An Efficient Viterbi Decoder Implementation for the ZSP500 DSP Core

Danny Wilson
LSI Logic, Advanced DSP Dev.
500 N. Central Expressway, Suite 430
Plano, TX 75074
011 1 972 244 5134
dannyw@lsil.com

ABSTRACT
This paper describes an efficient implementation of the Viterbi decoding algorithm on the ZSP500 digital signal processor (DSP) core. It starts with an introduction to convolutional coding and Viterbi decoding as a method of forward error correction in communication systems. An introduction to the ZSP500 architecture is followed by a description of special instructions for performing the Trellis butterfly. Examples of branch metrics calculation, Trellis butterfly, and trace-back are given. Performance benchmarks for the Viterbi algorithm running on ZSP500 are presented. Finally, the use of a Viterbi coprocessor for the task of state metric update and trace-back is discussed.

Categories and Subject Descriptors
B.2.4 [**Arithmetic And Logic Structures**]: High Speed Arithmetic – *algorithms,* J.2 [**Physical Sciences And Engineering**]: Engineering, E.4 [**Coding And Information Theory**]: Error Control Codes.

General Terms
Algorithms, Performance, Design.

Keywords
ZSP500, convolutional encoder, Viterbi, DSP, Trellis, branch metrics, path metrics, trace-back, coprocessor, signal processing, benchmarks, GSM.

1. INTRODUCTION
Andrew J. Viterbi, a co-founder of QUALCOMM, proposed the Viterbi algorithm in 1967 as a method for decoding of convolutional codes [1]. Since then, this technique has been widely used for forward error correction in data modems and wireless communications. Although the Viterbi algorithm has many other applications – including optical storage, image processing, speech recognition, multiple target tracking, holographic memory systems, predicting microwave propagation loss, and handwriting recognition – this paper will focus on using it for forward error correction.

1.1 FORWARD ERROR CORRECTION
Forward error correction is used to decrease bit error rate (BER) on noisy communication channels. This is achieved by a method known as channel coding, which adds redundant information to the transmitted data. With forward error correction, transmission errors are corrected at the decoder, without requesting a retransmission. Convolutional encoding and block coding are two major forms of channel coding.

Viterbi decoding is an optimal method [2] for decoding convolutional codes when the transmission occurs in the presence of additive white Gaussian noise, and the Euclidean distance metric is used.

1.2 CONVOLUTIONAL ENCODING
The Global System for Mobile Communications (GSM) voice channel was chosen to illustrate the Viterbi algorithm on ZSP500. GSM operates on 50 frames of speech data per second with each frame containing N=189 bits that need to be protected by forward error correction. Figure 1 illustrates the rate ½ convolutional encoder used for this purpose, generating 378 output bits for each frame. The encoder state at any instant is given by $S_3S_2S_1S_0$.

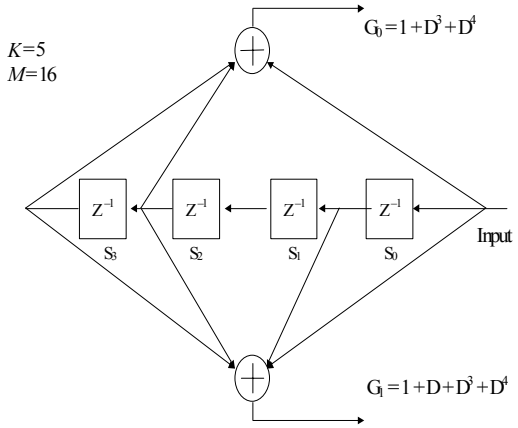

Figure 1: GSM Convolutional encoder for voice data.

The channel encoder has a constraint length of K=5 and can therefore attain $M = 2^{K-1} = 16$ unique states. For each data block, the encoder will start from a zero state ($S_3S_2S_1S_0 = 0000$) with all memory elements containing zeros, and at the end of the data block, it is again reset to the zero state by using a sequence of 4 zeros as tail bits, as part of the input data frame.

The Trellis diagram shown in Figure 2 can also represent convolutional encoding. Given an encoder state $(\mathbf{S_3S_2S_1S_0})_{(n-1)}$ at any stage (n-1) within each data frame, the Trellis diagram identifies the state at the following stage $(\mathbf{S_3S_2S_1S_0})_n$ based on the data bit input to the encoder. The two encoded output bits are shown above the old path, with the 0-input result first,

followed by 1-input result. The output bits (G_0, G_1) associated with each state transition are governed by the equations from Figure 1.

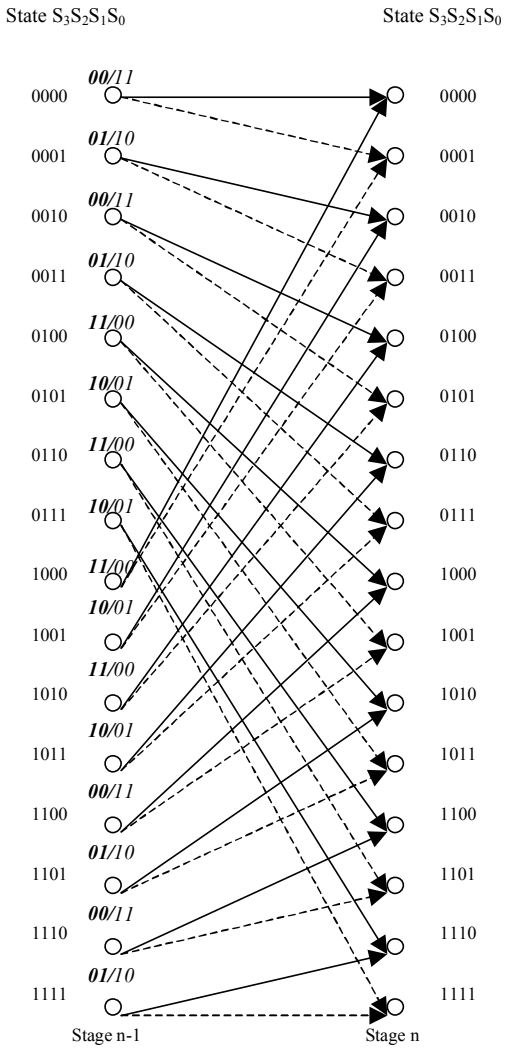

Figure 2: Trellis Diagram state transitions for K=5, M=16. Legend: Solid Lines–Zero Input. Dashed Lines–One Input

1.3 Viterbi Decoder

The Viterbi decoder finds the optimal path through an M-state, N-stage Trellis diagram, and then traces back through the Trellis to generate the N decoded output bits.

Without the Viterbi algorithm, the number of paths that the decoder must examine doubles at each stage. With N=189 stages for GSM, the amount of processing and memory required would be impractical.

With the Viterbi decoder, at most M paths survive, no matter the number of stages. At every stage, a cost metric is used to select the survivor path from the two incoming paths to each state; the other path is discarded. By this means, only M path costs are maintained, each being a stage-by-stage sum of the individual branch costs leading to that state. Given (Xn, Yn) is the decoder input symbol at stage n, decoding complexity is typically reduced by using the Manhattan distance cost metric for branch cost calculations [3]:

$BC_{00} = Xn + Yn$; $BC_{11} = -BC_{00} = -Xn - Yn$

$BC_{01} = Xn - Yn$; $BC_{10} = -BC_{01} = -Xn + Yn$

Inputs to the decoder represent the logarithmic probability of a 0 or 1 transition. "Soft decision" inputs are represented with multiple bits per transition, while "hard decision" inputs use a single bit. Using either approach, the branch metric of a transition is the logarithmic probability of the transition. The path metric is the logarithmic probability of a sequence of transitions, and can be calculated as a sum of branch metrics.

Symmetry in the Trellis diagram can be used to reduce the number of path metric calculations. Figure 3 shows the butterfly structure associated with the Viterbi decoder – pairing new states 2m and 2m+1 with previous states m and m+M/2. Even though there are four incoming paths, there are only two branch costs, and they are arranged so that one is the negative of the other.

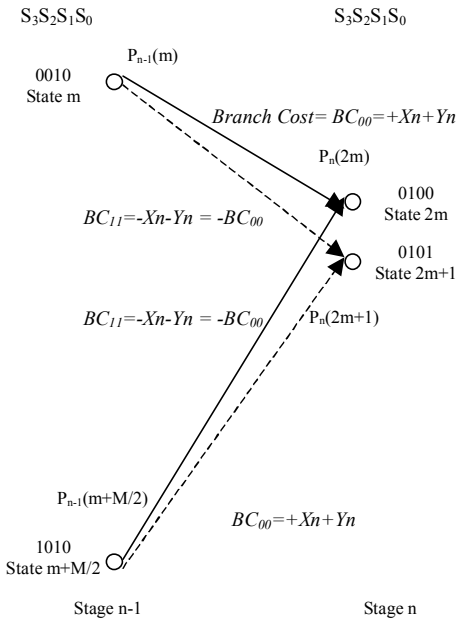

Figure 3: Butterfly diagram showing branch cost symmetry.

Path metrics for each new state are calculated using each incoming branch cost plus the previous path cost associated with that branch. The minimum of the two incoming paths is selected as the survivor. The butterfly computations consist of two "add-compare-select" (ACS) operations and updating the survivor path history. The two ACS operations are:

$$P_n(2m) = \min\{P_{n-1}(m) + B_x,\ P_{n-1}(m+M/2) + B_y\},\ \text{and}$$
$$P_n(2m+1) = \min\{P_{n-1}(m) + B_y,\ P_{n-1}(m+M/2) + B_x\}.$$

After completing N stages of decoding, one of the M survivor paths is selected for trace-back. Since GSM uses four tail bits to reset the encoder state to zero, there is no need to calculate the shortest of the M paths. The zero state is selected to begin traceback.

2. ZSP500 DSP CORE HARDWARE

2.1 Architecture Overview

The ZSP500 is a dual-MAC, superscalar, fixed-point, licensable digital signal processor (DSP) core. It can execute up to 4 simultaneous instructions each clock cycle. The core is based on the ZSP G2 architecture, which is the second generation of ZSP digital signal processor architecture from LSI Logic. The ZSP500 has 1 Multiply Accumulate Unit (MAU), 2 Arithmetic/Logic Units (ALU), and 2 Address Generation Units (AGU) as shown in Figure 4.

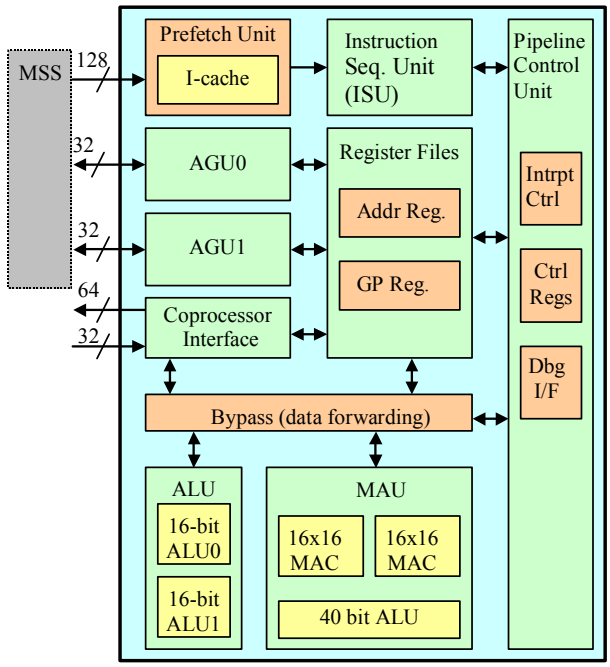

Figure 4: Simplified ZSP500 Architecture

In 1997, the first generation ZSP architecture was offered in a standard product. Following its successful introduction, LSI Logic introduced the ZSP400, a licensable core, to address system-on-a-chip (SOC) applications.

The ZSP G2 architecture is backwards compatible with the ZSP400 core at the assembler level, allowing ZSP400 code to be reused for the ZSP500 core without modification. The ZSP500 enhancements include additional 40-bit ALUs, 24-bit addressing for instructions and data, additional registers for addressing, and an enhanced instruction set.

In Figure 4, the external busses to the Instruction Prefetch Unit, AGU_0, and AGU_1 go to the memory sub-system (MSS). The MSS can be specified by the chip designer, and is independent from the ZSP500 core. Benchmark performance assumes the reference MSS provided by LSI Logic.

2.2 Viterbi Instructions

The ACS operation for selecting survivor paths is the most cycle intensive operation in the Viterbi algorithm. Through the use of special three-operand vit_a and vit_b instructions, ZSP500 can perform two ACS operations each cycle. The inputs to these instructions are two register pairs. One pair contains path costs for the previous stage. The other contains branch costs for transitions leading to the current stage. The destination register can be any general purpose register, and the LSBs of a 32-bit %vitr register record the selected branch (0 or 1) for the ACS operations. The vit_x instructions execute in either MAU_0 or across both ALUs, (ALU_0 and ALU_1) combined. The ZSP500 vit_x instruction definitions are:

```
vit_a rZ,rXe,rYe
      rZ = min {(rX+rY), (r(X+1)+r(Y+1))}
      if ((rX+rY) < (r(X+1)+r(Y+1)))
      {vitr = vitr << 1 | 0x0000_0001}
      else {vitr = vitr << 1}

vit_b rZ,rXe,rYe
      rZ = min {(rX+r(Y+1)), (r(X+1)+rY)}
      if (((rX+r(Y+1)) < (r(X+1)+rY)))
      {vitr = vitr << 1 | 0x0000_0001}
      else {vitr = vitr << 1}
```

For both these instructions, the register pair rXe holds the path costs, while the pair rYe holds the branch costs. Depending on how the branch costs are stored in the rYe pair, the vit_a instruction can even perform the task of vit_b and vice versa.

2.3 Other Supporting Features

2.3.1 Register Resources

The ZSP500 core includes 16 general purpose registers (GPR) in addition to 8 address and 8 index registers. This large set of registers provides significant flexibility in the decoder software design. The four branch costs BC_{00}, BC_{11}, BC_{01} and BC_{10} computed at each stage can be held in the registers while all the Viterbi butterflies at that stage are being processed. There are also sufficient registers available calculate four new path costs before storing results.

2.3.2 Data Bandwidth

The ZSP500 supports a peak data bandwidth of 128 bits per cycle, and a sustained bandwidth of 64 bits per cycle. For each AGU, there is a 24-bit address bus, a 32-bit load data bus, and a 32-bit store data bus. Either the two data load busses or the two data store busses can be concatenated for 64-bit data transfers.

In the ZSP500 implementation of the Viterbi symbol loop a peak data throughput of 96 bits per cycle (load 4 old path costs & store 2 new path costs) is achieved without memory cycle stalls.

2.3.3 Bit Manipulation

The ZSP500 has enhanced support for bit manipulation, including bit insert, bit extract operations. The traceback processing makes effective use of these bit-level operations.

2.3.4 Circular Buffers

The ZSP500 has hardware support for four circular buffers. If a pointer used for circular addressing is updated beyond the address range of a circular buffer, the address will wrap to a valid circular buffer location. The Viterbi decoder implementation uses circular buffers to minimize path cost pointer management costs.

3. ZSP500 VITERBI IMPLEMENTATION

3.1 Data Structures

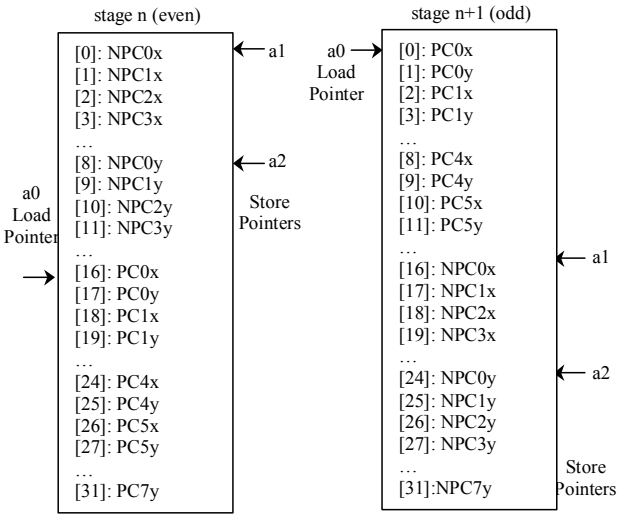

Figure 5: Path Cost Data Structure

The 32-element data structure shown in Figure 5 is used to maintain path costs. The new path cost (NPC) output at stage n becomes the old path cost (PC) in stage n+1. A single pointer (a0) is used to sequentially load up to 4 old path costs (PC) as input to the vit_x instructions. vit_x destination registers are selected so that path cost reordering is not needed prior to storing. Two pointers (a1 & a2) store data at the NPC(m) and NPC(m+M/2) locations. Data is stored to allow sequential access at the nest stage.

3.2 Symbol Loop

3.2.1 General Purpose Registers

r0	ACS_A result via vit_a
r1	ACS_A result via vit_b
r2	ACS_B result via vit_b
r3	ACS_B result via vit_a
r4	PCmx
r5	PCmy
r6	PC(m+1)x
r7	PC(m+1)y
r8	Constant 8 (for address calculation)
r12	BC_{11} = -RxX-RxY
r13	BC_{00}= +RxX+RxY
r14	BC_{10}= -RxY+RxY
r15	BC_{01}= +RxX-RxY

Registers r0-r4 are new path costs generated from vit_a and vit_b instructions. Using previous path costs in r4-r7 and branch costs in r12-r15, as above, new branch costs can be calculated as follows:

```
NPC0x =   Vit_a r0, r4, r12   // r0 = min ((r4+r12),(r5+r13))
NPC0y =   Vit_b r2, r4, r12   // r2 = min ((r4+r13), (r5_r12))
NPC1x =   Vit_b r1, r6, r12   // r1 = min ((r6+r12), (r7+r13))
NPC1y =   Vit_a r3, r6, r12   // r3 = min ((r6+r13), (r7+r12))
```

3.2.2 Address Registers

a0	*PATHCOST
a1	*NPC1x
a2	*(NPC1x+8) = *NPC1y
a4	*TRCBAK
a5	*RXDATA

3.2.3 Index Registers

n0	Constant 4

3.2.4 Code

Twelve cycles per stage are required for the symbol loop. The following variables are initialized prior to the start of the loop:

r12 = RxX; r13 = BC_{00}; r14 = RxY; r4 = PC0x; r5 = PC0y

```
SymbLoop: // Cycle_1:
// Load previous path costs (PC1x, PC1y) for use in cycle 3.
// Calculate BC10 & BC11; Update a2 store pointer
      lddu    r6, a0, 2  // r6 = PC1x, r7 = PC1y
      sub     r14, r12              // r14 = BC10
      neg     r12, r13  // r12 = BC11
      add     a2, r8     // a2 = (NPC0x+8) = NPC0y
Cycle_2:
// Calculate survivor path metrics for NPC0x & NPC0y.
// r0 = NPC0x  = min ((r4+r12),  (r5+r13))
// r2 = NPC0y = min  ((r4+r13),  (r5+r12))
      vit_a   r0, r4, r12
      vit_b   r2, r4, r12
Cycle_3:
// Path costs in r6/r7;  Branch costs in r12/r13;  R15 = BC01.
// Calculate survivor path metrics for NPC1x.
      neg     r15, r14              // r15 = BC01
      vit_b   r1, r6, r12
Cycle_4:
// Calculate survivor path metrics for NPC1y.
      vit_a   r3, r6, r12
      ldqxu   r4, a0     // r4=PC2x, r5=PC2y, r6=PC3x, r7=PC3y
      stdu    r0, a1, 2  // Store NPC0x, NPC1x
Cycle_5:
// Calculate survivor path metrics for NPC2x & NPC2y.
      stdu    r2, a2, 2   // Store NPC0y, NPC1y
      vit_a   r0, r4, r12
      vit_b   r2, r4, r12
Cycle_6:
// Calculate survivor path metrics for NPC3x & NPC3y.
      vit_b   r1, r6, r12
      vit_a   r3, r6, r12
```

```
    ldqxu    r4, a0          // r4=PC4x, r5=PC4y, r6=PC5x,
r7=PC5y
    stdu    r0, a1, 2    // Store NPC2x, NPC3x
Cycle_7:
// Calculate survivor path metrics for NPC4x & NPC4y
    stdu    r2, a2, 2    // Store NPC2y, NPC3y
    vit_a   r0, r4, r14
    vit_b   r2, r4, r14
Cycle_8:
// Calculate survivor path metrics for NPC5x & NPC5y.
    vit_b   r1, r6, r14
    vit_a   r3, r6, r14
    ldqxu   r4, a0      // r4=PC6x, r5=PC6y, r6=PC7x, r7=PC7y
    stdu    r0, a1, 2    // Store NPC4x, NPC5x
Cycle_9:
// Calculate survivor path metrics for NPC6x & NCP6y.
    stdu    r2, a2, 2    // Store NPC4y, NPC5y
    vit_a   r0, r4, r14
    vit_b   r2, r4, r14
Cycle_10:
// Calculate survivor path metrics for NPC7x & NPC7y.
    lddu    r12, a5, 2  // r12 = RxX, r13 = RxY (next symbol)
    lddu    r4, a0, 2    // r4 = PC0x, r5 = PC0y (for next stage)
    vit_b   r1, r6, r14
    vit_a   r3, r6, r14
Cycle_11:
    stdu    r0, a1, 2    // Store NPC6x, NPC7x
    stdu    r2, a2, 2    // Store NPC6y, NPC7y
    mov     r14, r13   // r14= RxY
    add     r13, r12    // r13 = RxX+RxY
Cycle_12:
// a2 now points to NPC0x for next stage
    mov     a1, a2      // a1 = NPC0x for next stage
    mov     r0, %vitr // only 16 bits of 32-bit %vitr used
    stu     r0, a4, 1    // Update Traceback Table
    agn1    SymbLoop
```

3.3 Traceback

3.3.1 General Purpose Registers

```
    r0      current state (rC)
    r1      current state (rC)
    r2      2*rem(rC,8) + 1 (r2 LSB is set to 1)
    r4      traceback bit
    r5      control word for traceback bit extraction
    r6      r6 = 0 (if rC < 8) or 1 (if rC > 7)
    r13     traceback info for current state
    r14     working register for accumulated output bits
```

3.3.2 Address Registers

```
    a3      *OUTPUT
    a4      *TRCBAK
```

3.3.3 Code

A traceback table is generated during the symbol loop, by storing 16 bits of the Viterbi register (%vitr) after stage. These bits are the history of survivor paths selected for each ACS function, and hence the history of encoder input bits that generated those paths. A "0" means the first path of the vit_x comparison was selected as minimum. A "1" means the second path of the vit_x comparison was selected.

Since GSM flushes the state to zero at the end of each frame, traceback starts with the zero state of the last stage. The traceback algorithm follows the Trellis diagram backwards through all 189 Viterbi register entries. It determines the current output bit for a stage by using the current state as an index into the Viterbi register bits. Output bits are recovered in the reverse order from their transmission.

```
TBLoops: // Cycle_TB_1:
// Begin building a control word (r5) to extract the traceback bit
    shll    r15, 1        // r15 << 1 and r15[LSB] = 0
    ins     r5, r0        // r5 = 0x0#01  Control: <r1 = 0x903>
    shrl    r6, 3         // r6 = 0 (if rC < 8) or 1 (if rC > 7)
    ldu     r13, a4, -1  // r13 (rT) = Traceback info (vitr)
Cycle_TB_2:
// Complete building the r5 control word
// Reverse the traceback info
    ins     r5, r6, 8, 1   // r5 = 0x0#01
    revb    r13, 15       // reverse Traceback info sequence
Cycle_TB_3:
// Using control in r5, extract the traceback bit from r13 to r4
// r4 now has current traceback bit
    ins     r2, r0, 1, 3   // r2 = 2*rem(rC,8) + 1 (r2 LSB is setto
1)
    ext     r4, r13        // r4[LSB] = "traceback bit" (r4 = 0 or 1)
Cycle_TB_4:
// Update output word with current output bit
    or      r15, r6       // r15[LSB]=current output bit
    sub     r0, r2, r4    // r0 = rC (updated)
    sub     r6, r2, r4    // r6 = rC (to extract bit rC[3])
    agn0    TBLoops
Cycle_TB_5:
// Store packed output; reload loop counter; repeat loop
    stu     r15, a3, -1    // Store O/P (packed word)
    mov     r15, 0         // Clear O/P word
    mov     %loop0, 15  // 16-bits per O/P word
    agn1    TBLoops
```

An average of 4 cycles per stage are used in the traceback. The outer loop is executed once per output word, for 12 iterations. The inner loop is executed once per bit in each output word -- 16 iterations for all but the 1st word, which uses 13 iterations. Total cycles for traceback is (16*4+1)*12 – (3*4) = 768 cycles.

4. PERFORMANCE

Viterbi benchmark cycle counts for general purpose DSPs are shown in Table 1. Cycles for Viterbi decoding, traceback and overhead are all combined into "Cycles per Decoded Output Bit." Sources for external information are documented in the references.

Table 1: Viterbi Benchmarks for DSPs – Rate=½, Constraint Length=5, Frame Size=189 Bits.

Processor	Cycles per Decoded Output Bit
Texas Instruments 'C54 [4]	**61**
ADSP-2106x SHARC [5]	**58**
Texas Instruments 'C62 [6]	**38**
3DSP SP-5 [7]	**33**
Infineon Carmel [8]	**22**
LSI Logic ZSP500	**16**
Texas Instruments 'C64 [9]	**14**
StarCore SC140 [10]	**10**

ZSP500 offers an efficient implementation of Viterbi decoding for the GSM speech channel. With 189 bits per frame at 50 frames per second, the ZSP500 processing load is 0.16 MHz. Depending on additional application tasks that need to be handled, multiple voice channels can be assigned to the ZSP500. This is also for true for recent 3G wireless systems, where convolutional encoders with longer constraint lengths (K=9) are used. The ZSP500 processing load for Viterbi decoding of each WCDMA speech channel is estimated at 2.4 MHz.

However for applications requiring hundreds of voice channels to be supported at a low cost and power consumption per channel, or when much higher data rate non-voice channels are involved, a Viterbi coprocessor might be an appropriate alternative.

5. COPROCESSOR

Combining ZSP500 with a Viterbi coprocessor provides an enhanced decoder that can evolve along with emerging standards and new algorithms. The DSP provides flexibility, software programmability, additional signal processing, and product differentiation. The coprocessor provides enhanced computational capabilities. Considering the long-term trend for DSPs to become more powerful, coprocessors functions may later become DSP subroutines.

The ZSP500 has two types of coprocessor instructions: cpcom and cpout. The cpcom instructions provide a "tightly coupled" coprocessor interface, where data returns while the instruction is in the pipeline. An example of a "tightly coupled" coprocessor is given in reference [11]. The cpout instruction provides a "loosely coupled" interface, with the coprocessor signaling when data is available. A Viterbi coprocessor can be supported with cpout instructions.

A coprocessor can be formed using readily available Viterbi decoder cores and a coprocessor wrapper (Figure 6). The wrapper provides the interfaces for the ZSP500 core, Viterbi core, and MSS.

The Multiresolution Viterbi Metacore from LSI Logic [12] is suitable for a ZSP500 Viterbi coprocessor. The Multiresolution algorithm uses a combination of soft decision and hard decision decoding to achieve its design goals. As a Metacore, it can be optimized across 8 degrees of freedom for the objectives of BER, area, and throughput. The algorithm exhibits the BER performance of higher resolution soft decoding while maintaining minimal area and delay.

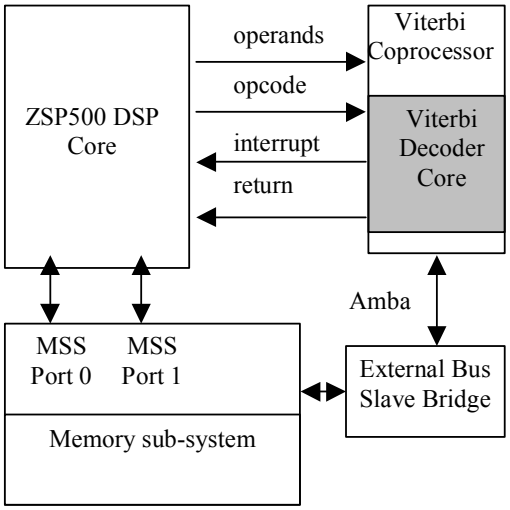

Figure 6: ZSP500 with Viterbi Coprocessor

6. CONCLUSION

This paper presented an introduction to Viterbi decoding, along with an efficient implementation of the Viterbi algorithm for the ZSP500 DSP core. Benchmarks were presented for the GSM voice channel decoder. At 16 cycles per decoded output bit, the ZSP500 Viterbi implementation outperforms mainstream dual-MAC DSPs and approaches performance seen in high performance quad-MAC DSPs with more functional units.

An efficient Viterbi coprocessor based on the Multiresolution Viterbi Metacore was also presented. This conbination of ZSP500 DSP and Multiresolution Viterbi coprocessor provides an efficient, flexible, Viterbi decoding solution with enhanced capability.

7. REFERENCES

[1] Viterbi, A. J., "Error Bounds for Convolutional Codes and an Asymptotically Optimal Decoding Algorithm," IEEE Transactions On Information Theory, Volume IT-13, No. 2, April 1967, 260-269.

[2] Forney, G.D. Jr., "Maximum-likelihood sequence estimation of digital sequences in the presence of

intersymbol interference," IEEE Transactions on Information Theory, vol. IT-18, March 1972, 363-378.

[3] Lou, H., "Viterbi decoder design for the IS-95 CDMA forward link", Vehicular Technology Conference, 1996. 'Mobile Technology for the Human Race'., IEEE 46th, Vol.2, Iss., 28 Apr-1 May 1996, 1346-1350.

[4] Hendrix, H., "Viterbi Decoding Techniques for the TMS320C54x DSP Generation (Rev A)," Texas Instruments Application Report, SPRA071A, January 2002.

[5] Sauve', P., Crozier, S., Hunt, A., "High-Speed DSP Implementations of Viterbi Decoders", International Mobile Satellite Conference, IMSC '99, 297-302.

[6] http://www.ti.com/sc/docs/products/dsp/c6000/62benchh.htm

[7] Bindra, A., "DSP Core Optimized for Physical Layer Processing," Electronic Design, October 15, 2001, 15.

[8] Sucher, R., Niggebaum, R., Fettweiss, G., Rom, A., "CARMEL – A New High Performance DSP Core Using CLIW," Proceedings of the International Conference on Signal Processing and Applications Technology, 1998.

[9] http://dspvillage.ti.com/docs/catalog/generation/details.jhtml?templateId=5154&path=templatedata/cm/dspdetail/data/c64_benchmark

[10] "How to Implement a Viterbi Decoder on the StarCore SC140," ANSC140VIT/D, 2000, 66.

[11] Wichman, S., Trombetta, R., Chiang, P., "Motion Estimation Using the ZSP500 DSP Core," Proceedings of the ISPC, March 2003.

[12] Meguerdichian, S., Koushanfar, F., Mogre, A., Petranovic, D., Potkonjak, M. "MetaCores: Design and Optimization Techniques", DAC 2001.