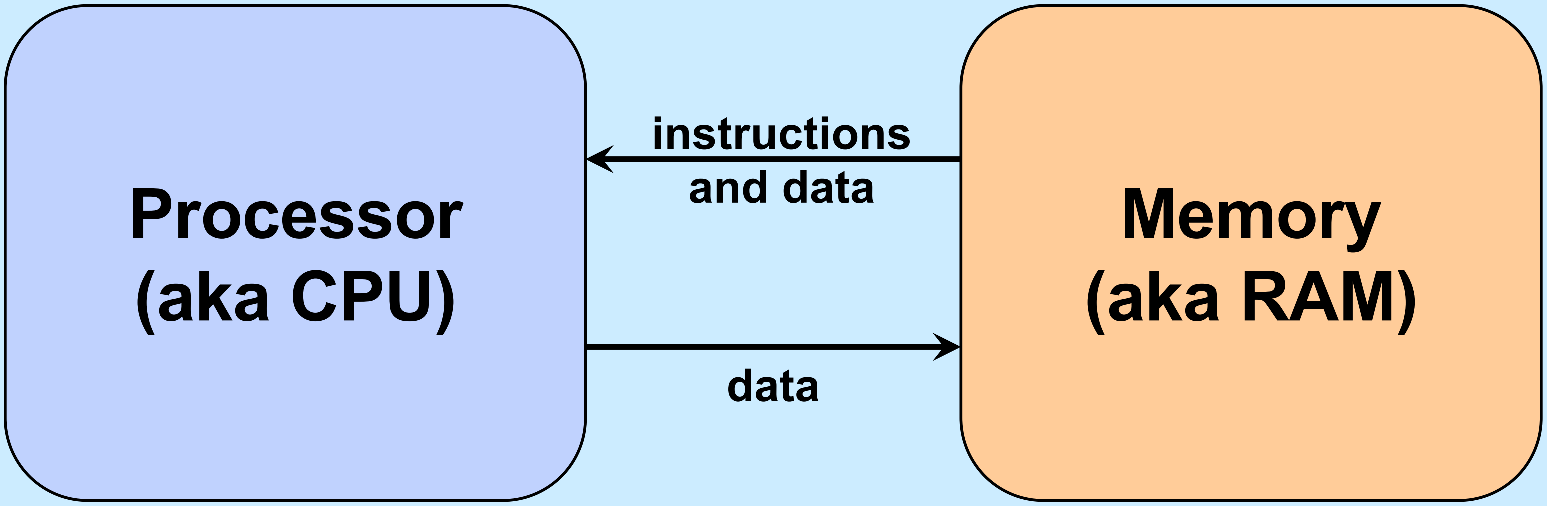
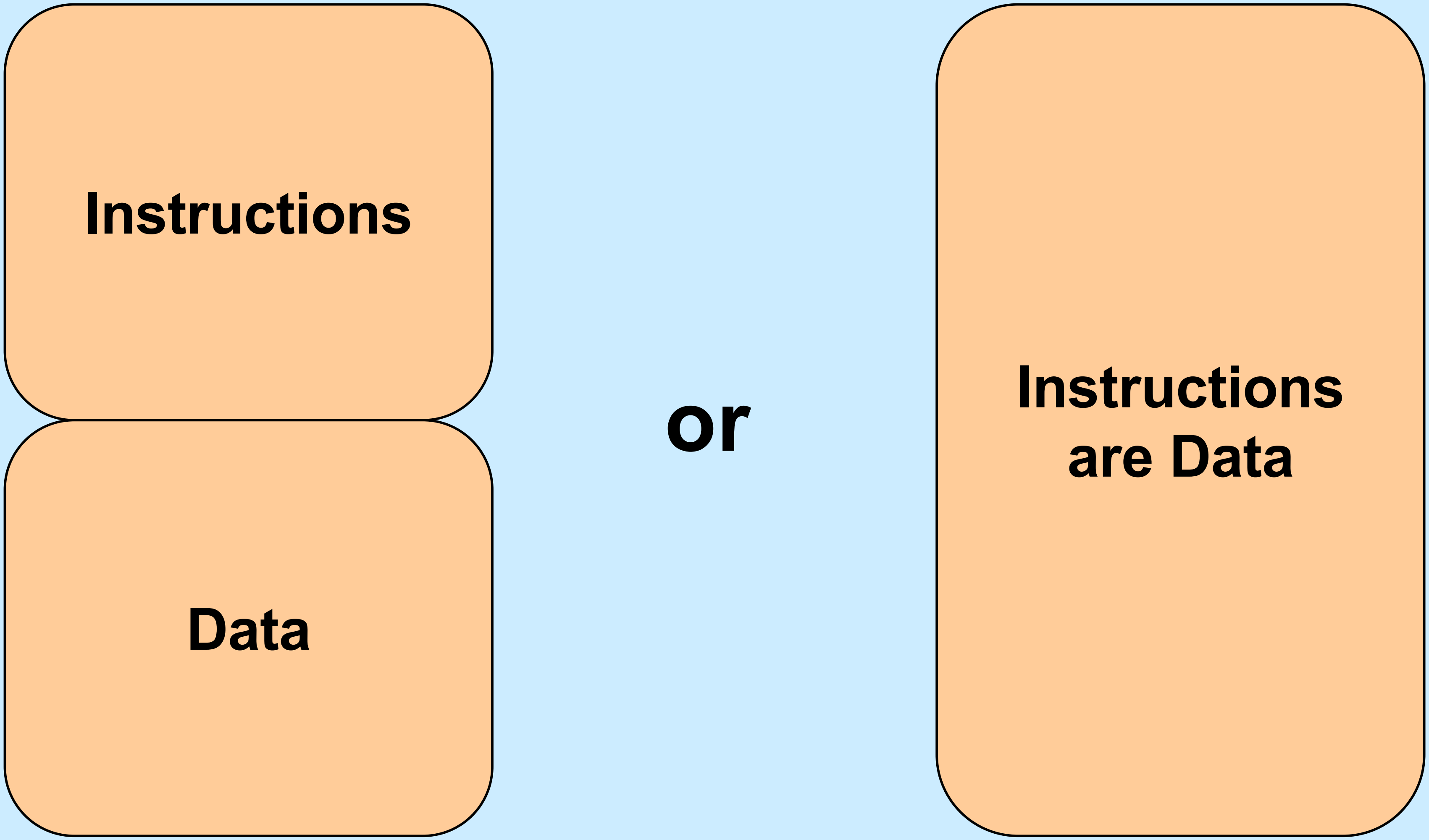
CS 33

Intro to Machine Programming

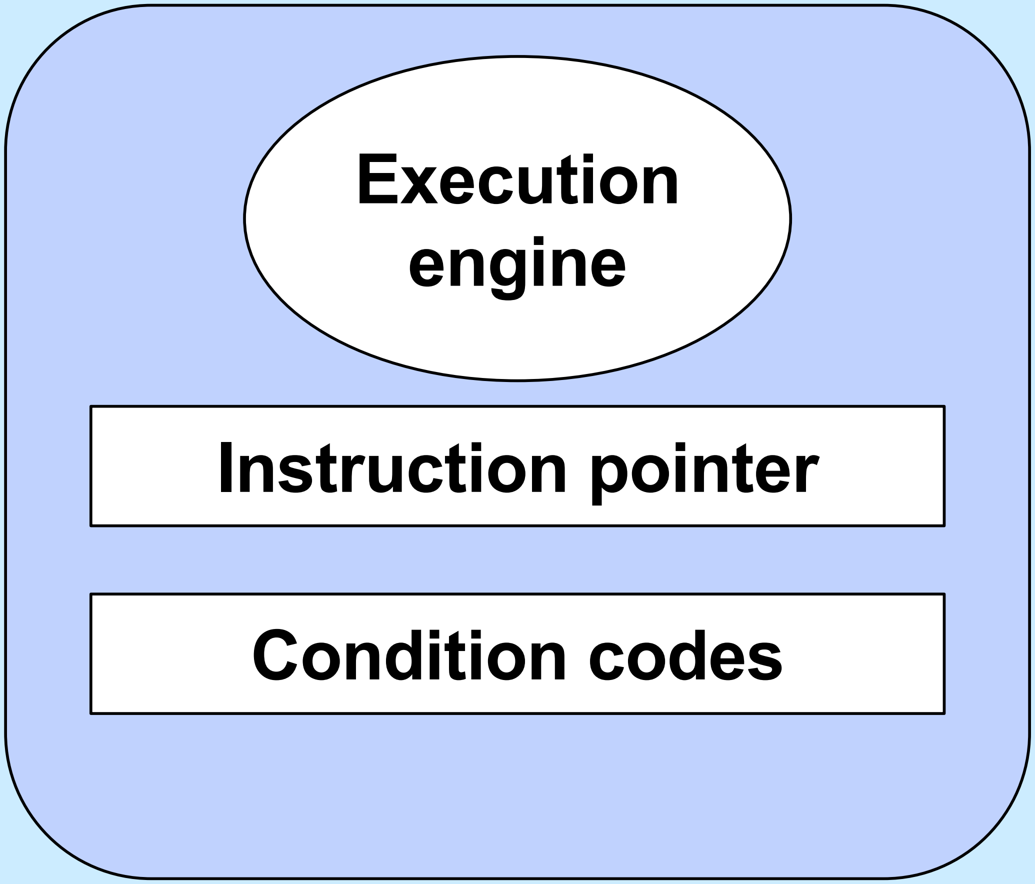
Machine Model



Memory



Processor: Some Details



Processor: Basic Operation

```
while (forever) {
  fetch instruction IP points at
  decode instruction
  fetch operands
  execute
  store results
  update IP and condition code
}
```

Instructions ...

Op code	Operand1	Operand2	...

Operands

- Form
  - immediate vs. reference
    - » value vs. address
- How many?
  - 3
    - » add a,b,c
      - c = a + b
  - 2
    - » add a,b
      - b += a

Operands (continued)

- **Accumulator**
  - special memory in the processor
    - » known as a *register*
    - » fast access
  - allows single-operand instructions
    - » add a
      - acc += a
    - » add b
      - acc += b

From C to Assembler ...

```
a = (b + c) * d;

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

```
if (a<b)
   c = 1;
else
   d = 1;

    cmp    a,b
    jge    .L1
    mov    $1,c
    jmp    .L2
.L1
    mov    $1,d
.L2
```

immediate operand

immediate operand

Condition Codes

- **Set of flags giving status of most recent operation:**
  - zero flag
    - » result was zero
  - sign flag
    - » for signed arithmetic interpretation: sign bit is set
  - overflow flag
    - » for signed arithmetic interpretation
  - carry flag (generated by carry or borrow out of most-significant bit)
    - » for unsigned arithmetic interpretation
- **Set implicitly by arithmetic instructions**
- **Set explicitly by compare instruction**
  - cmp a,b
    - » sets flags based on result of b-a

Examples (1)

- **Assume 32-bit arithmetic**

- **x is `0x80000000`**
  - **TMIN if interpreted as two's-complement**
  - **2^{31} if interpreted as unsigned**
- **x-1 (`0x7fffffff`)**
  - **TMAX if interpreted as two's-complement**
  - **2^{31}-1 if interpreted as unsigned**
  - **zero flag is not set**
  - **sign flag is not set**
  - **overflow flag is set**
  - **carry flag is not set**

Examples (2)

- **x is `0xffffffff`**
  - **-1 if interpreted as two's-complement**
  - **UMAX (2^{32}-1) if interpreted as unsigned**
- **x+1 (`0x00000000`)**
  - **zero under either interpretation**
  - **zero flag is set**
  - **sign flag is not set**
  - **overflow flag is not set**
  - **carry flag is set**

Examples (3)

- **x is `0xffffffff`**
  - -1 if interpreted as two's-complement
  - UMAX (2^{32}-1) if interpreted as unsigned
- **x+2 (`0x00000001`)**
  - (+)1 under either interpretation
  - zero flag is not set
  - sign flag is not set
  - overflow flag is not set
  - carry flag is set

Quiz 1

- Set of flags giving status of most recent operation:
  - zero flag
    - » result was zero
  - sign flag
    - » for signed arithmetic interpretation: sign bit is set
  - overflow flag
    - » for signed arithmetic interpretation
  - carry flag (generated by carry or borrow out of most-significant bit)
    - » for unsigned arithmetic interpretation
- Set explicitly by compare instruction
  - cmp a,b
    - » sets flags based on result of b-a

Which flags are set to one by “cmp 2,1”?

a) overflow flag only
b) carry flag only
c) sign and carry flags only
d) sign and overflow flags only
e) sign, overflow, and carry flags

Jump Instructions

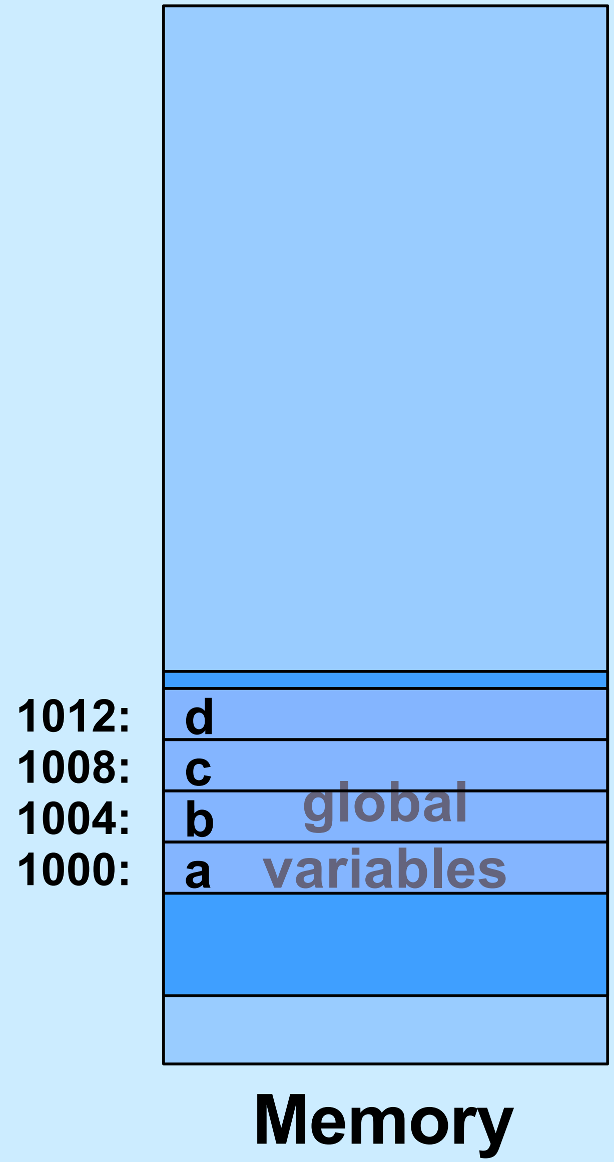
- **Unconditional jump**
  - just do it
- **Conditional jump**
  - to jump or not to jump determined by condition-code flags
  - field in the op code indicates how this is computed
  - in assembler language, simply say
    - » je
      - jump on equal
    - » jne
      - jump on not equal
    - » jg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

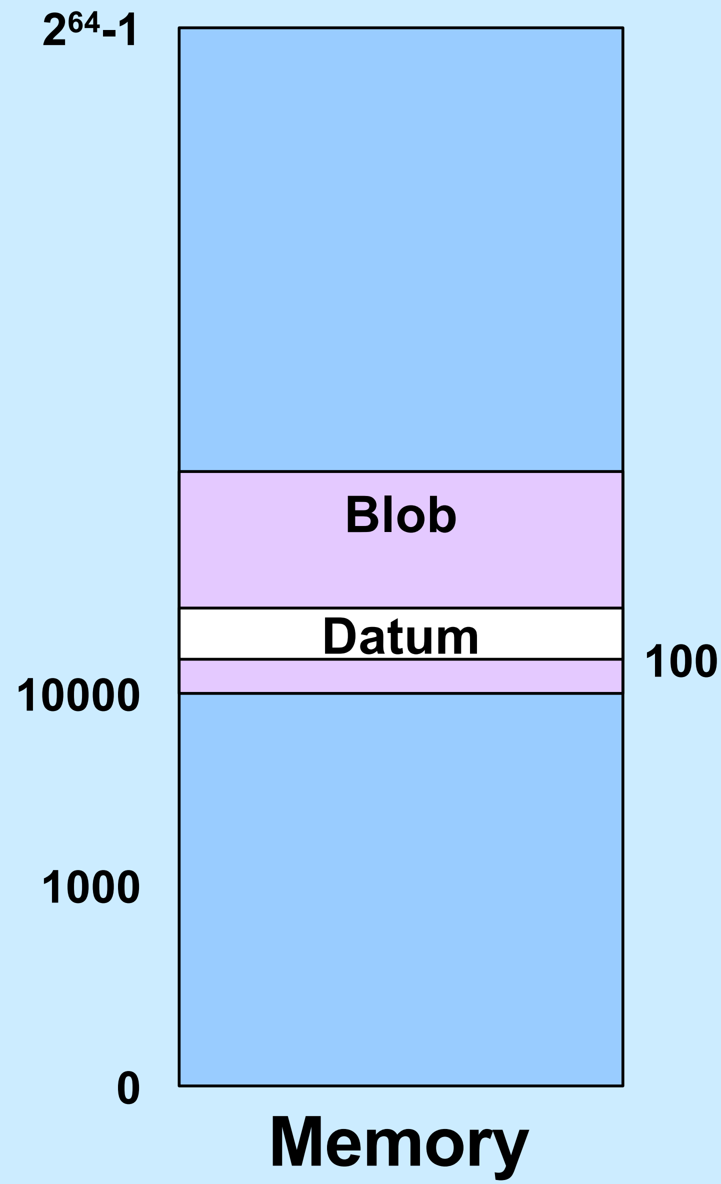
```
int b;

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 for each call to *func*
- Different copies of *a*, *c*, and *d* for each call to *func*
  - addresses are different in each call

Relative Addresses

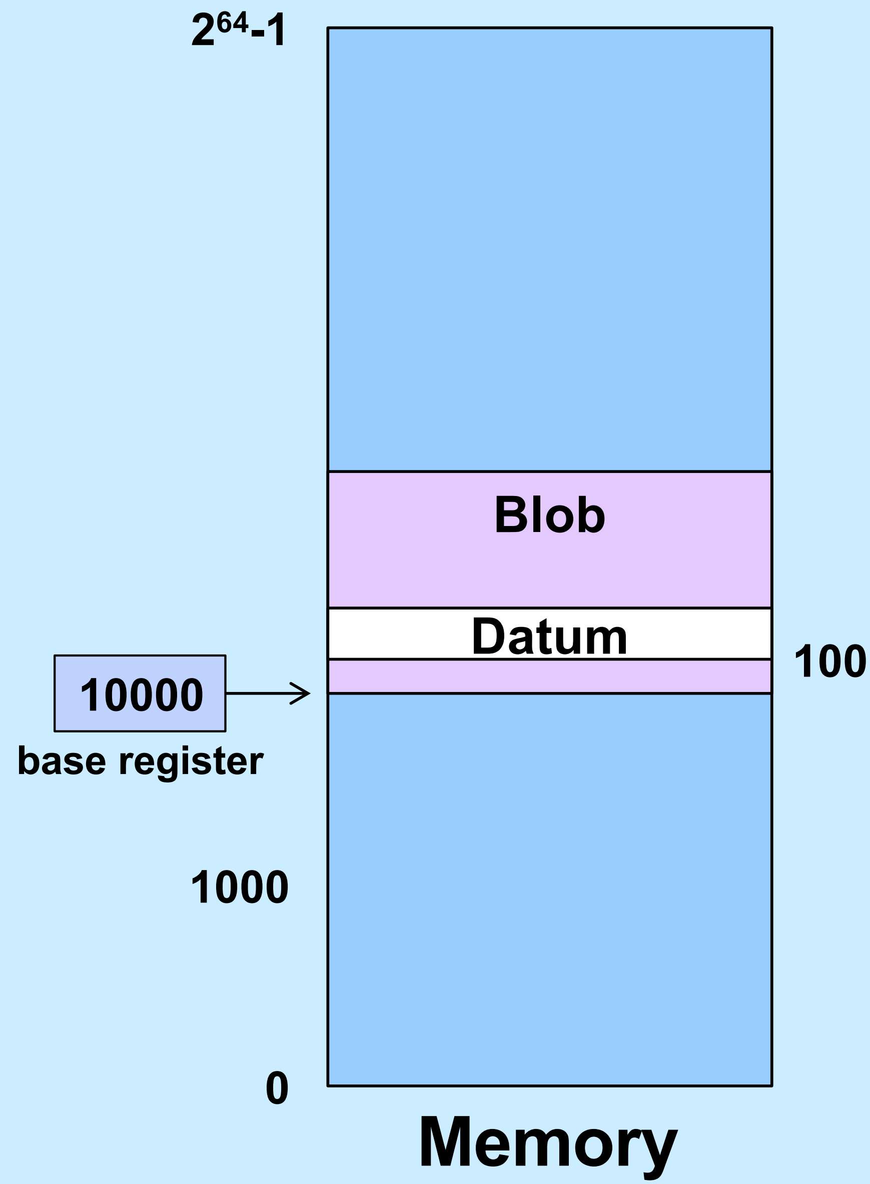
- **Absolute address**
  - actual location in memory
- **Relative address**
  - offset from some other location

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



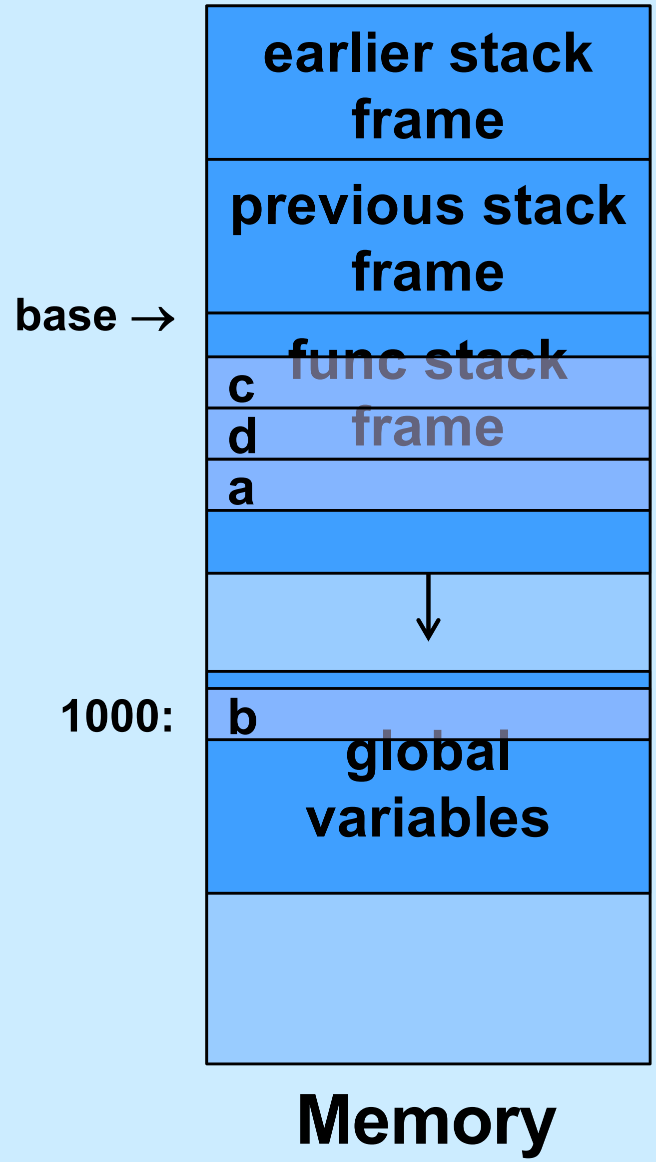
Base Registers

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



Addresses

```
int b;

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

    mov     1000,%acc
    add     -8(%base),%acc
    mul     -12(%base),%acc
    mov     %acc,-16(%base)
```

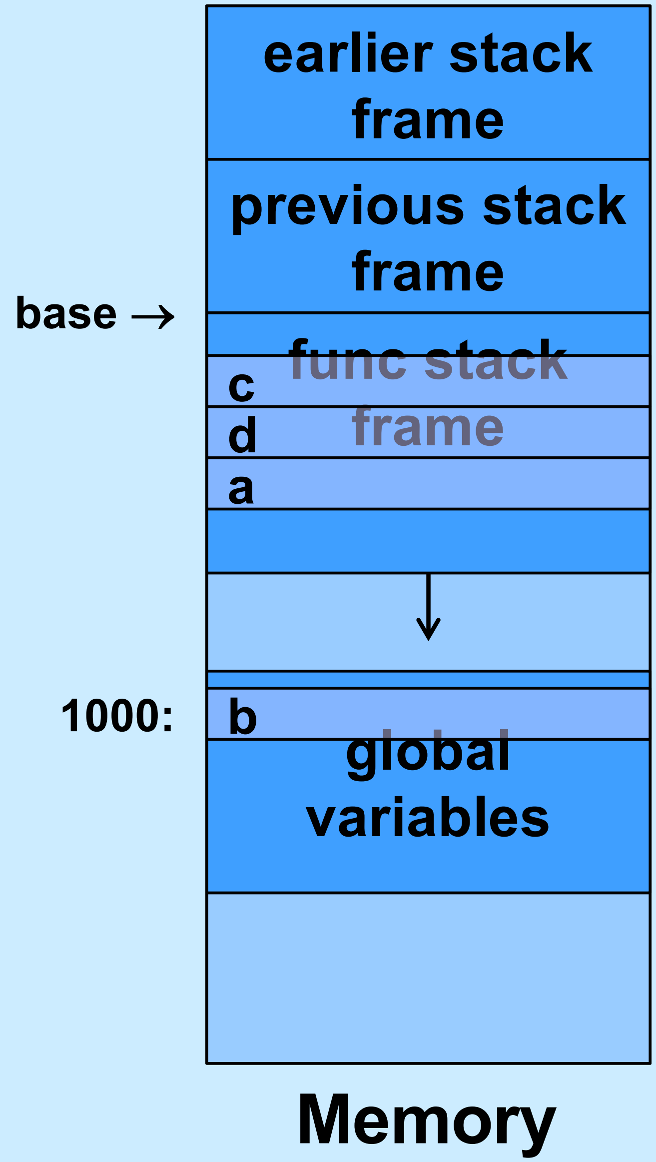


Quiz 2

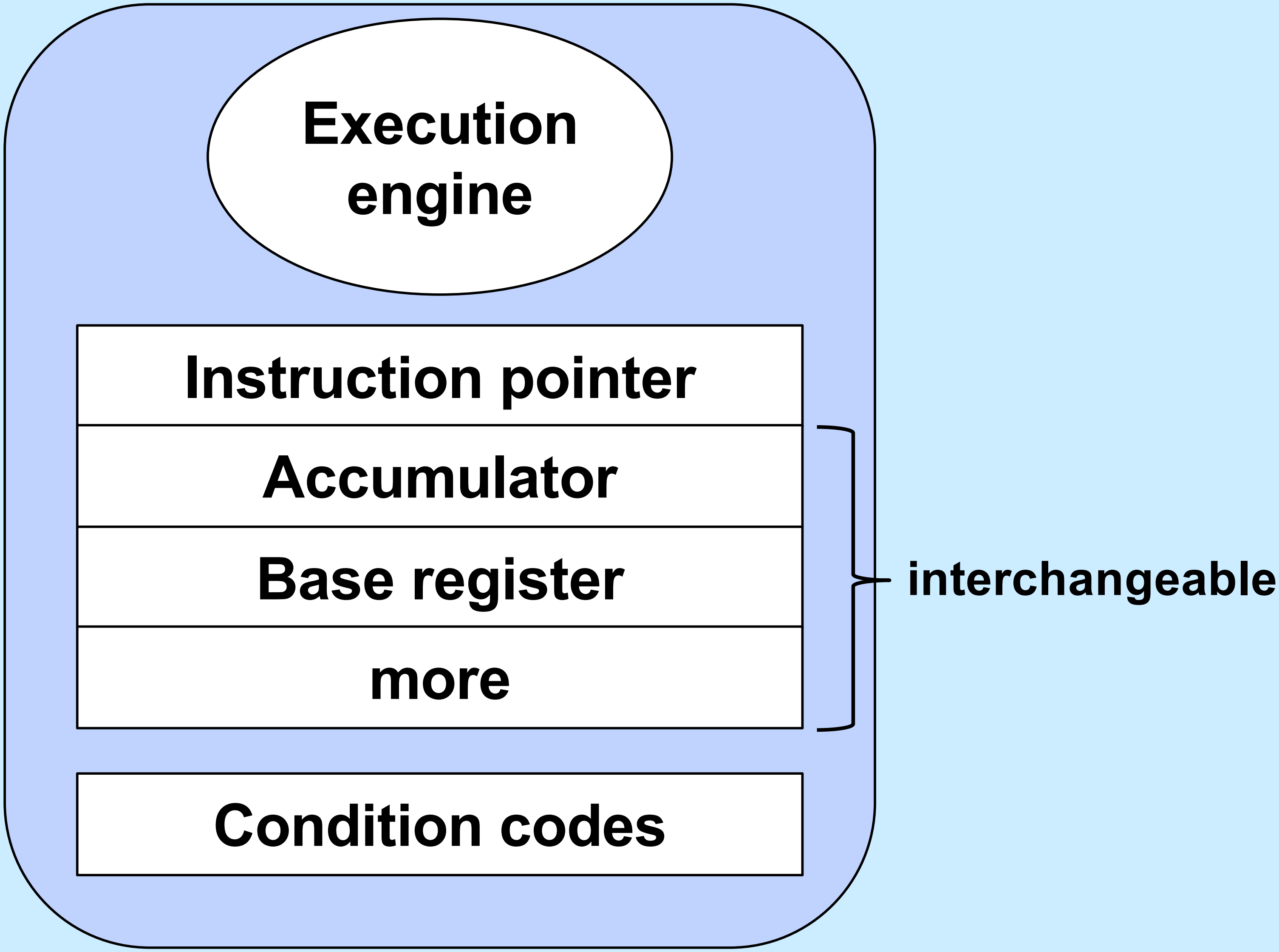
Suppose the value in *base* is 10,000. What is the address of *c*?

a) 10,008
b) 10,004
c) 9996
d) 9992

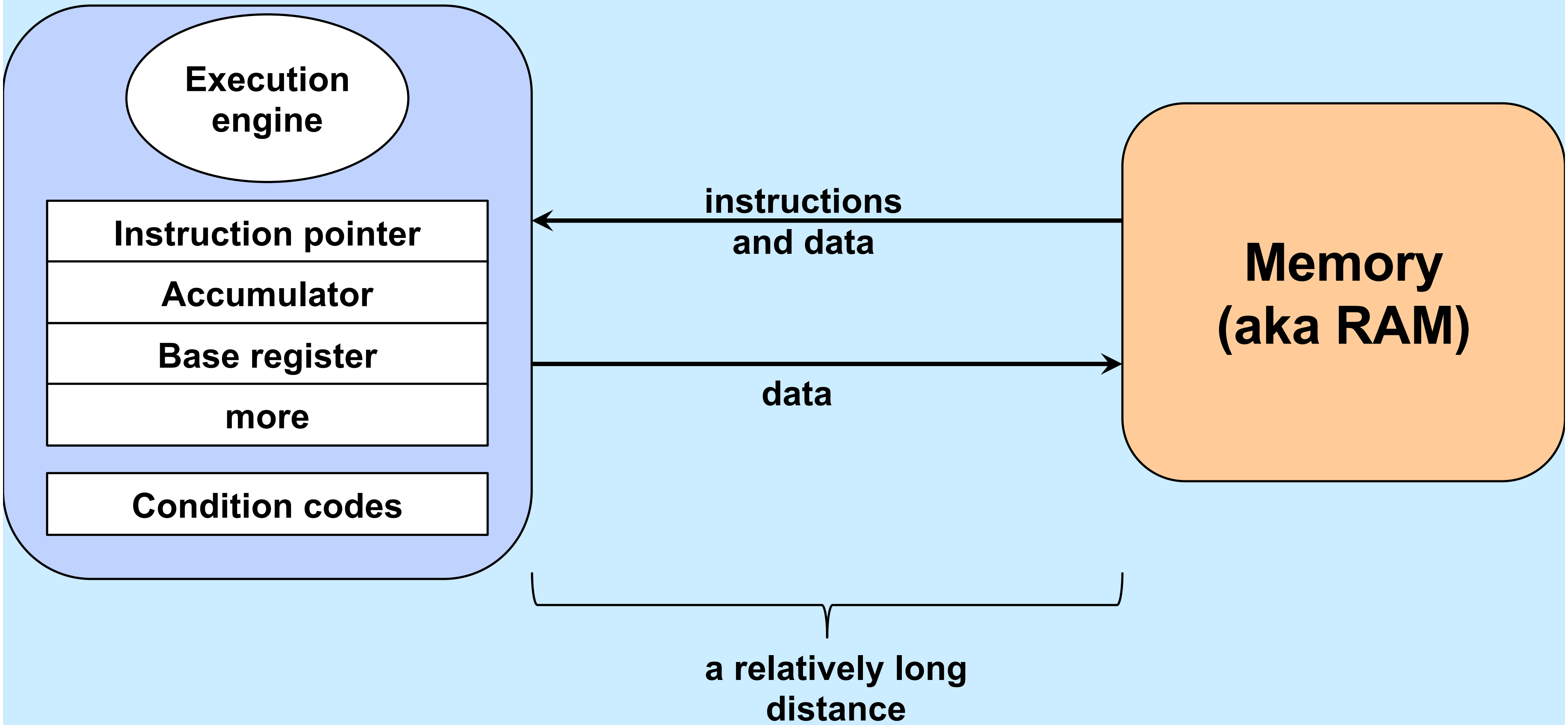
```
mov    1000,%acc
add    -8(%base),%acc
mul    -12(%base),%acc
mov    %acc,-16(%base)
```



Registers


Execution engine
Instruction pointer
Accumulator
Base register
more
Condition codes
interchangeable

Registers vs. Memory



Intel x86

- Intel created the 8008 (in 1972)
- 8008 begat 8080
- 8080 begat 8086
- 8086 begat 8088
- 8086 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

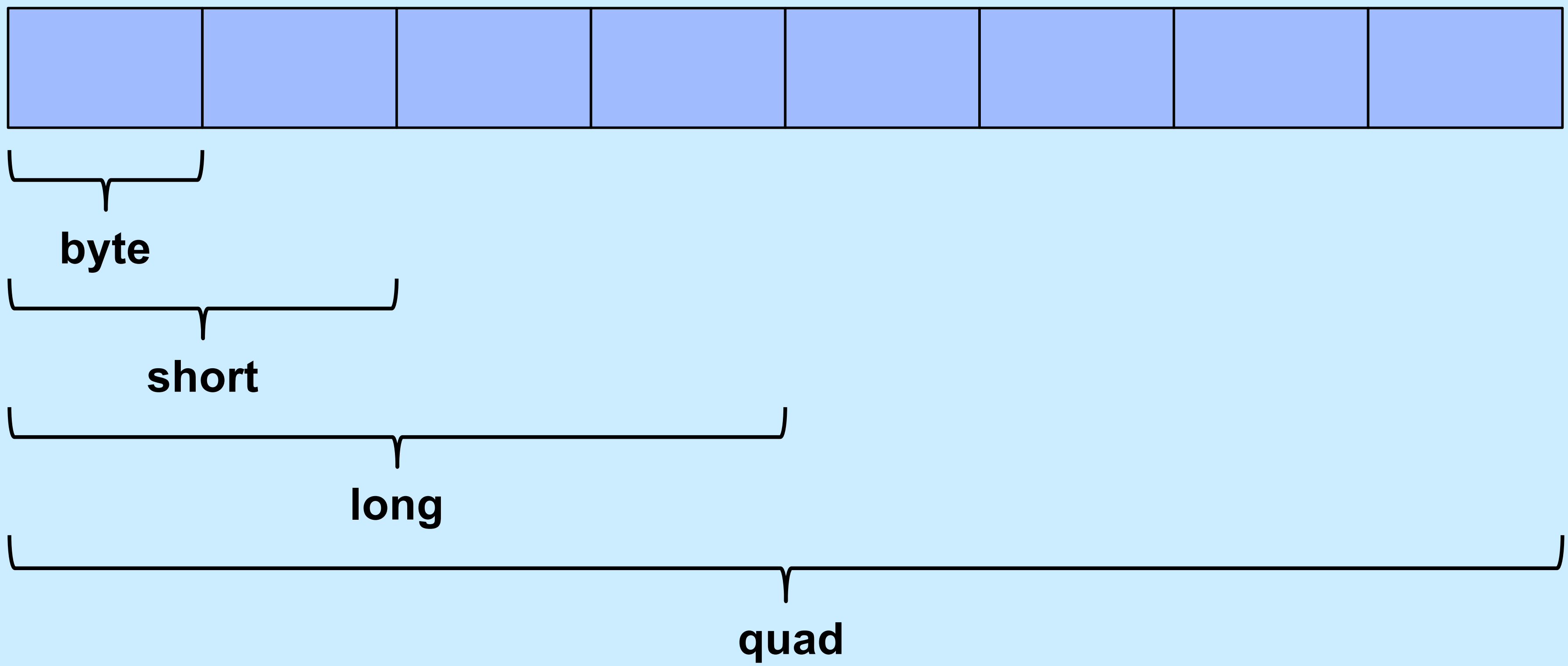
Why Intel?

- 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

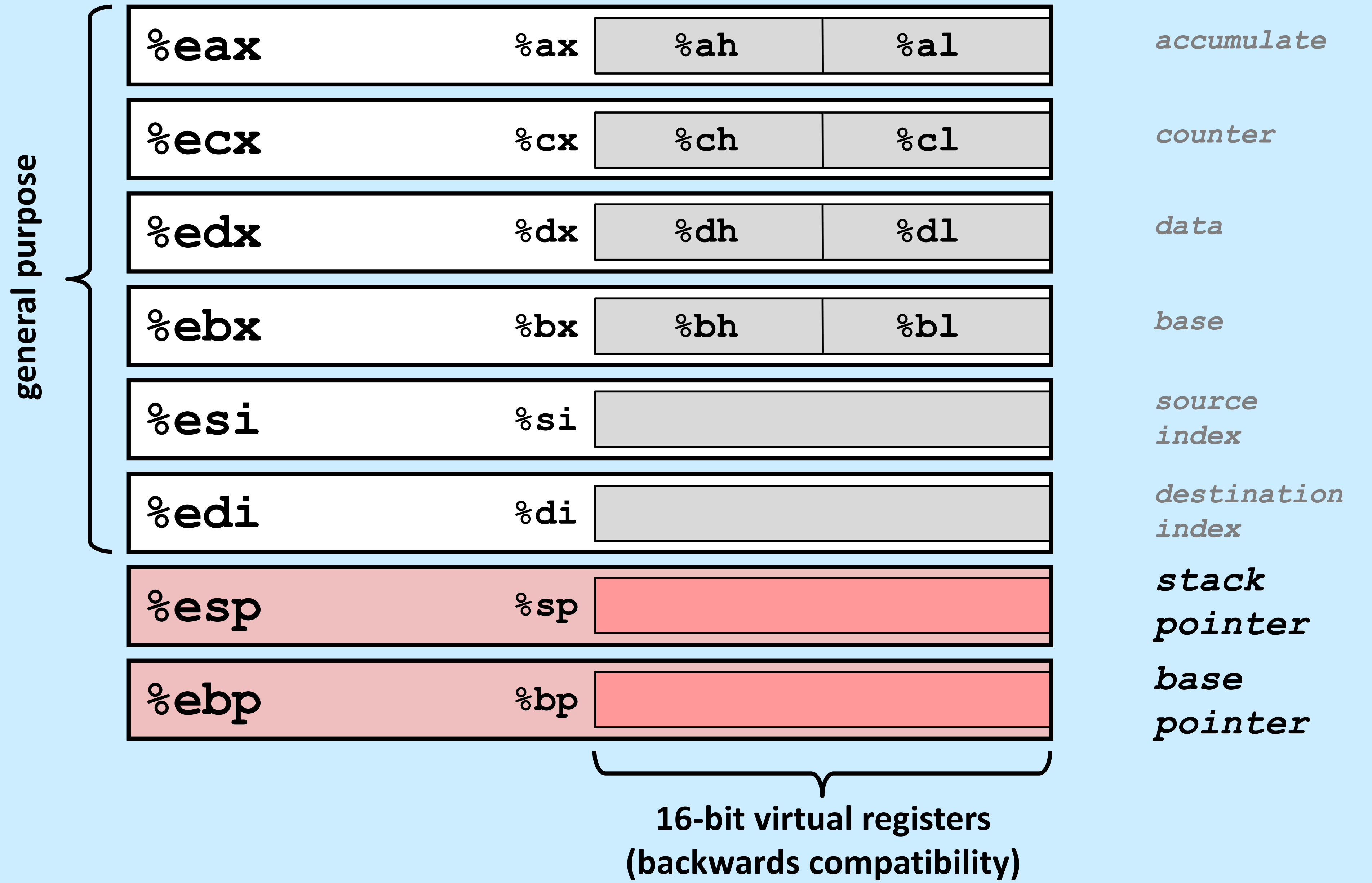
Operand Size



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

General-Purpose Registers (IA32)

Origin (mostly obsolete)



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

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

- **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"**

`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

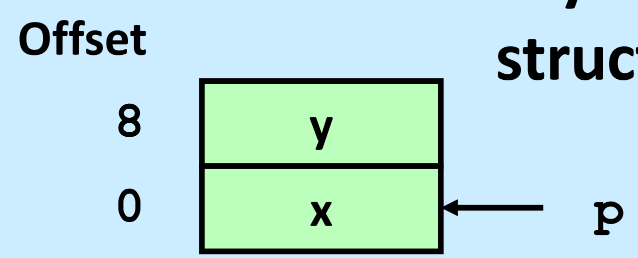
```
struct xy {
  long x;
  long y;
}
void swapxy(struct xy *p){
  long temp = p->x;
  p->x = p->y;
  p->y = temp;
}
```

```
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;
}
```

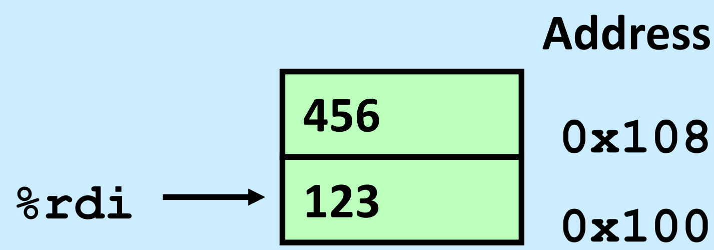
Layout of struct xy



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

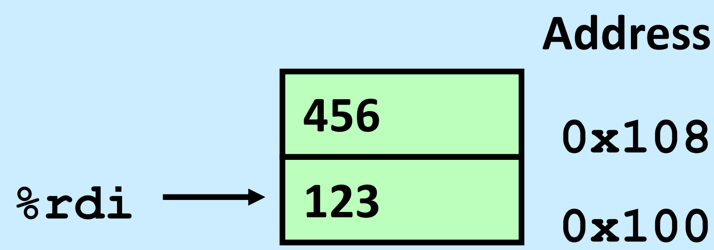


%rdi	0x100
%rax	
%rdx	

```
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

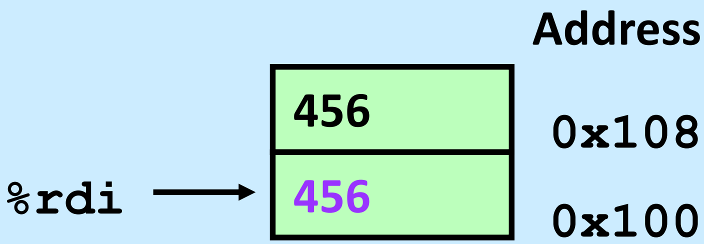
%rdi	0x100
%rax	123
%rdx	456

	Value	Address
	456	0x108
%rdi →	123	0x100

```
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
```

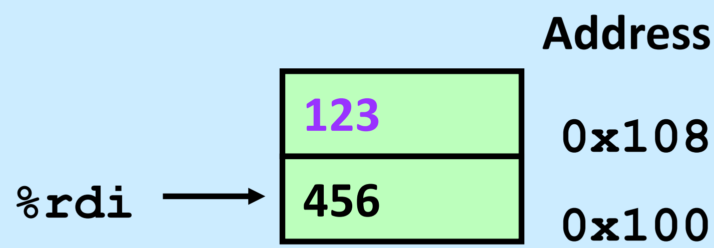
Understanding Swapxy

%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

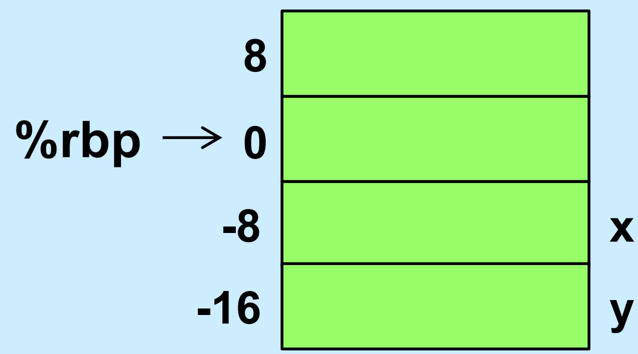


%rdi	0x100
%rax	123
%rdx	456

```
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 3

```
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 4

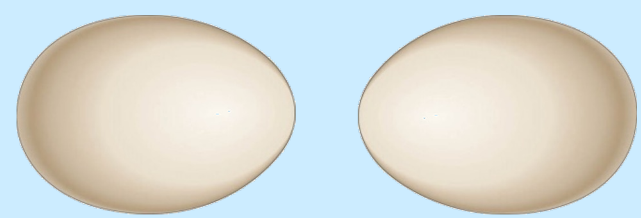
What value ends up in %ecx?

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

a) **0x04050607**
b) **0x07060504**
c) **0x06070809**
d) **0x09080706**

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

Hint:



Swapxy for Ints

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

```
swap:
  movl (%rdi), %eax
  movl 4(%rdi), %edx
  movl %edx, (%rdi)
  movl %eax, 4(%rdi)
  ret
```

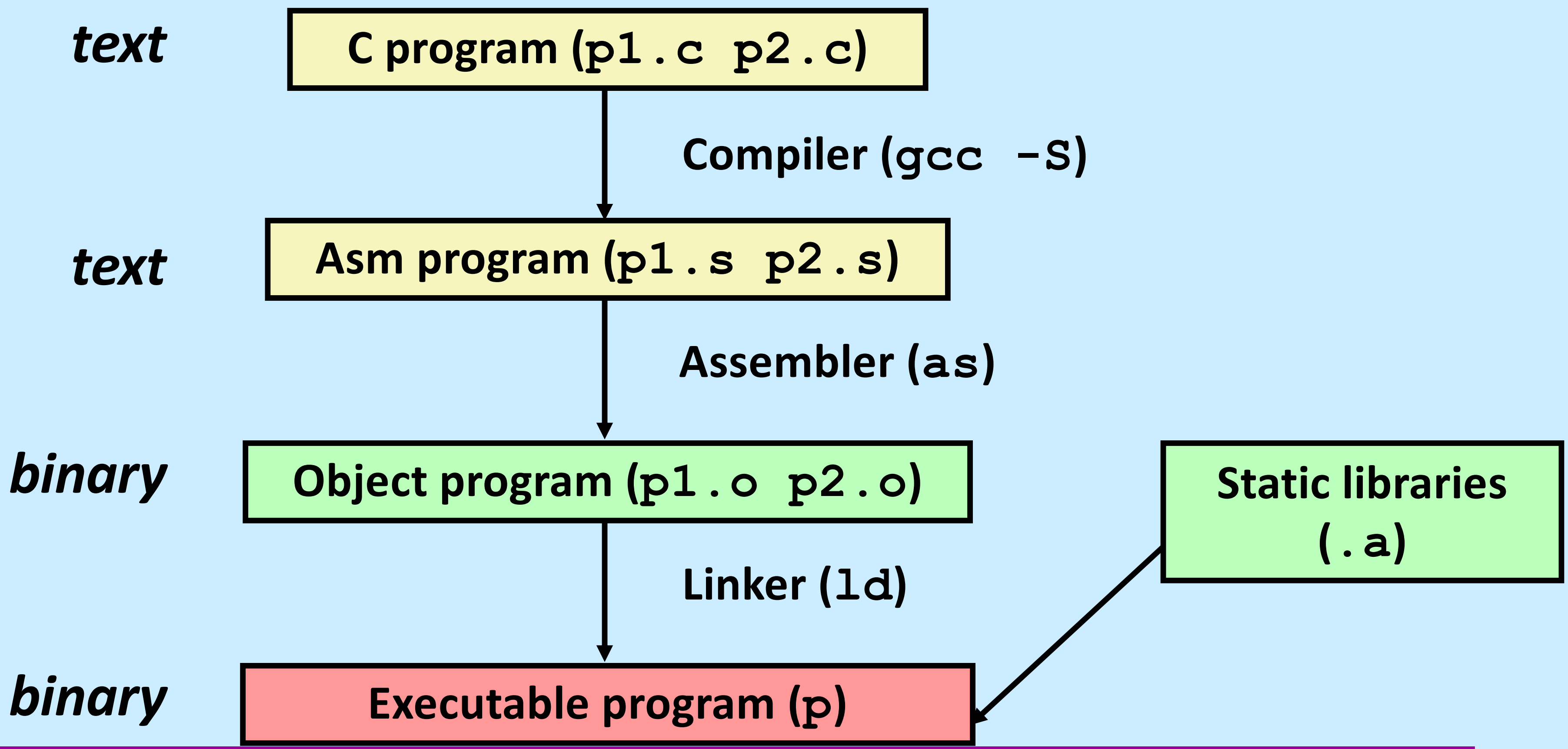
- **Pointers are 64 bits**
- **What they point to are 32 bits**

Bytes

- Each register has a byte version
  - e.g., %r10: %r10b; see earlier slide for x86 registers
- Needed for byte instructions
  - movb (%rax, %rsi), %r10b
  - sets *only* the low byte in %r10
    - » other seven bytes are unchanged
- Alternatives
  - movzbq (%rax, %rsi), %r10
    - » copies byte to low byte of %r10
    - » zeroes go to higher bytes
  - movsbq (%rax, %rsi), %r10
    - » copies byte to low byte of %r10
    - » sign is extended to all higher bits

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O1 p1.c p2.c -o p`
  - » use basic optimizations (`-O1`)
  - » put resulting binary in file `p`



Example

```
long ASum(long *a, unsigned long size) {
    long i, sum = 0;
    for (i=0; i<size; i++)
        sum += a[i];
    return sum;
}
```


Object Code

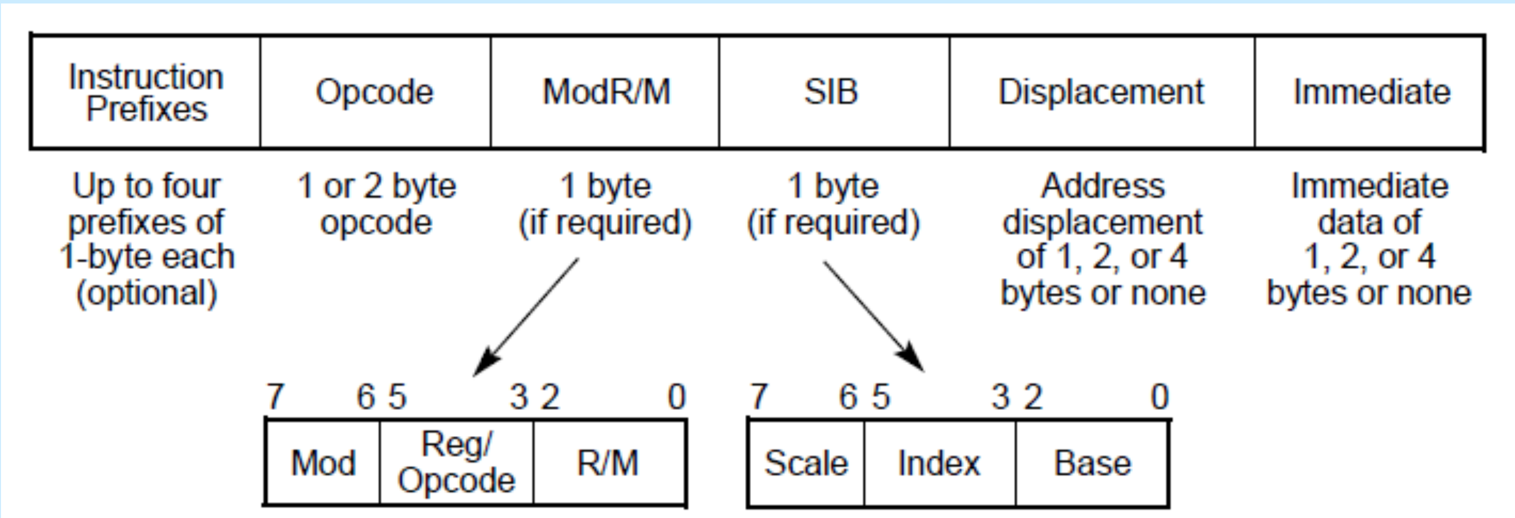
Code for `ASum`

```
0x112b <ASum>:
    0x48
    0x85
    0xf6
    0x74
    0x19
    0x48
    0x89
    0xfa
    0x48
    0x8d
    0x0c
    0xf7
     .
     .
     .
```

- Total of 35 bytes
- Each instruction: 1, 2, or 3 bytes
- Starts at address `0x112b`

- **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 `printf`
  - some libraries are *dynamically linked*
    - » linking occurs when program begins execution

Instruction Format



Disassembling Object Code

Disassembled

```
000000000000112b <ASum>:
    112b:  48 85 f6                 test   %rsi,%rsi
    112e:  74 19                    je     1149 <ASum+0x1e>
    1130:  48 89 fa                 mov    %rdi,%rdx
    1133:  48 8d 0c f7              lea    (%rdi,%rsi,8),%rcx
    1137:  b8 00 00 00 00           mov    $0x0,%eax
    113c:  48 03 02                 add    (%rdx),%rax
    113f:  48 83 c2 08              add    $0x8,%rdx
    1143:  48 39 ca                 cmp    %rcx,%rdx
    1146:  75 f4                    jne    113c <ASum+0x11>
    1148:  c3                       retq
    1149:  b8 00 00 00 00           mov    $0x0,%eax
    114e:  c3                       retq
```

- **Disassembler**

  `objdump -d <file>`

  – useful tool for examining object code

  – produces approximate rendition of assembly code

Alternate Disassembly

Object

```
0x112b:
   0x48
   0x85
   0xf6
   0x74
   0x19
   0x48
   0x89
   0xfa
   0x48
   0x8d
   0x0c
   0xf7
     .
     .
     .
```

Disassembled

```
Dump of assembler code for function ASum:
   0x112b <+0>:     test   %rsi,%rsi
   0x112e <+3>:     je     0x1149 <ASum+30>
   0x1130 <+5>:     mov    %rdi,%rdx
   0x1133 <+8>:     lea    (%rdi,%rsi,8),%rcx
   0x1137 <+12>:    mov    $0x0,%eax
     ...
```

- **Within gdb debugger**
  - `gdb <file>`
  - `disassemble ASum`
  - – disassemble the ASum object code
  - `x/35xb ASum`
  - – examine the 35 bytes starting at `ASum`

How Many Instructions are There?

- We cover ~30
- Implemented by Intel:
  - 80 in original 8086 architecture
  - 7 added with 80186
  - 17 added with 80286
  - 33 added with 386
  - 6 added with 486
  - 6 added with Pentium
  - 1 added with Pentium MMX
  - 4 added with Pentium Pro
  - 8 added with SSE
  - 8 added with SSE2
  - 2 added with SSE3
  - 14 added with x86-64
  - 10 added with VT-x
  - 2 added with SSE4a
- Total: 198
- Doesn't count:
  - floating-point instructions
    - ~100
  - SIMD instructions
    - lots
  - AMD-added instructions
  - undocumented instructions

Some Arithmetic Operations

- Two-operand instructions:

Format		Computation	
`addl`	Src,Dest	Dest = Dest + Src	
`subl`	Src,Dest	Dest = Dest – Src	
`imull`	Src,Dest	Dest = Dest * Src	
`shll`	Src,Dest	Dest = Dest << Src	Also called sall
`sarl`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrl`	Src,Dest	Dest = Dest >> Src	Logical
`xorl`	Src,Dest	Dest = Dest ^ Src	
`andl`	Src,Dest	Dest = Dest & Src	
`orl`	Src,Dest	Dest = Dest | Src	

  – watch out for argument order!

Some Arithmetic Operations

- **One-operand Instructions**

`incl`	Dest	= Dest + 1
`decl`	Dest	= Dest – 1
`negl`	Dest	= – Dest
`notl`	Dest	= ~Dest

- **See textbook for more instructions**

- **See Intel documentation for even more**

Arithmetic Expression Example

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

```
arith:
    leal   (%rdi,%rsi), %eax
    addl   %edx, %eax
    leal   (%rsi,%rsi,2), %edx
    shll   $4, %edx
    leal   4(%rdi,%rdx), %ecx
    imull  %ecx, %eax
    ret
```

Understanding `arith`

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

%rdx	z
%rsi	y
%rdi	x

```
leal   (%rdi,%rsi), %eax
addl   %edx, %eax
leal   (%rsi,%rsi,2), %edx
shll   $4, %edx
leal   4(%rdi,%rdx), %ecx
imull  %ecx, %eax
ret
```

Understanding `arith`

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

%rdx	z
%rsi	y
%rdi	x

```
leal   (%rdi,%rsi), %eax     # eax = x+y      (t1)
addl   %edx, %eax            # eax = t1+z     (t2)
leal   (%rsi,%rsi,2), %edx   # edx = 3*y      (t4)
shll   $4, %edx              # edx = t4*16    (t4)
leal   4(%rdi,%rdx), %ecx    # ecx = x+4+t4   (t5)
imull  %ecx, %eax            # eax *= t5      (rval)
ret
```

Observations about `arith`

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

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

```
leal   (%rdi,%rsi), %eax     # eax = x+y      (t1)
addl   %edx, %eax            # eax = t1+z     (t2)
leal   (%rsi,%rsi,2), %edx   # edx = 3*y      (t4)
shll   $4, %edx              # edx = t4*16    (t4)
leal   4(%rdi,%rdx), %ecx    # ecx = x+4+t4   (t5)
imull  %ecx, %eax            # eax *= t5      (rval)
ret
```

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192$, $2^{13} - 7 = 8185$

```
xorl %esi, %edi        # edi = x^y         (t1)
sarl $17, %edi         # edi = t1>>17      (t2)
movl %edi, %eax        # eax = edi
andl $8185, %eax       # eax = t2 & mask (rval)
```

Quiz 5

- **What is the final value in %ecx?**

```
xorl %ecx, %ecx
incl %ecx
shll %cl, %ecx  # %cl is the low byte of %ecx
addl %ecx, %ecx
```

a) 0
b) 2
c) 4
d) 8