Reference parameters and Pointers

In C, the address of a simple variable, array, or record can be passed to a function. These are often called reference parameters. We can also have reference variables that point to an address of another variable. Variables of type corresponding to an array or class are really pointers to objects in memory.
By accessing the variable indirectly through the reference, we can modify its value.
For example, suppose we want to write a function that takes the address of two integer variables as parameters, and swaps their values. We could write:

```
void swap( long *a, long *b ) {
register long temp;
temp = *a;
*a = *b;
*b = temp;
}
```

Note: *a means the address pointed to by a. The parameters a and b are not altered, only the data at the addresses pointed to by them.
In assembly language, this is:

```
block swap uses proc {
abs {
a = a0;
b = a1;
}
code {
public enter:
ldq $t0, ($a);
ldq $t1, ($b);
stq $t1, ($a);
stq $t0, ($b);
ret
}
}
```

We can invoke the function swap(), with the addresses of variables as parameters. In C:

```
long x = 3, y = 4;
void main() {
swap( &x, &y );// &x means the address of x.
}
```

In assembly language:

```
block main uses proc {
data {
x: quad 3;
y: quad 4;
}
```

```
code {
public enter:
ldiq $a0, x; // a0 = &x;
ldiq $a1, y; // a1 = &y;
bsr swap.enter; // swap( &x, &y );
clr $a0; // exit( 0 );
bsr exit.enter;
}
}
```

Suppose we want to make x and y local to a function, then invoke swap. In C:

```
void f() {
long x = 3, y = 4;
swap( &x, &y );
}
```

x and y cannot be stored in registers, because only memory can have an address. They have to be stored in the activation record.

In assembly language:

```
block f uses proc {
local extends sav0 {
x: quad;
y: quad;
size:
}
code {
public enter:
lda $sp, -size($sp); // Allocate space
stq $ra, savRet($sp); // Save ra
mov 3, $t0; // long x = 3;
stq $t0, x($sp);
mov 4, $t0; // long y = 4;
stq $t0, y($sp);
lda $a0, x($sp); // swap( &x, &y );
lda $a1, y($sp);
bsr swap.enter;
ldq $ra, savRet($sp); // Restore ra
lda $sp, size($sp); // Deallocate space
ret
}}
```

Recursion

When writing recursive functions, we have to be particularly careful about saving and restoring registers. There will always be conflicts between the registers used by the invoking and the invoked function (**same function !!**).

```
// long fibo(long rank)
//      {
//      if (rank = 0)
//      return 0;
//      else if (rank = 1)
//      return 1;
//      else
//      return fibo(rank-1)+fibo(rank-2);
//      }
block Ast3{
public block fibon uses proc {
          abs{i = s0;
               tmp1 = s1;}
          code {
               public enter:
                    lda       $sp, -sav2($sp);
                    stq       $ra, savRet($sp);
                    stq       $s0, sav0($sp);
                    stq       $s1, sav1($sp);
               body:
                    mov $a0, $i;
                    cmple $i, 1, $tmp1;
                    blbc $tmp1, continue;
                    mov $i, $v0;
                    br end;
               continue:
                    subq $i, 1, $a0;
                    bsr Ast3.fibon.enter;
                    mov $v0, $tmp1;
                    subq $i, 2, $a0;
                    bsr Ast3.fibon.enter;
                    addq $v0, $tmp1;
                    mov $tmp1, $t5;//Debuging line
               end:
                    ldq       $s1, sav1($sp);
                    ldq       $s0, sav0($sp);
                    ldq       $ra, savRet($sp);
                    lda       $sp, +sav2($sp);
                    ret;
               }
          }
     }
```

Local Arrays

Functions declaring local arrays need to allocate space for the local array on the stack. Because the stack pointer no longer points to the base of the activation record, another register, the fp (frame pointer) register will be used to point to the activation record. The template for the assembly language for such functions should look like the following:

```
block f uses proc {
abs {
arrayPtr = ...; // A saved register
}
code {
public enter:
//-------------------------------------------------------
// Entry Code
//-------------------------------------------------------
lda $sp, -frameSize($sp);
stq $ra, savRet($sp);
stq $fp, savFP($sp);
stq $s0, sav0($sp);
stq $s1, sav1($sp);
...
mov $sp, $fp // Set frame pointer
init:
//-------------------------------------------------------
// Allocate space for the array
//-------------------------------------------------------
// Allocate array
lda $sp, -elementSize*arraySize($sp);
mov $sp, $arrayPtr;
//-------------------------------------------------------
// Body of function
//-------------------------------------------------------
body:
//-------------------------------------------------------
// Exit Code
//-------------------------------------------------------
end:
mov $fp, $sp; // Deallocate array
```

```
...
ldq $s1, sav1($sp);
ldq $s0, sav0($sp);
ldq $fp, savFP($sp);
ldq $ra, savRet($sp);
lda $sp, +frameSize($sp);
ret;
}
}
```

If the array has a fixed size, it is possible to allocate space for the array within the activation record.

However, the above system allows us to allocate arrays with a size that depends on the parameters passed to the function. If we have local variables in the activation record, they can be accessed by offsets from the frame pointer.

In the next example, there are a large number of local variables. Therefore, we will use the stack frame of the procedures to store the variables.
Since the next example involving arrays, first let's learn instruction `s8addq` which is very useful for handling array elements.

`s8addq` can have two or three operands.
The first operand must be a register.
During the execution, the first operand is multiplied by 8 and added to the second 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
```

Example 25: Convert the C program below to an assembly language program. In your conversion, (a) array `a` in `proc` must be allocated in the stack frame, and (b) `n` and `j` in `proc`, `m` in `main` should be mapped to registers.

```
//void main() {
//   int m;
//   m = proc(5);
//}
//int proc(int i) {
//   int a[20], n, j;
//   n = i;
//   for (j = 0; j<20; j++)
//       a[j] = j+2;
//   if (n == 1)
//       n = a[n]*n;
//   else
//       n = a[n]*n*proc(i-1);
//   return n;
}


entry main.enter
main:
     mov      5, $A0        # A0 is the parameter.
     jsr      proc          # proc(5)
     mov      $V0, $S0      # m = proc(5)
     call_pal CALL_PAL_STOP
     .end main
     .ent proc
block proc uses proc:
     lda $SP, -176($SP) #Create the stack frame
     stq  $RA, 0($SP) #Save the return address.
     mov  $A0, $S0 # n = i, i.e. n is mapped to S0
     clr  $T0          # j is mapped to T0
     lda  $T3, 16($SP)# 16($SP) starting
                           # address of the array
1:   cmplt    $T0, 20, $T1 #for(j=0;j<20;j++)
     beq      $T1, 2f      #   a[j] = j+2
     s8addq   $T0, $T3, $T4
     addq$T0, 2, $T5
     stq      $T5, 0($T4)
     addq$T0, 1
     br       1b

2:   cmpeq    $S0, 1, $T0  # if (n == 1)
```

```
    bne    $T0, base_case #if(n==1) go to base case
    subq$S0, 1, $A0       # i - 1
    stq      $S0, 8($SP)       # Save S0
    jsr      proc          # proc(i-1)
    ldq      $S0, 8($SP)       # Restore S0
    mulq$V0, $S0     #n=a[n]*n*proc(i-1)
    lda      $T1, 16($SP)
    s8addq   $S0, $T1, $T2
    ldq      $T3, 0($T2)
    mulq$V0, $T3
    br       end

base_case:
    lda      $T1, 16($SP) # n = a[n]*n
    s8addq   $S0, $T1, $T2
    ldq      $T3, 0($T2)
    mulq     $S0, $T3, $V0

end:
    ldq      $RA, 0($SP)
    lda      $SP, 176($SP)
    ret
    .endproc
```

Stack frame size = 20 array elements + 2 registers = 22*8 = 176