**Floating Point** For + infinity, the bits of exp are 1, all else is 0, for –infinity the sign bit and all exp are 1, rest are 0. V = (-1)^s x M x 2^E

|  |  |  |  |
| --- | --- | --- | --- |
| CASE | DESC | M | E |
| 1 | Exp is neither all 0’s nor all 1’s | 1 + f (fraction bits unsigned) | e-bias |
| 2 | When exp is all 0’s | f | 1 – bias |
| 3 | Exp is all 1’s | If frac = 0, val is +- infinity, when frac is non 0, value is NaN | |

Bias = 2^(k-1) -1 (k is the # of bits of exp field); e = unsigned number represented by exp field. f goes like ½ + 1/8 + 1/16 …

Smallest denormalized > = 1/(2^(# of fraction bits). Largest normalized > 0 = 1 + (2^f – 1 / 2^5) i.e. 1 + 255/256

loopy:

pushl %ebp

movl $2, %eax

movl %esp, %ebp

movl 8(%ebp), %edx

movl 12(%ebp), %ecx

.L2:

subl $1, %edx

sall $2, %eax

cmpl %ecx, %edx

jb .L2

popl %ebp

ret

unsigned loopy (unsigned x, unsigned y)

{

int result = 2;

do {

result = result\*4;

x = x-1;

} while (x < y);

return result;

}

Type promotion (goes up towards more general)

Long double > double > float > unsigned long > long > unsigned int > int > unsigned short > short > char

Representing numbers:

-1 in signed binary is 11111111… flip the bits & add 1 to convert to and from 2’s comp For n bit 2’s complement representation:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TMin | Tmax | Tmax – Tmin | Tnin – Tmax | -2 is | -TMin |
| -2^n | 2^n -1 | -1 | 1 | -64 + 32 + 16 + 8 + 4 +2 | TMin |

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

|  |  |
| --- | --- |
| Instruction | Does |
| Compl scrc2, src1 | Sets cc to srcs1 – src2 |
| Testl src2, src1 | Sets ccs to src1 & src2 |
| movl Src, Dest | Dest = Src |
| leal Src, Dest | Dest = address of Src |
| jb label | jump below (unsigned) |
| jge label | jump greater or equal (signed) |
| 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 |
| subl %exc, %eax | eax = ecx - eax |

Addressing modes. eax has value 0x200 and mem @ 0x200 is 0x12

|  |  |  |
| --- | --- | --- |
| operand | location | value |
| (%edx,%ecx,8) | M[0x4+0x41\*8]=M[0x20c] |  |
| $0x204 |  | 0x204 (516) |
| %eax |  | 0x200 (512) |
| 0x1f8(,%edx,4) | M[0x1f8+0+0x4\*4]=M[0x208] | 0xd4 (212) |
| (%eax) | M[0x200] | 0x12 (18) |

Struct Alignment

struct s1{

char c1;

int i1, i2;

char c2; };

**sizeof(union u) is the max size of any of its fields**

size of (s1): 16 bytes

Offset c1: 0, c2, 12, i1: 4, i2:8

struct s2{

double d1, d1;

float f1, f2;

short s2;

char c3;

};

Sizeof(s2): 28

Offset: d1:0, d2:8, f1:16, f2:20, s2:24, c3:26

Loop Unrolling.

double inner\_prod (double a1[ ], double a2 [ ], int N) {

int i;

double sum = 0.0;

for (i = 0; i < N; i++) {

sum+= a1[i ] + \* a2 [ i ];

}

return sum; }

double inner\_prod\_unroll (double a1[ ], double a2 [ ], int N) {

int i;

double sum = 0, sum1 = 0;

int limit = N – N%2;

for (i = 0; i < limit ; i+=2) {

sum0 += a1[i ] \* a2 [ i ];

sum1 += a1[i + 1 ] \* a2[i +1];

}

for ( ; i <N; i++)

{

sum0+= a1[i ] \* a2 [ i ];

}

return sum0 + sum1; }

**Unroll by a factor of k:** limit = n – (k-1)

**Speedup:** **s = 1/(1 – a) + a/k**. So if part of a program takes 60% of the running time and we speed this up by a factor of 3. Then k = 3, a = 0.6 so s = 1.667

**Caches:**

* Cache size is B\* E \* S

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| S = 2^s | # of sets |  | m = log2(M) | # of physical addr bits |
| E | # of lines per set |  | s = log2(S) | # of set index bits |
| B = 2^b | Block Size (bytes) |  | b = log2(B) | # of block offset bits |
| M = 2^m | Max # of unique mem addrss |  | t = m – (s + b | # of tag bits |

Memory address of m bits is divided into 3 parts. tag, cache set index and offset. Cache set index has 2 bits and offset has b bits. [t0 t1 t2 t3 t4 s0 s21 o0 o1]

* If asked for the memory addresses that will hit in a set: 1. Get the tags of the valid lines in this set and convert them to binary. 2. You know CT and CI so you might have CT CT CT CT CT CT CT CI CI X X, count to 3 for XX, convert the whole thing(s) from R to L to get addresses in hex that will hit

**A [ i ] [ j ] is at memory xA + L(c \* i + j ) where L = sizeof(int ) and c = # of columns**

**Linking and ELF format:**

extern int buf [];

int \* bufp0 = &buf [0];

static int \*bufp1;

static void inc( )

{ static int count = 0; count ++ ; }

void swap ( )

{ int temp; inc ( ); bufp1 = &buf [1]; temp =

\*bufp0; \*bufp0 = \*bufp1; \*bufp1 = temp; }

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Symbol | swap.o symtab entry? | Symbol type | Module Defined | Section |
| Buf | yes | extern | main.o | .data |
| Bufp0 | yes | global | swap.o | .data |
| Bufp1 | tes | local | swap.o | .bss |
| Swap | yes | global | swap.o | .text |
| Temp | no |  |  |  |
| inc | yes | local | swap.o | .text |
| Count | yes | local | swap.o | .data |

Strongly defined means you have already assigned it, weakly means it just has been declared.

**Processes:**

* if you’re in the child process, the fork will return 0. variables are not shared amongst processes, each process gets its own copy of the stack

|  |  |  |
| --- | --- | --- |
|  | 32-bit | 64 -bit |
| char | 1 | 1 |
| short int | 2 | 2 |
| int | 4 | 4 |
| long int | 4 | 8 |
| char \* | 4 | 8 |
| float | 4 | 4 |
| double | 8 | 8 |

**Pointer Arithemetic:** for short S [ ] and index i let %edx = &S and %ecx = i. put a pointer result in eax, put short result in %ax

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| S+1 | short\* | xS+2 | leal 2(%edx),%eax |
| S[3] | short | \*(xS+6) | movw 6(%edx),%ax |
| &S[i] | short\* | xS+2\*i | leal (%edx,%ecx,2),%eax |
| S[4\*i+1] | short | \*(xS+8\*i+2) | movw 2(%edx,%ecx,8),%ax |
| S+i-5 | short\* | xS+2\*i-10 | leal -10(%edx,%ecx,2),%eax |

**Virtual Memory Exercise. Assume n = 14 (virtual) m = 12 (physical), P = 64 (page size)**

How many physical pages are available? = 2^m / log2(p) = 2^12/2^6 = 2^6 = 64

A. 8 VPN bits and 6 VPO bits (since there are 2^6 physical pages available) P P P P P P P P O O O O O O

B. Physical address P P P P P P O O O O O O (need 6 PPO since VPO is 2^6, fill the rest with PPN)

Given the virtual address 0x03d4, what is:

VPN – just use vpn bits. VPO – just use VPO bits, PPN – use P bits from line B, PPO – use 0 bits from line B. Physical address – convert to binary and just look at the first 12 bits from R to L

**if using TLB** if the TLB is n-way set associative you’ll need log2(n) set bits. the TLB tag bits will be the same amount as you have VPO bits, the rest will be offset bits.

For a 4-way set associative TLB with 16 entries you will have T T T T T T S S O O O O O O

1. First look in the TLB. if you find the appropriate entry, then check the entry in the L1 cache that has the PPN from the TLB as the tag in whichever set your bits indicate 2. if it’s not in the TLB you have a page fault

More Performance: relative performance can be expressed as Told/Tnew.

Suppose you are charged with improving the overall performance of a system by a factor of 2. However, you determine that only 60% of the system can be improved. By what factor k must you improve this part to meet the overall goal?

.4 + x = .5 => x = 0.1

so .6/k = 0.1 => k =6