CSO-1

X86 Assembly

Daniel G. Graham PhD

University of Virginia | ENGINEERING

Contents

1. Factorial Recursive Example
2. Swap Example With Mov
3. Swap Example with Lea
4. Jmp Instructions

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")

OUR WORKING EXAMPLE

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

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

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.

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.

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.

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

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

CONDITION CODES

Single bit registers

- CF Carry Flag (for unsigned)
- SF Sign Flag (for signed)
- ZF Zero Flag
- OF Overflow Flag (for signed)

Implicitly set (think of it as a side effect) by arithmetic operations

A value of 1 indicates the condition is true and a value 0 indicates that the flag is not set

IMPLICITLY SETTING THE FLAG

Implicitly set (think of it as a side effect) by arithmetic operations

Example: **`addq`** Src,Dest ↔ **`t = a+b`**

CF set if the carry out from most significant bit (unsigned overflow)

ZF set if **`t == 0`**

SF set if **`t < 0`** (as signed)

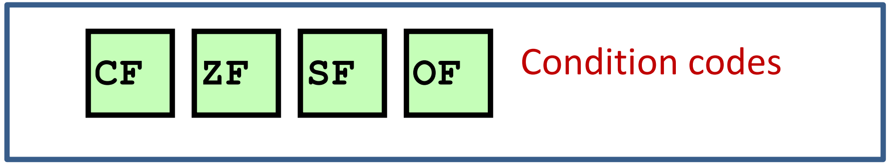
OF set if two's-complement (signed) overflow

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

COMPARE (CMP) INSTRUCTION

`cmp` does subtraction (but doesn't store result)

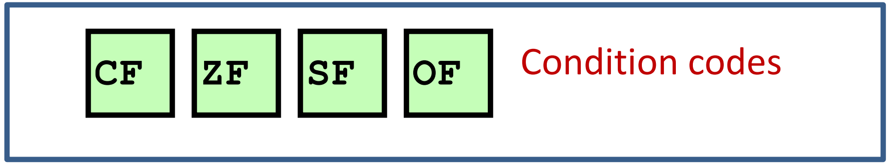
```
cmp %rax, %rdi -> rdi - rax
```



TEST INSTRUCTION

`test` does bitwize add

```
test %rax, %rdi -> rdi & rax
```



Sets zero flag if the result of bitwise is zero
Also sets the SF flag with the most significant bit of the result is one

CONDITION CODES AND JUMPS

- `jg`, `jle`, etc. read condition codes
- named based on interpreting result of subtraction
- 0: equal;
- negative: less than;
- positive: greater than

CF	ZF	SF	OF	Condition codes

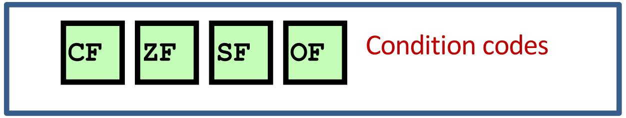
Normally a comparison is done before the jmp

JUMP INSTRUCTIONS AND CONDITION CODES

Instruction	Description	Condition Code
jle	Jump if less or equal	(SF XOR OF) OR ZF
jg	Jump if greater (signed)	NOT (SF XOR OF) & NOT ZF
je	Jump if equal	ZF

Talk to your neighbor.

Why check the overflow flag?



EXAMPLE 1

```
movq $-10, %rax
movq $20, %rbx
subq %rax, %rbx // %rbx - %rax = 30
   // result > 0: %rbx was > %rax
jle foo // not taken; 30 > 0
```

jle	Jump if less or equal	(SF XOR OF) OR ZF

CF ZF SF OF Condition codes

EXAMPLE 2 (TAKEN OR NOT TAKEN)

Talk to your neighbor.

```
movq $10, %rax
movq $-20, %rbx
subq %rax, %rbx
jle foo
```

jle	**Jump if less or equal**	**(SF XOR OF) OR ZF**

EXAMPLE 2 (TAKEN OR NOT TAKEN)

```
movq $10, %rax
movq $-20, %rbx
subq %rax, %rbx
jle foo
```

jle	**Jump if less or equal**	**(SF XOR OF) OR ZF**

CF ZF SF OF Condition codes

-20-10 = -30
Sign flag set

condition codes example (3)

```
movq $-10, %rax
movq $20, %rbx
subq %rax, %rbx
jle  foo // not taken, %rbx - %rax > 0 -> %rbx
```

Jump is taken if result in rbx is <= 0

Instruction	Description	Condition Code
jle	Jump if less or equal	(SF XOR OF) OR ZF

condition codes example (3)

```
movq $20, %rbx
addq $–20, %rbx
je   foo // taken, result is 0
         // x - y = 0 -> x = y
```

Instruction	Description	Condition Code
je	Jump if equal	ZF

What instructions set condition codes

most instructions that compute something set condition codes

some instructions only set condition codes:

- `cmp` ~ `sub`
- `test` ~ `and` (bitwise and)
  - Example: `testq %rax, %rax` — result is %rax

some instructions don't change condition codes:

- `lea`, `mov`
- control flow: `jmp`, `call`, `ret`, etc.

COMPUTED JUMPS

Computed jumps

Instruction	Description
jmpq %rax	goto address RAX
jmpq 1000(%rax,%rbx,8)	read address from memory at RAX + RBX * 8 + 1000 // go to that address

OVERLAPPING REGISTERS

setting 32-bit registers — clears corresponding 64-bit register

```
movq $0xFFFFFFFFFFFFFFFF, %rax
movl $0x1, %eax
```

%rax is `0x1` (not `0xFFFFFFFF00000001`)

setting 8/16-bit registers: doesn't clear 64-bit register

```
movq $0xFFFFFFFFFFFFFFFF, %rax
movb $0x1, %al
```

%rax is `0xFFFFFFFFFFFFFF01 not 0x01`

LIVE CODING SESSION

GNU nano 6.3 factorial.c Modified

```c
/**
This program demostrates how to calculate factorial
factorial(5) = 5*4*3*2*1
We can do this is a recursive way
**/


int factorial(int x){
        //Base case if x  = 1
        if (x == 1){
                return x;
        }else{
                return x*factorial(x-1);
        }
}

int main(){
        int result = factorial(3);
        return 0;
}
```

^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location M-U Undo
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo

[0] 0:nano* "portal09" 09:45 13-Oct-23

GNU nano 6.3 factorial.c Modified

```c
int factorial(int x){
	//Base case if x  = 1
	 if (x == 1){
		return x;
	 }else{
		return x*factorial(x-1);
	 }
}

int main(){
	 int result = factorial(3);
	 return 0;
}
```

^G Help ^O Write Out ^W Where Is ^K Cut
^X Exit ^R Read File ^\ Replace ^U Paste

GNU nano 6.3 factorial.s Modified

```asm
.globl main
.globl factorial

factorial:
	cmp $1, %rdi
	jne .recusive_call
	movq %rdi, %rax
	ret
.recusive_call:
	pushq %rdi
	subq $1, %rdi
	call factorial
	popq %rdi
	#multiplies %rdi * %rax and store result in %rax
	imulq %rdi, %rax
	ret
main:
	movq $3, %rdi
	call factorial
	xorq %rax, %rax
	ret
```

^G Help ^O Write Out ^W Where Is ^K Cut
^X Exit ^R Read File ^\ Replace ^U Paste

[0] 0:nano* "portal09" 09:43 13-Oct-23

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.