`movl` Src,Dest	Dest = Src	
`movsbl` Src,Dest	Dest (long) = Src (byte), sign extend	
`addl` Src,Dest	Dest = Dest + Src	
`subl` Src,Dest	Dest = Dest – Src	
`imull` Src,Dest	Dest = Dest * Src	
`sall` Src,Dest	Dest = Dest << Src	
`sarl` Src,Dest	Dest = Dest >> Src	arithmetic shift
`shrl` Src,Dest	Dest = Dest >> Src	logical shift
`xorl` Src,Dest	Dest = Dest ^ Src	
`andl` Src,Dest	Dest = Dest & Src	
`orl` Src,Dest	Dest = Dest | Src	
`incl` Dest	Dest = Dest + 1	
`decl` Dest	Dest = Dest - 1	
`negl` Dest	Dest = – Dest	
`notl` Dest	Dest = ~ Dest	
`leal` Src,Dest	Dest = address of Src	
`cmpl` Src2,Src1	Sets CCs Src1 – Src2	
`testl` Src2,Src1	Sets CCs Src1 & Src2	

- **Immediate**
  $val Val
  val: constant integer value
  `movl $17, %eax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movl (%ecx), %eax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movl 8(%ebp), %edx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: constant displacement 1, 2, or 4 bytes
  Rb: base register: any of 8 integer registers
  Ri: index register: any, except `%esp`
  S: scale: 1, 2, 4, or 8
  `movl 0x100(%ecx,%eax,4), %edx`

`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed)
`jge label`	jump greater or equal (signed)
`jl label`	jump less (signed)
`jle label`	jump less or equal (signed)
`ja label`	jump above (unsigned)
`jb label`	jump below (unsigned)
`push` Src	`%esp` = `%esp` – 4, Mem[`%esp`] = Src
`pop` Dest	Dest = Mem[`%esp`], `%esp` = `%esp` + 4
`call label`	push address of next instruction, `jmp label`
`ret`	`%eip` = Mem[`%esp`], `%esp` = `%esp` + 4

b byte
w word (2 bytes)
l long (4 bytes)

Condition codes

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

Registers

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

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111
Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Figure 2.2 **Hexadecimal notation.** Each Hex digit encodes one of 16 values.

~		&	0	1	|	0	1	^	0	1
0	1	0	0	0	0	0	1	0	0	1
1	0	1	0	1	1	1	1	1	1	0

Figure 2.7 **Operations of Boolean algebra.** Binary values 1 and 0 encode logic values TRUE and FALSE, while operations ~, &, |, and ^ encode logical operations NOT, AND, OR, and EXCLUSIVE-OR, respectively.

Endianness

Little Endian: Least significant digit comes first
Big endian: Least significant digit comes last
For example: 0xFFAE07
Little endian: 07 AE FF
Big endian: FF AE 07

Bit Shifting

Operation	Values	
Argument x	[01100011]	[10010101]
x << 4	[00110000]	[01010000]
x >> 4 (logical)	[00000110]	[00001001]
x >> 4 (arithmetic)	[00000110]	[11111001]

```c
void inplace_swap(int *x, int *y) {
    *y = *x ^ *y;  /* Step 1 */
    *x = *x ^ *y;  /* Step 2 */
    *y = *x ^ *y;  /* Step 3 */
}
```

Two's Complement

$$B2T_4([0001]) = -0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 0+0+0+1 = 1$$
$$B2T_4([0101]) = -0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 0+4+0+1 = 5$$
$$B2T_4([1011]) = -1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = -8+0+2+1 = -5$$
$$B2T_4([1111]) = -1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = -8+4+2+1 = -1$$

Multiplication by Bitwise Shifting Left

x*14
14 = 2^3 + 2^2 + 2^1
(x << 3) + (x << 2) + (x << 1)
14 = 2^4 – 2^1
(x << 4) - (x << 1)

Division by Powers of Two Using Bitwise Shifting Right

x/2^k

(x < 0 ? x + (1 << k) - 1 : x) >> k

IEEE Floating Point Representation

	sign	exp	mantissa	bias	
single	1	8	23	127	(32-bit)
double	1	11	52	1023	(64-bit)

bias = 2^(k - 1) - 1 where k = number of bits in exp

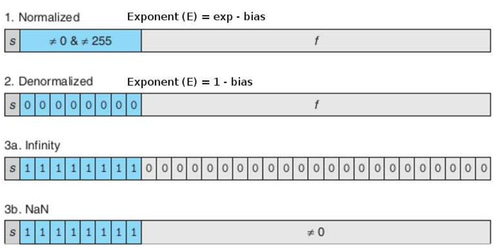


Figure 2.32 **Categories of single-precision, floating-point values.** The value of the exponent determines whether the number is (1) normalized, (2) denormalized, or a (3) special value.

Example

	sign	exp			mantissa			
-13.9 (base 10) in 8 bits	1	0	0	1	1	1	0	1

-13 (base 10) = ? (base 2)
13/2 = 6 r 1
6/2 = 3 r 0
3/2 = 1 r 1
1/2 = 0 r 1

0.9 (base 10) = ? (base 2)
0.9 * 2 = 1.8 1 .8
0.8 * 2 = 1.6 1 .6
0.6 * 2 = 1.2 1 .2
0.2 * 2 = 0.4 0 .4
0.4 * 2 = 0.8 0 .8
(Stop when repeat begins)

1101 . 11100
-13.9 in binary = 1101.11100 * 2^0
1.10111100 * 2^3
mantissa exp

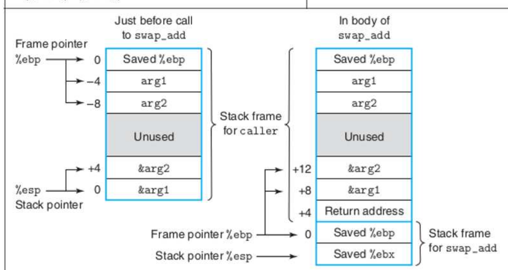


Figure 3.24 **Stack frames for `caller` and `swap_add`.** Procedure `swap_add` retrieves its arguments from the stack frame for `caller`.

Amdahl's Law

$$\text{Increase in speed (performance)} = \frac{1}{\frac{\text{fraction enhanced}}{\text{speedup enhancement}} + (1 - \text{fraction enhanced})}$$

where 'fraction enhanced' is the fractional part of the thing enhanced

Examples

Speedup of 80x with 100 processor cores

$$80 = \frac{1}{(\text{frac parallel} / 100) + (1 - \text{frac parallel})}$$

=> 0.8 * frac parallel + 80(1 - frac parallel) = 1

frac parallel = (80 - 1)/79.2 = 0.9975 99% of the project will have to change to bene

A trucker drives 2500 km averaging 100 km/hr (25 hours)

If for 1500 km of the trip the trucker traveled 150 km/hr, what will be the trucker's speedup for the trip?

enhancement = (150 / 100) = 1.5
fraction enhanced = (1500 km / 2500 km) = 0.6

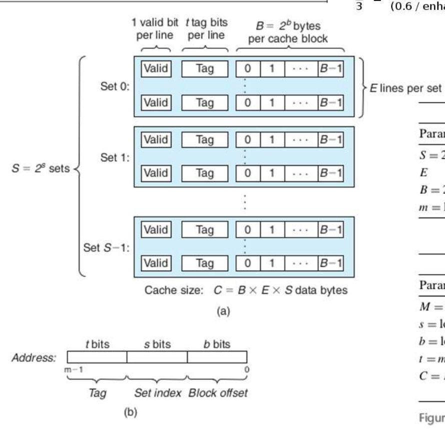
$$\frac{1}{(0.6 / 1.5) + (1 - 0.6)} = 1.25 \text{ times fas}$$

How fast would the trucker have to drive over the portion of 1500 km of the trip to get an overall speedup of 5/3?

$$\frac{5}{3} = \frac{1}{(0.6 / \text{enhancement}) + (1 - 0.6)}$$

enhancement = 3.0 times as fast => 300 km/h

Figure 6.27 **General organization of cache (S, E, B, m).** (a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and b block offset bits.



Fundamental parameters	
Parameter	**Description**
$S = 2^s$	Number of sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical (main memory) address bits

Derived quantities	
Parameter	**Description**
$M = 2^m$	Maximum number of unique memory addresses
$s = \log_2(S)$	Number of *set index bits*
$b = \log_2(B)$	Number of *block offset bits*
$t = m - (s + b)$	Number of *tag bits*
$C = B \times E \times S$	Cache size (bytes) not including overhead such as the valid and tag bits

Figure 6.28 **Summary of cache parameters.**

Instruction		Synonym	Effect	Set condition
`sete`	D	`setz`	D ← ZF	Equal / zero
`setne`	D	`setnz`	D ← ~ZF	Not equal / not zero
`sets`	D		D ← SF	Negative
`setns`	D		D ← ~SF	Nonnegative
`setg`	D	`setnle`	D ← ~(SF ^ OF) & ~ZF	Greater (signed >)
`setge`	D	`setnl`	D ← ~(SF ^ OF)	Greater or equal (signed >=)
`setl`	D	`setnge`	D ← SF ^ OF	Less (signed <)
`setle`	D	`setng`	D ← (SF ^ OF) | ZF	Less or equal (signed <=)
`seta`	D	`setnbe`	D ← ~CF & ~ZF	Above (unsigned >)
`setae`	D	`setnb`	D ← ~CF	Above or equal (unsigned >=)
`setb`	D	`setnae`	D ← CF	Below (unsigned <)
`setbe`	D	`setna`	D ← CF | ZF	Below or equal (unsigned <=)

Figure 3.11 **The SET instructions.** Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," i.e., alternate names for the same machine instruction.

Instruction		Based on	Description
CMP	S_2, S_1	$S_1 - S_2$	Compare
`cmpb`		Compare byte	
`cmpw`		Compare word	
`cmpl`		Compare double word	
TEST	S_2, S_1	S_1 & S_2	Test
`testb`		Test byte	
`testw`		Test word	
`testl`		Test double word	

Figure 3.10 **Comparison and test instructions.** These instructions set the condition codes without updating any other registers.