🧮Midterm Alternative Project: RISC-V Numeric Ops Simulator

📘Overview

Build a small, well-tested numeric operations simulator that:

1. Converts integers to/from **two's complement** (multiple bit widths).
2. Simulates **RISC-V M extension** integer multiply/divide instructions.
3. Implements **IEEE 754 single-precision (float32)** arithmetic: add, subtract, multiply.

Extra credit: Support double precision (float64) and additional rounding modes.

Implementation must be in a high-level language (Python, Java, C/C++, Rust, Go, etc.), include clear step-by-step traces for multi-step algorithms, and come with a comprehensive unit test suite.

🕑Learning Goals

- Master two's complement encoding/decoding across widths.
- Understand and simulate RISC-V M and F extensions.
- Manipulate IEEE-754 bitfields, rounding, and normalization.
- Design traceable arithmetic algorithms with correctness and overflow/underflow handling.

🧰Project Summary

Goal: Build a numeric simulator using **bit-level logic only** (no built-in arithmetic operators).

It must:

1. Convert signed integers to/from two's complement.
2. Implement RV32M multiply/divide instructions.
3. Perform IEEE-754 float32 operations.

📁Overall Requirements

Category	Requirement
Language	Any high-level language (Python, C/C++, Java, Rust, Go, etc.)
Representation	Use **bit vectors** (arrays/lists of 0/1 or booleans).

Category	Requirement
Arithmetic	Implement manually using bit logic (no built-ins).
Prohibited	`+ - * / % << >> **` or base conversion helpers (`int(x, base)`, `bin()`, `hex()`, etc.) in your implementation.
Allowed	Boolean logic (`and`, `or`, `xor`, `not`), control flow, string ops.
Testing	Automated tests verifying results and flags.
Tracing	Show intermediate states for multi-step algorithms.
Design	Keep functions pure, deterministic, and easy to test.

🧮 MODULE 1 — Two's Complement Toolkit

Purpose

Convert integers to/from 32-bit two's complement with overflow detection.

Implement

```
encode_twos_complement(value: int) -> {bin, hex, overflow_flag}
decode_twos_complement(bits: str | int) -> {value: int}
```

Behavior

- Fixed width = **32 bits**.
- `encode`: produce binary and hex strings.
- `decode`: return signed integer value.
- `overflow_flag`: 1 if value outside [-2^{31}, $2^{31}-1$].
- Add helpers:
- `sign_extend(bits, new_width)`
- `zero_extend(bits, new_width)`

Example

Input	Binary	Hex	Overflow
+13	00000000_00000000_00000000_00001101	0x0000000D	0
-13	11111111_11111111_11111111_11110011	0xFFFFFFF3	0
2^{31}	—	—	1

MODULE 2 — Integer ADD/SUB (RV32I)

Purpose

Perform 32-bit signed addition/subtraction with correct ALU flags.

Implement

```
alu(bitsA, bitsB, op) -> {result_bits, N, Z, C, V}
```

Flags

Flag	Meaning
N	Negative result
Z	Zero result
C	Carry out (1 = carry out of MSB; 1 = no borrow for SUB)
V	Signed overflow

Rules

- `ADD`: V=1 if operands have same sign, but result sign differs.
- `SUB`: use `rs1 + (~rs2 + 1)`; V=1 if operand signs differ and result sign ≠ rs1.
- Use your **own adder logic** (no built-in operators).

Example Cases

Operation	Result	V	C	N	Z
0x7FFFFFFF + 1	0x80000000	1	0	1	0
0x80000000 - 1	0x7FFFFFFF	1	1	0	0
-1 + -1	-2	0	1	1	0

MODULE 3 — Multiply/Divide Unit (RV32M)

Purpose

Simulate RISC-V integer multiply/divide operations.

Required Instructions

Group	Required	Optional
Multiply	`MUL` (low 32 bits)	`MULH`, `MULHU`, `MULHSU`
Divide	`DIV`	`DIVU`, `REM`, `REMU`

Division Edge Cases

Case	Result
`DIV x / 0`	q = -1 (0xFFFFFFFF), r = dividend
`DIVU x / 0`	q = 0xFFFFFFFF, r = dividend
`DIV INT_MIN / -1`	q = INT_MIN (0x80000000), r = 0
Remainder sign	Same as dividend

Overflow Flag

- `MUL`: flag if 64-bit product doesn't fit in 32 bits.
- `DIV`: flag overflow for INT_MIN ÷ -1.

Algorithm

- **MUL:** implement *shift-add* algorithm.
- Trace each iteration (accumulator, multiplier, carry, count).
- **DIV:** use *restoring* or *non-restoring* algorithm.
- Trace remainder, quotient, and decisions per step.

Example Tests

Operation	Expected
MUL 12345678 × -87654321	rd=0xD91D0712; overflow=1
MULH 12345678 × -87654321	rd=0xFFFC27C9
DIV -7 / 3	q=-2 (0xFFFFFFFE); r=-1
DIVU 0x80000000 / 3	q=0x2AAAAAAA; r=0x00000002

🕐MODULE 4 — Float32 Unit (FPU)

Purpose

Implement IEEE-754 single-precision encode/decode and arithmetic.

Implement

```
pack_f32(value: decimal) -> bits
unpack_f32(bits) -> float_value
fadd_f32(a, b), fsub_f32(a, b), fmul_f32(a, b)
```

Requirements

- Work at bit level (split into sign, exponent, fraction).
- Perform: align → add/sub → normalize → round → repack.
- Default rounding: **RoundTiesToEven**.
- Handle special values: ±0, ±∞, NaN.
- Optional: support subnormals.

Flags

Flag	When Set
overflow	exponent overflow → ±∞
underflow	result < 2^{-126}
invalid	NaN ops or ∞−∞, 0×∞
divide_by_zero	if you add division

Example

Operation	Result	Notes
1.5 + 2.25	0x40700000	= 3.75
0.1 + 0.2	0x3E99999A	Rounds to even
1e38 × 10	+∞	overflow=1
1e−38 × 1e−2	subnormal	underflow=1

⚙ Hardware-Style Components

Component	Description
Reg(width)	Simple register with load/clear.
RegisterFile	32 × 32-bit regs (x0 hardwired to 0). Optional FP regs.
FCSR	Holds rounding mode + exception flags.

Component	Description
ALU	Built from full adders; outputs N, Z, C, V.
Shifter	Implements SLL, SRL, SRA manually.
MDU	Handles MUL/DIV algorithms and flags.
FPU	Float add/sub/mul logic with bit operations only.

🧾 Testing Requirements

Each test must assert:

- **Numeric result** (decimal)
- **Bit pattern** (hex)
- **Flags** (N, Z, C, V or FP flags)
- Include **traces** for at least one multiply and one divide.

🧱 Deliverables Checklist

Deliverable	Description
GitHub Org/Repo	Invite instructor + team members.
README.md	Build/run instructions.
AI_USAGE.md	Explain any AI assistance.
ai_report.json	Line count summary for AI-assisted code.
Code	All modules + tests.
Traces	Output examples for algorithms.

🧰 Constraints Summary

Category	Forbidden	Allowed
Arithmetic	`+ - * / % **`	Bitwise full/half adders
Shifts	`<< >>`	Custom shifter logic
Base Conversion	`int(x, base)`, `bin()`, `hex()`	Manual lookup tables
Floats	Any float math	Your own IEEE-754 logic

Category	Forbidden	Allowed
Data	Host ints	Bit arrays only
Tests	Can use math for reference	✅Yes

🗓️Suggested Schedule

Time	Milestone
Week 0	Choose language, repo setup
Week 1	Finish Two's Complement + tests
Week 2	Complete RV32M MUL/DIV + traces
Week 3	Finish Float32 arithmetic + tests

Merge Guidance (for later CPU project)

- Keep a `NumericCore` API exposing pure functions.
- Use a simple `State{regs[32], fregs[32], flags}` structure.
- Avoid global state; make all functions deterministic.

🎓Academic Integrity

- Discuss high-level ideas with peers only.
- All code/tests must be your own.
- Cite any references in `README.md`.

End of Instructions. You got this!