Automated ECC Design Tool (AEDT)

Tool Motivation:

This tool provides a solution to the design of custom Error Correction Codes (ECCs) for memory protection. Normally, ECC design can be done using algebraic equations or computer search algorithms. When the former method is used, the algorithms and programs used are develop ad-hoc and do not support the design of general ECCs. In AEDT, a computer searching algorithm method is selected to be the kernel of the tool construction. This code design technique is based on the binary linear block codes theory can be seen as the Boolean satisfiability problem and solved by the use of the recursive backtracing algorithm. However, the traditional searching program is just applicable to the specific ECC requirement limited by the systematic structure, correction ability, parity bits and data bits length. If these parameters change, the searching program needs to be specially designed and updated. This poses a barrier for the ECC designer and limits its ability to design codes for specific cases. To overcome those issues, we developed the AEDT to help the ECC designers obtain what they want with the least effort and provide various ECC functions.

Tool Function Description:

To simplify the whole ECC design process, the complex searching algorithm is hidden to users and only the parameters of the desired code are needed. To that end, users need to define them in a file named Set.txt file and provide three input parameters of the program:

In more detail, the Set.txt file contains:

(1) Error patterns

Here, the users need define each error pattern to be corrected. If single errors should be corrected, the error pattern should be 1 0 0 0 0 0 0 0 0 0; If double adjacent errors should be corrected, the error pattern should be 1 1 0 0 0 0 0 0 0 0; If double almost adjacent errors should be corrected, the error pattern should be 1 0 1 0 0 0 0 0 0 0.

Each pattern is put in a different line and the maximum default length of the error pattern is ten.

(2) Error pattern range

Here, users need define the effective range of each error pattern. The bits of the codeword from left to right are numbered from 1 to the length of the codeword. Each error pattern range can be set by using two values, start bit and end bit and the order of the error pattern should correspond to the order of the error patterns defined in (1). To support the function of different correction ability in partial bits, this tool provides the opportunity of selecting five different blocks of partial bits. For example, for the (7, 4) Hamming code, the error pattern is 1 0 0 0 0 0 0 0 0 0 (Single Error Correction), the length of the codeword is 7, the error pattern range should be set as 1 11 0 0 0 0 0 0 0 0; If the single error correction ability is not applied to all the bits, the error pattern range can be set as 1 3 5 6 8 10 0 0 0 0 where bits 1-3 5-6 8-10 have the single error correction ability. One thing to be noticed is that the error pattern range should larger than the error pattern size. For example, if the error pattern is 1 0 1 0 0 0 0 0 0 0, then the size of the error pattern is 3, the error pattern cannot be 1-2 or 3-4, should be at least 1-3 or 4-6.

(3) Parity bit positions:

In some cases, the parity bits need to be placed in some specific positions to get the best performance. Here, the users can select the parity bit position in the word. Users need to provide the bit number of the parity bits. From left to right, the bit number is from low to high.

To run the program in addition to the Set.txt file we need to provide the following parameters:

(1) *Number of data bits*

The length of the data bits.

(2) Number of parity bits

The length of the parity bits.

(3) Number of the error patterns

The number of the error pattern to be corrected.

(4) Running time of the program

The time (second) of the program to run.

For example, to construct the SEC-DAEC code, if data bits are 16, parity bits are 6, the number of the error pattern is 3 (1,11,101) and program running time is 18000 second (5 hours) then execute the command ./Auto_ECC 16 6 3 18000
and the Set.txt file would be:

```
******************
1 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0
******************
1 22 0 0 0 0 0 0 0 0
1 22 0 0 0 0 0 0 0 0
1 22 0 0 0 0 0 0 0 0
******************
1 2 3 4 5 6
```

For example, to construct the non systematic code that can correct single error and double adjacent errors. The bit placement is *d p d p d p d p d p d p d d d d d d d d d d* (*d* is data bit, *p* is parity bit). if data bits are 16, parity bits are 6, the number of the error pattern is 3 (1,11,101) and program running time is 18000 second (5 hours) then execute the command ./Auto_ECC 16 6 3 18000
and the Set.txt file would be:

```
******************
1 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0
******************
1 22 0 0 0 0 0 0 0 0
1 22 0 0 0 0 0 0 0 0
1 22 0 0 0 0 0 0 0 0
******************
```

2 4 6 8 10 12

For example, to construct the non systematic code that can correct single error and double adjacent errors on only data bits. The bit placement is *d d d p p p d d d p p p d d d d d d d d d* (*d* is data bit, *p* is parity bit). if data bits are 16, parity bits are 6, the number of the error pattern is 3 (1,11,101) and program running time is 18000 second (5 hours) then execute the command ./Auto_ECC 16 6 3 18000
and the Set.txt file would be:

****
1 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0
****
1 3 7 9 10 22 0 0 0 0
1 3 7 9 10 22 0 0 0 0
1 3 7 9 10 22 0 0 0 0
****
4 5 6 10 11 12

For example, to construct the non systematic code that can correct single error and double adjacent errors on partial data bits. The bit placement is *d d d p p p d d d p p p d d d d d d d d d* (*d* is data bit, *p* is parity bit, Yellow area is SEC-DAEC, Green area is just 11 and 101, Purple area is SEC). if data bits are 16, parity bits are 6, the number of the error pattern is 3 (1,11,101) and program running time is 18000 second (5 hours) then execute the command ./Auto_ECC 16 6 3 18000
and the Set.txt file would be:

****
1 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0
****
1 3 10 22 0 0 0 0 0 0
7 9 10 22 0 0 0 0 0 0
7 9 0 0 0 0 0 0 0 0
****
4 5 6 10 11 12