

Subject: Designing MIPS CPU Architecture with Python Programming Language

Student: Amin Hasanzadeh Moghadam

Student’s Number: 9632413

Teacher: dr.Mousavi

In the name of god

Basically, Mips cpu architecture contains 5 stages and each stage has the specific work to do. These stages called Instruction Fetch (IF) , Decoding in register file (ID) , Execution with ALU (EX) , Read , write or none of them from Memory (MEM) and Sometimes writing data to a register in Write Back stage (WB). This project has 3 files which are Instruction Memory, Register file and Data Memory.

And it has been designed with some modules which will be introduced too. Here in this article we will show each stage with and without pipelining.

This project has 3 files which are Instruction Memory, Register file and Data Memory.

Main module:

In this project we have a specific file which contains written instruction for our cpu. At the main module we have an array which is called **Address.** after reading data from instruction memory file we will initialize values in this array. we have another array which called **a**. This array is a 2D (5\*5) array because we have 5 stages and in each stage we need 5 values. Here we show address array:

**a = [ Instruction name , Stage Name , RegisterRs , RegisterRt , RegisterRd , Constant address],[…],[…],[…],[…]**

at the first time of running, our clock cycle and pc value is 0 and we will update them after each of the loops (while). Now we read the data from the instruction memory file and update the Address array with each instruction. Actually each of the members of this array contains an instruction. So now we know mips cpu works all the time so we use while (True) and this method continuous till we have done all of the instructions works and our cpu is empty!

We initialize every value in binary form so after updating pc we have to change the 32 bit value to decimal value because our language understands just decimal values for calling each member of arrays. For converting binary form to decimal for we have a module which is called **BinaryToDecimal**.



We know that if pc wants to be upgraded, it has to be added by 4. So after converting pc to decimal we will divide it to 4 because our address array members have been called one by one not 4 by 4. For pipelining simulation we use **(a)** array and after each state we will shift members of this array to right to understand and see each of instructions are in which stages.

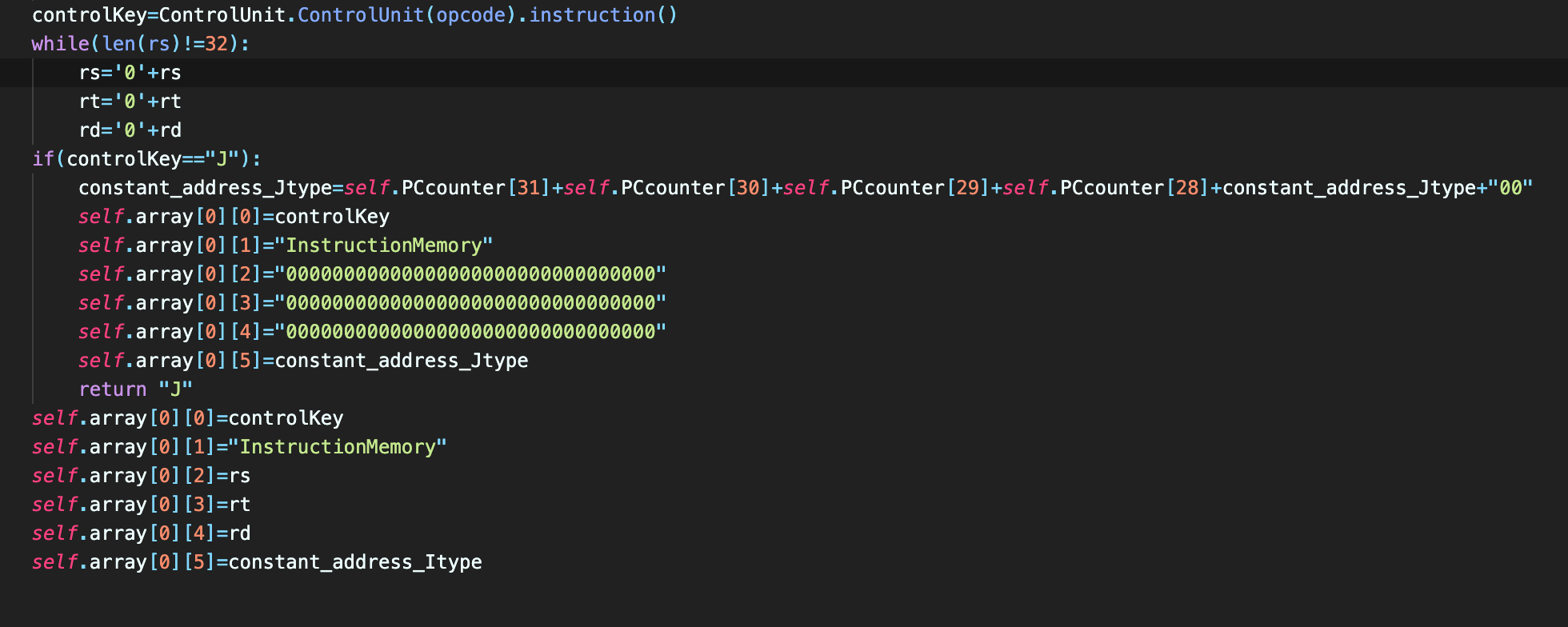
Our mips works until the end of the last instruction so before the last instruction we shouldn’t put any instruction into the **(a).** so we put something like nope in the array and for not happening any misunderstanding we named it **None.** Which is 32 bit none value. After all of these we call instruction memory module.

Instruction Memory (IF stage):



Our mpis has to do the last instruction in pipeline and comes back to the previous stage to do the previous instruction of the last instruction and it continuous until we arrive to main module and our loop updates then. So after initializing the values of this class we call the fetch instruction and after that we will call IFID register and we will see that at each stage we have done this work already. So when all of the next instructions have been done we will come back to this class to do things that our new instruction which is fetched needs. We will talk about 2 first ifs. So each instruction has some parts depends on its type. But in this project we don’t use the types. For example for any type we need (RegisterRs or RegisterRt or RegisterRd or constant Address for branch or constant address for jump)

So we initialize all of them and put them in some specific values.



Now we call control unit module for finding the name of the instruction.

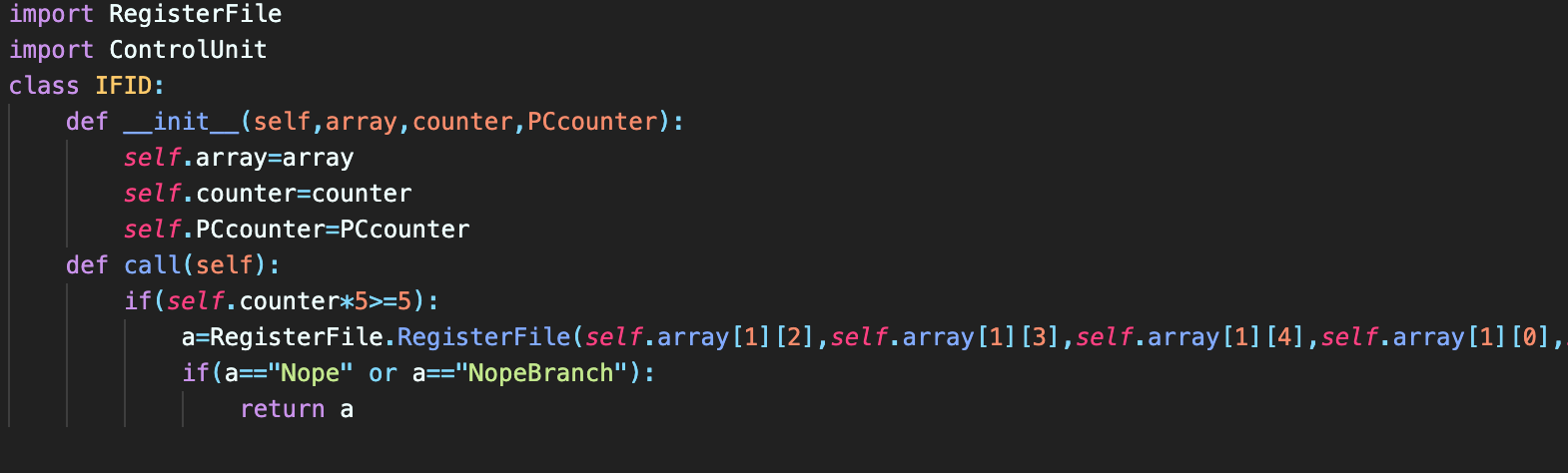


We designed instructions ourselves so the opcodes of the instructions are designed by us.

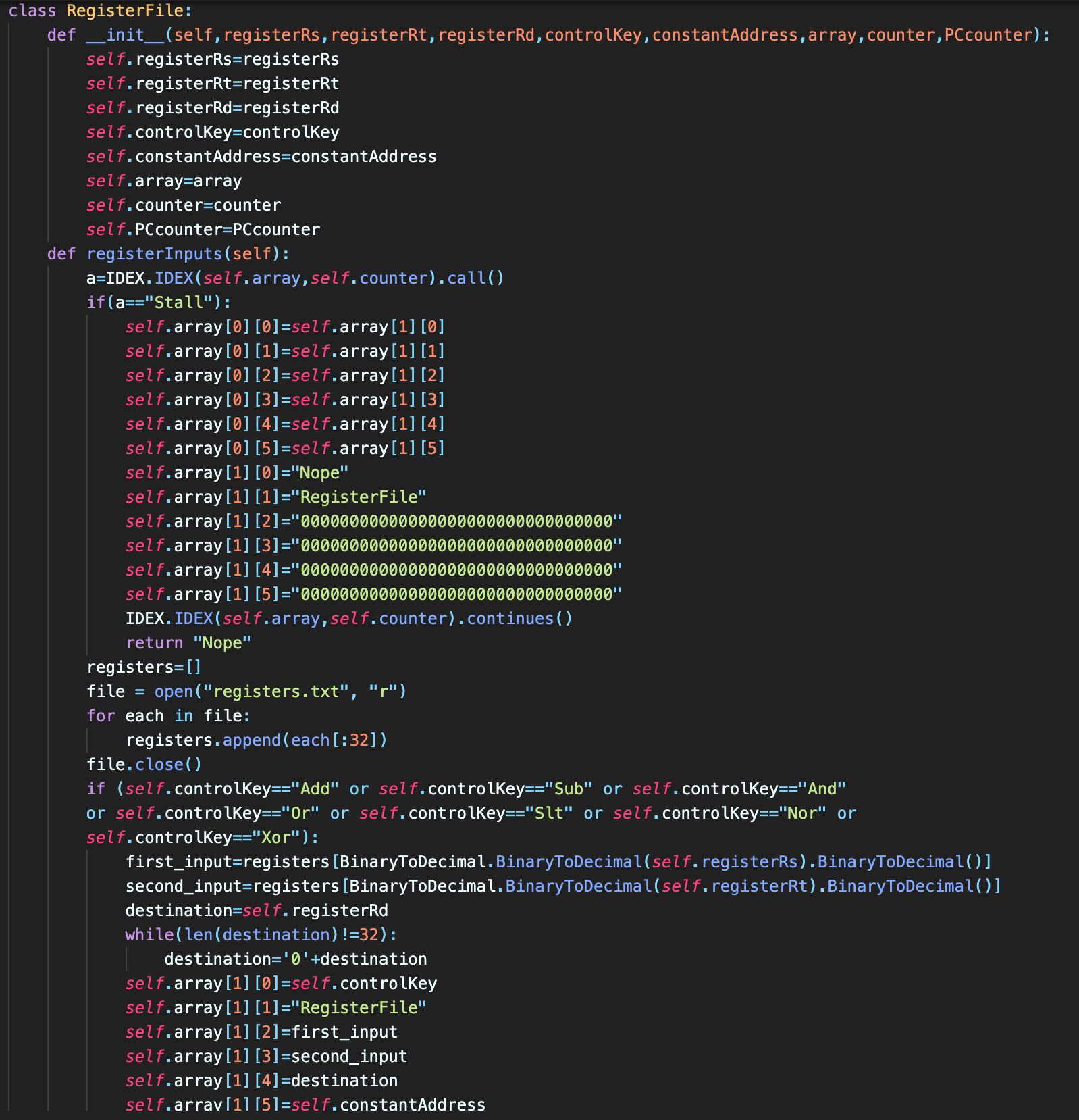
So we remove shift amount bits and function bits and we need just opcodes for understanding the instructions name because it is enough.

After that we will come back to instruction memory class. Jump instruction has been initialized in this class and has been passed to the main module to updating the value of pc.

After that we initialize the new values and put them to our array and come back to the main module. After shifting right in the main module our instruction is in the second value member of our array (which is **a**). so we will call instruction memory again and the we call IFID again.

IFID (register):

In this register we read data and the crucial data is array. in this register we decide to go to the next stage or not. If everything was ok , we will call register file class in RegisterFile module.

Register File (ID stage):

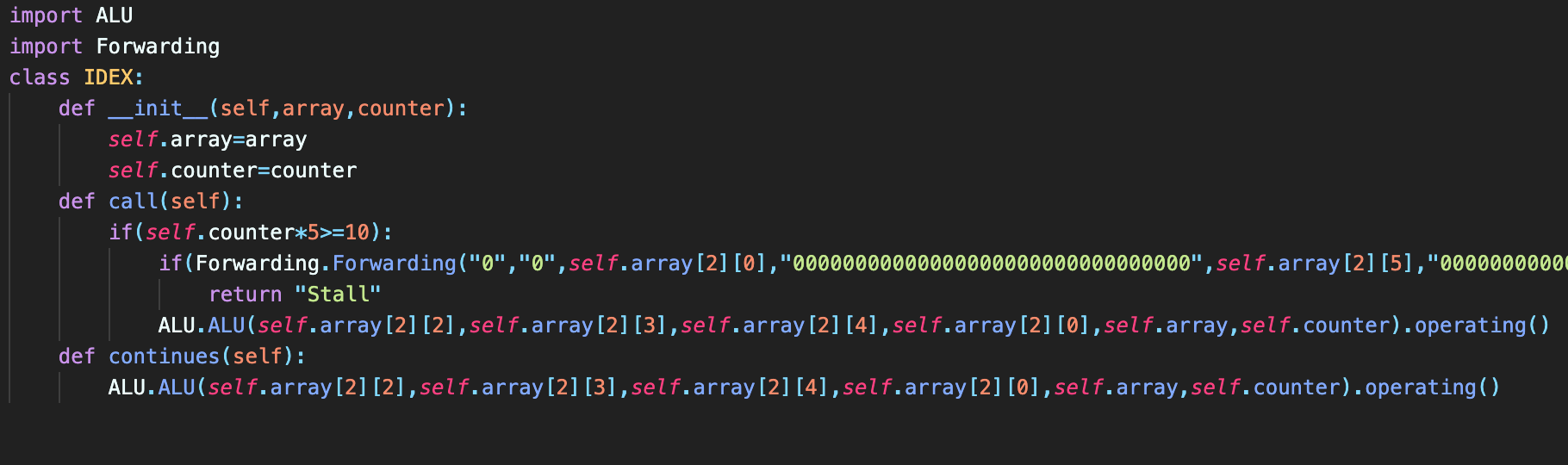
we will talk about the first if and Stall. After this if we read registers from register file which is actually a file and initialize the register array just like our address array in the main module. Then according to instruction fetch stage we know the first member of each members of our array is the instruction name and in IFID we passed it to this class with controlKey name.

now if our instruction is R-Type we will initialize registers (rs,rt,rd) which are called here by (first\_input , second\_input , destination) and initialize the values of our array memberes again.



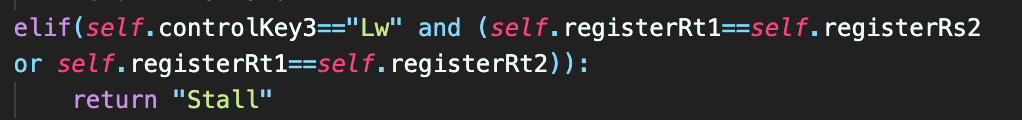
For example if our instruction is “Beq” , we will compare registers here and calculate the address of the target here (because of the hazard issues) and then we will save the address of the target in our array and return “NopeBranch” to tell the previous stages that there has to occur a nope after this branch. Because of that in instruction memory class we use an if to say this thing to this class and if nope occurs we mustn’t do anything and come back to the main module to update pc and continue our work.

In register file class we just initialize registers for the other instruction and if our instruction was jump or None or Nope we just have to go to the next stages without any changes.

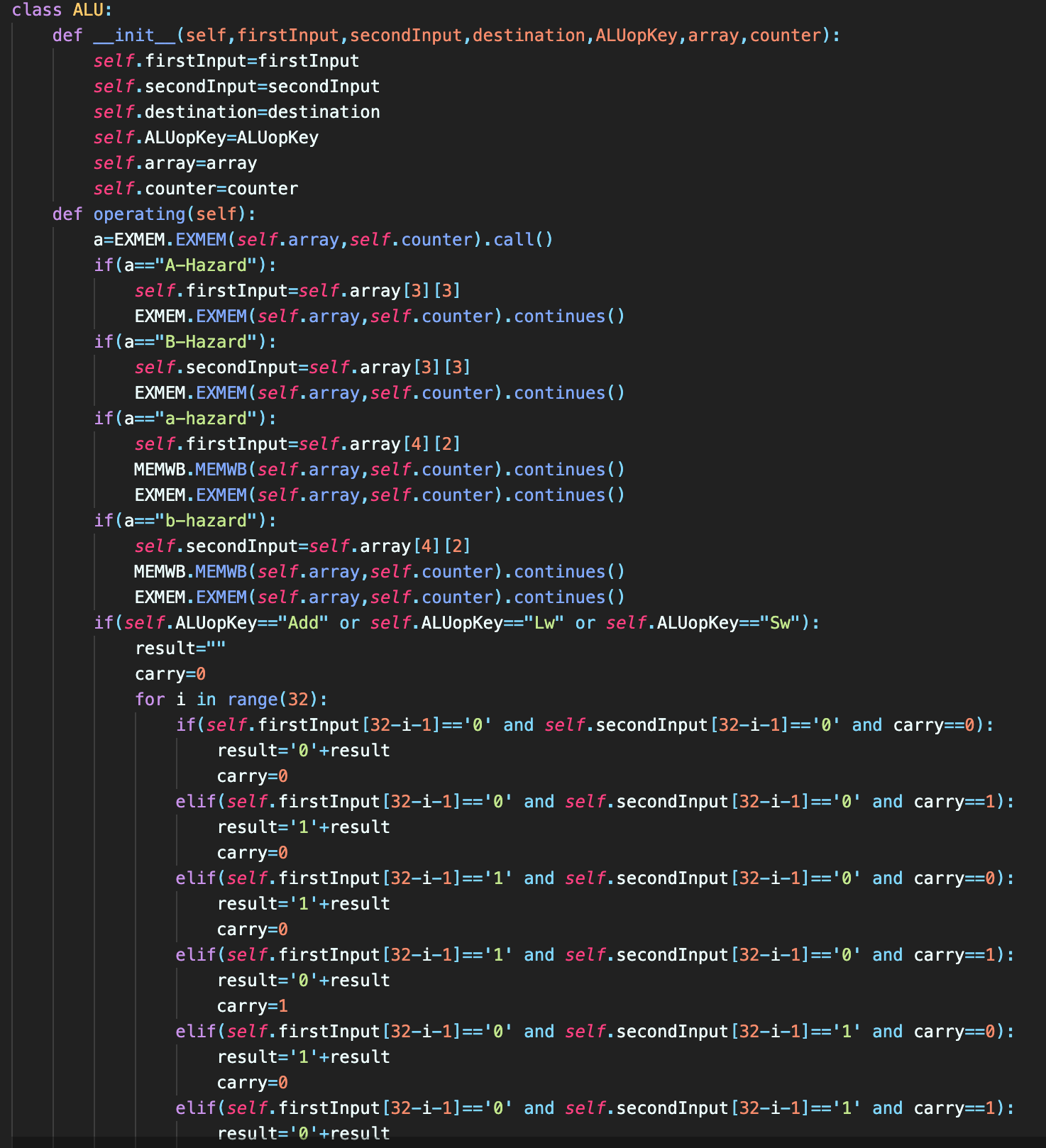
IDEX (register):

So like the previous stage we will call IDEX register at the beginning of register file class.

But here we will check any Stalls for “Lw” in forwarding class and if it was True we will return Stall String to tell the previous stages that the stall happened. If it wasn’t correct we continue the work and call ALU module. If stall happened we have to continue not stop anymore so when we insert nope for the stall we have to call ALU again to do instructions which are after “Lw” for example. So the continues function is for that.

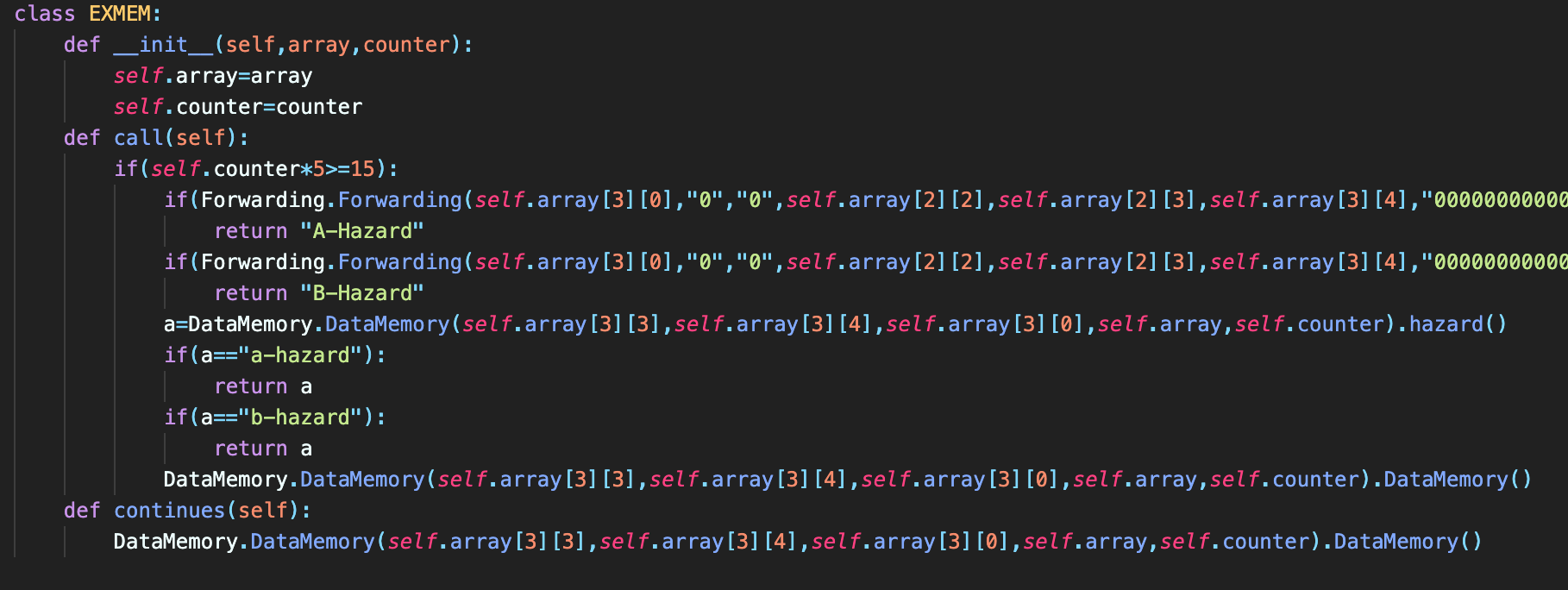


ALU (execution stage):



In this stage we calculate R-Type and I-Type operations (except branch which is done in register file stage). For example here we have add operation for add , sw and lw instructions with a full adder which is designed by us.

EXMEM (register):



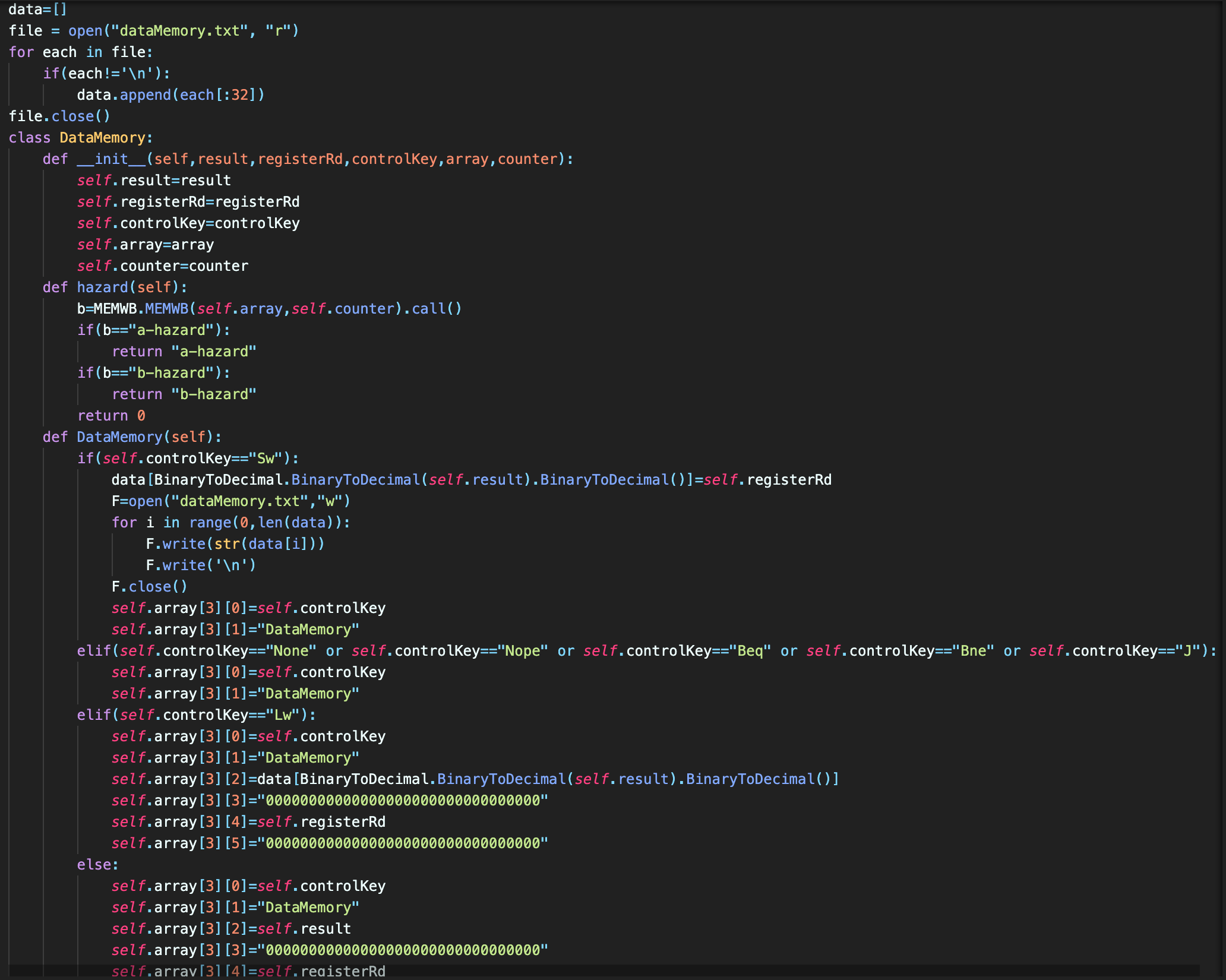
in this register we are looking for any forwarding and if it was true it will send the type of hazard to ALU stage to change the data of ALU inputs.

It is like IDEX register.



Data Memory (stage):

Firstly in this stage we read the data from data memory file and use them for “Lw” or write on this file for “Sw” or doing nothing for R-Type instructions , Nope , None , Branch and jump instructions. And we have response for any hazard like ALU stage and this response will be given by MEMWB register.

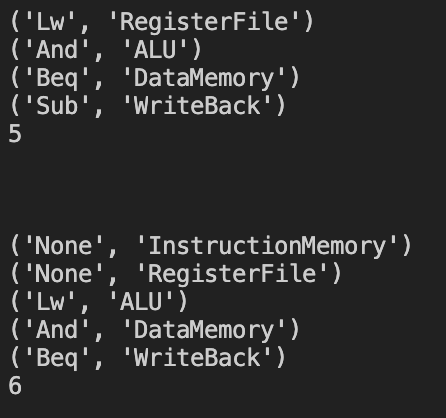
MEMWB (register):

It is same as the previous register.

Write Back (stage):

And finally we read the registers from register file and write the newest values there again.

And if our instructions weren’t R-Type or “Lw” we do nothing and continue the work.

And our output is like this.