LSU EE 4720 **Homework 1** Solution **Due: 29 September 2008**

To answer the first question below see the MIPS32 Architecture manual linked to the course references page.

Problem 1: The MIPS I `bgtz` and `bltz` instructions compare a register to zero, but can't compare two registers (unless the second one is the zero register). Consider an extension of MIPS I that allowed branch greater than and branch less than instructions to compare two registers, call the new instructions `bgt` and `blt`. Explain why the existing `bgtz` opcode could be used for `bgt` but why the `bltz` opcode could not be used for `blt`. *Hint: See* `bltzal`.

The opcode for `bgtz` is `0x07` and the value that the ISA specifies for the `rt` field is 0. Assuming that no other instruction uses opcode `0x07`, the `rt` field could be used for the second comparison register. (This would not be incompatible with its current use because in its current use it is comparing the `rs` register to 0 so it wouldn't matter if `rt` held a register number.)

In the `bltz` instruction the `rt` field is being used as an extension of the opcode field and so it cannot be used for a register number. A new `blt` instruction would need its own opcode.

Problem 2: A C function and a part of a MIPS equivalent are shown below. The C function looks at the attributes of a car and decides what to pack in a promotional giveaway to the car buyer. The assembler code corresponds to the C function up until the last line (checking for a sun roof).

```
#define FE_SPORTY 0x1
#define FE_OFF_ROAD 0x2
#define FE_EFFICIENT 0x4
#define FE_SUN_ROOF 0x10000
#define FE_MANUAL_TRANSMISSION 0x20000
enum Giveaways { G_Food, G_Hiking_Boots, G_Sunblock, G_Driving_Gloves };
void prepare_promotion_package(Car_Object *car) {
  int car_features = car->features;
  if ( car_features & FE_OFF_ROAD ) pack(car, 1200, G_Hiking_Boots);
  if ( car_features & FE_SUN_ROOF ) pack(car, 200,  G_Sunblock);
}

# MIPS-I Equivalent of C code.
#
#       $a0:   Address of car object.
#       Notes: Procedure call arguments placed in $a0, $a1, ...
#              Assume that pack does not change $a0-$a3 or $s0-$s7

        lw $s0, 16($a0)        # Load the features bit vector of car object.

        andi $t0, $s0, 2
        beq $t0, $0  SKIP1
        addi $a1, $0, 1200
        jal pack
        addi $a2, $0, 1
SKIP1:

#       PART b SOLUTION STARTS HERE
```

(*a*) The MIPS code above omits the last line of C code (checking for a sun roof); complete it using MIPS I instructions. (Do this on paper, there is no need to run it.) *Hint: A clever solution uses five instructions a straightforward solution uses six instructions. If you have more than ten instructions ask for help.*

The solution appears below.

This would be an insultingly trivial problem were it not for the fact that an `andi` instruction can't be used to mask off the `FE SUN ROOF` bit because the constant, `0x10000`, is too large for the immediate field.

The solution uses an `sll` (shift left logical) instruction instead of an `andi` instruction to move the sun roof bit to the most significant bit position, making it into a sign bit. Replacing `beq` with `bgez` achieves the desired functionality.

```
# MIPS-I Equivalent of C code.
#
#       $a0:   Address of car object.
#       Notes: Procedure call arguments placed in $a0, $a1, ...
#              Assume that pack does not change $a0-$a3 or $s0-$s7

        lw $s0, 16($a0)        # Load the features bit vector of car object.

        andi $t0, $s0, 2       # Extract OFF_ROAD bit from vector.
        beq $t0, $0  SKIP1     # If OFF_ROAD bit not set, skip ahead.
        addi $a1, $0, 1200     # Arg 1: Promotional item weight
        jal pack               # Insert promotional item in Car_Object
        addi $a2, $0, 1        # Arg 2: Promotional item model number.
SKIP1:

#       PART b SOLUTION STARTS HERE
#       SOLUTION BELOW
#
        sll $t0, $s0, 15       # Make SUN_ROOF bit the sign bit.
        bgez $t0  SKIP2        # If SUN_ROOF not set (t0 >=0) skip ahead.
        addi $a1, $0, 200      # Arg 1: Promotional item weight.
        jal pack               # Insert promotional item in Car_Object
        addi $a2, $0, 2        # Arg 2: Promotional item model number.
SKIP2:
```

(*b*) Add comments to the assembler code above. Write the comments for an experienced MIPS and C programmer, that is, the comments should describe what an instruction is doing **in terms of what the C code is trying to do**. The comments **should not** just describe how instructions change register values.

For example, **a bad** comment for the `lw` instruction would be: Compute address 16 + `$a0`, retrieve word starting at that address and write into `$s0`. This is a bad comment because an experienced MIPS programmer already knows what an `lw` instruction does. The comment for `lw` in the code (Load the features...) is good because it tells the reader what the `$s0` value is in terms of what the code is supposed to do.

The comments have been added to the solution code above.

Grading Notes: Many solutions included something like the following comment for the `jal pack` routine: "Save the return address and call the pack routine." Points were deducted for such comments because an experienced MIPS programmer, and even beginners, already know what `jal` does. Comments like that increase the amount of time it takes someone to read (and write) the code.

Problem 3: Consider the code from the previous problem. Invent a new branch instruction that can be used for the kind of branching used in the code: testing if a single bit in a register value is 1.

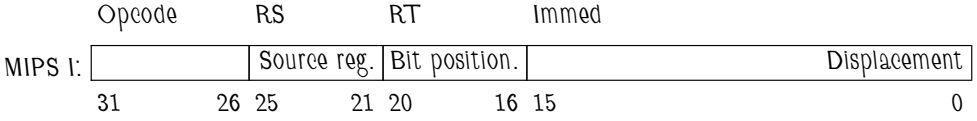
(a) Show the encoding for the new branch instruction. The new instruction must fit as naturally as possible with other instructions.

Call the new instruction `bbit`, branch if bit set. The `rs` register has the value to test and the `rt` field holds the bit number in the `rs` value to test. The `immed` field holds the displacement which is used like the displacement in any other branch.

In the example assembler below the branch is taken if bit position `16` in register `s0` is `1`. (This instruction could have been used in the first problem).

```
bbit $s0, 16,  SKIP2
```

The encoding appears below:



(b) Compare the implementation cost and performance of the new instruction to the existing MIPS-I `bltz` and to a hypothetical `blt` instruction. (With each instruction doing its own thing, not as part of functionally equivalent alternatives.)

Cost of `bbit`: A 32×1 bit multiplexer could be used to extract the desired bit position. The logic for that mux would implement the expression

$$b_0\overline{p_4}\,\overline{p_3}\,\overline{p_2}\,\overline{p_1}\,\overline{p_0} + b_1\overline{p_4}\,\overline{p_3}\,\overline{p_2}\,\overline{p_1}\,p_0 + b_2\overline{p_4}\,\overline{p_3}\,\overline{p_2}\,p_1\overline{p_0} \cdots + b_{31}p_4p_3p_2p_1p_0$$

where b_i is the bit at position i in the `rs` register value, and p_j, $0 \le j < 5$ is the bit at position j in the binary representation of `rt` (the field value, not the register value).

Cost of `blt`: A magnitude comparison unit is needed, the complexity of which is similar to an adder (or subtractor). Since it must be made fast, the cost would be comparable to the adder in the ALU. A ripple adder (or subtractor) is one of the least expensive designs, that requires about five gates per bit (counting an exclusive or as one gate). The higher-speed design would cost more.

The `bbit` logic could use a single six-input gate per bit (a term in the expression above), but to be conservative one might count it as five two-input gates. This cost is comparable to that of a binary full subtractor and so is certainly lower than the cost of the lookahead subtractor needed by `blt` to perform the comparison in `ID`. Therefore the cost of the hardware needed for `bbit` is less than the hardware needed for `blt`.

The bit test logic is two levels though it has a large fan in. The number of logic levels for the comparison depends on cost but is certainly greater than two levels. Therefore the `bbit` logic is probably faster than `blt`.

The `bltz` instruction only needs to test the sign bit, so the hardware cost of `bltz` is lower than `bbit` and `blt` and the logic for `bltz` is much faster than `bbit` and `blt`.

Problem 4: Solve Fall 2007 Homework 2 without looking at the solution. Then look at the solution and give yourself a grade on a scale of $[0, 1]$. **Warning:** test questions are based on the assumption that homework problems were completed, so make a full effort to solve it without first consulting the solution.