Example: Use move, branch, addition, subtraction instructions to convert the C program below to an assembly program. In the assembly program, the variables in the C program should be represented as registers.

```
void main () {
  int a, b;
  a = 11;
  b = -17;
  while (a != 0) {
    if (a > b) a--;
    if (b <= 0) b++;
  }
}
```

```
code{
        mov     11, $S0;            // S0 is a
        mov     -17, $S1;           //S1 is b
while:
        beq $S0, end;               // while (a != 0)
do:
        {
        if:
            cmplt $S1, $S0, $T0;     //check b < a
            beq   $T0, end:
        then:
            subq $S0, 1; // a--
        end:
        }
        {
        if:
            bgt $S1, end;    // check b <= 0
        then:
            addq $S1, 1; // b++
        end:
        }
        br while;
end:
    }
```

A shorter way of coding the same function, which requires dropping the *if* control structure labels.

```
code{
        mov     11, $S0;          // S0 is a
        mov     -17, $S1;         //S1 is b
while:
        beq $S0, stop;            // while (a != 0)
do:
        subq $S0, 1, $T0;         // store a-- in T0
        addq $S1, 1, $T1;         // store b++ in T1
        subq $S0, $S1, $T2; // store a-b in T2
        cmovgt $T2, $T0, $S0;   // if (a-b>0), a=a--
        cmovle   $S1, $T1;    // if (b <= 0) b++
        br      while;
stop:
    }
```

Loop structure assembly skeleton

```
entry main.enter;

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

//   void main() {
//        while ( TRUE ) {
//            }
//        }
block main uses proc {
     code {
     public enter:
          {
          loop:
               br        loop;
          end:
               }
          }
     }
```

Strings

Strings are represents as a sequence of bytes. In C, the end of a string is indicated by a zero byte. If you use the same convention then, you can create string constants by using the `asciiz` directive.
For creating static strings you will use:

```
const {
align quad;
message1:
asciiz "Type some input: ";
align quad;
message2:
asciiz "The input was: ";
}
```

If you want to create new strings, you need to allocate space, using the byte directive.

```
data {
buffer:
byte [ BUFFERSIZE + 1 ];
}
```

The string is limited to a maximum length of BUFFERSIZE

- The extra byte being for the zero byte terminator.

The address of an element of a string can be accessed as the base address plus the index. To get the character at that address, you need an extra load.

- The load instruction to load a character (unsigned byte) is ldbu

```
ldiq $t0, buffer;
addq $i, $t0, $t0; //Gives the address of the ith element of buffer.
ldbu $t0, ($t0); // Give the value of the ith element of buffer.
```

Integer Arrays

Integer arrays can be created by declaring an array of quadwords, longwords, words, bytes depending on the size of integers.

To allocate space for an array (of size DATASIZE) of quadwords (8 * DATASIZE bytes), we do:

```
// long array[DATASIZE];
data{
align quad;
array: quad[DATASIZE];
}
```

The address of an element of an integer array of quadwords can be accessed as the base address plus 8 times the index. An extra load is needed to get the integer at that address.

```
ldiq $t0, array;
mulq $i, 8, $t1;
addq $t1, $t0, $t0; // Gives the address of the ith element.
ldq $t0, ($t0); // Give the value of the ith element.
```

Special support for array indexing is available through the instruction s8addq. The s8addq instruction is an operate instruction especially designed for indexing arrays of quadwords.

SxADDQ Ra, Rb, Rc x = 4 or 8

- S4addq, used to index arrays of longwords.
- S8addq, used to index arrays of quadwords.

It multiplies the first operand (the array index) by 8 (the size of a quadword), adds it to the second operand (the address of the array) and stores the result (the address of the appropriate element) in the third operand.

```
s8addq   s_reg1, s_reg2, d_reg
s8addq   $T0, $T1, $T2
```

```
T0 = 1, T1 = 0x100, T2 = T0*8+T1 = 0x108


s8addq   s_reg, value, d_reg
s8addq   $T0, 0x200, $T1

T0 = 2 T1 = T0*8+0x200 = 0x210

s8addq   d_reg/s_reg1, s_reg2
s8addq   $T0, $T1

T0 = 1, T1 = 0x100, T0 = T0*8+T1 = 0x108

s8addq   d_reg/s_reg, value
s8addq   $T0, 0x200

T0 = 2 T0 = T0*8+0x200 = 0x210
```

- It can be used to compute the address of an array element, given the index and base address.
- To get the value, we use a load instruction.

```
ldiq $t0, array;
s8addq $i, $t0, $t0; // Gives the address of the ith element.
ldq $t0, ($t0); // Give the value of the ith element.
```

Still, the addq instruction is used to index simple arrays of bytes.
For arrays with elements of size other than 1, 4, or 8, an explicit multiplication of the index by the size of the elements is needed.

- If the size of the elements is a power of 2, the multiplication can be done by a shift.

Exercise: Rewrite the previous assembly code using shift instruction instead of s8addq instruction.

Library functions in the simulator

In block Sys:
• long getChar(). Reads a character from the simple terminal.
• long putChar(char c). Writes a character to the simple terminal.
• void exit(). Causes the process to exit.

In block IO:
• void newline(). Prints a newline.
• void print(char *s). Prints a string.
• void error(char *s). Prints a string then exits.
• void readLine(char *s, long max). Reads a line of input into a buffer, and terminates the text with a null byte. Discards text that will not fit into the buffer.

In block Number:
• long fromString(char *buffer, long base). Converts the text in the buffer from the specified base into internal form. If base is 0, determines the base from the start of the text.
• char *toUnsigned(unsigned long value, long base). Converts the unsigned number into a base.
• char *toSigned(long value, long base). Converts the signed number into a base.
In block String:
• char *fromChar(char c). Creates a string containing the character c.
• long compare(char *s, char *t). Compares two strings and indicates their order by a value <, ==, > 0 depending on whether s < t, s == t, s > t.
• long length(char *s). Returns the length of the string s.
• void copy(char *s, char *t). Copies the string pointed to by t into the buffer pointed to by s.
• char *padLeft(char *s, char padChar, long fieldWidth). Pads s on the left with the pad character, to create a string of length fieldWidth.
• char *padRight(char *s, char padChar, long fieldWidth). Pads s on the right with the pad character, to create a string of length fieldWidth.

Special instructions

Special instructions have the form
```
opcode constant;
```

The only special instruction we will use directly is the call_pal instruction, with the operand CALL_PAL_CALLSYS.

```
call_pal CALL_PAL_CALLSYS
```

Essentially this instruction, with this operand causes the invocation of a function in the operating system. Additional information is passed in registers a0, a1, ... a5, to specify the request (in a0) and parameters to the request. The operating system passes the result back (if there is any) in register v0.

With the simulator:
Output a character

- Store in $a1 the character to output
- Specify the request (*put a character*) in $a0
- Call to the operating system (via call_pal instruction)

```
            ldiq $a0, CALLSYS_PUTCHAR;
            call_pal CALL_PAL_CALLSYS;
```

Input a character

- Specify the request (*get a character*) in $a0
- Call to the operating system (via call_pal instruction)
- The input character is stored in $v0

```
            ldiq $a0, CALLSYS_GETCHAR;
            call_pal CALL_PAL_CALLSYS;
```

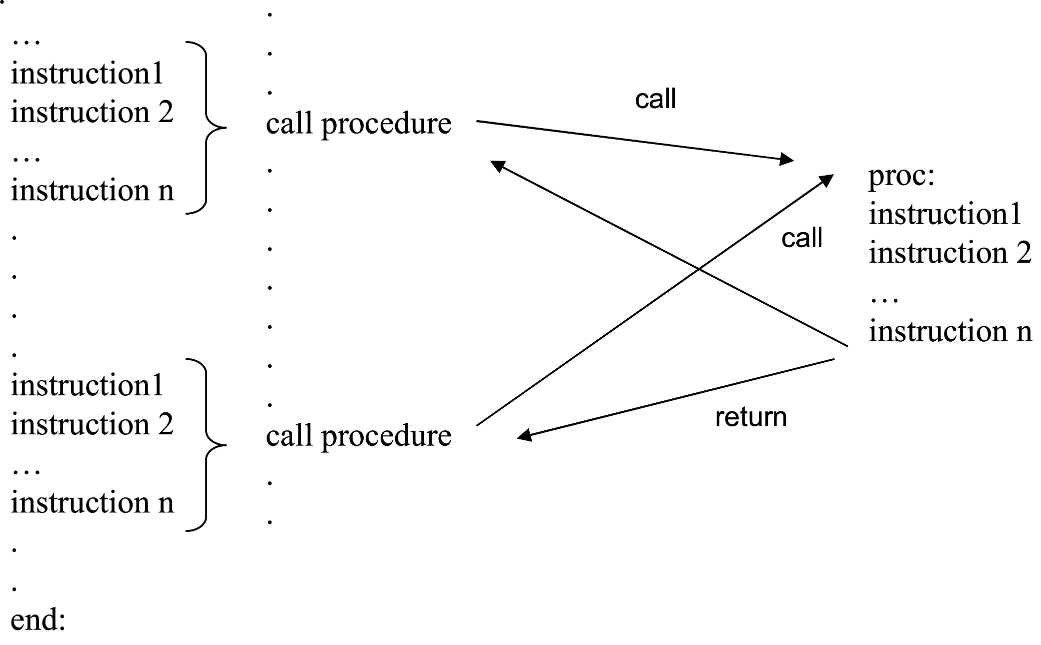
IO functions such as print, readline use these 2 special instructions.

Procedure

If a segment of code appears several times in a program, then the segment of code is normally written as a procedure and can be called from the program.

code{

public enter:



}

When a procedure is called, the machine starts executing the instructions in the procedure. A procedure is very similar to a program, It can access the memory locations as well as the registers.
When the execution of the procedure is completed, the machine returns to the main program and executes the instruction, which follows the procedure call instruction.
Different scenarios depending whether procedures require arguments or not, will return several values or will call other procedures or not.