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

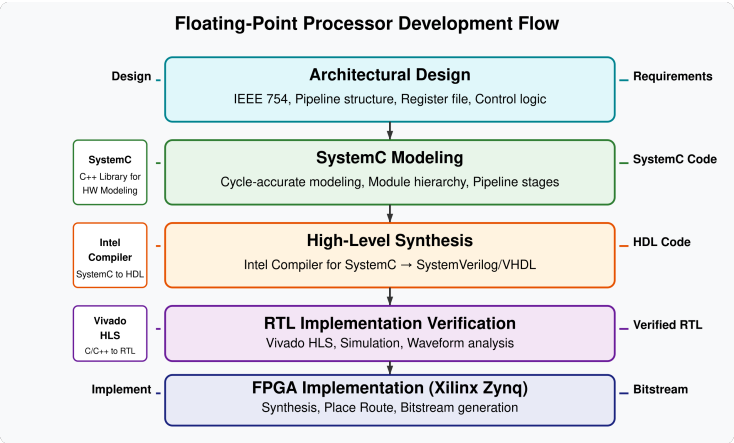

Figure 2.1 – Development flow stages of the pipelined floating-point processor

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 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].

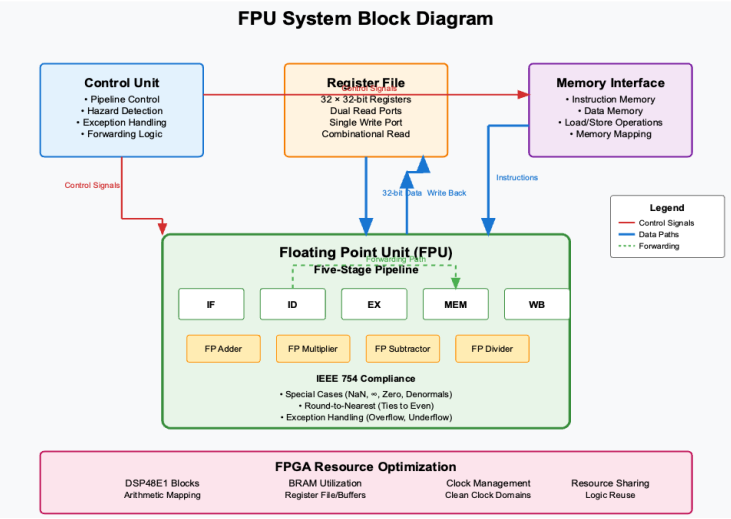

Figure 2.2 – Control logic architecture of the pipelined floating-point processor

2.3.1 Floating-Point Format and Representation

The processor implements the IEEE 754 single-precision binary floating-point format, comprising of these parts:

- **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:

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

2.3.2 Pipeline Structure

The floating-point processor uses a five-stage pipeline architecture, based of 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)
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.

2.3.3 Control Logic Design

The control logic is responsible for orchestrating all stages of the pipeline and for handling various hazards. The following are important control functions:

- **Pipeline Stall Logic:** Helps pause the pipeline when required for multi-cycle operations or when hazards must be resolved
- **Hazard Detection:** Operates to see if any data hazards exist between instructions in the pipeline and will take the necessary routes for handling these hazards
- **Forwarding Logic:** To resolve data hazards without actually stalling the pipeline, data forwarding can be in place to get accurate values
- **Exception Handling:** To detect and manage IEEE 754 exceptions (overflow, underflow, etc.)

2.4 Register File Design

The register file is designed to:

- Support a structure of 32 floating-point registers each 32 bit wide
- Support dual read ports and a single write port to meet instruction throughput
- Dependent on the positive clock edge for synchronous writing
- Combinational reads for minimal latency

2.5 Memory Interface

This implementation focused on floating point processing pipelines rather than memory operations but the architecture also has:

- **Instruction Memory Interface:** for instruction fetching
- **Data Memory Interface:** for loading and storing operations
- **Memory Mapping**

2.6 Design Considerations for FPGA Implementation

The architecture takes certain considerations for FPGA implementation:

- **Effective use of DSP blocks:** arithmetic units are made for effective mapping to DSP48E1 blocks
- **Effective use of memory resources:** register file and buffers have been created for effective use of BRAM
- **Pipelining granularity:** the depth of pipelining has been optimized to allow timing of clocks at target frequency
- **Clean clock management:** proper design with clear clock domain boundaries
- **Resource sharing:** controlled resource sharing between operations that required similar computations

The architectural decisions laid out here enable the SystemC implementation in the next chapter to effectively map to both simulation and FPGA hardware in the real world.

Chapter 3

SystemC Modelling

This chapter presents the SystemC implementation of our pipelined floating-point processor. The implementation translates the architectural design described in the last chapter into executable SystemC code. Using SystemC allows us to model hardware behavior at a more abstract level compared to high level programming languages. This design phase was crucial for validating our framework before proceeding to RTL design, enabling early detection and resolution of potential issues.

3.1 SystemC Modelling Approach

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

- **Modules:** Acts as “hardware building blocks” that represent a distinct part of system and hides internal details behind the interface
- **Ports:** Defines communication interfaces between modules
- **Signals:** They act as wires between modules carrying data from one port to another. They mimic the hardware connections
- **Processes:** Are like independent workers inside a module which run simultaneously which can be triggered by signals, clocks or events
- **Events:** Help synchronize different parts of design, ensuring they happen on time
- **Clocks:** Provide timing reference for synchronous logic with flip-flops, registers and state machines update on clock edges

3.2 Cycle-Accurate RTL Abstraction

For this pipeline implementation, we are going for 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:

- See how instructions flow in pipeline
- Model data dependencies and control flow logic
- Verify functional correctness with exact timing and no approximations

The model behaves like Verilog/VHDL making actual chip design easy. While abstraction levels could improve simulation speed, the cycle accurate approach was necessary to validate our processor architecture.

3.3 SystemC Module Hierarchy

Our SystemC implementation follows a hierarchical structure for the five-stage pipeline and the arithmetic units as well. Each stage connects to the next one on the pipeline through registers and ports forming the complete processor data path. The processor implementation consists of the following primary modules:

- **Instruction Fetch Unit**
- **Decode Unit**
- **Execute Unit**
- **Memory Access Unit**
- **Writeback Unit**
- **Instruction memory/Register File**

Each stage connects to the next one on the pipeline through registers and ports-forming the complete processor data path. Each process typically has two main sections: a reset section for initialization and configuration, and an infinite loop where communication and computation occur. Within this loop, the stage acquires new data from the previous stage, performs computation, and transfers processed information to the next stage [15].

Each `wait()` statement corresponds to a virtual-clock boundary. Code between consecutive `wait()` statements is typically specified as untimed logic, giving the HLS tool freedom to determine how many physical-clock cycles to use for implementation. [15].

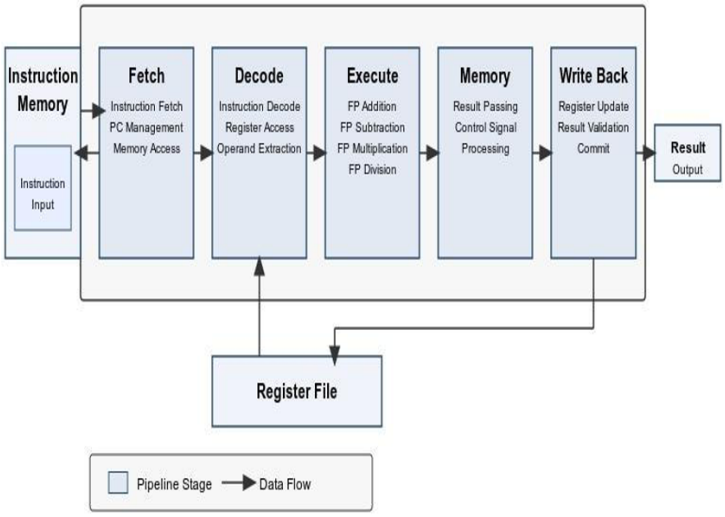

Figure 3.1 – SystemC module hierarchy of the pipelined floating-point processor

3.4 Interface Definitions

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

```
// Input ports
sc_in<bool> clk;                    // System clock
sc_in<bool> reset;                  // Active-high reset
sc_in<bool> stall;                  // Stall signal
sc_in<sc_uint<32>> instruction_in;  // Instruction input
sc_in<bool> valid_in;               // Validity flag for input

// Output ports
sc_out<sc_uint<32>> result_out;     // Operation result
sc_out<sc_uint<5>> rd_out;          // Destination register
sc_out<bool> reg_write_out;         // Register write enable
sc_out<bool> valid_out;             // Validity flag for output
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. The ALU is one of the basic blocks of the central processing unit and is mandatory within a computer because it will definitely have to perform basic math functions such as addition, subtraction, multiplication, and division

3.4.1 IEEE 754 Adder Implementation

The implementation is used for both IEEE 754 addition and subtraction algorithms with proper handling of special cases, where subtraction is performed by inverting the sign of the second operand. The implementation follows these key steps:

1. **Component Extraction:** Separate sign, exponent, and mantissa from input operands
2. **Operand Alignment:** Align mantissas based on exponent difference
3. **Mantissa Addition/Subtraction:** Perform significand operation based on signs
4. **Result Normalization:** Normalize result and adjust exponent
5. **Rounding:** Apply IEEE 754 rounding rules
6. **Special Case Handling:** Process zero, infinity, and NaN cases

```
1  void ieee754_adder_core::process() {
2      // Special case handling
3      bool a_is_nan = (exp_a == 0xFF) && (mant_a.range(22, 0) != ↘
           0);
4      bool b_is_nan = (exp_b == 0xFF) && (mant_b.range(22, 0) != ↘
           0);
5      // Normal case processing
6      if (exp_a > exp_b) {
7          sc_uint<8> diff = exp_a - exp_b;
8          sc_uint<24> tmp_mantissa = mant_b >> diff;
9          if (sign_a == sign_b) {
10             out_mantissa = mant_a + tmp_mantissa;
11         } else {
12             if (mant_a >= tmp_mantissa) {
13                 out_mantissa = mant_a - tmp_mantissa;
14                 out_sign = sign_a;
15             } else {
16                 out_mantissa = tmp_mantissa - mant_a;
17                 out_sign = sign_b;
18             }
19         }
20         out_exponent = exp_a;
21     }
22 }
```

Listing 3.2 – IEEE 754 Adder Implementation

3.4.2 IEEE 754 Multiplier Implementation

The floating-point multiplier implements the IEEE 754 multiplication algorithm, focusing on the following key operations:

1. **Sign Determination:** XOR of input signs
2. **Exponent Addition:** Sum input exponents and subtract bias
3. **Mantissa Multiplication:** Full-precision multiplication of significands
4. **Result Normalization:** Adjust for potential overflow in mantissa
5. **Rounding:** Apply IEEE 754 rounding rules
6. **Special Case Handling:** Process zero, infinity, and NaN cases

```
1  // Floating-point multiplication process
2  void FloatingPointMultiplier::multiply() {
3      if (reset) {
4          Temp_Mantissa = 0;
5          Temp_Exponent = 0;
6          Sign = 0;
7      } else {
8          // Multiply mantissas
9          Temp_Mantissa = A_Mantissa * B_Mantissa;
10
11         // Add exponents and subtract bias
12         Temp_Exponent = A_Exponent + B_Exponent - 127;
13
14         // Determine result sign
15         Sign = A_sign ^ B_sign;
16     }
17 }
```

Listing 3.3 – Floating-Point Multiplier

3.4.3 IEEE 754 Divider Implementation

The floating-point divider implements a non-restoring division algorithm with IEEE 754 compliance. The key operations include:

1. **Sign Determination:** XOR of input signs
2. **Exponent Subtraction:** Difference of exponents plus bias
3. **Mantissa Division:** Iterative division algorithm for significands
4. **Normalization and Rounding:** Adjust result format and apply rounding
5. **Special Case Handling:** Process division by zero, infinity, and NaN cases

```
void ComputeModule::compute() {
    if (reset) {
        result = 0;
    } else {
        // Prepare operands
        result_sign = a_sign ^ b_sign;
        result_exp = a_exp - b_exp + 127;
        // Initial alignment
        if (a_significand < b_significand) {
            a_significand <<= 1;
            result_exp--;
        }
        // Iterative division algorithm
        sc_uint<32> r = 0;
        for (int i = 0; i < 25; i++) {
            r <<= 1;
            if (a_significand >= b_significand) {
                a_significand = a_significand - b_significand;
                r |= 1;
            }
            a_significand <<= 1;
        }
        // Apply sign
        r |= (result_sign ? 0x80000000 : 0);
        result = r;
    }
}
```

Listing 3.4 – IEEE 754 Divider Implementation

3.4.4 Instruction Fetch Stage

The instruction fetch stage reads instructions from memory and provides them with the decode stage. Our SystemC implementation maintains a Program Counter (PC) register that holds the memory address of the next instruction to be processed. The internal stall signal ensures the fetch suspends when synchronization is happening.

Key functionality includes:

1. Program counter management
2. Instruction memory access
3. Instruction validity checking

```
// 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 3.5 – Instruction Fetch Process

The fetch stage then forwards this instruction (`ifu_instruction_out.write(instruction)`) along with validity information (`ifu_valid_out.write(instruction != 0)`) to the decode stage, while simultaneously incrementing the PC by 4 bytes (`pc = current_pc + 4`) to prepare for the next instruction fetch. The sequential fetching happens until the termination mechanism in which our case is a zero instruction triggers the end execution (`if (instruction == 0) { terminated = true }`).

3.4.5 Decode Stage

The decode stage extracts operation operands and control signals from instructions. Its primary functions include:

1. Instruction decoding
2. Register file access for operands
3. Destination register identification
4. Control signal generation
5. Propagation of validity signals

```
1  void decode_process() {
2      if (reset) {
3          op1_out = 0;
4          op2_out = 0;
5          rd_out = 0;
6          reg_write_out = false;
7          decode_valid_out = false;
8          decode_instruction_out = 0;
9      } else if (!internal_stall) {
10         decode_valid_out = ifu_valid_out;
11         decode_instruction_out = ifu_instruction_out;
12         if (ifu_valid_out && ifu_instruction_out != 0) {
13             sc_uint<5> rs1 = (ifu_instruction_out >> 15) & 0x1F;
14             sc_uint<5> rs2 = (ifu_instruction_out >> 20) & 0x1F;
15             sc_uint<5> rd = (ifu_instruction_out >> 7) & 0x1F;
16             op1_out = reg_file[rs1];
17             op2_out = reg_file[rs2];
18             rd_out = rd;
19             reg_write_out = true;
20         } else {
21             op1_out = 0;
22             op2_out = 0;
23             rd_out = 0;
24             reg_write_out = false;
25         }
26     }
27 }
```

Listing 3.6 – Decode Process

This stage interfaces with fetch stage by reading instruction (`ifu_instruction_out`) and the source registers rs1 and rs2 are identified by shifting and masking the appropriate bits (`rs1 = (ifu_instruction_out.read() » 15) & 0x1F` and `rs2 = (ifu_instruction_out.read() » 20) & 0x1F`), as is the destination register rd (`rd = (ifu_instruction_out.read() » 7) & 0x1F`).

3.4.6 Execute Stage

The execute stage is the heart of the pipelined processor where actual data processing takes place on decoded instruction which applies the appropriate operation by instruction's opcode. There are four dedicated floating-point units: adder, multiplier, subtractor and divider. Its responsibilities include:

1. Operation selection based on opcode
2. Floating-point operation execution (addition, subtraction, multiplication, division)
3. Result generation
4. Exception detection and handling
5. Control signal propagation

```
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 3.7 – Execute Process

The critical part happens when a switch statement that selects floating point operation based on opcode field: 0x00 for addition, 0x04 for subtraction, 0x08 for multiplication, and 0x0C for division. The result is forwarded through the result_out port.

3.4.7 Memory Stage

The memory stage primarily serves as a pipeline register for this implementation, as our focus is on the floating-point processing unit rather than memory operations.

```
// Memory process
void memory_process() {
    if (!reset && !stall) {
        result_out = result_in;
        rd_out = rd_in;
        reg_write_out = reg_write_in;
        valid_out = valid_in;
        instruction_out = instruction_in;
    }
}
```

Listing 3.8 – Memory Process

3.4.8 Writeback Stage

The writeback is the last stage closing the instruction execution cycle by updating the processor's architectural state. Its functions include:

1. Register write control
2. Result validity checking
3. Result integration into the processor state

```
// Writeback process
void writeback_process() {
    if (!reset && !stall) {
        result_out = result_in;
        rd_out = rd_in;
        bool do_write = reg_write_in && valid_in && ↘
            (instruction_in != 0);
        reg_write_en = do_write;
        valid_out = valid_in;
    }
}
```

Listing 3.9 – Writeback Process

Chapter 4

High-Level Synthesis and RTL Generation

4.1 Intel Compiler for System C

Our implementation used the Intel Compiler for SystemC (version 1.6.13) to facilitate the transition from SystemC to RTL(Register Transfer Level) design. RTL is an abstract design level for modeling a synchronous digital circuit, typically in terms of the transfer of digital signals from hardware register to hardware register, and the logical operations that are applied to these signals. This approach helps with faster development, seamless verification and multiple architecture experimentation. The Intel compiler essentially acted as our translator letting us describe our processor in SystemC while automatically generating the System Verilog equivalent, significantly reducing manual conversion errors and development time.

4.2 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 dis-

tributed 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 [16]. This prevents unintended latches during synthesis.

4.3 Generated Verilog Structure

The Intel SystemC Compiler managed to generate Verilog that closely matches our SystemC architecture with equivalent ports and internal logic:

- **Process Translation:** SystemC processes became either combinational always blocks (always_comb) or sequential always blocks (always_ff) when translated to Verilog procedural blocks based on their behavior and sensitivity lists
- **Signal Preservation:** The communication network defined by SystemC signals is translated to appropriate Verilog wires and registers
- **Module Hierarchy:** The hierarchical structure was maintained in the Verilog
- **Type Mapping:** SystemC-specific types are mapped to Verilog with proper bit-width preservation and structural integrity

```
module FPPipelinedProcessor (
    input logic clk,
    input logic reset,
    input logic stall,
    output logic monitor_valid,
    output logic [7:0] monitor_pc
);
// Internal signal declarations
logic internal_stall;
logic [31:0] pc_out;
logic [31:0] ifu_instruction_out;
logic ifu_valid_out;
// Combinational logic processes
always_comb begin : update_stall
    internal_stall = stall;
end
always_comb begin : update_opcode
```

```
    opcode = decode_instruction_out[31:25];
end
// Sequential processes
always_ff @(posedge clk) begin : ifu_process_ff
    if (reset) begin
        // Reset logic
    end else begin
        // Normal operation logic
    end
end
InstructionMemory imem (
    .address(imem_address),
    .instruction(imem_instruction)
);
Execute execute (
    // Ports and connections
);
// Additional module instantiations
```

Listing 4.1 – Pipelined Processor in Verilog

```
module ieee754_adder (
    input logic [31:0] A,
    input logic [31:0] B,
    output logic [31:0] O
);
// Internal signals
logic sign_a;
logic sign_b;
logic out_sign;
logic [7:0] exp_a;
logic [7:0] exp_b;
logic [7:0] out_exponent;
logic [23:0] mant_a;
logic [23:0] mant_b;
logic [24:0] out_mantissa;

ieee754_extractor extractA (
    .A(A),
    .sign(sign_a),
    .exponent(exp_a),
    .mantissa(mant_a)
);
// Additional components omitted for brevity
```

Listing 4.2 – IEEE 754 Adder in Verilog (partial implementation)

The generated Verilog maintains the modularity and clarity of the original SystemC design while ensuring compatibility with standard FPGA synthesis tools.

Chapter 5

Functional Verification

5.1 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.

5.1.1 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 5.1 – Processor state and register content snapshot showing pipeline execution

Figure 5.1 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) [16].
Figure 5.2 confirms precise matching between our implementation and the reference model, validating RISC-V RV32F extension compatibility.

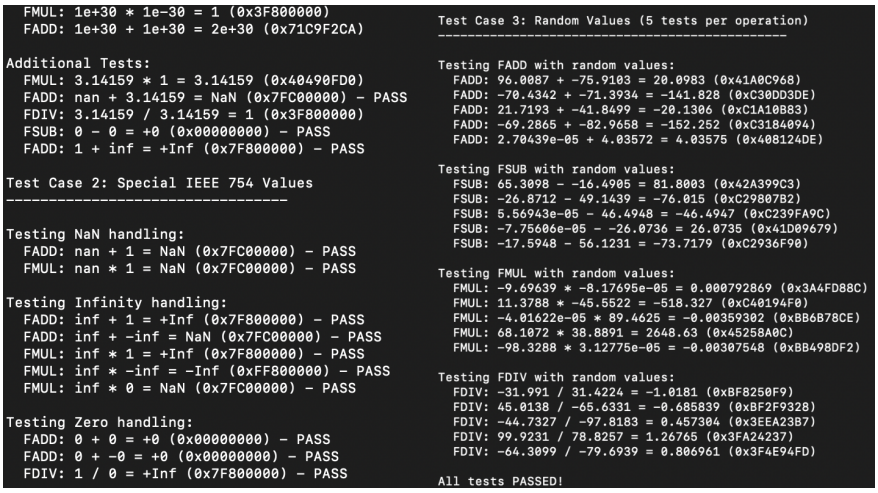

Figure 5.2 – Spike cross-verification results for IEEE 754 special cases and random value testing

5.1.2 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 [15].

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

5.1.3 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 5.3 through 5.5 show different waveforms during the simulation of floating-point instructions through the pipeline.

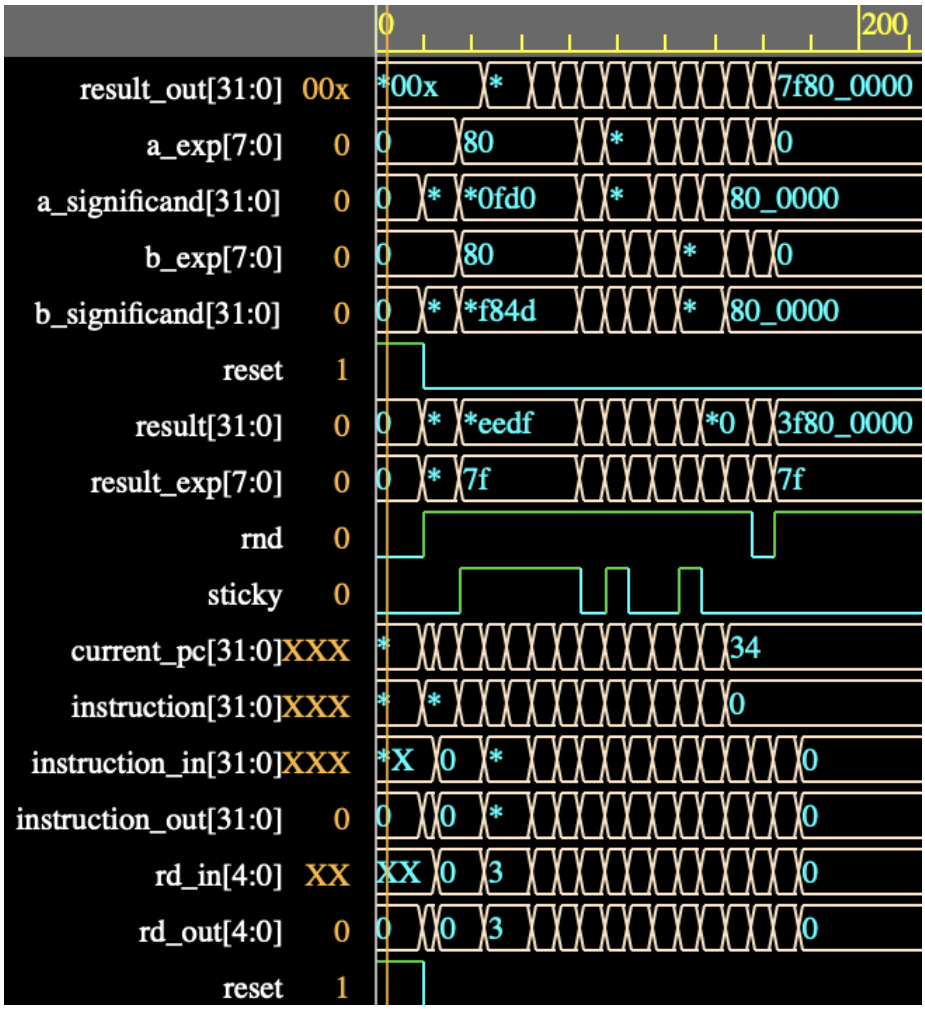

Figure 5.3 – Floating-Point Operand and Result Processing Waveform

Figure 5.3 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.

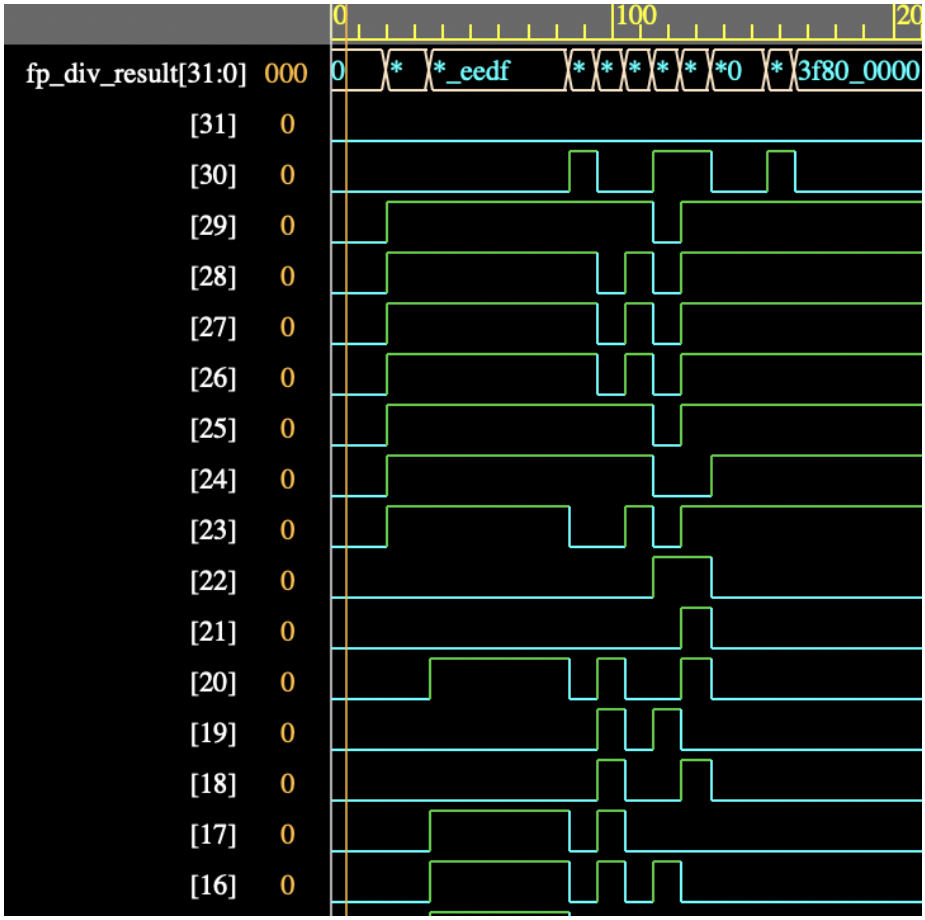

Figure 5.4 – Division Unit Signal Analysis Waveform

Figure 5.4 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.

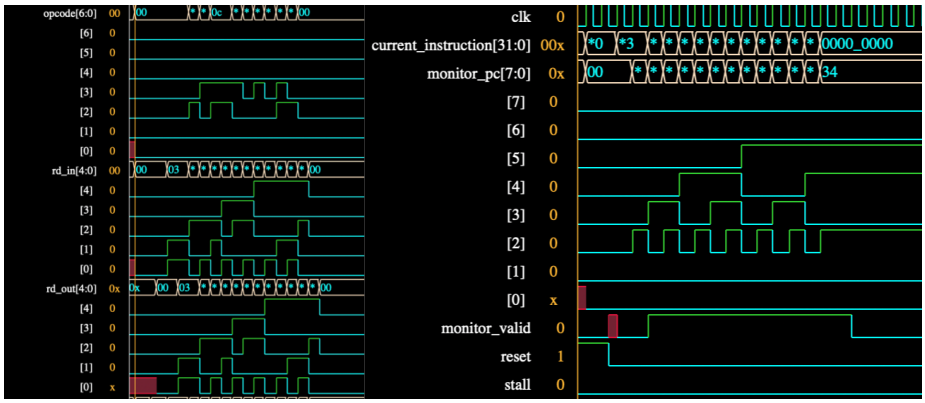

Figure 5.5 – Instruction Flow and Operand Handling Waveform

Figure 5.5 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.

5.2 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 5.6 and Figure 5.7 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 5.6 – 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 5.7 – Verification results showing correct execution of special cases handling

5.3 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 6

FPGA Implementation

6.1 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].

6.2 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 6.1 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

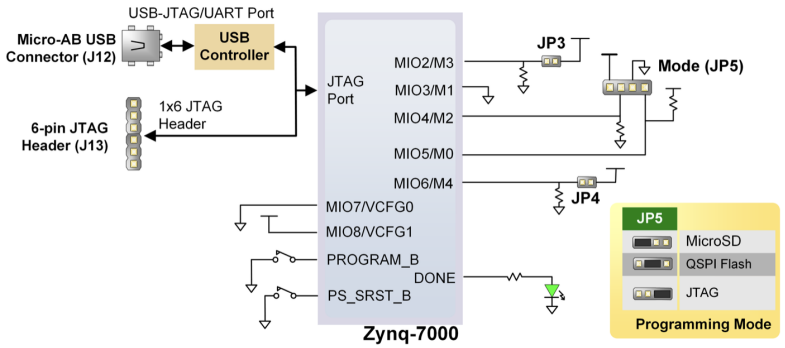

Figure 6.1 – Zybo Z7-20 programming and configuration interface block diagram

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].

6.3 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].

6.4 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 6.2 – FPGA resource utilization summary for the floating-point processor implementation

Figure 6.2 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 6.3 – 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 6.4 – Detailed slice logic utilization breakdown for FPGA resources

The routing analysis presented in Figure 6.3 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 6.4 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 7

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.

7.1 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.

7.2 Key Findings and Insights

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

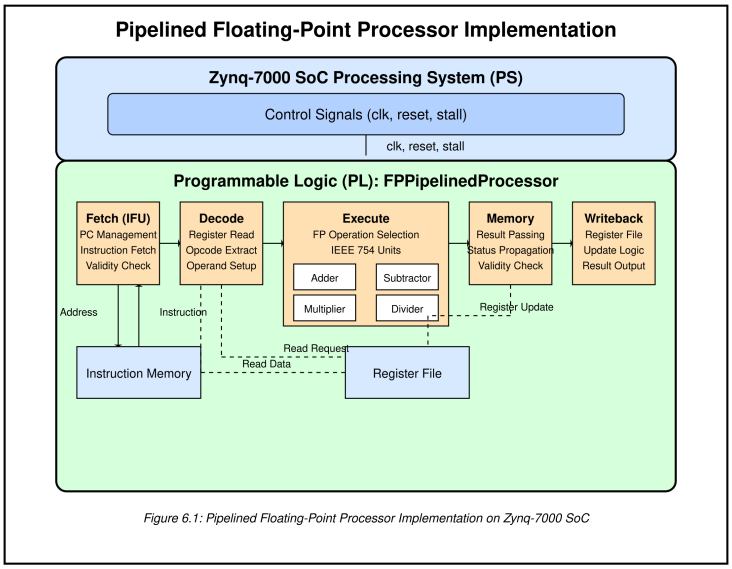

Figure 7.1 – Implemented floating-point pipelined processor architecture overview

- 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.

7.3 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.

7.4 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 8

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, 20).

[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. 7, 23).

[15] B. Bailey u. a., *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann, 2007 (siehe S. 11, 25).

[16] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd. Prentice Hall, 2003 (siehe S. 21, 24).

[17] Xilinx Inc., *7 Series FPGAs Data Sheet: Overview*, DS180 (v2.6), Feb. 2018 (siehe S. 30).

[18] Xilinx Inc., *7 Series FPGAs Configurable Logic Block User Guide*, UG474 (v1.8), Sep. 2016 (siehe S. 30).

[19] Digilent Inc., *Zybo Z7-20 Datasheet*, 2017 (siehe S. 30).

[20] Xilinx Inc., *Vivado Design Suite User Guide: Synthesis*, UG901 (v2021.1), Juni 2021 (siehe S. 31, 32).

[21] C. Maxfield, *The Design Warrior's Guide to FPGAs*. Newnes, 2004 (siehe S. 31).