Johanna Tuominen | Juha Plosila

Asynchronous Viterbi Decoder in Action Systems

Turku Centre *for* Computer Science

TUCS Technical Report
No 710, September 2005

Asynchronous Viterbi Decoder in Action Systems

Johanna Tuominen
Turku Center for Computer Science
Lemminkäisenkatu 14 A, 20520 Turku, Finland
joeltu@utu.fi
Juha Plosila
University of Turku, Dept. of Information Technology
Lemminkäisenkatu 14 A, 20520 Turku, Finland
juplos@utu.fi

TUCS Technical Report
No 710, September 2005

Abstract

Conventionally, the correctness of functional and non-functional properties of hardware components is ensured during design process by simulation. Moreover, different description languages are needed during development phases. Thus, by adopting the Action Systems, we are able to use the same formalism from specification down to implementation. Recently, we have been exploiting possibilities to formally model power consumption. That is the purpose is to develop formal power estimation flow, which can be used to monitor the power consumption from abstract level down to the gate level implementation. In this paper, we present a formal model for asynchronous Viterbi decoder, which will be used as a case study for the power estimation flow in the future.

Keywords: Viterbi, formalism, asynchronous

TUCS Laboratory
Communication Systems

1 Introduction

Formal methods provides an environment to design, analyze, and verify digital hardware with the benefits of rigorous mathematical basis. In this study, the Action Systems formalism is applied [1]. It is a framework for specification and correctness preserving development of concurrent systems, and it is based on an extended version of Dijkstra's language of guarded commands [3]. Development of the action system is done in a stepwise manner within the $refinement\ calculus$ [2]. The specification of a hardware system is transformed into an implementation using correctness preserving transformations. In conventional Action Systems, only the logical correctness of the system is verified, while non-functional properties, like time, power, and area are not validated.

Convolutional encoding and Viterbi decoding are widely used in modern communication systems, such as digital satellite TV, and digital mobile radios [6]. To satisfy the demands caused by the developments of the modern telecommunication, high-speed, low-power, and low-cost Viterbi decoders are required. In this paper, we present a formal model of an asynchronous Viterbi decoder. The asynchronous approach is chosen for the implementation because of its potential for low-power, and low-noise behavior [12].

Currently, we are exploiting the possibilities to formally model power consumption [10] [11]. The purpose is to develop a formal power estimation flow from initial specification down to implementation. To estimate the power consumption, there is a trade-off between the accuracy and the abstraction level of detail which the system is analyzed. The more detailed the description, the more accurate the simulation will be. But on the other hand, the more time consuming it will be. Moreover, the designer wants to make decisions as early as possible in the design flow to avoid design backtracking. Thus, the purpose is to use the asynchronous Viterbi decoder as a case study for the power estimation flow. That is, to estimate the power consumption of the decoder at different development phases. For instance, starting from the formal description presented here, and finally from the gate-level description.

2 Action Systems

An action A is defined by (for example):

$$
\begin{array}{ll}
A ::= abort & (abortion, non-termination) \\
\mid skip & (empty\ statement) \\
\mid A_1 [\!] \ldots [\!] A_n & (non-deterministic\ choice) \\
\mid A_1; \ldots; A_n & (sequential\ composition) \\
\mid x := e & ((multiple)\ assignement) \\
\mid g \rightarrow A & (guarded\ command)
\end{array}
$$

where A_i, $i = 0, ..., n$, are actions; x is a variable or a list of variables; x_0 is a value(s) of the variable(s); e is an expression or a list of expressions; g is a predicate.

Semantics of actions. Action is considered to be $atomic$, which means that only the initial and final states are observed by the system. Thus, when selected for execution, the action is completed without any interference from other actions. Atomic actions may be represented by simple assignments or by more complex action compositions, such as the atomic sequence. $Non - atomicity$ means that an action outside the composition can execute between two component actions of the construct, which is not possible in the $atomic$ composition structures. The notation differs whether the composition is atomic or not, for instance, the sequential composition is noted by ; ($atomic$), and **;** ($non - atomic$).

The actions are defined using weakest precondition for predicate transformers [3]. For instance, the correctness of an action A with respect to predicates P and Q (precondition and postcondition) is denoted by: $\{P\}A\{Q\} = P \Rightarrow wp(A, Q)$. The $wp(A, Q)$ is the weakest precondition for the action A to establish the postcondition Q. The $guard\ gA$ of an action A is defined by $gA = \neg wp(A, false)$. An action is enabled when its guard evaluates to $true$, otherwise disabled.

2.0.1 Action System

An action system $\mathcal{A}$ has a form:

sys $\mathcal{A}\ (g)\ [par]$
|[
type t
const c
var v
actions A
subsys S_A
init "$initialization\ of\ the\ variables\ g\ and\ v$"
exec
do "$composition\ of\ actions\ A$" **od**
]|

Three different parts can be identified from the action system description: $interface$, $declarations$, and $iteration$.

The interface part specifies global variables g, that is, variables that are visible outside the action system. In other words, global variables are accessible by other action systems. If an action system does not have any interface variables, it is a $closed$ action system otherwise it is an $open$ action system. The declaration part consists of type (t), variable (v), constant (c), and action (A) declarations. Furthermore, type definitions and initializations are described in the declaration part. Using the items introduced in the interface and declarative parts the operation of the system is described in the iteration section; in the **do** $-$ **od** loop.

The operation of an action system is started by initialization in which the variables are set to predefined values. Actions are selected for execution based on the composition operators and the enabledness of the actions. The operation is continued until there are no actions to enable, which temporarily aborts the system. Thus, the operation continues if some action enables it.

Quantified constructs Any action-level operator $\bullet \in [\!], ; (atomic), \mathbf{;} (non-atomic)$, and the system-level operator $\parallel$ can be quantified using the notation defined as follows:

$$[\bullet\ 1\ \leq\ i \leq\ n\ :\ A(i)] \widehat{=} A(1)\ \bullet\ ...\ \bullet\ A(n)$$
$$[\parallel\ 1\ \leq\ i \leq\ n\ :\ \mathcal{A}(i)] \widehat{=} \mathcal{A}(1) \parallel ... \parallel \mathcal{A}(n)$$

Composing Action Systems Consider two hierarchical action systems $\mathcal{A}$ and $\mathcal{B}$ with distinct local variables, local procedures, subsystem instances, and actions. The parallel composition of such systems is denoted by $\mathcal{A} \parallel \mathcal{B}$. It is defined to be another action system whose global and local identifiers (procedures, variables, subsystem instances, actions) consist of the identifiers of the component systems and whose **exec**-clause has the form: $do\ A \parallel\ B\ od \parallel S_A \parallel S_B$. Here A and B denote the action compositions, and S_A and S_B the subsystem compositions in $\mathcal{A}$ and $\mathcal{B}$, respectively. The definition of the parallel composition is used inversely in system derivation to decompose a system description into a composition of smaller separate systems or internal subsystems.

3 Viterbi Algorithm

3.1 Convolutional Coding

The convolutional codes differ from the block codes by the existence of memory in the encoding scheme. That is, the convolutional encoder achieves low error-rate transmission by adding redundancy to the input stream symbols. Thus, it produces more output bits than input bits shifted to its memory. The convolutional code is generated by passing the information sequence to be transmitted through a linear finite-state shift register [6].

The code rate of the convolutional encoder is defined as the number of input bits to output bits. In general, the code rate is defined as $R_c = k/n$, and the encoder is represented in (K, k, n) format, where

n : number of code symbols produced by the encoder
k : number of information bits coming into the encoder
K : constraint length of the code

The constraint length of the encoder is measure of the memory within the code. The structure of the (K, k, n) convolutional encoder is shown in Figure 1.

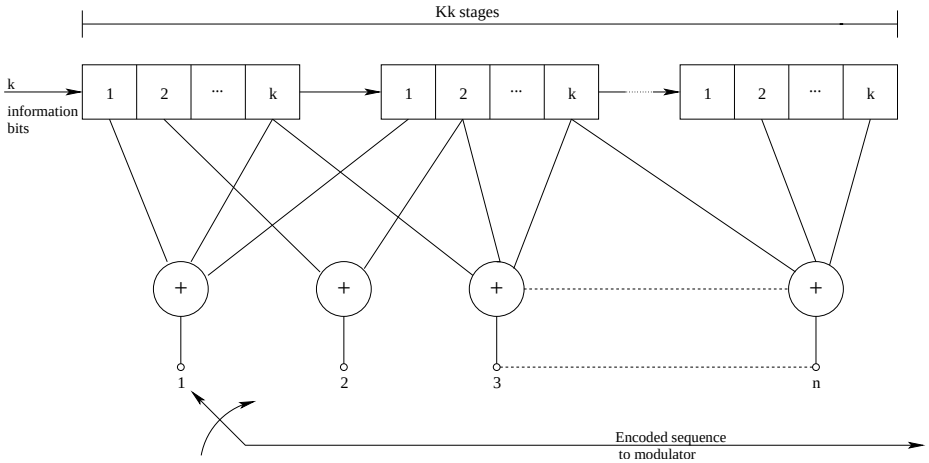

Figure 1: Convolutional encoder

The (K, k, n) convolutional code can be specified (for instance) by using a set of n vectors, one vector for each of the n modulo-2 adders [6]. Each vector has a Kk dimensions and contains the connections of the encoder to that modulo-2 adder. A 1 in the ith position of the vector indicates that the corresponding stage in the shift register is connected to the modulo-2 adder, and a 0 denotes that there is no connection between that stage and the modulo-2 adder.

To be specific, let us consider the binary convolutional encoder with constraint length $K = 3$, $k = 1$,and $n = 2$, which is shown in Figure 2.

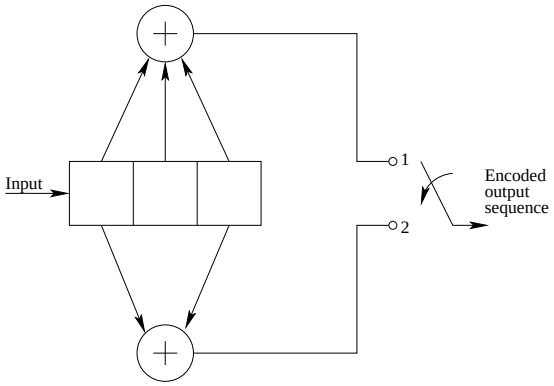

Figure 2: K=3, k=1, n=2 convolutional encoder

Initially, the shift register is assumed to be in the all-zero state. Assume that the first input bit is $'1'$. Then the output sequence of two bits is $''11''$. Then suppose that the second input bit is $'0'$, then the output sequence is $''10''$. If the third input bit is $'1'$, then the output bits are $''00''$, and so on. Next, we number the outputs of the function generators that generate each two bit output sequences 1,2 from top to bottom, and similarly number each function generator. The first function generator is connected to stages 1, 2, and 3, and therefore the generator is

$$\mathbf{g}_1 = [1, 1, 1]$$

The second function generator is connected to stages 1 and 2. Hence

$$\mathbf{g}_2 = [1, 0, 1]$$

The generators for this code are given in octal form as $(7, 5)$. In conclusion, we have that, when $k = 1$, we require n generators, each of dimension K to specify the encoder [6].

The convolutional encoder is a finite state machine, where any state is a content of its memory. The outputs of the encoder are dependent on the recent input bits and on its previous memory contents. Therefore, the convolutional encoder can be completely described by the state-transition diagram. The state transition diagram for the $(3, 1, 2)$ encoder is shown in Figure 3 [8].

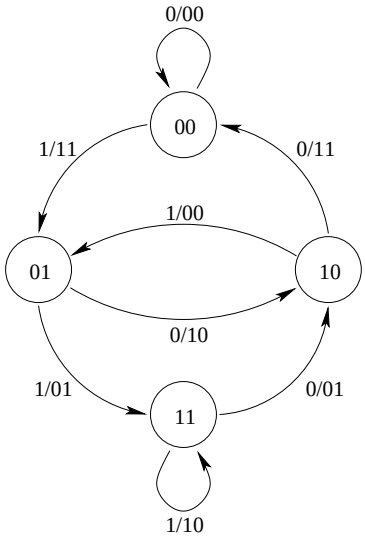

Figure 3: State diagram for rate 1/2, K=3 convolutional code.

In the Figure 3, the states are depicted as nodes, and the transitions from one state to another are indicated as arrows. Thus, depending on whether the current binary input is 1 or 0, there are two possible states where the encoder might enter.

To describe the transitions between states as a function of time, the convolutional codes can be described as a $trellis$ diagram [6]. The trellis diagram for the convolutional encoder, shown in Figure 2 is illustrated in Figure 4.

The nodes of the trellis diagram represent the memory contents of the encoder. The solid line denotes the output generated by the input bit $'0'$, and the dotted line the output generated by the input bit $'1'$. In the encoding procedure, it is often assumed that the encoder at $t = 0$ is set to initial state $(S0)$. Then at time $t = 1$, we have two possible states $S0$, and $S1$, depending on whether the input bit is $'0'$

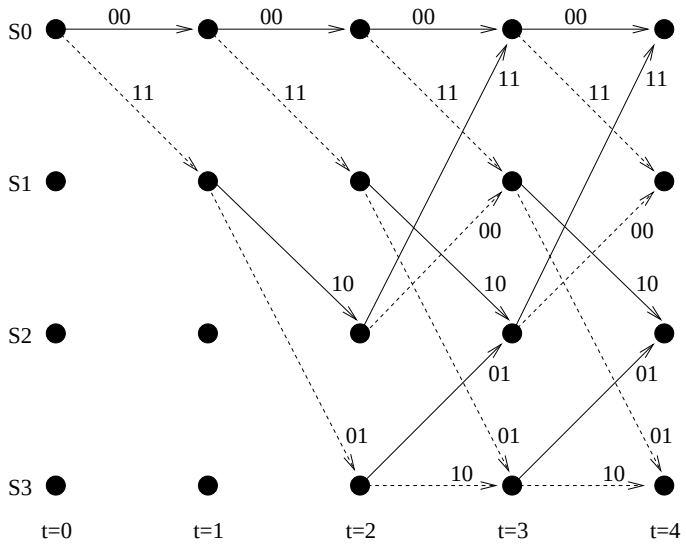

Figure 4: Trellis diagram for rate 1/2, K=3 convolutional code.

or $'1'$, respectively. After, the second stage $(t = 2)$, each node in the trellis has two incoming paths, and two outgoing paths.

3.2 The Algorithm

The Viterbi algorithm finds the most-likely state transition sequence in a state diagram, given a sequence of symbols. That is, given a sequence of input symbols and an initial state, one can derive a sequence of output symbols based on the transitions and their input/output relations in a given finite state diagram. The finite state diagram is often presented as $trellis$, shown in Figure 4. In other words Viterbi algorithm finds the sequence of symbols in the given trellis that is closest in distance to the received sequence of noisy symbols. This sequence computed is the global $most\ likely\ sequence$. When the $Euclidean$ distance is applied as a distance measure, the Viterbi algorithm is the optimal maximum-likelihood detection method, when the sequence of symbols is corrupted by the additive white Gaussian noise (AWGN) [4]. In practice, the $Hamming$ distance is often applied event though the performance of the Viterbi Algorithm is sub-optimal [5], for instance, in terms of noise optimation. However, regardless of the distance measure, the procedure to search for the most-likely sequence is the same. We will use, both the Euclidean and the Hamming distance, to illustrate the Viterbi Algorithm.

Consider the following communication system, shown in Figure 5.

The convolutional encoder adds redundancy to the input signal s, and the encoded output x symbols are transmitted over a noisy channel. The input of the convolutional decoder, that is the input for the Viterbi decoder r is the encoded symbols contaminated by noise. Then the decoder tries to extract the original

Figure 5: Encoding / decoding convolutional code.

information from the received sequence and generates an estimate y. The algorithm that maximizes the conditional probability $P(r|y)$ is called the maximum likelihood algorithm [4].

The maximum-likelihood algorithm finds the most likely code sequence for the received channel output sequence. Therefore, if the encoder output sequence is denoted by x_m, and the channel output sequence is denoted by r, the probability of receiving r when the channel input is x_m is

$$Pr(r|x_m) = \prod_{n'=0}^{\infty} Pr(r_{n'}|x_{mn'}) \tag{1}$$

The most likely path through the trellis for the channel output r is the one that maximizes the function 1 [4, 5]. Thus, the function, shown in Equation 1, is usually called the $metric$, and it is used in comparison between the code sequence and the received sequence. Notice that, the decoding metric in 1 requires product implementation, and therefore the metric $ln[Pr(r|x_m)]$ is more frequently applied than the metric $Pr(r|x_m)$ in the decoder. Moreover, finding the trellis path with the largest log-likelihood function corresponds the maximum likelihood decoding:

$$ln[Pr(r|x_m)] = ln[\sum_{n'=1}^{\infty} Pr(r_{n'}|x_{mn'})] \tag{2}$$

where the components of the summation are accumulated on the individual branches, and therefore they are denoted by $branch\ metrics$.

3.2.1 Hard-decision decoding

In the hard-decision decoding, the path through the trellis is determined using the Hamming distance measure. Thus, the most optimal path through the trellis is the path with the minimum Hamming distance. In other words, the Hamming distance can be defined as a number of bits that are different between the observed symbol at the decoder and the sent symbol from the encoder. Furthermore, the hard-decision decoding applies one bit quantization on the received bits. The channel noise is assumed to be white, and therefore the channel is assigned to be Binary Memoryless Channel (BMC) in the hard decision decoding [14].

The decoding is assumed to start from the initial state, and return back into the all zero state. Assuming that the optimal path covers total of m branches, and each of the branches is expected to represent n bits of the encoder output. Then

the overall number of bits in the trellis code word, and the quantized received sequence is nm. Thus, the Hamming distance between the trellis code word $\overrightarrow{c}$ and the quantized received sequence $\overrightarrow{y}$ each of them being of length n is [8]:

$$d(\overrightarrow{c}, \overrightarrow{y}) = \sum_{i=1}^{n} d(\overrightarrow{c}_i, \overrightarrow{y}_i) \quad 1 \leq i \leq n \tag{3}$$

3.2.2 Soft-decision decoding

Soft-decision decoding is applied for the maximum likelihood decoding, when the data is transmitted over the Gaussian channel. On the contrary to the hard-decision decoding, the soft-decision decoding uses multi-bit quantization for the received bits, and Euclidean distance as a distance measure instead of the Hamming distance.

In the soft-decision decoding, the vector output of the channel $\overrightarrow{r}$is applied instead of the quantized received sequence $\overrightarrow{y}$. Furthermore, the binary sequence $\overrightarrow{c}$ is frequently replaced by the antipodal sequence $\overrightarrow{c}'$ with:

$$c'_{ij} = \begin{cases} \sqrt{\epsilon} \\ \\ -\sqrt{\epsilon} \end{cases} \quad \text{for} \quad 1 \leq i \leq m \quad \text{and} \quad 1 \leq j \leq n \tag{4}$$

Thus, the Euclidean distance can be represented in a form:

$$d^2(\overrightarrow{c}', \overrightarrow{r}) = \sum_{i=1}^{m} d_E^2(\overrightarrow{c}'_i, \overrightarrow{r}_i) \tag{5}$$

3.2.3 Functionality of the algorithm

The maximum likelihood path is characterized with the aid of the $branch\ metric$ and a $path\ metric$. The algorithm uses a set of $path\ metrics$ to describe the various costs of different path through the trellis. Consider the example, shown in Figure 6.

At the beginning of the decoding process $(t = 0)$, the path metric of the $S0$ is initialized to zero. After the first time step, one of the two branches starting from the state $S0$ arrive at the state $S0$ with a branch metric. Thus, this branch metric is, for instance, the Hamming distance between the expected input 00 and the received input 01. Similarly, we can define the branch metric for the branch entering to the state $S2$. This procedure is repeated at time $t = 2$, and so on. The $path\ metric$ value is the sum of the previous path metric and the current branch metric. Notice that, at time $t = 1$ and $t = 2$ there are only one path leading to each state, and therefore the incoming branches are the $survivor$ branches. At time $t = 3$, each node in the trellis has two incoming paths and two out going paths. Thus, the trellis is reached its steady state [6]. For example, at the state $S0$,

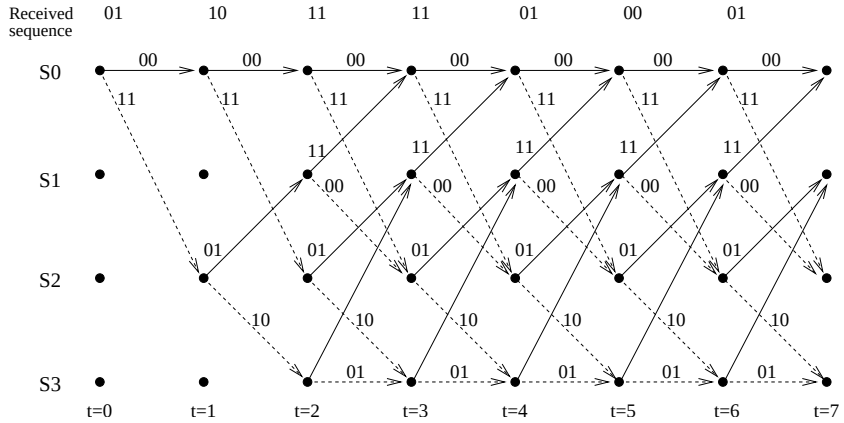

Figure 6: The trellis diagram for the Viterbi decoding example.

the arrival at this state is possible from states $S0$ and $S1$. The path entering from the state $S0$ has the path metric value of 2, while the other path is entered with the accumulated metric 5. Thus, the path through the state $S1$ is discarded, and the path through $S0$ survives. Notice that, if one or more paths have the smallest path metric, the selection of the most likely path will be chosen randomly.

Once the survivor path for the whole trellis is identified, the decoded output sequence is extracted by performing a $trace\ back$ operation. The trace back process starts from the state $S0$ at $t = 7$ and is executed by going backward in time. The survivor path traces back into the state $S0$ at time $t = 6$. By continuing the same procedure, the optimal path through trellis is determined.

4 Viterbi Decoding Implementation

The simplified structure of a Viterbi decoder is shown in Figure 7, which consists of three units:

1. Branch Metric Unit (BMU) generates branch metrics, which measure the difference between the received symbol, and the symbol that causes the transitions in the trellis.

2. Path Metric Unit (PMU) consists of two parts: The Add-Compare-Select Unit (ACSU) and the State Metric Memory (SMM). To find the survivor path for each state, the branch metric of a given transition is added to the recent path metric value stored in the state metric memory. This new path metric is then compared with other path metrics, which are entering to that state. The transition with the minimum path metric is chosen to be the survivor metric. The path metric of the survivor path of each state is updated and stored back into the state metric memory.

3. Trace Back Unit (TBU) stores the survivor paths and performs the trace back operation. Outputs the decoded sequence.

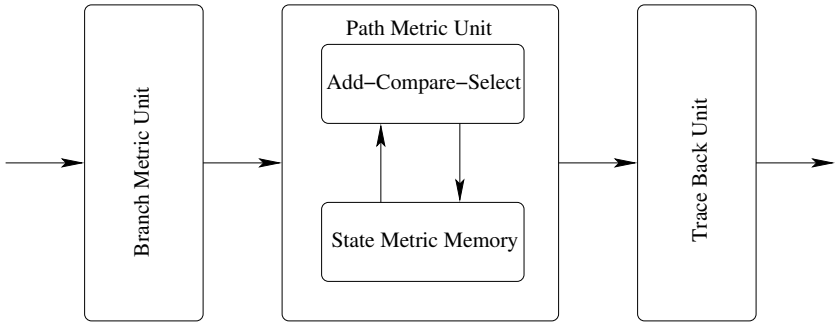

Figure 7: A simplified block diagram of the Viterbi decoder

The implementation of the digital decoder requires quantization, that is, the received analog signal is converted into digital format. In this design, we apply the multi-bit quantization, and therefore the $Euclidean$ distance (soft decision decoding). as a distance measure for the branch metrics generation. Thus, the branch metric is generated by calculating the squared distance between each noisy symbol and the expected sequence of symbols in that path. If the received sequence of noisy symbols is denoted by $\overrightarrow{y} = (y_1, y_2, y_3, ..., y_n)$, the branch metric for the transition from state i to state j is:

$$B_{i,j,n} = (y_n - C_{i,j})^2 \tag{6}$$

where the $C_{i,j}$ is the expected code symbol of the transition from state i to state j.

Table 1: Next state table corresponding the butterfly structure in Fig. 8

00	10
00	10
01	11
01	11

To achieve higher computational efficiency, the trellis diagram can be reordered into butterfly structures, as shown in Figure 8. The butterfly structures are applicable for a rate of $1/n$ convolutional encoders. The reordered trellis is partitioned into groups of states and state transitions, which are isolated from other groups. The next state table of the butterfly in Figure 8, is shown in Table 1, and the output table is depicted in Table 2. Notice that the columns of the tables corresponds the input bit $'0'$ and $'1'$, respectively.

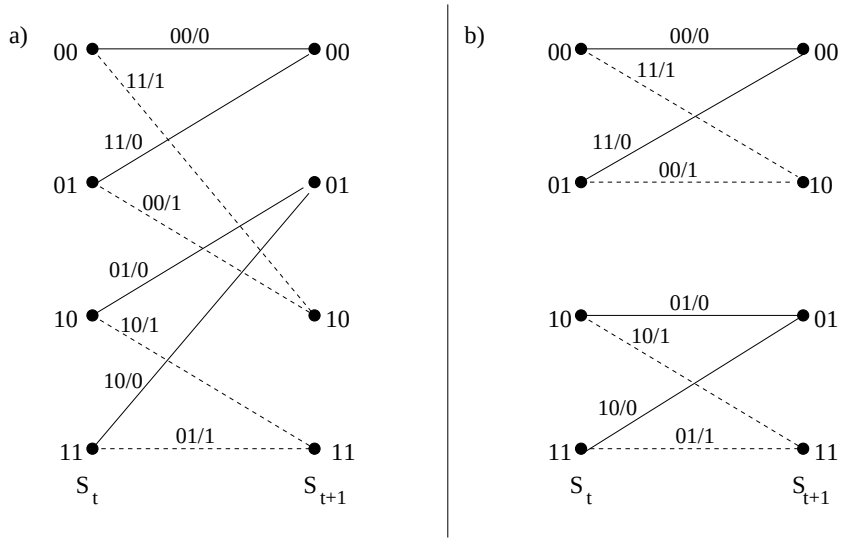

Figure 8: (a) One section of trellis diagram (b) Corresponding butterfly structure.

Table 2: Output table corresponding the butterfly structure in Fig. 8

00	11
11	00
01	10
10	01

The butterfly blocks are implemented by using Add-Compare-Select-Units (ACSU). The block diagram of the ACSU is shown in Figure 9.

The two adders calculate the sum between incoming branch metrics and the previous path metrics.

$$PM_1 = PM_1 + BM_1 \tag{7}$$

$$PM_0 = PM_0 + BM_0 \tag{8}$$

The comparison operation finds the most likely path that has the minimum metric:

$$D = (PM_0 < PM_1) \tag{9}$$

Furthermore, the selector determines the path with the minimum metric:

$$PM_{out} = min(PM_0, PM_1) \tag{10}$$

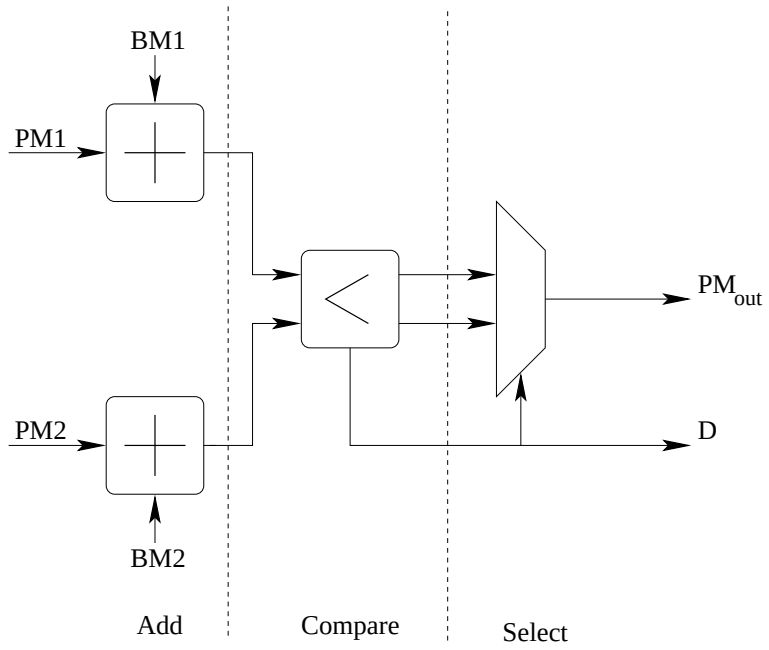

Figure 9: Block diagram of Add-Compare-Select Unit

By assuming that the total number of transitions in the trellis is M, and the number of states is N, a maximum of $(M - N)$ comparison operations are required, and $2N$ sums are required to initialize the metric for each state. The number of the memory accesses required to store the indexes for the survivor paths is dependent on the way the trace back unit (TBU) is implemented. Another real-time implementation issue for the trace back unit is the possibility for overflows in the registers that holds the path metrics. This is due to the finite length of the registers. A conventional approach to prevent the overflow in the registers is to use normalization. That is, subtracting a constant from all the path metrics to be compared from time to time. This approach requires an additional comparison, and subtraction operations to normalize the path metrics [5].

To avoid the extra computation required to normalize the path metrics is to design the registers to store the path metrics to be of length greater than $2D_{max}$, where the D_{max} is the maximum possible difference between the path metrics [9]. By adopting this methodology, we can imagine that due to the finite precision of the register, a path metric is running on a circle in the clockwise direction every time a positive number is added to it. For instance, consider the following example, shown in Figure 10, where the register length is 3 bits.

Thus, if the distance between the two path metrics to be compared on the circle is always less than half of the length of the circumference of the circle, then we know that the metric that is farther along in a clockwise direction has a larger value. Therefore, to determine the smaller or larger path metric, the comparison operation can be described as follows:

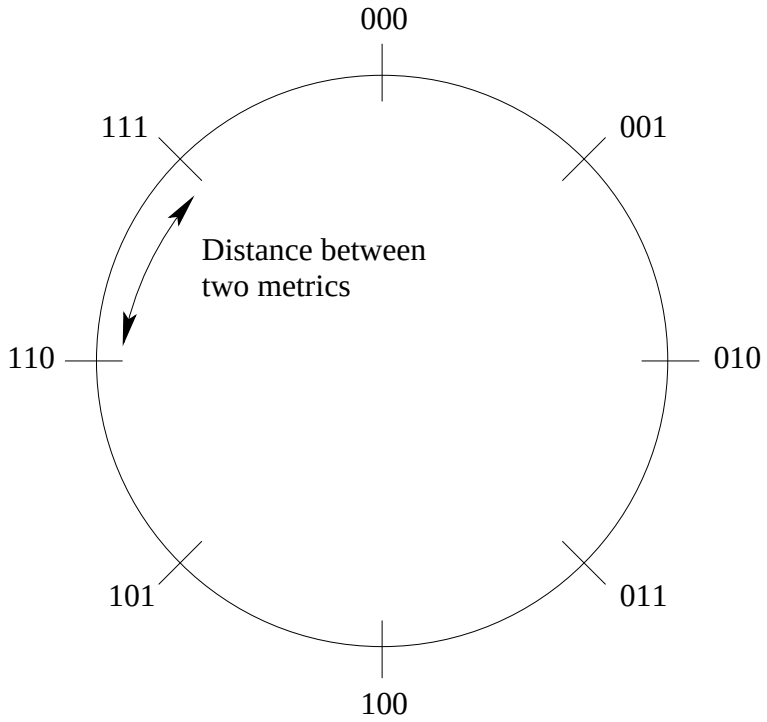

Figure 10: Finite precision effect of path metrics

Let $\overline{m_1} = (m_{1n}, ..., m_{10})$ and $\overline{m_2} = (m_{2n}, ..., m_{20})$ be the two metrics to be compared, and $\overline{d_n} = (d_n, ..., d_0) = (\overline{m_1} - \overline{m_2})$ using 2's complement arithmetic. If $Z(\overline{m_1}, \overline{m_2})$ denotes the logical result of comparing the path metrics $\overline{m_1}$ and $\overline{m_2}$, then

$$Z(\overline{m_1}, \overline{m_2}) = \begin{cases} 1, & \text{if } \overline{m_1} \leq \overline{m_2} \\ 0, & \text{otherwise} \end{cases}$$

and $Z(\overline{m_1}, \overline{m_2}) = d_n$

There are two commonly used techniques for the survivor memory design: The register exchange and the trace back techniques. We will adopt a technique that is based on the trace back technique, and therefore the name of the unit is trace back unit (TBU). An example of the memory structure used is shown in Figure 11, where the memory is divided into several blocks [5]. The number of blocks depends on how long one wants to keep the survivors. In other words, the number is dependent on the survivor path length, L. Each block in the memory corresponds to one stage of a trellis, and stores the survivor path pointer for each state. For instance, consider the example shown in Figure 11.

The survivor path memory,in this example, contains two locations in each block to store the survivor paths for state 1 and -1. Location 0 in each block stores the survivor path pointer for state 1, and the location 1 stores the pointer to

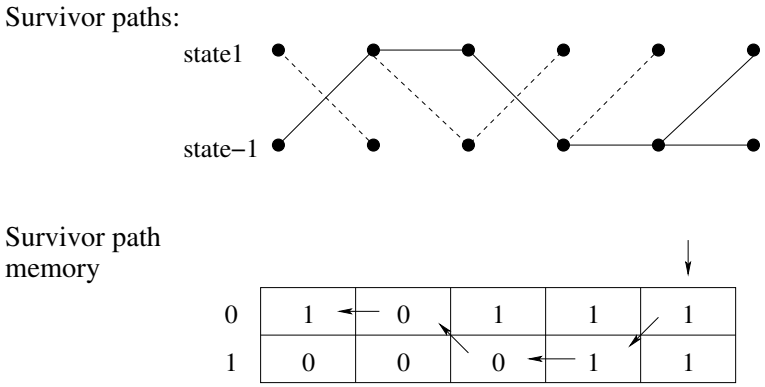

Figure 11: Example: tracing the survivor memory

the state -1. To trace the survivor path from state 1 to the most recently set block, we can set the pointer to read from location 0 of this block. Since, the content in location 0 has a value of 1, we decrement the block counter, and read the location 1 from the previous block.

After pointing to the last block of the survivor path memory, the block counter overflows, and resets to the location 0. Thus, the survivor memory can be illustrated as circular block of memory, shown in Figure 12.

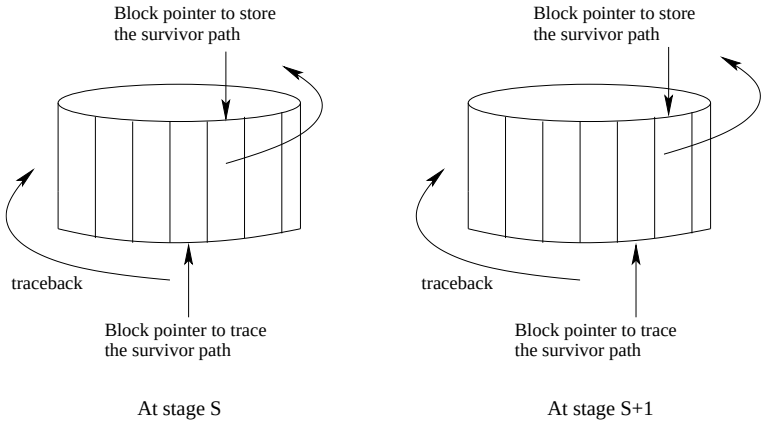

Figure 12: Organization of the survivor path memory

If the length of the survivor path is L, there should be at least $(L+1)$ blocks in the survivor path memory. Notice that, the survivor path updates can be implemented in a first-in first-out manner, and therefore the oldest block of data is overwritten continuously in the survivor path memory. Using this implementation method, the survivor path memory requires $(L+1)N$ locations and $log_2(N)$ bits for each location, where N is the total number of states in the given trellis [5]. Moreover, L memory indexing operations are required to trace the survivor path memory, and N memory accesses are required to update the survivor path pointers

for all states. For instance, a rate $1/n$ convolutional code the minimum value for L is $5(log_2(N))$.

5 Formal Specification

5.1 Convolutional encoder

In general, when a binary convolutional code with $k = 1$ (number of input bits), and a constraint length K is decoded by means of the Viterbi algorithm, there are 2^{K-1} states [6]. Therefore, there are 2^{K-1} surviving paths at each stage, and 2^{K-1} metrics, one for each surviving path. For the formal specification, we use (2,1,7) convolutional code, which is widely used, for example in satellite communication systems, and in NASA space programs. The Convolutional encoder is illustrated in the Figure 13.

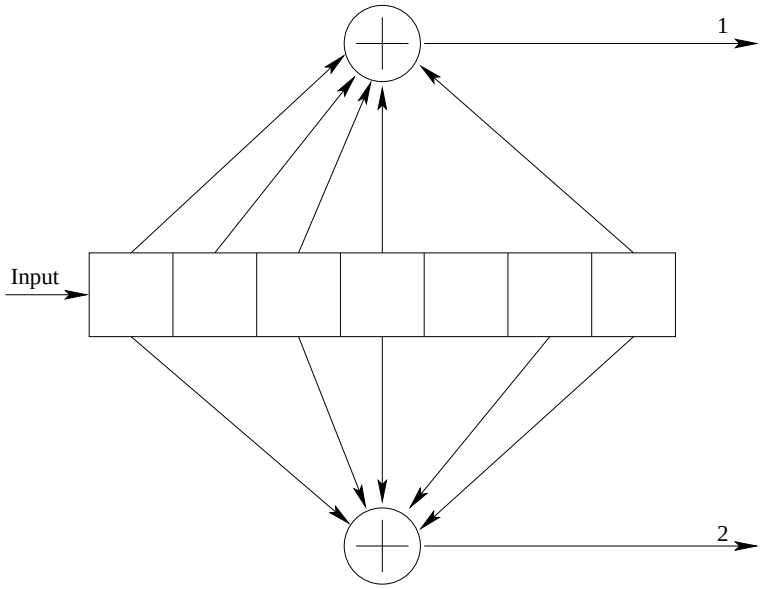

Figure 13: Encoder for rate R= 1/2, and constraint length K=7 convolutional code

Thus, we can define that the number of states in the trellis is 64. The function generators for the encoder is defined by:

$$\mathbf{g}_1 = [1, 1, 1, 1, 0, 0, 1]$$

$$\mathbf{g}_2 = [1, 0, 1, 1, 0, 1, 1]$$

In octal form the generators are $(171, 133)$. Equivalently, we can define the generator polynomials for each generator:

$$\mathbf{G}_1(D) = 1 \oplus D \oplus D^2 \oplus D^3 \oplus D^6$$

$$\mathbf{G}_2(D) = 1 \oplus D^2 \oplus D^3 \oplus D^5 \oplus D^6$$

where the D can be seen as a delay operator. For rate $R = 1/2$ convolutional code, the branch metric calculation between the received code word $(G1, G2)$, and the code word associated with the branch $(C1, C2)$ is defined by:

$$BM = (G1 - C1)^2 + (G2 - C2)^2 \tag{11}$$

For instance, the code word for the upper branch in state 0 is $(C1, C2) = 00$, and for the lower branch it is $(C1, C2) = 11$. These codewords are calculated with the aid of Matlab, for instance:

$$>> codeRate = 1/2;$$
$$>> constlen = 7;$$
$$>> codegen = [171, 133];$$
$$>> trellis = poly2trellis(constlen, codegen);$$

generates a trellis for the (2,1,7) convolutional code. Therefore, we are able to generate the next state table, and the table for output symbols for each branch, as shown in Tables 3, and 4.

$$>> trellis.nextStates(1:64,:)$$
$$>> trellis.outputs$$

Table 3: Next state table for the proposed Viterbi decoder.

Current state	Next state (Input '0')	Next state ('1')
S0	S0	S32
S1	S0	S32
S2	S1	S33
S3	S1	S33
—	—	—
S62	S31	S63
S63	S31	S63

Table 4: Look-up table for the outputs of the proposed Viterbi decoder.

Current state	Output (Input '0')	Output ('1')
S0	00	11
S1	11	00
S2	01	10
S3	10	01
—	—	—
S62	11	00
S63	00	11

5.2 The Viterbi decoder

The formal description of the Viterbi decoder is modeled as a hierarchical Action System. Thus, the top level description is fairly simple. It consists of control variables, and three subsystems: branch metric unit (BMU), path metric unit (PMU), and the trace back unit (TBU). The Viterbi decoder is defined by:

sys *ViterbiDecoder* $(enable_{decoder}, enable_{bmu} : bool)$
$|[$
const $states := 64;$
$depth := 2;$
$L := 30;$
$mem_{depth} = 2 * (L + 1)$
$D_{max} = 4;$
type $bit : bool;$
$array : bit[0..states - 1][0..D_{max}];$
$bvect : bit[0..1];$
subsys $BMU, PMU, TBU;$
init $enable_{decoder}, enable_{bmu} := F;$
actions $V1 : enable_{decoder} \rightarrow enable_{bmu} := T;$
$V2 : \neg enable_{decoder} \rightarrow enable_{bmu} := F;$
exec
do $V1 \parallel V2$ **od** $\parallel BMU \parallel PMU \parallel TBU$
$]|$

The decoder is enabled when there is valid data from the encoder, that is the variable $enable_{decoder}$ is set $true$. Then the branch metric calculation is enabled by setting the variable $enable_{bmu}$ to $true$ $(V1)$. On the contrary, when there is no data to process the BMU is disabled $(V2)$. For simplicity, most of the constant variables are defined in the top level description. Moreover, we define a type bit, which is of type $boolean$. The value $true$ indicates the logic '1', and the value $false$ indicates the logic '0'.

5.2.1 Branch Metric Unit (BMU)

The branch metric is the squared distance between the received noisy symbol Y_n, and the ideal noiseless symbol of that transition $C_{i,j}$. That is the branch metric from state i to state j at stage n is:

$$B_{i,j,n} = (Y_n - C_{i,j})^2 \tag{12}$$

Notice that, in this implementation, we apply the $Euclidean$ distance to the branch metrics generation, and multi-bit quantization for the input bits, (all though the quantization is not formally described here).

The interface of the BMU consists of the following input/output variables:

inputs:
$enable_{bmu}$: data valid variable, enables the branch metrics calculation.
din: data in variable, noisy input symbols from the encoder.
pmu_{ready}: The path metrics are ready, ready to accept new data.
outputs:
$bm0$: calculated branch metrics, for the '0' branch.
$bm1$: calculated branch metrics, for the '1' branch.
bm_{ready}: branch metric calculation is ready.
$enable_{pmu}$: enables path metric calculation.

The next state table, shown in Table 4, of the trellis for the (2,1,7) convolutional encoder is modeled as of type $array$, and the BMU receives the table as a parameter ($metrics$). The first column of the array contains the metric value if the assumed input bit is $'0'$, and the second one the value of the metric if the assumed input bit is $'1'$. The incoming symbols (din) from the encoder are defined as of type bit vector $bvect$. The output variables $bm1$, and $bm0$ are modeled as of type $array$. Notice that, the $bm0$ contains the metrics from the $'0'$ branch for each state. The external interface of the BMU is illustrated in Figure 14.

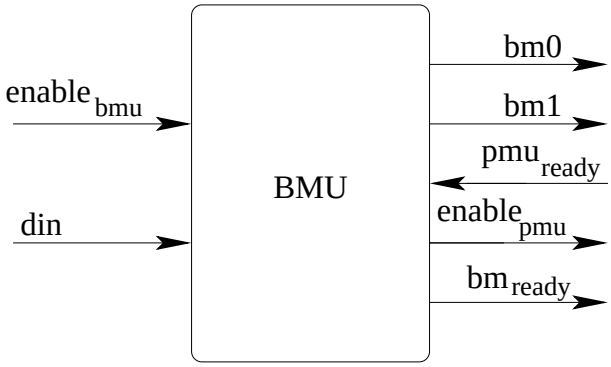

Figure 14: Branch Metric Unit

The BMU is defined by:

sys *BMU* $(din : bvect, bm0, bm1 : array, enable_{pmu} : bool)[metrics]$
$|[$
var $bm_{ready} : bool;$
proc $dist_{calc}(metrics[i,j], din) :$
$(bm0[l,(l,0..states-1] :=$
$(din - metrics[0,i,(i,0..states-1)])^2;$
$bm1[l,(l,0..states-1] :=$
$(din - metrics[1,j,(j,0..states-1)])^2);$
init $enable_{pmu}, bm_{ready} := F$
actions $B1 : enable_{bmu} \wedge \neg bm_{ready} \rightarrow dist_{calc};$
$bm_{ready}, enable_{pmu} := T;$
$B2 : pmu_{ready} \rightarrow bm_{ready} := F;$
$B3 : \neg enable_{bmu} \rightarrow enable_{pmu} := F;$
exec
do $B1 [\!] B2 [\!] B3$ **od**
$]|$

The branch metric calculation is enabled when the $enable_{bmu}$ is set to $true$, and the bm_{ready} variable is set to $false$. Thus, this indicates that there is valid data from the encoder, and that the BMU is idle. The branch metrics calculation is carried out in the procedure $dist_{calc}$, and the bm_{ready} is set to $true$ $(B1)$. Moreover, the PMU is enabled by setting the $enable_{pmu}$ variable to $true$. After the PMU is ready to accept new data it sets the pmu_{ready} variable to $true$. Then the bm_{ready} is set to $false$, which indicates that the BMU is ready to accept new data from the encoder ($B2$). The action ($B3$) disables the PMU when there is no data to process.

5.2.2 Path Metric Unit (PMU)

The path metric unit (PMU) is the core of the computation in the decoder. Thus, it adds the current branch metric to the previous path metric, compares the two metrics, and selects the smaller one as a next weight to a node. Therefore, this computation unit is denoted by Add-Compare-Select Unit (ACSU), and it corresponds to a node of the trellis.

The transition tables for the 64-state trellis are shown in Tables 3, and 4. For instance, if the current state is $S0$ and the symbol $'00'$ corresponding the input $'0'$ is received , The decoder will remain in state $S0$ and produces decoded output $'0'$. In other words, the whole decoding process is performed with aid of predetermined reference values of each state and their interconnects. As mentioned in previous section, the butterfly unit obeys the similar functionality. Therefore, we can represent the states of butterfly block, as shown in Figure 15, where the solid line represents the transition if the input bit is $'0'$, and dotted line the transition that correspond the input bit $'1'$. The relationships, shown in Figure 15, are valid for every $j, 0 \leq j \leq 31$.

The proposed decoder consists of 64 states, which are described using 32 butterfly blocks, shown in Figure 16. Notice that, each butterfly corresponds two states in trellis.

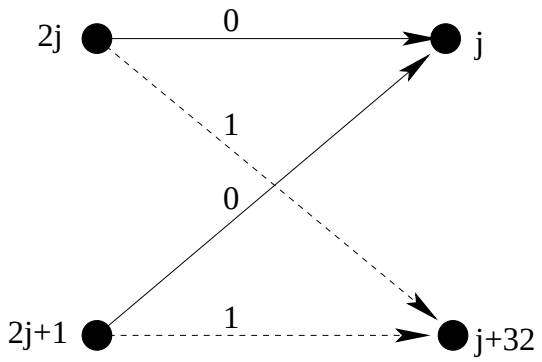

Figure 15: State relationship for the path metrics computation

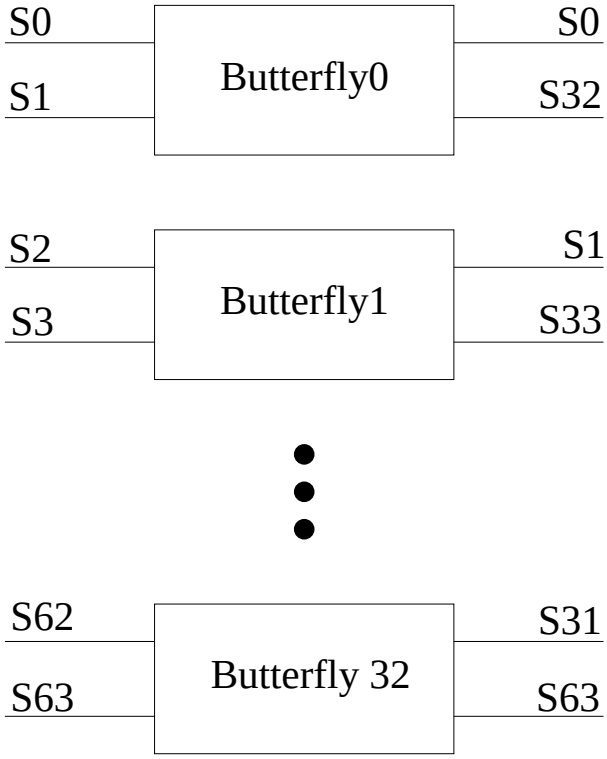

Figure 16: The butterfly blocks for the path metric calculation

The PMU consist of the state metric memory SMM, control logic, and 64 Add-Compare-Select Units ($ACSU$). The SMM is modeled as of type $smem$, which is an array. It stores the local winner for each state, which is used in the path metric calculation during next calculation cycle. The size of each cell is defined by $2D_{max} + 1$, where the D_{max} is the maximum possible difference in the path metrics. For instance, by assuming that the length of the each path metric

is four, then the size of the each cell in the given vector variable will be 9 bits. Thus, by adopting this approach the extra calculation needed for the normalization operation is avoided [5].

The simplified block diagram of the PMU is shown in Figure 17. The interconnects between the ACS-units and SMM illustrates the reading and writing events between these two sub blocks.

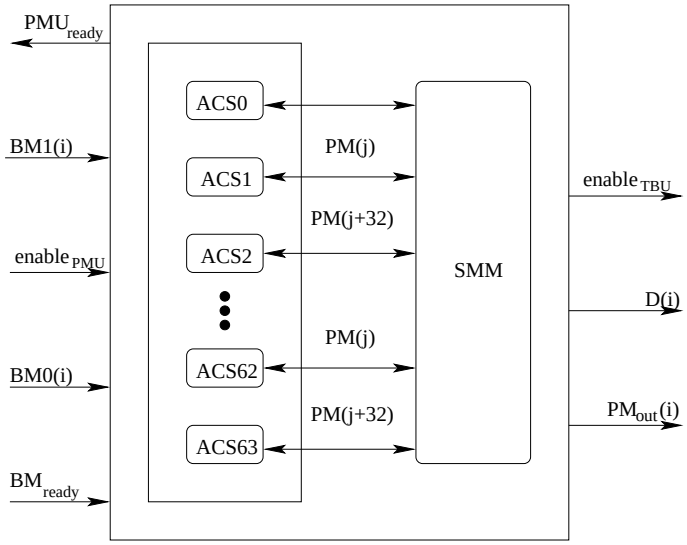

Figure 17: Path Metric Unit

The PMU is defined by:

sys $PMU\ (enable_{pmu}, pmu_{ready}, enable_{tbu})$
$|[$
type $smem : bit[0, ..., states - 1][0, ..., 2D_{max} + 1];$
var $SMM := smem;$
$enable_{acsu} := bool;$
proc $update(SMM) : SMM[i, (i, 0..states - 1)] :=$
$pm_{out}[i, (i, 0..states - 1)];$
subsys $ACSU[i];$
init $enable_{acsu}, pmu_{ready}, enable_{tbu} := F;$
actions $P1 : enable_{pmu} \wedge \neg pmu_{ready} \rightarrow$
$enable_{acsu} := T;$
$P2 : \neg enable_{pmu} \rightarrow enable_{acsu} := F;$
$enable_{tbu} := F;$
$P3 : acsu_{ready} \rightarrow update; pmu_{ready} := T;$
$P4 : pmu_{ready} \rightarrow acsu_{ready}, pmu_{ready} := F;$
$enable_{tbu} := T$
exec
do $P1 \ [\!] P2 \ [\!] \ P3 \ [\!] \ P4$ **od**$[\![\ [\!] 0 \leq i \leq states - 1 : ACSU[i]]$
$]|$

When the PMU is enabled, each ACS-unit is activated by setting the $enable_{acs}$ variable $true\ (P1)$. Similarly, if the PMU is disabled, the ACS units are disabled

by setting the $enable_{acs}$ to $false$. When the ACS calculation is ready, the state metric memory is updated $(P3)$. Finally, the trace back unit is enabled $(P4)$.

Since the ACS-units have similar structure, it is modeled as a single subsystem $ACSU$. The 64 ACS units are generated from this model by using quantified composition. The ACS units are indexed by the state number as follows: the first ACS unit is $ACSU(0)$, and the second one is $ACSU(1)$, and so on. The interface of the add-compare-select unit is:

inputs:
$enable_{acsu}$: enables ACSU when there is valid data.
bm_1 : branch metrics from the BMU (input '1').
bm_0 : branch metrics from the BMU (input '0').
outputs:
$acsu_{ready}$: acknowledgement, ready to accept new data.
pm_{out} : the chosen path metric (smallest), used in calculation at next state.
D : Decision bit, either '1' or '0', depends on, which branch the smallest metric is selected (input '1' or '0').

The formal model of ACSU consists of four actions: action A performs the addition operations, action C compares the path metrics, and outputs the decision bit, action S selects the smallest path metric, which is applied in the next calculation stage, and action U acknowledges that the ACS calculation is finished. The ACSU is defined by:

sys $ACSU\,(acsu_{ready} : bool, pm_{out} : smem, D : bit)[i, SMM]$
$|[$
var $pm_{new1}, pm_{new0} : smem;$
$dv, select, acsu_{ready} : bool;$
proc $dec_{bit}(pm_{new1}, pm_{new0}) : (pm_{new1} < pm_{new0} \rightarrow D := 1;$
$pm_{new0} < pm_{new1} \rightarrow D := 0);$
proc $min(pm_{new1}, pm_{new0}) :$
$(pm_{new1} < pm_{new0} \rightarrow pm_{out} := pm_{new1};$
$pm_{new0} < pm_{new1} \rightarrow pm_{out} := pm_{new0})$
init $acsu_{ready}, select, dv := F;$
actions $A : enable_{acsu} \rightarrow pm_{new1}, pm_{new0} :=$
$(bm1[i, (i, 0..states - 1)] +$
$SMM[j, (j, (j, 0..states - 1)],$
$(bm0[i, (i, 0..states - 1)] +$
$SMM[j + 32, (j, (j, 0..states - 1)];$
$dv := T;$
$C : dv \wedge \neg acsu_{ready} \rightarrow D := dec_{bit}(pm_{new1}, pm_{new0});$
$select := T;$
$S : dv \wedge select \rightarrow pm_{out} := min(pm_{new1}, pm_{new0});$
$acs_{ready}, select := T, F;$
$U : \neg enable_{acsu} \rightarrow dv, acsu_{ready}, select := F;$
exec
do $A \,[\!]\, C \,[\!]\, S \,[\!]\, U$ **od**
$]|$

The ACS unit consist of two adders, which calculates the sum of the incoming branch metrics, and the previous path metric from the state metric memory SMM (A). The comparison operation finds the most likely path that has the minimum metric. Thus, the decision bit D is calculated by the action C using the procedure dec_{bit}. The decision bit indicates the branch where the smallest metric came from (local winner). That is whether the local winner came from the input $'1'$ or input $'0'$ branch. The selector action S chooses the smallest path metric, which is then stored to the SMM. Notice that, this smallest metric at time t is used in the path metric calculation at time $t + 1$. By assuming that the total transition in trellis is M, and the number of states is N, a maximum of $(M - N)$ comparison operations are required, and $2N$ sums are required to initialize the metric for each state. The action U disables the operation of the ACS units if there is no data to process.

5.2.3 Trace Back Unit (TBU)

The trace back unit stores the survivor paths for each state, and performs the trace-back operation. Moreover, it outputs the decoded bit. The data structure of the trace back memory is illustrated in Table 5.

Table 5: Traceback unit data structure

State	Path Metrics	Decision bit
0	PM0	D0
1	PM1	D1
2	PM2	D2
—	—	—
62	PM62	D62
63	PM63	D63

At time t, the path metric, which is the local winner, and the corresponding decision bit is stored into the path metrics and decision bit category, respectively. Then, the same procedure is repeated at time $t + 1$. The state numbers are used as pointers, that is 0 corresponds to state $S0$, 1 corresponds to $S1$, and so on. For simplicity, the state numbers are illustrated as integers. However, in actual implementation, the length of the pointers is ($log_2(64) = 6$) 6 bits. The minimum value for the length of the survivor path is $L = 5(log_2(N))$, where the N is the number of states [5]. In other words, the number of stages to store in the memory before the trace back operation can begin is $L + 1$. In this case, we define that, the number of stages that has to be stored before the trace back is 31, and the overall length L for the trace back memory is 62. Thus, we can carry out the traceback from the memory location 30 down to 0, and at the same time write new data to

memory locations 61 down to 31. The simplified block diagram of the trace back unit TBU is shown in Figure 18.

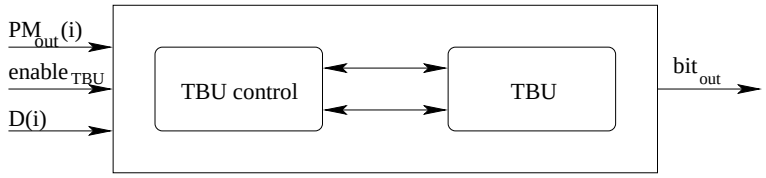

Figure 18: Trace Back Unit

The input / output signals of the TBU are:

Inputs:
$enable_{tbu}$: enables the trace back unit.
$PMU_{out}(i)$: The local winners (smallest metric) from each ACSU.
$D(i)$: Decision bits from each ACSU.
Output:
bit_{out}: decoded bit out

The trace back unit is defined by:

sys *TBU* $(decbit : bit)$
$|[$
type $tracemem : [0..states-3][0..2D_{max}+1];$
$decisionmem : [0..states-3];$
var $tr_{start}, inc, traceback : bool;$
$count : integer$
proc $gmin(trmem[k(k,0..states-1), tr_{index}]) :$
$PM_{min} := trmem[0, tr_{index}];$
proc $trmem[k(k,0..states-1), tr_{index}] \leq PM_{min} \rightarrow$
$PM_{min} := trmem[k, tr_{index}]; cstate := PM_{min}(k);$
subsys $TBU_{control};$
init $trmem := tracemem; decmem := decisionmem;$
$tr_{start} := F$
$count := 0, inc := F;$
actions $T1 : enable_{TBU} \rightarrow$
$trmem[i, count, (i, 0..states-1)] :=$
$PM_{out}[i, (i, 0..states-1)];$
$decmem[i, count, (i, 0..states-1)] :=$
$D[i, (i, 0..states-1)]; inc := T;$
$T2 : traceback \rightarrow$
$gmin(trmem[k, tr_{index}, (k, 0..states-1)]);$
$tr_{start} := T;$
$T3 : traceback \wedge tr_{start} \rightarrow$
$dec_{bit} := decmem[cstate, tr_{index}];$
$bit_{out} := decbit; tr_{index} := tr_{index} - 1; tr_{start} := F$
exec
do $T1 \; [] \; T2 \; [] \; T3$ **od**$\| \; TBU_{control}$
$]|$

The $TBU_{control}$ is a subsystem that controls the memory operations.

sys $TBU_{control}$
$|[$
actions $C1 : enable_{tbu} \wedge inc \rightarrow count := count + 1; inc := F;$
$C2 : count = L + 1 \rightarrow tr_{index}, traceback := L + 1, T;$
$C3 : count = 2 * (L + 1) \rightarrow$
$tr_{index}, traceback := 2 * (L + 1), T; count := 0;$
$C4 : tr_{index} := 0 \vee trace_{index} := L \rightarrow traceback := F;$
exec
do $C1 \, [\!] \, C2 \, [\!] \, C3 \, [\!] C4$ **od**
$]|$

Before the trace back operation is started, the global winner of the stage $L + 1$ has to be determined. That is the smallest metric in given stage is chosen to be the global winner, and the starting point of the traceback operation. This is carried out in the procedure $gmin$.

The TBU is enabled by the PMU when there is data to be stored. From each state, the TBU stores the smallest metric (local winner), and the corresponding decision bit $(T1)$. The $TBU_{control}$ is enabled to count the number of stages stored in the memory $(C1)$. When the number of stages reaches the survivor depth, that is 31 stages, the $TBU_{control}$ enables the trace back operation $(C2)$. The trace back is carried out by calculating the global winner at stage 31, or 62. That is the smallest metric from the local winners at that stage. This is carried out in the procedure $gmin$. The procedure returns the pointer to the global winner, that is the state number $cstate$. Then, the decision bit corresponding the global winner state is read from the decision memory $decmem$, and outputted $(T3)$. This is carried out as long as the tr_{index} is either 0 or L $(C4)$. In parallel with the trace back operation the incoming path metrics are written into the memory locations from 32 to 62. Thus, the trace back is carried out alternately with the memory write operation, and therefore the TBU is enabled as long as the PMU is enabled.

6 Conclusions and Future Work

In this paper, we presented a formal specification of asynchronous Viterbi decoder. The decoder is 64-state $1/2$- rate Viterbi decoder, and the generator polynomials used are the industrial standards $(171_8, 133_8)$. The asynchronous implementation is chosen due to its potential for low-power, and low-noise behavior.

The formal model presented will be used as a case study for the formal power consumption model. This model will be included into the formal design flow in the near future. The purpose of this model is to be able analyze the power consumption of a digital system starting from the formal model presented here down to the gate-level implementation. Moreover, since the first power estimation is done at early design phase (formal description), we are able to compare the

power consumption, for instance between different design solutions, and choose the most optimal one. The asynchronous Viterbi decoder will be used as a case study to compare and analyze the formal specification of the power consumption with the existing early power estimation methods.

References

[1] R. J. R. Back and K. Sere. *From Modular Systems to Action Systems*, in Proc. of Formal Methods Europe' 94, Spain, October 1994. Lecture notes on computer science, Springer-Verlag.

[2] R. J. Back and J. von Wrigth. *Refinement Calculus: A Systematic Introduction*, Springer-Verlag, April 1998.

[3] E. W. Dijkstra. *A Discipline of Programming*, Prentice-Hall International, 1976.

[4] G. D. Forney Jr. *Maximum-Likelihood Sequence Detection in the Presence of Intersymbol Interference*, IEEE Trans. on Information Theory, IT-18(3): 363-378, May 1972.

[5] H. -L. Lou. *Implementing the Viterbi Algorithm*, IEEE Signal Processing Magazine, September 1995.

[6] J. G. Proakis. *Digital Communications*, Third Edition, McGraw-Hill 1995.

[7] P. A. Riocreux, L. E. M. Brackenburry, M. Cumpstey, and S. B. Furber. *A Low-Power Self-Timed Viterbi Decoder*, in Proc. 7th International Symposium on Asynchronous Circuits and Systems, March 11-14, 2001, Salt Lake City, Utah, pp. 15-24.

[8] M. S. Ryan, and G. R. Nudd. *The Viterbi Algorithm*, Warwick Research Report 238, University of Warwick, England, February 1993.

[9] C. B. Shung, P. H. Siegel, G. Ungerboeck, and H. K. Thapar. *VLSI Architectures for Metric Normalization in the Viterbi Algorithm*, in Proc. International Conference on Communications, Atlanta, Georgia, April 1990.

[10] J. Tuominen, T. Säntti and J. Plosila. *Towards a Formal Power Estimation Framework*, Turku Center for Computer Science Technical Report Series, Number 672, March 2005, ISBN 952-12-1517-8.

[11] J. Tuominen and J. Plosila. *Formal Energy Estimation Framework*, Turku Center for Computer Science Technical Report Series, Number 696, June 2005, ISBN 952-12-1578-X.

[12] J. Tuominen, P. Liljeberg, and J. Isoaho. *Self-Timed Approach for Reducing On-Chip Switching Noise,* in Proc. IFIP SoC-VLSI 2003, December 2003, Darmstadt, Germany.

[13] A. J. Viterbi. *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm*, IEEE Transactions on Information Theory, April 1967; IT(2): pp. 260-267.

[14] A. J. Viterbi. *Convolutional Codes and Their Performance in Communication Systems*, IEEE Transactions on Communications Technology, October 1971; COM - 19(5): pp. 751-772.

Turku
Centre *for*
Computer
Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 952-12-1602-6
ISSN 1239-1891