LSU EE 4720 **Homework 3** Solution **Due: 19 March 2003**

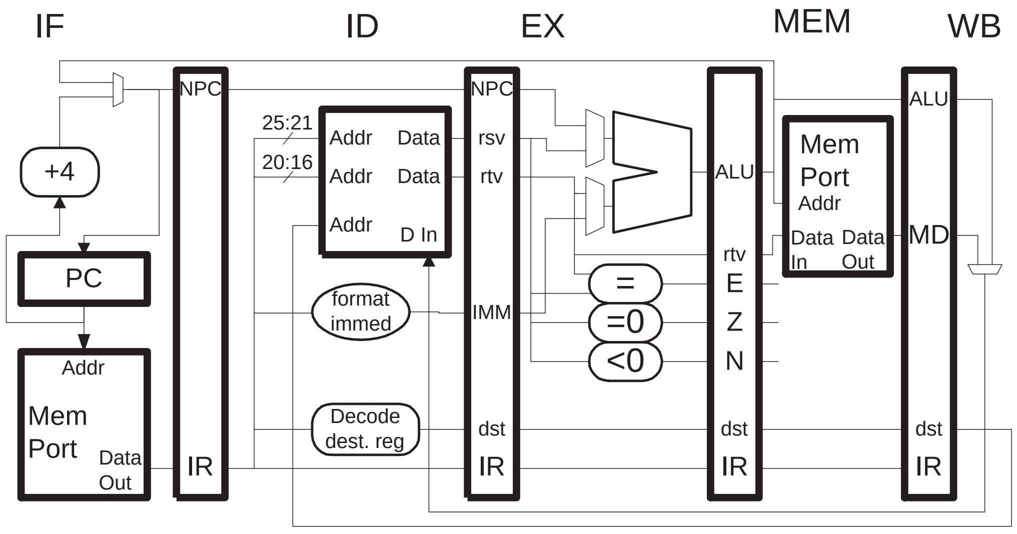
Problem 1: Consider the code below.

```
# Cycle               0  1
add $t1, $t2, $t3     IF ID
sub $t4, $t5, $t1
lw  $t6, 4($t1)
sw  0($t4), $t6
```

(*a*) Show a pipeline execution diagram for the code running on the following illustration. Note that the `add` is fetched in cycle zero.

- Take great care in determining the number of stall cycles.

```
# Solution
#
# Cycle                 0  1  2  3  4  5  6  7  8  9  10 11
add $t1, $t2, $t3       IF ID EX ME WB
sub $t4, $t5, $t1          IF ID ----> EX ME WB
lw  $t6, 4($t1)               IF ----> ID EX ME WB
sw  0($t4), $t6                        IF ID ----> EX ME WB
```



Problem 2: The code below is the same as in the previous problem.

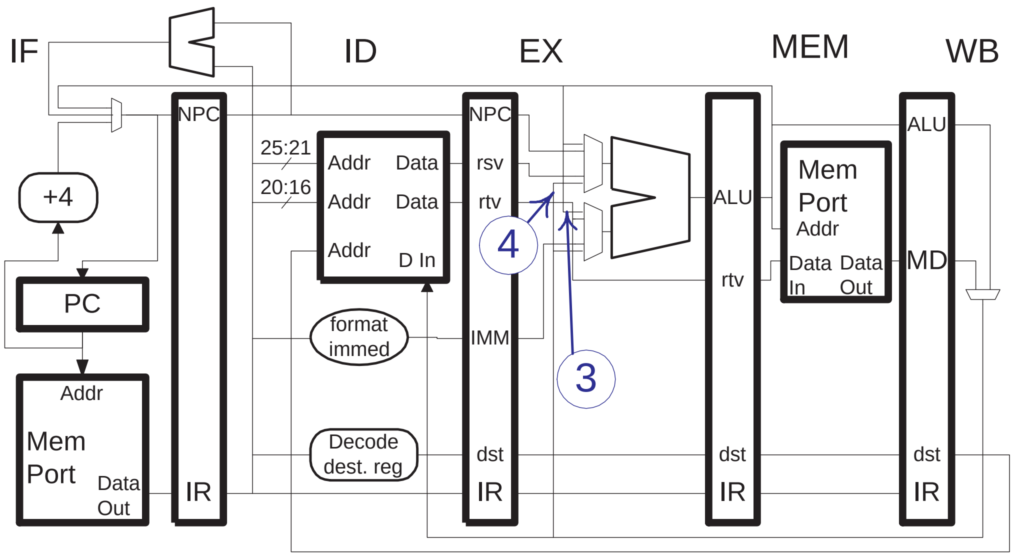
```
# Cycle              0  1
add $t1, $t2, $t3    IF ID
sub $t4, $t5, $t1
lw  $t6, 4($t1)
sw  0($t4), $t6
```

(*a*) Show a pipeline execution diagram (PED) of the code running on the system below.

```
# Cycle              0    1    2    3    4    5    6    7    8    9
add $t1, $t2, $t3    IF   ID   EX  (ME) (WB)
sub $t4, $t5, $t1         IF   ID   EX   ME   WB
lw  $t6, 4($t1)                IF   ID   EX   ME   WB
sw  0($t4), $t6                     IF   ID   ------>   EX   ME   WB
```

(*b*) In the PED circle each stage that *sends* a bypassed value. In the diagram label each bypass path with the cycle in which it is used. To avoid ambiguity, label the end of the path (at the mux input).

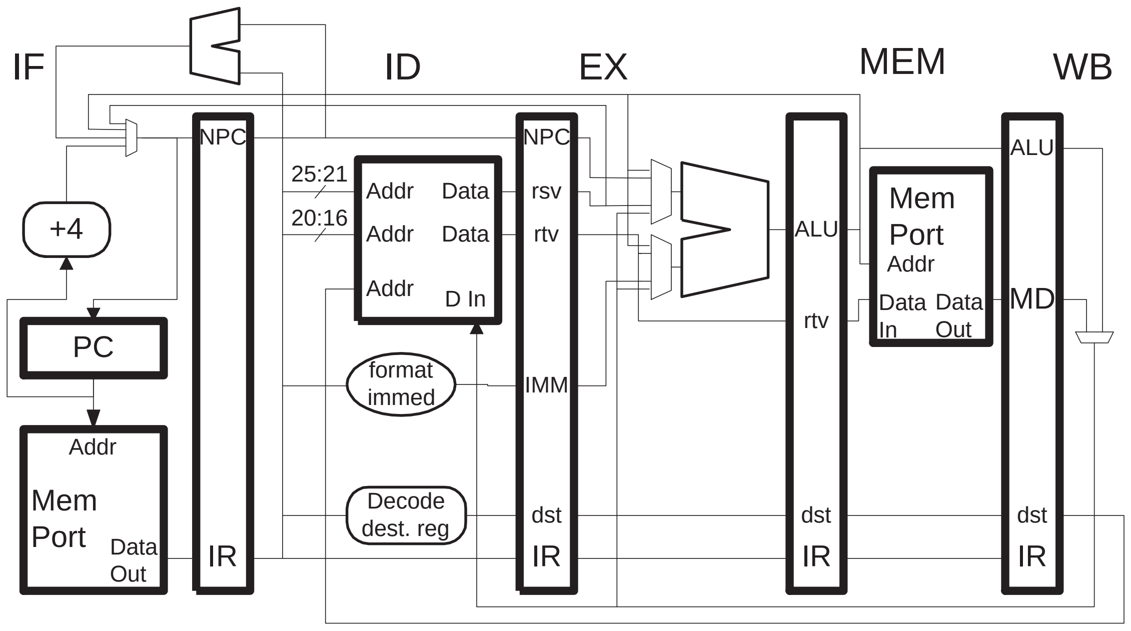
In the PED parenthesis are used instead of circles.



The problem below is tricky. If necessary use Spring 2001 Homework 2 problem 3 for practice.

Problem 3: The program below has an infinite loop and runs on the bypassed implementation below.

```
        # Initially $t0 = LOOP (address of jalr)
LOOP:
        jalr $t0
        addi $t0, $ra, -4
        bne $t0, $0 LOOP
        addi $t0, $t0, -4
```



(*a*) Show a pipeline execution diagram for this program up to a point at which a pattern starts repeating. Beware, the loop is tricky! Read the fine print below for hints.

Note that `jalr` reads and writes a register. The `jalr` instruction should be fetched twice per repeating pattern. The `addi` instruction should be fetched three times per repeating pattern.

```
# Code in dynamic order. (Same four static instructions repeated.)
#
# 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
jalr $t0            IF ID EX(ME)WB
addi $t0, $ra, -4      IF ID EX ME WB
bne $t0, $0 LOOP          IFx
addi $t0, $t0, -4

jalr $t0                     IF ID -> EX(ME)WB
addi $t0, $ra, -4               IF -> ID EX ME WB
bne $t0, $0 LOOP                      IFx
addi $t0, $t0, -4

jalr $t0
addi $t0, $ra, -4                        IF ID EX ME WB
bne $t0, $0 LOOP                            IF ID ----> EX ME WB
addi $t0, $t0, -4                              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
jalr $t0                                                IF ID ----> EX(ME)WB
addi $t0, $ra, -4                                          IF ----> ID EX ME WB
bne $t0, $0 LOOP                                                    IFx
addi $t0, $t0, -4

jalr $t0                                                               IF ID -> EX(ME)WB
addi $t0, $ra, -4                                                         IF -> ID EX ME WB
bne $t0, $0 LOOP                                                                IFx
addi $t0, $t0, -4
# 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*) In the PED circle each stage that *sends* a bypassed value. In the diagram label each bypass path with the cycle in which it is used. To avoid ambiguity, label the end of the path (at the mux input).

(*c*) Determine the CPI for a large number of iterations.

Iteration types:

First: A: Starts at cycle 0, no other loop instruction in pipeline.

Second: B: Starts at cycle 3, pipeline contents: jalr in ME, first addi in EX.

Third: C: Starts at cycle 7 with addi, pipeline contents: jalr in ME, addi in EX.

Fourth: D: Starts at cycle 12, pipeline contents: bne in EX, second addi in ID.

Fifth: B: Starts at cycle 17, pipeline contents: jalr in ME, first addi in EX.

Because state of pipeline at the beginning of second and fifth iterations are identical and because `t0` has the same values at those iterations, the pattern BCD will repeat. (The entire loop: ABCDBCDBCD...) The number of cycles in this three-iteration set is $17 - 3 = 14$ and the number of instructions is 7 and so the CPI is $\frac{14}{7} = 2$.

Problem 4: SPARC V9 has multiple floating-point condition code (FCC) registers. See the references pages for more information on SPARC V8 and V9.

(*a*) Write a program that uses multiple FCC's in a way that reduces program size. As an example, the SPARC program below uses a single FCC. (To solve this problem first find instructions that set and use the multiple FCC registers in the SPARC V9 Architecture Manual. Then write a program that needs the result of one comparison (say, $a < b$) several times while also using the result of another (say, $c > d$). A program not using multiple condition code registers should have to do the comparison multiple times whereas the program you write does each comparison once.)

The solution appears below. Note that it is possible to re-cast the code so that on a system with one FCC only one of each comparison is done. The point is to demonstrate use of the registers.

```
# Solution
#
  fcmpd %fcc0, %f0, %f2
  fcmpd %fcc1, %f4, %f6
  fbg %fcc0, SKIP1
  nop
  faddd %f10, %f10, %f14
SKIP1:
  fbg %fcc1, SKIP2
  nop
  fdivd %f10, %f10, %f12
SKIP2:
  fbg %fcc0, SKIP3
  nop
  faddd %f10, %f10, %f16
SKIP3:
```

(*b*) SPARC V9 is the successor to SPARC V8, which has only one FCC register. (SPARC V9 implementations can run SPARC V8 code.) Did the addition of multiple FCC's require the addition of new instructions or the extension of existing instructions? Answer the question by citing the old and new instructions and details of their coding.

Yes and no.

Yes, the SPARC V9 floating-point compare instructions (**fcmpd**, etc) are extensions of SPARC V9 instructions. (They have the same opcodes, the only difference is that the V9 version uses two bits of the rd field (bits 29-25) to specify the condition code register.)

No, the SPARC V9 floating-point branch instructions that can specify an FCC are different than the SPARC V8 branch instructions. (They have a different opcode.)

(*c*) Do you think the designers of SPARC V8 planned for multiple FCC's in a future version of the ISA?

Probably not, otherwise the V8 branch instructions would have bits reserved for a condition code register number (with instructions to set them to zero). It would take two bits away from the offset, but a 20-bit offset can still span over a million instructions, enough for a vast majority of branches.