Design Report - Lab 3 - Creating FPU HW

Mihir Divyansh E

Keywords: FPGA, FPU

Abstract. This document details the design of a Floating Point Unit in Verilog, and the methodolgy to verify on a PYNQ FPGA board.

1 Design Specifications

The goal of this project was to design, synthesize, and implement a hardware Floating Point Unit (FPU). The FPU was to be deployed and tested on an FPGA board with peripheral interfacing for data transfer. Below is the specification that the hardware adheres to.

- Arithmetic operations for single-precision (FP32) and double-precision (FP64), load and store:
  - Addition (`fadd`)
  - Subtraction (`fsub`)
  - Multiplication (`fmul`)
  - Comparision (`fcmp`)
  - Load (`ld`)
  - Store (`sd`)
- A 32×64-bit register file for operand storage.
- Exception flag outputs (`fex`)
- Currently truncates results
- 24 bit instruction format
- The Arithmetic operations are themselves pipelined, but the **FPU as a coprocessor is not hazard protected.**

2 Floating Point Adder

Floating point addition in IEEE-754 format is more involved than integer operations. The design is pipelined to 4 stages.

1. Input parse / Special Case
2. Subtract exponent, Shift mantissa, Pad Mantissa
3. Add the mantissas (sign aware)
4. Send to output

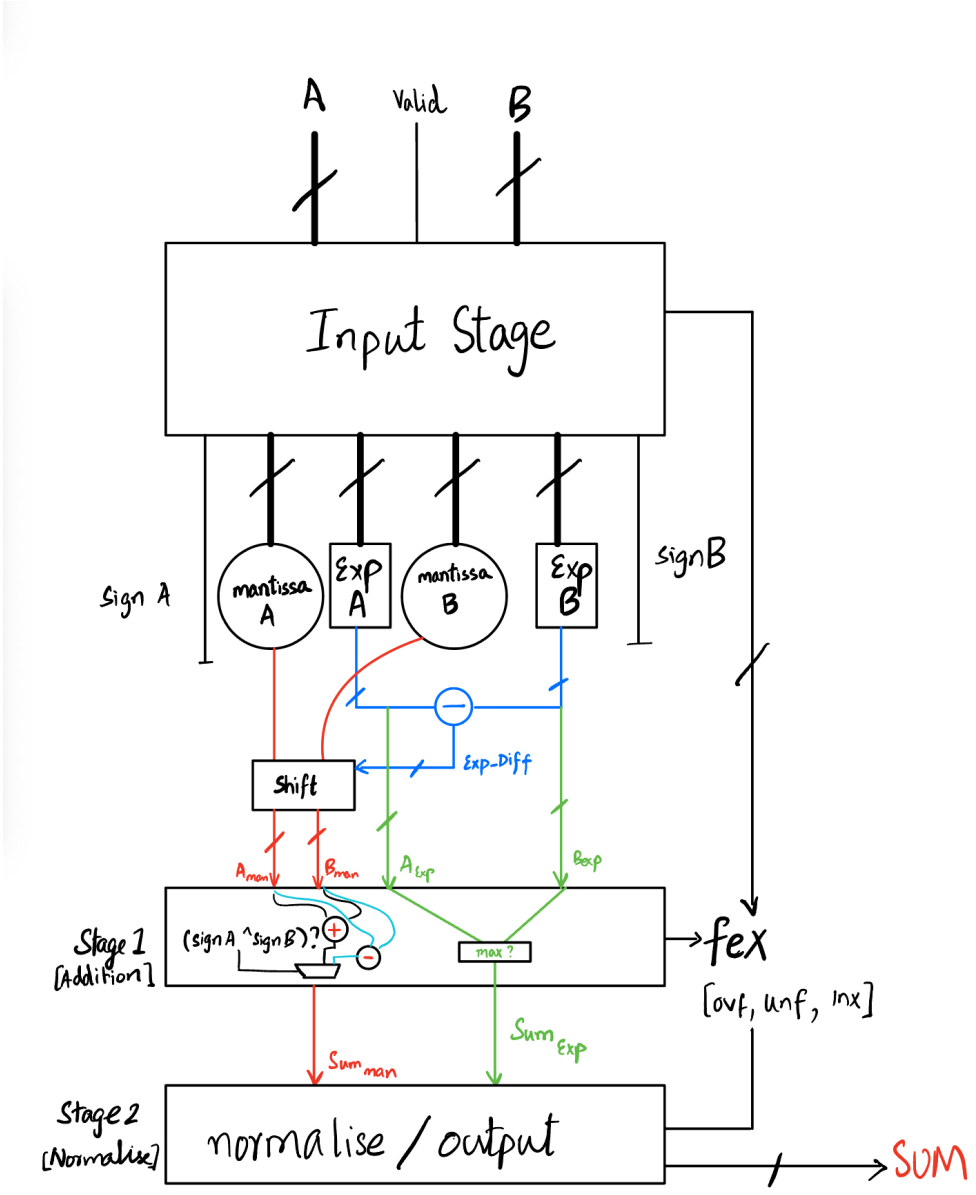

Figure 1: Floating Point Adder

2.1 Step 1: Operand Decomposition and Special Case Handling

Each operand is decomposed into three fields:

- Sign bit (S)
- Exponent (E)
- Mantissa or significand (M)

Then, we check if the operand(s) is/are

- Zero
- NaN
- Subnormal
- Inf

In such case that the operands are special, the result is defined in this stage, and are propagated to the output, though the same pipeline, so as to ensure a common output cycle.

2.2 Step 2: Exponent Alignment

- Compare the exponents of the two operands.
- Shift the mantissa of the operand with the smaller exponent to the right until both exponents are equal.
- Keep track of the larger exponent

2.3 Step 3: Mantissa Operation

- If the operands have the same sign, add the aligned mantissas.
- If the operands have different signs, perform subtraction of the mantissas.
- The resulting sign is determined by the operand with the larger magnitude.

2.4 Step 4: Normalization

- After addition or subtraction, the result may not be normalized.
- If there is a carry-out (e.g., $1.xx$ format), shift the mantissa right and increment the exponent.
- If the result is smaller than normalized (leading zeros), shift the mantissa left and decrement the exponent until the leading bit is ‘1‘. **In the implemented hardware, the leading zero detector is combinational. Design can be made faster by remaking this block**

2.5 Step 5: Rounding

Generally, there are multiple rounding methods, RNE being preferred in mainstream applications. The hardware supports RTZ (truncated).

- Round to Nearest, ties to Even (RNE)
- Round towards Zero (RTZ)
- Round Up (RUP)
- Round Down (RDN)
- Round to Nearest, ties Away (RNA)

2.6 Step 7: Result Assembly

Finally, the result is reassembled:

$$\text{Result} = \{S, E, M\} \tag{1}$$

3 Floating Point Multiplier

Consider two floating point numbers A and B represented in normalized form:

$$A = (-1)^{s_A} \times (1.M_A) \times 2^{E_A - Bias}$$
$$B = (-1)^{s_B} \times (1.M_B) \times 2^{E_B - Bias}$$

(For denormal numbers, there is detection to replace the hidden 1 bit, with 0) where s denotes the sign bit, M the fractional part of the mantissa, and E the biased exponent. The multiplication is performed in the following steps:

1. **Sign Calculation:** The sign of the product is obtained as the XOR of the input signs:

$$s_P = s_A \oplus s_B$$

2. **Exponent Calculation:** The exponents are added and the bias is subtracted to obtain the result exponent:

$$E_P = (E_A - Bias) + (E_B - Bias) + Bias = E_A + E_B - Bias$$

3. **Mantissa Multiplication:** The significands $(1.M_A)$ and $(1.M_B)$ are multiplied:

$$M_P = (1.M_A) \times (1.M_B)$$

   This results in a product in the range $[1.0, 4.0)$.

4. **Normalization:** If $M_P \in [2.0, 4.0)$, the product must be shifted right by one bit, and the exponent incremented by 1:

$$M_P \to M_P/2, \quad E_P \to E_P + 1$$

   The final mantissa stored is the fractional part of M_P after normalization.

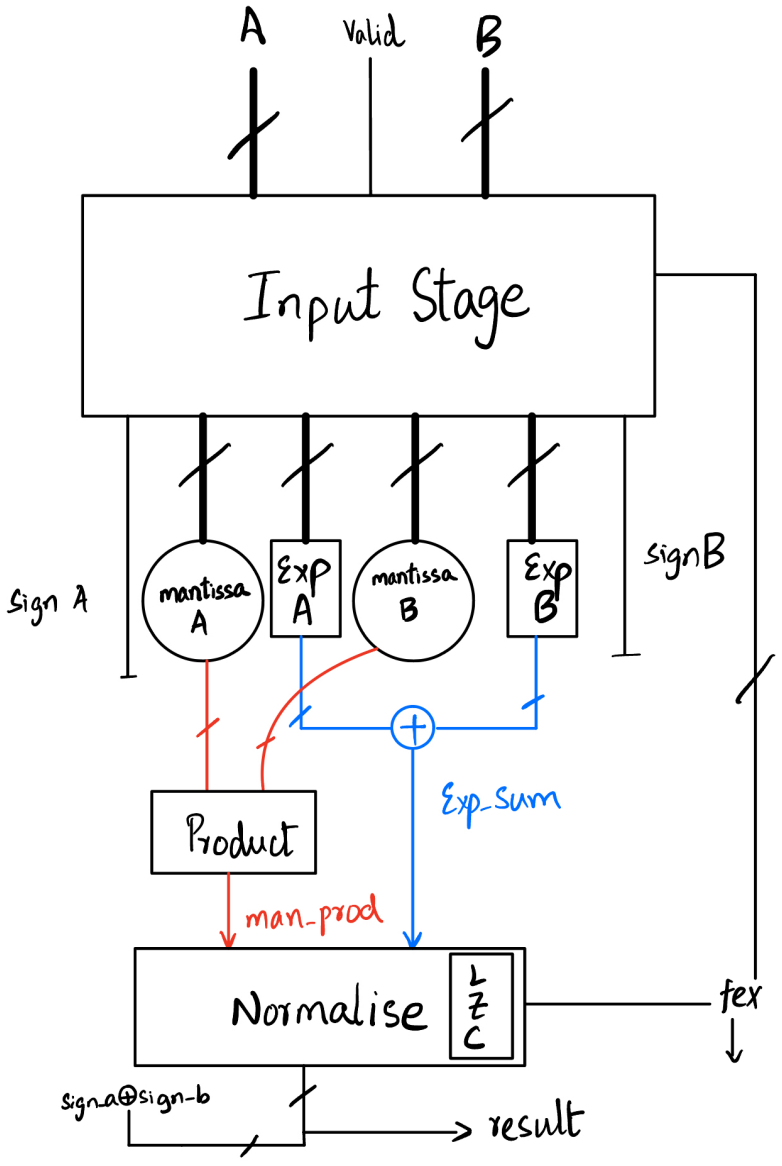

Figure 2: Floating Point Multiplier

5. **Subnormals:** If one of the numbers is subnormal, the lack of a leading 1 removes the guarantee of result being in $[1.0, 4.0)$. In this case, we need a leading zeros counter, to get the required shift amt, so as to set the exponent. In my case, the **LZD is purely combinational**. In case both are subnormals, ZERO is returned.

6. **Rounding:** The result is rounded according to the selected rounding mode (e.g., round-to-nearest-even, round-towards-zero). This ensures compliance with IEEE-754 precision.

7. **Handling Exceptions:** Special cases must be considered:

   - If either operand is `NaN`, the result is `NaN`.
   - If one operand is ∞ and the other is zero, the result is `NaN`.
   - If one operand is ∞ and the other is non-zero finite, the result is ∞ with the computed sign.
   - If the exponent result overflows, an `Overflow` exception occurs, yielding ∞.
   - If the exponent result underflows, an `Underflow` exception occurs, yielding zero or a denormalized number.

Thus, the final floating point product is expressed as:

$$P = (-1)^{s_P} \times (M_P) \times 2^{E_P}$$

My design is not guaranteed to work across non special corner cases (non inf, non nan numbers), especially very large or very small numbers, with perfect IEEE-754 compliance and accuracy.

4 Compare

The compare logic is pretty simple, happens in a single clock cycle. The output is set to 3 registers called flags. {`gt, eq, lt`}. Takes A, B as input, and gives the comparision of (A, B)

1. Compare signs
2. Compare Exponents if same sign
3. Compare Mantissa if same sign and same exponent

5 Instruction Formatting

After some trial, and not so much error, I decided to set my instruction length as 24 bits. The instruction set uses three different instruction formats: R-format, S-format, and L-format.

5.1 R-format

The R-format is used for register-to-register operations. The encoding is as follows:

$$\underbrace{\texttt{opcode}}_{5b} \mid \underbrace{\texttt{rs1}}_{5b} \mid \underbrace{\texttt{rs2}}_{5b} \mid \underbrace{\texttt{ext}}_{4b} \mid \underbrace{\texttt{rd}}_{5b}$$

5.2 S-format

The S-format is typically used for store-type instructions, where an immediate value is required.

$$\underbrace{\texttt{opcode}}_{5b} \mid \underbrace{\texttt{rs1}}_{5b} \mid \underbrace{\texttt{rs2}}_{5b} \mid \underbrace{\texttt{imm}}_{9b}$$

5.3 L-format

The L-format is used for load-type instructions, which combine register and immediate addressing.

$$\underbrace{\texttt{opcode}}_{5b} \mid \underbrace{\texttt{rs1}}_{5b} \mid \underbrace{\texttt{imm}}_{9b} \mid \underbrace{\texttt{rd}}_{5b}$$

5.4 Opcodes

To support the instructions, and reduce complexity of decoder, I went with 5 bits for opcode, to encode 6 instructions at 2 precision levels. (12 ins). The opcode is formatted as $\{\texttt{precision}(b_4),\ \texttt{switch}(b_3),\ \texttt{funct3}(b_{2,1,0})\}$. The precision bit decides the choice of 32 or 64 bit paths for the arithmetic units.

b_2	b_1	b_0	b_3	Operation
1	0	0	0	Addition (ADD)
0	1	0	0	Subtraction (SUB)
0	0	1	0	Multiplication (MUL)
1	0	0	1	Load (LD)
0	1	0	1	Store (SD)
0	0	1	1	Compare (CMP)

Table 1: Opcodes

6 Datapath

The datapath consists of the Register File, Adder (32b, 64b), Multiplier (32b, 64b), Compare (32b, 64b), along with the memory bus and address bus. As I mentioned previously, the adders and multipliers are pipelined, but the control is not correct for hazard protection. For now, the control is done by manually measuring the cycles for completion.

```
 reg [4:0] pending_dest_stage1, pending_dest_stage2,
    pending_dest_stage3, pending_dest_stage4, pending_dest_stage5;
reg pending_write_stage1, pending_write_stage2,
    pending_write_stage3, pending_write_stage4, pending_write_stage5;
reg pending_mul_stage1, pending_mul_stage2,
    pending_mul_stage3, pending_mul_stage4, pending_mul_stage5;
reg pending_double_stage1, pending_double_stage2,
    pending_double_stage3, pending_double_stage4, pending_double_stag
wire hazard_detected;
```

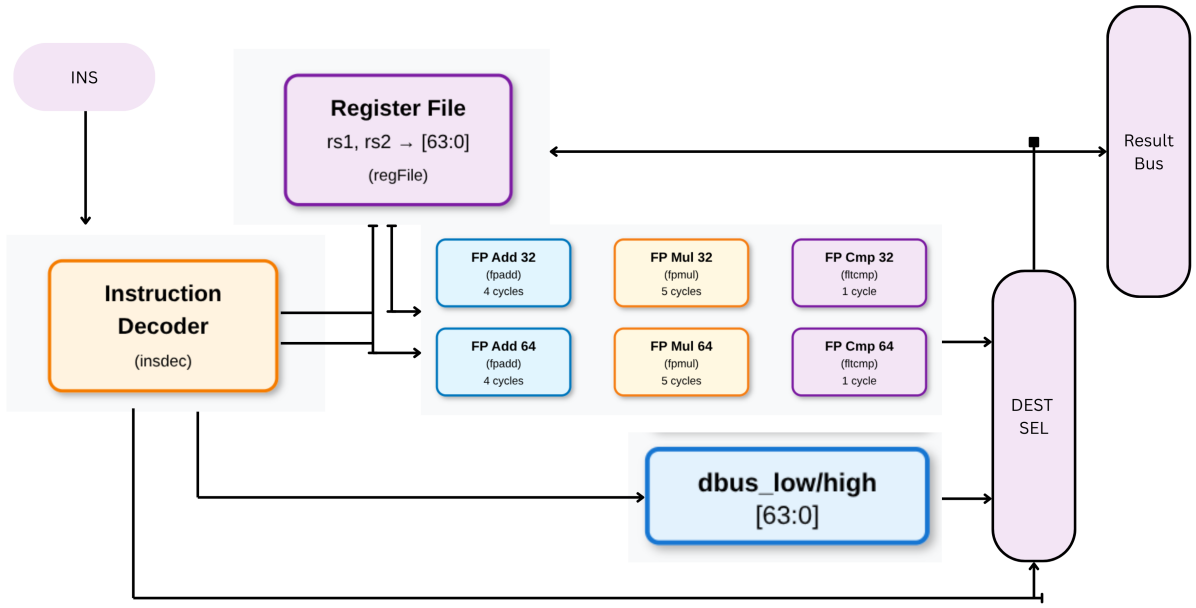

Figure 3: Datapath

In future improvements, I will aim to make the FP ALUs handshake with the hazard management. This will involve actual done signal assertion at the outputs.

7 Processor Interface

The top-level entity is the module `fputop_v1_0`. This module integrates the floating-point datapath with control logic, memory interface signals, and result buses. The interface signals are grouped into clock/reset, instruction input, data buses, and result outputs as described below.

7.1 Input Ports

- **clk**: System clock.
- **rst**: Active-high reset signal..
- **ins[23:0]**: 24-bit instruction input
- **dbus_low[31:0]**: Lower 32 bits of the input data bus for load
- **dbus_high[31:0]**: Upper 32 bits of the input data bus for load

7.2 Output Ports

- **result_low[31:0]**: Lower 32 bits of the computed result. This is wired to the input of the Register File, so as to see the transactions.
- **result_high[31:0]**: Upper 32 bits of the computed result.
- **bus1[31:0]**: Packed bus of flags, exception codes, and read write addresses.

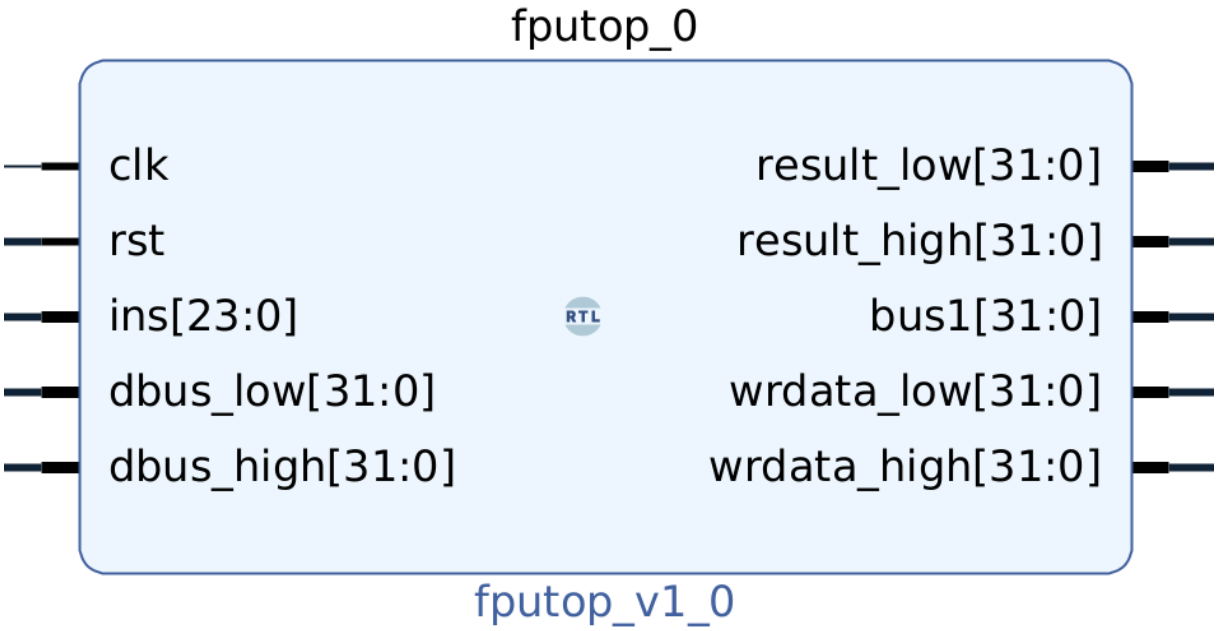

Figure 4: Top Module Ports

- **wrdata_low[31:0]**: Lower 32-bit data output intended for write-back into the system memory or register file.
- **wrdata_high[31:0]**: Upper 32-bit data output intended for write-back into the system memory or register file (for double-precision results).

7.3 External Interface

To get it working with any external interface, I used an AXI4 LITE peripheral, which was created by Vivado IP packager

When creating a custom AXI4-Lite peripheral, a register file is automatically generated to interface with the AXI bus. In this design, the peripheral was configured with 9 32-bit registers, denoted as `slv_reg0` through `slv_reg8`. Each register occupies a 4-byte aligned address location, as shown in Table 2.

Register	Address Offset
slv_reg0	0x00
slv_reg1	0x04
slv_reg2	0x08
slv_reg3	0x0C
slv_reg4	0x10
slv_reg5	0x14
slv_reg6	0x18
slv_reg7	0x1C
slv_reg8	0x20

Table 2: AXI4-Lite register map for the custom peripheral.

The IP packager automatically generates the AXI4-Lite protocol handling logic, including

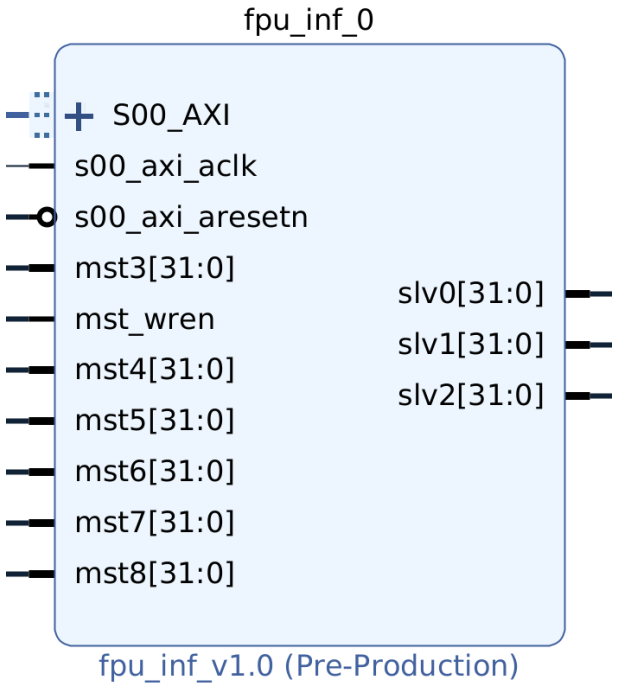

Figure 5: AXI Interface

read and write channels. To expose the registers, the code had to be modified. (Multiple times, because I kept miscalculating.) Watch out for the following.

1. They are not exposed as outputs in the IP package. This is not difficult to fix, as it just involves moving the declarations to i/o list (in 2 files).

```
//-- Signals for user logic register space example
//------------------------------------------------
//-- Number of Slave Registers 9
       // Just move these to the outputs port.
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg0;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg1;
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg2;
       .
       .
       .
reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg8;
```

2. All the registers are read-only by default. This one is a bit tricky, as it involves writing some sketchy, AXI non-compliant code.

```
        if (S_AXI_WVALID)
      begin // Code to write to regs
        .
        .
        else if (wr_en) begin // add this, can connect wr_en to 1
        slv_reg5 <= data5_in;
```

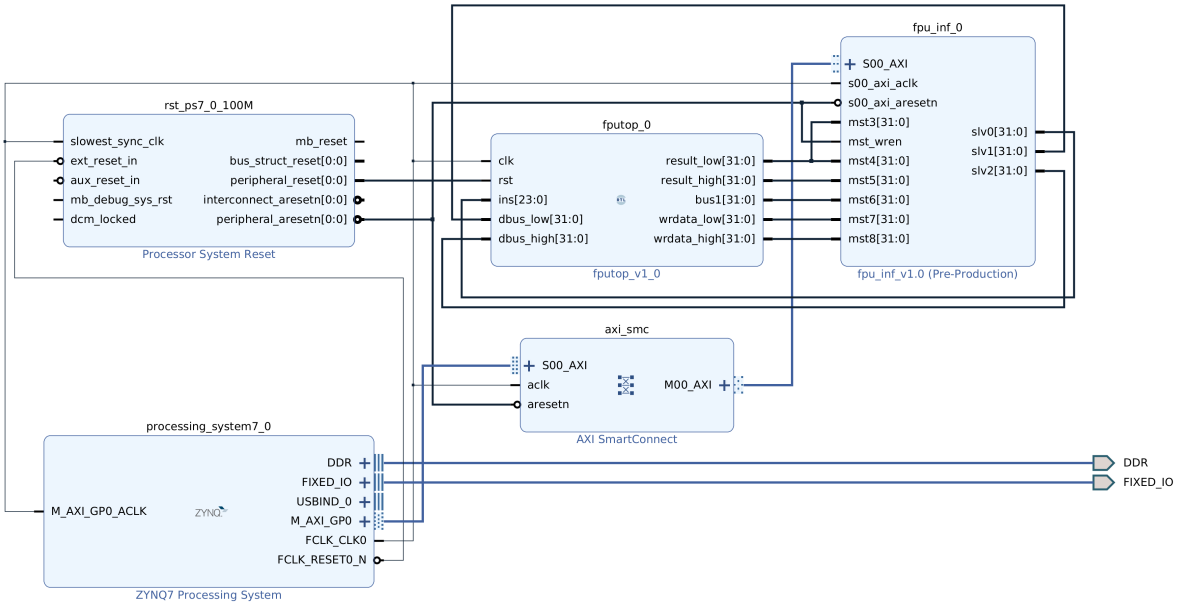

Figure 6: Zynq Setup

.

These registers are then connected to my `fputop` module, to provide inputs and probe outputs. You might be wondering why there are 2 registers connected to the same port. It is because I edited the IP too many times already, and it was a painstaking process. From the software side, the processor can access these registers through memory-mapped I/O.

8 Interfacing with ZYNQ

After writing the verilog for the FPU, it was time to setup the project in vivado.

8.1 Prerequisites and Board File Setup

8.1.1 Installing PYNQ-Z2 Board Definition Files

The PYNQ-Z2 board requires specific board definition files to properly configure the Zynq SoC

1. I downloaded the official PYNQ-Z2 board files from the TUL GitHub repository:

```
git clone https://github.com/TUL-Dev/TUL-PYNQ-Z2
```

2. Copied the board files to my Vivado installation.

```
cp -r pynq-z2 $XILINX_VIVADO/data/boards/board_files/
```

8.2 Vivado Project Creation and Configuration

Create a new project and configure with the PYNQ board.

8.2.1 Block Design Creation

After creating the project, first steps are to synthesize the verilog code that was written. After that, I did post-synthesis timing simulation to verify. After this, we instantiate the Processing System of ZYNQ, from the IP catalog, and run connection and block automations.

8.2.2 Block Design Integration

1. Added the custom AXI IP to the block design:
2. Connected the AXI interfaces (the registers to the top module and the AXI to the Master AXI of the ZYNQ7 PS)
3. The address mapper will auto generate the memory mapped IO and addresses.

8.3 Synthesis and Implementation

Before proceeding with synthesis, validate the design, and create a HDL wrapper. After setting this as top module, run synthesis, implementation and write bitstream. I had to fix a couple of errors on the way (before synthesis, in my verilog code). If all this works out, then we can proceed to exporting the generated bitstream file. Along with this, inside `project.gen/sources/bd/hw_handoff`, we find a `*.hwh` file, which is required.

9 Python Interface for the FPU Peripheral

On PYNQ, the AXI-mapped peripheral can be accessed directly from Python using the `pynq` library. Once the bitstream containing the packaged IP is loaded, the register space of the peripheral becomes accessible through memory-mapped I/O. The `Overlay` class provides an abstraction for the hardware design, and the `MMIO` class allows direct access to the AXI registers.

Listing 1: Python interface to the FPU peripheral.

```
from pynq import Overlay
from pynq import MMIO

# Load bitstream
ol = Overlay("design.bit")

# Base address and address range from Vivado Address Editor
base_addr = 0x43C00000
addr_range = 0x10000

# Create MMIO object
fpu = MMIO(base_addr, addr_range)

# Write operands and instruction
fpu.write(0x00, 0x00000002)    # ins (e.g., multiplication)
fpu.write(0x04, 0x3F800000)    # operand A = 1.0
fpu.write(0x08, 0x40000000)    # operand B = 2.0
```

```
18
19 # Read result
20 hex(fpu.read(0x0C))
```

Testing was done by writing an interface in Python, which assembled instructions 1 by 1, and checked the result register after sufficient number of clocks.