CSC 373 Spring 2019 Prof. Lytinen
Introduction to Assembly Language

Readings:

Bryant and O'Hallaron, sections 3.1-3.7; 3.10.

Notes do not cover 3.6.8

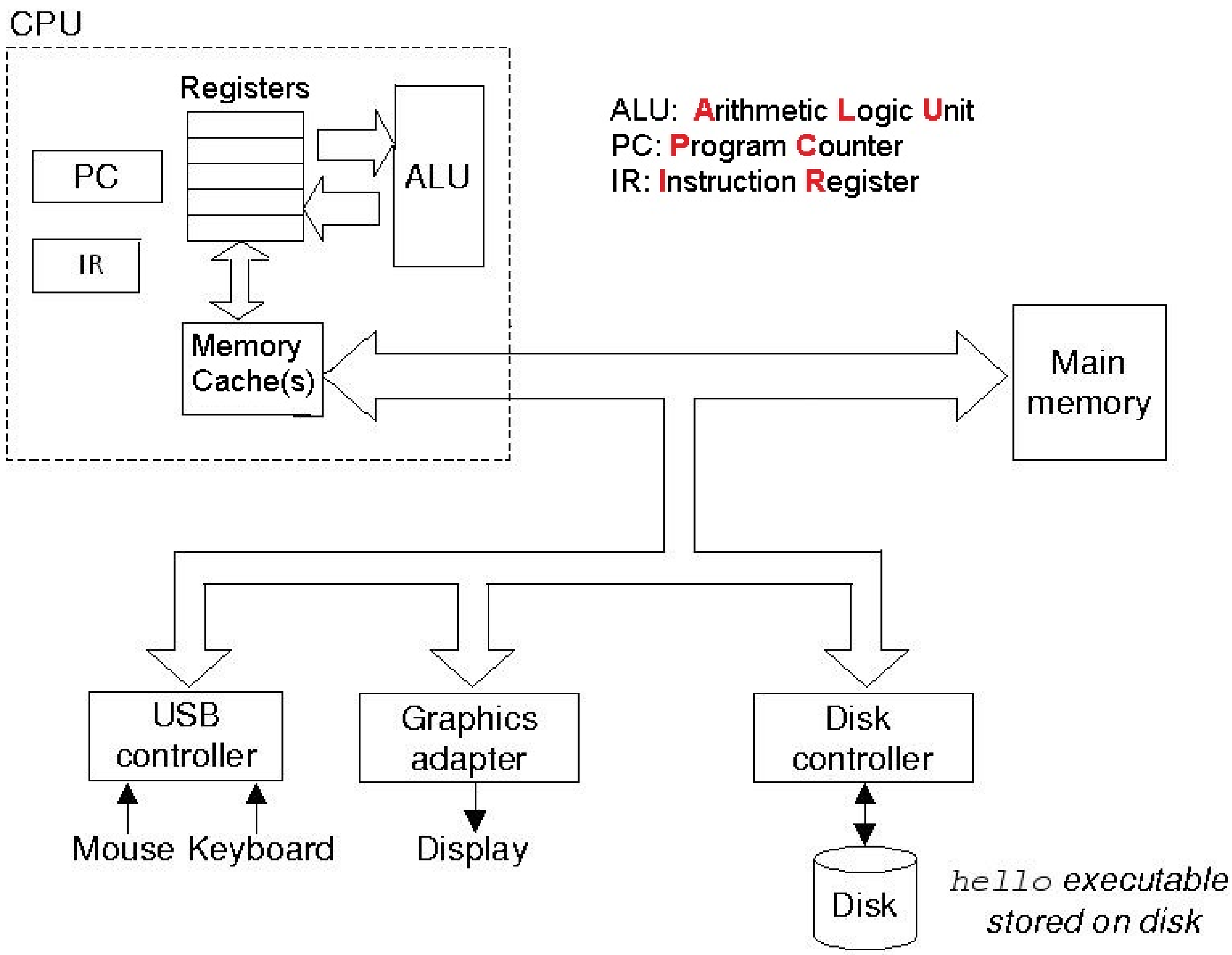
Machine Code

- When a C program is compiled, eventually it is translated into **machine code**
- Currently x86-64 machine code, named for historical reasons
- Original developed by **Advanced Micro Devices** (AMD) and was named AMD64
- x86-64 machine code has evolved from 16-bit processors to current 64-bit; backwards compatability has been maintained

Assembly Language

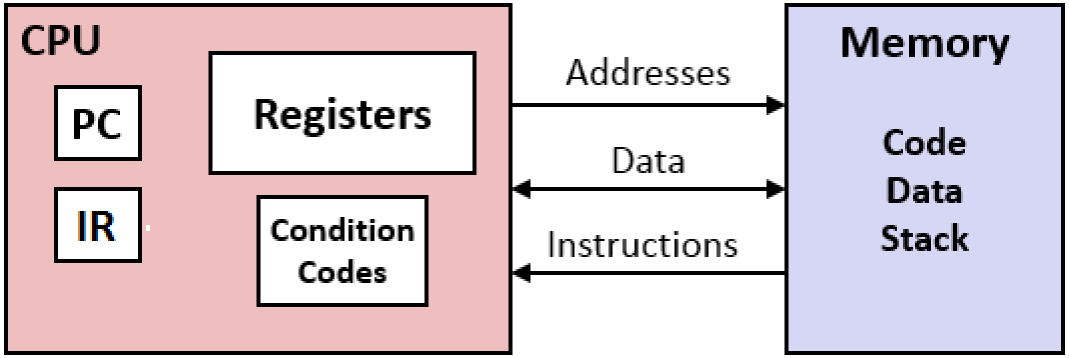
- In essence, a (somewhat) more readable version of machine code
- Translation to machine code is almost one-to-one (i.e., each assembly language instruction often translates to one machine code instruction)
- Instructions' names are used; no need to memorize op code (see below)
- Likewise for some operands (although hexadecimal is used to indicate addresses)
- x64 Assembly

x86-64 Architecture



ALU: Arithmetic Logic Unit
PC: Program Counter
IR: Instruction Register

What your program sees



Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Registers: 16 integer, 16 float point**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation

- **Memory**

From your program's perspective, every 64-bit computer has:

2^64 bytes ("virtual" memory) - no distinction between **disk/RAM/caches** (at least for the most part)

Memory from the point of view of a process

- Only 3 types of memory: **registers, main memory** and **files**
- Cache(s) are invisible to a process
- Processes also don't know what portions of their address space is currently in main memory and what is still on the disk.

Registers

- Program counter register (called %rip in assembly language) keeps track of execution location in main memory (a 64-bit address) . No direct access to this register.
- Instruction register contains the current instruction (1-15 bytes on x86-64)
- 16 regular 64-bit integer registers
- 16 64-bit floating point registers

Comparisons of speed

Memory Type	Latency (how many times slower)
Registers	1
L1 Cache	4
L2 Cache	10
L3 Cache	50
RAM	200
Disk	100,000

Chip history

- The first commercial 8-bit processor was the Intel 8008 (1972) (Wikipedia)
- chars, ints, floats(?), addresses, etc. all took up 1 byte
- 2^8 bytes of addressable memory!! 0x00 – 0xff. Everything else had to be on a disc (or another computer)

Intel 8008: 8-bit machine

- Eight 8-bit registers: A, B, C, D, SI, DI, SP, BP
- First 4 were general purpose; the other 4 were for specific purposes
- Naming made some sense at the time (SI = Source Index, DI = Destination Index, SP = Stack pointer, BP = Base pointer

Intel 8086: 16 bit machine

Intel 8086 registers

1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	0 9	0 8	0 7	0 6	0 5	0 4	0 3	0 2	0 1	0 0	*(bit position)*

Main registers H = higher, L = lower

	AH	AL	**AX** (primary accumulator)
	BH	BL	**BX** (base, accumulator)
	CH	CL	**CX** (counter, accumulator)
	DH	DL	**DX** (accumulator, extended acc.)

Index registers

16 bits each

0 0 0 0	SI	**S**ource **I**ndex
0 0 0 0	DI	**D**estination **I**ndex
0 0 0 0	BP	**B**ase **P**ointer
0 0 0 0	SP	**S**tack **P**ointer

Program counter

0 0 0 0	IP	**I**nstruction **P**ointer

32-bit machines

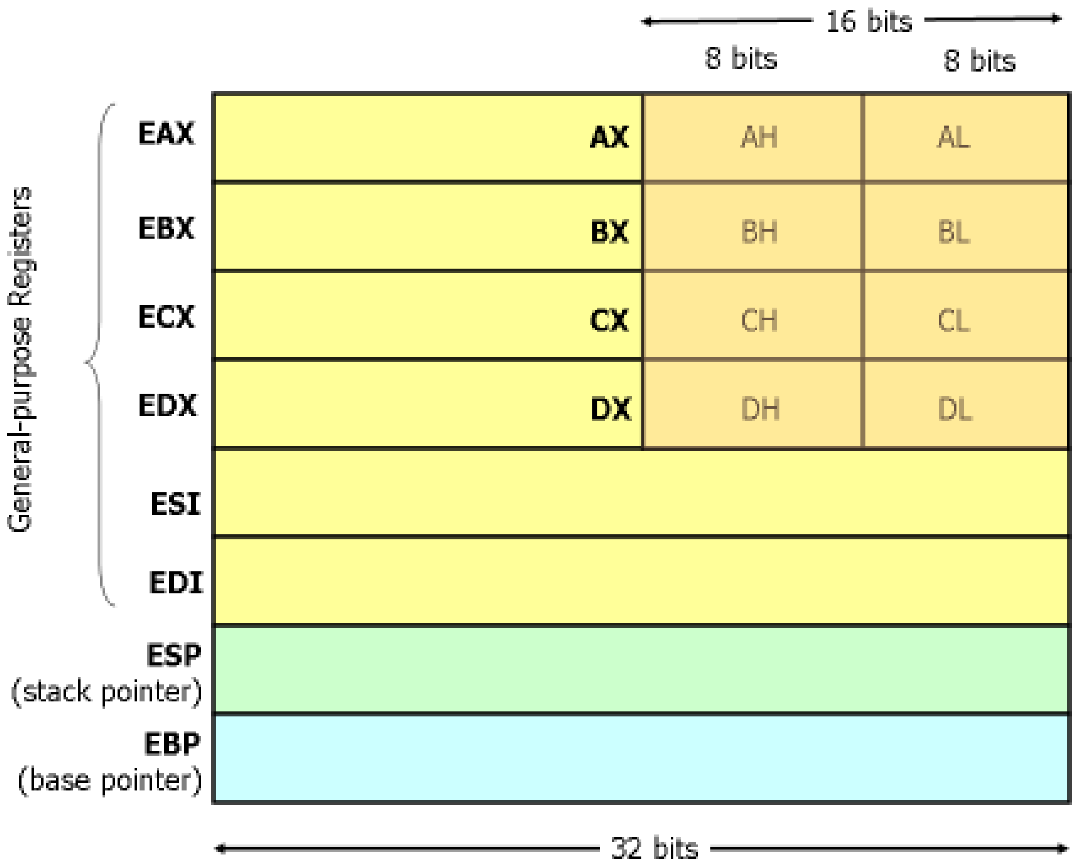


Figure 1. x86 Registers

X86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Example: %rdi, %edi, %di, %dil

x86-64 instructions

- Machine code instructions are composed of an **operation code** (op code) and its **operands**.   Of course, all are binary numbers.

- The first portion of an instruction (up to 3 bytes) encodes the op code

- Instructions are very low-level; e.g., 0x4889 is the op code which copies data from one 64-bit register to another

  - Example:

    `48 89 d8` means `movq %rbx,%rax`

  - Aside: why is the entire instruction 3 bytes long?

- Each opcode has a fixed number of operands (1 or 2), which takes up a fixed number of bytes (or possibly bits)

- Each operand specifies either a register or a (64-bit virtual) memory address
  Registers can be encoded in 4 bits, since there are 16 of them.

Register names

- General-purpose registers are legacy names for backwards compatibility

- First microprocessors were 8-bit (meaning the address space of a process was 2^8 bytes)

- General purpose registers were %a, %b, %c, %d

- Other registers had special purposes, and their names reflect this for historical reasons. For example, only certain registers could have pointer values (%di, %si, %bp, %sp) which we'll talk about later

- si: source index; di = destination index; bp = base pointer; sp = stack pointer

Register names

- When 16-bit architecture was introduced, integer registers were increased to 2 bytes (as well as address spaces). To distinguish, full 16-byte registers were name %ax, %bx, %cx, and %dx. However, for backwards compatibility, the lower-order byte of the 4 general-purpose registers could still be used with the name %a, %b, %c, and %d.

- 32-bit architecture: %eax, %ebx, %ecx, %edx, %edi, %esi, %ebp, %esp

- 6 of the 8 registers became general-purpose, but retained their special-purpose names again for backwards compatibility (%ebp and %esp were still special-purpose)

- To reference 8 bits, the register names changed to %al, %ah, etc.

- 64-bit architecture: there are now 16 integer registers, called %rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rbp, %rsp, %r8…%r15

X86-64 Integer Registers: Usage Conventions

While most registers can be used for general purposes, some are still also used for special purposes

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

Example of assembly language and machine code

```
int main() {
	printf("Hi\n");
	printf("Bye\n");
}
```

Memory address	Machine code of instruction in that address	Corresponding assembly language
400530	55	push %rbp
400531	48 89 e5	mov %rsp,%rbp
400534	bf e0 0f 40 00	mov $0x4005e0, %edi
400539	e9 d2 fe ff ff	callq 400410 <puts@plt>
40053e	bf 3e 0f 40 00	mov $0x4005e3, %edi
400543	e8 c8 fe ff ff	callq 400410 <puts@plt>
400548	5d	pop %rbp
400549	c3	retq

We can inspect further to see what the function does

- %edi is used to pass one of the parameters to a function (in this case `puts`)
- At 400534, the address 0x4005e0 is placed in %edi
- At 40053e, the address 0x4005e3 is placed in %edi
- In the debugger, we can type

  (gdb) x/s 0x4005e0
  0x4005e0: "Hi"
  (gdb) x/s 0x4005e3
  0x4005e3: "Bye“

- The op code `bf` is “move immediate” (also indicated by $

Memory address	Machine code	Corresponding assembly language
400530	55	push %rbp
400531	48 89 e5	mov %rsp,%rbp
400534	bf e0 05 40 00	mov **$0x4005e0**, %edi
400539	e9 d2 fe ff ff	callq 400410 <puts@plt>
40053e	bf e3 05 40 00	mov **$0x4005e3**, %edi
400543	e8 c8 fe ff ff	callq 400410 <puts@pllt>
400548	5d	pop %rbp
400549	c3	retq

How a program runs

./hello

1. A **process** is created

2. An **address space** is created; always 2^{64} bytes on a 64-bit machine
   The program now thinks it has 2^{64} bytes of memory
   - In reality, part of this address space may be on disk, in main memory, and/or in cache memory. Much of it will not be anywhere.

3. The starting address of the program in the address space is loaded into the CPU's **program counter** (%rip)
   - In the example above, 0x0000000000400530

How a program runs

4. Based on the contents of the PC, The first (next) instruction is copied from memory into the **instruction register**
   - First instruction: 0x55

5. The program counter is incremented by the appropriate number (1-15, depends on the length of the instruction)
   - After 1st instruction, increment %rip by 1
   - After 2nd instruction, increment %rip by 3

6. The instruction in the instruction register is executed. Some instructions modify the PC (to make loops, call other functions, etc.)

Repeat steps 4-6

PC

400530

IR

55

Memory address	Machine code	Corresponding assembly language
400530	**55**	**push %rbp**
400531	48 89 e5	mov %rsp,%rbp
400534	bf e0 05 40 00	mov **$0x4005e0**, %edi
400539	e9 d2 fe ff ff	callq 400410 <puts@plt>
40053e	bf e3 05 40 00	mov **$0x4005e3**, %edi
400543	e8 c8 fe ff ff	callq 400410 <puts@pllt>
400548	5d	pop %rbp
400549	c3	retq

PC

400531

IR

48 89 e5

Memory address	**Machine code**	**Corresponding assembly language**
400530	55	push %rbp
400531	**48 89 e5**	**mov %rsp,%rbp**
400534	bf e0 05 40 00	mov **$0x4005e0**, %edi
400539	e9 d2 fe ff ff	callq 400410 <puts@plt>
40053e	bf e3 05 40 00	mov **$0x4005e3**, %edi
400543	e8 c8 fe ff ff	callq 400410 <puts@pllt>
400548	5d	pop %rbp
400549	c3	retq

PC

400534

IR

bf e0 05 40 00

“little-endian”

Memory address	Machine code	Corresponding assembly language
400530	55	push %rbp
400531	48 89 e5	mov %rsp,%rbp
400534	**bf e0 05 40 00**	**mov $0x4005e0, %edi**
400539	e9 d2 fe ff ff	callq 400410 <puts@plt>
40053e	bf e3 05 40 00	mov **$0x4005e3**, %edi
400543	e8 c8 fe ff ff	callq 400410 <puts@pllt>
400548	5d	pop %rbp
400549	c3	retq

%edi

e0 0e 5 40 00

PC

400539

IR

e9 d2 fe ff ff

Memory address	Machine code	Corresponding assembly language
400530	55	push %rbp
400531	48 89 e5	mov %rsp,%rbp
400534	bf e0 05 40 00	mov $0x4005e0, %edi
400539	**e9 d2 fe ff ff**	**callq 400410 <puts@plt>**
40053e	bf e3 05 40 00	mov **$0x4005e3**, %edi
400543	e8 c8 fe ff ff	callq 400410 <puts@pllt>
400548	5d	pop %rbp
400549	c3	retq

%edi

e0 05 40 00

PC

400410 (start of puts)

IR

???

Memory address	Machine code	Corresponding assembly language
400530	55	push %rbp
400531	48 89 e5	mov %rsp,%rbp
400534	bf e0 05 40 00	mov $0x4005e0, %edi
400539	e9 d2 fe ff ff	callq 400410 <puts@plt>
40053e	bf e3 05 40 00	mov **$0x4005e3**, %edi
400543	e8 c8 fe ff ff	callq 400410 <puts@pllt>
400548	5d	pop %rbp
400549	c3	retq

%edi

e0 05 40 00

PC

40053e

IR

bf e3 05 40 00

Memory address	Machine code	Corresponding assembly language
400530	55	push %rbp
400531	48 89 e5	mov %rsp,%rbp
400534	bf e0 05 40 00	mov $0x4005e0, %edi
400539	e9 d2 fe ff ff	callq 400410 <puts@plt>
40053e	**bf e3 05 40 00**	**mov $0x4005e3, %edi**
400543	e8 c8 fe ff ff	callq 400410 <puts@pllt>
400548	5d	pop %rbp
400549	c3	retq

%edi

e3 05 40 00

PC

400549

IR

c3

Back to OS

Memory address	Machine code	Corresponding assembly language
400530	55	push %rbp
400531	48 89 e5	mov %rsp,%rbp
400534	bf e0 05 40 00	mov $0x4005e0, %edi
400539	e9 d2 fe ff ff	callq 400410 <puts@plt>
40053e	bf e3 05 40 00	mov $0x4005e3, %edi
400543	e8 c8 fe ff ff	callq 400410 <puts@pllt>
400548	5d	pop %rbp
400549	**c3**	**retq**

%edi

e3 05 40 00

Data formats

In Intel/AMD-speak, integer types have different names than in C. Again, this is for historical reasons

C declaration	Data type	AMD64 suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	double word	l	4
long	quad word	q	8
All pointers	quad word	q	8

Examples:

`movq %rax,%rbx` means move (copy) 64 bits from %rax to %rbx
`mov1 %eax,%ebx` means move (copy) the least significant 32 bits from %rax to %rbx. The most significant 32 bits are not affected.

Assembly Language Example

C Code: Add two signed integers

```
long add(long x, long y) { return x + y; }
```

Assembly

```
    movq %rsi, %rax  # copy y into %rax
    addq %rdi, %rax  # add  x   to $rax
    ret
```

By convention, the first 2 parameters are passed in %rdi and %rsi. The return value is passed in %rax.

Sort of like

```
    %rax = %rsi
    %rax += %rdi
    return %rax
```

Common x86-64 Instructions

In 2-operand instructions, 1st operand is *source*, 2nd operand is *destination*. In 1-operand instructions, operand is *destination*. Many instructions have variants, such as `mov, movq, movl, movw, movb.` For most of the operations, both operands may be registers, or one may be a memory address and the other a register. Almost no instructions allow 2 memory addresses as their operands.

Instruction	# of operands	Meaning
mov	2	copy from src to dest
add	2	add src to dest; store result in dest
sub	2	subtract src from dest; store result in dest
imul	2	integer multiplication; store result in dest
cmp	2	Compare dest with src; flags are set accordingly
inc	1	increment dest
dec	1	decrement dest
neg	1	negate dest (e.g., 1 becomes -1)

Common x86-64 Instructions

Instruction	# of operands	Meaning
not	1	bitwise not dest
and	2	bitwise and dest and src; store result in dest
or	2	bitwise or dest and src; store result in dest
xor	2	bitwise xor dest and src; store result in dest
sal	2	shift (arithmetic) left. dest is shifted by src bits.
sar	2	shift (arithmetic) right. dest is shifted by src bits.
shr	2	shift (logical) right. dest is shifted by src bits.
lea	2	Load effective address. To be discussed later.
push	1	Push a value onto the program stack. To be discussed later
pop	1	Pop a value from the program stack. To be discussed later.

Special registers

- Some registers are still **only** used for special purposes. These are:

  %rip: instruction register. Contains the instruction currently being executed. Almost never seen in assembly language code.

  %rsp: stack pointer. Almost always used in one particular way (more later)

  %rbp: base pointer. Almost always used in one particular way (more later)

- Others may be used for general purposes, but by convention are used as follows:

  %rdi, %rsi, %rdx, %rcx, %r8, %r9: used for parameter passing (although they may also be used for other purposes)

  %rax: used to return a value from a function (although it might be used for other purposes)

Examples

```
int same(int x) {
    return x; }

same:  movl %edi, %eax
       ret
```

```
int add(int x, int y) {
    int s = x + y;
    return s; }

add: movl %rsi, %eax
     addl %rdi, %eax
     ret
```

```
unsigned long times8(unsigned long x) {
    return x * 8; }

times8:
    movq %rdi, %rax
    shlq $3, %rax
    ret
```

Examples

```
int negative(int x) {
   return -x; }

negative: movl %edi,%eax
          negl %eax
          ret
```

```
char abs (char x) {
       if (x >= 0) return x;
       else return -x; }

abs:
     movl    %edi, %edx
     movl    %edi, %eax
     sarb    $7, %dl
     xorl    %edx, %eax
     subl    %edx, %eax
     ret
```

abs(1)

```
abs:
      movl    %edi, %edx

      movl    %edi, %eax

      sarb    $7, %dl       # why 7?

      xorl    %edx, %eax

      subl    %edx, %eax

ret
```

%rdi/%edi	%rdx/%edx	%rax%eax
0x01	**0x01**	
0x01	0x01	**0x01**
0x01	**0x00**	0x01
0x01	0x00	**0x01**
0x01	0x00	**0x01**

abs(1) is 1

abs(-3)

```
abs:
    movl    %edi, %edx
    movl    %edi, %eax
    sarb    $7, %dl
    xorl    %edx, %eax
    subl    %edx, %eax
    ret
```

#rsi /#esi	#rdx / #edx	#rax / %eax
0xfd	**0xfd**	
0xfd	0xfd	**0xfd**
0xfd	**0xff**	0xfd
0xfd	0xff	**0x02**
0xfd	0xff	**0x03**

Notes:

- -3_{10} is 0xfd (in 8-bit 2s complement)
- 0xff ^ 0xfd is 0x02 (see next slide)
- 0x02 – 0xff is 0x03 (using 8-bit 2s complement)

abs(-3) = 3

```
abs:
    movl    %edi, %edx
    movl    %edi, %eax
    sarb    $7, %dl
    xorl    %edx, %eax
    subl    %edx, %eax
    ret
```

abs(-3)

%rdi	%rdx	%rax
0xfd	0xff	0xfd -> 0x02

11111111 ^ 11111101 = 00000010

```
abs:
    movq %rdi,%rdx # %rdx = x
    sarq $63,%rdx  # shift right – why 63 bits?
    movq %rdx, %rax
    xorq %rdi, %rax
    subq %rdx, %rax
    ret
```

abs(-3)

%rdi	%rdx	%rax
0xfd	0xff	0x03

2 – (-1) = 3

Condition codes

- The CPU maintains 1-bit registers, which store information about the results of the previous instruction. Usually the instructions are a variant of `cmp` but other instructions also set the condition codes, such as add, subtract, shift, etc. However, there are very few instructions which allow us to directly examine these flags

Code	Meaning
CF (carry flag)	The most recent operation generated a carry out of the most significant bit. Overflow for unsigned operations, including left logical shifts.
ZF (zero flag)	The most recent operation yielded 0. Note that `cmp` performs a subtraction, so ZF indicates the the numbers are equal
SF (sign flag)	The most recent operation yielded a negative number.
OF (overflow flag)	The most recent operation yielded a 2s complement overflow (from positive to negative or vice versa)

The `cmp` and `test` instructions

- Both compare numbers
- `cmp` subtracts one from the other
  e.g., `cmp %edx, %eax` computes `%eax - %edx`
- `test` performs a bitwise-and
- yields different condition codes in certain cases
- Example: `%eax is 8, %edx is 6`

  `cmpl %edx, %eax` sets CF and SF
  `testl %edx, %eax` sets ZF

  `test` is often used to see if a register contains 0 (ZF is set)

`set` instructions

- Usually follow a `cmp or test` instruction
- Sets *dest* based on condition code registers

Instruction	# of operands	Meaning
sete	1	*Dest* is set to ZF
setne	1	*Dest* is set to ~ZF
setge	1	*Dest* is set to ZF | ~SF
setg	1	*Dest* is set to ~ZF & ~SF
setle	1	*Dest* is set to ZF | SF
setl	1	*Dest* is set to ~ZF & SF

Example

```
int less(x, y) {
  return x < y;
}
```

```
less:
      xorl    %eax, %eax          # %eax = 0
      cmpl    %esi, %edi          # compare %edi with %esi (computes %edi - %esi)
      setl    %al                 # if negative (%edi < %esi) , set %al to 1
      ret
```

Some (hopefully) easy ones

```
f:
        movl    %edi, %eax
        shrl    $31, %eax
         ret

g:
        movl    %edi, %eax
        andl    $1, %eax
        xorl    $1, %eax
        ret
```

```
h:
        cmpl    %esi, %edi
        setl    %cl
        xorl    %eax, %eax
        cmpl    %edx, %esi
        setl    %al
        andl    %ecx, %eax
        ret
```

64-bit register	32-bit register	Arg #
%rdi	%edi	1
%rsi	%esi	2
%rdx	%edx	3

Some (hopefully) easy ones

We didn't get to h on 1/30

```
f:
        movl    %edi, %eax
        shrl    $31, %eax
         ret

g:
        movl    %edi, %eax
        andl    $1, %eax
        xorl    $1, %eax
        ret
```

```
h:
        cmpl    %esi, %edi
        setl    %cl
        xorl    %eax, %eax
        cmpl    %edx, %esi
        setl    %al
        andl    %ecx, %eax
        ret
```

```
Int f(int x) {
   return x < 0;
}

int g(int x)
   return x % 2 == 0;
}
```

64-bit register	32-bit register	Arg #
%rdi	%edi	1
%rsi	%esi	2
%rdx	%edx	3

The `jmp` instructions

- Behavior depends on the condition flags
- Every `jmp` you could possibly imagine

OP	Jump if …	Flags
jmp	always	-
je	equal	ZF
js	Sign	SF
jns	Not sign	~SF
jg	greater	~(SF^OF)&~ZF
jge	greater or equal	~(SF^OF)

OP	Jump if …	Flags
jl	Less	SF ^ OF
jle	Less or equal	(SF ^ OF) | ZF
ja	Above	~CF & ~ZF
jae	Above or equal	~CF
jb	Below	CF
jbe	Below or equal	CF | ZF

The C goto statement

C

```
int greater(int x, int y) {
      int diff = x – y;
      if (diff > 0)
          goto retx;
      return y;
 retx: return x;
}
```

64-bit register	32-bit register	Arg #
%rdi	%edi	1
%rsi	%esi	2
%rdx	%edx	3

x86-64

```
greater:
   cmpl    %edi, %esi
   jl      .L2
   movl    %edi, %eax
   jmp     .L4
.L2:
   movl    %esi, %eax
.L4:
   ret
```

The C goto statement

```
int power(int x, int y) {
       int a = 1;
       int i=0;
start: if (i == y)
           goto end;
       a *= x;
       i++;
       goto start;
end:   return a;
}
```

```c
int power(int x, int y) {
        int a = 1;
        int i=0;
    L6: if (i == y)
            return a;
        a *= x;
        i++;
        goto L6;
}
```

```
power:
        xorl    %edx, %edx
        testl   %esi, %esi
        movl    $1, %eax
        je      .L5
.L6:
        addl    $1, %edx
        imull   %edi, %eax
        cmpl    %esi, %edx
        jne     .L6
.L5:
        ret
```

64-bit register	32-bit register	Arg #
%rdi	%edi	1
%rsi	%esi	2
%rdx	%edx	3

The C goto statement: factorial function

```
factorial:
        cmpl    $1, %edi
        jle     .L4
        addl    $1, %edi
        movl    $2, %edx
        movl    $1, %eax
.L3:
        imull   %edx, %eax
        addl    $1, %edx
        cmpl    %edi, %edx
        jne     .L3
        ret
.L4:
        movl    $1, %eax
        ret
```

Flow of control

- No if…else or looping constructs in assembly language
- Instead, `set`, `test`, `cmp`, `snd jmp` are used to create the same effects
- “Spaghetti code”

Another `goto` example in C

```
void print_equal(int x, int y) {
   if (x != y)
     printf("Not equal\n");
   else printf("Equal\n");

void print_equal(int x, int y) {
   if (x != y) {
     printf("Not equal\n");
     goto end;
   printf("Equal\n");
  end: return
}
```

64-bit register	32-bit register	Arg #
%rdi	%edi	1
%rsi	%esi	2
%rdx	%edx	3

```
print_equal:
        cmpl    %esi, %edi
        je      .L2
        movl    $.LC0, %edi
        jmp     puts
.L2:
        movl    $.LC1, %edi
        jmp     puts


print_equal_goto:
        cmpl    %esi, %edi
        je      .L5
        movl    $.LC0, %edi
        jmp     puts
.L5:
        movl    $.LC1, %edi
        jmp     puts
```

Memory addressing

- Finite number of registers

- Eventually, "main memory" must be used to store working data
    - **Virtual** address space

- Difficult to demonstrate with simple C programs, unless compilation is not optimized

```
int mult(int x, int y) {
   return x * y;
}

> gcc –o mult.s mult.c –S –O2

mult:
    movl    %edi, %eax
    imull   %esi, %eax
    ret
```

Memory addressing

```
int mult(int x, int y) {
   return x * y;
}

> gcc –o mult.s mult.c –S

mult:
    pushq   %rbp                    # ignore
    movq    %rsp, %rbp              # ignore
    movl    %edi, -4(%rbp)          # int a = x (a is in memory)
    movl    %esi, -8(%rbp)          # int b = y (b is in memory)
    movl    -4(%rbp), %eax          # ans = a
    imull   -8(%rbp), %eax          # ans *= b
    popq    %rbp                    # ignore
    ret                             # return ans
```

Operand types

- **Immediate**: constant value in decimal or Hex; number preceded by $
- **Register**: starts with %
- Memory reference

several different ways to specify an operand's memory address

Most generally D(B,I,s)

D = *displacement*
B = *base register*
I = *index register*
s = *scale*

Memory Operand types

D = *displacement*
B = *base register*
I = *index register*
s = *scale*

- **Absolute**: give memory location D (rarely used)
- **Indirect**: specify a register; it contains the memory location (B)
- **Base + displacement**: specify a register, add a value to its address (like pointer arithmetic) D(B)
- **Indexed**: specify 2 registers, or 2 registers + a constant (B,I) or D(B,I)
- **Scaled indexed**: multiply by the scale *s* (B,I,s) or D(B,I,s)

Addressing type	Syntax	Example	Result
Immediate	number preceded by $	movq $10, %rax	%rax set to 10
Register	% before name	movq %rax,%rdx	the contents of the register %rax are copied into %rdx
Memory	D(B,I,S)		

Memory operands Can have 4 components:
1. Displacement
2. Base
3. Index
4. Scale
 	
Base and index are registers
Displacement and scale are integers.
	Any may be left out, although the syntax varies a bit	movq %rax, (%rbx)
		movl 4(%rbx),%rax
		movl (%rbx,%rcx,4), %rdx
			Assume %rax is x, %rbx is the pointer p, %rdx is y, and %rcx contains 5
			
			*p = x
			x = p[1]
			y = p[5]

C examples

data is in memory, not a register **lea: Load Effective Address**

Loads a memory address but does not retrieve/store data from that address

leaq -4(%rbp), %rdx Store the address computed by subtracting 4 from the contents of %rbp

Can also be used for arithmetic computations

leaq (,%rdi,4), %rax   # %rax = x * 4

does **not** access memory

Scale is dependent on datatype size

Data type	C example	Instruction example
int	int x = 3;	movq $3,-4(%rbp)
int []	y[i] = 0;	movq $0,-32(%rbp,%rdx,4)
int *	p = &x;	leaq -4(%rbp),%rdx

Exercise

$D(B,I,S) = B + (S*I) + D$

Assume the following values are stores in the following values and registers. Fill in the following tables.

Memory Location	Value
0x100	0x000000ff
0x104	0x000000ab
0x108	0x00000013
0x10c	0x00000011

Register	Value
%rax	0x0000000000000100
%rcx	0x0000000000000001
%rdx	0x0000000000000003

Operand	Value
%rax	
$0x108	
(%rax)	
4(%rax)	

Operand	Value
9(%rax, %rdx)	
256(%rcx,%edx)	
0xfc(,%ecx,4)	
(%rax,%rdx,4)	

Fill in the following table showing the effect of each of the instructions below. Assume the values in in memory and registers are as specified above. Assume the instructions are **not** sequential.

Instruction	Destination	New Value in Destination
movq %rax, (%rax)		
addl 4(%rax), %ecx		
subq %rdx, (%rax, %rcx, 4)		
movq $-1, 4(%rax)		
movzbq $0x61, 4(%rax,%rcx,4)		
movsbq $-1,%rdx		

Note: the numbers in red should all be 64 bits, but for brevity I have written most of them as 16 bit numbers. This is also true of the previous slide.

Answers to Exercise

$D(B,I,S) = B + (S*I) + D$

Assume the following values are stores in the following values and registers. Fill in the following tables.

Memory Location	Value
0x100	0x000000ff
0x104	0x000000ab
0x108	0x00000013
0x10c	0x00000011

Register	Value
%rax	0x0000000000000100
%rcx	0x0000000000000001
%rdx	0x0000000000000003

Operand	Value
%rax	0x100
$0x108	0x108
(%rax)	0xff
4(%rax)	0xab

Operand	Value
9(%rax, %rdx)	0x11
256(%rcx,%edx)	0xab
0xfc(,%ecx,4)	0xff
(%rax,%rdx,4)	0x11

Answers to exercise

Fill in the following table showing the effect of each of the instructions below. Assume the values in in memory and registers are as specified above. Assume the instructions are **not** sequential.

Instruction	Destination	New Value in Destination
movq %rax, (%rax)	0x100	0x100
addl 4(%rax), %ecx	%ecx	0xac
subq %rdx, (%rax, %rcx, 4)	0x104	0xa8
movq $-1, 4(%rax)	0x104	0xffffffffffffffff
movzbq $0x61, 4(%rax,%rcx,4)	0x108	0x00000…00000061
movsbq $-1,%rdx	%rdx	0xffffffffffffffff

Note: the numbers in red should all be 64 bits, but for brevity I have written most of them as 16 bit numbers. This is also true of the previous slide.

What does this function do?

```
f:
        movl    (%rdi), %eax
        movl    (%rsi), %edx
        movl    %edx, (%rdi)
        movl    %eax, (%rsi)
        ret
```

64-bit register	32-bit register	Arg #
%rdi	%edi	1
%rsi	%esi	2
%rdx	%edx	3

lea – load effective address

Source: Memory operand D(B,I,s)

Value: D(B,I,s) NOT THE CONTENTS OF D(B,I,s)

Like & operator in C

Sometimes used to do arithmetic

Fill in the following table showing the effect of each of the instructions below. Assume the values in in memory and registers are as specified above. Assume the instructions are **not** sequential.

Instruction	Destination	New Value in Destination
leal (%rax%, %rax), %rdx	%rdx	0x200
leal 4(%rax), %ecx	%eax	0x104
leal (%rcx, %rdx), %rax	%rax	0x4

Note: the numbers in red should all be 64 bits, but for brevity I have written most of them as 16 bit numbers. This is also true of the previous slide.

Strings as parameters

```c
int mystrlen(char *s) {
  int i;
  for (i=0; s[i]!='\0'; i++);
  return i;
}
```

```
mystrlen:
        cmpb    $0, (%rdi)
        je      .L4
        addq    $1, %rdi
        xorl    %eax, %eax
.L3:
        addq    $1, %rdi
        addl    $1, %eax
        cmpb    $0, -1(%rdi)
        jne     .L3
        ret
.L4:
        xorl    %eax, %eax
        ret
```

Array vs. Pointer syntax example

```
int strlen373(char s[]) {
  int i;
  for (i=0; s[i] != '\0'; i++);
  return i;
}

int strlen373ptr(char *s) {
  int i;
  for (i=0; *s++ != '\0'; i++);
  return i;
}
```

```
strlen373:
    cmpb   $0, (%rdi)
    je     .L4
    addq   $1, %rdi
    xorl   %eax, %eax
.L3:
    addq   $1, %rdi
    addl   $1, %eax
    cmpb   $0, -1(%rdi)
    jne    .L3
    ret
.L4:
    xorl   %eax, %eax
    ret
```

```
strlen373ptr:
    xorl   %eax, %eax
    cmpb   $0, (%rdi)
    leaq   1(%rdi), %rdx
    je     .L9
.L8:
    addq   $1, %rdx
    addl   $1, %eax
    cmpb   $0, -1(%rdx)
    jne    .L8
    ret
.L9:
    ret
```

64-bit register	32-bit register	Arg #
%rdi	%edi	1
%rsi	%esi	2
%rdx	%edx	3

```
int sum_of_squares (int n) {
  int ans = 0;
  int i;
  for (i=1; i<=n; i++)
    ans += i*i;
  return ans;
  }
```

```
int sum_of_squares(int n) {
  int ans = 0;
  int i=1;
 L1: if (i>n) goto L2;
  ans += i*i;
  i++;
  goto L1;
 L2:  return ans;
}
```

64-bit register	32-bit register	Arg #
%rdi	%edi	1
%rsi	%esi	2
%rdx	%edx	3

```
 sum_of_squares:
        testl   %edi, %edi
        jle     .L4
        addl    $1, %edi
        movl    $1, %edx
        xorl    %eax, %eax
.L3:
        movl    %edx, %ecx
        imull   %edx, %ecx
        addl    $1, %edx
        addl    %ecx, %eax
        cmpl    %edi, %edx
        jne     .L3
        rep ret
.L4:
        xorl    %eax, %eax
        ret
```

The C code compiles the same in either case

```c
// convert 'F' to 'C' and vice versa
void scale(char s1, char *s2) {
   if (s1 == 'F')
     *s2 = 'C';
   else if (s1 == 'C')
     *s2 = 'F';
}
```

64-bit register	32-bit register	8 bits	Arg #
%rdi	%edi	%dil	1
%rsi	%esi	%sil	2
%rdx	%edx	%dl	3

```
scale:
      cmpb $70, %dil      # s1 == 'F'
      je .L6              # Yes? Goto L6
      cmpb $67, %dil      # s1 == 'C'
      je .L7              # Yes? Goto L7
      ret
.L7:  movb $70, (%rsi)    # *s2 = 'F‘; indirect addressing
      ret
.L6:  movb $67, (%rsi)    # *s2 = 'C‘; indirect addressing
      ret
```

What does this function do?

```
f:
        testl   %esi, %esi
        jle     .L4
        xorl    %edx, %edx
        xorl    %eax, %eax
.L3:
        addl    (%rdi,%rdx,4), %eax
        addq    $1, %rdx
        cmpl    %edx, %esi
        jg      .L3
        ret
.L4:
        xorl    %eax, %eax
        ret
```

64-bit register	32-bit register	Arg #
%rdi	%edi	1
%rsi	%esi	2
%rdx	%edx	3

- L3: memory operand base, index, scale
- How many parameters?
- What type of data?
- Is there a loop?

What does this function do?

```
f:
        testl   %esi, %esi
        jle     .L4
        xorl    %edx, %edx
        xorl    %eax, %eax
.L3:
        addl    (%rdi,%rdx,4), %eax
        addq    $1, %rdx
        cmpl    %edx, %esi
        jg      .L3
        ret
.L4:
        xorl    %eax, %eax
        ret
```

testl %esi, %esi ← parameter

xorl %edx, %edx ← Not parameter

cmpl %edx, %esi ← parameter

64-bit register	32-bit register	Arg #
%rdi	%edi	1
%rsi	%esi	2
%rdx	%edx	3

- L3: memory operand base, index, scale
- How many parameters? **2**
- Is there a loop?

Yes beginning with L3

What types of parameters?

%rdi: pointer
%rsi: integer

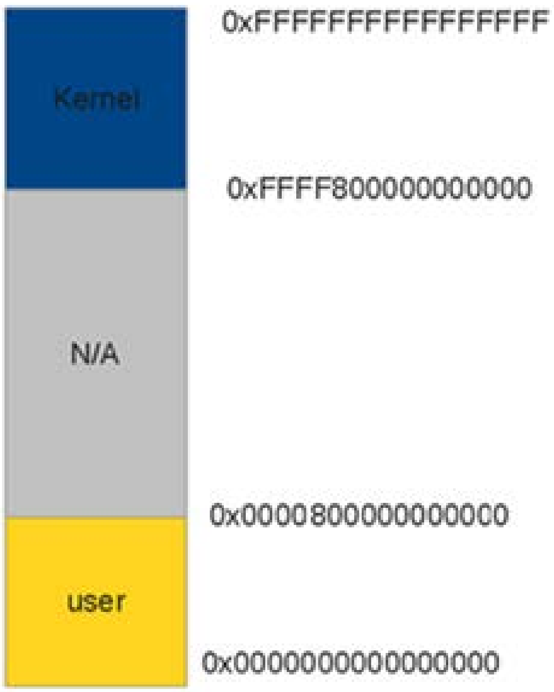
Returns the sum of an array of integers

Reverse Engineering problems

- Please see the accompanying .s file.

Division of address space

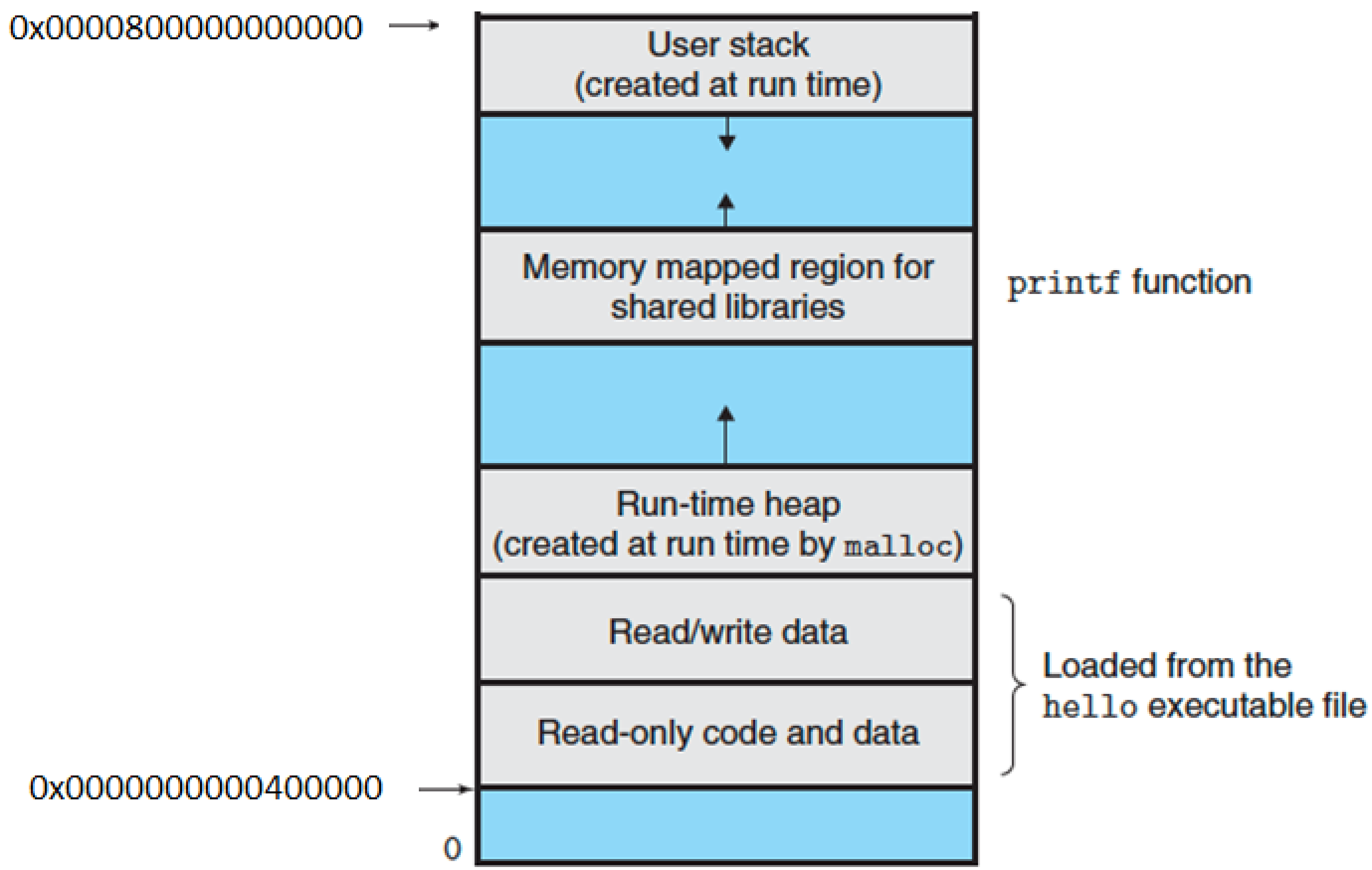
- Source: http://thinkiii.blogspot.com/2014/02/arm64-linux-kernel-virtual-address-space.html
- Not all 2^{64} bytes of the address space of a process are used;
- The "top" portion of the address space contains the OS "kernel" (its essentials); the bottom portion is your program and data. Note that the yellow is 2^{48} bytes, 2^{18} bytes are still unused.



kernel/arch/arm64 implementation

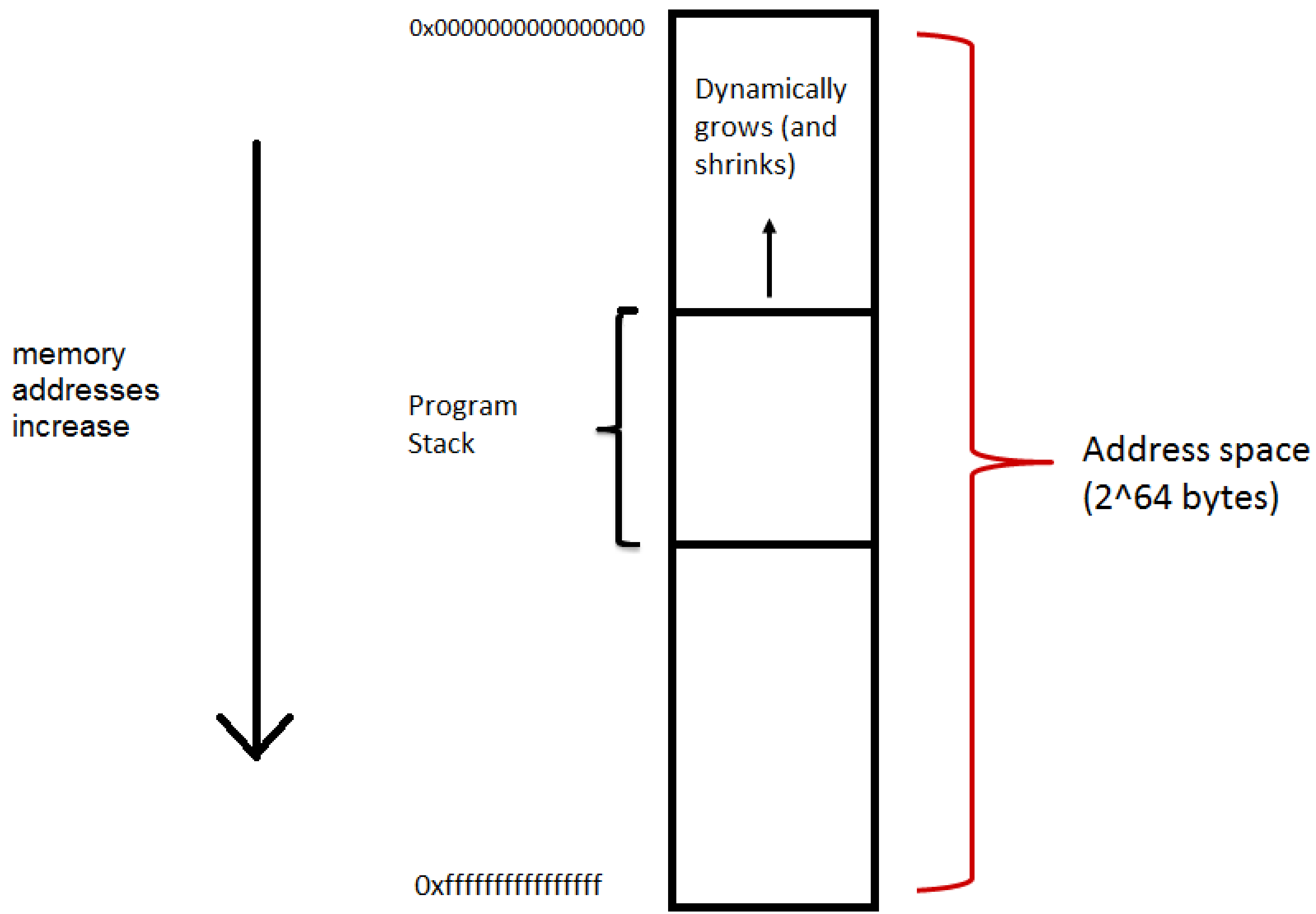
User address space

http://stackoverflow.com/questions/9511982/virtual-address-space-in-the-context-of-programming

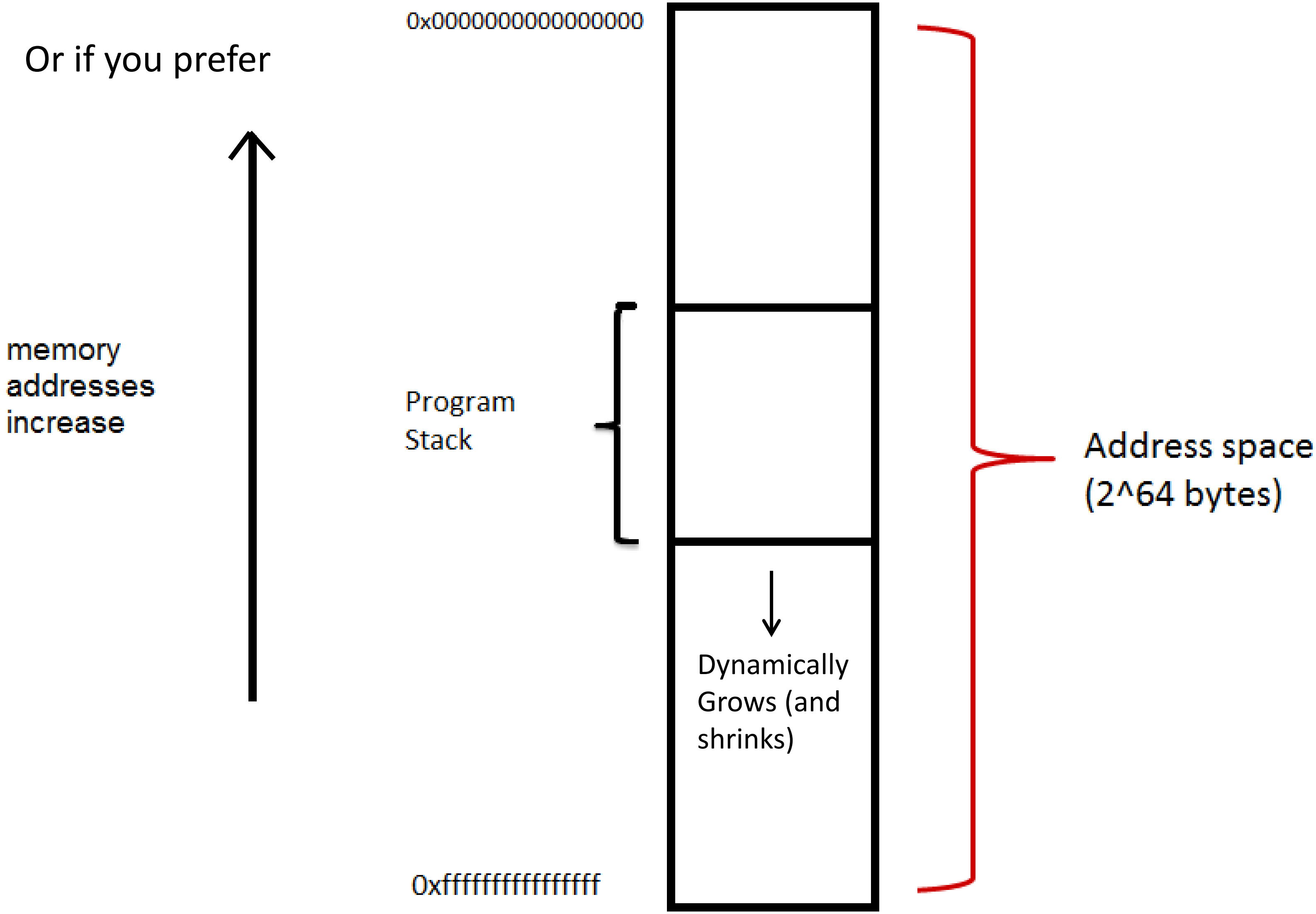


Program Stack

- Programs keep track of many things in a portion of memory called the **program stack**
- Instructions **push** and **pop** do what they sound like
- **%rsp** contains the address of the top of the stack
- When a function is called, it sets up a **stack frame** for itself
    - **%rsp** is a pointer to the top of the stack frame
    - **%rbp** is a pointer to the bottom of the stack frame


memory
addresses
increase
0x0000000000000000
Dynamically
grows (and
shrinks)
Program
Stack
0xffffffffffffffff
Address space
(2^64 bytes)

Or if you prefer



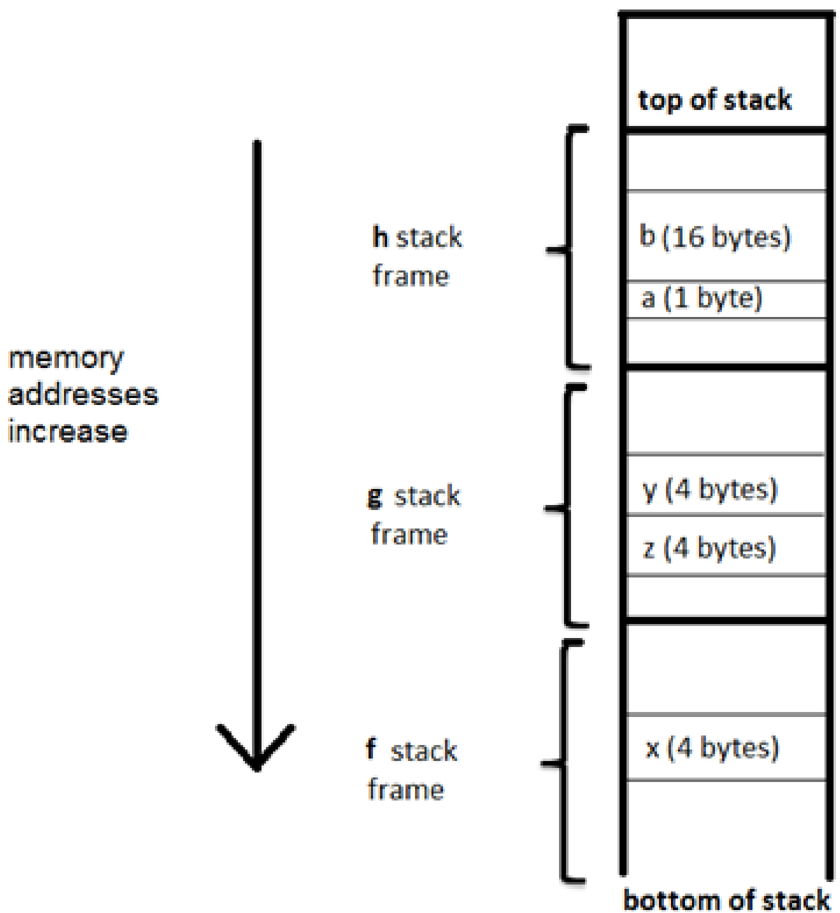
Orientation of the Program Stack

- Text. P. 90: "By convention, we draw stacks upside down, so that the "top" of the stack is shown at the bottom."

- My diagrams **do not** follow this convention, because it doesn't make sense.

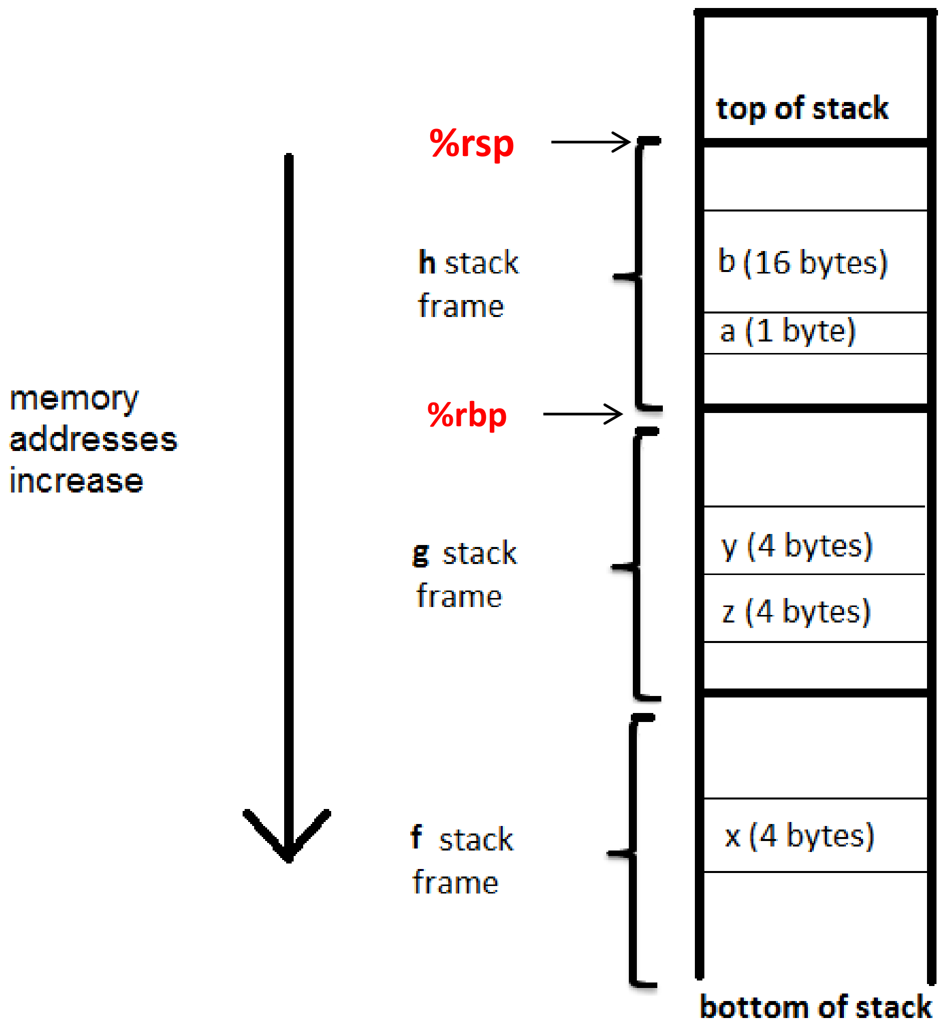
Use of the program stack

- When a function is called, it allocates room on the stack for its “stack frame”
- The frame contains space for local variables, register values that need to be stored away, and information needed to return to the calling function



State of stack while h is running

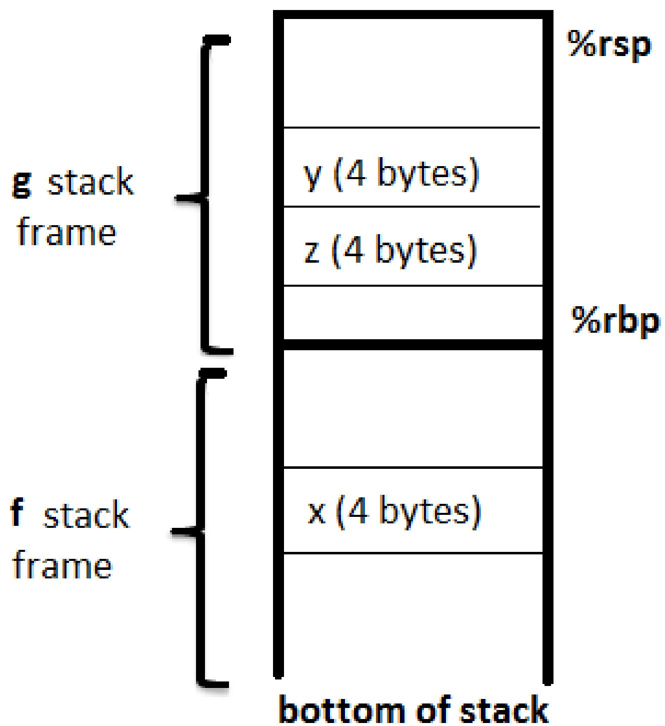
```
void f(...) {
        int x = 5;
        g(x);
}

void g(int x) {
        int y = 10;
        int z = 100;
        h();
}

void h() {
        char a;
        char b[16];
}
```



State of stack while h is running

```
void f(...) {
        int x = 5;
        g(x);
}

void g(int x) {
        int y = 10;
        int z = 100;
        h();
}

void h() {
        char a;
        char b[16];
}
```

- **Stack frame organization**




```
void f() {
        int x = 10;
        g(x);
}

void g(int x) {
        int y = 10;
        int z = 100;
        h();
}
```

Register %rsp contains the address of the "top" of the current stack frame

Register %rbp contains the address of the "bottom" of the current stack frame

pushq

- When a function is called, it creates its own stack frame

- It also needs to store information that is required to return to the calling function
  - Return address
  - Boundaries of the calling function’s stack frame

- Therefore, many functions start with

```
pushq %rbp
movq %rsp, %rbp
```

- push is a “macro”, short for

```
subq $16, %rsp
movq %rbp, (%rsp)
```

popq

- When a function returns, it restores the program stack to its previous state

- Therefore, many functions end with with

```
popq %rbp
```

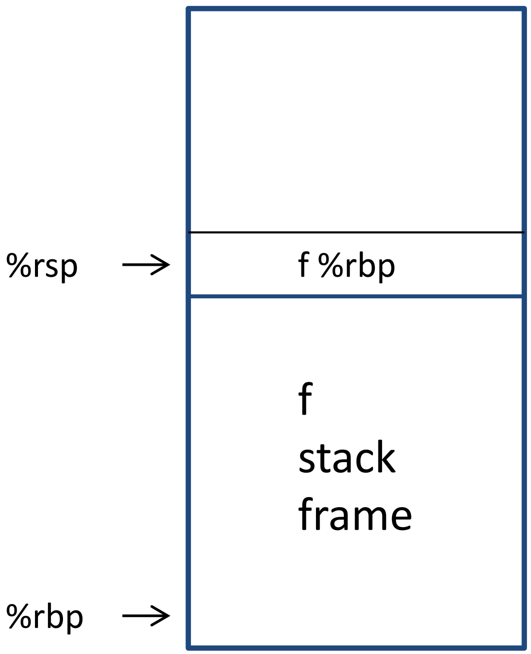
- pop is a “macro”, short for

```
movq (%rsp), %rbp
addq $16, %rsp
```

Example

```
int f(x) {
    return g(x);
}

int g(y) {
    return y * 2;
}
```

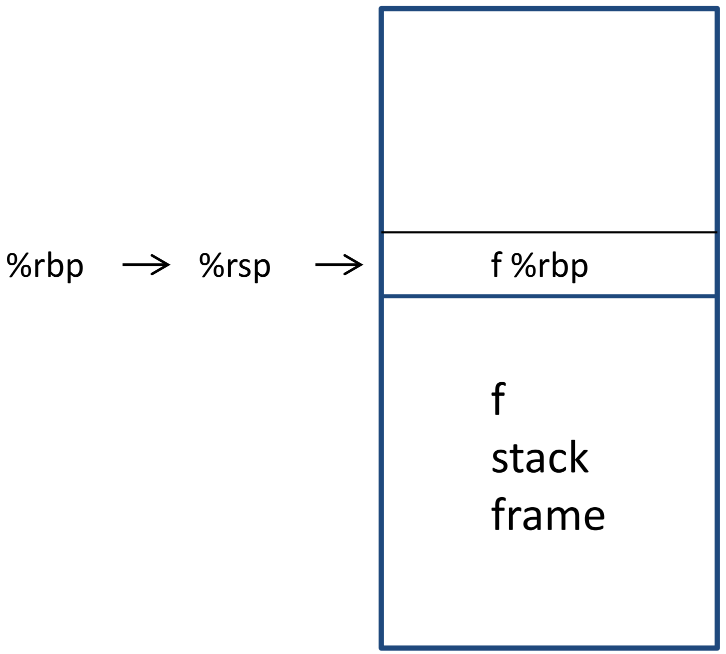
```
g:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $-4, %rsp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        addl    %eax, %eax
        addq    $4, %rsp
        popq    %rbp
        ret
```



Example

```
int f(x) {
    return g(x);
}

int g(y) {
    return y * 2;
}
```

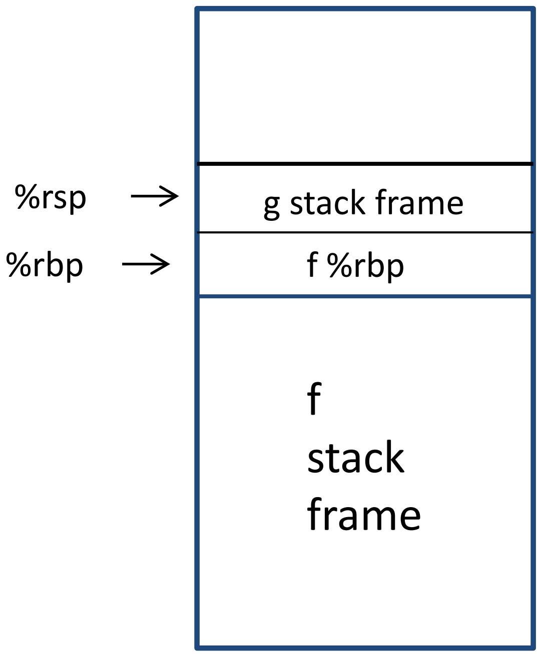
```
g:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $-4, %rsp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        addl    %eax, %eax
        addq    $4, %rsp
        popq    %rbp
        ret
```



Example

```
int f(x) {
    return g(x);
}

int g(y) {
    return y * 2;
}
```

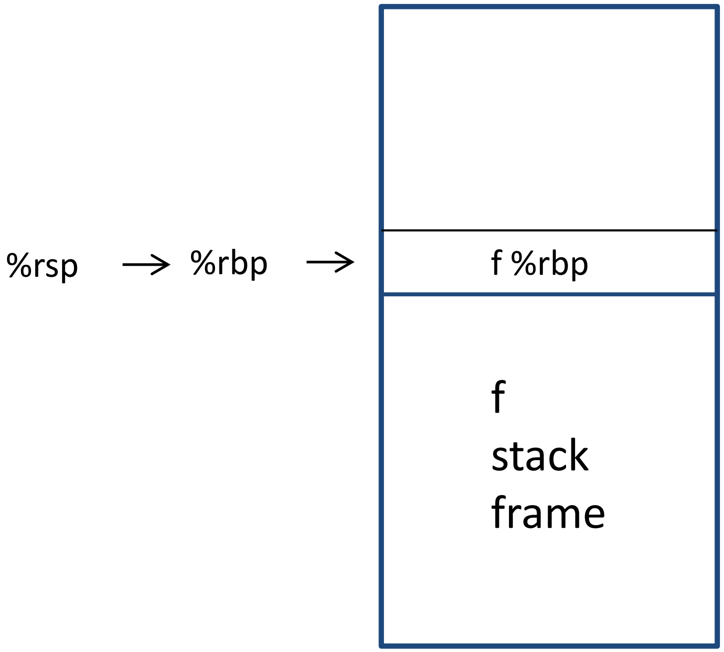
```
g:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $-4, %rsp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        addl    %eax, %eax
        addq    $4, %rsp
        popq    %rbp
        ret
```



Example

```
int f(x) {
    return g(x);
}

int g(y) {
    return y * 2;
}
```

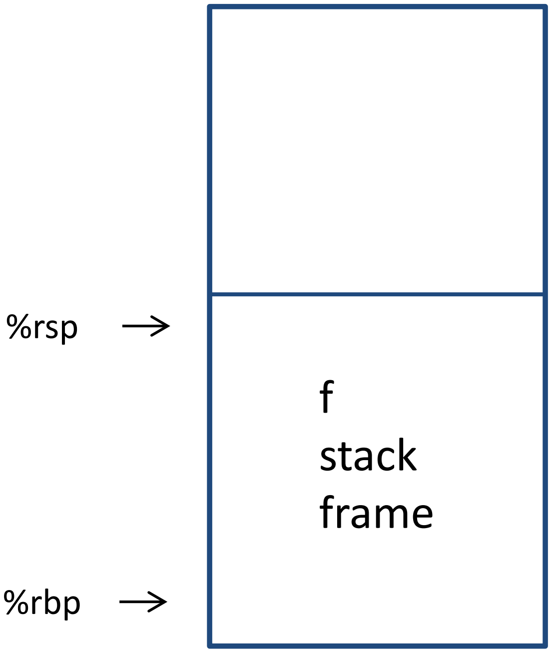
```
g:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $-4, %rsp
    movl    %edi, -4(%rbp)
    movl    -4(%rbp), %eax
    addl    %eax, %eax
    addq    $4, %rsp
    popq    %rbp
    ret
```



Example

```
int f(x) {
    return g(x);
}

int g(y) {
    return y * 2;
}
```

```
g:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $-4, %rsp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        addl    %eax, %eax
        addq    $4, %rsp
        popq    %rbp
        ret
```



pop example

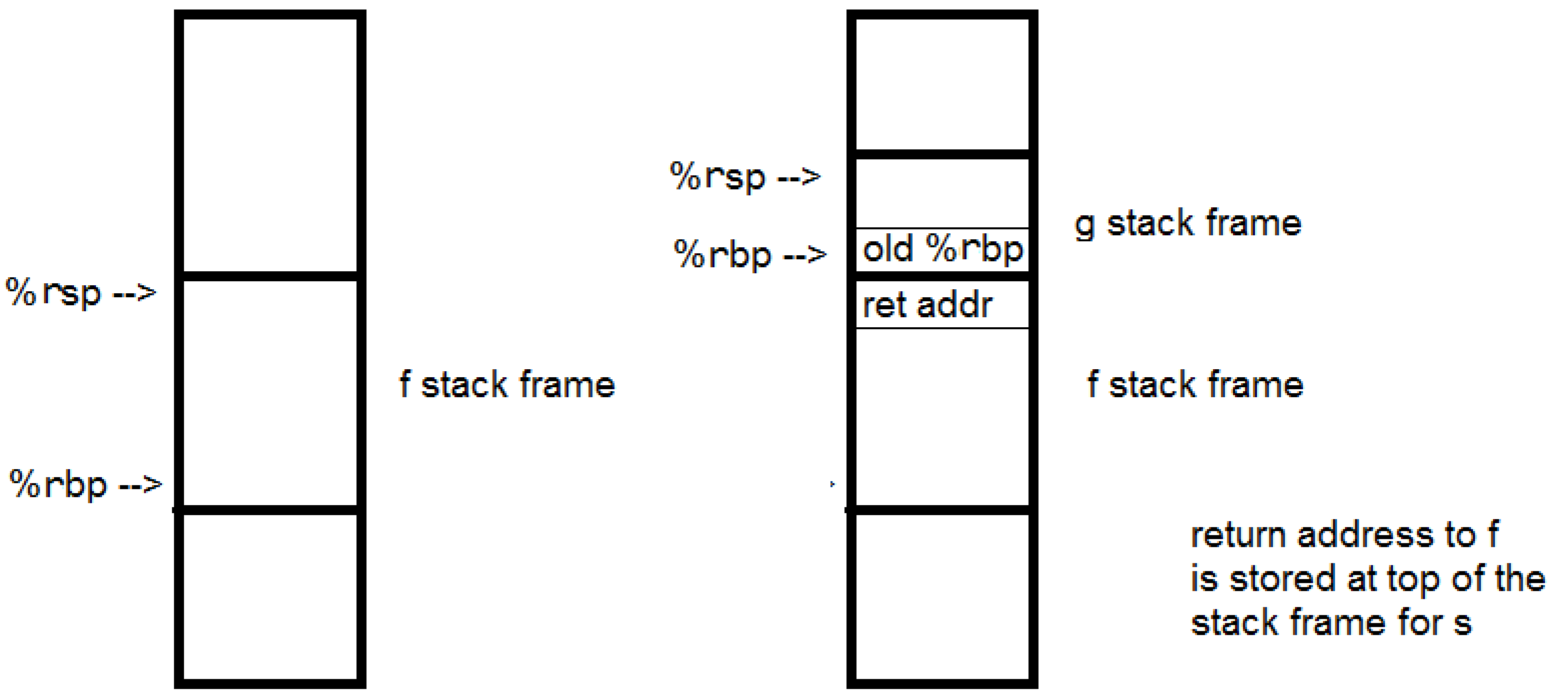
```
less:
    pushq   %rbp              # save %rbp from previous stack frame
    movq    %rsp, %rbp        # %rbp now points to the bottom of “less” stack frame
    subq    $16, %rsp         # adjust %rsp so that the stack frame is the right size
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %eax
    cmpl    -8(%rbp), %eax
    setl    %al
    movzbl  %al, %eax
    addq    $16, %rsp         # make “less” stack frame disappear
    popq    %rbp              # restore calling function’s base pointer
    ret
```

- `call` and `ret`
  - Also macros; `callq 0x400f01` means

```
pushq %rip
jmpq   0x400f01
```

and `ret` means

```
popq %rip
jmpq *%rip   # this is an indirect jump
```



Recursion

Recursive code is sometimes slower than iterative, because of the need to use the program stack.

Example: factorial

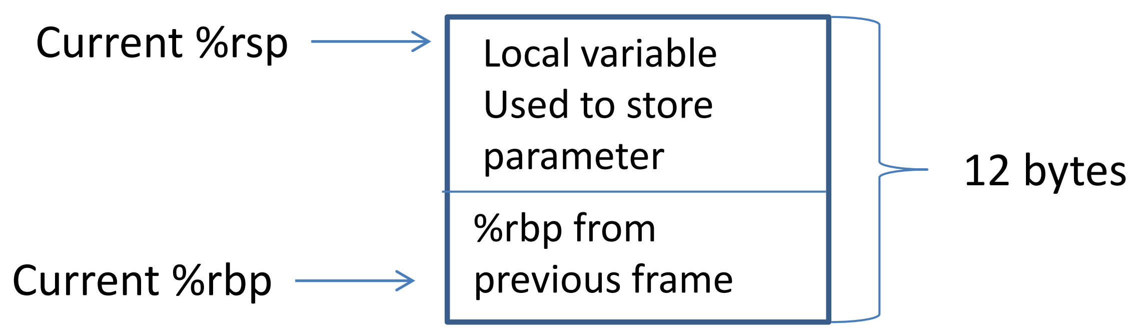
```
int fact_r(int x) {
  if (x <= 1)
    return 1;
  else return x * fact_r(x-1);
}
```

Where is x remembered?

First, using
gcc –o fact.s –s fact.c

```
fact_r:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    cmpl    $1, -4(%rbp)
    jg      .L2
    movl    $1, %eax
    jmp     .L3
.L2:
    movl    -4(%rbp), %eax
    subl    $1, %eax
    movl    %eax, %edi
    call    fact_r
    imull   -4(%rbp), %eax
.L3:
    leave
    ret
```

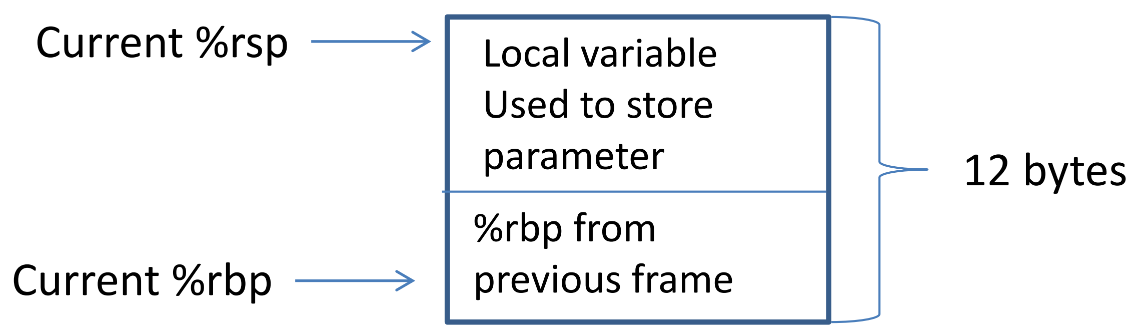
Each call to fact_r requires a stack frame



```
fact_r:
      pushq   %rbp
      movq    %rsp, %rbp
      subq    $16, %rsp
      movl    %edi, -4(%rbp)
      cmpl    $1, -4(%rbp)
      jg      .L2
      movl    $1, %eax
      jmp     .L3
.L2:
      movl    -4(%rbp), %eax
      subl    $1, %eax
      movl    %eax, %edi
      call    fact_r
      imull   -4(%rbp), %eax
.L3:
      leave
      ret
```

Each call to fact_r requires a stack frame

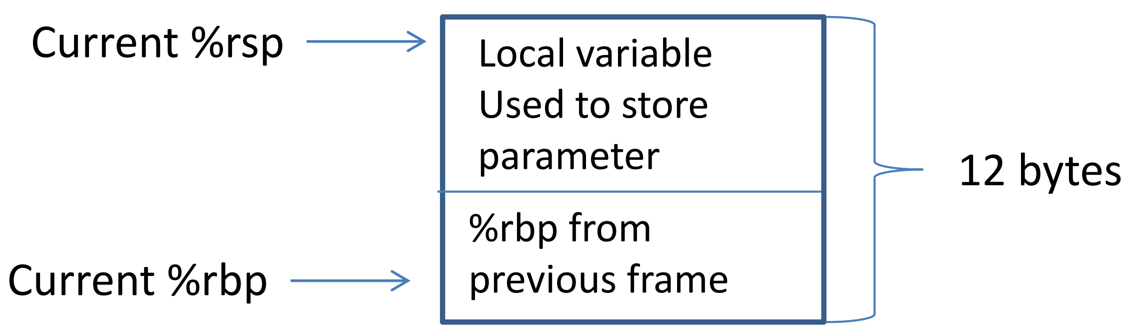
Establish stack frame



Each call to fact_r requires a stack frame

```
fact_r:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        cmpl    $1, -4(%rbp)
        jg      .L2
        movl    $1, %eax
        jmp     .L3
.L2:
        movl    -4(%rbp), %eax
        subl    $1, %eax
        movl    %eax, %edi
        call    fact_r
        imull   -4(%rbp), %eax
.L3:
        leave
        ret
```

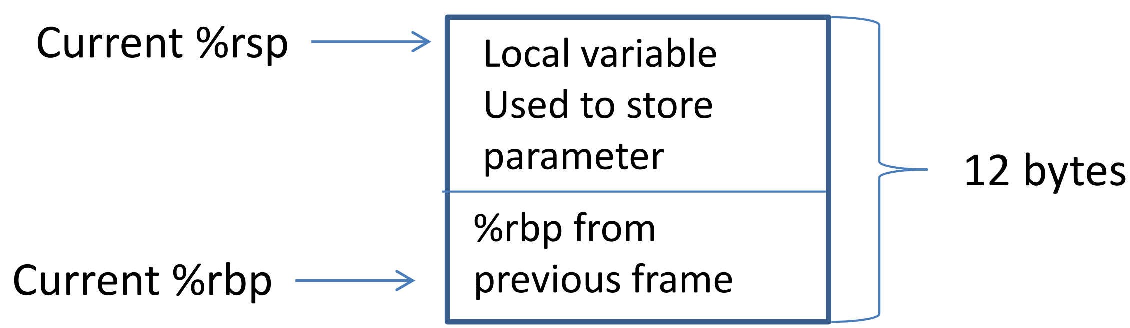
Store x



```
fact_r:
      pushq   %rbp
      movq    %rsp, %rbp
      subq    $16, %rsp
      movl    %edi, -4(%rbp)
      cmpl    $1, -4(%rbp)
      jg      .L2
      movl    $1, %eax
      jmp     .L3
.L2:
      movl    -4(%rbp), %eax
      subl    $1, %eax
      movl    %eax, %edi
      call    fact_r
      imull   -4(%rbp), %eax
.L3:
      leave
      ret
```

Each call to fact_r requires a stack frame

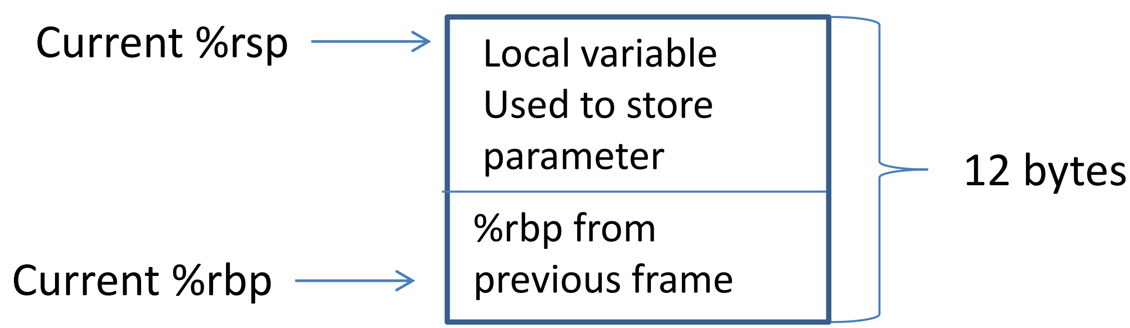
(1)! Is 1



```
fact_r:
      pushq   %rbp
      movq    %rsp, %rbp
      subq    $16, %rsp
      movl    %edi, -4(%rbp)
      cmpl    $1, -4(%rbp)
      jg      .L2
      movl    $1, %eax
      jmp     .L3
.L2:
      movl    -4(%rbp), %eax
      subl    $1, %eax
      movl    %eax, %edi
      call    fact_r
      imull   -4(%rbp), %eax
.L3:
      leave
      ret
```

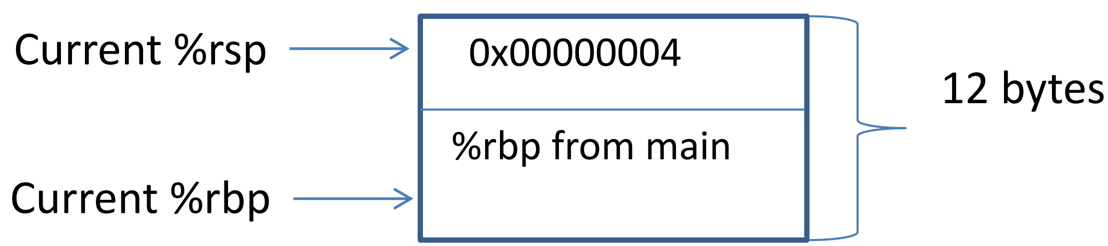
Each call to fact_r requires a stack frame

(n)! Is (n)[(n-1)!]



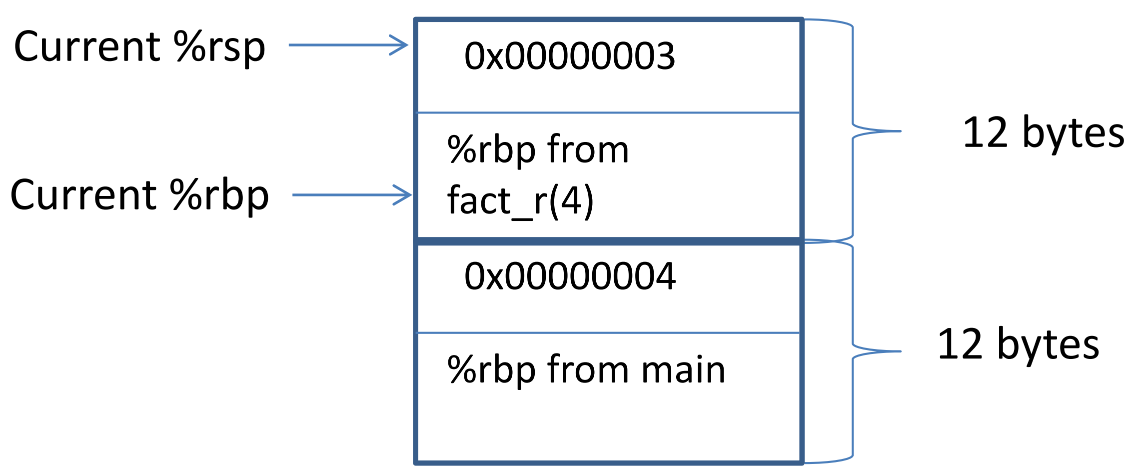
```
fact_r:
      pushq   %rbp
      movq    %rsp, %rbp
      subq    $16, %rsp
      movl    %edi, -4(%rbp)
      cmpl    $1, -4(%rbp)
      jg      .L2
      movl    $1, %eax
      jmp     .L3
.L2:
      movl    -4(%rbp), %eax
      subl    $1, %eax
      movl    %eax, %edi
      call    fact_r
      imull   -4(%rbp), %eax
.L3:
      leave
      ret
```

Consider fact_r(4):



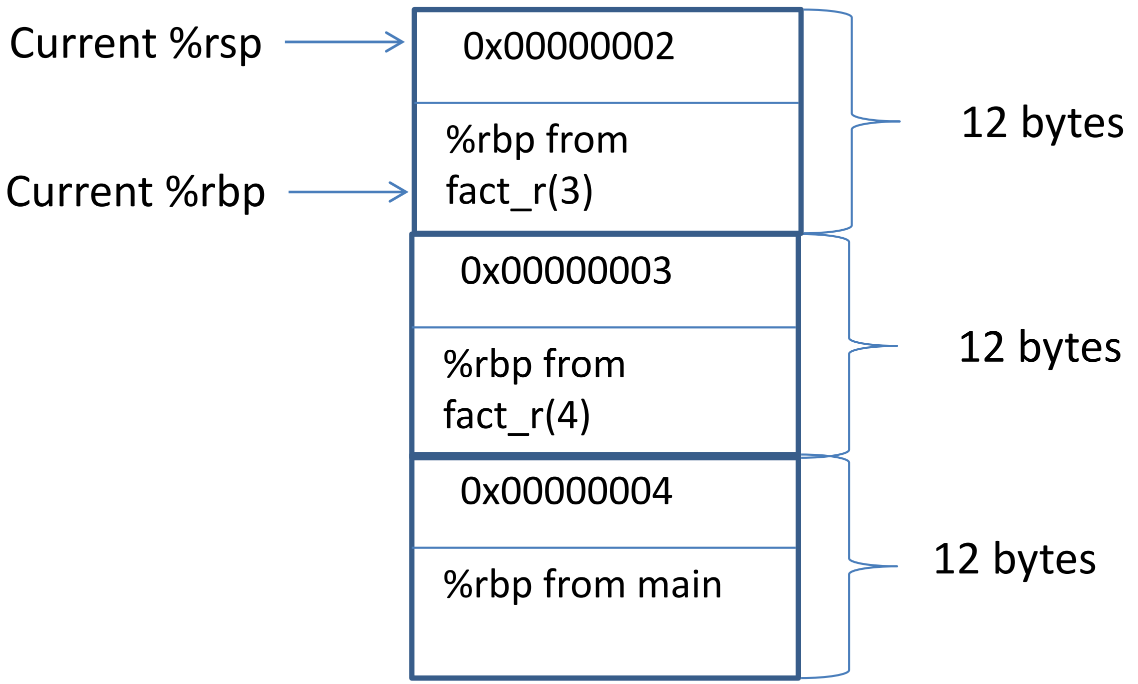
```
fact_r:
      pushq   %rbp
      movq    %rsp, %rbp
      subq    $16, %rsp
      movl    %edi, -4(%rbp)
      cmpl    $1, -4(%rbp)
      jg      .L2
      movl    $1, %eax
      jmp     .L3
.L2:
      movl    -4(%rbp), %eax
      subl    $1, %eax
      movl    %eax, %edi
      call    fact_r
      imull   -4(%rbp), %eax
.L3:
      leave
      ret
```

fact_r(3):



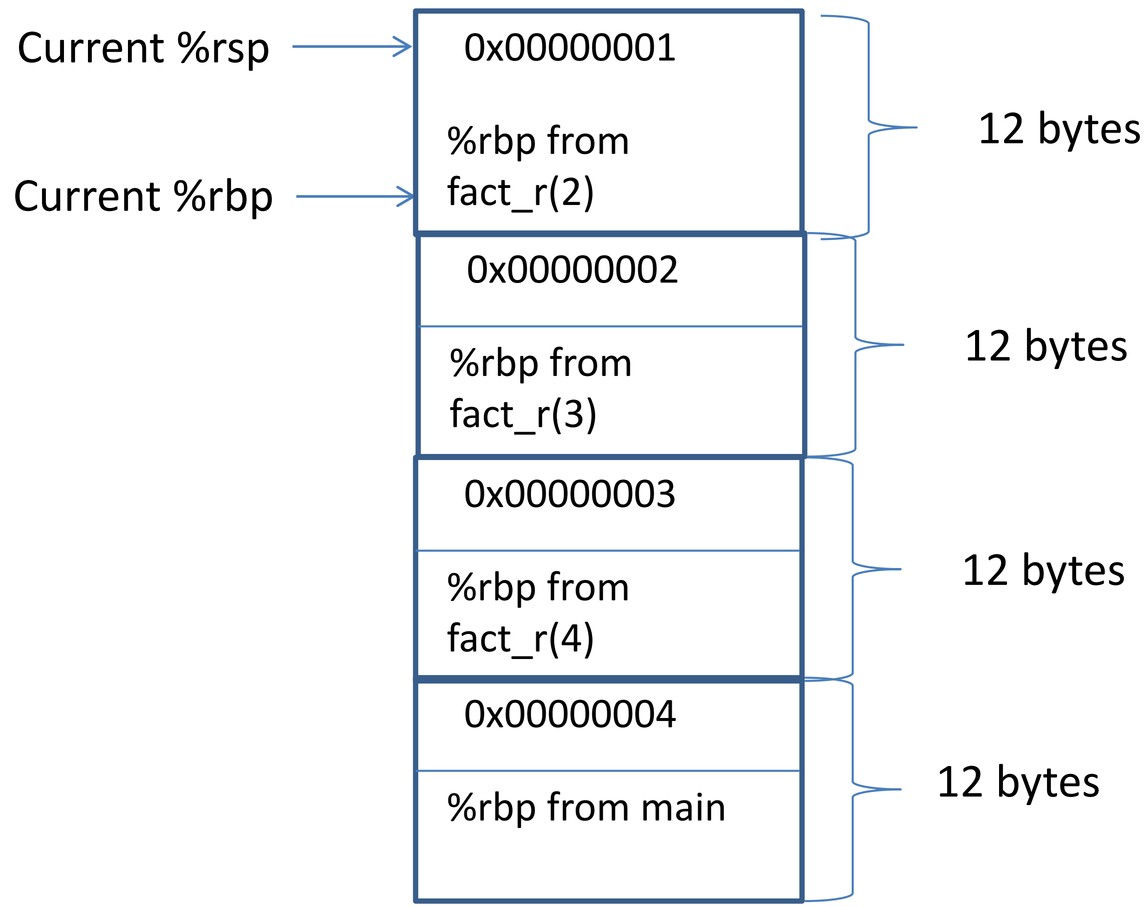
```
fact_r:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    cmpl    $1, -4(%rbp)
    jg      .L2
    movl    $1, %eax
    jmp     .L3
.L2:
    movl    -4(%rbp), %eax
    subl    $1, %eax
    movl    %eax, %edi
    call    fact_r
    imull   -4(%rbp), %eax
.L3:
    leave
    ret
```

fact_r(2):



```
fact_r:
      pushq   %rbp
      movq    %rsp, %rbp
      subq    $16, %rsp
      movl    %edi, -4(%rbp)
      cmpl    $1, -4(%rbp)
      jg      .L2
      movl    $1, %eax
      jmp     .L3
.L2:
      movl    -4(%rbp), %eax
      subl    $1, %eax
      movl    %eax, %edi
      call    fact_r
      imull   -4(%rbp), %eax
.L3:
      leave
      ret
```

fact_r(1):



```
int fact_r(int x) {
  if (x <= 1)
    return 1;
  else return x * fact_r(x-1);
}
```

Where is x remembered?

Now, using
gcc –o fact.s –s fact.c –O2

```
fact_r:
      cmpl    $1, %edi
      movl    $1, %eax
      jg      .L3
      jmp     .L2
.L9:
      movl    %edx, %edi
.L3:
      leal    -1(%rdi), %edx
      imull   %edi, %eax
      cmpl    $1, %edx
      jne     .L9
.L2:
      ret
```

This is no longer recursive!!

Compiler removes recursion whenever possible, because loops are faster

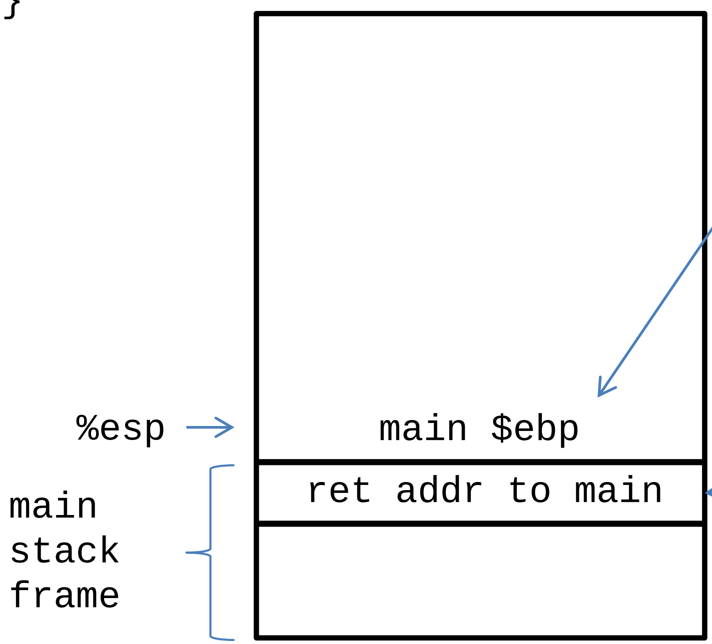
Function parameters and the program stack

- In x86-64, parameters are usually passed through registers
- In the unlikely event that more than 6 parameters are passed, then the program stack is used for the rest of the parameters
- To illustrate, we will switch to IA-32, and consider this code.

```
int g(int x, int y) {
    return x + y;
}

int f() {
   int a=1, b=2;
   return g(a,b);
}

int main() {
   printf(“%d\n”, f());
}
```

```c
int f() {
   int a=1, b=2;
   return g(a,b);
}

int main() {
   printf(“%d\n”, f());
}
```

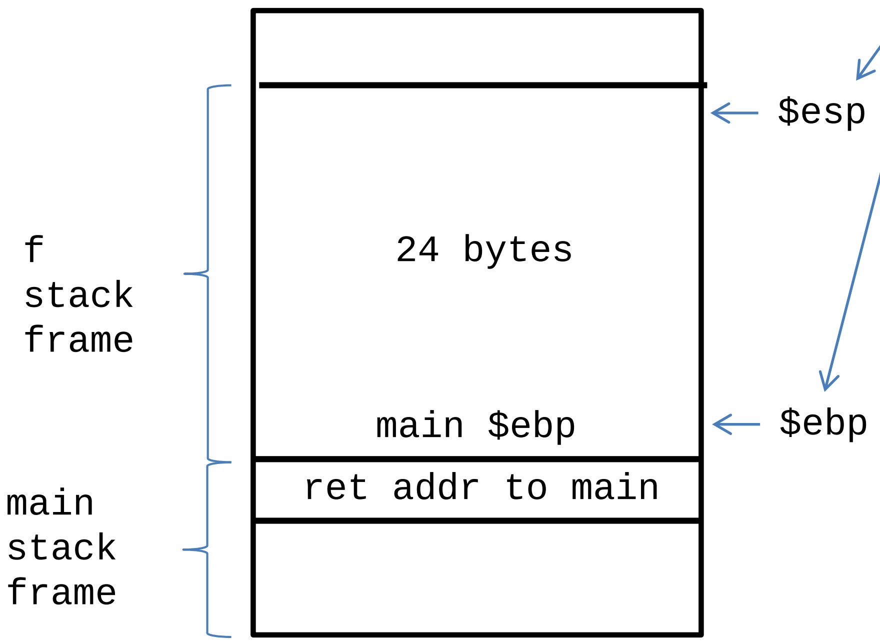
```
_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $1, -4(%ebp)
        movl    $2, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call    _g
        leave
        ret
_main:
        …
        call    _f
        …
```



Program stack

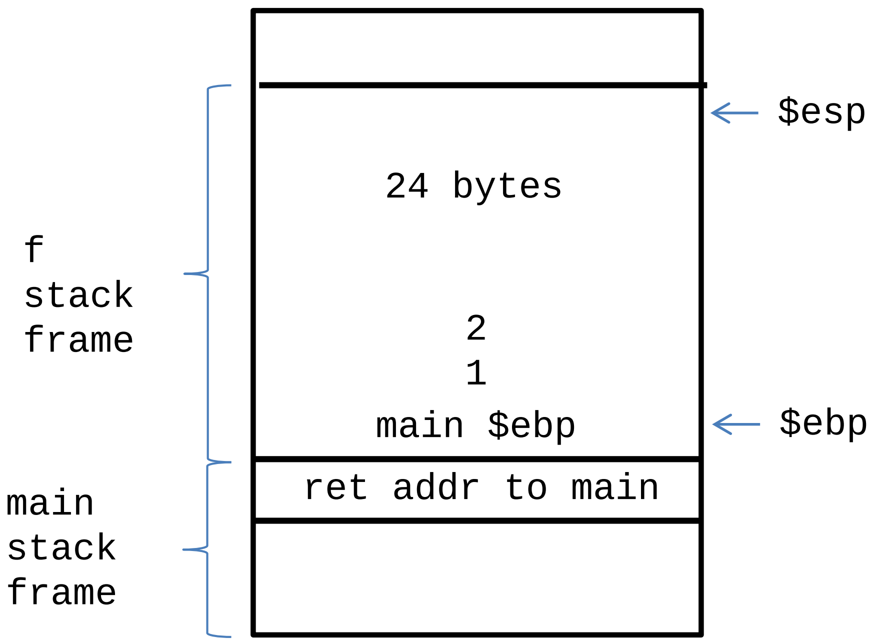
```
int f() {
   int a=1, b=2;
   return g(a,b);
}

int main() {
   printf("%d\n", f());
}
```

```
_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $1, -4(%ebp)
        movl    $2, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call    _g
        leave
        ret
```



```
int f() {
   int a=1, b=2;
   return g(a,b);
}

int main() {
   printf("%d\n", f());
}
```

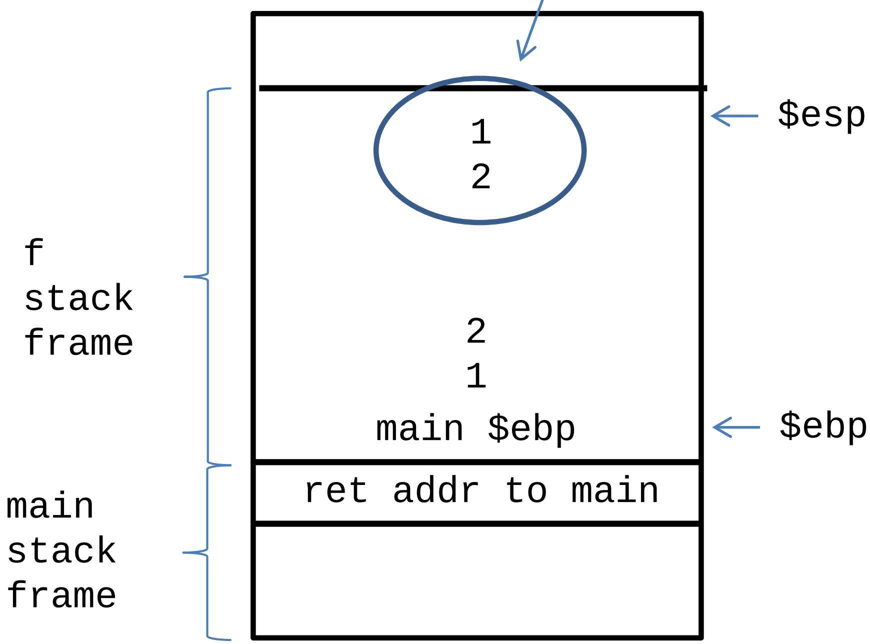
```
_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $1, -4(%ebp)
        movl    $2, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call    _g
        leave
        ret
```



```c
int f() {
   int a=1, b=2;
   return g(a,b);
}

int main() {
   printf("%d\n", f());
}
```

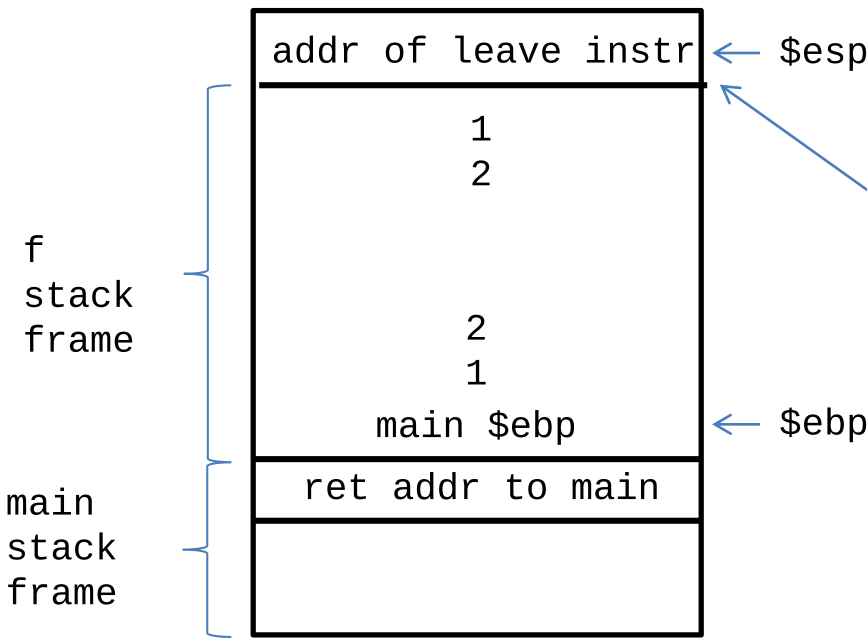
These are parameters that will be passed to g

```
_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $1, -4(%ebp)
        movl    $2, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call    _g
        leave
        ret
```



```
int g(int x, int y) {
   return x + y;
}

int f() {
   int a=1, b=2;
   return g(a,b);
}
```

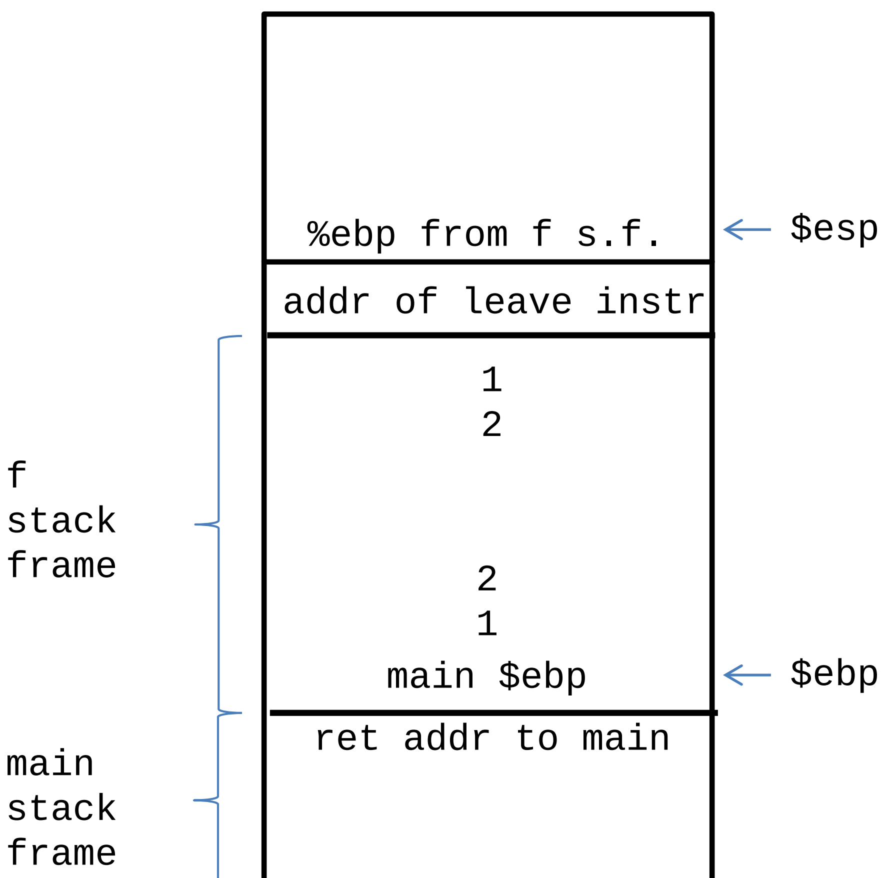
```
_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $1, -4(%ebp)
        movl    $2, -8(%ebp)
        movl    -8(%ebp), %eax # copy arg 2
        movl    %eax, 4(%esp)
        movl    -4(%ebp), %eax # copy arg 1
        movl    %eax, (%esp)
        call    _g
        leave
        ret
```



The **call** instruction pushes the return address onto the program stack, so that the calling function know where to return

```c
int g(int x, int y) {
   return x + y;
}

int f() {
   int a=1, b=2;
   return g(a,b);
}
```

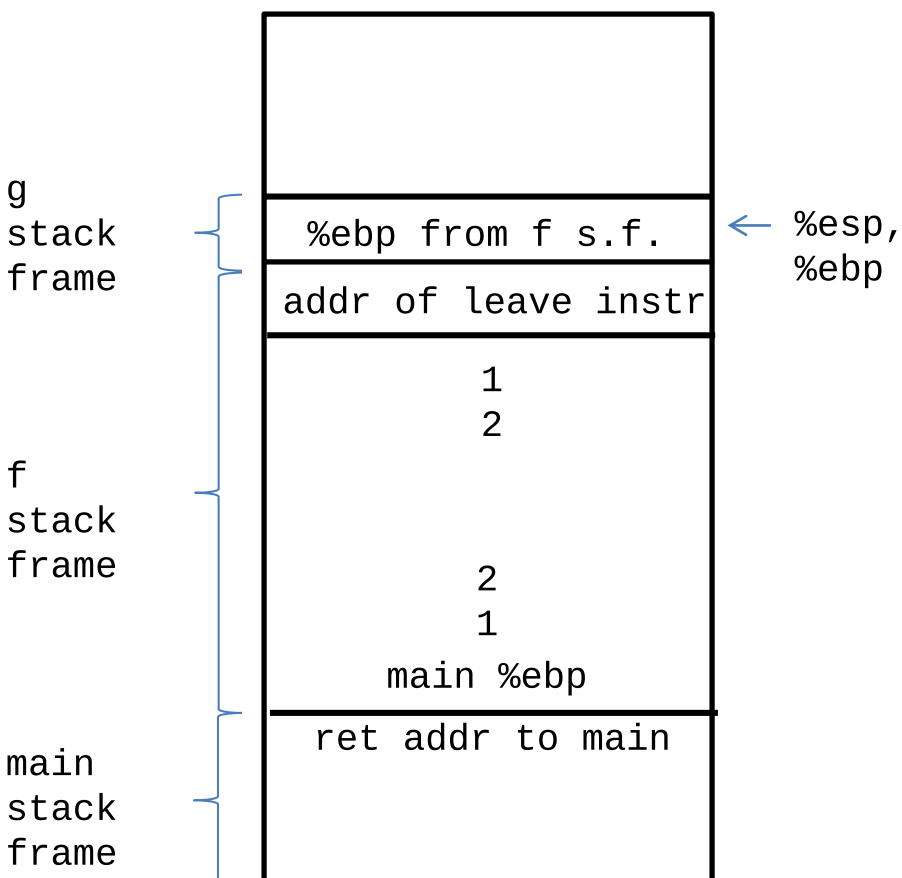
```
_g:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        popl    %ebp
        ret

_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $1, -4(%ebp)
        movl    $2, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call    _g
        leave
        ret
```



```c
int g(int x, int y) {
   return x + y;
}

int f() {
   int a=1, b=2;
   return g(a,b);
}
```

```
_g:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        popl    %ebp
        ret
```

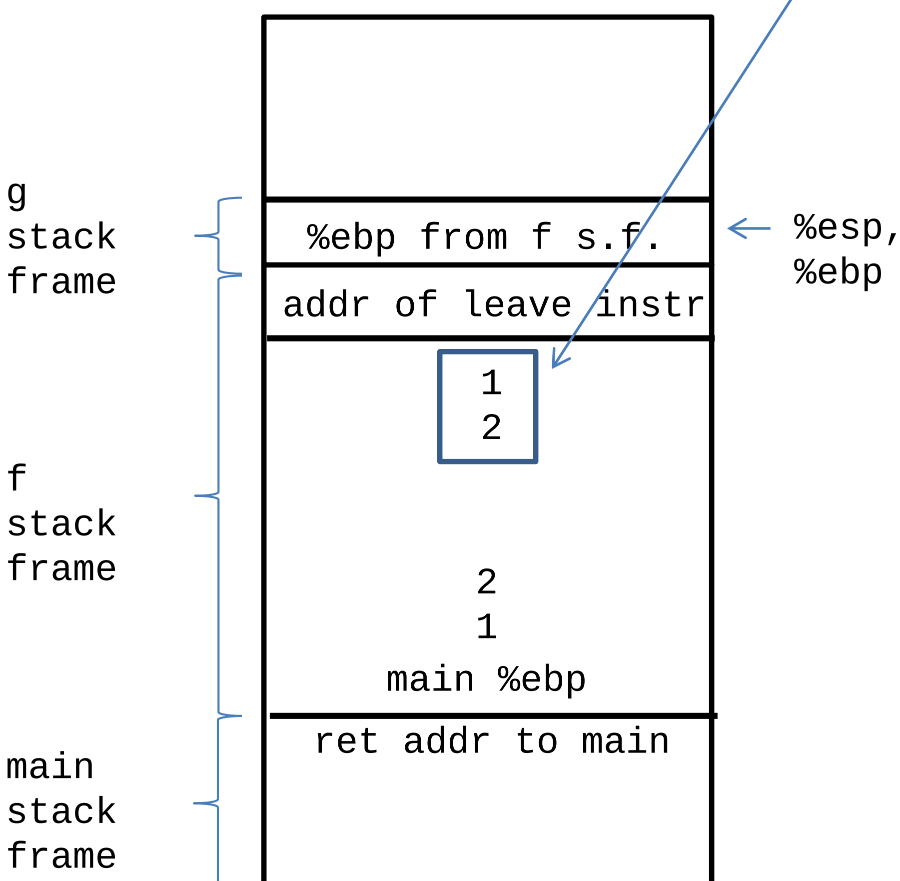
```
_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $1, -4(%ebp)
        movl    $2, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call    _g
        leave
        ret
```



```c
int g(int x, int y) {
   return x + y;
}

int f() {
   int a=1, b=2;
   return g(a,b);
}
```

-12(%ebp) and -8(%ebp)
Are the parameters
Passed to g

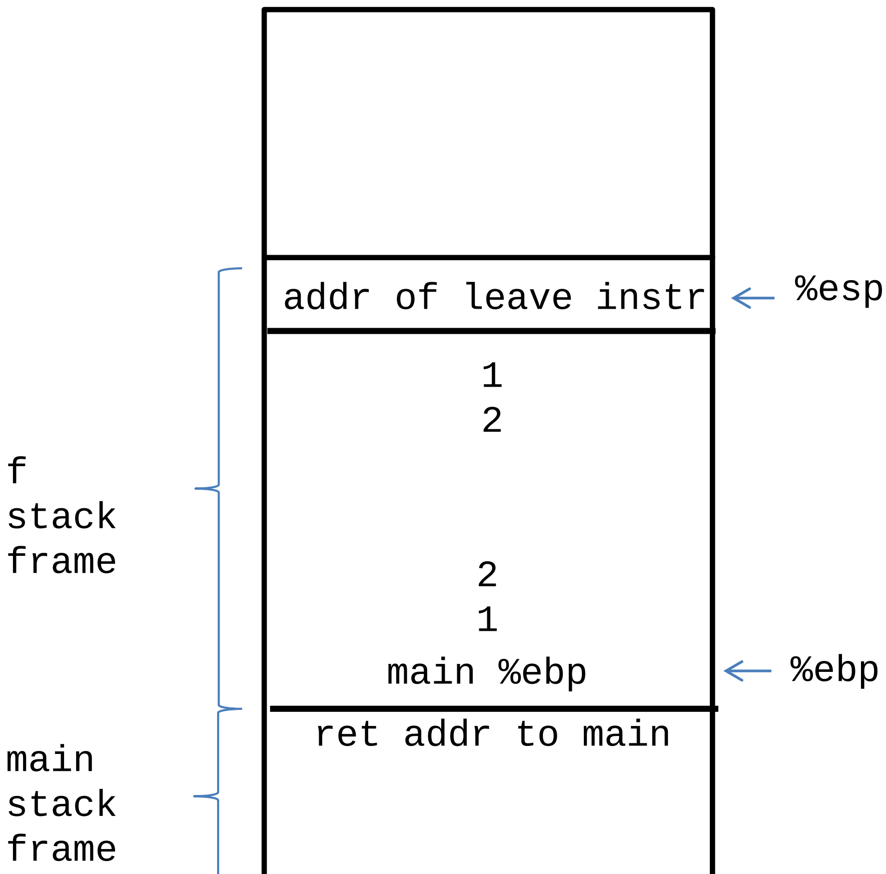
```
_g:
    pushl   %ebp
    movl    %esp, %ebp
    movl    -12(%ebp), %eax
    addl    -8(%ebp), %eax
    popl    %ebp
    ret

_f:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $24, %esp
    movl    $1, -4(%ebp)
    movl    $2, -8(%ebp)
    movl    -8(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    -4(%ebp), %eax
    movl    %eax, (%esp)
    call    _g
    leave
    ret
```



```c
int g(int x, int y) {
   return x + y;
}

int f() {
   int a=1, b=2;
   return g(a,b);
}
```

12(%ebp) and 8(%ebp)
Are the parameters
Passed to g

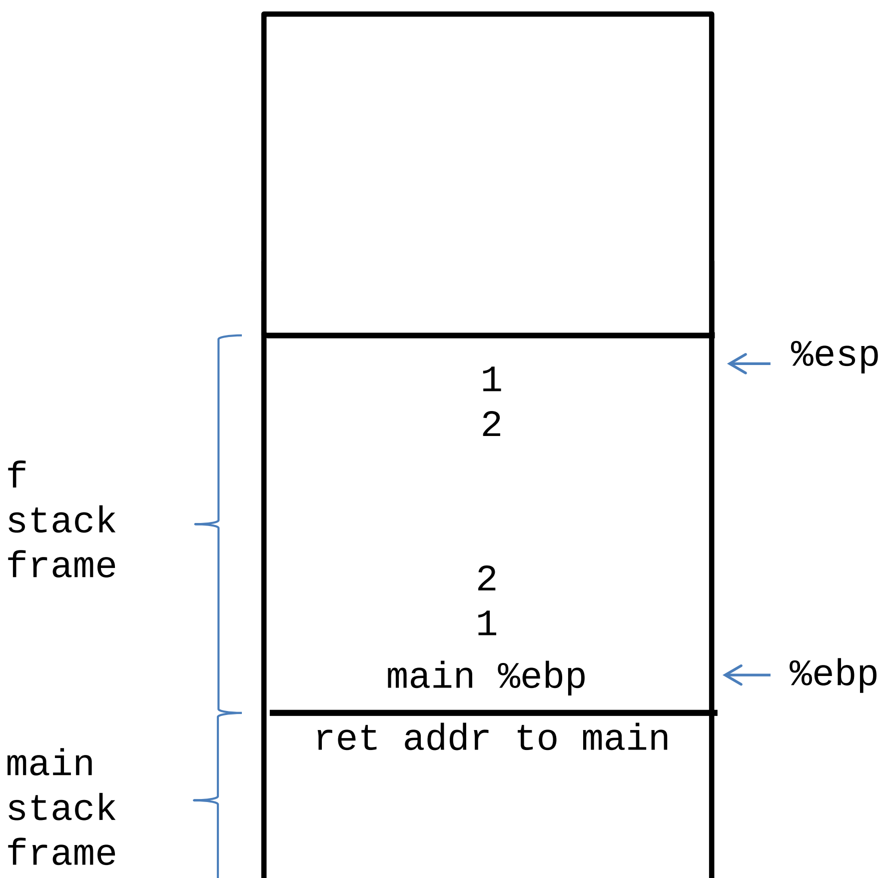
```
_g:
    pushl   %ebp
    movl    %esp, %ebp
    movl    12(%ebp), %eax
    addl    8(%ebp), %eax
    popl    %ebp
    ret

_f:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $24, %esp
    movl    $1, -4(%ebp)
    movl    $2, -8(%ebp)
    movl    -8(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    -4(%ebp), %eax
    movl    %eax, (%esp)
    call    _g
    leave
    ret
```



```
int g(int x, int y) {
   return x + y;
}

int f() {
   int a=1, b=2;
   return g(a,b);
}
```

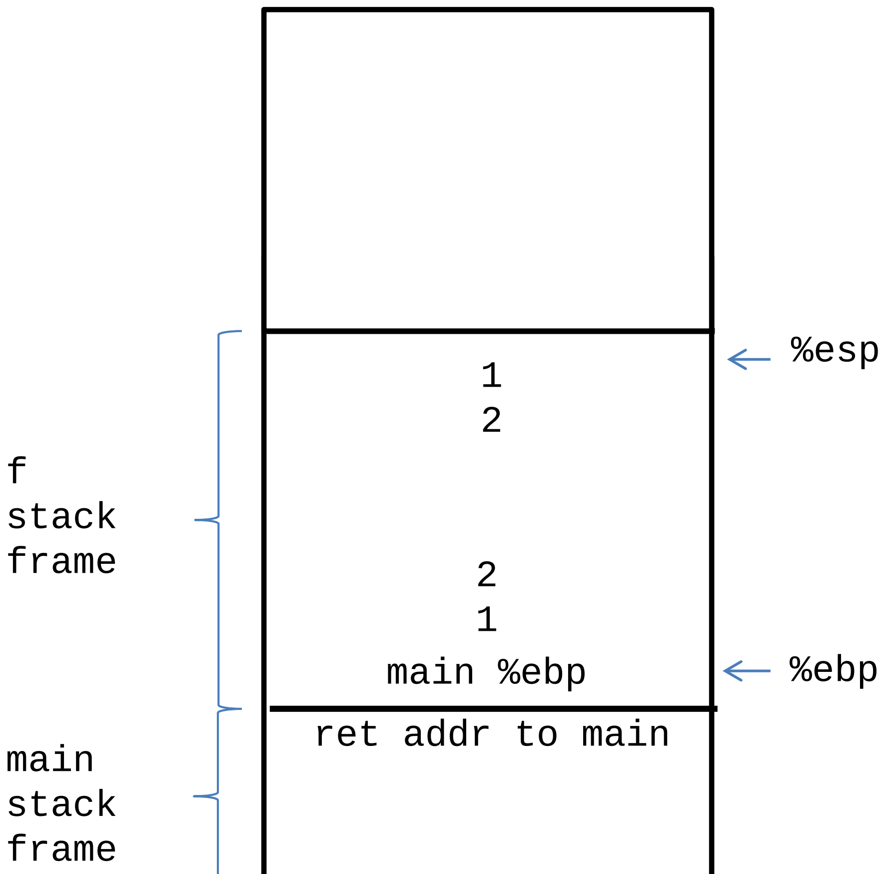
Next instruction is "leave" in f

```
_g:
    pushl   %ebp
    movl    %esp, %ebp
    movl    12(%ebp), %eax
    addl    8(%ebp), %eax
    popl    %ebp
    ret

_f:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $24, %esp
    movl    $1, -4(%ebp)
    movl    $2, -8(%ebp)
    movl    -8(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    -4(%ebp), %eax
    movl    %eax, (%esp)
    call    _g
    leave
    ret
```



```c
int g(int x, int y) {
   return x + y;
}

int f() {
   int a=1, b=2;
   return g(a,b);
}
```

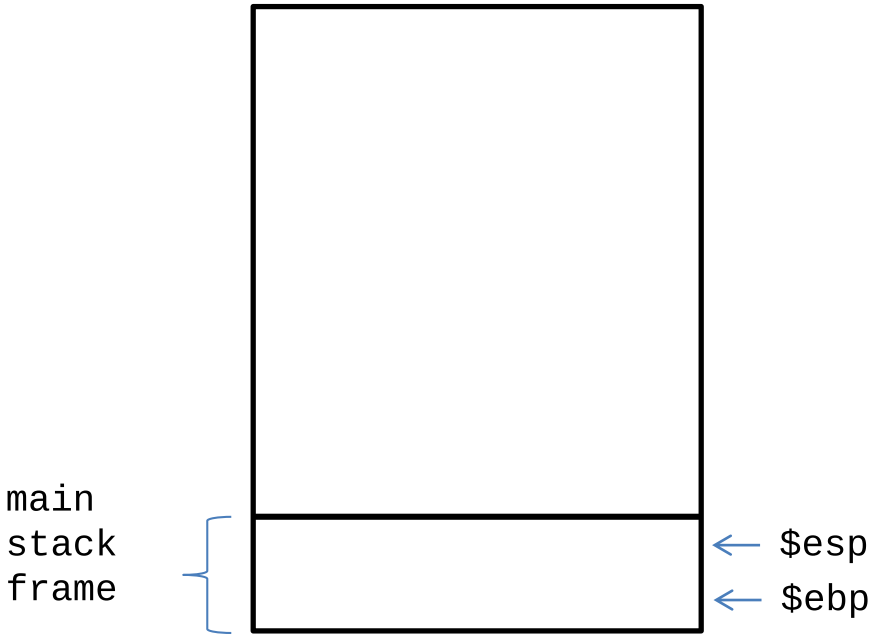
```
_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $1, -4(%ebp)
        movl    $2, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call    _g
        leave
        ret
```



leave is a macro for popping into %ebp. After **ret**, the next instruction is back in **main**

```
int g(int x, int y) {
   return x + y;
}

int f() {
   int a=1, b=2;
   return g(a,b);
}
```

```
_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $1, -4(%ebp)
        movl    $2, -8(%ebp)
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, (%esp)
        call    _g
        leave
        ret
```



Example illustrating program stack

```
int main( ) {
  int x, y, z;
  printf("Type 3 numbers\n");
  scanf("%d%d%d", &x, &y, &z);
  printf("%d+%d+%d=%d\n", x, y,
       z, sum3(x, y, z));
}

int sum3(int x, int y, int z) {
  return x + sum2(y,z);
}

int sum2(int x, int y) {
  int sum = x + y;
  return sum;
}
```

x86-64 Example

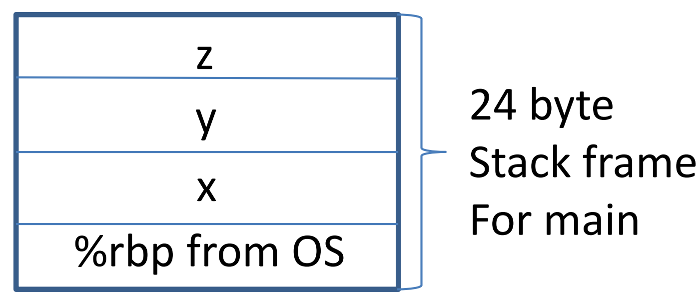
```
int main( ) {
  int x, y, z;
  printf("Type 3 numbers\n");
  scanf("%d%d%d", &x, &y, &z);
  printf("%d+%d+%d=%d\n", x, y,
        z, sum3(x, y, z));
}
```

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $.LC0, %edi
        call    puts
        leaq    -16(%rbp), %rcx
        leaq    -12(%rbp), %rdx
        leaq    -8(%rbp), %rsi
        movl    $.LC1, %edi
        call    __isoc99_scanf
        movl    -16(%rbp), %edx
        movl    -12(%rbp), %ecx
        movl    -8(%rbp), %eax
        movl    %ecx, %esi
        movl    %eax, %edi
        call    sum3
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC2, %edi
        call    printf
        leave
        ret
```

x86-64 Example

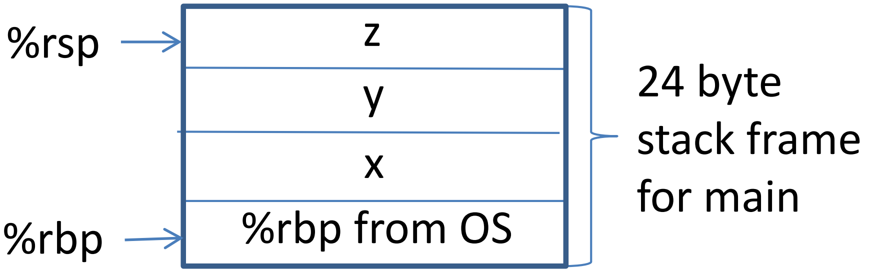
```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $.LC0, %edi
        call    puts
        leaq    -16(%rbp), %rcx
        leaq    -12(%rbp), %rdx
        leaq    -8(%rbp), %rsi
        movl    $.LC1, %edi
        call    __isoc99_scanf
        movl    -16(%rbp), %edx
        movl    -12(%rbp), %ecx
        movl    -8(%rbp), %eax
        movl    %ecx, %esi
        movl    %eax, %edi
        call    sum3
```

```
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC2, %edi
        call    printf
        leave
        ret
```



x86-64 Example

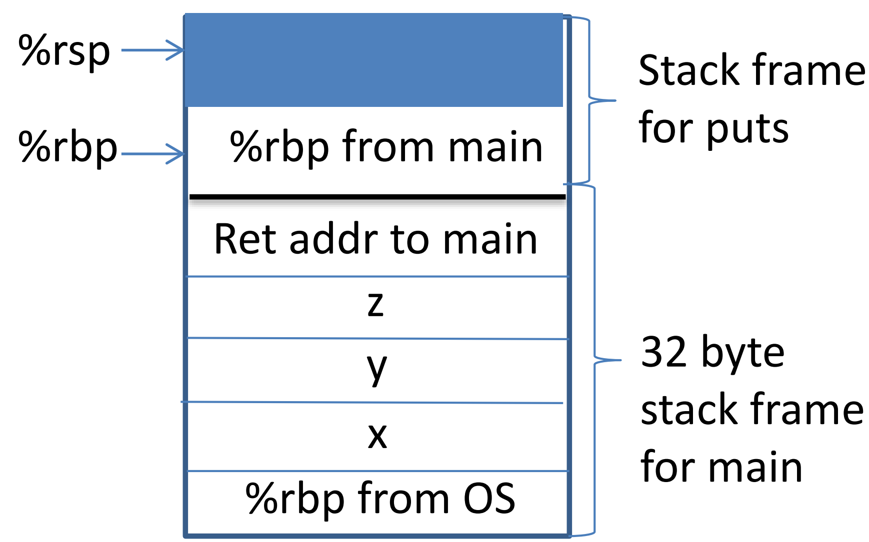
```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $.LC0, %edi
        call    puts
        leaq    -16(%rbp), %rcx
        leaq    -12(%rbp), %rdx
        leaq    -8(%rbp), %rsi
        movl    $.LC1, %edi
        call    __isoc99_scanf
        movl    -16(%rbp), %edx
        movl    -12(%rbp), %ecx
        movl    -8(%rbp), %eax
        movl    %ecx, %esi
        movl    %eax, %edi
        call    sum3
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC2, %edi
        call    printf
        leave
        ret
```



x86-64 Example

```
main:
            pushq   %rbp
            movq    %rsp, %rbp
            subq    $16, %rsp
Arg to      movl    $.LC0, %edi
puts        call    puts
            leaq    -16(%rbp), %rcx
            leaq    -12(%rbp), %rdx
            leaq    -8(%rbp), %rsi
            movl    $.LC1, %edi
            call    __isoc99_scanf
            movl    -16(%rbp), %edx
            movl    -12(%rbp), %ecx
            movl    -8(%rbp), %eax
            movl    %ecx, %esi
            movl    %eax, %edi
            call    sum3
```

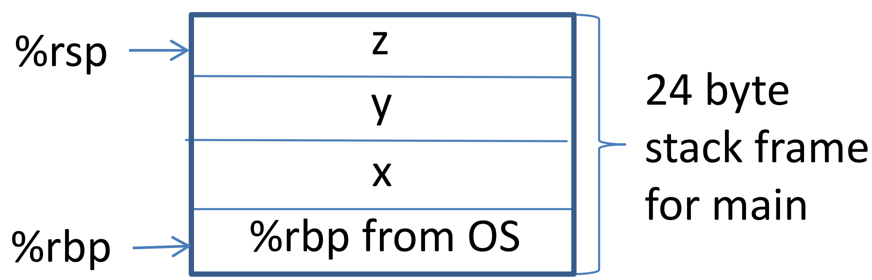
```
            movl    %eax, -4(%rbp)
            movl    -4(%rbp), %eax
            movl    %eax, %esi
            movl    $.LC2, %edi
            call    printf
            leave
            ret
```



x86-64 Example

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $.LC0, %edi
        call    puts    # prompt user
        leaq    -16(%rbp), %rcx # &z
        leaq    -12(%rbp), %rdx # &y
        leaq    -8(%rbp), %rsi  # &x
        movl    $.LC1, %edi
        call    __isoc99_scanf
        movl    -16(%rbp), %edx
        movl    -12(%rbp), %ecx
        movl    -8(%rbp), %eax
        movl    %ecx, %esi
        movl    %eax, %edi
        call    sum3
```

```
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC2, %edi
        call    printf
        leave
        ret
```

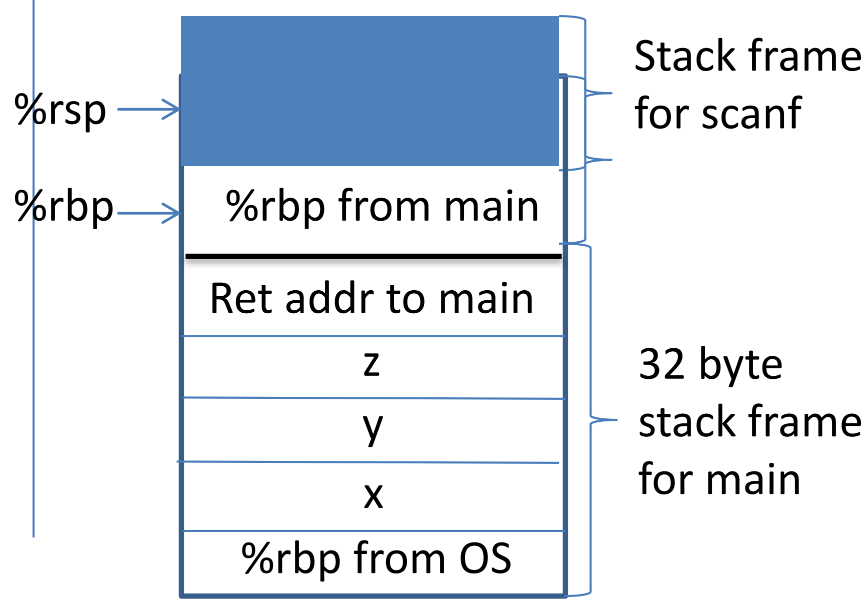


x86-64 Example

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    $.LC0, %edi
        call    puts   # prompt user
        leaq    -16(%rbp), %rcx
        leaq    -12(%rbp), %rdx
        leaq    -8(%rbp), %rsi
        movl    $.LC1, %edi
        call    __isoc99_scanf
        movl    -16(%rbp), %edx
        movl    -12(%rbp), %ecx
        movl    -8(%rbp), %eax
        movl    %ecx, %esi
        movl    %eax, %edi
        call    sum3
```

args to scanf

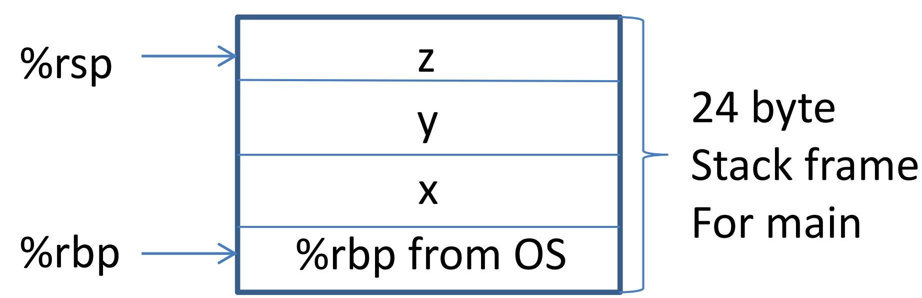
```
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC2, %edi
        call    printf
        leave
        ret
```



x86-64 Example

```
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp  #%rbp pushed
    movl    $.LC0, %edi
    call    puts   # prompt user
    leaq    -16(%rbp), %rcx
    leaq    -12(%rbp), %rdx
    leaq    -8(%rbp), %rsi
    movl    $.LC1, %edi
    call    __isoc99_scanf
    movl    -16(%rbp), %edx #z
    movl    -12(%rbp), %ecx
    movl    -8(%rbp), %eax
    movl    %ecx, %esi  #y
    movl    %eax, %edi  #z
    call    sum3
```

```
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    movl    $.LC2, %edi
    call    printf
    leave
    ret
```



x86-64 Example

```
movl    -16(%rbp), %edx #z
movl    -12(%rbp), %ecx
movl    -8(%rbp), %eax
movl    %ecx, %esi      #y
movl    %eax, %edi      #z
call    sum3
```

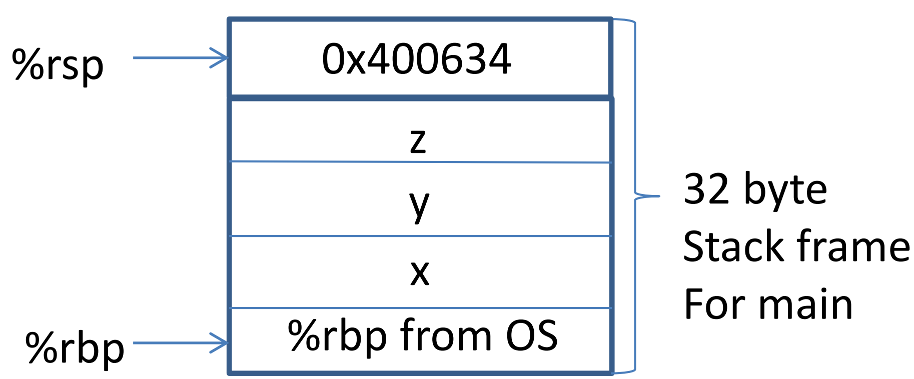
(gdb) x/x $rsp
0x7fffffffe978: 0x00400634

In main:
0x0040062f <+66>: callq 0x40064d <sum3>
0x00400634 <+71>: mov %eax,-0x4(%rbp)

When one function calls another,
The return address in the calling
function is pushed onto the program
stack

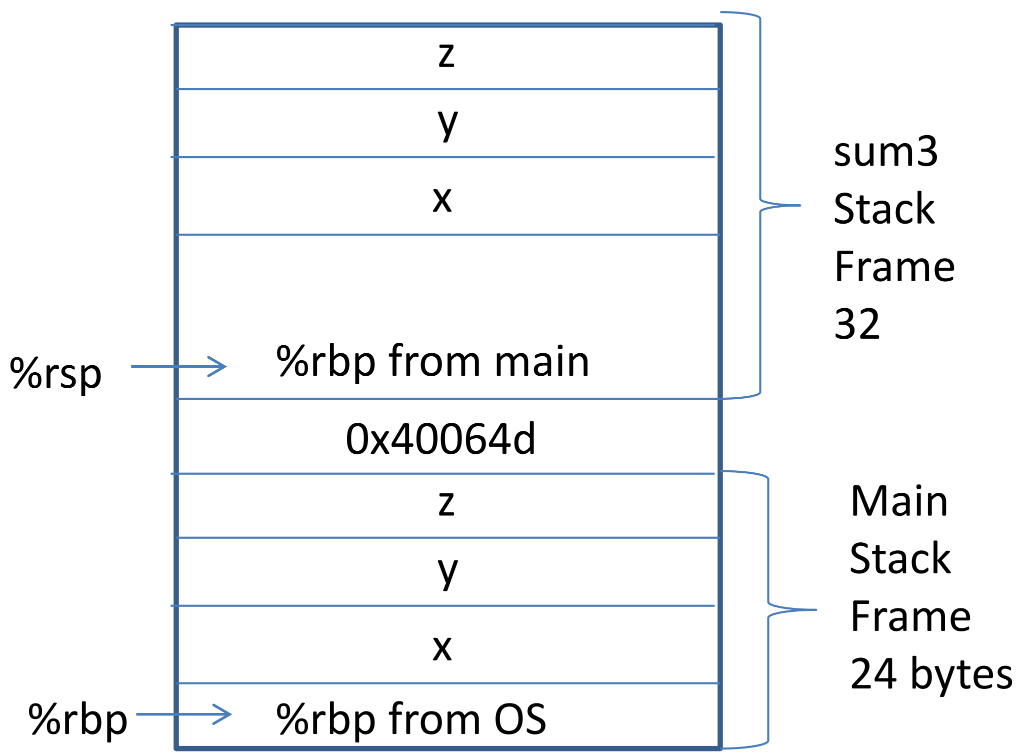
(gdb) break *sum3
Breakpoint 5 at 0x40064d
(gdb) cont
Continuing.

Breakpoint 5, 0x0000000000004006



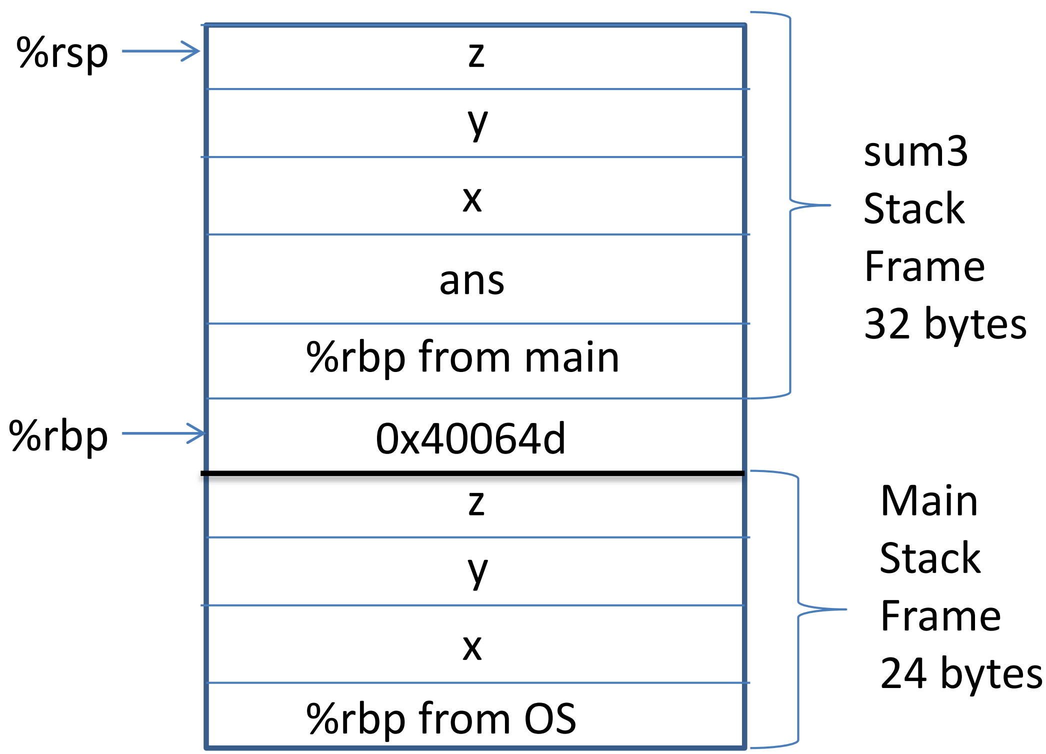
x86-64 Example

```
sum3:
     pushq   %rbp
      movq    %rsp, %rbp
     .cfi_def_cfa_register 6
     subq    $32, %rsp
     movl    %edi, -20(%rbp)
     movl    %esi, -24(%rbp)
     movl    %edx, -28(%rbp)
     movl    -28(%rbp), %edx
     movl    -24(%rbp), %eax
     movl    %edx, %esi
     movl    %eax, %edi
     call    sum2
      movl    %eax, -4(%rbp)
     movl    -4(%rbp), %eax
     movl    -20(%rbp), %edx
     addl    %edx, %eax
     leave
     ret
```



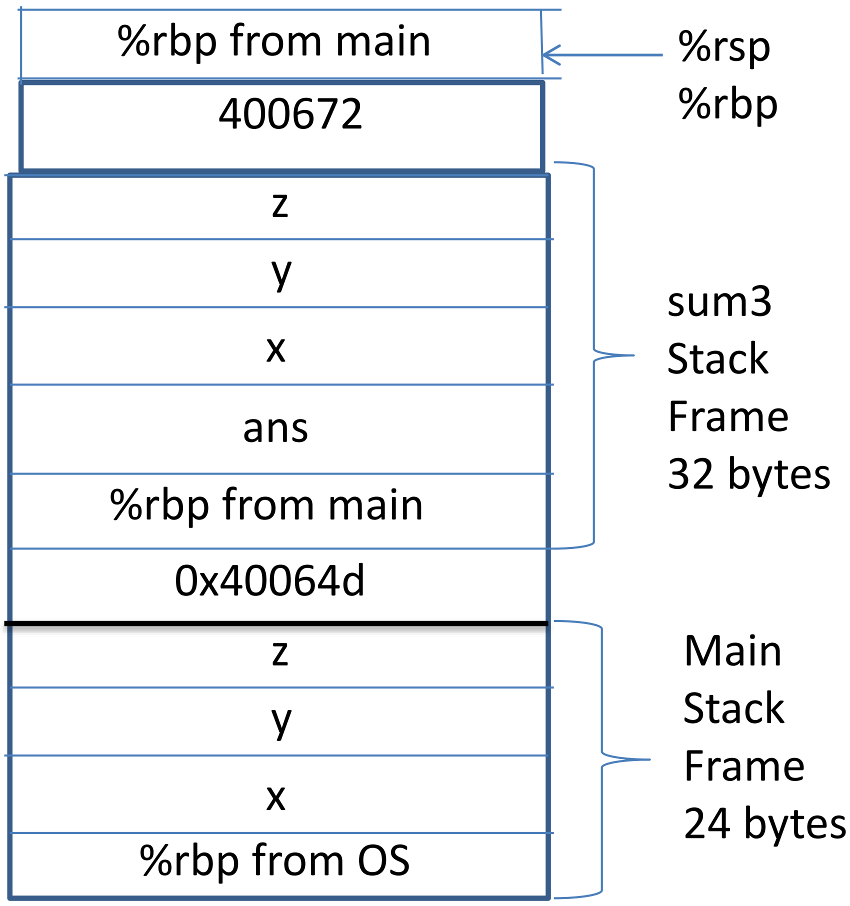
x86-64 Example

```
sum3:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $32, %rsp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    %edx, -28(%rbp)
    movl    -28(%rbp), %edx
    movl    -24(%rbp), %eax
    movl    %edx, %esi
    movl    %eax, %edi
    call    sum2
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    -20(%rbp), %edx
    addl    %edx, %eax
    leave
     ret
```



x86-64 Example

```
sum2:
   0x40067f <+0>:    push    %rbp
   0x400680 <+1>:    mov     %rsp,%rbp
   0x400683 <+4>:    mov     %edi,-0x4(%rbp)
   0x400686 <+7>:    mov     %esi,-0x8(%rbp)
   0x400689 <+10>:   mov     -0x8(%rbp),%eax
   0x40068c <+13>:   mov     -0x4(%rbp),%edx
   0x40068f <+16>:   add     %edx,%eax
   0x400691 <+18>:   pop     %rbp
   0x400692 <+19>:   retq
```



The GNU debugger

- Very low-level debugging / code inspection tool
- Operates on executable files
- Enables programmers to “disassemble” code, set breakpoints, inspect register and memory contents, step through code execution, etc.
- We will use it for code inspection, and to help us understand what machine code is doing and how
- We will also use it to understand a certain type of security vulnerability caused by the stack frame organization

Example gdb session

```
> gdb sum3
> disas sum3
(gdb) disas sum3
Dump of assembler code for function sum3:
   0x0000000000400608 <+0>:     push   %rbp
   0x0000000000400609 <+1>:     mov    %rsp,%rbp
   0x000000000040060c <+4>:     sub    $0x20,%rsp
   0x0000000000400610 <+8>:     mov    %rdi,-0x18(%rbp)
   0x0000000000400614 <+12>:    mov    -0x18(%rbp),%rax
   0x0000000000400618 <+16>:    mov    (%rax),%eax
   0x000000000040061a <+18>:    mov    %eax,-0x4(%rbp)
   0x000000000040061d <+21>:    mov    -0x18(%rbp),%rax
   0x0000000000400621 <+25>:    add    $0x4,%rax
   0x0000000000400625 <+29>:    mov    %rax,%rdi
   0x0000000000400628 <+32>:    mov    $0x0,%eax
   0x000000000040062d <+37>:    callq  0x40063a <sum2>
   0x0000000000400632 <+42>:    add    %eax,-0x4(%rbp)
   0x0000000000400635 <+45>:    mov    -0x4(%rbp),%eax
   0x0000000000400638 <+48>:    leaveq
   0x0000000000400639 <+49>:    retq
```

Example gdb session

```
(gdb) run
Starting program: /home/DPU/slytinen/406w18/x86/sum3_slow
Type 3 integers
1 2 3

Breakpoint 1, 0x0000000000400608 in sum3 ()
Missing separate debuginfos, use: debuginfo-install glibc-2.17-106.el7_2.6.x86_64
(gdb) print/x $rdi
$1 = 0x7fffffffe960
(gdb) x/3x $rdi
0x7fffffffe960: 0x00000001      0x00000002      0x00000003
(gdb) print/x $rsp
$2 = 0x7fffffffe958
(gdb) x/x $rsp
0x7fffffffe958: 0x004005ef
(gdb) x/x 0x4005ef
0x4005ef <main+79>:     0x8bf84589
(gdb) x/i main+79
 0x4005ef <main+79>:  mov    %eax,-0x8(%rbp)
(gdb) x/i main+74
   0x4005ea <main+74>:  callq  0x400608 <sum3>
```

Example gdb session

```
(gdb) break *sum3+48
Breakpoint 2 at 0x400638
(gdb) continue
Continuing.

Breakpoint 2, 0x0000000000400638 in sum3 ()
(gdb) print/d $rax
$3 = 6
(gdb) stepi
0x0000000000400639 in sum3 ()
(gdb) stepi
0x00000000004005ef in main ()
(gdb) disas 0x4005f2
   0x00000000004005ea <+74>:    callq  0x400608 <sum3>
   0x00000000004005ef <+79>:    mov    %eax,-0x8(%rbp)
=> 0x00000000004005f2 <+82>:    mov    -0x8(%rbp),%eax
   0x00000000004005f5 <+85>:    mov    %eax,%esi
   0x00000000004005f7 <+87>:    mov    $0x400713,%edi
   0x00000000004005fc <+92>:    mov    $0x0,%eax
   0x0000000000400601 <+97>:    callq  0x400470 <printf@plt>
   0x0000000000400606 <+102>:   leaveq
   0x0000000000400607 <+103>:   retq
```

Example gdb session

```
(gdb) break *main+97
Breakpoint 3 at 0x400601
(gdb) continue
Continuing.

Breakpoint 3, 0x0000000000400601 in main ()
(gdb) print/x $rdi
$4 = 0x400713
(gdb) x/x $rdi
0x400713:       0x20656854
(gdb) x/s $rdi
0x400713:       "The sum is %d\n“
(gdb) print/d $rsi
$5 = 6
(gdb) cont
Continuing.
The sum is 6
[Inferior 1 (process 27473) exited with code 015]
```

Basic gdb commands

- Start the debugger: **gdb** followed by the name of the executable
- **disas**: "disassemble" a function (translate from machine code back to assembly language). Can either be followed by a function name or an address
- **break**: set a breakpoint. Can either be followed by an address or an offset from the beginning of a function. Place an * beforehand, such as break *main
- **run**
- **cont:** resume execution from a breakpoint
- **print**: print the contents of a register. Qualifiers: /d, /x, /s, /c
  – Example: **print/d $rax**
- **x:** print the contents of a memory address. Also may use qualifiers
  – Example: x/x 0x400601
  – x/3x $rdi
  – x/s $rdi
- **stepi**: execute the next instruction (step into). If the current instruction is a function call, then the first instruction of the called function is executed