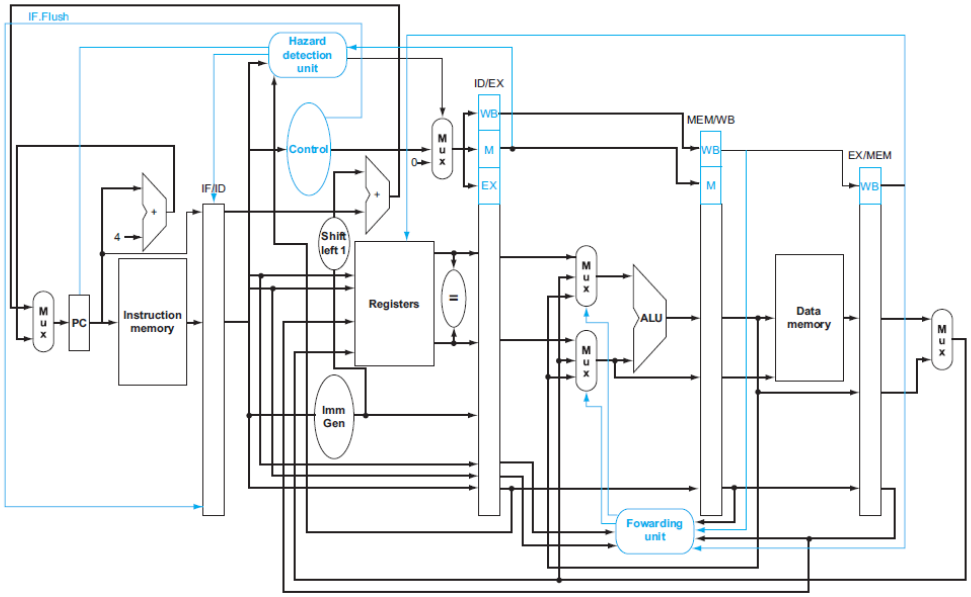
An outline of the design used to create the code



Dataflow for the program
(Image Source: CO and Design, Patterson and Hennessy)

Please note:

1. All the instructions are encoded in 32-bit format in `code.mc`, and the memory module is ***byte-addressable*** and ***Big Endian***. Therefore, to read and write to memory, different functions for word, byte etc. has been defined. After execution, all the data memory is written in the `memory.mc` file before the program terminates. (additional files `console.txt`, `datacache.txt`, `instructioncache.txt` and `register.mc` are used to facilitate the GUI operation)
2. Please install `bitstring module` and `PyQt5 module` in Python before running this code
3. Please read **README_phase3** file to know how to run the codes.

For Cache Implementation (newly designed in Phase 3):

`Class cache` has been defined to incorporate the cache implementation module. This class has functions `read_word(), read_Byte()` and `read_word(), read_Byte()` etc to read and write to and from the cache implemented. Two instances of this class called `Instruction_cache` and `Data_cache` have been made to show I$ and D$.

In both `class execute` and `class pipelineExecute`:
In `fetch()` function, the `Instruction_cache` instance is used to read from the I$, in order to fetch the instructions.
In `memAccess()` function, the `Data_cache` instance is used to read and write via the D$, for load and store instructions that access the memory.

For Pipelined/Non-Pipelined execution design (extended from Phase 2 design):

Two different classes have been created to deal with the pipelined and non-pipelined method of instruction execution.

The class that deals with non-pipelined execution is `class execute`

The class that deals with pipelined execution is `class pipelineExecute`. This class is a build-up on the `class execute` with added functionality and data members for improved data flow.

Both the classes use a five-stage setup to execute instruction namely: Fetch, Decode, Execute, Memory Access and Write Back.

Firstly, let us take a look at ***class execute***.

Some data members - *Register* (from `class register` – discussed later), a memory object named *Memory* (from `class memory` – discussed later), a stack pointer *sp,* an instruction register *instruction*, a pointer counter *PC*- are initialised values in the *__innit__()* function. The functions of these data members are the same as indicated by their names.

The `Run_program()` function takes input of the `code.mc` file to be opened which contains instructions and memory information. The function opens the file and stores all available information in the Memory object. The *run* function then starts execution by making a call to the *fetch* function till the time the next instruction is not available meaning till nextIR does not return zero.

In *fetch* the next available instruction is read from Memory, PC is incremented by four, and the number of cycles is incremented. This function then makes call to *decode* to move to next stage

The `Decode()` function splits the instruction into various bits and makes a call to the `checkFormat` function. The check format function returns the instruction type – R, I, S, SB, U or UJ. Accordingly, the `Decode` function makes a call to the concerned instruction type decode function to completely break down the instruction and retrieve the concerned register numbers, immediate values, operation types, etc. as the case may be. The concerned decoding function passes the operation information to `ALU`

The `ALU` then performs the data required operation and calls `memoryAccess`. memoryAccess operates only if the `memory_enable` data member is True in value. This is determined at the decode stage. The necessary operation is performed in accordance with the value of funct3. After this, `writeRegistor` is called. Just like `memory_enable`, `writeRegistor` does work only if `write_enable` is True, which is determined at the decode stage.

After completion of `writeRegistor` control returns to *run* which checks for next instruction and the process is repeated.

Coming to the ***class pipelineExecute*****,** the major difference between this class and *class execute* is data forwarding paths introduced in the decode and alu stage. Also, unlike *class execute* where the concerned functions repeatedly called the functions responsible for the next stage of execution, here, the calls are maintained by the `runPipeLine` *function*. This function is responsible for executing different stages of different instructions parallely. It takes into account whether the previous stage has successfully been run/completed by relying on data member *alu_run, decode_run, fetch_run* etc. and then proceeds to call the concerned function to proceed to the next stage. Data hazards in the code are identified by the use of `RDQueue` which holds information about destination registers currently under use.

`class register` and `class memory` create use dictionary objects to create the respective data structures. In `class memory`, data and instructions are stored in big-endian storage method. The registers store data in base 10.