```
/*
 * Copyright (c) 2003, 2007-8 Matteo Frigo
 * Copyright (c) 2003, 2007-8 Massachusetts Institute of Technology
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */

#ifndef FFTW_SINGLE
```
if FFTW_SINGLE is not defined, then compile code until endif

```
#error "SSE only works in single precision"
```
Puts out a compile time error message. This forces you to compile FFTW in single precision mode because that is all that is implemented for SSE. This guards against someone doing the wrong thing at compile time.

```
#endif
```
end #ifndef

```
#define VL 2            /* SIMD complex vector length */
#define ALIGNMENT 8     /* alignment for LD/ST */
#define ALIGNMENTA 16   /* alignment for LDA/STA */
#define SIMD_VSTRIDE_OKA(x) ((x) == 2)
#define SIMD_STRIDE_OKPAIR SIMD_STRIDE_OK

#define RIGHT_CPU X(have_sse)
```

A bunch of #defines. SIMD complex vector length of 2 means single instruction two data. Loads and stores are aligned to 8 (bytes?). If you have aligned data, then you can use these special instructions for aligned data, the LDA/STA. And, it's 18 instead of 8 for the regular load and store.

Stride is how far into the data you go for each step. When you go to the next set of data, how many do you go in? This is a function-like macro. When it's called later in the code, ((x) == 2) is a test. If x == 2 then it returns true. If x != 2 then it returns false. This is important in the context of wherever SIMD_VSTRIDE_OKA(x) is called. RIGHT_CPU X(have_sse) means that whenever RIGHT_CPU is invoked, then it's replaced with X(have_sse).

```
extern int RIGHT_CPU(void);
```
function declaration. Extern means the function is visible outside of this file. Int means the function returns an integer. RIGHT_CPU is from the #define list above, and expands to X(have_sse). (void) means it takes no arguments. So, it looks like this: extern int X(have_sse)(void); In ifftw.h, there is a #define X(name) for single and double precision. Now, we're in single precision land, so we expand the single precision X(name). It expands from X(name) to CONCAT(fftwf_, name). So, now we have extern int CONCAT(fftwf_, have_sse)(void); Now, ifftw.h has another useful #define for us. #define CONCAT(prefix, name) prefix ## name

gives you token pasting. It leaps out above the parser (like batman), and pastes the tokens together.

So, now we have extern int fftwf_have_sse(void);

This successfully name mangles up the right name for the function.

If we were compiling for double precision as well as single precision, then we would have two functions. One would have fftwf as a prefix and the other would have fftwl as a prefix.

```
/* gcc compiles the following code only when __SSE__ is defined */
#if defined(__SSE__) || !defined(__GNUC__)
```

#if defined(__SSE__) is something built into gcc to test for SSE? Similarly, !defined(__GNUC__) is to test if we're not using gcc, and therefore cannot rely on the __SSE__ built-in thingies. The #endif for this #if is at the very end of the file.

```
/* some versions of glibc's sys/cdefs.h define __inline to be empty,
   which is wrong because xmmintrin.h defines several inline
   procedures */
#undef __inline
```
Turn off that version of glibc's sys/cdefs.h define __inline. Make it work!

```
#include <xmmintrin.h>
```
Include the header file that defines several inline procedures. We fixed things up for this in the previous line. All SSE instructions and __m128 data types are defined in xmmintrin.h file.[1]

```
typedef __m128 V;
```
set up a new name (V) for an existing type (__m128). V probably stands for vector. __m128 data types are defined in xmmintrin.h. "The data type intended for user use" is the comment about __m128 in xmmintrin.h. As opposed, to, say, some other type of use!

---

[1] http://www.codeproject.com/KB/recipes/sseintro.aspx

```
#define VADD _mm_add_ps
#define VSUB _mm_sub_ps
#define VMUL _mm_mul_ps
#define VXOR _mm_xor_ps
#define SHUFPS _mm_shuffle_ps
#define STOREH(addr, val) _mm_storeh_pi((__m64 *)(addr), val)
#define STOREL(addr, val) _mm_storel_pi((__m64 *)(addr), val)
#define UNPCKH _mm_unpackhi_ps
#define UNPCKL _mm_unpacklo_ps
```

A bunch of #defines.

It looks assembly code instructions redefined to a different style of assembly code instructions.

__mm_add_ps is a thing that adds four single precision floats. We found this from a MSDN website. Microsoft developer network. VADD is a vector add. These seem to be the generic names that FFTW uses across all code. So, whatever on the particular SIMD works as a vector add, say, is then turned into VADD by this set of #defines. This maps the instructions from the SIMD processor into instructions for FFTW. Abstraction.

```
#ifdef __GNUC__
#  define DVK(var, val) const V var = __extension__ ({      \
     static const union fvec _var = { {val, val, val, val} };    \
     _var.v;                                          \
   })
#  define LDK(x) x

  /* we use inline asm because gcc generates slow code for
     _mm_loadh_pi().  gcc insists upon having an existing variable for
     VAL, which is however never used.  Thus, it generates code to move
     values in and out the variable.  Worse still, gcc-4.0 stores VAL on
     the stack, causing valgrind to complain about uninitialized reads.
  */


  static inline V LD(const R *x, INT ivs, const R *aligned_like)
  {
       V var;
       (void)aligned_like; /* UNUSED */
       __asm__("movlps %1, %0\n\tmovhps %2, %0"
           : "=x"(var) : "m"(x[0]), "m"(x[ivs]));
       return var;
  }
```

This is a bug workaround.

```
#else
```

We are not using gcc compiler, which means we can rely on _mm_loadh_pi() working.

```
# define DVK(var, val) const R var = K(val)
# define LDK(x) _mm_set_ps1(x)
# define LOADH(addr, val) _mm_loadh_pi(val, (const __m64 *)(addr))
# define LOADL0(addr, val) _mm_loadl_pi(val, (const __m64 *)(addr))
```

a bunch of #defines. K is probably another macro that expands.

For single precision, in ifftw.h, #define K(x) ((E) x)

So, K(val) turns into ((E) val)

So, DVK(var, val) const R var = ((E) val)

Also, from ifftw.h, E is a typedef for R. This isn't a textual substitution thing, though, like the earlier #defines that name mangled the correct function name.

Let's find a place where it gets used to see what happens.

There are lots and lots of them! DVK is used a lot. However, none are used in sse.

DVK(this name of type R, is this constant also of type R – under possible coercion)

Define vector constant?

#define LDK(x) _mm_set_ps1(x)

_mm_set_ps1(x) sets the four single-precision floating-point values to x. Constant load – initializes everything at once to some value. A useful SIMD command.

Again, we're mapping sse instructions to FFTW instruction names.

The next two #defines load, perhaps, the two halves of an __m128 thingie. They're just used here in this particular inline function definition.

```
static inline V LD(const R *x, INT ivs, const R *aligned_like)
{
     V var;
     (void)aligned_like; /* UNUSED */
     var = LOADL0(x, var);
     var = LOADH(x + ivs, var);
     return var;
}
```

Static qualifies a function that is being defined. A static function is not visible outside of the file. Inline is a keyword. This code is placed inline whenever the function call is made. It is a hint to the compiler. V means __m128, which is a vector, and it's the return type of this function which is called LD. LD has three arguments.

```
#endif
```
end of test for gcc compiler

```
union fvec {
     R f[4];
     V v;
};

union uvec {
     unsigned u[4];
     V v;
};

#define VFMA(a, b, c) VADD(c, VMUL(a, b))
#define VFNMS(a, b, c) VSUB(c, VMUL(a, b))
#define VFMS(a, b, c) VSUB(VMUL(a, b), c)

#define SHUFVAL(fp0,fp1,fp2,fp3) \
   (((fp3) << 6) | ((fp2) << 4) | ((fp1) << 2) | ((fp0)))


static inline V LDA(const R *x, INT ivs, const R *aligned_like)
{
     (void)aligned_like; /* UNUSED */
     (void)ivs; /* UNUSED */
     return *(const V *)x;
}

static inline void ST(R *x, V v, INT ovs, const R *aligned_like)
{
     (void)aligned_like; /* UNUSED */
     /* WARNING: the extra_iter hack depends upon STOREL occurring
    after STOREH */
     STOREH(x + ovs, v);
     STOREL(x, v);
}

static inline void STA(R *x, V v, INT ovs, const R *aligned_like)
{
     (void)aligned_like; /* UNUSED */
     (void)ovs; /* UNUSED */
     *(V *)x = v;
}

#if 0
/* this should be faster but it isn't. */
static inline void STN2(R *x, V v0, V v1, INT ovs)
{
     STA(x, SHUFPS(v0, v1, SHUFVAL(0, 1, 0, 1)), ovs, 0);
```

```
     STA(x + ovs, SHUFPS(v0, v1, SHUFVAL(2, 3, 2, 3)), ovs, 0);
}
#endif
#define STM2 ST
#define STN2(x, v0, v1, ovs) /* nop */

#define STM4(x, v, ovs, aligned_like) /* no-op */

#ifdef VISUAL_CXX_DOES_NOT_SUCK
static inline void STN4(R *x, V v0, V v1, V v2, V v3, INT ovs)
{
     V x0, x1, x2, x3;
     x0 = UNPCKL(v0, v2);
     x1 = UNPCKH(v0, v2);
     x2 = UNPCKL(v1, v3);
     x3 = UNPCKH(v1, v3);
     STA(x, UNPCKL(x0, x2), 0, 0);
     STA(x + ovs, UNPCKH(x0, x2), 0, 0);
     STA(x + 2 * ovs, UNPCKL(x1, x3), 0, 0);
     STA(x + 3 * ovs, UNPCKH(x1, x3), 0, 0);
}
#else /* Visual C++ sucks */

/*
  Straight from the mouth of the horse:

     We "reserved" the possibility of aligning arguments with
     __declspec(align(X)) passed by value by issuing this error.

     The first 3 parameters of type __m64 (or other MMX types) are
     passed in registers.  The rest would be passed on the stack.  We
     decided aligning the stack was wasteful, especially for __m128
     parameters.  Also, we thought it would be infrequent that people
     would want to pass more than 3 by value.

     If we didn't issue an error, we would have to binary compatibility
     in the future if we decided to align the arguments.


     Hope that explains it.
     --
     Jason Shirk, Visual C++ Compiler Team
     This posting is provided AS IS with no warranties, and confers no
rights
*/

#define STN4(x, v0, v1, v2, v3, ovs)             \
{                                  \
     V xxx0, xxx1, xxx2, xxx3;                   \
     xxx0 = UNPCKL(v0, v2);                \
```

```
     xxx1 = UNPCKH(v0, v2);                \
     xxx2 = UNPCKL(v1, v3);                \
     xxx3 = UNPCKH(v1, v3);                \
     STA(x, UNPCKL(xxx0, xxx2), 0, 0);           \
     STA(x + ovs, UNPCKH(xxx0, xxx2), 0, 0);          \
     STA(x + 2 * ovs, UNPCKL(xxx1, xxx3), 0, 0); \
     STA(x + 3 * ovs, UNPCKH(xxx1, xxx3), 0, 0); \
}
#endif

static inline V FLIP_RI(V x)
{
     return SHUFPS(x, x, SHUFVAL(1, 0, 3, 2));
}

extern const union uvec X(sse_pmpm);
static inline V VCONJ(V x)
{
     return VXOR(X(sse_pmpm).v, x);
}

static inline V VBYI(V x)
{
     return FLIP_RI(VCONJ(x));
}

static inline V VZMUL(V tx, V sr)
{
     V tr = SHUFPS(tx, tx, SHUFVAL(0, 0, 2, 2));
     V ti = SHUFPS(tx, tx, SHUFVAL(1, 1, 3, 3));
     tr = VMUL(tr, sr);
     sr = VBYI(sr);
     return VADD(tr, VMUL(ti, sr));
}

static inline V VZMULJ(V tx, V sr)
{
     V tr = SHUFPS(tx, tx, SHUFVAL(0, 0, 2, 2));
     V ti = SHUFPS(tx, tx, SHUFVAL(1, 1, 3, 3));
     tr = VMUL(tr, sr);
     sr = VBYI(sr);
     return VSUB(tr, VMUL(ti, sr));
}

static inline V VZMULI(V tx, V sr)
{
     V tr = SHUFPS(tx, tx, SHUFVAL(0, 0, 2, 2));
     V ti = SHUFPS(tx, tx, SHUFVAL(1, 1, 3, 3));
     ti = VMUL(ti, sr);
     sr = VBYI(sr);
```

```
     return VSUB(VMUL(tr, sr), ti);
}

static inline V VZMULIJ(V tx, V sr)
{
     V tr = SHUFPS(tx, tx, SHUFVAL(0, 0, 2, 2));
     V ti = SHUFPS(tx, tx, SHUFVAL(1, 1, 3, 3));
     ti = VMUL(ti, sr);
     sr = VBYI(sr);
     return VADD(VMUL(tr, sr), ti);
}

#define VFMAI(b, c) VADD(c, VBYI(b))
#define VFNMSI(b, c) VSUB(c, VBYI(b))

/* twiddle storage #1: compact, slower */
#define VTW1(v,x)  \
  {TW_COS, v, x}, {TW_COS, v+1, x}, {TW_SIN, v, x}, {TW_SIN, v+1, x}
#define TWVL1 (VL)

static inline V BYTW1(const R *t, V sr)
{
     const V *twp = (const V *)t;
     V tx = twp[0];
     V tr = UNPCKL(tx, tx);
     V ti = UNPCKH(tx, tx);
     tr = VMUL(tr, sr);
     sr = VBYI(sr);
     return VADD(tr, VMUL(ti, sr));
}

static inline V BYTWJ1(const R *t, V sr)
{
     const V *twp = (const V *)t;
     V tx = twp[0];
     V tr = UNPCKL(tx, tx);
     V ti = UNPCKH(tx, tx);
     tr = VMUL(tr, sr);
     sr = VBYI(sr);
     return VSUB(tr, VMUL(ti, sr));
}

/* twiddle storage #2: twice the space, faster (when in cache) */
#define VTW2(v,x)                                               \
  {TW_COS, v, x}, {TW_COS, v, x}, {TW_COS, v+1, x}, {TW_COS, v+1, x},  \
  {TW_SIN, v, -x}, {TW_SIN, v, x}, {TW_SIN, v+1, -x}, {TW_SIN, v+1, x}
#define TWVL2 (2 * VL)

static inline V BYTW2(const R *t, V sr)
{
```

```
     const V *twp = (const V *)t;
     V si = FLIP_RI(sr);
     V tr = twp[0], ti = twp[1];
     return VADD(VMUL(tr, sr), VMUL(ti, si));
}

static inline V BYTWJ2(const R *t, V sr)
{
     const V *twp = (const V *)t;
     V si = FLIP_RI(sr);
     V tr = twp[0], ti = twp[1];
     return VSUB(VMUL(tr, sr), VMUL(ti, si));
}

/* twiddle storage #3 */
#define VTW3(v,x) {TW_CEXP, v, x}, {TW_CEXP, v+1, x}
#define TWVL3 (VL)

/* twiddle storage for split arrays */
#define VTWS(v,x)                                                       \
  {TW_COS, v, x}, {TW_COS, v+1, x}, {TW_COS, v+2, x}, {TW_COS, v+3, x},
    \
  {TW_SIN, v, x}, {TW_SIN, v+1, x}, {TW_SIN, v+2, x}, {TW_SIN, v+3, x}
#define TWVLS (2 * VL)

#endif /* __SSE__ */
```