211: Computer Architecture
Fall 2021

Instructor: Prof. David Menendez

Topics:

- Hardware-Software Interface
- Assembly Programming
    - Reading: Chapter 3

Programming Meets Hardware

High-Level Language Program

```c
#include <stdio.h>
int main() {
    int x, y, temp;
    x=1; y=2;
    temp =x; x=y;  y=temp;
    printf("%d %d %d\n",x,y,temp);
}
```

Compiler

Assembly Language Program

```
movl    $1, -8(%ebp)
movl    $2, -12(%ebp)
movl    -8(%ebp), %eax
movl    %eax, -16(%ebp)
movl    -12(%ebp), %eax
movl    %eax, -8(%ebp)
movl    -16(%ebp), %eax
movl    %eax, -12(%ebp)
movl    -16(%ebp), %eax
movl    %eax, 12(%esp)
movl    -12(%ebp), %eax
movl    %eax, 8(%esp)
movl    -8(%ebp), %eax
movl    %eax, 4(%esp)
```

ISA

Assembler

Machine Language Program

```
7f 45 4c 46 01 01 01
 00 00 00 00 00 00 00
 00 00 02 00 03 00 01
00 00 00 f0 82 04 08
34 00 00 00 c4 0c 00
00 00 00 00 00 34 00
```

How do you get performance?

Performance with Programs

(1) Program: Data structures + algorithms

(2) Compiler translates code

(3) Instruction set architecture

(4) Hardware Implementation

Instruction Set Architecture

(1) Set of instructions that the CPU can execute

  (1) What instructions are available?

  (2) How the instructions are encoded? Eventually everything is binary.

(2) State of the system (Registers + memory state + program counter)

  (1) What instruction is going to execute next

  (2) How many registers? Width of each register?

  (3) How do we specify memory addresses?

    - Addressing modes

(3) Effect of instruction on the state of the system

IA32 (X86 ISA)

There are many different assembly languages because they are processor-specific

- IA32 (x86)
  - x86-64 for new 64-bit processors
  - IA-64 radically different for Itanium processors
  - Backward compatibility: instructions added with time
- PowerPC
- MIPS

We will focus on IA32/x86-64 because you can generate and run on iLab machines (as well as your own PC/laptop)

- IA32 is also dominant in the market although smart phone, eBook readers, etc. are changing this

X86 Evolution

8086 – 1978 – 29K transistors – 5-10MHz

I386 – 1985 – 275K transistors – 16-33 MHz

Pentium4 – 2005 – 230M transistors – 2800-3800 MHz

Haswell – 2013 – > 2B transistors – 3200-3900 MHz

Added features

- Large caches
- Multiple cores
- Support for data parallelism (SIMD) eg AVX extensions

CISC vs RISC

CISC: complex instructions : eg X86

- Instructions such as strcpy/AES and others
- Reduces code size
- Hardware implementation complex?

RISC: simple instructions: eg Alpha

- Instructions are simple add/ld/st
- Increases code size
- Hardware implementation simple?

Aside About Implementation of x86

About 30 years ago, the instruction set actually reflected the processor hardware

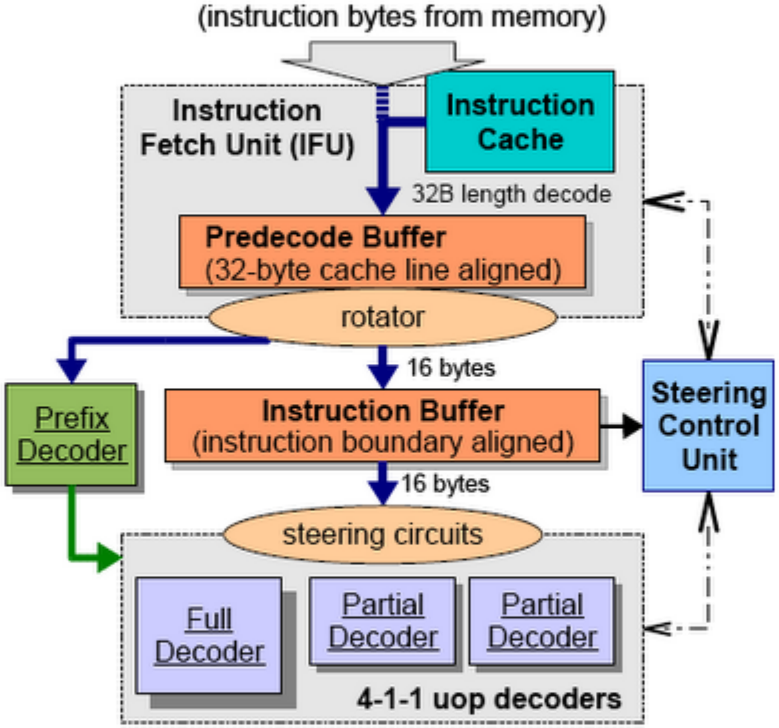
- E.g., the set of registers in the instruction set is actually what was present in the processor

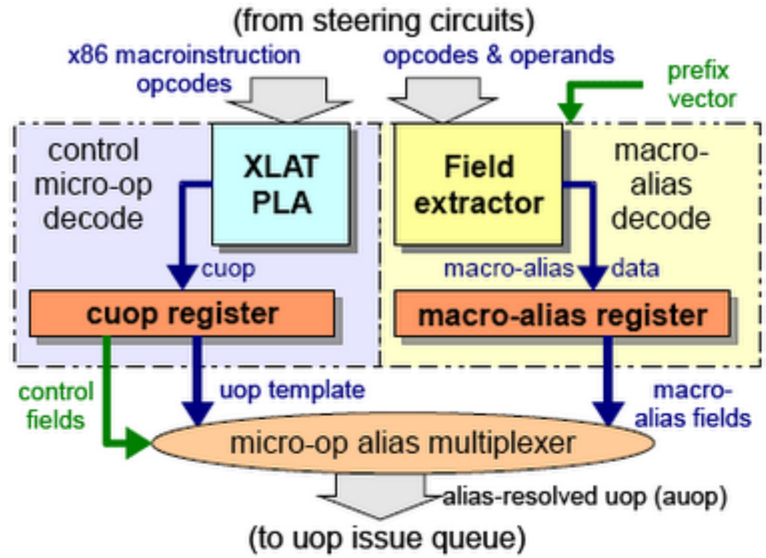
As hardware advanced, industry faced with choice

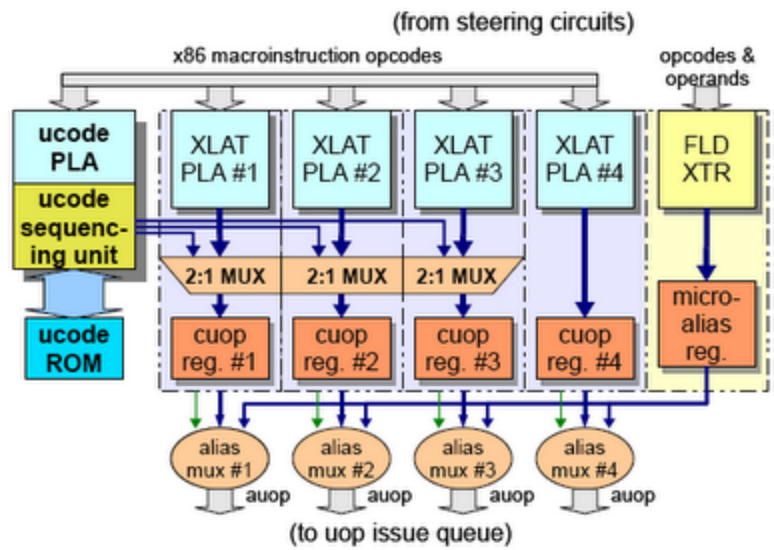
- Change the instruction set: bad for backward compatibility
- Keep the instruction set: harder to exploit hardware advances
  - Example: many more registers but only small set introduced circa 1980

Starting with the P6 (PentiumPro), IA32 actually got implemented by Intel using an “interpreter” that translates IA32 instructions into a simpler “micro” instruction set

P6 Decoder/Interpreter







Assembly Programming

Brief tour through assembly language programming

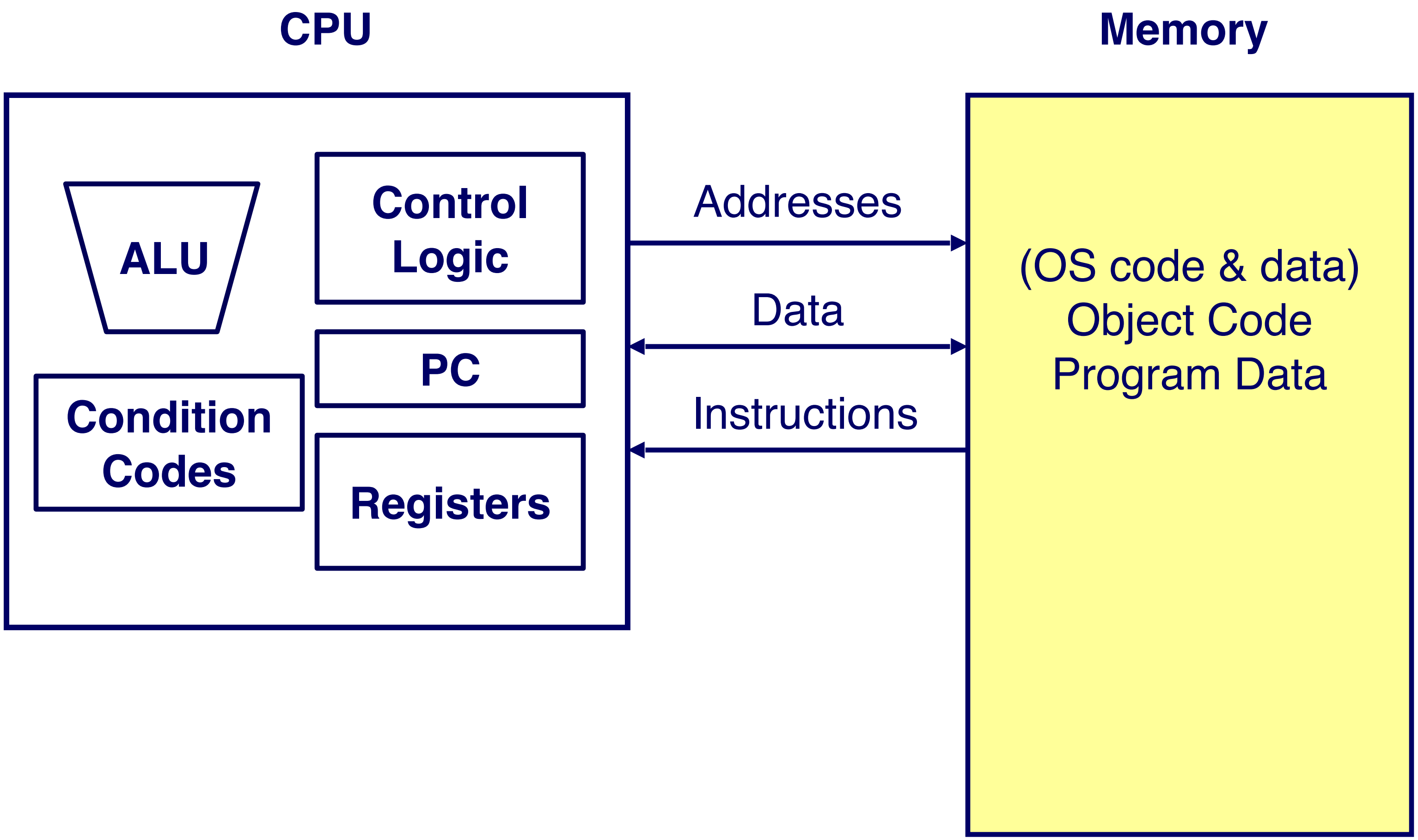
Why?

- Machine interface: where software meets hardware
- To understand how the hardware works, we have to understand the interface that it exports

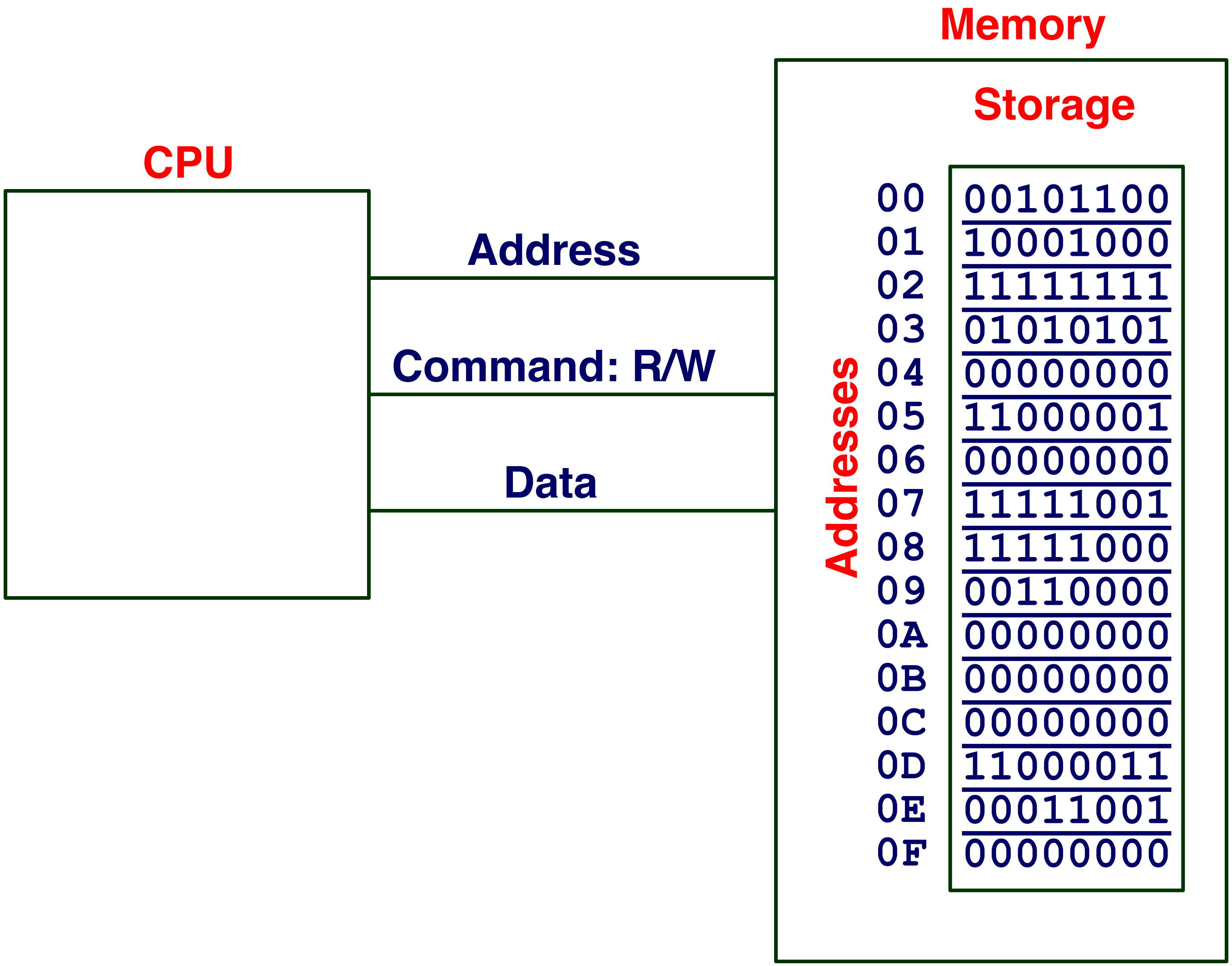
Why not binary language?

- Much easier for humans to read and reason about
- Major differences:
  - Human readable language instead of binary sequences
  - Relative instead of absolute addresses

Assembly Programmer's View

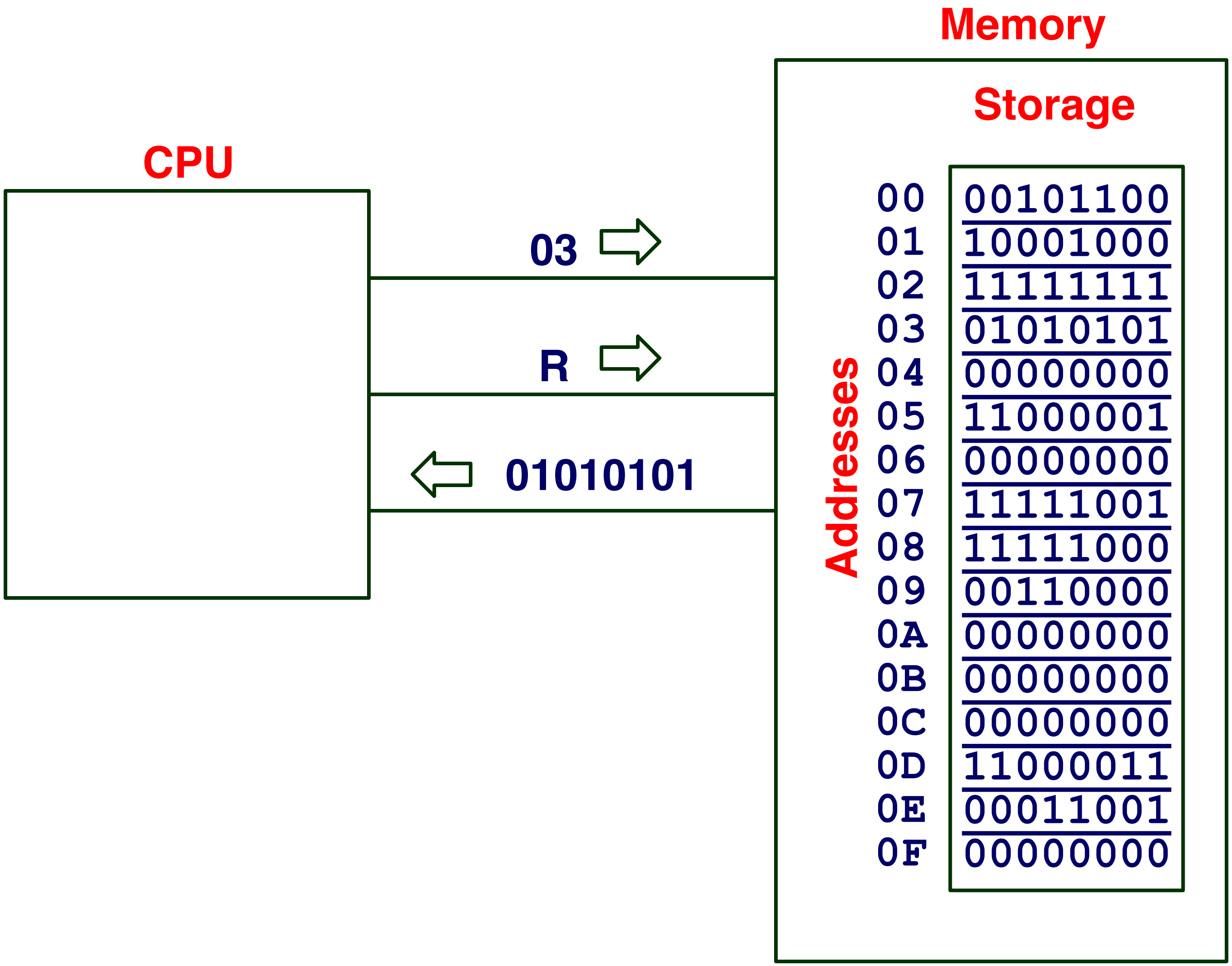


Memory

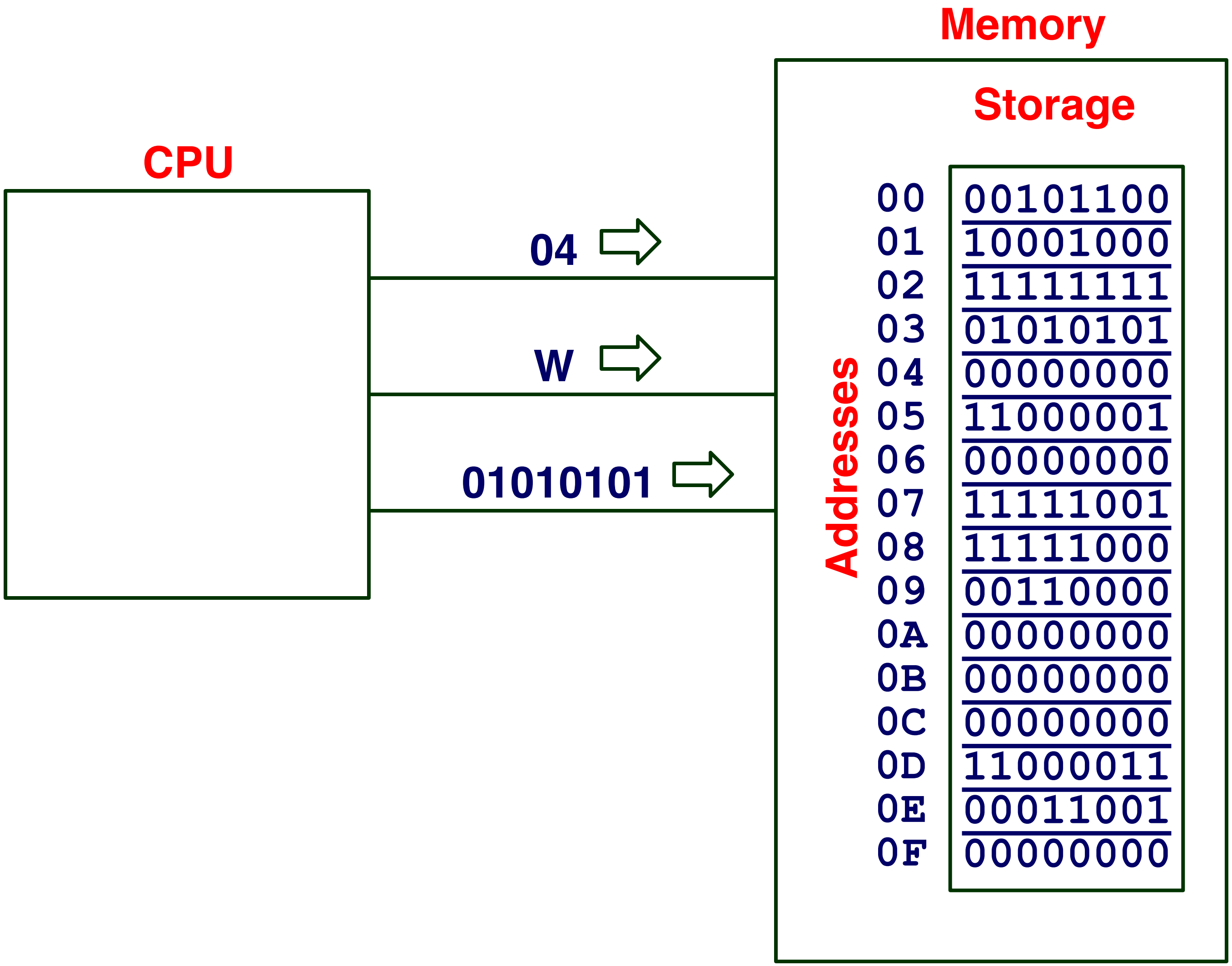


CPU
Address
Command: R/W
Data
Memory
Storage
Addresses
00 00101100
01 10001000
02 11111111
03 01010101
04 00000000
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000

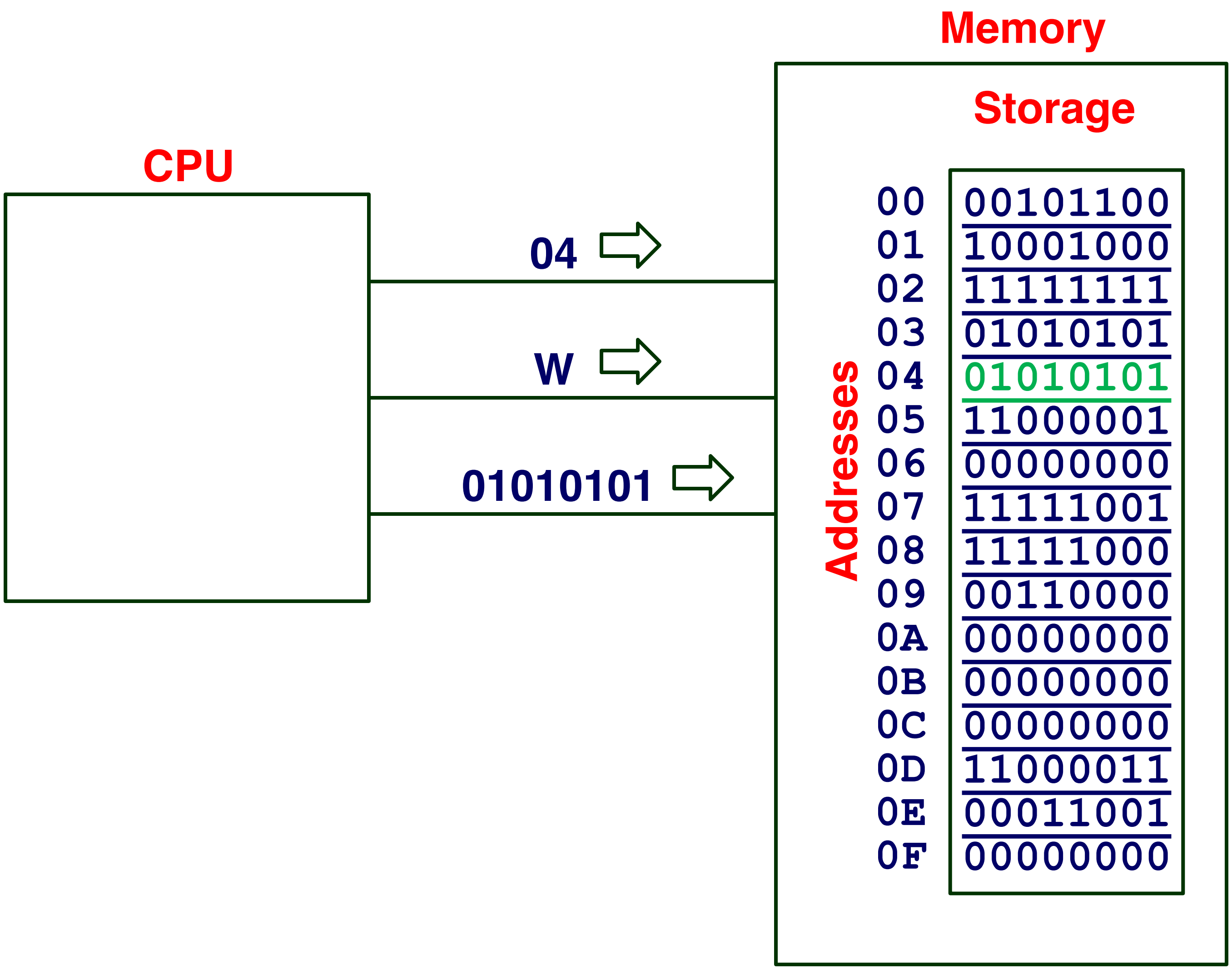
Memory Access: Read



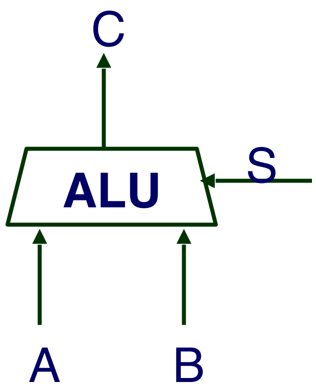
Memory Access: Write



Memory Access: Write



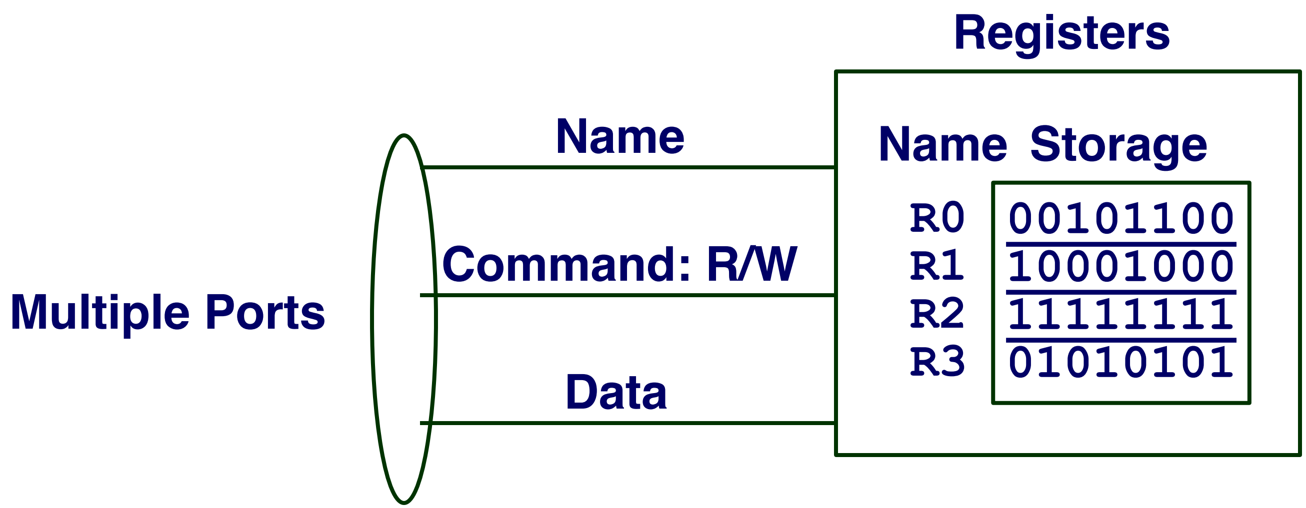
Processor: ALU & Registers



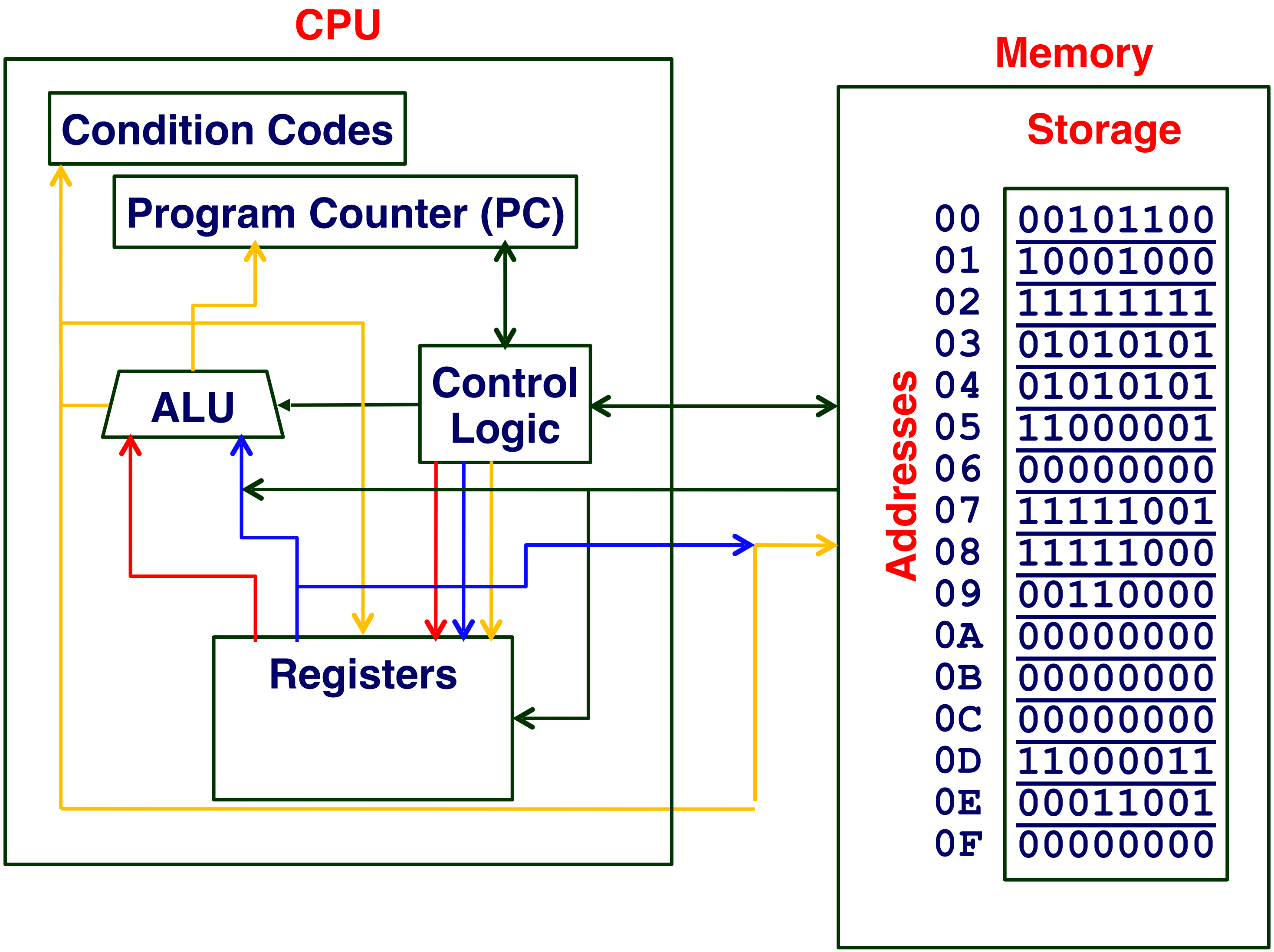
$C = F_S(A, B)$

F includes

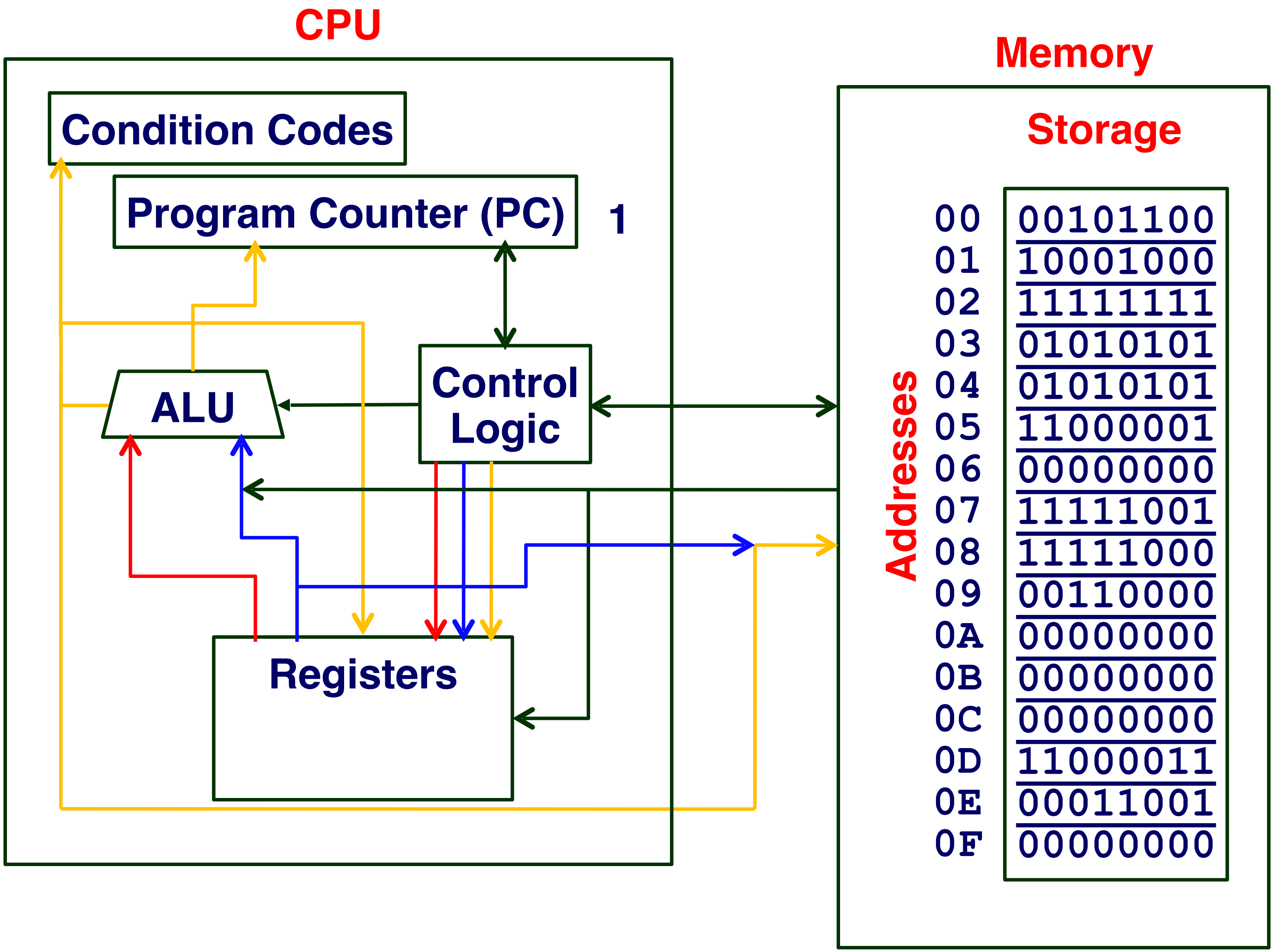
- **Arithmetic: +, -, *, /, ~, etc.**
- **Logical: <, >, =, etc.**



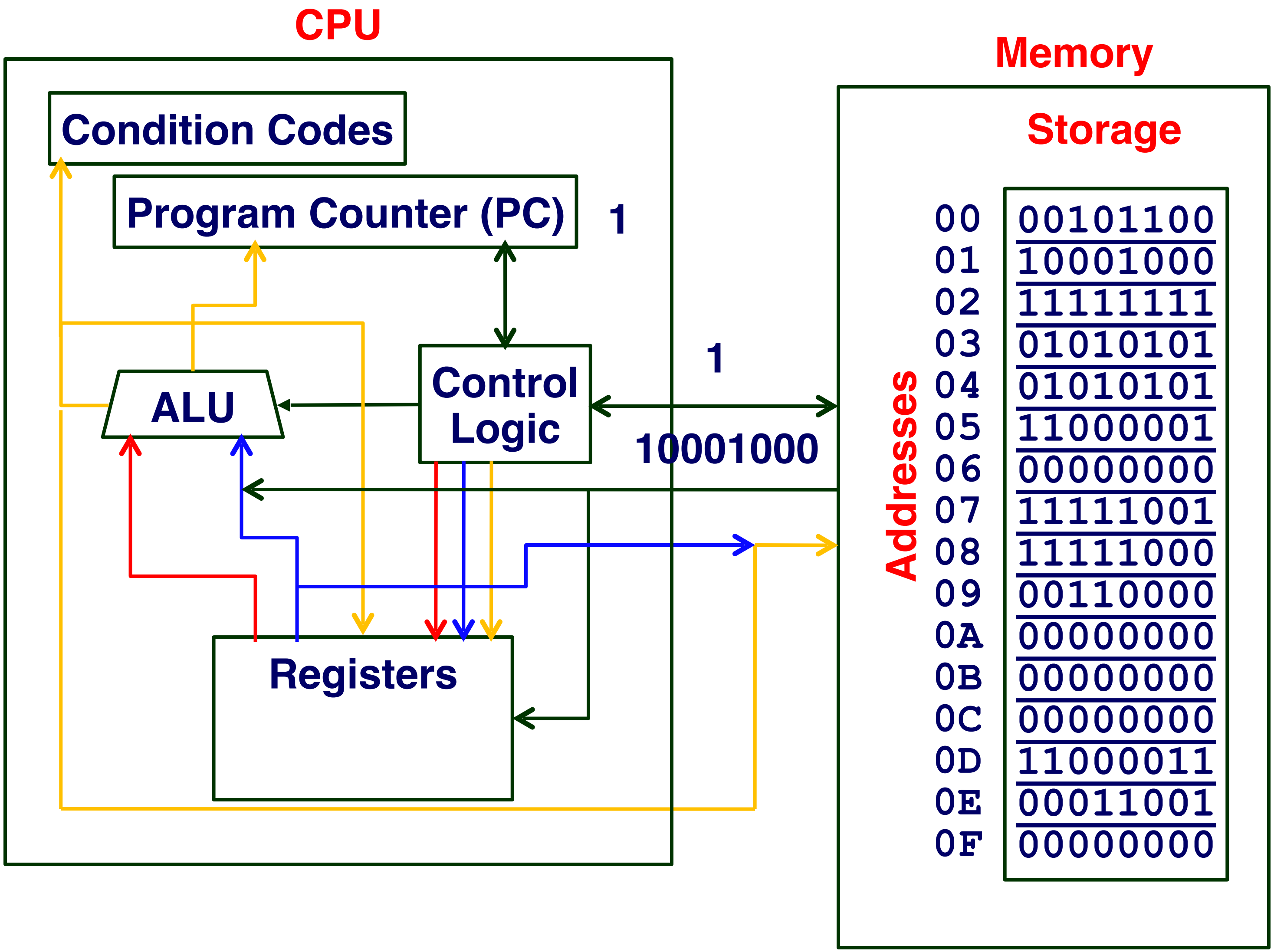
Putting It All Together



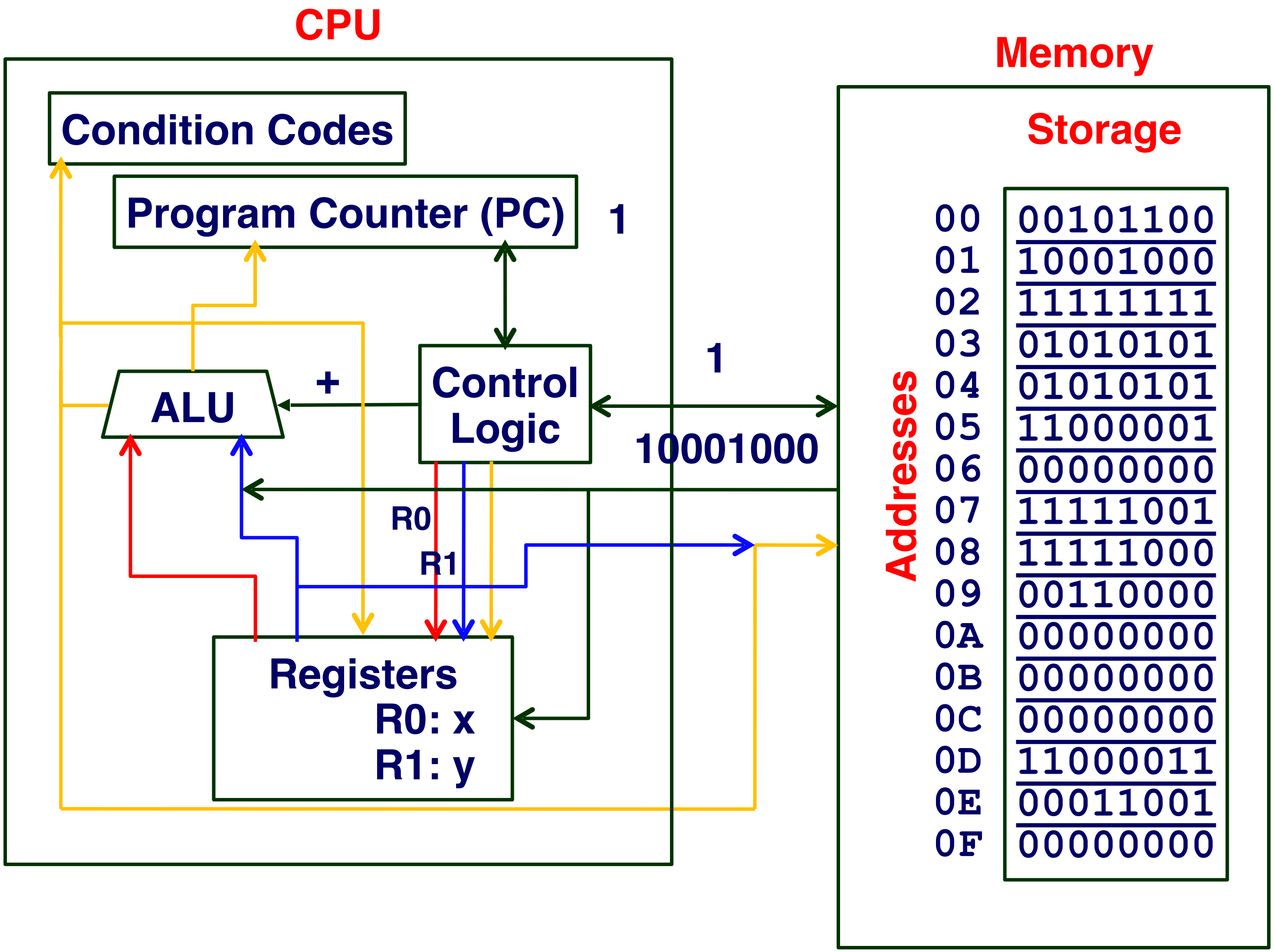
Putting It All Together



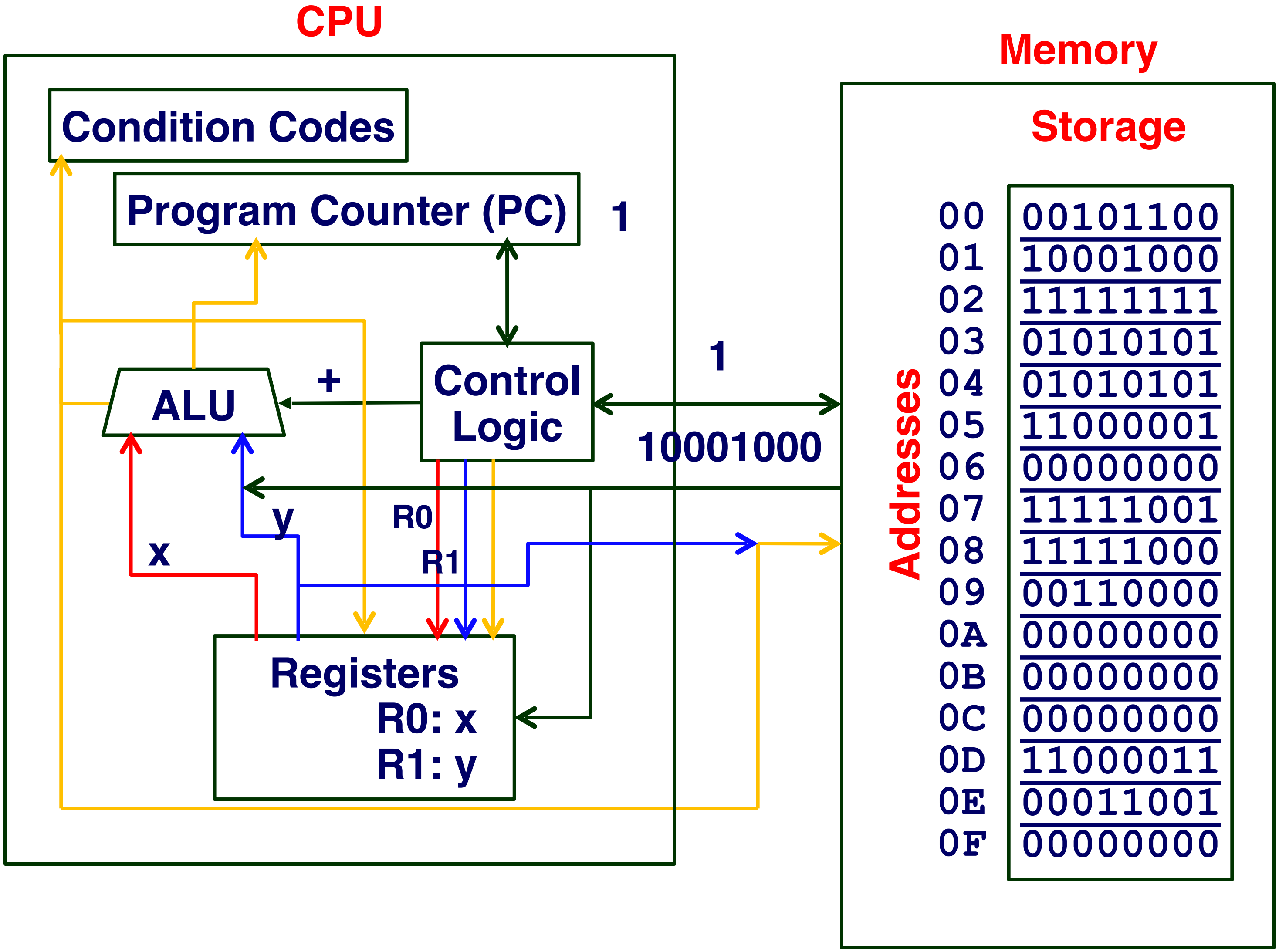
Putting It All Together



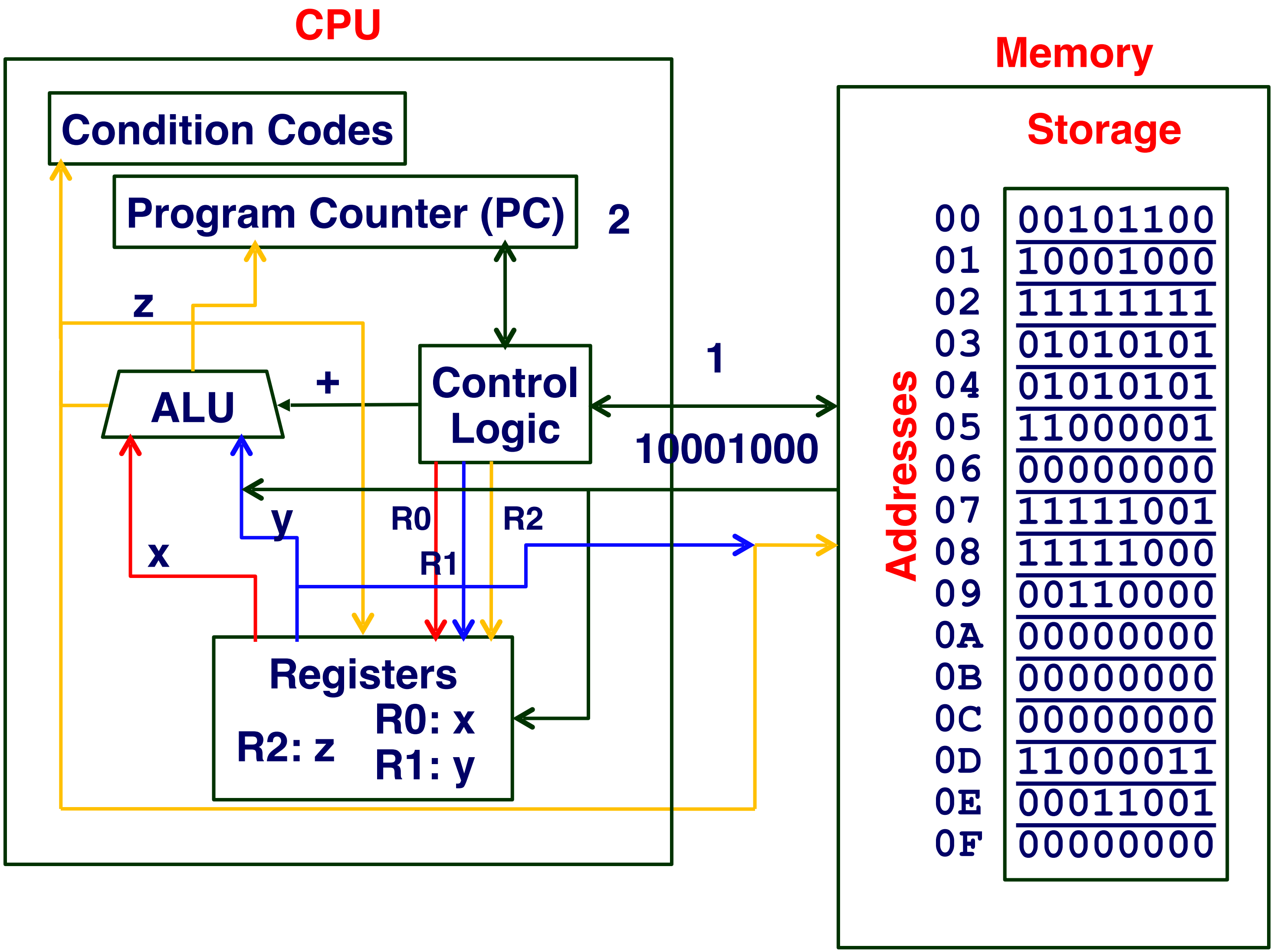
Putting It All Together



Putting It All Together



Putting It All Together



C & Assembly Code

Sample C Code

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

gcc -O1 -m32 –S code.c

Generated Assembly

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

C & Machine Code

Sample C Code

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

gcc -O1 -m32 –c code.c

objdump –d code.o

0000000 <sum>:

0:	55	push	%ebp
1:	89 e5	mov	%esp,%ebp
3:	8b 45 0c	mov	0xc(%ebp),%eax
6:	03 45 08	add	0x8(%ebp),%eax
9:	01 05 00 00 00 00	add	%eax, accum
f:	5d	pop	%ebp
10:	c3	ret	

gdb code.o

```
(gdb) x/100xb sum
```

```
        <sum>: 0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45
0x8   <sum+8>: 0x08 0x01 0x05 0x00 0x00 0x00 0x00 0x5d
0x10 <sum+16>: 0xc3      Cannot access memory at address 0x11
```

Assembly Characteristics

Sequence of simple instructions

Minimal Data Types

- "Integer" data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

No type checking

- Interpretation of data format depends on instruction
- No protection against misinterpretation of data

Assembly Characteristics

3 types of Primitive Operations

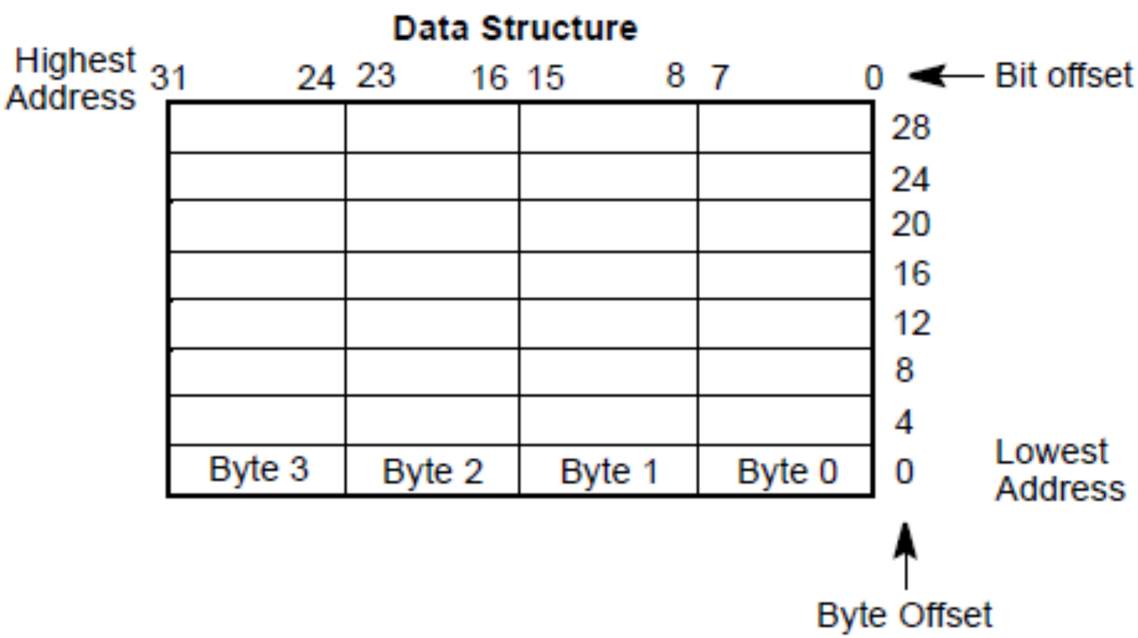
- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

x86 Characteristics

Variable length instructions: 1-15 bytes

Can address memory directly in most instructions

Uses Little-Endian format (Least significant byte in the lowest address)



Instruction Format

General format:

`opcode operands`

Opcode:

- Short mnemonic for instruction's purpose
  - `movb,addl, etc.`

Operands:

- Immediate, register, or memory
- Number of operands command-dependent

Example:

- movl %ebx, (%ecx)

Machine Representation

Remember, each assembly instruction translated to a sequence of 1-15 bytes

Opcode	addressing mode	other bytes

First, the binary representation of the opcode

Second, instruction specifies the addressing mode

- The type of operands (registers or register and memory)
- How to interpret the operands

Some instructions can be single-byte because operands and addressing mode are implicitly specified by the instruction

- E.g., pushl

x86 Registers

General purpose registers are 32 bit

- Although operations can access 8-bits or 16-bits portions

Originally categorized into two groups with different functionality

- Data registers (EAX, EBX, ECX, EDX)
  - Holds operands
- Pointer and Index registers (EBP, ESP, EIP,ESI,EDI)
  - Holds references to addresses as well as indexes

Now, the registers are mostly interchangeable

Segment registers

- Holds starting address of program segments
  - CS, DS, SS, ES

x86 Registers

16 BITS

8 BITS

EAX	AX	AH	AL
ECX	CX	CH	CL
EDX	DX	DH	DL
EBX	BX	BH	BL
ESP --Stack Pointer			
EBP --- Base register of current stack frame			
ESI --- Source index for string operations			
EDI --- Destination index for string operations			

32 BITS

x86 Programming

- Mov instructions to move data from/to memory
  - Operands and registers
- Addressing modes
- Understanding swap
- Arithmetic operations
- Condition codes
- Conditional and unconditional branches
- Loops and switch statements

Data Format

Byte: 8 bits

- E.g., char

Word: 16 bits (2 bytes)

- E.g., short int

Double Word: 32 bits (4 bytes)

- E.g., int, float

Quad Word: 64 bits (8 bytes)

- E.g., double

Instructions can operate on any data size

- **`movl, movw, movb`**
  - **`Move double word, word, byte, respectively`**
- End character specifies what data size to be used

MOV instruction

Most common instruction is data transfer instruction

- Mov SRC, DEST: Move source into destination
- SRC and DEST are operands
- DEST is a register or a location
- SRC can be the contents of register, memory location, constant, or a label.
- If you use gcc, you will see movl <src>, <dest>
- All the instructions in x86 are 32-bit

Used to copy data:

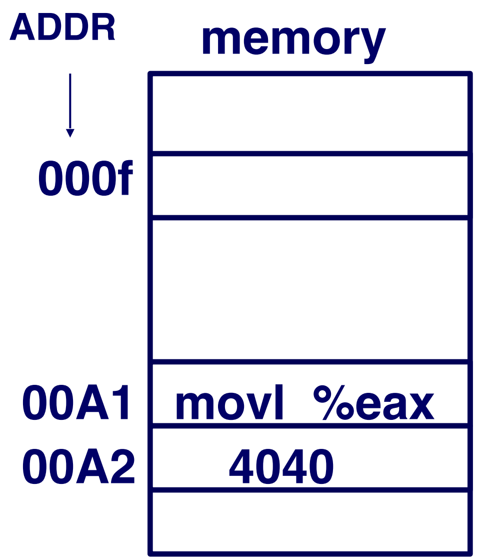
- Constant to register (immediate)
- Memory to register
- Register to memory
- Register to register

Cannot copy memory to memory in a single instruction

Immediate Addressing

Operand is immediate

- Operand value is found immediately following the instruction
- Encoded in 1, 2, or 4 bytes
- $ in front of immediate operand
- E.g., movl $0x4040, %eax



Register Mode Addressing

Use % to denote register

- E.g., %eax

Source operand: use value in specified register

Destination operand: use register as destination for value

Examples:

- movl %eax, %ebx
  - Copy content of %eax to %ebx
- movl $0x4040, %eax → immediate addressing
  - Copy 0x4040 to %eax
- movl %eax, 0x0000f → Absolute addressing
  - Copy content of %eax to memory location 0x0000f

Indirect Mode Addressing

Content of operand is an address

- Designated as parenthesis around operand

Offset can be specified as immediate mode

Examples:

- movl (%ebp), %eax
  - Copy value from memory location whose address is in ebp into eax
- movl -4(%ebp), %eax
  - Copy value from memory location whose address is -4 away from content of ebp into eax

Indexed Mode Addressing

Add content of two registers to get address of operand

- movl (%ebp, %esi), %eax
  - Copy value at (address = ebp + esi) into eax
- movl 8(%ebp, %esi),%eax
  - Copy value at (address = 8 + ebp + esi) into eax

Useful for dealing with arrays

- If you need to walk through the elements of an array
- Use one register to hold base address, one to hold index
  - E.g., implement C array access in a for loop
- Index cannot be ESP

Scaled Indexed Mode Addressing

Multiply the second operand by the scale (1, 2, 4 or 8)

- movl 0x80 (%ebx, %esi, 4), %eax
  - Copy value at (address = ebx + esi*4 + 0x80) into eax

Where is it useful?

Address Computation Examples

%edx	**0xf000**
%ecx	**0x100**

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

`movl` Operand Combinations

movl

	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;`

- Cannot do memory-memory transfers with single instruction

Stack Operations

By convention, %esp is used to maintain a stack in memory

- Used to support C function calls

%esp contains the address of top of stack

Instructions to push (pop) content onto (off of) the stack

- pushl %eax
  - esp = esp – 4
  - Memory[esp] = eax
- popl %ebx
  - ebx = Memory[esp]
  - esp = esp + 4

Where does the stack start? We'll discuss later

Using Simple Addressing Modes

```
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
```

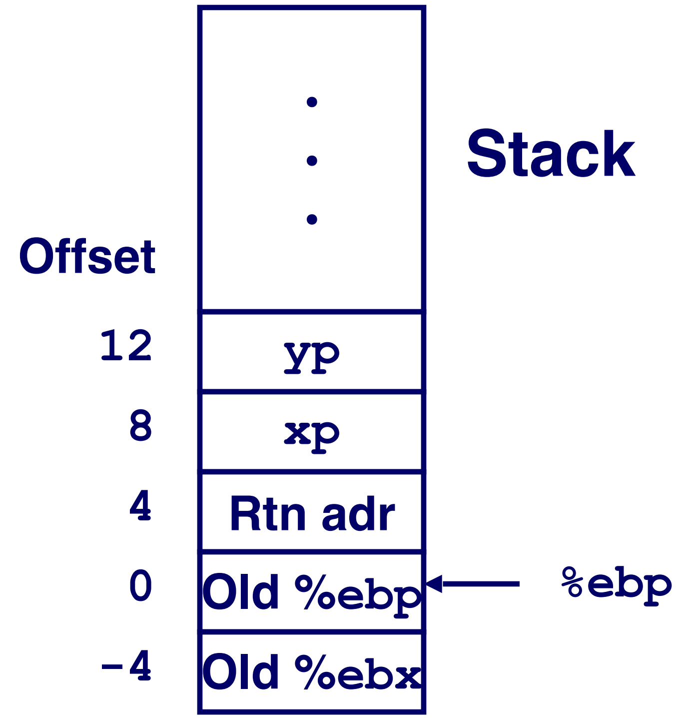
Set Up

Body

Finish

Understanding Swap

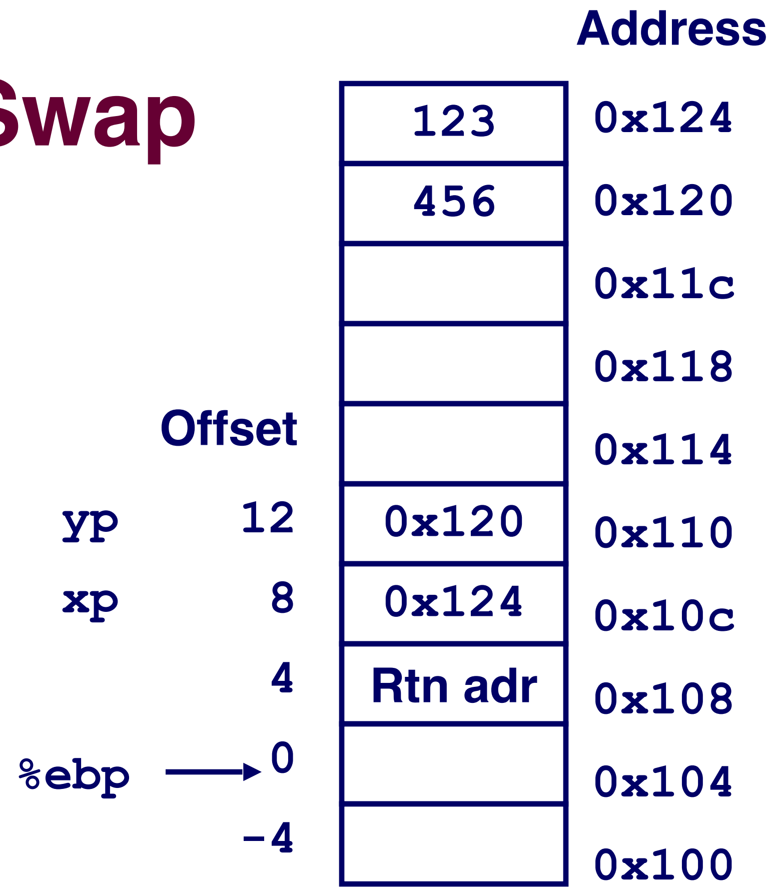
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	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




```
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	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset	Value	Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	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

Register	Value
%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	Rtn adr	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

	Offset		Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	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

Register	Value
%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	Rtn adr	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	Rtn adr	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
		123	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	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
```

Swap in x86-64: 64-bit Registers

rax	eax
rcx	ecx
rdx	edx
rbx	ebx
rsp	esp
rbp	ebp
rsi	esi
rdi	edi

r8
r9
r10
r11
r12
r13
r14
r15

Swap in x86-64 bit

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

```
swap:
    movl (%rdi), %edx
    movl (%rsi), %eax
    movl  %eax, (%rdi)
    movl  %edx, (%rsi)
    retq
```

Arguments passed in registers

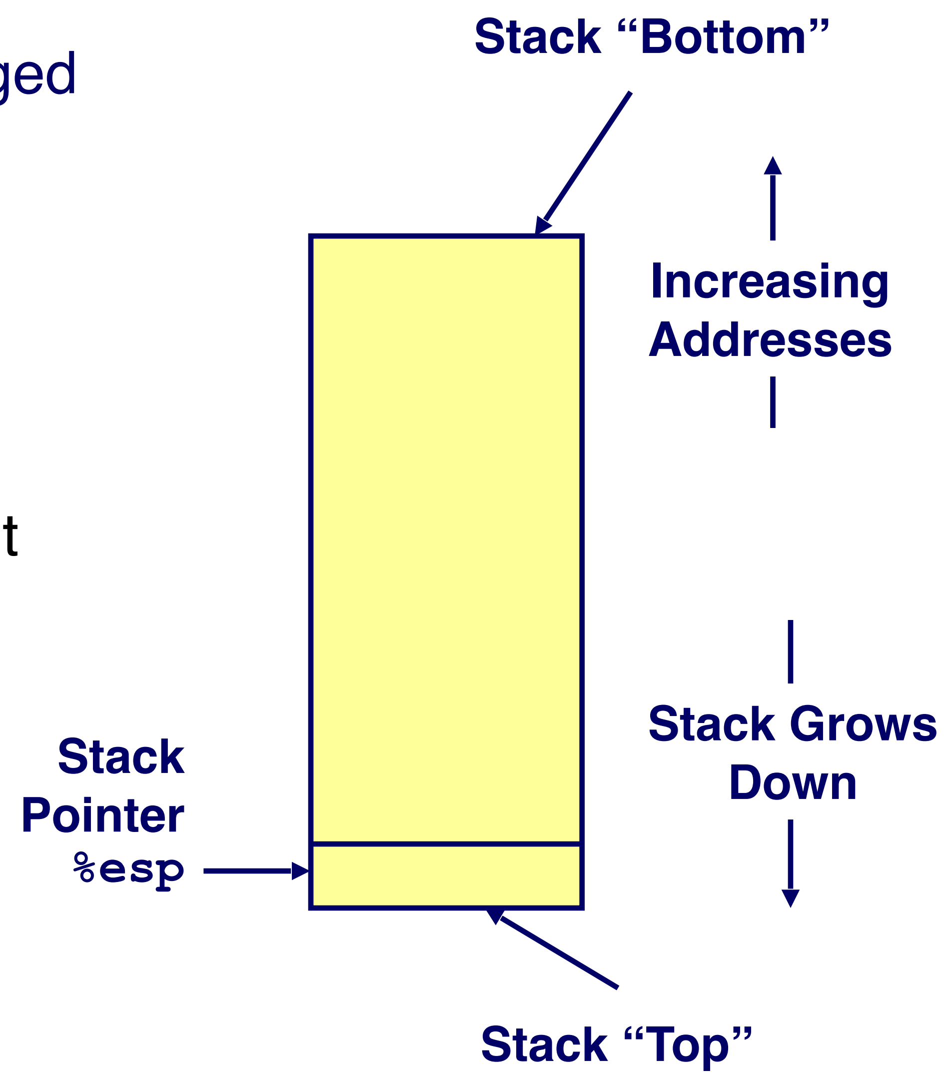
- First, xp in rdi and yp in rsi
- 64-bit pointers, data values are 32-bit ints, so uses eax/edx

No stack operations

What happens with long int?

IA32 Stack

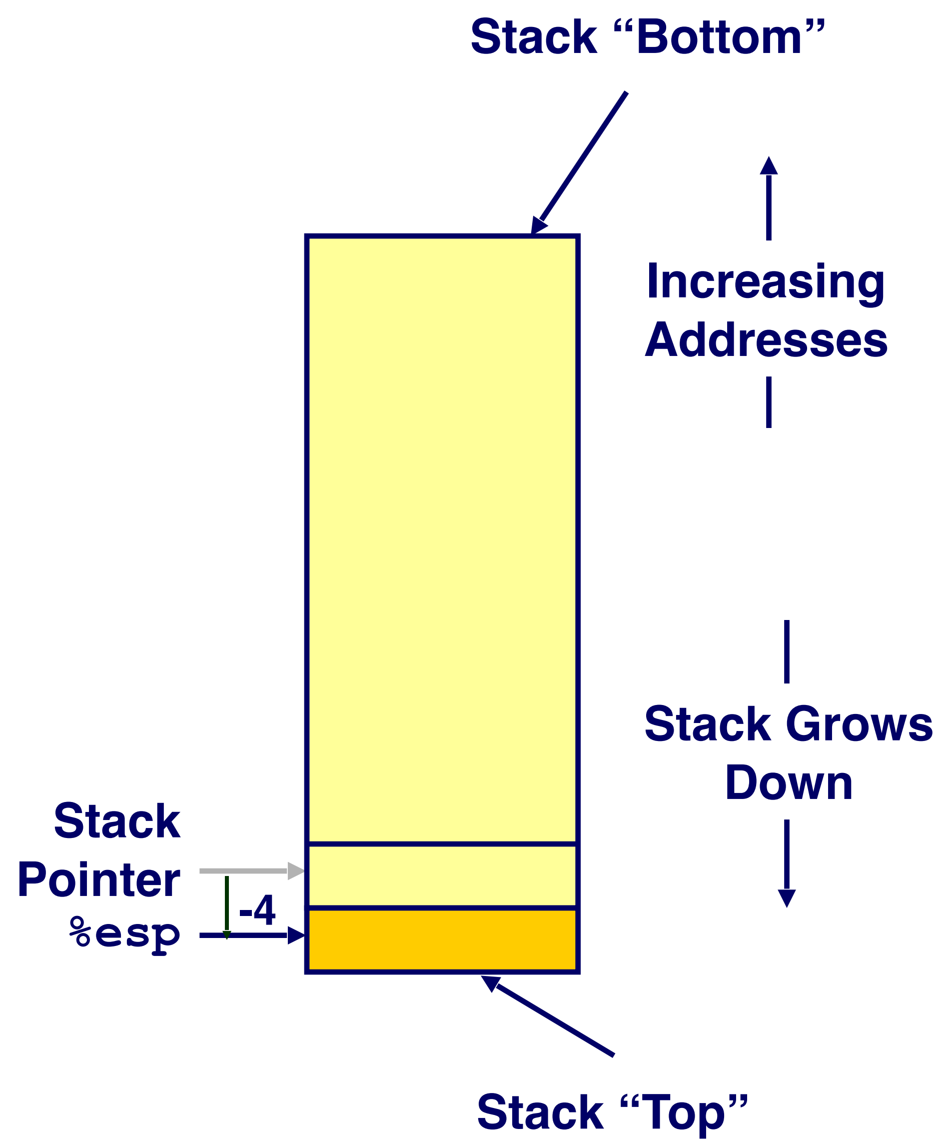
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of top element



IA32 Stack Pushing

Pushing

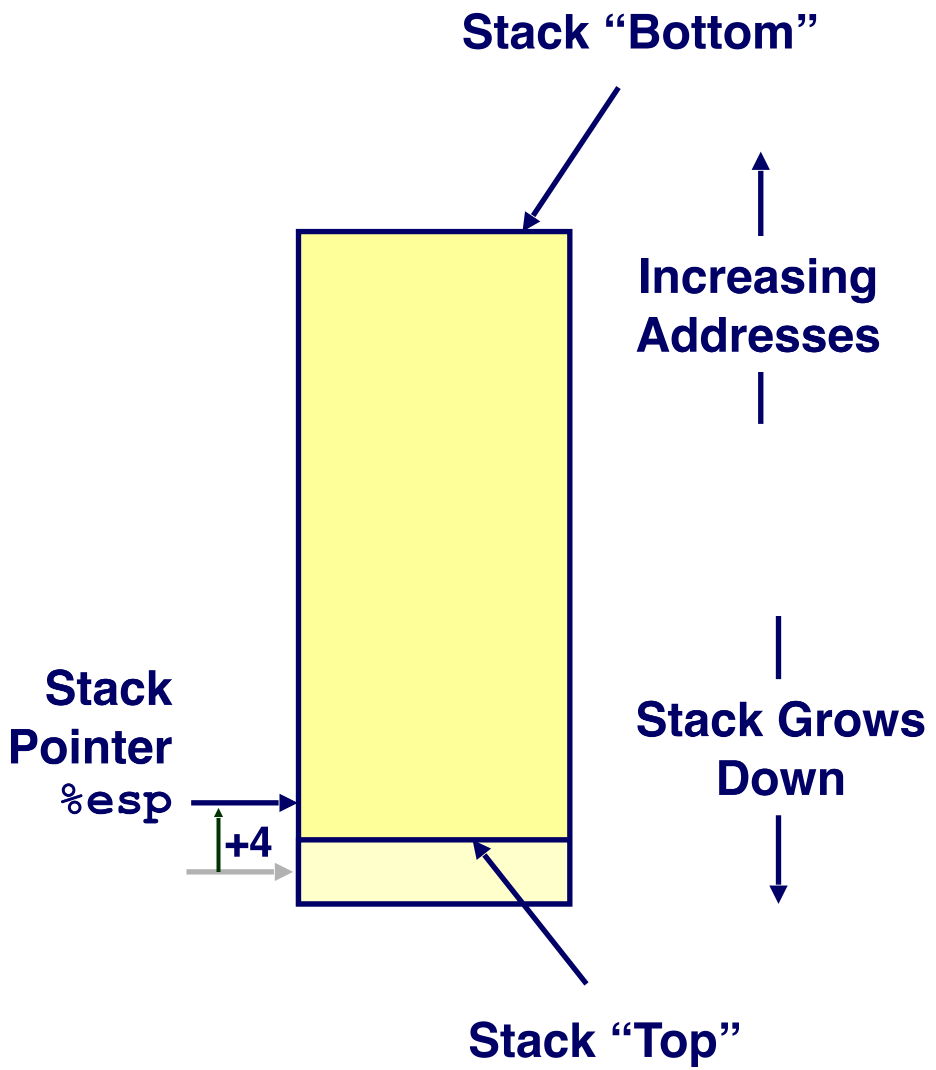
- `pushl` *Src*
- Fetch operand at *Src*
- Decrement `%esp` by 4
- Write operand at address given by `%esp`



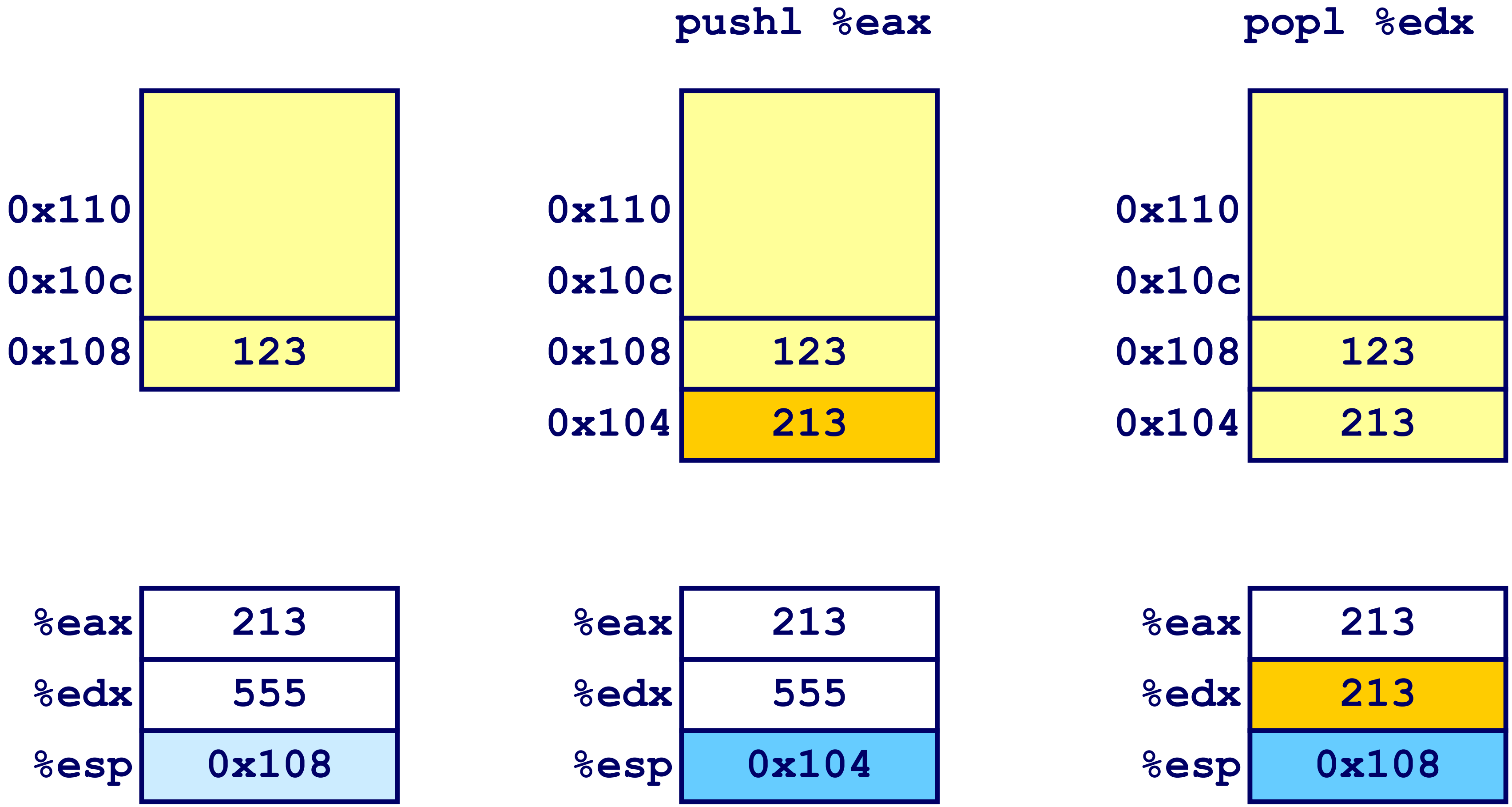
IA32 Stack Popping

Popping

- `popl` *Dest*
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to *Dest*



Stack Operation Examples



Procedure Control Flow

- Use stack to support procedure call and return

Procedure call:

`call` *`label`* Push return address on stack; Jump to *`label`*

Return address value

- Address of instruction beyond `call`
- Example from disassembly

```
804854e:    e8 3d 06 00 00    call   8048b90 <main>
8048553:    50                pushl  %eax
```

  - Return address = `0x8048553`

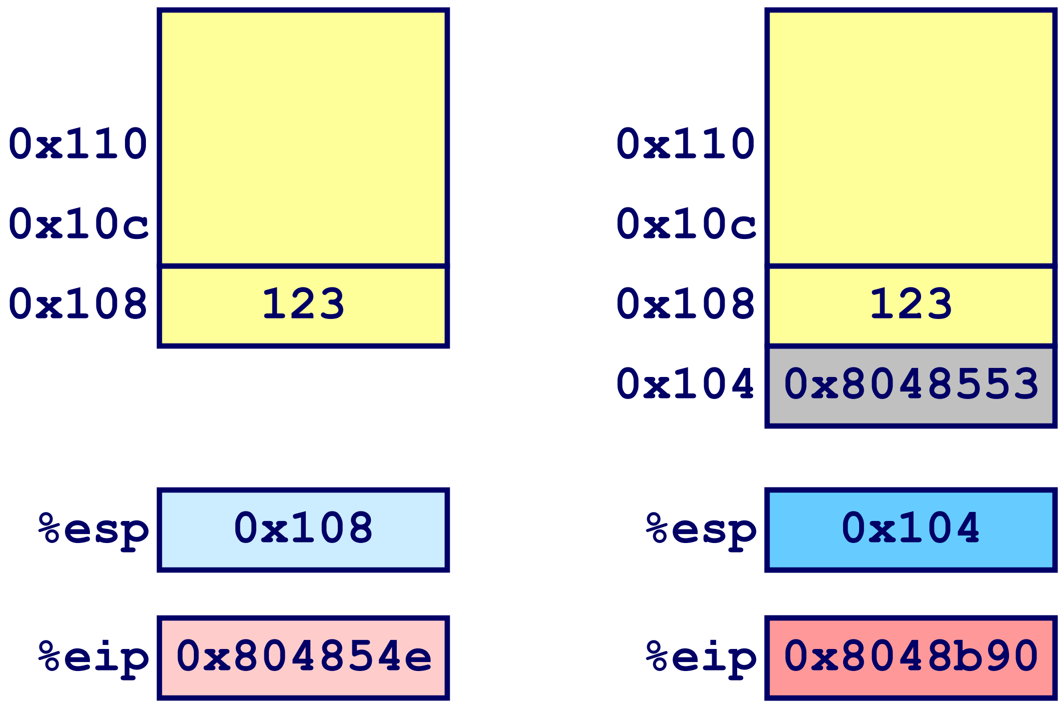
Procedure return:

- `ret` Pop address from stack; Jump to address

Procedure Call Example

```
804854e:  e8 3d 06 00 00      call    8048b90 <main>
8048553:  50                  pushl   %eax
```

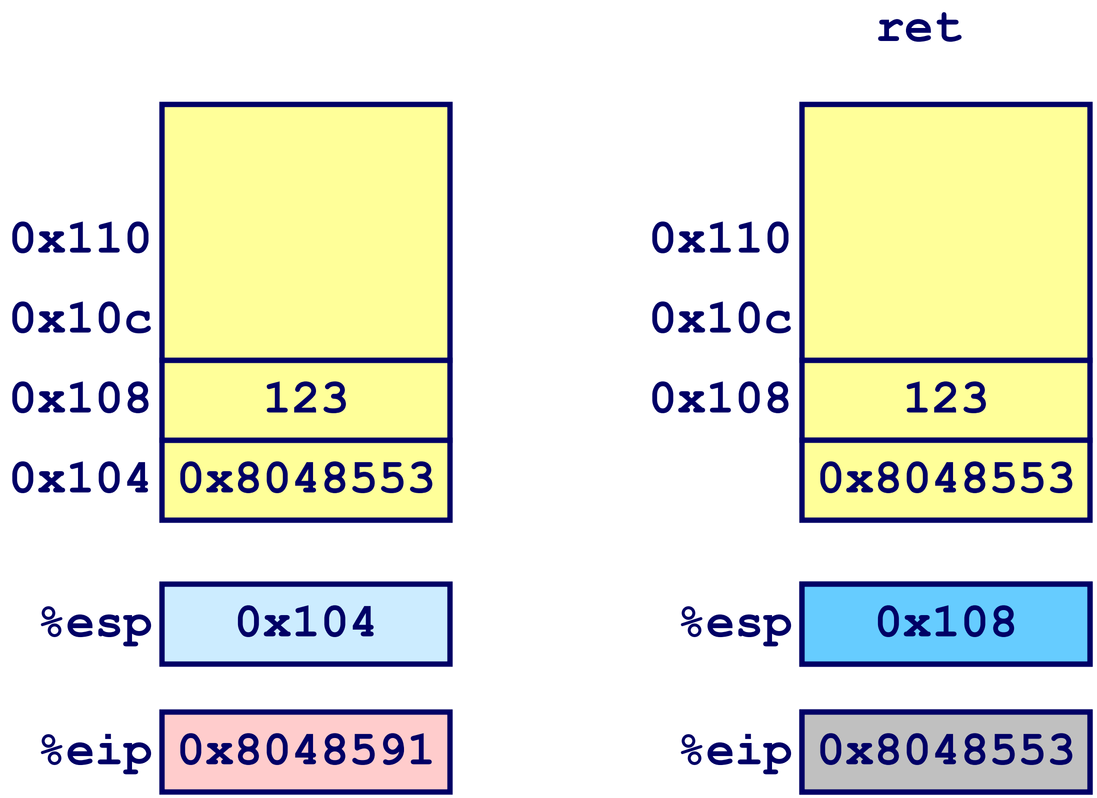
`call 8048b90`



`%eip` **is program counter**

Procedure Return Example

```
8048591:   c3                        ret
```



`%eip` **is program counter**

Address Computation Instruction

leal: compute address using addressing mode without accessing memory

leal src, dest

- src is address mode expression
- Set dest to address specified by src

Use

- Computing address without doing memory reference
  - E.g., translation of p = &x[i];

Example:

- leal 7(%edx, %edx, 4), %eax
  - eax = 4*edx + edx + 7 = 5*edx + 7

Some Arithmetic Operations

Instruction		Computation
`addl`	*Src,Dest*	*Dest* = *Dest* + *Src*
`subl`	*Src,Dest*	*Dest* = *Dest* - *Src*
`imull`	*Src,Dest*	*Dest* = *Dest* * *Src*
`sall`	*Src,Dest*	*Dest* = *Dest* << *Src (left shift)*
`sarl`	*Src,Dest*	*Dest* = *Dest* >> *Src (right shift)*
`xorl`	*Src,Dest*	*Dest* = *Dest* ^ *Src*
`andl`	*Src,Dest*	*Dest* = *Dest* & *Src*
`orl`	*Src,Dest*	*Dest* = *Dest* | *Src*

Some Arithmetic Operations

Instruction	Computation
`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* - 1
`negl` *Dest*	*Dest* = - *Dest*
`notl` *Dest*	*Dest* = ~ *Dest*

Using `leal` for Arithmetic Expressions

```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
    movl %esp,%ebp

    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

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

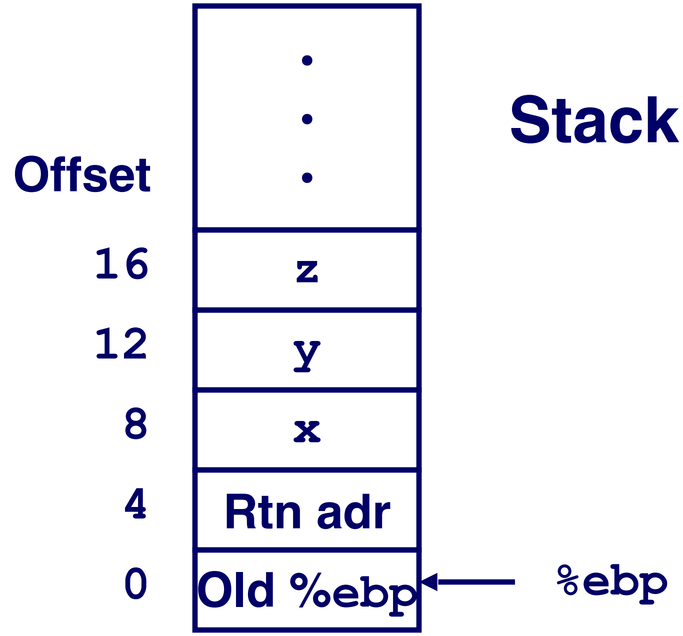
Set Up

Body

Finish

Understanding `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;
}
```




```
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)
```

Understanding `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;
}
```

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

Another Example

```
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

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

```
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
```

Mystery Function

What does the following piece of code do?

A. Add two variables
B. Subtract two variables
C. Swap two variables
D. No idea

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

What does this function do?

```
.globl foo
        .type   foo, @function
foo:
        pushl   %ebp

        movl    %esp, %ebp

        movl    16(%ebp), %eax

        imull   12(%ebp), %eax

        addl    8(%ebp), %eax

        popl    %ebp

        ret
```

Control Flow/Conditionals

How do we represent conditionals in assembly?

A conditional branch can implement all control flow constructs in higher level language

- Examples: if/then, while, for

A unconditional branch for constructs like break/ continue

Condition Codes

Single Bit Registers

`CF`	Carry Flag	`SF`	Sign Flag
`ZF`	Zero Flag	`OF`	Overflow Flag

Can be set either implicitly or explicitly.

- Implicitly by almost all logic and arithmetic operations
- Explicitly by specific comparison operations

Not Set by `leal` instruction

- Intended for use in address computation only

Jumping

jX Instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Condition Codes

Implicitly Set 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)`

Setting Condition Codes (cont.)

Explicit Setting by Compare Instruction

`cmpl` *Src2,Src1*

- `cmpl b,a` like computing `a-b` without setting destination
- NOTE: The operands are reversed. Source of confusion
- 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 Condition Codes (cont.)

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`

Conditional Branch Example

```
_max:
        pushl %ebp              Set
        movl %esp,%ebp          Up

        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle L9                  Body
        movl %edx,%eax
L9:

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

Conditional Branch Example

```
int max(int x, int y)
{
  if (x <= y)
    return y;
  else
    return x;
}
```

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

        movl 8(%ebp),%edx
        movl 12(%ebp),%eax
        cmpl %eax,%edx
        jle L9
        movl %edx,%eax
L9:

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

Set Up

Body

Finish

Conditional Branch Example (Cont.)

```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)
    goto done;
  rval = x;
done:
  return rval;
}
```

```
int max(int x, int y)
{
  if (x <= y)
    return y;
  else
    return x;
}
```

- C allows “goto” as means of transferring control
  - Closer to machine-level programming style
- Generally considered bad coding style

```
   movl 8(%ebp),%edx    # edx = x
   movl 12(%ebp),%eax   # eax = y
   cmpl %eax,%edx       # x : y
   jle L9               # if <= goto L9
   movl %edx,%eax       # eax = x
L9:                  # Done:
```

Skipped when x ≤ y

Mystery Function

```
.LC0:
        .string "%d”
        .text
.globl foo
        .type   foo, @function
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        leal    -12(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    scanf
        cmpl    $4, -12(%ebp)
        je      .L3
        call    explode_bomb
.L3:
        leave
        .p2align 4,,3
        ret
```

“Do-While” Loop Example

C Code

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

"Do-While" Loop Example

C Code

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Use backward branch to continue looping
- Only take branch when "while" condition holds

"Do-While" Loop Compilation

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

Assembly

```
_fact_goto:
  pushl %ebp          # Setup
  movl %esp,%ebp      # Setup
  movl $1,%eax        # eax = 1
  movl 8(%ebp),%edx   # edx = x

L11:
  imull %edx,%eax     # result *= x
  decl %edx           # x--
  cmpl $1,%edx        # Compare x : 1
  jg L11              # if > goto loop

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

Registers

```
%edx  x
%eax  result
```

General "Do-While" Translation

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- *Body* can be any C statement
  - Typically compound statement:

```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

- *Test* is expression returning integer
  = 0 interpreted as false ≠0 interpreted as true

"While" Loop Example #1

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

Actual "While" Loop Translation

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

Goto Version

```
int fact_while_goto2
  (int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

- Uses same inner loop as do-while version
- Guards loop entry with extra test

General "While" Translation

C Code

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

Switch Statements

```
typedef enum
 {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

Implementation Options

- Series of conditionals
  - Good if few cases
  - Slow if many
- Jump Table
  - Lookup branch target
  - Avoids conditionals
  - Possible when cases are small integer constants
- GCC
  - Picks one based on case structure
- Bug in example code
  - No default given

Jump Table Structure

Switch Form

```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

```
jtab:  Targ0
       Targ1
       Targ2
         •
         •
         •
       Targn-1
```

Jump Targets

```
Targ0:   Code Block 0
Targ1:   Code Block 1
Targ2:   Code Block 2
           •
           •
           •
Targn-1: Code Block n–1
```

Approx. Translation

```
target = JTab[op];
goto *target;
```

Switch Statement Example

Branching Possibilities

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

Enumerated Values

```
ADD    0
MULT   1
MINUS  2
DIV    3
MOD    4
BAD    5
```

Setup:

```
unparse_symbol:
   pushl %ebp              # Setup
   movl %esp,%ebp          # Setup
   movl 8(%ebp),%eax       # eax = op
   cmpl $5,%eax            # Compare op : 5
   ja .L49                 # If > goto done
   jmp *.L57(,%eax,4)      # goto Table[op]
```

Assembly Setup Explanation

Table Structure

- Each target requires 4 bytes
- Base address at `.L57`

Jumping

`jmp .L49`

- Jump target is denoted by label `.L49`

`jmp *.L57(,%eax,4)`

- Start of jump table denoted by label `.L57`
- Register `%eax` holds `op`
- Must scale by factor of 4 to get offset into table
- Fetch target from effective Address `.L57 + op*4`

Jump Table

Table Contents

```
.section .rodata
   .align 4
.L57:
  .long .L51 #Op = 0
  .long .L52 #Op = 1
  .long .L53 #Op = 2
  .long .L54 #Op = 3
  .long .L55 #Op = 4
  .long .L56 #Op = 5
```

Enumerated Values

```
ADD    0
MULT   1
MINUS  2
DIV    3
MOD    4
BAD    5
```

Targets & Completion

```
.L51:
    movl $43,%eax # '+'
    jmp .L49
.L52:
    movl $42,%eax # '*'
    jmp .L49
.L53:
    movl $45,%eax # '-'
    jmp .L49
.L54:
    movl $47,%eax # '/'
    jmp .L49
.L55:
    movl $37,%eax # '%'
    jmp .L49
.L56:
    movl $63,%eax # '?'
    # Fall Through to .L49
```

Switch Statement Completion

```
.L49:                        # Done:
    movl %ebp,%esp           # Finish
    popl %ebp                # Finish
    ret                      # Finish
```

Puzzle

- What value returned when `op` is invalid?

Answer

- Register `%eax` set to `op` at beginning of procedure
- This becomes the returned value

Advantage of Jump Table

- Can do *k*-way branch in *O*(1) operations

Reading Condition Codes

SetX Instructions

- Set single byte based on combinations of condition codes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Reading Condition Codes (Cont.)

SetX Instructions

- Set single byte based on combinations of condition codes
- One of 8 addressable byte registers
  - Embedded within first 4 integer registers
  - Does not alter remaining 3 bytes
  - Typically use `movzbl` to finish job

%eax	%ah	%al
%edx	%dh	%dl
%ecx	%ch	%cl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

```
int gt (int x, int y) {
  return x > y;
}
```

Body

```
movl 12(%ebp),%eax   # eax = y
cmpl %eax,8(%ebp)    # Compare x : y
setg %al             # al = x > y
movzbl %al,%eax      # Zero rest of %eax
```

Note inverted ordering!

Can you write the C code for this assembly?

```
.globl test
        .type   test, @function
test:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    movl    8(%ebp), %edx
    movl    12(%ebp), %ecx
    movl    $1, %eax
    cmpl    %ecx, %edx
    jge     .L3

.L6:
    leal    (%edx,%ecx), %ebx
    imull   %ebx, %eax
    addl    $1, %edx
    cmpl    %edx, %ecx
    jg .L6
.L3:
    popl    %ebx
    popl    %ebp
    ret
```

What does this function do?

What is the C code?

Stack-Based Languages

Languages that Support Recursion

- e.g., C, Pascal, Java
- Code must be "*Reentrant*"
  - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
  - Arguments, local variables, return pointer

Stack Discipline

- State for given procedure needed for limited time
  - From when called to when return
- Callee returns before caller does

Stack Allocated in *Frames (Activation records)*

- state for single procedure instantiation

Call Chain Example

Code Structure

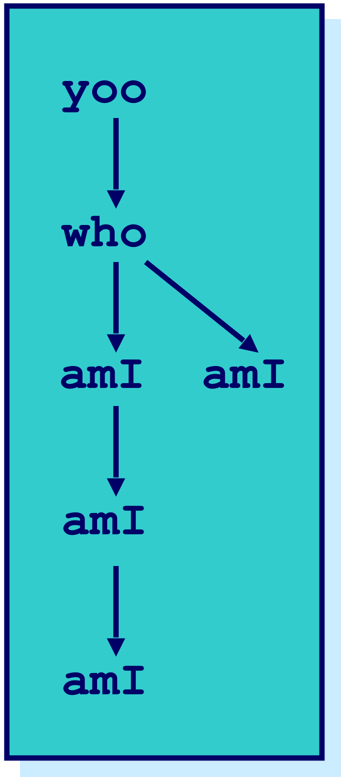
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

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

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

- **Procedure `amI` recursive**

Call Chain



Stack Frames

Contents

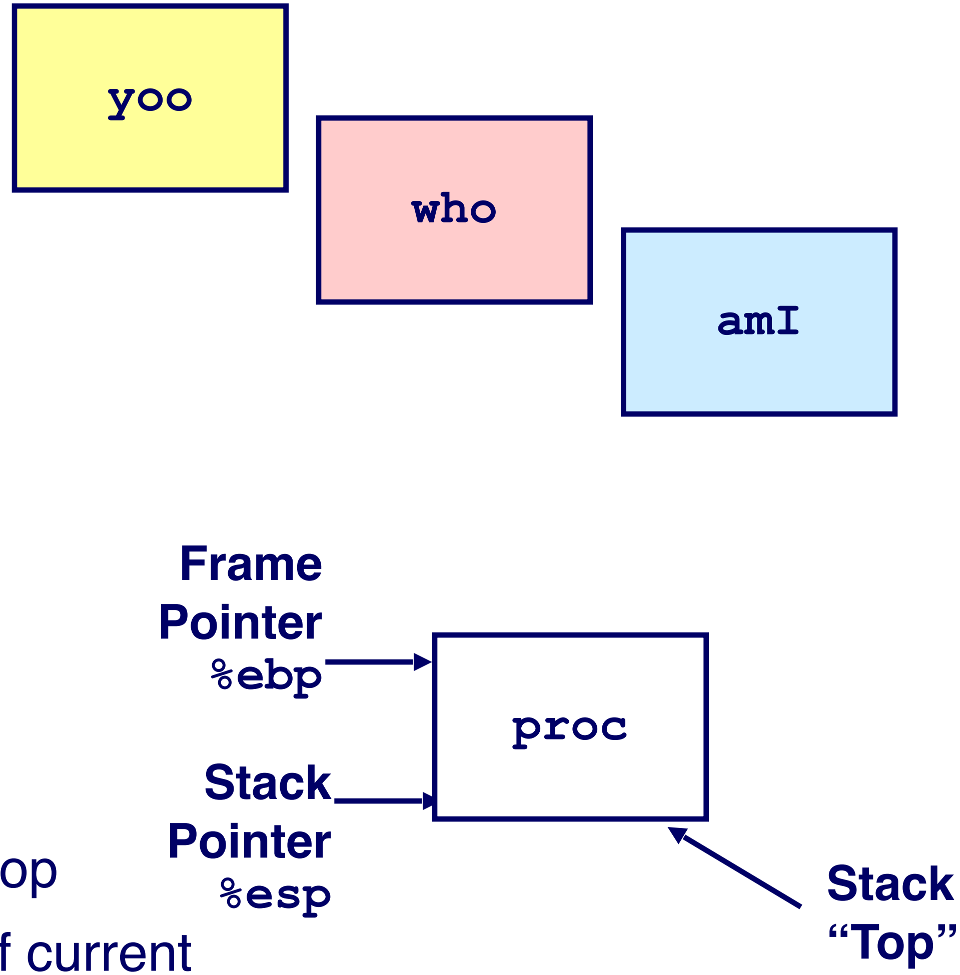
- Local variables, return value
- Temporary space

Management

- Space allocated when enter procedure
  - "Set-up" code
- Deallocated when return
  - "Finish" code

Pointers

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

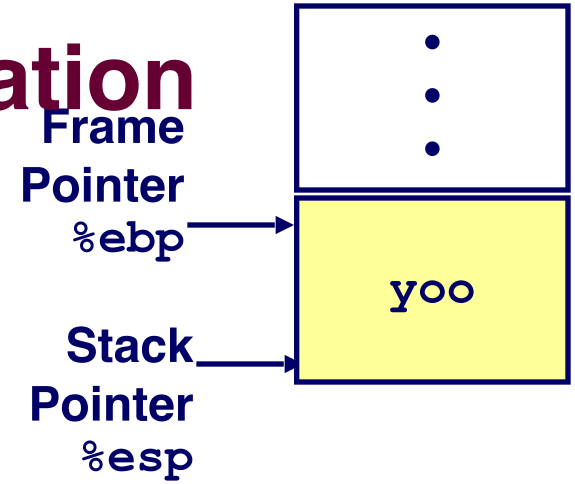


Stack Operation

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

Call Chain

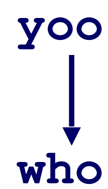
yoo

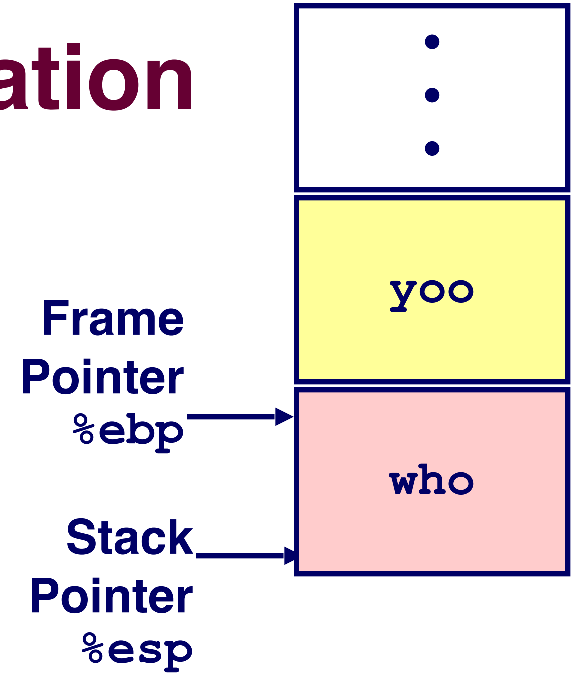


Stack Operation

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

Call Chain

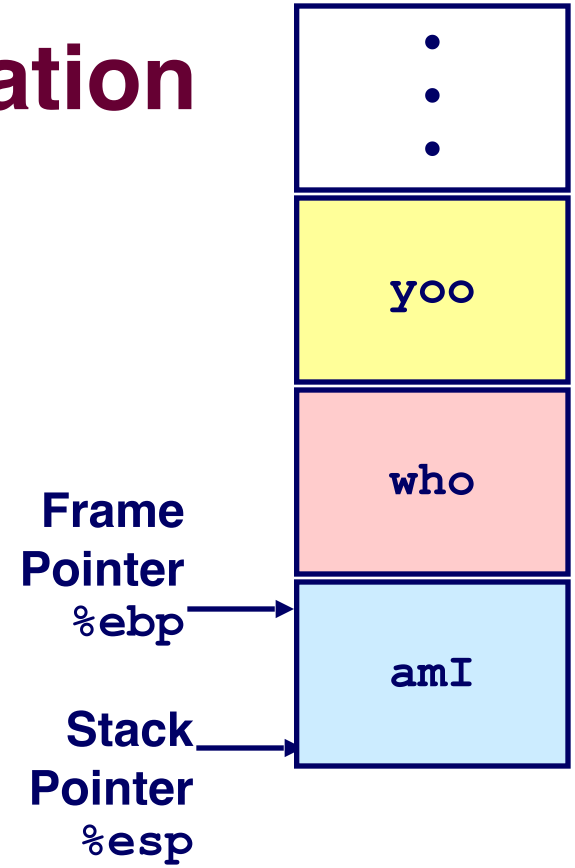




Stack Operation

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

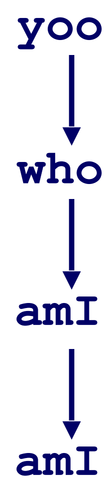
Call Chain

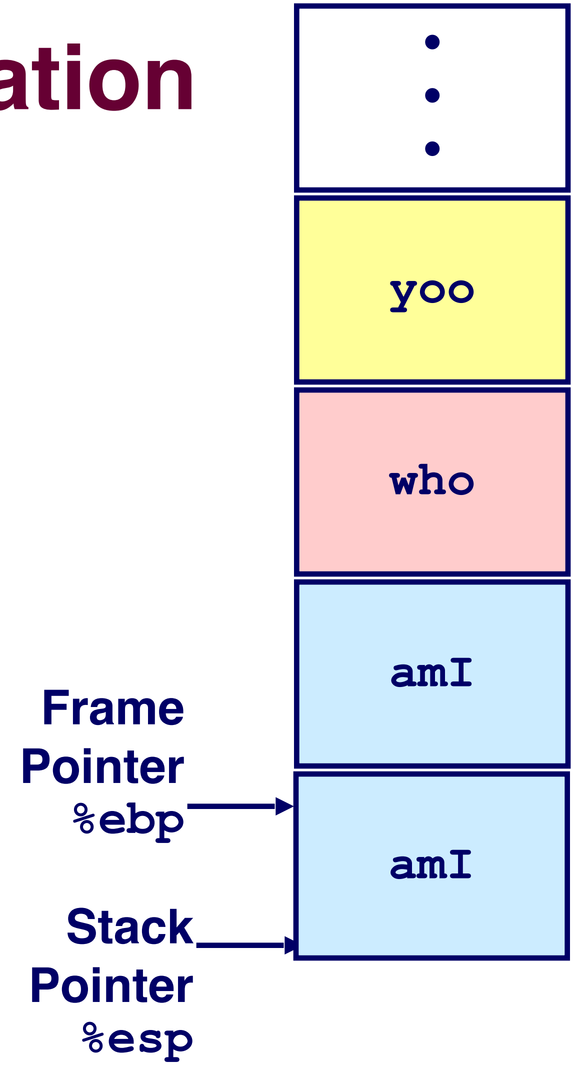


Stack Operation

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

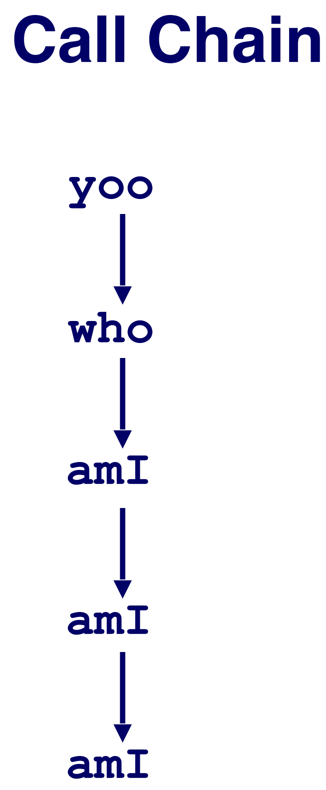
Call Chain

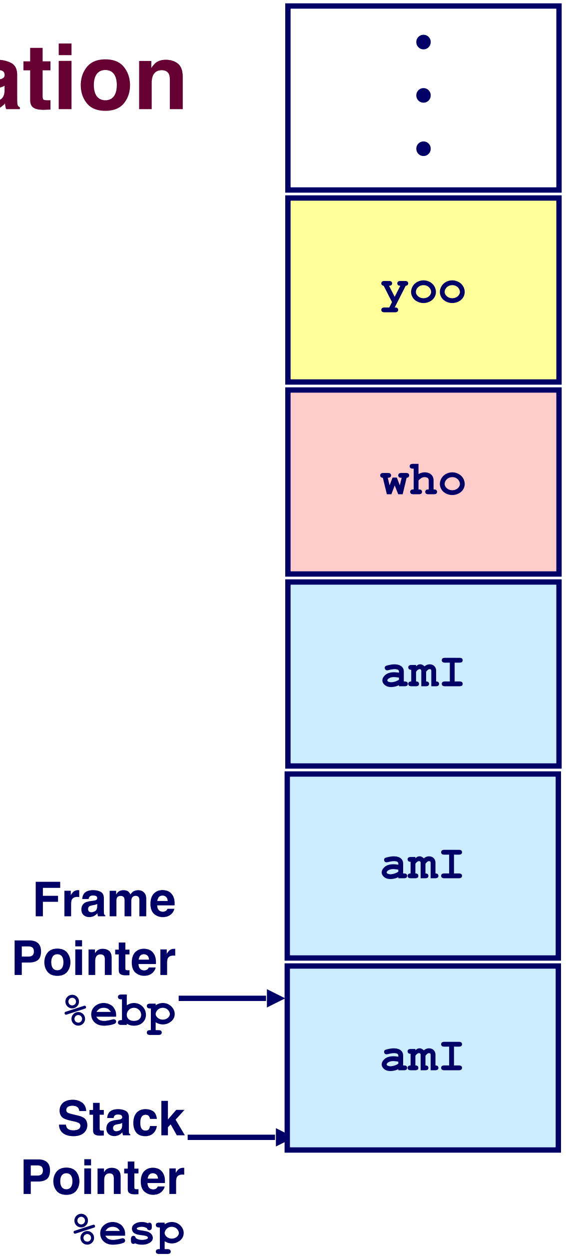




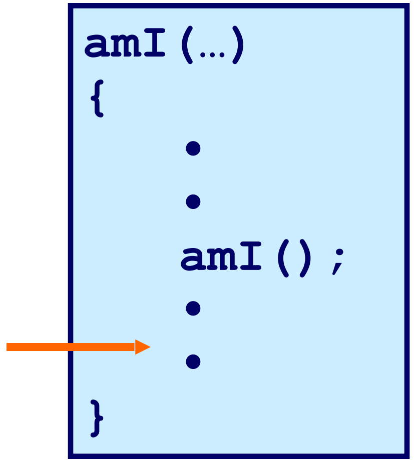
Stack Operation

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

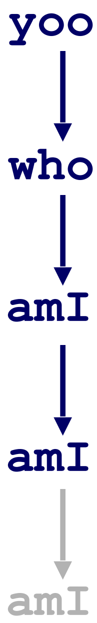


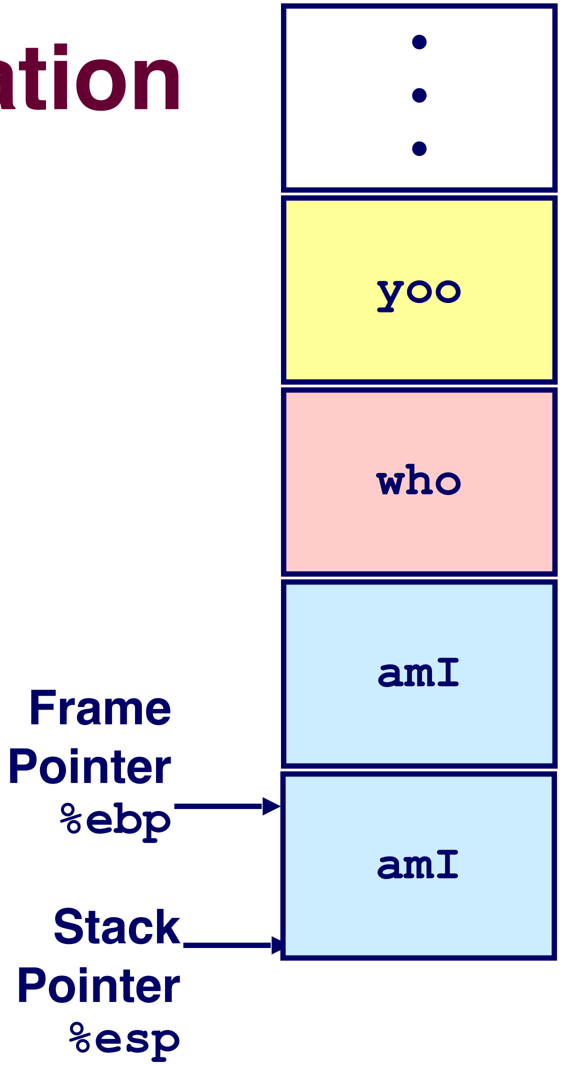


Stack Operation

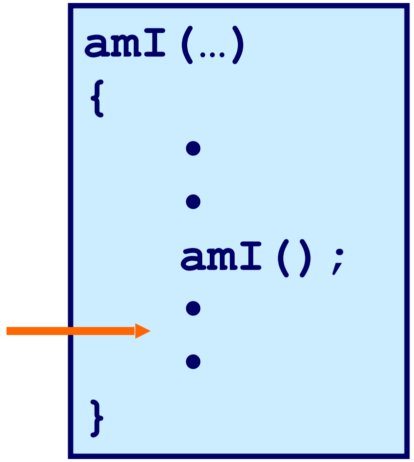


Call Chain

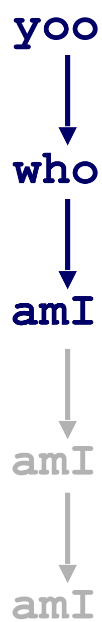


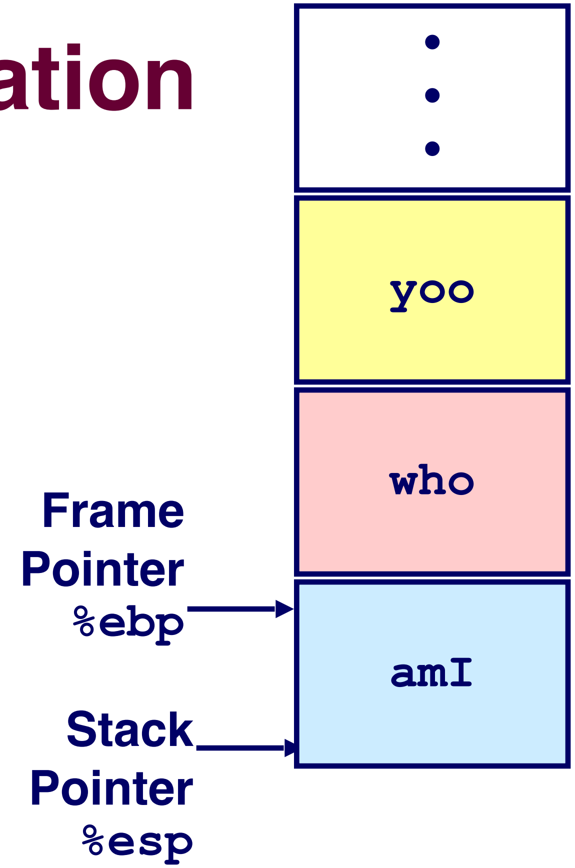


Stack Operation



Call Chain

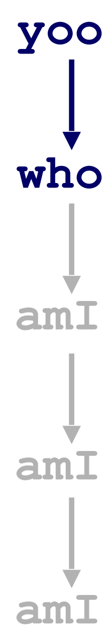


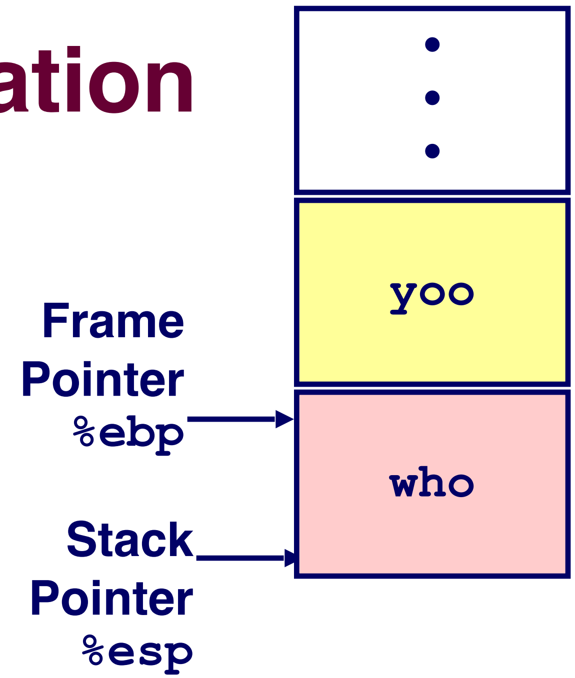


Stack Operation

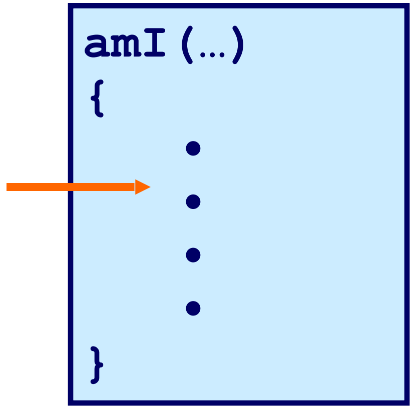
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain

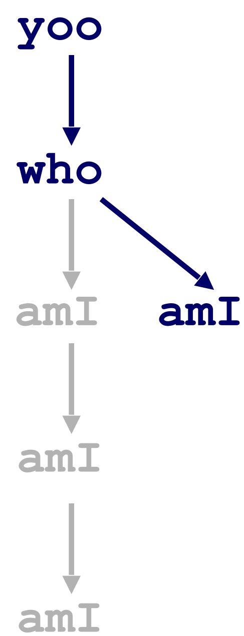


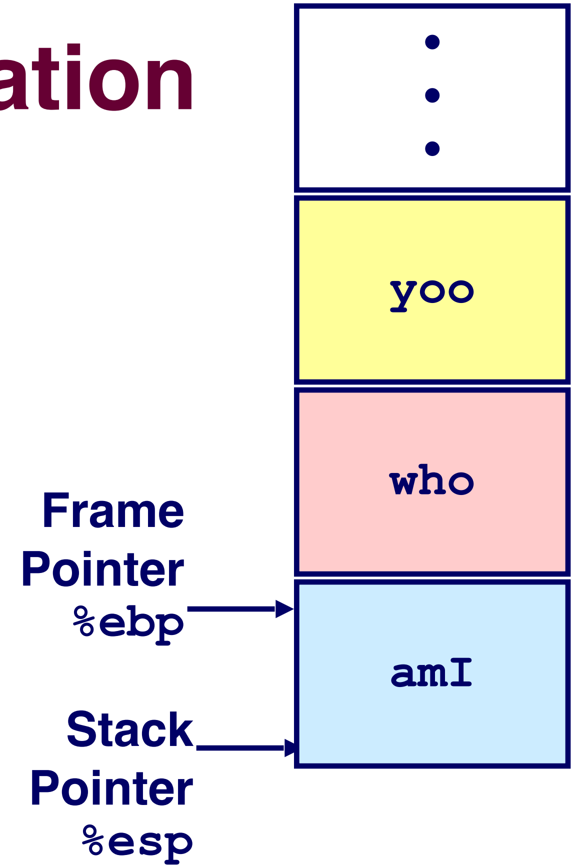


Stack Operation



Call Chain

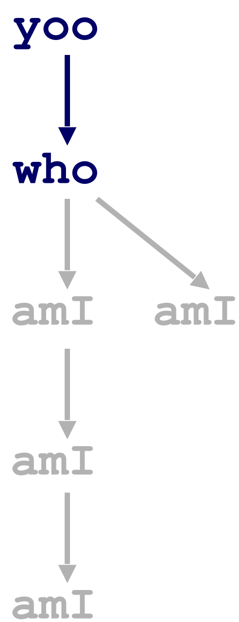


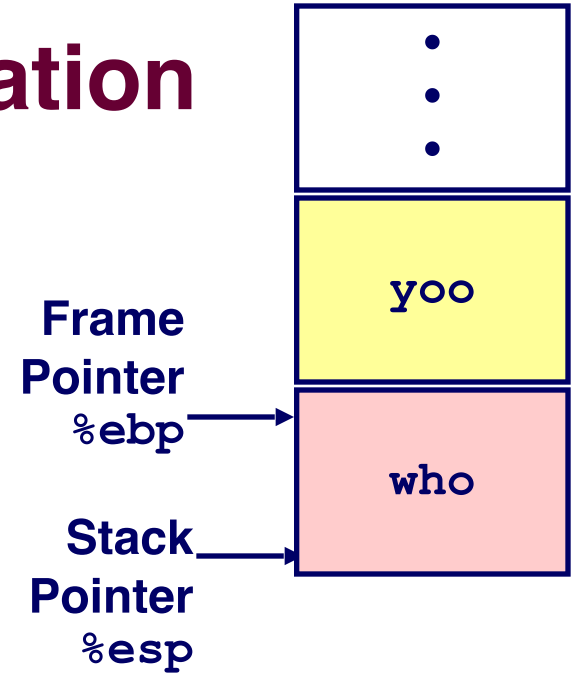


Stack Operation

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

Call Chain

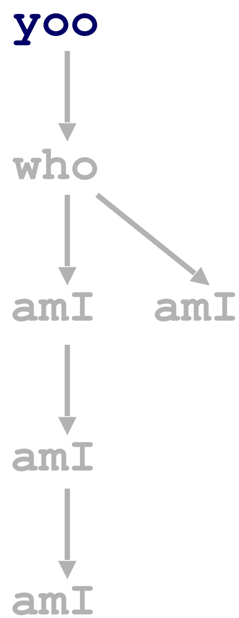


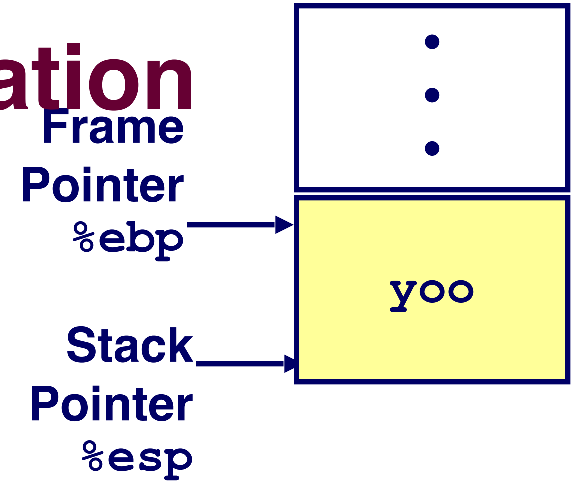


Stack Operation

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

Call Chain





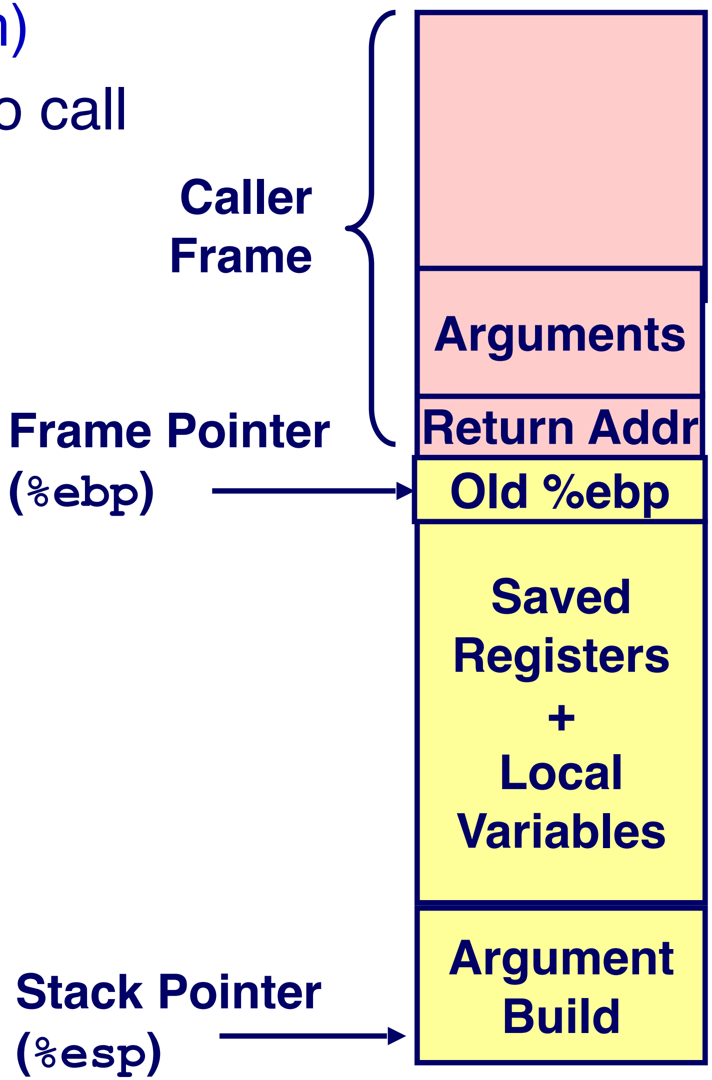
IA32/Linux Stack Frame

Current Stack Frame ("Top" to Bottom)

- Parameters for function about to call
  - "Argument build"
- Local variables
  - If can't keep in registers
- Saved register context
- Old frame pointer

Caller Stack Frame

- Return address
  - Pushed by `call` instruction
- Arguments for this call



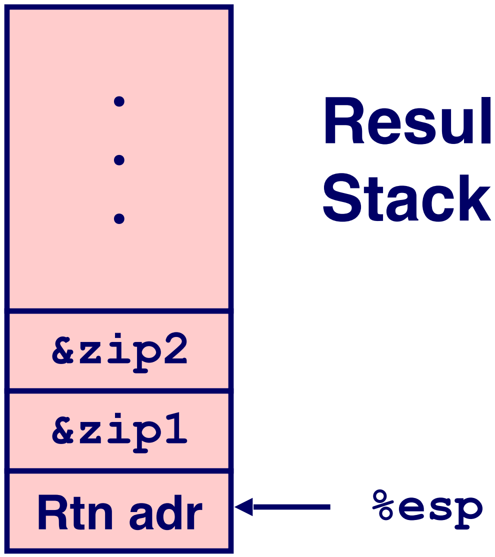
Revisiting swap

```
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
    • • •
```



Resulting Stack

Revisiting `swap`

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

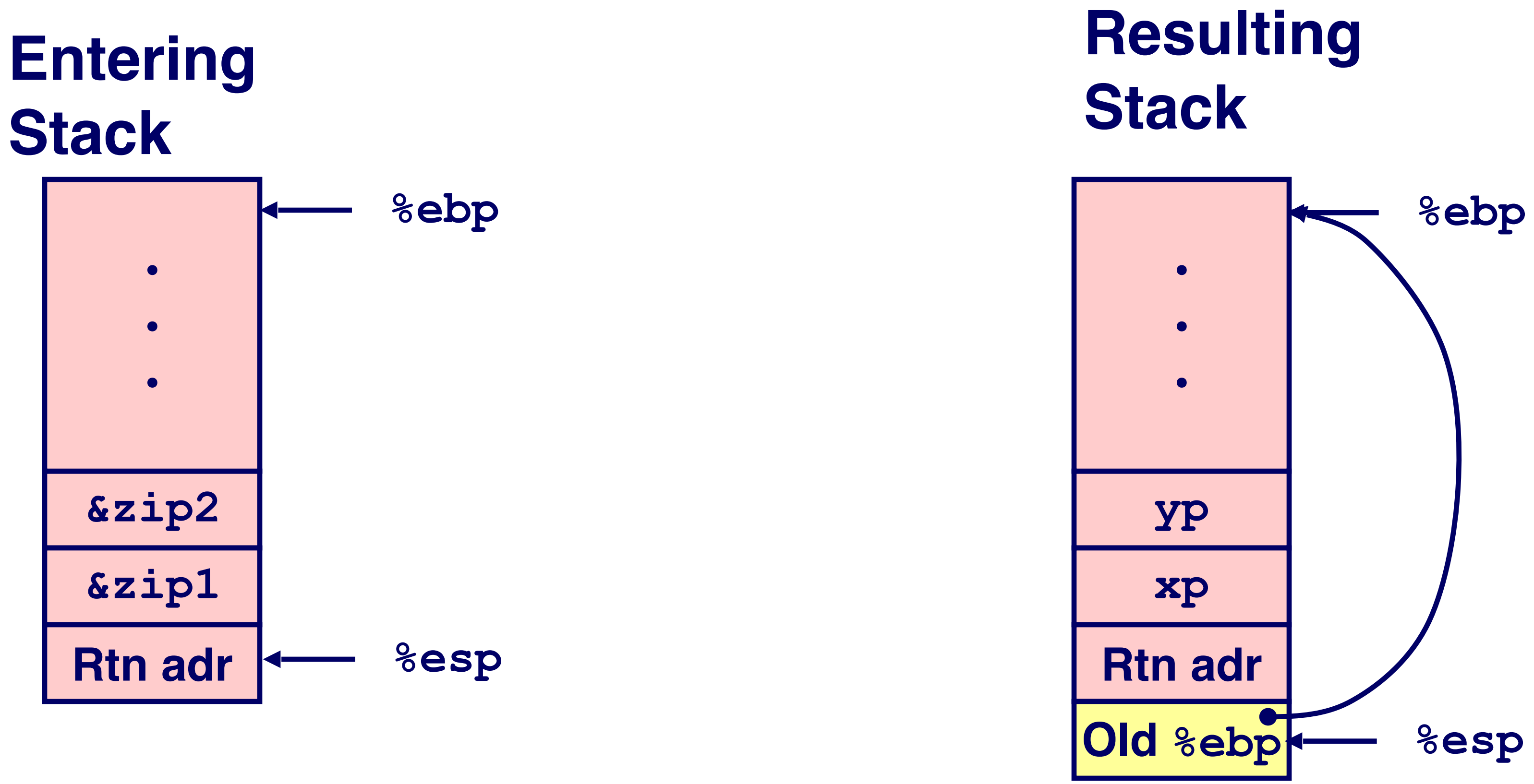
```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    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

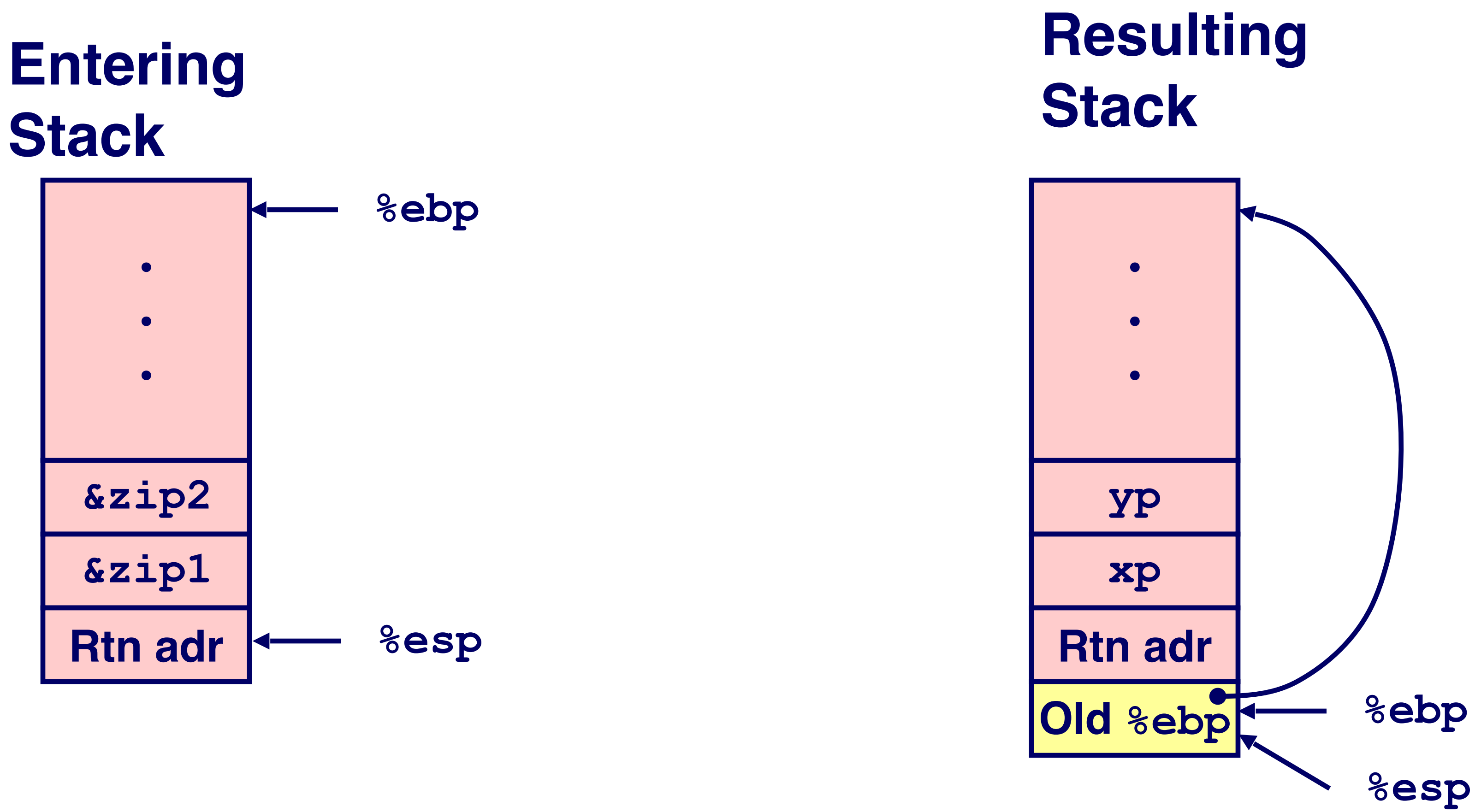
Finish

`swap` Setup #1




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

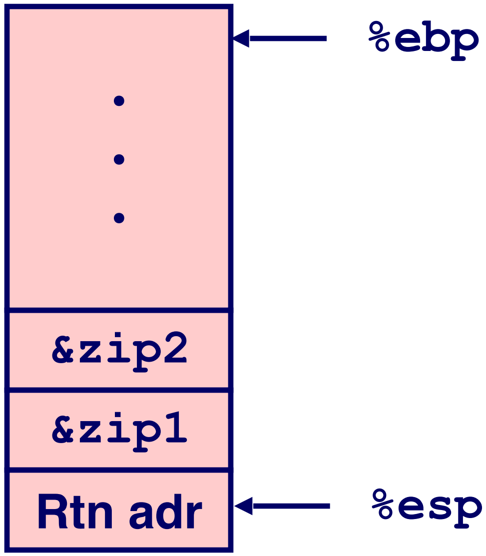
swap Setup #2



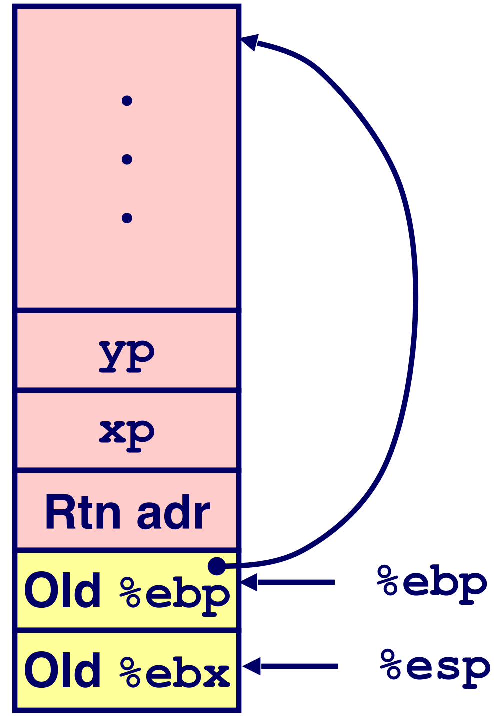

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
```

swap Setup #3

Entering Stack

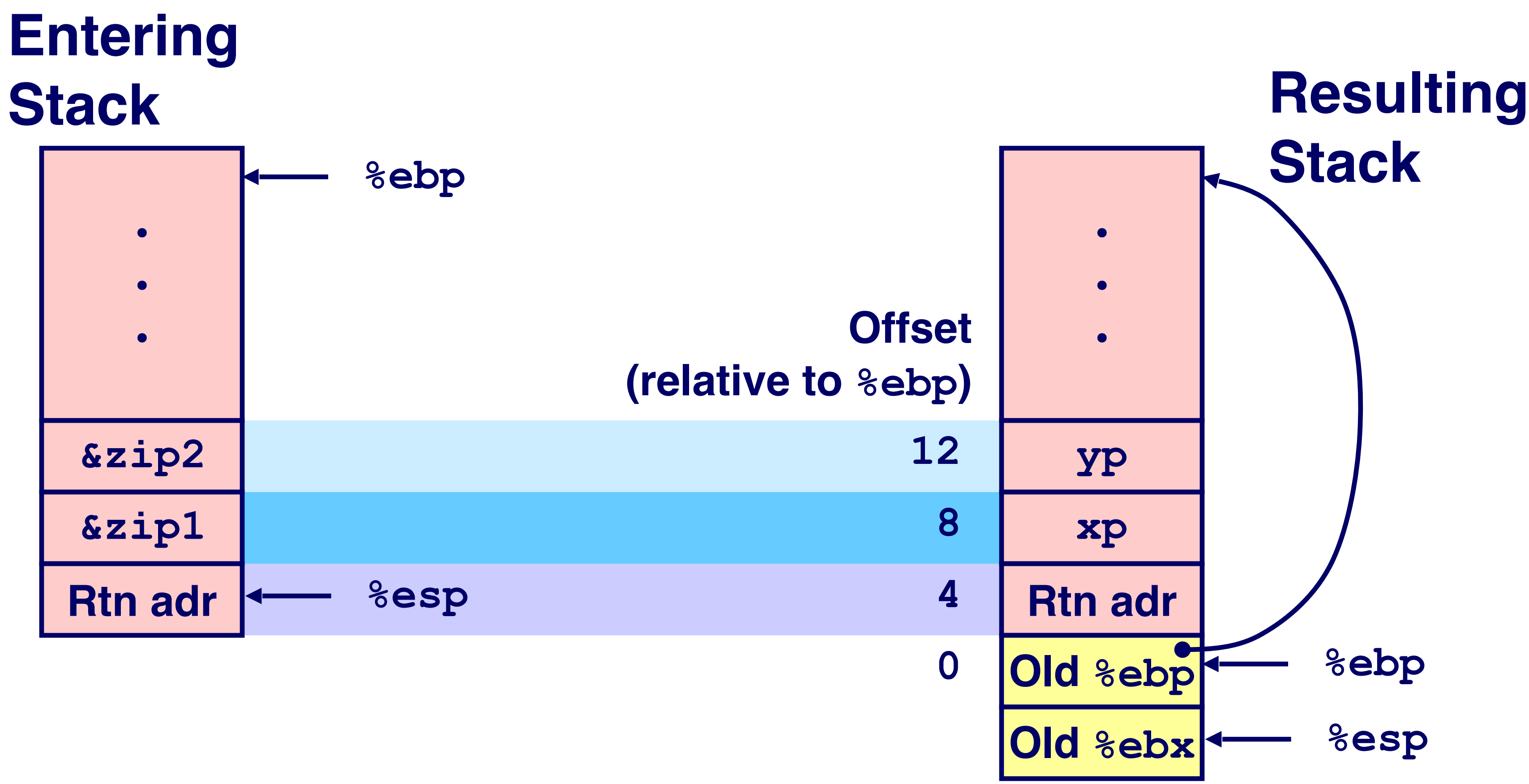


Resulting Stack




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

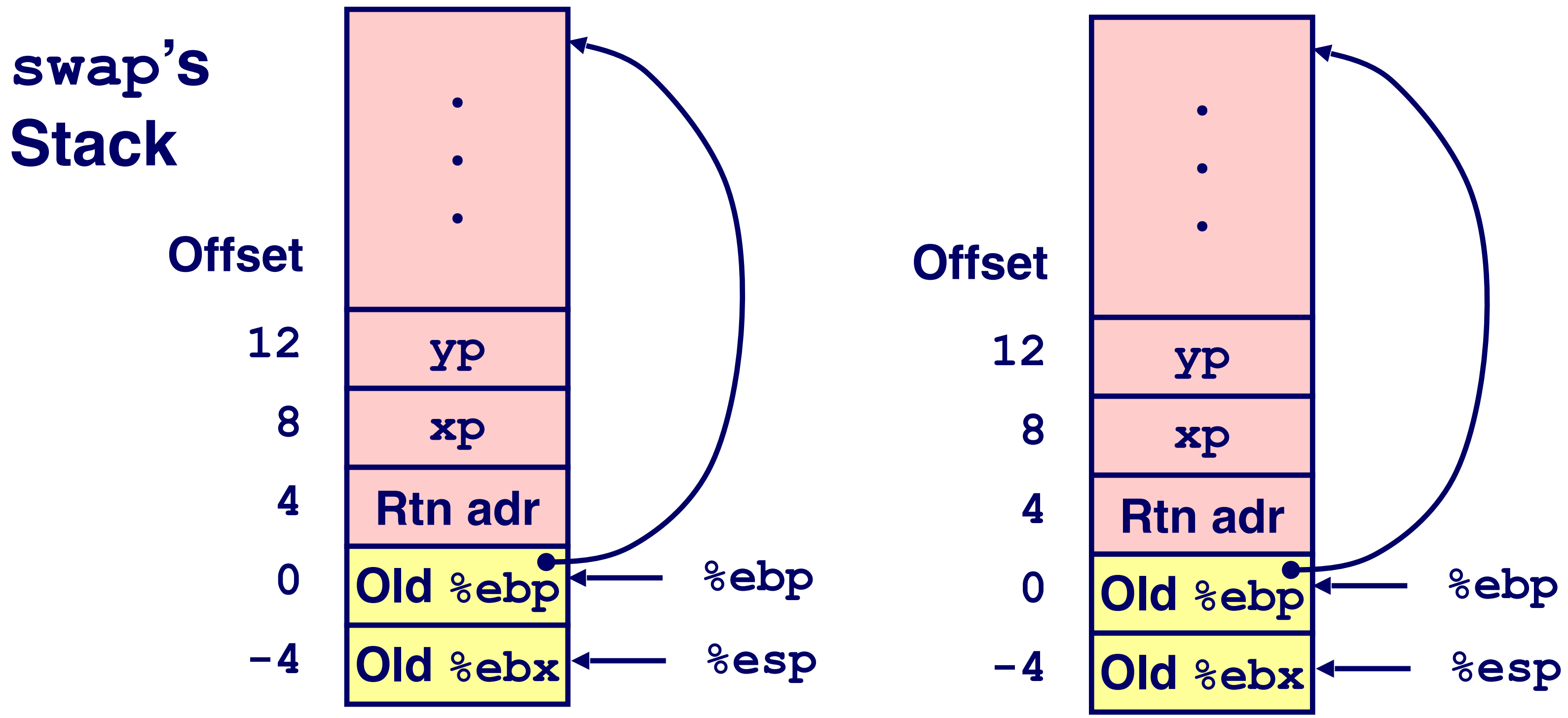
Effect of swap Setup




```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

Body

swap Finish #1

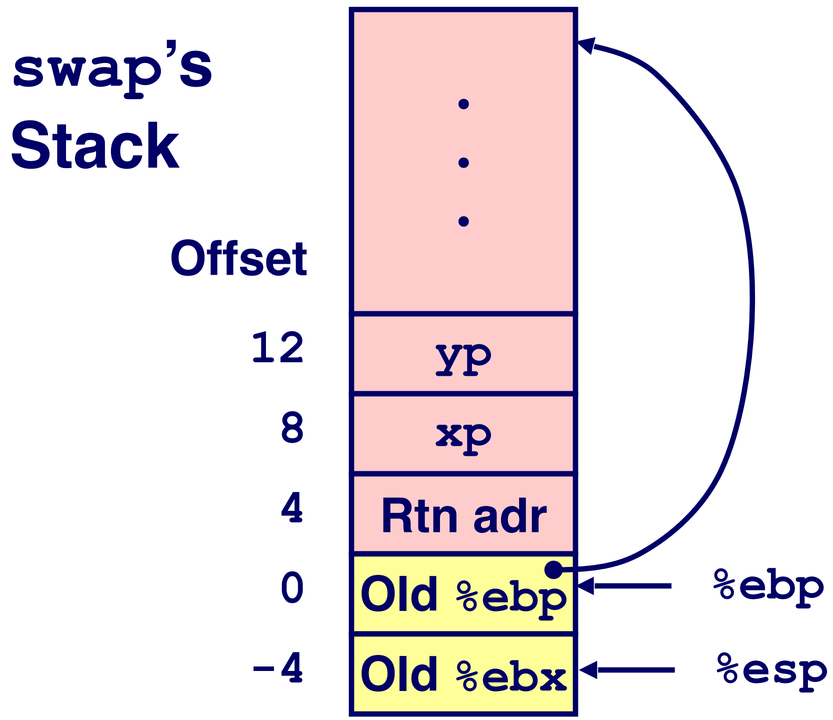


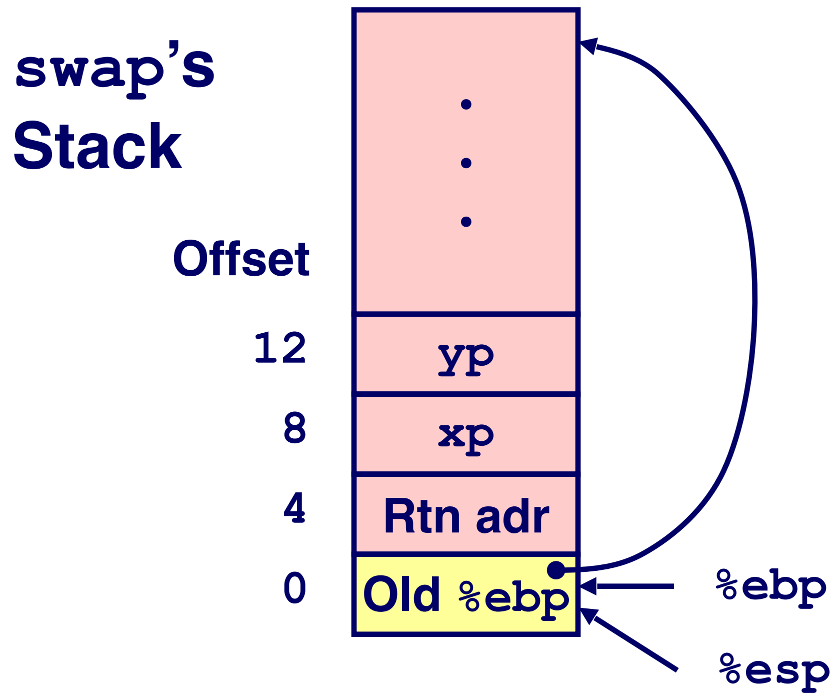

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register `%ebx`

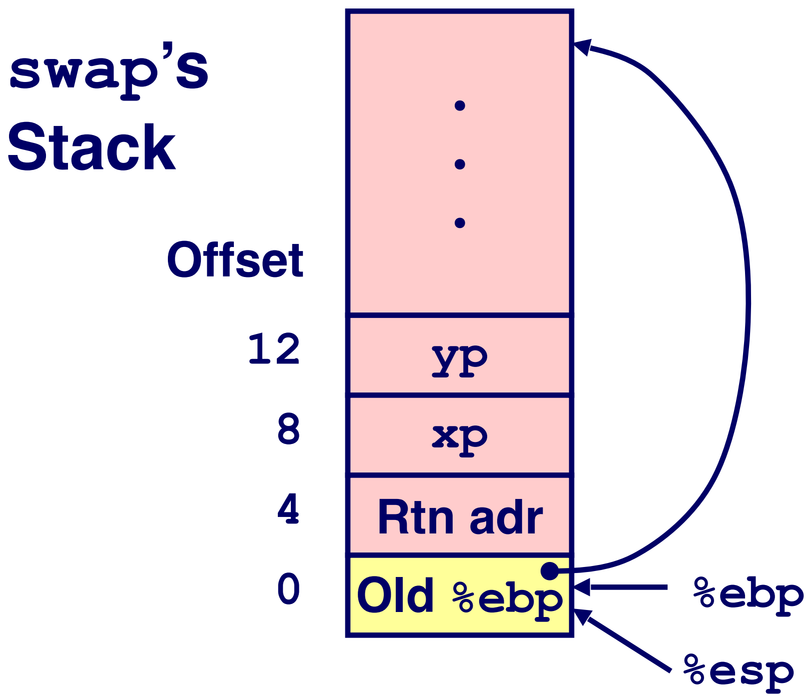
swap Finish #2

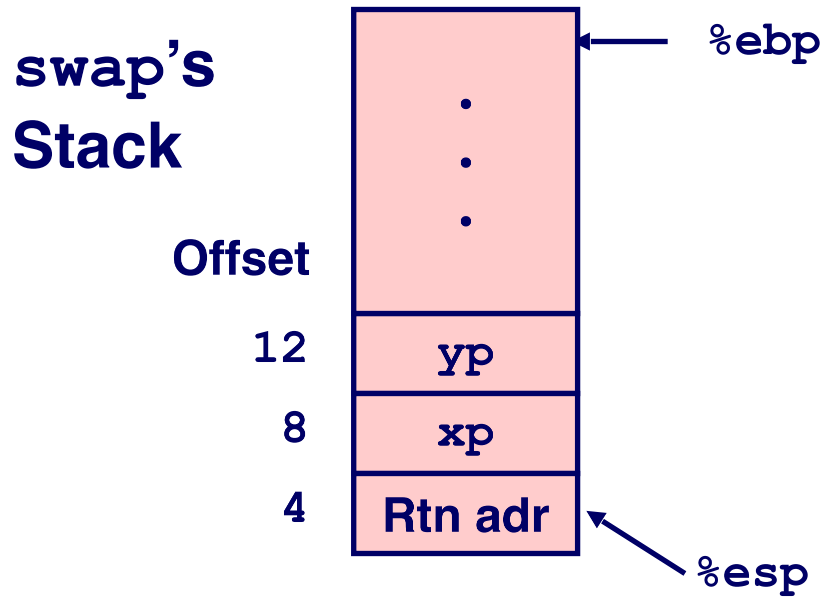





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

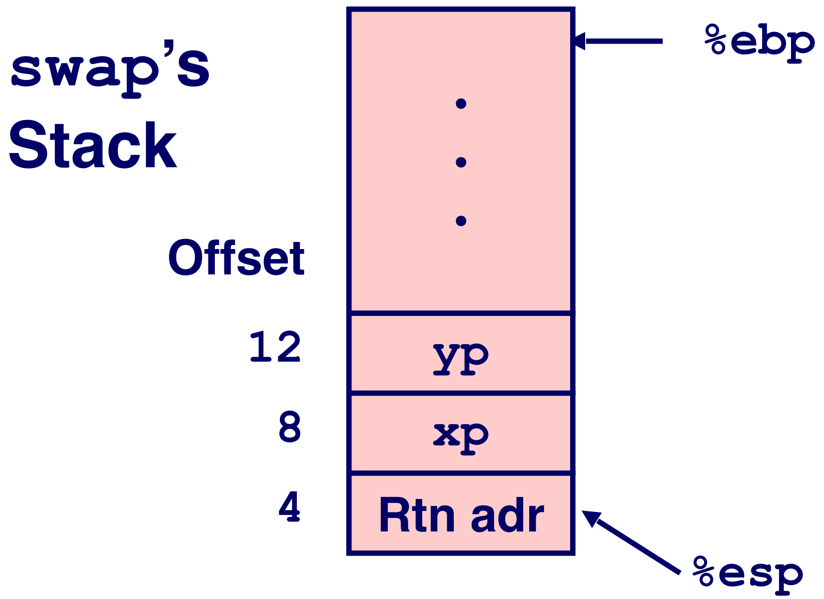
`swap` Finish #3

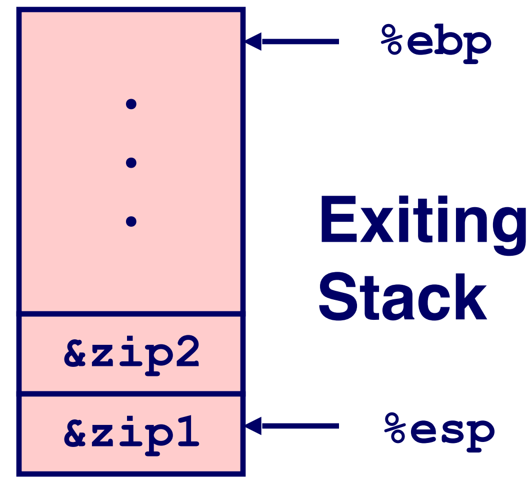





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

`swap` Finish #4






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

Observation

- Saved & restored register `%ebx`
- Didn't do so for `%eax`, `%ecx`, or `%edx`

Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the *caller*, `who` is the *callee*

Can Register be Used for Temporary Storage?

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

- Contents of register `%edx` overwritten by `who`

Register Saving Conventions

When procedure `yoo` calls `who`:

- `yoo` is the *caller*, `who` is the *callee*

Can Register be Used for Temporary Storage?

Conventions

- "Caller Save"
  - Caller saves temporary in its frame before calling
- "Callee Save"
  - Callee saves temporary in its frame before using

IA32/Linux Register Usage

Two have special uses

- `%ebp`, `%esp`

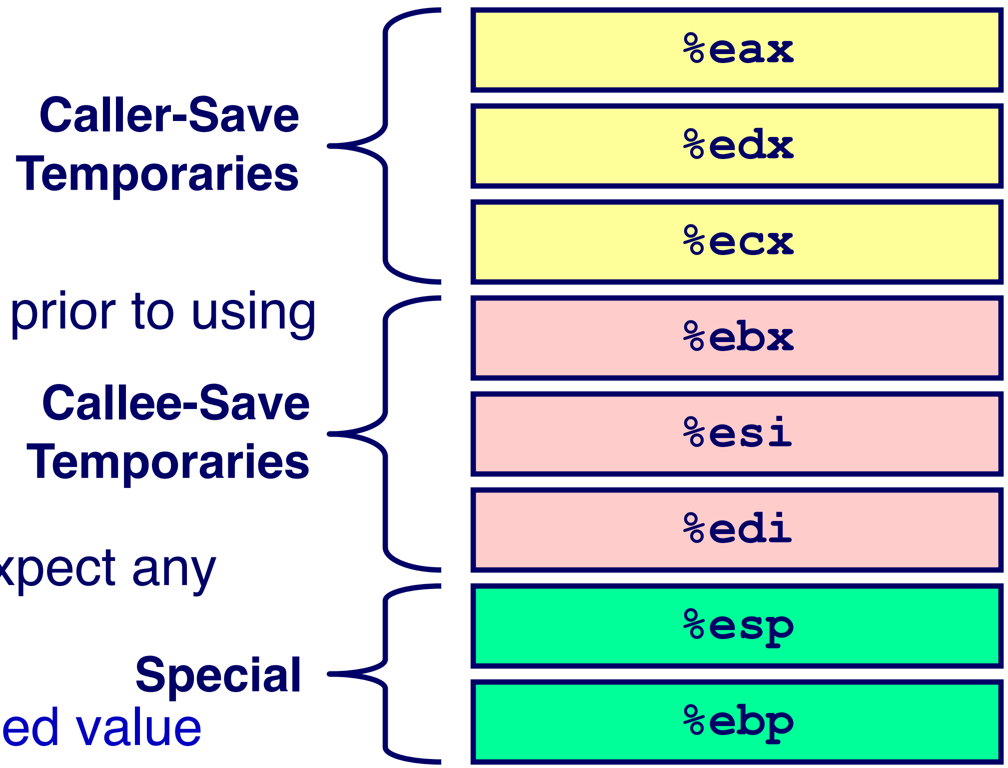
Three managed as callee-save

- `%ebx`, `%esi`, `%edi`
- Old values saved on stack prior to using

Three managed as caller-save

- `%eax`, `%edx`, `%ecx`
- Do what you please, but expect any callee to do so, as well

Register `%eax` also stores returned value



Recursive Function

```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Recursive Factorial

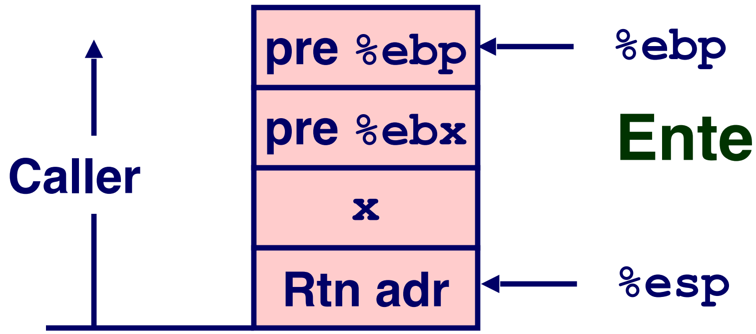
```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Registers

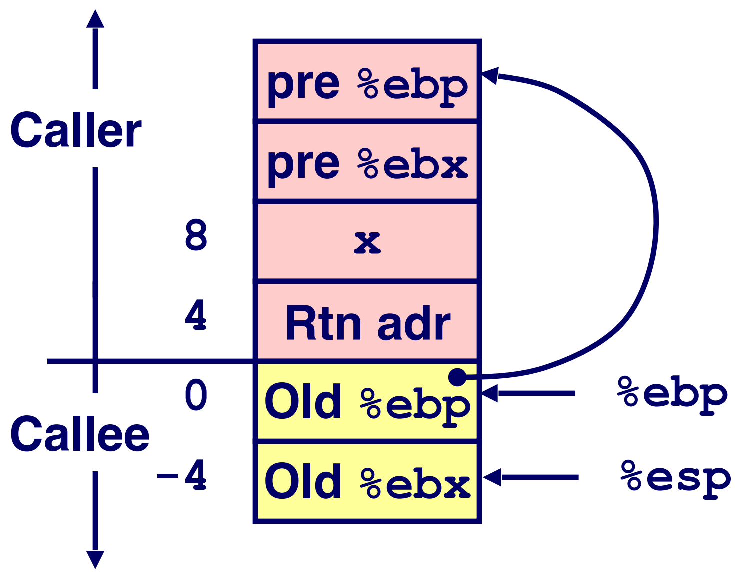
- `%eax` used without first saving
- `%ebx` used, but save at beginning & restore at end

Rfact Stack Setup



Entering Stack

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



Rfact Body

Recursion

```
 movl 8(%ebp),%ebx   # ebx = x
 cmpl $1,%ebx        # Compare x : 1
 jle .L78            # If <= goto Term
 leal -1(%ebx),%eax  # eax = x-1
 pushl %eax          # Push x-1
 call rfact          # rfact(x-1)
 imull %ebx,%eax     # rval * x
 jmp .L79            # Goto done
.L78:               # Term:
 movl $1,%eax        # return val = 1
.L79:               # Done:
```

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1) ;
  return rval * x;
}
```

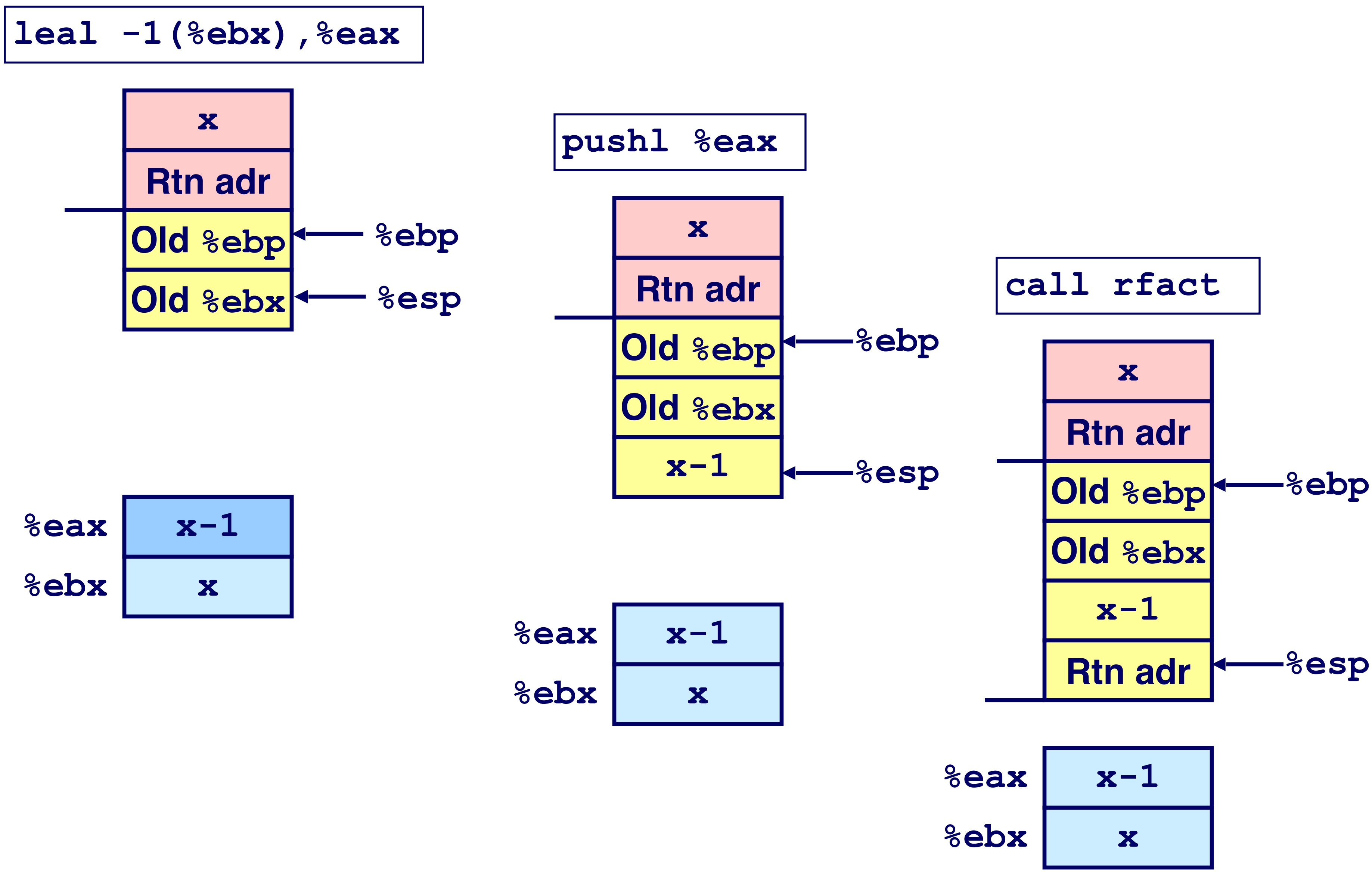
Registers

`%ebx` Stored value of x

`%eax`

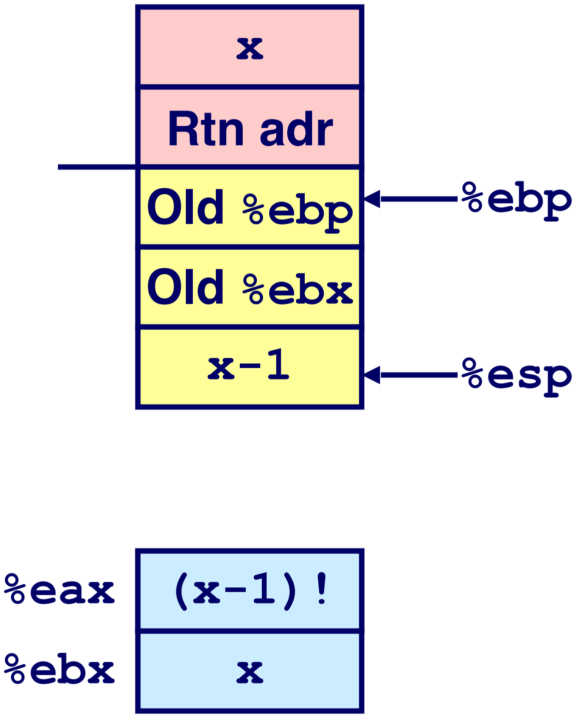
- Temporary value of `x-1`
- Returned value from `rfact(x-1)`
- Returned value from this call

Rfact Recursion



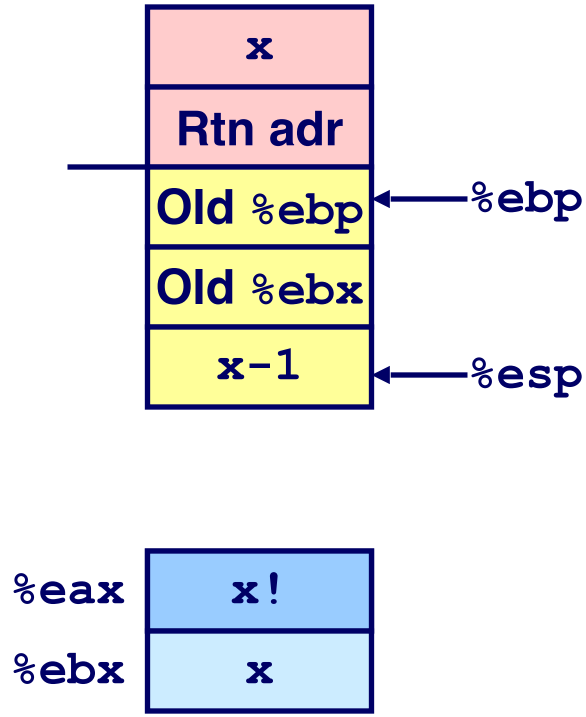
Rfact Result

Return from Call



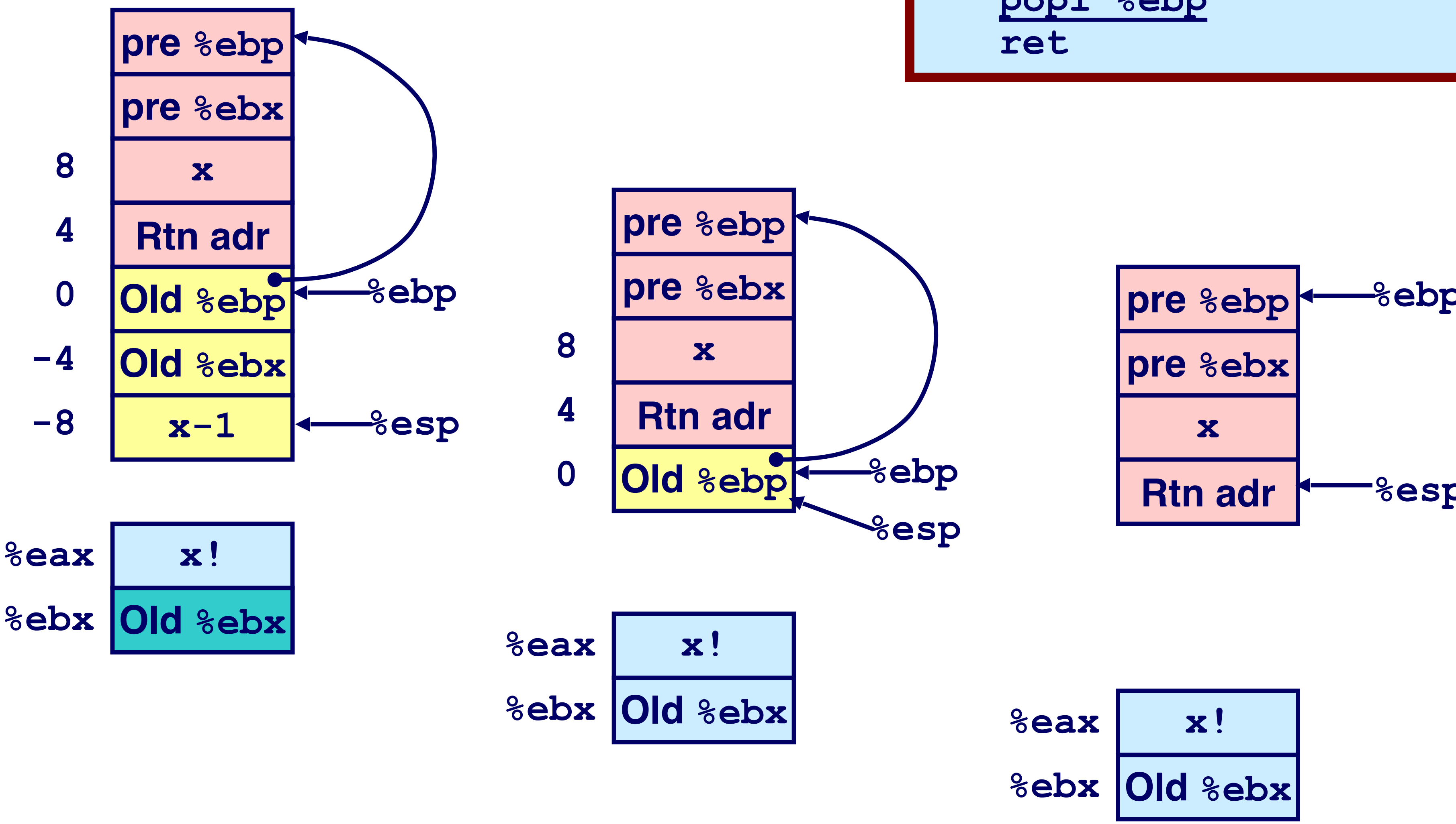
Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

```
imull %ebx,%eax
```



Rfact Completion

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



Basic Data Types

Integral

- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

Intel	GAS	Bytes	C
byte	`b`	1	`[unsigned] char`
word	`w`	2	`[unsigned] short`
double word	`l`	4	`[unsigned] int`

Floating Point

- Stored & operated on in floating point registers

Intel	GAS	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`
Extended	`t`	10/12	`long double`

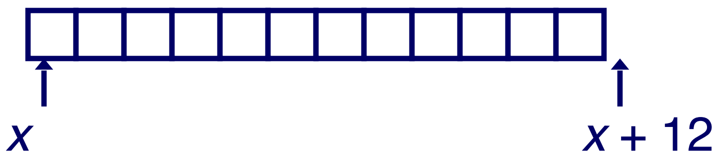
Array Allocation

Basic Principle

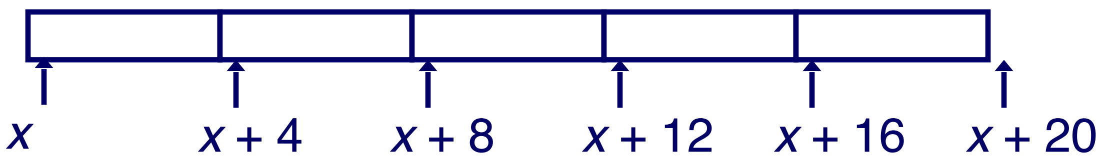
T `A[`*L*`];`

- Array of data type *T* and length *L*
- Contiguously allocated region of *L* * `sizeof(`*T*`)` bytes

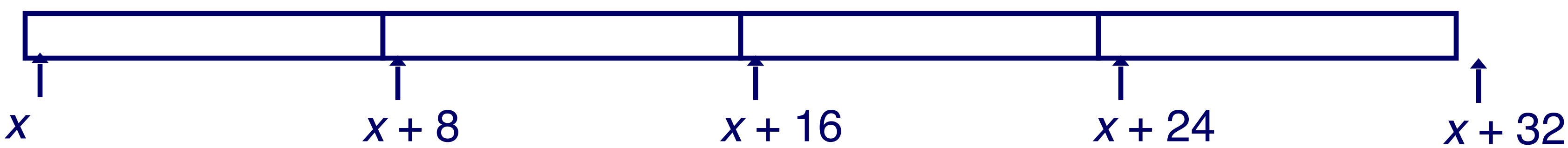
```
char string[12];
```



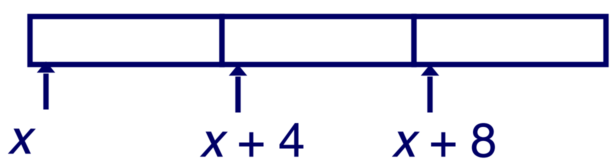

```
int val[5];
```




```
double a[4];
```

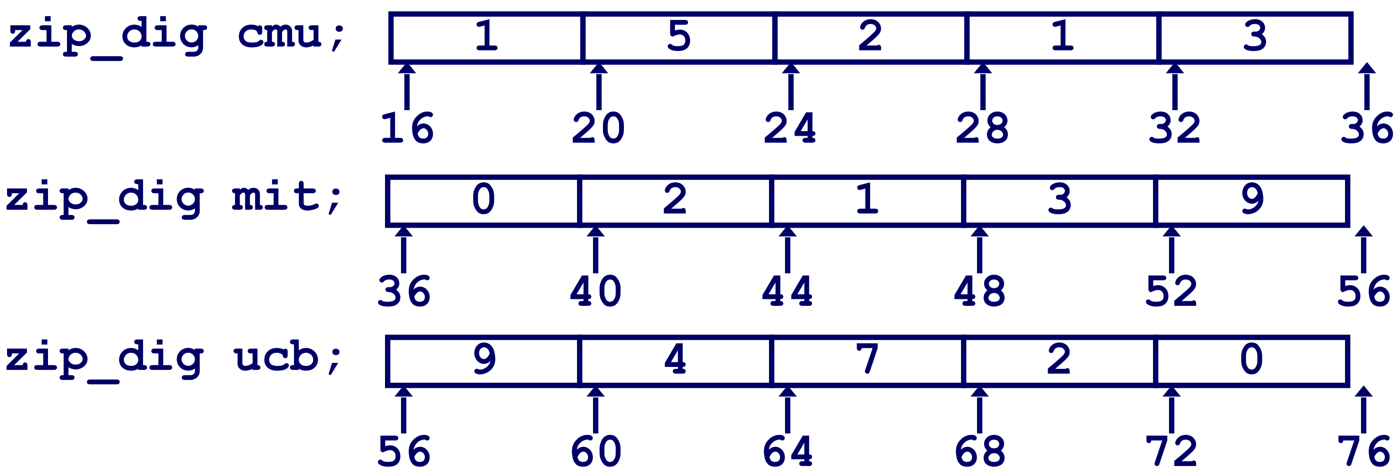



```
char *p[3];
```



Array Example

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



Notes

- Declaration “`zip_dig cmu`” equivalent to “`int cmu[5]`”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

Computation

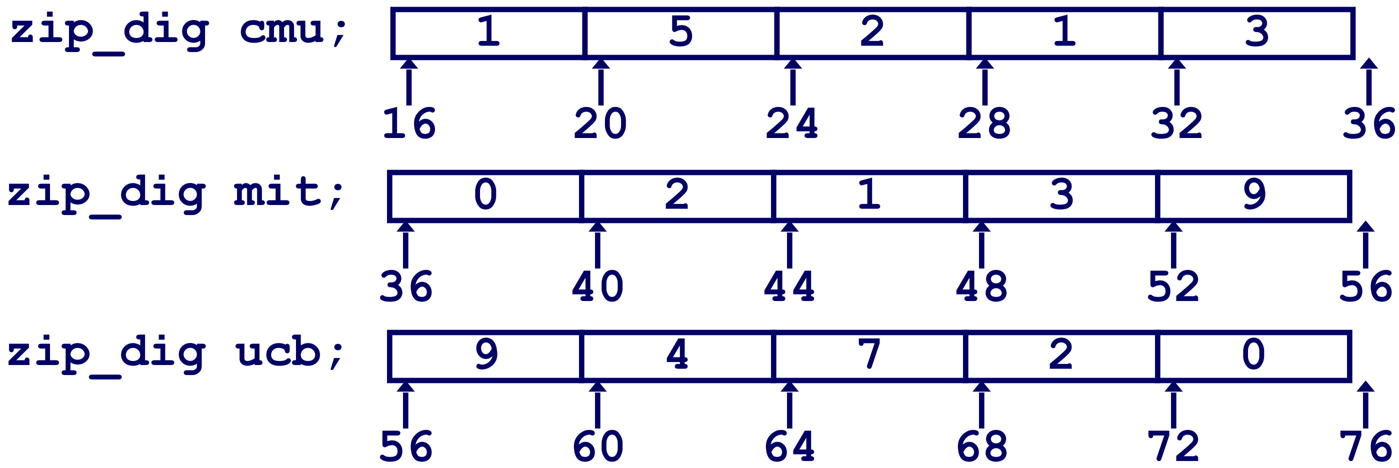
- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

```
int get_digit
   (zip_dig z, int dig)
{
  return z[dig];
}
```

Memory Reference Code

```
  # %edx = z
  # %eax = dig
   movl (%edx,%eax,4),%eax  # z[dig]
```

Referencing Examples



Code Does Not Do Any Bounds Checking!

Reference	Address	Value	Guaranteed?
mit[3]	36 + 4* 3 = 48	3	**Yes**
mit[5]	36 + 4* 5 = 56	9	**No**
mit[-1]	36 + 4*-1 = 32	3	**No**
cmu[15]	16 + 4*15 = 76	??	**No**

- Out of range behavior implementation-dependent
  - No guaranteed relative allocation of different arrays

Array Loop Example

Original Source

```
int zd2int(zip_dig z)
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

Transformed Version

- As generated by GCC
- Eliminate loop variable `i`
- Convert array code to pointer code
- Express in do-while form
  - No need to test at entrance

```
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

Array Loop Implementation

Registers

```
%ecx   z
%eax   zi
%ebx   zend
```

Computations

- `10*zi + *z` implemented as `*z + 2*(zi+4*zi)`
- `z++` increments by 4

```c
int zd2int(zip_dig z)
{
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

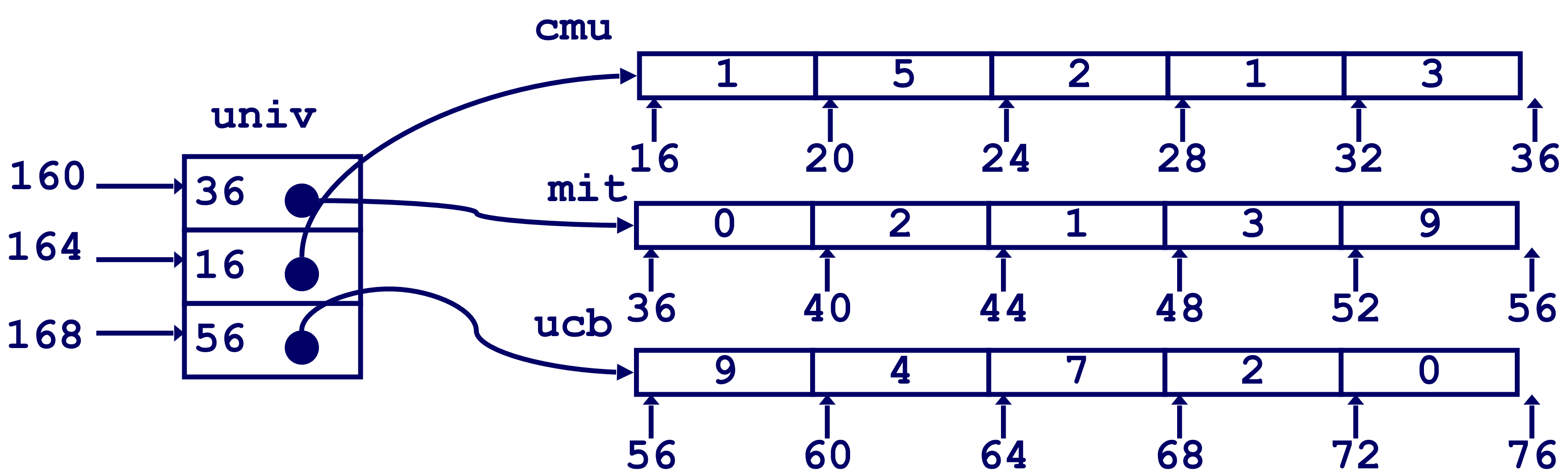
```
  # %ecx = z
  xorl %eax,%eax                 # zi = 0
  leal 16(%ecx),%ebx             # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx        # 5*zi
  movl (%ecx),%eax               # *z
  addl $4,%ecx                   # z++
  leal (%eax,%edx,2),%eax        # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx                 # z : zend
  jle .L59                       # if <= goto loop
```

Multi-Level Array Example

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of `int`'s

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```



Element Access in Multi-Level Array

```
int get_univ_digit
  (int index, int dig)
{
  return univ[index][dig];
}
```

Computation

- Element access
  `Mem[Mem[univ+4*index]+4*dig]`
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

```
  # %ecx = index
  # %eax = dig
  leal 0(,%ecx,4),%edx     # 4*index
  movl univ(%edx),%edx     # Mem[univ+4*index]
  movl (%edx,%eax,4),%eax  # Mem[...+4*dig]
```

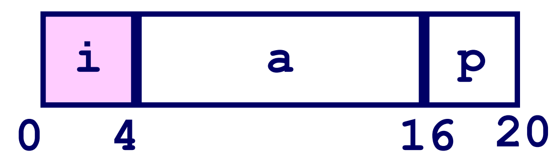
Structures

Concept

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout



Accessing Structure Member

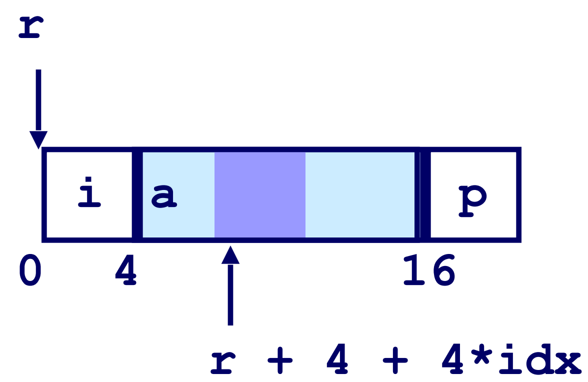
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Generating Pointer to Struct. Member

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



Generating Pointer to Array Element

- Offset of each structure member determined at compile time

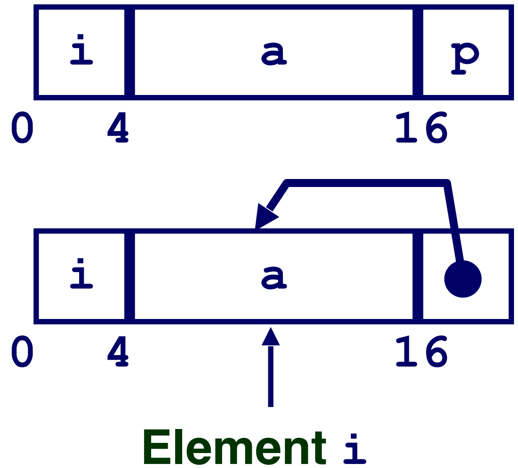
```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

Structure Referencing (Cont.)

C Code

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```




```
void
set_p(struct rec *r)
{
  r->p =
   &r->a[r->i];
}
```

```
 # %edx = r
movl (%edx),%ecx           # r->i
leal 0(,%ecx,4),%eax       # 4*(r->i)
leal 4(%edx,%eax),%eax     # r+4+4*(r->i)
movl %eax,16(%edx)         # Update r->p
```

Alignment

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on IA32
  - treated differently by Linux and Windows!

Motivation for Aligning Data

- Memory accessed by (aligned) double or quad-words
  - Inefficient to load or store datum that spans quad word boundaries

Compiler

- Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment

Size of Primitive Data Type:

- 1 byte (e.g., `char`)
  - no restrictions on address
- 2 bytes (e.g., `short`)
  - lowest 1 bit of address must be 0_2
- 4 bytes (e.g., `int`, `float`, `char *`, etc.)
  - lowest 2 bits of address must be 00_2
- 8 bytes (e.g., `double`)
  - Windows (and most other OS's & instruction sets):
    - » lowest 3 bits of address must be 000_2
  - Linux:
    - » lowest 2 bits of address must be 00_2
    - » i.e., treated the same as a 4-byte primitive data type
- 12 bytes (`long double`)
  - Linux:
    - » lowest 2 bits of address must be 00_2
    - » i.e., treated the same as a 4-byte primitive data type

Satisfying Alignment with Structures

Offsets Within Structure

- Must satisfy element's alignment requirement

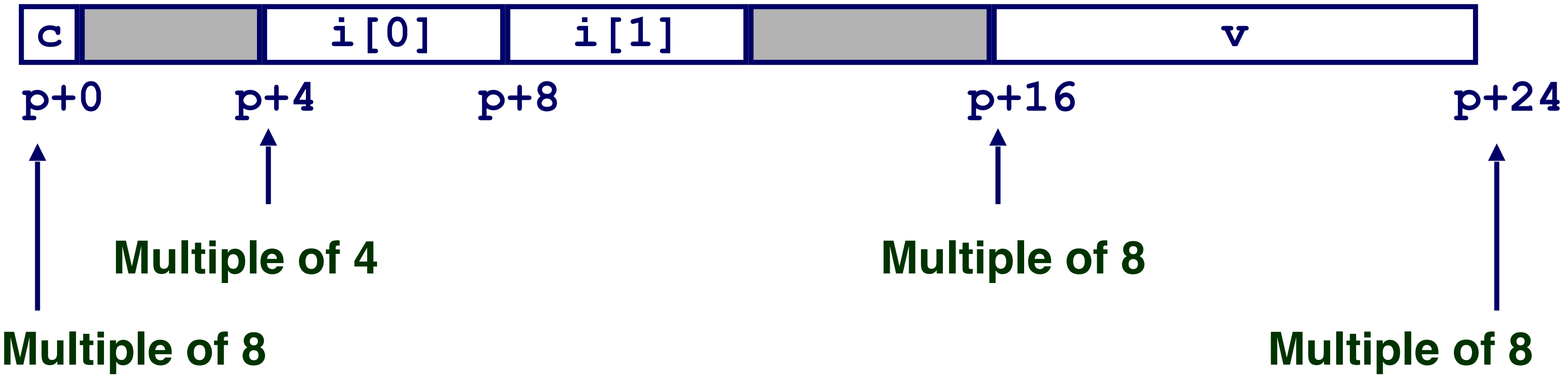
Overall Structure Placement

- Each structure has alignment requirement K
  - Largest alignment of any element
- Initial address & structure length must be multiples of K

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Example (under Windows):

- K = 8, due to `double` element

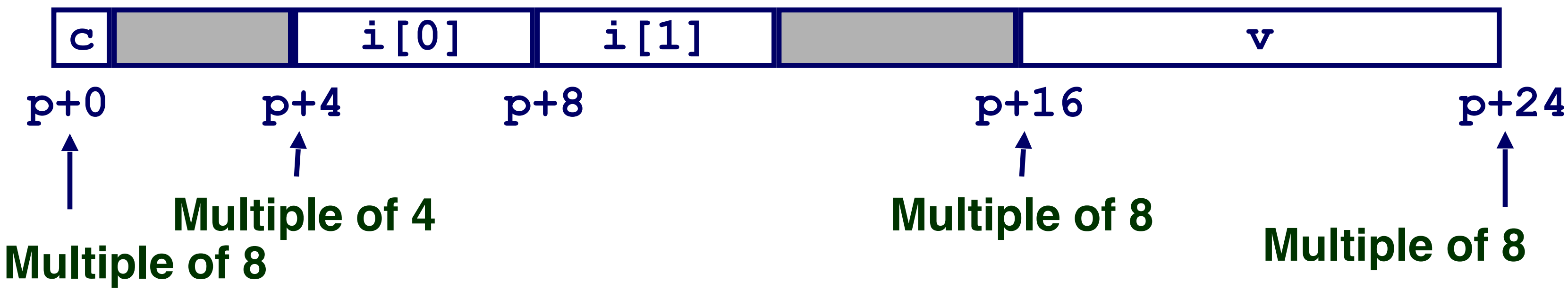


Linux vs. Windows

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

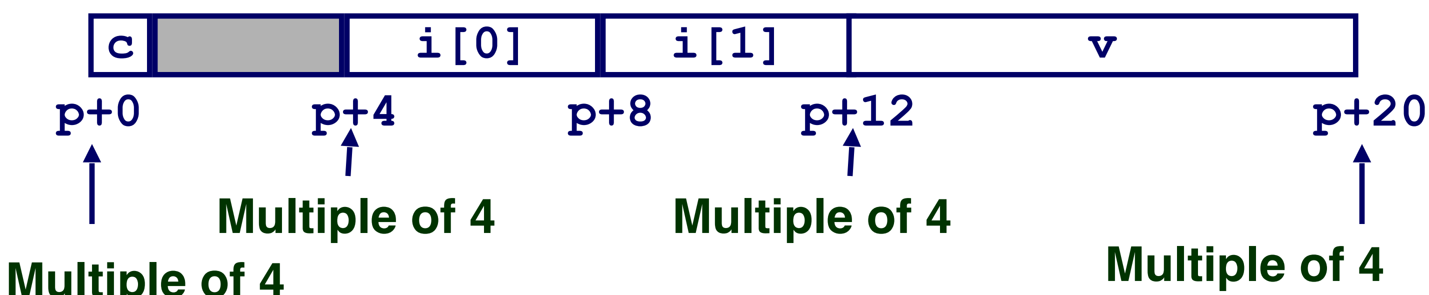
Windows (including Cygwin):

- K = 8, due to `double` element



Linux:

- K = 4; `double` treated like a 4-byte data type

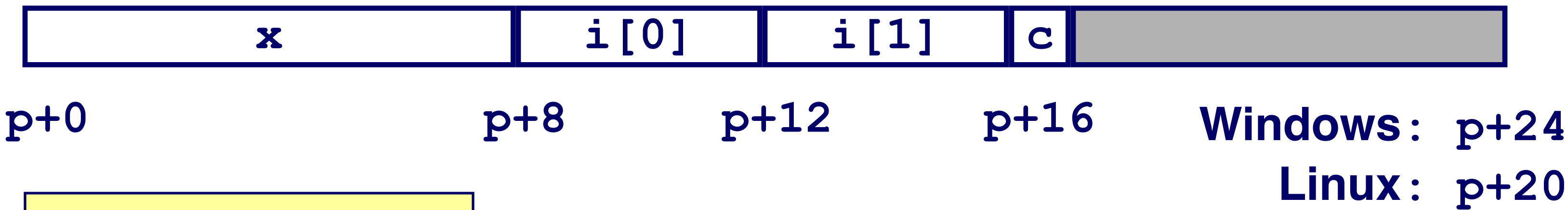


Overall Alignment Requirement

```
struct S2 {
  double x;
  int i[2];
  char c;
} *p;
```

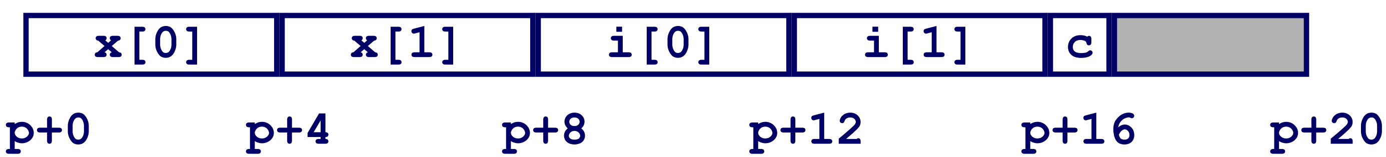
`p` must be multiple of:

- 8 for Windows
- 4 for Linux




```
struct S3 {
  float x[2];
  int i[2];
  char c;
} *p;
```

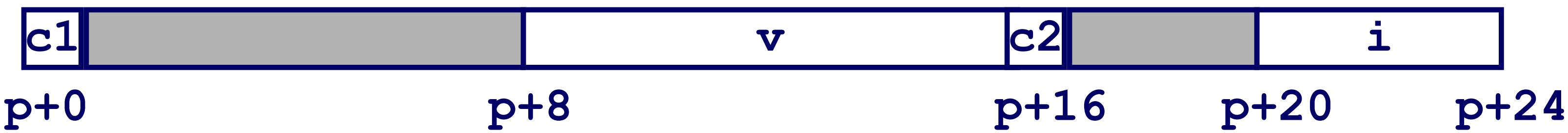
`p` must be multiple of 4 (in either OS)



Ordering Elements Within Structure

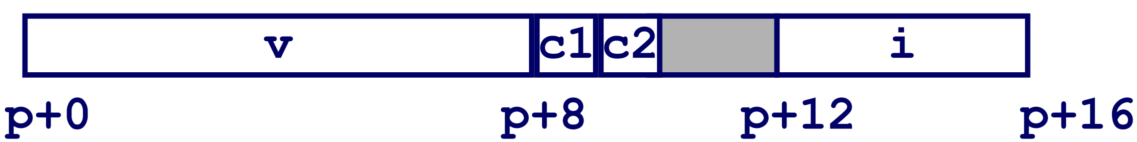
```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

10 bytes wasted space in Windows




```
struct S5 {
  double v;
  char c1;
  char c2;
  int i;
} *p;
```

2 bytes wasted space

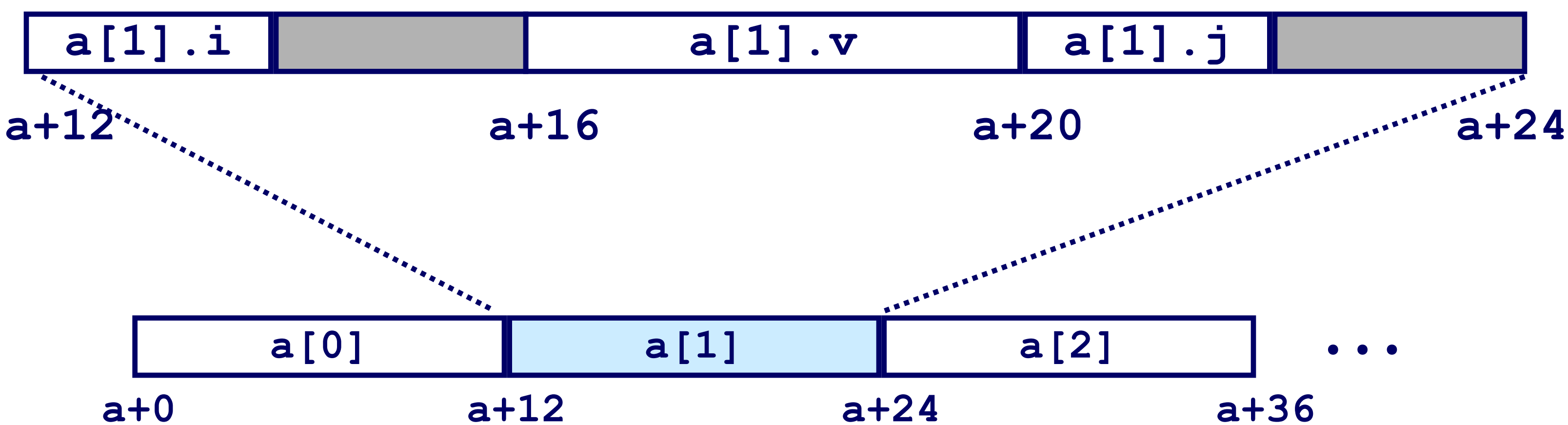


Arrays of Structures

Principle

- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```

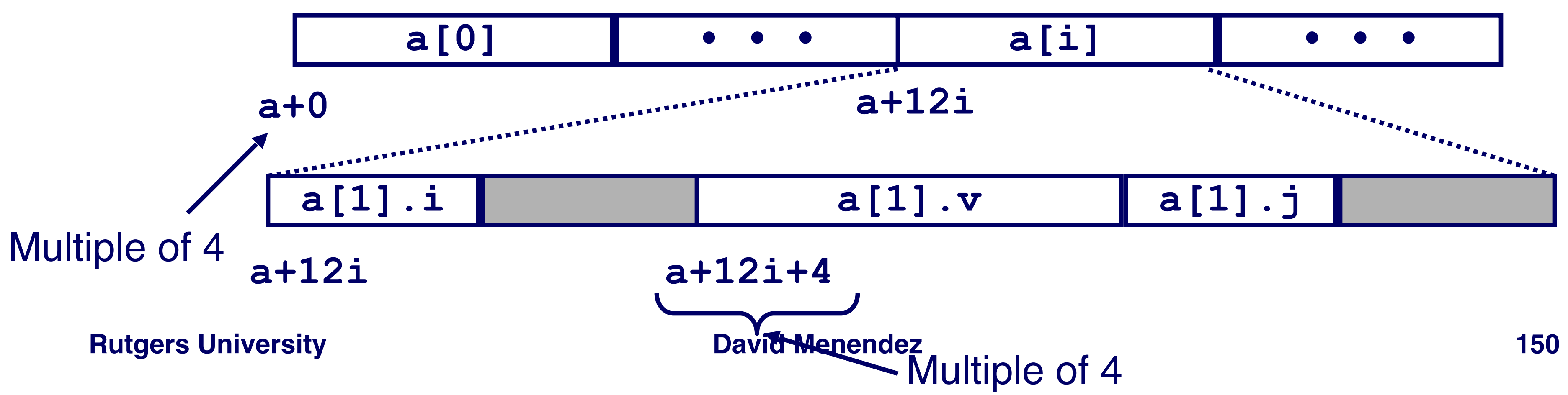


Satisfying Alignment within Structure

Achieving Alignment

- Starting address of structure array must be multiple of worst-case alignment for any element
  - `a` must be multiple of 4
- Offset of element within structure must be multiple of element's alignment requirement
  - `v`'s offset of 4 is a multiple of 4
- Overall size of structure must be multiple of worst-case alignment for any element
  - Structure padded with unused space to be 12 bytes

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```



Summary

Arrays in C

- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Compiler Optimizations

- Compiler often turns array code into pointer code (`zd2int`)
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops

Structures

- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment