Program layout for a small matrix calculator
Utils for matrix parsing
Memory layout for matrix objects
ASM 6

Jaime Meyer Beilis Michel

April 2025

1 Introduction

This is an implementation document of matrices utilizing row-major form in MIPS assembly. All code and this document can officially be found at:
https://github.com/cloudbongee/MIPS-matrix-big-dude.git

Special thank you to my professor Diana Diazh for introducing me to the amazing world of computer organization. And the support she's given me, specially when I broke my hand mid semester. I greatly appreciate this semester.

2 Functionality

This calculator works on a one time run per input, with no syntax for function definitions.
It permits the declarition of matrix objects and to operate with these accordingly.

Please note that while I am very fond of this project, and I hope it is noticeable, some of the mathematics and implementations I have done are not nearly even half of their best. I had to cut a lot of corners in order to make things work out properly in time to submit this assignment.

3 Assembling and linking

Since this is programmed in MIPS, the assembling process requires cross assemble tools that will permit linking adequately for the linux architecture.
I have provided a file that will automatically make sure the required packages

are installed utilizing **apt-get**, then compiles all files given in the repository.

Manual assembling

```
$ mips-linux-gnu-as matrix.asm -o matrix.o
$ # <assemble all other files>
$ mips-linux-gnu-ld matrix.o <all other files> -o matrix
```

Automatic assembling

```
$ # Give the proper file permissions
$ chmod +x assemble.sh
$ # execute
$ ./assemble.sh
```

4 Syntax

4.1 Example

The following is valid input syntax for a matrix:

```
// Inline declarition
Matrix M = [[1,2,3],[4,5,6],[7,8,9]];
```

As well as:

```
// Multiline declarition, tabs are ignored
Matrix N = [
    [1,0,0],
    [0,1,0]
    [0,0,1]
];
```

As it is noticeable, a valid statement is finished by a semicolon.
Finally, the following rules are true for well defined matrix operations.

```
// double slash is ignored as a comment

// Multiplication into a new space location
Matrix A = N * M;

// Multiplication into an existing address
M = N * M;
```

4.2 Fields

Given now a desired set of instructions, we can notice a common pattern, that is, we can divide our input into a tuple **(Instruction, location, value)** Where we can, at least for the purpose of a simple matrix calculator, make two types of instruction. As described below:

4.2.1 Input instruction

An input instruction will store the explicitly described input into the **location** specified field. We know that an instruction will be an input instruction when we declare "Matrix".

4.2.2 Operation instruction

An operation instruction will perform the defined operation as described by the input. Obviously, since we have three fields, it is not going to suffice to have only on input field, therefore, when passed an operation, it is only a matter of extending third field, value into a function representing the necessary instruction.

Thus far, I am planning on implement the following operations:

1. Sum, as inputted by $+$, with code 0

2. Multiplication, as inputted by $*$, with code 1

4.2.3 Processing

Since the type of instruction is determined by the first few elements of the command, one of the optimizations is not explicitly declaring any value, but instead doing the necessary branching once understood the type of instruction.

5 Utils

5.1 Hashing function

When implementing the symbol pool, we need to allocate uniquely but computationally fastly each of the strings into a memory location, therefore: I have implemented the DJB2 hash, appearing in the parsing function wherever the reading expects to have a symbol (that way there is no computational loss on sending the symbol directly to hashing). Secondarily, I am allocating a prime number of buckets on the hashtable to lessen the clustering.
Since this is kind of a primal implementation, I will rely on hardcoding a word table containing the differences between each prime number. This works for a limited amount of buckets, but in reality I doubt anyone outside of the machine learning community and partial differential equation enjoyers will ever require more than 10 matrices at a time. I really don't wish that for anybody.
Finally, I am doing a rehash every time the list is expanded.

5.2 Symbol pool

Implemented in the symbol_pool.asm file, the symbol pool is a hash map data structure utilized to track every declarition of a matrix given to the memory allocated for its value.

```
procedure InitializePool
    $v^* \leftarrow$ sbrk(84)                    ▷ Allocate 84 bytes
    for each byte $b$ in $v^*$ do                ▷ Cleanup
        $b \leftarrow 0$
    end for
    $v^*[0] \leftarrow$ functionTable
    return $v^*$                                 ▷ Return address
end procedure
```

5.3 File reader

The file reader proceeds utilizing the open file and read file syscall and flags that are usually utilized in MIPS. There is no further technical specification.

5.4 Automatic allocation

MIPS offers the `sbrk` allocation through the syscall `$v0 = 9`, however MIPS does not offer the `brk` operation which would deallocate the memory. Since matrices are usually memory heavy objects, but I also do not have the time to do systemwide implementations (nor the knowledge as far as I am aware). I will leave the afterthought of providing a memory pool to reutilize matrices as an alternate trash recollection.

When working with matrix operations redefining an already existing matrix, the way that memory is handled is by making a temporary function stack that simulates the original matrix's components. Then writes using the temporary stack matrix.