TURBO CODES ON THE FIXED POINT DSP TMS320C55x

Tri Ngo
Philips Semiconductors
30 Corporate Park, Suite 100
Irvine, CA 92606
Email: tri.ngo@vlsi.com

Ingrid Verbauwhede
Dept. of Electrical Engineering, UCLA
7440B Boelter Hall, Box 951594
Los Angeles, CA 90095-1594
Email: ingrid@ee.ucla.edu

Abstract: Turbo codes are introduced in 3rd generation wireless cellular standards for their superior coding gain. The MIPS requirements of turbo codes are however extremely high. This paper describes the implementation of a turbo-decoding algorithm on the TMS320C55x processor. The coding performance is evaluated with fixed-point arithmetic. A method to optimize the memory is also introduced to address the large data storage problem. The effect of finite word lengths is carefully examined to reduce the state metric normalization time and to achieve at the same time an acceptable bit error rate (BER). The coding gain of 5.8 dB for a BER of 10^{-3} in 6 iterations is achieved with frame size of 1K bits and 50 Hz frame rate. Total MIPS estimate when using the Max Log-MAP algorithm is 46 MIPS.

I. INTRODUCTION

Turbo codes are introduced in 3rd generation wireless cellular standards for their superior coding gain. The MIPS requirements of turbo codes are however extremely high. Therefore, its practical feasibility must be evaluated for the available technology. Unlike other previous DSP generations, the primary hardware resource of the fixed-point DSP C55x CPU consists of several application-specific instructions, which offers both high code density for control tasks and efficient execution for turbo decoding algorithm. For example, the index search (Max_diff) is twice as fast as the search instruction in the C54x. The powerful indirect addressing modes will accelerate the Viterbi butterfly through the entire decoding process.

Turbo codes have been implemented on the DSP-based, C6x [1] and on an ASIC [2]. Both implementations are designed for deep space applications. In that case the BER is the most important design issue and the complexity is less important design issue. In this implementation, we focus on the trade-off between complexity and BER performance. The intended is in next generation wireless standard. This paper describes a turbo code implementation on the Texas Instruments TMS320C55x. First, we describe the turbo code algorithm, that is being proposed for the CDMA 2000 system in section II. Next, in section III, we describe the implementation on the C55x. Important implementation issues are discussed such as reducing the state metric normalization, and maximizing the data throughput rate. In section IV, the performance is evaluated. Finally, in section V, the conclusions and future work are given.

0-7803-6488-0/00/$10.00 © 2000 IEEE

II. DECODING ALGORITHM

For a practical implementation, the continuous flow of information is split into frames. The advantage of using a block code is that the memory requirements of the Maximum A Posteriori (MAP) decoding can be reduced significantly. Also, the decoding delay is now set by the length of the block and does not depend on the transmission rate. The terminated turbo encoder scheme (Fig. 1) is being proposed in the CDMA 2000 system [3]. The switches inside the encoder scheme are used to erase the register contents between every frame length transmission. The interleaver has the same size of the frame and is designed as a pair of row-column registers, which are used alternatively for the reading and writing operations.

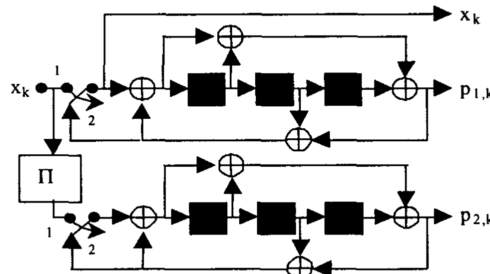


Fig.1 Turbo encoder of rate 1/3.

The turbo decoder is made up of two MAP decoding modules that cooperate in an iterative scheme (Fig. 2). The soft extrinsic output of one MAP module feeds into the other MAP module. This algorithm refines the estimate of the information

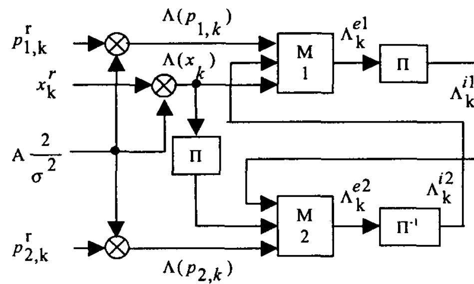


Fig. 2 MAP turbo decoder.

sequence until convergence is reached. The practical issues are thus to reduce the working memory and to increase the throughput. A sliding window (SW) technique on the Log-MAP algorithm [4] is well suited to fulfill those issues. The SW technique is used to transform the convolutional code into a block code by segmenting the transmitted sequence into adjacent blocks, and periodically force the trellis termination.

0-7803-6488-0/00/$10.00 © 2000 IEEE

The transmission system used in our simulations consists of a random frame generator, a turbo encoder, BPSK modulators, AWGN channels, and a turbo decoder as shown on Fig. 3. Each received symbol is given by a Gaussian distribution with mean ±1 and variance σ^2. The log likelihood ratios (LLR) of the received symbols $\{\Lambda(x), \Lambda(p_1), \Lambda(p_2)\}$ are computed by multiplying the received signals with a factor $2A/\sigma^2$. The scaling factor A is introduced to avoid overflow.

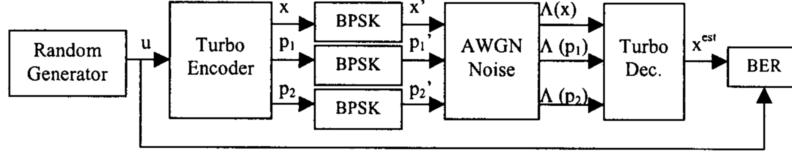


Fig. 3 Transmission system

The α metrics are started in known initial states at the beginning of each window and are computed using the forward recursion.

$$\alpha_{k+1}(s) = \mathrm{Max}\left[\{\alpha_k(s_1) + \Gamma_k(s,s_1) + x_k(s,s_1)\Lambda_k^{int}\},\{\alpha_k(s_2) + \Gamma_k(s,s_2) + x_k(s,s_2)\Lambda_k^{int}\}\right] \quad (1)$$

Where $\{s_1, s_2\}$ are the states at trellis stage k that merge into state s at trellis stage k+1 in the forward path. Similarly the β metrics are started in known initial states at the end of each window and are computed using the backward recursion.

$$\beta_{k-1}(t) = \mathrm{Max}\left[\{\beta_k(t_1) + \Gamma_k(t,t_1) + x_k(t,t_1)\Lambda_k^{int}\},\{\beta_k(t_2) + \Gamma_k(t,t_2) + x_k(t,t_2)\Lambda_k^{int}\}\right] \quad (2)$$

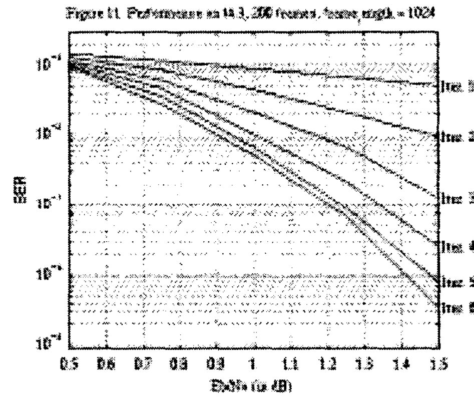
Where $\{t_1, t_2\}$ are the states at trellis stage k that merge into state t at trellis stage k-1 in the backward path. The branch metric $\Gamma(s, s')$ is given in (3), with p_i refers to either p_1 or p_2.

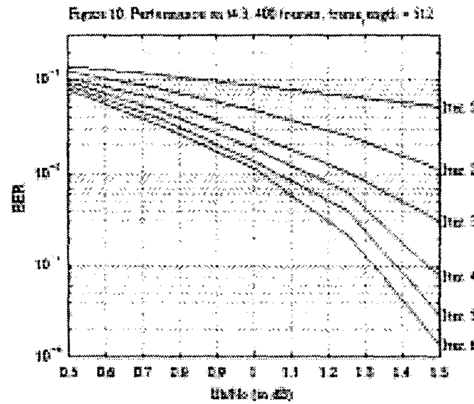
$$\Gamma_k(s,s') = x_k \Lambda(x_k) + p_{i,k}\Lambda(p_{i,k}) \quad (3)$$

Λ_k^{int} is called the intrinsic information which is used as a priori information by the next decoder. It has the value of the extrinsic information Λ_k^{ext} after interleaving or de-interleaving according to the diagram in Fig. 2. Λ_k^{ext} is determined by the following equations.

$$\Lambda_k^{ext} = \underset{x_k=1}{\mathrm{Max}}\left[\alpha_{k+1}(s') + \Gamma_k(s,s') + \beta_k(s)\right] - \underset{x_k=-1}{\mathrm{Max}}\left[\alpha_{k+1}(s') + \Gamma_k(s,s') + \beta_k(s)\right] \quad (4)$$

A fixed point C program has been written and evaluated for several different formats (X.Y) of 8bits received sample representation, where X and Y are the integer and fractional bits respectively. The best result for Max-Log-MAP algorithm (Fig. 4) is achieved for 4 bits of integer and 3 bits of fractional.





III. TMS320C55X IMPLEMENTATION

The C55x is a programmable fixed point DSP with a variable length instruction set and parallel execution of instructions. While the variable length instruction offers a high code density for control tasks, the parallel execution offers an efficient execution for many DSP applications. With twice the functional units (MACS, ALUS, and Accumulators) in the C55x core compared to the C54x core, the data computational unit (DU) supports crucial parallel instructions that increase cycle efficiency. The additional buses and address generators enable multiple operand operations and reduce memory bottlenecks.

This section focuses on optimizing the working memory through the SW technique. Fortunately, careful storage and memory reuse allows for high throughput with limited memory resources. A circular buffer implementation is the main key to minimize the pointer manipulation problem in the metric update with zero overhead hardware. In the metric update section, using parallel instructions and avoiding pipeline stalls are also investigated to speed up the program.

A. Memory Organization

All memory resources are organized around a unified program and data space of 16 Mbytes. The program memory space is linear byte-addressable. The data memory space is a 8Mword word-addressable, which is segmented into 128 main data pages of 64Kwords. The eight address registers (AR0-7) and four data registers (DR0-4) are used to indirectly access to one of the main data pages. Table I gives an overview of the memory map in the C55x architecture.

Program Address		Data Address
00_0000	MMRs	00_0000
00_0c00	DARAM (32K)	00_0060
01_0000	SARAM (128K)	00_8000
05_0000	External	02_8000
FF_FFFF		7F_FFFF

MMR = Memory Map Register
DARAM = Dual Access RAM
SARAM = Single Access RAM

Table I. C55x Unified Memory Map

The memory resources that were used in the decoder implementation have been divided into regions that characterize the size and speed of the memory. The fastest memory region MMR is used to store the actual DSP executable code. The DARAM and SARAM regions are used to store the stacks, the local variables and any variables that require high performance memory. The slowest external memory region is used for post-processing memory access. Table II shows the memory assignments for the MAP module.

Region	Variable	Description	Size (bytes)
DARAM	Γ	Branch metric	2*(WS+P)
	β_prolog	Initial backward recursion	32
	β	Backward metric	16*WS
SARAM	α	Forward metric	32
	Λ^{ext1}	Extrinsic data of MAP1	FS
	Λ^{ext2}	Extrinsic data of MAP2	FS
External	$\Lambda(x)$	LLR of x	(FS+6)*N
	$\Lambda(p_1)$	LLR of parity 1	(FS+3)*N
	$\Lambda(p_2)$	LLR of parity 2	(FS+3)*N
	x_out	Output binary data	FS*N/8

WS ≡ window size
P ≡ Prolog length
FS ≡ Frame size
N ≡ # frames

Table II. MAP Module Memory Section Assignments.

B. Metric Update

The computational complexity of a turbo decoder is dominated by the MAP module implementation. Since all of the states must be updated at each trellis stage, most of the decoding time is spent on the metric update. Therefore, much effort has gone into minimizing the metric update calculation time. The metric update process involves the four steps of computing Γ, α, β, and Λ^{ext}. The

extremely useful instruction from the C55x processor is "max_diff", which is well suited for either Log-MAP or Max-Log-MAP algorithms. This complex instruction executes several operations in parallel and hence it will speed up the program and reduce the code size. Also, the pre-modified indirect addressing mode will help us to compute the forward and backward recursions fast.

1. Compute Γ-metrics

First, all the received symbols are converted into LLR by scaling them by the factor $2A/\sigma^2$. Then, the Γ-metrics are calculated for each trellis stage k and are reused for the forward and backward recursions. Table III shows that only $\{\Gamma_{10}, \Gamma_{11}\}$ require to be stored in the Γ-buffer.

<table>
<tr><th colspan="2">stage k</th><th>Γ metrics routine</th><th colspan="2">Assembly code</th></tr>
<tr><td>x</td><td>p</td><td>Γ metrics</td><td>Hi (AC0) = *AR0+ + DR0,</td><td>; AC0_H = Λ(x)+Λ(p)</td></tr>
<tr><td>0</td><td>0</td><td>$\Gamma_{00} = -\Gamma_{11}$</td><td>Lo (AC0) = *AR0+ - DR0</td><td>; AC0_L = Λ(x)+Λ(p)</td></tr>
<tr><td>0</td><td>1</td><td>$\Gamma_{01} = -\Gamma_{10}$</td><td>|| DR0 = *AR1+</td><td>; DR0 = Next Λ(p)</td></tr>
<tr><td>1</td><td>0</td><td>$\Gamma_{10} = [\Lambda(x) - \Lambda(p)] >> A$</td><td>Hi (*AR2+) = Hi (AC0) >>1,</td><td>; Store Γ11[], Γ10[]</td></tr>
<tr><td>1</td><td>1</td><td>$\Gamma_{11} = [\Lambda(x) + \Lambda(p)] >> A$</td><td>Lo (*AR2+) = Lo (AC0) >>1</td><td>; Assume A = 1</td></tr>
</table>

Table III. Branch metrics

The scaling factor A is used to scale the Γ-metrics, to avoid overflow in the forward recursion. Two address registers AR0, and AR1 are used to access to $\Lambda(x_k)$, and $\Lambda(p_k)$, respectively. Using the *dual add/subtract* instruction and the explicit parallelism technique, the Γ-metrics are computed in just 1 cycle. The *dual add/subtract* instruction performs the complementary calculation, storing subtraction results in the lower accumulator and addition results in the upper accumulator. The indirect addressing mode will help us to load the results back to the Γ-buffer.

2. Backward recursion

The prolog β metrics are used to initialize the backward recursion in each window. The process is performed on the butterfly as shown in Fig. 5 over an interval of a prolog length, P. The process starts when all states, except state 0, are set to the same initial metric value. In this implementation, the state 0 is set to value of 0, while all other states are set to the minimum possible values (0x8000), providing room for growth as the metrics are updated. Except for the last window, the β-prologs are performed on the terminated trellis.

Due to the symmetry of the RSC code, two starting and ending states are paired in a butterfly structure including all branches between them (Fig. 5). This structure provides only one Γ metric is needed for each butterfly. It is alternately added and subtracted from the old β metrics to form the new β metrics. The complete butterfly structure can be implemented in C55x with the *dual add/subtract* and the *max_diff* instructions. The *max_diff* compares the two 16-bit signed values in the upper and lower halves of the accumulator and stores the maximum value to

0-7803-6488-0/00/$10.00 © 2000 IEEE

memory. In addition, *max_diff* instruction also generates the difference between these two numbers, which can be used as an index pointer to the right corrective term in the lookup table.

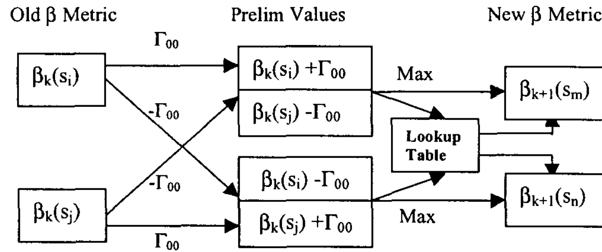


Fig. 5 Butterfly structure to compute β metrics.

The prolog β metric storage requires two buffers, each with a size equal to the number of states (8 words). At the end of the metric update, these buffers are swapped so that the recently updated metrics become the old metrics for the next stage. In order to minimize pointer manipulation, these buffers are usually configured as a single circular buffer as shown in Fig. 6. The old metrics are accessed in consecutive order, requiring only one pointer AR0. The new metrics are updated in the order $\{\beta_0, \beta_4, \beta_2, \beta_6, \beta_1, \beta_5, \beta_3, \beta_7\}$, requiring two pointers AR1, AR2 for addressing. Address register AR3 is used to access to the Γ buffer.

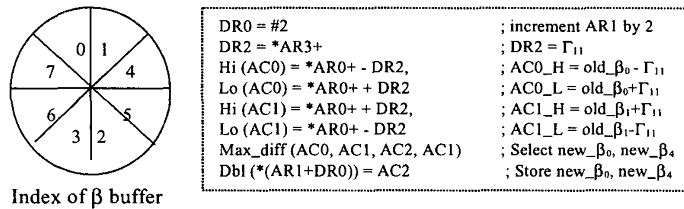


Fig.6 β buffer and assembly code to implement the butterfly.

The β computation routine is the same as the prolog β routine. However, the linear buffer is used to store every update metric. The read pointer AR0 is used to access to the previous array memory in the consecutive order, while two write pointers (AR1, AR2) are used to store the update metrics to the next memory array. These write pointers are swapped once during the updating process of each stage.

3. Forward recursion

Unlike the initialization process in the backward recursion, the last α-metrics of the recent window can be used as the initial values to start in the next window.

These values are normalized to prevent buffer overflow. Normalization is done only once for each window, except the first window: the state 0 is set to value of 0, while all other states are set to the minimum possible values (0x8000). At the final node of the forward recursion, the maximum possible value (0x7fff) is subtracted from each of them.

The α-metrics are also performed on the same butterfly as show in Fig 5. After normalization process, eight α-metrics are calculated for every trellis stage and immediately consumed to produce the extrinsic information Λ^{ext}. Two circular buffers are sufficient to perform the forward recursion. At the end of the α-metrics update, these buffers are interchanged so that the recently updated metrics become the old metrics for the next trellis stage. The index metrics in the α-buffer is similar as shown in β-buffer. Except two address registers AR0, AR1 are alternately used to update the old metrics. The address register AR2 is used to store the new metrics and is incremented by 1.

4. Extrinsic information

The extrinsic computation performs the composition of three sets of quantities (Γ, α, β) needed by the Max-Log-MAP algorithm. The algorithm takes all branches into its calculation, but splits them into two best branches that are associated with input bits 0 and 1. The extrinsic information is determined from the difference of these branches. Fig. 7 presents the circuit tree of searching for the extrinsic information bit where its Max* module is used to select the survivor path.

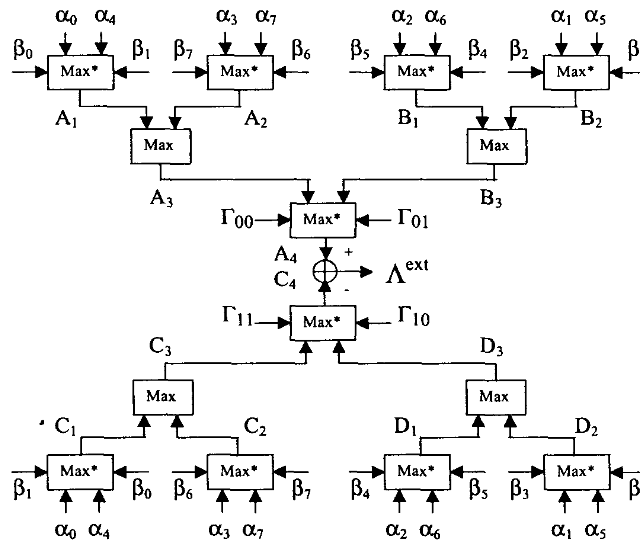


Fig. 7 Implementation of extrinsic bit information.

The following code illustrates the implementation of Max* module. A temporary buffer of size 8 words can be used as the output buffer for all three levels of computations. The buffer is also organized in table IV in order to simplify the pointer manipulation. Finally, the saturation mode will be set in order to convert the intrinsic output back to a byte format.

Temporary Buffer				Max* module assembly code	
word	1st level	2nd level	3rd level	Pair (DR2) = *(AR0 + DR0)	; DR2 = α_0, DR3 = α_4
				Hi (AC0) = Hi (*(AR1+DR1)) + DR2,	; AC0_H = $\beta_0 + \alpha_0$
0	A1	A3	A4	Lo (AC0) = Lo (*(AR1+DR1)) + DR2	; AC0_L = $\beta_1 + \alpha_0$
1	C1	C3	C4	Hi (AC1) = Hi (*AR1+) + DR3,	; AC1_H = $\beta_1 + \alpha_4$
2	B1	B3		Lo (AC1) = Lo (*AR1+) + DR3	; AC0_L = $\beta_0 + \alpha_4$
3	D1	D3		Max_diff (AC1, AC0, AC2, AC1)	; AC2_H = A1,
					; AC2_L = C1
4	A2				
5	C2				
6	B2				
7	D2				

Table IV. Extrinsic Output Computation

IV. CPU PERFORMANCES

Table V summarizes the code performance in terms of the memory usage and number of cycles for all the functions, which used to implement a fixed-point MAP module. The decoding cycles per frame is obtained from all the function cycles except minor processor-initialization tasks. The equivalent MIPS are found by multiplying the decoding cycles per frame with the frame rate, FR.

Function	Log-MAP Cycles/bit	Max Log-MAP Cycles/bit	MIPS/Frame
Γ metrics	2	2	1. Log-MAP Algorithm [86 + (8+20P)/WS]*FS*FR
Prolog β metrics	20	15	
β metrics	20	15	
α-normalization	8	8	2. Max Log-MAP Algorithm [71 + (8+15P)/WS]*FS*FR
α metrics	20	15	
Extrinsic output	42	37	
(De)Interleaver	2	2	

Table V. Code Performance for a MAP module

With (FS =1024, P = 24, WS = 100) bits, each MAP module requires 76630 (or 93270) cycles/frame by using Max Log-MAP (or Log-MAP) algorithm, which is equivalent to 3.83 (or 4.66) MIPS at a 50 Hz frame rate. Thus, the total MIPS for 6 iterations is 3.83*12 = 46 (or 4.66*12 = 56) MIPS. Table VI summarizes the trade-

off between complexity and BER performance of different designed frame sizes of using Max Log-MAP algorithm.

Frame Size	# Frames	Total MIPS	Memory (Mbytes)	BER
0.5K	400	23	1.231712	5.7 dB
1K	200	46	1.229312	5.8 dB
2K	100	92	1.228112	5.9 dB

Table VI. Total MIPS and BER trade-off

V. CONCLUSION

In this paper, a successful SW technique is introduced in the Log-MAP decoding algorithm. The methodology has resulted in a significant decrease in memory and decoding delay. That helps to break down the complexity issue of turbo decoder implementation in hardware. One single normalization step is sufficient in each window. Through the MAP module implementation, we show that the turbo decoder for a long bit stream can be implemented just using the DSP C55x alone. The memory model should be selected in order to divide the memory up into regions that characterize the size and speed of the memory. Future work consists of optimizing the C55x assembly code for both MAX* and Log-MAP algorithms.

ACKNOWLEDGMENTS:

This work was performed while Tri Ngo was a Master Student at UCLA. This research was supported in part by ATMEL Corporation under the Micro Program # 98-162. We would like to thank Wanda Gass from Texas Instruments for the help in the fixed-point assembly code implementation and Christina Fragouli for the help with the fixed-point C simulation.

References:

[1] William J. Ebel, "Turbo Code on The C6x", *Alexandria Research Institute, Virginia Tech.*

[2] G. Masera, G. Piccinini, M.R. Rock, M. Zamboni, "VLSI Architectures for Turbo Codes," *IEEE Trans. on VLSI systems,* Vol. 7, No. 3, Sept. 1999.

[3] Steve Dennett, "The cdma2000 ITU-R RTT Candidate Submission," *Telecom. Industry Association (TIA),* May 15, 1998.

[4] A. J. Viterbi, "An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes," *IEEE Journal on Selected Areas in Communications,* Vol. 16, No. 2, February 1998.