1 Translating between C and MIPS

Translate between the C and MIPS code. You may want to use the MIPS Green Sheet as a reference. In all of the C examples, we show you how the different variables map to registers – you don’t have to worry about the stack or any memory-related issues.

<table>
<thead>
<tr>
<th>C</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Strcpy:</td>
<td>addiu $t0, $0, 0</td>
</tr>
<tr>
<td>// $s1 -&gt; char s1[]</td>
<td>Loop: addu $t1, $s1, $t0 # s1[i]</td>
</tr>
<tr>
<td>// $s2 -&gt; char *s2 =</td>
<td>addu $t2, $s2, $t0 # s2[i]</td>
</tr>
<tr>
<td>malloc(sizeof(char)*7);</td>
<td>lb $t3, 0($t1) # char is</td>
</tr>
<tr>
<td>int i = 0;</td>
<td>sb $t3, 0($t2) # 1 byte!</td>
</tr>
<tr>
<td>do {</td>
<td>addiu $t0, $t0, 1</td>
</tr>
<tr>
<td>s2[i] = s1[i];</td>
<td>addiu $t1, $t1, 1</td>
</tr>
<tr>
<td>i++;</td>
<td># unnecessary line</td>
</tr>
<tr>
<td>} while(s1[i] != '\0');</td>
<td>lb $t4, 0($t1) # could use offset</td>
</tr>
<tr>
<td>s2[i] = '\0';</td>
<td>bne $t4, $0, Loop</td>
</tr>
<tr>
<td>// Nth_Fibonacci(n):</td>
<td>Done: sb $t4, 1($t2)</td>
</tr>
<tr>
<td>// $s0 -&gt; n, $s1 -&gt; fib</td>
<td></td>
</tr>
<tr>
<td>// $t0 -&gt; i, $t1 -&gt; j</td>
<td></td>
</tr>
<tr>
<td>// Assume fib, i, j are these values</td>
<td></td>
</tr>
<tr>
<td>int fib = 1, i = 1, j = 1;</td>
<td></td>
</tr>
<tr>
<td>if (n==0) return 0;</td>
<td></td>
</tr>
<tr>
<td>else if (n==1) return 1;</td>
<td></td>
</tr>
<tr>
<td>n -= 2;</td>
<td></td>
</tr>
<tr>
<td>while (n != 0) {</td>
<td></td>
</tr>
<tr>
<td>fib = i + j;</td>
<td></td>
</tr>
<tr>
<td>j = i;</td>
<td></td>
</tr>
<tr>
<td>i = fib; n--;</td>
<td></td>
</tr>
<tr>
<td>} return fib;</td>
<td></td>
</tr>
<tr>
<td>// Collatz conjecture</td>
<td></td>
</tr>
<tr>
<td>// $s0 -&gt; n</td>
<td></td>
</tr>
<tr>
<td>unsigned n;</td>
<td></td>
</tr>
<tr>
<td>L1: if (n % 2) goto L2;</td>
<td>L1: addiu $t0, $0, 2</td>
</tr>
<tr>
<td>goto L3;</td>
<td>div $s0, $t0 # puts (n%2) in $hi</td>
</tr>
<tr>
<td>L2: if (n == 1) goto L4;</td>
<td>mfhi $t0 # sets $t0 = (n%2)</td>
</tr>
<tr>
<td>n = 3 * n + 1;</td>
<td>bne $t0, $0, L2</td>
</tr>
<tr>
<td>goto L1;</td>
<td>j L3</td>
</tr>
<tr>
<td>L3: n = n &gt;&gt; 1;</td>
<td>L2: addiu $t0, $0, 1</td>
</tr>
<tr>
<td>goto L1;</td>
<td>beq $s0, $t0, $0, 1</td>
</tr>
<tr>
<td>L4: return n;</td>
<td>addiu $t0, $0, 3</td>
</tr>
<tr>
<td></td>
<td>mul $s0, $t0, L4</td>
</tr>
<tr>
<td></td>
<td>addiu $t0, $0, 3</td>
</tr>
<tr>
<td></td>
<td>j L1</td>
</tr>
<tr>
<td></td>
<td>L3: srl $s0, $s0, 1</td>
</tr>
<tr>
<td></td>
<td>j L1</td>
</tr>
<tr>
<td></td>
<td>L4: ...</td>
</tr>
</tbody>
</table>
2 MIPS Instruction Formats

Instructions are represented as bits (just like numbers), so it’s a good idea to store instructions in memory just like data (why?). In MIPS Instruction Format, every instruction is represented as a fixed 32-bit word, and a instruction is further divided into different fields.

(1) About MIPS Instruction Formats

(a) **I format**: used for instructions with immediates, lw and sw (since offset counts as an immediate), and branches (beq and bne)
   - opcode: 6 bits, rs: 5 bits, rt: 5 bits, immediate: 16 bits.
   - For branch: the immediate field is signed int and word-aligned.

(b) **J format**: used for general jumps, (j and jal). We may jump to “anywhere” in memory (why?).
   - opcode: 6 bits, target address: 26 bits.
   - New PC = \{(PC+ 4)[ 31 ... 28 ], target address, 00 \}

(c) **R format**: used for all other instructions.
   - opcode: 6 bits, rs: 5 bits, rt: 5 bits, rd: 5 bits, shamt: 5 bits, funct: 6 bits.
   - the opcode field for all R-type instruction is 0.

(2) Exercises:

(a) What instruction is 0x00008A03?

```
Hex -> bin: 0000 0000 0000 0000 1000 1010 0000 0011
0 opcode -> R-type: 000000 00000 00000 10001 01000 000011
sra $s1 $0 8
```

(b) What is the hexadecimal representation of instruction "addiu $a0, $0, 0xABC"

```
common mistake: 001001 00100 00000 0000ABC = 24800ABC = addiu $0, $a0, 0xABC
correct answer: 001001 00000 00100 0000ABC = 0x24040ABC
```

3 MIPS Addressing Modes

- We have several **addressing modes** to access memory (immediate not listed):

  (a) **Base displacement addressing**: Adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb)

  (b) **PC-relative addressing**: Uses the PC (actually the current PC plus four) and adds the I-value of the instruction (in word, so multiplied by 4) to create an address (used by I-format branching instructions like beq, bne)

  (c) **Pseudodirect addressing**: Uses the upper four bits of the PC and concatenates a 26-bit value from the instruction ("in word", with implicit 00 lowest bits) to make a 32-bit address (used by J-format instructions)
Register Addressing: Uses the value in a register as a memory address (jr)

1. You need to jump to an instruction that $2^{28} + 4$ bytes higher than the current PC. How do you do it? Assume you know the exact destination address at compile time. (Hint: you need multiple instructions)

   The jump instruction can only reach addresses that share the same upper 4 bits as the PC. A jump $2^{28} + 4$ bytes away would require changing the fourth highest bit, so a jump instruction is not sufficient. We must manually load our 32 bit address into a register and use jr.

   ```
   lui $at {upper 16 bits of Foo}
   ori $at $at {lower 16 bits of Foo}
   jr $at
   ```

2. You now need to branch to an instruction $2^{17} + 4$ bytes higher than the current PC, when $t0$ equals 0. Assume that were not jumping to a new $2^{28}$ byte block. Write MIPS to do this.

   The largest address a branch instruction can reach is PC + 4 + SignExtImm. The immediate field is 16 bits and signed, so the largest value is $2^{15} - 1$ words, or $2^{17} - 4$ Bytes. Thus, we cannot use a branch instruction to reach our goal, but by the problems assumption, we can use a jump. Assuming were jumping to label Foo

   ```
   bne $t0 $0 DontJump
   j Foo
   ```

   ```
   DontJump: ...
   ```

3. Given the following MIPS code (and instruction addresses), fill in the blank fields for the following instructions (youll need your green sheet!):

   ```
   0x002cff00: loop: addu $t0, $t0, $t0 | 0 | 8 | 8 | 8 | 0 | 0x21 |
   0x002cff04: jal foo | 3 | 0xc0001 |
   0x002cff08: bne $t0, $zero, loop | 5 | 8 | 0 | -3 = 0xfffd |
   ... ...
   0x00300004: foo: jr $ra $ra=__0x002cff08__
   ```