Assembly I: Basic Operations

Jo, Heeseung

Moving Data (1)

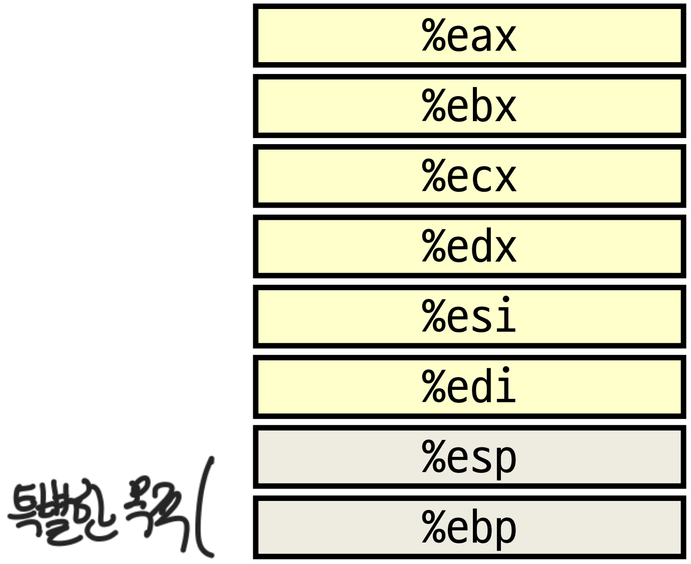
Moving data: movl source, dest

- Move 4-byte ("long") word
- Lots of these in typical code

Operand types

상수

- Immediate: constant integer data
  - Like C constant, but prefixed with '$'
  - e.g. $0x400, $-533
  - Encoded with 1, 2, or 4 bytes
- Register: one of 8 integer registers
  - But %esp and %ebp reserved for special use
  - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
  - Various "addressing modes"



Moving Data (2)

movl operand combinations

- Cannot do memory-memory transfers with single instruction

	Source	Destination		C Analog
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

Mem → Mem X
single instruction 안됨.

↓
pointer 변수 → 주소값을 가지고 있음.

Simple Addressing Modes

Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- e.g., movl (%ecx), %eax

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

- Register R specifies start of memory region
- Constant displacement D specifies offset
- e.g., movl 8(%ebp), %edx

↳ ebp 주소값에 8을 더해라.

leal ⇒ memory 접근 X, 레지스터에 값 세팅.

movl ⇒ 〃 O , 〃 .

Indexed Addressing Modes (1)

Most general form:

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

- D: constant "displacement": 1, 2, or 4 bytes
- Rb: Base register: any of 8 integer registers
- Ri: Index register: any, except for %esp & %ebp
- 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(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]+D]

- Useful to access arrays and structures

Indexed Addressing Modes (2)

Address computation example

%edx	0xf000
%ecx	0x0100

Expression	Computation	Address
0x8(%edx)	f000 + 8	f008
(%edx,%ecx)	0100 + f000	f100
(%edx,%ecx,4)	4*0100 + f000	f400
0x80(%ecx,%edx,2)	2*f000 + 80 + 0100	1e180

Ri (뒤) 레지스터 값에 곱하는 느낌

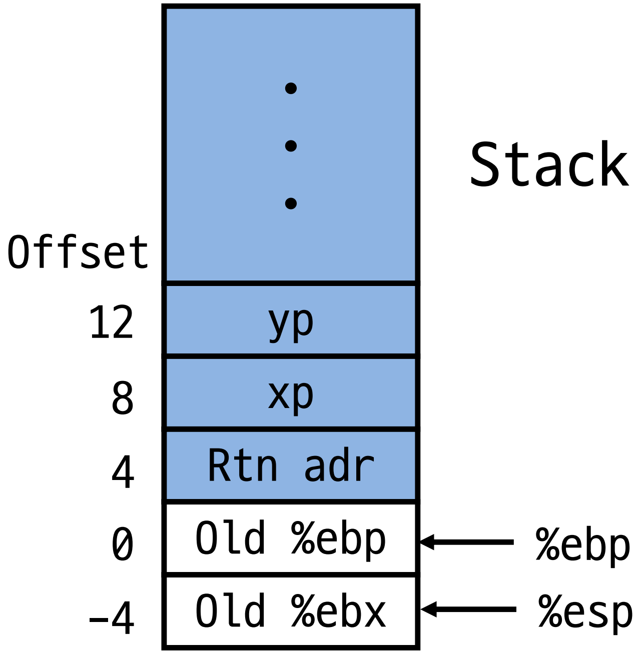
2 * f000) 1111 0000~

1 << 1|1110 0000|000~

1 e 0 0

Swap Example

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



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Setup

Body

Finish

ex) ebp = 0x10

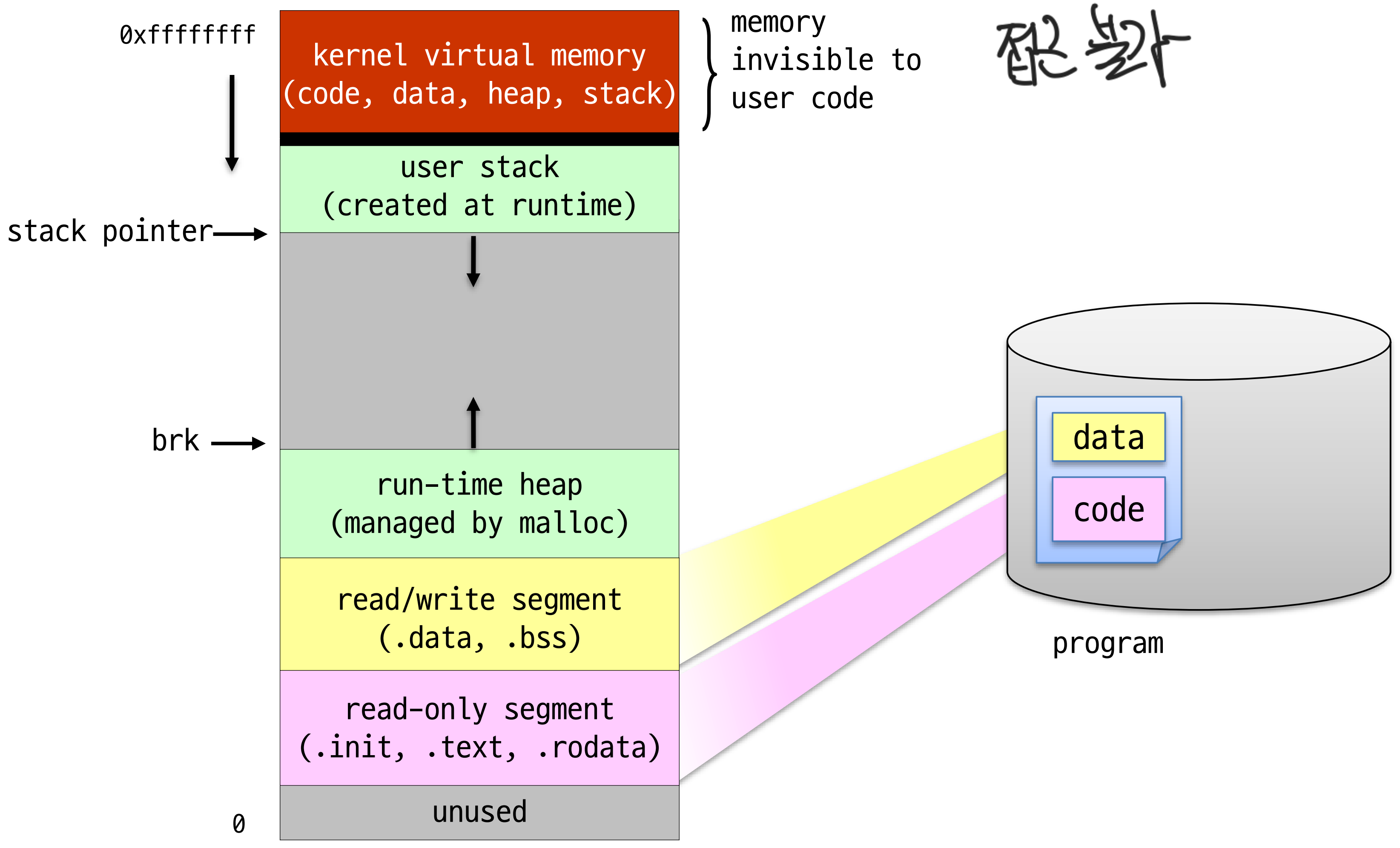
$10 \xrightarrow[16]{} 16$

12+16 = 28

☆

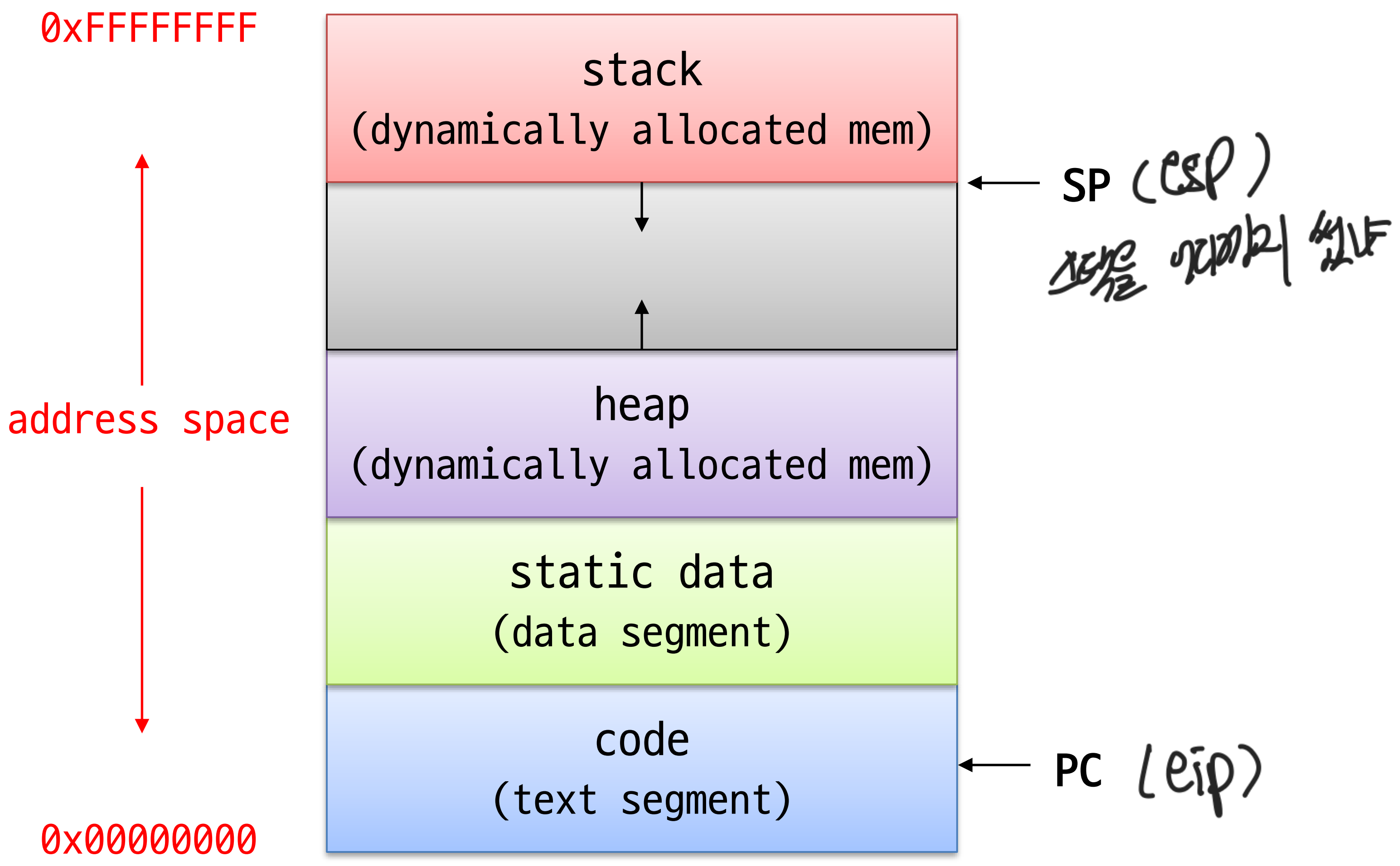
Process Address Space

Process in memory


0xffffffff
kernel virtual memory (code, data, heap, stack)
memory invisible to user code
user stack (created at runtime)
stack pointer
brk
run-time heap (managed by malloc)
read/write segment (.data, .bss)
read-only segment (.init, .text, .rodata)
unused
0
data
code
program

접근 불가

Process Address Space

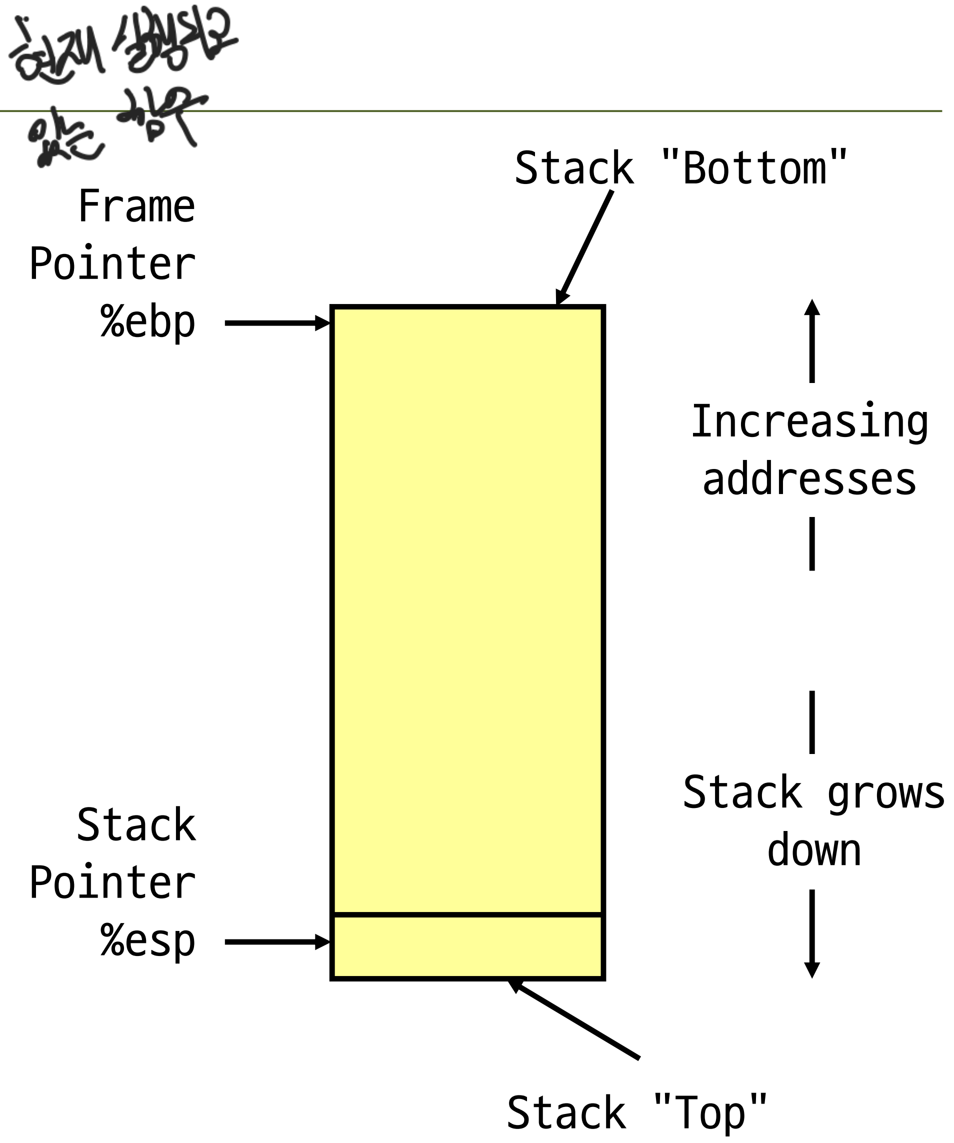


IA-32 Stack

Characteristics

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register %esp indicates lowest stack address
  - address of top element

- Stack pointer %esp indicates stack top
- Frame pointer %ebp indicates start of current frame

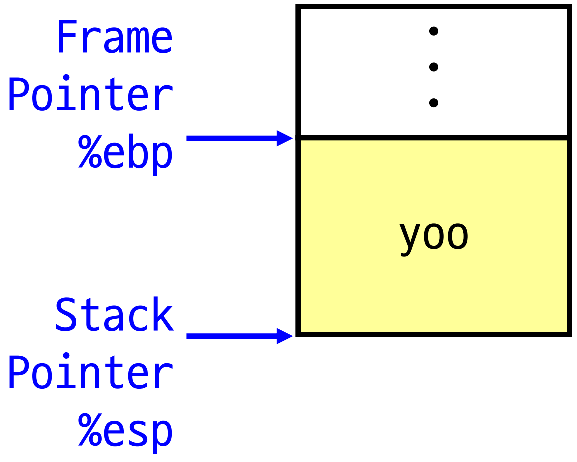


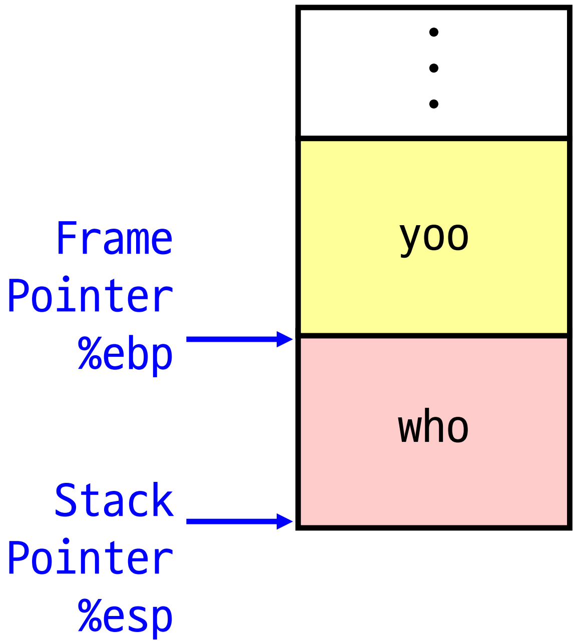
Stack Frames

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

Call Chain

$$\begin{tikzcd} \text{yoo} \arrow[d] \\ \text{who} \end{tikzcd}$$





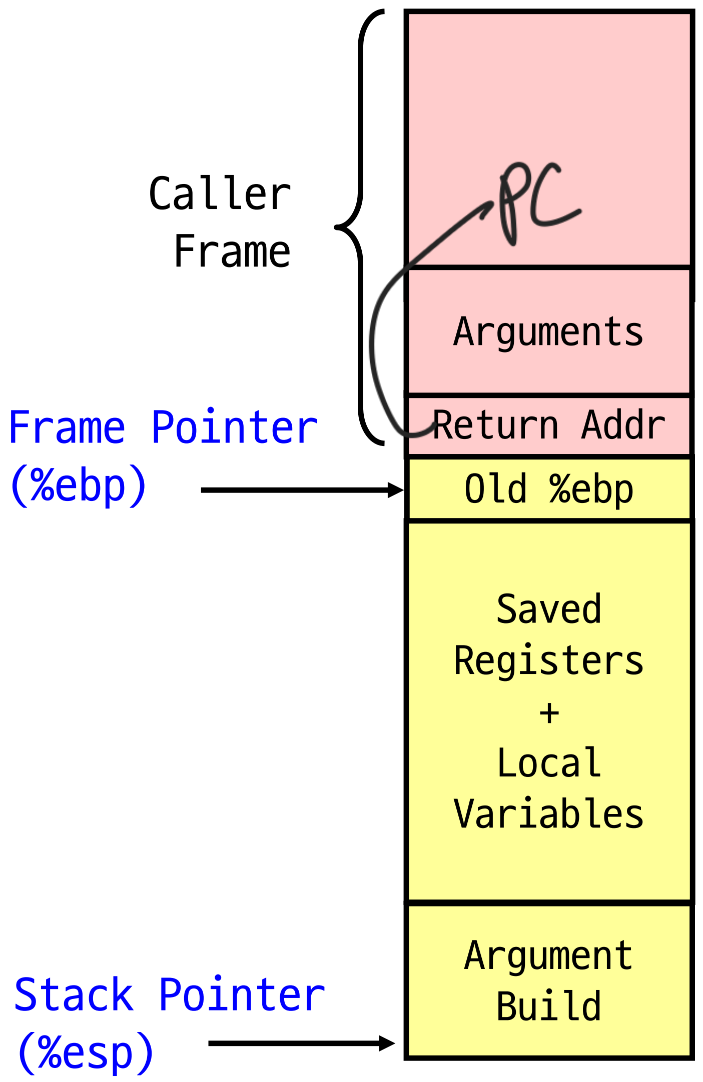
IA-32/Linux Stack Frame

Caller stack frame

- Ex) swap(&zip1, &zip2);
- Arguments to call
- Return address
  - Pushed by call instruction

Current stack frame ("Top" to Bottom)

- Old frame pointer



Understanding Swap (0)

```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

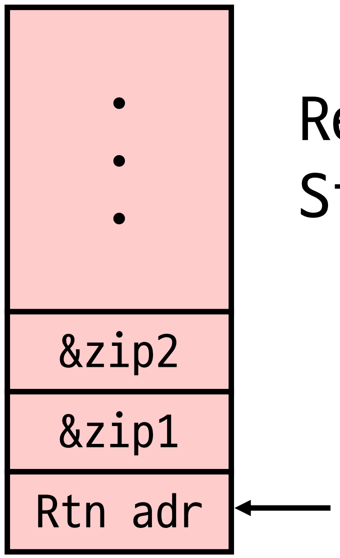
생성

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

Calling swap from call_swap

```
call_swap:
    • • •
    pushl $zip2      # Global Var
    pushl $zip1      # Global Var
    call swap
    • • •
```

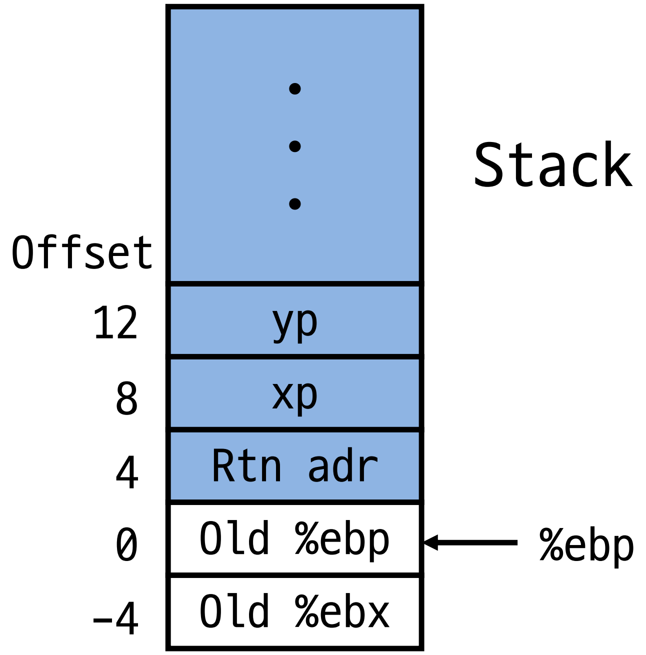
인자 쌓아넣고
esp 줄이고
이런 방식



Resulting Stack

Understanding Swap (1)

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

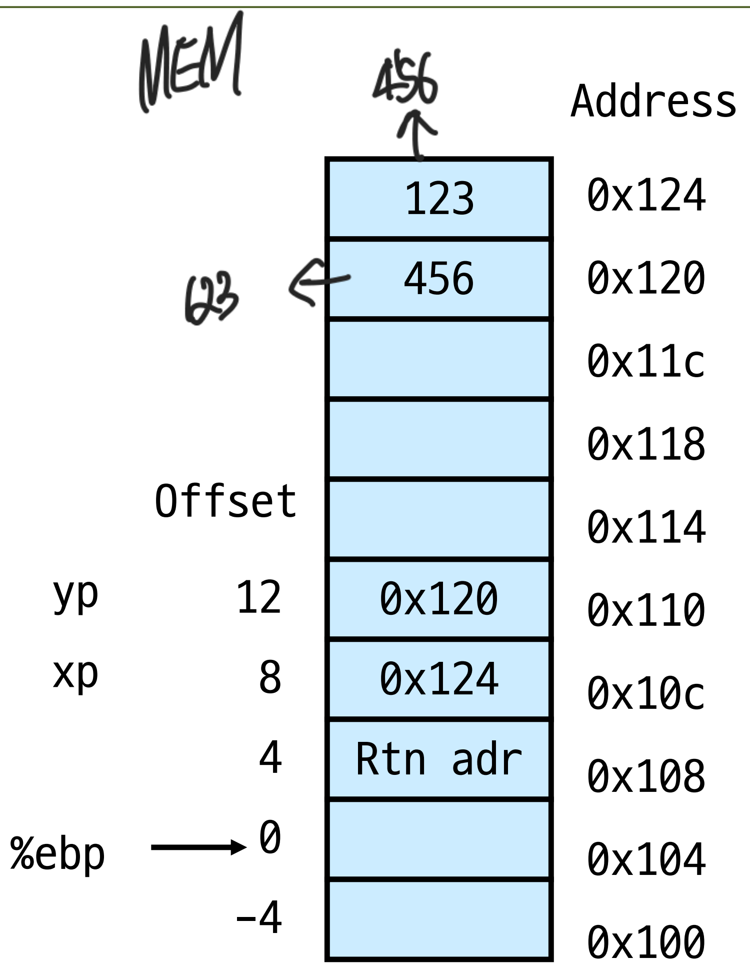


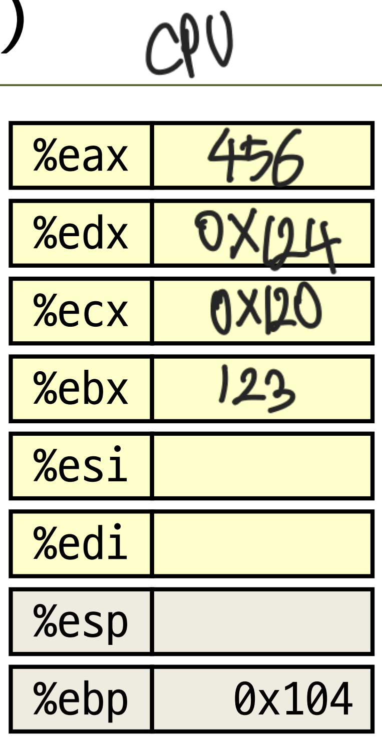
Register Allocation
(By compiler)

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

Understanding Swap (2)





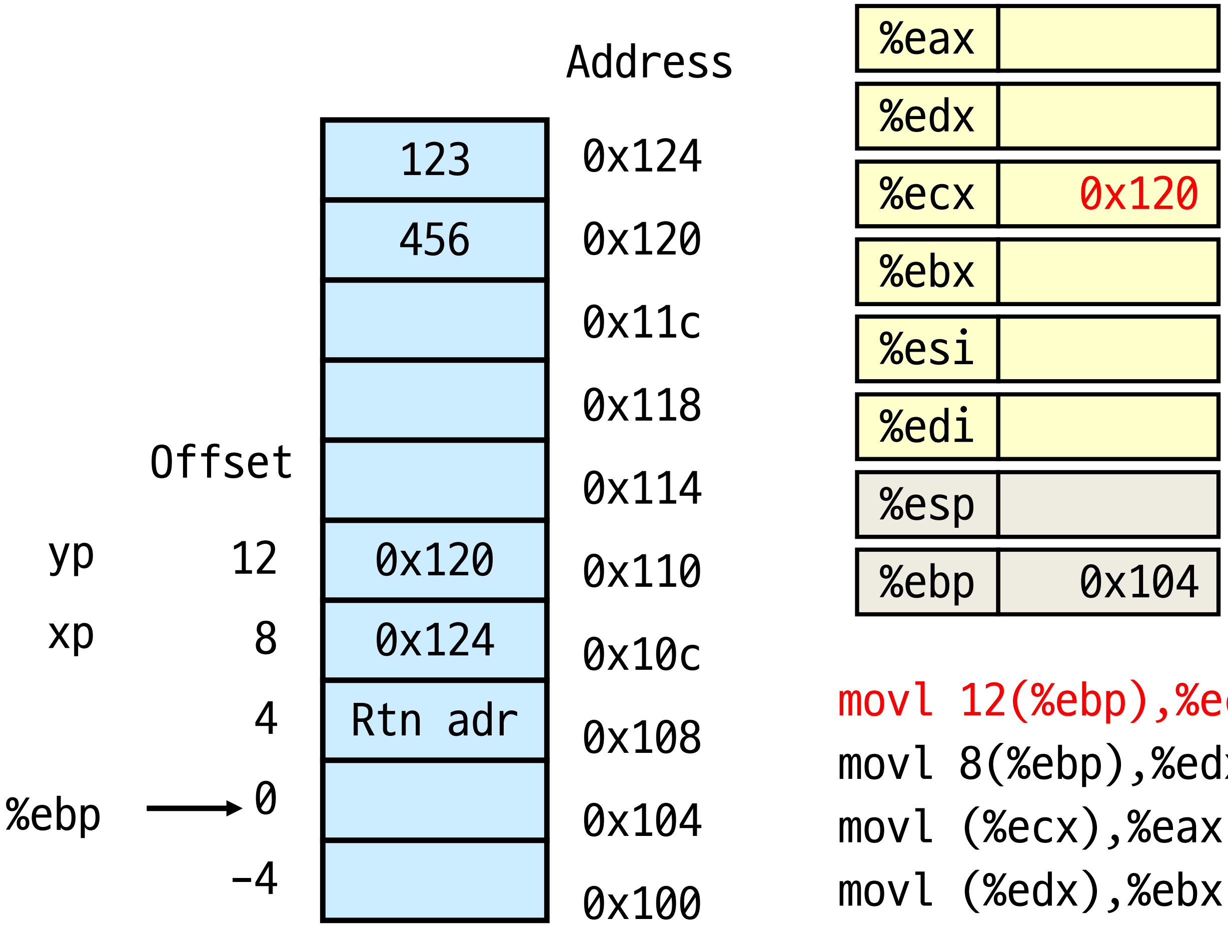
Body

Register Allocation
(By compiler)

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

Understanding Swap (3)

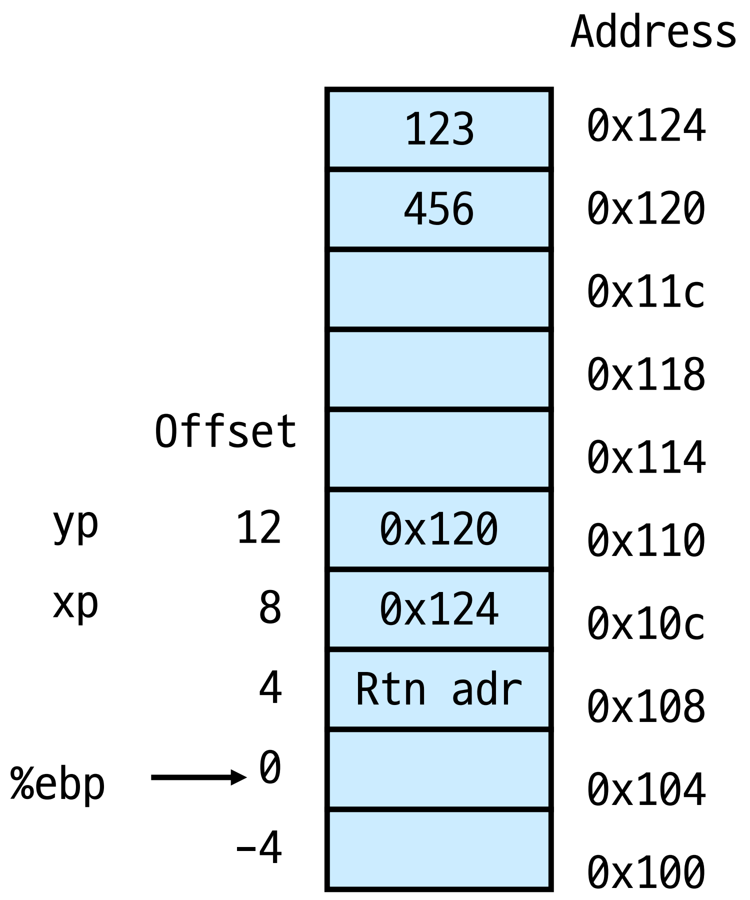


Register Allocation
(By compiler)

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

Understanding Swap (4)



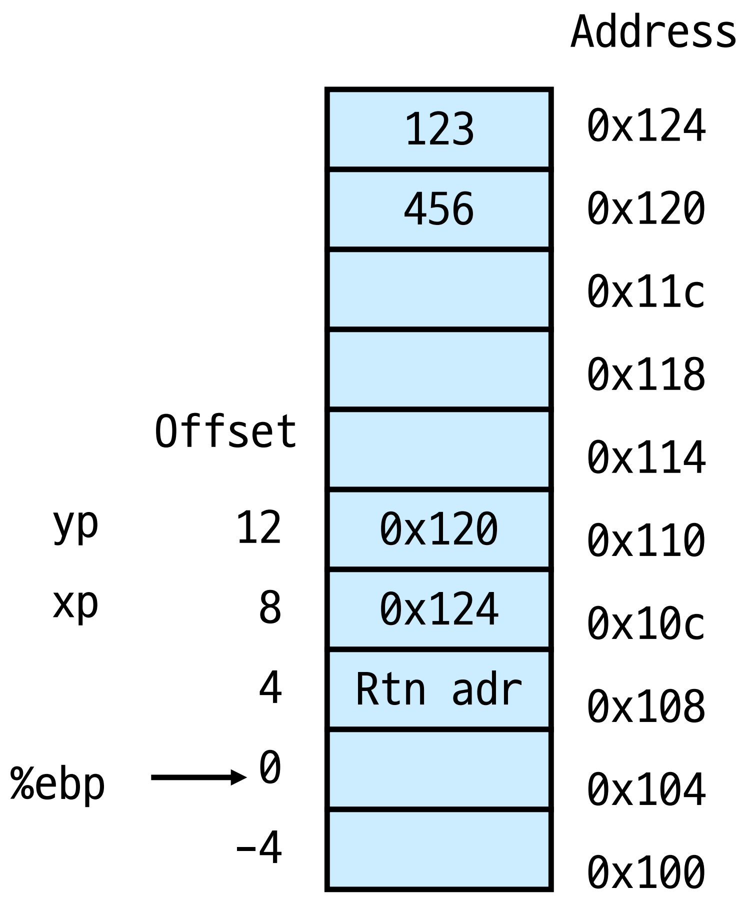
%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Register Allocation
(By compiler)

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

Understanding Swap (5)



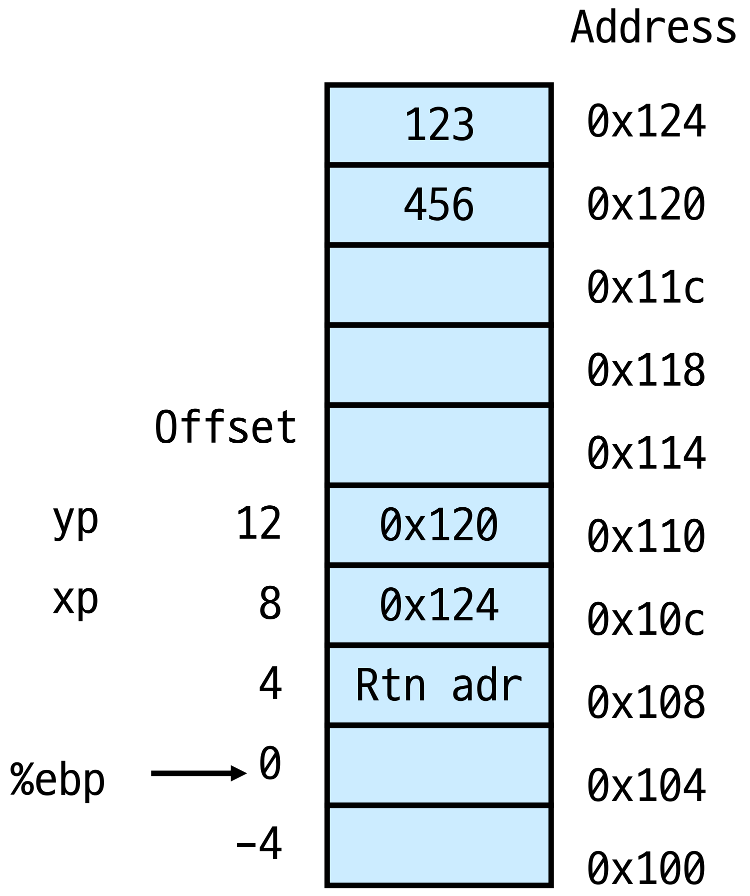
%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

Register Allocation
(By compiler)

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx        # ecx = yp
movl 8(%ebp),%edx         # edx = xp
movl (%ecx),%eax          # eax = *yp (t1)
movl (%edx),%ebx          # ebx = *xp (t0)
movl %eax,(%edx)          # *xp = eax
movl %ebx,(%ecx)          # *yp = ebx
```

Understanding Swap (6)



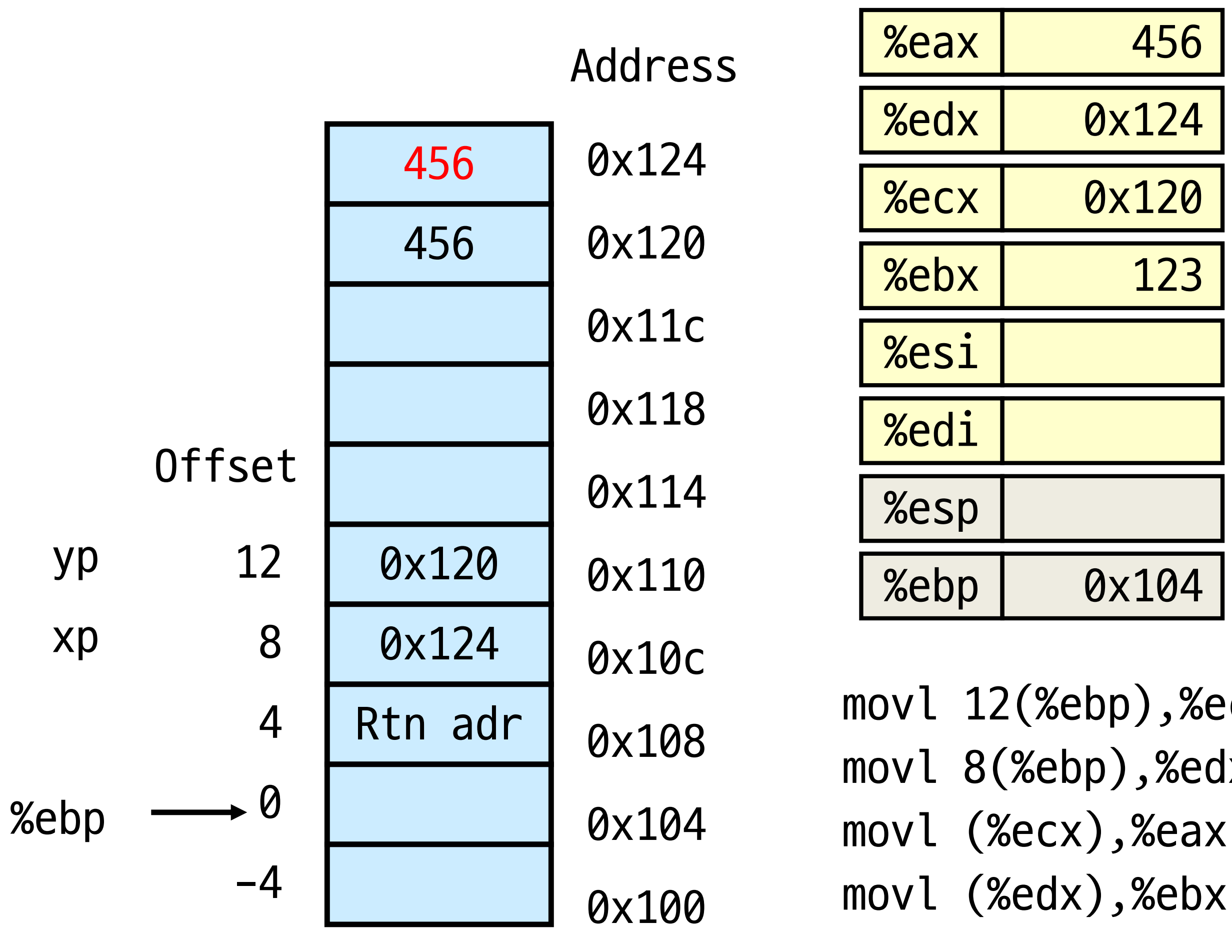
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

Register Allocation
(By compiler)

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

Understanding Swap (7)

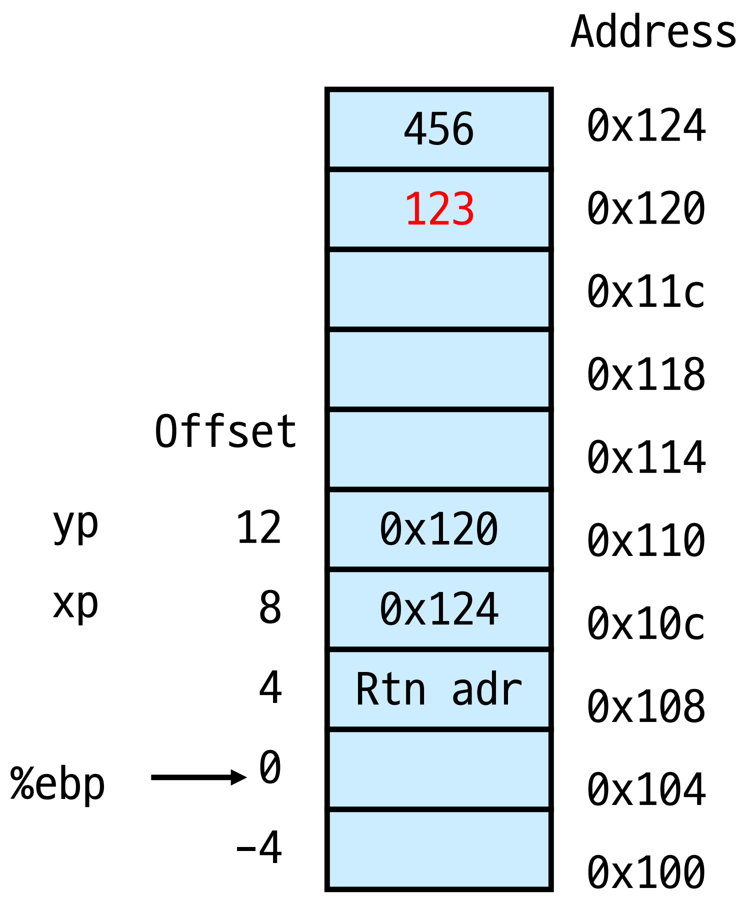


Register Allocation
(By compiler)

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

Understanding Swap (8)



%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

Register Allocation
(By compiler)

Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx      # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

Arithmetic/Logical Ops. (1)

Two operands instructions

• addl	Src, Dest	Dest = Dest + Src
• subl	Src, Dest	Dest = Dest - Src
• mull	Src, Dest	Dest = Dest * Src (unsigned)
• imull	Src, Dest	Dest = Dest * Src (signed)
• sall	Src, Dest	Dest = Dest << Src (= shll)
• sarl	Src, Dest	Dest = Dest >> Src (Arith.)
• 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

Arithmetic/Logical Ops. (2)

One operand instructions

incl	Dest	Dest = Dest + 1
decl	Dest	Dest = Dest - 1
negl	Dest	Dest = -Dest
notl	Dest	Dest = ~Dest

Address Computation

leal *Src*, *Dest*

- *Src* is address mode expression
- Set ***Dest*** to address denoted by expression

$x+x*2$ $\curvearrowright$

```
leal (%edx,%edx,2),%edx                x = 3 * x;
```

$3x$

```
movl (%edx,%edx,2),%edx
```

leal은 바로 뒤 레지스터에 집어넣음.

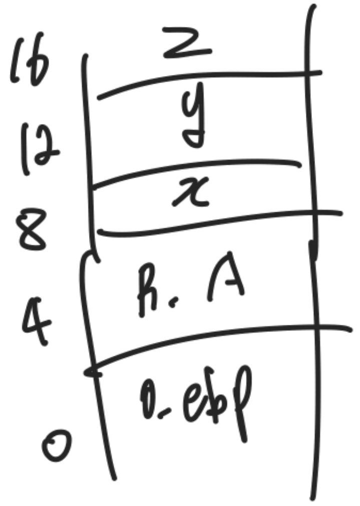
Uses

- Computing address without doing 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: arith (1)

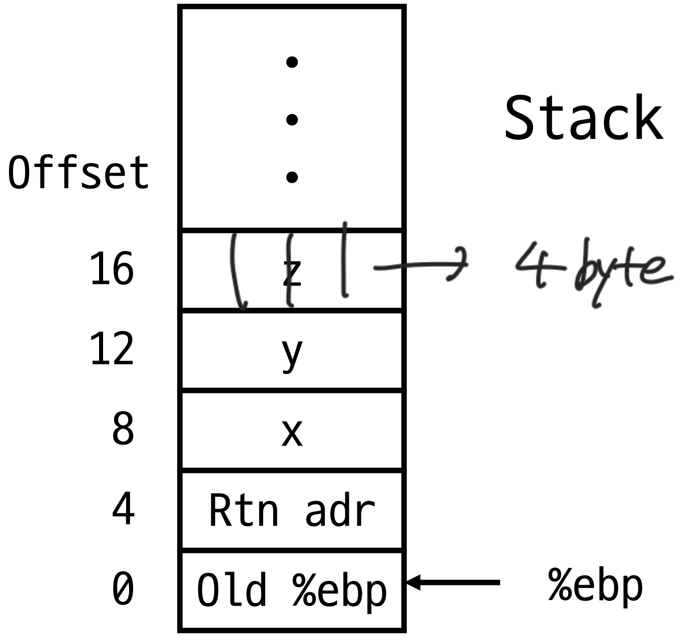
```c
int arith (int x, int y, int z)
{
  int t1 = x + y;
  int t2 = z + t1;
  int t3 = x + 4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;

  return rval;
}
```

```
arith:
    pushl %ebp                      } Set Up
    movl %esp,%ebp

    movl 8(%ebp),%eax       x
    movl 12(%ebp),%edx      y
    leal (%edx,%eax),%ecx   x+y
    leal (%edx,%edx,2),%edx   3y<<4 =48
    sall $4,%edx                    } Body
    addl 16(%ebp),%ecx      z
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

    movl %ebp,%esp
    popl %ebp                       } Finish
    ret
```



Example: arith (2)

```
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;
}
```

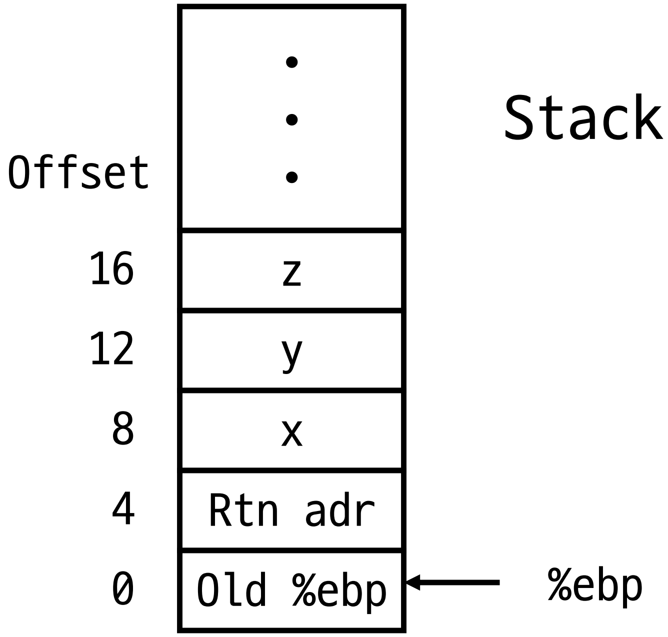


```
movl 8(%ebp),%eax           # eax = x
movl 12(%ebp),%edx          # edx = y
leal (%edx,%eax),%ecx       # ecx = x + y  (t1)
leal (%edx,%edx,2),%edx     # edx = 3 * y
sall $4,%edx                # edx = 48 * y (t4)
addl 16(%ebp),%ecx          # ecx = z + t1 (t2)
leal 4(%edx,%eax),%eax      # eax = x + t4 + 4 (t5)
imull %ecx,%eax             # eax = t2 * t5 (rval)
```

When a function ends, the value of %eax is the return value

Example: arith2

```
int arith2 (int x, int y, int z)
{
  int t1 = x + y + z;
  int t2 = x * y;
  int t3 = x + 4;
  int t4 = 16 * y;
  int rval = t2 * t4;
  return rval;
}
```



Sol) ① 1,2 line edx와 eax 교체
② 3 line edx 와 eax 교체

```
movl 8(%ebp),%edx
movl 12(%ebp),%eax
imull %edx,%eax
sall $4,%eax
imull %edx,%eax
```

return 값이 void 함수 등에서 없는 것이 아니라, 내릴거냐 이거임.

마지막 eax 레지스터 값을 return 값으로 사용하도록 되어있다. → 최종적 함수의 output이 eax 레지스터에 저장된다는 느낌

What's wrong?

Example: logical

```c
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;
}
```

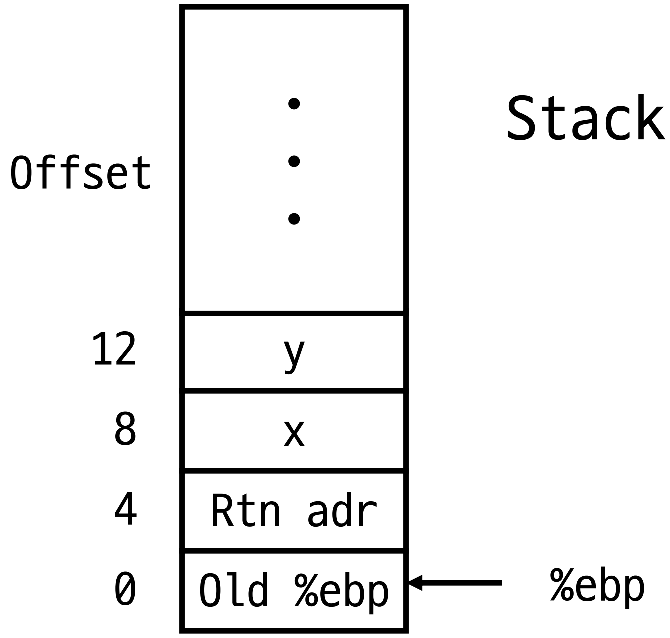
```
logical:
    pushl %ebp              } Set Up
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax      } Body
    sarl $17,%eax
    andl $8185,%eax
```

$2^{13}-7$

```
    movl %ebp,%esp
    popl %ebp               } Finish
    ret
```

Example: logical

```
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;
}
```



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

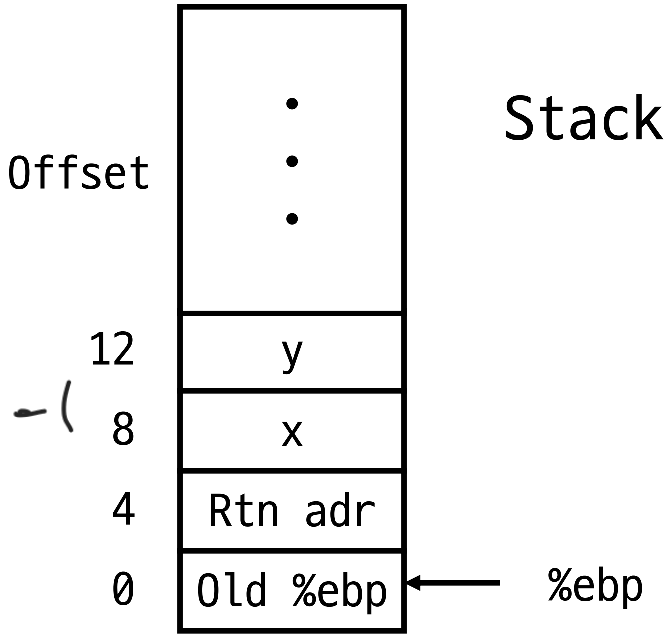
```
movl 8(%ebp),%eax          # eax = x
xorl 12(%ebp),%eax         # eax = x ^ y   (t1)
sarl $17,%eax              # eax = t1 >> 17    (t2)
andl $8185,%eax            # eax = t2 & 8185
```

$1 \ll n \Rightarrow 1 \to 2^n$ 이 됨

Example: andor

```
int andor (int x, int y)
{
  int t2 = x & y;
  int t3 = 0xffffffff;
  int rval = t3 ¦ t2;
  return rval;
}
```

↳ t2가 뭐든 상관 rval = -1 한 줄로 가능!



```
movl 12(%ebp),%eax          # eax = y
movl 8(%ebp),%edx           # edx = x
andl %edx,%eax              # eax = x & y (t2)
movl $-1,%edx               # edx = 0xffffffff (t3)
orl %edx,%eax               # eax = t2 ¦ t3
```

Make it short! ⇒ movl 12(%ebp), %eax ⇒ movl $-1, %eax
andl 8(%ebp), %eax
orl $-1, %eax

CISC Properties

CISC (Complex Instruction Set Computer)

- Instruction can reference different operand types
  - Immediate, register, memory
- Arithmetic operations can read/write memory
- Memory reference can involve complex computation
  - D(Rb, Ri, S) -> Rb + S*Ri + D
  - Useful for arithmetic expressions, too
- Instructions can have varying lengths
  - IA-32 instructions can range from 1 to 15 bytes

Mem 단기기억장치에서 Reg로 Load 하고 ~
연산하고 ~ Mem에 Store 하고 그런 flow 임

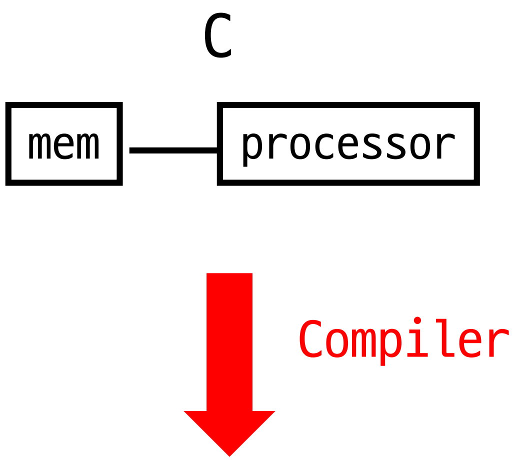
Summary (1)

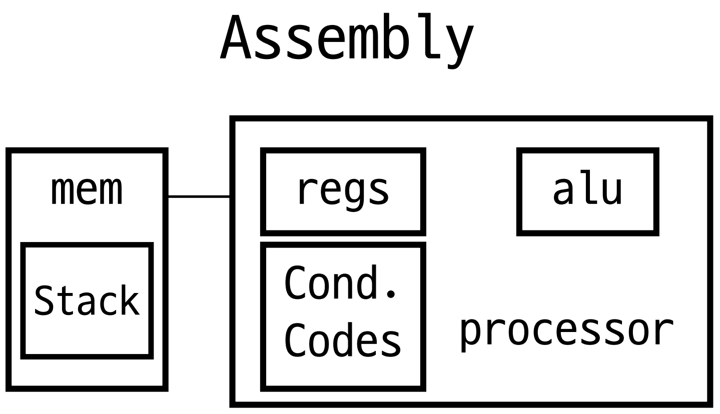
Machine level programming

- Assembly code is textual form of binary object code
- Low-level representation of program
  - Explicit manipulation of registers
  - Simple and explicit instructions
  - Minimal concept of data types
  - Many C control constructs must be implemented with multiple instructions

Summary (2)

Machine Models





Data

1) char
2) int, float
3) double
4) struct, array
5) pointer

1) 1 byte
2) 4 byte
3) 8 byte
4) contiguous byte allocation
5) address of initial byte

Control

1) loops
2) conditionals
3) switch
4) Proc. call
5) Proc. return

1) branch/jump
2) call
3) ret

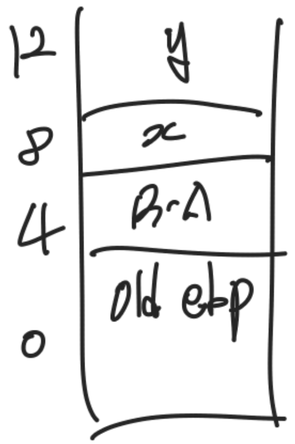
Exercise

ASM -> C

```
doit:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%edx),%eax
    movl %eax,(%edx)

    movl %ebp,%esp
    popl %ebp
    ret
```

```
doit (int x, int y)
{
    int rval = *x;
    int *x = rval;
}
```

실제에서 return 되는 고려하지 않는다!
주어진 어셈 작성

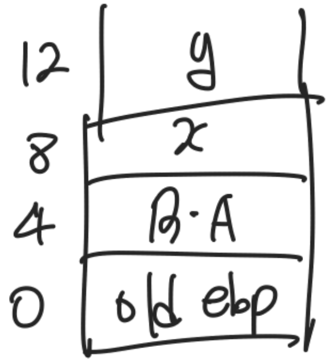


Exercise

C -> ASM

```
int doit (int x, int y)
{
  int rval;
  int t1 = x + y;
  t1 = t1 * 4;
  return rval;
}
```

```
doit:
    pushl  %ebp              ) 안써도 됨
    movl   %esp, %ebp        )

    movl   12(%ebp), %eax
    movl   8(%ebp), %edx

    leal   (%edx, %eax), %ecx
    sall   $2, %ecx
    movl   %ecx, %eax

    movl   %ebp, %esp        ) 안써도 됨
    popl   %ebp              )
    ret
```



12
y
8
x
4
R·A
0
old ebp