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

Zahid Khan, Tughrul Arslan

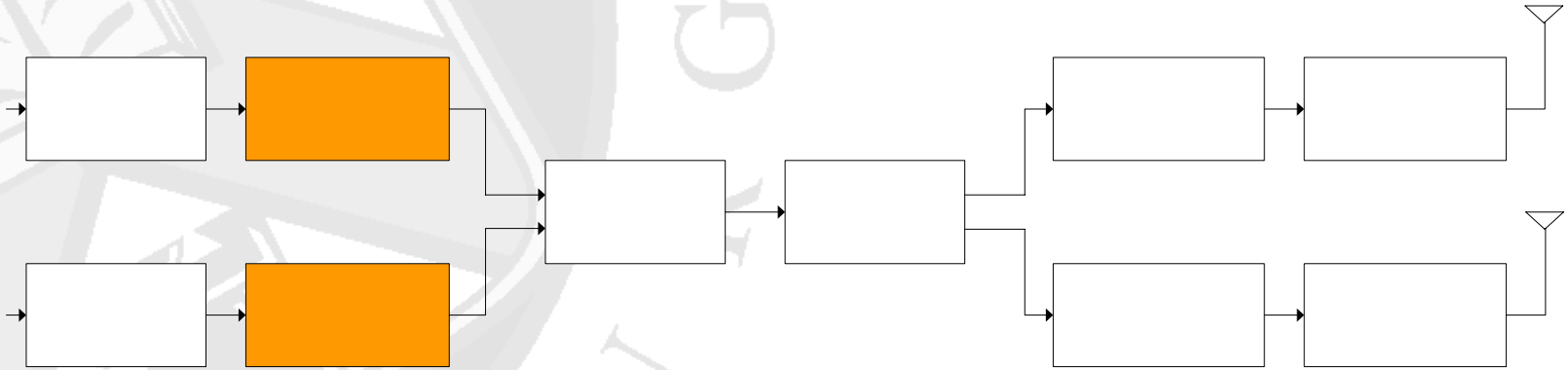
System Level Integration Group
School of Engineering and Electronics
The University of Edinburgh,
Scotland, UK

z.khan@ed.ac.uk, Tughrul.Arslan@ee.ed.ac.uk,

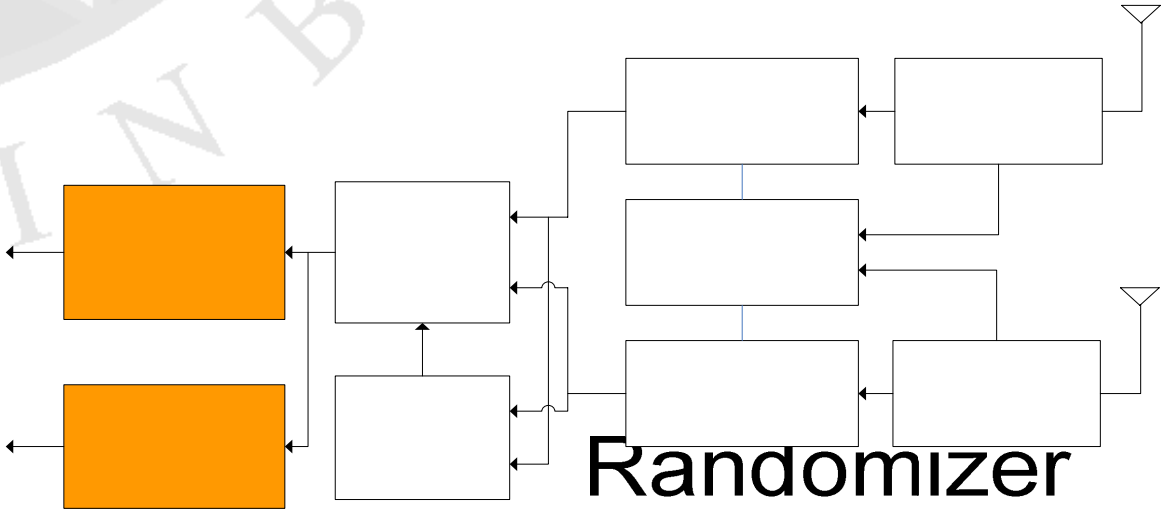
Outline

- System Overview
- Introduction to LDPC coding and encoding algorithms
- Need for Real time Encoding
- Real time Encoder
- Memory Optimization
- Features of Reconfigurable Instruction Cell Architecture (RICA)
- Implementation and optimization on RICA
- Results and Conclusion

System Overview of LDPC Coding in 802.16E For WiMax



WiMax Transmitter



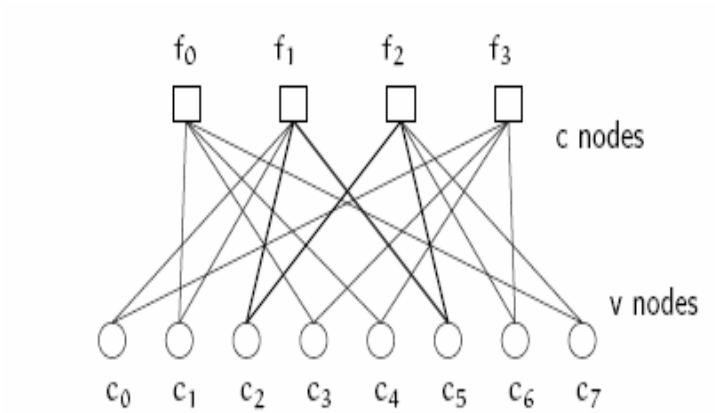
WiMax Receiver

LDPC
a
Modu

LDPC Representation

- Matrix Representation
- Parity check matrix with dimension *n x m*
- For low density matrix $w(H_c) \ll n$ and $w(H_r) \ll m$
- Graphical Representation
- Tanner introduced bipartite graphical representation for LDPC codes.
- Bipartite graph is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent.
- The two types of vertices in a Tanner graph are called variable nodes (v nodes) and check nodes (c nodes)

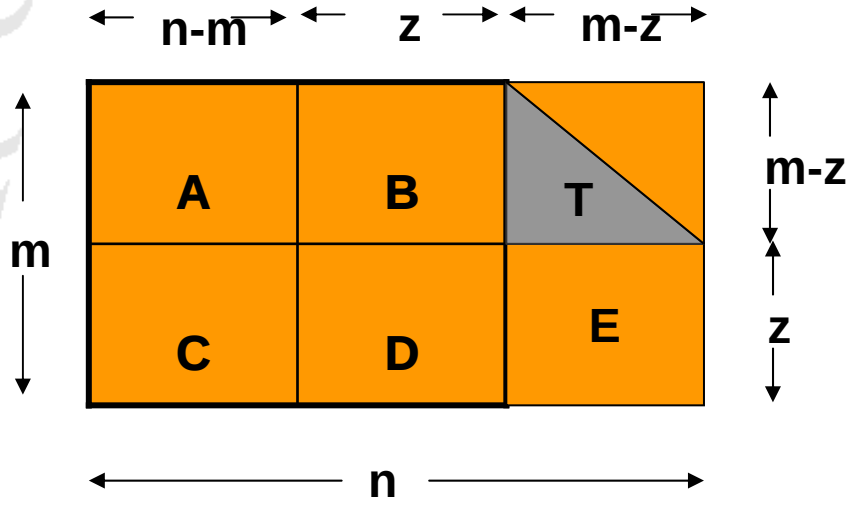
$$H = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$



Encoding Algorithms

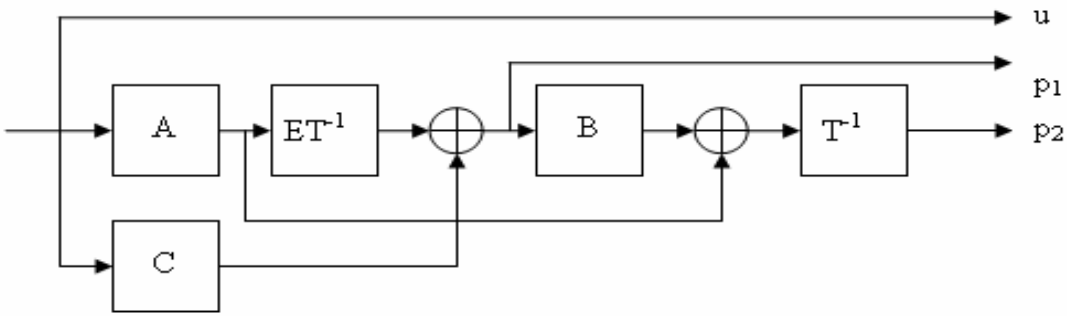
Algorithm

- ***H*** is constructed from identity matrices using right circular shift permutation
- ***H*** is divided into sub-matrices ***(A,B,C,D,E,T)*** as according to the IEEE specification.
- The ***T*** matrix is made lower triangular without losing the sparseness of the ***H*** matrix using column and row permutations.



- Since the ***H*** remains sparse, encoding Complexity is $O(n+z^2)$ which is almost linear with the length of the code.
- Various length codes can be accommodated easily.
- For code rate 1/2, number of effective cycles of computation are 0.017^2n^2+n
- .

Encoding Steps



- Construction of ***H*** as according to the IEE802.16E/D7 standard for variable code length and rate
- Permuting ***H*** row as well as column wise to make *T* approximately lower triangular.
- Base Model Matrix according to code length and rate
- Splitting ***H*** as according to Algorithm 4
- Encoding the information bits as according to Algorithm

Real Time Programmable Encoder

- Varying channel conditions and good QoS requires adaptation
- Configuration of both encoder and decoder is necessary for such adaptation.
- Adaptation can be with respect to Frame Size, Code Rate, modulation scheme and/or different encoding/decoding algorithms
- Inside a particular FEC, adaptation is w.r.t. Frame Size and Code Rate
- A Real time adaptable/programmable LDPC Encoder is proposed that can adapt on the fly to varying frame sizes and code rates as defiend by the IEEE 802.16 for WiMax Application

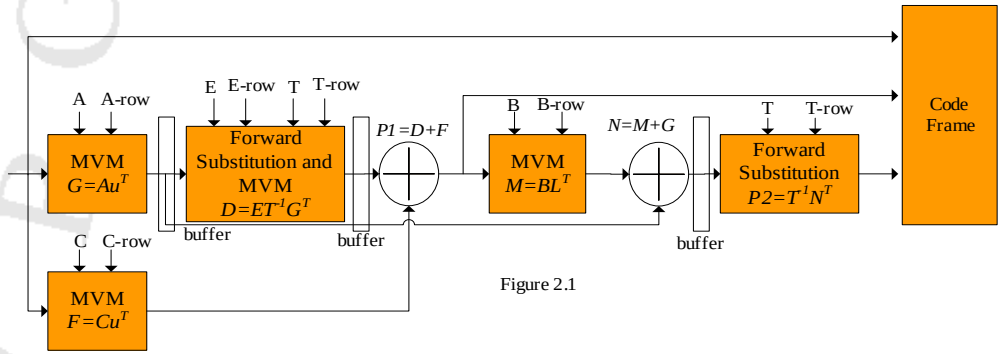


Figure 2.1

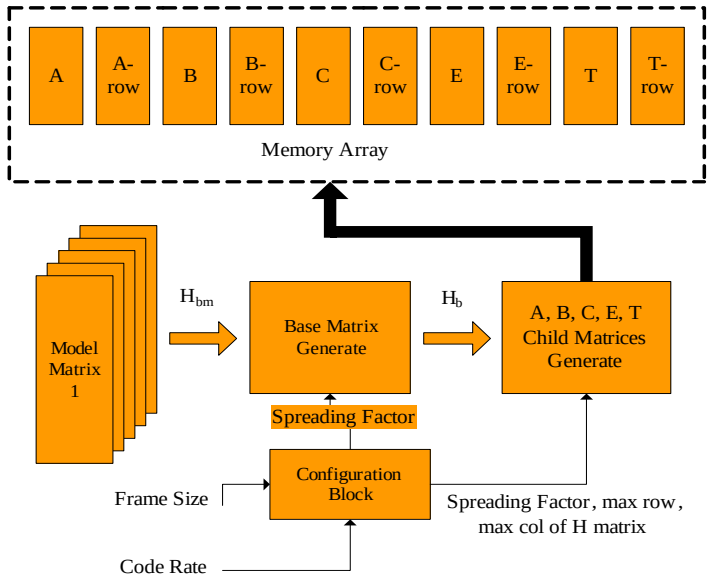


Figure 2.2

Real Time Programmable Encoder

Operation

- **Encoder has two blocks:**
  - H matrix Generate
  - Actual Encoding
- **Real Time H matrix Generate**
  - H matrix is generated from the model matrices defined in IEEE standard
  - Model matrix generates Base Matrice (H_b)
  - H_b then generates H in the form of child matrices A, B, C, E and T
- **Actual Encoding**
  - The encoder takes the child matrices (A, B, C, E and T) and the information bits to generate the parity bits according to the architecture shown in Figure 2.1

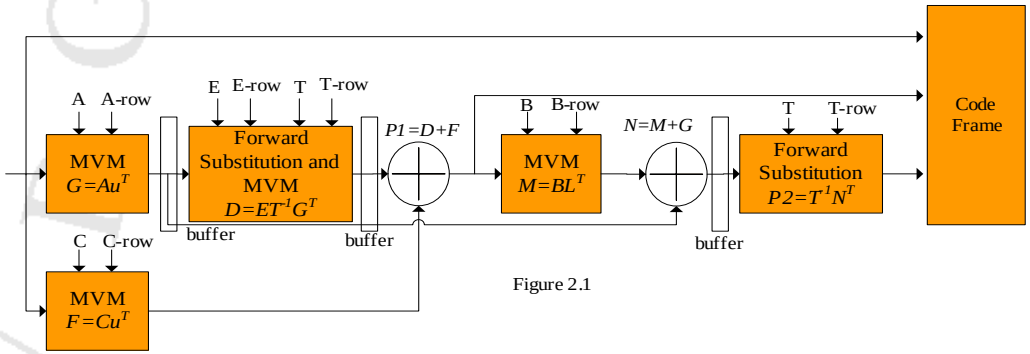


Figure 2.1

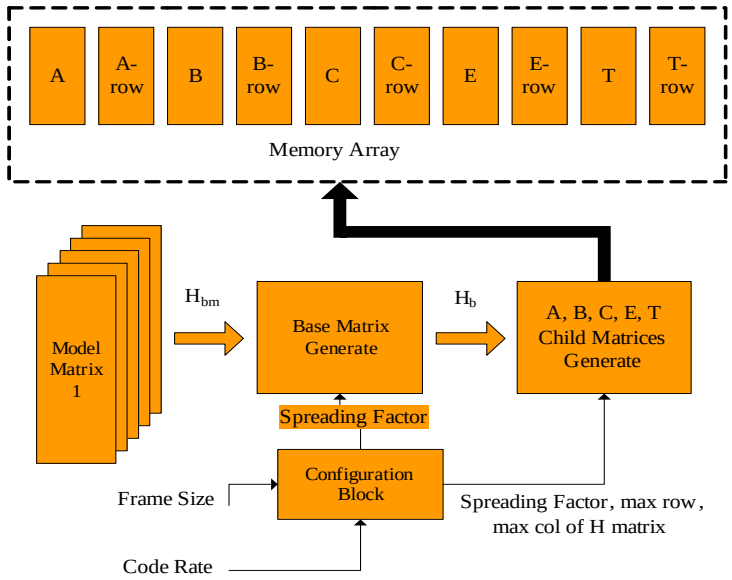
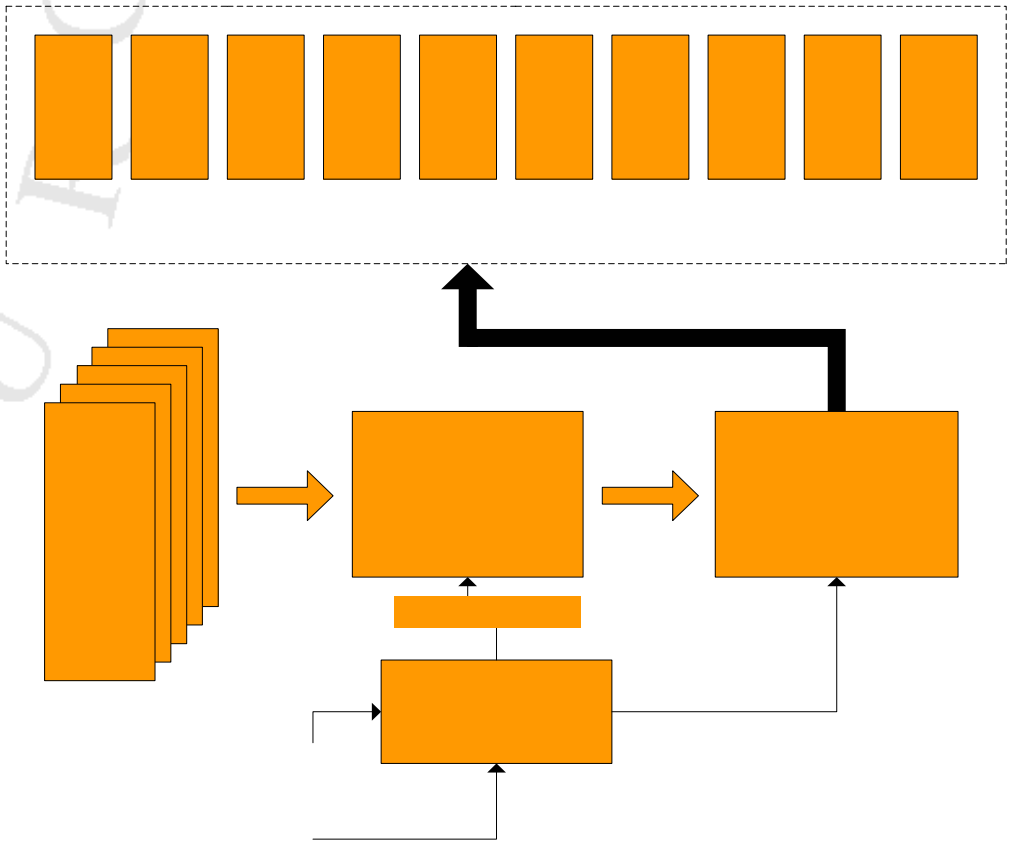


Figure 2.2

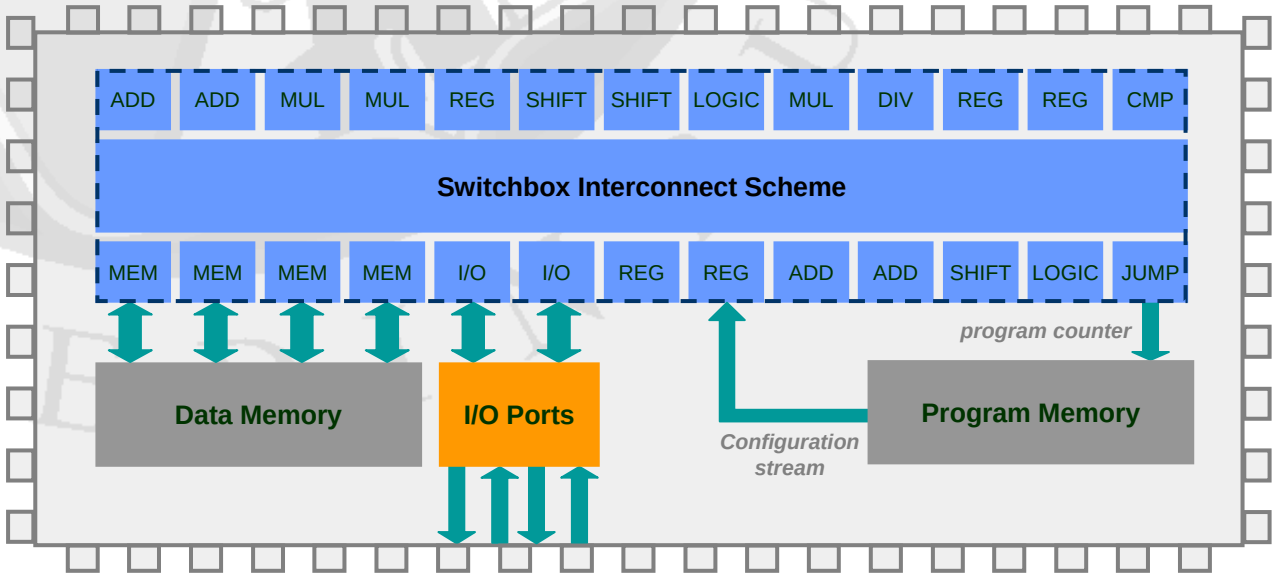
Memory Optimization in H Matrix Storage

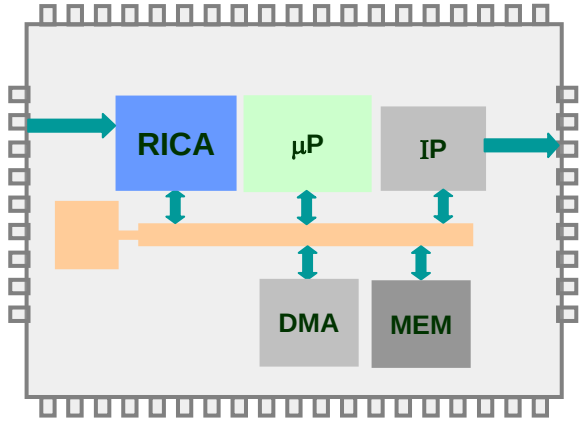
- H matrix consists of 1's and 0's
- Storing 0's and 1's will take huge memory
  - For Code Rate = ½ and Frame size = 2304 bits, the memory for H= 1152 * 2304 =2.53 Mbits = 316 Kbytes
- Direct storage is not recommended for huge memory requirement
- An alternative way is to store the indexes of 1's inside the H matrix
- Memory consumption is affordable
  - For Code Rate=1/2 and Frame size =2304, required memory is 16K bytes
- Its equivalent to 20 times reduction

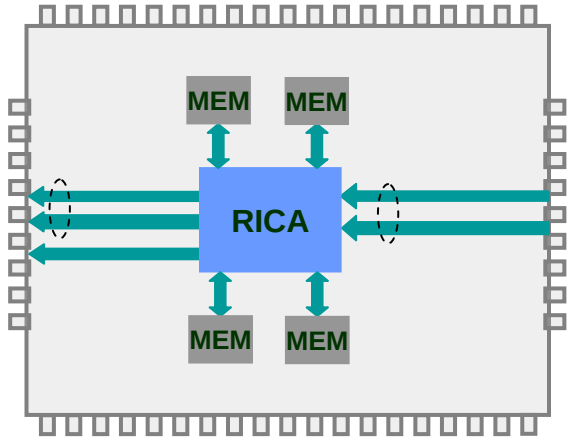


RICA

A single processing engine capable of converging data, cellular and multimedia processing on mobile devices

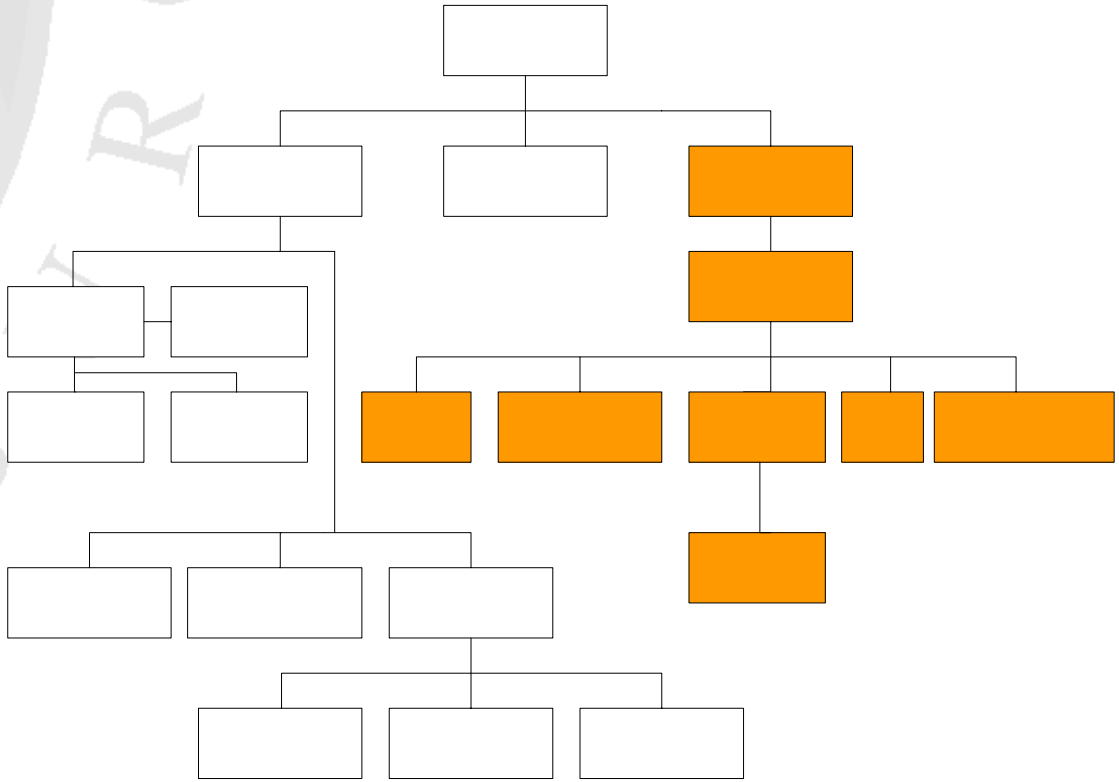






Code Optimization

- **Code Hierarchy**
  - Vectoradd.c, H_Matrix_Generate.c, Base_Matrix.c, MVM.c, Forward_Substitution.c
- **Simulation results** of the un-optimized code shows 3.5 Mbps for code rate 1/2.
- General, Algorithmic and RICA specific optimization techniques are used for manual optimization
- **Optimization** is focussed on Vectoradd.c, MVM.c and Forward_Substitution.c as they are used in actual encoding



Optimization Specific to Algorithm

```
{-1,94,73,-1,-1,-1,-1,-1,55,83,-1,-1, 7, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
{-1,27,-1,-1,-1,22,79, 9,-1,-1,-1,12,-1, 0, 0,-1,-1,-1,-1,-1,-1,-1,-1,-1},
{-1,-1,-1,24,22,81,-1,33,-1,-1,-1, 0,-1,-1, 0, 0,-1,-1,-1,-1,-1,-1,-1,-1},
{61,-1,47,-1,-1,-1,-1,-1,65,25,-1,-1,-1,-1,-1, 0, 0,-1,-1,-1,-1,-1,-1,-1},
{-1,-1,39,-1,-1,-1,84,-1,-1,41,72,-1,-1,-1,-1,-1, 0, 0,-1,-1,-1,-1,-1,-1},
{-1,-1,-1,-1,46,40,-1,82,-1,-1,-1,79, 0,-1,-1,-1,-1, 0, 0,-1,-1,-1,-1,-1},
{-1,-1,95,53,-1,-1,-1,-1,-1,14,18,-1,-1,-1,-1,-1,-1,-1, 0, 0,-1,-1,-1,-1},
{-1,11,73,-1,-1,-1, 2,-1,-1,47,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0, 0,-1,-1,-1},
{12,-1,-1,-1,83,24,-1,43,-1,-1,-1,51,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0, 0,-1,-1},
{-1,-1,-1,-1,-1,94,-1,59,-1,-1,70,72,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0, 0,-1},
{-1,-1, 7,65,-1,-1,-1,-1,39,49,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0, 0},
{43,-1,-1,-1,-1,66,-1,41,-1,-1,-1,26, 7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0},
```

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

- This is carried out mainly in the **T** child matrix
- Two types of child matrice
  - z_f x z_f zero matrix
  - z_f x z_f identity matrix
- T matrix is used in forward substitution **TY = x** to solve $\mathbf{Y=T^{-1}x}$
- No need to compute **T** due to uniformity in the distribution of 1's

Optimization Specific to algorithm

- Initial coding involves huge amount of memory accesses for the T matrix

```
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);
}
```

- Modified code involves less memory accesses due absence of 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;   }
```

RICA Specific Code Optimization

- Vector_Add
  - It adds modulo-2 two input vectors
  - Modulo-2 addition is bit wise, enough parallelism is present
  - Parallelism is exploited for increased throughput
  - Initial code has **$2*z_f$** read and **z_f** write access---- a total of **$3*z_f$**
  - 4 parallel memory banks in RICA reduces memory accesses to **$z_f/4$ (read) + $z_f/4$ (read) + $z_f/4$ (write) = $3*z_f/4$**
  - Significant reduction in execution time with loop unrolling has been achieved
- Memory Initialization
  - Loop unrolling is also used to initialize memory arrays. The loop is unrolled by a factor of 4 for optimum optimization

The original code is:

```
for (i=0;i<zf;i++)
   {
      Ptrtovout[i] = Ptrtovin1[i] ^ Ptrtovin2[i] ;
   }
```

```
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] ;
   }
```

```
for (i=0;i<zf;i+=4)
   {
      Ptrtovout[i+0] = 0 ;
      Ptrtovout[i+1] = 0 ;
      Ptrtovout[i+2] = 0 ;
      Ptrtovout[i+3] = 0 ;
   }
```

RICA Specific Code Optimization

- **Forward_Substitution**
  - The module performs **y = T^{-1}*x** using forward substitution **x = T * y**
  - Loop is unrolled by four
  - Significant reduction in cycle count has been achieved
- **Reducing Memory Accesses**
  - In coding style 2, *(PtrtoB+j+0) is used twice
  - The compiler calculates the effective address twice and then reads the value stored at the effective address twice as well.
  - This can be reduced to one access and one calculation by storing the value on stack in a temporary variable and then using the value stored in the variable for further processing.

```
count=0;

for (i=zf;i<indext;i+=4)
{
    Ptrtovout[i+0] =  Ptrtovout[count+0]   ^  Ptrtovin[i+0];
    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;
 }
```

Coding Style 1:

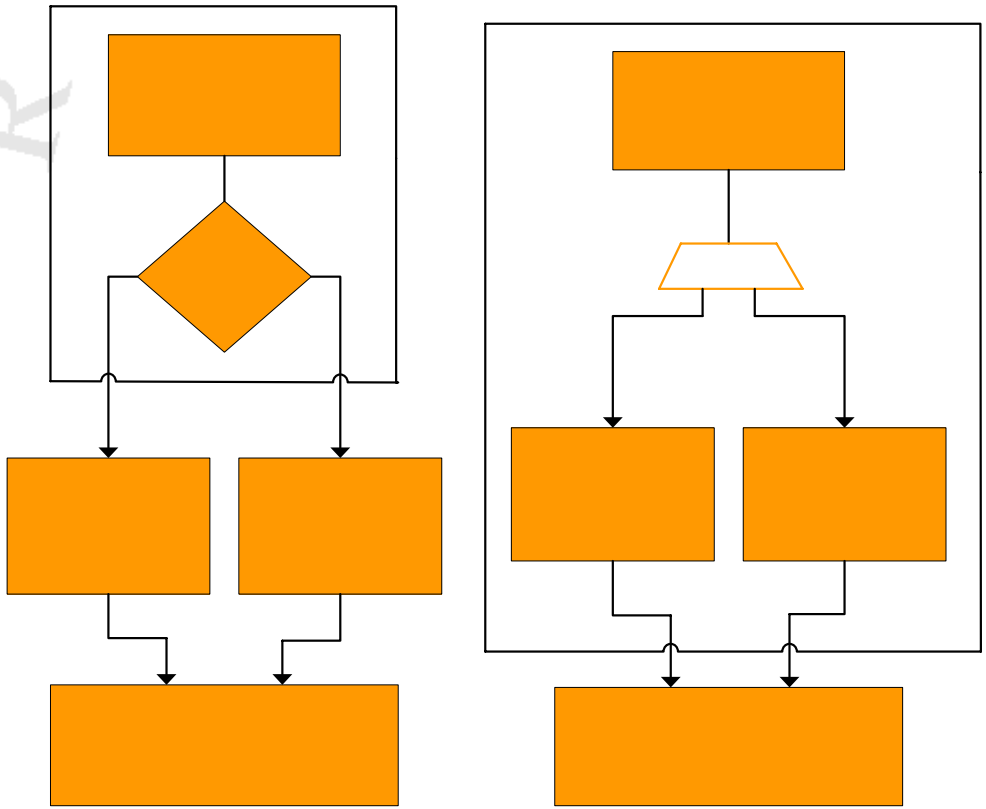
```
TmpBvalue=*(PtrtoB+j+0);
if(TmpBvalue==-1)       *(PtrtoBp1+i)  = 0;
else   *(PtrtoBp1+i) ^= *(TmpBvalue + Ptrtop1);
if(*(PtrtoBrow+j+0))            i++;
```

Coding Style 2:

```
if(*(PtrtoB+j+0)== -1)     *(PtrtoBp1+i)  = 0;
else              *(PtrtoBp1+i) ^= *(*(PtrtoB+j+0) + Ptrtop1);
if(*(PtrtoBrow+j+1))            i++;
```

Code Optimization

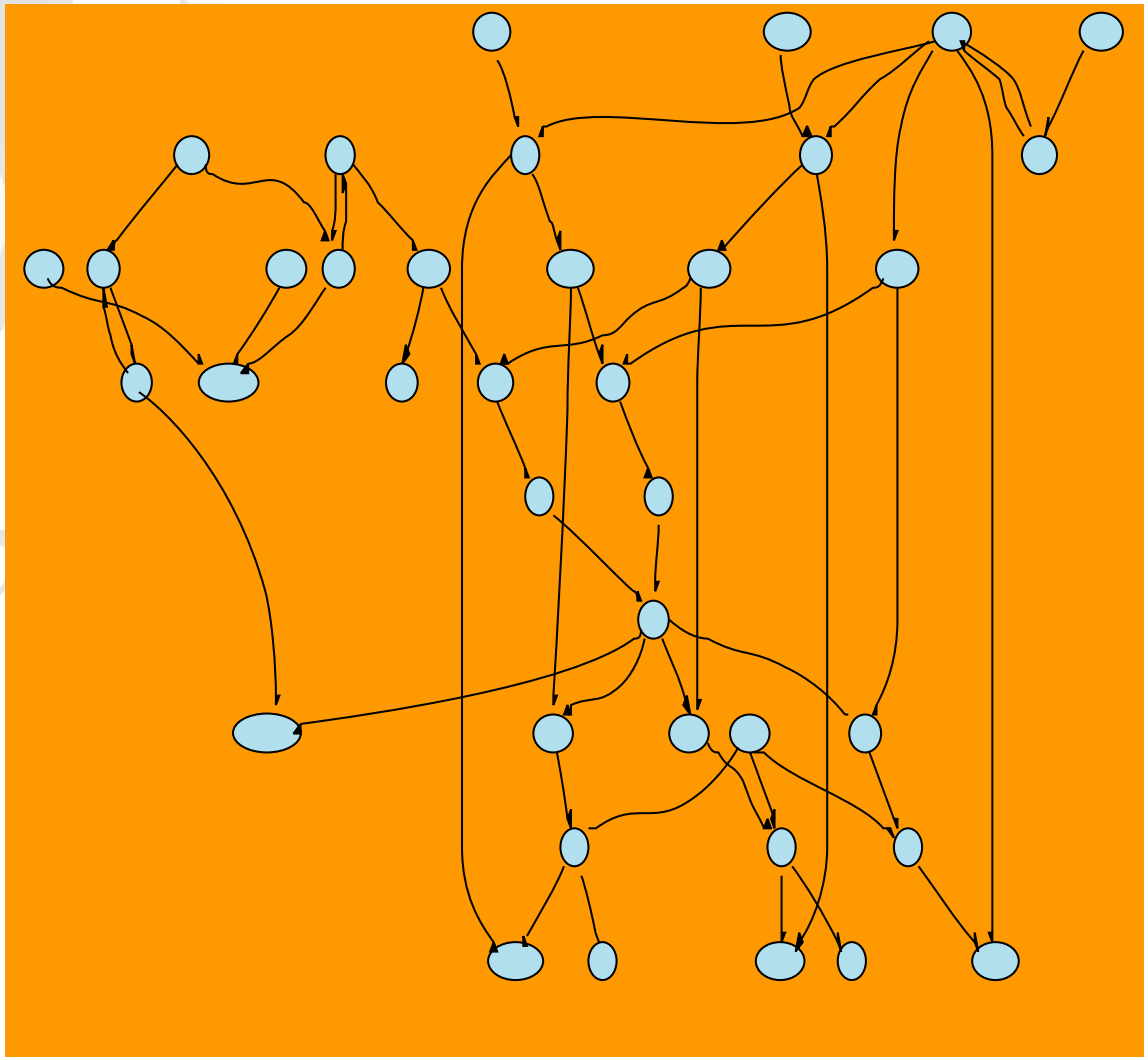
- **Replacing jumps with multiplexing**
  - **RICA executes the code in steps.** A step is defined as combination of instructions that can be executed in the fabric provided by RICA.
  - A step is determined by the number of available resources, conditional branch and the length of the critical path.
  - **RICA is structured to support only one jump per step.** The reduction in number of steps is related to reducing the execution time due to reduction in configuration time overhead as well as the possibility that the longer step will execute some of the code in parallel
  - Jumps are replaced as much as possible with multiplexers



RICA Specific Optimization

Hardware Pipelining

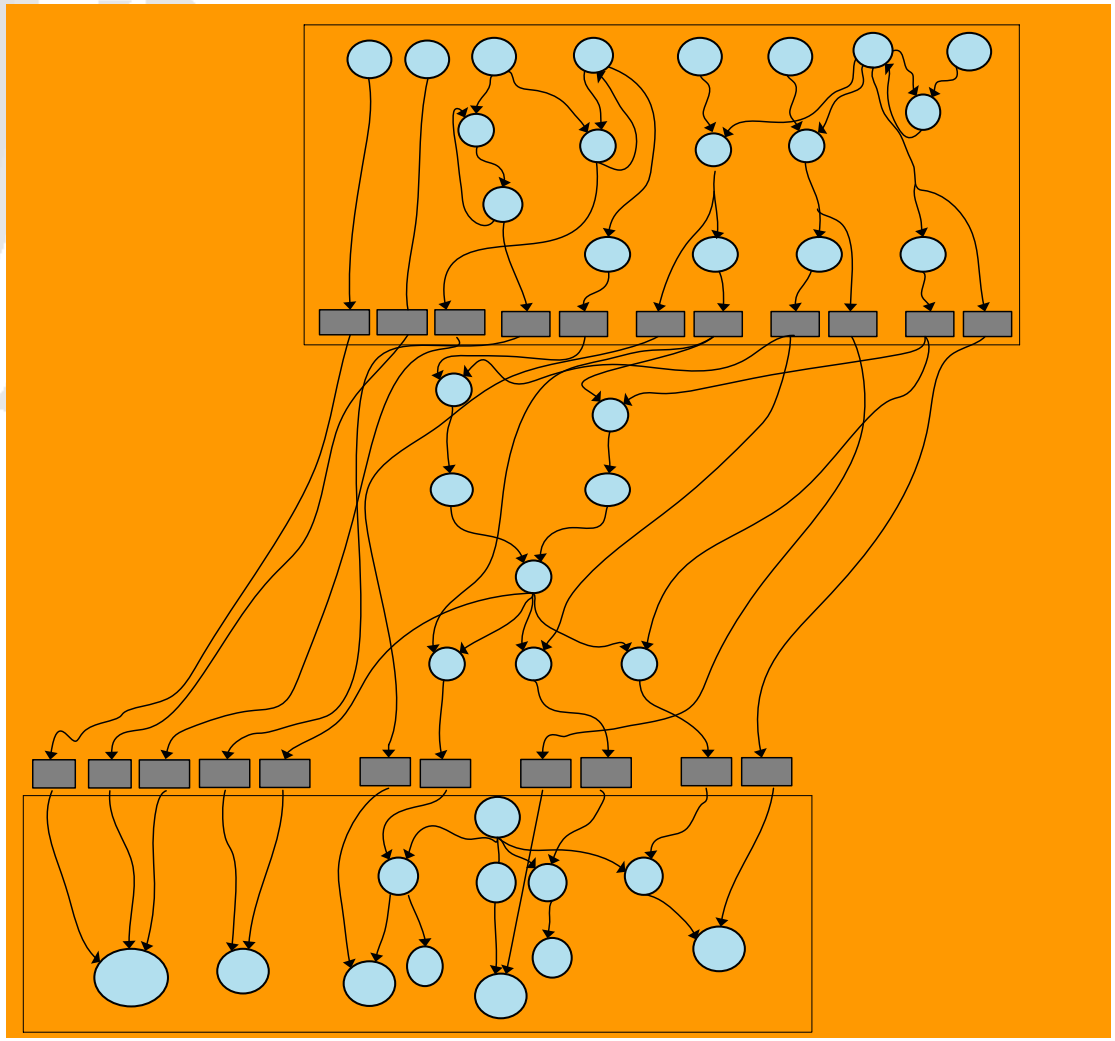
- An example step has been shown
- This step involves reading data from the memory, performing some computations and then writing the results to the memory
- This step loops to itself
- The execution time of this step has been computed to be 28 nsec per iteration and for 120 iterations, the total execution time **= 28 * 120 = 3.36 µsec**
- Read, Computation and Write operations can be pipelined to reduce the computation time



RICA Specific Optimization

Hardware Pipelining

- The step is pipelined
- Pipelined registers are inserted at two stages to increase the clock speed.
- Due to practical constraints, the pipelined step should not be less than 10 nsec.
- The execution time of this step has been computed to be 10 nsec per step and for 120 iterations, the total execution time 1.2 µsec
- Overhead of registers and of course area and power consumption



Results

One time execution of LDPC Encoder = 554.926 µsec
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.

With pipelining the code, the throughput are given below

For code rate ½, throughput = 26 Mbps
For cdoe rate ¾, throughput = 47 Mbps

Conclusion

- A Novel architecture for the Real time programmable LDPC Encoder for Mobile WiMax applications as specified in the IEEE P802.16E/D7 standard has been suggested
- This is the first implementation for the real time LDPC encoding for WiMax applications.
- The architecture has been implemented on RICA with RICA specific and generic optimizations applied to the code.
- We achieved 2.8 times improvement in throughput compared to the un-optimized code that corresponds to 10.4-19 Mbps.
- The pipelined version resulted in 26 to 47 Mbps throughput
- A similar but not real time FPGA implementation has resulted in 22 Mbps throughput. However, RICA implementation is C programmable compared to that of FPGA.
- Further reduction is still possible not only by exploiting the parallel processing elements but also by exploiting the uniformity inside the model matrices specified in the 802.16

Thank You