Error Correction Encoder & Decoder

Digital Design and Logical Synthesis for Electric Computer Engineering (36113611)

Digital High-Level Design

Version 0.1

Revision Log

Rev	Change	Description	Reason for change	Done By	Date
0.1	Part 1	First part of the project			
0.2					
0.3					

Contents

Table of Contents

List of figures

1. BLOCKS RTL DESCRIPTION

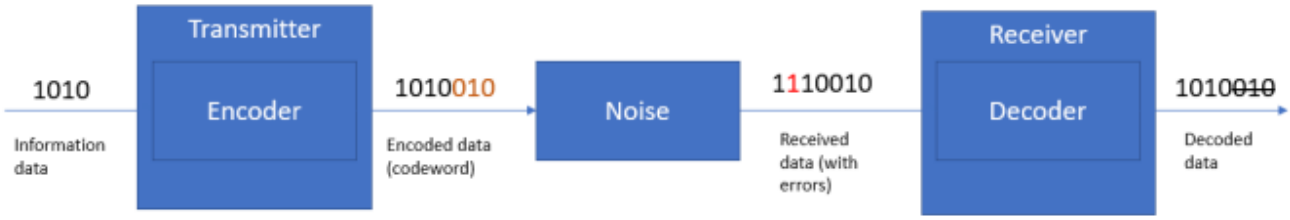

Figure 1: Error Correction System

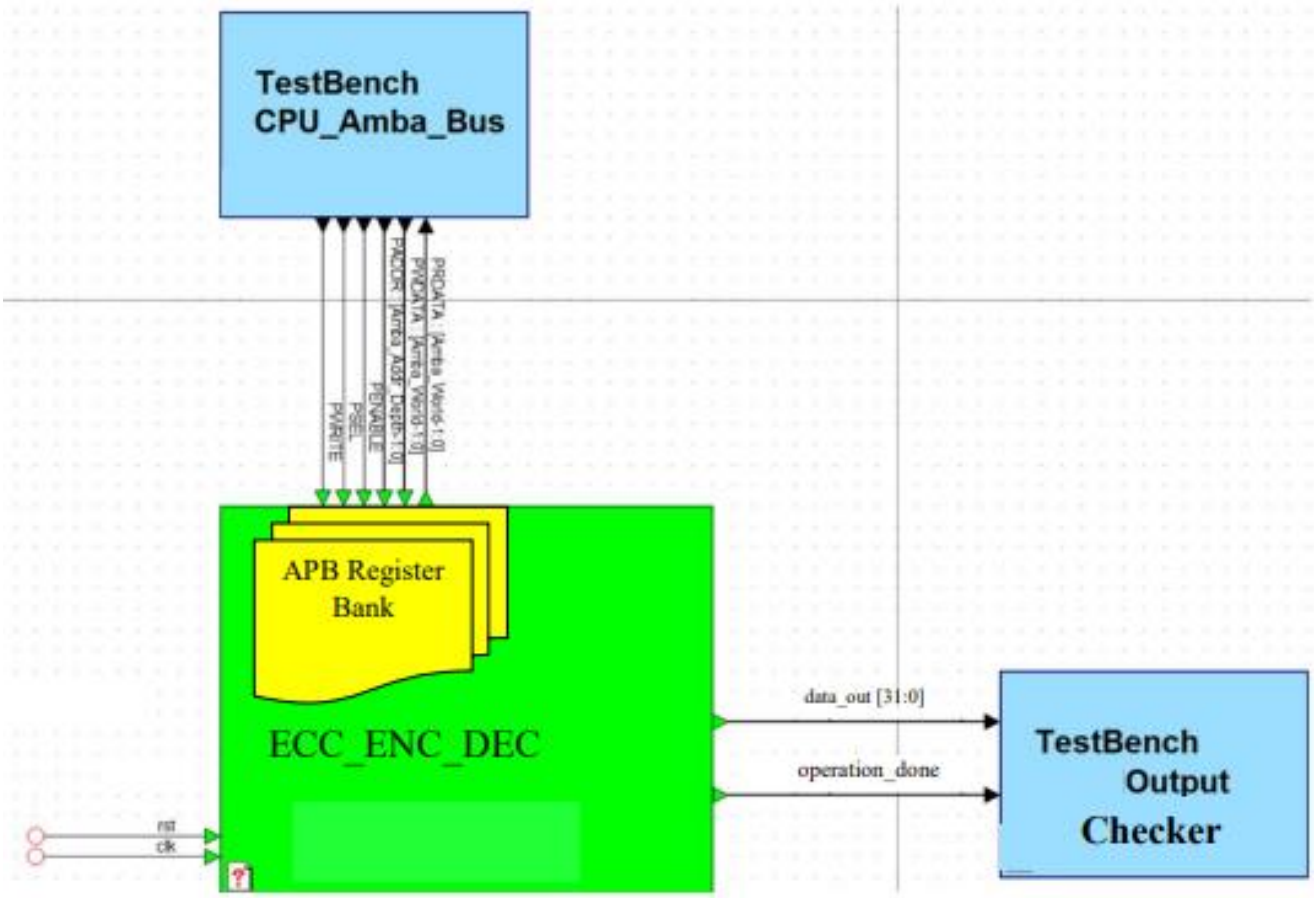

Figure 2: Top View of the Design

In this part we implemented the architectural block that handles encoding and decoding, as well as the APB registers bank, following the AMBA protocol, to enable communication between the CPU/memory and the main block.

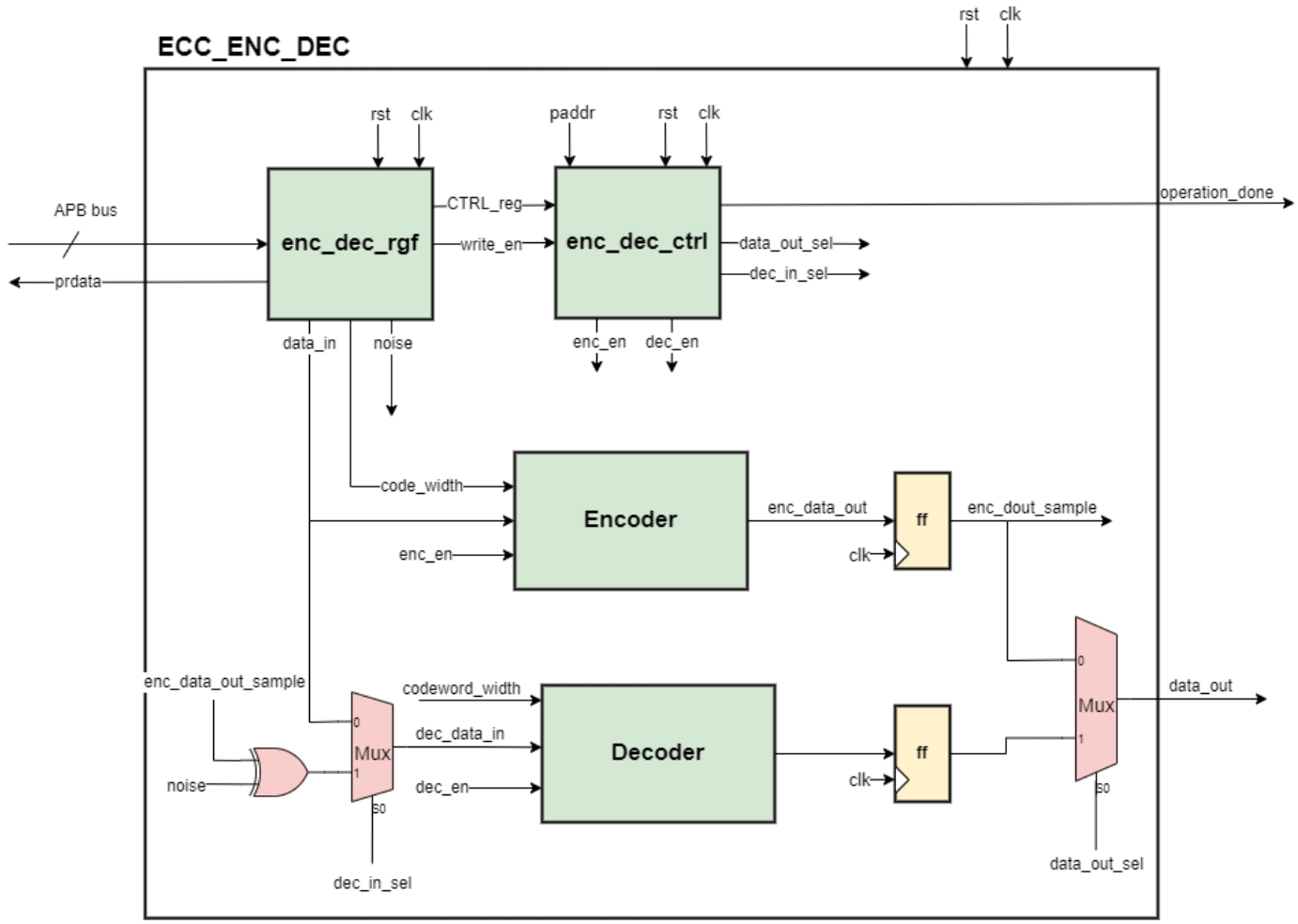

Figure 3: RTL view of the system with inputs and outputs of the blocks

1.1 ecc_enc_dec.v

The main module of the system. This module is controlled by the CPU through the APB Register Bank from within it. In this stage of the project, the test bench is replacing the CPU.

This module is designed using mixed approach: We first took the specification and broke it down into blocks and sub-blocks (Top-Down), and then designed optimized circuits for leaf-level cells. Using these cells we built higher-level cells (Bottom-Up).

The module includes the following sub-modules:

- **enc_dec_ctrl.sv**

  A module which handles current operation done and knows when the output is ready for reading.

- **enc_dec_rgf.sv**
  A module which keeps the APB protocol register files
- **encoder.sv**
  A module which handles encoding.
- **decoder.sv**
  A module which handles encoding.

The module also includes a register at the output of the encoder and the decoder as well as a few MUXes for data flow control.

It is important to note that this module does not calculate anything on its own but rather leaves the work for the sub-modules. Also an active-low reset is connected to every register in the system.

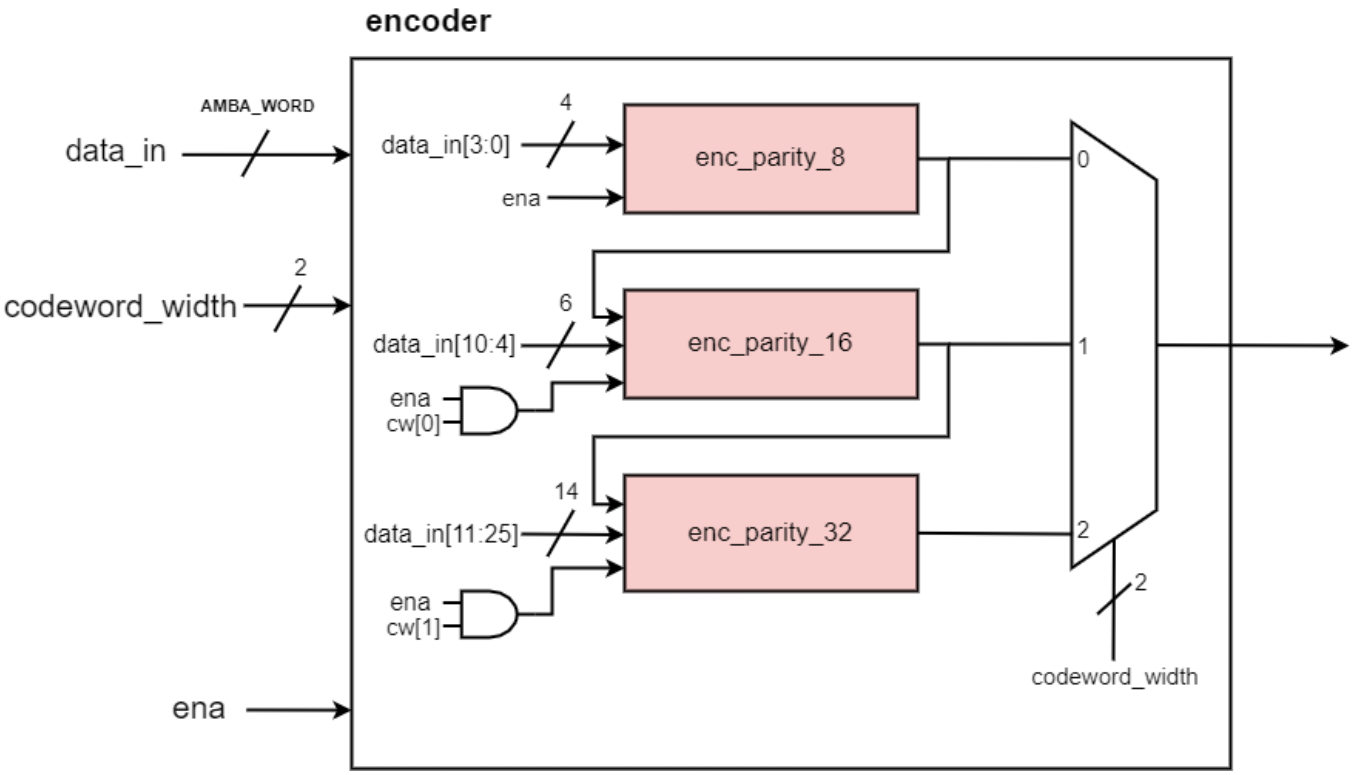

Figure 4: encoder block overview

1.2 encoder.sv

This is one of the sub-modules of the main module. “encoder.sv” is responsible for the calculations of a codeword given a word.

It takes the given word and uses a sub-module to calculate the parity as shown in the algorithm in app.1.

The sub-module used is decided by the value in the input codeword_width:

- 00 => use the 8-bit parity calculation module.
- 01 => use the 16-bit parity calculation module.
- 10 => use the 32-bit parity calculation module.

The encoder's output is a legal codeword according to the algorithm.

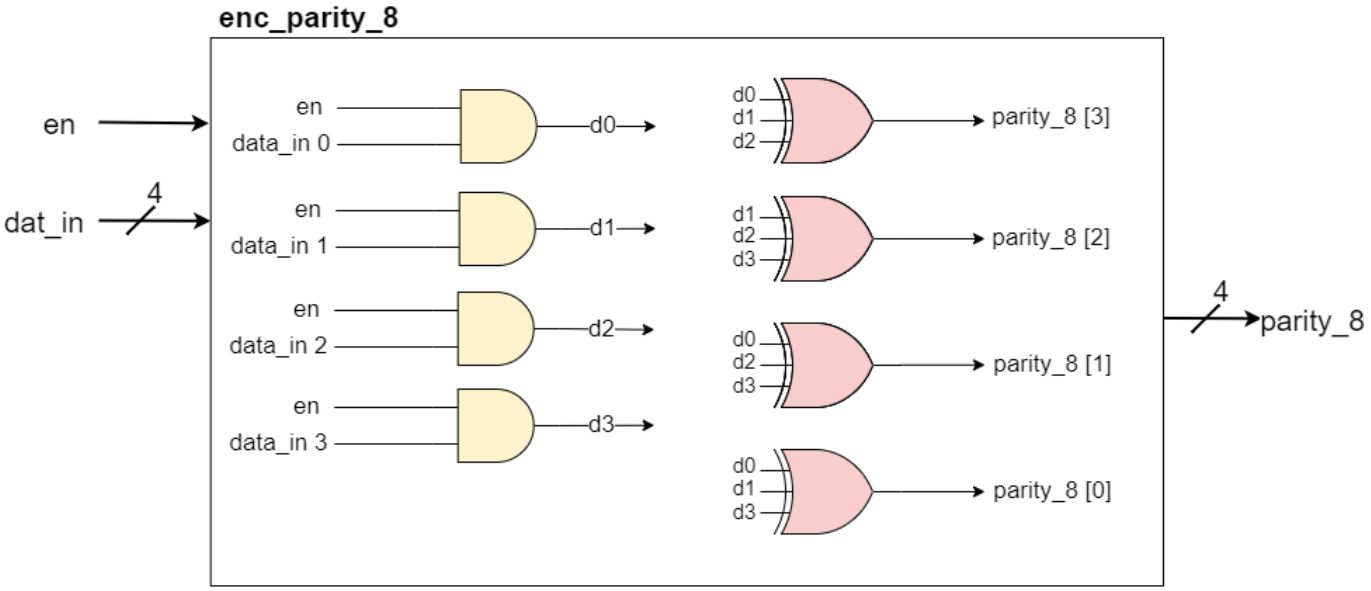

Figure 5: RTL view of enc_parity_8

1.3 enc_parity_8.sv

As an example, we show a detailed description of the parity calculation sub-module which handles the calculation of the 4-bit parity from a 4-bit word. The other parity calculation sub-modules work the same way.

It is noted in the specifications that the matrices we use all have right parts which are upper-triangular sub-matrices:

$$\boldsymbol{H} = \left(\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right)$$

Given a 4-bit word, we want the 8-bit codeword $\boldsymbol{c}$ (which includes the parity bits) to uphold $Hc^T = 0$ when $\boldsymbol{H}$ is the parity calculation matrix. Therefore the parity calculation comes down to a list of equations:

$$c_8 = c_1 + c_3 + c_4$$
$$c_7 = c_1 + c_2 + c_4$$
$$c_6 = c_1 + c_2 + c_3$$
$$c_5 = c_1 + c_2 + c_3 + c_4 + c_6 + c_7 + c_8$$

Where we know c_1, c_2, c_3, c_4 and we want to calculate c_5, c_6, c_7, c_8.

We noticed that the calculation for c_5 can be shorter by substituting c_6, c_7, c_8 for their c_1, c_2, c_3, c_4 calculation and by removing a xor between similar signals as it has no effect on the output.

$$c_5 = c_1 + c_2 + c_3 + c_4 + c_6 + c_7 + c_8 = c_1 + c_2 + c_3 + c_4 + c_1 + c_2 + c_3 + c_1 + c_2 + c_4 + c_1 + c_3 + c_4$$
$$= c_2 + c_3 + c_4$$

After this reduction, the implementation is a system of XOR gates between the different calculated inputs.

1.4 decoder.sv

This is one of the sub-modules of the main module. “decoder.sv” is responsible for finding out if a given codeword has 2, 1 or 0 errors, and decoding the codeword into a word if it has 1 or 0 errors.

It includes the following sub-modules:

- **dec_mat_multiplier_all_options.sv**
  This sub-module handles matrix multiplication according to the decoding algorithm. It is implemented in a similar manner as the parity calculation modules: the matrix multiplicaiton was not reduces this time as it was not possible, but still implemented using XORs.
  Given a codeword $\boldsymbol{y}$ it outputs its multiplication with the algorithm matrix $\boldsymbol{H}$. let us mark the output by $\boldsymbol{S}$.

- **dec_is_column.sv**
  Given $\boldsymbol{S}$, check if it is a column in the matrix. If it is a column in the matrix, output also which column it is.
  This is done using a series of comparators which compare **S** in parallel, and with a simple encoder to find out which column is equal.

- **dec_comparator.sv**
  Another single comparator is used to find out if $\boldsymbol{S}$ is all zeros, in which case the codeword has no errors.

- **dec_output_ctrl.sv**
  This module receives the results from the comparing modules as well as the original codeword. It decides whether the codeword has 0, 1 or 2 errors and outputs the according expected results. In the case of 1 error it uses the sub-module **dec_flip_a_bit.sv** which takes a codeword and the noise index to fix (indicated by the output from **dec_is_column.sv**), and outputs the fixed codeword.
  This module's output is the original word padded by zeros if it is possible to decode it, and the amount of errors found.

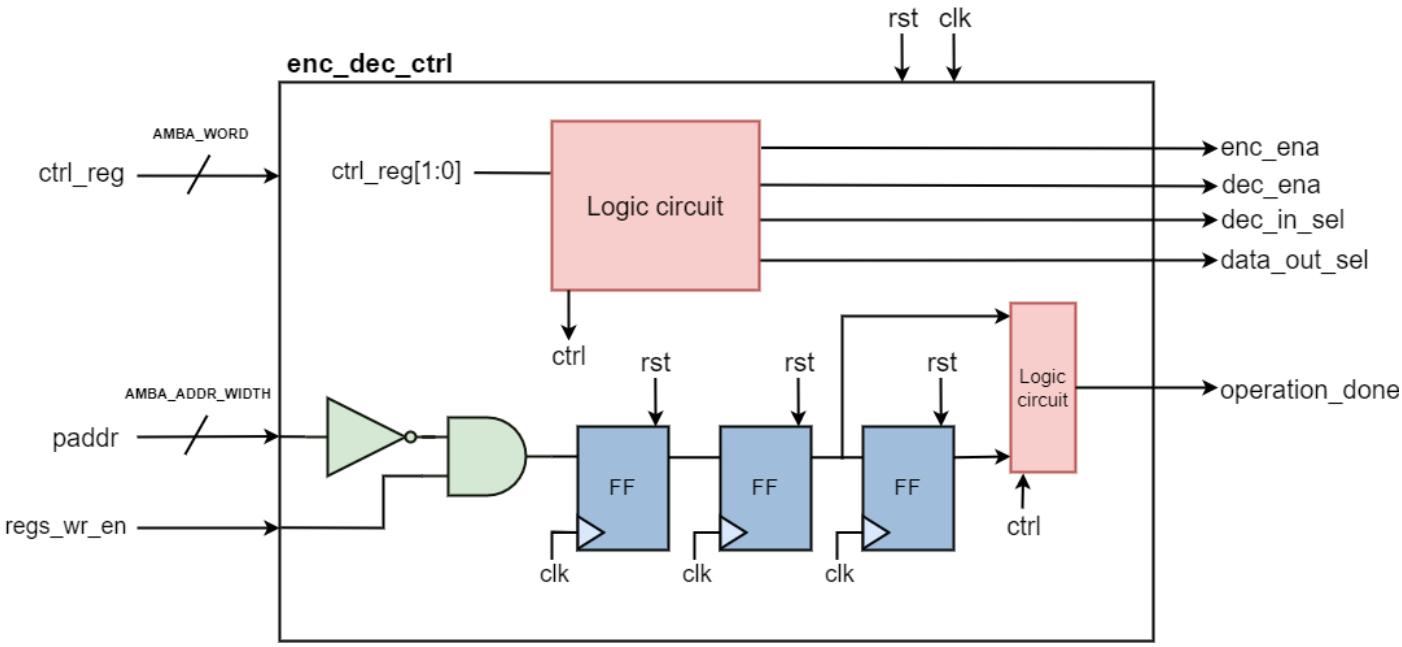

Figure 6: RTL overview of enc_dec_ctrl

1.5 enc_dec_ctrl.sv

This module is the data flow control unit. It uses the given required work to be done, and enables either the encoder or the decoder or both. It also waits for the amount of cycles needed to finish the operation:

- For **Encode** operation it waits just one cycle.
- For **Decode** operation it also waits on cycle.
- For **Full Channel** operation it waits for two cycles.

While designing this module we assumed that the combinatoric calculations done in the encoder or the decoder take no longer than one CPU cycle.

The logic circuit used after the flip flops in Figure6: RTL overview of enc_dec_ctrl could be removed, in which case the ecc_enc_dec would always finish it's operation after two cycles. We decided that the amount of hardware used to build this logic circuit is minor and preferred to improve time for one channel operation.

1.6 enc_dec_rgf.sv

The CPU controls the design through the APB Register Bank which is in this module. The APB is composed of a few registers (each of them of size AMBA_WORD). The CPU has read/write access to every register using the APB protocol.