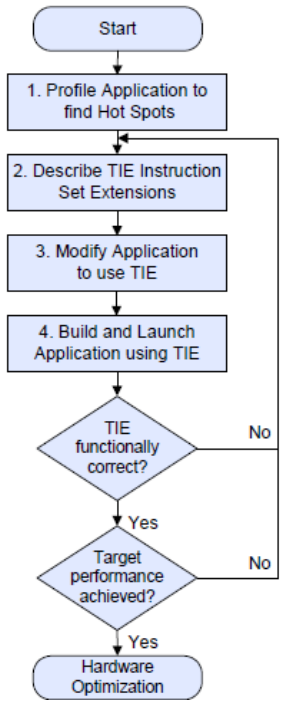
Introduction

The HW/SW-Codesign Lab is intended to give a first practical access to this topic and experiencing the potential of the conjoint design of hardware and software. For this purpose the reconfigurable processor flow from the company **Tensilica** is used (shown right). The tool suite provides convenient means for analyzing and optimizing the performance of the software. Especially the possibility to easily extend the processor's hardware architecture by customer specific instructions using the **Tensilica Instruction Extension (TIE)** language is well suited to demonstrate the interaction between hardware and software design.



Tensilica®, Instruction Extension (TIE) Language, User's Guide, 02/2014, p.4

Within the scope of this Lab the following 2 exercises should be done:

1. Implementation of a FIR filter design including performance analysis; HW/SW optimization: e.g. by Fusion, SIMD extension, etc.; also consider different implementation alternatives for the FIR
2. Improve the performance of a given FFT/IFFT algorithm by using basic instruction extension concepts (Fusion/SIMD/FLIX/etc.). Also consider different implementations such as the Decimation in Time (DIT) and Decimation in Frequency (DIF) algorithm.

The written report should only include the results of the second task and need not exceed 3 pages (source code excerpts not included)!

The detailed project description can be found at the lab webpage:
http://mns.ifn.et.tu-dresden.de/Teaching/Courses/Pages/P-HWSW-Codesign.aspx

Author(s)

Name	E-Mail
Jörg Thalheim	joerg@higgsboson.tk
Patrick Schöps	stoepsel@stoepsel.net
Alfred Krohmer	alfred.krohmer@tu-dresden.de

If desired, the names/email will be removed from the document before publishing it on our web page.

FFT/IFFT algorithm acceleration

Source code C files (only excerpts from the parts that have been changed; please add line numbers to the code and highlight important parts):

dit-flix.h:

```
#ifndef FFT_ASM_H
#define FFT_ASM_H

#define FLIX

#ifdef FLIX
#define FFT_FLIX_SIMD_STORE( fr,  fi,  i,  simd_r,  simd_i) \
        asm ("{                                          \n" \
             "    fft_simd_store %1, %0, %2              \n" \
             "    nop                                    \n" \
             "    fft_simd_store %3, %0, %4              \n" \
             "}" \
             :: "r" (_i), "r" (_fr), "r" (_simd_r), "r" (_fi), "r" (_simd_i));

#define FLIX_S16I(_p1, _v1, _p2, _v2) \
        asm ("{                         \n" \
             "    s16i %1, %0, 0        \n" \
             "    nop                   \n" \
             "    s16i %3, %2, 0        \n" \
             "}" \
             :: "r" ( p1), "r" ( v1), "r" ( p2), "r" ( v2) \
             : "memory");

#define FFT_FLIX_SIMD_THIRD(i, simd_r, simd_r2, simd_i, simd_i2, shift, inverse) \
        do { \
                /* also i += 4 in the next cycle */ \
                asm ("{                                                          \n" \
                                "    fft_simd_third %0, %1, %2, %3, %4       \n" \
                                "    addi %0, %0, 4                          \n" \
                                "    nop                                     \n" \
                                "}" \
                                :: "r" (i), "r" (simd_r), "r" (simd_i), "r" (shift), "r" (inverse)); \
                /* also i -= 4 in the next cycle */ \
                asm ("{                                                                  \n" \
                                "    fft simd third %0, %1, %2, %3, %4                   \n" \
                                "    addi %0, %0, -4                                     \n" \
                                "    nop                                                 \n" \
                                "}" \
                                :: "r" (i), "r" (simd_r2), "r" (simd_i2), "r" (shift), "r" (inverse)); \
        } while(0)
#define FFT_FLIX_SIMD_STORE_SHUFFLE(i, j, fr, simd_r, simd_r2, fi, simd_i, simd_i2) \
        do { \
                /* also i++ */ \
                asm ("{                                                          \n" \
                     "    fft_simd_store_shuffle %1, %0, %2, %5, 0              \n" \
                     "    addi %0, %0, 8                                        \n" \
                     "    fft simd store shuffle %3, %0, %4, %6, 0              \n" \
                     "}" \
                     /*  0          1          2              3          4*/ \
                     : "=r" (i) : "r" (fr), "r" (simd_r), "r" (fi), "r" (simd_i), \
                     "r" (simd_r2), "r" (simd_i2)); \
                   /* 5                6*/ \
                /* also j = i + 1; */ \
                asm ("{                                                          \n" \
                     "    fft_simd_store_shuffle %3, %0, %4, %7, 1              \n" \
                     "    add.n %0, %2, %9                                      \n" \
                     "    fft_simd_store_shuffle %5, %0, %6, %8, 1              \n" \
                     "}" \
                     /*  0             1          2              3             4*/ \
                     : "=r" (j)     : "r" (j),  "r" (i),       "r" (fr),      "r" (simd r), \
                     "r" (fi), "r" (simd i), "r" (simd r2), "r" (simd i2), "r" (1)); \
                         /* 5             6          7              8             9*/ \
        } while(0)
#else
```

```
#define FFT_FLIX_SIMD_STORE(_fr, _fi, _i, _simd_r, _simd_i) \
        FFT_SIMD_STORE(fr, i, simd_r); \
        FFT_SIMD_STORE(fi, i, simd_i); \

#define FFT_FLIX_SIMD_THIRD(i, simd_r, simd_r2, simd_i, simd_i2, shift, inverse) \
        FFT_SIMD_THIRD(i, simd_r, simd_i, shift, inverse); \
        FFT_SIMD_THIRD(i + 4, simd_r2, simd_i2, shift, inverse);
#define FLIX_S16I(_p1, _v1, _p2, _v2) \
        do { \
                *( p1) = ( v1); \
                *(_p2) = (_v2); \
        } while(0)
#define FFT_FLIX_SIMD_STORE_SHUFFLE(i, j, fr, simd_r, simd_r2, fi, simd_i, simd_i2) \
        do { \
                FFT_SIMD_STORE_SHUFFLE(fr, i, simd_r, simd_r2, 0); \
                FFT_SIMD_STORE_SHUFFLE(fi, i, simd_i, simd_i2, 0); \
                FFT_SIMD_STORE_SHUFFLE(fr, j, simd_r, simd_r2, 1); \
                FFT_SIMD_STORE_SHUFFLE(fi, j, simd_i, simd_i2, 1); \
                i++; \
                j = i + 1; \
        } while(0)

#endif // FLIX

#define FFT_FLIX_SIMD_LOAD_SHUFFLE(i, fr, simd_r, simd_r2, fi, simd_i, simd_i2, high) \
        do { \
                FFT_SIMD_LOAD_SHUFFLE(fr, i, simd_r, simd_r2, high); \
                FFT_SIMD_LOAD_SHUFFLE(fi, i, simd_i, simd_i2, high); \
        } while(0)

#endif // FFT_ASM_H
```

dit.c:

```
#include "dit.h"
#include "dit-flix.h"

#include <xtensa/tie/dit.h>


int fix dit fft(fixed fr[], fixed fi[], int size, int  inverse)
{
    int i, j, l, k, istep, n, m;
    xtbool shift, inverse = _inverse;
    register FFT_REG_SIMD simd_r, simd_i, simd_r2, simd_i2;

    //number of input data
    n = 1 << size;
    if(n > N_WAVE) return -1;

    int scale = 0;

    /* decimation in time - re-order data */
    for (m = 1; m < n; m++) {
        int mr = FFT BIT REVERSE(m, size);

        if(mr <= m) continue;
        int ti = fi[m];
        int tr = fr[m];

        FLIX_S16I(&fr[m], fr[mr], &fi[m], fi[mr]);
        FLIX_S16I(&fi[mr], ti, &fr[mr], tr);
    }

    l = 1;
    k = LOG2_N_WAVE-1;
    while (l < n)
    {
        if (inverse)
        {
            /* variable scaling, depending upon data */
            shift = 0;
            for (i = 0; i < (n / 8); i += 8)
            {
                if (FFT SHIFT CHECK(fr, i) | FFT SHIFT CHECK(fi, i))
                {
                    shift = 1;
                    ++scale;
                    break;
                }
```

```
            }
        }
        else
        {
            /* fixed scaling, for proper normalization -
               there will be log2(n) passes, so this
               results in an overall factor of 1/n,
               distributed to maximize arithmetic accuracy. */
            shift = 1;
        }

        /* it may not be obvious, but the shift will be performed
           on each data point exactly once, during this pass. */
        istep = l << 1;  //step width of current butterfly

        switch(l)
        {
            case 1:
                for (i = 0; i < n; i += 8)
                {
                    simd_r = FFT_SIMD_LOAD(fr, i);
                    simd_i = FFT_SIMD_LOAD(fi, i);
                    FFT SIMD FIRST(simd r, simd i, shift);
                    FFT FLIX SIMD STORE(fr, fi, i, simd r, simd i);
                }
                break;

            case 2:
                for (i = 0; i < n; i += 8)
                {
                    simd_r = FFT_SIMD_LOAD(fr, i);
                    simd_i = FFT_SIMD_LOAD(fi, i);
                    FFT_SIMD_SECOND(simd_r, simd_i, shift, inverse);
                    FFT_FLIX_SIMD_STORE(fr, fi, i, simd_r, simd_i);
                }
                break;

            case 4:
                WUR_FFT_SIMD_K(7);
                for (i = 0; i < n; i += 8)
                {
                    simd_r = FFT_SIMD_LOAD(fr, i);
                    simd i = FFT SIMD LOAD(fi, i);
                    FFT_SIMD_THIRD(i, simd_r, simd_i, shift, inverse);
                    FFT_FLIX_SIMD_STORE(fr, fi, i, simd_r, simd_i);
                }
                break;

            default:
                WUR FFT SIMD K(k);
                for (m = 0; m < n; m += istep)
                {
                    j = m + l;
                    for (i = m; i < m + l;)
                    {
                        FFT FLIX SIMD LOAD SHUFFLE(i, fr, simd r, simd r2, fi, simd i, simd i2, 0);
                        FFT FLIX SIMD LOAD SHUFFLE(j, fr, simd r, simd r2, fi, simd i, simd i2, 1);

                        FFT_FLIX_SIMD_THIRD(i, simd_r, simd_r2, simd_i, simd_i2, shift, inverse);

                        // inlines also i++ and j = i + l
                        FFT_FLIX_SIMD_STORE_SHUFFLE(i, j, fr, simd_r, simd_r2, fi, simd_i, simd_i2);
                    }
                }
                break;
        }
        --k;
        l = istep;
    }

    return scale;
}
```

Source code TIE files (please add line numbers to the code):

```
table SIN_WAVE 16 257 {
    0, 201, 402, 603, 804, 1005, 1206, 1406,
    1607, 1808, 2009, 2209, 2410, 2610, 2811, 3011,
    3211, 3411, 3611, 3811, 4011, 4210, 4409, 4608,
    4807, 5006, 5205, 5403, 5601, 5799, 5997, 6195,
    6392, 6589, 6786, 6982, 7179, 7375, 7571, 7766,
    7961, 8156, 8351, 8545, 8739, 8932, 9126, 9319,
    9511, 9703, 9895, 10087, 10278, 10469, 10659, 10849,
    11038, 11227, 11416, 11604, 11792, 11980, 12166, 12353,
    12539, 12724, 12909, 13094, 13278, 13462, 13645, 13827,
    14009, 14191, 14372, 14552, 14732, 14911, 15090, 15268,
    15446, 15623, 15799, 15975, 16150, 16325, 16499, 16672,
    16845, 17017, 17189, 17360, 17530, 17699, 17868, 18036,
    18204, 18371, 18537, 18702, 18867, 19031, 19194, 19357,
    19519, 19680, 19840, 20000, 20159, 20317, 20474, 20631,
    20787, 20942, 21096, 21249, 21402, 21554, 21705, 21855,
    22004, 22153, 22301, 22448, 22594, 22739, 22883, 23027,
    23169, 23311, 23452, 23592, 23731, 23869, 24006, 24143,
    24278, 24413, 24546, 24679, 24811, 24942, 25072, 25201,
    25329, 25456, 25582, 25707, 25831, 25954, 26077, 26198,
    26318, 26437, 26556, 26673, 26789, 26905, 27019, 27132,
    27244, 27355, 27466, 27575, 27683, 27790, 27896, 28001,
    28105, 28208, 28309, 28410, 28510, 28608, 28706, 28802,
    28897, 28992, 29085, 29177, 29268, 29358, 29446, 29534,
    29621, 29706, 29790, 29873, 29955, 30036, 30116, 30195,
    30272, 30349, 30424, 30498, 30571, 30643, 30713, 30783,
    30851, 30918, 30984, 31049, 31113, 31175, 31236, 31297,
    31356, 31413, 31470, 31525, 31580, 31633, 31684, 31735,
    31785, 31833, 31880, 31926, 31970, 32014, 32056, 32097,
    32137, 32176, 32213, 32249, 32284, 32318, 32350, 32382,
    32412, 32441, 32468, 32495, 32520, 32544, 32567, 32588,
    32609, 32628, 32646, 32662, 32678, 32692, 32705, 32717,
    32727, 32736, 32744, 32751, 32757, 32761, 32764, 32766,
    32767
}

regfile FFT_REG_SIMD 128 4 fftsv

function [31:0] FFT_VAR_SHIFT([31:0] data, [3:0] sh)
{
    assign FFT_VAR_SHIFT = TIEmux(sh,
            data[31:0],
            {data[30:0], 1'b0},
            {data[29:0], 2'b0},
            {data[28:0], 3'b0},
            {data[27:0], 4'b0},
            {data[26:0], 5'b0},
            {data[25:0], 6'b0},
            {data[24:0], 7'b0},
            {data[23:0], 8'b0},
            {data[22:0], 9'b0},
            {data[21:0], 10'b0},
            {data[20:0], 11'b0},
            {data[19:0], 12'b0},
            {data[18:0], 13'b0},
            {data[17:0], 14'b0},
            {data[16:0], 15'b0});
}

function [31:0] ADD32([31:0] a, [15:0] b) slot_shared
{
    assign ADD32 = TIEadd(a, b, 1'b0);
}

operation FFT_SIMD_LOAD {in AR *base, in AR offset, out FFT_REG_SIMD data} {out VAddr, in MemDataIn128}
{
    assign VAddr = ADD32(base, {offset[30:0], 1'b0});

    wire [15:0] o1 = MemDataIn128[15:0];
    wire [15:0] o2 = MemDataIn128[31:16];
    wire [15:0] o3 = MemDataIn128[47:32];
    wire [15:0] o4 = MemDataIn128[63:48];
    wire [15:0] o5 = MemDataIn128[79:64];
    wire [15:0] o6 = MemDataIn128[95:80];
    wire [15:0] o7 = MemDataIn128[111:96];
    wire [15:0] o8 = MemDataIn128[127:112];

    assign data = {o1, o2, o3, o4, o5, o6, o7, o8 };
```

```
}

operation FFT_SIMD_STORE {in AR *base, in AR offset, in FFT_REG_SIMD data} {out VAddr, out MemDataOut128}
{
    assign VAddr = ADD32(base, {offset[30:0], 1'b0});

    wire [15:0] o1 = data[15:0];
    wire [15:0] o2 = data[31:16];
    wire [15:0] o3 = data[47:32];
    wire [15:0] o4 = data[63:48];
    wire [15:0] o5 = data[79:64];
    wire [15:0] o6 = data[95:80];
    wire [15:0] o7 = data[111:96];
    wire [15:0] o8 = data[127:112];

    assign MemDataOut128 = {o1, o2, o3, o4, o5, o6, o7, o8 };
}

immediate_range offset 0 1 1

operation FFT_SIMD_LOAD_SHUFFLE {in AR *base, in AR offset, inout FFT_REG_SIMD d1, inout FFT_REG_SIMD d2,
in offset high} {out VAddr, in MemDataIn128}
{
    assign VAddr = ADD32(base, {offset[30:0], 1'b0});

    wire [15:0] o1 = MemDataIn128[15:0];
    wire [15:0] o2 = MemDataIn128[31:16];
    wire [15:0] o3 = MemDataIn128[47:32];
    wire [15:0] o4 = MemDataIn128[63:48];
    wire [15:0] o5 = MemDataIn128[79:64];
    wire [15:0] o6 = MemDataIn128[95:80];
    wire [15:0] o7 = MemDataIn128[111:96];
    wire [15:0] o8 = MemDataIn128[127:112];

    assign d1 = TIEmux(high[0], {o1, o2, o3, o4, d1[63:0]}, {d1[127:64], o1, o2, o3, o4});
    assign d2 = TIEmux(high[0], {o5, o6, o7, o8, d2[63:0]}, {d2[127:64], o5, o6, o7, o8});
}

operation FFT_SIMD_STORE_SHUFFLE {in AR *base, in AR offset, in FFT_REG_SIMD d1, in FFT_REG_SIMD d2, in
offset high} {out VAddr, out MemDataOut128}
{
    assign VAddr = ADD32(base, {offset[30:0], 1'b0});

    wire [15:0] o1 = TIEmux(high[0], d1[79:64],   d1[15:0]);
    wire [15:0] o2 = TIEmux(high[0], d1[95:80],   d1[31:16]);
    wire [15:0] o3 = TIEmux(high[0], d1[111:96],  d1[47:32]);
    wire [15:0] o4 = TIEmux(high[0], d1[127:112], d1[63:48]);
    wire [15:0] o5 = TIEmux(high[0], d2[79:64],   d2[15:0]);
    wire [15:0] o6 = TIEmux(high[0], d2[95:80],   d2[31:16]);
    wire [15:0] o7 = TIEmux(high[0], d2[111:96],  d2[47:32]);
    wire [15:0] o8 = TIEmux(high[0], d2[127:112], d2[63:48]);

    assign MemDataOut128 = {o5, o6, o7, o8, o1, o2, o3, o4 };
}

operation FFT_SHIFT_CHECK {in AR *addr, in AR offset, out AR needs_shift} {out VAddr, in MemDataIn128}
{
    assign VAddr = ADD32(addr, offset[31:1]);

    wire [15:0] o1 = MemDataIn128[15:0];
    wire [15:0] o2 = MemDataIn128[31:16];
    wire [15:0] o3 = MemDataIn128[47:32];
    wire [15:0] o4 = MemDataIn128[63:48];
    wire [15:0] o5 = MemDataIn128[79:64];
    wire [15:0] o6 = MemDataIn128[95:80];
    wire [15:0] o7 = MemDataIn128[111:96];
    wire [15:0] o8 = MemDataIn128[127:112];

    wire s1 = (!o1[15] && o1[14]) || (o1[15] && (!o1[14] || o1[13:0] == 14'b0));
    wire s2 = (!o2[15] && o2[14]) || (o2[15] && (!o2[14] || o2[13:0] == 14'b0));
    wire s3 = (!o3[15] && o3[14]) || (o3[15] && (!o3[14] || o3[13:0] == 14'b0));
    wire s4 = (!o4[15] && o4[14]) || (o4[15] && (!o4[14] || o4[13:0] == 14'b0));
    wire s5 = (!o5[15] && o5[14]) || (o5[15] && (!o5[14] || o5[13:0] == 14'b0));
    wire s6 = (!o6[15] && o6[14]) || (o6[15] && (!o6[14] || o6[13:0] == 14'b0));
    wire s7 = (!o7[15] && o7[14]) || (o7[15] && (!o7[14] || o7[13:0] == 14'b0));
    wire s8 = (!o8[15] && o8[14]) || (o8[15] && (!o8[14] || o8[13:0] == 14'b0));

    assign needs_shift = {31'b0, s1 || s2 || s3 || s4 || s5 || s6 || s7 || s8 };
}

operation FFT_BIT_REVERSE {in AR m, in AR mm, out AR mr} {}
```

```
{
    assign mr = {
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1], m[2]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1], m[2], m[3]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1], m[2], m[3], m[4]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1], m[2], m[3], m[4], m[5]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1], m[2], m[3], m[4], m[5], m[6]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1], m[2], m[3], m[4], m[5], m[6], m[7]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1], m[2], m[3], m[4], m[5], m[6], m[7], m[8]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1], m[2], m[3], m[4], m[5], m[6], m[7], m[8], m[9]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1], m[2], m[3], m[4], m[5], m[6], m[7], m[8], m[9], m[10]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, 1'b0, m[0], m[1], m[2], m[3], m[4], m[5], m[6], m[7], m[8], m[9], m[10],m[11]),
       TIEmux(mm[3:0], 1'b0, 1'b0, 1'b0, m[0], m[1], m[2], m[3], m[4], m[5], m[6], m[7], m[8], m[9], m[10],m[11],m[12]),
       TIEmux(mm[3:0], 1'b0, 1'b0, m[0], m[1], m[2], m[3], m[4], m[5], m[6], m[7], m[8], m[9], m[10],m[11],m[12],m[13]),
       TIEmux(mm[3:0], 1'b0, m[0], m[1], m[2], m[3], m[4], m[5], m[6], m[7], m[8], m[9], m[10],m[11],m[12],m[13],m[14])
    };
}

function [15:0] SIN ([9:0] idx, [0:0] not_negate)
{
    //wire [7:0] neg_idx = TIEadd(0, ~idx[7:0], 1'b1);
    wire [8:0] x2 =  TIEadd(8'h0, ~idx[7:0], 1'b1); // 512 - x
    wire [8:0] sin_idx = TIEmux(idx[8:0] == 256 || !idx[8:8], // if x mod 512 <= 256
            // then
            x2[8:0],
            // else
            idx[8:0]
            );
    wire [15:0] value = SIN_WAVE[sin_idx];

    assign SIN = TIEmux(((idx[9:0] == 512) || !idx[9:9] ) && not_negate,
            TIEadd(16'h0, ~value[15:0], 1'b1),
            value[15:0]);
}

function [31:0] FFT_TWIDDLE ([31:0] i, [1:0] x, [4:0] k, [0:0] shift, [0:0] inverse)
{
    wire [31:0] inc = TIEadd(i, x, 1'b0);
    wire [31:0] j = FFT_VAR_SHIFT(inc, k);

    // 256 = N_WAVE / 4
    wire [9:0] cos_idx = TIEadd(j, 256, 1'b0);
    // 1023 -> 0
    wire [15:0] sin = SIN(j, inverse);
    wire [15:0] wr1 = SIN(cos_idx, 1);

    assign FFT_TWIDDLE = {
        TIEmux(shift, wr1, {wr1[15], wr1[15:1]}),
        TIEmux(shift, sin, {sin[15], sin[15:1]})
    };
}

function [31:0] FFT_TWIDDLE1 ([31:0] i, [1:0] x, [4:0] k, [0:0] shift, [0:0] inverse) slot_shared {
    assign FFT_TWIDDLE1 = FFT_TWIDDLE(i, x, k, shift, inverse);
}
function [31:0] FFT_TWIDDLE2 ([31:0] i, [1:0] x, [4:0] k, [0:0] shift, [0:0] inverse) slot_shared {
    assign FFT_TWIDDLE2 = FFT_TWIDDLE(i, x, k, shift, inverse);
}

function [63:0] FFT_BUTTERFLY ([63:0] data, [15:0] wr, [15:0] wi, [0:0] shift) {

    // operands real parts
    wire [15:0] r1 = data[63:48];
    wire [15:0] r2 = data[47:32];

    // operands imaginary parts
    wire [15:0] i1 = data[31:16];
    wire [15:0] i2 = data[15:0];

    // odd real part
    wire [31:0] oddr1 = TIEmul(wr, r2, 1'b1);
    wire [31:0] oddr2 = TIEmul(wi, i2, 1'b1);
    wire [15:0] oddr = TIEaddn(oddr1[30:15], ~oddr2[30:15], 16'b1);

    // odd imaginary part
    wire [31:0] oddi1 = TIEmul(wr, i2, 1'b1);
    wire [31:0] oddi2 = TIEmul(wi, r2, 1'b1);
    wire [15:0] oddi = TIEadd(oddi1[30:15], oddi2[30:15], 1'b0);

    // even parts
    wire [15:0] evenr = TIEmux(shift[0], r1, {r1[15], r1[15:1]});
```

```
    wire [15:0] eveni = TIEmux(shift[0], i1, {i1[15], i1[15:1]});

    // final result
    wire [15:0] resr1 = TIEadd(evenr,  oddr, 1'b0);
    wire [15:0] resr2 = TIEadd(evenr, ~oddr, 1'b1);
    wire [15:0] resi1 = TIEadd(eveni,  oddi, 1'b0);
    wire [15:0] resi2 = TIEadd(eveni, ~oddi, 1'b1);

    assign FFT_BUTTERFLY = { resr1, resr2, resi1, resi2 };
}
function [63:0] FFT_BUTTERFLY1 ([63:0] data, [15:0] wr, [15:0] wi, [0:0] shift) slot_shared {
    assign FFT_BUTTERFLY1 = FFT_BUTTERFLY(data, wr, wi, shift);
}
function [63:0] FFT_BUTTERFLY2 ([63:0] data, [15:0] wr, [15:0] wi, [0:0] shift) slot_shared {
    assign FFT_BUTTERFLY2 = FFT_BUTTERFLY(data, wr, wi, shift);
}

// first stage: four non-interleaved butterflies
operation FFT_SIMD_FIRST {inout FFT_REG_SIMD fr, inout FFT_REG_SIMD fi, in BR shift} {}
{
    // 1
    wire [15:0] wr = TIEmux(shift, 16'h7fff, 16'h3fff);
    wire [15:0] wi = 16'b0;

    wire [63:0] res1 = FFT_BUTTERFLY1({fr[127:96], fi[127:96]}, wr, wi, shift);
    wire [63:0] res2 = FFT_BUTTERFLY1({fr[95:64],  fi[95:64]},  wr, wi, shift);
    wire [63:0] res3 = FFT_BUTTERFLY2({fr[63:32],  fi[63:32]},  wr, wi, shift);
    wire [63:0] res4 = FFT_BUTTERFLY2({fr[31:0],   fi[31:0]},   wr, wi, shift);

    assign fr = { res1[63:32], res2[63:32], res3[63:32], res4[63:32] };
    assign fi = { res1[31:0],  res2[31:0],  res3[31:0],  res4[31:0] };
}

schedule FFT_SIMD_FIRST_schedule {FFT_SIMD_FIRST}
{
    def fr 2;
    def fi 2;
}

// second stage: four butterflies, two interleaved at a time
operation FFT_SIMD_SECOND {inout FFT_REG_SIMD fr, inout FFT_REG_SIMD fi, in BR shift, in BR inverse} {}
{
    // 1
    wire [15:0] wr1 = TIEmux(shift, 16'h7fff, 16'h3fff);
    wire [15:0] wi1 = 16'b0;

    // -j
    wire [15:0] wr2 = 16'b0;
    wire [15:0] wi2 = TIEmux({inverse, shift}, 16'h8001, 16'hc000, 16'h7fff, 16'h3fff);

    wire [63:0] res1 = FFT_BUTTERFLY1({fr[127:112], fr[95:80], fi[127:112], fi[95:80]}, wr1, wi1, shift);
    wire [63:0] res2 = FFT_BUTTERFLY1({fr[111:96],  fr[79:64], fi[111:96],  fi[79:64]}, wr2, wi2, shift);
    wire [63:0] res3 = FFT_BUTTERFLY2({fr[63:48],   fr[31:16], fi[63:48],   fi[31:16]}, wr1, wi1, shift);
    wire [63:0] res4 = FFT_BUTTERFLY2({fr[47:32],   fr[15:0],  fi[47:32],   fi[15:0]},  wr2, wi2, shift);

    assign fr = { res1[63:48], res2[63:48], res1[47:32], res2[47:32], res3[63:48], res4[63:48], res3[47:32], res4[47:32] };
    assign fi = { res1[31:16], res2[31:16], res1[15:0],  res2[15:0],  res3[31:16], res4[31:16], res3[15:0],  res4[15:0] };
}

schedule FFT_SIMD_SECOND_schedule {FFT_SIMD_SECOND}
{
    def fr 2;
    def fi 2;
}

function [31:0] FFT_INC_SHIFT ([31:0] i, [1:0] x, [4:0] k)
{
    wire [31:0] inc = TIEadd(i, x, 1'b0);
    assign FFT_INC_SHIFT = FFT_VAR_SHIFT(inc, k);
}

state FFT_SIMD_K 5 add_read_write

// third stage: four interleaved butterflies
operation FFT_SIMD_THIRD {in AR i, inout FFT_REG_SIMD fr, inout FFT_REG_SIMD fi, in BR shift, in BR inverse} {in FFT_SIMD_K}
{
    wire [4:0] k = FFT_SIMD_K;

    wire [31:0] tw1 = FFT_TWIDDLE1(i, 0, k, shift, inverse);
    wire [15:0] wr1 = tw1[31:16];
    wire [15:0] wi1 = tw1[15:0];

```

```
    wire [31:0] tw2 = FFT_TWIDDLE1(i, 1, k, shift, inverse);
    wire [15:0] wr2 = tw2[31:16];
    wire [15:0] wi2 = tw2[15:0];

    wire [31:0] tw3 = FFT_TWIDDLE2(i, 2, k, shift, inverse);
    wire [15:0] wr3 = tw3[31:16];
    wire [15:0] wi3 = tw3[15:0];

    wire [31:0] tw4 = FFT_TWIDDLE2(i, 3, k, shift, inverse);
    wire [15:0] wr4 = tw4[31:16];
    wire [15:0] wi4 = tw4[15:0];

    wire [63:0] res1 = FFT_BUTTERFLY1({fr[127:112], fr[63:48], fi[127:112], fi[63:48]}, wr1, wi1, shift);
    wire [63:0] res2 = FFT_BUTTERFLY1({fr[111:96],  fr[47:32], fi[111:96],  fi[47:32]}, wr2, wi2, shift);
    wire [63:0] res3 = FFT_BUTTERFLY2({fr[95:80],   fr[31:16], fi[95:80],   fi[31:16]}, wr3, wi3, shift);
    wire [63:0] res4 = FFT_BUTTERFLY2({fr[79:64],   fr[15:0],  fi[79:64],   fi[15:0]},  wr4, wi4, shift);

    assign fr = { res1[63:48], res2[63:48], res3[63:48], res4[63:48], res1[47:32], res2[47:32], res3[47:32], res4[47:32] };
    assign fi = { res1[31:16], res2[31:16], res3[31:16], res4[31:16], res1[15:0],  res2[15:0],  res3[15:0],  res4[15:0] };
}

schedule FFT_SIMD_THIRD_schedule {FFT_SIMD_THIRD}
{
    def fr 2;
    def fi 2;
}


//------------------------------------------------------------------------

ctype FFT_REG_SIMD 128 128 FFT_REG_SIMD default

immediate_range st.FFT_REG_SIMD_immed2 -256 240 16
immediate_range ld.FFT_REG_SIMD_immed2 -256 240 16
format flix64_0 64 { flix64_0_slot0, flix64_0_slot1, flix64_0_slot2  }

slot_opcodes flix64_0_slot0 {
    FFT_SHIFT_CHECK, OR, FFT_SIMD_LOAD, FFT_SIMD_STORE, FFT_SIMD_LOAD_SHUFFLE, FFT_SIMD_STORE_SHUFFLE,
FFT_SIMD_THIRD, DIF_FFT_SIMD_THIRD,
    // XTENSA ISA OPS - usable without area overead
    MOVI, SSL, NOP, SLL, BGE, J, ADDI, S32I, ADDX2, L16SI, S16I, BLT, BGEI, ORBC, ANDBC, BBCI, BEQZ.N, ORB,
ADDI.N, MOVGEZ, SRAI, OR, BEQZ, L32I.N, BEQI, BNEI, MOVI.N, BLTI, SLLI, ADD.N, MOV.N, ADD, L32I, BNE
}
slot_opcodes flix64_0_slot1 { SSL, MOVI, ADDI, J, ADDX2, NOP, MOV.N, ADD.N }
slot_opcodes flix64_0_slot2 { NOP, S32I, ADDI, ADDX2, L16SI, S16I, MOVI, FFT_SHIFT_CHECK, FFT_SIMD_LOAD,
FFT_SIMD_STORE, FFT_SIMD_LOAD_SHUFFLE, FFT_SIMD_STORE_SHUFFLE }
```

Questions:

Which acceleration technique(s) has/have been used and why?

- FLIX instructions:
  - better utilize hardware by executing multiple independent instructions in parallel
  - (parallel load / store in particular, see SIMD)
- SIMD:
  - load, calculate and store multiple values at once in a single instruction
- Pipelining:
  - reduce critical path to increase maximum clock frequency
- Hardware Lookup Table for Sine function:
  - speed up lookup

Which part of the FFT algorithm did you accelerate and why?

- Bit reversal:
  - is achieved in a single instruction, without a loop like in the C implementation
  - swapping of values:
    - store operations are not parallelized via FLIX by the compiler due to possible memory overlapping
    - parallelization enforced with inline assembler
- Comparison for value ranges:
  - parallel comparison (greater than or less than a given value) in hardware
  - SIMD by processing 8 input values in one instruction
- Load / Store:
  - to load and store 128 bits at once, extra instructions are required
  - even loading and storing of 256 bits in one instruction is achieved by using the same load / store instruction in two FLIX slots
  - address calculation (base + offset) realized in hardware
- FFT calculation:
  - twiddle factor and butterfly calculations are implemented as TIE instructions
  - SIMD by processing 8 input values (4 butterflies) at once

What is the impact on execution time (speedup)?

Cycles used for FFT and inverse FFT combined for:

- unoptimized C implementation (-O3):
  - M=3 (N=8): **3 512**
  - M=8 (N=256): **230 627**
  - M=10 (N=1024): **1 114 289**
- our optimized implementation with TIE instructions, SIMD, pipelining and FLIX (-O2, compiled with feedback optimization):
  - M=3 (N=8): **327** (Speedup: **10.7**)
  - M=8 (N=256): **7 612** (Speedup: **30.3**)
  - M=10 (N=1024): **32 050** (Speedup: **34.8**)

What is the impact on the Hardware (Area)?

Our TIE implementation required about **90 000** additional gates for either DIT or DIF. When using both DIT and DIF at the same time, only about 108 000 additional gates are required because of shared functionalities.

What is the impact on the software (new instructions, modified source code, compiler intrinsics)?

By implementing parts of the algorithm in TIE, we introduced new instructions. Also, we had to restructure the core loop of the algorithm to fit the memory layout required by the parallelized load / store instructions.

Detailed Report (min. 1 page, max. 3 pages): *Step-by-step description of each optimization step that has been tried (even if no performance increase could be achieved):*

- *What software optimization or hardware acceleration strategies you tried?*
- *Refer your explanations to the C-/TIE-code making use of the lines numbers*
- *How did the performance increase (or decrease)?*
- *What are the costs for the performance increase?*
- *Make a trade off performance vs. increased costs.*

Conclude with an overall summary: Which combination of optimization and hardware acceleration strategies do you suggest, i.e., yields the best performance results or speed/area tradeoff? Summarize the experiences you gained with this task.

Note: For the following descriptions, we used M=8 (N=256) with –O3 but without feedback optimization to measure the spent cycles. Measured is the FFT as well as the inverse FFT combined.

At first we ran the profiling tool from within Xtensa Xplorer and found that the innermost loop core required the most cycles. Therefore we introduced a new TIE instruction that does the actual butterfly calculation. This early version required 109 832 cycles (speedup 2.1) and 16 444 gates.

The second hot-spot we optimized was the bit reverse at the beginning of the algorithm. We implemented the bit reversal itself as a single TIE instruction. This had the advantage that it used merely one cycle instead of using a loop. Also we implemented the calculation of the twiddle factors in another TIE instruction by using a hardware lookup table for the required sine values. We achieved a total of 89 107 cycles (speedup 2.6) with using 32 793 gates.

Next we wanted to optimize loop headers throughout the C implementation by introducing corresponding TIE instructions. However the achieved speedup was merely noticeable or the program was even slower. This might be because the optimizations the C compiler tries to do are impaired by the newly introduced instructions it can't comprehend. Therefore, we abandoned this approach.

Another hot-spot was the check if any of the input values are within the specified range. We optimized this step by doing both comparisons (greater than or less than a given value) in another TIE instruction (dit.tie: 132-155). Also we introduced auto-generated FLIX optimization for the first time. This version finished after 77 361 cycles (speedup 3.0) but required 82 932 gates which is a tremendous increase for such a small gain in speedup.

However, after several manual changes of the FLIX slot assignment (dit.tie: 359-365) we managed to reduce the number of gates to 45 079 while decreasing the number of required cycles to 60 309 (speedup 3.8).

For the next step, we tried to use the TIEmac function instead of TIEmul and TIEadd in our TIE instructions. This resulted only in slightly lower gate usage and no performance gain but produced results that differed by a small factor from the C implementation, which we wanted to avoid.

Next we introduced SIMD operation for the instruction that checks for the value ranges. This reduced the needed cycles to 48 327 (speedup 4.8) and required gates to 59 697.

After that we moved the twiddle factor calculation inside the butterfly calculation which resulted in a longer computation time, requiring 53594 cycles, but also reduced the number of used gates to 51597.

Our next big improvement was the introduction of 4-fold SIMD processing of the actual butterfly calculation as well as the use of some small snippets of inline assembly code to force the compiler to do parallel store operations (see dit-flix.h).
For this optimization to work, we needed to implement separate handling of the first three stages of the FFT (dit.c: 62-112). Also we implemented custom load and store operations (dit.tie: 65-95) to be able to transfer 128 bits of memory within one instruction (and 256 bits within one cycle when using FLIX).
For stage four and beyond we implemented another load and store operation (dit.tie: 99-130) so that we could fetch enough data (in the right order) for two 4-fold butterflies within 2 cycles. By outsourcing the butterfly calculation into shared functions (dit.tie: 251-256) we could share those four required hardware instances across the separate instructions for the different stages. For stage one and two, we hardcoded the required twiddle factors. For stage three and beyond, four instances of the twiddle factor calculation were needed.
Altogether this optimization resulted in huge performance gain as the number of spent cycles dropped to 9 640 (speedup 23.9). However, this came with a trade-off regarding hardware area usage as our TIE implementation now required 175 642 gates.
As a side effect of using 4-fold SIMD our optimized algorithm requires a problem size of at least M=3 (N=8) to work properly. Because of the generally limited size of the sine table the maximum problem size is M=10 (N=1024).

Several smaller optimizations followed, mainly aimed at reducing the number of required gates.

We managed to reduce the size of the hardware lookup table for the sine values to a quarter of the original size by using the symmetry of the sine function (dit.tie: 178-193). Together with code cleanup by removing now unneeded functions we reduced the gate count to 118 511. (9284 cycles)

Finally, by using the schedule option (e.g. dit.tie: 344-348), we pipelined the butterfly calculation instructions across two clock cycles, slightly reducing the performance, but also basically cutting the critical path in half, possibly allowing for higher clock rates and reducing the required gate count. We ended up using a total of 90 503 gates.

Surprisingly, using the –O2 instead of the –O3 compiler option, we achieved a better performance. Together with enabled software feedback optimization, we managed to reduce the number of required clock cycles to 7612 (speedup 30.3). An even higher speedup was achieved for M=10 (N=1024); see above.

Actually, using –O3 seems to break our current implementation as we apparently get random output values. This might be because of some optimizations the compiler tries to do on its own which interfere with our inline assembler instructions although we correctly annotated which variables we need to use and which we modify in the assembly code.

To compare DIT and DIF we also implemented DIF in C code. In contrast to DIT, we had to put the bit reverse at the end of the algorithm, reverse the loop direction and implement the modified butterfly. For optimization we reused most of the TIE instructions we implemented for DIT, we only had to rewrite the butterfly calculation for DIF.
For the difference between those two implementations we only noticed that the DIF implementation in C took more cycles to complete. When using the TIE implementation, execution time and hardware area usage were roughly the same as for DIT.

Summary:

- Although SIMD increases hardware area, it also improves performance by an even higher number, especially for bigger problem sizes.
- FLIX complicates instruction decoding and therefore also uses more area by multiplying the hardware components for instructions, but can increase the performance by parallelizing instructions in many places. Not only for custom TIE instructions. Also FLIX is the only option to use both load/store-units in one clock cycle.
- Pipelining delays single instructions resulting in lower overall performance when following instructions depend on previous results. On the other hand, it cuts the critical path and permits a higher clock rate. By sharing hardware resources between different pipeline stages, one can further reduce the hardware usage.

In this tasked we learned a lot about the interaction between hardware and software; how much performance certain hardware implementations yield and with which area trade-off they come. We also learned how to use profiling to detect hot-spots in given applications. For the first time we used a VLIW / FLIX platform and wrote inline assembly code.
After all we gained much experience and also had fun trying to optimize almost every bit of code to a maximum while still keeping the costs in mind.