



# 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 finite 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.

*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>

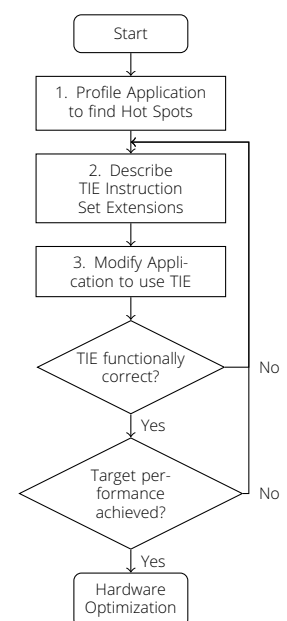


Figure 1: Similar to Tensilica Instruction Extension (TIE) Language, User's Guide. 02/2014, p. 4.

## 2 Source Code

### 2.1 Source code C files

fft.h

```
1  #ifndef FFT_H
2  #define FFT_H
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include <math.h>
6  #include <xtensa/tie/fft_tie.h>
7
8  /* FIX_MPY() - fixed-point multiplication macro.
9   This macro is a statement, not an expression (uses asm).
10  BEWARE: make sure _DX is not clobbered by evaluating (A) or DEST.
11  args are all of type fixed.
12  Scaling ensures that 32767*32767 = 32767. */
13 #define FIX_MPY(DEST,A,B)      DEST = ((long)(A) * (long)(B))>>15
14
15 #define N_WAVE          1024      /* dimension of Sinewave[] */
16 #define LOG2_N_WAVE     10        /* log2(N_WAVE) */
17
18 typedef int16_t fixed;
19 extern fixed Sinewave[N_WAVE];
20
21 //function prototypes
22 fixed fix_mpy(fixed a, fixed b);
23 int fix_fft(fixed *fr, fixed *fi, int m, int inverse);
24 int fix_fft_dif(fixed *fr, fixed *fi, int m, int inverse);
25
26 typedef struct cplx {
27     fixed R;
28     fixed I;
29 } cplx;
30
31 int fft_adv_dif(cplx*__restrict f, int m,
32                const cplx* __restrict coeffs);
33
```

```

34 int fft_adv_dif(cplx*__restrict f, int m,
35               const cplx* __restrict coeffs);
36
37 int fft_adv_dit(cplx*__restrict f, int m,
38               const cplx* __restrict coeffs);
39
40 int fft_adv_dit_inv(cplx*__restrict f, int m,
41                   const cplx* __restrict coeffs);
42
43 #endif //FFT_H

```

main.c

```

1  //main.c
2
3  #include      "fft.h"
4  #include      <stdio.h>
5  #include      <math.h>
6
7  //change the type to change the test to be done
8  #define FFT_Type    5
9
10 #define M          3
11
12 //number of points
13 #define N          (1<M)
14
15 fixed real[N], imag[N], real_fix[N], imag_fix[N], real_mul[N], imag_mul[N];
16 //for Backward propagation
17 static fixed bwd_real_mul[N], bwd_imag_mul[N], bwd_real_fix[N], bwd_imag_fix
    [N];
18 static cplx f[N], bwd_f[N];
19 bool equal;
20
21 static cplx fwd_coeffs[N / 2];
22 static cplx bwd_coeffs[N / 2];
23
24 int Test_fft_ref() { //Test for reference FFT and IFFT
25
26     fix_fft(real_fix, imag_fix, M, 0);
27     memcpy(bwd_real_fix, real_fix, sizeof(real_fix));
28     memcpy(bwd_imag_fix, imag_fix, sizeof(imag_fix));
29     fix_fft(bwd_real_fix, bwd_imag_fix, M, 1);
30
31 }
32
33 int Test_fft_adv_dit() {
34
35     fft_adv_dit(f, M, fwd_coeffs);
36     memcpy(bwd_f, f, sizeof(f));
37     fft_adv_dit_inv(bwd_f, M, bwd_coeffs);

```

```

38 }
39
40 int Test_only_fft_adv_dit() {
41     fix_fft(real_fix, imag_fix, M, 0);
42     fft_adv_dit(f, M, fwd_coeffs);
43 }
44
45 int Test_only_fft_adv_dit_inv() {
46     fix_fft(real_fix, imag_fix, M, 1);
47     fft_adv_dit_inv(f, M, bwd_coeffs);
48 }
49
50 int Test_only_fft_adv_dif() {
51     fft_adv_dif(f, M, fwd_coeffs);
52     fix_fft_dif(real_fix, imag_fix, M, 0);
53 }
54
55 int main() {
56     int i;
57     //precalculation of Coefficients
58     GenerateCoefficients(fwd_coeffs, N / 2, false);
59     GenerateCoefficients(bwd_coeffs, N / 2, true);
60     //Generating the input data
61     for (i = 0; i < N; i++) {
62         real[i] = 1000 * cos(i * 2 * 3.1415926535 / N);
63         imag[i] = 0;
64         real_fix[i] = real[i];
65         imag_fix[i] = imag[i];
66         real_mul[i] = real[i];
67         imag_mul[i] = imag[i];
68
69         f[i].R = real[i];
70         f[i].I = imag[i];
71     }
72
73
74     printf("\nInput Data\n");
75     for (i = 0; i < N; i++) {
76         printf("%d: %d, %d\n", i, real[i], imag[i]);
77     }
78     printf("\nInput Data Complex\n");
79     for (i = 0; i < N; i++) {
80         printf("%d: %d, %d\n", i, f[i].R, f[i].I);
81     }
82
83
84 #if (FFT_Type==0)//Test reference C code
85     fix_fft(real, imag, M, 0);
86
87     printf("\nFFT pure C\n");
88     for (i = 0; i < N; i++) {

```

```

89         printf("%d: %d, %d\n", i, real[i], imag[i]);
90     }
91
92     #elif (FFT_Type==1)//Test FFT2 and reference
93         fix_fft(real_fix, imag_fix, M, 0);
94
95         printf("\nFFT pure C\n");
96         for (i = 0; i < N; i++) {
97             printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
98         }
99
100         fft_FFT2(real_mul, imag_mul, M, 0);
101
102         printf("\nFFT TIE Node\n");
103         for (i = 0; i < N; i++) {
104             printf("%d: %d, %d\n", i, real_mul[i], imag_mul[i]);
105         }
106
107     #elif (FFT_Type==2)//Test DIT FFT and IFFT
108         Test_fft_ref();
109         Test_fft_adv_dit();
110
111         equal = true;
112         for (i = 0; i < N; i++) { //Compare the calculated values and generate
113             print
114                 if(f[i].R != real_fix[i] || f[i].I != imag_fix[i])
115                 {
116                     equal = false;
117                     printf("\nCorrect C value\n");
118                     printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
119                     printf("Wrong TIE value\n");
120                     printf("%d: %d, %d\n", i, f[i].R, f[i].I);
121                 }
122                 if(bwd_f[i].R != bwd_real_fix[i] || bwd_f[i].I != bwd_imag_fix[i])
123                 {
124                     equal = false;
125                     printf("\nCorrect C value\n");
126                     printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
127                     printf("Wrong TIE value\n");
128                     printf("%d: %d, %d\n", i, bwd_f[i].R, bwd_f[i].I);
129                 }
130         }
131         if(equal) {
132             printf("FFT Pass");
133         }
134         else {
135             printf("FFT Fail");
136         }
137
138     #elif(FFT_Type == 3)//Test DIT FFT
139         Test_only_fft_adv_dit();

```

```

139
140     equal = true;
141     for (i = 0; i < N; i++) {
142         if(f[i].R != real_fix[i] || f[i].I != imag_fix[i])
143         {
144             equal = false;
145             printf("\nCorrect C value\n");
146             printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
147             printf("Wrong TIE value\n");
148             printf("%d: %d, %d\n", i, f[i].R, f[i].I);
149         }
150     }
151     if(equal) {
152         printf("FFT Pass");
153     }
154     else {
155         printf("FFT Fail");
156     }
157
158 #elif(FFT_Type == 4)////Test DIT IFFT
159     Test_only_fft_adv_dit_inv();
160     equal = true;
161     for (i = 0; i < N; i++) {
162         if (f[i].R != real_fix[i] || f[i].I != imag_fix[i]) {
163             equal = false;
164             printf("\nCorrect C value\n");
165             printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
166             printf("Wrong TIE value\n");
167             printf("%d: %d, %d\n", i, f[i].R, f[i].I);
168         }
169     }
170     if (equal) {
171         printf("FFT Pass");
172     } else {
173         printf("FFT Fail");
174     }
175
176 #elif(FFT_Type == 5) //Test DIF FFT
177     Test_only_fft_adv_dif();
178     equal = true;
179     for (i = 0; i < N; i++) {
180         if (f[i].R != real_fix[i] || f[i].I != imag_fix[i]) {
181             equal = false;
182             printf("\nCorrect C value\n");
183             printf("%d: %d, %d\n", i, real_fix[i], imag_fix[i]);
184             printf("Wrong TIE value\n");
185             printf("%d: %d, %d\n", i, f[i].R, f[i].I);
186         }
187     }
188     if (equal) {
189         printf("FFT Pass");

```

```

190     } else {
191         printf("FFT Fail");
192     }
193
194 #endif
195
196     return 0;
197 }

```

## GenerateCoefficient.c

```

1     #include "fft.h"
2
3     static inline cplx
4     W(const uint32_t k, const uint32_t N) //define the angle
5     {
6         const double angle = M_PI * k / N;
7         const cplx res = { (fixed) (INT16_MAX * cos(angle)), (fixed) (-sin(angle)
8         ) * INT16_MAX) };
9         return res;
10    }
11
12    void
13    GenerateCoefficients(cplx* __restrict out_coeff, const uint32_t N, bool
14        inverse) //generate the coefficient before calculation
15    {
16        const fixed factor = inverse ? -1 : 1;
17        uint32_t i = 0;
18        for (; i < N; ++i)
19        {
20            cplx val = W(i, N);
21            val.I = (fixed) (val.I * factor);
22            out_coeff[i] = val;
23        }
24    }

```

## fft\_adv.c

```

1     #include "fft.h"
2
3     static inline uint32_t BitReverse(uint32_t n) {
4         n = (n & 0xffff0000) >> 16 | (n & 0x0000ffff) << 16;
5         n = (n & 0xff00ff00) >> 8 | (n & 0x00ff00ff) << 8;
6         n = (n & 0xf0f0f0f0) >> 4 | (n & 0x0f0f0f0f) << 4;
7         n = (n & 0xcccccccc) >> 2 | (n & 0x33333333) << 2;
8         n = (n & 0xaaaaaaaa) >> 1 | (n & 0x55555555) << 1;
9         return n;
10    }
11
12    uint32_t Prepare_Data(cplx* __restrict const x, const uint32_t N,
13        const uint32_t lg2_N, const uint32_t a_start) {

```



```

14 // Decimate
15 {
16     const uint32_t shift_amount = 32 - lg2_N;
17     uint32_t a = a_start;
18     uint32_t c = 0;
19     for (; a < N + a_start; ++a) {
20         c = a;
21         FFT_BIT_REVERSE(c);
22         const uint32_t b = c >> shift_amount;
23         const bool swap = a < b;
24         if (swap) {
25             const cplx tmp_a = x[a];
26             x[a] = x[b];
27             x[b] = tmp_a;
28         }
29     }
30 }
31 return 0;
32 }
33
34 uint32_t fft_exec_dit(const intptr_t f, const intptr_t w, const uint32_t n,
35     const uint32_t lg2_n, int shift) {
36     int i, done, l, even_i, step, debug_reg, odd_ix;
37     done = 0;
38     WUR_ptr_data(f);
39     WUR_ptr_w(w);
40     WUR_n(n);
41     WUR_lg2_n(lg2_n);
42     WUR_shift(shift);
43
44     WUR_time_decimation(1);
45
46     FFT_INIT();
47     while (FFT_UPDATE() == 0) {
48
49         FFT_LOAD_EVEN();
50         FFT_LOAD_ODD();
51         FFT_LOAD_W();
52
53         FFT_LOAD_EVEN();
54         FFT_LOAD_ODD();
55         FFT_LOAD_W();
56
57         FFT_LOAD_EVEN();
58         FFT_LOAD_ODD();
59         FFT_LOAD_W();
60
61         FFT_LOAD_EVEN();
62         FFT_LOAD_ODD();
63         FFT_LOAD_W();
64

```

```

65     FFT_8_FFT_DIT();
66
67     FFT_STORE_EVEN();
68     FFT_STORE_ODD();
69
70     FFT_STORE_EVEN();
71     FFT_STORE_ODD();
72
73     FFT_STORE_EVEN();
74     FFT_STORE_ODD();
75
76     FFT_STORE_EVEN();
77     FFT_STORE_ODD();
78
79     }
80 }
81
82 uint32_t fft_exec_dif(const intptr_t f, const intptr_t w, const uint32_t n,
83     const uint32_t lg2_n, int shift) {
84     int i, done, l, even_i, step, debug_reg, odd_ix;
85     done = 0;
86     WUR_ptr_data(f);
87     WUR_ptr_w(w);
88     WUR_n(n);
89     WUR_lg2_n(lg2_n);
90     WUR_shift(shift);
91
92     WUR_time_decimation(0);
93
94     FFT_INIT();
95     while (FFT_UPDATE() == 0) {
96
97         FFT_LOAD_EVEN();
98         FFT_LOAD_ODD();
99         FFT_LOAD_W();
100
101         FFT_LOAD_EVEN();
102         FFT_LOAD_ODD();
103         FFT_LOAD_W();
104
105         FFT_LOAD_EVEN();
106         FFT_LOAD_ODD();
107         FFT_LOAD_W();
108
109         FFT_LOAD_EVEN();
110         FFT_LOAD_ODD();
111         FFT_LOAD_W();
112
113         FFT_8_FFT_DIF();
114
115         FFT_STORE_EVEN();

```

```

116     FFT_STORE_ODD();
117
118     FFT_STORE_EVEN();
119     FFT_STORE_ODD();
120
121     FFT_STORE_EVEN();
122     FFT_STORE_ODD();
123
124     FFT_STORE_EVEN();
125     FFT_STORE_ODD();
126
127 }
128 }
129
130 int fft_adv_dit(cplx*__restrict f, int m,
131     const cplx* __restrict coeffs) {
132     int mr, nn, i, j, l, k, r, istep, n, scale, shift;
133     //number of input data
134     n = 1 << m;
135
136     cplx q; //even input
137     cplx t; //odd input
138     cplx u; //even output
139     cplx v; //odd output
140     cplx w[n / 2]; //twiddle factor
141     uint32_t lg2_n = m;
142
143     if (n > N_WAVE)
144         return -1;
145
146     mr = 0;
147     nn = n - 1;
148     scale = 0;
149     r = 0;
150
151     /* decimation in time - re-order data */
152
153     Prepare_Data(f, n, m, 0);
154     /* fixed scaling, for proper normalization -
155     there will be log2(n) passes, so this
156     results in an overall factor of 1/n,
157     distributed to maximize arithmetic accuracy. */
158     shift = 1;
159
160     fft_exec_dit((intptr_t) f, (intptr_t) coeffs, n, lg2_n, shift);
161
162     return scale;
163 }
164
165 int fft_adv_dit_inv(cplx*__restrict f, int m,
166     const cplx* __restrict coeffs) {

```

```

167     int mr, nn, i, j, l, k, r, istep, n, scale, shift;
168     //number of input data
169     n = 1 << m;
170
171     cplx q; //even input
172     cplx t; //odd input
173     cplx u; //even output
174     cplx v; //odd output
175     cplx w[n / 2]; //twiddle factor
176     uint32_t lg2_n = m;
177
178     if (n > N_WAVE)
179         return -1;
180
181     mr = 0;
182     nn = n - 1;
183     scale = 0;
184     r = 0;
185
186     /* decimation in time - re-order data */
187
188     Prepare_Data(f, n, m, 0);
189     /* variable scaling, depending upon data */
190     shift = 0;
191     for (i = 0; i < n; ++i) {
192         j = f[i].R;
193         if (j < 0)
194             j = -j;
195
196         m = f[i].I;
197         if (m < 0)
198             m = -m;
199
200         if (j > 16383 || m > 16383) {
201             shift = 1;
202             break;
203         }
204     }
205     if (shift)
206         ++scale;
207     fft_exec_dit((intptr_t) f, (intptr_t) coeffs, n, lg2_n, shift);
208     return scale;
209 }
210
211 int fft_adv_dif(cplx*__restrict f, int m,
212               const cplx* __restrict coeffs) {
213     int mr, nn, i, j, l, k, r, istep, n, scale, shift;
214     //number of input data
215     n = 1 << m;
216
217     cplx q; //even input

```

```

218     cplx t; //odd input
219     cplx u; //even output
220     cplx v; //odd output
221     cplx w[n / 2]; //twiddle factor
222     uint32_t lg2_n = m;
223
224     if (n > N_WAVE)
225         return -1;
226
227     mr = 0;
228     nn = n - 1;
229     scale = 0;
230     r = 0;
231
232     /* decimation in time - re-order data */
233     /* fixed scaling, for proper normalization -
234     there will be log2(n) passes, so this
235     results in an overall factor of 1/n,
236     distributed to maximize arithmetic accuracy. */
237     shift = 1;
238     fft_exec_dif((intptr_t) f, (intptr_t) coeffs, n, lg2_n, shift);
239     Prepare_Data(f, n, m, 0);
240
241     return scale;
242 }

```

fft\_tie.tie

```

1 state w_r      16 add_read_write
2 state w_i      16 add_read_write
3
4
5 format fft_flix 64 {fft_slot0, fft_slot1}
6 slot_opcodes fft_slot0 {FFT_LOAD_EVEN, FFT_STORE_EVEN}
7 slot_opcodes fft_slot1 {FFT_LOAD_ODD, FFT_STORE_ODD}
8
9
10
11
12 state a_r      16 add_read_write
13 state a_i      16 add_read_write
14 state b_r      16 add_read_write
15 state b_i      16 add_read_write
16 state u_r      16 add_read_write
17 state u_i      16 add_read_write
18 state v_r      16 add_read_write
19 state v_i      16 add_read_write
20 state q_r      16 add_read_write
21 state q_i      16 add_read_write
22
23

```

```

24 state wv          128 add_read_write
25
26 state av          128 add_read_write
27
28 state bv          128 add_read_write
29
30 state done         1 add_read_write
31 state shift        1 add_read_write
32 state even_ix      16 add_read_write
33 state odd_ix       16 add_read_write
34 state even_const   16 add_read_write
35
36 state w_ix         16 add_read_write
37 state w_inc        16 add_read_write
38 state m_ix         16 add_read_write
39 state l            16 add_read_write
40 state n            16 add_read_write
41 state n_2          16 add_read_write
42 state lg2_n        16 add_read_write
43 state step         16 add_read_write
44 state ptr_data     32 add_read_write
45 state ptr_w        32 add_read_write
46 state debug_reg    32 add_read_write
47 state time_decimation 1 add_read_write
48
49
50 state qv_r         64 add_read_write
51 state qv_i         64 add_read_write
52
53
54 function [31:0]COMPLEX_MUL ([15:0] a_r, [15:0] b_r, [15:0] a_i, [15:0] b_i)
55 {
56     wire [31:0] product1r = TIEmul(a_r[15:0],b_r[15:0],1'b1); // signed
multiplication
57     wire [31:0] product2r = TIEmul(a_r[15:0],b_i[15:0],1'b1); // signed
multiplication
58     wire [31:0] product1i = TIEmul(a_i[15:0],b_i[15:0],1'b1); // signed
multiplication
59     wire [31:0] product2i = TIEmul(a_i[15:0],b_r[15:0],1'b1); // signed
multiplication
60     wire [15:0] mult1 = product1r[30:15] - product1i[30:15];
61     wire [15:0] mult2 = product2r[30:15] + product2i[30:15];
62     assign COMPLEX_MUL = {mult2, mult1};
63 }
64
65
66
67
68 function [15:0]LEFT_SHIFT ([15:0] a)
69 {
70     wire [15:0] shift_temp = {a[14:0], 1'b0};

```

```

71     assign LEFT_SHIFT = shift_temp;
72 }
73
74 function [15:0]LEFT_SHIFT_SIGNED ([15:0] a)
75 {
76     wire [15:0] shift_temp = a[15] ? {a[15], a[13:0], 1'b0} : {a[15], a
[13:0], 1'b1};
77     assign LEFT_SHIFT_SIGNED = shift_temp;
78 }
79
80 function [15:0]RIGHT_SHIFT ([15:0] a)
81 {
82     wire [15:0] shift_temp = {a[15], a[15:1]};
83     assign RIGHT_SHIFT = shift_temp;
84 }
85
86 function [15:0]RIGHT_SHIFT_SIGNED ([15:0] a)
87 {
88     wire [15:0] shift_temp = {a[15], a[15:1]};
89     assign RIGHT_SHIFT_SIGNED = shift_temp;
90 }
91
92 function [31:0]LEFT_SHIFT_32 ([31:0] a)
93 {
94     wire [31:0] shift_temp = {a[30:0], 1'b0};
95     assign LEFT_SHIFT_32 = shift_temp;
96 }
97
98 function [31:0]RIGHT_SHIFT_32 ([31:0] a)
99 {
100     wire [31:0] shift_temp = {1'h0, a[31:1]};
101     assign RIGHT_SHIFT_32 = shift_temp;
102 }
103
104
105
106
107 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)
108 {
109
110     wire [15:0] v_r = b_r + a_r;
111     wire [15:0] v_i = b_i + a_i;
112
113     wire [15:0] v_r_shift = RIGHT_SHIFT_SIGNED(v_r);
114     wire [15:0] v_i_shift = RIGHT_SHIFT_SIGNED(v_i);
115
116     wire [15:0] t_r = b_r - a_r;
117     wire [15:0] t_i = b_i - a_i;
118
119     wire [31:0] com_mul = COMPLEX_MUL(t_r[15:0], w_r[15:0], t_i[15:0], w_i

```

```

[15:0]);
120   wire [15:0] u_r = com_mul[15:0];
121   wire [15:0] u_i = com_mul[31:16];
122
123   assign FFT_NODE_DIF = shift ? {v_i_shift, v_r_shift, u_i, u_r} : {v_i,
v_r, u_i, u_r};
124
125 }
126
127 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)
128 {
129
130   wire [31:0] com_mul = COMPLEX_MUL(a_r[15:0], w_r[15:0], a_i[15:0], w_i
[15:0]);
131   wire [15:0] t_r = com_mul[15:0];
132   wire [15:0] t_i = com_mul[31:16];
133
134   wire [15:0] u_r = b_r - t_r;
135   wire [15:0] u_i = b_i - t_i;
136   wire [15:0] v_r = b_r + t_r;
137   wire [15:0] v_i = b_i + t_i;
138   assign FFT_NODE_DIT = {v_i, v_r, u_i, u_r};
139
140 }
141
142
143
144
145
146 /*
147  * Complex mult
148  */
149
150 operation FFT_COM_MUL {inout AR a_r, in AR b_r, inout AR a_i, in AR b_i}{}
151 {
152   wire [31:0] com_mul = COMPLEX_MUL(a_r[15:0], b_r[15:0], a_i[15:0], b_i
[15:0]);
153   wire [15:0] t_r = com_mul[15:0];
154   wire [15:0] t_i = com_mul[31:16];
155
156   assign a_r = t_r;
157   assign a_i = t_i;
158 }
159
160
161
162
163
164 operation FFT_2_f_LD {}
165   {in b_r, in b_i, out q_r, out q_i}

```



```

166 {
167
168     assign q_r = b_r;
169     assign q_i = b_i;
170 }
171
172
173
174
175 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}
176 {
177
178     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]);
179     wire [15:0] u_r_t = fft_node[15:0];
180     wire [15:0] u_i_t = fft_node[31:16];
181     wire [15:0] v_r_t = fft_node[47:32];
182     wire [15:0] v_i_t = fft_node[63:48];
183
184     assign u_r = u_r_t;
185     assign u_i = u_i_t;
186     assign v_r = v_r_t;
187     assign v_i = v_i_t;
188
189 }
190
191 operation FFT_8_FFT_DIT {}{inout av, in wv, inout bv, out even_ix, in
    even_const, in step, out odd_ix}
192 {
193     assign even_ix = even_const;
194     assign odd_ix = even_const + step;
195
196     wire [15:0] a_r0 = av[15:0];
197     wire [15:0] a_i0 = av[31:16];
198     wire [15:0] a_r1 = av[47:32];
199     wire [15:0] a_i1 = av[63:48];
200     wire [15:0] a_r2 = av[79:64];
201     wire [15:0] a_i2 = av[95:80];
202     wire [15:0] a_r3 = av[111:96];
203     wire [15:0] a_i3 = av[127:112];
204
205
206     wire [15:0] b_r0 = bv[15:0];
207     wire [15:0] b_i0 = bv[31:16];
208     wire [15:0] b_r1 = bv[47:32];
209     wire [15:0] b_i1 = bv[63:48];
210     wire [15:0] b_r2 = bv[79:64];
211     wire [15:0] b_i2 = bv[95:80];
212     wire [15:0] b_r3 = bv[111:96];
213     wire [15:0] b_i3 = bv[127:112];

```

```

214
215     wire [15:0] w_r0 = wv[15:0];
216     wire [15:0] w_i0 = wv[31:16];
217     wire [15:0] w_r1 = wv[47:32];
218     wire [15:0] w_i1 = wv[63:48];
219     wire [15:0] w_r2 = wv[79:64];
220     wire [15:0] w_i2 = wv[95:80];
221     wire [15:0] w_r3 = wv[111:96];
222     wire [15:0] w_i3 = wv[127:112];
223
224
225     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]);
226     wire [15:0] u_r_t0 = fft_node0[15:0];
227     wire [15:0] u_i_t0 = fft_node0[31:16];
228     wire [15:0] v_r_t0 = fft_node0[47:32];
229     wire [15:0] v_i_t0 = fft_node0[63:48];
230
231     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]);
232     wire [15:0] u_r_t1 = fft_node1[15:0];
233     wire [15:0] u_i_t1 = fft_node1[31:16];
234     wire [15:0] v_r_t1 = fft_node1[47:32];
235     wire [15:0] v_i_t1 = fft_node1[63:48];
236
237     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]);
238     wire [15:0] u_r_t2 = fft_node2[15:0];
239     wire [15:0] u_i_t2 = fft_node2[31:16];
240     wire [15:0] v_r_t2 = fft_node2[47:32];
241     wire [15:0] v_i_t2 = fft_node2[63:48];
242
243     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]);
244     wire [15:0] u_r_t3 = fft_node3[15:0];
245     wire [15:0] u_i_t3 = fft_node3[31:16];
246     wire [15:0] v_r_t3 = fft_node3[47:32];
247     wire [15:0] v_i_t3 = fft_node3[63:48];
248
249
250     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};
251     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};
252
253 }
254
255 operation FFT_8_FFT_DIF {}{inout av, in wv, inout bv, out even_ix, in
    even_const, in step, out odd_ix, in shift}
256 {
257     assign even_ix = even_const;

```

```

258     assign odd_ix = even_const + step;
259
260     wire [15:0] a_r0 = av[15:0];
261     wire [15:0] a_i0 = av[31:16];
262     wire [15:0] a_r1 = av[47:32];
263     wire [15:0] a_i1 = av[63:48];
264     wire [15:0] a_r2 = av[79:64];
265     wire [15:0] a_i2 = av[95:80];
266     wire [15:0] a_r3 = av[111:96];
267     wire [15:0] a_i3 = av[127:112];
268
269
270     wire [15:0] b_r0 = bv[15:0];
271     wire [15:0] b_i0 = bv[31:16];
272     wire [15:0] b_r1 = bv[47:32];
273     wire [15:0] b_i1 = bv[63:48];
274     wire [15:0] b_r2 = bv[79:64];
275     wire [15:0] b_i2 = bv[95:80];
276     wire [15:0] b_r3 = bv[111:96];
277     wire [15:0] b_i3 = bv[127:112];
278
279     wire [15:0] w_r0 = wv[15:0];
280     wire [15:0] w_i0 = wv[31:16];
281     wire [15:0] w_r1 = wv[47:32];
282     wire [15:0] w_i1 = wv[63:48];
283     wire [15:0] w_r2 = wv[79:64];
284     wire [15:0] w_i2 = wv[95:80];
285     wire [15:0] w_r3 = wv[111:96];
286     wire [15:0] w_i3 = wv[127:112];
287
288
289     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);
290     wire [15:0] u_r_t0 = fft_node0[15:0];
291     wire [15:0] u_i_t0 = fft_node0[31:16];
292     wire [15:0] v_r_t0 = fft_node0[47:32];
293     wire [15:0] v_i_t0 = fft_node0[63:48];
294
295     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);
296     wire [15:0] u_r_t1 = fft_node1[15:0];
297     wire [15:0] u_i_t1 = fft_node1[31:16];
298     wire [15:0] v_r_t1 = fft_node1[47:32];
299     wire [15:0] v_i_t1 = fft_node1[63:48];
300
301     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);
302     wire [15:0] u_r_t2 = fft_node2[15:0];
303     wire [15:0] u_i_t2 = fft_node2[31:16];
304     wire [15:0] v_r_t2 = fft_node2[47:32];
305     wire [15:0] v_i_t2 = fft_node2[63:48];

```

```

306
307     wire [63:0] fft_node3 = FFT_NODE_DIF(a_r3[15:0], a_i3[15:0], w_r3[15:0],
308     wire [15:0] u_r_t3 = fft_node3[15:0];
309     wire [15:0] u_i_t3 = fft_node3[31:16];
310     wire [15:0] v_r_t3 = fft_node3[47:32];
311     wire [15:0] v_i_t3 = fft_node3[63:48];
312
313
314     assign bv = {v_i_t3, v_r_t3, v_i_t2, v_r_t2, v_i_t1, v_r_t1, v_i_t0,
315     assign av = {u_i_t3, u_r_t3, u_i_t2, u_r_t2, u_i_t1, u_r_t1, u_i_t0,
316     u_r_t0};
317 }
318
319 operation FFT_BIT_REVERSE {inout AR n}{}
320 {
321     wire [31:0] lv10 = (n & 32'hffff0000) >> 16 | (n & 32'h0000ffff) << 16;
322     wire [31:0] lv11 = (lv10 & 32'hff00ff00) >> 8 | (lv10 & 32'h00ff00ff) << 8;
323     wire [31:0] lv12 = (lv11 & 32'hf0f0f0f0) >> 4 | (lv11 & 32'h0f0f0f0f) << 4;
324     wire [31:0] lv13 = (lv12 & 32'hcccccccc) >> 2 | (lv12 & 32'h33333333) << 2;
325     wire [31:0] lv14 = (lv13 & 32'haaaaaaaa) >> 1 | (lv13 & 32'h55555555) << 1;
326     assign n = lv14;
327 }
328
329
330
331
332 operation FFT_INIT {} {out done, out even_ix, out odd_ix, out w_ix, out m_ix
333     , out l, out step, out w_inc, in n, out n_2, in time_decimation}
334 {
335     assign done = 1'h0;
336     assign even_ix = 16'h0;
337     assign odd_ix = time_decimation ? 16'h1 : n_2_temp;
338     assign w_ix = 16'h0;
339     assign m_ix = 16'h0;
340     assign l = 16'h0;
341     assign step = time_decimation ? 16'h1 : n_2_temp;
342     wire [15:0] n_2_temp = RIGHT_SHIFT(n);
343     assign w_inc = time_decimation ? n_2_temp : 16'h1;
344     assign n_2 = n_2_temp;
345 }
346 operation FFT_LOAD_EVEN {}{in even_ix, in ptr_data, in done, in shift, inout
347     bv, out VAddr, in MemDataIn32, out LoadByteDisable, out debug_reg, in
348     time_decimation}
349 {
350     wire [31:0] even_ix_ext = {16'h0, even_ix};
351     wire [31:0] even_ix_shift = LEFT_SHIFT_32(even_ix_ext);
352     wire [31:0] VAddr_temp = ptr_data + LEFT_SHIFT_32(even_ix_shift);

```

```

351     assign debug_reg = VAddr_temp;
352     assign VAddr = ptr_data + LEFT_SHIFT_32(even_ix_shift);
353     assign LoadByteDisable = {16{done}};
354
355
356     wire [31:0] data_temp = MemDataIn32;
357     wire [15:0] data_temp_r = RIGHT_SHIFT_SIGNED(data_temp[15:0]);
358     wire [15:0] data_temp_i = RIGHT_SHIFT_SIGNED(data_temp[31:16]);
359     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]};
360
361
362 }
363
364
365
366 operation FFT_LOAD_ODD {}{in odd_ix, in ptr_data, in done, inout av, out
VAddr, in MemDataIn32, out LoadByteDisable, out debug_reg}
367 {
368     wire [31:0] odd_ix_ext = {16'h0, odd_ix};
369     wire [31:0] odd_ix_ext_shift = LEFT_SHIFT_32(odd_ix_ext);
370     wire [31:0] VAddr_temp = ptr_data + LEFT_SHIFT_32(odd_ix_ext_shift);
371     assign debug_reg = VAddr_temp;
372     assign VAddr = ptr_data + LEFT_SHIFT_32(odd_ix_ext_shift);
373     assign LoadByteDisable = {16{done}};
374     wire [31:0] data_temp = MemDataIn32;
375     assign av = {data_temp[31:0], av[127:32]};
376
377 }
378
379
380
381 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}
382 {
383     wire [31:0] w_ix_ext = {16'h0, w_ix};
384     wire [31:0] w_ix_ext_shift = LEFT_SHIFT_32(w_ix_ext);
385     wire [31:0] VAddr_temp = ptr_w + LEFT_SHIFT_32(w_ix_ext_shift);
386     assign debug_reg = VAddr_temp;
387     assign VAddr = ptr_w + LEFT_SHIFT_32(w_ix_ext_shift);
388     assign LoadByteDisable = {16{done}};
389
390     wire [31:0] data_temp = MemDataIn32;
391     wire [15:0] data_temp_r = RIGHT_SHIFT_SIGNED(data_temp[15:0]);
392     wire [15:0] data_temp_i = RIGHT_SHIFT_SIGNED(data_temp[31:16]);
393     assign wv = shift ? {data_temp_i[15:0], data_temp_r[15:0], wv[127:32]} :
{data_temp[31:0], wv[127:32]};
394
395
396

```

```

397     wire [15:0] even_temp = even_ix + 16'h1;
398
399     wire [15:0] w_ix_temp = w_ix + w_inc;
400     wire jump = (w_ix_temp < n_2) ? 1'h0 : 1'h1;
401
402     wire [15:0] even_ix_wire = jump ? even_temp + step : even_temp;
403     assign even_ix = even_ix_wire;
404     assign odd_ix = even_ix_wire + step;
405     assign w_ix = jump ? 16'h0 : w_ix_temp;
406 }
407
408
409
410 operation FFT_STORE_EVEN {} {in even_ix, in ptr_data, in done, inout bv, out
    VAddr, out MemDataOut32, out StoreByteDisable}
411 {
412     wire [31:0] even_ix_ext = {16'h0, even_ix};
413     wire [31:0] even_ix_shift = LEFT_SHIFT_32(even_ix_ext);
414     assign VAddr = ptr_data + LEFT_SHIFT_32(even_ix_shift);
415     assign StoreByteDisable = {16{done}};
416
417
418     assign MemDataOut32 = bv[31:0];
419
420     assign bv = {32'h0, bv[127:32]};
421 }
422
423
424
425 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}
426 {
427     wire [31:0] odd_ix_ext = {16'h0, odd_ix};
428     wire [31:0] even_ix_shift = LEFT_SHIFT_32(odd_ix_ext);
429     assign VAddr = ptr_data + LEFT_SHIFT_32(even_ix_shift);
430     assign StoreByteDisable = {16{done}};
431
432     assign MemDataOut32 = av[31:0];
433
434     assign av = {32'h0, av[127:32]};
435
436     wire [15:0] even_temp = even_ix + 16'h1;
437     wire [15:0] m_temp = m_ix + 16'h1;
438     wire jump = (m_temp < step) ? 1'h0 : 1'h1;
439
440     wire [15:0] even_ix_wire = jump ? even_temp + step : even_temp;
441     assign even_ix = even_ix_wire;
442     assign odd_ix = even_ix_wire + step;
443     assign m_ix = jump ? 16'h0 : m_temp;
444 }

```

```

445
446
447
448 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}
449 {
450     wire done_temp = !(even_ix < n);
451     assign even_const = done_temp ? 16'h0 : even_ix;
452     wire [15:0] even_ix_temp = done_temp ? 16'h0 : even_ix;
453     assign even_ix = even_ix_temp;
454     wire [15:0] step_shift = time_decimation ? LEFT_SHIFT(step) :
      RIGHT_SHIFT(step);
455     wire [15:0] step_temp = done_temp ? step_shift : step;
456     assign odd_ix = even_ix_temp + step_temp;
457     assign step = step_temp;
458     wire [15:0] w_inc_shift = time_decimation ? RIGHT_SHIFT(w_inc) :
      LEFT_SHIFT(w_inc);
459     assign w_inc = done_temp ? w_inc_shift : w_inc;
460     wire [15:0] l_temp = done_temp ? l + 16'h1 : l;
461     wire l_0 = (l<0) ? 1'b1 : 1'b0;
462     assign shift = ((!time_decimation) & l_0) ? 1'h1 : shift;
463     assign l = l_temp;
464     assign done = (l_temp < lg2_n) ? 1'h0 : 1'h1;
465     assign done_out = (l_temp < lg2_n) ? 1'h0 : 1'h1;
466 }

```

## 3 Questions

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

**Precalculation of twiddle factors** Twiddle factors are precalculated in `GenerationCoefficient` 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.

**Improvement 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 read/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 greatly 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.