

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

- **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"

<code>%eax</code>
<code>%ebx</code>
<code>%ecx</code>
<code>%edx</code>
<code>%esi</code>
<code>%edi</code>
<code>%esp</code>
<code>%ebp</code>

Moving Data (2)

movl operand combinations

- Cannot do memory-memory transfers with single instruction

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

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`

Indexed Addressing Modes (1)

Most general form:

$$D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri] + D]$$

- 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) $\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]]$
- D(Rb,Ri) $\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D]$
- (Rb,Ri,S) $\text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri]]$
- D(Rb,Ri,S) $\text{Mem}[\text{Reg}[Rb] + S * \text{Reg}[Ri] + D]$
- Useful to access arrays and structures

Indexed Addressing Modes (2)

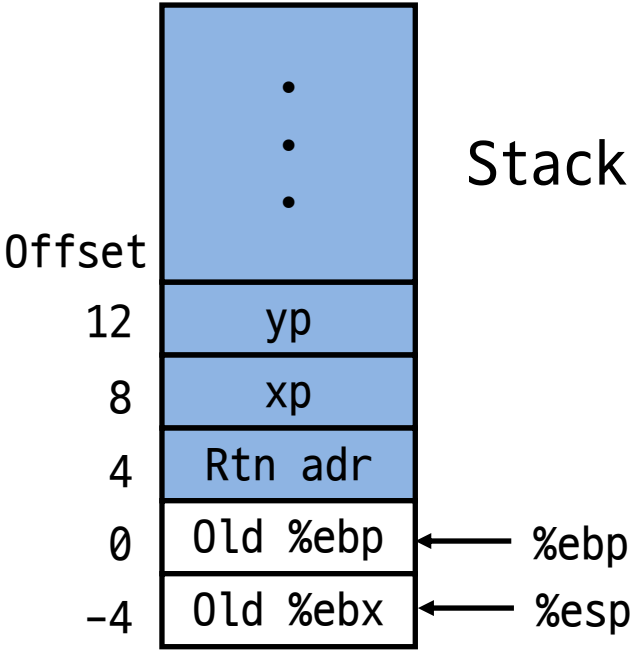
Address computation example

%edx	0xf000
%ecx	0x0100

Expression	Computation	Address
0x8(%edx)		
(%edx,%ecx)		
(%edx,%ecx,4)		
0x80(%ecx,%edx,4)		

Swap Example

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



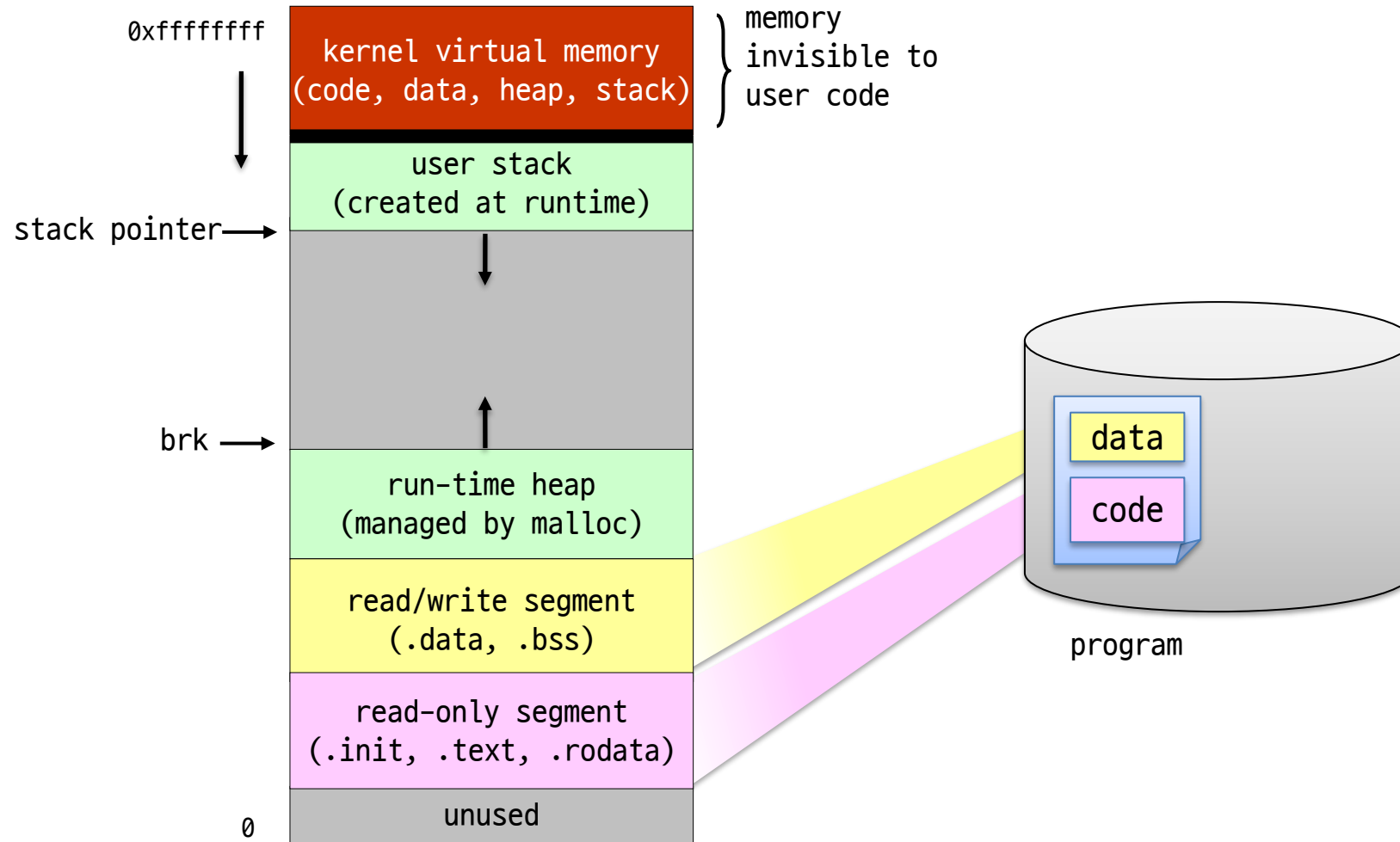
```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    } Setup

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

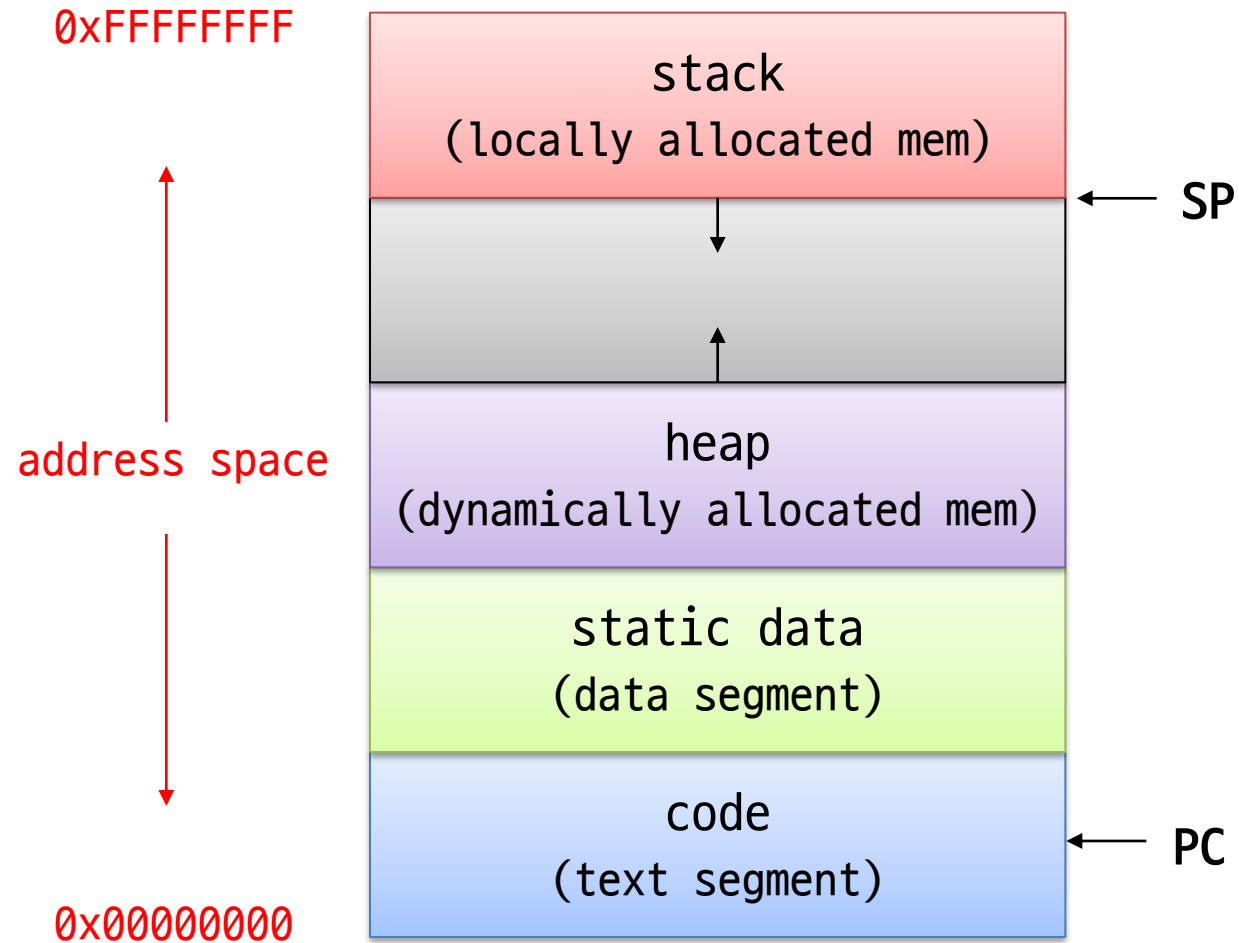
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
    } Finish
```

Process Address Space

Process in memory



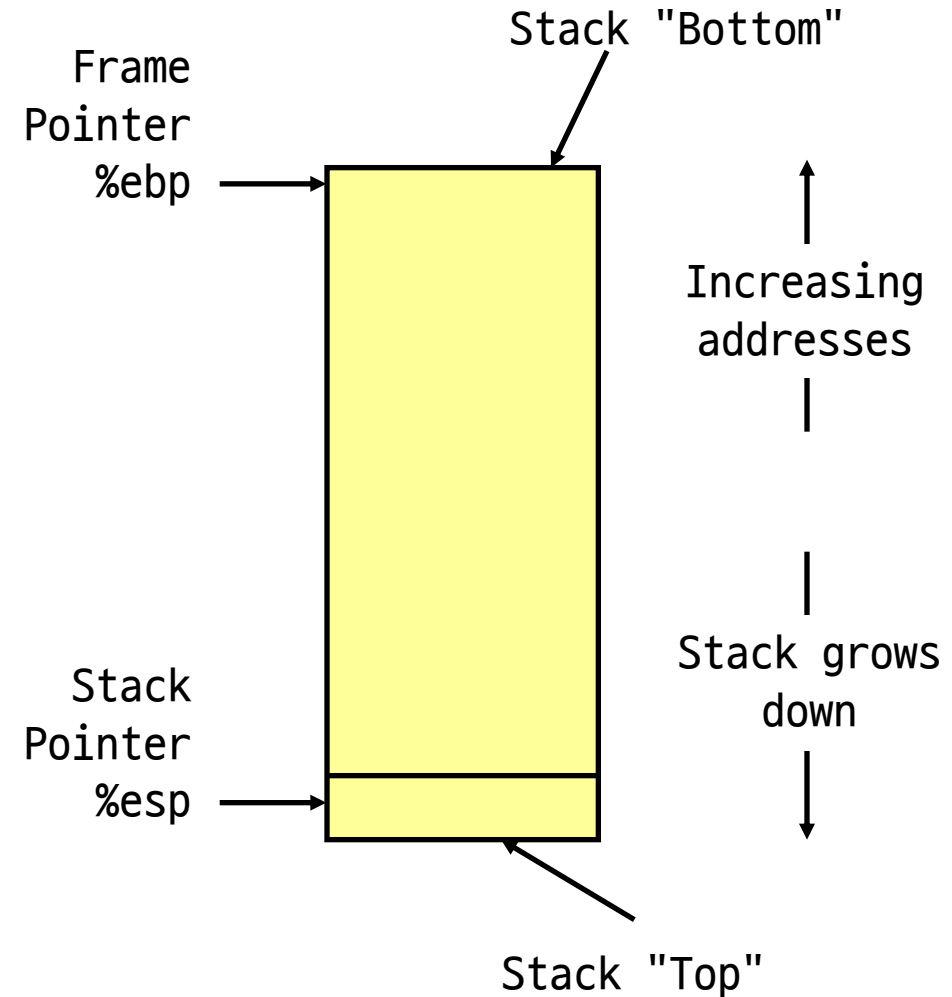
Process Address Space



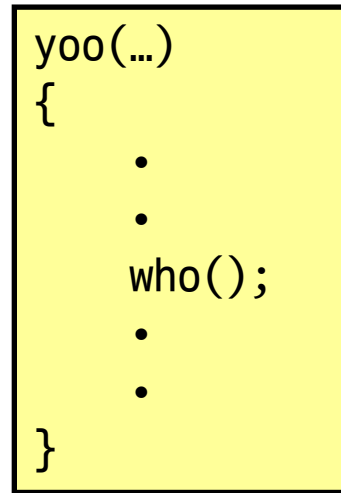
IA-32 Stack

Characteristics

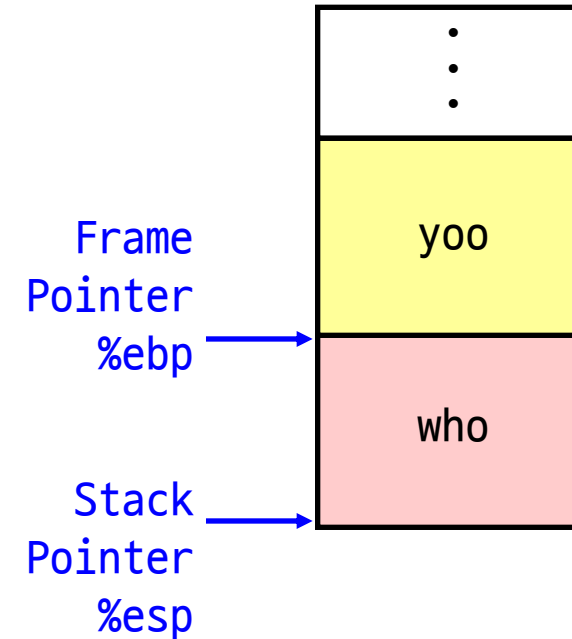
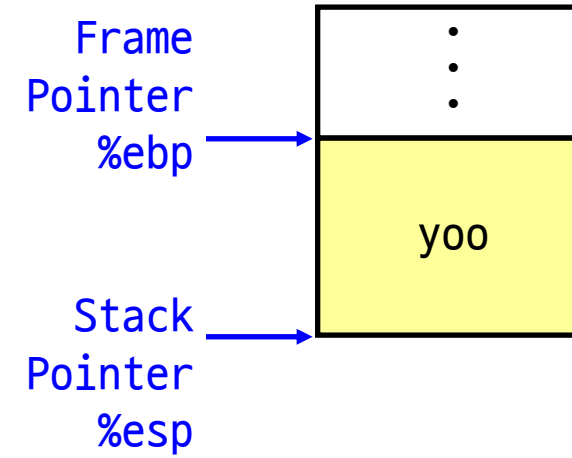
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
 - address of top element
- Stack pointer `%esp` indicates stack top
- Frame pointer `%ebp` indicates start of current frame



Stack Frames



Call Chain



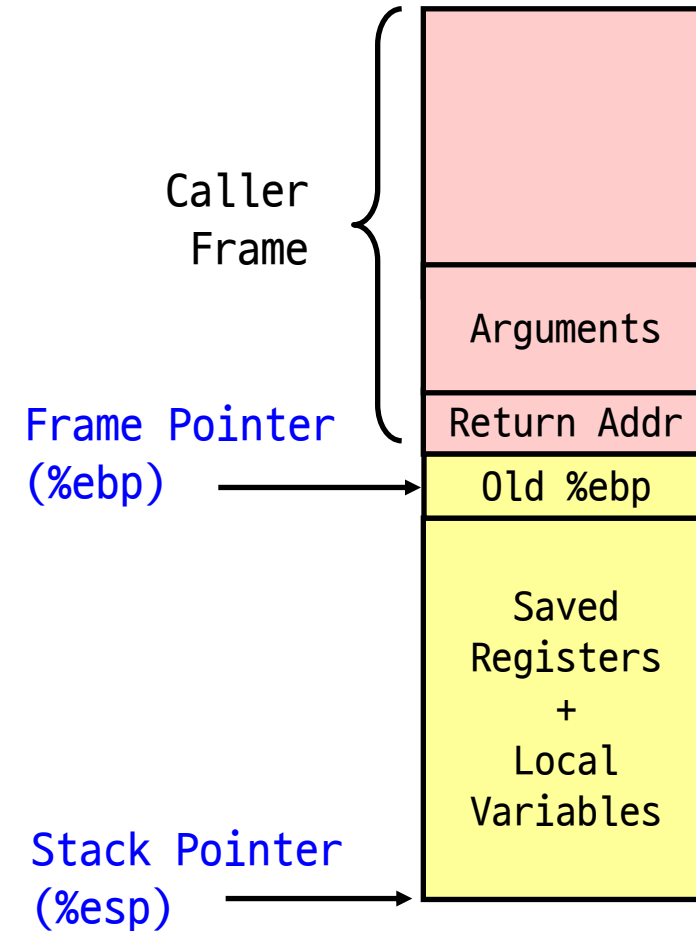
IA-32/Linux Stack Frame

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)

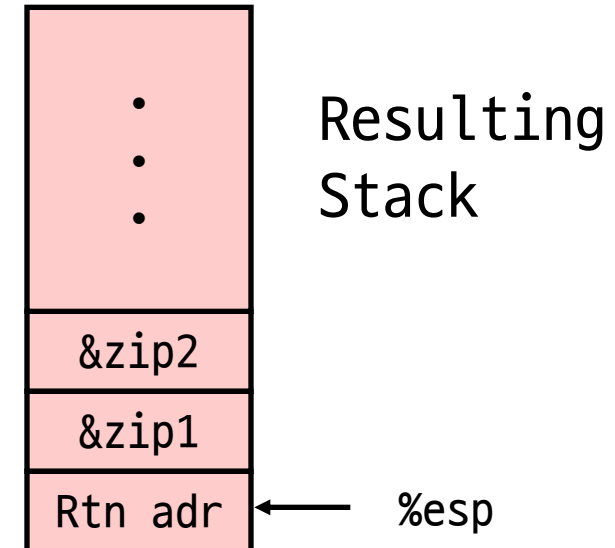
```
int zip1 = 15213;
int zip2 = 91125;

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

```
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
    . . .
```



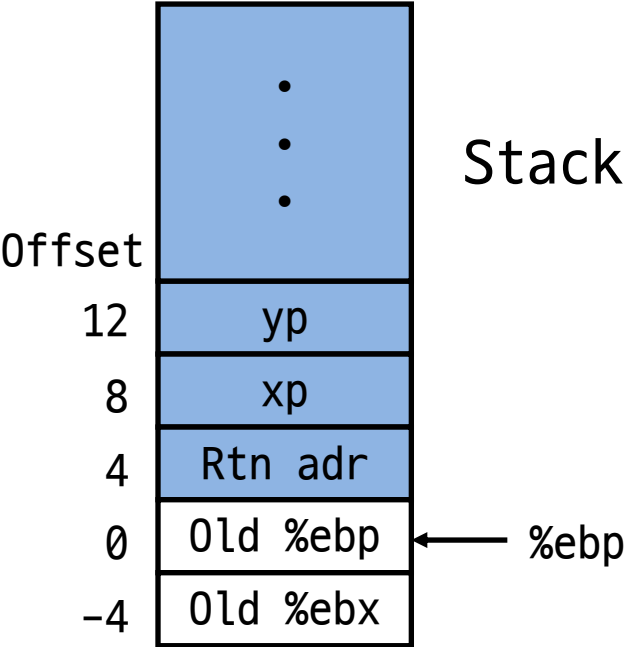
Understanding Swap (1)

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

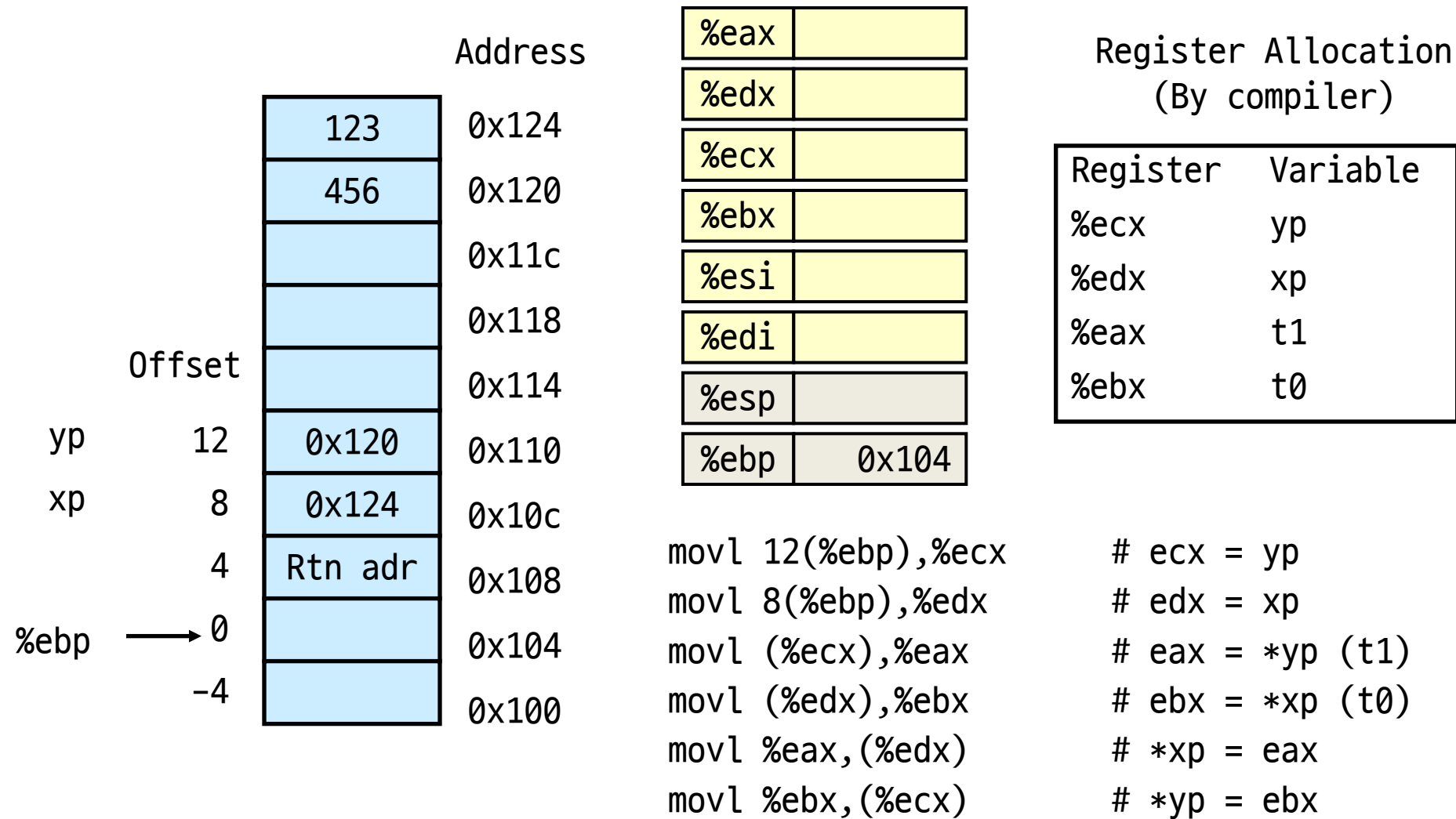
Register Allocation
(By compiler)

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

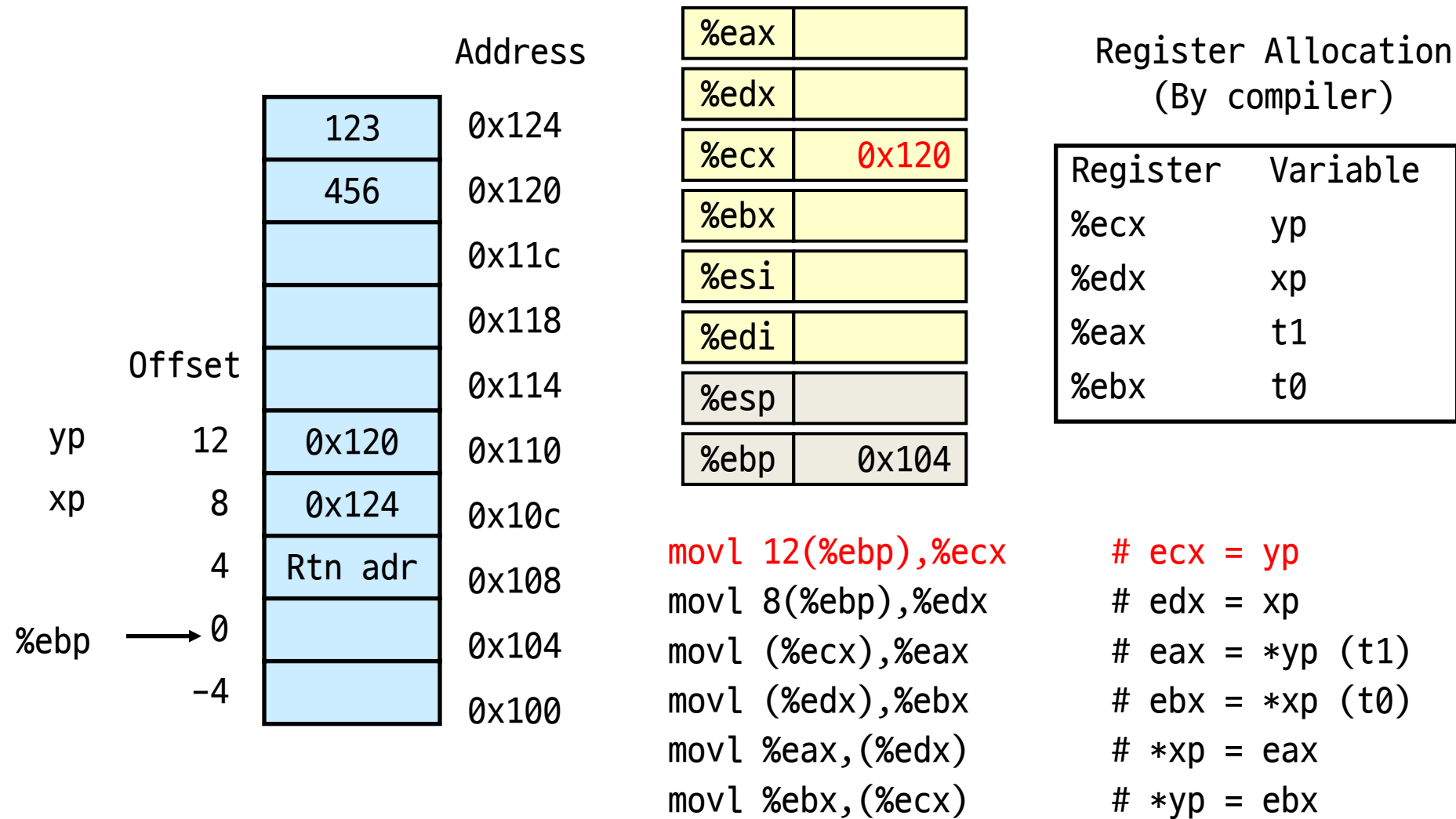
```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax, (%edx)     # *xp = eax
movl %ebx, (%ecx)     # *yp = ebx
```



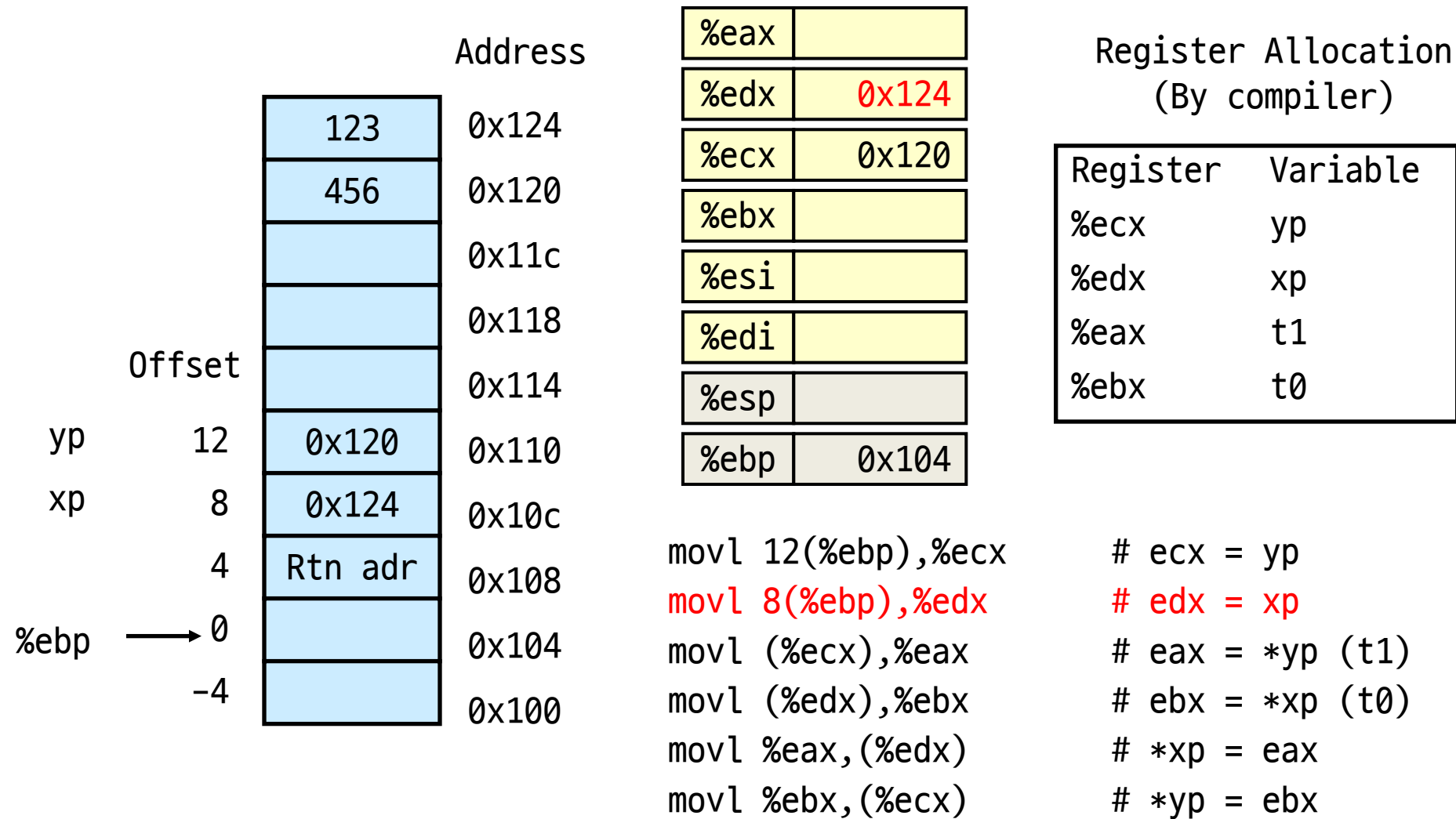
Understanding Swap (2)



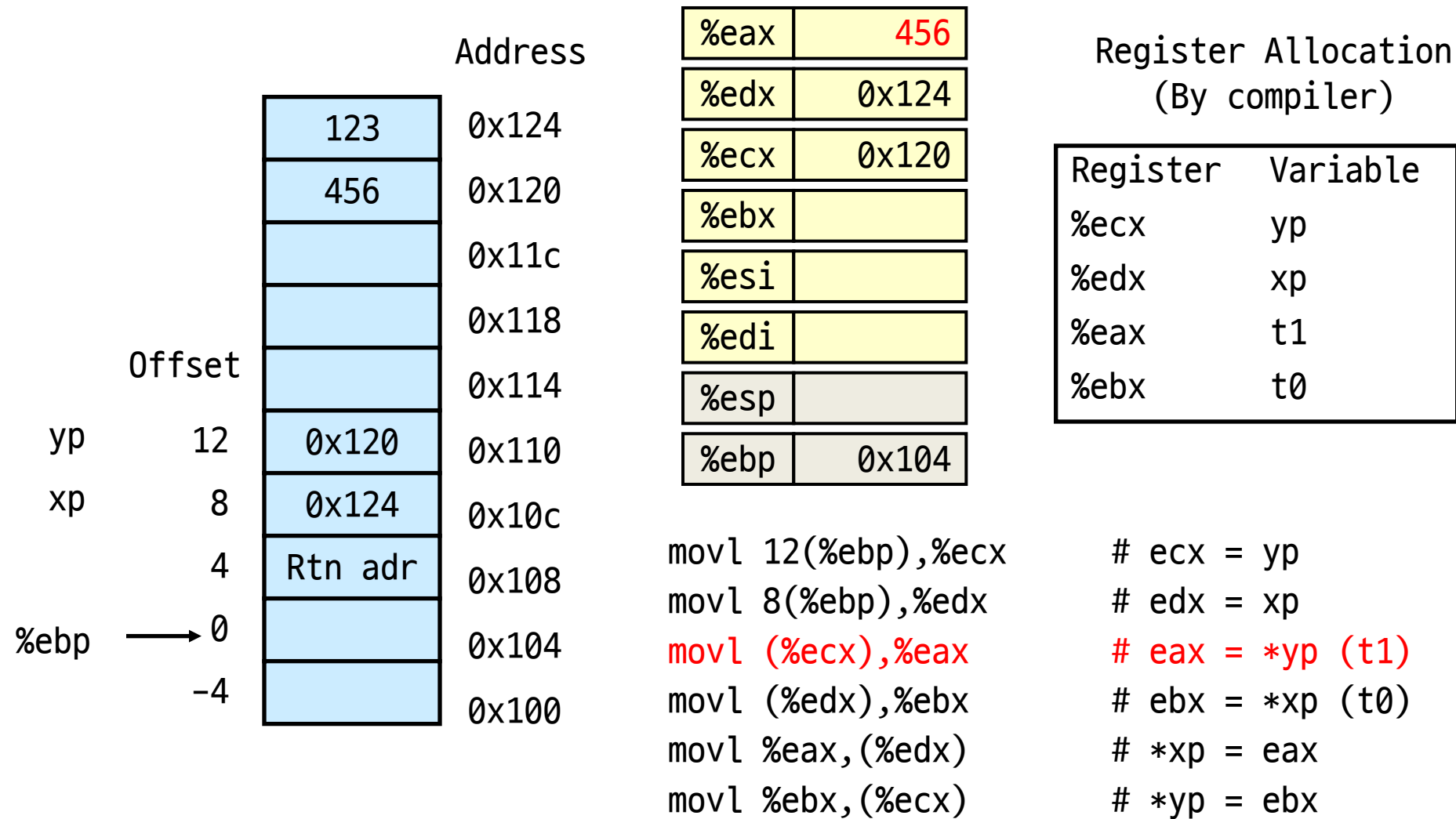
Understanding Swap (3)



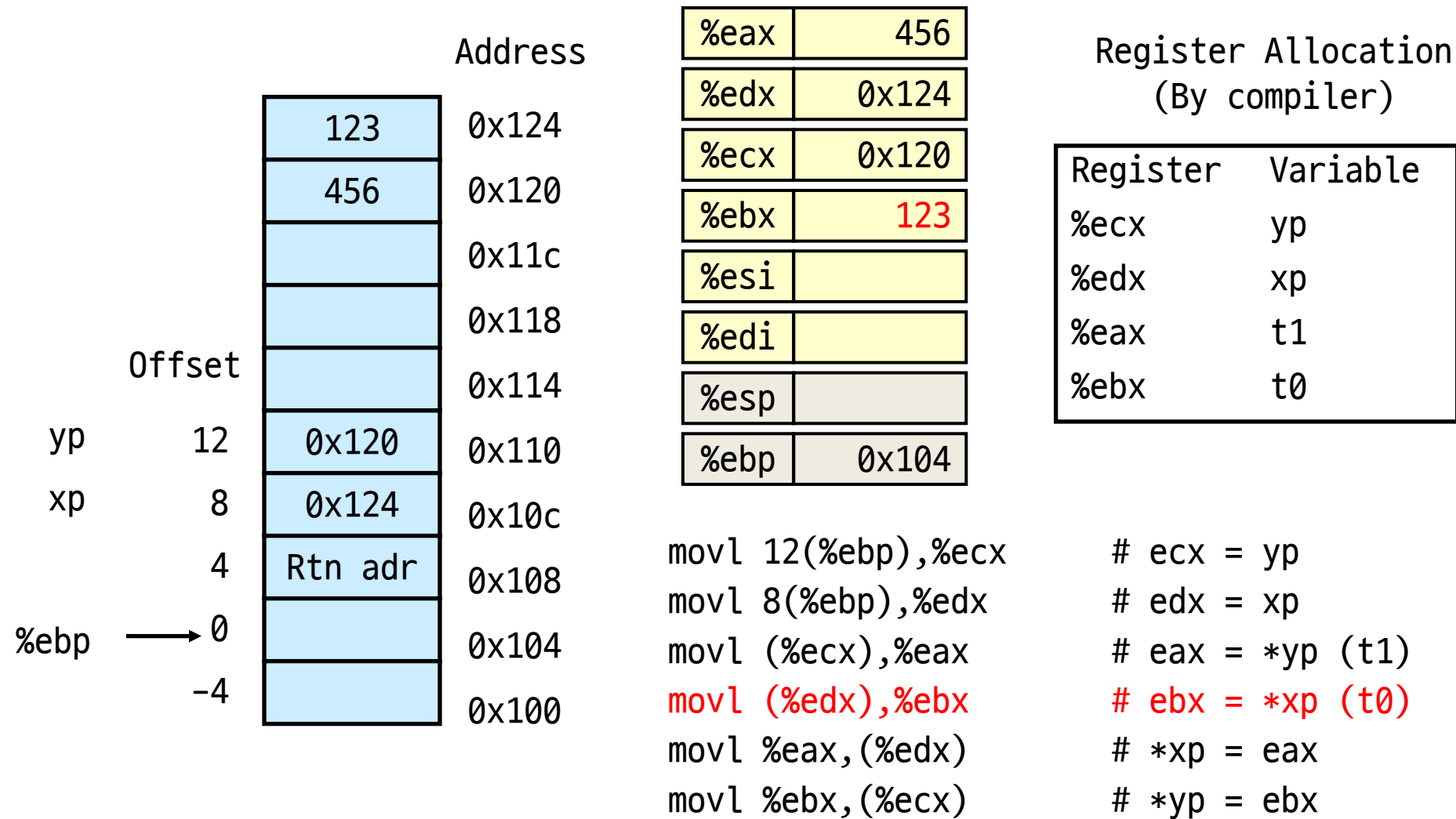
Understanding Swap (4)



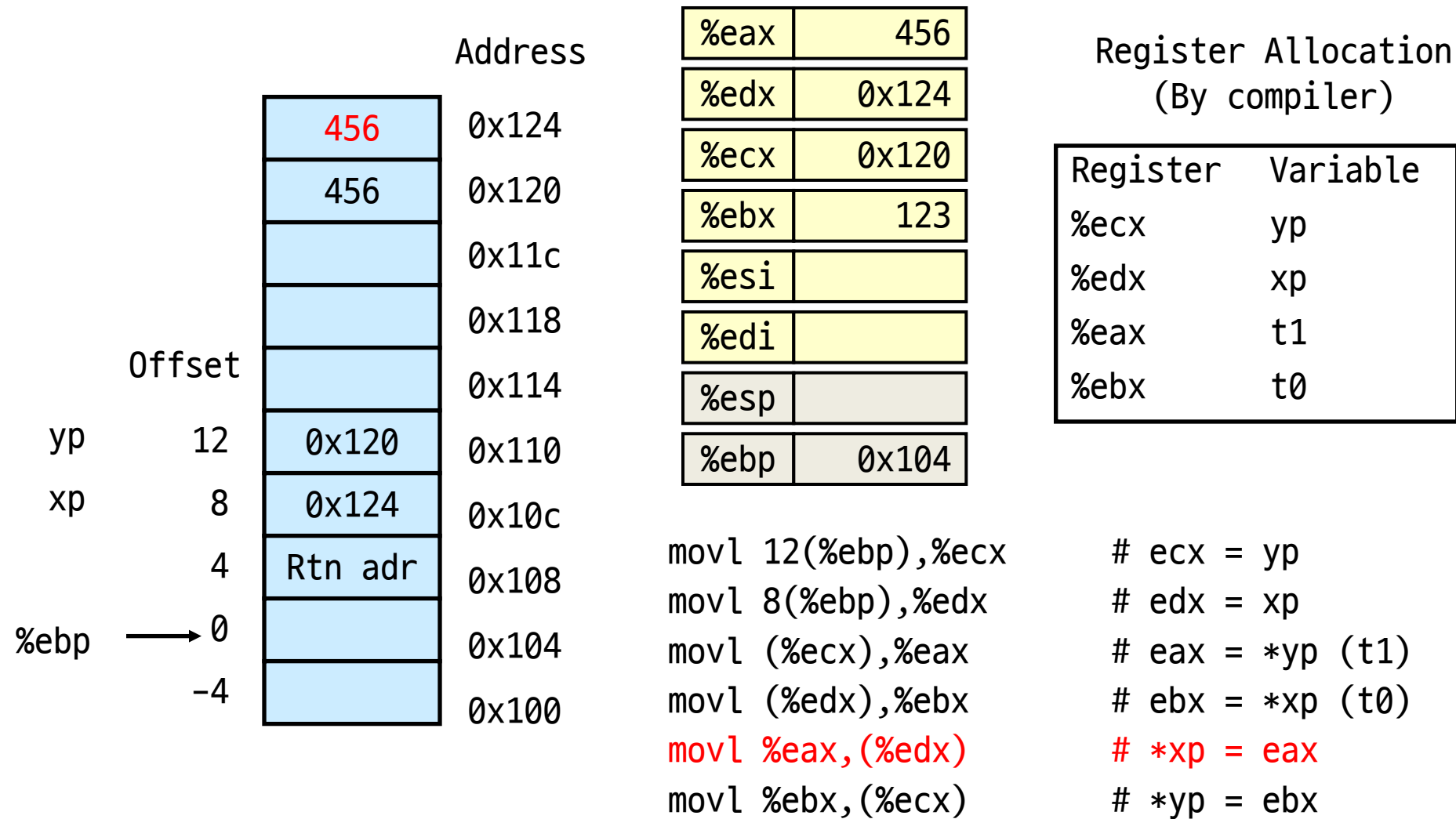
Understanding Swap (5)



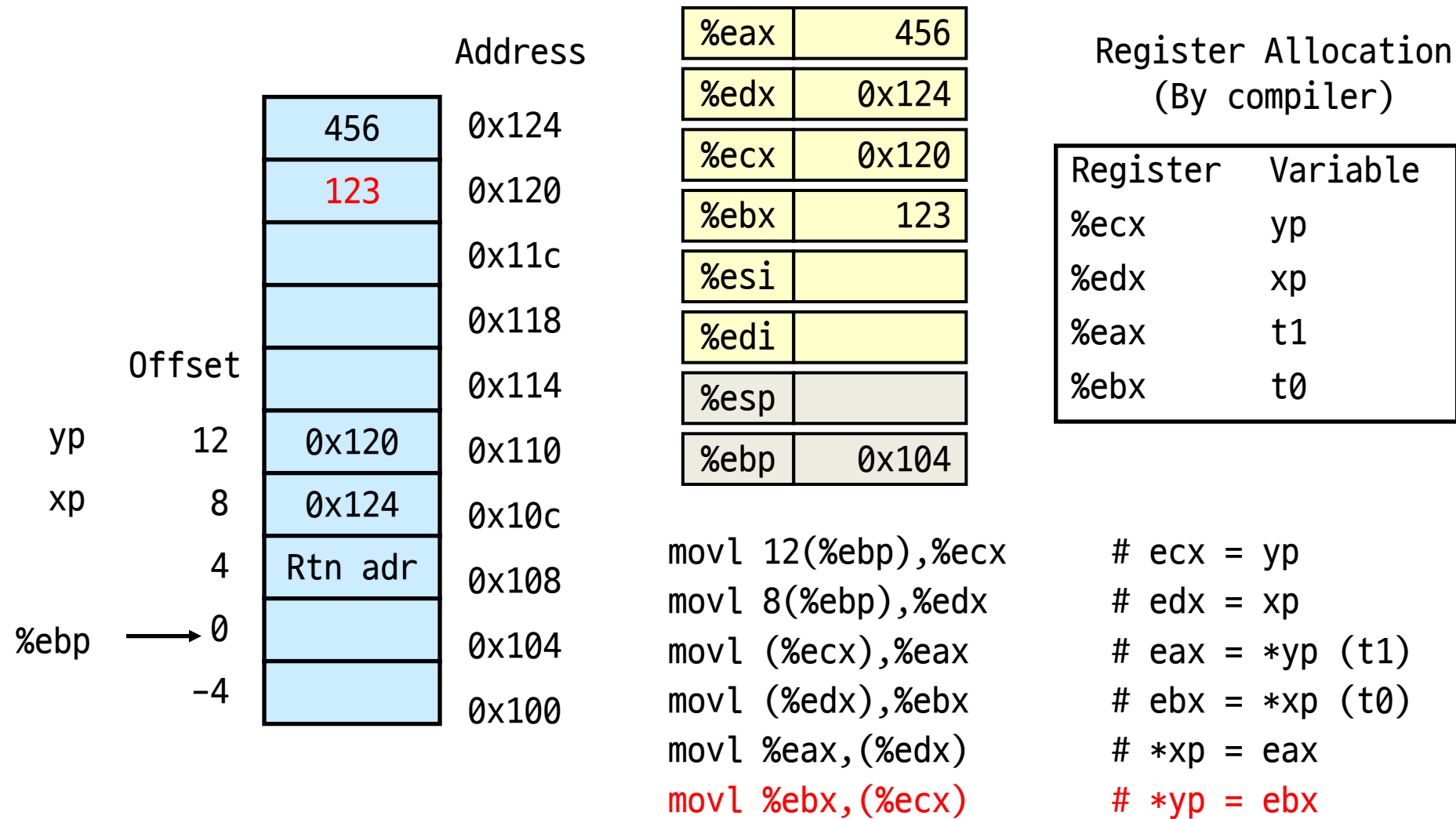
Understanding Swap (6)



Understanding Swap (7)



Understanding Swap (8)



Arithmetic/Logical Ops. (1)

Two operands instructions

• addl	Src, Dest	$\text{Dest} = \text{Dest} + \text{Src}$
• subl	Src, Dest	$\text{Dest} = \text{Dest} - \text{Src}$
• mull	Src, Dest	$\text{Dest} = \text{Dest} * \text{Src}$ (unsigned)
• imull	Src, Dest	$\text{Dest} = \text{Dest} * \text{Src}$ (signed)
• sall	Src, Dest	$\text{Dest} = \text{Dest} \ll \text{Src}$ (= shll)
• sarl	Src, Dest	$\text{Dest} = \text{Dest} \gg \text{Src}$ (Arith.)
• shrl	Src, Dest	$\text{Dest} = \text{Dest} \gg \text{Src}$ (Logical)
• xorl	Src, Dest	$\text{Dest} = \text{Dest} \wedge \text{Src}$
• andl	Src, Dest	$\text{Dest} = \text{Dest} \& \text{Src}$
• orl	Src, Dest	$\text{Dest} = \text{Dest} \mid \text{Src}$

Arithmetic/Logical Ops. (2)

One operand instructions

- `incl Dest` $\text{Dest} = \text{Dest} + 1$
- `decl Dest` $\text{Dest} = \text{Dest} - 1$
- `negl Dest` $\text{Dest} = -\text{Dest}$
- `notl Dest` $\text{Dest} = \sim\text{Dest}$

Address Computation

`leal Src, Dest`

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

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

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

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)

```
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
    movl %esp,%ebp
```

} Set Up

```
    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax
```

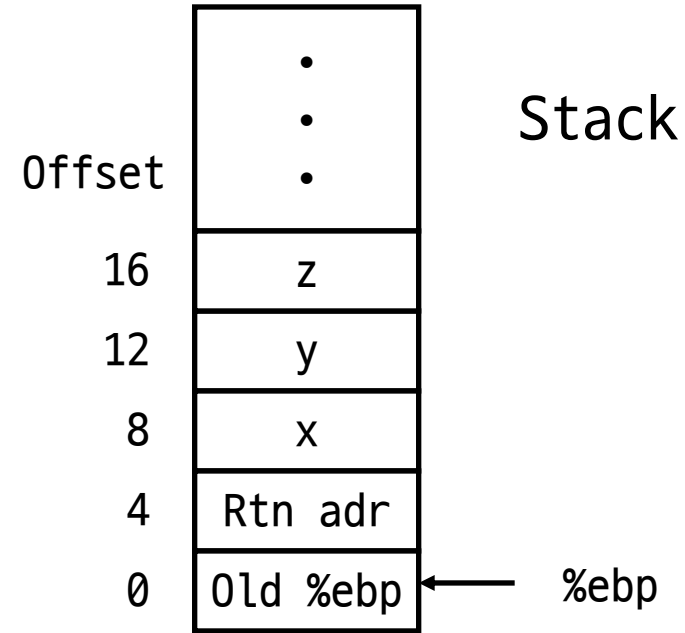
} Body

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

} Finish

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
movl 12(%ebp),%edx
leal (%edx,%eax),%ecx
leal (%edx,%edx,2),%edx
sall $4,%edx
addl 16(%ebp),%ecx
leal 4(%edx,%eax),%eax
imull %ecx,%eax
```

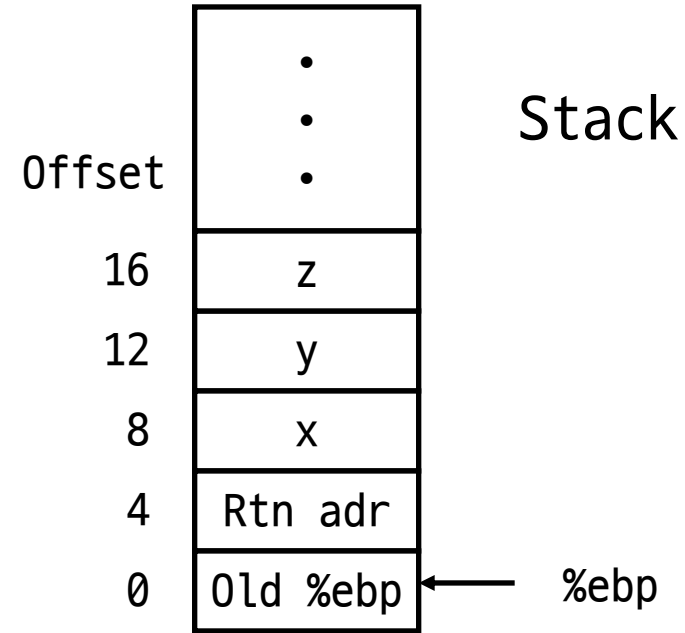
```
# eax = x
# edx = y
# ecx = x + y (t1)
# edx = 3 * y
# edx = 48 * y (t4)
# ecx = z + t1 (t2)
# eax = x + t4 + 4 (t5)
# eax = t2 * t5 (rval)
```

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;
}
```

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



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;
}
```

logical:

```
pushl %ebp
movl %esp,%ebp
```

} Set Up

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

} Body

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

} Finish

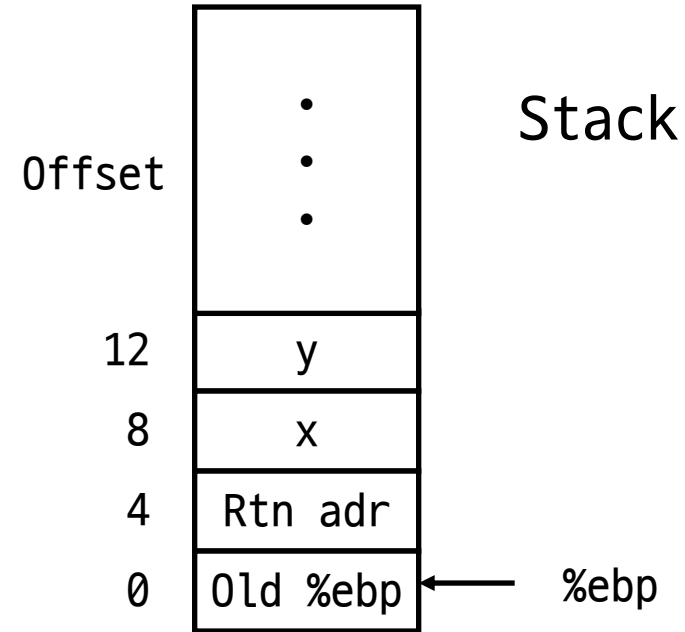
Example: logical

```
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;
}
```

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

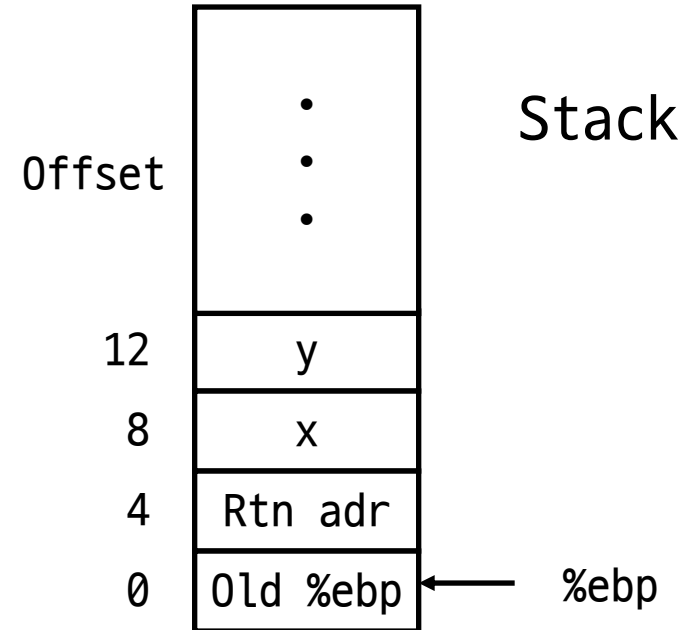
```
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sarl $17,%eax
andl $8185,%eax
```

```
# eax = x
# eax = x ^ y    (t1)
# eax = t1 >> 17  (t2)
# eax = t2 & 8185
```



Example: andor

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



```
movl 12(%ebp),%eax
movl 8(%ebp),%edx
andl %edx,%eax
movl $-1,%edx
orl %edx,%eax
```

```
# eax = y
# edx = x
# eax = x & y (t2)
# edx = 0xffffffff (t3)
# eax = t2 | t3
```

Make it short!

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
 - $D(Rb, Ri, S) \rightarrow Rb + S * Ri + D$
 - Useful for arithmetic expressions, too
- Instructions can have varying lengths
 - IA-32 instructions can range from 1 to 15 bytes

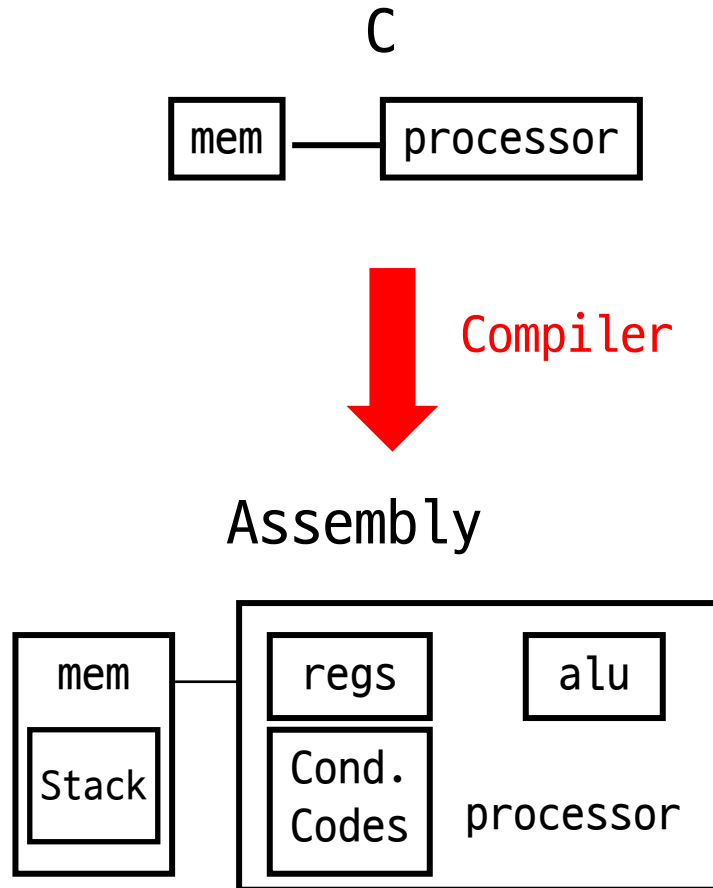
Summary (1)

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, (%ecx)

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

```
void doit(int *xp, int *yp)
```

Exercise

C → ASM

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

