Creating control structures

Any programs required testing variables and usually rely on different scenarios variable values depending. Different instructions are available such as *if*, *while*, *for*, *case* statements.

To create an if statement corresponding to:

```
if ( condition )
statement1;
else
statement2;
```

you have to write

```
{
if:
// Generate code to evaluate the condition, and
// branch to the label "then" if the condition is
// true or "else" if the condition is false;
then:
// Generate code for statement1;
br end;
else:
// Generate code for statement2;
end:
}
```

The label names are arbitrary.
Using the names "if", "then", "else" and "end" gives the appearance of a high-level control structure and make your code readable.

Examples

Assuming all variables already stored in memory, at starting addresses corresponding to label *a* and label *count*.

The following pseudocode

```
if ( a != 0 )
count = count + 1;
else
count = count - 1;
```

translate into

```
{
if:
ldiq $t0, a;      //$t0 <- label a address
ldq $t0, ($t0);   //$t0 <- variable a value
beq $t0, else;    //if-test on a
then:             //a != 0
ldiq $t0, count;  //$t0 <- label count address
ldq $t1, ($t0);   //$t1 <- variable count value
addq $t1, 1;
stq $t1, ($t0);   //count += 1
br end;
else:             //a==0
ldiq $t0, count;
ldq $t1, ($t0);
subq $t1, 1;
stq $t1, ($t0);   //count -= 1
end:
}
```

The "{ ... }" braces create a local "scope". The labels inside "{ ... }" can only be referred to inside "{ ... }".
The same identifiers for labels can be used in different control statements.

To create an if statement corresponding to

```
if ( condition )
statement1;
```

You need to write the following assembly pseudocode:

```
{
if:
// Generate code to evaluate the condition, and
// branch to the label "then" if the condition is
// true or "end" if the condition is false;
then:
// Generate code for statement1;
end:
}
```

Example: The following is a fragment of a C program.

```
if (a==b)
    d = 1;
else if (a > b)
    d = 2;
else if (a <= c)
    d = 3;
else
    d = 4;
```

Rewrite this program in assembly using branch instructions.

```
entry main.enter;

import "../IMPORT/register.h";
import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";
import "../IMPORT/io.lib.s";
import "../IMPORT/number.lib.s";
import "../IMPORT/string.lib.s";

block main uses proc {
```

```
data{
 // reserve locations for the variables
 align quad;
a:   quad 0x1;
align quad;
b:   quad 0x2;
align quad;
c:   quad 0x3;
align quad;
d:   quad 0;
}
code {
     public enter:
     ldiq $s0, a;
     ldq  $s0, ($s0);
     ldiq $s1, b;
     ldq  $s1, ($s1);
     ldiq $s2, c;
     ldq  $s2, ($s2);
     cmpeq     $s0, $s1, $t0; // if (a==b)
     beq  $t0, first;
     ldiq $s3, 1;        // d =1
     br   stop;

first:    subq $s0, $s1, $t0; // if (a > b)
     ble  $t0, second;
     ldiq $s3, 2;        // d =2
     br   stop;

second:   subq $s0, $s2, $t0; // if (a <= c)
     bgt  $t0, third;
     ldiq $s3, 3;        // d = 3
     br   stop;

third:    ldiq $s3, 4;        // d = 4
stop:
     ldiq $t0, d;
     stq  $s3, ($t0);
     }
}
```

Exercise:
Rewrite the previous program using only *cmp* instruction

While Statements

To create a while statement corresponding to

```
while ( condition )
statement1;
```

you have to write

```
{
while:
// Generate code to evaluate the condition, and
// branch to the label "do" if the condition is true
// or "end" if the condition is false;
do:
// Generate code for statement1;
br while;
end:
}
```

Consider:

```
result = 1;
i = 0;
while ( i < n ) {
result = result * a;
i++;
}
```

Suppose "result", "i", "n" and "a" are represented by registers $result, $i, $n and $a (create a new abs symbolic names section). Then you can write:

```
entry main.enter;
import "../IMPORT/register.h";
.
```

```
.
.
abs{
public i = t1;
public n = t2;
public a = t3;
public result = t4;
}
.
.
.
mov 1, $result;
clr $i;
{
while:
cmplt $i, $n, $t0; // i < n ?
blbc $t0, end;     //No
do:
mulq $result, $a; //result *= a
addq $i, 1;       // a+=1
br while;
end:
}
```

For Statement

To create a for statement corresponding to

```
for ( initialisation; condition; increment )
statement1;
```

We write

```
{
for:
// Generate code for initialisation;
while:
// Generate code to evaluate the condition, and
// branch to the label "do" if the condition is true
// or "end" if the condition is false;
do:
// Generate code for statement1;
continue:
// Generate code for the increment;
br while;
end:
}
```

Consider the same code as the above *while* loop:

```
result = 1;
for ( i = 0; i < n; i++ )
result = result * a;
```

The same code is generated, but with a couple of additional labels, to make it look more like a for loop.

```
mov 1, $result;
{
for:
    clr $i;
while:
    cmplt $i, $n, $t0;
    blbc $t0, end;
do:
    mulq $result, $a;
continue:
    addq $i, 1;
    br while;
end:
}
```

Break or continue statements inside the sub statement should be translated into *br end;* and *br continue;* respectively.

Switch statements may also be translated into assembly language.

The following pseudocode

```
switch ( expr ) {
case 0:
    stmt0;
break;
case 1:
    stmt1;
break;
case 2:
    stmt2;
break;
}
```

may be translated, using compare and branch instructions, into:

```
{
switch:
//Generate code to evaluate expression into $t0;
//Each value of the expression needs to be tested separately

cmpeq $t0, 0, $t1;
blbs $t1, case0;
cmpeq $t0, 1, $t1;
blbs $t1, case1;
cmpeq $t0, 2, $t1;
blbs $t1, case2;
...
case0:
Generate code to evaluate stmt0;
br end;
case1:
Generate code to evaluate stmt1;
br end;
case2:
Generate code to evaluate stmt2;
br end;
...
end:
}
```

Tips:

- Blbs and blbc instructions are used to check the result of a compare instruction
- Pseudocode loops using statements such as *for* or *while* may be producing the same assembly code
    - You must analyse your pseudocode before writing assembly programs
- Using braces helps you to limit the number of labels involved and then keep your code readable

Examples

How to increment the value of variable a initially set at 2.

```
entry main.enter;
import "../IMPORT/register.h";
import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";
import "../IMPORT/io.lib.s";

block main uses register {
data {
a: quad 0x02;
}
code {
public enter:
     {
     ldiq $t0, a;       // Get the address of a
     ldq $t1, ($t0);    // Get the value of a
     addq $t1, 1, $t2;       // Increment the value
     stq $t2, ($t0);    // Store the result back in a
     }
}
}
```

Assume that memory staring address is 0x1000000
Show memory contents after executing the program…

Isolating a byte in a quadword

This program intends to show you how to isolate a byte into a register.

```
entry main.enter;
import "../IMPORT/register.h";
import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";
import "../IMPORT/io.lib.s";

block main uses register {
data {
a: quad 0x1234567890abcdef;
}
code {
public enter:
     {
     ldiq $t0, a;       // Get the address of a
     ldq $t1, ($t0);    // Get the value of a
     sll $t1, 56, $t2;
     srl $t2, 56, $t3;  // This will isolate an unsigned byte
     srl $t2, 56, $t3;  // This will isolate a signed byte
     stq $t3, ($t0);    // Store the result back in a
     }
}
}
```

Assume that memory staring address is 0x1000000
Show memory contents after executing the program…

A little bit trickier

This program has no particular goal, just to show you how to use a few instructions or access a byte into a quadword…
Input this code into the simulator and check register and memory contents…

```
entry main.enter;
import "../IMPORT/register.h";
import "../IMPORT/callsys.h";
import "../IMPORT/proc.h";
import "../IMPORT/callsys.lib.s";
import "../IMPORT/io.lib.s";
block main uses register {
data {
a: quad 0x1010101002;
}
code {
public enter:
        {
        ldiq $t0, a;            // Get the address of a
        lda $t0, 4($t0);        //
        ldl $t1, ($t0);         // Load  a longword at the address
        sll $t1, 56;
        srl $t1, 56;
        addq $t1, 1, $t2;       // Increment the value
        stq $t2, ($t0);         // Store the result back in a
        }
}
}
```

Assume that memory starting address is 0x1000000
Show memory contents after executing the program…