Implementation of a Real Time Programmable Encoder for Low Density Parity Check Code on a Reconfigurable Instruction Cell Architecture

Zahid Khan, Tughrul Arslan

System Level Integration Group,
The University of Edinburgh,
Mayfield Road, Edinburgh, EH9 3JL, Scotland, UK
z.khan@ed.ac.uk, Tughrul.Arslan@ee.ed.ac.uk

Abstract - This paper presents a real time programmable irregular Low Density Parity Check (LDPC) Encoder as specified in the IEEE P802.16E/D7 standard. The encoder is programmable for frame sizes from 576 to 2304 and for five different code rates. H matrix is efficiently generated and stored for a particular frame size and code rate. The encoder is implemented on Reconfigurable Instruction Cell Architecture (RA) which has recently emerged as an ultra low power, high performance, ANSI-C programmable embedded core. Different general and architecture specific optimization techniques are applied to enhance the throughput. With RA, a throughput from 10 to 19 Mbps has been achieved.

I. Introduction

Low Density Parity Check (LDPC) codes are attributed to Gallager who proposed them in his 1960 PhD dissertation [1]. The research was lying in dormant due to high complex computation for encoders, introduction of the Reed-Solomon codes and the concatenation of RS and convolutional codes were considered perfectly suitable for error control coding. In 1980, Tanner [2] introduced graphical representation for LDPC codes well known as Tanner graphs. In mid-1990s, Mackay, Luby and others [3], [4], and [5] resurrected them and noticed their importance apparently independently of the work of Gallager.

LDPC provides transmission capacity approaching to Shannon's limit with decoding complexity which is linear in the block length. LDPC can provide faster communication, longer communication ranges and better transmission due to its consistently high error correction performance. LDPC processing is most suitable for parallel implementation which if properly exploited can enhance the throughput significantly.

In this paper, we present implementation of a real time programmable LDPC encoder that can support code lengths from 576 to 2304 and five different code rates which are ½, 2/3A, 2/3B, 3/4A and 3/4B. Each code rate and code length are supported by different parity check matrices that are computed in real time from the model matrices stored in the memory. The proposed architecture can be implemented in ASIC, FPGA or DSP. We implemented it on the RA [6]. This architecture belongs to the emerging field of Reconfigurable Computing and is an effort to combine the flexibility and programmability of DSP, performance of FPGA and low power consumption of ASIC in one unified core so that the core can meet the requirement of next generation mobile systems. This paper implements the encoder using the established and RA specific optimization techniques to exploit the parallelism identified inside the algorithm [6].

The rest of the paper is organized such that section 2 describes the encoding algorithm; section 3 presents the real time programmable encoder, section 4 discusses the implementation and optimization on RA while section 5 concludes the paper.

II. LDPC Encoding

This section describes the basic steps of LDPC coding as described in the IEEE P802.16e/D7 standard for producing systematic code bits.

- ***H*** is constructed from block circulant matrices using right circular shift permutation as specified by the model matrices inside the IEEE standard.
- ***H*** is divided into sub-matrices ***(A, B, C, D, E, T)*** as according to the specification. Figure 1 describes the partition of ***H*** into sub-matrices.

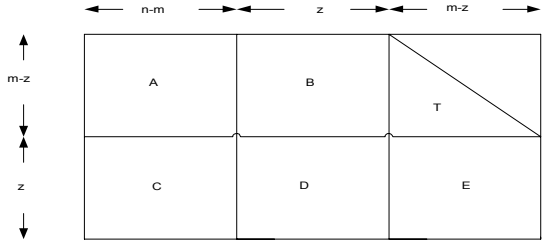


Figure 1: (Parity Check Matrix H)

- Encoding is performed as in the IEEE specifications.

III. Real Time Programmable LDPC Encoder

It is often necessary in any type of wireless communication system to adapt its transmission and reception to the varying channel conditions for maintaining a good QoS. For such adaptation, it is necessary that both transmitting and receiving sections can configure themselves in real time. Such configuration or adaptation can be with respect to code length, code rate, modulation scheme and or different encoding/decoding algorithms. Inside a particular Forward Error Correction (FEC) module, the adaptation is with respect to code length and code rate. LDPC is an example of FEC and the IEEE standard defines a specific range of both code lengths and rates which a WiMax system must support.

Such a real time adaptive architecture for IEEE specified LDPC Encoder is presented in Figure 2. This encoder consists of two parts, H matrix calculation, represented in Figure 2(B), and the actual encoding which is shown in Figure 2(A).

In previous work, H matrix calculation is done offline and the H matrix is then stored in the memory to be used for encoding. These designs can support only one code length and code rate and hence are not adaptable to real time systems.

The IEEE P802.16e/D7 [7] specification defines LDPC code which is based on a set of one or more fundamental LDPC codes. Each of the fundamental codes supports code lengths

1-4244-0630-7/07/$20.00 ©2007 IEEE.

from 576 to 2304 with code rates 1/2, 2/3A, 2/3B, 3/4A and 3/4B. Each LDPC code in the set of LDPC codes is defined by a matrix H of size m-by-n where n is the length of the code and m is the number of parity check bits in the code. The matrix H is constructed from a set of z-by-z permutation matrices or z-by-z zero matrix. The matrix H is expanded from a binary model matrix H_{bm} of size m_b-by-n_b where $n=z*n_b$ and $m= z*m_b$ where z is an integer ≥ 1. The model matrix has integer entries $\geq$ -1. H matrix is constructed by replacing each '-1' in the model matrix by z-by-z zero matrix. The '0' entry is replaced by z-by-z identity matrix while any other entry greater than 0 is replaced by a z-by-z permutation matrix. The permutations used are circular right shifts and the set of permutation matrices contains the z-by-z identity matrix and its circular right shifted versions.

The H matrix generate section in Figure 2(B) consists of three modules and memory arrays. One memory array stores the model matrices defined in the IEEE standard. Each code rate is represented by a unique model matrix. Five model matrices are stored that correspond to five code rates (1/2, 2/3A, 2/3B, 3/4A, 3/4B). The entire memory consumed by the five model matrices is 12*24 + 8*24 + 8*24 + 6*24 + 6*24 = 960 bytes. All model matrices have 24 columns whereas the rows depend upon code rates with 12 rows for ½, 8 rows for 2/3A, 2/3B, and 6 rows for 3/4A, 3/4B. The purpose of the Configuration Block is to compute the size of the H matrix, and the spreading factor 'z' for a particular code length and code rate. The spreading factor 'z' determines the size of the square matrix which is used to replace each entry in the model matrix to construct the H matrix.

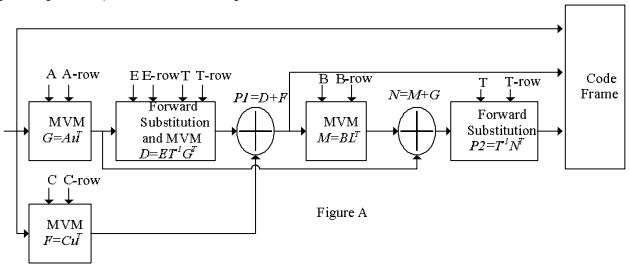
Each model matrix stores values that correspond to H matrix construction for the maximum code length of 2304. For any other code length, the entries in the model matrix is modified according to (1) and (2).

$$p(f,i,j) = \begin{cases} p(i,j), & p(i,j) \leq 0 \\ \left\lfloor \dfrac{p(i,j) * z}{z_0} \right\rfloor, & p(i,j) > 0 \end{cases} \quad (1)$$

$$p(f,i,j) = \begin{cases} p(i,j), & p(i,j) \leq 0 \\ \mathrm{mod}(\ p(i,j), z), & p(i,j) > 0 \end{cases} \quad (2)$$

where z and z_0 are the spreading factors for the H matrices of the desired code length and maximum code length respectively. The shift factor *p(f,i,j)* represents the shift sizes for the corresponding code size and *p(i,j)* is the shift size for code length of *2304* and this is the *i,jth* entry in the model matrix stored initially. The $\lfloor x \rfloor$ is flooring function giving the nearest integer towards $-\infty$.

The purpose of the Base Matrix Generate module is to take a model matrix and spreading factor corresponding to a particular code size and then establish a base matrix of the same size as the size of the model matrix but with entries defined by equations 1 and 2. The output of this module is the base matrix which is the basis for the H matrix generation. This matrix is applied to the next module that generates the child matrices A, B, C, E, and T. These children of the H matrix are generated directly instead of the H matrix. The dimension of the child matrices are defined in Figure 1. If the child matrices are stored conventionally then the total memory consumed by all these matrices for a code size of 2304 and code rate ½ is 1152*2304 = 2.53Mbits = 316Kbytes, where 1152 is the row size and 2304 is the column size of the H matrix. This is a very huge memory and dedicating this much memory to the H matrix alone is not recommend to be feasible for either DSP, FPGA or even ASIC implementation. This is for the reason that model matrices and intermediate vectors also require storage and the overall storage then becomes prohibitively high. The total memory required for an example code length of 2304 and code rate of ½ can be = 2.53M(H matrix) +1152(information bits) + (1152+1152+96)(for storing intermediate results as well as parity bits) = 324.4Kbytes.



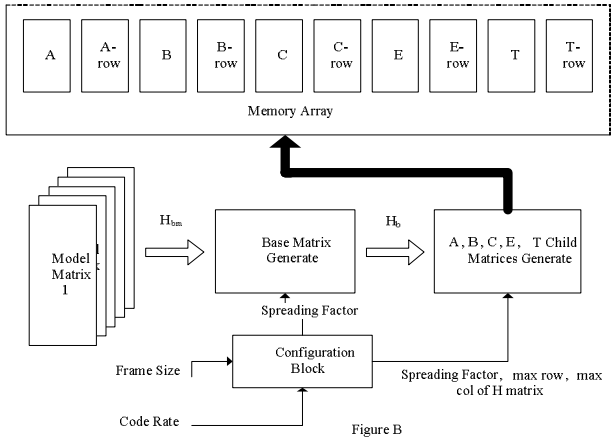


Figure 2 :(Real Time Programmable LDPC Encoder)

This huge memory storage requirement can be reduced by adopting the methodology in [8] for storing the child matrices. Since H is sparse regarding number of 1's, a huge memory reduction is possible if only the indexes of the 1's inside the H matrix or the child matrices are stored. As an example, the total number of 1's in the H matrix for 2304 code length and ½ code rate is 8024 or approximately 8K. If indexes of only the 1's inside the H matrix are stored then only 8Kwords or 16Kbytes of memory is needed in stead of 316Kbytes which is equivalent to a reduction of about 20 times in memory. The authors in [8] have stored the indexes of only 1's inside the child matrices. Each child matrix is accompanied by another array which stores a 1 for the index of H and zero for the end of the row. Therefore, a zero in this array will indicate the end of the row of the associated child matrix. This is an extra overhead which is a 1-bit array of size 8K.

Since the encoder is designed to support all code rates and code lengths, the maximum memory sizes are computed based on exhaustive MatLab simulation of the maximum code length and all code rates. The maximum memory sizes for A, B, C, E and T matrices are determined and then used in the design.

The *'**A**, **B**, **C**, **E**, **T**'* child matrices Generate' module generates the child matrices and stores the indexes of the 1's inside them in the corresponding memory array. The memory array x-row stores '0' for each entry and '1' for the end of each row. Here x represents a child matrix. This array is used

as pointer to indicate the end of row for matrix-vector computation.

After child matrices generation, they are applied to the encoder (figure 2(A)) together with the information bits. The processing in each block is similar to the processing defined in [8]. Here MVM is the matrix-vector-multiplication with a vector output. Forward Substitution is used to obtain $y^* T^{-1} = x$ using $y = T * x$.

The first MVM module in Figure 2(A) generate $F=C*u^T$ and $G = A*u^T$. The vector G is applied to the next module to generate $D=E*T^{-1}*G^T$. D and F are vector added in the next VA module shown by the encircled plus sign. The output of the VA module is $P1$ which is a subset of the parity check bits that contribute to the code bits. The next MVM module generates the vector $M=B*P1^T$. The VA adder module then adds using modulo-2 addition and generates $N=M+G$. The last module uses forward substitution to generate $P2=T^{-1}N^T$ and the frame is then given by $c = [u\ P1\ P2]$.

The serial implementation can be implemented on either ASIC, FPGA or DSP. For speed improvement on either ASIC or FPGA a pipelined version of the encoder is also presented as shown in Figure 3.

This is a four stage pipelined encoder which promises to improve the speed by approximately 2.5 times at the expense of increased memory of size (1152+1152)*3 bits. The 2.5 times is because of the second stage as the second stage is the most time consuming block and cannot be divided into smaller blocks. RA can implement both due to its flexibility.

IV. Implementation on RA

RA [6] is a dynamic reconfigurable fabric which has coarse grained heterogeneous functional units (cells) connected to each other through a reconfigurable interconnect structure. The functional units support primitive operators that can perform addition/subtraction, multiplication, logic, multiplex, shift and register operations. Additional functional units (cells) are provided to handle control/branch operations. Each functional unit operates on 32-bit operands, however it can be configured for 8, 16 and 32-bit operands. Some of the functional units like shifts, logic and multiplexers can be configured to work as four 8-bit or two 16-bits functional units inside one functional unit. This implies that a 32-bit adder can work as four 8-bit or two 16-bit adders independent of each other. This type of reconfigurability is very useful in packed data computation for speed improvement.

There are three major optimization techniques for increasing the throughput [10]. These are classified as Loop Unrolling, Split Computation and Multi Sampling. Loop unrolling is applied to independent iterations to use the resources as much as possible.

The current implementation of RA is structured so that it can have 8 memory read and 8 memory write interfaces and 16 memory banks. The memory width of each bank is 8-bit and 16 banks provide four 32-bit memory accesses in parallel. With each memory interface, 32-bit word can be accessed thus allowing four 32-bit memory access operations in parallel.

RA has other optimizations like converting small conditional branches into simple multiplexing operations, storing intermediate results in distributed registers to avoid more costly memory access and providing more functional units for

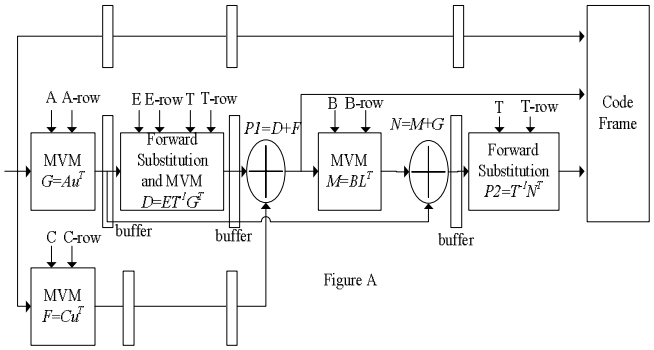


Figure 3: (Pipelined Real Time Encoder)

parallel execution of operations. The functional units and interconnects in RA allow the possible exploitation of instruction level parallelism inside the code to as much as the number of functional units. In this paper, the objective is to exploit generic and RA specific optimizations for increased throughput.

The code for the encoder consists of Vector_Add, Matrix_Vector_Mult, Forward_Substitution, Base_Matrix_Generate, and H_Matrix_Generate. Of all these modules Base_Matrix_Generate and H_Matrix_Generate are called once at start up or whenever the parameters on which H matrix generation depends get changed. These parameters are the code rate and the number of information bits used to produce code word of a particular length. The code is first simulated without applying any optimization techniques. The simulated result shows only 3.5 Mbps throughput using code length of 2304 with code rate ½. The throughput has been increased to 10 Mbps for the same rate and code length by applying generic and RA specific optimization to different modules. These are discussed below:

A. *Optimization specific to algorithm*

This is carried out mainly in the T child matrix which consists of a combination of two types of sub-matrices. One set is the non-permutated $z_f \times z_f$ identity matrix and the other is the $z_f \times z_f$ zero matrix. The construction of the T matrix inside a model matrix is given below by the red letters.

```
{ 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{ 0,  0, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{-1,  0,  0, -1, -1, -1, -1, -1, -1, -1, -1},
{-1, -1,  0,  0, -1, -1, -1, -1, -1, -1, -1},
{-1, -1, -1,  0,  0, -1, -1, -1, -1, -1, -1},
{-1, -1, -1, -1,  0,  0, -1, -1, -1, -1, -1},
{-1, -1, -1, -1, -1,  0,  0, -1, -1, -1, -1},
{-1, -1, -1, -1, -1, -1,  0,  0, -1, -1, -1},
{-1, -1, -1, -1, -1, -1, -1,  0,  0, -1, -1},
{-1, -1, -1, -1, -1, -1, -1, -1,  0,  0, -1},
{-1, -1, -1, -1, -1, -1, -1, -1, -1,  0,  0},
{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  0},
```

Figure 4: (Only T matrix inside the model matrix is shown)

Here **0** is $z_f \times z_f$ identity matrix while **-1** represents $z_f \times z_f$ zero matrix. The **T** matrix is used in the forward substitution block in the encoder. Usually it is required to find **Y** in $\mathbf{Y} = \mathbf{T}^{-1}\mathbf{x}$.

In a conventional way, the indexes of the 1's inside T matrix are stored in the PtrtoT array. This index (which is the location of 1 inside the T matrix) is then used as pointer to either the input array **x** which is represented by Ptrtovin or the output array **Y** represented by Ptrtovout. The equation

$\mathbf{Y} = \mathbf{T}^{-1}\mathbf{x}$ is solved using forward substitution $\mathbf{TY} = \mathbf{x}$ and the code for this equation is given below.

```
for (i=0; i<zf; i+=4)
   {  *(Ptrtovout + i+0) = *(*(PtrtoT+i+0)+ Ptrtovin);
      *(Ptrtovout + i+1) = *(*(PtrtoT+i+1)+ Ptrtovin);
      *(Ptrtovout + i+2) = *(*(PtrtoT+i+2)+ Ptrtovin);
      *(Ptrtovout + i+3) = *(*(PtrtoT+i+3)+ Ptrtovin);   }

for (i=zf; i<indext; i+=8)    {
*(Ptrtovout + count+0) = *(*(PtrtoT+i+0)+Ptrtovout) ^
*(*(PtrtoT+i+1)+Ptrtovin);
*(Ptrtovout + count+1) = *(*(PtrtoT+i+2)+Ptrtovout) ^
*(*(PtrtoT+i+3)+Ptrtovin);
*(Ptrtovout + count+2) = *(*(PtrtoT+i+4)+Ptrtovout) ^
*(*(PtrtoT+i+5)+Ptrtovin);
*(Ptrtovout + count+3) = *(*(PtrtoT+i+6)+Ptrtovout) ^
*(*(PtrtoT+i+7)+Ptrtovin);   }
```

Figure 5: (unmodified code for forward substitution)

If we look at the model matrix then the distribution of 1's inside the **T** matrix provides enough uniformity to avoid the computation of the **T** matrix. The first row and column of the **T** matrix starts with $z_f \, x \, z_f$ identity matrix while the rest of the columns of the first row are zeros and can be ignored. This implies that there exists only a single 1 in each row of the **T** matrix for the first z_f rows. The second row of the model matrix has two **0**'s which correspond to two $z_f \, x \, z_f$ identity matrices. This shows that each of the z_f rows represented by second row of the model matrix has two 1's respectively at locations i and $i+z_f$ where $i = 0, 1, 2, \ldots\ldots, z_f$ The third and subsequent rows except the last one follow the same pattern as of the second row.

This implies that there is no need to compute the indexes of the 1's inside the **T** matrix. All we need is to remember the indexes from the start which is 0, the size of the spreading factor and the number of rows inside the **T** matrix which is dependent upon the code rate selected. By doing this the memory operations required to read the indexes from the **T** array are eliminated which will result in subsequent saving of both energy and execution time. The following code has been written to implement the uniformity inside the **T** matrix.

```
index = m-zf;     // m is the total  rows and zf is the spreading factor
for (i=0; i<zf; i+=4)
   {    Ptrtovout[i]     =   Ptrtovin[i+0];
        Ptrtovout[i+1]  =  Ptrtovin[i+1];
        Ptrtovout[i+2]  =  Ptrtovin[i+2];
        Ptrtovout[i+3]  =  Ptrtovin[i+3];    }
count=0; //zf
for (i=zf;i<index;i+=4) //8)
  { Ptrtovout[i]     = Ptrtovout[count]   ^ Ptrtovin[i];
    Ptrtovout[i+1]  = Ptrtovout[count+1] ^ Ptrtovin[i+1];
    Ptrtovout[i+2]  = Ptrtovout[count+2] ^ Ptrtovin[i+2];
    Ptrtovout[i+3]  = Ptrtovout[count+3] ^ Ptrtovin[i+3];
    count +=4;   }
```

Figure 6: (Forward Substitution)

The code has been simulated on RA and the simulation time for unmodified and modified code came out to be 20.62 μsec and 14.224 μsec respectively. This corresponds to a reduction of 31% in execution time for the forward substitution module.

B. Optimization of Vector_Add

The aim of this module is to provide modulo-2 addition of two input vectors. Since the addition is bit wise, enough parallelism is present inside the module. This parallelism is exploited for increased throughput through simple loop unrolling. The original code is as:

```
for (i=0;i<zf;i++) {  // zf is spreading factor and is 24 to 96
*(Ptrtovout+i) = *(Ptrtovin1+i) ^ *(Ptrtovin2+i) ; }
```

The above code has 2*zf read and zf write memory operations. It will take 3*zf cycles in any DSP processor and if 4 nsec is taken as the memory access time then its execution will take 12*zf nsec on a DSP Processor. If the code is expanded as below then all the four memory reads from the same memory array can be done in just one 4 nsec time and all the four writes to the same memory array can also be completed in just one 4 nsec time. The memory access time is then (zf/4 (read)+zf/4(read)+ zf/4 (write))*4 = 3*zf which is equivalent to a 4 times reduction in memory access time.

```
for (i=0;i<zf;i+=4) {
*(Ptrtovout+i+0)= *(Ptrtovin1+i+0) ^ *(Ptrtovin2+i+0) ;
*(Ptrtovout+i+1)= *(Ptrtovin1+i+1) ^ *(Ptrtovin2+i+1) ;
*(Ptrtovout+i+2)= *(Ptrtovin1+i+2) ^ *(Ptrtovin2+i+2) ;
*(Ptrtovout+i+3)= *(Ptrtovin1+i+3) ^ *(Ptrtovin2+i+3) ;}
```

In the case of packed computation in which four independent data bytes are accessed as one data pack and processed as separate and independent 4 data bytes, only two read interfaces and one write interfaces are required thus reducing the burden on the resources. RA also supports 8 adders and 8 XOR operations. This implies that address calculation and XORing can be done in parallel thus saving more execution time.

C. Optimization of Matrix_Vector_Mult

This module performs multiplication of matrix (A) with a vector. The matrix consists of only 1's and 0's and only the indices of 1's inside the matrix are stored in PtrtoA. In such a case, the multiplication becomes picking the bits in the array Ptrtovin at the location pointed by the indices of 1's (inside the matrix) and XORing for one row of the matrix A. The code for this multiplication is given below:

```
for (j=0;j<=index;j++) {
        a = PtrtoA[j];
        *(Ptrtovout+j)  ^= *(a+Ptrtovin);}
```

The total memory access operations are 2*index (read)+ index(write) = 3*index. The memory access time is 3*index*4. If the code is written as below:

```
for (j=0;j<=index; j+=4)  {
  a0 = *(PtrtoA+j+0);      a1 = *(PtrtoA+j+1);
  a2 = *(PtrtoA+j+0);      a3 = *(PtrtoA+j+1);
*(Ptrtovout+i)  =  *(a0+Ptrtovin)  ^  *(a1+Ptrtovin)  ^  *(a2+Ptrtovin)  ^
                   *(a3+Ptrtovin);        }
```

In the modified code, 4 memory read operations a0 to a3 take two memory read accesses (2*4 nsec). The next 4 memory read operations take one memory access (4 nsec) and the write operation also takes 4 nsec. Thus 3*2*4*index/4= 6*index nsec is the execution time of the code which is equivalent to a reduction of 12/6 = 2.

D. Optimization in initializing fixed arrays

In situations where it is extremely necessary to initialize the fixed array, loop unrolling is used in array initialization. Significant reduction in memory access time has been achieved with this optimization.

E. Replacing Jumps with Multiplexing

RA executes the code in steps. A step is defined as combination of instructions that can be executed in the fabric provided by RA. The group of instructions need not be necessarily independent of each other. A step is determined by the number of available resources, conditional branch and the length of the critical path.

RA is structured to support only one jump per step. If a function or a piece of code generates more than one jumps then RA places them in separate steps. The main idea behind optimization is to place as much code inside one step as possible. If the entire function is placed in one step then maximum utilization of resources can be achieved and the time of execution will most probably be less than the case in which the same function takes more than one step. For the jump to be replaced with multiplexing, the variables and the paths for the execution must be independent of each other and must not include a memory read in its conditioning. This can be explained from the following code.

```
if(*(PtrtoMrow+j+4))
  {   *(Ptrtovout+i) = *(*(PtrtoM+j+4)+Ptrtovin) ^ *(Ptrtovout+i);
          j++;
  }
```

The above code will definitely produce a jump as the memory access in the code inside the parenthesis is made dependent upon the 'if' condition. If the 'if' condition is true then there will be memory access otherwise not. The jump produced by this code will cause underutilization of resources. This code can be structured like the following to avoid the instantiation of the jump cell.

```
temp1   =   *(PtrtoMrow+j+4)
temp2   =   *(*(PtrtoM+j+4)+Ptrtovin) ^ *(Ptrtovout+i);
temp3   =   *(Ptrtovout+i);

*(Ptrtovout+i)  = temp1  ?  temp2  :  temp3;
```

By storing the intermediate values in the registers and making the variables independent of each other, the instantiation of the jump cell has been avoided. The same strategy has been adopted in all other places inside the code to either reduce or eliminate the jumping and using multiplexers to perform the same functionality as is performed with jump cell.

Loop unrolling has been used in addition to the above mentioned techniques and the execution time and throughput has been calculated as follows:

```
One time execution of LDPC Encoder              = 554.926
Number of steps taken:                            106129
Two times execution of LDPC Encoder             =   666.232 µsec
Number of steps taken:                            123674
Execution time of actual encoding
to be used in real time              = 666.232 – 554.926 = 111.3 µsec
Number of steps of actual encoding   = 123674  – 106129 = 17545
Execution time per bit               = 111.3/1152 = 96.61 nsec/bit
Throughput                           = 1/96.61 = 10.4 Mbps (½ rate).
```

For code rate ¾ , the throughput is measured to be approximately 19 Mbps. This is the highest code rate that IEEE 802.16 defines for the irregular LDPC codes.

F. Hardware Pipelining the code

The hardware pipelining is about pipelining the memory read, computation and memory write operations inside a code. If a loop consists of all these processes and brought inside one step then the step will loop into itself. Such a step can be pipelined and using a two stage pipeline, the execution time can be reduced by about 2.5 times.

An example of a harware pipeling is shown in Figure 7 and 8. This is the graph of a data path that reads from the memory, adds together the words read from the memory and then writes the results to the same location inside the memory. The complete datapath has been brought into one RA step and it loops to itself. The execution time of this step has been measured to be 28 nsec and if the maximum iteration is 120, the execution time of the entire loop is 28*120 = 3.36 µsec.

Using pipelining, the data path can be divided into three smaller data paths with critical path length not exceeding 10 nsec. The time of execution of the step is now approximately 2.5 times less than the total time and is about 3.36/2.5 = 1.344 µsec. By using hardware pipelining, the throughput of the encoder is achieved in the range from 26 Mbps for ½ code rate to 47 Mbps for ¾ code rate. This is better than an FPGA [8] implementation which achieved 22 Mbps from a real time programmable LDPC encoder for regular codes.

V. Conclusion

In this paper, the authors have presented a novel architecture for the real time LDPC encoder as specified in the IEEE P802.16e/D7 standard targeting WiMax applications and discussed its optimization on a reconfigurable instruction cell architecture (RA). Several algorithmic and RA specific optimization techniques have been applied. The throughput achieved without pipelining is in the range from 10 to 19 Mbps while with pipelining, it is in the range from 26 to 47Mbps. This is a considerable throughput while providing the flexibility and programmability from a high level such as 'C' programming.

VI. References

[1] R. Gallager, "Low-Density parity-check codes", IRE Trans. Information Theory, pp. 21-28, Jan. 1962

[2] R.M. Tanner, "A recursive approach to low complexity codes", IEEE Trans. Information Theory, pp. 533-547, Sept. 1981

[3] D. MacKay and R. Neal, "Good codes based on very sparse matrices", in Cryptography and coding, 5th IMA Conf., C.Boyd, Ed., Lecture Notes in Computer Science, pp. 100-111, Berlin, Germany: Springer, 1995

[4] D. Mackay, "Good error correcting codes based on very sparse matrices", IEEE Trans. Information Theory, pp. 399-431, March 1999

[5] N. Alon and M. Luby, "A linear time erasure-resilient code with nearly optimal recovery", IEEE Trans. Information Theory, pp. 1732-1736, Nov. 1996.

[6] Yi. Ying, I. Nousias, M. Milward, S. Khawam, T. Arslan, I. Lindsay, "System-level Scheduling on Instruction Cell Based Reconfigurable Systems", Design, Automation and Test in Europe, 2006. DATE '06. Proceedings, Volume 1, 6-10 March 2006 Page(s):1 - 6

[7] IEEE P802.16E/D7 Specification published in 2006

[8] L. Dong-U, L. Wayne, W. Connie, J. Christopher, "A flexible hardware encoder for low density parity check codes", Proc. IEEE Symp. Field-Programmable Custom Computing, 2004

[9] SC140 Application Note: How to implement a Viterbi Decoder on the StarCore SC140. Application number ANSC140VIT: Available on www.freescale.com

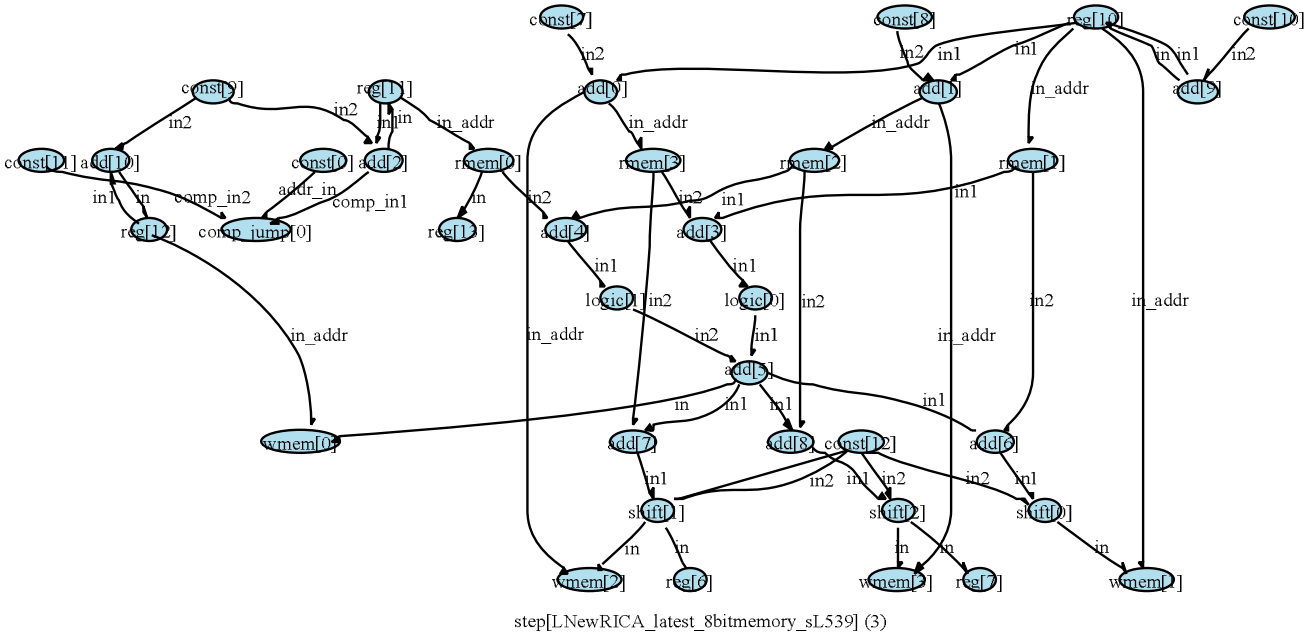


Figure 7:(Non-pipelined RA Step) (rmem is read memory cell, wmem is write memory cell, const is a cell to store constant values, add is adder cell, reg is register cell and shift is shifter cell, comp_jump is logical cell)

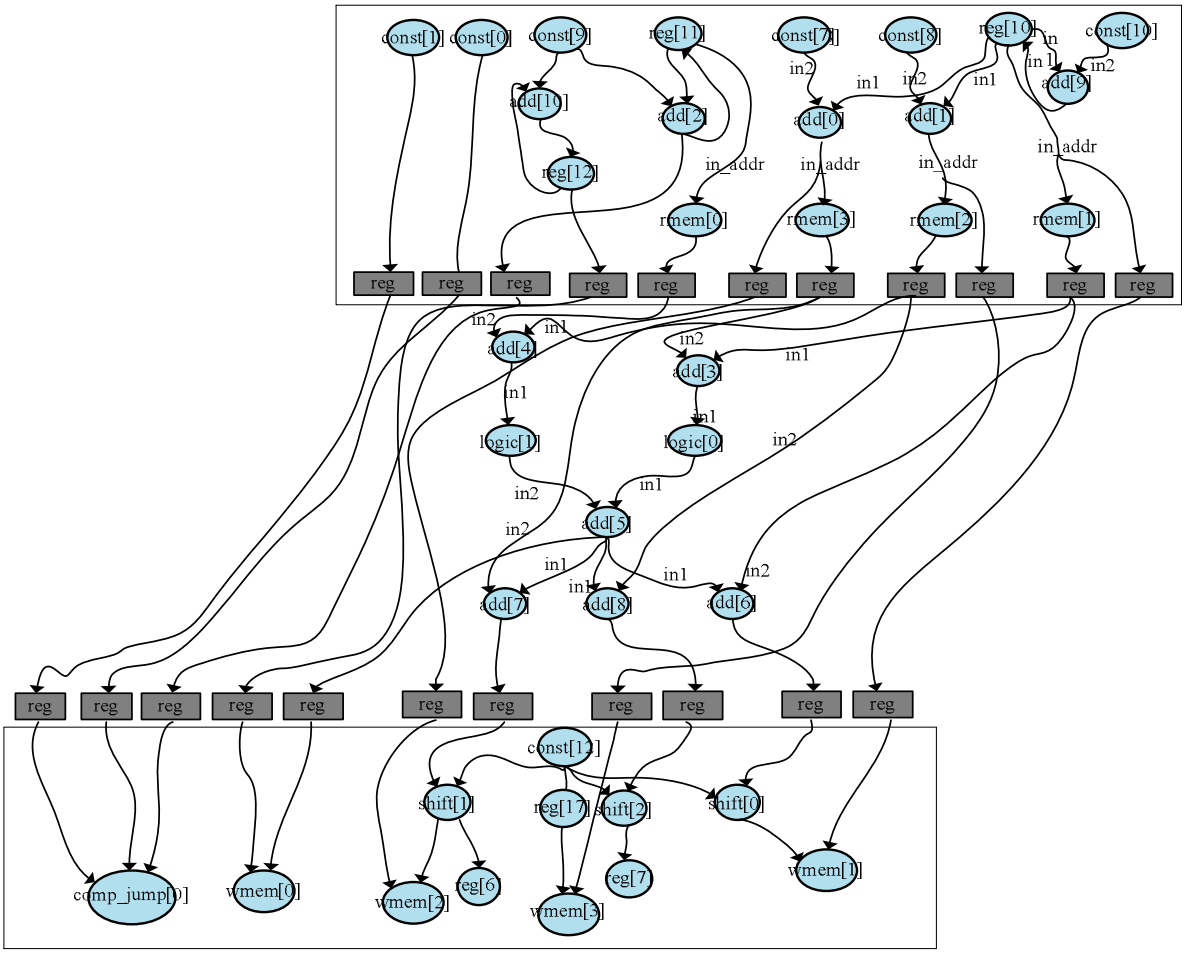


Figure 8: (Pipelined RA Step)