CSC 6580
Spring 2020

Instructor: Stacy Prowell

Homework: Binary Math

Checking argument number

```
main:
      push rbp
      mov rbp, rsp
      ; Expect to get two arguments.
      cmp rdi, 3
      je .okay
      mov rdi, badargs
      call puts
      mov rax, 1
      jmp .done
.okay:
      ;...
badargs:
      db "Expected exactly two integer arguments.",0
```

Parsing the arguments

```
.okay:  sub rsp, 16             ; Reserve space for two 8-byte values.
        push rsi                ; Save RSI which holds argv.
        mov rdi, [rsi+8]        ; Move argv[1] into RDI.
        call atol               ; Parse the string pointed to by RDI.
        mov [rbp-8], rax        ; Save parsed integer in reserved space.
        pop rdi                 ; Restore RSI (holds argv).
        mov rdi, [rdi+16]       ; Move argv[2] into RDI.
        call atol               ; Parse the string pointed to by RDI.
        mov [rbp-16], rax       ; Save parsed integer in reserved space.
```

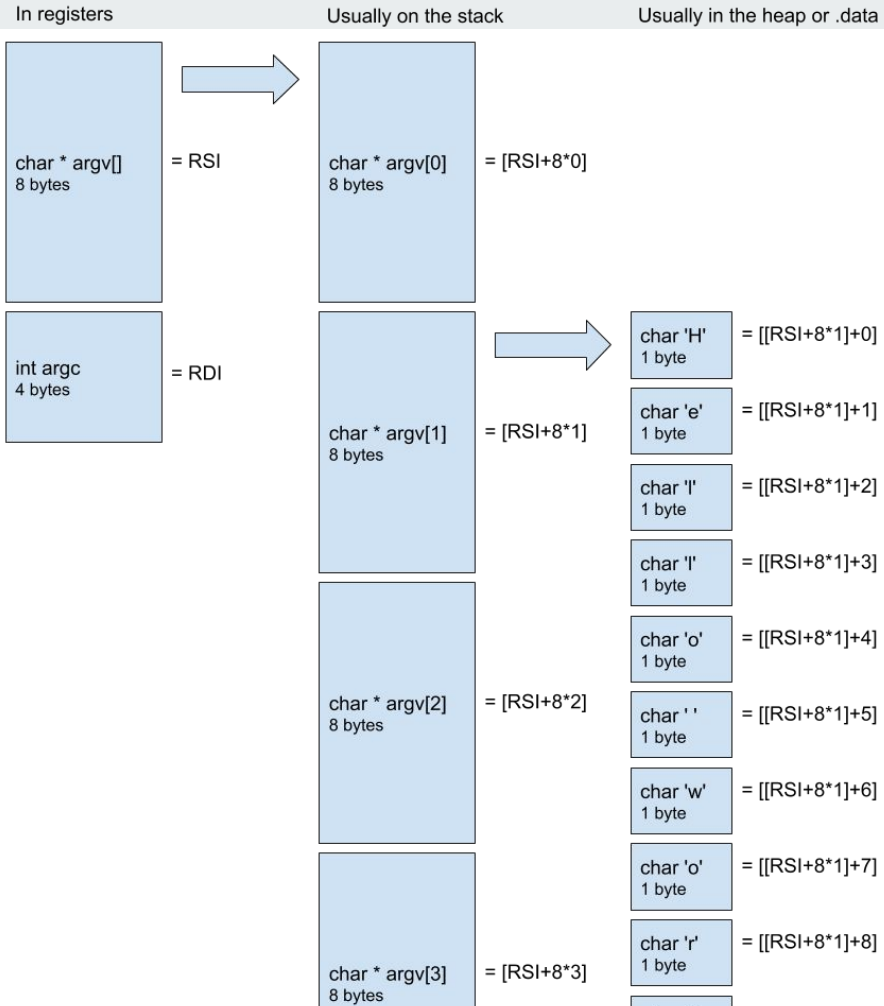
Aside:
Pointers and argc / argv

Thinking about argc / argv

```
; RSI holds the address of argv[0],
; but we want argv[1].
mov r11, [rsi+8]

; Now R11 holds the content of argv[1].
; This is a pointer to the first char.
mov ah, [r11+6]

; Now AH holds the content of R11+6.
; That is, AH holds the char 'w'.
```



Assembly vs C

```
; RSI holds the address of argv[0],
; but we want argv[1].
mov r11, [rsi+8]

; Now R11 holds the content of argv[1].
; This is a pointer to the first char.
mov ah, [r11+6]

; Now AH holds the content of R11+6.
; That is, AH holds the char 'w'.
```

The value in `argv` is the address of a `char *`. That is, it's a pointer to a `char *`, or a `char **`.

Since `sizeof(char *) == 8`, writing `argv[1]` is the same as writing `*(char **)((char *)argv + 8)`.

You can think of `[...]` as *dereferencing* a pointer, usually. The brackets indicate that the *content* of the memory is the argument, not the address itself.

Except with `lea`. Then you just get the address, not the contents!

Aside:
The stack and stack frame

Little Endian / Big Endian

You have to remember that *everything* (pretty much) is just *convention*. In particular, how to order the bytes of a multi-byte thing differs.

For instance, our *usual* encoding of the decimal value 678 won't fit in a single byte.

`678` = `0b 10 1010 0110`*

We typically split this into two bytes:

The high byte: `0000 0010`
The low byte: `1010 0110`

Aside:
Even this hasn't always been universal! Some machines (in the 80's even) stored `678` as:
`0b 0000 0110 0111 1000`

*I'll try to prefix binary numbers with `0b` and break them up by nybble.

Little Endian / Big Endian

Given the two bytes, how do we store them?

The high byte: `0b 0000 0010`
The low byte: `0b 1010 0110`

We can store them in consecutive memory addresses, say *A* and *A*+1. But which byte goes where?

The X86 ISA uses little endian. RISC is big endian. ARM is configurable.

Little Endian
Low order byte goes at low address (the "low end" is first)
A: `0b 1010 0110` (low)
A+1: `0b 0000 0010` (high)

Big Endian
High order byte goes at low address (the "big end" is first)
A: `0b 0000 0010` (high)
A+1: `0b 1010 0110` (low)

Little Endian / Big Endian

When we send a multi-byte number over a network connection, we *typically* send it in **network byte order**, sending the highest order byte first.

In network byte order the "big end" comes first; thus it is considered big endian.

Send `0b 0000 0010 1010 0110` over the network:

1. Send `0b 0000 0010`
2. Send `0b 1010 0110`

The stack pointer RSP

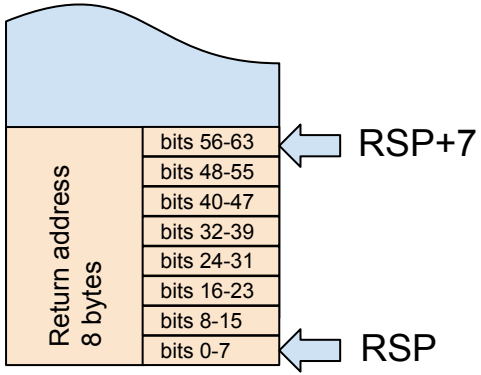
Always use RSP when referencing the stack! ESP is not the stack pointer.

The stack pointer RSP always points to the lowest address byte of the item on the top of the stack. This is confusing if we envision memory addresses increasing as we go up (like the y axis), but…

If you *just pushed* EAX (4 bytes), then you will find the lowest-order bits (0-7) in the byte pointed to by RSP. You will find the highest-order bits (24-31) in the byte pointed to by RSP+3.

Stack on entry

On entry the top of the stack is the return value.
The `ret` instruction will, essentially, `pop rip`.



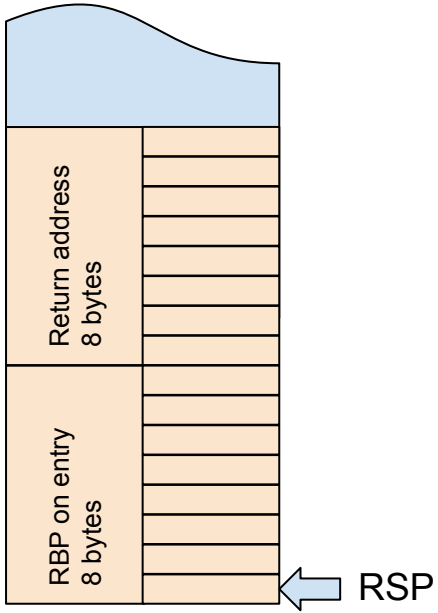
Save caller's RBP

The calling convention says that the base pointer `RBP` should be preserved across calls, so we have to save the caller's `RBP` with: `push rbp`

The stack pointer now points to the low-order byte of the stored value of `RBP`.

This is equivalent to:

```
sub rsp, 8
mov [rsp], rbp
```

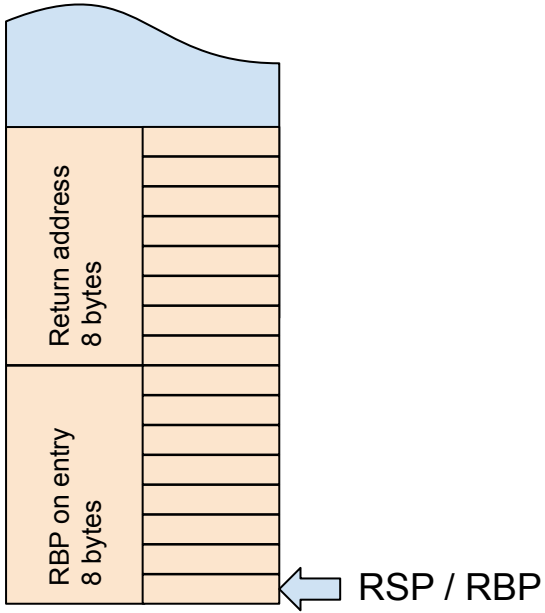


Set our own base

Next we establish our own base pointer with:

```
mov rbp, rsp
```

Now the two point to the same location. The idea is that we will never change `RBP`, though `RSP` may be changed as we modify the stack. `RBP` points to *our stack frame*.



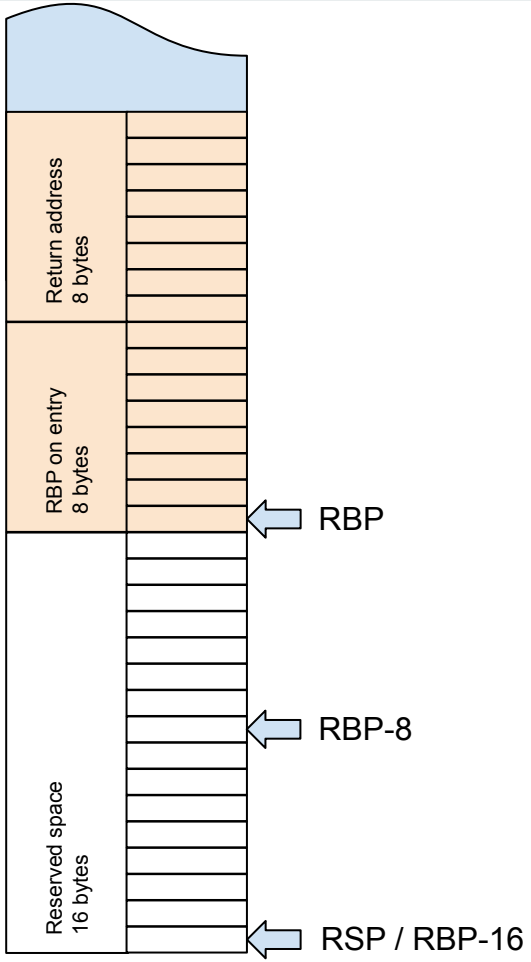
Save space for locals

It is really convenient to store local variables on the stack. This is how local variables that you create in C functions are allocated (well, usually). That's why they go away when you return (usually).

We reserve space for two 8-byte values with:

```
sub rsp, 16
```

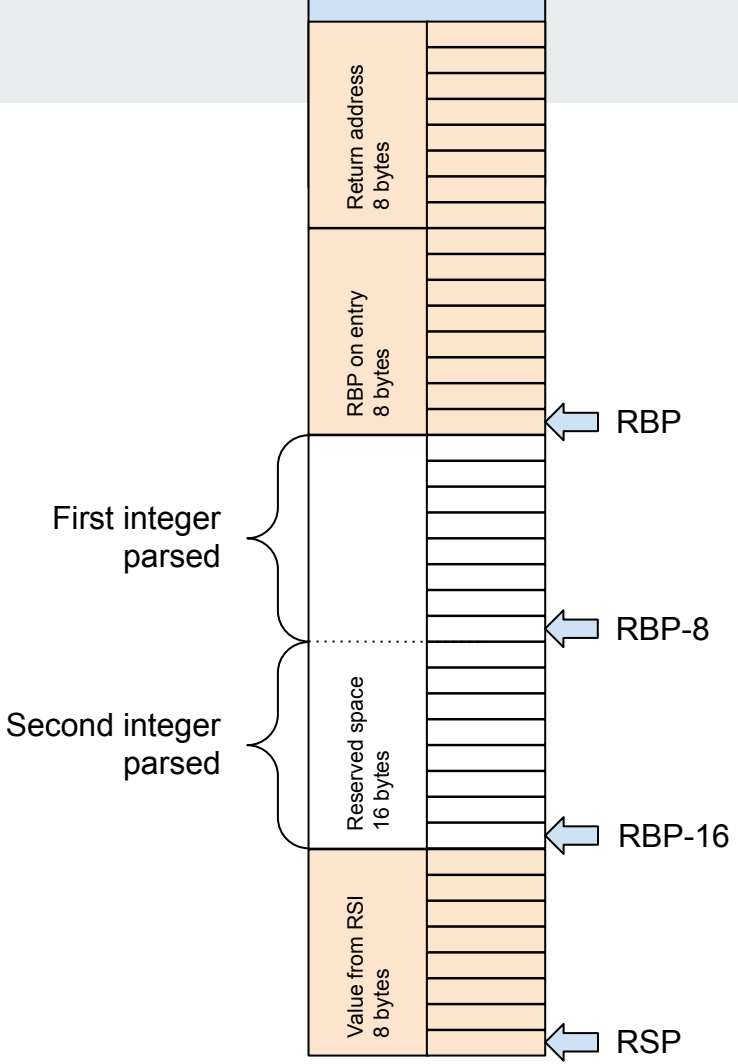
These locations are *not initialized*; they may contain anything!



All set up

The first integer will get stored in `[rbp-8]`, and the second one will get stored in `[rbp-16]`.

We can still use the stack, of course. We push `RSI` temporarily to save it for a few instructions later. This changes `RSP`... but not `RBP`!

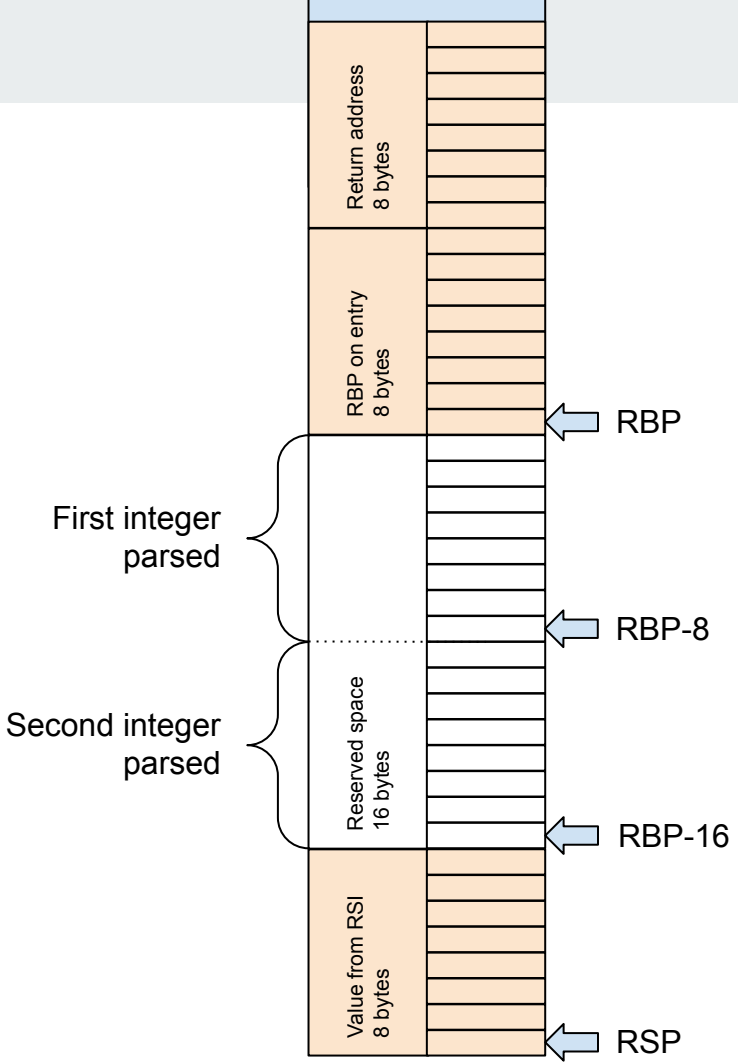


Ending the function

At the end of the function we have to clean up what we did. The simplest way is to do these two instructions:

```
mov rsp, rbp
pop rbp
```

These two instructions are equivalent to the `leave` instruction.

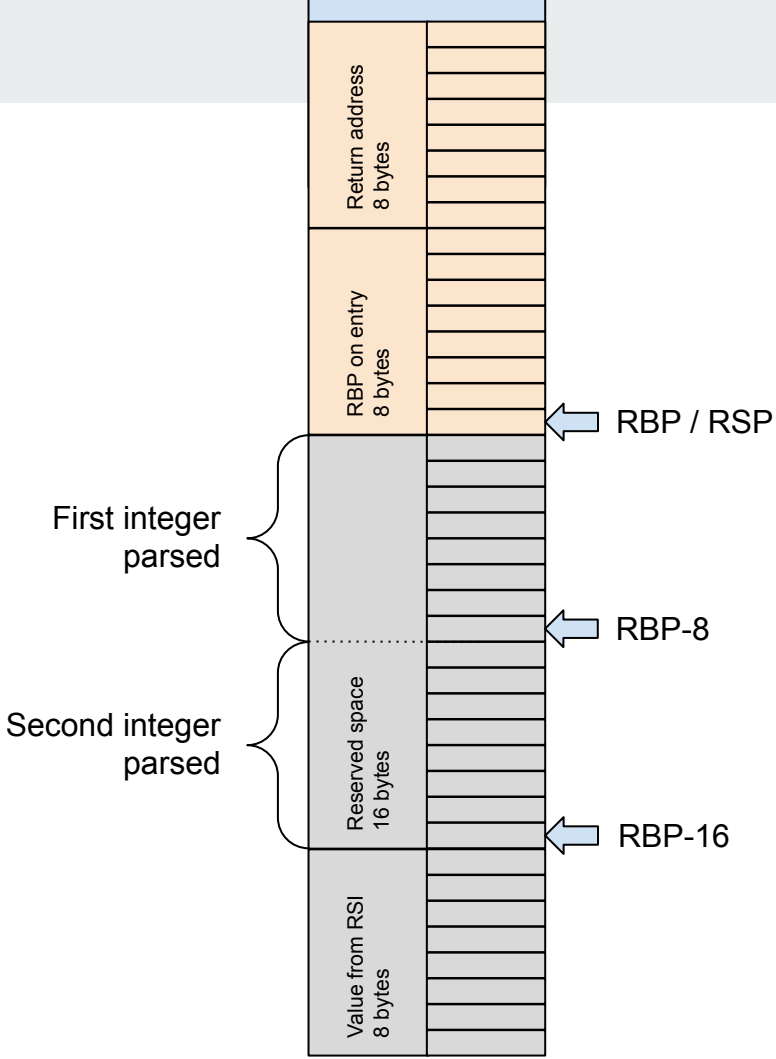


Reset the stack pointer

The `mov rsp, rbp` resets the stack pointer.

Everything below the stack pointer remains; this is called the *dirty stack*. It might contain old local variables, scratch space, old pointers, important cryptographic keys, your password, or a lot of other stuff!

Let's pretend it isn't there, just like most programmers do.



Restore the caller's base

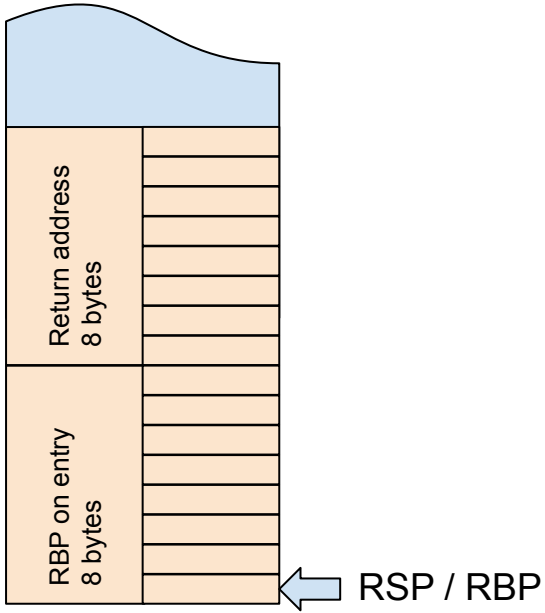
Now we can restore the caller's base pointer by popping it from the stack:

```
pop ebp
```

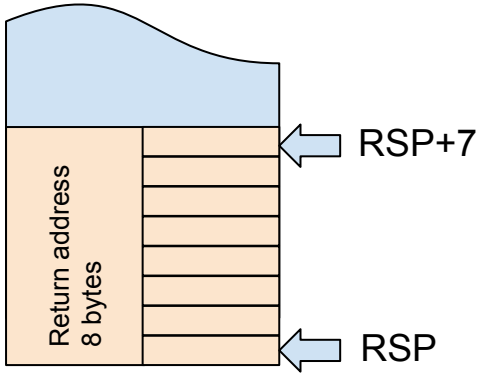
This is equivalent to:

```
mov rbp, [rsp]
add rsp, 8
```

(This is also done by the `leave` instruction.)



Ready to return



The ret instruction pops the value off the stack and places it in `RIP`.

This is (almost) equivalent to:

```
add rsp, 8
jmp [rsp-8]
```

Back to the homework

Adding

```
; Add
mov rdi, add
call puts
mov rdi, [rbp-8]        ; Get first integer from reserved space.
call write_binary_qword ; Write in binary.
call write_endl
mov rdi, [rbp-16]       ; Get second integer from reserved space.
call write_binary_qword ; Write in binary.
call write_endl
mov rdi, [rbp-8]        ; Load first integer into RDI.
add rdi, [rbp-16]       ; Add second integer into RDI.
call write_binary_qword ; Write in binary.
call write_endl
```

Subtracting

```
; Subtract
mov rdi, sub
call puts
mov rdi, [rbp-8]        ; Get first integer from reserved space.
call write_binary_qword ; Write in binary.
call write_endl
mov rdi, [rbp-16]       ; Get second integer from reserved space.
call write_binary_qword ; Write in binary.
call write_endl
mov rdi, [rbp-8]        ; Load first integer into RDI.
sub rdi, [rbp-16]       ; Subtract second integer from RDI.
call write_binary_qword ; Write in binary.
call write_endl
```

System calls

syscall

The provided code used the `syscall` instruction. This is the instruction that triggers a system call. If you did 32-bit stuff, this is roughly the equivalent of `int 80`.

It's all part of the ABI convention.

syscall

- User-level applications use as integer registers for passing the sequence `RDI`, `RSI`, `RDX`, `RCX`, `R8` and `R9`. **The kernel interface uses** `RDI`, `RSI`, `RDX`, `R10`, `R8` and `R9`**.**
- A system call is done via the **`syscall` instruction**. This clobbers `RCX` and `R11` as well as the `RAX` return value, but other registers are preserved.
- The number of the system call has to be passed in register `RAX`.
- System calls are limited to **six arguments**, and **no argument is passed directly on the stack**.
- Returning from the system call, register `RAX` contains the result of the system call. A value in the range between -4095 and -1 indicates an error (it is `-errno`).
- Only integer and pointer values are passed to the kernel.

Source: AMD X86-64 ABI calling conventions

syscall

This is stuff you need to know if you are implementing an operating system--or trying to analyze one. Otherwise you probably don't need to know it.

Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

source: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2

syscall

Read about Linux system calls (all 328) here:
https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

syscall

Some calls we've already used:

```
mov rax, 60          ; sys_exit
mov rdi, 17          ; the return value
syscall              ; do it!  does not return
hlt                  ; by convention
```

Use this to terminate a program if you are not using the C runtime.

syscall

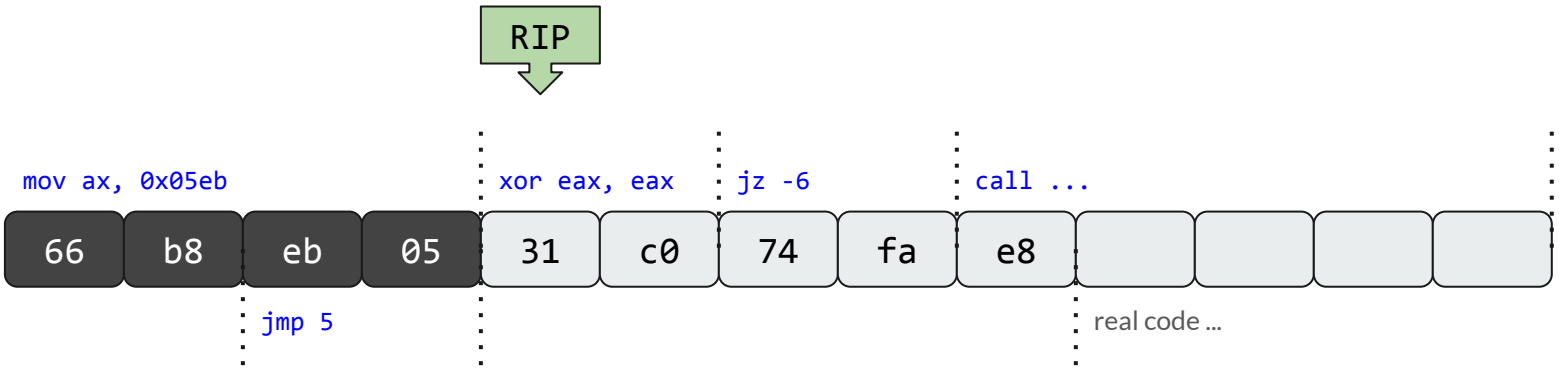
Some calls we've already used:

```
mov rax, 1          ; sys_write
mov rdi, 1          ; file descriptor 1 (stdout)
mov rsi, msg        ; the message to write (char *)
mov rdx, msglen     ; the length of the message
syscall             ; do it!
                    ; continue...
```

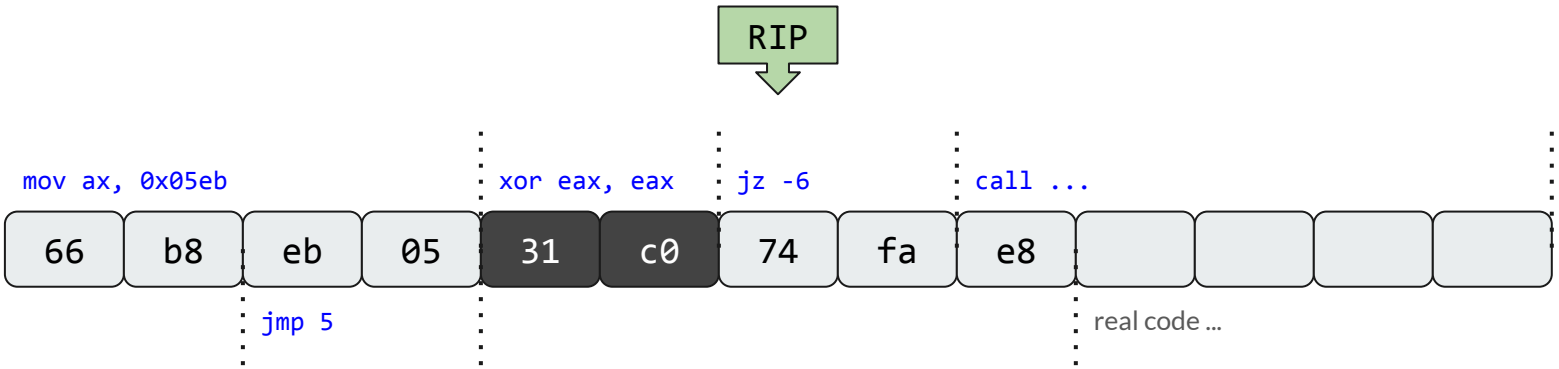
Use this to write a message without needing `puts` or other stdlib functions.

Anti-Disassembly

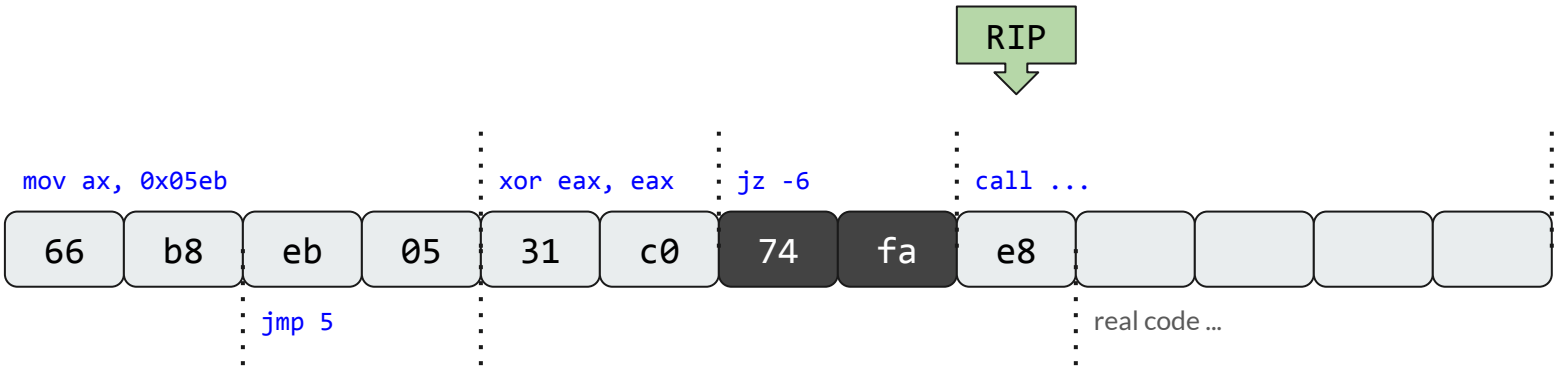
Recall this code



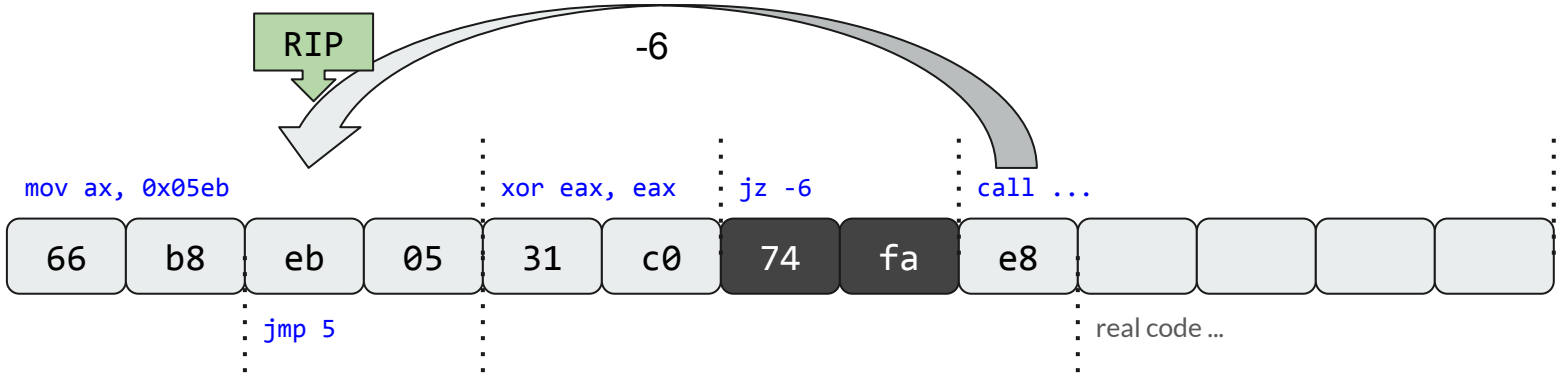
Recall this code



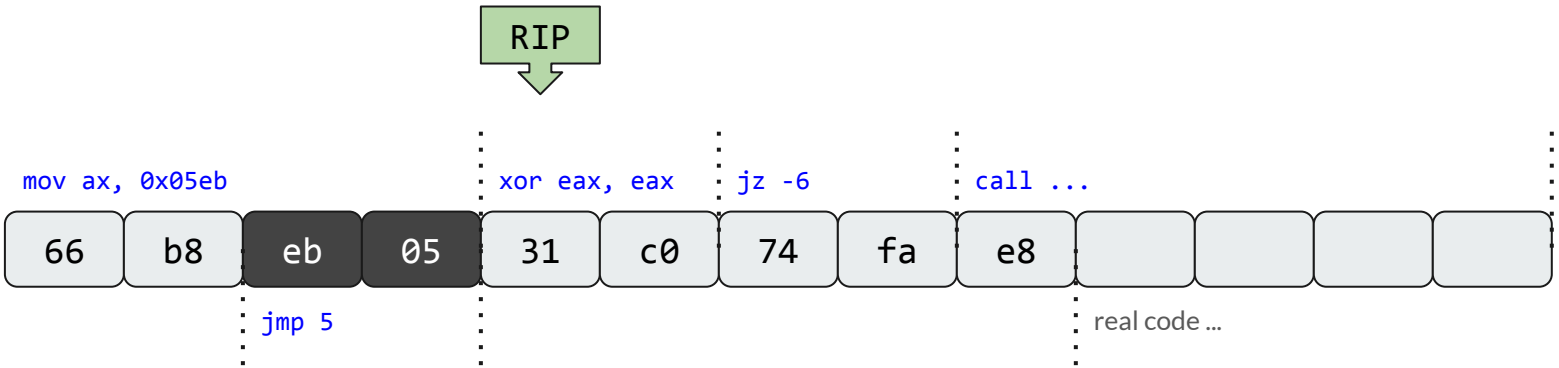
Recall this code



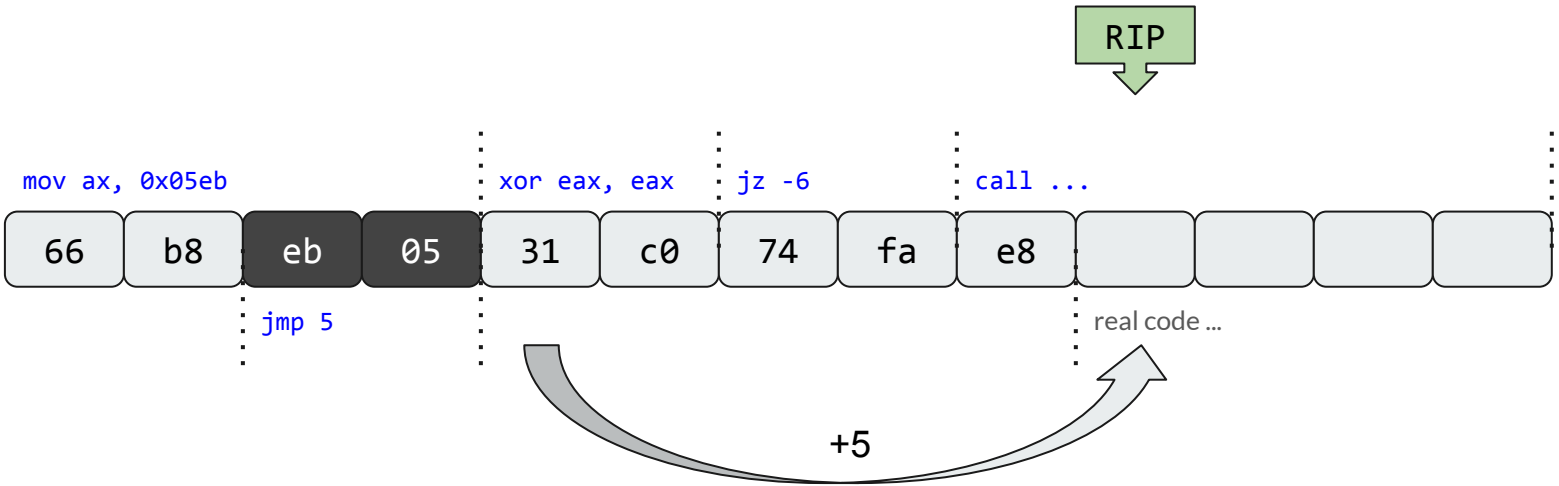
Recall this code



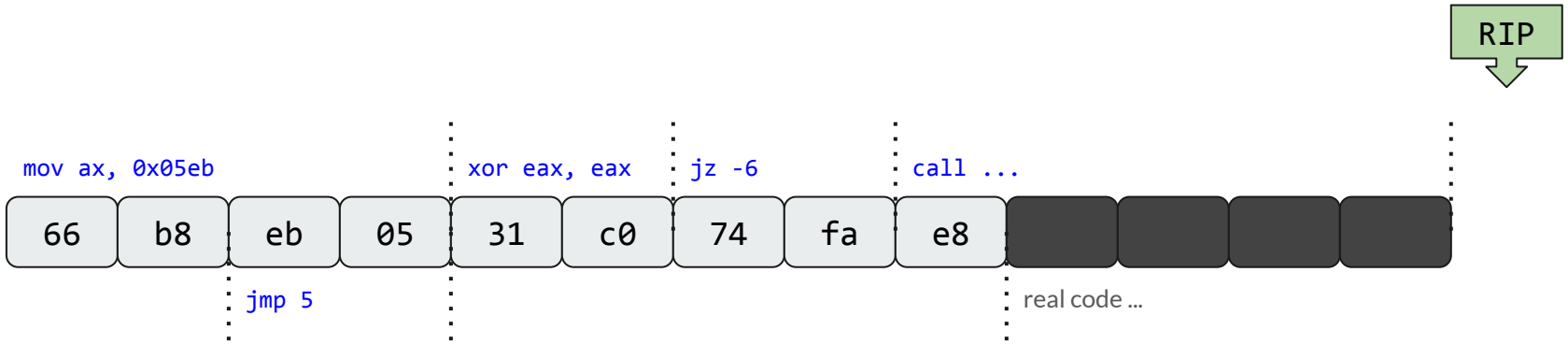
Recall this code



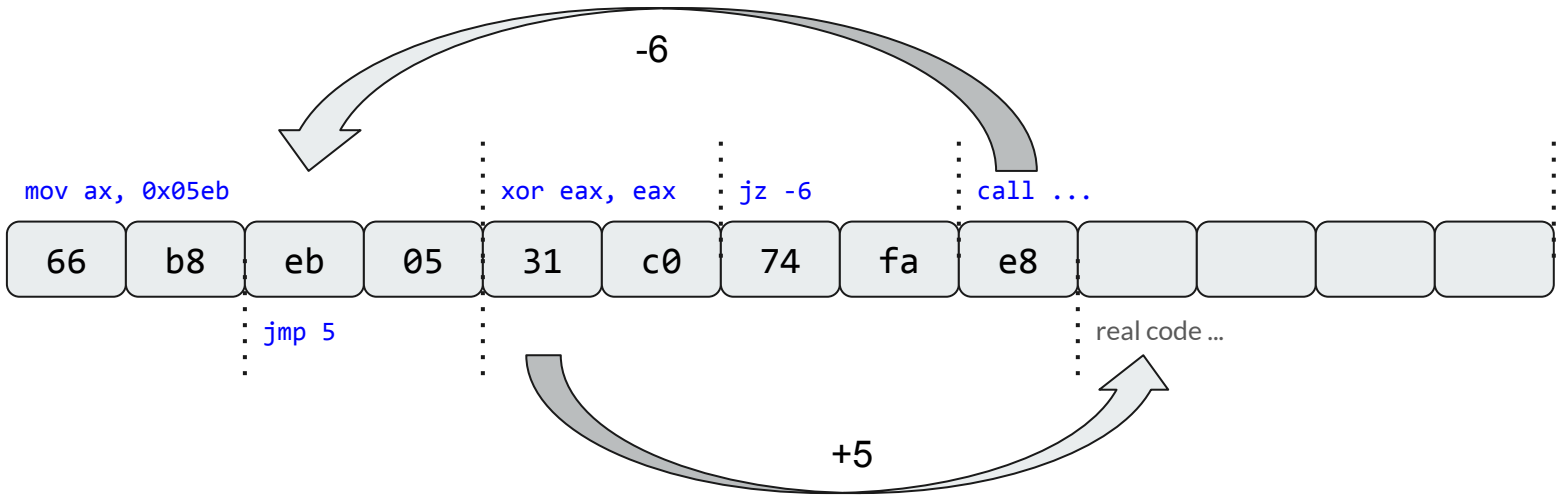
Recall this code



Recall this code



Recall this code



Control Flow

Structure

```
264             s->s_shrink.flags = SHRINKER_NUMA_AWARE | SHRINKER_MEMCG_AWARE;
265             if (prealloc_shrinker(&s->s_shrink))
266                     goto fail;
267             if (list_lru_init_memcg(&s->s_dentry_lru, &s->s_shrink))
268                     goto fail;
269             if (list_lru_init_memcg(&s->s_inode_lru, &s->s_shrink))
270                     goto fail;
271             return s;
272
273     fail:
274             destroy_unused_super(s);
275             return NULL;
```

All examples like this are from: https://elixir.bootlin.com/linux/latest/source/fs/super.c

Structure

```
500   retry:
501           spin_lock(&sb_lock);
502           if (test) {
503                   hlist_for_each_entry(old, &type->fs_supers, s_instances) {
504                           if (!test(old, data))
505                                   continue;
506                           if (user_ns != old->s_user_ns) {
507                                   spin_unlock(&sb_lock);
508                                   destroy_unused_super(s);
509                                   return ERR_PTR(-EBUSY);
510                           }
511                           if (!grab_super(old))
512                                   goto retry;
513                           destroy_unused_super(s);
514                           return old;
515                   }
516           }
517           if (!s) {
518                   spin_unlock(&sb_lock);
519                   s = alloc_super(type, (flags & ~SB_SUBMOUNT), user_ns);
520                   if (!s)
521                           return ERR_PTR(-ENOMEM);
522                   goto retry;
523           }
```

Structure

```
    do a;
    if (error) goto out_a;
    do b;
    if (error) goto out_b;
    do c;
    if (error) goto out_c;
    goto out;
out_c:
    undo c;
out_b:
    unto b;
out_a:
    undo a;
out:
    return ret;
```

```
do a;
if (!error) {
    do b;
    if (!error) {
        do c;
        if (!error) {
            return ret;
        }
        undo c;
    }
    undo b;
}
undo a;
```

The linux kernel uses goto. Is this a good / bad thing?

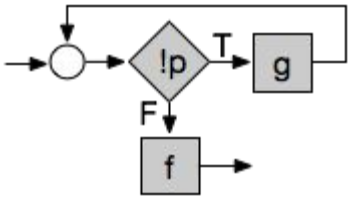
The kernel is written in C, and C lacks exception handling; local gotos can simplify the code, or reduce indentation.

However: goto complicates binary analysis… and in assembly it's mostly gotos.

Flowgraph

A flowgraph is a graph whose vertices are the program nodes (the single operations, or subprograms), and whose edges indicate the program run ordering.

```
while(!p) {
    g();
}
f();
```



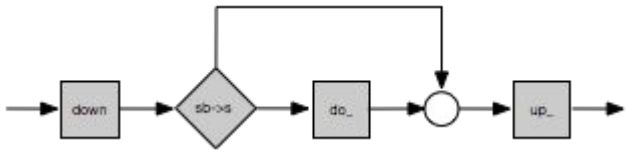
Proper Program

A *proper program* is a program with:

- A single entry point
- A single exit point (might not mean just what you think)
- For every node of the program, there is a path from entry to exit that contains that node (no unreachable or trap nodes)

Represents a single *function*.

```
static void do_emergency_remount_callback(struct super_block *sb)
{
        down_write(&sb->s_umount);
        if (sb->s_root && sb->s_bdev && (sb->s_flags & SB_BORN) &&
            !sb_rdonly(sb)) {
                /*
                 * What lock protects sb->s_flags??
                 */
                do_remount_sb(sb, SB_RDONLY, NULL, 1);
        }
        up_write(&sb->s_umount);
}
```



Proper Program

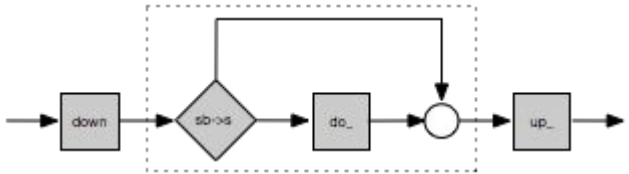
A *proper program* is a program with:

- A single entry point
- A single exit point (might not mean just what you think)
- For every node of the program, there is a path from entry to exit that contains that node (no unreachable or trap nodes)

Represents a single *function*.

Proper programs can contain other proper *sub*programs.

```
static void do_emergency_remount_callback(struct super_block *sb)
{
        down_write(&sb->s_umount);
        if (sb->s_root && sb->s_bdev && (sb->s_flags & SB_BORN) &&
            !sb_rdonly(sb)) {
                /*
                 * What lock protects sb->s_flags??
                 */
                do_remount_sb(sb, SB_RDONLY, NULL, 1);
        }
        up_write(&sb->s_umount);
}
```



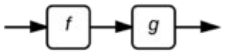
Program Primes

Single entry, single exit

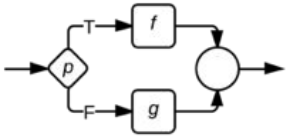
Path from entry to exit for each internal node

It does not contain any proper subprogram that is a prime; that is, you can't reduce it to a combination of other primes.

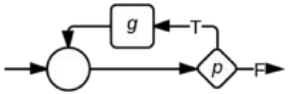
$g \circ f$



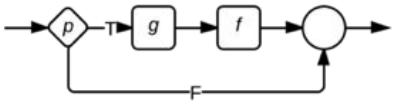
$$\begin{bmatrix} p & \mapsto f \\ \neg p & \mapsto g \end{bmatrix}$$



f



$$\begin{bmatrix} p & \mapsto f \circ g \\ \neg p & \mapsto I \end{bmatrix}$$

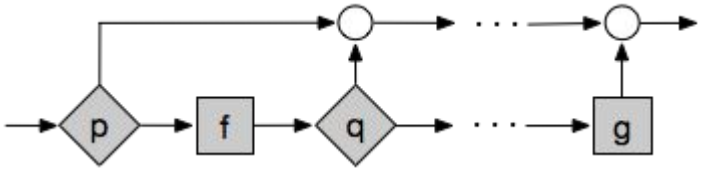


Not a prime!

Program Primes

There are lots of common primes:

- If-then-else
- While-do
- Do-until
- Do-while-do
- Case
- ...



There are, in fact, infinitely many primes.

Representation

Can *any* program with goto be transformed into a program *without* goto?

For that matter, can any program with a set of structures be transformed into a program with just a few structures?

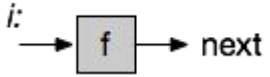
Functions and Predicates

Assign every operation in the program an integer, and assign the exit of the program the number zero.

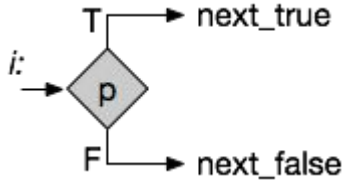
Divide up the operations into classes.

Is this the full story? What about jmp? What about jmp [eax]? What about the loop (dec ecx ; jnz ___) instruction?

- A *function* performs some operation and *then* flows to a single next operation.
  - add eax, ebx



- A *predicate* performs some comparison and *if* true flows to a single next operation, *else* it flows to some (possibly) other single next operation.
  - jz skip

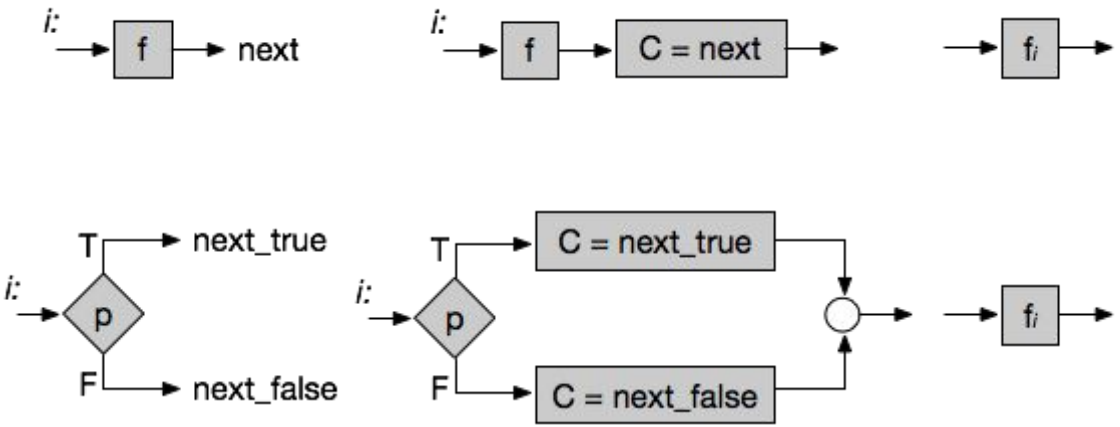


Create if-then-else and sequence

Add a counter variable, making the program counter explicit.

Convert each function into a sequence that sets the counter.

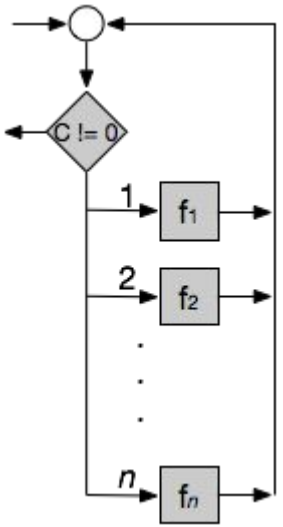
Convert each predicate into an if-then-else that sets the counter.



Simulate Original Control Flow

We can simulate the original control flow by:

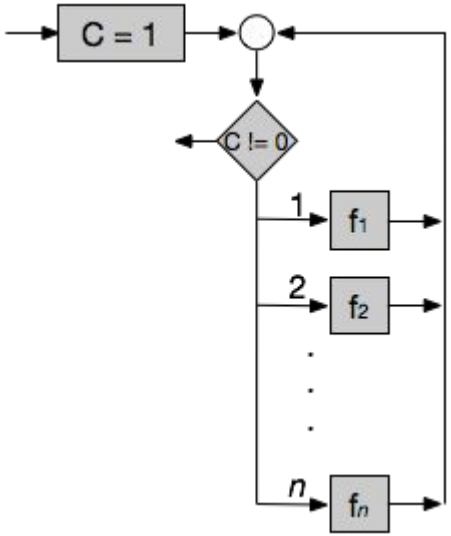
- Initialize the counter to the entry point.
- While the counter is not zero:
  - Enter a nested if (or case) that selects the correct sequence or if-then-else subprogram.



Simulate Original Control Flow

Why are we doing this?

Easier to analyze a smaller set of program structures?

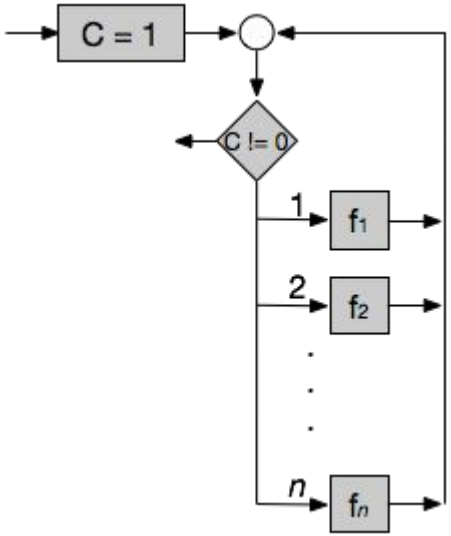


Simulate Original Control Flow

Why are we doing this?

Easier to analyze a smaller set of program structures?

Maybe to obfuscate?

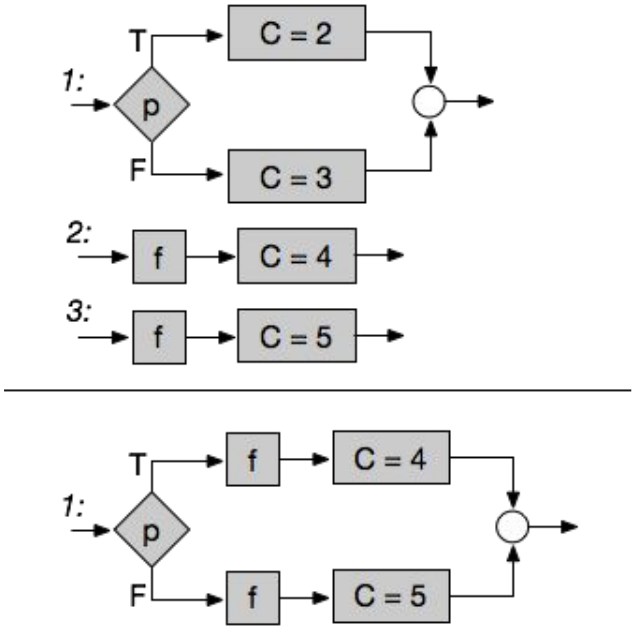


Counter

Can the counter be eliminated? If the original program is a structured program, then yes.

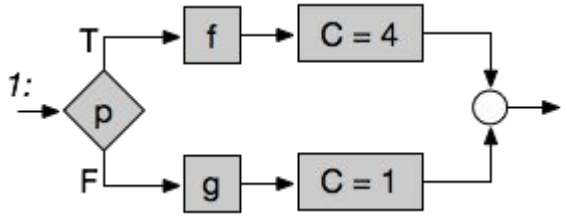
Substitute structures for the counter assignments.

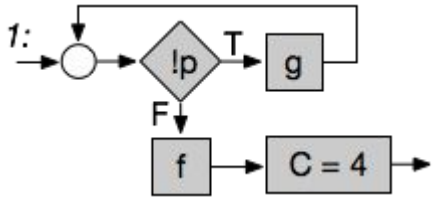
If you can eliminate all references to the counter variable, you have eliminated the variable.



Loop Discovery

If a structure references itself, it is an (implicit) loop. You can make it into an explicit loop.





Structure Theorem

Any **proper program** is **execution equivalent** to a **structured program** with **basis set** {sequence, while-do, if-then-else} with tests and assignments to a single additional counter variable of size ceil(lg(*n*)), where *n* is the number of nodes in the original program.

(The first correct proof is due to Harlan Mills.)

Proper Program (Again)

A *proper program* is a program with:

- A single entry point
- A single exit point (might not mean just what you think)
- For every node of the program, there is a path from entry to exit that contains that node (no unreachable or trap nodes)

This is a proper program. Is it a structured program?

```
static struct super_block *__get_super_thawed(struct block_device *bdev,
                                              bool excl)
{
        while (1) {
                struct super_block *s = __get_super(bdev, excl);
                if (!s || s->s_writers.frozen == SB_UNFROZEN)
                        return s;
                if (!excl)
                        up_read(&s->s_umount);
                else
                        up_write(&s->s_umount);
                wait_event(s->s_writers.wait_unfrozen,
                           s->s_writers.frozen == SB_UNFROZEN);
                put_super(s);
        }
}
```

Proper Program

Is this a structured program?

Is it a proper program?

```
struct super_block *get_active_super(struct block_device *bdev)
{
        struct super_block *sb;

        if (!bdev)
                return NULL;

restart:
        spin_lock(&sb_lock);
        list_for_each_entry(sb, &super_blocks, s_list) {
                if (hlist_unhashed(&sb->s_instances))
                        continue;
                if (sb->s_bdev == bdev) {
                        if (!grab_super(sb))
                                goto restart;
                        up_write(&sb->s_umount);
                        return sb;
                }
        }
        spin_unlock(&sb_lock);
        return NULL;
}
```

Proper Program

What isn't a proper program?

```
.text:004014f0
.text:004014f0 55                push rbp
.text:004014f1 4889e5            mov rbp,rsp
.text:004014f4 53                push rbx
.text:004014f5 4883ec08          sub rsp,0x8
.text:004014f9 803d805d200000    cmp BYTE PTR [rip+0x205d80],0x0 # 0x00607280
.text:00401500 754b              jne loc_0040154d
.text:00401502 bb386e6000        mov ebx,0x606e38
.text:00401507 488b057a5d2000    mov rax,QWORD PTR [rip+0x205d7a] # 0x00607288
.text:0040150e 4881eb306e6000    sub rbx,0x606e30
.text:00401515 48c1fb03          sar rbx,0x3
.text:00401519 4883eb01          sub rbx,0x1
.text:0040151d 4839d8            cmp rax,rbx
.text:00401520 7324              jae loc_00401546
.text:00401522 660f1f440000      nop WORD PTR [rax+rax*1+0x0]
.text:00401528
```

From Linux "whoami"

Proper Program

What isn't a proper program?

```
.text:004014f0
.text:004014f0 55                push rbp
.text:004014f1 4889e5            mov rbp,rsp
.text:004014f4 53                push rbx
.text:004014f5 4883ec08          sub rsp,0x8
.text:004014f9 803d805d200000    cmp BYTE PTR [rip+0x205d80],0x0 # 0x00607280
.text:00401500 754b              jne loc_0040154d
.text:00401502 bb386e6000        mov ebx,0x606e38
.text:00401507 488b057a5d2000    mov rax,QWORD PTR [rip+0x205d7a] # 0x00607288
.text:0040150e 4881eb306e6000    sub rbx,0x606e30
.text:00401515 48c1fb03          sar rbx,0x3
.text:00401519 4883eb01          sub rbx,0x1
.text:0040151d 4839d8            cmp rax,rbx
.text:00401520 7324              jae loc_00401546
.text:00401522 660f1f440000      nop WORD PTR [rax+rax*1+0x0]
.text:00401528
```

From Linux "whoami"

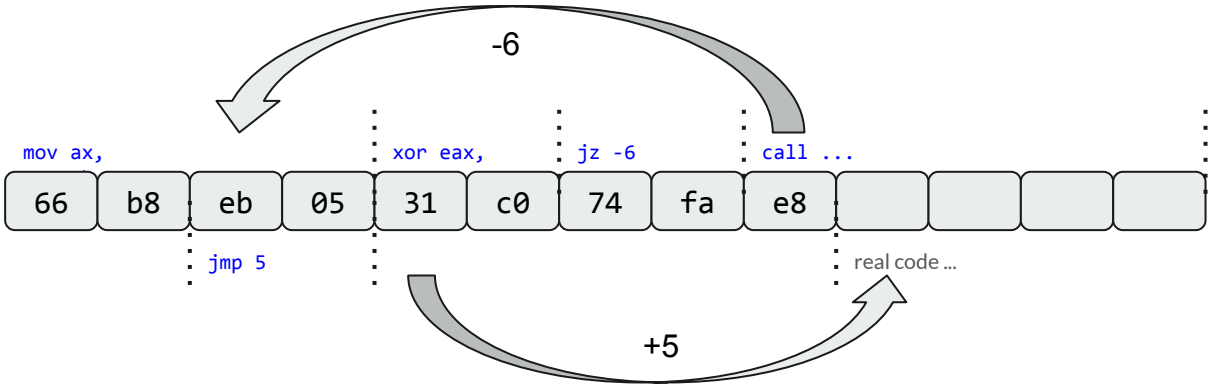
Proper Program

The point of a proper program is that it provides all the necessary *context* to be understood as a single function.

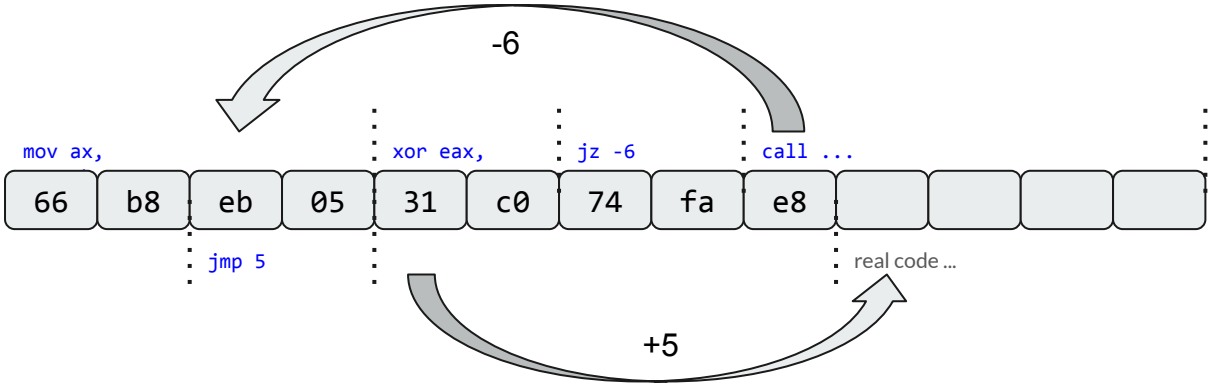
A proper program contains all its referents: it is *referentially transparent*.

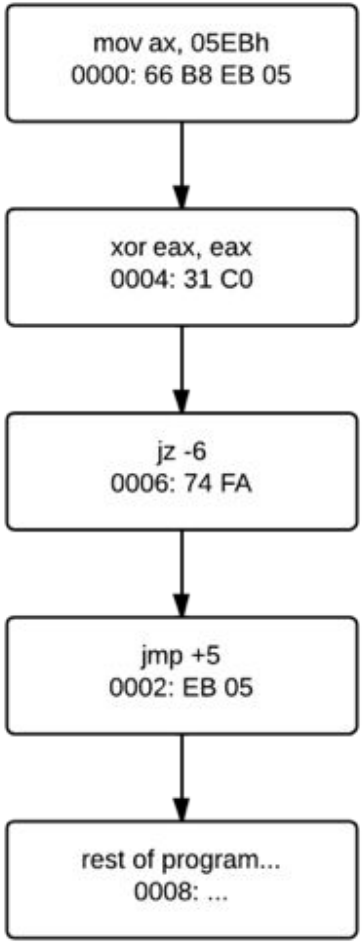
If you want to reason about a program - especially if you want to write programs that analyze other programs - then this is important!

What about this?



It's a proper program





Homework:
Due: Tuesday, February 18

Hexadecimal Math

Modify your prior solution to the binary math problem, or start with the provided solution, so that the output is in hexadecimal instead of binary. It is *suggested* to use *lower case* letters in your hexadecimal (they are easier to distinguish from digits). Place your code in a file called `addsub.asm` and make sure to correct the first line to correctly compile your code!

```
$ ./addsub 9000000000000000000 8999999999999999999
Adding:
7ce66c50e2840000
7ce66c50e283ffff
f9ccd8a1c507ffff
Subtracting:
7ce66c50e2840000
7ce66c50e283ffff
0000000000000001
```

Next time:
Structure discovery