CS 33

Machine Programming (2)

Jump Instructions

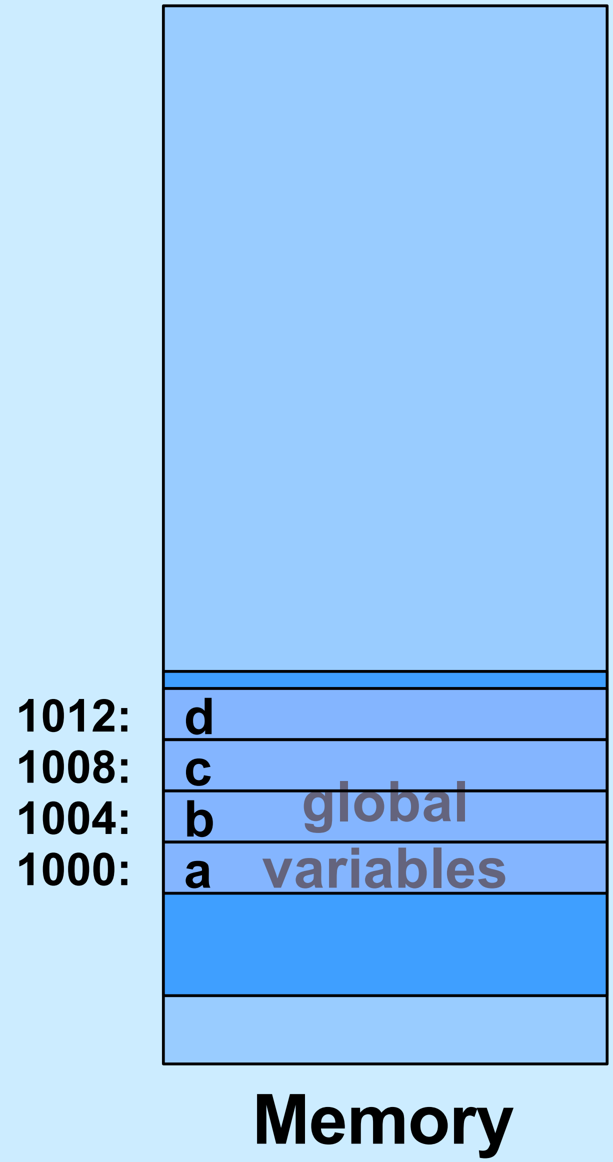
- **Unconditional jump**
  - just do it
- **Conditional jump**
  - to jump or not to jump determined by condition-code flags
  - field in the op code indicates how this is computed
  - in assembler language, simply say
    - » je
      - jump on equal
    - » jne
      - jump on not equal
    - » jg
      - jump on greater than (signed)
    - » etc.

Addresses

```
int a, b, c, d;

int main() {
    a = (b + c) * d;
    ...
}
```

```
mov    b,%acc
add    c,%acc
mul    d,%acc
mov    %acc,a
```

```
mov    1004,%acc
add    1008,%acc
mul    1012,%acc
mov    %acc,1000
```



Addresses

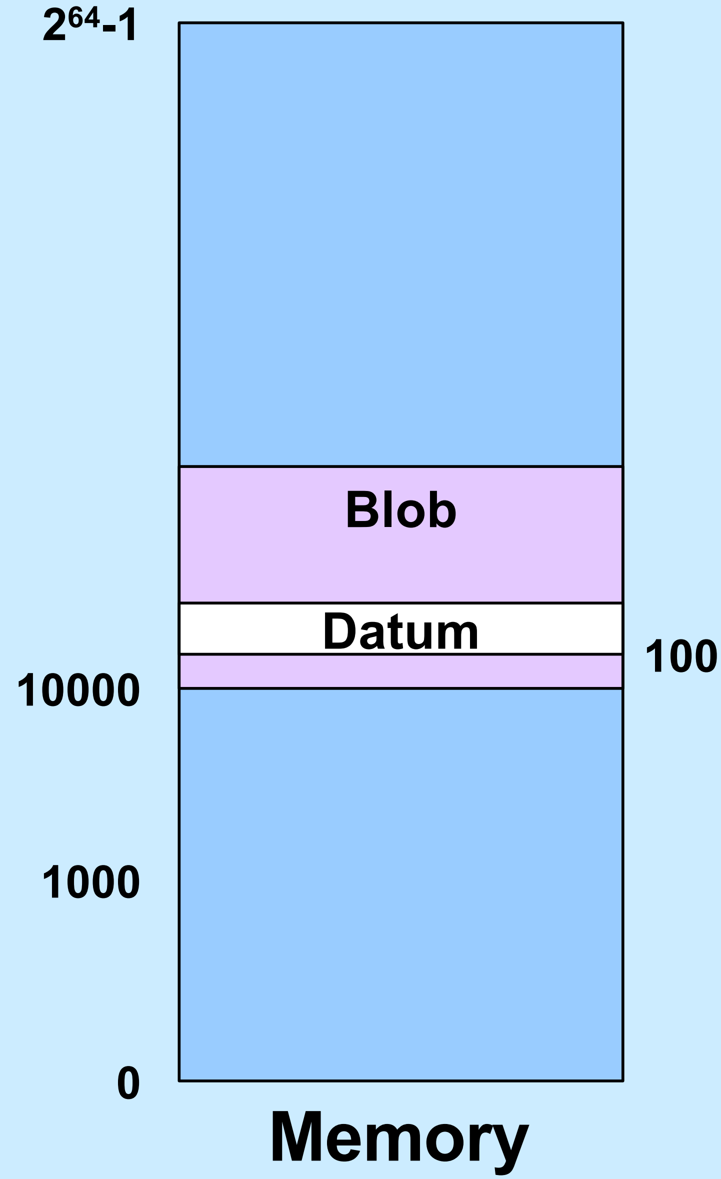
```
int b;

int func(int c, int d) {
    int a;
    a = (b + c) * d;
    ...
}

    mov    ?,%acc
    add    ?,%acc
    mul    ?,%acc
    mov    %acc,?
```

- One copy of *b* for duration of program's execution
  - *b*'s address is the same in each call to *func*
- Different copies of *a*, *c*, and *d* in each call to *func*
  - addresses are different in each call

Relative Addresses

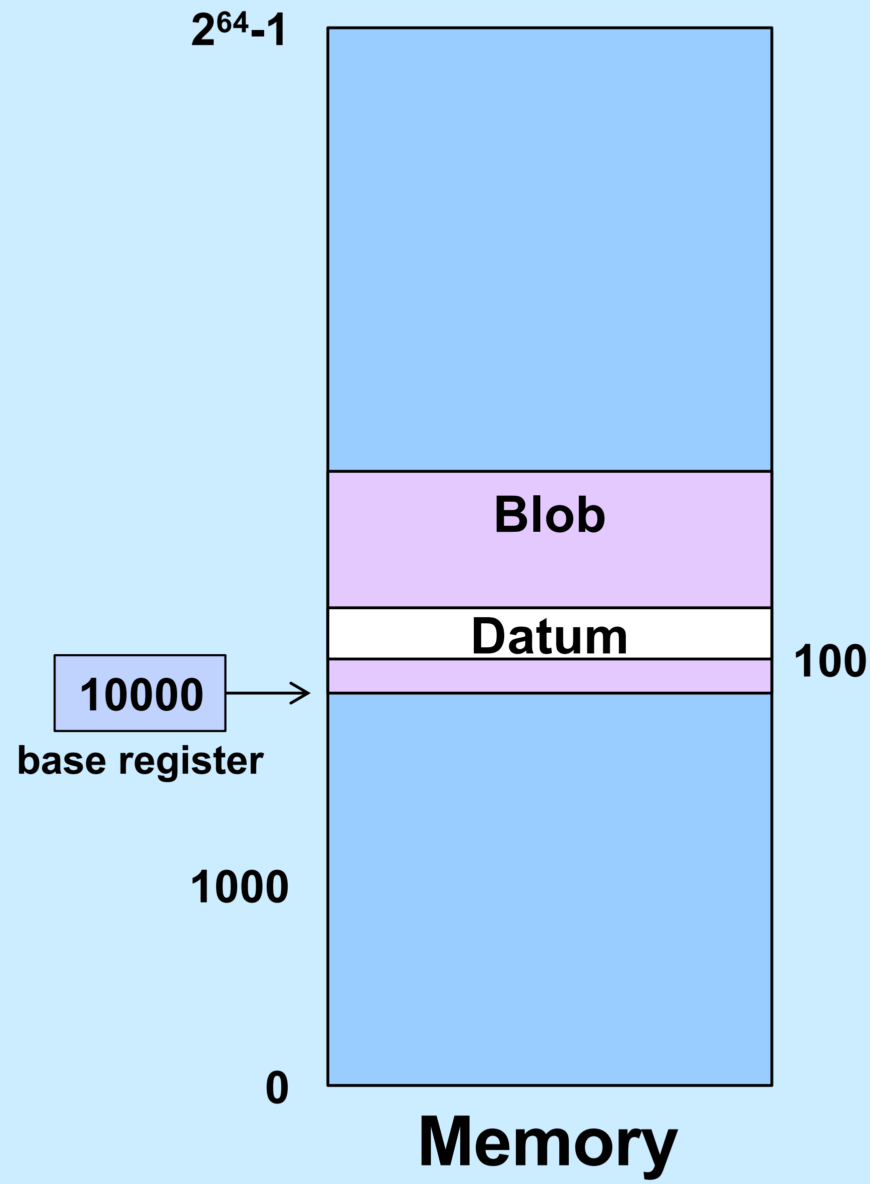
- **Absolute address**
  - actual location in memory
- **Relative address**
  - offset from some other location

- Blob's absolute address is 10000
- Datum's relative address (to Blob) is 100
  - its absolute address is 10100



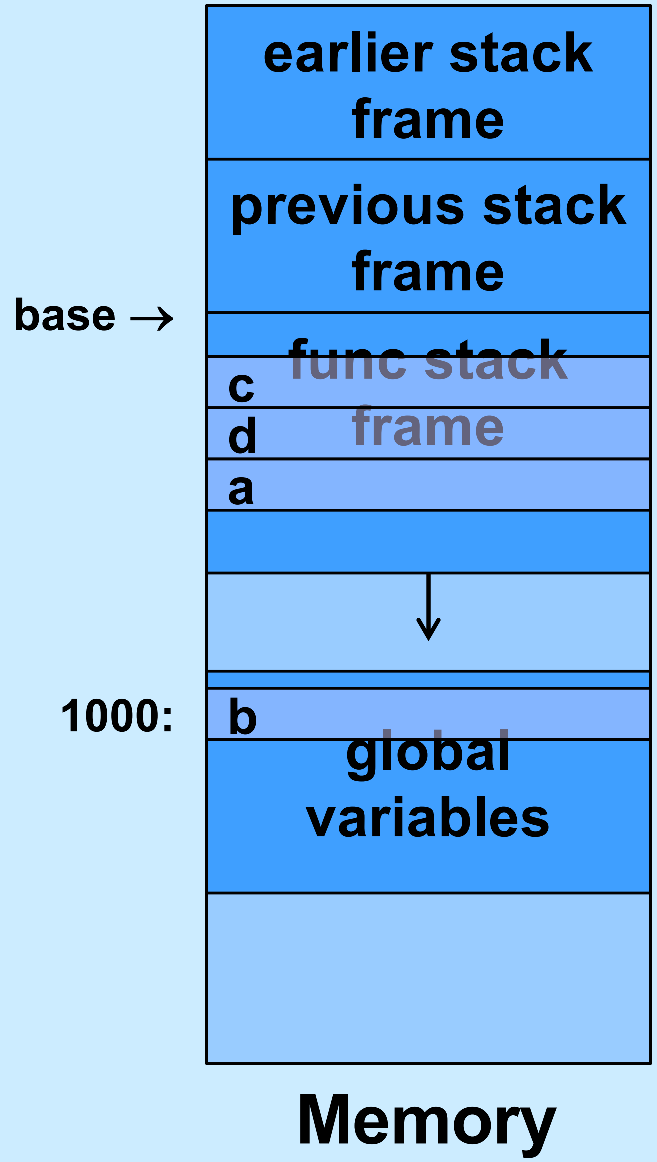
Base Registers

```
mov $10000, %base
mov $10, 100(%base)
```



Addresses

```
long b;

int func(long c, long d) {
    long a;
    a = (b + c) * d;
    ...
}

    mov    1000,%acc
    add    -8(%base),%acc
    mul    -16(%base),%acc
    mov    %acc,-24(%base)
```

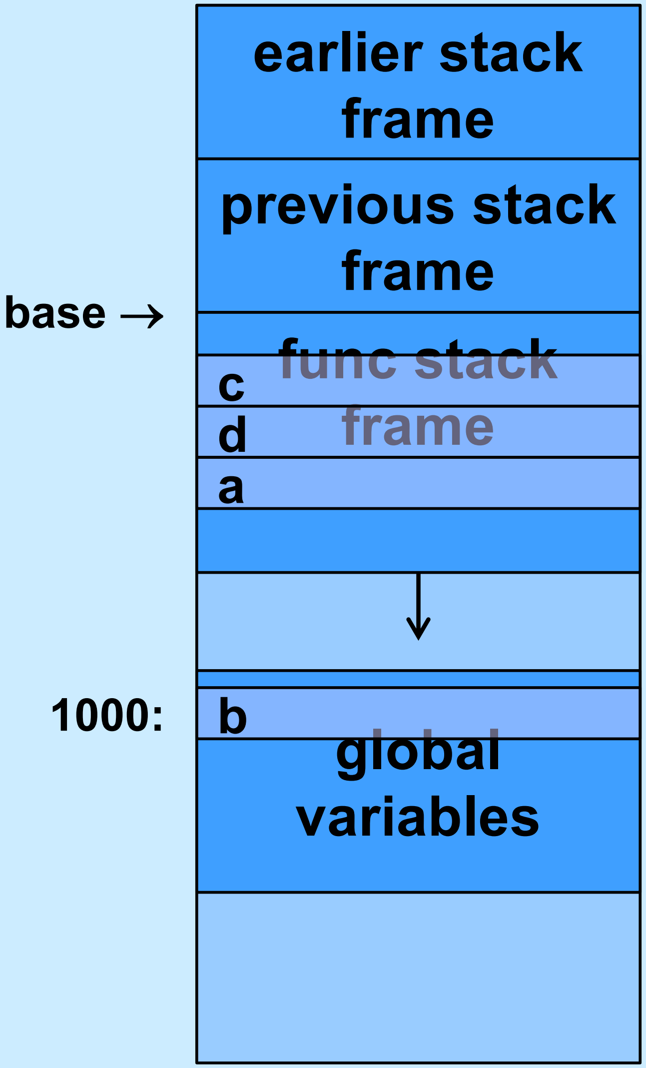


Quiz 1

Suppose the value in *base* is 10,000. What is the address of *c*?

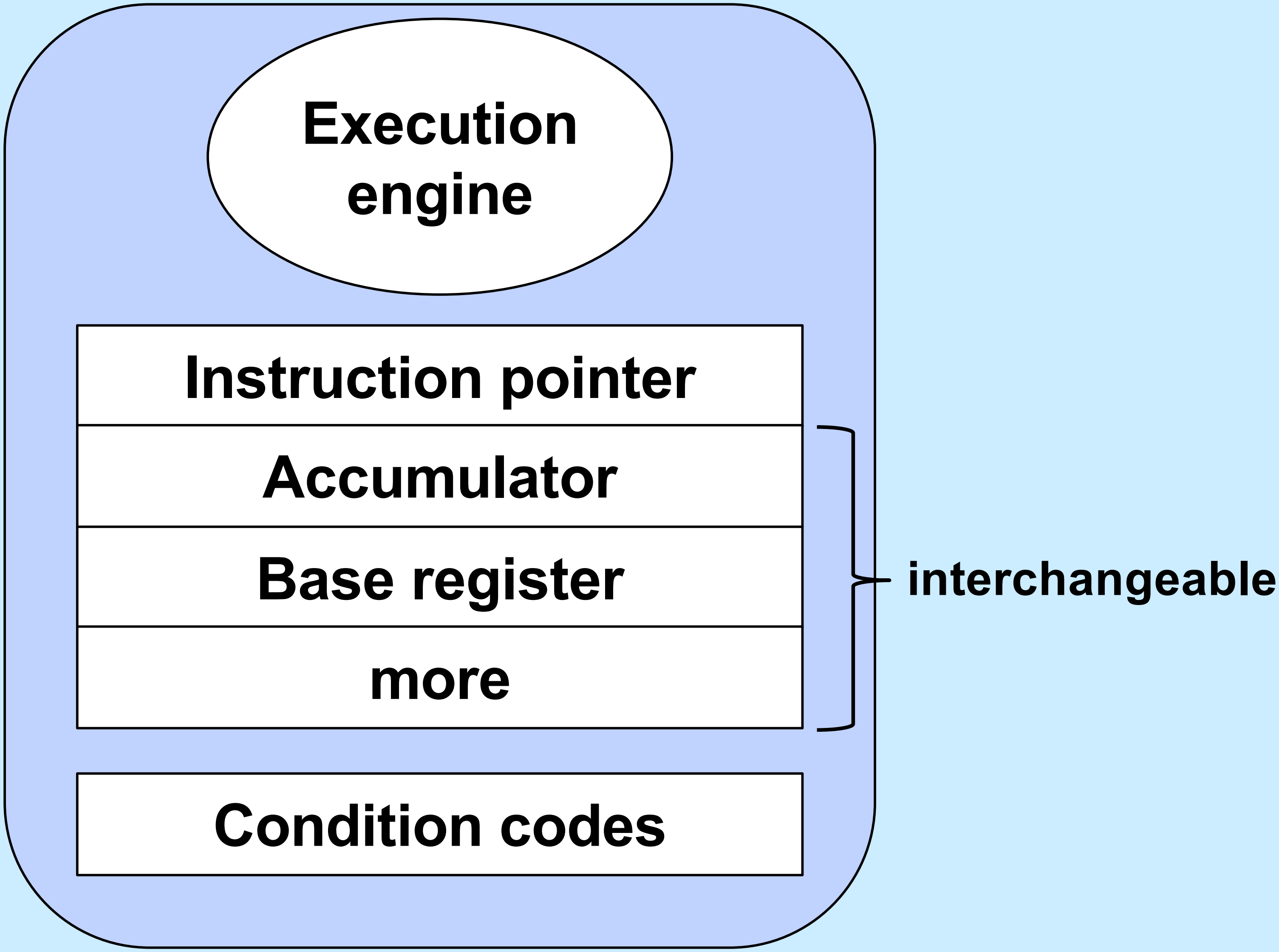
a) 9984
b) 9992
c) 10,008
d) 10,016

```
mov    1000,%acc
add    -8(%base),%acc
mul    -16(%base),%acc
mov    %acc,-24(%base)
```

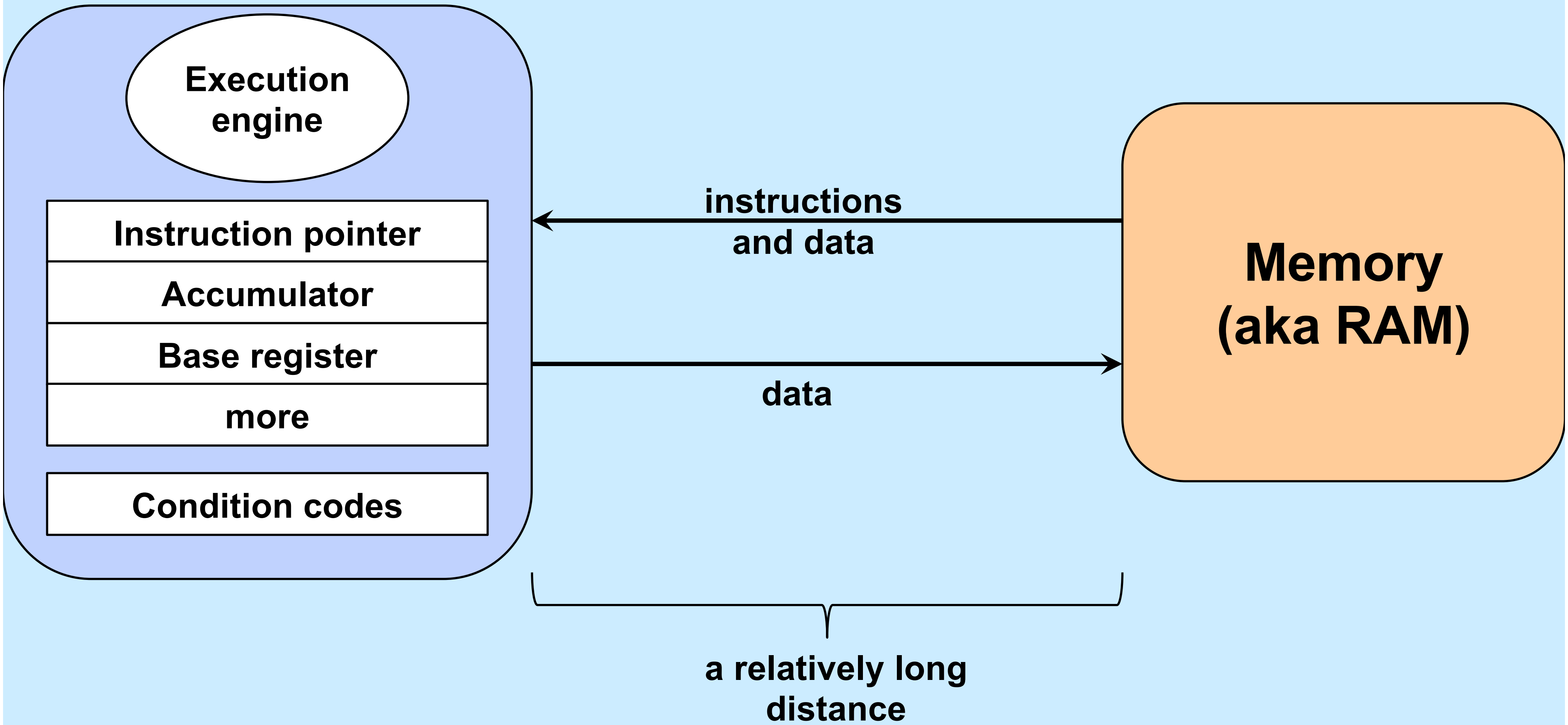


Memory

Registers


Execution engine
Instruction pointer
Accumulator
Base register
more
interchangeable
Condition codes

Registers vs. Memory



Intel x86

- Intel created the 8008 (in 1972)
- 8008 begat 8080
- 8080 begat 8086
- 8086 begat 8088
- 8088 begat 286
- 286 begat 386
- 386 begat 486
- 486 begat Pentium
- Pentium begat Pentium Pro
- Pentium Pro begat Pentium II
- ad infinitum

IA32

2^{64}

- 2^{32} used to be considered a large number
  - one couldn't afford 2^{32} bytes of memory, so no problem with that as an upper bound
- Intel (and others) saw need for machines with 64-bit addresses
  - devised IA64 architecture with HP
    - » became known as Itanium
    - » very different from x86
- AMD also saw such a need
  - developed 64-bit extension to x86, called x86-64
- Itanium flopped
- x86-64 dominated
- Intel, reluctantly, adopted x86-64

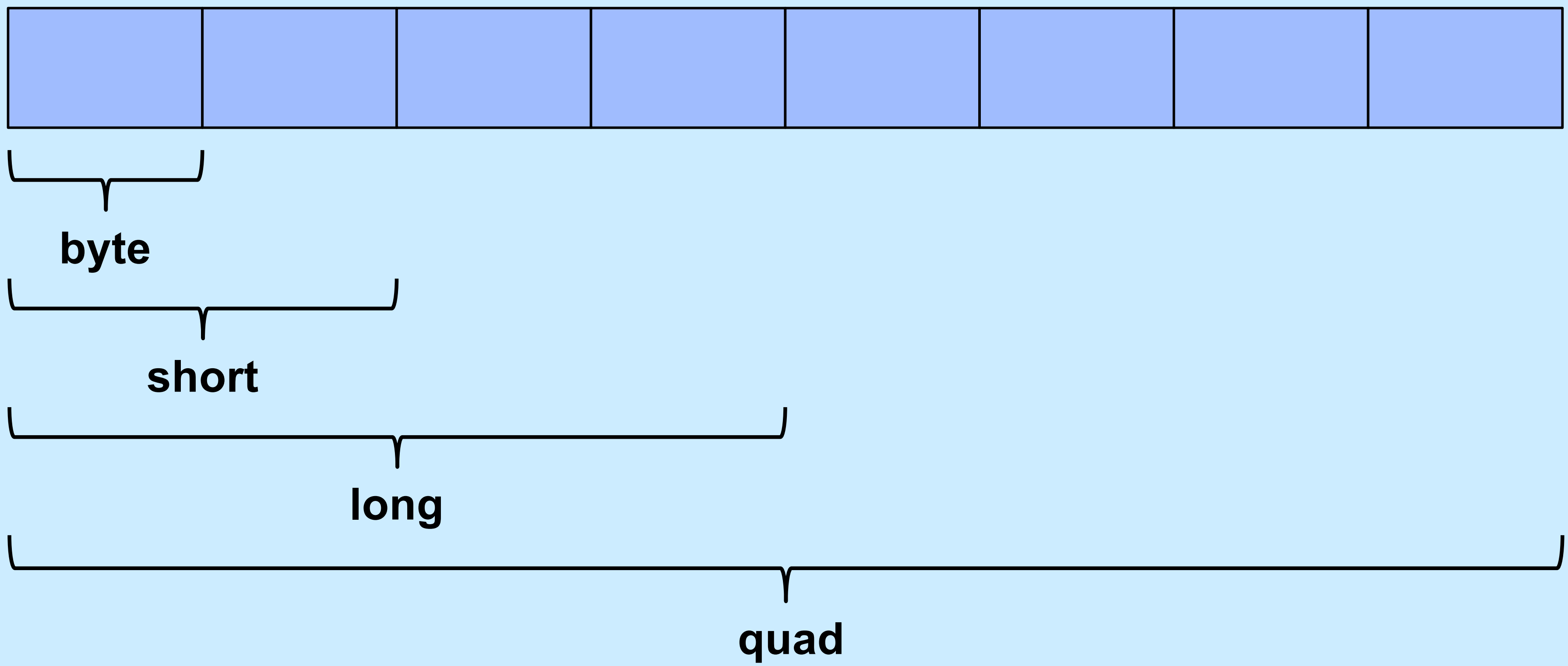
Why Intel?

- Most CS Department machines are Intel
- An increasing number of personal machines are not
  - Apple has switched to ARM
  - packaged into their M1, M2, etc. chips
    - » "Apple Silicon"
- Intel x86-64 is very different from ARM64 — internally
- Programming concepts are similar
- We cover Intel; most of the concepts apply to ARM

Data Types on IA32 and x86-64

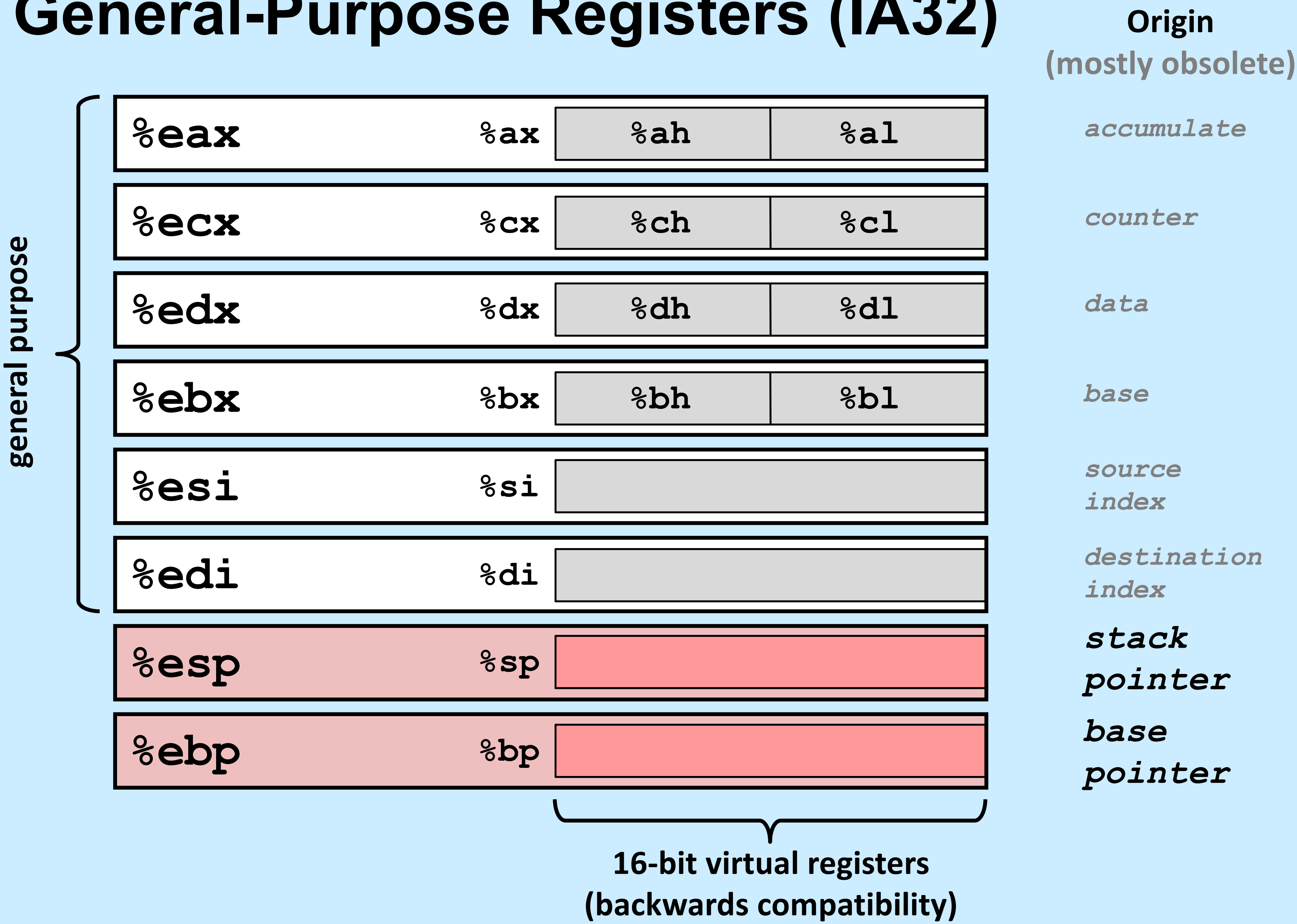
- "Integer" data of 1, 2, or 4 bytes (plus 8 bytes on x86-64)
  - data values
    - » whether signed or unsigned depends on interpretation
  - addresses (untyped pointers)
- Floating-point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - just contiguously allocated bytes in memory

Operand Size



- Rather than `mov` ...
  - `movb`
  - `movs`
  - `movl`
  - `movq` (x86-64 only)

General-Purpose Registers (IA32)



x86-64 General-Purpose Registers

	%rax	%eax	%r8	%r8d	a5
	%rbx	%ebx	%r9	%r9d	a6
a4	%rcx	%ecx	%r10	%r10d	
a3	%rdx	%edx	%r11	%r11d	
a2	%rsi	%esi	%r12	%r12d	
a1	%rdi	%edi	%r13	%r13d	
	%rsp	%esp	%r14	%r14d	
	%rbp	%ebp	%r15	%r15d	

– **Extend existing registers to 64 bits. Add 8 new ones.**

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, 4, or 8 bytes
  - ***Register:*** **one of 16 64-bit registers**
    - » example: `%rax`, `%rdx`
    - » `%rsp` and `%rbp` have some special uses
    - » others have special uses for particular instructions
  - ***Memory:*** **8 consecutive bytes of memory at address given by register(s)**
    - » simplest example: `(%rax)`
    - » various other "address modes"

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

`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 (normally) do memory-memory transfer with a single instruction

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
```

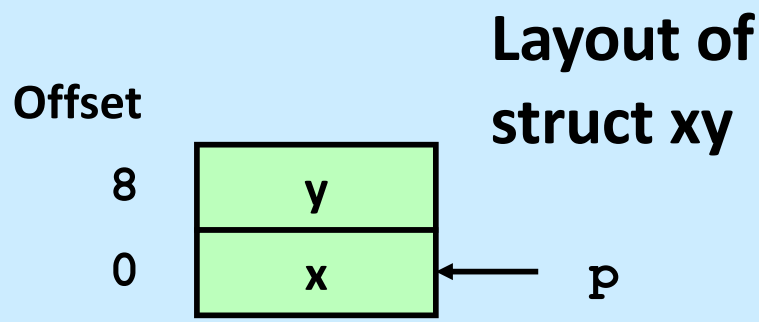
Using Simple Addressing Modes

```
struct xy {
  long x;
  long y;
}
void swapxy(struct xy *p){
  long temp = p->x;
  p->x = p->y;
  p->y = temp;
}
```

```
swap:
  movq (%rdi), %rax
  movq 8(%rdi), %rdx
  movq %rdx, (%rdi)
  movq %rax, 8(%rdi)
  ret
```

Understanding Swapxy

```
struct xy {
  long x;
  long y;
}
void swapxy(struct xy *p){
  long temp = p->x;
  p->x = p->y;
  p->y = temp;
}
```

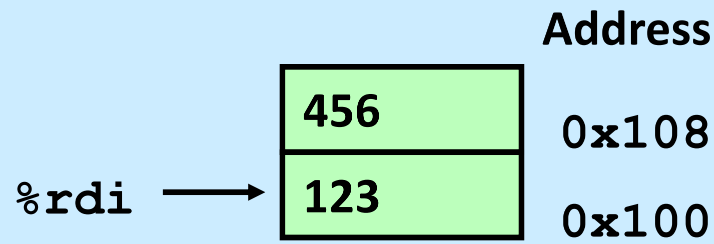


Register	Value
%rdi	p
%rax	temp
%rdx	p->y

```
movq (%rdi), %rax      # temp = p->x
movq 8(%rdi), %rdx     # %rdx = p->y
movq %rdx, (%rdi)      # p->x = %rdx
movq %rax, 8(%rdi)     # p->y = temp
ret
```

Understanding Swapxy

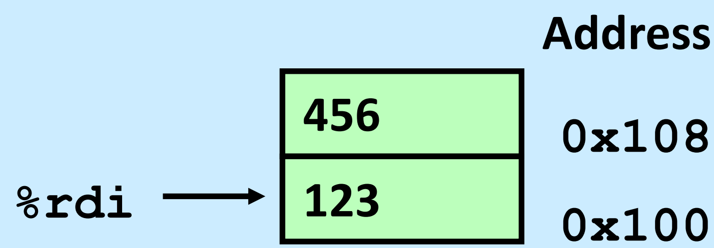
%rdi	0x100
%rax	
%rdx	




```
movq (%rdi), %rax       # temp = p->x
movq 8(%rdi), %rdx      # %rdx = p->y
movq %rdx, (%rdi)       # p->x = %rdx
movq %rax, 8(%rdi)      # p->y = temp
ret
```

Understanding Swapxy

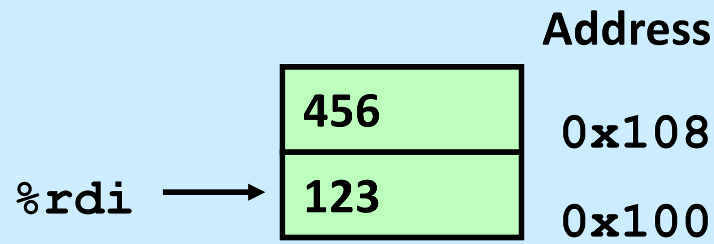
%rdi	0x100
%rax	123
%rdx	




```
movq (%rdi), %rax        # temp = p->x
movq 8(%rdi), %rdx       # %rdx = p->y
movq %rdx, (%rdi)        # p->x = %rdx
movq %rax, 8(%rdi)       # p->y = temp
ret
```

Understanding Swapxy

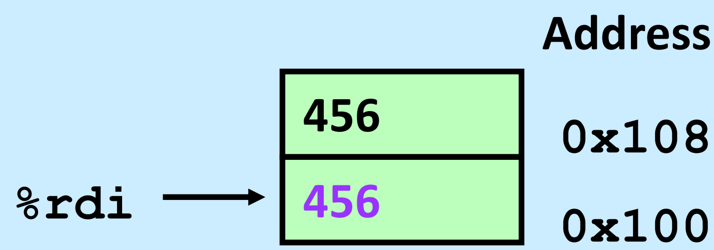
%rdi	0x100
%rax	123
%rdx	456




```
movq (%rdi), %rax      # temp = p->x
movq 8(%rdi), %rdx     # %rdx = p->y
movq %rdx, (%rdi)      # p->x = %rdx
movq %rax, 8(%rdi)     # p->y = temp
ret
```

Understanding Swapxy

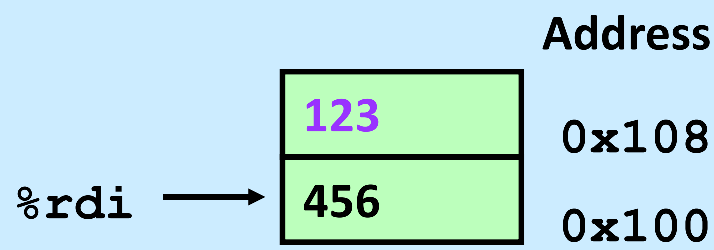
%rdi	0x100
%rax	123
%rdx	456




```
movq (%rdi), %rax       # temp = p->x
movq 8(%rdi), %rdx      # %rdx = p->y
movq %rdx, (%rdi)       # p->x = %rdx
movq %rax, 8(%rdi)      # p->y = temp
ret
```

Understanding Swapxy

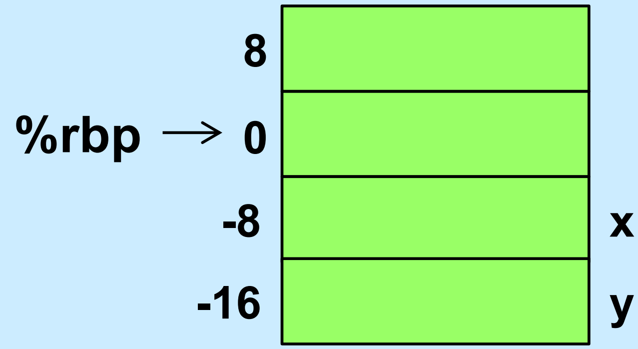
%rdi	0x100
%rax	123
%rdx	456




```
movq (%rdi), %rax       # temp = p->x
movq 8(%rdi), %rdx      # %rdx = p->y
movq %rdx, (%rdi)       # p->x = %rdx
movq %rax, 8(%rdi)      # p->y = temp
ret
```

Quiz 2

```
movq -8(%rbp), %rax
movq (%rax), %rax
movq (%rax), %rax
movq %rax, -16(%rbp)
```



Which C statements best describe the assembler code?

```
// a
long ***x;
long y;
y = ***x;
```

```
// b
long **x;
long y;
y = **x;
```

```
// c
long *x;
long y;
y = *x;
```

```
// d
long x;
long y;
y = x;
```

Complete Memory-Addressing Modes

- **Most general form**

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

  – D: constant "displacement"
  – Rb: base register: any of 16† registers
  – Ri: index register: any, except for `%rsp`
  – S: scale: 1, 2, 4, or 8

- **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]]
D	Mem[D]

†The instruction pointer may also be used (for a total of 17 registers)

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*0x0100	0xf400
0x80(,%rdx, 2)	2*0xf000 + 0x80	0x1e080

Address-Computation Instruction

- `leaq` src, dest
  - src is address mode expression
  - set *dest* 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 mul12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

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

32-bit Operands on x86-64

- **addl 4(%rdx), %eax**
  - memory address must be 64 bits
  - operands (in this case) are 32-bit
    - » result goes into %eax
      - lower half of %rax
      - upper half is filled with zeroes

Quiz 3

What value ends up in %ecx?

```
movq $1000,%rax
movq $1,%rbx
movl 2(%rax,%rbx,2),%ecx
```

a) **0x04050607**

b) **0x07060504**

c) **0x06070809**

d) **0x09080706**

	1009:	0x09
	1008:	0x08
	1007:	0x07
	1006:	0x06
	1005:	0x05
	1004:	0x04
	1003:	0x03
	1002:	0x02
	1001:	0x01
%rax →	1000:	0x00

Hint: