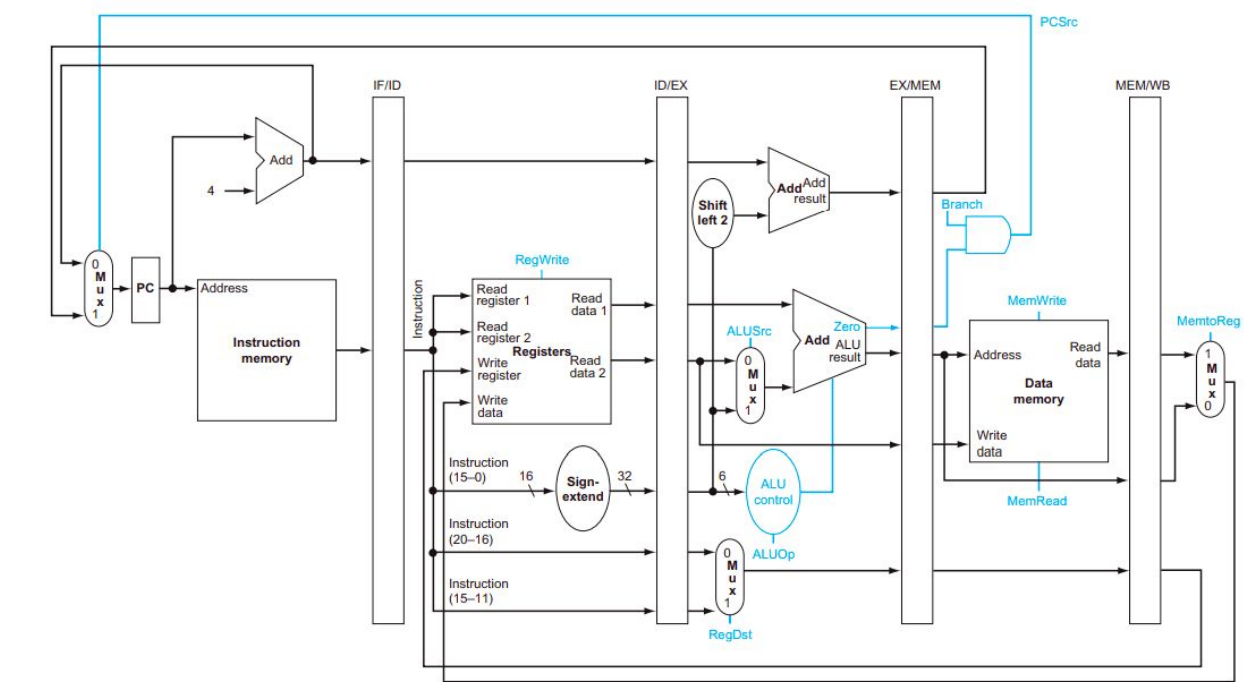
Simulation of a Pipelined MIPS Processor

A. Without forwarding

Aim: To simulate a MIPS pipelined processor without forwarding.

Architecture: We implement a 5-stage pipeline with the following steps:



- For simulation purposes, all pipeline registers were simulated like a **class(IFID, IDEX, EXMEM, MEMWB) with the program counter, the instruction and other important control values as data members**. The instruction is a long long and properties such as *rs, rt* and *rd,* are extracted from the instruction.
- For the entire simulation to end all instructions must be finished. **Hence, the instruction inside each of the pipeline registers should be NOP.**
- Assume two routines, writeReg(instr) and readReg(instr) which return the list of registers that an instruction will write and read from. Using this abstraction, we have the simple condition for a hazard between two conditions that is:

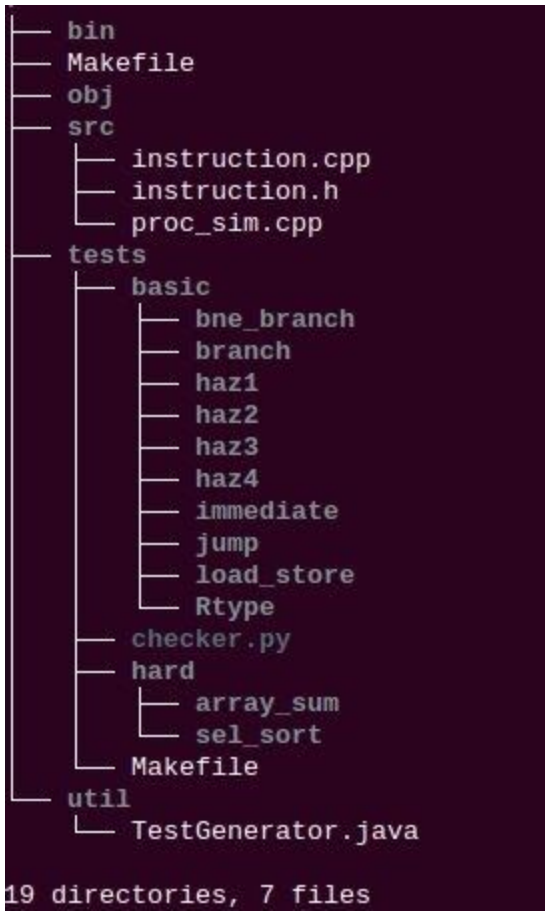
$$\exists\ r \mid r \in writeReg(i1)\ and\ r \in readReg(i2)$$

- We check for a **hazard between the instruction in the IFID register and the IDEX and EXMEM register.** In case of a hazard, a NOP is inserted in the **IDEX register** and the **updation of the PC is stopped.**

- **In case of a branch, we stop the pipeline until the branch condition reaches the end of EX stage.** This is implemented by pushing a NOP in the IFID register each time there is a branch instruction in the IFID or IDEX register.
- For jump instructions such as **j**, **jal,** and **jr,** we insert a **stall of one cycle** after the instruction has reached the ID stage and flush the IF register preceding it and replace it with a **NOP.** The PC is updated in the next cycle and execution continues.

Running the simulation:

The directory structure is as follows:



- The simulation is run by the proc_sim.cpp file. It takes the compiled instruction file and the memory files as input to produce the final output logs. The instruction.cpp file provides all the meta-data required from an instruction such as the type of the instruction, read-register set, write-register set, so on.
- TestGenerator.java takes a MIPS code file and writes the converted output(to binary) to a text file.
- The tests folder contains all the automated tests that any version must pass. To run the checker, the checker.py script is necessary.

To run the simulation, take the following steps:

1. Build the code using: “make”
2. To run the automated tests, run: “make test”

To run your own test case, do the following:

1. Add the text file containing the code into the bin folder
2. Convert the text file into binary by: java TestGenerator <file_name> <mips_name>
3. Now, you can run the simulation using: ./proc_sim <mips_name> <memory_file>

B. With forwarding

Aim: To implement forwarding in the processor implemented in part A.
Design:

- To check whether forwarding is to be done we use the following condition: If there is no stall, check whether value required in the EX stage is available in the EXMEM or MEMWB registers.
- There are cases when stalling is essential, these are:
    - When a hazardous instruction occurs after a load instruction. Since the load instruction cannot calculate it's result until the end of the MEM stage, we must insert a stall of one cycle.
    - Branches and jumps require a stall.

To run the simulation, take the following steps:

1. Build the code using: "make"
2. To run the automated tests, run: "make test".

To run your own test case, do the following:

3. Add the text file containing the code into the bin folder
4. Convert the text file into binary by: java TestGenerator <file_name> <mips_name>
5. Now, you can run the simulation using: ./proc_sim2 <mips_name> <memory_file>

C. With variable delays:

Aim: To simulate cache misses and hits using a probabilistic model.

Design:

- Whenever a load instruction has to read from the memory in the MEM stage, we generate a random number between 0 and 1. If the number is less than x, then we have a hit and we stall for N-1 cycles. In case the number is greater than x, we do not stall and continue execution as normal.

To run the simulation, take the following steps:

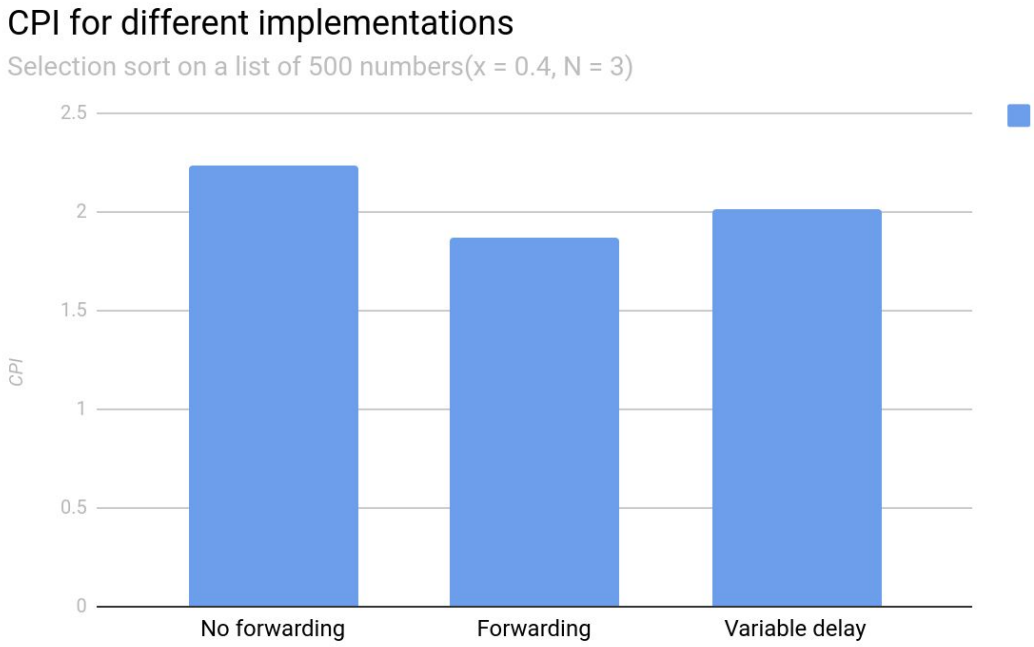
1. Build the code using: “make”
2. To run the automated tests, run: “make test”.

To run your own test case, do the following:

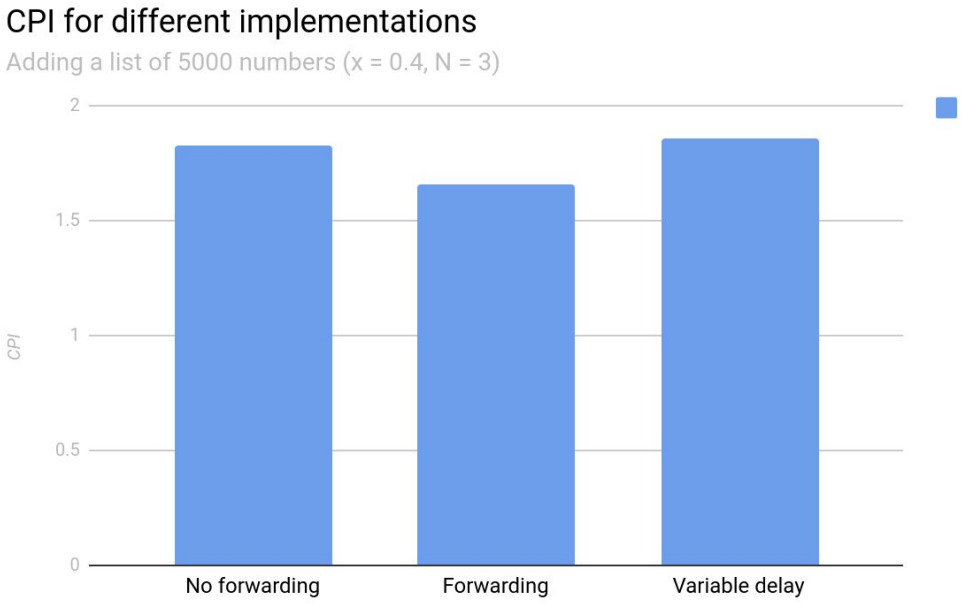
1. Add the text file containing the code into the bin folder
2. Convert the text file into binary by: java TestGenerator <file_name> <mips_name>
3. Now, you can run the simulation using: ./proc_sim2 <mips_name> <memory_file>

Comparative results:

For a program consisting of 18% load instructions, the result is as follows:



For a program consisting of 25% load instructions:



For different values of x and N, we have the plot of CPI looking something like this:

	N=3	N=10	N=33	N=100
x=0	2	3.16	7	18.16
x=0.2	1.93	2.87	5.92	14.91
x=0.4	1.87	2.56	4.84	11.39
x=0.6	1.79	2.27	3.74	8.2
x=0.8	1.73	1.98	2.79	4.98
x=1.0	1.66	1.66	1.66	1.66

The trend shows that the growth in the CPI is linear in both x and N.