Name Solution

Computer Architecture
EE 4720
Midterm Examination 1.0.4

Friday, 22 October 2004, 10:40–11:30 CDT

Problem 1		(30 pts)
Problem 2		(15 pts)
Problem 3		(25 pts)
Problem 4		(30 pts)
Exam Total		(100 pts)

Alias Titan at last!!!

Good Luck!

Problem 1: The routine below is to do a binary search on an array and return 1 if the item is found, 0 otherwise. Complete the MIPS program using the C++ code as a guide. [30 pts]

☑ The only allowable synthetic instruction is `nop`.

☑ Fill as many delay slots as possible. The order of the assembly code **does not** have to match the order of the C code.

```
#  unsigned int delta = size;  int pos = 0;
#  while( true ) {
#      delta = delta >> 1;
#      int next_pos = pos + delta;
#      int e = array[next_pos];
#      if( e == target ) return 1;
#      if( !delta ) return 0;
#      if( e < target ) pos = next_pos;
#    }
        # CALL VALUE: $a0,  array: Address of array. Each element is a WORD-SIZED int.
        # CALL VALUE: $a1,  size: Number of elements in array. A power of 2.
        # CALL VALUE: $a2,  target: Value to find.
        # RETURN VALUE: $v0,  1, if target in array; 0, if not in array.
        # Can modify registers $a0-a3, $t0-$t9

# Solution
#
# Common problems highlighted.  Solution below shows correct code.
#
bsearch:
        # $a1 is delta
        addi $t0, $0, 0     # pos = 0;
LOOP:
        srl $a1, $a1, 1     # delta = delta >> 1;
        add $t1, $t0, $a1   # next_pos = pos + delta
                            # COMMON PROBLEM: Index not scaled.
        sll $t2, $t1, 2     # Scale array index since elements are four bytes.
        add $t2, $a0, $t2   # t2 = &array[next_pos]
        lw $t2, 0($t2)      # t2 = e, array[next_pos]
        bne $t2, $a2  SKIP1
        nop
        jr $ra              # Element found, return 1.
        addi $v0, $0, 1
SKIP1:
        bne $a1, $0  SKIP2
        nop                 # Delta zero, return 0.
        jr $ra
        addi $v0, $0, 0     # COMMON PROBLEM: Jump delay slots ignored.
SKIP2:
        # if ( element < target ) pos = next_pos
        #
        slt $t3, $t2, $a2
        beq $t3, $0  LOOP
        nop
        j LOOP
        addi $t0, $t1, 0
```

Problem 2: In MIPS and many other ISAs floating-point instructions read and write a FP register file while most other instructions read and write general-purpose (integer) registers. Suppose MIPS-I were modified so that FP instructions used the same registers as the integer instructions. The registers are still 32 bits and can be used in pairs by instructions operating on double-precision operands. The new version of MIPS is not compatible with MIPS-I, don't worry about that.

(*a*) For each MIPS-I instruction below indicate whether it will be retained in the new ISA, "KEEP", or will not be needed, "OMIT." Also add any new instructions that might be needed. If an instruction is kept but operates differently describe the difference. [15 pts]

☑ Indicate KEEP or OMIT.

☑ If omitted show an existing instruction that would be used in its place.

☑ If kept, explain how it would operate differently.

lwc1: OMIT, use lw instead.

ldc1: KEEP, but writes GPRs (call it ld now).

cvt.w.d: KEEP (but operates on the integer registers.) Still need to convert from integer to FP.

cvt.d.w: KEEP (but operates on the integer registers.) Still need to convert from FP to integer.

mfc1: OMIT. No longer an FP reg file to move values from.

mtc1: OMIT. No longer an FP reg file to move values to.

add.s: Keep, but operates on integer registers.

add.d: KEEP, but operates on integer register pairs.

c.lt.s: KEEP, write result (0 or 1) to register.

c.le.d: KEEP, write result (0 or 1) to register.

bc1t: OMIT. Use bne or beq.

Hint: In `bc1t` *b is for branch, t is for true.*

Grading Notes:

Many suggested eliminating `c.lt.s` and using `slt` instead. This will almost work because if one IEEE 754 number is larger than another then its binary encoding interpreted as a two's complement number is larger than the other and so an integer comparison can be performed. Perhaps many had that in mind when they suggested using `slt` (I hope). The problem is that IEEE 754 has several non-number encodings and the comparison instruction is supposed to raise an exception if an attempt is made to compare them, slt won't do that.

The original exam had c.gt.d. There is no such instruction, ooops.

(*b*) If your answer for `bc1t` above was "OMIT" describe below how the instruction can be kept (possibly modified). If your answer was "KEEP" explain how it can be omitted. This will affect other instructions, list them and explain how they are affected.

Instruction `bc1t` branches if the FP condition code register is true. The instruction was omitted above because the comparisons write a general purpose register.

In this part keep `bc1t` and modify the comparison instructions `c.lt.d`, etc. so that they write the FP condition code register instead of an integer register, as they did in the previous part.

Problem 3: Using an interesting technique the program below returns the value of register X, where X is a number in register `$a0`. For example, if `$a0` held an 8 the program would return the value of `$8` (or `$t0` using the register name) in register `$v0`. Consider:

```
        jal copyreg
        addi $a0, $0, 8
        # After copyreg returns $v0 has contents of $t0.

        addi $v0, $t0, 0   # $t0 is $8, register number 8
        # At this point $v0 has contents of $t0.
```

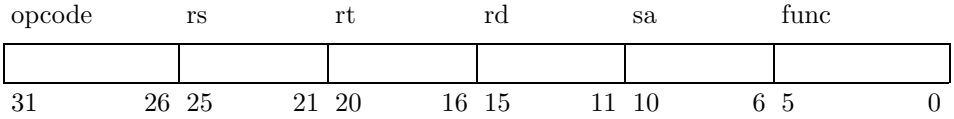
From the code above it looks like the `copyreg` routine is doing things the hard way. The one advantage is that it can copy a source register known only at run time or that might change. In contrast, the `addi` instruction always copies `$t0`. *Note: In the original exam the description and example above were not included.*

(*a*) Modify the program so that it takes a second call value, `a1`, which holds a second register number. The return value is put in that register. For example, if `a0` holds a 12 and `a1` holds a 3 then the routine copies the value in `r12` to `r3`. [17 pts]

☑ Show additions and changes. The added code must use a similar technique to the existing code.

Hint: Can be solved with two added instructions and a few minor modifications.

```
        # Solution.

        # CALL VALUE:   $a0   A register number. (Except v0)
        # CALL VALUE:   $a1   A register number.
        # RETURN VALUE:       Return value in register specified by $a1

copyreg:
        la $v0, template    # Put address of label "template" into $v0.
        lw $v0, 0($v0)      # Load template instruction into $v0.
        sll $a0, $a0, 21    # Shift register number to rs position.
        add $a0, $v0, $a0   # Insert source register number into instruction.

        # Solution (Next two instructions, and modification of template.)

        sll $a1, $a1, 11   # Shift register number to rd position.
        add $a0, $a0, $a0  # Insert destination register into instruction.

        la $v0, mod        # Load address of last instruction in this routine.
        sw $a0, 0($v0)     # Store created instruction at end of this routine.
        jr $ra             # Return ...
mod:
        add $v0, $0, $0    # ... and move registers.

template:
        # add $v0, $0, $0  Original instruction had dest of v0.
        add $0, $0, $0      # In solution make that a $0 since it's modified.
```



Problem 3, continued:

(*b*) Suppose one were considering adding a new machine instruction, `copyreg`, that does the same thing as the routine above (before modification for part 1). Provide arguments for and against adding the new instruction. The arguments can include made-up data as long as it does not conflict with what has been given in this problem or in class. At least one of the arguments must refer to features of the code from part 1. [8 pts]

☑ Provide an argument in favor of adding the instruction.

Programs needing to copy a register known only at run time would work much faster if they had a machine instruction to do it.

Grading Note: Though the routine above performs a load and a store, that's not the way an implementation would do it.

☑ Provide an argument against adding the instruction. Do not simply reverse an assumption made above.

Instructions like this are not needed very often.

Problem 4: Answer each question below.

(*a*) A company is developing an implementation of an ISA. To guide their design they are using a **SUBSET** of the SPECcpu benchmark programs. [7 pts]

Grading Note: A surprising number of students seemed to miss the word subset and answered the question as though it were about the full set of SPEC CPU benchmarks, not a subset.

☑ Describe a situation in which that's a foolish thing to do and explain why it's foolish.

It is foolish if the intended customers run a wide range of programs similar to the full set of benchmarks. By omitting certain programs the company will not be designing their implementation to run those programs fast.

☑ Describe a situation in which that's a smart thing to do.

It's smart if the subset matches what the intended customers to run. This way time is not wasted optimizing a design for programs their customers won't be running.

(*b*) Compiler optimizations are important. [8 pts]

☑ Describe a compiler optimization for which the compiler needs no knowledge of the ISA.

☑ Show an example.

Common subexpression elimination.

```
// Before
if( a+b < c ) x = a + b;

// After (optimization shown in C, but actually done in compiler intermediate
//        representation.)
int s = a + b;
if ( s < c ) x = s;
```

☑ Describe a compiler optimization for which the compiler does need to know the target ISA.

Register assignment. Compiler needs to know how many registers there are, not to mention what are the instructions which need register values in the first place.

(*c*) A goal in the design of MIPS and other RISC ISAs is to minimize the number of registers with specialized purposes. In MIPS `r31`, a.k.a. `$ra`, does have a specialized purpose. [7 pts]

☑ Why was it necessary to make an exception in this case?

The `jal` instruction stores the return address in register `ra`. It is coded in format J which only has an opcode and immediate (`ii`) field, there is no room for a register number. Therefore if there was to be a jump and link with a $26 + 2$-bit region it would have to save the return address in a fixed register.

(*d*) The target in MIPS branch instructions is specified by indicating the number of instructions to skip. [8 pts]

☑ Why couldn't such an approach be used in CISC ISAs?

Because CISC instructions are variable size and so computing a target address using a number of instructions to skip cannot be done by just multiplying that number by the instruction size. The implementation would have to know the sizes of the intervening instructions which would be extremely impractical. Instead branch targets typically specify the number of bytes to skip.

☑ Specifying the number of instructions to skip saves two bits over what is needed for CISC targets. Why is saving two bits not as important in the design of CISC instructions?

Because instructions are variable length their size can be increased to accommodate the size of any needed operands. In the case of a branch the instruction encoding can be made as large as needed to hold the displacement, even if the displacement is two bits larger (which isn't very much anyway) than in most RISC ISAs.