If and Loop Statements in MIPS

Branch Instructions

In the MIPS assembly language, there are only two types of conditional branch instructions. One branch if two registers are equal, the other if they are not equal.

- `beq` (branch if equal): `beq $value1, $value2, offset`  # if ($value1 == $value2) goto offset;
- `bne` (branch if not equal): `bne $value1, $value2, offset`  # if ($value1 != $value2) goto offset;

The third operand in the instruction is the offset. In MIPS, this is an 16 bit signed integer that represents where to continue if the comparison returns true. The offset represents how many instructions, counting from the instruction after the branch instruction, to pass over in order to get to the correct instruction. For example, look at the code below:

```assembly
addi $t0, $s1, 20
add $s1, $zero, $zero
bne $t0, $s1, -5
```

In this code, the branch instruction would move up two instructions from the instruction after itself. This means it would branch from position 0x4000:0008 to position 0x4000:0003 and then continue evaluating the instructions in sequence. It works going in the opposite direction as well.

```assembly
addi $t0, $s1, 20
add $s1, $zero, $zero
bne $t0, $s1, 5
```

In the case above, the branch would go from position 0x4000:0010 to position 0x4000:0014 before continuing evaluation of the instruction in sequence. Notice that the total number of bytes skipped is found by multiplying offset by 4.

Determining Inequalities

Once again the designers of MIPS chose to keep inequalities simple. They only allow you to check to see if one value is less than another value. However, there are four flavors of this instruction. Half of them take only registers, and the other half can compare to see if an immediate value is greater than a register value. For these two versions, each has an unsigned version in the occasion that you are only testing positive values. The answer is put into a register.

A `>` for false and a `1` for true.

### Signed Instructions:
- `slt` (smaller than): `slt $dest, $smaller, $greater`  # $dest = ($smaller < $greater) ? 1 : 0;
- `sle` (smaller or equal): `sle $dest, $smaller, $greater`  # $dest = ($smaller < $greater) ? 1 : 0;

### Unsigned Instructions:
- `ultr` (unsigned smaller than): `ultr $dest, $smaller, $greater`  # $dest = ($smaller < $greater) ? 1 : 0;
- `ule` (unsigned smaller or equal): `ule $dest, $smaller, $greater`  # $dest = ($smaller < $greater) ? 1 : 0;

Recall that the C expression: `if ($pred) { consequent } alternate;` is the same as the C expression: `if ($pred) consequent; else alternate;`

The C If and MIPS

Comparing the C if expression with MIPS branch statements may help in writing code. Especially when you know how to "express" yourself in C, but perhaps not as well in assembly language.

First let's examine a simple if expression and break it up into different parts:

```c
define if ($pred) consequent; else alternate;
```

if ($pred consequent)

- **Predicate**: `$pred` is a single expression that represents how many instructions, counting from the instruction after the branch instruction, to pass over in order to get to the correct instruction. For example, look at the code below:

```assembly
beq $t0, $zero, endif
```

- **Branch Statement to continue loop**

else

```assembly
addi $s2, $zero, 10
```

- **Consequent**: add `addi $s2, $zero, 10` for `else` above.

endif:

```assembly
```

Some similar can be done to if statements with `else` statements in them.

```assembly
```

Notice that we can divide our MIPS code into three regions, the predicate, the branch statement, and the consequent. The first of these regions is the predicate. Any number of statements that produce a zero or non-zero value in a register. The second region is the branch statement. If `beq` is used with `zero`, a non-zero value would be true, and if `bne` is used with `zero`, a zero value would be true. The third region is the consequent. This does whatever should be done if a true value results.

```assembly
```

The C while and MIPS

The C while expression closely resembles the if expression. It has a predictive something that happens continuously as long as the expression returns true.

```assembly
```

Notice how it's form doesn't really change that much from the while loop. It's really just a construct in C to make code more compressed and readable. The form in MIPS looks like:

```assembly
```

### Determining Inequalities

#### Signed Instructions:
- `slt` (smaller than): `slt $t0, $s1, $s2`  # if ($s1 < $s2) $t0, $s1, $s2
- `sle` (smaller or equal): `sle $t0, $s1, $s2`  # if ($s1 <= $s2) $t0, $s1, $s2

#### Unsigned Instructions:
- `ultr` (unsigned smaller than): `ultr $t0, $s1, $s2`  # if ($s1 < $s2) $t0, $s1, $s2
- `ule` (unsigned smaller or equal): `ule $t0, $s1, $s2`  # if ($s1 <= $s2) $t0, $s1, $s2

Recall that the C expression: `for ($pred) { consequent } alternate;` is the same as the C expression: `for ($pred) consequent; else alternate;`

The C do and MIPS

The C do expression resembles the while, except that it doesn’t have a loop back statement, that’s where the predicate and the branch statement both go to continue looping only if the predicate is true.

```assembly
```

Notice how it's form doesn't really change that much from the while loop. It's really just a construct in C to make code more compressed and readable. The form in MIPS looks like:

```assembly
```

### Branch Instructions

The C for and MIPS

The C for expression is like the while expression except it has two additional components to it. Not only does it have the consequent body which is evaluated continuously as long as the predicate returns a true value, it has **initialization** and **next statements** built into it. It would look as follows:

```assembly
```

The C for and MIPS

The C for expression is like the while expression except it has two additional components to it. Not only does it have the consequent body which is evaluated continuously as long as the predicate returns a true value, it has **initialization** and **next statements** built into it. It would look as follows:

```assembly
```

#### Initialization Statements

```assembly
```

#### Branch Statement to exit loop

#### Next Statements

```assembly
```

#### Looped Back Statement

```assembly
```

The C while and MIPS

The C while expression closely resembles the if expression. It has a predictive something that happens continuously as long as the expression returns true.

```assembly
```

Notice how it's form doesn't really change that much from the while loop. It's really just a construct in C to make code more compressed and readable. The form in MIPS looks like:

```assembly
```

#### Branch Statement to continue loop

```assembly
```

The C do and MIPS

The C do expression resembles the while, except that it doesn’t have a loop back statement, that’s where the predicate and the branch statement both go to continue looping only if the predicate is true.

```assembly
```

Notice how it's form doesn't really change that much from the while loop. It's really just a construct in C to make code more compressed and readable. The form in MIPS looks like:

```assembly
```