CSO-1

X86 Assembly

Daniel G. Graham PhD

C LANGUAGE CALLING CONVENTION

The calling convention is broken into two sets of rules.

1. The first set of rules is employed by the caller of the subroutine (function)
2. The second set of rules is observed by the writer of the subroutine/function (the "callee")

CALLER VS CALLEE

```
void printMessage(bool to_be){
    if(to_be){
        puts("Hello World");
    }else{
        puts("not to be");
    }
}
```

printMessage is the caller – The function that called another function
puts is the callee – the function being called

OUR WORKING EXAMPLE

```
//callee
int add(int x, int y){
    int result = x + y;
    return result;
}

//caller
int main(){
    return add(2, 3);
}
```

```
//callee
int add(int x, int y){
    int result = x + y;
    return result;
}

//caller
int main(){
    return add(2, 3);
}
```

```
add(int, int):
    push %rbx
    push %rbp
    movq %rdi, %rbx
    movq %rsi, %rbp
    addq %rbx, %rbp
    movq %rbp, %rax
    pop %rbp
    pop %rbx
    ret
main:
    movq $3, %rsi
    movq $2, %rdi
    call add(int, int)
    ret
```

16 REGISTERS

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

REGISTERS (STACK POINTER)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

THE CALLER

```
//caller
int main(){
    return add(2, 3);
}
```

CALLER RULES

Rule 1. The caller should save the content of the register designated caller saved

REGISTERS (CALLER SAVED)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

CALLER RULES

Rule 1. The caller should save the content of the register designated caller saved

Rule 2. To pass parameters to the subroutine, we put up to six of them into registers (in order: rdi, rsi, rdx, rcx, r8, r9). If there are more than six parameters to the subroutine, then push the rest onto the stack in *reverse order* (i.e. last parameter first) – since the stack grows down.

REGISTERS (CALLER SAVED)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

THE CALLER

```
//caller
int main(){
    return add(2, 3);
}
```

```
main:
    movq $3, %rsi
    movq $2, %rdi
```

CALLER RULES

Rule 1. The caller should save the content of the register designated caller saved

Rule 2. To pass parameters to the subroutine, we put up to six of them into registers (in order: rdi, rsi, rdx, rcx, r8, r9). If there are more than six parameters to the subroutine, then push the rest onto the stack in *reverse order* (i.e. last parameter first) – since the stack grows down.

Rule 3. To call the subroutine, use the call instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code.

Run the call instruction

.

THE CALLER

```
//caller
int main(){
    return add(2, 3);
}
```

```
main:
    movq $3, %rsi
    movq $2, %rdi
    call add(int, int)
    ret
```

CALLER RULES

Rule 1. The caller should save the content of the register designated caller saved

Rule 2. To pass parameters to the subroutine, we put up to six of them into registers (in order: rdi, rsi, rdx, rcx, r8, r9). If there are more than six parameters to the subroutine, then push the rest onto the stack in *reverse order* (i.e. last parameter first) – since the stack grows down.

Rule 3. To call the subroutine, use the call instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code.

Run the call instruction .

Rule 4. After the subroutine returns, (i.e. immediately following the call instruction) the caller must remove any additional parameters (beyond the six stored in registers) from stack. This restores the stack to its state before the call was performed

CALLER RULES

Rule 1. The caller should save the content of the register designated caller saved

Rule 2. To pass parameters to the subroutine, we put up to six of them into registers (in order: rdi, rsi, rdx, rcx, r8, r9). If there are more than six parameters to the subroutine, then push the rest onto the stack in *reverse order* (i.e. last parameter first) – since the stack grows down.

Rule 3. To call the subroutine, use the call instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code.

Run the call instruction .

Rule 4. After the subroutine returns, (i.e. immediately following the call instruction) the caller must remove any additional parameters (beyond the six stored in registers) from stack. This restores the stack to its state before the call was performed

Rule 5. The caller can expect to find the subroutine's return value in the register RAX.

Rule 1. The caller should save the content of the register designated caller saved

Rule 2. To pass parameters to the subroutine, we put up to six of them into registers (in order: rdi, rsi, rdx, rcx, r8, r9). If there are more than six parameters to the subroutine, then push the rest onto the stack in *reverse order* (i.e. last parameter first) – since the stack grows down.

Rule 3. To call the subroutine, use the call instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code.

Run the call instruction .

Rule 4. After the subroutine returns, (i.e. immediately following the call instruction) the caller must remove any additional parameters (beyond the six stored in registers) from stack. This restores the stack to its state before the call was performed

Rule 5. The caller can expect to find the subroutine's return value in the register RAX.

Rule 6. The caller restores the contents of caller-saved registers (r10, r11, and any in the parameter passing registers) by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine.

REGISTERS (CALLER SAVED)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

CALLEE

```
//callee
int add(int x, int y){
    int result = x + y;
    return result;
}
```

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

EXAMPLE OF ALLOCATING LOCAL VARIABLES

```
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
--snip--
```

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

REGISTERS (CALLEE SAVED)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

SAVING REGISTERS

```
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
    push %rbx
    push %rbp
    --snip--
```

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

Run the call instruction .

SAVING REGISTERS

```
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
    push %rbx
    push %rbp
    movq %rdi, %rbx
    movq %rsi, %rbp
    addq %rbx, %rbp
    --snip--
```

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

Run the call instruction .

Rule 3. When the function is done, the return value for the function should be placed in RAX

SAVING REGISTERS

```
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
    push %rbx
    push %rbp
    movq %rdi, %rbx
    movq %rsi, %rbp
    addq %rbx, %rbp
    movq %rbp, %rax
--snip--
```

REGISTERS (CALLER SAVED)

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Arguments #5
%r9	Arguments #6
%r10	Caller saved
%r11	Caller saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

Run the call instruction .

Rule 3. When the function is done, the return value for the function should be placed in RAX

Rule 4. The function must restore the old values of any callee-saved registers (RBX, RBP, and R12 through R15) that were modified. The registers should be popped in the inverse order that they were pushed.

SAVING REGISTERS

```
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
    push %rbx
    push %rbp
    movq %rdi, %rbx
    movq %rsi, %rbp
    addq %rbx, %rbp
    movq %rbp, %rax
    pop %rbp
    pop %rbx
    --snip--
```

CALLEE RULES

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

Run the call instruction .

Rule 3. When the function is done, the return value for the function should be placed in RAX

Rule 4. The function must restore the old values of any callee-saved registers (RBX, RBP, and R12 through R15) that were modified. The registers should be popped in the inverse order that they were pushed.

Rule 5. Next, we deallocate local variables. By subtracting from RSP

Rule 1. Allocate local variables by using registers or making space on the stack.

Rule 2. Next, the values of any registers that are designated callee-saved that will be used by the function must be saved. To save registers, push them onto the stack. RSP will be pushed to the stack by the call instruction.

Run the call instruction .

Rule 3. When the function is done, the return value for the function should be placed in RAX

Rule 4. The function must restore the old values of any callee-saved registers (RBX, RBP, and R12 through R15) that were modified. The registers should be popped in the inverse order that they were pushed.

Rule 5. Next, we deallocate local variables. By subtracting from RSP

Rule 6. Execute the `ret` instruction.

SAVING REGISTERS

```
//callee
int add(int x, int y){
    int result= x + y;
    return result;
}
```

```
add(int, int):
    push %rbx
    push %rbp
    movq %rdi, %rbx
    movq %rsi, %rbp
    addq %rbx, %rbp
    movq %rbp, %rax
    pop %rbp
    pop %rbx
    ret
```

CALLEE'S PROLOGUE AND EPILOGUE:

Sometimes you will see the following callee prologue and epilogue added the beginning and end of the function

```
push rbp                    ; at the start of the callee (prologue)
mov rbp, rsp
...
pop rbp                     ; just before the ending ' ret '  (epilogue)
```

This code is unnecessary and is a hold-over from the 32-bit calling convention. You can tell the compiler to not include this code by invoking it with the `-fomit-frame-pointer` flag.

NEXT TIME

Swap Example with Mov instruction

Swap Example with lea (load effective address) instruction.

Later:

- jmp instruction and condition codes (Building loops)
- switch statements.