For the invocation of a function:

- Generate code to evaluate the parameters and store them in a standard place.
  - On the Alpha, the first six parameters are stored in the argument registers a0, a1, ...a5.

- Generate a function invocation instruction that saves the program counter in a standard place and sets the program counter to the address of the start of the function.
  - On the Alpha, the bsr (branch to subroutine) instruction is used to invoke a function. The program counter is saved in the $RA (return address) register and set to the destination address.

- Generate code to use the result of the function, assuming the result of the function is stored in a standard place.
  - On the Alpha, functions return their result in the $V0 register.
  - If more than on result, use the $A registers.

Inside the function:

- You need to generate code to save the contents of any register you may use in the function, as long as you still want to access these values whenever the function is terminated.

- Generate code for the body of the function.
  - The code will access the argument values passed by the $A registers, modify local and global variables, and store the return value in a standard place. On the Alpha, the "saved" registers s0, s1, ... s5 are usually used for local variables, and the return value is stored in register v0.

- Generate an instruction return just after the bsr instruction.
  - On the Alpha, the *ret* instruction is used to return just after the invocation of the function.
  - The program counter is set to the contents of the $RA (return address) register.

Examples

To execute "x = f(5, 8, 2);", whatever f will do, and store the result into memory, we might write:

```
mov 5, $a0;  // a0 = 5
mov 8, $a1; // a1 = 8
mov 2, $a2; // a2 = 2
bsr f.enter; // call to function
ldiq $t0, x; // load into t0 the memory address where variable x is stored
stq $v0, ($t0); // store the value return by the function into x
```

Control is passed to the function by the execution of the bsr instruction.
Arguments passed to the function are stored in \$A0, \$A1 and \$A2.
On completion of the execution of the code for the function, control will be passed back to just after the bsr instruction.
\$V0 is used to return the result to the main program.

If the function f returns the sum of its three parameters, we might write:

```
// long f( long a, long b, long c ) {
// return a + b + c;
// }
block f uses proc {
code {
public enter:
addq $a0, $a1, $v0;
addq $v0, $a2;
ret;
}
}
```

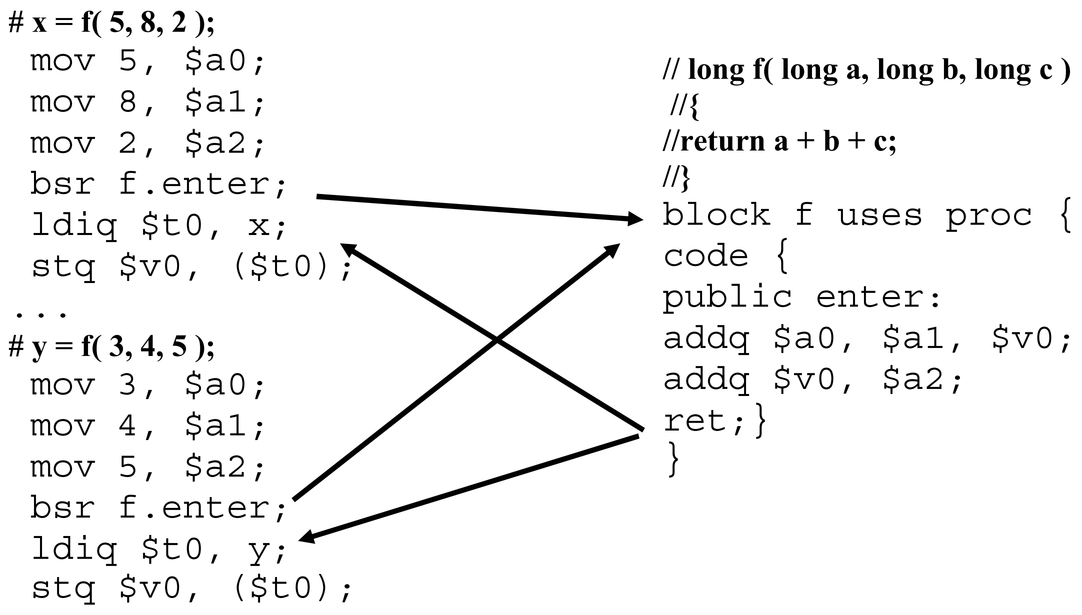
If the function is to be invoked only once in your program you could use branch instructions.

# x = f(5, 8, 2);
```
call:
     br f;
return:
ldiq $t0, x;
stq $v0, ($t0);
.
.
f:
     mov 5, $a0;
     mov 8, $a1;
     mov 2, $a2;
     addq $a0, $a1, $v0;
     addq $v0, $a2;
     br return;
...
```

If not:

- You need to remember where to return after completing the function.
- You need to pass values to the function.

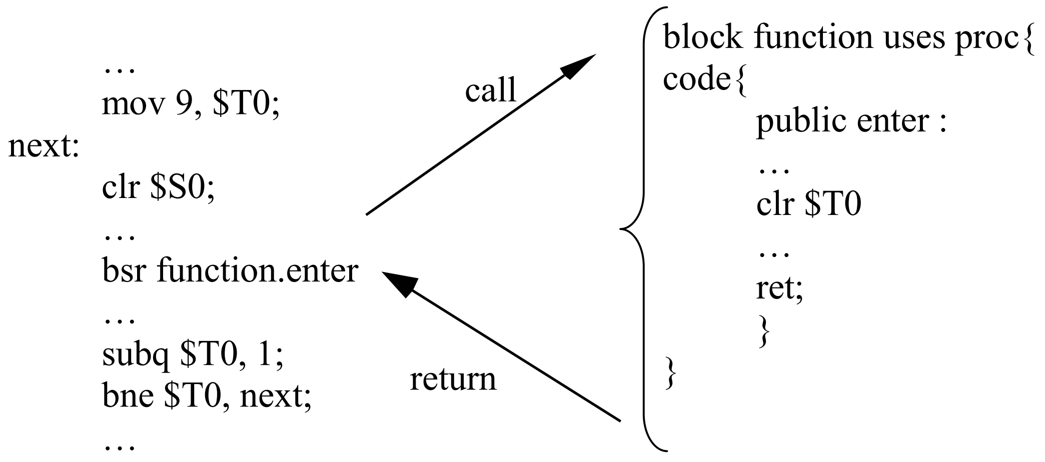
Thus we need to use the *bsr* instruction to enter and return from the function. Because a function may be invoked with different parameters, we need to evaluate the parameters and store them in the argument ($A) registers, rather than accessing them directly. <u>It's also safer.</u>
By convention, $v0 holds the result of the function.



# x = f(5, 8, 2);
```
 mov 5, $a0;
 mov 8, $a1;
 mov 2, $a2;
 bsr f.enter;
 ldiq $t0, x;
 stq $v0, ($t0);
...
```
# y = f(3, 4, 5);
```
 mov 3, $a0;
 mov 4, $a1;
 mov 5, $a2;
 bsr f.enter;
 ldiq $t0, y;
 stq $v0, ($t0);
```

// long f(long a, long b, long c)
//{
//return a + b + c;
//}
```
block f uses proc {
code {
public enter:
addq $a0, $a1, $v0;
addq $v0, $a2;
ret;}
}
```

Since a procedure can access registers, which are used by the main program, the values of the registers used by the main program might be changed while performing the procedure instructions:



```
       …
       mov 9, $T0;
next:
       clr $S0;
       …
       bsr function.enter
       …
       subq $T0, 1;
       bne $T0, next;
       …
```

```
block function uses proc{
code{
      public enter :
      …
      clr $T0
      …
      ret;
      }
}
```

This segment of a main program include a loop consisting in instructions from "`clr $S0`" to "`bne   $T0, next`".
The main program intends to execute the loop 9 times, while $T0 is not equal to 0.
When the procedure is called, the value of T0 is changed in the procedure.
When the main program resumes its execution, the value of $T0 is 0
The loop body in the main program will not be executed 9 times as expected.
To avoid the problem illustrated in this example, the value of the registers, which hold the values used by the main program need to be saved before the procedure, is executed.
When the procedure is completed, we restore the values of the registers needed by the main program. As a result, the registers in the main program have the values they had before the procedure was called.

- Alternative: You must ensure that no registers used in the main program will be modified when the function is called.