Hello everyone! Now that the ten weeks of GSoC are almost done, I will share some updates about my work and some insights about participating in GSoC.

My Project

My project builds upon a previously-implemented joint project between Embecosm and Southampton university, which recently developed an open source ISA extension for a RISC-V core to accelerate neural network inference acceleration. A CV32E40P RISC-V core was extended with an accelerator to handle vector instructions to speed up the inference of neural networks. This project saw great success, showing a 5-fold increase in performance on the TinyMLPerf benchmark.

The verification of the AI vector extensions project currently exists only as a Verilator model. This limits the measurement of performance to cycle counts, and does not provide insight to any impact on clock speed in actual silicon. Furthermore, the current project has no pipeline and restricts all operations to be "single cycle". The aim is to tackle these issues, by bringing up the existing work on the Nexys A7 FPGA platform.

GSoC program has been challenging as well as exciting for me. With the persistent support of my mentors, I have been able to complete a lot of tasks towards running the AI accelerator on the Nexys A7.

My work has been uploaded to this github repo:
https://github.com/veronia-iskandar/core-v-mcu/tree/AI-Accelerator

Following are some details about my contributions and the work stages throughout the project.

1. Verification

I started this project by looking into the verification of the current design. The Southampton project had implemented self-checking test programs for the vector instructions, meaning they know which outputs to expect from the test cases and they report a pass or fail depending on the value they produced in reality. These test programs are written in a combination of C and assembly.

The inputs for the test cases were all hard-coded into the C code. I worked on extending the test cases to make them randomly generated. I wrote a python script to generate test cases and edit the C code accordingly. It also computes the correct output for each case and inserts those into the C code. This allows a large number of cases to be run automatically for all tests. Functional and line coverage were checked using Verilator.

As it turns out, all test cases at this stage passed without error. This still does not implicate that the design is bug-free. Verification of RTL to a high quality is an incredibly difficult process, often taking more than double the amount of effort as designing the RTL. Bugs can be hard to find and only apply to very specific cases, but can ruin the functionality of the design.

2. Core-v-mcu

The next step in the project was to bring up the cv32e40p on FPGA. I started with investigating how to bring the core without the integration of the accelerator. For that, I used the OpenHW group core-v-mcu project. The purpose of the core-v-mcu is to showcase the cv32e40p fully verified RISC-V core available from the Open Hardware Group. The cv32e40p core is connected to a representative set of peripherals such as UART, I2C master, I2C slave, QSPI master, CAMERA, SDIO, PWM, eFPGA with 4 math units. Using this repo, I built the bitstream for the Nexys-A7 board and downloaded it to the board.

3. Debugging

After the bitstream is running on the FPGA, I started working on running elf files on the riscv core. Openocd and gdb are used for connection to the Digilent hs2 debugger on the board and hence debugging the elf binaries. Since we are compiling bare-metal code, this stage also involved getting a linker script to assign the program sections to the correct addresses of the core-v-mcu RAM. As the debugging flow was rather not straightforward and required a lot of work to get it done, I documented the steps in the README of my forked github repo. There also I added the linker script and the openocd configuration file I used.

4. Accelerator bitstream

At this stage, I worked on integrating the AI accelerator code into core-v-mcu. This required some code changes for support of vector extensions and interfacing the cv32e40p with the accelerator. The main changes described here are in rtl/vendor/openhwgroup_cv32e40p directory.

The accelerator uses the RISC-V vector extension as a basis for its custom instructions, but not all of it is needed. Only a minimal system was needed so not all parts of the specification have been added. Support for 25 instructions has been added as well as 32 vector registers. The max element width has been constrained to 32b, the same as the CPU core width, to simplify transfer of data between the two. The max value for LMUL, the number of vector registers in a group, has been limited to four.

The CV32E40P features an Auxiliary Processing Unit (APU) interface. This follows a subset of the OBI interface used to communicate with system memory. It sends three 32-bit operands along with a 6 bit operand and a 15 bit flag set. The receiving core must respond with apu rvalid once the operation is complete. It is expected to return a 32 bit result and a 5 bit flag set. This interface is used by the optional floating-point unit (FPU). Since the FPU is not used in this design, the interface may be used for the accelerator instead. The core allows the decoder to configure the source for these operands, as with an ‘R’ format instruction. (<op> rd, rs1, rs2). The decoder is able to direct any of these registers any operand

(designated A, B, C). These operands are used for other core functions like the ALU. This is primarily useful for the vmv instruction, where scalar data can be loaded into the vector registers. In this case operand B is used to transfer the data.

Several minor modifications were required to the core RTL in order to better support the architecture of the accelerator, primarily in regards to multi-cycle instructions. The ability for operand A to pass the current vector being decoded, allowing the accelerator to also then decode the instruction and perform the appropriate operation. This is controlled by the CPU decoder through the alu op a mux sel o multiplexer.

Modification to the execute stage was required to prevent write-back in certain cases. Without this the core would execute vector instructions and then overwrite critical scalar registers such as the stack pointer with vector data. A pipelined signal was added to the decoder to allow it to disable register file writes once the accelerator returns r valid. Without this the CPU would interpret a vector instruction with a vd as rd and write the return value of the accelerator (typically VL) into that position in the CPU register file, potentially corrupting the ABI stack pointer.

The RISC-V vector specification requires the addition of six control and status registers (CSRs). Some CSRs were not added because they were not necessary in this implementation, such as vstart, the vector start position, which is not useful as partial completion of vector instructions is not permitted. Other CSRs were not added due to time constraints.

Building the integrated bitstream passed timing analysis and DRC checks and the bitstream was successfully downloaded to the board.

5. Debugging again

Next comes the debugging step again. Openocd is once more used to establish a connection. At first the connection seemed to fail due to the changes made to the bitstream. Further debugging this issue showed that the new bitstream changed the value of the misa CSR by setting bit 21 to 1 as our cpu supports vector extensions. This led Openocd to look for the vlenb CSR, which was not defined in the Systemverilog code. That led openocd to enter into an infinite loop of waiting.

To resolve this issue, I looked into the definitions of the CSR registers. The CV32E40P processor contains its own CSRs which are required by the base RISC-V specification. The accelerator CSRs, however, were not included in the same register file, and thus not readable by privileged instructions. This decision was initially made to compartmentalise the accelerator and the processor core. However, to solve the Openocd connection issue, I modified the code to define vlenb along with the main processor CSRs and make its address available to Openocd.

Afterwards I set up gdb to debug elf files. At this point, pure C programs ran without issue while programs with assembly vector extensions instructions still did not perform correctly. An analysis of possible issues follows.

What is coming up next?

Future work involves more robust testing of the accelerator to identify why vector instructions fails on FPGA although they work on Verilator.

The first concern I had was clock timing. The complex features of the accelerator have been designed as a single-stage system with no pipeline. This is not a problem in Verilator, which has no concept of timing constraints. On a physical device however, the accelerator could be much slower than the CPU core, which has a four-stage pipeline. That said, the initial timing analysis while building the bitstream did pass the timing constraints and did not report any violations. A more detailed check of critical paths could be needed.

There is a second concern when synthesising the design on an FPGA. With complex designs, the gate-level design produced by the synthesis tool sometimes does not match the behavioural model in SystemVerilog, and therefore behaves differently to simulation. To test this would require further verification of gate-level designs produced by the synthesis tool.

Take-aways from my experience in GSOC

GSOC is a great platform to explore new areas, maybe even discover a new career path. GSoC is not an internship and should not be looked at as a way to earn money. It is definitely not something you should pursue half-heartedly or just for the sake of it. It is a great chance to learn new stuff. You have a chance to interact and be a part of a community. Mentors and community members are experts in their fields, and getting an opportunity to connect with them is incredible. You also get to work with the freshest technologies on a large codebase. Knowing the fact that your code will likely be used by many developers is itself a considerable motivation. GSoC will significantly increase your skillset and provide a sense of accomplishment.

The process starts with finding the right project. You will be working on the project for a lot of time, so it is very crucial that you find a project that interests you. For me, I wanted to do a RISC-V related project, so I opted for applying with the FOSSI organization. I checked the ideas list, and found many interesting projects. This particular project interested me because in my study program I had already been investigating porting a customized RISC-V core to an FPGA. Afterwards, I contacted the mentors and started working on the proposal. The proposal should be ambitious, while also being achievable. It wouldn't do if you just did something trivial, while overcommitting will lead to failure later on. The proposal should also include a risk register to plan for unexpected delays while you are working.

One key takeaway from my work so far - if you can’t figure out an answer to your doubts after a while of digging, ask your questions, understand that coming from a fresh eye, your perspective is valuable. However, also keep in mind that if you can’t express your questions clearly, it’s useless, so learn how to write a clear report to help the mentors and community members help you.