Robert Carff
Computer Architecture
October 30, 2019

The Mips_Pipeliner is essentially a two-pass scheduler. I liked this approach because when only operating with one entity of each hardware resource, we only need to check two things. If the specific hardware is free, and if the registers need to be stalled for.

The next instruction can not be executed until the required hardware resource is free. To keep track of this, we can keep a global variable for each resource that will be its "next available time". On the first pass of each instruction we store the current resource tick count, and then add the "completion time" of that instruction. This cascades across each instruction, allowing us to see their "soft" execution times. "Soft" meaning that we have not yet checked if the data being executed on is free. This turns out to be the C.1 diagram in our textbook.

The second pass is used to check of the two registers being operated on require a variable that is still being calculated. In our project, for simplification, we assumed that each 4 or 5 cycle instructions would lock a register - and force stalls on the ensuing instructions. If this be the case, we simply check how long we must stall for, and add cascade that many cycles down the pipeline until the end.

I added MIPS instructions from the back of the book as well as all four of your example program (mipscode1-4.txt). Being that we reduced each instruction to either two cycle, four cycle, or five cycle operation, I created a list that contains each type of instruction. When the instruction is checked in the second pass, it is classified as one of the three types and then stalled for appropriately.

I decided to implement my scheduler in java. The program takes in a file similar to the examples shown in class. I read in data and created two, two-by-two matrices. Matrix one stores the actual instruction and its required registers. This is incase we want to create a more aggressive approach and check each of the registers in use. Matrix two holds my formatted output. Each line contains an instruction, along with its resource-variable calculated time.

I provided makefile to compile and clean the directory. It is executed by piping in the .txt file.
*** Because of the way I parsed each line, THERE CAN NOT BE COMMENTS, a pound sign (#), ON THE SAME LINE AS AN INSTRUCTION. This will cause my readFile() function to classify that line as an instruction.

EX:

```
        li      $2,23           # this will break the line
        sw      $2,12($fp)
        movz    $31,$31,$0
```

```
li      $2,29           # need to remove these comments for li to work
sw      $2,16($fp)
lw      $3,12($fp)
```