# Assembly (part 1)

## Changelog

Corrections made in this version not seen in first lecture:

- 31 August 2017: slide 34: split out from previous slide; clarify zero/positive/negative
- 31 August 2017: slide 26: put pushq for caller-saved right before call
- 31 August 2017: slide 39-40: use r12 instead of rbx
- 31 August 2017: slide 40: fix typo in start_loop label
- 31 August 2017: slide 19: fix extra junk in assembly
- 4 September 2017: slide 25: %rbx is callee-saved, too

## last time: C hodgepodge

arrays are almost pointers

- arrays include elements — sizeof includes elements
- pointers are addresses — sizeof is size of address

misc. C features: goto, malloc/free, printf

structs in C

- like classes without methods

standards and undefined behavior

## logistics note: lab due times

lab generally due after 11PM on the lab day

- on future labs, please always submit what you have
- (partial credit on many labs is very generous)

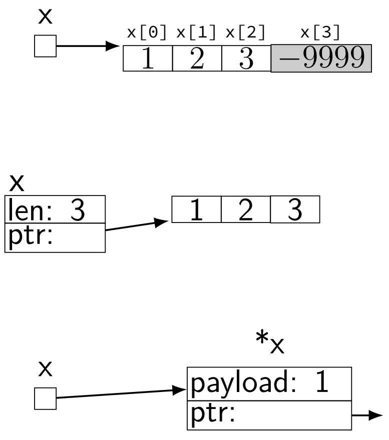
HW generally due next week at noon

## some lists

```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```

```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```

```
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```

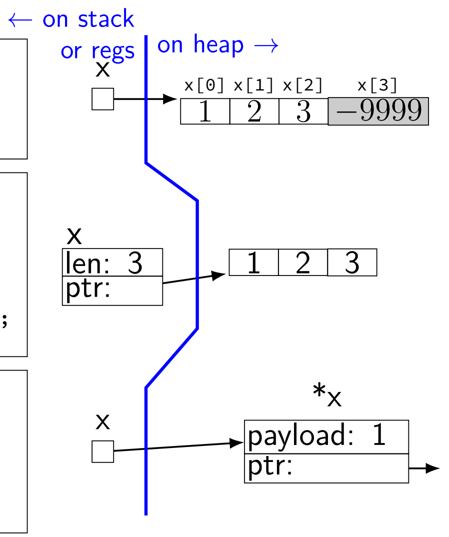


## some lists

```
short sentinel = -9999;
short *x;
x = malloc(sizeof(short)*4);
x[3] = sentinel;
...
```

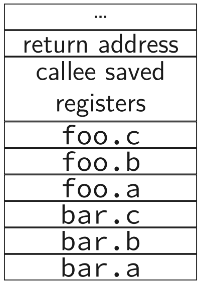
```
typedef struct range_t {
    unsigned int length;
    short *ptr;
} range;
range x;
x.length = 3;
x.ptr = malloc(sizeof(short)*3);
...
```

```
typedef struct node_t {
    short payload;
    list *next;
} node;
node *x;
x = malloc(sizeof(node_t));
...
```



## structs aren’t references

```
typedef struct {
    long a; long b; long c;
} triple;
...

triple foo;
foo.a = foo.b = foo.c = 3;
triple bar = foo;
bar.a = 4;
// foo is 3, 3, 3
// bar is 4, 3, 3
```



## x86-64 refresher

# AT&T versus Intel syntax (1)

AT&T syntax:

```
movq $42, (%rbx)
```

Intel syntax:

```
mov QWORD PTR [rbx], 42
```

effect (pseudo-C):

```
memory[rbx] <- 42
```

# AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

( )s represent value in memory

constants start with $

registers start with %

q (‘quad’) indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

# AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

( )s represent value in memory

constants start with $

registers start with %

q (‘quad’) indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

# AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

( )s represent value in memory

constants start with $

registers start with %

q (‘quad’) indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

## AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q (‘quad’) indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

## AT&T syntax example (1)

```
movq $42, (%rbx)
// memory[rbx] ← 42
```

destination last

()s represent value in memory

constants start with $

registers start with %

q (‘quad’) indicates length (8 bytes)

- l: 4; w: 2; b: 1
- sometimes can be omitted

## AT&T versus Intel syntax (2)

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

## AT&T versus Intel syntax (2)

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

# AT&T versus Intel syntax (2)

AT&T syntax:
```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:
```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):
```
memory[rbx + rcx * 4 + 100] <- 42
```

# AT&T versus Intel syntax (2)

AT&T syntax:
```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:
```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):
```
memory[rbx + rcx * 4 + 100] <- 42
```

# AT&T syntax: addressing

`100(%rbx)`: `memory[rbx + 100]`

`100(%rbx,8)`: `memory[rbx * 8 + 100]`

`100(,%rbx,8)`: `memory[rbx * 8 + 100]`

`100(%rcx,%rbx,8)`:
```
      memory[rcx + rbx * 8 + 100]
```

# AT&T versus Intel syntax (3)

`r8 ← r8 - rax`

Intel syntax: **`sub`** `r8, rax`
AT&T syntax: **`subq`** `%rax, %r8`

same for **`cmpq`**

# AT&T syntax: addresses

```
addq 0x1000, %rax
// Intel syntax: add rax, QWORD PTR [0x1000]
// rax ← rax + memory[0x1000]
addq $0x1000, %rax
// Intel syntax: add rax, 0x1000
// rax ← rax + 0x1000
```

no $ — probably memory address

# AT&T syntax in one slide

destination last

`()` means value in memory

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

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

`$` means constant

plain number/label means value in memory

# recall: x86-64 general purpose registers

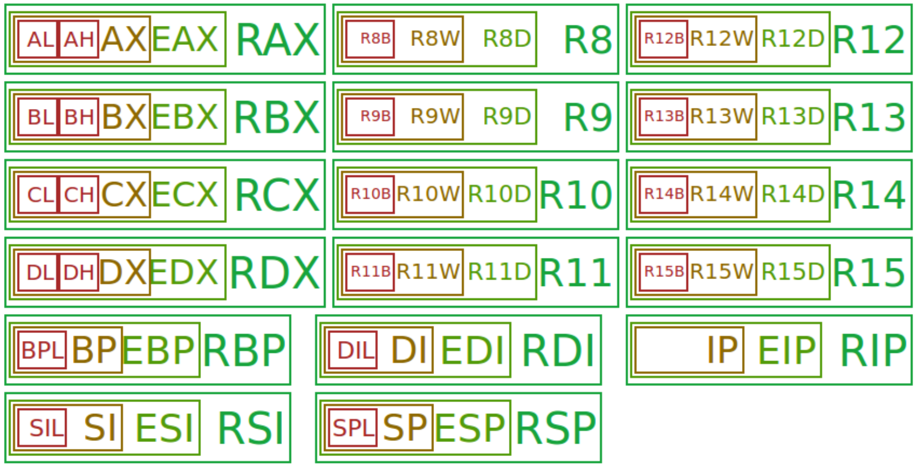


Figure: Immae via Wikipedia

# overlapping registers (1)

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

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

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

## overlapping registers (2)

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

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

%rax is `0xFFFFFFFFFFFFFF01`

## labels (1)

labels represent addresses

## labels (2)

```
    addq string, %rax
    // intel syntax: add rax, QWORD PTR [label]
    // rax ← rax + memory[address of "a string"]
    addq $string, %rax
    // intel syntax: add rax, OFFSET label
    // rax ← rax + address of "a string"
string:  .ascii "a␣string"
```

`addq label`: read value at the address

`addq $label`: use address as an integer constant

## 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

`leaq 4(%rax), %rax` ≈ `addq $4, %rax`

# 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

`leaq 4(%rax), %rax` ≈ `addq $4, %rax`

“address of memory[rax + 4]” = rax + 4

# LEA tricks

```
leaq (%rax,%rax,4), %rax
```

rax ← rax × 5

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

---

```
leaq (%rbx,%rcx), %rdx
```

rdx ← rbx + rcx

rdx ←`address-of(memory[rbx + rcx])`

# x86-64 calling convention

registers for first 6 arguments:

- `%rdi` (or `%edi` or `%di`, etc.), then
- `%rsi` (or `%esi` or `%si`, etc.), then
- `%rdx` (or `%edx` or `%dx`, etc.), then
- `%rcx` (or `%ecx` or `%cx`, etc.), then
- `%r8` (or `%r8d` or `%r8w`, etc.), then
- `%r9` (or `%r9d` or `%r9w`, etc.)

rest on stack

return value in `%rax`

don’t memorize: Figure 3.28 in book

# x86-64 calling convention example

```
int foo(int x, int y, int z) { return 42; }
...
    foo(1, 2, 3);
...
```

---

```
...
    // foo(1, 2, 3)
    movl $1, %edi
    movl $2, %esi
    movl $3, %edx
    call foo  // call pushes address of next instruction
              // then jumps to foo
...
foo:
    movl $42, %eax
```

## call/ret

call:

push address of next instruction on the stack

ret:

pop address from stack; jump

## callee-saved registers

functions must preserve these

%rsp (stack pointer), %rbx, %rbp (frame pointer, maybe)

%r12-%r15

## caller/callee-saved

```
foo:
    pushq %r12 // r12 is caller-saved
    ... use r12 ...
    popq %r12
    ret

...
other_function:
    ...
    pushq %r11 // r11 is caller-saved
    callq foo
    popq %r11
```

## question

```
pushq $0x1
pushq $0x2
addq $0x3, 8(%rsp)
popq %rax
popq %rbx
```

What is value of %rax and %rbx after this?

a. %rax = 0x2, %rbx = 0x4

b. %rax = 0x5, %rbx = 0x1

c. %rax = 0x2, %rbx = 0x1

d. the snippet has invalid syntax or will crash

e. more information is needed

f. something else?

## on %rip

%rip (**I**nstruction **P**ointer) = address of next instruction

`movq 500(%rip), %rax`

- rax ← memory[next instruction address + 500]

## on %rip

%rip (**I**nstruction **P**ointer) = address of next instruction

`movq 500(%rip), %rax`

- rax ← memory[next instruction address + 500]

`label(%rip)` ≈ `label`

- different ways of writing address of label in machine code
  (with %rip — relative to next instruction)

## things we won't cover (today)

floating point; vector operations (multiple values at once)

- special registers: %xmm0 through %xmm15

segmentation (special registers: %ds, %fs, %gs, ...)

lots and lots of instructions

## authoriative source

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D

## if-to-assembly (1)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

## if-to-assembly (1)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

---

```
            if (b >= 42) goto after_then;
            a += 10;
            goto after_else;
after_then: a *= b;
after_else:
```

## if-to-assembly (2)

```
if (b >= 42) {
    a += 10;
} else {
    a *= b;
}
```

---

```
// a is in %rax, b is in %rbx
    cmpq $42, %rbx   // computes rbx - 42 to 0
    jl after_then    // jump if rbx - 42 < 0
                     // AKA rbx < 42
    addq $10, %rax   // a += 1
    jmp after_else
after_then:
    imulq %rbx, %rax // rax = rax * rbx
after_else:
```

## condition codes

x86 has condition codes

set by (almost) all arithmetic instructions

- addq, subq, imulq, etc.

store info about last arithmetic result

- was it zero? was it negative? etc.

# condition codes and jumps

jg, jle, etc. read condition codes

named based on interpreting result of subtraction

0: equal; negative: less than; positive: greater than

# condition codes example (1)

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

# condition codes and cmpq

“last arithmetic result”???

then what is cmp, etc.?

cmp does subtraction (but doesn’t store result)

similar test does bitwise-and

testq %rax, %rax — result is %rax

# condition codes example (2)

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

# loop (1)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

# loop (1)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

```
    int x = 99;
start_loop:
    foo()
    x--;
    if (x >= 0) goto start_loop;
```

# loop (2)

```
int x = 99;
do {
    foo()
    x--;
} while (x >= 0);
```

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    cmpq $0, %r12
    // computes r12 - 0 = r12
    jge start_loop // jump if r12 - 0 >= 0
```

# omitting the cmp

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    cmpq $0, %r12
    // compute r12 - 0 + sets cond. codes
    jge  start_loop  // r12 >= 0?
                     // or result >= 0?
```

```
    movq $99, %r12 // register for x
start_loop:
    call foo
    subq $1, %r12
    // new r12 = old r12 - 1 + sets cond. codes
    jge  start_loop  // old r12 >= 1?
                     // or result >= 0?
```

## condition codes example (3)

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

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

## condition codes examples (4)

```
movq $20, %rbx
addq $-20, %rbx // result is 0
movq $1, %rax // irrelevant
je   foo // taken, result is 0
```

## setting condition codes

most instructions that compute something set condition codes

some instructions only set condition codes:

- `cmp` ∼ `sub`
- `test` ∼ `and` (bitwise and — next week)
- `testq %rax, %rax` — result is %rax

some instructions don't change condition codes:

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

## exercise

```
  movq $3, %rax
  movq $2, %rbx
start_loop:
  addq %rbx, %rbx
  cmpq $3, %rbx
  subq $1, %rax
  jg start_loop
```

What is the value of `%rbx` after this runs?

| | |
|---|---|
| A. 2 | D. 16 |
| B. 4 | E. 32 |
| C. 8 | F. something else |

# logical operators

return 1 for true or 0 for false

```
( 1 && 1 ) == 1        ( 1 || 1 ) == 1
( 2 && 4 ) == 1        ( 2 || 4 ) == 1
( 1 && 0 ) == 0        ( 1 || 0 ) == 1
( 0 && 0 ) == 0        ( 0 || 0 ) == 0
(-1 && -2) == 1        (-1 || -2) == 1
("" && "") == 1        ("" || "") == 1
! 1        == 0
! 4        == 0
!-1        == 0
! 0        == 1
```

# recall: short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

| AND | **false** | **true** |
|---|---|---|
| **false** | false | false |
| **true** | false | true |

# recall: short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

| AND | **false** | **true** |
|---|---|---|
| **false** | false | false |
| **true** | false | true |

# recall: short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

| AND | **false** | **true** |
|---|---|---|
| **false** | false | false |
| **true** | false | true |

## recall: short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

| AND | **false** | **true** |
|---|---|---|
| **false** | false | false |
| **true** | false | true |

## recall: short-circuit (&&)

```
#include <stdio.h>
int zero() { printf("zero()\n"); return 0; }
int one() { printf("one()\n"); return 1; }
int main() {
    printf(">␣%d\n", zero() && one());
    printf(">␣%d\n", one() && zero());
    return 0;
}
```

```
zero()
> 0
one()
zero()
> 0
```

| AND | **false** | **true** |
|---|---|---|
| **false** | false | false |
| **true** | false | true |

## && to assembly

```
return foo() && bar();
```

## && to assembly

```
return foo() && bar();

    result = foo();
    if (result == 0) goto skip_bar;
    result = bar();
skip_bar:
    result = (result != 0);
```

# x86-64 manuals

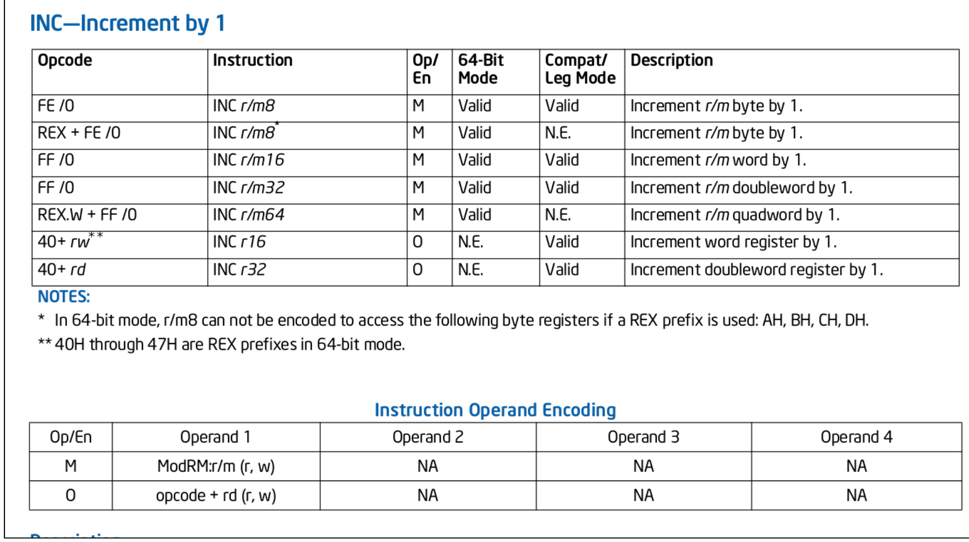
Intel manuals:

https://software.intel.com/en-us/articles/intel-sdm
24 MB, 4684 pages
Volume 2: instruction set reference (2190 pages)

AMD manuals:

https://support.amd.com/en-us/search/tech-docs
“AMD64 Architecture Programmer’s Manual”

# example manual page



INC—Increment by 1

| Opcode | Instruction | Op/ En | 64-Bit Mode | Compat/ Leg Mode | Description |
|---|---|---|---|---|---|
| FE /0 | INC *r/m8* | M | Valid | Valid | Increment *r/m* byte by 1. |
| REX + FE /0 | INC *r/m8** | M | Valid | N.E. | Increment *r/m* byte by 1. |
| FF /0 | INC *r/m16* | M | Valid | Valid | Increment *r/m* word by 1. |
| FF /0 | INC *r/m32* | M | Valid | Valid | Increment *r/m* doubleword by 1. |
| REX.W + FF /0 | INC *r/m64* | M | Valid | N.E. | Increment *r/m* quadword by 1. |
| 40+ *rw*** | INC *r16* | O | N.E. | Valid | Increment word register by 1. |
| 40+ *rd* | INC *r32* | O | N.E. | Valid | Increment doubleword register by 1. |

NOTES:

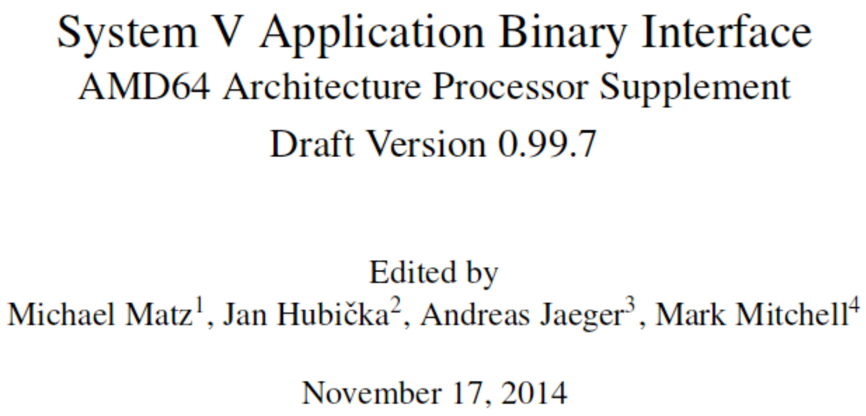
* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** 40H through 47H are REX prefixes in 64-bit mode.

Instruction Operand Encoding

| Op/En | Operand 1 | Operand 2 | Operand 3 | Operand 4 |
|---|---|---|---|---|
| M | ModRM:r/m (r, w) | NA | NA | NA |
| O | opcode + rd (r, w) | NA | NA | NA |

# Linux x86-64 calling convention



System V Application Binary Interface
AMD64 Architecture Processor Supplement
Draft Version 0.99.7

Edited by
Michael Matz[1], Jan Hubička[2], Andreas Jaeger[3], Mark Mitchell[4]

November 17, 2014

# hello.s

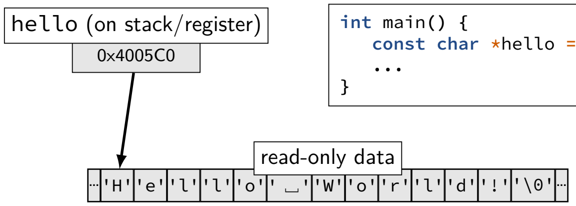
```
        .section        .rodata.str1.1,"aMS",@progb
.LC0:
        .string "Hello, World!"
        .text
        .globl  main
main:
        subq    $8, %rsp
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        addq    $8, %rsp
        ret
```

# hello.o

```
hello.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 g     F .text  0000000000000018 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000005 R_X86_64_32       .rodata.str1.1
000000000000000a R_X86_64_PC32     puts-0x0000000000000004

Contents of section .text:
 0000 4883ec08 bf000000 00e80000 0000b800  H...............
 0010 00000048 83c408c3                    ...H....
Contents of section .rodata.str1.1:
 0000 48656c6c 6f2c2057 6f726c64 2100      Hello, World!.
```

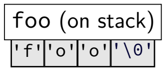
# strings in C

```
int main() {
    const char *hello = "Hello World!";
    ...
}
```


hello (on stack/register)
0x4005C0
read-only data
… 'H' 'e' 'l' 'l' 'o' '␣' 'W' 'o' 'r' 'l' 'd' '!' '\0' …

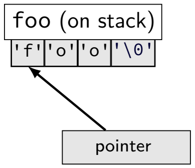
# exercise explanation

```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```



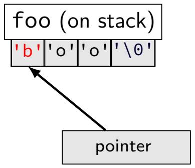
# exercise explanation

```
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```



# exercise explanation

```c
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

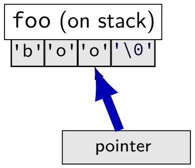

foo (on stack)

'b' 'o' 'o' '\0'

pointer

# exercise explanation

```c
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```


foo (on stack)

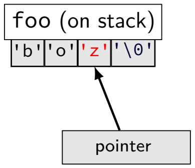
'b' 'o' 'o' '\0'

pointer

# exercise explanation

```c
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

better style: `*pointer = 'z';`


foo (on stack)

'b' 'o' 'z' '\0'

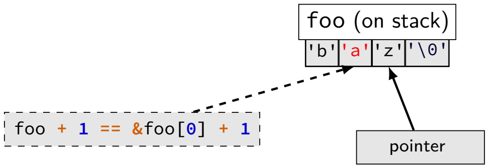
pointer

# exercise explanation

```c
char foo[4] = "foo";
    // {'f', 'o', 'o', '\0'}
char *pointer;
pointer = foo;
*pointer = 'b';
pointer = pointer + 2;
pointer[0] = 'z';
*(foo + 1) = 'a';
```

better style: `*pointer = 'z';`

better style: `foo[1] = 'a';`


foo (on stack)

'b' 'a' 'z' '\0'

`foo + 1 == &foo[0] + 1`

pointer

## middle of blocks?

Examples of things not allowed in 1989 ANSI C:

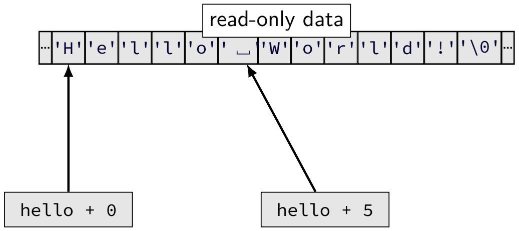
```
printf("Before calling malloc()\n");
int *pointer = malloc(sizeof(int) * 100);
```

`pointer` must be declared earlier

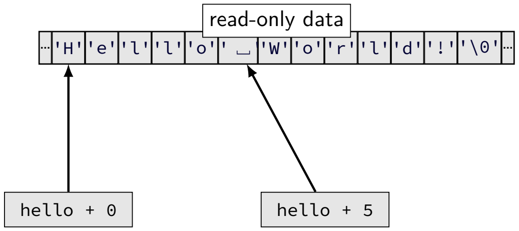
```
for (int x = 0; x < 10; ++x)
```

x must be declared earlier

## pointer arithmetic


read-only data
... 'H' 'e' 'l' 'l' 'o' '␣' 'W' 'o' 'r' 'l' 'd' '!' '\0' ...
hello + 0
hello + 5

## pointer arithmetic


read-only data
... 'H' 'e' 'l' 'l' 'o' '␣' 'W' 'o' 'r' 'l' 'd' '!' '\0' ...
hello + 0
hello + 5

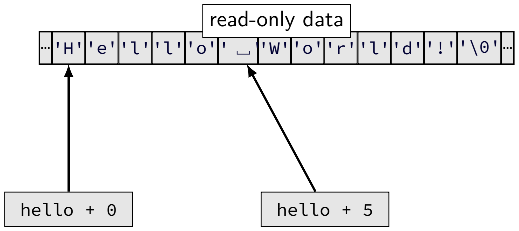
0x4005C0

*(hello + 0) is 'H'

0x4005C5

*(hello + 5) is '␣'

## pointer arithmetic


read-only data
... 'H' 'e' 'l' 'l' 'o' '␣' 'W' 'o' 'r' 'l' 'd' '!' '\0' ...
hello + 0
hello + 5

0x4005C0

*(hello + 0) is 'H'

hello[0] is 'H'

0x4005C5

*(hello + 5) is '␣'

hello[5] is '␣'