**Verifying the Modified HL5 RISC-V Processor by Instruction-Level Simulation**

**Objective:** Verify your modified **HLS5 RISC-V processor** using **instruction-level simulation** with **Vivado, Cadence Stratus, and SystemC modules**.

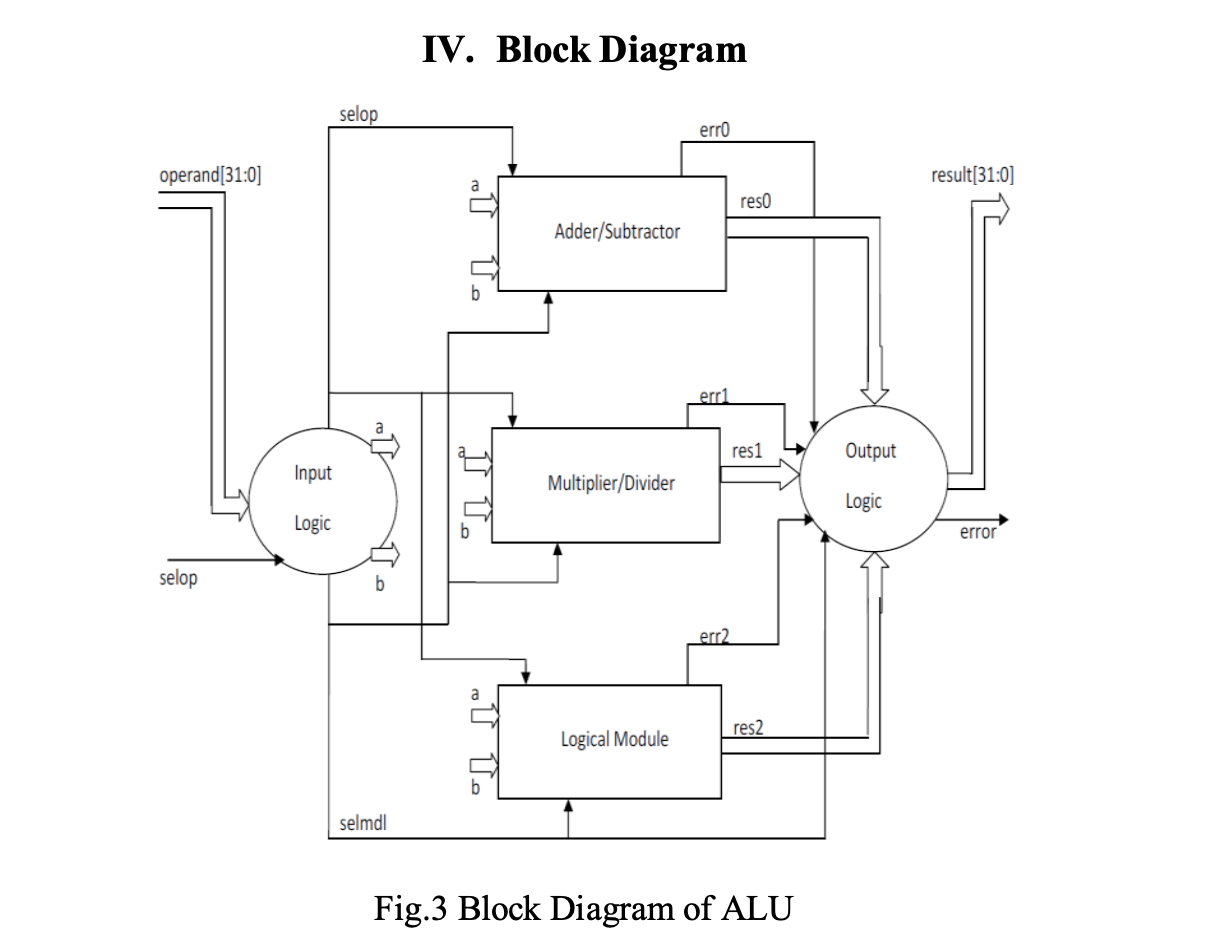
Abstract

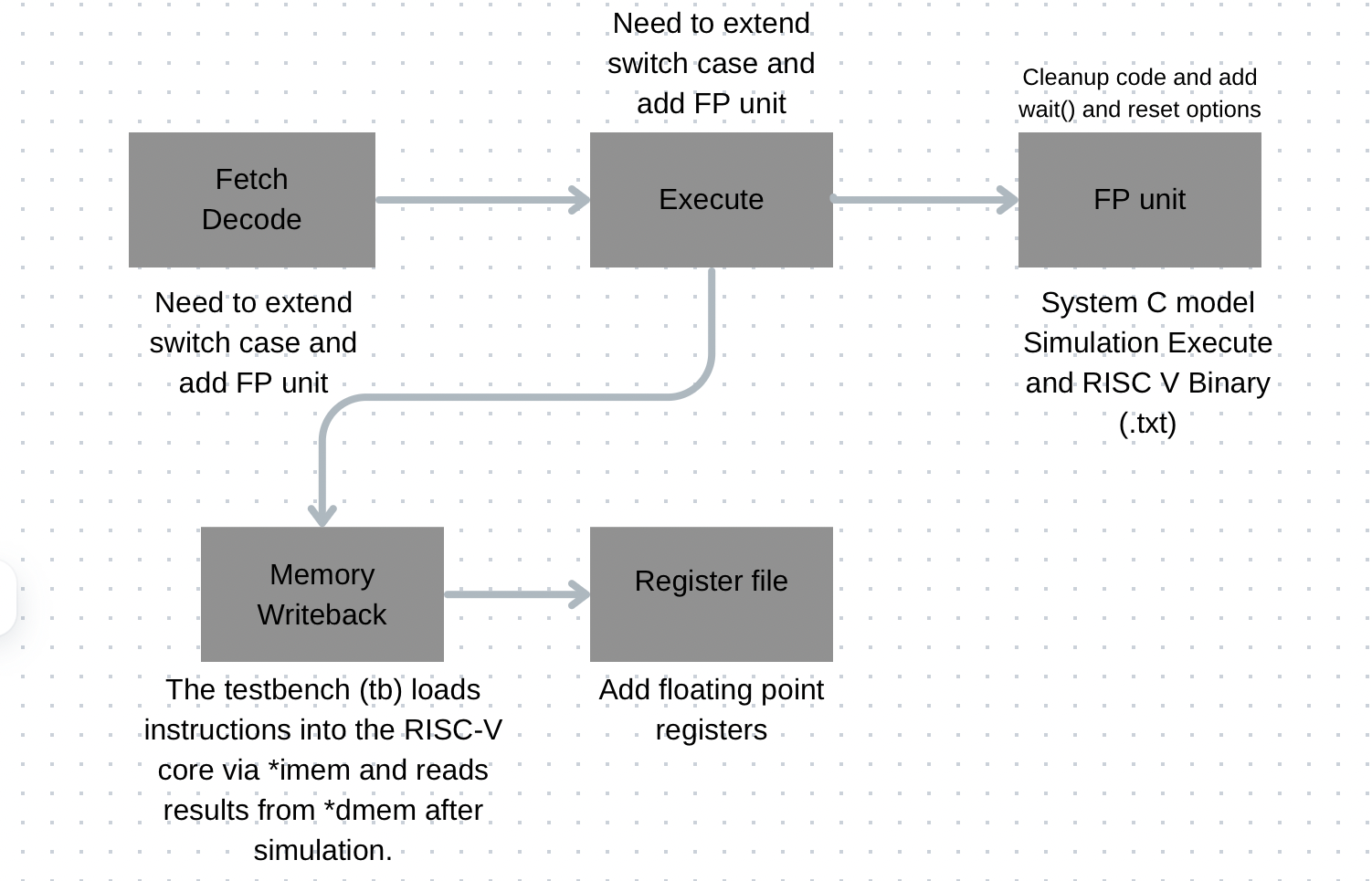
HL5 the first 32-bit RISC-V processor developed entirely with System C and optimized using a commercial HLS tool

The HL5 project is available on GitHub, providing access to the SystemC description of the in-order 32-bit RISC-V core.

[github.com](https://github.com/ic-lab-duth/DRIM4HLS?utm_source=chatgpt.com)

These developments underscore the growing interest and potential of high-level synthesis in the design of efficient and customizable processor architectures.





RISC-V processors have emerged as a pivotal open-standard architecture in modern computing, offering modularity and scalability for diverse applications. The HL5 processor, a RISC-V implementation, serves as the foundation for this work, with architectural modifications aimed at

Problem Statement

Processor verification is critical to ensure functional correctness, safety, and compliance with the ISA. Modified architectures like the HL5 introduce unique challenges, including:

Undetected pipeline hazards or control logic errors.

Inconsistencies in custom instructions or optimizations.

Gaps in simulation coverage masking hardware flaws.

Objectives

This work aims to:

Verify the modified HL5 processor’s correctness against the RISC-V ISA.

Leverage instruction-level simulation to validate register, memory, and control behavior.

Scope & Limitations

Focus: Instruction-level functional verification (excludes gate-level timing or FPGA prototyping).

Boundaries: No cycle-accurate modeling or physical design verification.

"Verifying of the modified HL5 RISC-V processor by instruction-level simulation" was registered by Prof. Dr.-Ing. Dietmar Fey for December 9, 2024.

"I certify that I have prepared this work without outside help and without using sources other than those specified and that this work has not been submitted to any other examination authority in the same or a similar form and has not been accepted by it as part of an examination. All statements that have been adopted verbatim or in essence are marked as such."  
  
The following also applies to theses at the Faculty of Engineering:  
  
1. The work must be submitted to the supervisor by the set deadline in the form of a printed and bound copy and a digital copy (PDF document on storage medium). The latest submission date is June 10, 2025.  
2. Only for the Industrial Engineering degree program (Bachelor's degree): When writing the Bachelor's thesis at WISO, the specifications of the respective department apply for the deadline for submission!  
3. The topic can only be returned once within the deadline specified in the examination regulations.  
4. If any reasons arise during the processing time that affect the proper completion of the work, these must be reported immediately in order to avoid disadvantages.  
5. The master thesis must contain a summary of the results at the end.  
6. Please submit the information sheet for the issue of the certificate to the Examination Office Tech. Fak. Halbmondstr. 6, 91054 Erlangen.  
<https://www.fau.de/files/2018/10/Angaben-fuer-die-Ausstellung-der-Abschlussdokumente.pdf>  
7. When submitting your thesis, you must provide proof by presenting your student ID that you are enrolled as a student at FAU in the semester in which you submit your thesis.

**Tools You Will Use**

1. **Cadence Stratus** (HLS – Convert C/SystemC to RTL)
2. **Vivado** (RTL Simulation & FPGA Synthesis)
3. **Cadence Xcelium** (Advanced RTL Simulation)
4. **SystemC** (For high-level modeling)
5. **RISC-V Toolchain** (Compiler/Assembler for test programs)

graph LR

A[Synthesis] --> B[Optimization]

B --> C[Placement]

C --> D[Routing]

D --> E[Bitstream]

create\_project -force sv\_adder ./sv\_adder -part xc7z020clg484-1

add\_files adder.sv

read\_xdc constraints.xdc

# Required for Zynq

create\_bd\_design "zynq\_bd"

source constraints.xdc

# Implementation flow

synth\_design -top adder

opt\_design

place\_design

route\_design

write\_bitstream -force adder\_sv.bit

exit

**Workflow Summary**

1. **Vitis HLS**:
   * Design hardware in C++ → Get verified RTL
   * *"What my algorithm should do"*
2. **Vivado**:
   * Take RTL → Make it run on actual FPGA hardware
   * *"How my algorithm runs on silicon"*
3. **Key Differences**

| **Feature** | **Vitis HLS** | **Vivado** |
| --- | --- | --- |
| **Input** | C/C++ | Verilog/VHDL |
| **Output** | RTL (Verilog) | Bitstream (.bit) |
| **Optimization** | Algorithmic (pipelines, interfaces) | Low-level (timing, placement) |
| **Simulation** | Co-sim (C++ vs. RTL) | Post-implementation timing sim |
| **When to Use** | Design hardware in C++ | Implement on FPGA |

**Ashwin Varkey**

Friedrich-Alexander University of Erlangen-Nuremberg  
Department Informatik (INF)  
Lehrstuhl für Informatik 3 (Rechnerarchitektur)

**Supervisor:** Prof. Dr.-Ing. Dietmar Fey

**Date of Submission:** 10th June 2025

This thesis presents the verification of a modified **HL5 RISC-V processor** through **instruction-level simulation**, ensuring functional correctness and reliability for future deployments. The primary **objective** is to validate the architectural modifications made to the HL5 core, including custom extensions or optimizations, by rigorously testing its compliance with the RISC-V ISA (Instruction Set Architecture).

The **methodology** employs **instruction-level simulation** using a tailored testbench to execute a comprehensive suite of RISC-V instructions, including edge cases and stress scenarios. Coverage-driven verification techniques are applied to assess the processor’s behavior, with a focus on detecting pipeline hazards, control logic errors, and data path inconsistencies. The simulation framework leverages industry-standard tools (e.g., Verilator, Spike, or custom HDL testbenches) to compare the modified HL5’s outputs against a golden reference model.

**Key results** demonstrate the successful verification of the modified HL5 core, with all implemented instructions functioning as specified. The work identifies and resolves critical design flaws, such as [briefly mention any specific bugs, e.g., "incorrect branch prediction logic" or "data forwarding hazards"]. The **contributions** include:

1. A robust verification framework for RISC-V processor modifications.
2. Documentation of uncovered edge cases and mitigation strategies.
3. Performance benchmarks showing the impact of modifications (e.g., speedup, area efficiency, or reduced power consumption).

This work provides a foundation for future enhancements to the HL5 architecture, ensuring its reliability for academic and industrial applications.

\*\*HL5 Design Methodology\*\*

The HL5 processor implements the RV32IM subset of the RISC-V instruction set architecture (ISA). Key features of the design include:

1. \*\*SystemC Specification:\*\* The processor is described entirely in SystemC, enabling a high level of abstraction. This approach simplifies debugging and design modifications.

2. \*\*HLS Optimization:\*\* Using Cadence Stratus, the SystemC model is synthesized into RTL, with extensive design space exploration (DSE) conducted through HLS configuration parameters (knobs).

3. \*\*Pipeline Architecture:\*\* HL5 features a pipelined design with stages for instruction fetch/decode (fedec), execution (execute), and memory/write-back (memwb). Communication between stages is managed using latency-insensitive channels, supporting flexibility in latency and throughput.

4. \*\*Customization and Scalability:\*\* The design allows easy ISA extensions and performance optimizations without altering the baseline architecture.

System-on-Chip (SoC) design complexity has surged, necessitating advanced design methodologies like HLS, which facilitates faster design cycles through high-level programming languages such as C/C++ and SystemC. 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. The HL5 project aims to address these challenges, demonstrating HLS's potential in creating efficient, customizable processor cores.

- HLS significantly reduces processor design complexity and time.

- HL5 demonstrates that HLS can produce competitive, customizable RISC-V processors.

- The methodology promotes easier ISA extensions and design modifications.

The HL5 project illustrates that HLS can effectively be used for general-purpose processor design, achieving performance levels comparable to traditional RTL-based designs while offering enhanced customization and reduced development effort. The study identifies current HLS tool limitations, such as handling control logic and memory dependencies, and suggests areas for future research and tool improvement.

"Does instruction-level simulation sufficiently verify the modified HL5 processor?"

"While instruction-level simulation (ILS) is a powerful tool for verifying the functional correctness of the modified HL5 RISC-V processor—ensuring that each instruction behaves as specified by the ISA—it has inherent limitations that may affect verification sufficiency. ILS excels at detecting logic errors in the data path, control flow, and register-file interactions, and it can validate custom extensions through targeted test cases. However, it typically abstracts away timing, microarchitectural features (e.g., pipeline hazards, speculative execution), and physical effects (e.g., power/area trade-offs). For thorough verification, ILS should be complemented with cycle-accurate simulation, formal methods (e.g., model checking), and FPGA prototyping to cover timing-dependent behavior and edge cases. Thus, while ILS is necessary for baseline validation, it is not wholly sufficient for guaranteeing robust correctness in all operational scenarios."

Instruction-Level Simulation (ILS) is a technique in computer architecture and software development that simulates program execution one instruction at a time, meticulously modeling changes to registers, memory, and CPU state. Unlike cycle-accurate simulation, ILS focuses on granular instruction effects, making it invaluable for debugging, performance analysis (e.g., instruction counts, cache misses), hardware verification (validating CPU designs), and security research (malware analysis). Simulators range from functional (e.g., QEMU, Gem5 in syscall mode) to timing-accurate (modeling pipelines/caches), with tools like SPIM (MIPS), Unicorn Engine, and GDB enabling step-by-step execution. While ILS offers high accuracy and hardware-free testing, it sacrifices speed and may omit low-level hardware behaviors (e.g., speculative execution side effects). Despite these trade-offs, ILS remains essential for bridging software abstraction with hardware implementation, ensuring correctness in embedded systems, processor design, and reverse engineering.

Instcruction level simulation has become an essential tool in processor verification offering insights into behaviour of individual instructions as they pass through processor pipeline . ILS provide an effective mechanism for checking functional correctnessby simulating the processor’s behaviour at the granularity of individual instructions.

Cadence Startus High level Synthesis automatically creates high quantiy register transfer level design implemnetations for ASIC, System on chip and FPGA targets from high level IEEE 1666 System C

With Startus HLS , we can quickly design and verify high quality RTL implementation

One short TCL file can configure and automate all of the following C++ compiliation and linking, Sytsem C and RTL simulation , Analysis and debugging of System C , waveform analysis

HL5 doesn’t simulate individual instrcutions at microarchitectural level. It focuses on functional behaviour and performance estiomation. HL5 is typically used for exploring microarchiutectural changes evaluating ISA modification and is faster than instruction level

Spike and QEMU instruction level

Create a Vivado project

Add .sv files

Add a testbench

Run simulation

Option B: Synthesis & FPGA Implementation

Check Synthesizability:

tcl

Copy

synth\_design -top riscv\_top -part xc7z020clg400-1

Fix errors (e.g., unsupported constructs).

Generate Bitstream:

tcl

Copy

place\_design

route\_design

write\_bitstream -force riscv.bit

When transitioning from SystemC to RTL (Verilog/VHDL), certain SystemC constructs like sc\_module and sc\_signal are unsupported and must be manually converted to equivalent Verilog modules and logic signals. Debugging common issues involves addressing simulation mismatches (e.g., behavioral differences between SystemC and RTL by comparing waveforms in tools like Vivado’s Waveform Viewer or inserting $display statements to log register values) and resolving clock/reset problems (e.g., registers not updating due to improper connections, fixed by ensuring correct sensitivity lists and reset conditions, as shown in the example always @(posedge clk or posedge reset) block). These steps ensure accurate RTL implementation while navigating the limitations of the design flow.

**Workflow Summary**

1. **Convert SystemC → Verilog (.sv)** using ICSC.
2. **Clean up .sv files** (remove non-synthesizable code).
3. **Simulate in Vivado** with a testbench.
4. **(Optional) Synthesize for FPGA** if the RTL is compatible.

| **Aspect** | **Vivado Compatibility** |
| --- | --- |
| **SystemVerilog Support** | ✅ Full support for .sv files. |
| **SystemC Constructs** | ❌ No direct support (must be converted). |
| **FPGA Synthesis** | ✅ Possible if RTL is clean. |
| **Simulation Speed** | ⚠️ Slower than Cadence Xcelium. |

The choice between Option A (functional simulation via Vivado/QEMU) and Option B (cycle-accurate simulation via Cadence Xcelium) depends on verification depth and design goals. Option A focuses on instruction correctness, validating register/memory updates and FPU/ALU logic (e.g., ensuring fadd.s produces the right result) without timing details—ideal for early-stage debugging or FPGA targets where timing is resolved later. In contrast, Option B models cycle-exact behavior, including pipeline hazards, multi-cycle latencies, and microarchitectural effects (e.g., stalls, data dependencies), making it critical for ASIC tapeouts or designs with aggressive optimizations (OoO execution, speculative pipelines). While Option A is faster and sufficient for basic validation, Option B is necessary when verifying timing-critical paths or complex processor modifications. A practical workflow starts with Option A to catch logical errors, then adopts Option B for final sign-off on timing and microarchitecture.

Suppose your HLS5 RISC-V has:

* A **5-stage pipeline** (IF, ID, EXE, MEM, WB).
* A **multi-cycle FPU** (takes 3 cycles to complete fadd).

**Option A (Vivado)** will show the **final result** of fadd, but:

* ❌ Misses **stalls** caused by FPU latency.
* ❌ Doesn’t catch **data hazards** (e.g., back-to-back FPU ops).

**Option B (Xcelium)** will:

* ✅ Reveal pipeline bubbles due to FPU latency.
* ✅ Flag RAW (Read-After-Write) hazards if registers aren’t forwarded correctly.

The document titled "Intel Compiler for SystemC" authored by Mikhail Moiseev in March 2021 provides a comprehensive overview of the Intel® Compiler for SystemC (ICSC), a tool designed to translate cycle-accurate SystemC code into synthesizable SystemVerilog. Below is a detailed explanation of the document's content, organized by key sections:

* **ICSC Functionality**: The tool translates SystemC code into SystemVerilog, aiming to enhance the productivity of design and verification engineers.
* **Key Differences**

| **Aspect** | **Synthesizable Verilog** | **Explicit Verilog** |
| --- | --- | --- |
| **Purpose** | Hardware implementation (FPGA/ASIC) | Simulation, testing, and verification |
| **Constructs** | Limited to synthesizable constructs | Can include non-synthesizable constructs |
| **Target** | Synthesis tools | Simulation tools (e.g., ModelSim, VCS) |
| **Abstraction Level** | Low-level (close to hardware) | High-level (behavioral modeling) |
| **Example Use** | Designing actual hardware modules | Writing testbenches or behavioral models |

The RISC GNU ELF GCC Compiler converts C/C++ code into executable machine code (.elf files) for RISC processors like RISC-V, used in embedded software development. In contrast, HLS/Synthesis compilers (like Vivado HLS or Quartus) transform high-level code (C++/SystemC) or RTL (Verilog/VHDL) into hardware circuits for FPGAs/ASICs. While GCC produces software binaries that run on CPUs, HLS outputs hardware netlists/bitstreams that configure physical logic gates. GCC is for processor programming; HLS is for hardware creation.

**Why Do People Generate .bit Files?**

A .bit file is a **binary configuration file** for FPGAs. It’s used to:  
✅ **Program the FPGA** (load your design onto hardware)  
✅ **Verify timing & power** (after place-and-route)  
✅ **Debug on real hardware** (with ILA/chipscope)

create\_project -force sv\_adder ./sv\_adder -part xc7a100tcsg324-1

add\_files adder.sv

synth\_design -top adder

exit

**Key Tips for RTL Design:**

* **Keep it simple and modular.**
* **Avoid latches** by ensuring all branches of conditional statements are covered.
* **Use non-blocking assignments (<=) for sequential logic** and blocking assignments (=) for combinational logic.
* **Follow coding guidelines** for readability and maintainability.

Design and Implementation of a 64/32-bit Floating-point Division,

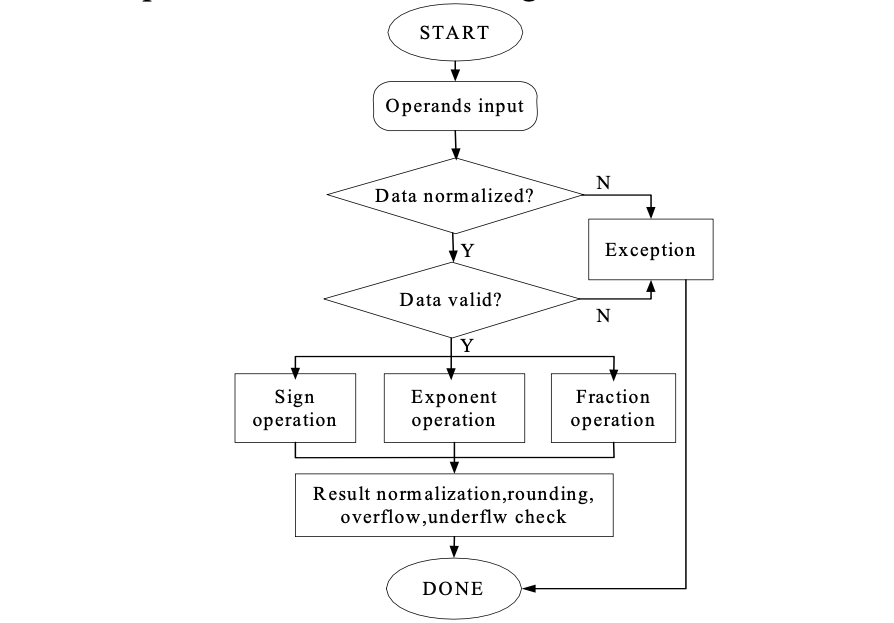
Reciprocal, Square root, and Inverse Square root Unit

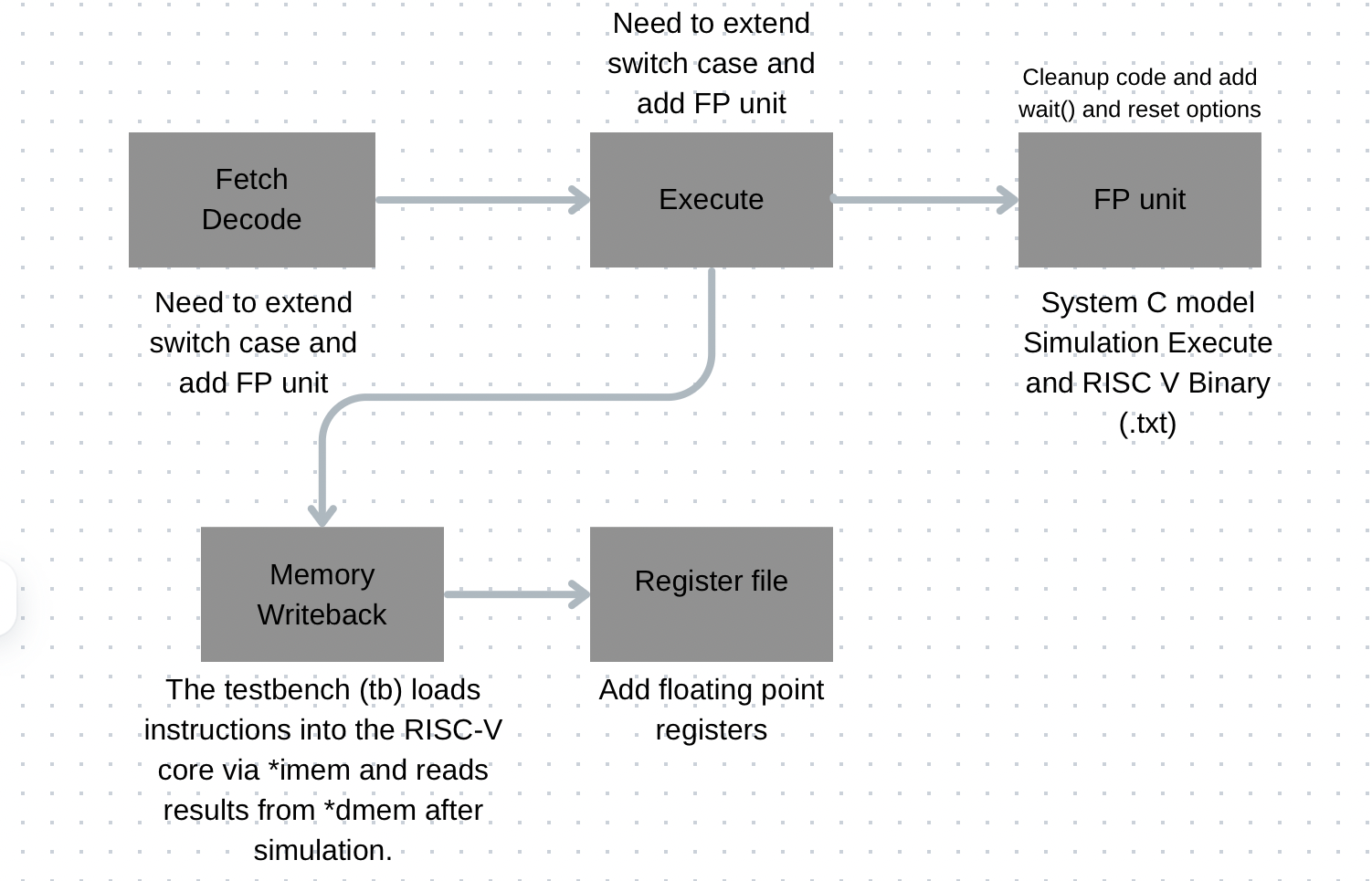
Chen Shuang-yan(1) (2), Wang Dong-hui(1), Zhang Tie-jun(1), Hou Chao-huan(1)

(1)(Institute of Acoustics, Chinese Academy of Sciences, 21, Beisihuanxi Road, Beijing, China, 100080)

(2)(Graduate School of Chinese Academy of Sciences Beijing, China,100039)

Email [shuangyanchen@gmail.com](mailto:shuangyanchen@gmail.com)





In the computer arithmetic, operations like add, sub

and multiplication are much more frequently used than

other operations. With the fast development of

sub-micron technology, the density of silicon grows

rapidly and the cost of hardware decreases, more and

more functionalities are transferred into hardware to

realize. In order to meet the ever-increasing demand in

high performance applications including scientific

computations, digital signal processing, computer

graphics, multimedia, etc the performance of division,

reciprocal, square root and inverse square root

computation is becoming more and more important.

This paper presents an efficient design and

implementation of a configurable multifunctional

computation unit for division, reciprocal, square root and

inverse square root, which is fully compatible with the

IEEE754 standard.

In computing, an arithmetic logic unit (ALU) is a digital circuit that performs arithmetic and logical

operations. The ALU is a fundamental building block of the central processing unit (CPU) of a computer, and

even the simplest microprocessors contain one for purposes such as maintaining timers. The processors found

inside modern CPUs and graphics processing units (GPUs) accommodate very powerful and very complex

ALUs; a single component may contain a number of ALUs. ALU is a necessity for a computer because it is

guaranteed that a computer will have to compute basic mathematical operations, including addition, subtraction,

multiplication, and division.

**2.1 Floating Point Unit**

When a CPU executes a program that is calling for a floating-point (FP) operation, there are

three ways by which it can carry out the operation. Firstly, it may call a floating-point unit emulator, which is a

floating-point library, using a series of simple fixed-point arithmetic operations which can run on the integer

ALU. These emulators can save the added hardware cost of a FPU but are significantly slow. Secondly, it may

use an add-on FPUs that are entirely separate from the CPU, and are typically sold as an optional add-ons which

are purchased only when they are needed to speed up math-intensive operations. Else it may use integrated FPU

present in the system . The FPU designed by us is a single precision IEEE754 compliant integrated unit. It can

handle not only basic floating point operations like addition, subtraction, multiplication and division but can also

handle operations like shifting, logical operations.

Floating point ALUs are used for high precision computing. This ALU uses 32 bit numbers, which is

the common computer word length. The numbers are represented in IEEE 754 standard. This standard is widely

used in floating point arithmetic. The ALU can perform the arithmetic operation: addition, subtraction,

multiplication and division. It can also perform some logical operations such as logical NOT, logical NAND and

shift right. To support RISC design the arithmetic operations are done in pipelined fashion. There are two

pipelined units in this ALU: the add/subtract pipeline and mult/div pipeline. The addition and subtraction

operations are done in the add/subtract pipeline and the multiplication and division operations are done in the

mult/div pipeline

IEEE754 standard is a technical standard established by IEEE and the most widely used standard for

floating-point computation, followed by many hardware (CPU and FPU) and software implementations [3].

Single-precision floating-point format is a computer number format that occupies 32 bits in a computer memory

and represents a widedynamic range of values by using a floating point. In IEEE 754-2008, the 32-bit with base

2 format is officially referred to as single precision or binary32. It was called single in IEEE 754-1985. The

IEEE 754 standard specifies a single precision number as having sign bit which is of 1 bit length, an exponent of

width 8 bits and a significant precision of 24 bits out of which 23 bits are explicitly stored and 1 bit is implicit 1.

Sign bit determines the sign of the number where 0 denotes a positive number and 1 denotes a negative

number. It is the sign of the mantissa as well. Exponent is an 8 bit signed integer from -128 to 127 (2's

Complement) or can be an 8 bit unsigned integer from 0 to 255 which is the accepted biased form in IEEE 754

single precision definition. In this case an exponent with value 127 represents actual zero. The true mantissa

includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point)

with value 1 unless the exponent is stored with all zeros. Thus only 23 fraction bits of the mantissa appear in the

memory format but the total precision is 24 bits. IEEE754 also defines certain formats wh

**3.1 Conversion Of The Binary Integer To Its Ieee754 Format**

As our FPU is IEEE754 compliant, the next step is to convert the input (here the effective operand into

the IEEE specified format. IEEE754 single precision can be encoded into 32 bits using 1 bit for the sign bit (the

most significant i.e. 31st bit), next eight bits are used for the exponent part and finally rest 23 bits are used for

the mantissa part.

For example:

S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF

31 30 23 22 0

However, it uses an implicit bit, so the significant part becomes 24 bits, even though it usually is

encoded using 23 bits.

This conversion can be done using the below algorithm of above example

Step1: Sign bit of the binary number becomes the sign bit (31st bit) of the IEEE equivalent.

Step 2: 30th bit to 8th bit of the binary number becomes the mantissa part of the IEEE equivalent.

Step 3: The exponent part is calculated by subtracting the position of the 1st one obtained in the algorithm

described in section 2.2.1.

Step 4: A bias of 127 is added to the above exponent value.

DOI: 10.9790/4200-05515459 www.iosrjournals.org 55 | Page*Design and Implementation of Floating Point ALU with Parity Generator Using Verilog HDL*

**3.2 Pre-Normalization Of The Operands**

Pre-normalization is the process of equalizing the exponents of the operands and accordingly adjusting

the entire IEEE754 expression of the inputs to produce correct results maintaining the IEEE754 standard

throughout all calculation steps inclusive of the intermediate calculations and their outputs.

This conversion can be done using the below algorithm

Step 1: Insert the implicit 1 in the mantissa part of each of the operands.

Step 2: Find positive difference between the exponents of the operands

Step 3: Set the lower operand.s exponent same as that of the operand with higher exponent.

Step 4: Right shift mantissa of the lower operand by steps equal to difference calculated.

**3.3 Performing The Selected Operation**

After completion of the preliminary steps the next step is to perform the actual operation. The choice of

operation is taken as input via a 4 bit wire oper. Following is the table 2.1 that describes the functions and their

corresponding operation code.

fpu\_op Operation

000 Add

001 Subtract

010 Multiply

011 Divide

100 Shifting

**3.4 Module Add**

Addition is a mathematical operation which represents combining a collection of objects together to

form larger collection. The process of developing an efficient addition module in our FPU was an iterative

process and with gradual improvement at each attempt.

**3.4.1 Add Using The ¡°+¡± Operator**

The initial attempt was to add using the simple in-built ¡°+¡± operator available in Verilog library. It

used a 23 bit register sum and a 1 bit register Co (for carry). The algorithm for the addition can be described

below

Step 1: Check if oper = 4.b000

Step 2: {Co,Sum} = Temp\_op1\_ieee[22:0] + Temp\_op2\_ieee[22:0]

Step 3: If carry is 1, then. Resultant\_exponent = Larger\_exponent + 1;

Else if carry is 0, then do

Resultant\_exponent = Larger\_exponent . (21-difference) (difference as in sec.2.2.3)

Step 4: Check for overflow and underflow-

.If for any of the operands (sign(operand with greater exponent)==0 & (exp\_greater + 1 > 255)) then, Set the

overflow flag to 1.

Else if (sign(operand with lesser exponent==0) & (exp\_lesser<0)), then set the underflow flag to 1

Step 5: Aggregate the result as concatenation of {Sign\_bit,Resultant\_exponent,Sum}

**3.5 Subtract Module**

Subtraction is an operation which is treated as inverse of addition operation. The process of developing

an efficient SUB module followed the iterative development of the ADD module.

**3.5.1 Sub Using The ¡°-¡± Operator**

The initial attempt was to subtract using the simple in-built ¡°-¡± operator available in Verilog library. It

used a 23 bit register diff and a 1 bit register borrow (for borrow). The algorithm for the subtraction module can

be described.

Step 1: Check if oper = 4.b0001

Step 2: {borrow.diff} = Temp\_op1\_ieee[22:0] - Temp\_op2\_ieee[22:0]

Step 3: Resultant\_exponent = Larger\_exponent + (21-difference)

Step 4: Check for overflow and underflow-

If for any operand (sign(operand with greater exponent)==1 AND (exp\_greater + 1 < 0))

DOI: 10.9790/4200-05515459 www.iosrjournals.org 56 | Page*Design and Implementation of Floating Point ALU with Parity Generator Using Verilog HDL*

Set the overflow flag to 1

If for any operand (sign(operand with exponent)==1'b1 AND (exp\_lesser>8'd255))

Set the underflow flag to 1

Step 5: Aggregate the result as concatenation of {Sign\_bit,Resultant\_exponent,diff}

**3.6 Multiplication Module**

The process of developing an efficient multiplication module was iterative and with gradual

improvement at each attempt. The product of two n-digit operands can be accommodated in 2n-digit operand.

**3.6.1multiplication Using ¡°\*¡± Operator**

It used a 47 bit register to store the product. The algorithm is explained

Step 1: Check if oper = 4.b0010

Step 2: product = Temp\_op1\_ieee[22:0] \* Temp\_op2\_ieee[22:0]

Step 3: Resultant\_exponent = op1\_ieee[30:23] + op2\_ieee[30:23] - 127

Step 4: If for product ( Resultant\_exponent >255 ), then do,

. Set the overflow flag to 1

Step 5: Sign\_bit = op1\_ieee[31] ^op2\_ieee[31]

Step 6: Aggregate the result as concatenation of { Sign\_bit, Resultant\_exponent, product }

**3.7 Module Division**

Division is regarded as the most complex and time-consuming of the four basic arithmetic operations.

Given two inputs, a dividend and a divisor, division operation has two components as its result, quotient and a

remainder.

**3.7.1 Division Using ./. Operator**

The initial attempt was to divide two numbers using the simple in-built ¡°/¡± operator available in

Verilog library. It used a 32 bit result\_div\_ieee register to store the quotient and register remainder to store the

remainder of the division operation.

Step 1: Check if the oper = 4 bit 0100

Step 2: result\_div\_ieee = temp\_op1\_ieee[22:0] / temp\_op2\_ieee[22:0]

Step 3: If op2\_ieee[30:0] is all 0

. Set div\_bby\_zero flag to 1

Step 4: Aggregate the result as concatenation of {Sign\_bit, Resultant\_exponent, result\_div\_ieee}

**IV. Block Diagram**

Fig.3 Block Diagram of ALU

In the ALU top module there are in total 70 I/O pins. It include 32 pins for input operand , 32 pins for

output result, 3 pins for selop, 1 pin for rst,1 pin for clk and 1 output pin for error. For actual arithmetic

DOI: 10.9790/4200-05515459 www.iosrjournals.org 57 | Page*Design and Implementation of Floating Point ALU with Parity Generator Using Verilog HDL*

calculations we need two operands but in a microprocessor there is only one databus, and we have to multiplex

this databus. If selop is 000, it is considered as first operand and if selop have any other values, then the operand

that present at that time is considered as the second operand.

Selop Operation

000 Select operand „a‟

001 Add operand „a‟ and „b‟

010 Subtract operands „a‟ and „b‟

011 Multiply operands „a‟ and „b‟

100 Divide operands „a‟ and „b‟

101 Logically NOT operand „a‟

110 Logically NAND operand „a‟ and „b‟

111 Shift right the operand „a‟

**4.2 Block diagram description**

The floating point ALU is internally divided in to three logical modules: the adder/subtracter

module, the multiplier/divider module and the logical module. All the input parameters are given to the input

logic and also the output is taken from the output logic. The input parameters to the ALU include operand,

selop, clk and reset. The output from the ALU includes 32 bit result and error signal. Here selop is used to selet

an operation. In other words the selop is equivalent to the instruction set of a microprocessor. The selop is three

bit wide, and so there are totally seven different operations are possible. All these operations are given in the

table.

**4.3 Input Logic**

The input logic receives two 32 bit operands one by one and store it in registers a and b. According to

the selop signal, the input logic produces the semdl signal to activate any one of the logical module.

**4.4 Adder/Sbtracter**

The the adder/subtracter module performs the addition and subtraction operations according to the selop

signal. Internally it is a four level pipelined add/sub unit.

**4.5 Multiplier/Divider**

The the multiplier/divider module performs the multiplication and division operations according to the

selop signal. Internally it is a four level pipelined add/sub unit.

**4.6 Logical Module**

The logical module can perform logical NOT, logical NAND and logical right shift operations. These

operations are performed according to the selop signal.

**4.7 Output Logic**

The output logic accepts the three results res0, res1 and res2 respectively from adder/subtracter,

multiplier/divider and logical module. It directs any one of them to the final result according to the selop signal.

<https://www.cadence.com/en_US/home/resources/datasheets/stratus-high-level-synthesis-ds.html>

<https://www.youtube.com/watch?v=LKB5I12LctU>

<https://www.youtube.com/watch?v=m8DqCTogb8w>

<https://github.com/johnwinans/rvalp?tab=readme-ov-file>

<https://msyksphinz-self.github.io/riscv-isadoc/html/rvfd.html#flt-d>

<https://xilinx.github.io/xup_high_level_synthesis_design_flow/Lab1.html>

**Design and Implementation of Floating Point ALU with Parity**

**Generator Using Verilog HDL**

Mohammad Ziaullah1

, Abdul Munaff2

*1(Asst. Professor, Electronics and Communication Engineering, S.I.E.T Vijayapur, Karnataka, India)*

*2(Asst. Professor, Electrical and Electronics Engineering, S.I.E.T Vijayapur, Karnataka, India)*

Abstract: A floating-point unit (FPU) colloquially is a math coprocessor, which is a part of a computer system

specially designed to carry out operations on floating point numbers. Typical operations that are handled by

FPU are addition, subtraction, multiplication and division. The functions performed are handling of Floating

Point data, converting data to IEEE754 format, perform any one of the following arithmetic operations like

addition, subtraction, multiplication, division . All the above algorithms have been evaluated under Modelsim

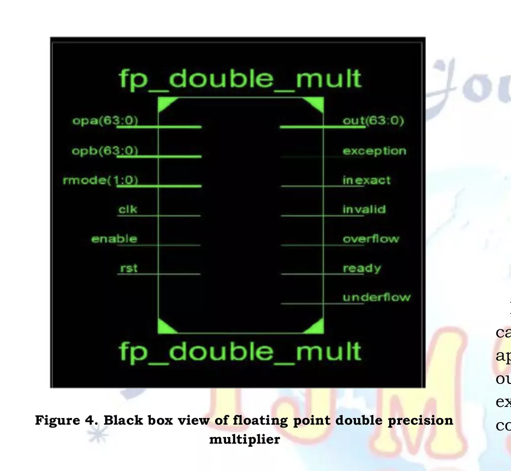
environment. All the functions are built by possible efficient algorithms with several changes incorporated at

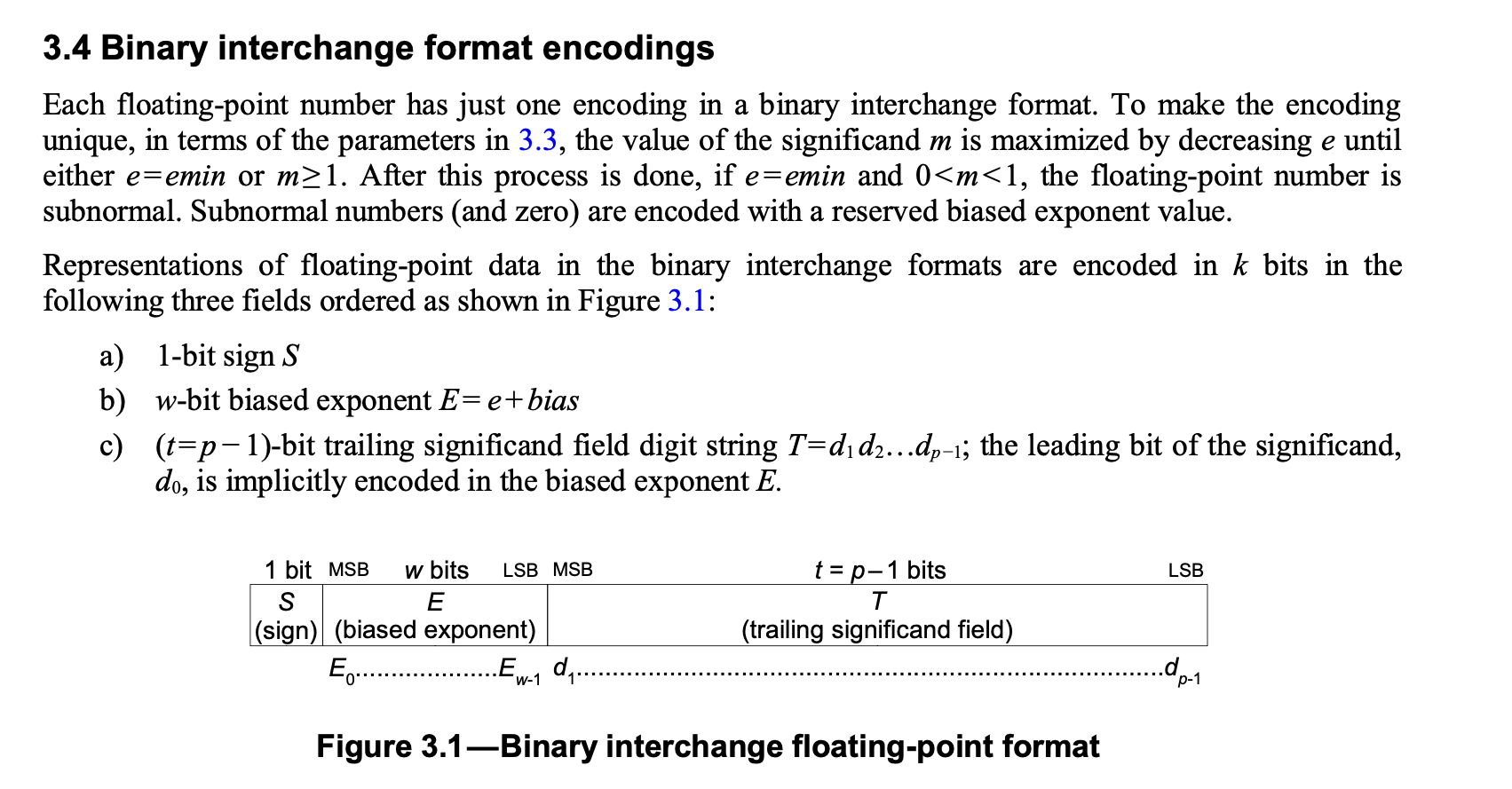
our end as far as the scope permitted. Consequently all of the unit functions are unique in certain aspects and

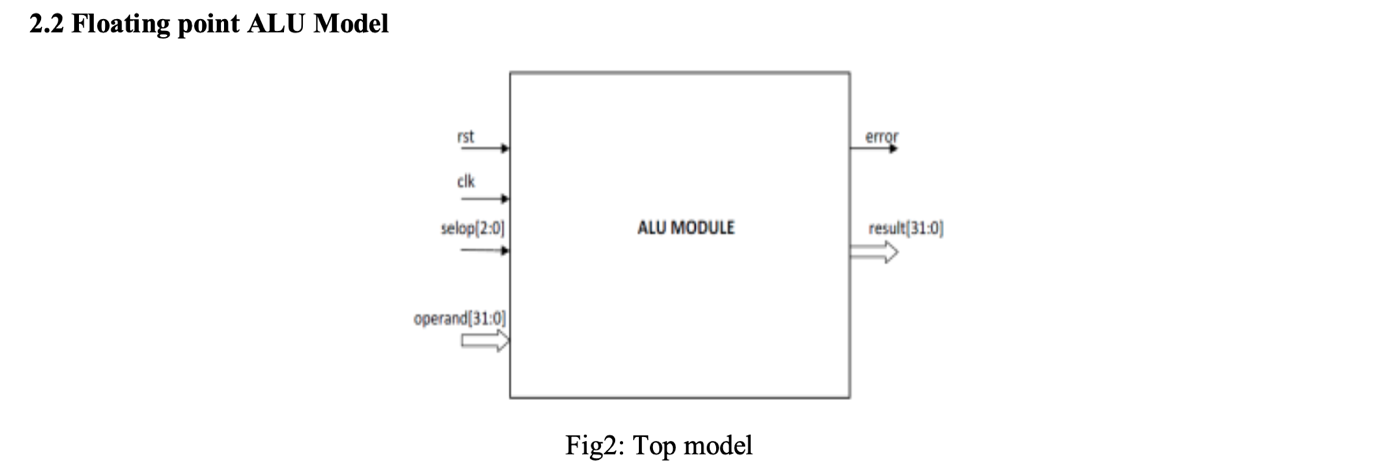
given the right environment these functions will tend to show comparable efficiency and speed ,and if pipelined

then higher throughput.

Keywords: Floating, Algorithm, coprocessor, pipelined, throughput.







module adder (

    input wire [3:0] a,

    input wire [3:0] b,

    output reg [4:0] sum

);

    always @(\*) begin

        sum = a + b;  // This will synthesize to a 4-bit adder

    end

endmodule

ICSC is a SystemC 2.3.3-compliant tool that performs lightweight source-to-source translation from synthesizable SystemC/C++17 (including STL containers like vector and array) to high-quality SystemVerilog while preserving original code structure and enabling traceable debugging of generated RTL. The Linux-based tool requires C++17, CMake and Git, supports dynamic design elaboration for IP reuse, and plans future enhancements for fixed/floating-point operations and clocked thread retiming, differentiating itself from traditional HLS tools through its focus on direct, efficient code translation.

**2. Cadence Stratus for HLS + Instruction-Level Analysis**

**Stratus** is a **High-Level Synthesis (HLS)** tool that converts C/C++/SystemC into RTL. While it doesn’t simulate instructions directly, it helps in:

* **Analyzing loop pipelining** (how software translates to hardware).
* **Checking control flow** (branches, function calls).

**Flow for Instruction-Aware HLS Simulation**

1. **Write C/C++ code** (e.g., a DSP algorithm).
2. **Run Stratus HLS** to generate RTL.
3. **Simulate RTL in Xcelium** to see how instructions map to hardware.

**Summary of Steps**

| **Step** | **Tool** | **Task** |
| --- | --- | --- |
| 1 | **Cadence Stratus** | Convert SystemC RISC-V → RTL |
| 2 | **RISC-V Toolchain** | Compile test programs (.hex) |
| 3 | **Vivado Simulator** | Basic instruction-level verification |
| 4 | **Cadence Xcelium** | Cycle-accurate simulation + coverage |
| 5 | **SystemC Model** | Golden reference comparison |
| 6 | **Vivado (FPGA)** | Hardware validation (optional) |

To implement a modified RISC-V processor, first convert your SystemC/C++ design into synthesizable RTL using Cadence Stratus HLS by running stratus\_hls -f riscv\_hls\_config.tcl, where the TCL script defines clock targets and synthesis constraints, producing verified Verilog output (riscv\_hls.v). Concurrently, prepare test programs by writing RISC-V assembly (e.g., basic arithmetic and load/store operations), then compile them to executable binaries using the RISC-V GNU toolchain (riscv64-unknown-elf-gcc for ELF generation and objcopy for Verilog hex format), creating test.hex files for subsequent RTL simulation in tools like Vivado or Xcelium.

Nowadays, an FPGA plays the role of the personal computer of the 70s: It

gives access to the logic gates. The High-Level Synthesis tool (HLS) plays the role

of the C compiler of the 70s: It gives access to the FPGA through a high-level

language.

Cadence® Stratus™ High-Level Synthesis (HLS) is a platform that enables engineering teams to design and verify high-quality RTL implementations from abstract SystemC™, C, or C++ models. The Stratus integrated design environment (IDE) facilitates the creation of these models by providing pre-defined design templates, simplifying the design process and reducing development time.

The Stratus analysis environment includes SystemC and RTL source linking, control and dataflow graphs, schematic viewer, and pipeline analysis, as well as QoR reporting and visualization to judge the impact of architectural optimizations. Although most commonly used via the GUI, this analysis is also available via the Stratus Tcl API.

Stratus HLS supports untimed and timed SystemC and C++ models, including a mix of both, providing maximum flexibility to the designer. The output can be fully pipelined (new data each cycle), pipelined at reduced throughput (new data every n cycles), or unpipelined. It also supports multiple pipeline stalling modes, including hard stalling and soft stalling, as well as pipeline draining and bubble squashing. Stratus HLS ensures the generated RTL finite-state machine (FSM), datapath, and memory interfaces correctly implement even the most complex stalling scenarios.

**4. Literature Review**

* **Processor Verification Techniques**:
  + Simulation-based, formal methods, emulation.
* **RISC-V Verification Efforts**:
  + Prior work on verifying RISC-V cores (e.g., Rocket Core, BOOM).
* **Instruction-Level Simulation**:
  + Tools like Spike, QEMU, or custom simulators.

**Processor Verification Methodologies**  
Modern processor verification employs three principal approaches: (1) simulation-based methods (dynamic verification through test cases), (2) formal methods (mathematical proof of correctness), and (3) hardware emulation (FPGA-based prototyping). Each technique offers distinct trade-offs between coverage depth and execution time, with simulation remaining the most widely adopted for functional verification.

Instruction-Level Simulation Frameworks  
Current practice utilizes several instruction-accurate simulation tools:

Spike (RISC-V reference simulator) provides golden-model reference

QEMU enables fast functional emulation

Custom simulators (e.g., Google's RISCV-DV) offer specialized verification capabilities  
Recent studies highlight growing adoption of hybrid approaches combining instruction-level simulation with formal property checking.

Obtain the two numbers to be added in IEEE-754 single precision format.

The number will be of the form: {sign (1-bit), exponent (8-bits), mantissa (23-bits)} = Total 32-bits

To automate this number conversion, you can use the [h-schmidt converter](https://www.h-schmidt.net/FloatConverter/IEEE754.html).

Note: When we say mantissa, it does not consider the 'implicit 1' that is present before the decimal point. However for computations, we include the mantissa with the 'implicit 1', perform the required computations and then discard the implicit 1.

For the below algorithm, the word mantissa represents {1'b1, 23-bit mantissa} = 24-bits

1. Compare the exponents of the two numbers. Find the absolute difference (abs\_diff) of the two exponents.
2. Now shift right the mantissa of the number with smaller exponent by abs\_diff times so that now the exponents become equal. This is the exponent value of the sum.
3. Check the sign bit of the two numbers. If both are positive, add the two mantissas. If negative, then subtract the mantissas accordingly. If borrow has occurred during subtraction, take 2s complement of the mantissa.
4. Call the obtained mantissa as sum\_mantissa. The sum will contain 25-bits to accommodate overflow.
5. Now the sum\_mantissa needs to be normalized if needed. As we know, the sum\_mantissa includes the implicit 1. This implicit 1 needs to be at bit location sum\_mantissa[23] to comply with IEEE-754 format. If sum\_mantissa[23] = 1, then no normalization is required.
6. For achieving this, search for the first 1 occurring in sum\_mantissa traversing right from MSB. Shift sum\_mantissa in such a way that this 1 occurs at location sum\_mantissa[23]. According to the shifts, the exponent of sum must also be adjusted.
7. Now discard the implicit 1 and consider only the lower 23-bits of sum\_mantissa. This will be the final value of sum\_mantissa.
8. Thus we have calculated the sign, exponent and mantissa of the sum. Combining the three in IEEE-754 format, we will have obtained the final result.

A floating-point unit (FPU), also referred to as numeric co-processor, is a dedicated processing unit that manipulates numbers more quickly than the basic microprocessor circuit. The FPU does this by means of instructions that focus entirely on large mathematical operations such as addition, subtraction, multiplication, or division etc. The F-extension added in user project area, also referred to as FPU is capable to perform floating point addition, subtraction, multiplication, division, square-root and fused multiplication.The block diagram is shown in figure 1.1.

https://github.com/Lampro-Mellon/Caravel\_FPU

FPU Architecture

The designed floating-point unit (FPU) is fully compliant with the IEEE-754 standard. It flags all five exceptions explained in subsection xx during result calculation as guided by IEEE-754 spec and has support to round result as per the five rounding modes. Additionally support for dynamic rounding is also added as indicated in RISC-V specifications. The rounding modes are explained in table 1.1. For being compliant with RISC-V F-Extension the unit is capable of performing basic arithmetic operations like comparison, addition, subtraction and multiplication. Moreover, the support for certain complex operations like division, square root of a number and computation of fused multiply-add. Certain other computational operations like integer to float conversion, float to integer conversion, computation of minimum and maximum of the two numbers along with sign injection is added. Caravel SoC core acts as a Wishbone master providing IP with control signals like data, address, cycle and strobe. The signals write FPU CSRs (explained in section 1.2.1) are decoded to provide information including operand A, operand B and operand C. Additionally, the operation that is to be performed and the rounding mode to be used are also provided. The selected module performs operation and the output result is written to an FPU CSR along with the exceptions flagged. The result and exceptions are then forwarded by wb\_s to the core when requested.

FPU Exceptions

FPU can flag five exceptions depending upon the result during computation. 1. Invalid operation (NV) The exception is raised if the operation performed is mathematically incorrect. 2. Divide by zero (DZ) This exception is flagged only by division module and as the name suggests it is raised when the divider is zero. 3. Overflow (OF) If the computed result exceeds the maximum re-presentable range of single precision the exception is raised. 4. Underflow (UF) If the result is smaller than the minimum representation of the single precision then UF exception is set high. 5. Inexact (NX) Whenever the data cannot be expressed accurately after rounding, the exception is raised.

/\* opcode chart

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| operation | opcode | result |

|\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

| Add | 000 | out = in1 + in2 |

| Sub | 001 | out = in1 - in2 |

| Mul | 010 | out = in1 \* in2 |

| Div | 011 | not implemented |

|\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

extra bit(MSB of the opcode) is for rounding modes and int <-> float convertions

this is also to be done..

\*/

**Objective:** Create a **SystemC/HLS model of a RISC-V processor** using your existing FPU (floating-point unit), adder, subtractor, multiplier, and divider modules, then verify it via instruction-level simulation.

**Phase 1: Design the RISC-V Core**

**Step 1: Define the RISC-V ISA Subset**

Since you’re building a custom processor, decide which **RISC-V instructions** to support.

* **Base Integer (RV32I):**
  + add, sub, lw, sw, beq, jal (essential for control flow).
* **Floating-Point (RV32F/D):**
  + fadd, fsub, fmul, fdiv (using your FPU modules).
* **Custom Extensions (Optional):**
  + Add custom instructions if needed.

Step 2: Block Diagram of the Processor

Your processor will need:

Instruction Fetch (IF) – Reads instructions from memory.

Decode (ID) – Decodes instructions into control signals.

Execution (EXE) – Uses ALU/FPU for arithmetic/logic ops.

Memory Access (MEM) – Handles lw/sw.

Writeback (WB) – Updates registers.

Copy

        +--------+       +--------+       +--------+       +--------+       +--------+

        |        |       |        |       |        |       |        |       |        |

PC ---> |   IF   | ----> |   ID   | ----> |   EXE  | ----> |  MEM   | ----> |   WB   |

        |        |       |        |       | (ALU/  |       |        |       |        |

        +--------+       +--------+       |  FPU)  |       +--------+       +--------+

                                          +--------+

Copy

riscv64-unknown-elf-gcc -march=rv32if -mabi=ilp32 -o test.elf test.s

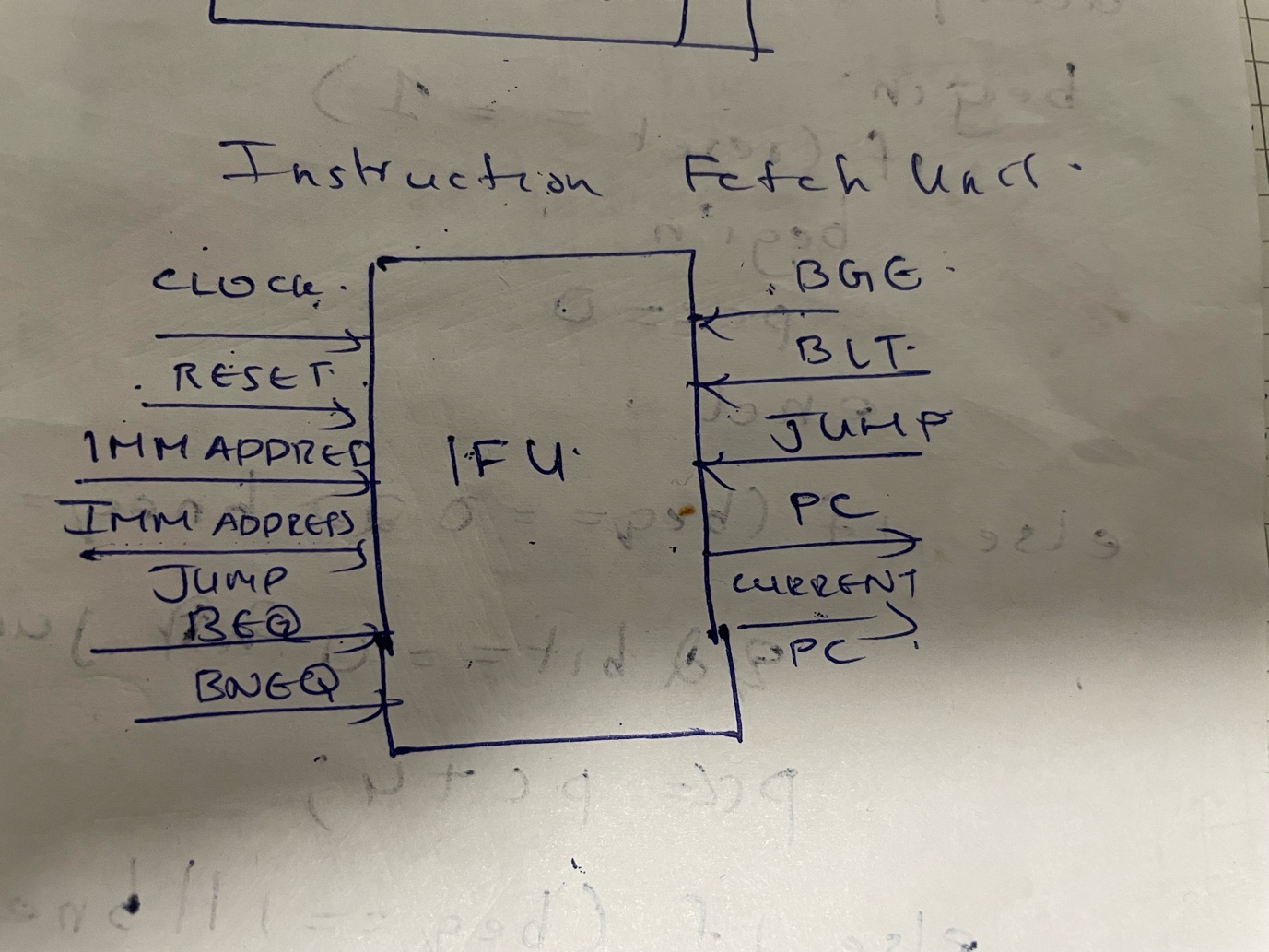
riscv64-unknown-elf-objcopy -O verilog test.elf test.hex

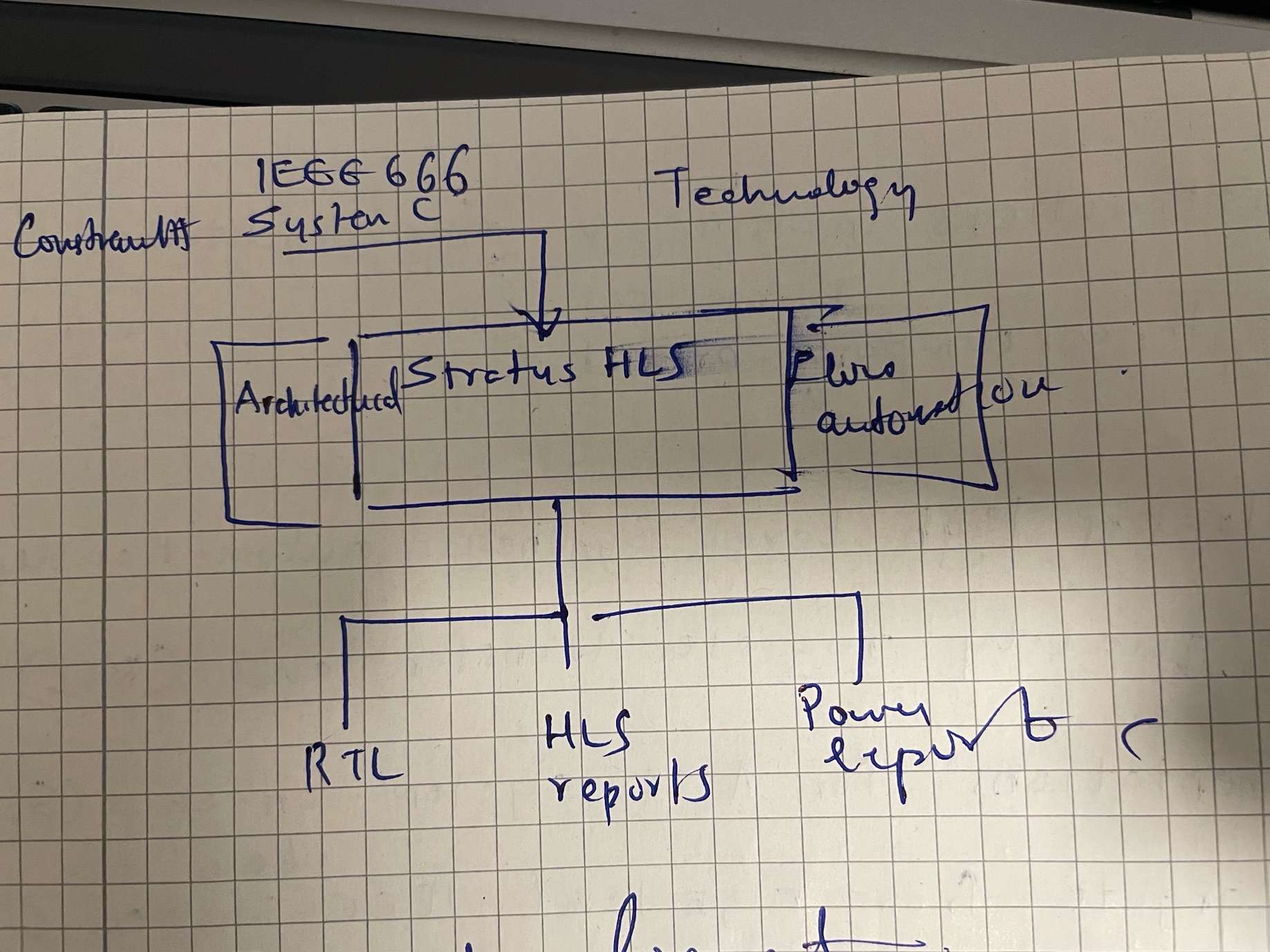
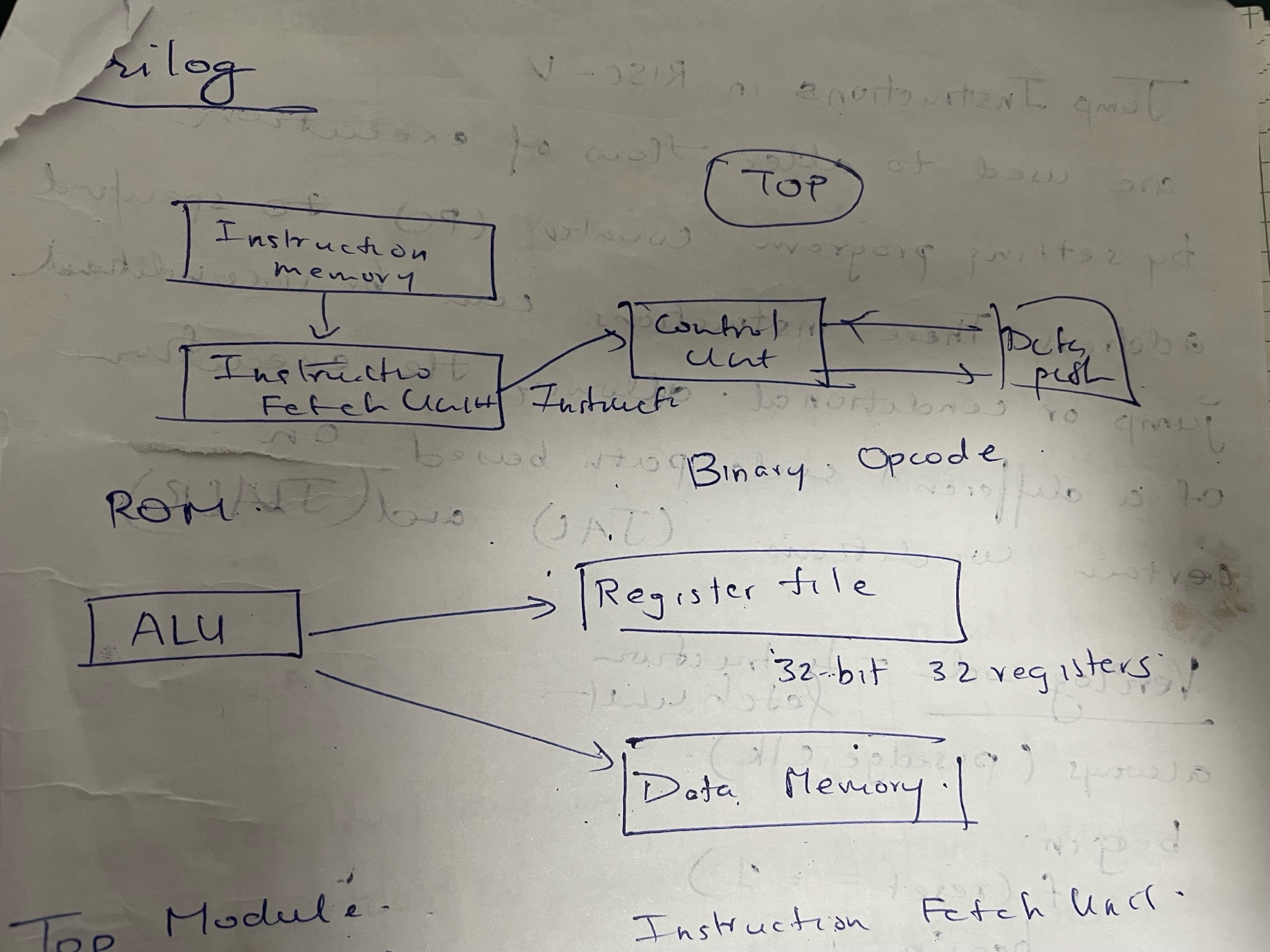
**Tools Summary**

| **Step** | **Tool** | **Task** |
| --- | --- | --- |
| 1 | **SystemC** | Design RISC-V pipeline stages |
| 2 | **Stratus HLS** | Convert to RTL (optional) |
| 3 | **Vivado/Xcelium** | Simulate RTL |
| 4 | **RISC-V Toolchain** | Compile test programs |
| 5 | **Waveform Debugger** | Verify execution |

**Key Takeaways**

1. **riscv64-unknown-elf-gcc**
   * Compiles RISC-V assembly → ELF (machine code + metadata).
   * Must specify **architecture (rv32if)** and **ABI (ilp32)** to match your processor.
2. **riscv64-unknown-elf-objcopy**
   * Converts ELF → **simulator-friendly hex format**.
   * Hex files are **loaded into RTL memory models** for testing.
3. **Why Not Use Raw Binary?**
   * ELF → Hex conversion preserves **address alignment** and **section ordering** (critical for correct execution).





Here’s a detailed two-paragraph explanation on writing and compiling RISC-V programs using the RISC-V GNU toolchain:

To write RISC-V assembly programs, you'll need to structure your code with proper directives and instructions. Start by creating a `.S` file with a `.text` section for code and use directives like `.align 2` for word alignment. The program typically begins with a `\_start:` label as the entry point. You can use base RISC-V instructions like `ebreak` for debugger breaks or `addi` for operations (which also serves as `nop` when written as `addi x0, x0, 0`). For register operations, pseudoinstructions like `mv t3, t4` (which assembles to `addi t3, t4, 0`) improve readability. The assembler supports PC-relative addressing with labels for jumps/branches (like `pcrel\_21` for ±1MB ranges). Programs should end with a termination instruction like `ebreak` to halt execution, as shown in your reference examples where consecutive `ebreak` instructions repeatedly trigger the debugger.

To compile the program, use the RISC-V GNU toolchain. First assemble with `riscv64-unknown-elf-as -march=rv32i -mabi=ilp32 program.S -o program.o`, then link with `riscv64-unknown-elf-ld program.o -o program.elf` to create an ELF executable. For debugging, convert to a hex dump using `riscv64-unknown-elf-objcopy -O verilog program.elf program.hex`. The reference shows how `rvddt` debugger can load the binary (`program.bin`) to inspect registers/memory - note how `objdump` listings (like in Listing 4.7) confirm that pseudoinstructions like `mv` compile to identical machine code (`000e8e13`) as their base instruction equivalents. The toolchain preserves alignment directives and handles PC-relative offsets automatically when using labels.

Key conversion notes:

1. Replaced SystemC modules with SystemVerilog modules
2. Used typedef union for float-uint32 conversion
3. Converted SC\_THREAD to always blocks with appropriate delays
4. Used SystemVerilog's real type for floating-point operations
5. Maintained the same pipeline structure and functionality
6. Split into design and testbench files as requested

### **Key Improvements**

1. **Forwarding Implemented**:
   * The Execute stage now checks for dependencies and **bypasses data** from EX/MEM or MEM/WB stages.
2. **No More Stalls**:
   * RAW hazards are resolved without inserting NOPs.
3. **Correct Output**:
   * No more incorrect intermediate values (like 30.0 in the earlier fmul.s output).

### **Expected Output (With Forwarding)**

### **Key Takeaways**

✅ **Forwarding removes stalls** while keeping all pipeline stages.  
✅ **MEM/WB still execute normally**—only the data path is optimized.  
✅ **No wrong intermediate values** (like 30.0 in your earlier output).

This is how modern CPUs (like RISC-V) handle data hazards efficiently! 🚀

### **What This Does**

1. **Detects RAW Hazards**:
   * If op1 or op2 depends on a result still in the pipeline (ex\_mem\_rd or mem\_wb\_rd), it **forwards the latest value** instead of stalling.
2. **Eliminates Stalls**:
   * No need to insert NOPs—dependent instructions get correct data immediately.
3. **Maintains Correctness**:
   * WB still happens normally, but forwarding allows earlier access to results.

No, **the MEM and WB stages are not eliminated in forwarding**—they still exist and are crucial for correct pipeline operation. Forwarding (also called **bypassing**) simply **shortcuts data dependencies**by allowing results to be used **before they are written back to the register file**.

### **How Forwarding Works Without Removing MEM/WB**

1. **MEM Stage Still Needed**:
   * For **load instructions (flw)**, data is only available **after MEM** (since memory reads happen here).
   * For **floating-point operations**, the MEM stage might just pass through results (since no memory access is needed).
2. **WB Stage Still Needed**:
   * Results must **eventually be written back** to the register file for correctness.
   * Forwarding **does not replace WB**—it just lets dependent instructions **use the result early**.

### **Example: Forwarding vs. No Forwarding**

#### **Without Forwarding (Stalling)**

Copy

Cycle 1: IF ADD f1, f2, f3

Cycle 2: ID ADD f1, f2, f3

Cycle 3: EX ADD f1, f2, f3

Cycle 4: MEM ADD f1, f2, f3 (result computed, not yet in regfile)

Cycle 5: WB ADD f1, f2, f3 (now f1 is updated)

Cycle 6: IF MUL f4, f1, f5 (STALL—must wait for WB)

Cycle 7: ID MUL f4, f1, f5

Cycle 8: EX MUL f4, f1, f5 (now f1 is available)

**→ 1 wasted cycle due to stall.**

#### **With Forwarding**

Copy

Cycle 1: IF ADD f1, f2, f3

Cycle 2: ID ADD f1, f2, f3

Cycle 3: EX ADD f1, f2, f3

Cycle 4: MEM ADD f1, f2, f3 (result forwarded to next instruction)

Cycle 5: WB ADD f1, f2, f3 (still writes back)

Cycle 6: IF MUL f4, f1, f5

Cycle 7: ID MUL f4, f1, f5

Cycle 8: EX MUL f4, f1, f5 (uses forwarded value from MEM stage)

**→ No stall, MEM/WB still happen.**

### **Key Takeaways**

1. **Forwarding does NOT remove MEM/WB**—it just **bypasses the register file** for RAW hazards.
2. **MEM is still needed** for:
   * Load instructions (data only available after MEM).
   * Passing through results (for non-memory ops).
3. **WB is still needed** to ensure **architectural correctness** (register file must eventually update).
4. **Only stalls are eliminated**, improving performance.

### **When Would MEM/WB Be Eliminated?**

* In **out-of-order execution** (like modern CPUs), instructions can execute **speculatively**, but WB still happens in-order.
* In **very deep pipelines**, some stages might split, but MEM/WB are still conceptually present.

### **Conclusion**

Forwarding **optimizes pipeline performance** but **does not remove stages**. MEM and WB remain essential for correctness. Your FPU pipeline should **use forwarding** to avoid stalls while keeping all stages intact.

In your **5-stage FPU pipeline**, the issue of **data hazards** (where an instruction depends on a result that hasn't been written back yet) can be resolved using either **bubbles (stalls)** or **forwarding (bypassing)**. Let's analyze which is better for your design.

### **Option 1: Pipeline Stalls (Bubbles)**

* **How it works**:
  + When a **RAW (Read-After-Write) hazard** is detected, the pipeline inserts **NOPs (bubbles)** to delay dependent instructions until the required data is available.
  + Example:

verilog

Copy

if (hazard\_detected) begin

// Stall by preventing IF/ID and ID/EX updates

// Insert NOP into EX stage

end

* **Pros**:
  + Simple to implement.
  + No extra hardware complexity.
* **Cons**:
  + **Performance penalty**: Every stall wastes **1 cycle per hazard**.
  + Inefficient for back-to-back dependent instructions.

### **Option 2: Forwarding (Bypassing)**

* **How it works**:
  + Instead of waiting for the result to be written back to the register file, **forward the result directly** from the EX/MEM or MEM/WB pipeline registers to the EX stage.
  + Example:

verilog

Copy

// Forwarding logic in Execute stage

if (ex\_mem\_reg\_write && (ex\_mem\_rd == id\_ex\_rs1)) begin

op1 = ex\_mem\_result; // Forward from EX/MEM

end

else if (mem\_wb\_reg\_write && (mem\_wb\_rd == id\_ex\_rs1)) begin

op1 = mem\_wb\_result; // Forward from MEM/WB

end

* **Pros**:
  + **No performance penalty** (avoids stalls).
  + More efficient for pipelined execution.
* **Cons**:
  + Requires **extra hardware** (multiplexers & hazard detection logic).
  + Slightly more complex to debug.

### **Which is Better for Your FPU Pipeline?**

| **Criteria** | **Stalls (Bubbles)** | **Forwarding** |
| --- | --- | --- |
| **Performance** | ❌ Slower (stalls) | ✅ No stalls |
| **Hardware Complexity** | ✅ Simple | ❌ More logic |
| **Power Efficiency** | ❌ Wastes cycles | ✅ Efficient |
| **Ease of Debugging** | ✅ Easier | ❌ Harder |

#### **Recommendation: Use Forwarding**

* Your FPU pipeline is **performance-sensitive** (floating-point ops are slow), so avoiding stalls is crucial.
* Forwarding is **standard in modern CPUs** (RISC-V, ARM, x86) for efficiency.
* The extra logic is justified by the **performance gain**.

### **How to Implement Forwarding in Your Design**

1. **Detect Hazards** in the ID stage:

verilog

Copy

// Check if rs1 or rs2 matches a pending write in EX/MEM or MEM/WB

assign hazard\_rs1 = (id\_ex\_rs1 != 0) &&

((id\_ex\_rs1 == ex\_mem\_rd && ex\_mem\_reg\_write) ||

(id\_ex\_rs1 == mem\_wb\_rd && mem\_wb\_reg\_write));

1. **Add Forwarding MUXes** in the EX stage:

verilog

Copy

always\_comb begin

// Forwarding for op1

if (ex\_mem\_reg\_write && (ex\_mem\_rd == id\_ex\_rs1))

op1 = ex\_mem\_result;

else if (mem\_wb\_reg\_write && (mem\_wb\_rd == id\_ex\_rs1))

op1 = mem\_wb\_result;

else

op1 = id\_ex\_op1;

// Repeat for op2

end

1. **Only Stall for Load-Use Hazards** (if needed):
   * If an instruction reads a register right after a load (flw), you may still need **1 stall cycle** since the data isn’t ready until MEM.

### **Final Answer**

**Use forwarding** for your FPU pipeline because:

1. It avoids unnecessary stalls, improving performance.
2. It’s the standard approach in real-world CPUs.
3. The extra logic is manageable and worth the trade-off.

If you’re constrained by hardware simplicity (e.g., academic exercise), **stalling** is acceptable but less efficient. For a **high-performance FPU**, forwarding is the clear winner.

Understanding RISC-V Instruction Formats and Floating-Point Operations

1. R-TYPE Instruction Format

The RISC-V architecture employs a structured approach to instruction encoding, with the R-TYPEformat being one of the most fundamental. This format is used for arithmetic and logical operations that operate on two source registers and store the result in a destination register.

Structure of R-TYPE Instructions

An R-TYPE instruction is encoded as a sextuplet consisting of:

func7 (7 bits) – Specifies the operation variant (e.g., distinguishes add from sub).

rs2 (5 bits) – The second source register.

rs1 (5 bits) – The first source register.

func3 (3 bits) – A minor opcode that further refines the operation.

rd (5 bits) – The destination register.

opcode (7 bits) – The major opcode (e.g., 0110011 for R-TYPE ALU operations).

For example, the instruction:

asm

Copy

sub a0, a1, a2

is encoded as:

func7 = 0100000 (indicating subtraction instead of addition)

rs2 = 01100 (register a2)

rs1 = 01011 (register a1)

func3 = 000 (standard for add/sub)

rd = 01010 (register a0)

opcode = 0110011 (R-TYPE operation)

This results in the 32-bit hexadecimal encoding 0x40C58533.

Decoding RISC-V Instructions

RISC-V instructions can be quickly identified by their least significant byte:

0x33 or 0xB3 → R-TYPE ALU instructions (e.g., add, sub, xor).

0x03 or 0x83 → LOAD instructions.

0x07 or 0x87 → Floating-Point LOAD (F extension).

This structured decoding allows for efficient hardware implementation and software disassembly.

2. Floating-Point Operations in RISC-V

RISC-V supports floating-point arithmetic through the F (Single-Precision) and D (Double-Precision) extensions. These operations follow a similar R-TYPE-like format but include additional control fields for rounding modes (rm).

Key Floating-Point Instructions

(1) Floating-Point Addition (fadd.s)

Format: fadd.s rd, rs1, rs2

Encoding:

func7 = 0000000

opcode = 1010011

Operation: f[rd] = f[rs1] + f[rs2]

(2) Floating-Point Subtraction (fsub.s)

Format: fsub.s rd, rs1, rs2

Encoding:

func7 = 0000100

opcode = 1010011

Operation: f[rd] = f[rs1] - f[rs2]

(3) Floating-Point Multiplication (fmul.s)

Format: fmul.s rd, rs1, rs2

Encoding:

func7 = 0001000

opcode = 1010011

Operation: f[rd] = f[rs1] × f[rs2]

(4) Floating-Point Division (fdiv.s)

Format: fdiv.s rd, rs1, rs2

Encoding:

func7 = 0001100

opcode = 1010011

Operation: f[rd] = f[rs1] / f[rs2]

Each of these instructions includes a rounding mode (rm) field (3 bits), allowing control over floating-point precision behavior (e.g., round-to-nearest, round-down, round-up).

Conclusion

The RISC-V instruction set architecture provides a clean and modular approach to both integer and floating-point operations. The R-TYPE format efficiently encodes register-to-register operations, while the F extension extends this logic to floating-point computations. Understanding these encoding schemes is crucial for compiler design, assembly programming, and hardware implementation in RISC-V-based systems.

By leveraging the structured opcode decoding and well-defined instruction formats, RISC-V achieves a balance between simplicity and performance, making it an increasingly popular choice in modern computing.

Programming an FPGA: From Hardware Description to Execution

Field Programmable Gate Arrays (FPGAs) represent a revolutionary approach to digital circuit design, offering programmable hardware that can be configured to implement specific functions. This essay explores the process of programming an FPGA, from its fundamental operating principles to modern high-level synthesis techniques, while also examining the role of RISC-V architecture in processor design.

FPGA Programming Fundamentals

An FPGA operates through two distinct modes: initialization and computation. During initialization, the FPGA's storage structures are populated with initial values. For Look-Up Tables (LUTs), this means loading the truth values of the Boolean functions they will represent. Equally important is configuring the connections between Configurable Logic Blocks (CLBs), achieved through multiplexers whose selection bits are set during this phase. This entire configuration process uses a bitstream - a sequence of bits transmitted serially from a programming station (typically a computer) to the FPGA. Once initialized, the FPGA transitions to computation mode, performing its designated function. Remarkably, this programming phase typically completes in just seconds.

Evolution of FPGA Design Methods

Early FPGA programming in the 1980s involved manually drawing logic gates, with translators handling their placement into LUTs. As FPGAs grew in complexity, Hardware Description Languages (HDLs) like VHDL and Verilog emerged, allowing designers to specify circuit behavior rather than draw individual gates. These languages capture not just the logical operations but also the critical temporal aspects of signal propagation.

The mid-1990s introduced High-Level Synthesis (HLS), a paradigm shift enabling circuit description through conventional programming languages like C/C++. HLS automatically handles temporal aspects, transforming code into Register Transfer Level (RTL) representations - an intermediate format that models synchronous digital circuits in terms of data flow between hardware registers. From RTL, placement and routing software maps the design onto CLBs, optimizing for minimal propagation delays before generating the final bitstream for FPGA configuration.

Processor Fundamentals and RISC-V Architecture

Understanding processor operation provides context for FPGA implementations. Processors follow a fetch-decode-execute-store cycle, guided by their Instruction Set Architecture (ISA) - the collection of instructions they can execute. Assemblers convert human-readable assembly into machine code that processors can execute directly.

The RISC-V architecture represents a significant advancement as a free, open standard suitable for devices ranging from small embedded systems to large computing systems. Its key features include:

Open standard with no licensing fees

Modular design with a base instruction set and optional extensions

32 general-purpose registers for fast data access

Scalability across different implementations

Notably, RISC-V's base integer instruction set (RV32I) supports only integer operations, requiring extensions for floating-point arithmetic. The architecture's extensibility makes it ideal for research and innovation in processor design, though this flexibility also increases verification complexity.

Practical FPGA Implementation

Modern FPGA design tools like Vitis HLS emphasize creating components with well-defined pinouts, prohibiting internal state observation or modification from outside. Components may be combinational (outputs depend solely on current inputs) or sequential (outputs depend on inputs and internal state that evolves with clock cycles). Pragmas (compiler directives) provide crucial synthesis guidance, such as specifying pinout mapping on the FPGA hardware.

The design process involves creating both the IP core (like an adder) and its testbench for verification through simulation. This rigorous approach ensures the hardware implementation faithfully represents the intended functionality before deployment on the physical FPGA.

Conclusion

FPGA programming has evolved from manual gate-level design to sophisticated high-level synthesis, paralleling advancements in processor architecture like RISC-V. This convergence of programmable hardware and open processor standards creates exciting opportunities for customizable computing solutions across diverse applications. The continued development of tools like HLS and modular architectures like RISC-V promises to further democratize hardware design while addressing the growing challenges of system verification in an increasingly complex computing landscape.

The Fundamentals of CPU Operation and RISC-V Architecture

Introduction to CPU Execution

At the heart of every computer lies the Central Processing Unit (CPU), which executes a continuous stream of instructions known as a program. These instructions are expressed in machine language, where each command is represented as a binary value. To simplify programming, a symbolic mapping called Assembly Language is used, replacing binary values with human-readable mnemonics and parameters. This abstraction allows programmers to write code without directly manipulating binary data.

CPU Structure: The Execution Unit and Harts

The Execution Unit is the core component of a CPU that coordinates instruction operations. It manages data transfers between the CPU and main memory, controls register storage, and ensures proper instruction execution.

In RISC-V architecture, a hart (hardware thread) is analogous to a core in traditional CPUs. Each hart consists of 32 registers, an instruction execution unit, and an Arithmetic Logic Unit (ALU). When multiple harts are present, they can execute different instruction streams simultaneously, enabling multithreaded programs. However, this text primarily focuses on single-hart CPUs for simplicity.

Instruction Set Architecture (ISA)

The Instruction Set Architecture (ISA) defines the rules governing a CPU's instructions and features. It specifies how binary instructions are interpreted and executed. The RISC-V ISA is modular, allowing CPU designers to select which features to implement.

Base Modules in RISC-V

The RISC-V ISA includes several base modules:

RV32I (32-bit general-purpose)

RV32E (32-bit embedded)

RV64I (64-bit general-purpose)

RV128I (128-bit general-purpose)

These modules provide essential integer operations, with varying bit-widths to accommodate different memory requirements. This text primarily explores RV32I, the 32-bit general-purpose module.

The CPU Execution Cycle

Program execution involves repeating a cycle of three phases:

Fetch – The CPU retrieves the next instruction from memory using the program counter (pc)register.

Decode – The CPU interprets the instruction, determining which registers and operations are involved.

Execute – The CPU performs the operation (e.g., arithmetic, memory access) and updates the pcaccordingly.

Most instructions proceed sequentially, incrementing the pc to the next memory address. However, jump (unconditional) and branch (conditional) instructions can redirect execution to a different address. For example:

"If register x8 < x24, continue to the next instruction; otherwise, branch to a specified address."

Instruction Alignment in RISC-V

RISC-V mandates 32-bit instruction alignment—all instructions must reside on full-word boundaries. An attempt to fetch an unaligned instruction triggers an alignment exception, often causing a program crash.

Writing Assembly Programs

Assembly programs are written in plain text, typically saved with a .S or .sx extension. Each line may include:

Labels (e.g., start:) – Mark positions in the code.

Mnemonics (e.g., addi) – Specify CPU operations.

Registers (e.g., x28, x0) – Hold data for operations.

Comments (after #) – Explain the code’s purpose.

For example, the following program sets four registers to zero using addi (add immediate):

assembly

Copy

.global start

.text

start:

addi x28, x0, 0 # Set x28 = 0 + 0

addi x29, x0, 0 # Set x29 = 0 + 0

addi x30, x0, 0 # Set x30 = 0 + 0

addi x31, x0, 0 # Set x31 = 0 + 0

Key Takeaways

The CPU executes programs through fetch-decode-execute cycles.

RISC-V’s modular ISA allows customizable CPU designs.

Assembly language simplifies machine code programming using mnemonics.

Proper instruction alignment is critical to avoid runtime errors.

Understanding these fundamentals is essential for low-level programming and computer architecture design.