Architectural Design of BWA for finding First W max/min values using Sorting Algorithms

A.Venkata Ratnam[1] **M.Narendra Babu**[2]
samu.syamala@gmail.com[1] muthakaninarendrababu@gmail.com[2]
[1]PG Scholar, Dept of ECE, Nalanda Institute of Engineering & Technology, Kantepudi, Sattenapalli, Guntur, A.P, India.
[2]Assistant Professor, Dept of ECE, Nalanda Institute Of Engineering & Technology, Kantepudi, Sattenapalli, Guntur, A.P, India

Abstract: Applications like non-binary LDPC decoders k-best MIMO detectors & turbo product codes requires VLSI architectures for finding first w(w>2) maxima/minima for implementation. A parallel radix-sort based VLSI architecture is proposed for finding the first w maxima/minima. A B.W.A architecture is explained, which uses a small logic circuits. This B.W.A architecture takes the input by analyzing from M.S.B to L.S.B. A high level of scalability is a key feature in the B.W.A architecture which enables by taking large range of both w & size of input data in a large spectrum of applications. When comparing to the other solutions in literature the practical results shows that the B.W.A architecture achieves less area, while implementing on a hi-speed CMOS standard cell technology. Among all cases which are considered shows that B.W.A gives lowest area-delay product.

Keywords— Decoding, field-programmable gate array (FPGA), forward error correction and low density parity check (LDPC).

I. Introduction

Low-density parity-check (LDPC) Sorting has attracted a great deal of attention over the past few decades of computer science research. It is easy to see why: sorting is a theoretically interesting problem with a great deal of practical significance. As many as a quarter of the world's computing cycles were once devoted to sorting. This is probably no longer the case, given the large number of microprocessors running dedicated control tasks. Nonetheless, sorting and other information-shuffling techniques are of great importance in the rapidly growing database industry. The sorting problem can be defined as the rearrangement of N input values so that they are in ascending order. This paper examines the complexity of the sorting problem, assuming it is to be solved on a VLSI chip. Much is already known about sorting on other types of computational structures, and much of this knowledge is applicable to VLSI sorting. However, VLSI is a novel computing medium in at least one respect: the size of a circuit is determined as much by its integrate wiring as by its gates themselves. This technological novelty makes it appropriate to -reevaluate sorting circuits and algorithms in the context of a "VLSI model of computation." Sorting is a deep-rooted problem in computer science and is a mean of operation in quite a lot of applications. Along with hardware implementation of sorting networks has been identified as well in the history. On the other side, VLSI architectures for partial sorting, which can also be derived from assortment networks (AN), are part of different algorithms in the communication field. Sorting is a well established problem in computer science and is a key operation in several applications.

Moreover, equipment usage of sorting systems has been tended to too previously. Then again, VLSI architectures for incomplete sorting, which can likewise be gotten from choice systems (SN) , are a piece of distinctive calculations in the correspondence field. Fractional sorting is utilized for the translating of turbo and parallel Low-DensityParity-Check (LDPC) codes, in for greatest probability deciphering of number juggling codes and in for K-best MIMO finders, non-twofold LDPC decoders and turbo item codes separately. Circuits for discovering the initial two base qualities are utilized as a part of parallel LDPC decoder architectures to actualize min-total close estimations and as of late they have likewise been proposed for the instance of non-twofold LDPC decoders However, not very many works, e.g. research the general issue of executing parallel architectures for discovering

the initial two most extreme/least values with an efficient methodology. So also, architectures for discovering the first W > 2 most extreme/least values in an arrangement of M components, with W ≤ M=2, are outlined in VLSI usage of i) K best MIMO identifiers ,ii) non-parallel LDPC decoders ,iii) turbo item codes . Tragically, to the best of our insight, no papers in the open writing present a general investigation for the case W > 2. Originating from the work depicted for sorting systems, a comparator-based SN is proposed. Not with standing, as contended in, different methodologies, for example, the one alluded to as radix sorting, can be utilized also. Radix sorting calculations depend on the bit-wise examination of the information to be sorted and can be reached out to determination and incomplete sorting issues. This paper proposes a parallel VLSI building design depending on the radix sorting methodology for discovering the first W > 2 greatest/least values in an arrangement of M qualities. To be specific, the proposed arrangement, alluded to as Bit-Wise-And (BWA) analyzing so as to build design, lives up to expectations the M hopefuls from the Most-Significant-Bit (MSB) to the Least-Significant-Bit (LSB).

II. Sorting Algorithm

Sorting algorithms are an important part of managing data. Most sorting algorithms work by comparing the data being sorted. In some cases, it may be desirable to sort a large chunk of data (for instance, a structure containing a name and address) based on only a portion of that data. The piece of data actually used to determine the sorted order is called the key. Sorting algorithms are usually judged by their efficiency. In this case, efficiency refers to the algorithmic efficiency as t he size of the input grows large and is generally based on the number of elements to sort. Most of the algorithms in use have an algorithmic efficiency of either O (n^2) or O(n*log(n)).

A. Criteria for Comparison

Many algorithms that have the same efficiency do not have the same speed on the same input. First, algorithms must be judged based on their average case, best case, and worst case efficiency. Some algorithms, such as quick sort, perform exceptionally well for some inputs, but horribly for others. Other algorithms, such as merge sort, are unaffected by the order of input data. A second factor is the "constant term". As big-O notation abstracts away many of the details of a process, it is quite useful for looking at the big picture. But one thing that gets dropped out is the constant in front of the expression: for instance(c*n) is just O(n). In the real world, the constant, c, will vary across different algorithms. A well-implemented quick sort should have a much smaller constant multiplier than heap sort. A second criterion for judging algorithms is their space requirement do they require scratch space or can the array be sorted in place (without additional memory beyond a few variables)? Some algorithms never require extra space, whereas some are most easily understood when implemented with extra space (heap sort, for instance, can be done in place, but conceptually it is much easier to think of a separate heap). Space requirements may even depend on the data structure used (merge sort on arrays versus merge sort on linked lists, for instance).

A third criterion is stability -- does the sort preserve the order of keys with equal values? Most simple sorts do just this, but some sorts, such as heap sort, do not. The following table compares sorting algorithms on the Some algorithms (selection, bubble) work by moving elements to their final position, one at a time. You sort an array of size N, put 1 item in place, and continue sorting an array of size N – 1

- Some algorithms (insertion, quick sort) put items into a temporary position, close(r) to their final position. You rescan, moving items closer to the final position with each iteration.
- One technique is to start with a ―sorted list‖ of one element, and merge unsorted items into it, one at a time.
- Complexity and running time
- Factors: algorithmic complexity, startup costs, additional space requirements, use of recursion (function calls are expensive and eat stack space), worst-case behavior, assumptions about input data, caching, and behavior on already-sorted or nearlysorted data
- O (N) time is possible if we make assumptions about the data and don't need to compare elements against each other (i.e., we know the data falls into a certain range or has some distribution).O(N) clearly is the minimum sorting time possible, since we must examine every element at least once

A. Bubble Sort [Best: O (n), Worst: O (N^2)]

Starting on the left, compare adjacent items and keep ―bubbling‖ the larger one to the right (it's in its final Place). Bubbles sort the remaining N -1 item.

- Though ―simple‖ I found bubble sort nontrivial. In general, sorts where you iterate backwards (decreasing some index) were counter-intuitive for me. With bubble-sort, either you bubble items ―forward‖ (left-to-right) and move the endpoint backwards (decreasing), or bubble items ―backward‖ (right-to-left) and increase the left endpoint. Either way, some index is decreasing.
- You also need to keep track of the next-to-last endpoint, so you don't swap with a non-exist ant item.

1) Advantage: Simplicity and ease-of-implementation
2) Disadvantage: Code inefficient

B. Selection Sort [Best/Worst: O(N^2)]

Scan all items and find the smallest. Swap it into position as the first item. Repeat the selection sort on the remaining N-1 items.

I found this the most intuitive and easiest to implement — you always iterate forward (i from 0 to N-1), and swap with the smallest element (always i).
1) Advantage: Simple and easy to implement
2) Disadvantage: Inefficient for large lists, so similar to the more efficient insertion sort, the insertion sort should be used in its place.

C. Insertion Sort [Best: O(N), Worst: O(N^2)]

Start with a sorted list of 1 element on the left, and N-1 unsorted items on the right. Take the first unsorted item (element #2) and insert it into the sorted list, moving elements as necessary.

We now have a sorted list of size 2, and N -2 unsorted elements. Repeat for all elements.

- Like bubble sort, I found this counter-intuitive because you step ―backwards‖
- This is a little like bubble sort for moving items, except when you encounter an item smaller than you, you stop. If the data is reverse-sorted, each item must travel to the head of the list, and this becomes bubble-sort.
- There are various ways to move the item leftwards — you can do a swap on each iteration, or copy each item over its neighbor

1) Advantage: Relative simple and easy to implement. Twice faster than bubble sort.
2) Disadvantage: Inefficient for large lists.

D.Quicksort [Best: O(N lg N), Avg: O(N lg N), Worst:O(N^2)]

There are many versions of Quicksort,which is one of the most popular sorting methods due to its speed (O (N lgN) average, but O (N^2) worst case). Here's a few:

1) Using external memory:

- Pick a ―pivot‖ item
- Partition the other items by adding them to a ―less than pivot‖ sublist, or ―greater than pivot‖ sublist
- The pivot goes between the two lists
- Repeat the quicksort on the sublists, until you get to a sublist of size 1 (which is sorted).
- Combine the lists — the entire list will be sorted.

III.Proposed Architecture

A. Initial BWA architecture description

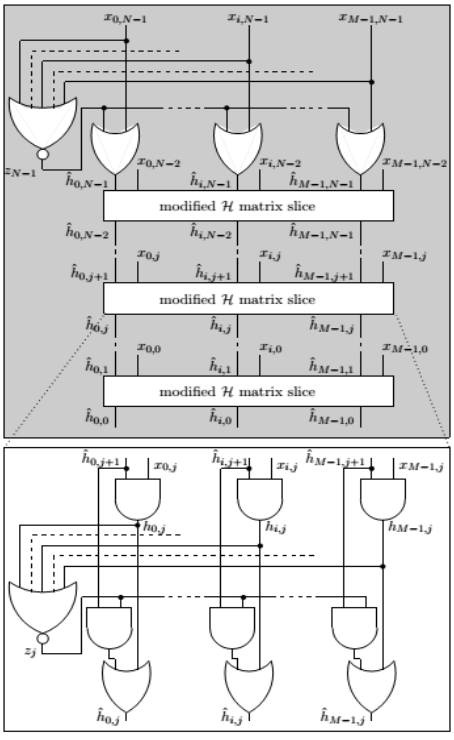


Figure 1: BWA architecture: modified $\mathcal{H}$ matrix.

Radix sorting relies on the analysis of X (M) values bit by bit from the MSB to the LSB. In the following, for the sake of simplicity, we will assume that the values in X (M) are non-negative. It is worth noting, that 2's complement values can be straightforwardly used as well. Indeed, any set of N -bit 2's complement values can be converted in nonnegative values, preserving the order relation, by flipping the MSB.

This paper proposes a parallel VLSI architecture relying on the radix sorting approach for finding the first W > 2 maximum/minimum values in a set of M values. Namely, the proposed solution, referred to as Bit-Wise-And (BWA) architecture, works by analyzing the M candidates from the Most-Significant-Bit (MSB) to the Least-Significant Bit (LSB).

Radix sorting relies on the analysis of X(M) values bit by bit from the MSB to the LSB. In the following, for the sake of simplicity, we will assume that the values in X (M) are non-negative. It is worth noting, that 2's complement values can be straightforwardly used as well. Indeed, any set of N -bit 2's complement values can be converted in nonnegative values, preserving the order relation, by flipping the MSB.

Thus, let x a = {Xa;$_{N-1}$ Xa;$_{N-2}$: : : X$_{a;1}$ X $_{a;0}$ } and x b = {X$_{b;N-1}$ X$_{b;N-2}$: : :X $_{b;1}$ X $_{b;0}$ } be two N -bit non-negative binary values, where x a;j and x b;j represent the j -th bit of x a and x b respectively. Assuming the first (N − j − 1)-th MSBs of x a and x b have the same value, we can easily obtain the relationship between x a and x b based on bit-wise analysis: Xa > Xb if X a;j = '1' and Xb;j = '0', and vice versa

The proposed BWA relies on performing recursively the logic-and operation between adjacent bits of each xi value from the MSB to the LSB. Let h i = {hi;N −2 : : : h i;0 } be the array of bit-wise logic-and operations on x I , where hi;N −2 = xi;N −1 ∧ xi;N −2 and.

$$h_{i,j} = h_{i,j+1} \wedge x_{i,j},$$

for j = N − 3; : : : ; 0 with ∧ representing the logic-and operation. If the MSB of all the Xi values is '1' and all the Xi are monotonic sequences of bits, that is only a transition from '1' to '0' is allowed as in the four x I values of Example 1, then, analyzing the content of h i for i = 0; : : : M − 1 from the LSB to the MSB allows to find the first W maximum values

Example 1:

$$x_0 = \{1\,1\,1\,1\}$$
$$x_1 = \{1\,1\,0\,0\}$$
$$x_2 = \{1\,0\,0\,0\}$$
$$x_3 = \{1\,1\,1\,0\}$$

$$\mathcal{H} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

As an example, hi;0 = '1' if and only if xi;j = '1' for every j = 0; : : : ; N − 1, namely xi= 2N− 1. Let H be the (N − 1) × M matrix whose columns are hi. If X(M) contains only distinct elements: i) moving from the MSB-row to the LSB-row all rows of H are different up to a certain j , then for j′< j all the rows are zero (j = 0 in Example 1), ii)when moving from the LSB-row to the MSB-row, after the first non-zero row, one additional '1' appears along a column. As a consequence, moving from the LSB-row to the MSB row of H, the columns of the first W non-zero rows are the positions of the first W maximum values. Since in general xi is not a monotonic sequence and repeated elements can exist in X(M), modifications to effectively employ the BWA technique are required.

A. Completed BWA architecture

As highlighted in Section II-A, the initial BWA principle can be employed on data that are monotonic sequences of bits whose MSB is '1'. If the data in X(M)do not meet this requirement, the architecture does not work correctly. As an example, the case xi;j = '0' for a certain j and for every i = 0; : : : ; M − 1 causes hi;j′ = '0' for every j′≤ j . In this case, the architecture cannot distinguish among different xi. A similar problem arises if two or more xi values are non-monotonic sequences of bits. Thus, we add some gates to handle these cases, referred to as zero-row conditions. To this purpose we modify (1) as hi;j =^hi;j+1 ∧ xi;j where

$$\hat{h}_{i,j} = \begin{cases} z_{N-1} \vee x_{i,N-1} & \text{if } j = N-1 \\ (z_j \wedge \hat{h}_{i,j+1}) \vee h_{i,j} & \text{if } 0 \le j < N-1 \end{cases}, \quad (2)$$

∨ is the logic-or operation and

$$z_j = \begin{cases} \text{not}\left(\bigvee_{i=0}^{M-1} x_{i,N-1}\right) & \text{if } j = N-1 \\ \text{not}\left(\bigvee_{i=0}^{M-1} h_{i,j}\right) & \text{if } 0 \le j < N-1 \end{cases} \quad (3)$$

detects a zero-row condition. Follows an example.

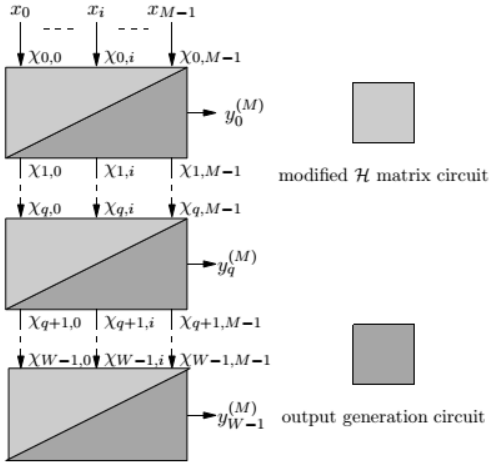


Figure 2: BWA architecture: cascade of W stages.

Example 2:

$$x_0 = \{0\ 1\ 1\ 1\} \quad z_3 = 1 \qquad \hat{\mathcal{H}} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$
$$x_1 = \{0\ 1\ 0\ 1\} \quad z_2 = 0$$
$$x_2 = \{0\ 0\ 0\ 0\} \quad z_1 = 0$$
$$x_3 = \{0\ 1\ 1\ 0\} \quad z_0 = 0$$

Example 2 shows a simple case, where the modified H matrix (^H), that is an N × M matrix, is given. Indeed, as explained in the next paragraphs, the maximum values are selected checking ^hi;0 values. Handling zero-rows leads tothe slice-architecture depicted in light gray in Fig. 1, where each slice corresponds to one row of ^H. The bottom part of Fig. 1 shows the circuit to implement (2) and (3), where ^hi;j acts as hi;j, but, if a zero-row condition occurs, then ^hi;j =^hi;j+1. As it can be observed in the modified H matrix, the proposed structure ensures^hi;0 = '1' for at least one value of i = 0; : : : M − 1. Thus, the selection of the maximum values in the proposed architecture is performed checking^hi;0 values. Let I be the set of indices i = 0; : : : ; M − 1 such that^hi;0 = '1'. If I = {k}, i.e. it contains only one element, theny0 = xk. Otherwise, X(M)contains more instances of the maximum value. If *I* contains W elements, then the first W maximum values are the elements {xi: i ∈ I} ⊂ X(M). If *I* contains less than W elements a new search is required.

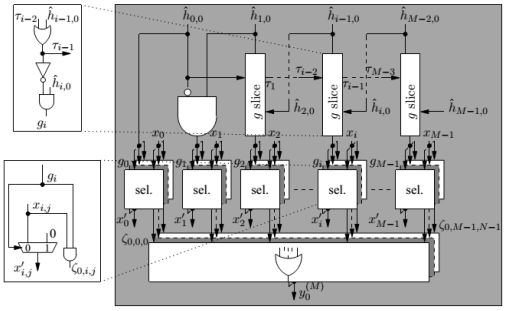


Figure 3: BWA architecture: output generation circuit in the case q = 0

To simplify the selection we use a circuit referred to as output generation circuit that, based on ^hi;0 values, is able to find the maximum among M elements and to produce a new set of M elements X′= x′0; : : : ; x′M−1where the maximum value is replaced by zero. Thus, the complete architecture, shown in Fig. 2, is made of W stages, where each stage contains one instance of the circuit to produce the modified H (light gray part) and one instance of the output generation circuit (dark gray part). As a consequence, the q-th stage finds y(M)q, that corresponds to the maximum value of the q-th input set. This operation is accomplished by the means of the output generation circuit shown in dark gray in Fig. 3 for the case q = 0. The output generation circuit relies on M − 1 blocks referred to as g slice, M ×N selection blocks and N combiners each made of an M -input logic-or, where M selection signals gi = not(Ti−1) ∧ ^hi;0 for i = 1; : : : ; M − 1 and g0 =^h0;0are used in the selection blocks to forward xi to the next slice or to replace it by zero. Each gi signal is implemented as a slice (see the top left part of Fig. 3), where the Ti−1 term is obtained as

$$\tau_{i-1} = \bigvee_{u=0}^{i-1} \hat{h}_{u,0}, \tag{4}$$

that is: when^hu;0 = '1', then the remaining T_l with l =u + 1; : : : ; M − 1 are '1' and so in the current stage only X_u is selected. More precisely, with reference to Fig. 3, gi is exploited in the selection blocks to compute ζq;i, one of the M candidates, where only one ζq;i′= 0. Each bit of ζq;i is computed as ζq;i;j = gi ∧ q;i;j where

$$\chi_{q,i} = \begin{cases} x_i & \text{if } x_i \notin \bigcup_{k=0}^{q-1} \{y_k^{(M)}\} \\ 0 & \text{otherwise} \end{cases}, \tag{5}$$

and the terms y(M)q;j and q;i;j represent the j -th bit of y(M)q and q;i respectively (see the sel. block in the left part of Fig.3). As an example, for q = 0 and q = 1 we have 0;i = xi and $X_{1;i}$ = x'_I respectively. Finally, the q-th maximum is obtained:

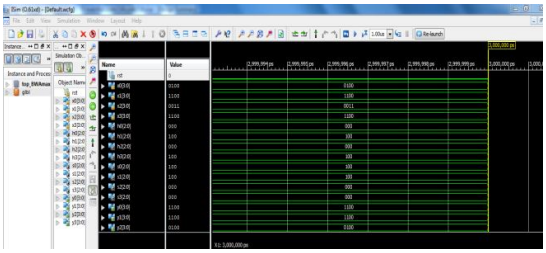
$$y_q^{(M)} = \bigvee_{i=0}^{M-1} \zeta_{q,i}, \qquad (6)$$

Corresponding to the N combiners each made of an M –input logic-or in the bottom part of Fig. 3. Pipelining the proposed architecture improves the throughput, but leads to an area overhead. As an example, adding one pipeline register between each of the W stages in Fig. 2, (i.e. W − 1 pipeline registers), implies adding W − 1 − q registers to each y (M)q , to increase the throughput by about W times.

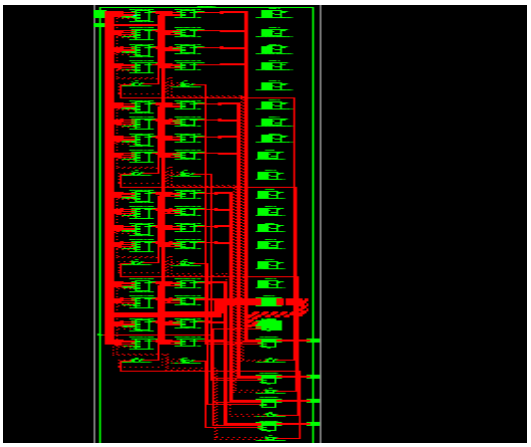
IV. RESULTS

Proposed System.

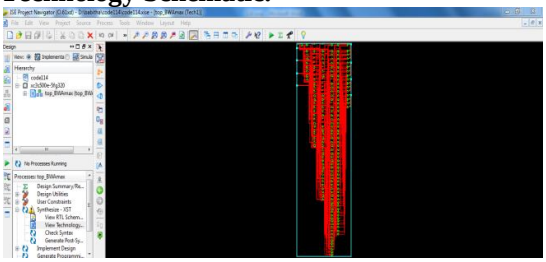
MAX Simulation.



RTL Schematic.



Technology Schematic.



Design Summary.

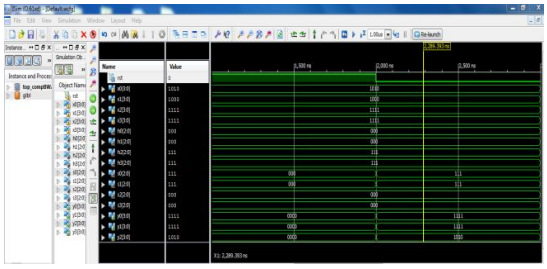
Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	98	4656	2%
Number of 4 input LUTs	171	9312	1%
Number of bonded IOBs	33	232	14%

Timing Summary.

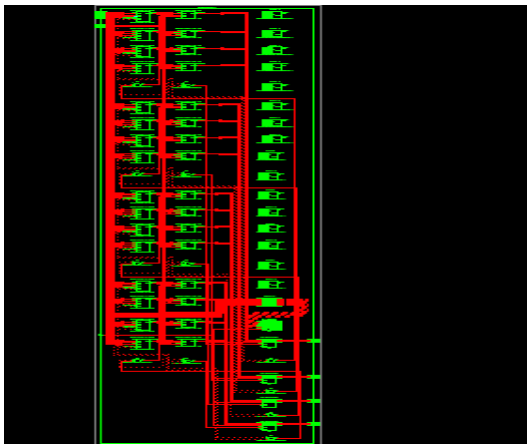
```
Timing Summary:
---------------
Speed Grade: -5

   Minimum period: No path found
   Minimum input arrival time before clock: 25.014ns
   Maximum output required time after clock: 4.114ns
   Maximum combinational path delay: No path found
```

EXTENSION.

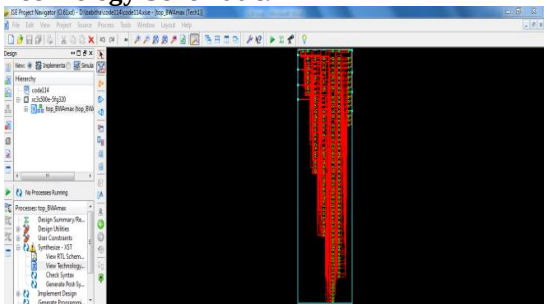
MAX Simulation.



RTL Schematic.



Technology Schematic.



Design Summary.

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	98	4656	2%
Number of 4 input LUTs	171	9312	1%
Number of bonded IOBs	33	232	14%

Timing Summary.

```
Timing Summary:
---------------
Speed Grade: -5

   Minimum period: No path found
   Minimum input arrival time before clock: 25.014ns
   Maximum output required time after clock: 4.114ns
   Maximum combinational path delay: No path found
```

V. Conclusion

The proposed architecture has an efficient architecture for layered LDPC decoding by reducing the interconnection complexity with proper and efficient decoding throughput. Our design requires only a single shuffle network, rather than the two shuffle networks used in prior designs. The results show a significant reduction in the number of required FPGA slices compared to a standard layered decoding architecture. The family of Low Density Parity Check (LDPC) codes is a strong candidate to be used as Forward Error Correction (FEC) in future communication systems due to its strong error correction capability. Most LDPC decoders use the Message Passing algorithm for decoding, which is an iterative algorithm that passes messages between its variable nodes and check nodes. It is not until recently that computation power has become strong enough to make Message Passing on LDPC codes feasible. Although locally simple, the LDPC codes are usually large, which increases the required computation power.

REFERENCES

[1] R. G. Gallager, "Low-density parity-check codes,"IRE Trans. Inform. Theory, vol. IT-8, pp. 21–28, Jan. 1968.
[2] D. J. C. MacKay, "Good error-correcting codes based on very sparsematrices,"IEEE Trans. Inform. Theory, vol. 46, pp. 399–431, Mar. 1999.
[3] T. Mittelholzer, A. Dholakia, and E. Eleftheriou, "Reduced-complexitydecoding of LDPC codes for generalized partial response channels,"IEEE Trans. Magn., vol. 37, pp. 721–728, Mar. 2001.
[4] J. Fan, A. Friedmann, E. Kurtas, and S. McLaughlin, "Low densityparity check codes for magnetic recording," inProc. 37th Allerton Conf. Commun., Control, and Computing, 1999.
[5] H. Song, R. M. Todd, and J. R. Cruz, "Applications of low-density parity-check codes to magnetic recording channels,"IEEE J. Select. Areas Commun., vol. 19, pp. 918–923, May 2001.
[6] M. C. Davey and D. J. C. MacKay, "Low density parity check codes over GF(q)," IEEE Commun. Lett., vol. 2, pp. 165–167, June 1998.
[7] , "Low density parity check codes over GF(q)," in Proc. IEEE Inform. Theory Workshop, June 1998, pp. 70–71.
[8] M. C. Davey, "Error-correction using low-density parity-check codes," Ph.D. dissertation, Univ. Cambridge, Cambridge, U.K., Dec. 1999.
[9] R. M. Todd and J. R. Cruz, "Designing good LDPC codes for partial response channels,", unpublished preprint.
[10] M. Yang and W. E. Ryan, "Performance of (quasi-) cyclic LDPC codes in noise bursts on the EPR4 channel," inProc. IEEE Global Tele commun. Conf., vol. 5, 2001, pp. 2961–2965.