



**TECHNISCHE
UNIVERSITÄT
DRESDEN**

Vodafone Chair for Mobile Communications Systems

LAB REPORT HW/SW-CODESIGN WINTER SEMESTER 17/18

Timo Nicolai

CONTENTS

1	Introduction	3
2	Overview	4
2.1	Utilized Techniques	4
2.2	Accelerated Sections of the Algorithm	5
3	Execution Time and Area Impact	6
3.1	Speedup	6
3.2	Area	6
4	Detailed Breakdown	7
5	Code Listings	10

1 INTRODUCTION

This report describes an implementation of the Cooley-Tukey FFT algorithm for one of Tensilica's Xtensa processor platforms. The Tensilica Instruction Extension hardware-description DSL is utilized to customize the core Xtensa core architecture by means of additional specialized registers and instructions which make it possible to perform the specialized task of computing a DIT/DIF FFT at significantly higher speeds than would ordinarily be possible with the base ISA alone.

Simulation shows that the resulting microprocessor architecture can achieve more than twenty times the throughput for FFT computations over a sufficiently large number of input points. A manufactured Xtensa processor using these extensions would approximately consume twice as much area as compared to an unmodified one.

The algorithm itself is implemented in assembly in the file `fsm.S`, the instruction set extensions are implemented in the `fft.tie` file in the aforementioned TIE language. While the code itself is short, a significant amount of thought has gone into this project to squeeze every last bit of performance out of the implementation (whenever the throughput / area tradeoff was reasonable). Most advanced features of the Xtensa ISA and the TIE language have been utilized as described in the following sections.

2 OVERVIEW

2.1 UTILIZED TECHNIQUES

23 new instructions were generated, of which the most performance critical, BFLY_DIT/DIF_MAIN is also by far the most area consuming (over 30.000 gates vs. about 2300 gates for the second most area consuming operation).

While the structure of the reference implementation was kept mainly intact, the whole final program was hand-written in assembly. Also, the first three (or last three in case of DIF) iterations of the main butterfly loop were unrolled because they require special treatment and the structure of the innermost loop was changed slightly to allow parallel processing of eight complex values in one iteration.

Similarly structured C code was initially also developed so that its optimized assembly could be consulted as a reference. In the end this code was completely disposed of because even with significant effort, the compiler could not compete with the hand-written assembly.

The following is a summary of the techniques that were utilized to achieve an increase in throughput.

SIMD :

The number of cycles in which a certain amount of values can be loaded from/stored to memory constitutes a bottleneck in multiple parts of the algorithm (especially the main butterfly operations). 128 bit SIMD processor states (states because the main butterfly operation needs to operate on eight 128 bit registers at once (inputs + outputs) which is not possible with user registers) make it possible to alleviate this by loading eight 16 bit values with one instruction.

FLIX :

FLIX improves this last measure even further by making it possible to load 2 times 128 bit with a single VLIW instruction.

Also, multiple other operations can be performed simultaneously this way, e.g. increasing a loop counter or performing a branch in parallel with two load/store operations without additional hardware overhead.

Internal Pipelining :

The program's most complex TIE operation which performs eight butterfly operations, including twiddle factor calculation, in parallel, was internally pipelined by scheduling its results one cycle in the future. If the circuit was to be synthesized, this would shorten the operation's critical path, possibly allowing the system to operate at a higher clock frequency.

Also, software pipelining was utilized in several places, especially when performing the butterfly operations. A set of separate TIE load and store buffers was necessary in order to make this possible.

Hardware Lookup :

The sine wave table needed for twiddle calculation was implemented as a hardware lookup table, both to avoid taking up RAM unnecessarily and to reduce the need for performance limiting memory accesses.

Hand-coded Assembly :

Especially when working with optimizations, the compiler often produced undesirable results by e.g. optimizing away seemingly unused pointers which might be essential in order for some TIE operation to function correctly or by producing functional but suboptimal solutions.

In addition, any modern compiler will struggle with producing highly optimized VLIW code. Since FLIX can potentially provide a large performance gain, this would have been a problem.

Because of this last point, inline assembly would have been necessary for critical parts of the program in any case. This would clutter up the code and make it less readable. In contrast, the final assembly code is reasonably short while also remaining easily understandable.

2.2 ACCELERATED SECTIONS OF THE ALGORITHM

DIT/DIF :

The more the butterfly operations were optimized, the more decimation in time contributed to the total execution time because parts of it can not be realized with SIMD memory accesses. Still, a vast improvement over the reference implementation was possible, mainly because dynamic bit-reversal is a rather complicated operation without dedicated hardware. Here it is realized with a bit-reverse ripple carry adder.

Scaling :

Whether scaling is necessary in each loop iteration is determined by repeatedly loading multiple input values into SIMD registers, calculating their absolute value in hardware and then checking whether their second most significant bit remains set, in which case shifts need to be applied later on (This applies only for the inverse FFT).

Twiddle-Factor Computation :

This could be easily realized with little more than a hardware lookup table, which eliminated the need to store the same information in (limited) RAM and also made it possible to directly calculate twiddle factors from within a butterfly operation, saving additional cycles, the hardware table's size was kept small with the help of some bit twiddling tricks in a function used to access it.

Butterfly Operations :

The butterfly operations are the part of the FFT that could benefit the most from parallelization so it made sense to implement them in hardware. The first three loop iterations (or the last three if DIF is used) repeatedly perform four butterfly operations in parallel with one operation while the main inner loop performs eight butterfly operations in one instruction.

3 EXECUTION TIME AND AREA IMPACT

3.1 SPEEDUP

Comparison between C reference implementation and the asm/tie implementation:

N	Cycles ¹ (C)	Cycles (asm/tie) ²	Speedup ³
32	11,270 (11,890)	≈670	≈16.8 (17.7)
256	119,084 (127,624)	≈4,900a	≈24.3 (26.0)
1024	563,642 (600,143)	≈21,600	≈26.0 (27.8)

From this comparison it can be seen that the gains in terms of speed are significantly larger for large values of N and also larger for DIF (because the C DIF implementation is slower than the respective C DIT implementation while the asm/tie DIF implementation is as fast as it's DIT counterpart (give or take less than 100 cycles)).

3.2 AREA

- Xtensa configuration without TIE: ≈93,000 gates.
- Xtensa configuration TIE only: ≈175,200 gates, area overhead is ≈188.4%.
- Xtensa configuration TIE only, DIT or DIF only⁴: ≈126,100 gates, area overhead is ≈135.6%.

The hardware components needed for the butterfly computations are responsible for most of the area increase (≈49.000 gates each for DIT and DIF) with DIT/DIF and scaling a distant second (less than 3000 gates each).

It can be seen that the are overhead is significant with the total area more than doubling when TIE operations are used. In an actual implementation a tradeoff would have to be made, a processing speed increase of more than 250 might not be desirable if a slower implementation can save valuable chip area instead.

¹Sum of cycles needed for non-inverse and inverse FFT for different problem sizes, code was compiled with O2 flag. Value in brackets refers to DIF.

²Same cycle metric. The IDE was not capable of giving overall cycle numbers for function implemented in assembly, the numbers here are the approximate total sums of cycles reported for each labelled section within the asm code. Performance differences between DIT/DIF were insignificantly small and thus only one value is given.

³Rounded, Value in brackets refers to DIF.

⁴Meaning that only hardware components needed for one approach are added to the hardware.

4 DETAILED BREAKDOWN

The first part of the algorithm that was optimized was the decimation in time, i.e. the initial reordering of elements in the input arrays. The first important realisation here was that memory accesses create a lower bound on the minimum cycle count of this (as well as all following) operation. Using 128 bit processor states as well as the two parallel memory accesses per cycle made possible by FLIX, at most eight 32 bit complex numbers (i.e. 8 x 16 bit real and 8 x 16 bit imaginary numbers). can be loaded/stored per cycle.

Thus the strategy used for efficient decimation in time is to repeatedly load eight complex numbers into two load buffers (one storing the real and one storing the imaginary parts), and then in 8 x 2 cycles to respectively load one complex number from the memory locations defined by the bit-reverse of a continuously increasing loop counter value into two additional store buffers and then overwriting the values at these memory locations with the corresponding complex value from the load buffers (see lines 62-80 (asm)). The store buffers are also 128 Bit in size and both load and store buffers are shifted by 16 Bits at each of these eight steps so that the next values from the load buffers and the next free slots in the store buffers can be easily accessed by the following step's operations. Afterwards, the store buffers contents are written to same memory locations from which the load buffers where initially populated. This whole procedure is then repeated for a total of $N/8$ times (note that N is always divisible by eight if N is greater than or equal to eight). Since memory locations defined by two consecutive bit-reverse loop-counter values are in general not adjacent, SIMD loads or stores are out of the question in this case. For this reason I believe that the approach chosen is close to optimal with regard to clock cycles spent.

One improvement over my initial approach -in which at each step, a bit reversed version of the loop counter was obtained through a single hardware operation- was to instead construct a reverse ripple-carry adder which made it possible to introduce an additional bit-reverse loop-counter which is incremented at each step (see lines 384-410 (tie)). This is more efficient, since the inverse of some counter value also depends on the problem size N , which makes one-step inversion complicated and costly. Speedup was significant and is more or less important depending on how heavily the butterfly operations are optimized. e.g. for $N = 32$, decimation in time (or frequency, both operations are equivalent) makes up more than 25% of the total cycle count despite these optimizations because the butterflies are implemented with powerful hardware support and thus do not dominate the total cycle count completely

At the start of each main loop iteration, it is determined whether scaling is necessary, this is a complicated operation in the case of inverse FFT. It is implemented by repeatedly loading multiple input values into SIMD registers, calculating their absolute values in hardware and then checking if those have the second most significant bit set in which case scaling should be performed (see lines 417-451 (tie)). The overall performance of this operation is very similar to that of decimation in time but since it has to be performed multiple times in case of inverse FFT it is slightly more performance critical, thus justifying a fast hardware implementation.

The actual butterfly calculations are carried out partly in shared functions. The first three (or last three for DIF) loop iterations are unrolled since they implement the parallel butterfly operations slightly differently. In these iterations, twiddle factors are hardwired and four butterfly operations are always carried out in parallel, these four butterflies are shared hardware (see e.g. lines 165-175 (asm) and 455-494 (tie)).

The innermost part of the main loop is the most performance critical part of the algorithm

and needs four additional butterfly structures for a total of eight (see lines 150-163 (asm) and 584-655 (tie)). Twiddle factors are calculated directly within the main TIE butterfly operation. It would also be possible to precalculate the twiddle factors, especially since the same ones are frequently reused within the algorithm. But this would require a significant additional number as intermediate storage memory. Because the innermost part of the main loop operates slightly differently from the reference implementation in order to allow parallel processing of eight butterflies in each iteration, twiddle factors can also not easily be reused across iterations. Calculating the twiddle factors is in essence mainly a lookup operation that uses a hardware table of sine-wave values. An optimization was made here by limiting the size of this table to 256 value and then providing an appropriate value for all 10 bit indices from 0 to 1023 by utilizing additional hardware logic (see lines 85-98 (tie)).

The heavily optimized TIE operation uses up more than 30.000 gates. Again, loads and stores dictate a lower bound on the loops cycle count. Thus, my initial approach was to first load 16 complex numbers (512 bit), process them in parallel with eight butterflies implemented in a single TIE operation and then store the results. With FLIX this could be achieved in only five cycles (discounting inevitable branch delay) while also simultaneously updating a loop counter with no further overhead.

A further improvement was to introduce a two stage pipeline: The load operations load data into four 128 bit processor states (load buffers) in two steps while simultaneously with the first step, the old contents of these states are processed by the butterfly operations and the results stored into four separate 128 bit processor states (store buffers) which are then stored to memory. This requires treating the first and last loop iteration separately but saves a whole cycle.

Because of the large area overhead of the main butterfly TIE operation, I also thought about splitting it into two separate operations consisting of four butterflies each. These could be shared with those used in the first three loop iterations, resulting in a large area reduction. The same cycle count can be achieved with this method by still loading/storing 16 complex numbers per iteration, but then operating only on one half of the load buffers in parallel with the first load cycle and then one the second half in the second loop cycle. This would of course require additional muxing hardware and also four additional 128 bit buffer states to prevent the first load cycle from overwriting the inputs of the second set of butterfly operations but still result in an overall area reduction. This could even be taken one step further by using four 128 bit buffer stages and distributing the butterfly operations across both the two load and store cycles But this would require an even more complicated handling of the first and last iterations and only reduce area significantly if the first three loop operations where also handled similarly (because otherwise four butterfly instances will need to exist in hardware anyway). In the end I decided against both these approaches because while reducing area, they might actually increase the main butterfly TIE operation's critical path because they contain additional muxing logic and can at the same time not be internally pipelined because they are always in at least one case immediately followed by store operations depending on their results.

So a more area effective approach is purposely not chosen in order to facilitate internal pipelining of the in any case very complex main butterfly TIE operation whose critical path includes multiple additions and one multiplication and would almost certainly constitute a bottleneck for the synthesized circuit's clock frequency (see lines 657-662 (tie)).

Overall I believe that it the implementation given is somewhat close to the upper limit achievable in terms of speed (since memory loads and stores form a hard lower limit on the total number of clock cycles needed). In an actual application it might be ideal to first determine a hard upper bound on the speed of the implementation and then optimize for area, More than doubling chip area might be too excessive, although this is impossible to say without concrete application demands.

If area needs to be reduced I would suggest making the (very large) components realising

the innermost butterfly loop smaller and less efficient, e.g. by only processing four butterflies at a time instead of eight. This would reduce area significantly while not killing performance completely. By working with the Xtensa IDE I have learned a lot about hand-optimizing assembly code and the pitfalls of compiler optimization as well as how to utilize techniques such as SIMD and VLIW. I have also improved my intuition about the relative strengths, weaknesses and performance differences between software and hardware implementations, seamlessly mixing the two together was an interesting experience.

5 CODE LISTINGS

Listing 1: fft.S

```
1 // fft.S: FFT function assembly source code
2
3 // Comment this in to switch from DIT to DIF. Note that the definition of this
4 // symbol should ideally be passed as a compiler/assembler option instead but
5 // this does not seem to be possible without manually modifying the Makefile
6 // generated by Xtensa Explorer.
7
8 // #define DIF
9
10 #define LOG2_N_WAVE 10
11
12 .align 4
13 .global fft
14 .type fft, @function
15
16 fft:
17     // a2 = fr, a3 = fi, a4 = m
18     entry a1, 128
19
20     // Verify that 5 <= m <= 10, implying that 32 <= n <= 1024.
21     // The upper limit is already present in the reference implementation
22     // and the lower limit is necessary because some processing steps of this
23     // function load up to 16 complex values from memory per clock cycle and
24     // process these in one-stage software pipelines.
25
26     bgei a4, 5, m_valid
27     movi.n a5, 11
28     blt a4, a5, m_valid
29     movi.n a2, -1
30     retw
31 m_valid:
32
33 #ifndef DIF
34     // save m because it is needed to determine the initial value of k
35     mov.n a13, a4
36 #endif
37
38     // a4 = n = 1 << m
39     ssl a4
40     movi.n a4, 1
41     sll a4, a4
42
43     // b0 = 0, b1 = 1, b2 = inverse, b3 = shift
44     movi.n a6, 2
45     wsr.BR a6
46     beqz a5, no_inverse
47     orb b2, b2, b1
48 no_inverse:
49
50     // scale = 0
51     movi.n a5, 0
52
53 /*--DECIMATION IN TIME-----*/
54
55 #ifndef DIF
56     // a7 = forward counter, a8 = reverse counter
57     { movi.n a7, 0; movi.n a8, 0; nop }
58     // a9 = real load pointer, a10 = imaginary load pointer
```

```

59 { mov.n a9, a2; mov.n a10, a3; nop }
60 // a11 = real store pointer, a12 = imaginary store pointer
61 { mov.n a11, a2; mov.n a12, a3; nop }
62 dit:
63 { load1 a9; load2 a10; nop }
64 { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
65 { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
66 { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
67 { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
68 { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
69 { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
70 { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
71 { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
72 { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
73 { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
74 { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
75 { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
76 { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
77 { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
78 { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
79 { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
80 { store1 a11; store2 a12; bne a7, a4, dit }
81 #endif
82
83 /*--MAIN LOOP-----*/
84
85 #ifdef DIF
86 // a6 = l = n/2
87 movi.n a6, 1
88 ssr a6
89 srl a6, a4
90 // a7 = k = LOG2_N_WAVE - m
91 neg a13, a13
92 addi.n a7, a13, LOG2_N_WAVE
93 #else
94 // a6 = l = 1
95 movi.n a6, 1
96 // a7 = k = LOG2_N_WAVE - 1
97 movi.n a7, LOG2_N_WAVE - 1
98 #endif
99
100 main_loop:
101
102 #ifdef DIF
103 beqz a6, main_loop_done // l == 0 ? => done
104 #else
105 beq a4, a6, main_loop_done // l == n ? => done
106 #endif
107
108 slli a8, a6, 1 // a8 = l*2
109
110 /*--SCALING-----*/
111
112 bf b2, no_scaling // skip scaling for non-inverse FFT
113 { mov.n a9, a2; mov.n a10, a3; andb b3, b3, b0 } // b3 = shift = 0
114 { load1 a9; load2 a10; movi.n a11, 8 }
115 check_scaling:
116 // pass n to "scaling" so that the loop can be broken by setting a11 = n
117 { load1 a9; load2 a10; scaling b3, a11, a4 }
118 bne a11, a4, check_scaling
119 scaling b3, a11, a4
120 bf b3, scaling_done
121 addi.n a5, a5, 1 // ++scale
122 j scaling_done
123 no_scaling:
124 orb b3, b3, b1 // shift = 1
125 scaling_done:
126
127 /*--BUTTERFLIES-----*/
128
129 #ifdef DIF
130 #define BFLY_MAIN bfly_dif_main
131 #define BFLY_UNROLL1 bfly_dif_unroll1

```

```

132 #define BFLY_UNROLL2 bfly_dif_unroll2
133 #define BFLY_UNROLL4 bfly_dif_unroll4
134 #else
135 #define BFLY_MAIN bfly_dit_main
136 #define BFLY_UNROLL1 bfly_dit_unroll1
137 #define BFLY_UNROLL2 bfly_dit_unroll2
138 #define BFLY_UNROLL4 bfly_dit_unroll4
139 #endif
140
141 { mov.n a9, a2; mov.n a10, a3; nop }
142 { mov.n a11, a2; mov.n a12, a3; nop }
143
144 // loop iterations for 1 in {1, 2, 4} are treated differently
145 beqi a6, 1, bfly_unroll1
146 beqi a6, 2, bfly_unroll2
147 beqi a6, 4, bfly_unroll4
148
149 movi.n a13, 0
150 bfly_outer:
151 { offs_load3 a9, a8; offs_load4 a10, a8; movi.n a14, 0 }
152 { load1 a9; load2 a10; beqi a6, 8, bfly_inner_done }
153 bfly_inner:
154 { offs_load3 a9, a8; offs_load4 a10, a8; BFLY_MAIN a14, a7, b2, b3 }
155 { load1 a9; load2 a10; addi a15, a14, 8 }
156 { offs_store3 a11, a8; offs_store4 a12, a8; nop }
157 { store1 a11; store2 a12; bne a6, a15, bfly_inner }
158 bfly_inner_done:
159 BFLY_MAIN a14, a7, b2, b3
160 { offs_store3 a11, a8; offs_store4 a12, a8; add.n a13, a13, a8 }
161 { store1 a11; store2 a12; nop }
162 { inc4 a9, a10, a11, a12, a8; nop; bne a4, a13, bfly_outer }
163 j next_iter
164
165 bfly_unroll1:
166 { load1 a9; load2 a10; movi.n a13, 16 }
167 { load1 a9; load2 a10; BFLY_UNROLL1 b2, b3 }
168 bfly_unroll1_loop:
169 { store1 a11; store2 a12; BFLY_UNROLL1 b2, b3 }
170 { load1 a9; load2 a10; addi.n a13, a13, 16 }
171 { store1 a11; store2 a12; BFLY_UNROLL1 b2, b3 }
172 { load1 a9; load2 a10; bne a4, a13, bfly_unroll1_loop }
173 bfly_unroll1_final:
174 { store1 a11; store2 a12; BFLY_UNROLL1 b2, b3 }
175 { store1 a11; store2 a12; j next_iter }
176
177 bfly_unroll2:
178 { load1 a9; load2 a10; movi.n a13, 16 }
179 { load1 a9; load2 a10; BFLY_UNROLL2 b2, b3 }
180 bfly_unroll2_loop:
181 { store1 a11; store2 a12; BFLY_UNROLL2 b2, b3 }
182 { load1 a9; load2 a10; addi.n a13, a13, 16 }
183 { store1 a11; store2 a12; BFLY_UNROLL2 b2, b3 }
184 { load1 a9; load2 a10; bne a4, a13, bfly_unroll2_loop }
185 bfly_unroll2_final:
186 { store1 a11; store2 a12; BFLY_UNROLL2 b2, b3 }
187 { store1 a11; store2 a12; j next_iter }
188
189 bfly_unroll4:
190 { load1 a9; load2 a10; movi.n a13, 16 }
191 { load1 a9; load2 a10; BFLY_UNROLL4 b2, b3 }
192 bfly_unroll4_loop:
193 { store1 a11; store2 a12; BFLY_UNROLL4 b2, b3 }
194 { load1 a9; load2 a10; addi.n a13, a13, 16 }
195 { store1 a11; store2 a12; BFLY_UNROLL4 b2, b3 }
196 { load1 a9; load2 a10; bne a4, a13, bfly_unroll4_loop }
197 bfly_unroll4_final:
198 { store1 a11; store2 a12; BFLY_UNROLL4 b2, b3 }
199 { store1 a11; store2 a12; nop }
200
201 next_iter:
202 #ifdef DIF
203 // l = l/2; ++k;
204 movi.n a8, 1

```

```

205     ssr a8
206     { srl a6, a6; addi.n a7, a7, 1; j main_loop }
207 #else
208     // l = l*2; --k;
209     { mov.n a6, a8; addi.n a7, a7, -1; j main_loop }
210 #endif
211
212 /*--DECIMATION IN FREQUENCY-----*/
213
214 main_loop_done:
215
216 #ifndef DIF
217     // a7 = forward counter, a8 = reverse counter
218     { movi.n a7, 0; movi.n a8, 0; nop }
219     // a9 = real load pointer, a10 = imaginary load pointer
220     { mov.n a9, a2; mov.n a10, a3; nop }
221     // a11 = real store pointer, a12 = imaginary store pointer
222     { mov.n a11, a2; mov.n a12, a3; nop }
223 dif:
224     { load1 a9; load2 a10; nop }
225     { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
226     { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
227     { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
228     { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
229     { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
230     { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
231     { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
232     { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
233     { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
234     { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
235     { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
236     { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
237     { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
238     { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
239     { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
240     { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
241     { store1 a11; store2 a12; bne a4, a7, dif }
242 #endif
243
244     mov.n a2, a5
245     retw

```

Listing 2: fft.tie

```

1 /*--TABLES-----*/
2
3 /* Sine wave hardware lookup table. Only contains values from zero up to pi/2,
4 * accessing values in the full range from 0 to 2pi is made possible by several
5 * bit twiddling tricks in the auxilliary function "sin" which is used to
6 * by hardware operations which access this table.
7 */
8 table SINE_WAVE 16 257 {
9     0,      201,    402,    603,    804,    1005,    1206,    1406,
10    1607,    1808,    2009,    2209,    2410,    2610,    2811,    3011,
11    3211,    3411,    3611,    3811,    4011,    4210,    4409,    4608,
12    4807,    5006,    5205,    5403,    5601,    5799,    5997,    6195,
13    6392,    6589,    6786,    6982,    7179,    7375,    7571,    7766,
14    7961,    8156,    8351,    8545,    8739,    8932,    9126,    9319,
15    9511,    9703,    9895,    10087,    10278,    10469,    10659,    10849,
16    11038,    11227,    11416,    11604,    11792,    11980,    12166,    12353,
17    12539,    12724,    12909,    13094,    13278,    13462,    13645,    13827,
18    14009,    14191,    14372,    14552,    14732,    14911,    15090,    15268,
19    15446,    15623,    15799,    15975,    16150,    16325,    16499,    16672,
20    16845,    17017,    17189,    17360,    17530,    17699,    17868,    18036,
21    18204,    18371,    18537,    18702,    18867,    19031,    19194,    19357,
22    19519,    19680,    19840,    20000,    20159,    20317,    20474,    20631,
23    20787,    20942,    21096,    21249,    21402,    21554,    21705,    21855,
24    22004,    22153,    22301,    22448,    22594,    22739,    22883,    23027,
25    23169,    23311,    23452,    23592,    23731,    23869,    24006,    24143,
26    24278,    24413,    24546,    24679,    24811,    24942,    25072,    25201,
27    25329,    25456,    25582,    25707,    25831,    25954,    26077,    26198,

```

```

28 26318, 26437, 26556, 26673, 26789, 26905, 27019, 27132,
29 27244, 27355, 27466, 27575, 27683, 27790, 27896, 28001,
30 28105, 28208, 28309, 28410, 28510, 28608, 28706, 28802,
31 28897, 28992, 29085, 29177, 29268, 29358, 29446, 29534,
32 29621, 29706, 29790, 29873, 29955, 30036, 30116, 30195,
33 30272, 30349, 30424, 30498, 30571, 30643, 30713, 30783,
34 30851, 30918, 30984, 31049, 31113, 31175, 31236, 31297,
35 31356, 31413, 31470, 31525, 31580, 31633, 31684, 31735,
36 31785, 31833, 31880, 31926, 31970, 32014, 32056, 32097,
37 32137, 32176, 32213, 32249, 32284, 32318, 32350, 32382,
38 32412, 32441, 32468, 32495, 32520, 32544, 32567, 32588,
39 32609, 32628, 32646, 32662, 32678, 32692, 32705, 32717,
40 32727, 32736, 32744, 32751, 32757, 32761, 32764, 32766,
41 32767
42 }
43
44 /*--STATES-----*/
45
46 /* Load and store buffers used by several SIMD operations (both DIT and
47 * butterfly). Both load and store buffers are needed because of the specific
48 * way DIT is implemented and also in order to allow the butterfly operations
49 * to be (software) pipelined. Four buffers (of each type) are necessary
50 * because the main butterfly TIE operation performs eight butterfly
51 * computations in parallel.
52 */
53
54 state load_buf1 128
55 state load_buf2 128
56 state load_buf3 128
57 state load_buf4 128
58
59 state store_buf1 128
60 state store_buf2 128
61 state store_buf3 128
62 state store_buf4 128
63
64 /*--FUNCTIONS-----*/
65
66 /* Computes the absolute value of a 16 bit two's complement number. */
67 function [15:0] abs([15:0] x)
68 {
69   assign abs = TIEmux(x[15], x, TIEadd(0, ~x, 1'b1));
70 }
71
72 /* Performs a 16 bit arithmetic right shift. */
73 function [15:0] sra1([15:0] x)
74 {
75   assign sra1 = {x[15], x[15:1]};
76 }
77
78 /* Performs a 16 bit fixed point multiplication. */
79 function [15:0] fix_mpy([15:0] a, [15:0] b)
80 {
81   wire [31:0] prod = TIEmul(a, b, 1'b1);
82   assign fix_mpy = prod >> 15;
83 }
84
85 /* Auxilliary function for accessing the hardware sine table, helps reduce that
86 * table's size by three quarters by performing index inversion and/or result
87 * negation depending on the index's two most significant bits.
88 */
89 function [15:0] sin([9:0] index)
90 {
91   wire [8:0] index_mirrored = TIEadd(512, ~index[7:0], 1'b1);
92   wire [8:0] base_index = TIEmux(index[8], index[7:0], index_mirrored);
93
94   wire [15:0] base_val = SINE_WAVE[base_index];
95   wire [15:0] base_val_negated = TIEadd(0, ~base_val, 1'b1);
96
97   assign sin = TIEmux(index[9], base_val, base_val_negated);
98 }
99
100 /* Computes twiddle factors. */

```

```

101 function [31:0] twiddles([9:0] m, [3:0] k)
102 {
103     wire [9:0] ti_index = m << k;
104     wire [9:0] tr_index = TIEadd(ti_index, 256, 1'b0);
105
106     assign twiddles = {sin(tr_index), sin(ti_index)};
107 }
108
109 /* Performs a single butterfly operation, ar/ai and br/bi are real and
110 * imaginary parts of the two input operands and tr/ti are the respective
111 * twiddle factors.
112 */
113 function [63:0] bfly([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
114                    [15:0] tr, [15:0] ti, inv, shift)
115 {
116     wire [15:0] twiddle_r = TIEmux(shift, tr, sra1(tr));
117     wire [15:0] tmp = TIEmux(inv, TIEadd(16'h0, ~ti, 1'b1), ti);
118     wire [15:0] twiddle_i = TIEmux(shift, tmp, sra1(tmp));
119
120     wire [15:0] tmp_r = TIEadd(fix_mpy(twiddle_r, br), ~fix_mpy(twiddle_i, bi), 1'b1);
121     wire [15:0] tmp_i = TIEadd(fix_mpy(twiddle_r, bi), fix_mpy(twiddle_i, br), 1'b0);
122
123     wire [15:0] ar_out = TIEadd(TIEmux(shift, ar, sra1(ar)), tmp_r, 1'b0);
124     wire [15:0] ai_out = TIEadd(TIEmux(shift, ai, sra1(ai)), tmp_i, 1'b0);
125     wire [15:0] br_out = TIEadd(TIEmux(shift, ar, sra1(ar)), ~tmp_r, 1'b1);
126     wire [15:0] bi_out = TIEadd(TIEmux(shift, ai, sra1(ai)), ~tmp_i, 1'b1);
127
128     assign bfly = {ar_out, ai_out, br_out, bi_out};
129 }
130
131 /* Same as above except for DIF approach. */
132 function [63:0] bfly_dif([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
133                       [15:0] tr, [15:0] ti, inv, shift)
134 {
135     wire [15:0] twiddle_r = TIEmux(shift, tr, sra1(tr));
136     wire [15:0] twiddle_i_tmp = TIEmux(inv, TIEadd(16'h0, ~ti, 1'b1), ti);
137     wire [15:0] twiddle_i = TIEmux(shift, twiddle_i_tmp, sra1(twiddle_i_tmp));
138
139     wire [15:0] ar_out_tmp = TIEadd(ar, br, 1'b0);
140     wire [15:0] ai_out_tmp = TIEadd(ai, bi, 1'b0);
141     wire [15:0] ar_out = TIEmux(shift, ar_out_tmp, sra1(ar_out_tmp));
142     wire [15:0] ai_out = TIEmux(shift, ai_out_tmp, sra1(ai_out_tmp));
143
144     wire [15:0] tmp_r = TIEadd(ar, ~br, 1'b1);
145     wire [15:0] tmp_i = TIEadd(bi, ~ai, 1'b1);
146     wire [15:0] br_out = TIEadd(fix_mpy(twiddle_r, tmp_r), fix_mpy(twiddle_i, tmp_i), 1'b0);
147     wire [15:0] bi_out = TIEadd(fix_mpy(twiddle_i, tmp_r), ~fix_mpy(twiddle_r, tmp_i), 1'b1);
148
149     assign bfly_dif = {ar_out, ai_out, br_out, bi_out};
150 }
151
152 /* Four wrapper functions around "bfly" declared as shared so that the
153 * resulting hardware units can be shared between the three unrolled butterfly
154 * loop stages as well as the main butterfly operation. The remaining four
155 * butterfly units needed by the main butterfly loop are not shared and can
156 * thus the bare "bfly" function can be used to implement them.
157 */
158
159 function [63:0] bfly1([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
160                    [15:0] tr, [15:0] ti, inv, shift) shared
161 {
162     assign bfly1 = bfly(ar, ai, br, bi, tr, ti, inv, shift);
163 }
164 function [63:0] bfly2([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
165                    [15:0] tr, [15:0] ti, inv, shift) shared
166 {
167     assign bfly2 = bfly(ar, ai, br, bi, tr, ti, inv, shift);
168 }
169 function [63:0] bfly3([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
170                    [15:0] tr, [15:0] ti, inv, shift) shared
171 {
172     assign bfly3 = bfly(ar, ai, br, bi, tr, ti, inv, shift);
173 }

```

```

174 function [63:0] bfly4([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
175                      [15:0] tr, [15:0] ti, inv, shift) shared
176 {
177     assign bfly4 = bfly(ar, ai, br, bi, tr, ti, inv, shift);
178 }
179
180 /* Same as above except for DIF approach. */
181
182 function [63:0] bfly_dif1([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
183                          [15:0] tr, [15:0] ti, inv, shift) shared
184 {
185     assign bfly_dif1 = bfly_dif(ar, ai, br, bi, tr, ti, inv, shift);
186 }
187 function [63:0] bfly_dif2([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
188                          [15:0] tr, [15:0] ti, inv, shift) shared
189 {
190     assign bfly_dif2 = bfly_dif(ar, ai, br, bi, tr, ti, inv, shift);
191 }
192 function [63:0] bfly_dif3([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
193                          [15:0] tr, [15:0] ti, inv, shift) shared
194 {
195     assign bfly_dif3 = bfly_dif(ar, ai, br, bi, tr, ti, inv, shift);
196 }
197 function [63:0] bfly_dif4([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
198                          [15:0] tr, [15:0] ti, inv, shift) shared
199 {
200     assign bfly_dif4 = bfly_dif(ar, ai, br, bi, tr, ti, inv, shift);
201 }
202
203 /*--FLIX SLOTS-----*/
204
205 format flix_dit 64 {flix_dit_slot0, flix_dit_slot1, flix_dit_slot2}
206 slot_opcodes flix_dit_slot0 {MOVI.N, MOV.N, ADD.N, SRL, INC4,
207                             LOAD1, STORE1, OFFS_LOAD3, OFFS_STORE3,
208                             DSWAP11, DSWAP21}
209 slot_opcodes flix_dit_slot1 {MOVI.N, MOV.N, ADDI.N, ADD.N,
210                             LOAD2, STORE2, OFFS_LOAD4, OFFS_STORE4,
211                             DSWAP12, DSWAP22}
212 slot_opcodes flix_dit_slot2 {NOP, MOVI.N, ADDI, ADD.N, ANDB, BNE, BEQI, J,
213                             DINC, SCALING,
214                             BFLY_DIT_UNROLL1, BFLY_DIT_UNROLL2,
215                             BFLY_DIT_UNROLL4, BFLY_DIT_MAIN,
216                             BFLY_DIF_UNROLL1, BFLY_DIF_UNROLL2,
217                             BFLY_DIF_UNROLL4, BFLY_DIF_MAIN}
218
219 /*--OPERATIONS-----*/
220
221 /* Add a value to four registers in parallel, used by the main butterfly
222  * operation.
223  */
224 operation INC4
225 {inout AR r1, inout AR r2, inout AR r3, inout AR r4, in AR plus}
226 {}
227 {
228     assign r1 = TIEadd(r1, plus, 1'b0);
229     assign r2 = TIEadd(r2, plus, 1'b0);
230     assign r3 = TIEadd(r3, plus, 1'b0);
231     assign r4 = TIEadd(r4, plus, 1'b0);
232 }
233
234 /*--SIMD LOADS AND STORES-----*/
235
236 /* Load from memory to the first 128 bit load buffer and increment the pointer
237  * used accordingly. This operation is used both during DIT and all butterfly
238  * stages.
239  */
240 operation LOAD1
241 {inout AR *data_ptr}
242 {out VAddr, in MemDataIn128, out load_buf1}
243 {
244     assign VAddr = data_ptr;
245     assign load_buf1 = MemDataIn128;
246

```



```

247     assign data_ptr = TIEadd(data_ptr, 16, 1'b0);
248 }
249
250 /* Same as above except that the result is loaded into the second load
251 * buffer.
252 */
253 operation LOAD2
254 {inout AR *data_ptr}
255 {out VAddr, in MemDataIn128, out load_buf2}
256 {
257     assign VAddr = data_ptr;
258     assign load_buf2 = MemDataIn128;
259
260     assign data_ptr = TIEadd(data_ptr, 16, 1'b0);
261 }
262
263 /* Load from memory to the third load buffer. The memory location
264 * is determined from a pointer plus an offset (passed in a second
265 * register). This load operation is used together with the two
266 * previous ones in the butterfly main loop.
267 */
268 operation OFFS_LOAD3
269 {in AR *load_ptr, in AR offs}
270 {in MemDataIn128, out VAddr, out load_buf3}
271 {
272     assign VAddr = TIEadd(load_ptr, offs, 1'b0);
273     assign load_buf3 = MemDataIn128;
274 }
275
276 /* Same as above except that the result is loaded into the fourth load
277 * buffer.
278 */
279 operation OFFS_LOAD4
280 {in AR *load_ptr, in AR offs}
281 {in MemDataIn128, out VAddr, out load_buf4}
282 {
283     assign VAddr = TIEadd(load_ptr, offs, 1'b0);
284     assign load_buf4 = MemDataIn128;
285 }
286
287 /* The following store operations complement the load operations above. */
288
289 operation STORE1
290 {inout AR *data_ptr}
291 {out VAddr, out MemDataOut128, in store_buf1}
292 {
293     assign VAddr = data_ptr;
294     assign MemDataOut128 = store_buf1;
295     assign data_ptr = TIEadd(data_ptr, 16, 1'b0);
296 }
297
298 operation STORE2
299 {inout AR *data_ptr}
300 {out VAddr, out MemDataOut128, in store_buf2}
301 {
302     assign VAddr = data_ptr;
303     assign MemDataOut128 = store_buf2;
304     assign data_ptr = TIEadd(data_ptr, 16, 1'b0);
305 }
306
307 operation OFFS_STORE3
308 {in AR *store_ptr, in AR offs}
309 {out MemDataOut128, out VAddr, in store_buf3}
310 {
311     assign VAddr = TIEadd(store_ptr, offs, 1'b0);
312     assign MemDataOut128 = store_buf3;
313 }
314
315 operation OFFS_STORE4
316 {in AR *store_ptr, in AR offs}
317 {out MemDataOut128, out VAddr, in store_buf4}
318 {
319     assign VAddr = TIEadd(store_ptr, offs, 1'b0);

```

```

320     assign MemDataOut128 = store_buf4;
321 }
322
323 /*--DECIMATION IN TIME/FREQUENCY-----*/
324
325 /* Shift a single 16 bit value from a memory location given by the
326 * forward/reverse DIT loop counters into the first store buffer.
327 */
328 operation DSWAP11
329 {in AR *data_ptr, in AR dit_ctr, in AR dit_rev_ctr}
330 {out VAddr, in MemDataIn16, inout store_buf1}
331 {
332     wire [15:0] ctr_diff = TIEadd(dit_rev_ctr, ~dit_ctr, 1'b1);
333     wire [15:0] offs = TIEmux(ctr_diff[15], dit_rev_ctr, dit_ctr);
334
335     assign VAddr = TIEadd(data_ptr, offs << 1, 1'b0);
336
337     wire [111:0] tmp = store_buf1 >> 16;
338     assign store_buf1 = {MemDataIn16, tmp};
339 }
340
341 /* Same as above except for the second store buffer. */
342 operation DSWAP12
343 {in AR *data_ptr, in AR dit_ctr, in AR dit_rev_ctr}
344 {out VAddr, in MemDataIn16, inout store_buf2}
345 {
346     wire [15:0] ctr_diff = TIEadd(dit_rev_ctr, ~dit_ctr, 1'b1);
347     wire [15:0] offs = TIEmux(ctr_diff[15], dit_rev_ctr, dit_ctr);
348
349     assign VAddr = TIEadd(data_ptr, offs << 1, 1'b0);
350
351     wire [111:0] tmp = store_buf2 >> 16;
352     assign store_buf2 = {MemDataIn16, tmp};
353 }
354
355 /* Store a single 16 bit value from the first load buffer to memory and then
356 * shift the buffer to the right by the same amount. */
357 operation DSWAP21
358 {in AR *data_ptr, in AR dit_ctr, in AR dit_rev_ctr}
359 {out VAddr, out MemDataOut16, inout load_buf1}
360 {
361     wire [15:0] ctr_diff = TIEadd(dit_rev_ctr, ~dit_ctr, 1'b1);
362     wire [15:0] offs = TIEmux(ctr_diff[15], dit_rev_ctr, dit_ctr);
363
364     assign VAddr = TIEadd(data_ptr, offs << 1, 1'b0);
365
366     assign MemDataOut16 = load_buf1[15:0];
367     assign load_buf1 = load_buf1 >> 16;
368 }
369
370 /* Same as above except for the second load buffer. */
371 operation DSWAP22
372 {in AR *data_ptr, in AR dit_ctr, in AR dit_rev_ctr}
373 {out VAddr, out MemDataOut16, inout load_buf2}
374 {
375     wire [15:0] ctr_diff = TIEadd(dit_rev_ctr, ~dit_ctr, 1'b1);
376     wire [15:0] offs = TIEmux(ctr_diff[15], dit_rev_ctr, dit_ctr);
377
378     assign VAddr = TIEadd(data_ptr, offs << 1, 1'b0);
379
380     assign MemDataOut16 = load_buf2[15:0];
381     assign load_buf2 = load_buf2 >> 16;
382 }
383
384 /* Increment both forward and reverse DIT loop counters. The reverse counter
385 * is incremented using a custom reverse ripple carry adder, this makes
386 * expensive hardware counter inversion (which depends on N!) in every loop
387 * unnecessary. It might also be a good idea to use an adder architecture with
388 * a shorter critical path but whether this would have a significant
389 * performance impact is hard to judge.
390 */
391 operation DINC {inout AR dit_ctr, inout AR dit_rev_ctr, in AR n} {}
392 {

```

```

393 wire c1, c2, c3, c4, c5, c6, c7, c8, c9;
394 wire s0, s1, s2, s3, s4, s5, s6, s7, s8, s9;
395 wire [9:0] mask = n >> 1;
396
397 assign {c1, s9} = TIEcsa(dit_rev_ctr[9], mask[9], 1'b0);
398 assign {c2, s8} = TIEcsa(dit_rev_ctr[8], mask[8], c1);
399 assign {c3, s7} = TIEcsa(dit_rev_ctr[7], mask[7], c2);
400 assign {c4, s6} = TIEcsa(dit_rev_ctr[6], mask[6], c3);
401 assign {c5, s5} = TIEcsa(dit_rev_ctr[5], mask[5], c4);
402 assign {c6, s4} = TIEcsa(dit_rev_ctr[4], mask[4], c5);
403 assign {c7, s3} = TIEcsa(dit_rev_ctr[3], mask[3], c6);
404 assign {c8, s2} = TIEcsa(dit_rev_ctr[2], mask[2], c7);
405 assign {c9, s1} = TIEcsa(dit_rev_ctr[1], mask[1], c8);
406 assign s0      = TIEadd(dit_rev_ctr[0], mask[0], c9);
407
408 assign dit_ctr = TIEadd(dit_ctr, 1'b1, 1'b0);
409 assign dit_rev_ctr = {s9, s8, s7, s6, s5, s4, s3, s2, s1, s0};
410 }
411
412 /*--SCALING-----*/
413
414 /* Check if "shift" should be set to one by looking at four complex input
415 * numbers at a time.
416 */
417 operation SCALING
418 {inout BR shift, inout AR i, in AR n}
419 {in load_buf1, in load_buf2}
420 {
421   wire [15:0] abs_real1 = abs(load_buf1[15:0]);
422   wire [15:0] abs_real2 = abs(load_buf1[31:16]);
423   wire [15:0] abs_real3 = abs(load_buf1[47:32]);
424   wire [15:0] abs_real4 = abs(load_buf1[63:48]);
425   wire [15:0] abs_real5 = abs(load_buf1[79:64]);
426   wire [15:0] abs_real6 = abs(load_buf1[95:80]);
427   wire [15:0] abs_real7 = abs(load_buf1[111:96]);
428   wire [15:0] abs_real8 = abs(load_buf1[127:112]);
429
430   wire [15:0] abs_imag1 = abs(load_buf2[15:0]);
431   wire [15:0] abs_imag2 = abs(load_buf2[31:16]);
432   wire [15:0] abs_imag3 = abs(load_buf2[47:32]);
433   wire [15:0] abs_imag4 = abs(load_buf2[63:48]);
434   wire [15:0] abs_imag5 = abs(load_buf2[79:64]);
435   wire [15:0] abs_imag6 = abs(load_buf2[95:80]);
436   wire [15:0] abs_imag7 = abs(load_buf2[111:96]);
437   wire [15:0] abs_imag8 = abs(load_buf2[127:112]);
438
439   wire tmp = shift |
440             abs_real1[14] | abs_real2[14] |
441             abs_real3[14] | abs_real4[14] |
442             abs_real5[14] | abs_real6[14] |
443             abs_real7[14] | abs_real8[14] |
444             abs_imag1[14] | abs_imag2[14] |
445             abs_imag3[14] | abs_imag4[14] |
446             abs_imag5[14] | abs_imag6[14] |
447             abs_imag7[14] | abs_imag8[14];
448
449   assign shift = tmp;
450   assign i = TIEmux(tmp, TIEadd(i, 8, 1'b0), n);
451 }
452
453 /*--BUTTERFLY-----*/
454
455 /* Butterfly operation used in the first unrolled butterfly loop iteration.
456 * Performs four butterfly operations in parallel.
457 */
458 operation BFLY_DIT_UNROLL1
459 {in BR inv, in BR shift}
460 {out store_buf1, out store_buf2, in load_buf1, in load_buf2}
461 {
462   wire [15:0] tr = 16'h7fff;
463   wire [15:0] ti = 16'h0000;
464
465   wire [15:0] a0r = load_buf1[15:0];

```

```

466 wire [15:0] a0i = load_buf2[15:0];
467 wire [15:0] b0r = load_buf1[31:16];
468 wire [15:0] b0i = load_buf2[31:16];
469
470 wire [15:0] a1r = load_buf1[47:32];
471 wire [15:0] a1i = load_buf2[47:32];
472 wire [15:0] b1r = load_buf1[63:48];
473 wire [15:0] b1i = load_buf2[63:48];
474
475 wire [15:0] a2r = load_buf1[79:64];
476 wire [15:0] a2i = load_buf2[79:64];
477 wire [15:0] b2r = load_buf1[95:80];
478 wire [15:0] b2i = load_buf2[95:80];
479
480 wire [15:0] a3r = load_buf1[111:96];
481 wire [15:0] a3i = load_buf2[111:96];
482 wire [15:0] b3r = load_buf1[127:112];
483 wire [15:0] b3i = load_buf2[127:112];
484
485 wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;
486
487 assign {c0r, c0i, d0r, d0i} = bfly1(a0r, a0i, b0r, b0i, tr, ti, inv, shift);
488 assign {c1r, c1i, d1r, d1i} = bfly2(a1r, a1i, b1r, b1i, tr, ti, inv, shift);
489 assign {c2r, c2i, d2r, d2i} = bfly3(a2r, a2i, b2r, b2i, tr, ti, inv, shift);
490 assign {c3r, c3i, d3r, d3i} = bfly4(a3r, a3i, b3r, b3i, tr, ti, inv, shift);
491
492 assign store_buf1 = {d3r, c3r, d2r, c2r, d1r, c1r, d0r, c0r};
493 assign store_buf2 = {d3i, c3i, d2i, c2i, d1i, c1i, d0i, c0i};
494 }
495
496 /* Butterfly operation used in the second unrolled butterfly loop iteration.
497 * Performs four butterfly operations in parallel.
498 */
499 operation BFLY_DIT_UNROLL2
500 {in BR inv, in BR shift}
501 {out store_buf1, out store_buf2, in load_buf1, in load_buf2}
502 {
503 wire [15:0] t1 = 16'h7fff;
504 wire [15:0] t2 = 16'h0000;
505
506 wire [15:0] a0r = load_buf1[15:0];
507 wire [15:0] a0i = load_buf2[15:0];
508 wire [15:0] b0r = load_buf1[47:32];
509 wire [15:0] b0i = load_buf2[47:32];
510
511 wire [15:0] a1r = load_buf1[31:16];
512 wire [15:0] a1i = load_buf2[31:16];
513 wire [15:0] b1r = load_buf1[63:48];
514 wire [15:0] b1i = load_buf2[63:48];
515
516 wire [15:0] a2r = load_buf1[79:64];
517 wire [15:0] a2i = load_buf2[79:64];
518 wire [15:0] b2r = load_buf1[111:96];
519 wire [15:0] b2i = load_buf2[111:96];
520
521 wire [15:0] a3r = load_buf1[95:80];
522 wire [15:0] a3i = load_buf2[95:80];
523 wire [15:0] b3r = load_buf1[127:112];
524 wire [15:0] b3i = load_buf2[127:112];
525
526 wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;
527
528 assign {c0r, c0i, d0r, d0i} = bfly1(a0r, a0i, b0r, b0i, t1, t2, inv, shift);
529 assign {c1r, c1i, d1r, d1i} = bfly2(a1r, a1i, b1r, b1i, t2, t1, inv, shift);
530 assign {c2r, c2i, d2r, d2i} = bfly3(a2r, a2i, b2r, b2i, t1, t2, inv, shift);
531 assign {c3r, c3i, d3r, d3i} = bfly4(a3r, a3i, b3r, b3i, t2, t1, inv, shift);
532
533 assign store_buf1 = {d3r, d2r, c3r, c2r, d1r, d0r, c1r, c0r};
534 assign store_buf2 = {d3i, d2i, c3i, c2i, d1i, d0i, c1i, c0i};
535 }
536
537 /* Butterfly operation used in the fourth unrolled butterfly loop iteration.
538 * Performs four butterfly operations in parallel.

```

```

539  */
540  operation BFLY_DIT_UNROLL4
541  {in BR inv, in BR shift}
542  {out store_buf1, out store_buf2, in load_buf1, in load_buf2}
543  {
544      wire [15:0] t0r = 16'h7fff;
545      wire [15:0] t0i = 16'h0000;
546      wire [15:0] t1r = 16'h5a81;
547      wire [15:0] t1i = 16'h5a81;
548      wire [15:0] t2r = 16'h0000;
549      wire [15:0] t2i = 16'h7fff;
550      wire [15:0] t3r = 16'ha57f;
551      wire [15:0] t3i = 16'h5a81;
552
553      wire [15:0] a0r = load_buf1[15:0];
554      wire [15:0] a0i = load_buf2[15:0];
555      wire [15:0] b0r = load_buf1[79:64];
556      wire [15:0] b0i = load_buf2[79:64];
557
558      wire [15:0] a1r = load_buf1[31:16];
559      wire [15:0] a1i = load_buf2[31:16];
560      wire [15:0] b1r = load_buf1[95:80];
561      wire [15:0] b1i = load_buf2[95:80];
562
563      wire [15:0] a2r = load_buf1[47:32];
564      wire [15:0] a2i = load_buf2[47:32];
565      wire [15:0] b2r = load_buf1[111:96];
566      wire [15:0] b2i = load_buf2[111:96];
567
568      wire [15:0] a3r = load_buf1[63:48];
569      wire [15:0] a3i = load_buf2[63:48];
570      wire [15:0] b3r = load_buf1[127:112];
571      wire [15:0] b3i = load_buf2[127:112];
572
573      wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;
574
575      assign {c0r, c0i, d0r, d0i} = bfly1(a0r, a0i, b0r, b0i, t0r, t0i, inv, shift);
576      assign {c1r, c1i, d1r, d1i} = bfly2(a1r, a1i, b1r, b1i, t1r, t1i, inv, shift);
577      assign {c2r, c2i, d2r, d2i} = bfly3(a2r, a2i, b2r, b2i, t2r, t2i, inv, shift);
578      assign {c3r, c3i, d3r, d3i} = bfly4(a3r, a3i, b3r, b3i, t3r, t3i, inv, shift);
579
580      assign store_buf1 = {d3r, d2r, d1r, d0r, c3r, c2r, c1r, c0r};
581      assign store_buf2 = {d3i, d2i, d1i, d0i, c3i, c2i, c1i, c0i};
582  }
583
584  /* Butterfly operation used in the main butterfly loop. Perfoms eight butterfly
585     * operations in parallel.
586     */
587  operation BFLY_DIT_MAIN {inout AR m1, in AR k, in BR inv, in BR shift}
588  {out store_buf1, out store_buf2,
589   out store_buf3, out store_buf4,
590   in load_buf1, in load_buf2,
591   in load_buf3, in load_buf4}
592  {
593      wire [15:0] bf1_tr, bf1_ti, bf2_tr, bf2_ti,
594                bf3_tr, bf3_ti, bf4_tr, bf4_ti,
595                bf5_tr, bf5_ti, bf6_tr, bf6_ti,
596                bf7_tr, bf7_ti, bf8_tr, bf8_ti;
597
598      wire [9:0] m2 = TIEadd(m1, 1, 1'b0);
599      wire [9:0] m3 = TIEadd(m1, 2, 1'b0);
600      wire [9:0] m4 = TIEadd(m1, 3, 1'b0);
601      wire [9:0] m5 = TIEadd(m1, 4, 1'b0);
602      wire [9:0] m6 = TIEadd(m1, 5, 1'b0);
603      wire [9:0] m7 = TIEadd(m1, 6, 1'b0);
604      wire [9:0] m8 = TIEadd(m1, 7, 1'b0);
605
606      assign {bf1_tr, bf1_ti} = twiddles(m1, k);
607      assign {bf2_tr, bf2_ti} = twiddles(m2, k);
608      assign {bf3_tr, bf3_ti} = twiddles(m3, k);
609      assign {bf4_tr, bf4_ti} = twiddles(m4, k);
610      assign {bf5_tr, bf5_ti} = twiddles(m5, k);
611      assign {bf6_tr, bf6_ti} = twiddles(m6, k);

```

```

612 assign {bf7_tr, bf7_ti} = twiddles(m7, k);
613 assign {bf8_tr, bf8_ti} = twiddles(m8, k);
614
615 wire [15:0] bf1_r1, bf1_i1, bf1_r2, bf1_i2,
616             bf2_r1, bf2_i1, bf2_r2, bf2_i2,
617             bf3_r1, bf3_i1, bf3_r2, bf3_i2,
618             bf4_r1, bf4_i1, bf4_r2, bf4_i2,
619             bf5_r1, bf5_i1, bf5_r2, bf5_i2,
620             bf6_r1, bf6_i1, bf6_r2, bf6_i2,
621             bf7_r1, bf7_i1, bf7_r2, bf7_i2,
622             bf8_r1, bf8_i1, bf8_r2, bf8_i2;
623
624 assign {bf1_r1, bf1_i1, bf1_r2, bf1_i2} =
625     bfly1(load_buf1[15:0], load_buf2[15:0], load_buf3[15:0], load_buf4[15:0],
626           bf1_tr, bf1_ti, inv, shift);
627 assign {bf2_r1, bf2_i1, bf2_r2, bf2_i2} =
628     bfly2(load_buf1[31:16], load_buf2[31:16], load_buf3[31:16], load_buf4[31:16],
629           bf2_tr, bf2_ti, inv, shift);
630 assign {bf3_r1, bf3_i1, bf3_r2, bf3_i2} =
631     bfly3(load_buf1[47:32], load_buf2[47:32], load_buf3[47:32], load_buf4[47:32],
632           bf3_tr, bf3_ti, inv, shift);
633 assign {bf4_r1, bf4_i1, bf4_r2, bf4_i2} =
634     bfly4(load_buf1[63:48], load_buf2[63:48], load_buf3[63:48], load_buf4[63:48],
635           bf4_tr, bf4_ti, inv, shift);
636 assign {bf5_r1, bf5_i1, bf5_r2, bf5_i2} =
637     bfly(load_buf1[79:64], load_buf2[79:64], load_buf3[79:64], load_buf4[79:64],
638           bf5_tr, bf5_ti, inv, shift);
639 assign {bf6_r1, bf6_i1, bf6_r2, bf6_i2} =
640     bfly(load_buf1[95:80], load_buf2[95:80], load_buf3[95:80], load_buf4[95:80],
641           bf6_tr, bf6_ti, inv, shift);
642 assign {bf7_r1, bf7_i1, bf7_r2, bf7_i2} =
643     bfly(load_buf1[111:96], load_buf2[111:96], load_buf3[111:96], load_buf4[111:96],
644           bf7_tr, bf7_ti, inv, shift);
645 assign {bf8_r1, bf8_i1, bf8_r2, bf8_i2} =
646     bfly(load_buf1[127:112], load_buf2[127:112], load_buf3[127:112], load_buf4[127:112],
647           bf8_tr, bf8_ti, inv, shift);
648
649 assign store_buf1 = {bf8_r1, bf7_r1, bf6_r1, bf5_r1, bf4_r1, bf3_r1, bf2_r1, bf1_r1};
650 assign store_buf2 = {bf8_i1, bf7_i1, bf6_i1, bf5_i1, bf4_i1, bf3_i1, bf2_i1, bf1_i1};
651 assign store_buf3 = {bf8_r2, bf7_r2, bf6_r2, bf5_r2, bf4_r2, bf3_r2, bf2_r2, bf1_r2};
652 assign store_buf4 = {bf8_i2, bf7_i2, bf6_i2, bf5_i2, bf4_i2, bf3_i2, bf2_i2, bf1_i2};
653
654 assign m1 = TIEadd(m1, 8, 1'b0);
655 }
656
657 schedule bfly_dit_main_sched {BFLY_DIT_MAIN} {
658     def store_buf1 2;
659     def store_buf2 2;
660     def store_buf3 2;
661     def store_buf4 2;
662 }
663
664 /*--BUTTERFLY (DIF)-----*/
665
666 /* Butterfly operation used in the unrolled butterfly loop iteration m.
667  * Performs four butterfly operations in parallel.
668  */
669 operation BFLY_DIF_UNROLL1
670 {in BR inv, in BR shift}
671 {out store_buf1, out store_buf2, in load_buf1, in load_buf2}
672 {
673     wire [15:0] tr = 16'h7fff;
674     wire [15:0] ti = 16'h0000;
675
676     wire [15:0] a0r = load_buf1[15:0];
677     wire [15:0] a0i = load_buf2[15:0];
678     wire [15:0] b0r = load_buf1[31:16];
679     wire [15:0] b0i = load_buf2[31:16];
680
681     wire [15:0] a1r = load_buf1[47:32];
682     wire [15:0] a1i = load_buf2[47:32];
683     wire [15:0] b1r = load_buf1[63:48];
684     wire [15:0] b1i = load_buf2[63:48];

```

```

685
686 wire [15:0] a2r = load_buf1[79:64];
687 wire [15:0] a2i = load_buf2[79:64];
688 wire [15:0] b2r = load_buf1[95:80];
689 wire [15:0] b2i = load_buf2[95:80];
690
691 wire [15:0] a3r = load_buf1[111:96];
692 wire [15:0] a3i = load_buf2[111:96];
693 wire [15:0] b3r = load_buf1[127:112];
694 wire [15:0] b3i = load_buf2[127:112];
695
696 wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;
697
698 assign {c0r, c0i, d0r, d0i} = bfly_dif1(a0r, a0i, b0r, b0i, tr, ti, inv, shift);
699 assign {c1r, c1i, d1r, d1i} = bfly_dif2(a1r, a1i, b1r, b1i, tr, ti, inv, shift);
700 assign {c2r, c2i, d2r, d2i} = bfly_dif3(a2r, a2i, b2r, b2i, tr, ti, inv, shift);
701 assign {c3r, c3i, d3r, d3i} = bfly_dif4(a3r, a3i, b3r, b3i, tr, ti, inv, shift);
702
703 assign store_buf1 = {d3r, c3r, d2r, c2r, d1r, c1r, d0r, c0r};
704 assign store_buf2 = {d3i, c3i, d2i, c2i, d1i, c1i, d0i, c0i};
705 }
706
707 /* Butterfly operation used in the unrolled butterfly loop iteration m - 1.
708 * Performs four butterfly operations in parallel.
709 */
710 operation BFLY_DIF_UNROLL2
711 {in BR inv, in BR shift}
712 {out store_buf1, out store_buf2, in load_buf1, in load_buf2}
713 {
714   wire [15:0] t1 = 16'h7fff;
715   wire [15:0] t2 = 16'h0000;
716
717   wire [15:0] a0r = load_buf1[15:0];
718   wire [15:0] a0i = load_buf2[15:0];
719   wire [15:0] b0r = load_buf1[47:32];
720   wire [15:0] b0i = load_buf2[47:32];
721
722   wire [15:0] a1r = load_buf1[31:16];
723   wire [15:0] a1i = load_buf2[31:16];
724   wire [15:0] b1r = load_buf1[63:48];
725   wire [15:0] b1i = load_buf2[63:48];
726
727   wire [15:0] a2r = load_buf1[79:64];
728   wire [15:0] a2i = load_buf2[79:64];
729   wire [15:0] b2r = load_buf1[111:96];
730   wire [15:0] b2i = load_buf2[111:96];
731
732   wire [15:0] a3r = load_buf1[95:80];
733   wire [15:0] a3i = load_buf2[95:80];
734   wire [15:0] b3r = load_buf1[127:112];
735   wire [15:0] b3i = load_buf2[127:112];
736
737   wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;
738
739   assign {c0r, c0i, d0r, d0i} = bfly_dif1(a0r, a0i, b0r, b0i, t1, t2, inv, shift);
740   assign {c1r, c1i, d1r, d1i} = bfly_dif2(a1r, a1i, b1r, b1i, t2, t1, inv, shift);
741   assign {c2r, c2i, d2r, d2i} = bfly_dif3(a2r, a2i, b2r, b2i, t1, t2, inv, shift);
742   assign {c3r, c3i, d3r, d3i} = bfly_dif4(a3r, a3i, b3r, b3i, t2, t1, inv, shift);
743
744   assign store_buf1 = {d3r, d2r, c3r, c2r, d1r, d0r, c1r, c0r};
745   assign store_buf2 = {d3i, d2i, c3i, c2i, d1i, d0i, c1i, c0i};
746 }
747
748 /* Butterfly operation used in the unrolled butterfly loop iteration m - 2.
749 * Performs four butterfly operations in parallel.
750 */
751 operation BFLY_DIF_UNROLL4
752 {in BR inv, in BR shift}
753 {out store_buf1, out store_buf2, in load_buf1, in load_buf2}
754 {
755   wire [15:0] t0r = 16'h7fff;
756   wire [15:0] t0i = 16'h0000;
757   wire [15:0] t1r = 16'h5a81;

```

```

758 wire [15:0] t1i = 16'h5a81;
759 wire [15:0] t2r = 16'h0000;
760 wire [15:0] t2i = 16'h7fff;
761 wire [15:0] t3r = 16'ha57f;
762 wire [15:0] t3i = 16'h5a81;
763
764 wire [15:0] a0r = load_buf1[15:0];
765 wire [15:0] a0i = load_buf2[15:0];
766 wire [15:0] b0r = load_buf1[79:64];
767 wire [15:0] b0i = load_buf2[79:64];
768
769 wire [15:0] a1r = load_buf1[31:16];
770 wire [15:0] a1i = load_buf2[31:16];
771 wire [15:0] b1r = load_buf1[95:80];
772 wire [15:0] b1i = load_buf2[95:80];
773
774 wire [15:0] a2r = load_buf1[47:32];
775 wire [15:0] a2i = load_buf2[47:32];
776 wire [15:0] b2r = load_buf1[111:96];
777 wire [15:0] b2i = load_buf2[111:96];
778
779 wire [15:0] a3r = load_buf1[63:48];
780 wire [15:0] a3i = load_buf2[63:48];
781 wire [15:0] b3r = load_buf1[127:112];
782 wire [15:0] b3i = load_buf2[127:112];
783
784 wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;
785
786 assign {c0r, c0i, d0r, d0i} = bfly_dif1(a0r, a0i, b0r, b0i, t0r, t0i, inv, shift);
787 assign {c1r, c1i, d1r, d1i} = bfly_dif2(a1r, a1i, b1r, b1i, t1r, t1i, inv, shift);
788 assign {c2r, c2i, d2r, d2i} = bfly_dif3(a2r, a2i, b2r, b2i, t2r, t2i, inv, shift);
789 assign {c3r, c3i, d3r, d3i} = bfly_dif4(a3r, a3i, b3r, b3i, t3r, t3i, inv, shift);
790
791 assign store_buf1 = {d3r, d2r, d1r, d0r, c3r, c2r, c1r, c0r};
792 assign store_buf2 = {d3i, d2i, d1i, d0i, c3i, c2i, c1i, c0i};
793 }
794
795 /* Butterfly operation used in the main butterfly loop. Perfoms eight butterfly
796 * operations in parallel.
797 */
798 operation BFLY_DIF_MAIN {inout AR m1, in AR k, in BR inv, in BR shift}
799 {out store_buf1, out store_buf2,
800 out store_buf3, out store_buf4,
801 in load_buf1, in load_buf2,
802 in load_buf3, in load_buf4}
803 {
804 wire [15:0] bf1_tr, bf1_ti, bf2_tr, bf2_ti,
805 bf3_tr, bf3_ti, bf4_tr, bf4_ti,
806 bf5_tr, bf5_ti, bf6_tr, bf6_ti,
807 bf7_tr, bf7_ti, bf8_tr, bf8_ti;
808
809 wire [9:0] m2 = TIEadd(m1, 1, 1'b0);
810 wire [9:0] m3 = TIEadd(m1, 2, 1'b0);
811 wire [9:0] m4 = TIEadd(m1, 3, 1'b0);
812 wire [9:0] m5 = TIEadd(m1, 4, 1'b0);
813 wire [9:0] m6 = TIEadd(m1, 5, 1'b0);
814 wire [9:0] m7 = TIEadd(m1, 6, 1'b0);
815 wire [9:0] m8 = TIEadd(m1, 7, 1'b0);
816
817 assign {bf1_tr, bf1_ti} = twiddles(m1, k);
818 assign {bf2_tr, bf2_ti} = twiddles(m2, k);
819 assign {bf3_tr, bf3_ti} = twiddles(m3, k);
820 assign {bf4_tr, bf4_ti} = twiddles(m4, k);
821 assign {bf5_tr, bf5_ti} = twiddles(m5, k);
822 assign {bf6_tr, bf6_ti} = twiddles(m6, k);
823 assign {bf7_tr, bf7_ti} = twiddles(m7, k);
824 assign {bf8_tr, bf8_ti} = twiddles(m8, k);
825
826 wire [15:0] bf1_r1, bf1_i1, bf1_r2, bf1_i2,
827 bf2_r1, bf2_i1, bf2_r2, bf2_i2,
828 bf3_r1, bf3_i1, bf3_r2, bf3_i2,
829 bf4_r1, bf4_i1, bf4_r2, bf4_i2,
830 bf5_r1, bf5_i1, bf5_r2, bf5_i2,

```



```

831         bf6_r1, bf6_i1, bf6_r2, bf6_i2,
832         bf7_r1, bf7_i1, bf7_r2, bf7_i2,
833         bf8_r1, bf8_i1, bf8_r2, bf8_i2;
834
835     assign {bf1_r1, bf1_i1, bf1_r2, bf1_i2} =
836         bfly_dif1(load_buf1[15:0], load_buf2[15:0], load_buf3[15:0], load_buf4[15:0],
837             bf1_tr, bf1_ti, inv, shift);
838     assign {bf2_r1, bf2_i1, bf2_r2, bf2_i2} =
839         bfly_dif2(load_buf1[31:16], load_buf2[31:16], load_buf3[31:16], load_buf4[31:16],
840             bf2_tr, bf2_ti, inv, shift);
841     assign {bf3_r1, bf3_i1, bf3_r2, bf3_i2} =
842         bfly_dif3(load_buf1[47:32], load_buf2[47:32], load_buf3[47:32], load_buf4[47:32],
843             bf3_tr, bf3_ti, inv, shift);
844     assign {bf4_r1, bf4_i1, bf4_r2, bf4_i2} =
845         bfly_dif4(load_buf1[63:48], load_buf2[63:48], load_buf3[63:48], load_buf4[63:48],
846             bf4_tr, bf4_ti, inv, shift);
847     assign {bf5_r1, bf5_i1, bf5_r2, bf5_i2} =
848         bfly_dif(load_buf1[79:64], load_buf2[79:64], load_buf3[79:64], load_buf4[79:64],
849             bf5_tr, bf5_ti, inv, shift);
850     assign {bf6_r1, bf6_i1, bf6_r2, bf6_i2} =
851         bfly_dif(load_buf1[95:80], load_buf2[95:80], load_buf3[95:80], load_buf4[95:80],
852             bf6_tr, bf6_ti, inv, shift);
853     assign {bf7_r1, bf7_i1, bf7_r2, bf7_i2} =
854         bfly_dif(load_buf1[111:96], load_buf2[111:96], load_buf3[111:96], load_buf4[111:96],
855             bf7_tr, bf7_ti, inv, shift);
856     assign {bf8_r1, bf8_i1, bf8_r2, bf8_i2} =
857         bfly_dif(load_buf1[127:112], load_buf2[127:112], load_buf3[127:112], load_buf4[127:112],
858             bf8_tr, bf8_ti, inv, shift);
859
860     assign store_buf1 = {bf8_r1, bf7_r1, bf6_r1, bf5_r1, bf4_r1, bf3_r1, bf2_r1, bf1_r1};
861     assign store_buf2 = {bf8_i1, bf7_i1, bf6_i1, bf5_i1, bf4_i1, bf3_i1, bf2_i1, bf1_i1};
862     assign store_buf3 = {bf8_r2, bf7_r2, bf6_r2, bf5_r2, bf4_r2, bf3_r2, bf2_r2, bf1_r2};
863     assign store_buf4 = {bf8_i2, bf7_i2, bf6_i2, bf5_i2, bf4_i2, bf3_i2, bf2_i2, bf1_i2};
864
865     assign m1 = TIEadd(m1, 8, 1'b0);
866 }
867
868 schedule bfly_dif_main_sched {BFLY_DIF_MAIN} {
869     def store_buf1 2;
870     def store_buf2 2;
871     def store_buf3 2;
872     def store_buf4 2;
873 }

```