Asynchronous Logic Implementation of Tree-Structured SISOs

Peter A. Beerel, Keith M. Chugg, Georgios D. Dimou, Pankaj Golani, and Mallika Prakash
Ming Hsieh Dept. of Electrical Engineering
University of Southern California
Los Angeles, CA 90089-2565, USA
Email: {pabeerel, chugg, dimou, pgolani, mallikap}@usc.edu

Abstract— **Tree-structured soft-in/soft-out (SISO) processors provide an exponential speed-up relative to the standard forward-backward algorithm (FBA). These Tree-SISOs were originally described analogously to fast tree-structured adders and later as standard message-passing on a binary tree graphical model for a finite state machine (FSM). In this paper, we summarize and unify these theoretical results and also summarize recent efforts to implement high-speed iterative decoders based on Tree-SISOs. Specifically, we design a Tree-SISO based on a traditional synchronous design flow and another based on our asynchronous design flow. The asynchronous design offers significant advantages in terms of throughput/area of the resulting high-speed iterative decoder at the cost of some additional energy consumption.**

I. Introduction

The primary building block for iterative decoding of modern codes is the forward-backward algorithm (FBA) (*e.g.,* [1], [2], [3]). The FBA implements the locally optimal soft-in/soft-out (SISO) decoding for a code modeled by a trellis (*e.g.,* a convolutional code or block code). In this paper we summarize the so-called Tree-SISO algorithm which computes the same SISO results as the FBA with lower latency. These two algorithms can be viewed and compared in two frameworks. First, one can view the SISO computation problem as a prefix/suffix computation and draw upon the various architectures for implementing this computation (*e.g.,* adder circuits). With this view, the FBA is a serial prefix/suffix computational architecture and the Tree-SISO is a parallel prefix/suffix computational architecture. Second, both the FBA and the Tree-SISO can be viewed as standard message-passing on an acyclic graphical model of the code. If the diameter of the graph associated with the FBA (*i.e.,* a trellis) is K, then the graph associated with the Tree-SISO has diameter $\log_2(K)$. This yields an exponential reduction in latency in executing a complete activation schedule on the acyclic graph.

Both the FBA and Tree-SISO can be implemented using standard digital logic techniques. In this paper, however, we focus on the potential advantages of implementing the Tree-SISO using asynchronous digital logic methods, where no global clock is used to time the logic components. Asynchronous methods have the potential for better area/power/speed trade-offs, especially in data-driven circuits such as iterative decoders. As way of comparison, we designed two decoders based on Tree-SISOs for a serially concatenated convolutional code (SCCC). The first is based on standard cell synchronous design methods. The second is based on recently developed standard cell asynchronous design flow called Static Single Track Full Buffer (SSTFB). These designs were implemented using the IBM 0.18 micron process and the decoders achieve data rates near OC-12. The asynchronous design shows significant advantages in terms of throughput per area. These advantages are most notable at smaller block sizes when SISO latency can be a dominant factor in the iterative decoder throughput due to pipeline overheads.

In Section II we review the Tree-SISO architecture and the relationship between the two interpretations. Section III motivates asynchronous Tree-SISO designs and Section IV describes our specific designs. Comparisons of iterative decoders based on synchronous and asynchronous Tree-SISO designs are made in Section V.

II. Tree-SISO Overview

The SISO operation for a system modeled as a finite state machine (FSM) is typically performed using the FBA, which is standard message-passing running on a specific graphical model for the FSM [2]. There are several forms of message-passing depending on the optimality criterion and the format used to store soft information on variables. In the following, soft information is assumed to be stored in the metric domain (*i.e.,* negative-log of probabilities) and min-sum processing is assumed. All of the algorithms described are so-called semi-ring algorithms [3] so that they may be directly translated to different message formats and/or optimality criteria.

At time index k, the FSM has state s_k, input a_k, output x_k and state transition $t_k = (s_k, a_k, x_k, s_{k+1})$. Incoming metrics on a_k and x_k define a transition metric $\mathrm{M}_k[t_k]$ for each transition t_k. Define the minimum sequence metric (MSM) conditioned on a variable u as the minimum summed metric over all valid state transitions from time k_1 to k_2

$$\mathrm{MSM}_{k_1}^{k_2}[u] \triangleq \min_{t_{k_1}^{k_2}:u} \sum_{k=k_1}^{k_2} \mathrm{M}_k[t_k] \tag{1}$$

where v_i^j indicates a sequence $\{v_n\}_{n=i}^{j}$, and $t_{k_1}^{k_2} : u$ is shorthand for all paths consistent with the particular value of u. Note that $\mathrm{MSM}_{k_1}^{k_2}[u]$ should be viewed as a table of values, one for each of the finite values that u can take.

The FBA computes the forward and backward state metrics via the forward and backward state metric recursions

$$\mathrm{F}[s_{k+1}] \triangleq \mathrm{MSM}_0^k[s_{k+1}] = \min_{t_k:s_{k+1}} (\mathrm{F}[s_k] + \mathrm{M}_k[t_k]) \tag{2a}$$

$$\mathrm{B}[s_k] \triangleq \mathrm{MSM}_k^{K-1}[s_k] = \min_{t_k:s_k} (\mathrm{M}_k[t_k] + \mathrm{B}[s_{k+1}]) \tag{2b}$$

where it is assumed that k runs from 0 to $K-1$. For a given k, once $\mathrm{F}[s_k]$ and $\mathrm{B}[s_{k+1}]$ are available, the soft-out metrics for a_k and x_k can be computed (*i.e.,* the completion operation). Note that there is data dependency in the state metric update – *e.g.,* $\mathrm{F}[s_{10}]$ cannot be updated without $\mathrm{F}[s_9]$ – resulting in the so-called add-compare-select (ACS) bottleneck. So that, in its direct form, the FBA requires latency K times the latency of a state metric update.

The Tree-SISO architecture allows for the same computation with latency $\mathcal{O}(\log_2(K))$. This was described in [4], [5] using analogies to tree-adder architectures from the digital circuits literature. A similar architecture was presented from the perspective of standard message-passing on an acyclic graphical model in [6]. In the next two sections, we briefly summarize these results and describe their relation.

A. Parallel Prefix/Suffix Architectures

A prefix computation [7, Section 29.2.2] is

$$z_k = y_0 \circ y_1 \circ \cdots \circ y_k \qquad k = 0, 1, \ldots K-1 \tag{3}$$

where $\circ$ is any *binary associative* operator. To compute $\{z_k\}_{k=0}^{K-1}$, one may specify an association (*i.e.,* assign parentheses) in a number of ways. For example, assigning parentheses as

$$z_3 = (((y_0 \circ y_1) \circ y_2) \circ y_3) \tag{4}$$

yields a *serial prefix architecture*. This architecture has high latency and low hardware logic complexity because it executes each $\circ$ once at a time. A more parallel architecture can result from

$$z_3 = (y_0 \circ y_1) \circ (y_2 \circ y_3) \tag{5}$$

in which the two $\circ$ operations in parentheses can be carried out in parallel using parallel hardware to reduce latency. This illustrates the basic idea behind *parallel prefix architectures.*

Various parallel prefix architectures have been studies in the context of circuits for fast adders (*e.g.,* [7], [8]) where fan-out from a particular $\circ$ operator is a primary design consideration. In the context of SISO processing for an FSM, the analogous suffix computation

$$z_k = y_k \circ y_{k+1} \circ \cdots \circ y_{K-1} \qquad k = 0, 1, \ldots K-1 \tag{6}$$

is also of interest. In particular, the forward and backward state metrics can be obtained using prefix/suffix computations on the multi-stage transition metrics defined as

$$\mathrm{C}[s_k, s_m] = \mathrm{MSM}_k^{m-1}[s_k, s_m] \tag{7}$$

which is a table of the metrics of the best way to get from state s_k at time k to state s_m at time $m \geq k$. Note that if $m = k$, these are the transition metrics if there are no parallel state transitions. The *C-fusion* operator fuses $\mathrm{C}[s_k, s_l]$ with $\mathrm{C}[s_l, s_m]$ to obtain $\mathrm{C}[s_k, s_m]$ by marginalizing out the intermediate state s_l

$$\mathrm{C}[s_k, s_m] = \mathrm{C}[s_k, s_l] \circ \mathrm{C}[s_l, s_m] \iff \tag{8}$$
$$\mathrm{C}[s_k, s_m] = \min_{s_l}\left(\mathrm{C}[s_k, s_l] + \mathrm{C}[s_l, s_m]\right) \quad \forall s_k, s_m$$

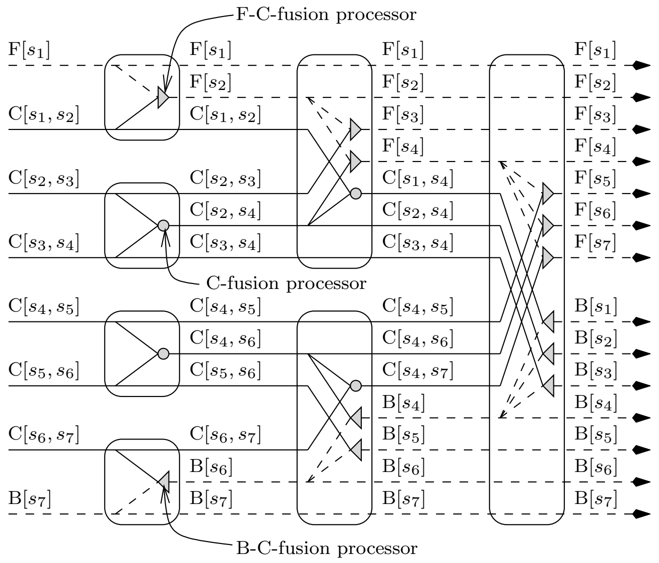

Fig. 1. A Tree-SISO processing flow diagram for $K = 8$.

This is a binary associative operator so that

$$\mathrm{C}[s_0, s_k] = \mathrm{C}[s_0, s_1] \circ \mathrm{C}[s_1, s_2] \circ \cdots \circ \mathrm{C}[s_{k-1}, s_k] \tag{9a}$$

$$\mathrm{C}[s_k, s_K] = \mathrm{C}[s_k, s_{k+1}] \circ \mathrm{C}[s_{k+1}, s_{k+2}] \cdots \circ \circ \mathrm{C}[s_{K-1}, s_K] \tag{9b}$$

define a prefix/suffix computation.

The forward and backward state metrics can be obtained by marginalizing $\mathrm{C}[s_0, s_k]$ and $\mathrm{C}[s_k, s_K]$ over the initial and final state

$$\mathrm{F}[s_k] = \min_{s_0} \mathrm{C}[s_0, s_k] \tag{10a}$$

$$\mathrm{B}[s_k] = \min_{s_K} \mathrm{C}[s_k, s_K] \tag{10b}$$

Therefore, applying parallel prefix/suffix architectures to the problem in (9) yields SISOs with latency logarithmic in the block size K by first producing the $\mathrm{C}[s_0, s_k]$ and $\mathrm{C}[s_k, s_K]$ values for each value of k, then marginalizing as in (10) (see [3, Fig. 2.48]). Some computation can be saved by marginalizing as early as possible in the computation. This is accomplished via

$$\mathrm{F}[s_k] = \min_{s_m}\left(\mathrm{F}[s_m] + \mathrm{C}[s_m, s_k]\right) \tag{11a}$$

$$\mathrm{B}[s_k] = \min_{s_m}\left(\mathrm{C}[s_k, s_m] + \mathrm{B}[s_m]\right) \tag{11b}$$

which are referred to as "FC" and "BC" fusion operations, respectively. A minimal latency Tree-SISO architecture is shown in Fig. 1 using these concepts.

B. Message-Passing on Graphs

The FBA is equivalent to standard message-passing on the graphical model with linear topology shown in Fig. 2(a). The graphical modeling convention is that edges are variables and boxes are constraints on variables connected. For standard message-passing, incoming messages at each constraint processor are combined (*e.g.,* summed), then marginalized (*e.g.,* minimized) over all allowable local configurations

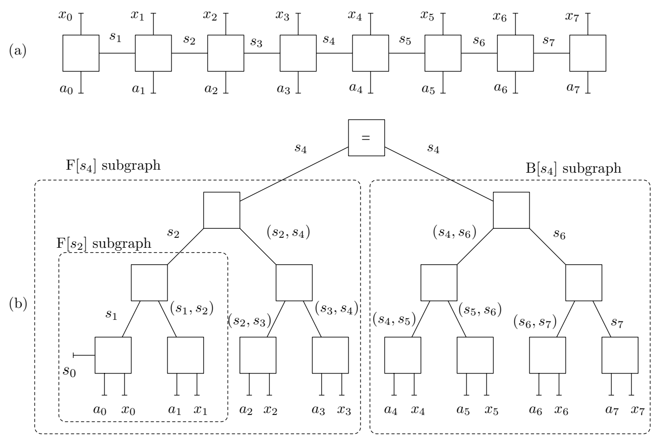

Fig. 2. Graphical models that yield (a) the FBA and (b) the FB-Tree-SISO for $K = 8$.

(*e.g.,* [2], [3]). A Tree-SISO architecture can be viewed as the result of standard message-passing on the binary tree graph shown in Fig. 2(b). Note that multiple edges on the graph are labeled with the same variable. This is a notational simplification used to keep the figure less cluttered and more precisely these are distinct variables that are constrained to take equal values through the constraints. A more precise notation would distinguish these variables as $s_k^l(i)$ and $s_k^r(i)$, where i is the depth of the tree and l and r denote "left" and "right", respectively. For example, indexing i from 0 at the bottom, the two s_2 variables at depth 1 would be $s_2^l(1)$ and $s_2^r(1)$ and constraints deeper in the tree ensure that these variables are equal.

A natural, convergent schedule on the graph of Fig. 2(b) is to simultaneously activate nodes at each level and pass messages first upward and then back downward. This is often called an inward-outward or forward-backward schedule and the resulting Tree-SISO was called the forward-backward Tree-SISO (FB-Tree-SISO) in [6] for this reason. Note that at the first level, upward messages are the one-step transition metrics with parallel transitions marginalized if necessary. Processing at nodes at subsequent depths on the upward recursion correspond to C, FC, or BC fusion operations. At the bottom level on the downward recursion, the messages correspond to the sum of the forward and backward state metrics and the bottom nodes use these to perform completion (*i.e.,* final soft-out computations). For example, the node with a_5 and x_5 connected will receive the message $\mathrm{F}[s_5] + \mathrm{B}[s_6]$ from above. For an more detailed example with message interpretation see [3, Figs. 2.64-2.65].

Note that on the upward recursion, the messages on s_4 from the left and right are $\mathrm{F}[s_4]$ and $\mathrm{B}[s_4]$, respectively. Thus, if only the forward and backward state metrics for s_4 were desired, one need only run the upward recursion. Similar tree-structured graphs could be defined to obtain the forward and backward state metrics at different times. For example, several of these are identified as subgraphs in Fig. 2(b) – *i.e.,* running upward on the subgraph label $\mathrm{F}[s_2]$ yields this forward state

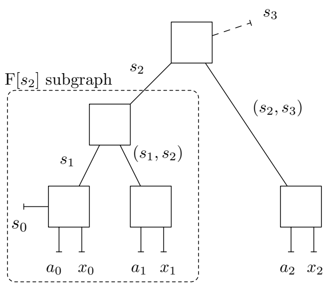

Fig. 3. A graphical model yielding $\mathrm{F}[s_3]$ on the upward recursion.

metric. As another example, consider the graph in Fig. 3, which is targeted to produce $\mathrm{F}[s_3]$ at the end of the inward recursion. Note that this graph and the $\mathrm{F}[s_4]$ subgraph in Fig. 2(b) both contain the $\mathrm{F}[s_2]$ subgraph. Therefore, if one were running a set of parallel upward recursions on graphs targeting different time indices, it would be natural to share common information (messages). For example, $\mathrm{F}[s_2]$ could be used by two different upward recursions, one targeting $\mathrm{F}[s_4]$ and the other targeting $\mathrm{F}[s_3]$. Following this logic leads precisely to the Tree-SISO structure shown in Fig. 1. For example, the computations leading to $\mathrm{F}[s_4]$ and $\mathrm{B}[s_4]$ in Fig. 1, can be identified with the upward message updates on the $\mathrm{F}[s_4]$ and $\mathrm{B}[s_4]$ subgraphs shown in Fig. 2. Similarly, the sharing of the $\mathrm{F}[s_2]$ subgraph in the $\mathrm{F}[s_3]$ and $\mathrm{F}[s_4]$ subgraphs (see Figs. 2 and 3) corresponds to the fan-out of the $\mathrm{F}[s_2]$ value in Fig. 1. The rest of the fan-out connects can be similarly interpreted. Other parallel prefix tree architectures (*e.g.,* [7], [8]) can be viewed as a trade-off between the extremes of the Tree-SISO in Fig. 1 and the FB-Tree-SISO. Finally, note that this is analogous to running forward and backward only to time k in the FBA, which can be executed in parallel for different k. The difference is that when this is performed on a graph of the form in Fig. 2(a), there is no shared subgraph structure and all targeted inward recursions process independently (this corresponds to a flooding schedule on the graph in Fig. 2(a)).

As a final, simple example, consider a single parity check (SPC) constraint. The graphs corresponding to those in Fig. 2 for this special case are shown in Fig. 4. In this simple case, the constraints all reduce to local SPC constraints for which the min-sum message update rule is trivial. For a set of incoming messages on the 8 variables constrained by the SPC, a set of convergent messages is shown. Since all variables involved (including state variables) are binary, messages are shown as normalized metrics (*i.e.,* negative log-likelihood ratio format). Note that the output messages on the visible variables are the same in each case and that the forward and backward metrics $\mathrm{F}[s_4]$ and $\mathrm{B}[s_4]$ can be found in each of the two algorithms as described above.

III. Design Considerations

A. Advantages of Tree-SISO in high-speed decoding

A given SISO architecture in a given process will have a maximum speed. If the required throughput is higher than that provided by a single SISO, M processors can be used,

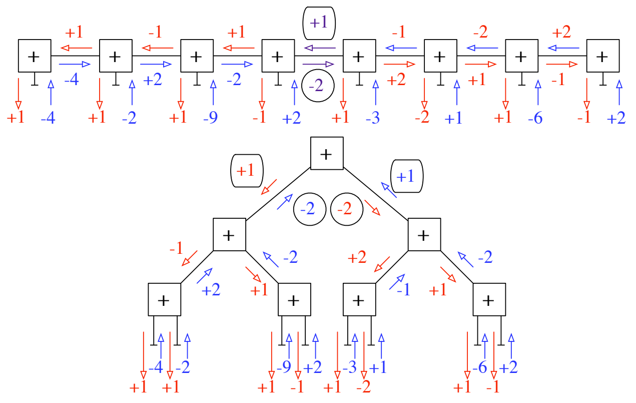
Fig. 4. Linear (top) and binary tree (bottom) models for a degree 8 SPC with convergent min-sum mesages shown.

each processing a subset of the trellis sections. The parameter M is often called the degree of parallelism. The speed of the iterative decoder can therefore be increased by increasing the speed of an individual SISO unit and/or using parallel SISOs. A rough analysis suggests that the throughput increases linearly with M, but this is not achieved in practice. In the following, we describe why it is more effective to increase the SISO speed than to use large M and how the Tree-SISO offers advantages toward this end.

First, increasing M causes issues related to memory access. After being processed, messages have to be interleaved between SISO modules. The interleaver should have a random-like structure for good code performance, so that for large M many bits per cycle have to first be stored into a RAM structure and then retrieved in random order. Multiple banks of RAM can be used, each receiving data corresponding to one processed bit. As the data comes out of those banks in random order, it then has to be multiplexed and distributed back to the SISOs. Constraints should be placed on the interleaver to ensure it is clash-free. This not only adds significant complexity to the decoder, due to the crossbar switch that has to be built into the interleaver, but also places constraints on the interleaver design that could yield poor decoding performance. From a hardware perspective, it also requires the instantiation of many more RAM cores that are extremely small and shallow, that consequently require significantly more area and power than fewer larger and narrower RAM instances.

A second problem with increasing M is also a consequence of the interleaver presence. The entire block of data has to be written into the interleaver before the data can be read to start the next SISO process. This is due to the random-like structure of the interleaver, which implies that some of the first bits of data to be fetched are likely to be among the last bits of data previously stored. Consequently, as the degree of parallelism is increased the processing time can be linearly reduced, but the pipeline latency remains constant yielding diminishing benefits.

Performance degradation is more pronounced in cases with small data block sizes where the pipeline latency is comparable

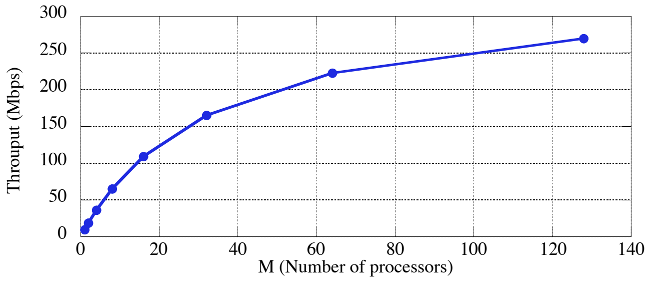
Fig. 5. Throughput vs. number of processors M.

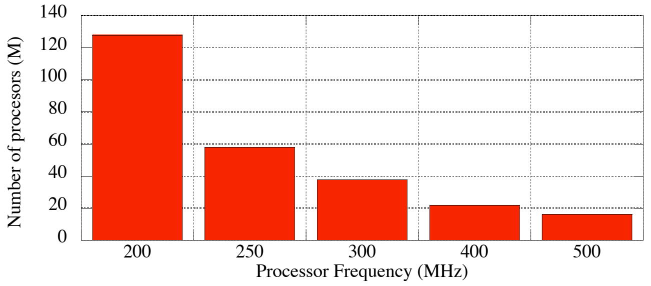
Fig. 6. Number of processors M vs. processor frequency.

to, or in extreme situations larger than, the actual processing time. This occurs every SISO operation. Figure 5 illustrates this point by showing the achievable decoder throughput as a function of M. The plot assumes that each processor runs at 100 MHz for 5 iterations for a code that has two convolutional codes and a data block size of 2 Kbits. Figure 6 also illustrates this point by showing the number of processors needed to achieve a throughput of 540 Mbps with varying processor frequency (assuming 5 iterations and 2 Kbit data block size). Thus, the required M decreases more quickly with the processor frequency than the linear rate predicted by a rough analysis. Alternatively, it can be concluded that increasing the processor speed is more effective than increasing the number of processors for increasing the overall decoder throughput.

The Tree-SISO has an advantage, relative to the FBA-based designs, with regard to the achievable clock frequency. The ACS bottleneck is a well-known problem in the design of high-speed implementations of the Viterbi algorithm and the FBA. This ACS operation usually limits the speed of a standard FBA implementation, since it is generally accepted that this operation cannot be pipelined any further due to the underlying data dependency. Other architectures have been proposed to resolve this dependency [9], but suffer from significantly greater logic requirements. In the Tree-SISO architecture, however, the ACS-bottleneck is inherently broken since the state metrics can be computed largely in parallel. This enables finer pipelining of the unit without additional logic and therefore higher clock speeds.

B. Advantages of asynchronous circuits

Based on the above argument, one should focus on increasing the processor speed rather than using large M. However, this inevitably leads to designs that have very deep pipeline structures. These structures suffer from larger pipeline overhead, meaning that several processing cycles are wasted loading the pipeline before any useful results are produced. For larger block sizes this latency is small as compared to the total processing time for the block, but for smaller block sizes it can become significant. In synchronous design, each pipeline stage requires a full clock cycle to be loaded with data. Therefore, the pipeline overhead is equal to as many cycles as there are pipeline stages. In asynchronous design, however, each pipeline stage is associated with a forward latency that represents the amount of time required for the data to propagate through the pipeline stage. It is also associated with a backward latency that is associated with the additional amount of time it takes the pipeline stage to communicate with its neighbors or "handshake" and return to the state where it is ready again to receive data. The local cycle time of a pipeline stage is equal to the sum of the two latencies. When a pipeline is empty the data propagate into the pipeline in time that is proportional to the forward latency, which is a fraction of the total pipeline latency. Therefore the pipeline overhead in an asynchronous design is much smaller since the pipeline can be loaded in a fraction of the time. Additionally, in many asynchronous design styles each gate holds the data just as a sequential element does in synchronous designs. So a gate could be perceived as a gate followed by a flip-flop. Therefore some of the additional pipelining comes inherently with the design style. It should be noted however that usually these gates are substantially larger than their synchronous counterparts.

IV. Tree-SISO Designs

We have designed two Tree-SISOs 8 trellis sections wide, one that uses a synchronous design style with a standard cell Artisan library and one that uses our asynchronous Static Single Track Full Buffer (SSTFB) design style. This is an evolution of the Single Track Full Buffer (STFB) library presented earlier [10], [11], which has proven its functionality and performance, achieving approximately 1.2GHz equivalent performance in a 0.25μm TCMC process. SSTFB has similar advantages while being more robust to higher process variability, crosstalk noise, and leakage currents [12], [13]. Both designs were designed for a 0.18μm IBM technology with an ASIC-like methodology using commercial tools. Both STFB and SSTFB are a dual-rail design styles, meaning that for every signal the gates use both a true and a false rail to characterize the data, and our library gates internally use dynamic logic that requires less area and achieves better performance than typical CMOS libraries. The handshaking between gates is also encoded on the data rails. The transmitter gate raises one of the two rails (true or false) and when one of them is asserted the following gate can detect the presence of data. It then stops driving the wires and the receiver gate assumes this task. When the receiver has used the data it resets the wires indicating it is ready to receive new data. Figure 7 shows a typical STFB cells transistor-level diagram of a n-bit input 1-bit logic function.

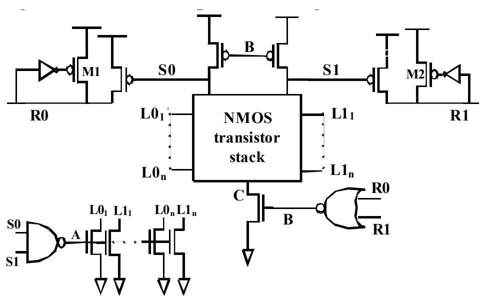

Fig. 7. Typical STFB transistor-level schematic.

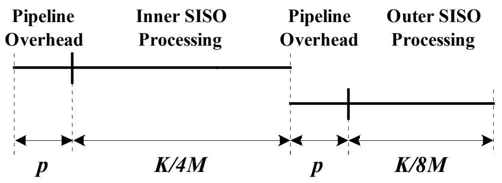

Fig. 8. Execution schedule for each decoder iteration.

A. Iterative Decoder Considerations

We chose a simple SCCC structure with two 2-state convolutional codes to reduce the size of the decoder circuit. The data is first encoded using a rate $1/2$ non-recursive convolutional code with polynomials $[1 + D, 1 + D]$ and the results are interleaved. A rate 1 accumulator (*i.e.,* $[1/(1+D)]$) is used to encode the interleaved bits before transmission. This yields an overall code rate of $1/2$. Puncturing could be used to achieve higher code rates and increase flexibility, with minor modifications to the design, but this was not considered for simplicity. For a block size of K bits the outer code has a trellis length of K and the inner trellis has $2K$ sections. Therefore, the throughput is

$$T = \frac{fK}{I\left(2p + \frac{3K}{8M}\right)} \tag{12}$$

where f is the clock frequency in Hz (or in the case of the asynchronous design the equivalent throughput), I is the number of iterations and $8M$ is the number of bits that can be processed in parallel. Finally p is the pipeline latency in terms of cycles. Each SISO operation has to finish and store data back into memory, therefore the pipeline overhead is present for every SISO activation (half-iteration). The execution schedule for each iteration is shown in Figure 8.

V. Design Results and Comparisons

The frequency of the post-place-and-route synchronous Tree-SISO core is 475 MHz. From post-layout simulations, the asynchronous Tree-SISO core frequency was estimated to be approximately 1.15 GHz. Thus, we expect the asynchronous core to run 2.4 times faster then its synchronous counterpart.

Block Size (bits)	Async. T (Mbps)	Sync. T (Mbps)	M_{sync}	Sync. area (mm^2)	T/area ratio (S/A)	Energy ratio (S/A)
512	383	-	-	-	-	-
768	418	415	11	27.06	3.91	1.23
1024	438	440	6	14.76	2.13	0.66
2048	471	519	4	9.84	1.28	0.4
4096	490	513	3	7.38	1.03	0.32

TABLE I
THROUGHPUT (T) PER AREA AND ENERGY CONSUMPTION COMPARISON.

A. *Area comparison*

The logic area of the synchronous and asynchronous Tree-SISO cores were 2.46 mm^2 and 6.92 mm^2, respectively. Both asynchronous and synchronous cores implement the exact same function. Due to the performance advantage of the asynchronous core, the synchronous core must be instantiated many times in order to match the throughput (*i.e.,* $M_{\text{sync}} > 1$ is required while $M_{\text{async}} > 1$). Substituting into (12), shows that for equivalent throughput with 6 iterations and pipeline latencies of 60 cycles for the synchronous design and 32 equivalent cycles for the asynchronous design, the number of required synchronous cores varies from $M_{\text{sync}} = 11$ for a block size of 768 bits to $M_{\text{sync}} = 3$ for a block size of 4 Kbits.

B. *Throughput/area comparison*

The ratio of throughput to area provides a fair comparison at different block sizes. For the commonly used block size of $K = 1024$, Table I shows that the asynchronous design has a throughput to area ratio that is 2.13 larger than that of the synchronous design. The advantages are more significant for smaller block sizes, and for block sizes of 512 or smaller, the synchronous design cannot match the throughput of the asynchronous counterpart, regardless of the M_{sync}. As the block size increases, latency becomes less of a critical factor and the two designs become more comparable.

C. *Energy Comparisons*

From the post-layout spice simulation the power consumed by the complete asynchronous Tree SISO is estimated at 15.5 W. The power for a single synchronous core ($M_{\text{sync}} = 1$) is 1.72 W. The last column in Table I translates the M_{sync} requirements into energy consumption comparisons showing that the synchronous design is more energy efficient except at small block sizes. Although these calculations were based on peak power and are likely conservative, the ratio should be representative of the relative power consumption of the two designs. Leakage power comparisons have not been included in the calculations, but given the smaller area of the asynchronous design, we expect it to have an advantage in that respect.

VI. CONCLUSIONS

The Tree-SISO architecture was introduced using the observation that the computation of forward and backward state metrics is closely elated to a prefix/suffix computation. Subsequent work showed that similar Tree-SISO structures can be viewed as standard message-passing on an alternative, binary tree-structured, graphical model for an FSM. In this paper we summarized these results and showed the specific relation between the two interpretations.

Our circuit design results demonstrate that SSTFB asynchronous iterative decoder is beneficial for small to medium block sizes. Preliminary comparisons show that the asynchronous iterative decoder can offer more than a $2X$ advantage in throughput per area for block sizes of 1 Kbits or less and smaller energy per block for block sizes of 768 bits or less. Thus, the asynchronous design is particularly useful in low latency wireless applications in which block sizes are often small. This chip design also motivates a number of areas of future work. The current SSTFB library has only one size per cell. While this is sufficient to achieve high performance, multiple sizes for each cell can significantly reduce the overall capacitance and power consumption. In addition, 64% of the cell instances in the Tree SISO design are dual-rail buffers for slack matching [14]. If these are replaced by 1-of-4 or 1-of-8 buffers (that have less switching activity per bit), significant reductions in power consumption are likely.

REFERENCES

[1] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Information Theory*, vol. IT-20, pp. 284–287, March 1974.

[2] N. Wiberg, *Codes and Decoding on General Graphs*. PhD thesis, Linköping University (Sweden), 1996.

[3] K. M. Chugg, A. Anastasopoulos, and X. Chen, *Iterative Detection: Adaptivity, Complexity Reduction, and Applications*. Kluwer Academic Publishers, 2001.

[4] P. A. Beerel and K. M. Chugg, "An $O(\log_2 N)$-latency SISO with application to broadband turbo decoding," in *Proc. IEEE Military Comm. Conf.*, (Los Angeles, CA), pp. 194–201, October 2000.

[5] P. A. Beerel and K. M. Chugg, "A low latency SISO with application to broadband turbo decoding," *IEEE J. Select. Areas Commun.*, vol. 19, pp. 860–870, May 2001.

[6] P. Thiennviboon and K. M. Chugg, "A low-latency SISO via message passing on a binary tree," in *Proc. Allerton Conf. Commun., Control, Comp.*, pp. 959–960, October 2000.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, Mass.: MIT Press, 1990.

[8] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Computers*, vol. C-31, pp. 260–264, March 1982.

[9] G. Masera, G. Piccinini, M. R. Roch, and M. Zamboni, "VLSI architectures for turbo codes," *IEEE Trans. VLSI*, vol. 7, September 1999.

[10] M. Ferretti, R. Ozdag, and P. A. Beerel, "High performance asynchronous ASIC back-end design flow using single-track full-buffer standard cells," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 95–105, April 2004.

[11] M. Ferretti and P. A. Beerel, "Single-track asynchronous pipeline templates using 1-of-N encoding," in *Proc. Design, Automation and Test in Europe (DATE)*, pp. 1008–1015, March 2002.

[12] P. Golani and P. A. Beerel, "High-performance noise-robust standard-cell asynchronous library," in *Proc. International Symposium on VLSI Design*, pp. 256–261, March 2006.

[13] M. Ferretti, *Single-Track Asynchronous Pipeline Template*. PhD thesis, University of Southern California, August 2004.

[14] P. Beerel, A. Lines, M. Davies, and N.-H. Kim, "Slack matching asynchronous designs," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 1008–1015, March 2006.