Assignment 1

José Antonio Ruiz Heredia

Exercise 1:

For this first exercise, we are asked to determine the minimum number of resources: '***number of reservation stations***', '***load buffers***', and '***store buffers***') needed so that the issue of instructions is never stopped at any moment.

The given code in ***ex1.s*** and ***ex1.d*** have the following istructions and their respectives types:

ld f2, a (***Load:*** NO dependencies with other instructions)
add r1, r0, xtop (***Integer:*** NO dependencies with other instructions)
ld f0, 0(r1) (***Load:*** It has a dependency with 'Instruction 2', as it uses **r1**)
sub r1, r1, #8 (***Integer:*** NO dependencies with other instructions)
multd f4, f0, f2 (***Floating Point:*** It has a dependency with 'Instruction 3', as it uses **f0**)
bnez r1, loop (***Branch:*** It has a dependency with 'Instruction 4', as it verifies **r1**)
sd 8(r1), f4 (***Store:*** It has a dependency with 'Instruction 5', as it uses **f4**)
trap #0 (***Trap***)

Then, the maximum instructions in flight:

- ***Loads*** (**2**): *ld f2, a* and *ld f0, 0(r1):*
- ***Integer*** (**2**): *add r1, r0, xtop* and *sub r1, r1, #8*
- ***Floating Point*** (**1**): *multd f4, f0, f2*
- ***Branch*** (**1**): *bnez r1, loop*
- ***Store*** (**1**): *sd 8(r1), f4*

The ***maximum number*** of instructions in flight is **7** instructions in total.

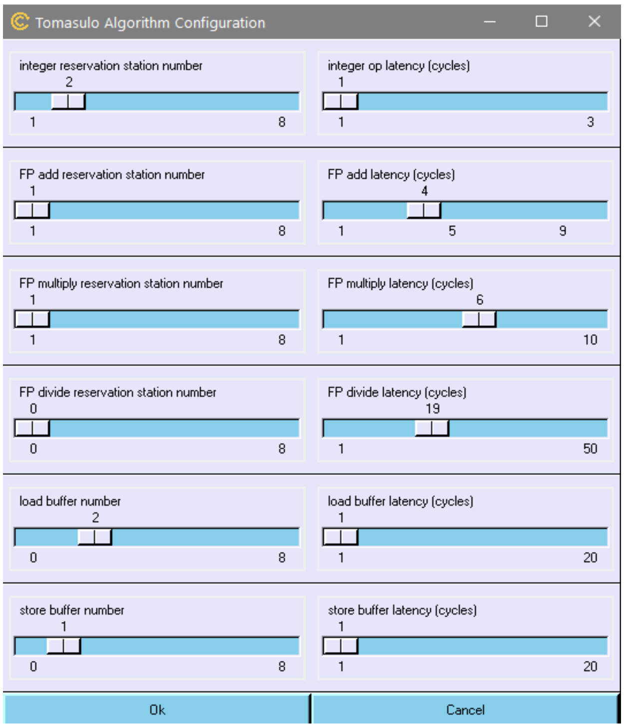
<u>Load buffers</u>: **2** ***store instructions***, then at least **2** ***store buffers***.

<u>Store buffers</u>: **1** ***load instructions***, then at least **1** ***load buffer***.

<u>Integer instructions</u>: **2** ***integer instructions***, then at least **2** ***INT reservation stations***.

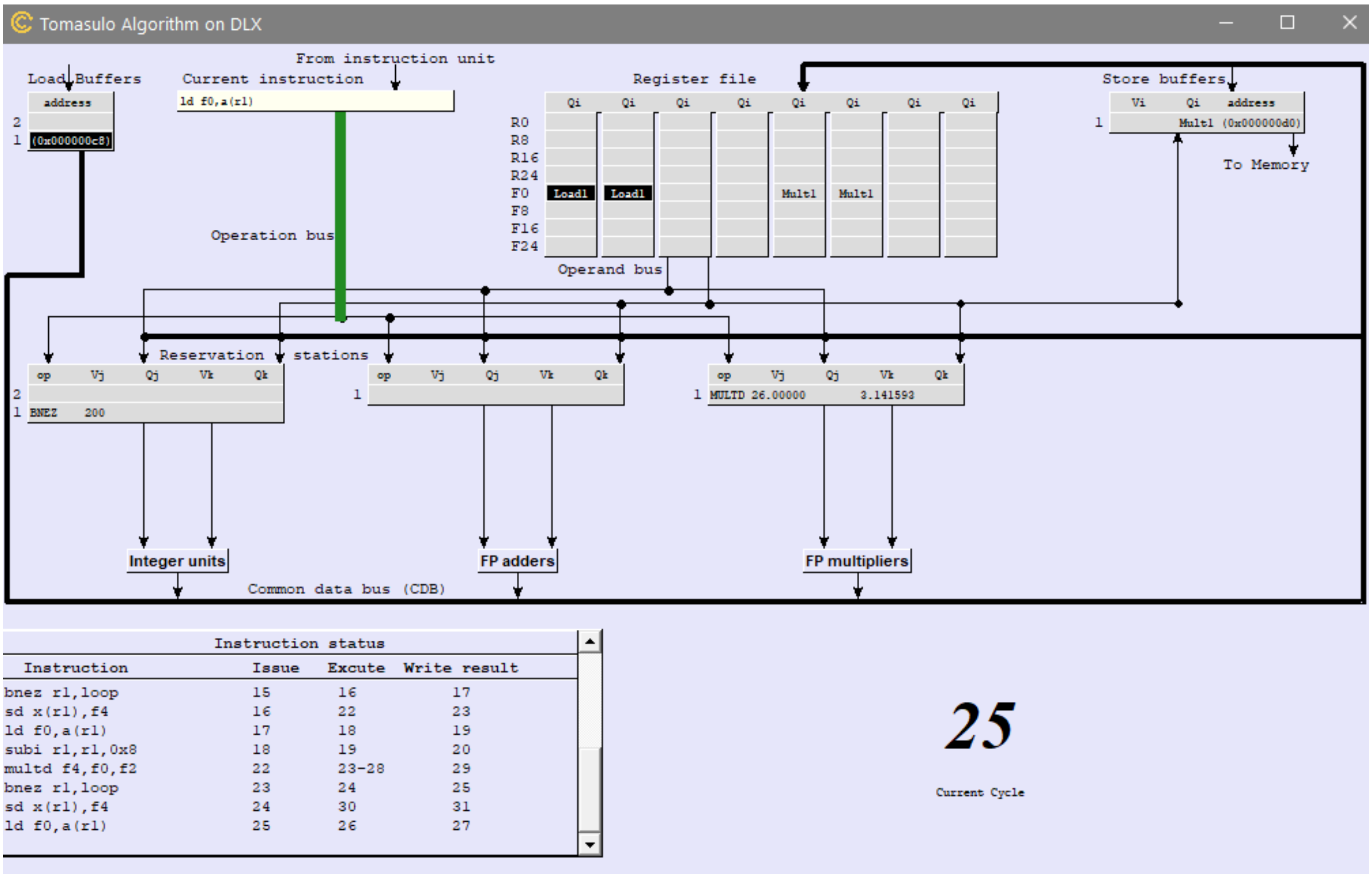
<u>Floating Point Add instructions</u>: **1** ***FPadd reservation station***.

<u>Floating Point Multiply instructions</u>: **1** ***FPmul instruction***, then at least **1** ***FPml reservation station***.



- Based on that, now we have to ***configure the Tomasulo algorithm in DLXview*** with the chosen resources :

Now we can see the ***execution process*** in the Tomasulo Algorithm:



And, as we observe, the only ***'problem'*** or ***'bottleneck'*** executing is result of the latency of the multiplication (*multd f4, f0, f2*). When we complete a loop cycle, we have to wait 6 cycles (**23** - **28**) for the multiplication to write its result (**29**) so we can execute the store instruction (**30**).

To avoid the impact of this latency and keep a continuous flow of the instructions, we might consider configuring *more reservation stations for FPmul*.

Exercise 2:

For this new exercise we, firstly, need to create a file "***ex2.d***" to declare all the variables needed: ***W***, ***B***, ***U***, ***X***, ***Y***.

```
# ex2.d - Data file for neural network
.data
# Matrix with weights W[2][5]
        .global W
W:      .float 0.3, 0.45, 1.2, 6.8, 3.2, 1.1, 0.8, 2.2, 1.5, 0.68
# Bias vector B[2]
        .global B
B:      .float 0.3, 1.1
# Threshold vector U[2]
        .global U
U:      .float 2.3, 3.8
# Input vector X[5]
        .global X
X:      .float 0.1, 0.1, 0.2, 0.3, 0.4
# Output vector Y[2]
        .global Y
Y:      .float 0.0, 0.0
```

Secondly, for the second file "***ex2.s***" we have to code in assembly the algorithm given in C.

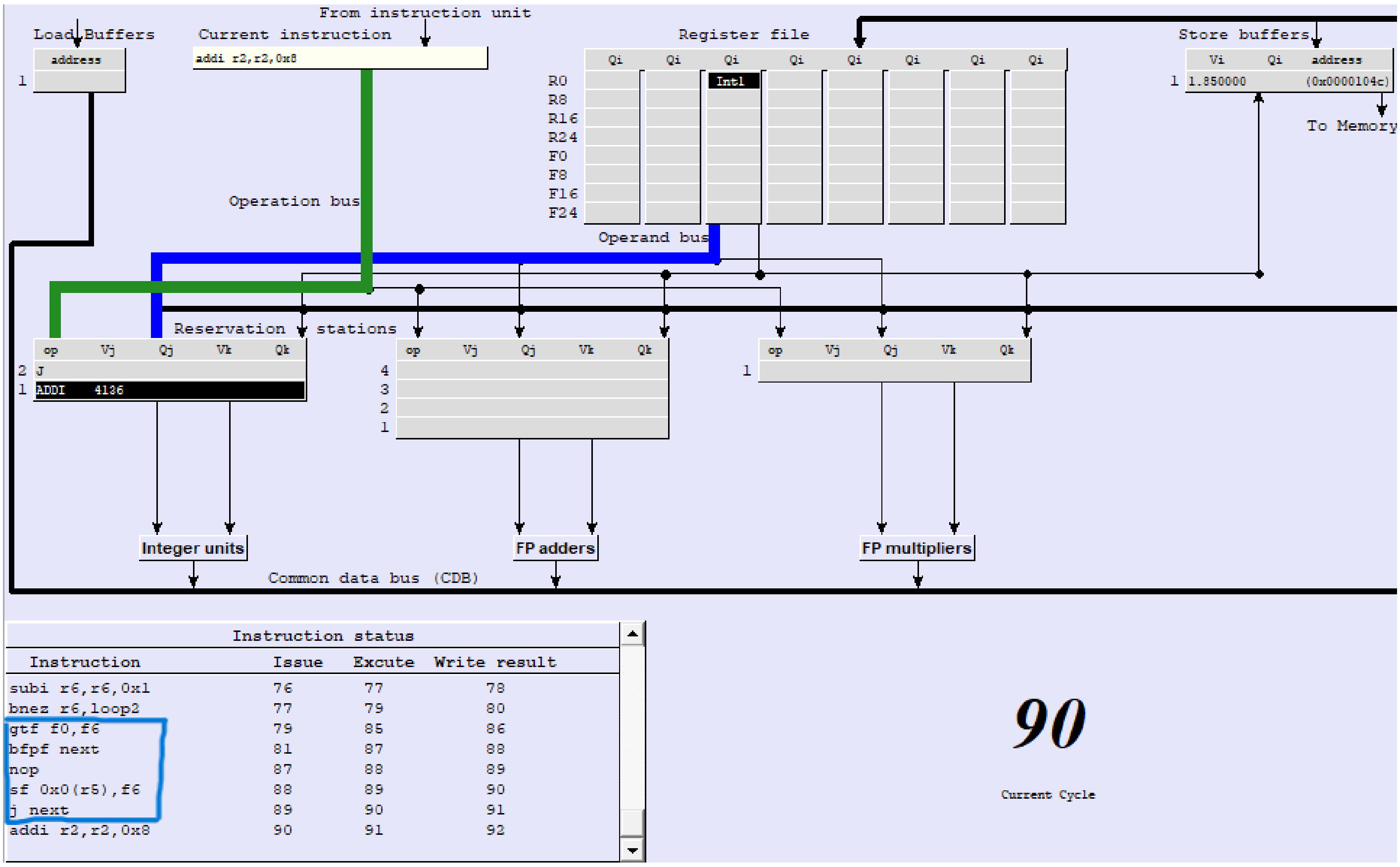
 # ex2.s - DLX assembly code for neural network

```
        add     r1, r0, W       # Register to store the address position W[0][0]
        add     r2, r0, B       # Register to store the address position B[0]
        add     r3, r0, U       # Register to store the address position U[0]
        add     r5, r0, Y       # Register to store the address position Y[0]
        add     r7, r7, #2      # Register to store i = 2 (to decrease i– in each iteration)

        # We inicialize r4 here as it uses 'j' so we have to reset its index to 0
loop1:  add     r4, r0, X       # Rregister to store the address position X[0]
        lf      f3, 0(r3)       # Load U[i]
        lf      f6, 0(r2)       # Load tmp = B[i]
        add     r6, r6, #5      # Register to store j = 5 (to decrease j– in each iteration)

loop2:  lf      f0, 0(r1)       # Load tmp = W[i][j]
        lf      f2, 0(r4)       # Load X[j]
        multf   f4, f0, f2      # f4 = W[i][j] * X[j]
        addf    f6, f6, f4      # tmp = tmp + f4
        addi    r4, r4, #8      # Increase address X[++]
        addi    r1, r1, #8      # Increase address W[++]
        subi    r6, r6, #1      # j–
        bnez    r6, loop2       # if (j !=0)  -> loop2
        gtf     f6, f3          # if (tmp > U[i])
        # If tmp > U[i] we continue to sf 0(r5), f6 so we enter the if statement, else we skip
        this part so we keep the initial value of Y[i] = 0.0
```

```
        bfpf   next
        nop                    # Wait for bfpf in order to not execute the store
        sf     0(r5), f6       # Y[i] = tmp
        j      next
next:   addi   r2, r2, #8      # Increase address B[++]
        addi   r5, r5, #8      # Increase address Y[++]
        addi   r3, r3, #8      # Increase address U[++]
        subi   r7, r7, #1      # i–
        bnez   r7, loop1       # if (j !=0)  -> loop1
        nop
        trap   #0
```

Here we can see a summary of the process of execution:

Firstly, after the first iteration of the ***loop1*** it *'enters'* the **if** and stores ***Y[i] = tmp;***



However, for the second iteration it *'enters'* the **else** so we keep ***Y[1] = 0.0*** (**skip sf and j**)