EE705 Course Project – LDPC Decoder

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

Final Report: May 2, 2019

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

1 LDPC Background

In communication system normally the transmitter appends parity bits to message bits in order allow the receiver to correct or detect errors. These are called error correcting codes. There is a generator matrix which encapsulates the implementation of the parity check equations :

$$C = M * G$$

Here, C is the coded message, M is the original message and G is the generator matrix. At the receiver side we have the received message bit and the parity check matrix, from which we decode the message.

In LDPC codes there is a property that every code digit is contained in the same no. of equations and each equations contains the same no. of code symbols. LDPC codes are usually represented by the tanner graph. Tanner graph contains two set of vertices:

1. n vertices for the code word bits called a bit nodes.
2. m vertices for the check equation called check node.

1.1 The Parity Matrix

The parity matrix dictates the connections between the bit-nodes and check-nodes and encodes a tanner graph. So, in order to build a hardware LDPC decoder, we have to fix this parity matrix. We have chosen the following matrix which has been provided to us by Mandar J. Datar from the HPC Lab:

$$H = \begin{bmatrix} 1101000 \\ 0110100 \\ 0011010 \\ 0001101 \\ 1000110 \\ 0100011 \\ 1010001 \end{bmatrix}$$

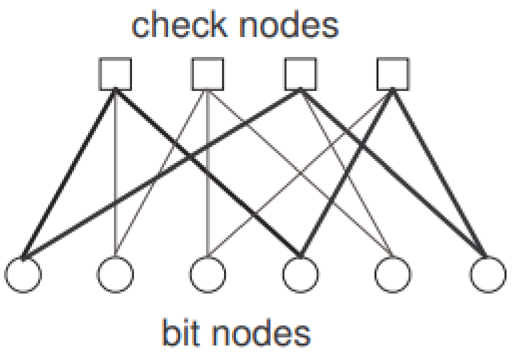


Figure 1: Representation in a tanner graph

A 1 at row i, column j indicates a connection between i^{th} checknode and j^{th} bitnode as show in figure **??**.

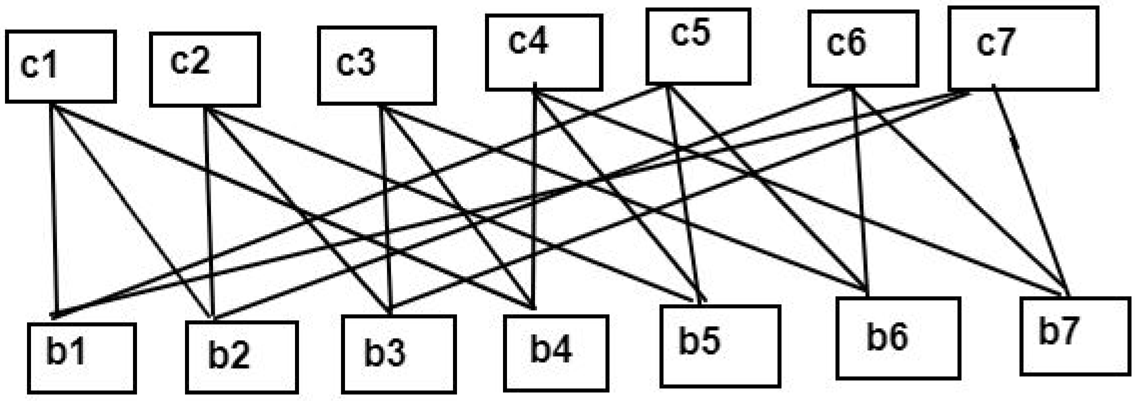


Figure 2: Bitnodes and Checknodes connections based on the given H matrix

1.2 Decoder Algorithm

Here we used the Bit Flipping algorithm, in which the received message bits are decoded according to the parity matrix and if the bit obtained is not correct, then the respective bit is flipped and then again the equations are checked. The bit flip decoder directly gives it output whenever a valid code word has obtained which satisfies all the parity check equation. Here the c_1 means the first bit of coded word is taking the input from the b_1, b_2, b_4, then bit which it will send back to the b_0 is:

$$c_1 = b_1 \oplus b_2 \oplus b_4$$

$$c_2 = b_2 \oplus b_3 \oplus b_5$$

These are sent back to the respective bitnodes. For instance bitnode-1 will receive inputs from checknodes 1, 6 and 7 and so on. Now, the bit assigned to the b1 is the majority of the bits which are send by the c1, c6 and c7.

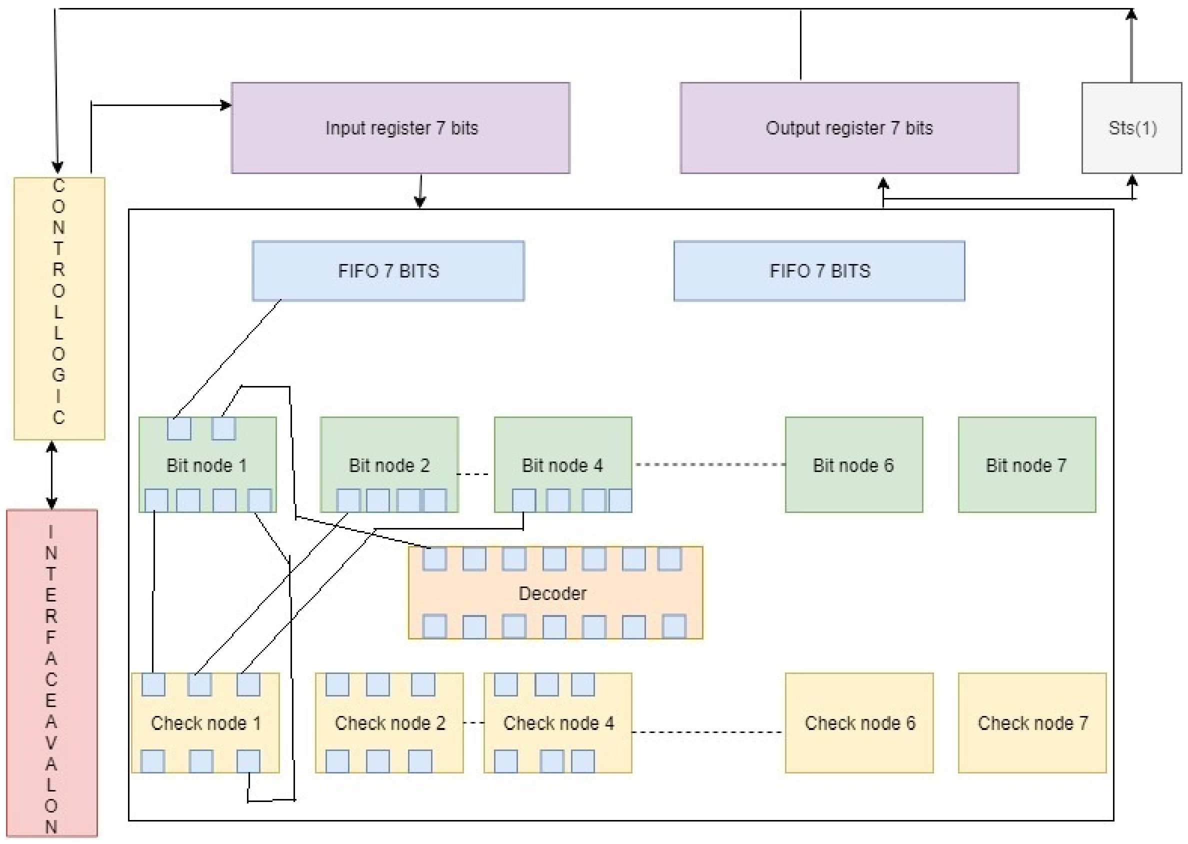
Like this all the bits have one value assigned to them, now to check whether the decoded messages bits are correct or not all the parity equations should be checked using:

$$(b_1 \oplus b_2 \oplus b_4) \vee (b_2 \oplus b3 \oplus b5) \vee (b3 \oplus b4 \oplus b6) \vee (b4 \oplus b5 \oplus b7) \vee (b1 \oplus b5 \oplus b6) \vee (b2 \oplus b6 \oplus b7) \vee (b1 \oplus b3 \oplus b7)$$

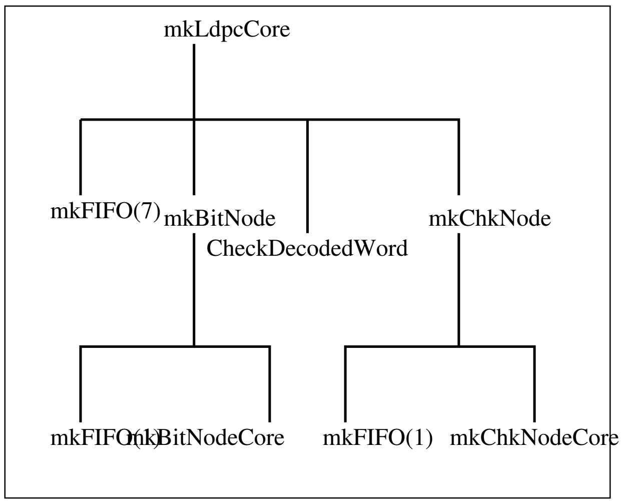
If the output of the above equation is zero means the decoded message bits are correct otherwise, it will again send the bit obtained to the check nodes and again do the calculations until the result of the equation is zero.

2 LDPC Decoder Block Diagram

The LDPC decoder core represents the design which was developed and implemented upto GDSII. The LDPC decoder top-level represents the top-level for functional verification on a DE-Nano FPGA board using Quartus. For verification the LDPC decoder core was integrated as a NIOS peripheral with an Avalon slave interface.



Appendix A describes in detail how the RTL design of the LDPC core and its components were carried out. The design is hierarchical:



3 From RTL to GDSII

In the course of the project we experimented with two different design flows to realise a circuit for the LDPC decoder. The starting point of both flows was after the design had been sufficiently verified through functional simulations as well as by running as an Avalon slave on the DE-Nano board.

3.1 Bottom-up Flow Using Java Electric

Figure **??** captures the steps involved in a bottom-up flow using the Electric tool-flow introduced as part of the course:

The Yosys synthesis tool reads in behavioural RTL and generates a structural netlist, given a cell library. The tool uses the abc tool in the background to actually do the synthesis.

The structural RTL is then converted to structural VHDL. This step usually involved running multiple scripts to get it into just the correct format for input to the Electric tool suite.

In the bottom-up flow we begin at the lowest level of the hierarchy, individually, synthesizing, and then creating layouts and schematic for each module. Each individual module is taken through DRC and LVS.

As the leaf module got bigger, manual schematic entry became problemmatic. So we decided to carry out LVS by comparing the spice netlist dumped from the layout versus the spice netlist created by Yosys.

Java Electric was not able to check DRC on the complete design repeatedly crashing after a few hours of running DRC.

3.1.1 Manual stitching in Electric

We have used electric 9.07 using bottom-up appraoch for creating layout out of the VHDL files. It has created layout for each node with successful design rule check(DRC) and earlier the plan was to make layout for individual module and then make layout manually for the overall LDPC decoder using these individual module. But the layouts for the individual modules were quite big and creating layout for the whole circuit using those big layout became extremely

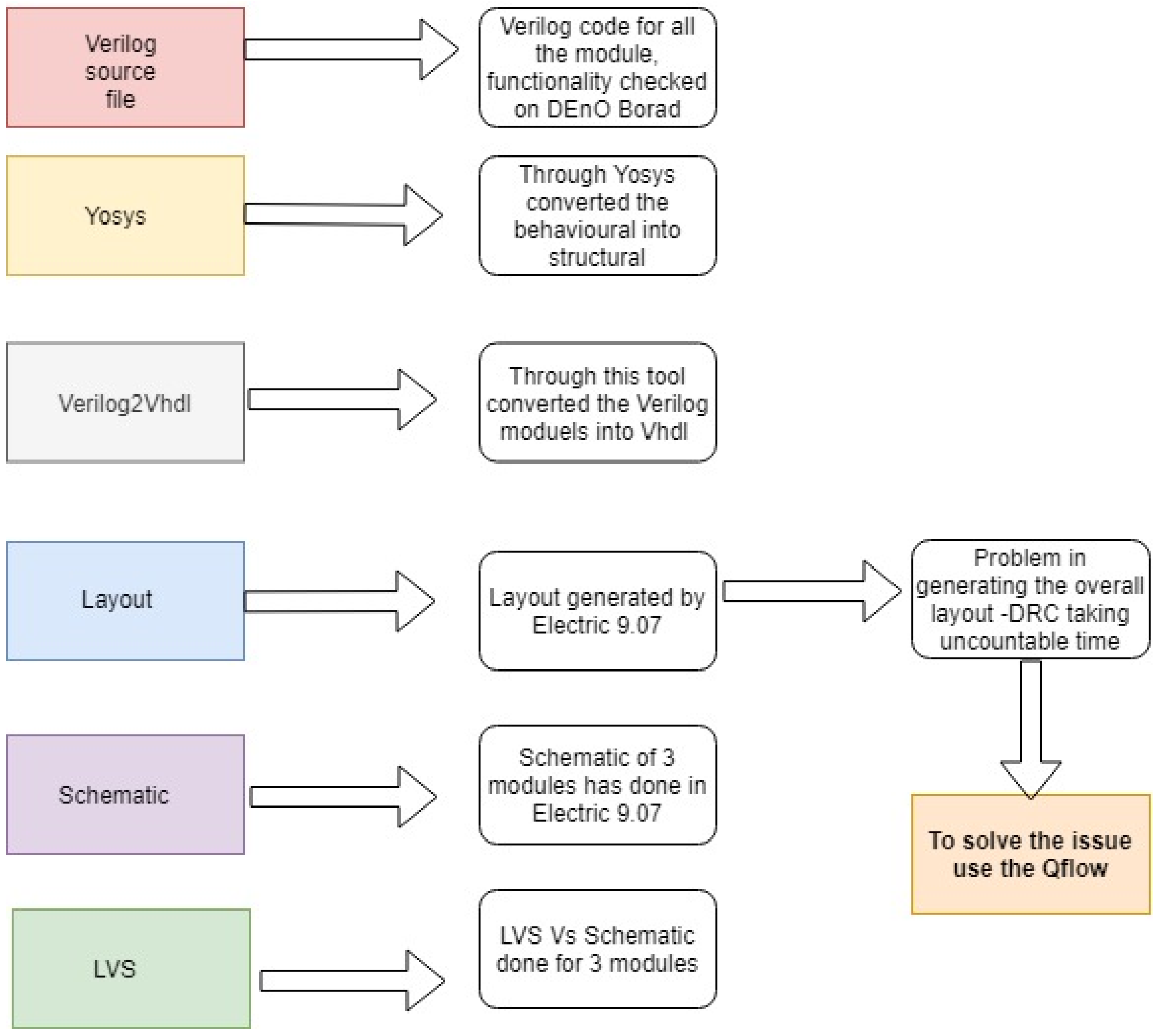

Verilog source file
Verilog code for all the module, functionality checked on DEnO Borad
Yosys
Through Yosys converted the behavioural into structural
Verilog2Vhdl
Through this tool converted the Verilog moduels into Vhdl
Layout
Layout generated by Electric 9.07
Problem in generating the overall layout -DRC taking uncountable time
To solve the issue use the Qflow
Schematic
Schematic of 3 modules has done in Electric 9.07
LVS
LVS Vs Schematic done for 3 modules

Figure 3: Flow using Java Electric

difficult. We are using 7 bitnodes and 7 checknodes and for each node there is an individual layout means 14 layouts only for these bitnodes and checknodes, leaving other nodes aside. This was becoming something which was going out of reach for the mannual layout.

Then we decided to combine all the vhdl code files for respective modules in the vhdl code file of the top level circuit and tried to generate the automatic layout from electric 9.07. We have tried to make changes in the placement settings,routing settings and to use metal layers more than 2 but electric remains unresponsive and creating layout with the default settings. But there was a problem in running DRC for this top level circuit layout because it was taking too long to complete the DRC and still not displaying the results/errors.To solve this problem we decided to shift towards Qflow open circuit design tool.

3.2 Top-down Flow Using Q-Flow

The limitations with DRC on Java Electric led us to explore an alternate design flow. For this we settled on a top-down flow called qflow, which integrates a set of open-source tools. In this flow we have to explicitly do placement and routing - it is not integrated under a single tool like Java Electric. On the other hand, this flow has complete PDKs for 350nm and 180nm processes which makes it possible to generate a GDS-II.

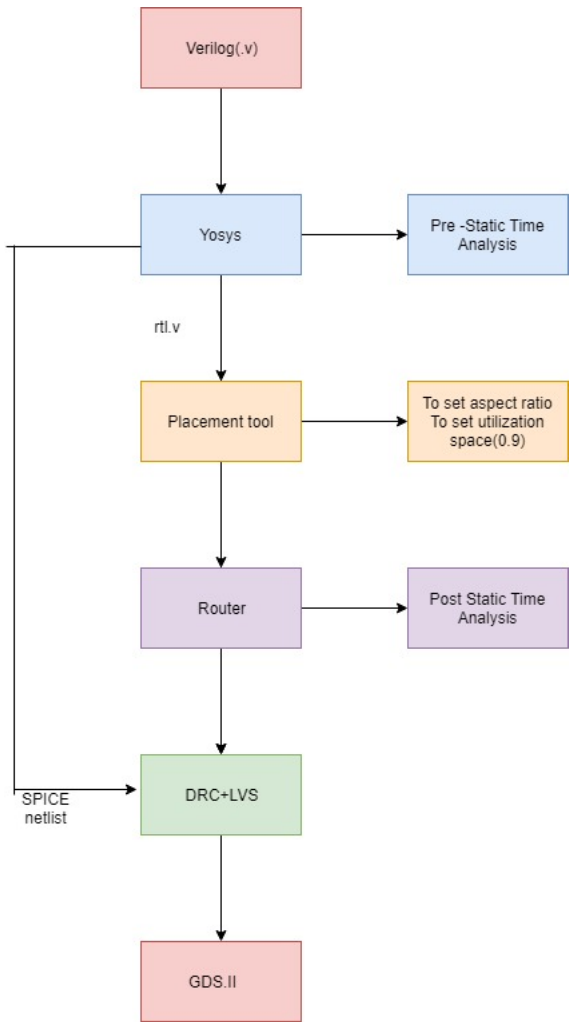


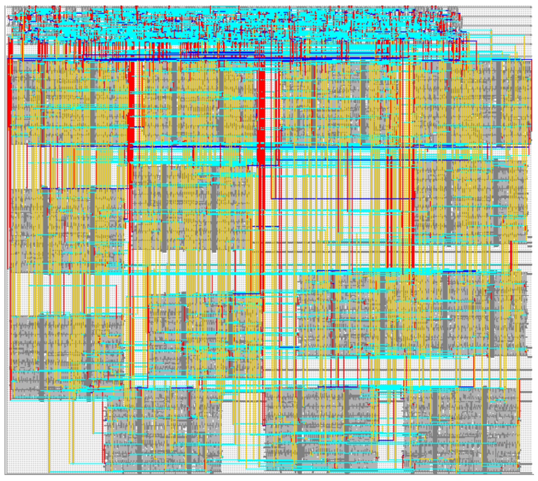
Figure 4: Flow using QFlow

The other big difference from the Electric based flow is the use of a top-down approach where the whole LDPC core is flattened allowing the place and route tool greater flexibility. We also did try a bottom-up approach with this flow, but it resulted in several DRC violations at the top-level which we haven't yet managed to debug.

The QFlow based flow also integrates static timing analysis – something which was missing on the Electric based flow.

4 Results

These results were obtained using OSU's 350nm PDK. A hierarchical layout and flat layout are contrasted on figure **??**. The flat layout is clearly superior in terms of timing and area due to the much higher compaction obtained by automatic layout.


(a) Hierarchical

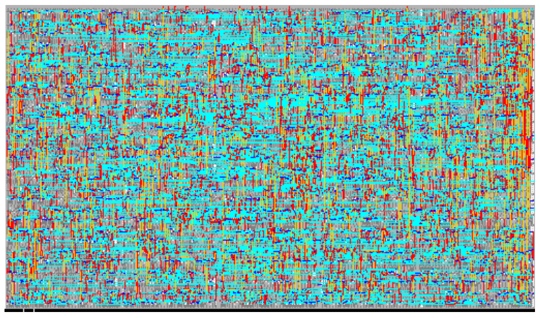

(b) Flat

Figure 5: Routes – Hierarchical and Flat

From our bottom-up synthesis of the bitnode and checknode, the pre and post-layout maximum operating frequencies are:

mkBitNode: 535.41 MHz, 533 MHz

mkChkNode: 481.60 MHz, 479.11 MHz

A 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.

8a ⟨*boilerplate* 8a⟩≡

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

package Nodes;

// ------------------------------------------------------------------
// This package defines:
//
//    ChkNode        : Interface to the Check Nodes
//    BitNode        : Interface to the Bit Nodes
//    mkChkNode      : Micro-arch of the Check Node
//    mkBitNode      : Micro-arch of the Bit Node
//    mkBitNodeCore  : Core computation of the Bit Node
//    mkChkNodeCore  : Core computation of the Check 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* 8b⟩

⟨*module definition* 10a⟩

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).

8b ⟨*type definitions* 8b⟩≡ (8a)

⟨*interface definition* 9a⟩

B 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.

9a ⟨*interface definition* 9a⟩≡ (8b) 9b ▷

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

9b ⟨*interface definition* 9a⟩+≡ (8b) ◁ 9a 9c ▷

```
// 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.

9c ⟨*interface definition* 9a⟩+≡ (8b) ◁ 9b

```

   // 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.

C The Bit Node

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

10a ⟨*module definition* 10a⟩≡ (8a) 13b ▷

```
// Core function of the bitNode
⟨functions bitNode 11b⟩

//
// Bit Node Module definition
(* synthesize *)
module mkBitNode (BitNode);
   ⟨state bitNode 10b⟩
   ⟨rules bitNode 11a⟩
   ⟨interfaces bitNode 12c⟩
⟨wrap up bitNode 13a⟩
// -----------------------------------------------------------------
```

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

10b ⟨*state bitNode* 10b⟩≡ (10a) 10c ▷

```
// 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 and this completes one iteration of processing.

10c ⟨*state bitNode* 10b⟩+≡ (10a) ◁ 10b

```
// 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.

11a ⟨*rules bitNode* 11a⟩≡ (10a) 12a ▷

```
// Rules and behaviour

// Rule to process new data received from the top-level
rule rlProcessNewData;
   // 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;

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

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

The function `fnBitNodeCore`, 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. In fact this module was developed entirely in verilog, and then merged with this module later.

11b ⟨*functions bitNode* 11b⟩≡ (10a)

```
(* noinline *)
function Symbol fnBitNodeCore (Vector #(NConnections, Symbol) i);
   return ((i[1]&i[2])|(i[0]&i[2])|(i[0]&i[1]));
endfunction
```

The rule `rlProcessChkNodeResult` executes the actions on receiving a response from the check node. 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 as per `fnBitNodeCore`
- Enqueue the result into the `ffDataOut` for global checks at the top-level

12a ⟨*rules bitNode* 11a⟩+≡ (10a) ◁11a

```
⟨mutex pragma 12b⟩
// Rule to process remaining iterations
rule rlProcessChkNodeResult;
   // 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

   // Send the processed code word to the output
   ffDataOut.enq (fnBitNodeCore (codeIn));
endrule

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

The `rlProcessChkNodeResult` rule will only run when there is a partial result from the check nodes. On the other hand the `rlProcessNewData` will only run when there is new data input from the top-level. These two rules are mutually exclusive in a non-pipelined implementation such as this one. The mutex pragma specifies this as an assertion to the compiler.

12b ⟨*mutex pragma* 12b⟩≡ (12a)

```
(* mutually_exclusive = "rlProcessChkNodeResult, rlProcessNewData" *)
```

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.

12c ⟨*interfaces bitNode* 12c⟩≡ (10a)

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

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

13a ⟨*wrap up bitNode* 13a⟩≡ (10a)
```
  endmodule : mkBitNode
```

D The Check Node

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

13b ⟨*module definition* 10a⟩+≡ (8a) ◁10a
```
  // Core function of the check-node
  ⟨functions checkNode 14b⟩

  //
  // Check Node Module definition
  (* synthesize *)
  module mkChkNode (ChkNode);
      ⟨state checkNode 13c⟩
      ⟨rules checkNode 14a⟩
      ⟨interfaces checkNode 15a⟩
      ⟨wrap up checkNode 15b⟩
  // -----------------------------------------------------------------
```

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.

13c ⟨*state checkNode* 13c⟩≡ (13b)
```
  // Sub-modules and state
  // Input FIFO - code word
  Vector #(
       NConnections
     , FIFO #(Symbol)) vffB2C        <- replicateM (mkFIFO);

  // Output FIFO - decoded code word
  Vector #(
       NConnections
     , FIFO #(Symbol)) vffC2B        <- replicateM (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

14a ⟨*rules checkNode* 14a⟩≡ (13b)

```
// 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

   // Send the partial result to the bit node
   let res = fnChkNodeCore (codeIn);
   for (Integer i=0; i<valueOf(NConnections); i=i+1)
      vffC2B[i].enq (res[i]);
endrule

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

The function `fnChkNodeCore`, 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.

14b ⟨*functions checkNode* 14b⟩≡ (13b)

```
(* noinline *)
function Vector#(NConnections, Symbol) fnChkNodeCore (Vector #(NConnections, Symbol) i);
   Vector #(NConnections, Symbol) o = newVector;
   o[0]=i[1]^i[2];
   o[1]=i[0]^i[2];
   o[2]=i[0]^i[1];
   return (o);
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.

15a ⟨*interfaces checkNode* 15a⟩≡ (13b)

```
// Interface
interface c2b = map (toGet, vffC2B);
interface b2c = map (toPut, vffB2C);

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

15b ⟨*wrap up checkNode* 15b⟩≡ (13b)

```
endmodule : mkChkNode
endpackage
```