6.172 Performance Engineering of Software Systems

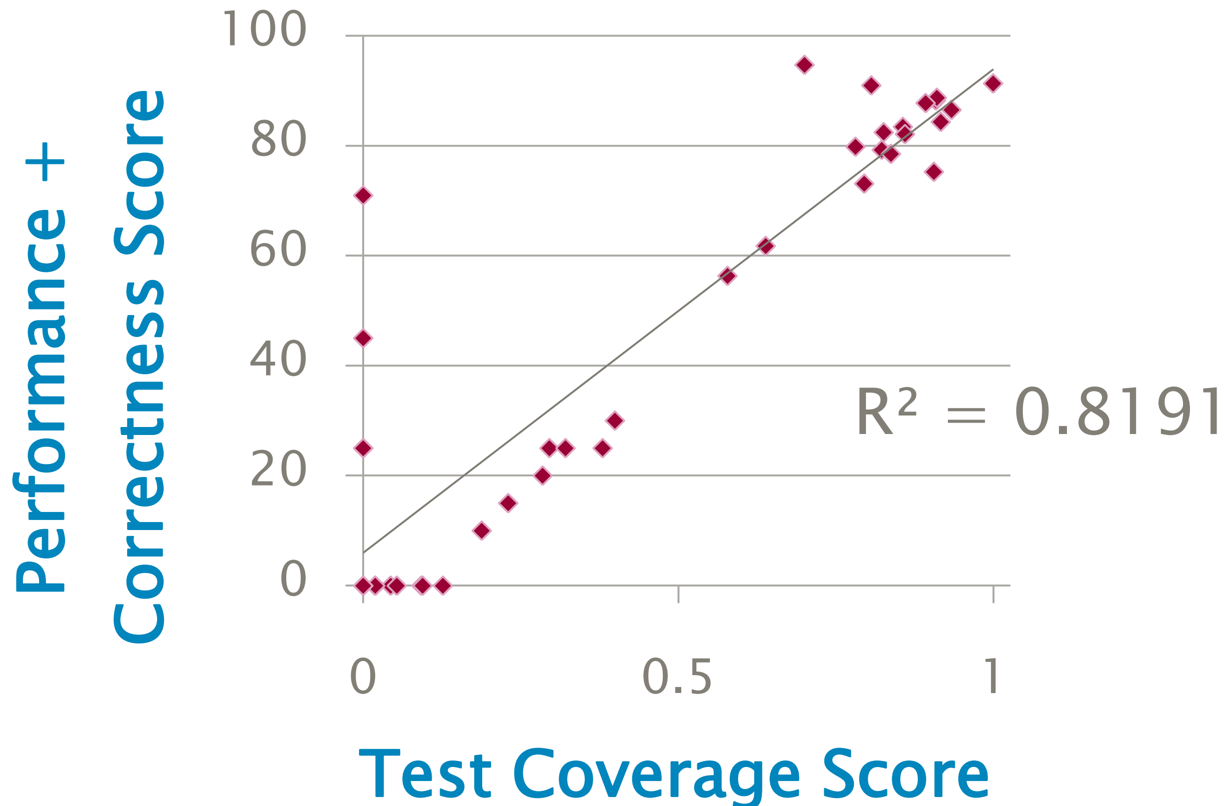
LECTURE 6
C to Assembler

Charles E. Leiserson

September 28, 2010

© 2010 Charles E. Leiserson

Everybit Beta Scores



Lesson 1: Before coding, write tests. A good regression suite speeds the development of fast correct code.

Lesson 2: Pair programming, not divide-and-conquer.

© 2010 Charles E. Leiserson and Saman Amarasinghe

Single-Threaded Performance

- Today's computing milieu: networks of multicore clusters
  - Shared memory among processors within a chip
  - Message passing among machines in a cluster
  - Network protocols among clusters
- Why study single-threaded performance?
  - Foundation of good performance is making single threads execute fast.
  - Lessons of single-threaded performance often generalize.

Generic Single-Threaded Machine

Processor Core

- Registers
- Functional units (arithmetic and logical operations)
- Floating-point units
- Vector units
- Instruction execution and coordination

Memory Hierarchy

- Registers
- L1-caches (instr & data)
- L2-cache
- L3-cache
- DRAM memory
- Solid-state drive
- Disk

© 2010 Charles E. Leiserson and Saman Amarasinghe

Source Code to Execution

Source code `fib.c`

```
uint64_t fib(uint64_t n) {
  if (n < 2) return n;
  return (fib(n-1) + fib(n-2));
}
```

```
% gcc fib.c -o fib
```

4 stages
- Preprocessing
- Compiling
- Assembling
- Linking

Machine code `fib`

```
01010101 01001000 10001001
11100101 01010011 01001000
10000011 11101100 00001000
10001001 01111101 11110100
10000011 01111101 11110100
00000001 01111111 00001000
10001011 01000101 11110100
10001001 01000101 11110000
11101011 00011101 10001011
01000101 11110100 10001101
01111000 11111111 11101000
11011011 11111111 11111111
11111111 10001001 11000011
10001011 01000101 11110100
10001101 01111000 11111110
11101000 11001110 11111111
11111111 11111111 00000001
11000011 10001001 01011101
11110000 10001011 01000101
11110000 01001000 10000011
11000100 00001000 01011011
11001001 11000011
```

Hardware interpretation

```
% ./fib
```

Execution

Source Code to Assembly Code

Source code fib.c

```
uint64_t fib(uint64_t n) {
  if (n < 2) return n;
  return (fib(n-1) + fib(n-2));
}
```

```
% gcc fib.c -g -S
```

Assembly language provides a convenient symbolic representation of machine language.

Assembly code fib.s

```
.globl fib
          .type     fib, @function
fib:
.LFB3:
          .loc 1 16 0
          pushq     %rbp
.LCFI3:
          movq      %rsp, %rbp
.LCFI4:
          pushq     %rbx
.LCFI5:
          subq      $24, %rsp
.LCFI6:
          movq      %rdi, -16(%rbp)
          .loc 1 17 0
          cmpq      $1, -16(%rbp)
          ja        .L4
          movq      -16(%rbp), %rax
          movl      %eax, -20(%rbp)
          jmp       .L5
.L4:
          .loc 1 18 0
          movq      -16(%rbp), %rax
          ...
```

See http://sourceware.org/binutils/docs/as/index.html.

© 2010 Charles E. Leiserson and Saman Amarasinghe

Disassembling

Source, machine, & assembly

Binary executable
fib with debug symbols

```
% objdump -S fib
```

```
uint64_t fib(uint64_t n) {
  4004f0: 55                   push   %rbp
  4004f1: 48 89 e5             mov    %rsp,%rbp
  4004f4: 53                   push   %rbx
  4004f5: 48 83 ec 18          sub    $0x18,%rsp
  4004f9: 48 89 7d f0          mov    %rdi,-0x10(%rbp
  if (n < 2) return n;
  4004fd: 48 83 7d f0 01       cmpq   $0x1,-0x10(%rbp)
  400502: 77 0a                ja     40050e <fib+0x1e>
  400504: 48 8b 45 f0          mov    -0x10(%rbp),%rax
  400508: 48 89 45 e8          mov    %rax,-0x18(%rbp)
  40050c: eb 24                jmp    400532 <fib+0x42>
  return (fib(n-1) + fib(n-2));
  40050e: 48 8b 45 f0          mov    -0x10(%rbp),%rax
  400512: 48 8d 78 ff          lea    -0x1(%rax),%rdi
  400516: e8 d5 ff ff ff       callq  4004f0 <fib>
  40051b: 48 89 c3             mov    %rax,%rbx
  40051e: 48 8b 45 f0          mov    -0x10(%rbp),%ra
  400522: 48 8d 78 fe          lea    -0x2(%rax),%rdi
  400526: e8 c5 ff ff ff       callq  4004f0 <fib>
  40052b: 48 01 c3             add    %rax,%rbx
  40052e: 48 89 5d e8          mov    %rbx,-0x18(%rbp
  400532: 48 8b 45 e8          mov    -0x18(%rbp),%rax
}
  400536: 48 83 c4 18          add    $0x18,%rsp
  40053a: 5b                   pop    %rbx
  40053b: c9                   leaveq
  40053c: c3                   retq
```

© 2010 Charles E. Leiserson and Saman Amarasinghe

Assembly Code to Executable

Assembly code

```
.globl fib
        .type   fib, @function
fib:
.LFB3:
        .loc 1 16 0
        pushq   %rbp
.LCFI3:
        movq    %rsp, %rbp
.LCFI4:
        pushq   %rbx
.LCFI5:
        subq    $24, %rsp
.LCFI6:
        movq    %rdi, -16(%rbp)
        .loc 1 17 0
        cmpq    $1, -16(%rbp)
        ja      .L4
        movq    -16(%rbp), %rax
        movl    %eax, -20(%rbp)
        jmp     .L5
.L4:
        .loc 1 18 0
        movq    -16(%rbp), %rax
        ...
```

```
% gcc fib.s -o fib
```

Machine code

```
01010101 01001000
10001001 11100101
01010011 01001000
10000011 11101100
00001000 10001001
01111101 11110100
10000011 01111101
11110100 00000001
01111111 00001000
10001011 01000101
11110100 10001001
01000101 11110000
11101011 00011101
10001011 01000101
11110100 10001101
01111000 11111111
11101000 11011011
11111111 11111111
11111111 10001001
11000011 10001011
01000101 11110100
10001101 01111000
11111110 11101000
...
```

You can edit `fib.s` in Emacs and assemble with gcc.

Expectations of Students

- Understand how a compiler implements C linguistic constructs using x86 instructions.
- Demonstrate a proficiency in reading x86 assembly language (with the aid of an architecture manual).
- Be able to make simple modifications to the x86 assembly language generated by a compiler.
- Know how to go about writing your own machine code from scratch if the situation demands it.

X86-64 Machine Model

- Flat 64-bit address space
- 16 64-bit general-purpose registers
- 6 16-bit segment registers
- 64-bit RFLAGS register
- 64-bit instruction pointer register (`%rip`)
- 8 80-bit floating-point data registers
- 16-bit control register
- 16-bit status register
- 11-bit opcode register
- 64-bit floating-point instruction pointer register
- 64-bit floating-point data pointer register
- 8 64-bit MMX registers
- 16 128-bit XMM registers (for SSE)
- 32-bit MXCSR register

© 2010 Charles E. Leiserson and Saman Amarasinghe

x86-64 General Registers

C linkage	63	31	15	7 0
Return value	%rax	%eax	%ax	%al
Callee saved	%rbx	%ebx	%bx	%bl
4th argument	%rcx	%ecx	%cx	%cl
3rd argument	%rdx	%edx	%dx	%dl
2nd argument	%rsi	%esi	%si	%sil
1st argument	%rdi	%edi	%di	%dil
Base pointer	%rbp	%ebp	%bp	%bpl
Stack pointer	%rsp	%esp	%sp	%spl
5th argument	%r8	%r8d	%r8w	%r8b
6th argument	%r9	%r9d	%r9w	%r9b
Callee saved	%r10	%r10d	%r10w	%r10b
For linking	%r11	%r11d	%r11w	%rllb
Unused for C	%r12	%r12d	%r12w	%r12b
Callee saved	%r13	%r13d	%r13w	%r13b
Callee saved	%r14	%r14d	%r14w	%r14b
Callee saved	%r15	%r15d	%r15w	%r15b

Also, the high-order bytes of %ax, %bx, %cx, and %dx are available as %ah, %bh, %ch, and %dh.

© 2010 Charles E. Leiserson and Saman Amarasinghe

x86-64 Data Types

C declaration	C constant	x86-64 size in bytes	Assembly suffix
char	'c'	1	b
short	L'cs'	2	w
int	172	4	l
unsigned	172U	4	l
long	172L	8	q
unsigned long	172UL	8	q
char *	"6.172"	8	q
float	6.172F	4	s
double	6.172	8	d
long double	6.172L	16(10)	t

Example: `movq -16(%rbp), %rax`

© 2010 Charles E. Leiserson and Saman Amarasinghe

Instruction Format

⟨opcode⟩ ⟨operand_list⟩

- ⟨opcode⟩ is a short mnemonic identifying the type of instruction with a single-character suffix indicating the data type.
  - If the suffix is missing, it can usually be inferred from the sizes of operand registers.
- ⟨operand_list⟩ is 0, 1, 2, or (rarely) 3 operands separated by commas.
  - One of the operands (the final operand in AT&T assembly format) is the destination.
  - The other operands are read-only (const).

© 2010 Charles E. Leiserson and Saman Amarasinghe

Assembler Directives

- Labels:

```
x: movq %rax, %rbx
```

- Storage directives:

```
x: .space 20    // allocate 20 bytes at location x
y: .long 172    // store constant 172L at y
z: .asciz "6.172"    // store string "6.172\0" at z
   .align 8    // advance loc ptr to multiple of 8
```

- Segment directives:

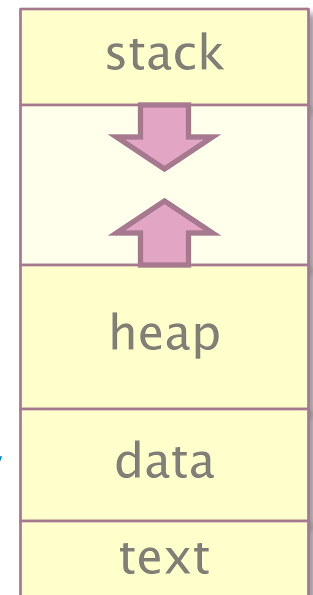
```
.text    // loc ptr in text segment
.data    // loc ptr in data segment
```

- Scope and linkage directives:

```
.globl fib  // make fib externally visible
```

See assembler manual.

Memory layout



x86-64 Opcode Examples

- **Data-transfer:** `mov`, `push`, `pop`, …
  - `movslq %eax, %rdx` (move sign extended)
  - **Careful:** Results of 32-bit operations are implicitly zero-extended to 64-bit values, unlike results of 8- and 16-bit operations.
- **Arithmetic and logical:** `add`, `sub`, `mult`, `and`, `or`, `not`, `cmp`, …
  - `subq %rdx, %rax (%rax = %rax - %rdx)`
- **Shift/rotate instructions:** `sar`, `sal`, …
- **Control transfer:** `call`, `ret`, `jmp`, `j`⟨condition⟩, …
- …

See http://siyobik.info/index.php?module=x86, but watch: 32-bit only and Intel syntax.

© 2010 Charles E. Leiserson and Saman Amarasinghe

X86-64 Addressing Modes

Only one operand may address memory.

- Register: `addq %rbx, %rax`
- Direct: `movq x, %rdi` //contents of x
- Immediate: `movq $x, %rdi` //address of x
- Register indirect: `movq %rbx, (%rax)`
- Register indexed: `movq $6, 172(%rax)`
- Base indexed scale displacement:
  - base and index are registers
  - scale is 2, 4, or 8 (absent implies 1)
  - displacement is 8-, 16-, or 32-bit value

  `addq 172(%rdi,%rdx,8), %rax`
- Instruction-pointer relative:
  `movq 6(%rip), %rax`

© 2010 Charles E. Leiserson and Saman Amarasinghe

Translating Expressions

```
uint64_t foo1()
{
  uint64_t x, y, z;
  x = 34; y = 7; z = 45;
  return (x + y) | z;
}

uint64_t foo2(uint64_t x,
              uint64_t y,
              uint64_t z)
{
  return (x + y) | z;
}

uint64_t x, y, z;

uint64_t foo3()
{
  return (x + y) | z;
}
```

```
foo1:
  movl      $45, %eax
  ret

foo2:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
  leaq   (%rsi,%rdi), %rax
  orq %rdx, %rax
  ret

foo3:
  movq   y(%rip), %rax
  addq   x(%rip), %rax
  orq z(%rip), %rax
  ret
```

Code depends on where x, y, and z are allocated!

© 2010 Charles E. Leiserson and Saman Amarasinghe

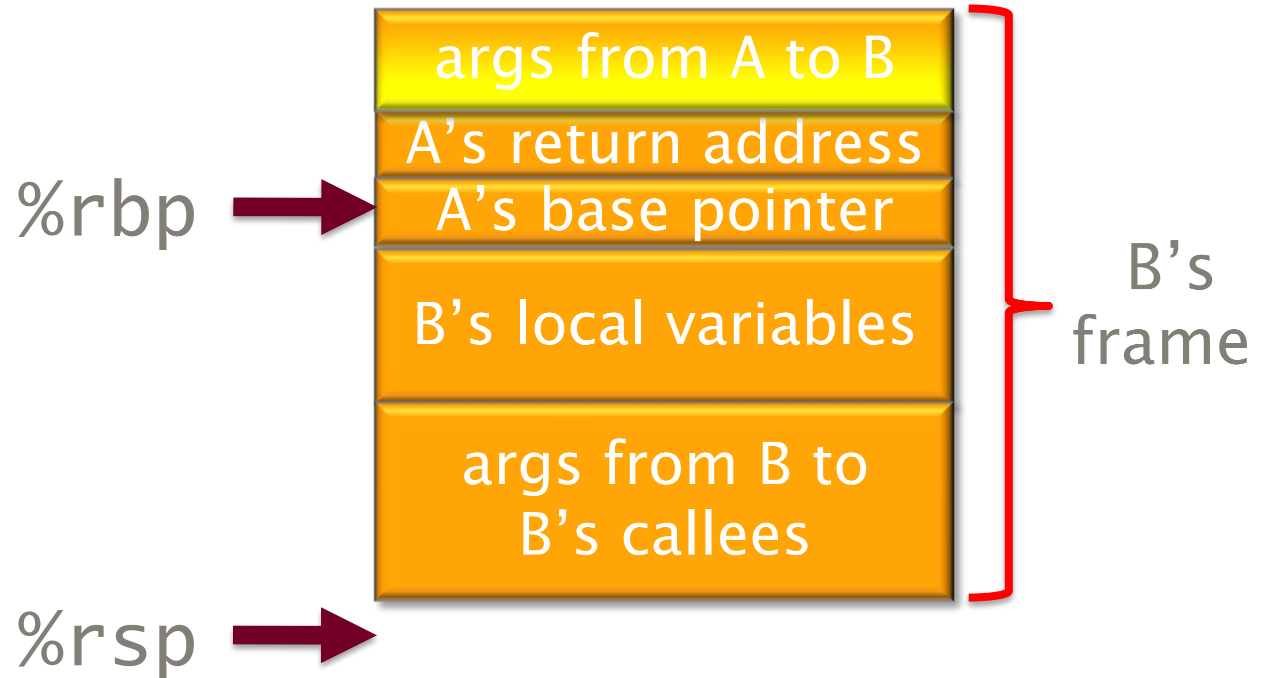
Linux x86-64 Calling Convention

- `%rsp` points to function-call stack in memory
  - stack grows downward in memory
  - `call` instruction pushes `%rip` on stack, jumps to call target operand (address of procedure)
  - `ret` instruction pops `%rip` from stack, returns to caller
- Software conventions
  - Caller-save registers (%r10, %r11)
  - Callee-save registers (%rbx, %rbp, %r12-%r15)

© 2010 Charles E. Leiserson and Saman Amarasinghe

GCC/Linux C Subroutine Linkage

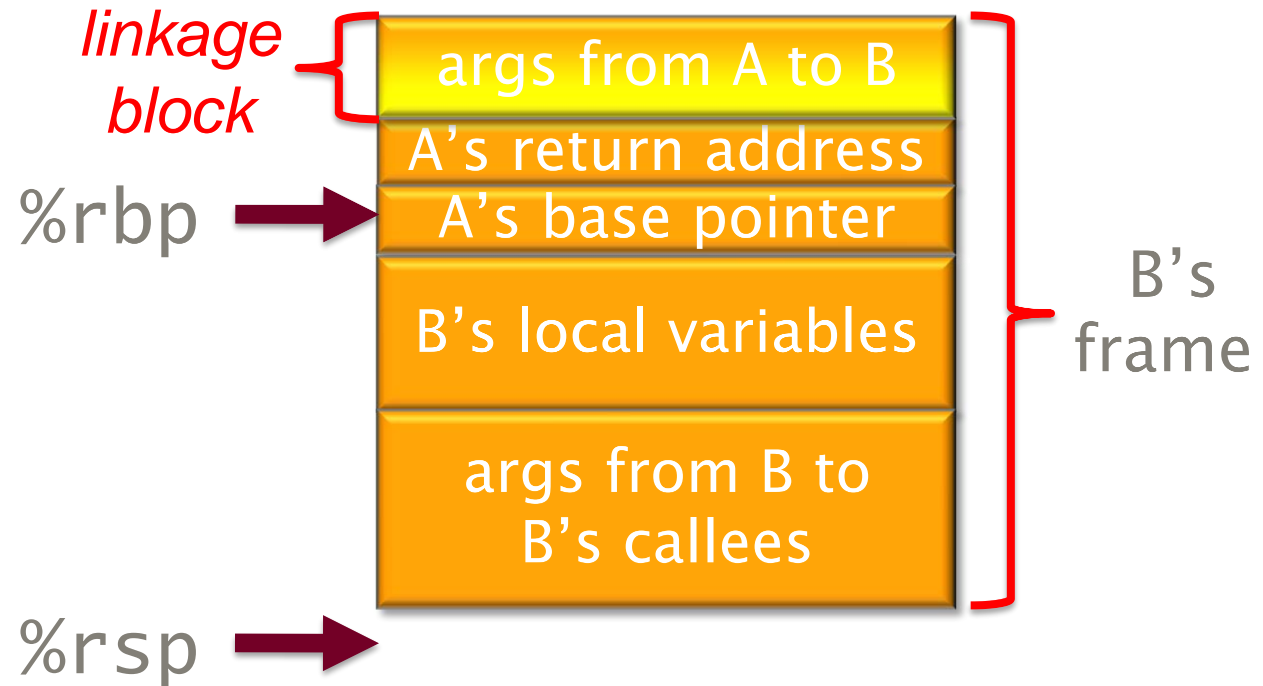
Function A calls function B which will call function C.



© 2010 Charles E. Leiserson and Saman Amarasinghe

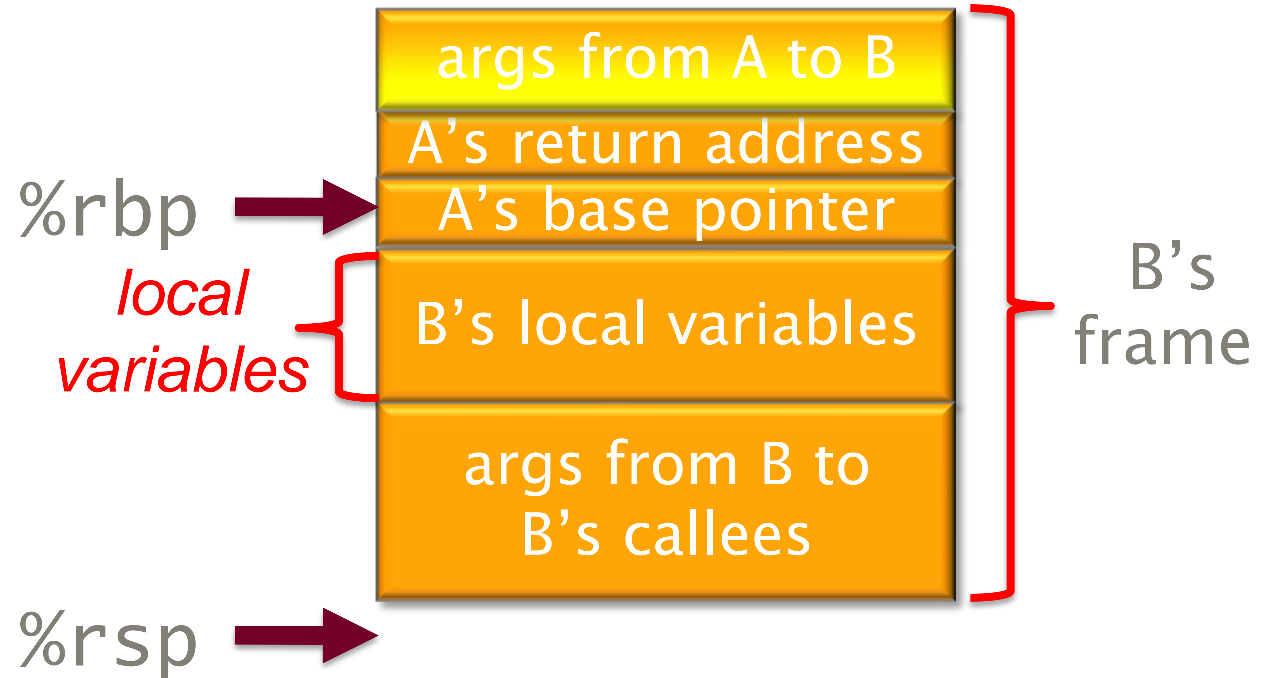
GCC/Linux C Subroutine Linkage

Function B accesses its nonregister arguments from A, which lie in a linkage block, by indexing %rbp with positive offset.



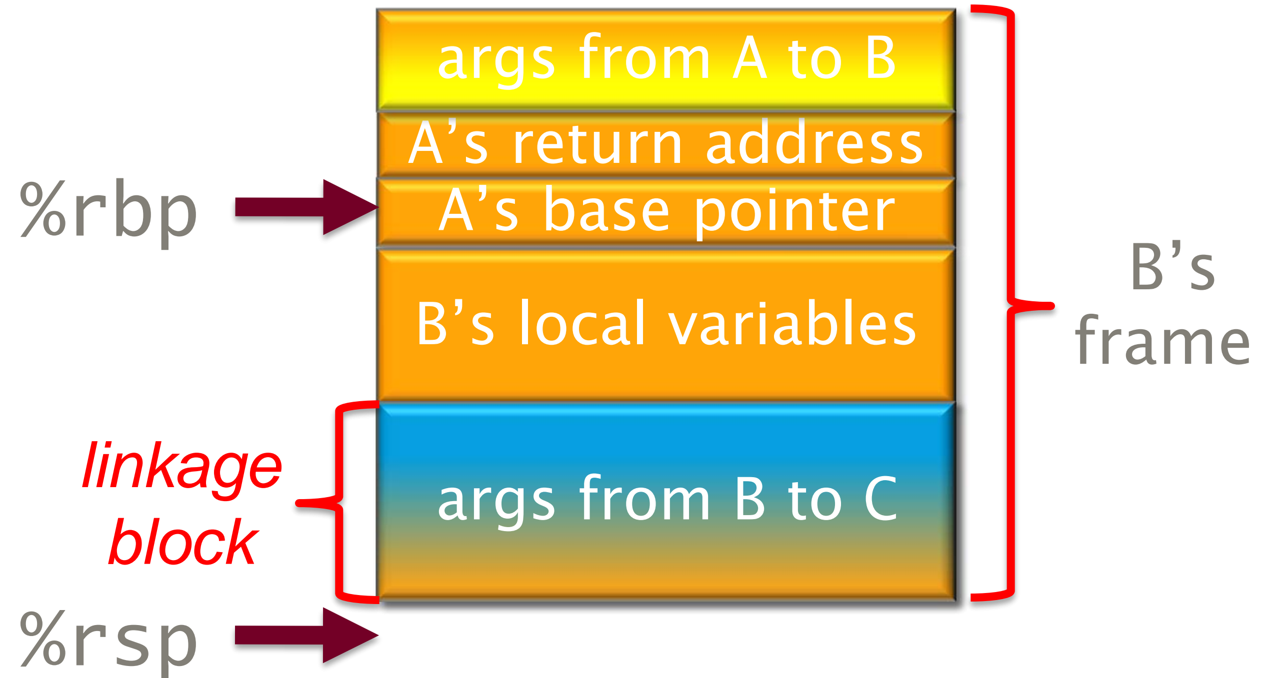
GCC/Linux C Subroutine Linkage

Function B accesses its local variables by indexing %rbp with negative offsets.



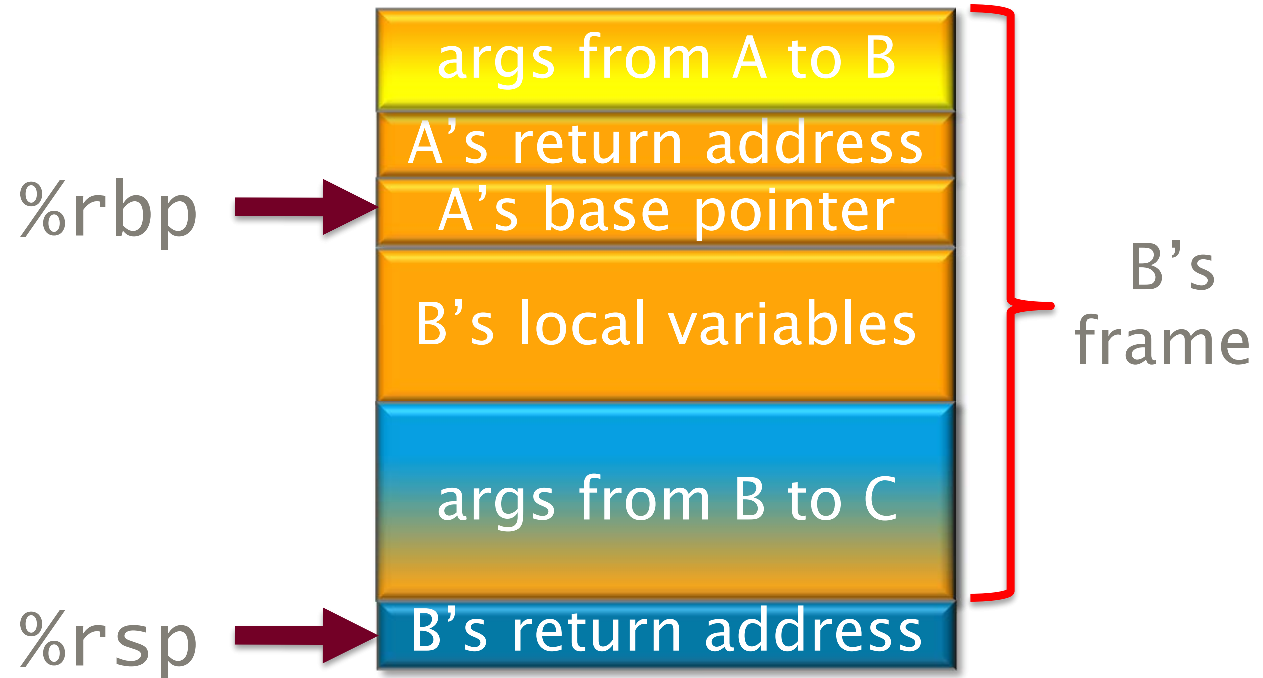
GCC/Linux C Subroutine Linkage

Before calling C, B places the **nonregister arguments** for C into the reserved linkage block it will share with C, which B accesses by indexing %rbp with **negative offsets**.



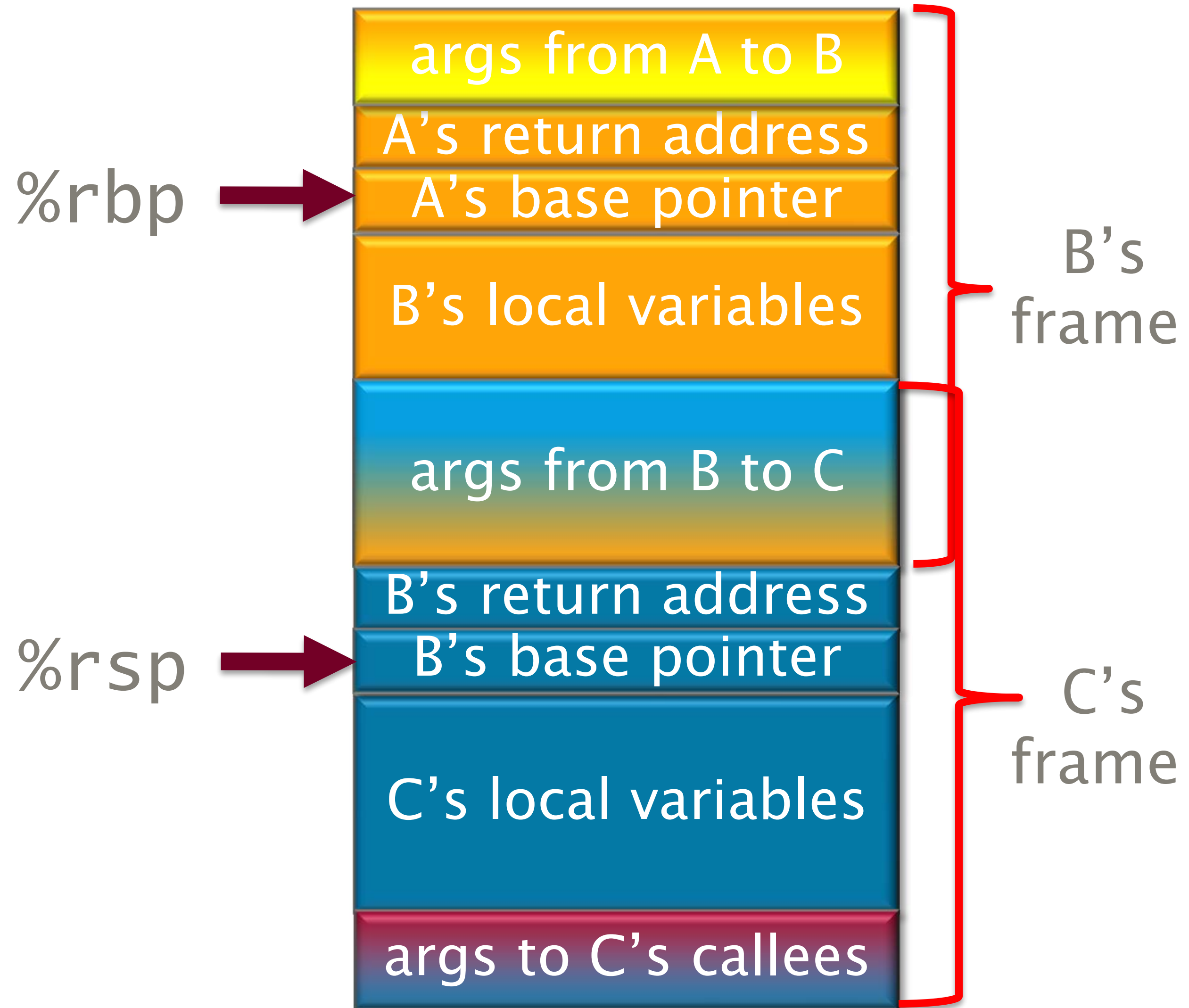
GCC/Linux C Subroutine Linkage

B calls C, which **saves the return address** for B on the stack and transfers control to C.



GCC/Linux C Subroutine Linkage

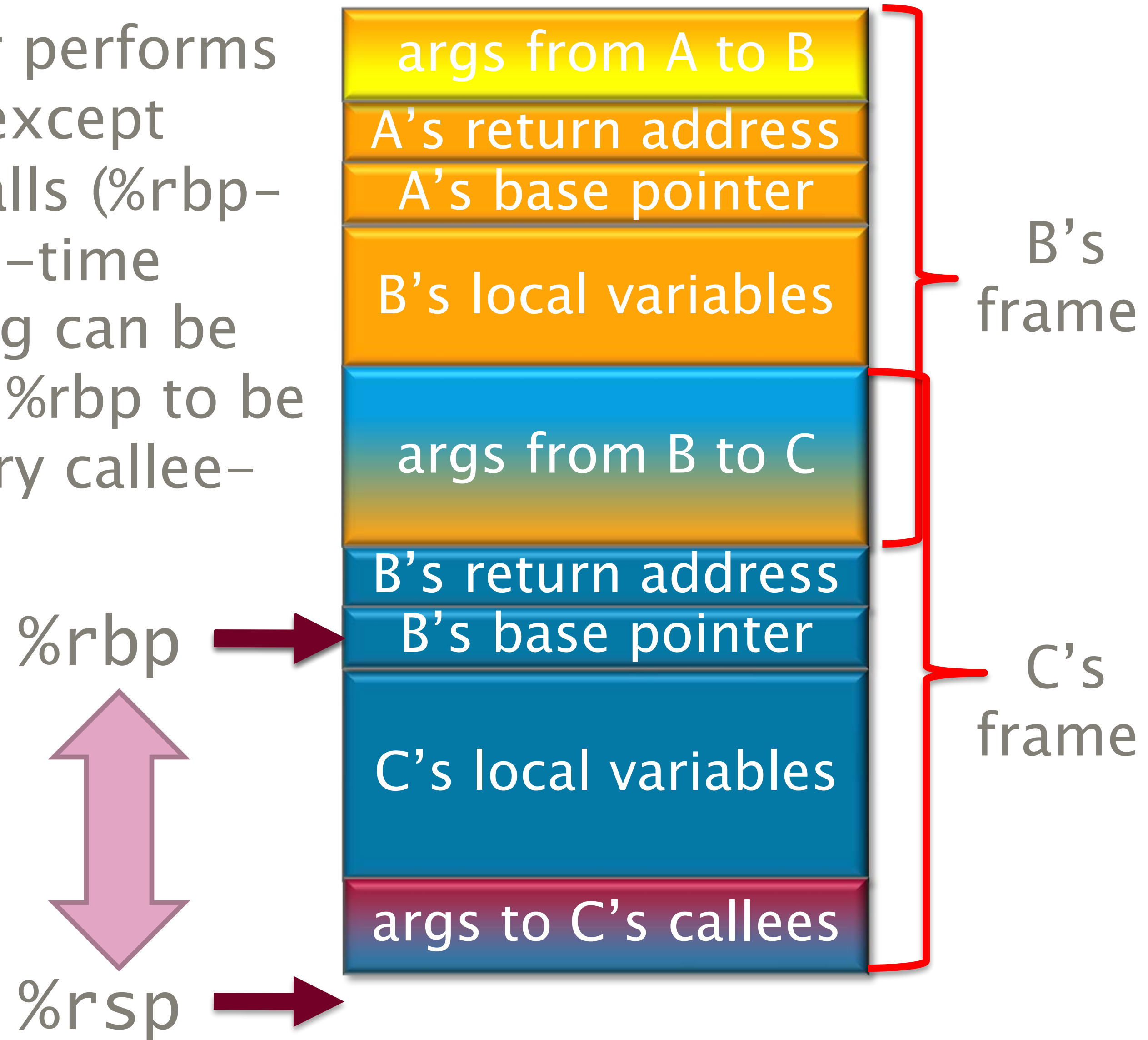
Function C
- saves B's base pointer on the stack,
- sets %rbp=%rsp,
- advances %rsp to allocate space for C's local variables and linkage block.



© 2010 Charles E. Leiserson and Saman Amarasinghe

GCC/Linux C Subroutine Linkage

If a function never performs stack allocations except during function calls (%rbp–%rsp is a compile-time constant), indexing can be off %rsp, allowing %rbp to be used as an ordinary callee-saved register.



.

© 2010 Charles E. Leiserson and Saman Amarasinghe

Procedure Calls and Recursion

```c
uint64_t fib(uint64_t n) {
    if (n < 2) return n;
    return (fib(n-1) + fib(n-2));
}
```

```
.globl fib
        .type   fib, @function
fib:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $24, %rsp
        movq    %rdi, -16(%rbp)
        cmpq    $1, -16(%rbp)
        ja      .L4
        movq    -16(%rbp), %rax
        movq    %rax, -24(%rbp)
        jmp     .L5
.L4:
        movq    -16(%rbp), %rax
        leaq    -1(%rax), %rdi
        call    fib
        movq    %rax, %rbx
        movq    -16(%rbp), %rax
        leaq    -2(%rax), %rdi
        call    fib
        addq    %rax, %rbx
        movq    %rbx, -24(%rbp)
.L5:
        movq    -24(%rbp), %rax
        addq    $24, %rsp
        popq    %rbx
        leave
        ret
```

Procedure Calls and Recursion

```
uint64_t fib(uint64_t n) {
    if (n < 2)
    return (f
}
```

Save base pointer and advance stack pointer.

```
.globl fib
        .type   fib, @functi
fib:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $24, %rsp
        movq    %rdi, -16(%rbp)

.L4:
        movq    -16(%rbp), %rax
        leaq    -1(%rax), %rdi
        call    fib
        movq    %rax, %rbx
        movq    -16(%rbp), %rax
        leaq    -2(%rax), %rdi
        call    fib
        addq    %rax, %rbx
        movq    %rbx, -24(%rbp)
.L5:
        movq    -24(%rbp), %rax
        addq    $24, %rsp
        popq    %rbx
        leave
        ret
```

Equivalent to

```
movq %rbp, %rsp
popq %rbp
```

Optimization

```
uint64_t fib(uint64_t n) {
    if (n < 2) return n;
    return (fib(n-1) + fib(n-2));
}
```

```
.globl fib
        .type   fib, @function
fib:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $24, %rsp
        movq    %rdi, -16(%rbp)
        cmpq    $1, -16(%rbp)
        ja      .L4
        movq    -16(%rbp), %rax
        movq    %rax, -24(%rbp)
        jmp     .L5
.L4:
        movq    -16(%rbp), %rax
        leaq    -1(%rax), %rdi
        call    fib
        movq    %rax, %rbx
        movq    -16(%rbp), %rax
        leaq    -2(%rax), %rdi
        call    fib
        addq    %rax, %rbx
        movq    %rbx, -24(%rbp)
.L5:
        movq    -24(%rbp), %rax
        addq    $24, %rsp
        popq    %rbx
        leave
        ret
```

© 2010 Charles E. Leiserson and Saman Amarasinghe

Optimization

```
uint64_t fib(uint64_t n) {
    if (n < 2) return n;
    return (fib(n-1) + fib(n-2));
}
```

```
.globl fib
        .type   fib, @function
fib:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $24, %rsp
        movq    %rdi, -16(%rbp)
        cmpq    $1, %rdi
        ja      .L4
        movq    %rdi, %rax
        movq    %rax, -24(%rbp)
        jmp     .L5
.L4:
        movq    %rdi, %rax
        leaq    -1(%rax), %rdi
        call    fib
        movq    %rax, %rbx
        movq    -16(%rbp), %rax
        leaq    -2(%rax), %rdi
        call    fib
        addq    %rax, %rbx
        movq    %rbx, -24(%rbp)
.L5:
        movq    -24(%rbp), %rax
        addq    $24, %rsp
        popq    %rbx
        leave
        ret
```

fib(43):
5.45s ⇨ 4.09s

© 2010 Charles E. Leiserson and Saman Amarasinghe

Optimization

```
uint64_t fib(uint64_t n) {
    if (n < 2) return n;
    return (fib(n-1) + fib(n-2));
}
```

```
.globl fib
        .type   fib, @function
fib:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $24, %rsp
        movq    %rdi, -16(%rbp)
        cmpq    $1, %rdi
        ja      .L4
        movq    %rdi, %rax
        movq    %rax, -24(%rbp)
        jmp     .L5
.L4:
        movq    %rdi, %rax
        leaq    -1(%rax), %rdi
        call    fib
        movq    %rax, %rbx
        movq    -16(%rbp), %rax
        leaq    -2(%rax), %rdi
        call    fib
        addq    %rax, %rbx
        movq    %rbx, -24(%rbp)
.L5:
        movq    -24(%rbp), %rax
        addq    $24, %rsp
        popq    %rbx
        leave
        ret
```

Optimization

```
uint64_t fib(uint64_t n) {
    if (n < 2) return n;
    return (fib(n-1) + fib(n-2));
}
```

```
.globl fib
        .type   fib, @function
fib:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $16, %rsp
        movq    %rdi, -16(%rbp)
        cmpq    $1, %rdi
        ja      .L4
        movq    %rdi, %rax
        movq    %rax, -24(%rbp)
        jmp     .L5
.L4:
        movq    %rdi, %rax
        leaq    -1(%rax), %rdi
        call    fib
        movq    %rax, %rbx
        movq    -16(%rbp), %rax
        leaq    -2(%rax), %rdi
        call    fib
        addq    %rbx, %rax
        movq    %rbx, -24(%rbp)
.L5:
        movq    -24(%rbp), %rax
        addq    $16, %rsp
        popq    %rbx
        leave
        ret
```

(The lines `movq %rax, -24(%rbp)`, `movq %rbx, -24(%rbp)` and `movq -24(%rbp), %rax` are struck through; `$16` and `addq %rbx, %rax` are highlighted in red.)

fib(43):
5.45s ⇨ 4.09s ⇨ 3.90s

© 2010 Charles E. Leiserson and Saman Amarasinghe

Optimization

```c
uint64_t fib(uint64_t n) {
    if (n < 2) return n;
    return (fib(n-1) + fib(n-2));
}
```

```
.globl fib
        .type   fib, @function
fib:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $16, %rsp
        movq    %rdi, -16(%rbp)
        cmpq    $1, %rdi
        ja      .L4
        movq    %rdi, %rax

        jmp     .L5
.L4:
        movq    %rdi, %rax
        leaq    -1(%rax), %rdi
        call    fib
        movq    %rax, %rbx
        movq    -16(%rbp), %rax
        leaq    -2(%rax), %rdi
        call    fib
        addq    %rbx, %rax

.L5:

        addq    $16, %rsp
        popq    %rbx
        leave
        ret
```

Optimization

```
uint64_t fib(uint64_t n) {
    if (n < 2) return n;
    return (fib(n-1) + fib(n-2));
}
```

```
.globl fib
        .type   fib, @function
fib:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $16, %rsp
        movq    %rdi, %rbx
        cmpq    $1, %rdi
        ja      .L4
        movq    %rdi, %rax

        jmp     .L5
.L4:
        movq    %rdi, %rax
        leaq    -1(%rax), %rdi
        call    fib
        movq    %rbx, %rdi
        movq    %rax, %rbx
        leaq    -2(%rdi), %rdi
        call    fib
        addq    %rbx, %rax
.L5:

        addq    $16, %rsp
        popq    %rbx
        leave
        ret
```

(Struck out: `subq $16, %rsp` and `addq $16, %rsp`. Highlighted in red: `%rbx` in `movq %rdi, %rbx`, plus `movq %rbx, %rdi`, `movq %rax, %rbx` and `leaq -2(%rdi), %rdi`.)

fib(43):

5.45s ⇨ 4.09s ⇨ 3.90s ⇨ 3.61s

© 2010 Charles E. Leiserson and Saman Amarasinghe

Simple Optimization Strategies

- Keep values in registers to eliminate excess memory traffic.
- Optimize naive function-call linkage.
- Constant fold!

Compiling Conditionals

```
if (p) {
    ctrue;
}
else {
    cfalse;
}
```

```
    ⟨ instructions to evaluate p ⟩
    j⟨p false⟩ Else_clause
Then_clause:
    ⟨ instructions for ctrue ⟩
    jmp end_if
Else_clause:
    ⟨ instructions for cfalse ⟩
end_if:
```

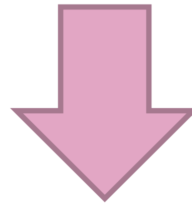
Compiling `while` Loops

```
while (p) {
    c;
}
```

```
        jmp Loop_test
Loop:
        ⟨instructions for c⟩
Loop_test:
        ⟨ instructions to evaluate p ⟩
        j⟨p true⟩  Loop
```

Compiling for Loops

```
for (initcode; p; nextcode) {
    code;
}
```



```
initcode;
while (p) {
    code;
    nextcode;
 }
```

© 2010 Charles E. Leiserson and Saman Amarasinghe

Implementing Arrays

- Arrays are just blocks of memory
  - Static array: allocated in data segment
  - Dynamic array: allocated on the heap
  - Local array: allocated on the stack
- Array/pointer equivalence
  - *a ≡ a[0].
  - A pointer is merely an index into the array of all memory.
  - What is 8[a]?

© 2010 Charles E. Leiserson and Saman Amarasinghe

Implementing Structs

- Structs are just blocks of memory
  - `struct {char x; int i; double d; } s;`
- Fields stored next to each other
- Be careful about alignment issues.
  - It's generally better to declare longer fields before shorter fields.
- Like arrays, there are static, dynamic, and local structs.

XMM Stuff

- SIMD instructions — operate on small vectors
- 16 128-bit XMM registers
  - 2 64-bit values
  - 4 32-bit values
- Instructions operate on multiple values
  - // move 4 32-bit ints to %xmm0
    ```
    movdqa y(%rax), %xmm0
    ```
  - // add 4 32-bit ints in z to corresponding
    // 4 32-bit ints in %xmm0)
    ```
    paddd z(%rax), %xmm0
    ```

© 2010 Charles E. Leiserson and Saman Amarasinghe

More C/C++ Constructs

- Arrays of structs
- Structs of arrays
- Function pointers
- Bit fields in arrays
- Objects, virtual function tables
- Memory-management techniques

MIT OpenCourseWare
http://ocw.mit.edu

6.172 Performance Engineering of Software Systems
Fall 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.