TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Electrical Engineering and Computer Engineering Institute of Communication Technology

Vodafone Chair of Mobile Communications Systems

Project Report: HW/SW Co-Design Lab

SS 2024

Eduardo Schwarz Danni

eduardo.schwarz_danni@tu-dresden.de

Nick Bracher

nick.bracher@tu-dresden.de

15.08.2024

Supervisor

Viktor Razilov

1 Introduction

The HW/SW Co-Design 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 in fig. 1). 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.

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

1. Implementation of a finitie impulse response (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 fast Fourier transform (FFT)/inverse fast Fourier transform (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.

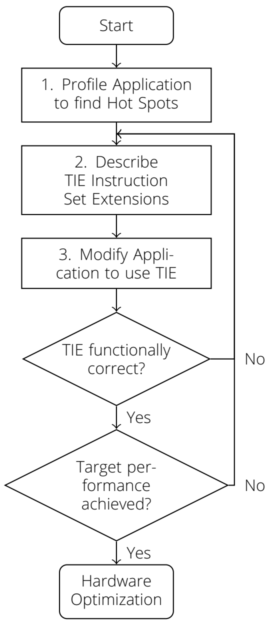
Start
1. Profile Application to find Hot Spots
2. Describe TIE Instruction Set Extensions
3. Modify Application to use TIE
TIE functionally correct?
No
Yes
Target performance achieved?
No
Yes
Hardware Optimization

Figure 1: Similar to Tensilica Instruction Extension (TIE) Language, User's Guide. 02/2014, p. 4.

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:
`https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/21521498113`

2 Source Code

2.1 Source code C files

fft.h

```c
#ifndef FFT_H
#define FFT_H
#include <stdint.h>
#include <stdbool.h>
#include <math.h>
#include <xtensa/tie/fft_tie.h>

/* FIX_MPY() - fixed-point multiplication macro.
 This macro is a statement, not an expression (uses asm).
 BEWARE: make sure _DX is not clobbered by evaluating (A) or DEST.
 args are all of type fixed.
 Scaling ensures that 32767*32767 = 32767. */
#define FIX_MPY(DEST,A,B)       DEST = ((long)(A) * (long)(B))>>15

#define N_WAVE          1024    /* dimension of Sinewave[] */
#define LOG2_N_WAVE     10      /* log2(N_WAVE) */

typedef int16_t fixed;
extern fixed Sinewave[N_WAVE];

//function prototypes
fixed fix_mpy(fixed a, fixed b);
int fix_fft(fixed *fr, fixed *fi, int m, int inverse);
int fix_fft_dif(fixed *fr, fixed *fi, int m, int inverse);

typedef struct cplx {
    fixed R;
    fixed I;
} cplx;

int fft_adv_dif(cplx*__restrict f, int m,
        const cplx* __restrict coeffs);

```

```
int fft_adv_dif(cplx*__restrict f, int m,
        const cplx* __restrict coeffs);

int fft_adv_dit(cplx*__restrict f, int m,
        const cplx* __restrict coeffs);

int fft_adv_dit_inv(cplx*__restrict f, int m,
        const cplx* __restrict coeffs);

#endif  //FFT_H
```

main.c

```
//main.c

#include        "fft.h"
#include        <stdio.h>
#include        <math.h>

//change the type to change the test to be done
#define FFT_Type    5

#define M       3

//number of points
#define N       (1<<M)

fixed real[N], imag[N], real_fix[N], imag_fix[N], real_mul[N], imag_mul[N];
//for Backwward propagation
static fixed bwd_real_mul[N], bwd_imag_mul[N], bwd_real_fix[N], bwd_imag_fix
    [N];
static cplx f[N], bwd_f[N];
bool equal;

static cplx fwd_coeffs[N / 2];
static cplx bwd_coeffs[N / 2];

int Test_fft_ref() { //Test for reference FFT and IFFT

    fix_fft(real_fix, imag_fix, M, 0);
    memcpy(bwd_real_fix, real_fix, sizeof(real_fix));
    memcpy(bwd_imag_fix, imag_fix, sizeof(imag_fix));
    fix_fft(bwd_real_fix, bwd_imag_fix, M, 1);

}

int Test_fft_adv_dit() {

    fft_adv_dit(f, M, fwd_coeffs);
    memcpy(bwd_f, f, sizeof(f));
    fft_adv_dit_inv(bwd_f, M, bwd_coeffs);
```

```
}

int Test_only_fft_adv_dit() {
    fix_fft(real_fix, imag_fix, M, 0);
    fft_adv_dit(f, M, fwd_coeffs);
}

int Test_only_fft_adv_dit_inv() {
    fix_fft(real_fix, imag_fix, M, 1);
    fft_adv_dit_inv(f, M, bwd_coeffs);
}

int Test_only_fft_adv_dif() {
    fft_adv_dif(f, M, fwd_coeffs);
    fix_fft_dif(real_fix, imag_fix, M, 0);
}

int main() {
    int i;
    //precalculation of Coefficients
    GenerateCoefficients(fwd_coeffs, N / 2, false);
    GenerateCoefficients(bwd_coeffs, N / 2, true);
    //Generating the input data
    for (i = 0; i < N; i++) {
        real[i] = 1000 * cos(i * 2 * 3.1415926535 / N);
        imag[i] = 0;
        real_fix[i] = real[i];
        imag_fix[i] = imag[i];
        real_mul[i] = real[i];
        imag_mul[i] = imag[i];

        f[i].R = real[i];
        f[i].I = imag[i];
    }


     printf("\nInput Data\n");
     for (i = 0; i < N; i++) {
     printf("%d: %d, %d\n", i, real[i], imag[i]);
     }
     printf("\nInput Data Complex\n");
     for (i = 0; i < N; i++) {
     printf("%d: %d, %d\n", i, f[i].R, f[i].I);
     }


#if (FFT_Type==0)//Test reference C code
    fix_fft(real, imag, M, 0);

    printf("\nFFT pure C\n");
    for (i = 0; i < N; i++) {
```

```c
        printf("%d: %d, %d\n", i, real[i], imag[i]);
    }

#elif (FFT_Type==1)//Test FFT2 and reference
    fix_fft(real_fix, imag_fix, M, 0);

    printf("\nFFT pure C\n");
    for (i = 0; i < N; i++) {
        printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
    }

    fft_FFT2(real_mul, imag_mul, M, 0);

    printf("\nFFT TIE Node\n");
    for (i = 0; i < N; i++) {
        printf("%d: %d, %d\n", i, real_mul[i], imag_mul[i]);
    }

#elif (FFT_Type==2)//Test DIT FFT and IFFT
    Test_fft_ref();
    Test_fft_adv_dit();

    equal = true;
    for (i = 0; i < N; i++) { //Compare the calculated values and generate
       print
        if(f[i].R != real_fix[i] || f[i].I != imag_fix[i])
        {
            equal = false;
            printf("\nCorrect C value\n");
            printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
            printf("Wrong TIE value\n");
            printf("%d: %d, %d\n", i, f[i].R, f[i].I);
        }
        if(bwd_f[i].R != bwd_real_fix[i] || bwd_f[i].I != bwd_imag_fix[i])
        {
            equal = false;
            printf("\nCorrect C value\n");
            printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
            printf("Wrong TIE value\n");
            printf("%d: %d, %d\n", i, bwd_f[i].R, bwd_f[i].I);
        }
    }
    if(equal) {
        printf("FFT Pass");
    }
    else {
        printf("FFT Fail");
    }

#elif(FFT_Type == 3)//Test DIT FFT
    Test_only_fft_adv_dit();

    equal = true;
    for (i = 0; i < N; i++) {
        if(f[i].R != real_fix[i] || f[i].I != imag_fix[i])
        {
            equal = false;
            printf("\nCorrect C value\n");
            printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
            printf("Wrong TIE value\n");
            printf("%d: %d, %d\n", i, f[i].R, f[i].I);
        }
    }
    if(equal) {
        printf("FFT Pass");
    }
    else {
        printf("FFT Fail");
    }

#elif(FFT_Type == 4)////Test DIT IFFT
    Test_only_fft_adv_dit_inv();
    equal = true;
    for (i = 0; i < N; i++) {
        if (f[i].R != real_fix[i] || f[i].I != imag_fix[i]) {
            equal = false;
            printf("\nCorrect C value\n");
            printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
            printf("Wrong TIE value\n");
            printf("%d: %d, %d\n", i, f[i].R, f[i].I);
        }
    }
    if (equal) {
        printf("FFT Pass");
    } else {
        printf("FFT Fail");
    }

#elif(FFT_Type == 5) //Test DIF FFT
    Test_only_fft_adv_dif();
    equal = true;
    for (i = 0; i < N; i++) {
        if (f[i].R != real_fix[i] || f[i].I != imag_fix[i]) {
            equal = false;
            printf("\nCorrect C value\n");
            printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
            printf("Wrong TIE value\n");
            printf("%d: %d, %d\n", i, f[i].R, f[i].I);
        }
    }
    if (equal) {
        printf("FFT Pass");
```

```c
    } else {
        printf("FFT Fail");
    }

#endif

    return 0;
}
```

GenerateCoefficient.c

```c
    #include "fft.h"

static inline cplx
W(const uint32_t k, const uint32_t N) //define the angle
{
    const double angle = M_PI * k / N;
    const cplx res = { (fixed) (INT16_MAX * cos(angle)), (fixed) (-sin(angle
    ) * INT16_MAX) };
    return res;
}

void
GenerateCoefficients(cplx* __restrict out_coeff, const uint32_t N, bool
    inverse) //generate the coeffient before calculation
{
    const fixed factor = inverse ? -1 : 1;
    uint32_t i = 0;
    for (; i < N; ++i)
    {
        cplx val = W(i, N);
        val.I = (fixed) (val.I * factor);
        out_coeff[i] = val;
    }
}
```

fft_adv.c

```c
#include "fft.h"

static inline uint32_t BitReverse(uint32_t n) {
    n = (n & 0xffff0000) >> 16 | (n & 0x0000ffff) << 16;
    n = (n & 0xff00ff00) >> 8 | (n & 0x00ff00ff) << 8;
    n = (n & 0xf0f0f0f0) >> 4 | (n & 0x0f0f0f0f) << 4;
    n = (n & 0xcccccccc) >> 2 | (n & 0x33333333) << 2;
    n = (n & 0xaaaaaaaa) >> 1 | (n & 0x55555555) << 1;
    return n;
}

uint32_t Prepare_Data(cplx* __restrict const x, const uint32_t N,
        const uint32_t lg2_N, const uint32_t a_start) {
```

```
    // Decimate
    {
        const uint32_t shift_amount = 32 - lg2_N;
        uint32_t a = a_start;
        uint32_t c = 0;
        for (; a < N + a_start; ++a) {
            c = a;
            FFT_BIT_REVERSE(c);
            const uint32_t b = c >> shift_amount;
            const bool swap = a < b;
            if (swap) {
                const cplx tmp_a = x[a];
                x[a] = x[b];
                x[b] = tmp_a;
            }
        }
    }
    return 0;
}

uint32_t fft_exec_dit(const intptr_t f, const intptr_t w, const uint32_t n,
        const uint32_t lg2_n, int shift) {
    int i, done, l, even_i, step, debug_reg, odd_ix;
    done = 0;
    WUR_ptr_data(f);
    WUR_ptr_w(w);
    WUR_n(n);
    WUR_lg2_n(lg2_n);
    WUR_shift(shift);

    WUR_time_decimation(1);

    FFT_INIT();
    while (FFT_UPDATE() == 0) {

        FFT_LOAD_EVEN();
        FFT_LOAD_ODD();
        FFT_LOAD_W();

        FFT_LOAD_EVEN();
        FFT_LOAD_ODD();
        FFT_LOAD_W();

        FFT_LOAD_EVEN();
        FFT_LOAD_ODD();
        FFT_LOAD_W();

        FFT_LOAD_EVEN();
        FFT_LOAD_ODD();
        FFT_LOAD_W();

        FFT_8_FFT_DIT();

        FFT_STORE_EVEN();
        FFT_STORE_ODD();

        FFT_STORE_EVEN();
        FFT_STORE_ODD();

        FFT_STORE_EVEN();
        FFT_STORE_ODD();

        FFT_STORE_EVEN();
        FFT_STORE_ODD();

    }
}

uint32_t fft_exec_dif(const intptr_t f, const intptr_t w, const uint32_t n,
        const uint32_t lg2_n, int shift) {
    int i, done, l, even_i, step, debug_reg, odd_ix;
    done = 0;
    WUR_ptr_data(f);
    WUR_ptr_w(w);
    WUR_n(n);
    WUR_lg2_n(lg2_n);
    WUR_shift(shift);

    WUR_time_decimation(0);

    FFT_INIT();
    while (FFT_UPDATE() == 0) {

        FFT_LOAD_EVEN();
        FFT_LOAD_ODD();
        FFT_LOAD_W();

        FFT_LOAD_EVEN();
        FFT_LOAD_ODD();
        FFT_LOAD_W();

        FFT_LOAD_EVEN();
        FFT_LOAD_ODD();
        FFT_LOAD_W();

        FFT_LOAD_EVEN();
        FFT_LOAD_ODD();
        FFT_LOAD_W();

        FFT_8_FFT_DIF();

        FFT_STORE_EVEN();
        FFT_STORE_ODD();

        FFT_STORE_EVEN();
        FFT_STORE_ODD();

        FFT_STORE_EVEN();
        FFT_STORE_ODD();

        FFT_STORE_EVEN();
        FFT_STORE_ODD();

    }
}

int fft_adv_dit(cplx*__restrict f, int m,
        const cplx* __restrict coeffs) {
    int mr, nn, i, j, l, k, r, istep, n, scale, shift;
    //number of input data
    n = 1 << m;

    cplx q; //even input
    cplx t; //odd input
    cplx u; //even output
    cplx v; //odd output
    cplx w[n / 2]; //twiddle factor
    uint32_t lg2_n = m;

    if (n > N_WAVE)
        return -1;

    mr = 0;
    nn = n - 1;
    scale = 0;
    r = 0;

    /* decimation in time - re-order data */

    Prepare_Data(f, n, m, 0);
    /* 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;

    fft_exec_dit((intptr_t) f, (intptr_t) coeffs, n, lg2_n, shift);

    return scale;
}

int fft_adv_dit_inv(cplx*__restrict f, int m,
        const cplx* __restrict coeffs) {
```

```
    int mr, nn, i, j, l, k, r, istep, n, scale, shift;
    //number of input data
    n = 1 << m;

    cplx q; //even input
    cplx t; //odd input
    cplx u; //even output
    cplx v; //odd output
    cplx w[n / 2]; //twiddle factor
    uint32_t lg2_n = m;

    if (n > N_WAVE)
        return -1;

    mr = 0;
    nn = n - 1;
    scale = 0;
    r = 0;

    /* decimation in time - re-order data */

    Prepare_Data(f, n, m, 0);
        /* variable scaling, depending upon data */
        shift = 0;
        for (i = 0; i < n; ++i) {
            j = f[i].R;
            if (j < 0)
                j = -j;

            m = f[i].I;
            if (m < 0)
                m = -m;

            if (j > 16383 || m > 16383) {
                shift = 1;
                break;
            }
        }
        if (shift)
            ++scale;
    fft_exec_dit((intptr_t) f, (intptr_t) coeffs, n, lg2_n, shift);
    return scale;
}

int fft_adv_dif(cplx*__restrict f, int m,
        const cplx* __restrict coeffs) {
    int mr, nn, i, j, l, k, r, istep, n, scale, shift;
    //number of input data
    n = 1 << m;

    cplx q; //even input
    cplx t; //odd input
    cplx u; //even output
    cplx v; //odd output
    cplx w[n / 2]; //twiddle factor
    uint32_t lg2_n = m;

    if (n > N_WAVE)
        return -1;

    mr = 0;
    nn = n - 1;
    scale = 0;
    r = 0;

    /* decimation in time - re-order data */
    /* 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;
    fft_exec_dif((intptr_t) f, (intptr_t) coeffs, n, lg2_n, shift);
    Prepare_Data(f, n, m, 0);

    return scale;
}
```

fft_tie.tie

```
state w_r       16 add_read_write
state w_i       16 add_read_write


format fft_flix 64 {fft_slot0, fft_slot1}
slot_opcodes fft_slot0 {FFT_LOAD_EVEN, FFT_STORE_EVEN}
slot_opcodes fft_slot1 {FFT_LOAD_ODD, FFT_STORE_ODD}




state a_r       16 add_read_write
state a_i       16 add_read_write
state b_r       16 add_read_write
state b_i       16 add_read_write
state u_r       16 add_read_write
state u_i       16 add_read_write
state v_r       16 add_read_write
state v_i       16 add_read_write
state q_r       16 add_read_write
state q_i       16 add_read_write


state wv        128 add_read_write

state av        128 add_read_write

state bv        128 add_read_write

state done          1 add_read_write
state shift         1 add_read_write
state even_ix       16 add_read_write
state odd_ix        16 add_read_write
state even_const    16 add_read_write

state w_ix          16 add_read_write
state w_inc         16 add_read_write
state m_ix          16 add_read_write
state l             16 add_read_write
state n             16 add_read_write
state n_2           16 add_read_write
state lg2_n         16 add_read_write
state step          16 add_read_write
state ptr_data      32 add_read_write
state ptr_w         32 add_read_write
state debug_reg     32 add_read_write
state time_decimation 1 add_read_write


state qv_r      64 add_read_write
state qv_i      64 add_read_write


function [31:0]COMPLEX_MUL ([15:0] a_r, [15:0] b_r, [15:0] a_i, [15:0] b_i)
{
    wire [31:0] product1r = TIEmul(a_r[15:0],b_r[15:0],1'b1); // signed multiplication
    wire [31:0] product2r = TIEmul(a_r[15:0],b_i[15:0],1'b1); // signed multiplication
    wire [31:0] product1i = TIEmul(a_i[15:0],b_i[15:0],1'b1); // signed multiplication
    wire [31:0] product2i = TIEmul(a_i[15:0],b_r[15:0],1'b1); // signed multiplication
    wire [15:0] mult1 = product1r[30:15] - product1i[30:15];
    wire [15:0] mult2 = product2r[30:15] + product2i[30:15];
    assign COMPLEX_MUL = {mult2, mult1};
}




function [15:0]LEFT_SHIFT ([15:0] a)
{
    wire [15:0] shift_temp = {a[14:0], 1'b0};
    assign LEFT_SHIFT = shift_temp;
}

function [15:0]LEFT_SHIFT_SIGNED ([15:0] a)
{
    wire [15:0] shift_temp = a[15] ? {a[15], a[13:0], 1'b0} : {a[15], a
    [13:0], 1'b1};
    assign LEFT_SHIFT_SIGNED =  shift_temp;
}

function [15:0]RIGHT_SHIFT ([15:0] a)
{
    wire [15:0] shift_temp = {a[15], a[15:1]};
    assign RIGHT_SHIFT = shift_temp;
}

function [15:0]RIGHT_SHIFT_SIGNED ([15:0] a)
{
    wire [15:0] shift_temp = {a[15], a[15:1]};
    assign RIGHT_SHIFT_SIGNED = shift_temp;
}

function [31:0]LEFT_SHIFT_32 ([31:0] a)
{
    wire [31:0] shift_temp = {a[30:0], 1'b0};
    assign LEFT_SHIFT_32 =  shift_temp;
}

function [31:0]RIGHT_SHIFT_32 ([31:0] a)
{
    wire [31:0] shift_temp = {1'h0, a[31:1]};
    assign RIGHT_SHIFT_32 = shift_temp;
}




function [63:0]FFT_NODE_DIF ([15:0] a_r, [15:0] a_i, [15:0] w_r, [15:0] w_i,
     [15:0] b_r, [15:0] b_i, shift)
{

    wire [15:0] v_r = b_r + a_r;
    wire [15:0] v_i = b_i + a_i;

    wire [15:0] v_r_shift = RIGHT_SHIFT_SIGNED(v_r);
    wire [15:0] v_i_shift = RIGHT_SHIFT_SIGNED(v_i);

    wire [15:0] t_r = b_r - a_r;
    wire [15:0] t_i = b_i - a_i;

    wire [31:0] com_mul = COMPLEX_MUL(t_r[15:0], w_r[15:0], t_i[15:0], w_i
    [15:0]);
    wire [15:0] u_r = com_mul[15:0];
    wire [15:0] u_i = com_mul[31:16];

    assign FFT_NODE_DIF = shift ? {v_i_shift, v_r_shift, u_i, u_r} : {v_i,
    v_r, u_i, u_r};

}

function [63:0]FFT_NODE_DIT ([15:0] a_r, [15:0] a_i, [15:0] w_r, [15:0] w_i,
     [15:0] b_r, [15:0] b_i)
{

    wire [31:0] com_mul = COMPLEX_MUL(a_r[15:0], w_r[15:0], a_i[15:0], w_i
    [15:0]);
    wire [15:0] t_r = com_mul[15:0];
    wire [15:0] t_i = com_mul[31:16];

    wire [15:0] u_r = b_r - t_r;
    wire [15:0] u_i = b_i - t_i;
    wire [15:0] v_r = b_r + t_r;
    wire [15:0] v_i = b_i + t_i;
    assign FFT_NODE_DIT = {v_i, v_r, u_i, u_r};

}





/*
 * Complex mult
 */

operation FFT_COM_MUL {inout AR a_r, in AR b_r, inout AR a_i, in AR b_i}{}
{
    wire [31:0] com_mul = COMPLEX_MUL(a_r[15:0], b_r[15:0], a_i[15:0], b_i
    [15:0]);
    wire [15:0] t_r = com_mul[15:0];
    wire [15:0] t_i = com_mul[31:16];

    assign a_r = t_r;
    assign a_i = t_i;
}





operation FFT_2_f_LD {}
            {in b_r, in b_i, out q_r, out q_i}
```

```
{

    assign q_r = b_r;
    assign q_i = b_i;
}




operation FFT_2_FFT {}{in a_r, in a_i, in w_r, in w_i, in b_r, in b_i, out
   u_r, out u_i, out v_r, out v_i}
{

    wire [63:0] fft_node = FFT_NODE_DIT(a_r[15:0], a_i[15:0], w_r[15:0], w_i
   [15:0], b_r[15:0], b_i[15:0]);
    wire [15:0] u_r_t = fft_node[15:0];
    wire [15:0] u_i_t = fft_node[31:16];
    wire [15:0] v_r_t = fft_node[47:32];
    wire [15:0] v_i_t = fft_node[63:48];

    assign u_r = u_r_t;
    assign u_i = u_i_t;
    assign v_r = v_r_t;
    assign v_i = v_i_t;

}

operation FFT_8_FFT_DIT {}{inout av, in wv, inout bv, out even_ix, in
   even_const, in step, out odd_ix}
{
    assign even_ix = even_const;
    assign odd_ix = even_const + step;

    wire [15:0] a_r0 = av[15:0];
    wire [15:0] a_i0 = av[31:16];
    wire [15:0] a_r1 = av[47:32];
    wire [15:0] a_i1 = av[63:48];
    wire [15:0] a_r2 = av[79:64];
    wire [15:0] a_i2 = av[95:80];
    wire [15:0] a_r3 = av[111:96];
    wire [15:0] a_i3 = av[127:112];


    wire [15:0] b_r0 = bv[15:0];
    wire [15:0] b_i0 = bv[31:16];
    wire [15:0] b_r1 = bv[47:32];
    wire [15:0] b_i1 = bv[63:48];
    wire [15:0] b_r2 = bv[79:64];
    wire [15:0] b_i2 = bv[95:80];
    wire [15:0] b_r3 = bv[111:96];
    wire [15:0] b_i3 = bv[127:112];

    wire [15:0] w_r0 = wv[15:0];
    wire [15:0] w_i0 = wv[31:16];
    wire [15:0] w_r1 = wv[47:32];
    wire [15:0] w_i1 = wv[63:48];
    wire [15:0] w_r2 = wv[79:64];
    wire [15:0] w_i2 = wv[95:80];
    wire [15:0] w_r3 = wv[111:96];
    wire [15:0] w_i3 = wv[127:112];


    wire [63:0] fft_node0 = FFT_NODE_DIT(a_r0[15:0], a_i0[15:0], w_r0[15:0],
     w_i0[15:0], b_r0[15:0], b_i0[15:0]);
    wire [15:0] u_r_t0 = fft_node0[15:0];
    wire [15:0] u_i_t0 = fft_node0[31:16];
    wire [15:0] v_r_t0 = fft_node0[47:32];
    wire [15:0] v_i_t0 = fft_node0[63:48];

    wire [63:0] fft_node1 = FFT_NODE_DIT(a_r1[15:0], a_i1[15:0], w_r1[15:0],
     w_i1[15:0], b_r1[15:0], b_i1[15:0]);
    wire [15:0] u_r_t1 = fft_node1[15:0];
    wire [15:0] u_i_t1 = fft_node1[31:16];
    wire [15:0] v_r_t1 = fft_node1[47:32];
    wire [15:0] v_i_t1 = fft_node1[63:48];

    wire [63:0] fft_node2 = FFT_NODE_DIT(a_r2[15:0], a_i2[15:0], w_r2[15:0],
     w_i2[15:0], b_r2[15:0], b_i2[15:0]);
    wire [15:0] u_r_t2 = fft_node2[15:0];
    wire [15:0] u_i_t2 = fft_node2[31:16];
    wire [15:0] v_r_t2 = fft_node2[47:32];
    wire [15:0] v_i_t2 = fft_node2[63:48];

    wire [63:0] fft_node3 = FFT_NODE_DIT(a_r3[15:0], a_i3[15:0], w_r3[15:0],
     w_i3[15:0], b_r3[15:0], b_i3[15:0]);
    wire [15:0] u_r_t3 = fft_node3[15:0];
    wire [15:0] u_i_t3 = fft_node3[31:16];
    wire [15:0] v_r_t3 = fft_node3[47:32];
    wire [15:0] v_i_t3 = fft_node3[63:48];


    assign bv = {v_i_t3, v_r_t3, v_i_t2, v_r_t2, v_i_t1, v_r_t1, v_i_t0,
   v_r_t0};
    assign av = {u_i_t3, u_r_t3, u_i_t2, u_r_t2, u_i_t1, u_r_t1, u_i_t0,
   u_r_t0};

}

operation FFT_8_FFT_DIF {}{inout av, in wv, inout bv, out even_ix, in
    even_const, in step, out odd_ix, in shift}
{
    assign even_ix = even_const;
    assign odd_ix = even_const + step;

    wire [15:0] a_r0 = av[15:0];
    wire [15:0] a_i0 = av[31:16];
    wire [15:0] a_r1 = av[47:32];
    wire [15:0] a_i1 = av[63:48];
    wire [15:0] a_r2 = av[79:64];
    wire [15:0] a_i2 = av[95:80];
    wire [15:0] a_r3 = av[111:96];
    wire [15:0] a_i3 = av[127:112];


    wire [15:0] b_r0 = bv[15:0];
    wire [15:0] b_i0 = bv[31:16];
    wire [15:0] b_r1 = bv[47:32];
    wire [15:0] b_i1 = bv[63:48];
    wire [15:0] b_r2 = bv[79:64];
    wire [15:0] b_i2 = bv[95:80];
    wire [15:0] b_r3 = bv[111:96];
    wire [15:0] b_i3 = bv[127:112];

    wire [15:0] w_r0 = wv[15:0];
    wire [15:0] w_i0 = wv[31:16];
    wire [15:0] w_r1 = wv[47:32];
    wire [15:0] w_i1 = wv[63:48];
    wire [15:0] w_r2 = wv[79:64];
    wire [15:0] w_i2 = wv[95:80];
    wire [15:0] w_r3 = wv[111:96];
    wire [15:0] w_i3 = wv[127:112];


    wire [63:0] fft_node0 = FFT_NODE_DIF(a_r0[15:0], a_i0[15:0], w_r0[15:0],
     w_i0[15:0], b_r0[15:0], b_i0[15:0], shift);
    wire [15:0] u_r_t0 = fft_node0[15:0];
    wire [15:0] u_i_t0 = fft_node0[31:16];
    wire [15:0] v_r_t0 = fft_node0[47:32];
    wire [15:0] v_i_t0 = fft_node0[63:48];

    wire [63:0] fft_node1 = FFT_NODE_DIF(a_r1[15:0], a_i1[15:0], w_r1[15:0],
     w_i1[15:0], b_r1[15:0], b_i1[15:0], shift);
    wire [15:0] u_r_t1 = fft_node1[15:0];
    wire [15:0] u_i_t1 = fft_node1[31:16];
    wire [15:0] v_r_t1 = fft_node1[47:32];
    wire [15:0] v_i_t1 = fft_node1[63:48];

    wire [63:0] fft_node2 = FFT_NODE_DIF(a_r2[15:0], a_i2[15:0], w_r2[15:0],
     w_i2[15:0], b_r2[15:0], b_i2[15:0], shift);
    wire [15:0] u_r_t2 = fft_node2[15:0];
    wire [15:0] u_i_t2 = fft_node2[31:16];
    wire [15:0] v_r_t2 = fft_node2[47:32];
    wire [15:0] v_i_t2 = fft_node2[63:48];

    wire [63:0] fft_node3 = FFT_NODE_DIF(a_r3[15:0], a_i3[15:0], w_r3[15:0],
     w_i3[15:0], b_r3[15:0], b_i3[15:0], shift);
    wire [15:0] u_r_t3 = fft_node3[15:0];
    wire [15:0] u_i_t3 = fft_node3[31:16];
    wire [15:0] v_r_t3 = fft_node3[47:32];
    wire [15:0] v_i_t3 = fft_node3[63:48];


    assign bv = {v_i_t3, v_r_t3, v_i_t2, v_r_t2, v_i_t1, v_r_t1, v_i_t0,
    v_r_t0};
    assign av = {u_i_t3, u_r_t3, u_i_t2, u_r_t2, u_i_t1, u_r_t1, u_i_t0,
    u_r_t0};

}

operation FFT_BIT_REVERSE {inout AR n}{}
{
wire [31:0] lvl0 = (n & 32'hffff0000) >> 16 | (n & 32'h0000ffff) << 16;
wire [31:0] lvl1 = (lvl0 & 32'hff00ff00) >> 8 | (lvl0& 32'h00ff00ff) << 8;
wire [31:0] lvl2 = (lvl1 & 32'hf0f0f0f0) >> 4 | (lvl1& 32'h0f0f0f0f) << 4;
wire [31:0] lvl3 = (lvl2 & 32'hcccccccc) >> 2 | (lvl2& 32'h33333333) << 2;
wire [31:0] lvl4 = (lvl3 & 32'haaaaaaaa) >> 1 | (lvl3& 32'h55555555) << 1;
assign n = lvl4;
}




operation FFT_INIT {} {out done, out even_ix, out odd_ix, out w_ix, out m_ix
    , out l, out step, out w_inc, in n, out n_2, in time_decimation}
{
    assign done = 1'h0;
    assign even_ix = 16'h0;
    assign odd_ix = time_decimation ? 16'h1 : n_2_temp;
    assign w_ix = 16'h0;
    assign m_ix = 16'h0;
    assign l = 16'h0;
    assign step = time_decimation ? 16'h1 : n_2_temp;
    wire [15:0] n_2_temp = RIGHT_SHIFT(n);
    assign w_inc = time_decimation ? n_2_temp : 16'h1;
    assign n_2 = n_2_temp;
}

operation FFT_LOAD_EVEN {}{in even_ix, in ptr_data, in done, in shift, inout
     bv, out VAddr, in MemDataIn32, out LoadByteDisable, out debug_reg, in
    time_decimation}
{
    wire [31:0] even_ix_ext = {16'h0, even_ix};
    wire [31:0] even_ix_shift = LEFT_SHIFT_32(even_ix_ext);
    wire [31:0] VAddr_temp = ptr_data + LEFT_SHIFT_32(even_ix_shift);
    assign debug_reg = VAddr_temp;
    assign VAddr = ptr_data + LEFT_SHIFT_32(even_ix_shift);
    assign LoadByteDisable = {16{done}};


    wire [31:0] data_temp = MemDataIn32;
    wire [15:0] data_temp_r = RIGHT_SHIFT_SIGNED(data_temp[15:0]);
    wire [15:0] data_temp_i = RIGHT_SHIFT_SIGNED(data_temp[31:16]);
    assign bv = (shift & time_decimation) ? {data_temp_i[15:0], data_temp_r
   [15:0], bv[127:32]} : {data_temp[31:0], bv[127:32]};


}



operation FFT_LOAD_ODD {}{in odd_ix, in ptr_data, in done, inout av, out
   VAddr, in MemDataIn32, out LoadByteDisable, out debug_reg}
{
    wire [31:0] odd_ix_ext = {16'h0, odd_ix};
    wire [31:0] odd_ix_ext_shift = LEFT_SHIFT_32(odd_ix_ext);
    wire [31:0] VAddr_temp = ptr_data + LEFT_SHIFT_32(odd_ix_ext_shift);
    assign debug_reg = VAddr_temp;
    assign VAddr = ptr_data + LEFT_SHIFT_32(odd_ix_ext_shift);
    assign LoadByteDisable = {16{done}};
    wire [31:0] data_temp = MemDataIn32;
    assign av = {data_temp[31:0], av[127:32]};

}



operation FFT_LOAD_W {}{inout w_ix, inout even_ix, out odd_ix, in ptr_w, in
   done, in n_2, in shift, inout wv, in step, out VAddr, in MemDataIn32, out
    LoadByteDisable, out debug_reg, in w_inc}
{
    wire [31:0] w_ix_ext = {16'h0, w_ix};
    wire [31:0] w_ix_ext_shift = LEFT_SHIFT_32(w_ix_ext);
    wire [31:0] VAddr_temp = ptr_w + LEFT_SHIFT_32(w_ix_ext_shift);
    assign debug_reg = VAddr_temp;
    assign VAddr = ptr_w + LEFT_SHIFT_32(w_ix_ext_shift);
    assign LoadByteDisable = {16{done}};

    wire [31:0] data_temp = MemDataIn32;
    wire [15:0] data_temp_r = RIGHT_SHIFT_SIGNED(data_temp[15:0]);
    wire [15:0] data_temp_i = RIGHT_SHIFT_SIGNED(data_temp[31:16]);
    assign wv = shift ? {data_temp_i[15:0], data_temp_r[15:0], wv[127:32]} :
    {data_temp[31:0], wv[127:32]};



    wire [15:0] even_temp = even_ix + 16'h1;

    wire [15:0] w_ix_temp =  w_ix + w_inc;
    wire jump = (w_ix_temp < n_2) ? 1'h0 : 1'h1;

    wire [15:0] even_ix_wire = jump ? even_temp + step : even_temp;
    assign even_ix = even_ix_wire;
    assign odd_ix = even_ix_wire + step;
    assign w_ix = jump ? 16'h0 : w_ix_temp;
}



operation FFT_STORE_EVEN {} {in even_ix, in ptr_data, in done, inout bv, out
     VAddr, out MemDataOut32, out StoreByteDisable}
{
    wire [31:0] even_ix_ext = {16'h0, even_ix};
    wire [31:0] even_ix_shift = LEFT_SHIFT_32(even_ix_ext);
    assign VAddr = ptr_data + LEFT_SHIFT_32(even_ix_shift);
    assign StoreByteDisable = {16{done}};


    assign MemDataOut32 = bv[31:0];

    assign bv = {32'h0, bv[127:32]};
}



operation FFT_STORE_ODD {} {inout odd_ix, in ptr_data, in done, in step,
    inout av, inout m_ix, inout even_ix, out VAddr, out MemDataOut32, out
    StoreByteDisable}
{
    wire [31:0] odd_ix_ext = {16'h0, odd_ix};
    wire [31:0] even_ix_shift = LEFT_SHIFT_32(odd_ix_ext);
    assign VAddr = ptr_data + LEFT_SHIFT_32(even_ix_shift);
    assign StoreByteDisable = {16{done}};

    assign MemDataOut32 = av[31:0];

    assign av = {32'h0, av[127:32]};

    wire [15:0] even_temp = even_ix + 16'h1;
    wire [15:0] m_temp =  m_ix + 16'h1;
    wire jump = (m_temp < step) ? 1'h0 : 1'h1;

    wire [15:0] even_ix_wire = jump ? even_temp + step : even_temp;
    assign even_ix = even_ix_wire;
    assign odd_ix = even_ix_wire + step;
    assign m_ix = jump ? 16'h0 : m_temp;
}
```

```



operation FFT_UPDATE {out AR done_out} {inout even_ix, inout l, out done,
    inout step, out odd_ix, out even_const, in n, in lg2_n, inout w_inc, in
    time_decimation, inout shift}
{
    wire done_temp = !(even_ix < n);
    assign even_const = done_temp ? 16'h0 : even_ix;
    wire [15:0] even_ix_temp = done_temp ? 16'h0 : even_ix;
    assign even_ix = even_ix_temp;
    wire [15:0] step_shift = time_decimation ? LEFT_SHIFT(step) :
    RIGHT_SHIFT(step);
    wire [15:0] step_temp = done_temp ? step_shift : step;
    assign odd_ix = even_ix_temp + step_temp;
    assign step = step_temp;
    wire [15:0] w_inc_shift = time_decimation ? RIGHT_SHIFT(w_inc) :
    LEFT_SHIFT(w_inc);
    assign w_inc = done_temp ? w_inc_shift : w_inc;
    wire [15:0] l_temp = done_temp ? l + 16'h1 : l;
    wire l_0 = (l<0) ? 1'b1 : 1'b0;
    assign shift = ((!time_decimation) & l_0) ? 1'h1 : shift;
    assign l = l_temp;
    assign done = (l_temp < lg2_n) ? 1'h0 : 1'h1;
    assign done_out = (l_temp < lg2_n) ? 1'h0 : 1'h1;
}
```

3 Questions

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

Precalculation of twiddle factors Twiddle factors are precalculated in GenerationCoeffition for a signal of variable length and put in a contiguous array. This reduces overhead in FFT calculation as they don't need to be re-calculated at each step.

Improvment of Data Locality The real and imaginary signals are composed into one single array with complex entries. This reduces the load operation overhead as both real and imaginary components can be loaded with one single memory load operation. This also reduces the number of pointers that need to exist in registers.

Branchless Programming Branches inside the fft_adv hot calculation loop are converted into branchless variants. (shown in fft_exec [fft_adv.c:47-79]) This reduces the overhead caused by branch instructions as they might cause pipeline flushes, especially if the conditions are based on the input data where branch predictors fail to help.

Loop Unrolling Operations are duplicated in hot loops [fft_adv.c:49-63] and the total number of iterations is reduced. This reduces the ratio of loop overhead to payload calculation.

SIMD Operations The SIMD TIE instruction "FFT_8_FFT" [fft_adv.c:65] performs FFT on complex data vectors by parallelizing the computation. It calculates indices, extracts real and imaginary parts, performs FFT operations using the "FFT_NODE" function, and combines results back into output vectors. This approach reduces loop overhead, improves cache utilization, and processes multiple data points simultaneously, significantly enhancing performance compared to implementations in pure C code. The "FFT_NODE" function execute 4 multiplications in parallel, as well the sum and subtraction needed for the complex multiplication.

HW Bit-Reversal Added a bit-reverse instruction in hardware. This reduces the output reordering overhead.

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

There were many parts that were accelerated as mentioned before, but the most important one is the node calculation. There the node operations, sum, subtraction and complex multiplication, are done in parallel, because in each step of the FFT the different nodes are independent from one another, allowing for parallelization.

What is the impact on execution time (speedup)?

Original implementation 1024pt 1092610 Cycles (base)

Optimized implementation dit 1024pt 96449 Cycles (11.33 speedup)

What is the impact on the hardware (area)?

68846 new gates (5833 decode logic, 63013 instructions, states, regfiles, etc)

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

- The library software was changed to use the newly added TIE instructions. User code (caller code) didn't require major changes.
- Because of the initial twiddle factor calculation, the limitation of 1024 point FFT is lifted.
- The Tie functions were used in place of normal C functions.
- The not needed loops were removed.

4 Detailed Report

4.1 Software Optimization

Before attempting to optimize the given FFT calculation algorithm, we analyzed the existing solution and identified some performance issues. In the following, we discuss some issues we have identified with the existing implementation and their respective solutions.

1024 FFT Point Limitation

The existing solution uses a precalculated fixed point sine wave as a lookup table [fft.c:172] for the required twiddle factors. The lookup table not only imposes a limitation of how long the input signal can be, but also adds overhead in the hot-loop for the index calculation [fft.c:120] of the table lookup.

In order to remove the index calculation, the twiddle factors that will be used needed to be stored in sequential order [main.c:20]. Since the application will ultimately be running with a fixed set of input signal sizes, the according twiddle factor lookup tables can be generated at application initialization time [main.c:58-59, GenerateCoefficient.c]

Data Locality

The given implementation uses two distinct arrays [fft.c:44] for representing complex signals, one for the real component and another for the imaginary component. This forces the FFT algorithm to use two distinct pointers and ultimately two memory load and store operations when reading or writing an item in these two arrays. Since the FFT algorithm always uses the imaginary and the real component at the same time, it is reasonable to merge the two arrays into one [main.c:18] and use an aggregate datatype to represent the complex signal.

Decimation Calculation

The decimation calculation that is implemented in the given solution [fft.c:62-76] uses nested loops in the decimation calculation for reordering the input before the butterfly calculation. We replaced this with a simple branchless bit-reverse [fft_adv.c:3-9] and noticed a slight speedup.

The given implementation also only implements a dit implementation, because of that a dif implementation was made in order to better debug the optimized dif implementation.

4.2 Hardware Optimization

After applying the software optimizations, we started converting all hot-loops in the implementation to possibly vectorized hardware FSM implementations. In the following, we discuss some important points regarding the hardware optimization.

Data Reordering

The HW instruction FFT_BIT_REVERSE [fft_tie.tie:319-326] does a bit-reversal used in the data reordering process..

Vectorized Loads/Stores

The HW instructions FFT_LOAD_EVEN [fft_tie.tie:346-362], FFT_LOAD_ODD [fft_tie.tie:366-377] and FFT_LOAD_W [fft_tie.tie:381-406] load the input data in shift registers, shifts the data if needed and update the indexes. The HW instructions FFT_STORE_EVEN [fft_tie.tie:410-421] and FFT_STORE_ODD [fft_tie.tie:425-444] store the data after the calculations back in memory.

The even and odd loads/stores are defined in separate slots, what allows to reas/write data in parallel as their addresses are never the same.

Butterfly Calculation

The butterfly calculation for both FFT and IFFT [fft_tie.tie:191-253] for dit and [fft_tie.tie:255-317] for dif calculates 4 nodes in parallel, what greatly improves performance.

Node Calculation

The node calculations [fft_tie.tie:107-125] for dit and [fft_tie.tie:127-140] for dif execute the complex multiplication and the sum/subtraction for each node. As this is defined as a function, each time it is defined in the FFT_8_FFT mentioned above, a new hardware instance is created.

Complex multiplication

The complex multiplication function [fft_tie.tie:54-63] uses TIEmul to execute 4 multiplications in parallel. After that it does the sum and subtraction in parallel. As multiplication is a costly operation, that gratly increases the performance.

Control operations

FFT_INIT [fft_tie.tie:332-344] initializes the control signals. FFT_UPDATE [fft_tie.tie:448-466] updates after each calculation the control signals, that determine if the next step should be done, the new indexes and if the FFT is done.

4.3 Testing

For testing both the correctness and the performance of each optimization step, a simple test bench [main.c] has been utilized.

4.4 Conclusion

Most Impactful Optimizations

During optimization, we tested every optimization step and measured its impact on the performance. The most effective optimization steps were:

- Doing the bit-reverse in hardware instead of software
- Doing the the node operations as a single TIE instruction.

Remarks on the Task

The task gave us an insight into HW-SW Codesign using the Verilog-like TIE language for the reconfigurable cores of Xtensa-series. The task also was not just practice but was accompanied with a great amount of research on FFT implementation strategies. Both, experimenting with different optimization approaches and the cycle of develop-test-measure, were an informative and practical experience.