# MIPS

Sam Khosravi

Spring Term 2022

## Introduction

This report consists of four modules and a test. Each module is explained and parts of the code that are relevant to the explanation will be shown

## Test

To test the program I used these instructions that were given in the task. The store word instruction was not used here but will be explained in other modules. Load word does not take in an atom in this case but instead it takes in a offset immediate value directly.

```
def prgm() do
    [{:addi, 1, 0, 5},
    {:lw, 2, 0, 3},
    {:add, 4, 2, 1},
    {:addi, 5, 0, 1},
    {:label, :loop},
    {:sub, 4, 4, 5},
    {:out, 4},
    {:bne, 4, 0, :loop},
    {:halt}]
end
```

## Emulator module

This could be considered the "Main" of our program. The out, reg, code, mem and label are the parameters sent in to our two case clauses. We have one case that first goes through all instructions and extracts all labels and stores them in a list as tuples with the name of the label and its adress (pc). The other case will go through each instruction and do whatever the instruction does (if it is for example an addi then it will get the value on the register and add it with an immediate value and store it to the target

register) and then increment the pc counter everytime it calls on the run function again.

Out is set to a list of all outputs, and now we are instantiating it, which will result in us sending in an empty. list. Reg will contain our registers, and will to begin with be a tuple of 32 zeros. The code will be set equal to our prgm list, but the list will instead be made to a tuple in the assemble function in the Program module. This is so that we can use the built in elem/2 function to systematically walk through our instructions and return every instruction on each pc. Mem will have the same principles as reg, but will only be a list of tuples. Here, as we did before, we instansiate mem so we will to start with send in an empty list to "run". Last we have our label, which is implemented the same way as mem is. Lastly, we will call run with these parameter. The 0 will represent our pc (adress), which starts at zero.

```
def run(prgm) do
    out = Out.new()
    reg = Register.new()
    code = Program.assemble(prgm) #####
    mem = Register.memnew()
    label = Register.newlabelmem()
    run(0, code, reg, mem, out, label)
end
```

## Instructions being called in Emulator module

I feel like the biggest struggle was implementing one of the following: load word, store word and branch. Therefore I am picking one at random to explain how it is done. As we are discussing load word, I will continue to base the explanations of the following modules also on which of their functions are relevant to the load word instruction.

Load word is supposed to take a word from a memory adress and set it to a register, in this case to rt. To begin with we store the value of rd in a variable d. This value is added to the offset immediate value and will be considered our adress. We then use the memget function in our Register module to get the value on the memory adress (addr) we are looking for, and we set this value to a variable called loaded. We then write this value to our target register. Out will put the value in a list of outputs, and the pc counter will be incremented when we call the run function to get to the next instruction.

```
{:lw, rd, rt, offset} -> #rt = rd + offset,
    d = Register.read(reg, rd)
```

```
addr = d + offset
loaded = Register.memget(mem, addr)
reg = Register.write(reg, rt, loaded)
out = Out.put(out, loaded)
pc = pc + 1
run(pc, code, reg, mem, out, label)
```

## Register module

The functions explained here will be the ones we have used so far. This will be these: read/2, write/3, memget/2, new/0, memnew/0 and any helper functions that are necessary to speak about.

So to begin with we first used new/0 and memnew/0, but as these are trivial and explained earlier their code will not be shown. However, in the load word instruction we see that we first use read/2. This function will take in two parameters: The reg tuple which contains our 32 registers and their values, and then the register we want to read. The first read function will return 0 if we want to read at register 0, as register 0 will always be set to 0. The next read function will use the built in elem/2 function to find the value at the given index, i, and then return this value.

```
def read(_, 0) do 0 end
def read(reg, i) do elem(reg, i) end
```

The next function is memget/2. To explain this function, it is usefull to know that the memory is a list made of tuples, where each tuple has an adress and a value. Now to explain the code. The first memget function returns zero if our memory is empty and there is nothing to get. The second memget function will return the value x if the address we send in is at the beginning of our memory list. The last memget function will be called on if none of the above are executed. When we get to this function, we know that the first element of our memory list will not be the adress we are looking for, so what we do is that we recursively call on memget, but this time we remove the first element and send in the rest of the list (the tail). This picking apart of the list will be done until the adress we are looking for is at the head of our picked apart list, or if we pick apart the whole list we will return 0, as we then will know that whatever we are looking for does not exist in our memory.

```
def memget([],_) do 0 end
def memget([{addr,x}| t], addr) do x end
def memget([h|tail], addr) do memget(tail, addr) end
```

The next function to explain is the write/3 function. This function will take in our register tuple, the register we want to write to, and the value we want to write. To begin with, if we want to write to register 0, we will just return the reg tuple, as we can not write to register 0. The second write function will make use of the putelem/3 built in function to store a value val at an index i in our register tuple called reg.

```
def write(reg,0,_) do reg end
def write(reg, i, val) do put_elem(reg, i, val) end
```

## Program module

This module is very simple and is used to traverse through our instructions. No code will be explained as it is very trivial.

## Out module

This module is likewise the Program module very simple and the code is trivial. All this module is used for is to create a list of the outputs of our instructions so that we one, can debug the code easier and confirm that our code is correct, and two because the lab instructions wanted us to print the output.

## Output when running the code

This is the output given when i run the code in the terminal. The empty list is what is stored in memory, but as I did not use the store word instruction there is nothing in our memory. That is also why the second value in the list is a zero, as that is where we call the load word instruction, and if you recall from earlier we will return zero if the address we want to load from is not present in the memory. Also this output is returned as a tuple, even though it does not show up on latex.

[5, 0, 5, 1, 4, 4, 3, 3, 2, 2, 1, 1, 0, 0], []

## Discussion

This lab was very challenging. It took me 3-4 8-10 hour days to complete, and I felt like it was a big jump in complexity from the previous exercise. Now that I think about it I maybe should've discussed the store word instruction instead, as I struggled a bit more with that one. My problem there was that I did not really understand how the memory should be implemented. However, I discussed with Johan and when I asked if I should

use a list where I just keep on appending to he said that it was the correct way to implement the memory. One error with my memory implementation is that I don't overwrite memory addresses. The code still works as it will go from beginning to end of the list, so the new value at an address will still replace the old one in the way that we will come across the new value first and return it. However, this will take up unnecessary place in the memory. Other than this, I struggled a bit with the conditional clauses in the branch operation, but I came forward to how to implement them correctly pretty fast. The discussion forum on canvas and the "Övningar" were very helpful for me.