6.172
Performance
Engineering
of Software
Systems

LECTURE 3

C to Assembly

I-Ting Angelina Lee

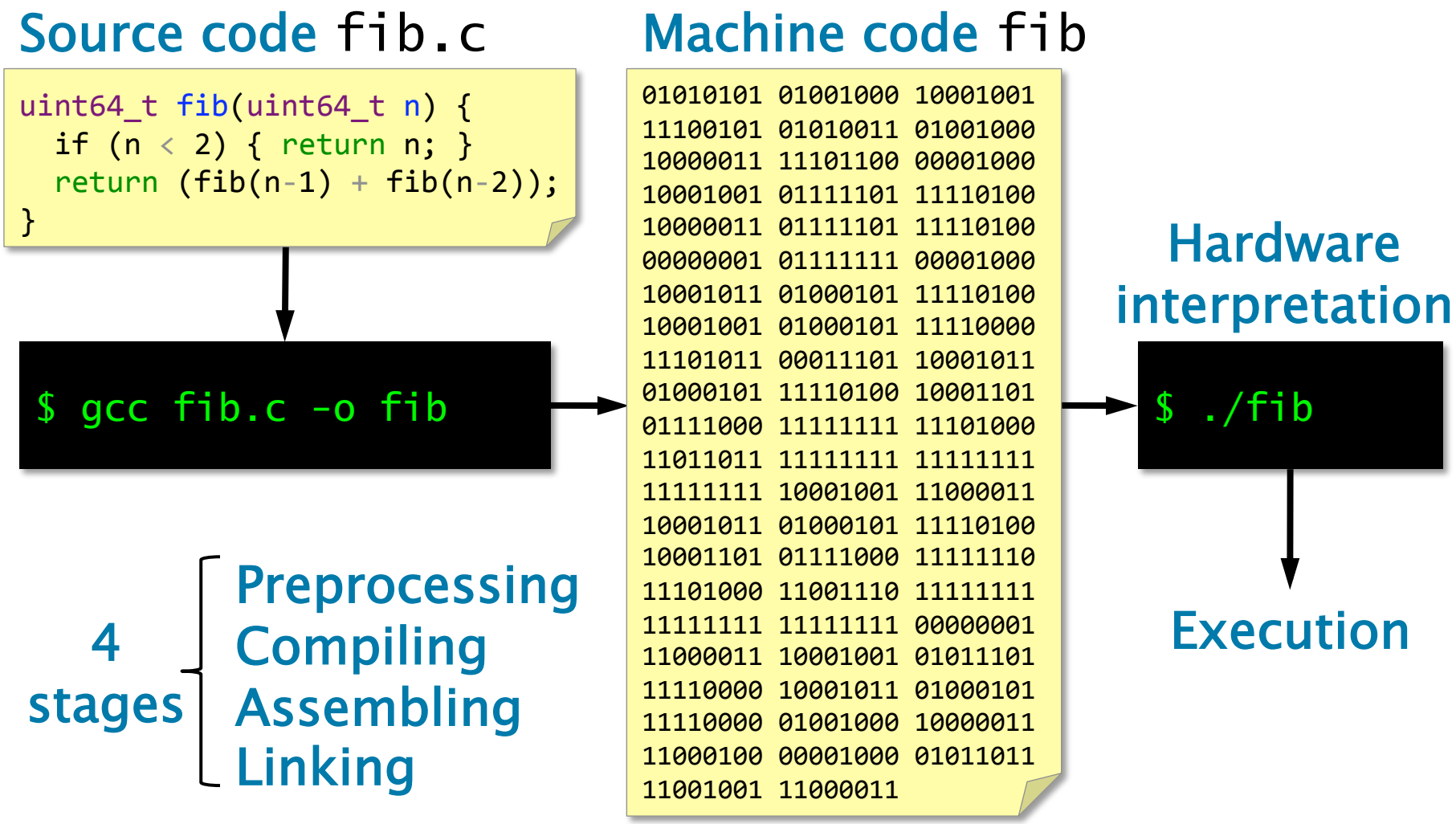
September 13, 2012

Why Assembly?

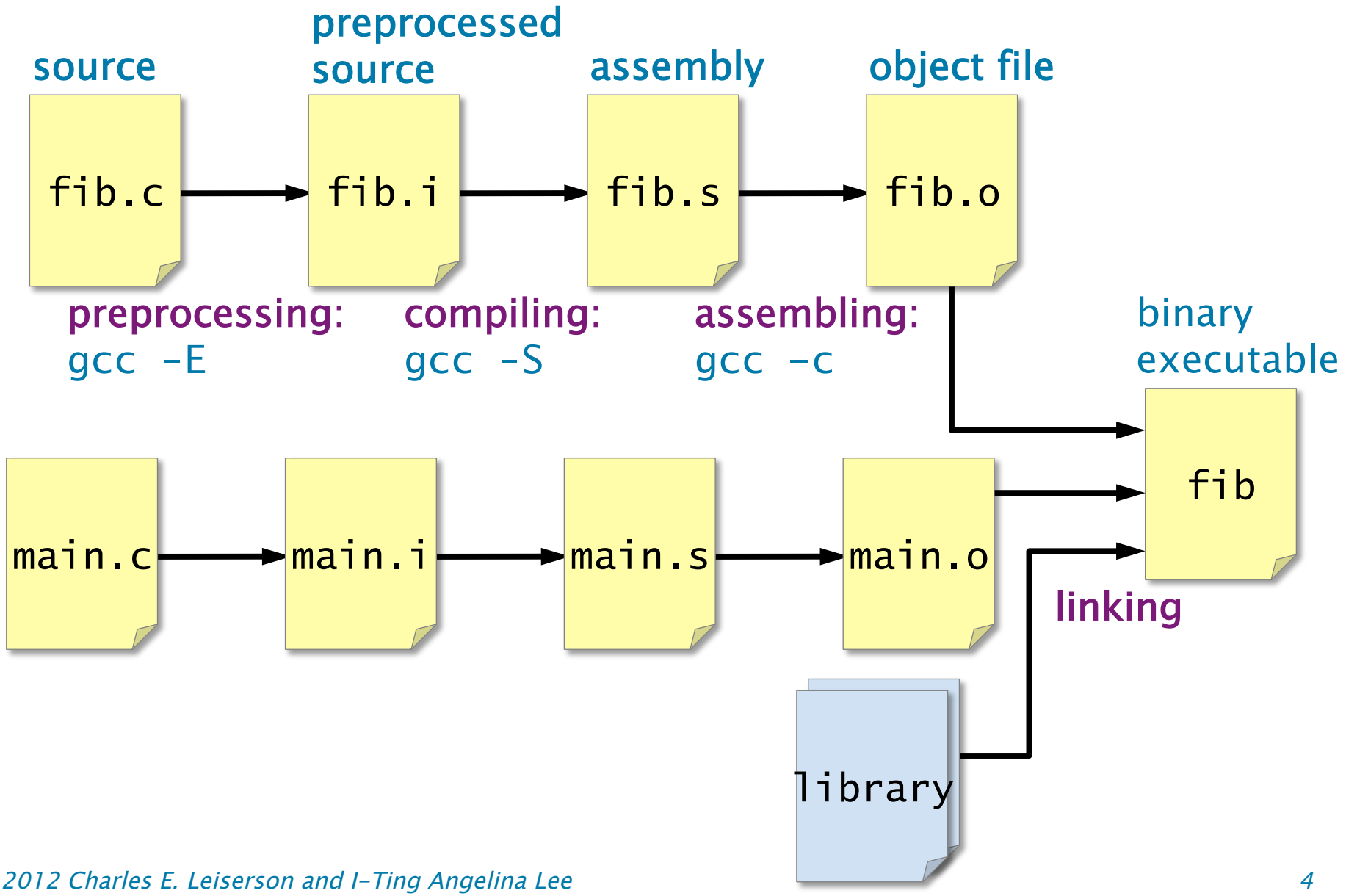
- Bugs happen
  (more often than you'd like)
- Optimization
- Reverse engineering
- It's good to know what your compiler is doing

- ... Homework 2?

Source Code to Execution



The Four Stages of Compilation



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 -S -o fib.s fib.c
```

Assembly code `fib.s`

```
        .globl fib
        .type  fib, @function
fib:
        pushq       %rbp
        movq        %rsp, %rbp
        pushq       %rbx
        subq        $24, %rsp
        movq        %rdi, -24(%rbp)
        cmpq        $1, -24(%rbp)
        ja          .L2
        movq        -24(%rbp), %rax
        jmp         .L3
.L2:
        movq        -24(%rbp), %rax
        subq        $1, %rax
        movq        %rax, %rdi
        call        fib
        movq        %rax, %rbx
        movq        -24(%rbp), %rax
        subq        $2, %rax
        movq        %rax, %rdi
        call        fib
        addq        %rbx, %rax
        ...
```

Assembly language provides a convenient symbolic representation of machine language.

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

Assembly Code to Executable

Assembly code `fib.s`

```
        .globl fib
        .type  fib, @function
fib:
        pushq       %rbp
        movq        %rsp, %rbp
        pushq       %rbx
        subq        $24, %rsp
        movq        %rdi, -24(%rbp)
        cmpq        $1, -24(%rbp)
        ja          .L2
        movq        -24(%rbp), %rax
        jmp         .L3
.L2:
        movq        -24(%rbp), %rax
        subq        $1, %rax
        movq        %rax, %rdi
        call        fib
        movq        %rax, %rbx
        movq        -24(%rbp), %rax
        subq        $2, %rax
        movq        %rax, %rdi
        call        fib
        addq        %rbx, %rax
        ...
```

```
$ gcc fib.s -o fib.o
```

Machine code

```
01010101 01001000
10001001 11100101
01010011 01001000
10000011 11101100
00001000 10001001
01111101 11110100
10000011 01111101
11110100 00000001
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` and assemble with gcc.

Disassembling

Source, machine, & assembly

Binary executable `fib` with debug symbols (compiled with -g)

```
$ objdump -S fib
```

```
uint64_t fib(uint64_t n) {
  400704:  55                    push   %rbp
  400705:  48 89 e5              mov    %rsp,%rbp
  400708:  53                    push   %rbx
  400709:  48 83 ec 18           sub    $0x18,%rsp
  40070d:  48 89 7d e8           mov    %rdi,-0x18(%rbp)
if (n < 2) { return n; }
  400711:  48 83 7d e8 01        cmpq   $0x1,-0x18(%rbp)
  400716:  77 06                 ja     40071e <fib+0x1a>
  400718:  48 8b 45 e8           mov    -0x18(%rbp),%rax
  40071c:  eb 26                 jmp    400744 <fib+0x40>
return (fib(n-1) + fib(n-2));
  40071e:  48 8b 45 e8           mov    -0x18(%rbp),%rax
  400722:  48 83 e8 01           sub    $0x1,%rax
  400726:  48 89 c7              mov    %rax,%rdi
  400729:  e8 d6 ff ff ff        callq  400704 <fib>
  40072e:  48 89 c3              mov    %rax,%rbx
  400731:  48 8b 45 e8           mov    -0x18(%rbp),%rax
  400735:  48 83 e8 02           sub    $0x2,%rax
  400739:  48 89 c7              mov    %rax,%rdi
  40073c:  e8 c3 ff ff ff        callq  400704 <fib>
  400741:  48 01 d8              add    %rbx,%rax
}
  400744:  48 83 c4 18           add    $0x18,%rsp
  400748:  5b                    pop    %rbx
  400749:  5d                    pop    %rbp
  40074a:  c3                    retq
```

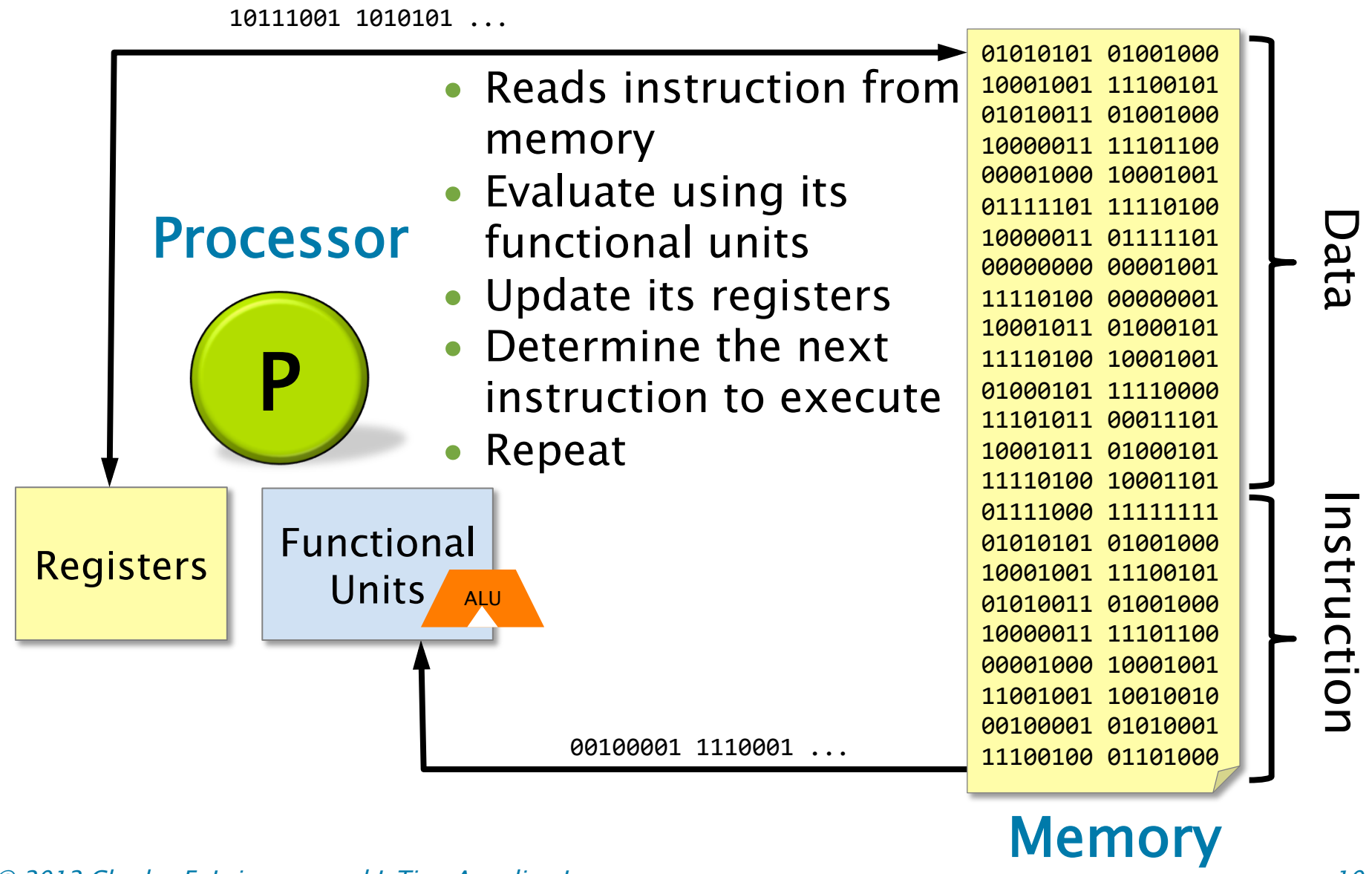
Expectations of Students

- 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.

- General registers
- Instruction formats
- Data allocation and addressing modes

x86-64 Assembly 101

Generic Single-Core Machine



X86-64 Registers

16	64-bit	general-purpose registers
6	16-bit	segment registers
1	64-bit	RFLAGS register
1	64-bit	instruction pointer register (%rip)
7	64-bit	control register
8	64-bit	MMX registers
16	128-bit	XMM registers (for SSE)
1	32-bit	MXCSR register (SSE2 control register)
16	256-bit	YMM registers (for AVX)
8	80-bit	x87 FPU data registers
1	16-bit	x87 FPU control register
1	16-bit	x87 FPU status register
1	48-bit	x87 FPU instruction pointer register
1	48-bit	x87 FPU data operant pointer register
1	16-bit	x87 FPU tag register
1	11-bit	x87 FPU opcode register

x86-64 General Registers

63	31	15	7 0
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%rllb
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

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

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.

Instruction Format

⟨opcode⟩ ⟨operand_list⟩

- ⟨opcode⟩ is a short mnemonic identifying the type of instruction with a single-character suffix indicating the data type.
  - **Example:** `movq -16(%rbp), %rax`
  - 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).

x86-64 Data Types

C declaration	C constant	x86-64 size in bytes	Assembly suffix	Data type
char	'c'	1	b	Byte
short	172	2	w	Word
int	172	4	l	Double word
unsigned int	172U	4	l	Double word
long	172L	8	q	Quad word
unsigned long	172UL	8	q	Quad word
char *	"6.172"	8	q	Quad word
float	6.172F	4	s	Single precision
double	6.172	8	d	Double precision
long double	6.172L	16(10)	t	Extended precision

x86-64 Opcode Examples

Arithmetic and logical: `add`, `sub`, `mult`, `and`, `or`, `not`, `cmp`, ...

- `subq %rdx, %rax` (`%rax = %rax - %rdx`)

Shift/rotate instructions: `sar`, `sal`, `shr`, `shl` ...

- `sar` and `sal` are arithmetic (signed) shift.

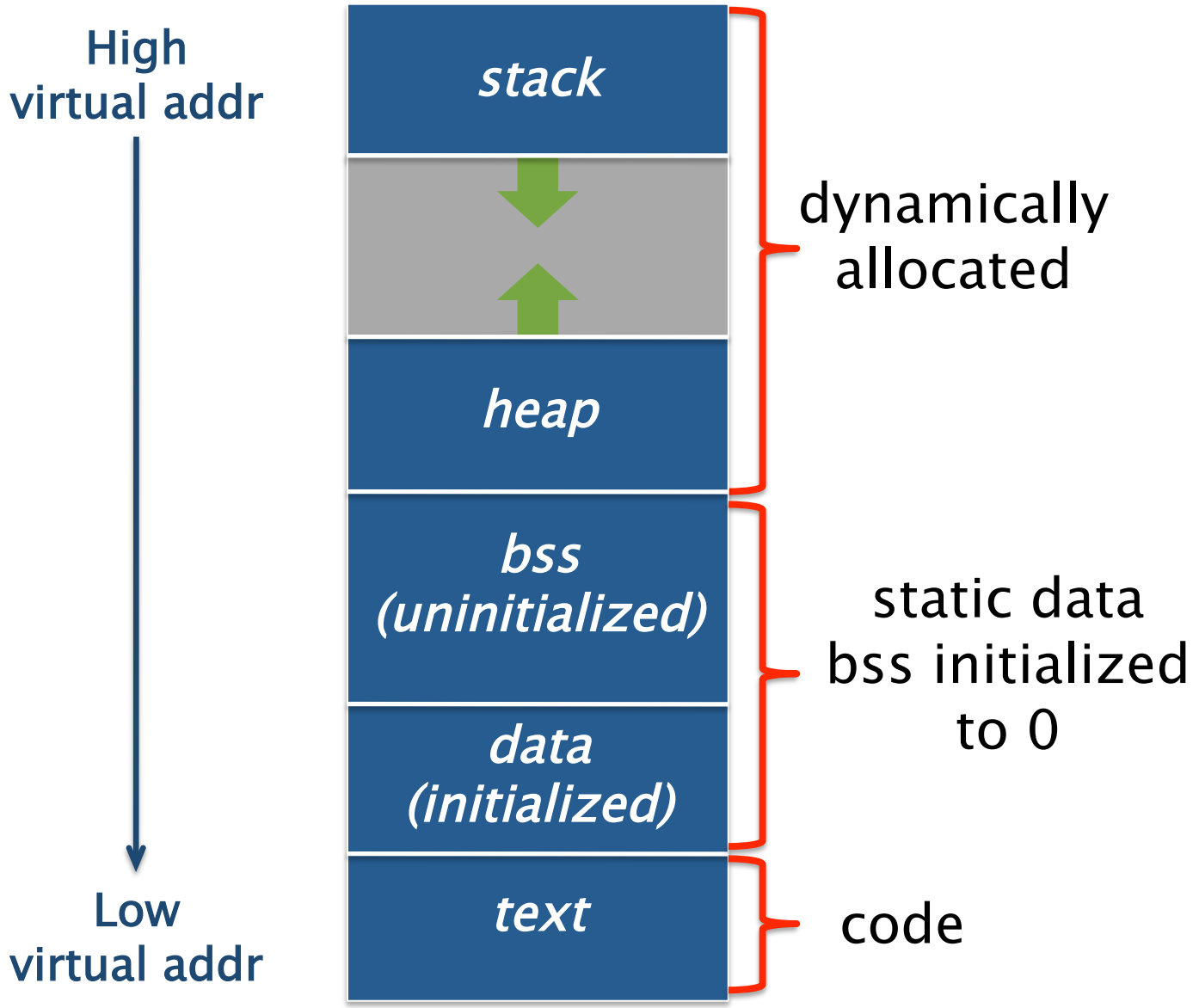
Control transfer: `call`, `ret`, `jmp`, `j⟨condition⟩`, ...

Data-transfer: `mov`, `push`, `pop`, ...

- **Careful:** Results of 32-bit operations are implicitly zero-extended to 64-bit values, unlike results of 8- and 16-bit operations.
- `movl $-1, %eax` ≡ `movq 0x00000000ffffffff, %rax`
- To preserve the sign bit:
  `movslq %eax, %rdx` (move sign extended)

See `http://www.x86-64.org/documentation/assembly.html`.

Memory Layout



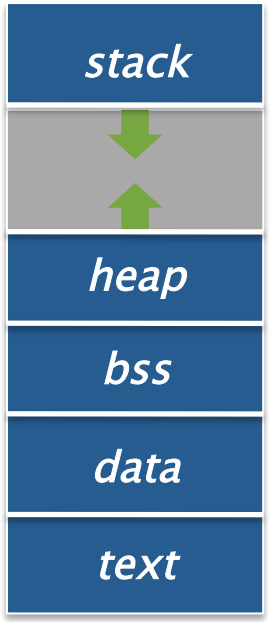
Assembler Directives

- Labels:

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

- Segment directives:

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



- 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
```

- Scope and linkage directives:

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

See assembler manual.

X86-64 Addressing Modes

Register:

```
addq %rbx, %rax          //value in register rax
```

Direct:

```
movq 0x172, %rdi         //contents at address 172 hex  (370 in dec)
```

Immediate:

```
movq $172, %rdi          // value 172
```

Register indirect:

```
movq %rbx, (%rax)        // data at addr in reg
```

Register indexed:

```
movq $6, 172(%rax)       // address is 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   // address is %rdi+8*rdx+172
```

Instruction-pointer relative:

```
movq 6(%rip), %rax
```

Only one operand may address memory.

Translating Expressions

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

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

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

```
foo2:
# params in %rdi, %rsi, %rdx
  leaq    (%rsi,%rdi), %rax
  orq     %rdx, %rax
  ret
```

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

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

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

Linux x86-64 Calling Convention

Linux x86-64 Calling Convention

`%rbp` and `%rsp` point to the top and the bottom of the (currently executing) stack frame in memory.

`%rip` points to the currently instruction in memory.

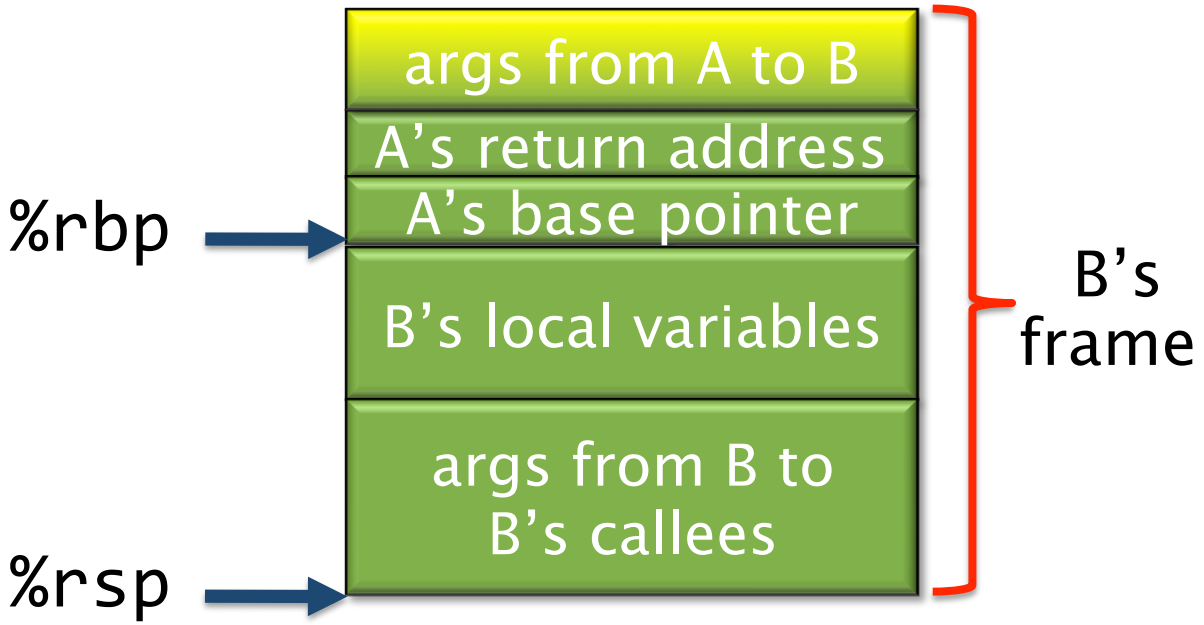
- call instruction pushes `%rip` on stack, jumps to call target operand (address of procedure)
- ret instruction pops `%rip` from stack, returns to caller

Software conventions

- Caller-save registers (`%r10`, `%r11`)
- Callee-save registers (`%rbx`, `%rbp`, `%r12-%r15`)

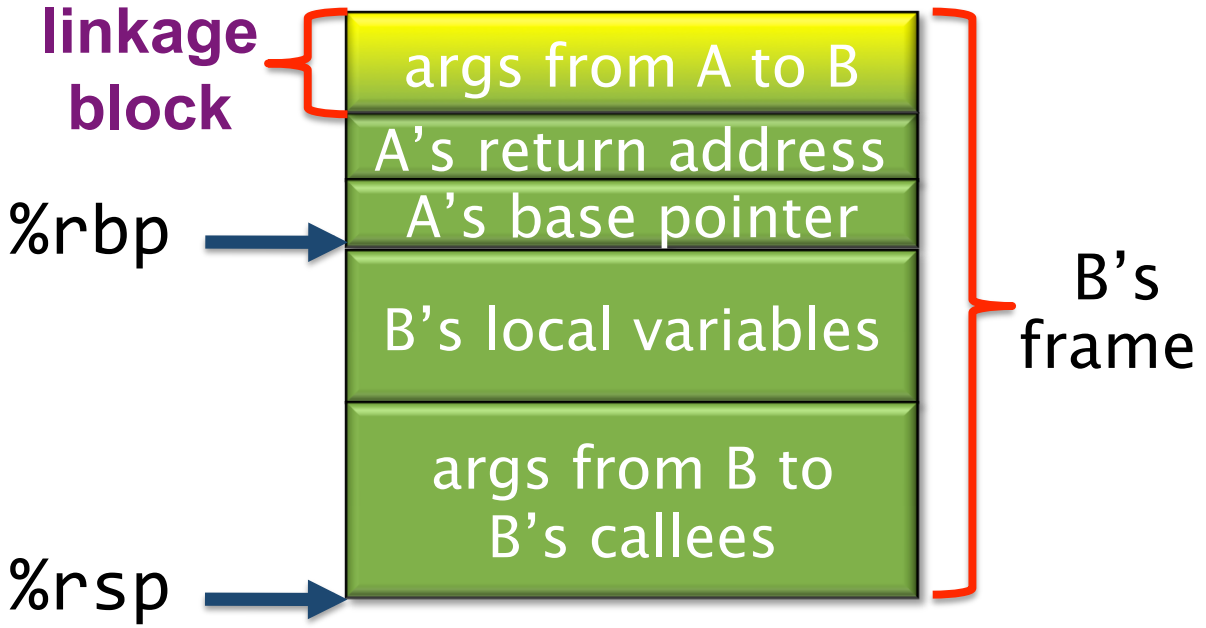
GCC/Linux C Subroutine Linkage

Function A calls function B which will call function C.



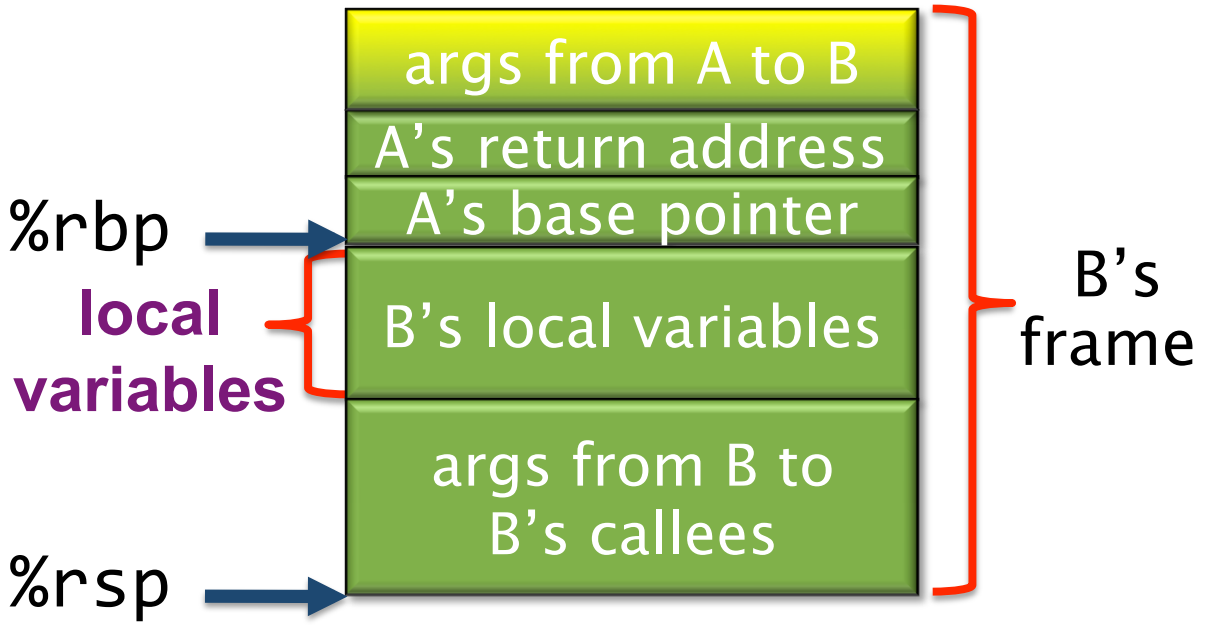
GCC/Linux C Subroutine Linkage

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



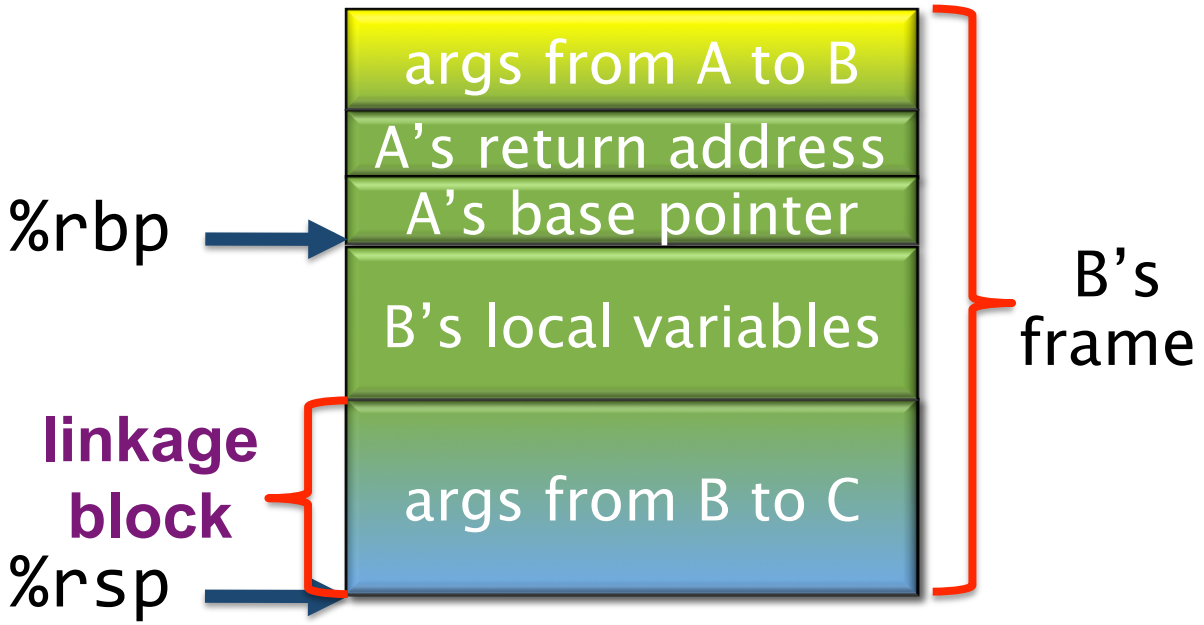
GCC/Linux C Subroutine Linkage

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



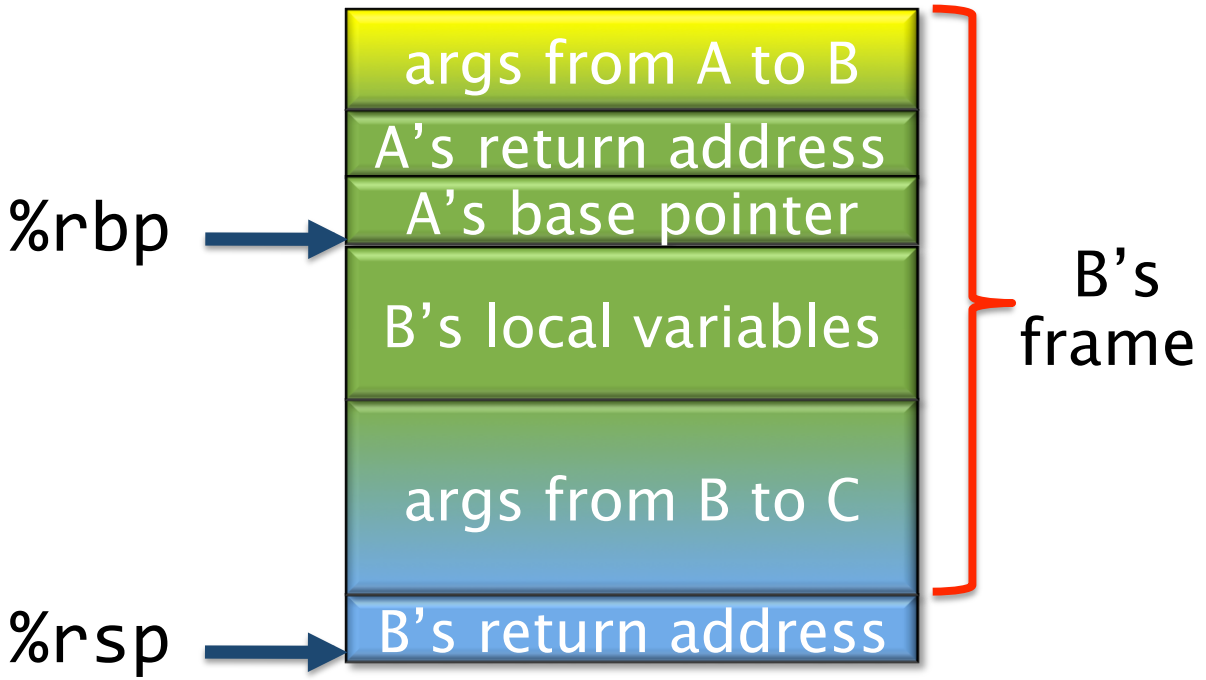
GCC/Linux C Subroutine Linkage

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

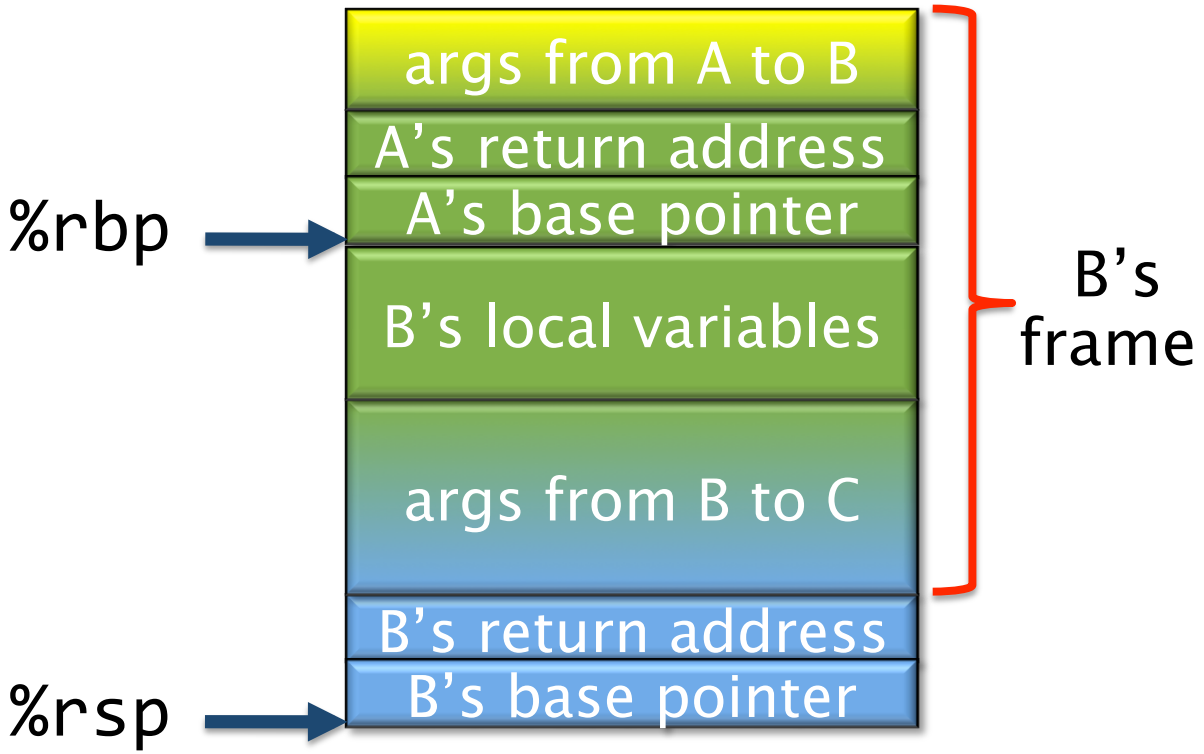
B calls C, which saves the return address for B on the stack and transfers control to C.



GCC/Linux C Subroutine Linkage

Prologue for `function C`

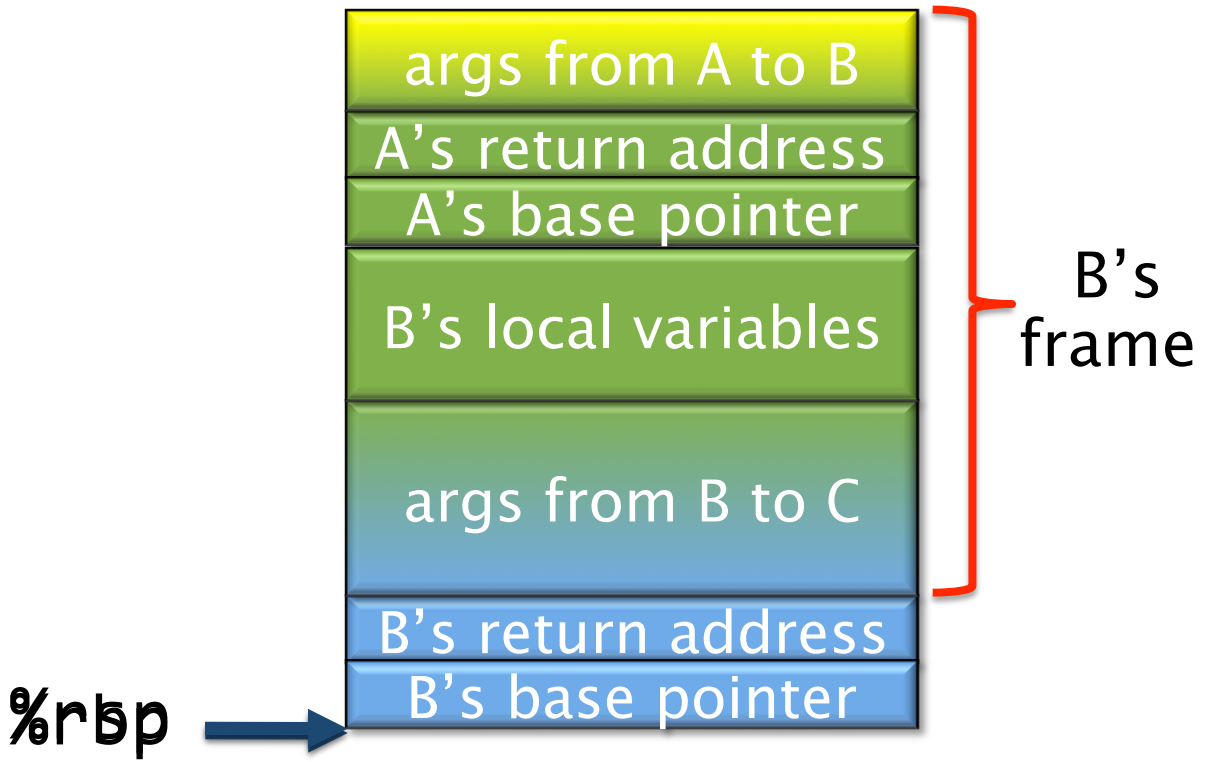
- saves B's base pointer on the stack,



GCC/Linux C Subroutine Linkage

Prologue for `function C`

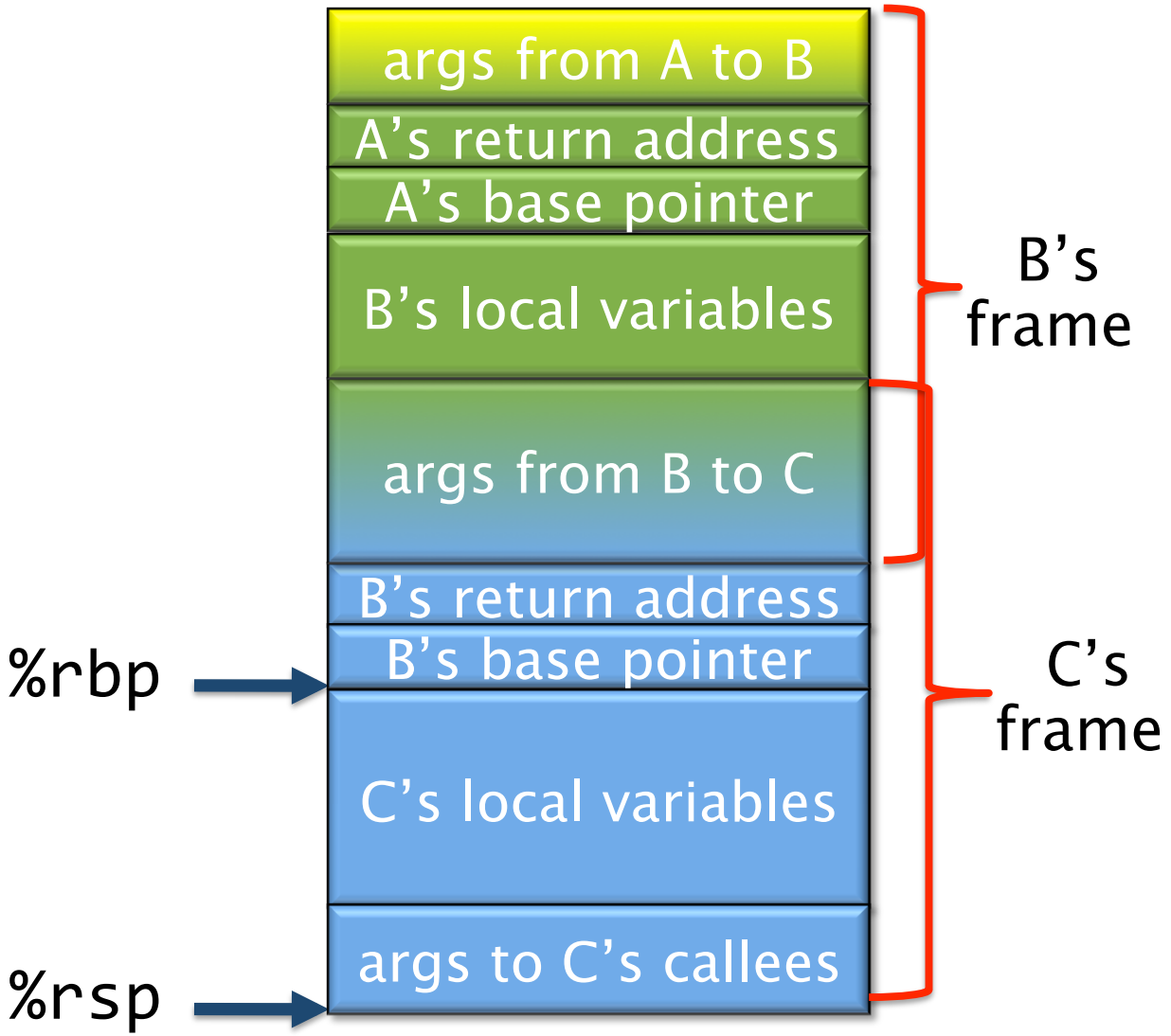
- saves B's base pointer on the stack,
- sets `%rbp=%rsp`,



GCC/Linux C Subroutine Linkage

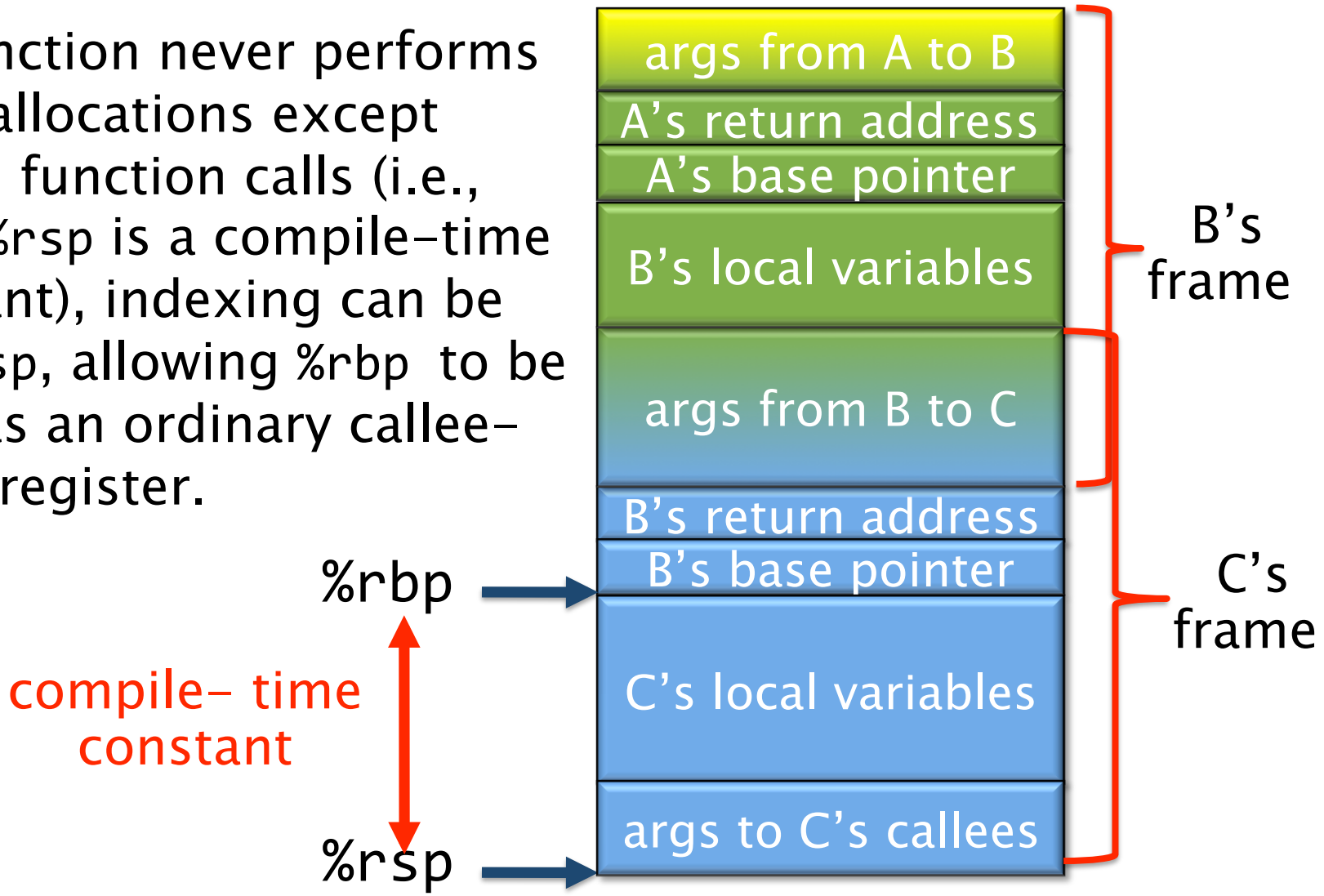
Prologue for `function C`

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



GCC/Linux C Subroutine Linkage

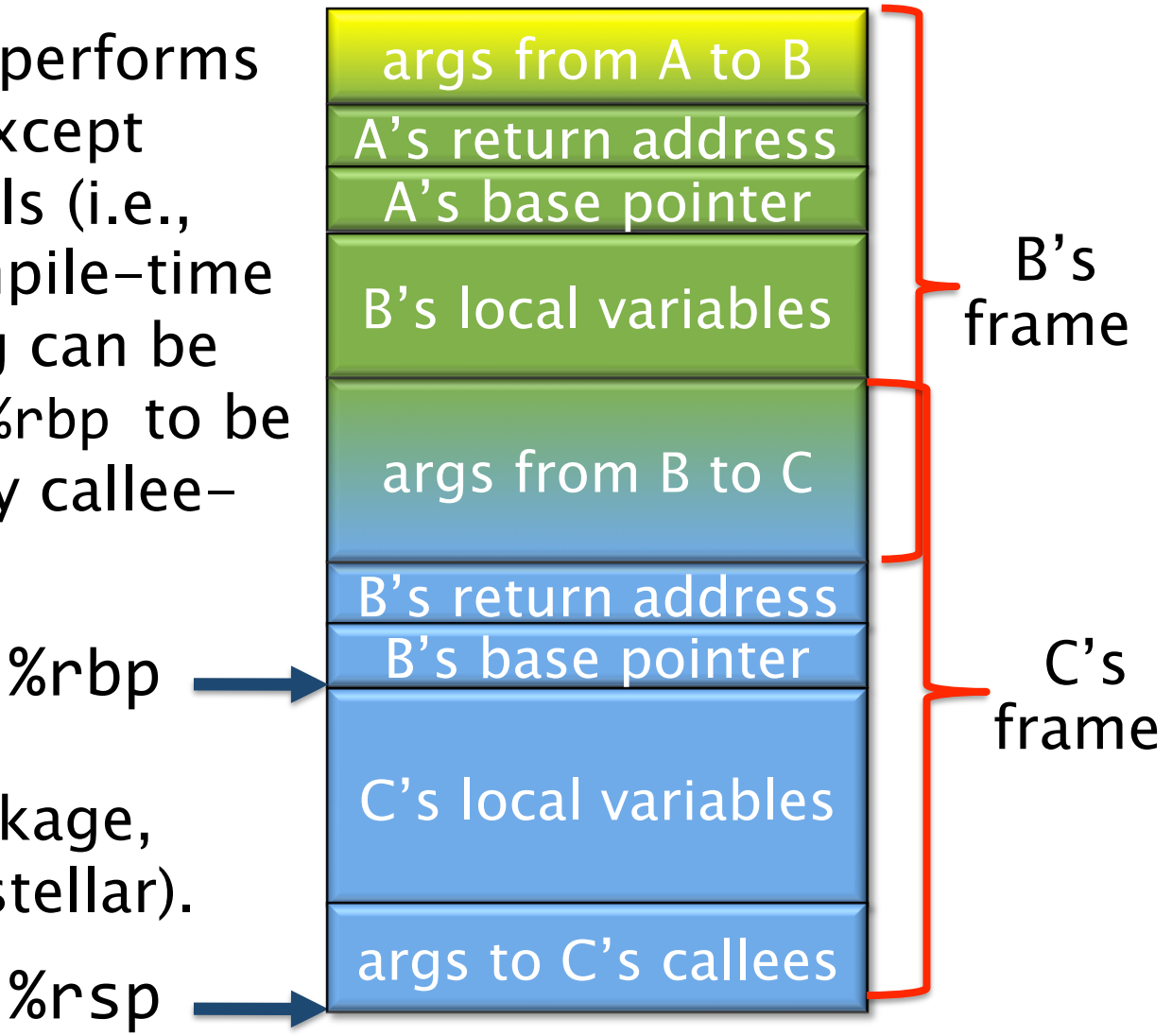
If a function never performs stack allocations except during function calls (i.e., `%rbp-%rsp` is a compile-time constant), indexing can be off `%rsp`, allowing `%rbp` to be used as an ordinary callee-saved register.



GCC/Linux C Subroutine Linkage

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

For details on C linkage, see System V ABI (stellar).



Case Study: Fibonacci

Computing Fibonacci

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Computing Fibonacci

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

```
 1      .globl fib
 2      .type  fib, @function
 3  fib:
 4      pushq   %rbp
 5      movq    %rsp, %rbp
 6      pushq   %rbx
 7      subq    $24, %rsp
 8      movq    %rdi, -24(%rbp)
 9      cmpq    $1, -24(%rbp)
10      ja      .L2
11      movq    -24(%rbp), %rax
12      jmp     .L3
13  .L2:
14      movq    -24(%rbp), %rax
15      subq    $1, %rax
16      movq    %rax, %rdi
17      call    fib
18      movq    %rax, %rbx
19      movq    -24(%rbp), %rax
20      subq    $2, %rax
21      movq    %rax, %rdi
22      call    fib
23      addq    %rbx, %rax
24  .L3:
25      addq    $24, %rsp
26      popq    %rbx
27      popq    %rbp
28      ret
```

Declare fib to be global

Computing Fibonacci

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Function prologue:
save %rbp and advance %rsp

Computing Fibonacci

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Save callee-saved register

Computing Fibonacci

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Function prologue / epilogue

Computing Fibonacci

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

```
 1      .globl fib
 2      .type  fib, @function
 3  fib:
 4      pushq   %rbp
 5      movq    %rsp, %rbp
 6      pushq   %rbx
 7      subq    $24, %rsp
 8      movq    %rdi, -24(%rbp)
 9      cmpq    $1, -24(%rbp)
10      ja      .L2
11      movq    -24(%rbp), %rax
12      jmp     .L3
13  .L2:
14      movq    -24(%rbp), %rax
15      subq    $1, %rax
16      movq    %rax, %rdi
17      call    fib
18      movq    %rax, %rbx
19      movq    -24(%rbp), %rax
20      subq    $2, %rax
21      movq    %rax, %rdi
22      call    fib
23      addq    %rbx, %rax
24  .L3:
25      addq    $24, %rsp
26      popq    %rbx
27      popq    %rbp
28      ret
```

Save incoming argument n

Computing Fibonacci

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Check for the base case

Computing Fibonacci

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

ja = jump if above
the else branch

Computing Fibonacci

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Take the `if` branch

Computing Fibonacci

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Return n (already in %rax)

Computing Fibonacci

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

The `else` branch: invoke `fib(n-1)`, arg in `%rdi`

Computing Fibonacci

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

```
 1      .globl fib
 2      .type  fib, @function
 3  fib:
 4      pushq   %rbp
 5      movq    %rsp, %rbp
 6      pushq   %rbx
 7      subq    $24, %rsp
 8      movq    %rdi, -24(%rbp)
 9      cmpq    $1, -24(%rbp)
10      ja      .L2
11      movq    -24(%rbp), %rax
12      jmp     .L3
13  .L2:
14      movq    -24(%rbp), %rax
15      subq    $1, %rax
16      movq    %rax, %rdi
17      call    fib
18      movq    %rax, %rbx
19      movq    -24(%rbp), %rax
20      subq    $2, %rax
21      movq    %rax, %rdi
22      call    fib
23      addq    %rbx, %rax
24  .L3:
25      addq    $24, %rsp
26      popq    %rbx
27      popq    %rbp
28      ret
```

Move return value from %rax into %rbx

Computing Fibonacci

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

```
 1      .globl fib
 2      .type  fib, @function
 3  fib:
 4      pushq   %rbp
 5      movq    %rsp, %rbp
 6      pushq   %rbx
 7      subq    $24, %rsp
 8      movq    %rdi, -24(%rbp)
 9      cmpq    $1, -24(%rbp)
10      ja      .L2
11      movq    -24(%rbp), %rax
12      jmp     .L3
13  .L2:
14      movq    -24(%rbp), %rax
15      subq    $1, %rax
16      movq    %rax, %rdi
17      call    fib
18      movq    %rax, %rbx
19      movq    -24(%rbp), %rax
20      subq    $2, %rax
21      movq    %rax, %rdi
22      call    fib
23      addq    %rbx, %rax
24  .L3:
25      addq    $24, %rsp
26      popq    %rbx
27      popq    %rbp
28      ret
```

Invoke fib(n-2)

Computing Fibonacci

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

```
 1      .globl fib
 2      .type  fib, @function
 3  fib:
 4      pushq   %rbp
 5      movq    %rsp, %rbp
 6      pushq   %rbx
 7      subq    $24, %rsp
 8      movq    %rdi, -24(%rbp)
 9      cmpq    $1, -24(%rbp)
10      ja      .L2
11      movq    -24(%rbp), %rax
12      jmp     .L3
13  .L2:
14      movq    -24(%rbp), %rax
15      subq    $1, %rax
16      movq    %rax, %rdi
17      call    fib
18      movq    %rax, %rbx
19      movq    -24(%rbp), %rax
20      subq    $2, %rax
21      movq    %rax, %rdi
22      call    fib
23      addq    %rbx, %rax
24  .L3:
25      addq    $24, %rsp
26      popq    %rbx
27      popq    %rbp
28      ret
```

Sum the two results in %rax

Computing Fibonacci

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Function epilogue and return

Optimization

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Optimization

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, -24(%rbp)
10       ja      .L2
11       movq    -24(%rbp), %rax
12       jmp     .L3
13  .L2:
14       movq    -24(%rbp), %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Optimization

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, %rdi
10       ja      .L2
11       movq    %rdi, %rax
12       jmp     .L3
13  .L2:
14       movq    %rdi, %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Keep values in registers

fib(43): 5.49s ⇨ 4.09s

Optimization

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, -24(%rbp)
 9       cmpq    $1, %rdi
10       ja      .L2
11       movq    %rdi, %rax
12       jmp     .L3
13  .L2:
14       movq    %rdi, %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    -24(%rbp), %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

Optimization

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

```
 1      .globl fib
 2      .type  fib, @function
 3  fib:
 4      pushq   %rbp
 5      movq    %rsp, %rbp
 6      pushq   %rbx
 7      subq    $24, %rsp
 8      movq    %rdi, -24(%rbp)
 9      cmpq    $1, %rdi
10      ja      .L2
11      movq    %rdi, %rax
12      jmp     .L3
13  .L2:
14      movq    %rdi, %rax
15      subq    $1, %rax
16      movq    %rax, %rdi
17      call    fib
18      movq    %rax, %rbx
19      movq    -24(%rbp), %rax
20      subq    $2, %rax
21      movq    %rax, %rdi
22      call    fib
23      addq    %rbx, %rax
24  .L3:
25      addq    $24, %rsp
26      popq    %rbx
27      popq    %rbp
28      ret
```

n-2

Use the base pointer %rbp

Optimization

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       subq    $24, %rsp
 8       movq    %rdi, %rbp
 9       cmpq    $1, %rdi
10       ja      .L2
11       movq    %rdi, %rax
12       jmp     .L3
13  .L2:
14       movq    %rdi, %rax
15       subq    $1, %rax
16       movq    %rax, %rdi
17       call    fib
18       movq    %rax, %rbx
19       movq    %rbp, %rax
20       subq    $2, %rax
21       movq    %rax, %rdi
22       call    fib
23       addq    %rbx, %rax
24  .L3:
25       addq    $24, %rsp
26       popq    %rbx
27       popq    %rbp
28       ret
```

n-2

Use the base pointer %rbp

Eliminate linkage block

fib(43): 5.49s ⇨ 4.09s ⇨ 3.81s

Optimization

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       movq    %rdi, %rbp
 8       cmpq    $1, %rdi
 9       ja      .L2
10       movq    %rdi, %rax
11       jmp     .L3
12  .L2:
13       movq    %rdi, %rax
14       subq    $1, %rax
15       movq    %rax, %rdi
16       call    fib
17       movq    %rax, %rbx
18       movq    %rbp, %rax
19       subq    $2, %rax
20       movq    %rax, %rdi
21       call    fib
22       addq    %rbx, %rax
23  .L3:
24       popq    %rbx
25       popq    %rbp
26       ret
```

Optimization

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

```
 1      .globl fib
 2      .type  fib, @function
 3  fib:
 4      pushq   %rbp
 5      movq    %rsp, %rbp
 6      pushq   %rbx
 7      movq    %rdi, %rbp
 8      movq    %rdi, %rax
 9      cmpq    $1, %rdi
10      ja      .L2
11      jmp     .L3
12  .L2:
13      subq    $1, %rax
14      movq    %rax, %rdi
15      call    fib
16      movq    %rax, %rbx
17      movq    %rbp, %rax
18      subq    $2, %rax
19      movq    %rax, %rdi
20      call    fib
21      addq    %rbx, %rax
22  .L3:
23      popq    %rbx
24      popq    %rbp
25      ret
```

Optimization

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

```
 1       .globl fib
 2       .type  fib, @function
 3  fib:
 4       pushq   %rbp
 5       movq    %rsp, %rbp
 6       pushq   %rbx
 7       movq    %rdi, %rbp
 8       movq    %rdi, %rax
 9       cmpq    $1, %rdi
10       jbe     .L3
11  .L2:
12       subq    $1, %rax
13       movq    %rax, %rdi
14       call    fib
15       movq    %rax, %rbx
16       movq    %rbp, %rax
17       subq    $2, %rax
18       movq    %rax, %rdi
19       call    fib
20       addq    %rbx, %rax
21  .L3:
22       popq    %rbx
23       popq    %rbp
24       ret
```

gcc –O2 ⇨ 2.06s

Eliminate branches

fib(43): 5.49s ⇨ 4.09s ⇨ 3.81s ⇨ 3.23s

Simple Optimization Strategies

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

Caveat: "Optimization" or not is system-dependent — remember, these optimizations reduce work, but they may not reduce running time.

Disclaimer: Your results may vary. 

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:
```

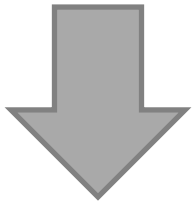
Compiling `while` Loops

```
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;
 }
```

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]?

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.

More C/C++ Constructs

Arrays of structs

Structs of arrays

Function pointers

Bit fields in arrays

Objects, virtual function tables

Memory-management techniques

References

Gcc assembler manual:

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

Quick reference on assembly instructions:

http://en.wikipedia.org/wiki/X86_instruction_listings

http://www.x86-64.org/documentation/assembly.html

Full details:

Intel Software Developer Manuals (on Stellar)

C subroutine linkage:

System V Application Binary Interface (on Stellar)

Compiler intrinsic for inline assembly:

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html