EE 4720 **Homework 3** **Due: 3 March 1997**

In *predicated execution* a condition-code bit or register is used to determine if an instruction will execute. In a simple version some machine instructions have predicated forms, for example, with a predicated load word the following conventional code

```
 BNEQZ skip    ! Branch if condition codes indicate != 0.
 LW R1,12(R2)  ! Load register R1.
skip:
 ADD R3, R1, R2
```

can be replaced with

```
 LW,EQ R1,12(R2)  ! Load register R1 if condition codes indicate = 0, otherwise do nothing.
 ADD R3, R1, R2
```

saving not just one instruction, but also possible stall cycles.

The branch and `LW,EQ` use condition code registers, while DLX uses regular registers for conditions. The means that there would be no place to specify a condition register in a DLX `LW,EQ` without removing the immediate.

Another way of adding predicated execution to DLX makes use of a special `PM`, *predicate mask*, instruction. The instruction specifies a condition register and two 8-bit execution masks, the first mask is used if the register contents is non-zero, the other if the contents is zero. The 8 bits in a mask refer to the next 8 instructions in program order (*e.g.*, following branches) to be executed, the LSB (bit 0) refers to the next instruction, bit 1 refers to the instruction after that, and so on. (Control flow instructions make things interesting, for now ignore them.) If a mask bit is zero the corresponding instruction does not execute, if the bit is one it does. An instruction not executed in this manner is said to be *nullified*. The mnemonic for `PM` indicates the register to test and the mask to use if the register contents is non-zero and the mask to use if the register contents is zero. For example, if the instruction indicated by

```
 PM R1, 0x00, 0xff ! Note, 0x00 = 00000000 (binary) and 0xff = 11111111 (binary)
```

executes and R1 is not equal to zero, the next eight instructions are not executed; if R1 is equal to zero the next eight instructions are executed.

The `PM` instruction is used in the first code fragment as follows:

```
 PM R4, 0xfe, 0xff      ! Note: 0xfe = 11111110 (binary)
 LW R1,12(R2)           ! Load register R1 if R4 equals zero.
 ADD R3, R1, R2         ! Always executes.
```

Problem 1: Show two DLX code fragments, one with and one without `PM` designed to show `PM` in the best possible light (using analysis from next problem).

Problem 2: Consider the pipelined DLX implementation presented in class and the text. Suppose taken branches stall this implementation by three cycles but no stalls result from `PM` instructions. (The nullified instructions are not counted as stalls.) Determine the CPI and execution time on this machine of the code fragments developed above. (Nullified instructions are not tallied in the instruction count.)

Problem 3: Show how the pipelined DLX implementation would have to be modified to implement `PM`. Be sure to show any registers added and to explain what additional actions are performed by the controller.

Problem 4: The description above did not specify constraints on what instructions can follow a `PM`. Determine which instructions would cause problems and provide suggestions on what to do about them. Problems might include ambiguity on which instructions execute and confusing execution paths with no apparent computational benefit.