By: Tom Kessous – 206018749

Dan Ben Ami - 316333079

Error Correction Encoder & Decoder

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

ECC ENC DEC

Digital High Level Design

Version 0.1

Table of Content

LIST OF TABLES

1. BLOCKS FUNCTIONAL DESCRIPTIONS

This module is error correction encoder & decoder, which communicate the CPU via APB bus. Our module can identify up to 2 errors (bits flipped) in a word and correct 1 error in a word. There are 3 operations modes to the system: Encode, Decode and Full channel.

- In the **encode** operation the input data contain uncoded word, the module will encode it (add parity bits) and put the codeword result on the output.
- In the **decode** operation, the input data contain a codeword with up to two errors. The module will decode the data and output the num of errors, If the num of errors was 0 or 1 the module will output the corrected data.
- In the **full channel** operation, the input data will encode in the Encoder module. From there, it will move to the Noise Adder module which add noise to it according to the data from the noise register. Afterwards, the noisy codeword will decoded in the decoder module. The module output the num of errors, and if there are less then two errors the module will also output the corrected data.

Our module designed to be generic in term of data width and APB bus width. The module is written in system Verilog and designed to be fast as possible while consuming low power.

1.1.1 Top level ECC ENC DEC

The top level entity include all the entities descript in the figure below in blue and connect all of them. Moreover it contains the two mux that control the operation mode of the module.

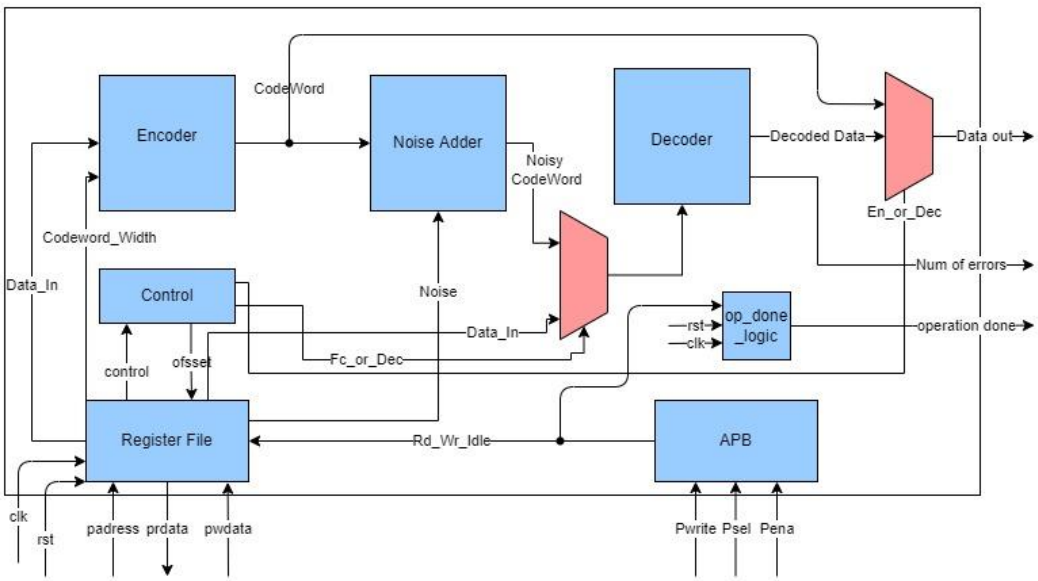

Figure 1: Top level design

Name	Mode	Type	Bounds	Delay	comment
PADDR	Input	Logic	[AMBA_ADDE_WIDTH-1:0]		APB Address bus
PENABLE	Input	Logic			APB Bus Enable/clk
PSEL	Input	Logic			APB Bus Select
PWDATA	Input	Logic	[AMBA_WORD-1:0]		APB Write Data Bus
PWRITE	Input	Logic			APB Bus Write
clk	Input	Logic			system clock
rst	input	Logic			Asynchronous Reset active low
PRDATA	Output	Logic	[AMBA_WORD-1:0]		APB Read Data Bus
Data_out	Output	Logic	[DATA_WIDTH-1:0]	1	Encoded/Decoded data (Valid when operation_done is asserted)
Operation_done	Output	Logic			Indicates an operation is finished. (Pulse , asserts for one cycle)
Num_of_errors	Output	Logic	[1:0]		numer of bit errors after decode operation (valid only after decode/full channel operations)

Table 1: Top level interface.

1.1.2 Encoder

Encoder unit receive a word to encode (Data_In) from the register file unit and encode the word according to DATA_WIDTH parameter and Codeword_Width. The calculation of the parity bits is made by modules encode_H1, encode_H2 and encode_H3 each module compute the parity bits with respect to matrices H1,H2,H3. The output of the encoder unit is CodeWord in size 8,16,32 depend on Codeword_Width.

Name	Mode	bounds	comment
Data_In	In	[DATA_WIDTH-1:0]	Word to encode
CodeWord_Width	In	[1:0]	Size of the encoded word
CodeWord	Out	[DATA_WIDTH-1:0]	The word with parity bits

Table 2: Encoder interface.

1.1.3 Decoder

Decoder unit receive a CodeWord from the mux that choose the CodeWord base on the current task of the system. If the current task is "decode" then the input to the decoder is Data_In from register file, Else the current task of the system is "Full Channel" then the input to the decoder is NoisyCodeWord.

The decoder build from few entities:

- Mat_mult entity which it self build from 3 entities:
    - Mat_mult_1 in case of DATA_WIDTH parameter is 8, responsible to calculate H1 matrix multiply by codeword vector in length of 8 bits.
    - Mat_mult_2 in case of DATA_WIDTH parameter is 16, responsible to calculate H1 and H2 matrix multiply by codeword vector in length of 8 or 16 bits depend on Codeword_Width.
    - Mat_mult_3 in case of DATA_WIDTH parameter is 32, responsible to calculate H1 and H2 and H3 matrix multiply by codeword vector in length of 8 or 16 or 32 bits depend on Codeword_Width.

- Mux_H1 entity output the decoded data and num of errors based on the mat_mult result from H1 matrix.
- Mux_H2 entity output the decoded data and num of errors based on the mat_mult result from H2 matrix.
- Mux_H3 entity output the decoded data and num of errors based on the mat_mult result from H3 matrix.

The decoder can fix one bit flipped (one error in the CodeWord) and recognize two bits flipped (two errors in the CodeWord). There are two outputs from the decoder: the number of errors in the noisy CodeWord, decoded data.

The decoded data from the decoder can be one of the above:

a. the word without the parity bits, in case there are no errors.

b. the fixed word in case of one error.

c. zeros in case of two errors.

Name	Mode	bounds	comment
NoisyCodeWord	In	[DATA_WIDTH-1:0]	CodeWord with errors
Codeword_Width	In	[1:0]	Indicate the Size of the CodeWord
Decoded_Data	Out	[DATA_WIDTH-1:0]	The fixed word
NumOfErrors	out	[1:0]	Number of errors

Table 3: Decoder interface.

1.1.4 Noise Adder

This unit receive Noise from the register file and receive CodeWord from the encoder. Then add (bitwise xor operation) the noise to the CodeWord. The output is the CodeWord with noise namely NoisyCodeWord.

Name	Mode	bounds	comment
noise	In	[DATA_WIDTH -1:0]	Noise from register file
CodeWord	In	[DATA_WIDTH -1:0]	CodeWord from encoder
NoisyCodeWord	Out	[DATA_WIDTH-1:0]	CodeWord with noise

Table 4: Noise Adder interface.

1.1.5 Control

The control unit generate control signals according to the value in the Control register. There are two control signals, Full channel/decode (namely FC_or_Dec) and encoder/decoder (namely En_or_Dec). The first one control the input data to decoder unit via mux, The second control the output of the system via mux.

Name	Mode	bounds	comment
CTRL	In	[1:0]	Value of control register
FC_or_Dec	out		Control data input of decoder
En_or_Dec	Out		Control system output

Table 5: Control interface.

1.1.6 Register File

Register file have 4 registers. In case of reset occurs we write zeros to all registers. In case of write operation the data is written to the desired register according to the address. In case of read operation we access the desired register based on the address and output its content.

Name	Mode	bounds	comment
clk	In		System clock
rst	In		Asynchronous Reset active low
Rd_Wr_Id	In	[1:0]	Operation select: read/write/idle
offset	In	[AMBA_ADDR_WIDTH-3:0]	Address of the register
Data_to_reg	In	[AMBA_WORD-1:0]	The data to write in the register
Data_out	Out	[AMBA_WORD-1:0]	The read data from register
AMBA_WORD_registers[1:0]	Out	[AMBA_WORD - 1:0]	Data & Noise registers
two_bits_registers[1:0]	Out	[1:0]	Control & CodeWord width registers

Table 6: Register file interface.

1.1.7 APB

APB unit receive pwrite, psel and penable and generate Rd_Wr_Id signal which indicate to the register file which operation to execute. If the Rd_Wr_Id is 0 then read operation will execute, If the Rd_Wr_Id is 1 then write operation will execute and If the Rd_Wr_Id is 2 then no read nor write operation will execute (idle state).

Name	Mode	bounds	comment
pwrite	In		APB protocol signals
psel	In		APB protocol signals
penable	In		APB protocol signals
Rd_Wr_Id	Out	[1:0]	Signal that indicate which operation (write,read or idle)

Table 7: APB interface.

1.1.8 Op_done_logic

This unit responsible to generate the operation done signal. Operation done rise to '1' for 1 cycle after the data_out is valid.

Name	Mode	bounds	comment
clk	In		System clock
rst	In		Asynchronous Reset active low
Rd_Wr_Id	In	[1:0]	Signal that indicate which operation (write,read or idle)
Operation done	Out		Rise to '1' for one cycle after the data_out is valid

Table 8: Op_done_logic interface.

1.2 Flow Chart

The flow chart describe the data flow in the system.

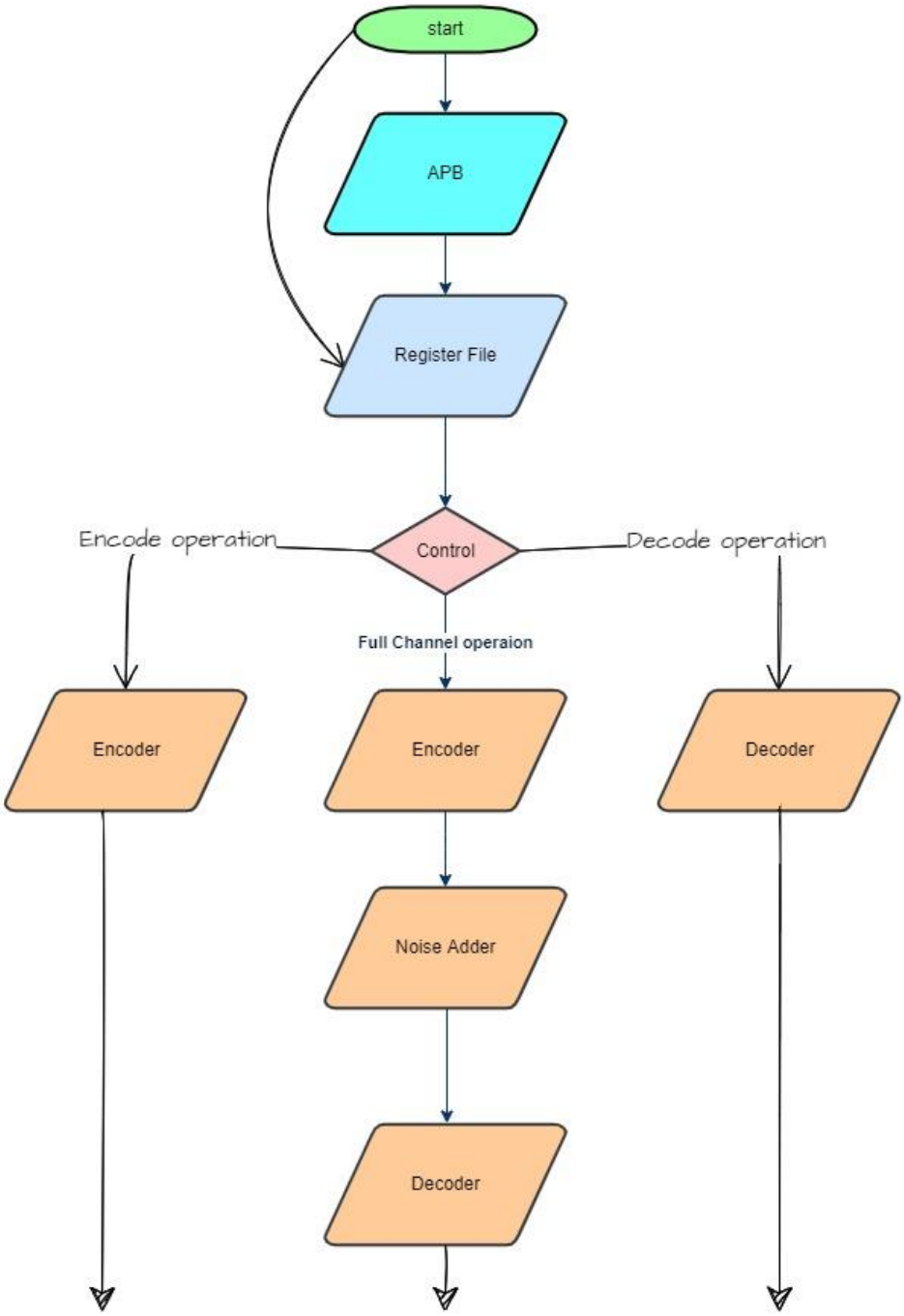

Figure 2: Flow Chart

2. WAVED RULES – DESIGN CHECKER

We waved the following error/warnings:

- Encoder:
  - In case DATA_WIDTH=32:
    - "Input port Data_IN[31:26] is never used" (1)
  - In case DATA_WIDTH=16:
    - "Input port Data_IN[15:11] is never used" (1)
    - "codeword3 is never used" (1)
  - In case DATA_WIDTH=8:
    - "Input port DATA_IN[7:4] is never used" (1)
    - "Input port 'Codeword_Width' is never used" (1)
    - "CodeWord1 is never used" (1)
    - "CodeWord2 is never used" (1)
    - "CodeWord3 is never used" (1)
- Mat_mult:
  - In case DATA_WIDTH=8: "input port 'Codeword_width' is never used" (1)
- Decoder:
  - In case DATA_WIDTH=16:
    - "colums[5] is never used" (1)
    - "Decoded_Data3 is never used" (1)
    - "Decoded_Data_full_length 3 is never assigned" (1)
  - In case DATA_WIDTH=8:
    - "colums[5:4] is never used" (1)
    - "Decoded_Data2 is never used" (1)
    - "Decoded_Data3 is never used" (1)
    - "Decoded_Data_full_length 2 is never assigned" (1)
    - "Decoded_Data_full_length 3 is never assigned" (1)
    - "input Codeword_width is never used" (1)

- Op_done_logic:
  - "Asynchronous reset 'op_done_logic.rst' is not synchronized before being used".(2)
- Register file:
  - "Asynchronous reset register_file.rst' is not synchronized before being used".(2)
- Ecc_enc_dec:
  - some signals are never used in case when DATA_WIDTH change. (1)
  - "Asynchronous reset register_file.rst' is not synchronized before being used".(2)

1. Warnings with type "never used" logic variables are caused from the fact that our code is generic. Some of our main concerns are scalability, genericity and reused of the code. therefore, we designed the code for generating different hardware in case of different parameters. As a result, some of the variables (logics) are unused in some of the cases. furthermore, some of the "never used" warnings were as a result of the project's instructions (to set the input's width too long or the addresses etc.).

2. Warnings " Asynchronous reset 'moudule.rst' is not synchronized before being used" caused due to project instructions, The reset should be asynchronous by definition.

3. DESIGN OBJECTIVES

This module designed to be fast as possible while consuming low power. Our module is designed with combinatorial logic except of the register file which must contain flip flops. Due to the combinatorial design the valid output is available one cycle after writing data to the register. This is the minimal cycle latency that we could achieve, because just the write and then read data from the register file take one cycle so overall one cycle latency is minimal. Although we get minimal cycle latency we think that the maximal clock frequency could be better if we would implement pipelined architecture. The critical path of the combinatorial logic will be relatively long so the maximum frequency will be relatively small. We faced a tradeoff of low cycle latency vs. clock frequency. Another advantage of combinatorial logic is relatively low power consuming comparing sequential logic that use flip flops.