### MIPS Control Flow

1) What are the instructions to branch on each of the following conditions?

<table>
<thead>
<tr>
<th>Condition</th>
<th>Instruction(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s0 &lt; $s1</td>
<td><code>slt $t0, $s0, $s1</code>; <code>bne $t0, $0, foo</code></td>
</tr>
<tr>
<td>$s0 &lt;= $s1</td>
<td><code>slt $t0, $s1, $s0</code>; <code>beq $t0, $0, foo</code></td>
</tr>
<tr>
<td>$s0 &gt; 1</td>
<td><code>addi $t1, $0, 1</code>; <code>slt $t0, $t1, $s0</code>; <code>bne $t0, $0, foo</code></td>
</tr>
<tr>
<td>$s0 &gt;= 1</td>
<td><code>slt $t0, $s0, 1</code>; <code>beq $t0, $0, foo</code></td>
</tr>
</tbody>
</table>

2) Translate the following C code into MIPS.

```c
// Strcpy:
// $s1 -> char s1[] = "Hello!";
// $s2 -> char *s2 = malloc(sizeof(char)*7);
int i=0;
do {
   s2[i] = s1[i];
i++;
} while(s1[i] != '\0');
s2[i] = '\0';
```

```mips
addiu $t0, $0, 0
Loop: addu $t1, $s1, $t0 # s1[i]
addu $t2, $s2, $t0 # s2[i]
lb $t3, 0($t1) # char is
sb $t3, 0($t2) # 1 byte!
addiu $t0, $t0, 1
addiu $t1, $t1, 1
lb $t4, 0($t1)
bne $t4, 0, Loop
Done: addiu $t2, $t2, 1
      sb $t4, 0($t2)
```

```c
// Nth_Fibonacci(n):
// $s0 -> n, $s1 -> fib
// $t0 -> i, $t1 -> j
int fib = 1, i = 1, j = 1;
if(n==0) return 0;
else if(n==1) return 1;
n-=2;
while(n != 0) {
   fib = i + j;
   j = i;
   i = fib;
   n--;
}
return fib;
```

```mips
... 
beq $s0, $0, Ret0
addi $t2, $0, 1
beq $s0, $t2, Ret1
addiu $s0, $s0, -2
Loop: beq $s0, $0, RetF
addu $s1, $t0, $t1
addiu $t0, $t1, 0
addiu $t1, $s1, 0
addiu $s0, $s0, -1
j Loop
Ret0: addiu $v0, $0, 0
      j Done
Ret1: addiu $v0, $0, 1
      j Done
RetF: addu $v0, $0, $s1
      ... 
Done: jr $ra
```
Instruction Formats

MIPS instructions come in three tasty flavors!

**R-Instruction format (register-to-register)**  Examples: addu, and, sll, jr

<table>
<thead>
<tr>
<th>op code</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

See green sheet to see what registers are read from and what is written to

**I-Instruction Format (register immediate)**  Examples: addiu, andi, bne

<table>
<thead>
<tr>
<th>op code</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Note: Immediate is 0 or sign-extended depending on instruction (see green sheet)

**J-Instruction Format (jump format)**  For j and jal

<table>
<thead>
<tr>
<th>op code</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

KEY: An instruction is R-Format if the op code is 0. If the opcode is 2 or 3, it is J-format. Otherwise, it is I-format. Different R-format instructions are determined by the “funct”.

1. How many total possible instructions can we represent with this format?

We count the number of possible instructions in each format:
R – 64 (op code 0, all the bits of func), I – 61, J – 2, 127 total.

2. What could we do to increase the number of possible instructions?

There are a number of possible solutions, all of which roughly take the form, “borrow bits from another field and add them to opcode/func.” Examples of this would be sacrificing bits of the I-format immediate for extra opcode bits. This costs us range in the immediates we can represent and the range of our branch instructions.

MIPS Addressing Modes

- We have several addressing modes to access memory (immediate not listed):
  - **Base displacement addressing**: Adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb)
  - **PC-relative addressing**: Uses the PC (actually the current PC plus four) and adds the I-value of the instruction (multiplied by 4) to create an address (used by I-format branching instructions like beq, bne)
  - **Pseudodirect addressing**: Uses the upper four bits of the PC and concatenates a 26-bit value from the instruction (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 memory (jr)
3. You need to jump to an instruction that $2^{28} + 4$ bytes higher than the current PC. How do you do it? (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
```

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

The total range of a branch instruction is $-2^{17} \rightarrow (2^{17}) - 4$ bytes (a 16 bit signed integer that counts by words). Thus, we cannot use a branch instruction to reach our goal, but by the problem’s assumption, we can use a jump. Assuming we’re jumping to label Foo:

```
beq $t0 $0 DoJump
[...]
DoJump: j Foo
```

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

```
0x002cff00: loop: addu $t0, $t0, $t0 | 0 | 8 | 8 | 8 | 0 | 0x21 |
0x002cff04:     jal foo   | 3 | 0x00001 |
0x002cff08:     bne $t0, $zero, loop | 5 | 8 | -3 = 0xfffd |
...  
0x00300004: foo:  jr $ra $ra=___0x002cff08___
```

6. 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
```