A Collection of Float Tricks

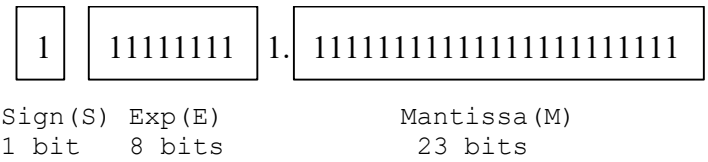
Kasper Nielsen
karmafx@karmafx.net

Abstract

This document is a collection of functions and information for doing fast floating point calculations. Mainly Intel CPU/FPU specific but in many cases apply to other platforms as well. Some of them were made by me, but most have been found on the Internet, in papers, or given to me by generous contributors.

Floating Point Format

Float has three components: sign (S), mantissa (M) and biased exponent (E). For single precision floats the 32 bit word is split as follows:



The value of the number is computed as:

```
(-1)^S * (1 + ( M / (2^P) )) * 2^(E - B)
```

where `P=23` and `B=127` for 32 bit, single precision floats.

Sign (S): Sign bit: 0: positive number, 1: negative number.

Bias (B): The exponent bias: a positive number depending on the precision. B = 127 for 32-bit.

Exponent (E):

All ones: Floating-point number has one of the special values of plus or minus infinity, or "not a number" (NaN). NaN is the result of certain operations, such as division by zero.
All zeros: A *denormalized* floating-point number (=implicit leading bit is zero instead of one). The power of two in this case is the same as the lowest power of two available to a normalized mantissa.
All other: Indicates a normalized floating-point number and the *power of two* (plus bias*)* by which to multiply the normalized mantissa.

Mantissa (P=Number of Mantissa bits):

A positive base-two (non twos-complement) number. If the sign bit is one, the floating-point value is negative, but the mantissa is still interpreted as a positive number that must be multiplied by -1. The mantissa is really 24 bits (53 bits for doubles), where the first most significant (implicit) bit is always 1 in normalized form, and 0 in denormalized form.

Selected Single Precision Floating Point Numbers in Hex (IEEE)

```
  +Inf: 0x7f800000        +Zero: 0x00000000         Nan: 0x7fc00000
  -Inf: 0xff800000        -Zero: 0x80000000          PI: 0x40490fdb
     1: 0x3f800000        0.25: 0x3e800000            2: 0x40000000
```

Fast Float Conversion Tricks

Float to Integer without Rounding

```
inline long int lrintf (float flt) {
     int intgr;
     _asm {
            fld flt
            fistp intgr
     }
     return intgr;
}
```

This is faster than a normal: `(int)myfloat`, which may take up to 80 cycles due to rounding mode changes. But this is not accurate: Result depends on rounding mode (see "Fast and Exact Rounding Methods" below).

Float to Integer with Rounding

```
inline int fast_float_to_int(float flt) {
     int i;
     static const double half = 0.5f;
     _asm {
           fld flt
           fsub half
           fistp i
     }
     return i;
}
```

This is faster than a normal: `(int)myfloat`, which may take up to 80 cycles due to rounding mode changes. But this is not accurate: Result depends on rounding mode (see "Fast and Exact Rounding Methods" below).

Float to Integer (posted by Terje Mathisen on comp.lang.asm.x86)

```
inline long float2long(float d) {
   const double MAGIC = (((65536.0*65536.0*16.0)+32768.0)*65536.0);
   double dtemp = d + MAGIC;
   return (*(long *) &dtemp) - 0x80000000;
}
```

Float to Integer

```
inline int float2int(float d) {
       const double MAGIC = 2251799813685248+4503599627370496; // 2^51+2^52
       double dtemp = d + MAGIC;
       return (*(int*)&dtemp);
}
```

Based on code by [Karvonen98]: Adding 2^51+2^52 has the effect that any integer between -2^31 and +2^31 will be aligned in the lowest 32 bit of a double. Results are the same as using a regular fistp instruction. It is not necessarily faster but has better scheduling opportunities.

Big Endian Compatible Float to Integer Conversion [Herf00]

```
#define DOUBLE2FIXMAGIC 68719476736.0 * 1.5 // 2^36 * 1.5
#define SHITAMT 16  // 16.16 fixed point representation.
inline int fast_ftoi(float val)  {
     double dval = (double)val + DOUBLE2FIXMAGIC;
     //return ((int *)&dval)[1] >> SHITAMT;  // Big Endian
     return ((int *)&dval)[0] >> SHITAMT; // Little Endian
}
```

Exact Rounding Methods [DeSoras04]

Assumes the default processor rounding mode (round to nearest integer, *even* mode: i.e., 2.5 => 2, 3.5 => 4: half the time round up, other half down). The [fadd st, st (0) / sar i,1] trick fixes the "round to nearest even number" behaviour. Thus, round_int (N+0.5) always returns N+1 and floor_int function is appropriate to convert floating point numbers into fixed point numbers. But be careful, this trick can overflow.

```
int floor_int (double x) {
        int i;
        static const float round_toward_m_i = -0.5f;
        __asm {
                fld      x
                fadd     st, st (0)
                fadd     round_toward_m_i
                fistp    i
                sar      i, 1
        }
        return (i);
}

int round_int (double x)  {
    assert (x >static_cast <double>(INT_MIN /2)- 1.0);
    assert (x <static_cast <double>(INT_MAX /2)+1.0);
    const float round_to_nearest =0.5f;
    int i;
    __asm  {
       fld x
       fadd st,st (0)
       fadd round_to_nearest
       fistp i
       sar i,1
    }
    return (i);
}

int ceil_int (double x)  {
    assert (x >static_cast <double>(INT_MIN /2)- 1.0);
    assert (x <static_cast <double>(INT_MAX /2)+1.0);
    const float round_towards_m_i =-0.5f;
    int i;
    __asm     {
       fld x
       fadd st,st (0)
       fsubr round_towards_m_i
       fistp i
       sar i,1
    }
    return (-i);
}

int truncate_int (double x)  {
    assert (x >static_cast <double>(INT_MIN /2)- 1.0);
    assert (x <static_cast <double>(INT_MAX /2)+1.0);
    const float round_towards_m_i =-0.5f;
    int i;
    __asm  {
       fld x
       fadd st,st (0)
       fabs st (0)
       fadd round_towards_m_i
       fistp i
       sar i,1
    }
    if (x <0) i =-i;
    return (i);
}
```

Float Rounding, Using Only Integer Operations

```
inline int intChop (const float& f)
{
    INT32 a         = *reinterpret_cast<const INT32*>(&f); // take bit pattern of float into a reg
    INT32 sign      = (a>>31);   // sign = 0xFFFFFFFF if original value is negative, 0 if positive
    INT32 mantissa  = (a&((1<<23)-1))|(1<<23); // extract mantissa and add the hidden bit
    INT32 exponent  = ((a&0x7fffffff)>>23)-127; // extract the exponent
    INT32 r         = ((UINT32)(mantissa)<<8)>>(31-exponent); // ((1<<exponent)*mantissa)>>24
    return ((r ^ (sign)) - sign ) &~ (exponent>>31); // add original sign. If exponent was negative, return 0.
}

inline int intFloor (const float& f)
{
    INT32 a         = *reinterpret_cast<const INT32*>(&f); // take bit pattern of float into a register
    INT32 sign      = (a>>31);     // sign = 0xFFFFFFFF if original value is negative, 0 if positive
    INT32 exponent  = ((a&0x7fffffff)>>23)-127; // extract the exponent
    INT32 expsign     = ~(exponent>>31);    // 0xFFFFFFFF if exponent is positive, 0 otherwise
    INT32 imask     = ( (1<<(31-(exponent&expsign))))-1;  // mask for true integer values
    INT32 mantissa  = (a&((1<<23)-1)); // extract mantissa (without the hidden bit)
    INT32 r         = ((UINT32)(mantissa|(1<<23))<<8)>>(31-exponent);// ((1<<exponent)*(mantissa|hidden bit))>>24
    r = ((r & expsign) ^ (sign)) + ((!((mantissa<<8)&imask))&sign); // if (fabs(value)<1.0) value = 0; copy sign;
                                                                    //    if (value < 0 && value==(int)(value)) value++;
    return r;
}

inline int intCeil (const float& f)
{
    INT32 a         = *reinterpret_cast<const INT32*>(&f);  // take bit pattern of float into a register
    a ^= 0x80000000;                     // change floating point value to negative
    INT32 sign      = (a>>31);    // sign = 0xFFFFFFFF if original value is negative, 0 if positive
    INT32 exponent  = ((a&0x7fffffff)>>23)-127;    // extract the exponent
    INT32 expsign     = ~(exponent>>31);    // 0xFFFFFFFF if exponent is positive, 0 otherwise
    INT32 imask     = ( (1<<(31-(exponent&expsign))))-1;  // mask for true integer values
    INT32 mantissa  = (a&((1<<23)-1));     // extract mantissa (without the hidden bit)
    INT32 r         = ((UINT32)(mantissa|(1<<23))<<8)>>(31-exponent); // ((1<<exponent)*(mantissa|hidden bit))>>24
    // if (fabs(value)<1.0) value = 0; copy sign; if (value < 0 && value==(int)(value)) value++;
     r = ((r & expsign) ^ (sign)) + ((!((mantissa<<8)&imask))&sign&expsign);
     return -r;  // negate value (as we negated it originally)
}
```

SSE Float to Int Conversion

When using SSE/SSE2 one instruction can covert four floats to four ints in one go:

```
cvtps2pi mm1, xmm0; //4 floats packed in XMM0 to four ints in MMX register 1
```

Float32 Integer and Fractional part (without storage) (by Nervus)

On machines using 32bit floating point internally you can get the integer part of a float by adding a very big float (the highest possible integer, `0x4b400000`) and subtract it again.

Note, this will only work on a Pentium when using SSE, since the FPU has more than 32bits of precision.

To make it work on Pentium FPU the value used should be `0x59c00000` (FP representation of 2^51 + 2^52) [Karvonen98].

Fixed Point Arithmetic, 32 bit

```
inline long int2fixed(short x) { return(x<<16); }

inline short fixed2int(long x) { return(short(x>>16)); }

inline float fixed2float(long x) { return(float(x)/65536.0f); }

// float to 16:16 fixed point (adapted from a method posted by Chris Babcock)
inline long float2fixed(float d)  {
    const double MAGIC = 103079215104.00;
    double dtemp = d + MAGIC;
    return (*(long *) &dtemp);
}

//inline long float2fixed(float x) { return(floor(x*65536+0.5)); }

inline long fixedmul(long m1, long m2)  {
    long r;
    __asm  {
          mov eax, m1
          mov edx, m2
          imul edx
          shrd eax, edx, 16
          mov r, eax
    }
    return r;
}

inline long fixeddiv(long divisor, long dividend) {
    long r;
    __asm  {
          mov ecx, dividend
          mov edx, divisor
          xor eax, eax
          shrd eax, edx, 16 ;make 32-bit value in DX:AX
          sar edx, 16
          idiv ecx ;returns EAX ie 16:16 divided figure
          mov r, eax
    }
    return r;
}
```

Fixed Point Arithmetic, 64 bit

```
typedef union {
      unsigned int dword[2];    // 0 is low part, 1 is high part
      __int64  qword;
} fp64; // fixedpoint 64


__forceinline fp64 fp64add(fp64 target, const fp64 source)
{
      __asm
      {
            mov eax, dword ptr [source.dword+0]
            mov edx, dword ptr [source.dword+4]
            add dword ptr [target.dword+0], eax
            adc dword ptr [target.dword+4], edx
      }
      return target;
}


__forceinline void fp64set(fp64& target, const int high, const unsigned int low)
{
      target.dword[0]=low;
      target.dword[1]=high;
}


// floating point to 64 bit fixed point (by khn)
__forceinline void intToFP64 (fp64 &r, const float& f) {
#if 0
      #pragma message ("*** using unoptimized inttofp64")
      r.qword = __int64(f*4294967296);
#else
      const int a = *(const int*)(&f);
      const int sign = (a>>31); // 0xffffffff or 0
      // 1 is implicit leading 1 (normalized float):
      const int mantissa = (a&((1<<23)-1))|(1<<23);
      const int exponent = ((a&0x7fffffff)>>23)-127;

      int r1 = ((unsigned int)(mantissa)<<8)>>(31-exponent);
      r1 &= ~(exponent>>31);  //if (exponent<0) r1=0;
      r1 ^= sign;

      const int lowexp = -exponent-1; //31-exponent-32;
      unsigned int r2 = ((unsigned int)(mantissa)<<8);
      if (lowexp>=0) r2 >>= lowexp; else r2 <<= -lowexp;

      r2 = ((r2 ^ sign) - sign );
      if (!r2) r1 -=sign;

      r.dword[0] = r2;
      r.dword[1] = r1;

#endif

}
```

Float 32 to float 16 conversion (by Sparky)

```
WORD float32_to_float16(float f) {
   // convert 32 bit float: 1 sign bit, 8 exp bits, 23 mantissa bits
   // to -> 16 bit float: 1 sign bit, 5 exp bits, 10 mantissa bits:
   // exponent = exponent - 127 + 15...=> range 0..31 .
   // mantissa is simply clamped from 23 to 10 bits.
   // sign bit is transferred.
   // handles 0 and -0 as special case.
   // clamps too small values to lowest possible value.

   unsigned int fi = *((unsigned int*)&f);
   unsigned int fo;
   if (fi==0x80000000 || fi==0x00000000) {
         fo = 0;
   } else {
         int e = (fi&0x7f800000) >> 23;
         int m = (fi&0x7fffff);
         e = e - 127 + 15;
         if (e<0) {
               e = 0;
               m = 1<<13;
         }

         ASSERT(e<32);

         m = m>>13;
         fo = ((e << 10) | m) | ((fi&0x80000000)>>16);

   }
   return (WORD)fo;
}
```

Fast Float Math Functions

Absolute

```
__forceinline float fast_abs(float a) {
    unsigned int f = (*(unsigned int*)&(a))&0x7FFFFFFF;
    return (*(float*)&(f));
}
```

Negate

```
__forceinline float fast_neg(float f) {
   int temp = (*((int*)&f))^0x80000000;
   return *((float*)&temp);
}
```

Sign

```
__forceinline int fast_sgn(float f) {
  return 1+((*((int*)&f)>>31)<<1);
}
```

Power

fastpow(f,n) gives a rather *rough* estimate of a float number f to the power of an integer number n (y=f^n). It is fast but result can be quite a bit off, since we directly mess with the floating point exponent.-> use it only for getting rough estimates of the values and where precision is not that important [MusicDSP].

```
float fastpower(float f,int n) {
    long *lp,l;
    lp=(long*)(&f);
    l=*lp;l-=0x3F800000l;l<<=(n-1);l+=0x3F800000l;*lp=l;
    return f;
}
```

Root

fastroot(f,n) gives the n-th root of f. Same thing concerning precision applies here [MusicDSP].

```
float fastroot(float f,int n) {
    long *lp,l;
    lp=(long*)(&f);
    l=*lp;l-=0x3F800000l;l>>=(n-1);l+=0x3F800000l;
    *lp=l;
    return f;
}
```

Reciprocal Square Root Approximation (various sources, see [Blinn97] and [Lomont03])

```
float InvSqrt(float x)  {
    float xhalf = 0.5f*x;
    int i = *(int*)&x; // get bits for floating value
    i = 0x5f3759df - (i>>1); // gives initial guess y0
    x = *(float*)&i; // convert bits back to float
    x = x*(1.5f-xhalf*x*x); // Newton step, repeating increases accuracy
    return x;
}
```

Alternative Reciprocal Square Root Approximation [Rocatis04]

```
 float FastRSQRT(float val) {
     const float magicValue = 1597358720.0f;
     float tmp = (float)*((uint*)&val);
     tmp = (tmp * -0.5f) + magicValue;
     uint tmp2 = (uint)tmp;
     return *(float*)&tmp2;
 }
```

Note: This is less precise than the one above since it skips the Newton step (See [Intel803-99]).

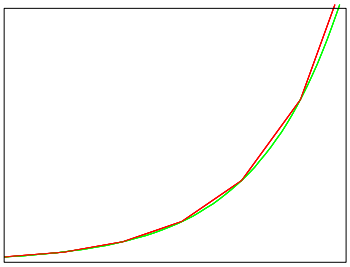
Same Principle used for General Power / Root Approximations (adapted from newsgroup post):

```
float PrecomputeMagicValue(float fPower, float fScale) // power and final scale
   const float oneRepresentationAsFloat = float(0x3f800000);
   float magicValue = float( (*((unsigned int*)&fScale))) -
(oneRepresentationAsFloat * fPower);
   return magicValue;
}
```

e.g.:

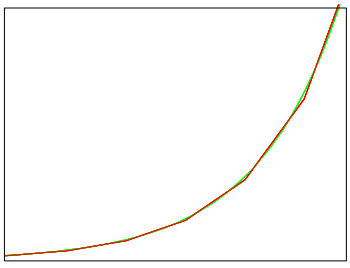
```
float FastCubeRoot(float val) {
   const float fPower = 0.25f;
   float magicValue = PrecomputeMagicValue(fPower, 1.0f);
   float tmp = (float)*((unsigned int*)&val);
   tmp = (tmp * fPower) + magicValue;
   unsigned int tmp2 = (unsigned int)tmp;
   return *(float*)&tmp2;
}
```

Exp (Nice, piecewise linear approximation, adapted from code by Nvidia)



```
__forceinline float FP_EXP ( float p ) {
    int _i;
    float e = 1.44269504f * (float)0x00800000 * (p);
    _i = (int)e + 0x3F800000;
    e = *(float *)&_i;
    return e;
}
```

Big Endian Compatible Exp [Schraudolph98]

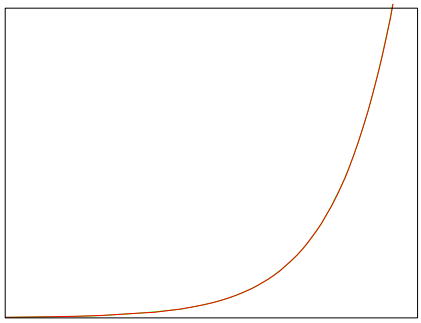


```
static union {
    double d;
    struct {
        int j, i; // use int i, j for big endian
   } n;
} _eco;
#define M_LN2 0.693147180559945309417232121458 18
double fast_exp(double val) {
    const double a = 1048576/M_LN2;
    const double b_c = 1072632447;
    _eco.n.i = (int)(a*val + b_c);
    return _eco.d;
}
```

Pow2 (c) Ian Stephenson [Stephenson] (This is quite accurate compared to the previous ones)

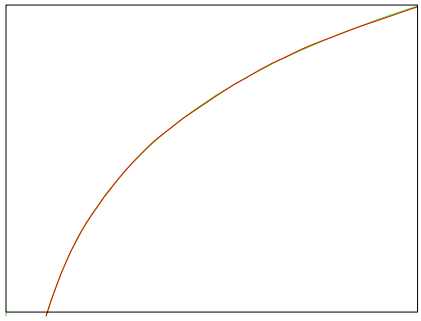
In order to increase accuracy Ian calculates a rough result using the cast-float-to-int-trick, and then approximates the error with a quadratic. This places the error within 1% for most sensible (>0) values.

```
float _fast_pow2(const float val)
{
    float result;
    float mp = 0.33971f;
    float tmp = val - floorf(val);
    tmp = (tmp - tmp*tmp) * mp;
    result = val + 127 - tmp;
    result *= (1<<23);
    *(int*)&result = (int)result;
    return result;
}
```



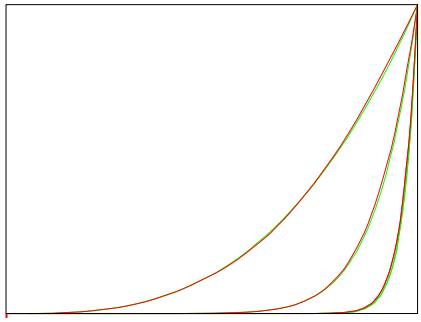
Log2 (c) Ian Stephenson [Stephenson] (This is quite accurate compared to the previous ones)

```
float _fast_log2(const float val)
{
    float result, tmp;
    float mp = 0.346607f;
    result = (float)*((int*)&val);
    result *= 1.0/(1<<23);
    result = result - 127;
    tmp = result - floorf(result);
    tmp = (tmp - tmp*tmp) * mp;
    return tmp + result;
}
```



Pow (Adapted from `[Stephenson]`)

```
inline float fast_pow(float v1, float v2)
{
    return _fast_pow2(v2 * _fast_log2(v1));
}
```



Log

```
inline float fast_log (const float &v)
{
   return (fast_log2 (v)* 0.69314718f);
}
```

Log10

```
inline float fast_log10 (const float &v)
{
   return (fast_log2(v)/3.321928095f);
}
```

Exact Pow (Faster than regular powf since it avoids fscale. Based on code by Agner Fog [Fog06])

```
__forceinline float FPOW(float a,float b)
{
  float result;
  __asm
  {
    fld   dword ptr [b];
    fld   dword ptr [a];
    ftst;
    fstsw ax;
    sahf;
    jz    zero;
    fyl2x;
    fist  dword ptr [a];
    sub   esp, 12;
    mov   dword ptr [esp],0;
    mov   dword ptr [esp+4],0x80000000;
    fisub dword ptr [a];
    mov   eax, dword ptr [a];
    add   eax, 0x3fff;
    mov   [esp+8], eax;
    jle   underflow;
    cmp   eax, 0x8000;
    jge   overflow;
    f2xm1;
    fld1;
    fadd;
    fld   tbyte ptr [esp];
    add   esp, 12;
    fmul;
    jmp   end;
underflow:
    fstp  st;
    fldz;
    add   esp, 12;
    jmp   end;
overflow:
    push  0x7f800000;
    fstp  st;
    fld   dword ptr [esp];
    add   esp, 16;
    jmp   end;
zero:
    fstp  st(1);
end:
    fstp dword ptr [result]
  }
  return result;
}
```

Inverse, Pow2, Log2, Sqrt and InvSqrt [Blinn97]

This is the code from Blinn's original article on float tricks. They all use the same IEEE float-as-int trick to make piecewise linear approximations to well-known math functions.

```
// Blinn's code !

inline long int AsInteger (float f)  { return *(long int*)&f; }
inline float    AsFloat (long int i) { return *(float*)&i; }

const long int  OneAsInteger = AsInteger(1.0f); // 0x3F800000
const float     ScaleUp      = float(0x00800000);
const float     ScaleDwn     = 1.f/ScaleUp;

// first approximation (relative worst-case errors is about 10 percent)
inline float Alog2(float x)
      { return float (AsInteger(x)-OneAsInteger)*ScaleDwn; }
inline float Apow2(float x)
      { return AsFloat(int(x*ScaleUp)+OneAsInteger); }
inline float Asqrt(float x)
      { int i=(AsInteger(x)>>1)+(OneAsInteger>>1); return AsFloat(i); }
inline float Ainverse(float x)
      { int i=-AsInteger(x)+2*OneAsInteger; return AsFloat(i); }
inline float AinverseSqrt(float f)
      { int i=(OneAsInteger + (OneAsInteger>>1))-(AsInteger(f)>>1);
        return AsFloat(i); }

// second approximation, uses first as seed for 1 Newton-Raphson iteration).
inline float Binverse(float x)
      { float y = Ainverse(x); return y*(2-x*y); }
inline float Bsqrt(float x)
      { float y = Asqrt(x); return (y*y+x)/(2*y); }
inline float BinverseSqrt(float x)
      { float y = AinverseSqrt(x); return y*(1.5f-.5f*x*y*y); }
inline float Bpow2(float x)
      { return Apow2(x/2)*Apow2(x/2+.5f)*.657f; }
inline float Blog2(float x)
      { return .5f*(Alog2(x)+Alog2(x*0.6666f))+.344f; }
```

Exact Exp (Slightly faster than expf. Warning: No over/underflow test!)

```
__forceinline float FEXP(float f)         {
  __asm
  {
    fld   dword ptr [f];
    fldl2e;
    fmulp st(1), st;
    fld1;
    fld   st(1);
    fprem;
    f2xm1;
    faddp st(1), st;
    fscale;
    fstp  st(1);
    fstp  dword ptr [f];
  }
  return f;
}
```

Log2 (posted by Laurent de Soras on flipcode)

```
inline float fast_log2 (float val) {
  int * const    exp_ptr = reinterpret_cast <int *> (&val);
  int x = *exp_ptr;
  const int log_2 = ((x >> 23) & 255) - 128;
  x &= ~(255 << 23);
  x += 127 << 23;
  *exp_ptr = x;
  val = ((-1.0f/3) * val + 2) * val - 2.0f/3;   // (1)
  return (val + log_2);
}
```

Another Log2 [Vellocet03]

```
// calculate log2(x) by exploiting the identity:
// f2x = 2^floor(log2(x))
// log2(x) = log2(f2x) + log2(x/f2x)
// pre: x>=1e-38 x<=1e38
// error [-0.00075, 0.0012]
inline float fastlog2f(float x) {
  long *p = (long*)(&x);  // store address of float as long pointer
  const long e = ((*p)>>23) - 127;     // extract the exponent
  *p &= 0x007fffff;                    // mask away the exponent
  *p |= 0x3f800000;                    // set exponent to 127 (0)
  x  -= 1.0f;
  return (float)(e) +  // cubic approximation of log2(x) in range [1, 2]
        x*(1.4201157697141027f + x*(-0.5747927782450741f +
        x*(0.15468105905881002f)));
}
```

Ln [Vellocet03]

```
inline float fastlnf(float x) {
  long *p = (long*)(&x); // store address of float as long pointer
  const long e = ((*p)>>23) - 127;     // extract the exponent
  *p &= 0x007fffff;                    // mask away the exponent
  *p |= 0x3f800000;                    // set exponent to 127 (0)
  x  -= 1.0f;
  // cubic approximation of log2(x)/log2 (e) in range [1, 2]:
  return 0.6931471805599453f*( (float)(e) + x*(1.4201157697141027f +
        x*(-0.5747927782450741f + x*(0.15468105905881002f))));
}
```

Pow2 [Vellocet03]

```
inline float fastpow2(float x) {
  long *px = (long*)(&x); // store address of float as long pointer
  const float tx = (x-0.5f) + (3<<22); // temporary value for truncation
  const long  lx = *((long*)&tx) - 0x4b400000; // integer power of 2
  const float dx = x-(float)(lx); // float remainder of power of 2
  x = 1.0f + dx*(0.6960656421638072f + // cubic approximation of 2^x
             dx*(0.224494337302845f +  // for x in the range [0, 1]
             dx*(0.07944023841053369f)));
  *px += (lx<<23); // add integer power of 2 to exponent
  return x;
}
```

Approximate Log (Robin Green on gmane.games.devel.algorithms)

```
float approxlog(float log)
{
   long& nat = ((*(long *)&log));
   float exp2 = ((nat & 0x7f800000) >> 23) - 127;

   // interpolate the mantissa using a minimax polynomial
   float mantissa = float(nat & 0x007fffff) * inverseMantissa;
   float mantissa2 = mantissa * mantissa;
   float mantissa3 = mantissa2 * mantissa;
   float mantissa4 = mantissa2 * mantissa2;
   float mantissa5 = mantissa3 * mantissa2;

   float loge_mantissa = mantissa -
    0.495858579 * (mantissa2) +
    0.236771441  * mantissa3 +
    0.0946189007 * mantissa4 -
    -0.172685158 * mantissa5;

#ifdef _DEBUG
   printf ("exp:%f\napx:%f\napm:%f\n\n", logf(log), exp2 * log2_e, exp2 *
log2_e + loge_mantissa);
#endif

   return exp2 * log2_e + loge_mantissa;
};
```

Another approximate Pow2 (posted by Johannes M.R. on musicdsp.org)

Piecewise quadratic approximate exponential function, that assumes round-to-zero mode, and ieee 754.

```
inline float fpow2(const float y)
{
  union
  {
      float f;
      int i;
  } c;

  int integer = (int)y;
  if(y < 0) integer = integer-1;

  float frac = y - (float)integer;

  c.i = (integer+127) << 23;
  c.f *= 0.33977f*frac*frac + (1.0f-0.33977f)*frac + 1.0f;

  return c.f;
}
```

Simple Pow & Log Approximations [Schlick94]

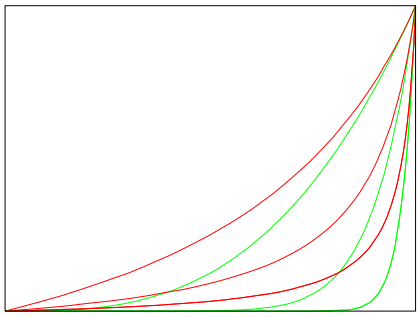
An approximate but useful Pow-like function is:

$$pow(x,n) \approx \frac{x}{n - nx + x}$$

or:

```
float y = (x/(n-n*x+x));
```

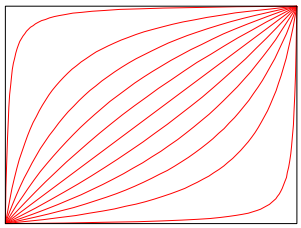
where *x* is in the interval [0:1].



It can be made to handle Log-like functions too, by remapping the power:

```
a = (1-1/t);
y = x/(x+a*x-a);
```

where *t* controls the slope of the function.



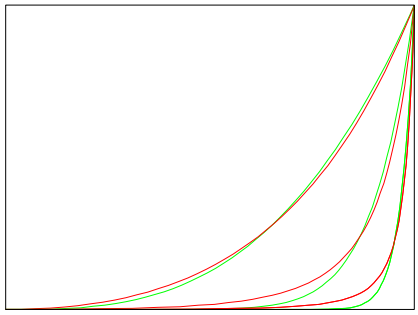
Improved Pow & Log Approximations [posted by Sparky on the playstation2-linux.com forum]

A better Pow approximation is:

$$pow(x,n) \approx \frac{x^2}{(kx-(k+1))^2}, \quad \text{where } k = \frac{\frac{3}{4}n^2}{n+1} - 1$$

where *x* is in the interval [0:1], *n* should be >=2. In code:

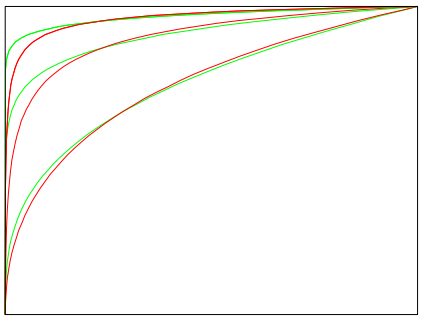
```
k = (0.75*n*n)/(n+1) - 1; // should be precomputed
float temp = (k*x-(k+1));
y = (x*x)/(temp*temp);
```



The derived Pow(x,1/n) approximation is:

$$pow(x,1/n) \approx r - \frac{r}{k\sqrt{x}+1}, \quad \text{where } r = \frac{k+1}{k}$$

```
k = (0.75*n*n)/(n+1) - 1; // should be precomputed
float val = (k+1.0)/k;    // should be precomputed
y= val - val/(sqrt(x)*k+1);
```



Slightly better results (for large n, worse for small n) can be achieved by computing the constant *k* using a taylor expansion:

```
const double d0 = -1.592174325;
const double d1 = 0.7287670208;
const double d2 = 0.8637189877e-3;
const double d3 = -0.1030067486e-4;
const double d4 = 0.5760166761e-7;
const double d5 = -0.1213914585e-9;
double k = d0+n*(d1+n*(d2+n*(d3+n*(d4+d5*n))));
```

Square Root [KVR DSP forum]

Maximum error is reached in x=2^n, with n integer and odd. Correct results are in x=2^n, with n even.

```
inline float fsqrt(float x) {
  float y;
  _asm {
    mov ecx, x
    and ecx, 0x7f800000
    fld x
    mov eax, ecx
    add ecx, 0x3f800000
    or  ecx, 0x00800000
    shr ecx, 1
    mov y, eax
    and ecx, 0x3f800000
    fld y
    fadd
    fstp y
    and [y], 0x007fffff
    or  [y], ecx
  }
  return y;
}
```

Square Root [Hsieh96]

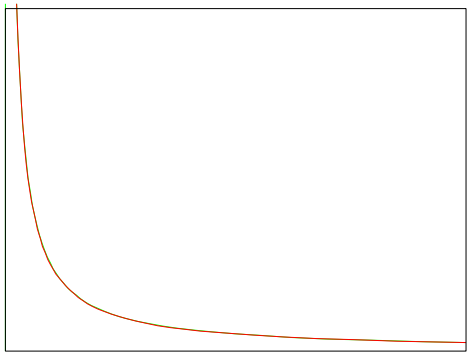
```
double fsqrt(double r) {
    double x,y;
    double tempf;
    unsigned long *tfptr = ((unsigned long *)&tempf)+1;
    tempf = r;
    *tfptr=(0xbfcd4600-*tfptr)>>1;
    x=tempf;
    y=r*0.5;
    x*=1.5-x*x*y;
    x*=1.5-x*x*y;
    x*=1.5-x*x*y;
    x*=1.5-x*x*y;
    return x*r;
}
```

SSE Square Root [Posted by Borogrove on KVR DSP Forum 2005]

22-bit accuracy in about 20 cycles total.

```
float sse_approx_sqrt( float x ) {
   float z;
   static float half = 0.5f;
   _asm  {
      movss   xmm1, x         ;// x1: x
      rsqrtss xmm2, x         ;// x2: ~1/sqrt(x) = 1/z
      rcpss   xmm0, xmm2      ;// x0: z == ~sqrt(x) to 0.05%
      movss   xmm4, xmm0      ;// x4: z
      movss   xmm3, half      ;// x3: 0.5
      mulss   xmm4, xmm4      ;// x4: z*z
      mulss   xmm2, xmm3      ;// x2: 1 / 2z
      subss   xmm4, xmm1      ;// x4: z*z-x
      mulss   xmm4, xmm2      ;// x4: (z*z-x)/2z
      subss   xmm0, xmm4      ;// x0: z' to 0.000015%
      movss   z, xmm0         ;//
   }
   return z;
}
```

Reciprocal, r = 1/p (Simon Hughes / codeproject.com)



This approximation is ~2.12 times faster than using 1.0f / n.

```
__forceinline float FP_INV(float p)
{
      int _i = 2 * 0x3F800000 - *(int *)&(p);
      float r = *(float *)&_i;
      r = r * (2.0f - (p) * (r));
      return r;
}
```

4 x Division, r = dividend/divisor [Fog06]

```
x0 = rcpss(d);
x1 = x0 * (2 - d * x0) = 2 * x0 - d * x0 * x0;
```

where x0 is the first approximation to the reciprocal of the divisor d, and x1 is a better approximation. You must use this formula before multiplying with the dividend.

```
movaps xmm1, [divisors] ; load divisors
rcpps xmm0, xmm1 ; approximate reciprocal
mulps xmm1, xmm0 ; newton-raphson formula
mulps xmm1, xmm0
addps xmm0, xmm0
subps xmm0, xmm1
mulps xmm0, [dividends] ; results in xmm0
```

This makes four divisions in approximately 23 clock cycles on a Pentium M with a precision of 23 bits. Increasing the precision further by repeating the Newton-Raphson formula [Intel803-99] with double precision is possible, but not very advantageous [Fog06].

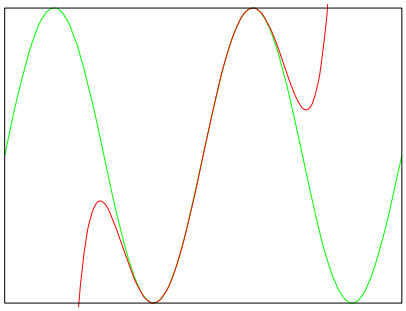
SSE Reciprocal, r = 1/p (Posted by kaleja@estarcion.com on musicdsp.org)

```
// ~12 clocks on Pentium M vs. ~16 for single precision divide
__forceinline float reciprocal_sse_22bits( float x ) {
    float z;
    _asm   {
        rcpss    xmm0, x          // x0: z ~= 1/x
        movss    xmm2, x          // x2: x
        movss    xmm1, xmm0       // x1: z ~= 1/x
        addss    xmm0, xmm0       // x0: 2z
        mulss    xmm1, xmm1       // x1: z^2
        mulss    xmm1, xmm2       // x1: xz^2
        subss    xmm0, xmm1       // x0: z' ~= 1/x to 0.000012%
        movss    z, xmm0
    }
    return z;
}
```

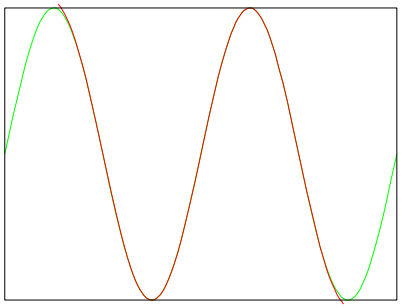
Cos, Sin, Tan and Inv Sin, etc. Approximations [MusicDSP]

Surprisingly accurate and very usable (from http://www.wild-magic.com/SourceCode.html)
Sin0 is faster but less accurate than Sin1, same for the other pairs.
The domains are: Sin/Cos [0, pi/2], Tan [0,pi/4], InvSin/Cos [0, 1], InvTan [-1, 1]

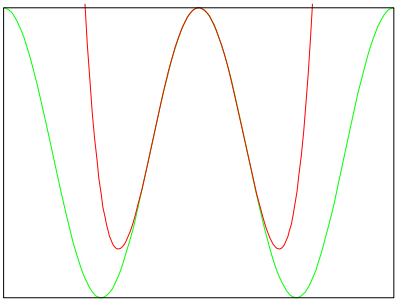
```
float FastSin0 (float fAngle)
{
    float fASqr = fAngle*fAngle;
    float fResult = 7.61e-03f;
    fResult *= fASqr;
    fResult -= 1.6605e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    fResult *= fAngle;
    return fResult;
}
```



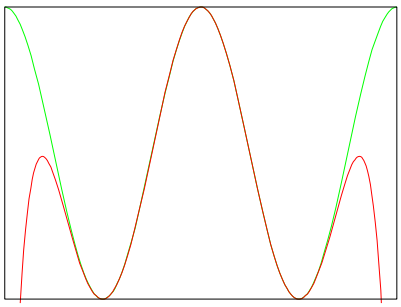
```
float FastSin1 (float fAngle)
{
    float fASqr = fAngle*fAngle;
    float fResult = -2.39e-08f;
    fResult *= fASqr;
    fResult += 2.7526e-06f;
    fResult *= fASqr;
    fResult -= 1.98409e-04f;
    fResult *= fASqr;
    fResult += 8.3333315e-03f;
    fResult *= fASqr;
    fResult -= 1.666666664e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    fResult *= fAngle;
    return fResult;
}
```



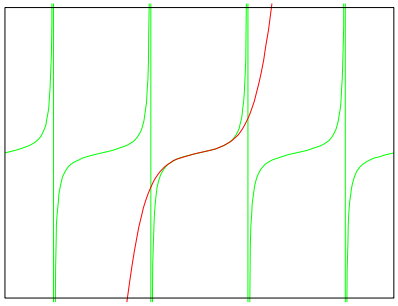
```
float FastCos0 (float fAngle)
{
    float fASqr = fAngle*fAngle;
    float fResult = 3.705e-02f;
    fResult *= fASqr;
    fResult -= 4.967e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    return fResult;
}
```



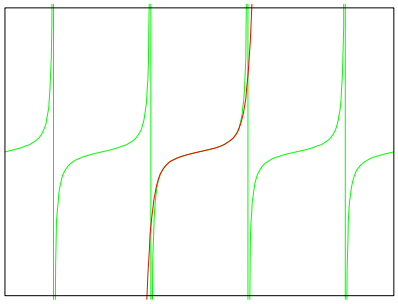
```
float FastCos1 (float fAngle)
{
    float fASqr = fAngle*fAngle;
    float fResult = -2.605e-07f;
    fResult *= fASqr;
    fResult += 2.47609e-05f;
    fResult *= fASqr;
    fResult -= 1.3888397e-03f;
    fResult *= fASqr;
    fResult += 4.16666418e-02f;
    fResult *= fASqr;
    fResult -= 4.999999963e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    return fResult;
}
```



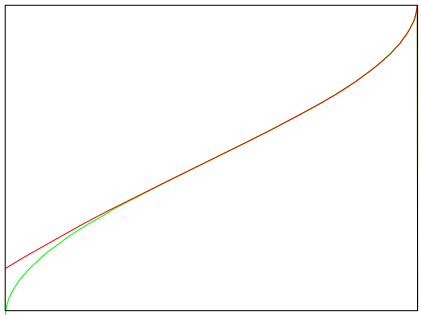
```
float FastTan0 (float fAngle)
{
    float fASqr = fAngle*fAngle;
    float fResult = 2.033e-01f;
    fResult *= fASqr;
    fResult += 3.1755e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    fResult *= fAngle;
    return fResult;
}
```



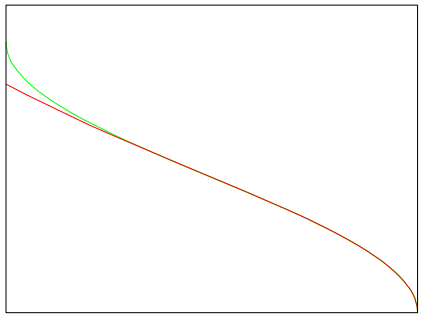
```
float FastTan1 (float fAngle)
{
    float fASqr = fAngle*fAngle;
    float fResult = 9.5168091e-03f;
    fResult *= fASqr;
    fResult += 2.900525e-03f;
    fResult *= fASqr;
    fResult += 2.45650893e-02f;
    fResult *= fASqr;
    fResult += 5.33740603e-02f;
    fResult *= fASqr;
    fResult += 1.333923995e-01f;
    fResult *= fASqr;
    fResult += 3.333314036e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    fResult *= fAngle;
    return fResult;
}
```



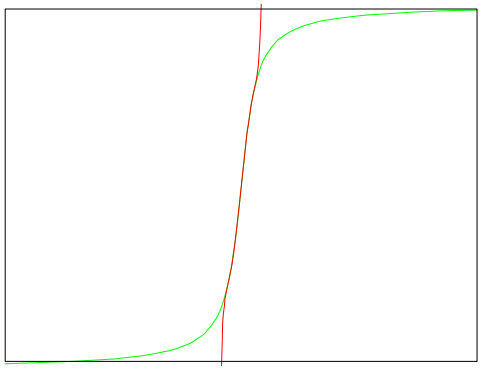
```
float FastInvSin (float fValue)
{
    float fRoot = sqrtf(1.0f-fValue);
    float fResult = -0.0187293f;
    fResult *= fValue;
    fResult += 0.0742610f;
    fResult *= fValue;
    fResult -= 0.2121144f;
    fResult *= fValue;
    fResult += 1.5707288f;
    fResult = HALF_PI - fRoot*fResult;
    return fResult;
}
```



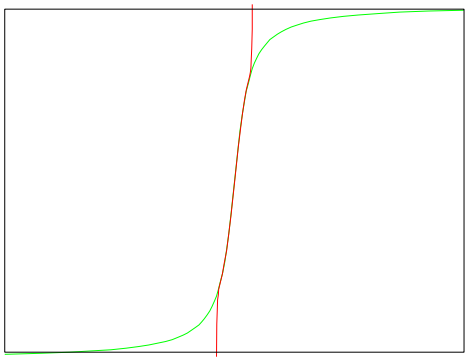
```
float FastInvCos (float fValue)
{
    float fRoot = sqrtf(1.0f-fValue);
    float fResult = -0.0187293f;
    fResult *= fValue;
    fResult += 0.0742610f;
    fResult *= fValue;
    fResult -= 0.2121144f;
    fResult *= fValue;
    fResult += 1.5707288f;
    fResult *= fRoot;
    return fResult;
}
```



```
float FastInvTan0 (float fValue)
{
    float fVSqr = fValue*fValue;
    float fResult = 0.0208351f;
    fResult *= fVSqr;
    fResult -= 0.085133f;
    fResult *= fVSqr;
    fResult += 0.180141f;
    fResult *= fVSqr;
    fResult -= 0.3302995f;
    fResult *= fVSqr;
    fResult += 0.999866f;
    fResult *= fValue;
    return fResult;
}
```

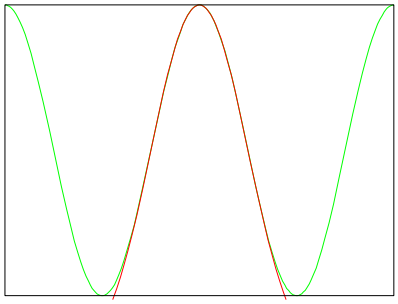


```
float FastInvTan1 (float fValue)
{
    float fVSqr = fValue*fValue;
    float fResult = 0.0028662257f;
    fResult *= fVSqr;
    fResult -= 0.0161657367f;
    fResult *= fVSqr;
    fResult += 0.0429096138f;
    fResult *= fVSqr;
    fResult -= 0.0752896400f;
    fResult *= fVSqr;
    fResult += 0.1065626393f;
    fResult *= fVSqr;
    fResult -= 0.1420889944f;
    fResult *= fVSqr;
    fResult += 0.1999355085f;
    fResult *= fVSqr;
    fResult -= 0.3333314528f;
    fResult *= fVSqr;
    fResult += 1.0f;
    fResult *= fValue;
    return fResult;
}
```

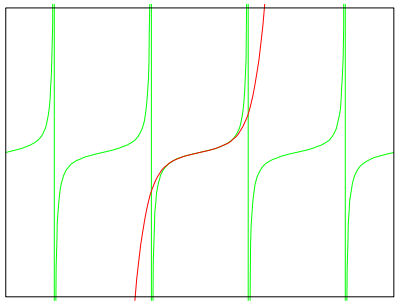


Even More Trig Approximations (adapted from post on devmaster.net forum)

```
__forceinline float fastCos(float x)
{
    const float x2 = x*x;
    const float x4 = x2*x*x;
    const float x6 = x4*x*x;
    return 1.0f - (x2 * 0.5f)
                + (x4 * 0.041666666666f)
                - (x6 * 0.00138888888888f);
}
```

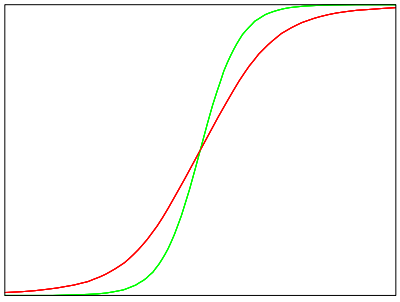


```
__forceinline float fastTan(float x)
{
   const float x2 = x*x;
   const float x3 = x2*x;
   const float x5 = x3*x2;
   const float x7 = x5*x2;
   return x + (x3 * 0.3333333333f)
            + (x5 * 0.1333333333333f)
            + (x7 * 0.0539682539f);
}
```



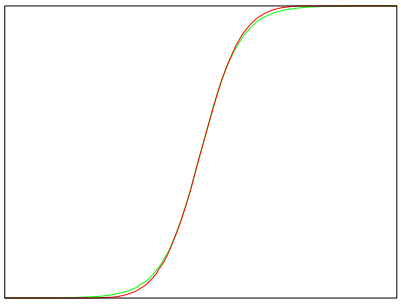
Tanh Approximation (adapted from musicdsp.org forum post)

```
__forceinline float fast_tanh(float x)
{
   float a=fabs(x);
   float b=12+a*(6+a*(3+a));
   return (x*b)/(a*b+24);
}
```



Better Tanh Approximation (posted by cschueler on musicdsp.org)

```
float rational_tanh(float x)
{
  if( x < -3 )
      return -1;
  else if( x > 3 )
      return 1;
  else
      return x * ( 27 + x * x ) / ( 27 + 9 * x * x );
}
```



Interpolation

Some often used float-functions and approximations.

Simple Linear interpolation

$$y = (1 - \alpha)y_0 + \alpha y_1$$

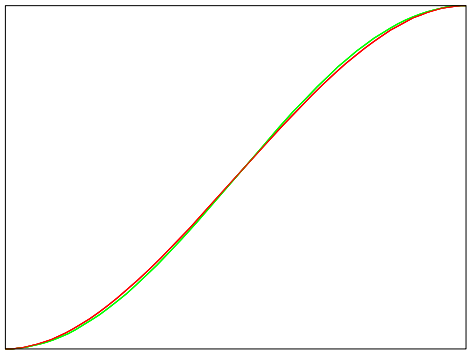
Smoothstep

For ease-in/ease-out and smoothing hard edges (flat tangents at *x=0* and *x=1*) [Green 2003]:

$$f(x) = \frac{1}{2} - \frac{\cos(\pi x)}{2}$$

Cheap polynomial approximation (Hermite) [Green 2003]:

$$f_{approx}(x) = 3x^2 - 2x^3$$



```
__forceinline float smoothstep(float x)
{
    return x*(3*x-2*x*x);
}
```

Miscellaneous Float Tricks

Simple Floating Point Compares

A multistage zero check works great with floats, e.g.:

```
(f==0.0f && g==0.0f)
```

becomes

```
( ((*(int*)&f | *(int*)&g)&0x7fffffff) == 0 )
```

Even simple comparisons can be faster as integer operations on the bitwise float representation:
f<g is simply *(int*)&f < *(int*)&g, for f and g >= 0 [Colwell00]. The replacement code below is IEEE-754 compliant for all classes of floating-point operands except NaNs. However, NaNs do not occur in properly working software [AMD02].

```
#define FLOAT2INTCAST(f)(*((int *)(&f)))
#define FLOAT2UINTCAST(f)(*((unsigned int *)(&f)))

//comparisons against zero
if (f<0.0f) ==> if (FLOAT2UINTCAST(f)>0x80000000U)
if (f<=0.0f)==> if (FLOAT2INCAST(f)<=0)
if (f>0.0f) ==> if (FLOAT2INTCAST(f)>0)
if (f>=0.0f)==> if (FLOAT2UINTCAST(f)<=0x80000000U)

//comparisons against positive constant
if (f<3.0f) ==> if (FLOAT2INTCAST(f)<0x40400000)
if (f<=3.0f)==> if (FLOAT2INTCAST(f)<=0x40400000)
if (f>3.0f) ==> if (FLOAT2INTCAST(f)>0x40400000)
if (f>=3.0f)==> if (FLOAT2INTCAST(f)>=0x40400000)

//comparisons among two floats
if (f1<f2)  ==> float t =f1-f2;if (FLOAT2UINTCAST(t)>0x80000000U)
if (f1<=f2) ==> float t =f1-f2;if (FLOAT2INTCAST(t)<=0)
if (f1>f2)  ==> float t =f1-f2;if (FLOAT2INTCAST(t)>0)
if (f1>=f2) ==> float t =f1-f2;if (FLOAT2UINTCAST(f)<=0x80000000U)
```

Branchless Floating Point Compare [Lomont05]

```
bool LomontCompare1(float af, float bf, int maxDiff)
{ // Fast, non constant time routine, portable
      int ai = *reinterpret_cast<int*>(&af);
      int bi = *reinterpret_cast<int*>(&bf);
      int diff = ai - bi; /* assumes both same sign */
      if (0x80000000 & (ai^bi)) // sign differed, so...
      diff = (0x80000000 - ai) - bi; // reverse one
      int v1 = maxDiff + diff;
      int v2 = maxDiff - diff;
      return (v1|v2) >= 0;
}
```

Branchless Floating Point Clamp

```
 __forceinline float f_clip(float x, float a, float b)
{
      return 0.5f * (fast_abs(x - a) + a + b - fast_abs(x - b));
}
```

Verify x87 Stack is Empty [Chad09]

Handy utility function which checks if the x87 is empty. Useful for checking if the FPU is left in an unpredictable state, either by your own code or due to a compiler issue.

```
__forceinline void verify_x87_stack_empty()
{
      unsigned z[8];
      __asm {
            fldz
            fldz
            fldz
            fldz
            fldz
            fldz
            fldz
            fldz
            fstp dword ptr [z+0x00]
            fstp dword ptr [z+0x04]
            fstp dword ptr [z+0x08]
            fstp dword ptr [z+0x0c]
            fstp dword ptr [z+0x10]
            fstp dword ptr [z+0x14]
            fstp dword ptr [z+0x18]
            fstp dword ptr [z+0x1c]
      }

      // Verify bit patterns. 0 = 0.0
      for (unsigned i = 0; i < 8; ++i)
      {
            if (z[i] != 0)
            {
                  __asm int 3;
            }
      }
}
```

Simultaneous Sin and Cos

```
__inline void FastSinCos(float angle, float &ress, float &resc) {
      float c, s
      __asm {
            fld angle
            fsincos
            fstp c
            fstp s
      }
      resc = c;
      ress = s;
}
```

Integer Log2 [Colwell00]

A neat trick is to use the FPU's fast integer to floating point conversion as a way to find the highest set bit position. This takes only a few cycles, much better than the usual linear or binary bit-searching loop:

```
int intlog2(int i) {
    float x=i;
    return (*(int*)&x >> 23) - 127;
}
```

Float 32 Bit Shifts

```
void fast_BSL(float &x, register unsigned long shiftAmount) {
        *(unsigned long*)&x+=shiftAmount<<23;
}

void fast_BSR(float &x, register unsigned long shiftAmount) {
        *(unsigned long*)&x-=shiftAmount<<23;
}
```

Single Precision Mode

Single precision mode makes **divides** and **sqrts** faster [AMD02, Fog06].
(sin, acos, log, etc., all run the same no matter what: 100+ cycles [Herf00]).

To set the FPU to single precision mode [Herf00]: `_controlfp( _PC_24, MCW_PC );`
To set it back to default (double) precision [Herf00]: `_controlfp( _CW_DEFAULT, 0xfffff );`

Rounding Mode

On the PC, the FPU has four different rounding modes. Those modes determine the behavior of float-to-int conversions through the *fist* or *fistp* functions [Terdiman06]:

	Chop (“Floor”)	Up (“Ceil”)	Down	Near (“Best”)
1.2	1	2	1	1
1.6	1	2	1	2
-1.2	-1	-1	-2	-1
-1.6	-1	-1	-2	-2

You can select a rounding mode by including <float.h> and using one of the following calls:

```
_controlfp(_RC_CHOP,    _MCW_RC);
_controlfp(_RC_UP,      _MCW_RC);
_controlfp(_RC_DOWN,    _MCW_RC);
_controlfp(_RC_NEAR,    _MCW_RC);
```

The default FPU rounding mode is “Near”. Most likely your program runs in this mode. By default the compiler will make code like this:

```
volatile float f = 1.5f;
int i = (int)f;
```

…call an _ftol function which apart from being an actual function-call also is slow in most compilers since it switches rounding mode on the FPU, does the cast and then switches back. The typical trick is to enable fast casts using the /QIfist compilation flag (works in VC6 and VC7). The _ftol call is then replaced with a direct “fistp” instruction inlined within the main code. But this means that the result depends on the current FPU rounding mode. One way to make it ANSI compliant is to set the FPU rounding mode to “floor” in the beginning of your program (or thread), and let /QIfist handle the rest:

```
_controlfp(_RC_CHOP,    _MCW_RC); // place in beginning of app.
volatile float f = 1.6f;
int i = (int)f;
```

However, [Terdiman06] reports that switching overall rounding mode, may have side-effects on the precision on internal floating point math functions in debug mode. So beware!

While the /Qifist switch is very convenient, you may get even better performance by using SSE and SSE2 conversion instructions. Specifically, the instructions CVTTSS2SI and CVTTSD2SI perform conversion with truncation for floats and doubles, respectively. Rather than inlining assembly code, these instructions are best implemented with intrinsic functions [Intel-fistp]:

```
i = _mm_cvtt_ss2si(_mm_load_ss(&x));  // single precision
i = _mm_cvttsd_si32(_mm_load_sd(&x)); // double precision
```

Here, the "x" variable holds a floating-point value, and "i" is of type int or long. (use #include "emmintrin.h" and C++ Processor Pack (Service Pack 5)).
You can use this _ftol replacement with an appropriate breakpoint to track down nasty _ftol calls [Herf00]:

```
// only use this code to test and track nasty ftol calls
#pragma message ("** Tracking nasty ftol calls!")
#define ANSI_FTOL 1
extern "C" {
    __declspec(naked) void _ftol() { __asm    {
#if ANSI_FTOL
            fnstcw WORD PTR [esp-2]
            mov    ax, WORD PTR [esp-2]
            OR AX, 0C00h
            mov    WORD PTR [esp-4], ax
            fldcw  WORD PTR [esp-4]
            fistp  QWORD PTR [esp-12]
            fldcw  WORD PTR [esp-2]
            mov    eax, DWORD PTR [esp-12]
            mov    edx, DWORD PTR [esp-8]
#else
            fistp  DWORD PTR [esp-12]
            mov      eax, DWORD PTR [esp-12]
            mov      ecx, DWORD PTR [esp-8]
#endif
            ret
        }
    }
}
```

In VC7 and beyond this is called _ftol2 and _ftol2_sse. Alternatively, simply set a function breakpoint.

Pseudo Random Number Generator [by iq/rgba]

Faster and better quality than using a scaled rand(), since 23 random bits are generated instead of 15. Density distribution is uniform. Returned number is between -1.0 and 1.0.

```
static unsigned int mirand = 1;
float sfrand( void )
{
    unsigned int a;
    mirand *= 16807;
    a = (mirand&0x007fffff) | 0x40000000;
    return( *((float*)&a) - 3.0f );
}
```

Denormals / Subnormals

Very small floats cannot be coded with the ”normal” format. The denormal coding is specified by a null biased exponent [DeSoras02]:

$$(-1)^S * M * 2^{(1 - B - P)}$$

The implicit 1 is removed to reach the smallest numbers by zeroing the most significant bits of the mantissa. But denormals are much slower than normal represented numbers ! The penalty for handling denormals is up to 1500 cycles [DevCon05].

Slow but educational Denormal killer [DeSoras02]

```
void test_and_kill_denormal (float &val) {
   const int x = *reinterpret_cast <const int *> (&val);
   const int abs_mantissa = x & 0x007FFFFF;
   const int biased_exponent = x & 0x7F800000;
   if (biased_exponent == 0 && abs_mantissa != 0) val =0;
}
```

Macro that kills denormals:

```
#define DENORMALIZE(fv)  (((*(unsigned int*)&(fv))&0x7f800000)<0x08000000)?0.0f:(fv)
```

A more conservative denormalizer that kills numbers below 1.e-30 :

```
#define DENORMALIZE(fv) (((*(unsigned int*)&(fv))&0x7FFFFFFF)<0x0da24260)?0.0f:(fv)
```

Alternative denormalizer: adds and subtracts a small (anti-denormal) value [DeSoras02]:

```
#define DENORMALIZE(fv) { fv += 1.e-18f; fv -= 1.e-18f; }
```

Extra test macros:

```
#define IS_DENORMAL(f) (((*(unsigned int *)&(f))&0x7f800000) == 0)

#define IS_ALMOST_DENORMAL(f) (((*(unsigned int *)&(f))&0x7f800000) < 0x08000000)
```

Check to see if FPU is in denormal mode [DeSoras02]:

```
bool is_denormalized ()  {
    short int status;
    __asm {
          fstsw word ptr [status] ; Retrieve FPU status word
          fclex ; Clear FPU exceptions (denormal flag)
    }
    return ((status & 0x0002) != 0);
}
```

Macro to break on FPU exceptions:

```
#define ENABLE_FPU_EXCEPTIONS { \
    _control87(0,_EM_OVERFLOW | _EM_UNDERFLOW | _EM_ZERODIVIDE | _EM_INVALID); \
}
```

Traps overflow, underflow(denormals), division by zero, and invalid operations.

Generally you can control the FPU control word by doing stuff like:

```
unsigned int cw = _control87( 0, 0 );     // get previous control word
_control87( _PC_24, MCW_PC )  // set precision to 24 bits (_PC_24, _PC_53, _PC_64 )
_control87( cw, 0xffffff );    // restore to previous control word ...or...
_control87( _CW_DEFAULT, 0xffffff );   // restore to default
```

Clamp SSE denormals to zero:

```
int SSE_AntiDenormal_Begin() {
    int oldMXCSR = _mm_getcsr(); //read the old MXCSR setting
    int newMXCSR = oldMXCSR | 0x8040; // set DAZ and FZ bits
    _mm_setcsr( newMXCSR ); //write the new MXCSR setting to the MXCSR
    return oldMXCSR;
}

void SSE_AntiDenormal_End(int oldMXCSR) {
    _mm_setcsr( oldMXCSR );  //restore old MXCSR settings
}
```

Before running this code one should check that SSE and DAZ mode is supported [Intel485-05], like:

```
static int CheckDAZ() {
      static __declspec(align(16)) unsigned char fxsavedata[512+16];
      memset(fxsavedata, 0, 512+16);
      unsigned char* pFxSaveData = (unsigned char*)fxsavedata;
      int result = 0;
      __asm {
            pusha
            ; Get CPU feature flags...
            ; Verify FXSAVE and either SSE or SSE2 are supported
            mov eax, 1
            cpuid
            bt edx, 24 ; Feature Flags Bit 24 is FXSAVE support
            jnc nodaz ; jump if FXSAVE not supported
            bt edx, 25 ; Feature Flags Bit 25 is SSE support
            jnc nodaz ; jump if SSE is not supported
            bt edx, 26 ; Feature Flags Bit 26 is SSE2 support
            jnc nodaz ; jump if SSE2 is not supported
            mov edx, pFxSaveData
            fxsave [edx]
            mov eax, DWORD PTR [edx][28] ; Get MXCSR_MASK
            cmp eax, 0 ; Check for valid mask
            jne check_mxcsr_mask
            mov eax, 0FFBFh ; Force use of default MXCSR_MASK
            check_mxcsr_mask:
            ; EAX contains MXCSR_MASK from FXSAVE buffer or default mask
            bt eax, 6 ; MXCSR_MASK Bit 6 is DAZ support
            jnc nodaz ; Jump if DAZ supported
            mov result, 1
            nodaz:
            popa
      }
      return result;
}
```

References

[AMD02] "AMD Athlon™ Processor x86 Code Optimization Guide", AMD, 2002.

[Blinn97] Jim Blinn. Jim blinn's corner: "Floating-point tricks", IEEE Computer Graphics & Applications, 17(4), July--August 1997.

[Chad09] Chad Austin, "#IND and #QNaN with /fp:fast", https://chadaustin.me/2009/02/ind-and-qnan-with-fpfast/, 2009.

[Colwell00] Steve Colwell, "CodeTips", http://stevecolwell.com/codetips.html, 2000.

[DevCon05] "SSE Performance Programming", Apple Developer Connection, http://developer.apple.com/hardware/ve/sse.html, 2005.

[Fog06] Agner Fog, "Optimizing subroutines in assembly language: An optimization guide for x86 platforms", http://www.agner.org/optimize/ , 2006.

[Goldberg 91] David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", ACM Computing Surveys, 1991.

[Green03] Robin Green. "Faster Math Functions", Presentation, Game Developers Conference, 2003

[Hsieh96] Paul Hsieh, "How to calculate square roots", http://www.azillionmonkeys.com/qed/sqroot.html, 1996.

[Intel485-05] "Intel Processor Identification and the CPUID Instruction", Application Note 485, 2005.

[Intel803-99] "Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method", Application Note AP-803, http://cache-www.intel.com/cd/00/00/04/10/41007_nrmethod.pdf, 1999

[Intel-fistp] "Fast Floating Point to Integer Conversions", http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/xeon/optimization/19938.htm, 2005.

[Karvonen98] Vesa Karvonen and Agner Fog: "Fast Float to Int", *The Assembly Gems page*, http://www.df.lth.se/~john_e/fr_gems.html, 1998.

[Lomont03] Chris Lomont, "Fast Inverse Square Root", 2003.

[Lomont05] Chris Lomont, "Taming the Floating Point Beast", http://www.lomont.org/Math/Papers/2005/CompareFloat.pdf, 2005

[DeSoras02] Laurent de Soras, "Denormal numbers in floating point signal processing applications", 2002.

[DeSoras04] Laurent de Soras, "Fast Rounding of Floating Point Numbers in C/C++on Wintel Platform", http://ldesoras.free.fr, 2004.

[Herf00] Michael Herf, “Know your FPU”, http://www.stereopsis.com/FPU.html, 2000.

[MusicDSP] Music DSP archive, http://www.musicdsp.org/archive.php?classid=0, 2002.

[Rocatis04] Jon Rocatis, “Fast Reciprocal Square Root Approximation”, http://playstation2-linux.com/download/p2lsd/fastrsqrt.pdf, 2004.

[Schraudolph98] Nicol N. Schraudolph , “A Fast, Compact Approximation of the Exponential Function”, IDSIA, Lugano, Switzerland, http://users.rsise.anu.edu.au/~nici/pubs/Schraudolph99.pdf, 1998.

[Schlick94] Christophe Schlick, “A Fast Alternative to Phong’s Specular Model”, Graphics Gems 4, 1994.

[Stephenson] Ian Stephenson, “fast floating point power computation”, http://playstation2-linux.com/download/adam/power.c .

[Terdiman06] Pierre Terdiman, “FPU fun” , http://www.codercorner.com/FPUFun.htm, 2006

[Vellocet03] “Fast Log Approximation”, http://www.vellocet.com/dsp, 2003.