

The UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Comp 120 Computer Organization
Spring 2005

Problem Set #7

Issued Tuesday, 3/10/05; Due Thursday, 3/24/05

Homework Information: Some of the problems are probably too long to be done the night before the due date, so plan accordingly. Late homework will not be accepted. Feel free to get help from others, but the work you hand in should be your own.

Problem 1. “Life Without LUI”

The MIPS instruction set architecture provides the instruction `lui` to support loading registers with 32-bit constants. The following sequence is typical of its usage:

```
lui $t0, 0xfeed
ori $t0, 0xface      # $t0 = 0xfeedface
```

In the following series of questions assume that the `lui` instruction is not implemented, or is nonfunctional.

- (A) Give a two instruction sequence that emulates:

```
lui rd, constant
```

as a pseudoinstruction. If you need to use a register, limit yourself to `$at` (the register reserved for the assembler). Does your implementation function identically to the MIPS `lui` instruction? Explain your answer.

Consider the following instruction sequence:

```
...
jal skip
.word 0xfeedface
skip: lw $t0, 0($31)
...
```

- (B) What value is loaded into the `$t0` after the execution of the instruction labeled “`skip`”? Explain how this instruction sequence can be adapted to load large constants? Compare it to the standard approach (a `lui` followed by a `ori`) and the technique described in part A. Discuss any advantages, disadvantages, and other implications of this method.
- (C) In the MIPS architecture the `$gp` register is often used as a base address for accessing variables. Consider the implications of the following proposed convention for `$gp` usage:

All global and static variables should be placed within a non-negative offset of the address stored in `$gp`, whereas negative offsets reference 32-bit constants needed by the program.

Assuming that the \$gp register has been appropriately initialized, what is the instruction sequence for loading the i^{th} 32-bit constant? How does this approach compare to the standard approach using `lui`, and the other approaches discussed in this problem? What limitations are imposed by the proposed convention?

- (D) Suppose that that your MIPS implementation supported neither the `lui` nor any shift instructions (i.e. pseudoMIPS). Generate an instruction sequence that loads any arbitrary 32-bit constant into memory.
- (E) Discuss the hardware and ISA overhead, associated with supporting the MIPS `lui` instruction. Explain the benefits of this addition.

Problem 2. “Curse or Recurse?”

- (A) Write an assembly language implementation of the following C function for computing the greatest common denominator of its two arguments. Make sure that your code adheres to the procedure calling conventions discussed in class.

```
int gcd(int x, int y) {
    if (x == y) return x;
    if (y > x)
        y = y - x;
    else
        x = x - y;
    return gcd(x, y);
}
```

- (B) What is the minimum number of registers that `gcd(x, y)` **must** be save in the stack frame prior to calling itself?
- (C) What value is `gcd(2000000000, 2)` likely to return? Explain why?
- (D) Next, write an assembly language implementation of the following modified version of `gcd(x, y)`. You should still adhere to the procedure calling conventions.

```
int gcd(int x, int y) {
    while (x != y) {
        if (y > x)
            y = y - x;
        else
            x = x - y;
    }
    return x;
}
```

- (E) Discuss the various advantages and disadvantages of the two `gcd()` approaches. Which routine is shorter? Faster? More robust?

Problem 3. “Be Fruitful and Multiply”

You may have noticed that miniMIPS does not support the MIPS multiply and divide instructions. Write an assembly language function using only those instructions implemented by miniMIPS that computes the same result as the following C function.

```
int mul(int x, int y) {
    return x*y;
}
```

You should test your multiply function using the SPIM simulator, which can be downloaded from the “links” page on the course web site. Be sure to consider all combinations of positive and negative inputs, as well as results that overflow the 32-bit signed integer representation.

Problem 4. “Benchmarking Bandwagon”

The following statistics were gathered by running a benchmark program on the MIPS and counting how many instructions of each type were executed:

<i>Instruction type</i>	<i>Number of executions (x1000)</i>
lw	1149
sw	507
add, addi, addu, addiu	623
sub, subu	308
slt, slti, sltu, sltiu	93
sll, srl, sra, sllv, srlv, srav	156
and, andi	55
or, ori	186
beq, bne	532
j	40
jal	41
jr	61
Total	3751

- (A) Using the statistics from the table above, find the percentage of all memory accesses that are data accesses, the percentage of data accesses that are reads, and the percentage of all memory accesses that are reads. (Note: A *memory access* is any access to memory, i.e., instruction fetch, data read (lw), or data write (sw). A *data access* is any memory access generated by load (lw) or store (sw) instructions)
- (B) Assume instructions that change the PC (branches, jumps, calls, and returns) are uniformly distributed throughout the code, what is the length of the “average” basic block, based on the statistics in the table above? A basic block is a sequence of instructions stored in consecutive memory locations having the property that if the first instruction in the block is executed, all instructions in the block will be executed (i.e., it contains no branches, calls, or jumps, except for the last instruction in the block).

Next, consider the implications of adding a memory-register ALU instruction type to MIPS. The idea is to replace sequences like:

	lw	\$t0, offset(\$t1)	Reg[8] ← Mem[Reg[9] + offset]
	add	\$t2, \$t2, \$t0	Reg[10] ← Reg[10] + Reg[8]
with			
	addm	\$t2, offset(\$t1)	Reg[10] ← Reg[10] + Mem[Reg[9] + offset]

Assume that this new instruction type supports the following arithmetic and logical operations (addm, andm, orm, xorm, sltm), but its inclusion increases the clock cycle by 10%.

- (C) Based on the instruction frequencies from the table above, what percentage of loads must be replaced with the new instruction type (with its slower clock cycle) to have at least the same performance on the benchmark program?
- (D) Marketing literature claims that, despite the slower clock rate, this new instruction type improves the performance by nearly a factor of 2. Is this possible? Explain why or why not.