Design of a High-Speed Asynchronous Turbo Decoder

Pankaj Golani, Georgios D. Dimou, Mallika Prakash, Peter A. Beerel
Department of Electrical Engineering-Systems
University of Southern California, Los Angeles, CA 90089
{pgolani,dimou,mallikap,pabeerel}@usc.edu

Abstract

This paper explores the advantages of high performance asynchronous circuits in a semi-custom standard cell environment for high-throughput turbo coding. Turbo codes are high-performance error correction codes used in applications where maximal information transfer is needed over a limited-bandwidth communication link in the presence of data corrupting noise. Specifically we designed an asynchronous high-speed Turbo decoder that can be potentially used for new wireless communications protocols with close to OC-12 throughputs. The design has been implemented using a new ***static*** *single-track-full-buffer (SSTFB) standard cell library in IBM 0.18µm technology that provides low latency, fast cycle-time, and more robustness to noise than previously studied single-track full-buffer technology (STFB). A high-speed synchronous counterpart using the same high-speed architecture is designed in the same technology for comparison. The results demonstrate that for a variety of network constraints, the asynchronous design provides advantages in throughput per area. Moreover, the asynchronous design can support very low-latency network constraints not achievable with the synchronous alternative.*

1. Introduction

Driven by overwhelming design-time constraints, standard-cell based synchronous design styles supported by mature CAD design tools and a largely automated flow dominate the ASSP and ASIC market places. As device feature sizes shrink and process variability increases, however, the reliance on a global clock becomes increasingly difficult, yielding far-from-optimal solutions. Because standard-cell designs use very conservative circuit families and are often over-designed to accommodate worst-case variations, the performance and power gap between full-custom and standard-cell designs continuously widens [20].

Recent research demonstrates that it is possible to narrow this gap using conventional standard-cell techniques with asynchronous cell libraries. For example, two prototype standard-cell libraries in TSMC 0.25µm technology for two different asynchronous templates have demonstrated high-performance [1][2]. One using a *Single Track Full Buffer (STFB)* library [1][2] successfully operate over a wide range of temperatures and voltages, with a measured frequency of over 1.2 GHz at a nominal 2.5 Volts [1]. This should be compared to the typical 300 MHz standard-cell synchronous designs achievable in the same process. In addition, a second generation single track family called *Static Single Track Full Buffer (SSTFB)* has been proposed which promises to have similar advantages while being more robust to higher process variability, crosstalk noise, and leakage currents [4][6].

This paper explores the library development and application of SSTFB to the asynchronous core of an ultra-high-throughput turbo decoder designed for close to OC-12 data rates [18] with different data block sizes.

We designed both synchronous and asynchronous ASIC cores to compare area, throughput, and power. The synchronous design requires substantially more parallelism to match the throughput of the asynchronous design. Consequently, our post-layout results on a significant portion of the design, including the critical path, indicate that we achieve up to a 2X improvement in throughput per area for small to medium block sizes and can support smaller block sizes at higher throughputs than achievable by the synchronous counterpart.

The remainder of this paper is organized as follows. Section 2 will cover necessary circuit and an introduction to turbo coding, with emphasis on the implementation challenges of a high-speed turbo decoder. Section 3 will introduce an efficient high-speed turbo decoding tree-SISO architecture and our synchronous baseline implementation. Section 4 will then describe our asynchronous implementation,

covering both the IBM 0.18μm SSTFB library development as well as P&R design flow. Section 5 will then describe the design verification and post-layout performance of both designs. Section 6 details the comparison between synchronous and asynchronous cores. Finally, Section 7 concludes and outlines areas of future work.

2. Background

This section begins by describing the STFB and SSTFB templates, followed by necessary background on Turbo decoding.

2.1. STFB Template

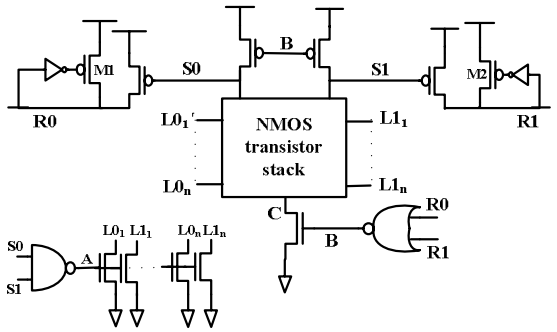

Figure 1. Typical STFB transistor-level schematic

Figure 1 shows a typical STFB cell's transistor-level diagram of a n-bit input 1-bit logic function. When there is no token in the dual-rail output channel (R0/R1), the 2-input NOR acts as *right completion detection (RCD)* block and asserts the "B" signal, enabling the processing of next set of input token. In particular, when the next set of tokens arrive at the *n* dual-rail input channels ($L0_i/L1_i$) the logic function is evaluated and one of the state signal "S0" or "S1" is lowered, dependent upon if the logic function evaluates to 0 or 1. Simultaneously, the state signal lowering causes the 2-input NAND gate which acts as a *state completion detection block (SCD)* to assert "A", resetting the tokens from the left channels by driving all inputs low, overpowering the corresponding staticizers (M1, M2). The presence of the output token on the right channel resets the "B" signal which activates the two PMOS transistors at the top of the N-stack, restoring "S0/S1", and deactivating the NMOS transistor at the bottom of the N-stack, thus disabling the stage from firing while the output channel is busy. The cycle time of the STFB template is 6 transitions with the forward latency of 2 transitions.

Notice that the output and input channels can have different tokens at the same time, highlighting why this template is a *full buffer* [1][2]. Notice also that the NMOS transistor stack (N-stack) is designed to be semi-weak-conditioned in that it will not evaluate until all expected input tokens arrive. This combination of functionality and input completion detection removes the need of using a *left environment completion detection block (LCD)*, reducing the template complexity, size and cycle-time. Alternatives such as the PCHB quasi-delay-insensitive template separate the N-stack and LCD functionality, but are substantially larger and slower [2].

2.2. Static STFB Template

The basic concern of the STFB circuit family is that the communication wires can be tri-stated for some period of time with only a small staticizer fighting leakage and crosstalk noise. While effective for 250nm, the noise margin for this technology may be too low in deeper submicron processes. In particular, a cross-coupling noise event on a long tri-stated wire can either create a new token or remove a token from the system causing system failure (often in the form of a deadlock). Moreover, in smaller geometries leakage currents may become so high that the staticizers would need to be made stronger to the point that they cannot be easily over-powered.

For this reason, Ferretti et al. [6] proposed a new STFB family called static STFB (SSTFB) in which the channel wires are always actively driven with modified driver circuits. The functionality of a SSTFB cell is the same as STFB cell with a noticeable difference that after the sender drives the line high, the receiver is responsible for actively keeping the line high until it wants to drive it low as shown in Figure 2. Similarly, after the receiver drives the line low, the sender is responsible for actively keeping the line low until it wants to drive it high. The line is always statically driven and no fight with staticizers exists. This means that the keeper circuitry can be sized to a suitable strength creating a tradeoff between performance/power/area and robustness to noise. The inverters in the keeper circuitry can be also be skewed such that they turn on early creating an overlap between the driving and hold logic (as suggested in [11]). This overlap avoids the channel wire being in a tri-state condition thus making the circuit family more robust to noise. The overlap also helps ensure that the channel wires are always driven close to the power supplies further increasing noise margins [4]. In this way, the lines are never tri-stated and are statically driven; explaining whey the circuit family is called *static* STFB. Figure 3 shows a transistor level diagram of a SSTFB Buffer.

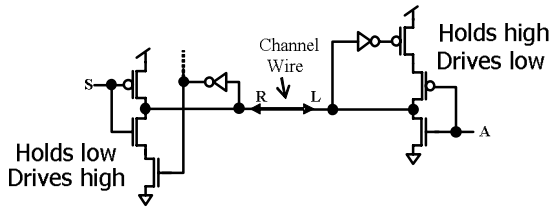

Figure 2. SSTFB driver circuitry

The SSTFB template is very flexible and can be expanded to implement different functionalities and non linear pipeline stages, including a merge, fork, and full-adder. We refer the reader to [4] for more details.

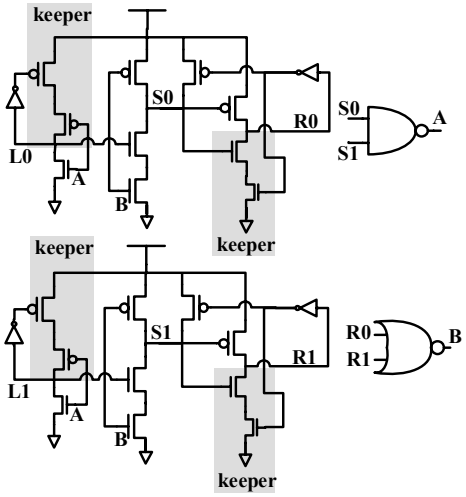

Figure 3. Schematic of SSTFB BUFFER

2.3. Turbo decoding

Turbo decoding is becoming a very popular solution for error correction especially in wireless applications [13]. Turbo coding started with the introduction of Parallel Concatenated Convolutional Codes (PCCC) that were proven to achieve performance that is very close to the theoretical coding bound defined by Shannon's Capacity. Since then several variations have been introduced, such as Serially Concatenated Convolutional Codes (SCCC) and Low Density Parity Check Codes (LDPCC). The same Turbo-Like decoding schedule could be used to process all of the above. In the case of LDPCC this is true when the code belongs to a class of LDPCC that can be described as a Generalized Repeat-Accumulate code (GRA) [21]. All these codes achieve Turbo-Like performance, but vary slightly in terms of performance and computational complexity and the selection for a particular design is made based on the operational conditions of the final system.

In our design we have chosen to implement a SCCC, which is known for its very low error floor capabilities, but the design proposed could be easily modified to decode any of the other types of codes listed above.

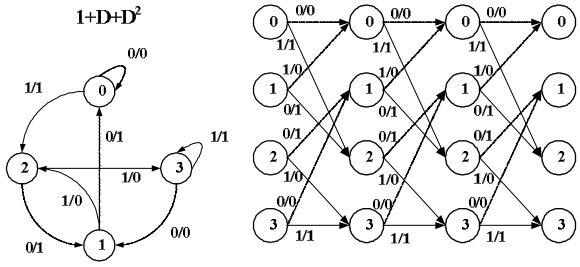

Figure 4. An example of a 4-state FSM encoder and the corresponding trellis used for decoding

2.3.1. Theoretical background. The basic Turbo-Like encoder structure involves an interleaver and a set of simple error correction/detection codes (most commonly convolutional codes). The structure on the decoder side looks very similar to that of the encoder, with two key differences and is illustrated in Figure 5. First every code in the encoder is replaced by a Soft-In-Soft-Out (SISO) module. Second the data flow on the decoder is bi-directional and iterative, and an interleaver/de-interleaver module in the decoder replaces the interleaver on the encoder side. In our implementation one SISO is used that can perform both operations to reduce design complexity.

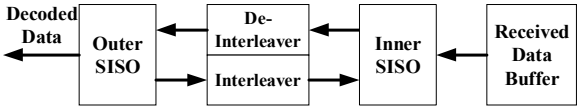

Figure 5. The decoder structure where each SISO is used to decode one CC.

During each SISO operation the data block is processed along a trellis that represents the state of the encoder during the transmission process, as illustrated in Figure 4. The state number is indicated inside each state, while each branch is characterized by the values of the inputs and outputs of the state machine (shown as x/y on each branch in Figure 4). The length of the trellis matches the number of bits in the data block. Each branch in the trellis has a branch metric associated with it (not shown in Figure 6), updated each iteration, which corresponds to a notion of the relative probability assuming that branch took place in the encoder. The SISO module is responsible for updating the probability that at time k the value b was encoded by finding the

shortest path through the entire trellis that has value *b* at time *k*.

To do this, for each trellis transition the decoder computes the *forward state metrics* which represent the shortest path from the beginning of the trellis up to that point and the *backward state metrics* which represent the shortest path information from the end of the trellis up to that transition. This is shown in Figure 6, where the shortest path is shown by the bold lines in the trellis and can be derived by the values of the state metrics. The decoder then can find the shortest path where $b_k=1$ and the shortest path on which $b_k=0$. Using this information it can produce the new probability for this bit being a 1 relative to being a 0 based on the information available across all the branches of the trellis.

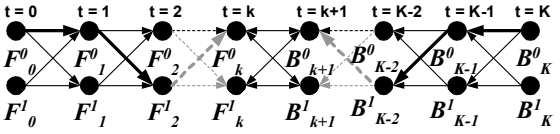

Figure 6. An example of a 2-state trellis and the associated metrics during the decoding process.

In particular, we have chosen to use the Min-Sum algorithm for the SISO operation. The Soft Input to the SISO is defined for each bit at time *k* as the negative log-likelihood ratio of the probability of a 1 being transmitted over that of a 0:

$$SI_{b_k} = \left(-\log\left(P\{b_k = 1\}\right)\right) - \left(-\log\left(P\{b_k = 0\}\right)\right). \quad (1)$$

The branch metrics between state *i* and state *j* (if such exists) is the joint probability for the particular trellis branch given all SI, or:

$$BM_k^{i,j} = \sum_{b_k=1} SI_{b_k}. \quad (2)$$

The forward and backward state metrics for time instance *k* are then defined recursively as follows:

$$F_k^j = \min_{i,j}\left(F_{k-1}^i + BM_{k-1}^{i,j}\right) \quad (3)$$

$$B_k^i = \min_{i,j}\left(B_{k+1}^j + BM_k^{i,j}\right) \quad (4)$$

where, F_k^i is the value of the forward state metric for state *i* at time *k*. The index *j* only takes the values for which the transition from state *j* to state *i* is valid. The state metric calculations are also referred to as the Add-Compare-Select operation or ACS and constitute the majority of the processing taking place in the SISO. An example 4-bit ACS is shown in Figure 7.

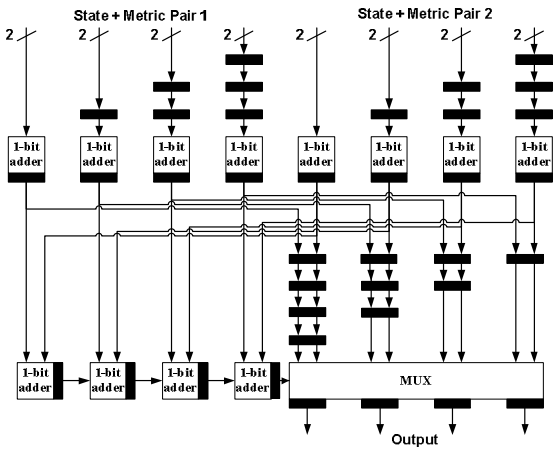

Figure 7. A bit-pipelined 4-bit ACS operator. The black rectangles indicate pipeline boundaries

The SISO outputs are called the Soft Outputs (SO) for all input and output bits of the encoder FSM. To prevent the values that are sent between SISO modules in the decoder from growing indefinitely, instead of the actual value the SISO outputs the differential (called the extrinsic SO) between the actual value calculated (also called intrinsic SO) and the original SI inputted. The final quantities are also saturated to a fixed bitwidth, to reduce the complexity of the SISO modules. So the Soft Output for bit *b* at time *k* is defined as:

$$SO_{\text{int}r b_k} = \min_{\substack{i,j \\ b_k=1}}\left(F_k^i + BM_k^{i,j} + B_{k+1}^j\right) - \min_{\substack{i,j \\ b_k=0}}\left(F_k^i + BM_k^{i,j} + B_{k+1}^j\right) \quad (5)$$

$$SO_{extr b_k} = SO_{\text{int}r b_k} - SI_{b_k}. \quad (6)$$

The decoding process from a top-level standpoint starts by the received signal being translated into metrics that represent the probabilities for each of the received bits. Then a SISO module uses the received sequence as inputs and produces soft outputs. The soft outputs are then interleaved (or de-interleaved depending on the code) and passed onto the SISO modeling the next convolutional encoder in the transmit sequence as Soft Inputs. During the decoding process the SISOs that correspond to all the codes exchange Soft Input data both in the encoder sequence and in the reverse direction. Each SISO is fired several times and the process iterates until the metrics stop improving, or the maximum number of iterations is reached. For our comparisons we use 6 iterations which achieves most of the coding gain without being too computationally intensive.

2.3.2. High-speed implementation challenges. The immediate effect of this iterative approach is that in order to achieve a certain decoded data rate the SISO

has to run several times faster internally in order to keep up with the data. For example a system using a code with two convolutional codes and decoding using 5 iterations would have to run roughly 10 times faster internally than the target throughput. The calculation is iterative and cannot be speed up easily. Several tiling approaches have been developed that break the block into sub-blocks that can be processed in parallel, but normally the logic itself cannot be sped up further than the state metric calculation process (F_k^i and B_k^i) due to the data dependency in that calculation. The adopted Tree-SISO architecture addresses this problem and will be described in detail in a later section.

Even if the state update loop is broken (as in Massera's and Tree-SISO architectures [14][7][15]) there are other practical problems that hinder the throughput. A popular approach to increase throughput is to use many units in parallel. Although this generally works, it has implementation problems that make the design of very high-speed Turbo decoders extremely challenging.

The first problem is related to memory access. As mentioned above the data after being processed has to be interleaved between SISO modules. In order to get good coding performance, the interleaver must use a permutation that is ideally random. Therefore, with a high degree of parallelism many bits per cycle have to first be stored into a RAM structure and then retrieved in random order. The usual approach is to have multiple banks of RAM that each receives 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 are placed on the interleaver to ensure it is *clash-free*. That 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 very sub-optimal interleaver performance, due to lack of randomness. From a hardware standpoint it also requires the instantiation of many more RAM cores that are extremely small and shallow, that consequently require a lot more area and power than fewer larger and narrower RAM instances.

The second problem is also a side effect of the interleaver presence. The entire block of data has to be written into the interleaver before the data can read to start the next SISO process. This is due to the interleaver's random permutation, which implies that the first bits of data that have 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.

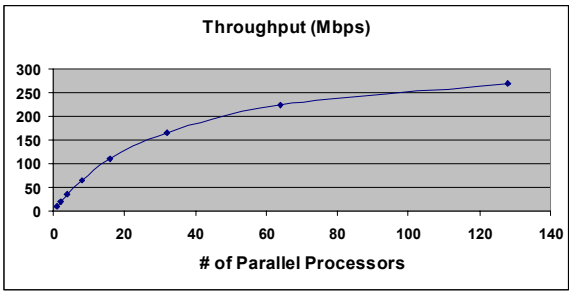

Figure 8. Throughput vs. # of processors

Performance degradation is more pronounced in cases with small data block sizes where the pipeline latency is comparable or in extreme situations larger than the actual processing time. This does not only occur once, but occurs every SISO operation. Figure 8 illustrates this point by showing the throughput that a decoder can achieve as a function of processors to perform one SISO operation. The graph 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.

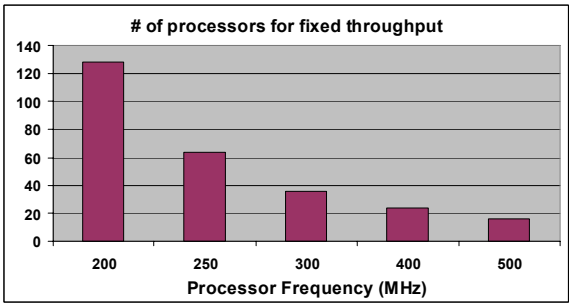

Figure 9. # of processors vs. processor frequency

Figure 9 illustrates this point from a different perspective by showing the number of processors that can be used to achieve a throughput of 540 Mbps with varying processor frequency (assuming 5 iterations and 2 Kbit data block size). It is easy to see that increasing the processor frequency can achieve much larger reduction in the number of processors than the expected linear function.

3. Synchronous high speed Turbo

In order to evaluate the performance of our design and demonstrate the capabilities that our asynchronous technology has to offer, we designed a synchronous core as well to be able to compare area, performance and power between the two designs. We chose to

design the same unit using both technologies, using our SSTFB library for the one and the Artisan library for the other.

3.1. Tree SISO

Designing a very fast Turbo decoder structure has many challenges as mentioned above. Our goal based on our analysis was to design the fastest SISO unit possible so that we can keep the degree of parallelism required to a minimum. For this reason we chose to use a Tree SISO structure [7][15] which removes the recursive nature in the data path and thus enables fine grain pipelining.

In order to illustrate this implementation, we must define one additional operation that merges adjacent transitions of the trellis into larger trellis *sections* that correspond to more than one time index. In this manner, several state metrics can be computed simultaneously when the appropriate state metric becomes available. The new operation, called the fusion operation, is defined as:

$$BM_{k,l}^{i,j} = \min_{p}(BM_{k,m}^{i,p} + BM_{m,l}^{p,j}), \forall m \in (k,l) \quad (7)$$

The min operation is defined over all valid combinations of branch metric pairs of starting and ending states. The new branch metrics correspond to the shortest possible path derived from the merged trellis sections between any pair of starting and ending states. The structure used to implement the SISO is borrowed from prefix adder structures, but with the addition of a suffix path that is used to perform the operation backwards for the Backward State Metric calculation.

3.2. The code

We chose a typical SCCC turbo code structure with two 2-state convolutional codes to reduce the size of the decoder circuit. The data is first encoded using a rate ½ non-recursive convolutional code with polynomials [1+D, 1+D] and the results are interleaved. Next a rate 1 recursive code with a polynomial [1/(1+D)] is used to re-encode the interleaved data before transmission. Overall this yields a rate ½ code. Puncturing could be used to achieve higher code rates and increase flexibility, with minor modifications to the design, but this was not done at this stage for simplicity.

For a block size of K bits the first code has a trellis length of K and the second one of 2K. Therefore the throughput equation is as follows:

$$T = \frac{f * K}{I\left(2p + \frac{3K}{8M}\right)} \quad (8)$$

where K is the block size in bits, 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 half-iteration. The execution schedule for every iteration is shown in Figure 10. It should also be noted that only the last iteration produces decoded data, which is why the throughput is inversely proportional to the number of iterations.

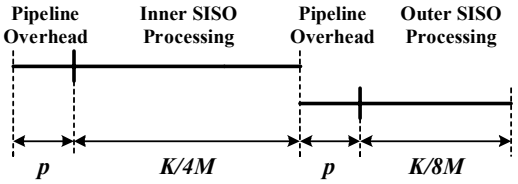

Figure 10. Execution schedule of every decoder iteration

In the case of our asynchronous design M is 1 since we are going to use a single 8-bit wide SISO processor to achieve the desired throughput. In the case of the synchronous design, M will have to be higher since the synchronous design is much slower than our asynchronous one and multiple processors of size 8 would be required to achieve the same throughput. We chose 8 since it is the minimum size Tree-SISO for a 2-state code, and it should be noted that due to the structure of the Tree-SISO a 16-bit wide processor is more complex than two 8-bit wide ones.

3.3. P&R results

After the schematic design was finished it was exported to a Verilog netlist and imported into SOC Encounter for P&R. The libraries for the IBM 0.18µm technology were used for characterization, and timing constraints were written to define the target frequency. The core was placed as a standalone module without IO pads. The design was placed using timing-driven placement and was then routed using timing-driven routing. After routing was done the clock tree was synthesized and the design was taken through further processing to fix hold time violations and then the final timing analysis was performed. The clock frequency that was achieved was 475MHz for the entire 8-bit wide core. The area of the core was 2.46mm^2. We assume

that for higher degrees of parallelism multiple copies of this core could be routed separately and that no performance degradation would be induced due to the added circuitry. We also assumed that the clock circuit would be mostly unaffected and that it would just be multiplied in size just like the rest of the circuitry.

As a point of comparison, the previously published fastest turbo design achieves approximately 1Gbps at 6 iterations in a 0.18µm process, using 32mm^2 area and a single-buffered input memory [19]. The code in [19] is a PCCC which requires the processing of 2K trellis steps/iteration, so that structure would decode approximately 667Mbps for an SCCC code like the one we chose, which requires processing of 3K trellis steps/iteration. Our synchronous design has similar throughput (653Mbps) for M=6 and using 14.76mm^2 of core area and 2.17mm^2 of memory area, which indicates that our synchronous design is comparable with state-of-the-art decoders found in the literature. Since our synchronous and asynchronous cores would use the same memory area, our comparison in Section 6 only considers the core area.

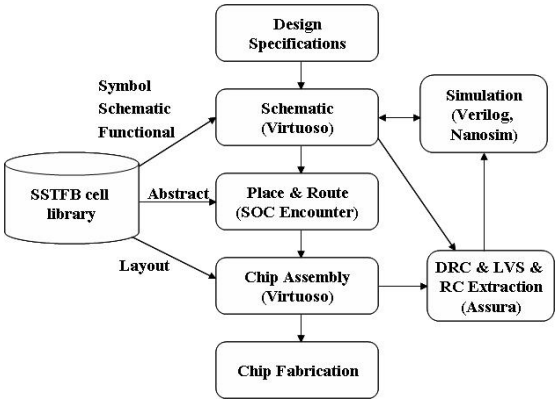

Figure 11. Asynchronous ASIC design flow

4. Asynchronous Turbo

This section covers the asynchronous Turbo design, including a brief description of the overall design flow, the library development, gate-level design, and physical design.

4.1. Asynchronous ASIC Design Flow

For each SSTFB cell needed we created four library views: functional views contains the behavioral description of the cell in Verilog HDL, schematic views contains the transistor level implementation of the cell, layout view containing detailed GDSII data, an abstract view to support placement and routing in LEF format, and finally its symbol. Using this library, a largely conventional standard-cell ASIC back-end design flow using conventional place and route tools can be used to create the layout, as illustrated in Figure 11.

Note that we currently use Nanosim to perform analog transistor-level simulations to verify both correctness and measure performance. Characterizing the library in LIBERTY format which enables faster back-annotated gate-level Verilog simulation, as suggested by [3] is an area of future work.

4.2. Library design

The computation in the chip consists of additions, comparisons, and selections. Consequently, the SSTFB library needed only 14 cells along with a variety of sub-cells used to simplify the development of new library cells. Extensive spice simulations were done on the schematic to verify that all the timing assumptions and specifications were achieved. DRC and LVS checks using DIVA and ASSURA verification suites were performed on the layout views to verify their correctness.

Our SSTFB library currently includes only a single size for each cell. With this circuit technology there are five ways to combat noise and process variations: 1) increase the size of the keeper transistors 2) increase the minimum separation between wires in the place and route flow 3) decrease the maximum allowable length of any route 4) shield long communication wires and 5) skew the hold inverters (INV_HI and INV_LO) shown in Figure 3 to create more time overlap between the driving and hold transistors. During library development, we focused on techniques 1) and 5), enabling the use of minimum separation between wires with no shielding for a maximum wire length of 400 µm.

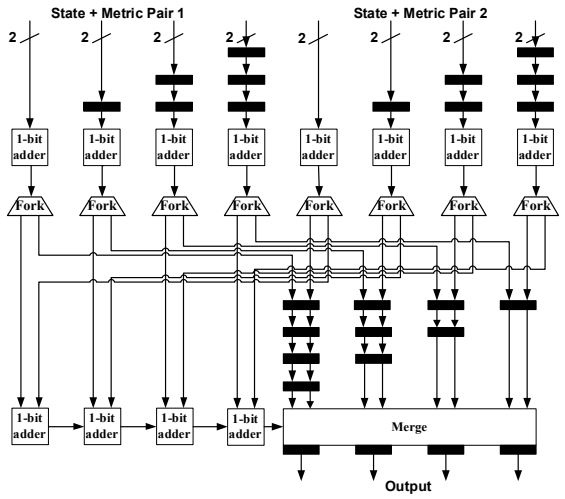

Figure 12. An example of 4-bit ACS asynchronous block. The black rectangles are slack matching buffers and the connections are dual rail static single-track channels

4.3. Gate-level design of asynchronous Turbo

The implementation of the asynchronous Turbo core was done using the same approach as that of the synchronous version. Schematic entry was used for all the modules, starting with the smaller ones and moving up the hierarchy to the top level, shown in Figure 12. The hierarchical structure for the two designs was kept largely identical. The differences in the asynchronous implementation are listed below:

4.3.1. Use of dedicated Fork cells. The SSTFB library implements point-to-point communication between cells and a particular signal cannot fork and go to two destinations. To solve this problem we created and used dedicated FORK cells to support fanout.

4.3.2. Slack matching. Special design considerations had to be taken to balance the pipelines in the asynchronous design to avoid pipeline stalling or starvation. Another aspect of slack matching, which we have not yet taken full advantage of, is the fact that the buffer cells are generally much faster than other logic (e.g., full-adders) cells. This allows us to use shorter buffer chains for delaying signals, because these chains can absorb stalled tokens while not degrading overall performance [5][8].

4.3.3. Incorporating FORKs inside cells. The use of dedicated FORK cells creates additional pipeline stages, which can increase the number of slack matching buffers needed.. In order to mitigate this problem we decided to incorporate the FORK inside some of the logic cells. For example consider a full adder cell with only sum output (FA_S). In order to distribute the sum output to two cells, conventionally we used a dedicated FORK cell after at FA_S output. Instead, we created a special cell, full adder with two sum outputs (FA_2S) by copying the sum output token internally, thus having two sum output channels. In addition to reducing the need for slack cells, the new full adder cell (FA_2S) is 45% less area than the combination of FA_S and an external FORK. The same concept can be applied to the other logic cells also.

4.3.4. SLACK2 and SLACK4 cells. We observed that that most of our initial design was dominated by SSTFB Buffer cells for slack matching. Most of the times these buffer cells existed in linear chains i.e. one buffer cell driving another buffer cell as shown in **Figure 12**. In order to save area, we decided to create two types of special cells SLACK2 and SLACK4 cells which are functionally equivalent to 2 SSTFB Buffers and 4 SSTFB Buffers in series, respectively. The transistor level diagram of a SLACK2 cell is shown in Figure 13.

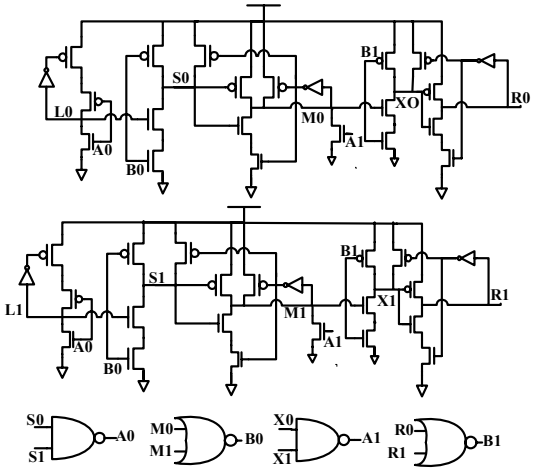

Figure 13. SLACK2 transistor level diagram

In SLACK2 the internal channel (M0-M1) is dynamically driven by a POUT driver quite similar to the output driver used in STFB cells [6]. Because they are short and internal to the cell, they have minimal potential crosstalk noise due to coupling capacitance and thus are relatively safe. Moreover, because the internal wires are short, we used minimum transistor sizes, saving both area and power. In particular, SLACK2 and SLACK4 cells have 17% and 30% less layout area compared to 2 and 4 Buffers, respectively. Moreover, pre-layout transistor level simulations indicate that SLACK2 cells and SLACK4 consume 10% and 19% less power compared to 2 Buffers and 4 Buffers, respectively.

4.4. P&R and design results

The design was place and routed using Cadence SOC Encounter in a similar fashion as the synchronous counterpart. Congestion based placement was performed and the routing was performed on the design using Nanoroute. The final core has 70% utilization and the final area consumed by the logic is 6.92mm^2. This version of the core is preliminary because there is still some slack matching needed to achieve our target throughput and some library development needed to remove all DRC violations. However, the core is fully routed using 6 metal layers showing that the design is routable.

5. Verification and Simulation Results

This section covers the design verification of the asynchronous turbo and its simulation results.

5.1. Design Verification

The schematic of the design was converted into a Verilog netlist and the simulation was performed using our Verilog models for the SSTFB cells in NC-Sim. Due to the size of the design and the instantiation of multiple identical components, the verification was performed in a bottom-up fashion. We started with simple cells such as adders and moved up the hierarchy to ACS units, state update nodes, branch metric calculation units, the completion logic, and finally the top-level module. For each module a set of vectors were generated that would test all corner cases of its behavior and the results were verified and cross referenced to the synchronous counterpart.

5.2. Post-Layout ECO and simulation results

To estimate the performance of the chip we simulated the 55K-transistor module that implements Equations 5 and 6. This module contains an 8-bit ripple carry chain of full adders which includes the critical cycle of the design. This module represents around 1/30th of the complete design but is the most computationally intensive module in the SISO.

To improve the performance of the design we added SSTFB Buffers on long wires using the ECO flow in SOC Encounter. The addition of the buffers increased the modules throughput by 26% and increased the utilization factor from 70% to 76% but otherwise did not impact area. The final layout was extracted using Assura RC in *coupled* mode and the circuit was simulated using Nanosim, yielding a throughput of 1.15GHz.

6. Preliminary Comparisons

The previous two sections described the synchronous baseline and asynchronous designs in detail and showed their post P&R results. This section compares the performance, area, throughput/area, energy and power/area results for the two designs. Because the SSTFB standard cell library has only one size per cell, the number we present are conservative and may improve if we developed and used more sizes per cell.

Block Size (bits)	Async T (Mbps)	Sync T (Mbps)	M	Sync area (mm^2)	T/area ratio
512	383	-	-	-	-
768	418	415	11	27.06	3.91
1024	438	440	6	14.76	2.13
2048	471	519	4	9.84	1.28
4096	490	513	3	7.38	1.03

Table 1. Throughput per area comparison

6.1. Performance estimates and comparison

The frequency of the post P&R synchronous core is 475 MHz. From the post layout simulation explained in Section 5.2 we expect the asynchronous core frequency to be approximately 1.15GHz. Thus, we expect the asynchronous core to run 2.4 times faster then its synchronous counterpart.

6.2. Area comparison

The logic area of the synchronous and asynchronous cores are 2.46mm^2 and 6.92mm^2, respectively. Both asynchronous and synchronous cores implement the exact same function with the same degree of parallelism. However since the asynchronous core is 2.4 times faster and has smaller pipeline latency than the synchronous core, we must instantiate the synchronous core many times in order to match the throughput, as described in Section 2.3.2. Substituting the numbers in Equation (8), we compute that for equivalent throughput with 6 iterations and pipeline latencies of 60 cycles for the synchronous design and 32 equivalent cycles for the asynchronous one, the

number of required synchronous cores varies from 11 for a throughput of 418Mbps for block size of 768bits to 3 for a throughput of 490Mbps for block size of 4Kbits.

6.3. Throughput/area comparison

Throughput/area is another important metric for the comparison, since it indicates the performance in relation to the area used to achieve it. We chose to use throughput/area instead of just area for comparable throughputs as a metric for our comparisons. This is because the throughput that is achievable by each design is not exactly equal, so the ratio comparison provides a normalized metric that is fairer. From Table 1 we can see for example that for a block size of 1Kbits which is a very common block size used in wireless applications we obtain a throughput per area advantage of 2.13. The advantages are even bigger 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 degree of parallelism M. As the block size increases, latency becomes less of a critical factor and the two designs become more comparable.

Block Size (bits)	Energy per block (sync)	Energy per block (async)	Ratio
768	3.5E-05	2.84E-05	0.81
1024	2.40E-05	3.63E-05	1.5
2048	2.714E-05	6.73E-05	2.5
4096	4.11E-05	12.9E-05	3.1

Table 2. Energy per block comparison

6.4. Energy Comparisons

From the post-layout spice simulation the power consumed of the selected module is 0.53W. If we extrapolate the number we expect the power of the complete Tree SISO to be approximately 15.5W. The power for a single synchronous core (M=1) is 1.72W. From Table 2 we can see that for smaller block sizes we are more energy efficient than the synchronous design, but for larger block sizes the synchronous design more efficient. It should be stated that the power calculation for both designs was performed for the worst case scenario, namely with the units processing data at the maximum rate. Even though the calculation is based on peak power, we believe that the numbers might be conservative, but the ratio should be representative of the relative power consumption of the two designs. We have also not included leakage power comparisons in the calculations, but given that the asynchronous design requires less area for the same throughput and leakage power is proportional to the total area, we expect to have an advantage in respect to that aspect as well.

6.5. Design Time Comparisons

Another important metric to be compared is design time. The design of synchronous turbo decoder was done using front end views of the standard cell library provided by Artisan. The main steps involved were design conceptualization, schematic entry of the deisgn, verification of the design, Placement and Routing and finally timing enclosure of the design using PrimeTime. This design effort took approximately 3-4 graduate-student months. The design of the asynchronous counterpart involved almost the same steps with two noticeable difference. First, we had to create our own standard cell library. Second, due to the lack of static timing analysis tools, timing verification and closure was performed using time-consuming Nanosim simulations. The creation of standard cell library took 5 graduate-student months and the timing closure and verification took approximately 10 graduate-student months. Also we want to make clear that the asynchronous design is not yet complete as there remains some DRC errors to be removed in the final layout.

7. Summary and conclusions

Our results demonstrate that SSTFB asynchronous turbo decoder is beneficial for small to medium block sizes. Preliminary comparisons show that the asynchronous turbo decoder can offer more than 2X advantage in throughput per area for block sizes of 1K bits 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 size must be small.

More generally, this design experiment demonstrates the potential benefits of high-performance low-latency asynchronous libraries and standard-cell design flows for processing intensive applications. However, 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. 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 is likely.

Finally, in order to obtain our target performance we must perform an ECO slack-matching flow on the entire design in order to mitigate the performance degradation of long wires. This process is currently manual, however we hope to automate this process with a CAD tool that uses the slack matching technique described in [5] and the back-annotation flow described in [3].

Acknowledgements

We would like to thank the anonymous reviewers for helpful comments on earlier drafts of this work. This work was partially supported by NSF ITR Award No. CCR 0086036.

References

[1] M. Ferretti and P. A. Beerel. High Performance Asynchronous Design Using Single-Track Full-Buffer Standard Cells, *IEEE Journal of Solid-State Circuits*, Vol. 41, No. 6, pp. 1444-1454, June 2006.

[2] M. Ferretti and P. A. Beerel. Single-Track Asynchronous Pipeline Templates using 1-of-N Encoding, *DATE'02*, Mar. 2002.

[3] P. Golani and P. A. Beerel. Back-Annotation in High-Speed Asynchronous Design, *Journal of Low power Electronics*, Vol. 2, pp. 37-44, 2006.

[4] P. Golani and P. A. Beerel. High Speed Noise Robust Asynchronous Circuits, *ISVLSI'06*, March, 2006. Pages: 173-178

[5] P. A Beerel, A. Lines, M. Davies, N.-H. Kim. Slack matching asynchronous designs. *ASYNC'06*, March 2006.

[6] M. Ferretti. *Single-track Asynchronous Pipeline Template*, Ph.D. Thesis, University of Southern California, Aug., 2004.

[7] P. A. Beerel and K. M. Chugg. A Low Latency SISO with Application to Broadband Turbo Decoding, *IEEE Journal on Selected Areas in Communications*, Vol. 19 Issue 5, May, 2001.

[8] R. Manohar and A. J. Martin. Slack Elasticity in Concurrent Computing. *Proceedings of the Fourth International Conference on the Mathematics of Program Construction, Lecture Notes in Computer Science 1422*, pp. 272-285, Springer-Verlag 1998.

[9] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings and T. K. Lee. The Design of an Asynchronous MIPS R3000 Microprocessor. *ARVLSI'97*, 1997.

[10] M. Nyström, E. Ou, A. J. Martin. An Eight-bit Divider Implemented in Asynchronous Pulse Logic. *ASYNC'04*, April 2004.

[11] K. van Berkel, and A. Bink. Single-Track Handshake Signaling with Application to Micropipelines and Handshake Circuits, *ASYNC'06*, pp. 122–133, 1996.

[12] I. E. Sutherland and S. Fairbanks. GasP: a Minimal FIFO Control. *ASYNC'01*, pp. 46–53, March 2001.

[13] K. M. Chugg, A. Anastasopoulos, and X. Chen. *Iterative Detection: Adaptivity, Complexity Reduction, and Applications*. Kluwer Academic Press, 2000.

[14] G. Masera, G. Piccinini, M. Ruo Roch, and M. Zamboni. VLSI Architectures for Turbo Codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 7, No. 3, September 1999

[15] P. Thiennviboon and K. M. Chugg. A Low-Latency SISO via Message Passing on a Binary Tree, *Allerton Conf.*, Urbana, IL, Oct. 2000.

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

[17] R. Dobkin and M. Peleg. Parallel Interleaver Design and VLSI Architecture for Low Latency Map Turbo Decoders, *IEEE Transactions on VLSI Systems*, April 2005.

[18] IEEE 802.11 Working Groups Web Page. http://grouper.ieee.org/groups/802/11/.

[19] B. Bougard, A. Giulietti, L.Van der Perre, F. Catthoor. A Class of Power Efficient VLSI Architectures for High Speed Turbo-Decoding, *IEEE Global Telecommunications Conference*, Vol. 1, pp. 549 - 553, 2002

[20] D. Chinery and K. Keutzer, *Closing the Gap between ASIC & Custom. Tools and Techniques for High Performance ASIC design*. Kluwer Academic Publishers, ISBN 1-4020-7113-2.

[21] K. M. Chugg, P. Thiennviboon, G.D. Dimou, P. Gray, and J. Melzer, A new class of turbo-like codes with universally good performance and high-speed decoding, in *IEEE Military Communications Conference*, Vol. 5, pp. 3117 – 3126, Oct. 2005