RISCY ML

Armando Moncada
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
moncadaa@mit.edu

Lee Morgan
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
leeban@mit.edu

***Abstract*—The proposed project is a pipelined RISC-V System-On-Chip composed of two main modules: a RISC-V processor that supports RV32I instructions, and a machine learning module, which contains several processing units to quickly perform common machine learning computations. The goal is to provide a complete SoC that can provide clear performance differences in ML computing against a standard RISC-V processor.**

***Index Terms*—Digital Systems, Processor, Pipelined, Matrix, Field-Programmable Gate Array (FPGA)**

I. Introduction

This system has two main components:

- A 4-Stage RISC-V Processor capable of executing one instruction per cycle. Any types of hazards are handled accordingly and sacrifice performance to ensure correctness of the program. The processor can accept a file containing a program to be run, and when the processor gets compiled it can run the program provided in that file.
- A Matrix-Multiplier module that can perform a multiplication of matrices of generic size. This module uses a shared memory space with the processor, allowing for programs to dynamically create matrices that uses the module.

The two components work seamlessly together via an AXI-like interface, and can be accessed via a few custom assembly instructions. The RISC-V GNU Compiler Toolchain was also utilized to write programs in C and compile them into binaries that the processor can read.

Source code can be found here.

II. Processor

A. Current Implementation

At this point, we have implemented a 4-stage pipelined processor capable of performing instructions in the RISC-V RV32I dataset. To ensure proper encoding and execution of the instructions, we have referenced the official published RISC-V ISA manual [1]. The four stages included are the fetch, decode, execute, and writeback stages.

a) Fetch stage: The Fetch stage contains the instruction memory BRAM module and its interface. This stage is in charge of keeping track of a program's current place in memory, and making requests to the instruction memory. The BRAM that is utilized is currently 32 bits wide, and can hold 128 instructions. When mapping memory addresses to memory addresses, the bottom two bits are ignored and then searched based on the remaining bits. This is in place to keep consistent with the byte offset requirements set forth in the RISC-V ISA. For example, if the current PC that needs to be fetched is 0x1C, the fetch module will remove the bottom two bits and send a request for the instruction at address 0x7.

The BRAM used in this module have a clock latency of two cycles. Without any jumps in memory, this does not produce any problems. The pipeline will just lag by 2 cycles but will still process an instruction every cycle. However, we found this limitation to be quite problematic whenever branches are taken. However, once a branch misprediction takes place, all instructions that are currently being processed in the BRAM must be annuled. We can a accomplish this by setting a flag for exactly 3 cycles before proceeding with the requested branch pc. For any cycles where that flag is set, we will simply ignore those outputs. Once the flag is reset, we can proceed with the pc corresponding with the branch. This does cause a loss of 3 cycles, so more work is needed for mitigating this issue. Loops will be quite detrimental to performance on this processor.

b) Decode stage: The Decode stage is where 32-bit words are processed to prepare for execution. This is especially important for detecting hazards within the pipeline, where a source register matches a destination register of an instruction in the execute or Writeback stage. If this does occur, we need to ensure the processor does not move the decoded instruction to the Execute stage, and that the Fetch stage does not send a new instruction to the Decode stage, which would overwrite the current instruction that initiated the stall. Signals will be sent downstream to execute to not process the next instruction, and sent upstream to Fetch to maintain the same instruction at its output.

c) Execute stage: The Execute stage has two main roles: processing decoded instructions or sending requests to external modules. Processing decoded instructions will vary based on the type of instruction, but fundamentally they are all taking 2 parameters as inputs and producing a single output. The two parameters could be two numbers to be added together, or two numbers to compute bitwise XOR. Once the result is produced, it is moved along to the Writeback stage where it can be written back to the register file.

Sending requests is a more involved process, and opens the door to a new type of stall. To keep the system as modular as possible, any external modules will follow a similar process for making requests. Each external module, whether that be memory or Machine Learning modules (to be discussed in detail in section III), will be required to provide a "busy" and

”done” signal, so that the processor can know when it can make a new request. If the busy flag is low, the processor sends in all the required signals, and proceeds with the instruction to the Writeback stage. But if the busy flag is high, then the processor must stall. This means that Execute cannot pass its current instruction to the Writeback stage. Additionally, the Decode cannot pass an instruction to Execute, and Fetch cannot pass an instruction to Decode. All stages will be frozen in this manner until the external module is no longer busy and the processor can send the request.

d) Writeback stage: The Writeback stage is the last stage for the processor before an instruction is fully processed. If the current instruction must write to a destination register, then we must send in the request to the register file to write to that register, using the data provided from execute. This works without issue for all instructions except for LOAD type, where it uses an external module and it also writes to a register. Here, we must wait for the memory module’s ”done” flag to let the processor know that the LOAD data is ready to be retrieved and saved into a register. If the done flag is not set, then we must not make any write requests to the register file, and additionally let all upstream stages know to not move any requests to the next stage. In other words, Execute should not move the instruction to Writeback, Decode should not move to Execute, and Fetch should not move to Decode.

B. C Compilation

To make uses cases more realistic, we looked into finding a way to compile C programs into binary that the processor can read. We learned that the RISC-V GNU Compiler Toolchain is currently the standard for creating such binaries. The translation from the compiled object files to our processor was not straightforward, however.

The first main issue was getting the address spaces for the processor to compile code correctly. It seems that there were default starting addresses for the code, stack, and heap that did not correlate with our programs, and were often much too large, requiring more BRAM than what we had space for on the FPGA. We got around this by designing some entry assembly code in sw/entry.S and then utilizing a linker script in sw/link.ld to correctly map our address spaces and start with the entry program that eventually calls the main() of the compiled C program.

Another issue occurred while initializing the processor’s BRAM to contain the program’s code. The Vivado builder looks for these files in a specific format, and we could not get GNU’s objcopy tool to get it in the format that we specifically needed for the initialization. In the end, we decided to use a tool provided by SiFive called elf2hex that can help us get the string hexadecimal dump in a format that we could use. It still required some editing in order to manually remove some problematic metadata, but overall, it make the translation from compiled object files to hexadecimal much more bearable.

C. Memory-Mapped Input/Output (MMIO)

A powerful abstraction commonly used in embedded systems is MMIO. This allows users to perform special functions that interact with hardware through their typical load and store functions. Our primary use case of this was printing data or characters to the console.

When a program attempts to store to address 0x40000000, an address well out of the range of the program’s address space, the processor instead uses the store data and sends it, along with a few flags, through the USB UART bus. A corresponding Manta program senses these data writes by looking out for those flags, and then prints to the terminal that is running the python program.

While this abstraction was quite useful, we quickly ran into issues with timing. After sending a packet via UART, the python overhead with receiving the data, printing, and responding back to the processor was so large that the processor would be done with the C program before the python program could even write a single character. To fix this, we made this MMIO call a blocking call, where we wait for the the python program to give a full confirmation that it indeed received data and is done printing to the console. This resulted in a functionally correct interface, but also one that was horribly slow. This often costed us millons of clock cyles that would not have been there normally.

D. Further Discussion and Areas for Improvement

There are definitely improvements that we can make to the processor to improve CPI. Our biggest improvement will likely come from more intelligent caching. To preserve functionality, we ended up just fetching an instruction every 3 cycles, just how the BRAM can provide. This helped make our CPI much more consistent, even in the face of many for loops.

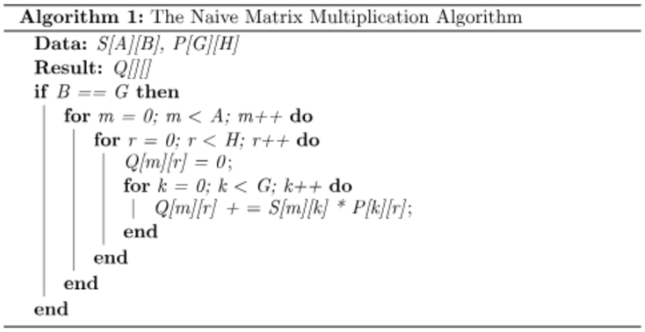
We would also consider expanding the MMIO to utilize more of the hardware on the FPGA. We could have easily made abstractions to map the LEDs, GPIO, switches, and other peripherals on the board. Another consideration would be to develop an I2C or SPI bus that was abstracted away by MMIO. Finding a different way to print to a console without leading to millions of wasted cycles would also be a crucial next step.

We are also soon looking into a superscalar design to help improve performance. We define superscalar as a processor capable of handling multiple instructions per cycle. Initially, we had anticipated this to be a difficult task, and one that would be part of our reach goals. However, there are simple ways we can use multiple processor pipelines to improve performance. One primary method we are considering is utilizing a secondary pipeline that takes branches, to avoid the huge cost in cycles every time a branch is mispredicted. If we have two pipelines, one where the branch is taken and another where it is not, we can proceed with instructions seamlessly by simply choosing the correct pipeline when a branch instruction is executed.

III. Matrix Multiplication

A. Top Level View

We have implemented a matrix multiplication module (MMM) that can perform multiplications on matrices of variable sizes. Because we wanted to be able to do small, more accurate matrix multiplications, and larger, quantized multiplications, we decided to give it the capability to treat matrix elements as 32 bit values, or 8 bit values, both in reading and writing. If the write_8 _bits signal is 1, then it will write the output elements to memory as four 8 bit values, stored together in one 32 bit value, since each address in memory corresponds to 32 bits. If it is zero, then each value will be stored with 32 bits. If read_8 its signal is 1, then it will read each 32 bit value coming from memory as four 8 bit values. If it is zero then it will be read as one 32 bit value. The multiplication begins when the 1 cycle start _mult signal is received. Along with that comes the addresses and dimensions of each matrix. Also, the multiplier and shift values are sent to the MMM too. These values are used for quantizing results from 32 bits to 8 bits when we try to write with 8 bit values.



```
Algorithm 1: The Naive Matrix Multiplication Algorithm
Data: S[A][B], P[G][H]
Result: Q[][]
if B == G then
    for m = 0; m < A; m++ do
        for r = 0; r < H; r++ do
            Q[m][r] = 0;
            for k = 0; k < G; k++ do
                Q[m][r] + = S[m][k] * P[k][r];
            end
        end
    end
end
```

Fig. 1. The Naive Matrix Multiplication Algorithm Used

The MMM sends back a busy and one-cycle done signal to the caller, along with the dimensions and address of the output matrix. It sends other values to the memory management unit (MMU), including the read _enabled and write _enabled signals, which signal to the MMU if we need a read or write to be completed. It also sends in indices which correspond to values that we want to grab from the right matrix and left matrix, which are calculated inside the MMM. The MMU sends a busy and one-cycle done signal to the MMM, along with the left _val and right _val that were attained from memory reads. Lastly, it outputs the write _floor value to the MMM, which signifies the lowest unoccupied index in memory that has all indexes above unoccupied. The MMU works by checking if it should read or write, waiting for the read or write to complete, then returning to the WAITING_FOR _OPERATION state. It also interfaces with the BRAM in order to make reads and writes. The MMU and MMM also take in clock and reset signals. A top level matrix multiplication setup can be viewed on the last page.

The operation of the MMM (when reading and writing 32 bits) starts when it is in the IDLE state. If it is in this state, and the start multiplication signal has risen, then the module starts following the algorithm in Figure 5 below [2], and going to the RUNNING state, where the iteration and setup for reads are handled. Once a read is necessary and possible, it goes into the WAITING_FOR_READ state, where it gets S[m,k] and P[k,r] through calling the MMU to read both values simultaneously. This works because we use a dual port BRAM. Also, the matrices are flattened and stored in memory in row-major order. Once the reads are complete, they are multiplied and added to the total. Once k==G, however, we stop reading and instead go to the WAIT_FOR_WRITE stage, where we wait for the MMU to write the total value to memory such that the output matrix is also stored in row-major order starting at region_floor. Then the total is reset and we go back to the running state and continue this algorithm until we reach m==A, meaning that all rows have been read. This is the end of the multiplication, and the done signal is set to 1 for one cycle, the busy signal goes to zero, and we return to the idle state.

The operation is similar when reading 8 bits, except that we now grab four values from each matrix at a time. Also, the right matrices are stored in column-major order, so we can grab multiple contiguous column values at a time. We also quantize and store our output values four times before concatenating them and writing to memory. To be able to have the 32 bit and 8 bit multiplication in the same module, we shifted down the values of the indices we used by two when reading/writing 8 bit values instead of 32 bit values.

B. *Performance Evaluation*

In order to test the performance of the MMM, we set up multiple testbenches, running it with matrices of varying lengths. We first did a quick multiplication between two 4x4 matrices, reading the values as 32 bits or 8 bits. The 32 bit implementation took 4730 ns and the 8 bit implementation took 1370 ns. For 8x8 matrices, we see 9 microseconds for the 8 bit implementation and 34 microseconds. We can see that the 8 bit implementation runs about four times faster than the 32 bit implementation. Upon further inspection, one can notice $O(n^3)$ time complexity, as a doubling in length led to a 8x increase in time. Clearly, the 8 bit alternative performs faster but at the cost of resolution of the data. The performance was tested in hardware with the rest of the system.

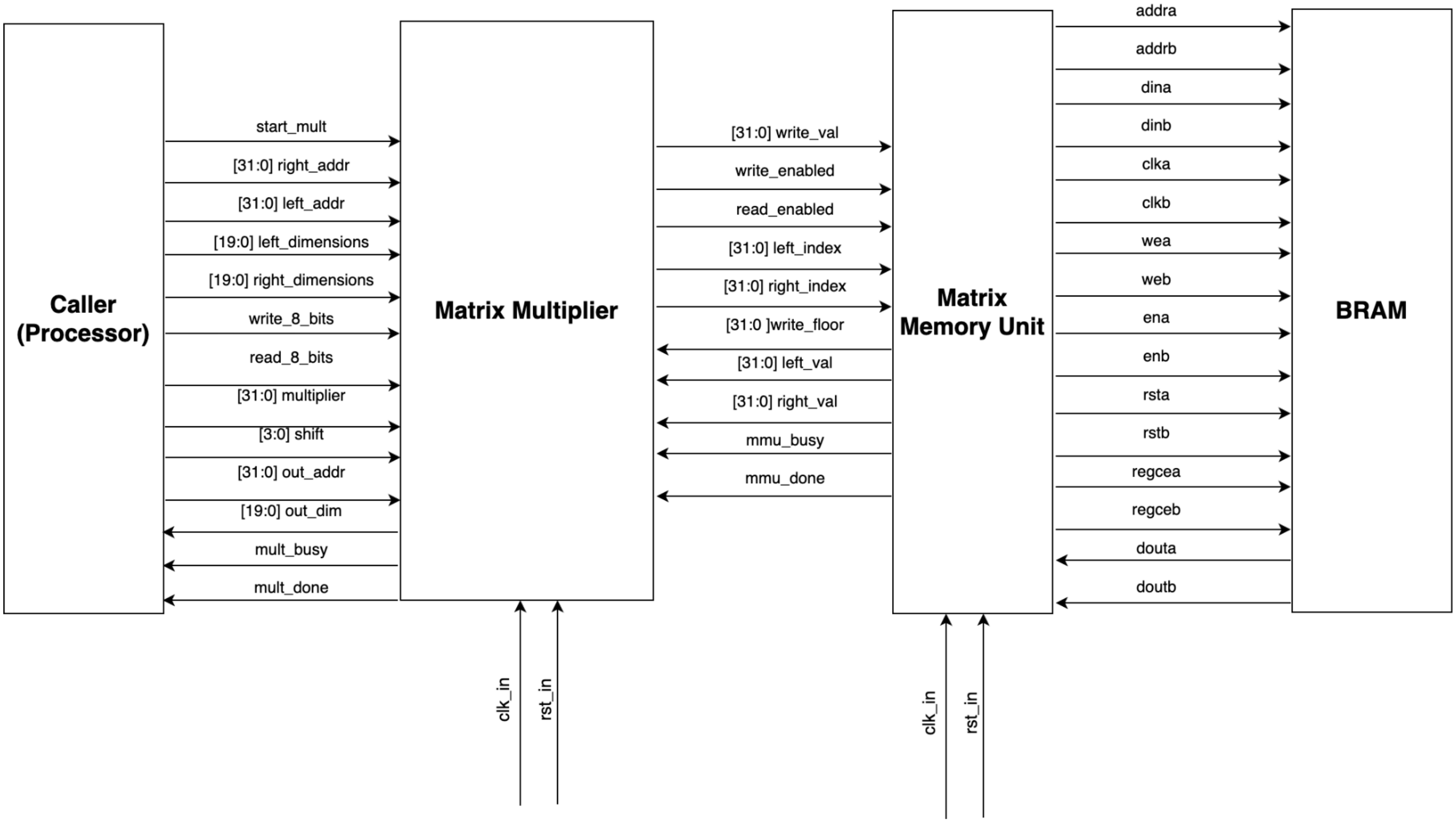
C. *Further Discussion and Areas for Improvement*

In the age of information and internet of things, it has become increasingly important to have fast running computations on resource constrained devices. Through this project, we have created a dynamic processor which can run assembly code, including code that we had compiled from C. This allows us to write C code that runs on an FPGA, with the addition of a custom matrix multiplication function. This processor can be used to complete a variety of tasks and has shown to be much faster than doing a traditional for loop within the C program. For further work, we plan on increasing the performance of the matrix multiplication unit through further optimization, and eventually adding in the capability to perform small machine learning prediction tasks, like classifying images of the MNIST dataset.

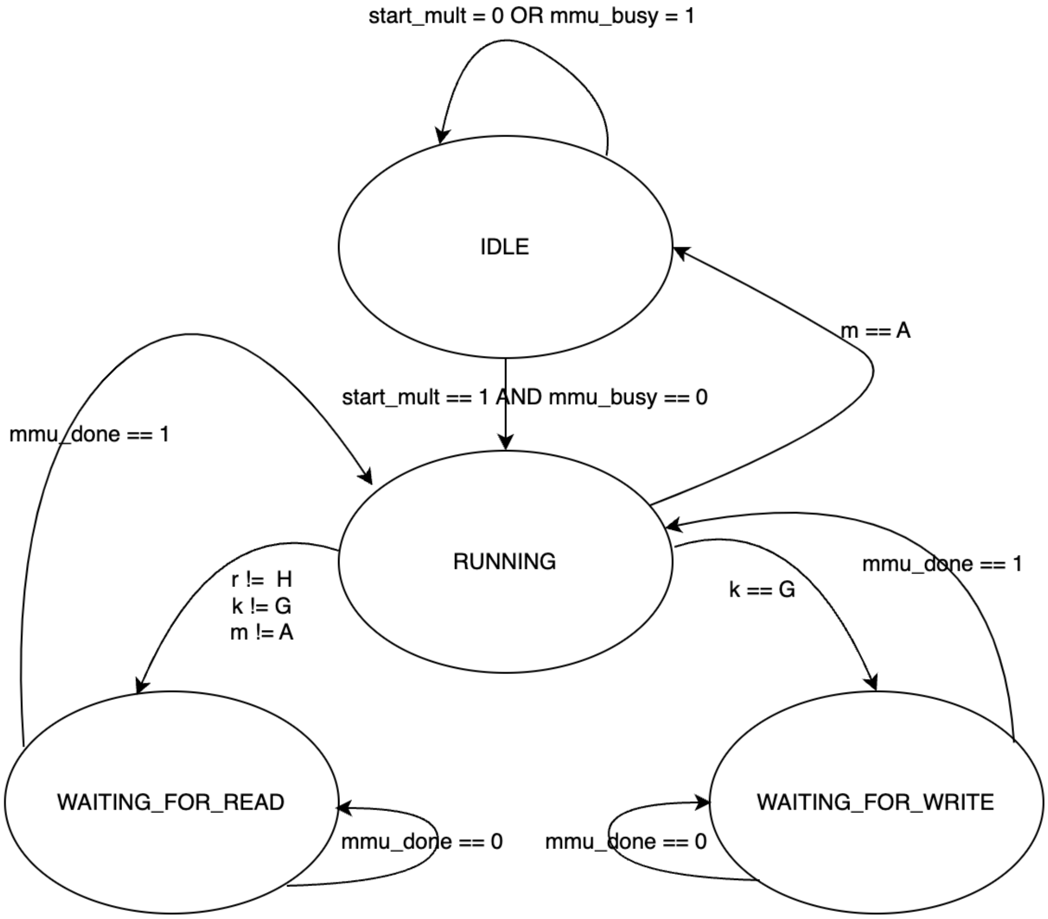
References

[1] A. Waterman, et al. ”The RISC-V Instruction Set Manual,” SiFive Inc., https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf (accessed Nov. 13, 2023).

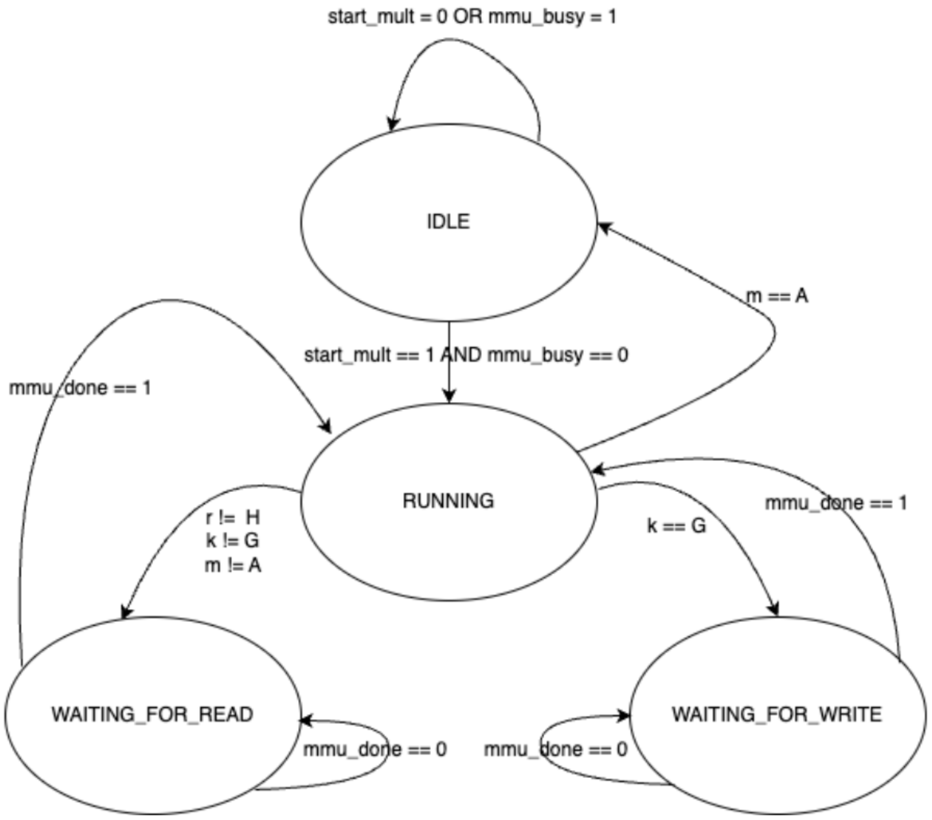
[2] S. Datta, “Matrix multiplication algorithm time complexity,” Baeldung on Computer Science, https://www.baeldung.com/cs/matrix-multiplication-algorithms (accessed Nov. 15, 2023).



Extra Fig 1: Top Level View of Matrix Multiplication Setup



Extra Fig 2: State Diagram for Matrix Multiplication



Extra Fig 3: Memory Management Unit State Diagram