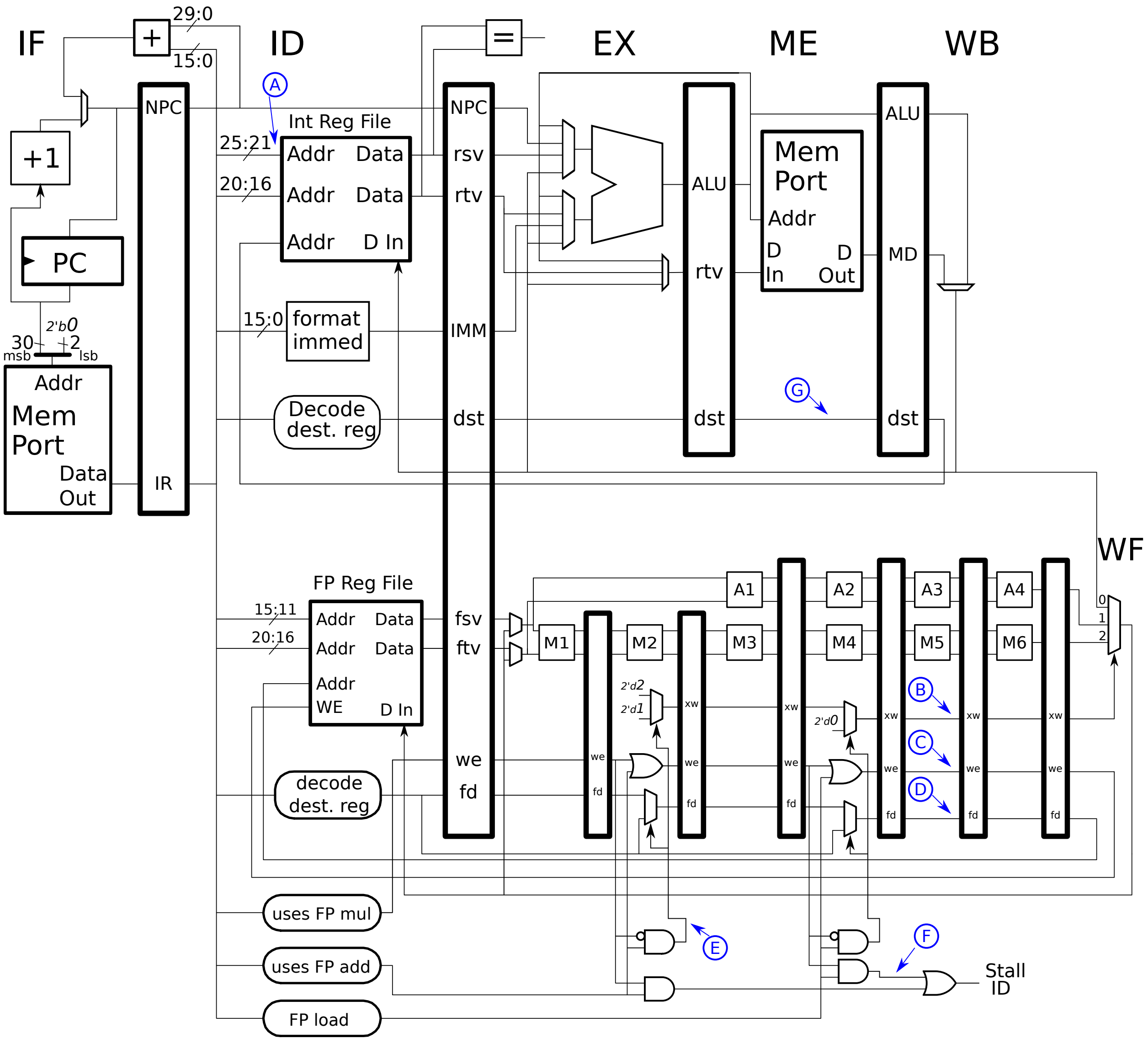
LSU EE 4720 **Homework 6** Solution **Due: 23 April 2018**

Problem 1: Solve 2014 Homework 4 Problem 1.

Problem 2: Appearing on the next page is a MIPS implementation and the execution of some code on that implementation.



(*a*) Wires in the implementation (on the previous page) are labeled in blue. Show the values on those wires each cycle that they are affected by the executing instructions. The values for label A are already filled in.

Solution appears below.

A common mistake for signals B, C, and D was to omit the `lwc1` instructions.

Signal E is `1` when there is an `add.s` or similar instruction in `ID` that (that, not which) will enter `A1` in the next cycle. Because both `add.s` instructions below are stalled signal E affects the respective instructions in *the last cycle of the stall* and so E is 1 in cycles 9 and 18 instead of 4 and 13.

Signal F stalls the pipeline if a `lwc1` can't enter the pipeline due to a structural hazard at `WF`. This doesn't happen for the two `lwc1` instructions below and so the signals are 0 in cycles 1 and 10, when the respective instructions are in `ID`.

Signal G carries the register number to write back into the integer register file. The only instruction below that actually writes an integer register is `addi`. It writes register 1 and so G is 1 in cycle 22, which is when `addi` is in `ME`.

```
LOOP: # Cycle       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
 lwc1 f0, 0(r1)     IF ID EX ME WF
 mul.s f1, f0, f10     IF ID -> M1 M2 M3 M4 M5 M6 WF
 add.s f2, f2, f1         IF -> ID -------------> A1 A2 A3 A4 WF
 lwc1 f0, 4(r1)                 IF -------------> ID EX ME WF
 mul.s f1, f0, f11                                IF ID -> M1 M2 M3 M4 M5 M6 WF
 add.s f2, f2, f1                                    IF -> ID -------------> A1 A2 A3 A4 WF
 bne r2, r1 LOOP                                           IF -------------> ID EX ME WB
 addi r1, r1, 8                                                              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 19 20 21 22 23
A                      1                          1                          2   1 # Sample
B                         0                 2        0  1              2           1
C  (Default 0)            1                 1        1  1              1           1
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
D                         0                 1        0  2              1           2
E  (Default 0)                                 1                          1
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
F  (Default 0)         0                          0
G  (Default 0)                                                                        1
# Cycle             0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

(*b*) Schedule the code above so that it suffers fewer stalls. Register numbers can be changed but in the end the correct value must be in register `f2`. It is okay to add a few instructions before and after the loop. Each iteration must do the same amount of work as the original code.

Show a pipeline execution diagram for two iterations.

Two solutions appear below. The first reorganizes instructions within an iteration, but stalls six cycles per iteration. The second solution stalls just three cycles per iteration by executing the `add.s` instructions in the iteration after the corresponding `mul.s` was executed. All of the stalls in the second solution are due to structural hazards. (This is an example of *software pipelining*. Software pipelining was not covered in this course, though a related technique, loop unrolling, was covered.)

```
# SOLUTION: Easy, but not best solution.

LOOP: # Cycle:      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
 lwc1 f0, 0(r1)     IF ID EX ME WF                                                                     FIRST ITERATION
 lwc1 f5, 4(r1)        IF ID EX ME WF
 mul.s f1, f0, f10        IF ID M1 M2 M3 M4 M5 M6 WF
 mul.s f6, f5, f11           IF ID M1 M2 M3 M4 M5 M6 WF
 addi r1, r1, 8                 IF ID EX ME WB
 add.s f2, f2, f1                  IF ID -------> A1 A2 A3 A4 WF
 bne r1, r2, LOOP                     IF -------> ID EX ME WB
 add.s f2, f2, f6                                 IF ID ----> A1 A2 A3 A4 WF
LOOP: # Cycle:      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
 lwc1 f0, 0(r1)                                      IF ----> ID EX ME WF                             SECOND ITERATION
 lwc1 f5, 4(r1)                                               IF ID -> EX ME WF
 mul.s f1, f0, f10                                               IF -> ID M1 M2 M3 M4 M5 M6 WF
 mul.s f6, f5, f11                                                     IF ID M1 M2 M3 M4 M5 M6 WF
 addi r1, r1, 8                                                           IF ID EX ME WB
 add.s f2, f2, f1                                                            IF ID -------> A1 A2 A3 A4 WF
 bne r1, r2, LOOP                                                               IF -------> ID EX ME WB
 add.s f2, f2, f6                                                                           IF ID ----> A1 A2 A3 A4 WF
LOOP: # Cycle:      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
 lwc1 f0, 0(r1)     THIRD ITERATION                                                            IF ----> ID EX ME WF
```

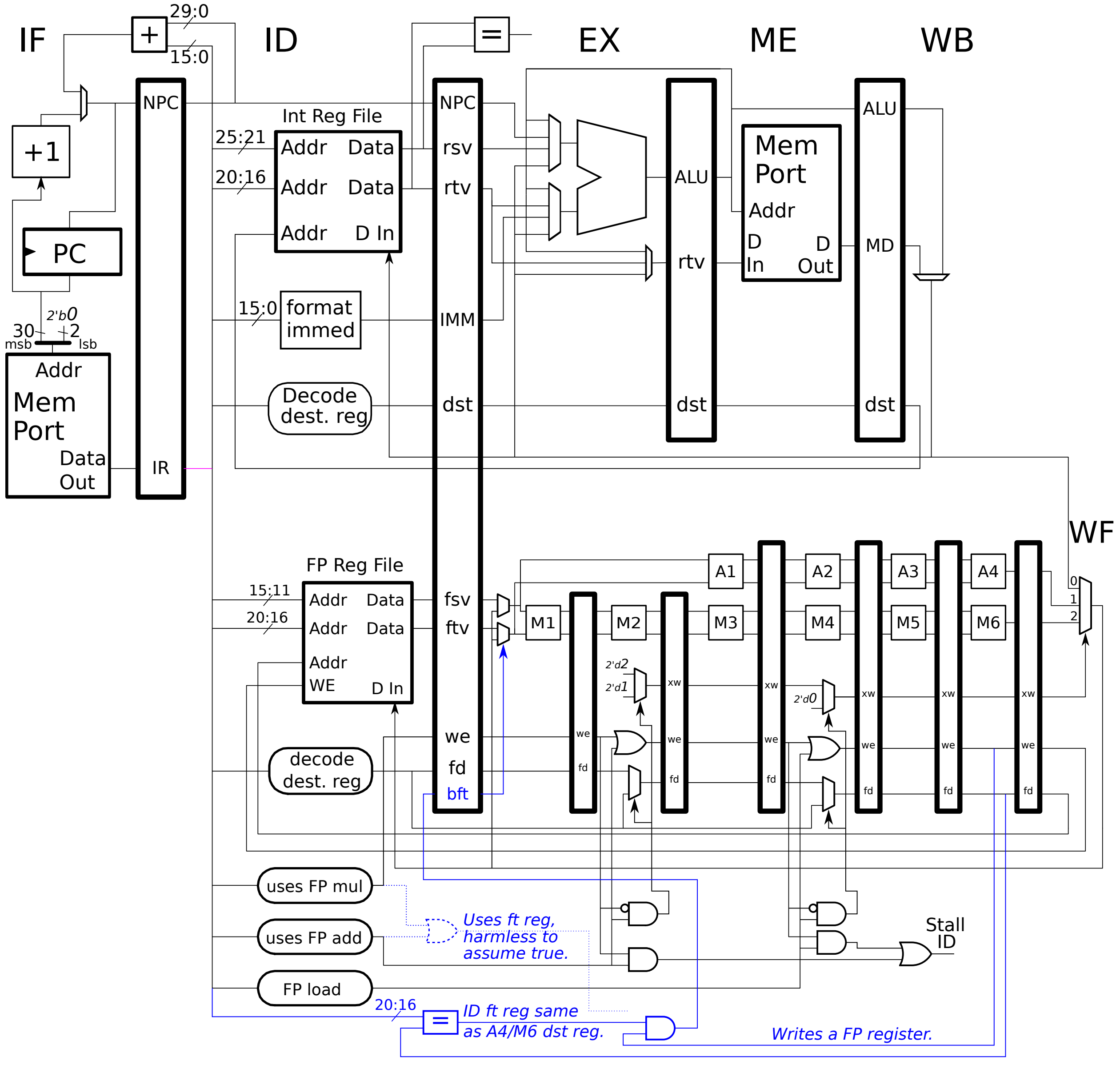
Better solution on next page.

```
# SOLUTION: Better solution: avoid more stalls by executing add.s's in next iteration.

# Prologue Code.  Performs first iteration (except for adds).
 lwc1 f0, 0(r1)
 lwc1 f5, 4(r1)
 mul.s f1, f0, f10
 mul.s f6, f5, f11
 addi r1, r1, 8
 addi r2, r2, 8  # Increment because branch now checks r2 after r1 is incremented.

LOOP: # Cycle:       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21
 lwc1 f0, 0(r1)      IF ID EX ME WF                                    FIRST ITERATION
 add.s f2, f2, f1       IF ID A1 A2 A3 A4 WF
 lwc1 f5, 4(r1)            IF ID EX ME WF
 addi r1, r1, 8               IF ID EX ME WB
 mul.s f1, f0, f10               IF ID M1 M2 M3 M4 M5 M6 WF
 add.s f2, f2, f6                   IF ID A1 A2 A3 A4 WF
 bne r1, r2, LOOP                      IF ID EX ME WB
 mul.s f6, f5, f11                        IF ID M1 M2 M3 M4 M5 M6 WF
LOOP: # Cycle:       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
 lwc1 f0, 0(r1)                              IF ID -> EX ME WF
 add.s f2, f2, f1                               IF -> ID A1 A2 A3 A4 WF
 lwc1 f5, 4(r1)                                       IF ID ----> EX ME WF
 addi r1, r1, 8                                          IF ----> ID EX ME WB
 mul.s f1, f0, f10                                                IF ID M1 M2 M3 M4 M5 M6 WF
 add.s f2, f2, f6                                                    IF ID A1 A2 A3 A4 WF
 bne r1, r2, LOOP                                                       IF ID EX ME WB
 mul.s f6, f5, f11   SECOND ITERATION                                      IF ID M1 M2 M3 M4 M5 M6 WF
LOOP: # Cycle:       0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
 lwc1 f0, 0(r1)                                                               IF ID -> EX ME WF


# Epilogue Code: Performs add.s's for last iteration
 add.s f2, f2, f1
 add.s f2, f2, f6
```

Problem 3: Design control logic for the lower M1-stage bypass multiplexor. Note that this is fairly easy since the mux has two inputs, so it's only necessary to detect the dependence.

An SVG source for the FP diagram can be found at:
`http://www.ece.lsu.edu/ee4720/2018/mpipei-fp-by.svg`.

Solution appears below in blue. The control signal is named `bft`. Its value is computed in ID and used in `M1A1`. The logic compares the `ft` register of the instruction in ID with the `fd` register of the instruction in `A4M6`, as well as checking whether the instruction in `A4M6` is going to write a FP register (the `we` [write enable] bit is set). The logic drawn with dashed lines checks whether the instruction in ID uses an `ft` source register (uses the multiply or add pipeline). Conceptually this should be AND'ed with the output of the register comparison and the `we` signal, but in the solution its value is assumed true. That eliminates an OR gate and an input on the AND gate and does not interfere with correct operation.


IF
ID
EX
ME
WB
WF
29:0
15:0
+
+1
PC
NPC
IR
Int Reg File
Addr Data
Addr Data
Addr D In
25:21
20:16
15:0
format immed
Decode dest. reg
2'b0
30
2
msb
lsb
Addr
Mem Port
Data Out
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem Port
Addr
D In
D Out
ALU
MD
dst
FP Reg File
15:11
20:16
Addr Data
Addr Data
Addr
WE
D In
fsv
ftv
we
fd
bft
decode dest. reg
M1
M2
M3
M4
M5
M6
A1
A2
A3
A4
2'd2
2'd1
2'd0
xw
we
fd
0
1
2
uses FP mul
uses FP add
FP load
20:16
Uses ft reg, harmless to assume true.
ID ft reg same as A4/M6 dst reg.
Writes a FP register.
Stall ID