程序的机器级表示（3）

谢旻晖

2025.10

Data Move

Move Instructions

- Format
  - mov src, dest (-->)
  - src and dest can only be one of the following
    - Immediate
    - Register
    - Memory

Move Instructions

	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

Data Movement

Instruction	Effect	Description
movq S, D	D ← S	Move quad word
movl S, D	D ← S	Move double word
movw S, D	D ← S	Move word
movb S, D	D ← S	Move byte
movsbl S, D	D ← SignedExtend(S)	Move sign-extended byte
movzbl S, D	D ← ZeroExtend(S)	Move zero-extended byte
pushl S	R[%esp] ← R[%esp]-4 M[R[%esp]] ← S	Push
popl D	D ← M[R[%esp]] R[%esp] ← R[%esp]+4	Pop

Data Movement Example

movl $0x4050, %eax	immediate	register
movl %ebp, %esp	register	register
movl (%edx, %ecx), %eax	memory	register
movl $-17, (%esp)	immediate	memory
movl %eax, -12(%ebp)	register	memory

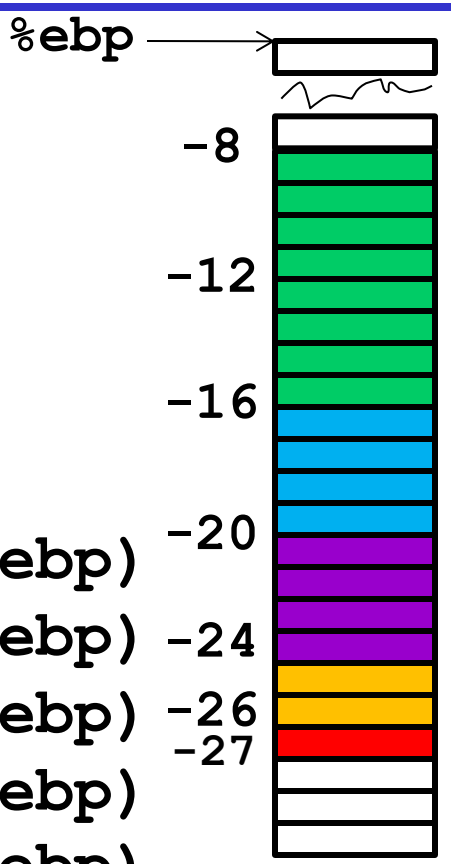
Data Formats

- Move data instruction
  - mov (general)
  - movb (move byte)
  - movw (move word)
  - movl (move double word)
  - movq (move quadruple word)

Access Objects with Different Sizes

```
int main(void){
   char c = 1;   short s = 2;
   int i = 4;    long l = 4L;
   long long ll = 8LL;
   return;
}
```

```
8048335:c6 movb   $0x1,0xffffffe5(%ebp)
8048339:66 movw   $0x2,0xffffffe6(%ebp)
804833f:c7 movl   $0x4,0xffffffe8(%ebp)
8048346:c7 movl   $0x4,0xffffffec(%ebp)
804834d:c7 movl   $0x8,0xfffffff0(%ebp)
8048354:c7 movl   $0x0,0xfffffff4(%ebp)
```



Array in Assembly

Persistent usage

- Store the base address

```
void f(void){
    int i, a[16];
    for(i=0; i<16; i++)
      a[i]=i;
}
movl %edx,-0x44(%ebp,%edx,4)

a:   -0x44(%ebp)
i:   %edx
```

Data Movement Example

Initial value %dh=8d %eax =98765432

1 movb %dh, %al %eax=9876548d
2 movsbl %dh, %eax %eax=ffffff8d
3 movzbl %dh, %eax %eax=0000008d

- 1-byte registers
  - %al, %ah, %cl, %ch, %dl, %dh, %bl, %bh
- 2-byte registers
  - %ax, %cx, %dx, %bx, %si, %di, %sp, %bp

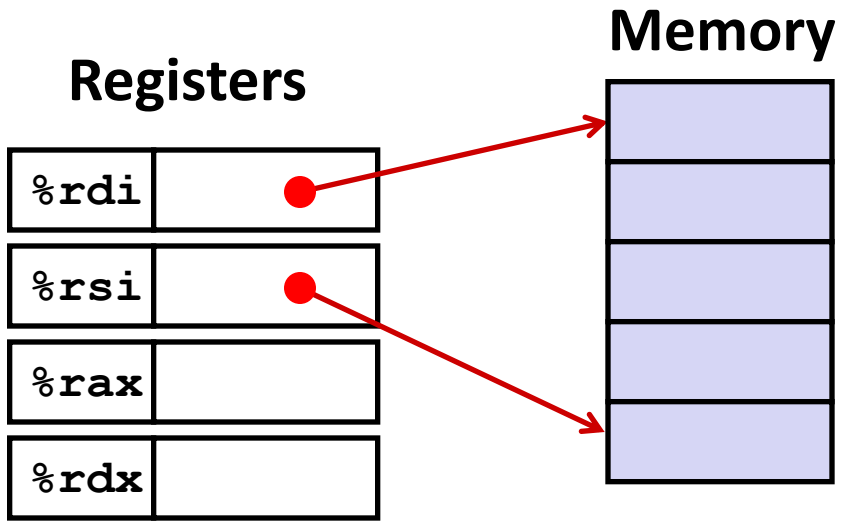
Example of Simple Addressing Modes

```
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;
}
```

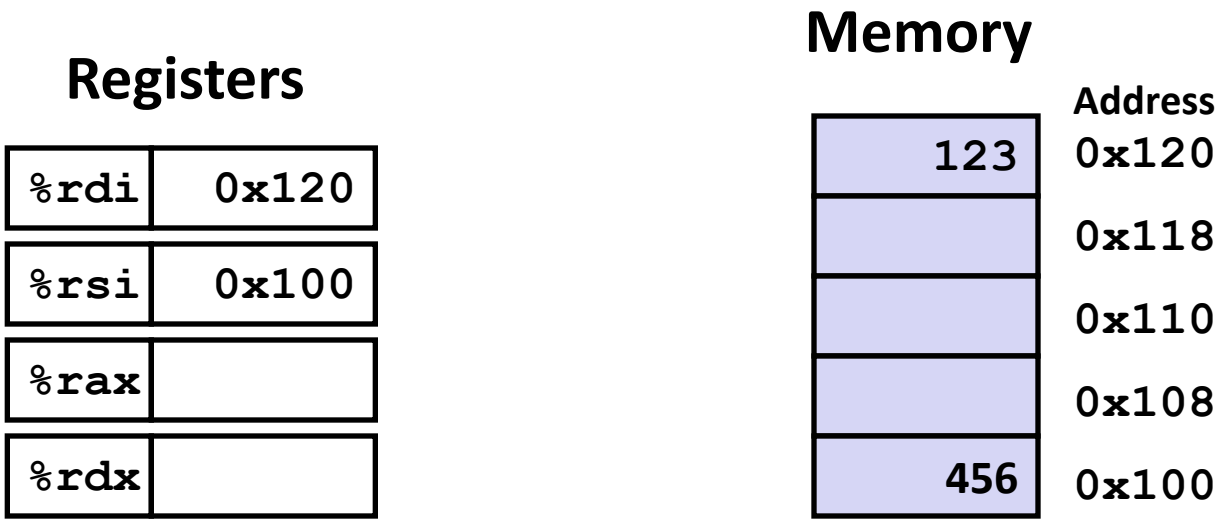


约定俗成

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

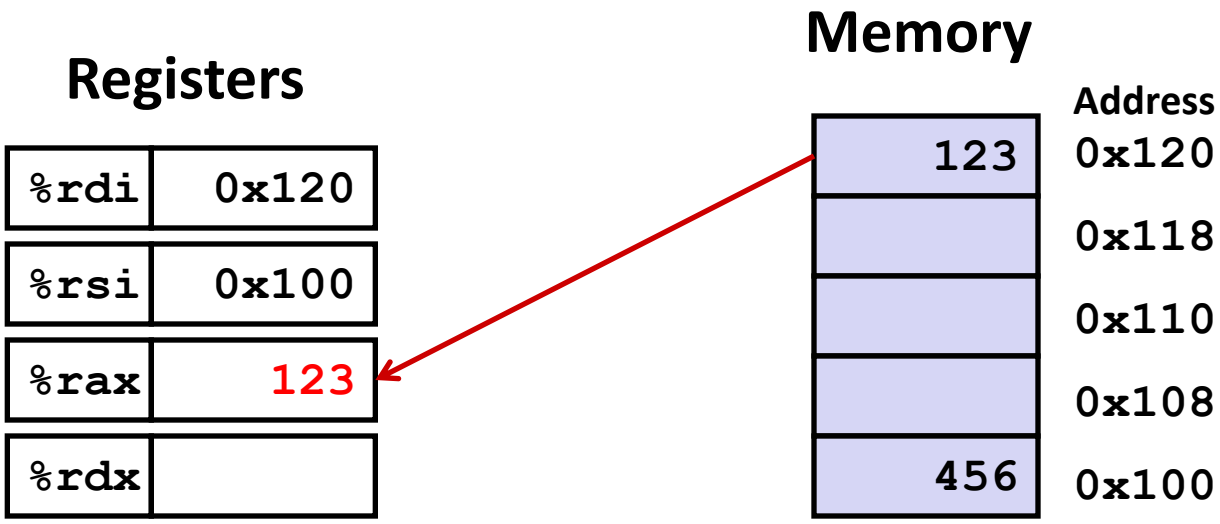
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



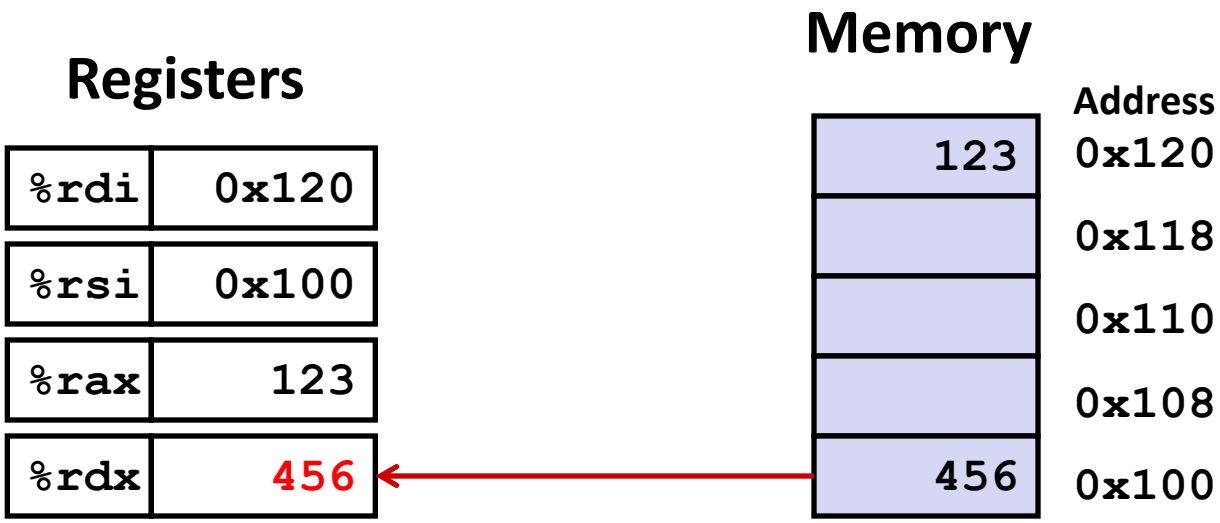

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding Swap()



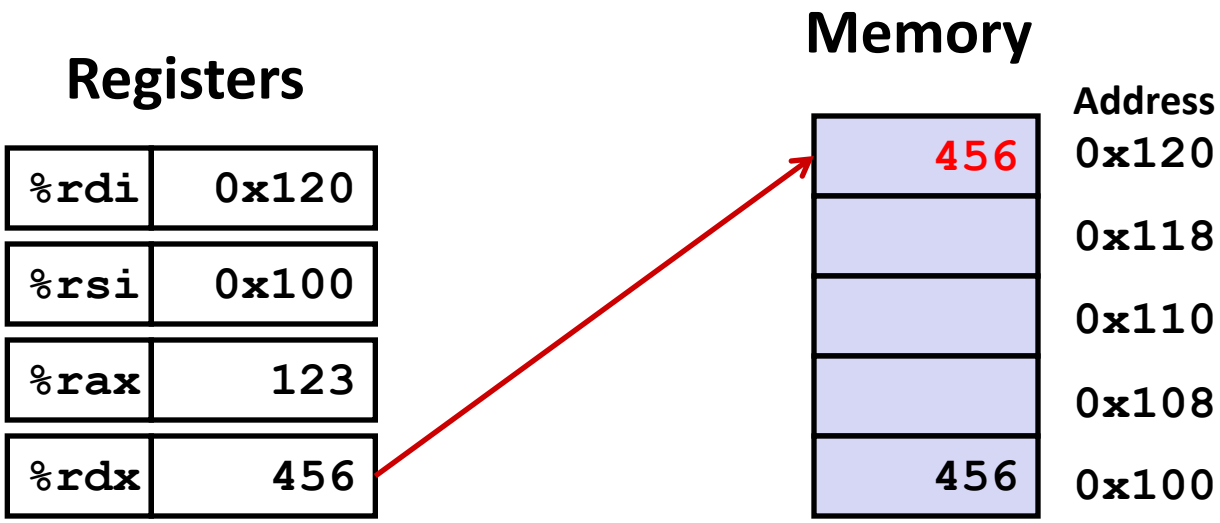

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



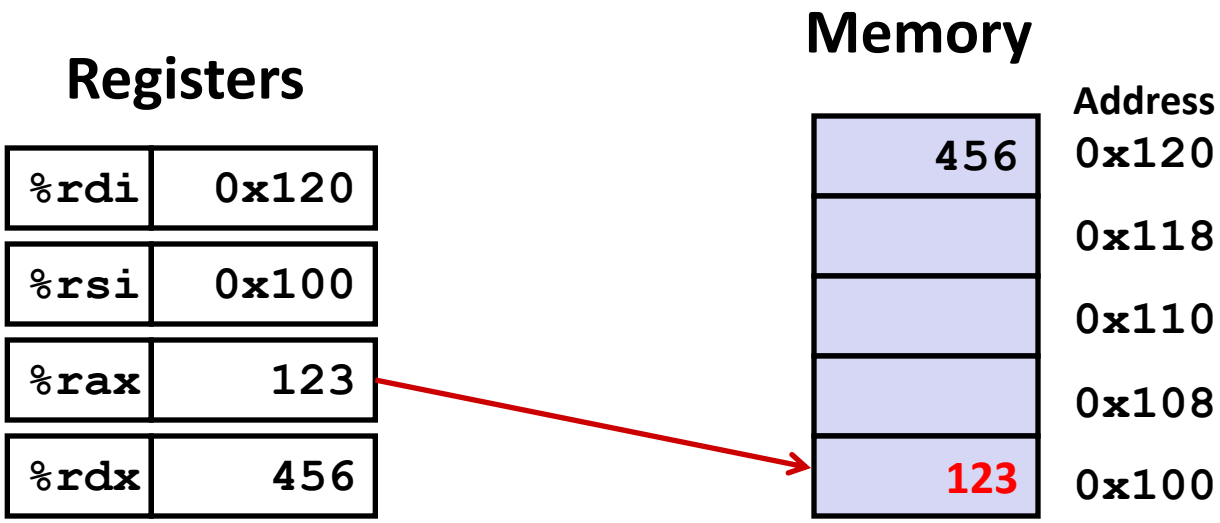

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

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
```

MOV 指令遵循的规则

- 两个操作数的尺寸必须一致
- 两个操作数不能同时为内存操作数
- 目的操作数不能是CS，EIP和IP
- 立即数不能直接送至段寄存器
- 段寄存器之间不能直接传送数据
- 不能一次传送多个对象
- 标识寄存器（EFlags）和指令指针寄存器（IP）不能用MOV指令设置

课堂练习

- 下列汇编语句错在哪里？
  - movb $0xF, (%bl)
  - movl %ax, (%esp)
  - movw (%eax), 4(%esp)
  - movl %eax, %dx
  - movb %si, 8(%esp)
  - movb %ah, %sh
  - movl %eax, 0x123

主观题 10分

设置

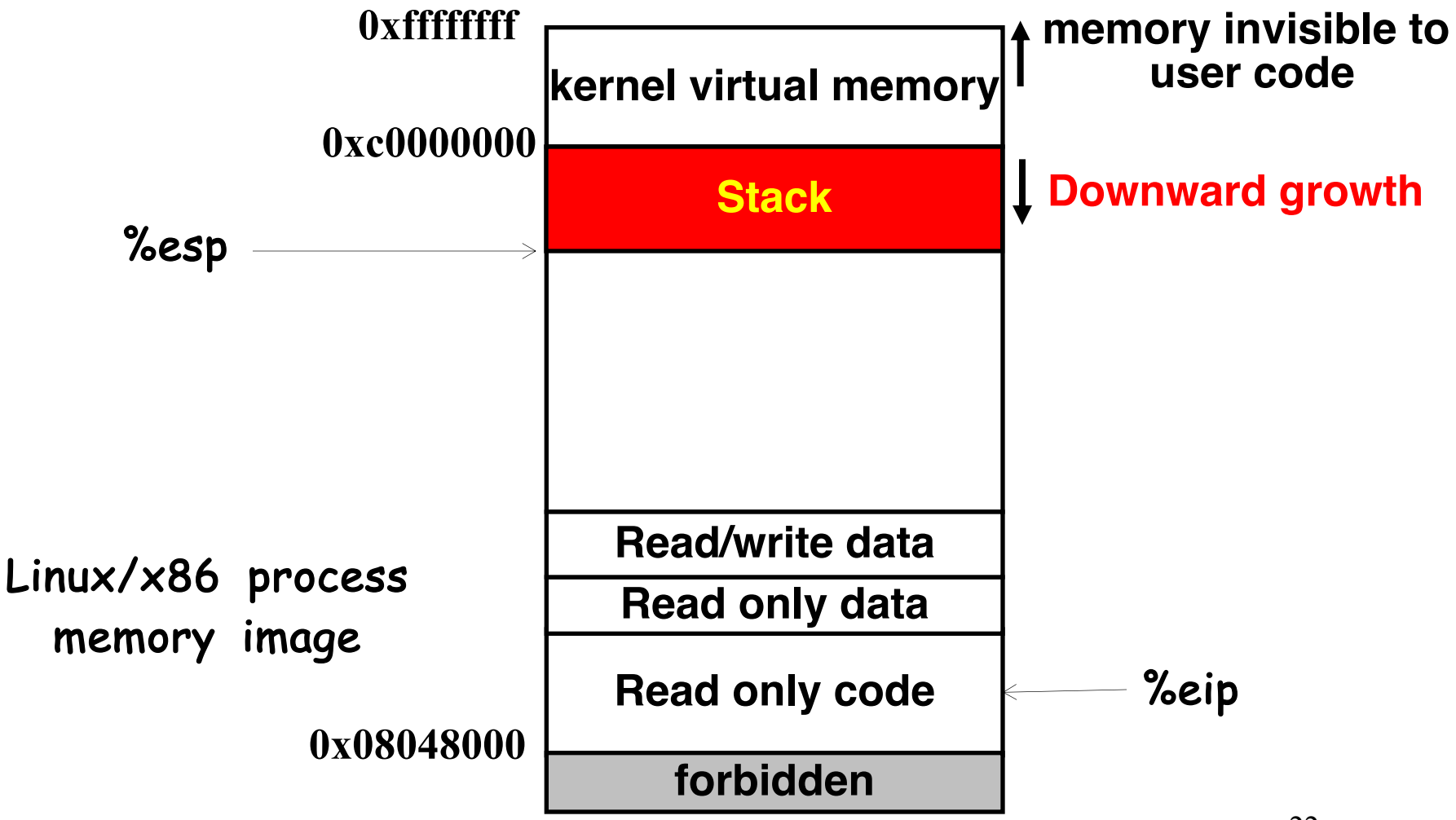
- 已知原型为
- Void decode1 (long *xp, long *yp, long *zp);
- 的函数编译成汇编，得到如下代码
- # xp in %rdi, yp in %rsi, zp in %rdx
- Decode1:
  - Movq (%rdi), %r8
  - Movq (%rsi), %rcx
  - Movq (%rdx), %rax
  - Movq %r8, (%rsi)
  - Movq %rcx, (%rdx)
  - Movq %rax, (%rdi)
- 请写出等效的C语言代码

作答

Stack operation

- Stack is a special kind of data structure
  - Last in first out (LIFO)
- The top of the stack must be explicitly specified
  - It is denoted as top
- There are two operations on the stack
  - push and pop
- There is a hardware stack in x86
  - 速度快，通过push, pop等指令操作
  - its bottom has high address number
  - its top is indicated by %esp

Stack Layout



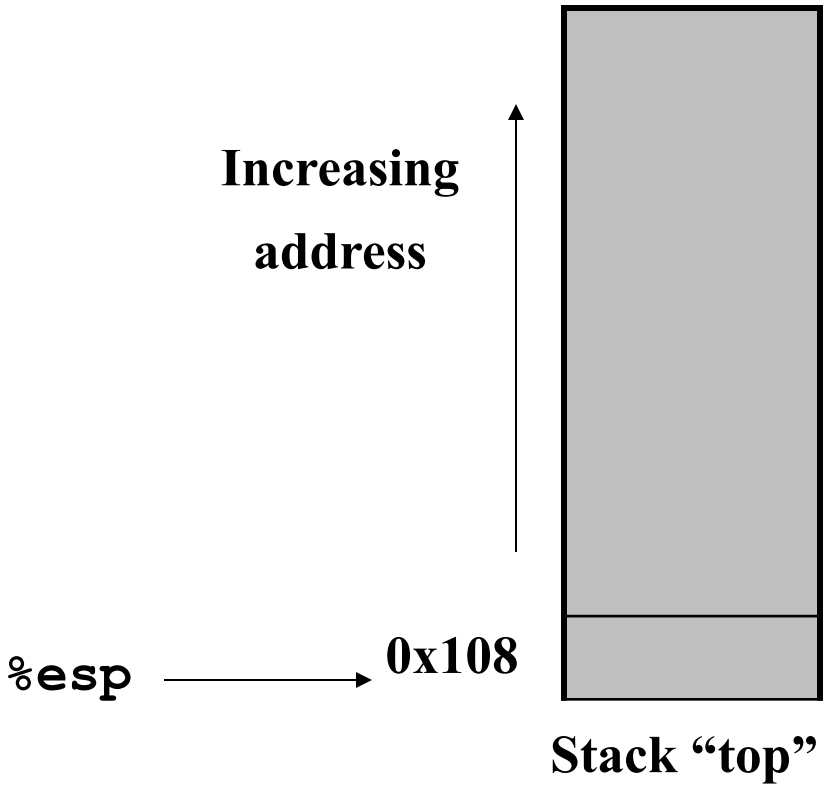
Stack operation

- There are two stack operation instructions
  - Push and Pop
- Push
  - decreases the %esp (enlarge the stack)
  - stores the value in a register into the stack
- Pop
  - stores the value in the top of the stack into a register
  - increases the %esp (shrink the stack)

Stack operations

%eax	0x123
%edx	0
%esp	0x108

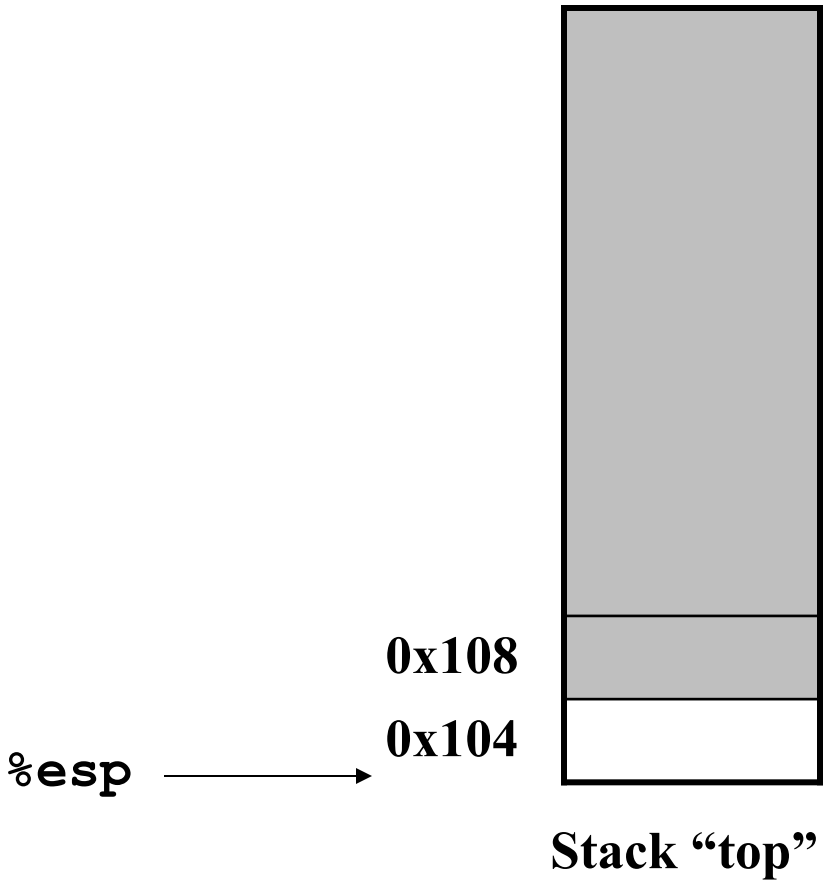
pushl %eax ?



Stack operations

%eax	0x123
%edx	0
%esp	0x104

pushl %eax

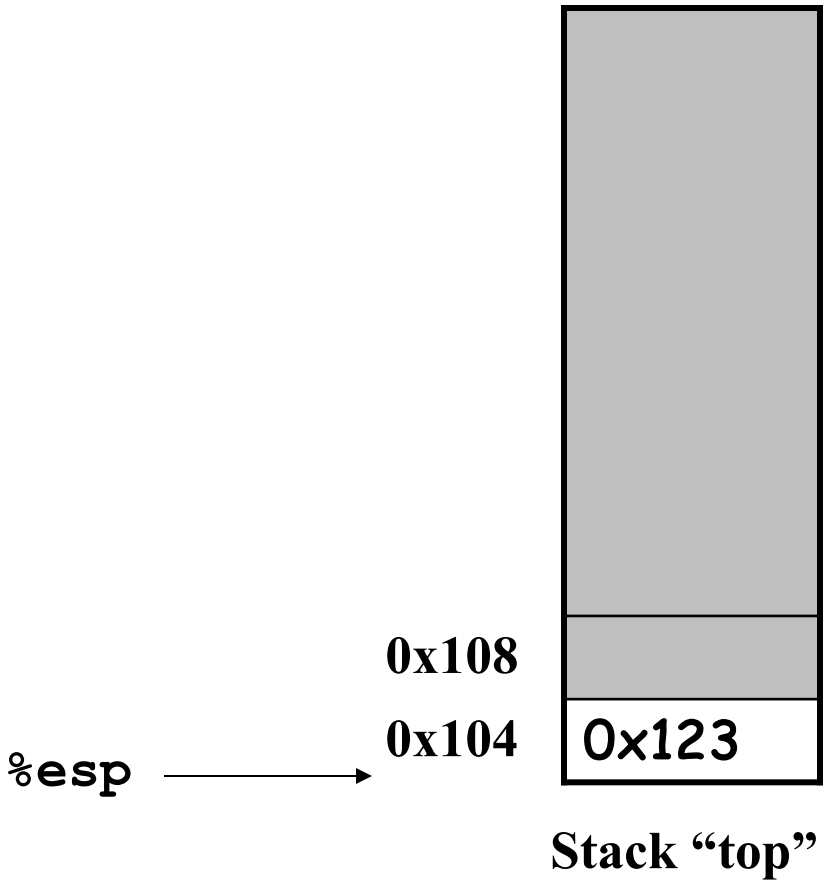


Stack operations

%eax	0x123
%edx	0
%esp	0x104

pushl %eax

popl %edx ?

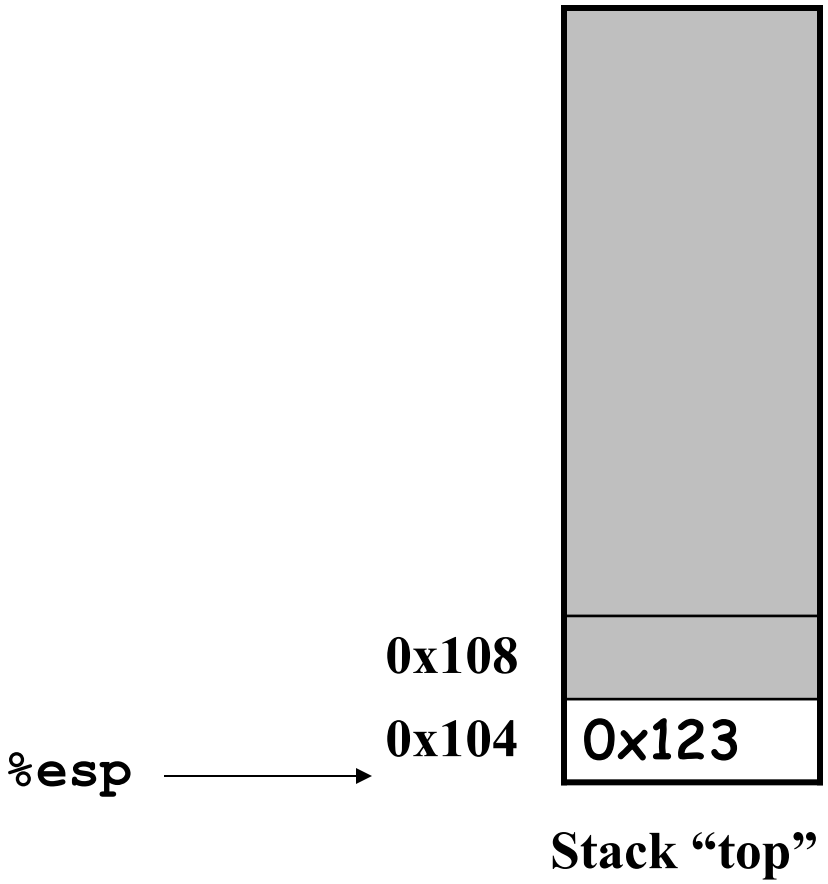


Stack operations

%eax	0x123
%edx	0x123
%esp	0x104

pushl %eax

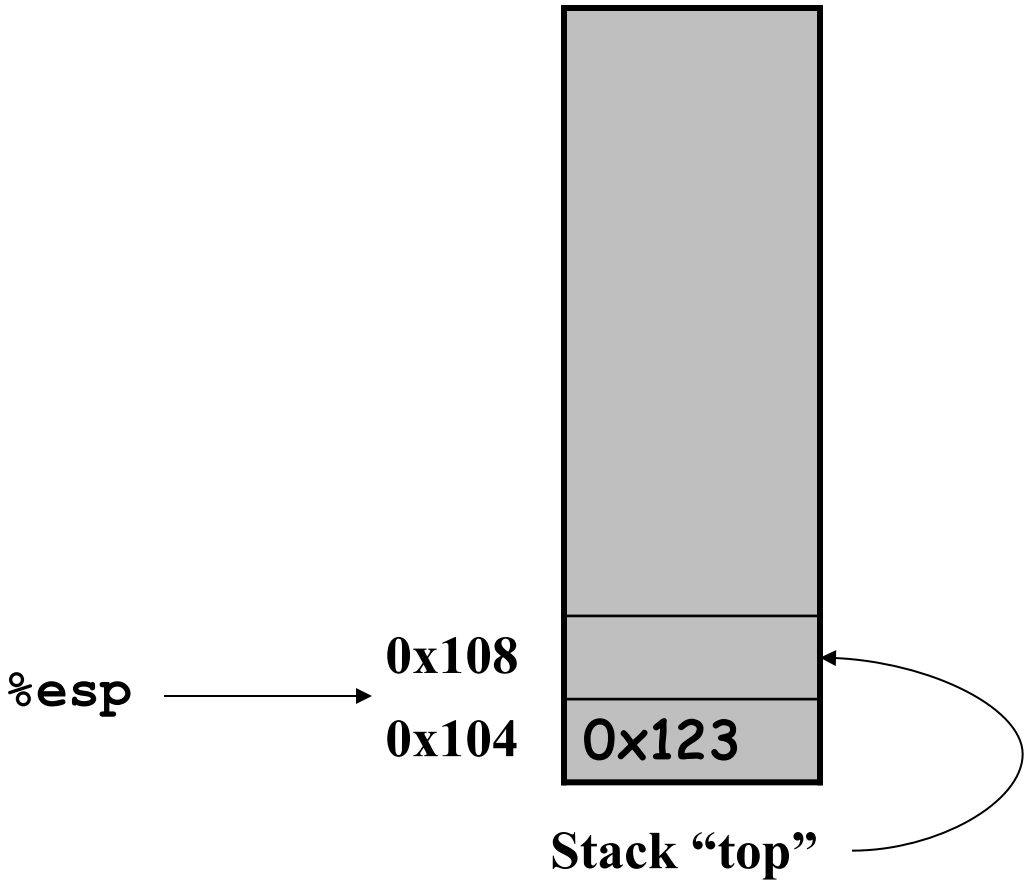
popl %edx ?



Stack operations

%eax	0x123
%edx	0x123
%esp	0x108

popl %edx



Stack Operation

Instruction	Effect	Description
pushl S	R[%esp] ← R[%esp]-4 M[R[%esp]] ← S	Push
popl D	D ← M[R[%esp]] R[%esp] ← R[%esp]+4	Pop

pushq/popq

Pointers vs. Addresses

Outline

- Pointer in C
- Data Movement Example

Pointers in C

- In C, an object may be
  - an integer
  - a structure or
  - some other program unit
- Declaring Pointers

```
T *p;
```

Pointers in C

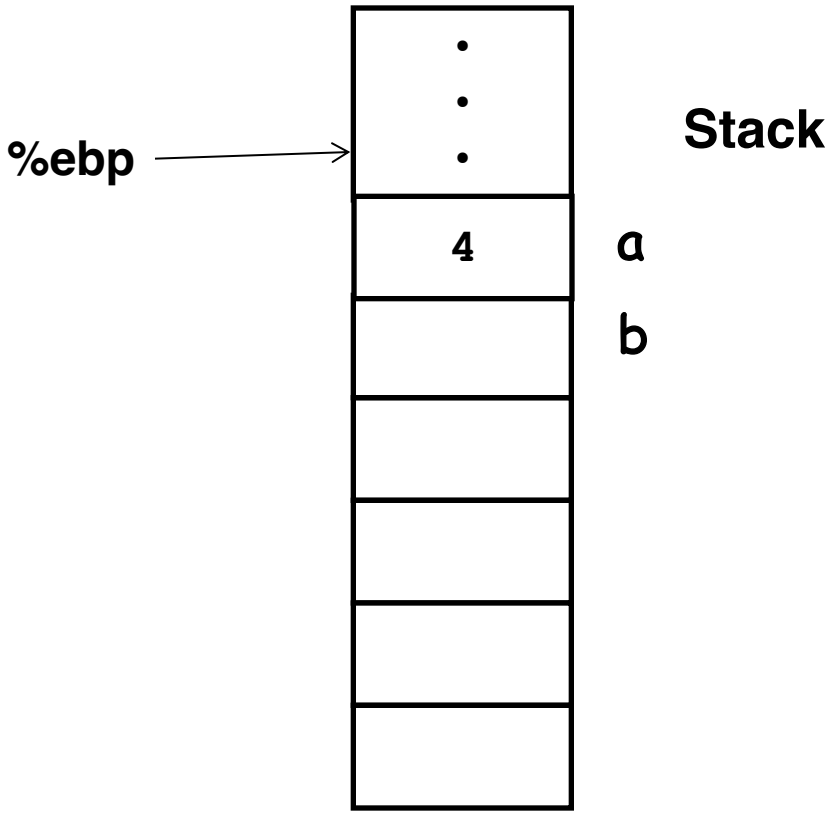
- In C, the value of a pointer in C is
  - the virtual address of the first byte of some block of storage
  - Type information is used to
    - identify the length of the object
    - interpret the object *p
- Generating Pointers

  p = &obj

Example

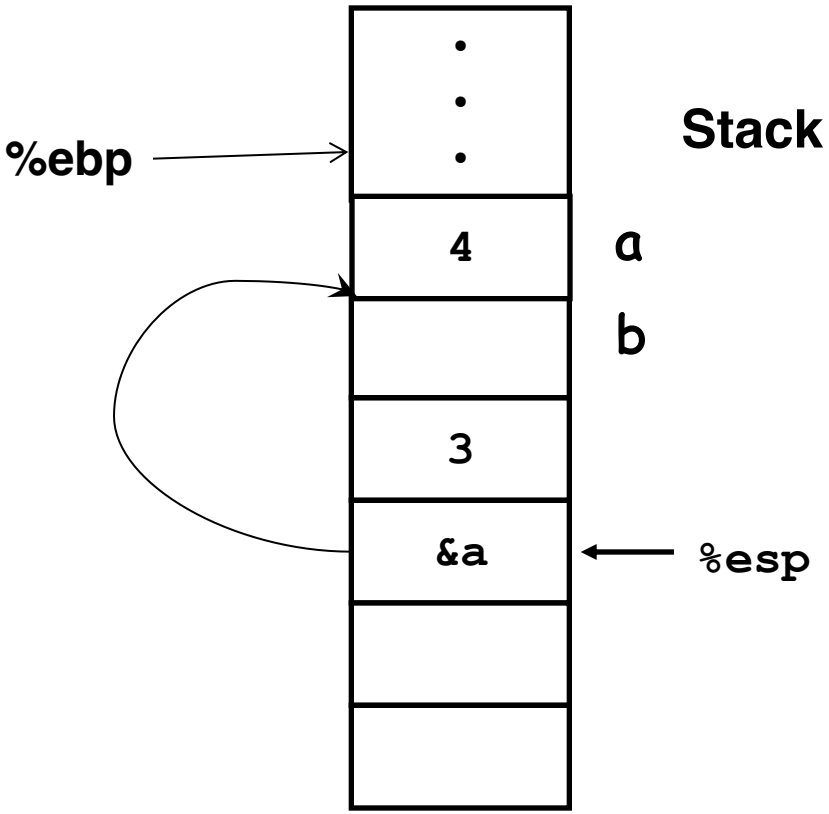
```
int main()
{
    int a = 4 ;
    /* “address of” operator creates a pointer */
    int b = exchange(&a, 3);
    printf(“a = %d, b = %d\n”, a, b);
}
```

Example



Can the variable a be in a register?

Example



Example

```
int exchange(int *xp, int y)
{
     /* operator * performs deferencing */
    int x = *xp ;
    *xp = y ;
    return x ;
}
```

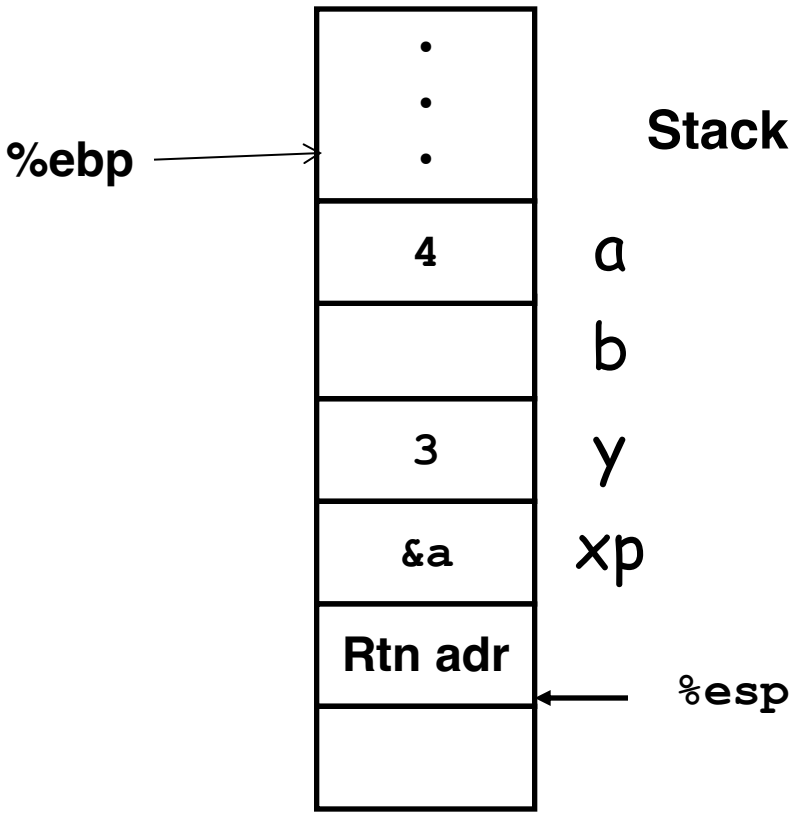
Example

```
int exchange(int *xp, int y)
{
    int x = *xp ;

    *xp = y ;
    return x ;
}
```

```
1 pushl  %ebp
2 movl   %esp, %ebp
3 movl   8(%ebp),  %eax #xp
4 movl   12(%ebp), %edx #y
5 movl   (%eax), %ecx # x
6 movl   %edx,    (%eax)
7 movl   %ecx,   %eax
8 movl   %ebp, %esp
9 popl   %ebp
```

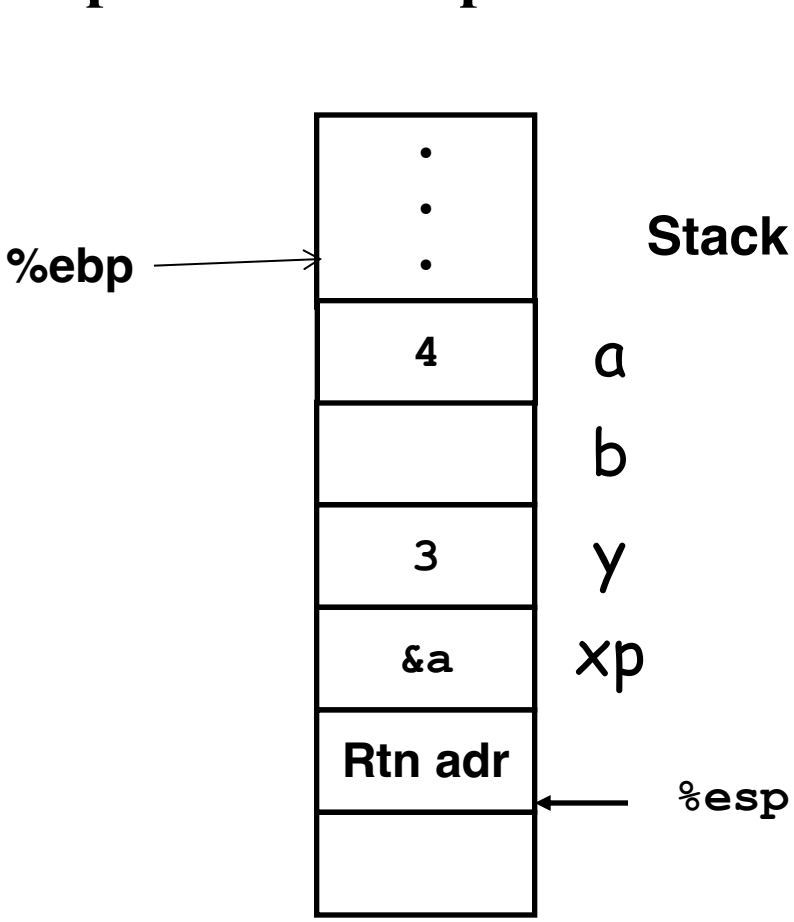
Example

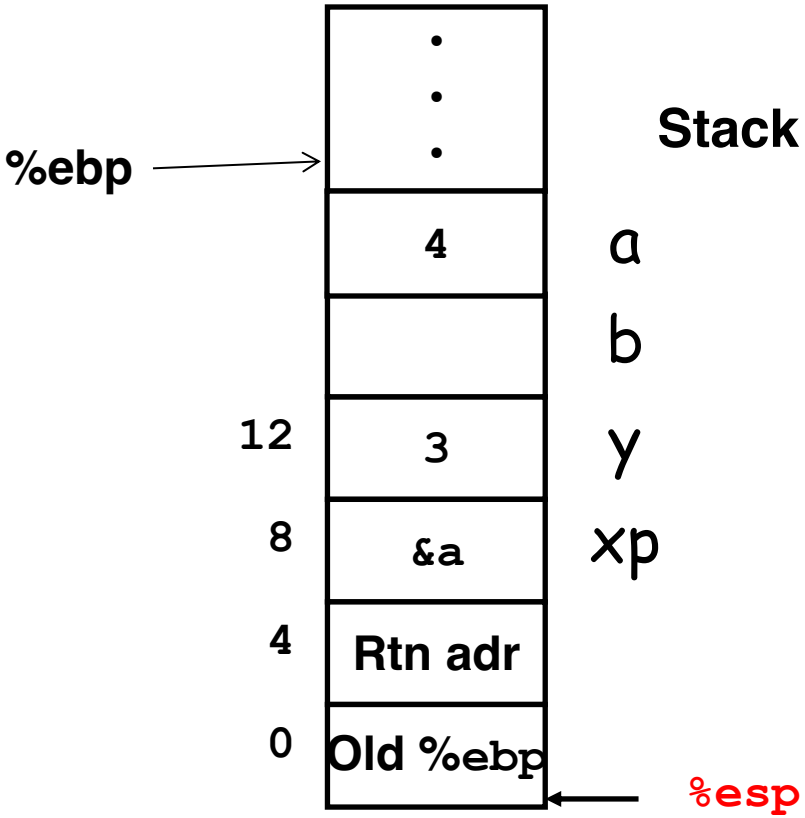


- 上次讲到这里

Example

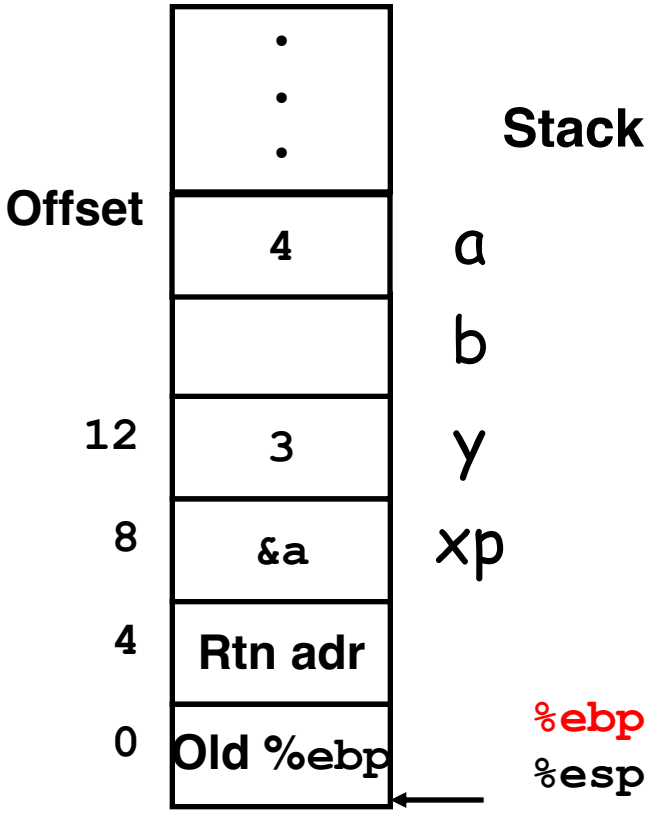
1 pushl %ebp





Example

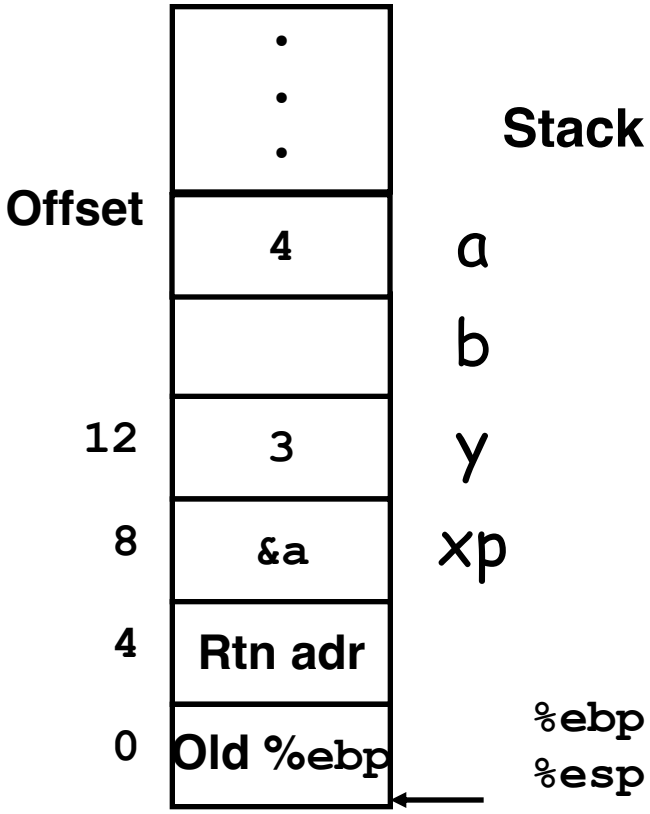
2 movl %esp, %ebp



Example

3 movl 8(%ebp), %eax

%eax: xp



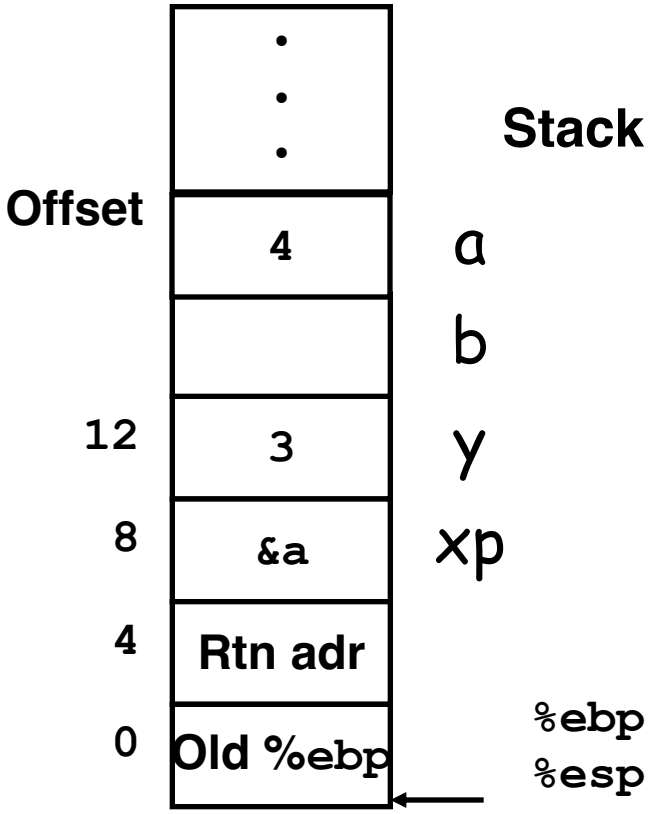
Example

3 movl 8(%ebp), %eax

4 movl 12(%ebp), %edx

%eax: xp

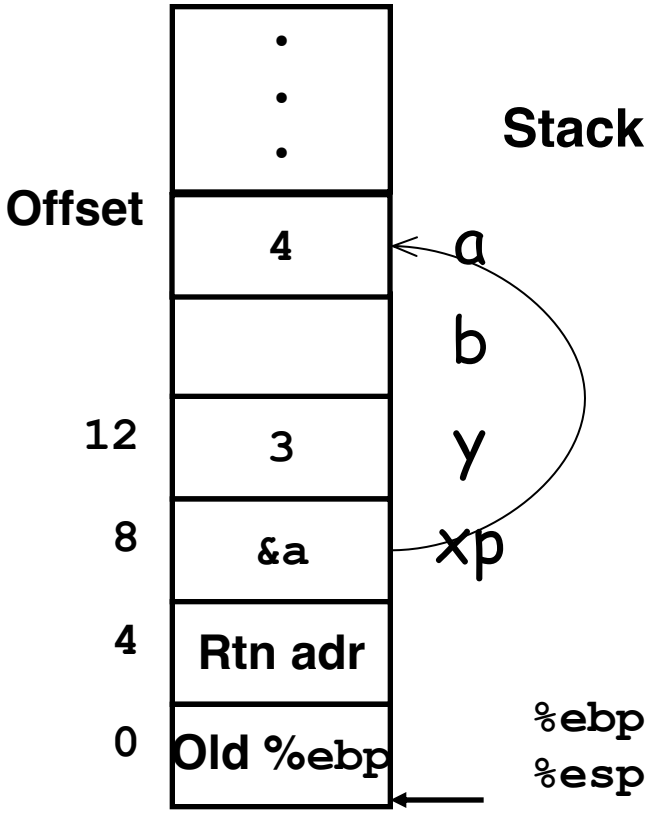
%edx: 3



Example

```
3 movl     8(%ebp),  %eax
4 movl     12(%ebp), %edx
5 movl     (%eax),   %ecx
```

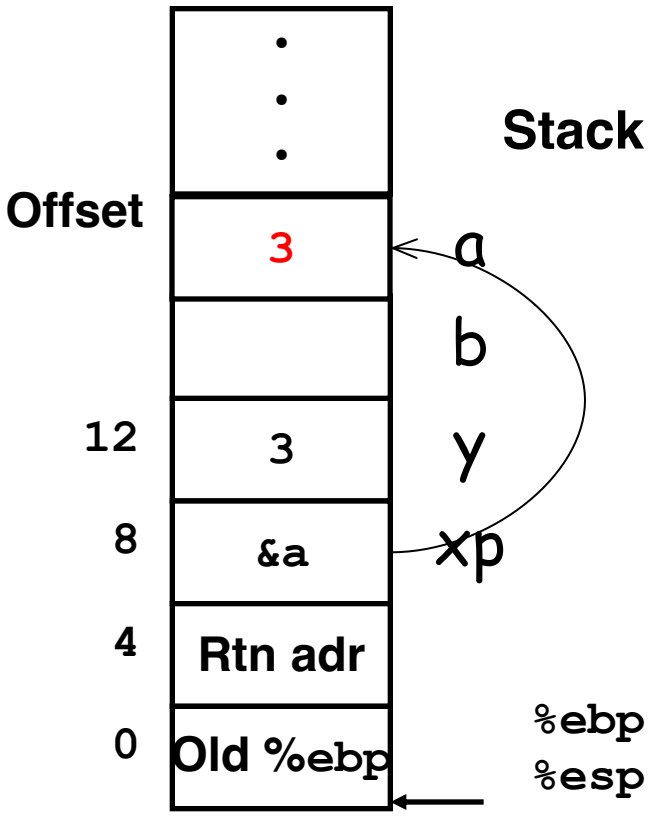
%eax: xp
%edx: 3
%ecx: 4



Example

```
3 movl    8(%ebp),  %eax
4 movl    12(%ebp), %edx
5 movl    (%eax),   %ecx
6 movl    %edx,     (%eax)
```

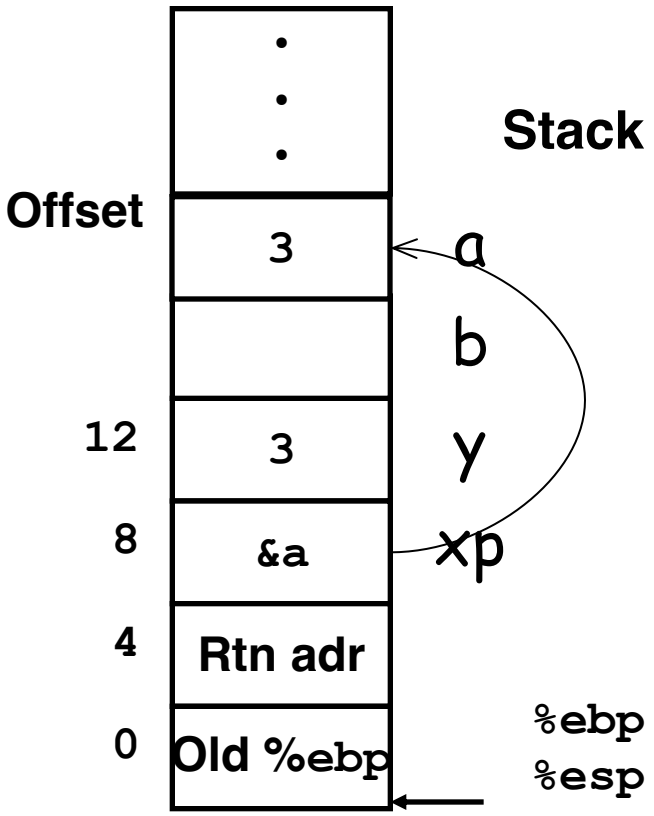
%eax: xp
%edx: 3
%ecx: 4
*xp(a): 3



Data Movement Example

```
3 movl    8(%ebp),  %eax
4 movl    12(%ebp), %edx
5 movl    (%eax),   %ecx
6 movl    %edx,     (%eax)
7 movl    %ecx,     %eax
```

%eax: xp
%edx: y
%ecx: 4
*xp(a): 3
%eax: 4 (old *xp) return value

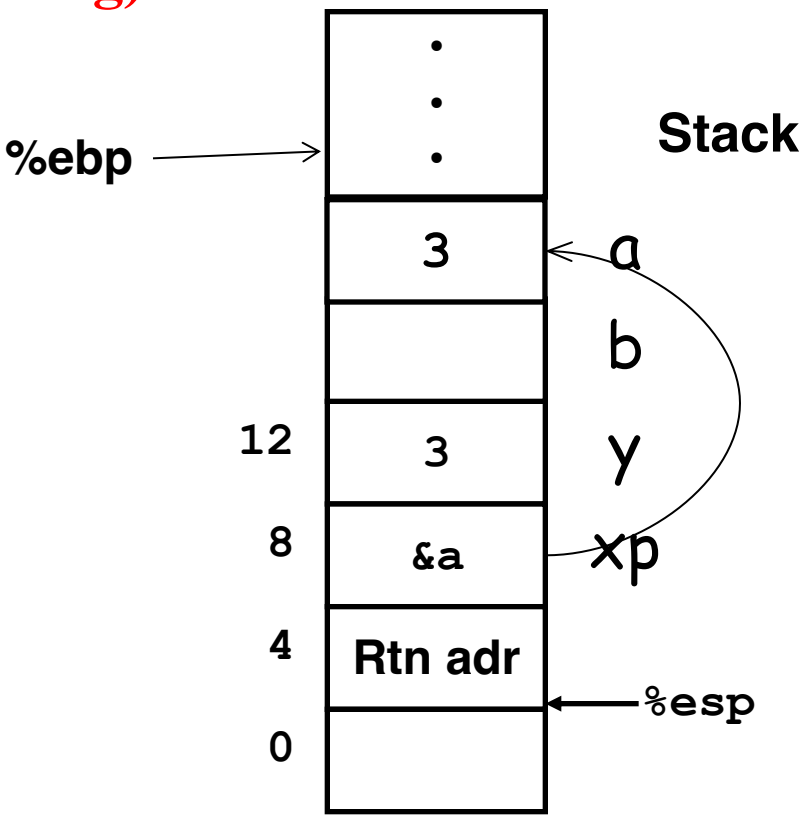


默认eax为返回值寄存器

Data Movement Example

8 movl %ebp, %esp (do nothing)

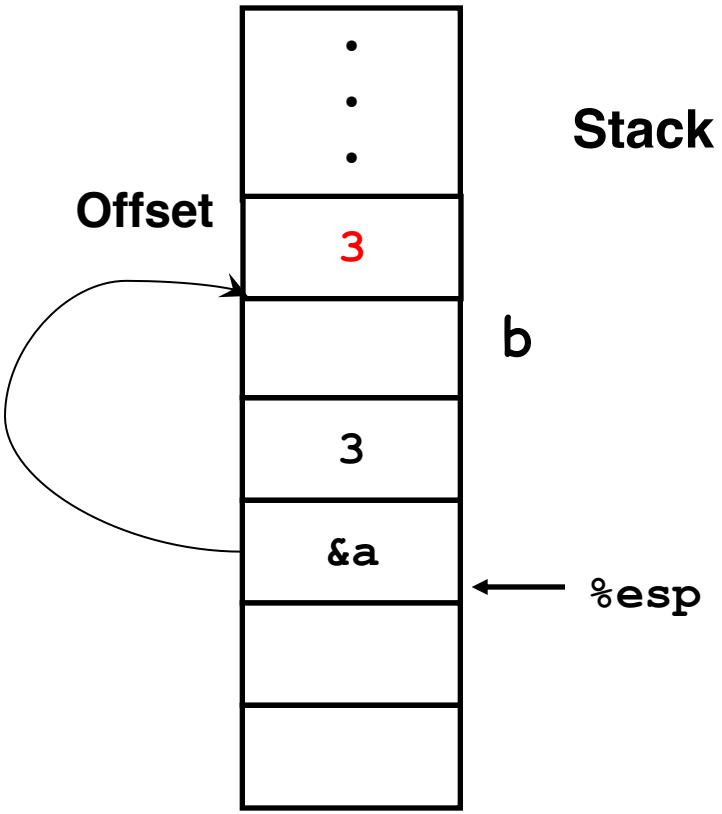
9 popl %ebp



Example

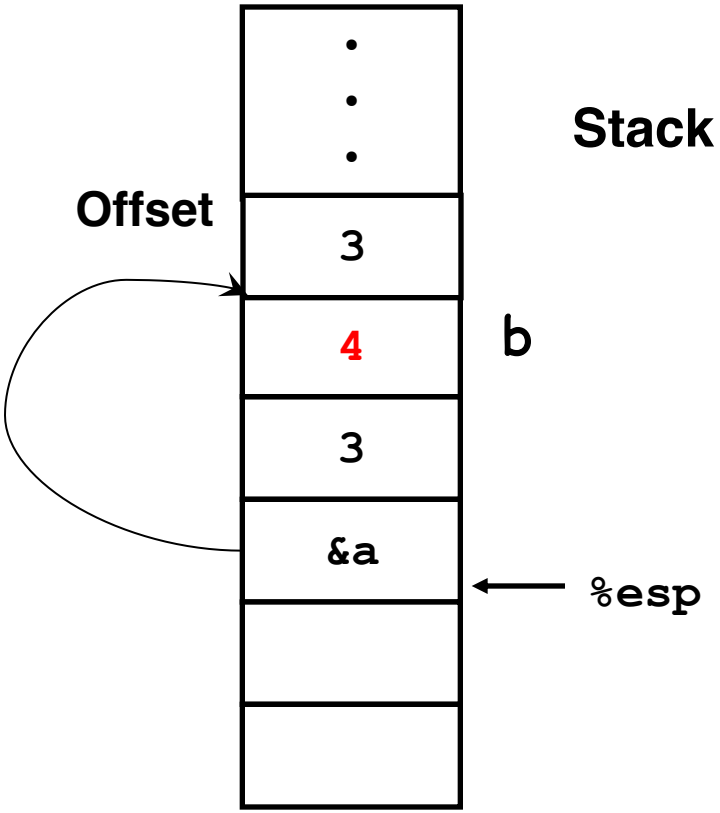
```
int main()
{
    int a = 4 ;
    /* “address of” operator creates a local pointer
    variable */
    int b = exchange(&a, 3);
    printf(“a = %d, b = %d\n”, a, b);
}
```

Example



%eax 4
Return value

Example



Data Manipulation

Arithmetic and Logical Operations

Instruction	Effect	Description
leal S, D	D ← &S	Load effective address
incl D	D ← D + 1	Increment
decl D	D ← D – 1	Decrement
negl D	D ← -D	Negate
notl D	D ← ~D	Complement
addl S, D	D ← D + S	Add
subl S, D	D ← D – S	Subtract
cmpl S, D	D – S	Subtract
imull S, D	D ← D * S	Multiply

Arithmetic and Logical Operations

Instruction	Effect	Description
xorl S, D	D ← D ^ S	Exclusive-or
orl S, D	D ← D | S	Or
andl S, D	D ← D & S	And
testl S, D	D & S	And
sall k, D	D ← D << k	Left shift, k为8 bit
shll k, D	D ← D << k	Left shift
sarl k, D	D ← D >> k	Arithmetic right shift
shrl k, D	D ← D >> k	Logical right shift

Arithmetic and Logical Operations

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Instruction	Destination	Value
addl %ecx, (%eax)	0x100	0x100
subl %edx, 4(%eax)	0x104	0xA8
imull $16, (%eax, %edx, 4)	0x10C	0x110
incl 8(%eax)	0x108	0x14
decl %ecx	%ecx	0x0
subl %edx, %eax	%eax	0xFD

Examples for Lea Instruction

%eax holds x, %ecx holds y

Expression	Result
leal 6(%eax), %edx	6+x
leal (%eax, %ecx), %edx	x+y
leal (%eax, %ecx, 4), %edx	x+4*y
leal 7(%eax, %eax, 8), %edx	7+9*x
leal 0xA(, %ecx, 4), %edx	10+4*y
leal 9(%eax, %ecx, 2), %edx	9+x+2*y

Assembly Code for Arithmetic Expressions

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z*48;
  int t3 = t1&0xFFFF;
  int t4 = t2*t3;
  return t4;
}
```

```
movl 12(%ebp),%eax        Get y
movl 16(%ebp),%edx        Get z
addl 8(%ebp),%eax         Compute t1=x+y
leal (%edx,%edx,2),%edx   Compute 3*z
sall $4,%edx              Compute t2=48*z
andl $0xFFFF,%eax         Compute t3=t1&FFFF
imull %edx,%eax           Set t4 as return val
```

Special Arithmetic Operations

imull S	R[%edx]:R[%eax] ←S*R[%eax]	Signed full multiply
mull S	R[%edx]:R[%eax] ←S*R[%eax]	Unsigned full multiply
Cltd	R[%edx]:R[%eax] ← SignExtend(R[%eax])	Convert to quad word
idiv S	R[%edx] ← R[%edx]:R[%eax] mod S R[%eax] ← R[%edx]:R[%eax] ÷ S	Signed divide
divl S	R[%edx] ←R[%edx]:R[%eax] mod S R[%eax] ← R[%edx]:R[%eax] ÷ S	Unsigned divide

Examples

Initially x at %ebp+8, y at %ebp+12

```
1 movl 8(%ebp), %eax
2 imull 12(%ebp)
3 pushl %edx   #高位
4 pushl %eax   #低位
```

```
1 movl 8(%ebp), %eax
2 cltd
3 idivl 12(%ebp)
4 pushl %eax  #商
5 pushl %edx  #余数
```

课堂练习

- long shift_left4_rightn(long x, long n) {
  - x <<= 4;
  - x >>= n;
  - return x;
- }对应的汇编为：
- # x in %rdi, n in %rsi
- Shift_left4_rightn:
  - Movq　　%rdi, %rax
  - _____　　# x <<= 4
  - Movl　　%esi, %ecx
  - _____　　# x >>= n
  - 请参考注释，补全代码

填空题 2分

设置

此题未设置答案，请点击右侧设置按钮

```
long shift_left4_rightn(long x, long n) {
    x <<= 4;
    x >>= n;
    return x;
}对应的汇编为：

# x in %rdi, n in %rsi
Shift_left4_rightn:
    Movq        %rdi, %rax
    _______ [填空1]_________        # x <<= 4
    Movl        %esi, %ecx
    _______ [填空2] _________       # x >>= n
    请参考注释，补全代码
```

作答

练习答案

- long shift_left4_rightn(long x, long n) {
  - x <<= 4;
  - x >>= n;
  - return x;
- }对应的汇编为：
- # x in %rdi, n in %rsi
- Shift_left4_rightn:
  - Movq　　%rdi, %rax
  - __ Salq $4, %rax_____　　# x <<= 4
  - Movl　　%esi, %ecx
  - __ Sarq %cl, %rax_____　　# x >>= n
  - 请参考注释，补全代码

课堂练习

- Long arith2(long x, long y, long z) {
  - Long t1 = _____;
  - Long t2 = _____;
  - Long t3 = _____;
  - Long t4 = _____;
  - Return t4;
- }

- Arith2:
  - # x@%rdi, y@rsi, z@rdx
  - Orq %rsi, %rdi
  - Sarq $3, %rdi
  - Notq %rdi
  - Movq %rdx, %rax
  - Subq %rdi, %rax
  - Ret
- 填写空缺的C语言语句

课堂答案

- Long arith2(long x, long y, long z) {
  - Long t1 = _x | y_;
  - Long t2 = _t1 >> 3_;
  - Long t3 = _~t2__;
  - Long t4 = _z-t3_;
  - Return t4;
- }

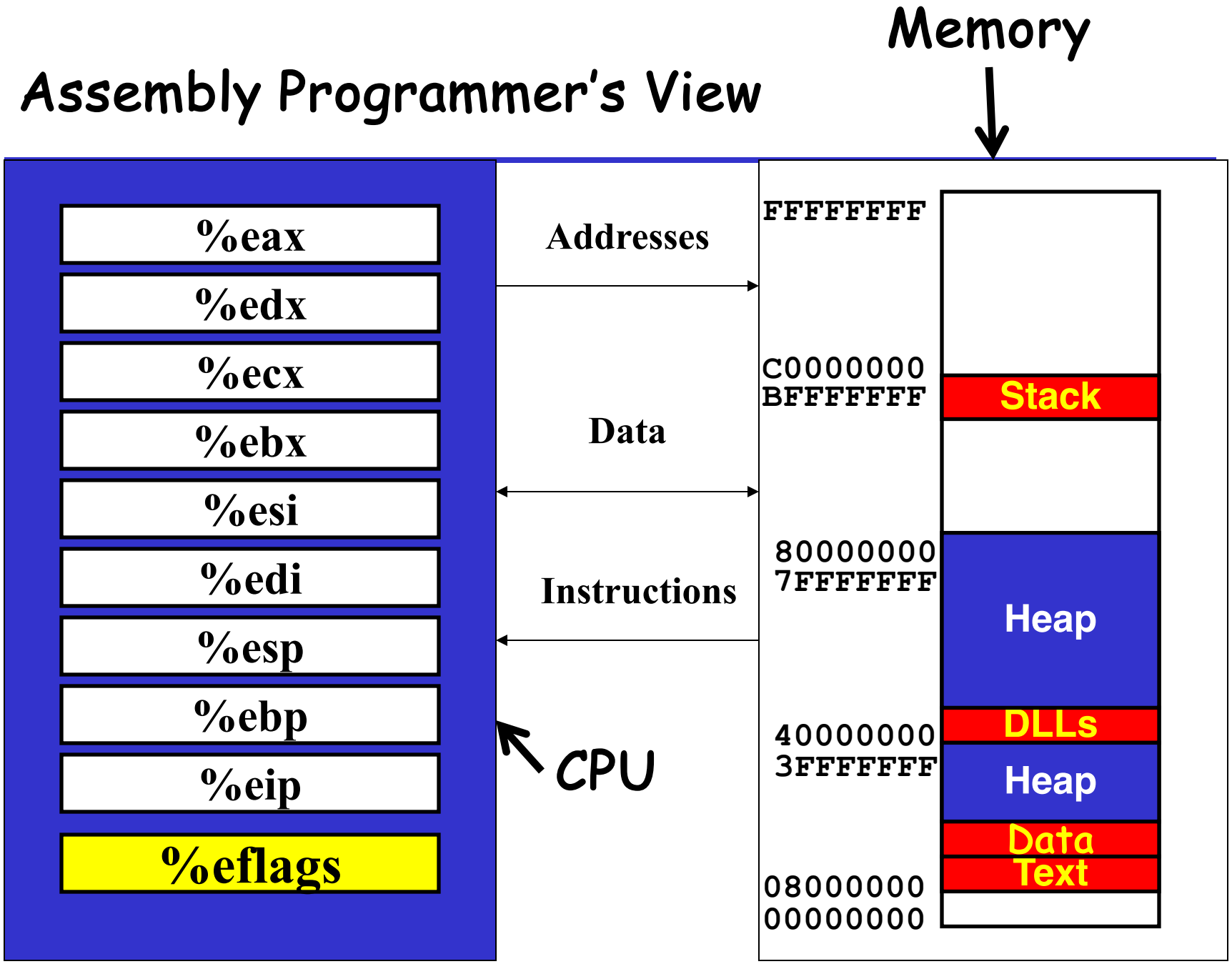
- Arith2:
  - #x@%rdi, y@rsi, z@rdx
  - Orq %rsi, %rdi
  - Sarq $3, %rdi
  - Notq %rdi
  - Movq %rdx, %rax
  - Subq %rdi, %rax
  - Ret
- 填写空缺的C语言语句

Conditional Codes

Outline

- Conditional Codes
- Accessing the Conditional Codes

Assembly Programmer's View



Condition codes

- Condition codes
  - A set of single-bit
  - Maintained in a condition code register (Eflags)
  - Describe attributes of the most recently arithmetic or logical operation

Condition codes

- EFLAGS (运算过程型)
  - CF: Carry Flag
    - The most recent operation generated a carry out of the most significant bit
    - Used to detect overflow for unsigned operations
  - OF: Overflow Flag
    - The most recent operation caused a two's complement overflow — either negative or positive

Condition codes

- EFLAGS (结果状态型)
  - ZF: Zero Flag
    - The most recent operation yielded zero
  - SF: Sign Flag
    - The most recent operation yielded a negative value

Setting Conditional Codes

- Implicit Setting By Arithmetic Operations

  `addl` *Src,Dest*

  C analog: `t = a+b`

  - CF set if carry out from most significant bit
    - Used to detect unsigned overflow
  - ZF set if `t == 0`
  - SF set if `t < 0`
  - OF set if two's complement overflow

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

Conditional Code (特殊规则)

- lea instruction (数据移动类的都不影响EFlags)
  - has no effect on condition codes
- Xorl instruction
  - The carry and overflow flags are set to 0
- Shift instruction
  - Carry flag is set to the last bit shifted out
  - Overflow flag is set to 0
- Inc/dec instruction
  - Do NOT set carry flag
- Mull/imull instruction
  - CF=OF=0 iff the first half of result is useless; else, CF=OF=1

Setting Conditional Codes

- Explicit Setting by Compare Instruction
  - **`cmpl`** ***Src2,Src1***
  - **`cmpl 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`
  - OF set if two's complement overflow

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

Setting Conditional Codes

- Explicit Setting by Test instruction
  - **`testl`** ***Src2,Src1***
  - Sets condition codes based on value of *Src1* `&` *Src2*
    - Useful to have one of the operands be a mask
  - **`testl b,a`** like computing `a&b` without setting destination
  - ZF set when `a&b == 0`
  - SF set when `a&b < 0`

例：加法运算

判断SF, ZF, CF, OF

```
movw $0075h, %ax                ;ax=0075h
movw $01a2h, %cx                ;cx=01a2h
addw, %cx, %ax                  ;ax=0217h
```

SF=0、ZF=0、CF=0 、 OF=0

```
movw $77ach, %ax,               ;ax=77ach
movw $4b35h, %cx                ;cx=4b35h
addw %cx, %ax                   ;ax=c2e1h
```

SF=1、ZF=0、CF=0 、 OF=1

例：减法运算

```
movl $75h, %eax                ;ax=00000075h
movl $1a2h, %ecx               ;cx=000001a1h
subl %ecx, %eax                ;ax=fffffed3h
```

SF=1、ZF=0、CF=1(借位) 、 OF=0

```
movw $c36eh, %ax               ;ax=c36eh
movw $ef00h, (%ebx)            ; (%ebx)=ef00h
subw %ax, (%ebx)               ; (%ebx)=2b92h
```

SF=0、ZF=0、CF=0 、 OF=0

Accessing Conditional Codes

- Conditional codes不能直接读写（像其他寄存器一样用move等指令）
- 写入：主要是通过相关的运算来设置标识位
- 读取：
  - 通过set系列指令，根据eflags状态来设置参数的值
    - Sete %al
  - 通过一些特定指令来读取判断（条件跳转语句，条件传送数据等）
- 特殊方法：pushf, popf可以用于修改eflags

Accessing Conditional Codes

Instruction	Synonym	Effect	Set Condition
Sete	Setz	ZF	Equal/zero
Setne	Setnz	~ZF	Not equal/not zero
Sets		SF	Negative
Setns		~SF	Nonnegative
Setl	Setnge	SF^OF	Less
Setle	Setng	(SF^OF)|ZF	Less or Equal
Setg	Setnle	~(SF^OF)&~ZF	Greater
Setge	Setnl	~(SF^OF)	Greater or Equal
Seta	Setnbe	~CF&~ZF	Above
Setae	Setnb	~CF	Above or equal
Setb	Setnae	CF	Below
Setbe	Setna	CF|ZF	Below or equal

有符号数

无符号数

Accessing Conditional Codes

- 无符号数 a<b
  - a-b
  - 如果a<b，不够减，借位，CF=1
  - 否则（a>=b）,CF=0
  - 所以a<b <==> CF
- 进而可以推出：
  - a<=b <==> CF|ZF
  - a>b <==> ~(CF|ZF)=~CF&~ZF （摩根定律）
  - a>=b <==> ~CF

Accessing Conditional Codes

- 有符号数 a<b
  - A-b
  - 如果结果不溢出（OF=0),则一定得到一个负数（SF=1）
  - 如果结果溢出（OF=1），则一定得到一个正数（SF=0）
  - 所以a<b <==> OF^SF
- 进而可以推出：
  - a<=b <==> (OF^SF)|ZF
  - a>b <==> ~((OF^SF)|ZF)=~(OF^SF)&~ZF
  - a>=b <==> ~(OF^SF)

例：

				flags				interpretation	
	operand1	operand2	difference	CF	OF	SF	ZF	signed	unsigned
1	3B	3B	00	0	0	0	1	*op1=op2*	*op1=op2*
2	3B	15	26	0	0	0	0	*op1>op2*	*op1>op2*
3	15	3B	DA	1	0	1	0	*op1<op2*	op1<op2
4	F9	F6	03	0	0	0	0	*op1>op2*	*op1>op2*
5	F6	F9	FD	1	0	1	0	*op1<op2*	op1<op2
6	15	F6	1F	1	0	0	0	*op1>op2*	op1<op2
7	F6	15	E1	0	0	1	0	*op1<op2*	*op1>op2*
8	68	A5	C3	1	1	1	0	*op1>op2*	op1<op2
9	A5	68	3D	0	1	0	0	*op1<op2*	*op1>op2*
						of xor sf			cf

Accessing Conditional Codes

- The destination operand of **setxx** is either
  - one of the eight single-byte register elements or
  - a memory location where the single byte is to be stored
- To generate a 32-bit result
  - we must also clear the high-order 24 bits

Accessing Conditional Codes

Initially a is in %edx, b is in %eax

```
1 cmpl %eax, %edx      #compare a:b
2 setl       %al       #set low order by to 0 or 1
3 movzbl %al, %eax
                       #set remaining bytes of %eax to 0
```

主观题 10分

设置

- int comp (data_t a, data_t b) {
  - return a COMP b;
- }
- 64位机环境下，对于下面每个汇编指令序列，确定哪种数据类型data_t和比较操作COMP会导致编译器产生如下代码：
  - A. cmpl %esi, %edi; setl %al
  - B. cmpw %si, %di; setge %al
  - C. cmpb %sil, %dil; setbe %al
  - D. cmpq %rsi, %rdi; setne %al

作答

练习答案

- int comp (data_t a, data_t b) {
  - return a COMP b;
- }
- 64位机环境下，对于下面每个汇编指令序列，确定哪种数据类型data_t和比较操作COMP会导致编译器产生如下代码：
  - A. cmpl %esi, %edi; setl %al int, <
  - B. cmpw %si, %di; setge %al short, >=
  - C. cmpb %sil, %dil; setbe %al unsigned char, <=
  - D. cmpq %rsi, %rdi; setne %al long, unsigned long, 或指针，!=