3G Baseband Symbol-Rate Processing Using the Intrinsity™ FastMATH™ Processor

1. Overview

This document presents an evaluation of the use of the Intrinsity™ FastMATH™ processor in the symbol-rate baseband processing of a typical 3G base station receiver system. Even though many portions of this document may be generally applicable to CDMA-based systems, it specifically focuses on the 3GPP, UTRA-FDD specification [1, 2]. 3GPP baseband receiver processing for voice users encompasses the Viterbi decoder, a deinterleaver, rate matching, and turbo encoder and decoder for data users. This evaluation identifies and focuses on those blocks in the 3GPP system that require a lot of processing bandwidth and that would greatly benefit from the implementation flexibility or programmability that a FastMATH solution offers. The analysis presented in this document is approximate and is based on the design assumptions stated in the relevant sections.

Based on the results presented in this document, we estimate that the processing for approximately 630 voice users or 20 data users (at 384 kbps) can be supported using one FastMATH processor running at 2 GHz. Thus, for a typical 3GPP base station supporting 64 voice users, or about three high-speed data users, the FastMATH solution is expected to implement symbol-rate processing and other tasks such as multi-user detection and/or smart antenna processing. Additionally, such a software solution using the FastMATH processor provides an easy migration path via software upgrades as standards and receiver technology evolve.

2. Overview of the FastMATH Processor

This section provides a very brief overview of the FastMATH processor. The FastMATH processor is a high-performance processor designed to handle problems that require large amounts of vector- and matrix-based computation. It consists of:

- A MIPS32™ processor core with 16-Kbyte instruction and data caches
- A matrix and vector math unit: a 4 × 4 array of 32-bit processing elements interconnected in a row/column mesh
- An integrated 1-Mbyte L2 cache
- Two bidirectional RapidIO™ ports
- An integrated 64-bit DDR SDRAM controller

The FastMATH processor executes the standard MIPS32 instruction set on a scalar processor, along with a set of vector and matrix instruction set extensions implemented through the MIPS® coprocessor 2 interface. It can issue and execute a core (scalar) instruction and a matrix coprocessor instruction every cycle.

The matrix unit is a 4 × 4 array of 32-bit processing elements. Each element consists of a 16-entry, 32-bit register file, a 32-bit, single-cycle ALU that can perform 32-bit and paired 16-bit single instruction, multiple data (SIMD) operations, and a 16-bit × 16-bit multiplier with two 40-bit accumulators. The elements are connected in a row/column mesh with the ability to broadcast to and receive values from the other elements in the same row or column.

A matrix coprocessor instruction is executed by all elements of the matrix unit in a parallel, SIMD fashion to provide a peak execution rate of 64 GOPS (e.g., 16 parallel multiply-accumulates at 2 GHz). In addition to standard parallel operations like addition and multiplication, the matrix unit implements instructions that can exploit the two-dimen-

sional matrix interconnections, such as matrix-multiply, transpose, and block rearrangement. Data is supplied to the matrix unit via a 64-byte wide direct connection to the integrated L2 cache.

3. 3GPP Base Station Architecture Overview

The 3GPP physical-layer specification [2, 4-8] provides a very detailed description of the baseband processing requirements of a 3GPP base station transmitter and user equipment (UE), signifying mobile users, transmitter working in FDD mode. Figure 1 illustrates the overall design of a 3GPP base station receiver. This figure shows some of the important functional blocks required in the baseband receiver. In the interest of space, it does NOT explicitly show ALL the processing blocks of the baseband receiver. The baseband processing can be broadly divided into the so-called "symbol-rate processing" and "chip-rate processing". As the name suggests, chip-rate processing involves the processing of the input data at the chip-rate or a multiple of the chip-rate. In this document, we concentrate on the symbol-rate processing blocks, which follow the chip-rate blocks.

Figure 1 assumes that there is one antenna servicing U UEs for a given user i having H_i channels. It is assumed that there is a mix of voice and data users. The symbol-rate processing for voice and data users is somewhat different and is described in this document. Our first-order analysis shows that among the symbol-rate processing blocks the turbo decoder (used for data users) requires the most processing bandwidth. Hence, a detailed analysis of the FastMATH implementation of the turbo decoder is provided in Section 4.1. The remainder of this document discusses (in lesser detail) other symbol-rate blocks of the transmitter and receiver. In particular, we discuss the FastMATH implementation of the two interleaver blocks, the corresponding deinterleavers, the rate-matching block (and rate-dematching), convolutional encoder, Viterbi decoder, and the turbo encoder. A brief discussion about the FastMATH implementation of generic bit and byte interleaving schemes (Section 4.5) is provided, as this also applies to other symbol-rate processing blocks such as TrCh multiplexing and DTX insertion.

4. Baseband Processing Using the FastMATH Processor

In this section, we evaluate the design and FastMATH implementation of some of the symbol-rate processing blocks that arises in a typical 3GPP base station. The analysis presented in this document is approximate as our design assumptions and code have *not* yet been verified using WCDMA system-level simulators. All the relevant design assumptions are explicitly listed in each subsection.

We estimate the performance (number of cycles) of the FastMATH implementation for functional blocks assuming that the FastMATH processor is running at a clock speed of 2 GHz. Our estimate of the number of cycles for the different functional blocks is based on a detailed analysis of the required code. However, these estimates have not yet been verified with the FastMATH cycle accurate simulator.

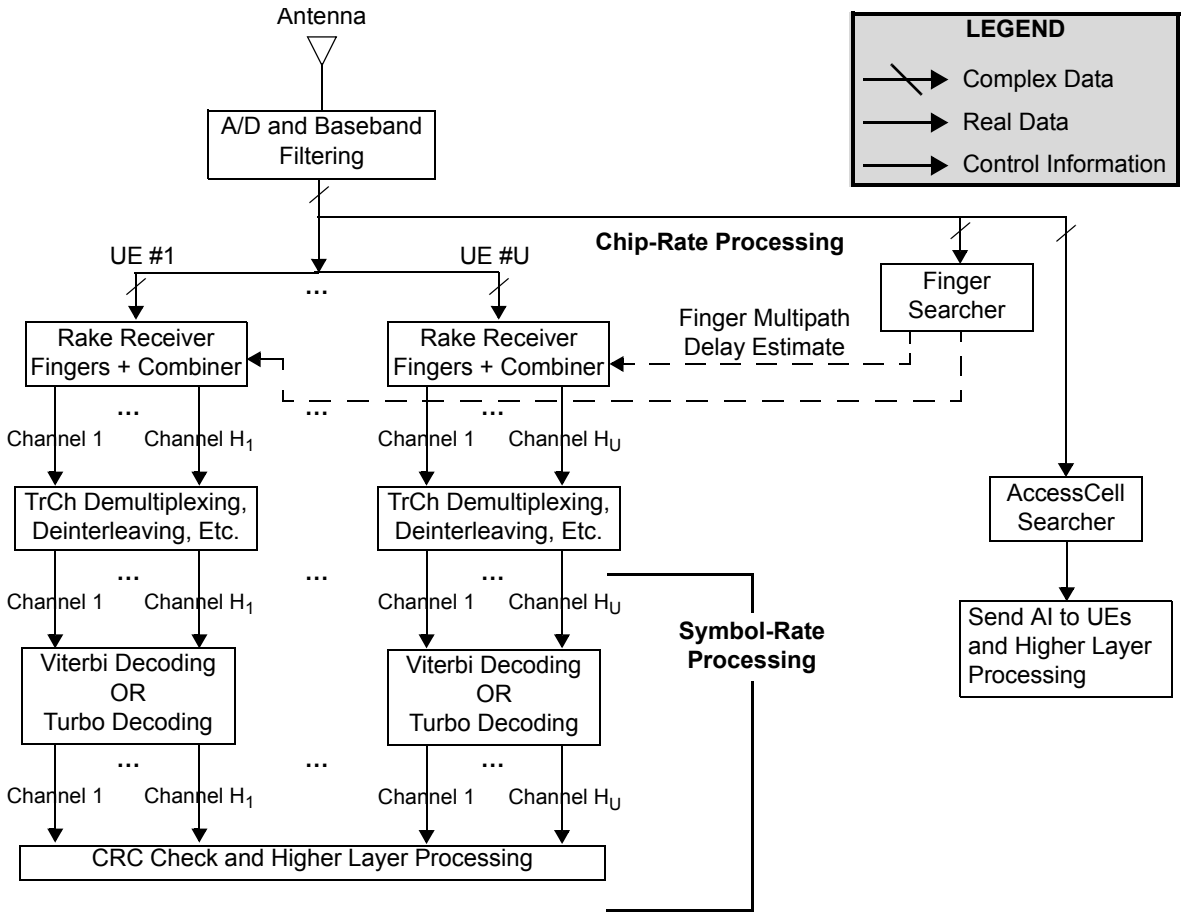

Figure 1: Overall design of a 3GPP base station receiver

4.1. Turbo Decoder

This section provides the high-level design of the turbo decoder block of a typical 3GPP base station and evaluates the computational requirements of implementing this block using the FastMATH processor.

4.1.1. Design

4.1.1.1. Turbo encoder:

The 3GPP turbo encoder [7], is based on a parallel concatenated convolutional code (PCCC), which is comprised of two parallel, eight-state constituent encoders and an internal turbo code interleaver. The coding rate of the turbo encoder is 1/3. The transfer function of the eight-state constituent code is:

$$G(D) = \left[1, \frac{g_1(D)}{g_0(D)}\right], \qquad \textit{Eq.(1)}$$

where: $g_0(D) = 1 + D^2 + D^3$, $g_1(D) = 1 + D + D^3$, and D is the delay.

Figure 2 below shows the structure of the rate 1/3 turbo coder described in the 3GPP spec [7]. The dashed lines apply for trellis termination.

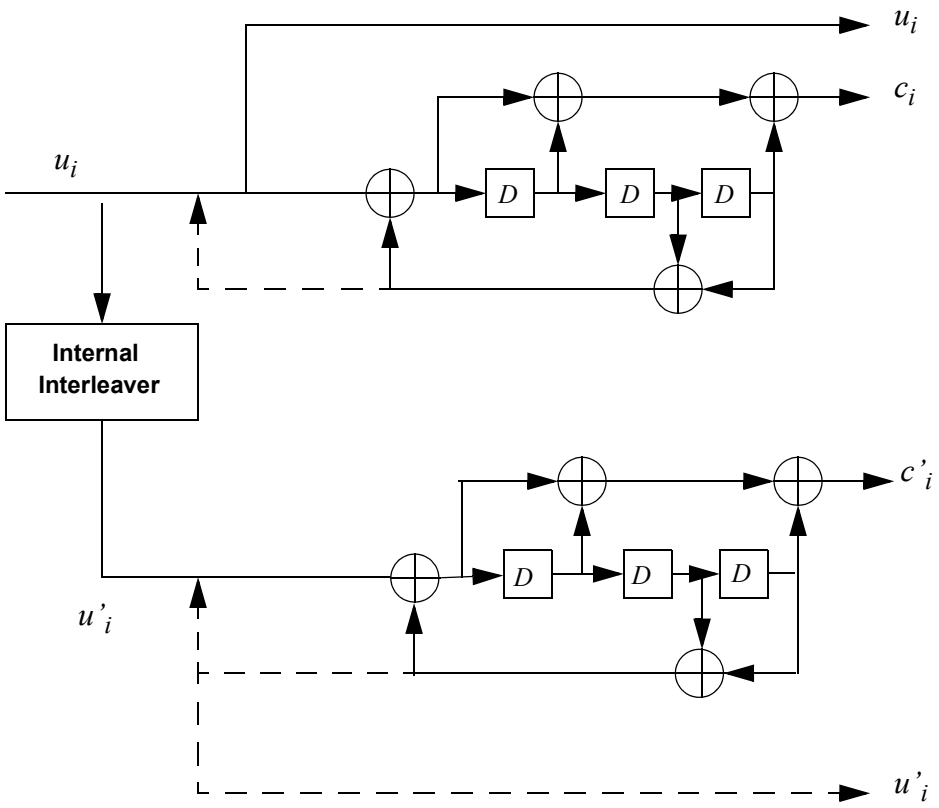

Figure 2: Structure of the 3GPP rate 1/3 turbo coder

The output of the turbo encoder is:

`u[1],c[1],c'[1],u[2],c[2],c'[2],..,u[k],c[k],c'[k]` ,

where *k* is the number of bits of the frame or the frame length, *u* is the array of input (systematic) bits, and *c* and *c'* are the arrays of output (parity or coded) bits from the first and second eight-state constituent encoders, respectively.

4.1.1.2. Turbo decoder

The turbo decoder consists of two blocks. Each block is made of a decoder and an interleaver. Each block corresponds to one of the two encoders. The decoder of a given block receives three input streams: the a priori stream of information from the previous block, the parity bit stream associated with the encoder corresponding to the decoding block, and the systematic encoded channel output [11].

Figure 3 shows that the turbo decoder is iterative. The first maximum a posteriori (MAP) decoder computes an extrinsic log likelihood ratio that is the a priori information for the second MAP decoder. The a priori information is first interleaved to match the interleaved data of the parity bit information. The interleaved parity information corresponds to the second encoder (Figure 2). The extrinsic information from the second decoder is deinterleaved and then used as the a priori information for the first decoder. The process is iterated for a certain number of times until a satisfactory BER/SNR level is achieved or a predetermined number of iterations have been executed.

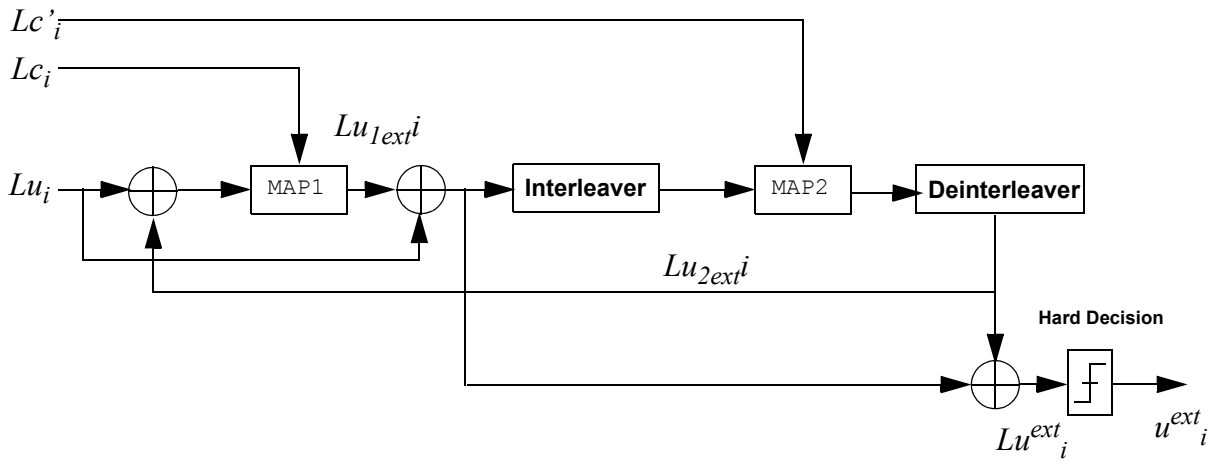

Figure 3: Turbo decoder system

The inputs to the system (*Lu[i],Lc[i],Lc'[i]*) are probability distributions of the systematic and parity bits in a frame. The probability distributions can be conveniently represented as log-likelihood ratios (LLRs):

$$Lu[i] = \log\frac{Pr\{u[i]=1\}}{Pr\{u[i]=0\}}, \qquad Eq.(2)$$

where *Pr{u[i]}* is the probability distribution of *u[i]*.

The LLR of what is called the channel LLR of *u[i]* is calculated using the channel model, which contains additive Gaussian white noise and the broadcast bits *u[i],* and assumes that the frame is broadcast over a fading channel. *Lu[k]* is then calculated by adding the output of the previous MAP decoder ($Lu_{ext}[k]$). If this is the first iteration of the decoder, then the additive output is set to zero.

4.1.1.3. Max log MAP decoder

The main goal of the MAP decoder is to increase the reliability with which the maximum LLR is able to correctly identify a systematic bit *Lu[k]*. The LLR is improved if used in an iterative process as was discussed above. The MAP decoder block used is based on the max-log-MAP algorithm derived from the BCJR algorithm [11].

The derivation of the max-log-MAP algorithm has been reported in the literature [11]. What follows is a summary of the four main steps of the max-log-MAP algorithm.

Doing math in the log-arithmetic domain simplifies the calculations required for the BCJR with the approximation (Eq.(3)), which is from [11]. This is the basis of the max-log-MAP advantage.

$$\ln\left(\sum_i e^{x_i}\right) \approx max(x_i). \qquad Eq.(3)$$

The first step computes the branch metric log-γ (Eq.(4)):

$$\log(\gamma_i(l,l')) = Lu_i(u(l,l')+0.5) + Lc_i(c(l,l')-0.5), \qquad Eq.(4)$$

where *i* is the stage index, *l* is the state index at stage *i*-1, and *l'* is the state index at time *i*. The pair (*u(l,l'), c(l,l')*) can take the values (0,0), (0,1), (1,0), (1,1). The term Lu_i is the sum of the systematic information and the a priori information entering the decoder.

The forward state metric log-α is computed using:

$$\log(\alpha_i(l)) = max_{l'}(\log(\gamma_i(l,l')) + \log(\alpha_{(i-1)}(l'))), \qquad Eq.(5)$$

where *l* is the current state at time *i* and *l'* takes the value of the two states at time *i*-1 which connect to state *l* at time *i* (see the trellis diagram).

The backward state metric log-β (Eq.(6)) is computed as follows:

$$\log(\beta_i(l')) = max_l(\log(\gamma_{(i+1)}(l, l')) + \log(\beta_{(i+1)}(l))) \quad , \qquad Eq.(6)$$

where *i*, *l* and *l'* have the same meaning as for the log-α equation.

The extrinsic computation (Eq.(7)):

$$Lu_{i+1}^{ext} = max_{(l,l')}(\log(\alpha_i(l')) + \log(\gamma_{(i+1)}(l, l')) + \log(\beta_{(i+1)}(l))) - max_{(k,k')}(\log(\alpha_i(k')) + \log(\gamma_{(i+1)}(k, k')) + \log(\beta_{(i+1)}(k))) - Lu_i \quad , \qquad Eq.(7)$$

where (*l, l'*) is the pair of indices at times *i*-1 and *i,* respectively, that define the path leading to u_i equal 1, and *(k, k')* is the pair of indices of the states at times *i* -1 and *i*, respectively, that define the paths leading to u_i equal 0. The term Lu_i is the sum of the systematic and a priori information entered in the decoder.

The beginning and end of the frame being decoded needs to have initialized α and β states, respectively. The initial α at the front of the frame, at time 0, is:

log(α_0(0)) = 0,

log(α_0(l)) = -infinity, for all states l not equal to 0.

Similarly, the final times' state to compute backward the log-β is at the final time of the frame, time *N:*

l*og(β_N(0)) = 0,*

log(β_N(l)) = -infinity, for all states l not equal to 0.

4.1.2. Implementation Using the FastMATH Processor

4.1.2.1. MAP decoder

4.1.2.2. Parameters

The inputs to the MAP decoder are:

- One parity LLR, *Lc[k]*, which is an array of eight-bit fixed point (5,2) signed integers. The size of the array is equivalent to the length of the frame.
- One systematic LLR, *Lu[k]*, which is an array eight-bit fixed point (5,2) signed integers. It is the combined channel LLR of the raw input stream added to the a priori output of the previous MAP decoder. The size of the array is equivalent to the length of the frame.
- The frame length is anywhere from 40 to 5114 symbols in length [7].
- The γ, α, and β intermediate arrays are of 32-bit fixed point (29,2) signed integers.

The output of the MAP decoder is called the extrinsic LLR, $Lu_{ext}[k]$, and is an array of eight-bit fixed point (5,2), signed integers that is equivalent in size to the length of the frame.

4.1.2.2.1. Frame parallelization

The MAP decoder receives two frames of input symbols: the channel LLR for the systematic bits *Lu[k],* and the channel LLR for the parity bits *Lc[k].* As can be gathered from the functional description, the algorithm is highly iterative. It is also highly recursive, with the α and β stages being forward and backward recursive, respectively. With frame sizes running from 40 to 5,114 symbols, a very large register system would be needed to hold the entire frame's worth of α and β data.

Fortunately, it is possible to break up the frames into smaller sub-frame (window) sizes using the sliding window technique [12]. The windows are of length $WL = R + 2 \times P$, where *WL* is window length, *R* is the reliability size and *P* is the prolog length. *R* is the length of the actual data being processed and *P* is the length of the initialization data for α and β processing on each window. This approach exploits the SIMD nature of the FastMATH processor, enabling maximum usage of all matrix elements by processing up to 16 windows of data in parallel, since the processing is

highly iterative and the recursive dependencies are broken up into smaller windows. The current system design assumes full use of all 16 matrix elements, so the frame is parallelized into 16 windows, giving $R = F/16$, where F is the frame length (Figure 4). Essentially this means that there will be 16 independent frames being processed through 16 SIMD elements. Each SIMD element can operate on 32-bit or 16-bit data types. 32-bit data types were chosen for the γ, α, and β arrays due to overflow issues.

The two input frames (*Lu[k],Lc[k])* each are made up of *F* number of eight-bit soft-coded symbols, which are in signed, fixed-point (5,2) format.

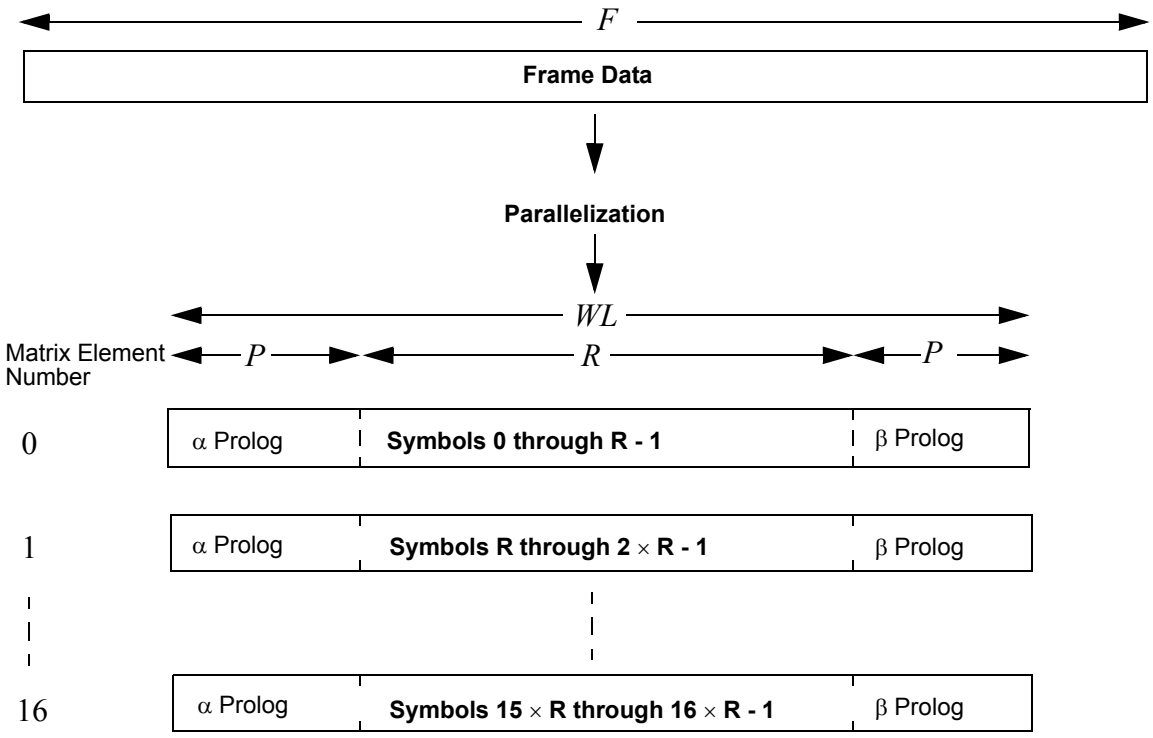

Figure 4: Frame parallelization

4.1.2.2.2. γ

From Figure 2 on page 4 one can see that the turbo encoder's constituent encoder is based on an eight-state, 1/2-rate, convolutional encoder. The trellis for this is shown in Figure 5.

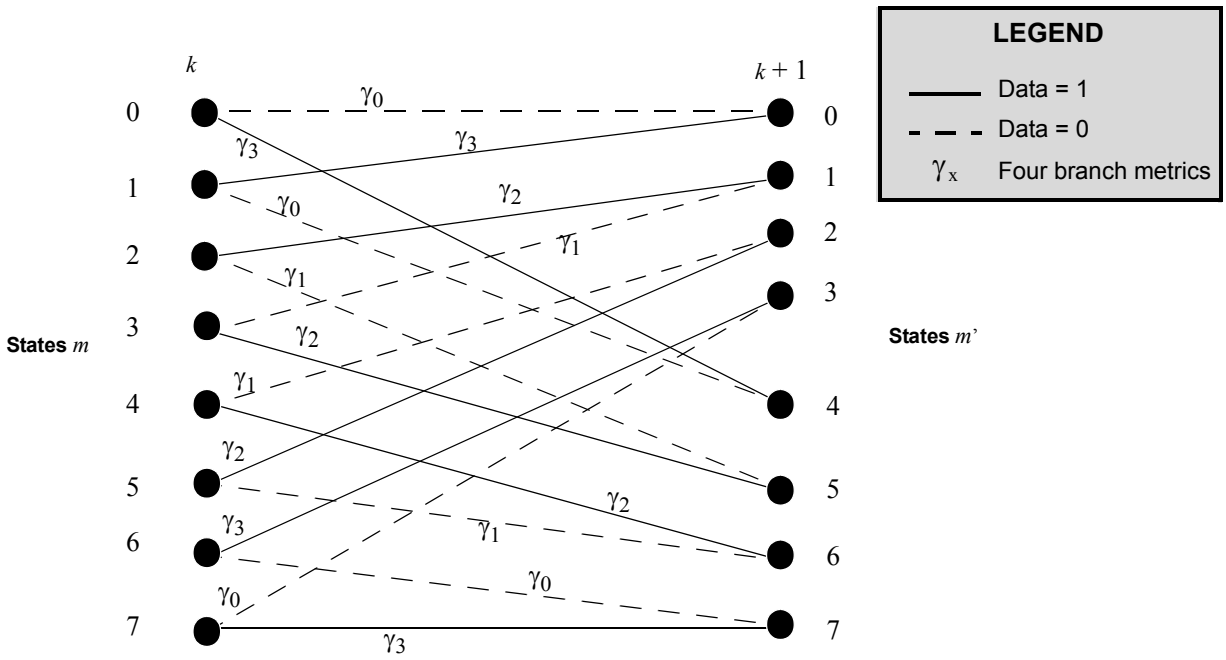

Figure 5: State trellis

Since *u* and *c* are binary values there are only four possible γ values from Eq.(4).

And upon inspection of the trellis and Eq.(4), there is an inverse symmetry displayed between γ_0 and γ_3, as well as between γ_1 and γ_2. This leaves only two γ values to calculate.

The algorithm appears as:

$$\gamma_3 = (Lu[k] + Lc[k])/2;$$
$$\gamma_2 = (Lu[k] - Lc[k])/2;$$
$$\gamma_0 = -\gamma_3$$
$$\gamma_1 = -\gamma_2$$

Eq.(8)

With arrays `Lu` and `Lc` having already been broken down into windows, `k` ranges from 0 to *WL*.

Since the LLRs are arrays of 8-bit symbols, there are four 8-bit symbols packed into a 32-bit matrix element per load. Not shown in the pseudo code above is the cycle cost for the loading (the cycle count of which is hidden with proper scheduling) and unpacking of the `Lu` and `Lc` symbols from 8-bit values into 16-bit halfwords. The cycle count impact is minimal since it is amortized over four γ calculations (four 8-bit symbols per load).

Table 1: γ pseudo code *(Sheet 1 of 2)*

	Pseudo Code	Comments	Cycles
Unpack	*Lu_u* = `MatrixUnpackHighHalfword(`*Lu*`)` *Lc_u* = `MatrixUnpackHighHalfword(`*Lc*`)`	Unpack 16 bit γs into 8- bit halfwords	2

Table 1: γ pseudo code *(Sheet 2 of 2)*

	Pseudo Code	Comments	Cycles
Add	$\gamma_3 \times 2 =$ `MatrixAddHalfword(`Lu_u,Lc_u`)`	Begins γ_3 calculation	1
Sub	$\gamma_2 \times 2$ `= MatrixSubFromHalfword(`Lc_u`,`Lu_u`)`	Begins γ_2 calculation	1
Shift	$\gamma_3 =$ `MatrixShiftRightArithmeticHalfwordImmed(`$\gamma_3 \times 2$,1`)` $\gamma_2 =$ `MatrixShiftRightArithmeticHalfwordImmed(`$\gamma_2 \times 2$,1`)`	Divide above result by 2, for γ_3 and γ_2	2
Store		Stores γs into arrays (No cycle count impact if scheduled properly)	0

4.1.2.2.3. β

The β calculations are actually executed in the same code block as the γ operations. This is because the γ calculations take up very little register resources and the most efficient way of calculating the βs is to have the γs resident in register space.

From examination of the trellis (Figure 5), each symbol instance has eight states associated with it. The β calculations involve a backward recursive summation. To calculate each state, the β values from the two future states connected to it on the trellis are needed. The branch metrics for the branches that connect the two future states are added to the β values. Then the greater value between the two is chosen to be the current times' state value. This is repeated for the state of all eight current times. The β values are all 32-bit integers to accommodate any overflow issues that may arise due to the cumulative nature of the β algorithm.

β algorithm:

From the trellis diagram shown in Figure 5, Figure 6 is extracted.

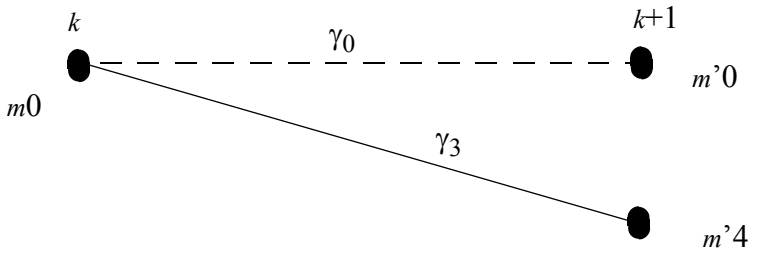

Figure 6: β trellis extraction

Knowing from Eq.(8) that $\gamma_0 = -\gamma_3$, and from Eq.(6) that the current $\beta_k(m)$ is dependent on the two $\beta_{k+1}(m')$ values that are connected to $\beta_k(m)$ on the trellis, the β equation breaks down to the following algorithm:

$$\begin{aligned} &tempBeta0 = \beta_{k+1}(m0) + \gamma(m0) \\ &tempBeta1 = \beta_{k+1}(m1) + \gamma(m1) \\ &\beta_k(m) = (tempBeta0 > tempBeta1)\ ?\ tempBeta0\ :\ tempBeta1 \end{aligned} \quad , \qquad Eq.(9)$$

where *m* is the current state being calculated and *m0* and *m1* are the two future states connected to the current state *m* on the trellis. The above calculation is repeated for all eight states.

Substituting the information from Figure 6 into Eq.(9) will produce the following algorithm snippet for finding $\beta_k(0)$:

`tempBeta0` = $\beta_{k+1}(0) - \gamma_3$

`tempBeta1` = $\beta_{k+1}(4) + \gamma_3$

$\beta_k(0)$ = (`tempBeta0` > `tempBeta1`) ? `tempBeta0` : `tempBeta1`

As stated above, this procedure must be performed for each of the eight states.

Table 2: β pseudo code

	Pseudo Code	Comments	Cycles
Sub	*tempBeta0* = `MatrixSubFromHalfword`(γ_3,β_{k+1})	Begins *tempBeta0* calculation	1
Add	*tempBeta1* = `MatrixAddHalfword`(γ_3,β_{k+1})	Begins *tempBeta1* calculation	1
Compare	`MatrixCompareGreaterThan` (*ConditionCode,tempBeta0,tempBeta1*)	Determine whether *tempBeta0* or *tempBeta1* is greater	1
Select	β_k = `MatrixSelectHalfword` (*ConditionCode*, *tempBeta0*, *tempBeta1*)	Select the result of the `compare` instruction	1

The basic kernel of the β calculation takes four cycles. This is processed for eight states so that the basic β calculation takes 32 cycles. This does not include minor overhead from load-store conflicts that may appear. Essentially, eight loads and eight stores need to be distributed among the eight core kernels of operations to load the eight future state βs and store the eight present state βs. Our current code indicates that there will be about six extra cycles from conflicts. Of course, this is subject to change as the code evolves and is optimized.

Since the γ and β code blocks are integrated, they both perform iterations on the window sizes, from *k = (WL - 1)* down to *k = 0.*

4.1.2.2.4. α

The α calculations are very similar to the β calculations, except that they are forward recursive versus the β calculation's backwards recursion. The calculations for $\alpha_k(m)$ depend on the two $\alpha_{k-1}(m')$ values that are connected to it on the trellis in Figure 7, which is extracted from the trellis in Figure 5.

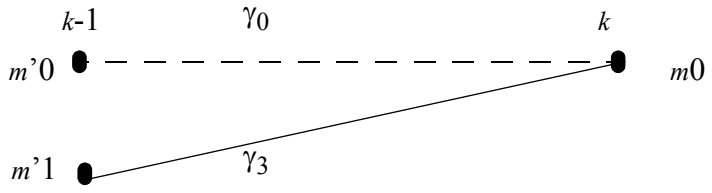

Figure 7: α trellis extraction

Knowing from Eq.(8) that $\gamma_0 = -\gamma_3$, and from Eq.(5) that the current $\alpha_k(m)$ is dependent on the two $\alpha_{k-1}(m')$ values that are connected to it on the trellis, the α equation breaks down to the following algorithm:

$$\begin{aligned} &tempAlpha0 = \alpha_{k-1}(m0) + \gamma(m0) \\ &tempAlpha1 = \alpha_{k-1}(m1) + \gamma(m1) \\ &\alpha_k(m) = (tempAlpha0 > tempAlpha1)\ ?\ tempAlpha0 : tempAlpha1 \end{aligned} \quad , \qquad Eq.(10)$$

where *m* is the current state being calculated and *m0* and *m1* are the two past states connected to the current state *m* on the trellis. The above calculation is repeated for all eight states.

Substituting the information from Figure 7 into Eq.(10) will produce the following algorithm snippet for finding $\alpha_k(0)$:

tempAlpha0 = $\alpha_{k-1}(0) - \gamma_3$

tempAlpha1 = $\alpha_{k-1}(1) + \gamma_3$

$\alpha_k(0)$ = (tempAlpha0 > tempAlpha1) ? tempAlpha0 : tempAlpha1

As stated above, this procedure must be performed eight times for each of the eight states.

Table 3: α pseudo code

	Pseudo Code	Comments	Cycles
Sub	*tempAlpha0* = MatrixSubFromHalfword (γ_3,α_{k-1})	Begins *tempAlpha0* calculation	1
Add	*tempAlpha1* = MatrixAddHalfword (γ_3,α_{k-1})	Begins *tempAlpha1* calculation	1
Compare	MatrixCompareGreaterThan (*ConditionCode*, *tempBeta0*, *tempBeta1*)	Determine whether *tempAlpha0* or *tempAlpha1* is greater	1
Select	βk = MatrixSelectHalfword (*ConditionCode*, *tempAlpha0*, *tempAlpha1*)	Select the result of the compare instruction	1

The load overhead of the past state αs and the current state γs, as well as the store overhead of the α is ignored since most of the cycle overhead is hidden. The α code block gives a cycle count of about 32 cycles/16 output symbols.

The α array is an array of 32-bit signed integers.

The α code block is merged with the extrinsic code block to eliminate the cycle cost of α loading in the extrinsic calculation.

4.1.2.2.5. Extrinsic

The extrinsic block is the final piece of the BCJR max-log-MAP decoder. From Eq.(7), the α and β predictions are combined along with γ to get the probability (*Pr*) that a branch equals 1 and the *Pr* that a branch equals 0. The maximum *Pr* for branch equals 1 and for branch equals 0 are calculated. Then the *Pr* of branch equals 0 is subtracted from the *Pr* of branch equals 1. That result then has subtracted from it *Lu(k)*.

Extrinsic Algorithm:

tempExtrinsicDen = $\alpha_{k-1}(m'0) + \beta_k(m0) + \gamma(m'0)$

Denominator = (Denominator > tempExtrinsicDen) ? Denominator : tempExtrinsicDen

tempExtrinsicNum = $\alpha_{k-1}(m'0) + \beta_k(m1) - \gamma(m'0)$

Numerator = (Numerator > tempExtrinsicNum) ? Numerator : tempExtrinsicNum

where *m'0* is the past state and *m0* and *m1* are the two current states being calculated. The above calculation is repeated for all eight α states to calculate the overall extrinsic value. Once the maximum denominator and numerator are found, the denominator and *Lu(k)* are subtracted from the numerator. This result is a 32-bit signed integer that is shifted down to the final MAP output of 8 bits [signed fixed-point (5,2) representation].

Table 4: Extrinsic pseudo code

	Pseudo Code	Comments	Cycles
Sub	*tempExtrinsicDen* = `MatrixSubFromHalfword(γ,α)`	Begins *tempExtrinsicDen* calculation (α - γ)	1
Add	*tempExtrinsicDen*= `MatrixAddHalfword(β,`*tempExtrinsicDen*`)`	Continues *tempExtrinsicDen* calculation (M5 + β)	1
Compare	`MatrixCompareGreaterThan(`*ConditionCode*`,` *tempExtrinsicDen*`,`*Denominator*`)`	Determine whether *tempExtrinsicDen* or *Denominator* is greater	1
Select	*Denominator* = `MatrixSelectHalfword(`*ConditionCode*`,` *tempExtrinsicDen*`,`*Denominator*`)`	Select the result of the `compare` instruction	1
Sub	*tempExtrinsicNum* = `MatrixSubFromHalfword(γ,α)`	Begins *tempExtrinsicNum* calculation (α - γ)	1
Add	*tempExtrinsicNum* = `MatrixAddHalfword(β,`*tempExtrinsicNum*`)`	Continues *tempExtrinsicNum* calculation (M5 + β)	1
Compare	`MatrixCompareGreaterThan(`*ConditionCode*`,` *tempExtrinsicNum*`,`*Numerator*`)`	Determine whether *tempExtrinsicNum* or *Numerator* is greater	1
Select	*Numerator* = `MatrixSelectHalfword(`*ConditionCode*`,` *tempExtrinsicNum*`,`*Numerator*`)`	Select the result of the `compare` instruction	1

Upon inspection, the above algorithm is inefficient and in actual implementation it is modified to eliminate repetitive steps in the approach. Using the above approach, the extrinsic block would take approximately 66 cycles to calculate. The optimized approach eliminates four `Compare-Select` operations (eight cycles), as well as eight γ additions (eight cycles), giving a final extrinsic of 50 cycles. With current code, it appears that the α/extrinsic block will have about four extra cycles of `load-store` conflict, bringing the overall cycle count up to 54 cycles for the extrinsic step.

4.1.2.2.6. Cycle count performance of the MAP decoder

The max-log-MAP decoder, as it is currently implemented, takes approximately 8.5 cycles per output symbol. The breakdown of cycle count is as follows.

Frame Parallelization:

Takes 100 cycles per 1,024 symbols. There are two input frames that need to be parallelized per one output frame. In addition, one output frame needs to be deparallelized and the operation is identical in cycle count, so there are approximately 3 × (100/1024) = 0.29 cycle per output symbol.

γ/β Block:

The γ operation takes 6 cycles. The β code takes 32 cycles, plus 6 cycles for `load-store` overhead. This gives 44 cycles per 16 symbols or 2.75 cycles per output symbol

α/Extrinsic Block:

The α operation takes 32 cycles. The extrinsic operation takes 54 cycles. This gives 86 cycles per 16 symbols or 5.375 cycles per output symbol for the α/extrinsic block

This gives a MAP block count of: 0.29 + 2.75 + 5.375 = 8.4 cycles per output symbol

4.1.2.3. Turbo interleaver/deinterleaver

The turbo interleaver/deinterleaver spreads symbol information in a fashion that limits the impact of burst noise upon the transmitted information. It is an intra-row and inter-row algorithm [7]. It should be noted that the deinterleaver is the inverse operation of the interleaver. Also to be noted, is that the interleaver here is the exact same interleaver as found in the turbo encoder, but this one operates on bytes instead of bits.

Current implementation does not use the matrix unit exclusively due to the exponential growth of cycle count with a pure permutation/table lookup approach. The current implementation takes four cycles per output symbol for either the interleaver or deinterleaver. The operations consist of loading a table value, adding the table value to a base number, loading input data, and storing the data to a new address.

4.1.2.4. Overall turbo decoder system

The turbo decoder consists of two MAP decoders and two turbo interleavers, from Figure 3. As noted earlier, the analysis presented in this document is approximate, as our design assumptions and code have *not* yet been verified using system level simulations (such as BER/SNR analysis). Also, the FastMATH cycle count estimates have not yet been verified by using the FastMATH cycle accurate simulator.

The following additional comments should be noted regarding the performance of the overall turbo decoder system:

- The α and β prologs are 20 symbols long, based on the observation that Viterbi backward recursion converges at four to five times the constraint length of the encoder [13]. Note that the paper [14], introduces the possibility of having zero length prologs for the sliding windows, instead of using iterative historical data for the α and β initializations. There are three possible approaches: the standard prolog sliding window approach, the new approach where iterative historical data is used, and a third approach that is a hybrid of the two. The point is that the fewer symbols to process, the better the FastMATH performance.
- The sliding window length is assumed to be no less than 100 symbols long.
- Considering that the sliding window is of length 100, and that the best performance for the FastMATH processor is to use all 16 SIMD matrix elements, the minimum frame length should be approximately 1,600 symbols for optimum performance. Frames of smaller length would have to use fewer matrix elements, thereby slowing down the cycle count performance by a factor of (number of matrix elements being used)/16. Other avenues of implementation include processing several frames in parallel, thereby filling the matrix elements to the maximum.
- The γ, α, and β operations are all done in 32-bit integer format to prevent overflow issues. Experiments are being conducted to see if/how a 16-bit format could be used to hold the appropriate amount of data. If it is found that this is possible, a 30 to 50 percent increase in cycle count performance is expected for the MAP decoder.
- It is assumed that six iterations of the turbo decoder are sufficient for an acceptable BER/SNR ratio [11].
- The hard decision slicer generates an output bit of 1 if the number is positive, 0 otherwise. The impact on overall cycle count for this is minimal and is not factored into these numbers.

4.1.3. Summary of Results

In this section we summarize the results for the FastMATH implementation of the turbo decoder for three different frame sizes – 5114, 2688, and 1344. The general assumptions are summarized in Table 11.

Table 5: General turbo decoder assumptions

General Assumptions for Turbo Decoder	
Assumption	**Definition and/or Result**
Input arrays (*Lu[k],Lc[k],Lc'[k]*)	8-bit signed fixed point (5,2)
Output array (before slicer) $Lu_{ext}[k]$	8-bit signed fixed point (5,2)
γ, β, and α	32-bit signed fixed point (28,2)
Frame length range	40 to 5,114 symbols
Window size	minimum of 100 symbols
α and β prolog size	20 symbols apiece
Frame parallelizer cycle count	0.29 cycle per symbol
γ/β cycle count	2.75 cycles per symbol
α/extrinsic cycle count	5.375 cycles per symbol
Overall MAP decoder cycle count	8.5 cycles per symbol
Turbo interleaver/deinterleaver cycle count	4 cycles per symbol
Number of turbo decoder iterations	6
Number of MAP decoder blocks	2
Number of turbo interleaver/deinterleaver blocks	2

Tables 13, 15, and 17 summarize the assumptions and results for the turbo decoder with frame sizes 5,114, 2,688, and 1,344. The following briefly describes (for frame size 5,114) the procedure used to derive the results in the tables:

- Frame length 5,114. This gives a sliding window length of 5,120/16 = 320.
- With a sliding window length of 320, and prologs of 20, the number of symbols to be processed in a sliding window pass-through is 360, for a window of 320 symbols in length, given a bloat ratio of 1.125.
- The number of cycles for one turbo decoder iteration is: (5,114 × 1.125 × 8.5 × 2) + (5,114 × 4 × 2)= 138,716 cycles per frame.
- For six iterations, this gives 832,300 cycles per frame.
- A 384 kbps channel contains roughly 77 frames of length 5,114. This gives 64 million cycles used to process one 384 kbps channel. The FastMATH processor has 2000 million cycles available for use at 2 GHz, giving **31.25** 384 kbps channels per FastMATH processor with the constraints outlined above. In other words, it only takes **3.2 percent** of one FastMATH processor to process one 384 kbps channel.
- The FastMATH processor can turbo decode **12.2 Mbps** when using a frame size of 5,114.

Table 6: Turbo decoder cycle count performance for a frame size of 5,114

Assumptions and Results for Frame Length = 5,114 (Best Case)	
	Definition and/or Result
Frame length for the following calculations	5,114 (best case)
Window size	approximately 320 symbols
α and β prolog size	20 symbols a piece = 40 symbols extra
Overall impact of prolog symbol bloat	1.125 times
Cycles to process one frame through a MAP decoder (MAP Cycles × Frame Size × Prolog Bloat)	8.5 × 5,114 × 1.125 = 48,902 cycles per frame
Cycles to process one frame through a turbo interleaver/deinterleaver (Frame Size × Interleaver Cycles)	5,114 × 4 = 20,456 cycles per frame
Cycles to process one frame through one turbo decode iteration (MAP + Turbo) × 2	(48,902 + 20,456) × 2 = 138,716 cycles per frame per iteration
Cycle count for six turbo decoder iterations	138,716 × 6 = 832,300 cycles per frame
Cycle count to process one 384 kbps 3GPP channel	(384 kbps/5,114) × 832,300 = 64 million cycles
Number of 384 kbps channels one FastMATH processor can turbo decode	**31.25 channels**
Percentage of one FastMATH processor used while processing one 384 kpbs channel	**3.2%**
Overall turbo decoding throughput using one FastMATH processor for a frame size of 5,114	**12.2 Mbps**

Table 7: Turbo decoder cycle count performance for a frame size of 2,688 *(Sheet 1 of 2)*

Assumptions and Results for Frame Length = 2,688	
	Definition and/or Result
Frame length for the following calculations	2,688
Window size	168 symbols
α and β prolog size	20 symbols a piece = 40 symbols extra
Overall impact of prolog symbol bloat	1.238 times
Cycles to process one frame through a MAP decoder (MAP Cycles × Frame Size × Prolog Bloat)	8.5 × 2,688 × 1.238 = 28,285 cycles per frame
Cycles to process one frame through a turbo inter-leaver/deinterleaver (Frame Size × Interleaver Cycles)	2,688 × 4 = 10,752 cycles per frame

Table 7: Turbo decoder cycle count performance for a frame size of 2,688 *(Sheet 2 of 2)*

Assumptions and Results for Frame Length = 2,688	
	Definition and/or Result
Cycles to process one frame through one turbo decode iteration (MAP + Turbo) × 2	(28,285 + 10,752) × 2 = 78,074 cycles per frame per iteration
Cycle count for six turbo decoder iterations	78,074 × 6 = 468,444 cycles per frame
Cycle count to process one 384 kbps 3GPP channel	(384 kbps/2,688) × 468,444 = 68.5 million cycles
Number of 384 kbps channels one FastMATH processor can turbo decode	**29.2 channels**
Percentage of one FastMATH processor used while processing one 384 kpbs channel	**3.4%**
Overall turbo decoding throughput using one FastMATH processor for a frame size of 2,688	**11.5 Mbps**

Table 8: Turbo decoder cycle count performance for a frame size of 1,344 *(Sheet 1 of 2)*

Assumptions and Results for Frame Length = 1,344	
	Definition and/or Result
Frame length for the following calculations	1,344
Window size	84 symbols
α and β prolog size	20 symbols a piece = 40 symbols extra
Overall impact of prolog symbol bloat	1.476 times
Cycles to process one frame through a MAP decoder (MAP Cycles × Frame Size × Prolog Bloat)	8.5 × 1,344 × 1.476 = 16,861 cycles per frame
Cycles to process one frame through a turbo interleaver/deinterleaver (Frame Size × Interleaver Cycles)	1,344 × 4 = 5,376 cycles per frame
Cycles to process one frame through one turbo decode iteration (MAP + Turbo) × 2	(16,861 + 5,376) × 2 = 44,474 cycles per frame per iteration
Cycle count for six turbo decoder iterations	44,474 × 6 = 266,844 cycles per frame
Cycle count to process one 384 kbps 3GPP channel	(384 kbps/1,344) × 266,844 = 78 million cycles
Number of 384 kbps channels one FastMATH processor can turbo decode	**25.6 channels**
Percent of one FastMATH processor used while processing one 384 kpbs channel	**3.9%**

Table 8: Turbo decoder cycle count performance for a frame size of 1,344 *(Sheet 2 of 2)*

Assumptions and Results for Frame Length = 1,344	
	Definition and/or Result
Overall turbo decoding throughput using one FastMATH processor for a frame size of 1,344	**10.1 Mbps**

The turbo decoder proposed here is an extremely fast software solution for 3GPP turbo decoding. The decoder currently has a maximum throughput of **12.2 Mbps**, when the FastMATH processor is running at 2 GHz. The SIMD nature of the FastMATH processor lends itself nicely to the sliding windows approach of turbo decoding. The flexibility of a software approach to turbo decoding is fully exploited here. The solution allows for system optimizations that can be deployed as the code is improved or the systems are altered to incorporate new theories that are developed.

Appendix A to this document investigates several approaches to further speed up the performance of the solution when decoding small frame sizes. Approaches being tested include processing several frames in parallel using a sliding windows approach or processing several frames in parallel using the standard max-log-MAP approach. Another approach being investigated is to store γs, αs, and βs in a 16-bit format, saving approximately 30%-50% of the MAP decoder cycle count.

4.2. Viterbi Decoder

The Viterbi decoder block is used to decode a convolutionally encoded voice channel. The 3GPP convolutional encoder has a constraint length, $K = 9$, and a rate, R, of either 1/2 or 1/3. The encoder produces two or three streams of output, depending on the rate. The decoder takes a soft input and produces a hard decision.

Currently, we have only coded a $R = 1/2$, $K = 6$ Viterbi decoder, but the algorithm is scalable upwards from this kernel size allowing accurate estimates can be generated.

The Viterbi decoder consists of two steps: the forward search and the traceback.

The forward search consists of an add-compare-select butterfly operation, where its cycle count is dependent on the constraint length. The variable encoding rate has negligible to no impact on the forward search step, as the calculation differences between the 1/2 and 1/3 rate can be hidden.

The traceback step consists of a compare-select operation, which chooses the best answer for the output symbol.

The following bullets summarize the cycle count performance estimate of the solution.

- The number of cycles to compute the forward search portion of the $K = 6$ implementation is 20 cycles per output symbol. This is expected to increase by a factor of $2^{(K-6)}$ for higher values of K, since the inner loop contains butterfly operations and the number of butterflies scales as $2^{(K-2)}$. Thus, for $K = 9$ we estimate the inner loop to consume 160 cycles per output symbol.
- We estimate that the traceback step of the $K = 9$ decoder will take 20 cycles. This step breaks down to a conditional statement that takes 5 cycles to determine true or false, then 13 cycles of further calculation if true, or 14 cycles of mispredict if false.

From the above estimates, it can be shown that the overall cycle count performance for a $K = 9$ and $R = 1/2$ or 1/3 Viterbi decoder implemented on a FastMATH processor is 180 cycles per output symbol.

4.3. Turbo Encoder

The 3GPP turbo encoder is based on a parallel concatenated convolutional code that is shown in Section 4.1.1.1 above. Please refer to that section for details about the 3GPP turbo encoder specification.

This section deals with performance estimates for a FastMATH implementation of a 3GPP turbo encoder. The FastMATH turbo encoder consists of the following sub-blocks: turbo interleaver, channel primer, and the constituent encoder.

4.3.1. Turbo interleaver

The turbo interleaver is implemented with the generic bit permute function mentioned in Section 4.5. Performance is identical to the numbers given in Table 14.

4.3.2. Channel primer

The channel primer component takes up to eight frames of data, pairs each frame with its turbo interleaved counterpart, and then rearranges the data into 16 parallel streams. This is done to take advantage the FastMATH processor's SIMD nature. The channel primer block takes 96 cycles per 512 bits to parallelize up to 16 frames.

4.3.3. Constituent encoder

The constituent encoder is comprised of three sub-blocks: the feedback sub-block, the convolutional encoder, and the trellis termination. All operations are the same for the interleaved and non-interleaved frames, which is why they are done in parallel across the FastMATH processor's 16 SIMD elements.

Feedback sub-block:

- Not a standard component of the turbo encoder specification, this is a decomposition of the algorithm to maximize its performance on a SIMD processor. This is due to the use of feedback in the convolutional encoder. This component removes the feedback from the encoder and creates a new virtual input to a simple convolutional encoder. This virtual input is what would be generated by the feedback portion of encoder. It can also be visualized as breaking the feedback driven constituent encoder down into two convolutional encoders in series.
- The algorithm operates on 16 different 32-bit chunks at a time (data has already been prepared by the channel primer).
- It exploits the minor parallelism in the feedback loop (halves the feedback steps)
- To process across the 32 bits of an element (for 16 streams of data) takes 135 cycles.

Convolutional encoder:

- This is a standard convolutional encoder with $K = 4$, $R = 1$.
- Its generator function is constant.
- It takes 11 cycles to process 32 bits of an element (for 16 streams of data).

Trellis termination:

- This creates the tail bits of turbo encoder output
- This operation is to be done after the data has been rearranged from their parallelized state (inverse operation of the original channel primer) back into frames.
- It takes 160 cycles to create the tail bits for all 24 output frames. It is used only once.

4.3.4. Conclusion

The turbo encoder solution for the FastMATH processor, as presented here, is optimized for processing up to eight equal length frames at the same time. Real world situations may dictate that this is an unlikely occurrence, but the algorithm can be modified to deal with frames of unequal length, with a slight overhead. Also, if it is known ahead of time that only one frame is to be dealt with at a time, the algorithm can be modified to not deal with parallelization issues, thereby reducing overhead.

4.4. Convolutional Encoder

The 3GPP convolutional encoder has a constraint length, $K = 9$, and a rate, R, of either 1/2 or 1/3. The maximum frame length that the convolutional encoder must process is 504 bits as stated in [7]. The number of cycles for the generic implementation for any size LESS than 504 bits (a maximum W-CDMA voice channel frame size) is 110 cycles. This constant performance across that 504 bits is derived from the fact that the FastMATH processor can handle vectors of 512 bits per register, so masking and convolving (`XOR` operations) can be done across registers.

4.5. First Interleaver, Second Interleaver, and Rate Matching

The first interleaver, second interleaver, and the rate matching blocks are analyzed together in this section due to the similarities found in their solutions on the FastMATH processor. All involve the movement of bits (on the downlink side) or bytes (on the uplink side) of information in a patterned format. Each of the interleavers consists of a permutation operation with different rules, whereas the rate matching block involves either puncturing or padding a frame of information. The basic premise for implementing these blocks hinges on the fact that there is a permute pattern that can be forced on a general permutation algorithm.

The algorithms can be found in [7]. A brief overview of each algorithm is summarized in the following:

- The first interleaver consists of a one-way, two-way, four-way, or eight-way transposition of the data frame. Data is written by rows into a matrix and then taken out by columns.
- The second interleaver is a 30-way transposition of the data. Data is written by rows into a matrix and then taken out by columns.
- The rate matching block either increases or decreases the length of a frame, where symbols are added or removed to achieve the desired packet length.

Again, it must be noted that all of these operations could be handled by a generic permutation algorithm. This generic permutation algorithm has two flavors: one that operates on the bit-level and one that operates on the byte-level. This generic solution is what we are currently using to perform these algorithms. Since there is some simple structure to the permutations, specialized algorithms may execute faster. The following tables list the performance of the generic permute function.

The generic permute timings are:

Table 9: Cycle count performance of the generic bit permute function

Generic Bit Permute Performance	
Number of Bits	**Number of Cycles**
< 512	256
< 1,024	1,024
< 2,048	2,530
< 4,096	5,060
< 8,192	11,170
< 16,384	28,100

Table 10: Cycle count performance of the generic byte permute function

Generic Byte Permute Performance	
Number of Bytes	**Number of Cycles**
< 64	30
< 128	120
< 256	240
≥ 256	4 cycles per byte

Generation of the interleave pattern can be done either with a small set of precomputed tables which are combined at runtime to produce the interleave pattern, or by pregenerating the interleave pattern for all (or all expected) packet lengths.

For small packet sizes (less than 512 bytes) the interleave can be done entirely in the FastMATH matrix unit in approximately one cycle per byte. Very small packets (64 bytes) can be permuted in 30 cycles or 0.5 cycle per symbol.

For larger packet sizes, the essentially random nature of the interleave makes the matrix unit less useful. For the largest packet size, cache and memory issues become the limiting factor in interleaver performance and an unrolled loop using the scalar unit will perform as well as the matrix unit.

The loop moves one symbol every four cycles, consisting of a “load halfword” of the table value, an add of the table value to the base, a “load byte” or “load word” for the input data and a “store byte” or “store word” to the destination.

As was noted before, special permute functions in the future will be used for each interleaver and the rate matcher. For example, the first interleaver with a transposition of two is a simple two-way bit shuffle which can be done in about 100 cycles for a 1,024-bit packet, which translates into a 10x speedup over the generic permute.

These specially tailored permute functions will provide a great boost in the performance of the interleavers and the rate matcher.

5. Overall Results of FastMATH Implementation for Symbol-Rate Processing

In this document, we have evaluated the use of the Intrinsity FastMATH processor in the symbol-rate baseband processing of a typical 3GPP base station receiver system. The turbo decoder block was identified as the block requiring high processing bandwidth and that would greatly benefit from the implementation flexibility and programmability provided by a FastMATH solution. The programmed solution for symbol-rate processing on the FastMATH processor is a flexible, high-performance system. This section summarizes the symbol-rate processing results for the base station transmitter and receiver using the FastMATH processor for a voice user (12.2 kbps AMR) and a data user (384 kbps).

The first case that we will examine concerns a 12.2 kbps AMR voice channel. In this case, the major blocks involved in the symbol-rate processing of a voice channel include: convolutional encoder, first interleaver, second interleaver, rate matcher on the downlink (base station transmit), and rate matcher, second deinterleaver, first deinterleaver, Viterbi decoder on the uplink (receiver). It should be noted that there are other pieces involved in 3GPP symbol-rate processing (CRC, multiplexing, block concatenation, etc.) that have not been analyzed in this document since the performance requirements for these blocks is expected to be very small.

The frame size being used for the 12.2 kbps is 103 bits. (From [8]; note that this example simplifies the 12.2 kbps AMR channel used in [8] for purposes of clarity. This simplification in no way hinders the performance analysis pre-

sented here). To determine frame sizes we go through the following (simplified) breakdown of what the downlink baseband side transmitter produces. Once the frame has been passed through the CRC, 12 bits are added to the end of the frame, giving 115 bits to be processed by the convolutional encoder. In this example, the parameters of the convolutional encoder are $K = 9$, $R = 1/3$. This will give an interleaved output frame of length (115 + 8 tailbits) × 3 = 369 bits. The frame will then pass through the rate matcher (which may slightly increase or decrease the frame size) and the two interleavers. Tables 13 and 14 summarize the assumptions and results for the FastMATH implementation of symbol-rate processing (12.2 kbps voice channel) for the transmitter (downlink) and the receiver (uplink), respectively.

Table 11: Assumptions and results of the FastMATH implementation of a 12.2 kbps AMR voice channel (base station transmit/downlink)

Assumptions and Results for 12.2 kbps AMR Voice Channel (Downlink)	
	Definition and/or Result
Original frame length for the following calculations	103 symbols
Convolutional encoder cycle count	110 cycles
First interleaver	256 cycles (369 bits < 512 bits)
Rate matcher	256 cycles (369 bits < 512 bits)
Second interleaver	256 cycles (369 bits < 512 bits)
Cycles to process one frame through on the downlink side for a 12.2 kbps voice channel	878 cycles per frame =110 + (256 × 3)
Number of 103 symbol-sized frames in a 12.2 kbps channel	122 = 12.2 kbps/103 symbols
Cycle count to process one entire 12.2 kbps channel on the downlink side	107,116 cycles/channel = 122 frames × 878 cycles per frame

Table 12: Assumptions and results of the FastMATH implementation of a 12.2 kbps AMR channel (uplink and total results) *(Sheet 1 of 2)*

Assumptions and Results for 12.2 kbps AMR Voice Channel (Uplink)	
	Definition and/or Result
Original frame length for the following calculations	103 symbols
Second deinterleaver	1,476 = 369 symbols × 4 cycles per symbol
Rate matcher	1,476 = 369 symbols × 4 cycles per symbol
First deinterleaver	1,476 = 369 symbols × 4 cycles per symbol
Viterbi decoder cycle count for one frame	20,700 = (103 + 12) symbols × 180 cycles per symbol
Cycles to process one frame through on the uplink side for a 12.2 kbps voice channel	25,128 cycles per frame = 20,700 + (1,476 × 3)

Table 12: Assumptions and results of the FastMATH implementation of a 12.2 kbps AMR channel (uplink and total results) *(Sheet 2 of 2)*

Assumptions and Results for 12.2 kbps AMR Voice Channel (Uplink)	
	Definition and/or Result
Number of 103 symbol-sized frames in a 12.2 kbps channel	122 = 12.2 kbps/103 symbols
Cycle count to process one entire 12.2 kbps channel on the uplink side	3,065,616 cycles per channel = 122 frames × 25,128 cycles/frame
Cycle count to process one entire 12.2 kbps channel on the downlink side	107,116 cycles per channel = 122 frames × 878 cycles per frame
Cycle count to process one entire 12.2 kbps channel (both downlink and uplink)	3,172,732 cycles/channel = 107,116 + 3,065,616
Percentage use of a FastMATH processor for one 12.2 kbps voice channel	**0.158% = 3,172,732 / 2 GHz**

The second case to be examined concerns a 384 kbps data channel. The major blocks involved in processing a data channel include: turbo encoder, first interleaver, second interleaver, rate matcher on the downlink side, and rate matcher, second deinterleaver, first deinterleaver, turbo decoder on the uplink side. It should be noted that there are other pieces involved in 3GPP symbol-rate processing (CRC, multiplexing, block concatenation, etc.) that have not been analyzed in this document, since the performance requirements for these blocks is expected to be very small.

The frame being used for the 384 kbps data channel is made up of twelve sub-blocks of length 336 bits that are processed through a CRC, which adds 16 bits to each sub-block and then concatenates. The resulting frame size is 4,224 bits (from [8]). To determine frame sizes, we go through the following (simplified) breakdown of what the downlink baseband side transmitter produces. Once the frame has been passed through the CRC and concatenated, it is handed off to the turbo encoding block. After it is passed through the turbo encoder, the frame is now 12,684 bits (symbols) long. The frame will then pass through the first interleaver, the rate matcher (which may slightly increase or decrease the frame size) and the second interleaver. Tables 27 and 29 summarize the assumptions and results for the FastMATH implementation of symbol-rate processing (384 kbps data channel) for the transmitter (downlink) and the receiver (uplink) respectively.

Table 13: Performance chart for a 384 kbps data channel (downlink side) *(Sheet 1 of 2)*

Assumptions and Results for 384 kbps Data channel (Downlink)	
	Definition and/or Result
Original frame length for the following calculations	4,032 symbols
Turbo encoder cycle count for eight frames (max performance...cycle count would be the same for one frame)	110,520 cycles/8 frames = (8 × 11,170) + (2 × 9 × 96) + (132 × 135) + (132 × 11) + 160
First deinterleaver cycle count per frame	28,100 cycles (12,684 bits < 16,384 bits)
Rate matcher cycle count per frame	28,100 cycles (12,84 bits < 16,384 bits)
Second deinterleaver cycle count per frame	28,100 cycles (12,684 bits < 16,384 bits)

Table 13: Performance chart for a 384 kbps data channel (downlink side) *(Sheet 2 of 2)*

Assumptions and Results for 384 kbps Data channel (Downlink)	
	Definition and/or Result
Cycles to process one frame through on the downlink side for a 384 kbps data channel	98,115 = (110,520/8) + (3 × 28,100)
Number of 4,032 symbol-sized frames in a 384 kbps channel	98 = 384 kbps/4,032 symbols
Cycle count to process one entire 384 kbps channel on the downlink side	9,615,270 cycles/channel = 98 frames × 98,115 cycles per frame

Table 14: Performance chart for a 384 kbps data channel (uplink and overall total) *(Sheet 1 of 2)*

Assumptions and Results for One 384 kbps Data Channel (Uplink)	
	Definition and/or Result
Original frame length for the following calculations	4,032 symbols
Second deinterleaver cycle count for a frame	50,736 cycles = 4 cycles × 12,684 symbols
Rate matcher cycle count for a frame	50,736 cycles = 4 cycles × 12,684 symbols
First deinterleaver cycle count for a frame	50,736 cycles = 4 cycles × 12,684 symbols
Turbo decoder window size for a 4,224 symbol-sized frame	264 symbols per window = 4,224/16
Overall prolog symbol bloat	1.15 = ((264 + 40) × 16)/4,224
Cycles to process one frame through a MAP decoder (MAP Cycles × Frame Size × Prolog Bloat)	41,290 cycles per frame = 8.5 × 4,224 × 1.15
Cycles to process one frame through a turbo interleaver/deinterleaver (Frame Size × Interleaver Cycles)	16,896 cycles per frame = 4,224 × 4
Cycles to process one frame through one turbo decode iteration (MAP + Turbo) × 2	116,372 cycles per frame per iteration = (41,290 + 16,896)) × 2
Cycle count for 6 turbo decoder iterations for one frame	689,232 cycles per frame = 116,372 × 6
Cycles to process one frame through on the uplink side for a 384 kbps data channel	841,440 cycles per frame = (50,736 × 3) + 689,232
Number of 4032 symbol-sized frames in a 384 kbps channel	98 = 384 kbps/4,032 symbols
Cycle count to process one entire 384 kbps channel on the uplink side	82,461,120 cycles = 841,440 cycles per frame × 98 frames
Cycle count to process one entire 384 kbps channel on the downlink side	9,615,270 cycles per channel = 98 frames × 98,115 cycles per frame

Table 14: Performance chart for a 384 kbps data channel (uplink and overall total) *(Sheet 2 of 2)*

Assumptions and Results for One 384 kbps Data Channel (Uplink)	
	Definition and/or Result
Cycle count to process one entire 384 kbps channel (both downlink and uplink)	92,076,390 cycles per channel = 9,615,270 + 82,461,120
Percentage use of a FastMATH processor for one 384 kbps data channel	**4.6% = 92,076,390 / 2 GHz**

Thus, one voice user requires the processing power of only 0.158 percent of a FastMATH processor, and one high data rate (384 kbps) data channel requires the processing power of only five percent of a FastMATH processor.

As can be seen from the above estimates, the FastMATH processor is an extremely fast software solution for 3GPP symbol-rate processing. The estimates above will actually improve with respect to performance in the future, due to the fact that a generic permutation algorithm has currently been used in place of the specialized transpositions defined in the 3GPP WCDMA specification. As we analyze the system more thoroughly and move away from using prologs, shortening window sizes, and shifting from 32-bit operations to 16-bit operations another area of great improvement will be in the turbo decoder. The cumulative effect of those improvements will be to increase performance by anywhere from 30 to 60 percent of the current numbers.

The processing power and flexibility of the FastMATH processor can eliminate the need for extra processors to handle symbol-rate processing while it handles more intensive tasks, such as MUD or smart antennas. The FastMATH processor's programmability can ensure that symbol-rate systems are up to date with changing specifications, in addition to allowing systems to be upgraded on the fly with the current state of the art solutions to problems such as turbo decoding. This flexibility will ensure maximum performance for changing systems.

Again, note that some blocks that take minimal calculation time have been omitted. The blocks presented here account for the vast majority of cycle count overhead and are representative of the symbol-rate processing requirements.

6. Acronyms

3GPP	Third Generation Partnership Project
AI	Acquisition indicator
ALU	Arithmetic logic unit
ASC	Access service class
CDMA	Code division multiple access system
CPCH	Common packet channel
DPCH	Dedicated physical channel
DS-CDMA	Direct sequence CDMA system
FDD	Frequency division duplexing
MAC	Multiply-accumulator
PRACH	Physical random access channel
RACH	Random access channel
SIMD	Single-instruction, multiple-data
UE	User equipment or mobile unit
UMTS	Universal mobile telephone system
UTRA	UMTS terrestrial radio access

7. References

[1] http://www.3gpp.org/

[2] 3GPP TS 25.201 v4.0.0 (2001-03): "Physical layer - General description"

[3] "CDMA: Principles of Spread Spectrum Communication", A. J. Viterbi, Addison-Wesley Wireless Communication Series, 1995

[4] 3GPP TS 25.213 v4.1.0 (2001-06): "Radio Access Network Spreading and Modulation (FDD)"

[5] 3GPP TS 25.214 v4.1.0 (2001-06): "Radio Access Network Physical layer procedures (FDD)"

[6] 3GPP TS 25.211 v4.1.0 (2001-06): "Radio Access Network Physical channels and mapping of transport channels onto physical channels (FDD)"

[7] 3GPP TS 25.212 v4.1.0 (2001-06): "Radio Access Network Multiplexing and channel coding (FDD)"

[8] 3GPP TS 25.944 v4.1.0 (2001-06): "Radio Access Network Channel coding and multiplexing examples"

[9] L.M. Correia, ed., Wireless flexible personalized communications - COST 259: European cooperation in mobile radio research, John Wiley & Sons 2001.

[10] 3GPP TS 25.943 v4.0.0 (2001-06): "Radio Access Network Deployment aspects"

[11] J.P. Woodard, L. Hanzo, "Comparative Study of Turbo Decoding Techniques: An Overview", *IEEE Trans. On Vehicular Technology*, Vol. 49, No. 6, November 2000 pp 2208-2233.

[12] S. Benedetto, D. Divsalar, G. Montorsi, F. Pollara, "Soft-Output Decoding Algorithms for Continuous Decoding of Parallel Concatenated Convolutional Codes", *Proc. Int. Conference on Communications (ICC '96)*, Dallas, USA, June 1996, pp. 112 - 117.

[13] P.J. Black, T. H-Y. Meng, "A 1-GB/s, Four-State, Sliding Block Viterbi Decoder", *IEEE Journal of Solid State Circuits*, Vol.32, No. 6, June 1997.

[14] A. Dingninou, "Implémentation de Turbo Code pour trames courtes (Implementation of Turbo Codes for short frames)", *Ph. D. Thesis* Université de Bretagne Occidentale, Brest, FRANCE, July 3 2001.

8. Revision History

Revision	Date	Brief Description
1.0	2/20/02	Created from 3Gppbaseband.doc
1.1	2/22/02	Customer release
1.2	3/1/02	Minor changes
1.3	4/11/02	Cosmetic changes
1.4	6/19/02	Cosmetic changes
1.5	3/26/03	New format and logo with tag line
1.6	4/21/03	Data updates to Table 1 and Table 4

No express or implied licenses are granted hereunder for the design, manufacture or dissemination of any information or technology described herein or the use of any trademarks used herein.

Without in any way limiting any obligations provided for in any confidentiality or non-disclosure agreement between the recipient hereof and Intrinsity, Inc., which shall apply in full with respect to all information contained herein, recipients of this document, by their acceptance and retention of this document and the accompanying materials, acknowledge and agree to the foregoing and to preserve the confidentiality of the contents of this document and all accompanying documents and materials and to return all such documents and materials to Intrinsity, Inc. upon request or upon conclusion of recipient's evaluation of the information contained herein.

Any and all information, including technical data, computer software, documentation or other commercial materials contained in or delivered in conjunction with this document (collectively, "Technical Data") were developed exclusively at private expense, and such Technical Data is made up entirely of commercial items and/or commercial computer software. Any and all Technical Data that may be delivered to the United States Government or any governmental agency or political subdivision of the United States Government (the "Government") are delivered with restricted rights in accordance with Subpart 12.2 of the Federal Acquisition Regulation and Parts 227 and 252 of the Defense Federal Acquisition Regulation Supplement. The use of Technical Data is restricted in accordance with the terms set forth herein and the terms of any license agreement(s) and/or contract terms and conditions covering information containing Technical Data received between Intrinsity, Inc. or any third party and the Government, and the Government is granted no rights in the Technical Data except as may be provided expressly in such documents.

Portions of this document and/or the materials provided herewith may have been extracted from documents entitled MIPS32™ Architecture for Programmers (the "MIPS Extracts"), provided by MIPS Technologies, Inc. ("MIPS") and any such portions are reproduced under agreement with MIPS. Any and all MIPS Extracts are subject to the following additional legend: