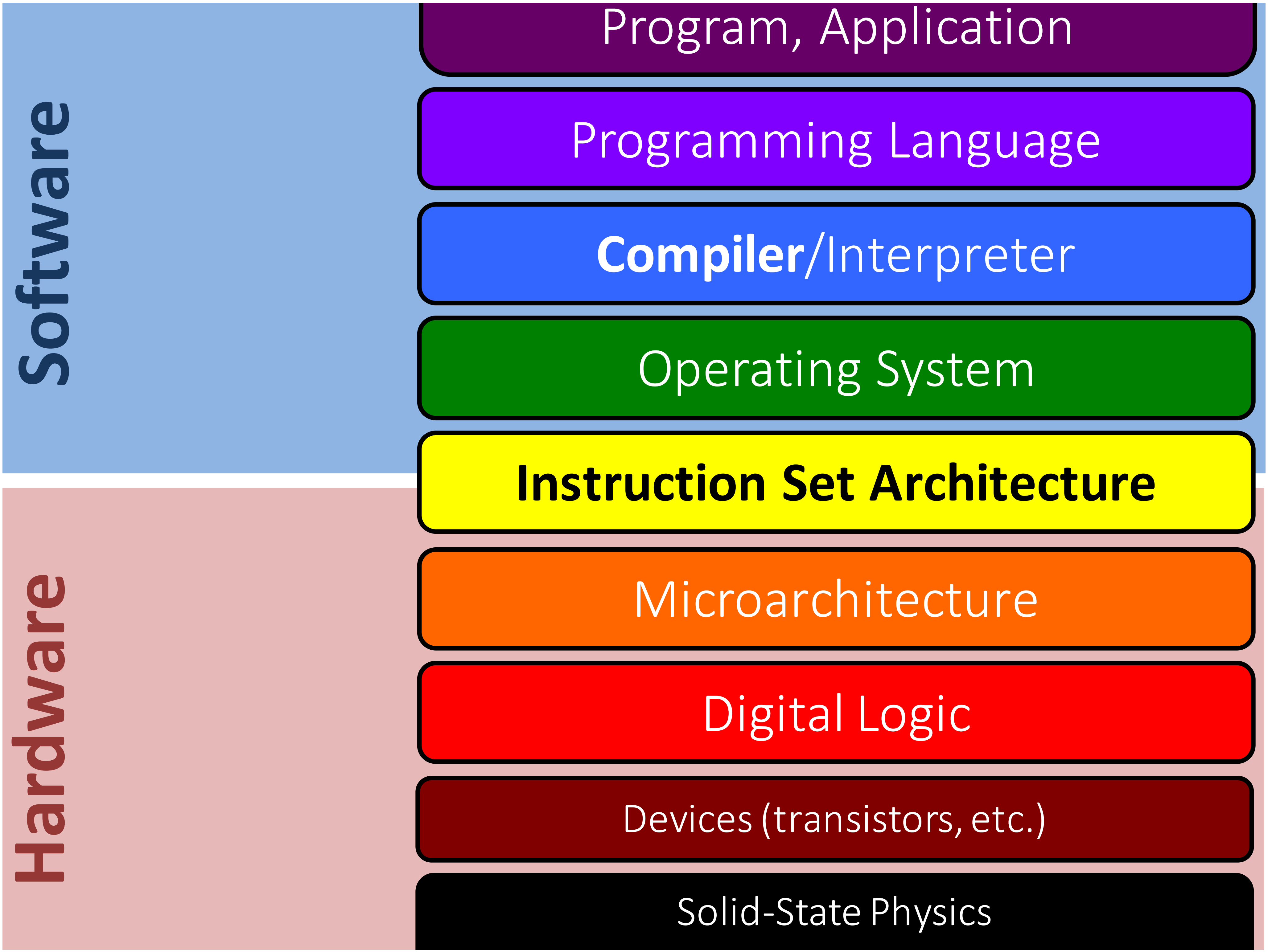
x86 basics

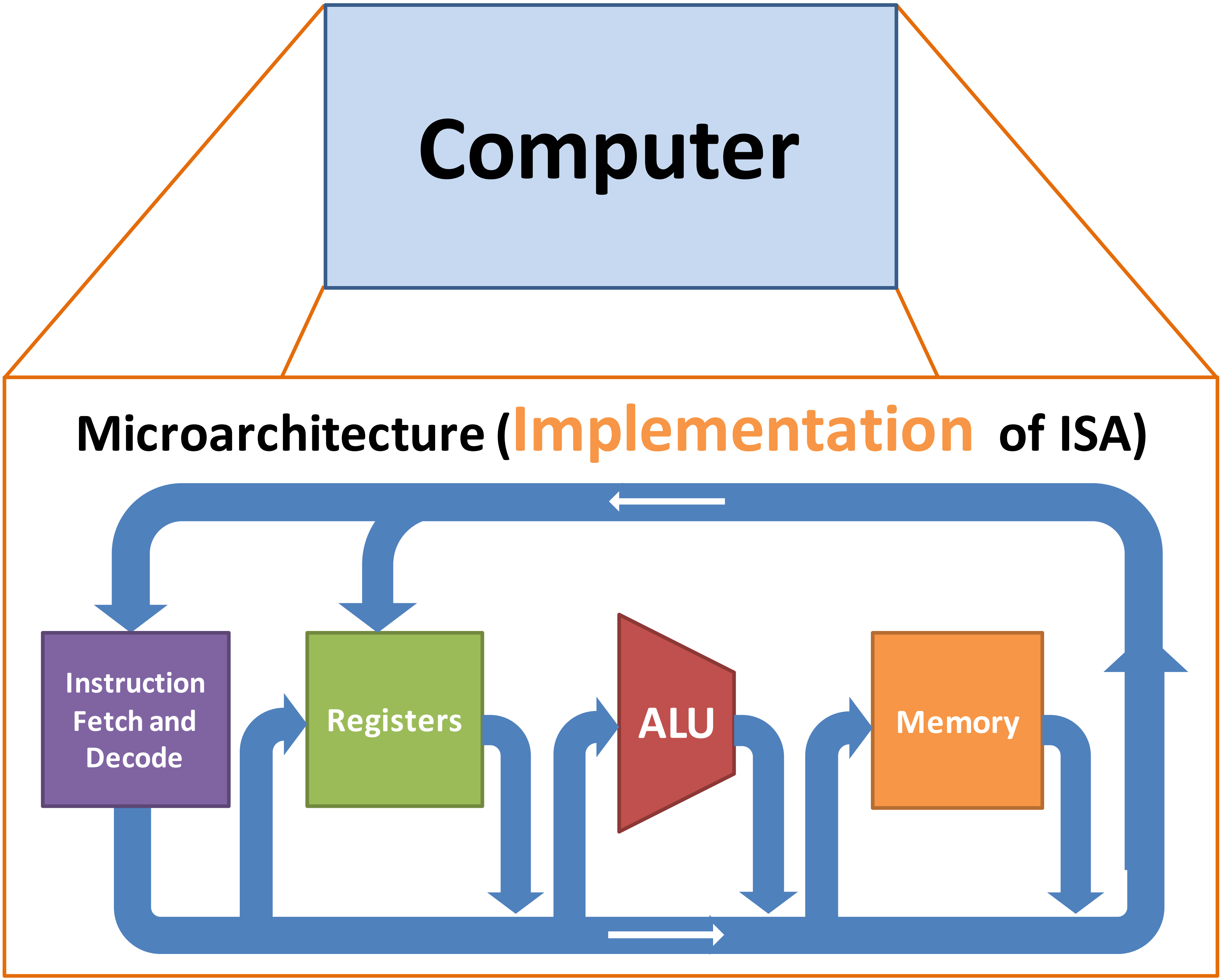
ISA context and x86 history

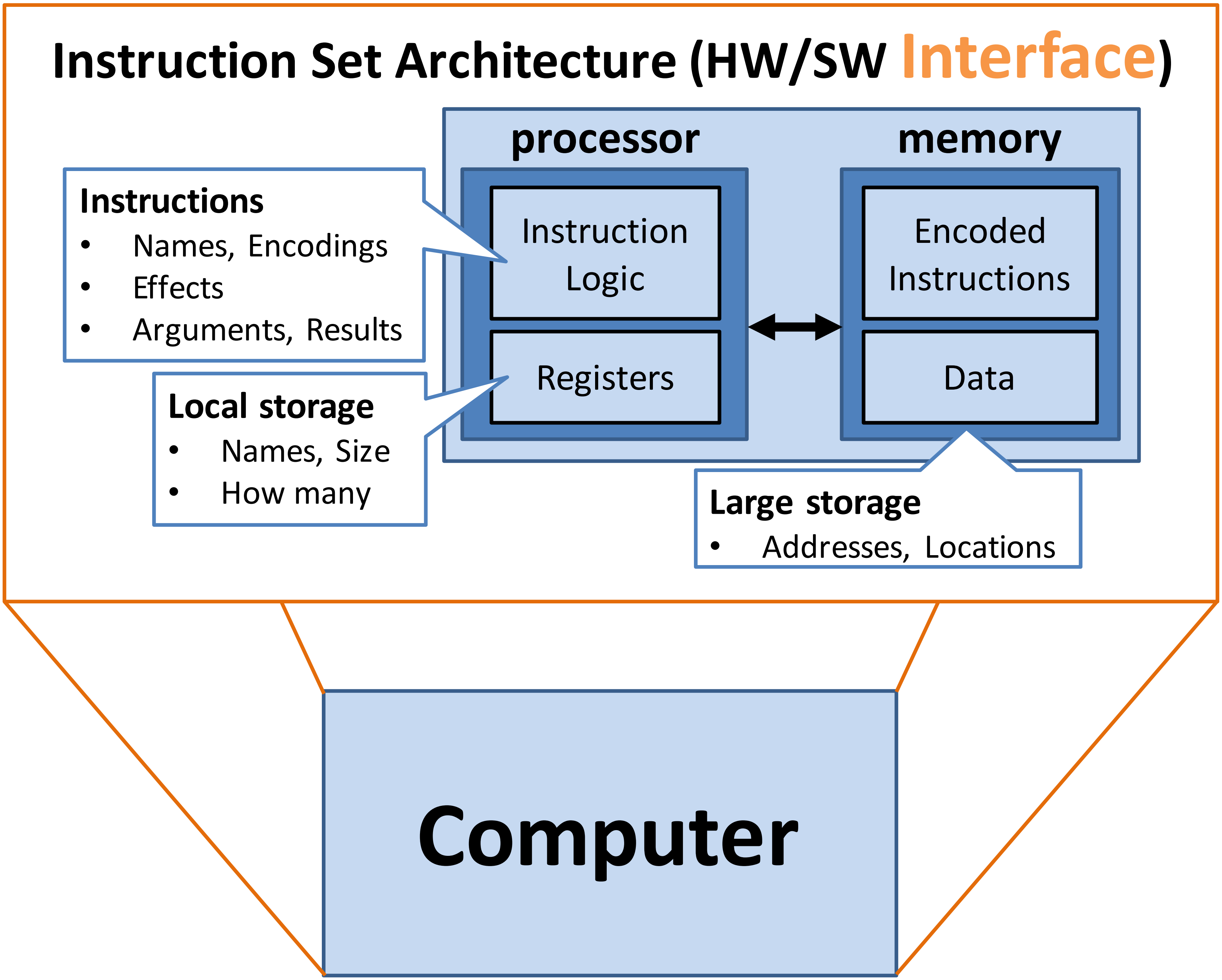
Translation tools: C --> assembly <--> machine code

x86 Basics:

- Registers
- Data movement instructions
- Memory addressing modes
- Arithmetic instructions


Software
Hardware
Program, Application
Programming Language
Compiler/Interpreter
Operating System
Instruction Set Architecture
Microarchitecture
Digital Logic
Devices (transistors, etc.)
Solid-State Physics


Computer
Microarchitecture (Implementation of ISA)
Instruction Fetch and Decode
Registers
ALU
Memory


Instruction Set Architecture (HW/SW Interface)
processor
memory
Instructions
• Names, Encodings
• Effects
• Arguments, Results
Instruction Logic
Registers
Encoded Instructions
Data
Local storage
• Names, Size
• How many
Large storage
• Addresses, Locations
Computer

a brief history of x86

CISC
(vs. RISC)

Word Size	*ISA*	*First*	*Year*
16	**8086**	**Intel 8086**	1978

First 16-bit processor. Basis for IBM PC & DOS
1MB address space

240 now: 32	**IA32**	Intel 386	1985

First 32-bit ISA.
Flat addressing, improved OS support

2015: most laptops, desktops, servers.

240 soon: 64	**x86-64**	AMD Opteron	2003*

Slow AMD/Intel conversion, slow adoption.
*Not actually x86-64 until few years later.
Mainstream only after ~10 years.

Turning C into Machine Code

C Code

```
int sum(int x, int y) {
  int t = x+y;
  return t;
}
```

`code.c`

compiler

`gcc -O1 -S code.c`

Generated IA32 Assembly Code

Human-readable language close to machine code.

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

`code.s`

assembler

Object Code

```
01010101100010011110010110
00101101000101000011000000
00110100010100001000100010
01111011000101110111000011
```

`code.o`

Linker: create full executable

Resolve references between object files, libraries, (re)locate data

Integer Registers (IA32)

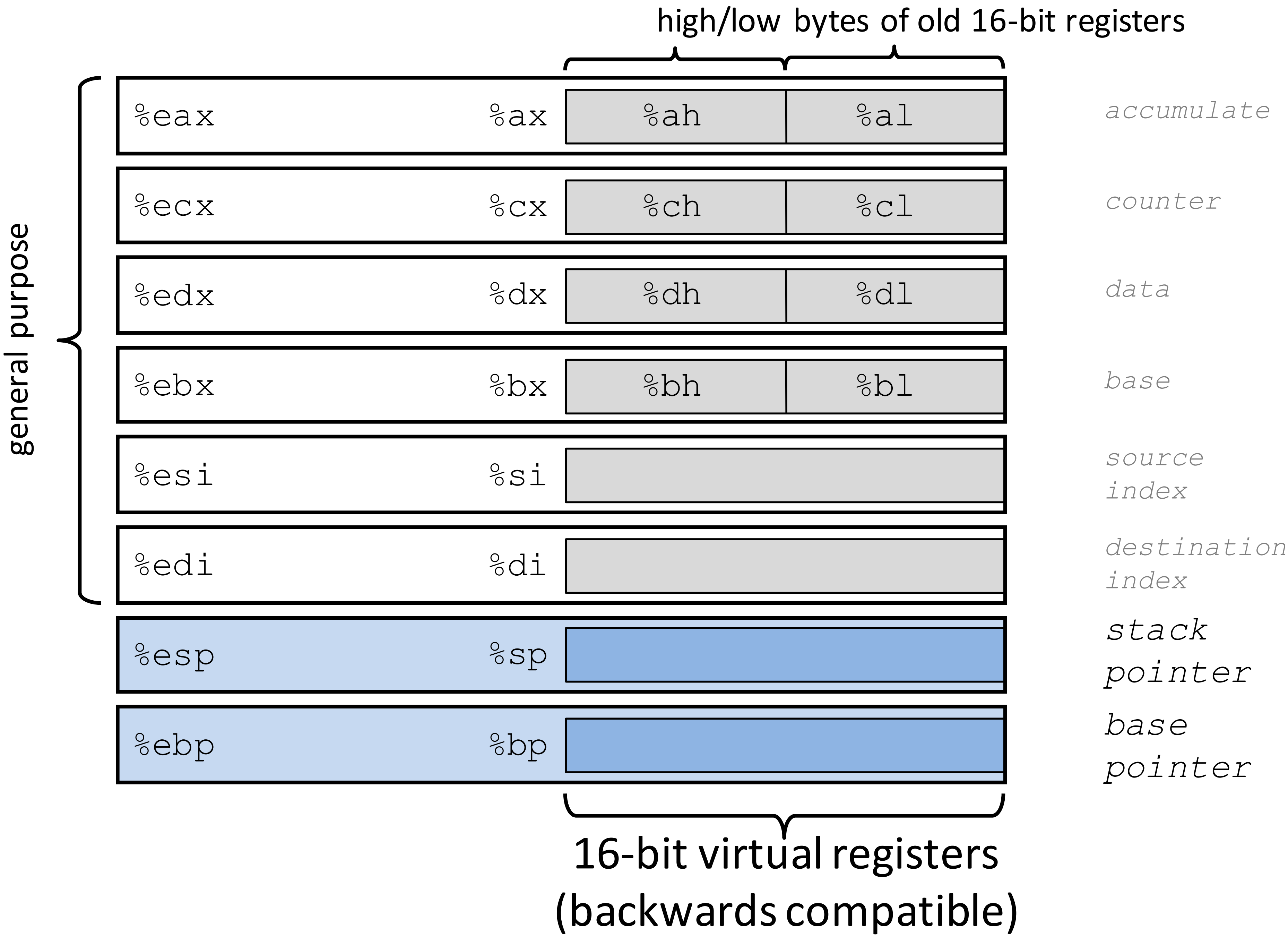
general purpose

		Origin (mostly obsolete)
`%eax`		*accumulate*
`%ecx`		*counter*
`%edx`		*data*
`%ebx`		*base*
`%esi`		*source index*
`%edi`		*destination index*
`%esp`	**special purpose**	***stack pointer***
`%ebp`	**special purpose**	***base pointer***

32-bits wide

Some have special uses
for particular instructions

Integer Registers (historical artifacts)



IA32: Three Basic Kinds of Instructions

1. **Data movement** between memory and register

   Load data from memory into register

   %reg = Mem[address]

   Store register data into memory

   Mem[address] = %reg

Memory is an array[] of bytes!

2. **Arithmetic/logic** on register or memory data

   c = a + b; z = x << y; i = h & g;

3. **Comparisons and Control flow** to choose next instruction

   Unconditional jumps to/from procedures

   Conditional branches

Data movement instructions

mov*x* ***Source*, *Dest***

x is one of {**b, w, l**}

gives size of data

movl ***Source*, *Dest*:**

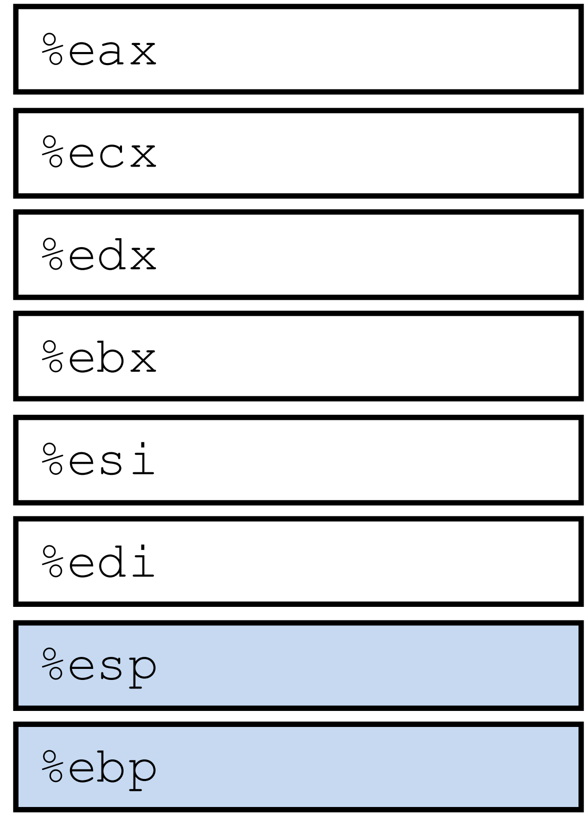
Move 4-byte "long word"

movw ***Source*, *Dest*:**

Move 2-byte "word"

movb ***Source*, *Dest*:**

Move 1-byte "byte"



historical terms from the 16-bit days
not the current machine word size

Data movement instructions

`movl` ***Source*, *Dest*:**

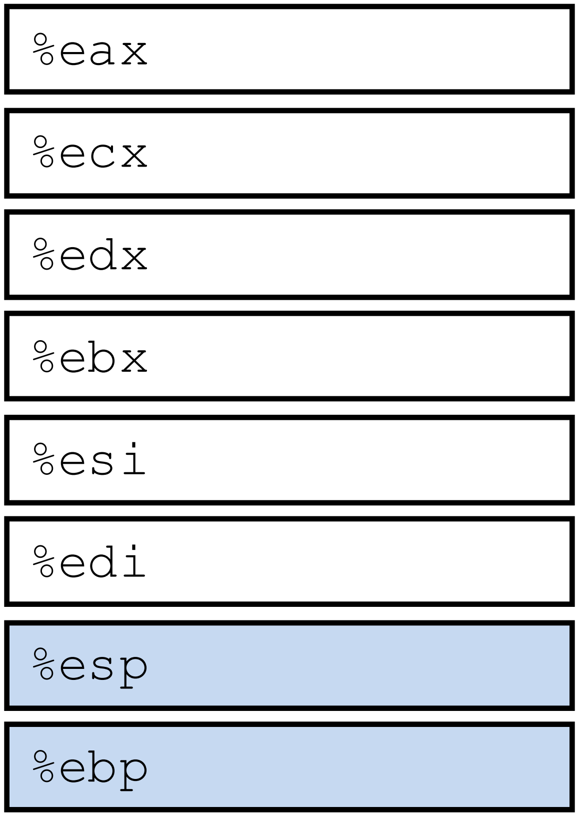
Operand Types:

Immediate: Literal integer data

Examples: **`$0x400`**, **`$-533`**

Register: One of 8 integer registers

Examples: **`%eax`**, **`%edx`**



Memory: 4 consecutive bytes in memory, at address held by register

Simplest example: **`(%eax)`**

Various other "address modes"

`movl` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`var_a = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p_a = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`var_d = var_a;`
		Mem	`movl %eax,(%edx)`	`*p_d = var_a;`
	Mem	*Reg*	`movl (%eax),%edx`	`var_d = *p_a;`

Cannot do memory-memory transfer with a single instruction.

How would you do it?

Basic Memory Addressing Modes

Indirect **(R)** **Mem[Reg[R]]**

Register R specifies the memory address

```
movl (%ecx),%eax
```

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

Register R specifies a memory address

(e.g. the start of an object)

Constant displacement D specifies the offset from that address

(e.g. a field in the object)

```
movl 8(%ebp),%edx
```

Using Basic Addressing Modes

```c
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
```

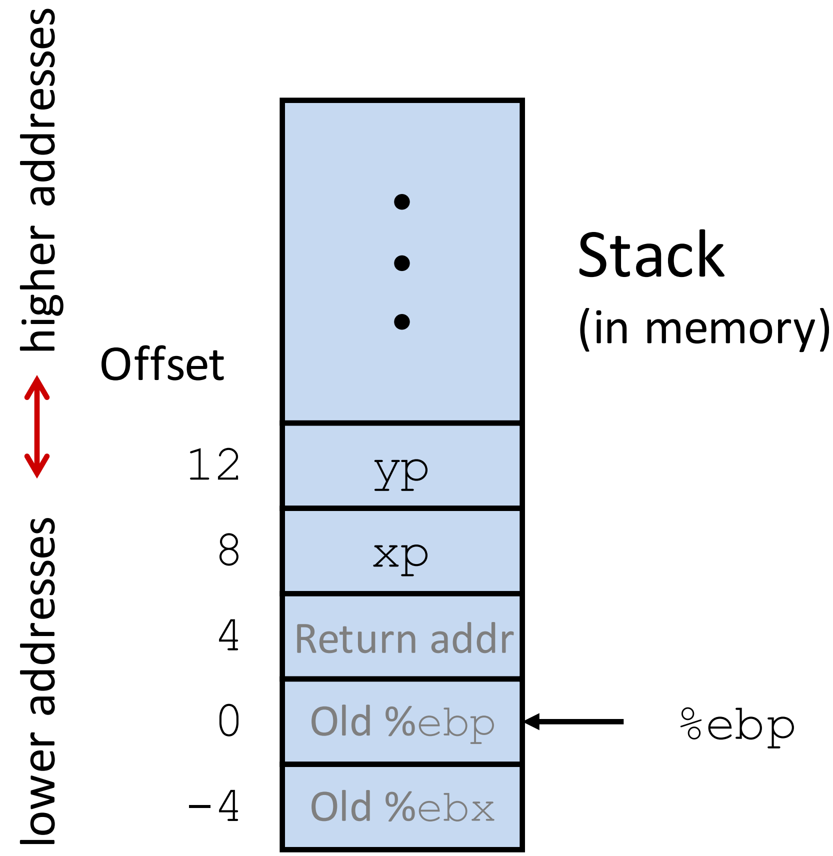
Set Up

Body

Finish

Understanding Swap

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



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

register <-> variable mapping

```
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

Register	Value
%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

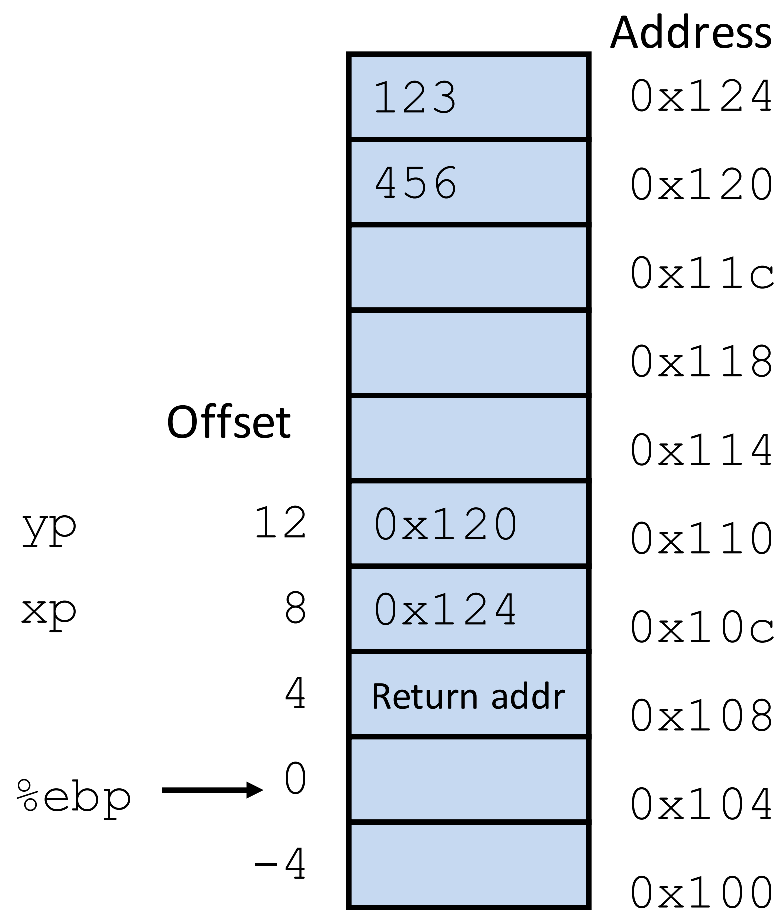
	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Return addr	0x108
%ebp →	0		0x104
	-4		0x100

higher addresses ↕ lower addresses

```
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

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




```
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

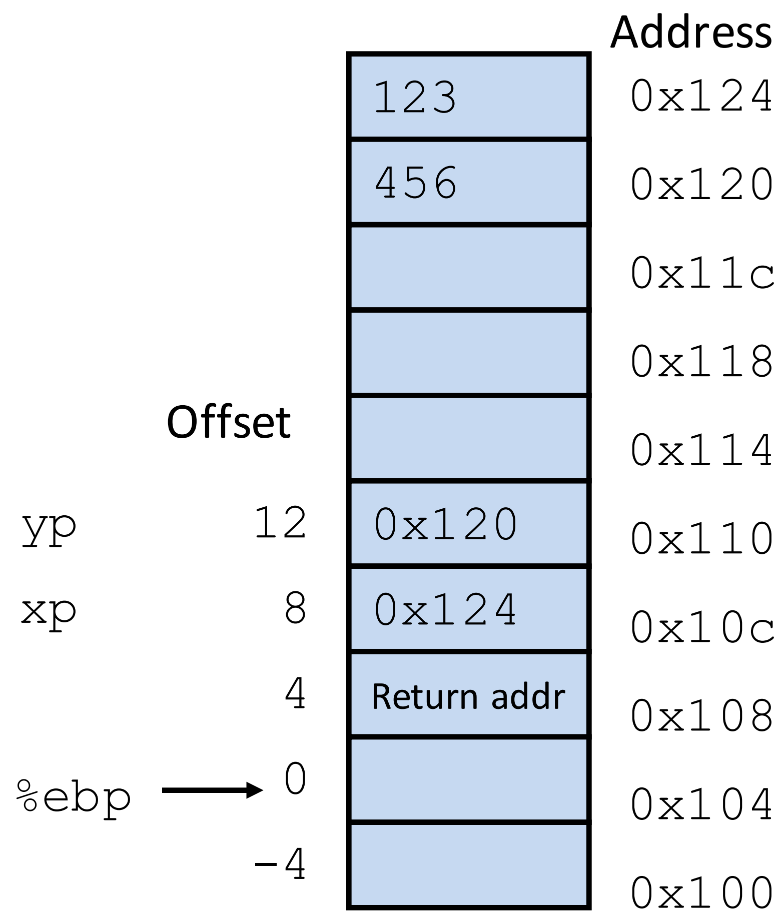
%eax	
%edx	**0x124**
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Return addr	0x108
%ebp →	0		0x104
	-4		0x100

```
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

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




```
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

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

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Return addr	0x108
%ebp →	0		0x104
	-4		0x100

```
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

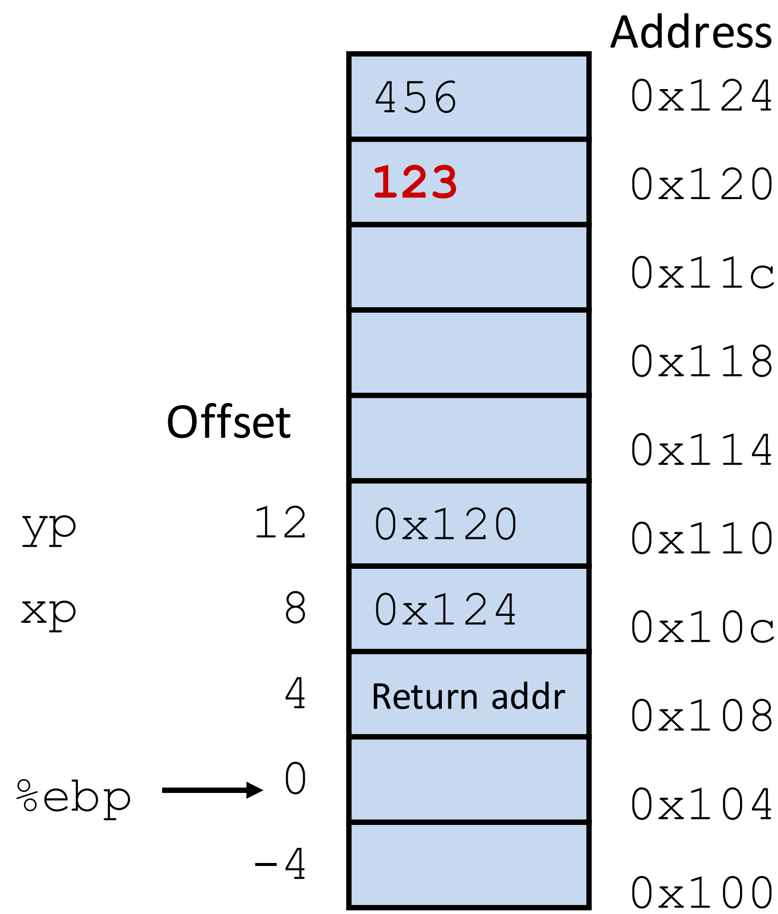
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	Offset		Address
		456	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Return addr	0x108
%ebp →	0		0x104
	-4		0x100

```
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

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




```
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
```

Complete Memory Addressing Modes

General Form:

	D(Rb,Ri,S)	Mem[Reg[Rb] + S*Reg[Ri] + D]
D:	Literal "displacement" value represented in 1, 2, or 4 bytes	
Rb:	Base register: Any register	
Ri:	Index register: Any except `%esp`; `%ebp` unlikely	
S:	Scale: 1, 2, 4, or 8 (*why these numbers?*)	

Special Cases: can use any combination of D, Rb, Ri and S

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]	(S=1,D=0)
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]	(S=1)
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]	(D=0)

Address Computation Examples

ex

Register contents

`%edx`	`0xf000`
`%ecx`	`0x100`

Addressing modes

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

Address Expression	Address Computation	Address
`0x8(%edx)`		
`(%edx,%ecx)`		
`(%edx,%ecx,4)`		
`0x80(,%edx,2)`		

`leal` ***Src,Dest*** *load effective address*

- *Src* is address mode expression
- Set *Dest* to address computed by expression
- Example: **`leal (%edx,%ecx,4), %eax`**

DOES NOT ACCESS MEMORY

!!!

Uses

- Computing addresses, *e.g.,:* translation of **`p = &x[i];`**
- Computing arithmetic expressions of the form x + k*i
  - k = 1, 2, 4, or 8

Arithmetic Operations

Two-operand instructions:

Format		*Computation*	
`addl`	*Src,Dest*	*Dest = Dest + Src*	
`subl`	***Src,Dest***	***Dest = Dest − Src***	***argument order***
`imull`	*Src,Dest*	*Dest = Dest * Src*	
`shll`	*Src,Dest*	*Dest = Dest << Src*	*a.k.a 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*	

No distinction between signed and unsigned int

except arithmetic vs. logical shift right

Arithmetic Operations

One-operand (unary) instructions

`incl`	*Dest*	*Dest* = *Dest* + 1	increment
`decl`	*Dest*	*Dest* = *Dest* − 1	decrement
`negl`	*Dest*	*Dest* = −*Dest*	*negate*
`notl`	*Dest*	*Dest* = ~*Dest*	*bitwise complement*

`leal` for arithmetic (IA32)

```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;
}
```

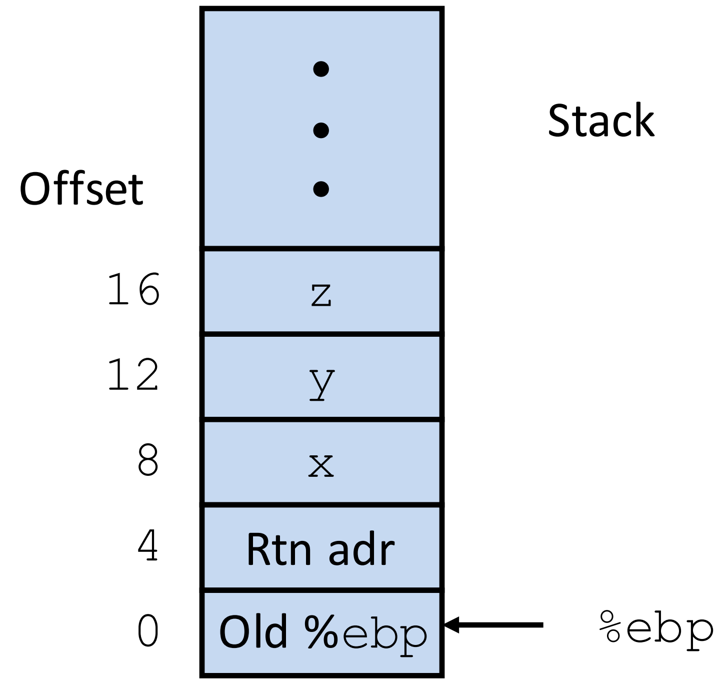
```
arith:
    pushl %ebp                    } Set Up
    movl %esp,%ebp

    movl 8(%ebp),%eax             } 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
```

Understanding `arith` (IA32)

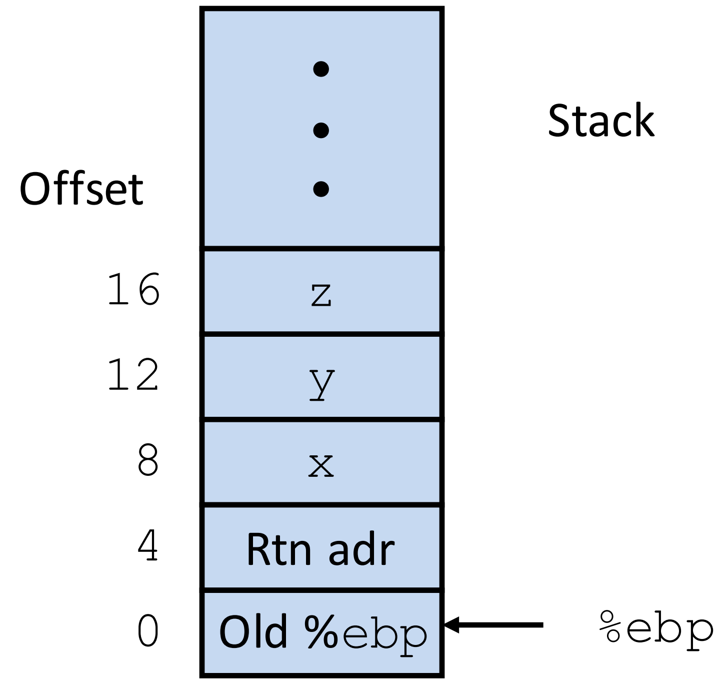
```
int arith(int x, int y, int z){
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




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

Understanding `arith` (IA32)

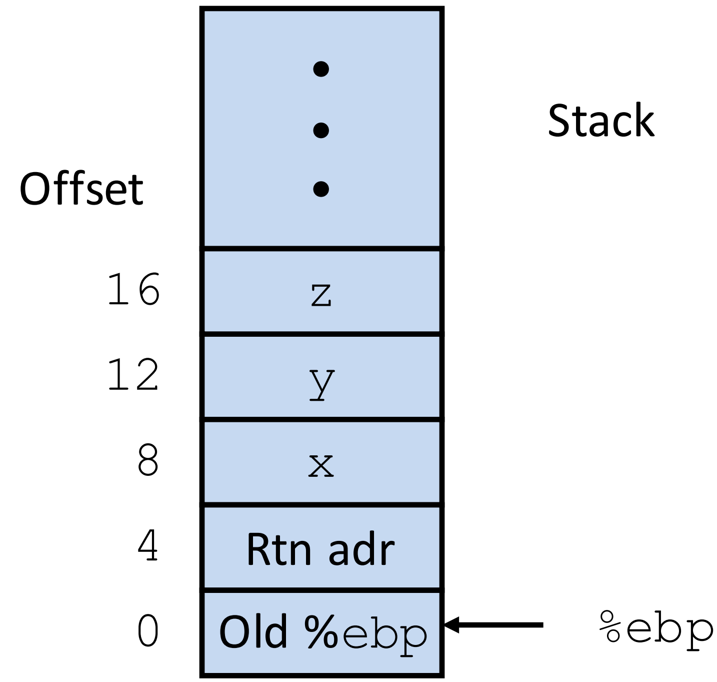
```c
int arith(int x, int y, int z){
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
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 = y + 2*y = 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)
```

Understanding `arith` (IA32)

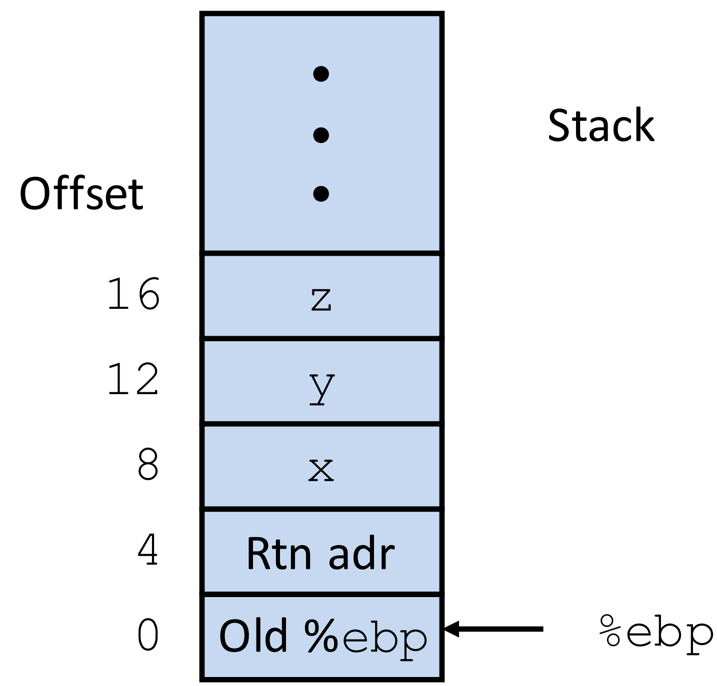
```
int arith(int x, int y, int z){
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
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 = y + 2*y = 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)
```

Understanding `arith` (IA32)

```
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 = y + 2*y = 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)
```

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;
}
```

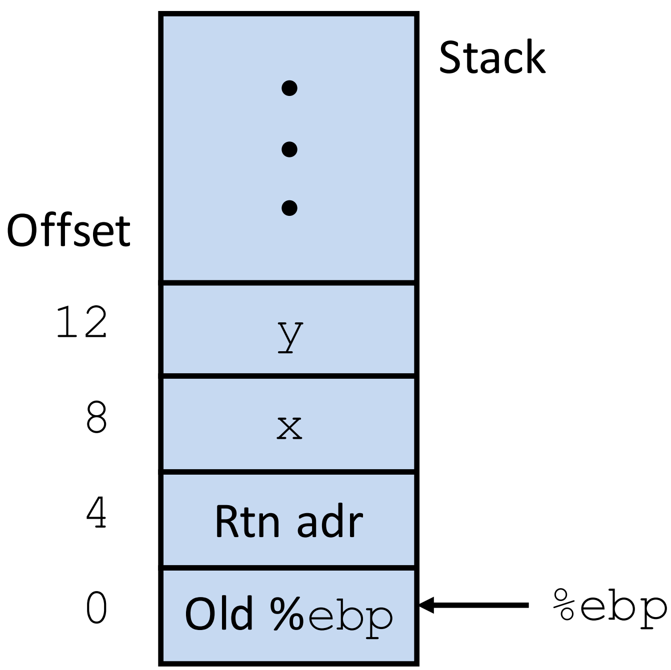
- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Same x86 code by compiling:

  `(x+y+z)*(x+4+48*y)`

```
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 = y + 2*y = 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)
```

Another Example (IA32)

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

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

Another Example (IA32)

```
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

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

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

Set Up

Body

Finish

```
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
```

Another Example (IA32)

```
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

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

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

Set Up

Body

Finish

```
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
```

Another Example (IA32)

```
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

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

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

Set Up

Body

Finish

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

...0010000000000000, ...0001111111111001

```
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
```

compiler optimization