2. Instructions Are Just Numbers!

Key concept: Both instructions and data are indistinguishable.

This means code can be moved around and saved just like data, and data can be loaded and executed as code. If we call the jump instruction and go to the address of a variable in say static space, that variable's data will be executed as if it were a MIPS instruction. We can also modify code with code.

3. 3 Types of MIPS Instructions

Type	**-31-**			**format (bits)**		**-0-**
R	opcode(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)
I	opcode(6)	rs(5)	rt(5)	immediate(16)		
J	opcode(6)	address(26)				

R type instructions usually operate on 3 registers except for shifts. E.g. add, sub, sll, srl, sra, slt
I type instructions perform operations with intermediates. E.g. addi, lw, beq, slti
J type instructions are j or jal, both of which change the *PC* to the address given. E.g. j, jal

There are 6 bits used for the *opcode*, which specifies what type of instruction we have. *Opcode* 0 = R type, 2 or 3 = J type. Anything else is I. *Funct* allows us to further categorize R type instructions.

Note that for R types, the order of registers is different from what we're used to in assembly code, which is $rd, $rs, $rt. Maybe use the mnemonics STD vs DeST.

4. Branches and jumps

Branches are immediate instructions, which means that we can't access all of memory. Instead, we treat the immediate as a signed integer offset from the *PC*, or the address of the current instruction in memory. Also, since instructions must be word aligned in memory, we can assume that the bottom two bits of any address are 00. Thus, we can reach +/- 2^{15} words or +/- 2^{17} bytes from the current address.

This is usually sufficient because branches usually jump to somewhere close in code; *if* comparisons and *loops* aren't usually very long.
Jumps are handled differently because they are more likely to target somewhere distant in memory.

All in all, there are several methods of addressing memory:

- Base displacement mode
  - Used by lw, lb, sw, sb. Add an intermediate to a register and use the result as the new address.
- PC-relative addressing
  - Used by beq, bne. Use (intermediate * 4) as an offset from the PC as the new address.
- Pseudodirect addressing
  - J type instructions do this. Use the upper four bits from the PC, 26 bits from the instruction, and

00 for the bottom two bits. Go to that address.

- Register addressing
  - jr does this. Useful when we do need to use all 32 bits as an address.

5. Solved Problems (Some problems taken from Andy Carle/Aaron Staley/David Jacobs)

0. What type of instruction (R, I, J) is jr? Ans: R

1. Addi cannot be used with large intermediates because there are only 16 bits in the field for the intermediate.

```
addi $s0, $0, LARGE_NUM
```

Therefore, the assembler uses $at to break the above instruction into the following:

```
add $at, $0, $0
lui $at, UPPER
addiu $at, $at, LOWER
add $s0, $0, $at
```

Are there any issues with this? If so, explain what the problem is and how you would solve it.

Ans: addiu sign extends the intermediate, so we will have overflow and the incorrect # of upper bits.
How about addiu and then lui? No, because lui sets the lowest bits to 0.
Need to do lui and then ori.

2. How many distinct instructions are representable using the MIPS instruction format?

2^32 instructions, though not all are used.

3. Which instruction is 0x00008A03?

000000 00000 00000 10001 01000 000011
0 0 0 17 8 3
sra $s1 $0 8

4. If there are only 26 bits for the address in the J types, what is the maximum range of memory that a J instruction can reach?

256MB, because 26 actually = 28 bits byte addressed. 2^28 = 256 mebibytes.

5. Write a simple program to perform multiplication in hardware (without using the MUL instruction)

```
int mult (a0, a1); // assume correct input - two unsigned integers whose product
can fit into a register

move $v0, $0
Loop: beq $a1, $0, End
      addi $a1, $a1, -1
      addu $v0, $v0, $a0
```

```
      j Loop
End:  jr $ra
```

Now write a faster version that uses more lines of code and more registers but is more efficient.

```
move $v0, $0
Loop: beq $a1, $0, End
      andi $t0, $a1, 1
      beq $t0, $0, skip
      addu $v0, $v0, $a0
skip: srl $a1, $a1, 1
      sll $a0, $a0, 1
      j Loop
End:  jr $ra
```