**EE(P) 538 A/E : Matrix to Machine—GPU Hardware Design on FPGA for AI**

**Project Milestone 1 : ALUs**

**DO NOT OPEN-SOURCE ANY MATERIAL FROM THE PROJECT IN THIS CLASS**

**Introduction :**

Welcome to the first milestone of the “EE(P) 538 A/E: Matrix to Machine—GPU Hardware Design on FPGA for AI” project. Over the next several weeks, you will progress from high-level C++ code all the way down to an FPGA-synthesized matrix multiplication deployed on AWS.

In **Milestone 1**, your goal is to implement fundamental arithmetic logic units (ALUs) using HDL and verify their correctness through the testbenches provided.

In particular, you will design two separate ALUs:

1. **Integer ALU**
   * Chunked Addition
   * Chunked Subtraction
   * Multiplication
2. **Floating-Point (FP) ALU**
   * FP Addition
   * FP Multiplication

These modules form the core building blocks of larger arithmetic operations and will be integrated into subsequent milestones. By the end of this assignment, you should have a working design capable of executing integer and floating-point operations, along with sufficient testbench evidence confirming its functionality. In future milestones, these components will be scaled up and combined to form a GPU-like architecture capable of running matrix operations on FPGAs.

Use Vivado to simulate your designs, and remember that the primary objective in this milestone is architectural correctness and clarity. Provide thorough documentation alongside your HDL code, and make sure to run the supplied testbenches to validate your ALUs. This milestone sets the stage for more advanced optimizations and system integrations in the subsequent steps of the project. Good luck!

**Code Setup :**

Below is a concise guide to the directory structure and instructions for setting up and running behavioral simulations of your Integer (Int) and Floating-Point (Float) ALUs in Vivado. This complements the previous information on what needs to be implemented (chunked add, chunked sub, integer multiplication, FP addition, and FP multiplication) for **Milestone 1**.

Please find an image of the directory structure of milestone 1 on the following page :

**A screenshot of a computer program

AI-generated content may be incorrect.**

This is a short explanation of the files in the directory :

1. **src/**
   * **float/** contains all Floating-Point ALU design files and their testbenches.
     + **project\_1/** contains the Vivado project file project\_1.xpr for the Floating-Point ALU design.
     + **float\_add\_pipeline.sv** / **float\_mul\_pipeline.sv**: Top-level RTL for FP add/mul pipelines.
     + **float\_add\_pipeline\_test.sv** / **float\_mul\_pipeline\_test.sv**: Testbenches for the above modules.
     + **partial\_multi\_cycle\_24bit\_2bpc\_float.v**: A specialized partial multiplier component used within the FP multiplier pipeline.
     + **float\_params.sv**: Parameter definitions for the floating-point modules (e.g., exponent/mantissa sizes).
     + **float\_test\_funcs\_test.sv**: Example test code that uses floating-point test functions (see **lib/float\_test\_funcs.sv**).
   * **int/** contains all Integer ALU design files and their testbenches.
     + **project\_2/** contains the Vivado project file project\_2.xpr for the Integer ALU design.
     + **chunked\_add.sv** / **chunked\_sub.sv**: Implementations of the chunked add/sub modules.
     + **mul\_pipeline\_32bit.sv** : 32-bit integer multiplication pipeline. Note that the other mul\_pipeline file in redundant and can be ignored.
     + **testbenches/**: Subdirectory containing testbenches for integer operations:
       - **chunked\_add\_test.sv**
       - **chunked\_sub\_test.sv**
       - **mul\_clocked\_test.sv**
   * **const.sv**, **assert.sv**, **assert\_ignore.sv**: Utility/SystemVerilog files that handle constants and assertions (commonly used in testbenches).
   * **LICENSE**: Open-source license information.
2. **lib/**
   * **float\_test\_funcs.sv**: Additional helper functions for floating-point testing.

**Opening the Projects in Vivado :**

To work on and simulate either the Integer ALU or the Floating-Point ALU, follow these steps:

1. **Launch Vivado**.
2. **Open the Appropriate Project File**
   * **Floating-Point ALU**:
     + Go to **File → Open Project**, then navigate to: ./src/float/project\_1/project\_1.xpr
     + Select project\_1.xpr and click **Open**.
   * **Integer ALU**:
     + Go to **File → Open Project**, then navigate to: ./src/int/project\_2/project\_2.xpr
     + Select project\_2.xpr and click **Open**.

Once loaded, Vivado will show the RTL source files and testbenches in the **Sources** window.

**Running Behavioral Simulations :**

Because we’re currently focusing on verifying functionality (and not on synthesis/implementation yet), you will run **Behavioral Simulations**:

1. **Select a Testbench (using “Set as Top”)**
   * For **FP ALU** (e.g., float\_add\_pipeline\_test.sv or float\_mul\_pipeline\_test.sv).
   * For **Int ALU** (e.g., chunked\_add\_test.sv, chunked\_sub\_test.sv, or mul\_clocked\_test.sv).
2. **Run the Simulation**
   * In Vivado’s **Flow Navigator**, expand **Simulation**.
   * Click **Run Simulation → Run Behavioral Simulation**.
   * The simulator will compile your design and testbench, then open a waveform window showing signals in real-time as the simulation runs.
3. **View/Analyze Waveforms**
   * Use Vivado’s waveform viewer to analyze the outputs of your ALU modules.
   * Confirm that each operation (Add, Sub, Multiply) produces the correct results.

Once you have completed the behavioral simulation, you can fix any issues, re-run the simulation, and iterate until the tests pass successfully.

**What you need to do for Assignment 1 :**

**Integer Add/Sub :**

1. Start off with the Integer ALU part of this milestone
2. Go to src/int/project\_2/project\_2.xpr and open this file on Vivado
3. First select chunked\_add\_test.sv as the top file.
4. chunked\_add had already been implemented completely. Try running the behavioural simulation and verifying the output.
5. Now go through the code and answer the following question in your writeup :
6. We named the adder “chunked” adder and not just the adder since we are “chunking” its operation. Explain more about how we are doing this. What is the advantage of “chunking” this adder over a non-chunked implementation?
7. Now, set chunked\_sub\_test.sv as the top file. Fill in any blanks with relevant code to make the behavioural simulation pass.
8. *Over here, you are free to use subtract (“-“) directly in Verilog.*

**Integer Multiply :**

1. Now set the top file as “mul\_clocked\_test.sv”.
2. We want you to finish a **4 stage multiplier**  with the below specs :
3. **IDLE**
   * **Function**: Waits for req to go high
   * **Transition**: Moves to the **MULTIPLY** state once req is detected
4. **MULTIPLY**
   * **Function**: Performs the multi-cycle multiplication steps
   * **Actions**:
     + Processes 2 bits of the multiplication each clock cycle
     + Accumulates partial results in an internal product register
   * **Transition**: Remains in **MULTIPLY** until all required bits of the multiplier have been processed, then moves to **FINISH**
   * **This is the state that needs to be filled in**, where we actually multiply the numbers with each other to get the final product
5. **FINISH**
   * **Function**: Signals the end of multiplication
   * **Transition**: Moves to **WAIT\_FOR\_REQ\_LOW** to prevent starting a new multiplication immediately
6. **WAIT\_FOR\_REQ\_LOW**
   * **Function**: Waits for req to return to 0, ensuring the system recognizes that the current multiplication cycle has completed
   * **Transition**: Returns to **IDLE**, ready for a new multiplication request

Run the behavioural simulation for this multiplier and ensure it passes for all test cases. *The blank for this section is on the “MULTIPLY” stage and CANNOT use the direct multiply Verilog operator (“\*”). We expect you to run a cycle by cycle multiplication, however, you are free to use more than 2 bits per cycle, if you want to.*

**Floating Add :**

Great! Now that you have completed your integer multiply, let’s move to the floating add!

1. Open a new Vivado window by opening src/float/project\_1/project\_1.xpr
2. Set “float\_add\_pipeline\_test.sv” as the top level file
3. We want you to finish a **3 stage FP adder** with the below specs :
4. **IDLE State**
   * **Objective**: Wait for a valid request (req) to begin the addition process.
   * **Behavior**:
     + When req is **low**, the adder remains idle, keeping output and acknowledge signals (out, ack) inactive.
     + When req is **high**, the module:
     + Captures the two input floating-point numbers ‘a‘,‘b‘‘a‘,‘b‘.
     + Aligns their mantissas. This typically involves comparing exponents and shifting the smaller mantissa so both numbers share the same exponent reference.
     + After alignment, the FSM **transitions** to **S1**.
5. **S1 State**
   * **Objective**: Perform the **core addition/subtraction** based on the signs of the inputs.
   * **Behavior**:
     + If the signs of the two inputs differ, the hardware effectively computes a subtraction (i.e., the larger mantissa minus the smaller).
     + If the signs are the same, it simply adds the two mantissas together.
     + The intermediate result (which may be wider than the final mantissa) is stored internally, and the FSM **moves** to **S2**.
6. **S2 State**
   * **Objective**: Normalize and finalize the result; then output it.
   * **Behavior**:
     + The adder checks if the result’s mantissa needs shifting (normalization). If so, it adjusts the mantissa (left- or right-shift) and updates the exponent accordingly.
     + Special cases are handled here, such as:
       - **Zero**
       - **Overflow**
     + Once normalization is complete, the final floating-point number is **assembled** and placed on the output bus (out).
     + The FSM then **returns** to **IDLE**, deasserting ack and waiting for the next operation.

Finish the blanks in the code and run the behavioural simulation to ensure all test cases are passing. Now, answer the following questions in your write-up :

1. Give the lines of the code handling the special cases in S2 ( Zero and Overflow). Explain these cases in detail and why they need to be handled separately.
2. How many cycles is the module output valid for in the floating add?
3. From the code, can you tell how the floating point values are stored in the registers? Can you explain the representation? Do you know what this representation is called?

**Note that you may see rounding errors in your answer for floating operations. You will not lose any marks for these as long as these errors are less than 1%**

**Floating Multiply :**

1. Now, let’s move onto the floating multiply unit. Set “float\_mul\_pipeline\_test.sv” as the top level file.
2. We want you to finish a **6 stage FP multiplier** :
3. **IDLE**
   1. **Objective**: Wait for a multiplication request (req).
   2. **Behavior**:
      1. If req is **low**, the multiplier stays idle, ack is held low, and no operation is performed.
      2. Once req is **high**, the FSM transitions to **UNPACK**, preparing to process the inputs.
4. **UNPACK**
   1. **Objective**: Extract and prepare the sign, exponent, and mantissa from the inputs.
   2. **Behavior**:
      1. Compute the sign of the product by XORing the input signs.
      2. Extract exponents (e.g., bits [30:23] in single-precision).
      3. Extract mantissas and prepend the implicit leading 1 (assuming normalized inputs).
      4. Trigger the internal mantissa multiplication hardware.
      5. Transition to **MULTIPLY**.
5. **MULTIPLY**
   1. **Objective**: Perform the actual mantissa multiplication in hardware.
   2. **Behavior**:
      1. Deassert start\_mul to let the submodule run autonomously.
      2. Wait for mul\_ack (or an equivalent signal), indicating the mantissa multiplication is complete.
      3. Once acknowledged, **add** the exponents of a and b, then **subtract the bias** (e.g., 127 for single-precision).
      4. Transition to **NORMALIZE** after capturing these intermediate results.
6. **NORMALIZE**
   1. **Objective**: Adjust the product’s mantissa so its leading bit is in the correct position.
   2. **Behavior**:
      1. Check the highest bit of the mantissa product
      2. If it’s set (meaning an overflow in the mantissa), **shift right by 1** and **increment** the exponent by 1 (think why this is required!)
      3. Otherwise, just select the correct slice of the product as the normalized mantissa.
      4. Transition to **SET\_EXP**.
7. **SET\_EXP**
   1. **Objective**: Finalize the exponent within valid IEEE 754 bounds.
   2. **Behavior**:
      1. If the computed exponent is **negative**, clamp or set it to zero (underflow condition).
      2. If the computed exponent is **above** the representable range, clamp it at max (overflow condition).
      3. Otherwise, simply keep the computed exponent.
      4. Move to **PACK**.
8. **PACK**
   1. **Objective**: Combine the sign, exponent, and mantissa back into the final floating-point format.
   2. **Behavior**:
      1. **Assemble** the output bits:
         1. [sign]
         2. [final\_exponent]
         3. [normalized\_mantissa]
         4. **Assert ack** to indicate the output is valid.
         5. Return to **IDLE** for the next operation.
9. Now, further note that for stage 3 (MULTIPLY), we have initialised an additional circuit to perform the multiplication in “partial\_multi\_cycle\_24bit\_2bpc\_float.v”. This multiplier should not be very different from the integer multiplier.

Finish the blank spaces in both “partial\_multi\_cycle\_24bit\_2bpc\_float.v” and “float\_mul\_pipeline.sv” and run the behavioural simulation to ensure all test cases are passing. Now, answer the following questions in your write-up :

1. Explain the mechanism of the “NORMALISE” state of the FP multiplier. Why does the mechanism we suggested work?
2. How many cycles is the module output valid for in the floating mul? Is this different from the floating\_add module?

**Note that you may see rounding errors in your answer for floating operations. You will not lose any marks for these as long as these errors are less than 1%**

**Grading and Submission :**

And with that, we’re done with milestone 1 of this project! Note that each group should submit the following on Gradescope by **January 30 2025, 11:59 pm** :

1. Zip file containing your code for the ALU. Name it as **Group\_<group\_number>.zip. We will send you your group numbers shortly**
2. Write-up containing answers A-F. Please name this write-up as **Group\_<group\_number>.pdf**

**Please keep all your code from this milestone for future ones.**  **You will need to re-use your code for milestone 2 and beyond.**

We will grade you for this milestone based on your answers to the questions in this milestone and the values outputted by your testbench. **Note that we do not have any hidden tests and will run only the test cases provided.** Since this is the first time we are offering this course and this project, we would love to get feedback on how we could have improved the set up of this milestone to improve the future ones. Please feel free to add any feedback you may have to your write-up.

For any doubts while working on this, please feel free to reach out to us on Ed!

**DO NOT OPEN-SOURCE ANY MATERIAL FROM THE PROJECT IN THIS CLASS**