TECHNISCHE UNIVERSITÄT DRESDEN

Vodafone Chair for Mobile Communications Systems

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

Timo Nicolai

CONTENTS

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 where 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 where 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 it's 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 of 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 it's 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 where 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 only[4]: ≈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 250might not be desirable if a slower implementation can save valuable chip area instead.

[1]Sum 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.

[2]Same cycle metric. The IDE was not capable of giving overall cycle numbers for function simplemented 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.

[3]Rounded, Value in brackets refers to DIF.

[4]Meaning 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

```
// fft.S: FFT function assembly source code

// Comment this in to switch from DIT to DIF. Note that the definition of this
// symbol should ideally be passed as a compiler/assembler option instead but
// this does not seem to be possible without manually modifying the Makefile
// generated by Xtensa Xplorer.

//#define DIF

#define LOG2_N_WAVE 10

.align 4
.global fft
.type fft, @function

fft:
  // a2 = fr, a3 = fi, a4 = m
  entry a1, 128

  // Verify that 5 <= m <= 10, implying that 32 <= n <= 1024.
  // The upper limit is already present in the reference implementation
  // and the lower limit is necessary because some processing steps of this
  // function load up to 16 complex values from memory per clock cycle and
  // process these in one-stage software pipelines.

  bgei a4, 5, m_valid
  movi.n a5, 11
  blt a4, a5, m_valid
  movi.n a2, -1
  retw
m_valid:

#ifdef DIF
  // save m because it is needed to determine the initial value of k
  mov.n a13, a4
#endif

  // a4 = n = 1 << m
  ssl a4
  movi.n a4, 1
  sll a4, a4

  // b0 = 0, b1 = 1, b2 = inverse, b3 = shift
  movi.n a6, 2
  wsr.BR a6
  beqz a5, no_inverse
  orb b2, b2, b1
no_inverse:

  // scale = 0
  movi.n a5, 0

/*--DECIMATION IN TIME-------------------------------------------------------*/

#ifndef DIF
  // a7 = forward counter, a8 = reverse counter
  { movi.n a7, 0; movi.n a8, 0; nop }
  // a9 = real load pointer, a10 = imaginary load pointer
  { mov.n a9, a2; mov.n a10, a3; nop }
  // a11 = real store pointer, a12 = imaginary store pointer
  { mov.n a11, a2; mov.n a12, a3; nop }
dit:
  { load1 a9; load2 a10; nop }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { store1 a11; store2 a12; bne a7, a4, dit }
#endif

/*--MAIN LOOP-----------------------------------------------------------------*/

#ifdef DIF
  // a6 = l = n/2
  movi.n a6, 1
  ssr a6
  srl a6, a4
  // a7 = k = LOG2_N_WAVE - m
  neg a13, a13
  addi.n a7, a13, LOG2_N_WAVE
#else
  // a6 = l = 1
  movi.n a6, 1
  // a7 = k = LOG2_N_WAVE - 1
  movi.n a7, LOG2_N_WAVE - 1
#endif

main_loop:

#ifdef DIF
  beqz a6, main_loop_done // l == 0 ? => done
#else
  beq a4, a6, main_loop_done // l == n ? => done
#endif

  slli a8, a6, 1 // a8 = l*2

/*--SCALING-------------------------------------------------------------------*/

  bf b2, no_scaling // skip scaling for non-inverse FFT
  { mov.n a9, a2; mov.n a10, a3; andb b3, b3, b0 } // b3 = shift = 0
  { load1 a9; load2 a10; movi.n a11, 8 }
check_scaling:
  // pass n to "scaling" so that the loop can be broken by setting a11 = n
  { load1 a9; load2 a10; scaling b3, a11, a4 }
  bne a11, a4, check_scaling
  scaling b3, a11, a4
  bf b3, scaling_done
  addi.n a5, a5, 1 // ++scale
  j scaling_done
no_scaling:
  orb b3, b3, b1 // shift = 1
scaling_done:

/*--BUTTERFLIES---------------------------------------------------------------*/

#ifdef DIF
  #define BFLY_MAIN bfly_dif_main
  #define BFLY_UNROLL1 bfly_dif_unroll1
  #define BFLY_UNROLL2 bfly_dif_unroll2
  #define BFLY_UNROLL4 bfly_dif_unroll4
#else
  #define BFLY_MAIN bfly_dit_main
  #define BFLY_UNROLL1 bfly_dit_unroll1
  #define BFLY_UNROLL2 bfly_dit_unroll2
  #define BFLY_UNROLL4 bfly_dit_unroll4
#endif

  { mov.n a9, a2; mov.n a10, a3; nop }
  { mov.n a11, a2; mov.n a12, a3; nop }

  // loop iterations for l in {1, 2, 4} are treated differently
  beqi a6, 1, bfly_unroll1
  beqi a6, 2, bfly_unroll2
  beqi a6, 4, bfly_unroll4

  movi.n a13, 0
bfly_outer:
  { offs_load3 a9, a8; offs_load4 a10, a8; movi.n a14, 0 }
  { load1 a9; load2 a10; beqi a6, 8, bfly_inner_done }
bfly_inner:
  { offs_load3 a9, a8; offs_load4 a10, a8; BFLY_MAIN a14, a7, b2, b3 }
  { load1 a9; load2 a10; addi a15, a14, 8 }
  { offs_store3 a11, a8; offs_store4 a12, a8; nop }
  { store1 a11; store2 a12; bne a6, a15, bfly_inner }
bfly_inner_done:
  BFLY_MAIN a14, a7, b2, b3
  { offs_store3 a11, a8; offs_store4 a12, a8; add.n a13, a13, a8 }
  { store1 a11; store2 a12; nop }
  { inc4 a9, a10, a11, a12, a8; nop; bne a4, a13, bfly_outer }
  j next_iter

bfly_unroll1:
  { load1 a9; load2 a10; movi.n a13, 16 }
  { load1 a9; load2 a10; BFLY_UNROLL1 b2, b3 }
bfly_unroll1_loop:
  { store1 a11; store2 a12; BFLY_UNROLL1 b2, b3 }
  { load1 a9; load2 a10; addi.n a13, a13, 16 }
  { store1 a11; store2 a12; BFLY_UNROLL1 b2, b3 }
  { load1 a9; load2 a10; bne a4, a13, bfly_unroll1_loop }
bfly_unroll1_final:
  { store1 a11; store2 a12; BFLY_UNROLL1 b2, b3 }
  { store1 a11; store2 a12; j next_iter }

bfly_unroll2:
  { load1 a9; load2 a10; movi.n a13, 16 }
  { load1 a9; load2 a10; BFLY_UNROLL2 b2, b3 }
bfly_unroll2_loop:
  { store1 a11; store2 a12; BFLY_UNROLL2 b2, b3 }
  { load1 a9; load2 a10; addi.n a13, a13, 16 }
  { store1 a11; store2 a12; BFLY_UNROLL2 b2, b3 }
  { load1 a9; load2 a10; bne a4, a13, bfly_unroll2_loop }
bfly_unroll2_final:
  { store1 a11; store2 a12; BFLY_UNROLL2 b2, b3 }
  { store1 a11; store2 a12; j next_iter }

bfly_unroll4:
  { load1 a9; load2 a10; movi.n a13, 16 }
  { load1 a9; load2 a10; BFLY_UNROLL4 b2, b3 }
bfly_unroll4_loop:
  { store1 a11; store2 a12; BFLY_UNROLL4 b2, b3 }
  { load1 a9; load2 a10; addi.n a13, a13, 16 }
  { store1 a11; store2 a12; BFLY_UNROLL4 b2, b3 }
  { load1 a9; load2 a10; bne a4, a13, bfly_unroll4_loop }
bfly_unroll4_final:
  { store1 a11; store2 a12; BFLY_UNROLL4 b2, b3 }
  { store1 a11; store2 a12; nop }

next_iter:
#ifdef DIF
  // l = l/2; ++k;
  movi.n a8, 1
  ssr a8
  { srl a6, a6; addi.n a7, a7, 1; j main_loop }
#else
  // l = l*2; --k;
  { mov.n a6, a8; addi.n a7, a7, -1; j main_loop }
#endif

/*--DECIMATION IN FREQUENCY--------------------------------------------------*/

main_loop_done:

#ifdef DIF
  // a7 = forward counter, a8 = reverse counter
  { movi.n a7, 0; movi.n a8, 0; nop }
  // a9 = real load pointer, a10 = imaginary load pointer
  { mov.n a9, a2; mov.n a10, a3; nop }
  // a11 = real store pointer, a12 = imaginary store pointer
  { mov.n a11, a2; mov.n a12, a3; nop }
dif:
  { load1 a9; load2 a10; nop }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { dswap11 a2, a7, a8; dswap12 a3, a7, a8; nop }
  { dswap21 a2, a7, a8; dswap22 a3, a7, a8; dinc a7, a8, a4 }
  { store1 a11; store2 a12; bne a4, a7, dif }
#endif

  mov.n a2, a5
  retw
```

Listing 2: fft.tie

```
/*--TABLES-------------------------------------------------------------------*/

/* Sine wave hardware lookup table. Only contains values from zero up to pi/2,
 * accessing values in the full range from 0 to 2pi is made possible by several
 * bit twiddling tricks in the auxilliary function "sin" which is used to
 * by hardware operations which access this table.
 */
table SINE_WAVE 16 257 {
      0,    201,    402,    603,    804,   1005,   1206,   1406,
   1607,   1808,   2009,   2209,   2410,   2610,   2811,   3011,
   3211,   3411,   3611,   3811,   4011,   4210,   4409,   4608,
   4807,   5006,   5205,   5403,   5601,   5799,   5997,   6195,
   6392,   6589,   6786,   6982,   7179,   7375,   7571,   7766,
   7961,   8156,   8351,   8545,   8739,   8932,   9126,   9319,
   9511,   9703,   9895,  10087,  10278,  10469,  10659,  10849,
  11038,  11227,  11416,  11604,  11792,  11980,  12166,  12353,
  12539,  12724,  12909,  13094,  13278,  13462,  13645,  13827,
  14009,  14191,  14372,  14552,  14732,  14911,  15090,  15268,
  15446,  15623,  15799,  15975,  16150,  16325,  16499,  16672,
  16845,  17017,  17189,  17360,  17530,  17699,  17868,  18036,
  18204,  18371,  18537,  18702,  18867,  19031,  19194,  19357,
  19519,  19680,  19840,  20000,  20159,  20317,  20474,  20631,
  20787,  20942,  21096,  21249,  21402,  21554,  21705,  21855,
  22004,  22153,  22301,  22448,  22594,  22739,  22883,  23027,
  23169,  23311,  23452,  23592,  23731,  23869,  24006,  24143,
  24278,  24413,  24546,  24679,  24811,  24942,  25072,  25201,
  25329,  25456,  25582,  25707,  25831,  25954,  26077,  26198,
  26318,  26437,  26556,  26673,  26789,  26905,  27019,  27132,
  27244,  27355,  27466,  27575,  27683,  27790,  27896,  28001,
  28105,  28208,  28309,  28410,  28510,  28608,  28706,  28802,
  28897,  28992,  29085,  29177,  29268,  29358,  29446,  29534,
  29621,  29706,  29790,  29873,  29955,  30036,  30116,  30195,
  30272,  30349,  30424,  30498,  30571,  30643,  30713,  30783,
  30851,  30918,  30984,  31049,  31113,  31175,  31236,  31297,
  31356,  31413,  31470,  31525,  31580,  31633,  31684,  31735,
  31785,  31833,  31880,  31926,  31970,  32014,  32056,  32097,
  32137,  32176,  32213,  32249,  32284,  32318,  32350,  32382,
  32412,  32441,  32468,  32495,  32520,  32544,  32567,  32588,
  32609,  32628,  32646,  32662,  32678,  32692,  32705,  32717,
  32727,  32736,  32744,  32751,  32757,  32761,  32764,  32766,
  32767
}

/*--STATES-------------------------------------------------------------------*/

/* Load and store buffers used by several SIMD operations (both DIT and
 * butterfly). Both load and store buffers are needed because of the specific
 * way DIT is implemented and also in order to allow the butterfly operations
 * to be (software) pipelined. Four buffers (of each type) are necessary
 * because the main butterfly TIE operation performs eight butterfly
 * computations in parallel.
 */

state load_buf1 128
state load_buf2 128
state load_buf3 128
state load_buf4 128

state store_buf1 128
state store_buf2 128
state store_buf3 128
state store_buf4 128

/*--FUNCTIONS----------------------------------------------------------------*/

/* Computes the absolute value of a 16 bit two's complement number. */
function [15:0] abs([15:0] x)
{
  assign abs = TIEmux(x[15], x, TIEadd(0, ~x, 1'b1));
}

/* Performs a 16 bit arithmetic right shift. */
function [15:0] sra1([15:0] x)
{
  assign sra1 = {x[15], x[15:1]};
}

/* Performs a 16 bit fixed point multiplication. */
function [15:0] fix_mpy([15:0] a, [15:0] b)
{
  wire [31:0] prod = TIEmul(a, b, 1'b1);
  assign fix_mpy = prod >> 15;
}

/* Auxilliary function for accessing the hardware sine table, helps reduce that
 * table's size by three quarters by performind index inversion and/or result
 * negation depending on the index's two most significant bits.
 */
function [15:0] sin([9:0] index)
{
  wire [8:0] index_mirrored = TIEadd(512, ~index[7:0], 1'b1);
  wire [8:0] base_index = TIEmux(index[8], index[7:0], index_mirrored);

  wire [15:0] base_val = SINE_WAVE[base_index];
  wire [15:0] base_val_negated = TIEadd(0, ~base_val, 1'b1);

  assign sin = TIEmux(index[9], base_val, base_val_negated);
}

/* Computes twiddle factors. */
```

```
function [31:0] twiddles([9:0] m, [3:0] k)
{
  wire [9:0] ti_index = m << k;
  wire [9:0] tr_index = TIEadd(ti_index, 256, 1'b0);

  assign twiddles = {sin(tr_index), sin(ti_index)};
}

/* Performs a single butterfly operation, ar/ai and br/bi are real and
 * imaginary parts of the two input operands and tr/ti are the respective
 * twiddle factors.
 */
function [63:0] bfly([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
                     [15:0] tr, [15:0] ti, inv, shift)
{
  wire [15:0] twiddle_r = TIEmux(shift, tr, sra1(tr));
  wire [15:0] tmp = TIEmux(inv, TIEadd(16'h0, ~ti, 1'b1), ti);
  wire [15:0] twiddle_i = TIEmux(shift, tmp, sra1(tmp));

  wire [15:0] tmp_r = TIEadd(fix_mpy(twiddle_r, br), ~fix_mpy(twiddle_i, bi), 1'b1);
  wire [15:0] tmp_i = TIEadd(fix_mpy(twiddle_r, bi), fix_mpy(twiddle_i, br), 1'b0);

  wire [15:0] ar_out = TIEadd(TIEmux(shift, ar, sra1(ar)), tmp_r, 1'b0);
  wire [15:0] ai_out = TIEadd(TIEmux(shift, ai, sra1(ai)), tmp_i, 1'b0);
  wire [15:0] br_out= TIEadd(TIEmux(shift, ar, sra1(ar)), ~tmp_r, 1'b1);
  wire [15:0] bi_out = TIEadd(TIEmux(shift, ai, sra1(ai)), ~tmp_i, 1'b1);

  assign bfly = {ar_out, ai_out, br_out, bi_out};
}

/* Same as above except for DIF approach. */
function [63:0] bfly_dif([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
                         [15:0] tr, [15:0] ti, inv, shift)
{
  wire [15:0] twiddle_r = TIEmux(shift, tr, sra1(tr));
  wire [15:0] twiddle_i_tmp = TIEmux(inv, TIEadd(16'h0, ~ti, 1'b1), ti);
  wire [15:0] twiddle_i = TIEmux(shift, twiddle_i_tmp, sra1(twiddle_i_tmp));

  wire [15:0] ar_out_tmp = TIEadd(ar, br, 1'b0);
  wire [15:0] ai_out_tmp = TIEadd(ai, bi, 1'b0);
  wire [15:0] ar_out = TIEmux(shift, ar_out_tmp, sra1(ar_out_tmp));
  wire [15:0] ai_out = TIEmux(shift, ai_out_tmp, sra1(ai_out_tmp));

  wire [15:0] tmp_r = TIEadd(ar, ~br, 1'b1);
  wire [15:0] tmp_i = TIEadd(bi, ~ai, 1'b1);
  wire [15:0] br_out= TIEadd(fix_mpy(twiddle_r, tmp_r), fix_mpy(twiddle_i, tmp_i), 1'b0);
  wire [15:0] bi_out = TIEadd(fix_mpy(twiddle_i, tmp_r), ~fix_mpy(twiddle_r, tmp_i), 1'b1);

  assign bfly_dif = {ar_out, ai_out, br_out, bi_out};
}

/* Four wrapper functions around "bfly" declared as shared so that the
 * resulting hardware units can be shared between the three unrolled butterfly
 * loop stages as well as the main butterfly operation. The remaining four
 * butterfly units needed by the main butterfly loop are not shared and can
 * thus the bare "bfly" function can be used to implement them.
 */

function [63:0] bfly1([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
                      [15:0] tr, [15:0] ti, inv, shift) shared
{
    assign bfly1 = bfly(ar, ai, br, bi, tr, ti, inv, shift);
}
function [63:0] bfly2([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
                      [15:0] tr, [15:0] ti, inv, shift) shared
{
    assign bfly2 = bfly(ar, ai, br, bi, tr, ti, inv, shift);
}
function [63:0] bfly3([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
                      [15:0] tr, [15:0] ti, inv, shift) shared
{
    assign bfly3 = bfly(ar, ai, br, bi, tr, ti, inv, shift);
}
```

```
function [63:0] bfly4([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
                      [15:0] tr, [15:0] ti, inv, shift) shared
{
    assign bfly4 = bfly(ar, ai, br, bi, tr, ti, inv, shift);
}

/* Same as above excecpt for DIF approach. */

function [63:0] bfly_dif1([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
                      [15:0] tr, [15:0] ti, inv, shift) shared
{
    assign bfly_dif1 = bfly_dif(ar, ai, br, bi, tr, ti, inv, shift);
}
function [63:0] bfly_dif2([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
                      [15:0] tr, [15:0] ti, inv, shift) shared
{
    assign bfly_dif2 = bfly_dif(ar, ai, br, bi, tr, ti, inv, shift);
}
function [63:0] bfly_dif3([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
                      [15:0] tr, [15:0] ti, inv, shift) shared
{
    assign bfly_dif3 = bfly_dif(ar, ai, br, bi, tr, ti, inv, shift);
}
function [63:0] bfly_dif4([15:0] ar, [15:0] ai, [15:0] br, [15:0] bi,
                      [15:0] tr, [15:0] ti, inv, shift) shared
{
    assign bfly_dif4 = bfly_dif(ar, ai, br, bi, tr, ti, inv, shift);
}

/*--FLIX SLOTS--------------------------------------------------------------*/

format flix_dit 64 {flix_dit_slot0, flix_dit_slot1, flix_dit_slot2}
slot_opcodes flix_dit_slot0 {MOVI.N, MOV.N, ADD.N, SRL, INC4,
                             LOAD1, STORE1, OFFS_LOAD3, OFFS_STORE3,
                             DSWAP11, DSWAP21}
slot_opcodes flix_dit_slot1 {MOVI.N, MOV.N, ADDI.N, ADD.N,
                             LOAD2, STORE2, OFFS_LOAD4, OFFS_STORE4,
                             DSWAP12, DSWAP22}
slot_opcodes flix_dit_slot2 {NOP, MOVI.N, ADDI, ADD.N, ANDB, BNE, BEQI, J,
                             DINC, SCALING,
                             BFLY_DIT_UNROLL1, BFLY_DIT_UNROLL2,
                             BFLY_DIT_UNROLL4, BFLY_DIT_MAIN,
                             BFLY_DIF_UNROLL1, BFLY_DIF_UNROLL2,
                             BFLY_DIF_UNROLL4, BFLY_DIF_MAIN}

/*--OPERATIONS--------------------------------------------------------------*/

/* Add a value to four registers in parallel, used by the main butterly
 * operation.
 */
operation INC4
{inout AR r1, inout AR r2, inout AR r3, inout AR r4, in AR plus}
{}
{
  assign r1 = TIEadd(r1, plus, 1'b0);
  assign r2 = TIEadd(r2, plus, 1'b0);
  assign r3 = TIEadd(r3, plus, 1'b0);
  assign r4 = TIEadd(r4, plus, 1'b0);
}

/*--SIMD LOADS AND STORES---------------------------------------------------*/

/* Load from memory to the first 128 bit load buffer and increment the pointer
 * used accordingly. This operation is used both during DIT and all butterfly
 * stages.
 */
operation LOAD1
{inout AR *data_ptr}
{out VAddr, in MemDataIn128, out load_buf1}
{
  assign VAddr = data_ptr;
  assign load_buf1 = MemDataIn128;

  assign data_ptr = TIEadd(data_ptr, 16, 1'b0);
}

/* Same as above except that the result is loaded into the second load
 * buffer.
 */
operation LOAD2
{inout AR *data_ptr}
{out VAddr, in MemDataIn128, out load_buf2}
{
  assign VAddr = data_ptr;
  assign load_buf2 = MemDataIn128;

  assign data_ptr = TIEadd(data_ptr, 16, 1'b0);
}

/* Load from memory to the third load buffer. The memory location
 * is determined from a pointer plus an offset (passed in a second
 * register). This load operation is used together with the two
 * previous ones in the butterfly main loop.
 */
operation OFFS_LOAD3
{in AR *load_ptr, in AR offs}
{in MemDataIn128, out VAddr, out load_buf3}
{
  assign VAddr = TIEadd(load_ptr, offs, 1'b0);
  assign load_buf3 = MemDataIn128;
}

/* Same as above except that the result is loaded into the fourth load
 * buffer.
 */
operation OFFS_LOAD4
{in AR *load_ptr, in AR offs}
{in MemDataIn128, out VAddr, out load_buf4}
{
  assign VAddr = TIEadd(load_ptr, offs, 1'b0);
  assign load_buf4 = MemDataIn128;
}

/* The following store operations complement the load operations above. */

operation STORE1
{inout AR *data_ptr}
{out VAddr, out MemDataOut128, in store_buf1}
{
  assign VAddr = data_ptr;
  assign MemDataOut128 = store_buf1;
  assign data_ptr = TIEadd(data_ptr, 16, 1'b0);
}

operation STORE2
{inout AR *data_ptr}
{out VAddr, out MemDataOut128, in store_buf2}
{
  assign VAddr = data_ptr;
  assign MemDataOut128 = store_buf2;
  assign data_ptr = TIEadd(data_ptr, 16, 1'b0);
}

operation OFFS_STORE3
{in AR *store_ptr, in AR offs}
{out MemDataOut128, out VAddr, in store_buf3}
{
  assign VAddr = TIEadd(store_ptr, offs, 1'b0);
  assign MemDataOut128 = store_buf3;
}

operation OFFS_STORE4
{in AR *store_ptr, in AR offs}
{out MemDataOut128, out VAddr, in store_buf4}
{
  assign VAddr = TIEadd(store_ptr, offs, 1'b0);
  assign MemDataOut128 = store_buf4;
}

/*--DECIMATION IN TIME/FREQUENCY--------------------------------------------*/

/* Shift a single 16 bit value from a memory location given by the
 * forward/reverse DIT loop counters into the first store buffer.
 */
operation DSWAP11
{in AR *data_ptr, in AR dit_ctr, in AR dit_rev_ctr}
{out VAddr, in MemDataIn16, inout store_buf1}
{
  wire [15:0] ctr_diff = TIEadd(dit_rev_ctr, ~dit_ctr, 1'b1);
  wire [15:0] offs = TIEmux(ctr_diff[15], dit_rev_ctr, dit_ctr);

  assign VAddr = TIEadd(data_ptr, offs << 1, 1'b0);

  wire [111:0] tmp = store_buf1 >> 16;
  assign store_buf1 = {MemDataIn16, tmp};
}

/* Same as above except for the second store buffer. */
operation DSWAP12
{in AR *data_ptr, in AR dit_ctr, in AR dit_rev_ctr}
{out VAddr, in MemDataIn16, inout store_buf2}
{
  wire [15:0] ctr_diff = TIEadd(dit_rev_ctr, ~dit_ctr, 1'b1);
  wire [15:0] offs = TIEmux(ctr_diff[15], dit_rev_ctr, dit_ctr);

  assign VAddr = TIEadd(data_ptr, offs << 1, 1'b0);

  wire [111:0] tmp = store_buf2 >> 16;
  assign store_buf2 = {MemDataIn16, tmp};
}

/* Store a single 16 bit value from the first load buffer to memory and then
 * shift the buffer to the right by the same amount. */
operation DSWAP21
{in AR *data_ptr, in AR dit_ctr, in AR dit_rev_ctr}
{out VAddr, out MemDataOut16, inout load_buf1}
{
  wire [15:0] ctr_diff = TIEadd(dit_rev_ctr, ~dit_ctr, 1'b1);
  wire [15:0] offs = TIEmux(ctr_diff[15], dit_rev_ctr, dit_ctr);

  assign VAddr = TIEadd(data_ptr, offs << 1, 1'b0);

  assign MemDataOut16 = load_buf1[15:0];
  assign load_buf1 = load_buf1 >> 16;
}

/* Same as above except for the second load buffer. */
operation DSWAP22
{in AR *data_ptr, in AR dit_ctr, in AR dit_rev_ctr}
{out VAddr, out MemDataOut16, inout load_buf2}
{
  wire [15:0] ctr_diff = TIEadd(dit_rev_ctr, ~dit_ctr, 1'b1);
  wire [15:0] offs = TIEmux(ctr_diff[15], dit_rev_ctr, dit_ctr);

  assign VAddr = TIEadd(data_ptr, offs << 1, 1'b0);

  assign MemDataOut16 = load_buf2[15:0];
  assign load_buf2 = load_buf2 >> 16;
}

/* Increment both forward and reverse DIT loop counters. The reverse counter
 * is incremented using a custom reverse ripple carry adder, this makes
 * expensive hardware counter inversion (which depends on N!) in every loop
 * unnecessary. It might also be a good idea to use an adder architecture with
 * a shorter critical path but whether this would have a significant
 * performance impact is hard to judge.
 */
operation DINC {inout AR dit_ctr, inout AR dit_rev_ctr, in AR n} {}
{
  wire c1, c2, c3, c4, c5, c6, c7, c8, c9;
  wire s0, s1, s2, s3, s4, s5, s6, s7, s8, s9;
  wire [9:0] mask = n >> 1;

  assign {c1, s9} = TIEcsa(dit_rev_ctr[9], mask[9], 1'b0);
  assign {c2, s8} = TIEcsa(dit_rev_ctr[8], mask[8], c1);
  assign {c3, s7} = TIEcsa(dit_rev_ctr[7], mask[7], c2);
  assign {c4, s6} = TIEcsa(dit_rev_ctr[6], mask[6], c3);
  assign {c5, s5} = TIEcsa(dit_rev_ctr[5], mask[5], c4);
  assign {c6, s4} = TIEcsa(dit_rev_ctr[4], mask[4], c5);
  assign {c7, s3} = TIEcsa(dit_rev_ctr[3], mask[3], c6);
  assign {c8, s2} = TIEcsa(dit_rev_ctr[2], mask[2], c7);
  assign {c9, s1} = TIEcsa(dit_rev_ctr[1], mask[1], c8);
  assign s0       = TIEadd(dit_rev_ctr[0], mask[0], c9);

  assign dit_ctr = TIEadd(dit_ctr, 1'b1, 1'b0);
  assign dit_rev_ctr = {s9, s8, s7, s6, s5, s4, s3, s2, s1, s0};
}

/*--SCALING---------------------------------------------------------------*/

/* Check if "shift" should be set to one by looking at four complex input
 * numbers at a time.
 */
operation SCALING
{inout BR shift, inout AR i, in AR n}
{in load_buf1, in load_buf2}
{
  wire [15:0] abs_real1 = abs(load_buf1[15:0]);
  wire [15:0] abs_real2 = abs(load_buf1[31:16]);
  wire [15:0] abs_real3 = abs(load_buf1[47:32]);
  wire [15:0] abs_real4 = abs(load_buf1[63:48]);
  wire [15:0] abs_real5 = abs(load_buf1[79:64]);
  wire [15:0] abs_real6 = abs(load_buf1[95:80]);
  wire [15:0] abs_real7 = abs(load_buf1[111:96]);
  wire [15:0] abs_real8 = abs(load_buf1[127:112]);

  wire [15:0] abs_imag1 = abs(load_buf2[15:0]);
  wire [15:0] abs_imag2 = abs(load_buf2[31:16]);
  wire [15:0] abs_imag3 = abs(load_buf2[47:32]);
  wire [15:0] abs_imag4 = abs(load_buf2[63:48]);
  wire [15:0] abs_imag5 = abs(load_buf2[79:64]);
  wire [15:0] abs_imag6 = abs(load_buf2[95:80]);
  wire [15:0] abs_imag7 = abs(load_buf2[111:96]);
  wire [15:0] abs_imag8 = abs(load_buf2[127:112]);

  wire tmp = shift |
             abs_real1[14] | abs_real2[14] |
             abs_real3[14] | abs_real4[14] |
             abs_real5[14] | abs_real6[14] |
             abs_real7[14] | abs_real8[14] |
             abs_imag1[14] | abs_imag2[14] |
             abs_imag3[14] | abs_imag4[14] |
             abs_imag5[14] | abs_imag6[14] |
             abs_imag7[14] | abs_imag8[14];

  assign shift = tmp;
  assign i = TIEmux(tmp, TIEadd(i, 8, 1'b0), n);
}

/*--BUTTERFLY-------------------------------------------------------------*/

/* Butterfly operation used in the first unrolled butterfly loop iteration.
 * Performs four butterfly operations in parallel.
 */
operation BFLY_DIT_UNROLL1
{in BR inv, in BR shift}
{out store_buf1, out store_buf2, in load_buf1, in load_buf2}
{
  wire [15:0] tr = 16'h7fff;
  wire [15:0] ti = 16'h0000;

  wire [15:0] a0r = load_buf1[15:0];
  wire [15:0] a0i = load_buf2[15:0];
  wire [15:0] b0r = load_buf1[31:16];
  wire [15:0] b0i = load_buf2[31:16];

  wire [15:0] a1r = load_buf1[47:32];
  wire [15:0] a1i = load_buf2[47:32];
  wire [15:0] b1r = load_buf1[63:48];
  wire [15:0] b1i = load_buf2[63:48];

  wire [15:0] a2r = load_buf1[79:64];
  wire [15:0] a2i = load_buf2[79:64];
  wire [15:0] b2r = load_buf1[95:80];
  wire [15:0] b2i = load_buf2[95:80];

  wire [15:0] a3r = load_buf1[111:96];
  wire [15:0] a3i = load_buf2[111:96];
  wire [15:0] b3r = load_buf1[127:112];
  wire [15:0] b3i = load_buf2[127:112];

  wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;

  assign {c0r, c0i, d0r, d0i} = bfly1(a0r, a0i, b0r, b0i, tr, ti, inv, shift);
  assign {c1r, c1i, d1r, d1i} = bfly2(a1r, a1i, b1r, b1i, tr, ti, inv, shift);
  assign {c2r, c2i, d2r, d2i} = bfly3(a2r, a2i, b2r, b2i, tr, ti, inv, shift);
  assign {c3r, c3i, d3r, d3i} = bfly4(a3r, a3i, b3r, b3i, tr, ti, inv, shift);

  assign store_buf1 = {d3r, c3r, d2r, c2r, d1r, c1r, d0r, c0r};
  assign store_buf2 = {d3i, c3i, d2i, c2i, d1i, c1i, d0i, c0i};
}

/* Butterfly operation used in the second unrolled butterfly loop iteration.
 * Performs four butterfly operations in parallel.
 */
operation BFLY_DIT_UNROLL2
{in BR inv, in BR shift}
{out store_buf1, out store_buf2, in load_buf1, in load_buf2}
{
  wire [15:0] t1 = 16'h7fff;
  wire [15:0] t2 = 16'h0000;

  wire [15:0] a0r = load_buf1[15:0];
  wire [15:0] a0i = load_buf2[15:0];
  wire [15:0] b0r = load_buf1[47:32];
  wire [15:0] b0i = load_buf2[47:32];

  wire [15:0] a1r = load_buf1[31:16];
  wire [15:0] a1i = load_buf2[31:16];
  wire [15:0] b1r = load_buf1[63:48];
  wire [15:0] b1i = load_buf2[63:48];

  wire [15:0] a2r = load_buf1[79:64];
  wire [15:0] a2i = load_buf2[79:64];
  wire [15:0] b2r = load_buf1[111:96];
  wire [15:0] b2i = load_buf2[111:96];

  wire [15:0] a3r = load_buf1[95:80];
  wire [15:0] a3i = load_buf2[95:80];
  wire [15:0] b3r = load_buf1[127:112];
  wire [15:0] b3i = load_buf2[127:112];

  wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;

  assign {c0r, c0i, d0r, d0i} = bfly1(a0r, a0i, b0r, b0i, t1, t2, inv, shift);
  assign {c1r, c1i, d1r, d1i} = bfly2(a1r, a1i, b1r, b1i, t2, t1, inv, shift);
  assign {c2r, c2i, d2r, d2i} = bfly3(a2r, a2i, b2r, b2i, t1, t2, inv, shift);
  assign {c3r, c3i, d3r, d3i} = bfly4(a3r, a3i, b3r, b3i, t2, t1, inv, shift);

  assign store_buf1 = {d3r, d2r, c3r, c2r, d1r, d0r, c1r, c0r};
  assign store_buf2 = {d3i, d2i, c3i, c2i, d1i, d0i, c1i, c0i};
}

/* Butterfly operation used in the fourth unrolled butterfly loop iteration.
 * Performs four butterfly operations in parallel.
 */
operation BFLY_DIT_UNROLL4
{in BR inv, in BR shift}
{out store_buf1, out store_buf2, in load_buf1, in load_buf2}
{
  wire [15:0] t0r = 16'h7fff;
  wire [15:0] t0i = 16'h0000;
  wire [15:0] t1r = 16'h5a81;
  wire [15:0] t1i = 16'h5a81;
  wire [15:0] t2r = 16'h0000;
  wire [15:0] t2i = 16'h7fff;
  wire [15:0] t3r = 16'ha57f;
  wire [15:0] t3i = 16'h5a81;

  wire [15:0] a0r = load_buf1[15:0];
  wire [15:0] a0i = load_buf2[15:0];
  wire [15:0] b0r = load_buf1[79:64];
  wire [15:0] b0i = load_buf2[79:64];

  wire [15:0] a1r = load_buf1[31:16];
  wire [15:0] a1i = load_buf2[31:16];
  wire [15:0] b1r = load_buf1[95:80];
  wire [15:0] b1i = load_buf2[95:80];

  wire [15:0] a2r = load_buf1[47:32];
  wire [15:0] a2i = load_buf2[47:32];
  wire [15:0] b2r = load_buf1[111:96];
  wire [15:0] b2i = load_buf2[111:96];

  wire [15:0] a3r = load_buf1[63:48];
  wire [15:0] a3i = load_buf2[63:48];
  wire [15:0] b3r = load_buf1[127:112];
  wire [15:0] b3i = load_buf2[127:112];

  wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;

  assign {c0r, c0i, d0r, d0i} = bfly1(a0r, a0i, b0r, b0i, t0r, t0i, inv, shift);
  assign {c1r, c1i, d1r, d1i} = bfly2(a1r, a1i, b1r, b1i, t1r, t1i, inv, shift);
  assign {c2r, c2i, d2r, d2i} = bfly3(a2r, a2i, b2r, b2i, t2r, t2i, inv, shift);
  assign {c3r, c3i, d3r, d3i} = bfly4(a3r, a3i, b3r, b3i, t3r, t3i, inv, shift);

  assign store_buf1 = {d3r, d2r, d1r, d0r, c3r, c2r, c1r, c0r};
  assign store_buf2 = {d3i, d2i, d1i, d0i, c3i, c2i, c1i, c0i};
}

/* Butterfly operation used in the main butterfly loop. Perfoms eight butterfly
 * operations in parallel.
 */
operation BFLY_DIT_MAIN {inout AR m1, in AR k, in BR inv, in BR shift}
{out store_buf1, out store_buf2,
 out store_buf3, out store_buf4,
 in load_buf1, in load_buf2,
 in load_buf3, in load_buf4}
{
  wire [15:0] bf1_tr, bf1_ti, bf2_tr, bf2_ti,
              bf3_tr, bf3_ti, bf4_tr, bf4_ti,
              bf5_tr, bf5_ti, bf6_tr, bf6_ti,
              bf7_tr, bf7_ti, bf8_tr, bf8_ti;

  wire [9:0] m2 = TIEadd(m1, 1, 1'b0);
  wire [9:0] m3 = TIEadd(m1, 2, 1'b0);
  wire [9:0] m4 = TIEadd(m1, 3, 1'b0);
  wire [9:0] m5 = TIEadd(m1, 4, 1'b0);
  wire [9:0] m6 = TIEadd(m1, 5, 1'b0);
  wire [9:0] m7 = TIEadd(m1, 6, 1'b0);
  wire [9:0] m8 = TIEadd(m1, 7, 1'b0);

  assign {bf1_tr, bf1_ti} = twiddles(m1, k);
  assign {bf2_tr, bf2_ti} = twiddles(m2, k);
  assign {bf3_tr, bf3_ti} = twiddles(m3, k);
  assign {bf4_tr, bf4_ti} = twiddles(m4, k);
  assign {bf5_tr, bf5_ti} = twiddles(m5, k);
  assign {bf6_tr, bf6_ti} = twiddles(m6, k);
  assign {bf7_tr, bf7_ti} = twiddles(m7, k);
  assign {bf8_tr, bf8_ti} = twiddles(m8, k);

  wire [15:0] bf1_r1, bf1_i1, bf1_r2, bf1_i2,
              bf2_r1, bf2_i1, bf2_r2, bf2_i2,
              bf3_r1, bf3_i1, bf3_r2, bf3_i2,
              bf4_r1, bf4_i1, bf4_r2, bf4_i2,
              bf5_r1, bf5_i1, bf5_r2, bf5_i2,
              bf6_r1, bf6_i1, bf6_r2, bf6_i2,
              bf7_r1, bf7_i1, bf7_r2, bf7_i2,
              bf8_r1, bf8_i1, bf8_r2, bf8_i2;

  assign {bf1_r1, bf1_i1, bf1_r2, bf1_i2} =
    bfly1(load_buf1[15:0], load_buf2[15:0], load_buf3[15:0], load_buf4[15:0],
          bf1_tr, bf1_ti, inv, shift);
  assign {bf2_r1, bf2_i1, bf2_r2, bf2_i2} =
    bfly2(load_buf1[31:16], load_buf2[31:16], load_buf3[31:16], load_buf4[31:16],
          bf2_tr, bf2_ti, inv, shift);
  assign {bf3_r1, bf3_i1, bf3_r2, bf3_i2} =
    bfly3(load_buf1[47:32], load_buf2[47:32], load_buf3[47:32], load_buf4[47:32],
          bf3_tr, bf3_ti, inv, shift);
  assign {bf4_r1, bf4_i1, bf4_r2, bf4_i2} =
    bfly4(load_buf1[63:48], load_buf2[63:48], load_buf3[63:48], load_buf4[63:48],
          bf4_tr, bf4_ti, inv, shift);
  assign {bf5_r1, bf5_i1, bf5_r2, bf5_i2} =
    bfly(load_buf1[79:64], load_buf2[79:64], load_buf3[79:64], load_buf4[79:64],
         bf5_tr, bf5_ti, inv, shift);
  assign {bf6_r1, bf6_i1, bf6_r2, bf6_i2} =
    bfly(load_buf1[95:80], load_buf2[95:80], load_buf3[95:80], load_buf4[95:80],
         bf6_tr, bf6_ti, inv, shift);
  assign {bf7_r1, bf7_i1, bf7_r2, bf7_i2} =
    bfly(load_buf1[111:96], load_buf2[111:96], load_buf3[111:96], load_buf4[111:96],
         bf7_tr, bf7_ti, inv, shift);
  assign {bf8_r1, bf8_i1, bf8_r2, bf8_i2} =
    bfly(load_buf1[127:112], load_buf2[127:112], load_buf3[127:112], load_buf4[127:112],
         bf8_tr, bf8_ti, inv, shift);

  assign store_buf1 = {bf8_r1, bf7_r1, bf6_r1, bf5_r1, bf4_r1, bf3_r1, bf2_r1, bf1_r1};
  assign store_buf2 = {bf8_i1, bf7_i1, bf6_i1, bf5_i1, bf4_i1, bf3_i1, bf2_i1, bf1_i1};
  assign store_buf3 = {bf8_r2, bf7_r2, bf6_r2, bf5_r2, bf4_r2, bf3_r2, bf2_r2, bf1_r2};
  assign store_buf4 = {bf8_i2, bf7_i2, bf6_i2, bf5_i2, bf4_i2, bf3_i2, bf2_i2, bf1_i2};

  assign m1 = TIEadd(m1, 8, 1'b0);
}

schedule bfly_dit_main_sched {BFLY_DIT_MAIN} {
  def store_buf1 2;
  def store_buf2 2;
  def store_buf3 2;
  def store_buf4 2;
}

/*--BUTTERFLY (DIF)----------------------------------------------------------*/

/* Butterfly operation used in the unrolled butterfly loop iteration m.
 * Performs four butterfly operations in parallel.
 */
operation BFLY_DIF_UNROLL1
{in BR inv, in BR shift}
{out store_buf1, out store_buf2, in load_buf1, in load_buf2}
{
  wire [15:0] tr = 16'h7fff;
  wire [15:0] ti = 16'h0000;

  wire [15:0] a0r = load_buf1[15:0];
  wire [15:0] a0i = load_buf2[15:0];
  wire [15:0] b0r = load_buf1[31:16];
  wire [15:0] b0i = load_buf2[31:16];

  wire [15:0] a1r = load_buf1[47:32];
  wire [15:0] a1i = load_buf2[47:32];
  wire [15:0] b1r = load_buf1[63:48];
  wire [15:0] b1i = load_buf2[63:48];

  wire [15:0] a2r = load_buf1[79:64];
  wire [15:0] a2i = load_buf2[79:64];
  wire [15:0] b2r = load_buf1[95:80];
  wire [15:0] b2i = load_buf2[95:80];

  wire [15:0] a3r = load_buf1[111:96];
  wire [15:0] a3i = load_buf2[111:96];
  wire [15:0] b3r = load_buf1[127:112];
  wire [15:0] b3i = load_buf2[127:112];

  wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;

  assign {c0r, c0i, d0r, d0i} = bfly_dif1(a0r, a0i, b0r, b0i, tr, ti, inv, shift);
  assign {c1r, c1i, d1r, d1i} = bfly_dif2(a1r, a1i, b1r, b1i, tr, ti, inv, shift);
  assign {c2r, c2i, d2r, d2i} = bfly_dif3(a2r, a2i, b2r, b2i, tr, ti, inv, shift);
  assign {c3r, c3i, d3r, d3i} = bfly_dif4(a3r, a3i, b3r, b3i, tr, ti, inv, shift);

  assign store_buf1 = {d3r, c3r, d2r, c2r, d1r, c1r, d0r, c0r};
  assign store_buf2 = {d3i, c3i, d2i, c2i, d1i, c1i, d0i, c0i};
}

/* Butterfly operation used in the unrolled butterfly loop iteration m - 1.
 * Performs four butterfly operations in parallel.
 */
operation BFLY_DIF_UNROLL2
{in BR inv, in BR shift}
{out store_buf1, out store_buf2, in load_buf1, in load_buf2}
{
  wire [15:0] t1 = 16'h7fff;
  wire [15:0] t2 = 16'h0000;

  wire [15:0] a0r = load_buf1[15:0];
  wire [15:0] a0i = load_buf2[15:0];
  wire [15:0] b0r = load_buf1[47:32];
  wire [15:0] b0i = load_buf2[47:32];

  wire [15:0] a1r = load_buf1[31:16];
  wire [15:0] a1i = load_buf2[31:16];
  wire [15:0] b1r = load_buf1[63:48];
  wire [15:0] b1i = load_buf2[63:48];

  wire [15:0] a2r = load_buf1[79:64];
  wire [15:0] a2i = load_buf2[79:64];
  wire [15:0] b2r = load_buf1[111:96];
  wire [15:0] b2i = load_buf2[111:96];

  wire [15:0] a3r = load_buf1[95:80];
  wire [15:0] a3i = load_buf2[95:80];
  wire [15:0] b3r = load_buf1[127:112];
  wire [15:0] b3i = load_buf2[127:112];

  wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;

  assign {c0r, c0i, d0r, d0i} = bfly_dif1(a0r, a0i, b0r, b0i, t1, t2, inv, shift);
  assign {c1r, c1i, d1r, d1i} = bfly_dif2(a1r, a1i, b1r, b1i, t2, t1, inv, shift);
  assign {c2r, c2i, d2r, d2i} = bfly_dif3(a2r, a2i, b2r, b2i, t1, t2, inv, shift);
  assign {c3r, c3i, d3r, d3i} = bfly_dif4(a3r, a3i, b3r, b3i, t2, t1, inv, shift);

  assign store_buf1 = {d3r, d2r, c3r, c2r, d1r, d0r, c1r, c0r};
  assign store_buf2 = {d3i, d2i, c3i, c2i, d1i, d0i, c1i, c0i};
}

/* Butterfly operation used in the unrolled butterfly loop iteration m - 2.
 * Performs four butterfly operations in parallel.
 */
operation BFLY_DIF_UNROLL4
{in BR inv, in BR shift}
{out store_buf1, out store_buf2, in load_buf1, in load_buf2}
{
  wire [15:0] t0r = 16'h7fff;
  wire [15:0] t0i = 16'h0000;
  wire [15:0] t1r = 16'h5a81;
  wire [15:0] t1i = 16'h5a81;
  wire [15:0] t2r = 16'h0000;
  wire [15:0] t2i = 16'h7fff;
  wire [15:0] t3r = 16'ha57f;
  wire [15:0] t3i = 16'h5a81;

  wire [15:0] a0r = load_buf1[15:0];
  wire [15:0] a0i = load_buf2[15:0];
  wire [15:0] b0r = load_buf1[79:64];
  wire [15:0] b0i = load_buf2[79:64];

  wire [15:0] a1r = load_buf1[31:16];
  wire [15:0] a1i = load_buf2[31:16];
  wire [15:0] b1r = load_buf1[95:80];
  wire [15:0] b1i = load_buf2[95:80];

  wire [15:0] a2r = load_buf1[47:32];
  wire [15:0] a2i = load_buf2[47:32];
  wire [15:0] b2r = load_buf1[111:96];
  wire [15:0] b2i = load_buf2[111:96];

  wire [15:0] a3r = load_buf1[63:48];
  wire [15:0] a3i = load_buf2[63:48];
  wire [15:0] b3r = load_buf1[127:112];
  wire [15:0] b3i = load_buf2[127:112];

  wire [15:0] c0r, c0i, d0r, d0i, c1r, c1i, d1r, d1i, c2r, c2i, d2r, d2i, c3r, c3i, d3r, d3i;

  assign {c0r, c0i, d0r, d0i} = bfly_dif1(a0r, a0i, b0r, b0i, t0r, t0i, inv, shift);
  assign {c1r, c1i, d1r, d1i} = bfly_dif2(a1r, a1i, b1r, b1i, t1r, t1i, inv, shift);
  assign {c2r, c2i, d2r, d2i} = bfly_dif3(a2r, a2i, b2r, b2i, t2r, t2i, inv, shift);
  assign {c3r, c3i, d3r, d3i} = bfly_dif4(a3r, a3i, b3r, b3i, t3r, t3i, inv, shift);

  assign store_buf1 = {d3r, d2r, d1r, d0r, c3r, c2r, c1r, c0r};
  assign store_buf2 = {d3i, d2i, d1i, d0i, c3i, c2i, c1i, c0i};
}

/* Butterfly operation used in the main butterfly loop. Perfoms eight butterfly
 * operations in parallel.
 */
operation BFLY_DIF_MAIN {inout AR m1, in AR k, in BR inv, in BR shift}
{out store_buf1, out store_buf2,
 out store_buf3, out store_buf4,
 in load_buf1, in load_buf2,
 in load_buf3, in load_buf4}
{
  wire [15:0] bf1_tr, bf1_ti, bf2_tr, bf2_ti,
              bf3_tr, bf3_ti, bf4_tr, bf4_ti,
              bf5_tr, bf5_ti, bf6_tr, bf6_ti,
              bf7_tr, bf7_ti, bf8_tr, bf8_ti;

  wire [9:0] m2 = TIEadd(m1, 1, 1'b0);
  wire [9:0] m3 = TIEadd(m1, 2, 1'b0);
  wire [9:0] m4 = TIEadd(m1, 3, 1'b0);
  wire [9:0] m5 = TIEadd(m1, 4, 1'b0);
  wire [9:0] m6 = TIEadd(m1, 5, 1'b0);
  wire [9:0] m7 = TIEadd(m1, 6, 1'b0);
  wire [9:0] m8 = TIEadd(m1, 7, 1'b0);

  assign {bf1_tr, bf1_ti} = twiddles(m1, k);
  assign {bf2_tr, bf2_ti} = twiddles(m2, k);
  assign {bf3_tr, bf3_ti} = twiddles(m3, k);
  assign {bf4_tr, bf4_ti} = twiddles(m4, k);
  assign {bf5_tr, bf5_ti} = twiddles(m5, k);
  assign {bf6_tr, bf6_ti} = twiddles(m6, k);
  assign {bf7_tr, bf7_ti} = twiddles(m7, k);
  assign {bf8_tr, bf8_ti} = twiddles(m8, k);

  wire [15:0] bf1_r1, bf1_i1, bf1_r2, bf1_i2,
              bf2_r1, bf2_i1, bf2_r2, bf2_i2,
              bf3_r1, bf3_i1, bf3_r2, bf3_i2,
              bf4_r1, bf4_i1, bf4_r2, bf4_i2,
              bf5_r1, bf5_i1, bf5_r2, bf5_i2,
             bf6_r1, bf6_i1, bf6_r2, bf6_i2,
             bf7_r1, bf7_i1, bf7_r2, bf7_i2,
             bf8_r1, bf8_i1, bf8_r2, bf8_i2;

  assign {bf1_r1, bf1_i1, bf1_r2, bf1_i2} =
    bfly_dif1(load_buf1[15:0], load_buf2[15:0], load_buf3[15:0], load_buf4[15:0],
             bf1_tr, bf1_ti, inv, shift);
  assign {bf2_r1, bf2_i1, bf2_r2, bf2_i2} =
    bfly_dif2(load_buf1[31:16], load_buf2[31:16], load_buf3[31:16], load_buf4[31:16],
             bf2_tr, bf2_ti, inv, shift);
  assign {bf3_r1, bf3_i1, bf3_r2, bf3_i2} =
    bfly_dif3(load_buf1[47:32], load_buf2[47:32], load_buf3[47:32], load_buf4[47:32],
             bf3_tr, bf3_ti, inv, shift);
  assign {bf4_r1, bf4_i1, bf4_r2, bf4_i2} =
    bfly_dif4(load_buf1[63:48], load_buf2[63:48], load_buf3[63:48], load_buf4[63:48],
             bf4_tr, bf4_ti, inv, shift);
  assign {bf5_r1, bf5_i1, bf5_r2, bf5_i2} =
    bfly_dif(load_buf1[79:64], load_buf2[79:64], load_buf3[79:64], load_buf4[79:64],
            bf5_tr, bf5_ti, inv, shift);
  assign {bf6_r1, bf6_i1, bf6_r2, bf6_i2} =
    bfly_dif(load_buf1[95:80], load_buf2[95:80], load_buf3[95:80], load_buf4[95:80],
            bf6_tr, bf6_ti, inv, shift);
  assign {bf7_r1, bf7_i1, bf7_r2, bf7_i2} =
    bfly_dif(load_buf1[111:96], load_buf2[111:96], load_buf3[111:96], load_buf4[111:96],
            bf7_tr, bf7_ti, inv, shift);
  assign {bf8_r1, bf8_i1, bf8_r2, bf8_i2} =
    bfly_dif(load_buf1[127:112], load_buf2[127:112], load_buf3[127:112], load_buf4[127:112],
            bf8_tr, bf8_ti, inv, shift);

  assign store_buf1 = {bf8_r1, bf7_r1, bf6_r1, bf5_r1, bf4_r1, bf3_r1, bf2_r1, bf1_r1};
  assign store_buf2 = {bf8_i1, bf7_i1, bf6_i1, bf5_i1, bf4_i1, bf3_i1, bf2_i1, bf1_i1};
  assign store_buf3 = {bf8_r2, bf7_r2, bf6_r2, bf5_r2, bf4_r2, bf3_r2, bf2_r2, bf1_r2};
  assign store_buf4 = {bf8_i2, bf7_i2, bf6_i2, bf5_i2, bf4_i2, bf3_i2, bf2_i2, bf1_i2};

  assign m1 = TIEadd(m1, 8, 1'b0);
}

schedule bfly_dif_main_sched {BFLY_DIF_MAIN} {
  def store_buf1 2;
  def store_buf2 2;
  def store_buf3 2;
  def store_buf4 2;
}
```