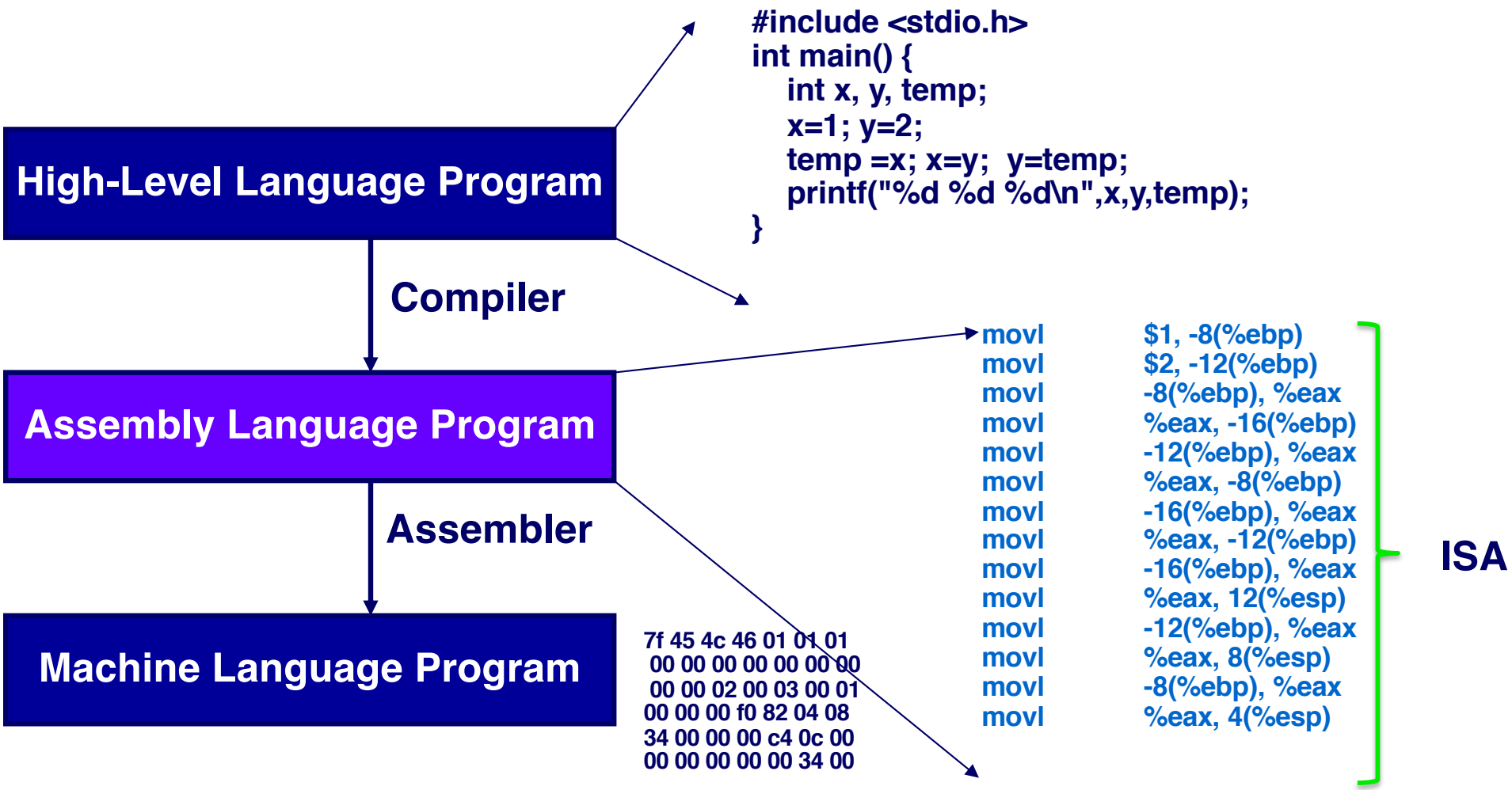
211: Computer Architecture
Fall 2019

Instructor: Prof. Santosh Nagarakatte

Topics:

- Hardware-Software Interface
- Assembly Programming
  - Reading: Chapter 3

Programming Meets Hardware



How do you get performance?

Performance with Programs

(1) Program: Data structures + algorithms

(2) Compiler translates code

(3) Instruction set architecture

(4) Hardware Implementation

Instruction Set Architecture

(1) Set of instructions that the CPU can execute

   (1) What instructions are available?

   (2) How the instructions are encoded? Eventually everything is binary.

(2) State of the system (Registers + memory state + program counter)

   (1) What instruction is going to execute next

   (2) How many registers? Width of each register?

   (3) How do we specify memory addresses?

      - Addressing modes

(3) Effect of instruction on the state of the system

IA32 (X86 ISA)

There are many different assembly languages because they are processor-specific

- IA32 (x86)
  - x86-64 for new 64-bit processors
  - IA-64 radically different for Itanium processors
  - Backward compatibility: instructions added with time
- PowerPC
- MIPS

We will focus on IA32/x86-64 because you can generate and run on iLab machines (as well as your own PC/laptop)

- IA32 is also dominant in the market although smart phone, eBook readers, etc. are changing this

Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
8086	1978	29K	5-10

- First 16-bit Intel processor. Basis for IBM PC & DOS
- 1MB address space

386	1985	275K	16-33

- First 32 bit Intel processor , referred to as IA32
- Added “flat addressing”, capable of running Unix

Pentium 4E	2004	125M	2800-3800

- First 64-bit Intel x86 processor, referred to as x86-64

Core 2	2006	291M	1060-3500

- First multi-core Intel processor

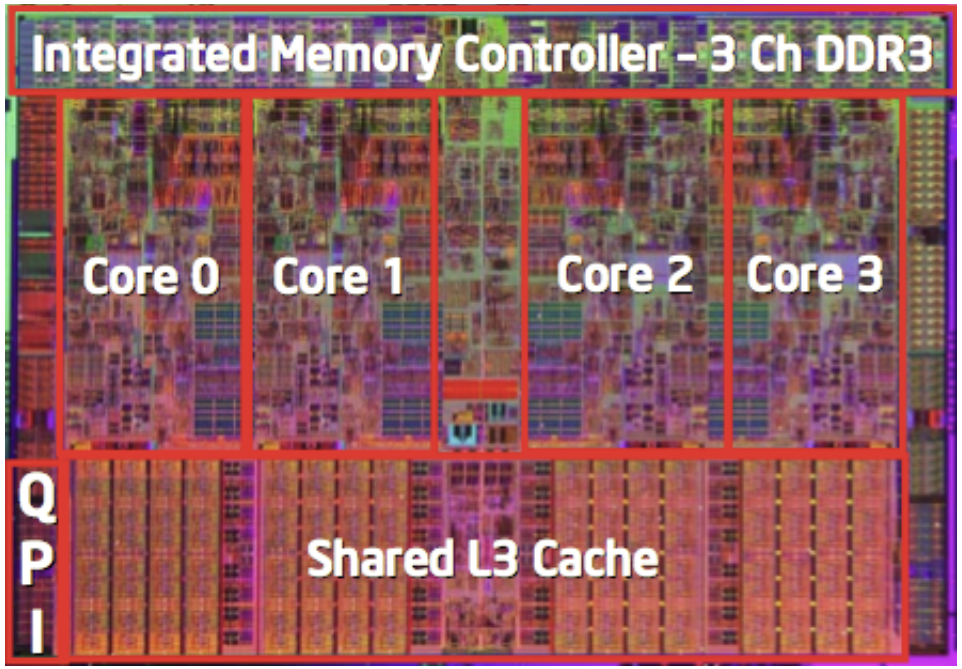
Core i7	2008	731M	1700-3900

- Four cores (our shark machines)

Intel x86 Processors, cont.

Machine Evolution

■ 386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2001	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M



Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores

Core i7 Broadwell Processor

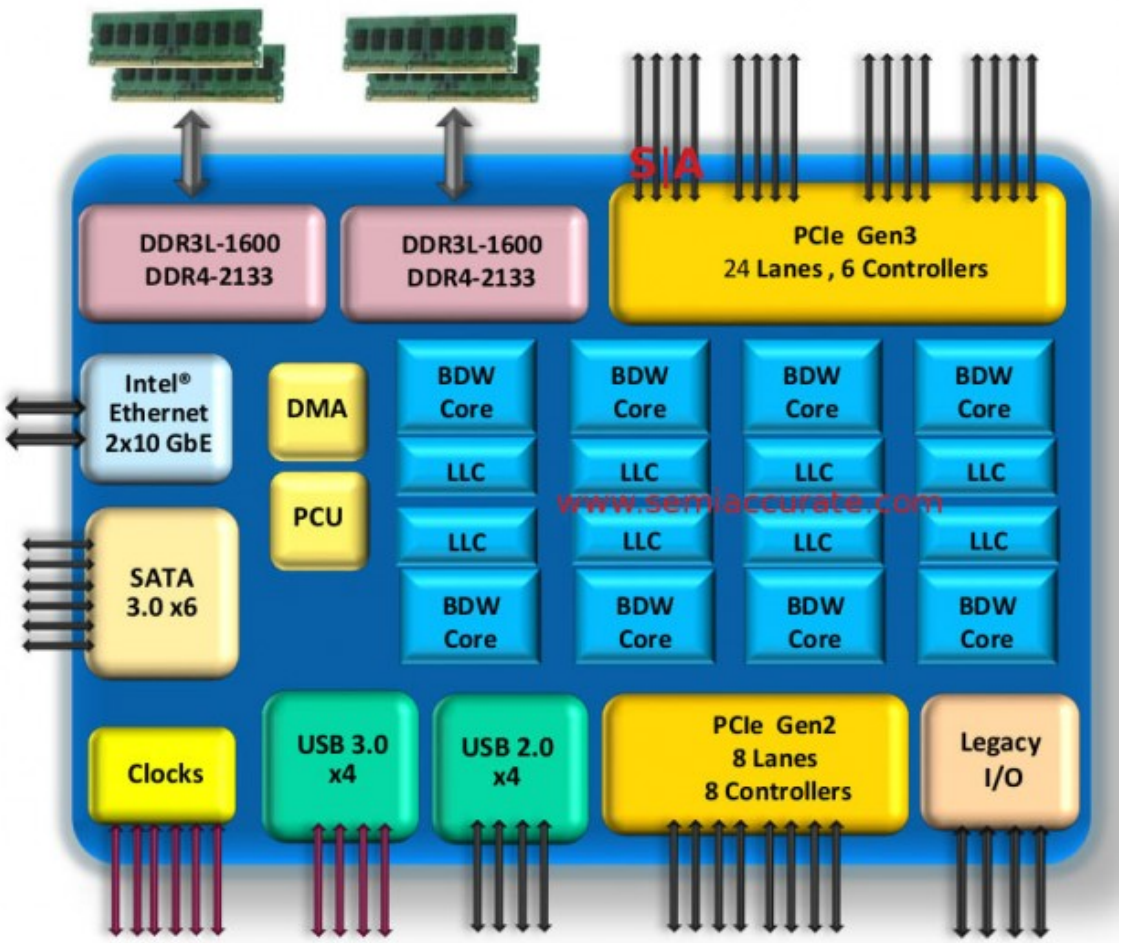
- Core i7 Broadwell 2015

Desktop Model

- 4 cores
- Integrated graphics
- 3.3-3.8 GHz
- 65W

Server Model

- 8 cores
- Integrated I/O
- 2-2.6 GHz
- 45W



CISC vs RISC

CISC: complex instructions : eg X86

- Instructions such as strcpy/AES and others
- Reduces code size
- Hardware implementation complex?

RISC: simple instructions: eg Alpha

- Instructions are simple add/ld/st
- Increases code size
- Hardware implementation simple?

Aside About Implementation of x86

About 30 years ago, the instruction set actually reflected the processor hardware

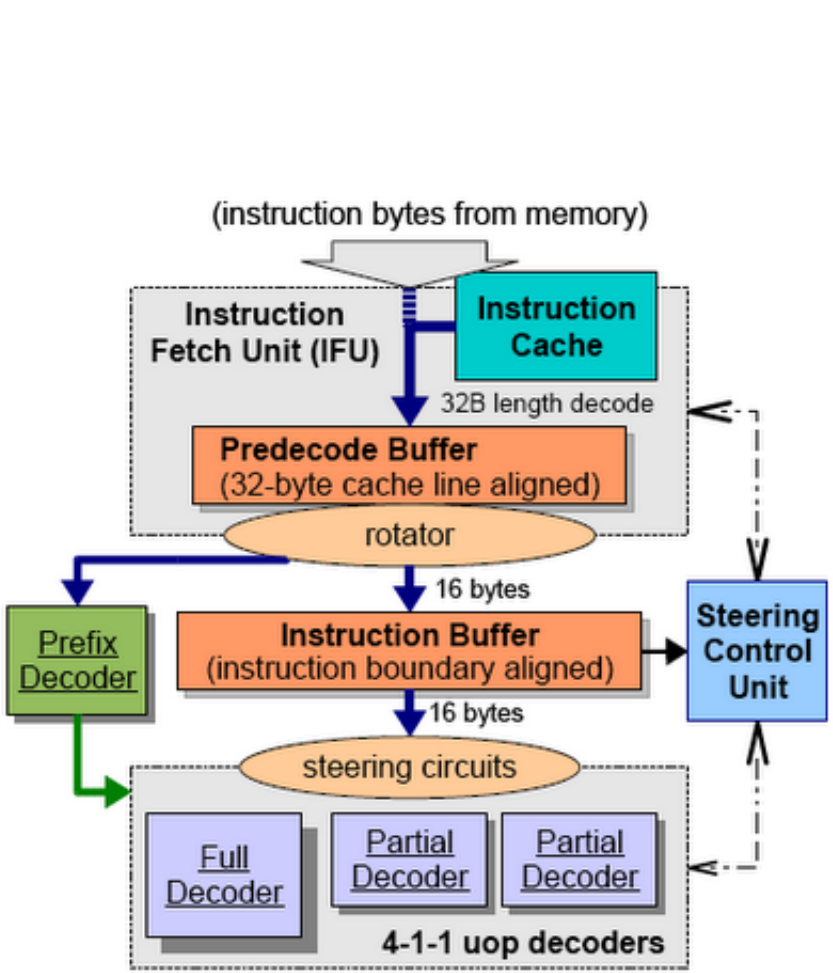
- E.g., the set of registers in the instruction set is actually what was present in the processor

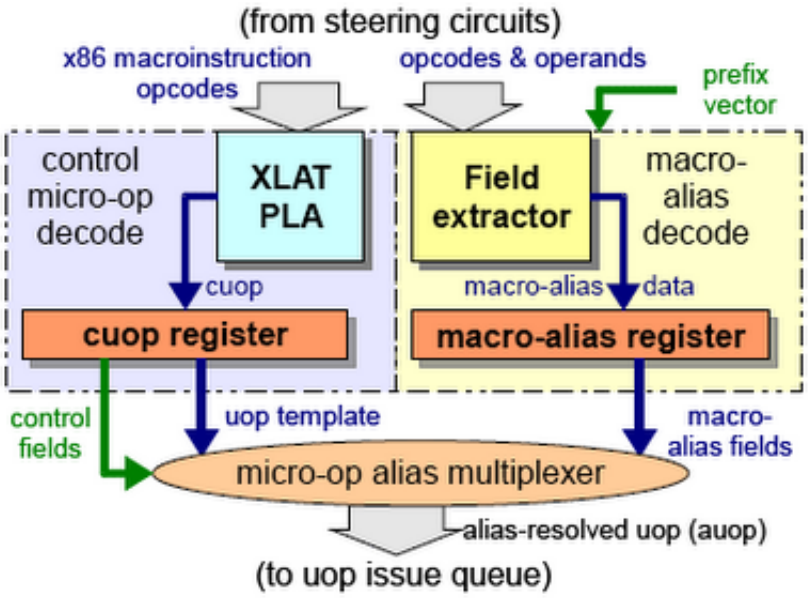
As hardware advanced, industry faced with choice

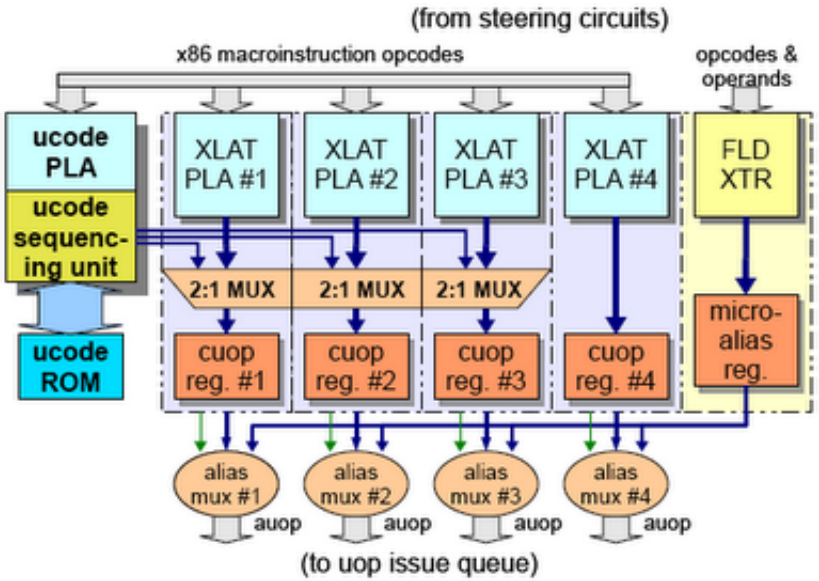
- Change the instruction set: bad for backward compatibility
- Keep the instruction set: harder to exploit hardware advances
  - Example: many more registers but only small set introduced circa 1980

Starting with the P6 (PentiumPro), IA32 actually got implemented by Intel using an “interpreter” that translates IA32 instructions into a simpler “micro” instruction set

P6 Decoder/Interpreter







Assembly Programming

Brief tour through assembly language programming

Why?

- Machine interface: where software meets hardware
- To understand how the hardware works, we have to understand the interface that it exports

Why not binary language?

- Much easier for humans to read and reason about
- Major differences:
  - Human readable language instead of binary sequences
  - Relative instead of absolute addresses

Definitions

Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.

- Examples: instruction set specification, registers.

Microarchitecture: Implementation of the architecture.

- Examples: cache sizes and core frequency.

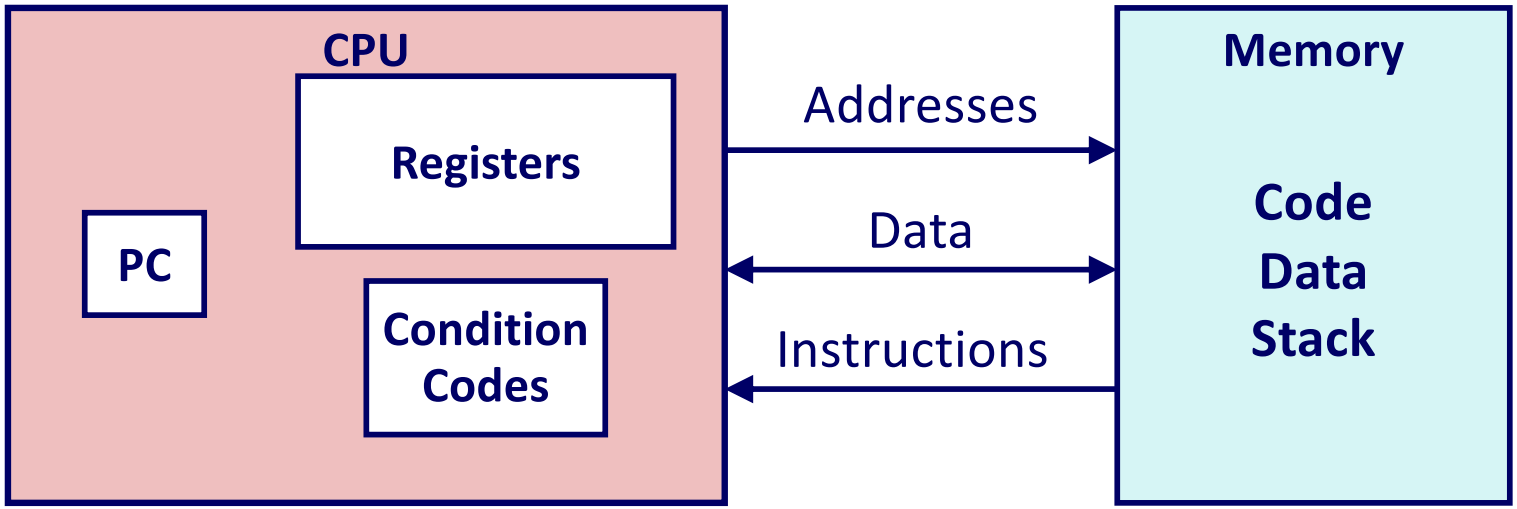
Code Forms:

- Machine Code: The byte-level programs that a processor executes
- Assembly Code: A text representation of machine code

Example ISAs:

- Intel: x86, IA32, Itanium, x86-64
- ARM: Used in almost all mobile phones

Assembly/Machine Code View



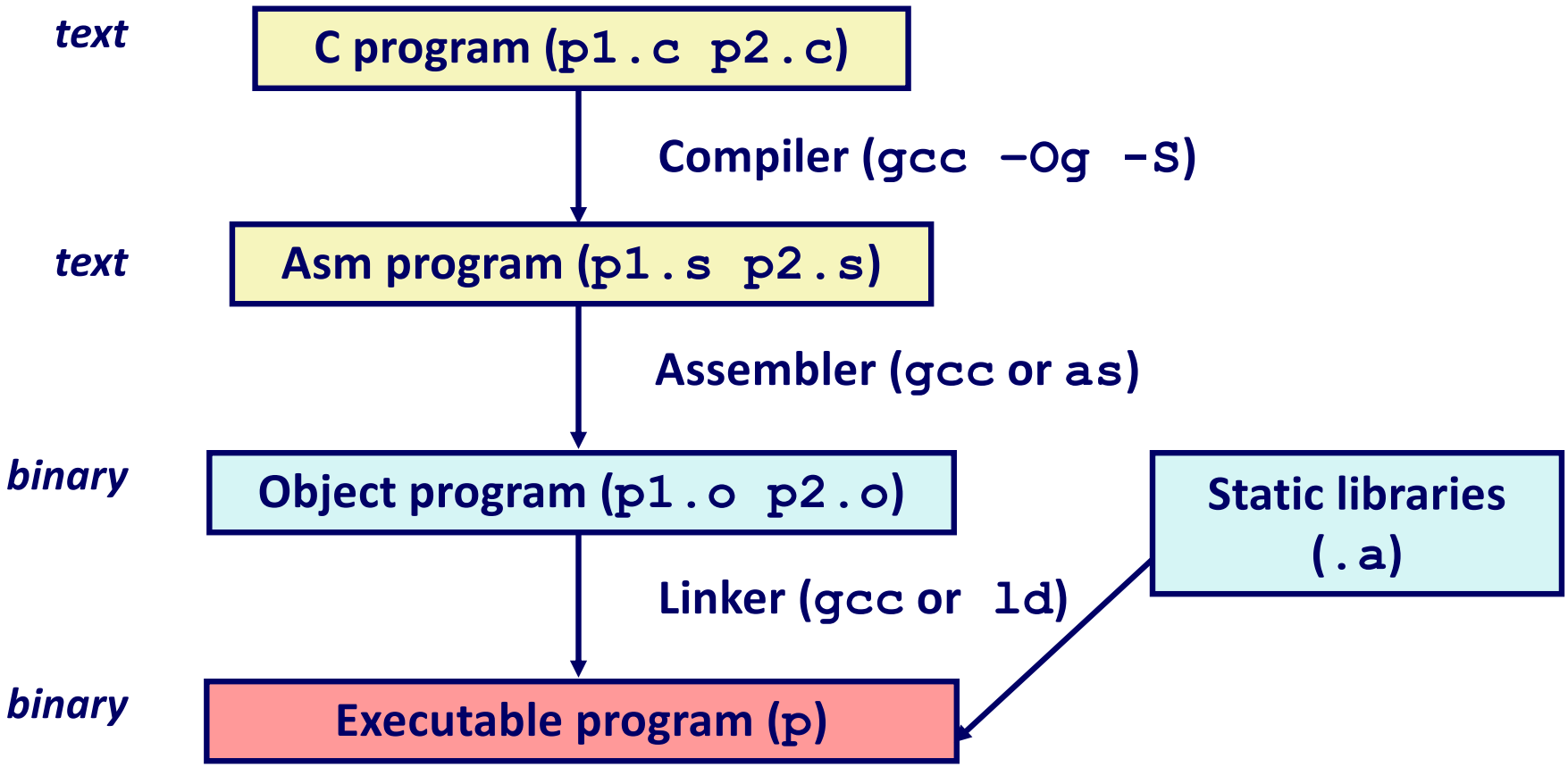
Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc –Og p1.c p2.c -o p`
  - Use basic optimizations (`–Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

On iLab machines, you will obtain an assembly file with command

```
gcc –Og –S sum.c
```

Produces file `sum.s`

Warning: Will get very different results on different machines due to different versions of gcc and different compiler settings.

Assembly Characteristics: Data Types

"Integer" data of 1, 2, 4, or 8 bytes

- Data values
- Addresses (untyped pointers)

Floating point data of 4, 8, or 10 bytes

Code: Byte sequences encoding series of instructions

No aggregate types such as arrays or structures

- Just contiguously allocated bytes in memory

Assembly Characteristics: Operations

Perform arithmetic function on register or memory data

Transfer data between memory and register

- Load data from memory into register
- Store register data into memory

Transfer control

- Unconditional jumps to/from procedures
- Conditional branches

Object Code

Code for `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- **Total of 14 bytes**
- **Each instruction 1, 3, or 5 bytes**
- **Starts at address `0x0400595`**

Assembler

- Translates `.s` into `.o`
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

Linker

- Resolves references between files
- Combines with static run-time libraries
  - E.g., code for **`malloc`**, **`printf`**
- Some libraries are *dynamically linked*
  - Linking occurs when program begins execution

Machine Instruction Example

```
*dest = t;
```

C Code

- Store value `t` where designated by **`dest`**

```
movq %rax, (%rbx)
```

Assembly

- Move 8-byte value to memory
  - Quad words in x86-64 parlance
- Operands:
  - **`t:`** Register **`%rax`**
  - **`dest:`** Register **`%rbx`**
  - **`*dest:`** Memory **`M[%rbx]`**

```
0x40059e:  48 89 03
```

Object Code

- 3-byte instruction
- Stored at address **`0x40059e`**

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

Disassembler

`objdump -d sum`

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

Within gdb Debugger

`gdb sum`

`disassemble sumstore`

- Disassemble procedure

`x/14xb sumstore`

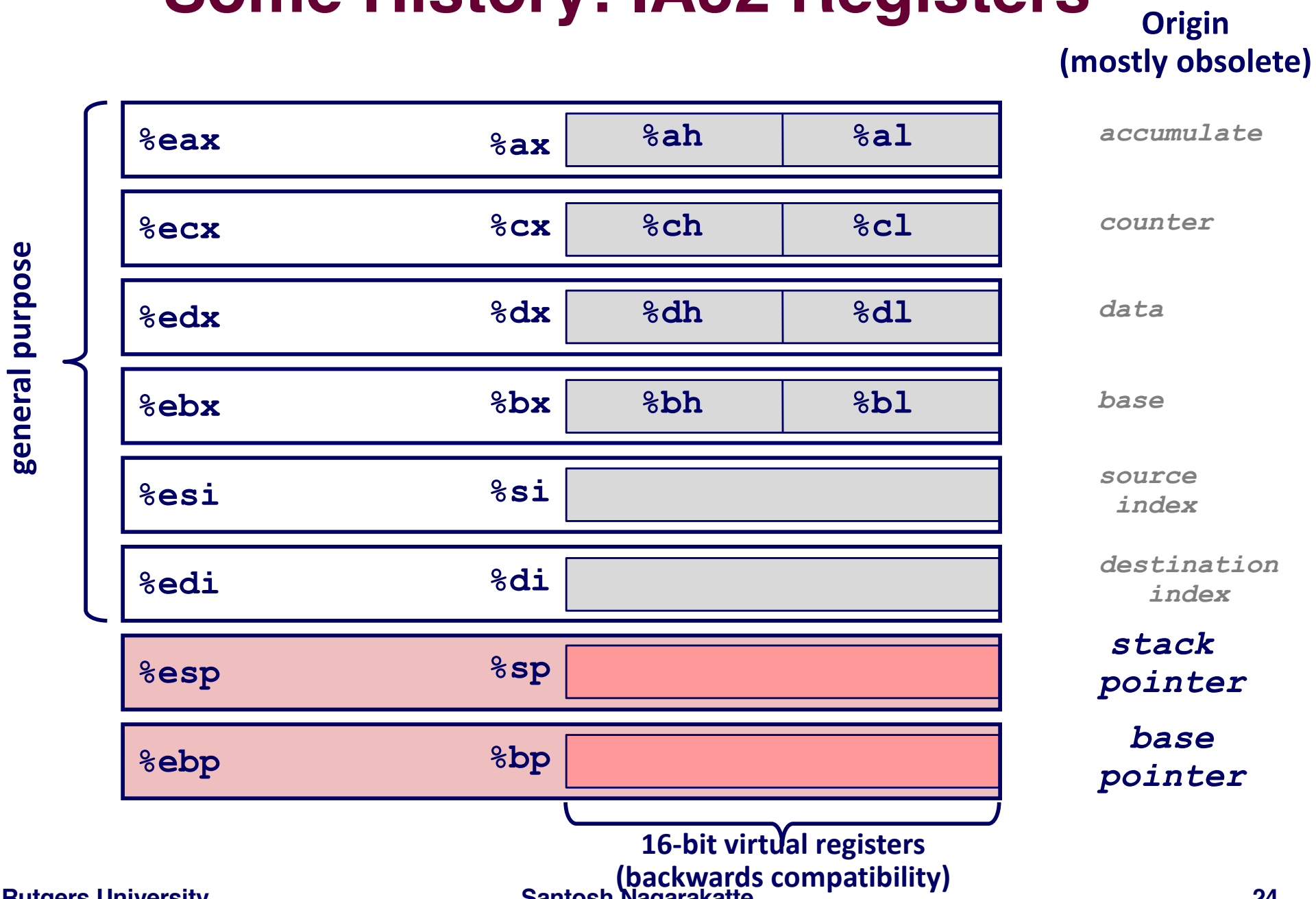
- Examine the 14 bytes starting at `sumstore`

x86-64 Integer Registers

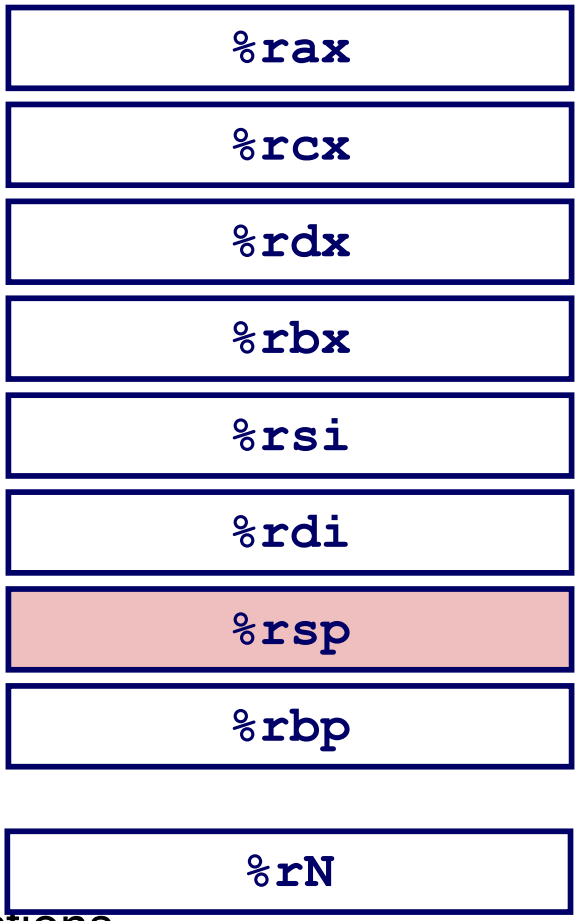
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Some History: IA32 Registers



Moving Data



Moving Data

`movq` *Source*, *Dest*:

Operand Types

- ***Immediate:*** Constant integer data
  - Example: **`$0x400`**, **`$-533`**
  - Like C constant, but prefixed with **`'$'`**
  - Encoded with 1, 2, or 4 bytes
- ***Register:*** One of 16 integer registers
  - Example: **`%rax`**, **`%r13`**
  - But **`%rsp`** reserved for special use
  - Others have special uses for particular instructions
- ***Memory:*** 8 consecutive bytes of memory at address given by register
  - Simplest example: **`(%rax)`**
  - Various other “address modes”

`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

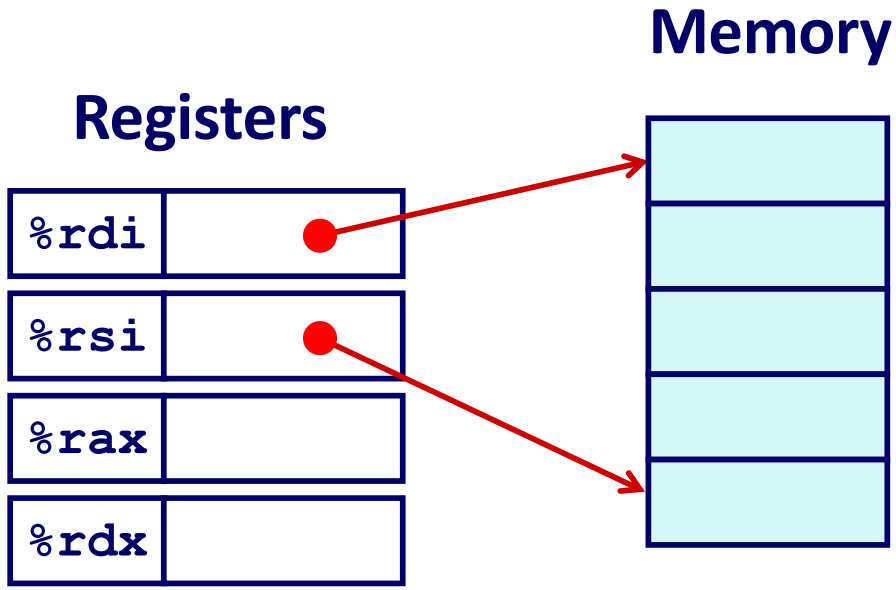
Example of Simple Addressing Modes

```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

Understanding Swap()

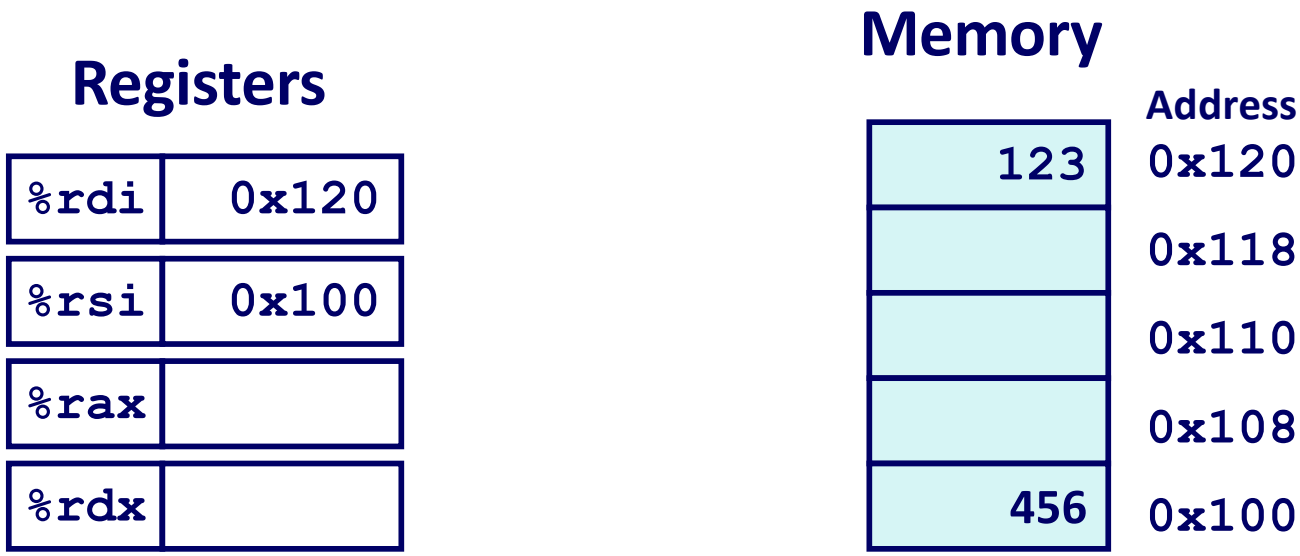
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

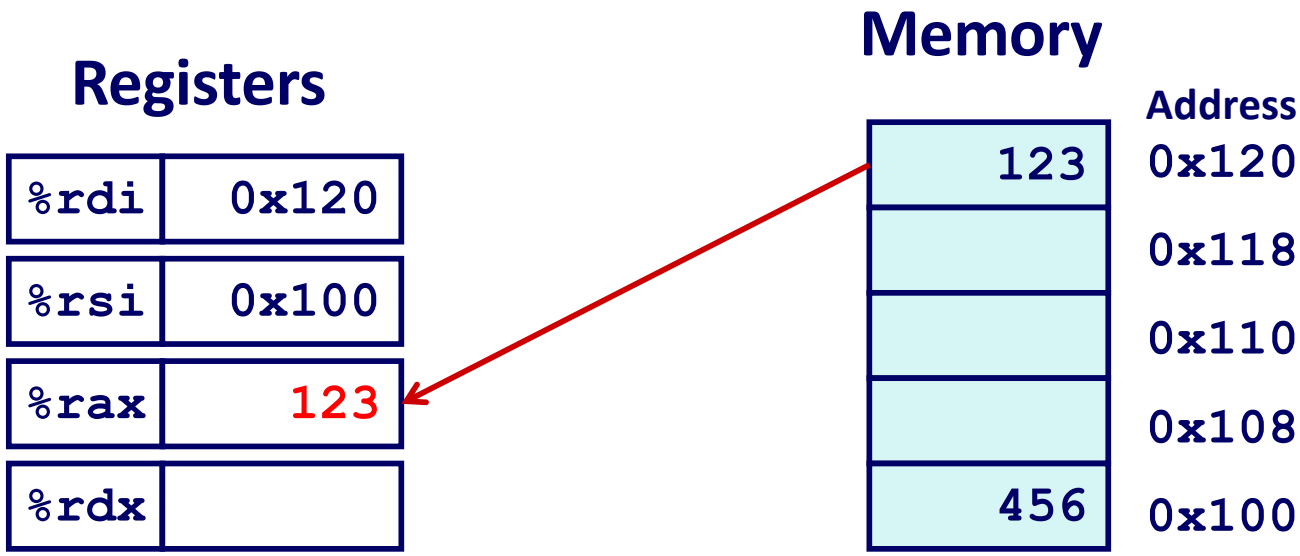
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding `Swap()`



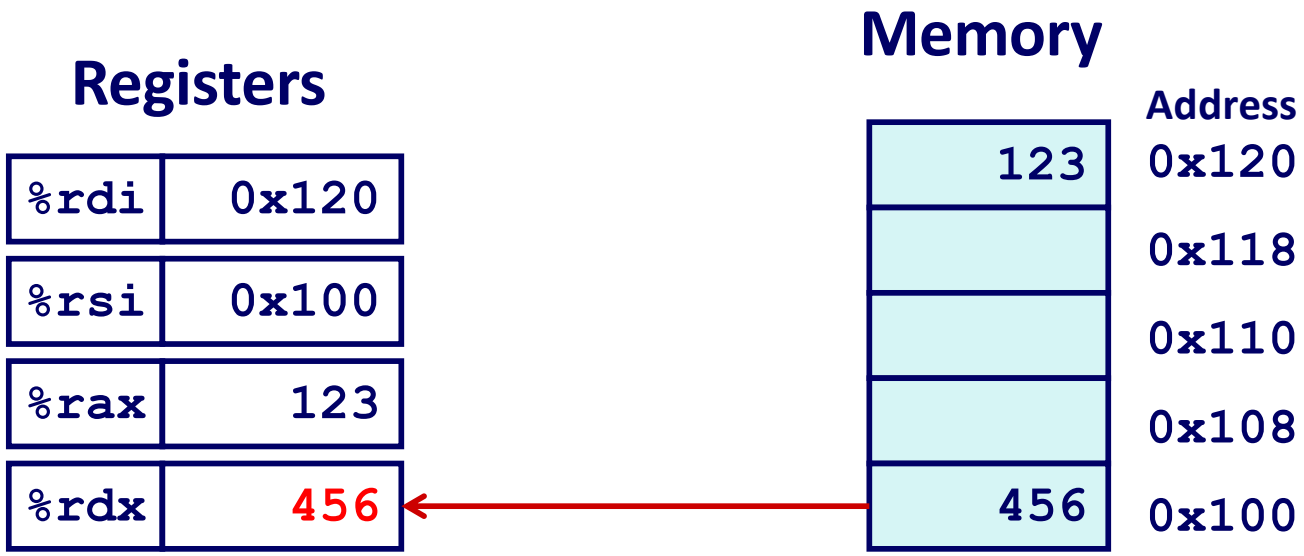

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



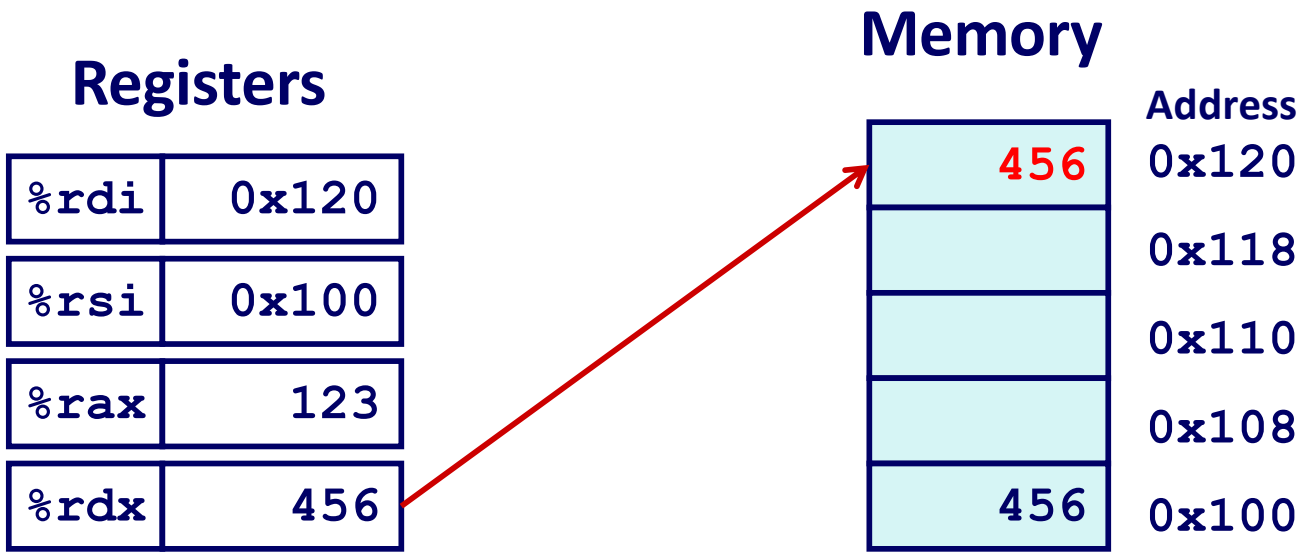

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



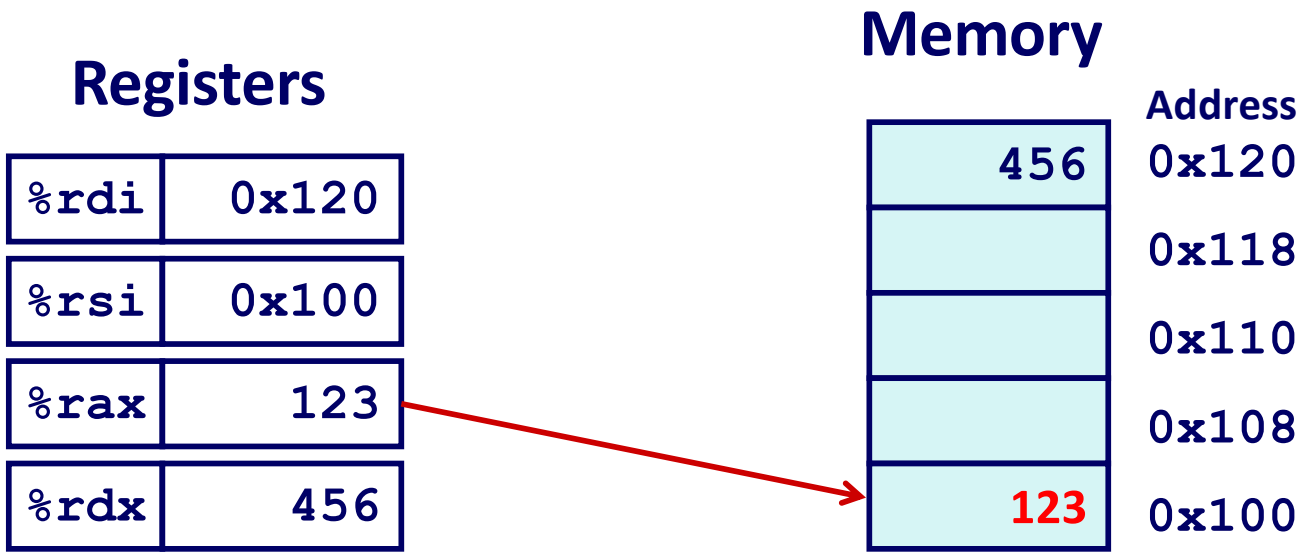

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Simple Memory Addressing Modes

Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Complete Memory Addressing Modes

Most General Form

D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Special Cases

(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]

D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]

(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Address Computation Instruction

`leaq` Src, Dst

- Src is address mode expression
- Set Dst to address denoted by expression

Uses

- Computing addresses without a memory reference
  - E.g., translation of **`p = &x[i];`**
- Computing arithmetic expressions of the form x + k*y
  - k = 1, 2, 4, or 8

Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Some Arithmetic Operations

Two Operand Instructions:

Format		Computation	
`addq`	Src,Dest	Dest = Dest + Src	
`subq`	Src,Dest	Dest = Dest – Src	
`imulq`	Src,Dest	Dest = Dest * Src	
`salq`	Src,Dest	Dest = Dest << Src	Also called shlq
`sarq`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrq`	Src,Dest	Dest = Dest >> Src	Logical
`xorq`	Src,Dest	Dest = Dest ^ Src	
`andq`	Src,Dest	Dest = Dest & Src	
`orq`	Src,Dest	Dest = Dest | Src	

Watch out for argument order!

No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

One Operand Instructions

`incq`	Dest	Dest = Dest + 1
`decq`	Dest	Dest = Dest – 1
`negq`	Dest	Dest = – Dest
`notq`	Dest	Dest = ~Dest

See book for more instructions

Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax
   addq    %rdx, %rax
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx
   leaq    4(%rdi,%rdx), %rcx
   imulq   %rcx, %rax
   ret
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	**t1**, **t2**, **rval**
%rdx	**t4**
%rcx	**t5**

Control Flow/Conditionals

How do we represent conditionals in assembly?

A conditional branch can implement all control flow constructs in higher level language

- Examples: if/then, while, for

A unconditional branch for constructs like break/ continue

Conditionals/Control Flow

Control: Condition codes

Conditional branches

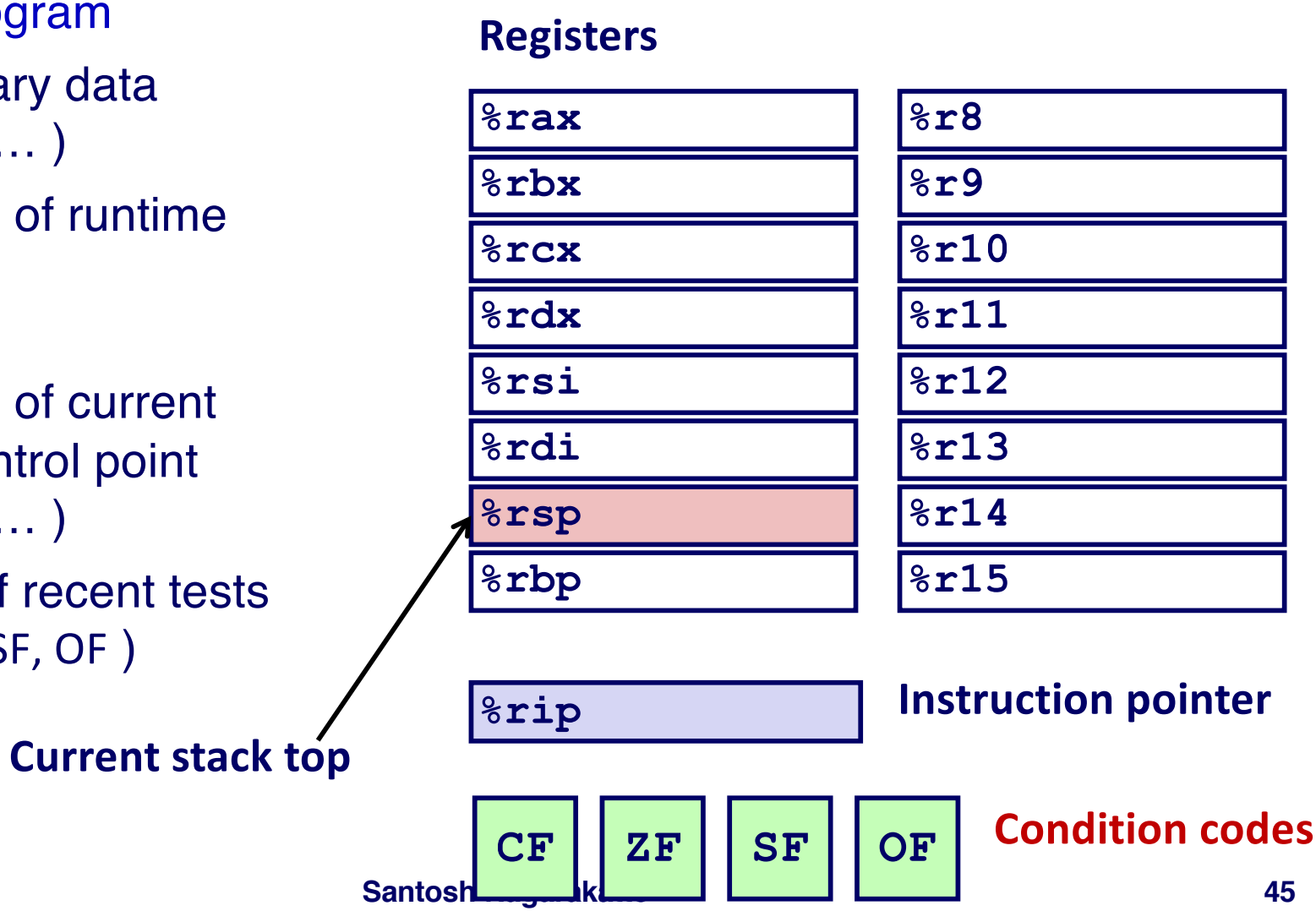
Loops

Switch Statements

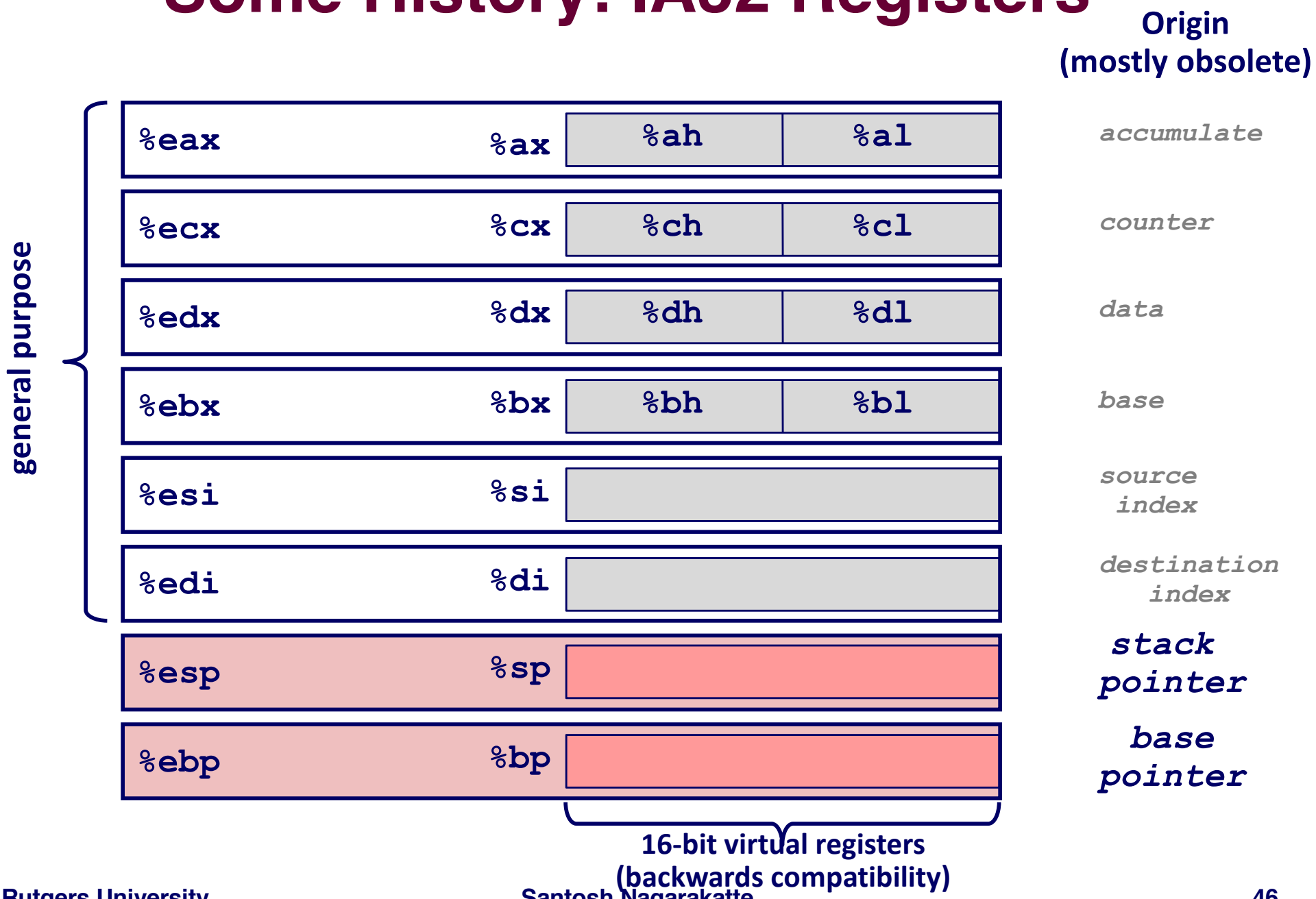
Processor State (x86-64, Partial)

Information about currently executing program

- Temporary data (**%rax**, …)
- Location of runtime stack (**%rsp**)
- Location of current code control point (**%rip**, …)
- Status of recent tests (CF, ZF, SF, OF)



Some History: IA32 Registers



Condition Codes (Implicit Setting)

Single bit registers

- CF Carry Flag (for unsigned)
- SF Sign Flag (for signed)
- ZF Zero Flag
- OF Overflow Flag (for signed)

Implicitly set (think of it as side effect) by arithmetic operations

Example: **`addq`** Src,Dest **`t = a+b`**

CF set if carry out from most significant bit (unsigned overflow)

ZF set if **`t == 0`**

SF set if **`t < 0`** (as signed)

OF set if two's-complement (signed) overflow

```
(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
```

Not set by **`leaq`** instruction

Condition Codes (Explicit Setting: Compare)

Explicit Setting by Compare Instruction

- **`cmpq`** Src2, Src1
- **`cmpq b,a`** like computing **`a-b`** without setting destination

- CF set if carry out from most significant bit (used for unsigned comparisons)
- ZF set if **`a == b`**
- SF set if **`(a-b) < 0`** (as signed)
- OF set if two's-complement (signed) overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Condition Codes (Explicit Setting: Test)

Explicit Setting by Test instruction

- **`testq`** Src2, Src1
  - **`testq b,a`** like computing **`a&b`** without setting destination

- Sets condition codes based on value of Src1 & Src2
- Useful to have one of the operands be a mask

- ZF set when **`a&b == 0`**
- SF set when **`a&b < 0`**

Reading Condition Codes

SetX Instructions

- Set low-order byte of destination to 0 or 1 based on combinations of condition codes
- Does not alter remaining 7 bytes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Reading Condition Codes (Cont.)

SetX Instructions:

- Set single byte based on combination of condition codes

One of addressable byte registers

- Does not alter remaining bytes
- Typically use **`movzbl`** to finish job
  - 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

Jumping

jX Instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Conditional Branch Example (Old Style)

Generation

```
gcc -Og -S -fno-if-conversion control.c
```

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

Expressing with Goto Code

C allows `goto` statement

Jump to position designated by label

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
  (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

Conditional Move Instructions

- Instruction supports:

  if (Test) Dest ← Src
- Supported in post-1995 x86 processors
- GCC tries to use them
  - But, only when known to be safe

Why?

- Branches are very disruptive to instruction flow through pipelines
- Conditional moves do not require control transfer

C Code

```
val = Test
    ? Then_Expr
    : Else_Expr;
```

Goto Version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

Conditional Move Example

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
    movq    %rdi, %rax  # x
    subq    %rsi, %rax  # result = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx  # eval = y-x
    cmpq    %rsi, %rdi  # x:y
    cmovle  %rdx, %rax  # if <=, result = eval
    ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

Both values get computed

Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- **May have undesirable effects**

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- **Must be side-effect free**

"Do-While" Loop Example

C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Count number of 1's in argument `x` ("popcount")

Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

Goto Version

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
   movl    $0, %eax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx
   andl    $1, %edx    #  t = x & 0x1
   addq    %rdx, %rax  #  result += t
   shrq    %rdi        #  x >>= 1
   jne     .L2         #  if (x) goto loop
   rep; ret
```

General "Do-While" Translation

C Code

```
do
   Body
   while (Test);
```

Body:

```
{
   Statement1;
   Statement2;
      …
   Statementn;
}
```

Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

General "While" Translation #1

"Jump-to-middle" translation

Used with `-Og`

While version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

While Loop Example #1

C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

Jump to Middle

```
long pcount_goto_jtm
   (unsigned long x) {
   long result = 0;
   goto test;
 loop:
   result += x & 0x1;
   x >>= 1;
 test:
   if(x) goto loop;
   return result;
}
```

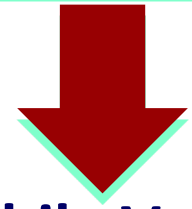
Compare to do-while version of function

Initial goto starts loop at test

General "While" Translation #2

While version

```
while (Test)
  Body
```

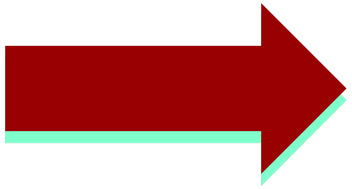


"Do-while" conversion

Used with –O1

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```



Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

While Loop Example #2

C Code

```
long pcount_while
   (unsigned long x) {
   long result = 0;
   while (x) {
     result += x & 0x1;
     x >>= 1;
   }
   return result;
}
```

Do-While Version

```
long pcount_goto_dw
   (unsigned long x) {
   long result = 0;
   if (!x) goto done;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
 done:
   return result;
}
```

Compare to do-while version of function

Initial conditional guards entrance to loop

“For” Loop Form

General Form

```
for (Init; Test; Update)
    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

“For” Loop → While Loop

For Version

```
for (Init; Test; Update)
        Body
```

While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

"For" Loop Do-While Conversion

C Code

```
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Initial test can be optimized away

Goto Version

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {
    unsigned bit =                Body
      (x >> i) & 0x1;
    result += bit;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

```
long switch_eg
   (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch Statement Example

Multiple case labels

- Here: 5 & 6

Fall through cases

- Here: 2

Missing cases

- Here: 4

Jump Table Structure

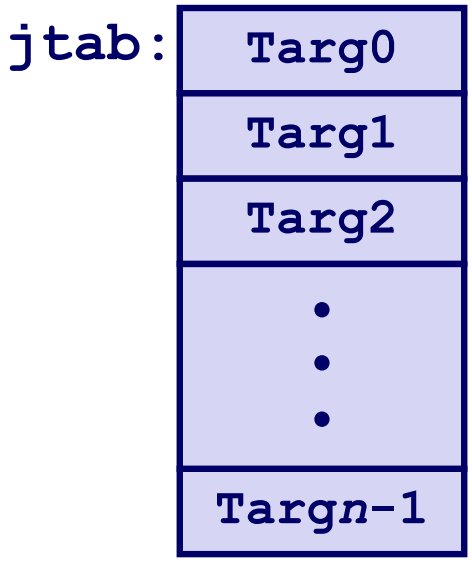
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

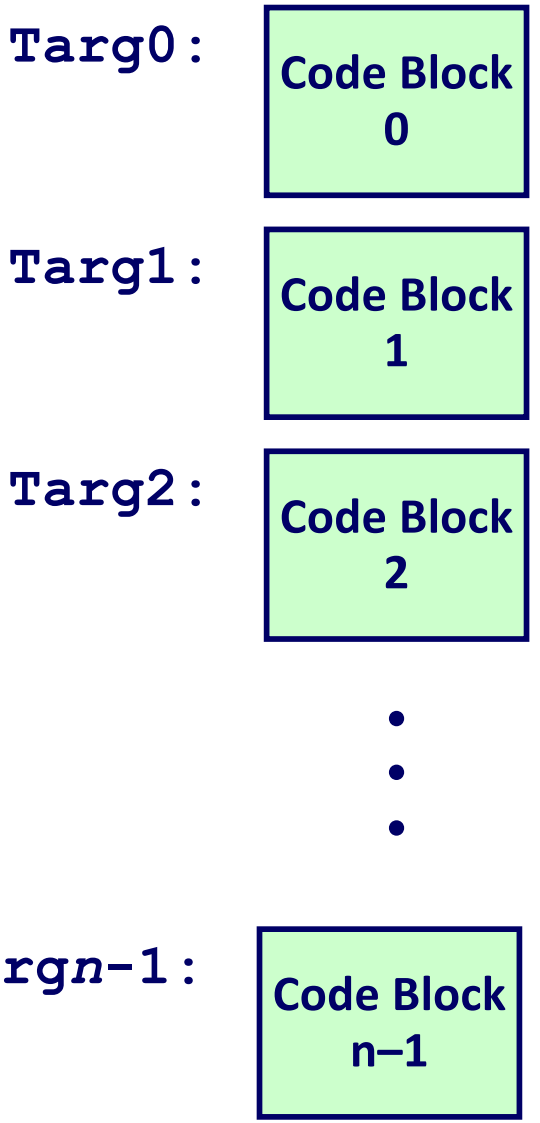
Translation (Extended C)

```
goto *JTab[x];
```

Jump Table



Jump Targets



Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja      .L8
    jmp     *.L4(,%rdi,8)
```

What range of values takes default?

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Note that w not initialized here

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

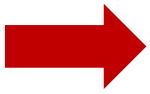
Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Indirect jump



Assembly Setup Explanation

Table Structure

- Each target requires 8 bytes
- Base address at `.L4`

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Jumping

- Direct: `jmp .L8`
- Jump target is denoted by label `.L8`

- Indirect: `jmp *.L4(,%rdi,8)`
- Start of jump table: `.L4`
- Must scale by factor of 8 (addresses are 8 bytes)
- Fetch target from effective Address `.L4 + x*8`
  - Only for $0 \le x \le 6$

Jump Table

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

```
switch(x) {
case 1:         // .L3
    w = y*z;
    break;
case 2:         // .L5
    w = y/z;
    /* Fall Through */
case 3:         // .L9
    w += z;
    break;
case 5:
case 6:         // .L7
    w -= z;
    break;
default:        // .L8
    w = 2;
}
```

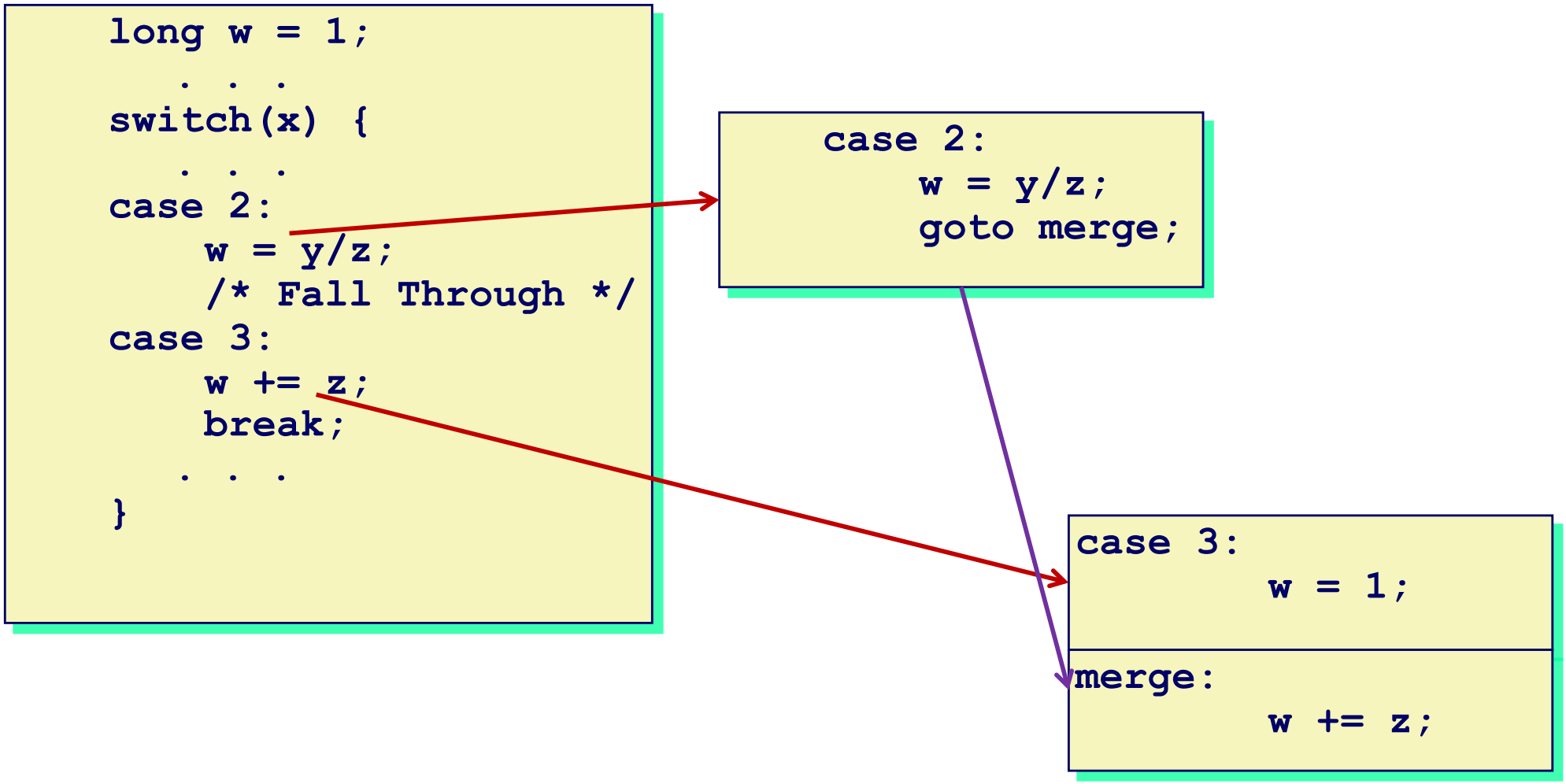
Code Blocks (x == 1)

```
  switch(x) {
  case 1:        // .L3
         w = y*z;
         break;
    . . .
}
```

```
.L3:
   movq     %rsi, %rax   # y
   imulq    %rdx, %rax   # y*z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Handling Fall-Through



Code Blocks (x == 2, x == 3)

```
    long w = 1;
       . . .
    switch(x) {
       . . .
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
       . . .
    }
```

```
.L5:                            # Case 2
   movq     %rsi, %rax
   cqto
   idivq    %rcx               #  y/z
   jmp      .L6                #  goto merge
.L9:                            # Case 3
   movl     $1, %eax           #  w = 1
.L6:                            # merge:
   addq     %rcx, %rax         #  w += z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Code Blocks (x == 5, x == 6, default)

```
  switch(x) {
    . . .
    case 5:  // .L7
    case 6:  // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:                    # Case 5,6
  movl   $1, %eax       #  w = 1
  subq   %rdx, %rax     #  w -= z
  ret
.L8:                    # Default:
  movl   $2, %eax       #  2
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Mechanisms in Procedures

Passing control

- To beginning of procedure code
- Back to return point

Passing data

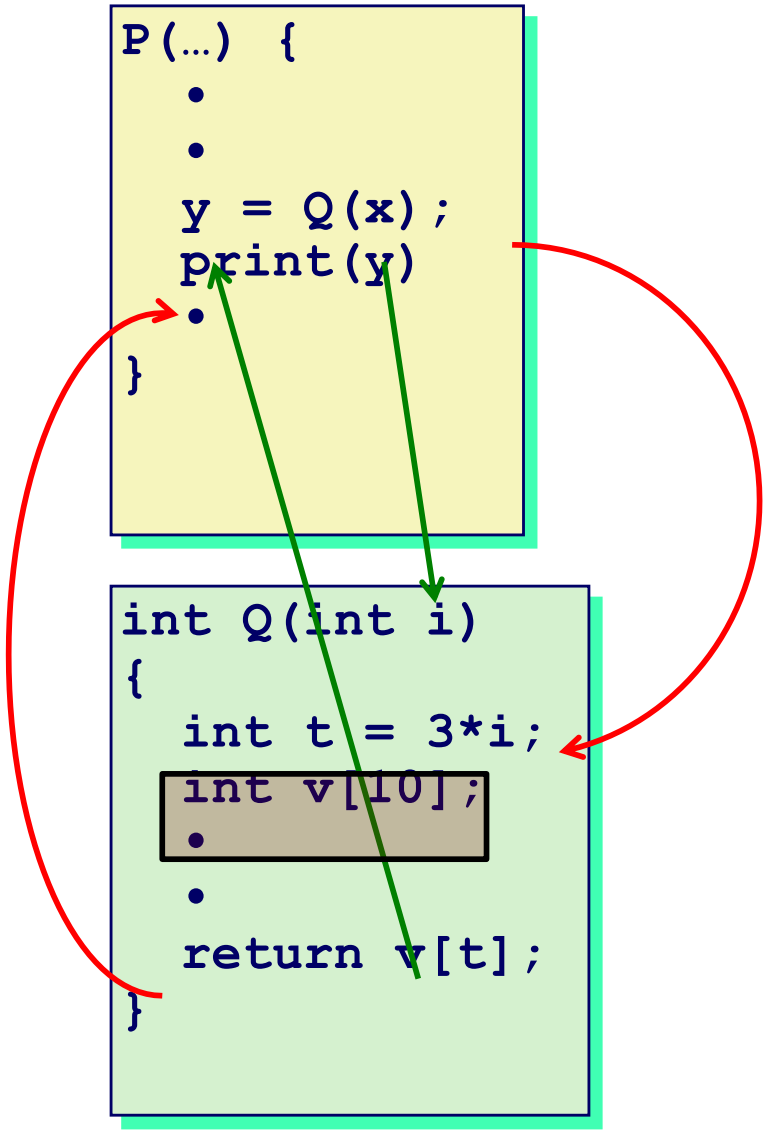
- Procedure arguments
- Return value

Memory management

- Allocate during procedure execution
- Deallocate upon return

Mechanisms all implemented with machine instructions

x86-64 implementation of a procedure uses only those mechanisms required



Agenda with Procedures

Things to consider with procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Illustration of Recursion

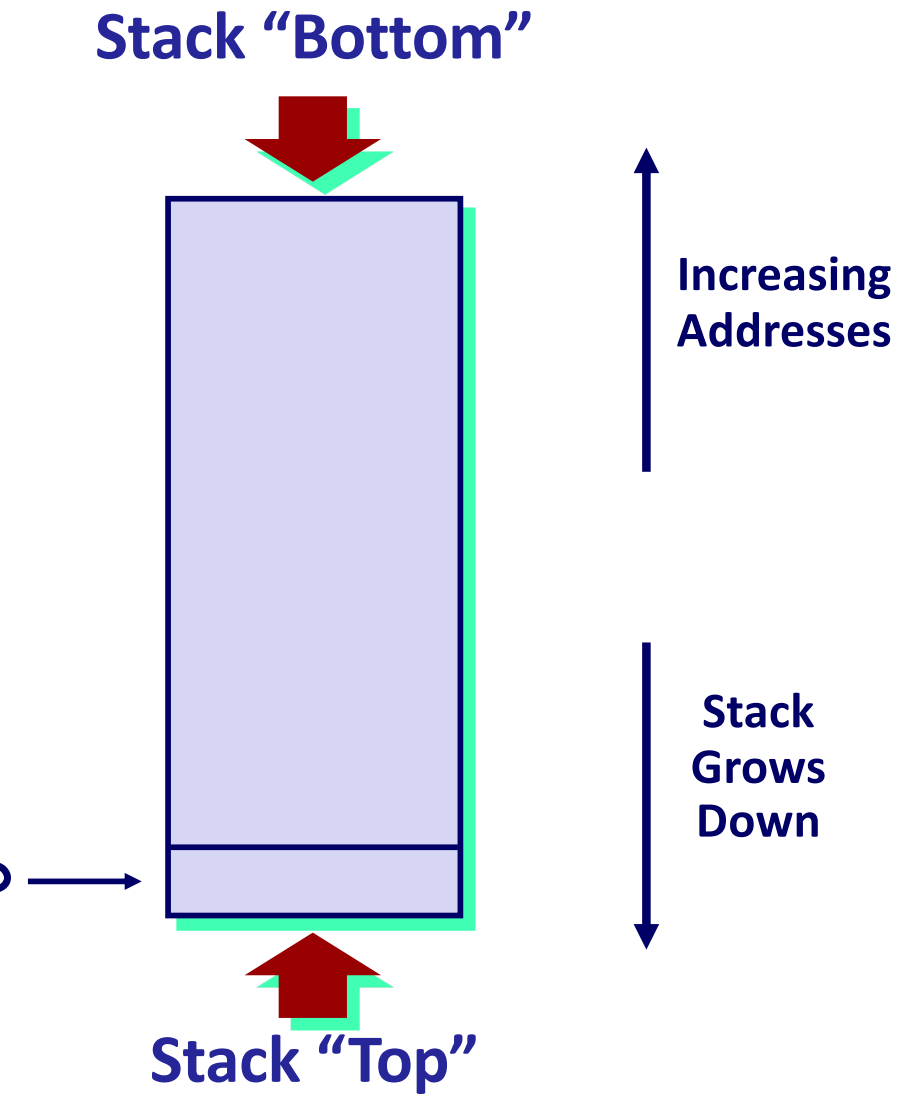
x86-64 Stack

Region of memory managed with stack discipline

Grows toward lower addresses

Register `%rsp` contains lowest stack address

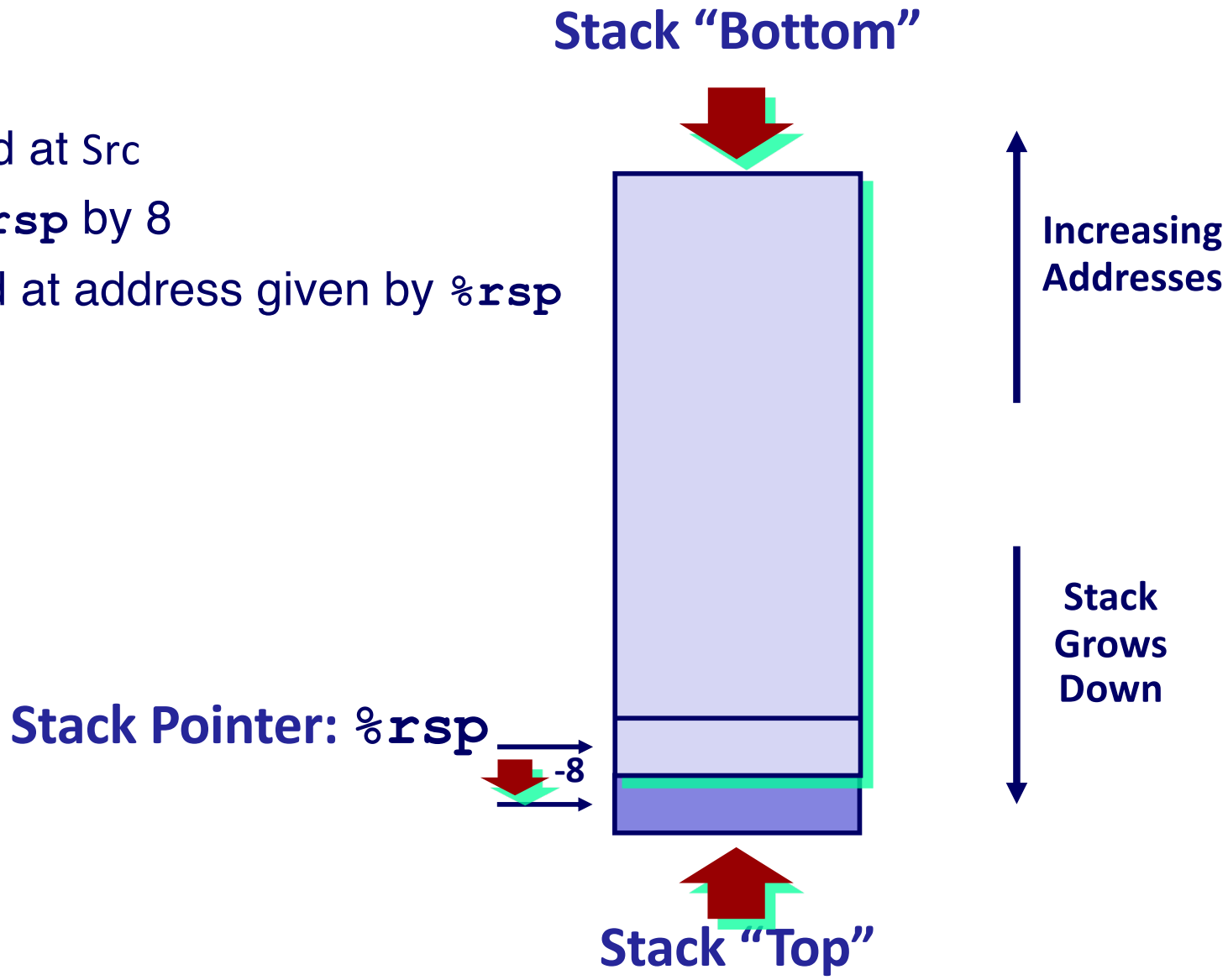
- address of "top" element



x86-64 Stack: Push

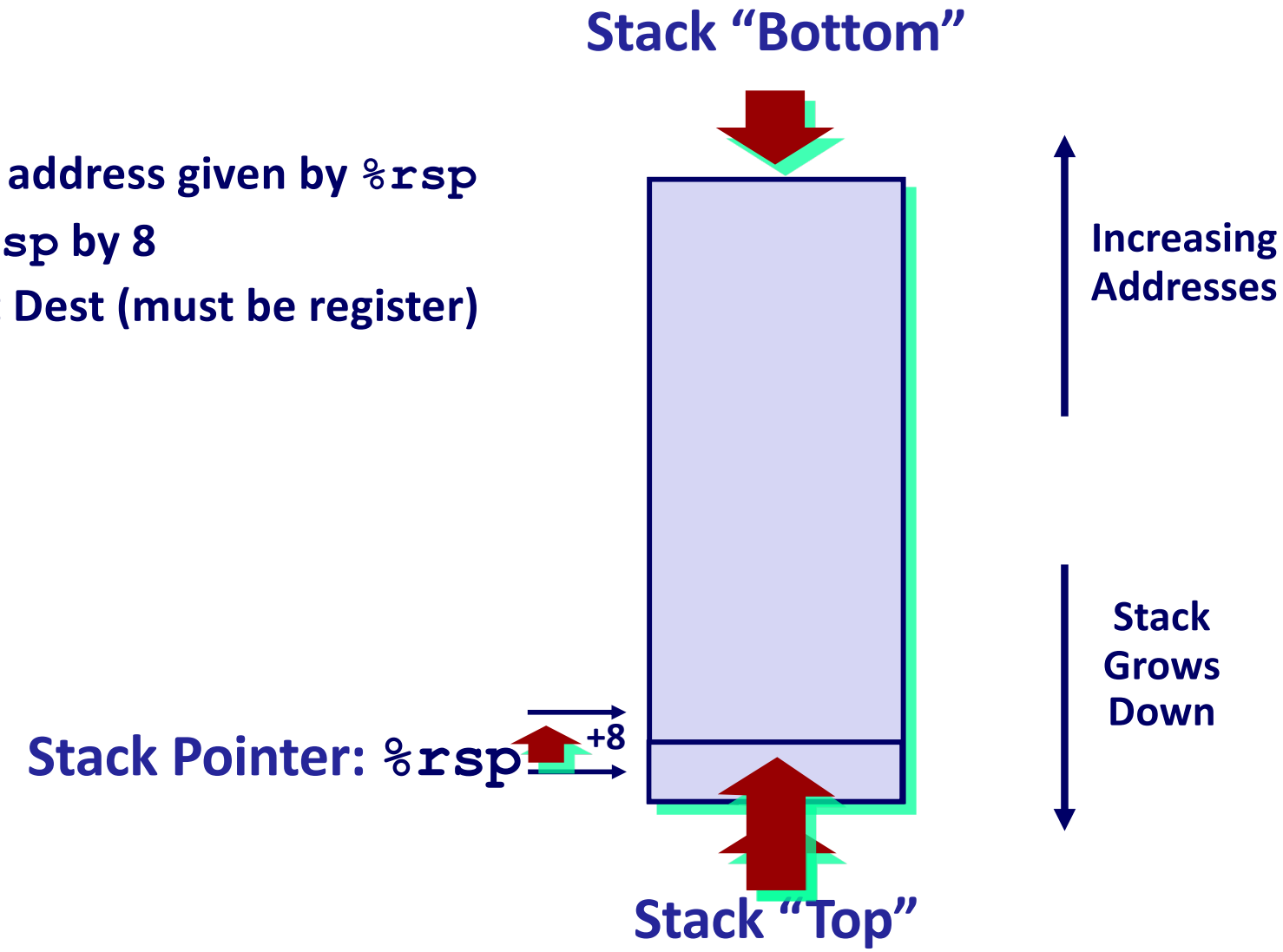
`pushq Src`

- Fetch operand at Src
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`



x86-64 Stack: Pop

- `popq` Dest
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (must be register)



Passing Control

Code Examples

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx                  # Save %rbx
  400541: mov    %rdx,%rbx             # Save dest
  400544: callq  400550 <mult2>        # mult2(x,y)
  400549: mov    %rax,(%rbx)           # Save at dest
  40054c: pop    %rbx                  # Restore %rbx
  40054d: retq                         # Return
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax            # a
  400553:  imul   %rsi,%rax            # a * b
  400557:  retq                        # Return
```

Procedure Control Flow

Use stack to support procedure call and return

Procedure call: **`call label`**

- Push return address on stack
- Jump to label

Return address:

- Address of the next instruction right after call
- Example from disassembly

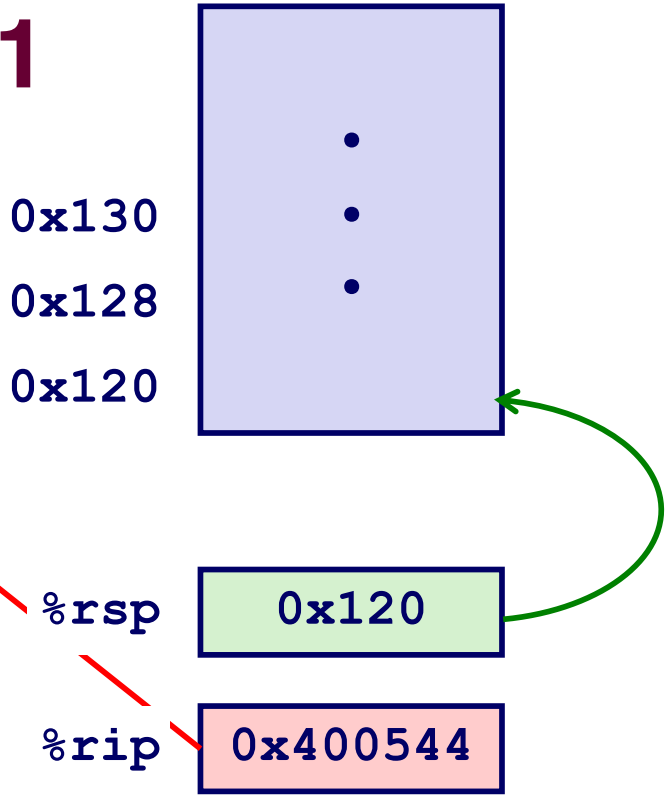
Procedure return: **`ret`**

- Pop address from stack
- Jump to address

Control Flow Example #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

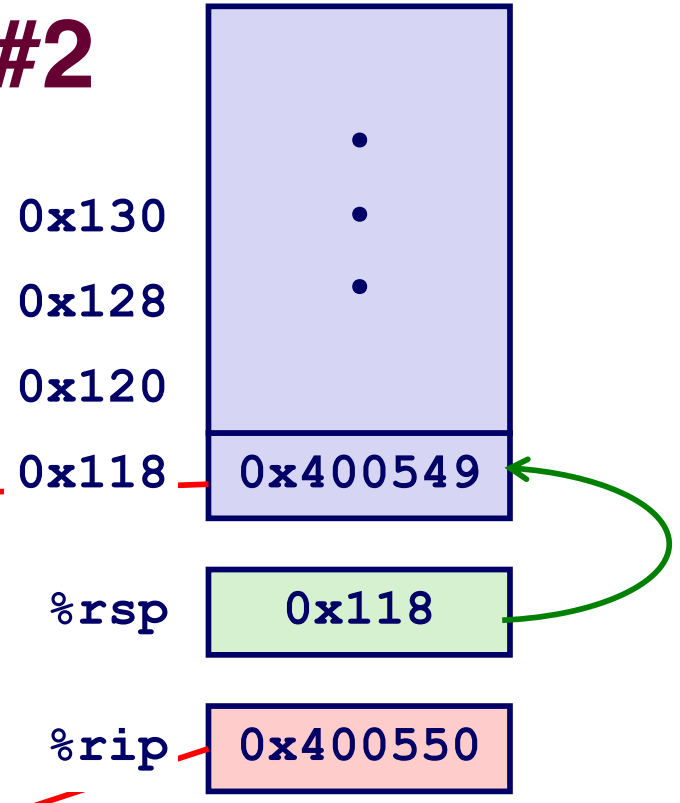
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

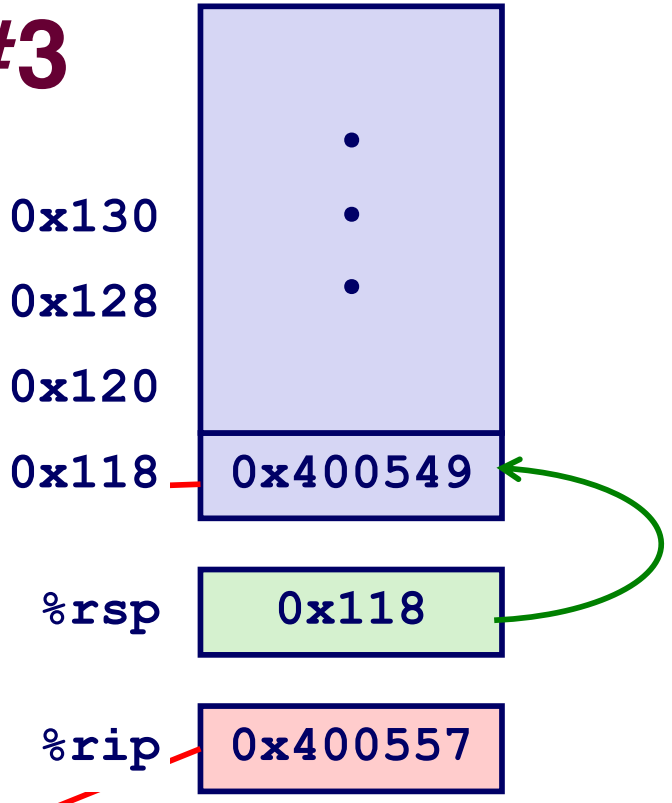
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

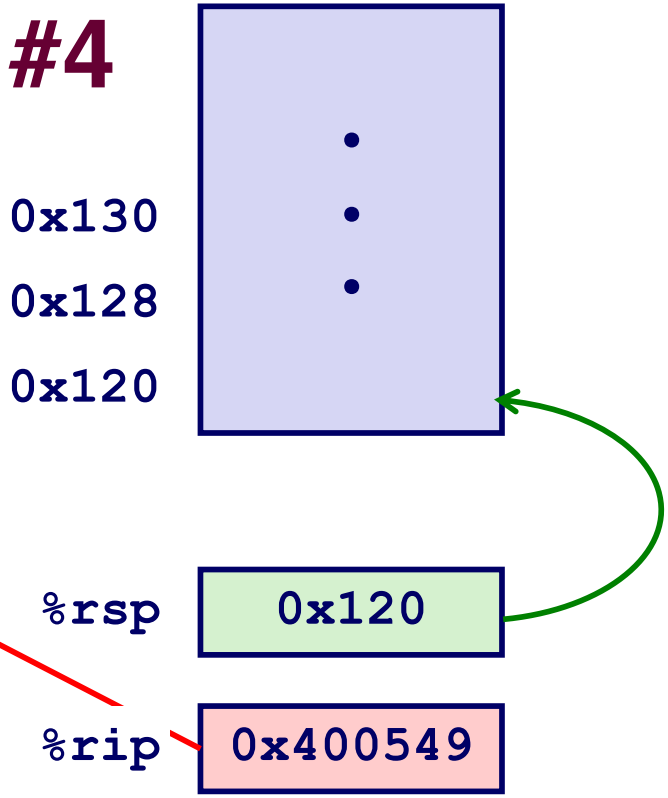
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #4

```
0000000000400540 <multstore>:
   •
   •
   400544: callq  400550 <mult2>
   400549: mov    %rax,(%rbx)
   •
   •
```

```
0000000000400550 <mult2>:
   400550:  mov    %rdi,%rax
   •
   •
   400557:  retq
```

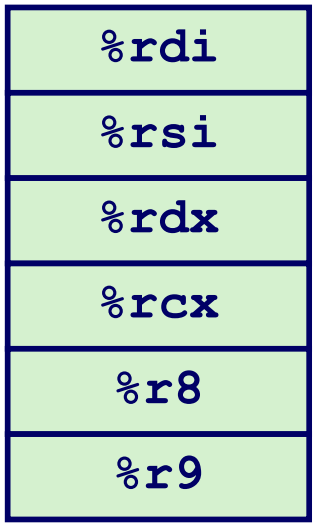


Passing Data

Procedure Data Flow

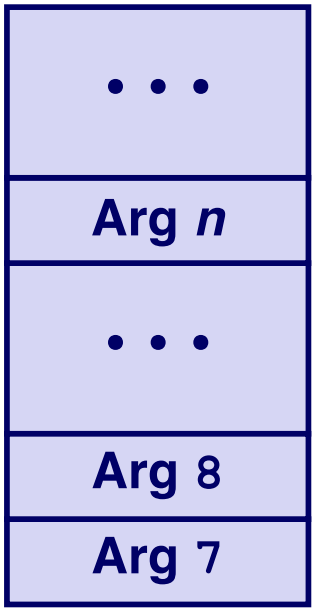
Registers

First 6 arguments



Return value

Stack



Only allocate stack space when needed

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  ...
  400541: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)        # Save at dest
  ...
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax         # a
  400553:  imul   %rsi,%rax         # a * b
  # s in %rax
  400557:  retq                     # Return
```

Managing Local Data

Stack-Based Languages

Languages that support recursion

- e.g., C, Pascal, Java
- Code must be "Reentrant"
  - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
  - Arguments
  - Local variables
  - Return pointer

Stack discipline

- State for given procedure needed for limited time
  - From when called to when return
- Callee returns before caller does

Stack allocated in Frames

- state for single procedure instantiation

Call Chain Example

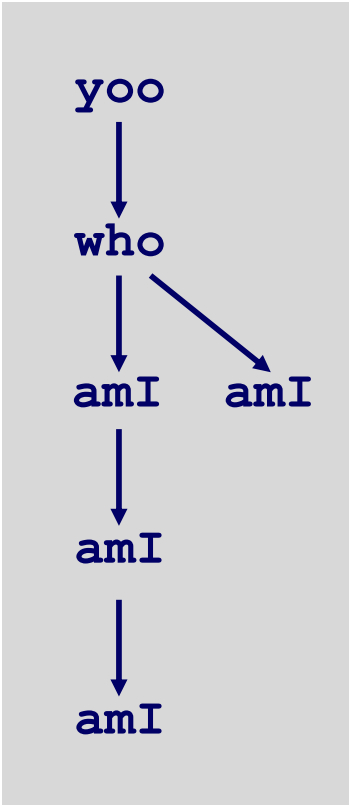
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

Procedure `amI()` is recursive

Example
Call Chain



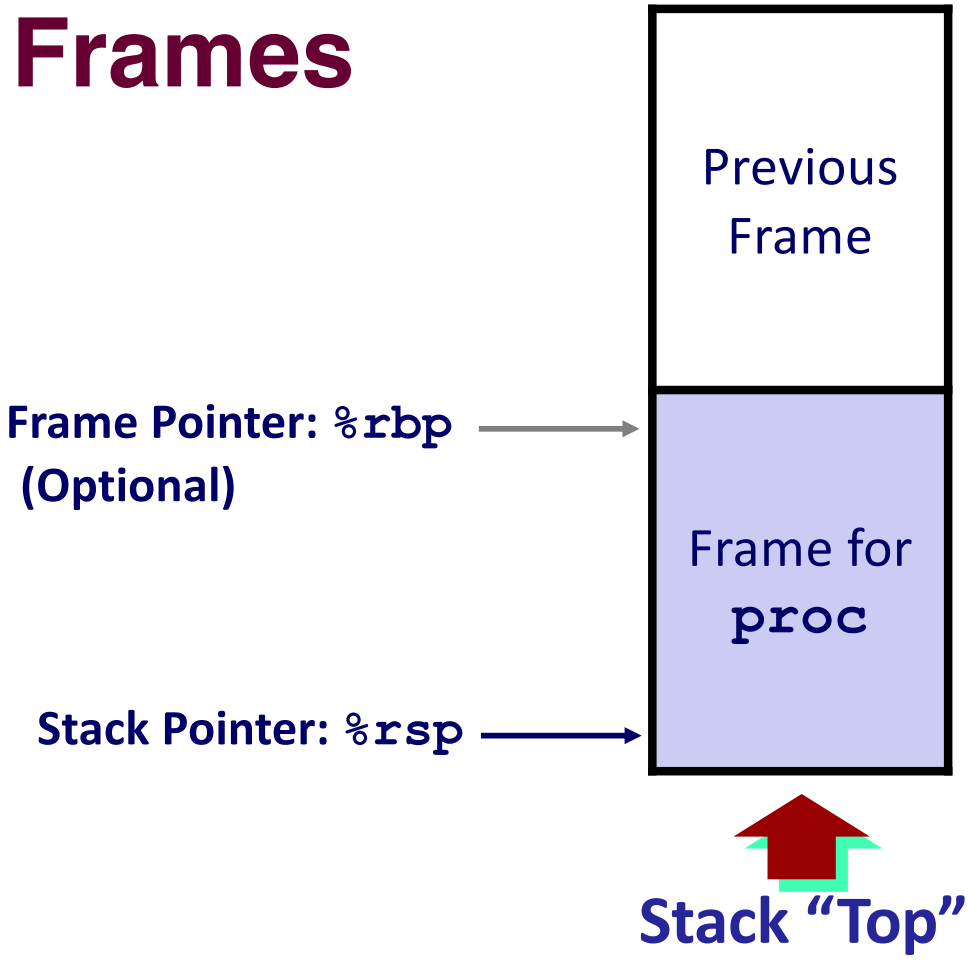
Stack Frames

Contents

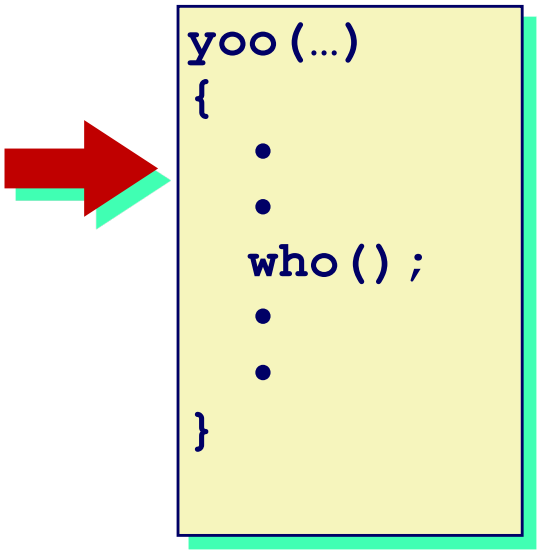
- Return information
- Local storage (if needed)
- Temporary space (if needed)

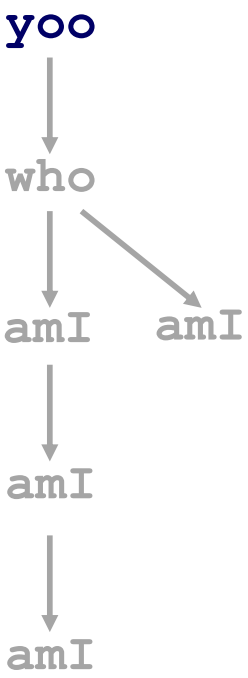
Management

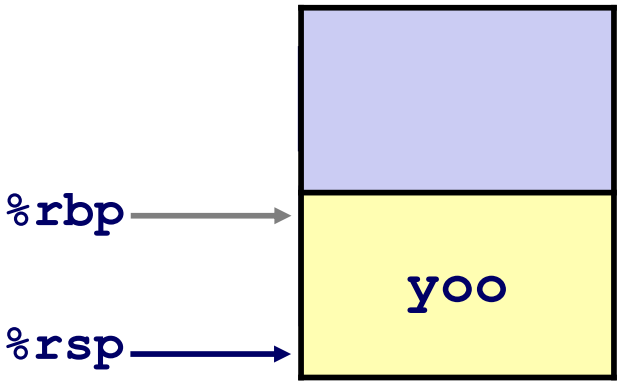
- Space allocated when enter procedure
  - "Set-up" code
  - Includes push by **`call`** instruction
- Deallocated when return
  - "Finish" code
  - Includes pop by **`ret`** instruction



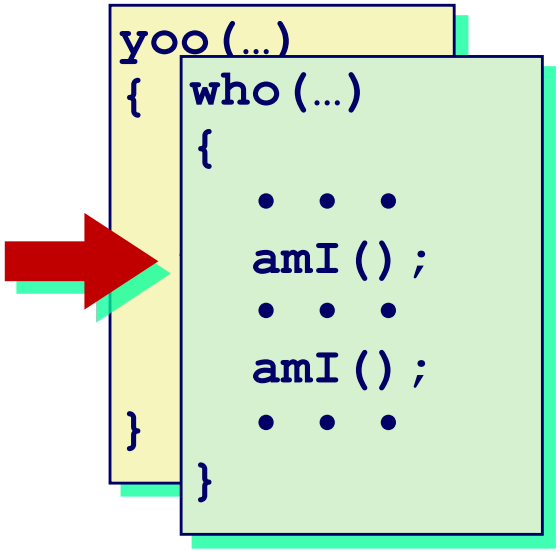
Example

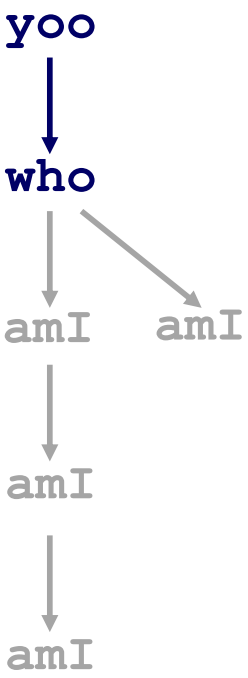


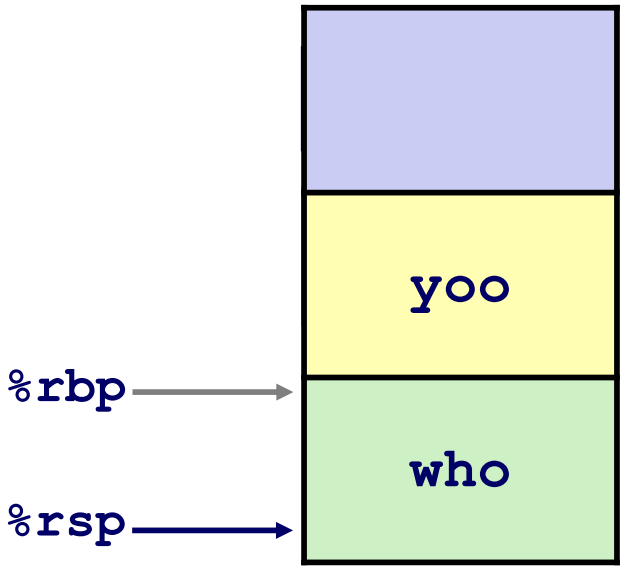




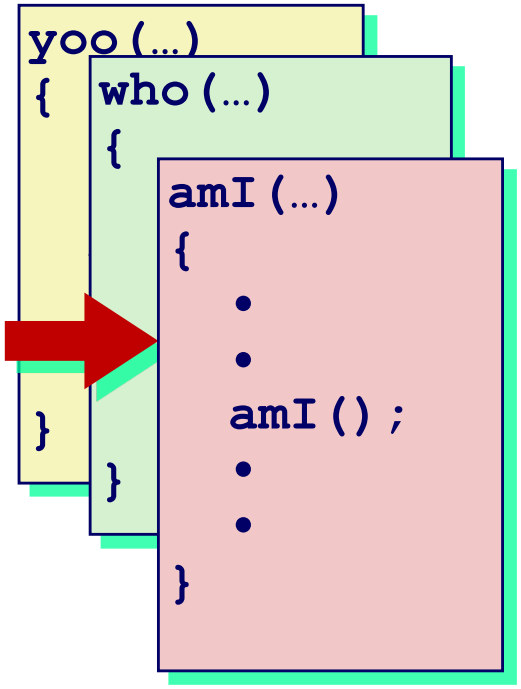
Example

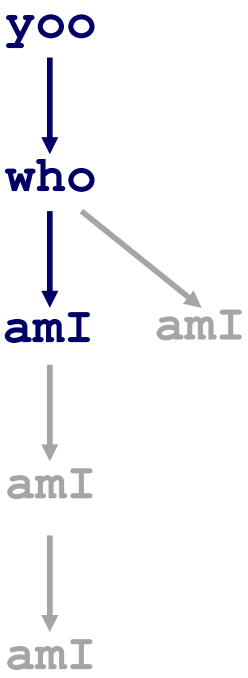


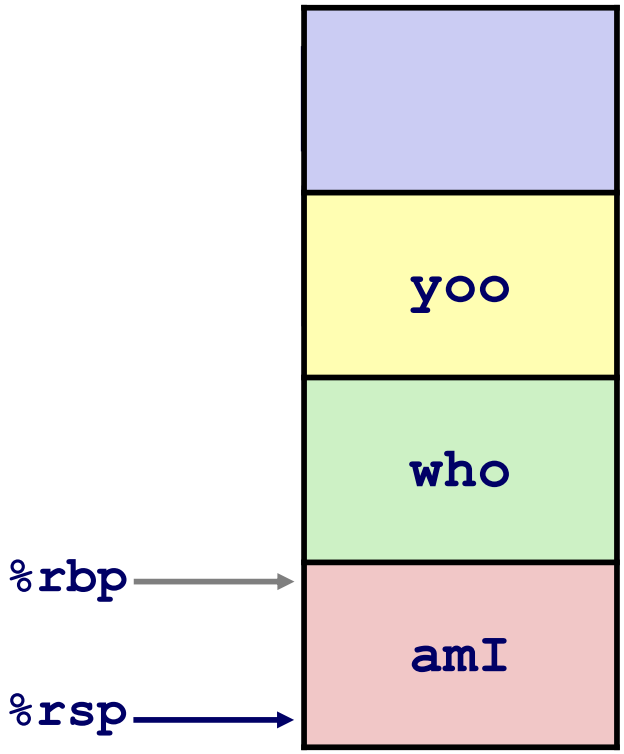




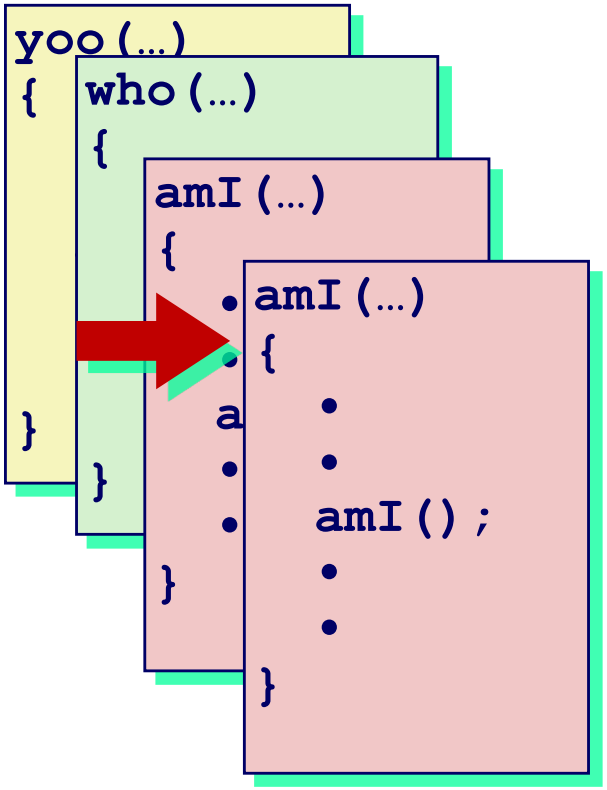
Example

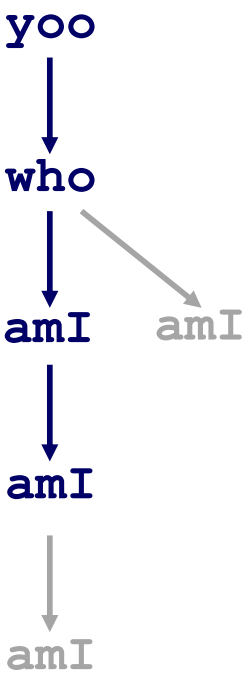

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

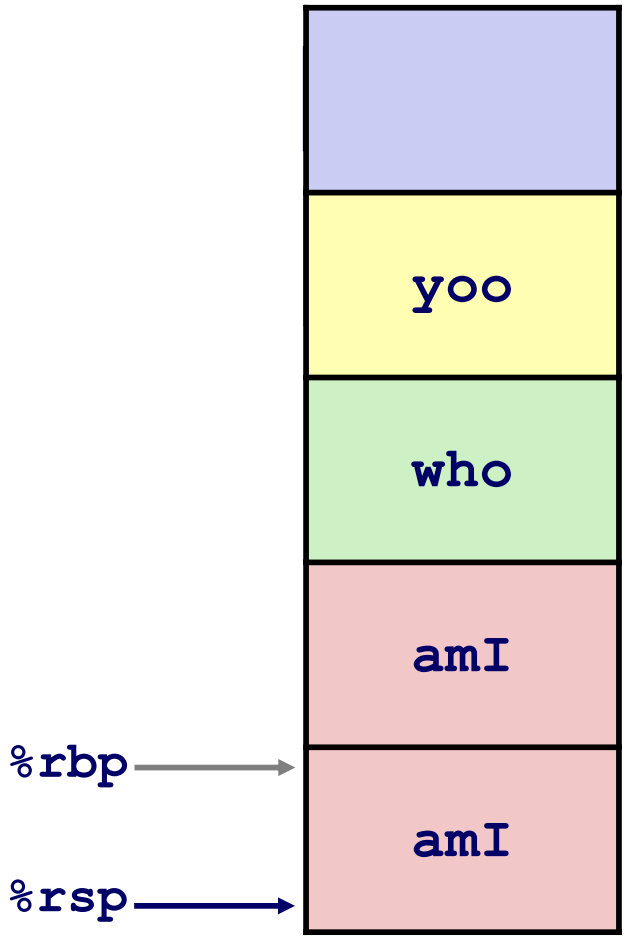

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

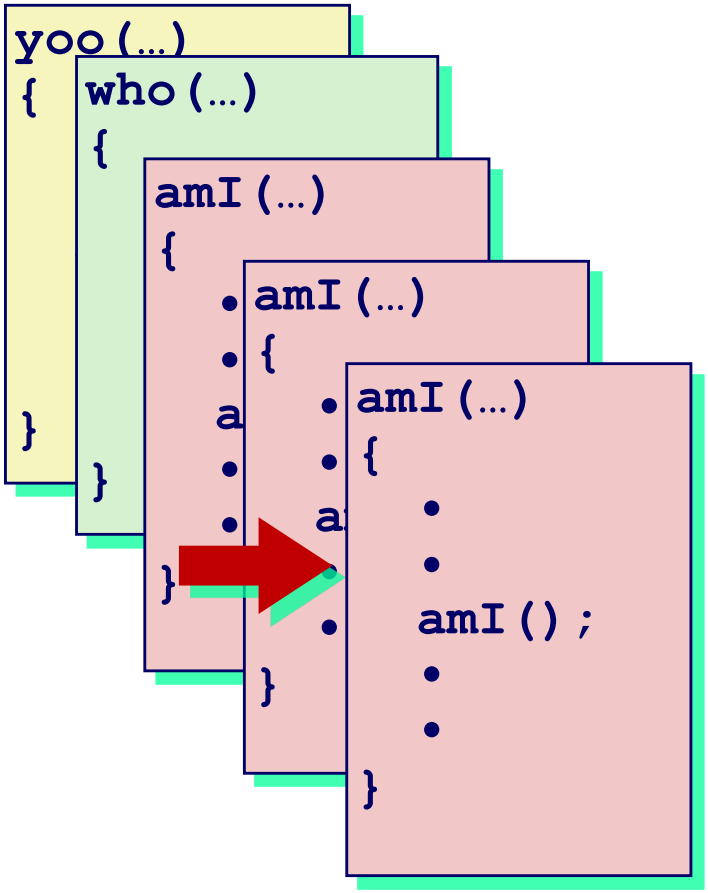
Example

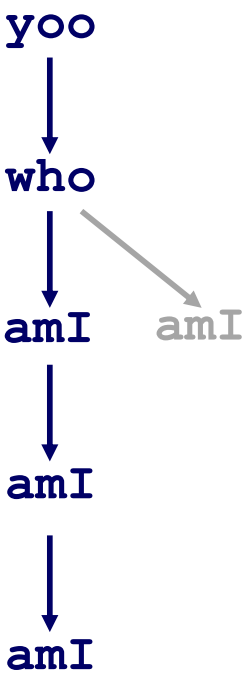

yoo (…)
{
who (…)
{
amI (…)
{
amI (…)
{
amI();
}
}
}
}

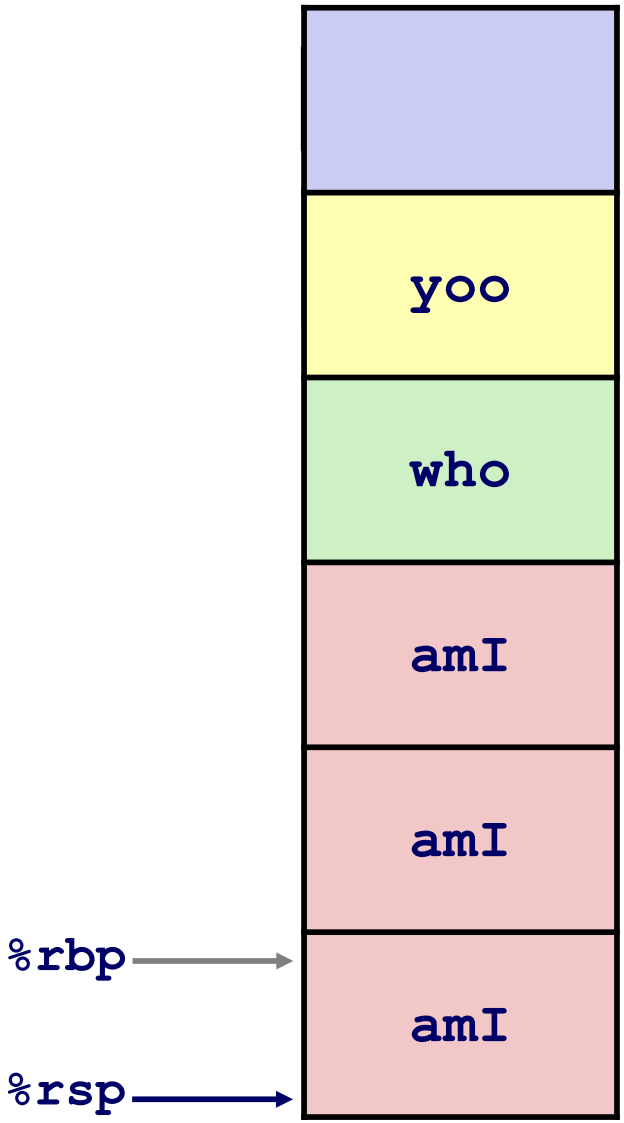

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

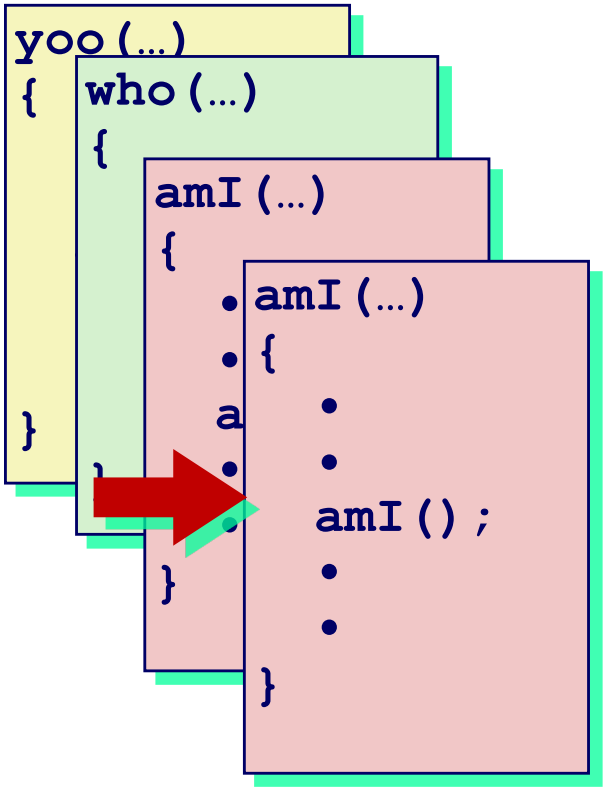
Example

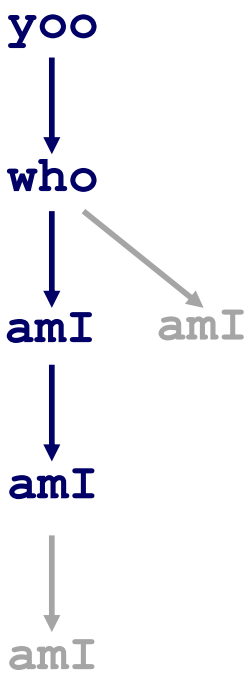

yoo (...)
{
who (...)
{
amI (...)
{
amI (...)
{
amI (...)
{
amI();
}
}
}

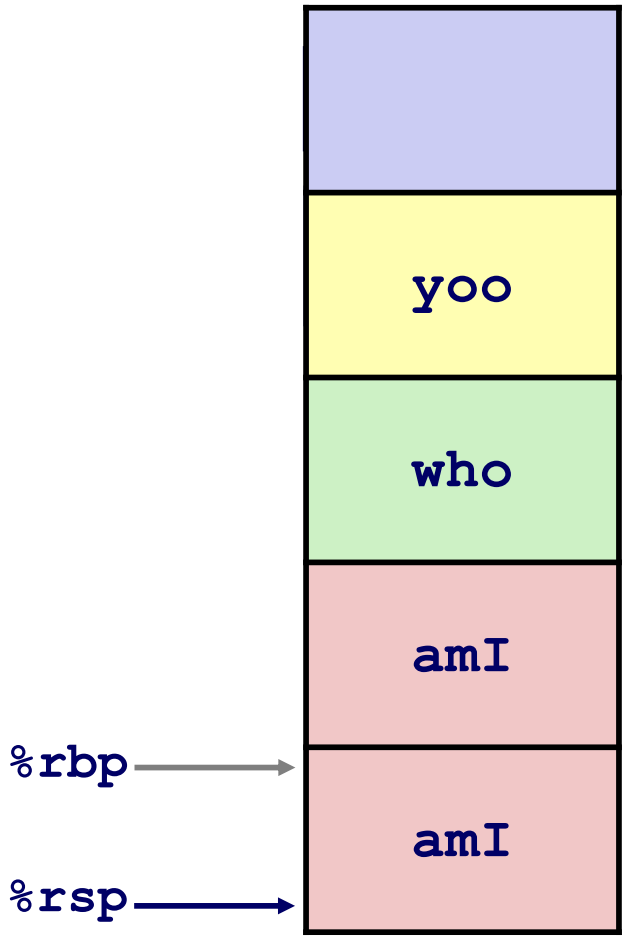

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

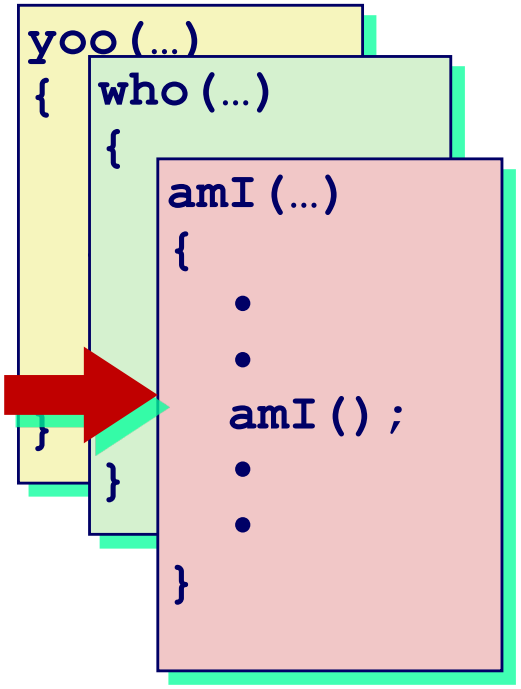
Example

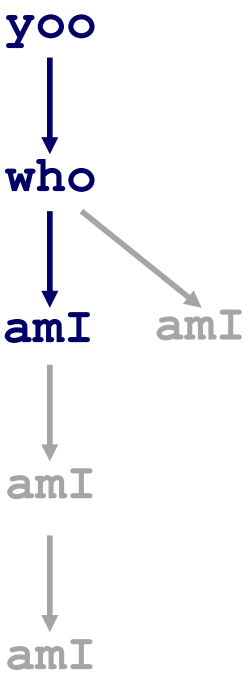

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}

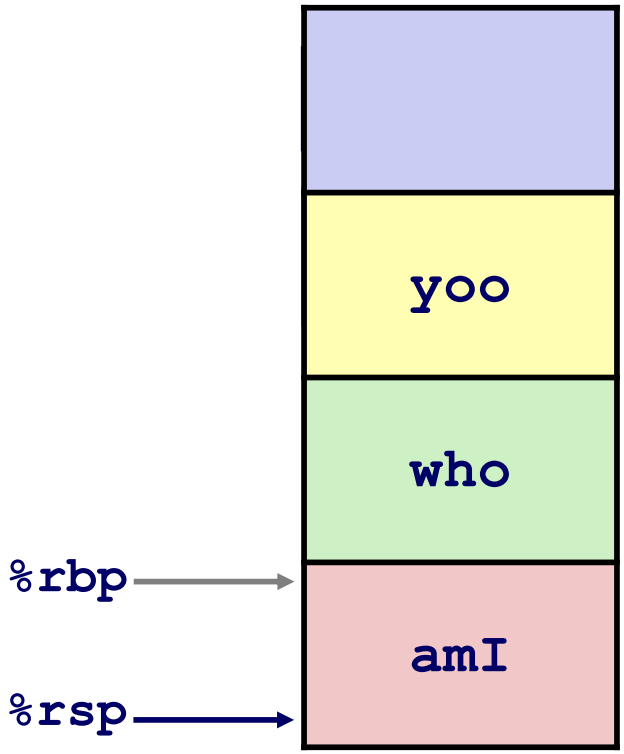

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

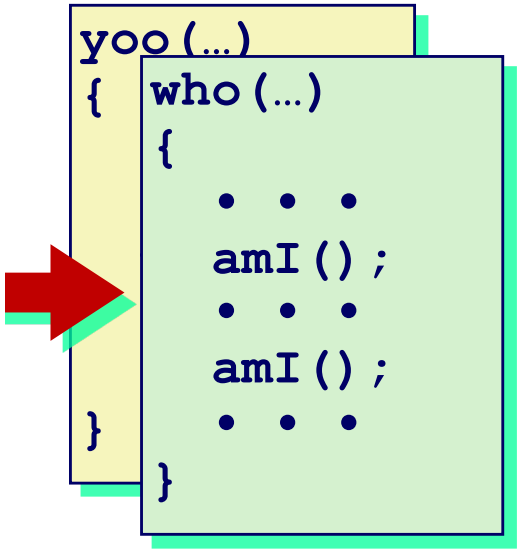
Example

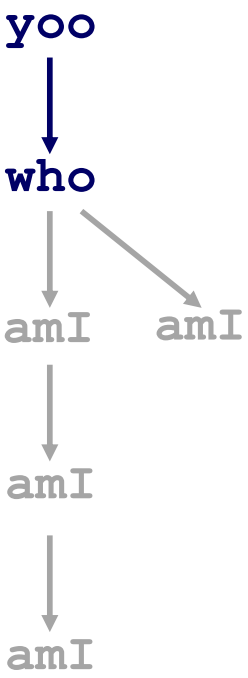

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

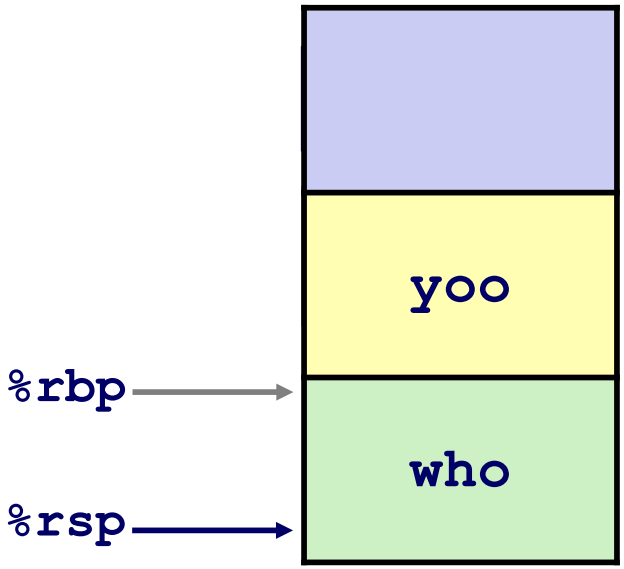

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

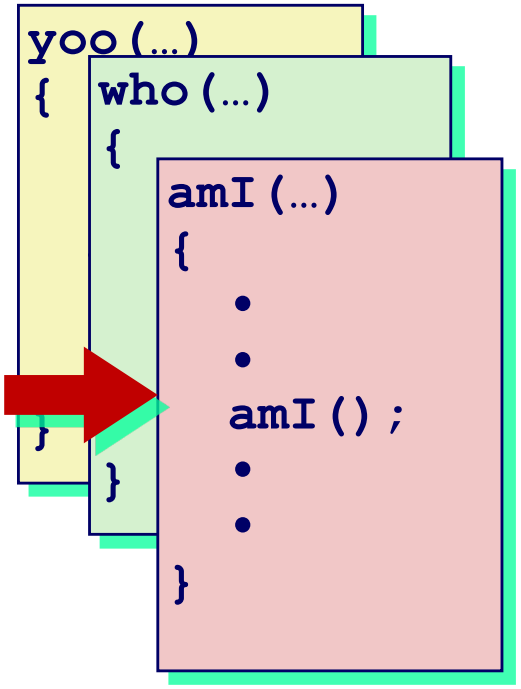
Example

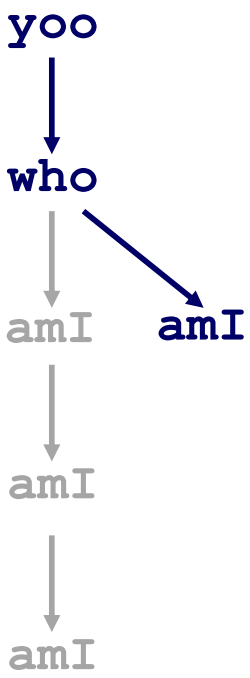


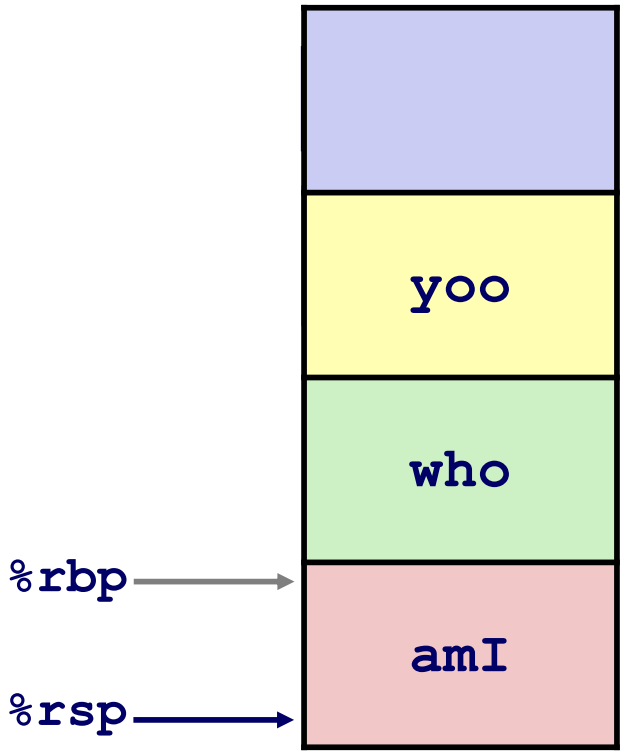




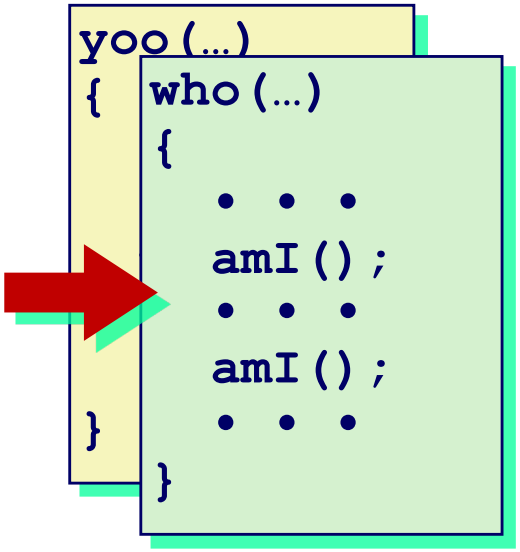
Example

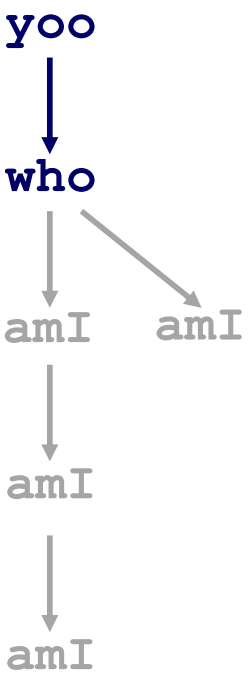

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

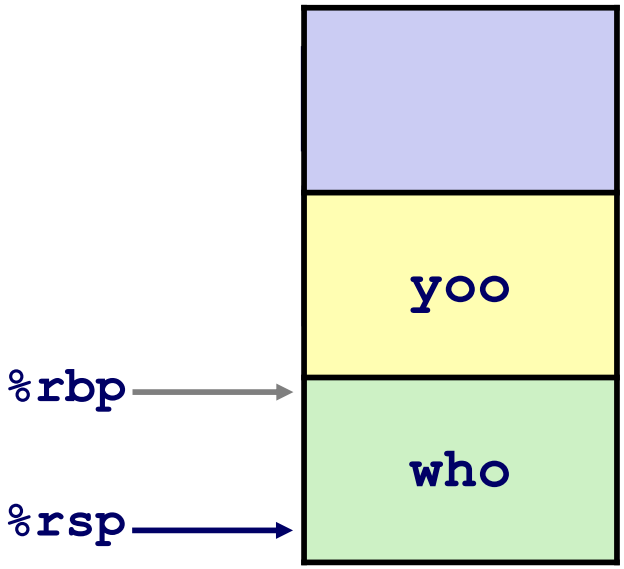

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

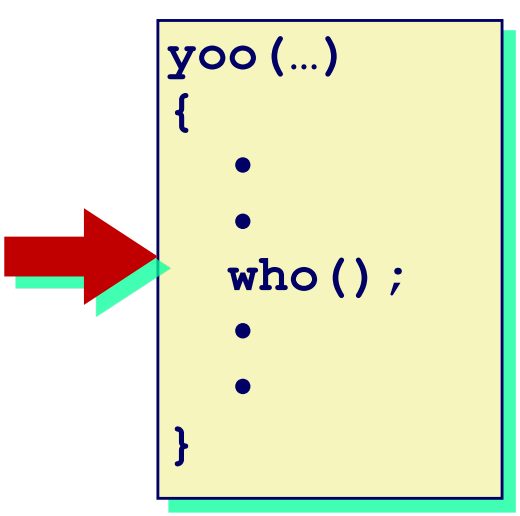
Example

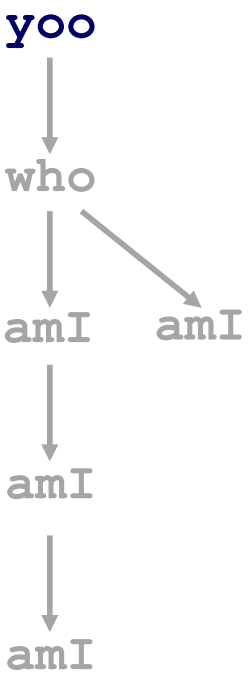


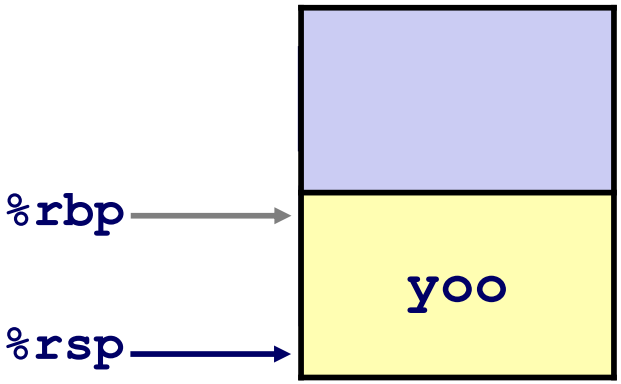




Example







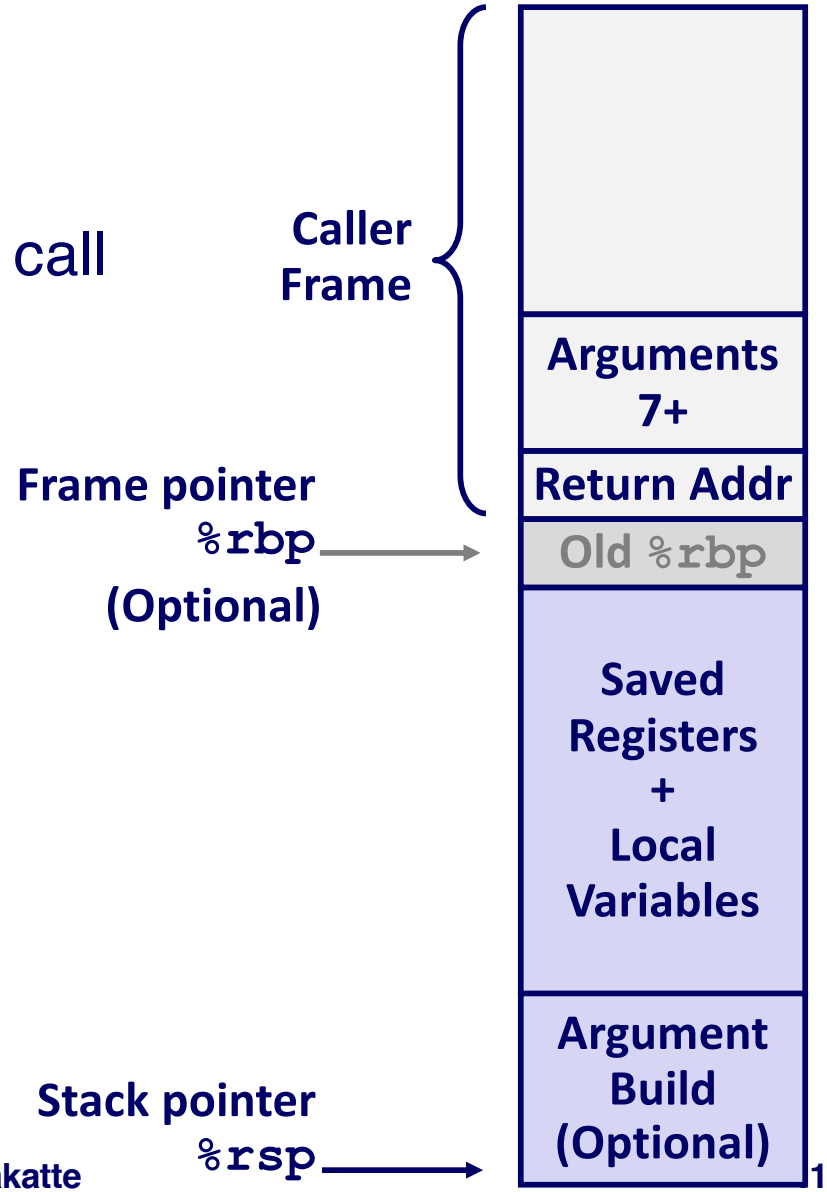
x86-64/Linux Stack Frame

Current Stack Frame ("Top" to Bottom)

- "Argument build:"
  Parameters for function about to call
- Local variables
  If can't keep in registers
- Saved register context
- Old frame pointer (optional)

Caller Stack Frame

- Return address
  - Pushed by `call` instruction
- Arguments for this call



Example: `incr`

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
incr:
  movq     (%rdi), %rax
  addq     %rax, %rsi
  movq     %rsi, (%rdi)
  ret
```

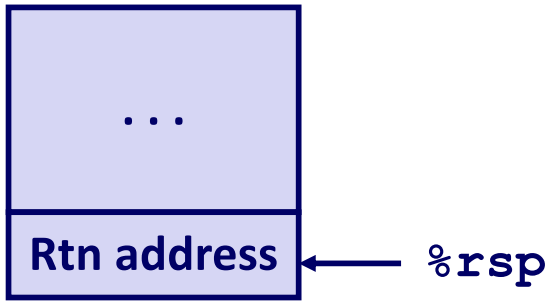
Register	Use(s)
%rdi	Argument **p**
%rsi	Argument **val**, **y**
%rax	**x**, Return value

Example: Calling `incr` #1

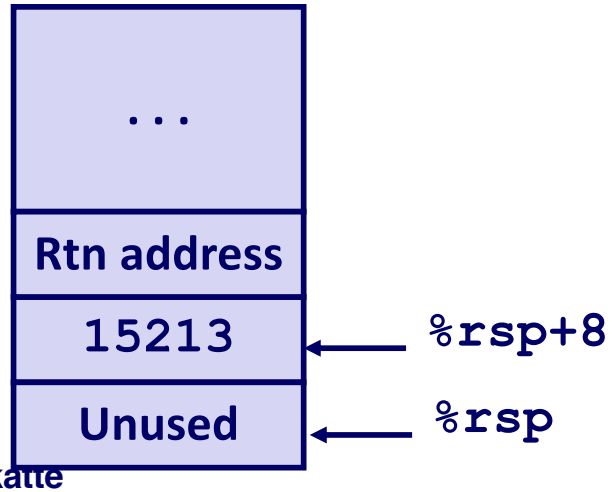
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



Resulting Stack Structure

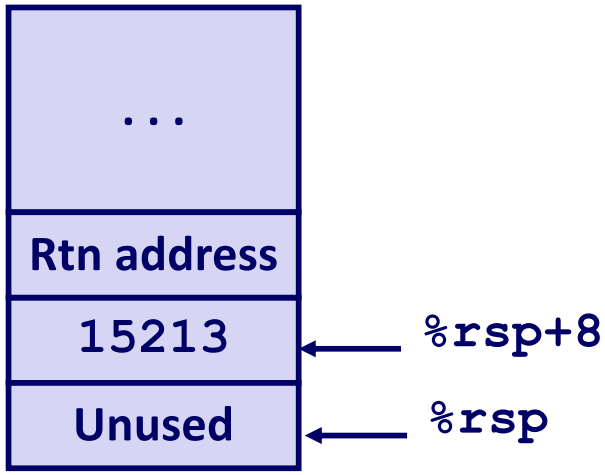


Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
   subq     $16, %rsp
   movq     $15213, 8(%rsp)
   movl     $3000, %esi
   leaq     8(%rsp), %rdi
   call     incr
   addq     8(%rsp), %rax
   addq     $16, %rsp
   ret
```

Stack Structure



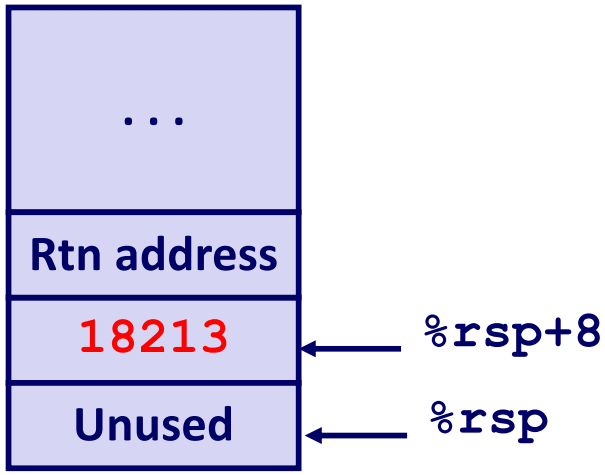
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure

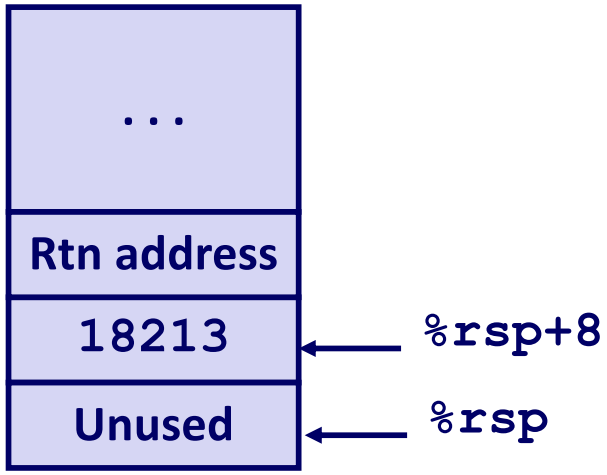


Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

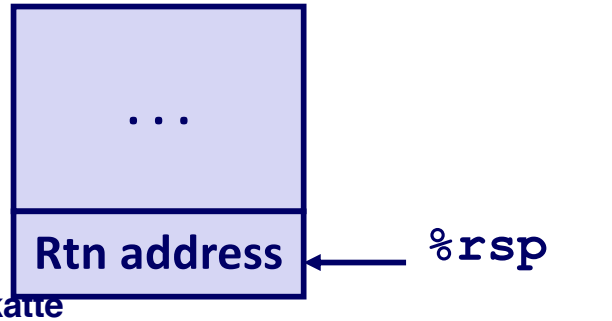
Stack Structure




```
call_incr:
   subq     $16, %rsp
   movq     $15213, 8(%rsp)
   movl     $3000, %esi
   leaq     8(%rsp), %rdi
   call     incr
   addq     8(%rsp), %rax
   addq     $16, %rsp
   ret
```

Register	Use(s)
%rax	Return value

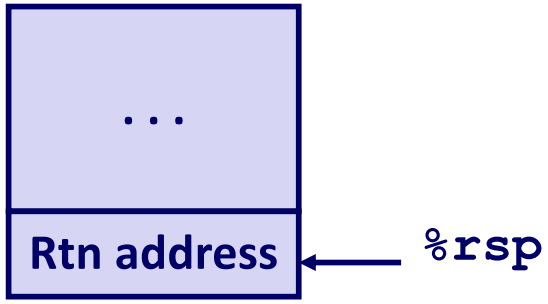
Updated Stack Structure



Example: Calling `incr` #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

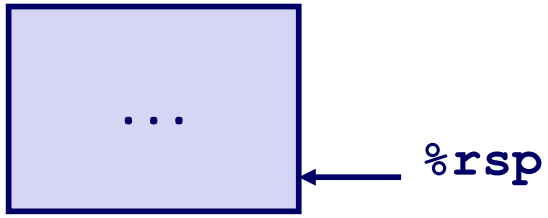
Updated Stack Structure




```
call_incr:
   subq     $16, %rsp
   movq     $15213, 8(%rsp)
   movl     $3000, %esi
   leaq     8(%rsp), %rdi
   call     incr
   addq     8(%rsp), %rax
   addq     $16, %rsp
   ret
```

Register	Use(s)
%rax	Return value

Final Stack Structure



Register Saving Conventions

When procedure **yoo** calls **who**:

- **yoo** is the caller
- **who** is the callee

Can register be used for temporary storage?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

- Contents of register **%rdx** overwritten by **who**
- This could be trouble → something should be done!
  - Need some coordination

Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the caller
- `who` is the callee

Can register be used for temporary storage?

Conventions

- "Caller Saved"
  - Caller saves temporary values in its frame before the call
- "Callee Saved"
  - Callee saves temporary values in its frame before using
  - Callee restores them before returning to caller

x86-64 Linux Register Usage #1

`%rax`

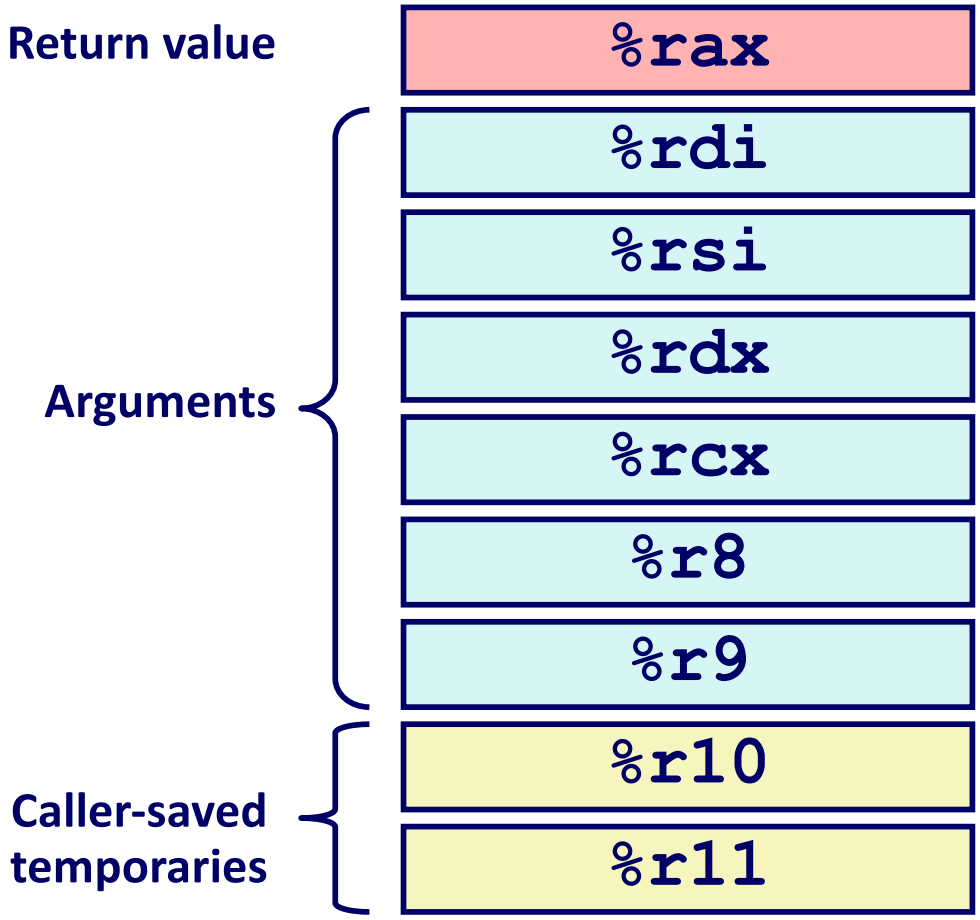
- Return value
- Also caller-saved
- Can be modified by procedure

`%rdi`, ..., `%r9`

- Arguments
- Also caller-saved
- Can be modified by procedure

`%r10`, `%r11`

- Caller-saved
- Can be modified by procedure



x86-64 Linux Register Usage #2

`%rbx`, `%r12`, `%r13`, `%r14`

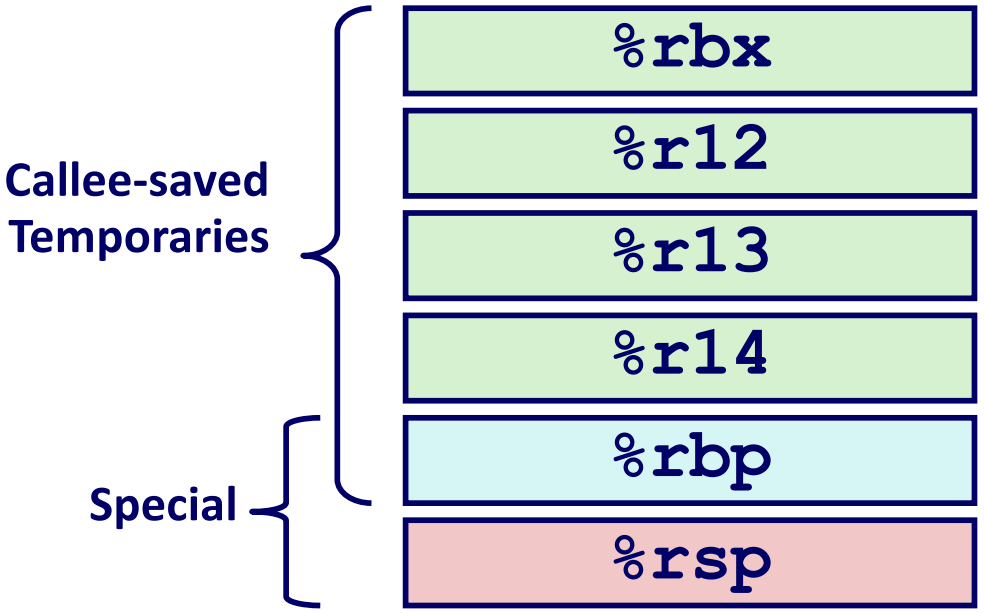
- Callee-saved
- Callee must save & restore

`%rbp`

- Callee-saved
- Callee must save & restore
- May be used as frame pointer
- Can mix & match

`%rsp`

- Special form of callee save
- Restored to original value upon exit from procedure

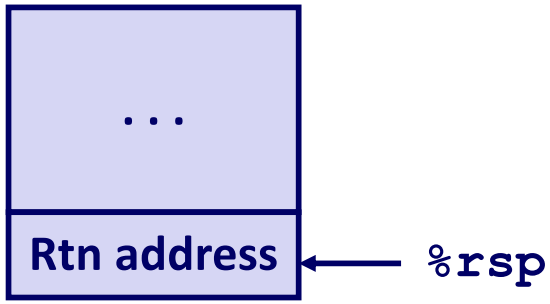


Callee-Saved Example #1

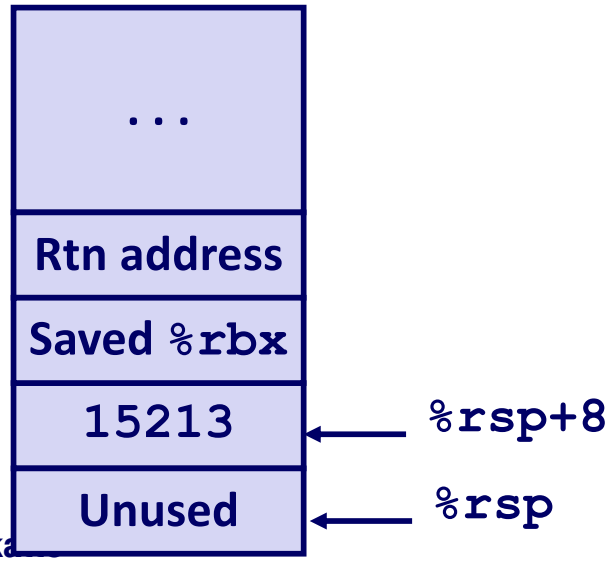
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

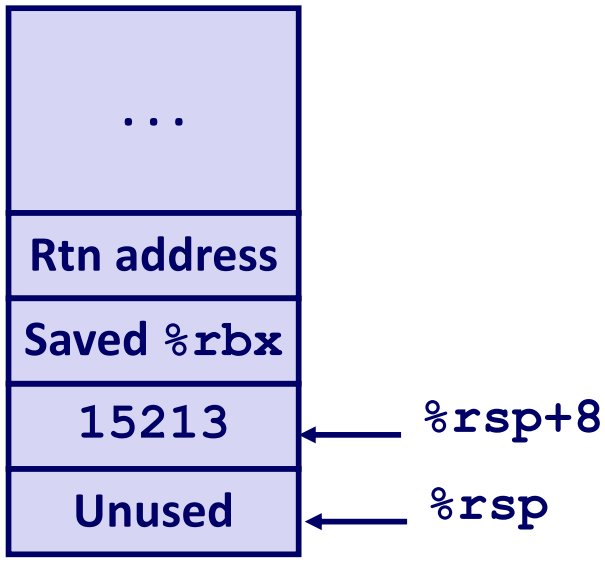


Callee-Saved Example #2

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure



Pre-return Stack Structure

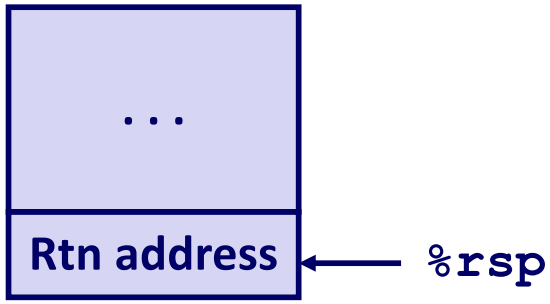


Illustration of Recursion

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

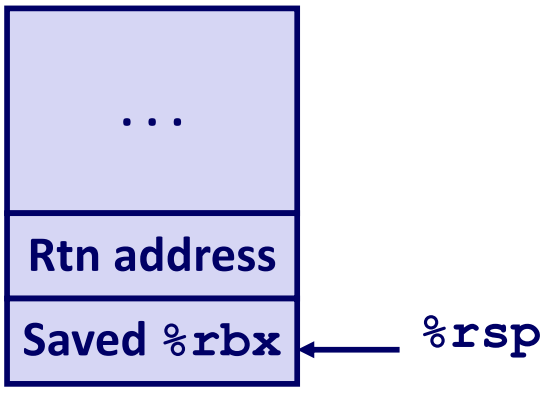
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

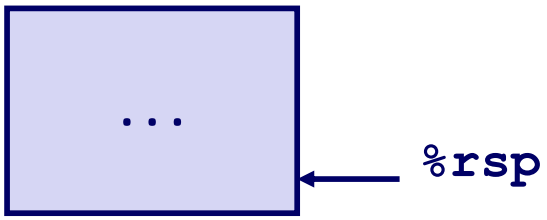
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rax	Return value	Return value



Observations About Recursion

Handled Without Special Consideration

- Stack frames mean that each function call has private storage
  - Saved registers & local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another's data
  - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
- Stack discipline follows call / return pattern
  - If P calls Q, then Q returns before P
  - Last-In, First-Out

Also works for mutual recursion

- P calls Q; Q calls P

x86-64 Procedure Summary

Important Points

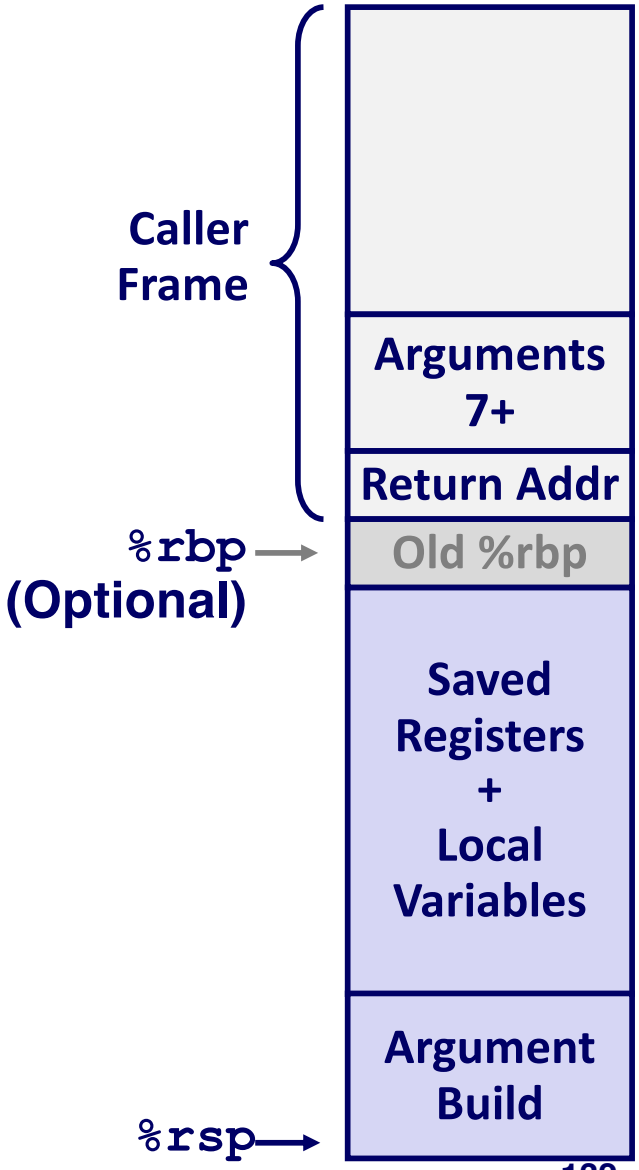
- Stack is the right data structure for procedure call / return
  - If P calls Q, then Q returns before P

Recursion (& mutual recursion) handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result return in `%rax`

Pointers are addresses of values

- On stack or global



Basic Data Types

Integral

- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

Intel	GAS	Bytes	C
byte	`b`	1	[`unsigned`] `char`
word	`w`	2	[`unsigned`] `short`
double word	`l`	4	[`unsigned`] `int`

Floating Point

- Stored & operated on in floating point registers

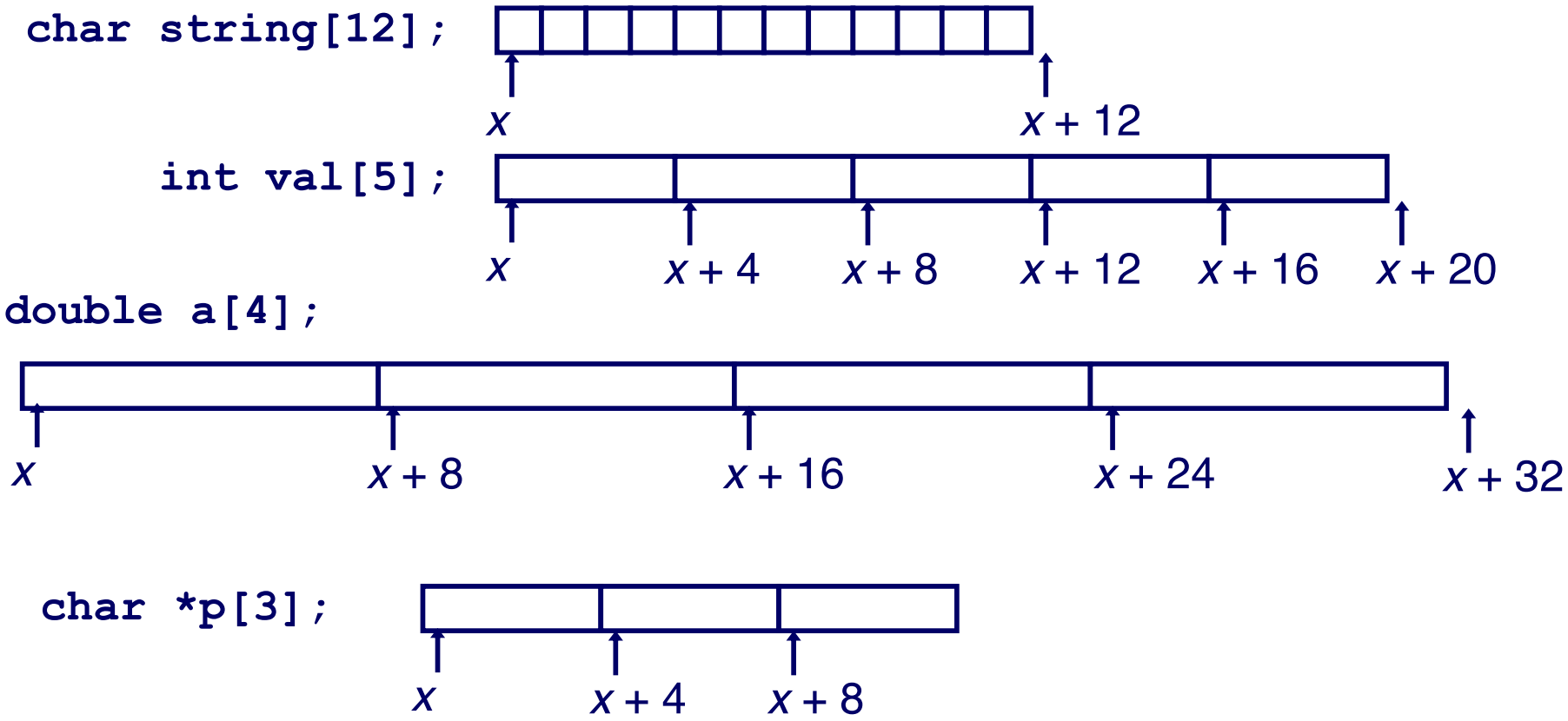
Intel	GAS	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`
Extended	`t`	10/12	`long double`

Array Allocation

Basic Principle

T `A[`*L*`];`

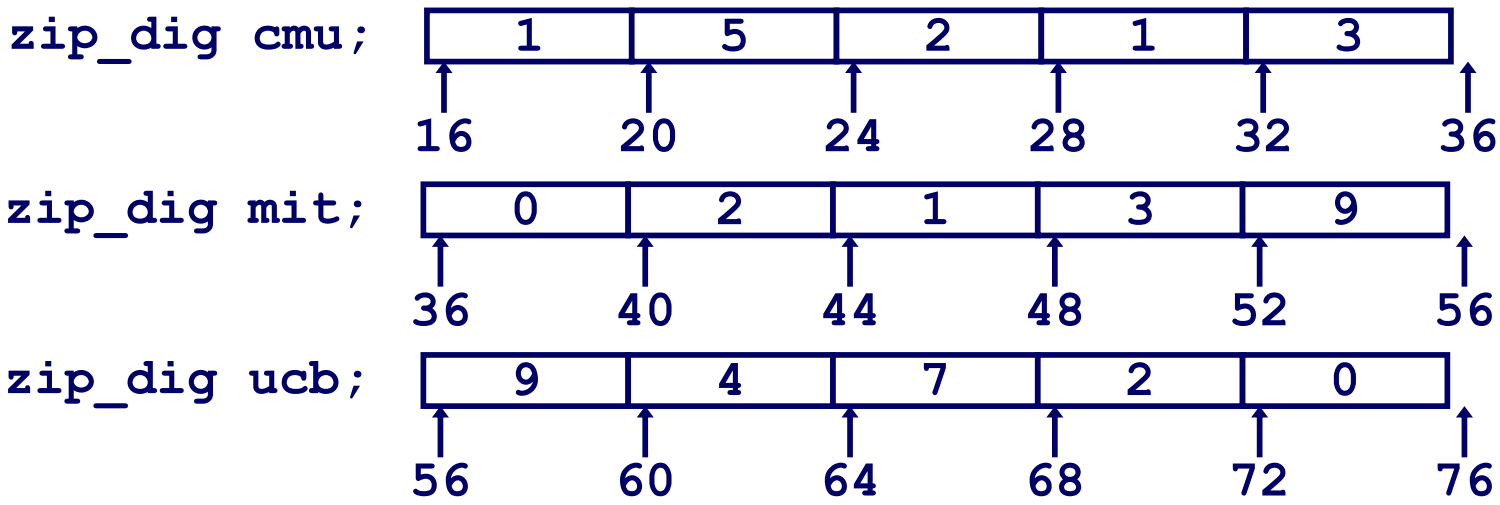
- Array of data type *T* and length *L*
- Contiguously allocated region of *L* * `sizeof(`*T*`)` bytes



Array Example

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



Notes

- Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

Computation

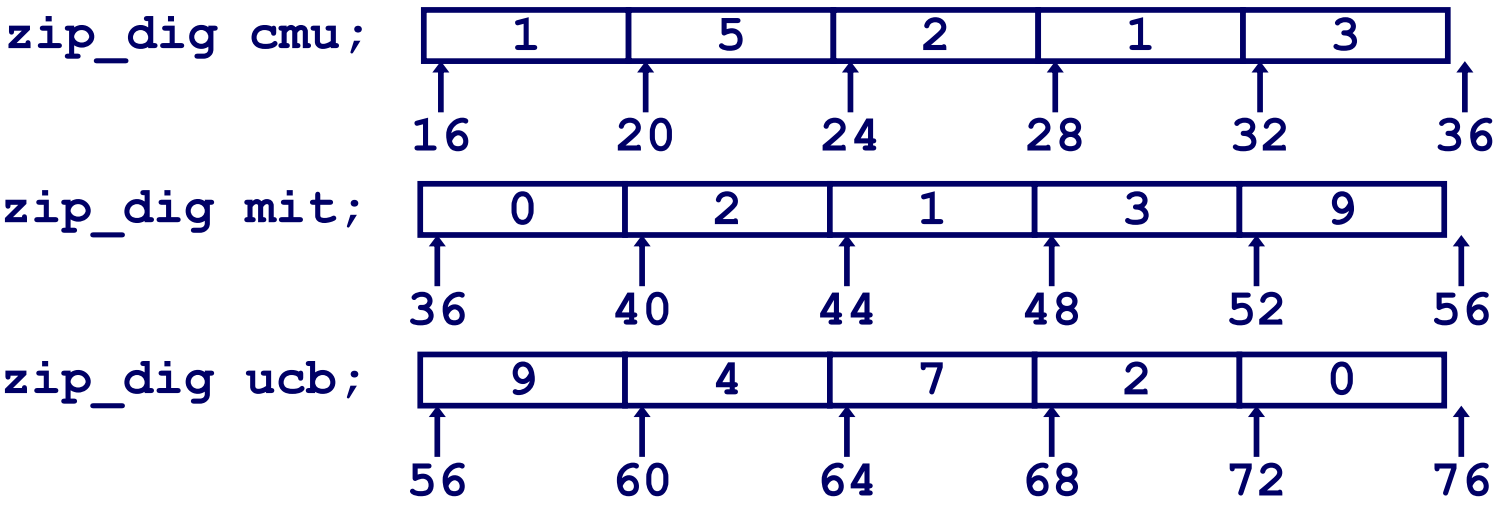
- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

Memory Reference Code

```
# %edx = z
# %eax = dig
 movl (%edx,%eax,4),%eax # z[dig]
```

Referencing Examples



Code Does Not Do Any Bounds Checking!

Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	`3`	**Yes**
`mit[5]`	`36 + 4* 5 = 56`	`9`	**No**
`mit[-1]`	`36 + 4*-1 = 32`	`3`	**No**
`cmu[15]`	`16 + 4*15 = 76`	`??`	**No**

- Out of range behavior implementation-dependent
  - No guaranteed relative allocation of different arrays

Array Loop Example

Original Source

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

Transformed Version

- As generated by GCC
- Eliminate loop variable `i`
- Convert array code to pointer code
- Express in do-while form
  - No need to test at entrance

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

Array Loop Implementation

Registers

```
%ecx   z
%eax   zi
%ebx   zend
```

Computations

- `10*zi + *z` implemented as `*z + 2*(zi+4*zi)`
- `z++` increments by 4

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

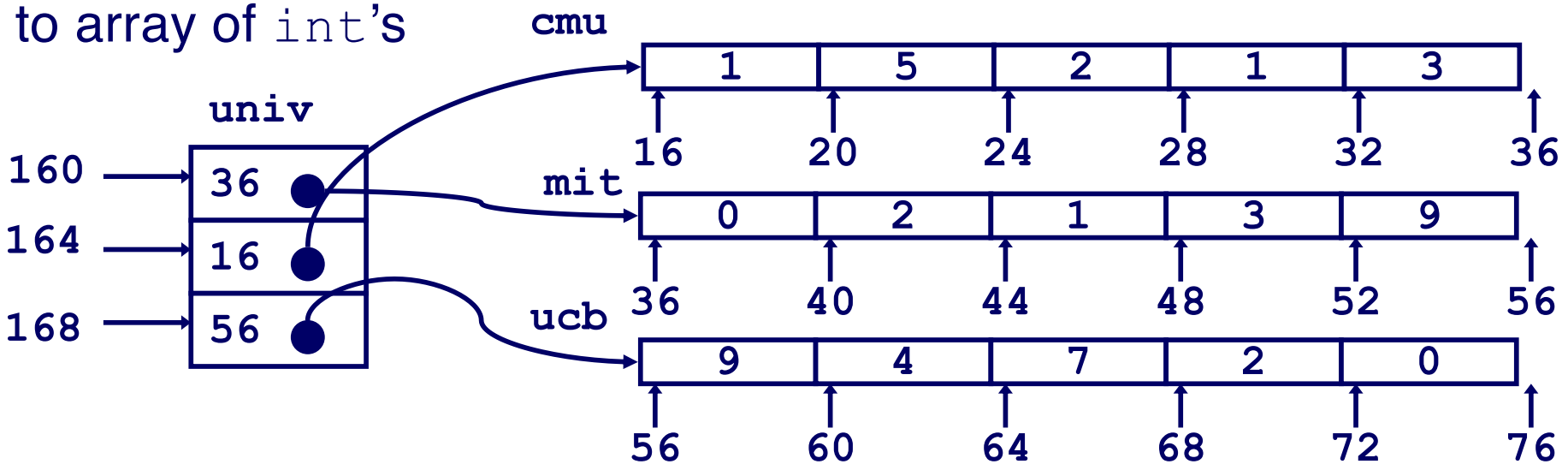
```
  # %ecx = z
  xorl %eax,%eax          # zi = 0
  leal 16(%ecx),%ebx      # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx # 5*zi
  movl (%ecx),%eax        # *z
  addl $4,%ecx            # z++
  leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx          # z : zend
  jle .L59                # if <= goto loop
```

Multi-Level Array Example

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of `int`'s

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```



Element Access in Multi-Level Array

```
int get_univ_digit
   (int index, int dig)
{
   return univ[index][dig];
}
```

Computation

- Element access

  `Mem[Mem[univ+4*index]+4*dig]`

- Must do two memory reads
  - First get pointer to row array

```
# %ecx = index
# %eax = dig
leal 0(,%ecx,4),%edx     # 4*index
movl univ(%edx),%edx     # Mem[univ+4*index]
movl (%edx,%eax,4),%eax  # Mem[...+4*dig]
```

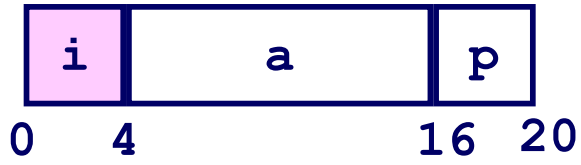
Structures

Concept

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout



Accessing Structure Member

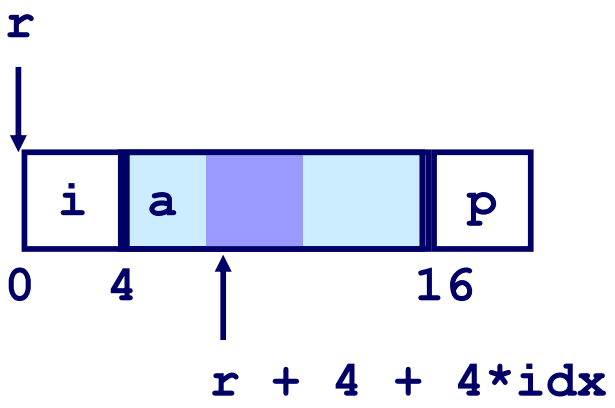
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Generating Pointer to Struct. Member

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



Generating Pointer to Array Element

- Offset of each structure member determined at compile time

```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

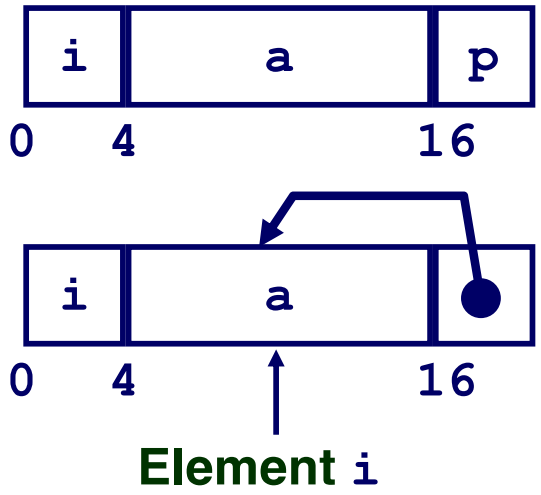
```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax   # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

Structure Referencing (Cont.)

C Code

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```




```
 # %edx = r
movl  (%edx),%ecx          # r->i
leal  0(,%ecx,4),%eax      # 4*(r->i)
leal  4(%edx,%eax),%eax    # r+4+4*(r->i)
movl  %eax,16(%edx)        # Update r->p
```

Alignment

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on IA32
  - treated differently by Linux and Windows!

Motivation for Aligning Data

- Memory accessed by (aligned) double or quad-words
  - Inefficient to load or store datum that spans quad word boundaries

Compiler

- Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment

Size of Primitive Data Type:

- 1 byte (e.g., `char`)
  - no restrictions on address
- 2 bytes (e.g., `short`)
  - lowest 1 bit of address must be 0_2
- 4 bytes (e.g., `int`, `float`, `char *`, etc.)
  - lowest 2 bits of address must be 00_2
- 8 bytes (e.g., `double`)
  - Windows (and most other OS's & instruction sets):
    - » lowest 3 bits of address must be 000_2
  - Linux:
    - » lowest 2 bits of address must be 00_2
    - » i.e., treated the same as a 4-byte primitive data type
- 12 bytes (`long double`)
  - Linux:
    - » lowest 2 bits of address must be 00_2
    - » i.e., treated the same as a 4-byte primitive data type

Satisfying Alignment with Structures

Offsets Within Structure

- Must satisfy element's alignment requirement

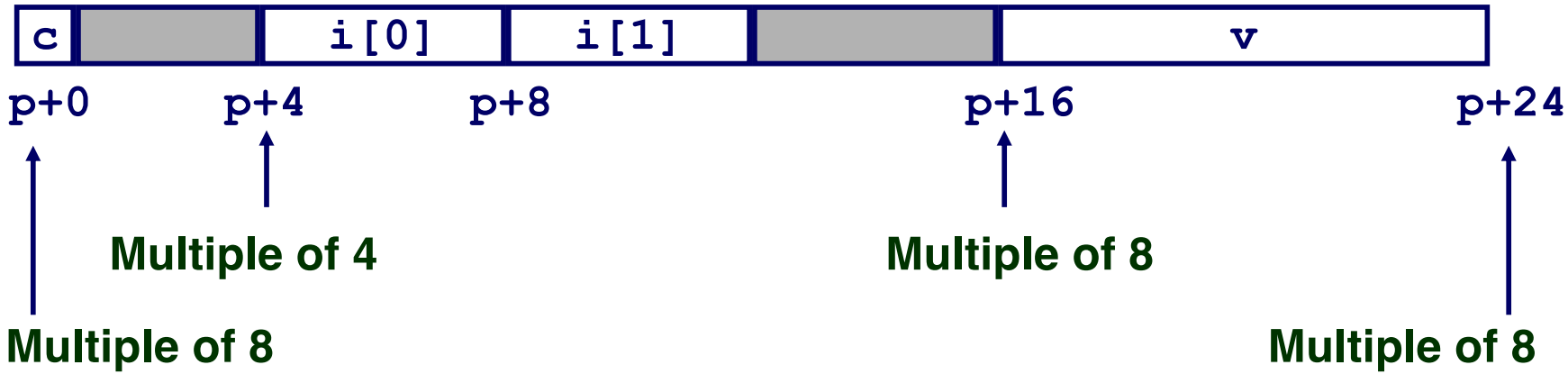
Overall Structure Placement

- Each structure has alignment requirement K
  - Largest alignment of any element
- Initial address & structure length must be multiples of K

Example (under Windows):

- K = 8, due to `double` element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

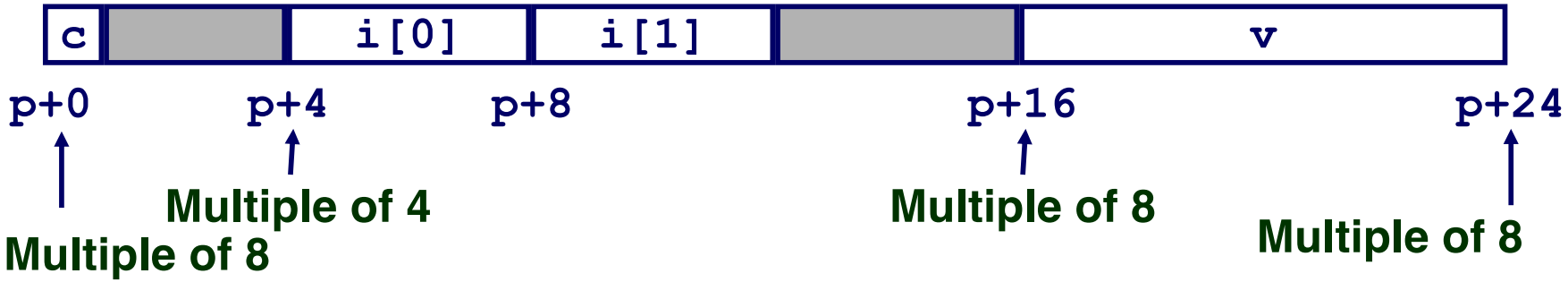


Linux vs. Windows

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

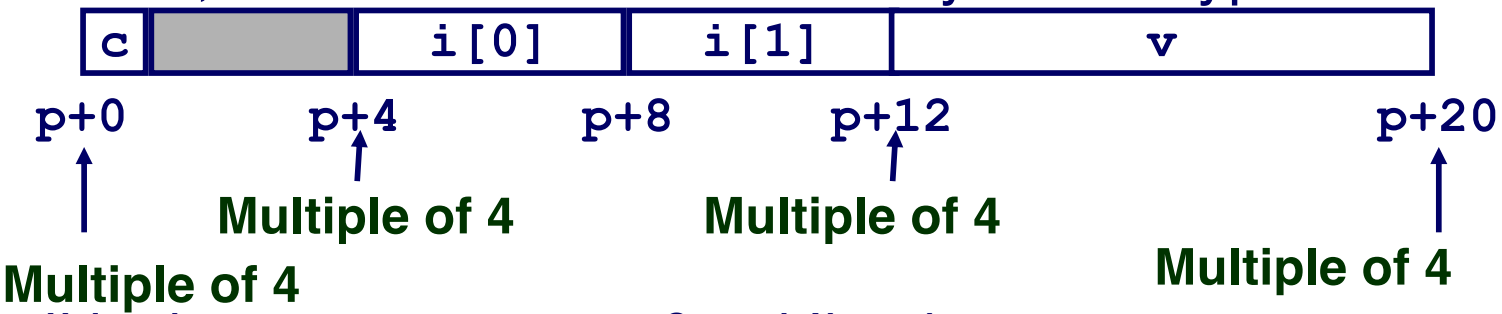
Windows (including Cygwin):

- K = 8, due to `double` element



Linux:

- K = 4; `double` treated like a 4-byte data type



Overall Alignment Requirement

```
struct S2 {
  double x;
  int i[2];
  char c;
} *p;
```

p must be multiple of:
- **8 for Windows**
- **4 for Linux**

p+0 p+8 p+12 p+16 **Windows:** p+24 **Linux:** p+20

```
struct S3 {
  float x[2];
  int i[2];
  char c;
} *p;
```

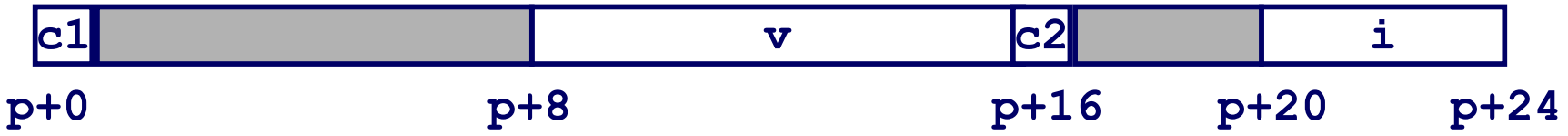
p must be multiple of 4 (in either OS)

p+0 p+4 p+8 p+12 p+16 p+20

Ordering Elements Within Structure

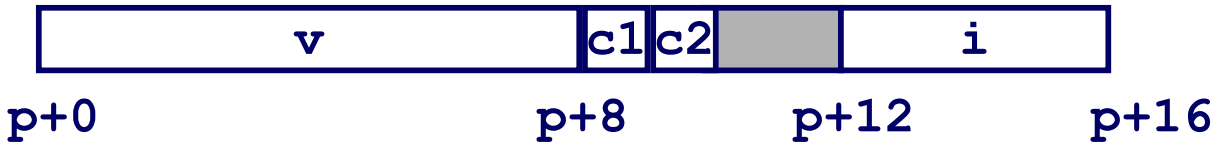
```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

10 bytes wasted space in Windows




```
struct S5 {
  double v;
  char c1;
  char c2;
  int i;
} *p;
```

2 bytes wasted space

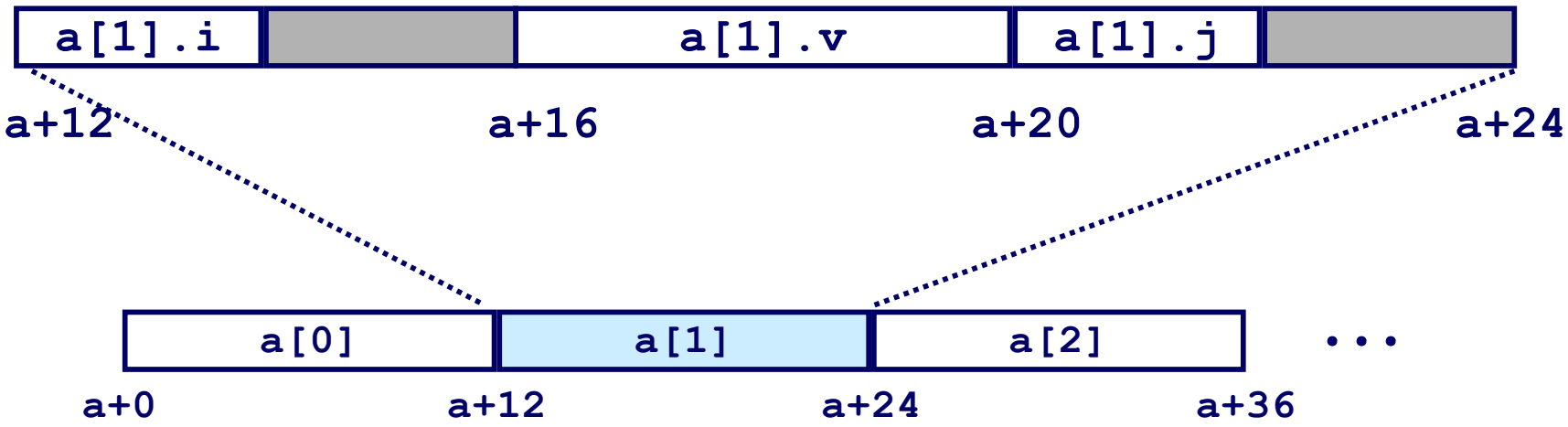


Arrays of Structures

Principle

- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```

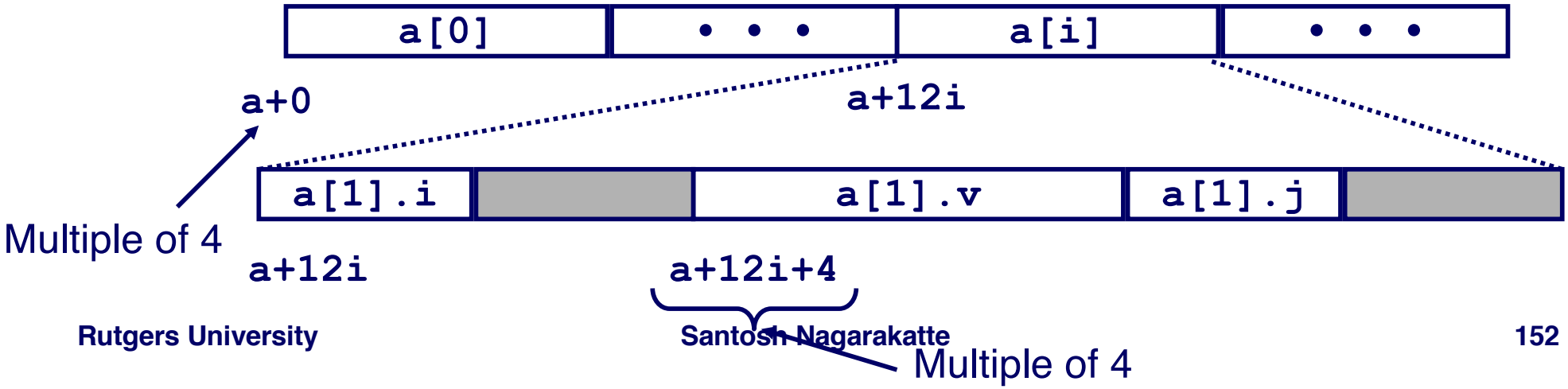


Satisfying Alignment within Structure

Achieving Alignment

- Starting address of structure array must be multiple of worst-case alignment for any element
  - `a` must be multiple of 4
- Offset of element within structure must be multiple of element's alignment requirement
  - `v`'s offset of 4 is a multiple of 4
- Overall size of structure must be multiple of worst-case alignment for any element
  - Structure padded with unused space to be 12 bytes

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```



Summary

Arrays in C

- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Compiler Optimizations

- Compiler often turns array code into pointer code (`zd2int`)
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops

Structures

- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment