# AnalyzingMARCH

November 25, 2021

# 1 MARCH Tests

## 1.1 March C-nv-SRAM

In this example, we are investigating a 128kx8 SRAM that was exposed to radiation in March C & D tests. Hence, the content inside the memory was continuosly written and red. At any rate, from the point of view of statistical analysisis, this test is equivalent to a pseudostatic one, with two different patterns and with addresses in switching order.

## 1.2 Loading packages

The very first thing we must do is to load the packages required to load files (*DelimitedFiles*) as well as the LELAPE module. I suppose you have installed both. Load is done with:

[4]:
```
push!(LOAD_PATH,"/home/francisco/Escritorio/LELAPE-main/LELAPE/src")
using DelimitedFiles, LELAPE
```

## 1.3 Defining variables

Previous paragraph allows us to define several variables for checking the tests:

- Word width : 8 bits
- Memory size in words: 2M is just 2^21.
- In SRAMs, it seems more likely to succeed the XOR operation.
- Tests were pseudostatic. Therefore, it is intelligent to keep information about the different cycles.

Ok, let us use this information to set these variables:

[5]:
```
LA = 2^17 # Memory size in words
WordWidth = 8 # Selfexplaining.
Operation = "XOR" # Only "XOR" or "POS" are allowed.
KeepCycles = true # This is a Bool variable and only true false are accepted.
```

[5]:
```
true
```

## 1.4 Loading data

Results are stored in three different files following the required format: * CSV files * Every row is formed as WORD ADDRESS, READ VALUE, PATTERN, CYCLE. Besides, the first row contains column heading (must be skipped), separators are commas and EOL character is the standard.

We will use the *readdlm* function provided by the *DelimitedFiles* package to load the first CSV file and to store everything in the new variable, DATA. Finally, it is important to indicate that DATA must be an array of UInt32 numbers.

```
[7]: DATA1 = readdlm("March C-nv-SRAM.csv", ',', UInt32, '\n', skipstart=1)
```

```
[7]: 429×4 Matrix{UInt32}:
      0x00000536  0x00000004  0x00000000  0x00000001
      0x00000e66  0x00000040  0x00000000  0x00000001
      0x00003588  0x00000080  0x00000000  0x00000001
      0x0000362d  0x00000010  0x00000000  0x00000001
      0x00005611  0x00000080  0x00000000  0x00000001
      0x000063bc  0x00000002  0x00000000  0x00000001
      0x000065d6  0x00000002  0x00000000  0x00000001
      0x000095e9  0x00000010  0x00000000  0x00000001
      0x0000c6ca  0x00000010  0x00000000  0x00000001
      0x0000d161  0x00000002  0x00000000  0x00000001
      0x0000d629  0x00000001  0x00000000  0x00000001
      0x0000d9f7  0x00000080  0x00000000  0x00000001
      0x0000ec57  0x00000001  0x00000000  0x00000001

      0x0001a815  0x00000020  0x00000000  0x0000000a
      0x0001a8dc  0x00000010  0x00000000  0x0000000a
      0x0001ad05  0x00000040  0x00000000  0x0000000a
      0x0001afb1  0x00000004  0x00000000  0x0000000a
      0x0001bcc4  0x00000004  0x00000000  0x0000000a
      0x0001cca5  0x00000020  0x00000000  0x0000000a
      0x0001d95b  0x00000001  0x00000000  0x0000000a
      0x0001e15e  0x00000008  0x00000000  0x0000000a
      0x0001e467  0x00000080  0x00000000  0x0000000a
      0x0001e8c3  0x00000010  0x00000000  0x0000000a
      0x0001fbbc  0x00000002  0x00000000  0x0000000a
      0x0001fcdc  0x00000010  0x00000000  0x0000000a
```

## 1.5 Looking for MBUs

This analyisis is quite simple. We will call the *CheckMBUs* functionthat returns the MBUs present in DATA.Input arguments are the second and third columns, and the wordwidth.

This function returns two vectors. The first one indicates in position $k$ the number of bitflips observed in the *kth* word. The second one is a vector of vectors and contains more detailed information: not only the number of bitflips per word but the position of the flipped bit (0 = LSB, WordWidth-1 = MSB).

```
[8]: MBUSize, MBU_bit_pos = CheckMBUs(DATA1[:,2], DATA1[:,3], WordWidth)
```

```
[8]: ([1, 1, 1, 1, 1, 1, 1, 1, 1, 1  …  1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Any[[2], [6],
[7], [4], [7], [1], [1], [4], [4], [1]  …  [6], [2], [2], [5], [0], [3], [7],
[4], [1], [4]])
```

The following loop will show how many MBUs per number of flipped bits were observed:

```
[9]: for size = 1: WordWidth
         println("$size-bit MBUs: ", length(findall(MBUSize.==size)))
     end
```

```
1-bit MBUs: 429
2-bit MBUs: 0
3-bit MBUs: 0
4-bit MBUs: 0
5-bit MBUs: 0
6-bit MBUs: 0
7-bit MBUs: 0
8-bit MBUs: 0
```

## 1.6 Looking for MCUs

As modern memories are interleaved, it is not worth investigating MBUs but MCUs. Now, the system will combine addresses in all the possible pairs and operate them to create a DV set. If there were no MCUs, their characteristics are known.

In particular, we can state that if the expected number of elements repeated $k$ times in this set is lower than a very low positive number, it is impossible to observe this number of repetitions unless the Only SBU assumption fails. We will define this threshold as 0.001 (default, 0.05).

Although without a solid theoretical background, it seems that using pseudoaddress instead of word address provides better results.

Some experiments seem to show that if an element with very few number of 1s in binary format is too often repeated, it is indicative of the presence of MCUs. This is the Trace Rule and, in our analyisis, we want to keep all those too often repeated elements such that contain 2 ones or less in binary format.

Finally, perhaps we know that MCUs will not very large. For example, we may guess that MCUs with more than 20 bitflips are totally rejected. Therefore, to help the software and to avoid running out of memory, we will say the program *"Don't be silly and do not expect events larger than 20!!"* If somehow this idea was wrong, we can change this value again and repeat the calculations.

```
[10]:  = 0.001    # If the expected number of elements repeated k times is lower than␣
     ↪ ,
                 # we can afirm that this is virtually impossible.
     UsePseudoAddress = true
     TraceRuleLength = 2
     LargestMCUSize = 20
```

```
[10]: 20
```

Time to test!!! We will call the function. Deppending on the set size or even if this is your first test, it will take you more or less time (Don't get up from your chair, though!!!!)

The following instruction will look for: 1. Values that pass the self-consistency test (C1_SCY) 2. Values found after inspecting MCUs derived from self-consistency-test (C1_MCU). 3. Values with less than or equal to *TraceRuleLength* 1s in binary format that appear too often in the DV set (C1_TRC). 4. Values that, after combining in pairs the union of all the previous three sets and applying the operation and that appear too many times within the DV set (C1_SHF).

The first column of each matrix are the possible values and the second one the times it appeared.

[11]:
```
C1_SCY, C1_MCU, C1_TRC, C1_SHF = DetectAnomalies_FullCheck(DATA1, WordWidth,␣
↪LA, Operation, TraceRuleLength, UsePseudoAddress, KeepCycles,  ,␣
↪LargestMCUSize)
```

[11]:
```
(Matrix{UInt32}(undef, 0, 2), Matrix{UInt32}(undef, 0, 2), Matrix{UInt32}(undef,
0, 2), Matrix{UInt32}(undef, 0, 2))
```

Perhaps these matrices are hard to read since, for efficiency, they were returned in UInt32 format, even the number of occurrences!!! Execute the following instrucction for a better comprehension.

[12]:
```
println("Elements appearing more than expected and passing the Self-Consistency␣
↪test:\n")
for index in 1:length(C1_SCY[:, 1])
    println("Value: 0x", string(C1_SCY[index, 1], base=16, pad = 6), " --> ",␣
↪Int(C1_SCY[index, 2]),".")
end

UsePseudoAddress ? L = LA*WordWidth : L = LA

print("\nOnly up to ", MaxExpectedRepetitions(NPairs(DATA1, UsePseudoAddress,␣
↪WordWidth, KeepCycles), L, Operation,  )-1, " repetitions are explained by␣
↪randomness.")
```

```
Elements appearing more than expected and passing the Self-Consistency test:


Only up to 4 repetitions are explained by randomness.
```

In this example, it is not worth to check the other sets since they did not yield any positive result. If you had had success, you would only have to do the following:

[13]:
```
C1_All = [C1_SCY; C1_MCU; C1_TRC; C1_SHF]
```

[13]:
```
0×2 Matrix{UInt32}
```

### 1.7 Grouping bitflips

Now, we have discovered those values relating pairs of pseudoaddresses. Now, let us go to group events in DATA.

The first step consists in labeling all the pseudoaddresses and grouping their assigned indexes to a matrix containing information for the possible MCUs. It is an intermediate step and is done with the instruction *MCU_Indexes* with the required and already defined parameters.

```
[14]: Labeled_addresses = MCU_Indexes(DATA1, Operation, C1_All[:, 1],␣
       ↪UsePseudoAddress, WordWidth)
```

```
[14]: 0×2 Matrix{Int64}
```

Using this information, we can group the addresses.

```
[15]: Events = Classify_Addresses_in_MCU(DATA1, Labeled_addresses, UsePseudoAddress,␣
       ↪WordWidth)
```

```
[15]: 2-element Vector{Any}:
       0×2 Matrix{UInt32}
       UInt32[0x000029b2, 0x00007336, 0x0001ac47, 0x0001b16c, 0x0002b08f, 0x00031de1,
      0x00032eb1, 0x0004af4c, 0x00063654, 0x00068b09  …  0x000d682e, 0x000d7d8a,
      0x000de622, 0x000e652d, 0x000ecad8, 0x000f0af3, 0x000f233f, 0x000f461c,
      0x000fdde1, 0x000fe6e4]
```

---

Difficult to read, isn't it? The following instruction makes the content more readable:

```
[16]: for k = 1:length(Events)
          NMCUs = length(Events[k][:, 1])
          println("Pseudoaddresses involved in $(length(Events)-k+1)-bit MCUs ($NMCUs␣
       ↪events):")
          for row = 1:NMCUs
              for bit = 1:length(Events)-k+1
                  print("0x", string(Events[k][row, bit], base=16, pad = 6), )

                  bit != length(Events)-k+1 ? print(", ") : print("\n")

              end
          end
          println()
      end
```

```
Pseudoaddresses involved in 2-bit MCUs (0 events):

Pseudoaddresses involved in 1-bit MCUs (429 events):
0x0029b2
0x007336
0x01ac47
0x01b16c
0x02b08f
0x031de1
```

```
0x032eb1
0x04af4c
0x063654
0x068b09
0x06b148
0x06cfbf
0x0762b8
0x078795
0x07ceec
0x082482
0x0829af
0x08d10e
0x0945be
0x096016
0x09789d
0x09bd00
0x0b13e6
0x0b197d
0x0b550b
0x0b703b
0x0b7e2b
0x0b918a
0x0be79b
0x0cdd6c
0x0d71e7
0x0da1f5
0x0e44d0
0x0ed88a
0x0f357c
0x0f3afc
0x0f424b
0x0fe99e
0x0ff398
0x0036c3
0x003a44
0x0144be
0x015819
0x016120
0x01ce04
0x01fcc6
0x026c15
0x02926b
0x02a099
0x035549
0x038e12
0x03b13d
0x04b8d1
0x04c359
```

```
0x058ae5
0x05adfd
0x05bac4
0x063803
0x063e34
0x0652a3
0x067bb9
0x06b13d
0x06cd51
0x06fd7d
0x076cda
0x07c0cf
0x07c28a
0x07e9c0
0x09120b
0x0abbfd
0x0ad4f5
0x0add44
0x0c1530
0x0c3dbd
0x0c7f97
0x0cbff7
0x0cc5d0
0x0cdce1
0x0cf786
0x0d1288
0x0db2e9
0x0dbb73
0x0dfe84
0x0e18c1
0x0e80ef
0x0e8797
0x0ead21
0x0facc6
0x0fad2b
0x0fe13a
0x0deb01
0x0d620a
0x0cb1eb
0x0bba52
0x0b77f6
0x0a372a
0x08c5c5
0x0859f7
0x082b89
0x08111e
0x07da1f
0x0759a2
```

```
0x06730c
0x0585ec
0x050377
0x04fceb
0x04d85f
0x04986f
0x04836a
0x046cf8
0x044719
0x042d44
0x041e5f
0x041b0f
0x03c83f
0x02fc77
0x02e560
0x02b9f3
0x02b9e3
0x02838b
0x027ad1
0x0265ef
0x02644f
0x022318
0x01dbb3
0x01dac9
0x01c69a
0x01b7fe
0x017fb3
0x017a11
0x01658e
0x01654c
0x01539f
0x013a61
0x012407
0x01075e
0x00ceb0
0x00cac1
0x008654
0x0036ec
0x0fa545
0x0fa218
0x0f9884
0x0f8fee
0x0f5ad5
0x0efe5d
0x0eb4c2
0x0e8572
0x0e7501
0x0dd41f
```

```
0x0d7edd
0x0d1e78
0x0c851e
0x0c2b89
0x0bb343
0x0b1d3b
0x09d989
0x09ce1f
0x096c26
0x0874b4
0x084ff9
0x0813d3
0x07eb1e
0x07a141
0x067771
0x05f3c2
0x05db07
0x059a3b
0x057a53
0x0547f8
0x05425f
0x04597b
0x033166
0x02f99d
0x02f156
0x0213bb
0x01fa22
0x01cd4c
0x015054
0x00bdd4
0x00bbc6
0x001ad9
0x02eb3c
0x02f7db
0x030b62
0x03b51b
0x059800
0x0665e5
0x077dfd
0x077eb0
0x0821f2
0x086aa9
0x09088f
0x097d78
0x0a8b9f
0x0b2e70
0x0c5589
0x0cae20
```

```
0x0caf0e
0x0cb070
0x0cbf93
0x0cc6a1
0x0d1f2c
0x0d2804
0x0d8c5e
0x0df60e
0x0e1f3b
0x0e5899
0x0e5fa1
0x0e9899
0x0effed
0x0f0df3
0x0f7a45
0x0f9e91
0x0322fb
0x032a89
0x03eab0
0x05424a
0x05c370
0x062437
0x06b42f
0x06f3ab
0x0711f6
0x0748af
0x0876c2
0x0923df
0x09264d
0x09c074
0x0a71f9
0x0a8063
0x0ad92d
0x0b4797
0x0b8797
0x0cafdb
0x0d9a3a
0x0dadc5
0x0db73e
0x0dd779
0x0de48a
0x0e2d1e
0x0ea420
0x0f024a
0x0f424a
0x0fd808
0x0fead7
0x003194
```

```
0x005bc8
0x007005
0x0146ba
0x018c38
0x018e0a
0x019fcf
0x01a3f4
0x01c4b6
0x01d026
0x01e3f4
0x022473
0x024548
0x02b47b
0x02fd17
0x0370e3
0x038845
0x03ad52
0x03b123
0x055e48
0x057397
0x05b2a8
0x063c4c
0x0659c0
0x06bf7f
0x06ef09
0x0764ee
0x0772e4
0x077358
0x07752a
0x07a4d7
0x07cf9f
0x081512
0x08341f
0x086cf2
0x08b4af
0x096e77
0x09a572
0x09d4a2
0x0a0809
0x0a2ca9
0x0a6e62
0x0ac59a
0x0accd8
0x0b1330
0x0b3985
0x0b3d88
0x0b6687
0x0cdaab
```

```
0x0cf7a3
0x0d394b
0x0dfef3
0x0e3f62
0x0e81cb
0x0e8adb
0x0f13b9
0x0f4d44
0x0f5563
0x0fa1d7
0x0fc7c0
0x0fccb7
0x0fd36a
0x0cdb9b
0x0c1c62
0x0af8c9
0x0acd97
0x0aac59
0x0a987f
0x0a4a1c
0x09ed2a
0x09dd4b
0x091621
0x08234e
0x07b1c8
0x076ca7
0x072de4
0x068c63
0x05eb85
0x05e8b0
0x056d02
0x052023
0x04bc76
0x04bc56
0x04bc46
0x040360
0x03bf3e
0x039dc3
0x039ceb
0x0382be
0x032c7d
0x0307af
0x02cea1
0x02be7b
0x025ede
0x02439c
0x023cd7
0x01d8d3
```

```
0x019463
0x013f0a
0x013b64
0x0114a5
0x010895
0x010225
0x00df8f
0x00abd0
0x009a15
0x00895b
0x001dd7
0x00102e
0x0fb286
0x0f931e
0x0f92f2
0x0f52f2
0x0f5241
0x0ebcf5
0x0e269f
0x0e2270
0x0db0ec
0x0da607
0x0c8882
0x0c593f
0x0ba6a3
0x0ab4f4
0x0a74ea
0x096ab3
0x07fea4
0x0751cd
0x0708fe
0x06abb8
0x068ea1
0x05b597
0x05a73c
0x052562
0x04cf4f
0x049cee
0x040e86
0x03d6ee
0x03352d
0x030950
0x02d05b
0x025708
0x023d32
0x0233f8
0x01d50f
0x01b7a8
```

```
0x00c735
0x0088c5
0x00877e
0x003e4c
0x0341fc
0x03f5ca
0x056502
0x05c30f
0x080e6f
0x08326e
0x083b1c
0x0889f2
0x089e6f
0x0a2f49
0x0a85a5
0x0accab
0x0af307
0x0b774f
0x0ba2cb
0x0ba571
0x0bc2ad
0x0be76f
0x0bf8ae
0x0c6a8b
0x0c8289
0x0ca6a1
0x0cbd6a
0x0d40ad
0x0d46e4
0x0d682e
0x0d7d8a
0x0de622
0x0e652d
0x0ecad8
0x0f0af3
0x0f233f
0x0f461c
0x0fdde1
0x0fe6e4
```

## 1.8 Analysis completed!