CSCI-GA.1144-001

PAC II

Lecture 4: x86_64 Assembly Language

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
- Randy Bryant
- Dave O'Hallaron

Intel x86 Processors

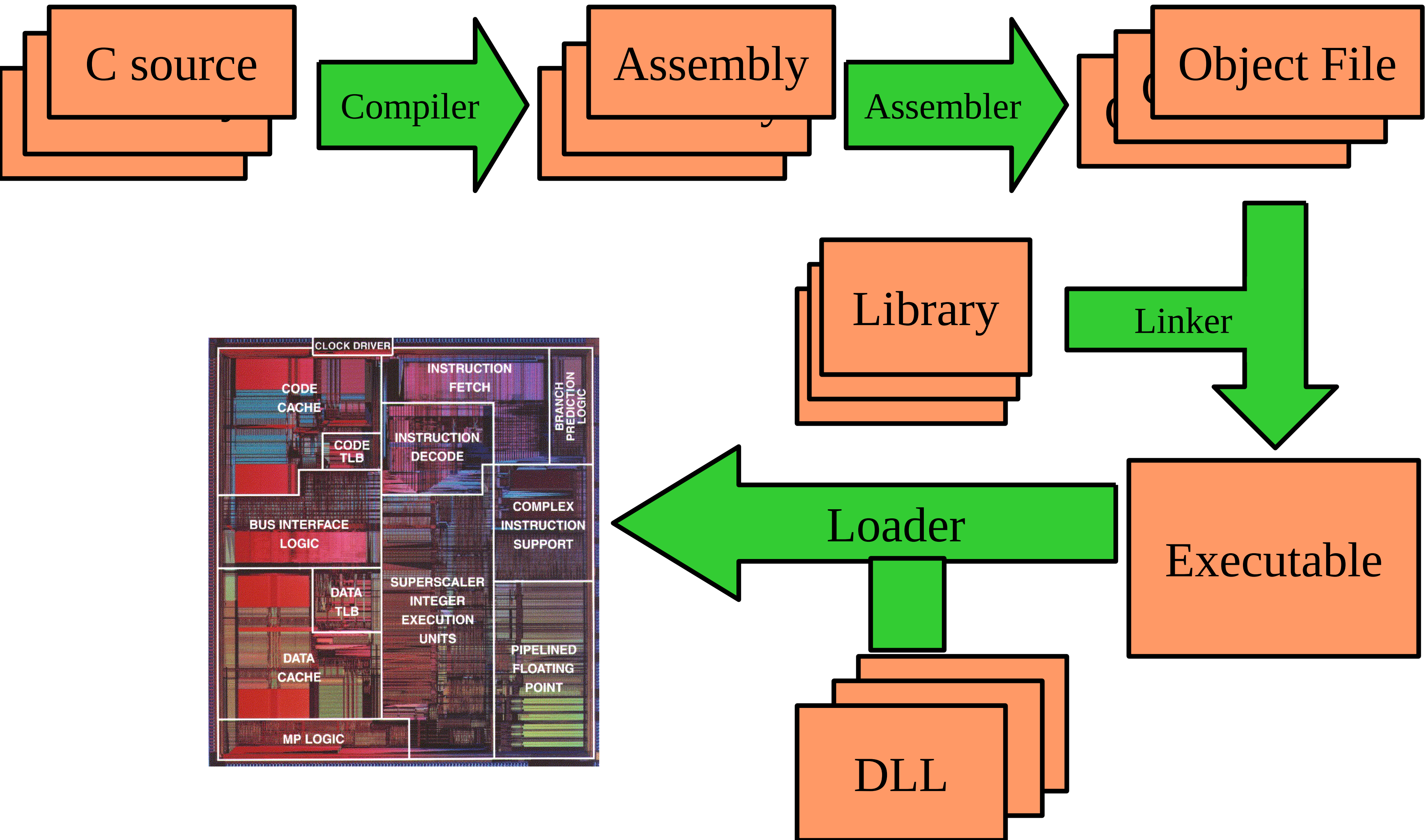
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
- Complex instruction set computer (CISC)
  - Many instructions, many formats
  - By contrast, ARM architecture (in most cell phones) is RISC

Intel x86 Evolution: Milestones

Name	*Transistors*	*MHz*
• 8086 (1978)	29K	5-10

- 8086 (1978) 29K 5-10
  - First 16-bit processor. Basis for IBM PC & DOS
  - 1MB address space
- 386 (1985) 275K 16-33
  - First 32 bit processor , referred to as **IA32**
  - Capable of running Unix
- Pentium 4 (2004) 125M 2800-3800
  - First 64-bit processor, referred to as **x86-64**
- Core i7 (2008) 731M 2667-3333
- Xeon E7 (2011) 2.2B ~2400

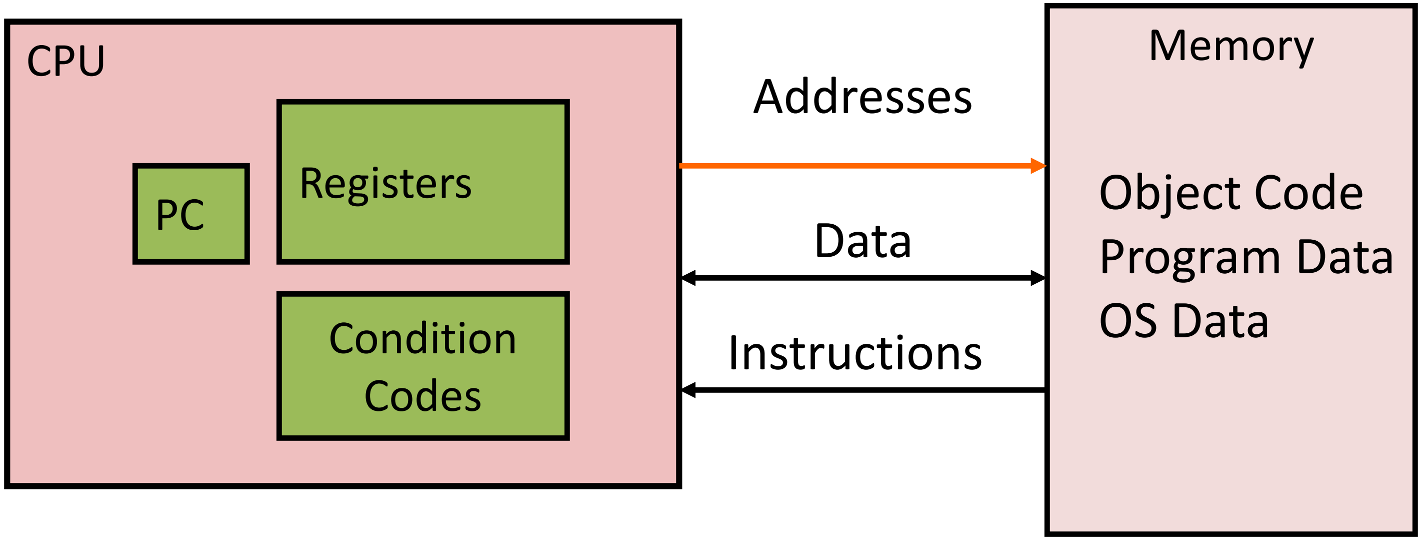
Source Code to Execution



Outline

- Assembly primer
- Moving data
- Arithmetic and logic operations
- Control
- Procedures and the stack
- Register saving convention
- Manipulating data
  - Arrays
  - Structures
  - Alignment

Assembly Programmer's View



- Execution context
  - PC: Program counter
    - Address of next instruction
    - Called "RIP" (x86-64)
  - Registers
    - Heavily used program data
  - Condition code registers
    - 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

Assembly Data Types

- “Integer” data of 1, 2, or 4 bytes
  - Represent either data value
  - or address (untyped pointer)
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No arrays or structures

3 Kind of Assembly Operations

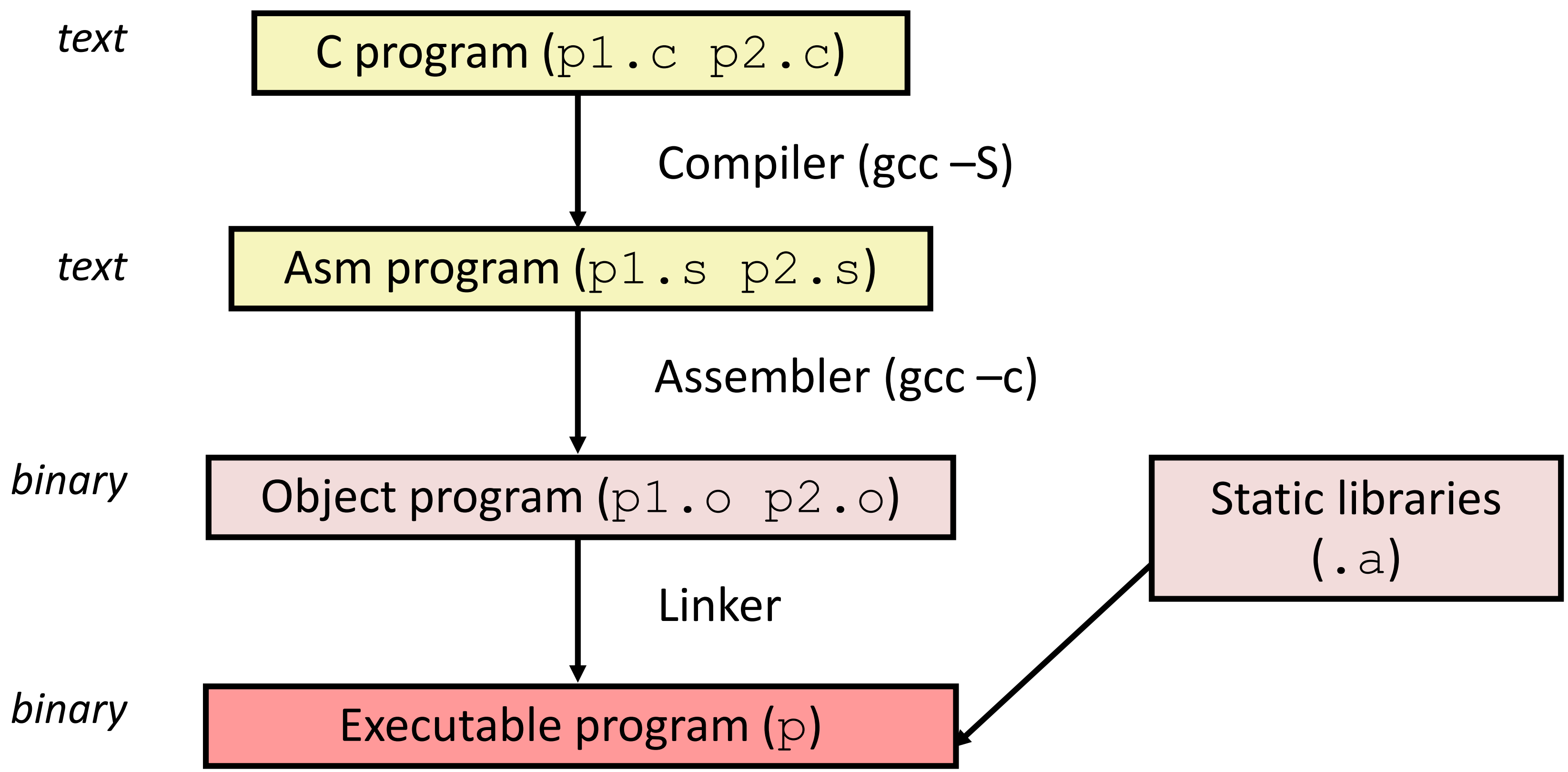
- Perform arithmetic on register or memory data
  - Add, subtract, multiplication…
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Turning C into Object Code

– Code in files **`p1.c p2.c`**
– Compile with command: **`gcc –Og p1.c p2.c -o p`**

Optimization level

Output file is 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
```

Obtain with command

```
gcc -Og -S sum.c
```

Produces file `sum.s`

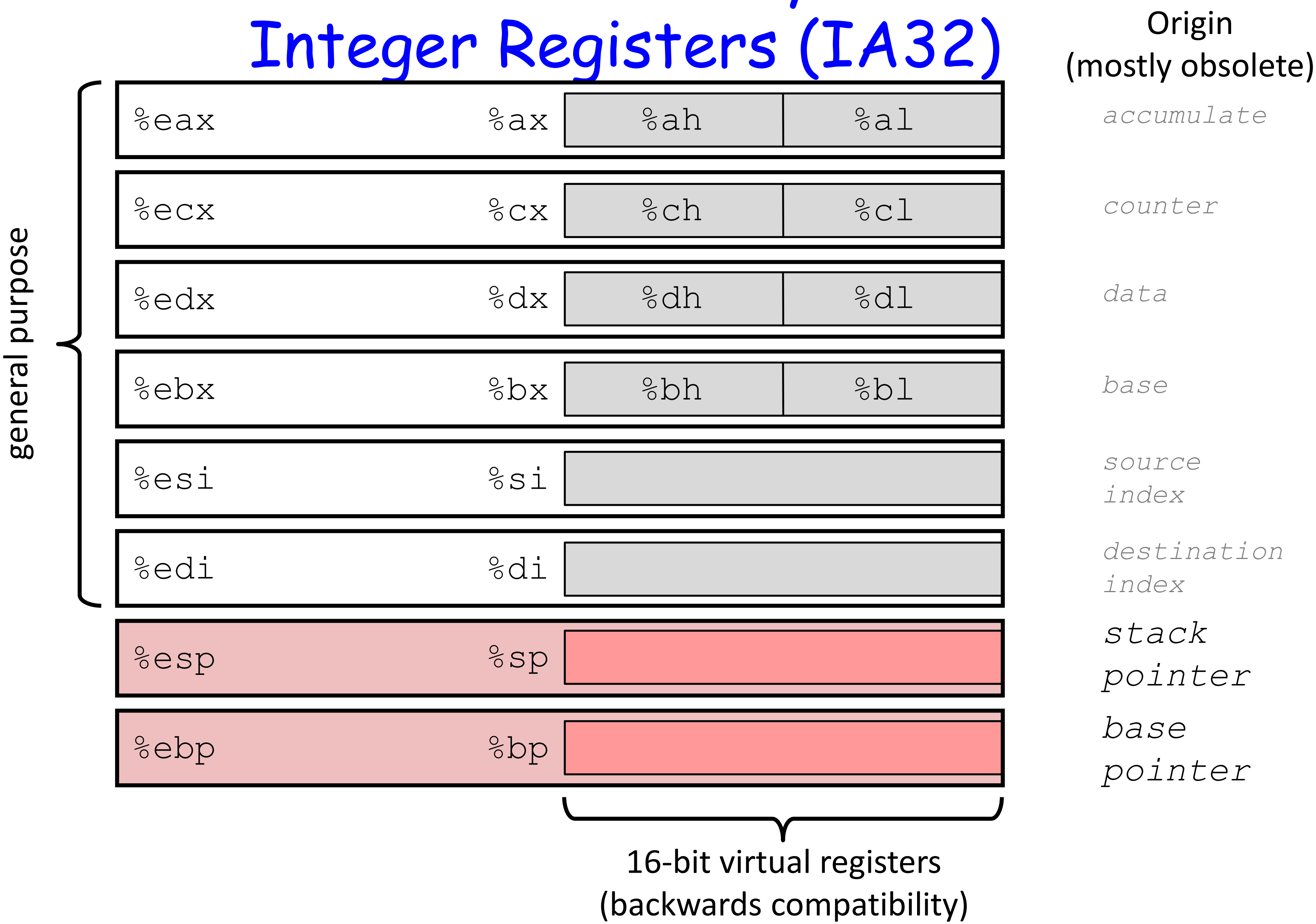
Warning: Will get very different results on different machines due to different versions of gcc and different compiler settings.

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: Integer Registers (IA32)



Moving Data

Moving Data

- Moving Data

  movq ***Source, Dest***

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

%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp
%rbp
%rN

`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movl`	*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;`

No memory-to-memory instruction

movq

C Declaration	Intel Data Type	Assembly code suffix	Size (bytes)
Char	Byte	b	1
Short	Word	w	2
Int	Double Word	l	4
Long	Quad Word	q	8
Pointer	Quad Word	q	8

If one of the operands is memory. How do we represent a memory address?

Simple Memory Addressing Modes

- Normal (R) → Mem[Reg[R]]
  - Register R specifies memory address

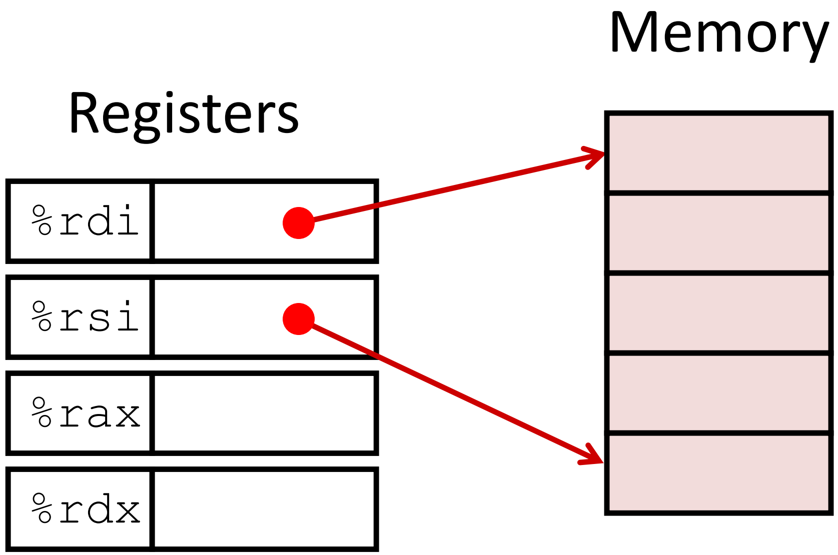
```
movq (%rcx),%rax
```

- Displacement D(R) → Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

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

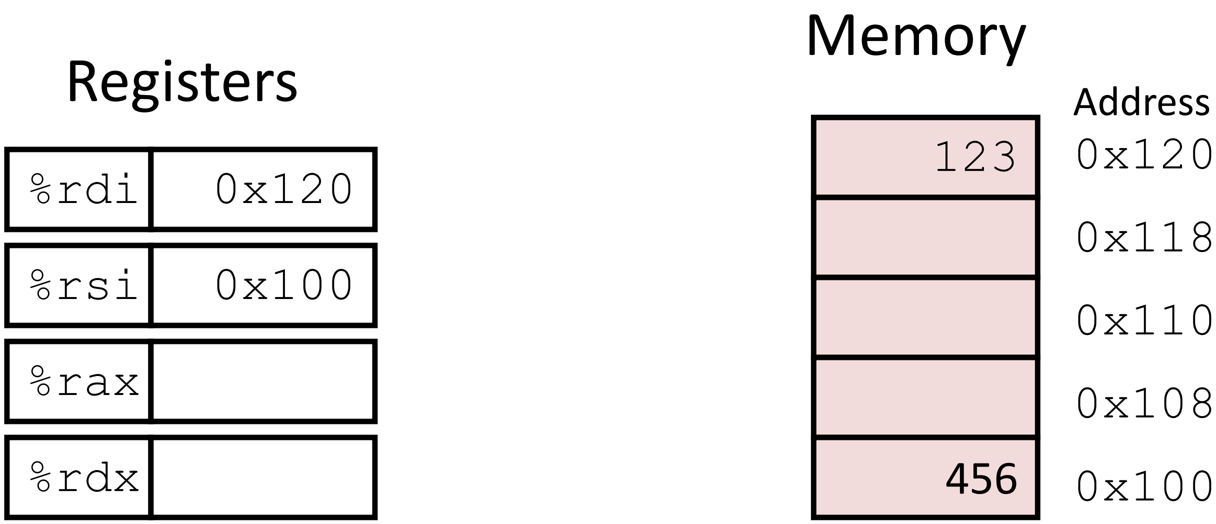
Example of Simple Addressing Modes

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



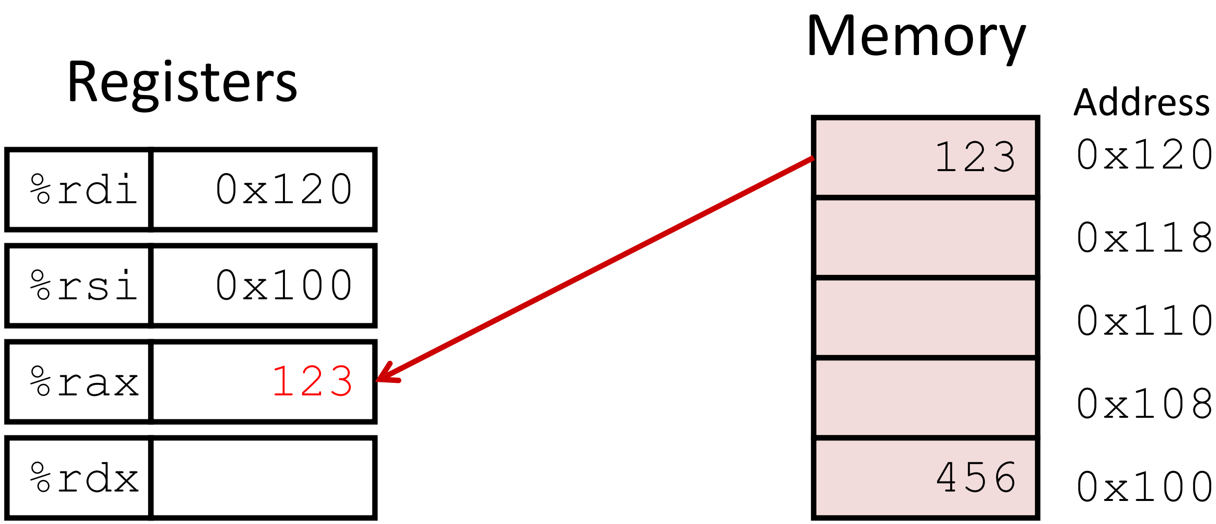

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Example of Simple Addressing Modes



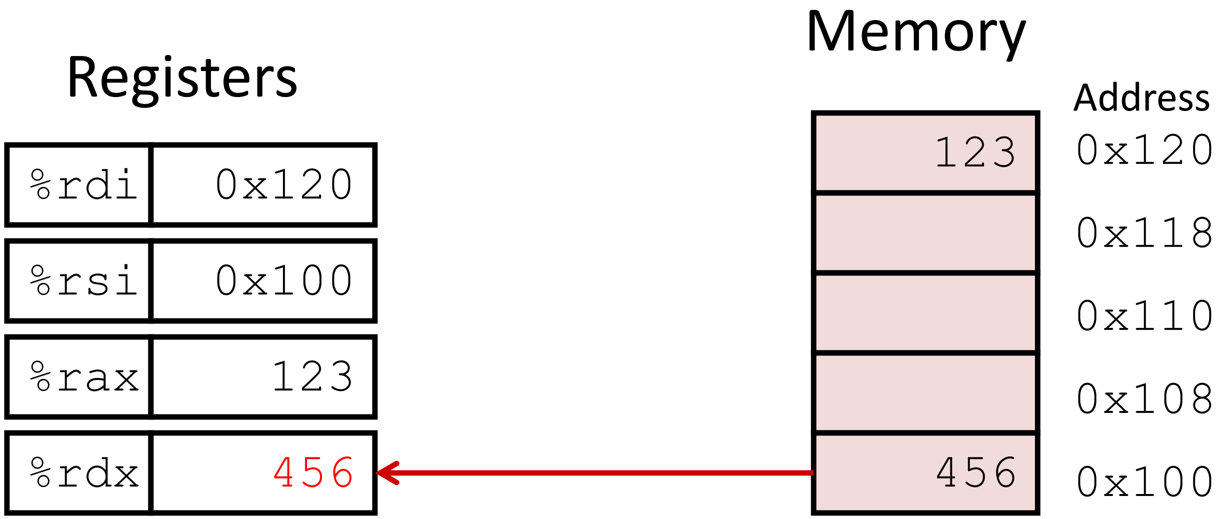

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Example of Simple Addressing Modes



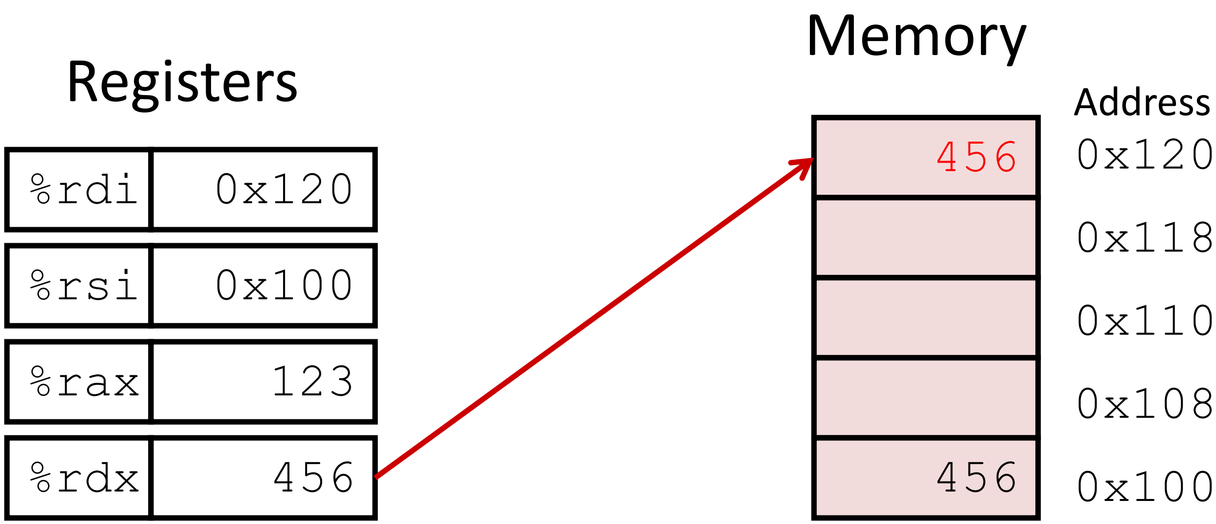

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Example of Simple Addressing Modes



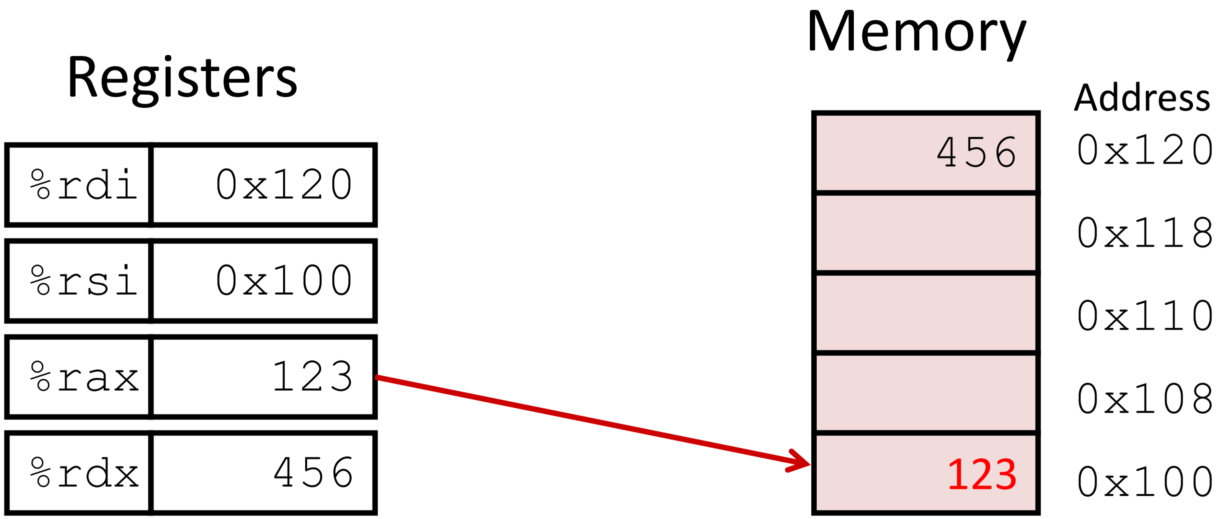

```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Example of Simple Addressing Modes




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

Example of Simple Addressing Modes




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

General Memory Addressing Modes

- Most General Form

D (Rb, Ri, S)

Constant displacement (cannot be bigger than 4 bytes)

Base register (any of the 16 registers)

Index register (any if the 16 registers except %rsp)

Scale (1,2,4,8)

Mem[Reg[Rb]+S*Reg[Ri]+ D]

- 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 <u>without a memory access</u>
    - 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
```

Arithmetic & Logic Operations

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*

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
```

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**

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
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions

Multiplication

- **Unsigned**
  - form 1: imulq s, d
    - d = s * d
    - multiply two 64-bit operands and put the result in 64-bit operand
  - form 2: mulq s
    - one operand is rax
    - The other operand given in the instruction
    - product is stored in rdx (high-order part) and rax (low order part) → full 128-bit result
- **Signed**
  - form 1: imulq s, d
    - d = s * d
    - multiply two 64-bit operands and put the result in 64-bit operand
  - form 2: imulq s
    - one operand is rax
    - The other operand given in the instruction
    - product is stored in rdx (high-order part) and rax (low order part) → full 128-bit result

Division

- **Unsigned**
  - divq s
    - Dividend given in rdx (high order) and rax (low order)
    - Divisor is s
    - Quotient stored in rax
    - Remainder stored in rdx
- **Signed**
  - idivq s
    - Dividend given in rdx (high order) and rax (low order)
    - Divisor is s
    - Quotient stored in rax
    - Remainder stored in rdx

Useful Instruction for Division

cqto

- No operands
- Takes the sign bit from rax and replicates it in rdx

Control

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**)

Registers

`%rax`	`%r8`
`%rbx`	`%r9`
`%rcx`	`%r10`
`%rdx`	`%r11`
`%rsi`	`%r12`
`%rdi`	`%r13`
`%rsp`	`%r14`
`%rbp`	`%r15`

`%rip`	**Instruction pointer**

`CF`	`ZF`	`SF`	`OF`	**Condition codes**

Setting Condition Codes Implicitly

- Can be implicitly set by arithmetic operations

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

  CF (Carry flag) set if carry out from most significant bit (<u>unsigned</u> overflow)

  ZF (Zero flag) set if `t == 0`

  SF (Sign flag) set if `t < 0` (as signed)

  OF (Overflow flag) set if two's complement overflow (signed)

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

- Not set by **`lea`** instruction

Effect of Logical Operations

- The carry and overflow flags are set to zero.
- For shift instructions:
  - The carry flag is set to the value of the last bit shifted out.
  - Overflow flag is set to zero.

INC and DEC instructions

- Affect the overflow and zero flags
- Leave carry flag unchanged

Setting Condition Codes Explicitly

- Can also be explicitly set

cmpl b,a set condition codes based on a-b (computing a-b without storing the result in any 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 (a-b) results in signed overflow

Setting Condition Codes Explicitly

- Can also be explicitly set

  testq b,a set condition codes based on value of **(*a* & *b*)** *without storing the result in any destination*

ZF set if `(a&b)= zero`

SF set if `(a&b) < 0`

Setting Condition Codes

Important

The processor does not know if you are using signed or unsigned integers.
OF and CF are set for every arithmetic operation.

What do we do with condition codes?

1. Setting a single byte to 0 or 1 based on some combination of the condition codes.
2. Conditionally jump to other parts of the program.
3. Conditionally transfer data.

Reading Condition Codes

- ***SetX dest***

Sets the lower byte of some register based on combinations of condition codes and does not alter remaining 7 bytes. Destination can also be memory location.

This set of instructions is usually used after a comparison.

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)

Example

```c
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 >
        movzbq  %al, %rax       # Zero rest of %rax
        ret
```

x86-64 Integer Registers

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

– Can reference low-order byte

What do we do with condition codes?

1. Setting a single byte to 0 or 1 based on some combination of the condition codes.
2. Conditionally jump to other parts of the program.
3. Conditionally transfer data.

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)

What do we do with condition codes?

1. Setting a single byte to 0 or 1 based on some combination of the condition codes.
2. Conditionally jump to other parts of the program.
3. Conditionally transfer data.

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
```

How to implement loops?

- do-while
- while
- for

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

General "Do-While" Translation

C Code

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

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;
}
```

Procedures and the Stack

Suppose P calls Q

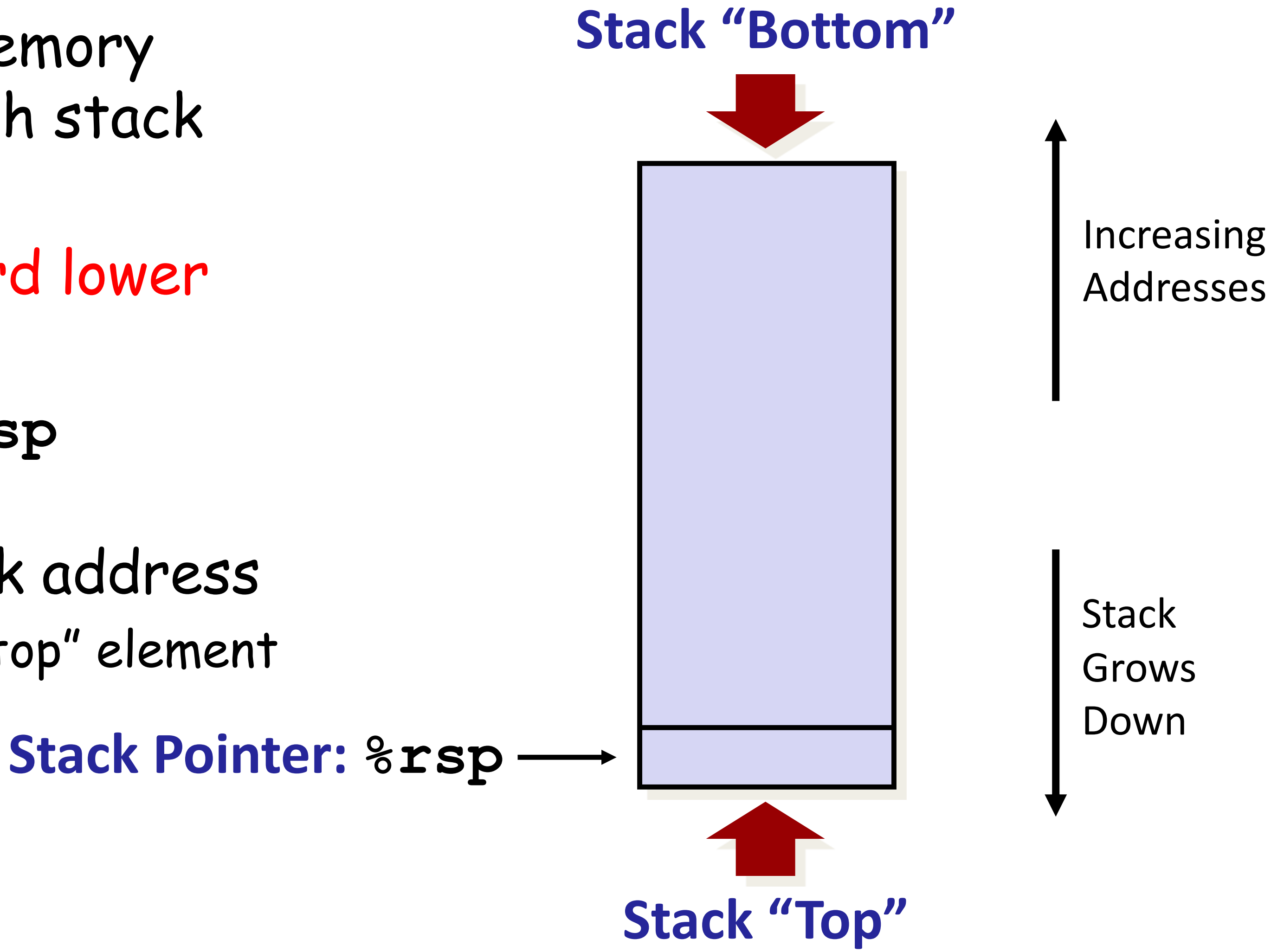
- **Passing control**
  - To beginning of procedure code
  - Back to return point
- **Passing data**
  - Procedure arguments
  - Return value
- **Memory management**
  - Allocate during procedure execution
  - Deallocate upon return

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

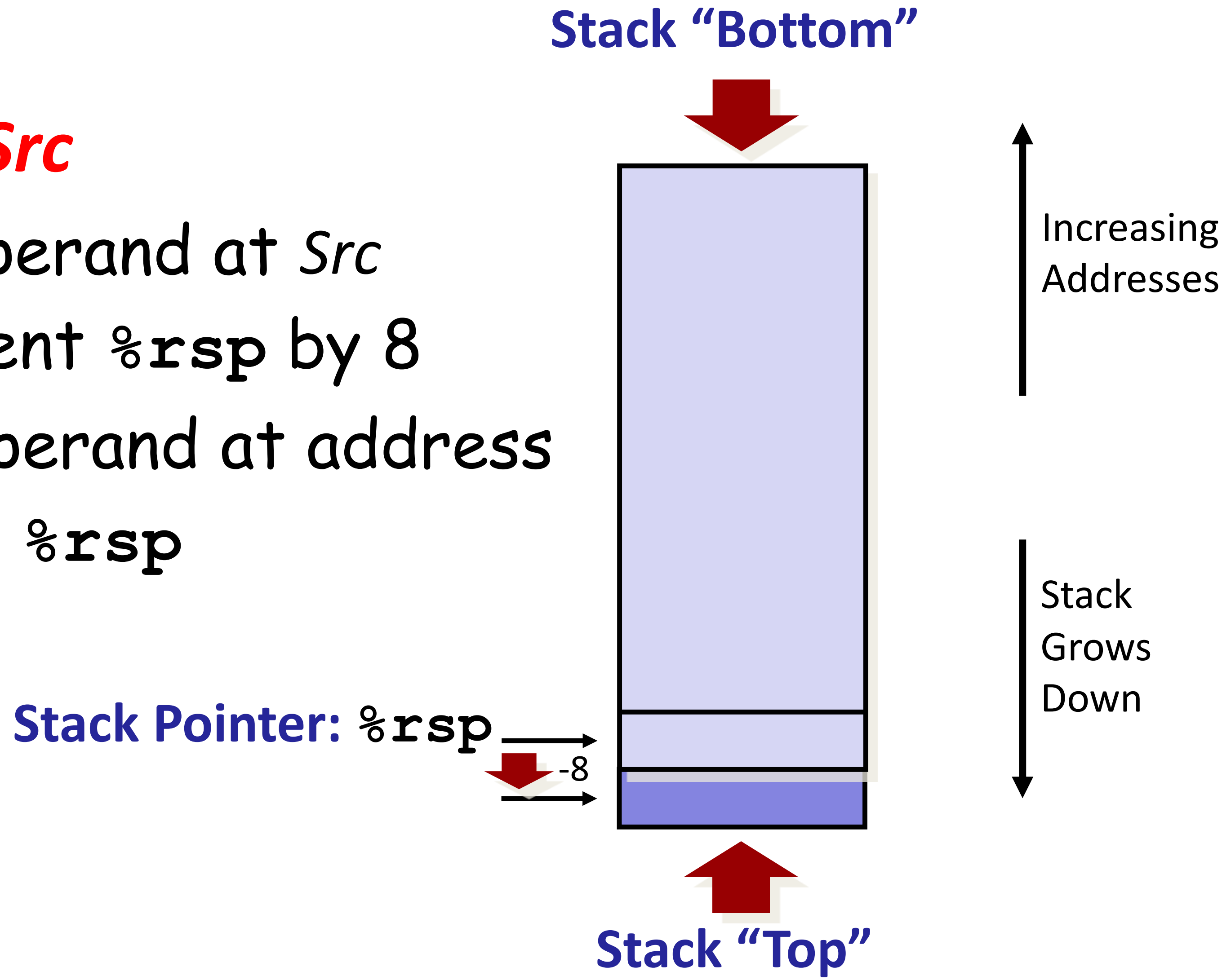
x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
  - address of "top" element



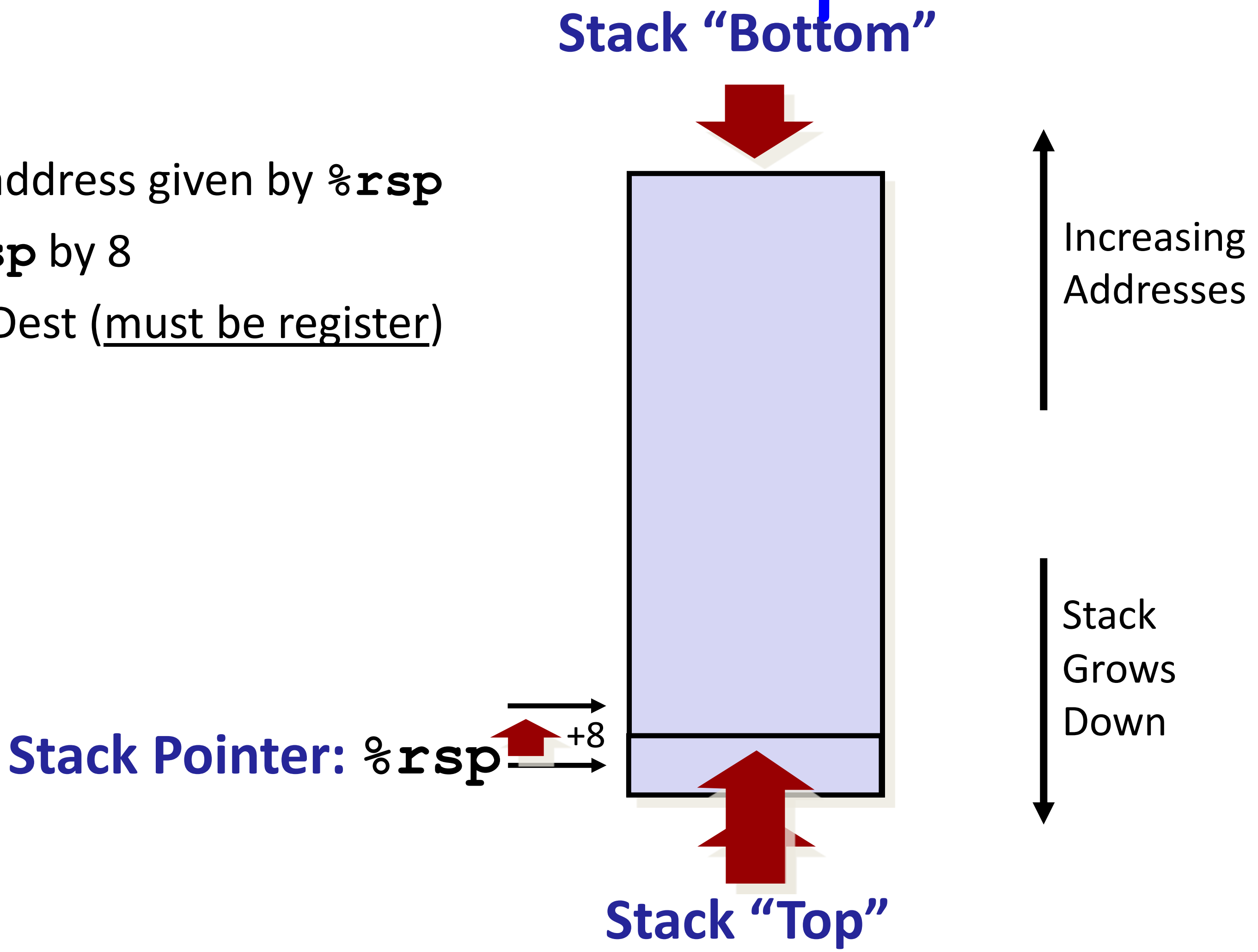
x86-64 Stack: Push

- **`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 (<u>must be register</u>)



Examples:

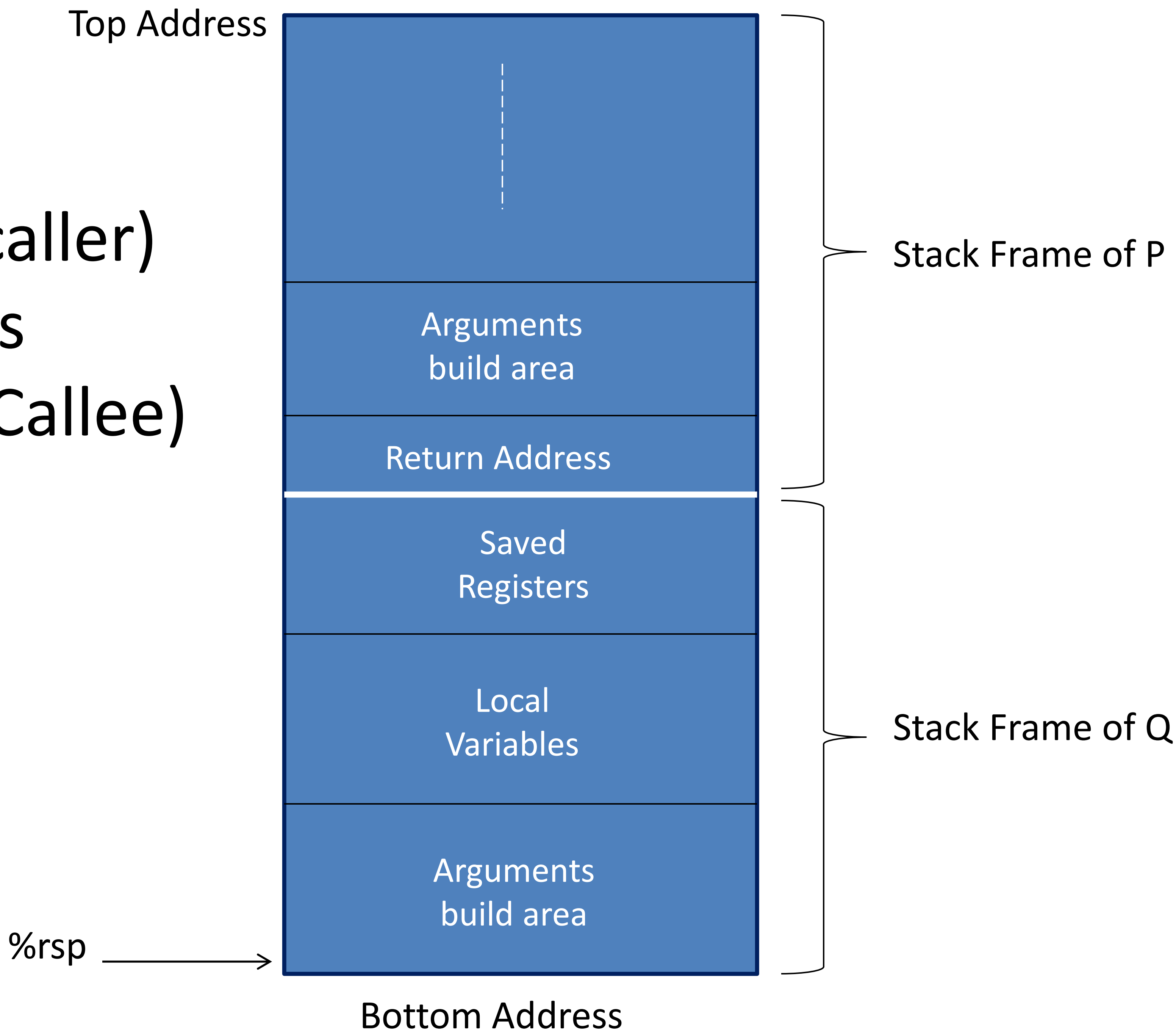
```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

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

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

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

P (caller)
calls
Q (Callee)



When P calls Q

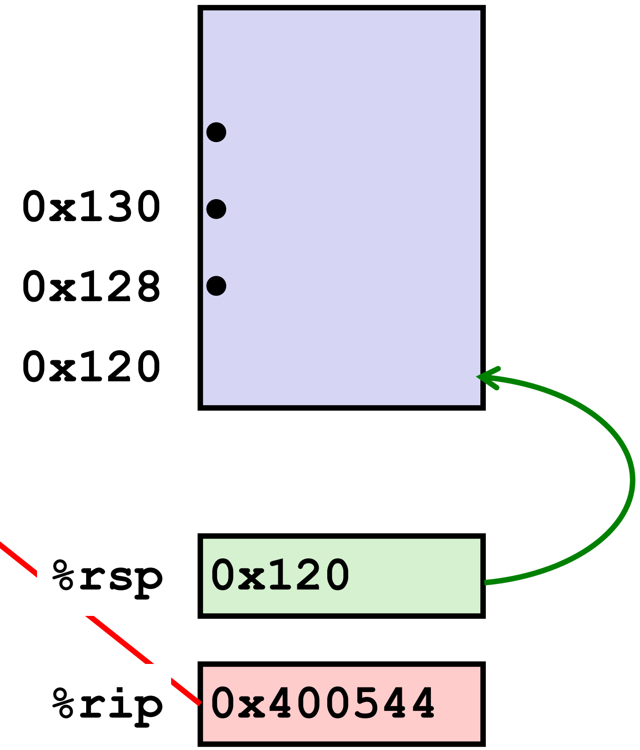
- P is suspended and control moves to Q.
- A stack frame is setup on top of the stack for Q
- That stack frame contains:
  - saved registers
  - local variables
  - arguments if Q is calling another function
- Some procedures may not need a stack frame (why?).

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
- Procedure return: `ret`
  - Pop address from stack
  - Jump to address

Example

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

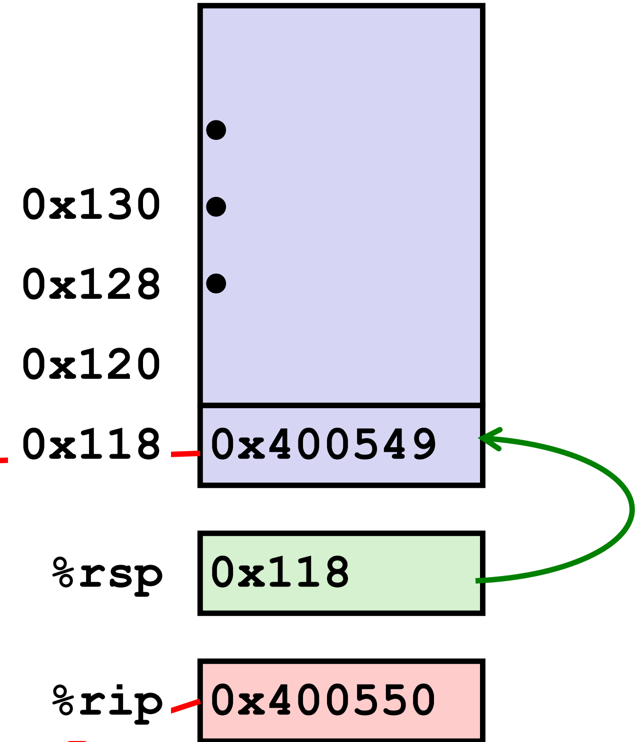



```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

Example

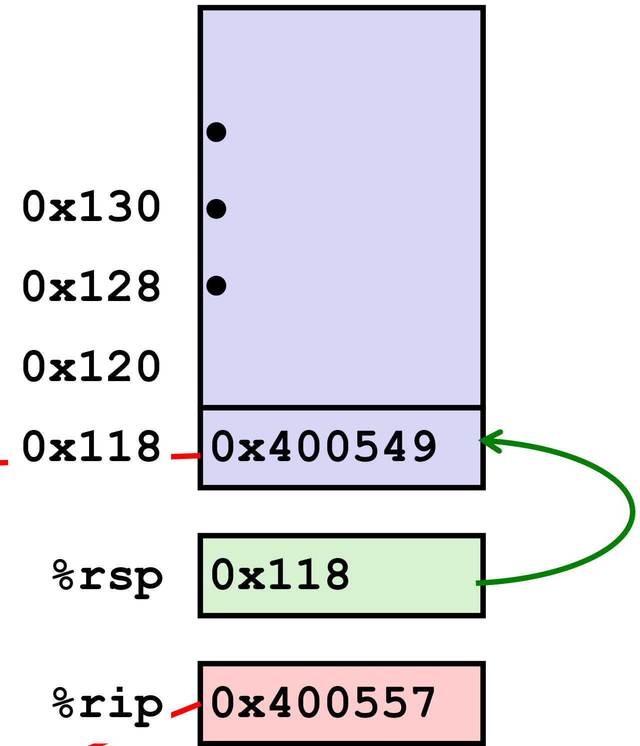
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

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



Example

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

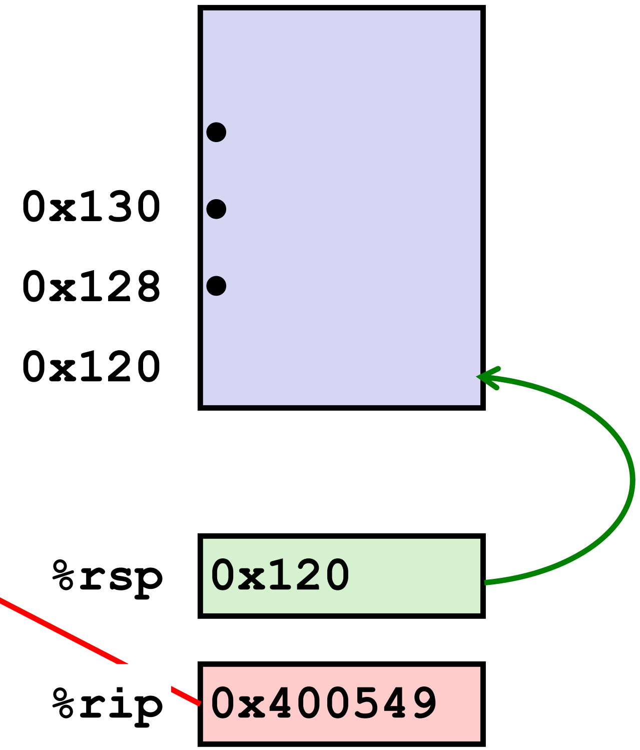



```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

Example

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

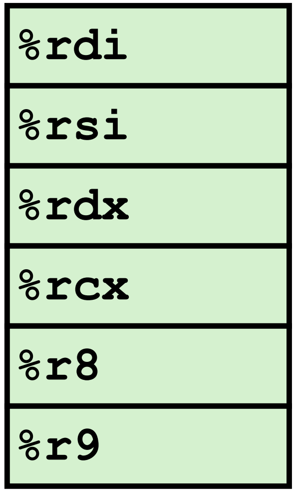
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



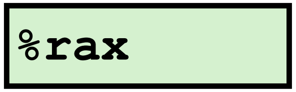
Procedure Data Flow

Registers

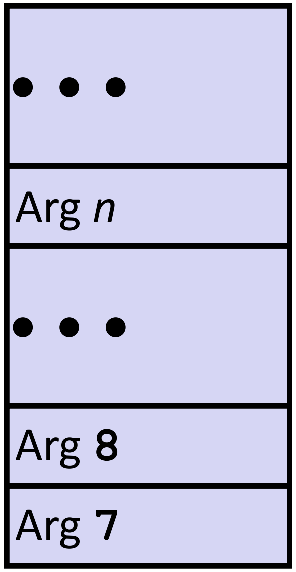
- First 6 arguments



- Return value



Stack



- Only allocate stack space when needed
- When passing parameters on the stack, all data sizes are rounded up to be multiple of eight.

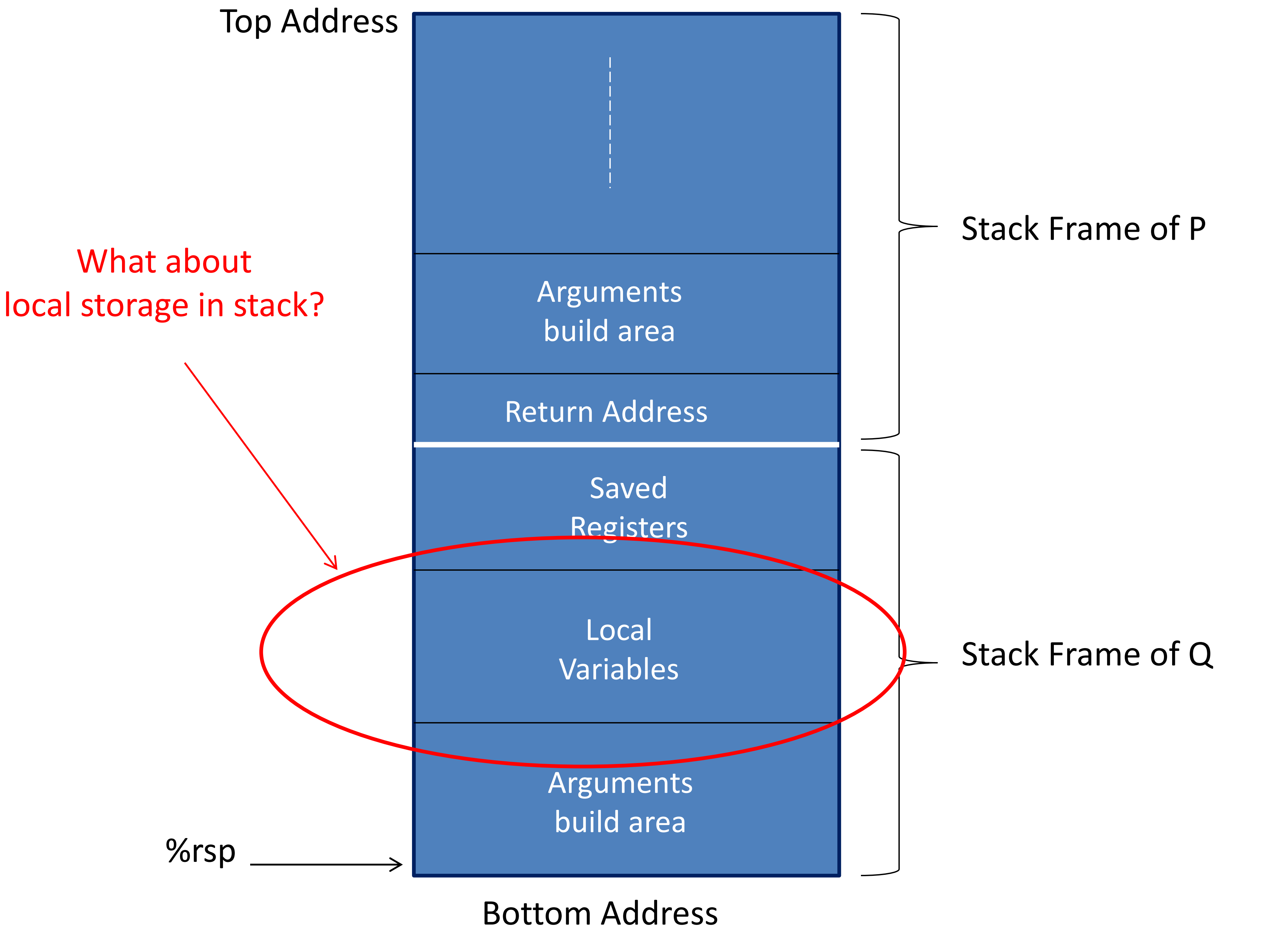
Example:
multstore calls mult2

```
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
```


Top Address
Stack Frame of P
What about
local storage in stack?
Arguments
build area
Return Address
Saved
Registers
Local
Variables
Stack Frame of Q
Arguments
build area
%rsp
Bottom Address

When is local storage needed?

- Not enough registers
- A variable in high-level language is referred by its ("&" in C) so needs to have address!
- Arrays, structures, ...

Registers Usage Convention

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
```

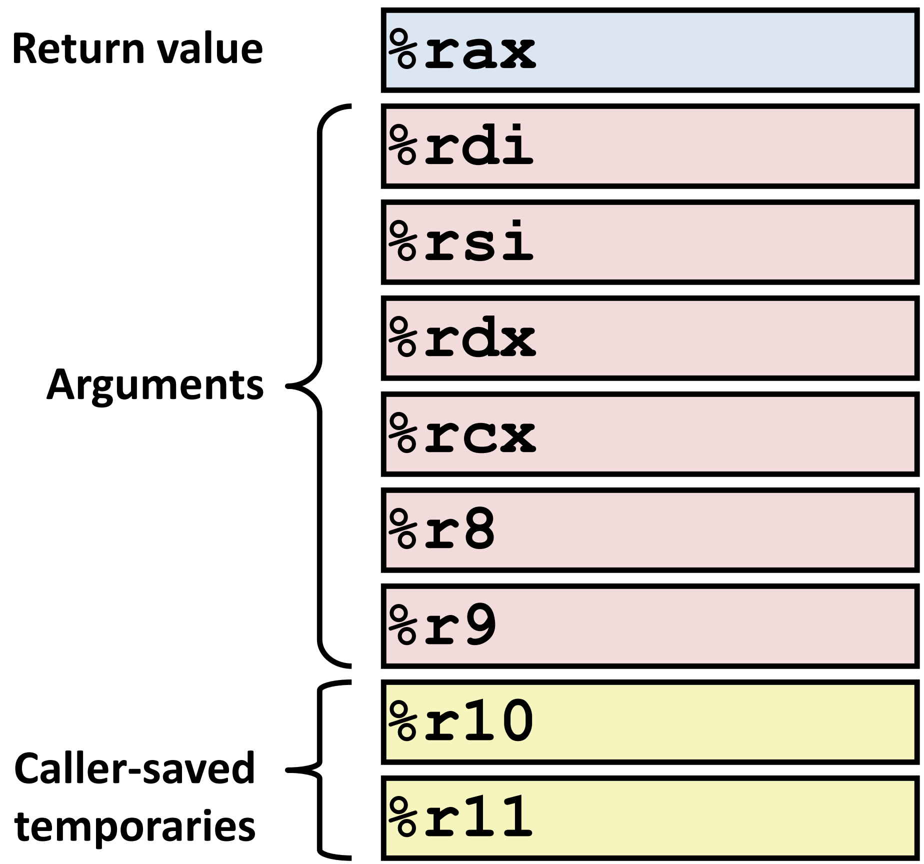
  - Contents of register `%rdx` overwritten by `who`
  - This could be trouble → something should be done!
    - Need some coordination

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

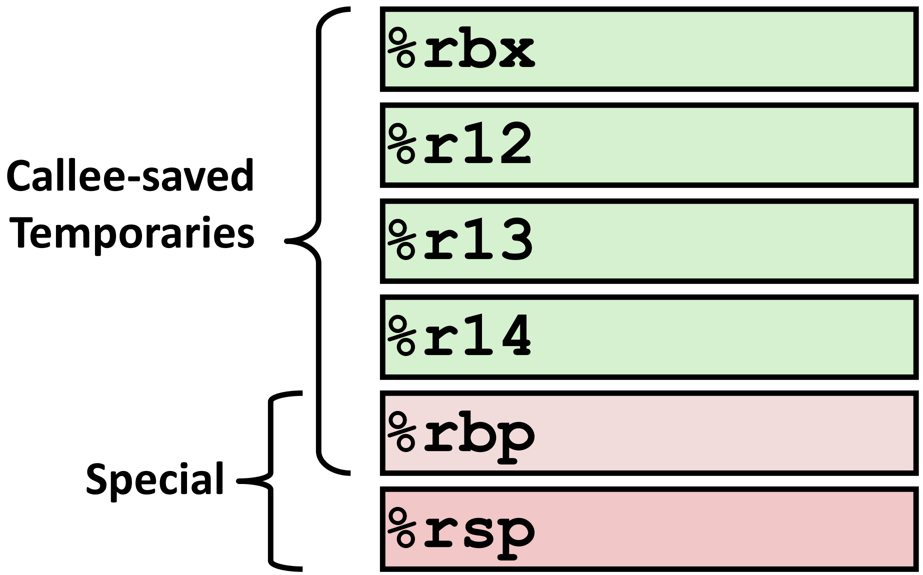
x86-64 Linux Register Usage #1

- `%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

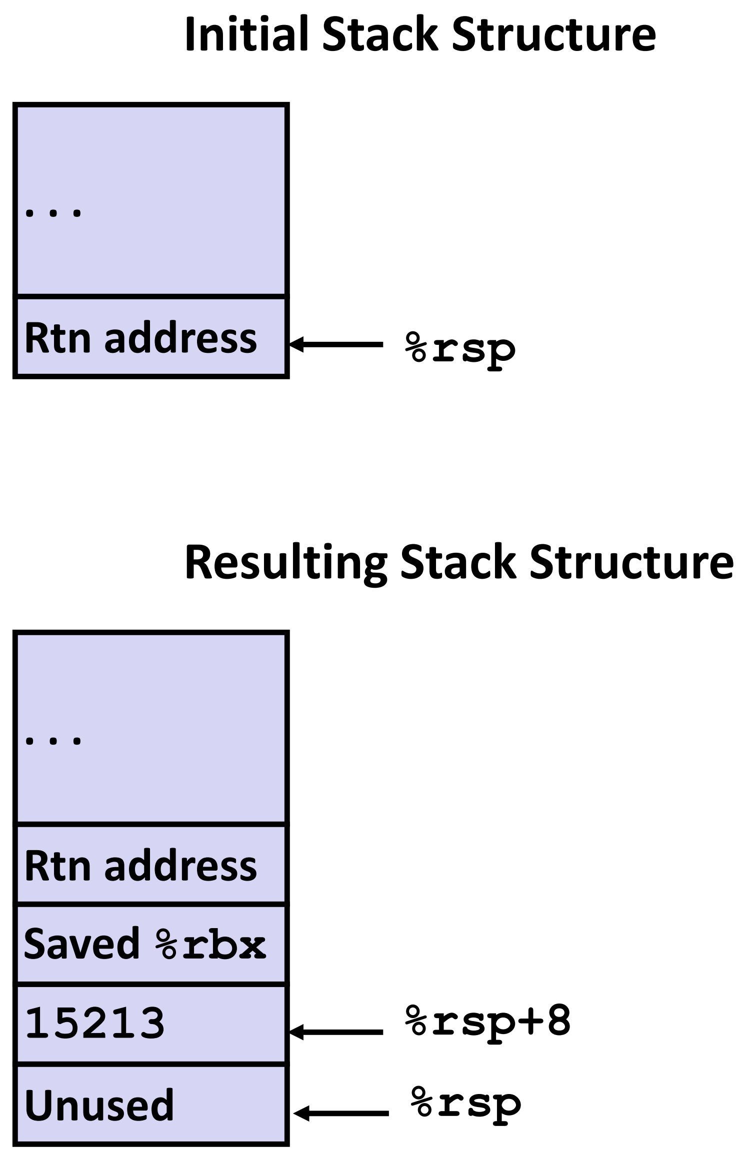
- **%rbx**, **%r12**, **%r13**, **%r14**
  - Callee-saved
  - Callee must save & restore
- **%rbp**
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- **%rsp**
  - Special form of callee save
  - Restored to original value upon exit from procedure



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
```

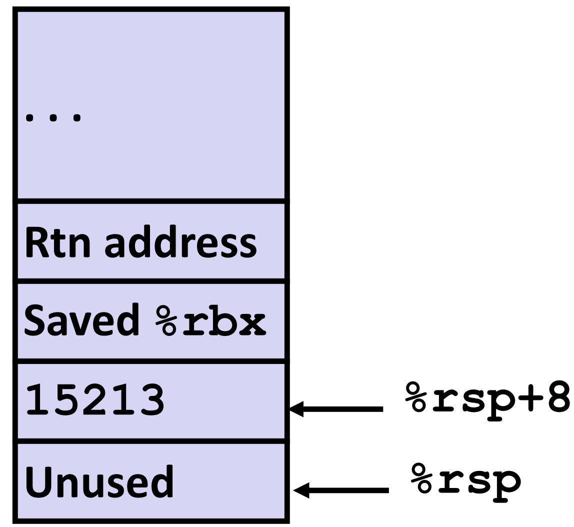


Callee-Saved Example #2

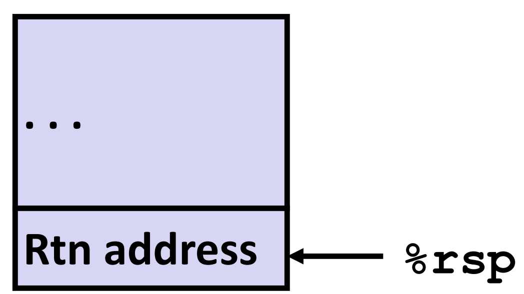
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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



Manipulating Data

How are data structures, like arrays, presented and manipulated in assembly?

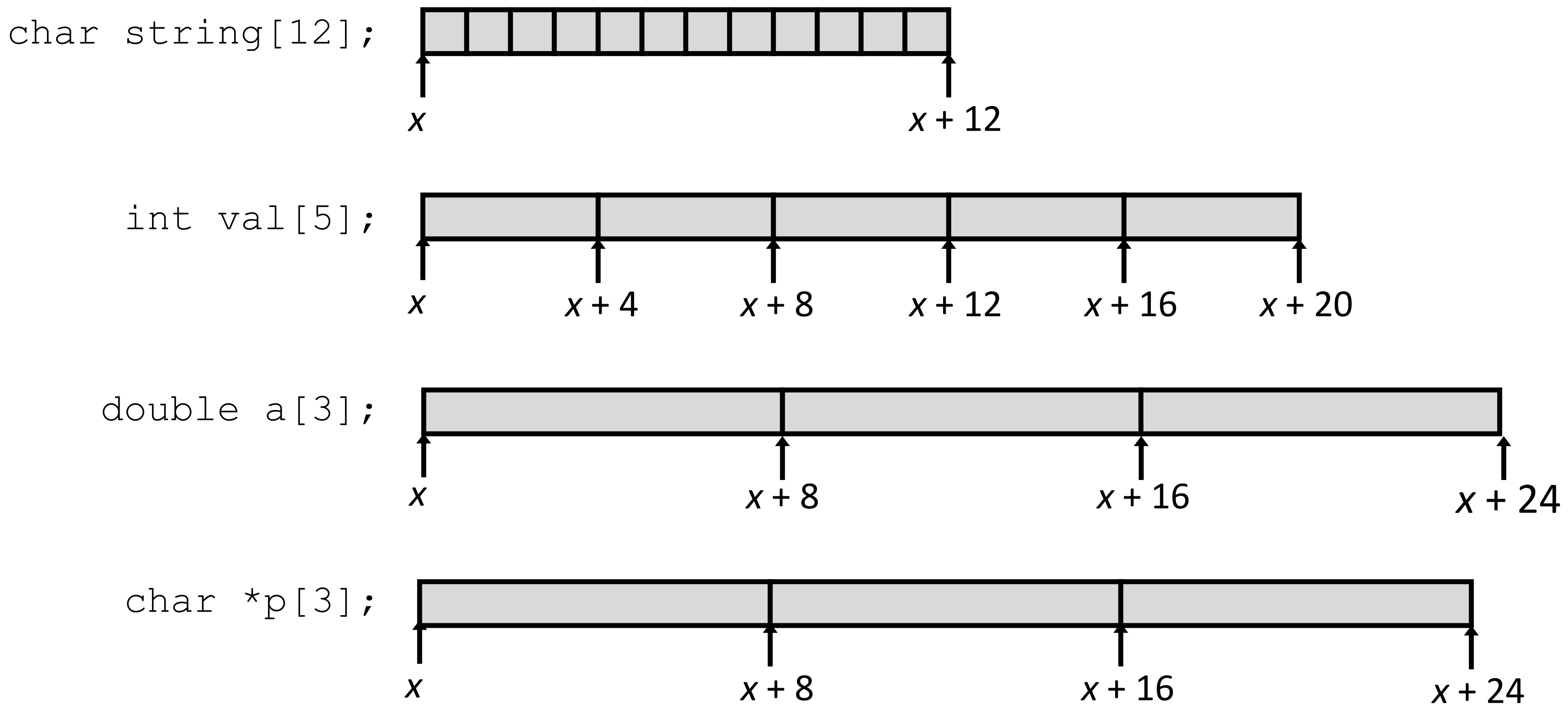
Array Allocation

- Basic Principle

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

  – Array of data type *T* and length *L*

  – Contiguously allocated region of *L* * **sizeof**(*T*) bytes in memory



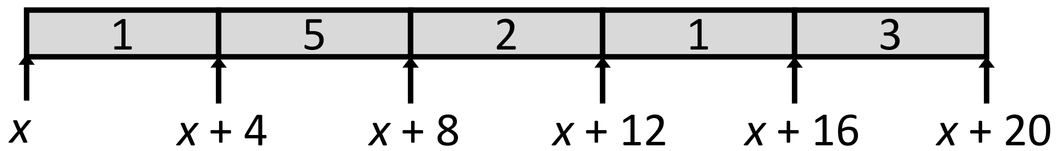
Array Access

- Basic Principle

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

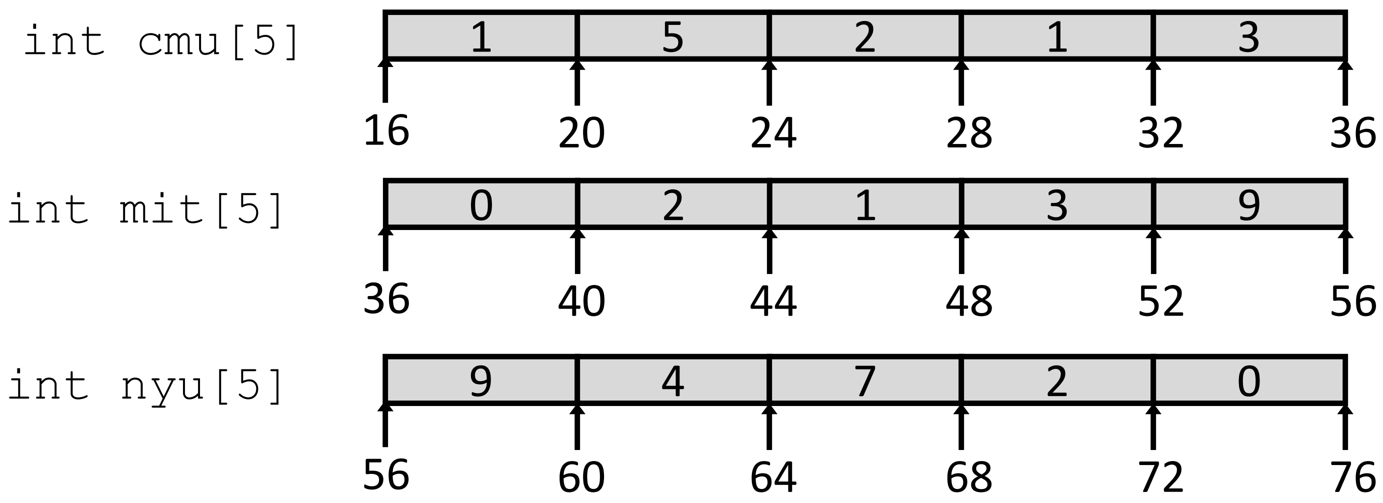
  – Array of data type *T* and length *L*

int val[5];



Array Example

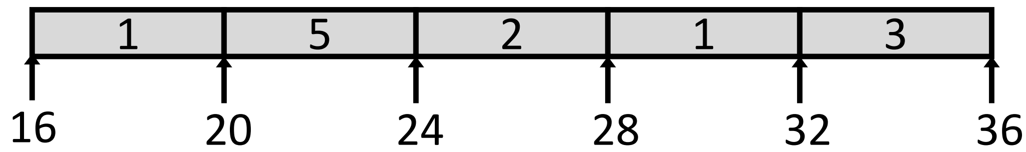
```c
#define ZLEN 5
typedef int zip_dig[ZLEN];

int cmu[5] = { 1, 5, 2, 1, 3 };
int mit[5] = { 0, 2, 1, 3, 9 };
int nyu[5] = { 9, 4, 7, 2, 0 };
```



- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

```
int nyu[5];
```




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

IA32

```
  # %rdi = z
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

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

Array Loop Example

```
void zincr(int * z) {
  int i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

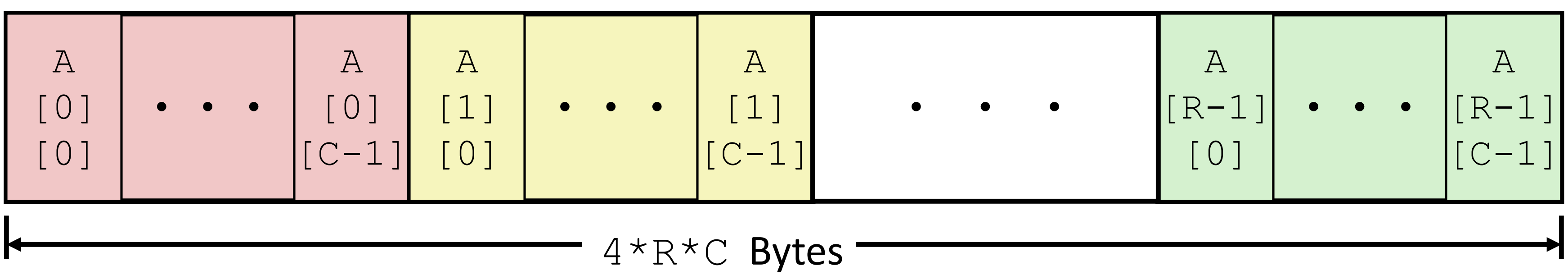
```
  # %rdi = z
  # ZLEN is 5
  movl    $0, %eax              #   i = 0
  jmp     .L3                   #   goto middle
.L4:                            # loop:
  addl    $1, (%rdi,%rax,4)     #   z[i]++
  addq    $1, %rax              #   i++
.L3:                            # middle
  cmpq    $4, %rax              #   i:4
  jbe     .L4                   #   if <=, goto loop
  rep; ret
```

Multidimensional (Nested) Arrays

- Declaration
  `T A[R][C];`
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- Array Size
  - *R* * *C* * *K* bytes
- Arrangement
  - Row-Major Ordering

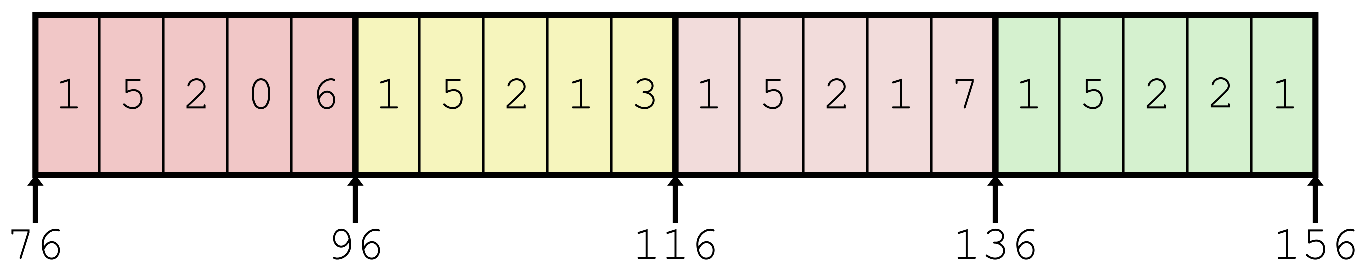
$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

```
int A[R][C];
```



Nested Array Example

```
int pgh[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

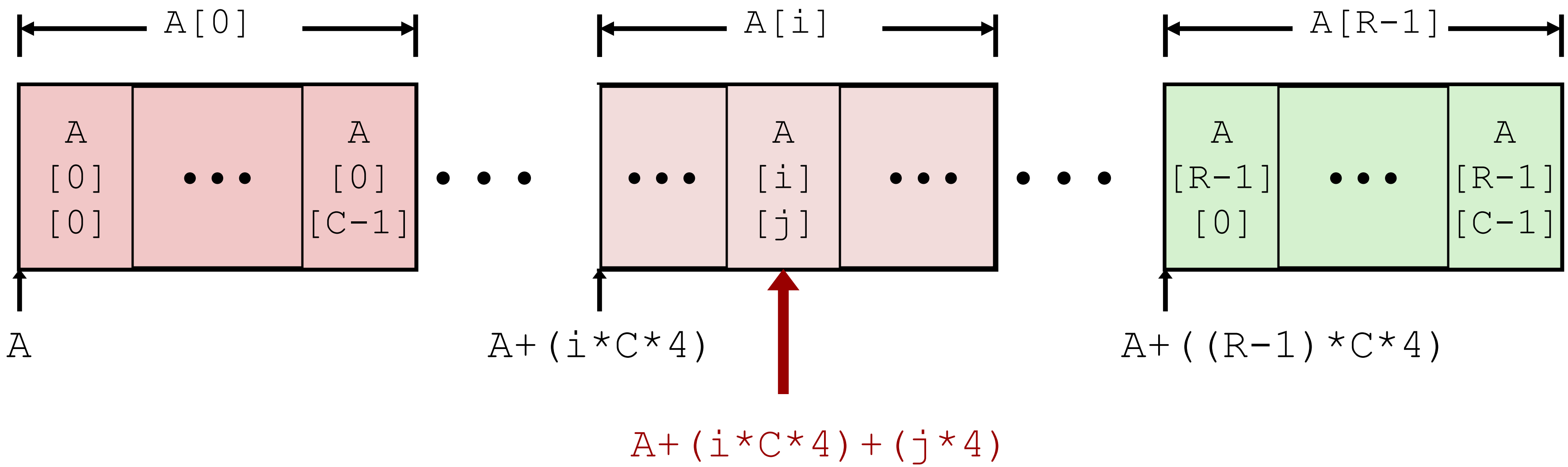


- – Variable **`pgh`**: array of 4 elements, allocated contiguously
- – Each element is an array of 5 **`int`**'s, allocated contiguously
- "Row-Major" ordering of all elements in memory

Nested Array Element Access

- Array Elements
  - **`A[i][j]`** is element of type *T*, which requires *K* bytes
  - Address **`A +`** $i * (C * K) + j * K = A + (i * C + j) * K$

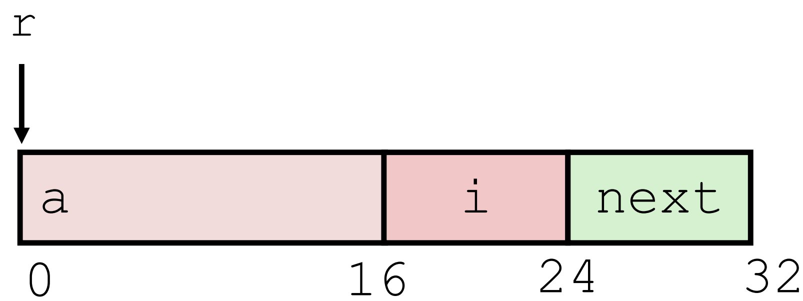
```
int A[R][C];
```



How about structures?

Structure Representation

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields
  - **Machine-level program has no understanding of the structures in the source code**

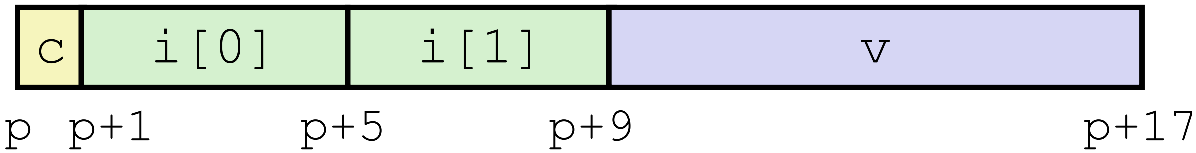
Alignment

Alignment Principles

- Aligned Data
  - Primitive data type requires *K* bytes
  - Address must be multiple of *K*
  - Required on some machines; advised on x86-64
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

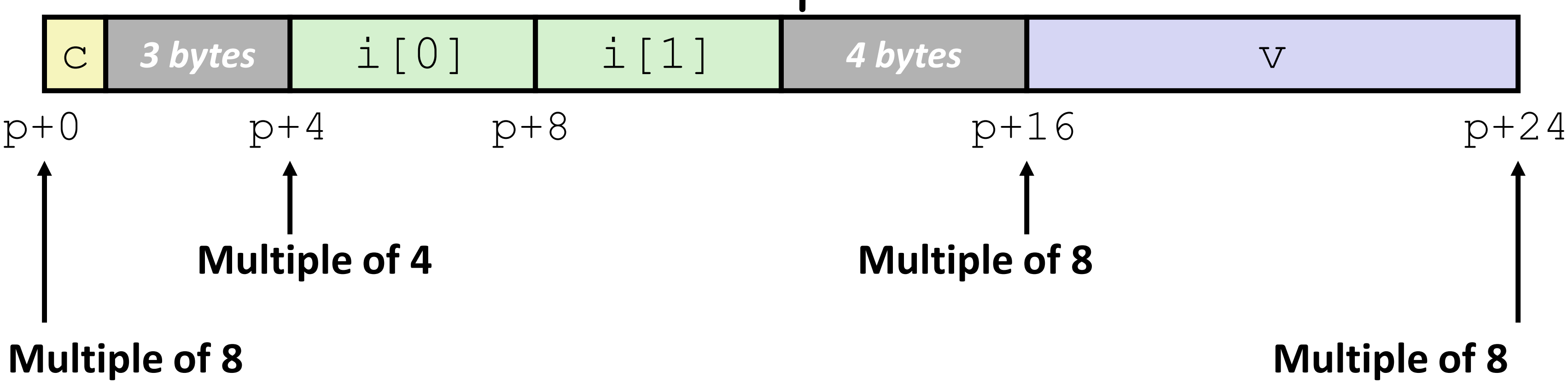
Structures & Alignment

- Unaligned Data




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

- Aligned Data
  - Primitive data type requires *K* bytes
  - Address must be multiple of *K*



Specific Cases of Alignment (x86-64)

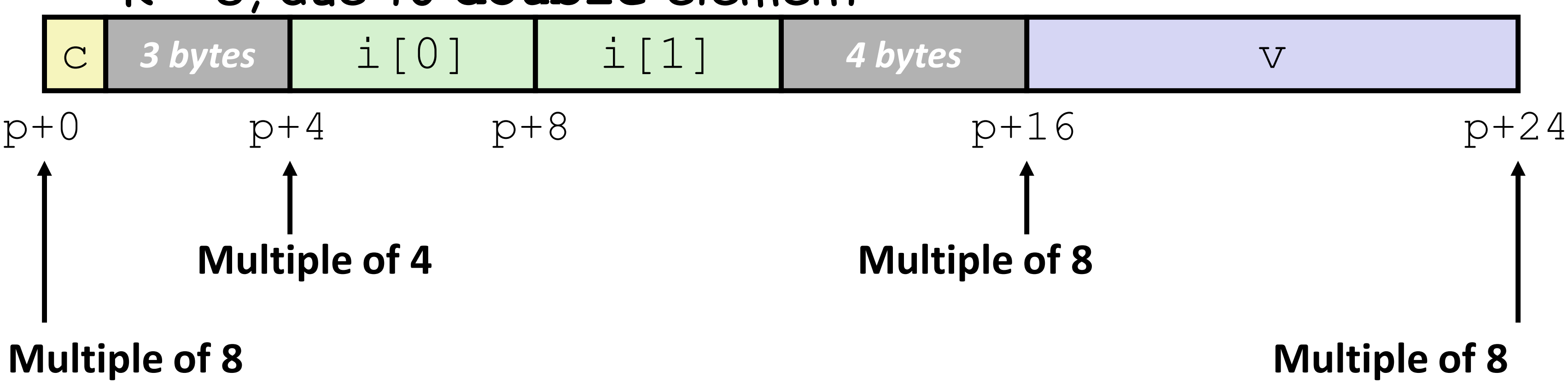
- 1 byte: **`char`**, …
  - no restrictions on address
- 2 bytes: **`short`**, …
  - address must be multiple of 2
- 4 bytes: **`int`**, **`float`**, …
  - address must be multiple of 4
- 8 bytes: **`double`**, **`long`**, **`char *`**, …
  - address must be multiple of 8
- 16 bytes: **`long double`** (GCC on Linux)
  - address must be multiple of 16

How about structures?

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

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**

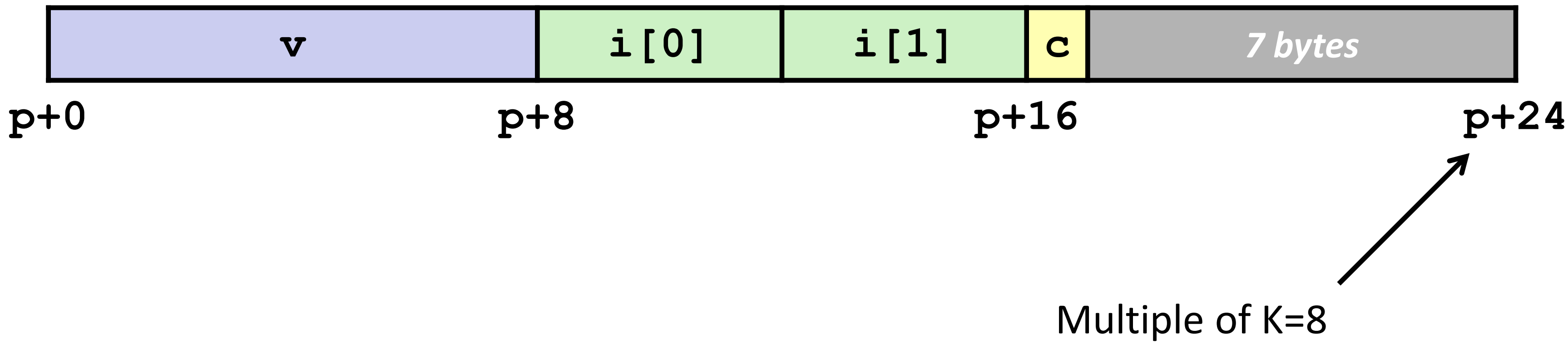
- Example:
  - K = 8, due to **`double`** element



Meeting Overall Alignment Requirement

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

- For largest alignment requirement K
- Overall structure must be multiple of K



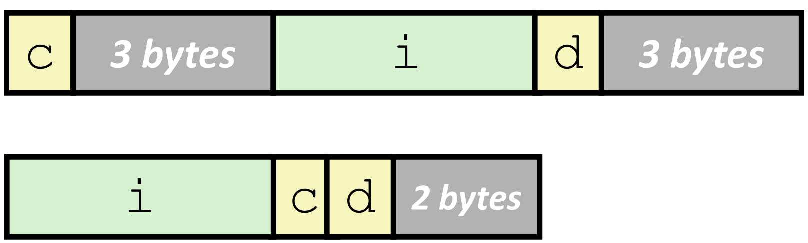
Saving Space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- Effect (K=4)

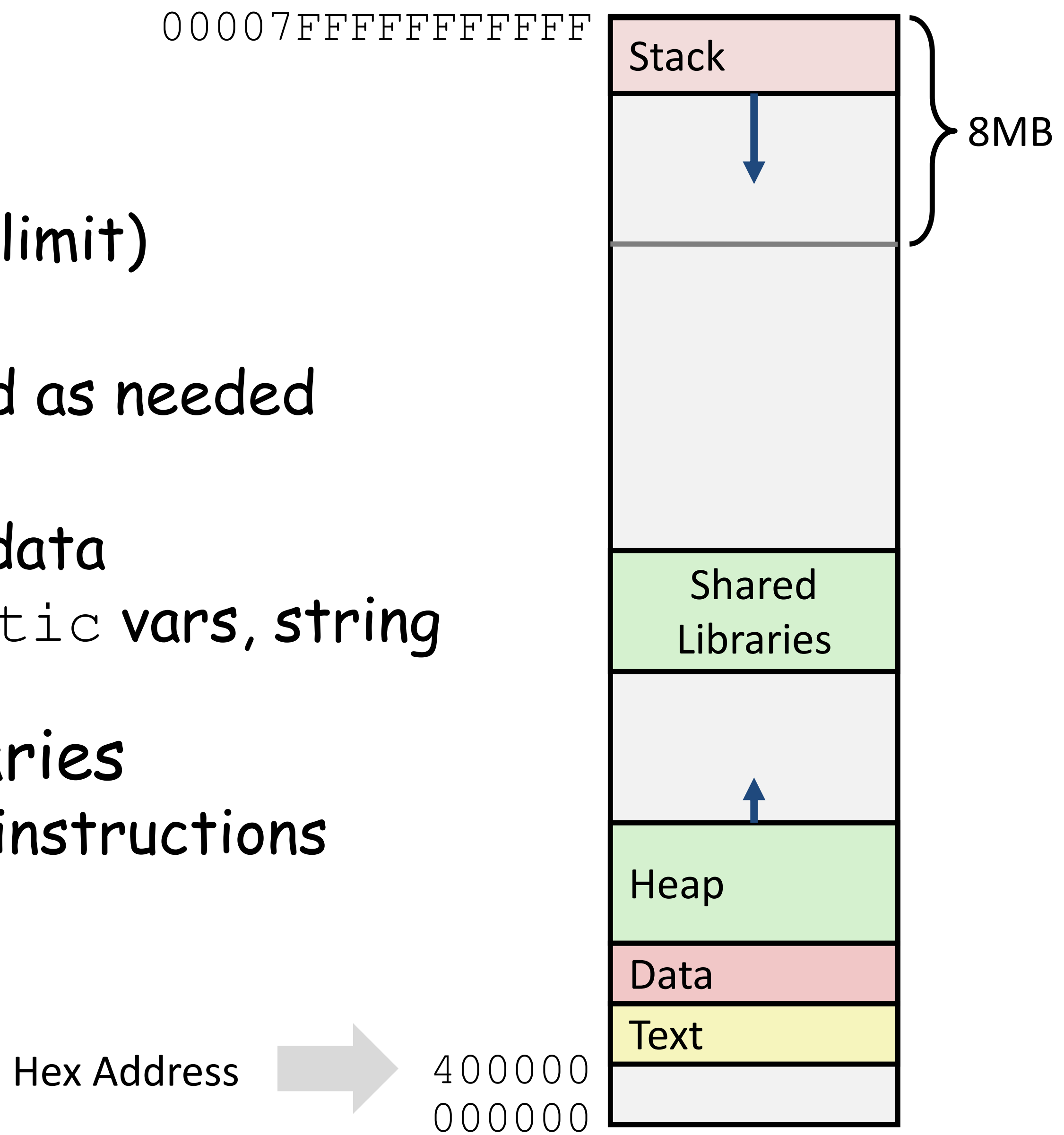


Final Look at Memory Layout

x86-64 Linux Memory Layout

not drawn to scale

- Stack
  - Runtime stack (8MB limit)
- Heap
  - Dynamically allocated as needed
- Data
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only



Conclusions

- We have not covered everything in x86-64, just gave you a glimpse and a feel for it.
- Compiler does more than blind translating your HLL code:
  - It manages the stack.
  - It translates the sophisticated data structure access to assembly
  - It optimizes your code
- No matter how sophisticated your HLL language code, it will be translated to assembly with 16 registers and basic data types!