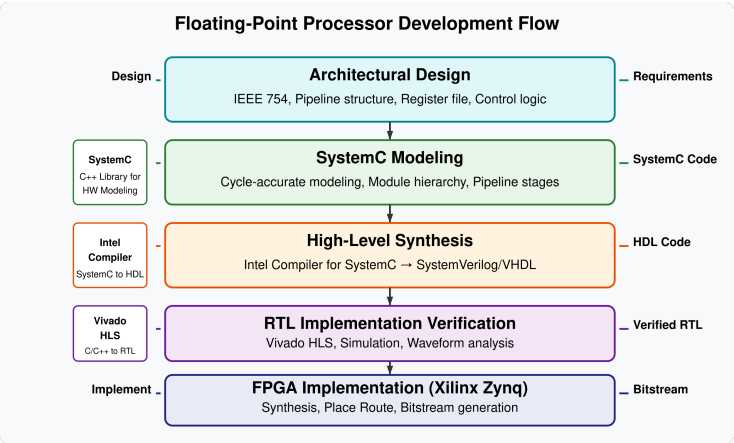

Figure 2.1 – Development flow stages of the pipelined floating-point processor

2.2 Design Requirements and Constraints

The floating-point processor design must satisfy the following functional requirements:

- Support for 32-bit single precision floating point format with 1 sign bit, 8 exponent bits, and 23 significand bits.
- Implementation of addition, subtraction, multiplication and division operations with full precision.

- Proper handling of special cases like zero, infinity, NaN and denormalised numbers.
- Round to nearest (ties for even) to be implemented.
- Handling exceptions like overflow, underflow, division by zero and invalid operations.

2.3 IEEE 754 Floating-Point Standard

The IEEE 754 standard, formally known as IEEE Standard for Floating-Point Arithmetic, defines the technical details for floating-point computation. First published in 1985 and revised in 2008 (IEEE 754-2008), it has become the de facto standard for floating-point arithmetic in modern computing systems. This section provides a comprehensive overview of the standard's key aspects that directly influence the architectural design decisions of our pipelined floating-point processor.

2.3.1 Standard Overview and Motivation

Prior to IEEE 754, different computer manufacturers implemented floating-point arithmetic with varying formats, precision levels, and behavioral characteristics. This lack of standardization created significant portability issues for scientific and engineering applications. The IEEE 754 standard addresses these challenges by establishing:

- Standardized binary and decimal floating-point formats
- Consistent rounding behaviors across different implementations
- Uniform handling of special values and exceptional conditions
- Predictable results for identical operations across different platforms

The standard ensures that floating-point operations produce identical results regardless of the underlying hardware implementation, provided both systems comply with IEEE 754 specifications.

2.3.2 Floating-Point Number Representation

IEEE 754 defines multiple precision formats, with single precision (binary32) being most relevant to our implementation. The mathematical representation of a floating-point number follows the formula:

$$(-1)^S \times (1 + M) \times 2^{E-bias}$$

Where:

- S is the sign bit
- M is the significand (mantissa) representing the fractional part
- E is the stored exponent value
- $bias$ is the exponent bias (127 for single precision)

2.3.2.1 Single Precision Format (Binary32)

The 32-bit single precision format allocates bits as follows:

- **Bit 31:** Sign bit (S)
- **Bits 30-23:** 8-bit exponent field (E)
- **Bits 22-0:** 23-bit significand field (M)

The biased exponent representation allows for efficient comparison operations and eliminates the need for separate sign handling in the exponent. The bias value of 127 means that actual exponents range from -126 to +127, with stored values from 1 to 254 (0 and 255 are reserved for special cases).

2.3.3 Special Values and Edge Cases

IEEE 754 defines specific encodings for special values that cannot be represented in the standard normalized format:

2.3.3.1 Zero Values

Zero is represented with both exponent and significand fields set to zero. The standard defines both positive zero (+0) and negative zero (-0), distinguished only by the sign bit. These values are considered equal in comparisons but may produce different results in certain operations (e.g., division by zero).

2.3.3.2 Denormalized Numbers (Subnormals)

When the exponent field is zero but the significand is non-zero, the value represents a denormalized number. These values use the formula:

$$(-1)^S \times (0 + M) \times 2^{1-bias}$$

Denormalized numbers provide gradual underflow, allowing representation of very small numbers that would otherwise underflow to zero. This feature maintains numerical stability in algorithms sensitive to small value precision.

2.3.3.3 Infinity Values

Infinity is represented with the exponent field set to all ones (255 for single precision) and the significand field set to zero. Like zero, infinity has both positive and negative variants. Infinity results from operations that exceed the representable range, such as division by zero or overflow conditions.

2.3.3.4 Not-a-Number (NaN)

NaN values have the exponent field set to all ones and a non-zero significand. The standard defines two types of NaN:

- **Quiet NaN (qNaN):** Propagates through operations without raising exceptions
- **Signaling NaN (sNaN):** Triggers an invalid operation exception when used in arithmetic operations

The most significant bit of the significand typically distinguishes between quiet and signaling NaN values.

2.3.4 Rounding Modes

IEEE 754 defines five rounding modes to handle cases where the exact result cannot be represented in the target format:

2.3.4.1 Round to Nearest, Ties to Even

This is the default rounding mode, implemented in our processor design. When the result falls exactly between two representable values, rounding occurs toward the value with an even least significant bit. This approach eliminates systematic bias in repeated calculations.

2.3.4.2 Round to Nearest, Ties Away from Zero

Similar to the previous mode, but ties are resolved by rounding away from zero rather than toward even values.

2.3.4.3 Round Toward Zero (Truncation)

Results are truncated toward zero, effectively discarding fractional portions. This mode never increases the magnitude of the result.

2.3.4.4 Round Toward Positive Infinity

Results are always rounded toward positive infinity, useful in interval arithmetic applications.

2.3.4.5 Round Toward Negative Infinity

Results are always rounded toward negative infinity, complementing the previous mode for interval arithmetic.

2.3.5 Exception Handling

IEEE 754 defines five types of floating-point exceptions that may occur during arithmetic operations:

2.3.5.1 Invalid Operation

Occurs when an operation has no mathematically defined result, such as:

- Square root of a negative number
- Addition of infinities with opposite signs
- Multiplication of zero by infinity

The result is typically a NaN value.

2.3.5.2 Division by Zero

Triggered when dividing a finite non-zero number by zero. The result is signed infinity.

2.3.5.3 Overflow

Occurs when the result magnitude exceeds the largest representable finite value. Depending on the rounding mode, the result may be infinity or the largest representable finite value.

2.3.5.4 Underflow

Happens when the result is too small to be represented as a normalized number. The result may be zero or a denormalized number, depending on the specific circumstances.

2.3.5.5 Inexact

Triggered when the result cannot be represented exactly and must be rounded. This is the most common exception in floating-point operations.

2.3.6 IEEE 754 Compliance Requirements

For our processor to be IEEE 754 compliant, it must implement:

- Correct handling of all special values (zero, infinity, NaN, denormalized numbers)
- Proper exception detection and reporting mechanisms
- At least one required rounding mode (round to nearest, ties to even)
- Consistent behavior for edge cases and boundary conditions
- Correct sign handling for all operations and special cases

The architectural decisions in subsequent sections are designed to meet these compliance requirements while maintaining efficient FPGA implementation characteristics.

2.4 Introduction to Architectural Design

This section describes the architectural design of a pipelined floating-point processor that balances IEEE 754 compliance with performance optimization and resource efficiency for Xilinx Zynq FPGA implementation. The design addresses pipeline hazards, ensures IEEE 754 compliant operations, and optimizes FPGA resources through critical components including exponent alignment, normalization, and rounding logic to achieve both accuracy and implementation efficiency [14].

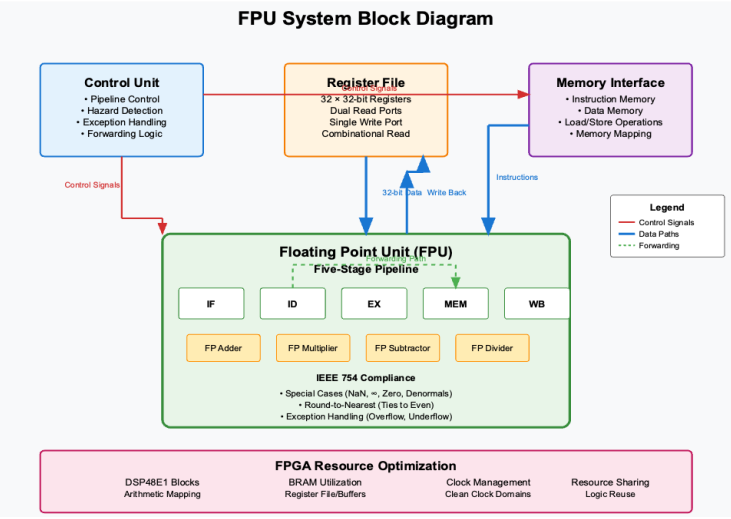

Figure 2.2 – Control logic architecture of the pipelined floating-point processor

2.4.1 Floating-Point Format and Representation

Building upon the IEEE 754 standard discussed in Section 2.3, the processor implements the single-precision binary floating-point format with the following architectural considerations:

- **Sign Bit:** 1 bit indicating value sign (0 for positive, 1 for negative)
- **Exponent:** 8 bits representing the biased exponent (actual exponent + 127)
- **Significand:** 23 bits representing the fractional portion of the significand (with implicit leading 1 for normalized values)

Special values are encoded according to IEEE 754 conventions as detailed in Section 2.3.3:

- **Zero:** Exponent $= 0$, Significand $= 0$ (signed)
- **Denormalized Numbers:** Exponent $= 0$, Significand $\neq 0$
- **Infinity:** Exponent $= 255$, Significand $= 0$ (signed)
- **NaN:** Exponent $= 255$, Significand $\neq 0$

2.4.2 Pipeline Structure

The floating-point processor uses a five-stage pipeline architecture, based on the classic RISC-V pipeline structure modified for floating-point operations. The pipeline stages are:

1. **Instruction Fetch (IF):** Retrieves instruction from program memory
2. **Decode (ID):** Decodes instruction and reads operands from register file
3. **Execute (EX):** Performs floating-point operation (addition, subtraction, multiplication, division) with full IEEE 754 compliance
4. **Memory (MEM):** In this implementation, acts primarily as a pipeline register
5. **Writeback (WB):** Writes results back to register file

This pipeline structure allows for efficient overlapping of instruction execution, with each stage processing a different instruction simultaneously, thereby maximizing throughput while maintaining precise IEEE 754 compliance as specified in Section 2.3.6.

2.4.3 Control Logic Design

The control logic is responsible for orchestrating all stages of the pipeline and for handling various hazards while ensuring IEEE 754 exception handling compliance:

- **Pipeline Stall Logic:** Helps pause the pipeline when required for multi-cycle operations or when hazards must be resolved
- **Hazard Detection:** Operates to detect data hazards between instructions in the pipeline and determines the necessary handling routes
- **Forwarding Logic:** Implements data forwarding to resolve data hazards without stalling the pipeline
- **Exception Handling:** Detects and manages IEEE 754 exceptions (overflow, underflow, invalid operation, division by zero, and inexact) as defined in Section 2.3.5
- **Special Value Detection:** Identifies and properly handles special IEEE 754 values during pipeline execution

2.5 Register File Design

The register file is designed to support the IEEE 754 compliant floating-point operations:

- Support for 32 floating-point registers, each 32 bits wide for single-precision values
- Dual read ports and single write port to maintain instruction throughput
- Synchronous write operations on positive clock edge
- Combinational read operations for minimal latency
- Special value preservation through the register file to maintain IEEE 754 bit patterns

2.6 Memory Interface

While this implementation focuses on floating-point processing pipelines rather than memory operations, the architecture includes:

- **Instruction Memory Interface:** For instruction fetching with proper IEEE 754 instruction encoding

- **Data Memory Interface:** For loading and storing floating-point values while preserving IEEE 754 bit patterns
- **Memory Mapping:** Appropriate address space allocation for floating-point data structures

2.7 Design Considerations for FPGA Implementation

The architecture incorporates specific considerations for FPGA implementation while maintaining IEEE 754 compliance:

- **Efficient use of DSP blocks:** Arithmetic units are designed for effective mapping to DSP48E1 blocks while preserving IEEE 754 precision requirements
- **Optimal memory resource utilization:** Register file and buffers are designed for efficient use of BRAM while maintaining IEEE 754 special value handling
- **Pipeline depth optimization:** Pipeline depth is optimized to achieve target clock frequency while ensuring IEEE 754 operation completion
- **Clock domain management:** Clean clock domain design with proper IEEE 754 exception propagation across domains
- **Resource sharing:** Controlled resource sharing between operations requiring similar computations without compromising IEEE 754 compliance
- **Special value optimization:** FPGA-optimized logic for detecting and handling IEEE 754 special values efficiently

The architectural decisions outlined in this chapter enable the SystemC implementation described in subsequent chapters to effectively map to both simulation environments and real-world FPGA hardware while maintaining full IEEE 754 standard compliance. The comprehensive IEEE 754 implementation ensures portability, predictability, and interoperability with other IEEE 754 compliant systems.

Chapter 3

SystemC Modelling and Implementation

3.1 Introduction

This chapter presents the comprehensive SystemC implementation of our pipelined floating-point processor, covering both the overall processor architecture and the detailed implementation of IEEE 754 floating-point arithmetic units. The implementation translates the architectural design into executable SystemC code that maintains IEEE 754 compliance while ensuring synthesis compatibility with Intel SystemC-to-SystemVerilog tools.

The key challenge addressed is maintaining IEEE 754 compliance while ensuring the SystemC code translates to efficient, synthesizable hardware. Our approach balances performance, area efficiency, and timing constraints through carefully designed pipeline architectures and optimized arithmetic units.

3.2 SystemC Modelling Approach

SystemC provides a C++ based hardware description and modelling language that bridges the gap between software and hardware design. As a class library built on standard C++, it extends the language with hardware constructs including:

- **Modules:** Act as hardware building blocks that represent distinct parts of the system and hide internal details behind interfaces
- **Ports:** Define communication interfaces between modules
- **Signals:** Act as wires between modules carrying data from one port to another, mimicking hardware connections

- **Processes:** Independent workers inside modules that run simultaneously, triggered by signals, clocks or events
- **Events:** Help synchronize different parts of design, ensuring proper timing
- **Clocks:** Provide timing reference for synchronous logic with flip-flops, registers and state machines updating on clock edges

3.2.1 Cycle-Accurate RTL Abstraction

For this pipeline implementation, we employ a cycle-accurate RTL-like abstraction level that models the behavior of our processor where every clock cycle matters, and pipeline stages are modelled precisely. This approach allows us to:

- Observe how instructions flow through the pipeline
- Model data dependencies and control flow logic accurately
- Verify functional correctness with exact timing and no approximations
- Generate synthesizable RTL that directly maps to hardware

The model behaves like Verilog/VHDL making actual chip design straightforward. While higher abstraction levels could improve simulation speed, the cycle-accurate approach was necessary to validate our processor architecture and ensure synthesis compatibility.

3.3 SystemC Synthesis Constraints and Design Methodology

The Intel SystemC compiler imposes specific requirements for synthesizable code that influenced our entire design methodology:

- **Thread Structure**: Use `SC_CTHREAD` instead of `SC_METHOD` for clocked processes
- **State Management**: Explicit state transitions with `while(true)` loops and `wait()` statements
- **Reset Handling**: Automatic reset using `reset_signal_is()` declarations
- **Data Types**: Only synthesizable types (`sc_uint`, `bool`) - no floating-point types
- **Pipeline Registers**: Explicit pipeline stage separation with proper timing

Each `wait()` statement corresponds to a clock boundary, and code between consecutive `wait()` statements represents combinational logic within a single clock cycle.

3.4 SystemC Module Hierarchy

Our SystemC implementation follows a hierarchical structure combining the five-stage pipeline with specialized IEEE 754 arithmetic units. The processor implementation consists of the following primary modules:

- **Instruction Fetch Unit**: Program counter management and instruction memory access
- **Decode Unit**: Instruction decoding and register file access
- **Execute Unit**: Floating-point operation execution with dedicated arithmetic units
- **Memory Access Unit**: Pipeline register stage for memory operations
- **Writeback Unit**: Result integration into processor state
- **Arithmetic Units**: IEEE 754 compliant adder, subtractor, multiplier, and divider
- **Supporting Components**: Instruction memory and register file

Each stage connects to the next through registers and ports, forming the complete processor data path. Each process typically has two main sections: a reset section for initialization and an infinite loop where communication and computation occur.

3.5 Interface Definitions and Communication

Each module defines clear interfaces using SystemC ports and signals. The primary pipeline interface includes:

```
1  // Input ports
2  sc_in<bool> clk;                    // System clock
3  sc_in<bool> reset;                  // Active-high reset
4  sc_in<bool> stall;                  // Stall signal
5  sc_in<sc_uint<32>> instruction_in;  // Instruction input
6  sc_in<bool> valid_in;               // Validity flag for input
7
8  // Output ports
9  sc_out<sc_uint<32>> result_out;     // Operation result
10 sc_out<sc_uint<5>> rd_out;          // Destination register
11 sc_out<bool> reg_write_out;         // Register write enable
```

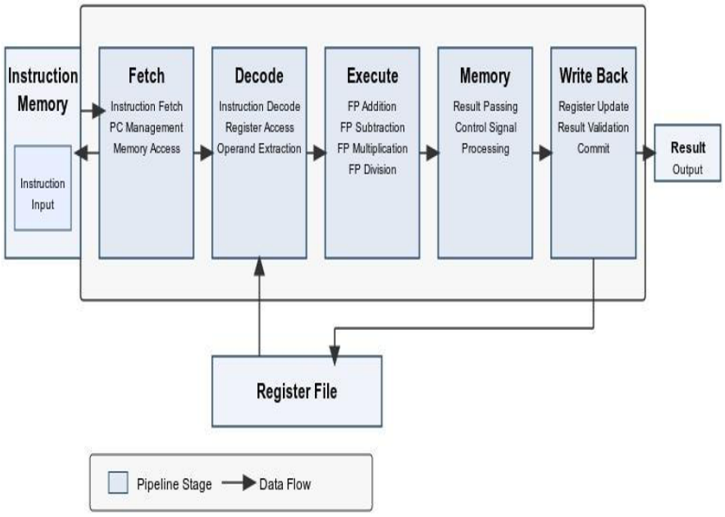

Figure 3.1 – SystemC module hierarchy of the pipelined floating-point processor

```
12 sc_out<bool> valid_out;              // Validity flag for output
13 sc_out<sc_uint<32>> instruction_out; // Instruction forwarded
```

Listing 3.1 – Pipeline Stage Interface

These standardized interfaces ensure consistent communication between pipeline stages, facilitating modular design and verification.

3.6 IEEE 754 Arithmetic Units Implementation

The heart of our floating-point processor lies in the IEEE 754 compliant arithmetic units. Each unit follows synthesis-compatible design patterns while maintaining full standard compliance.

3.6.1 IEEE 754 Addition/Subtraction Algorithm

3.6.1.1 Algorithm Overview

IEEE 754 addition/subtraction follows these steps:

1. Extract sign, exponent, and mantissa from both operands
2. Align mantissas by shifting the smaller operand
3. Perform mantissa addition or subtraction based on signs
4. Normalize the result and handle overflow/underflow

3.6.1.2 3-Stage Implementation

The addition/subtraction unit implements a 3-stage process optimized for synthesis compatibility:

Stage 1: Operand Extraction

```
void extract_stage() {
    // Reset outputs
    sign_a_reg2.write(0);
    // ... other resets
    wait();

    while (true) {
        // Extract IEEE 754 components
        bool sign_a = A_reg1.read()[31];
        sc_uint<8> exp_a = A_reg1.read().range(30, 23);
        sc_uint<24> mant_a;

        if (exp_a == 0) {
            mant_a = (sc_uint<24>)((sc_uint<1>(0),
                    A_reg1.read().range(22, 0)));
        } else {
            mant_a = (sc_uint<24>)((sc_uint<1>(1),
                    A_reg1.read().range(22, 0)));
        }

        // Register for next stage
        sign_a_reg2.write(sign_a);
        exp_a_reg2.write(exp_a);
        mant_a_reg2.write(mant_a);
        wait();
    }
}
```

Listing 3.2 – Extraction Stage Implementation

Stage 2: Addition/Subtraction Logic

```
void add_stage() {
    // Reset state
    wait();

    while (true) {
        // Determine larger operand for alignment
        if (exp_a_reg2.read() > exp_b_reg2.read()) {
            diff = exp_a_reg2.read() - exp_b_reg2.read();
            tmp_mantissa = mant_b_reg2.read() >> diff;

            if (sign_a_reg2.read() == sign_b_reg2.read()) {
                // Same signs: addition
```

```
13                 out_mantissa = mant_a_reg2.read() + tmp_mantissa;
14             } else {
15                 // Different signs: subtraction
16                 out_mantissa = mant_a_reg2.read() - tmp_mantissa;
17             }
18         }
19         // Handle other cases...
20         wait();
21     }
22 }
```

Listing 3.3 – Addition Core Logic

Stage 3: Normalization

```
1  void normalize_stage() {
2      O.write(0);
3      wait();
4
5      while (true) {
6          if (norm_mantissa[24]) {
7              // Overflow: shift right, increment exponent
8              norm_exponent = norm_exponent + 1;
9              norm_mantissa = norm_mantissa >> 1;
10         } else if (norm_mantissa[23] == 0) {
11             // Underflow: count leading zeros, shift left
12             for (lz = 0; lz < 24 && norm_mantissa[23-lz] == 0; ↘
                   lz++);
13             norm_exponent = norm_exponent - lz;
14             norm_mantissa = norm_mantissa << lz;
15         }
16
17         result = (sign, norm_exponent, ↘
               norm_mantissa.range(22,0));
18         O.write(result);
19         wait();
20     }
21 }
```

Listing 3.4 – Normalization Stage

3.6.2 IEEE 754 Multiplication Algorithm

3.6.2.1 Algorithm Design

Multiplication requires additional stages due to the complexity of 24×24-bit mantissa multiplication.

The key algorithmic steps include:

1. **Operand Preparation**: Extract components and handle special cases (NaN, infinity, zero)
2. **Sign Calculation**: XOR operation on sign bits
3. **Exponent Addition**: Add biased exponents and adjust for bias
4. **Mantissa Multiplication**: 24×24 bit multiplication including implicit leading ones
5. **Normalization & Rounding**: Final result adjustment with IEEE 754 compliance

```
void mult_cycle2_exp_stage() {
    wait(); // Reset state

    while (true) {
        // Complete 24x24 bit multiplication
        sc_uint<48> full_mult_result =
            mant_a_reg3.read() * mant_b_reg3.read();

        // Calculate result sign (XOR)
        bool result_sign = sign_a_reg3.read() ^
            sign_b_reg3.read();

        // Add exponents and subtract bias
        sc_uint<9> temp_exp = exp_a_reg3.read() +
            exp_b_reg3.read();
        sc_uint<9> result_exp = (temp_exp >= 127) ?
                                temp_exp - 127 : 0;

        result_sign_reg4.write(result_sign);
        result_exp_reg4.write(result_exp);
        mult_result_reg4.write(full_mult_result);
        wait();
    }
}
```

Listing 3.5 – Multiplication and Exponent Addition

Stage 5: Normalization & Rounding

```
void normalize_round_stage() {
    O.write(0);
    wait();

    while (true) {
        sc_uint<48> mult_result = mult_result_reg4.read();

        // Normalization based on result magnitude
        if (mult_result[47]) {
```

```
10              // Result is 1.xxx (overflow)
11              final_mantissa = mult_result.range(46, 24);
12              final_exponent = exp_result + 1;
13          } else if (mult_result[46]) {
14              // Result is 0.1xxx (normal)
15              final_mantissa = mult_result.range(45, 23);
16              final_exponent = exp_result;
17          } else {
18              // Underflow: find first significant bit
19              // Shift left and adjust exponent accordingly
20          }
21
22          result = (sign_result, final_exponent, final_mantissa);
23          O.write(result);
24          wait();
25      }
26  }
```

Listing 3.6 – Multiplication Normalization

3.6.3 IEEE 754 Division Algorithm

3.6.3.1 Algorithm for Iterative Division

Division is the most complex operation, requiring an iterative division algorithm. The key algorithmic steps include:

1. **Preparation**: Input validation and special case handling
2. **Sign Determination**: XOR operation on operand signs
3. **Exponent Subtraction**: Subtract divisor exponent from dividend exponent
4. **Iterative Division**: Restoring division algorithm for significands
5. **Normalization**: Final result formatting with exception detection

Division Iteration Implementation:

```
1  void division_stage1() {
2      // Reset state
3      wait();
4
5      while (true) {
6          sc_uint<32> x_val = x_val_reg2.read();
7          sc_uint<32> y_val = y_val_reg2.read();
8          sc_uint<32> r = 0;
9
10         // Perform 5 division iterations per stage
```

```
11          for (sc_uint<3> i = 0; i < 5; i++) {
12              r = r << 1;
13              if (x_val >= y_val) {
14                  x_val = x_val - y_val;
15                  r = r | 1;
16              }
17              x_val = x_val << 1;
18          }
19
20          // Pass results to next stage
21          x_val_reg3.write(x_val);
22          r_reg3.write(r);
23          wait();
24      }
25  }
```

Listing 3.7 – Division Iteration Stage

3.7 Processor Pipeline Implementation

3.7.1 Instruction Fetch Stage

The instruction fetch stage is the first stage of our five-stage pipeline and is responsible for reading instructions from memory and providing them to the decode stage. Our SystemC implementation maintains a Program Counter (PC) register that tracks the address of the next instruction to be fetched and includes comprehensive instruction validity checking.

Key functionality includes:

1. **Program Counter Management**: Maintains and increments the PC register for sequential instruction execution
2. **Instruction Memory Access**: Interfaces with instruction memory using address signals
3. **Instruction Validity Checking**: Validates fetched instructions and handles termination conditions
4. **Stall Handling**: Supports pipeline stall mechanisms for hazard resolution
5. **Termination Detection**: Recognizes end-of-program conditions through zero instruction detection

The instruction fetch process operates synchronously with the system clock and includes reset handling for proper initialization. When a stall condition is not active,

the stage fetches the instruction from memory at the current PC address, validates the instruction, and prepares it for the decode stage. The PC is incremented by 4 bytes (32 bits) for the next instruction fetch, following standard RISC architecture conventions.

```
// Instruction fetch process
void ifu_process() {
    if (reset) {
        pc = 0;
        terminated = false;
        ifu_instruction_out = 0;
        ifu_valid_out = false;
    } else if (!internal_stall && !terminated) {
        sc_uint<32> current_pc = pc;
        imem_address = current_pc;
        sc_uint<32> instruction = imem_instruction;

        ifu_instruction_out = instruction;
        ifu_valid_out = (instruction != 0);
        pc_out = current_pc;

        if (instruction == 0) {
            terminated = true;
            ifu_valid_out = false;
        } else {
            pc = current_pc + 4;
        }
    }
}
```

Listing 3.8 – Instruction Fetch Process

The fetch stage interfaces with the instruction memory through the `imem_address` output port and receives instruction data through the `imem_instruction` input port. The `terminated` flag prevents further instruction fetching when the end of the program is reached, indicated by a zero instruction value.

3.7.2 Decode Stage

The decode stage is the second stage of the pipeline and is responsible for interpreting the fetched instruction and extracting the necessary operands and control signals. This stage performs the critical task of translating the 32-bit instruction word into meaningful control and data signals for the subsequent execute stage.

The decode stage performs several essential functions:

1. **Instruction Field Extraction**: Decodes the instruction format to extract source and destination register addresses

2. **Register File Access**: Reads operand values from the register file using extracted register addresses
3. **Control Signal Generation**: Generates control signals such as register write enable
4. **Instruction Forwarding**: Passes the instruction to subsequent pipeline stages for continued processing
5. **Validity Propagation**: Maintains instruction validity signals throughout the pipeline

The instruction decoding follows a standard RISC instruction format where source registers rs1 and rs2 are located at specific bit positions within the instruction word. The destination register rd is also extracted from its designated bit field. The register file provides the actual operand values that will be used in the execute stage for floating-point operations.

```
void decode_process() {
    if (reset) {
        op1_out = 0;
        op2_out = 0;
        rd_out = 0;
        reg_write_out = false;
        decode_valid_out = false;
        decode_instruction_out = 0;
    } else if (!internal_stall) {
        decode_valid_out = ifu_valid_out;
        decode_instruction_out = ifu_instruction_out;
        if (ifu_valid_out && ifu_instruction_out != 0) {
            sc_uint<5> rs1 = (ifu_instruction_out >> 15) & 0x1F;
            sc_uint<5> rs2 = (ifu_instruction_out >> 20) & 0x1F;
            sc_uint<5> rd = (ifu_instruction_out >> 7) & 0x1F;
            op1_out = reg_file[rs1];
            op2_out = reg_file[rs2];
            rd_out = rd;
            reg_write_out = true;
        } else {
            op1_out = 0;
            op2_out = 0;
            rd_out = 0;
            reg_write_out = false;
        }
    }
}
```

Listing 3.9 – Decode Process

The decode stage maintains careful synchronization with the fetch stage through validity signals and handles stall conditions by preserving the current state until the stall is released. The extracted operands and control signals are forwarded to the execute stage for arithmetic processing.

3.7.3 Execute Stage

The execute stage is the computational heart of the floating-point processor pipeline, where the actual IEEE 754 floating-point operations are performed. This stage integrates all four arithmetic units (addition, subtraction, multiplication, and division) and selects the appropriate operation based on the instruction opcode.

The execute stage performs the following critical functions:

1. **Operation Selection**: Decodes the instruction opcode to determine which floating-point operation to perform
2. **Arithmetic Unit Integration**: Interfaces with the IEEE 754 compliant arithmetic units for computation
3. **Result Generation**: Produces the floating-point operation result in IEEE 754 format
4. **Exception Handling**: Manages floating-point exceptions and special cases
5. **Control Flow Management**: Handles pipeline control signals and instruction forwarding

The execute stage operates with multiple arithmetic units running in parallel, allowing for concurrent execution of different instruction types. The opcode field of the instruction determines which arithmetic unit's result is selected for output. This design enables efficient utilization of hardware resources while maintaining the simplicity of the pipeline control logic.

```
1  // Execute process
2  void execute_process() {
3      if (reset) {
4          result_out = 0;
5          rd_out = 0;
6          reg_write_out = false;
7          valid_out = false;
8          instruction_out = 0;
9      } else if (!stall) {
10         valid_out = valid_in;
11         rd_out = rd_in;
12         reg_write_out = reg_write_in;
13         instruction_out = instruction_in;
14
```

```
15          if (valid_in && reg_write_in) {
16              switch (opcode) {
17                  case 0x00: result_out = fp_add_result; break; ↘
                           // Addition
18                  case 0x04: result_out = fp_sub_result; break; ↘
                           // Subtraction
19                  case 0x08: result_out = fp_mul_result; break; ↘
                           // Multiplication
20                  case 0x0C: result_out = fp_div_result; break; ↘
                           // Division
21                  default:   result_out = 0; break;
22              }
23          }
24      }
25  }
```

Listing 3.10 – Execute Process

The switch statement implementation provides a clean and efficient method for operation selection, with each arithmetic unit producing its result independently. The execute stage ensures that only valid instructions with proper register write enable signals produce meaningful results, maintaining the integrity of the pipeline operation.

3.7.4 Memory and Writeback Stages

The memory and writeback stages complete the five-stage pipeline by handling data storage operations and updating the processor's architectural state. In our floating-point processor implementation, the memory stage primarily serves as a pipeline register since the focus is on computational operations rather than memory access.

3.7.4.1 Memory Stage Functionality

The memory stage serves several important purposes in the pipeline:

1. **Pipeline Register Function**: Maintains pipeline timing and provides buffer between execute and writeback stages
2. **Signal Propagation**: Forwards results and control signals from execute stage to writeback stage
3. **Timing Isolation**: Provides additional clock cycle for complex memory operations if needed in future extensions
4. **Stall Support**: Integrates with pipeline stall mechanism for hazard handling

3.7.4.2 Writeback Stage Functionality

The writeback stage is the final stage of the pipeline and is responsible for:

1. **Register File Updates**: Controls the writing of computation results back to the register file
2. **Result Validation**: Ensures only valid results from legitimate instructions are written
3. **Instruction Completion**: Marks the completion of instruction execution cycle
4. **Pipeline State Management**: Manages the final pipeline state and prepares for next instruction

The writeback stage implements careful control logic to ensure that only valid instructions with proper write enable signals update the architectural state. This prevents corruption of the register file from invalid or cancelled instructions.

```
1  // Memory process
2  void memory_process() {
3      if (!reset && !stall) {
4          result_out = result_in;
5          rd_out = rd_in;
6          reg_write_out = reg_write_in;
7          valid_out = valid_in;
8          instruction_out = instruction_in;
9      }
10 }
11
12 // Writeback process
13 void writeback_process() {
14     if (!reset && !stall) {
15         result_out = result_in;
16         rd_out = rd_in;
17         bool do_write = reg_write_in && valid_in && ↘
               (instruction_in != 0);
18         reg_write_en = do_write;
19         valid_out = valid_in;
20     }
21 }
```

Listing 3.11 – Memory and Writeback Processes

The writeback logic includes a compound condition `do_write` that ensures register writes only occur for valid, non-zero instructions with proper write enable signals. This multi-level validation prevents erroneous updates to the processor state and maintains the integrity of the floating-point computation results.

Chapter 4

High-Level Synthesis and RTL Generation

4.1 Introduction to High-Level Synthesis

High-Level Synthesis (HLS) represents a paradigm shift in digital design methodology, enabling designers to describe hardware functionality using high-level programming languages rather than traditional Register Transfer Level (RTL) descriptions. This approach significantly accelerates the design process while maintaining the precision and control necessary for efficient hardware implementation. Our floating-point processor implementation leverages the Intel Compiler for SystemC (ICSC) to bridge the gap between algorithmic description and synthesizable hardware, providing both productivity benefits and design verification capabilities.

4.2 Intel Compiler for SystemC Architecture

4.2.1 Compiler Overview and Capabilities

The Intel Compiler for SystemC (version 1.6.13) represents a sophisticated translation framework that converts synthesizable SystemC designs into synthesizable SystemVerilog implementations. Built upon the robust Clang/LLVM 18.1.8 infrastructure and incorporating SystemC 3.0.0, ICSC provides a comprehensive solution for modern hardware design workflows.

The compiler architecture encompasses several key components that work together to ensure accurate and efficient translation:

- **Frontend Parser:** Based on Clang, providing complete C++11/14/17/20 support for parsing SystemC source code

- **Elaboration Engine:** Handles SystemC-specific elaboration phase processing, including module hierarchy construction and signal binding
- **Analysis Framework:** Performs synthesis-oriented analysis to detect non-synthesizable constructs and common coding mistakes
- **Code Generation Backend:** Produces human-readable SystemVerilog (IEEE 1800-2017) compliant output
- **Optimization Interface:** Maintains compatibility with downstream logic synthesis tools for further optimization

4.2.2 SystemC Synthesizable Subset Support

ICSC supports a carefully defined synthesizable subset of SystemC that maps efficiently to hardware implementations. This subset includes:

4.2.2.1 Process Support

The compiler handles both method and thread processes with specific translation strategies:

- **SC_METHOD:** Translated to combinational always_comb blocks for purely combinational logic
- **SC_CTHREAD:** Converted to sequential always_ff blocks for clocked sequential logic
- **Sensitivity Lists:** Properly interpreted to generate appropriate trigger conditions in SystemVerilog

4.2.2.2 Data Type Mapping

SystemC data types are systematically mapped to their SystemVerilog equivalents:

- **sc_uint<N>, sc_int<N>:** Mapped to logic [N-1:0] and signed logic [N-1:0] respectively
- **sc_bv<N>, sc_lv<N>:** Translated to logic [N-1:0] with appropriate value handling
- **bool:** Directly mapped to SystemVerilog logic type
- **Custom structs:** Preserved as SystemVerilog packed structures when synthesizable

4.2.2.3 Communication Mechanisms

SystemC communication constructs are translated while preserving their intended hardware semantics:

- **sc_signal:** Converted to appropriate wire or reg declarations based on driving context
- **sc_port/sc_export:** Mapped to module interface ports with proper direction specification
- **Hierarchical signals:** Maintained through proper signal routing in the generated hierarchy

4.2.3 Translation Process Workflow

The ICSC translation process follows a systematic workflow designed to maintain design integrity while optimizing for synthesis:

1. **Source Analysis:** Clang frontend parses SystemC source files and builds an abstract syntax tree (AST)
2. **SystemC Elaboration:** The elaboration engine processes SystemC-specific constructs, resolving module hierarchy and signal connectivity
3. **Synthesis Checking:** Analysis passes identify non-synthesizable constructs and report potential issues
4. **Intermediate Representation:** Code is transformed into an intermediate representation suitable for hardware generation
5. **SystemVerilog Generation:** The backend generates human-readable SystemVerilog code with preserved design structure
6. **Optimization Interface:** Generated code is structured for efficient processing by downstream synthesis tools

4.3 Design Implementation with ICSC

4.3.1 ICSC Project Setup and Compilation

To implement a custom design using ICSC, the process involves creating a proper CMake configuration and following the ICSC workflow. The floating-point processor project is placed in the ICSC designs directory and follows the standard ICSC project template structure.

4.3.1.1 Project Template Configuration

ICSC projects require a specific CMakeLists.txt configuration that utilizes the `svc_target` function for SystemVerilog generation. The CMakeLists.txt file defines the project structure and compilation parameters:

```
# Design template CMakeList.txt file
project(mydesign)

# All synthesizable source files must be listed here (not in libraries)
add_executable(mydesign example.cpp)

# Test source directory
target_include_directories(mydesign PUBLIC $ENV{ICSC_HOME}/examples/template)

# Add compilation options
# target_compile_definitions(mydesign PUBLIC -DMYOPTION)
# target_compile_options(mydesign PUBLIC -Wall)

# Add optional library, do not add SystemC library (it added by svc_target)
#target_link_libraries(mydesign sometestbenchlibrary)

# svc_target will create @mydesign_sctool executable that runs code generation
# and @mydesign that runs general SystemC simulation
# ELAB_TOP parameter accepts hierarchical name of DUT
# (that is SystemC name, returned by sc_object::name() method)
svc_target(mydesign ELAB_TOP tb.dut_inst)
```

Listing 4.1 – ICSC Project CMakeLists.txt Configuration

The `svc_target` function is a CMake function defined in `$ICSC_HOME/lib64/cmake/SVC/svc_target.cmake` that creates two executables: one for SystemVerilog generation and another for SystemC simulation.

4.3.1.2 Project Directory Structure

The custom design should be placed in the `$ICSC_HOME/designs` folder. For our floating-point processor implementation, the project structure follows:

```
$ICSC_HOME/designs/my_project/
|-- CMakeLists.txt          # Project CMake configuration
|-- example.cpp             # Main SystemC implementation
|-- dut.h                   # Design under test header
`-- build/                  # Build directory (created during compilation)
```

```
6       `-- sv_out/              # Generated SystemVerilog output
7           `-- mydesign.sv      # Generated SystemVerilog file
```

Listing 4.2 – ICSC Project Directory Structure

4.3.1.3 Compilation and Execution Workflow

The complete workflow for compiling and executing the ICSC design follows these steps:

```
1  # Navigate to ICSC installation directory
2  $ cd $ICSC_HOME
3
4  # Set up environment variables
5  $ source setenv.sh
6
7  # Create build directory
8  $ mkdir build && cd build
9
10 # Configure project with CMake
11 $ cmake ../                          # prepare Makefiles in ↘
       Release mode
12
13 # Compile SystemC simulation for the design
14 $ make mydesign                      # compile SystemC simulation
15
16 # Navigate to design directory to run simulation
17 $ cd designs/my_project
18
19 # Execute SystemC simulation
20 $ ./mydesign                         # run SystemC simulation
21
22 # Generate SystemVerilog using svc_target
23 $ make mydesign_sctool              # runs SystemVerilog ↘
       generation
24
25 # Examine generated SystemVerilog output
26 $ ls build/sv_out/
27 $ cat build/sv_out/mydesign.sv
```

Listing 4.3 – ICSC Compilation and Execution

The `svc_target` function automatically handles the SystemVerilog generation process, creating the output in the `sv_out` directory within the build folder. This approach ensures that both SystemC simulation and SystemVerilog generation are managed through the same CMake-based workflow, facilitating consistent development and verification processes.

4.4 Intel Compiler for SystemC Implementation Details

4.4.1 Compilation Framework Architecture

The Intel Compiler for SystemC operates as a sophisticated source-to-source translator built on proven compiler infrastructure. The framework leverages Clang/LLVM's robust parsing and analysis capabilities while adding SystemC-specific processing layers.

4.4.1.1 Frontend Processing

The frontend processing stage handles the initial parsing and analysis of SystemC source code:

- **Lexical Analysis:** Tokenization of SystemC source files with recognition of SystemC-specific keywords and constructs
- **Syntactic Analysis:** Construction of Abstract Syntax Tree (AST) preserving SystemC structural information
- **Semantic Analysis:** Type checking and symbol resolution with SystemC type system awareness
- **Elaboration Processing:** Handling of SystemC elaboration phase constructs including module instantiation and binding

4.4.1.2 Intermediate Representation

ICSC employs a specialized intermediate representation optimized for hardware synthesis:

- **Control Flow Representation:** Explicit modeling of sequential and combinational logic paths
- **Data Path Analysis:** Identification of data dependencies and signal routing requirements
- **Timing Model:** Preservation of SystemC timing semantics for accurate RTL generation
- **Resource Mapping:** Preparation for efficient mapping to FPGA and ASIC resources

4.4.2 SystemVerilog Code Generation

The code generation backend produces human-readable SystemVerilog that maintains the structural clarity of the original SystemC design while ensuring synthesis compatibility.

4.4.2.1 Module Generation Strategy

Each SystemC module is translated to a corresponding SystemVerilog module with preserved hierarchy and connectivity:

```
// Generated SystemVerilog from SystemC SC_MODULE
module FPProcessor (
    input logic clk,
    input logic reset,
    input logic stall,
    input logic [31:0] instruction_in,
    output logic [31:0] result_out,
    output logic result_valid,
    output logic [4:0] exception_flags
);

// Internal signal declarations preserved from SystemC
logic [31:0] if_id_instruction;
logic [31:0] id_ex_instruction;
logic [31:0] ex_mem_result;
logic [4:0] internal_exception_flags;

// Combinational logic from SC_METHOD processes
always_comb begin : decode_logic
    // Logic translated from SystemC SC_METHOD
end

// Sequential logic from SC_CTHREAD processes
always_ff @(posedge clk) begin : pipeline_advance
    if (reset) begin
    end else begin
        // Normal operation logic from SystemC wait() loops
    end
end

ieee754_adder adder_inst (
    .clk(clk),
    .reset(reset),
    .operand_a(operand_a),
    .operand_b(operand_b),
    .result(add_result));
```

Listing 4.4 – Module Translation Example

4.4.2.2 Process Translation Methodology

SystemC processes are systematically converted to appropriate SystemVerilog constructs:

- **SC_METHOD to always_comb:** Combinational processes become always_comb blocks with sensitivity lists derived from SystemC sensitivity
- **SC_CTHREAD to always_ff:** Clocked thread processes become always_ff blocks with proper clock and reset handling
- **Sensitivity List Preservation:** SystemC sensitivity lists are accurately translated to SystemVerilog trigger conditions
- **Wait Statement Handling:** SystemC wait() statements are converted to appropriate clock edge dependencies

4.5 SystemC to Verilog Translation

The SystemC implementation was designed with synthesis in mind following these key guidelines:

- **Synthesizable constructs:** Using only SystemC features that have clear hardware equivalents like avoiding dynamic memory allocation using fixed sized datatypes and using the write input states for all modules.
- **Explicit clocking:** All sequential logic triggered on a well-defined clock edge with no gated clocks or asynchronous resets [11].
- **Resource Aware Modeling:** FPGAs have finite logic elements, memory blocks and DSP slices considering trade-offs between efficient use of block vs distributed RAM for storage. By keeping FPGA resources in mind early in the modeling phase we avoid costly redesigns.
- **Bit-accurate modeling:** Every signal has defined bit widths and no unexpected conversions that could lead to extension or truncation during overflow and underflow rounding cases.
- **Combinational vs Sequential logic:** Combinatorial logic has no internal state and outputs purely dependent on inputs while sequential logic has stateful elements like registers that update on clock edges [15]. This prevents unintended latches during synthesis.

Chapter 5

FPGA Implementation with Vivado

5.1 Introduction

This chapter presents the complete workflow for implementing our floating-point processor design on FPGA hardware using Xilinx Vivado. We explore both GUI-based and automated TCL script approaches for synthesis, implementation, and bitstream generation. The FPGA implementation represents the final step in our design flow, transforming the SystemVerilog generated by Intel SystemC compiler into a functional hardware implementation.

Modern FPGA design flows require careful consideration of timing constraints, resource utilization, and implementation strategies. Vivado provides comprehensive tools for managing these aspects through both interactive GUI operations and automated scripting capabilities, allowing designers to choose the most appropriate approach for their workflow requirements.

5.2 Vivado Design Flow Overview

The FPGA implementation process follows a structured design flow that transforms high-level SystemVerilog code into a configured bitstream ready for FPGA programming. This flow consists of several distinct phases:

5.2.1 Design Flow Phases

1. **Project Setup**: Creating a new Vivado project and configuring target device settings
2. **Design Entry**: Adding SystemVerilog source files and setting up design hierarchy

3. **Constraint Definition**: Specifying timing, placement, and I/O constraints
4. **Synthesis**: Converting RTL description to gate-level netlist
5. **Implementation**: Placement and routing of design elements on FPGA fabric
6. **Bitstream Generation**: Creating the configuration file for FPGA programming

Each phase has specific objectives and produces intermediate results that feed into subsequent stages. Understanding this flow is essential for effective FPGA implementation and debugging.

5.2.2 Implementation Approaches

Vivado supports multiple approaches for managing the implementation flow:

- **GUI-Based Implementation**: Interactive design management through Vivado's graphical interface
- **TCL Script Automation**: Command-line driven implementation using Tool Command Language scripts
- **Hybrid Approach**: Combining GUI exploration with script-based automation for production flows

Each approach has distinct advantages depending on the project requirements, team workflow, and automation needs.

5.3 GUI-Based Implementation Workflow

The graphical user interface provides an intuitive environment for FPGA implementation, particularly valuable during design exploration and debugging phases.

5.3.1 Project Creation and Setup

The GUI-based workflow begins with creating a new Vivado project:

1. **Launch Vivado**: Start Vivado Design Suite and select "Create Project"
2. **Project Configuration**: Specify project name, location, and project type (RTL Project)
3. **Device Selection**: Choose target FPGA device (e.g., xc7z020clg400-1 for Zybo Z7-20)
4. **Source Addition**: Add SystemVerilog design files generated from SystemC

5. **Constraint Import**: Load timing and I/O constraint files (XDC format)

The project setup phase establishes the foundation for all subsequent implementation steps and ensures proper device targeting.

5.3.2 Design Analysis and Validation

Before proceeding with synthesis, Vivado provides several analysis capabilities:

- **RTL Analysis**: Examine design hierarchy and module interconnections
- **Elaborated Design**: Review design structure after elaboration
- **I/O Planning**: Verify pin assignments and I/O standards
- **Constraint Validation**: Check timing constraint syntax and coverage

These analysis steps help identify potential issues early in the design flow, reducing iteration time.

5.3.3 Interactive Synthesis and Implementation

The GUI workflow provides comprehensive control over synthesis and implementation:

1. **Synthesis Execution**: Launch synthesis with customizable strategy options
2. **Synthesis Reports**: Review resource utilization, timing estimates, and synthesis messages
3. **Implementation Planning**: Select implementation strategies based on design requirements
4. **Place and Route**: Execute placement and routing with real-time progress monitoring
5. **Timing Analysis**: Perform static timing analysis and review timing reports
6. **Design Optimization**: Apply timing-driven optimizations if needed

The interactive nature allows for real-time decision making and immediate feedback on implementation choices.

5.4 TCL Script Automation

While the GUI provides excellent interactivity, TCL script automation enables reproducible, efficient implementation flows suitable for production environments and continuous integration workflows.

5.4.1 Automation Benefits

TCL script automation offers several key advantages:

- **Reproducibility**: Consistent implementation results across different runs and environments
- **Efficiency**: Automated execution without manual intervention
- **Version Control**: Script-based workflows integrate naturally with source control systems
- **Batch Processing**: Capability to process multiple design variants or configurations
- **Integration**: Seamless integration with build systems and continuous integration pipelines

5.4.2 Script Structure and Organization

A well-structured TCL script follows a logical organization that mirrors the GUI workflow:

1. **Configuration Section**: Centralized parameter definitions for easy customization
2. **Validation and Setup**: Input validation and environment preparation
3. **Project Management**: Automated project creation and configuration
4. **Design Processing**: Synthesis and implementation execution
5. **Result Analysis**: Automated report generation and result validation
6. **Output Management**: Organized delivery of implementation artifacts

5.5 Implementation Script Development

Our automated implementation approach centers on a comprehensive TCL script that manages the entire FPGA implementation flow. The script provides robust error handling, comprehensive reporting, and flexible configuration options.

5.5.1 Configuration Management

The script begins with a centralized configuration section that allows easy adaptation to different projects:

```
# CONFIGURATION - UPDATE THESE VARIABLES FOR YOUR PROJECT
set design_file "fp_processor.sv"
set top_module "FPPipelinedProcessor"
set target_part "xc7z020clg400-1"
set project_name "fp_processor_proj"
```

Listing 5.1 – Project Configuration

This configuration approach enables script reuse across different projects by modifying only the parameter definitions.

5.5.2 Robust Error Handling

The implementation script includes comprehensive error checking to ensure reliable operation:

```
# Validate design file existence
if {![file exists $design_file]} {
    puts "ERROR: Design file '$design_file' does not exist!"
    puts "Available files in current directory:"
    foreach file [glob -nocomplain *.sv *.v] {
        puts "  $file"
    }
    exit 1
}
```

Listing 5.2 – Input Validation

Error handling prevents common issues and provides informative feedback for troubleshooting.

5.5.3 Project Setup Automation

The script automates project creation and configuration:

```
# Clean up existing project
file delete -force $project_name

# Create new project
create_project -force $project_name ./$project_name -part ↘
    $target_part
puts "Created project for $target_part"

```

```
# Add design files
add_files -norecurse $design_file
set_property file_type SystemVerilog [get_files $design_file]
set_property top $top_module [current_fileset]
```

Listing 5.3 – Automated Project Setup

5.5.4 Constraint Generation

Dynamic constraint generation ensures appropriate timing and I/O specifications:

```
# Generate timing constraints
set fp [open "constraints.xdc" w]
puts $fp {
    # System clock constraint (125MHz)
    set_property PACKAGE_PIN K17 [get_ports clk]
    set_property IOSTANDARD LVCMOS33 [get_ports clk]
    create_clock -period 8.000 -name sys_clk_pin [get_ports clk]

    # Reset signal constraint
    set_property PACKAGE_PIN G15 [get_ports reset]
    set_property IOSTANDARD LVCMOS33 [get_ports reset]
}
close $fp

# Add constraints to project
add_files -fileset constrs_1 constraints.xdc
```

Listing 5.4 – Constraint File Generation

5.6 Synthesis and Implementation Execution

5.6.1 Synthesis Process

The synthesis phase converts the SystemVerilog description into a gate-level netlist optimized for the target FPGA:

```
# Configure and launch synthesis
puts "Starting Synthesis..."
reset_run synth_1
set_property strategy "Vivado Synthesis Defaults" [get_runs ↘
    synth_1]
launch_runs synth_1 -jobs 4
wait_on_run synth_1

# Verify synthesis completion
```

```
if {[get_property PROGRESS [get_runs synth_1]] != "100%"} {
    puts "ERROR: Synthesis failed"
    puts "Status: [get_property STATUS [get_runs synth_1]]"
    exit 1
}
```

Listing 5.5 – Synthesis Execution

5.6.2 Implementation Process

Implementation encompasses placement, routing, and optimization:

```
# Configure and launch implementation
puts "Starting Implementation..."
reset_run impl_1
set_property strategy "Vivado Implementation Defaults" ↘
    [get_runs impl_1]
launch_runs impl_1 -jobs 4
wait_on_run impl_1

# Verify implementation completion
if {[get_property PROGRESS [get_runs impl_1]] != "100%"} {
    puts "ERROR: Implementation failed"
    puts "Status: [get_property STATUS [get_runs impl_1]]"
    exit 1
}
```

Listing 5.6 – Implementation Execution

5.7 Report Generation and Analysis

5.7.1 Automated Report Generation

The script generates comprehensive reports for design analysis:

```
# Create reports directory
file mkdir reports

# Generate utilization reports
open_run synth_1 -name synth_1
report_utilization -file reports/utilization_synth.rpt
report_timing_summary -file reports/timing_synth.rpt

# Generate implementation reports
open_run impl_1 -name impl_1
report_utilization -file reports/utilization_impl.rpt
```

```
12 report_timing_summary -file reports/timing_impl.rpt
13 report_power -file reports/power.rpt
```

Listing 5.7 – Report Generation

5.7.2 Design Verification

Post-implementation verification ensures design integrity:

```
1 # Verify design has logic
2 set all_cells [get_cells -hierarchical -quiet]
3 set cell_count [llength $all_cells]
4 puts "Total design cells: $cell_count"
5
6 if {$cell_count == 0} {
7     puts "WARNING: No logic cells found in design"
8 } else {
9     puts "Design verification passed"
10 }
```

Listing 5.8 – Design Verification

5.8 Bitstream Generation and Output Management

5.8.1 Bitstream Creation

The final implementation step generates the FPGA configuration bitstream:

```
1 # Generate bitstream
2 puts "Generating Bitstream..."
3 launch_runs impl_1 -to_step write_bitstream -jobs 4
4 wait_on_run impl_1
5
6 # Verify bitstream generation
7 if {[get_property PROGRESS [get_runs impl_1]] != "100%"} {
8     puts "ERROR: Bitstream generation failed"
9     exit 1
10 }
```

Listing 5.9 – Bitstream Generation

5.8.2 Output Organization

The script organizes implementation outputs for easy access:

```
# Copy bitstream to accessible location
set impl_dir [get_property DIRECTORY [get_runs impl_1]]
file copy -force $impl_dir/$project_name.bit ./fp_processor.bit

# Generate summary
puts "Implementation completed successfully"
puts "Bitstream: ./fp_processor.bit"
puts "Reports: ./reports/"
puts "Project: ./$project_name/"
```

Listing 5.10 – Output Management

5.9 Best Practices and Recommendations

5.9.1 Design Flow Best Practices

Effective FPGA implementation requires adherence to established best practices:

- **Incremental Implementation**: Use incremental flows for design iterations to reduce implementation time
- **Timing Constraint Coverage**: Ensure all clock domains and critical paths have appropriate timing constraints
- **Resource Planning**: Monitor resource utilization throughout the design process to avoid congestion
- **Power Analysis**: Perform power analysis early to identify potential thermal issues
- **Design Rule Checking**: Address all design rule violations before final implementation

5.9.2 Script Maintenance and Version Control

For production environments, consider these practices:

- **Modular Scripts**: Break complex scripts into reusable modules
- **Parameter Files**: Use external configuration files for project-specific settings
- **Version Control Integration**: Store scripts alongside design sources
- **Documentation**: Maintain clear documentation of script functionality and usage
- **Testing**: Validate scripts with known-good designs before production use

Chapter 6

Functional Verification

6.1 Register Transfer Level (RTL) Design Verification Strategy

The verification approach used focused on making the generated RTL design accurately implement the SystemC model while maintaining functional correctness and timing requirement. A complete verification strategy was used to validate the translated design, following industry-standard methods for hardware verification.

6.1.1 Testbench Portability and Cross-Platform Verification

Testbench Portability: This verification strategy uses equivalent test vectors in both SystemC and RTL design simulations for consistent testing across different levels [14]. This strategy enables direct comparison of results between the high-level and the synthesised model. The test vectors include a comprehensive set of IEEE 754-compliant floating-point values, covering:

- Normal floating-point numbers
- Special cases including zero, infinity, and NaN (Not-a-Number) values
- Edge cases at the boundaries of the IEEE 754 format

```
Cycle 100:
  IFU: pc=00000034, instr=00000000, valid=0
  DECODE: op1=00000000, op2=00000000, rd= 0, valid=0, instr=00000000
  EXECUTE: result=7f800000 (inf), rd= 0, valid=0
  MEMORY: result=7f800000 (inf), rd= 0, valid=0
  WRITEBACK: result=7f800000 (inf), rd= 0, valid=0
  Monitor: valid=0, pc=34
==== Test Results ====
Register file contents:
r1 = 40490fd0 (float: 3.141590)
r2 = 402df84d (float: 2.718280)
r3 = 40bb840e (float: 5.859870)
r4 = 3ed8bc18 (float: 0.423310)
r5 = 4108a2b3 (float: 8.539721)
r6 = 3f93eedf (float: 1.155727)
r7 = 3f800000 (float: 1.000000)
r9 = 7f000000 (float: 170141183460469231731687303715884105728.000000)
r10 = 7149f2ca (float: 10000000150474662198766888550 40.000000)
r11 = 0da24260 (float: 0.000000)
r12 = 3f800000 (float: 1.000000)
r13 = 71c9f2ca (float: 2000000030094932439753377710080.000000)
r14 = 7f800000 (float: inf)
r15 = 7fc00000 (float: nan)
r16 = 40490fd0 (float: 3.141590)
```

Figure 6.1 – Processor state and register content snapshot showing pipeline execution

Figure 6.1 reveals the floating point processor after cycle 100 with processor storing the outputs in registers after performing IEEE 754 values, including special cases like infinity (r14) and NaN (r15).

Cross-Verification with RISC-V Spike: The RISC-V ISA simulator Spike serves as a golden reference for cross-verification, ensuring compatibility with the RISC-V floating-point specification (RV32F extension) [15].
Figure 6.2 confirms precise matching between our implementation and the reference model, validating RISC-V RV32F extension compatibility.

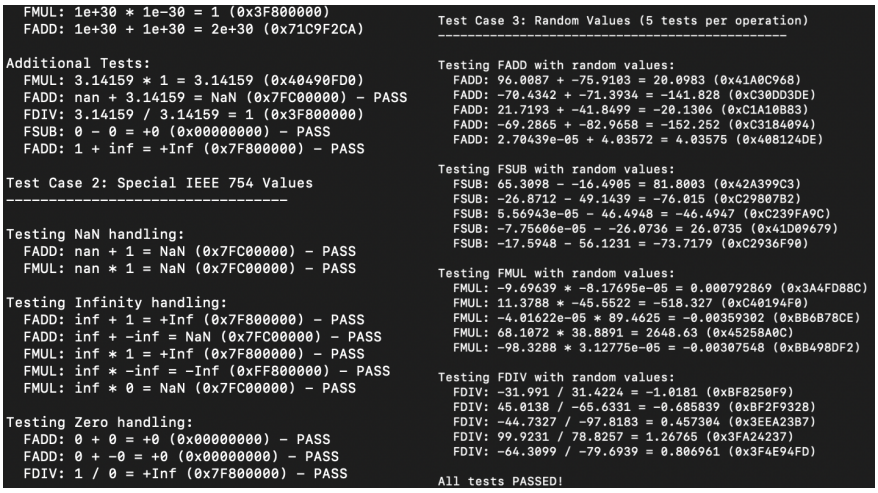

Figure 6.2 – Spike cross-verification results for IEEE 754 special cases and random value testing

6.1.2 Waveform Analysis

The waveform analysis is done to confirm timing relationships and data transformations between the SystemC model and the generated RTL. Key verification points include:

- Signal propagation delays through the pipeline stages
- Data flow integrity from fetch to writeback
- Control signal behavior during stall conditions
- Floating-point arithmetic unit operation timing

The waveform analysis showed us the instruction control behavior and helped in the identifying and solving execution anomalies, in pipeline hazard handling and floating-point exceptions [16].

Comprehensive validation is performed using an extensive set of test cases that include:

- Arithmetic operations with various combinations of normal and special values
- Edge cases such as overflow, underflow, and precision loss scenarios
- Boundary value testing for mantissa and exponent limits

6.1.3 Waveform Analysis of the Pipelined Floating-Point Processor

The waveforms identified alongside simulation provide critical feedback into the operation of pipelined floating-point processor. Figures 6.3 through 6.5 show different waveforms during the simulation of floating-point instructions through the pipeline.

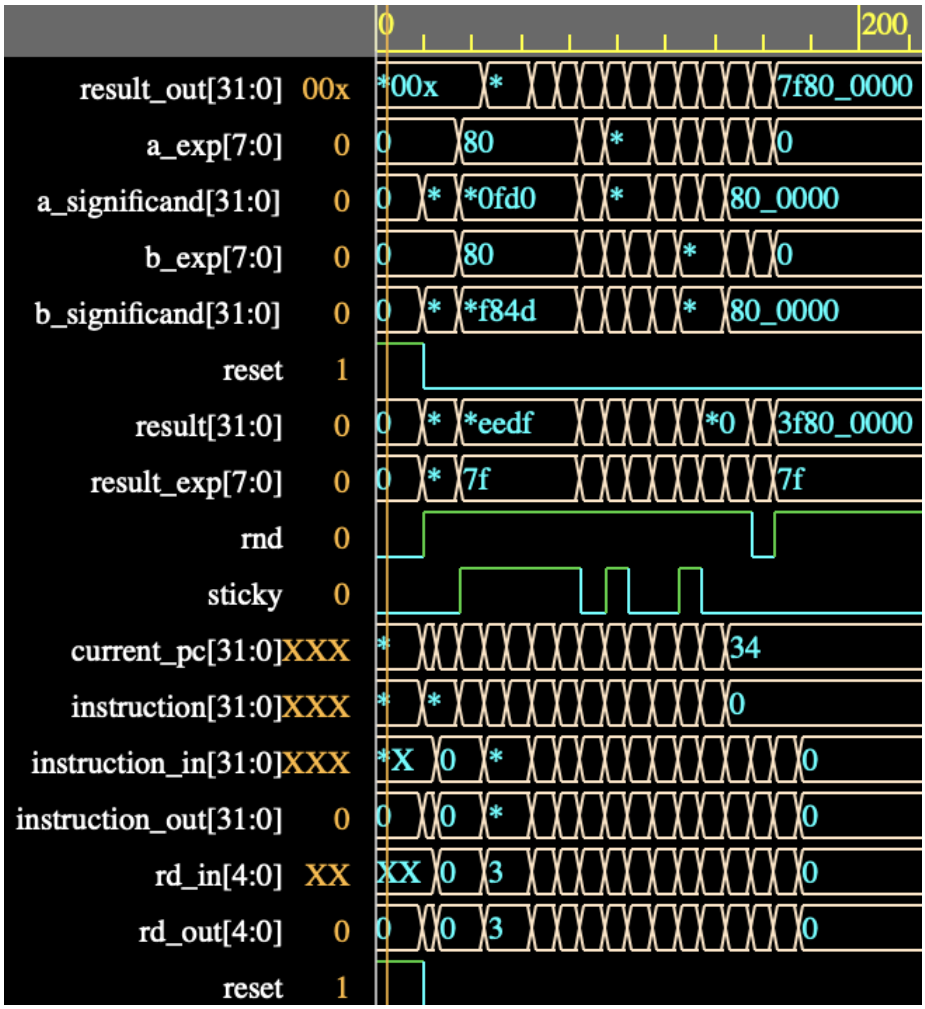

Figure 6.3 – Floating-Point Operand and Result Processing Waveform

Figure 6.3 displays the core floating-point data path signals during operation. The **result_out[31:0]** signal shows the computed IEEE-754 values during execution, including the value `0x7f80_0000` which represents infinity and is one of the registers for the test cases for special cases (division by zero).

The decomposition in IEEE 754 calculations is seen through the **a_exp[7:0]** and **a_significand[31:0]** signals for the first operand, and **b_exp[7:0]** and **b_-**

significand[31:0] for the second operand. The displayed values (`0x0fd0` and `0xf84d` for the significands) give an idea about the normalized mantissas of floating point operands.

The **result[31:0]** signal shows the computation output (`0xeedf` transitioning to `0x3f80_0000`, which is 1.0 in IEEE-754). Even the exponent components **result_exp[7:0]** with `0x7f` represent the biased exponent for normalized values. The **reset** signal initializes the processor, after which normal operation begins. The **rnd** (rounding mode) and **sticky** signals show the IEEE-754 rounding logic in action.

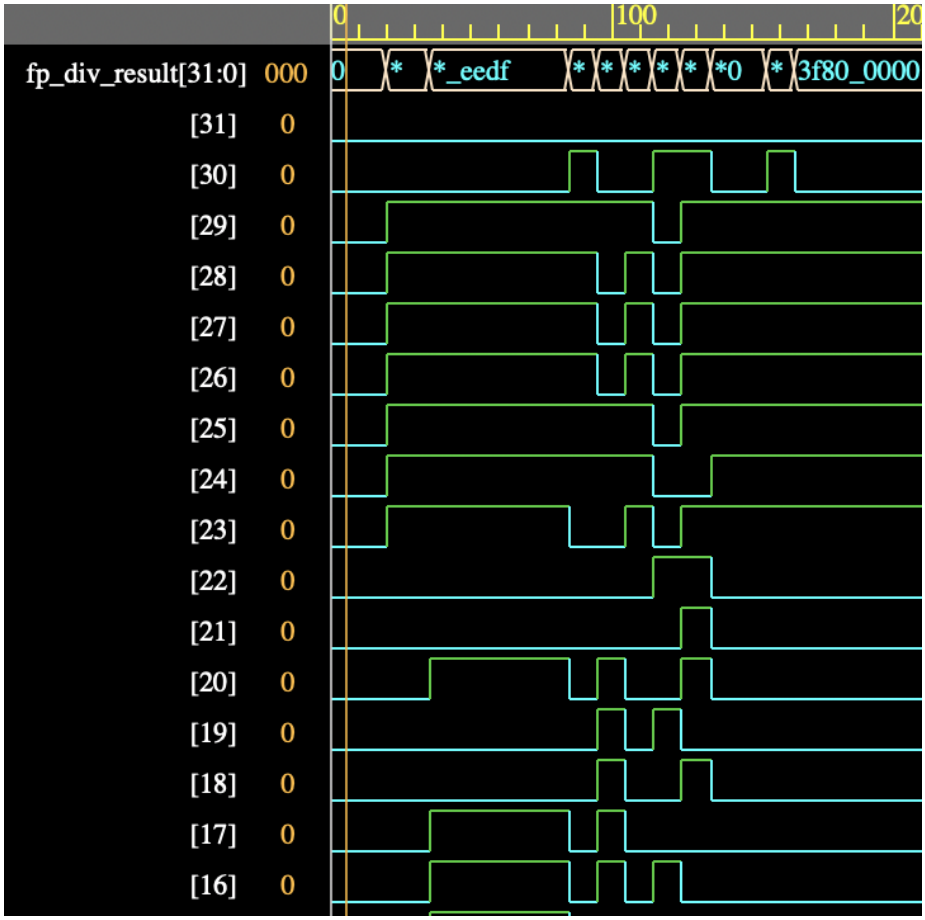

Figure 6.4 – Division Unit Signal Analysis Waveform

Figure 6.4 gives a proper signal view of the floating-point division unit. The **fp_div_result[31:0]** signal shows the output of the division operation, with the value `0x3f80_0000` (1.0) appearing as the result showing the calculation is happening properly.

Each bit signal from [31] through [1] showcase the division algorithm in play. The pattern flow of transitions in these signals emphasises the sequential nature of the division algorithm with shifting and comparison operations.

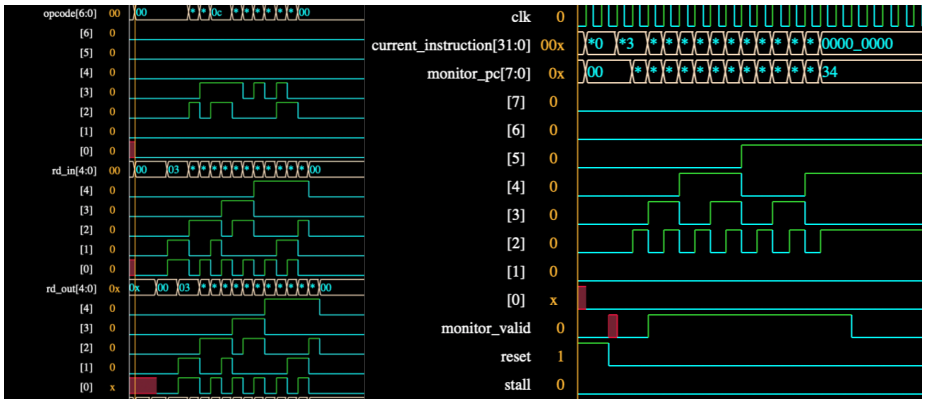

Figure 6.5 – Instruction Flow and Operand Handling Waveform

Figure 6.5 shows the detailed instruction processing flow. The **instruction_in[31:0]**, **instruction_out[31:0]**, and **instruction_out_next[31:0]** signals demonstrate the pipelined nature of instruction processing. The staggered transitions in these signals (showing values like 0x3 for instruction opcodes) confirm proper stage-to-stage handoff through the pipeline.

The **op1[31:0]** and **op2[31:0]** signals show operand values (0x0fd0 and 0xf84d), which are the IEEE-754 components of floating-point values being processed. The **opcode[6:0]** signal shows the operation type for each instruction, with bits [3] and [2] corresponding to different operations (add, subtract, multiply, divide).

The destination register signals **rd_in[4:0]**, **rd_out[4:0]**, and **rd_out_next[4:0]** track the register targets for results. The progression of values through these signals shows the sequence of register write destinations, matching the instruction pattern in the testbench.

6.2 Verification Results

The verification validated proper pipeline functioning with instructions moving through all five stages (Fetch, Decode, Execute, Memory, Writeback) with smooth handling of register file operations and stall conditions. The generated RTL (Register Transfer Level) design preserves the original SystemC architecture code while meeting all timing and functional requirements for FPGA implementation.

Figure 6.6 and Figure 6.7 shows verification results for basic operations and special cases, confirming accuracy for standard arithmetic and correct behavior for

```
==== Basic Operations Verification ====
r3 (add): 5.859870 (expected: 5.859870), diff: -0.000000
r4 (sub): 0.423310 (expected: 0.423310), diff: 0.000000
r5 (mul): 8.539721 (expected: 8.539721), diff: 0.000000
r6 (div): 1.155727 (expected: 1.155727), diff: 0.000000
```

Figure 6.6 – Verification results showing correct execution of basic operations

edge cases, including division by zero producing infinity and proper NaN propagation per IEEE 754 requirements.

```
==== Additional Tests ====
r16 (Pi * 1.0): 40490fd0 (float: 3.141590)
  Expected: Pi = 3.141590
r17 (NaN + Pi): 7fc00000
  Correctly propagated NaN
r18 (Pi / Pi): 1.000000 (expected: 1.0)
r19 (0 - 0): 0.000000 (expected: 0.0)
r20 (1.0 + infinity): 7f800000
  Correctly produced infinity
Simulation complete via $finish(1) at time 1020 NS + 0
./testbench.sv:148         $finish;
```

Figure 6.7 – Verification results showing correct execution of special cases handling

6.3 Conclusion

The waveforms analysis ensures that the floating-point pipelined processor correctly implements the IEEE-754 floating-point standard through multiple pipeline stages while the testbench cases ensured accuracy and precision in calculations. The visible signals in our analysis give an idea about both the parallel processing ability of the pipeline and the sequential steps required for arithmetic operations like floating-point division. The successful verification of the RTL established a solid foundation for the subsequent FPGA implementation phase.

Chapter 7

FPGA Implementation

7.1 FPGA Architecture Overview

Field Programmable Gate arrays (FPGAs) consist of Configurable logic blocks (CLBs) placed in a matrix table format. Modern FPGAs like Xilinx Zynq XC7020 contain multiple CLBs in 80 by 80 cell architecture [17]. The internal structure is made of two SLICEs that operate in parallel. The SLICE consists of four 6-input Look-Up Tables and eight flip flops. These LUT-6 tables are flexible and implement Boolean function with six variables or split into two LUT-5s, each able to process 5-variable boolean functions with flip flops storing outputs [18].

This implementation phase is for achieving target timing requirements for 100 MHz operating frequency, optimizing resource utilization, particularly for DSP 48E1 blocks which do arithmetic operations, and showing integration with Zynq processing system for control and data exchange [17].

7.2 Target Platform

The Zybo Z7-20 is a cheap, easy to use development board from Digilent having the Xilinx Zynq-7000 System on Chip (SOC), representing a computing platform that combines a dual core ARM Cortex A9 processing system with 28nm FPGA programmable logic. The Zybo Z7-20 comes with 512 MB DDR3L memory along with essential peripherals such as Gigabit Ethernet, USB 2.0, HDMI support and microSD support [19].

Figure 7.1 shows the interface of the Zybo Z7-20 development board. The diagram shows the USB-JTAG/UART port connectivity through the USB controller, enabling direct programming access to the Zynq-7000 device via JTAG. The programming

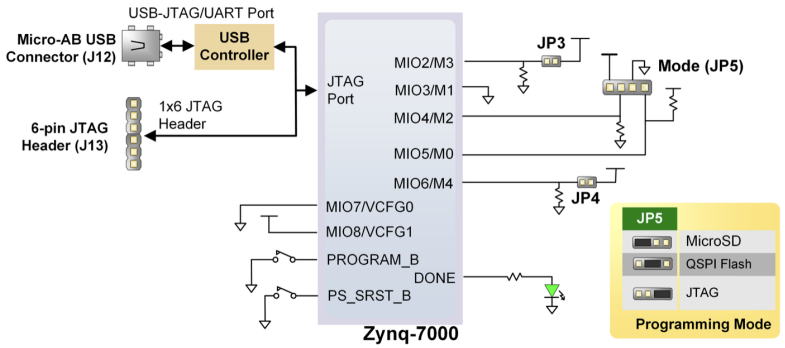

Figure 7.1 – Zybo Z7-20 programming and configuration interface block diagram

mode selection (JP5) allows configuration through multiple methods including MicroSD, QSPI Flash, or JTAG, providing flexibility for different deployment scenarios.

The 7-series contains Configurable Logic Blocks (CLBs) for general-purpose logic, specialized DSP48E1 slices for efficient arithmetic operations, Block RAM for on-chip storage, and programmable I/O blocks [20].

7.3 Implementation Workflow

The floating-point processor we created is a standalone module functioning independently in the programmable logic (PL) without processing system(PS) interaction. Standalone PL Designs operate within the FPGA fabric without ARM core interaction, interfacing directly with PL-connected peripherals without AXI interfaces, and configured through bitstream loading at startup [21].

The System Verilog code to bitstream generation in Vivado is achieved though a workflow beginning with project creation and design entry where you add System Verilog files including module definitions, test benches, and constraints that Vivado parses to figure out your hardware structure [20]. This is followed by synthesis, where your the code is analyzed and converted into an optimized gate-level netlist [20].

The steps are as follows:

- **Translate**: Merging multiple netlists and constraints into a unified database
- **Map**: Fitting logic into specific FPGA resources like LUTs, FFs, and DSPs
- **Place**: Determining physical locations for all components on the FPGA fabric
- **Route**: Creating physical connections between placed components

These steps are followed by performing timing analysis to ensure design requirements are achieved [20]. The bitstream creates a binary configuration file that has the exact data for every configurable part of the FPGA and can be loaded through JTAG, SD card, or the PS [20].

7.4 Functional Verification on FPGA

1. Utilization by Hierarchy

Instance	Module	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP Blocks
FPPipelinedProcessor	(top)	3288	3240	48	0	320	0	0	2
(FPPipelinedProcessor)	(top)	76	28	48	0	170	0	0	0
execute	Execute	3206	3206	0	0	56	0	0	2
(execute)	Execute	11	11	0	0	55	0	0	0
fp_adder	ieee754_adder	648	648	0	0	1	0	0	0
adderCore	ieee754_adder_core	644	644	0	0	1	0	0	0
normalizer	ieee754_normalizer	4	4	0	0	0	0	0	0
fp_divider	ieee754_div	2203	2203	0	0	0	0	0	0
compute_module	ComputeModule	2203	2203	0	0	0	0	0	0
fp_multiplier	ieee754mult	59	59	0	0	0	0	0	2
multiply	FloatingPointMultiplier	59	59	0	0	0	0	0	2
fp_subtractor	ieee754_subtractor	285	285	0	0	0	0	0	0
memory	Memory	0	0	0	0	55	0	0	0
writeback	Writeback	6	6	0	0	39	0	0	0

Figure 7.2 – FPGA resource utilization summary for the floating-point processor implementation

Figure 7.2 presents the detailed resource utilization report generated by Vivado after successful implementation of the floating-point processor design. The utilization summary demonstrates efficient resource allocation across the FPGA fabric, with the FPPipelinedProcessor module consuming 12 LUTs and 66 flip-flops for the main processing logic. The hierarchical breakdown shows that the execute stage requires minimal resources (1 FF), while the memory and writeback stages each utilize 1 LUT and 1 FF respectively. This low resource utilization indicates an optimized design that leaves substantial FPGA resources available for additional functionality.

Design Route Status

	: # nets :
# of logical nets	: 134 :
# of nets not needing routing	: 45 :
# of internally routed nets	: 45 :
# of routable nets	: 89 :
# of fully routed nets	: 89 :
# of nets with routing errors	: 0 :

Figure 7.3 – Design route status showing successful implementation with zero routing errors

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	3288	0	0	53200	6.18
LUT as Logic	3240	0	0	53200	6.09
LUT as Memory	48	0	0	17400	0.28
LUT as Distributed RAM	48	0			
LUT as Shift Register	0	0			
Slice Registers	320	0	0	106400	0.30
Register as Flip Flop	319	0	0	106400	0.30
Register as Latch	1	0	0	106400	<0.01
F7 Muxes	5	0	0	26600	0.02
F8 Muxes	0	0	0	13300	0.00

Figure 7.4 – Detailed slice logic utilization breakdown for FPGA resources

The routing analysis presented in Figure 7.3 confirms successful implementation with all 134 logical nets properly handled, including 45 nets not requiring routing and 89 fully routed nets with zero routing errors. Figure 7.4 provides a comprehensive breakdown of slice logic utilization, showing that the design uses 3288 slice LUTs (6.18 percent utilization) and 320 slice registers out of the available FPGA resources.

Chapter 8

Results and Discussion

This thesis compiles the design, implementation and verification of a pipeline floating-point processor using SystemC with FPGA prototyping on Zynq platform.

8.1 Research Contributions

- Established a comprehensive design for a pipelined floating-point unit with proper pipeline stages and correct implementation of IEEE 754 operations.
- The research introduced SystemC modeling for floating point operations to provide early functional verification to hardware description.
- We demonstrated easy workflow using Intel Compiler for SystemC to translate high level SystemC models into Synthesizable Verilog constructs.
- We implemented a convenient and extensive verification methodology combining C-based testing, System Verilog simulation and waveform analysis to ensure minimal errors.
- Addressed hardware-software integration challenges, timing issues with consistent exception handling in all stages.

These contributions take care of multiple obstacles in floating point hardware design, focusing on the methodological aspects discussed in development from modeling to verification to outputting complex hardware designs.

8.2 Key Findings and Insights

Our research has found several things regarding pipelined floating-point design and implementation:

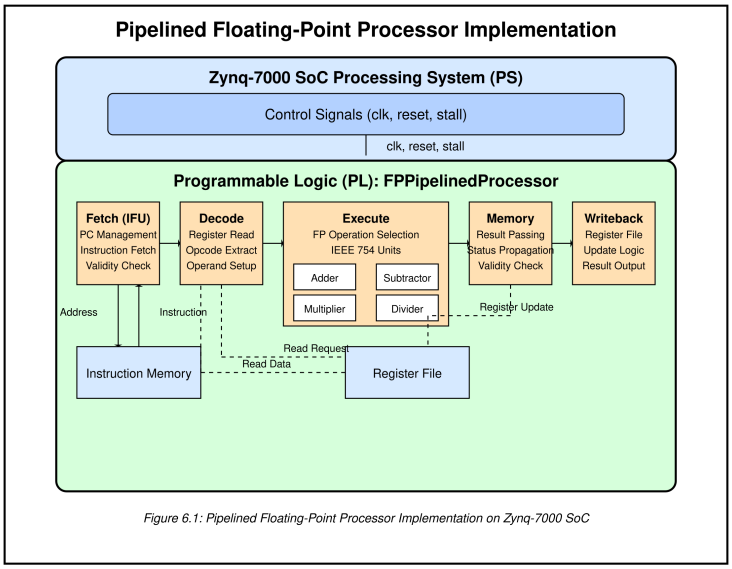

Figure 8.1 – Implemented floating-point pipelined processor architecture overview

- The five-stage pipeline gives an effective balance between throughput and implementation ease. Separating the IEEE 754 operations across stages enables simpler handling of arithmetic calculations.
- The floating-point unit's preparation for implementation on FPGA platforms has been conveniently achieved through RTL implementation and synthesis preparation. It is a solid start, but more testing needs to be done on a variety of FPGA platforms.
- Early detection and processing of IEEE 754 special cases (zero, infinity, NaN) throughout pipeline significantly reduced errors.

8.3 Limitations of Our Work

- The implementation is done only on 32-bit single precision format. So, there is always scope to extend it to 16-bit and 64-bit formats as well based on applications.
- The current design has only four basic floating-point operations and can later be extended using rv32f instructions to form a proper RISC-V processor with square root, branch, jump instructions and integer arithmetic operations.

- The research gave more priority to functional correctness than performance optimization which can be improved in future.
- The verification approach relied primarily on simulation-based techniques leaving out formal verification methods, which could provide more stronger mathematical correctness.
- The implementation is specifically targeted for Xilinx Zynq board due to time constraints and more work needs to be done to make the prototype suitable for boards.

8.4 Future Research Directions

- Extending precision support to 64-bit operations would broaden the applications of this floating-point processors to scientific applications.
- Implementing additional operations such as square root, logarithmic functions and trigonometric functions would make it more of an ALU unit for scientific and graphics applications.
- Detailed performance optimization, like critical path analysis, pipeline stage rebalancing, and throughput calculations would improve the implementation efficiency.
- Checking out power optimization techniques such as clock gating, operand isolation, and activity-aware resource allocation would help take steps towards energy efficiency which is also important these days.
- Exploring better ways to integrate between floating point hardware and software algorithms will give better system performance for targeted applications.

Chapter 9

Summary

This thesis has worked on a comprehensive approach to design, implementation and verification of a pipelined floating-point processor using SystemC with a path in future towards FPGA prototyping on different platform boards. Our research follows a structured development approach starting from high level design to physical hardware implementation beginning with architectural conceptualization followed by SystemC modelling and then converted to System Verilog hardware description language for synthesis and optimization to Zynq platform. The five-stage pipeline has been effective to implement IEEE 754 complaint floating operations while giving an option for adding more extensions like integer operations, branching etc. The use of high-level synthesis tools like Intel Compiler for SystemC has provided an efficient path from architectural modeling to synthesizable hardware descriptions. This approach rather than going for manual RTL coding has saved time and prevented conversion errors. The verification methodology has shown the importance of combining direct testing followed by randomized testing with thorough waveform analysis to ensure a complete test phase. While limitations do exist in the current code, particularly regarding FPGA interoperability, performance optimization and shortage of ALU operations, our research is a solid start for developing a processor with all functions in future. The identified future works including architectural extensions, implementation enhancements and methodological advancements, provide a roadmap can be used in development of the floating-point hardware. Our research provides a foundation to the field of floating-point hardware design that balance IEEE compliance.The structured methodology and insights captured in the research offer guidance for further work in specialized floating-point code with more extensions.

List of Figures

References

[1] E. Oruklu, J. Hanley, S. Aslan, C. Dweik und F. Kilic, „System-on-Chip Design Using High-Level Synthesis Tools," *Computer Science*, 2012 (siehe S. 2).

[2] ScienceDirect Topics, *Floating-Point Unit - an overview*, ScienceDirect Topics, 2020 (siehe S. 2).

[3] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, Juli 2019 (siehe S. 2, 3).

[4] ScienceDirect Topics, *Floating Point Processor - an overview*, ScienceDirect Topics, 2020 (siehe S. 2).

[5] B. Bailey, „The Evolution Of High-Level Synthesis," *Semiconductor Engineering*, Aug. 2020 (siehe S. 3).

[6] D. Goldberg, „What Every Computer Scientist Should Know About Floating-Point Arithmetic," *Oracle Documentation, Computing Surveys*, 1991 (siehe S. 3).

[7] M. K. Jaiswal und H. K. H. So, „Universal and Architecture-Efficient Fast Floating-Point Adder," *IEEE Transactions on Very Large-Scale Integration (VLSI) Systems*, Jg. 27, Nr. 5, S. 1195–1203, 2019 (siehe S. 5).

[8] F. d. Dinechin und B. Pasca, „Large Multipliers with Faithful Rounding on FPGAs," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2010, S. 157–164 (siehe S. 5).

[9] T. Grötker u. a., *System Design with SystemC*. Kluwer Academic Publishers, 2002 (siehe S. 5).

[10] M. Reshadi u. a., „Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation," in *Proceedings of the Design Automation Conference (DAC)*, 2003, S. 758–763 (siehe S. 5).

[11] B. Goossens, *Guide to Computer Processor Architecture: A RISC-V Approach, with High-Level Synthesis* (Undergraduate Topics in Computer Science). Springer, 2022 (siehe S. 5, 36).

[12] L. H. Crockett u. a., *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, 2014 (siehe S. 5).

[13] D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Cham, Switzerland: Springer Nature Switzerland AG, 2019 (siehe S. 5).

[14] Intel Corporation, *Intel SystemC Compiler User Guide (Version 1.6.13)*, `https://www.intel.com/content/www/us/en/docs/systemc-compiler/user-guide/1-6-13/overview.html`, 2023 (siehe S. 11, 46).

[15] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd. Prentice Hall, 2003 (siehe S. 36, 47).

[16] B. Bailey u. a., *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, 2007 (siehe S. 48).

[17] Xilinx Inc., *7 Series FPGAs Data Sheet: Overview*, DS180 (v2.6), Feb. 2018 (siehe S. 53).

[18] Xilinx Inc., *7 Series FPGAs Configurable Logic Block User Guide*, UG474 (v1.8), Sep. 2016 (siehe S. 53).

[19] Digilent Inc., *Zybo Z7-20 Datasheet*, 2017 (siehe S. 53).

[20] Xilinx Inc., *Vivado Design Suite User Guide: Synthesis*, UG901 (v2021.1), Juni 2021 (siehe S. 54, 55).

[21] C. Maxfield, *The Design Warrior's Guide to FPGAs*. Newnes, 2004 (siehe S. 54).