EE 4720 **Homework 3** Solution **Due: 20 March 2002**

Problem 1: The exception mechanism used in the MIPS 32 ISA differs in some ways from the SPARC V8 mechanism covered in class. See Chapter 7 in
`http://www.ece.lsu.edu/ee4720/sam.pdf` for the SPARC V8 exception information and
`http://www.ece.lsu.edu/ee4720/mips32v3.pdf` for a description of the MIPS mechanism. The MIPS description is a bit dense, so start early and ask for help if needed.

(*a*) Describe how the methods used to determine which exception was raised differ in SPARC V8 and MIPS 32. Use an illegal (reserved) instruction error as an example. Shorter answers will get more credit so concentrate on explaining how the processor identifies the exception (was it an illegal instruction, an arithmetic overflow, etc) and avoid irrelevant details. For example, details on how the processor switches to system mode is irrelevant.

The way the question should be answered:

The difference is that in SPARC a particular exception causes a particular handler to run, so that if the illegal instruction handler is run it must mean that an illegal instruction exception occurred. In MIPS the handler must read a cause register to find out which exception occurred.

Additional Information:

In both ISAs a number is associated with each kind of exception, in SPARC it is called the trap type, and in MIPS it is referred to as an exception type; it will be called an exception code here. The question is asking how the methods used by the handler to determine the exception code differ. In both cases the hardware generates an exception code.

In SPARC V8 the exception code is used to form a trap table entry address; a control transfer is made to this address. At this address is the first four instructions of the handler (first eight in V9). The handler for the illegal instruction exception "knows" an illegal instruction exception occurred because that's the only exception that would cause it to run. (The trap type or an illegal instruction exception is 16.)

MIPS 32 also has an exception table but it has far fewer entries. To determine which exception type caused it to run the MIPS handlers read a *cause* register which contains the exception type.

(*b*) Where do the two ISAs store the address of the faulting instruction? Both ISAs have delayed branches, so why does SPARC store two return addresses while MIPS gets away with one?

(SPARC registers are organized like a stack, on a procedure call a `save` instruction "pushes" a fresh set of registers on the stack, and a `restore` instruction "pops" the registers, returning to the previous set. The set of visible registers is called a window. This mechanism reduces the need to save and restore registers in memory. This piece of information is needed for the previous problem.)

In SPARC V8 an exception will cause the current window pointer to advance, saving the interrupted code's registers and providing a fresh set of registers to the handler. The PC and NPC of the faulting instruction will be stored in registers `l1` and `l2`. In MIPS 32 only the PC is saved, it is saved in a special EPC register. If the faulting instruction is in a branch delay slot the PC *of the branch* is saved, otherwise the PC of the faulting instruction is saved.

Suppose the instruction in a branch delay slot of a taken branch raises an exception and is to be re-executed. In MIPS 32 control returns to the branch before the instruction so both the branch and the faulting instruction re-execute. (Since the instruction before the faulting instruction re-executes this is not a precise exception by the definition given in class. Since the branch does not modify registers [other than PC] or memory it can be used in the same way a precise exception is used, and so in MIPS 32 such exceptions are called precise.) Control is returned to the branch using something like an ordinary jump instruction, except that the processor switches back to user mode. Jumping directly to the faulting instruction would be a challenge because after the faulting instruction is executed the branch target needs to be executed. MIPS has no way to do these kinds of jumps and so there is no need to store NPC.

SPARC on the other hand can return directly to an instruction in the delay slot. It does so using two consecutive control transfer instructions, something forbidden in MIPS. A `jmpl` instruction jumps to the saved PC, a `rett` (return from trap) instruction jumps to the NPC.

To summarize, SPARC saves two addresses because it needs both of them to restart an instruction in a branch delay slot. MIPS stores only one because it never returns from exceptions to a branch in a delay slot, instead it re-executes the branch.

Problem 2: The pipeline execution diagram below is for code running on a MIPS implementation developed just for this homework problem! Note that the program itself is missing. The dog deleted it. The `M_` and `A_` refer to parts of the multiply and add functional units with segment numbers omitted for this problem. A `WBx` indicates that an instruction does not write back to avoid a WAW hazard.

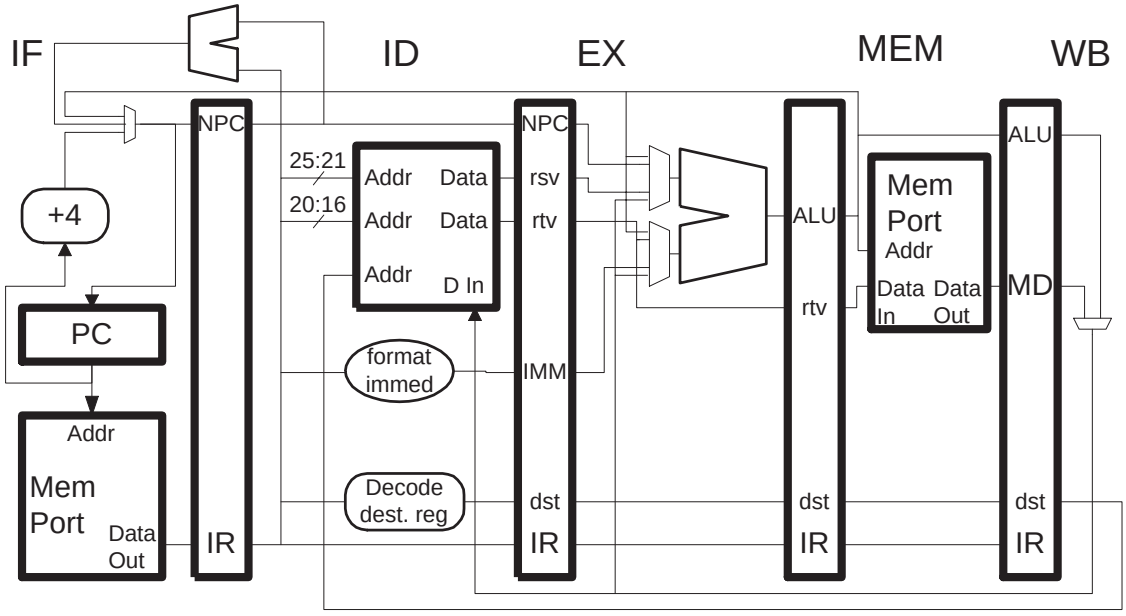
```
IF ID M_ M_ M_ M_ M_ M_ WB
   IF ID ----> M_ M_ M_ M_ M_ M_ WB
      IF ----> ID ----> A_ A_ WB
               IF ----> ID M_ M_ M_ M_ M_ M_ WBx
                        IF ID A_ A_ WB
                           IF ID A_ A_ WB
```

(*a*) Write a program consistent with the diagram. Pay attention to dependencies.

```
# Solution
mul.d f0, f2, f4         IF ID M1 M1 M1 M2 M2 M2 WB
mul.d f6, f8, f10           IF ID ----> M1 M1 M1 M2 M2 M2 WB
add.d f12, f0, f14             IF ----> ID ----> A1 A2 WB
mul.d f16, f18, f20                     IF ----> ID M1 M1 M1 M1 M1 M1 WBx
add.d f16, f22, f24                              IF ID A1 A2 WB
add.d f26, f28, f30                                 IF ID A1 A2 WB
```

(*b*) Identify the latency and initiation interval of the functional units. Fill in the segment numbers.
Multiply: latency, 5; initiation interval, 3. Add: latency, 1; initiation interval, 1.

Problem 3: In the MIPS implementation below (also shown in class) branches are resolved in the ID stage. Resolution of a branch direction (determining whether it was taken) must wait for register values to be retrieved and, for some branches, compared to each other. Suppose this takes too long.



(*a*) Show the modifications needed to do the equality comparison in the EX stage. The modified hardware must use as little additional hardware as possible and, to maximize performance, should only do an EX-stage equality comparison when necessary. Don't forget about branch target address handling. *Hint: The modifications are easy.*

The ID-stage adder that computes branch displacements is also connected to a new ID/EX latch, the output of this new latch is connected to the PC multiplexor. The ALU in the EX stage does the register comparison for the branch. Note that the only added hardware is the latch and the new paths. (A diagram may be added to this solution at some point.)

(*b*) Write a code fragment that runs differently on the two implementations and show pipeline execution diagrams for the code on the two implementations.

```
# Solution

 # Execution on original system.
 #
 # Cycle          0  1  2  3  4  5  6
 beq $2, $3 TARG  IF ID EX ME WB
 add $4, $5, $6      IF ID EX ME WB
 #...
TARG:
 xor $6, $7             IF ID

 # Execution on modified system.
 #
 # Cycle          0  1  2  3  4  5  6
 beq $2, $3 TARG  IF ID EX ME WB
 add $4, $5, $6      IF ID EX ME WB
 sub $7, $8, $9         IFx
```

```
 #...
TARG:
 xor $6, $7                IF ID
```

(*c*) The table below lists SPARC instructions and indicates how frequently they were used when running TEX to prepare this homework assignment. (Many rows were omitted to save space, so the "%exec" column will not add to 100%.) Suppose that the instruction percentages are identical for MIPS (which means totally ignoring the `cc` instructions). Assume that SPARC `be` and `be,a` are equivalent to MIPS `beq`, SPARC `bne` and `bne,a` are equivalent to MIPS `bne`, and that the other branch instructions (they begin with a b), are equivalent to branch instructions that compare to zero (`bgez`, etc.).

Suppose the clock frequency of the original design is 1.0000 GHz. Based on the data below and making any necessary assumptions, for what clock frequency would the new design run a program in the same amount of time as the old one? What column would you add (what additional data do you need) to the table to make your answer more precise?

Assume that floating-point instructions are insignificant and that there are no stalls due to memory access.

opcode	#exec	%exec
subcc	4659360	12.6187%
lduw	4521722	12.2459%
add	4159629	11.2653%
or	3110542	8.4241%
sethi	3066797	8.3056%
stw	1848293	5.0056%
sll	1402122	3.7973%
be	1393475	3.7739%
jmpl	1140223	3.0880%
call	1088068	2.9467%
ldub	1064918	2.8841%
bne	936493	2.5362%
stb	687981	1.8632%
srl	609402	1.6504%
save	526477	1.4258%
restore	526474	1.4258%
bne,a	453545	1.2283%
nop	433253	1.1734%
bge	429978	1.1645%
ldsb	429497	1.1632%
orcc	382947	1.0371%
and	370967	1.0047%
be,a	360057	0.9751%
sub	354847	0.9610%
ba	321970	0.8720%
bl	297715	0.8063%
andcc	270465	0.7325%
bgu	235304	0.6373%
bl,a	216074	0.5852%
sra	204610	0.5541%
ble	198154	0.5366%
xor	185137	0.5014%
bcs	182153	0.4933%
addcc	155156	0.4202%
bleu	142755	0.3866%
bg	117582	0.3184%
mulscc	88681	0.2402%

In the new design there will be a bubble added for taken branches that compare two registers. Assume the original system has a CPI of 1 and that half the branches are taken. The percentage of branches that add a bubble is found by adding the percentages for **be**, **bne**, **bne,a**, and **be,a**: and dividing by two: $\frac{8.5135\%}{2} = 4.25675\%$. The new CPI will then be 1.0425675. To find the clock frequency of the new system for which it will run as fast as the old system solve: $\frac{1.0}{\phi_{\text{old}}} = \frac{1.0425675}{\phi_{\text{new}}}$ for ϕ_{new} to get $\phi_{\text{new}} = 1.0425675\,\text{GHz}$, where $\phi_{\text{old}} = 1\,\text{GHz}$.

To make the answer more precise two things are needed, the CPI on the original system and the fraction of times a branch is taken.