RACH Preamble Detection

Appendix to 3G Baseband Chip-Rate Processing Using the Intrinsity™ FastMATH™ Processor

1. Overview

Three methods have been considered to perform WCDMA RACH preamble detection:

1. The brute force, direct method that is currently described in our chip-rate white paper [2]. This is NOT the most efficient method to solve the problem.
2. "Despread" the Rx sequence using the 4096 chip scrambling code in (interleaved) 256 chip chunks, and subsequently use FHT to "uncover" the orthogonal Hadamard codes used in the preamble.
3. Use FFTs to implement the searcher. The analysis with respect to this method is still in progress. This method is expected to be efficient when the cell radius over which the search is to be performed is very large.

This document describes how the "the FHT method" will be implemented using the FastMATH processor. It is estimated that approximately 0.08 FastMATH processors (2 GHz) can be used to meet the real-time requirements of the access searcher task for a 20 km cell site that supports the access of up to 16 UEs in each RACH access slot.

Offset frequency estimation and correction based on the received RACH preamble is NOT part of the current analysis.

2. Brief Overview of the RACH Preamble Detector Design

The 3GPP specification provides the formula used by the mobile to construct and to transmit the RACH preamble. A TSG-RAN working group proposal [1] by Motorola/TI describes one possible structure of the RACH preamble detector. The basic principle of this design is to first perform the "despreading" operation on the Rx sequence using the 4096 chip scrambling code sequence in (interleaved) 256 chip chunks, and subsequently use the fast Hadamard transform (FHT) to "uncover" the (orthogonal Hadamard code) preamble signature. Figure 1 shows the block diagram of the preamble coherent receiver using FHT. The corresponding designs for a differential receiver or a non-coherent segmented receiver are very similar with respect to the processing requirements.

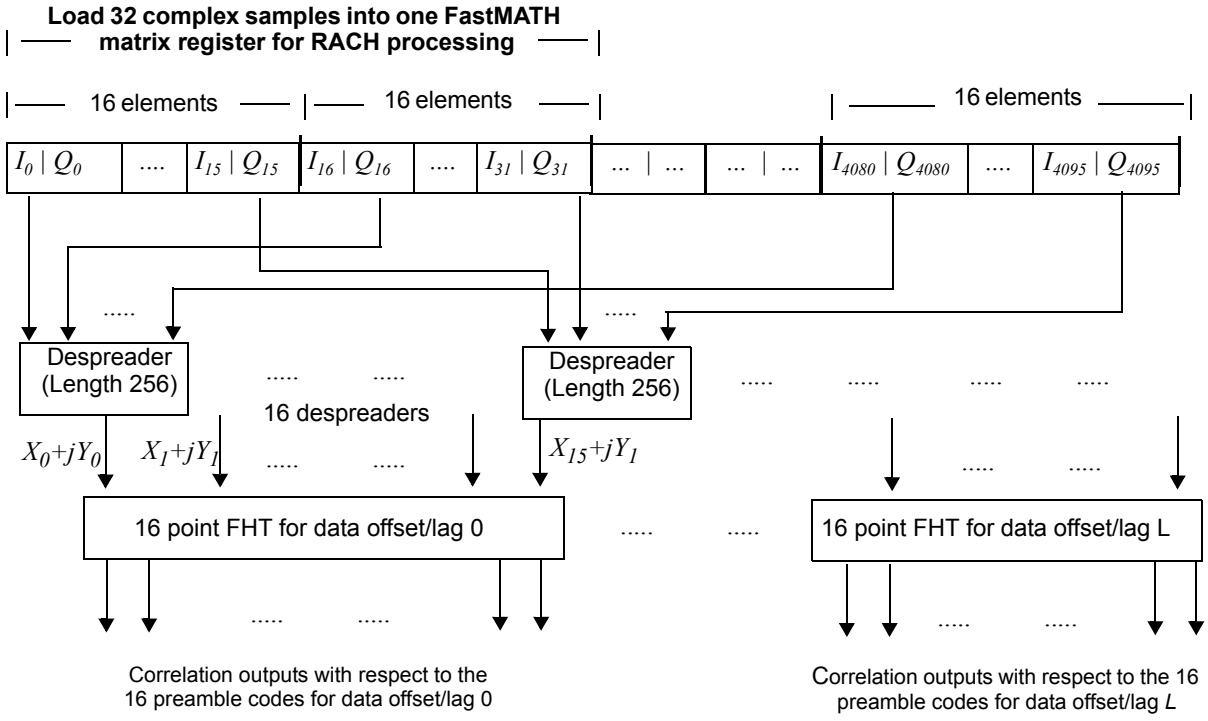

Figure 1: Block diagram for preamble coherent receiver using FHT

The 256 length despreading operation is implemented by using an element by element multiplication of the input data sequence (I, Q) with the long (4,096 chip) scrambling code sequence, and subsequently summing the 256 elements corresponding to a given symbol of the preamble signature (Hadamard code). As shown in Figure 1, this involves deinterleaving all the 256 elements required to perform the sum. Finally, the FHT is used to efficiently obtain the correlation output with respect to the 16 preamble codes. As shown in Figure 1, this process is repeated for all offsets or lags (L) between the input data sequence and the scrambling code sequence. For a 20 km cell radius, L = 512 chips.

The complex scrambling code used in the despread operation is $S_k e^{j\left(\frac{\pi}{4}+\frac{\pi}{2}k\right)}$, where S_k is real. Thus, we note that the despread operation of the complex sequence I_k+jQ_k requires the computation of only two product terms S_kI_k and S_kQ_k (instead of four products usually required in a complex multiply). The real (X_k) and imaginary (Y_k) terms of the output despread symbols are finally obtained by appropriately adding or subtracting the S_kI_k and S_kQ_k terms (based on the phase determined by k). Correspondingly, we note that the same FHT operation is applied to both the real (X_k) and imaginary (Y_k) components of the despread symbols to produce the complex correlation output.

3. Implementation Using the FastMATH Processor

This section briefly describes the FastMATH implementation of the despreading and the FHT operations required to implement the RACH preamble detector shown in Figure 1. The salient features of this efficient FastMATH implementation are:

- The matrix coprocessor is used very efficiently to perform 64 simultaneous byte operations (such as 8-bit XORs and adds) arising in despreading.

- The special deinterleaving (Figure 1) required to implement the despreading operation is very efficiently implemented, since the matrix coprocessor naturally supports vectors of length 16 (and multiples of 16).
- The matrix coprocessor is used very efficiently to perform 32 simultaneous 16-bit operations (such as halfword adds) that occur in the FHT block.

All the implementation specific assumptions are described in the chip-rate paper [2]. Additionally, note the following assumptions relevant to the FastMATH implementation of the RACH preamble detector.

- The input data sequence is fed to the searcher at the chip-rate (3.84 Mchips per second). Thus, the resolution of the access search is assumed to be no better than one chip
- The input data is assumed to be a sequence of (interleaved) complex bytes, which contains only six bits of significant data each for *I* and *Q*. The output data is assumed to be complex bytes
- The searcher processes the input data sequence for one slot, within a time period of one slot. This provides the real-time requirement for the searcher.

Figure 2 shows the steps involved in the FastMATH implementation of the despreading operation. This implementation uses the XOR method described in the chip-rate paper [2] to replace the multiplications (by ± 1) with XOR operations (by using the XOR-based relations $a \times 1 = a \wedge 0$ and $a \times -1 = (a \wedge -1) + 1$). As shown in Figure 1, the 4,096 sample input sequence (corresponding to one access slot) is processed by working on 32 complex elements at a time. This allows the FastMATH implementation to easily handle the special deinterleaving (choose every 16th element of the input data sequence, and perform operations between these elements) of data required to implement the despreading operation.

The input data sequence (32 complex samples) and the long scrambling code sequence are each loaded into a separate matrix register, and an element-wise XOR operation is implemented between the two matrix registers. This is performed 4,096/32 times. Subsequently, the resulting matrices are added together using the `matrixadd` and `matrixaddhw` instructions, while taking special care to avoid overflow errors (after a series of two adds, the 8-bit data register is unpacked into two registers containing 16-bit data so as to preserve accuracy during subsequent operations). Thus, we obtain one matrix register that contains the "*I*-terms" and a register that contains the "*Q*-terms". The first 16-bit element of the "*I*-terms" register contains $S_0I_0+S_{32}I_{32}+...$, the second element contains $S_1I_1+S_{33}I_{33}+...$, the sixteenth element contains $S_{16}I_{16}+S_{48}I_{48}+...$, and the last (thirty-second) element contains $S_{32}I_{32}+S_{64}I_{64}+....$ Correspondingly, the "*Q*-terms" register contains elements with *I* is replaced by *Q*. The real and imaginary terms of the despread symbols are obtained by appropriately adding or subtracting the *I*-terms and *Q*-terms. The real and imaginary despread symbols are then interleaved using the FastMATH `block2` instruction to produce two matrix registers, where the first register contains $S_0I_0+S_{32}I_{32}+...$, $S_1I_1+S_{33}I_{33}+...$, to $S_{15}I_{15}+S_{47}I_{47}+...$, and the second register contains $S_{16}I_{16}+S_{48}I_{48}+...$, $S_{17}I_{17}+S_{49}I_{49}+...$, to $S_{32}I_{32}+S_{64}I_{64}+....$ Thus, the 256 length despread symbols are finally obtained by adding together these two matrix registers that contain the separate 128 length results.

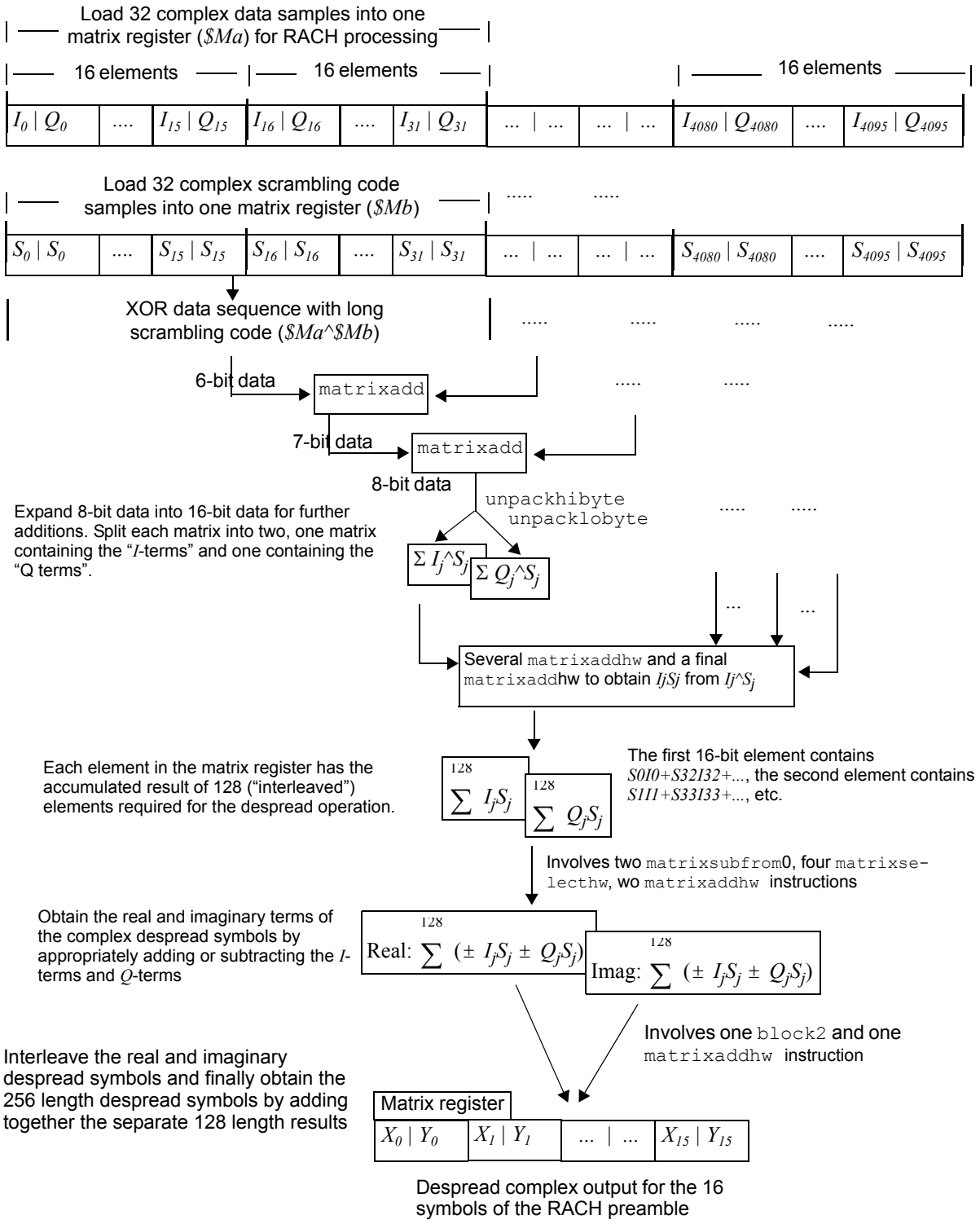

Figure 2: FastMATH implementation of the despreading operation for the FHT-based coherent RACH preamble detector

Next, we describe the FastMATH implementation of the 16-point FHT. This FastMATH implementation is most efficient while operating on 32 independent FHTs (or 16 independent complex FHTs) in parallel. In the case of the RACH preamble detector, this corresponds to performing the complex FHTs for 16 offsets/lags in parallel. The input data is 16 consecutive blocks of 16 complex values (16-bit real, imaginary packed into a 32-bit word). Thus, each block fits in a matrix register. The `fht16()` routine performs 16 fast Hadamard transforms in four basic steps:

First, four matrix registers are loaded from four consecutive blocks:

```
M0 = |a0 b0 c0 d0 |    M1 = |a1 b1 c1 d1 |
     |e0 f0 g0 h0 |          |e1 f1 g1 h1 |
     |i0 j0 k0 l0 |          |i1 j1 k1 l1 |
     |m0 n0 o0 p0 |          |m1 n1 o1 p1 |  ... for M2, M3
```

Next, perform a `block4` operation to interleave the rows of the four matrices:

```
M0 = |a0 b0 c0 d0 |    M1 = |e0 f0 g0 h0 |
     |a1 b1 c1 d1 |          |e1 f1 g1 h1 |
     |a2 b2 c2 d2 |          |e2 f2 g2 h2 |
     |a3 b3 c3 d3 |          |e3 f3 g3 h3 |  ... for M2, M3
```

Now, we perform four matrix adds and four matrix subtracts between M0, M1, M2, and M3 which execute the first two stages of the FHT butterfly (Figure 3) on four independent blocks:

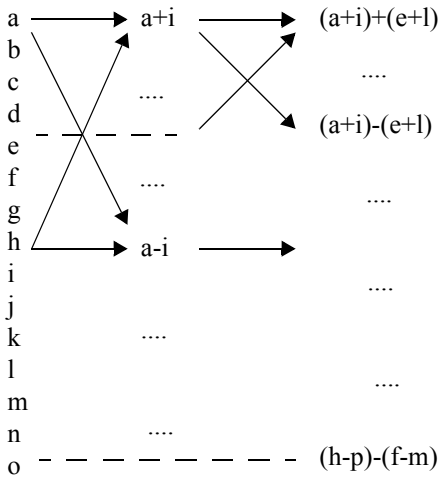

Figure 3: Butterfly operation required in the FHT

Now, we need to perform butterflies on the column data of the butterfly results we just generated. We perform a `block4v` to interleave the column data among four matrix registers and perform the butterfly operation again. At the end of this operation (the last two stages of FHT), the data in the four matrix registers is:

```
M0 = |H0[0] H4[0] H8[0] H12[0]|    M1 = |H1[0] H5[0] H9[0] H13[0]|
     |H0[1] H4[1] H8[1] H12[1]|          |H1[1] H5[1] H9[1] H13[1]|
     |H0[2] H4[2] H8[2] H12[2]|          |H1[2] H5[2] H9[2] H13[2]|
     |H0[3] H4[3] H8[3] H12[3]|          [H1[3] H5[3] H9[3] H13[3]|
```

etc., where `Hn[m]` is the value of the nth sequence of the nth Hadamard transform.

This entire sequence is repeated four times, and is followed by a final pass, which uses `block4v` to interleave on columns to collect all `Hn` sequence values from each of the 16 blocks together to obtain:

```
M0 = |H0[0] H0[4] H0[8]  H0[12]|
     |H0[1] H0[5] H0[9]  H0[13]|
     |H0[2] H0[6] H0[10] H0[14]|
     |H0[3] H0[7] H0[11] H0[15]|
```

4. Summary of Results

Table 1 summarizes the *estimated* performance of the FastMATH processor for RACH preamble detection.

Table 1: RACH preamble detection--assumptions and estimated results

	Assumed or Calculated Value	Comments
Input sample rate	3.84 Mchips per second	Chip-rate
Data type of input samples	6-bit complex	
Number of RACH scrambling codes in use per cell	1	
Maximum data offsets per lags	L = 512 chips	Cell radius of 20 km
Number of UEs to be searched per access-slot per cell	U = 16	16 preamble signatures
Number of samples per slot	N = 4096	1 sample per chip is assumed
Despreading operation	0.2048 Mcycles	Sum of 3 items listed below
XOR and add between the input data sequence and scrambling code	$10 \times (N/128) \times L$ cycles	4 loads, 4 XORs, 3 adds, 2 unpacks for every 128 input samples
Accumulate partial sums	$2 \times ((N/128) - 1) \times L$ cycles	2 `addhw` per 128 samples
Final accumulation of 16 despread symbols	$18 \times L$ cycles	2 `addhw`, 2 subfromzero, 4 selecthw, 2 addhw, 1 `block2` + 1 final addhw, 1 store
FHT operation	6400 cycles	Sum of 1 item listed below.
16-point FHT	$200 \times (L/16)$ cycles	Based on FastMATH cycle accurate simulation, we estimate 200 cycles per 16 lags
Estimated number of cycles required to search 1 access slot	0.2112 Mcycles per slot	Sum of despread and FHT operations
Processing time *estimate* per slot at 2 GHz	0.11 msec	(Cycles per slot) / (2 Gcycles per second)
Latency allowed for computation	1.33 msec	Corresponds to 1 access slot (5,120 chips).
Fractional usage of one 2 GHz FastMATH chip	0.08	(Processing time per slot) / (Latency allowed for computation)

As summarized in Table 1, approximately 0.08 (2 GHz) FastMATH processors can be used to meet the real-time requirements of the access searcher task for a 20 km cell site that supports the access of up to 16 UEs in each RACH access slot.

5. References

[1] "Proposal for RACH Preambles", Motorola/TI, TSG-RAN Working Group 1 meeting #6, Document number 3GPP/TSGR1#6(99)893, July 13-16, 1999.

[2] "3G Baseband Chip-Rate Processing Using the Intrinsity™ FastMATH™ Processor," Intrinsity White Paper, July 2002.

6. Revision History

Revision	Date	Description
1.0	11/15/02	Initial publication
1.1	3/26/03	New format and logo with tag line