12 SUBPROGRAMS

"Subprograms" (also called "functions", "subroutines", etc.) are sequences of instructions that are "re-entrant". This means they can be placed in any part of memory without affecting their ability to perform their intended function. In particular:

1. They can be executed at any time by any other program from any point in the other program.
2. They can be loaded into memory at any place independent of the location of the other program.

Subprograms generally define a task that has general utility. That is, they define sequences of instructions that are likely to be used frequently.

The provision of a capability for defining subprograms is a desirable feature of any programming language. Subprograms improve the efficiency of memory use by permitting the same instructions to be used in a variety of contexts; that is, the algorithm defined by the subprogram can be used repeatedly by a main program without the need to duplicate the source code of the algorithm on each occasion.

Requiring that a program be re-entrant places constraints on how the program can be written:

- There should be no explicit references by the subprogram to address labels in any calling program.
- There should be no explicit references by the subprogram to any values in the calling program.
- There should be no references to the values in any registers used by the calling program.
- The subprogram should use only dynamic memory to meet any data storage requirements.

In assembly language programming, provision must be made for addressing explicitly a number of issues that permit a program that "calls" a subprogram to communicate with the subprogram:

- The calling program must provide values that will be needed by the subprogram to perform its calculations. These values are called *arguments*.
- The subprogram must provide access to the computed values. These values are called the *return values* of the subprogram.
- Temporary storage space may be required by the subprogram to perform its computation. This space is called *local space allocation*.
- The subprogram may need registers already being used by the calling program. The contents of these registers must be saved and restored later, before returning to the calling program. These steps are called "prolog steps".
- The calling program must provide the subprogram with the location of the instruction that should be executed when the subprogram has finished its task. This address is called the "return address".

In dynamic memory, a list called called a "stack frame" must be constructed to provide some or all of the storage space that will be required by a subprogram. A "stack frame diagram" illustrates what is required to account for all the data that will be transferred between a calling program and a given subprogram.

In the x86-64, some of the internal memory (registers) are used to share data between a calling program (caller) and a subprogram (callee). This improves the efficiency of execution since the access times for data stored in dynamic memory are longer that the access times for data stored in internal memory. Specifically:

Registers saved by the callee : Any of the following registers must be saved by the callee (subprogram) if they will be used in the source code of the subprogram: `rbx, rbp, r12, r13, r14, r15`. The contents of these registers are saved by storing them in stack memory with a `push` instruction.

Registers saved by the caller : The callee can assume that registers `r10, r11` will be saved by the caller and therefore can be used in the subprogram without saving their contents first.

Registers not saved : It is the responsibility of any program that uses `rax, rcx, rdx, rsi, rdi, rbp, r8, r9` to save these registers before a subprogram call if their contents will be required after returning from the subprogram call.

Subprogram Argument Registers : By convention the first six arguments of a subprogram call are stored in internal memory in the following order:

1st argument is in register `rdi`;

2nd argument is in register `rsi`;

3rd argument is in register `rdx`;

4th argument is in register `rcx`;

5th argument is in register `r8`;

6th argument is in register `r9`;

return value is stored in register `rax`.

If there are more than six arguments, they are pushed into stack memory. The occurrence of subprograms with more than six is relatively rare. If the subprogram is to return more than one value, space is allocated for additional return values in stack memory. Again, such occurrences are relatively rare.

The order in which the space in stack memory is allocated is important for it to be used successfully. Therefore locations in a stack frame diagram are allocated in the following order:

1. Before calling the subprogram, the calling program (caller):
   (a) Places the arguments 7 through N in the stack frame;
   (b) Allocates space for the return values if there is more than one;
   (c) Places the return address in the stack frame via the "`call`" instruction.

   The calling program now transfers control to the subprogram.

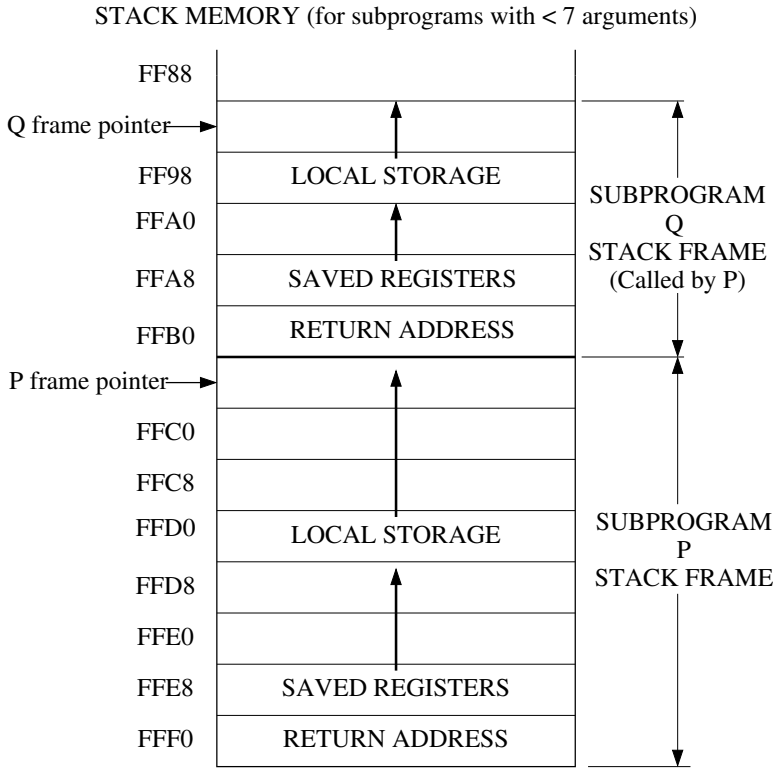
2. The subprogram:
   (a) Saves the state of all registers as required by convention. That is, it places the current values of all such registers into the stack frame;
   (b) Allocates space in the stack frame for any temporary local storage that may be required;
   (c) Determines the "frame pointer" (base address) of the stack frame;
   (d) Performs the instructions that define the algorithm, saving the result in register `tax` and any addition return values in the locations of the stack frame that were allocated for the additional return values;
   (e) Restores the state of the saved registers to the values that they held before the subprogram was executed;
   (f) Using the "`ret` instruction, branches back to the instruction in the calling program whose address was stored in the stack frame in step 1.d.

Within the subprogram, locations in dynamic memory will be accessed by computing "effective addresses." Therefore the displacement from the base address specified by the frame pointer to each location of the stack frame must be determined and shown on the stack frame diagram.

NOTE: Memory is allocated from "bottom to top" in a stack frame. That is, values are placed in stack memory in order of ever decreasing addresses, because whenever a "`push`" instruction is executed, the stack pointer is decreased, resulting in an address that is 8 bytes before the previous address.

Similarly, memory is recovered from a stack frame from "top to bottom". "Pull" instructions increase the stack pointer resulting in a new stack pointer address that is one more than the previous address.

A typical stack frame diagram for a subprogram with fewer than 7 arguments and at most one return value organizes the required information in the order shown on the next page. The initial value of the stack pointer, `rep` is shown as]`FFF0` for illustration purposes. However this is not the default value of the stack pointer in the x86-64.



NOTES:

1. The "frame pointer (FP)" is the address of the last value placed in dynamic memory when all the locations defined by the stack frame diagram are defined in memory.
2. All registers are 64-bit registers. Since only 8 bits can be stored at each location of memory, 8 bytes of memory are required to store any register.
3. The return address requires 8 bytes of memory because addresses are 64-bit binary sequences. They are stored in memory in the same way as the registers. The return address is placed in stack memory by the "`call <subprogram name>`" instruction. When executed this instruction also transfers control to the subprogram whose starting address is `<subprogram name>`.

   The return address is recovered from stack memory with the "`ret`" instruction. The value retrieved from stack memory during its execution is placed in the PC.

Because a subprogram cannot reference a symbolic address to access data, a different address mode is required than direct mode. Instead base+displacement mode can be used because this mode allows the programmer to treat the stack frame as a 1-d array and so access locations in the stack frame by a displacement from the frame pointer address.

By saving the value of the stack pointer in a register after the last value in the stack frame has been allocated to stack memory, that register becomes the "frame pointer", that is, the base address of an array. Using the stack frame diagram, it is possible to determine the displacement of a given value in the stack frame.

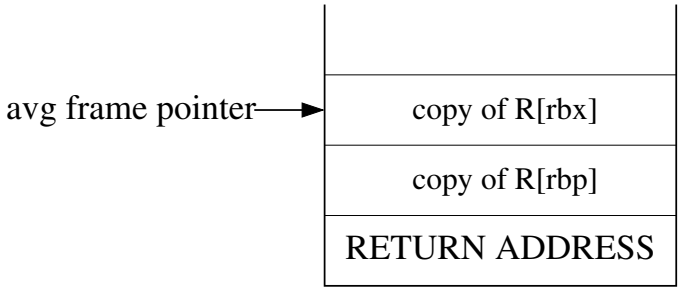
13 IMPLEMENTING A SUBPROGRAM

The following C source code describes a calling program to compute the average of two values using a function subprogram:

```
//Calling program (Caller):
#include <stdio.h>

void main(){
     unsigned x = 5;
     unsigned y = 7;
     unsigned z;
     z = avg(x, y);
     printf("z = %d", z);
}



// Subprogram (Callee):

unsigned avg( unsigned a, unsigned b){
     unsigned q;
     unsigned rl;
     q = (a + b) / 2;
     r =  (a + b) % 2;
     if (r ==  0) then
          return q;
     else
          return (q + 1);
}
```

Implementation of an equivalent assembly language subprogram consists of three steps:

STEP 1 : Construct the stack frame diagram. It is the responsibility of the callee to save `rbp` and `rbx` if they are to be used in the source code.



STEP 2 : Implementation of the Calling Program:

```
        .data
A:      .quad   5
B:      .quad   7
Z:      .quad

        .text
        .globl  main
main:   mov     $A, %rbx
        mov     0(%rbx),%rdi  #1st argument for avg
        mov     8(%rbx),%rsi  #2nd argument for avg
        sub     $8,%rsp       # Stack alignment
        call    avg          #return value in R[rax]
        add     $8, %rsp     #restore stack pointer
        mov     %rax, 16(%rbx)
        ret
```

STEP 3 : Implementation of the Subprogram:

Version 1:

```
        .text
        .globl  avg
        #
        # prolog (assumes r12 - r15 not used)
avg:    push    %rbp
        mov     %rsp, %rbp   # save stack pointer
        push    %rbx
        #
        # algorithm
        mov     %rdi, %rbx
        add     %rsi, %rbx
        mov     $10, %rcx
        mov     $0, %rdx     # dvdnd(127:64)
        mov     %rbx, %rax   # dvdnd(63:0)
        div     %rcx
        cmp     $0, %rdx     # rem = 0?
        je      skip
        add     $1, %rax     # rem = rem+1
        #
        # epilog (assumes r12 - r15 not used)
skip:   pop     %rbx
        mov     %rbp, %rsp   # restore stack pointer
        pop     %rbx
        ret
```

14 EXPANDING THE INSTRUCTION SET

While it is possible to implement most programs and subprograms using the instructions provided so far, one of the reasons for programming in assembly language is performance and there are instructions that are preferred over the ones provided so far because that improve performance. The following instructions enhance the existing instructions:

Shifting Instructions

INST	OPND(S)	SEMANTICS
shl	k, D	D ← D << k (fill = '0')
shr	k, D	D ← D >> k (fill = '0')
sar	k, D	D ← D >> k (fill = msb(D))

Another testing Instruction

INST	OPND(S)	SEMANTICS
test	S_1, S_2	ZF ← S_2 & S_1 SF ← msb(S_2 & S_1)

The following examples show how these instructions can improve performance:

Eliminating the `div` instruction :

Division is an extremely slow operation in almost any CPU. Therefore when it is possible to eliminate it, one should do so. For example, multiplication or division by any power of 2 can be done much more efficiently using shifting operations. To illustrate, suppose we wish to divide the value in register `rax` by 2^3. To do so using `div` requires the following code:

```
mov   $0, %rdx
mov   $8, %rcx
div   %rcx
```

Afterwards, `rax` will hold the quotient. However this can be more easily and efficiently done with:

```
shr   $3, %rax
```

Determining Divisibility by 2^n :

That is, is there a remainder after dividing the contents of a register by n? Suppose $n = 3$. One solution is:

```
mov   $0, %rdx
mov   $8, %rcx
div   %rcx
cmp   $0, %rdx
je    even
. . .
```

A better solution is:

```
test   $0x8, %rax
je     even
. . .
```

With the addition of these instructions, the subprogram fort "`avg`" can be rewritten as:

Version 2:

```
        .text
        .globl  avg
        #
        # prolog (assumes r12 - r15 not used)
avg:    push    %rbp
        mov     %rsp, %rbp
        push    %rbx
        #
        # algorithm
        mov     %rdi, %rbx
        add     %rsi, %rbx
        mov     %rbx, %rax
        shr     $1,%rax
        test    $1,%rbx
        je      skip
        add     $1,%rbx
        #
        # epilog (assumes r12 - r15 not used)
skip: pop     %rbx
        mov     %rbp, %rsp
        pop     %rbx
        ret
```

15 THE NEED FOR A PROLOG & EPILOG

The previous subprogram implementations required "prolog" and "epilog" instructions to build and remove the stack frame specified by the stack frame diagram in order to following the register saving conventions. However, the need for these instructions is dependent on the requirements of the subprogram:

- If the subprogram is a "leaf function" then the stack pointer, "`rsp`" does not need to be saved. A leaf function is one that does not call any subprograms.
- If any of the registers `rbx, rbp, r12, 13, 14, 15` are not used in the subprogram, then these registers to not need to be saved.
- if the subprogram is a leaf function then any registers among { `rax, rcx, rdx, rsi, rdi, rbp, r8, r9, r10, r11` }used by the subprogram do not need to be saved. Note that if a subprogram is not a leaf function then any register whose contents are required after a function call must be saved in stack memory before the "`call`" instruction and restored immediately afterwards.

With this in mind, analysis of the previous example confirms the following:

- The function "avg" is a leaf function. Therefore it is not necessary to save rsp
- Since the only role for rbp is to save rsp, rbp is no longer required and no longer needs to be saved.
- If rbx is used, then by convention, it must be saved. However, by replacing it with another register that does not need to be saved, the need to save rbx is eliminated.

With these observations, the function "avg" can be rewritten as:

Version 3:

```
        .text
        .globl  avg
        #
        #  no prolog
        # (leaf fn, rbp, rbx, r12 - r15 not used)
        #
        # algorithm
avg:    add     %rdi, %rsi
        mov     %rsi, %rax
        shr     $1,%rax
        test    $1,%sil
        je      skip
        add     $1,%rax
        #
        # no epilog
        #(leaf fn, rbp, rbx, r12 - r15 not used)
skip:   ret
```

16 PASSING ARRAYS AS ARGUMENTS

Passing an array of values as individual arguments is not feasible when the array is large. So an alternate method of communicating with the subprogram must be adopted: "call by reference."

Two types of argument passing are commonly employed when defining subprograms:

Call by Value : The actual value of the argument is passed to the Callee

Call by Reference : The address of the argument is passed to the Callee.

The example on the next page illustrates the use of call by reference to pass a 1-d array of ASCII character codewords that provide format information, to the C output function, "printf."

```
        .data
A:      .quad   5
B:      .quad   7
Z:      .quad
MSG:    .string "Average is...%d\n"

        .text
        .globl  main
main:   mov     $A, %rbx
        mov     0(%rbx),%rdi #1st argument for avg
        mov     8(%rbx),%rsi #2nd argument for avg
        sub     $8,%rsp      # Stack alignment
        call    avg          #return value in R[rax]
        add     $8, %rsp     #restore stack pointer

        mov     %rax, %rsi   #value to be printed
        mov     $MSG, %rdi   #format to e used
        mov     $0, %rax     #(indicates to printf that there are no floating point args)
        sub     $8, %rsp     #stack alignment
        call    printf
        add     $8, %Rsp
        ret
```