Assembly I: Basic Operations

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

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"

%eax
%ebx
%ecx
%edx
%esi
%edi
%esp
%ebp

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;**

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) **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]]
  - Useful to access arrays and structures

Indexed Addressing Modes (2)

- **Address computation example**

%edx	0xf000
%ecx	0x0100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

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
```

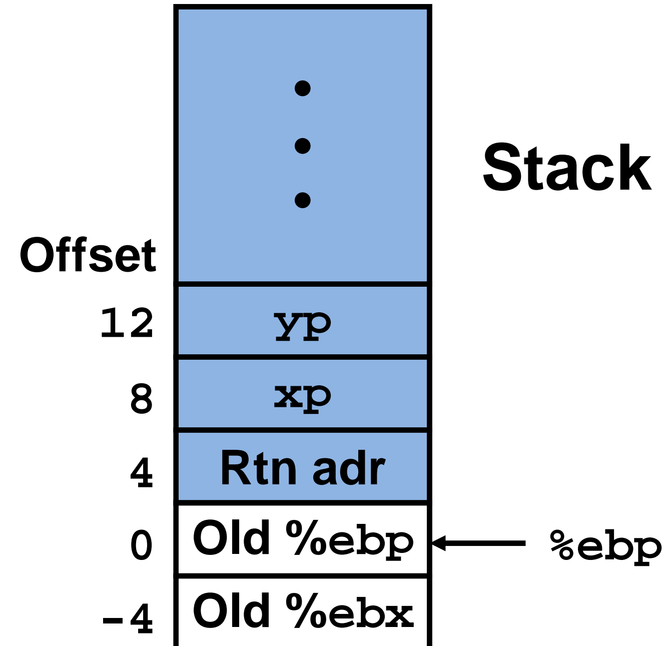
Setup

Body

Finish

Understanding Swap (1)

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

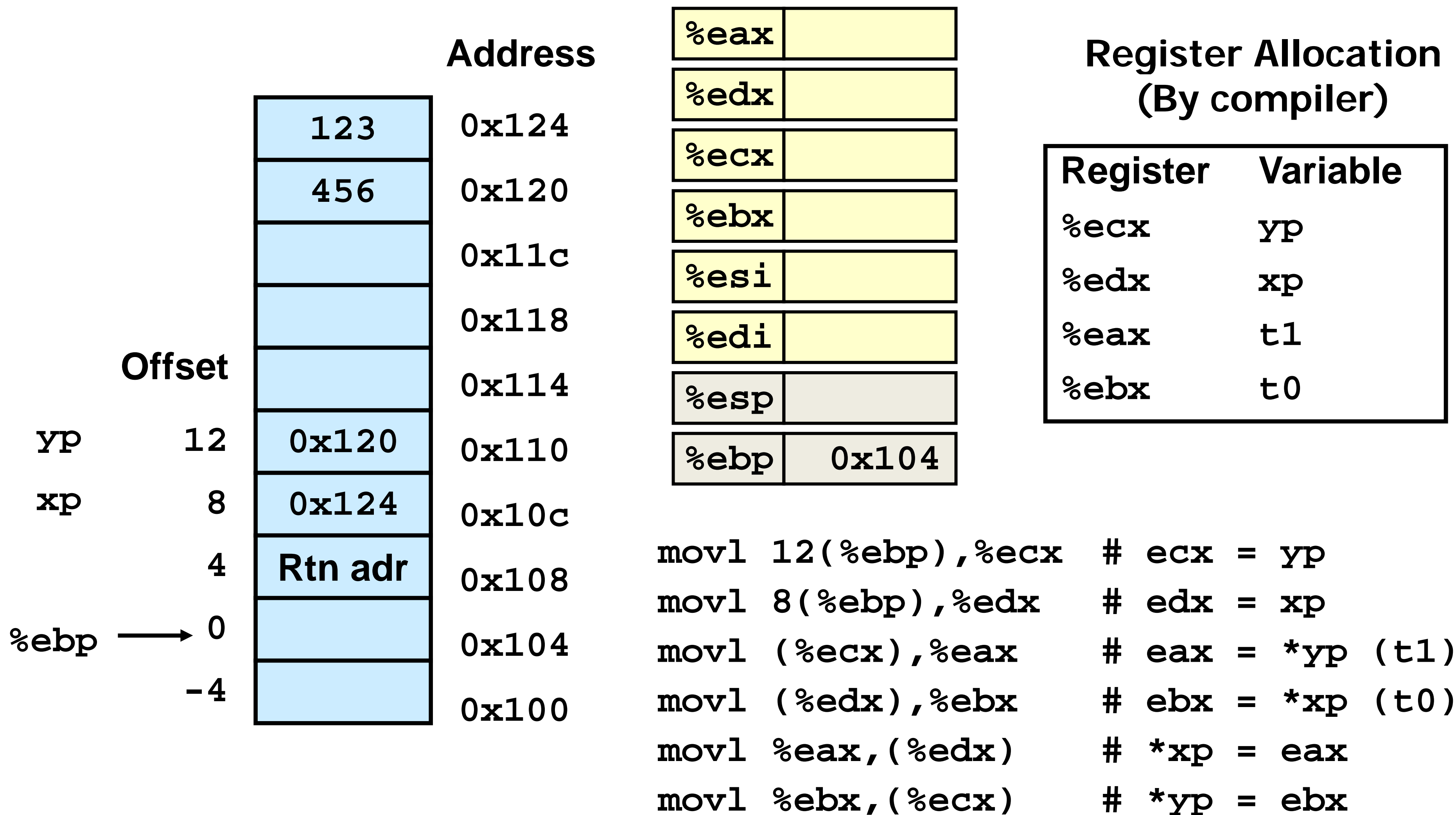


Register Allocation
(By compiler)

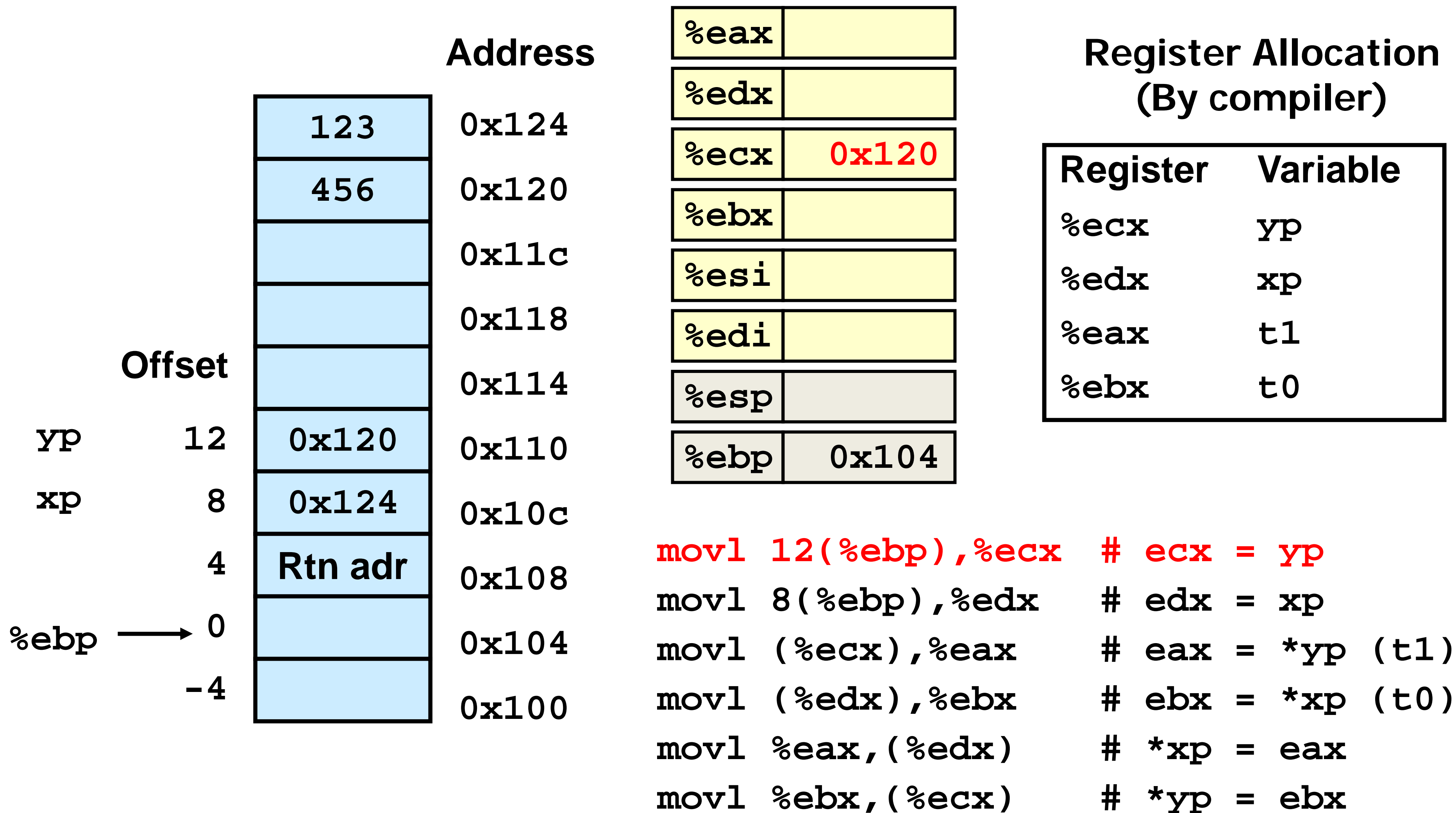
Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
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)

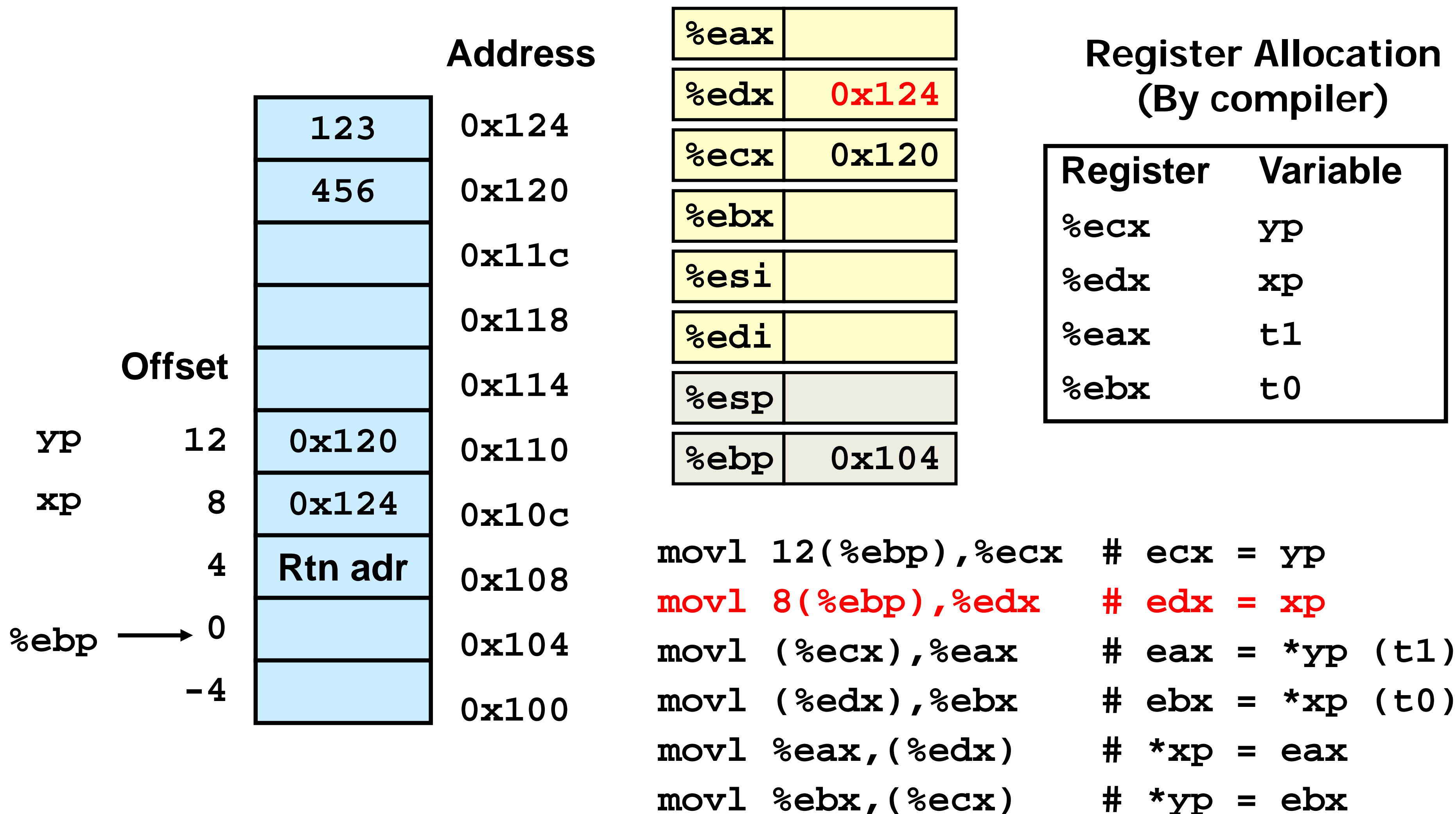


Register Allocation (By compiler)

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)

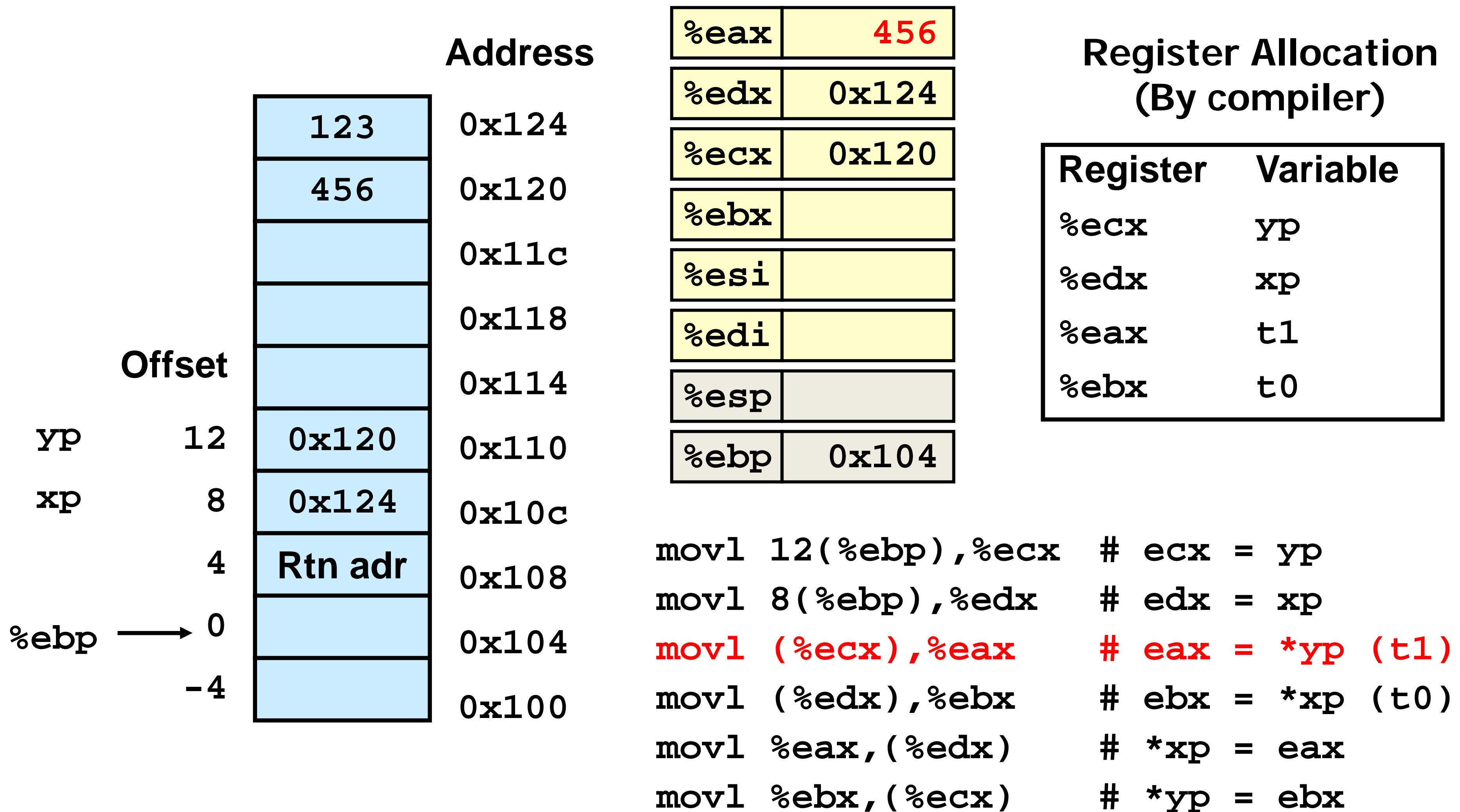


Register Allocation (By compiler)

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

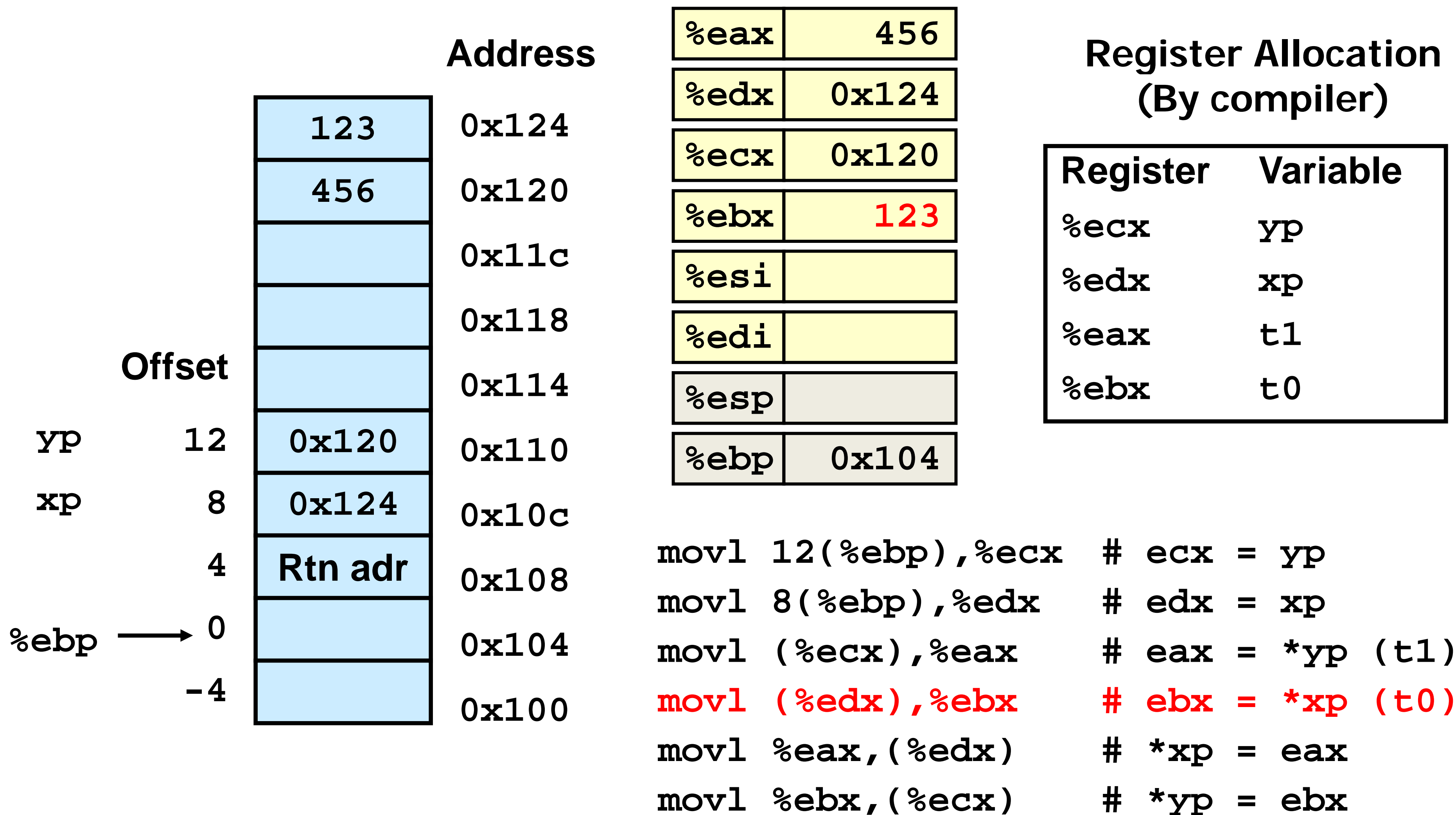
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap (5)




```
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)

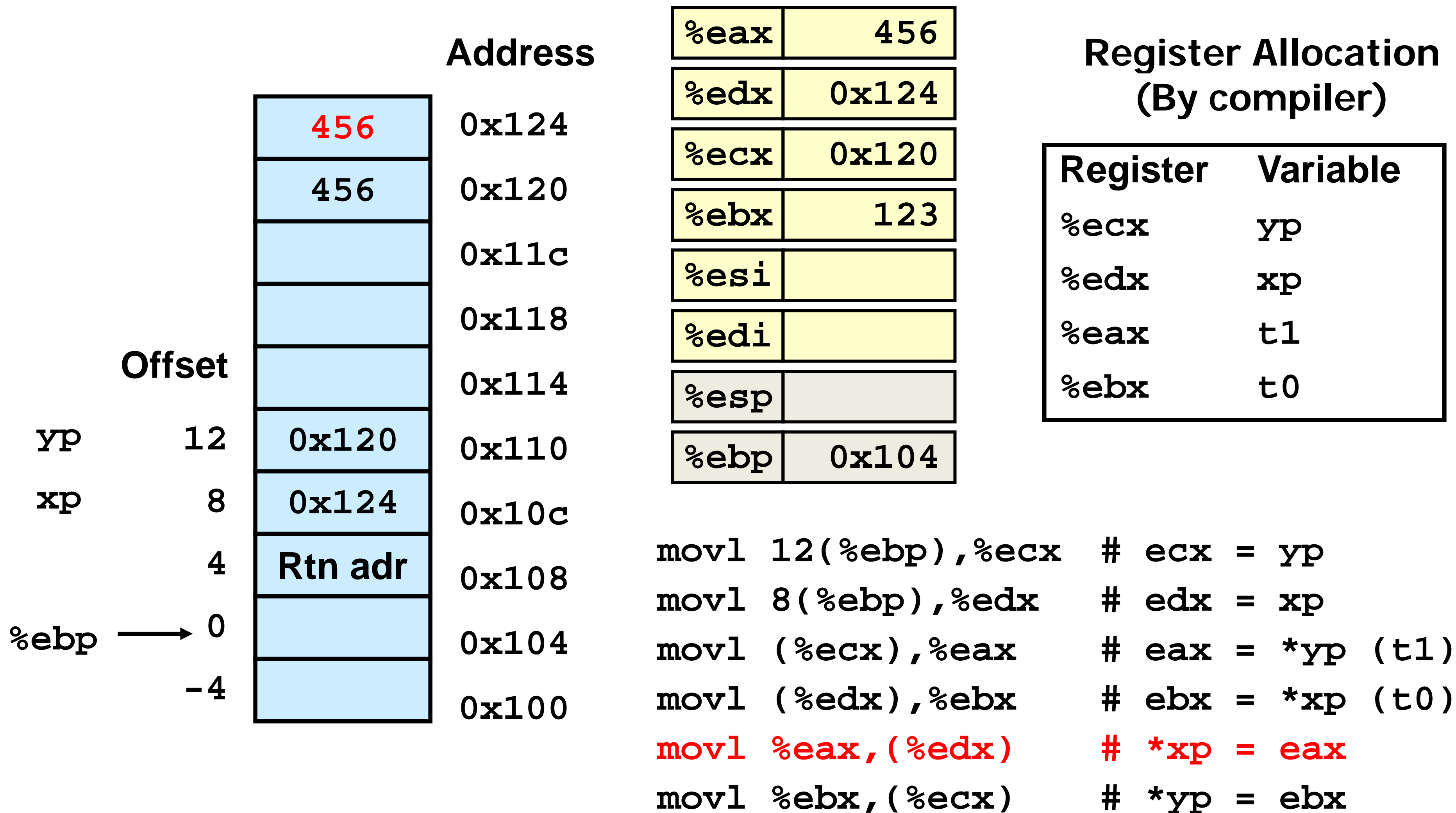


Register Allocation
(By compiler)

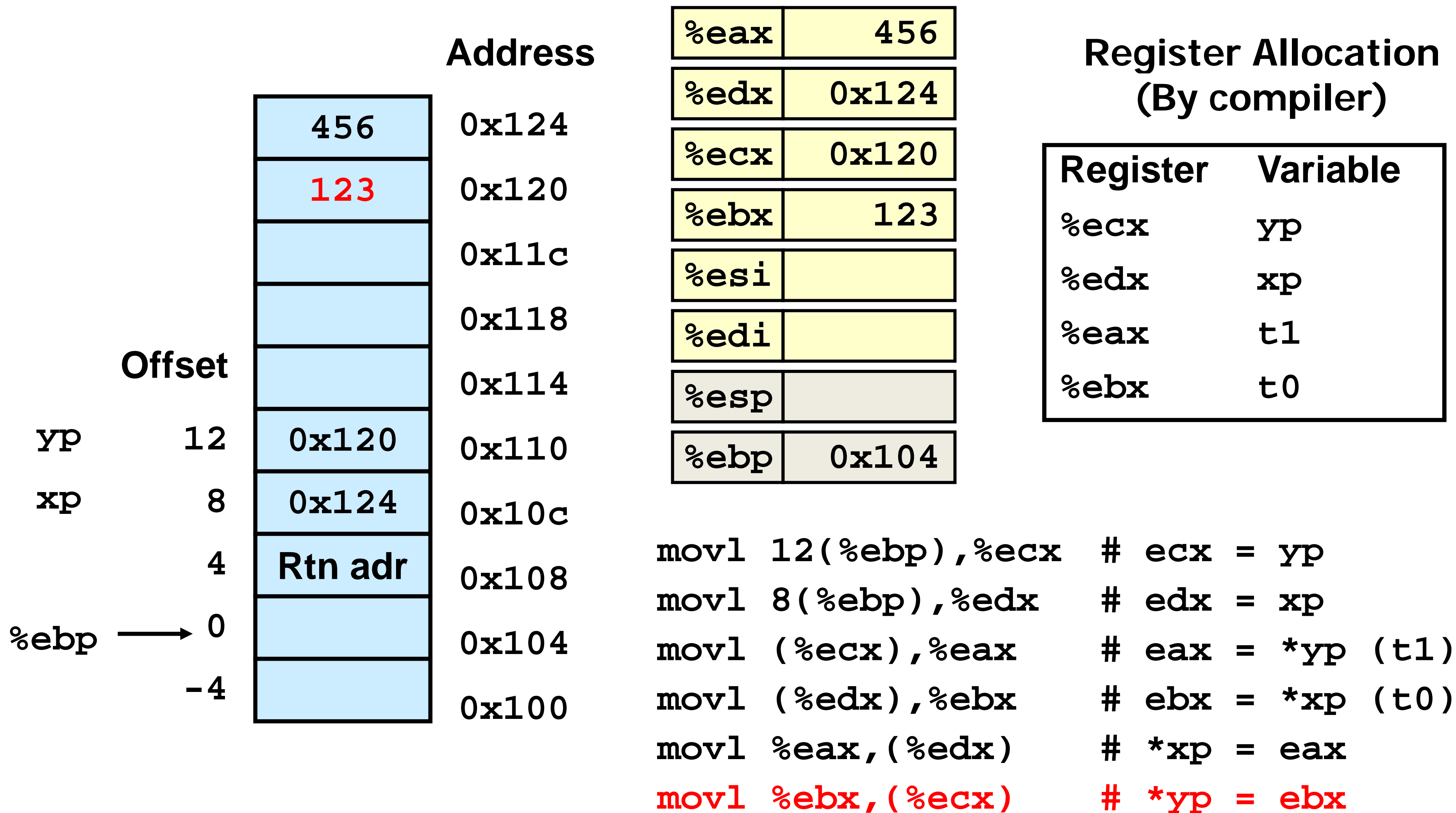
Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
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)



Understanding Swap (8)



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 (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

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

- **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;
}
```

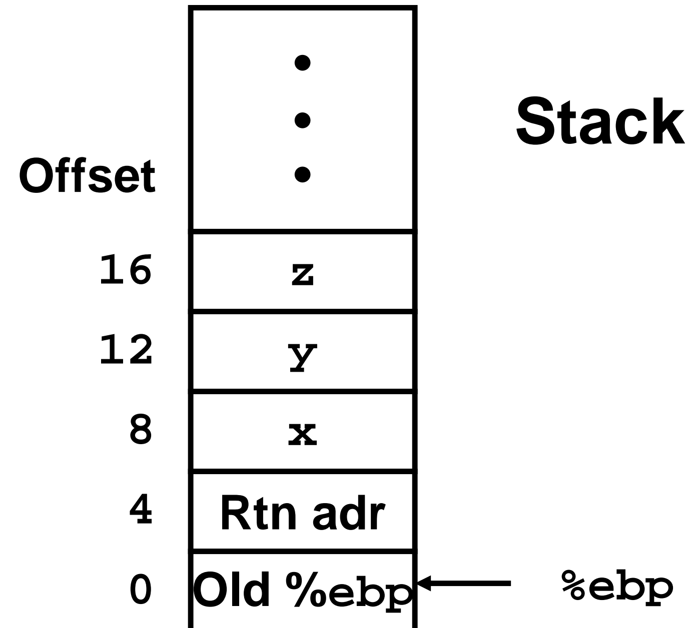
```
arith:
    pushl %ebp                    } Set Up
    movl %esp,%ebp

    movl 8(%ebp),%eax             } Body
    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

    movl %ebp,%esp                } Finish
    popl %ebp
    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 = 4+t4+x (t5)
imull %ecx,%eax             # eax = t5*t2 (rval)
```

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           } Set Up
    movl %esp,%ebp

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

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

$2^{13} = 8192, 2^{13} – 7 = 8185$

```
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
```

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

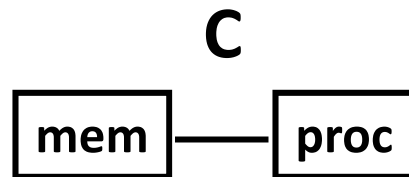
Summary (1)

- **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

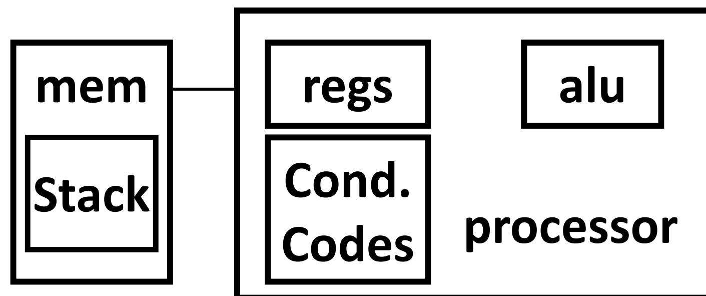
Summary (2)

Machine Models

C



Assembly



Data

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

1) byte
2) 2-byte word
3) 4-byte long word
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