By Joshua Cantrell
jjc@cory.berkeley.edu

If and Loop Statements in MIPS

Branch Instructions

In the MIPS assembly language, there are only two types of conditional branch instructions. This means you don't have to remember any great variety of special case branching mechanisms. One branches if two registers are equal, the other if they are not equal.

```
beq $value1, $value2, offset        # if ($value1 == $value2) goto offset;
bne $value1, $value2, offset        # if ($value1 != $value2) goto offset;
```

The third operand in the instruction is the *offset*. In MIPS, this is a 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:

```
0x4000:0000   add $t1, $t2, $t3
0x4000:0004   beq $t1, $t3, -2
0x4000:0008   sub $t1, $t1, $t3
```

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 0x4000:0000 and then continue evaluating the instructions in sequence. It works going in the opposite direction as well.

```
0x4000:000C   bne $t1, $t3, 1
0x4000:0010   addi $t1, $t3, 20
0x4000:0014   addi $t3, $t3, -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 version, each has an unsigned version in the occasion that you are only testing positive values. The answer is put into a register. A '0' for false and a '1' for true.

```
# Signed Instructions:
slt $dest, $smaller, $greater       # $dest = ($smaller < $greater) ? 1 : 0;
slti $dest, $smaller, greater       # $dest = ($smaller < greater) ? 1 : 0;

# Unsigned Instructions:
sltu $dest, $smaller, $greater      # $dest = ($smaller < $greater) ? 1 : 0;
sltiu $dest, $smaller, greater      # $dest = ($smaller < greater) ? 1 : 0;
```

Recall that the C expression: `pred ? consequent : alternate;`
Is the same as the C expression: `if (pred) consequent; else alternate;`

By Joshua Cantrell
jjc@cory.berkeley.edu

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:

```
if (pred) consequent
```

If consists of both a *predicate* and a *consequent*. The predicate is itself a single expression that results in either a "true" or "false" value (non-zero or zero value). The consequent is a single statement expression, or multiple statement expressions surrounded by braces, '{' and '}'. We can make a similar construct using MIPS assembly code:

```
predicate:  slt $t0, $s1, $s2           # if ($s1 < $s2)
            beq $t0, $zero, endif       #
consequent: addi $s1, $s1, 1            #    $s1++;
endif:                                  #
```

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.

```
predicate:  slt $t0, $s1, $s2           } Predicate ($t0 = 1 if true)
            beq $t0, $zero, endif       } Branch Statement
consequent: addi $s1, $s1, 1            } Consequent (skipped if not true)
endif:
```

Something similar can be done to if statements with *else* statements in them.

```
predicate:  slt $t0, $s1, $s2           # if ($s1 < $s2)
            beq $t0, $zero, alternate   #
consequent: addi $s1, $s1, 1            #    $s1++;
            j endif                     # else
alternate:  addi $s2, $s2, 1            #    $s2++;
endif:                                  #
```

```
predicate:  slt $t0, $s1, $s2           } Predicate ($t0 = 1 if true)
            beq $t0, $zero, endif       } Branch Statement to alternate
consequent: addi $s1, $s1, 1            }
            j endif                     } Consequent (skipped if not true)
alternate:  addi $s2, $s2, 1            } Alternate
endif:
```


By Joshua Cantrell
jjc@cory.berkeley.edu

Notice the use of the jump instruction for MIPS. This instruction jumps without condition to the location given. The location is specified by a 28bit number. There is also an instruction that jumps without condition to the location given inside a register.

```
j location              # goto location;
jr $location            # goto $location;
```

The C *while* and MIPS

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

```
predicate:  slt $t0, $s1, $s2           # while ($s1 < $s2)
            beq $t0, $zero, endwhile    #    {
consequent: addi $s1, $s1, 1            #    $s1++;
            j predicate                 #    }
endwhile:                               #
```

```
predicate:  slt $t0, $s1, $s2           } Predicate ($t0 = 1 if true)
            beq $t0, $zero, endwhile    } Branch Statement to exit loop
consequent: addi $s1, $s1, 1            } Consequent (skipped if not true)
            j predicate                 } Loop Back Statement
endwhile:
```

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 returns true.

```
doloop:     addi $s1, $s1, 1            # do { $s1++;
predicate:  slt $t0, $s1, $s2           #    }
            bne $t0, $zero, doloop      # while ($s1 < $s2);
enddo:                                  #
```

```
doloop:     addi $s1, $s1, 1            } Looped Statement
predicate:  slt $t0, $s1, $s2           } Predicate ($t0 = 1 if true)
            bne $t0, $zero, doloop      } Branch Statement to continue loop
enddo:
```


By Joshua Cantrell
jjc@cory.berkeley.edu

The C *for* and MIPS

The *for* expression is like the while expression except it has two addition 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:

```
initialize: add $s1, $zero, $zero       # for ($s1 = 0,
            addi $s2, $zero, 10         #      $s2 = 10;
predicate:  slt $t0, $s1, $s2           #      $s1 < $2; $s1++)
            beq $t0, $zero, endfor      #    {
consequent: addi $s1, $s1, 0            #    $s1 += 0;
            addi $s1, $s1, 1            #                /* $s1 ++; */
            j predicate                 #    }
endfor:
```

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:

```
initialize: add $s1, $zero, $zero       } Initialization Statements
            addi $s2, $zero, 10         }
predicate:  slt $t0, $s1, $s2           } Predicate ($t0 = 1 if true)
            beq $t0, $zero, endfor      } Branch Statement to exit loop
consequent: addi $s1, $s1, 0            } Consequent (skipped if not true)
            addi $s1, $s1, 1            } Next Statements
            j predicate                 } Loop Back Statement
endfor:
```