The UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

Comp 411 Computer Organization
Spring 2007

Problem Set #2
Issued Tuesday, 1/23/07; Due Thursday, 2/1/07

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. "Some Assembly Required"

The conversion of a mnemonic instruction to its binary representation is called assembly. This tedious process is generally delegated to a computer program for a variety of reasons. The first is that it alleviates the need to keep track of all the various bit encodings for each type of instruction. A second reason is that frequently the precise encoding of an instruction cannot be determined in a single pass. This is particularly true when referencing labels. In the following exercises, you will get a taste of what the task of translating from assembly to machine language is like.

Give binary and hexadecimal encodings for the following instructions:

```
(A)        sll  $t2,$t1,1
(B)        and  $t3,$a0,1
(C)        or   $t1,$t2,$t3
(D)        sra  $a0,$a0,1
(E)        add  $t0,$t0,-2
(F)  loop: bne  $t0,$0,loop
(G)        lui  $t0,0xbead
```

Problem 2. "Diss Assembly"

The inverse of assembly is disassembly, which involves translating an encoded binary instruction into its mnemonic representation. The process involves breaking an instruction into its constitute fields and decoding each instruction part.

For each of the following 32-bit numbers, given in hexadecimal, decode the corresponding MIPS instruction mnemonics, or otherwise indicate that it is an illegal instruction.

```
(A)  0x00004820
(B)  0x308a0001
(C)  0x11400003
(D)  0x00042042
(E)  0x21290001
(F)  0x1480fffd
(G)  0x308a0001
```

Problem 3. "Faking it"

MIPS assembly language provides opcode mnemonics for instructions that are not part of the instruction set architecture. For the most part, these pseudoinstructions can be generated using a sequence of one or more "true" MIPS instructions.

Find a "true-instruction" equivalent for each of the following pseudo-instructions (some are official MIPS pseudoinstructions, others are made up). Each of these can be implemented using only one real MIPS instruction. Discuss of your implementations, if any, and whether or not your implementation is unique (i.e. could some other instruction be used to achieve the same effect).

(A) `move rA, rB`

  Reg[rA] ← Reg[rB]
  Move register rB to rA

(B) `not rA, rB`

  Reg[rA] ← ~Reg[rB]
  Put the bitwise complement of register rB into register rA

(C) `neg rA, rB`

  Reg[rA] ← -Reg[rB]
  Put the negative of register rB into register rA

(D) `cmp rA, rB`

  if (Reg[rB] == 0)
    Reg[rA] ← 1
  else
    Reg[rA] ← 0
  Put the logical negation of register rB into register rA

(E) `subi rA, rB, imm`

  Reg[rA] ← Reg[rB] - imm
  Subtract sign-extended immediate constant 'imm' from rB and place result in rA

(F) `sign rA,rB`

  if (Reg[rB] < 0)
    Reg[rA] ← -1
  else
    Reg[rA] ← 0
  Set rA to -1 if rB is negative, otherwise set rB to 0

Problem 4. "Loading up at the Store"

The MIPS ISA provides access to memory exclusively through load (lw) and store (sw) instructions. Both instructions are encoded using the I-format, thus providing three operands, two registers and a 16-bit sign-extended constant. The memory address is computed by adding the contents of the register specified in the `rs` register field to the sign-extended 16-bit constant. The contents of the specified memory location are either loaded in the register specified in rt instruction field (lw), or the contents of that register are stored the indicated memory location (sw).

(A) It is possible to "directly" address a limited range of 32-bit memory locations by encoding the `rs` field as $0. How many memory locations can be addressed this way? Is this range of memory locations contiguous?

The intermediate result implied when computing a memory location is often called the instruction's "effective address". In the MIPS ISA the effective address is computed as

$$\text{Reg[rs]} + \text{imm}_{\text{sign\_extend}}$$

(B) When addressing words in memory what restrictions, if any, must be placed on the result of the effective address calculations?

(C) MIPS assemblers often provide a pseudoinstruction (see problem 3) for loading an effective address into a register called `la` for load address. The syntax of this pseudoinstruction matches the `lw` instruction, and an example is shown below:

```
la    $t0, 200($t1)
```

What actual instruction or instruction sequence is used to implement this pseudoinsruction?

(D) MIPS does not provide any instruction for specifying a memory address with a variable offset from `rs`. Such a construct is useful for implementing array accesses. Give a multiple-instruction sequence to accomplish this type of memory access using available MIPS instructions. Assume the array's base address (the location of it's 0^{th} member) is in register $t0, and the index is located in $t1. Comment on any restrictions, or additional processing that must be performed, on the index before the lw.

(E) In what way are store instructions, like `sw`, unique among the MIPS ISA in their use of the rt register field?