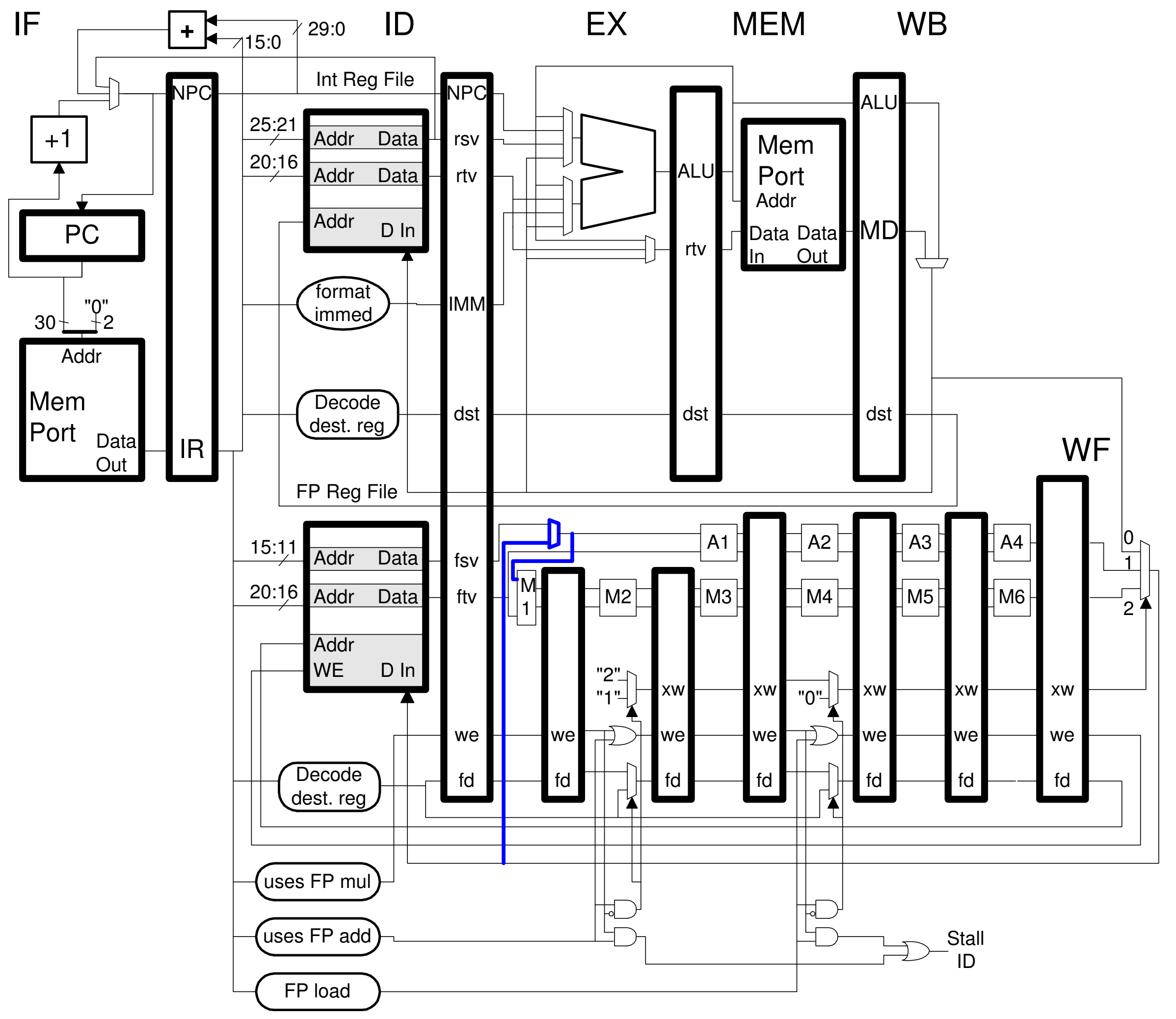
LSU EE 4720 **Homework 4** Solution **Due: 17 April 2006**

Problem 1: The code below executes on the illustrated MIPS implementation. The FP pipeline is fully bypassed but the bypass connections are not shown.



(*a*) Show a pipeline execution diagram. Solution shown below. The stall is for the dependency through register `f2`.

(*b*) Determine the CPI for a large number of iterations.

Because the second and third iterations start with the pipeline in the same state, the time for the second iteration can be used as a basis for computing CPI. The second iteration starts in cycle 3 (first instruction in IF), the third iteration starts in cycle 9, each iteration is 3 instructions to the $\boxed{\text{CPI is } \frac{9-3}{3} = 2}$.

(*c*) Add exactly the bypass connections that are needed.

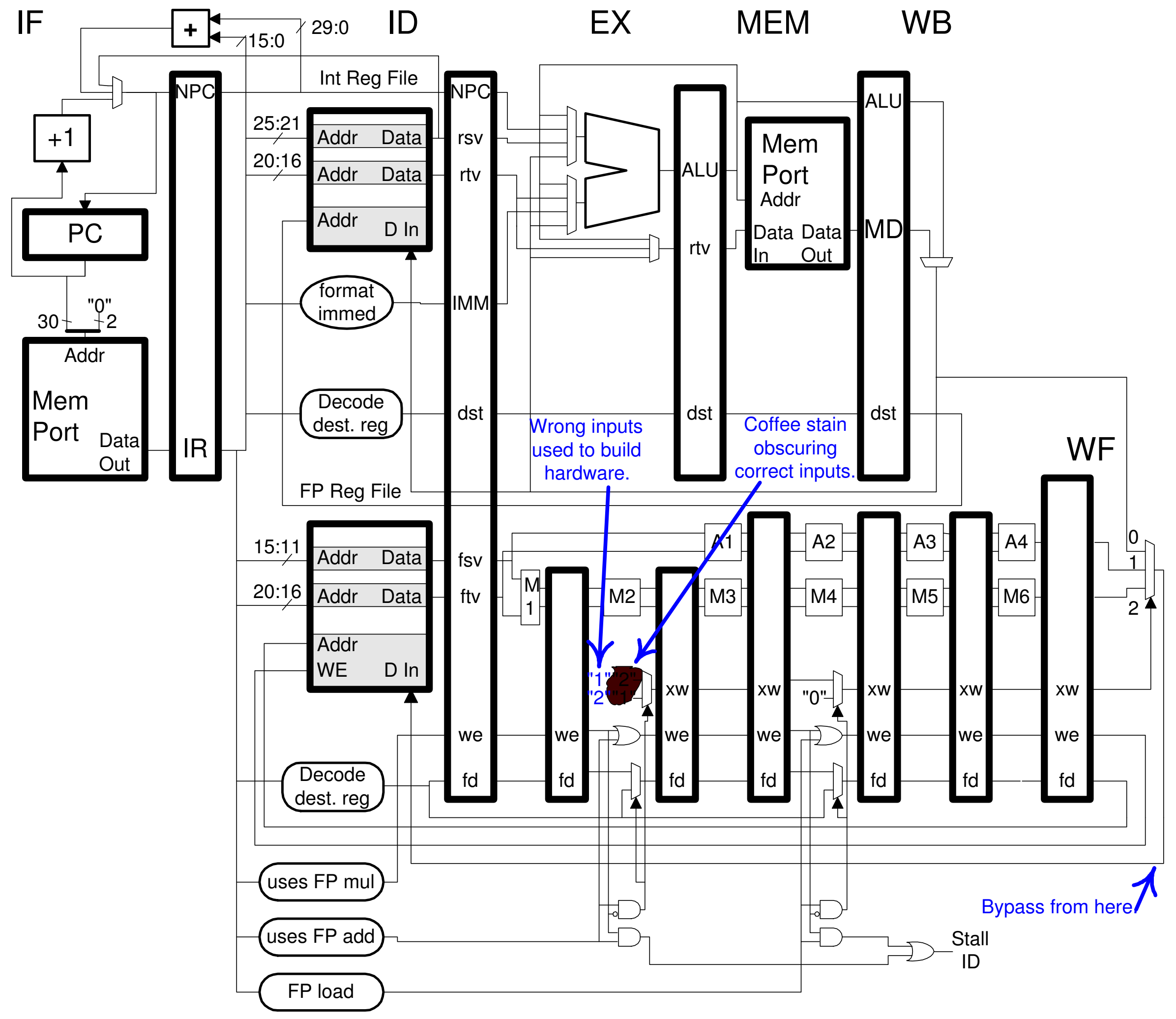
Solution shown in the diagram above. Added bypass connection shown in **blue bold**.

See next page for solution to first part.

```
 Solution to Problem 1a
LOOP:
# Cycle:           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 mul.d f2, f2, f4  IF ID M1 M2 M3 M4 M5 M6 WF
 bneq r1,0 LOOP       IF ID EX ME WB
 addi r1, r1, -1         IF ID EX ME WB
# Cycle:           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 mul.d f2, f2, f4           IF ID -------> M1 M2 M3 M4 M5 M6 WF
 bneq r1,0 LOOP                IF -------> ID EX ME WB
 addi r1, r1, -1                           IF ID EX ME WB
# Cycle:           0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
 mul.d f2, f2, f4                             IF ...
```

Problem 2: Due to a coffee spill the implementation below has a flaw: The inputs to the M2-stage XW mux have been reversed, the top input should be a 2 but is a 1, and the lower input should be a 1 but is a 2. There are no other flaws, in particular the control signal for the mux has been designed for a 2 at the upper input and a 1 at the lower input.

You are stranded alone on an island with this flawed implementation and to get off the island you need the result computed by the code below. The code was written for a normal MIPS implementation and will not compute the correct result on the flawed one. Re-write it so that it computes the correct result on the flawed implementation. (The solution must use the FP arithmetic units, do not simply implement IEEE 754 floating point using integer instructions.)



Solution on next page.

The problem with the implementation above is that when an **add.d** is in WF the mux will send the output of the FP multiply unit, not the FP add unit, to the register file. When a **mul.d** reaches WF the mux will choose the FP add unit for writeback.

Simply substituting a **mul.d** for the **add.d** won't work because the add unit would have gotten the wrong inputs. This happens in the example below where in cycle 8 the output of the adder, not the multiplier, is written back. The input values for the adder are determined by the instruction that was in ID in cycle 3, which is not **mul.d**.

```
# Cycle            0  1  2  3  4  5  6  7  8
mul.d f2, f4, f6   IF ID M1 M2 M3 M4 M5 M6 WF
                            ID A1 A2 A3 A4
```

To get the correct input to the add functional unit a second **mul.d** needs to be added. Suppose our goal is to execute **add.d f2, f4, f6**. Then start with a **mul.d** with the desired destination register but using dummy source registers. (See the code below.) Follow that with a **nop** and then another **mul.d** having a dummy destination but the desired source registers; when this second multiply reaches M1 its source operands will go to both the multiply unit (M1) and the add unit (A1). The first multiply will write the output of that add unit to the register file and so it will be as though **add.d f2, f4, f6** were executed.

```
# Code below effectively executes add.d f2, f4, f6
# Cycle            0  1  2  3  4  5  6  7  8  9  10
mul.d f2, f30, f30  IF ID M1 M2 M3 M4 M5 M6 WF        # Dummy sources.
nop                   IF ID EX ME WB
mul.d f30, f4, f6        IF ID M1 M2 M3 M4 M5 M6 WF  # Dummy destination
```

To get the original code below running on the faulty computer replace the **add.d** with a pair of multiplies. The only additional complication is that a source and destination register match, and that would create a confounding dependence stall if the exact technique above were used. Instead, the loop is unrolled so that one iteration of the re-written loop does the work of two iterations in the original loop. The first half-iteration writes f12 instead of f2, the second half-iteration reads f12 instead of f2.

The timing for the first iteration is shown. In the second iteration the first multiply should stall. The code is written assuming an even number of iterations in the original loop.

```
# Original code to be executed on the faulty computer.
LOOP:
 add.d f2, f2, f4
 bneq r1,0 LOOP
 addi r1, r1, -1
```

```
# Modified code that produces the same result as the code above.

 # Dummy registers: F26, F28, F30

 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
LOOP:
 mul.d f12, f2, F30  IF ID M1 M2 M3 M4 M5 M6 WF  # Dummy sources, but f2 needed for stall
 nop                   IF ID EX ME WB
 mul.d F26, f2, f4        IF ID M1 M2 M3 M4 M5 M6 WF                    # Dummy dest.

 mul.d f2, f12, F30          IF ID -------> M1 M2 M3 M4 M5 M6 WF        # Dummy source.
 nop                            IF -------> ID EX ME WB
 mul.d F28, f12, f4                         IF ID M1 M2 M3 M4 M5 M6 WF  # Dummy dest.

 beq r1, 0 LOOP                                IF ID EX ME WB
 addi r1, r2, -2                                  IF ID EX ME WB
 # Cycle            0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18
```