EE705 Course Project – LDPC Decoder

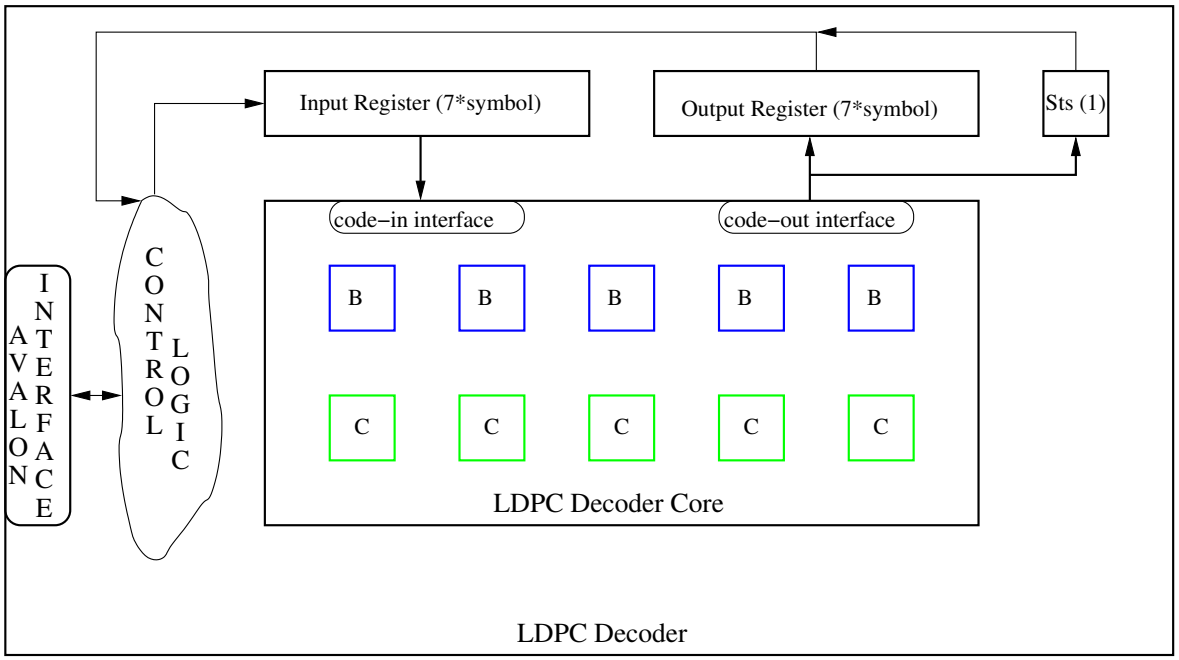
Anuja Singh (184074001), Niraj N Sharma (184077001), Sandeep Mishra (184076005)
Sonali Shukla (184070014)

Interim Report: April, 2019

The project involves the design and layout of a LDPC decoder block. After the creation of behavioural RTL we have chosen to use an open-source tool based design flow to generate the GDS2 for the design.

1 Block Diagram

The LDPC decoder core represents the design which will be developed and implemented upto GDS2. The LDPC decoder top-level represents the top-level for functional verification on a Nano FPGA board using Quartus.



2 Current Status and Plan Ahead

The design of the behavioural verilog for the decoder is nearly complete. The remaining steps are described below:

Task	Tool	Status
Functional Verification	Quartus	Ongoing
Synthesis to Gate level	Yosys	Ongoing
CTS and STA	??	TBD
Layout	Java Electric	TBD

The rest of the document describes the RTL design of the LDPC Core and its components.

3 Nodes of the LDPC Decoder

The work-horses of the decoder are the bit and check nodes. These iteratively decode the input code-word passing the partially decoded result back-and-forth for a fixed number of iterations.

3a ⟨*boilerplate* 3a⟩≡

```
// EE-705 Course Project -- LDPC Decoder

package Nodes;

// ------------------------------------------------------------------
// This package defines:
//
//    CheckNode   : Interface to the Check Nodes
//    BitNode     : Interface to the Bit Nodes
//    mkCheckNode : Micro-arch of the Check Node
//    mkBitNode   : Micro-arch of the Bit Node
//
//    v1.0        : The nodes can handle one code-word at a time
//
// ------------------------------------------------------------------

import ClientServer     :: *;
import GetPut           :: *;
import FIFO             :: *;
import Vector           :: *;
import LdpcTypes        :: *;
```

⟨*type definitions* 3b⟩

⟨*module definition* 5a⟩

The Bit-node and the Check-node are the two types of computation nodes in the LDPC decoder. One iteration consists of processing at both of these nodes. The interconnections between them is described by the incidence matrix (H matrix).

3b ⟨*type definitions* 3b⟩≡ (3a)

⟨*interface definition* 4a⟩

4 Interfaces of the Bit and Check Nodes

Two interface types, one each for the Bit and Check nodes are defined in this package. Both interfaces are parameterized by the numeric types `NConnections`.

The numeric type, `NConnections` indicates the number of ones along a particular row of the incidence matrix, which is also same as the number of ones along a particular column of the incidence matrix.

4a ⟨*interface definition* 4a⟩≡ (3b) 4b▷

```
// NConnections indicates the number of connections that a particular
// check-node has with the bit-nodes
interface CheckNode;
   interface Vector #(NConnections, Put #(Symbol)) b2c;
   interface Get #(Symbol) c2b;
endinterface
```

A check node can be imagined to receive `Symbol` values from a vector of interfaces. Each interface is connected to a different bit nodes. In turn, the check node returns a single `Symbol` values broadcast to all its bit nodes.

4b ⟨*interface definition* 4a⟩+≡ (3b) ◁4a 4c▷

```
// NConnections indicates the number of connections that a particular
// bit-node has with the check-nodes
interface BitNode;
   // Bit Node-Check Node Interface
   interface Vector #(NConnections, Put #(Symbol)) c2b;
   interface Get #(Symbol) b2c;
```

The inverse applies for the bit-check node connection from the bit node's perspective. It sends a single `Symbol` values broadcast to all its check nodes, and collects responses through a vector of interfaces, each connected to a different check node.

4c ⟨*interface definition* 4a⟩+≡ (3b) ◁4b

```
   // External interfaces for receving code word and returning result
   interface Put     #(Symbol)   codeIn;
   interface Get     #(Symbol)   dataOut;

endinterface

// ------------------------------------------------------------------
```

In addition to the sub-interfaces to connect the check and bit nodes, bit nodes also have the additional interfaces to receive the code and send back the decoded data. Since each bit-node only deals with a part of the code-word, it is sufficient to receive those symbols only. The final decoded word is also constructed from the responses of the different bit-nodes. Each bit node receives one symbol to decode at a time.

5 The Bit Node

The `mkBitNode` module receives the code word and initiates the iterations. Based on `NConnections` it is connected to a set of `mkCheckNodes` representing the edges of the bipartite tanner graph. The `mkBitNode` provides an interface of type `BitNode`.

5a ⟨*module definition* 5a⟩≡ (3a) 9c▷

```
// Core function of the bitNode
⟨functions bitNode 7a⟩

//
// Bit Node Module definition
(* synthesize *)
module mkBitNode (BitNode);
    ⟨state bitNode 5b⟩
    ⟨rules bitNode 6a⟩
    ⟨interfaces bitNode 9a⟩
⟨wrap up bitNode 9b⟩
// ----------------------------------------------------------------------
```

The input FIFO `ffCodeIn` receives the symbol of the code word meant for this bit-node. The output FIFO `ffDataOut` holds the decoded symbol.

5b ⟨*state bitNode* 5b⟩≡ (5a) 5c▷

```
// Sub-modules and state
// Input FIFO - code word
FIFO  #(Symbol)   ffCodeIn    <- mkFIFO;

// Output FIFO - decoded code word
FIFO  #(Symbol)   ffDataOut   <- mkFIFO;
```

The `ffB2C` FIFO holds the partially processed codeword. The contents of this FIFO will be consumed by the check node when they are ready. The `vffC2B` FIFO receives partially processed codewords from the check nodes for the next iteration of processing.

5c ⟨*state bitNode* 5b⟩+≡ (5a) ◁5b 6b▷

```
// Partially processed codeword meant for the checknodes
FIFO  #(Symbol)      ffB2C       <- mkFIFO;

// Partially processed codeword from the checknodes
Vector #(
     NConnections
   , FIFO #(Symbol)) vffC2B      <- replicateM (mkFIFO);
```

Behaviour is described in terms of atomic sets of actions called rules. The rule, `rlProcessFirstIteration` executes the actions for the first iteration of processing a new code word.

- Consume the codeword which is currently in `ffCodeIn`
- Carry out some initial processing on the codeword
- Enqueue the result into the `ffB2C` for checknode processing
- Update the iteration count - this is updated by 2 as the count is maintained only in the bit node, and the check node acts as a purely passive device.

The first iteration is counted when the code word goes through the check node for the first time.

6a ⟨*rules bitNode* 6a⟩≡ (5a) 8▷

```
// Rules and behaviour

// Rule to process the first iteration of a new code word
rule rlProcessFirstIteration (rgIterationCount == 0);
   // As this is the first iteration, consume the codeword which is
   // currently in ffCodeIn. Carry out the computation on the codeword
   let codeIn = ffCodeIn.first; ffCodeIn.deq;
   let codeOut = fnInitialBitNodeProcessing (codeIn);

   // Send the output to the check nodes
   ffB2C.enq (codeOut);

   // Bookkeeping - keep track of iterations to know when to stop. The
   // first iteration is treated as the first time the code word goes
   // through the check node
   rgIterationCount  <= rgIterationCount + 1;
endrule
```

The register, `rgIterationCount` keeps track of the number of iterations for this particular code word. When a certain prefixed iteration limit is reached, processing stops. This is a system-wide setting.

6b ⟨*state bitNode* 5b⟩+≡ (5a) ◁5c

```
Reg   #(Bit #(4))                rgIterationCount  <- mkReg (0);

// ----------------------------------------------------------------
```

The function `fnBitNodeProcessing`, carries out the actual bit manipulation of the codeword symbols as per the min-sum-algorithm. The noinline attribute ensures that a separate verilog module is generated for the function.

7a ⟨*functions bitNode* 7a⟩≡ (5a) 7b▷

```
(* noinline *)
function Symbol fnBitNodeProcessing (Vector #(NConnections, Symbol) x);
   // XXX This is at present a dummy function which simply does an
   // XOR of the symbols
   // It will actually do some sort of a zip function which goes:
   // Vector#(N,Symbol) -> Symbol
   return (foldl1 (\^ , x));
endfunction
```

The function `fnInitialBitNodeProcessing`, does not work on a vector of Symbols like `fnBitNodeProcessing`. It can be thought of as a function which prepares the input symbol for iterative processing.

7b ⟨*functions bitNode* 7a⟩+≡ (5a) ◁7a

```
function Symbol fnInitialBitNodeProcessing (Symbol x);
   // XXX This is at present a dummy function which simply does an
   // forwards the signal to the output
   return (x);
endfunction
```

The rule `rlProcessRemIteration` executes the actions for the remaining iterations for processing a code word. The input for these iterations is from the partially processed word in `vffC2B`.

- Consume the codeword which is currently in `vffC2B`
- Carry out the computation on the codeword
- Enqueue the result into the `ffB2C` for checknode processing
- Update the iteration count - this is updated by 2 as the count is maintained only in the bit node, and the check node acts as a purely passive node.

8 ⟨*rules bitNode* 6a⟩+≡ (5a) ◁6a

```
// Rule to process remaining iterations
rule rlProcessRemIteration (
      (rgIterationCount > 0)
   && (rgIterationCount < fromInteger (valueOf (NIterations))));
   // As this iteration works of a partial result from the checknode,
   // the input comes from the vector of fifos vffC2B
   Vector #(NConnections, Symbol) codeIn;
   for (Integer i=0; i<valueOf(NConnections); i=i+1) begin
      codeIn[i] = vffC2B[i].first;
      vffC2B[i].deq;
   end

   let codeOut = fnBitNodeProcessing (codeIn);

   // Bookkeeping - keep track of iterations to know when to stop
   if (rgIterationCount == (fromInteger (valueOf (NIterations))-1)) begin
      // time to stop
      rgIterationCount <= 0;

      // Send the processed code word to the output
      ffDataOut.enq (codeOut);
   end

   // continue with more iterations
   else begin
      rgIterationCount  <= rgIterationCount + 2;

      // Send the output to the check nodes
      ffB2C.enq (codeOut);
   end
endrule

// ----------------------------------------------------------------
```

The `rlProcessRemIteration` rule also needs to check if the iterations are complete. If so, it should reset the `rgIterationCount` and instead of sending the output to the `ffB2C` it should sent it to `ffDataOut`.

Creating the interfaces simply involves stitching up the connections to the input and output FIFOs using library functions – `toPut`, `toGet`. Since the `c2b` interface is a vector, the `map` higher-order function is applied.

9a ⟨*interfaces bitNode* 9a⟩≡ (5a)

```
// Interface
interface codeIn     = toPut (ffCodeIn);
interface dataOut    = toGet (ffDataOut);
interface c2b        = map (toPut, vffC2B);
interface b2c        = toGet (ffB2C);

// ------------------------------------------------------------------
```

9b ⟨*wrap up bitNode* 9b⟩≡ (5a)

```
endmodule : mkBitNode
```

6 The Check Node

The `mkCheckNode` module receives the partially decoded code word from the `mkBitNode`. It operates in *slave* mode and processes all inputs in the same manner. The `mkCheckNode` does not keep track of iterations. The `mkCheckNode` provides an interface of type `CheckNode`.

9c ⟨*module definition* 5a⟩+≡ (3a) ◁5a

```
// Core function of the check-node
⟨functions checkNode 11a⟩

//
// Check Node Module definition
(* synthesize *)
module mkCheckNode (CheckNode);
    ⟨state checkNode 10a⟩
    ⟨rules checkNode 10b⟩
    ⟨interfaces checkNode 11b⟩
    ⟨wrap up checkNode 11c⟩
// ------------------------------------------------------------------
```

The input FIFOs `vffB2C` receives the partially processed part of the code word meant for this check-node. The output FIFO `ffC2B` holds the partially decoded code word.

10a ⟨*state checkNode* 10a⟩≡ (9c)

```
// Sub-modules and state
// Input FIFO - code word
Vector #(
     NConnections
   , FIFO #(Symbol)) vffB2C        <- replicateM (mkFIFO);

// Output FIFO - decoded code word
FIFO  #(Symbol)      ffC2B         <- mkFIFO;
```

The rule `rlProcessIteration` executes the actions for process the input from the bit nodes.

- Consume the codeword which is currently in `vffB2C`
- Carry out the computation on the codeword
- Enqueue the result into the `ffC2B` for bit-node processing

10b ⟨*rules checkNode* 10b⟩≡ (9c)

```
// Rules and behaviour
rule rlProcessIteration;
   // get the partial result
   Vector #(NConnections, Symbol) codeIn;
   for (Integer i=0; i<valueOf(NConnections); i=i+1) begin
      codeIn[i] = vffB2C[i].first;
      vffB2C[i].deq;
   end

   // Process the partial result further
   let codeOut = fnCheckNodeProcessing (codeIn);

   // Send the partial result to the bit node
   ffC2B.enq (codeOut);
endrule

// ------------------------------------------------------------------
```

The function `fnCheckNodeProcessing`, carries out the actual bit manipulation of the codeword bits as per the min-sum algorithm. The noinline attribute ensures that a separate verilog module is generated for the function.

11a ⟨*functions checkNode* 11a⟩≡ (9c)

```
(* noinline *)
function Symbol fnCheckNodeProcessing (Vector #(NConnections, Symbol) x);
   // XXX This is at present a dummy function which simply does an
   // inverting of the bits
   // It will actually do some sort of a zip function which goes:
   // Vector#(N,Symbol) -> Symbol
   return (foldl1 (\^ , x));
endfunction
```

Creating the interfaces simply involves stitching up the connections to the input and output FIFOs using library functions – `toPut` and `toGet`. Since the `b2c` interface is a vector, the `map` higher-order function is applied.

11b ⟨*interfaces checkNode* 11b⟩≡ (9c)

```
// Interface
interface c2b = toGet (ffC2B);
interface b2c = map (toPut, vffB2C);

// ----------------------------------------------------------------
```

11c ⟨*wrap up checkNode* 11c⟩≡ (9c)

```
endmodule : mkCheckNode
endpackage
```