SoCET Floating Point Unit in RISC-V

Justin Sanchez, Jain Iftesam, Duc Pham Minh, Saandiya KPS Mohan

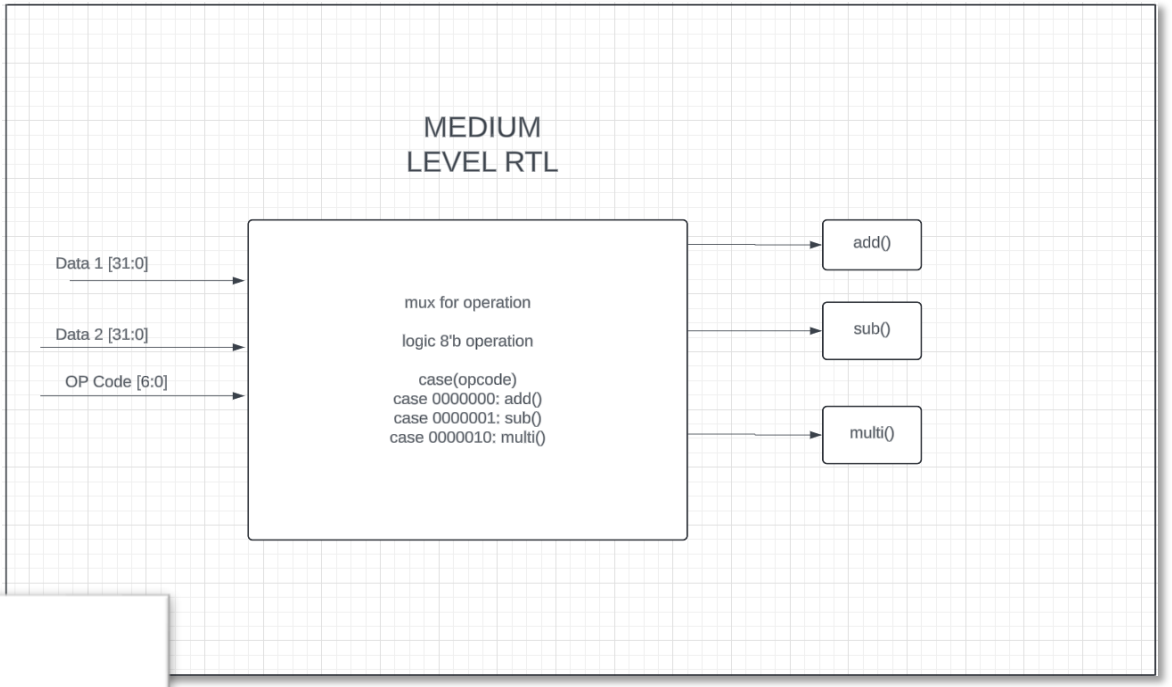
PURDUE UNIVERSITY | School of Electrical and Computer Engineering

What is FPU

Arithmetic Operations on Floating Point Numbers

Compatible with IEEE 754 Standard

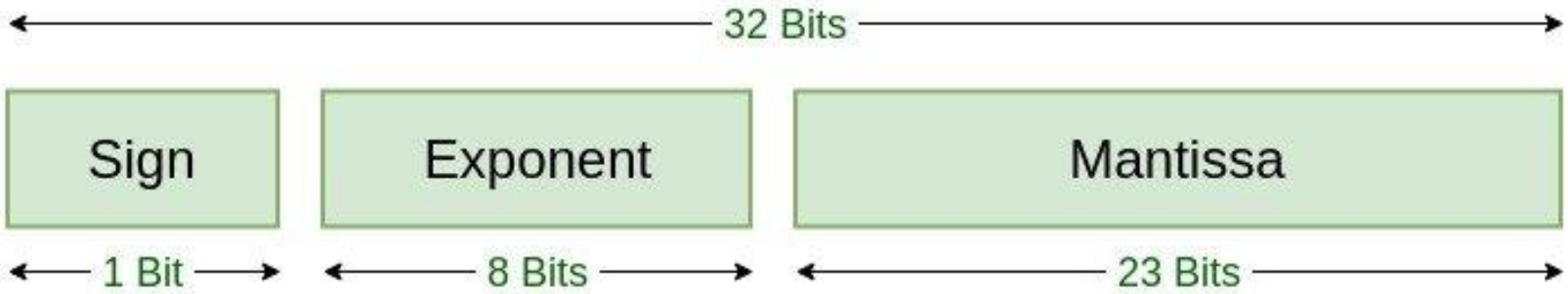
Supports:
- Half Precision
- **Single Precision**
- Double Precision



Advantages of FPU

- Critical for Precision-based Control Systems (Adherence to IEEE754 Standard)
- Reduces Latency
- Improves Overall Throughput
- Broad Application Support
  - Graphics
  - AI
  - Avionics
  - Scientific Computing

IEEE-754



Single Precision
IEEE 754 Floating-Point Standard

1.010101001×2^{6}

Floating point number in decimal format

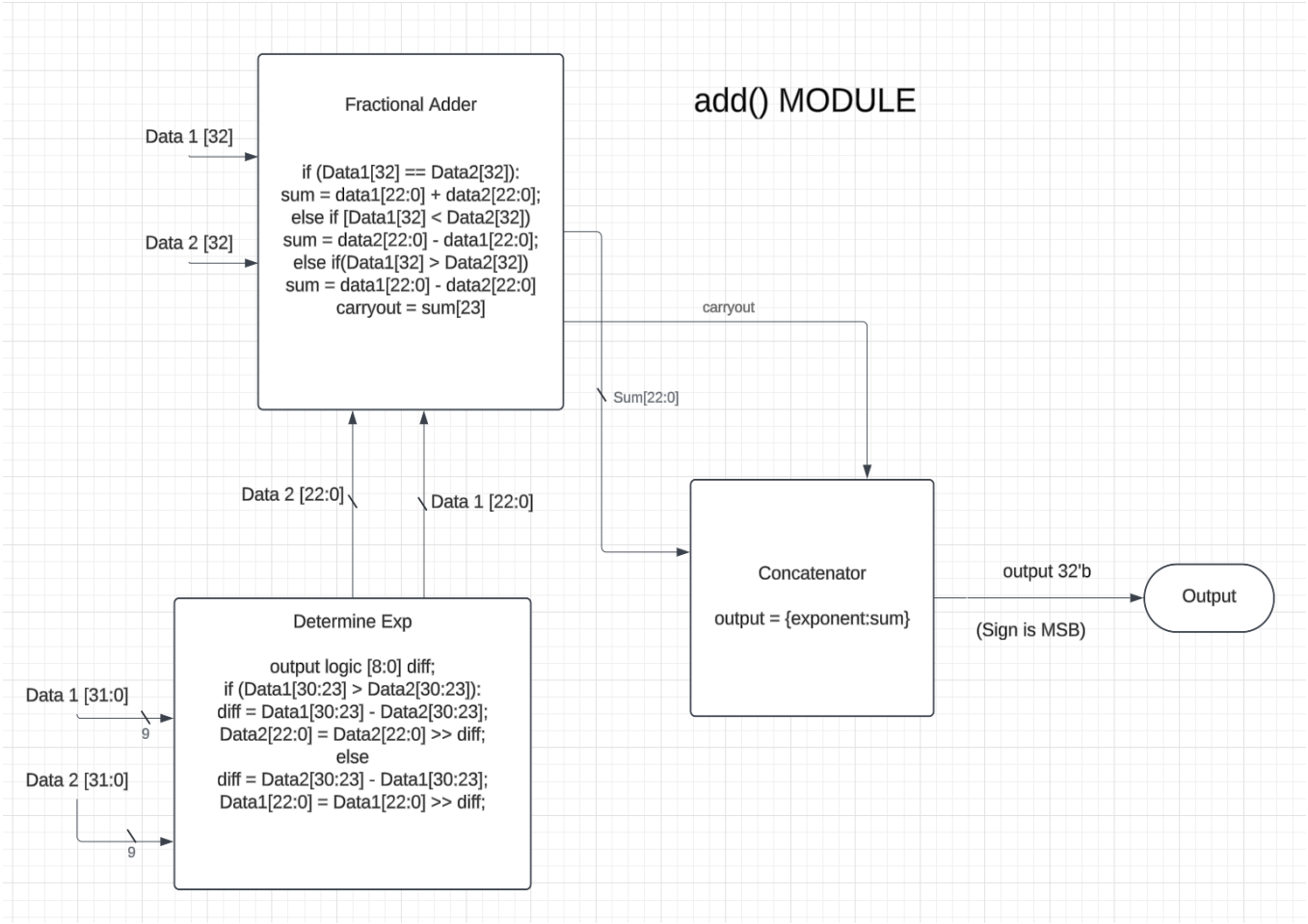
https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/

Adder & Subtractor Module

Normalize exponents

Align mantissas and perform addition

Normalize result

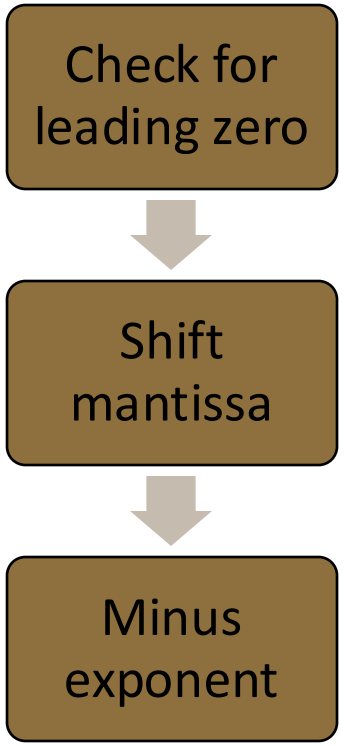


MSB of the second data input can be flipped to calculate subtraction.

Carryout

Checking for first carry

```
if (carry & ~mantissaResult[25]) begin
    // check for carry out, shift right and increment exponent
    normalized_mant = mantissaResult[24:1];
    normalized_exp = biggerExp + 1;
end
```



Important catches:

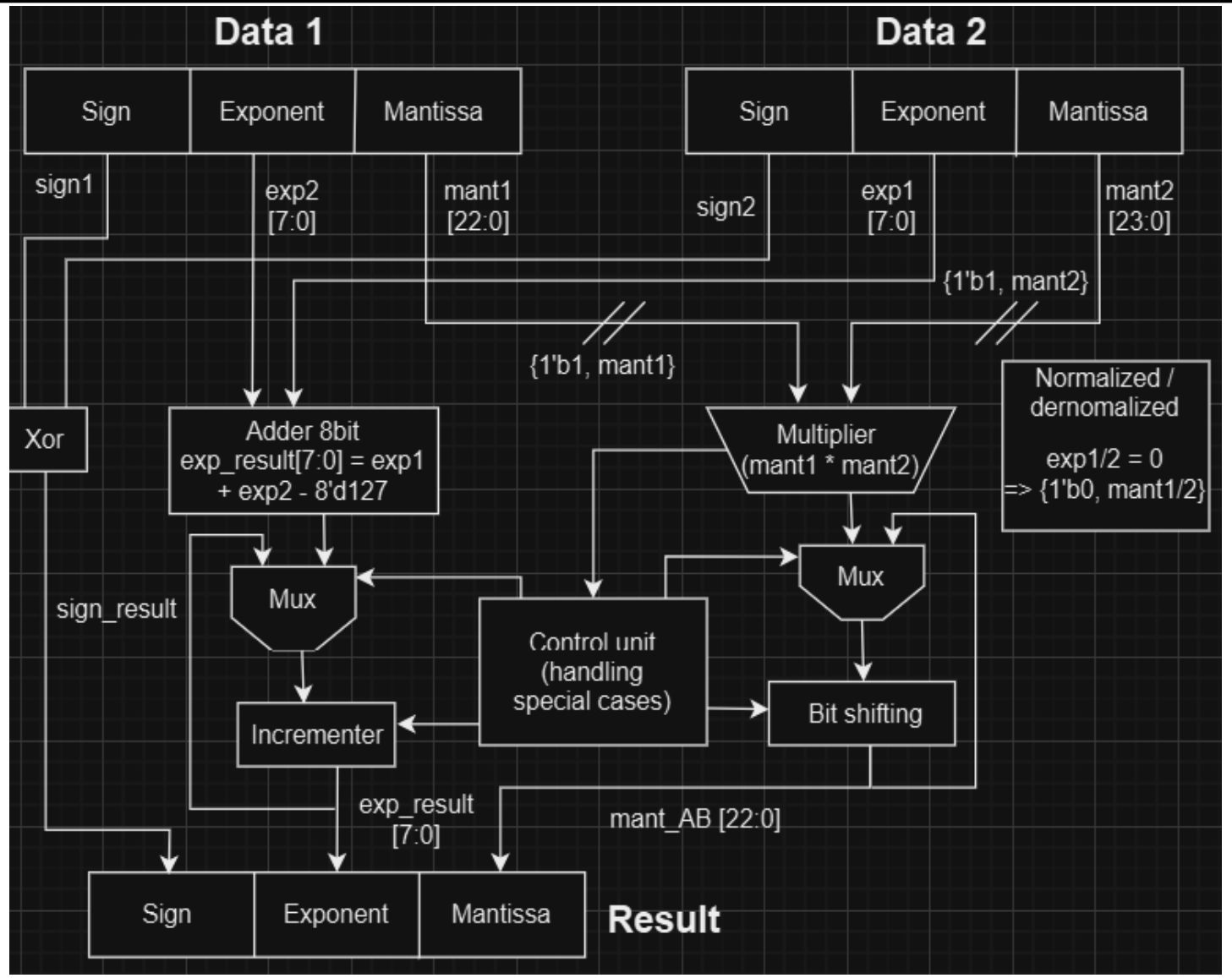
- 2 bit carry out value if MSB of mantissa is 1
- Append implicit 1's in mantissa before calculations
  - E.g: 1.345 x 2^16

```
casez (mantissaResult[24:0])
    25'b01?????????????????????????: normalized_mant = mantissaResult[23:0];
    25'b001????????????????????????: begin
        normalized_mant = {mantissaResult[22:0],1'b0};
        normalized_exp -= 1; end
    25'b0001???????????????????????: begin
        normalized_mant = {mantissaResult[21:0],2'd0};
        normalized_exp -= 2; end
    25'b00001??????????????????????: begin
        normalized_mant = {mantissaResult[20:0], 3'd0};
        normalized_exp -= 3; end
```

Using case statements to check leading zero

Multiplication – RTL

- Extract Components
- Calculate the resultant sign
- Add Exponent (subtract the bias)
- Multiply mantissas
- Normalize result
- Handle special cases



Multiplication – Special Cases

1. Zero Multiplication: either operand is zero
=> result is zero.

```
// case zero
if (zero1 || zero2) begin
    // zero times any number is zero
    result = {sign_result, 31'b0};
end
```

2. Infinity Multiplication: either operand is infinity
=> result is infinity.

```
// case infinity
else if (infi1 || infi2) begin
    // if one operand is infinite value, result is infinity
    result = {sign_result, 8'hFF, 23'b0};
end
```

3. Zero with Infinity: multiplying zero with infinity
=> result is NaN (not a number).

```
// Undefined case
else if (infi1 && zero2) begin
    //  infinity times with zero is undefined value
    result = 32'h7FC00000; // Quiet NaN
end
```

Multiplication – Special Cases

4. NaN Inputs: any operand is NaN => result is NaN.

```
// Undefined case
else if (na1 || na2) begin
    // if one operand is NaN, result is NaN
    result = 32'h7FC00000; // Quiet NaN
end
```

5. Overflow: The resultant exponent exceeds the max => result is set to infinity, overflow flag is raised.

```
// handling and detecting overflow
if (exp_result >= 8'd255) begin // when MSB is 1, expo
    overflow = 1;
    // exp_result = 8'hff;
    result = {sign_result, 8'hFF, 23'b0}; // infinity
end
```

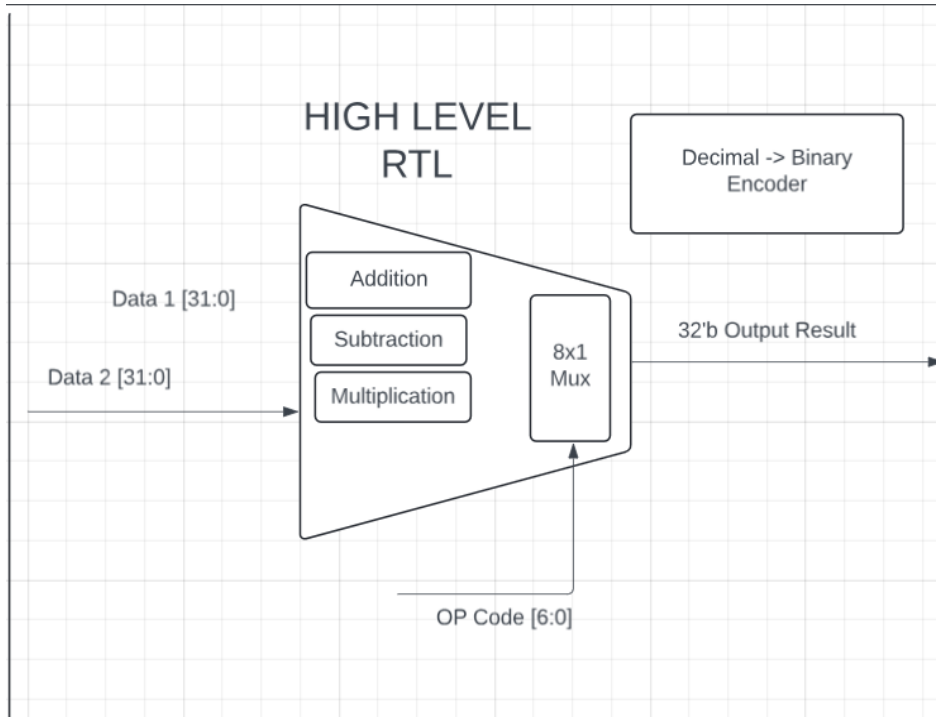
6. Underflow: resultant exponent is too small => result may be denormalized to 0, underflow flag is raised

```
else if (exp_result == 0) begin
    underflow = 1;
    if (exp_result < -23) begin // the result is too small for denormalized
        mant_AB = 24'b0; // set underflow to zero
        mant_result = mant_AB[22:0];
    end
    else begin // the result can be represented as denormalized number
        mant_AB = mant_temp >> (1 - exp_result);
        mant_result = mant_AB[22:0];
    end
    result = {sign_result, 8'b0, mant_AB[22:0]};
end
```

7. Normalization: ensure mantissa must be normalized in range 1.0 to 2.0

Top level Module

- Included a decoder to calculate opcode values.
- Instantiated decode, adder and multiplication module in top FPU.

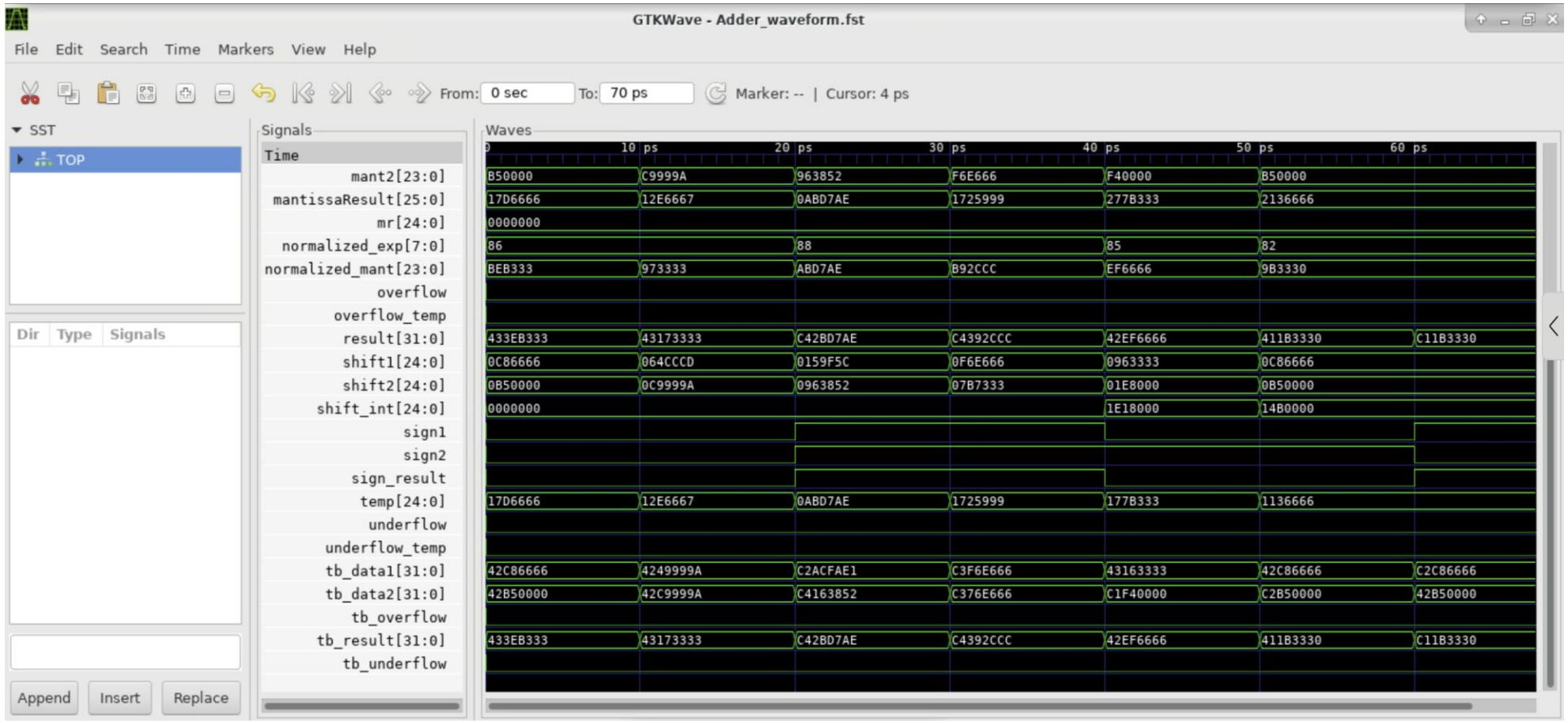


Top module

Testing

```
socet211@asicfab:~/FPU (fpu_fa24 $)$ fusesoc --cores-root . run --target=sim --setup --build adder
WARNING: Parse error. Ignoring file ./%.core: while scanning for the next token
found character that cannot start any token
  in "<unicode string>", line 33, column 19
WARNING: Parse error. Ignoring file ./subtraction.core: mapping values are not allowed in this context
  in "<unicode string>", line 30, column 16
INFO: Preparing socet:aft:adder:1.0.0
INFO: Setting up project
INFO: Building simulation model
socet211@asicfab:~/FPU (fpu_fa24 $)$
```

For testing, our team used fusesoc to compile our test benches and GTKwave to simulate our designs.



Future Plans

- Multiplication, division modules, square root operation.
- For accurate handling, test these modules against edge cases such as extreme values (e.g., max/min exponent and mantissa), subnormal numbers, and rounding scenarios.
- Focus on optimizing the latency and area of each module to achieve a balance between performance and resource efficiency
- Essential for embedding the FPU in open-source RISC-V architectures without compromising scalability.
- Support double-precision floating-point calculations.