ECE-GY-9413 Course Project

Part 3 - Submission due on April 29, 2024 at 11:59 PM

For the third part of the project, you are required to implement a parameterized performance model simulating the timing of the vector microarchitecture specified in Part-1. The microarchitectural specifications are listed in this document. 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.

You should use the functional simulator to verify correctness of programs, then simulate performance with this (Part 3) simulator.

Timing Simulator - Introduction:

A timing simulator is a type of performance model that takes a function (e.g., assembly program) as input and outputs the time it takes for a particular microarchitecture to process the sequence of instructions. The simulator need not show the cycle by cycle movement of data through the microarchitecture, nor functionally simulate the program (that is a much harder task). Instead, the timing simulator uses its understanding of the time it takes to execute different instructions to abstract many details. For example, when a multiply instruction is issued to the lanes, you do not need to simulate the movement of data and each cycle of the pipeline (as you did with the MIPS simulators), instead you can simply count how long the instruction takes.

Thus, the majority of the effort in this project will be in the frontend model of the simulator. This includes fetching, decoding, and dispatching instructions to the backend. You should pay close attention to how dependence and cycle delay is calculated, think–how long does a vector load instruction take given banked VDM?

Specifics:

Your simulator must take the directory containing the input files as a command line input. Your simulator must run with the following command:

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

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

1. Code.asm - The file should contain your assembly code for the test function. Refer the sample file (Code.asm) released with Part 1 for syntax.
2. 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.
3. 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.
4. Cong.txt - The file should contain the configuration (list of parameters and their values) to which the model should be constructed for. A sample configuration file showing the initial set of parameters is released with this document.

Output: The simulator should output the cycles it takes to run the input function.

Tasks:

- Design the base microarchitecture and make a block diagram to show the control and the data path. The more details expressed in the block diagram the better.
- Implement the timing simulator to model your design.
- Measure and plot the performance of your design for the assembly functions you have created in Parts 1 & 2 (Dot Product, Fully Connected Layer and Convolution Layer) by varying the model parameters (different configurations).
- Extra credit (or for groups of 3): Propose one optimization that will improve the performance of the design, implement them in the model and measure speedup for extra credit! These optimizations can be anything ranging from rearranging code to architectural optimization to microarchitectural optimization. The optimizations can be generic or specific to the workload/kernels we are using for measurement.
- Write a conference style report outlining the principles behind your optimizations and explaining your results. Please use the latex format given for the weekly report submission.

Base Microarchitecture Specifications:

Frontend:

1. Fetch Stage: A single instruction memory will hold the entire program - same as in part 1. You should assume it takes 1 cycle to fetch the instructions from the Imem. The branch instructions get resolved at this stage.
2. Decode Stage: The decode stage interprets the instructions and checks for data and structural hazards, this takes 1 cycle. The data dependencies can be tracked by using a Busy Board. The busy board is a binary vector that tracks whether or not a register is being used by an inflight instruction. When instructions leave the decode stage, they are marked (set high) in the busy board. When instructions complete, the registers in the busy board are cleared. The decoded instructions are then placed into one of the two dispatch queues.
3. Dispatch Queues: There are three queues for instructions: one for vector data, one for vector compute, and one for scalar operations. The queues are *decoupled* meaning that they can overlap execution of compute and memory instructions. Once the structural hazards in the backend are resolved, instructions at the head of each queue can run in parallel. Only one instruction from each can pop off the queue per cycle. You should assume that the scalar memory instructions are placed in the scalar queue. The depth of the queues must be configurable using the parameters: *dataQueueDepth, computeQueueDepth*.
4. Get the dynamic flow of the instructions from the Functional Simulator you designed for the first part of the project and run that through the Timing Simulator. This is to resolve the control flow of the test program (branching instructions) and the addresses for the load/store instructions. An example program and its dynamic flow are shown below:

Original program flow run on functional simulator	Resolved program flow to be run on timing simulator
LS SR1 SR0 0 # Loads 128 into SR1 **LS SR2 SR0 1** # Loads 64 into SR2 **LS SR4 SR0 2** # Loads 2 into SR4 **ADD SR0 SR0 SR2** # increments SR0 by 64 **LV VR0 SR3** # Loads 64 elements into VR0 starting from address 0 with stride 1. **LVWS VR1 SR3 SR4** # Loads 64 elements into VR1 starting from address 0 with stride 2. **LVI VR2 SR3 VR0** # Gathers 64 elements into VR2 with base address 0 and offsets in VR0.	**LS SR1 (0)** # Load scalar from address 0. **LS SR2 (1)** # Load scalar from address 1. **LS SR3 (2)** # Load scalar from address 2. **ADD SR0 SR0 SR2** # This should go into the scalar queue. **LV VR0 (0, 1, 2, ... 63)** # The addresses of the vector loads are resolved. **LVWS VR1 (0, 2, 4, ... 126)** # The addresses of the vector loads are resolved and given explicitly by adding the stride value to the offset VLR number of times.

BLE SR0 SR1 -4 # branch to PC 3 till SR0 is 128 **HALT**	**LVI VR2 (0+x, 0 +y, ... 0+z)** # The addresses of the vector loads are resolved by adding the base address with values in VR0. **B (3)** # Branch to PC 3 - Address of the branch resolved and given explicitly. **ADD SR0 SR0 SR2** # This should go into the scalar queue. **LV VR0 (64, 65, 66, ... 127)** # The addresses of the vector loads are resolved. **LVWS VR1 (128, 130, 132, ... 254)** # The addresses of the vector loads are resolved and given explicitly by adding the stride value to the offset VLR number of times. **LVI VR2 (64+x, 64+y, ... 64+z)** # The addresses of the vector loads are resolved by adding the base address with values in VR0. **B (8)** # Branch to PC 8 - Branch not taken. **HALT**

Backend:

1. All scalar instructions, the CVM and the POP operations take 1 cycle each in the backend.
2. You should assume that all program data is preloaded into the data memory and everything goes to/from the vector register file. The ISA supports VDM accesses as unit, strided, and scatter/gather. The VDM is multi-banked and bank conflicts must be resolved in the microarchitecture.
3. The number of banks in the VDM and the depth of the vector Load Store Pipeline should be configurable using the parameters: *vdmNumBanks, vlsPipelineDepth*.
4. You can assume that all registers in the vector register file can be accessed simultaneously and that sufficient bandwidth is available to facilitate the incoming/outgoing data to/from the compute lanes and VDM.
5. You should implement separate vector pipelines that support multiplication, addition/subtraction, division and register shuffle.. The number of lanes and the depth of the pipelines should be configurable using the parameters: *numLanes, pipelineDepthMul, pipelineDepthAdd, pipelineDepthDiv, pipelineDepthShuffle*.
6. You should think about how to model the latency of instructions based on bank conflicts, VDM partitions, and number of lanes. You do not and should not model the actual data moving through the machine, you simply need to time how long each instruction takes.

Evaluation Criteria:

- Show you are able to appropriately process the 3 test function codes using your performance and functional simulator.
- The functional simulator will be used to verify the functions are computed correctly using real inputs.
- The performance/timing simulator will be used to check the performance of the functions using a variety of parameter settings (e.g., VDM partitions, number of lanes, etc).
- You should carefully state any assumptions and provide detailed pipeline diagrams of your machines in your final report.
- Proper care must be taken to make sure all code is usable. Please follow the instructions carefully.

Deliverables:
A conference-style report explaining your work, experimentation methodology, and results. This should include a link to your git repository containing the implementation with commits made no later than 4/29/2024. Make sure to add a README file to the repository to list your team members and explain the usage of your simulator.

Suggestions, References, and Useful Links:

- For block diagrams: https://app.diagrams.net/
- For Latex reports: https://www.overleaf.com/
