Description of the processor

The Core was written in C++ which give us the ability to create a class to represent it and options for future utilization (upgrades and interfaces), in this class were define the methods for the Core prived ones are the internal functions of a processor and are not relevant for the rest of the system, the core class was divide as a normal nips pipe processor with the methods been (FETCH,DECODE, EXECUTE, MEMORY and WRITE\_BACK) trying to emulate as close as possible a Mips. The public methods are the way the core interacts with the rest of the system, in our case they are “CoreClock(), Reset() and report()”. The data types used are the one provides by the API “\_regFile and pipeStageState”

Internal build:

fetch() : this method reads from the memory and fill an pipeStageState structure to pass it to the decode() method

Id(): this method sets pipeStageState.srcVal from the register for the next the execute part of the core

Exe(): this method reads the command and depending of it will do the mathematical operation corresponding then stores the result in the result register

Mem(): depending of the command in this part of the pipe, the method writes or reads to from the memory and pass the value. If necessary it can also just pass the value that receive (example with the command add)

Wb (): writes the desirable value to the destiny register

All of these steps are clock independent and they don’t need to run in an specific order in this way we can simulate the parallel proprieties of a core (modules of a processor are run at the same time and in the same cycle, each of module is independent of the rest until the next clock)

To emulate the synchronic proprieties of the core we use CoreClock() method, this method does what a real core does on a normal cycle(based on the Mips core learned in class) “runs all the stages of the pipe and afterwards update all the register setting ready all the information for the next cycle”

Here(CoreClock() ) we set the Hazzard detection unit(HDU), the forward unit(FU) and the SplitRegister option, depending of the flag (the mode the core is working) the CoreClock() will choose a different mode

Without flags: “yonathan write something small”

With Forwarding:

With SplitRegister

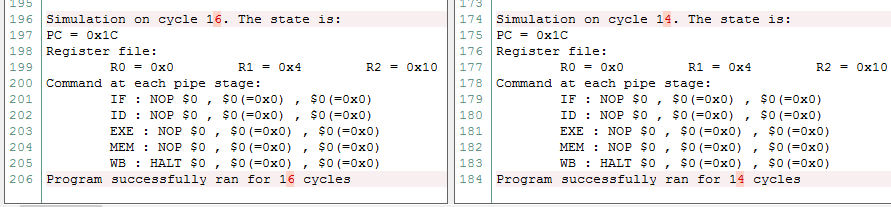
The nop\_stage() method saves the information from changes on the specific stage need it and the Reset() method does what the name says “resets the core”

Difference on test

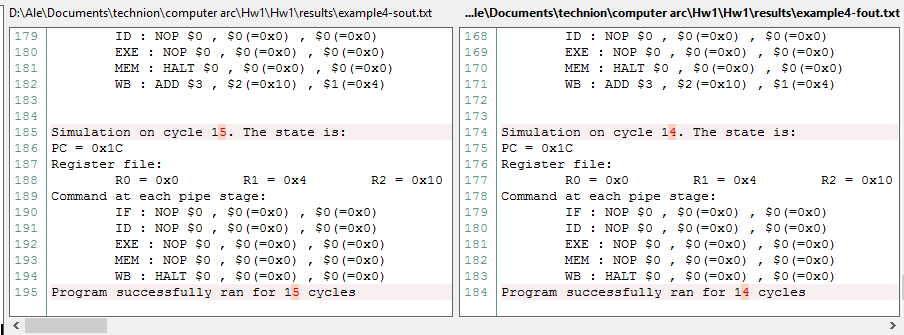
We can see in all the test that the most efficient way to run our processor is with the -f command which always finish our set of instructions in less\* cycles (the results are obtain after running the compile program on the virtual computer provide by the curse)

\*this is a processor simulator for that reason we use cycles need it to complete a set of instructions

Without Forwarding (Left) vs Forwarding -f (Right)



Split register(Left) vs Only forwarding(Right)



In this example, we can see the improvements got on example4

We did the same analysis for all the test provide and also for another test\* to get a better idea of the improvement



In conclusion after running some test we can see that the margin of improvement its relative to the set of instructions running but it always fallows the same trend :

Test\*

