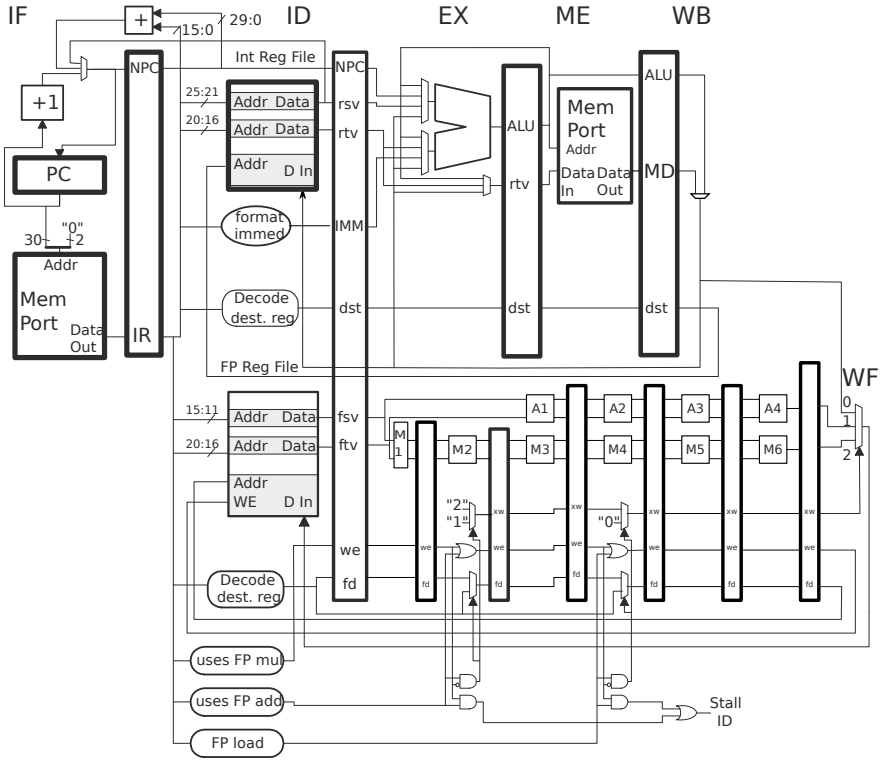
LSU EE 4720 **Homework 4** Solution **Due: 24 March 2014**

Problem 1: The following code fragments execute incorrectly on the following pipeline. For each fragment describe the problem and correct the problem.



(*a*) Describe problem and fix problem.

```
lwc1  f2, 0(r1)   IF ID EX ME WF
add.s f1, f2, f3     IF ID A1 A2 A3 A4 WF
```

There is a dependence between the `lwc1` and the `add.s` through `f2` and so the `add.s` should have stalled. Correct execution appears below for a pipeline in which bypass paths exist from `WF` into `A1` and `M1`.

```
# SOLUTION        0  1  2  3  4  5  6  7  8  9
lwc1  f2, 0(r1)   IF ID EX ME WF
add.s f1, f2, f3     IF ID -> A1 A2 A3 A4 WF
```

(*b*) Describe problem and fix problem.

```
# Cycle              0  1  2  3  4  5  6
add.s f1, f2, f3     IF ID A1 A2 A3 A4 WF
addi r1, r1, 4          IF ID EX ME WB
lwc1  f2, 0(r1)            IF ID EX ME WF
```

There are two instructions in `WF` in cycle `6`, which is impossible on this pipeline. The solution is to stall `lwc1` by one cycle, shown below.

```
# SOLUTION           0  1  2  3  4  5  6
add.s f1, f2, f3     IF ID A1 A2 A3 A4 WF
addi r1, r1, 4          IF ID EX ME WB
lwc1  f2, 0(r1)            IF ID -> EX ME WF
```

Note that if the `addi` and the `lwc1` changed places there would be no reason to stall. However the problem wasn't a stall, the problem was that the hardware should have stalled and didn't, presumably resulting in an incorrect value in either `f2` or `f1`. So swapping the two instruction doesn't fix the problem it just works around (avoids) it.

(*c*) Describe problem and fix problem.

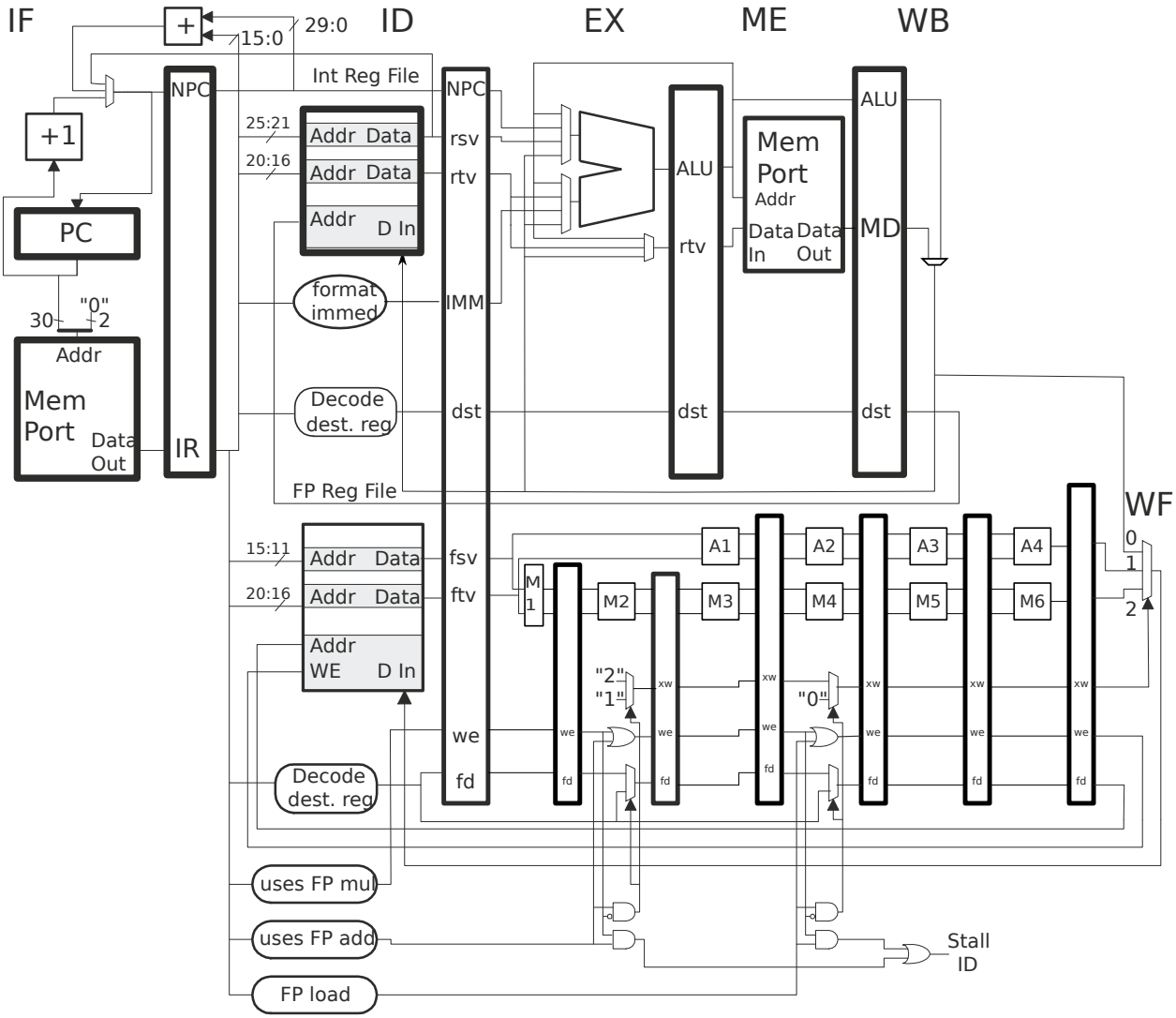
```
add.d  f1, f2, f3      IF ID A1 A2 A3 A4 WF
```

The instruction above is a double-precision add (notice the `.d` at the end). In MIPS-I and other 32-bit RISC ISAs double precision instructions can only use even-numbered registers as operands. (Each 64-bit operand is obtained from two registers, the even-numbered register and the next register, for example, `f10` and `f11`.

Lets fix this under the assumption that the programmer meant to use a double-precision add but used the wrong registers. In that case make the odd registers even:

```
add.d  f10, f2, f4     IF ID A1 A2 A3 A4 WF
```

Problem 2: The code fragment below contains a MIPS floating-point comparison instruction and branch. The pipeline illustrated below does not have a comparison unit, in this problem we will add one. The comparison unit to be used has two stages, named `C1` and `C2`. The output of `C2` is one bit, indicating if the comparison was true.




```
c.gt.d f2, f4
bc1t TARG
add.d  f2, f2, f10
...
TARG: xor r1, r2, r3
```

(*a*) Add the comparison unit to the pipeline above. Also add a new register `FCC` (floating point condition code) that is written by the comparison instruction and is used by the control logic to determine if a floating-point branch is taken.

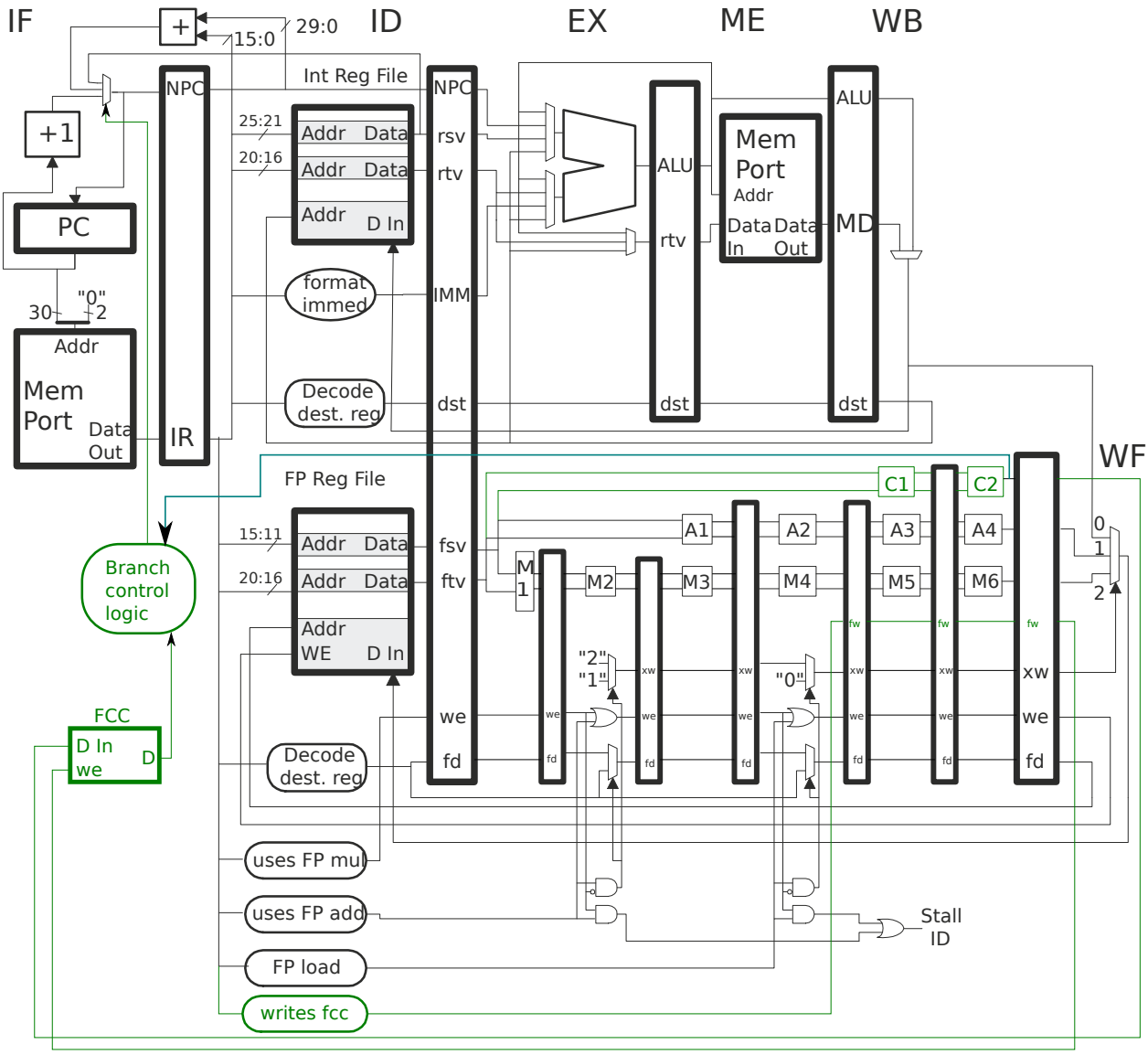
The `FCC` register should have a data and write-enable input, show the control logic generating the write-enable signal. Show a cloud labeled "branch control logic" and connect it to appropriate datapath components.

Solution appears below in green. (The solution to part c is in blue.)

The comparison units were added to the two stages before WF. The write enable signal enters the pipeline in M4, using a dedicated latch segment named fw. The fw signal is set by logic in the ID stage detecting instructions that write the FCC register, including c. (The c instruction cannot use the we signal used by other FP instructions because that is meant for the FP register file.)

The new FCC register is shown in the ID stage, with its output connected to a new branch control logic cloud. The output of that cloud connects to the PC mux.

Notice that there is no stall logic for the c instruction. That was not asked for in this problem, and it would not be needed if only five-stage instructions could write FCC.



(*b*) Show the execution of the code sample above on your modified hardware, but without any bypass paths for the added hardware.

Solution appears below. Note that the `bc1t` instruction must wait in `ID` until the `c` instruction reaches `WF`.

```
# SOLUTION          0  1  2  3  4  5  6  7  8  9  10
c.gt.d f2, f4       IF ID C1 C2 WF
bc1t TARG              IF ID ----> EX ME WB
add.d  f2, f2, f10        IF ----> ID A1 A2 A3 A4 WF
...
TARG: xor r1, r2, r3                IF ID EX ME WB
```

(*c*) Add whatever bypass paths are needed so that the code executes with as few stalls as possible **but** without having a major impact on clock frequency. Assume that `C2` produces a result in about 80% of the clock period.

The added bypass path are in sky blue on the diagram above. (The path goes from the output of `C2` to the branch-control-logic cloud.) With this bypass path one fewer stall is needed. That execution is shown below.

```
# SOLUTION          0  1  2  3  4  5  6  7  8  9
c.gt.d f2, f4       IF ID C1 C2 WF
bc1t TARG              IF ID -> EX ME WB
add.d  f2, f2, f10        IF -> ID A1 A2 A3 A4 WF
...
TARG: xor r1, r2, r3            IF ID EX ME WB
```

Source files for the diagram are at:
`http://www.ece.lsu.edu/ee4720/2014/mpipeifp.eps`,
`http://www.ece.lsu.edu/ee4720/2014/mpipeifp.svg`.
The `svg` file can be edited using Inkscape.