x86-64 assembly

x86 history

seven 8-bit registers

- 1971: Intel 8008

eight 16-bit registers:

- 1978: Intel 8086
- 1982: Intel 80286

eight 32-bit registers:

- 1985: Intel 80386
- 1989: Intel 80486
- 1993: Intel Pentium
- 1997: Intel Pentium II
- 1998: Intel Pentium III
- 2000: Intel Pentium IV/Xeon

sixteen 64-bit registers:

- 2003: AMD64 Opteron
- 2004: Intel Pentium IV/Xeon
- (and most more recent AMD/Intel/Via chips)

two syntaxes

there are two ways of writing x86 assembly
- AT&T syntax (default on Linux, OS X)
- Intel syntax (default on Windows)

different operand order, way of writing addresses, punctuation, etc.

we mostly show Intel syntax

different directives

non-instruction parts of assembly are called *directives*

IBCM example: `one dw 1`

- there is no IBCM instruction called "dw"

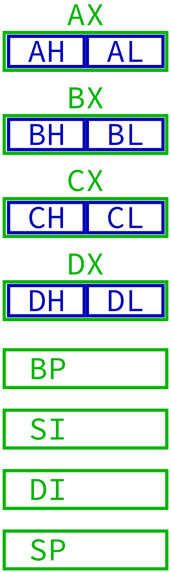
these differ *a lot* between assemblers

our main assember: `NASM`

our compiler's assembler: `GAS`

x86 registers

1978 – Intel 8086 — 8 16-bit registers



← AX, etc. — "general purpose"

(but some instructions use AX or BX only)

← "base pointer"

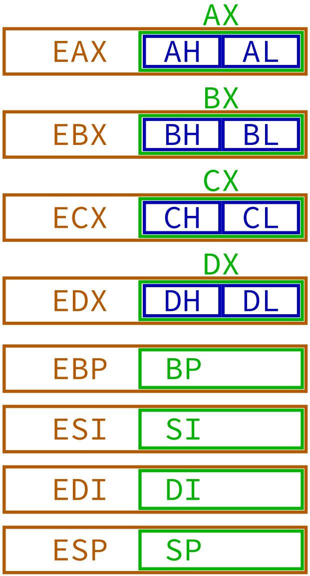
← "source index"

← "destination index"

special for *some* instrs.

← "stack pointer" — push/pop instrs.

x86 registers

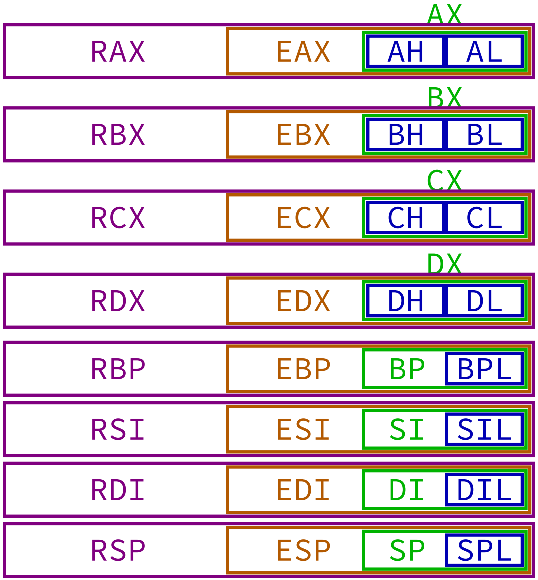


1988 – Intel 386 — 8 32-bit registers

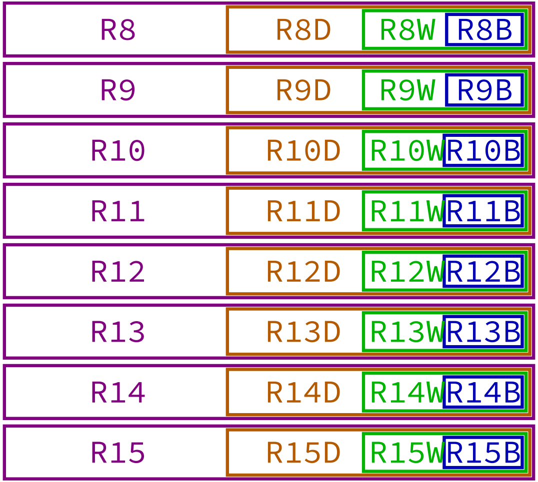
"**E**xtended" versions of each register

x86 registers

2003 – AMD64 — 16 64-bit registers



new registers just numbered
name for bottom byte of each register



some registers not shown

floating point/“vector” registers (`ST(0)`, `XMM0`, `YMM0`, `ZMM0`, …)

the program counter (`RIP`/`EIP`/`IP` — “instruction pointer”)

“flags” (used by conditional jumps)

registers for the operating system

…

x86 fetch/execute cycle

```
while (true) {
    IR <- memory[PC]
    execute(IR)
    if (instruction didn't change PC)
        PC <- PC + length-of-instruction(IR)
}
```

same as IBCM

(execpt instructions are variable-length)

declaring variables/constants

(*NASM*-only syntax)

`section .data`	".data" — data (not code) part of memory
`a DB 23`	DB: declare byte
`b DW ?`	DW: word (2 byte)
`c DD 3000`	DD: doubleword (4 bytes)
`d DQ -800`	DQ: quadword (8 byte)
`x DD 1, 2, 3`	? — don't care about value
`y TIMES 8 DB 0`	eight 0 bytes (e.g. 8-byte array)

a note on labels

NASM allows labels like:

```
LABEL add RAX, RBX
```

or like:

```
LABEL: add RAX, RBX
```

other assemblers: require `:` always

I recommend `:`

  what if label name = instruction name?

declaring variables/constants (GAS)

(*GAS*-only syntax)

```
.data
a:        .byte     23
b:        .short    0
c:        .long     3000
d:        .quad    −800
x:        .long    1, 2, 3
y         .fill 8, 1, 0
```

".data" — data (not code) part of memory

short — 2 bytes

long — 4 bytes

quad — 8 bytes

eight 0 bytes (e.g. 8-byte array)
(1 is length of value to repeat)

mov

`mov DEST, SRC`

possible DEST and SRC:

- register: `RAX`, `EAX`, ...
- constant: `0x1234`, `42`, ...
- label name: `someLabel`, ...
- memory address: `[0x1234]`, `[RAX]`, `[someLabel]`...

special rule: no moving from memory to memory

instruction operands generally

if we don't specify otherwise...

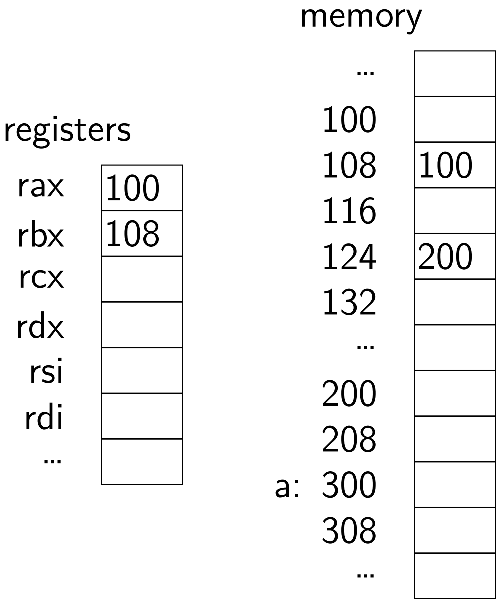
same as mov:

- destination: register or memory location
- source: register or constant or memory location

and same special rule: both can't be memory location

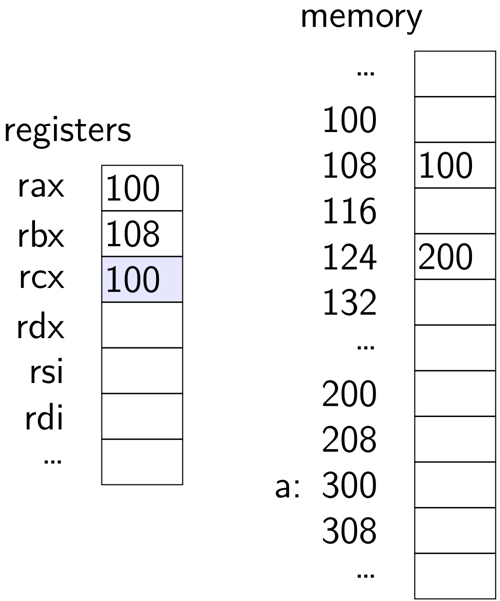
mov example

```
mov rcx, rax
mov rdx, [rbx]
mov rsi, [rdx+24]
mov [rsi], 45
mov [a], 15
```



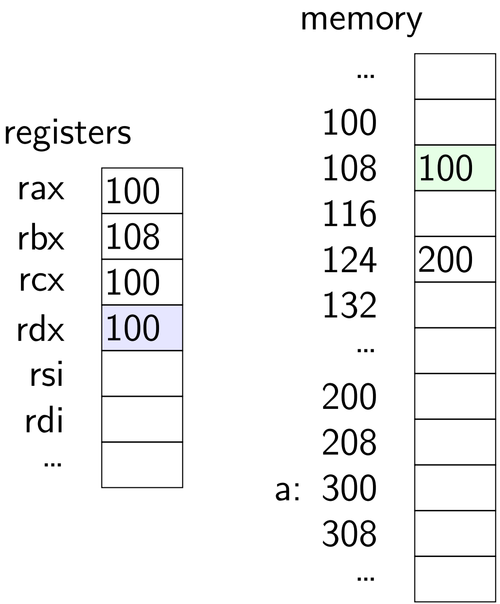
mov example

```
mov rcx, rax
mov rdx, [rbx]
mov rsi, [rdx+24]
mov [rsi], 45
mov [a], 15
```



mov example

```
mov rcx, rax
mov rdx, [rbx]
mov rsi, [rdx+24]
mov [rsi], 45
mov [a], 15
```



mov example

```
mov rcx, rax
mov rdx, [rbx]
mov rsi, [rdx+24]
mov [rsi], 45
mov [a], 15
```

registers

rax	100
rbx	108
rcx	100
rdx	100
rsi	200
rdi	
…	

memory

…	
100	
108	100
116	
124	200
132	
…	
200	
208	
a: 300	
308	
…	

mov example

```
mov rcx, rax
mov rdx, [rbx]
mov rsi, [rdx+24]
mov [rsi], 45
mov [a], 15
```

registers

rax	100
rbx	108
rcx	100
rdx	100
rsi	200
rdi	
…	

memory

…	
100	
108	100
116	
124	200
132	
…	
200	45
208	
a: 300	
308	
…	

mov example

```
mov rcx, rax
mov rdx, [rbx]
mov rsi, [rdx+24]
mov [rsi], 45
mov [a], 15
```

registers

rax	100
rbx	108
rcx	100
rdx	100
rsi	200
rdi	
...	

memory

...	
100	
108	100
116	
124	200
132	
...	
200	45
208	
a: 300	15
308	
...	

later: what types of addresses?

`[rdx]` allowed

`[someLabel]` allowed

`[rdx+24]` allowed

what else?

not everything — has to be encoded in machine code

explain rules: later

push/pop

RSP — "top" of stack which grows down

push RBX

$\text{RSP} \leftarrow \text{RSP} - 8$

$\text{memory[RSP]} \leftarrow \text{RBX}$

pop RBX

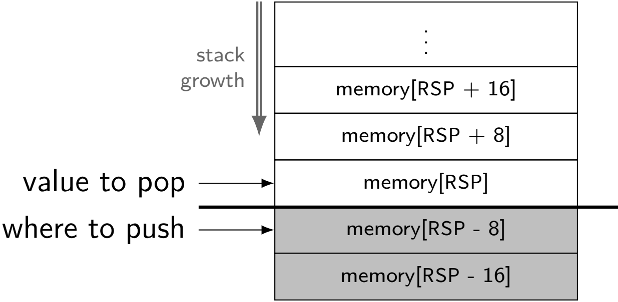
$\text{RBX} \leftarrow \text{memory[RSP]}$

$\text{RSP} \leftarrow \text{RSP} + 8$

also okay:

push [RAX], etc.

push 42, etc.



push/pop replacement

instead of:

```
push RAX
```

could write:

```
sub RSP, 8
mov [RSP], RAX
```

push/pop instructions are for convenience

add/sub

```
add first, second
add RAX, RBX
add [RDX], RBX
...
sub first, second
sub RSP, 16
...
```

first $\leftarrow$ first + second (`add`), or first $\leftarrow$ first − second (`sub`)

support same operands as mov:

- can use registers, constants, locations in memory
- can't use two memory locations (mov to a register instead)
- destination can't be constant

jmp

```
    jmp foo

foo: ...
```

jmp — go to instruction at label

conditon testing

`cmp <first>, <second>`

compare first and second

(compute first - second, compare to `0`)

set *flags* AKA *machine status word* based on result

`je label`

if (compare result was equal) go to label

conditional jmp example

```
if (RAX > 4)
    stuff();
```

```
            cmp RAX, 4
            jle skip_call
            call stuff
skip_call:  ...
```

jump conditions and cmp

```
cmp A, B
jXX label
```

$R = A - B$

`je`	equal	$R = 0$ or $A = B$
`jz`	zero	$R = 0$ or $A = B$
`jne`	not equal	$R \neq 0$ or $A \neq B$
`jl`	less than	$A < B$ (signed)
`jle`	less than or equal	$A \leq B$ (signed)
`jg`	greater than	$A > B$ (signed)
`jb`	less than (unsigned)	$A < B$ (unsigned)
`ja`	greater than (unsigned)	$A > B$ (unsigned)
`js`	sign bit set	$R < 0$
`jns`	sign bit unset	$R \geq 0$
…	…	…

C to assembly example

```c
int n = 5;
int i = 1;
int sum = 0;
...
  while (i <= n) {
    sum += i;
    i++;
  }
```

```nasm
section .data
n       DQ  5
i       DQ  1
sum     DQ  0
section .text
...
loop:   mov RCX, [i]
        cmp RCX, [n]
        jg endOfLoop
        add [sum], RCX
        add QWORD PTR [i], 1
        jmp loop
endOfLoop:
```

C to assembly example

```c
int n = 5;
int i = 1;
int sum = 0;
...
   while (i <= n) {
      sum += i;
      i++;
   }
```

```asm
section .data
n       DQ  5
i       DQ  1
sum     DQ  0
section .text
...
loop:   mov RCX, [i]
        cmp RCX, [n]
        jg endOfLoop
        add [sum], RCX
        add QWORD PTR [i], 1
        jmp loop
endOfLoop:
```

C to assembly example

```
int n = 5;
int i = 1;
int sum = 0;
...
  while (i <= n) {
    sum += i;
    i++;
  }
```

```
section .data
n       DQ  5
i       DQ  1
sum     DQ  0
section .text
...
loop:   mov RCX, [i]
        cmp RCX, [n]
        jg endOfLoop
        add [sum], RCX
...
endOfLoop:
```

`cmp [i], [n]` is not allowed
only one memory operand per (most) instructions

C to assembly example

```c
int n = 5;
int i = 1;
int sum = 0;
...
  while (i <= n) {
    sum += i;
    i++;
  }
```

```nasm
section .data
n       DQ  5
i       DQ  1
sum     DQ  0
section .text
...
loop:   mov RCX, [i]
        cmp RCX, [n]
        jg endOfLoop
        add [sum], RCX
        add QWORD PTR [i], 1
        jmp loop
endOfLoop:
```

C to assembly example

```
int n = 5;
int i = 1;
int sum = 0;
...
  while (i <= n) {
    sum += i;
    i++;
  }
```

```
section .data
n       DQ  5
i       DQ  1
sum     DQ  0
section .text
...
loop:   mov RCX, [i]
        cmp RCX, [n]
        jg endOfLoop
        add [sum], RCX
        add QWORD PTR [i], 1
        jmp loop
```

`QWORD PTR[i]` 8 bytes at location `i`
otherwise, no way to know how big otherwise
(more on this later)

C to assembly example

```c
int n = 5;
int i = 1;
int sum = 0;
...
  while (i <= n) {
    sum += i;
    i++;
  }
```

```nasm
section .data
n       DQ  5
i       DQ  1
sum     DQ  0
section .text
...
loop:   mov RCX, [i]
        cmp RCX, [n]
        jg endOfLoop
        add [sum], RCX
        add QWORD PTR [i], 1
        jmp loop
endOfLoop:
```

call

```
        call LABEL
        ...
```

is about the same as:

```
        push after_this_call
        jmp LABEL
after_this_call:
        ...
```

pushed address called the “return address”

call/ret

`call LABEL`

push next instruction address ("return address") to stack
jump to LABEL

`ret` — opposite of call

pop address from the stack
jump to that address

return addresses using a stack

```
max:     ...
         ...
         ret

main:    ...
         ...
         call max
after:   ...
         ret
```

return addresses using a stack

```
max:    ...
        ...
        ret

main:   ...
        ...
        call max
after:  ...
        ret
```

stack when `main` starts:

return address for `main` (OS) ← RSP

↓

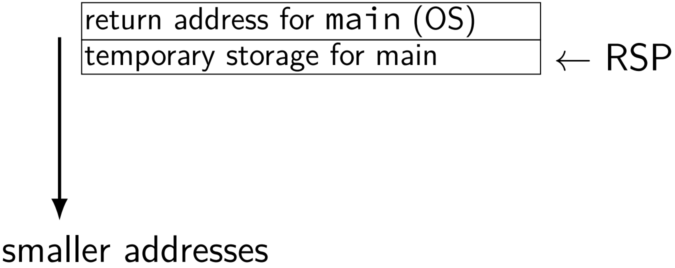
smaller addresses

return addresses using a stack

```
max:      ...
          ...
          ret

main:     ...
          ...
          call max
after:    ...
          ret
```

stack in the middle of `main`:



return addresses using a stack

```
max:    ...
        ...
        ret

main:   ...
        ...
        call max
after:  ...
        ret
```

stack just before `call max`:

return address for `main` (OS)	
temporary storage for main	
other things related to call???	← RSP

smaller addresses

return addresses using a stack

```
max:    ...
        ...
        ret

main:   ...
        ...
        call max
after:  ...
        ret
```

stack just after `call max`:

return address for `main` (OS)
temporary storage for main
other things related to call???
return address for `max` (`after:`) ← RSP

smaller addresses

return addresses using a stack

```
max:      ...
          ...
          ret

main:     ...
          ...
          call max
after:    ...
          ret
```

stack in the middle of `max`:

return address for `main` (OS)
temporary storage for main
other things related to call???
return address for `max` (`after:`)
temporary storage for max ← RSP

↓ smaller addresses

return addresses using a stack

```
max:     ...
         ...
         ret

main:    ...
         ...
         call max
after:   ...
         ret
```

stack just before `max`'s `ret`:

return address for `main` (OS)
temporary storage for main
other things related to call???
return address for `max` (`after:`) ← RSP

smaller addresses

return addresses using a stack

```
max:      ...
          ...
          ret

main:     ...
          ...
          call max
after:    ...
          ret
```

stack just after `max`'s `ret`:

return address for `main` (OS)
temporary storage for main
other things related to call??? ← RSP

smaller addresses

return addresses using a stack

```
max:     ...
         ...
         ret

main:    ...
         ...
         call max
after:   ...
         ret
```

stack just before `main`'s `ret`:

return address for `main` (OS) ← RSP

smaller addresses

function calls use the stack

"the" stack

- convention: RSP points to top
- grows 'down' (towards address 0)
- used by `pop`, `push`, `call`, `ret`

used to implement function calls

main reason: support recursive calls

where do (place to return/arguments/local variables/etc.) go?

- when in doubt — use the stack
- optimization: sometimes use registers

calling convention preview

`call` FUNC and RET instructions

...but where do arguments, local variables, etc. go?

what registers can a function call change?

compiler/OS choice! — much more detail later

Linux calling convention preview

return value: RAX

argument 1: RDI; argument 2: RSI

- argument 3: RDX; argument 4: RCX; argument 5: R8; argument 6: R9

local variables: stack or “free” registers

value of RBP, RBX, R12, R13, R14, R15 can’t be changed by function call

- can use them, but must save/restore

simple recursion (C++)

```
long sum(long count) {
    if (count > 0) {
        long partial_sum = sum(count - 1);
        return partial_sum + count;
    } else {
        return 0;
    }
}
```

simple recursion (assembly)

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 --> do base case
    push RDI // save a copy of original RDI
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of original RDI
    add RAX, RDI // ret val = sum(count-1) + count
    ret
base_case:
    mov RAX, 0
    ret
```

simple recursion (assembly)

the stack

return address for `sum(100)`
saved RDI: `100`

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 --> do base case
    push RDI // save a copy of original RDI
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of original RDI
    add RAX, RDI // ret val = sum(count-1) + count
    ret
base_case:
    mov RAX, 0
    ret
```

simple recursion (assembly)

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 --> do base case
    push RDI // save a copy of original RDI
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of original RDI
    add RAX, RDI // ret val = sum(count-1) + count
    ret
base_case:
    mov RAX, 0
    ret
```

the stack

return address for sum(100)
saved RDI: 100
return address for sum(99)

simple recursion (assembly)

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 --> do base case
    push RDI // save a copy of original RDI
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of original RDI
    add RAX, RDI // ret val = sum(count-1) + count
    ret
base_case:
    mov RAX, 0
    ret
```

the stack

return address for sum(100)
saved RDI: 100
return address for sum(99)
saved RDI: 99

simple recursion (assembly)

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 -->
    push RDI // save a copy of origina
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of origin
    add RAX, RDI // ret val = sum(count-1) + count
    ret
base_case:
    mov RAX, 0
    ret
```

the stack

return address for sum(100)
saved RDI: 100
return address for sum(99)
saved RDI: 99
return address for sum(98)
saved RDI: 98
…
return address for sum(1)
saved RDI: 1
return address for sum(0)

simple recursion (assembly)

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 -->
    push RDI // save a copy of origina
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of origin
    add RAX, RDI // ret val = sum(coun
    ret
base_case:
    mov RAX, 0
    ret
```

the stack

return address for sum(100)
saved RDI: 100
return address for sum(99)
saved RDI: 99
return address for sum(98)
saved RDI: 98
…
return address for sum(2)
saved RDI: 2
return address for sum(1)
saved RDI: 1

simple recursion (assembly)

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 -->
    push RDI // save a copy of ori
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of or
    add RAX, RDI // ret val = sum(
    ret
base_case:
    mov RAX, 0
    ret
```

the stack

return address for sum(100)
saved RDI: 100
return address for sum(99)
saved RDI: 99
return address for sum(98)
saved RDI: 98
…
return address for sum(2)
saved RDI: 2
return address for sum(1)

simple recursion (assembly)

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 -->
    push RDI // save a copy of origina
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of origin
    add RAX, RDI // ret val = sum(count-1) + count
    ret
base_case:
    mov RAX, 0
    ret
```

the stack

return address for sum(100)
saved RDI: 100
return address for sum(99)
saved RDI: 99
return address for sum(98)
saved RDI: 98
…
return address for sum(2)
saved RDI: 2

simple recursion (assembly)

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 -->
    push RDI // save a copy of origina
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of original RDI
    add RAX, RDI // ret val = sum(count-1) + count
    ret
base_case:
    mov RAX, 0
    ret
```

the stack

return address for sum(100)
saved RDI: 100
return address for sum(99)
saved RDI: 99
return address for sum(98)
saved RDI: 98
…
return address for sum(2)

simple recursion (assembly)

the stack

return address for `sum(100)`
saved RDI: `100`

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 --> do base case
    push RDI // save a copy of original RDI
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of original RDI
    add RAX, RDI // ret val = sum(count-1) + count
    ret
base_case:
    mov RAX, 0
    ret
```

simple recursion (assembly)

the stack

return address for sum(100)

```
# RDI (arg 1) is count
sum:
    cmp RDI, 0
    jle base_case // if count <= 0 --> do base case
    push RDI // save a copy of original RDI
    sub RDI, 1
    call sum // sum(count-1)
    pop RDI  // restore copy of original RDI
    add RAX, RDI // ret val = sum(count-1) + count
    ret
base_case:
    mov RAX, 0
    ret
```

specifying pointers

```
[RAX + 2 * RBX + 0x1234]
```

optional 64-bit base register *plus*

- example: RAX

optional 64-bit index register times 1 (default), 2, 4, or 8 *plus*

- example: RBX times 2

optional 32-bit signed constant

- labels count as constants

example valid movs

```
mov rax, rbx              // RAX ← RBX
mov rax, [rbx]            // RAX ← memory[RBX]
mov [someLabel], rbx      // memory[someLabel] ← RBX
mov rax, [r13 - 4]        // RAX ← memory[R13 + (-4)]
mov [rsi + rax], cl       // memory[RSI + RAX] ← CL
mov rdx, [rsi + 4*rbx]    // RDX ← memory[RSI + 4 * RE
```

INVALID movs

```
mov rax, [r11 - rcx]
```

can't subtract register

```
mov [rax + r5 + rdi], rbx
```

```
mov [4*rax + 2*rbx], rcx
```

only multiply one register

memory access lengths

move one byte:

```
mov bl, [rax]
mov [rax], bl
mov BYTE PTR [rax], bl
mov BYTE PTR [rbx], 42
```

move four bytes:

```
mov ebx, [rax]
mov [rax], ebx
mov DWORD PTR [rax], ebx
mov DWORD PTR [rbx], 10
```

(BYTE, WORD (2 bytes), DWORD (4 bytes), QWORD (8 bytes))

inc/dec

```
dec RAX
inc QWORD PTR [RBX + RCX]
```

increment or **dec**rement

register or memory operand

(same effect as add/sub 1)

multiply

```
imul <first>, <second>
imul RAX, RBX
imul RAX, [RCX + RDX]
```

first ← first × second

first operand must be register

```
imul <first>, <second>, <third>
imul RAX, RBX, 42
imul RAX, [RCX + RDX], 42
```

first ← second × third

first: must be register; third: must be constant

multiply (with big result)

```
imul <first>
imul RBX
imul QWORD PTR [RCX + RDX]
```

$\{\text{RDX, RAX}\} \leftarrow \text{RAX} \times \text{first}$

RDX gets most significant 64 bits
RAX gets least signiciant 64 bits

```
imul EBX
imul DWORD PTR [RCX + RDX]
```

$\{\text{EDX, EAX}\} \leftarrow \text{EAX} \times \text{first}$

EDX gets most significant 32 bits
EAX gets least signiciant 32 bits

multiply — signed/unsigned

with result size = source size:
signed and unsigned multiply is the same

with bigger results:
`imul` — signed multiply
`mul` — unsigned multiply

divide

```
idiv <first>
idiv RBX
idiv QWORD PTR [RCX + RDX]
```

RAX $\leftarrow$ {RDX,RAX} $\div$ first

RDX $\leftarrow$ {RDX,RAX} $\mod$ first

128-bit divided by 64-bit

or 64-bit by 32-bit with 32-bit first operand, etc.

also `div <first>` — same, but unsigned division

on LEA

LEA = **L**oad **E**ffective **A**ddress

effective address = computed address for memory access

syntax looks like a **mov** from memory, but…

skips the memory access — just uses the address

(sort of like & operator in C?)

`lea RAX, [RAX + 4]` ≈ `add RAX, 4`

on LEA

LEA = **L**oad **E**ffective **A**ddress

effective address = computed address for memory access

syntax looks like a **mov** from memory, but…

skips the memory access — just uses the address

(sort of like & operator in C?)

`lea RAX, [RAX + 4]` ≈ `add RAX, 4`

"address of memory[rax + 4]" = rax + 4

LEA tricks

```
lea RAX, [RAX + RAX * 4]
```

$\text{rax} \leftarrow \text{rax} \times 5$

rax ← `address-of(memory[rax + rax * 4])`

```
lea RDX, [RBX + RCX]
```

$\text{rdx} \leftarrow \text{rbx} + \text{rcx}$

rdx ←`address-of(memory[rbx + rcx])`

call example

```
int max(int x, int y) {
    int theMax;
    if (x > y)
        theMax = x;
    else
        theMax = y;
    return theMax;
}

int main() {
    int maxVal, a = 5, b = 6;
    maxVal = max(a, b);
    cout << "max␣value:␣" << maxVal << endl;
    return 0;
}
```

call example

```
int max(int x, int y) {
    int theMax;
    if (x > y)
        theMax = x;
    else
        theMax = y;
    return theMax;
}

int main() {
    int maxVal, a = 5, b = 6;
    maxVal = max(a, b);
    cout << "max value: " << maxVal << endl;
    return 0;
}
```

where do arguments go?

where do local variables go?

where does the return value go?

how does return know where to go?

calling conventions

calling convention: rules about how function calls work

choice of compiler and OS NOT the processor itself

...but processor might make instructions to help

- x86-64: `call`, `ret`, `push`, `pop`

basic calling convention questions (1)

how does return know where to go?

where do arguments go?

basic calling convention questions (1)

how does return know where to go?

- x86-64: on the stack (otherwise can't use `call`/`ret`)

where do arguments go?

basic calling convention questions (1)

how does return know where to go?

- x86-64: on the stack (otherwise can't use `call`/`ret`)

where do arguments go?

- Linux+x86-64: arguments 1-6: `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`
- Linux+x86-64: arguments 7-: push on the stack (*last* argument first)
- last argument first: so arguments are `pop`'d in order
- (exceptions: objects that don't fit in a register, floating point, ...)

basic calling convention questions (2)

where do local variables go?

where does the return value go?

basic calling convention questions (2)

where do local variables go?

- Linux+x86-64: in registers (if room) or on the stack
- caveat: what registers can function calls change?

where does the return value go?

basic calling convention questions (2)

where do local variables go?

- Linux+x86-64: in registers (if room) or on the stack
- caveat: what registers can function calls change?

where does the return value go?

- Linux+x86-64: RAX

basic calling convention questions (2)

where do local variables go?

- Linux+x86-64: in registers (if room) or on the stack
- caveat: what registers can function calls change?

where does the return value go?

- Linux+x86-64: RAX

saved registers

what registers can function calls change?

- Linux+x86-64: `RAX`, `RCX`, `RDX`, `RSI`, `RDI`, `R8`, `R9`, `R10`, `R11`, floating point registers
- if using for local variables — be careful about function calls

other registers: must have same value when function returns

- if using for local variables — save old value and restore before returning

caller versus callee

```
void foo() {
    ...
}

int main() {
    foo();
    return 0;
}
```

`main` is *caller*

`foo` is *callee*

a function call

```
...
globalVar =
    foo(1, 2, 3, 4,
        5, 6, 7, 8);
...
```

```
// assuming R11
// used for
// local var
// in caller
push R11
mov RDI, 1
mov RSI, 2
mov RDX, 3
mov RCX, 4
mov R8, 5
mov R9, 6
push 8
push 7
call foo
add RSP, 16
pop R11
mov [globalVar], RAX
```

a function call

```
...
globalVar =
    foo(1, 2, 3, 4,
        5, 6, 7, 8);
...
```

```
// assuming R11
// used for
// local var
// in caller
push R11
mov RDI, 1
mov RSI, 2
mov RDX, 3
mov RCX, 4
mov R8, 5
mov R9, 6
push 8
push 7
call foo
add RSP, 16
pop R11
mov [globalVar], RAX
```

save important registers foo *might* change

…and restore saved regs

a function call

```
...
globalVar =
    foo(1, 2, 3, 4,
        5, 6, 7, 8);
...
```

```
// assuming R11
// used for
// local var
// in caller
push R11
mov RDI, 1
mov RSI, 2
mov RDX, 3
mov RCX, 4
mov R8, 5
mov R9, 6
push 8
push 7
call foo
add RSP, 16
pop R11
mov [globalVar], RAX
```

save important registers foo *might* change

place arguments in registers and (if necessary) on stack

…and restore saved regs

a function call

```
...
globalVar =
    foo(1, 2, 3, 4,
        5, 6, 7, 8);
...
```

```
// assuming R11
// used for
// local var
// in caller
push R11
mov RDI, 1
mov RSI, 2
mov RDX, 3
mov RCX, 4
mov R8, 5
mov R9, 6
push 8
push 7
call foo
add RSP, 16
pop R11
mov [globalVar], RAX
```

save important registers foo *might* change

place arguments in registers and (if necessary) on stack

← and actually call function

…and restore saved regs

a function call

```
...
globalVar =
    foo(1, 2, 3, 4,
        5, 6, 7, 8);
...
```

```
// assuming R11
// used for
// local var
// in caller
push R11
mov RDI, 1
mov RSI, 2
mov RDX, 3
mov RCX, 4
mov R8, 5
mov R9, 6
push 8
push 7
call foo
add RSP, 16
pop R11
mov [globalVar], RAX
```

save important registers foo *might* change

place arguments in registers and (if necessary) on stack

← and actually call function

← and pop args from stack (if any)

}...and restore saved regs

a function call

```
...
globalVar =
    foo(1, 2, 3, 4,
        5, 6, 7, 8);
...
```

```
// assuming R11
// used for
// local var
// in caller
push R11
mov RDI, 1
mov RSI, 2
mov RDX, 3
mov RCX, 4
mov R8, 5
mov R9, 6
push 8
push 7
call foo
add RSP, 16
pop R11
mov [globalVar], RAX
```

save important registers foo *might* change

place arguments in registers and (if necessary) on stack

← and actually call function

← and pop args from stack (if any)

…and restore saved regs

…and use return value

caller task summarized

save registers that the function might change (consult list)

place parameters in registers, stack

call

remove any parameters from stack

restore registers that the function might change

use return value in RAX

callee code example (naive version)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a;
    result += b;
    result += c;
    return result;
}
```

address	value
…	
0xF0000000	(caller's stuff)
0xEFFFFFF8	return address for myFunc
0xEFFFFFF0	value of result
0xEFFFFFE8	(next stack allocation)
…	

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0   // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI // result += b
    add QWORD PTR [RSP], RDX // result += c
    mov RAX, QWORD PTR [RSP] // ret val = result
    // deallocate space
    add RSP, 8
    ret
```

callee code example (naive version)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a;
    result += b;
    result += c;
    return result;
}
```

address	value
…	
0xF0000000	(caller's stuff)
0xEFFFFFF8	return address for myFunc
0xEFFFFFF0	value of result
0xEFFFFFE8	(next stack allocation)
…	

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0    // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI //
    add QWORD PTR [RSP], RDX //
    mov RAX, QWORD PTR [RSP] //
    // deallocate space
    add RSP, 8
    ret
```

one policy:

local vars (`result`) lives on stack

accesses arguments directly

callee code example (animated)

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0   // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI // result += b
    add QWORD PTR [RSP], RDX // result += c
    mov RAX, QWORD PTR [RSP] // ret val = result
    // deallocate space
    add RSP, 8
    ret
```

RSP	0x7FFF8
RDI	2
RSI	3
RDX	4
RAX	
…	

	…	
RSP→	0x7FFF8	(ret address)
	0x7FFF0	
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	
	…	

callee code example (animated)

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0   // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI // result += b
    add QWORD PTR [RSP], RDX // result += c
    mov RAX, QWORD PTR [RSP] // ret val = result
    // deallocate space
    add RSP, 8
    ret
```

RSP	0x7FFF0
RDI	2
RSI	3
RDX	4
RAX	
…	

	…	
	0x7FFF8	(ret address)
RSP→	0x7FFF0	
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	
	…	

callee code example (animated)

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0   // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI // result += b
    add QWORD PTR [RSP], RDX // result += c
    mov RAX, QWORD PTR [RSP] // ret val = result
    // deallocate space
    add RSP, 8
    ret
```

RSP	0x7FFF0
RDI	2
RSI	3
RDX	4
RAX	
…	

	…	
	0x7FFF8	(ret address)
RSP→	0x7FFF0	0
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	
	…	

callee code example (animated)

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0  // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI // result += b
    add QWORD PTR [RSP], RDX // result += c
    mov RAX, QWORD PTR [RSP] // ret val = result
    // deallocate space
    add RSP, 8
    ret
```

RSP	0x7FFF0
RDI	2
RSI	3
RDX	4
RAX	
…	

	…	
	0x7FFF8	(ret address)
RSP→	0x7FFF0	2
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	
	…	

callee code example (animated)

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0   // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI // result += b
    add QWORD PTR [RSP], RDX // result += c
    mov RAX, QWORD PTR [RSP] // ret val = result
    // deallocate space
    add RSP, 8
    ret
```

RSP	0x7FFF0
RDI	2
RSI	3
RDX	4
RAX	
…	

	…	
	0x7FFF8	(ret address)
RSP→	0x7FFF0	5
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	
	…	

callee code example (animated)

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0   // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI // result += b
    add QWORD PTR [RSP], RDX // result += c
    mov RAX, QWORD PTR [RSP] // ret val = result
    // deallocate space
    add RSP, 8
    ret
```

RSP	0x7FFF0
RDI	2
RSI	3
RDX	4
RAX	
...	

	...	
	0x7FFF8	(ret address)
RSP→	0x7FFF0	9
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	
	...	

callee code example (animated)

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0   // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI // result += b
    add QWORD PTR [RSP], RDX // result += c
    mov RAX, QWORD PTR [RSP] // ret val = result
    // deallocate space
    add RSP, 8
    ret
```

RSP	0x7FFF0
RDI	2
RSI	3
RDX	4
RAX	9
…	

	…	
	0x7FFF8	(ret address)
RSP→	0x7FFF0	9
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	
	…	

callee code example (animated)

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0   // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI // result += b
    add QWORD PTR [RSP], RDX // result += c
    mov RAX, QWORD PTR [RSP] // ret val = result
    // deallocate space
    add RSP, 8
    ret
```

RSP	0x7FFF8
RDI	2
RSI	3
RDX	4
RAX	9
…	

	…	
RSP→	0x7FFF8	(ret address)
	0x7FFF0	9
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	
	…	

callee code example (animated)

```
myFunc:
    // allocate space for result
    sub RSP, 8
    mov QWORD PTR [RSP], 0   // result = 0
    add QWORD PTR [RSP], RDI // result += a
    add QWORD PTR [RSP], RSI // result += b
    add QWORD PTR [RSP], RDX // result += c
    mov RAX, QWORD PTR [RSP] // ret val = result
    // deallocate space
    add RSP, 8
    ret
```

RSP	0x80000
RDI	2
RSI	3
RDX	4
RAX	9
…	

RSP→	…	
	0x7FFF8	(ret address)
	0x7FFF0	9
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	
	…	

callee code example (allocate registers)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a; result += b; result += c;
    return result;
}
```

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI    // store a in R8 (not callee-saved)
    mov R9, RSI    // store b in RBP
    mov RBX, RDX   // store c in RBX
    mov R12, 0     // result = 0
    add R12, R8    // result += a
    add R12, R9    // result += b
    add R12, RBX   // result += c
    mov RAX, R12   // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

address	value
...	
0xFF000	(caller's stuff)
0xEFFF8	return address ...
0xEFFF0	saved RBX
0xEFFE8	saved R12
...	

callee code example (allocate registers)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a; result += b; result += c;
    return result;
}
```

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

address	**value**
…	
0xFF000	(caller's stuff)
0xEFFF8	return address …
0xEFFF0	saved RBX
0xEFFE8	saved R12
…	

callee code example (allocate registers)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a; result += b; result += c;
    return result;
}
```

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI    // store a in R8 (not callee-saved)
    mov R9, RSI    // store b in RBP
    mov RBX, RDX   // store c in RBX
    mov R12, 0     // result = 0
    add R12, R8    // result += a
    add R12, R9    // result += b
    add R12, RBX   // result += c
    mov RAX, R1
    pop R12
    pop RBX
    ret
```

address	value
...	
0xFF000	(caller's stuff)
0xEFFF8	return address ...
0xEFFF0	saved RBX
	saved R12

another policy:
allocate new registers for local vars
...and aren't a, b, c local vars?

callee code example (allocate registers)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a; result += b; result += c;
    return result;
}
```

```
myFunc:
    push RBX   // save old RBX, which we've decided to use for c
    push R12   // save old R12, to be used for result
    mov R8, RDI    // store a in R8 (not callee-saved)
    mov R9, RSI    // store b in RBP
    mov RBX, RDX   // store c in RBX
    mov R12, 0     // result = 0
    add R12, R8    // result += a
    add R12, R9    // result += b
    add R12, RBX   // result += c
    mov RAX, R1
    pop R12   /
    pop RBX
    ret
```

address	value
...	
0xFF000	(caller's stuff)
0xEFFF8	return address ...
0xEFFF0	saved RBX
	saved R12

using registers for variables?
if callee-saved, save and restore old

callee code example (allocate registers)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a; result += b; result += c;
    return result;
}
```

```
myFunc:
    push RBX   // save old RBX, which we've decided to use for c
    push R12   // save old R12, to be used for result
    mov R8, RDI    // store a in R8 (not callee-saved)
    mov R9, RSI    // store b in RBP
    mov RBX, RDX   // store c in RBX
    mov R12, 0     // result = 0
    add R12, R8    // result += a
    add R12, R9    // result += b
    add R12, RBX   // result += c
    mov R
    pop R
    pop R
    ret
```

address	value
...	
0xFF000	(caller's stuff)
0xEFFF8	return address ...
0xEFFF0	saved RBX
	ed R12

using registers for variables?

if caller-saved, it's okay to overwrite w/o saving

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFF8
RDI	2
RSI	3
RDX	4
R8	4
R9	4
R12	0x5678
RAX	
RBX	0x1234
…	

	…	
RSP→	0x7FFF8	(ret address)
	0x7FFF0	
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFF0
RDI	2
RSI	3
RDX	4
R8	4
R9	4
R12	0x5678
RAX	
RBX	0x1234
…	

	…	
	0x7FFF8	(ret address)
RSP→	0x7FFF0	0x1234
	0x7FFE8	
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	4
R9	4
R12	0x5678
RAX	
RBX	0x1234
...	

	...	
	0x7FFF8	(ret address)
	0x7FFF0	0x1234
RSP→	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX   // save old RBX, which we've decided to use for c
    push R12   // save old R12, to be used for result
    mov R8, RDI    // store a in R8 (not callee-saved)
    mov R9, RSI    // store b in RBP
    mov RBX, RDX   // store c in RBX
    mov R12, 0     // result = 0
    add R12, R8    // result += a
    add R12, R9    // result += b
    add R12, RBX   // result += c
    mov RAX, R12   // ret val = result
    pop R12    // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	2
R9	4
R12	0x5678
RAX	
RBX	0x1234
…	

	…	
	0x7FFF8	(ret address)
	0x7FFF0	0x1234
RSP→	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12  // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	2
R9	3
R12	0x5678
RAX	
RBX	0x1234
…	

	…	
	0x7FFF8	(ret address)
	0x7FFF0	0x1234
RSP→	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	2
R9	3
R12	0x5678
RAX	
RBX	4
…	

	…	
	0x7FFF8	(ret address)
	0x7FFF0	0x1234
RSP→	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	2
R9	3
R12	0
RAX	
RBX	4
...	

	...	
	0x7FFF8	(ret address)
	0x7FFF0	0x1234
RSP→	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX   // save old RBX, which we've decided to use for c
    push R12   // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12    // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	4
R9	3
R12	4
RAX	
RBX	4
…	

	…	
	0x7FFF8	(ret address)
	0x7FFF0	0x1234
RSP→	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	4
R9	3
R12	7
RAX	
RBX	4
…	

	…	
	0x7FFF8	(ret address)
	0x7FFF0	0x1234
RSP→	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	4
R9	3
R12	9
RAX	
RBX	2
…	

	…	
	0x7FFF8	(ret address)
	0x7FFF0	0x1234
RSP→	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	4
R9	3
R12	9
RAX	9
RBX	2
...	

	...	
	0x7FFF8	(ret address)
	0x7FFF0	0x1234
RSP→	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFF0
RDI	2
RSI	3
RDX	4
R8	4
R9	3
R12	0x5678
RAX	9
RBX	2
…	

	…	
	0x7FFF8	(ret address)
RSP→	0x7FFF0	0x1234
	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	4
R9	3
R12	0x5678
RAX	9
RBX	0x1234
…	

	…	
RSP→	0x7FFF8	(ret address)
	0x7FFF0	0x1234
	0x7FFE8	0x5678
	0x7FFE0	
	0x7FFD8	
	0x7FFD0	

callee code example (animated)

```
myFunc:
    push RBX  // save old RBX, which we've decided to use for c
    push R12  // save old R12, to be used for result
    mov R8, RDI   // store a in R8 (not callee-saved)
    mov R9, RSI   // store b in RBP
    mov RBX, RDX  // store c in RBX
    mov R12, 0    // result = 0
    add R12, R8   // result += a
    add R12, R9   // result += b
    add R12, RBX  // result += c
    mov RAX, R12  // ret val = result
    pop R12   // restore old R12
    pop RBX
    ret
```

RSP	0x7FFE8
RDI	2
RSI	3
RDX	4
R8	4
R9	3
R12	0x5678
RAX	9
RBX	0x1234
…	

RSP→ …	
0x7FFF8	(ret address)
0x7FFF0	0x1234
0x7FFE8	0x5678
0x7FFE0	
0x7FFD8	
0x7FFD0	

what do compilers do?

must:

deallocate any allocated stack space
save/restore certain registers
look for arguments in certain places
put return value in certain place

but lots of policies for where to put locals...

what do compilers actually do?

it depends...

callee code example (no optimizations)

```
myFunc:
  // allocate memory for a, b, c, result
  sub     rsp, 32
  mov     qword ptr [rsp + 24], rdi // copy a from arg
  mov     qword ptr [rsp + 16], rsi // copy b from arg
  mov     qword ptr [rsp + 8], rdx  // copy c from arg
  mov     qword ptr [rsp], 0        // result = 0
  mov     rdx, qword ptr [rsp + 24] // rdx = a
  add     rdx, qword ptr [rsp]      // rdx += result
  mov     qword ptr [rsp], rdx      // result = rdx
  mov     rdx, qword ptr [rsp + 16] // rdx = b
  add     rdx, qword ptr [rsp]      // rdx += result
  mov     qword ptr [rsp], rdx      // result = rdx
  mov     rdx, qword ptr [rsp + 8]  // rdx = c
  add     rdx, qword ptr [rsp]      // ...
  mov     qword ptr [rsp], rdx
  mov     rax, qword ptr [rsp]      // ret val = result
  // dealocate memory for a, b, c, result
  add     rsp, 32
  ret
```

callee code example (no optimizations)

```
myFunc:
  // allocate memory for a, b, c, result
  sub     rsp, 32
  mov     qword ptr [rsp + 24], rdi // copy a from arg
  mov     qword ptr [rsp + 16], rsi // copy b from arg
  mov     qword ptr [rsp + 8], rdx  // copy c from arg
  mov     qword ptr [rsp], 0        // result = 0
  mov     rdx, qword ptr [rsp + 24] // rdx = a
  add     rdx, qword ptr [rsp]      // rdx += result
  mov     qword ptr [rsp], rdx
  mov     rdx, qword ptr [rsp + 16
  add     rdx, qword ptr [rsp]
  mov     qword ptr [rsp], rdx
  mov     rdx, qword ptr [rsp + 8]
  add     rdx, qword ptr [rsp]
  mov     qword ptr [rsp], rdx
  mov     rax, qword ptr [rsp]
  // dealocate memory for a, b, c,
  add     rsp, 32
  ret
```

address	value
…	
0xF000	(caller's stuff)
0xEFF8	return address …
0xEFF0	value of a
0xEFE8	value of b
0xEFE0	value of c
0xEFD8	value of result
…	

callee code example (no optimizations)

```
myFunc:
  // allocate memory for a, b, c, result
  sub     rsp, 32
  mov     qword ptr [rsp + 24], rdi // copy a from arg
  mov     qword ptr [rsp + 16], rsi // copy b from arg
  mov     qword ptr [rsp + 8], rdx  // copy c from arg
  mov     qword ptr [rsp], 0        // result = 0
  mov     rdx, qword ptr [rsp + 24] // rdx = a
  add     rdx, qword ptr [rsp]      // rdx += result
  mov     qword ptr [rsp], rdx
  mov     rdx, qword ptr [rsp + 16
  add     rdx, qword ptr [rsp]
  mov     qword ptr [rsp], rdx
  mov     rdx, qword ptr [rsp + 8]
  add     rdx, qword ptr [rsp]
  mov     qword ptr [rsp], rdx
  m
  /
  a
  ret
```

address	value
…	
0xF000	(caller's stuff)
0xEFF8	return address …
0xEFF0	value of a
0xEFE8	value of b
	of c
	of result

pretty inefficient — but obeys calling convention
one thing `clang` can generate without optimizations

optimizations versus no

things that always work:

- allocate stack space for local variables
- always put values in their variable right away
- don't reuse argument/return value registers

things clever compilers can do

- place some local variables in registers
- skip storing values that aren't used
- reuse argument/return value registers when not calling/returning

callee code example (better version)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a;
    result += b;
    result += c;
    return result;
}
```

```
myFunc:
  mov RAX, 0
  add RAX, RSI
  add RAX, RDI
  add RAX, RDX
  ret
```

address	value
...	
0xF0000000	(caller's stuff)
0xEFFFFFF8	return address for myFunc
0xEFFFFFE8	(next stack allocation)
...	

callee code example (better version)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a;
    result += b;
    result += c;
    return result;
}
```

```
myFunc:
   mov RAX, 0
   add RAX, RSI
   add RAX, RDI
   add RAX, RDX
   ret
```

address	value
...	
0xF0000000	(caller's stuff)
0xEFFFFFF8	return address for myFunc
0xEFFFFFE8	(next stack allocation)
...	

callee code example (better version)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a;
    result += b;
    result += c;
    return result;
}
```

```
myFunc:
   mov RAX, 0
   add RAX, RSI
   add RAX, RDI
   add RAX, RDX
   ret
```

address	value
…	
0xF0000000	(caller's stuff)
0xEFFFFFF8	return address for myFunc
0xEFFFFFE8	(next stack allocation)
…	

optimization: place result in RAX — avoid copy at end
caller can't tell — RAX will be overwritten anyways

callee code example (better version)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a;
    result += b;
    result += c;
    return result;
}
```

```
myFunc:
  mov RAX, 0
  add RAX, RSI
  add RAX, RDI
  add RAX, RDX
  ret
```

address	value
…	
0xF0000000	(caller's stuff)
0xEFFFFFF8	return address for myFunc
0xEFFFFFE8	(next stack allocation)
…	

optimization: use argument registers directly
avoid copy at beginning (caller can't tell)

callee code example (good version)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a;
    result += b;
    result += c;
    return result;
}
```

address	value
…	
0xF0000000	(caller's stuff)
0xEFFFFFF8	return address for myFunc
0xEFFFFFE8	(next stack allocation)
…	

```
myFunc:
  lea rax, [rdi + rsi]  // return value = a + b
  add rax, rdx          // return value += c
  ret
```

callee code example (good version)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a;
    result += b;
    result += c;
    return result;
}
```

address	value
…	
0xF0000000	(caller's stuff)
0xEFFFFFF8	return address for myFunc
0xEFFFFFE8	(next stack allocation)
…	

```
myFunc:
  lea rax, [rdi + rsi]  // return value = a + b
  add rax, rdx          // return value += c
  ret
```

callee code example (good version)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a;
    result += b;
    result += c;
    return result;
}
```

address	value
…	
0xF0000000	(caller's stuff)
0xEFFFFFF8	return address for myFunc
0xEFFFFFE8	(next stack allocation)
…	

```
myFunc:
  lea rax, [rdi + rsi]  // return value = a + b
  add rax, rdx          // return value += c
  ret
```

what `clang` generates with optimizations

writing called functions

save any callee-saved registers function uses

RBP, RBX, R12-R15,

allocate stack space for local variables or temporary storage

(actual function body)

place return value in RAX

deallocate stack space

restore any saved registers

callee code example (save registers weirdly)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a; result += b; result += c;
    return result;
}

myFunc:
    mov R8, RBX // save old RBX, but to R8
    mov R9, RBP // save old RBP, but to R9
    push R12  // save old R12, which we've decided to use for resu
    mov RAX, RDI             // store a in RAX
    mov RBP, RSI             // store b in RBP
    mov RBX, RDX             // store c in RBX
    mov R12, 0               // result = 0
    add R12, RAX    // result += a
    add R12, RBP    // result += b
    add R12, RBX    // result += c
    mov RAX, R12    // ret val = result
    mov RBX, R8 // restore old RBX
    pop R12    // restore old R12
    mov RBP, R9 // restore old RBP
```

callee code example (save registers weirdly)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a; result += b; result += c;
    return result;
}

myFunc:
    mov R8, RBX // save old RBX, but to R8
    mov R9, RBP // save old RBP, but to R9
    push R12  // save old R12, which we've decided to use for resu
    mov RAX, RDI              // store a in RAX
    mov RBP, RSI              // store b in RBP
    mov RBX, RDX              // store c in RBX
    mov R12, 0                // result = 0
    add R12, RAX    // result += a
    add R12, RBP    // result += b
    add R12, RBX    // result += c
    mov RAX, R12    // ret val = result
    mov RBX, R8 // restore old RBX
    pop R12   // restore old R12
    mov RBP, R9 // restore old RBP
```

callee code example (save registers weirdly)

```
long myFunc(long a, long b, long c) {
    long result = 0;
    result += a; result += b; result += c;
    return result;
}

myFunc:
    mov R8, RBX // save old RBX, but to R8
    mov R9, RBP // save old RBP, but to R9
    push R12  // save old R12, which we've decided to use for resu
    mov RAX, RDI                // store a in RAX
    mov RBP, RSI                // store b in RBP
    mov RBX, RDX                // store c in RBX
    mov R12, 0                  // result = 0
    add R12, RAX    // result += a
    add R12, RBP    // result += b
    add R12, RBX    // result += c
    mov RAX, R12    // ret val = result
```

calling convention doesn't specify *how* you save/restore registers
anything is fine as long as values are restored

activation records

calling subroutine puts some things on stack:

- saved register values
- parameters (if not in registers)
- local variables
- return address

together called the
activation record
for the subroutine

	...
foo's activation record	caller saved registers
	return address of `foo`
	local variables of `foo`
	callee saved registers
bar's activation record	caller saved registers
	return address of `bar`
	local variables of `bar`
	callee saved registers
	...

missing calling conv. parts

floating point arguments/return values?
- floating point registers...

arguments/return values too big for register
- arguments: passed on stack
- return value: caller allocates space, passes pointer

class methods
- implicit `this` argument, usually
- extra stuff for inheritence

calling convention complete version (C)

System V Application Binary Interface
AMD64 Architecture Processor Supplement
(With LP64 and ILP32 Programming Models)
Version 1.0

Edited by
H.J. Lu[1], Michael Matz[2], Milind Girkar[3], Jan Hubička[4], Andreas Jaeger[5], Mark Mitchell[6]

January 28, 2018

https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI

section 3.2 covers calling convention

C++ calling convention

Itanium C++ ABI

Revised March 14, 2017

Introduction

The Itanium C++ ABI is an ABI for C++. As an ABI, it gives precise rules for implementing the language, ensuring that separately-compiled parts of a program can successfully interoperate. Although it was initially developed for the Itanium architecture, it is not platform-specific and can be layered portably on top of an arbitrary C ABI. Accordingly, it is used as the standard C++ ABI for many major operating systems on all major architectures, and is implemented in many major C++ compilers, including GCC and Clang.

`https://itanium-cxx-abi.github.io/cxx-abi/`

and/or/xor

```
and <first>, <second>
xor <first>, <second>
or <first>, <second>
```

bit-by-bit and, or, xor

e.g. if RAX $= 1110_{\text{TWO}}$ and RBX $= 0101_{\text{TWO}}$

`and RAX, RBX` $\rightarrow$ RAX becomes 0100_{TWO}
`xor RAX, RBX` $\rightarrow$ RAX becomes 1011_{TWO}
`or RAX, RBX` $\rightarrow$ RAX becomes 1111_{TWO}

cmp+jmp

earlier idea: pair of compare + conditional jump

actually `CMP` one of many instruction that sets *flags*

other flag setting instructions

compilers omit CMP by using subtraction, etc.

implicit compare result to 0 (almost)

e.g.:

```
loop:   add RBX, RBX
        sub RAX, 1
        jne loop
```

is the same as

```
loop:   add RBX, RBX
        sub RAX, 1
        cmp RAX, 0
        jne loop
```

TEST/CMP

TEST instruction:

performs bitwise and, set flags, discard result

`TEST RAX, RAX` $\approx$ `CMP RAX, 0`

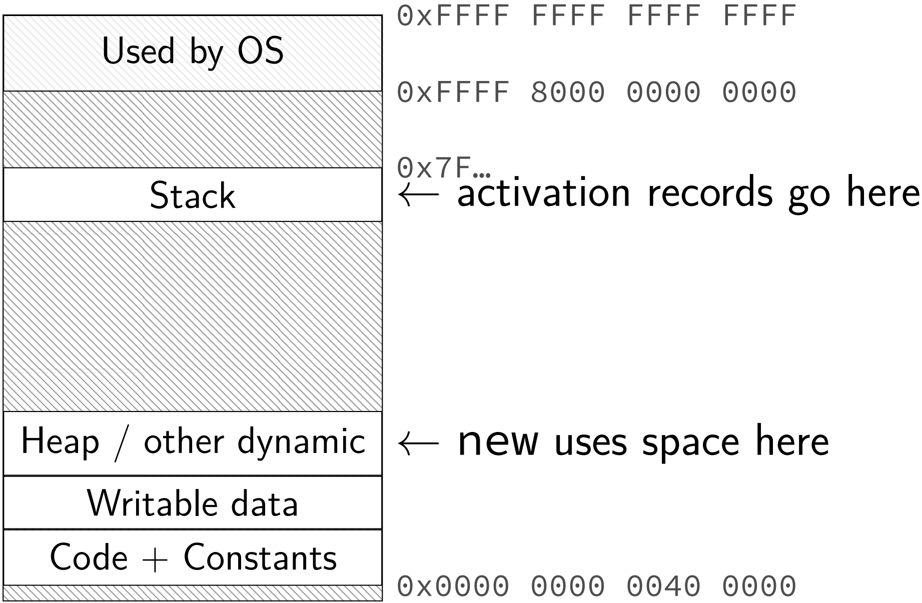
`TEST RAX, RAX` $\approx$ `AND RAX, RAX`

CMP instruction:

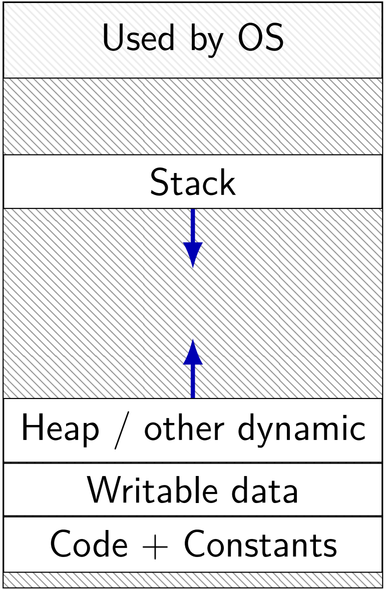
perform subtraction, set flags, discard result

`CMP RAX, RBX` $\approx$ `PUSH RBX; SUB RAX, RBX; POP RBX`

program memory (x86-64 Linux)



program memory (x86-64 Linux)



`0xFFFF FFFF FFFF FFFF`

`0xFFFF 8000 0000 0000`

`0x7F...`

← activation records go here

stack grows towards heap (activation records)
heap grows towards stack (allocations with new)
hopefully never meet

← new uses space here

`0x0000 0000 0040 0000`

godbolt.org

"compiler explorer"

many, many C++ compilers

does work of extracting just the relevant assembly

also does "demangling"

- translate 'mangled' assembly names to C++ names

optimizing away

```
int foo() { return 42; }
int example() { return 1 + foo(); }
```

possible generated asm:

```
...
_Z8example1v:
    mov EAX, 43
    ret
```

```
int foo();
int example() { return 1 + foo(); }
```

possible asm:

```
_Z8example1v:
    push RAX
    call _Z4foo1v
    add EAX, 1
    pop RCX
    ret
```

getting assembly output from clang

`clang++ -S ... file.cpp` — write assembly to `file.s`

- in machine's AT&T assembly syntax
- not the syntax you will be coding

`clang++ -mllvm --x86-asm-syntax=intel -S ... file.cpp` — …in Intel-like syntax

- much closer to syntax you will be coding
- but won't work with `nasm`

test_abs.cpp

```cpp
#include <iostream>
using namespace std;
extern "C" long absolute_value(long x);

long absolute_value(long x) {
    if (x<0)    // if x is negative
        x = -x;    // negate x
    return x;    // return x
}

int main() {
    long theValue=0;
    cout << "Enter␣a␣value:␣" << endl;
    cin >> theValue;
    long theResult = absolute_value(theValue);
    cout << "The␣result␣is:␣" << theResult << endl;
    return 0;
}
```

absolute_value

`clang++ -S`: (AT&T syntax)

...

```
absolute_value:
    movq    %rdi, -8(%rsp)
    cmpq    $0, -8(%rsp)
    jge    .LBB1_2
    xorl    %eax, %eax
    movl    %eax, %ecx
    subq    -8(%rsp), %rcx
    movq    %rcx, -8(%rsp)
.LBB1_2:
    movq    -8(%rsp), %rax
    retq
```

...

AT&T syntax

destination last

% = register

`disp(base)` same as `memory[disp + base]`

`disp(base, index, scale)` same as
`memory[disp + base + index * scale]`

can omit disp /or omit base (defaults to `0`) and/or scale (defualts to `1`)

$ means constant/number

plain number/label means value in memory

`movq/addq/...` — 8 bytes (quad) mov/add

`movl` — 4 bytes (long); `movw` — 2 bytes (word); `movb` 1 byte

absolute_value (unoptimized)

`clang++ -S --mllvm --x86-asm-syntax=intel -S -fomit-frame-pointer`:

```
absolute_value:
    mov     qword ptr [rsp - 8], rdi
    cmp     qword ptr [rsp - 8], 0
    jge     .LBB1_2
    xor     eax, eax
    mov     ecx, eax
    sub     rcx, qword ptr [rsp - 8]
    mov     qword ptr [rsp - 8], rcx
.LBB1_2:
    mov     rax, qword ptr [rsp - 8]
    ret
```

absolute_value_int (unoptimized)

longs replaced with ints

`clang++ -S --mllvm --x86-asm-syntax=intel -S -fomit-frame-pointer`:

```
absolute_value_int:
  mov dword ptr [rsp - 4], edi
  cmp dword ptr [rsp - 4], 0
  jge .LBB0_2
  xor eax, eax
  sub eax, dword ptr [rsp - 4]
  mov dword ptr [rsp - 4], eax
.LBB0_2:
  mov eax, dword ptr [rsp - 4]
  ret
```

absolute_value (optimized)

`clang++ -S -O2 --mllvm --x86-asm-syntax=intel -S -fomit-frame-pointer`:

```
absolute_value:
  mov rax, rdi
  neg rax
  cmovl rax, rdi
  ret
```

(`cmovl` — mov if flags say less than;
and negate sets those flags)

---

my recommendation: use some optimization option when generating assembly to look at

absolute value without cmov (1)

what if we didn't know about cmovXX...?

```
// NASM syntax:
global absolute_value
// GNU assembler syntax: .global absolute_value

absolute_value:
    mov rax, rdi       // x = return value ← arg 1
    cmp rax, 0         // x == 0?
    jge end_of_procedure
    neg rax            // NEGate
end_of_procedure:
    ret
```

absolute value without cmov (2)

what if we didn't know about cmov*XX* and neg...?

```
// NASM syntax:
global absolute_value
// GNU assembler syntax: .global absolute_value

absolute_value:
    mov rax, rdi     // x = return value ← arg 1
    cmp rax, 0       // x == 0?
    jge end_of_procedure
    mov rax, 0
    sub rax, rdi
end_of_procedure:
    ret
```

rest of the .s file

I've shown you a little bit of the .s file

there's alot of extra stuff in there...

in context (1)

“text segment” (code)
file information:

```
        .text
        .intel_syntax noprefix
        .file   "test_abs.cpp"
```

in context (2)

```
        .section        .text.startup,"ax",@progbits
        .align  16, 0x90
        .type   __cxx_global_var_init,@function
__cxx_global_var_init:                  # @__cxx_global_var_in
        .cfi_startproc
# BB#0:
        push    rax
.Ltmp0:
        .cfi_def_cfa_offset 16
        movabs  rdi, _ZStL8__ioinit
        call    _ZNSt8ios_base4InitC1Ev
        movabs  rdi, _ZNSt8ios_base4InitD1Ev
        movabs  rsi, _ZStL8__ioinit
        movabs  rdx, __dso_handle
        call    __cxa_atexit
        mov     dword ptr [rsp + 4], eax # 4-byte Spill
```

in context (2)

`__cxx_global_var_init` — function to call global variable constructors/etc.

```
        .s                                              ts
        .align  16, 0x90
        .type   __cxx_global_var_init,@function
__cxx_global_var_init:                  # @__cxx_global_var_in
        .cfi_startproc
# BB#0:
        push    rax
.Ltmp0:
        .cfi_def_cfa_offset 16
        movabs  rdi, _ZStL8__ioinit
        call    _ZNSt8ios_base4InitC1Ev
        movabs  rdi, _ZNSt8ios_base4InitD1Ev
        movabs  rsi, _ZStL8__ioinit
        movabs  rdx, __dso_handle
        call    __cxa_atexit
        mov     dword ptr [rsp + 4], eax # 4-byte Spill
```

in context (2)

`_ZStL8__ioinit` = `std::__ioinit` (global var.)
`_ZNSt8ios_base4InitC1Ev` = `ios_base::Init::Init()` (constructor)

```
        .type   __cxx_global_var_init,@function
__cxx_global_var_init:                  # @__cxx_global_var_in
        .cfi_startproc
# BB#0:
        push    rax
.Ltmp0:
        .cfi_def_cfa_offset 16
        movabs  rdi, _ZStL8__ioinit
        call    _ZNSt8ios_base4InitC1Ev
        movabs  rdi, _ZNSt8ios_base4InitD1Ev
        movabs  rsi, _ZStL8__ioinit
        movabs  rdx, __dso_handle
        call    __cxa_atexit
        mov     dword ptr [rsp + 4], eax # 4-byte Spill
```

in context (2)

.cfi_… — for debugger/exceptions

```
        .secti                                          ogbits
        .align  16, 0x90
        .type   __cxx_global_var_init,@function
__cxx_global_var_init:                    # @__cxx_global_var_in
        .cfi_startproc
# BB#0:
        push    rax
.Ltmp0:
        .cfi_def_cfa_offset 16
        movabs  rdi, _ZStL8__ioinit
        call    _ZNSt8ios_base4InitC1Ev
        movabs  rdi, _ZNSt8ios_base4InitD1Ev
        movabs  rsi, _ZStL8__ioinit
        movabs  rdx, __dso_handle
        call    __cxa_atexit
        mov     dword ptr [rsp + 4], eax # 4-byte Spill
```

in context (3)

```
        .text
        .globl  absolute_value
        .align  16, 0x90
        .type   absolute_value,@function
absolute_value:                          # @absolute_value
        .cfi_startproc
# BB#0:
        mov     qword ptr [rsp - 8], rdi
        cmp     qword ptr [rsp - 8], 0
        jge     .LBB1_2
# BB#1:
        xor     eax, eax
        mov     ecx, eax
        sub     rcx, qword ptr [rsp - 8]
        mov     qword ptr [rsp - 8], rcx
.LBB1_2:
```

in context (3)

```
        .text
        .globl  absolute_value
        .align  16, 0x90
        .type   absolute_value,@function
absolute_value:                                 # @absolute_value
        .
# BB#0:
        mov     qword ptr [rsp - 8], rdi
        cmp     qword ptr [rsp - 8], 0
        jge     .LBB1_2
# BB#1:
        xor     eax, eax
        mov     ecx, eax
        sub     rcx, qword ptr [rsp - 8]
        mov     qword ptr [rsp - 8], rcx
.LBB1_2:
        mov     rax, qword ptr [rsp - 8]
```

.globl — make this label accessible in other files

.type — help linker/debugger/etc.

in context (4)

```
    .globl      main
    .align      16, 0x90
    .type       main,@function
main:                                   # @main
    .cfi_startproc
# BB#0:
    sub rsp, 56
.Ltmp1:
    .cfi_def_cfa_offset 64
    movabs  rdi, _ZSt4cout
    movabs  rsi, .L.str
    mov dword ptr [rsp + 52], 0
    mov qword ptr [rsp + 40], 0
    call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
    movabs  rsi, _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_
    mov rdi, ra_end1-absolute_value
    ...
```

in context (4)

```
	.globl	main
	.align	16, 0x90
	.type	main,@function
main:                                   # @main
	.cfi_startproc
# BB#0:
```

_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc =
ostream& operator<<(ostream&, char const*)

```
	movabs	rdi, _ZSt4cout
	movabs	rsi, .L.str
	mov dword ptr [rsp + 52], 0
	mov qword ptr [rsp + 40], 0
	call	_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
	movabs	rsi, _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_
	mov rdi, ra_end1-absolute_value
	...
```

extern "C"

```
#include <iostream>
using namespace std;
extern "C" long absolute_value(long x);

long absolute_value(long x) {
    if (x<0)          // if x is negative
        x = -x;       // negate x
    return x;         // return x
}

int main() {
    long theValue=0;
    cout << "Enter a value: " << endl;
    cin >> theValue;
    long theResult = absolute_value(theValue);
    cout << "The result is: " << theResult << endl;
    return 0;
}
```

extern "C" — name mangling

with `extern "C"`:

```
absolute_value:
    ...
```

---

without `extern "C"`:

```
_Z14absolute_valuel:
    ...
```

extern C — different args

This not allowed:

```
extern "C" long absolute_value(long x);
extern "C" int absolute_value(int x);
```

because C doesn't allow it, and `extern "C"` means 'C-compatible'.

This is fine:

```
long absolute_value(long x);
int absolute_value(int x);
```

because C++ allows functions with different args, but same name

assembly on Linux:

`_Z14absolute_valuel`, and
`_Z14absolute_valuei`

c++filt

`c++filt` — command line program to translate C++ symbol names

```
$ c++filt
The function is _Z14absolute_valuellll
^D
```

Output: `The function is absolute_value(long, long, long, long)`

frame pointers

stack pointer: points to “top” of stack
- x86 register RSP used for this
- i.e. lowest address on stack
- i.e. location of next stack allocation

frame pointer: pointer to allocation record AKA “stack frame”
- x86 register RBP intended for this

not required by the calling convention
- function can use RSP instead

frame pointer defaults

some systems default to using frame pointers

- easier to deallocate stack space (`mov RSP, RBP`)
- can support "dynamic" stack allocations (`alloca()`)
- easier to write debuggers

our lab machines don't

- at least with optimizations

clang/GCC flags:

- `-fomit-frame-pointer`/`-fno-omit-frame-pointer`
- (clang only) `-mno-omit-leaf-frame-pointer`
- ("leaf" = function that doesn't call anything)

frame pointer code

```
someFunction:
    push RBP  // save old frame pointer
    mov RBP, RSP // top of stack is frame pointer
    sub RSP, 32  // allocate 32 bytes for local vari
    ...
    add [RBP - 8], 1 // someLocalVar += 1
    ...
    mov RSP, RBP // restore old stack pointer
        // instead of: add RSP, 32
    pop RBP
    ret
```

int max(int x, int y)

```
int max(int x, int y) {
    int theMax;
    if (x > y)          // if x > y then x is max
        theMax = x;
    else                // else y is the max
        theMax = y;
    return theMax;      // return the max
}
```

max assembly (unoptimized)

```
max:
    mov     dword ptr [rsp - 4], edi
    mov     dword ptr [rsp - 8], esi
    mov     esi, dword ptr [rsp - 4]
    cmp     esi, dword ptr [rsp - 8]
    jle     .LBB1_2
    mov     eax, dword ptr [rsp - 4]
    mov     dword ptr [rsp - 12], eax
    jmp     .LBB1_3
.LBB1_2:
    mov     eax, dword ptr [rsp - 8]
    mov     dword ptr [rsp - 12], eax
.LBB1_3:
    mov     eax, dword ptr [rsp - 12]
    ret
```

max assembly (unoptimized)

```
max:
    mov     dword ptr [rsp - 4], edi
    mov     dword ptr [rsp - 8], esi
    mov     esi, dword ptr [rsp - 4]
    cmp     esi, dword ptr [rsp - 8]
    jle     .LBB1_2
    mov     eax, dword ptr [rsp - 4]
    mov     dword ptr [rsp - 12], eax
    jmp     .LBB1_3
.LBB1_2:
    mov     eax, dword ptr [rsp - 8]
    mov     dword ptr [rsp - 12], eax
.LBB1_3:
    mov     eax, dword ptr [rsp - 12]
    ret
```

max assembly (unoptimized)

```
max:
    mov     dword ptr [rsp - 4], edi
    mov     dword ptr [rsp - 8], esi
    mov     esi, dword ptr [rsp - 4]
    cmp     esi, dword ptr [rsp - 8]
    jle     .LBB1_2
    mov     eax, dword ptr [rsp - 4]
    mov     dword ptr [rsp - 12], eax
    jmp     .LBB1_3
.LBB1_2:
    mov     eax, dword ptr [rsp - 8]
    mov     dword ptr [rsp - 12], eax
.LBB1_3:
    mov     eax, dword ptr [rsp - 12]
    ret
```

max assembly (unoptimized)

```
max:
    mov     dword ptr [rsp - 4], edi
    mov     dword ptr [rsp - 8], esi
    mov     esi, dword ptr [rsp - 4]
    cmp     esi, dword ptr [rsp - 8]
    jle     .LBB1_2
    mov     eax, dword ptr [rsp - 4]
    mov     dword ptr [rsp - 12], eax
    jmp     .LBB1_3
.LBB1_2:
    mov     eax, dword ptr [rsp - 8]
    mov     dword ptr [rsp - 12], eax
.LBB1_3:
    mov     eax, dword ptr [rsp - 12]
    ret
```

max assembly (optimized)

```
max:
    cmp     edi, esi
    cmovge  esi, edi
    mov     eax, esi
    ret
```

max assembly (optimized)

```
max:
    cmp     edi, esi
    cmovge  esi, edi
    mov     eax, esi
    ret
```

cmovge: mov if greater than or equal

compare_string

```
bool compare_string (const char *theStr1,
                     const char *theStr2) {
    // while *theStr1 is not nul terminator
    // and the current corresponding bytes are equal
    while( (*theStr1 != '\0')
           && (*theStr1 == *theStr2) ) {
        theStr1++;          // increment the pointers to
        theStr2++;          // the next char / byte
    }
    return (*theStr1==*theStr2);
}
```

compare_string (optimized; part 1)

```
compare_string:
    mov     al, byte ptr [rdi]
    test    al, al
    je      .LBB0_4
    inc     rdi
.LBB0_2:
    movzx   ecx, byte ptr [rsi]
    movzx   edx, al
    cmp     edx, ecx
    jne     .LBB0_5
    inc     rsi
    mov     al, byte ptr [rdi]
    inc     rdi
    test    al, al
    jne     .LBB0_2
    ...
```

compare_string (optimized; part 1)

```
compare_string:
    mov     al, byte ptr [rdi]
    test    al, al
    je      .LBB0_4
    inc     rdi
.LBB0_2:
    movzx   ecx, byte ptr [rsi]
    movzx   edx, al
    cmp     edx, ecx
    jne     .LBB0_5
    inc     rsi
    mov     al, byte ptr [rdi]
    inc     rdi
    test    al, al
    jne     .LBB0_2
    ...
```

compare_string (optimized; part 1)

```
compare_string:
    mov     al, byte ptr [rdi]
    test    al, al
    je      .LBB0_4
    inc     rdi
.LBB0_2:
    movzx   ecx, byte ptr [rsi]
    movzx   edx, al
    cmp     edx, ecx
    jne     .LBB0_5
    inc     rsi
    mov     al, byte ptr [rdi]
    inc     rdi
    test    al, al
    jne     .LBB0_2
    ...
```

compare_string (optimized; part 1)

```
compare_string:
    mov      al, byte ptr [rdi]
    test     al, al
    je       .LBB0_4
    inc      rdi
.LBB0_2:
    movzx    ecx, byte ptr [rsi]
    movzx    edx, al
    cmp      edx, ecx
    jne      .LBB0_5
    inc      rsi
    mov      al, byte ptr [rdi]
    inc      rdi
    test     al, al
    jne      .LBB0_2
    ...
```

compare_string (optimized; part 2)

```
.LBB0_4:
    xor      eax, eax
.LBB0_5:
    movzx    ecx, byte ptr [rsi]
    movzx    eax, al
    cmp      eax, ecx
    sete     al
    ret
```

compare_string (optimized; part 2)

```
.LBB0_4:
    xor     eax, eax
.LBB0_5:
    movzx   ecx, byte ptr [rsi]
    movzx   eax, al
    cmp     eax, ecx
    sete    al
    ret
```

fib

```
long fib(unsigned int n) {
    if ((n==0) || (n==1))
        return 1;
    return fib(n-1) + fib(n-2);
}
```

fib

```
long fib(unsigned int n) {
    if ((n==0) || (n==1))
        return 1;
    return fib(n-1) + fib(n-2);
}
```

fib (optimized; part 1)

```
fib:
    push    r14
    push    rbx
    push    rax
    mov     ebx, edi
    mov     eax, ebx
    or      eax, 1
    mov     r14d, 1
    cmp     eax, 1
    je      .LBB0_3
    ...
```

fib (optimized; part 1)

```
fib:
    push    r14
    push    rbx
    push    rax
    mov     ebx, edi
    mov     eax, ebx
    or      eax, 1
    mov     r14d, 1
    cmp     eax, 1
    je      .LBB0_3
    ...
```

save two callee-saved registers

fib (optimized; part 1)

```
fib:
    push    r14
    push    rbx
    push    rax
    mov     ebx, edi
    mov     eax, ebx
    or      eax, 1
    mov     r14d, 1
    cmp     eax, 1
    je      .LBB0_3
    ...
```

x86-64 rule: RSP must be multiple of 16 when `call` happens
(rax not actually restored)

fib (optimized; part 1)

```
fib:
    push    r14
    push    rbx
    push    rax
    mov     ebx, edi
    mov     eax, ebx
    or      eax, 1
    mov     r14d, 1
    cmp     eax, 1
    je      .LBB0_3
    ...
```

if n is 0 or 1...
jumps to code that returns R14

fib (optimized; part 1)

```
fib:
    push    r14
    push    rbx
    push    rax
    mov     ebx, edi
    mov     eax, ebx
    or      eax, 1
    mov     r14d, 1
    cmp     eax, 1
    je      .LBB0_3
    ...
```

edi, ebx both copies of n

fib (optimized; part 2)

```
    add     ebx, -2
    mov     r14d, 1
.LBB0_2:
    lea     edi, [rbx + 1]
    call    fib
    add     r14, rax
    mov     eax, ebx
    or      eax, 1
    add     ebx, -2
    cmp     eax, 1
    jne     .LBB0_2
.LBB0_3:
    mov     rax, r14
    add     rsp, 8
    pop     rbx
    pop     r14
    ret
```

fib (optimized; part 2)

```
    add     ebx, -2
    mov     r14d, 1
.LBB0_2:
    lea     edi, [rbx + 1]
    call    fib
    add     r14, rax
    mov     eax, ebx
    or      eax, 1
    add     ebx, -2
    cmp     eax, 1
    jne     .LBB0_2
.LBB0_3:
    mov     rax, r14
    add     rsp, 8
    pop     rbx
    pop     r14
    ret
```

return r14
undo stack adjustment
restore rbx, r14

fib (optimized; part 2)

```
    add     ebx, −2
    mov     r14d, 1
.LBB0_2:
    lea     edi, [rbx + 1]
    call    fib
    add     r14, rax
    mov     eax, ebx
    or      eax, 1
    add     ebx, −2
    cmp     eax, 1
    jne     .LBB0_2
.LBB0_3:
    mov     rax, r14
    add     rsp, 8
    pop     rbx
    pop     r14
    ret
```

ebx previously set to n=edi

`fib(n-1)`

fib (optimized; part 2)

```
    add     ebx, -2
    mov     r14d, 1
.LBB0_2:
    lea     edi, [rbx + 1]
    call    fib
    add     r14, rax
    mov     eax, ebx
    or      eax, 1
    add     ebx, -2
    cmp     eax, 1
    jne     .LBB0_2
.LBB0_3:
    mov     rax, r14
    add     rsp, 8
    pop     rbx
    pop     r1
    ret
```

trick: replace `fib(n-2)` call with loop

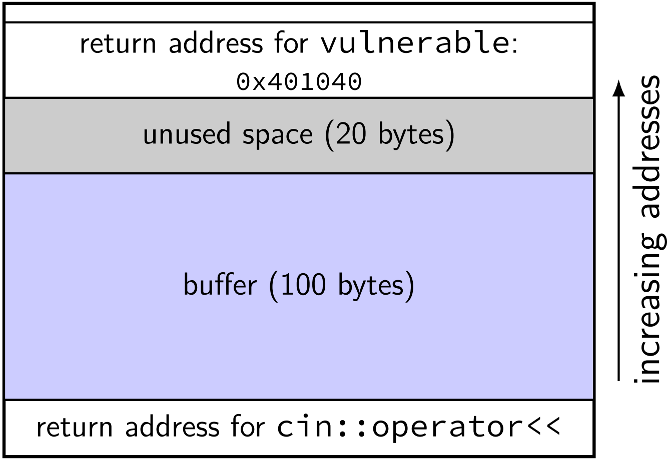
a vulnerable function

```
void vulnerable() {
    char buffer[100];
    cin >> buffer;
}
```

```
  sub rsp, 120
  mov rsi, rsp
  mov edi, /* cin */
  call /* operator>>(istream,char*) */
  add rsp, 120
  ret
```

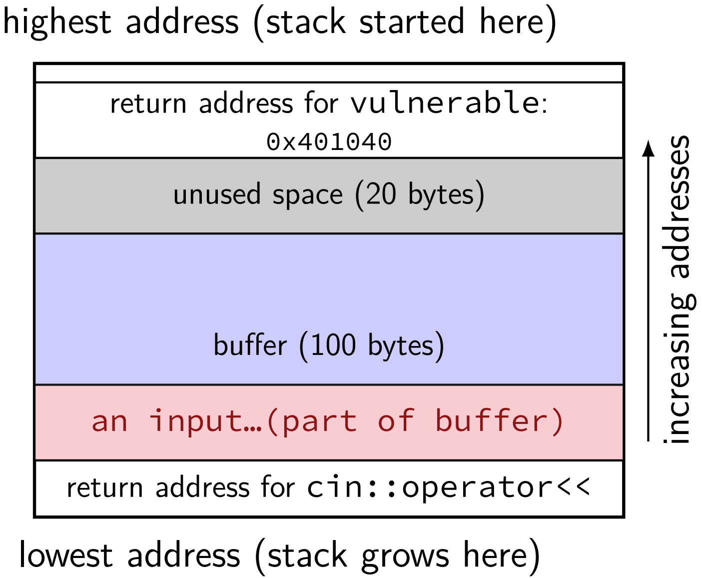
buffer overflows

highest address (stack started here)



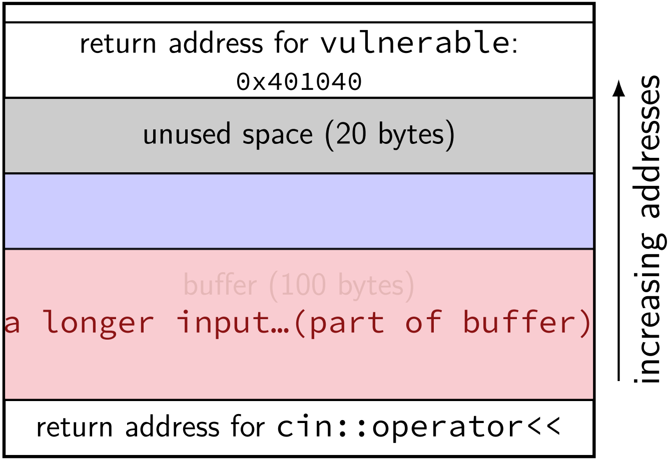
lowest address (stack grows here)

buffer overflows



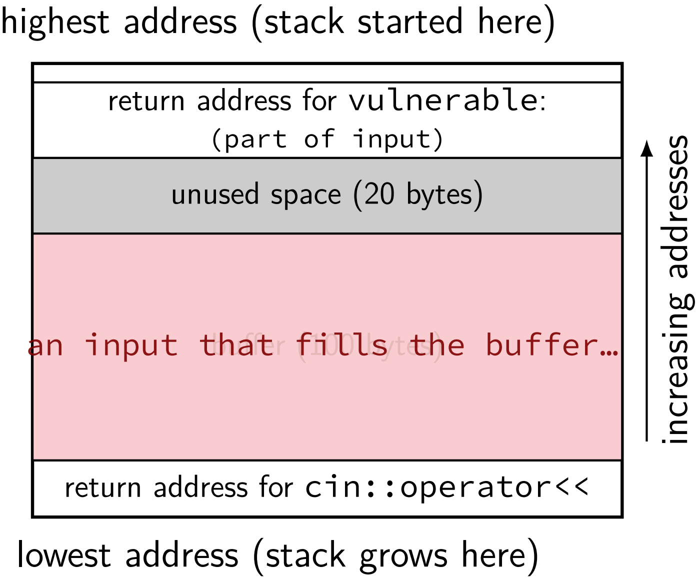
buffer overflows

highest address (stack started here)

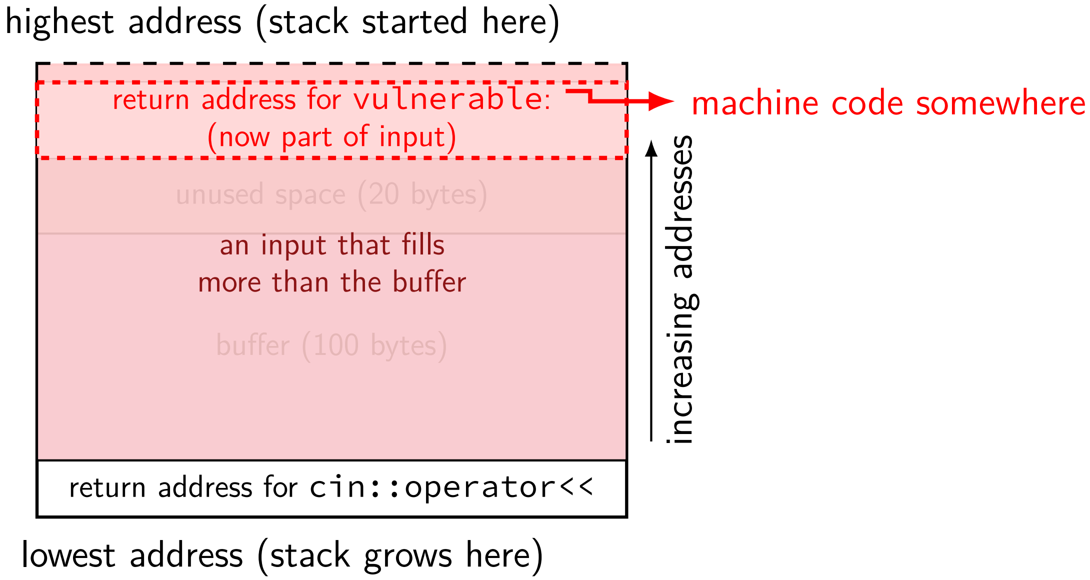


lowest address (stack grows here)

buffer overflows



buffer overflows



variable argument functions

C++ — multiple versions of functions — different assembly names:

> `long foo(long a)` becomes `_Z3fool`
> `long foo(long a, long b)` becomes `_Z3fooll`

can also have variable argument functions — more common in C

example: `void printf(const char *format, ...)` (C equiv. of cout)

```
printf("The␣number␣is␣%d.\n", 42);
```

```
  mov edi, .L.str
  mov esi, 42
  xor eax, eax  // # of floating point args
  call printf
  ...
```