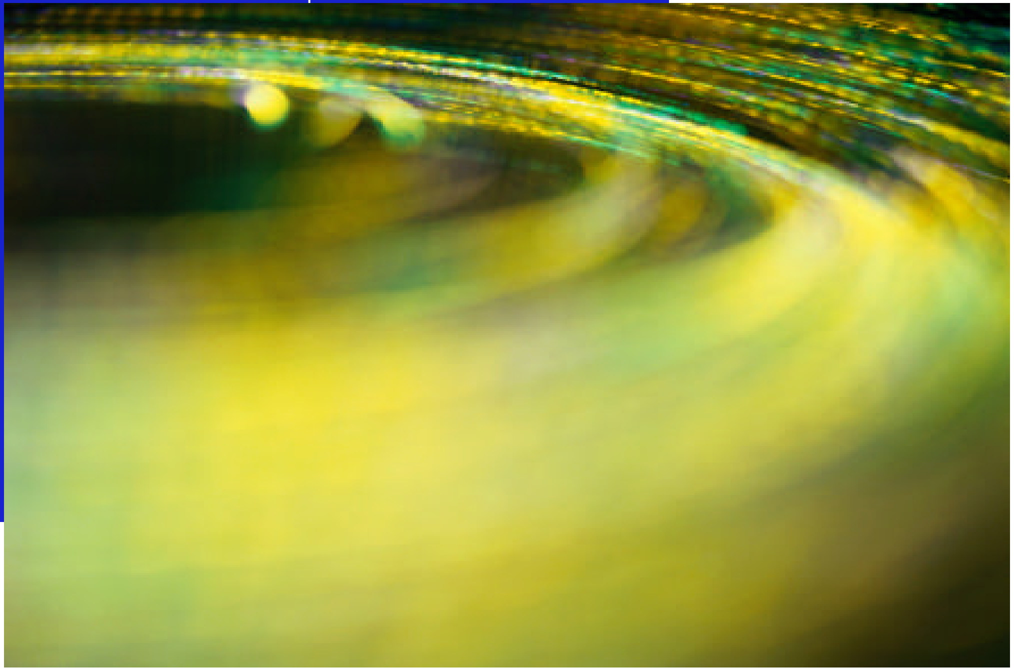
AltiVec™ Technology Training



AltiVec is a trademark of Motorola, Inc.

MOTOROLA
intelligence everywhere™

digital dna™

What is a Vector Architecture?

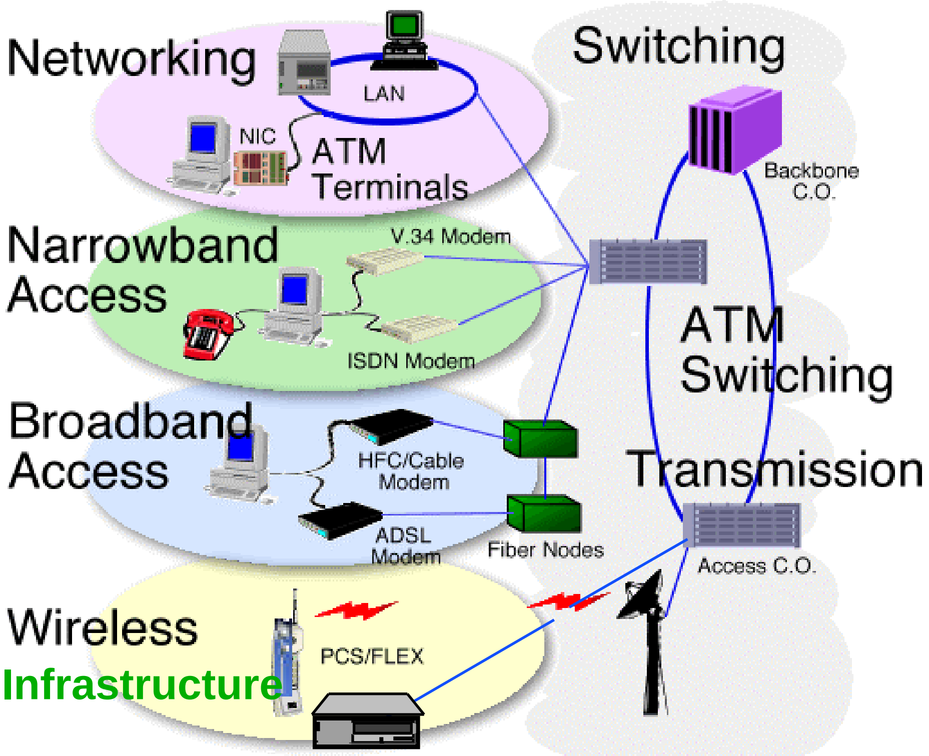
- **A** vector architecture **allows the simultaneous processing of many data items in parallel**
- **Vector architecture has roots in supercomputing, which attempted to extract large amounts of parallelism from software**
  - Massive parallel capabilities but limited data types
  - Great for computation intensive applications but not for systems requiring more diverse processing and real-time constraints
- **Operations are performed on multiple data elements by a single instruction – referred to as Single Instruction Multiple Data (SIMD) parallel processing**
- **AltiVec technology is a short vector architecture**
  - Uses 128-bit wide registers to provide 4-, 8-, or 16-way parallelism
  - Supports a wide variety of data types
- **SIMD extension to PowerPC ISA**
  - Processes multiple data streams/blocks in a single cycle
  - Common approach to accelerate processing of next-gen data types (audio, video, packet data)

Benefits of AltiVec

- **Provides a single high-performance RISC microprocessor with DSP-like compute power for controller and signal processing functions**
  - Supplements the performance-leading PowerPC architecture with a new class of execution unit
  - New vector processing engine provides for highly parallel operations, allowing for the simultaneous execution of up to 16 operations in a single clock cycle
  - Can accelerate many traditional computing and embedded processing operations with its wide datapaths and wide field operations
- **Provides product designers and customers with a new "one part – one code base" approach to product design while also providing a tremendous jump in performance**
- **Offers a programmable solution that can easily migrate via software upgrades to follow changing standards and customer requirements**
  - Because this integrated solution is still 100% compatible with the industry standard PowerPC architecture, design and support are simplified
  - Leverage PowerPC compatibility and legacy code, add AltiVec performance as you need it

AltiVec – A Solution to Many Embedded Computing Problems

- **Access Concentrators/DSLAMs**
  - ADSL and Digital Data Concentra
- **Speech Recognition**
- **Voice/Sound Processing**
- **Image and Video Processing**
- **Array Numeric Processing**
- **Basestation Processing**



AltiVec – The Best Solution to Computing Problems

- **High bandwidth data communications**
- **Realtime Continuous Speech I/O**
  - HMM, Viterbi Acceleration, Neural Algorithms
- **Soft-Modem**
  - V.34, 56K
- **3D Graphics**
  - Games, Entertainment
  - High precision CAD
- **Virtual Reality**
- **Motion Video**
  - MPEG2, MPEG4
  - H.234
- **High Fidelity Audio**
  - 3D Audio, AC-3
- **Machine Intelligence**

MOTOROLA intelligence everywhere™ | digital dna™

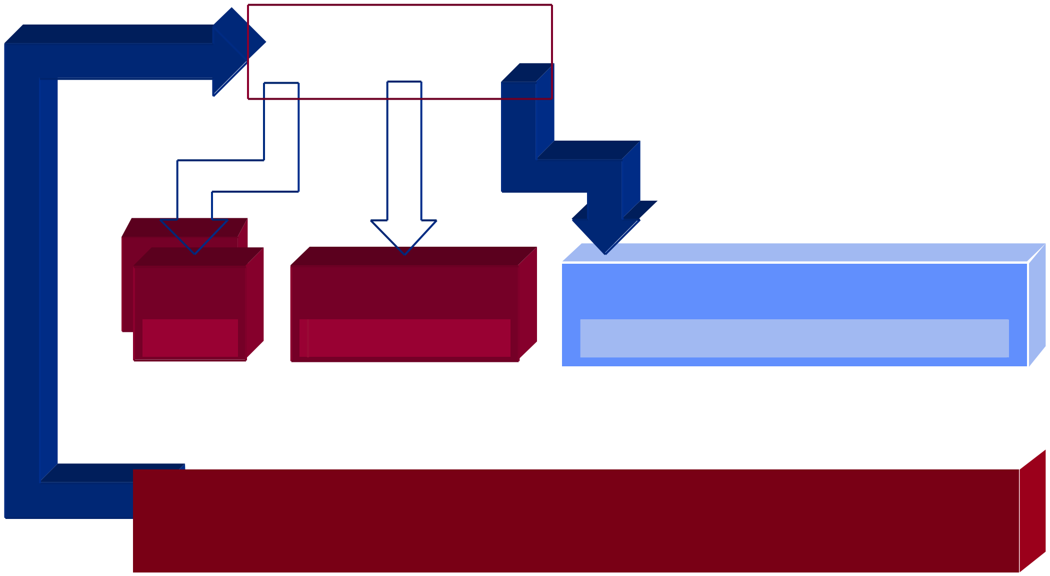
AltiVec Features

- **Provides SIMD (Single Instruction, Multiple Data) functionality for embedded applications with massive data processing needs**
- **Key elements of Motorola's AltiVec technology include:**
  - 128-bit vector execution unit with 32-entry 128-bit register file
  - Parallel processing with Vector permute unit and Vector ALU
  - 162 new instructions
  - New data types:
    - Packed byte, halfword, and word integers
    - Packed IEEE single-precision floats
  - Saturation arithmetic
- **Simplified architecture**
  - No interrupts other than data storage interrupt on loads and stores
  - No hardware unaligned access support
  - No penalty for running AltiVec and standard PowerPC instructions simultaneously
  - Streamlined architecture to facilitate efficient implementation
- **Maintains PowerPC architecture's RISC register-to-register programming model**

AltiVec Features

- **Supports parallel operation on byte, halfword, word and 128-bit operands**
  - Intra and inter-element arithmetic instructions
  - Intra and inter-element conditional instructions
  - Powerful Permute, Shift and Rotate instructions
- **Vector integer and floating-point arithmetic**
  - Data types
    - 8-, 16- and 32-bit signed- and unsigned-integer data types
    - 32-bit IEEE single-precision floating-point data ype
    - 8-, 16-, and 32-bit Boolean data types (e.g. 0xFFFF % 16-bit TRUE)
  - Modulo & saturation integer arithmetic
  - 32-bit “IEEE-default” single-precision floating-point arithmetic
    - IEEE-default exception handling
    - IEEE-default “round-to-nearest”
    - fast non-IEEE mode (e.g. denorms flushed to zero)
- **Control flow with highly flexible bit manipulation engine**
  - Compare creates field mask used by select function
  - Compare Rc bit enables setting Condition Register
    - Trivial accept/reject in 3D graphics
    - Exception detection via software polling

AltiVec’s Vector Execution Unit



- Concurrency with integer and floating-point units
- Separate, dedicated 32 128-bit vector registers
  - Larger namespace = reduced register pressure/spillage
  - Longer vector length = more data-level parallelism
  - Separate files can all be accessed by execution units in parallel
  - Deeper register files allow more sophisticated software optimizations
- No penalty for mingling integer, floating point and AltiVec technology operations

128-bit Vector Architecture

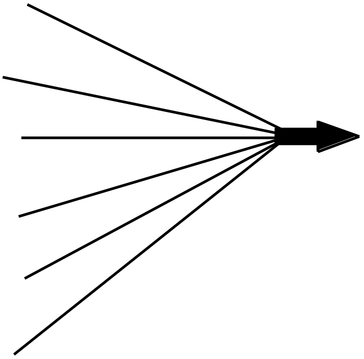
- **128-bit wide data paths between L1 cache, L2 cache, Load/Store Units and Registers**
  - Wider data paths speed save and restore operations
- **Offers SIMD processing support for**
  - 16-way parallelism for 8-bit signed and unsigned integers and characters
  - 8-way parallelism for 16-bit signed and unsigned integers
  - 4-way parallelism for 32-bit signed and unsigned integers and IEEE floating point numbers
- **Two fully pipelined independent execution units**
  - Vector Permute Unit is a highly flexible byte manipulation engine
  - Vector ALU (Arithmetic Logical Unit) performs up to 16 operations in a single clock cycle
    - Contains Vector Simple Fixed-Point, Vector Complex Fixed-Point, and Vector Floating-Point execution engines
  - Dual AltiVec instruction issue: One arithmetic, one “permute”

Sample Based Processing

SISD

AC3 - Audio Decode

```
do {
        decode ( channel 1 )
        decode ( channel 2 )
        decode ( channel 3 )
        decode ( channel 4 )
        decode ( channel 5 )
        decode ( channel 6 )
} while (Amplifier is on; step
     time)
```



SIMD

AC3 - Audio Decode

```
do {

     decode (ch 1, ch2, c3, c4, c5, c6)

} while (Amplifier is on; step time)
```

Approx 6x performance improvement.

Simplified Systems Design

- **Existing DSP based systems model is Single MPU with Multiple DSPs**
  - 2 different architectures, software code bases, and hardware types
  - DSPs limit frequency and algorithm support
  - Upgrades in performance require new hardware, and at least 1 major software change
  - **Result: High-cost upgrades, slow time-to-market, in-field hardware swap required**
- **New Model is Single or Multiple MPUs**
  - One architecture, one code base, and one hardware type
  - Very high frequency with total algorithm support
  - Performance upgrades available with only a software change
  - **Result: Low cost upgrades, fast time-to-market, no in-field hardware swap**

AltiVec Instruction Set Features

- **162 new instructions added to the PowerPC ISA**
- **4-operand, non-destructive instructions**
  - Up to three source operands and a single destination operand
  - Supports advanced “multiply-add/sum” and permute primitives
- **All instructions fully pipelined with single-cycle throughput**
  - Simple ops: 1 cycle latency
  - Compound ops: 3-4 cycle latency
  - No restriction on issue with scalar instructions
- **Enhanced cache/memory interface**
  - Software hints for data re-use probability
  - Prefetch support (stride-N access)
- **Simplified load/store architecture**
  - Simple byte, halfword, word and quadword loads & stores
  - No unaligned accesses – software-managed via permute instruction

AltiVec Benchmarks

- EEMBC AltiVec Benchmarks

MPC7455 Highest Performance Embedded Microprocessor

- **As measured by the EEMBC Benchmarks**
- **EEMBC – EDN Embedded Microprocessor Benchmarking Consortium**
- **50+ semiconductor, IP, compiler, and RTOS members**
- **Benchmark suites targeting telecommunications, networking, and other market segments**
- **Certified “out-of-the-box” and optimized results from Motorola and competitors available at EEMBC website**

The EEMBC Benchmarks

- **Multiple suites representing different market segments**
- **Multiple "kernels" or benchmarks in a suite**

– Auto/Industrial	16
– Consumer	5
– Networking	5
– Office Automation	3-5
– Telecomm	16

- **Intended to be ANSI-compliant C code and easily portable across many architectures**
- **Code is licensed from EEMBC to member companies**
- **Results are certified by EEMBC Certification Labs (ECL)**
- **Results cannot be publically disclosed without certification**
- **EEMBC cannot publically disclose without vendor's permission**

Are EEMBC Results Relevant?

- Benchmark results are only useful if they are representative of a customer's application.
- However, the next slide reports an assessment by Motorola Labs, who did have access to the EEMBC source code, on the relevance of the EEMBC networking benchmarks to a router application.
- The resulting packet-per-second rating is higher than customer's can expect in the real world (we've heard customers are achieving 1 million pkts/sec with the MPC7455 at 933MHz) because the EEMBC benchmarks do not include the actual I/O transfers, i.e. the packets are compiled into the benchmark and pre-loaded into memory.
- Nevertheless the resulting packets per second metric is valuable for comparing performance between processors that are running the EEMBC code.
- Because the EEMBC packet flow and route lookup show high L1 cache hit rates for the 512KB buffer and small route lookup table, it is reasonable to scale performance on these benchmarks with core frequency for processors that have 32KB I/D L1 caches.

Comparing Performance on Networking

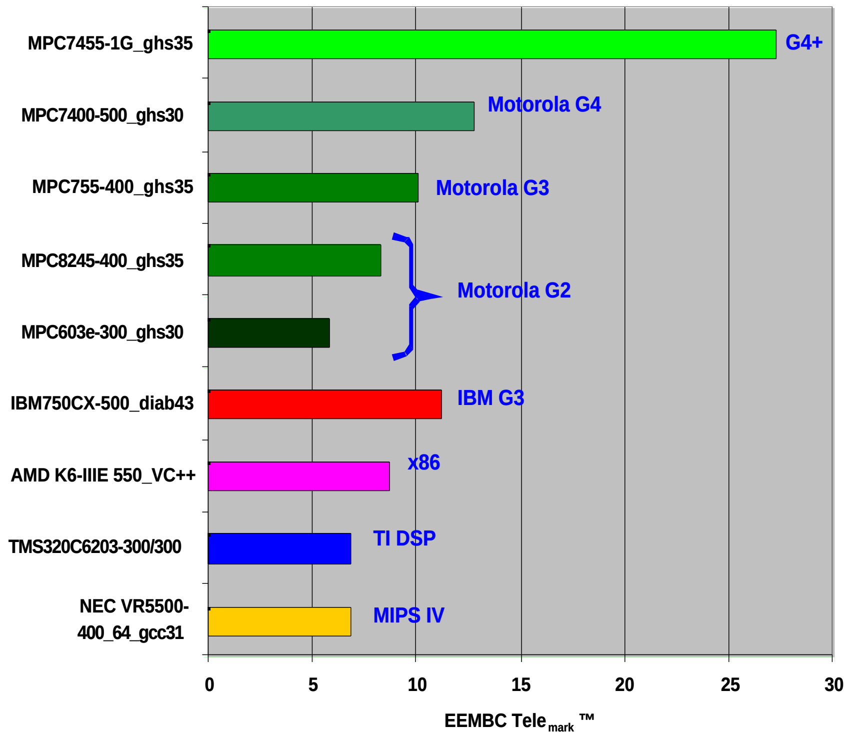
	MPC7455	AMD K6	NECVR5000	IDT6457
Core Frequency	1GHz	550MHz	250MHz	250MHz
Bus Frequency	133MHz	100MHz	100MHz	50MHz
L1	32K/32K 8w	32K/32K 8w	32K/32K	32K/32K
L2	256K 8w	256K		
Compiler	GHS	Diab	GHS	Algorithmic
Native data type	32	32	64	64
Reported Iter/Sec:				
ospf	14,662	10,157	3,473	2,882
Route lookup	4,442	2,184	837	974
Packet Flow – 512KB	31,335	9,246	3,268	4,177
-1MB	16,428	4,649	1,499	1,937
-2MB	7,407	2,277	685	773
Equivalent				
Packets per second				
Route lookup	8,884,000	4,368,000	1,674,000	1,948,000
Packet Flow	20,681,100	6,102,360	2,156,880	2,756,820
Combined	6,225,426	2,545,768	942,503	1,141,443
Scaled to 1GHz				
Route lookup	8,884,000	7,941,818	6,696,000	7,792,000
Packet Flow	20,681,100	11,095,200	8,627,520	11,027,280
Combined	6,225,426	4,628,669	3,770,013	4,565,773

Packet flow transfers ~775 bytes per packet per a subset of rfc1812 without lookup. Route lookup does 2000 lookups in a 200 entry trie per iteration.

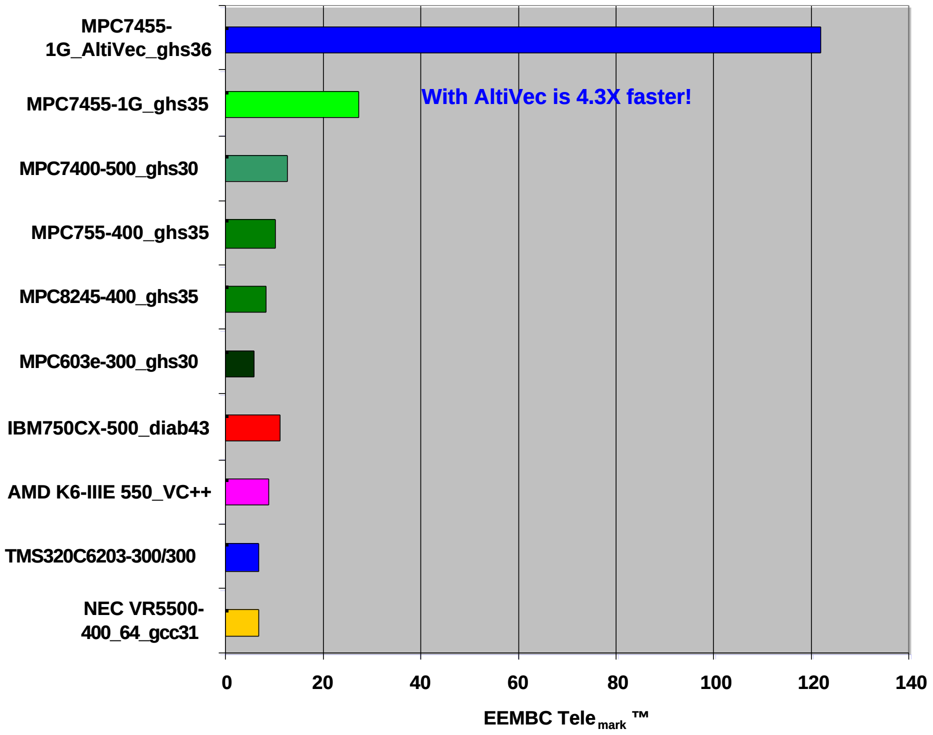
MOTOROLA
intelligence everywhere

digital dna

EEMBC Results: Telecom - Production Silicon



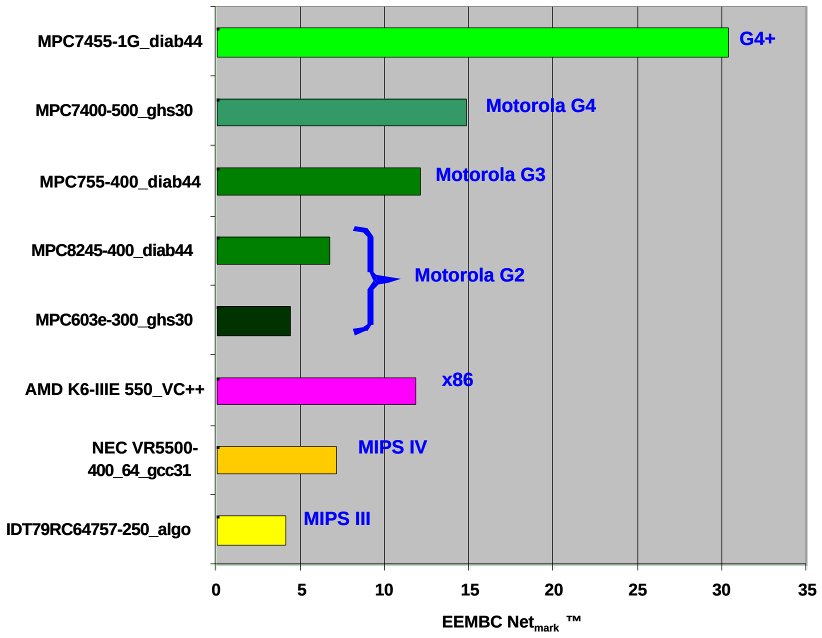
EEMBC Results: Telecom with AltiVec



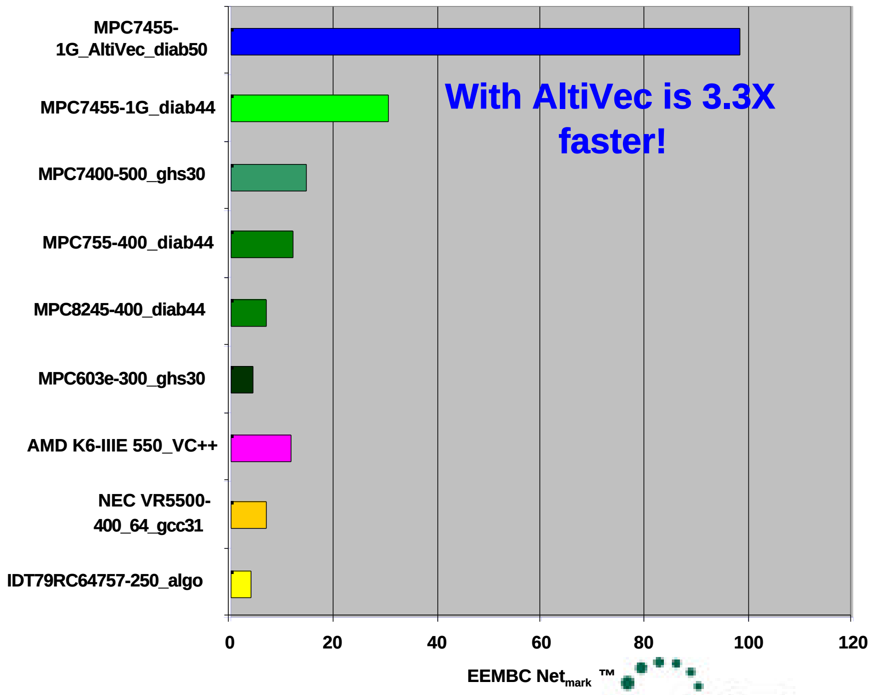
EMBC

An Industry Standard Benchmark Consortium

EEMBC Results: Networking



EEMBC Results: Networking with AltiVec



With AltiVec is 3.3X faster!

eCL Certified!

MOTOROLA intelligence everywhere

digital dna

AltVec Product Offering

- **Motorola's Product Offering for AltiVec is to provide software engineers ease of use access to the AltiVec technology which is embedded in its G4 PowerPC microprocessors.**
  - – Libraries - customers may obtain AltiVec libraries at Motorola web site www.motorola.com/Altivec ie: memcpy, strgcmpr,
    - These libraries may be linked via standard 3rd party compiliers
    - The libraries contain elements which have been shown to be effective in the networking and telecom benchmark suites of EEMBC
  - – Application notes - software application notes of specific application problems are provided for customer evaluation. These the software code included in these notes may be incorporated into customer's specific code. Ie. Fft, dct, Invert, At Motorola web side

    www.motorola.com/Altivec
  - – Customer code – on a limited basis, Motorola will provide software engineers to assist customers in the enabling their code to take advantage of AltiVec.

In summary . . .

- **AltiVec™ Technology provides a robust instruction set**
  - In terms of operations
  - In terms of data types and sizes
  - In terms of other options (such as saturation, data movement)
  - Allows a streamlined hardware implementation (cycle time, latency, throughput)
- **AltiVec enables a broad range of embedded and computing applications**

AltiVec Vectorizing Tools and Resources

- **Motorola Team Background:**
  - Primarily Media algorithms (video, voice, etc.); some encryption
  - Compiler optimizations to improve performance (gcc, etc)

- **Suggested AltiVec Resources:**
  - Motorola AltiVec Programming Interface Manual (PIM) & Environment Manual (PEM).
  - Motorola 7450 Software Optimization Guide
  - Motorola web site and internal application notes
  - Apple's web site
  - Simdtech.org
  - Mercury's VSIPL LITE routines

- **Possible Algorithm Development & Tuning Environments**
  - 3rd party preprocessors and compilers (VAST, Diab, Metrowerks…)
  - gcc 3.1+ cross compilers (Motorola/Red Hat effort…)
  - Apple PowerMac with Linux PPC + timing routines
  - MVP platform?

Algorithms for Discussion

Selected Algorithms Examination

- **Encryption**
  - Rijndael/AES
  - DES/3DES
  - Kasumi

- **3G Symbol Rate Functions**
  - CRC
  - Convolutional Encoding
  - Viterbi Decoding
  - Turbo

AES/Rijndael

- **History:**
  - Compared Reference Implementation to Optimized Lookup Table
  - Optimized Lookup Table version provides on the order of 10X speedup compared to scalar reference version of algorithm.
  - Some preliminary investigation of translation to AltiVec in ‘01

- **Sources:**
  - Brian Gladman’s open source optimized code version used as starting point
  - Referenced Helger Lipmaa’s published comparisons

  - Note: Rickard Holmberg (02/02) claims 94 + 207 N cycles for N 128 bit blocks.

    =>Better than scalar (366 cycles per 128 bit block)

- **AltiVec Vectorization Efforts:**
  - Preliminary AltiVec version extensions to Gladman’s work of reference AES got to within 17% of optimized lookup table version
  - There appears to be room in the scalar code for further parallelization (Scalar table lookup + AltiVec)
  - The pure AltiVec implementation has further handcoding opportunities as well.

CRC

- **History:**
  - Background:
    - Developing library element for Q1 offering
    - Software implementation typically uses table lookup algorithms.
  - Sources:
    - “A New Parallel Algorithm for CRC Generation” (Joshi, Dubey, Kaplan)

- **AltiVec Vectorization Efforts:**
  - Estimate improvement of 4.57x over classical table lookup for 32 bit CRC using AltiVec version; from paper.
  - Improvement uses table lookup and perms.

Convolutional Encoding

- **History:**
  - Background:
    - PowerPC application note and preliminary reference example.
  - Sources:
    - TIA/EIA IS136; GSM CC(2,1,5)
    - AltiVec reference example
    - EEMBC telecommunication using constraint length 3.

- **AltiVec Vectorization Efforts:**
  - For rate 1/2 constraint length 3, 4, and 5 encoding shows 28-30X speedup.
  - Speed up obtained using perm, shifts
  - Data packing may yield further improvements

Viterbi

- **History:**
  - Background:
    - PowerPC application discussion and preliminary reference examples
  - Sources:
    - Application discussion – EEMBC Viterbi TIA/EIA IS136
    - AltiVec reference example - Soft decision viterbi decoding GSM CC(2,1,5)

- **AltiVec Vectorization Efforts:**
  - Viterbi IS136: 5X speedup seen from scalar code
  - Improvements realized by parallelizing branch metric calculations

Backup Material

- Convolutional Encoder presentation
- Viterbi Decoder presentation
- Misc. AltiVec Algorithm Analysis
- APU future modifications discussion

Convolutional Encoder Benchmark
Optimization thru Vectorization

AltiVec Team

PowerPC Applications

April 2000

Convolutional Encoder Definition

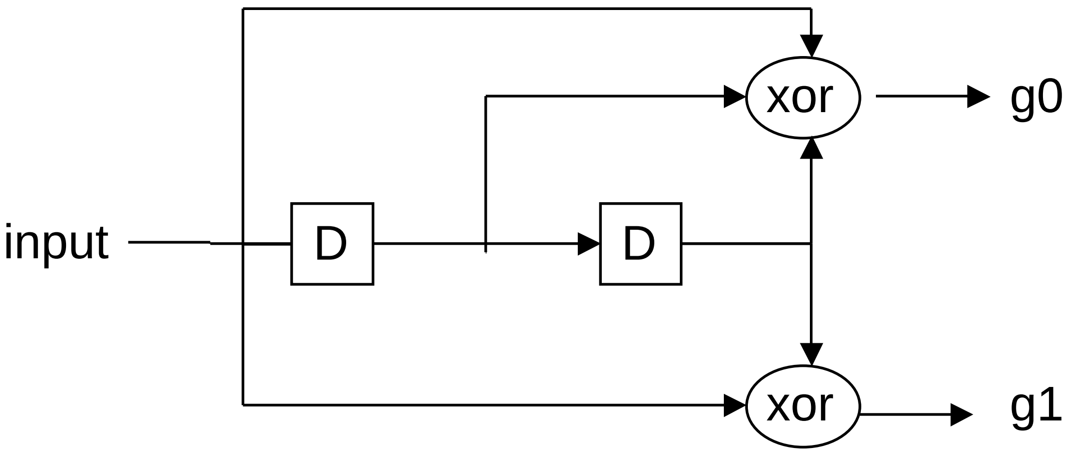
Convolutional Encoder is a way of encoding digital data before transmission through **noisy or error-prone** channels.

During encoding, **k** input bits are mapped to **n** output bits to give a rate **k/n** coded bitstream. The encoder consists of a shift register of **kL** stages, where **L** is described as the constraint length of the code.

Common application is in cellular phones and other communication systems such IS136 (Interim Standard 136) and GSM CC(2,1,5) (Global System for Mobile Systems).

Viterbi Decoder decodes the Convolutionally encoded bit stream.

Convolutional Encoding



D is a delay unit. Number of Delays+1 = **Constraint Length**

g0 and g1 are represented as **Generator Polynomials**

g0 = input $_{t=0}$ (xor) input $_{t=-1}$ (xor) input $_{t=-2}$ = $1 + D + D^2$
g1 = input $_{t=0}$ (xor) input $_{t=-2}$ = $1 + D^2$

For each input bit there are two output bits (g0g1) hence this encoder is **Half Rate Encoder**

Convolutional Encoding (Example)

Delayed inputs are generated using **Shift Register**

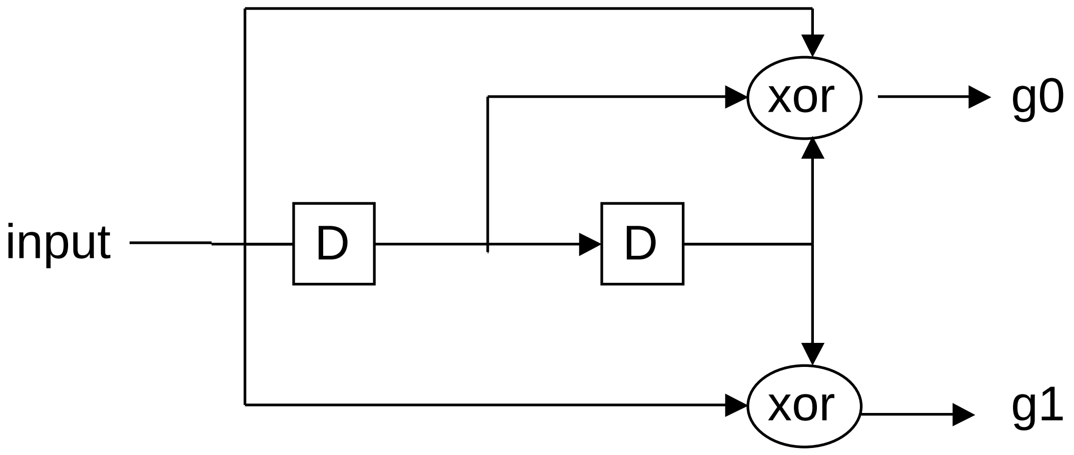
Example: $g0 = D^0 \otimes D^1 \otimes D^2$ $g1 = D^0 \otimes D^2$

input = 01001110

Input	**D^1**	**D^2**	**g0**	**g1**	**output**
0	0	0	0	0	00
1	0	0	1	1	11
0	1	0	1	0	10
0	0	1	1	1	11
1	0	0	1	1	11
1	1	0	0	1	01
1	1	1	1	0	10
0	1	1	0	1	01
	0	1			
		0			

Convolutional Encoding (EEMBC)

In **EEMBC**, g0 and g1 are not hard coded inside the Encoder. Instead, the Encoder takes in a **Code Matrix** for the calculation**.**



Code Matrix

$$\begin{array}{cc} g0 & g1 \\ \hline 1 & 1 \\ 1 & 0 \\ 1 & 1 \end{array}$$

Number of Rows = Constraint Length = 3
Number of Columns= 1/Rate = Number of **Code Vectors** = 2

Convolutional Encoder Algorithm(EEMBC)

The EEMBC Convolutional Encoder takes in the following inputs:

- A stream (array) of data input bits (bits are represented as unsigned char)
- Constraint Length (signed short)
- Data size (signed short)
- Number of Code Vectors (signed short)
- Code matrix 2 dimensional array of bits (chars)
  - Number of Rows = Constraint Length
  - Number of Columns = Number of Code Vectors

Internally the Algorithm uses

- A shift register of length equal to Constraint Length

The Algorithm outputs the following:

- A stream (array) of data output bits of size = Number of Code Vectors x Data size

Convolutional Encoder Algorithm (cont.)

- **Assume the following inputs**

0	1	0	0	1	1	1	0

Data input of size 8

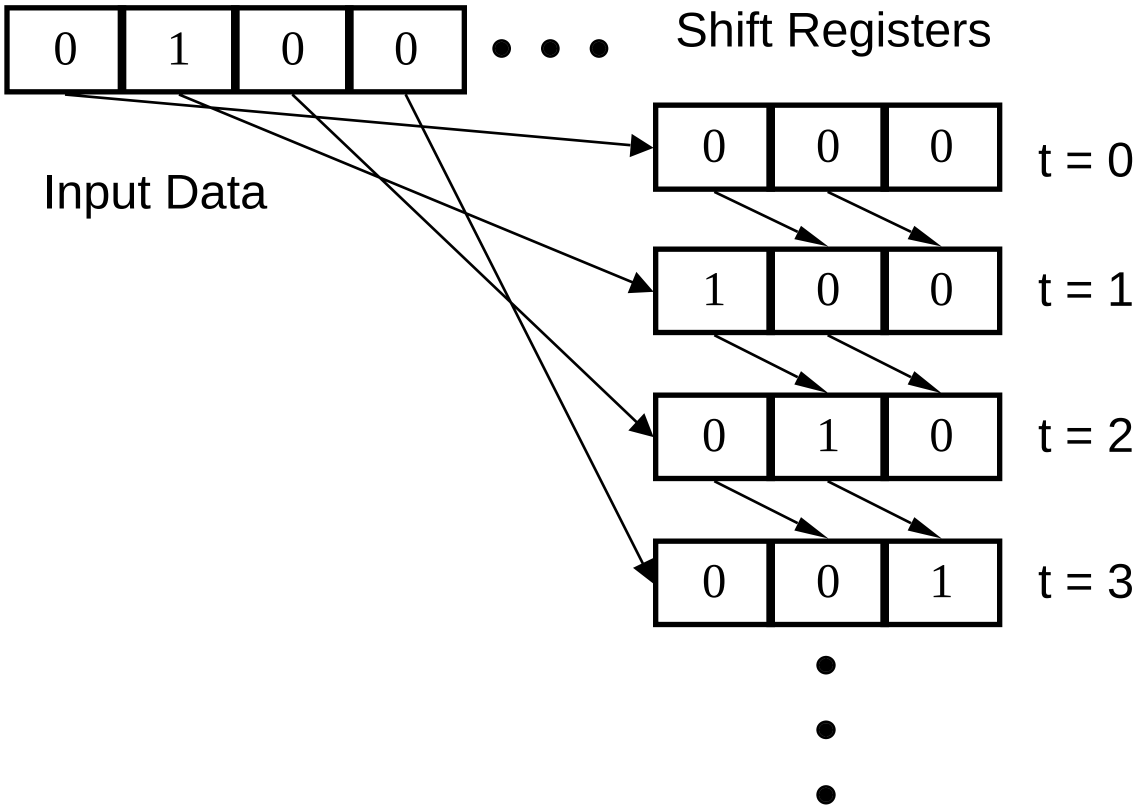
1	1
1	0
1	1

Code Matrix of
Number of Column = Code Vector size = 2
Number of Row = Constraint Length = 3

These inputs will be referred to as DataInput[i] where i = index starts at 0 and CodeMatrix[i][j] where i = row and j = column. i and j start at 0.

Convolutional Encoder Algorithm (cont.)

Shift Register of Size = Constraint Length = 3.

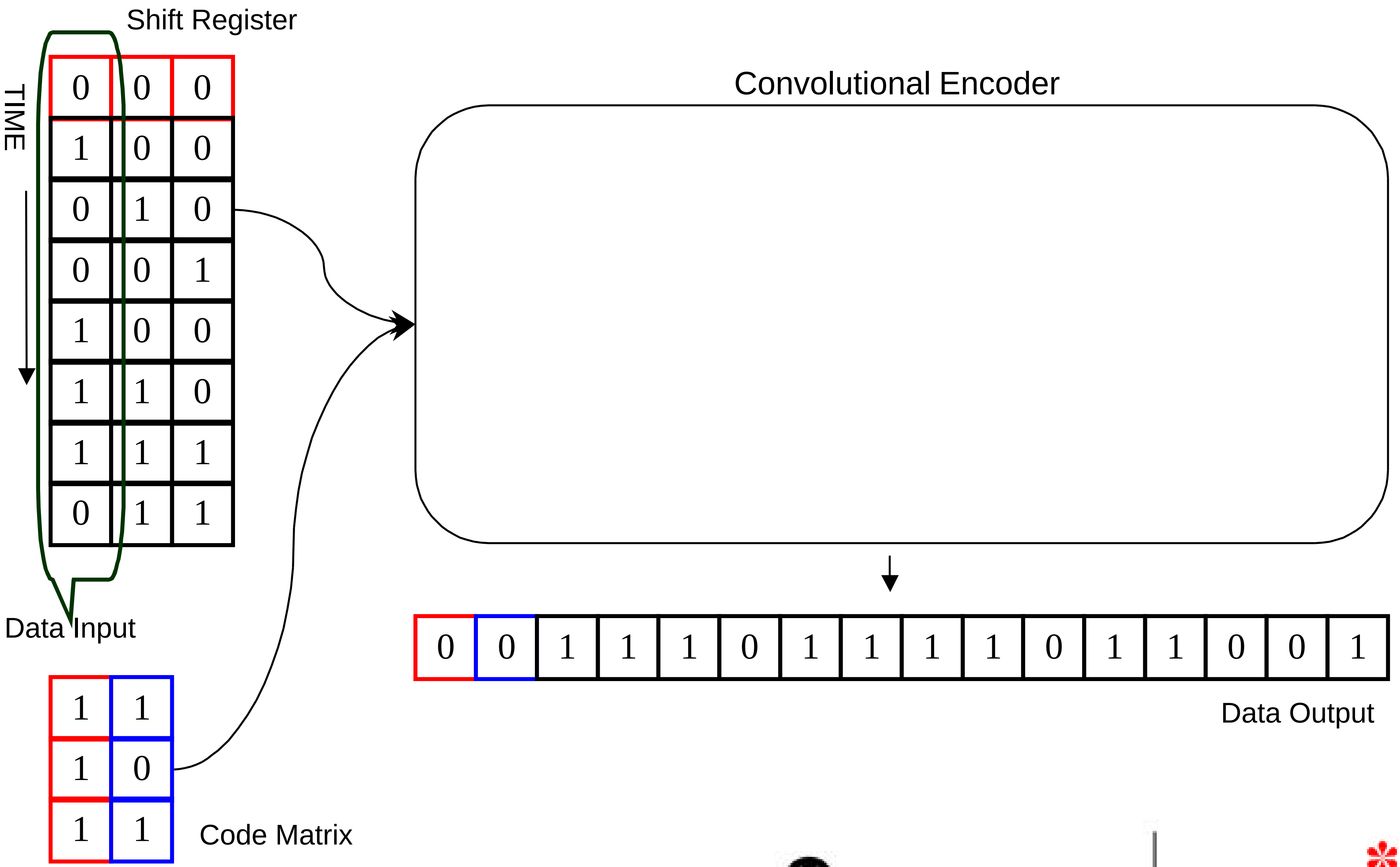


Convolutional Encoder Algorithm (cont.)

The Data Output (of size = Number of Code Vectors * Data Size = 2*8 = 16) which originally contains all 0’s, is determined in the following manner

```
DOIndex = 0;
for(DIndex = 0; DIndex < DataSize; DIndex++)
{
   for(CVIndex = 0; CVIndex < Number of Code Vectors; CVIndex++)
   {
      for(SRIndex = 0; SRIndex < Constraint Length; SRIndex++)
       {
         if( Code Matrix[SRIndex][CVIndex] )
           DataOutput[DOIndex] = DataOutput[DOIndex] ⊗
                                              ShiftRegister[DIndex][SRIndex];
       }
       DOIndex++;
   }
}
```

Convolutional Encoder Algorithm (cont.)



Shift Register

TIME

0	0	0
1	0	0
0	1	0
0	0	1
1	0	0
1	1	0
1	1	1
0	1	1

Data Input

Convolutional Encoder

0	0	1	1	1	0	1	1	1	1	0	1	1	0	0	1

Data Output

1	1
1	0
1	1

Code Matrix

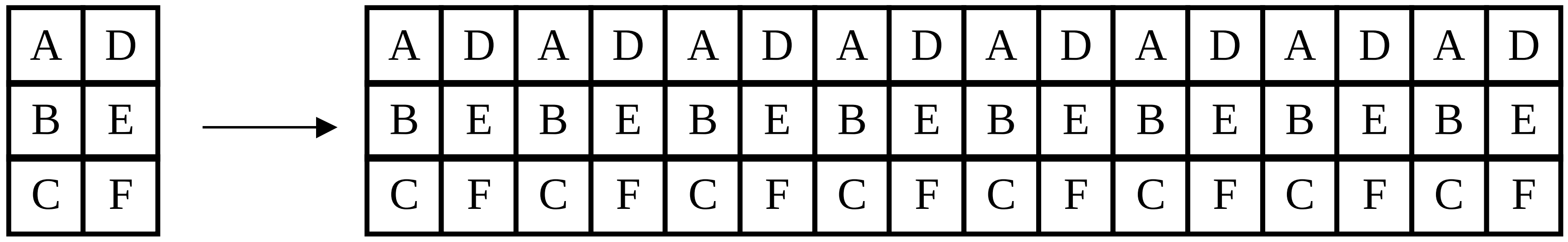
Vectorization Concept

The main idea of vectorization is to do many basic Convolutional Encoder calculations in parallel.

From previous slide for all processes (for n=0, 1, ... DataSize):

- Process 2n uses Column 0 of the Code Matrix and Row n of Shift Register and writes to Data Output[2n]
- Process 2n+1 uses Column 1 of the Code Matrix and Row n of Shift Register and writes to Data Output[2n+1]

Vectorization (cont.)



A	D
B	E
C	F

Code Matrix
(Scalar)

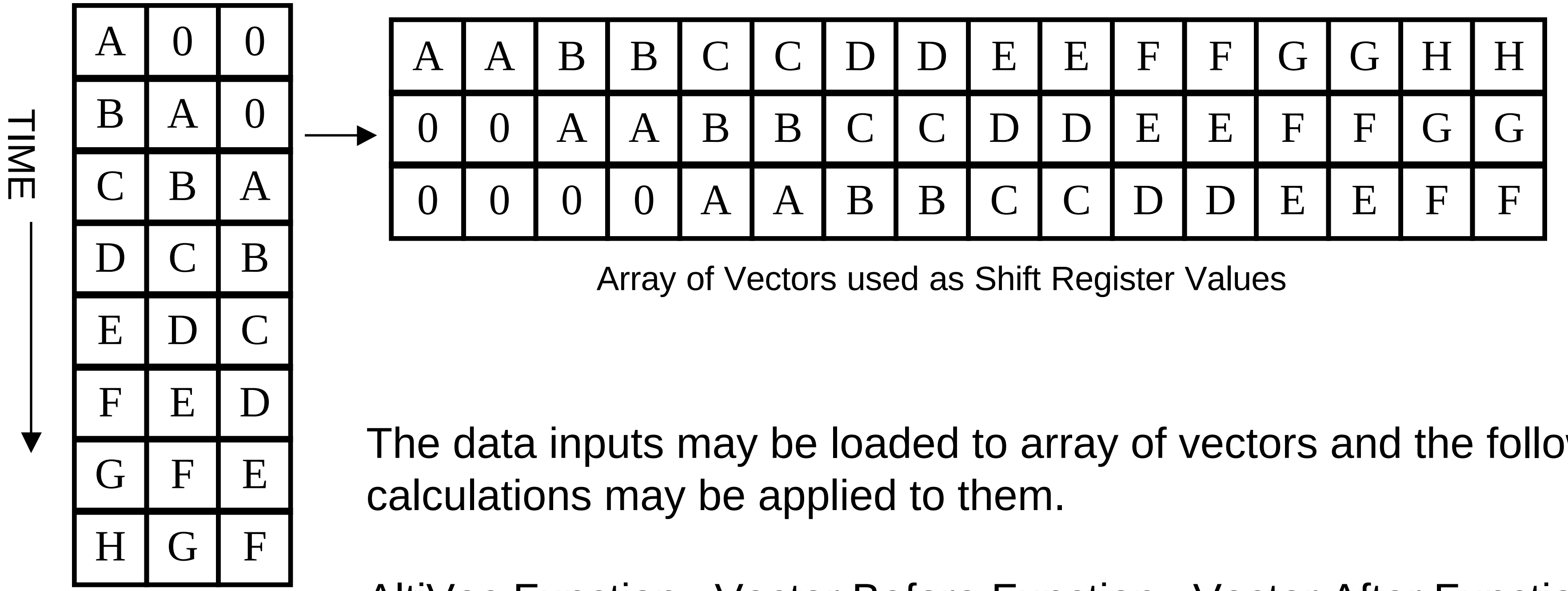
A	D	A	D	A	D	A	D	A	D	A	D	A	D	A	D
B	E	B	E	B	E	B	E	B	E	B	E	B	E	B	E
C	F	C	F	C	F	C	F	C	F	C	F	C	F	C	F

Array of Vectors used as Code Matrix

The Code Matrix is a small 2 dimensional constant value. Similarly the array of vectors may be declared as a constant and no AltiVec functions may be needed to load the values.

Vectorization (cont.)

For convenience all shifting for the Shift Register will be done ahead of time and the array of vectors that will be used will be as follows:



A	0	0
B	A	0
C	B	A
D	C	B
E	D	C
F	E	D
G	F	E
H	G	F

Shift Register (scalar)

A	A	B	B	C	C	D	D	E	E	F	F	G	G	H	H
0	0	A	A	B	B	C	C	D	D	E	E	F	F	G	G
0	0	0	0	A	A	B	B	C	C	D	D	E	E	F	F

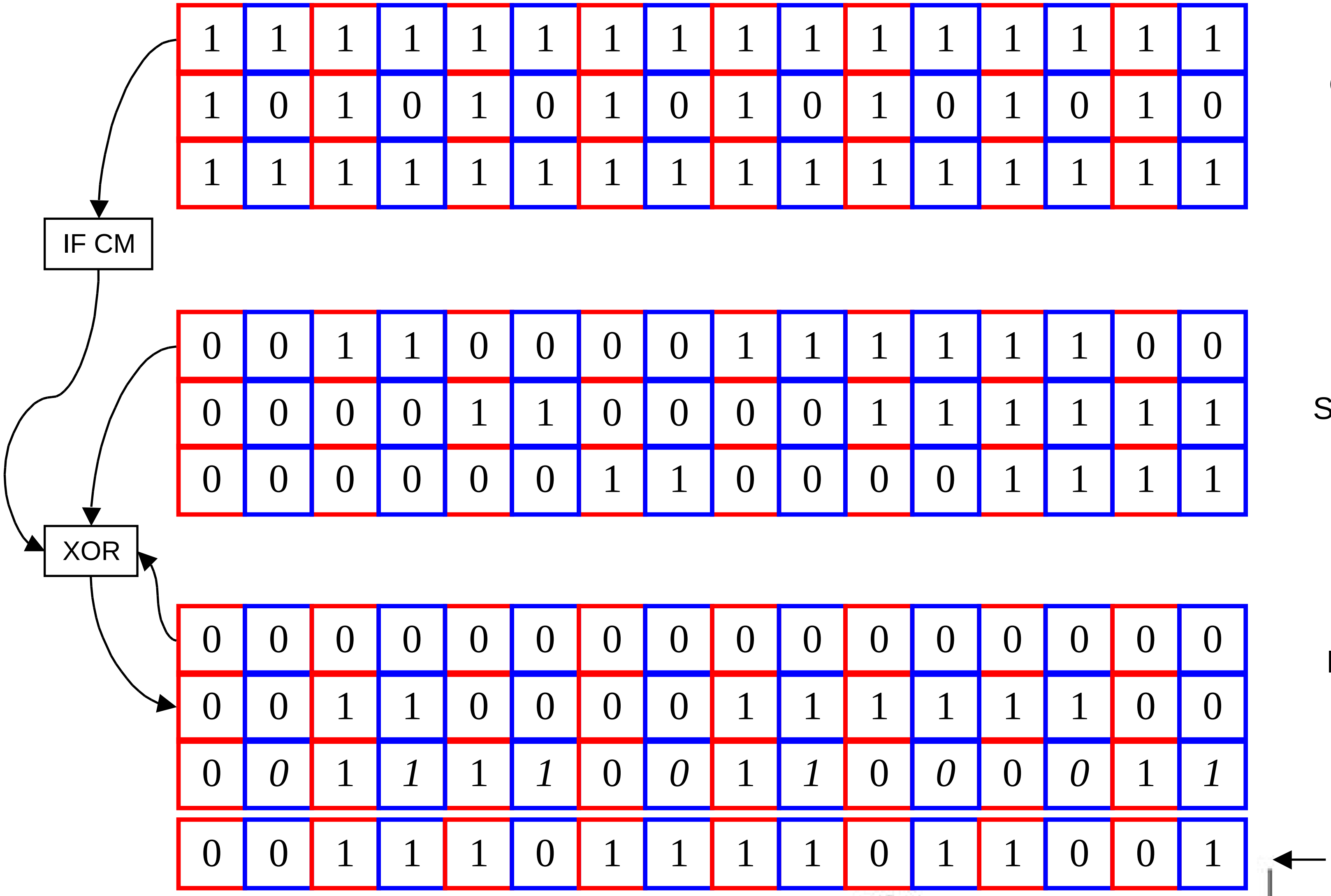
Array of Vectors used as Shift Register Values

The data inputs may be loaded to array of vectors and the following calculations may be applied to them.

AltiVec Function	Vector Before Function	Vector After Function
vec_perm	A,B,C,D,E,F,G,H,A..	A,A,B,B,C,C,D,D,E,...
vec_sld	A,A,B,B,C,C,D,D,E..	0,0,A,A,B,B,C,C,D,....
vec_sld	0,0,A,A,B,B,C,C,D,...	0,0,0,0,A,A,B,B,C,...

repeat vec_sld *(Constraint Length - 1)* times ...

Vectorization (cont.)



Code Matrix Vectors

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Shift Register Vectors

0	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	0	1	1	0	0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	1	1	0	0	0	0	1	1	1	1

Intermediate Values

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0
0	*0*	1	*1*	1	*1*	0	*0*	1	*1*	0	*0*	0	*0*	1	*1*

Data Output

0	0	1	1	1	0	1	1	1	1	0	1	1	0	0	1

Summary and Speedups

The vectorization process yields a good speedup because the algorithm lends itself easily to “parallel processing”. The following table shows the speedups gained (tested on a 400MHz G4 running Linux). The numbers are in iterations per second where an iteration is the time the algorithm takes to process 512 bits (represented as chars) of data.

Possible further improvements may be done by packing the char represented bits into actual bit stream. i.e. 0,1,0,0,0,0,1,0 (8 chars) can be packed into 0x42 (1 char). Packing may lead into a tremendous speedup value.

Constraint Length	Scalar Code	Vector Code	Speedup
3	8928	255,232	28.6
4	7462	211864	28.4
5	6024	181028	30.0

Viterbi Decoder Benchmark Optimization thru Vectorization

AltiVec Team

PowerPC Applications

MOTOROLA
intelligence everywhere™

digital dna™

Viterbi Decoder Definition

Viterbi Decoder is a **maximum likelihood** decoder for the Convolutional Encoder.

Viterbi Decoder was first presented by Viterbi, A .J. and Forney, G.D.Jr, in 1967 as "Asymptotically Optimum Decoding Algorithm”. They named that algorithm as Viterbi Algorithm or VA.

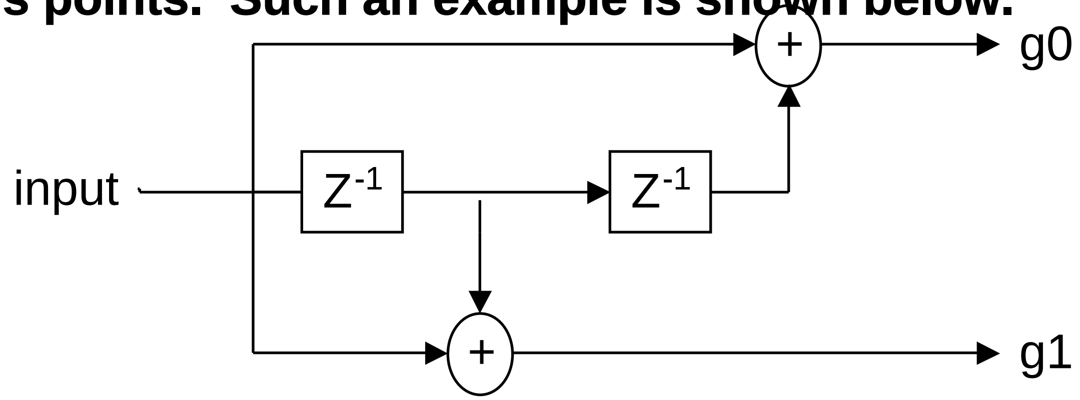
Convolutional Encoding and Viterbi Decoding is used for transmission and reception of data in a noisy and error-prone medium.

Viterbi Decoder uses state transition tables to approximate data output.

Convolutional Encoding

- **Convolutional Encoders work by convolving the input bit with previous**
- **uncoded bits. The input bits are input to a shift register with taps at**
- **various points. Such an example is shown below.**

CIRCUIT



GENERATOR POLYNOMIALS g0 = 1 + D^2; g1= 1 +D; where D = Delay

CODE MATRIX

$$\begin{matrix}1 & 1\\0 & 1\\1 & 0\end{matrix}$$

CONSTRAINT LENGTH 3

RATE 1/2

Convolutional Encoding

The encoded output would be the g0 and g1 values for each input.

For input: 0111010101100000

input	0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 0
D	0 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 0
D^2	0 0 0 1 1 1 0 1 0 1 0 1 1 0 0 0 0 0
$g0=1+D^2$	0 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0
$g1=1+D$	0 1 0 0 1 1 1 1 1 1 0 1 0 0 0 0
encoded output	00111000110101010101101110000000

Viterbi Decoding

There are two types of Viterbi Decoders:

Hard Decision Decoders : take in the actual output of the Convolutional encoder. In the previous example this would be the stream of 32 bits. Each g0g1 values are known as symbols. Therefore, the previous 32 bits are equal to 16 symbols.

Soft Decision Decoders : take in a multiple bit representation for each bit output of the encoder. For example, each output can be represented as a 3 bit value where 111 represents a strong 0 and 000 represents a strong 1, or vice versa. These soft-Decision values typically come from a ***Viterbi equalizer.***
A symbol is a set of these soft decision values for g0g1. For the previous example a symbol would have six bits or two soft bits.
e.g. 001110... => 111 111 000 000 000 111...

A group of Symbols identified by a certain property (such as a chunk of data) are known as Frames.

Viterbi Decoders

Examples of Viterbi Decoders include:

TIA/EIA IS136 Decoder:

TIA/EIA stands for Telecommunications Industry/Electronic Industries Association. IS stands for Interim Standard

$g0 = 1 + D + D^3 + D^5$

$g1 = 1 + D^2 + D^3 + D^4 + D^5$

Soft Decision Decoder where Strong 0 = 111 Strong 1 = 000

Used in Cellular Phones, EEMBC Benchmark

GSM CC (2,1,5) Decoder:

GSM stands for Global System for Mobile Systems

$g0 = 1 + D^3 + D^4$

$g1 = 1 + D + D^3 + D^4$

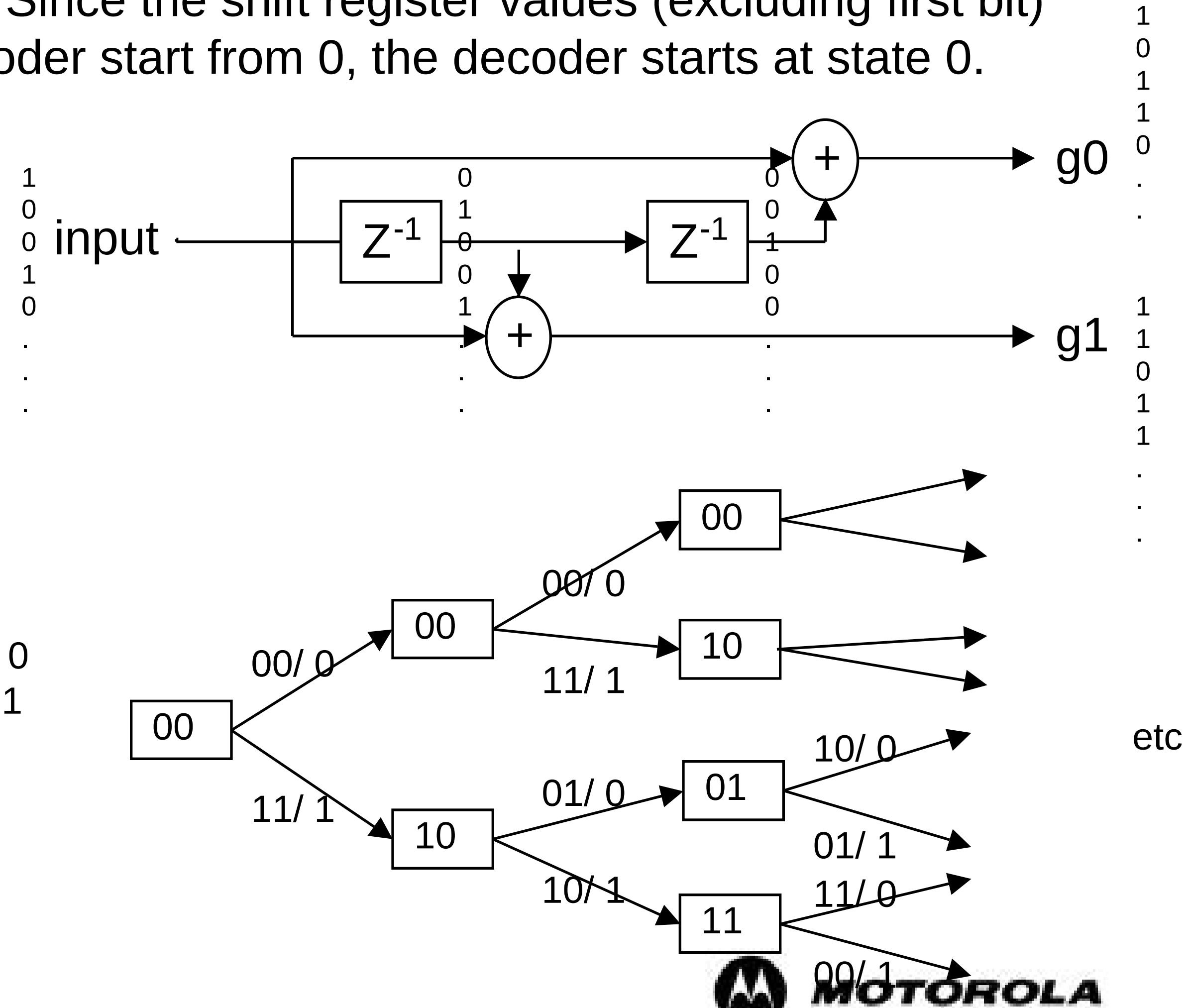
Soft Decision where Strong 0 = 00000000 Strong 1 = 11111111

Also used in Cellular Phones.

(more info at http://cctpwww.cityu.edu.hk/network/l3_wireless.htm)

Viterbi Decoding

The decoder goes through some states to figure out the original data encoded by the encoder. States represent the possible bit values in the encoder's shift registers. Since the shift register values (excluding first bit) in the encoder start from 0, the decoder starts at state 0.



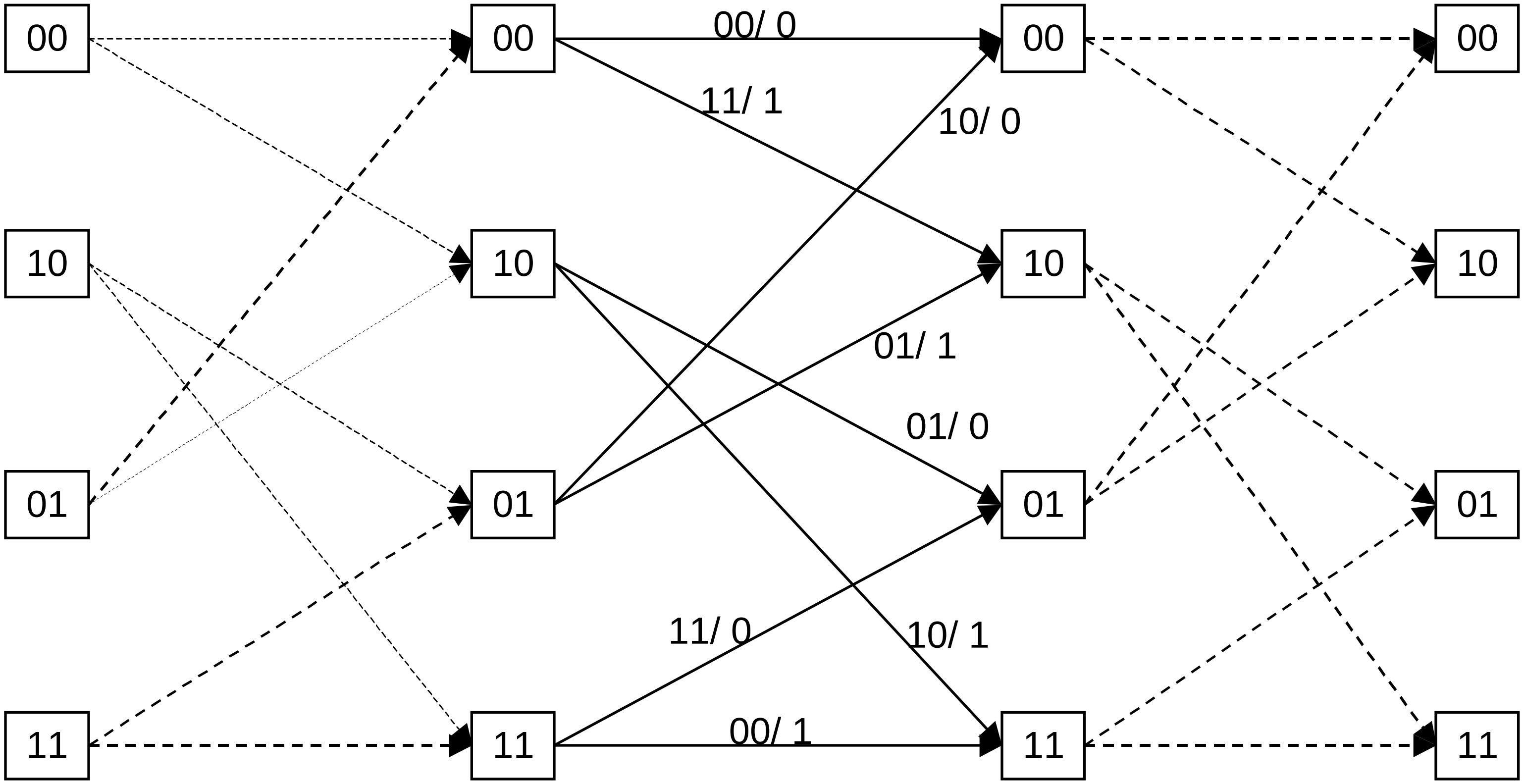
ENCODER

upper arrows = 0
lower arrows = 1

etc ..

DECODER

Viterbi Decoder



Viterbi Decoder

At each interval the algorithm calculates and accumulates the **metric** (the difference between expected value and the actual symbol received) for each state. After processing a group of symbols (a **frame**), the algorithm traces back the path with the lowest metric.

For Hard Decision Decoders Hamming Distance is used while for Soft Decision Decoders Euclidean Distance is used. For numbers:

$a = a_1a_2a_3a_4...a_n$ and
$b = b_1b_2b_3b_4...b_n$

Hamming Distance(a, b) ≈ $(a_1 - b_1) + (a_2 - b_2) + (a_3 - b_3) + (a_4 - b_4) + .. + (a_n - b_n)$

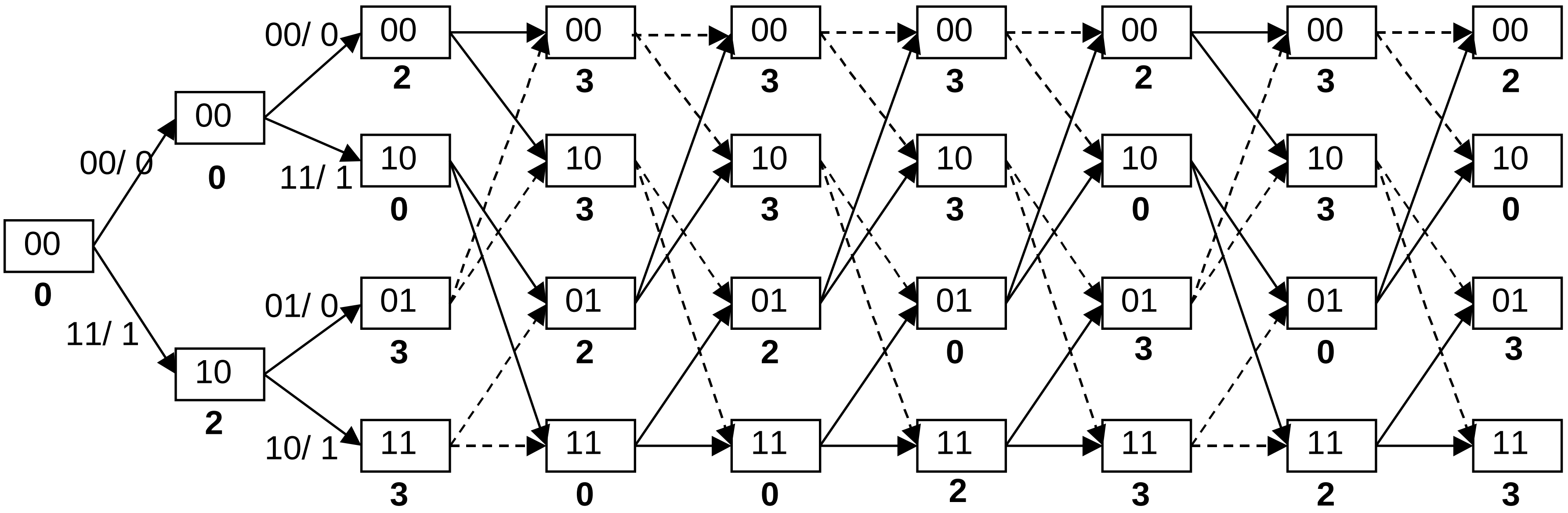
Euclidean Distance(a, b) ≈ $(a_1a_2a_3a_4...a_n) - (b_1b_2b_3b_4...b_n)$

e.g HD(01,11) < HD(01,10) but ED(01,11) > ED(01,10)

Viterbi Decoding

upper arrows = 0
lower arrows = 1

00 11 10 00 11 01 01 01

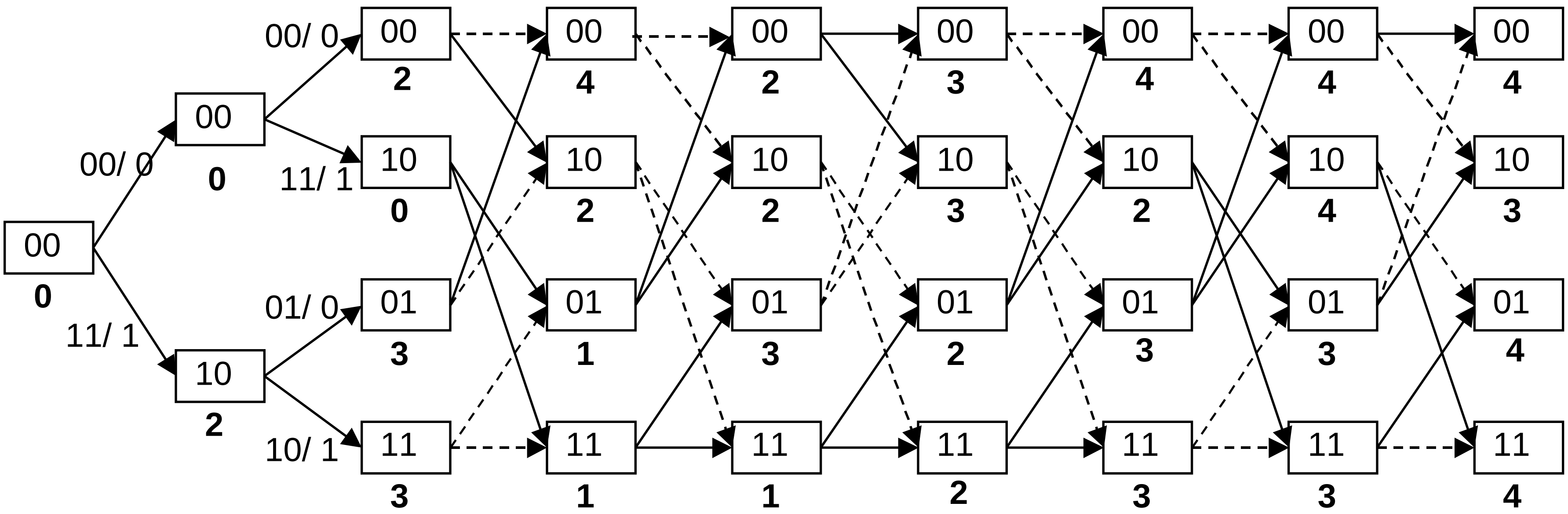


0 1 1 1 0 1 0 1

Viterbi Decoding

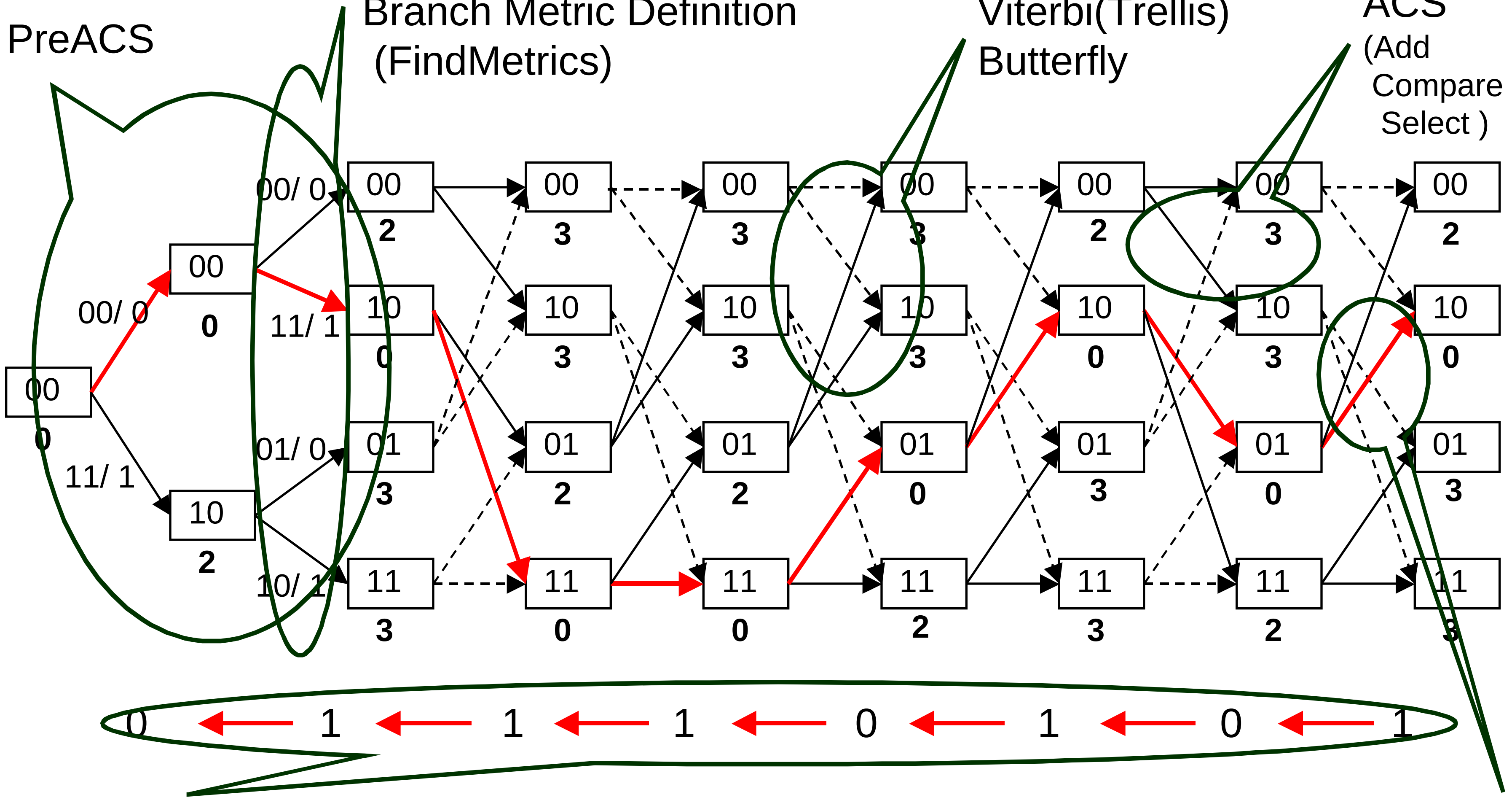
upper arrows = 0
lower arrows = 1

00 11 *11* 00 *01* 01 *11* 01



0 1 1 1 0 1 0 1

Viterbi Decoding



PreACS

Branch Metric Definition
(FindMetrics)

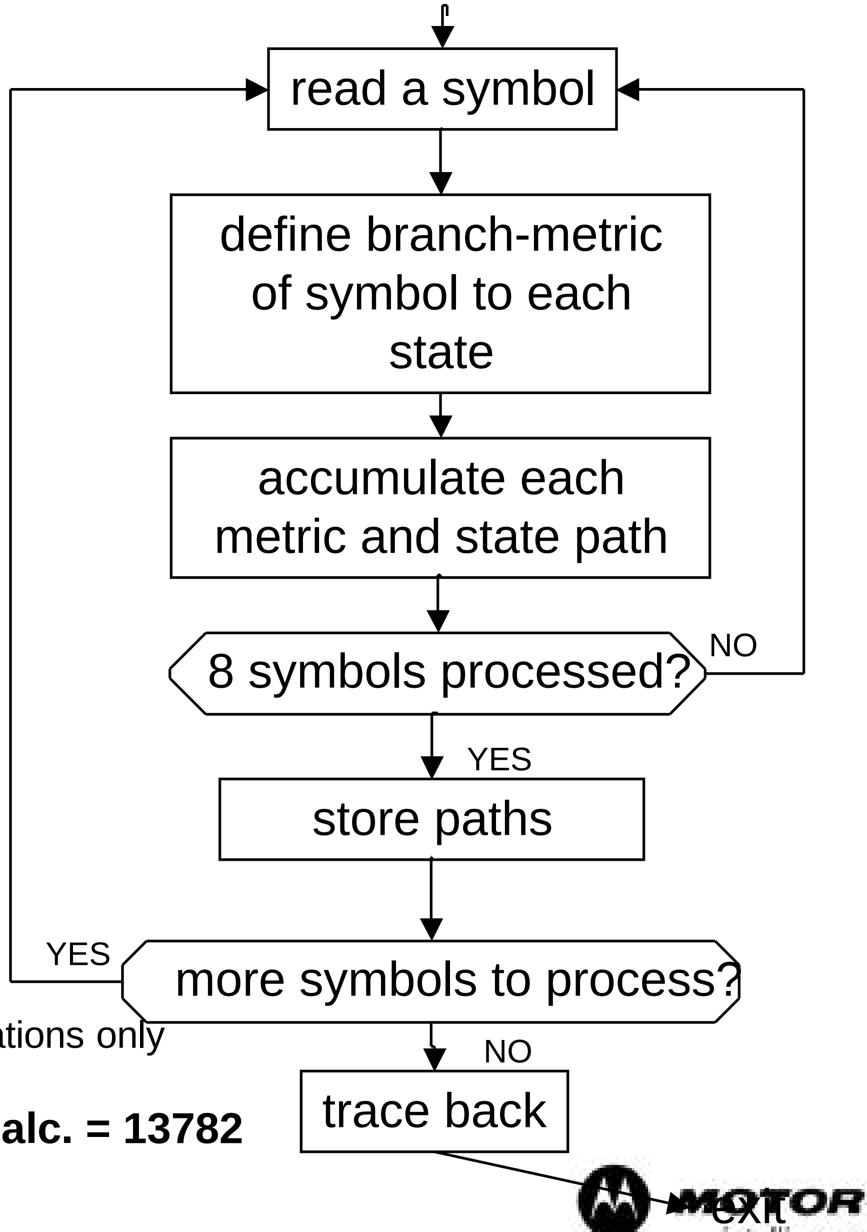
Viterbi(Trellis)
Butterfly

ACS
(Add
Compare
Select)

Path Store
(Trace Back)

Trellis
Path

EEMBC Viterbi Flow Chart*



read a symbol

define branch-metric of symbol to each state

accumulate each metric and state path

8 symbols processed?

NO

YES

store paths

more symbols to process?

YES

NO

trace back

exit

#of calculations**	% of calc
4	9.98%
32	79.8%
32	9.98%
#of symbols/16	0.15%

* PreACS not shown

** counting major calculations only

Approx. Total # of Calc. = 13782 (for 344 symbols)

IS136 Viterbi Decoder First 5 Stages#

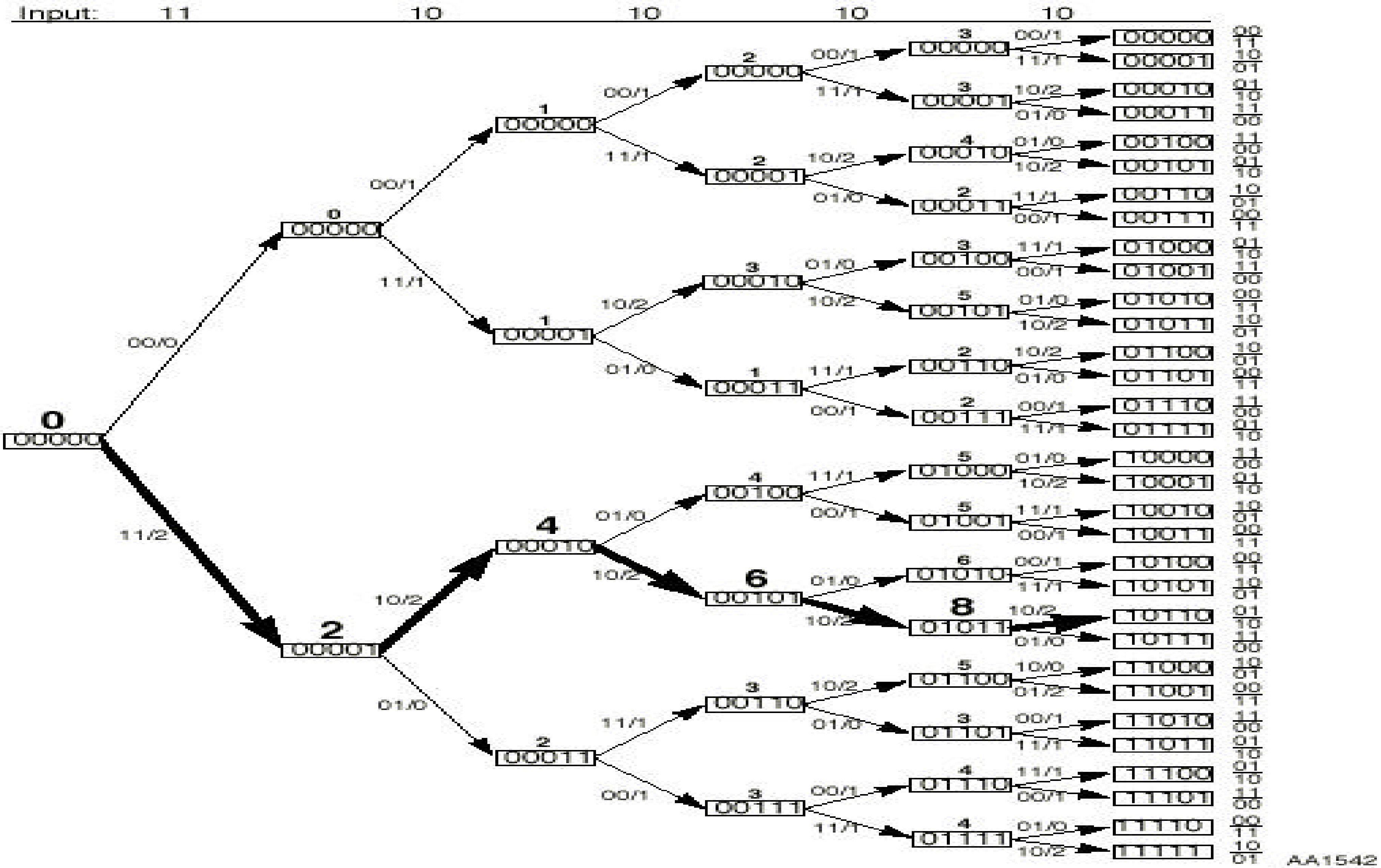
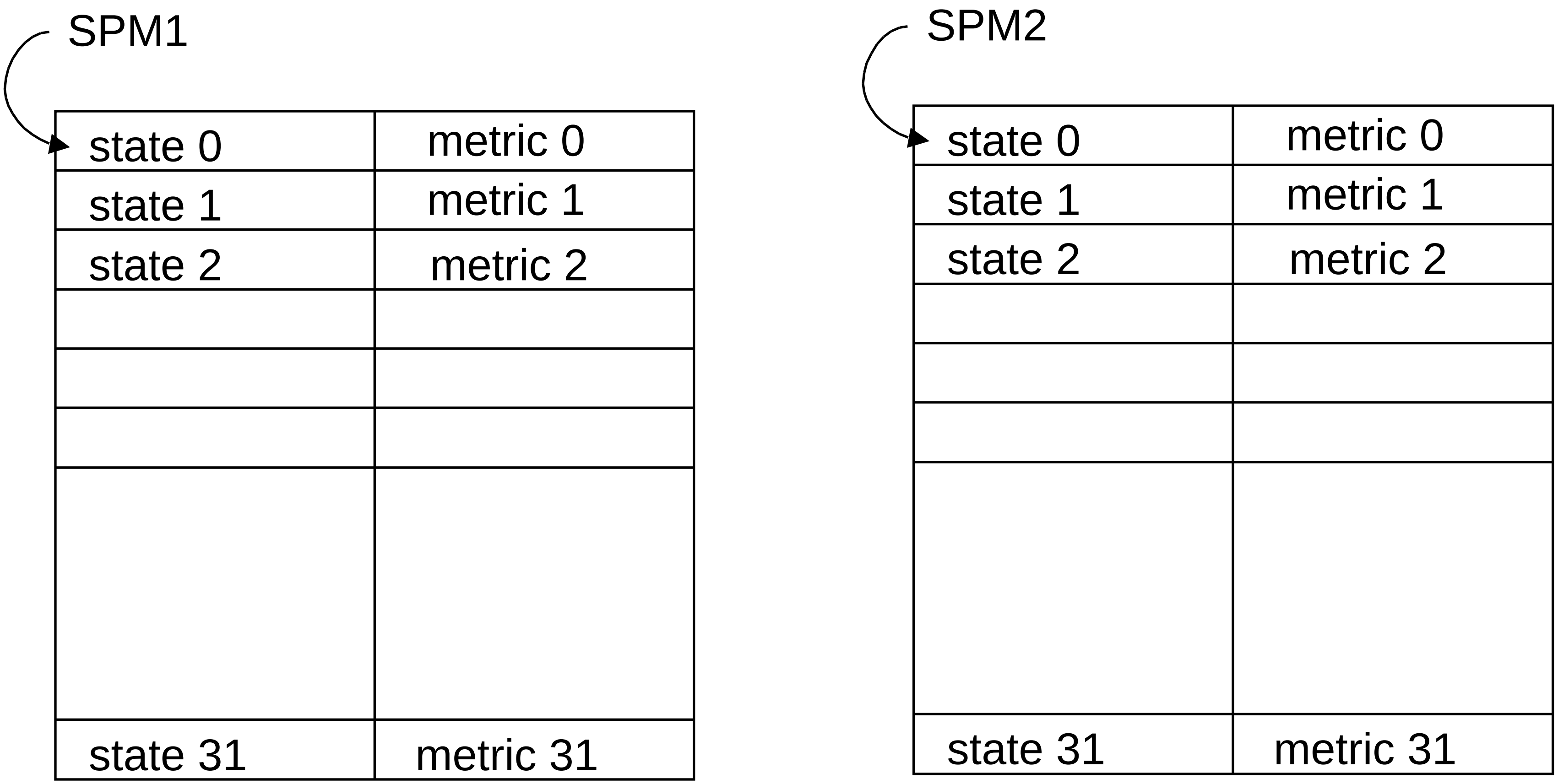


Figure 2-3 First Five Levels of the Encoder State Tree

EEMBC Basic Data Structure#



SPM1

state 0	metric 0
state 1	metric 1
state 2	metric 2
state 31	metric 31

SPM2

state 0	metric 0
state 1	metric 1
state 2	metric 2
state 31	metric 31

On each symbol processing interval metric and previous state values are read from one buffer to the other. On the next interval the buffers are switched.

EEBMC Basic Data Structures#

pBranchMetrics

cost00	cost10	cost01	cost11	cost11	cost01	cost10	cost00
cost01	cost11	cost00	cost10	cost10	cost00	cost11	cost01

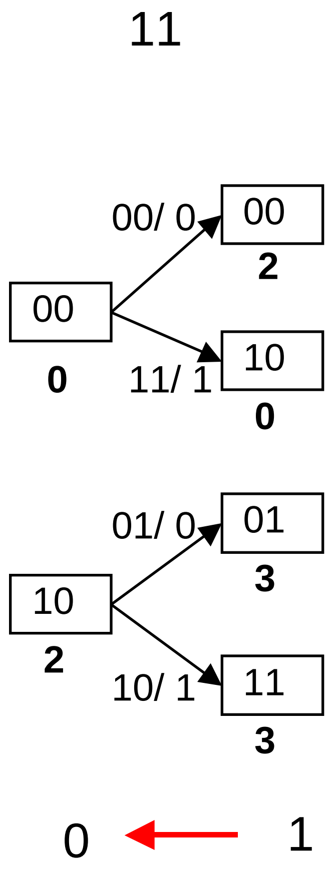
This table is used to hold the cost (metric) of the current symbol to 00, 01, 10, and 11. The order of the above table defines the Generator Polynomial Functions. This order indicates the (g0g1) outputs for all states given the input is 1. For an input of 0, the (g0g1) values can be obtained by negating the above table.

Vectorization

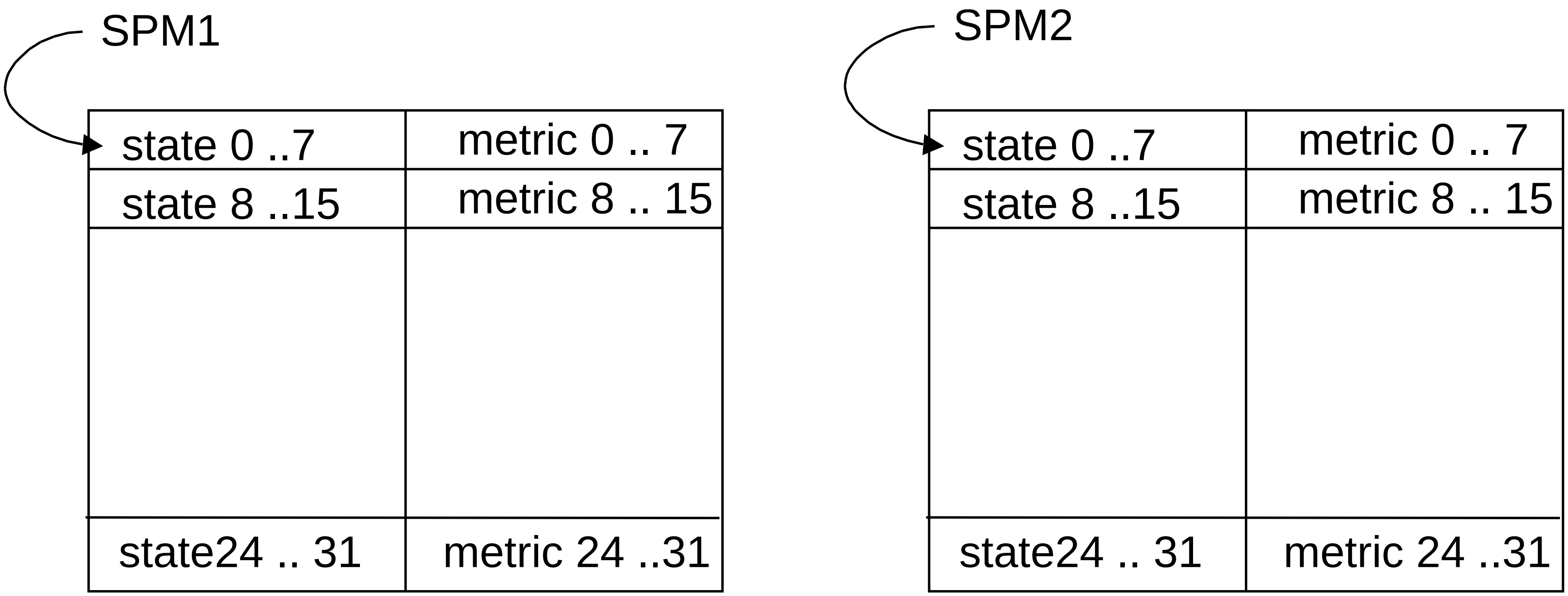
The major parts of the Algorithm are:

1) Defining the Branch Metrics
2) Calculating the branch cost for each symbol
3) Accumulating the branch cost for each state in each interval
4) Tracing back

3 and 4 cannot be totally vectorized because of dependencies
1 and 2 can be vectorized



Vectorized Code Data Structure#



SPM1

state 0 ..7	metric 0 .. 7
state 8 ..15	metric 8 .. 15
state24 .. 31	metric 24 ..31

SPM2

state 0 ..7	metric 0 .. 7
state 8 ..15	metric 8 .. 15
state24 .. 31	metric 24 ..31

On each symbol processing interval metric and previous state values are read from one buffer to the other. On the next interval the buffers are switched.

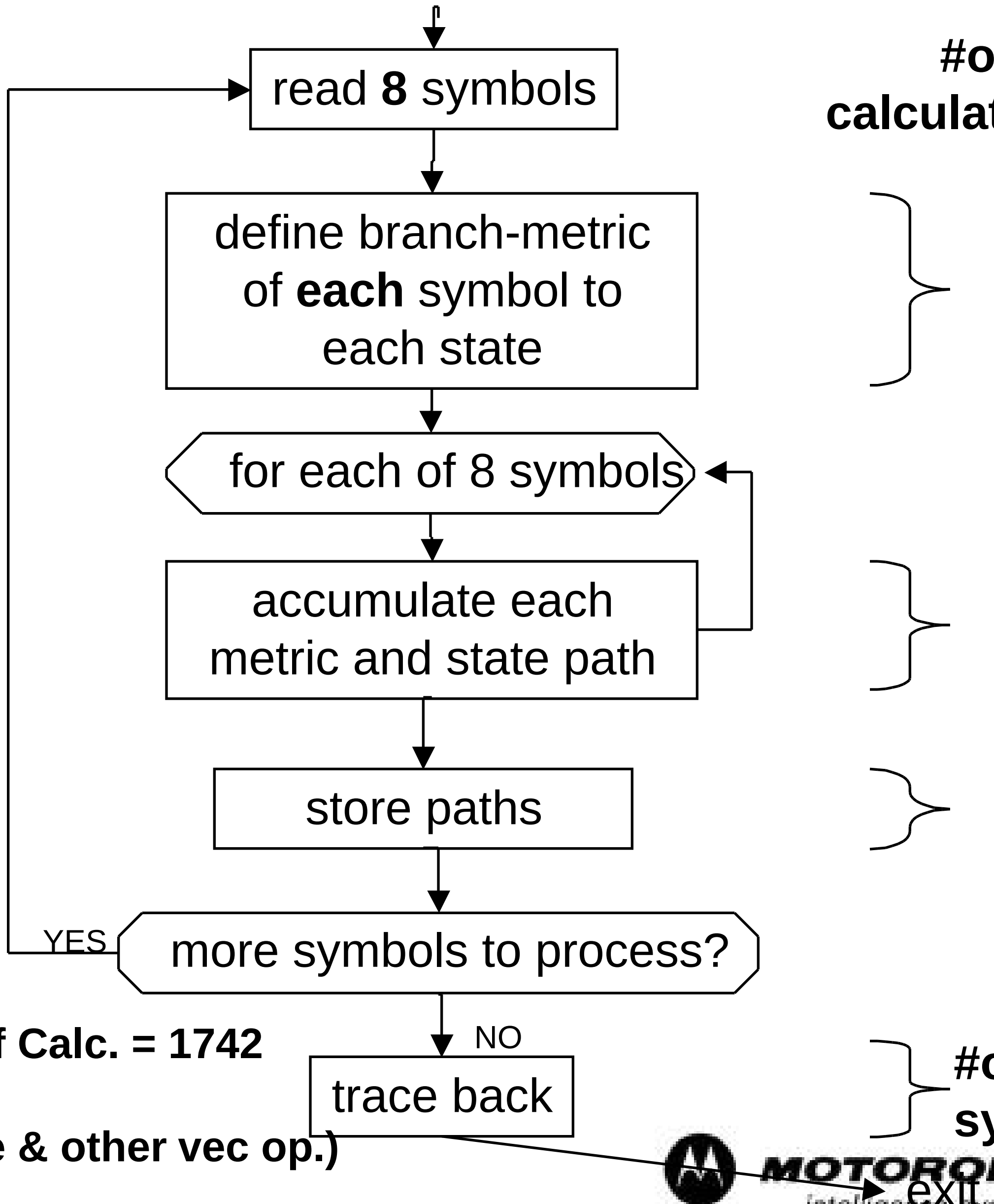
Vectorized Code Data Structure#

pBranchMetrics

c 00_0	c10_0	c01_0	c11_0	c11_0	c 01_0	c10_0	c 00_0
c 01_0	c11_0	c 00_0	c10_0	c10_0	c 00_0	c11_0	c 01_0
c 00_1	c10_1	c 01_1	c11_1	c11_1	c 01_1	c10_1	c 00_1
c 01_1	c11_1	c 00_1	c10_1	c10_1	c 00_1	c11_1	c 01_1
c 00_2	c10_2	c 01_2	c11_2	c11_2	c 01_2	c10_2	c 00_2
c 01_2	c11_2	c 00_2	c10_2	c10_2	c 00_2	c11_2	c 01_2
...	...	...	...	...	...	...	...

where cXY_Z stands for cost through state path XY for symbol Z.
For example: c 01_2 stands for the cost of transitioning through path 01 for symbol 2 in the group.

Vectorized IS136 Viterbi Flow Chart*



read **8** symbols

define branch-metric of **each** symbol to each state

for each of 8 symbols

accumulate each metric and state path

store paths

more symbols to process?

YES

NO

trace back

exit

	#of calculations**	**% of calc**
define branch-metric of each symbol to each state	**4**	**9.87%**
accumulate each metric and state path	**4**	**78.9%**
store paths	**4**	**9.87%**
trace back	**#of symbols/16**	**1.26%**

* PreACS not shown
** counting major calculations only

Approx. Total # of Calc. = 1742 (for 344 symbols, excludes permute & other vec op.)

Summary and Speedups

Viterbi Algorithm has dependency among the symbol intervals. Hence parallel inter-symbol calculations are not successfully achieved. However, for each symbol there are 2^n metric calculations (where n+1 is the constraint length). These calculations are independent among each other. Hence these calculations can be done in parallel.

Since shorts (which are eight per vector) are used for the data structures the speedup is ideally around that number. When overhead for vector loading and permuting is added, however, the speedup is limited to about 5x.

Data Input Type	Scalar Code	Vector Code	Speedup
all zeros	2579.5	13452.9	5.215
message	2548.9	13392.9	5.254
1-0 toggling	2531.6	13392.9	5.290
all ones	2535.9	13392.9	5.281

References

GSM Soft Decision Viterbi Decoder
http://www.mot.com/SPS/PowerPC/AltiVec

Viterbi Decoding Techniques in the TMS320C54x Family
http://www.edtn.com/scribe/reference/appnotes/md003f19.htm

Implementing Viterbi Decoders Using the VSL Instruction on DSP Families DSP56300 and DSP56600
http://www.mot.com/pub/SPS/DSP/LIBRARY/APPNOTES/APR40.PDF

Alantro’s Convolutional Encoding/Viterbi Decoding Page
http://www.alantro.com/viterbi/background.htm

AltiVec UMAC - Universal Message Authentication Code

$$Y = Y +_{64} (M[i] + K[i]) *_{64} (M[i+4] + K[i+4])$$

- 32x32 ==> 64 bit multiply
- AltiVec does not directly provide any of these operations:
  - 32x32 --> 64
  - 32x32 --> upper32
  - 32x32 --> lower32
- AltiVec ***does*** provide several 16x16 --> 32 multiplies, so it may be possible to achieve some speedup by breaking the 32x32 multiply into several 16x16 multiplies; See the MMH results.

UMAC also allows shorter word operations (16x16 --> 32)

- $\mathbf{Y = Y +_{32} (M[i] +_{16} K[i]) *_{32} (M[i+4] +_{16} K[i+4])}$
- Existing AltiVec implementation uses this approach, and achieves a 14.5x speedup (0.67 cycles per byte vs. around 10 cycles/byte) on very large messages, 10x on 1024-byte messages, 5.5x on 256-byte messages

AltiVec MMH - Multilinear Modular Hashing

- **Simple Multiply Accumulate inner loop:**

```
uint64 acc;
uint32 a[N], b[N];
for (I = 0; I < N; I++)
   a += b[I] * c[I];
```

- **Inner loop handled by following classic code sequence:**

```
mulhwu
mullw
addc
adde
```

Carry bit dependency on addc/adde

- **Estimated 10-15% speedup by emulated 64bit multiply via partial products in AltiVec**

AltiVec Reed Solomon - Cauchy

- **Main loop in both encode and decode:**

```
uint32 a[N], b[N];
for (I = 0; I < N; I++) // N = 25
   a[N] ^= b[N];
```

- **Theoretical speedup of 4x, however overhead associated with handling of alignment will reduce this some.**

AltiVec Reed Solomon - Conventional and Deflate

- **Reed Solomon - Conventional:**
  - Unlikely that AltiVec will be able to provide any speedup. Existing code uses table lookups to handle main loop

- **Deflate:**
  - Given zlib implementation of deflate algorithm code does not appear to be vectorizable