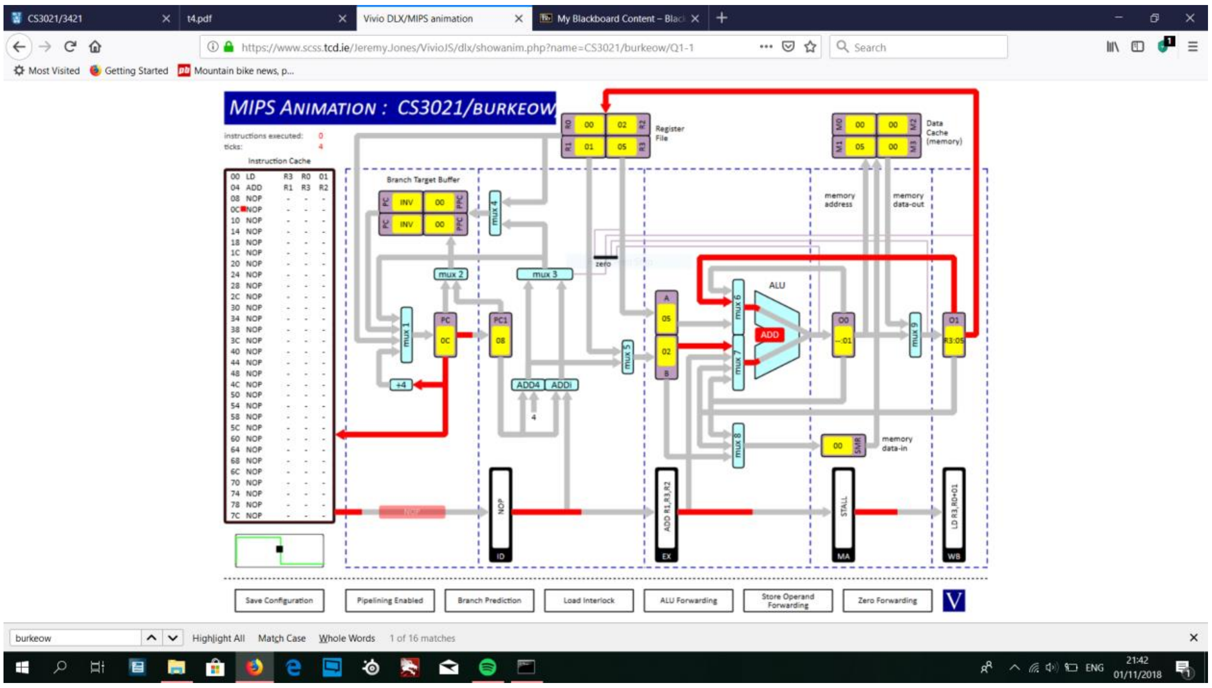
CS3021 Tutorial 4

Q1:

1)

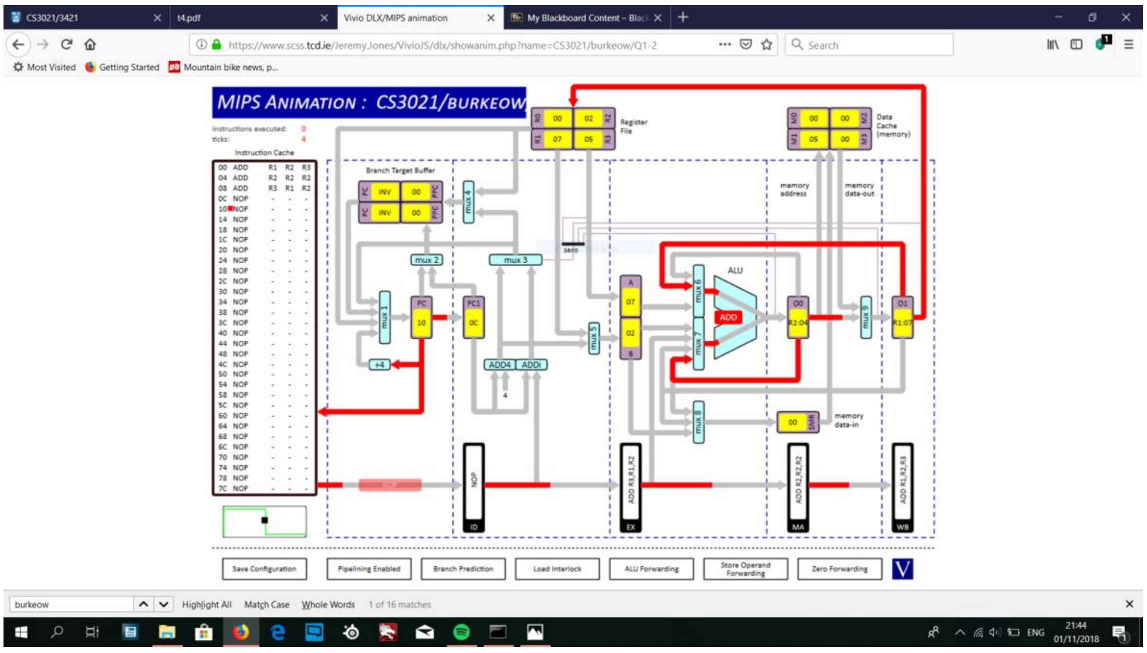


Code:

LD R3, R0, 01

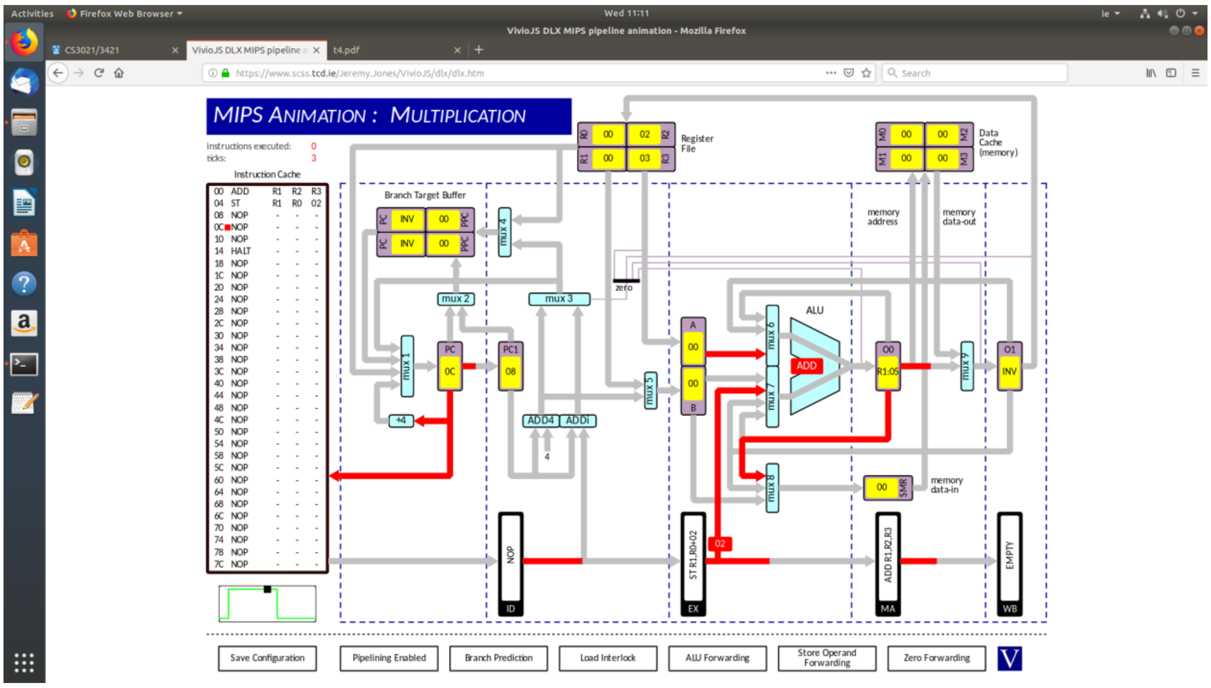
ADD R1, R3, R2

2)



Code:
ADD R1, R2, R3
ADD R2, R2, R2
ADD R3, R1, R2

3)

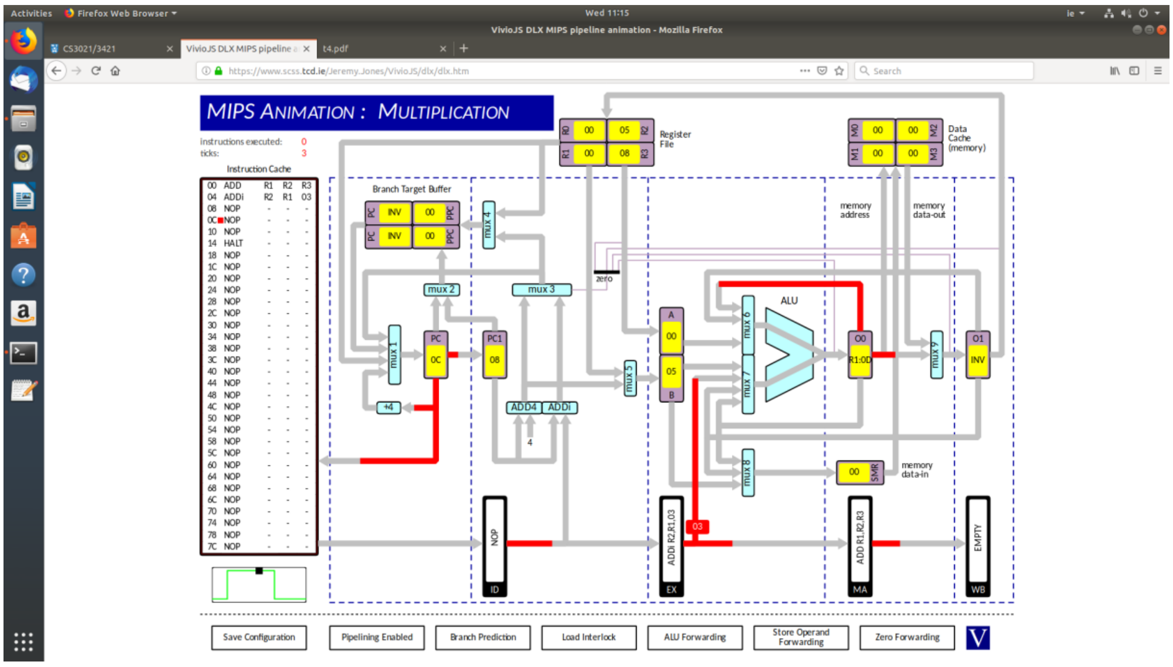


```
instructions executed:   0
ticks:                   3

Instruction Cache
00  ADD    R1  R2  R3
04  ST     R1  R0  02
08  NOP    -   -   -
0C  NOP    -   -   -
10  NOP    -   -   -
14  HALT   -   -   -
18  NOP    -   -   -
1C  NOP    -   -   -
20  NOP    -   -   -
24  NOP    -   -   -
28  NOP    -   -   -
2C  NOP    -   -   -
30  NOP    -   -   -
34  NOP    -   -   -
38  NOP    -   -   -
3C  NOP    -   -   -
40  NOP    -   -   -
44  NOP    -   -   -
48  NOP    -   -   -
4C  NOP    -   -   -
50  NOP    -   -   -
54  NOP    -   -   -
58  NOP    -   -   -
5C  NOP    -   -   -
60  NOP    -   -   -
64  NOP    -   -   -
68  NOP    -   -   -
6C  NOP    -   -   -
70  NOP    -   -   -
74  NOP    -   -   -
78  NOP    -   -   -
7C  NOP    -   -   -
```

Code:
ADD R1, R2, R3
ST R1, R0, 02

4)

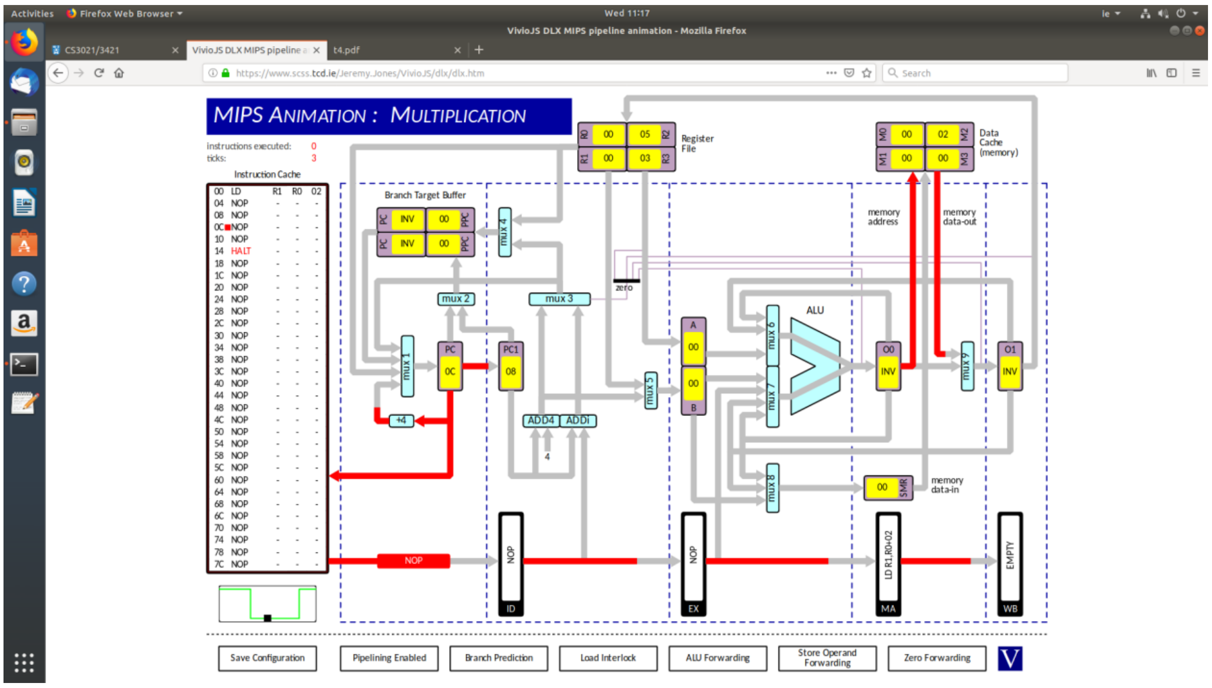


```
instructions executed:   0
ticks:                   3

Instruction Cache
00  ADD    R1  R2  R3
04  ADDi   R2  R1  03
08  NOP    -   -   -
0C  NOP    -   -   -
10  NOP    -   -   -
14  HALT   -   -   -
18  NOP    -   -   -
1C  NOP    -   -   -
20  NOP    -   -   -
24  NOP    -   -   -
28  NOP    -   -   -
2C  NOP    -   -   -
30  NOP    -   -   -
34  NOP    -   -   -
38  NOP    -   -   -
3C  NOP    -   -   -
40  NOP    -   -   -
44  NOP    -   -   -
48  NOP    -   -   -
4C  NOP    -   -   -
50  NOP    -   -   -
54  NOP    -   -   -
58  NOP    -   -   -
5C  NOP    -   -   -
60  NOP    -   -   -
64  NOP    -   -   -
68  NOP    -   -   -
6C  NOP    -   -   -
70  NOP    -   -   -
74  NOP    -   -   -
78  NOP    -   -   -
7C  NOP    -   -   -
```

Code:
ADD R1, R2, R3
ADDi R2, R1, 03

5)

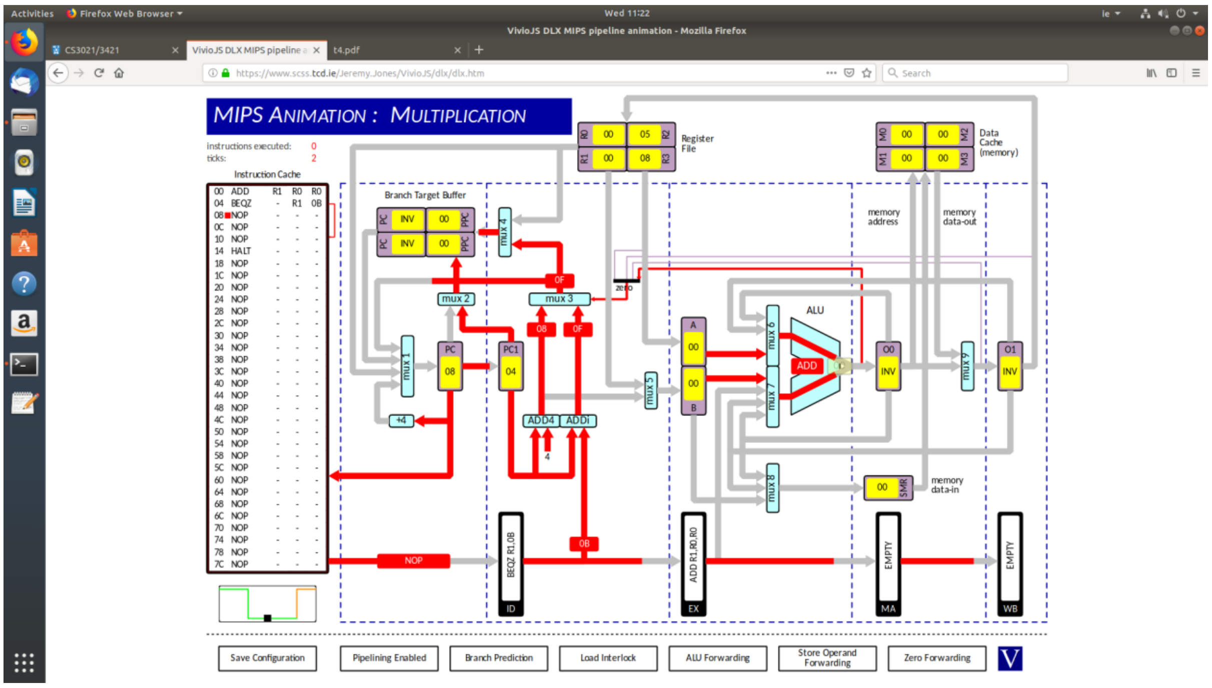


instructions executed: 0
ticks: 3

Instruction Cache
```
00  LD    R1  R0  02
04  NOP   -   -   -
08  NOP   -   -   -
0C  NOP   -   -   -
10  NOP   -   -   -
14  HALT  -   -   -
18  NOP   -   -   -
1C  NOP   -   -   -
20  NOP   -   -   -
24  NOP   -   -   -
28  NOP   -   -   -
2C  NOP   -   -   -
30  NOP   -   -   -
34  NOP   -   -   -
38  NOP   -   -   -
3C  NOP   -   -   -
40  NOP   -   -   -
44  NOP   -   -   -
48  NOP   -   -   -
4C  NOP   -   -   -
50  NOP   -   -   -
54  NOP   -   -   -
58  NOP   -   -   -
5C  NOP   -   -   -
60  NOP   -   -   -
64  NOP   -   -   -
68  NOP   -   -   -
6C  NOP   -   -   -
70  NOP   -   -   -
74  NOP   -   -   -
78  NOP   -   -   -
7C  NOP   -   -   -
```

Code:
LD R1, R0, 02

6)

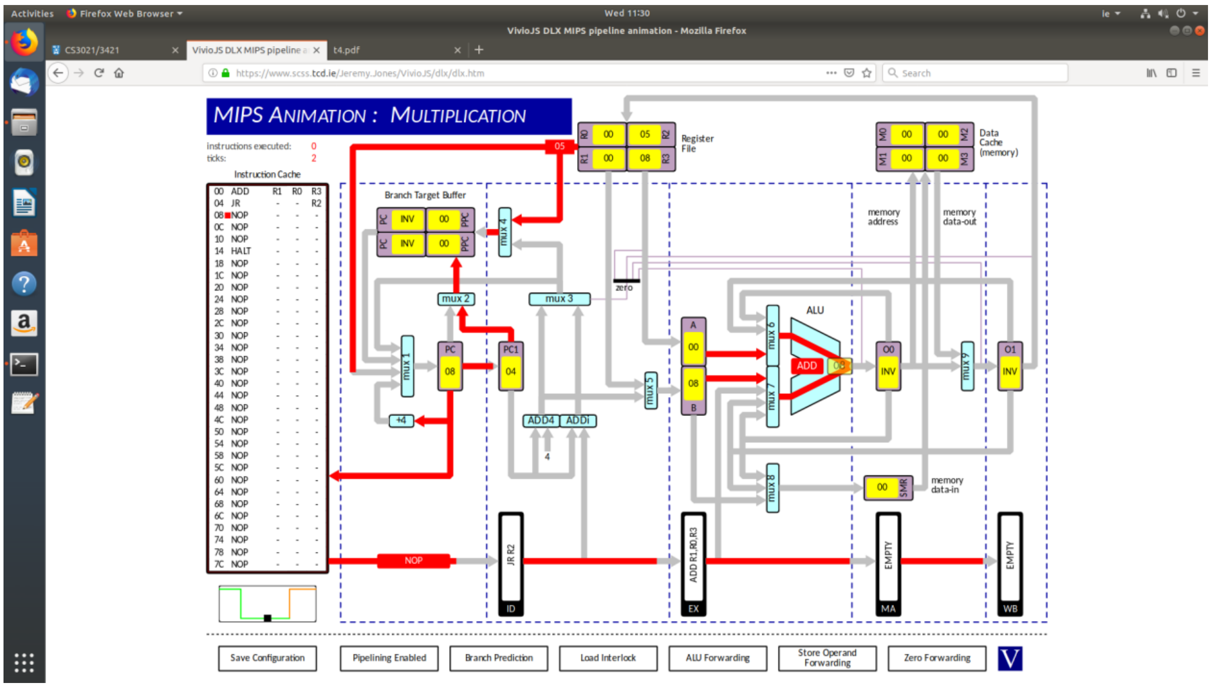


instructions executed: 0
ticks: 2

Instruction Cache
```
00  ADD   R1  R0  R0
04  BEQZ  -   R1  0B
08  NOP   -   -   -
0C  NOP   -   -   -
10  NOP   -   -   -
14  HALT  -   -   -
18  NOP   -   -   -
1C  NOP   -   -   -
20  NOP   -   -   -
24  NOP   -   -   -
28  NOP   -   -   -
2C  NOP   -   -   -
30  NOP   -   -   -
34  NOP   -   -   -
38  NOP   -   -   -
3C  NOP   -   -   -
40  NOP   -   -   -
44  NOP   -   -   -
48  NOP   -   -   -
4C  NOP   -   -   -
50  NOP   -   -   -
54  NOP   -   -   -
58  NOP   -   -   -
5C  NOP   -   -   -
60  NOP   -   -   -
64  NOP   -   -   -
68  NOP   -   -   -
6C  NOP   -   -   -
70  NOP   -   -   -
74  NOP   -   -   -
78  NOP   -   -   -
7C  NOP   -   -   -
```

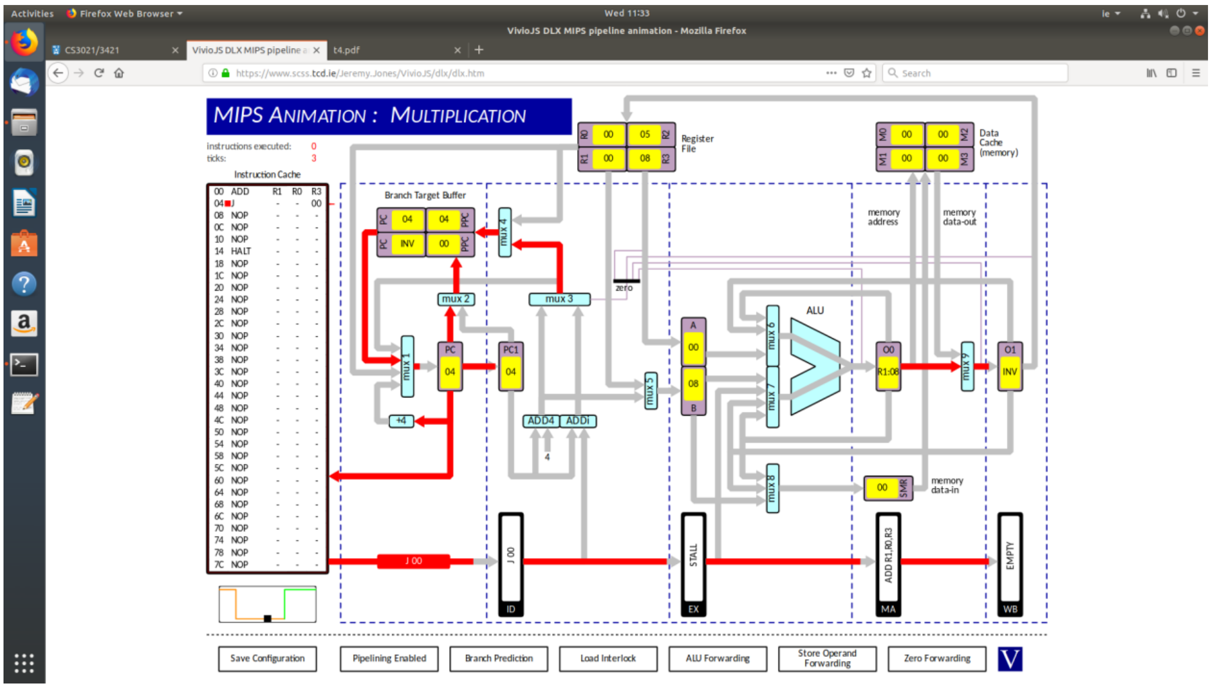
Code:
ADD R1, R0, R0
BEQZ -, R1, 0B

7)



Code:
ADD R1, R0, R3
JR -, -, R2

8)



Code:
```
ADD R1, R0, R3
J       -, -, 00
```

Q2: i) 10 cycles, Result = 21

ii) 18 cycles, Result = 21

iii) 10 cycles, Result = 6

The results and number of clock cycles are different due to the issues introduced by data dependencies and pipeline stalls.

When ALU forwarding is enabled, the cycle cycles are as low as can be achieved (10) and the result is as expected from the logic of the code. This is due to ALU forwarding, which feeds back the results from a previous instruction (from O0 in the simulation) if that result is required in the current instruction, avoiding data dependency issues while also maintaining a low number of clock cycles that are required.

If ALU forwarding is disabled, then CPU data dependency interlocks are required if the correct result is to be obtained. Dependency interlocks essentially stalls the pipeline between instructions that depend on each other's data, which is why it takes more clocks cycles to complete, due to the fact that more time is required to execute the instructions as some clock cycles are taken up by the stall.

If CPU data dependency interlocks are DISABLED, then the same number of clock cycles are required to complete the program as when ALU forwarding is enabled because there will be no pipeline stalls. However, the correct result will not be stored in R1. This is due to the fact that the instructions simply use the values that are present in the registers specified at that time rather than, for example, waiting until the correct data is loaded into the register from the previous instruction which has data it depends on. For example, take the second instruction (add r2, r1, r2). This will simply add 2(R2) & 1(R1) to get 3(R1), rather than 3(R1) & 2(R2) to get 5(R2) (the correct result should be 5, as the previous instruction should result in 3 in R1).

Q3)

i) Instructions Executed = 39
Clock cycles = 51

These two numbers are not equal, firstly, due to the fact that the processor is pipelined so for four instructions more than four clock cycles will be required in order for the instructions to propagate through the pipeline (i.e. more clock cycles than instructions will occur), and secondly, due to the fact that the pipeline stalls on certain instructions, effectively wasting clock cycles and hence increasing the number of clock cycles required will the number of instructions executed will remain as expected.

Firstly, the pipeline stalls on “SRLi R2, R2, 01” due to a data dependency (the previous instruction alters the value in R2, which must be updated before the SRLi instruction is executed in order to obtain the correct result, therefore stalling the pipeline and taking up clock cycles.

The pipeline also stalls the first time the unconditional jump is encountered. This is due to the fact that the offset from the PC must be calculated, and instructions sequentially following the jump must not be executed. However, it doesn’t stall the next time it encounters it due to predictive branching (i.e. it assumes the jump will be taken and hence the pipeline doesn’t need to stall.)

The pipeline also stalls on each conditional branch at certain points. The reason it doesn’t stall the first few times it meets these instructions is because the processor assumes that the branches won’t be taken, thereby avoiding stalls. However, this means the pipeline will then stall when the condition is actually met and the branch must be taken (when the loop is exited for the first conditional branch instruction (at 04), as the condition is met) & (when the value in R2 is 2 and that AND-ed with 01 equals 0, meeting the condition at 10 (the BEQZ, following the AND)).

ii) Instructions Executed = 39
Clock cycles = 53

These extra clock cycles are due to the fact that predictive branching is turned off, so there is a stall each time the processor hits the unconditional jump, as it must compute the offset every time rather than predicting it’s jump/offset, resulting in extra stalls, increasing the number of clock cycles required.

iii) When the shift instructions are swapped:
Instructions executed = 39
Clock cycles = 47

This is due to the fact that the data dependency has been eliminated by swapping the shift instructions (the load and shift right share a data dependency, and filling the space between with the shift left means that the correct result is written back into R2 by the time the shift right instruction is executed). Hence, the data dependency is removed and the need to have stalls in that case is also removed, resulting in less clock ticks required.