

RUN TIME ENVIRONMENTS IN COMPILER DESIGN

Supratim Biswas

Retired Professor, IITB & Visiting Professor BITM

sb@cse.iitb.ac.in

supratim.biswas@bitmesra.ac.in

Course Material (Revised Version Designed in 2018)

Re-Revised Version, April 2024 for CS333

April 2024



Department of Computer Science & Engineering

Indian Institute of Technology, Bombay

Birla Institute of Technology, MESRA

We shall give a quick review of the following before discussing the basic issues in the domain of runtime environments.

1. Intermediate Code Form

2. Architecture of x86 – 64 bit architecture : a brief review. The run time environment is better explained using examples from a standard architecture and this is the architecture which we are using for our experimentation in the laboratory.

Readers who are familiar with both of above may skip and directly go to Section 3 on Page 8.

1. INTERMEDIATE CODE FORM

The front end typically outputs an explicit form of the source program for subsequent analysis by the back end.

The semantic actions incorporated along with the grammar rules are responsible for emitting them.

Among the several existing intermediate code forms, we shall consider one that is known as Three-Address Code.

In three address code, each statement contains 3 addresses, two for the operands and one for the result. The form is similar to assembly code and such statements may have a symbolic label.

The commonly used three address statements are given below.

1. Assignment statements of the form :
 $x := y \text{ op } z$, where op is binary, or
 $x := \text{op } y$, when op is unary
2. Copy statements of the form $x := y$
3. Unconditional jumps, goto L. The three address code with label L is the next instruction where control flows.
4. Conditional jumps, such as
 if $x \text{ relop } y$ goto L
 if x goto L

The first instruction applies a relational operator, relop, to x and y and executes statement with label L, if the relation is true. Else the statement following if $x \text{ relop } y$ goto L is executed. The semantics of the other one is similar.

5. For procedure calls, this language provides the following :

param x

call p,n

where n indicates the number of parameters in the call of p.

6. For handling arrays and indexed statements, it supports statements of the form :

$x := y[i]$

$x[i] := y$

The first is used to assign to x the value in the location that is i units beyond location y, where x, y and i refer to data objects.

7. For pointer and address assignments, it has the statements

$x := \& y$

$x := * y$

$* x := y .$

In the first form, y should be an object (perhaps an expression) that admits a l-value. In the second one, y is a pointer whose r-value is a location. The last sets r-value of the object pointed to by x to the r-value of y.

A three address code is an abstract form of intermediate code. It can be realised in several ways, one of them is called **Quadruples**.

A quadruple is a record structure with 4 fields, usually denoted by op, arg1, arg2 and result. For example, the quadruples for the expression $a + b * c$ is

arg1	arg2	result	op	Equivalent form	
*	b	c		t_1	$t_1 = b * c$
+	a	t_1		t_2	$t_2 = a + t_1$

where t_1 and t_2 are compiler generated temporaries.

2. X86-64 bit Architecture at a Glance

- x86 assembly code has been popular for a long time.
- With replacement of 32-bit PCs with 64-bit ones, the assembly code has changed.
- x64 is a generic name for the 64-bit extensions to Intel and AMD architectures.

Registers :

16 general purpose 64-bit registers, the first 8 are named as RAX, RBX, RCX, RDX, RBP, RSI, RDI, and RSP.

- The second eight are named R8-R15. By replacing the initial R with an E on the first eight registers, the lower 32 bits can be accessed. For instance, EAX for RAX and similar for RAX, RBX, RCX, and RDX. The lower 16 bits are also accessible by dropping the prefix R, such as AX for RAX, etc.
- The lower byte of these registers are accessible by replacing the X for L, such as AL for AX. Similarly for the higher byte of the low 16 bits of X, such as AH for AX.
- The new registers R8 to R15 can be accessed in a similar manner, such as : R8 (qword), R8D (lower dword), R8W (lowest word), R8B or R8L. Note there is no R8H.

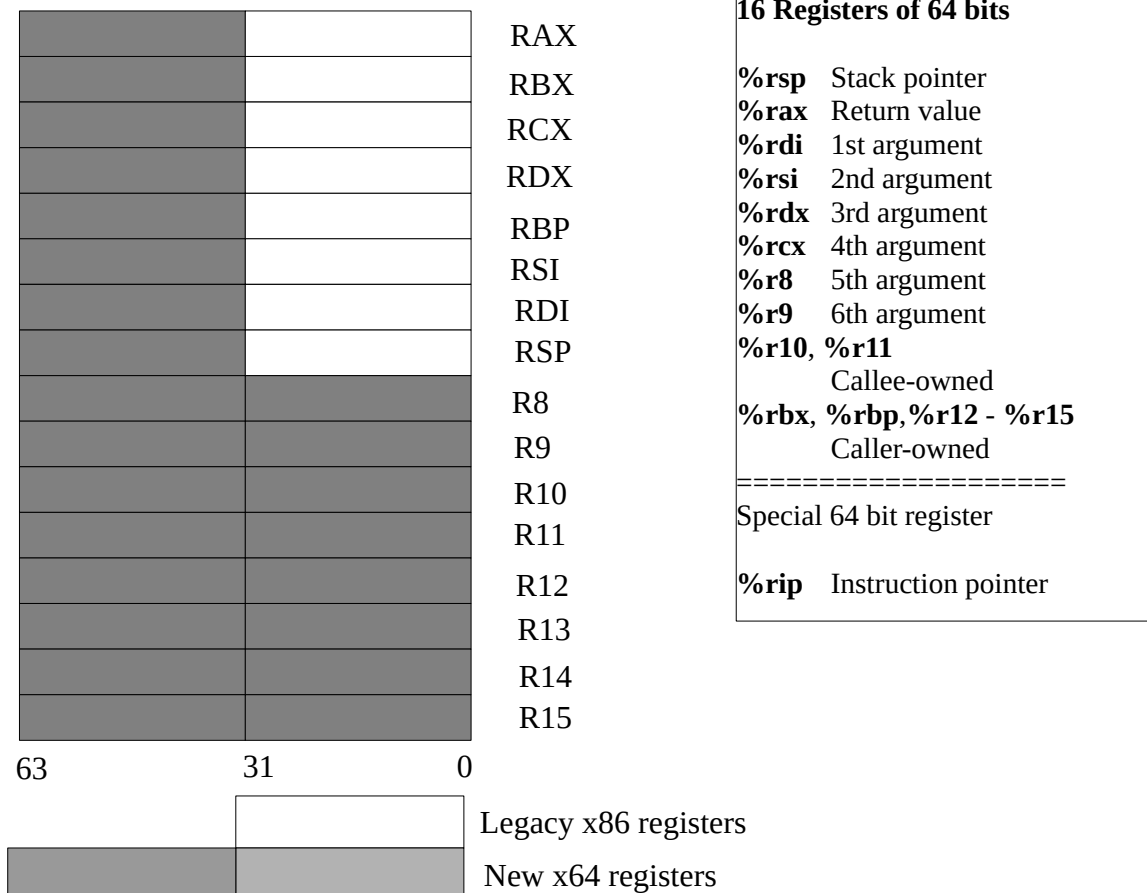
RIP is a 64-bit instruction pointer which points to the next instruction to be executed.

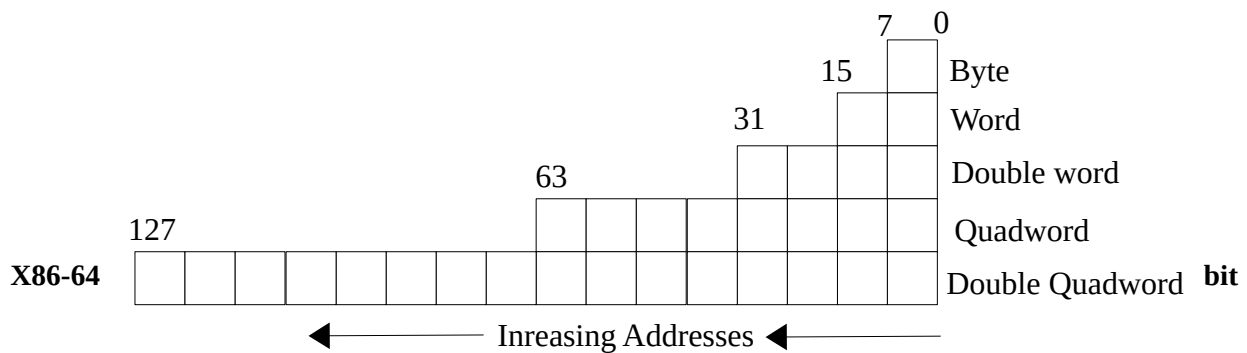
The stack pointer RSP points to the last item pushed onto the stack, which grows toward lower addresses.

The stack is used to store return addresses for functions, for passing parameters, saving registers, etc.

The registers and their usage conventions are listed here.

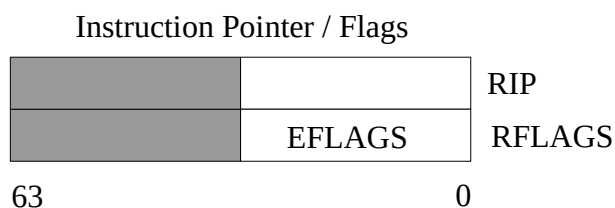
X86-64 bit Architecture





Architecture

The RFLAGS register stores flags used for results of operations and for controlling the processor. The useful flags are listed below.



Condition codes / Flags

CF	Carry Flag
OF	Overflow Flag
SF	Sign Flag
ZF	Zero Flag

Common Flags

Symbol	Bit	Name	Set when
CF	0	Carry	Operation generated a carry or a borrow
PF	2	Parity	Last byte has even number of 1's, else 0
ZF	6	Zero	Result was 0
SF	7	Sign	Most significant bit of result is 1
OF	11	Overflow	Overflow on signed operation

Common Assembly Instructions

Code	Operands	Semantics
mov	src, dst	# dst = src
movsbl	src, dst	# byte to int, signl-extend
movzbl	src, dst	
lea	addr, dst	# dst = addr
add	src, dst	# dst += src
sub	src, dst	# dst -= src
imul	src, dst	# dst *= src
neg	dst	# dst = - dst (arith inverse)
sal	count, dst	# dst <<= count (arith shift)
sar	count, dst	# dst >>= count (arith shift)
shr	count, dst	# dst >>= count (logical shift)

and	src, dst	# dst &= src
or	src, dst	# dst = src
Code	Operands	Semantics
xor	src, dst	# dst ^= src
not	dst	# dst = ~dst (bitwise inverse)
jmp	label	# unconditional jump to label
je	label	# jump on equal; ZF=1
jne	label	# jump on not equal; ZF=0
js	label	# jump on negative; SF=1
jns	label	# jump on not negative; SF=0
jg	label	# jump on > (signed); ZF=0 and SF=OF
jge	label	# jump on >= (signed); SF=OF
jl	label	# jump on < (signed); SF!=OF
jle	label	# jump on <= (signed); ZF=1 or SF!=OF
ja	label	# jump on > (unsigned); CF=0 and ZF=0
jb	label	# jump on < (unsigned); CF=1
cmp	a, b	# b-a, set flags
test	a, b	# a&b, set flags
push	src	# add to top of stack; Mem[--%rsp] = src
pop	dst	# remove top from stack; # dst = Mem[%rsp++]
call fn		# push %rip, jmp to fn
ret		# pop %rip

Instruction suffixes

b	byte
w	word (2 bytes)
l	long /doubleword (4 bytes)
q	quadword (8 bytes)

Suffix is omitted when it can be inferred from operands, %rax implies q, %eax implies l, etc.

Addressing modes : Operands to MOV instruction

Immediate

mov \$0x5, dst

\$val : source is **constant value**

Register

mov %rax, dst

%R: R is register; source in **%R**

Direct

mov 0x4033d0, dst

0xaddr : source read from **Mem[0xaddr]**

Indirect

mov (%rax), dst

(%R) : R is register; source read from **Mem[%R]**

Indirect displacement

mov 8(%rax), dst

D(%R); R is reg; D is displacement; source from **Mem[%R + D]**

Indirect scaled-index

mov 8(%rsp, %rcx, 4), dst

D(%RB,%RI,S); RB is register for base

RI is register for index (0 if empty); D is displacement (0 if empty)

S is scale 1, 2, 4 or 8 (1 if empty);

source from **Mem[%RB + D + S*%RI]**

3. Understanding the Key Tasks performed by a Compiler under Run Time Environment (RTE)

The compiler on completion of syntax and semantic analysis, has in a broad sense done the following two key tasks.

- Constructed a linked symbol table for all features that can define symbols in that language.
- Verified that the use of every symbol is valid and resolved this use to exactly one definition of that symbol. In case there are errors in definition or use of a symbol, error messages(s) are communicated and no further semantic analysis of the remaining code is done. However the compiler continues to perform syntax analysis of the remaining program, after the error point, to discover and report syntax errors therein.
- Generated intermediate code, wherever relevant, and has saved intermediate code for the entire source program, for further processing. The syntax and semantics of intermediate code, that is being used in this course, are reproduced in a separate section.
- The tasks of the compiler after successful syntax and semantic analyses, are
 - Optimization of the intermediate code, if this option is invoked by the user, and
 - Code generation for the target architecture. In this course, we assume the target code to the assembly language of a given architecture, namely x86 64 bit architecture. A quick review of this architecture is included as a separate section.
- Run time environment comprises of a series of activities undertaken by the compiler to prepare the intermediate code towards generation of assembly code.
 - Use the symbol table data to decide physical memory for data in the program.
 - Use the symbol table to decide the layout of memory for keeping the assembled code of all functions of the program.
 - Based on the compiler decisions of memory layout of data and code fragments, change the references to code and data in the intermediate code of the functions to correspond the assigned layout in the assembly code.
 - Function **call** and **return** require special handling to deal with the context changes when the transfer of control happens
 - from the code of the caller function (the function that makes a call) to the callee function (the function that is called) at the point of call in the body of the caller.
 - From callee function to its caller, when a return is encountered in the body of the callee.

These notes address all the issues mentioned above with illustrative examples. Note that

- we have chosen to use the combination of C language as source and X86 64 bit architecture as the target to explicate the issues in a concrete form without any guesses. This approach is expected to demystify this often underrated task of a compiler, and also provide good grasp about the generated assembly code.

- The RTE issues and solutions discussed in this document are however more general than the customization for C and x86 64bit architecture combination. The generic issues are pointed at relevant places in this document.
- With minor changes in the framework, the solutions discussed can be adapted to other combination pairs of (language, m/c). However one needs to know the language features and specifications for data and code and for function call and return.

Example 1 : Explore thorough experimentation the layout of data and code as done by gcc compiler for such features present in programs.

Source program “mem-layout.c”	Output on execution
<pre>int c = 4; static float val = 5.0; float f(float x) { return x*x - 2.0;} int main() { int a, b; a = a + b + c; int (*pm) () = &main; printf("&c = %p &a = %p &b = %p &val = %p\n", &c, &a, &b, &val); printf(" &main() %p \n", pm); printf(" f(val) = %f \n", f(val)); return 0; }</pre>	<pre>c : int global val : static float f : function with x : float argument returns float main : function returns int &c = 0x55944a1e2010 &a = 0x7ffc91b4ca88 &b = 0x7ffc91b4ca8c &val = 0x55944a1e2014 &main() 0x55944a1df18d f(val) = 23.000000</pre>

Let us interpret the addresses assigned by the compiler to data and code. The addresses when sorted in ascending order are (leading prefixes 0x are removed, since we know that all addresses are in hex representation only). The layout as done by gcc is depicted below.



The following may be noted in this context.

1. The code for main() occupies the lowest address among the rest. This is the starting address of the assembly code for main(). The right end of the address of main() will be decided by the size of code of main() [**Q. can you find out the size of main()’s assembly code in bytes ?**].
2. The global data c and the static data val reside at adjacent locations at a certain offset from the start address of main(). Note the gap between the start address of main() and c is “2e83” in hex which is quite close. **Q. Is the small gap accidental or intentional ?**
3. The gap in the memory between address of static **val** and local **a** of main() is huge.

The observations noted above are true for most languages. The following is the recommended memory layout for code / data at a generic level across levels and architectures.

INITIAL DIVISION OF MEMORY

The memory is usually divided into 4 parts initially.

1. **Code area**: Contains the generated target code, for example the code for functions, `f()`, `g()`, `main()`, etc.
2. **Static area** : Contains data whose absolute addresses can be determined at compile time. For example,
 - In C the addresses of globals, static data are kept in static area.
 - In languages like Pascal, the addresses of the local variables of the outermost procedure can be determined at compile time.
 - In language like Fortran IV, the addresses of all variables can be determined statically (recursion not permitted).
3. **Stack area** : For data objects of procedures which can have more than one incarnations active at the same time. The local data of a recursive function is an example.
4. **Heap area** : This is required for dynamically allocated data (like objects pointed to by pointer types). Since the sizes of the stack and the heap are not known at compile time, they are arranged to grow in opposite directions for effective utilization of space allocated to stack and heap.



The gaps between the different areas are also intentional.

- Keeping the code and global data in close proximity permits the compiler to share the instruction pointer, `%rip` for our architecture, (also known as program counter PC). The instruction pointer register, `%rip`, is a dedicated register used to hold the address of the next instruction to be executed in the code of the currently executing function.
- By allocating memory close to the code area, `%rip` may be used to access the globals by using the offset of the global from the start of code area. This is a read only use of `%rip` so it does not affect the basic role of `%rip`.
- The huge gap between {code, global} and local variables of a function is necessitated by the need to support functions with large body and global data of large sizes. Function call and return, specially recursive functions, are realized using a stack. All contemporary architectures provide a stack implemented in the hardware, to support recursive functions and associated issues related to parameter passing, local data, call and return.
- Most Languages of today, including C, support dynamically allocated data. The area of memory that is allocated for this purpose, to realize `malloc()`, `free()` and related functions, is known as a **Heap**. Since our sample example did not use dynamic data, address of an element in **heap memory** is not seen here. You can extend the given program to create a dynamic data and find out the part of memory that gcc uses to layout the heap.

4. ILLUSTRATION OF COMPILATION ISSUES THROUGH EXAMPLES

Example 1 – Simple Program in C

```
int a,b;           // global data
int main ()
{
    a = a + b;
}
```

\$ gcc -S -fverbose-asm run1.c

<pre>.file "run1.c" .globl a .data .align 4 .type a, @object .size a, 4 a: .long 4 .globl b .align 4 .type b, @object .size b, 4 b: .long 16 .text</pre>	<pre>.globl main .type main, @function main: .LFB0: pushq %rbp # movq %rsp, %rbp #, movl a(%rip), %edx # a movl b(%rip), %eax # b addl %edx, %eax # movl %eax, a(%rip) # %eax, a movl \$0, %eax # popq %rbp # ret Both a and b are given absolute addresses</pre>
--	---

Note that `%rip` is the instruction pointer; `a(%rip)` and `b(%rip)` show that the addresses of a and b are with respect to `%rip`; they are in the code area and hence absolute addresses.

Example 2 – Function with local variables

Source C Program : run2.c	Relevant Assembly Code
<pre>void f() { int a, b; // local variables to f() a = a + b; }</pre>	<pre>.file "run2.c" f: .LFB0: pushq %rbp # movq %rsp, %rbp #, movl -4(%rbp), %eax # b &b = %rbp - 4 addl %eax, -8(%rbp) # a &a = %rbp - 8; a = a + b nop popq %rbp # ret</pre>

a and b have been given relative address in the stack using %rbp

Note that %rbp is the base register that is a pointer to the stack denoting the area on stack allocated for a function.

The contents on the stack between %rbp and %rsp is the area allocated during the execution of a function call.

Example 3 : Function using both local and global data

Source program “run3.c”	Assembly code (relevant part)
<pre>#include <stdio.h> int a = 10, c; void f() { int a, b; a = a + b + c; }</pre>	<pre>.file "run3.c" f: .LFB0: endbr64 pushq %rbp movq %rsp, %rbp movl -8(%rbp), %edx movl -4(%rbp), %eax addl %eax, %edx movl c(%rip), %eax addl %edx, %eax movl %eax, -8(%rbp) nop popq %rbp ret</pre>

a and b have been given relative address in the stack using %rbp, while c has absolute address

Example 4 : Function using local and parameters

Source program "run4.c"	Assembly code (relevant part)
<pre>#include <stdio.h> void f(int x) { int a, b; a = b + x; }</pre>	<pre>.file "run4.c" .text f: .LFB0: endbr64 pushq %rbp movq %rsp, %rbp movl %edi, -20(%rbp) ## param x movl -8(%rbp), %edx ## b movl -20(%rbp), %eax addl %edx, %eax movl %eax, -4(%rbp) ## a nop popq %rbp ret</pre>

Example 5 : Function with multiple parameters and local variables

Source program "run5.c"	Assembly code (relevant part)
<pre>void f(int x, int y, int z, int*p) { int a, b; a = b + x + y + z + *p; }</pre>	<pre>.file "run5.c" f: .LFB0: endbr64 pushq %rbp # movq %rsp, %rbp #, movl %edi, -20(%rbp) # x, x movl %esi, -24(%rbp) # y, y movl %edx, -28(%rbp) # z, z movq %rcx, -40(%rbp) # p, p movl -8(%rbp), %edx # b, tmp86 movl -20(%rbp), %eax # x, tmp87 addl %eax, %edx # tmp87, _1 movl -24(%rbp), %eax # y, tmp88 addl %eax, %edx # tmp88, _2 movl -28(%rbp), %eax # z, tmp89 addl %eax, %edx # tmp89, _3 movq -40(%rbp), %rax # p, tmp90 movl (%rax), %eax # *p_10(D), _4 addl %edx, %eax # _3, tmp91 movl %eax, -4(%rbp) # tmp91, a nop popq %rbp ret .globl main</pre>

Source program “run5.c”	Assembly code (relevant part)
<pre> int main() { int n=10, m=11, p=12; int *q = &p; f(n, m, p, q); return 0; } </pre>	<pre> .type main, @function main: .LFB1: endbr64 pushq %rbp # movq %rsp, %rbp #, subq \$32, %rsp #, movq %fs:40, %rax # movq %rax, -8(%rbp) # tmp90, D.2333 xorl %eax, %eax # tmp90 movl \$10, -24(%rbp) #, n movl \$11, -20(%rbp) #, m movl \$12, -28(%rbp) #, p leaq -28(%rbp), %rax #, tmp85 movq %rax, -16(%rbp) # tmp85, q movl -28(%rbp), %edx # p, p.0_1 movq -16(%rbp), %rcx # q, tmp86 movl -20(%rbp), %esi # m, tmp87 movl -24(%rbp), %eax # n, tmp88 movl %eax, %edi # tmp88, call f # movl \$0, %eax #, _8 movq -8(%rbp), %rdi # D.2333, tmp91 xorq %fs:40, %rdi leave ret </pre>

Issue : **Q. How does f() refer to its formal arguments ? Q. Who supplies the actual argument to f() and how ?**

Example 6 : Global and Local Data and Parameters

Source program “run6.c”	Assembly code (relevant part)
<pre> #include <stdio.h> int a; short b; float x[16]; double y[20]; void f(int z) { int x, y; x = x + y; z = x + z; a = a + b; b = b + z; } </pre>	<pre> .file "run6.c" f: .LFB0: endbr64 pushq %rbp # movq %rsp, %rbp #, movl %edi, -20(%rbp) # z, z movl -4(%rbp), %eax # y, tmp91 addl %eax, -8(%rbp) # tmp91, x movl -8(%rbp), %eax # x, tmp92 addl %eax, -20(%rbp) # tmp92, z movzwl b(%rip), %eax # b, b.0_1 movswl %ax, %edx # b.0_1, _2 movl a(%rip), %eax # a, a.1_3 addl %edx, %eax # _2, _4 movl %eax, a(%rip) # _4, a movl -20(%rbp), %eax # z, tmp93 movl %eax, %edx # tmp93, _5 movzwl b(%rip), %eax # b, b.2_6 addl %edx, %eax # _5, _8 movw %ax, b(%rip) # _9, b nop popq %rbp # ret </pre>

Example 7 : Accessing array variables

Source C program	Assembly Code
<pre>void p() { int a[6], b, i; for (i = 0; i < 6; i++) b = b + a[i]; }</pre>	<pre>p: .LFB0: pushq %rbp # movq %rsp, %rbp #, subq \$48, %rsp #, movq %fs:40, %rax # movq %rax, -8(%rbp) # xorl %eax, %eax # movl \$0, -36(%rbp) # i jmp .L2 # .L3: movl -36(%rbp), %eax # i cltq movl -32(%rbp,%rax,4), %eax addl %eax, -40(%rbp) # b addl \$1, -36(%rbp) #, i .L2: cmpl \$5, -36(%rbp) # i jle .L3 # nop movq -8(%rbp), %rax xorq %fs:40, %rax # je .L4 # .L4: leave ret</pre>

Note that p() does not have any arguments. Note how the loop has been translated.

1. Initialization $i = 0$: `movl $0, -36(%rbp) # i`
2. Test condition with code at .L2
3. Body of the loop starts at label .L3

Example 8 : Recursive Function

Source program “run7.c”	Assembly code (relevant part)
<pre>int f(int x) { int a; if (x == 0) return 1; { a = f(x-1); return (x*a); } }</pre>	<pre>.file "run8.c" f: .LFB0: endbr64 pushq %rbp # movq %rsp, %rbp #, subq \$32, %rsp #, movl %edi, -20(%rbp) # x, x cmpl \$0, -20(%rbp) #, x jne .L2 #, movl \$1, %eax #, _2 jmp .L3 # .L2: movl -20(%rbp), %eax # x, tmp85 subl \$1, %eax #, _1 movl %eax, %edi # _1, call f # movl %eax, -4(%rbp) # tmp86, a movl -20(%rbp), %eax # x, tmp87 imull -4(%rbp), %eax # a, _2 .L3: leave ret .LC0: .string " n = %d n! = %d \n" .text .globl main .type main, @function main: .LFB1: endbr64 pushq %rbp # movq %rsp, %rbp #, subq \$16, %rsp #, movl \$7, -4(%rbp) #, num movl -4(%rbp), %eax # num, tmp85 movl %eax, %edi # tmp85, call f # movl %eax, %edx #, _1 movl -4(%rbp), %eax # num, tmp86 movl %eax, %esi # tmp86, leaq .LC0(%rip), %rdi #, movl \$0, %eax #, call printf@PLT # movl \$0, %eax #, _6 leave ret</pre>

Q. Identify the compilation of the call statement in the source:

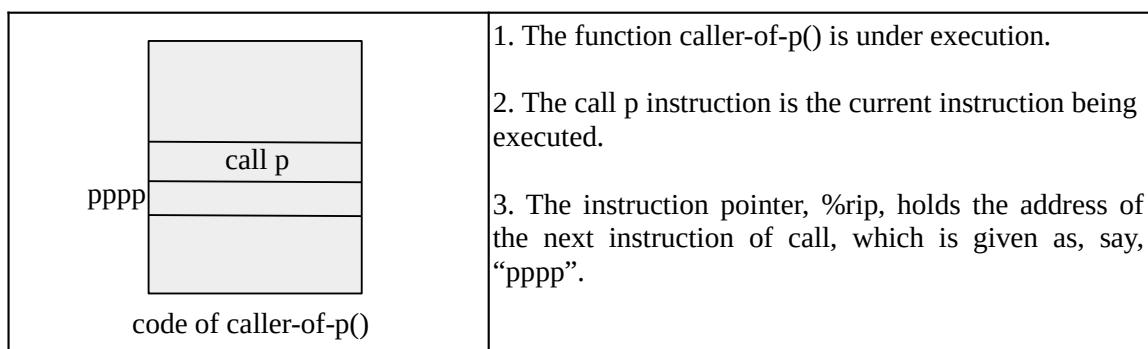
Example 9 : The program fragment in this example, has a global struct definition followed by a function p(). The function declares an object x of the struct rec and a pointer y to struct rec. The assembly code, after stripping off parts that are not directly relevant is shown in column 2.

Source program “run8.c”	Assembly code (relevant part)
<pre>#include <stdio.h> #include <stdlib.h> typedef struct list { int data; struct list* next; }rec; void p() { rec x; rec*y; y = (rec*) malloc(sizeof(rec*)); x.next = y; }</pre>	<pre>.file "run8.c" p: .LFB6: endbr64 pushq %rbp # movq %rsp, %rbp #, subq \$32, %rsp #, movl \$8, %edi # arg 1 call malloc@PLT # movq %rax, -24(%rbp) # tmp82, y movq -24(%rbp), %rax # y, tmp83 movq %rax, -8(%rbp) # tmp83, x.next nop leave ret</pre>

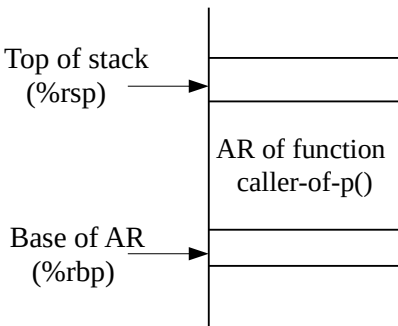
We have seen the data layout for various data that are defined and referred to in a program. However for complete understanding of the procedure call and return mechanism, we shall open up the execution of the given source through a blow by blow account as the execution proceeds at run time. However, it may be noted that while actions are taken at run time, the code that executes at run time have been carefully inserted by the compiler at the compilation time itself. The ensuing discussion should be helpful in understanding this non-trivial task performed by the compiler.

The purpose and presence of the run time environment related aspects in the assembly code are explained below. We make the following assumptions to complete the story.

- There will some function that calls p(), let us name it the “caller-of-p()”. Execution control gets transferred to p() from “caller-of-p()”, because of a call statement in the body of “caller-of-p()”. A possible scenario for this to happen is drawn below.



- The compiler allocates a certain amount of space on the stack for a function to deal with its data and other associated information. The extent of space allocated depends on the size of its local data, parameters and such information. In fact every information of a function, except its code, are on the stack. This space on the stack is referred to as the Activation Record (AR) of the function.

AR of caller-of-p()	
 <p>Top of stack (%rsp) →</p> <p>Base of AR (%rbp) →</p> <p>AR of function caller-of-p()</p>	<p>The AR of a function is addressed by two separate pointers,</p> <ol style="list-style-type: none"> 1. base pointer of the function in AR, usually kept in a dedicated register, for us it is %rbp 2. The other extent of the AR is denoted by another dedicated register, which is %rsp for us. 3. While most information related to a function lies in the AR region between its %rbp and %rsp, a function may access the stack area below its AR also, if required, depending on the communication protocol between the caller and callee functions.

- The AR is created when a function call is made and remains in force till the call is active. When the function returns from the call, its space on the AR is no longer valid, and for all practical purposes, it may be assumed that this space has been deallocated and no longer accessible.
- The division of responsibility between the functions, the caller and the callee, has to be clearly specified in order to take care of the multiple tasks, such as
 - creation of AR
 - transfer control from caller to callee
 - enable passing of parameters from caller to callee
 - access return value sent from callee to caller
 - transfer of control from callee back to caller

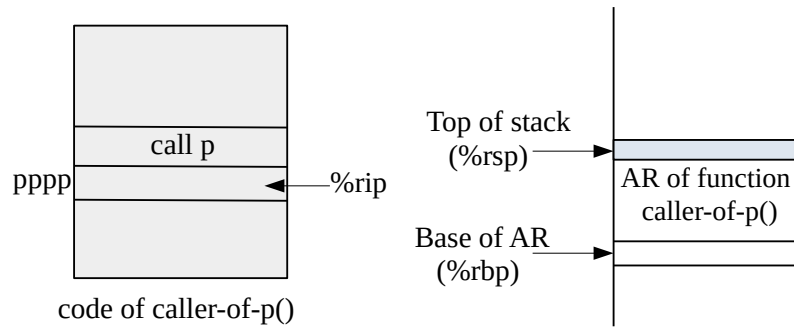
The division depends primarily on the protocols defined and agreed between designers of architecture and compiler writers.

- We study the protocol for x86 64 bit architecture and C language designers / compiler writers. The approach taken by us is to understand how gcc performs this task in complete detail. Therefore we examine assembly code generated by gcc for various simple C program fragments so as to comprehend the issues involved.
- The call instruction involves the following two actions.

call p	<ol style="list-style-type: none"> 1. Save the return address in the code of the caller, which is the address of the instruction immediately after the call in the AR 2. Transfer control from caller-of-p() to the first instruction in body of p() 	<ol style="list-style-type: none"> 1. push %rip 2. jmp @p
--------	--	---

- Let us trace the actions that are taken at execution time from the point the call is made from within the body of caller-to-p() to p() and continue till the call from p() returns to its caller.

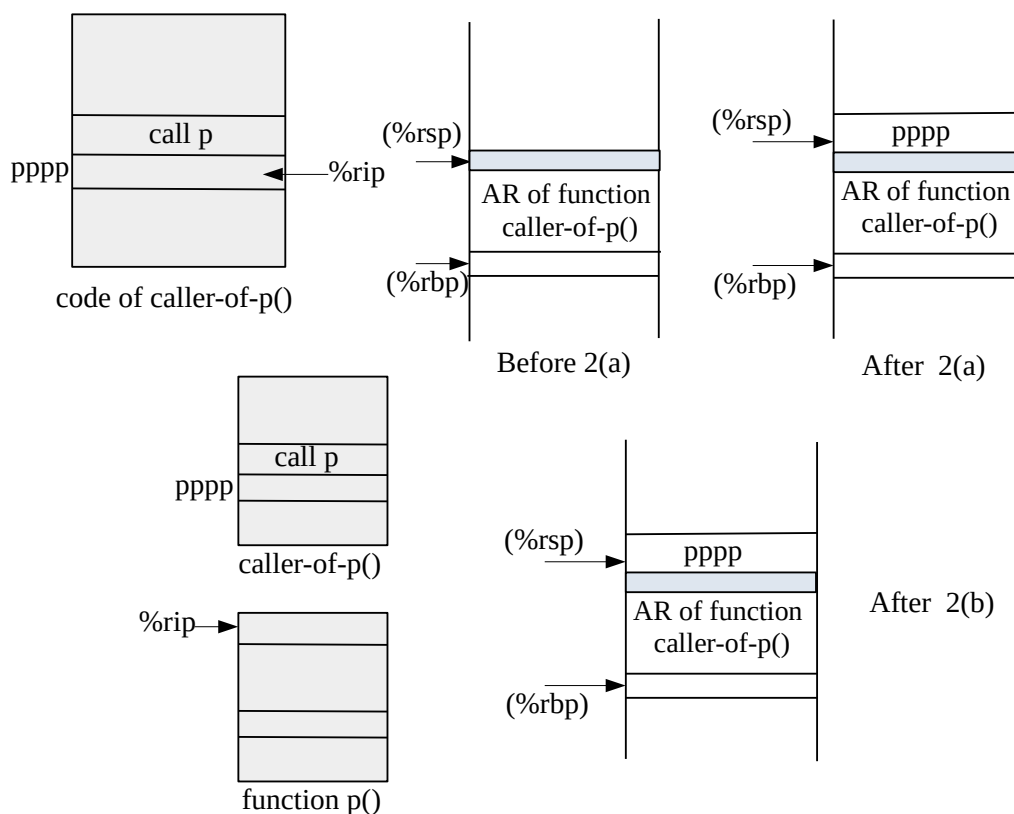
1. The starting point is when the call instruction is getting executed in caller-to-p().



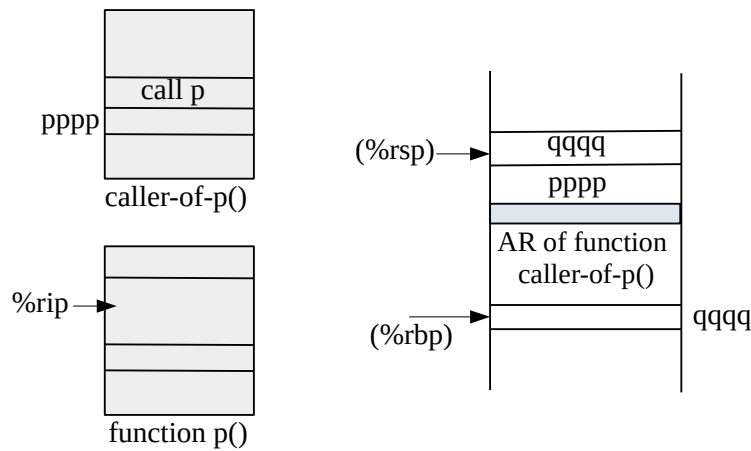
2. (a) The first part of call instruction is to save the return address on the stack, for our architecture the instruction is : pushq %rip

(b) The 2nd part is to transfer control to the first instruction of the callee function, p(); this is achieved by jmp @p

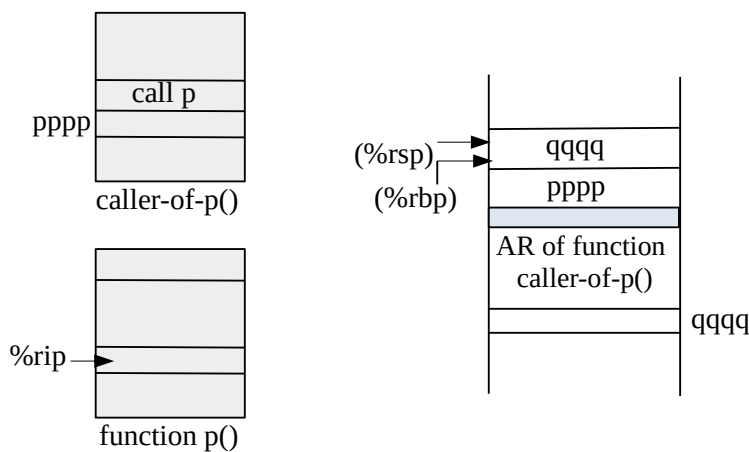
The changes in the code area and ARs on the stack after execution of each instruction is shown in the figures that follow.



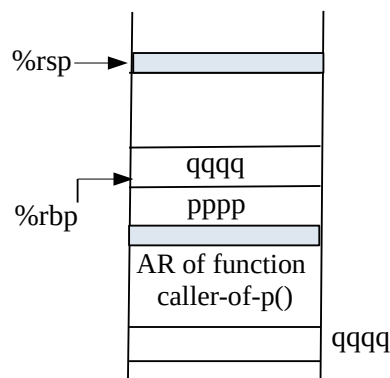
3. Since the `%rip` has been moved to the code of `p()`, the 1st instruction in the code of `p()`, that is “`endbr64`” is executed. This instruction is introduced by Intel as a security measure and hence skipped. The next instruction, “`pushq %rbp`” is then executed. The changed contents are shown below.



4. The next instruction in `p()` is “`movq %rsp, %rbp`”. The changed configuration follows.



5. The instruction to be executed at this point is : “`subq $32, %rsp`”. In our architecture, the stack grows from higher addresses to low addresses, hence this instruction moves `%rsp` by 32 bytes upward. The objective is to create space for locals of `p()`, as shown below by depicting the stack only.



6. “movl \$8, %edi” is the next instruction. The 1st executable statement in the source of p() and the corresponding assembly instruction are highlighted in the following table. The size of a pointer is 8 bytes and the compiler is preparing to call malloc(8).

Source program “run8.c”	Assembly code (relevant part)
<pre>void p() { rec x; rec*y; y = (rec*) malloc(sizeof(rec*)); x.next = y; }</pre>	<pre>movl \$8, %edi # arg 1 call malloc@PLT # movq %rax, -24(%rbp) # tmp82, y movq -24(%rbp), %rax # y, tmp83 movq %rax, -8(%rbp) # tmp83, x.next nop leave ret</pre>

In the intermediate code this call statement in C would be translated to

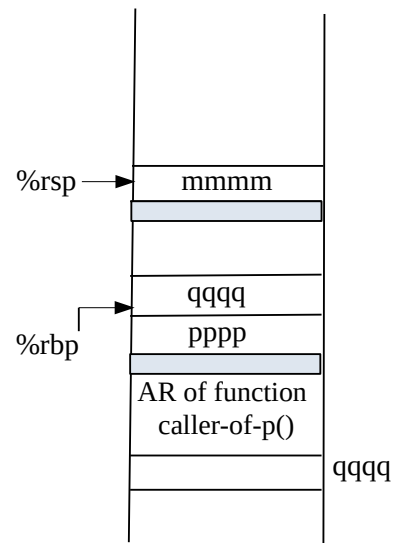
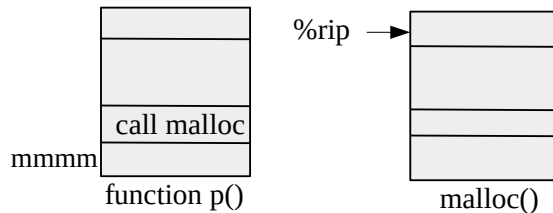
C Source	IC	Assembly
y = (rec*) malloc(sizeof(rec*));	param 8 call malloc	movl \$8, %edi # arg 1 call malloc@PLT #

Recall argument passing conventions in our architecture : first 6 actual arguments of the callee function are passed by the caller using 6 specific registers. If the callee has more than 6 arguments, the the caller places the remaining arguments on the stack for the callee to access them from its AR. The use of %rdi to pass first argument, value 8, is consistent with this architecture.

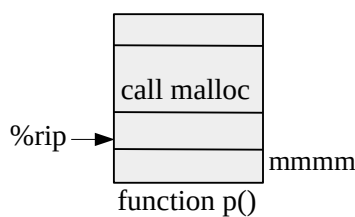
Argument	Register used to pass	Example 9
1	%rdi	8
2	%rsi	
3	%rdx	
4	%rcx	
5	%r8	
6	%r9	

7. After setting up the lone actual argument, p() now calls malloc(), which is a library call in C, the call is serviced using definitions present in “stdlib.h”. The changed configuration follows.

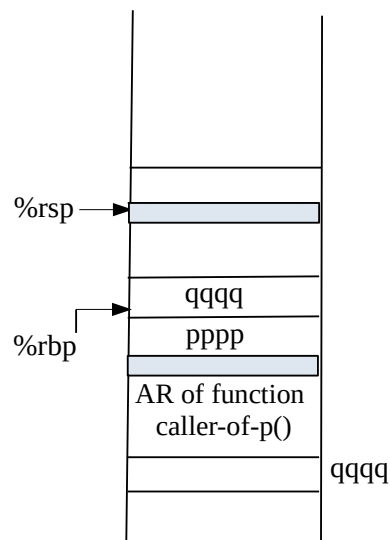
Argument	Register used to pass	Example 9
1	%rdi	8
...



The code of malloc(), though not seen by us, because of calling convention, has the same protocol. However since our focus is on the body of p(), the scene immediately after malloc() returns control back to p() needs to be drawn.



Argument 1	%rdi
Return value	%rax



8. In the above configuration, p() executes the instruction : **movq %rax, -24(%rbp)**. We need some clarification about the operand “-24 (%rbp)”. It should be obvious that this area is in the AR of p(), since %rsp was moved by -32 in an earlier instruction. The immediate question is about the local variables of p(), namely, rec x and rec*y. The object x needs 12 bytes to hold an int and pointer, while y needs 8 bytes. The order of the locals are : {x.data, x.next, y} and they have to be in the AR of p() on the stack.

It is easy to see from the assembly code that address(x.next) is -8(%rbp) and -24(%rbp) respectively. Though not given in the assembly code, because we have not accessed x.data, address(x.data) is -4(%rbp). These details can be now be shown on the AR of p(). Also the purpose of the instruction of “movq %rax, -24(%rbp)” is to save the return value of malloc(), which is a 8 byte pointer, in the location of y on the stack. The gaps in space that we often find between consecutive data on

stack is because of padding used by the compiler to use double word address boundaries to access data.

Reader : Draw the configuration here with relative addresses of locals.

9. The next useful instruction is “leave”.

leave	1.Set the stack pointer to that of caller 2. Set the base pointer to that of caller function	1. movl %rbp, %rsp 2. popq %rbp
--------------	---	------------------------------------

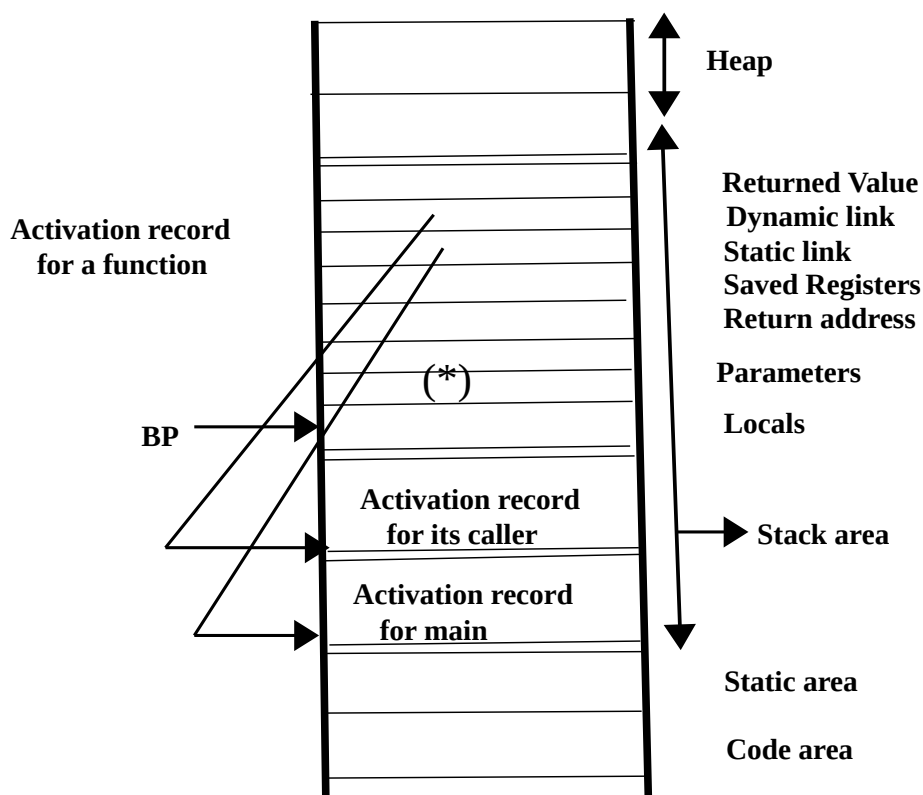
Reader : Draw the configuration at this point in time

10. The last instruction in the code of p() is “ret”.

ret	1. Restore instruction pointer to that of the caller function	1. popq %rip
------------	---	--------------

Reader : Draw the Final configuration here

Now that we have a good insight into the Run Time Environment issues that arise in the context of C language and x86 64bit architecture, the generic issues in RTE for most combinations of PL and an architecture are briefly outlined.



This is one of many possible organizations of an activation record.

ACCESSING INFORMATION IN ACTIVATION RECORD

Within an activation record, any information is referred to using an offset relative to a pointer (BP) pointing to the base of the activation record. Thus the address of the local variable y is BP (contents of register BP – offset for y) and the parameter x often has the address (contents of BP + offset) in a possible activation record layout for a function.

Some machines use a dedicated register for holding the base pointer. Actual contents of the activation record depends on the programming language and the machine architecture.

PARAMETER PASSING

The common methods to associate an actual parameter with a formal parameter are:

1. **Call by value** : The calling procedure evaluates the actual parameter and places the resulting value in the activation record of the called procedure.
2. **Call by reference** : The actual parameter is a variable. The calling procedure places the address of this variable in the activation record of the called procedure. A reference to the formal parameter, then, becomes an indirect reference through the address passed to the called procedure.

There are other methods of parameter passing that have been defined in different languages and also successfully implemented.

In this course, we shall consider the above two parameter passing methods only.

COMPILING PROCEDURE CALLS

During a procedure call, there are many possible ways of dividing the bookkeeping work between the caller and the callee.

One possible way is:

During a procedure call

Tasks performed by Caller

1. Makes space on the stack for a return value.
2. Puts the actual parameters on the stack.
3. Sets the static link. **[Not relevant for C/C++: May be ignored]**
4. Jumps to the called procedure. Return address is saved on the stack.

Tasks performed by Callee

1. Sets the dynamic link.
2. Sets the base of the new activation record.
3. Saves registers on the stack.
4. Makes space for local variables on the stack.

COMPILING PROCEDURE CALLS

Note that if we follow this sequence, the base pointer does not actually point to the base of the activation record, but somewhere in the middle. The actual parameters are now referenced using a negative offset, and the local variables with a positive offset.

During a return

Tasks performed by Callee

1. Restores registers.
2. Sets the base pointer to the activation record of the calling procedure.
3. Returns to the caller.

Tasks performed by Caller

1. Restores stackpointer to the location containing the returned value.

COMPILING PROCEDURE CALLS

X86-64 BIT ARCHITECTURAL SUPPORT FOR COMPILATION

During a procedure call : *Tasks performed by Caller*

1. Makes space on the stack for a return value / use a dedicated register

mov \$0, %eax

2. Passing actual arguments to callee

- First 6 arguments through dedicated registers :

movl -64(%rbp), %rcx # arg4

movl -68(%rbp), %rdx # arg3

movl -56(%rbp), %rsi # arg2

movq -16(%rbp), %rdi # arg1

- The remaining arguments are saved on the stack

movl -44(%rbp), %edi # arg9

pushq %rdi #

movl -48(%rbp), %edi # arg8

pushq %rdi #

movl -52(%rbp), %edi # arg7

pushq %rdi #

3. Sets the static link.

Not required for C / C++

4. Jumps to the called procedure. Return address is saved on the stack.

call f

Equivalent to **pushq %rip**

jmp \$ 0x0604420 OR **addl \$0x0200, %rip**

depending on whether address of f() is referred using an absolute address, such as **\$0x0604420** or a relative address, **\$0x0200**, with respect to %rip

During a procedure call : Tasks performed by Callee

1. Sets the dynamic link.

pushq %rbp // pushes the base pointer on the stack

2. Sets the base of the new activation record.

movq %rsp, %rbp //Sets the base pointer to the top of the stack

3. Saves registers on the stack.

If caller has passed arguments through dedicated registers, then these are saved by the callee on the stack.

movl %edi, -20(%rbp) # save arg1 on stack

movl %esi, -24(%rbp) # save arg2 on stack

movl %edx, -28(%rbp) # save arg3 on stack

movq %rcx, -40(%rbp) # save arg4 on stack

On some architectures, there are special instructions to save a group of registers on the stack.

4. Makes space for local variables on the stack.

subq \$48, %rsp

Makes disp amount of space on the top of the stack.

disp is usually negative because the stack grows towards decreasing # memory locations in x86.

During a return from a call : tasks performed by Callee

1. Restores registers. Not used here unless it is inevitable. The pre-assigned protocol of registers to caller and callee normally suffices without a need for restoring registers across a call.

2. Sets the base pointer to the activation record of the calling procedure.

Leave

This instruction is equivalent to

movl %rbp, %rsp

popq %rbp

3. Returns to the caller.

ret

This instruction restores the return address by resetting the old value of %rip saved on the stack at the time of call.

During a return from a call : tasks performed by Caller

1. Restores stack pointer to the location containing the returned value.

```
movq %rax, -20(%rbp) # return value is assigned to a name
```

Take Home Assignment :

For all the program fragments used in the examples, draw the AR for each of the functions with respect to the caller-callee conventions. Show the relative address of all the local variables and parameters on the stack (if applicable) and the passing of parameters from actual to formal using the registers dedicated for our architecture.

RUNTIME ENVIRONMENTS : A SUMMARY

BASIC CONCEPTS & ISSUES

To decide on the organization of data objects, so that their addresses can be resolved at compile time.

- The data objects (represented by variables) come into existence during the execution of the program.
- The generated code must refer to the data objects using their addresses, which must be resolved during compilation.
- The addresses of data objects depend on their organization in memory. This is largely decided by source language features.

We shall restrict this module to study of **Programming language environments** in which

- The sizes of objects and their relative positions in memory are known at compile time.
- Recursion is permitted.
- Data structures can be created at run time.

We call such a allocation as **stack allocation**. C, C++ and Pascal are among the languages that support such environments.

The organization of data is largely determined by answers to questions like the following.

1. **Does the language permit recursion?** There may be many incarnations of factorial active at the same time, each with its copy of local variables and parameters which must be allocated storage. Moreover, the number of such incarnations are only known at run-time.
2. **What are the parameter passing mechanisms?** Inside factorial, the access mechanisms for the formal parameters *r* and *x* have to be different. This is because *r* is **call-by-reference**, whereas *x* is **call-by-value**.
3. **How does a procedure refer to the non-local names?** The **rules of static scoping** decide that the *z* being referenced in factorial refers to the *z* in the main function. Therefore, apart from the local variables and formal parameters, factorial should be able to access this variable at run-time.

4. **Does the language permit creation of dynamic data structures?** Space must be created at run-time, each time new(y) is executed. What are the requirements of space allocation for locals, non-locals, parameters and dynamic data ?

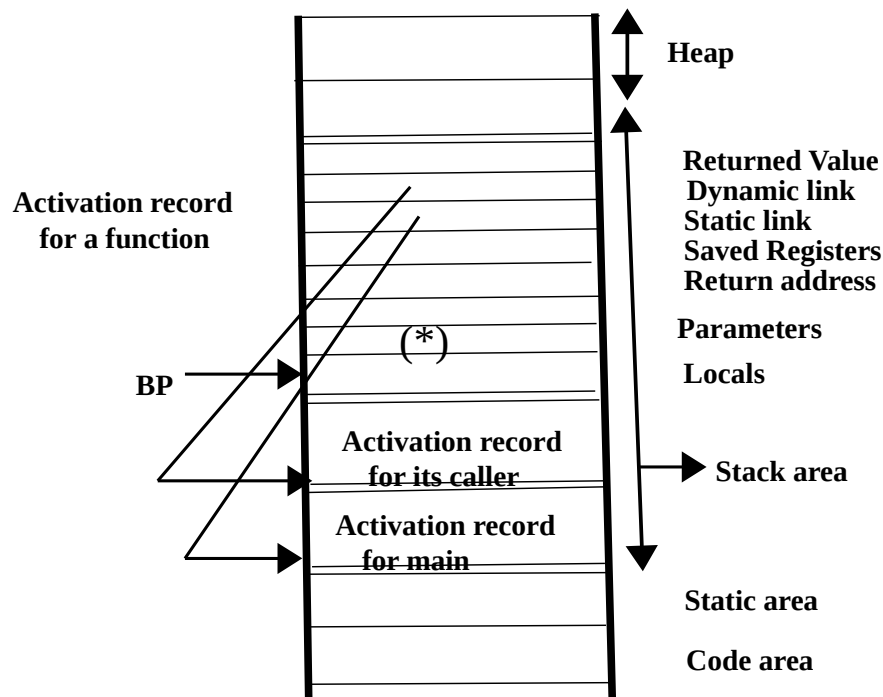
ACTIVATION RECORDS

An activation record contains the information required for a single activation of a procedure. Activation records are held in the static area for languages like Fortran and in the stack area for languages like C, C++, Pascal. Typically space must be provided in the activation record for the following.

1. **Local variables:** Part of the local environment of a procedure.
2. **Parameters:** Also part of the local environment.
3. **Return address:** To return to the appropriate control point of the calling procedure.
4. **Saved registers :** If the called procedure wants to use the registers used by the calling procedure, these have to be saved before and restored after the execution of the called procedure.
5. **Static link :** For accessing the non-local environment. The **static link** of a procedure points to the latest activation record of the immediately enclosing procedure. Static links, may be avoided by using a **display mechanism**. A static link is not required for Fortran IV like languages.
6. **Dynamic link :** Pointer to activation record of calling procedure.
7. **Returned value:** Used to store the result of a function call.

HANDLING DYNAMIC DATA STRUCTURES

A pointer variable p is allocated memory on the stack. On execution of , memory is allocated from the heap and the home location of the pointer variable (on the stack) is made to point to the allocated location in the heap. Any reference to p-> is an indirect reference to the heap location through the address on the stack



***** End of Document *****