# ECE-GY-9413 Course Project

**Part 1 - Submission due on February 23, 2024 at 11:59 PM**

---

The first part of the course project requires you to implement a functional simulator for the presented vector ISA following the prescribed specifications. The machine is reminiscent of VMIPS but we will not consider off-chip data movement for simplicity.

You should implement your simulators in Python 3. Please DO NOT use any external libraries that are not part of the original Python package. You must write your own test programs and show they execute correctly. To limit engineering effort, you can directly process assembly instructions, there is no need to consider actual instruction encodings and machine code.

Machine learning will be used as a motivating example and you are expected to use your simulator to implement a fully connected and convolution layer. In addition, this year we will ask you to also implement an FFT and, optionally, a sparse matrix multiplication.
For the initial functional simulator you only need to submit a dot product. Details follow below.

## Functional Simulator:

For part 1, you will only be required to show your implementation of the functional simulator for the given ISA specification and its output from running a dot product with it. The functional simulator should only contain registers and memory that form the architectural state and should simulate how each instruction modifies that state.

The simulator must take as input the instructions in assembly format and the initial states of the Scalar and the Vector Data memory and give out the final state of the vector register file, scalar register file, and the data memories. The simulator must take the directory containing the input files as a command line input. The output files must be placed in the same directory. Your simulator must run with the following command:

```
<yournetid>_funcsimulator.py --iodir <path/to/the/directory/containing/your/io/files>
```

**Input:** The simulator should take the following files as inputs:

- Code.asm: The file should contain your assembly code for the test function. The sample file (Code.asm) released with this document shows the syntax for all the instructions in the ISA.
- SDMEM.txt: The file should contain the initial state of the SDMEM containing the data required for the test function in integer format. Each line in this file represents one word (32 bit) of data in the SDMEM.
- VDMEM.txt: The file should contain the initial state of the VDMEM containing the data required for the test function in integer format. Each line in this file represents one word (32 bit) of data in the VDMEM.

**Output:** The simulator should output the following files:

- VRF.txt: The file should show the final state of the Vector Register File after the execution of all the instructions in the input Code.asm file. Each line should contain a comma separated list of integer values showing all the elements of a vector register.

- SRF.txt: The file should show the final state of the Scalar Register File after the execution of all the instructions in the input Code.asm file. Each line should contain an integer value showing a scalar register data.
- SDMEMOP.txt: The file should contain the final state of the SDMEM after the execution of the Code.asm file. The format should be the same as the input SDMEM.txt.
- VDMEMOP.txt: The file should contain the final state of the VDMEM after the execution of the Code.asm file. The format should be the same as the input VDMEM.txt.

A sample input and output files are released along with this document. Please ensure that your input and output files conform to the format in the sample files.

## ISA Specification:

Vector length of 64, 32b integer elements.

The architectural state of the vector processor has the following components:
**VDMEM:** Vector Data Memory with a capacity of **512 KB**, word addressable.
**SDMEM:** Scalar Data Memory with a capacity of **32 KB**, word addressable.
**Register Files:**

- Scalar Register File: 8 Scalar Registers of 32 bits each.
- Vector Register File: 8 Vector Registers each with a maximum capacity of 2048 bits or 256 bytes or 64 32-bit elements. Maximum Vector Length is set to 64.
- Vector Mask Register: 1 Vector Mask Register also known as the Flag Register. Contains 64 1 bit values. The vector operations are valid only for the elements with corresponding flag register value set. Clear this register if all the elements are valid.
- Vector Length Register: 1 Vector Length Register of size 32 bits to contain the number of vector element operations. Set this to MVL if all the elements of the vector register inputs are to be evaluated.

The table below provides the set of instructions to be supported by the processor and their details:

| Instruction Type | # | Instruction | Operands | Semantics |
|---|---|---|---|---|
| Vector Operations | 1 | ADD/SUBVV | VR1, VR2, VR3 | Add/Sub VR2 and VR3 to VR1 |
| | 2 | ADD/SUBVS | VR1, VR2, SR1 | Add/Sub scalar SR1 to each VR2, store in VR1. |
| | 3 | MUL/DIVVV | VR1, VR2, VR3 | Multiply/Divide VR2 and VR3 to VR1. |
| | 4 | MUL/DIVVS | VR1, VR2, SR1 | Multiply/Divide scalar SR1 to each VR2, store in VR1. |
| Vector Mask Register Operations (Set VM | 5 | S__VV | VR1, VR2 | Compare each element of VR1 with the corresponding element in VR2 and set the respective bit in the Vector Mask Register to 1. |

| | | | | |
|---|---|---|---|---|
| instructions take 6 forms for __: EQ, NE, GT, LT, GE, LE) | 6 | S__VS | VR1, SR1 | Compare each element of VR1 with the scalar value in SR2 and set the respective bit in the Vector Mask Register to 1. |
| | 7 | CVM | - | Clear the Vector Mask Register - set all bits to 1. |
| | 8 | POP | SR1 | Count the number of 1s in the Vector Mask Register and store the scalar value in SR1. |
| Vector Length Register Operations | 9 | MTCL | SR1 | Move the contents of the Scalar Register SR1 into the Vector Length Register. |
| | 10 | MFCL | SR1 | Move the contents of the Vector Length Register into the Scalar Register SR1 . |
| Memory Access Operations | 11 | LV | VR1, SR1 | Load vector elements into VR1 from the Vector Data Memory starting at address SR1 with unit stride. |
| | 12 | SV | VR1, SR1 | Store vector elements in VR1 to Vector Data Memory starting at address SR1 with unit stride. |
| | 13 | LVWS | VR1, SR1, SR2 | Load vector elements into VR1 from the Vector Data Memory starting at address SR1 with a stride value in SR2. |
| | 14 | SVWS | VR1, SR1, SR2 | Store vector elements in VR1 to Vector Data Memory starting at address SR1 with a stride value in SR2. |
| | 15 | LVI | VR1, SR1, VR2 | Gather elements into VR1 from VDM with SR1 as the base address and corresponding elements in VR2 as offset |
| | 16 | SVI | VR1, SR1, VR2 | Scatter elements in VR1 to VDM with SR1 as the base address and corresponding elements in VR2 as offset |
| | 17 | LS | SR2, SR1, Imm | Load Scalar value into SR2 from the Scalar Data Memory at address (SR1 + Imm) |
| | 18 | SS | SR2, SR1, Imm | Store Scalar value in SR2 into the Scalar Data Memory at address (SR1 + Imm) |
| Scalar Operations | 19 | ADD/SUB | SR3, SR1, SR2 | Add/Sub scalar values in SR1 and SR2 and store result in SR3. |
| | 20 | AND/OR/XOR | SR3, SR1, SR2 | Perform bitwise And, Or, and Xor on scalar values in SR1 and SR2 and store result in SR3. |

| | | | | |
|---|---|---|---|---|
| | 21 | SLL, SRL | SR3, SR1, SR2 | Logical Left/Right Shift the value in SR1 by the value in SR2 and store the result in SR3. |
| | 22 | SRA | SR3, SR1, SR2 | Arithmetic Right Shift the value in SR1 by the value in SR2 and store the result in SR3. |
| Control (__ takes six options: EQ, NE, GT, LT, GE, LE) | 23 | B__ | SR1, SR2, Imm | Compare scalar values in SR1 and SR2 and update PC to PC + signed(imm) if comparison result is true. Imm can take values from -(2^20) to (2^20). |
| Register-Register Shuffle | 24 | UNPACKLO | VR1, VR2, VR3 | Store the first (lower) half of vector elements of VR2 and the first (lower) half of vector elements of VR3 into VR1 in interleaved fashion.<br>E.g., VR1[0] = VR2[0]<br>VR1[1] = VR3[0]<br>VR1[2] = VR2[1]<br>VR1[3] = VR3[1] |
| | 25 | UNPACKHI | VR1, VR2, VR3 | Store the second (upper) half of vector elements of VR2 and the second (upper) half of vector elements of VR3 into VR1 in interleaved fashion.<br>E.g., VR1[0] = VR2[N/2]<br>VR1[1] = VR3[N/2]<br>VR1[2] = VR2[N/2 + 1]<br>VR1[3] = VR3[N/2 + 1] |
| | 26 | PACKLO | VR1, VR2, VR3 | Store even indexed elements of VR2 to the first half of VR1 and even indexed elements of VR3 to the second half of VR1.<br>E.g., VR1[0] = VR2[0]<br>VR1[N/2] = VR3[0]<br>VR1[1] = VR2[2]<br>VR1[N/2 + 1] = VR3[2] |
| | 27 | PACKHI | VR1, VR2, VR3 | Store odd indexed elements of VR2 to the first half of VR1 and odd indexed elements of VR3 to the second half of VR1.<br>E.g., VR1[0] = VR2[1]<br>VR1[N/2] = VR3[1]<br>VR1[1] = VR2[3]<br>VR1[N/2 + 1] = VR3[3] |
| Halt | 28 | HALT | - | Stop execution. |

**Verification:** You should verify your functional simulator using a function that computes the dot product of two vectors of 450 elements each and store the result in to address 2048 of VDMEM.

The function should implement the following equation,

$DP = a[0] * b[0] + a[1] * b[1] + .... + a[449] * b[449]$,
where,
$a = \{0, 1, 2, 3, .... 449\}$
$b = \{0, 1, 2, 3, .... 449\}$

Additionally, you have to implement a function that verifies all 28 base instructions and variants (e.g., SGTVV) in the ISA and submit its output.

You should include a script that runs through all test programs and verifies correctness. A (1 page or less) document should include a description of the test program and show which instructions are used in each test.

**Evaluation Criteria:**
Your implementation will be evaluated using the test files submitted and our test file implementing the same dot product function. The output from your implementation will be matched against a golden test result. Points will be given based on the following criteria:

- Correct Dot Product: 50%
- Correct implementation and test of each instruction: 30%
- Proper comments and following submission instructions: 20%

**Deliverables:**

1. Your Python script(s) implementing the functional simulator. The name of the main script should be <netid>_funcsimulator.py.
2. Two test input folders: One containing the dot product program and the ISA test and one with the outputs from running them through your simulator.
3. A short description of the test program and instructions to show completeness.

Please zip the files and use your net id in the final zip file name.