Chapter 3
Machine-Level Programming II
(Sections 3.4 – 3.9)

with material from Dr. Bin Ren, College of William & Mary

Outline

- **Introduction of IA32**
- **IA32 operations**
  - Data movement operations
  - Stack operations and function calls
  - Arithmetic and logic operations
  - Compare and jump operations
- **Instruction encoding format**
- **Array and structures allocation and access**

RISC instruction sets

- Reduced Instruction Set Computer
- Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)

Fewer, simpler instructions

- Might take more to get given task done
- Can execute them with small and fast hardware

Register-oriented instruction set

- Many more (typically 32) registers
- Use for arguments, return pointer, temporaries

Only load and store instructions can access memory

- Similar to Y86 `mrmovl` and `rmmovl`

No Condition codes

- Test instructions return 0/1 in register

CISC instruction sets

- Complex Instruction Set Computer
- Dominant style through mid-80's

Stack-oriented instruction set

- Use stack to pass arguments, save program counter
- Explicit push and pop instructions

Arithmetic instructions can access memory

- `addl %eax, 12(%ebx,%ecx,4)`
  - requires memory read and write
  - Complex address calculation

Condition codes

- Set as side effect of arithmetic and logical instructions

Philosophy

- Add instructions to perform "typical" programming tasks

RISC and CISC

- **Which is IA32?**
  - CISC
- **Which is Y86?**
  - Includes attributes of both.
  - CISC
    - Condition codes
    - Variable length instructions
    - Stack intensive procedure linkages
  - RISC
    - Load-store architecture
    - Regular encoding
- **Which is better: RISC or CISC?**

Compare Y86 and IA32

- **Y86 is:**
  - Little endian
  - Load/store
    - Can only access memory on read/write
    - On move statements in Y86 (mrmovl/rmmovl)
  - Combination of CISC and RISC
  - Word = 4 bytes
- **IA32 is:**
  - Little endian
  - NOT load/store
  - CISC
  - Byte (1 byte), word (2 bytes), long (4 bytes)

C program to IA32 and Y86

- **Computes the sum of an integer array**

```
int Sum (int *Start, int Count)
{
    int sum = 0;
    while (Count)
    {
            sum += *Start;
            Start++;
            Count--;
    }
}
```

ASSEMBLY COMPARISON ON NEXT SLIDE

Why not using array indexing (i.e. subscripting)?
No scaled addressing modes in Y86

Uses stack and frame pointers

For simplicity, does not follow IA32 convention of having some registers designated as callee-save registers (convention so adopt or ignore as we please)

IA32/Y86 comparison

IA32 code

```
  int Sum(int *Start, int Count)
Sum:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%ecx      ecx = Start
  movl 12(%ebp),%edx     edx = Count
  xorl %eax,%eax         sum = 0
  testl %edx,%edx
  je .L34
.L35:
  addl (%ecx),%eax      add *Start to sum
  addl $4,%ecx          Start++
  decl %edx             Count--
  jnz .L35              Stop when 0
.L34:
  movl %ebp,%esp
  popl %ebp
  ret
```

Y86 code

```
  int Sum(int *Start, int Count)
Sum:
  pushl %ebp
  rrmovl %esp,%ebp
  mrmovl 8(%ebp),%ecx       ecx = Start
  mrmovl 12(%ebp),%edx      edx = Count
  xorl %eax,%eax            sum = 0
  andl   %edx,%edx          Set condition codes
  je     End
Loop:
  mrmovl (%ecx),%esi        get *Start
  addl %esi,%eax            add to sum
  irmovl $4,%ebx
  addl %ebx,%ecx            Start++
  irmovl $-1,%ebx
  addl %ebx,%edx            Count--
  jne    Loop               Stop when 0
End:
  rrmovl %ebp,%esp
  popl %ebp
  ret
```

Figure 4.6: **Comparison of Y86 and IA32 assembly programs.** The `Sum` function computes the sum of an integer array. The Y86 code differs from the IA32 mainly in that it may require multiple instructions to perform what can be done with a single IA32 instruction.

CHAPTER 3.2 Program Encodings

- **GOAL → examine assembly code and map it back to the constructs found in high-level programming languages**
- **%gcc –O1 –m32 –S code.c → code.s**
- **%more code.s**
  - Runs the compiler only
  - -S options = generates an assembly (.s) file
  - -O1 is an optimization level
  - All information about local variables names or data types have been stripped away
  - Still see global variable “accum”
    - Compiler has not yet determined where in memory this variable will be stored
- **%gcc –O1 –c –m32 code.c → code.o**
- **%objdump –d code.o**
  - -c compiles and assembles the code
  - Generates an object-code file (.o) = binary format
  - DISASSEMBLER – re-engineers the object code back into assembly language
  - %uname –p
  - -m32 is a gcc option to run/build 32-bit applications on a 64-bit machine

Machine code vs C code

- **Program Counter (PC)**
  - Register %eip (X86-64)
  - Address in memory of the next instruction to be executed
- **Integer Register File**
  - Contains eight named locations for storing 32-bit values
    - Can hold addresses (C pointers) or integer data
    - Have other special duties
- **Condition Code registers**
  - Hold status information
    - About arithmetic or logical instruction executed
      - CF (carry flag)
      - OF (overflow flag)
      - SF (sign flag)
      - ZF (zero flag)
- **Floating point registers**

Machine Instruction Example

- **C code**
  - Add two signed integers
- **Assembly**
  - Add 2 4-byte integers
- **Operands**
  - X: register %eax
  - Y: memory M[%ebp+8]
  - T: register %eax
  - Return function value in %eax
- **Object code**
  - 3 byte instruction
  - Stored at address: 0x????????

```
int  t = x + y;
```

```
addl 8(%ebp),%eax
```

```
03 45 08
```

IA32 – Intel Architecture

- **32-bit address bus**
  - normal physical address space of 4 GBytes (2^{32} bytes)
  - addresses ranging continuously from 0 to 0xFFFFFFFF
- **Complex instruction set (CISC) machine**
- **Data formats →**
  - Primitive data types of C
  - Single byte suffix
    - denotes size of operand
  - No aggregate types
    - Arrays, structures

C Declaration	Suffix	Name	Size
char	B	BYTE	8 bits
short	W	WORD	16 bits
int	L	LONG	32 bits
char * (pointer)	L	LONG	32 bits
float	S	SINGLE	32 bits

- **Registers**
  - six (almost) general purpose 32-bit registers:
    - %eax, %ebx, %ecx, %edx, %esi, %edi
  - two specialty → stack pointer and base/frame pointer:
    - %esp, %ebp
  - Float values are in different registers (later)
    - a floating-point processing unit (FPU) with eight 80-bit wide registers: st(0) to st(7)

Outline

- **Introduction of IA32**
- **IA32 operations**
  - Data movement operations
  - Stack operations and function calls
  - Arithmetic and logic operations
  - Compare and jump operations
- **Instruction encoding format**
- **Array and structures allocation and access**

Operand Specifiers

- **Source operand**
  - Constants, registers, or memory
- **Destination operand**
  - Registers or memory
- **CANNOT DO MEMORY-MEMORY TRANSFER WITH A SINGLE INSTRUCTION**
- **3 types of operands**
  - Immediate – for constant values
  - Register
  - Memory

Operand Combinations example

Source	Dest	Src,Dest*	C analog
Immediate	Register	movl $0x4, %eax	temp = 0x4;
Immediate	Memory	movl $-147, (%eax)	*p = -147;
Register	Register	movl %eax, %edx	temp2 = temp1;
Register	Memory	movl %eax, (%edx)	*p = temp;
Memory	Register	movl (%eax), %edx	temp = *p;

- Each statement should be viewed separately.
- REMINDER: cannot do memory-memory transfer with a single instruction.
- The parentheses around the register tell the assembler to use the register as a pointer.

Addressing Modes

Examples on next slide

- An *addressing mode* is a mechanism for specifying an address.
  - Immediate
  - Register
  - Memory
    - **Absolute**
      - **specify the address of the data**
    - **Indirect**
      - **use register to calculate address**
    - **Base + displacement**
      - **use register plus absolute address to calculate address**
    - **Indexed**
      - **Indexed**
        - » **Add contents of an index register**
      - **Scaled index**
        - » **Add contents of an index register scaled by a constant**

Operand addressing example

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11
Register	**Value**
%eax	0x100
%ecx	0x1
%edx	0x3

Operand	Value	Comment
%eax	0x100	Register
0x104	0xAB	Absolute Address - memory
$0x108	0x108	Immediate
(%eax)	0xFF	Address 0x100 - indirect
4(%eax)	0XAB	Address 0x104 - base+displacement
9(%eax,%edx)	0X11	Address 0x10C - indexed
260(%ecx,%edx)	0X13	Address 0x108 - indexed
0xFC(,%ecx,4)	0XFF	Address 0x100 - scaled index*
(%eax,%edx,4)	0X11	Address 0x10C - scaled index*

First two columns on left are given as is the Operand
FYI: 260 decimal = 0x104
*scaled index multiplies the 2nd argument by the scaled value (the 3rd argument) which must be a value of 1, 2, 4 or 8 (sizes of the primitive data types)

Operand addressing example EXPLAINED

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11
Register	**Value**
%eax	0x100
%ecx	0x1
%edx	0x3

Operand	Value	Comment
%eax	0x100	Value is in the register
0x104	0xAB	Value is at the address
$0x108	0x108	Value is the value ($ says "I'm an immediate, i.e. constant, value")
(%eax)	0xFF	Value is at the address stored in the register → GTV@(reg)
4(%eax)	0XAB	GTV@(4+ reg)
9(%eax,%edx)	0X11	GTV@(9 + reg + reg)
260(%ecx,%edx)	0X13	Same as above; be careful, in decimal
0xFC(,%ecx,4)	0XFF	GTV@(0xFC + 0 + reg*4)
(%eax,%edx,4)	0X11	GTV@(reg + reg*4)

In red are memory types of operands which is why you get the value at the address; because you are accessing memory

FYI: last two, the 3rd value in () is the scaling factor which must be 1, 2, 4 or 8

NOTE: Do not put '$' in front of constants when they are addressing indexes, only when they are literals.

Data movement instructions

- **Move, push and pop**
- **MOVE example**
- **Operands**
  - source,dest
- **Fill-in**
  - S = sign extend
  - Z = zero extend
- **b,w,l = byte, word, long**
  - 8, 16, 32 bits respectively
- **Instructions (a sample set)**
  - movb, movw, movl = S → D
  - movsbw, movsbl, movswl = SignExtend(S) → D
  - movzbw, movzbl, movzwl = ZeroExtend(S) → D

Given %dh = 0xCD and %eax = 0x98765432
What is in %eax after each instruction?

1. **movb %dh, %al** 987654CD
2. **movsbl %dh, %eax** FFFFFFCD
3. **movzbl %dh, %eax** 000000CD

Outline

- **Introduction of IA32**
- **IA32 operations**
  - Data movement operations
  - Stack operations and function calls
  - Arithmetic and logic operations
  - Compare and jump operations
- **Instruction encoding format**
- **Array and structures allocation and access**

Stack operations
Data movement instructions (cont)

- **Push and Pop**
- **Stack = LIFO**
- **pushl S**
  - R[%esp] – 4 → R[%esp]… decrement stack ptr
  - S → M[R[%esp]]… store to memory
  - Order matters!
- **popl D**
  - M[R[%ESP]] → D… reading from memory
  - R[%esp] + 4 → R[%esp]… increment stack ptr
  - Order matters!
- **By convention, we draw stacks upside down**
  - "top" of the stack is shown at the bottom
- **Stack "grows" toward lower addresses (push)**
  - Top element of the stack has the lowest address of all stack elements

The stack

```
subl $4, %esp
movl %eax, (%esp)
```

```
movl (%esp), %edx
addl $4, %esp
```

Initially

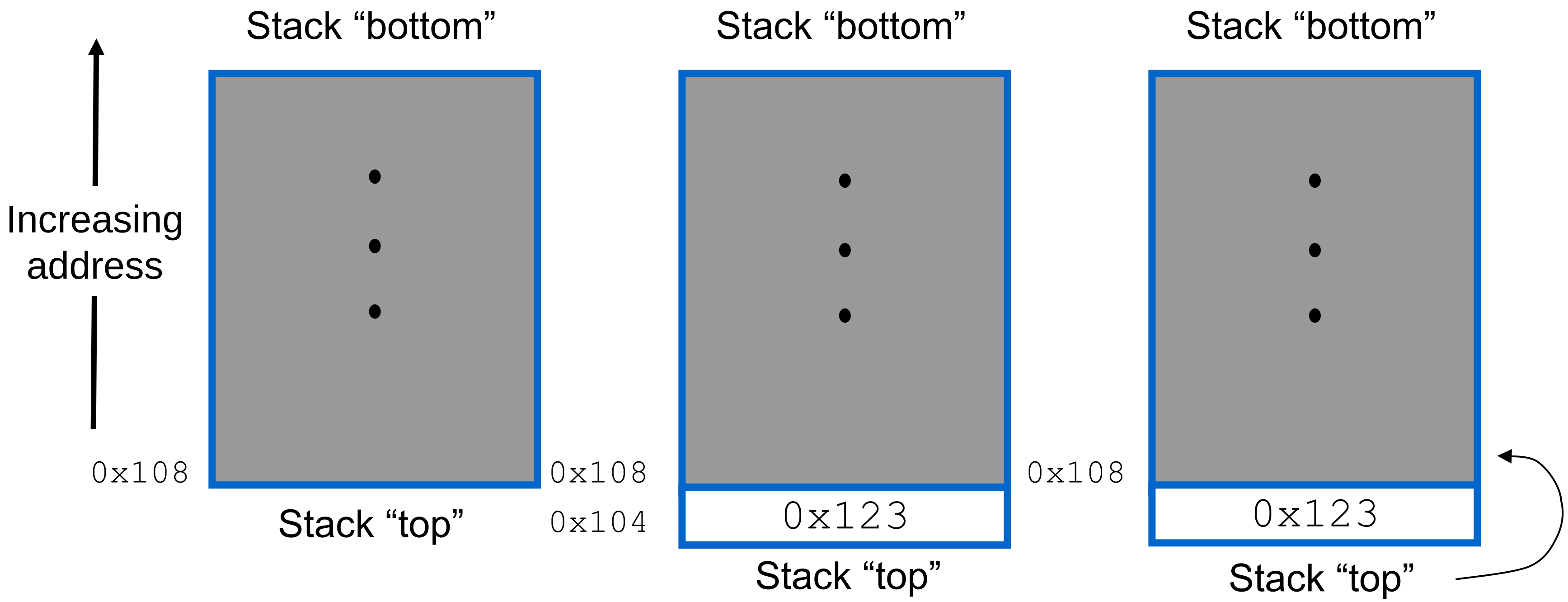
%eax	0x123
%edx	0
%esp	0x108

`pushl %eax`

%eax	0x123
%edx	0
%esp	0x104

`popl %edx`

%eax	0x123
%edx	0x123
%esp	0x108



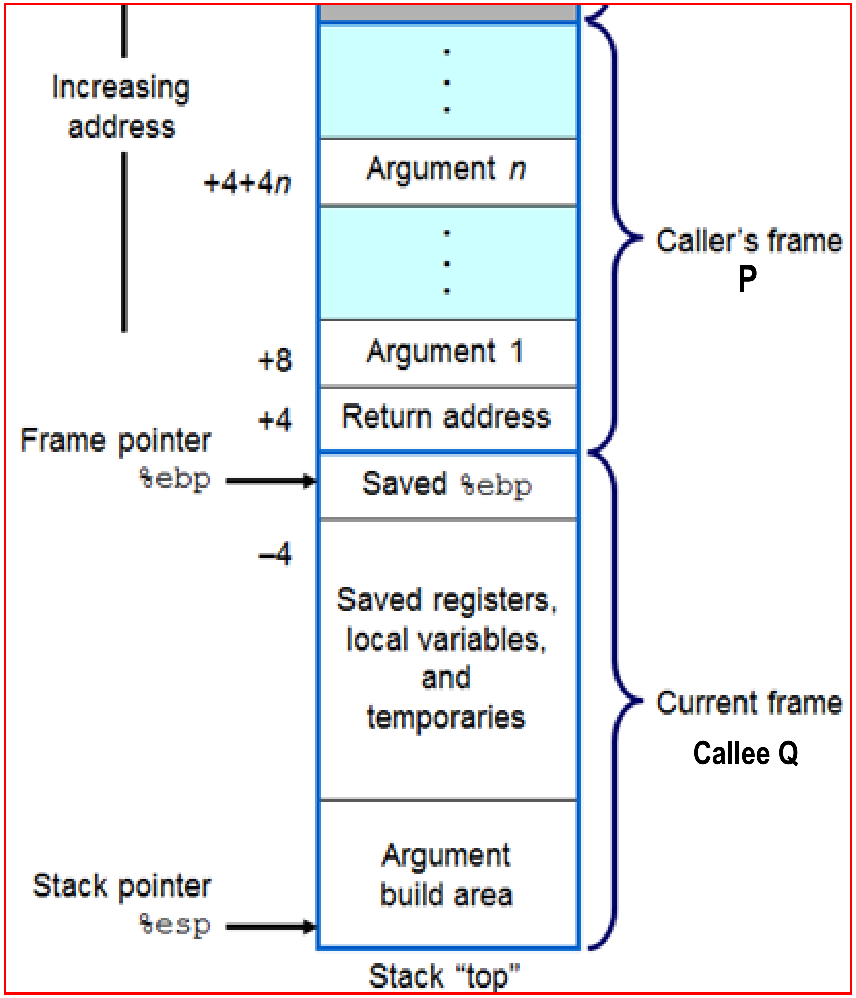
Procedure calls

- **The machine uses the stack to**
  - Pass procedure arguments
  - Store return information
  - Save registers for later restoration
  - Local storage
- **Stack frame**
  - Portion of the stack allocated for a single procedure call
  - The topmost stack frame is delimited by two pointers
    - Register %ebp – the frame/base pointer
    - Register %esp – the stack pointer
      - Can move while the procedure is executing HENCE
      - MOST INFORMATION IS ACCESSED RELATIVE TO THE FRAME/BASE POINTER
      - Indicates lowest stack address i.e. address of top element

Procedure calls (cont)

```
int P(int x)     {
  int y=x*x;
  int z=Q(y);
  return y+z;    }
```

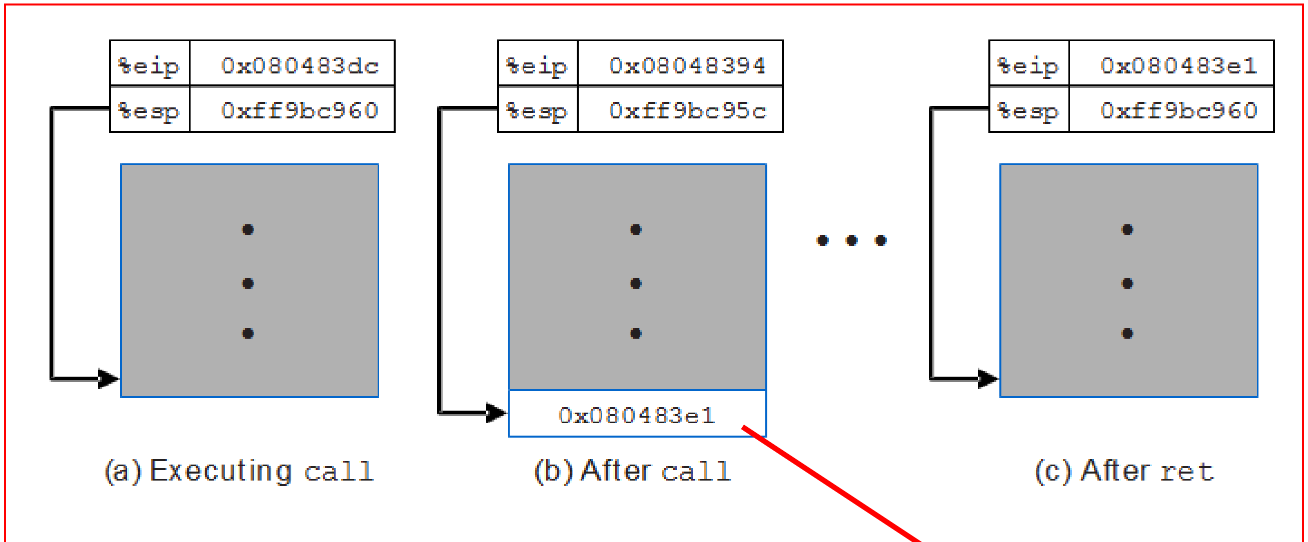
- **Procedure P (the "caller") calls procedure Q (the "callee")**
- **Caller stack frame (P)**
  - The arguments to Q are contained within the stack frame for P
  - The first argument is always positioned at offset 8 relative to %ebp
  - Remaining arguments stored in successive bytes (typically 4 bytes each but not always)… +4+4n is return address plus 4 bytes for each argument.
  - When P calls Q, the return address within P where the program should resume execution when it returns from Q is pushed on to the stack
- **Callee stack frame (Q)**
  - Saved value of the frame pointer
  - Copies of other saved registers
  - Local variables that cannot all be stored in registers (see next slide)
  - Stores arguments to any procedures it calls.



Procedure call and return

- **Call instruction**
  - Has a label which is a target indicating the address of the instruction where the called procedure (the callee) starts
  - Direct or indirect label
  - Push a return address on the stack
    - the address of the instruction immediately ***following the call*** in the (assembly) program
  - Jump to the start of the called procedure
- **Return instruction**
  - Pops an address off the stack
  - Jumps to this location
  - FYI: proper use is to have prepared the stack so that the stack pointer points to the place where the preceding call instruction stored its return address
- **Leave instruction is equivalent to:**
  - movl %ebp, %esp
  - popl %ebp

Procedure call and return



Return address

```
// Beginning of function sum
08048394  <sum>:
   8048394:   55                      push %ebp
…
//return from function sum
   80493a4:   c3                      ret
…
// call to sum from main  - START HERE!
   80483dc:   e8 b3 ff ff ff          call  8048394 <sum>
   80483e1:    83 c4 14               add $0x14,%esp
```

Callee function

Caller function

Register usage conventions

- **Program registers are a shared resource**
- **One procedure is active at a given time**
- **Don't want the callee to overwrite a value the caller planned to use later**
- **BY CONVENTION/PROTOCOL**
  - "Caller-save" registers: %eax, %edx and %ecx
    - When Q is called by P, it can overwrite these registers without destroying any data required by P
  - "Callee-save" registers: %ebx, %esi and %edi
    - Q must save these values on the stack before overwriting them, and restore them before returning
  - %ebp and %esp must be maintained
  - Register %eax is used for returning the value from any function that returns an integer or pointer.

```
int P(int x)
{
  int y=x*x;
  int z=Q(y);
  return y+z;
}
```

1. The caller, P, can save the value y.
2. P can store the value in a callee-save register (saved and restored).

Swap example

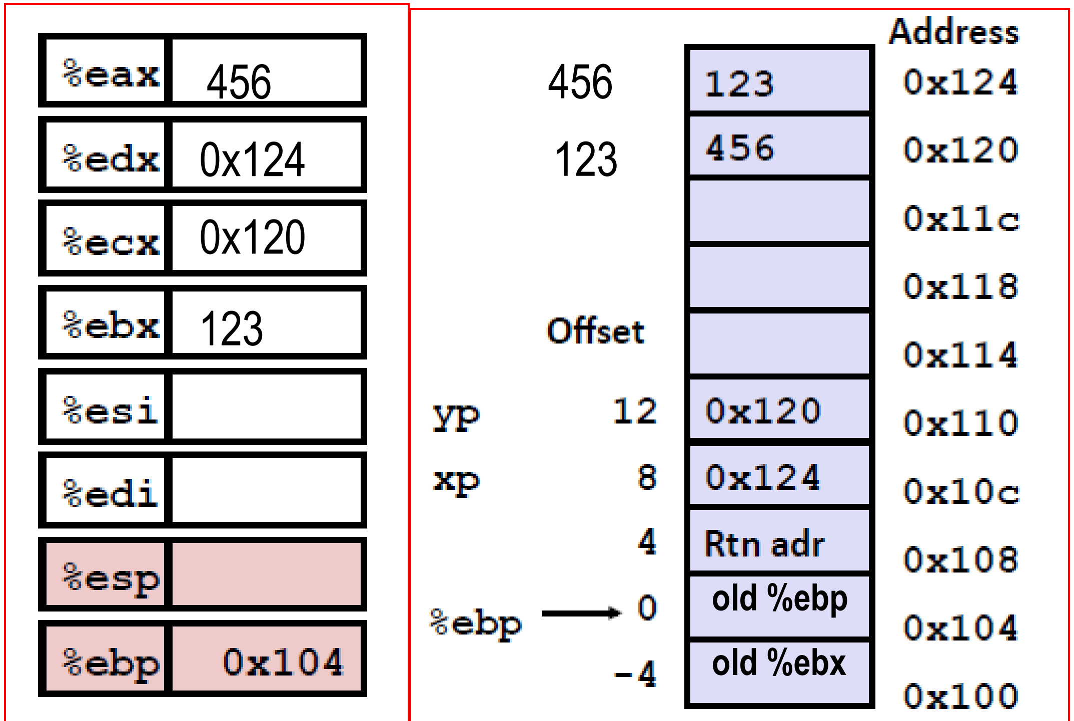
REMINDER: we use pointers so can pass address since can't pass values back outside of the function

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

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

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

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

    popl    %ebx              } Finish/epilogue
    popl    %ebp
    ret
```

Understanding Swap




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

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

1. Move 0x124 to %edx
2. Move 0x120 to %ecx
3. Move 123 to %ebx
4. Move 456 to %eax
5. Move 456 to M[0x124]
6. Move 123 to M[0x120]

```
popl    %ebx
popl    %ebp
ret
```

Example procedure call

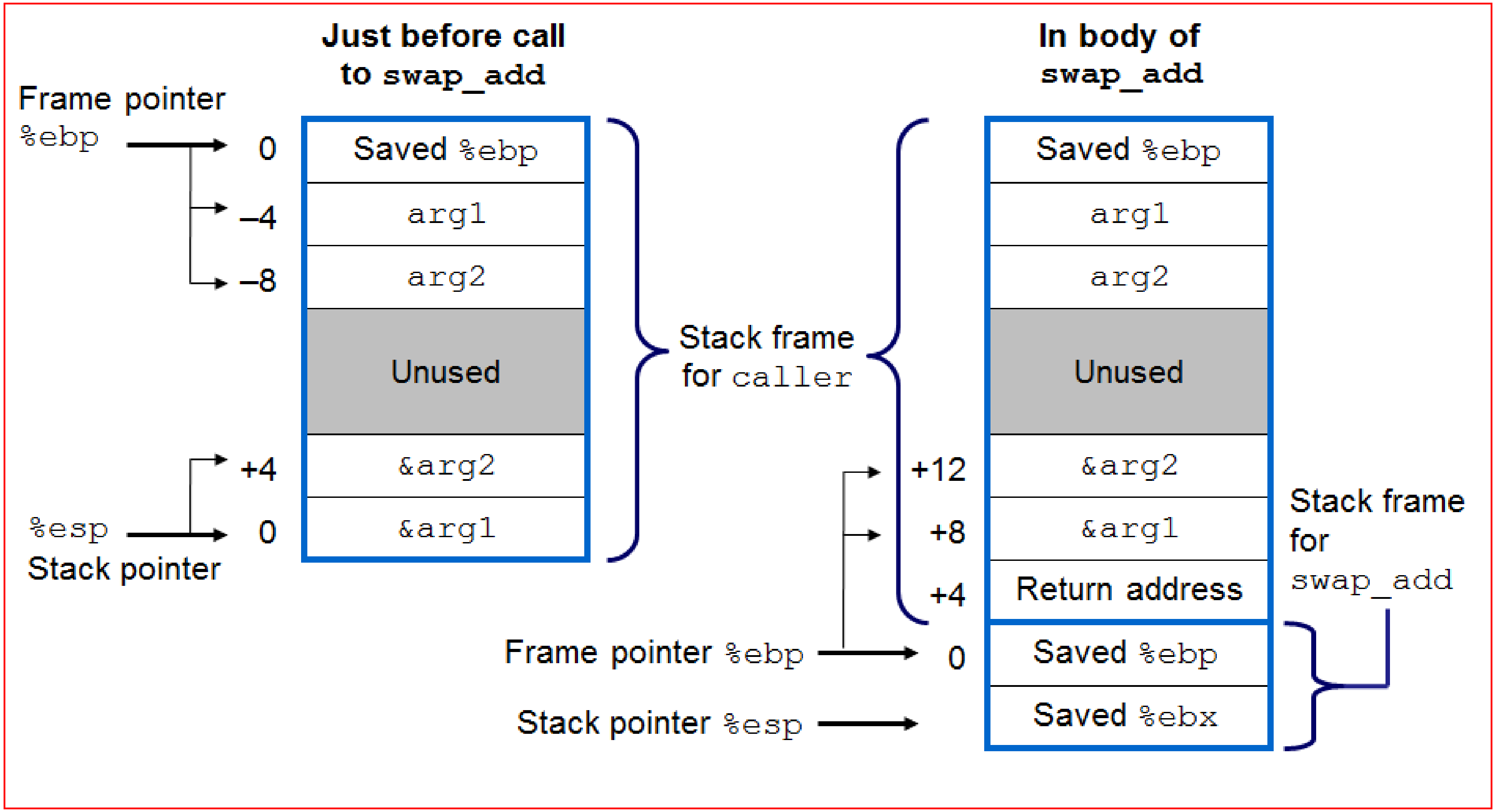
```
int swap_add(int *xp, int *yp)
{
  int x = *xp;
  int y = *yp;
  *xp = y;
  *yp = x;
  return x+y;
}

int caller()
{
  int arg1 = 534;
  int arg2 = 1057;
  int sum = swap_add(&arg1, &arg2);
  int diff = arg1 - arg2;
  return sum * diff;
} // callswap.c  and figure 3.23
```

```
caller:
  pushl   %ebp
  movl    %esp, %ebp
  subl    $24, %esp
  movl    $534, -4(%ebp)
  movl    $1057, -8(%ebp)
  leal    -8(%ebp), %eax
  movl    %eax, 4(%esp)
  leal    -4(%ebp), %eax
  movl    %eax, (%esp)
  call    swap_add
  movl    -4(%ebp), %edx
  subl    -8(%ebp), %edx
  imull   %edx, %eax
  leave
  ret
```

```
swap_add:
  pushl   %ebp
  movl    %esp, %ebp
  pushl   %ebx
  movl    8(%ebp), %ebx
  movl    12(%ebp), %ecx
  movl    (%ebx), %eax
  movl    (%ecx), %edx
  movl    %edx, (%ebx)
  movl    %eax, (%ecx)
  leal    (%edx,%eax), %eax
  popl    %ebx
  popl    %ebp
  ret
```

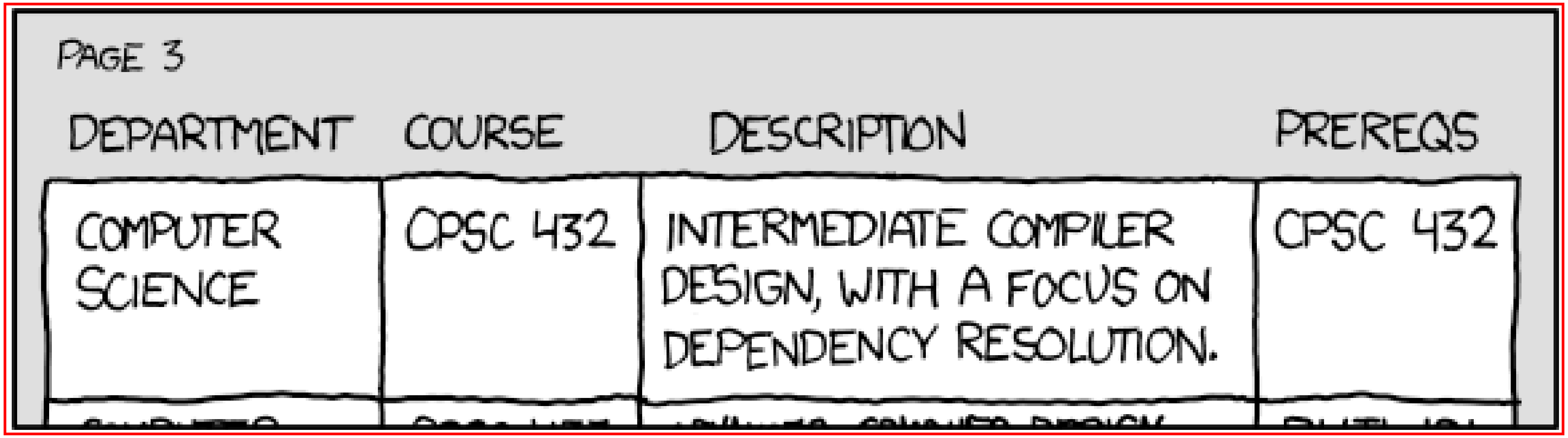
Stack frames for caller and swap_add

Fig 3.24



Recursion

- **Definition:**
  - In order to understand recursion, you must understand recursion



Recursive procedure

addr	Stack	comment
%esp	n = 3	
%esp	return addr	caller
%esp %ebp	%ebp	
%esp	%ebx	Caller value
	-4 to -16	unused
%esp	-20: 2	%ebx=3 %eax=1,2
%esp	return address	rfact
%esp %ebp	%ebp	
%esp	%ebx = 3	rfact value
	-4 to -16	unused
%esp	-20: 1	%ebx=2 %eax=1,1
%esp	return address	rfact
%esp %ebp	%ebp	
%esp	%ebx = 2	rfact value
	-4 to -16	unused
%esp	-20:	%ebx=1 %eax=1 jle .L3

POPPING:
%ebx = 2, 3
%eax = 1, 2, 6

```
int rfact(int n)    {
  int result;
  if (n <=1)
    result = 1;
  else
    result = n * rfact(n-1);
return result;    }
```

"multiple of 16 bytes" x86 programming guideline; including 4 bytes for the old %ebp and 4 bytes for the return address, caller uses 32 bytes; alignment issues (3.9.3)

CALL → Pushes the return address onto the stack (%esp-4 and mov); RETURN → pops it

```
rfact:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $20, %esp
    movl    8(%ebp), %ebx
    movl    $1, %eax
    cmpl    $1, %ebx
    jle     .L3
    leal    -1(%ebx), %eax
    movl    %eax, (%esp)
    call    rfact
    imull   %ebx, %eax
.L3:
    addl    $20, %esp
    popl    %ebx
    popl    %ebp
    ret
```

Outline

- **Introduction of IA32**
- **IA32 operations**
  - Data movement operations
  - Stack operations and function calls
  - Arithmetic and logic operations
  - Compare and jump operations
- **Instruction encoding format**
- **Array and structures allocation and access**

Arithmetic and Logical Operations

Instruction		Effect	Description
leal	S,D	&S --> D	load effective address
INC	D	D+1 --> D	increment
DEC	D	D-1 --> D	decrement
NEG	D	-D --> D	negate
NOT	D	~D --> D	complement
ADD	S, D	D + S --> D	add
SUB	S, D	D - S --> D	subtract
IMUL	S, D	D * S --> D	multiply
XOR	S, D	D ^ S --> D	exclusive-or
OR	S, D	D | S --> D	or
AND	S, D	D & S --> D	and
SAL	k, D	D << k --> D	left shift
SHL	k, D	D << k --> D	left shift (same as SAL)
SAR	k, D	D >> k -->$_A$ D	arithmetic right shift
SHR	k, D	D >> k -->$_L$ D	logical right shift

- **Watch out for argument order! see SUB**
- **No distinction between signed and unsigned int**
- **Notice A/L for arithmetic and logical right shifts**
- **Operation Groups**
  - Variant of the move
  - Unary
  - Binary
  - Shifts
- **Reminder: Note the difference in instruction between assemble and disassemble – just like the movl vs mov**

LEA – load effective address

- **Does not reference memory at all**
  - You don't get the value at the address… just the address (&x)
- **Copies the effective address to the destination**
- **Used to generate pointers for later memory references**
- **Can also be used to compactly describe common arithmetic operations**
- **The destination operand must be a register**

Example: leal 7 (%edx, %edx, 4) , %eax

Sets register %eax to 5x+7

%edx + %edx*4 + 7

Assume: %eax = x and %ecx= y	
INSTRUCTION	**RESULT**
leal 6(%eax), %edx	6 + x
leal (%eax, %ecx), %edx	x + y
leal (%eax, %ecx, 4), %edx	x + 4y
leal 7(%eax, %eax,8), %edx	7 + 9x
leal 0xA(,%ecx,4),%edx	10 + 4y
leal 9(%eax,%ecx,2), %edx	9 + x + 2y

Unary and Binary operations

- **Unary**
  - Single operand serves as both source and destination
  - Register or memory location
  - Similar to C ++ and -- operators
- **Binary**
  - Second operand is both source and destination
    - Thus cannot be an immediate value
    - Can be memory or register
  - First operand can be immediate, memory, or register
  - Reminder: both cannot be memory
  - Similar to C operations such as x += y

ADDRESS	VALUE
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

REGISTER	VALUE
%eax	0x100
%ecx	0x1
%edx	0x3

INSTRUCTION	DESTINATION	VALUE
addl %ecx, (%eax)	0x100	0x100
subl %edx, 4(%eax)	0x104	0xA8
imull $16,(%eax,%edx,4)	0x10C	0x110
incl 8(%eax)	0x108	0x14
decl %ecx	%ecx	0x0
subl %edx, %eax	%eax	0xFD

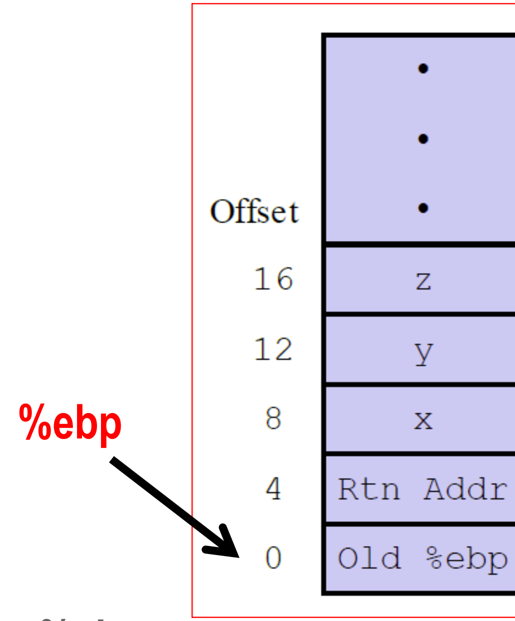
Shift operations

- **Shift amount given in first operand**
  - Coded as a single byte
  - Only shift amounts between 0 and 31 possible
    - Only low order 5 bits are considered
  - Immediate value or in the single byte register element %cl (unusual!)
- **Value to shift in second operand**
- **Arithmetic and logical**
  - Left shifts behave the same, though
    - Zero fill
  - Right shifts
    - sign extend (arithmetic)
    - zero fill (logical)

Discussion

- **Instructions work for unsigned or two's complement arithmetic**
  - Except right shift
- **Makes 2's comp arithmetic the preferred way to implement signed integer arithmetic**

Arithmetic example




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

```
00000000 <arith>:
  0:  55              push   %ebp
  1:  89 e5           mov    %esp,%ebp
  3:  8b 4d 08        mov    0x8(%ebp),%ecx
  6:  8b 55 0c        mov    0xc(%ebp),%edx
  9:  8d 04 52        lea    (%edx,%edx,2),%eax
  c:  c1 e0 04        shl    $0x4,%eax
  f:  8d 44 01 04     lea    0x4(%ecx,%eax,1),%eax
 13: 01 ca            add    %ecx,%edx
 15: 03 55 10         add    0x10(%ebp),%edx
 18: 0f af c2         imul   %edx,%eax
 1b: 5d               pop    %ebp
 1c: c3               ret
```

Outline

- **Introduction of IA32**
- **IA32 operations**
  - Data movement operations
  - Stack operations and function calls
  - Arithmetic and logic operations
  - Compare and jump operations
- **Instruction encoding format**
- **Array and structures allocation and access**

Overview of Compare and Jump

- Introduction with some examples
- Conditional codes & how to set CC
- How to use CC
- Control structures in assembly code

Control structures (in C)

- **Machine code provides two basic low-level mechanisms for implementing conditional behavior, tests data values then either**
  - Alters the control flow (conditional statement)
  - Alters the data flow (conditional expression)

```
int absdiff(int x, int y)    {
   if (x < y) return y – x;
   else       return x – y;   }
```

VS

```
int absdiff(int x, int y) {
   return x < y ? y – x : x-y;
}
```

```
int gotodiff(int x, int y)      {
     int result;
     if (x >= y)  goto x_ge_y;
     result = y – x;
     goto done;
x_ge_y:   result = x – y;
done:      return result;        }
```

```
int cmovdiff(int x, int y)   {
     int tval = y-x;
     int rval = x-y;
     int test = x < y;
     if (test) rval = tval;
     return rval;
}
```

Compares and Jumps Example

Using the JMP instruction, we may create a simple infinite loop that counts up from zero using the %eax register:

```
        MOVL $0, %eax
loop:   INCL %eax
        JMP loop
// unconditional  jump
```

Loop to count %eax from 0 to 5:

```
        MOVL $0, %eax
loop:   INCL %eax
        CMPL $5, %eax
        JLE loop
// conditional jump
//if %eax <=  5 then go to loop
```

- The jmp *label* instruction causes the processor to execute the next instruction at the location given by the label (i.e., the %eip is set to *label*).
- Conditional jump instructions will only transfer control if to the target of the appropriate flags are set.

Condition Code Flags

- **EXAMPLE: t = a + b**
  - a (= **1011**) + b (= **1000**) = 1 **0011**
- **CF set when: // unsigned overflow**
  - unsigned t < unsigned a
  - reminder: only positive values
  - carry-out == 1
  - How about unsigned sub: t = a – b, a < b, borrow == 1
- **ZF set when: t == 0 // zero**
- **SF set when: t < 0 // negative**
- **OF set when: // signed overflow**
  - (a<0 == b<0) && (t<0 != a<0)
    - (a<0 && b<0 && t>=0) || (a>0 && b>0 && t<0)

Technically…

- **Arithmetic and logical operators set the EFLAGS**

Instruction		Effect	Description
leal	S,D	&S --> D	load effective address
INC	D	D+1 --> D	increment
DEC	D	D-1 --> D	decrement
NEG	D	-D --> D	negate
NOT	D	~D --> D	complement
ADD	S, D	D + S --> D	add
SUB	S, D	D - S --> D	subtract
IMUL	S, D	D * S --> D	multiply
XOR	S, D	D ^ S --> D	exclusive-or
OR	S, D	D | S --> D	or
AND	S, D	D & S --> D	and
SAL	k, D	D << k --> D	left shift
SHL	k, D	D << k --> D	left shift (same as SAL)
SAR	k, D	D >> k -->$_A$ D	arithmetic right shift
SHR	k, D	D >> k -->$_L$ D	logical right shift

Leal does not alter any condition codes (since intended use is address computations – pg. 420)

Logical operations carry and overflow flags are set to 0 (ex. XOR pg. 845)

Shift operations, the carry flags is set to the last bit shifted out; the overflow flag is set to 0 (pg. 741)

INC/DEC set overflow and zero flags; and leave carry flag unchanged.

* Check ISA manual

Compare instruction

- **These instructions set the condition codes without updating any other registers**
- **CMPx S1, S2 → S2-S1**
  - The x can be a b, w or l for byte, word or long
- **CMP acts like the SUB without updating the destination**
  - ZF set if a == b
  - SF set if (a-b) < 0
  - CF set if carry out from MSB = 1
  - OF set if 2’s comp overflow
    - (a>0 && b<0 && (a-b)<0 || (a<0 && b>0 && (a-b)>0)

Test instruction

- **The TEST operation sets the flags CF and OF to zero. The SF is set to the MSB of the result of the AND. If the result of the AND is 0, the ZF is set to 1, otherwise set to 0.**
- **TEST acts like the AND without updating the destination… testx s1, s2 → s1 & s2**
  - ZF set when a&b == 0
  - SF set when a&b < 0
  - OF/CF are set to 0 (not used)
  - Example: same operand repeated to see whether the operand is negative, zero or positive
    - testl %eax, %eax
      - sets ZF to 1 if %eax == 0
      - sets SF to 1 if %eax < 0 (i.e. negative) and 0 if %eax > 0 (i.e. positive)
  - One of the operands is a mask indicating which bits should be tested
    - testl 0xFF, %eax

Accessing the Condition Codes

- **3 common ways to use condition codes:**
  - SET
    - Set a single byte to 0 or 1 depending on some combination of the condition codes
  - JMP
    - Conditionally jump to some other part of the program
  - CMOV
    - Conditionally transfer data

Set instructions

- **Sets a single byte to 0 or 1 based on combinations of condition codes**
- **Each set instruction has a designated destination:**
  - Byte register
    - One of 8 addressable byte registers embedded within first 4 integer registers
    - Does not alter remaining 3 bytes
    - Typically use movzbl to finish the job
  - Single-byte memory location

SET instruction options

Instruction	Condition	Synonym	Description
sete	D ← ZF	setz	equal / zero
setne	D ← ~ZF	setnz	not equal / not zero
sets	D ← SF		negative
setns	D ← ~SF		nonnegative
setg	D ← ~(SF ^ OF) & ~ZF	setnle	greater (signed >)
setge	D ← ~(SF ^ OF)	setnl	greater or equal (signed >=)
setl	D ← SF ^ OF	setnge	less (signed <)
setle	D ← (SF ^ OF) | ZF	setng	less or equal (signed <=)
seta	D ← ~CF & ~ZF	setnbe	above (unsigned >)
setb	D ← CF	setnae	below (unsigned <)

Multiple possible names for the instructions called synonyms.
Compilers and disassemblers make arbitrary choices of which names to use.
Note CF only on unsigned options

Set instruction examples

```
// is a < b?
                        // a = %edx, b = %eax
cmpl %eax, %edx         // a-b i.e. %edx - %eax
                        // flags set by cmpl
setl %al                // D ← SF ^ OF
movzbl %al, %eax        // clear high order 3 bytes
// if %al has a 1 in it, then the answer is yes
// if %al has a 0 in it, then the answer is no
```

notice cmp**L** and set**L** are NOT the same thing

```
// another example
movl 12(%ebp), %eax     // eax = y
cmpl %eax, 8(%ebp)      // compare x:y (x-y)
setg %al                 //  al = x > y
movzbl %al, %eax         //  zero rest of eax
```

FLAGS:
If a = b then ZF = 1 → a-b=0
If a < b then SF = 1 → a-b<0 (#2)
If a > b then SF = 0 → a-b>0
If a<0, b>0, t>0 then OF=1 (#1)
If a>0, b<0, t<0 then OF=1
If unsigned… CF (not interested)

SF ^	OF	→	D	
0	0	=	0	
0	1	=	1	(see #1 below)
1	0	=	1	(see #2 below)
1	1	=	0	

So, a < b when D = 1
#1 a is neg, b is pos, t is pos
#2 a-b<0 means a<b

Jump instructions

Instruction	Condition	Description
jmp	1	unconditional
je *label*	ZF	equal
jne *label*	~ZF	not equal
js *label*	SF	negative
jns *label*	~SF	nonnegative
jg *label*	~(SF ^ OF) & ~ZF	greater (signed)
jge *label*	~(SF ^ OF)	greater or equal (signed)
jl *label*	SF ^ OF	less (signed)
jle *label*	(SF ^ OF) | ZF	less or equal (signed)
ja *label*	~CF & ~ZF	above (unsigned)
jb *label*	CF	below (unsigned)

There are synonyms for jump instructions as well

The test and cmp instructions are combined with the *conditional and unconditional jmp instructions* to implement most relational and logical expressions and all control structures.

Set allows us to know what the condition evaluates to if something other than jmp to be done.

Conditional moves

```
#define OP _______
int arith(int x) {  return x OP 4;  }
```

```
// What operation is OP? Fill in the comments to explain how the code works.
// x is in %edx… for example, what if x = 16? What if x = -8?
leal     3(%edx), %eax      //  temp = x+3
testl    %edx, %edx         //  test x – sets ZF and SF
cmovns   %edx, %eax         //  if x >= 0, temp = x
sarl     $2, %eax           //  return temp >> 2  =  x/4  return value in %eax
```

Instruction	Synonym	Move condition	Description
cmove *S, D*	cmovz	ZF	Equal / zero
cmovne *S, D*	cmovnz	~ZF	Not equal / not zero
cmovs *S, D*		SF	Negative
cmovns *S, D*		~SF	Nonnegative
cmovg *S, D*	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge *S, D*	cmovnl	~(SF ^ OF)	Greater or equal (signed >=)
cmovl *S, D*	cmovnge	SF ^ OF	Less (signed <)
cmovle *S, D*	cmovng	(SF ^ OF) | ZF	Less or equal (signed <=)
cmova *S, D*	cmovnbe	~CF & ~ZF	Above (unsigned >)
cmovae *S, D*	cmovnb	~CF	Above or equal (Unsigned >=)
cmovb *S, D*	cmovnae	CF	Below (unsigned <)
cmovbe *S, D*	cmovna	CF | ZF	below or equal (unsigned <=)

ANSWER:
Divide is the OP
Add 3 because:
If x is negative, it requires biasing in order to divide by 4 i.e.
2^k-1 = 3
Since and k = 2

Example overview

```
if ( a == b ) x = 1;

    cmpl a, b       // (b-a) == 0
    jne skip        //not equal, so skip
    movl $1, x    //  since a == b, x = 1
skip:
    nop             // no operation…???
```

```
if ( a > b ) x = 1;

    cmpl b, a  // (a-b) > 0
    jle skip     // skip if a <= b
    movl $1, x
skip:
```

```
cmpl a,b
jge skip
```

```
// Counts the number of bits set to 1
int count = 0;
int loop = 32;
do {
     if ( x & 1 ) count++;
      x >>= 1;
      loop--;
} while ( loop != 0 )

    movl $0, count
    movl $32, loop
.L2:
    movl x, %eax
    andl $1, %eax
    testl %eax, %eax
    je .L5
     incl count
.L5:
    sarl x
    decl loop
    cmpl $0, loop
    jne .L2
```

Conditional branch example

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

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

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

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;
}
```

Use backward branch to continue looping

Only take branch when “while” condition holds

Reminder: “Test” is expression return an integer of 1 when true and 0 when false

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;
}
```

"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;
}
```

Registers

```
%edx   x
%eax   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
```

"While" loop translation

Is this code equivalent to the do-while version? Must jump out of loop if test fails

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

C Code

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

First Goto Version

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

Second 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;
}
```

While vs DoWhile

```
fact_while:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        movl    $1, %eax
        cmpl    $1, %edx
        jle     .L3
.L6:
        imull   %edx, %eax
        subl    $1, %edx
        cmpl    $1, %edx
        jne     .L6
.L3:
        popl    %ebp
        ret
```

```
fact_dowhile:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        movl    $1, %eax
.L2:
        imull   %edx, %eax
        subl    $1, %edx
        cmpl    $1, %edx
        jg      .L2
        popl    %ebp
        ret
```

"For" loop translation

```
int result;
for (result = 1;
     p != 0;
     p = p>>1) {
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

Goto Version

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

→

```
  result = 1;
  if (p == 0)
    goto done;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
  if (p != 0)
    goto loop;
done:
```

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

"For" loop example

cmov (conditional move) only transfers the data if the condition is true

```
// compute x raised to the
// nonnegative power p
int ipwr_for(int x, unsigned p)
{
  int result;
  for (result = 1; p != 0; p = p>>1)
   {
     if (p & 0x1)
              result *= x;
     x = x * x;
   }
  return result;
}
```

Example walkthrough
x=2, p=4

```
ipwr_for:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        movl    8(%ebp), %ecx      // x
        movl    12(%ebp), %edx     // p
        movl    $1, %eax           // result
        testl   %edx, %edx         // set cc
        je      .L4                // ZF=1 iff %edx == 0
.L5:
        movl    %eax, %ebx   //  temp result in ebx
        imull   %ecx, %ebx   // new result (* x)
        testb   $1, %dl      // If cond
        cmovne  %ebx, %eax   //  ~ZF update result
        shrl    %edx
        je      .L4
        imull   %ecx, %ecx   // x*x
        jmp     .L5
.L4:
        popl    %ebx
        popl    %ebp
        ret
```

Disassembly of ipwr_for

```
0:   55              push   %ebp
1:   89 e5           mov    %esp,%ebp
3:   53              push   %ebx
4:   8b 4d 08        mov    0x8(%ebp),%ecx
7:   8b 55 0c        mov    0xc(%ebp),%edx
a:   b8 01 00 00 00  mov    $0x1,%eax
f:   85 d2           test   %edx,%edx
11:  74 14           je     27 <ipwr_for+0x27>
13:  89 c3           mov    %eax,%ebx
15:  0f af d9        imul   %ecx,%ebx
18:  f6 c2 01        test   $0x1,%dl
1b:  0f 45 c3        cmovne %ebx,%eax
1e:  d1 ea           shr    %edx
20:  74 05           je     27 <ipwr_for+0x27>
22:  0f af c9        imul   %ecx,%ecx
25:  eb ec           jmp    13 <ipwr_for+0x13>
27:  5b              pop    %ebx
28:  5d              pop    %ebp
29:  c3              ret
```

- **cmov (conditional move) only transfers the data if the condition is true**

Switch Statements

- **Implementation options**
  - Series of conditionals
    - Good in 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
  - Usually should also specify “default:” case

```
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 '?';
  }
}
```

switchasm.c

Jump table structure

FYI: ***Direct jump*** **is an encoded target as part of the instruction**

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;
```

- ***Indirect jump*** → *operand
  - Operand is typically a register
    - *%eax where reg is the target value; OR
    - *(%eax) where jump target is read from memory

JUMP TABLE:

An array where entry i is the address of a code segment implementing the action the program should take when the switch index equals i.

Lookup branch target

Avoids conditionals

Possible when cases are small integer constants

Switch statement example

Branching Possibilities

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

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

Symbolic Labels

- Labels of form `.LXX` translated into addresses by assembler

Table Structure

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

```
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]
```

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`

Enumerated Values

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

Sparse "switch" example

```
/* Return x/111 if x is multiple
   && <= 999.  -1 otherwise */
int div111(int x)
{
  switch(x) {
  case   0: return 0;
  case 111: return 1;
  case 222: return 2;
  case 333: return 3;
  case 444: return 4;
  case 555: return 5;
  case 666: return 6;
  case 777: return 7;
  case 888: return 8;
  case 999: return 9;
  default: return -1;
  }
}
```

Not practical to use jump table

- Would require 1000 entries

Obvious translation into if-then-else would have max. of 9 tests

Outline

- **Introduction of IA32**
- **IA32 operations**
  - Data movement operations
  - Stack operations and function calls
  - Arithmetic and logic operations
  - Compare and jump operations
- **Instruction encoding format**
- **Array and structures allocation and access**

Instruction formats for swap

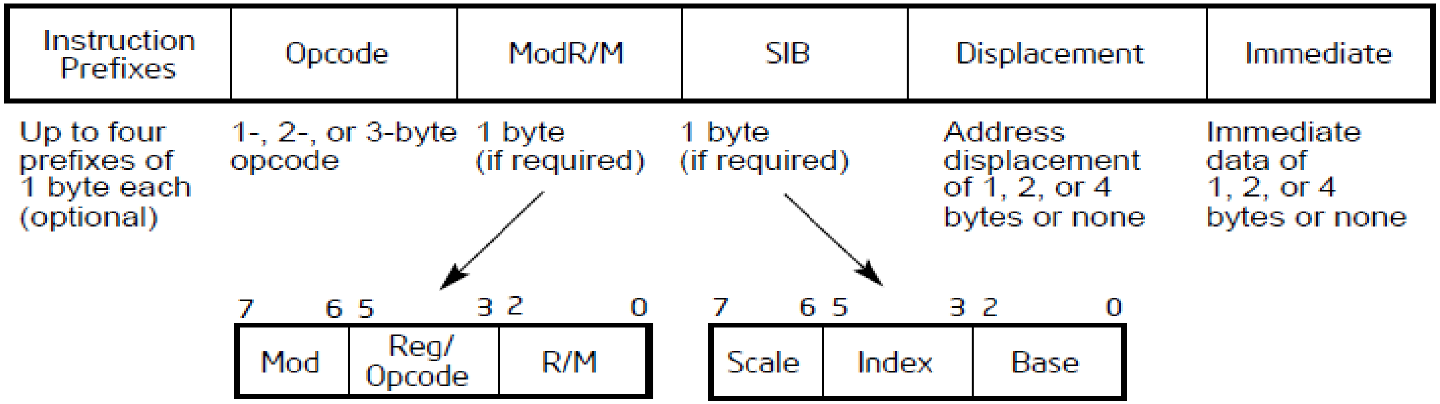
00000000 <swap>:	opcode	ModR/M	SIB	Displacement	Immediate
0: 55 push %ebp	55				
1: 89 e5 mov %esp,%ebp	89	11 100 101			
3: 53 push %ebx	53				
4: 8b 55 08 mov 0x8(%ebp),%edx	8b	01 010 101		0000 1000	
7: 8b 45 0c mov 0xc(%ebp),%eax	8b	01 000 101		0000 1100	
a: 8b 0a mov (%edx),%ecx	8b	00 001 010			
c: 8b 18 mov (%eax),%ebx	8b	00 011 000			
e: 89 1a mov %ebx,(%edx)	89	00 011 010			
10: 89 08 mov %ecx,(%eax)	89	00 001 000			
12: 5b pop %ebx	5b				
13: 5d pop %ebp	5d				
14: c3 ret	c3				

http://www.cs.princeton.edu/courses/archive/spr11/cos217/reading/ia32vol2.pdf

PUSH pg 701; MOV pg 479; POP pg 637; RET pg 28

Instruction Format

- **All IA-32 instruction encodings are subsets of the general instruction format shown below, in the given order**
- **Instructions consist of:**
  - optional instruction prefixes (in any order)
  - 1-3 opcode bytes – determines the action of the statement
  - an addressing-form specifier (if required) consisting of:
    - the ModR/M byte - addressing modes register/memory
    - sometimes the SIB (Scale-Index-Base) byte
    - a displacement (if required)
    - an immediate data field (if required).



ModR/M

ModR/M

Mod	Reg #	R/M
2 bits	3 bits	3 bits

- **Mod=00,**
  - First operand a register, specified by Reg #
  - Second operand in memory; address stored in a register numbered by R/M.
    - That is, Memory[Reg[R/M]]
  - Exceptions:
    - R/M=100 (SP): SIB needed
    - R/M=101 (BP): disp32 needed
- **Mod=01, same as Mod 00 with 8-bit displacement.**
  - Second operand: Memory[disp8+Reg[R/M].
  - Exception: SIB needed when R/M=100
- **Mod=10, same as Mod 01 with 32-bit displacement**
- **Mod=11**
  - Second operand is also a register, numbered by R/M.
- **Do not confuse displacement width with data width.**
  - Data width is specified by the opcode.
  - For example, the use of disp8 does not imply 8-bit data.

For some opcodes, the reg# is used as an extension of the opcode.

SIB displacement and immediate

- **SIB**
  - Specify how a memory address is calculated
  - Address = Reg[base] + Reg[Index] * 2^{scale}
  - Exceptions:
    - SP cannot be an index, and
    - BP cannot be a base

Scale	Index	Base
2 bits	3 bits	3 bits

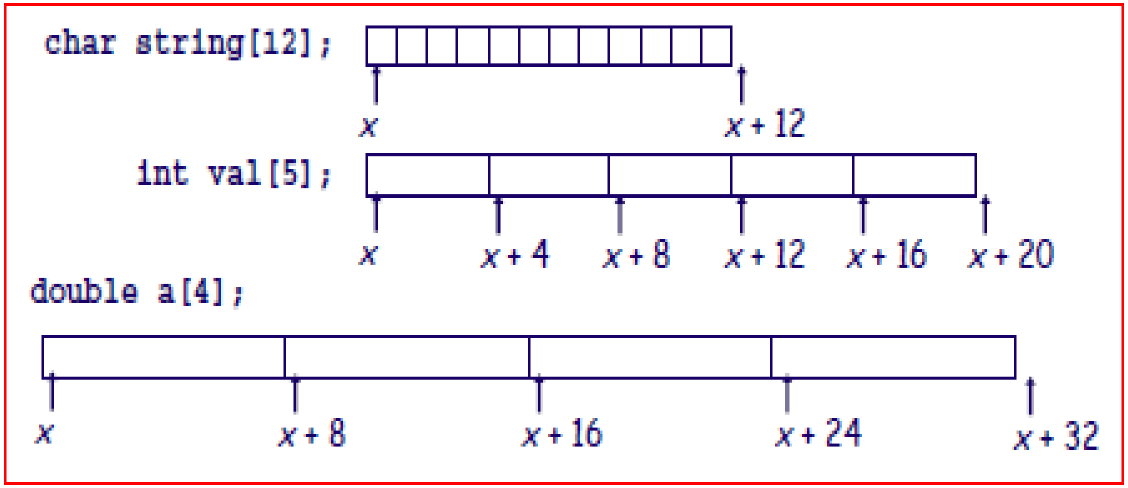
- **Displacement**
  - Can immediately follow ModR/M byte
  - 1, 2, or 4 bytes
- **Immediate**
  - Immediate operand value always follows any displacement bytes
  - 1, 2 or 4 bytes

Outline

- **Introduction of IA32**
- **IA32 operations**
  - Data movement operations
  - Stack operations and function calls
  - Arithmetic and logic operations
  - Compare and jump operations
- **Instruction encoding format**
- **Array and structures allocation and access**

Array allocation and access

- **type array[length]**
  - Contiguously <u>*allocated*</u> region of length * sizeof(T) bytes
  - Starting location of array is a pointer (x)
  - <u>*Access*</u> array elements using integer index i ranging between 0 and length-1 (i.e. the subscript)
    - Array element i will be stored at address x+sizeof(T)*i



Total size: 12, 20, & 32

Element i:

x + 1*i

x + 4*i

x + 8*i

Address of array in %edx and i stored in %ecx

➔ **movl (%edx,%ecx,4)**

Array allocation and access (cont)

- **Explains why scaled factors are 1, 2, 4, and 8**
  - The primitive data types
- **Problem 3.35 (pg 233)**
- **IA32**
  - A pointer of any kind is 4 bytes long
  - GCC allocates 12 bytes for the data type long double
    - 4 bytes for float and pointers, 8 bytes for double, 12 bytes for long double

Given	Array	Element size	Total Size	Start address	Element i
short S[7]	S	2	14	x_s	x_s + 2i
short *T[3]	T	4	12	x_t	x_t + 4i
long double V[8]	V	12	96	x_v	x_v + 12i
long double *W[4]	W	4	16	x_w	x_w + 4i

Pointer arithmetic

- **Reminders...**
  - C allows arithmetic on pointers, where the computed value is scaled according to the size of the data type referenced by the pointer
    - So, if p is a pointer to data type T
    - And, the value of p is x_p
    - Then, then p+i has value x_p + L*i
    - Where, L is the size of data type T
    - Thus A[i] == *(A+i)
- **Example**
  - %edx → starting address of array E
  - %ecx → integer index i

Expression	Type	Value	Assembly code… result in %eax	Comment
E	int *	x_e	movl %edx, %eax	
E[0]	int	M[x_e]	movl (%edx, %ecx,4), %eax	Reference memory
E[i]	int	M[x_e + 4i]	movl (%edx, %ecx,4), %eax	Reference memory
&E[2]	int *	x_e + 8	leal 8(%edx), %eax	Generate address
E+i-1	int *	x_e + 4i - 4	leal -4(%edx,%ecx,4), %eax	Generate address
*(E+i-3)	int *	M[x_e + 4i -12]	movl -12(%edx, %ecx,4), %eax	Reference memory
&E[i]-E	int	i	movl %ecx, %eax	

Structures

- **Reminder… the C struct declaration creates a data type that groups objects of possibly different types into a single object**
- **Implementation similar to arrays**
  - All components are stored in a contiguous region of memory
  - A pointer to a structure is the address of its first byte
- **The compiler maintains information about each structure type indicating the byte offset of each field**
  - Generates references to structure elements using these offsets as displacements in memory referencing instructions

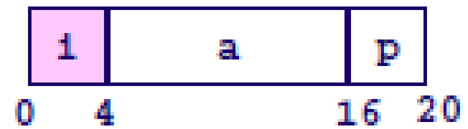
Structure allocation

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;
}
```

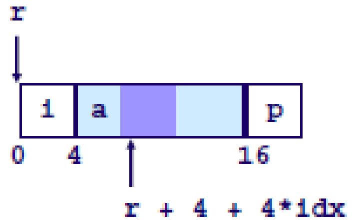
Assembly

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

Structure Access

Generating Ptr to Structure 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
```

leal 4(%edx, %ecx, 4)

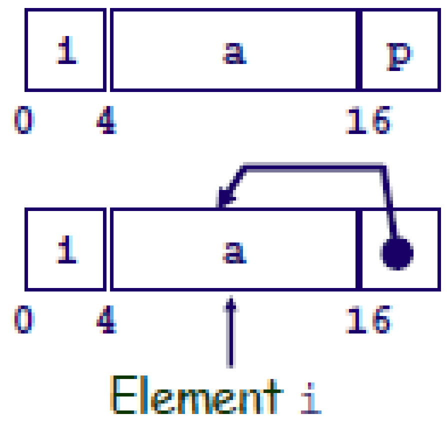
```
find_a:
  pushl   %ebp
  movl    %esp, %ebp
  movl    12(%ebp), %eax     // idx (2nd arg)
  sall     $2, %eax           // mult by 4
  addl    8(%ebp), %eax      // ptr to struct (1st arg)
  addl    $4, %eax
  popl    %ebp
  ret
```

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];
}
```



"i" represents the element of "a" that I want "p" to point to

```
 # %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
```

Data 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
  - Virtual memory very tricky when datum spans 2 pages

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

<u>IA32/LINUX address</u>

2 bytes hex: ends in even hex digit (0, 2, 4, 6, 8, A, C, E)

4 bytes hex: ends in divisible by 4 hex digit (0,4,8,C)

8 bytes hex: ends in divisible by 8 hex digit (0,8)

Satisfying alignment in structures

Offsets Within Structure

- Must satisfy element's alignment requirement

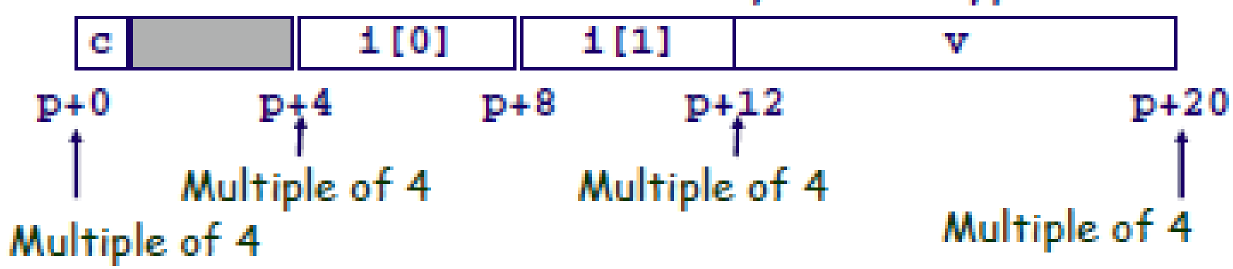
Overall Structure Placement

- 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;
```

Linux:

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



Long long treated like 8-byte data type

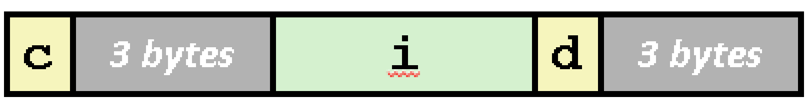
Saving space

- **Put large data types first**

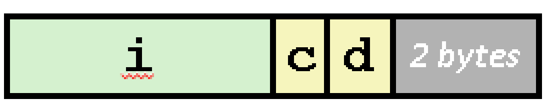
```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- **Effect (K=4)**



Total bytes = 12

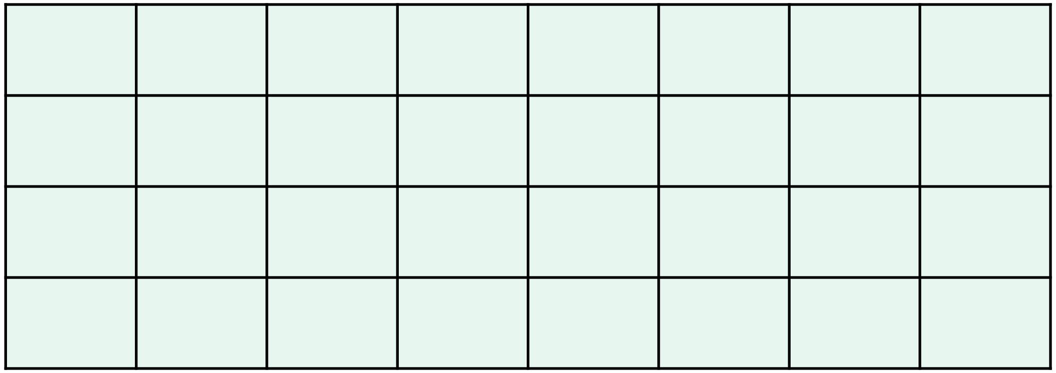


Total bytes = 8

Another Example

```
struct a_struct {
        char             a;
        struct a_strcut *b;
};

struct b_struct {
        char            c;
        int             i;
        double *        d;
        short           e[3];
        struct a_struct m;
};
```

Each block is a byte



```
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
 +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 | c   X   X   X | i   i   i   i | d   d   d   d | e   e   e   e |
 +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 | e   e   X   X | a   X   X   X | b   b   b   b |               |
 +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

End IA32