LSU EE 4720 **Homework 1** Solution **Due: 3 March 2010**

Problem 1: Re-write each code fragment below so that it uses fewer instructions (but still does the same thing). *Note: In the original assignment the branch instruction was* `blt r1, r0 TARG`.

```
# Fragment 1
lw r1, 0(r2)
addi r2, r2, 4
lw r3, 0(r2)
addi r2, r2, 4

#
# SOLUTION
lw r1, 0(r2)
lw r3, 4(r2)
addi r2, r2, 8
```

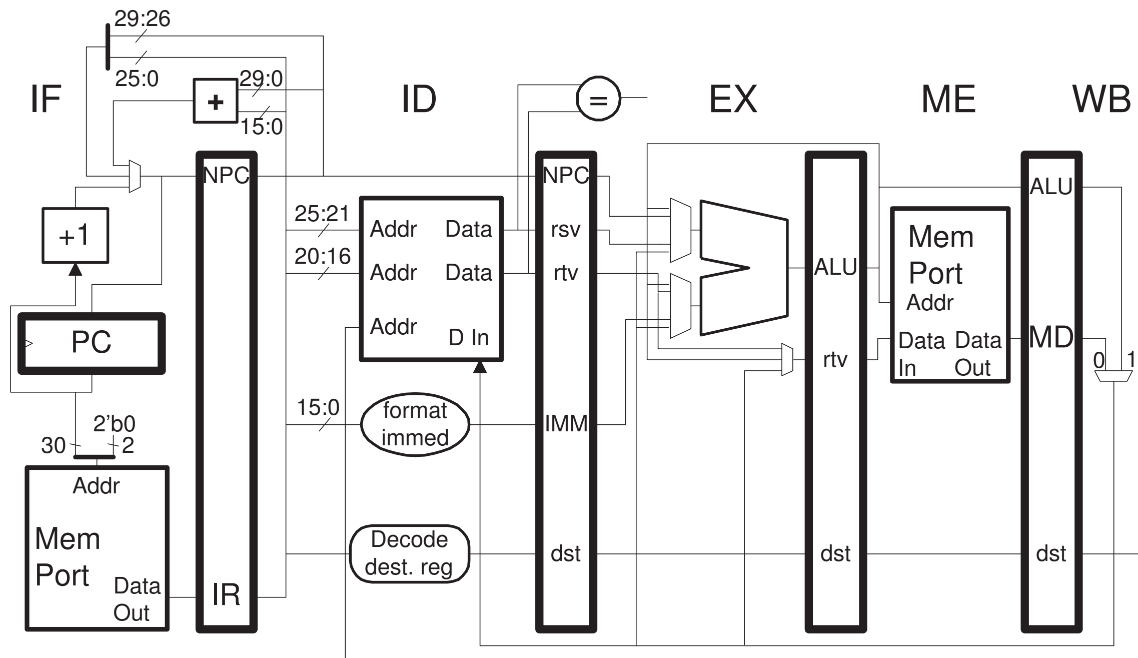
```
# Fragment 2
sub r1, r2, r3
bne r1, r0  TARG
add r1, r5, r6

#
# SOLUTION
bne r2, r3  TARG
add r1, r5, r6
```

```
# Fragment 3
ori r1, r0, 0x1234
sll r1, r1, 16
ori r1, r1, 0x5678

#
# SOLUTION
lui r1, 0x1234
ori r1, r1, 0x5678
```

Problem 2: The MIPS code below runs on the illustrated implementation. Assume that the number of iterations is very large.




```
LOOP:
 lw r3, 0(r1)
 addi r2, r2, 1
 beq r3, r4 LOOP
 lw r1, 4(r1)
```

(*a*) Show a pipeline execution diagram with enough iterations to determine the CPI.

Diagram shown below. To determine the CPI we need a repeating pattern of iterations. An iteration begins when the first instruction of the loop is in IF, the first, second, and third iterations begin in cycle 0, 5, and 11, respectively. At the beginning of the first iteration only `lw r3` is in the pipeline, at the beginning of the second iteration `lw r3` is in IF, `lw r1` is in ID, etc. (look at the stages directly above the IF in cycle 5). So the pipeline state is different at the beginning of the first and second iterations. At the beginning of the third iteration, in cycle 11, the pipeline contents (state) is the same as the beginning of the second. Therefore we expect the pattern to repeat and so can determine the CPI using the second iteration.

```
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r3, 0(r1)      IF ID EX ME WB
 addi r2, r2, 1       IF ID EX ME WB
 beq r3, r4 LOOP         IF ID -> EX ME WB
 lw r1, 4(r1)               IF -> ID EX ME WB
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r3, 0(r1)                     IF ID -> EX ME WB
 addi r2, r2, 1                      IF -> ID EX ME WB
 beq r3, r4 LOOP                           IF ID -> EX ME WB
 lw r1, 4(r1)                                 IF -> ID EX ME WB
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r3, 0(r1)                                       IF ID -> EX ME WB
```

(*b*) Determine the CPI.

The number of cycles is $11 - 5 = 6$ (the difference between the start times of the second and third iterations), so the CPI is $\frac{6}{4}$ CPI.

(*c*) Schedule (re-arrange) the code to remove as many stalls as possible.

There are two solutions below. The first removes one of the two stalls, the second code fragment removes both stalls. The first code fragment loads exactly the same items as the original code, but the second one loads an extra `0(r1)`, which can possibly result in loading an illegal memory address. Either solution would get full credit.

```
LOOP:             # Solution 1, still has 1 stall.
 lw r3, 0(r1)
 lw r1, 4(r1)
 beq r3, r4 LOOP
 addi r2, r2, 1

 #                  Solution 2, no stalls, but risks bad addr on last iter.
 lw r3, 0(r1)
 bne r3, r4 DONE
 nop
LOOP:
 lw r1, 4(r1)     IF ID EX ME WB
 addi r2, r2, 1      IF ID EX ME WB
 beq r3, r4 LOOP        IF ID EX ME WB
 lw r3, 0(r1)              IF ID EX ME WB
DONE:
```

Problem 3: The MIPS implementation from the previous problem has three multiplexors in the EX stage.

(a) Write a program that executes without stalls and which uses the eight ALU multiplexer inputs in order (perhaps starting at cycle 3) in consecutive cycles. That is, in cycle 3 the top input of the upper ALU mux would be used, (bypass from memory), in cycle 4 the second one would be used (NPC), in cycle 5 rsv, in cycle 6 bypass from WB, in cycle 7 we switch to the lower ALU mux with the bypass from ME input, in cycle 8 rtv, etc.

Solution shown below. Note that the `jal` instruction writes register `r31`.

```
# Bypass                 Upper-Mux-- Lower-Mux--
# Bypass                 ME NP RS WB ME RT IM WB
# Cycle         0  1  2  3  4  5  6  7  8  9  10 11 12
add r1, r2, r3  IF ID EX ME WB
add r4, r1, r5     IF ID EX ME WB
jal                   IF ID EX ME WB
add r6, r7, r4           IF ID EX ME WB
add r1, r31, r8             IF ID EX ME WB
add r10, r11, r1               IF ID EX ME WB
add r12, r13, r14                 IF ID EX ME WB
addi r15, r16, 123                   IF ID EX ME WB
add r17, r18, r12                       IF ID EX ME WB
# Cycle         0  1  2  3  4  5  6  7  8  9  10 11 12
# Bypass                 ME NP RS WB ME RT IM WB
# Bypass                 Upper-Mux-- Lower-Mux--
```

(b) Explain why it would be impossible to use the EX-stage rtv mux inputs in order in consecutive cycles.

The middle rtv mux input is the bypass from the memory stage. For that input to be used the immediately preceding instruction would have to write a register, which stores don't do. Therefore it is impossible. If they didn't have to be in order (but still consecutive) then it would be easy, see the code below.

```
# Code using all the RTV inputs in consecutive cycles but not in order.

# Bypass                  RT ME WB  <- Mux inputs, in order.
# Bypass                  ME WB RT  <- Mux inputs, but not in order.
# Cycle          0  1  2  3  4  5  6  7
add r1, r3, r4   IF ID EX ME WB
sw r1, 0(r2)        IF ID EX ME WB
sw r1, 4(r2)           IF ID EX ME WB
sw r1, 8(r2)              IF ID EX ME WB
# Cycle          0  1  2  3  4  5  6  7
```