



Fast OFDM on Xtensa[®] Processors

Application Note

Tensilica, Inc.
3255-6 Scott Blvd.
Santa Clara, CA 95054
(408) 986-8000
Fax (408) 986-8919
www.tensilica.com



© 2007 Tensilica, Inc.

Printed in the United States of America

All Rights Reserved

This publication is provided "AS IS." Tensilica, Inc. (hereafter "Tensilica") does not make any warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Information in this document is provided solely to enable system and software developers to use Tensilica processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereunder to design or fabricate Tensilica integrated circuits or integrated circuits based on the information in this document. Tensilica does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

Tensilica is a registered trademark of Tensilica, Inc. The following terms are trademarks of Tensilica, Inc.: FLIX, OSKit, Sea of Processors, TurboXim, Vectra, Xenergy, Xplorer, and XPRES. All other trademarks and registered trademarks are the property of their respective companies.



Contents

1	Introduction	1
2	OFDM Engine	3
3	Implementing Fast Radix-4 FFT	4
4	Complex FIR Support	7
5	Results	8
6	Summary	9
	Appendix A – FFT Example C Source	11
	Appendix B – OFDM Engine TIE Source	17

Abstract

This application note looks briefly at fast signal processing for wireless modems. In particular, this application note describes the power of a configurable processor in handling the performance-intensive signal processing demands of supporting FFT transforms and FIR filter algorithms.

The Xtensa processor is extensible and can easily be customized to accelerate any algorithm. This application note describes TIE (Tensilica Instruction Extension) language instructions that accelerate the complex FFT and FIR operations that dominate many OFDM channel modulation and demodulation systems.

This application note is not meant as an OFDM primer and makes the assumption that the reader is already familiar with basic DSP algorithms such as FIR and FFT algorithms. In addition, this application note assumes that the reader is familiar with the Xtensa Instruction Set Architecture and the Tensilica Instruction Extension language. As an additional prerequisite, the application note *Convolution Coding on Xtensa Processors* by Tensilica, Inc. should be understood by the reader.

1 Introduction

An increasing number of emerging wireless communication standards are adopting the advanced modulation methods of Orthogonal Frequency Division Multiplexing (OFDM). OFDM plays a growing role in wired and wireless modem design. It has been adopted across a wide range of communications standards, including

- ◆ WiFi (802.11a/g)
- ◆ Digital Audio Broadcast (for example, HD Radio, T-DMB, ISDB-TSB, Digital Radio Mondiale)
- ◆ ADSL and VDSL
- ◆ Terrestrial DTV broadcast (DVB-T, DVB-H, T-DMB, ISDB-T, MediaFLO)
- ◆ WiMax (802.16)
- ◆ Flash-OFDM cellular
- ◆ WiMedia UWB
- ◆ MoCA home networking

OFDM is a multi-carrier modulation method, implementing a large number of closely-spaced, but orthogonal sub-carriers, where each sub-carrier is modulated using conventional schemes such as QAM or QPSK. The large number of sub-carriers mandates heavy use of large FFTs at the heart of the transmitter and receiver computation. OFDM reduces the demands on channel equalization and tuned sub-channel filters. It is naturally robust inter-symbol interference (ISI), against fading by multi-path propagation, and against narrow-band co-channel interference. It provides high spectral efficiency, though it is sensitive to Doppler shift, especially at high speeds, and to frequency synchronization problems.

OFDM is combined with forward error correction to form the PHY stage of the typical transmitter, as shown in Figure 1.

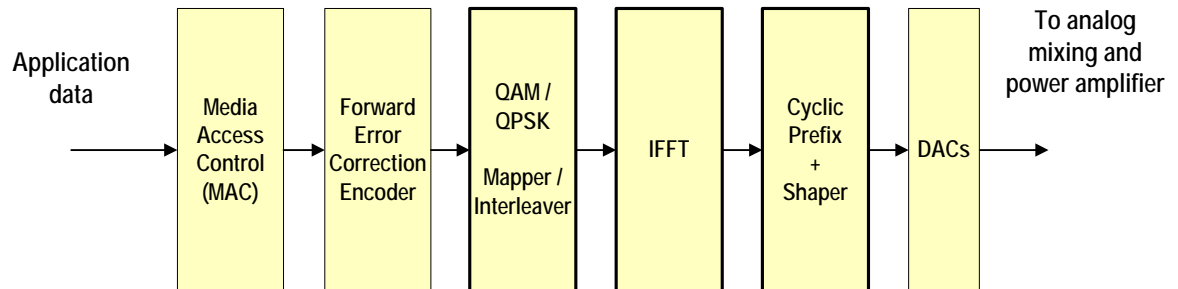


FIGURE 1 TYPICAL OFDM TRANSMITTER

and typical receiver, as shown in Figure 2. The highlighted blocks, especially the FFTs, make up the heart of OFDM computation.

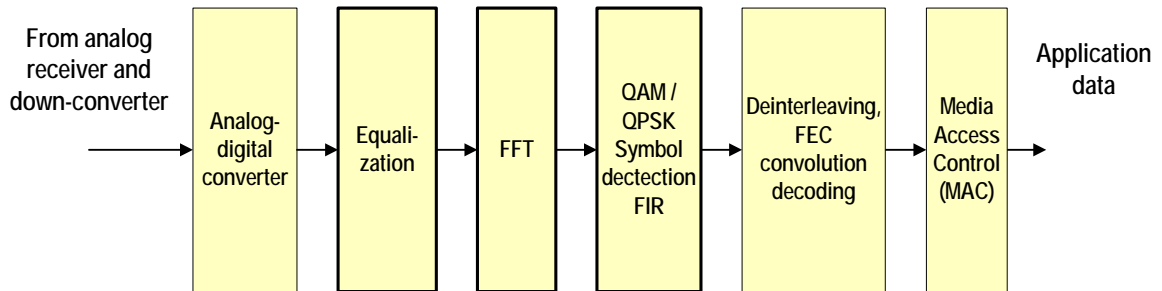


FIGURE 2 TYPICAL OFDM RECEIVER

This application note suggests an efficient yet programmable implementation approach for the OFDM section of the transmitter and receiver. The note concentrates on the demands of the FFT section, but the design also optionally implements high-performance FIR filter capability, which can be important in equalization, shaping, symbol mapping and symbol detection.

OFDM-based communication offers higher channel efficiency, but poses a daunting computational challenge. The complexity of the standards, the likelihood of continuous feature evolution, and the need to support several standards on the same hardware all provide strong motivation for a programmable solution. Each OFDM-based modem has somewhat different algorithmic needs, though all are built around FFT-intensive computation, plus a variety of FIR filters and similar computations. Each standard may support variants in the size of the FFT, the organization of data and interaction between core OFDM algorithms and other functions, such as error correction and channel adaptation mechanisms. The computational load of the OFDM within the PHY layer requires very high performance general-purpose DSPs (probably beyond the capacity of even power-hungry devices), hardwired, non-programmable DSP computation logic, or application-optimized DSPs.

The ideal application-optimized DSP would have the following characteristics:

- ◆ Delivered as a fully-programmable IP core for integration into the SOC design
- ◆ Include a complete tools environment, including compilers, debuggers and simulators to allow easy development using standard high-level languages, source-level debuggers, simulators and prototyping platforms
- ◆ Small area (including small code size), low power and high performance

In this note, we propose the architecture for a high-performance specialized DSP. The design was motivated by the challenge of running very high data-rate modulation/demodulation functions, where many of the details such as data organization, filter types and sizes, FFT types and sizes, are uncertain at the time of chip design, but rough requirements for the RTL hardware level of efficiency (e.g., area, power, and performance) are known. Variations of OFDM may include real-to-complex FFT, time-domain equalization (TEQ), frequency-domain scaling (FEQ) and mapping to QAM constellation. Depending on the exact requirements, the basic instruction set outlined here (or small variations thereof) can be used to implement these additional functions. Moreover, error correction functions, such as Viterbi and Turbo encoding and decoding, are also easily accelerated using Xtensa processor extensions but the extensions do not share as much hardware with the FIR/FFT features described here and may be implemented in a second Xtensa core optimized for error coding.

2 OFDM Engine

This design example uses Xtensa LX2 with dual 128b load/store and 32b Flexible Length Instructions eXtensions (FLIX) capability. Dual 128b memory ports achieve the memory access rate of 256b of load or store per cycle and can support one radix-4 complex FFT butterfly operation per cycle. In fact, the computational units used for the FFT and FIR operations dwarf the base processor itself, so that the complete programmable OFDM processor is only slightly larger than FFT/FIR data-path alone. Moving from a hardwired function unit, suitable for only narrow set of pre-set FFT and/or FIR operations, to a programmable OFDM processor adds relatively little silicon area and has little impact on OFDM throughput.

Figure 3 sketches the physical structure and resources of the OFDM engine.

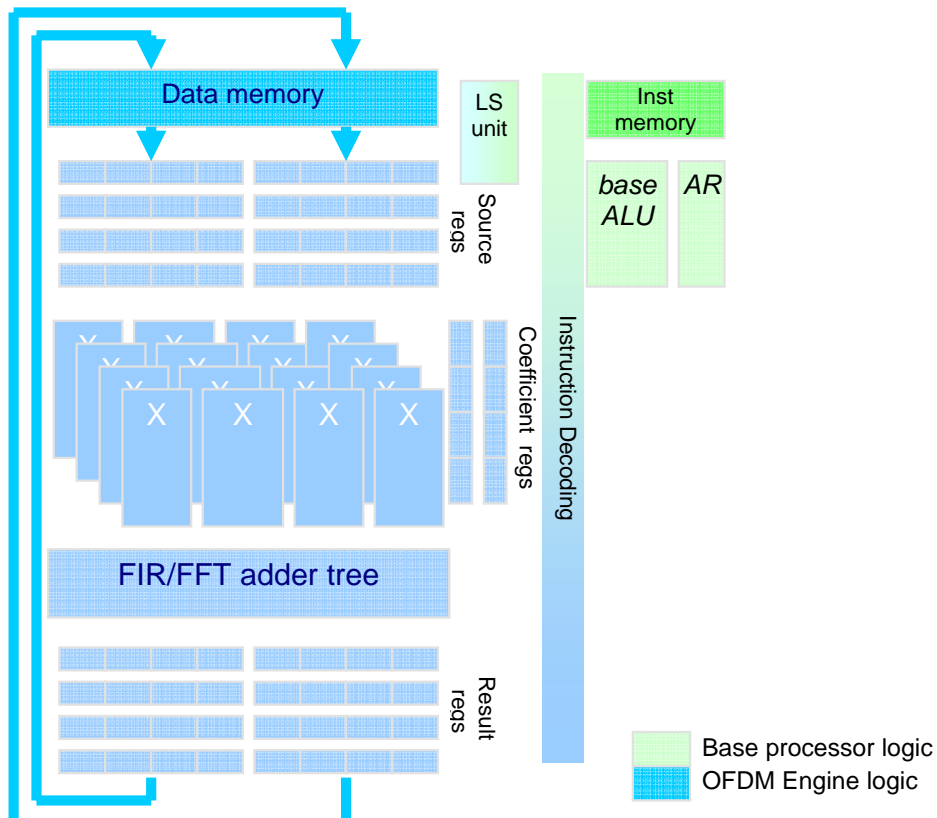


FIGURE 3 OFDM ENGINE STRUCTURE AND RESOURCES

The OFDM engine adds roughly 60 instructions to the processor, including the following groups:

Group	Instruction Names
Twiddle factor loads	lt03i, lt47i, lt811i
Loads to special source registers and accumulators:	lp03xu, lr03xu, lp47xu, lr47xu lq03xu, ls03xu, lq47xu, ls47xu
Stores from special result registers	sw03xu, sy03xu, sw47xu, sy47xu, sx03xu, sz03xu, sx47xu, sz47xu
Stores with digit reversed addressing:	sw03ru, sy03ru, sw47ru, sy47ru, sx03ru, sz03ru, sx47ru, sz47ru, sw03bru, sy03bru, sw47bru, sy47bru, sx03bru, sz03bru, sx47bru, sz47bru

Group	Instruction Names
Accumulator stores	sq03xu, sq47xu
Radix-4 butterfly operations	bf0, bf1, bf2, bf3, bf4, bf5, bf6, bf7
Quad complex FIR operations	fir0, fir1, fir2, fir3, fir4, fir5, fir6, fir7
Butterfly for last pass	bfp03,bfq03,bfr03,bfs03,bfp47,bfq47,bfr47,bfs47

The behavior of butterfly and bit reversed addressing instructions depends on modes set in a mode register. The bits of a 7-bit mode register are defined as follows:

Bits	Definition	Settings
0	Perform additional one bit right shift of multiply results during normalization and saturation	0: no shift 1: one bit right shift
2:1	Do trivial radix-2 or trivial radix-4 computation for last pass butterfly operations (bfp03, bfq03, bfr03, bfs03, bfp47, bfq47, bfr47, bfs47)	00: reserved 01: radix-4 10: radix-2 11: reserved
7:3	Set bit reversed addressing increment point. Carry propagates down from this bit position	7:3 is bit position – 4 bits

3 Implementing Fast Radix-4 FFT

The design implements instructions for the decimation-in-frequency FFT. The basic butterfly operation uses four complex pairs {a, b, c, d} as input, and computes four complex pairs {w, x, y, z} as output, according to the following equations:

$$\begin{aligned}
 w &= a + b + c + d \\
 x &= (a - j * b - c + j * d) * t1 \\
 y &= (a - b + c - d) * t2 \\
 z &= (a + j * b - c - j * d) * t3
 \end{aligned}$$

The twiddle factors {t1, t2, t3} depend on the pass and offset of the butterfly in the overall calculation. For details on radix-4 FFT computation, please refer to a standard reference, or to http://en.wikipedia.org/wiki/Fast_Fourier_transform.

The basic radix-4 decimation-in-frequency does three complex multiplies, or 12 real multiplies per butterfly, plus a number of additions. The OFDM engine optionally implements an additional four multipliers to support FIR filtering. In each cycle of the inner loop, two 128b loads or two 128b stores are performed in parallel with one radix-4 FFT butterfly. Each load fetches four adjacent complex pairs (16b + 16b) and puts the pairs in state registers p, q, r, and s (or more accurately a set of four 32b state registers) - that is:

```

load {p0,p1,p2,p3}
load {q0,q1,q2,q3}
load {r0,r1,r2,r3}
load {s0,s1,s2,s3}
    
```

Therefore, four loads get four pairs from four different addresses in memory. Successive butterfly operations work on the first, then the second, the third, and then the fourth pair of each state register:

```

{w0,x0,y0,z0} = butterfly (p0,q0,r0,s0) /*using appropriate twiddle factors loaded
in state registers t1,t2,t3*/
{w1,x1,y1,z1} = butterfly (p1,q1,r1,s1)
{w2,x2,y2,z2} = butterfly (p2,q2,r2,s3)
{w3,x3,y3,z3} = butterfly (p3,q3,r3,s3)
    
```

while the four complex pairs making up a butterfly result are written into successive elements of four result register sets, w, x, y, z, each of which is a 32b register holding a complex pair. The butterfly calculations perform 16b signed multiplies, yielding two 32b partial products for each multiply. Appropriate products of real and imaginary components are combined in a compound adder tree – 12 adds per final real or imaginary part. The adder tree maintains 34 bits of precision, and is followed by a saturation step to return each real and imaginary component to a 16b precision. The saturation step also allows for a 0 to 3 bit right normalization shift to ensure that values stay in the appropriate range. The shift amount is controlled by a 2-bit read-write register. The real and imaginary results are paired and placed in the registers w, x, y, and z. Four successive stores then write four adjacent pairs back to the appropriate place in memory - that is:

```
store {w0,w1,w2,w3}
store {x0,x1,x2,x3}
store {y0,y1,y2,y3}
store {z0,z1,z2,z3}
```

So far, we have ignored the pipelining of operations. The loads, the butterflies and the stores are all pipelined within the processor. Loads automatically have a one-cycle latency, so that results are not available until the second cycle after the load itself. The butterfly multiply-add operations should be pipelined across three cycles to avoid limiting the maximum operating frequency of the hardware implementation. This pipelining is performed automatically simply by specifying the result stage (the "def stage") of the result registers. This is specified in a TIE `schedule` statement. The store can be performed as soon as the result is ready. With two load/stores per cycle, the basic schedule of one 4-way butterfly operation looks like the following code sequence:

```
load p, load q
load r, load s
<one cycle load delay>
butterfly0
butterfly1
butterfly2
butterfly3
<two cycle latency of butterfly>
store w, store x
store y, store z
```

We can also overlap the butterfly and the load/store operations, so that a set of 4 loads, 4 stores and 4 butterflies can be packed into 4 instructions. This is done automatically in Xtensa LX2 with FLIX, but packing of a load and ALU op can also be done in Xtensa 7 simply by specifying instructions that perform both tasks as a single operation. Note, however, that we cannot start loading new values into p, q, r and s until the fourth butterfly operation has started (and we can't start a new set of four butterfly operations until the store of y and z has started.) We can, however, load an alternate set of p, q, r and s registers, and then do a second set of four butterfly operations on this alternate set, while we store the results of the first butterfly sequence. By ping-ponging between two sets of source and destination registers (unrolling the loop one time), the depth of the pipeline is completely hidden and we can sustain a rate of 1 128b load, 1 128b store and one butterfly per cycle.

This TIE package includes support for power of two FFT sizes. For even powers of two (e.g., 256, 1024, and 4096 points) and problem size of N, the FFT computation requires $\log_4(N)$ radix-4 butterflies. However, the last pass has trivial twiddle factors and requires no multiplies. For odd-powers of two (e.g., 128, 512, and 2048 points) radix-4 computation is used for all but the last pass. Radix-2 with trivial twiddles is used for the last pass.

FFT computations may make several different assumptions about the ordering of results and the use of input and output buffers. The simplest FFT computations do not naturally produce results in proper sequential order. Instead, this "natural ordering" is bit-reversed – the data for result k is placed at an offset in the result buffer reflected by the bit reversal of the offset k . Alternatively, the processor can also take into account this bit reversal during the storing of data

on the last pass of the FFT, so that results appear with “normal ordering”. In addition, FFT may do the computation “in place”, overwriting the working data, or into a separate output buffer. The combination of “in place” and “in normal order” presents some challenges. Loads and stores must be carefully ordered on the last pass to read all source data from destination locations that will be written. This makes the last pass somewhat slower in the in place/normal order case.

Here is the prototype code for the inner loop of a typical pass of the FFT: *p* and *q* are source data address pointers, *w* and *x* are destination address pointers. Multiple pointers are used so that *p* and *q* can be updated in the same cycle, and *w* and *x* can be updated in the same cycle when two load or stores are executed in a single VLIW instruction. The variable *dstride* indicates twice the offset between the four source (or destination) addresses for data for the butterfly. *wstepdown* is the offset in the pointers required to go back to the beginning of the array for the next iteration.

```
/* initialize pointers and offsets */
dstride = 512;
wstepdown = 16 - 512;
p = data;
w = p - wstepdown;
q = p + 256;
x = q - wstepdown;
/* load first 12 twiddle factors */
lt03i((unsigned)t,0);
lt47i((unsigned)t,16);
lt811i((unsigned)t,32);
/* load data for first four butterflies */
lp03xu((unsigned)p,0);
lq03xu((unsigned)q,0);
lr03xu((unsigned)p,dstride);
ls03xu((unsigned)q,dstride);
/* perform first four butterflies into first set of dest regs */
bf0();
bf1();
bf2();
bf3();
/* load next 12 twiddle factors */
lt03i((unsigned)t,48);
lt47i((unsigned)t,64);
lt811i((unsigned)t,80);
/* and update pointer to twiddles */
inc((unsigned)t,96);
/* load data into next four butterflies into second set of source regs */
lp47xu((unsigned)p,wstepdown);
lq47xu((unsigned)q,wstepdown);
lr47xu((unsigned)p,dstride);
ls47xu((unsigned)q,dstride);
for (i = 0; i < 8; i += 1) { /* 8 radix 4 butterflies per iteration */
    /* store results of four butterflies from first set of dest regs*/
    sw03xu((unsigned)w,wstepdown);
    sy03xu((unsigned)x,wstepdown);
    sx03xu((unsigned)w,dstride);
    sz03xu((unsigned)x,dstride);
    /* perform four butterflies into second set of dest regs*/
    bf4();
    bf5();
    bf6();
    bf7();
    /* store results of four butterflies from second set of dest regs */
    sw47xu((unsigned)w,wstepdown);
    sy47xu((unsigned)x,wstepdown);
    sx47xu((unsigned)w,dstride);
    sz47xu((unsigned)x,dstride);
    /* load data into first set of source regs */
    lp03xu((unsigned)p,wstepdown);
    lq03xu((unsigned)q,wstepdown);
    lr03xu((unsigned)p,dstride);
    ls03xu((unsigned)q,dstride);
}
```

```

/* load 12 twiddle factors */
lt03i((unsigned)t,0);
lt47i((unsigned)t,16);
lt811i((unsigned)t,32);
/* perform four butterflies into first set of dest regs */
bf0();
bf1();
bf2();
bf3();
/* load data into second set of source regs */
lp47xu((unsigned)p,wstepdown);
lq47xu((unsigned)q,wstepdown);
lr47xu((unsigned)p,dstride);
ls47xu((unsigned)q,dstride);
/* load 12 twiddle factors */
lt03i((unsigned)t,48);
lt47i((unsigned)t,64);
lt811i((unsigned)t,80);
/* update pointer to twiddles */
inc((unsigned)t,96);
}

```

After compilation, the inner loop is tightly scheduled so that in each iteration, eight quad loads, eight quad stores, eight butterflies and 12 twiddle factors are executed in 12 cycles.

```

0x60000c11 <fft_c+73>: loopgtz a9, 0x60000c44 <fft_c+124>
0x60000c14 <fft_c+76>: { lp03xu a4, a8; sy03xu a3, a8; nop }
0x60000c18 <fft_c+80>: { sw03xu a2, a8; lq03xu a5, a8; bf6 }
0x60000c1c <fft_c+84>: { lr03xu a4, a7; sz03xu a3, a7; bf7 }
0x60000c20 <fft_c+88>: { sx03xu a2, a7; nop; bf5 }
0x60000c24 <fft_c+92>: { lt811i a6, 32; ls03xu a5, a7; bf4 }
0x60000c28 <fft_c+96>: { lt03i a6, 0; lt47i a6, 16; nop }
0x60000c2c <fft_c+100>: { sw47xu a2, a8; lq47xu a5, a8; nop }
0x60000c30 <fft_c+104>: { lp47xu a4, a8; sy47xu a3, a8; bf2 }
0x60000c34 <fft_c+108>: { lr47xu a4, a7; ls47xu a5, a7; bf1 }
0x60000c38 <fft_c+112>: { sx47xu a2, a7; sz47xu a3, a7; bf3 }
0x60000c3c <fft_c+116>: { lt03i a6, 48; lt47i a6, 64; bf0 }
0x60000c40 <fft_c+120>: { lt811i a6, 80; inc a6, 96; nop }

```

Prototype C code for FFT, tuned to 256 points, is attached as Appendix A. Prototype TIE code for the full FFT/FIR extension for Xtensa LX2 is attached as Appendix B. Also note that this is offered as a design example for illustration purposes.

4 Complex FIR Support

FIR filtering commonly occurs in QAM and QPSK modulation and demodulation. It may also be used in signal shaping or time-domain equalization or to remove effects of ISI (Inter-Symbol interference). This note does not go into detail on the FIR features of the architecture. Each FIR instruction operates on a single complex pair accumulator at a time, but the eight FIR instructions use the same data and coefficients at different offsets to perform part of eight different accumulations for each load of the data. This minimizes the number of loads required to feed the multiply-accumulate units, when computing FIR filters with large numbers of taps. However, data loads and accumulator loads and stores can be executed in parallel with the FIR filter operations, so other data organizations are simple. The implementation uses p0..p7 as data registers, t0..t3 as coefficient registers and q0..q7 as accumulators. There are eight FIR instructions, which perform one of the following accumulation instructions (*i.e.*, four complex taps per cycle).

```

q0 = q0 + p0*t0 + p1*t1 + p2*t2 + p3*t3 /*each * and + is complex*/
q1 = q1 + p1*t0 + p2*t1 + p3*t2 + p4*t3
...
q7 = q7 + p7*t0 + p0*t1 + p1*t2 + p2*t3

```

5 Results

The approach outlined here creates a special purpose processor with high throughput, modest area and low power. The processor not only includes the OFDM instruction set, but also implements the full Xtensa baseline instruction set, so any C-based application can be compiled and run on this processor. This specialized OFDM configuration does not implement a full-range of high-performance general-purpose DSP instructions, but the extensions for general-purpose DSP would be only a modest extra adder in area and typical power. The most expensive features, the dual-ported memory, the multipliers and adders are already in place.

Power and Area

The power and area for the OFDM Engine (Processor Area), instruction and data memories are shown in the following table. Power dissipation assumes fine-grained clock gating and was obtained by using the in-order 256 complex points FFT diag.

TABLE 1 AREA AND POWER FOR OFDM ENGINE WITH FIR FILTER

90G process/ 350MHz processor	Instruction Memory	Data Memory	Processor Area	Processor + Memory Area
Gates (Post Syn) / Bytes	1KB (Width 32b)	4KB (Width 128b)	202K gates	202K gates + 5KB memory
Post route Cell Area (mm ²)	0.044	0.14	0.92	1.104
Power (μW/MHz)	17	131	233	381

TABLE 2 AREA AND POWER FOR OFDM ENGINE WITHOUT FIR FILTER

90G process/ 350MHz processor	Instruction Memory	Data Memory	Processor Area	Processor + Memory Area
Gates (Post Syn) / Bytes	1KB (Width 32b)	4KB (Width 128b)	175K gates + 5KB	175K gates + 5KB memory
Post route Cell area (mm ²)	0.044	0.14	0.79	0.974
Power (μW/MHz)	16.5	123	166	305.5

Performance

TABLE 3 PERFORMANCE NUMBERS FOR OFDM OPERATION

	In order, in place FFT	Out order /In order, not in place FFT
256 point FFT	476 Cycles	374 Cycles
4096 point FFT	8655 Cycles	6385 Cycles

The design combines a generic Xtensa LX2 processor of about approximately 55K gates of logic. Of this about 44K gates is taken up by the load store unit which would be part of any RTL implementation. The processor solution only adds about 11K gates of extra logic as compared to the RTL implementation of the OFDM Engine

The Xtensa LX2 processor is roughly twice the size of the smallest possible Xtensa because this configuration includes zero-overhead loop support and dual-ported 128b data memory. Several possible enhancements to the core might also be considered. A general-purpose processor interface could be included, at a cost of about 27K gates, and this can be easily removed at the cost of increasing the local memory size. Alternatively, OFDM data could be streamed in and out of the processor over optimized queues. Input and output queues, each 128 bits wide would add about 15K gates. The logic implementing the OFDM Engine, with the FIR extension requires roughly 147K gates plus the cost of the base processor (which includes load store unit). Without the FIR filter support, the total logic is reduced by about 27K gates.

Power is also a critical consideration for modem design. Conventional wisdom says that processors consume more power than hardwired DSP data-paths. Automatic processor generation changes the picture. The Xtensa Processor Generator includes automatic pipeline analysis for each function unit, pipe stage and result computation. It creates clock gating for each unit based on the dynamic scheduling of instructions in the pipeline. This means more fine-grained and accurate clock shut-off for the data-path and control logic than can be achieved by simple RTL level analysis during logic synthesis. In theory, of course, the designer of the hypothetical RTL data-path could do such fine-grained clock gating, but in practice schedule pressure rarely allows this luxury. We can look at estimated power dissipation for the block running the FFT kernel. Without fine grained clock gating, the overall processor power would consume two times the power as compared to a processor with fine grained clock gating. The processor with OFDM extensions is estimated to run at up to 350MHz in 90G technology under worst case conditions.

Execution of 256 point complex FFT requires about 374 cycles for natural ordered or normally ordered, non-in-place computation. Performance would degrade if the instructions were loaded from system memory instead of local memories. For in-place computation, about 476 cycles is required. This is largely limited by the number of data read and write cycles. The other cycles are needed for priming the FFT pipeline and control overhead, especially between passes. FFT for 4096 complex points requires about 6385 cycles for non-in-place or naturally ordered results and about 8655 cycles for in-place, normally ordered results. For the 4096 complex points FFT data memory size needs to be increased to 64Kbytes assuming a configuration without any system memory.

Code size depends on the degree of loop unrolling used. Unrolling the outer loop simplifies addressing calculations, but causes some code duplication. Nevertheless, the code for an FFT is not terribly large. An aggressively unrolled implementation for FFT256 requires about 486 bytes for the `fft_c` function. In 90nm technology, the incremental area of 486 bytes of RAM is less than 0.01 mm², or less than the area of 2000 logic gates.

6 Summary

This note shows an efficient method for implementing high performance FFT and FIR operations using Xtensa LX2 extensible processors. These processors are small, energy-efficient, easily configured and highly programmable. A single Xtensa processor, optimized for OFDM can achieve FFT times as low as 18us for 4096pt complex FFT and as low as 1us for 256pt complex FFT @ 350MHz (worst case, 90nm foundry process). This combination of programmability and performance enables more rapid and flexible communications platform designs.



Appendix A – FFT Example C Source

fft256.c

```

/* Copyright (c) 2004-2007 by Tensilica Inc. ALL RIGHTS RESERVED.
 * These coded instructions, statements, and computer programs are the
 * copyrighted works and confidential proprietary information of Tensilica Inc.
 * Tensilica Inc. They may be adapted and modified by bona fide
 * purchasers for internal use, but no adapted or modified version may be
 * disclosed or distributed to third parties in any manner, medium, or
 * form, in whole or in part, without the prior written consent of
 * Tensilica Inc.
 *
 * This software and its derivatives are to be executed solely on products
 * incorporating the Xtensa(R) Microprocessor.
 */

#include <xtensa/tie/OFDMEngine3.h>
#include <xtensa/tie/xt_misc.h>
#include "fft_problem_256.h"
#include <stdio.h>

void fft_c() __attribute__((section(".iram0.text")));
void showdata(long int *d, int sz,int dpl) {
    int i,j;
    signed short rl,im;
    for (i=0;i<sz;i+=dpl) {
        //printf("%3d: ",i);
        for (j=0;j<dpl;j++) {
            rl = d[i+j]&0xffff;
            im = (d[i+j]>>16)&0xffff;
            //printf(" r %5d i %5d",rl,im);
        }
        //printf("\n");
    }
}

void show(unsigned a, unsigned b) {
    printf("store addr %08x base %08x offset %08x: ",a,b,a-b);
    printf("store data %04x/%04x %04x/%04x %04x/%04x %04x/%04x\n", (unsigned
short)*((unsigned short *)a), (unsigned short)*(((unsigned short *)a)+1), (unsigned
short)*(((short *)a)+2), (unsigned short)*(((unsigned short *)a)+3), (unsigned
short)*(((unsigned short *)a)+4), (unsigned short)*(((unsigned short *)a)+5), (unsigned
short)*(((unsigned short *)a)+6), (unsigned short)*(((unsigned short *)a)+7));
}

void showload(unsigned a, unsigned b) {
    printf("load addr %08x base %08x offset %08x: ",a,b,a-b);
    printf("load data %04x/%04x %04x/%04x %04x/%04x %04x/%04x\n", (unsigned short)*((unsigned
short *)a), (unsigned short)*(((unsigned short *)a)+1), (unsigned short)*(((short
*)a)+2), (unsigned short)*(((unsigned short *)a)+3), (unsigned short)*(((unsigned short
*)a)+4), (unsigned short)*(((unsigned short *)a)+5), (unsigned short)*(((unsigned short
*)a)+6), (unsigned short)*(((unsigned short *)a)+7));
}

/* Currently handles powers of 2 size, with last pass results either
 * unswapped to proper order or left in natural swapped order*/
void
fft_c(char          data[],
      char          outdata[],
      const char *twiddles)
{
    register unsigned int dstride = 512; /* char address - two quads per stride */

    register unsigned int i,j;

    register int wstepdown;

```

```

register char *p,*q,*w,*x,*y,*z; /* pointers into src, dst */
register char *t; /*pointer to twiddles */

t = (char*)twiddles;
/* in place radix-4 processing - leaves output data in swapped order */
/* preset pointers for pre-update computation*/

wstepdown = 16 - 512;
p = data;
w = p - wstepdown;
q = p + 256;
x = q - wstepdown;
lt03i((unsigned)t,0);
lt47i((unsigned)t,16);
lt811i((unsigned)t,32);
lp03xu((unsigned)p,0);          lq03xu((unsigned)q,0);
lr03xu((unsigned)p,dstride);   ls03xu((unsigned)q,dstride);
                                bf0();
                                bf1();
                                bf2();
                                bf3();
lt03i((unsigned)t,48);          lt47i((unsigned)t,64);
lt811i((unsigned)t,80);          inc((unsigned)t,96);
lp47xu((unsigned)p,wstepdown);  lq47xu((unsigned)q,wstepdown);
lr47xu((unsigned)p,dstride);    ls47xu((unsigned)q,dstride);
for (i = 0; i < 8; i += 1) {
    /* 8 radix 4 butterflies per iteration */
    lp03xu((unsigned)p,wstepdown); lq03xu((unsigned)q,wstepdown); bf4();
    lr03xu((unsigned)p,dstride);   ls03xu((unsigned)q,dstride);   bf5();
    sw03xu((unsigned)w,wstepdown); sy03xu((unsigned)x,wstepdown); bf6();
    sx03xu((unsigned)w,dstride);   sz03xu((unsigned)x,dstride);   bf7();
    lt03i((unsigned)t,0);           lt47i((unsigned)t,16);
    lt811i((unsigned)t,32);
    lp47xu((unsigned)p,wstepdown); lq47xu((unsigned)q,wstepdown); bf0();
    lr47xu((unsigned)p,dstride);   ls47xu((unsigned)q,dstride);   bf1();
    sw47xu((unsigned)w,wstepdown); sy47xu((unsigned)x,wstepdown); bf2();
    sx47xu((unsigned)w,dstride);   sz47xu((unsigned)x,dstride);   bf3();
    lt03i((unsigned)t,48);          lt47i((unsigned)t,64);
    lt811i((unsigned)t,80);          inc((unsigned)t,96);
}

dstride = 128;
/* second pass: interval 16, repeat 4 */
/* unroll inner loop 2x, outer loop 4x */
p = data;
w = p - (16-7*dstride);
q = p + dstride/2;
x = q - (16-7*dstride);
t = t - 96; /* roll back from extra twiddle loads */
lp03xu((unsigned)p,0);          lq03xu((unsigned)q,0);
lr03xu((unsigned)p,dstride);   ls03xu((unsigned)q,dstride);
lt03i((unsigned)t,0);          lt47i((unsigned)t,16);
lt811i((unsigned)t,32);        inc((unsigned)t,48);
lp47xu((unsigned)p,dstride);   lq47xu((unsigned)q,dstride); bf0();
lr47xu((unsigned)p,dstride);   ls47xu((unsigned)q,dstride); bf1();
                                bf2();
                                bf3();

for (i = 0; i < 4; i += 1) {
    /* four iterations of outer loop at a time */
    /* inner loop fully unrolled */
    /*16 radix 4 butterflies per iteration */

    lp03xu((unsigned)p,dstride);   lq03xu((unsigned)q,dstride);   bf4();
    lr03xu((unsigned)p,dstride);   ls03xu((unsigned)q,dstride);   bf5();
    sw03xu((unsigned)w,16 - 7*dstride); sy03xu((unsigned)x,16 - 7*dstride); bf6();
    sx03xu((unsigned)w,dstride);   sz03xu((unsigned)x,dstride);   bf7();
    lp47xu((unsigned)p,dstride);   lq47xu((unsigned)q,dstride);   bf0();
    lr47xu((unsigned)p,dstride);   ls47xu((unsigned)q,dstride);   bf1();
    sw47xu((unsigned)w,dstride);   sy47xu((unsigned)x,dstride);   bf2();
    sx47xu((unsigned)w,dstride);   sz47xu((unsigned)x,dstride);   bf3();
    /* start fetching from beginning, +4 */
}

```

```

lp03xu((unsigned)p,16 - 7*dstride);    lq03xu((unsigned)q,16 - 7*dstride); bf4();
lr03xu((unsigned)p,dstride);          ls03xu((unsigned)q,dstride);    bf5();
sw03xu((unsigned)w,dstride);          sy03xu((unsigned)x,dstride);    bf6();
sx03xu((unsigned)w,dstride);          sz03xu((unsigned)x,dstride);    bf7();
/* load twiddles for next outer loop iteration */
lt03i((unsigned)t,0);                  lt47i((unsigned)t,16);
lt81li((unsigned)t,32);                inc((unsigned)t,48);

lp47xu((unsigned)p,dstride);          lq47xu((unsigned)q,dstride);  bf0();
lr47xu((unsigned)p,dstride);          ls47xu((unsigned)q,dstride);  bf1();
sw47xu((unsigned)w,dstride);          sy47xu((unsigned)x,dstride);  bf2();
sx47xu((unsigned)w,dstride);          sz47xu((unsigned)x,dstride);  bf3();
}
/* done with second pass */
dstride = 32;
w = p = data - dstride;
x = q = p + 16;
lp03xu((unsigned)p,dstride);          lq03xu((unsigned)q,dstride);
lr03xu((unsigned)p,dstride);          ls03xu((unsigned)q,dstride);
/* twiddles already loaded in last iteration of previous pass */
lp47xu((unsigned)p,dstride);          lq47xu((unsigned)q,dstride);    bf0();
lr47xu((unsigned)p,dstride);          ls47xu((unsigned)q,dstride);    bf1();
                                        bf2();
                                        bf3();

for (j = 0; j < 8; j += 1) {
/* outer loop fully unrolled */
/* inner loop unrolled 2 times*/
/*16 radix 4 butterflies per iteration */
lp03xu((unsigned)p,dstride);          lq03xu((unsigned)q,dstride);  bf4();
lr03xu((unsigned)p,dstride);          ls03xu((unsigned)q,dstride);  bf5();
sw03xu((unsigned)w,dstride);          sy03xu((unsigned)x,dstride);  bf6();
sx03xu((unsigned)w,dstride);          sz03xu((unsigned)x,dstride);  bf7();
lp47xu((unsigned)p,dstride);          lq47xu((unsigned)q,dstride);  bf0();
lr47xu((unsigned)p,dstride);          ls47xu((unsigned)q,dstride);  bf1();
sw47xu((unsigned)w,dstride);          sy47xu((unsigned)x,dstride);  bf2();
sx47xu((unsigned)w,dstride);          sz47xu((unsigned)x,dstride);  bf3();
}
if ( bitrev_table != 0) {
if (data == outdata) {
unsigned int k;
/* do standard third pass, then permute in last pass */
/* Trip counts in table are biased for SWP windup, so we add 1 to each here. */
unsigned int palindrome_count = bitrev_table[0] + 1;
unsigned int nonpalindrome_count = bitrev_table[1] + 1;
j = 2;
for (k = 0; k < palindrome_count/2; k++) {
p = q = x = w = data + (bitrev_table[j++]);
lp03xu((unsigned)p,0);                lq03xu((unsigned)q,256);
lr03xu((unsigned)p,512);              ls03xu((unsigned)q,512);
p = q = z = y = data + (bitrev_table[j++]);
bfp03();
bfq03();
bfr03();
bfs03();
lp47xu((unsigned)p,0);                lq47xu((unsigned)q,256);
lr47xu((unsigned)p,512);              ls47xu((unsigned)q,512);
sw03xu((unsigned)w,0);                sy03xu((unsigned)x,512);

sx03xu((unsigned)w,256);              sz03xu((unsigned)x,256);
bfp47();
bfq47();
bfr47();
bfs47();
sw47xu((unsigned)y,0);                sy47xu((unsigned)z,512);

sx47xu((unsigned)y,256);              sz47xu((unsigned)z,256);
}
for (k = 0; k < nonpalindrome_count; k++) {
p = q = x = w = data + (bitrev_table[j++]);

```

```

        lp03xu((unsigned)p,0);          lq03xu((unsigned)q,256);
        lr03xu((unsigned)p,512);        ls03xu((unsigned)q,512);
        p = q = z = y = data + (bitrev_table[j++]);
        bfp03();
        bfq03();
        bfr03();
        bfs03();
        lp47xu((unsigned)p,0);          lq47xu((unsigned)q,256);
        lr47xu((unsigned)p,512);        ls47xu((unsigned)q,512);
        sw03xu((unsigned)y,0);          sy03xu((unsigned)z,512);

        sx03xu((unsigned)y,256);        sz03xu((unsigned)z,256);
        bfp47();
        bfq47();
        bfr47();
        bfs47();
        sw47xu((unsigned)w,0);          sy47xu((unsigned)x,512);
        sx47xu((unsigned)w,256);        sz47xu((unsigned)x,256);
    }
} else { /* do bit reversed addressing into outdata buffer */
    Wmode(0x1a);
    dstride = 512;
    q = p = data;
    w = outdata;
    x = w + 256;
    lp03xu((unsigned)p,0);              lq03xu((unsigned)q,256);
    lr03xu((unsigned)p,512);            ls03xu((unsigned)q,512);
    lp47xu((unsigned)p,16-512);        lq47xu((unsigned)q,16-512);
bfp03();                                lr47xu((unsigned)p,512);        ls47xu((unsigned)q,512);
bfq03();
bfr03();
bfs03();

    for (j = 0; j < 8; j += 1) {
        /* outer loop fully unrolled */
        /* inner loop unrolled 2 times*/
        /*16 radix 4 butterflies per iteration */
        lp03xu((unsigned)(unsigned)p,16-512);
    lq03xu((unsigned)(unsigned)q,16-512);    bfp47();
        lr03xu((unsigned)(unsigned)p,512);
    ls03xu((unsigned)(unsigned)q,512);        bfq47();
        sw03xu((unsigned)(unsigned)w,0);
    sx03xu((unsigned)(unsigned)x,0);          bfr47();
        sy03bru((unsigned)w,512);
    sz03bru((unsigned)x,512);                bfs47();
        lp47xu((unsigned)(unsigned)p,16-512);
    lq47xu((unsigned)(unsigned)q,16-512);    bfp03();
        lr47xu((unsigned)(unsigned)p,512);
    ls47xu((unsigned)(unsigned)q,512);        bfq03();
        sw47xu((unsigned)(unsigned)w,0);
    sx47xu((unsigned)(unsigned)x,0);          bfr03();
        sy47bru((unsigned)w,512);
    sz47bru((unsigned)x,512);                bfs03();
    }
}

} else {
    Wmode(0x0);
    dstride = 512;
    q = p = data;
    w = outdata;
    x = w + 256;
    lp03xu((unsigned)p,0);              lq03xu((unsigned)q,256);
    lr03xu((unsigned)p,512);            ls03xu((unsigned)q,512);

```

```

        lp47xu((unsigned)p,dstride);        lq47xu((unsigned)q,dstride); bfp03();
        lr47xu((unsigned)p,dstride);        ls47xu((unsigned)q,dstride); bfq03();

bfr03();

bfs03();
    for (j = 0; j < 8; j += 1) {
        /* outer loop fully unrolled */
        /* inner loop unrolled 2 times*/
        /*16 radix 4 butterflies per iteration */
        lp03xu((unsigned)p,dstride);        lq03xu((unsigned)q,dstride); bfp47();
        lr03xu((unsigned)p,dstride);        ls03xu((unsigned)q,dstride); bfq47();
        sw03xu((unsigned)w,dstride);        sy03xu((unsigned)x,dstride); bfr47();
        sx03xu((unsigned)w,dstride);        sz03xu((unsigned)x,dstride); bfs47();
        lp47xu((unsigned)p,dstride);        lq47xu((unsigned)q,dstride); bfp03();
        lr47xu((unsigned)p,dstride);        ls47xu((unsigned)q,dstride); bfq03();
        sw47xu((unsigned)w,dstride);        sy47xu((unsigned)x,dstride); bfr03();
        sx47xu((unsigned)w,dstride);        sz47xu((unsigned)x,dstride); bfs03();
    }
}

void main()
{
short *s,*d;
int pass = 1;
int i;

fft_c(
(char *) (data), /* location of in data in local RAM */
(char *) (data), /* location of out data, if necessary in local RAM */
(char *) (twiddles) /* long int twiddle[4096] in local RAM */);

for (i = 0, s = expected_result,d = (short *)data; i < problem_size*2; i++,s++,d++)
if (*s != *d) pass = 0;
if (pass)
printf("PASS\n");
else
printf("FAIL\n");
}

```



Appendix B – OFDM Engine TIE Source

OFDMEngine3.tie

```

//Copyright Tensilica Inc. 2005-2006
//FFT accelerator
//FFT based on Decimation in Freq radix 4 butterfly per cycle
// slot assignments for 3 slot in 32b FLIX
;my $FIR = 1;
length l_flix32_0 32 {InstBuf[3:1] == 3'b111}
format flix32_0 l_flix32_0 { ldst0, ldst1, arith }
slot_opcodes ldst0 {lp03xu, lr03xu, lp47xu, lr47xu, sw03xu, sx03xu, sw47xu, sx47xu, sw03bru,
sy03bru, sw47bru, sx47bru, lt03i, lt811i, MOV.N}
slot_opcodes ldst1 {lq03xu, ls03xu, lq47xu, ls47xu, sy03xu, sz03xu, sy47xu, sz47xu, sx03bru,
sz03bru, sy47bru, sz47bru, lt47i, inc, MOV.N}
slot_opcodes arith {bf0, bf1, bf2, bf3, bf4, bf5, bf6,
bf7,bfp03,bfq03,bfr03,bfs03,bfp47,bfq47,bfr47,bfs47
;if ($FIR) {
    ,fir0, fir1, fir2, fir3, fir4, fir5, fir6 ,fir7
;}
}
//normalization shift amount register
//bit 0 : normalization shift
//bit 2:1 : 00 standard, 01: trivial r4 butterfly of last pass, 10: trivial r2 butterfly of
last pass, 11: undefined, */
//bit 6:3 :bit reversed increment position for br addressing on stores - 4
state mode 7 7'h0 add_read_write
// source states, written by load, read by butterfly, 0-3 and 4-7 are separate butterflies
// result state, written by butterfly, read by store

;foreach my $i (0..7) {
state p`$i` 32
state q`$i` 32
state r`$i` 32
state s`$i` 32
state w`$i` 32
state x`$i` 32
state y`$i` 32
state z`$i` 32
;}
//twiddle states, written by load, read by butterfly
;foreach my $i (0..11) {
state t`$i` 32
;}

;foreach my $j (0..1) {
; foreach my $k ("p","q","r","s") {
//memory operations - offset is word offset
// load instructions
operation l`$k``$j*4``$j*4+3`xu {inout AR a, in AR i} {out VAddr, in MemDataIn128, out
`$k``$j*4`, out `$k``$j*4+1`, out `$k``$j*4+2`, out `$k``$j*4+3`} {
wire [31:0] address = a + i;
assign a = address;
assign VAddr = address;
assign `$k``$j*4` = MemDataIn128[31:0];
assign `$k``$j*4+1` = MemDataIn128[63:32];
assign `$k``$j*4+2` = MemDataIn128[95:64];
assign `$k``$j*4+3` = MemDataIn128[127:96];}
; }
;}

immediate_range iq 0 112 16
operation inc {inout AR r, in iq i} {} {assign r = r + i;}

;foreach my $i (0..2) {

```

```

operation lt`$i*4`$i*4+3`i {in AR a, in iq i} {out VAddr, in MemDataIn128, out t`$i*4`, out
t`$i*4+1`, out t`$i*4+2`, out t`$i*4+3`} {
wire [31:0] address = a + i;
assign VAddr = address;
assign t`$i*4` = MemDataIn128[31:0];
assign t`$i*4+1` = MemDataIn128[63:32];
assign t`$i*4+2` = MemDataIn128[95:64];
assign t`$i*4+3` = MemDataIn128[127:96];
}
};
// bit reversed address calculation, increment at bit position brp+4, with downward carry
function [31:0] bitrev([31:0]a,[3:0]brp) slot_shared {
wire [15:0] f = a[19:4];
wire [15:0] brfld =
{f[0],f[1],f[2],f[3],f[4],f[5],f[6],f[7],f[8],f[9],f[10],f[11],f[12],f[13],f[14],f[15]};
wire [15:0] g = brfld + (16'h8000>>brp);
wire [15:0] fldbrinc =
{g[0],g[1],g[2],g[3],g[4],g[5],g[6],g[7],g[8],g[9],g[10],g[11],g[12],g[13],g[14],g[15]};
// include stride decrement in result
assign bitrev = {a[31:20],fldbrinc,a[3:0]};
}

// store instructions
;foreach my $j (0..1) {
;if ($FIR) {
operation sq`$j*4`$j*4+3`xu {inout AR a, in AR i} {out VAddr, out MemDataOut128, in q`$j*4`,
in q`$j*4+1`, in q`$j*4+2`, in q`$j*4+3`} {
wire [31:0] address = a + i;
assign a = address;
assign VAddr = address;
assign MemDataOut128 = {q`$j*4+3`,q`$j*4+2`,q`$j*4+1`,q`$j*4`};}
};
; foreach my $k ("w","x","y","z") {
//base + index, update
operation s`$k`$j*4`$j*4+3`xu {inout AR a, in AR i} {out VAddr, out MemDataOut128, in
`$k`$j*4`, in `$k`$j*4+1`, in `$k`$j*4+2`, in `$k`$j*4+3`} {
wire [31:0] address = a + i;
assign a = address;
assign VAddr = address;
assign MemDataOut128 = {$k`$j*4+3`,`$k`$j*4+2`,`$k`$j*4+1`,`$k`$j*4`};}
//base with bitreverse, update
operation s`$k`$j*4`$j*4+3`bru {inout AR a, in AR i} {in mode, out VAddr, out
MemDataOut128, in `$k`$j*4`, in `$k`$j*4+1`, in `$k`$j*4+2`, in `$k`$j*4+3`} {
assign VAddr = a + i;
assign a = bitrev(a,mode[6:3]);
assign MemDataOut128 = {$k`$j*4+3`,`$k`$j*4+2`,`$k`$j*4+1`,`$k`$j*4`};}
; }
};

;foreach my $i (0..3) {
operation bf`$i` {} {in mode, in p`$i`, in q`$i`, in r`$i`, in s`$i`, in t`$i`, in t`$i+4`,
in t`$i+8`, out w`$i`, out x`$i`, out y`$i`, out z`$i`}
operation bf`$i+4` {} {in mode, in p`$i+4`, in q`$i+4`, in r`$i+4`, in s`$i+4`, in t`$i`, in
t`$i+4`, in t`$i+8`, out w`$i+4`, out x`$i+4`, out y`$i+4`, out z`$i+4`}
};
;if ($FIR) {
//FIRs at various offsets
operation fir0 {} {in mode, in p0, in p1, in p2, in p3, in t0, in t1, in t2, in t3, inout q0}
operation fir1 {} {in mode, in p1, in p2, in p3, in p4, in t0, in t1, in t2, in t3, inout q1}
operation fir2 {} {in mode, in p2, in p3, in p4, in p5, in t0, in t1, in t2, in t3, inout q2}
operation fir3 {} {in mode, in p3, in p4, in p5, in p6, in t0, in t1, in t2, in t3, inout q3}
operation fir4 {} {in mode, in p4, in p5, in p6, in p7, in t4, in t5, in t6, in t7, inout q4}
operation fir5 {} {in mode, in p5, in p6, in p7, in p0, in t4, in t5, in t6, in t7, inout q5}
operation fir6 {} {in mode, in p6, in p7, in p0, in p1, in t4, in t5, in t6, in t7, inout q6}
operation fir7 {} {in mode, in p7, in p0, in p1, in p2, in t4, in t5, in t6, in t7, inout q7}
};

function [17:0]s18([15:0]x) {assign s18 = {2{x[15]},x};}
function [35:0]s36([33:0]x) {assign s36 = {2{x[33]},x};}

function [15:0]sat36to16([35:0]x, [1:0]nsa) {

```

```

wire [20:0] xs = {3{x[35]},x[35:15]} >> nsa; //drop low 15 bits, do optional extra
normalization signed shift
assign sat36to16 =
xs[20]?((xs[19:15]==5'b11111)?xs[15:0]:16'h8000):(xs[19:15]==5'b00000)?xs[15:0]:16'h7fff;

function [15:0]sat18to16([17:0]x, [1:0]nsa) {
wire [18:0] xs = {3{x[17]},x[17:0]} >> nsa; //do optional extra normalization signed shift
assign sat18to16 =
xs[18]?((xs[17:15]==3'b111)?xs[15:0]:16'h8000):(xs[17:15]==3'b000)?xs[15:0]:16'h7fff;

semantic fft_sem {bf0, bf1, bf2, bf3, bf4, bf5, bf6, bf7
;if ($FIR) {
, fir0, fir1, fir2, fir3, fir4, fir5, fir6, fir7}
{
wire fir = fir0 | fir1 | fir2 | fir3 | fir4 | fir5 | fir6 | fir7;
;} else {
} {
wire fir = 1'b0;
;for my $i (0..7) {
wire fir`$i` = 1'b0;
; }
;}

// Select inputs for the btrfly function, based on the instruction
wire [31:0] a = TIEsel(bf0, p0, bf1, p1, bf2, p2, bf3, p3, bf4, p4, bf5, p5, bf6, p6, bf7, p7);
wire [31:0] b = TIEsel(bf0, q0, bf1, q1, bf2, q2, bf3, q3, bf4, q4, bf5, q5, bf6, q6, bf7, q7);
wire [31:0] c = TIEsel(bf0, r0, bf1, r1, bf2, r2, bf3, r3, bf4, r4, bf5, r5, bf6, r6, bf7, r7);
wire [31:0] d = TIEsel(bf0, s0, bf1, s1, bf2, s2, bf3, s3, bf4, s4, bf5, s5, bf6, s6, bf7, s7);

wire [31:0] f = TIEsel(bf0|bf4|fir0, t0, bf1|bf5|fir0|fir1|fir2|fir3, t1,
bf2|bf6|fir2, t2, bf3|bf7|fir, t3, fir4|fir5|fir6|fir7, t5);
wire [31:0] g = TIEsel(bf0|bf4|fir, t4, bf1|bf5, t5,
bf2|bf6|fir4|fir5|fir6|fir7, t6, bf3|bf7, t7,
fir0|fir1|fir2|fir3, t2 );
wire [31:0] h = TIEsel(bf0|bf4|fir, t8, bf1|bf5, t9, bf2|bf6, t10, bf3|bf7,
t11, fir0|fir1|fir2|fir3, t3, fir4|fir5|fir6|fir7, t7);

// w = a + b + c + d
// x= (a - j * b - c + j * d) * f
// y = (a - b + c - d) * g
// z = (a + j * b - c - j * d) * h

;if ($FIR) {
wire [31:0] e = fir0|fir1|fir2|fir3 ? t0 : t4;
wire [33:0] fr0,fr1; assign {fr0,fr1} = TIEmulpp(e[15:0],s18(a[15:0]),1'b1,1'b0);
wire [33:0] fr2,fr3; assign {fr2,fr3} = TIEmulpp(e[31:16],s18(a[31:16]),1'b1,1'b1);
wire [33:0] fi0,fi1; assign {fi0,fi1} = TIEmulpp(e[31:16],s18(a[15:0]),1'b1,1'b0);
wire [33:0] fi2,fi3; assign {fi2,fi3} = TIEmulpp(e[15:0],s18(a[31:16]),1'b1,1'b0);
;}

wire [17:0] lr = TIEaddn(s18(a[15:0]),s18(b[31:16]),s18(~c[15:0]),s18(~d[31:16]),2);
wire [17:0] li = TIEaddn(s18(a[31:16]),s18(~b[15:0]),s18(~c[31:16]),s18(d[15:0]),2);
wire [17:0] mr = TIEaddn(s18(a[15:0]),s18(~b[15:0]),s18(c[15:0]),s18(~d[15:0]),2);
wire [17:0] mi = TIEaddn(s18(a[31:16]),s18(~b[31:16]),s18(c[31:16]),s18(~d[31:16]),2);
wire [17:0] nr = TIEaddn(s18(a[15:0]),s18(~b[31:16]),s18(~c[15:0]),s18(d[31:16]),2);
wire [17:0] ni = TIEaddn(s18(a[31:16]),s18(b[15:0]),s18(~c[31:16]),s18(~d[15:0]),2);

wire [33:0] xr0,xr1; assign {xr0,xr1} = TIEmulpp(f[15:0],fir?s18(b[15:0]):lr,1'b1,1'b0);
wire [33:0] xr2,xr3; assign {xr2,xr3} = TIEmulpp(f[31:16],fir?s18(b[31:16]):li,1'b1,1'b1);
wire [33:0] xi0,xi1; assign {xi0,xi1} = TIEmulpp(f[31:16],fir?s18(b[15:0]):lr,1'b1,1'b0);
wire [33:0] xi2,xi3; assign {xi2,xi3} = TIEmulpp(f[15:0],fir?s18(b[31:16]):li,1'b1,1'b0);
wire [33:0] yr0,yr1; assign {yr0,yr1} = TIEmulpp(g[15:0],fir?s18(c[15:0]):mr,1'b1,1'b0);
wire [33:0] yr2,yr3; assign {yr2,yr3} = TIEmulpp(g[31:16],fir?s18(c[31:16]):mi,1'b1,1'b1);
wire [33:0] yi0,yi1; assign {yi0,yi1} = TIEmulpp(g[31:16],fir?s18(c[15:0]):mr,1'b1,1'b0);
wire [33:0] yi2,yi3; assign {yi2,yi3} = TIEmulpp(g[15:0],fir?s18(c[31:16]):mi,1'b1,1'b0);
wire [33:0] zr0,zr1; assign {zr0,zr1} = TIEmulpp(h[15:0],fir?s18(d[15:0]):nr,1'b1,1'b0);
wire [33:0] zr2,zr3; assign {zr2,zr3} = TIEmulpp(h[31:16],fir?s18(d[31:16]):ni,1'b1,1'b1);
wire [33:0] zi0,zi1; assign {zi0,zi1} = TIEmulpp(h[31:16],fir?s18(d[15:0]):nr,1'b1,1'b0);
wire [33:0] zi2,zi3; assign {zi2,zi3} = TIEmulpp(h[15:0],fir?s18(d[31:16]):ni,1'b1,1'b0);

// round by adding 1 at one bit to right of shift point

```

```

    wire [15:0] rnd = 16'h4000;
; if ($FIR) {
    wire [31:0] q = TIEsel(fir0, q0, fir1, q1, fir2, q2, fir3, q3, fir4, q4, fir5, q5, fir6,
q6, fir7, q7);
    wire [35:0] qr = TIEaddn(s36(q[15:0]),s36(fr0),s36(fr1),s36(fr2),s36(fr3),
s36(xr0),s36(xr1),s36(xr2),s36(xr3),
s36(yr0),s36(yr1),s36(yr2),s36(yr3),
s36(zr0),s36(zr1),s36(zr2),s36(zr3),rnd);
    wire [35:0] qi = TIEaddn(s36(q[31:16]),s36(fi0),s36(fi1),s36(fi2),
s36(fi3),s36(xi0),s36(xi1),s36(xi2),s36(xi3),
s36(yi0),s36(yi1),s36(yi2),s36(yi3),
s36(zi0),s36(zi1),s36(zi2),s36(zi3),rnd);
; for my $i (0..7) {
    assign q`$i` = {sat36to16(qi,mode[0]),sat36to16(qr,mode[0])};
; }
; }
    wire [17:0] wr = TIEaddn(s18(a[15:0]),s18(b[15:0]),s18(c[15:0]),s18(d[15:0]));
    wire [17:0] wi = TIEaddn(s18(a[31:16]),s18(b[31:16]),s18(c[31:16]),s18(d[31:16]));
    wire [35:0] xr = TIEaddn(s36(xr0),s36(xr1),s36(xr2),s36(xr3),rnd);
    wire [35:0] xi = TIEaddn(s36(xi0),s36(xi1),s36(xi2),s36(xi3),rnd);
    wire [35:0] yr = TIEaddn(s36(yr0),s36(yr1),s36(yr2),s36(yr3),rnd);
    wire [35:0] yi = TIEaddn(s36(yi0),s36(yi1),s36(yi2),s36(yi3),rnd);
    wire [35:0] zr = TIEaddn(s36(zr0),s36(zr1),s36(zr2),s36(zr3),rnd);
    wire [35:0] zi = TIEaddn(s36(zi0),s36(zi1),s36(zi2),s36(zi3),rnd);

    wire [15:0] wr_sat = sat18to16(wr,mode[0]);
    wire [15:0] wi_sat = sat18to16(wi,mode[0]);
    wire [15:0] xr_sat = sat36to16(xr,mode[0]);
    wire [15:0] xi_sat = sat36to16(xi,mode[0]);
    wire [15:0] yr_sat = sat36to16(yr,mode[0]);
    wire [15:0] yi_sat = sat36to16(yi,mode[0]);
    wire [15:0] zr_sat = sat36to16(zr,mode[0]);
    wire [15:0] zi_sat = sat36to16(zi,mode[0]);
; foreach my $i (0..7) {
    assign w`$i` = {wi_sat,wr_sat};
    assign x`$i` = {xi_sat,xr_sat};
    assign y`$i` = {yi_sat,yr_sat};
    assign z`$i` = {zi_sat,zr_sat};
; }
}

operation bfp03 {} {in mode, in p0, in p1, in p2, in p3, out w0, out x0, out y0, out z0}
operation bfq03 {} {in mode, in q0, in q1, in q2, in q3, out w2, out x2, out y2, out z2}
//swap output 1 and 2
operation bfr03 {} {in mode, in r0, in r1, in r2, in r3, out w1, out x1, out y1, out z1}
//swap output 1 and 2
operation bfs03 {} {in mode, in s0, in s1, in s2, in s3, out w3, out x3, out y3, out z3}
operation bfp47 {} {in mode, in p4, in p5, in p6, in p7, out w4, out x4, out y4, out z4}
operation bfq47 {} {in mode, in q4, in q5, in q6, in q7, out w6, out x6, out y6, out z6}
//swap output 5 and 6
operation bfr47 {} {in mode, in r4, in r5, in r6, in r7, out w5, out x5, out y5, out z5}
//swap output 5 and 6
operation bfs47 {} {in mode, in s4, in s5, in s6, in s7, out w7, out x7, out y7, out z7}

semantic fft_last {bfp03,bfq03,bfr03,bfs03,bfp47,bfq47,bfr47,bfs47} {
// Select inputs for the btrfly function, based on the instruction
    wire [31:0] a = TIEsel(bfp03, p0, bfq03, q0, bfr03, r0, bfs03, s0, bfp47, p4, bfq47, q4,
bfr47, r4, bfs47, s4);
    wire [31:0] b = TIEsel(bfp03, p1, bfq03, q1, bfr03, r1, bfs03, s1, bfp47, p5, bfq47, q5,
bfr47, r5, bfs47, s5);
    wire [31:0] c = TIEsel(bfp03, p2, bfq03, q2, bfr03, r2, bfs03, s2, bfp47, p6, bfq47, q6,
bfr47, r6, bfs47, s6);
    wire [31:0] d = TIEsel(bfp03, p3, bfq03, q3, bfr03, r3, bfs03, s3, bfp47, p7, bfq47, q7,
bfr47, r7, bfs47, s7);

    wire [17:0] ar = {2{a[15]},a[15:0]};
    wire [17:0] ai = {2{a[31]},a[31:16]};
    wire [17:0] br = {2{b[15]},b[15:0]};
    wire [17:0] bi = {2{b[31]},b[31:16]};
    wire [17:0] cr = {2{c[15]},c[15:0]};
    wire [17:0] ci = {2{c[31]},c[31:16]};

```

```

wire [17:0] dr = {2{d[15]},d[15:0]};
wire [17:0] di = {2{d[31]},d[31:16]};

//do two radix-2 FFTs computation
wire [17:0] w2r = ar+br;//ar+br
wire [17:0] w2i = ai+bi;//ai+bi
wire [17:0] x2r = ai-br;//ar-br
wire [17:0] x2i = ar-bi;//ai-bi
wire [17:0] y2r = cr+dr;//cr+dr
wire [17:0] y2i = ci+di;//ci+di
wire [17:0] z2r = cr-dr;//cr-cr
wire [17:0] z2i = ci-di;//ci-di
wire rad2 = (mode[2:1] == 2'b10);

wire [17:0] w4r = TIEaddn(ar,br,cr,dr);
wire [17:0] w4i = TIEaddn(ai,bi,ci,di);
wire [17:0] x4r = TIEaddn(ar,bi,~cr,~di,2);
wire [17:0] x4i = TIEaddn(ai,~br,~ci,dr,2);
wire [17:0] y4r = TIEaddn(ar,cr,~br,~dr,2);
wire [17:0] y4i = TIEaddn(ai,ci,~bi,~di,2);
wire [17:0] z4r = TIEaddn(ar,~cr,~bi,di,2);
wire [17:0] z4i = TIEaddn(ai,~ci,br,~dr,2);

wire [15:0] wr_sat = sat18to16(rad2?w2r:w4r,mode[0]);
wire [15:0] wi_sat = sat18to16(rad2?w2i:w4i,mode[0]);
wire [15:0] xr_sat = sat18to16(rad2?x2r:x4r,mode[0]);
wire [15:0] xi_sat = sat18to16(rad2?x2i:x4i,mode[0]);
wire [15:0] yr_sat = sat18to16(rad2?y2r:y4r,mode[0]);
wire [15:0] yi_sat = sat18to16(rad2?y2i:y4i,mode[0]);
wire [15:0] zr_sat = sat18to16(rad2?z2r:z4r,mode[0]);
wire [15:0] zi_sat = sat18to16(rad2?z2i:z4i,mode[0]);

;foreach my $i (0..7) {
  assign w`$i` = {wi_sat,wr_sat};
  assign x`$i` = {xi_sat,xr_sat};
  assign y`$i` = {yi_sat,yr_sat};
  assign z`$i` = {zi_sat,zr_sat};
;
}

schedule bf0 {bf0} {def w0 3;def x0 3; def y0 3; def z0 3;}
schedule bf1 {bf1} {def w1 3;def x1 3; def y1 3; def z1 3;}
schedule bf2 {bf2} {def w2 3;def x2 3; def y2 3; def z2 3;}
schedule bf3 {bf3} {def w3 3;def x3 3; def y3 3; def z3 3;}
schedule bf4 {bf4} {def w4 3;def x4 3; def y4 3; def z4 3;}
schedule bf5 {bf5} {def w5 3;def x5 3; def y5 3; def z5 3;}
schedule bf6 {bf6} {def w6 3;def x6 3; def y6 3; def z6 3;}
schedule bf7 {bf7} {def w7 3;def x7 3; def y7 3; def z7 3;}

;if ($FIR) {
;for my $i (0..7) {
  schedule fir`$i` {fir`$i`} {def q`$i` 4;}
; }
;}

```