

Problem 1: The code below executes on a dynamically scheduled four-way superscalar DLX implementation that uses reorder buffer entry numbers to name destination registers.

- Loads and stores use the load/store unit, which consists of segments L1 and L2.
- The floating-point multiply unit is fully pipelined and consists of six segments.
- The usual number of instructions (for a 4-way superscalar machine) can be fetched, decoded, and committed per cycle.
- An unlimited number of instructions can complete per cycle. (This makes the solution easier.)
- There are an unlimited number of reservation stations and reorder buffer entries.
- The target of a branch is fetched in the cycle after the branch is in ID, *whether or not the branch condition is available*. (We'll cover that later.)

(a) Show a pipeline execution diagram for the code below until the beginning of the fourth iteration. Show where instructions commit.

See diagram below.

(b) What is the CPI for a large number of iterations? *Hint: There should be less than six cycles per iteration.*

The CPI is $\frac{3}{6} = 0.5$.

(c) Show the entries in the register map for registers `f0` and `r1` for each cycle. (Make up reorder buffer entry numbers.)

See pipeline execution diagram on the next page.

```

! Solution
LOOP:
! Cycle      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
ld f0, 0(r1)  IF ID L1 L2 WC
                IF ID L1 L2 WB                C
                    IF ID L1 L2 WB                C
                        IF ID L1 L2 WB
muld f0, f0, f2  IF ID RS RS M1 M2 M3 M4 M5 M6 WC
                    IF ID RS RS M1 M2 M3 M4 M5 M6 WC
                        IF ID RS RS M1 M2 M3 M4 M5 M6 WC
                            IF ID RS RS M1 M2 M3 M4 M5 M6 WC
sw 0(r1), f0     IF ID L1                L2 WC
                    IF ID L1                L2 WC
                        IF ID L1                L2 WC
addi r1, r1, #8  IF ID EX WB                C
                    IF ID EX WB                C
                        IF ID EX WB                C
sub r2, r1, r3   IF ID EX WB                C
                    IF ID EX WB                C
                        IF ID EX WB                C
bnez r2, LOOP    IF ID RS B  WB                C
                    IF ID RS B  WB                C
                        IF ID RS B  WB                C
! Cycle      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17

```

```

ID Map
f0 1.0*      #7,8    #13,14  #19,20  #25,26  #32,#33  #38,39
r1 0x1000*   #10    0x1008  #22    0x1018  #34    0x1028
r1          #16    0x1010  #28    0x1020  #40
! Note: Because of space restrictions r1 is shown on two lines. The
! first character of an entry is the cycle number for the entry. For
! example, 0x1008 is written in to the map at cycle 3 and #16 at cycle 4.

```

```

Commit Map
f0 1.0*      10.0                11    20 22    30 33
r1 0x1000*           0x1008    0x1010    0x1018
! Cycle      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17

```

```

* Initial Values
f0: 1.0
r1: 0x1000
f2: 1.1
Mem[0x1000] = 10.0
Mem[0x1008] = 20.0
Mem[0x1010] = 30.0

```

(d) The first two instructions of the code below are different than the code above, the other instructions are identical. It runs on a system identical to the one above except that there are only 1000 reorder buffer entries. (That's actually a lot, but it's not unlimited.) What is the CPI for a large number of iterations? Is the CPI really lower in the period before reorder buffers are used up? If you can, solve the problem without drawing a complete pipeline execution diagram.

```

LOOP: ! LOOP = 0x1000
  ld f4, 0(r1)
  muld f0, f0, f4
  sw 0(r1), f0
  addi r1, r1, #8
  sub r2, r1, r3
  bnez r2, LOOP

```

The CPI is $\frac{6}{6} = 1$. Though iterations start every three cycles before the reorder buffer fills, the state of the system is different at each start (in particular, the number of instructions waiting in the reorder buffer increases), and so one cannot base CPI on an iteration time of three cycles. The number of cycles per iteration is limited by the time needed to multiply, which is six.

```

ld f0, 0(r1)
muld f0, f0, f2  M1 M2 M3 M4 M5 M6 WC
                  M1 M2 M3 M4 M5 M6 WC
                  M1 M2 M3 M4 M5 M6 WC
sw 0(r1), f0
addi r1, r1, #8
sub r2, r1, r3
bnez r2, LOOP

```

Problem 2: When the MIPS program below starts register `$t0` holds the address of a string, the program converts the string to upper case.

(For MIPS documentation see <http://www.ece.lsu.edu/ee4720/mips32v1.pdf> and <http://www.ece.lsu.edu/ee4720/mips32v2.pdf>. Here are the relevant differences with DLX: branches and jumps are delayed (1 cycle). Some branch instructions compare two registers. Register `$0` works like DLX `r0`.)

LOOP:

```
lbu $t1, 0($t0)
addi $t0, $t0, 1
beq $t1, $0, DONE
slti $t2, $t1, 97 # < 'a'
bne $t2, $0 LOOP
slti $t2, $t1, 123 # 'z' + 1
beq $t2, $0, LOOP
addi $t1, $t1, -32
j LOOP
sb $t1, -1($t0)
```

DONE:

Convert the program to IA-64 assembly language using predicated instructions. (You're not expected to know it at this point.) IA-64 is described in the IA-64 Application Developer's Architecture Guide, available at <http://www.ece.lsu.edu/ee4720/ia-64.pdf>.

For this problem one can ignore a lot of IA-64's features. Here is what you will need to know: IA-64 has 64 1-bit predicate registers, `p0` to `p63`, which are written by `cmp` (compare) and other instructions. Predicates can be specified for most instructions, including `cmp` itself. See 11.2.2 for a description of how to use IA-64 predicates.

To solve the problem look at the following sections: 9.3, 9.3.1, and 9.3.2 (a brief description of where to place stops); 11.2.2 (predicate description and some more information on stops); and Chapter 7 (for instruction descriptions). The following instructions will be needed: `cmp` (compare, look at the normal [none] type) `br` (branch), load, store, and add.

- Use general-purpose registers `r0-r31` and predicate registers `p1-p63` in your solution. (There are 128 general-purpose registers, but those above `r31` must be allocated.)
- Minimize the number of instructions per iteration assuming about half the characters are lower case.
- Use predicates to eliminate some branches.
- Make use of post-increment loads or stores.
- Pay attention to data type sizes.
- Show stops but do not show bundle boundaries.

Solution on next page.

```
// Solution
```

```
LOOP:
```

```
    ld1      r1 = [r2];;
    cmp.eq   p3,p4 = r0,r1
    cmp.le   p1,p2 = 97,r1;; // p1 = r1 >= 97;   p2 = !p1 = r1 < 97
(p1) cmp.ge  p1,p2 = 122,r1;; // p1 = r1 <= 122;  p2 = r1 > 122
(p1) add     r1 = -32, r1;;
(p4) st1     [r2],1 = r1
(p4) br     LOOP;;
    br      DONE
```