Name Solution

Computer Architecture
EE 4720
Midterm Examination

Wednesday, 22 October 2003, 10:40–11:30 CDT

Problem 1		(25 pts)
Problem 2		(15 pts)
Problem 3		(18 pts)
Problem 4		(18 pts)
Problem 5		(24 pts)
Exam Total		(100 pts)

Alias 5d 5d 5d 5d 5d

Good Luck!

Problem 1: The routine below is called with an unsigned integer in register `$a0` and the address of some allocated memory in register `$a1`. When it returns the memory at should `$a1` contain the hexadecimal representation of `$a0` as a null-terminated (C format) string. Complete the routine, follow the guidelines in the comments. (For **partial credit** write a routine that converts a string holding a hexadecimal number to an integer.) [25 pts]

A solution including test code can be found at `http://www.ece.lsu.edu/ee4720/2003f/mtconv.html`

```
################################################################################
# utoh: Convert unsigned integer to hexadecimal string.
#
        # $a0: Call Value: Unsigned integer to convert.
        # $a1: Call Value: Address of allocated memory.
        #      Write converted string to this address, assume there is enough.
        #      Sample strings: "1F3", "1", "0" written at $a1.

        # [ ] String should not have leading zeros. (Good: "123", Bad "00123".)
        # [ ] Fill as many delay slots as possible.
        # [ ] Registers $a0-$a3 and $t0-$t7 can be modified.
        # The ASCII value of '0' is 48,  the ASCII value of 'A' is 65


        ## Step 1: Count number of leading zeros in hexadecimal representation
        # of $a0 (actually $a0  1 ) and use count to set $t3 to the
        # address of what will be the last character in the string.
        #
utoh:   lui $t1, 0xf000     # Mask used for extracting digits (starting at MSD).
        addi $t3, $a1, 7    # Init $t3 to address of last char of 8-digit number.
        ori $t5, $a0, 1     # OR in a 1 so that when a0 = 0 output is not "".

ILOOP:  and $t0, $t5, $t1   # Extract a digit.
        slti $t4, $t0, 1    # Set $t4 to 1 if digit is zero.
        sub $t3, $t3, $t4   # Adjust end-of-string pointer.
        bne $t4, $0, ILOOP  # Loop if digit is zero.
        srl $t1, $t1, 4     # Shift the mask to the next digit.

        ## Step 2: Convert the number to a string.
        #
        sb $0, 1($t3)       # Null-terminate the yet-to-be-written string.
LOOP:   andi $t0, $a0, 0xf  # Extract LSD (least significant digit).
        slti $t1, $t0, 10   # Check if it will be 0-9 or A-F
        bne $t1, $0, SKIP
        srl $a0, $a0, 4     # Shift in next digit (for next iteration).
        addi $t0, $t0, 7    # Add 7 to digit if it is a letter.
SKIP:   addi $t0, $t0, 48 # Add 48 to digit. (Seven also added if it's a letter.)
        sb $t0, 0($t3)      # Store the ASCII value in string.
        bne $t3, $a1, LOOP  # Loop if we have not written the first character.
        addi $t3, $t3, -1

EXIT:   jr $ra
        nop
```

Problem 2: Code similar to the histogram program presented in class appears below. MIPS instruction formats are shown below for reference. [15 pts]

(*a*) Design three new MIPS instructions to reduce the size of the program fragment below

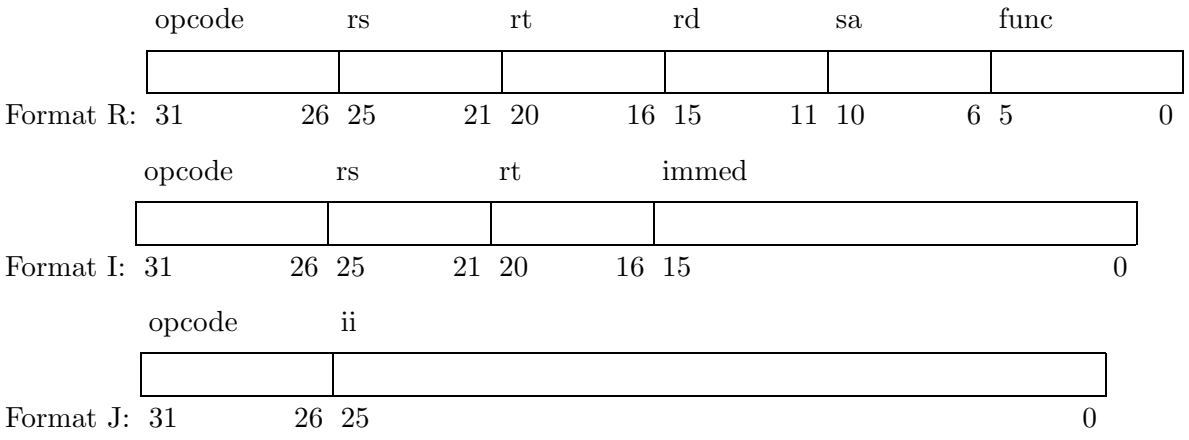
- Each new instruction should replace at least two related instructions in the program below. (Do not combine two *unrelated* instructions, such as `j` and `sw`.)

☑ Show the coding for the new instruction, making up the opcode and other field values as necessary. The coding should use one of MIPS' existing formats and should fit as naturally as possible.

- Do not worry whether the instruction is appropriate for a RISC ISA.

The solution is on the next page.

```
 addi $t7, $0, 26
LOOP:
 lbu $t1, 0($t0)
 addi $t0, $t0, 1
 beq $t1, $0, DONE
 addi $t1, $t1, -65
 sltu $t2, $t1, $t7
 beq $t2, $0, LOOP
 sll $t1, $t1, 2
 add $t3, $a1, $t1
 lw $t4, 0($t3)
 addi $t4, $t4, 1
 j LOOP
 sw $t4, 0($t3)
```



(*b*) Show an instruction that can be created by combining several instructions from the program but that would be impossible to code. Explain why it would be impossible to code.

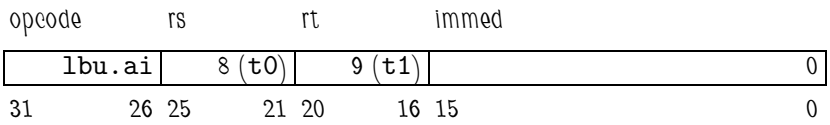
The solution is on the next page.

Solution to part (a).

Combine `lbu` and `addi` to form an autoincrement add, `lbu.ai`. The coding for `lbu` provides all the information needed by `lbu.ai` so the only difference is in the opcode field. (The amount to autoincrement by is the size of the item loaded, in this case one character, it would be two characters for a `lhu.ai`, etc.)

MIPS does not include an autoincrement load because a non-superscalar implementation would either have to write back to two registers simultaneously (an added cost) or else would be subject to stalls. (An n-way superscalar implementation might have to write back to $2n$ register per cycle.) At least one RISC ISA, PA-RISC, has an autoincrement add despite the difficulties.

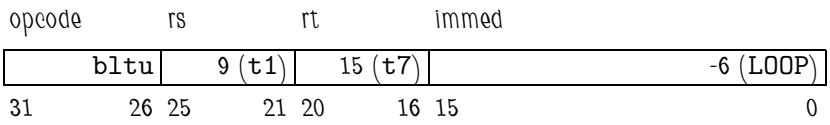
```
Combine:
 lbu $t1, 0($t0)
 addi $t0, $t0, 1
Into:
 lbu $t1, 0($t0)+
```



The `sltu` and `beq` can be combined to create a `bltu` instruction. As with the `lbu.ai` instruction, no new fields are needed, the coding would be the same as the `beq` instruction except for the opcode.

(MIPS does not include such an instruction because the time needed to compare to values would be a bit too long. Some other RISC ISAs do include branch instructions that can compare if one register is less than another, for example, SPARC V9.)

```
Combine:
 sltu $t2, $t1, $t7
 beq $t2, $0, LOOP
Into:
 bltu $t1, $t2, LOOP
```

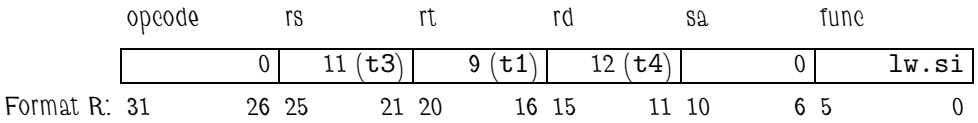


The `sll` and `add` could be combined to make a scaled add, in which the second source operand is shifted by two. But why stop there, the `lw` could also be included to create a scaled indexed load, `lw.si`. The scaled indexed load has three register arguments and so format R must be used. This means that it cannot use a 16-bit displacement. If the scaled indexed load is used only to load word-sized array elements then there would be no need to include the shift amount in the instruction (for the same reason there is no need to include the increment amount in an autoincrement instruction). The sa field might be used for small displacements or it might be used for other shift amounts (for example, loading a word from an array of 64-character structures). Since neither is needed here, the field will be unused and set to zero.

If `lw.si` is used then a value for `t3` is not written and so something must be done for the `sw`. The natural thing to do would be to add a `sw.si` instruction.

(MIPS does not include an instruction like `lw.si` because of implementation complexity. At least one other RISC ISA does, PA-RISC.)

```
Combine:
 sll $t1, $t1, 2
 add $t3, $a1, $t1
 lw $t4, 0($t3)
Into:
 lw.si $t4, ($t3,$t1)
```



Solution to part (b):

An instruction is impossible to code if there is not enough room in the format for all the registers and constant needed. One handy instruction that can't be coded is `bgei`, a branch that compares a register's contents to an immediate. Since the immediate field is also needed for the displacement that would be impossible.

It is possible to add a new format to MIPS in which this instruction can be coded, but the problem stated existing formats must be used. Though it is possible to add a new format, doing so would increase the complexity of implementations because additional paths must be added to move the immediate values to where they are needed.

```
Combine:
 addi $t7, $0, 26
 sltu $t2, $t1, $t7
 beq $t2, $0, LOOP
Into:
 bgei $t1, 26, LOOP
```

Problem 3: Answer each question below.

(*a*) In MIPS (and similar ISAs) there is a `lb`, `lbu` (load byte unsigned), and a `sb` but there is no `sbu` (store byte unsigned). Why not? [6 pts]

Because an unsigned variant is only needed when writing a value to a location that's larger than the value. For `lb` an 8-bit value is written to a 32-bit register, and so the instruction must specify what to do with the other 24 bits (sign extend for `lb`, set to zero for `lbu`). For `sb` an 8-bit value is written to an 8-bit memory location and so there are no extra bits to set.

(*b*) Explain why the MIPS instruction below won't work: [6 pts]

```
add.d $f1, $f2, $f3
```

The instruction uses double-precision floating-point operands, these can only be retrieved from even-numbered registers.

(*c*) Show the result of each add instruction below. The instructions execute on a machine with 32-bit registers that is capable of BCD, and packed integer, and ordinary integer arithmetic. The packed integer operations all use saturating unsigned arithmetic. [6 pts]

```
# r1 = 0x8080888
# r2 = 0x1020999

                                Solution Shown Below
# Ordinary Integer Add
add r3, r1, r2            r3 =  0x90a1221  = 151654945 (decimal)

# BCD Add
add.bcd r3, r1, r2        r3 =  0x9101887  = 9101887 (decimal)

# Packed Integer (4 bits per int.)
add.p4 r3, r1, r2         r3 =  0x90a0fff  = { 9, 0, 10, 0, 15, 15, 15 } (decimal)

# Packed Integer (8 bits per int.)
add.p8 r3, r1, r2         r3 =  0x90a11ff  = { 9, 10, 17, 255 } (decimal)
```

Problem 4: Answer each question below.

(*a*) A company compiles and runs the SPECint2000 benchmarks on its new system, complying with all rules except one: it refuses to divulge the steps it used to compile the programs. Nevermind that it's against the rules, is it in the company's interest to keep this information secret? Explain. [6 pts]

No, because customers would want to know how to compile their programs to run quickly, and it would only help the company it they succeed.

(*b*) A program is compiled two ways, one for ISA A, implementation x, the other also for ISA A, but for implementation y. Explain what would be common to the two executables and what would be different. Provide an example. [6 pts]

The kinds of instructions would be the same since it's the same ISA. The choice of instructions in a particular place and their order would be different.

Grading Note: An executable is the compiled and linked program. Many answered the question as though an executable were a system running a program.

(*c*) Explain the dead-code elimination (DCE) optimization using an example. [6 pts]

A compiler performing the DCE optimization ignores code that writes register values that are never used.

```
Before:
 a = 3;
 a = b + 1;
After:
 a = b + 1;
```

Problem 5: Answer each question below.

(*a*) CISC ISAs have variable length instructions. What advantages over RISC ISAs does that enable? Name at least two and briefly explain each advantage. [6 pts]

Smaller code size because when space isn't needed it's not there.

Larger immediates, because the instruction can be made as large as necessary to fit the immediates.

Grading Note: Many gave advantages of CISC that were unrelated to variable instruction size, such as arithmetic instructions being able to load operands from memory. To get maximum credit, please answer the question that was asked.

(*b*) RISC ISAs have fixed length instructions. What advantages over CISC ISAs does that enable? Name at least one and briefly explain each advantage. [6 pts]

Branch displacements can go further since they can specify the number of instructions to skip rather than the number of characters to skip. The instruction fetch mechanism is easier to design.

(*c*) Why are programs written in a stack ISA small? Be brief but as specific as possible. [6 pts]

Because there is no need to provide space for register operands in many instructions.

(*d*) Listed below are several behaviors or features. For each, explain whether it is usually an ISA feature, an implementation feature, or an ABI (application binary interface) feature. **Briefly explain why.** If it can fit into more than one category (for example, ISA or implementation) say so and explain why. [6 pts]

☑ An `add` instruction raises an exception on an overflow.

ISA. Because this determines what a program would do and so it should be part of the ISA.

☑ The unused field in an instruction must be set to zero.

ISA. Specifies what instructions should look like and specifies the behavior of an illegal instruction (an otherwise legal instruction with an unused field that is not set to zero).

☑ Procedure call saves return address in a particular register.

ABI or ISA. Many ISAs do not have special call instructions, instead they have jump-and-link instructions that can save a return address in any register. The ABI defines which of those registers should be used for a return address. Some ISAs do have special call instructions that save to a particular register.

☑ Multiply instruction takes four cycles.

Implementation.