linux > gcc | -Og | -O1 | -O2 | -o p p1.c p2.c

↳ optimization flags

- ✓ Program Counter (PC) → %rip in x86-64
- ✓ Integer Register File
- ✓ Condition Code Register
- ✓ A set of vector registers

A program memory contains

* Executable machine code for the program
* Some info for OS
* A run time stack for managing procedure calls and returns
* Blocks of memory allocated by the user → malloc, ...

recall: The program memory is addressed using virtual addresses.

To see the assembly version of a code, compile in this way:

gcc -Og -S mstore.c

To see the binary object code for a program

1. gdb mstore.o
2. x/14xb multstore

To both compile and assemble

gcc -Og -c mstore.c → will generate mstore.o

53 48 89 d3 e8 00 00 00 00 48 89 03 5b c3

← not human readable

objdump -d mstore.o ← disassemble

↳ will generate mstore.d → assembly language equivalent version

```
Disassembly of function sum in binary file mstore.o
0000000000000000 <multstore>:
Offset   Bytes              Equivalent assembly language
   0:    53                 push   %rbx
   1:    48 89 d3           mov    %rdx,%rbx
   4:    e8 00 00 00 00     callq  9 <multstore+0x9>
   9:    48 89 03           mov    %rax,(%rbx)
   c:    5b                 pop    %rbx
   d:    c3                 retq
```

Source Code

↓ preprocessor expands the source code with #include and #define

↓ compiler generates assembly of two source files

p1.s p2.s

↓ assembler converts assembly into binary object-code

p1.o p2.o

↓ linker merges these along with library functions

p

↳ final executable code

A linker matches function calls with the location of the executable code for those funcs.

```
Disassembly of function sum in binary file prog
0000000000400540 <multstore>:
  400540:  53                   push   %rbx
  400541:  48 89 d3             mov    %rdx,%rbx
  400544:  e8 42 00 00 00       callq  40058b <mult2>
  400549:  48 89 03             mov    %rax,(%rbx)
  40054c:  5b                   pop    %rbx
  40054d:  c3                   retq
  40054e:  90                   nop
  40054f:  90                   nop
```

function

Inserted for grow the function to 16 bytes for a better memory replacement.

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8

Figure 3.1 Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.

In later sections, we will present a number of instructions for copying and generating 1-, 2-, 4-, and 8-byte values. When these instructions have registers as destinations, two conventions arise for what happens to the remaining bytes in the register for instructions that generate less than 8 bytes: Those that generate 1- or 2-byte quantities leave the remaining bytes unchanged. Those that generate 4-byte quantities set the upper 4 bytes of the register to zero. The latter convention was adopted as part of the expansion from IA32 to x86-64.

There is some instructions that copies and generates 1-, 2-, 4-, 8- byte values. When these registers have registers as instructions, two conventions for what happens to the remaining bytes in the register for instructions that generate less than 8 byte

generate 1-, or, 2- bytes → leave the remaining bytes unchanged

generate 4 bytes → set the upper 4 bytes of the register to zero

% rsp indicates the end position in the run-time stack

constant values written with $

Type	Form	Operand value	Name
Immediate	Imm	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

mov instructions mov S, D

movb, movw, movl, movq

exception: if movl has a register as the destination

↳ it will set the high-order 4 bytes to zero

```
1    movl $0x4050,%eax          Immediate--Register, 4 bytes
2    movw %bp,%sp               Register--Register,  2 bytes
3    movb (%rdi,%rcx),%al       Memory--Register,    1 byte
4    movb $-17,(%esp)           Immediate--Memory,   1 byte
5    movq %rax,-12(%rbp)        Register--Memory,    8 bytes
```

exception: movq Imm, D

32-bit two's complement numbers

diate data. The regular `movq` instruction can only have immediate source operands that can be represented as 32-bit two's-complement numbers. This value is then sign extended to produce the 64-bit value for the destination. The `movabsq` instruction can have an arbitrary 64-bit immediate value as its source operand and can only have a register as a destination.

movabsq 64-bit Imm, Reg

movq 64bit Imm, Reg X

movabsq Imm, Reg ✓

movq Reg, Mem ✓

```
1    movabsq $0x0011223344556677, %rax    %rax = 0011223344556677
2    movb    $-1, %al   - 1 byte          %rax = 00112233445566FF
3    movw    $-1, %ax   - 2 byte          %rax = 0011223344556FFF
4    movl    $-1, %eax  - 4 byte          %rax = 00000000FFFFFFFF
5    movq    $-1, %rax  - 8 byte          %rax = FFFFFFFFFFFFFFFF
```

64-bit imm.

$-1_{10} = 111\ldots1_{2\text{'s}}$

movl has a register as the destination

movz & movs

movz → fill out the remaining high-order-bits with 0

movs → fill out the remaining high-order-bits with the MSB of the source operand

only for the cases where Destination is larger than the source

Instruction	Effect	Description
MOVZ S, R	R ← ZeroExtend(S)	Move with zero extension
movzbw		Move zero-extended byte to word
movzbl		Move zero-extended byte to double word
movzwl		Move zero-extended word to double word
movzbq		Move zero-extended byte to quad word
movzwq		Move zero-extended word to quad word

Figure 3.5 Zero-extending data movement instructions. These instructions have a register or memory location as the source and a register as the destination.

↳ movzlq yok → movl S, D(reg) kullan

Instruction	Effect	Description
MOVS S, R	R ← SignExtend(S)	Move with sign extension
movsbw		Move sign-extended byte to word
movsbl		Move sign-extended byte to double word
movswl		Move sign-extended word to double word
movsbq		Move sign-extended byte to quad word
movswq		Move sign-extended word to quad word
movslq		Move sign-extended double word to quad word
cltq	%rax ← SignExtend(%eax)	Sign-extend %eax to %rax

Figure 3.6 Sign-extending data movement instructions. The MOVS instructions have a register or memory location as the source and a register as the destination. The `cltq` instruction is specific to registers %eax and %rax.

Such is not allowed → movb $0xF, (%ebx) → can not use %ebx as address register

Recall: When performing a cast that involves both a size change and a change of signedness in C, the operation should change the size first.

push & pop instructions

Instruction	Effect	Description
pushq S	R[%rsp] ← R[%rsp] − 8; M[R[%rsp]] ← S	Push quad word
popq D	D ← M[R[%rsp]]; R[%rsp] ← R[%rsp] + 8	Pop quad word

Figure 3.8 **Push and pop instructions.**

Arithmetic and Logical Operations

Instruction		Effect	Description
leaq	S, D	D ← &S	Load effective address
INC	D	D ← D+1	Increment
DEC	D	D ← D−1	Decrement
NEG	D	D ← −D	Negate
NOT	D	D ← ~D	Complement
ADD	S, D	D ← $D + S$	Add
SUB	S, D	D ← $D - S$	Subtract
IMUL	S, D	D ← $D * S$	Multiply
XOR	S, D	D ← D ^ S	Exclusive-or
OR	S, D	D ← D | S	Or
AND	S, D	D ← D & S	And
SAL	k, D	D ← $D << k$	Left shift
SHL	k, D	D ← $D << k$	Left shift (same as SAL)
SAR	k, D	D ← $D >>_A k$	Arithmetic right shift
SHR	k, D	D ← $D >>_L k$	Logical right shift

Figure 3.10 **Integer arithmetic operations.** The load effective address (leaq) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $>>_A$ and $>>_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

leaq 7(%rdx,%rdx,4), %rax
(rdx: x) → 5x+7

For the leaq, the destination operand must be a register.

leaq calculates the effective address and stores that, but movq does → M[effective address]

subq %rax, %rdx
↳ R[%rdx] − R[%rax]

The two operands cannot both be Memory loc.

When the second operand is a memory location
↳ the processor must read the value from memory
↳ perform the operation
↳ write the result back to memory

Shift operations

SAL k, D
SHL k, D → shift left

SAR k, D → arithmetic shift right → fill with copies of the sign bit

SHR k, D → logical shift right → fill with zero

k: shift amount
↳ could be either immediate
↳ or single-byte %cl register

%cl, where $2^m = w$. The higher-order bits are ignored. So, for example, when register %cl has hexadecimal value 0xFF, then instruction salb would shift by 7, while salw would shift by 15, sall would shift by 31, and salq would shift by 63.

→ 11111111

imulq , mulq

Instruction		Effect	Description
imulq	S	R[%rdx]:R[%rax] ← $S \times$ R[%rax]	Signed full multiply
mulq	S	R[%rdx]:R[%rax] ← $S \times$ R[%rax]	Unsigned full multiply
cqto		R[%rdx]:R[%rax] ← SignExtend(R[%rax])	Convert to oct word
idivq	S	R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S	Signed divide → quotient in %rax, remainder in %rdx
divq	S	R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S	Unsigned divide

Figure 3.12 Special arithmetic operations. These operations provide full 128-bit multiplication and division, for both signed and unsigned numbers. The pair of registers %rdx and %rax are viewed as forming a single 128-bit oct word.

↳ One argument to be multiplied must be in the %rax

↳ S will take the other argument to be multiplied

↳ The result will be stored in %rdx (high-order 64 bits) and in %rax (low-order 64 bits)

```
    void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
    dest in %rdi, x in %rsi, y in %rdx
1   store_uprod:
2     movq    %rsi, %rax      Copy x to multiplicand
3     mulq    %rdx            Multiply by y
4     movq    %rax, (%rdi)    Store lower 8 bytes at dest
5     movq    %rdx, 8(%rdi)   Store upper 8 bytes at dest+8
6     ret
```

→ little endian

Condition Code Registers

Most recent arithmetic or logical operation determines them.

CF: Carry flag

Overflow for unsigned operation

ZF: Zero flag

SF: Sign flag

Most recent operation yielded a negative value

OF: Overflow flag

Two's complement overflow

t `t = a+b`, where variables `a`, `b`, and `t` are integers.

CF	`(unsigned) t < (unsigned) a`	Unsigned overflow
ZF	`(t == 0)`	Zero
SF	`(t < 0)`	Negative
OF	`(a < 0 == b < 0) && (t < 0 != a < 0)`	Signed overflow

leaq does not alter condition codes.

Logical operations (XOR, AND, ...)

↳ CF = 0 , OF = 0

Shift operations

↳ CF = last bit shifted out

↳ OF = 0

INC , DEC

↳ OF = 1 , ZF = 1 , CF = unchanged

cmp instructions

Set the condition codes according to the difference of their two operands. without updating their destinations.

Instruction		Based on	Description
CMP	S_1, S_2	$S_2 - S_1$	Compare
`cmpb`			Compare byte
`cmpw`			Compare word
`cmpl`			Compare double word
`cmpq`			Compare quad word

cmp sets the ZF = 0 if two operands are equal.

Test instructions behave in the same manner as AND, but does not alter destination. Rather, sets the condition codes.

↳ testq %rax, %rax to see whether %rax is negative, zero, or positive

TEST	S_1, S_2	$S_1 \& S_2$	Test
`testb`			Test byte
`testw`			Test word
`testl`			Test double word
`testq`			Test quad word

negative olduğunu nasıl anlıyoruz ⇒

SF	ZF	
0	0	a > 0
0	1	a == 0
1	0	a < 0
1	1	Nonsense

mesela a < 0 ⇒ $1101_2 = -3$

$$\begin{array}{r} 1101 \\ \& \; 1101 \\ \hline 1101 \end{array}$$

SET Instructions

Instruction		Synonym	Effect	Set condition
sete	*D*	setz	*D* ← ZF	Equal / zero
setne	*D*	setnz	*D* ← ~ZF	Not equal / not zero
sets	*D*		*D* ← SF	Negative
setns	*D*		*D* ← ~SF	Nonnegative
setg	*D*	setnle	*D* ← ~(SF ^ OF) & ~ZF	Greater (signed >)
setge	*D*	setnl	*D* ← ~(SF ^ OF)	Greater or equal (signed >=)
setl	*D*	setnge	*D* ← SF ^ OF	Less (signed <)
setle	*D*	setng	*D* ← (SF ^ OF) | ZF	Less or equal (signed <=)
seta	*D*	setnbe	*D* ← ~CF & ~ZF	Above (unsigned >)
setae	*D*	setnb	*D* ← ~CF	Above or equal (unsigned >=)
setb	*D*	setnae	*D* ← CF	Below (unsigned <)
setbe	*D*	setna	*D* ← CF | ZF	Below or equal (unsigned <=)

Figure 3.14 The SET instructions. Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," that is, alternate names for the same machine instruction.

Jump Instructions

↳ can cause the execution to switch to a completely new position in the progr.

↳ jump destinations are indicated by a label in the assembly code

Instruction		Synonym	Jump condition	Description
jmp	*Label*		1	Direct jump
jmp	**Operand*		1	Indirect jump
je	*Label*	jz	ZF	Equal / zero
jne	*Label*	jnz	~ZF	Not equal / not zero
js	*Label*		SF	Negative
jns	*Label*		~SF	Nonnegative
jg	*Label*	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge	*Label*	jnl	~(SF ^ OF)	Greater or equal (signed >=)
jl	*Label*	jnge	SF ^ OF	Less (signed <)
jle	*Label*	jng	(SF ^ OF) | ZF	Less or equal (signed <=)
ja	*Label*	jnbe	~CF & ~ZF	Above (unsigned >)
jae	*Label*	jnb	~CF	Above or equal (unsigned >=)
jb	*Label*	jnae	CF	Below (unsigned <)
jbe	*Label*	jna	CF | ZF	Below or equal (unsigned <=)

jump target is read from memory location or a register

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Direct jump → jmp .L1

Indirect jump → jmp *%rax read the jump target from register

Indirect jump → jmp *(%rax) read the jump target from memory

Conditional jumps can only be direct.

Jump instructions encodings

↳ They encode the difference between the address of the target instruction

↳ Or use an absolute address to directly specify the target

The assembler and the linker select the appropriate encodings for D.

```
1      0:   48 89 f8              mov    %rdi,%rax
2      3:   eb 03                 jmp    8 <loop+0x8>
3      5:   48 d1 f8              sar    %rax
4      8:   48 85 c0              test   %rax,%rax
5      b:   7f f8                 jg     5 <loop+0x5>
6      d:   f3 c3                 repz retq
```

0x03 + 0x5 = 0x8

0xd + 0xf8 = 0x5

0xf8: single byte ⇓ −8

The following shows the disassembled version of the program after linking:

jump target is found in this way: → use single byte two's complementation

```
1   4004d0:  48 89 f8              mov    %rdi,%rax
2   4004d3:  eb 03                 jmp    4004d8 <loop+0x8>
3   4004d5:  48 d1 f8              sar    %rax
4   4004d8:  48 85 c0              test   %rax,%rax
5   4004db:  7f f8                 jg     4004d5 <loop+0x5>
6   4004dd:  f3 c3                 repz retq
```

03 → add this to the next instruction

1111 1000 ≡ −8 ⇒

if-else

```
t = test-expr;
if (!t)
    goto false;
then-statement
goto done;
false:
    else-statement
done:
```

```
t = test-expr;
if (t)
    goto true;
else-statement
goto done;
true:
    then-statement
done:
```

cmov – Conditional move Instructions

Instruction		Synonym	Move condition	Description
cmove	*S*, *R*	cmovz	ZF	Equal / zero
cmovne	*S*, *R*	cmovnz	~ZF	Not equal / not zero
cmovs	*S*, *R*		SF	Negative
cmovns	*S*, *R*		~SF	Nonnegative
cmovg	*S*, *R*	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge	*S*, *R*	cmovnl	~(SF ^ OF)	Greater or equal (signed >=)
cmovl	*S*, *R*	cmovnge	SF ^ OF	Less (signed <)
cmovle	*S*, *R*	cmovng	(SF ^ OF) | ZF	Less or equal (signed <=)
cmova	*S*, *R*	cmovnbe	~CF & ~ZF	Above (unsigned >)
cmovae	*S*, *R*	cmovnb	~CF	Above or equal (Unsigned >=)
cmovb	*S*, *R*	cmovnae	CF	Below (unsigned <)
cmovbe	*S*, *R*	cmovna	CF | ZF	Below or equal (unsigned <=)

Figure 3.18 The conditional move instructions. These instructions copy the source value *S* to its destination *R* when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

v = test-expr ? then-exp : else-exp ;

using conditional control transfer

```
    if (!test-expr)
        goto false;
    v = then-expr;
    goto done;
false:
    v = else-expr;
done:
```

using conditional move

```
v  = then-expr;
ve = else-expr;
t  = test-expr;
if (!t) v = ve;
```

Do-while loops

```
do
    body-statement
while (test-expr);
```

goto form

```
loop:
    body-statement
    t = test-expr;
    if (t)
        goto loop;
```

While Loops

```
while (test-expr)
    body-statement
```

GCC optimization with -Og required

jump-to-middle

```
    goto test;
loop:
```

```
    short loop_while(short a, short b)
    a in %rdi, b in %rsi
1   loop_while:
2     movl    $0, %eax    result=0
3     jmp     .L2         jump to test
4   .L3:
5     leaq    (,%rsi,%rdi), %rdx
6     addq    %rdx, %rax
7     subq    $1, %rdi
8   .L2:
9     cmpq    %rsi, %rdi      while (a>b)
10    jg      .L3
11    rep; ret
```

```
    body-statement
test:
    t = test-expr;
    if (t)
        goto loop;
```

GCC optimization with -O1 required

guarded do

→ while loop is turned into a do-while loop first

```
    t = test-expr;
    if (!t)
        goto done;
do
    body-statement
    while (test-expr);
done:
```

goto code

```
    t = test-expr;
    if (!t)
        goto done;
loop:
    body-statement
    t = test-expr;
    if (t)
        goto loop;
done:
```

For Loops

```
for (init-expr; test-expr; update-expr)
    body-statement
```

while format

```
init-expr;
while (test-expr) {
    body-statement
    update-expr;
}
```

jump-to-middle strategy yields the goto cod

```
    init-expr;
    goto test;
loop:
    body-statement
    update-expr;
test:
    t = test-expr;
    if (t)
        goto loop;
```

while the guarded-do strategy yields

```
    init-expr;
    t = test-expr;
    if (!t)
        goto done;
loop:
    body-statement
    update-expr;
    t = test-expr;
    if (t)
        goto loop;
done:
```

```
/* Example of for loop containing a continue statement */
/* Sum even numbers between 0 and 9 */
long sum = 0;
long i;
for (i = 0; i < 10; i++) {
    if (i & 1)
        continue;
    sum += i;
}
```

inside for loop continue means that

Switch Statements

uses jump table

Jump table

↳ is an array where entry i is the address of a code segment implementing the action the program should take when the switch index equals i.

Time taken to execute a switch statement is independent of the number of conditions (switch statements)

Jump tables are used when there are a number of cases (four or more) they span a small range of values.

↳ Mesela n == 100, 102, 104, 106 için switch caselerin kullanıyor.

n ∈ [100, 106] or n ∉ [100, 106]

her bir caseden 100 çıkarıp n-100 'ü bu caseler arasında sayalım

n-100 ∈ [0, 6] or n-100 ∉ [0, 6] → we created index

↳ unsigned

(a) Switch statement

```
void switch_eg(long x, long n,
               long *dest)
{
    long val = x;

    switch (n) {

    case 100:
        val *= 13;
        break;

    case 102:
        val += 10;
        /* Fall through */

    case 103:
        val += 11;
        break;

    case 104:
    case 106:
        val *= val;
        break;

    default:
        val = 0;
    }
    *dest = val;
}
```

(b) Translation into extended C

```
1  void switch_eg_impl(long x, long n,
2                      long *dest)
3  {
4      /* Table of code pointers */
5      static void *jt[7] = {
6          &&loc_A, &&loc_def, &&loc_B,
7          &&loc_C, &&loc_D, &&loc_def,
8          &&loc_D
9      };
10     unsigned long index = n - 100;
11     long val;
12
13     if (index > 6)
14         goto loc_def;
15     /* Multiway branch */
16     goto *jt[index];
17
18  loc_A:    /* Case 100 */
19     val = x * 13;
20     goto done;
21  loc_B:    /* Case 102 */
22     x = x + 10;
23     /* Fall through */
24  loc_C:    /* Case 103 */
25     val = x + 11;
26     goto done;
27  loc_D:    /* Cases 104, 106 */
28     val = x * x;
29     goto done;
30  loc_def:  /* Default case */
31     val = 0;
32  done:
33     *dest = val;
34 }
```

```
    void switch_eg(long x, long n, long *dest)
    x in %rdi, n in %rsi, dest in %rdx
1   switch_eg:
2     subq    $100, %rsi            Compute index = n-100
3     cmpq    $6, %rsi              Compare index:6
4     ja      .L8                   If >, goto loc_def
5     jmp     *.L4(,%rsi,8)         Goto *jg[index]
6   .L3:                          loc_A:
7     leaq    (%rdi,%rdi,2), %rax     3*x
8     leaq    (%rdi,%rax,4), %rdi     val = 13*x
9     jmp     .L2                   Goto done
10  .L5:                          loc_B:
11    addq    $10, %rdi               x = x + 10
12  .L6:                          loc_C:
13    addq    $11, %rdi               val = x + 11
14    jmp     .L2                   Goto done
15  .L7:                          loc_D:
16    imulq   %rdi, %rdi              val = x * x
17    jmp     .L2                   Goto done
18  .L8:                          loc_def:
19    movl    $0, %edi                val = 0
20  .L2:                          done:
21    movq    %rdi, (%rdx)            *dest = val
22    ret                           Return
```

Figure 3.23 Assembly code for `switch` statement example in Figure 3.22.

```
1     .section        .rodata
2     .align 8          Align address to multiple of 8
3   .L4:
4     .quad   .L3       Case 100: loc_A
5     .quad   .L8       Case 101: loc_def
6     .quad   .L5       Case 102: loc_B
7     .quad   .L6       Case 103: loc_C
8     .quad   .L7       Case 104: loc_D
9     .quad   .L8       Case 105: loc_def
10    .quad   .L7       Case 106: loc_D
```

n=104

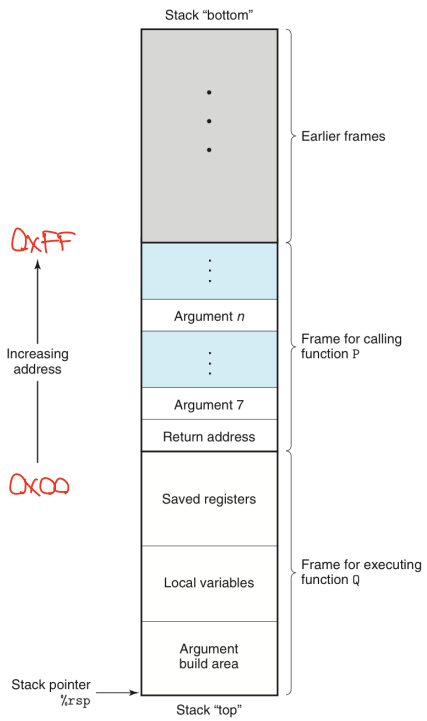
n=106

Handles duplicate cases by having the same code labels (i.e. entries 4 and 6)

Handles missing cases by using the label for the default case (i.e. entries 5 and 7)

3.6 PROCEDURES

Run-time stack



Stack's growth direction ⇒ 0xFF ⟶ 0x00

When procedure P calls procedure Q, it will push the return address onto the stack, indicating where within P the program should resume execution once Q returns.

Procedure P can pass up to six integral values
↳ (i.e pointers and integers) on the stack, but if Q requires more arguments, these can be stored by P within its stack frame prior to the call.

CONTROL TRANSFER

When P calls Q

↳ Passing control from function P to Q

call q

1. Push return address A to the stack
2. Set the PC to the starting address of the code for Q

ret

1. Pop the address A off the stack
2. Set PC to A

Instruction	Description
`call` *Label*	Procedure call
`call` **Operand*	Procedure call
`ret`	Return from call

can be either direct or indirect

(a) Disassembled code for demonstrating procedure calls and returns

```
     Disassembly of leaf(long y)
     y in %rdi
1    0000000000400540 <leaf>:
2      400540:  48 8d 47 02           lea    0x2(%rdi),%rax    L1: z+2
3      400544:  c3                    retq                     L2: Return

4    0000000000400545 <top>:
     Disassembly of top(long x)
     x in %rdi
5      400545:  48 83 ef 05           sub    $0x5,%rdi         T1: x-5
6      400549:  e8 f2 ff ff ff        callq  400540 <leaf>     T2: Call leaf(x-5)
7      40054e:  48 01 c0              add    %rax,%rax         T3: Double result
8      400551:  c3                    retq                     T4: Return

     . . .
      Call to top from function main
9      40055b:  e8 e5 ff ff ff        callq  400545 <top>      M1: Call top(100)
10     400560:  48 89 c2              mov    %rax,%rdx         M2: Resume
```

(b) Execution trace of example code

Instruction			State values (at beginning)				
Label	PC	Instruction	%rdi	%rax	%rsp	*%rsp	Description
M1	0x40055b	callq	100	—	0x7fffffffe820	—	Call top(100)
T1	0x400545	sub	100	—	0x7fffffffe818	0x400560	Entry of top
T2	0x400549	callq	95	—	0x7fffffffe818	0x400560	Call leaf(95)
L1	0x400540	lea	95	—	0x7fffffffe810	0x40054e	Entry of leaf
L2	0x400544	retq	—	97	0x7fffffffe810	0x40054e	Return 97 from leaf
T3	0x40054e	add	—	97	0x7fffffffe818	0x400560	Resume top
T4	0x400551	retq	—	194	0x7fffffffe818	0x400560	Return 194 from top
					0x7fffffffe820	—	Resume main

ving procedure calls and returns. Using the stack to he right point in the procedures.

is 32 bits, it can be accessed as `%edi`.

When a function has more than six integral arguments, the other ones are passed on the stack. Assume that procedure `P` calls procedure `Q` with n integral arguments, such that $n > 6$. Then the code for `P` must allocate a stack frame with enough storage for arguments 7 through n, as illustrated in Figure 3.25. It copies arguments 1–6 into the appropriate registers, and it puts arguments 7 through n onto the stack, with argument 7 at the top of the stack. When passing parameters on the stack, all data sizes are rounded up to be multiples of eight. With the arguments in place, the program can then execute a `call` instruction to transfer

DATA TRANSFER

When a function has more than 6 arguments, the other ones (from 7 to N) are passed on the stack.

P calls Q which has N arguments (N > 6)

↳ It copies arguments 1-6 into the appropriate registes

↳ it puts arguments 7 through n onto the stack

with argument 7 at the top of the stack

When passing parameters on the stack, all data sizes are rounded up to be multiples of eight.

(a) C code for calling function

```
long call_proc()
{
    long  x1 = 1; int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

(b) Generated assembly code

```
     long call_proc()
1    call_proc:
       Set up arguments to proc
2      subq    $32, %rsp          Allocate 32-byte stack frame
3      movq    $1, 24(%rsp)       Store 1 in &x1
4      movl    $2, 20(%rsp)       Store 2 in &x2
5      movw    $3, 18(%rsp)       Store 3 in &x3
6      movb    $4, 17(%rsp)       Store 4 in &x4
7      leaq    17(%rsp), %rax     Create &x4
8      movq    %rax, 8(%rsp)      Store &x4 as argument 8
9      movl    $4, (%rsp)         Store 4 as argument 7
10     leaq    18(%rsp), %r9      Pass &x3 as argument 6
11     movl    $3, %r8d           Pass 3 as argument 5
12     leaq    20(%rsp), %rcx     Pass &x2 as argument 4
13     movl    $2, %edx           Pass 2 as argument 3
14     leaq    24(%rsp), %rsi     Pass &x1 as argument 2
15     movl    $1, %edi           Pass 1 as argument 1
       Call proc
16     call    proc
       Retrieve changes to memory
17     movslq  20(%rsp), %rdx     Get x2 and convert to long
18     addq    24(%rsp), %rdx     Compute x1+x2
19     movswl  18(%rsp), %eax     Get x3 and convert to int
20     movsbl  17(%rsp), %ecx     Get x4 and convert to int
21     subl    %ecx, %eax         Compute x3-x4
22     cltq                       Convert to long
23     imulq   %rdx, %rax         Compute (x1+x2) * (x3-x4)
24     addq    $32, %rsp          Deallocate stack frame
25     ret                        Return
```

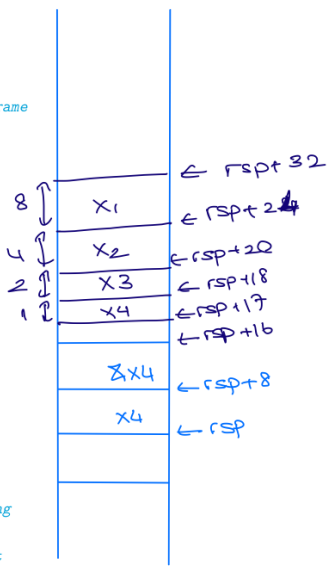


Figure 3.32 Example of code to call function `proc`, defined in Figure 3.29. This code creates a stack frame.

LOCAL STORAGE IN REGISTERS

- Registers %rbx, %rbp and %r12 – %r15 are calle saved registers.
  - ↳ When procedure P calls procedure Q, Q must preserve the values of these registers.
    - ↳ Either not changing them OR push on the stack and pop back before return.
- All other registers, except for the stack pointer %rsp are caller saved registers.
  - ↳ They can be modified by any function.
- Local variables are stored in callee-saved registers or stack, in general.

ARRAY ALLOCATION AND ACCESS

Let us denote the starting location as x_A. The declaration has two effects. First, it allocates a contiguous region of $L \cdot N$ bytes in memory, where L is the size (in bytes) of data type T. Second, it introduces an identifier `A` that can be used as a pointer to the beginning of the array. The value of this pointer will be x_A. The array elements can be accessed using an integer index ranging between 0 and $N-1$. Array element i will be stored at address $x_A + L \cdot i$.

As examples, consider the following declarations:

```
char    A[12];
char   *B[8];
int     C[6];
double *D[5];
```

These declarations will generate arrays with the following parameters:

Array	Element size	Total size	Start address	Element i
A	1	12	x_A	$x_A + i$
B	8	64	x_B	$x_B + 8i$
C	4	24	x_C	$x_C + 4i$
D	8	40	x_D	$x_D + 8i$

The memory referencing instructions of x86-64 are designed to simplify array access. For example, suppose `E` is an array of values of type `int` and we wish to evaluate `E[i]`, where the address of `E` is stored in register `%rdx` and i is stored in register `%rcx`. Then the instruction

```
movl (%rdx,%rcx,4),%eax
```

will perform the address computation $x_E + 4i$, read that memory location, and copy the result to register `%eax`. The allowed scaling factors of 1, 2, 4, and 8 cover the sizes of the common primitive data types.

POINTER ARITHMETIC

If p is a pointer of type T, and the value of p is x_p,
↳ then p+i has value $x_p + L*i$ where L is the size of type T.

A[i] ≡ *(A+i)

Array → E → %rdx
index → i → %rcx

Expression	Type	Value	Assembly Code
E	int *	x_E	movl %rdx, %rax
E[0]	int	M[x_E]	movl (%rdx), %eax
E[i]	int	M[x_E+4i]	movl (%rdx,%rcx,4), %eax
&E[2]	int *	x_E+8	leaq 8(%rdx), %rax
E+i-1	int *	x_E+4i-4	leaq -4(%rdx,%rcx,4), %rax
*(E+i-3)	int	M[x_E+4i-12]	leaq -12(%rdx,%rcx,4), %eax
&E[i] - E	long	i	movq %rcx, %rax

Nested arrays

T D[R][C];

array element D[i][j] is at memory address

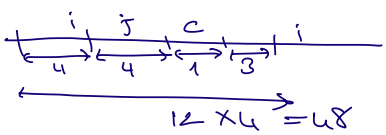
↳ &D[i][j] = X_D + L*(C*i + j)

Row	Element	Address
A[0]	A[0][0]	x_A
	A[0][1]	$x_A + 4$
	A[0][2]	$x_A + 8$
A[1]	A[1][0]	$x_A + 12$
	A[1][1]	$x_A + 16$
	A[1][2]	$x_A + 20$
A[2]	A[2][0]	$x_A + 24$
	A[2][1]	$x_A + 28$
	A[2][2]	$x_A + 32$
A[3]	A[3][0]	$x_A + 36$
	A[3][1]	$x_A + 40$
	A[4][2]	$x_A + 44$
A[4]	A[4][0]	$x_A + 48$
	A[4][1]	$x_A + 52$
	A[4][2]	$x_A + 56$

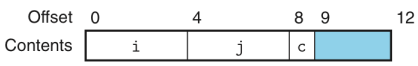
```
struct S2 {
    int  i;
    int  j;
    char c;
};
```

If we pack this structure into 9 bytes, we can still satisfy the alignment requirements for fields `i` and `j` by making sure that the starting address of the structure satisfies a 4-byte alignment requirement. Consider, however, the following declaration:

```
struct S2 d[4];
```



With the 9-byte allocation, it is not possible to satisfy the alignment requirement for each element of `d`, because these elements will have addresses x_d, $x_d + 9$, $x_d + 18$, and $x_d + 27$. Instead, the compiler allocates 12 bytes for structure `S2`, with the final 3 bytes being wasted space:



That way, the elements of `d` will have addresses x_d, $x_d + 12$, $x_d + 24$, and $x_d + 36$. As long as x_d is a multiple of 4, all of the alignment restrictions will be satisfied.

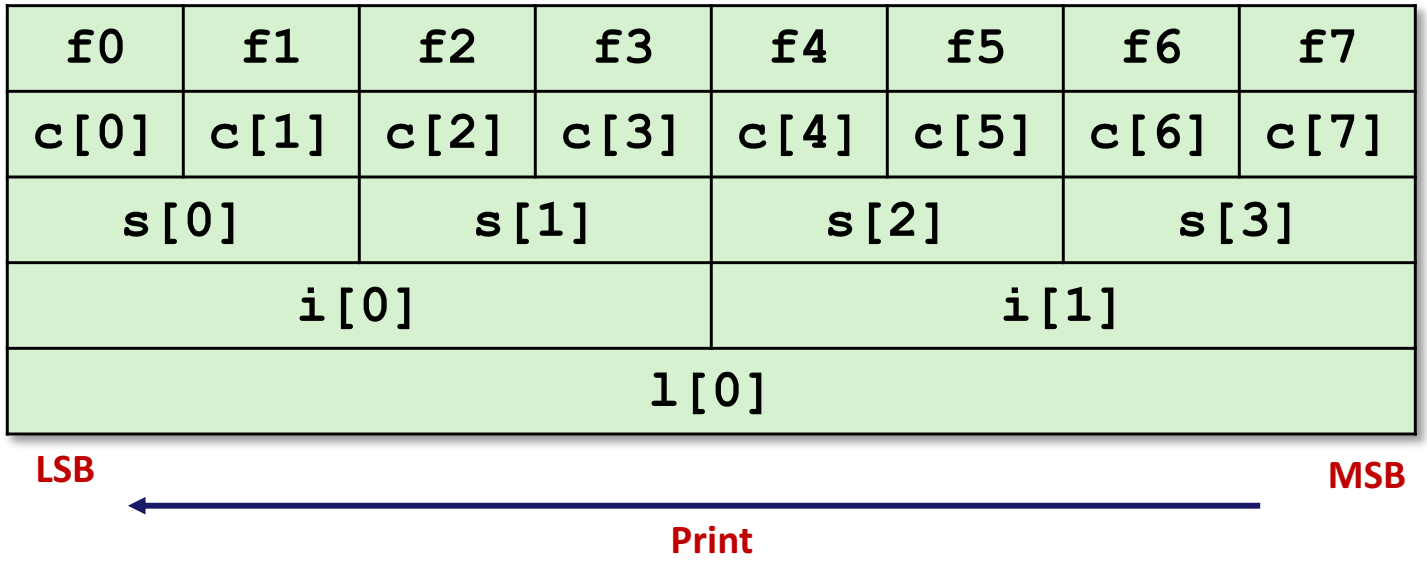
- **1 byte: `char`, ...**
  - no restrictions on address
- **2 bytes: `short`, ...**
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, ...**
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, `long`, `char *`, ...**
  - lowest 3 bits of address must be 000_2

→ multiple of 8 bytes ends with this

Byte Ordering on x86-64

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

Little Endian

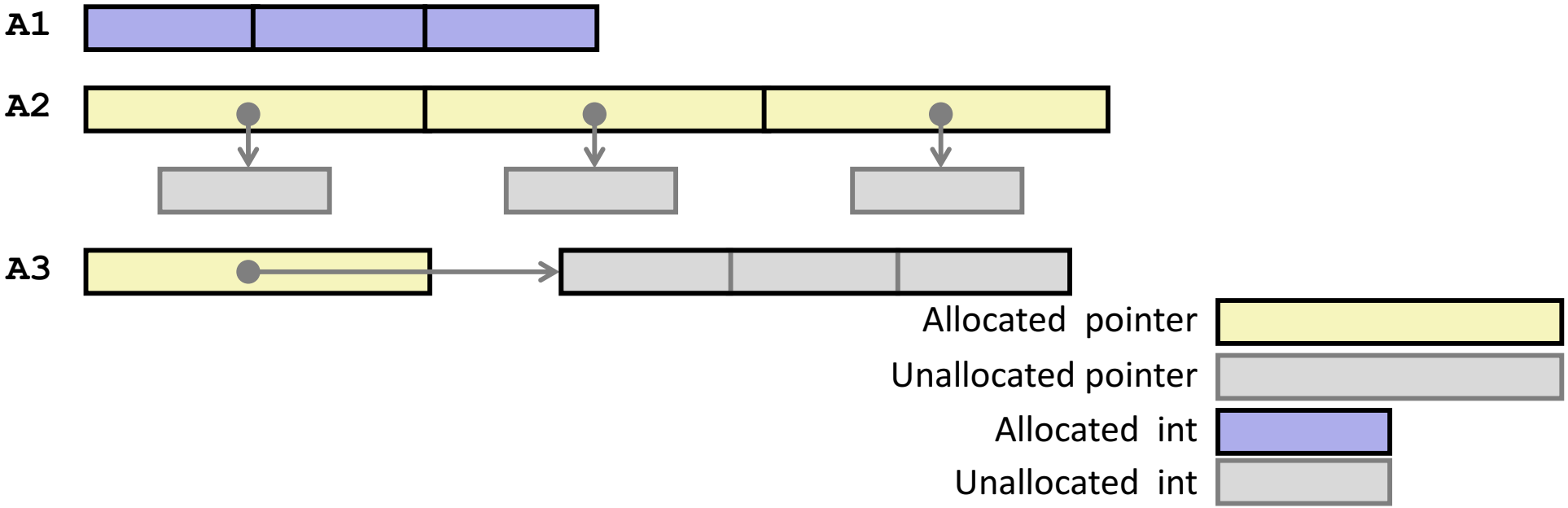


Output on x86-64:

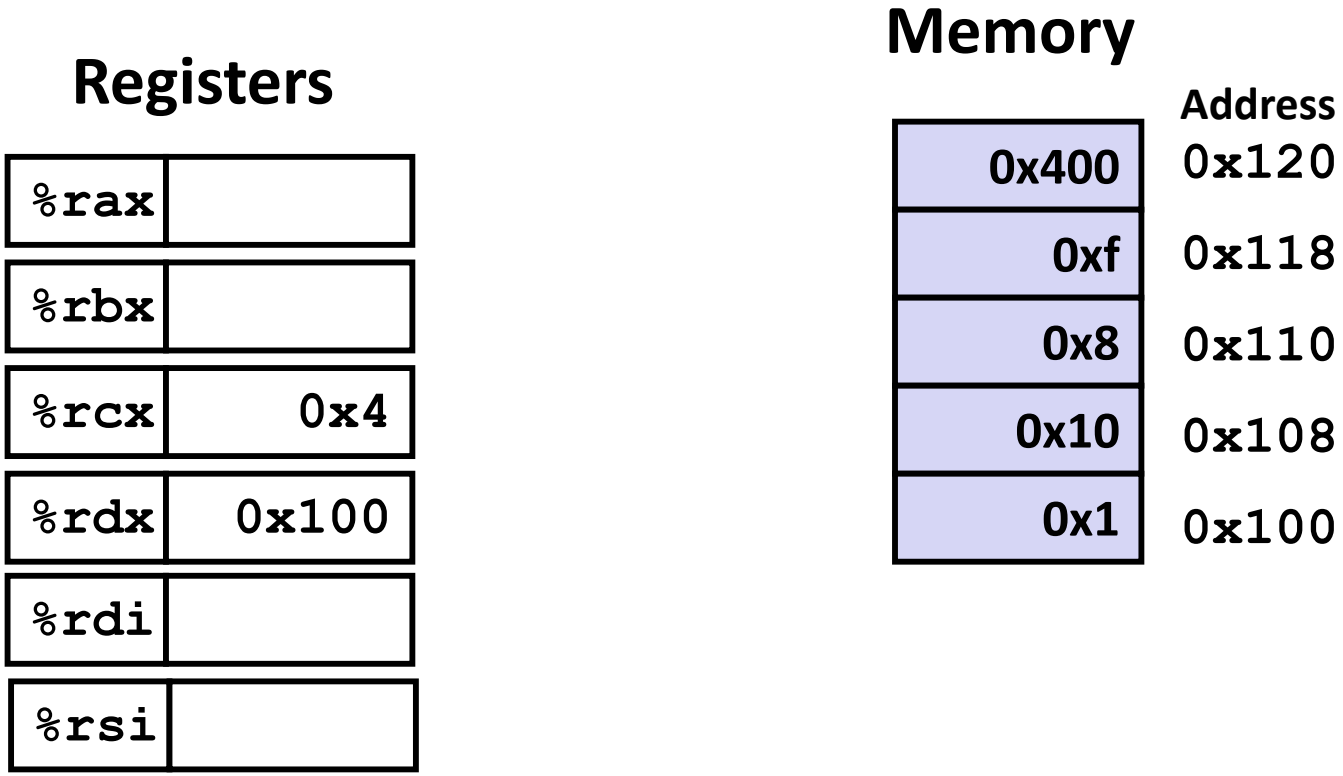
```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4	N	–	–
`int *A2[3]`	Y	N	24	Y	N	8	Y	Y	4
`int (*A3)[3]`	Y	N	8	Y	Y	12	Y	Y	4



leaq vs. movq example




```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

`leaq` vs. `movq` example (solution)

Registers

`%rax`	`0x110`
`%rbx`	`0x8`
`%rcx`	`0x4`
`%rdx`	`0x100`
`%rdi`	`0x100`
`%rsi`	`0x1`

Memory

	Address
0x400	0x120
0xf	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```