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

WS 2022/23

Keerthan Kopparam Radhakrishna

keerthan.kopparam_radhakrishna@mailbox.tu-dresden.de

Nitin Krishna Venkatesan

nitin.krishna_venkatesan@mailbox.tu-dresden.de

Rohan Krishna Vijayaraghavan

rohan.krishna_vijayaraghavan@mailbox.tu-dresden.de

10.04.2023

Supervisors

Viktor Razilov

Dr. Emil Matúš

Introduction

The HW/SW Co-Design Lab is intended to give a first practical access to this topic and experience 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 are done:

1. Implementation of a finite impulse response (FIR) filter design, including performance analysis; HW/SW optimization: e.g., by Fusion, SIMD extension, etc.; also 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 using different implementations such as the Decimation in Time (DIT) and Decimation in Frequency (DIF) algorithm.

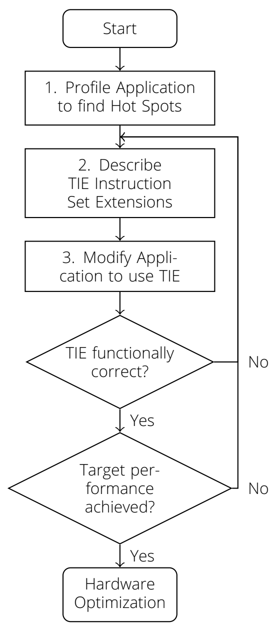

Figure 1: Similar to Tensilica Instruction Extension (TIE) Language, User's Guide. 02/2014, p. 4.

The detailed project description can be found at the lab webpage:
`https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/21521498113`

Source Code

Source code C files (only excerpts from the parts that have been changed).

dit.c

```c
#include "fft.h"
#include <xtensa/tie/tie_fft.h>

/*
 *  fix_fft_dit() - perform fast Fourier transform.
 *
 *  if n>0 FFT is done, if n<0 inverse IFFT is done
 *  fr[n],fi[n] are real,imaginary arrays, INPUT AND RESULT.
 *  size of data = 2^m
 *  set inverse to 0=dft, 1=idft
 */
#define aligned_by_16 __attribute__ ((aligned(16)))
#define aligned_by_8 __attribute__ ((aligned(8)))
#define aligned_by_4 __attribute__ ((aligned(4)))
#define fixed_complex int

int fix_fft_dit(fixed fr[], fixed fi[], int m, int inverse)
{
    int mr,nn,i,j,l,k,istep, n, scale, shift;

    fixed qr,qi;        //even input
    fixed tr,ti;        //odd input
    fixed wr,wi;        //twiddle factor

    //number of input data
    n = 1<<m;

    if(n > N_WAVE) return -1;

    mr = 0;
```

```
    nn = n - 1;
    scale = 0;

    /* decimation in time - re-order data */
    for(m=1; m<=nn; ++m) {
        l = n;
        do{
            l >>= 1;
        }while(mr+l > nn);
        mr = (mr & (l-1)) + l;

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

        ti = fi[m];
        fi[m] = fi[mr];
        fi[mr] = ti;
    }


    l = 1;
    k = LOG2_N_WAVE-1;
    while(l < n)
        {
            if(inverse)
            {
                /* variable scaling, depending upon data */
                shift = 0;
                for(i=0; i<n; i=i+8)
                {
                    SIMD_REG_128 j = FFT_SIMD_LOAD(fr, i);
                    SIMD_REG_128 m = FFT_SIMD_LOAD(fi, i);

                    if (FFT_CHECK_SHIFT_CONDITION(j, m))
                    {
                        shift = 1;
                        break;
                    }
                }
                if(shift) ++scale;
            }
            else
            {
                shift = 1;
            }

            // For values of N = {1,2,3}
            if (l == 1)
            {
```

```
            register SIMD_REG_128 real, imag;

            for (i=0; i<n; i = i+8)
            {
                k = LOG2_N_WAVE-1;

                real = FFT_SIMD_LOAD(fr, i);
                imag = FFT_SIMD_LOAD(fi, i);

                SIMD_DIT_STAGE1(k, inverse, shift, real, imag);

                --k;

                SIMD_DIT_STAGE2(k, inverse, shift, real, imag);

                --k;

                SIMD_DIT_STAGE3(k, inverse, shift, real, imag);

                FFT_SIMD_STORE_SHUFFLE(fr, i, real, 1);
                FFT_SIMD_STORE_SHUFFLE(fi, i, imag, 1);
            }

            l = 4;
        }
        else { // For values of N > 3

            register SIMD_REG_128 real_1, imag_1, real_2, imag_2;

            WUR_STATE_K(k);

            for (i=0; i<n; i = i+2*l)
            {
                for (m = i; m<l+i; m+=8)
                {
                    // Load Values
                    FFT_SIMD_LOAD_IL(fr, m, real_1, real_2, 1);
                    FFT_SIMD_LOAD_IL(fi, m, imag_1, imag_2, 1);

                    FFT_SIMD_LOAD_IL(fr, m+l, real_1, real_2, 0);
                    FFT_SIMD_LOAD_IL(fi, m+l, imag_1, imag_2, 0);

                    // Do the actual calculation
                    SIMD_DIT_STAGE_N(real_1, imag_1, shift, m, inverse);
                    SIMD_DIT_STAGE_N(real_2, imag_2, shift, m+4, inverse
);

                    // Store Values
                    FFT_SIMD_STORE_IL(fr, m, real_1, real_2, 1);
                    FFT_SIMD_STORE_IL(fi, m, imag_1, imag_2, 1);

```

```
                        FFT_SIMD_STORE_IL(fr, m+l, real_1, real_2, 0);
                        FFT_SIMD_STORE_IL(fi, m+l, imag_1, imag_2, 0);
                    }
                }
            }
            --k;
            l <<= 1;
        }
        return scale;
    }
```

dif.c

```
/*
 * dif.c
 *
 *  Created on: 13.04.2023
 *      Author: keerthan.kopparam
 */

#include "fft.h"
#include <xtensa/tie/tie_fft.h>

/*
 *  fix_fft_dif() - perform fast Fourier transform.
 *
 *  if n>0 FFT is done, if n<0 inverse IFFT is done
 *  fr[n],fi[n] are real,imaginary arrays, INPUT AND RESULT.
 *  size of data = 2^m
 *  set inverse to 0=dft, 1=idft
 */
#define aligned_by_16 __attribute__ ((aligned(16)))
#define aligned_by_8 __attribute__ ((aligned(8)))
#define aligned_by_4 __attribute__ ((aligned(4)))
#define fixed_complex int

int fix_fft_dif(fixed fr[], fixed fi[], int m, int inverse)
    {
        int mr,nn,i,j,l,k, n, scale, shift;

        fixed tr,ti;
        n = 1 << m;
        int size = m;

        if(n > N_WAVE) return -1;

        mr = 0;
        nn = n - 1;
```

```
        scale = 0;

        l = n>>1;
        k = LOG2_N_WAVE-m;

        while(l > 0)
        {
            if(inverse)
            {
                shift = 0;
                for(i=0; i<n; i=i+8)
                {
                    SIMD_REG_128 j = FFT_SIMD_LOAD(fr, i);
                    SIMD_REG_128 m = FFT_SIMD_LOAD(fi, i);

                    if (FFT_CHECK_SHIFT_CONDITION(j, m))
                    {
                        shift = 1;
                        break;
                    }
                }
                if(shift) ++scale;
            }
            else
            {
                shift = 1;
            }


            if (l > 4) {
                WUR_STATE_K(k);

                for (i=0; i<n; i = i+(2*l)) {
                    for (m = i; m<l+i; m+=8) {
                        SIMD_REG_128 real_1;
                        SIMD_REG_128 imag_1;
                        SIMD_REG_128 real_2;
                        SIMD_REG_128 imag_2;

                        // Load Values
                        FFT_SIMD_LOAD_IL(fr, m, real_1, real_2, 1);
                        FFT_SIMD_LOAD_IL(fi, m, imag_1, imag_2, 1);

                        FFT_SIMD_LOAD_IL(fr, m+l, real_1, real_2, 0);
                        FFT_SIMD_LOAD_IL(fi, m+l, imag_1, imag_2, 0);

                        // Do the actual calculation
                        SIMD_DIF_STAGE_N(real_1, imag_1, shift, m, inverse);
                        SIMD_DIF_STAGE_N(real_2, imag_2, shift, m+4, inverse
  );

```

```
                        // Store Values
                        FFT_SIMD_STORE_IL(fr, m, real_1, real_2, 1);
                        FFT_SIMD_STORE_IL(fi, m, imag_1, imag_2, 1);

                        FFT_SIMD_STORE_IL(fr, m+l, real_1, real_2, 0);
                        FFT_SIMD_STORE_IL(fi, m+l, imag_1, imag_2, 0);
                    }
                }
            } else { /

                for (i=0; i<n; i = i+8) {
                    register SIMD_REG_128 real, imag;

                    k = 7;

                    real = FFT_SIMD_LOAD_SHUFFLE(fr, i, 0);
                    imag = FFT_SIMD_LOAD_SHUFFLE(fi, i, 0);

                    SIMD_DIF_THIRD_LAST_STAGE(k, inverse, shift, real, imag)
;

                    ++k;

                    SIMD_DIF_SECOND_LAST_STAGE(k, inverse, shift, real, imag
);

                    ++k;

                    SIMD_DIF_LAST_STAGE(k, inverse, shift, real, imag);

                    FFT_SIMD_STORE(fr, i, real);
                    FFT_SIMD_STORE(fi, i, imag);
                }
                break;
            }
            ++k;
            l >>= 1;
        }



            /* decimation in frequency - re-order data */
        for(m=1; m<=nn; ++m) {
                l = n;
                do{
                    l >>= 1;
                }while(mr+l > nn);
                mr = (mr & (l-1)) + l;

                if(mr <= m) continue;
                tr = fr[m];
```

```
                fr[m] = fr[mr];
                fr[mr] = tr;

                ti = fi[m];
                fi[m] = fi[mr];
                fi[mr] = ti;
            }


        return scale;
    }
```

fft_tie.tie

```
// Register declaration
regfile SIMD_REG_128 128 4 v2rf

// Immediate values
immediate_range immediate 0 1 1

// Generic Functions

function [15:0] MUL16 ([15:0] a, [15:0] b)
{
    wire [31:0] c = TIEmul(a, b, 1'b1);
    assign MUL16 = c[30:15];
}

function [15:0] SIN ([9:0] idx)
{
    wire [9:0] idx1 = TIEadd(idx, ~12'h200, 1'b1);
    wire [15:0] sin = SIN_LUT[idx1];
    wire [15:0] minus_sin = TIEadd(16'b0, ~sin, 1'b1);
    assign SIN = TIEmux(idx[9], SIN_LUT[idx], minus_sin);
}

function [63:0] even_part([127:0] input)
{
    wire [15:0] even_0 = input[127:112];
    wire [15:0] even_2 = input[95:80];
    wire [15:0] even_4 = input[63:48];
    wire [15:0] even_6 = input[31:16];
    assign even_part = {even_0, even_2, even_4, even_6};
}

function [63:0] odd_part([127:0] input)
{
    wire [15:0] odd_1 = input[111:96];
```

```
    wire [15:0] odd_3 = input[79:64];
    wire [15:0] odd_5 = input[47:32];
    wire [15:0] odd_7 = input[15:0];

    assign odd_part = {odd_1, odd_3, odd_5, odd_7};
}

function [15:0] absolute([15:0] input)
{
    wire [15:0] neg = TIEadd(15'b0, ~input, 1'b1);
    assign absolute = TIEmux(input[15:15], input, neg);
}

function [31:0] twiddle_factor([31:0] j, [0:0] inverse, [0:0] shift)
{
    wire [9:0] idx = TIEadd(j, 256, 1'b0);
    wire [15:0] wr = SIN(idx);
    wire [15:0] wi = TIEadd(16'b0, ~SIN(j[9:0]), 1'b1);

    wire [15:0] temp = TIEmux(inverse, wi, TIEadd(16'b0, ~wi, 1'b1));

    wire [15:0] wr_res = TIEmux(shift, wr, {1{wr[15]}, wr[15:1]});
    wire [15:0] wi_res = TIEmux(shift, temp, {1{temp[15]}, temp[15:1]});

    assign twiddle_factor = {wr_res, wi_res};
}

function [127:0] twiddle4 ([31:0] m, [31:0] k, inverse, shift) shared
{
    wire [31:0] temp1 = m << k;
    wire [31:0] temp2 = (m+1) << k;
    wire [31:0] temp3 = (m+2) << k;
    wire [31:0] temp4 = (m+3) << k;

    wire [31:0] t1 = twiddle_factor(temp1, inverse, shift);
    wire [31:0] t2 = twiddle_factor(temp2, inverse, shift);
    wire [31:0] t3 = twiddle_factor(temp3, inverse, shift);
    wire [31:0] t4 = twiddle_factor(temp4, inverse, shift);

    assign twiddle4 = {t1, t2, t3, t4};
}

state STATE_K 32 32'b0 add_read_write

// DIT Butterfly
function [63:0] fft_dit_butterfly ([31:0] even, [31:0] odd, [31:0] twiddle,
    [0:0] shift)
{
    wire [15:0] wr      = twiddle[31:16];
    wire [15:0] wi      = twiddle[15:0];
```

```
    wire [15:0] tr  = TIEmux(shift, even[31:16], {1{even[31]}, even[31:17]});
    wire [15:0] ti  = TIEmux(shift, even[15:0], {1{even[15]}, even[15:1]});

    wire [15:0] tr_op1  = MUL16(wr, odd[31:16]);
    wire [15:0] tr_op2  = MUL16(wi, odd[15:0]);
    wire [15:0] tr_temp     = TIEadd(tr_op1, ~tr_op2, 1'b1);

    wire [15:0] ti_op1  = MUL16(wi, odd[31:16]);
    wire [15:0] ti_op2  = MUL16(wr, odd[15:0]);
    wire [15:0] ti_temp         = TIEadd(ti_op1, ti_op2, 1'b0);

    wire [15:0] fri_res = TIEadd( tr, tr_temp, 1'b0);
    wire [15:0] frj_res = TIEadd( ti, ti_temp, 1'b0);
    wire [15:0] fii_res  = TIEadd(tr, ~tr_temp, 1'b1);
    wire [15:0] fij_res  = TIEadd(ti, ~ti_temp, 1'b1);

    assign fft_dit_butterfly = {fri_res, frj_res, fii_res, fij_res};
}

// DIF Butterfly
function [63:0] fft_dif_butterfly ([31:0] even, [31:0] odd, [31:0] twiddle,
    [0:0] shift)
{
    wire [15:0] wr      = twiddle[31:16];
    wire [15:0] wi      = twiddle[15:0];
    wire [15:0] tr  = even[31:16];
    wire [15:0] ti  = even[15:0];

    wire [15:0] temp1 = TIEadd(tr, ~odd[31:16], 1'b1);
    wire [15:0] temp2 = TIEadd(odd[15:0], ~ti, 1'b1);
    wire [15:0] temp1_wr = MUL16(temp1, wr);
    wire [15:0] temp2_wi = MUL16(temp2, wi);
    wire [15:0] temp1_wi = MUL16(temp1, wi);
    wire [15:0] temp2_wr = MUL16(temp2, wr);

    wire [15:0] fri_res = TIEadd( tr, odd[31:16], 1'b0);
    wire [15:0] frj_res = TIEadd( ti, odd[15:0], 1'b0);
    wire [15:0] even_rs = TIEmux(shift[0], fri_res, {fri_res[15], fri_res
    [15:1]});
    wire [15:0] even_ri = TIEmux(shift[0], frj_res, {frj_res[15], frj_res
    [15:1]});

    wire [15:0] fii_res  = TIEadd(temp1_wr, temp2_wi, 1'b0);
    wire [15:0] fij_res  = TIEadd(temp1_wi, ~temp2_wr, 1'b1);

    assign fft_dif_butterfly = {even_rs, even_ri, fii_res, fij_res};
}

function [255:0] dif_butterfly_chain([63:0] qr, [63:0] qi, [63:0] fr, [63:0]
     fi, [127:0] twiddle_reg128, [0:0] shift) shared
```

```
{
    wire [31:0] twiddle1 = twiddle_reg128[127:96];
    wire [31:0] twiddle2 = twiddle_reg128[95:64];
    wire [31:0] twiddle3 = twiddle_reg128[63:32];
    wire [31:0] twiddle4 = twiddle_reg128[31:0];

    wire [63:0] result1 = fft_dif_butterfly({qr[63:48],qi[63:48]}, {fr[63:48],fi[63:48]}, twiddle1, shift);
    wire [63:0] result2 = fft_dif_butterfly({qr[47:32],qi[47:32]}, {fr[47:32],fi[47:32]}, twiddle2, shift);
    wire [63:0] result3 = fft_dif_butterfly({qr[31:16],qi[31:16]}, {fr[31:16],fi[31:16]}, twiddle3, shift);
    wire [63:0] result4 = fft_dif_butterfly({qr[15:0 ],qi[15:0 ]}, {fr[15:0 ],fi[15:0 ]}, twiddle4, shift);

    assign dif_butterfly_chain = {result1, result2, result3, result4};
}

function [255:0] dit_butterfly_chain([63:0] qr, [63:0] qi, [63:0] fr, [63:0] fi, [127:0] twiddle_reg128, [0:0] shift) shared
{
    wire [31:0] twiddle1 = twiddle_reg128[127:96];
    wire [31:0] twiddle2 = twiddle_reg128[95:64];
    wire [31:0] twiddle3 = twiddle_reg128[63:32];
    wire [31:0] twiddle4 = twiddle_reg128[31:0];

    wire [63:0] result1 = fft_dit_butterfly({qr[63:48],qi[63:48]}, {fr[63:48],fi[63:48]}, twiddle1, shift);
    wire [63:0] result2 = fft_dit_butterfly({qr[47:32],qi[47:32]}, {fr[47:32],fi[47:32]}, twiddle2, shift);
    wire [63:0] result3 = fft_dit_butterfly({qr[31:16],qi[31:16]}, {fr[31:16],fi[31:16]}, twiddle3, shift);
    wire [63:0] result4 = fft_dit_butterfly({qr[15:0 ],qi[15:0 ]}, {fr[15:0 ],fi[15:0 ]}, twiddle4, shift);

    assign dit_butterfly_chain = {result1, result2, result3, result4};
}

function [127:0] rev_shuffle ([127:0] input_data)
{
    wire [15:0] ip1 = input_data[127:112];
    wire [15:0] ip2 = input_data[111:96];
    wire [15:0] ip3 = input_data[95:80];
    wire [15:0] ip4 = input_data[79:64];
    wire [15:0] ip5 = input_data[63:48];
    wire [15:0] ip6 = input_data[47:32];
    wire [15:0] ip7 = input_data[31:16];
    wire [15:0] ip8 = input_data[15:0];

    assign rev_shuffle = {ip1, ip3, ip5, ip7, ip2, ip4, ip6, ip8};
}
```

```

function [127:0] shuffle ([127:0] input_data)
{
    wire [15:0] ip1 = input_data[127:112];
    wire [15:0] ip2 = input_data[111:96];
    wire [15:0] ip3 = input_data[95:80];
    wire [15:0] ip4 = input_data[79:64];
    wire [15:0] ip5 = input_data[63:48];
    wire [15:0] ip6 = input_data[47:32];
    wire [15:0] ip7 = input_data[31:16];
    wire [15:0] ip8 = input_data[15:0];

    assign shuffle = {ip1, ip5, ip2, ip6, ip3, ip7, ip4, ip8};
}

function [127:0] real_part([255:0] input)
{
    wire [63:0] part1 = input[255:192];
    wire [63:0] part2 = input[191:128];
    wire [63:0] part3 = input[127:64];
    wire [63:0] part4 = input[63 :0];

    wire [15:0] part1_even  = part1[63:48];
    wire [15:0] part1_odd   = part1[31:16];
    wire [15:0] part2_even  = part2[63:48];
    wire [15:0] part2_odd   = part2[31:16];
    wire [15:0] part3_even  = part3[63:48];
    wire [15:0] part3_odd   = part3[31:16];
    wire [15:0] part4_even  = part4[63:48];
    wire [15:0] part4_odd   = part4[31:16];

    assign real_part =      {part1_even, part1_odd, part2_even, part2_odd,
   part3_even, part3_odd, part4_even, part4_odd};
}

function [127:0] imaginary_part([255:0] input)
{
    wire [63:0] part1 = input[255:192];
    wire [63:0] part2 = input[191:128];
    wire [63:0] part3 = input[127:64];
    wire [63:0] part4 = input[63 :0];

    wire [15:0] part1_even  = part1[47:32];
    wire [15:0] part1_odd   = part1[15:0];
    wire [15:0] part2_even  = part2[47:32];
    wire [15:0] part2_odd   = part2[15:0];
    wire [15:0] part3_even  = part3[47:32];
    wire [15:0] part3_odd   = part3[15:0];
    wire [15:0] part4_even  = part4[47:32];
    wire [15:0] part4_odd   = part4[15:0];

```

```
    assign imaginary_part = {part1_even, part1_odd, part2_even, part2_odd,
    part3_even, part3_odd, part4_even, part4_odd};
}
//DIT STAGES
operation SIMD_DIT_STAGE1
{in AR k, in BR inverse, in BR shift, inout SIMD_REG_128 fr_128, inout
    SIMD_REG_128 fi_128} {}
{
    wire [63:0] tr  = even_part(fr_128);
    wire [63:0] ti  = even_part(fi_128);
    wire [63:0] qr  = odd_part(fr_128);
    wire [63:0] qi  = odd_part(fi_128);

    wire [127:0] twiddle4_reg = twiddle4(0, k, inverse, shift);

    wire [31:0] twiddle = twiddle4_reg[127:96];

    wire [127:0] twiddle_128 = {twiddle, twiddle, twiddle, twiddle};

    wire [255:0] simd_butterfly_result = dit_butterfly_chain(tr, ti, qr, qi,
     twiddle_128, shift);
    wire [127:0] imagValues     = imaginary_part(simd_butterfly_result);
    wire [127:0] realValues     = real_part(simd_butterfly_result);

    assign fr_128 = rev_shuffle(realValues);
    assign fi_128 = rev_shuffle(imagValues);
}

operation SIMD_DIT_STAGE2
{in AR k, in BR inverse, in BR shift, inout SIMD_REG_128 fr_128, inout
    SIMD_REG_128 fi_128}
{}
{
    wire [63:0] tr  = even_part(fr_128);
    wire [63:0] ti  = even_part(fi_128);
    wire [63:0] qr  = odd_part(fr_128);
    wire [63:0] qi  = odd_part(fi_128);

    wire [127:0] twiddle4_reg = twiddle4(0, k, inverse, shift);

    wire [63:0] twiddle = twiddle4_reg[127:64];

    wire [127:0] twiddle_128 = {twiddle[63:32], twiddle[63:32], twiddle
    [31:0], twiddle[31:0]};

    wire [255:0] simd_butterfly_result      = dit_butterfly_chain(tr, ti, qr
    , qi, twiddle_128, shift);
    wire [127:0] imagValues     = imaginary_part(simd_butterfly_result);
    wire [127:0] realValues     = real_part(simd_butterfly_result);

    assign fr_128 = rev_shuffle(realValues);
```

```
    assign fi_128 = rev_shuffle(imagValues);
}

operation SIMD_DIT_STAGE3
{in AR k, in BR inverse, in BR shift, inout SIMD_REG_128 fr_128, inout SIMD_REG_128 fi_128}
{}
{
    wire [63:0] tr  = even_part(fr_128);
    wire [63:0] ti  = even_part(fi_128);
    wire [63:0] qr  = odd_part(fr_128);
    wire [63:0] qi  = odd_part(fi_128);

    wire [127:0] twiddle4_reg = twiddle4(0, k, inverse, shift);

    wire [255:0] simd_butterfly_result = dit_butterfly_chain(tr, ti, qr, qi, twiddle4_reg, shift);

    assign fr_128 = real_part(simd_butterfly_result);
    assign fi_128   = imaginary_part(simd_butterfly_result);
}

operation SIMD_DIT_STAGE_N
{inout SIMD_REG_128 fr_128, inout SIMD_REG_128 fi_128, in BR shift, in AR m, in BR inverse}
{in STATE_K}
{
    wire [63:0] tr  = fr_128[127:64];
    wire [63:0] ti  = fi_128[127:64];
    wire [63:0] qr  = fr_128[63:0];
    wire [63:0] qi  = fi_128[63:0];

    wire [127:0] twiddle4_reg = twiddle4(m, STATE_K, inverse, shift);

    wire [255:0] simd_butterfly_result = dit_butterfly_chain(tr, ti, qr, qi, twiddle4_reg, shift);

    assign fr_128 = rev_shuffle(real_part(simd_butterfly_result));
    assign fi_128 = rev_shuffle(imaginary_part(simd_butterfly_result));
}

// DIF STAGES
operation SIMD_DIF_THIRD_LAST_STAGE
{in AR k, in BR inverse, in BR shift, inout SIMD_REG_128 fr_128, inout SIMD_REG_128 fi_128}
{}
{
    wire [63:0] tr  = even_part(fr_128);
    wire [63:0] ti  = even_part(fi_128);
    wire [63:0] qr  = odd_part(fr_128);
    wire [63:0] qi  = odd_part(fi_128);
```

```

    wire [127:0] twiddle4_reg = twiddle4(0, k, inverse, shift);

    wire [255:0] simd_butterfly_result      = dif_butterfly_chain(tr, ti, qr, qi, twiddle4_reg, shift);
    wire [127:0] imagValues     = imaginary_part(simd_butterfly_result);
    wire [127:0] realValues     = real_part(simd_butterfly_result);

    assign fr_128 = shuffle(realValues);
    assign fi_128 = shuffle(imagValues);
}

operation SIMD_DIF_SECOND_LAST_STAGE
{in AR k, in BR inverse, in BR shift, inout SIMD_REG_128 fr_128, inout SIMD_REG_128 fi_128}
{}
{
    wire [63:0] tr  = even_part(fr_128);
    wire [63:0] ti  = even_part(fi_128);
    wire [63:0] qr  = odd_part(fr_128);
    wire [63:0] qi  = odd_part(fi_128);

    wire [127:0] twiddle4_reg = twiddle4(0, k, inverse, shift);

    wire [63:0] twiddle = twiddle4_reg[127:64];

    wire [127:0] twiddle_128 = {twiddle[63:32], twiddle[63:32], twiddle[31:0], twiddle[31:0]};

    wire [255:0] simd_butterfly_result      = dif_butterfly_chain(tr, ti, qr, qi, twiddle_128, shift);
    wire [127:0] imagValues     = imaginary_part(simd_butterfly_result);
    wire [127:0] realValues     = real_part(simd_butterfly_result);

    assign fr_128 = shuffle(realValues);
    assign fi_128 = shuffle(imagValues);
}

operation SIMD_DIF_LAST_STAGE
{in AR k, in BR inverse, in BR shift, inout SIMD_REG_128 fr_128, inout SIMD_REG_128 fi_128}
{}
{
    wire [63:0] tr  = even_part(fr_128);
    wire [63:0] ti  = even_part(fi_128);
    wire [63:0] qr  = odd_part(fr_128);
    wire [63:0] qi  = odd_part(fi_128);

    wire [127:0] twiddle4_reg = twiddle4(0, k, inverse, shift);

    wire [31:0] twiddle = twiddle4_reg[127:96];
```

```

    wire [127:0] twiddle_128 = {twiddle, twiddle, twiddle, twiddle};

    wire [255:0] simd_butterfly_result = dif_butterfly_chain(tr, ti, qr, qi,
     twiddle_128, shift);
    assign fi_128   = imaginary_part(simd_butterfly_result);
    assign fr_128   = real_part(simd_butterfly_result);
}

operation SIMD_DIF_STAGE_N
{inout SIMD_REG_128 fr_128, inout SIMD_REG_128 fi_128, in BR shift, in AR m,
     in BR inverse}
{in STATE_K}
{
    wire [63:0] tr  = fr_128[127:64];
    wire [63:0] ti  = fi_128[127:64];
    wire [63:0] qr  = fr_128[63:0];
    wire [63:0] qi  = fi_128[63:0];

    wire [127:0] twiddle4_reg = twiddle4(m, STATE_K, inverse, shift);

    wire [255:0] simd_butterfly_result = dif_butterfly_chain(tr, ti, qr, qi,
     twiddle4_reg, shift);

    assign fr_128 = rev_shuffle(real_part(simd_butterfly_result));
    assign fi_128 = rev_shuffle(imaginary_part(simd_butterfly_result));
}


operation FFT_CHECK_SHIFT_CONDITION {in SIMD_REG_128 j, in SIMD_REG_128 m,
    out BR from_shift_check} {}
{
    wire [15:0] j1 = absolute(j[127:112]);
    wire [15:0] j2 = absolute(j[111:96 ]);
    wire [15:0] j3 = absolute(j[95 :80 ]);
    wire [15:0] j4 = absolute(j[79 :64 ]);
    wire [15:0] j5 = absolute(j[63 :48 ]);
    wire [15:0] j6 = absolute(j[47 :32 ]);
    wire [15:0] j7 = absolute(j[31 :16 ]);
    wire [15:0] j8 = absolute(j[15 :0  ]);

    wire [15:0] m1 = absolute(m[127:112]);
    wire [15:0] m2 = absolute(m[111:96 ]);
    wire [15:0] m3 = absolute(m[95 :80 ]);
    wire [15:0] m4 = absolute(m[79 :64 ]);
    wire [15:0] m5 = absolute(m[63 :48 ]);
    wire [15:0] m6 = absolute(m[47 :32 ]);
    wire [15:0] m7 = absolute(m[31 :16 ]);
    wire [15:0] m8 = absolute(m[15 :0  ]);

```

```
    assign from_shift_check =       j1[14:14] | j2[14:14] | j3[14:14] | j4
    [14:14] | j5[14:14] | j6[14:14]
                                |    j7[14:14] | j8[14:14] | m1[14:14] | m2
    [14:14] | m3[14:14] | m4[14:14]
                                |    m5[14:14] | m6[14:14] | m7[14:14] | m8
    [14:14];
}

function [31:0] compute_addr ([31:0] base_address, [31:0] offset)
    slot_shared
{
    assign compute_addr = TIEadd(base_address, {offset[30:0], 1'b0}, 1'b0);
}

function [127:0] dif_ip_reorder ([127:0] input_data)
{
    wire [15:0] ip1 = input_data[127:112];
    wire [15:0] ip2 = input_data[111:96];
    wire [15:0] ip3 = input_data[95:80];
    wire [15:0] ip4 = input_data[79:64];
    wire [15:0] ip5 = input_data[63:48];
    wire [15:0] ip6 = input_data[47:32];
    wire [15:0] ip7 = input_data[31:16];
    wire [15:0] ip8 = input_data[15:0];

    assign dif_ip_reorder = {ip8, ip7, ip6, ip5, ip4, ip3, ip2, ip1};
}

operation FFT_SIMD_LOAD
{in AR *addr, in AR offset, out SIMD_REG_128 output}
{out VAddr, in MemDataIn128}
{
    assign VAddr = compute_addr(addr, offset);
    assign output = dif_ip_reorder(MemDataIn128);
}

operation FFT_SIMD_STORE
{in AR *addr, in AR offset, in SIMD_REG_128 data}
{out VAddr, out MemDataOut128}
{
    assign VAddr                = compute_addr(addr, offset);
    assign MemDataOut128        = dif_ip_reorder(data);
}

operation FFT_SIMD_LOAD_SHUFFLE
{in AR *addr, in AR offset, out SIMD_REG_128 output, in immediate reverse}
{out VAddr, in MemDataIn128}
{
    assign VAddr =  compute_addr(addr, offset);
    wire [127:0] rdin = dif_ip_reorder(MemDataIn128);
    assign output = TIEmux(reverse[0], shuffle(rdin), rev_shuffle(rdin));
```

```
}

operation FFT_SIMD_STORE_SHUFFLE
{in AR *addr, in AR offset, in SIMD_REG_128 input, in immediate reverse}
{out VAddr, out MemDataOut128}
{
    assign VAddr                = compute_addr(addr, offset);
    wire [127:0] shuffled_vec = TIEmux(reverse[0], shuffle(input), rev_shuffle(input));
    assign MemDataOut128        = dif_ip_reorder(shuffled_vec);
}

operation FFT_SIMD_LOAD_IL
{in AR *addr, in AR offset, inout SIMD_REG_128 vec128_1, inout SIMD_REG_128 vec128_2, in immediate load_upper}
{out VAddr, in MemDataIn128}
{
    assign VAddr = compute_addr(addr, offset);
    wire [127:0] mem = dif_ip_reorder(MemDataIn128);
    assign vec128_1 = TIEmux(load_upper[0], {vec128_1[127:64], mem[127:64]}, {mem[127:64], vec128_1[63:0]});
    assign vec128_2 = TIEmux(load_upper[0], {vec128_2[127:64], mem[63:0]}, {mem[63:0], vec128_2[63:0]});
}

operation FFT_SIMD_STORE_IL
{in AR *addr, in AR offset, in SIMD_REG_128 vec128_1, in SIMD_REG_128 vec128_2, in immediate store_upper}
{out VAddr, out MemDataOut128}
{
    assign VAddr = compute_addr(addr, offset);

    wire [63:0] data1 = TIEmux(store_upper[0], vec128_1[63:0], vec128_1[127:64]);
    wire [63:0] data2 = TIEmux(store_upper[0], vec128_2[63:0], vec128_2[127:64]);

    wire [127:0] data = {data1, data2};
    assign MemDataOut128 = dif_ip_reorder(data);
}

format flix64 64 {slot_0, slot_1, slot_2}
slot_opcodes slot_0 {mv.SIMD_REG_128, st.SIMD_REG_128, ld.SIMD_REG_128, FFT_SIMD_LOAD_SHUFFLE, L16SI, S16I, FFT_SIMD_LOAD}
slot_opcodes slot_1 {NOP}
slot_opcodes slot_2 {mv.SIMD_REG_128, st.SIMD_REG_128, ld.SIMD_REG_128, FFT_SIMD_LOAD_SHUFFLE, L16SI, S16I, FFT_SIMD_LOAD}

table SIN_LUT 16 512 {
      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,  32766,  32764,  32761,  32757,  32751,  32744,  32736,
 32727,  32717,  32705,  32692,  32678,  32662,  32646,  32628,
 32609,  32588,  32567,  32544,  32520,  32495,  32468,  32441,
 32412,  32382,  32350,  32318,  32284,  32249,  32213,  32176,
 32137,  32097,  32056,  32014,  31970,  31926,  31880,  31833,
 31785,  31735,  31684,  31633,  31580,  31525,  31470,  31413,
 31356,  31297,  31236,  31175,  31113,  31049,  30984,  30918,
 30851,  30783,  30713,  30643,  30571,  30498,  30424,  30349,
 30272,  30195,  30116,  30036,  29955,  29873,  29790,  29706,
 29621,  29534,  29446,  29358,  29268,  29177,  29085,  28992,
 28897,  28802,  28706,  28608,  28510,  28410,  28309,  28208,
 28105,  28001,  27896,  27790,  27683,  27575,  27466,  27355,
 27244,  27132,  27019,  26905,  26789,  26673,  26556,  26437,
 26318,  26198,  26077,  25954,  25831,  25707,  25582,  25456,
 25329,  25201,  25072,  24942,  24811,  24679,  24546,  24413,
 24278,  24143,  24006,  23869,  23731,  23592,  23452,  23311,
 23169,  23027,  22883,  22739,  22594,  22448,  22301,  22153,
 22004,  21855,  21705,  21554,  21402,  21249,  21096,  20942,
 20787,  20631,  20474,  20317,  20159,  20000,  19840,  19680,
 19519,  19357,  19194,  19031,  18867,  18702,  18537,  18371,
 18204,  18036,  17868,  17699,  17530,  17360,  17189,  17017,
```

```
    16845,   16672,   16499,   16325,   16150,   15975,   15799,   15623,
    15446,   15268,   15090,   14911,   14732,   14552,   14372,   14191,
    14009,   13827,   13645,   13462,   13278,   13094,   12909,   12724,
    12539,   12353,   12166,   11980,   11792,   11604,   11416,   11227,
    11038,   10849,   10659,   10469,   10278,   10087,    9895,    9703,
     9511,    9319,    9126,    8932,    8739,    8545,    8351,    8156,
     7961,    7766,    7571,    7375,    7179,    6982,    6786,    6589,
     6392,    6195,    5997,    5799,    5601,    5403,    5205,    5006,
     4807,    4608,    4409,    4210,    4011,    3811,    3611,    3411,
     3211,    3011,    2811,    2610,    2410,    2209,    2009,    1808,
     1607,    1406,    1206,    1005,     804,     603,     402,     201
}
```

main.c

```
#include        "fft.h"
#include <xtensa/tie/tie_fft.h>
#include        <stdio.h>
#include        <math.h>

#define DO_FFT_DIT  1
#define DO_FFT_DIF 1

#define M       3

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

fixed real[N], imag[N];

int main()
{
    int i;

    for(i=0; i<N; i++)
    {
        real[i] = 1000*cos(i*2*3.1415926535/N);
        imag[i] = 0;
    }

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


    //FFT DIT
```

```
#if (DO_FFT_DIT==1)
    fix_fft_dit(real, imag, M, 0);

    printf("\nFFT DIT\n");
    for (i=0; i<N; i++)
    {
        printf("%d: %d, %d\n", i, real[i], imag[i]);
    }

#else

    //IFFT DIT
    fix_fft_dit(real, imag, M, 1);

    printf("\nIFFT DIT\n");
    for (i=0; i<N; i++)
    {
        printf("%d: %d, %d\n", i, real[i], imag[i]);
    }

#endif

    //FFT DIF
#if (DO_FFT_DIF==1)
    fix_fft_dif(real, imag, M, 0);

    printf("\nFFT DIF\n");
    for (i=0; i<N; i++)
    {
        printf("%d: %d, %d\n", i, real[i], imag[i]);
    }

#else

    //IFFT DIF
    fix_fft_dif(real, imag, M, 1);

    printf("\nIFFT DIF\n");
    for (i=0; i<N; i++)
    {
        printf("%d: %d, %d\n", i, real[i], imag[i]);
    }

#endif

    return 0;
}
```

Questions

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

- SIMD: Improved performance by implementing SIMD technique for efficient loading, storing, and computation of multiple data elements in a single instruction.
- FLIX: Improved hardware efficiency by implementing Flix technique for executing multiple instructions in parallel.
- LookUp Table: Leveraging the symmetry of the sine function to reduce the size of the hardware lookup table and enhance faster data retrieval.

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

- FFT Shift Check: Improved performance achieved by optimizing value range comparison in C code with hardware acceleration.
- FFT Twiddle factor calculation: Optimizing the computation of twiddle factors using TIE code from C. Hardware efficiency is improved for use in SIMD and Flix, resulting in enhanced performance.
- FFT Butterfly node computation: Hardware-accelerated computation of butterfly nodes in TIE code to enable parallel processing of multiple nodes, resulting in improved performance.
- Load/Store: Hardware acceleration of load/store address computation, along with the use of two Flix slots to load and store two 128-bit data in a single instruction

What is the impact on execution time (speedup)?

Execution time in terms of Cycle Count			
Configuration	**Unoptimized** C implementation (-o3)	**Optimized** TIE implementation using **DIT** (-o3)	**Optimized** TIE implementation using **DIF** (-o3)
FFT (M = 3; N = 8)	3,156	685 (*Speed Up = 4.6*)	714 (*Speed Up = 4.4*)
FFT (M = 8; N = 256)	224,562	44,892 (*Speed Up = 5*)	44,718 (*Speed Up = 5*)
FFT (M = 10; N = 1024)	9,667,196	211,202 (*Speed Up = 45*)	208,328 (*Speed Up = 46*)
IFFT (M = 3; N = 8)	4,146	726 (*Speed Up = 5.7*)	754 (*Speed Up = 5.4*)
IFFT (M = 8; N = 256)	288,185	48,718 (*Speed Up = 5.9*)	50,058 (*Speed Up = 5.7*)
IFFT (M = 10; N = 1024)	1,345,047	2226,874 (*Speed Up = 5.9*)	235,057 (*Speed Up = 5.7*)

What is the impact on the hardware (area)?

Our TIE area utilizes approximately 85,206 gates for DIT or DIF, with 4,988 for decode, muxing, and other functions and 80,218 for instructions, states, and register files.

When both DIT and DIF are used simultaneously, only 106,760 gates are needed due to shared resources.

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

- Introducing new instructions resulted in implementing algorithm parts in TIE to help improve speedup.
- The inner loop of the FFT algorithm is restructured to fit the memory layout necessary for parallel load/store instructions.
- Compiler optimization is set to -o3, the highest optimisation level done by the compiler using advanced techniques. Also, the generated binary is smaller in size.

Detailed Report

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.*Step-by-step description* ***(min 1. page, max 3 pages)*** *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 line numbers*
- *How did the performance increase (or decrease)? If you obtain a performance decrease, explain why and due to which acceleration method.*
- *What are the costs for the performance increase?*
- *Make a tradeoff 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.

This report describes the implementation of FFT and IFFT (inverse FFT) with Decimation-in-time (DIT) and decimation-in-frequency (DIF) networks. Performance analysis of the designs at progressive stages has been carried out. The optimized DIF and DIT implementations work for generic N points. The design goal for our implementation of the FFT kernel is low latency. Significant tradeoffs with area and power are expected from the beginning.

Profiling the provided C implementation of the FFT filter revealed that the two innermost for-loops with the butterfly computation chain are the major hotspots in the program. It is clearly evident that optimizing these loops with hardware acceleration techniques would enhance the speed of the FFT kernel significantly.

For the design's performance analysis, FFT- DIT with M = 8 (N = 256) has been selected with -an O3 optimization level. Before optimization, the FFT kernel consumes 224,562 clock cycles. We started our optimization by replacing the arithmetic operations in the loops with predefined TIE instructions. This way, the implementation of the arithmetic operations is optimized in hardware, and a noticeable speedup is achieved. However, this speedup was not significant enough to qualify as an optimization step, so the loop itself had to be optimized.

Then the entire butterfly chain was replaced with hardware-optimized TIE instructions (TIE_MUX, TIE_ADD, etc.) in order to reduce the theoretical complexity of the function from O(N) to O(1). This optimization provided a considerable speedup of approximately 2.5, and the loop component in the software was removed.

We also optimized the computation of the twiddle factors required for each stage of the butterflies using a Lookup table for the sin function. We realized that the sine table had redundant samples. By making use of the symmetry of the provided sin array, we reduced it to half of its original size (from 1024 samples to 512 samples). We also tried reducing it further by a factor of 2 (to 256 samples), but that led to certain complications affecting the functional correctness of the FFT kernel. So, in turn, we used a Lookup table of 512 entries. In some cases (M=3), for the first two stages, we tried hard coding the twiddle factors.

Our next approach was to make use of SIMD operations. By exploiting data level parallelism in the FFT kernel through SIMD instruction, the performance of the FFT kernel could be improved significantly. We used SIMD operations, namely FFT_SIMD_LOAD (TIE: 429 -– 443) and FFT_SIMD_STORE_SHUFFLE (TIE: 454 -– 461) to simultaneously load 8 input data and store the computed results in a shuffled order. We then focused on implementing multiple stages of the FFT kernel by making use of the multiple data available in a single vector.

Adapting the SIMD approach to compute the correct butterfly results required a thorough understanding of the C code. We realize that to adopt a generic 4-way SIMD method to calculate different stages of the FFT-DIT kernel, we require a minimum size of M = 3 (N = 8). Hence, to support lower data points, we implement customized hardware units that account for the first 3 stages of the FFT algorithm (TIE: 225 - 286). To implement the kernel for larger data points (M>3), we created hardware units defined in [TIE: 287 – 302, 463 – 483].

For further optimization, we have used a Flix64 format instruction with three slots (slot_0, slot_1, slot_2) and their corresponding slot opcodes (TIE: 486 - 489). Each slot opcode specifies the operation to be performed on the data in the corresponding slot. This way we could execute multiple FLIX instructions in parallel, improving the throughput significantly. Furthermore, we optimized the segment of the code, which computes whether scaling is required. For this, we repeatedly load multiple input values into the SIMD registers and calculate their absolute values using hardware, which in turn determines whether a shift is necessary (TIE: 385 - 408).

Thus, after all the above-mentioned optimization strategies, for M = 8 (N = 256), we could observe that the number of clock cycles required reduced from the initial 224,562 to 44,892. Thus, indicating a speedup of five. In terms of hardware costs, the utilization is 85,206 gates, out of which 4,988 are for decoding, muxing, and the rest are for instructions, states, and register files. These figures remain the same for larger M values as a result of hardware reuse. Similarly, for M = 10 (N = 1024 points), the total number of cycles reduced from 9,667,196 (unoptimized C program) to 211,202 (speedup = 45). From the above results, we can see that as the problem definition increases (for larger values of N), the speedup increases exponentially.

As far as the Decimation-in-frequency (DIF) is concerned, we modified the provided DIT C program by changing the computational part of the FFT butterfly, shuffling of outputs, etc. For M = 8 (N = 256), the pure C implementation consumes 224,562 clock cycles. By using similar hardware accelerating techniques as in the case of DIT, we could notice a speedup of 5 (44,718 clock cycles). Problem scaling to larger M values reveals that DIT and DIF provide relatively similar speedup results.

Summary: *The main objective of the implementation is to improve the performance of the FFT algorithm, focusing on optimizing the computation of butterfly nodes and the calculation of twiddle factors, which are identified as the major hotspots in the algorithm. To achieve this objective, hardware acceleration techniques like FLIX, SIMD, etc., are used to enhance these spots, thus resulting in the improved overall performance of the FFT algorithm.*

In conclusion, the best combination of software optimization and hardware acceleration strategies includes using SIMD and FLIX to improve throughput using parallelism. In this task, we have gained experience in analyzing the performance of FFT and IFFT algorithms using hardware architectures by identifying bottlenecks and opportunities for optimization and implementing and testing various optimization and hardware acceleration strategies.

Future Scope:

- *Reduction of SIN Lookup table size to 256.*
- *Hardware reordering of inputs/outputs.*
- *Pipelining the critical paths to improve the design throughput.*
- *Increasing slots to accommodate more custom FLIX instructions.*