VMIPS SIMULATOR

Parallel and Customized Computer Architecture
ECE 9413 A

Jayanth Rajaram Sastry
NYU Tandon School of Engineering
MS Computer Engineering
js12891

Dhana Laxmi Sirigireddy
NYU Tandon School of Engineering
MS Computer Engineering
ds6992

***Abstract*—The goal of this project is to implement a functional and timing simulator for a vector machine ISA [1]. The functional simulator models the architectural state and simulates how each instruction modifies the state. The timing simulator extends the functional simulator to model the pipeline and timing aspects of the microarchitecture. The implementation was carried out using Python 3, without relying on external libraries, and was tested on test cases for the Dot Product and Fully Connected Layer operations.**

***Index Terms*—Vector, Simulator, Microarchitecture, Pipeline, Python3, Assembly Language**

I. Introduction

A simulator is a computer program that imitates the behavior of a system or process by simulating its operation under different conditions. Designing a simulator is necessary because it allows architects and designers to evaluate the performance of a processor or system before actual implementation. This helps identify potential bottlenecks and performance issues early on in the design process, leading to better designs and more efficient processors.

Functional Simulator - To simulate the behavior of a processor at the level of instruction execution. In this project, for the given ISA specification we implemented the functional Simulator using Python and demonstrated its output by running a dot product. The registers and memory that form the architectural state are modified as per how each instruction is executed(simulated).

It takes the instructions in assembly format stored in Code.asm and the initial states of the Scalar and the Vector Data memory, stored in SDMEM.txt and VDMEM.txt respectively as input. The simulator then executes these instructions and modifies the state of the processor's vector register file, scalar register file, and data memories based on their execution. The final state of the vector register file is stored in VRF.txt, the scalar register file in SRF.txt, and the data memories in SDMEMOP.txt and VDMEMOP.txt.

Timing Simulator - A performance model that takes an assembly program as input and predicts the time it takes for a microarchitecture to execute the sequence of instructions. Unlike functional simulators that only simulate how instructions modify the architectural state, timing simulators consider the time it takes for each instruction to execute in a specific microarchitecture.

In this project, we designed a timing simulator with a frontend and backend model. The frontend model is responsible for fetching, decoding, and dispatching instructions to the backend model. In the frontend model, the instructions are read from Code.asm file, decoded into micro-operations typically using the functional simulator, and dispatched to the backend for execution. This is where instruction dependencies and scheduling are determined. The backend model, on the other hand, executes the micro-operations and simulates the timing of the processor across the pipeline stages as per the configuration parameters mentioned in the Config.txt file.

II. Timing Simulator Working Mechanism

Every Instruction goes through 3 stages- Fetch, Decode and Execute. Fig. 1 shows the block diagram of the timing Simulator for the default parameters given the Config file.

- In our simulator, we have defined several flags for tracking and understanding the functioning of our logic. The Fetch flag, decode flag, and pipeline flag are used to indicate whether their corresponding function can be called or not. If the flag is set to 1, the function cannot be called.
- The Halt flag is set when the HALT instruction arrives in the fetch stage. This flag stops the simulation at the end of the current cycle. The Terminate flag is set to indicate the termination of the simulation. When this flag is set, the simulation stops at the end of the current cycle and returns the final output.
- The Bank Conflict flag is set to define if there are any conflicts accessing the memory banks. When this flag is set, the simulator handles bank conflicts in later stages to ensure that the instructions are executed correctly.
- We have used Modified IMEM which contains all instructions in the order of execution along with address

list and vlr. This is generated upon running the functional simulator.

- Data Hazards are handled by 2 busy boards- 1 for scalar and 1 for vector which keep track of the registers being used.
- We have separate instruction memory , vector data memory (VDMEM) and scalar data memory(SDMEM) hence we don't have to worry about control hazards.
- The vector data memory is banked.
- 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.
- **DJ formula** is determined to calculate the number of elements that gets into the Pipeline per lane.
- We also handle memory banking conflicts and make use of a first come first serve algorithm: If 2 lanes try to access the same memory bank the lane which tried to access the bank first is given priority if both access at the same time the lane with the lower number is given priority. Whenever many lanes have a bank conflict we move one lane and stall the others.
- **Jayanth Lane Bank Theory** formulated for uniform strides - an observation related to memory access in instructions such as LVWS or SVWS, and are spread across multiple banks.
- Multiple lanes are considered to improve parallel execution.
- We optimized our timing simulator by implementing the concept of chaining

A. FRONTEND

1) ***STAGE 1 - FETCH:*** The fetch stage retrieves instructions from a modified instruction memory using the point, Program Counter and provides them as input to the decoder. The modified IMEM is generated by a functional simulator, which unwraps all branch instructions. Each line in the modified IMEM includes the instruction (with its destination and source registers), Address list, and a Vector length register (VLR). For load and store instructions, the address list for the respective element is provided, making it easier to handle bank conflicts in later stages. The fetch stage is called only if the fetch flag is 0.

2) ***STAGE 2 - DECODE:*** The decode function takes input from the fetch function and uses a helper function to decode the instruction, which produces a variable called "sim". This "sim" variable contains information about the instruction, such as its destination, sources, VLR, and instruction code.
The decode logic is called only if the decode flag is set to 0. If the decode flag is not 0, then the fetch flag is also set to 1, indicating a pause in the fetching of instructions. Once the "sim" variable is produced, it is sent to the Dispatch Queue for further processing. The Dispatch Queue is likely an array that manages the ordering and execution of instructions in the pipeline.

3) ***DISPATCH QUEUE - umpire:*** The Dispatch Queue is an array that dispatches instructions into the pipeline for the execution stage, acting as an intermediary between the frontend and backend of the timing simulator.

A busy board is maintained to dynamically push instructions into the next stage. It's an array of register indices indicating whether the corresponding register is currently being used by an instruction. If the busy board source register indices are 1 or if the queue's length is full, the instruction cannot proceed to the next stage, and all previous stage flags are set to 1 to indicate that the pipeline is stalled. If the busy board index is already 0, there's no dependency of that register in front of the pipeline. The instruction is pushed into the dispatch queue, and the corresponding busy board destination register index is set to 1 for further processing.

The dispatch stage manages the order in which instructions are executed in the pipeline and ensures that data dependencies are satisfied to avoid data hazards. For scalar instructions, a scalar queue and a scalar busy board are maintained. Three queues are maintained for data type instructions (Load Store), Computational type (Arithmetic and logical instructions), and scalar only instructions. Two busy boards are maintained for vector and scalar register indices. The dispatch queue depth is given in the Config file as dataQueueDepth and computeQueueDepth

if *busy board source register index is 1 or dispatch queue is full* **then**
  1) set previous stage flags(fetch and decode) to 1;
else
  1) set busy board destination register index to 1;
  2) append the instruction into Dispatch queue;
  3) set all stage flags to 0;
end

Algorithm 1: Dispatch Queue logic

B. BACKEND

1) ***STAGE 3 - EXECUTE: PIPELINING:*** After instructions are dispatched from their queues, they travel across the pipeline depth until they reach the end, where the corresponding instruction is executed, only if the pipeline flag is 0. The number of lanes and pipeline depth parameters are specified in the configuration file. Each lane has one VLS pipeline (for load store), MUL, DIV, and ADD pipeline. Scalar instructions are executed in one cycle, indicating a pipeline depth of one, irrespective of data type or compute type instruction. We implement proper cycle by cycle movement of the instructions in the pipeline, so that it would be easy to debug.

The DJ formula, we formulated, is used for determining the number of elements that go into the pipeline per lane in

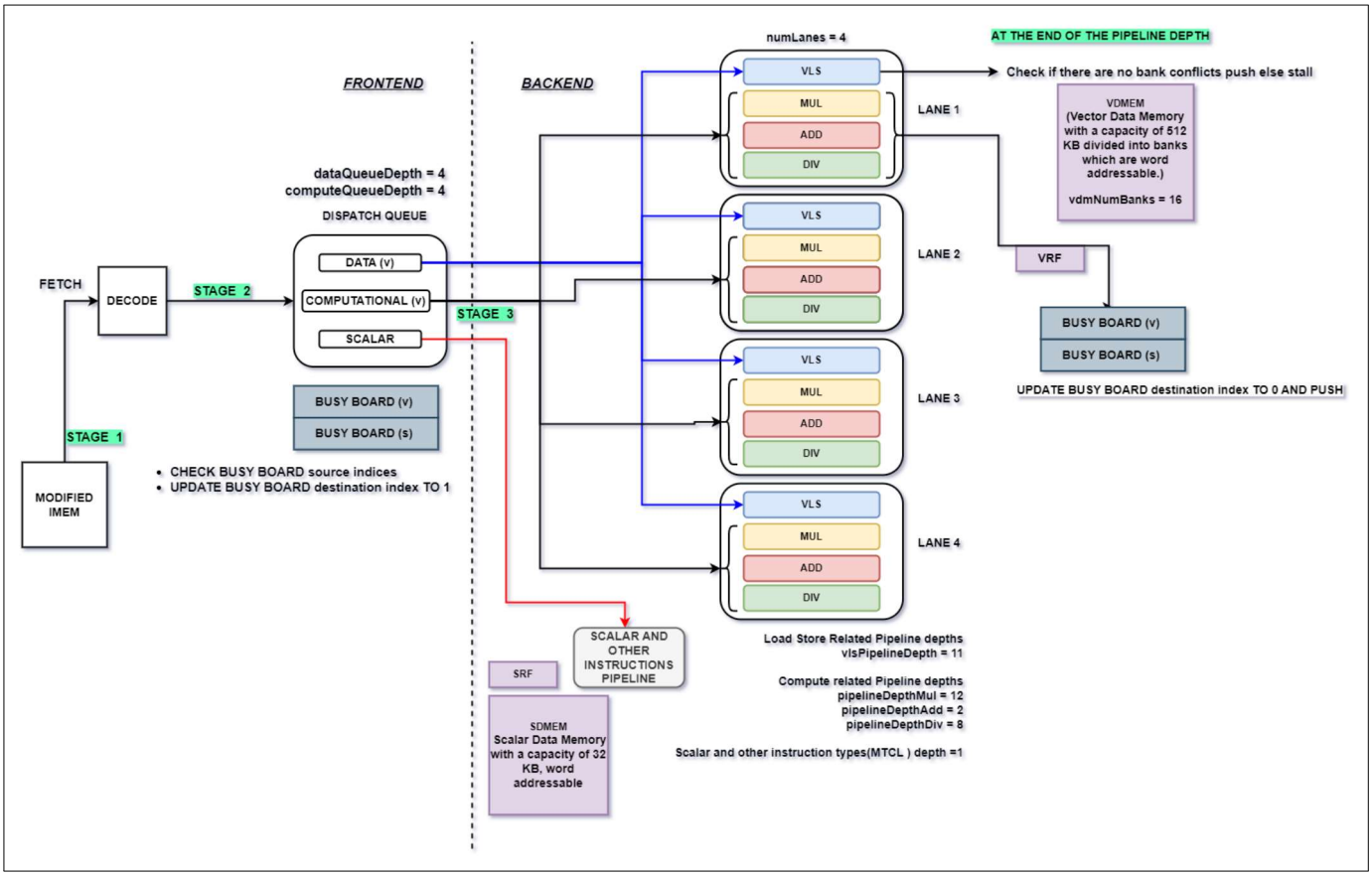


Fig. 1: Block diagram of the Timing Simulator

a vectorized operation. If the length of the vector instruction (vlr) is less than the number of lanes (l), then all elements get into each of the lanes, travel across pipeline depth, and get executed at the end. If the vlr is greater than the number of lanes, then the number of elements (m) that get into the pipeline per lane is: m/l + min(1,(m mod l)), where m is the length of the vector instruction and l is the number of lanes.

if *vlr less than l* **then**
 | All elements get into each lane
end
else
 | $numPerLane$ = m/l + min(1, m mod l);
end

Algorithm 2: DJ Formula

In case of Load store type instructions, Bank Conflict is the real problem. Bank conflicts occur when multiple load/store instructions try to access the same memory bank at the same time. In order to avoid such conflicts, the memory banks are divided into multiple groups, and each group is accessed by a specific set of load/store instructions.

When a load/store instruction reaches the end of the pipeline, the bank conflict check is performed. If a conflict is detected, the Bank Conflict flag is set to 1. If a conflict occurs, then only one instruction is allowed to proceed across the pipeline depth, and the rest are stalled for a specified bank busy time. This ensures that each memory bank is accessed by only one instruction at a time, and conflicts are avoided. The scalar instructions accesses memory from SDMEM (32 KB, word addressable) and Vector instructions accesses memory from VDMEM(512 KB divided into banks which are word addressable).

Jayanth formulated Jayanth's Lane Bank Theory for uniform strides. During the building and execution of our simulator, we made an interesting observation regarding memory access that pertains to instructions such as LVWS or SVWS, which have uniform strides and are spread across multiple banks. If the starting element has an address of x and a stride of x, then the subsequent element's address would be x+s, where s is the stride. In the event that these addresses conflict and map to the same bank, the equation X mod M = (X+S) mod M can be solved for S mod M = 0. This leads us to consider the nth element, whose address would be x + (n-1)*s, and its bank number would be (x + (n-1)*s) mod M, which is the same as (x mod m + (s Mod M … n-1 times)) mod m, or x mod m. This implies that the first and nth address are also in conflict, meaning that if any

two addresses conflict for a uniform stride, then all addresses will conflict. As a result, whenever there is a conflict between addresses in instructions with a uniform stride, the advantage of multiple lanes is nullified.

In case of Computational Instructions, Busy board checking takes care of the data hazard. When the instruction reaches end of the pipeline, the busy board destination register index is set to 0, indicating that particular register has no more dependency in the pipeline. After the busy board is set to 0, the instruction is deleted from dispatch queue. The scalar and vector registers access their values from SRF (Scalar Register file) and VRF(Vector Register File) respectively.

If all the queues and pipelines have no instructions and also the halt flag is already 1, then the timing simulator is terminated.

if *Instruction type is load/store* **then**
  if *Data queue is not empty* **then**
    1) Check for bank conflicts at pipe-end;
    2) Set pipe-end busy board register index to 0;
    3) Shift the pipeline;
    4) Delete the instruction from the queue;
  end
  else
    Perform steps 1, 2 and 3;
  end
end
else if *Instruction type is arithmetic* **then**
  if *Compute queue is not empty* **then**
    1) Set pipe-end busy board register index to 0;
    2) Shift the pipeline (multiply, add, divide);
    3) Delete the instruction from the queue;
  end
  else
    Perform steps 1 and 2;
  end
end
if *All queues and pipelines are 0 and the halt flag is 1* **then**
  Set the terminate flag to 1;
end

Algorithm 3: Pipelining Logic for Vector Instructions

III. Test Cases

We are running our timing simulator using default configuration parameters given in the Config.txt file. We changed one parameter and kept all others constant. We tested our timing simulator on the dot product and the Fully connected layer. We changed the values of each parameter in the range specified in Table 1.

A. Dot product

Dot product is written in asm code using VMIPS ISA given. We have generated Modified assembly code using functional simulator. To perform the dot product operation in Assembly language, the programmer needs to load the two vectors into memory and use a loop to iterate through each element of the vectors. Inside the loop, the corresponding elements of the vectors are multiplied using the multiply instruction and the result is accumulated using the add instruction. The loop continues until all elements of the vectors are processed. Fig 2 and 3 shows the results for different parameter changes.

Parameter	Default value	Variation range
dataQueueDepth	4	1-10
computeQueueDepth	4	1-10
vdmNumBanks	16	$1 - 2^4$
vlsPipelineDepth	11	5-15
numLanes	4	$1 - 2^4$
pipelineDepthMul	12	5-15
pipelineDepthAdd	2	1-14
pipelineDepthDiv	8	1-10

TABLE I: Experimental range of values

B. Fully Connected Layer

A fully connected layer is a key component of a neural network, where each neuron in the current layer is connected to every neuron in the previous layer. In assembly language, implementing a fully connected layer involves multiplying the input values with a set of weights and adding a bias value to each result, followed by the application of an activation function. The weights and biases are typically stored in memory, and the dot product is calculated. Once the dot product is calculated, it can be added to the bias term and passed through the activation function to obtain the output of the neuron. The process is repeated for each neuron in the layer, and the resulting values are passed to the next layer in the neural network for further processing. Writing an efficient fully connected layer in assembly language requires careful consideration of memory access patterns, instruction scheduling, and register allocation to minimize latency and maximize throughput. Fig 4 and 5 shows the results for different parameter changes.

IV. Observations

Lets us observe the change in number of cycles in both kernels by varying only one parameter and keeping all other parameters constant.

A. Dot Product

Fig 2 and 3 shows the plots for varied parameter and cycles for Dot Product test case.

- **Variation in Data Queue depth** : Increasing the data queue depth does not increase the parallelism of the load/store pipeline, since there is only one pipeline for load and store instructions in one lane. Therefore, the number of cycles required to complete the instructions remains the same regardless of the depth of the data queue
- **Variation in Compute Queue depth** : Increasing the compute queue length from 1 to 2 allows for more computational instructions to be queued up and executed in parallel across the ADD, MUL, and DIV pipelines.

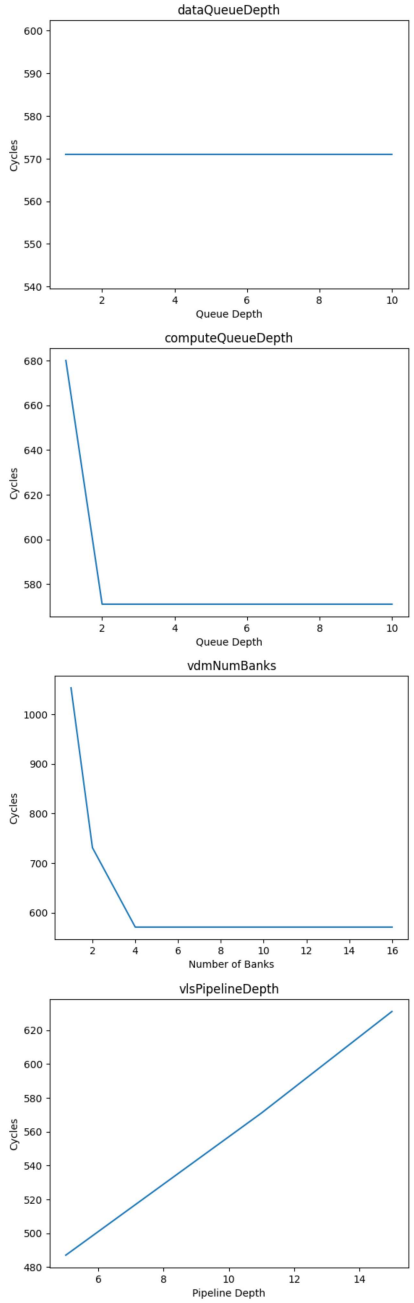


Fig. 2: Dot Product results

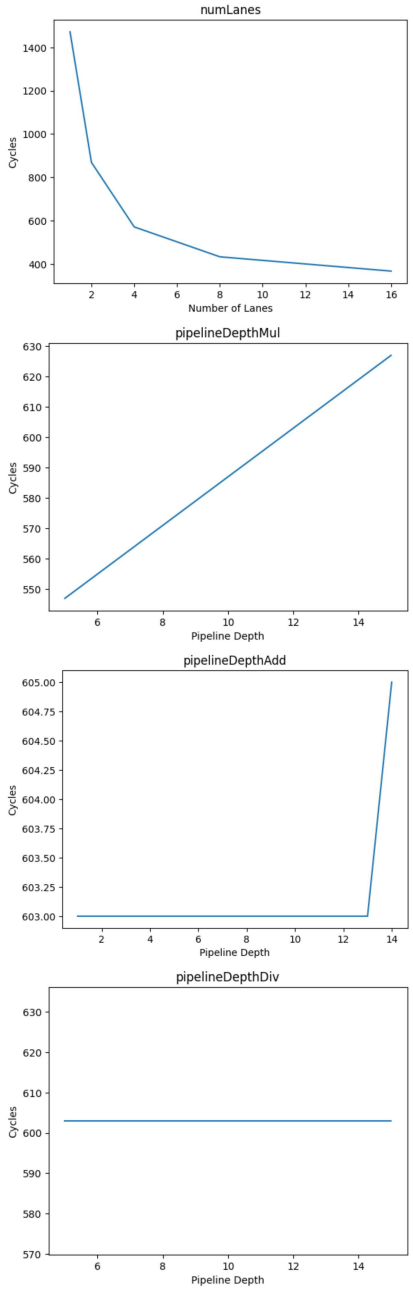


Fig. 3: Dot Product results

This increased parallelism leads to a reduction in the number of cycles needed to complete a given task, as more work can be done simultaneously. However on further increasing the compute queue length does not result in any additional performance gains this because there is no more parallelism left to exploit because we have only MULVV and ADVV instructions hence can execute at most 2 instructions in parallel.

- **Variation in Number of banks** : Increasing the number of banks can reduce the number of bank conflicts and enable smoother flow of instructions across the pipeline. However, after a certain point, increasing the number of banks will no longer have an effect on reducing the number of cycles as there are no longer any bank conflicts. In our example, when the number of banks increased to 4, there were no possibilities for a bank conflict, and therefore no further reduction in the number of cycles.
- **Variation in vlspipeline** : When we increase the VLS pipeline depth, it increases the startup penalty for vector load and store instructions. This is because deeper pipelines require more time to fill up with data, resulting in longer startup latencies. As a result, the number of

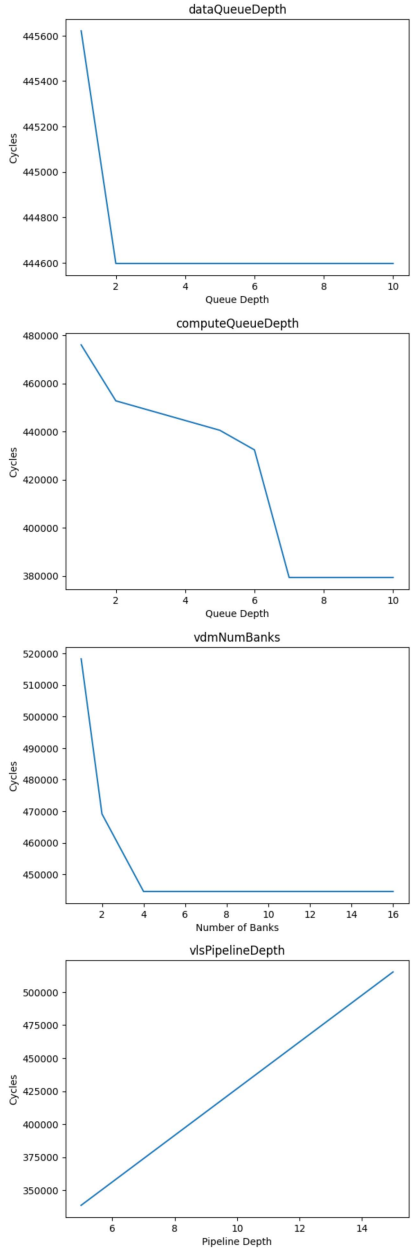


Fig. 4: Fully Connected layer results

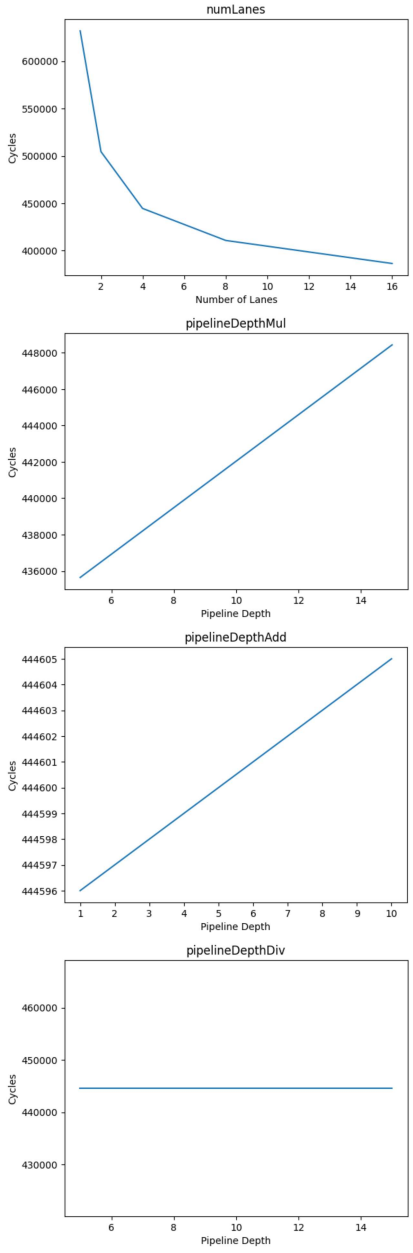


Fig. 5: Fully Connected layer results

cycles increases linearly as we increase the pipeline depth.

- **Variation in Number of lanes** : Increasing the number of lanes allows the processor to execute more instructions in parallel, which can result in a significant decrease in the number of cycles required to complete a given workload. This is because different instructions can be executed in different lanes simultaneously, without interfering with each other. So, increasing the number of lanes is an efficient way to exploit parallelism and improve the performance of a processor.
- **Variation in MUL pipeline** : When the pipeline depth for multiply instructions is increased, it takes longer for the first instruction to produce a valid output as it has to wait for previous instructions to complete their execution. This leads to an increase in the startup penalty for multiply instructions, which causes an increase in the number of cycles required to execute the program.
- **Variation in ADD pipeline** : Even though we increased the ADD pipeline depth, the number of cycles remained constant in our dot product implementation. This is because each ADDVV instruction is always accompanied by a MULVV instruction in the dot product computation. The default pipeline depth for MULVV instructions is 12. This means that until the pipeline depth for ADDVV instructions reaches 12 or above, increasing the pipeline

depth will not result in any change in the number of cycles.
- **Variation in DIVV pipeline** : There is no DIVV instruction in the dot product assembly code, the DIV pipeline size won't affect the number of cycles.

B. Fully Connected Layer

Fig 4 and 5 shows the plots for varied parameter and cycles for Fully Connected Layer case.

- **Variation in Data Queue depth** : Increasing the data queue size from 1 to 2 helps to reduce the delay caused by the consecutive data instructions in the assembly code. However, since the maximum consecutive data instruction is only two in Fully Connected layer assembly code, increasing the queue size beyond 2 does not provide any further reduction in delay, and therefore the number of cycles remains constant.
- **Variation in Compute Queue depth** : Increasing the compute queue length allows for more computational instructions to be queued up and executed in parallel across the ADD, MUL, and DIV pipelines. This increased parallelism leads to a reduction in the number of cycles
- **Variation in Number of banks** : Increasing the number of banks can reduce the number of bank conflicts and enable smoother flow of instructions across the pipeline. Increasing the number of banks beyond the point where bank conflicts are eliminated will not result in any additional performance gains.
- **Variation in vlspipeline** : Increasing the VLS pipeline depth leads to longer startup penalties for vector load and store instructions, resulting in increased number of cycles needed to complete a given task. This is because deeper pipelines take more time to fill up with data, which increases the latency for these instructions to start executing. Therefore, increasing the VLS pipeline depth may not always result in performance gains and should be balanced against other factors such as instruction parallelism and resource utilization.
- **Variation in Number of lanes** : We see a drastic decrease in number of cycles. It shows it is an efficient way of exploring parallelism.
- **Variation in MUL pipeline** : Increasing the pipeline depth for multiply instructions leads to longer startup penalties for the MULVV instructions, which in turn causes an increase in the number of cycles required to execute the program.
- **Variation in ADD pipeline** : Increasing the pipeline depth for multiply instructions leads to longer startup penalties for the ADDVV instructions, which in turn causes an increase in the number of cycles required to execute the program.
- **Variation in DIVV pipeline** : There is no DIVV instruction in the Fully Connected assembly code, the DIV pipeline size won't affect the number of cycles.

V. Optimization

We have implemented the concept of chaining for optimizing our timing simulator. We have applied this concept only to computational instructions. Chaining optimizes performance by not treating a vector register as a single register rather it treats it a group of small er registers. Does an instruction that shares a vector register with the preceding instruction doesnt have to wait for the entire instruction to complete to begin its execution instead it can began its execution after the first element of the previous instruction is executed. The concept of chaining is a bit similar to forwarding MIPS processor. Fig 6 and 7 show results for chaining in case of Dot Product and Fig 8 and 9 show results for chaining in the case of Fully Connected Layer.

Acknowledgment

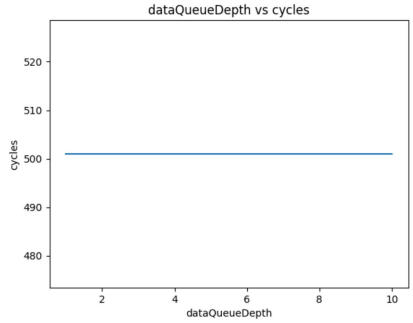
We would like to express our heartfelt thanks to Professor Brandon Reagen and our Course Assistant Swarnashri for guiding us through this course. Without their support and guidance, We would have been lost in the pipeline. And a special thanks to our laptops for not crashing during the final project presentation. And last but not least, we would like to thank Google for being the knight in shining armor whenever we needed answers to the most obscure questions. Thank you all for making this course an unforgettable experience!

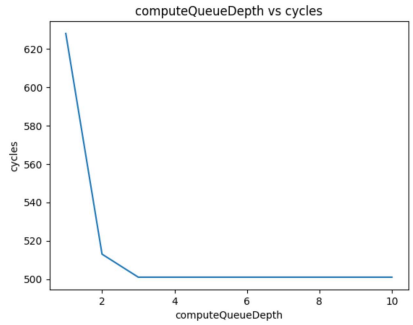
References

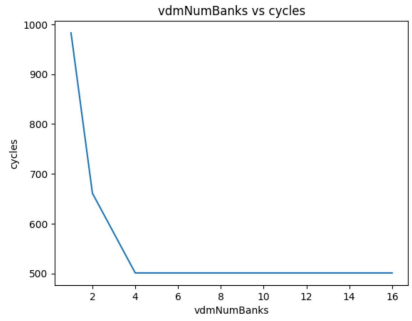
[1] K. Asanovic, *Vector microprocessors*. University of California, Berkeley, 1998.

Github Repository

https://github.com/D-girl-11t/VMIPS-Simulator







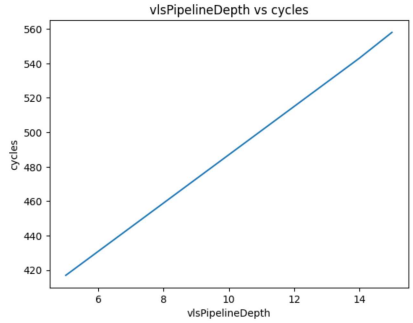
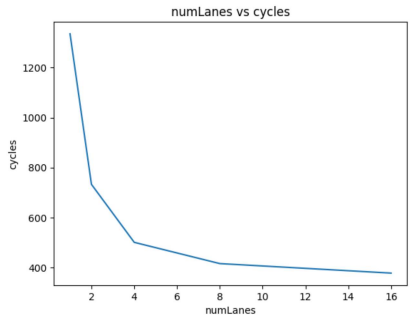
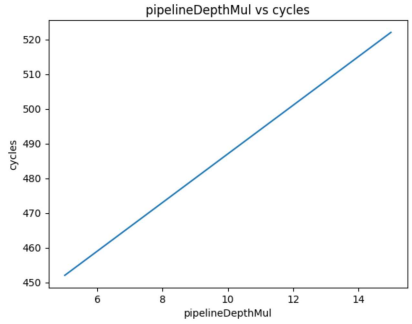
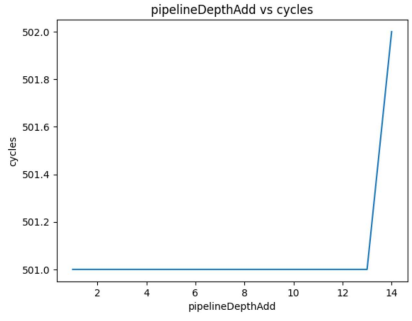


Fig. 6: Dot product results using Chaining







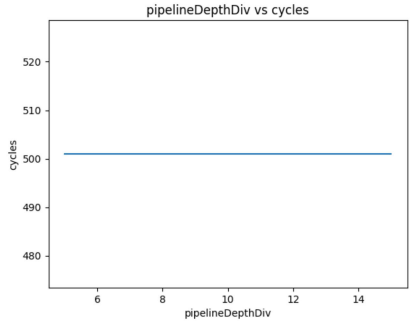
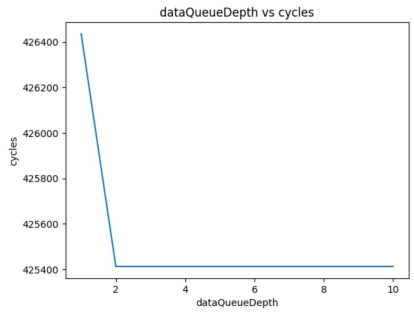
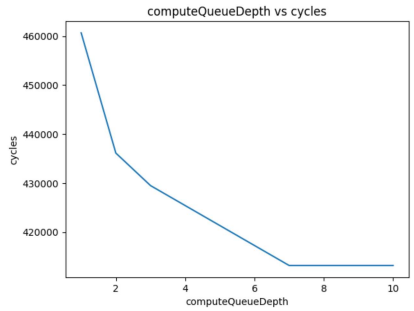
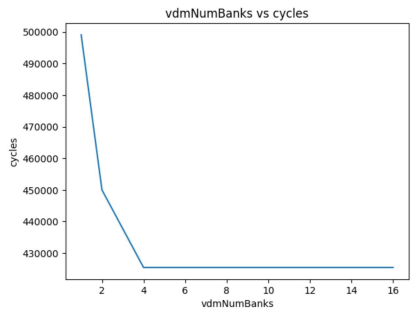


Fig. 7: Dot product results using Chaining







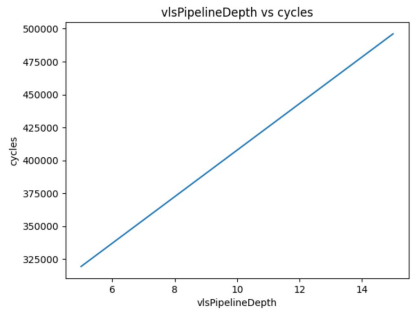
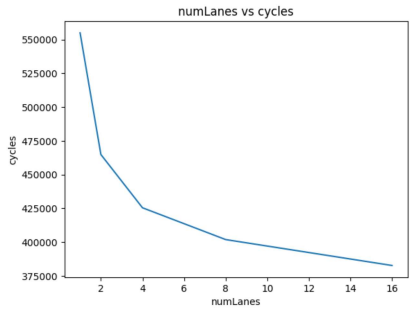
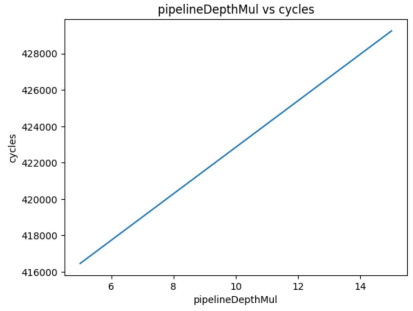
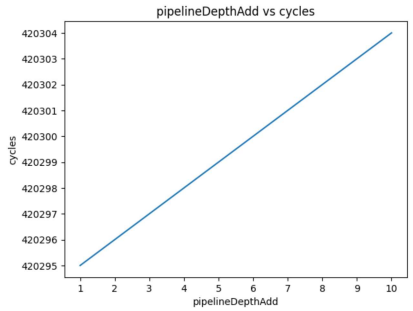


Fig. 8: Fully Connected layer results using Chaining







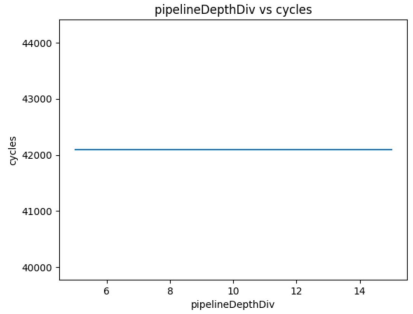


Fig. 9: Fully Connected layer results using Chaining