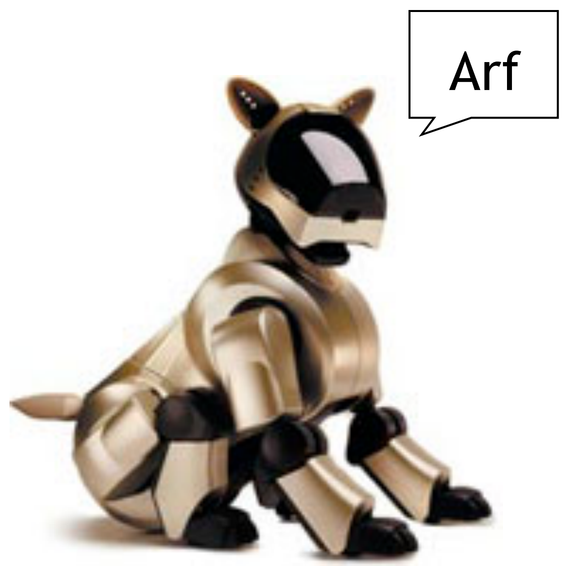
What does this code do?

```
label:  sub  $a0, $a0, 1
        bne  $a0, $zero, label
```



Today's Lecture

- We'll go into more detail about the ISA.
  - Pseudo-instructions
  - Using branches for conditionals

Pseudo-instructions

- MIPS assemblers support pseudo-instructions that give the illusion of a more expressive instruction set, but are actually translated into one or more simpler, "real" instructions.
- In addition to the la (load address) we saw on last lecture, you can use the li and move pseudo-instructions:

```
li    $a0, 2000          # Load immediate 2000 into $a0
move        $a1, $t0     # Copy $t0 into $a1
```

- They are probably clearer than their corresponding MIPS instructions:

```
addi        $a0, $0, 2000    # Initialize $a0 to 2000
add   $a1, $t0, $0           # Copy $t0 into $a1
```

- We'll see lots more pseudo-instructions this semester.
  - A complete list of instructions is given in Appendix A of the text.
  - Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams.

Control flow in high-level languages

- The instructions in a program usually execute one after another, but it's often necessary to alter the normal control flow.
- Conditional statements execute only if some test expression is true.

```
// Find the absolute value of *a0
v0 = *a0;
if (v0 < 0)
     v0 = -v0;              // This might not be executed
v1 = v0 + v0;
```

- Loops cause some statements to be executed many times.

```
// Sum the elements of a five-element array a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
     v0 = v0 + a0[t0];      // These statements will
     t0++;                  // be executed five times
}
```

Control-flow graphs

- It can be useful to draw control-flow graphs when writing loops and conditionals in assembly:

```
// Find the absolute value of *a0
v0 = *a0;
if (v0 < 0)
     v0 = -v0;
v1 = v0 + v0;
```

```
// Sum the elements of a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
     v0 = v0 + a0[t0];
     t0++;
}
```

MIPS control instructions

- In section, we introduced some of MIPS's control-flow instructions

j	// for unconditional jumps
bne and beq	// for conditional branches
slt and slti	// set if less than (w/ and w/o an immediate)

- And how to implement loops

- Today, we'll talk about
  – MIPS's pseudo branches
  – if/else
  – case/switch (bonus material)

Pseudo-branches

- The MIPS processor only supports two branch instructions, beq and bne, but to simplify your life the assembler provides the following other branches:

```
blt  $t0, $t1, L1   // Branch if $t0 < $t1
ble  $t0, $t1, L2   // Branch if $t0 <= $t1
bgt  $t0, $t1, L3   // Branch if $t0 > $t1
bge  $t0, $t1, L4   // Branch if $t0 >= $t1
```

- There are also immediate versions of these branches, where the second source is a constant instead of a register.

- Later this semester we'll see how supporting just beq and bne simplifies the processor design.

Implementing pseudo-branches

- Most pseudo-branches are implemented using slt. For example, a branch-if-less-than instruction `blt $a0, $a1, Label` is translated into the following.

```
slt   $at, $a0, $a1           // $at = 1 if $a0 < $a1
bne   $at, $0, Label          // Branch if $at != 0
```

- This supports immediate branches, which are also pseudo-instructions. For example, `blti $a0, 5, Label` is translated into two instructions.

```
slti  $at, $a0, 5             // $at = 1if $a0 < 5
bne   $at, $0, Label          // Branch if $a0 < 5
```

- All of the pseudo-branches need a register to save the result of slt, even though it's not needed afterwards.
  - MIPS assemblers use register $1, or $at, for temporary storage.
  - You should be careful in using $at in your own programs, as it may be overwritten by assembler-generated code.

Translating an if-then statement

- We can use branch instructions to translate if-then statements into MIPS assembly code.

```
v0 = *a0;
if (v0 < 0)
     v0 = -v0;
v1 = v0 + v0;
```

```
      lw  $v0, 0($a0)
      bgt $v0, 0, skip
      sub $v0, $zero, $v0
skip: add $v1, $v0, $v0
```

- Sometimes it's easier to *invert* the original condition.
  - In this case, we changed "continue if v0 < 0" to "skip if v0 >= 0".
  - This saves a few instructions in the resulting assembly code.

Control-flow Example

- Let's write a program to see if a number is a power of 3.

See supplementary material.

Translating an if-then-else statements

- If there is an else clause, it is the target of the conditional branch
  - And the then clause needs a jump over the else clause

```
// increase the magnitude of v0 by one
if (v0 < 0)
     v0 --;

else
     v0 ++;
v1 = v0;
```

```
          bge  $v0, $0, E
          sub  $v0, $v0, 1
      j   L

      E:  add  $v0, $v0, 1
      L:  move      $v1, $v0
```

- Dealing with else-if code is similar, but the target of the first branch will be another if statement.
  - Drawing the control-flow graph can help you out.

Bonus Material

Case/Switch Statement

- Many high-level languages support multi-way branches, e.g.

```
switch (two_bits) {
    case 0:     break;
    case 1:     /* fall through */
    case 2:     count ++;     break;
    case 3:     count += 2;   break;
}
```

- We could just translate the code to if, thens, and elses:

```
if ((two_bits == 1) || (two_bits == 2)) {
    count ++;
} else if (two_bits == 3) {
    count += 2;
}
```

- This isn't very efficient if there are many, many cases.

Case/Switch Statement

```
switch (two_bits) {
    case 0:       break;
    case 1:       /* fall through */
    case 2:       count ++;     break;
    case 3:       count += 2;   break;
}
```

- Alternatively, we can:
  1. Create an array of jump targets
  2. Load the entry indexed by the variable two_bits
  3. Jump to that address using the jump register, or jr, instruction

- This is much easier to show than to tell.
  – (see the example with the lecture notes online)