1. Floating Point Accelerator & Testbench

Why it’s necessary to use an fp accelerator

Most devices nowadays use IEEE-754 standard floating point to record, compute and exchange non-integer data. Floating-point computation offers higher precision and dynamic range over fixed-point. However, such advantage comes with a cost, floating point units are more expensive and power-consuming over fixed-point units. Nevertheless, system that requires the range of floating-point arithmetic could implement their design using software emulation or dedicated floating-units.

Floating point arithmetic is applicable in IOT devices to record and process the environment statistics to increase precision. Most microcontrollers would handle floating point numbers by software emulation, yet a lightweight DSP could potentially accelerate the computation with lower power consumption. Neural networks are millions of neurons with sophisticated interconnections. They operate and train on large quantities of floating-point numbers. Since inner-product is the core operation to CNN networks, the ability of the hardware to multiply and sum a large number of floating-points is the key to success. In this homework, we aim to integrate a floating-point accelerator IP into the Aquila SoC.

To validate our accelerator, a benchmark program is designed. The program would initialize to vectors with defined length with random numbers, then calculate the inner-product of the two vectors using both software simulation and with our integrated floating-point accelerator. We would compare and discuss performances between the two implementation methods. The work would further instantiate our inner-product testbench to an array multiplication testbench, we would also discuss the results obtained.

1. Integrating Floating-point accelerator

The domain-specific accelerator (DSA), in our case the FPU, should communicate with the microprocessor through a bus protocol. Since Memory Mapped I/O (MMIO) architecture is implemented in Aquila, through an address decoder, we could directly access the accelerator through unified load/store instructions. To bridge the MMIO signals sending from the microprocessor and the AXI interface communicating with the DSA, we would construct a middlemen IP to move data around. The overall architecture is shown in

Xilinx Floating-point IP

Inner product calculation is a procedure of interleaving multiplications and additions, such operation could be break down into continuous fused multiply-add (a.k.a. Multiply-accumulate). A MAC accelerator outshines dedicated multiplier and adder in several reasons. Firstly, dedicated multiplication and addition introduces two times rounding, which MAC accelerator only rounds once. Such design would significantly increase accuracy, especially under the circumstance the operators are short is bit length like IEEE-754 single precision or IEEE754 half precision. Secondly, MAC units could merge addition and multiplication operations to achieve parallelism for better speed. Thirdly, introducing dedicated multiplier and adder would require data to flow between two accelerators. We have to design control signals to regulate the behavior which would require extra controllers. To conclude, it’s clear that we could benefit most from using a MAC as our floating-point accelerator.

Xilinx provides a Floating-point IP supporting fused multiply-add operations. We instantiate the IP operating on IEEE-754 single precision. The IP operates with blocking mode and the configuration is optimized for resource usage, it means with a slightly longer calculation time but only 2 DSP slice on our FPGA would be used. The IP runs with AXI4-stream interface with our middlemen IP.

Middlemen IP

The middlemen IP communicates with the MAC unit with AXI4-stream bus. There are 3 input lanes to send in two multiplicands and the addend. The answer flows out the MAC by the result lane. Each lane has dedicated interconnects with a data wire, a valid wire a ready wire. The master circuit sends the data with valid turned on, the slave responses with the ready wire.

The middlemen IP communicates with Aquila with MMIO interface. An address decoder would enable the middlemen IP with predefined memory address R/W actions. In our design the middlemen IP would buffer the data sent from Aquila, and keep sending them to the MAC. After the calculation is done we may answer the answer reading instruction coming from Aquila and deliver the answer.

Because we store the vectors in side a C array, memory access is in a patterned behavior. Little cache miss is observed in our experiment. Aquila is able to send in one floating point on about 7 clock cycles. On the other side of the circuit, we could process one MAC operation on average 19 cycles. The most naïve method is to store read in three operand and store them in registers. Once all data is buffered, we start the calculation. Such design is cheap in hardware with exchange of a fairly low speed, since it would take about 19 + 7\*3 cycles.

If we would sacrifice some extra LUT to build distributed RAM for both vectors, we may receive operands from Aquila while calculating at the same time. Distributed RAM is slightly faster than Block RAMs due to their reading is combinational. However, BRAM are larger in size and could support much larger amount of storage. We employ two 8192-unit buffers for both vectors.

More time could be saved if we eliminate unnecessary communications between the data-feeder circuit and Aquila. Since we aim to accelerate inner product calculation and the length of the vector is fixed. We could design the calculation behavior as follow: Aquila sends in the length of the vector first by MMIO, it is stored in 0xC202\_0000; After the vector length is received, Aquila would continuously send data from both vectors in an interleaving manner, where MAC operates at the same time. Vectors are stored from 0xC200 and 0xC201. Where the answer is continuously accumulated during the process (stored in 0x C203\_0000). After the calculation is completed, Aquila could again receive answer through the MMIO interface. Such design could effectively accelerate the calculation process, The calculation time now is about the time of MAC calculation, which is the bottleneck of the system.