The Stored Program Concept

- All programs (instructions) are just data represented by combinations of bytes!
- Any block of memory can be code. Consequently, self-modifying code is possible!
- The Program Counter (PC) is a special register (not directly accessible) which holds a pointer to the current instruction.

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 instructions are representable 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.

The 2^12 solution additionally mentioned in section is actually incorrect – when the opcode is non-zero the func field no longer exists – it becomes part of either the immediate field of the I format or the address field of the J format.

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 is 257Mb up from the current PC. How do you do it? (HINT: you need multiple instructions)

257Mb from our current PC is guaranteed to lie in a separate 256 Mb block of memory, so we cannot rely on a jump instruction to get there. Assuming the address we are jumping to is Foo:

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

4. You now need to branch to an instruction 129Kb up from the current PC when $t0 equals 0. Assume that we’re not jumping to a new 256Mb block. Write MIPS to do this. Assuming our destination is the label Foo:

```
beq $t0 $0 DoJump
[...]
```

```
DoJump: j Foo
```

5. Given the following MIPS code (and instruction addresses), fill in the highlighted instructions (you’ll 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 | -3 = 0xfffd |
... 
0x00300004: foo: jr $ra $ra=__0x002cff08___
```

5. **What instruction is** 0x00008A03?

| 0000 0000 0000 0000 1000 1010 0000 0011 |
| 000000 00000 00000 10000 01000 000011 - R type |

```
sra $s1 $0 8
```