LSU EE 4720 Homework 4 Solution Due: 31 March 2003

Problem 1: The two code fragments below call trap number 7. How do the respective handlers determine that trap 7 was called?

```
! SPARC V8
ta %g0,7

# MIPS
teq $0, $0, 7
```

In SPARC V8 each trap number has its own handler routine, the first four instructions of which are in the trap table. Since a particular handler routine is only called for a particular exception number, there is no need for the handler to determine which exception occurred. That is, if the trap 7 handler is running trap 7 must have occurred.

In MIPS the only way for the hander routine to get the trap number is to load the trap instruction itself and look at the field holding the code, bits 15:6. The handler can get the address of the instruction from the EPC register.

Problem 2: There is a difference between the software emulation of unimplemented SPARC V8 instructions triggered by an illegal opcode exception, such as `faddq`, and Alpha's use of PALcode for certain instructions. (See the respective ISA manuals on the references Web page. For SPARC, see Appendix G, it should not be difficult to find the PALcode information for Alpha.)

(*a*) What is similar about the two?

In both cases a single instruction in a program can trigger something like a subroutine that has privileged access to machine state.

(*b*) What is the difference between the kinds of instructions emulated using the two techniques? Why would it not make sense to use PALcode for quad-precision arithmetic instructions?

PALCode instructions are intended for functions that are too complex for a single RISC instruction and which vary from machine to machine (because the way the function is coded depends upon the underlying hardware). Among other things, PALCode instructions are used the way trap instructions are used in other ISAs, to perform system calls.

A PALCode instruction has a particular opcode, and an immediate operand specifying which PAL routine to execute. The PALCode instructions are used something like trap instructions, in which a trap code is specified in the instruction and operands are placed in fixed registers.

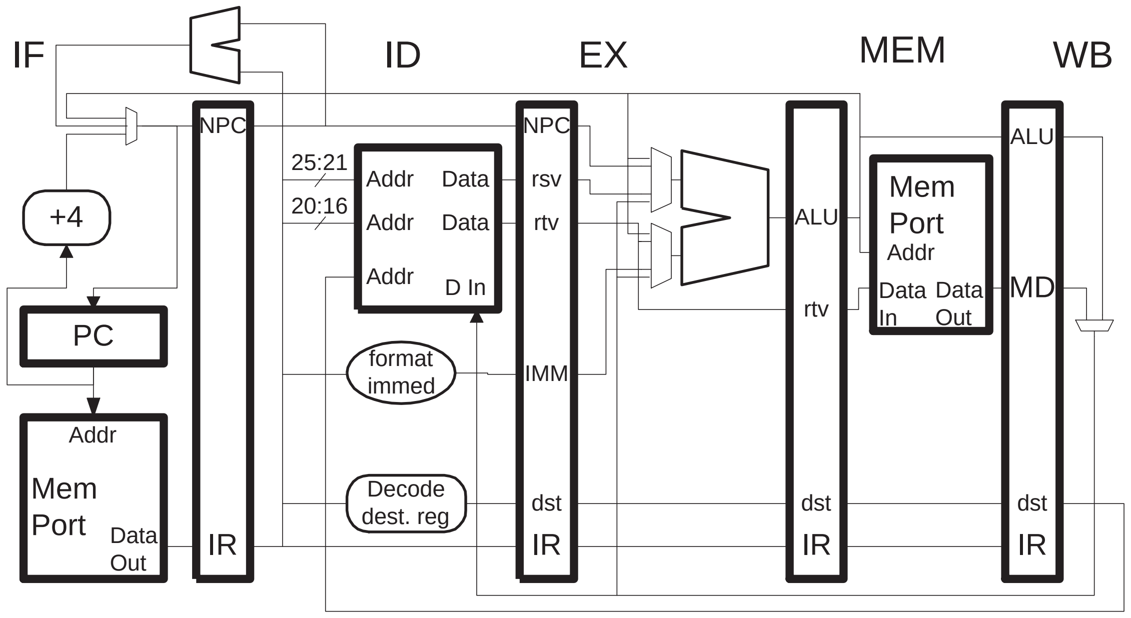
An illegal opcode exception can be raised by any instruction that the implementation does not recognize. If such an instruction is defined in the ISA the handler could emulate it, putting the correct result in the destination register. Illegal opcodes exceptions can be used to emulate instructions that would require alot of hardware and are expected to be rarely used in the implementation.

It would not make sense to emulate quad precision instructions with PALCode because they would not look like other arithmetic instructions and so would be awkward to use. In particular, source operands would have to be placed in fixed registers, say `f0` and `f2` and destinations would be written to another fixed register, say `f4`. PALCode instructions are always intended for software emulation, and so in an implementation that had quad precision hardware the PALCode would be called anyway. (It could use the new instruction to do the arithmetic, but it would not be quick as just having a quad precision instruction.)

Quad precision instructions emulated using illegal instruction exceptions look like normal instructions, for example, the source operands can come from any register (perhaps the register number must be a multiple of 4). If an implementation does have quad-precision hardware, the instructions execute normally.

Problem 3: In both SPARC and MIPS each trap table entry contains the first few instructions of the respective trap handler. On some ISAs a *vector table* is used instead, each vector table entry holds the **address** of the respective handler.

Why would the use of a vector table (rather than a trap table) be difficult for the MIPS implementation below?



When an exception occurs the processor must branch to the handler routine. With a trap table the address of the handler routine can be determined by combining the trap base register (SPARC) or a fixed address (MIPS) with the exception code, this requires little or no hardware. If a vector table were used the address of the trap handler would have to be read from memory. First, the address of the vector table entry (holding the handler address) would need to be computed, that can also be done easily. Next the vector table entry must be read from memory. That would require the use of the memory stage which would complicate things because (1) the memory stage is not being used to execute an ordinary instruction (complicating control), (2) a new path must be added for sending the vector table address to the memory address input, and (3) a path must be added from the memory output port to the PC input. All of this can be done of course, but at best it might save only a few cycles from a rarely occurring event.

Problem 4: One way of implementing a vector table interrupt system on the MIPS implementation above would be by injecting hardware-generated instructions into the pipeline to initiate the handler. These instructions would be existing ISA instructions or new instructions similar to existing instructions.

What sort of instructions would be injected and how would they be generated? Show changes needed to the hardware, including the injection of instructions. In the hardware diagram the instructions can be generated by a magic cloud [tm] but the cloud must have all the inputs for information it needs.

Include a program and pipeline execution diagram to show how your scheme works.

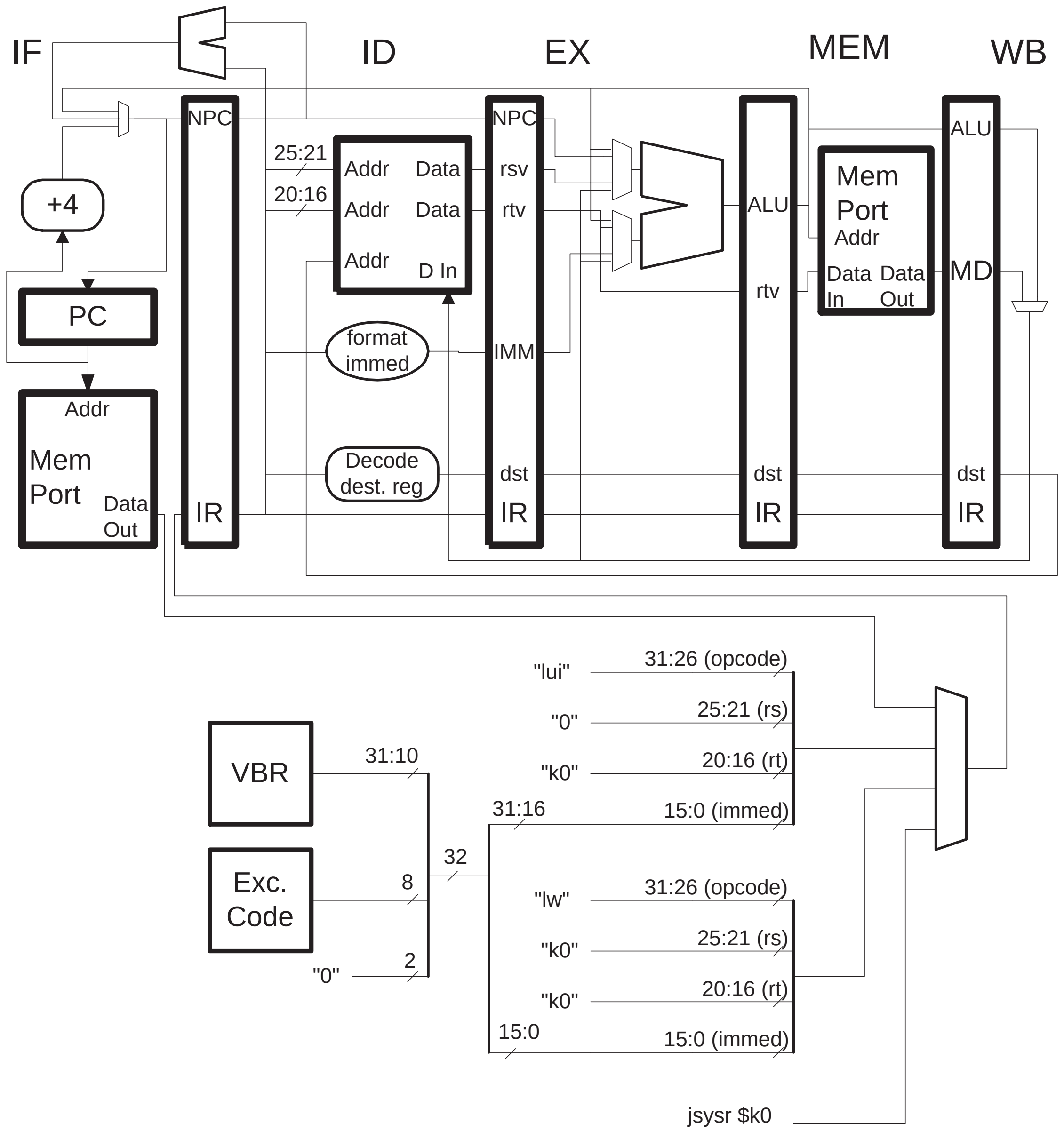
- Assume an exception code is available in the MEM stage.
- Include a vector base register, (VBR), which holds the address of the first table entry.

The solution is on the next page.

The solution appears below. Two new registers, **VBR** (vector table base register) and `Exc Code` (exception code) are shown. The inputs to these registers are omitted for clarity. Register **VBR** is loaded by a system instruction (not shown or discussed further) while `Exc Code` is loaded by the hardware when an instruction raising an exception passes through the MEM stage. The hardware injects three instructions, `lui`, `lw`, and `jsysr`. Instruction `jsysr` is new, it jumps to the address in its operand register and switches the processor to system mode. It does not have a delay slot. The code uses register `k0` which ordinary code must not use.

Normally the top input of the multiplexor is used. When an exception occurs the other inputs are used in sequence, injecting the three instructions. The immediate portion of the `lui` and `lw` instructions are inserted by the hardware, based on the contents of the VBR and `Exc. Code` registers.

A pipeline execution diagram is shown below. Notice that the injected instructions do not use IF.



```
# Solution, Continued

# Cycle             0  1  2  3  4
# Part of program
 ant $s0, $s1, $s2  IF ID*EX ME WB
 or  $t0, $t1, $t2     IF ID EXx
 xori $t3, $t3, 1         IF IDx
 andi $s3, $s3, 7            IFx

# Injected by hardware. Assumed exception code is 1.
 lui $k0, 0x1234                ID EX ME WB
 lw  $k0, 0x5404                   ID EX ME WB
 jsysr $k0                            ID EX ME WB

# Handler
 lui $k0 $0x9000                                 IF ID
 sw  0($k0), $r1                                    IF
 ...
```