ECC in CVA6-Ariane processor

Sergio García Esteban
Asier Fernández de Lecea Navarro

What we’ll talk about today

What we'll talk about today

- Fundamentals of Error Correction Codes (ECC)
  - Parity, Hamming codes, Hamming + parity, [Hsiao70] paper on error correction

What we'll talk about today

- Fundamentals of Error Correction Codes (ECC)
  - Parity, Hamming codes, Hamming + parity, [Hsiao70] paper on error correction
- Modules for implementing and testing ECC functionalities
  - Encoder, decoder, fault injector…

What we'll talk about today

- Fundamentals of Error Correction Codes (ECC)
  - Parity, Hamming codes, Hamming + parity, [Hsiao70] paper on error correction
- Modules for implementing and testing ECC functionalities
  - Encoder, decoder, fault injector…
- Where ECC modules should go in your processor
  - What to do and where to put them in your processor

What we'll talk about today

- Fundamentals of Error Correction Codes (ECC)
  - Parity, Hamming codes, Hamming + parity, [Hsiao70] paper on error correction
- Modules for implementing and testing ECC functionalities
  - Encoder, decoder, fault injector…
- Where ECC modules should go in your processor
  - What to do and where to put them in your processor
- Implementation details: a perilous and headache-ridden path
  - Final account of what we have managed to accomplish

Fundamentals of Error Correction Codes (ECC)

The main idea being error correction codes is this:

Fundamentals of Error Correction Codes (ECC)

The main idea being error correction codes is this:

- You have some important data in memory that is susceptible to corruption (strong radiation environments, for example)

Fundamentals of Error Correction Codes (ECC)

The main idea being error correction codes is this:

- You have some important data in memory that is susceptible to corruption (strong radiation environments, for example)
- The integrity of data in memory is essential for programs to run correctly (obviously)

Fundamentals of Error Correction Codes (ECC)

The main idea being error correction codes is this:

- You have some important data in memory that is susceptible to corruption (strong radiation environments, for example)
- The integrity of data in memory is essential for programs to run correctly (obviously)
- If that’s the case, how does one (1) detect errors in data words and (2) correct them?

Fundamentals of Error Correction Codes (ECC)

Quick and dirty explanation:

Fundamentals of Error Correction Codes (ECC)

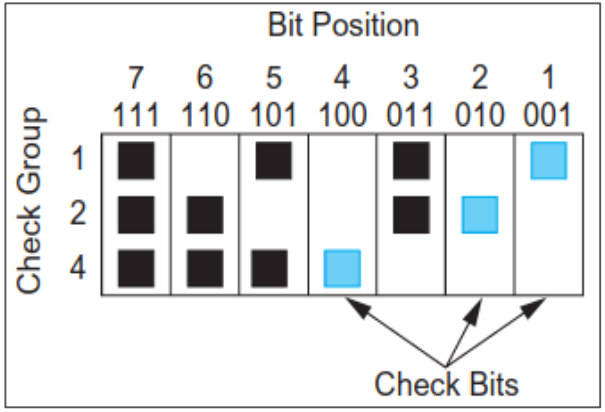
Quick and dirty explanation:

- Group different bits of a data word (of some length N) and keep track of their parity value, aka check bits

Fundamentals of Error Correction Codes (ECC)

Quick and dirty explanation:

- Group different bits of a data word (of some length N) and keep track of their parity value, aka check bits

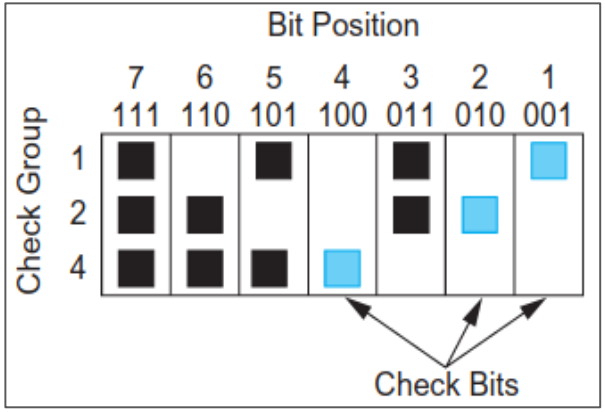


Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

Quick and dirty explanation:

- Group different bits of a data word (of some length N) and keep track of their parity value, aka check bits
- Tracking the parity of different groups of bits simply consists of XOR-ing those bits



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

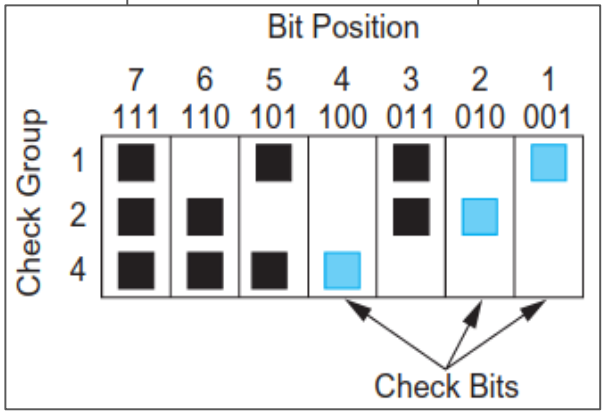
Quick and dirty explanation:

- Group different bits of a data word (of some length N) and keep track of their parity value, aka check bits
- Tracking the parity of different groups of bits simply consists of XOR-ing those bits

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

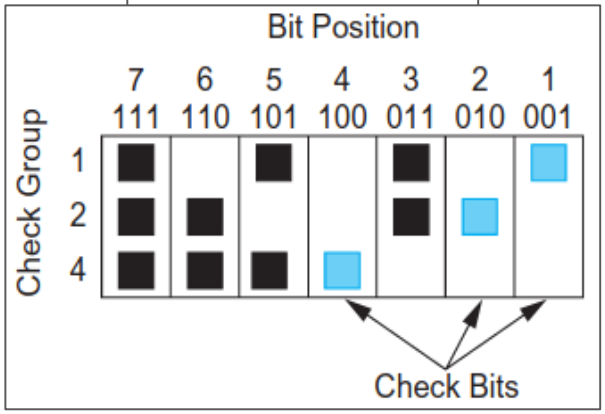
Quick and dirty explanation:

- Group different bits of a data word (of some length N) and keep track of their parity value, aka check bits
- Tracking the parity of different groups of bits simply consists of XOR-ing those bits
- That's it, we can now detect and correct one-bit errors!

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

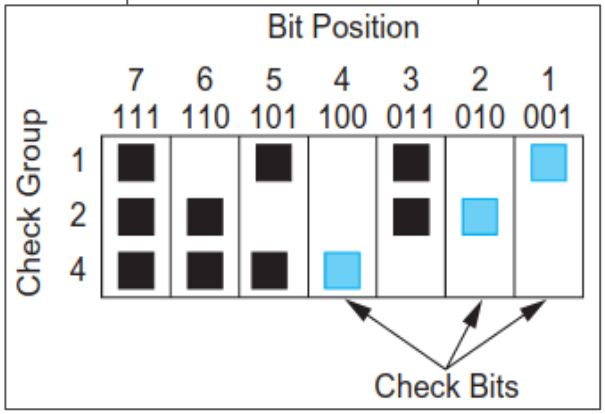
Quick and dirty explanation:

- Group different bits of a data word (of some length N) and keep track of their parity value, aka check bits
- Tracking the parity of different groups of bits simply consists of XOR-ing those bits
- That's it, we can now detect and correct one-bit errors!
- More data bits means more groups of bits to keep a check-bit for

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

Quick and dirty explanation:

- Group different bits of a data word (of some length N) and keep track of their parity value, aka check bits
- Tracking the parity of different groups of bits simply consists of XOR-ing those bits
- That’s it, we can now detect and correct one-bit errors!
- More data bits means more groups of bits to keep a check-bit for

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

FIGURE 5.23 Parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits.

Source: Computer Organization and Design
The Hardware Software Interface - 2nd Edition - Patterson, Hennessy

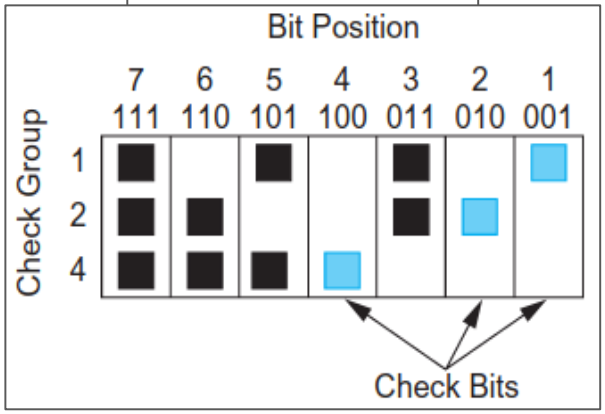
Fundamentals of Error Correction Codes (ECC)

Wait, how do you detect and correct errors again?

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

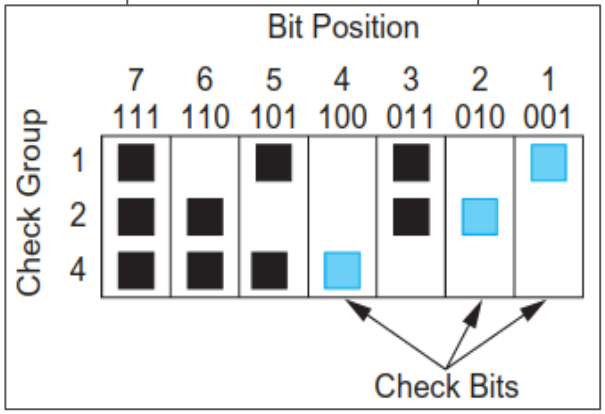
Wait, how do you detect and correct errors again?

- Data bits: 1001

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

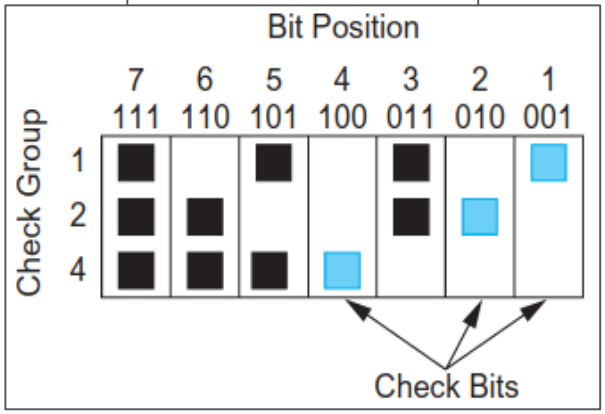
Wait, how do you detect and correct errors again?

- Data bits: 1001
- Check bits for the above data: 100

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

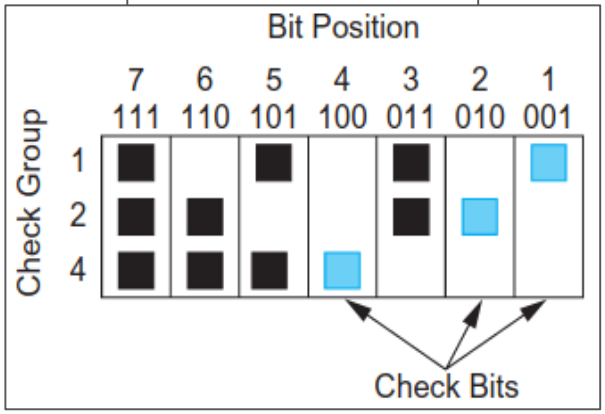
Wait, how do you detect and correct errors again?

- Data bits: 1001
- Check bits for the above data: 100
- Repr: **1001**<u>100</u>

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

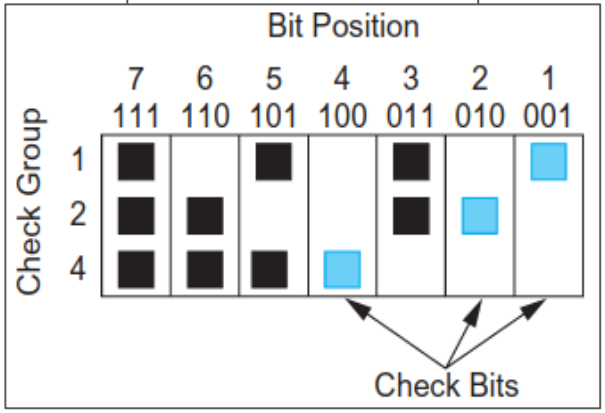
Wait, how do you detect and correct errors again?

- Data bits: 1001
- Check bits for the above data: 100
- Repr: **1001**<u>100</u>
- Let's say some alpha particle comes in and modifies the data: **1101**<u>100</u>

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

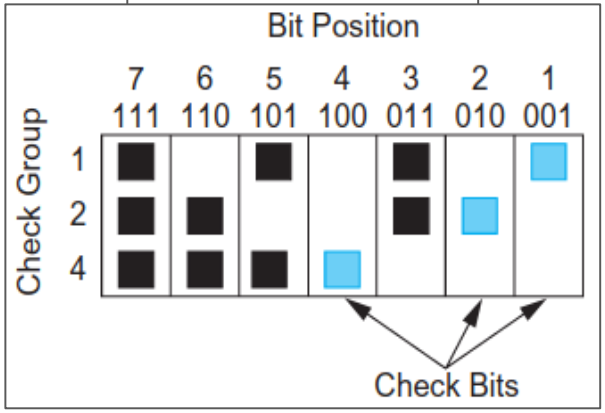
Wait, how do you detect and correct errors again?

- Data bits: 1001
- Check bits for the above data: 100
- Repr: **1001**<u>100</u>
- Let's say some alpha particle comes in and modifies the data: **1101**<u>100</u>

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

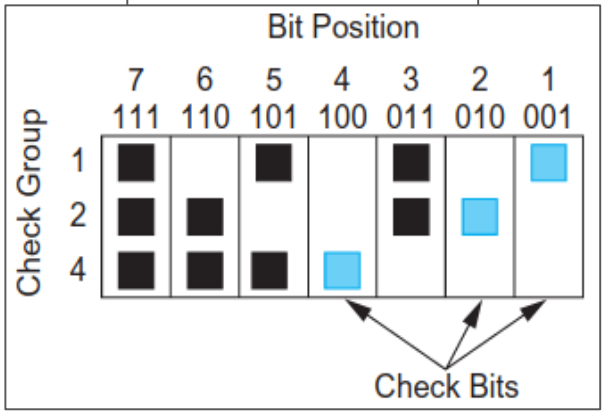
Wait, how do you detect and correct errors again?

- Data bits: 1001
- Check bits for the above data: 100
- Repr: **1001**<u>100</u>
- Let's say some alpha particle comes in and modifies the data: **1101**<u>100</u>
- In theory, **1101** should yield <u>010</u>

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

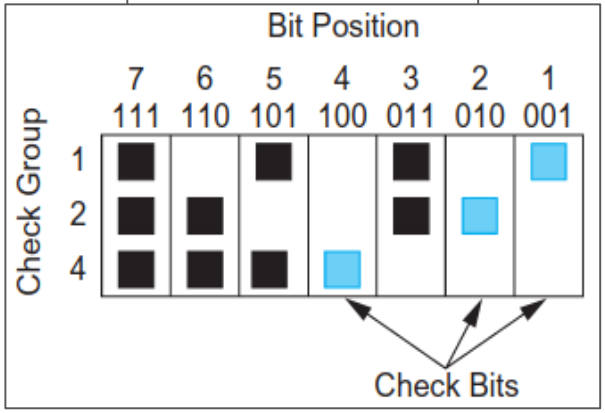
Wait, how do you detect and correct errors again?

- Data bits: 1001
- Check bits for the above data: 100
- Repr: **1001**<u>100</u>
- Let's say some alpha particle comes in and modifies the data: **1101**<u>100</u>
- In theory, **1101** should yield <u>010</u>
- XOR(<u>100</u>, <u>010</u>) = ***<u>110</u>***

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

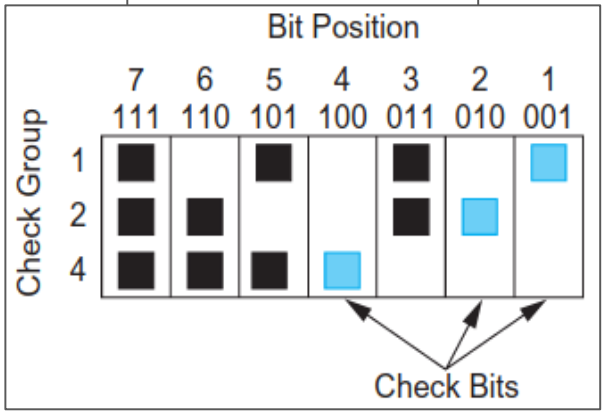
Wait, how do you detect and correct errors again?

- Data bits: 1001
- Check bits for the above data: 100
- Repr: **1001**<u>100</u>
- Let's say some alpha particle comes in and modifies the data: **1101**<u>100</u>
- In theory, **1101** should yield <u>010</u>
- XOR(<u>100</u>, <u>010</u>) = ***<u>110</u>***
- Wait, ***<u>110</u>*** is ***<u>6</u>*** in binary, the position of the bit that was flipped by the alpha particle!

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

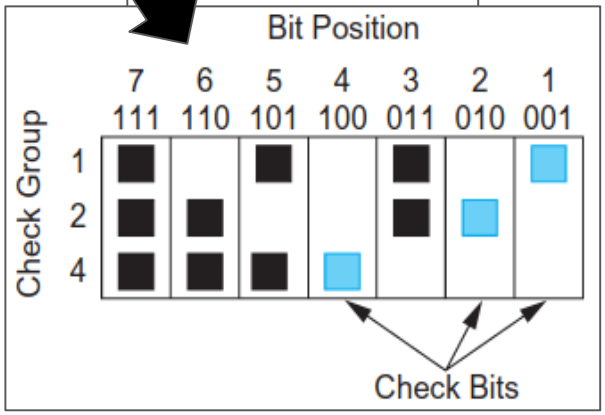
Wait, how do you detect and correct errors again?

- Data bits: 1001
- Check bits for the above data: 100
- Repr: **1001**<u>100</u>
- Let's say some alpha particle comes in and modifies the data: **1101**<u>100</u>
- In theory, **1101** should yield <u>010</u>
- XOR(<u>100</u>, <u>010</u>) = ***<u>110</u>***
- Wait, ***<u>110</u>*** is ***<u>6</u>*** in binary, the position of the bit that was flipped by the alpha particle!

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

Fundamentals of Error Correction Codes (ECC)

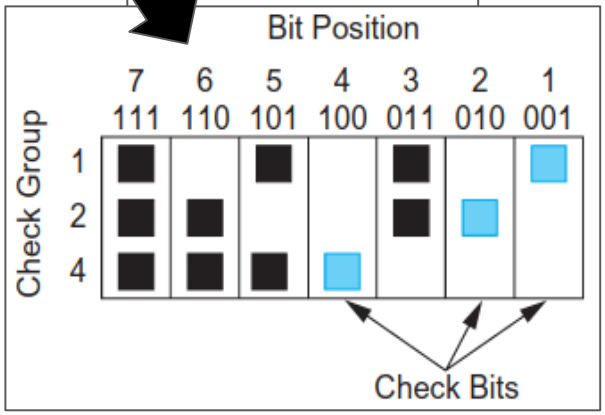
Wait, how do you detect and correct errors again?

- Data bits: 1001
- Check bits for the above data: 100
- Repr: **1001**<u>100</u>
- Let's say some alpha particle comes in and modifies the data: **1101**<u>100</u>
- In theory, **1101** should yield <u>010</u>
- XOR(<u>100</u>, <u>010</u>) = ***<u>110</u>***
- Wait, ***<u>110</u>*** is ***<u>6</u>*** in binary, the position of the bit that was flipped by the alpha particle!
- XOR-ing the unmodified check bits with the supposed check bits yields the ***<u>SYNDROME: the erroneous bit flipped!!</u>***

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

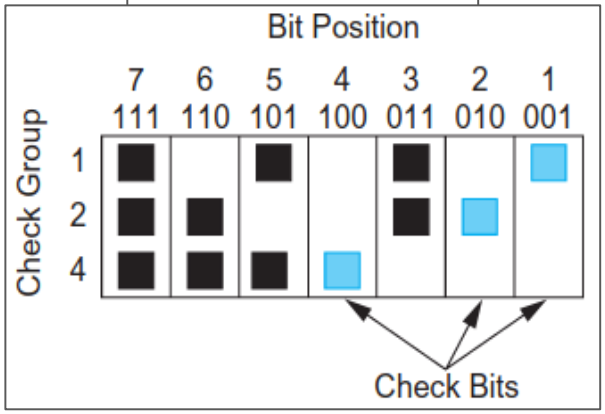
Fundamentals of Error Correction Codes (ECC)

- Furthermore, adding a parity bit to a data word with check bits (such as the one in the image on the right) results in a SEC-DED code: a Single-Error-Correcting Double-Error-Detecting correction code

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

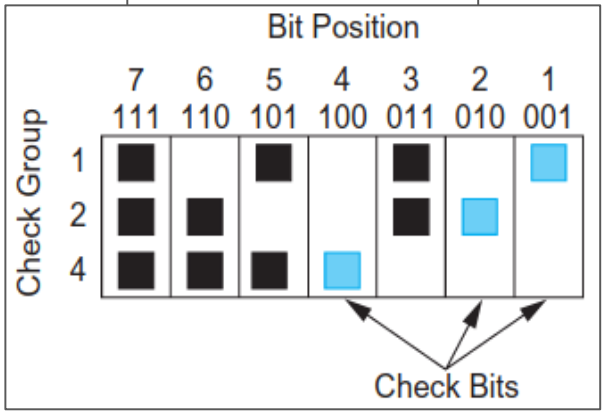
Fundamentals of Error Correction Codes (ECC)

- Furthermore, adding a parity bit to a data word with check bits (such as the one in the image on the right) results in a SEC-DED code: a Single-Error-Correcting Double-Error-Detecting correction code
- SED-DED and SEC-DEC also exist

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

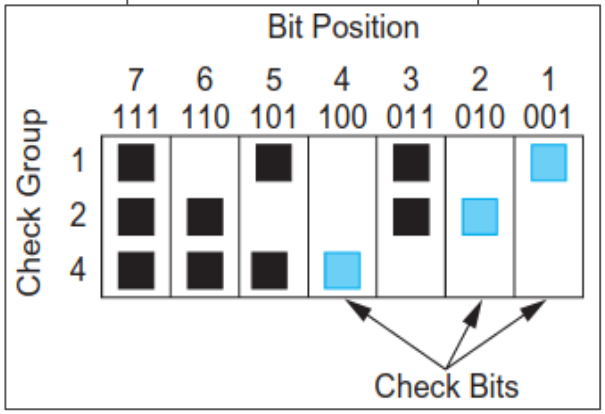
Fundamentals of Error Correction Codes (ECC)

- Furthermore, adding a parity bit to a data word with check bits (such as the one in the image on the right) results in a SEC-DED code: a Single-Error-Correcting Double-Error-Detecting correction code
- SED-DED and SEC-DEC also exist
- As the name suggests, with this we can correct single bit errors (watch out for odd-number bit-flips…) but only detect double-bit errors (even-numbered bit-flips, to be more specific…)

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

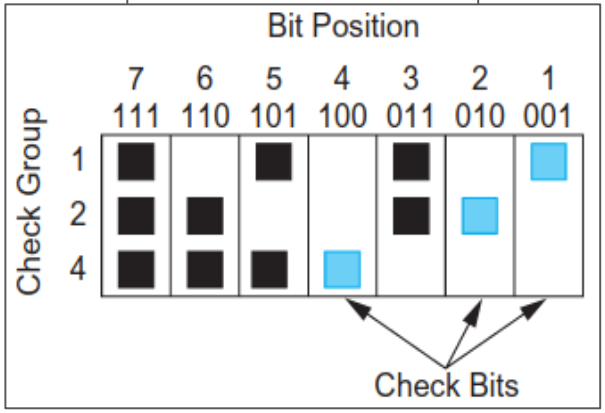
Fundamentals of Error Correction Codes (ECC)

- Furthermore, adding a parity bit to a data word with check bits (such as the one in the image on the right) results in a SEC-DED code: a Single-Error-Correcting Double-Error-Detecting correction code
- SED-DED and SEC-DEC also exist
- As the name suggests, with this we can correct single bit errors (watch out for odd-number bit-flips…) but only detect double-bit errors (even-numbered bit-flips, to be more specific…)
- If the parity bit of the possibly-erroneous data matches with the supposed parity bit <u>AND</u> the check bits don't match, there has been a double-bit error!

$$C_0 = D_3 \oplus D_1 \oplus D_0$$

$$C_1 = D_3 \oplus D_2 \oplus D_0$$

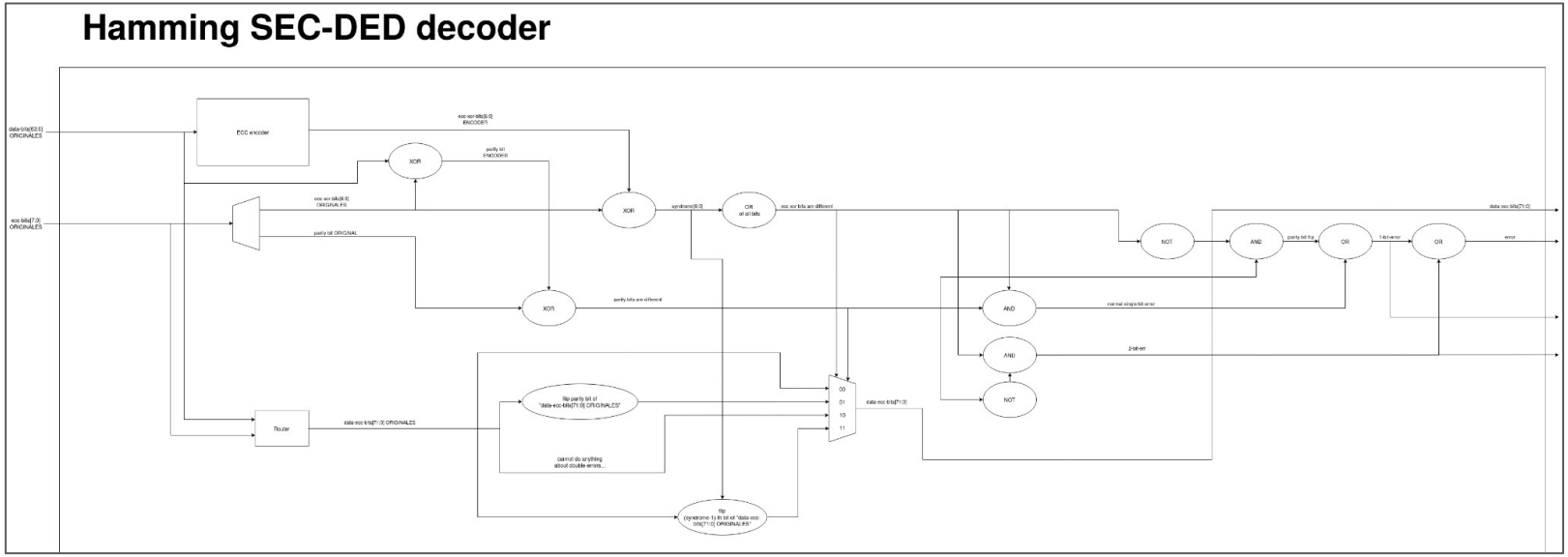
$$C_2 = D_3 \oplus D_2 \oplus D_1$$



Source: CMOS VLSI Design
A Circuits and Systems Perspective - 4th Edition - Weste, Harris

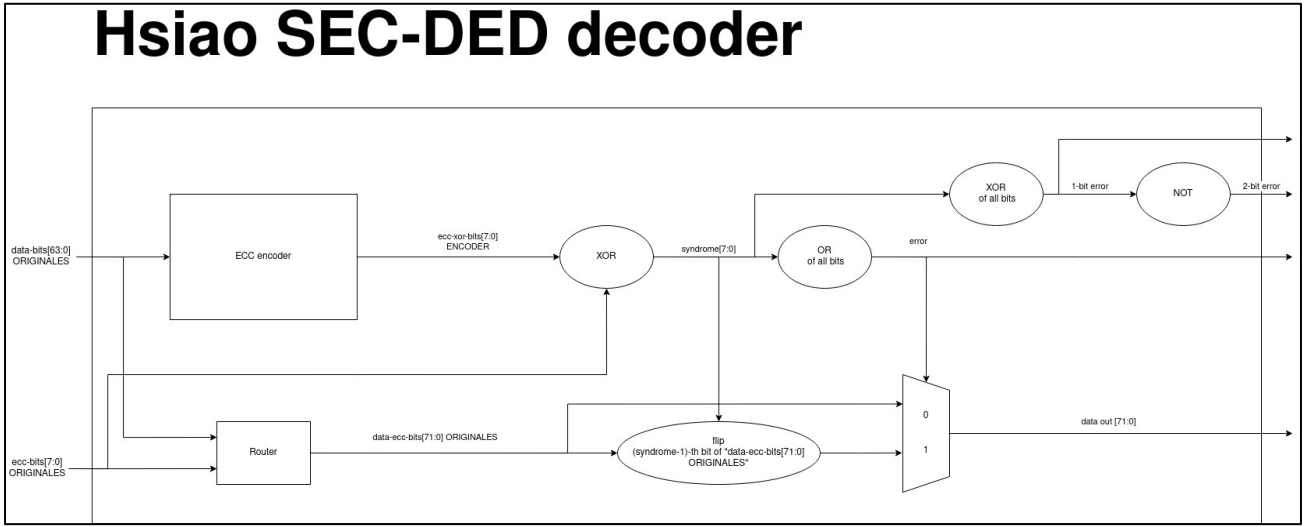
Fundamentals of Error Correction Codes (ECC)

The logic for managing SEC-DED codes is a bit cumbersome…



Fundamentals of Error Correction Codes (ECC)

But a paper from the 70’s simplifies the logic by a lot… This is what we ended up implementing!



Fundamentals of Error Correction Codes (ECC)

But a paper from the 70’s simplifies the logic by a lot… This is what we ended up implementing!

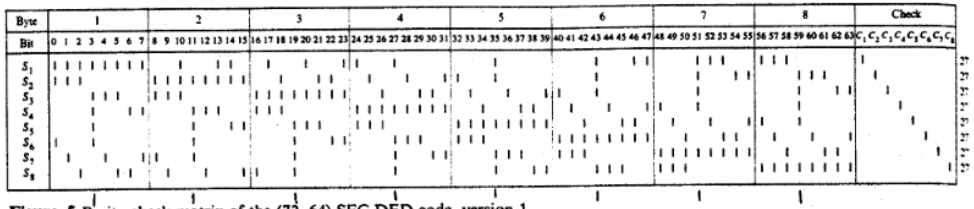
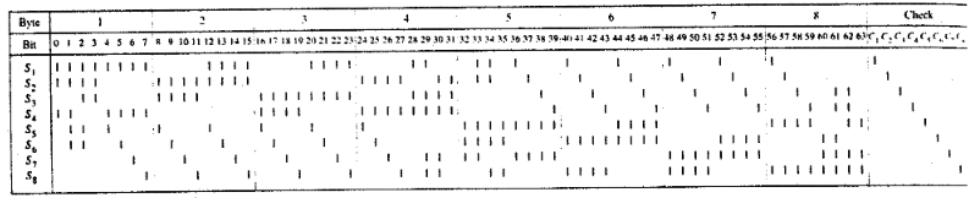


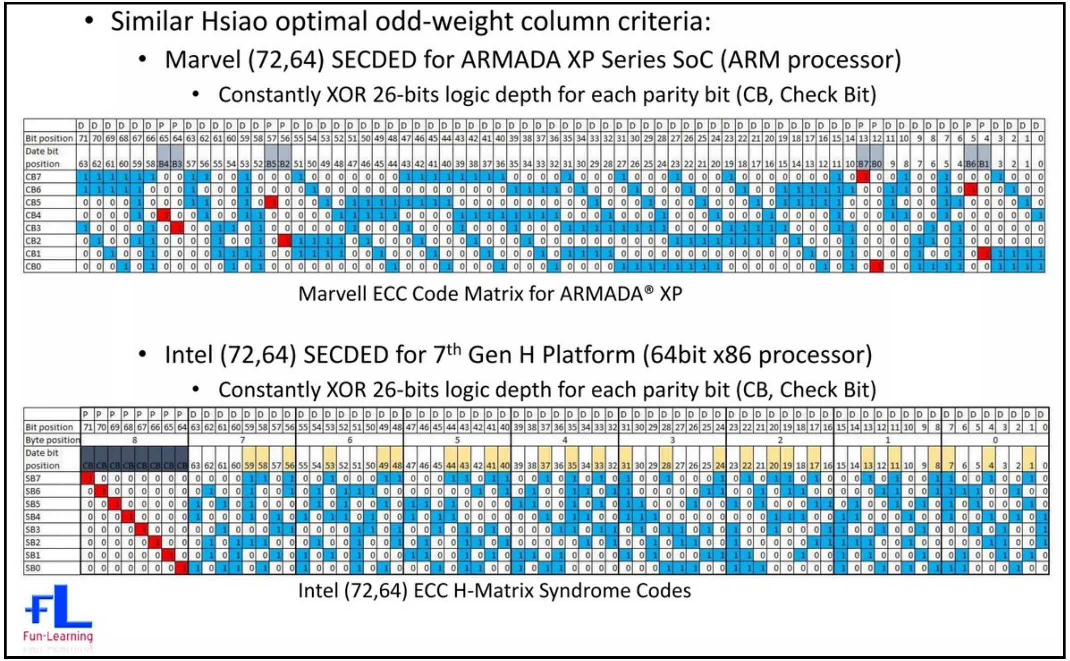
Figure 5 Parity-check matrix of the (72, 64) SEC-DED code, version 1.

Figure 6 Parity-check matrix of the (72, 64) SEC-DED code, version 2.



Fundamentals of Error Correction Codes (ECC)

Other check bit patterns also exist…



- Similar Hsiao optimal odd-weight column criteria:
  - Marvel (72,64) SECDED for ARMADA XP Series SoC (ARM processor)
    - Constantly XOR 26-bits logic depth for each parity bit (CB, Check Bit)

	D	D	D	D	D	D	P	P	D	D	D	D	D	D	P	P	D	D
Bit position	71	70	69	68	67	66	65	64	63	62	61	60	59	58	57	56	55	54
Date bit position	63	62	61	60	59	58	B4	B3	57	56	55	54	53	52	B5	B2	51	50
CB7	1	1	1	1	1	1	0	0	1	1	0	0	1	0	0	0	1	0
CB6	1	1	1	1	1	0	0	0	1	0	0	0	1	0	0	0	0	1
CB5	0	0	0	0	1	0	0	0	1	1	0	0	1	0	1	0	0	0
CB4	0	0	0	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0
CB3	1	0	0	0	0	1	0	1	0	0	1	1	0	1	0	0	0	0
CB2	0	1	0	0	1	1	0	0	0	0	1	0	0	1	0	1	1	1
CB1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	0	0	1	1
CB0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0

	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
Bit position	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36
Date bit position	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
CB7	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
CB6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
CB5	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
CB4	0	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1
CB3	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
CB2	1	1	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
CB1	1	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0
CB0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1

	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
Bit position	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18
Date bit position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14
CB7	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0
CB6	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1	1
CB5	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	1	1
CB4	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
CB3	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	0
CB2	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	1
CB1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CB0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0

	D	D	D	D	P	P	D	D	D	D	D	D	P	P	D	D	D	D
Bit position	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Date bit position	13	12	11	10	B7	B0	9	8	7	6	5	4	B6	B1	3	2	1	0
CB7	0	0	1	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0
CB6	1	1	1	1	0	0	1	1	0	0	1	0	1	0	0	1	0	0
CB5	1	1	1	0	0	0	1	0	0	0	1	1	0	0	0	0	1	0
CB4	0	0	1	0	0	0	1	1	0	0	1	0	0	0	0	0	0	1
CB3	0	0	1	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0
CB2	0	0	0	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0
CB1	1	0	0	1	0	0	0	0	1	0	0	1	0	1	1	1	1	1
CB0	0	1	0	1	0	1	0	0	1	1	1	1	0	0	1	1	1	1

Marvell ECC Code Matrix for ARMADA® XP

- Intel (72,64) SECDED for 7th Gen H Platform (64bit x86 processor)
  - Constantly XOR 26-bits logic depth for each parity bit (CB, Check Bit)

	P	P	P	P	P	P	P	P	D	D	D	D	D	D	D	D	D	D
Bit position	71	70	69	68	67	66	65	64	63	62	61	60	59	58	57	56	55	54
Byte position	8								7								6	
Date bit position	CB	CB	CB	CB	CB	CB	CB	CB	63	62	61	60	59	58	57	56	55	54
SB7	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0
SB6	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	1
SB5	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	0	1
SB4	0	0	0	1	0	0	0	0	1	0	0	0	1	0	1	0	1	0
SB3	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	1	0	0
SB2	0	0	0	0	0	1	0	0	0	1	0	1	1	1	0	0	0	1
SB1	0	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0
SB0	0	0	0	0	0	0	0	1	1	0	1	1	0	1	0	0	1	0

	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
Bit position	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36
Byte position							5								4			
Date bit position	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36
SB7	1	0	0	0	1	1	0	0	0	1	1	0	1	1	0	0	1	0
SB6	0	1	1	1	0	0	0	0	0	0	0	1	0	1	0	1	0	0
SB5	0	1	0	0	1	0	1	1	0	1	0	0	0	0	1	1	0	0
SB4	1	1	0	1	0	0	1	0	1	1	0	0	0	0	0	0	1	0
SB3	0	0	1	1	0	1	0	0	1	0	1	0	1	0	0	0	0	1
SB2	0	0	1	0	0	1	0	1	0	0	1	1	0	1	0	0	0	0
SB1	1	0	1	0	0	0	1	1	0	0	1	0	1	0	1	1	0	1
SB0	0	0	1	0	1	0	0	0	1	0	1	1	0	0	1	0	1	1

	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
Bit position	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18
Byte position					3								2					
Date bit position	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18
SB7	1	0	1	0	1	0	0	1	0	0	0	1	0	1	0	1	1	0
SB6	1	1	0	1	1	0	0	0	0	0	1	1	0	0	1	0	1	1
SB5	1	0	1	0	1	1	0	0	0	1	0	0	1	0	1	0	0	0
SB4	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	1	1	1
SB3	1	0	1	1	0	1	0	1	1	0	1	0	1	0	0	1	0	0
SB2	0	1	0	1	0	0	1	0	1	1	0	0	1	0	0	0	0	0
SB1	0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	0	0	1
SB0	0	0	0	0	0	0	0	1	1	1	0	0	1	1	1	0	0	0

	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D
Bit position	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Byte position			1								0							
Date bit position	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SB7	1	0	0	0	1	0	1	0	0	1	1	0	0	1	0	0	1	0
SB6	0	0	0	0	0	1	1	0	0	1	1	1	1	0	0	1	0	0
SB5	1	1	0	1	0	0	1	0	1	0	1	0	1	0	0	0	1	0
SB4	0	0	1	1	0	0	0	1	0	0	1	0	0	1	0	1	0	1
SB3	0	1	1	0	0	1	0	0	1	0	0	0	1	0	1	0	0	1
SB2	1	1	1	1	1	0	0	1	0	0	0	0	0	1	1	0	0	1
SB1	0	0	1	0	1	0	0	0	1	0	0	1	0	0	1	0	1	0
SB0	0	0	1	0	0	1	0	1	0	1	1	1	0	0	0	1	0	0

Intel (72,64) ECC H-Matrix Syndrome Codes

Fun-Learning

Fundamentals of Error Correction Codes (ECC)

Sources:

CMOS VLSI Design: A Circuits and Systems Perspective - 4th Edition - Weste, Harris

→ pages 468-470

→ pages 543-544

Computer Organization and Design: The Hardware Software Interface - 2nd RISC-V Edition - Patterson, Hennessy

→ pages 433-436

→ pages A-(64-66)

https://people.eecs.berkeley.edu/~culler/cs252-s02/papers/hsiao70.pdf

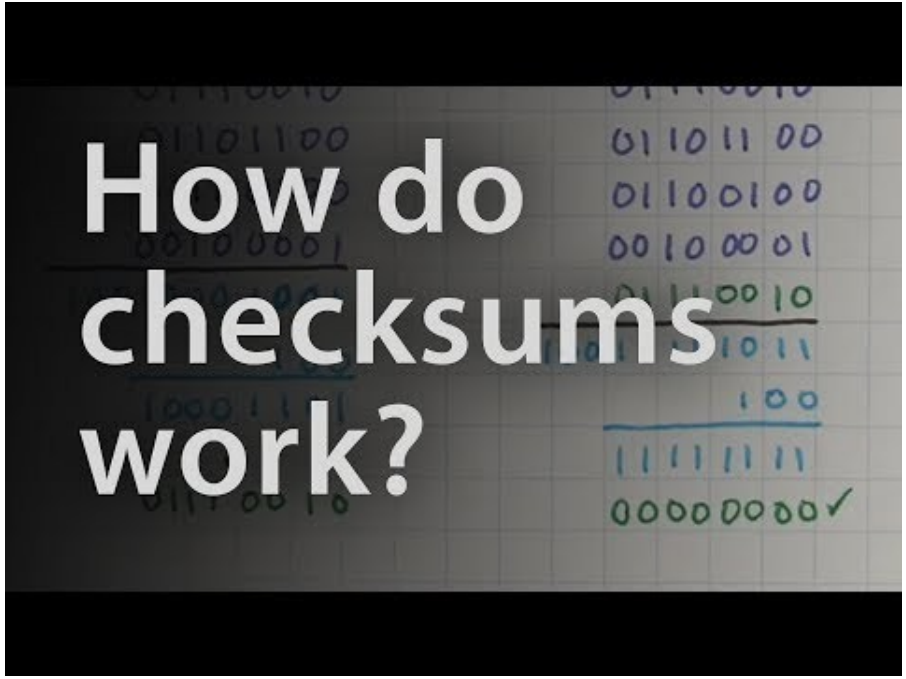
https://www.slideshare.net/SkCheah/memory-ecc-the-comprehensive-of-secded

Fundamentals of Error Correction Codes (ECC)

Video by Ben Eater:

Checksums and Hamming distance

https://www.youtube.com/watch?v=ppU41c15Xho

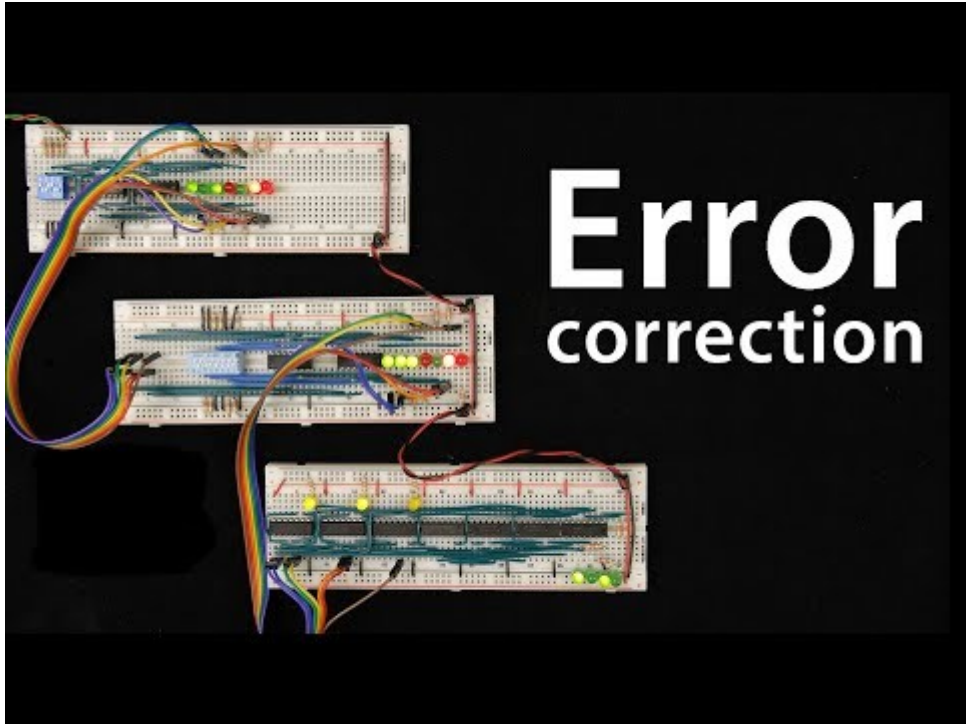


Fundamentals of Error Correction Codes (ECC)

Video by Ben Eater:

What is error correction? Hamming codes in hardware

https://www.youtube.com/watch?v=h0jloehRKas

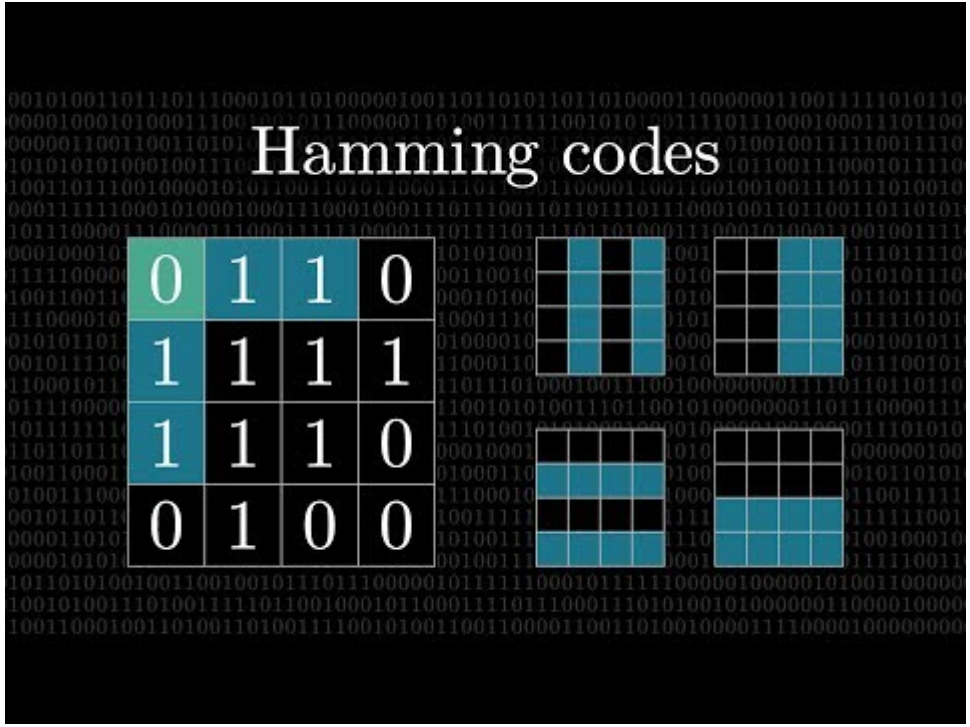


Fundamentals of Error Correction Codes (ECC)

Video by 3Blue1Brown (more math-related):

How to send a self-correcting message (Hamming codes)

https://www.youtube.com/watch?v=X8jsijhllIA

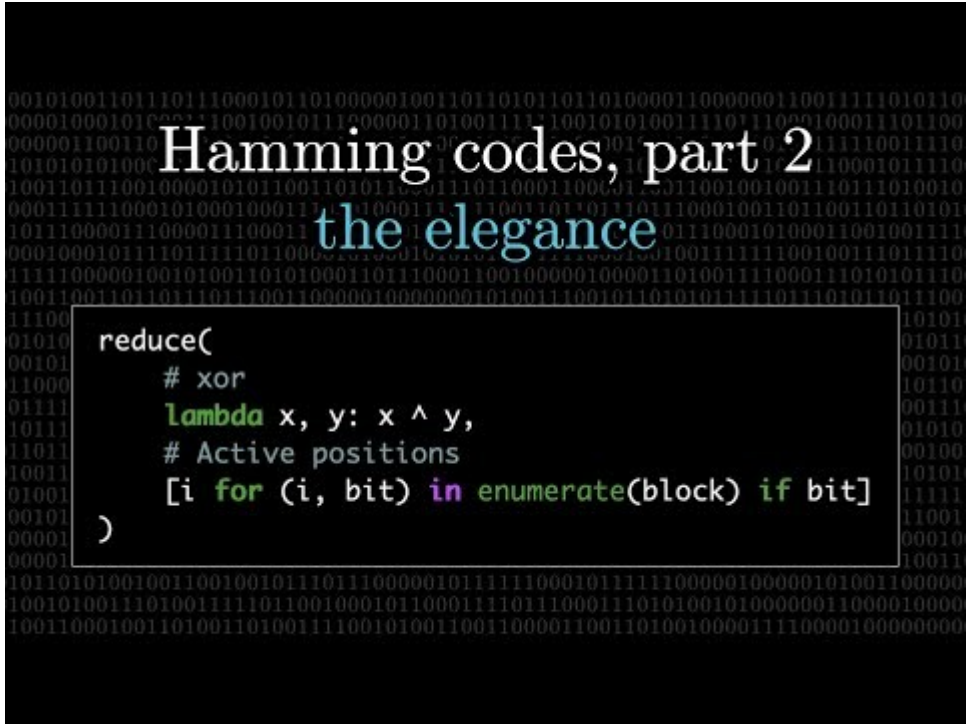


Fundamentals of Error Correction Codes (ECC)

Video by 3Blue1Brown (more math-related):

Hamming codes part 2, the elegance of it all

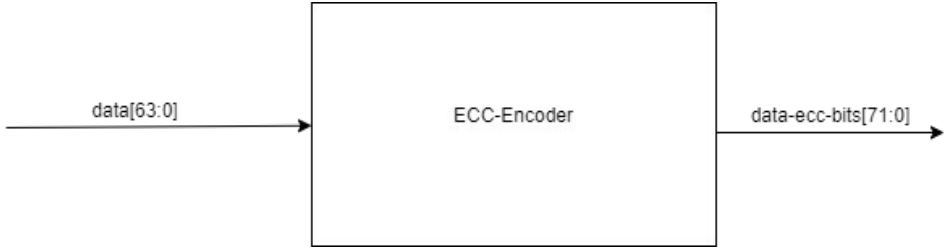
https://www.youtube.com/watch?v=b3NxrZOu_CE



Modules for implementing and testing ECC functionalities

Encoder

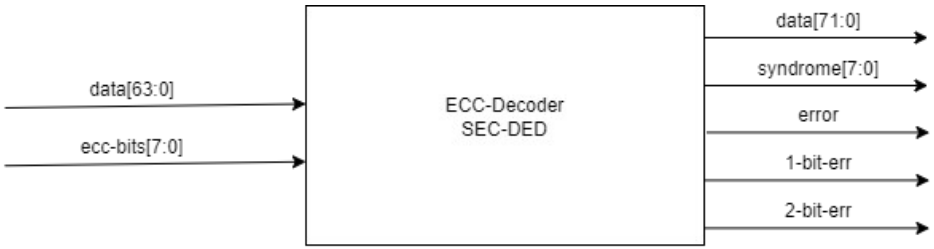
- Generate ECC bits
- Hsiao algorithm

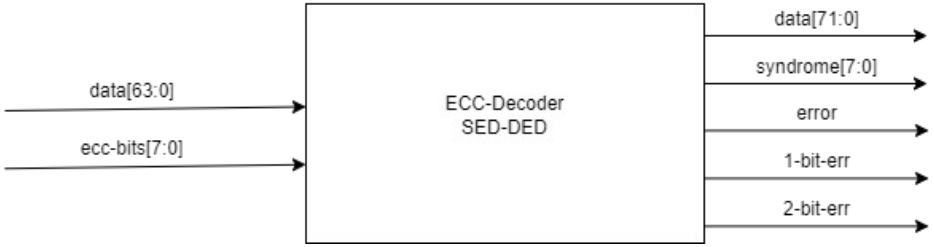


Modules for implementing and testing ECC functionalities

Decoder

- Recalculate syndrome
- Detect/Correct 1-bit errors
- Detect 2-bit errors
- Error signals





Modules for implementing and testing ECC functionalities

Fault Injector

- Functionality test
- Inject errors
  - No error
  - 1-bit error
  - 2-bit error
  - Random

Where ECC modules should go in Ariane

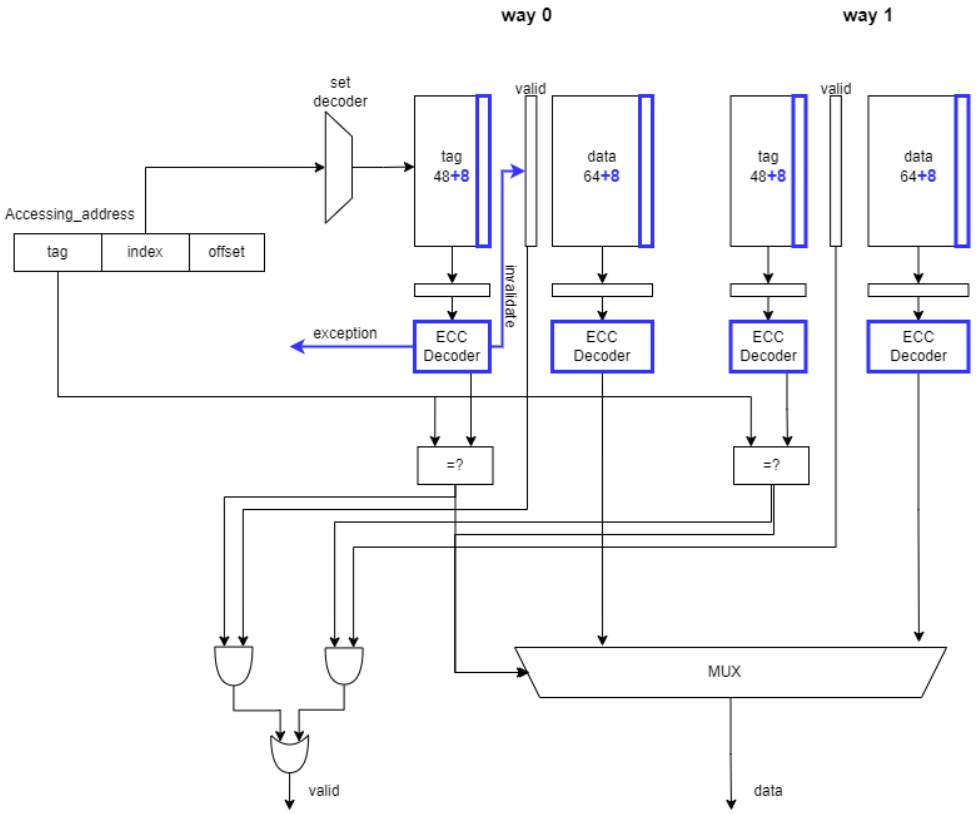
Ariane cache

- 8-way 32 KB associative cache with 16 byte line sizes
- write-back with no write allocation

Where ECC modules should go in Ariane

Simple implementation

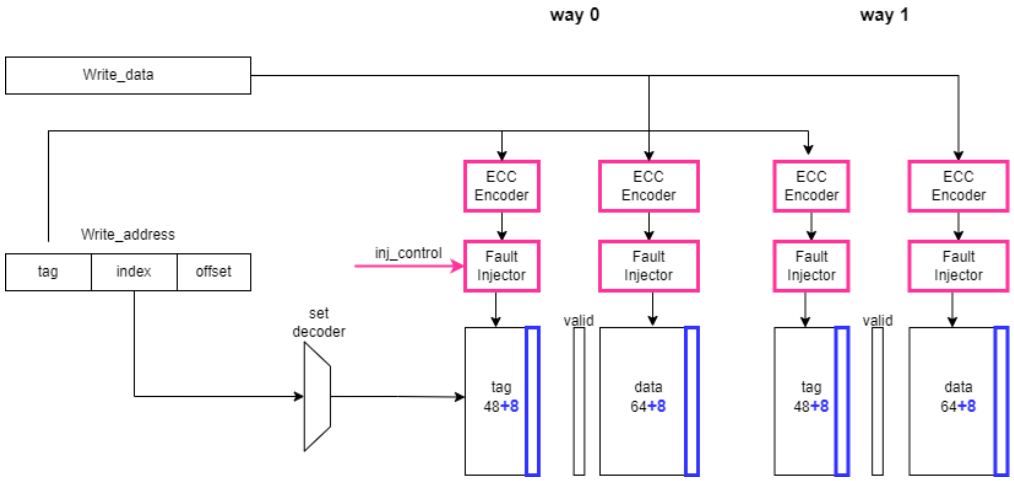
- Extend tag and data sizes to store ecc bits
- Insert Decoder to correct single bit errors
- Connect output signals to invalidate or launch exception in case of double bit errors



Where ECC modules should go in Ariane

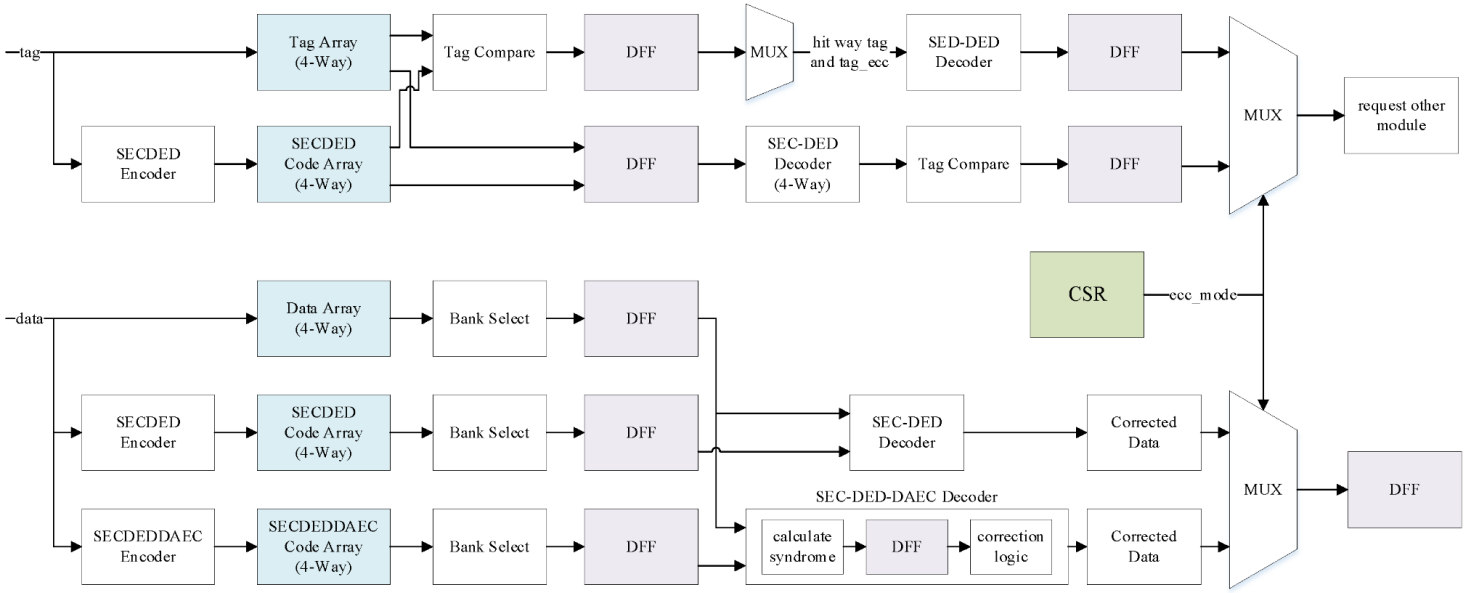
Testing implementation

- Insert Encoder to simulate ECC in top-level cache
- Insert Fault Injector to simulate errors



Where ECC modules should go in Ariane

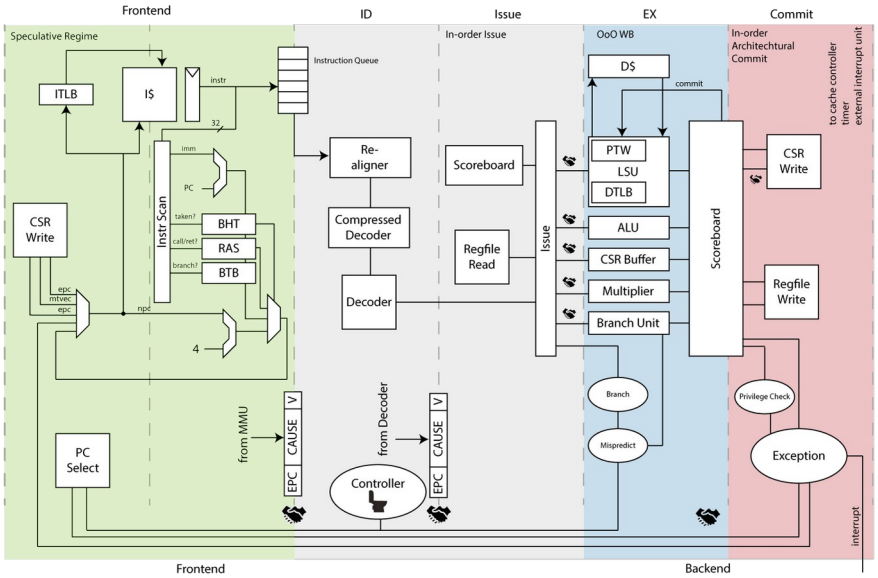
Actual High-End implementation example



Zhou, Y.; Liu, H.; Xiang, Q.; Yin, C. High-Performance and Flexible Design Scheme with ECC Protection in the Cache. *Micromachines* **2022**, *13*, 1931. https://doi.org/10.3390/mi13111931

Implementation details

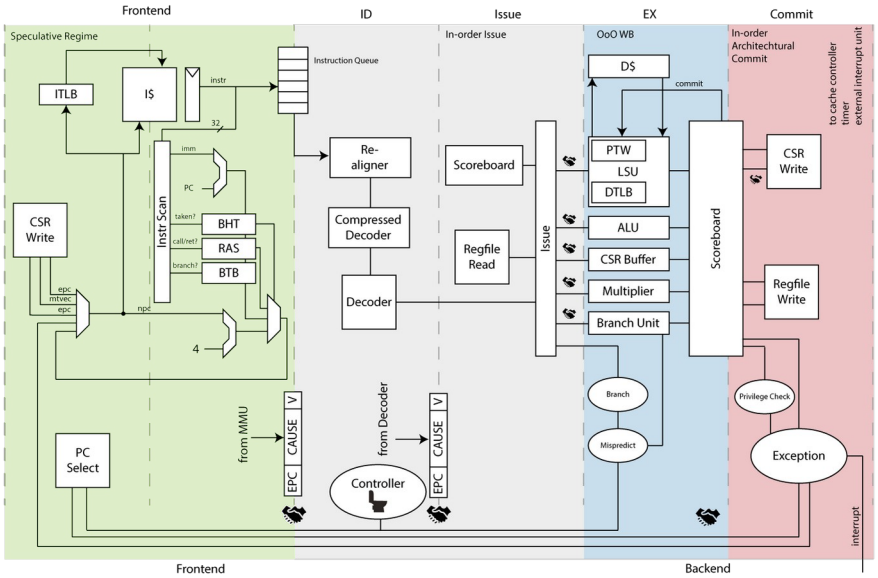
- The Ariane processor (now called CVA6) is one of the seven RISC-V cores developed and maintained by OpenHW Group



CVA6 diagram, from https://github.com/openhwgroup/cva6

Implementation details

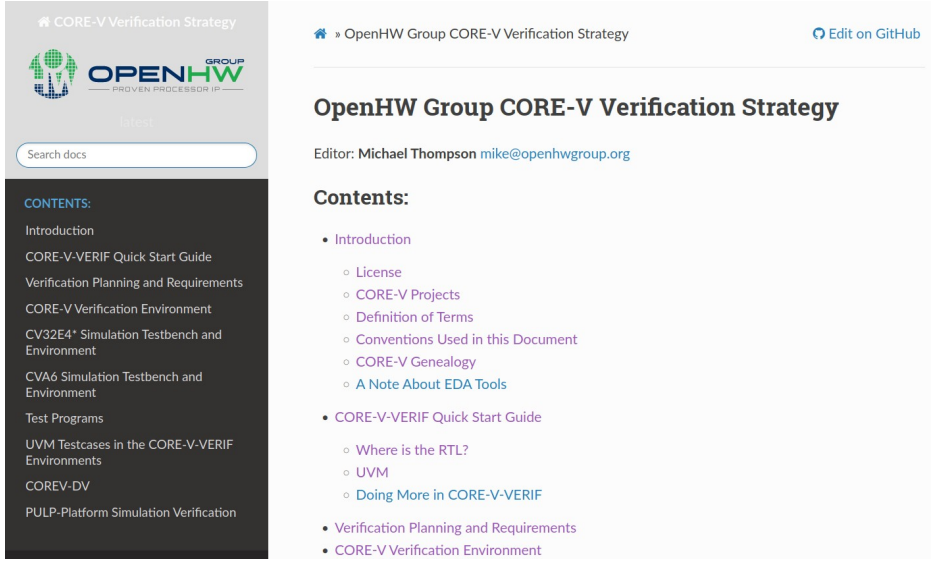
- The Ariane processor (now called CVA6) is one of the seven RISC-V cores developed and maintained by OpenHW Group
- Their projects include (all of them “open source”):
  - Two application-class cores in CVA5 and CVA6 (with M, S, U privilege levels for operating system support and the like)
  - Five smaller embedded cores in CV32E40{P,X,S}, CV32E41P and CVE2



CVA6 diagram, from https://github.com/openhwgroup/cva6

Implementation details

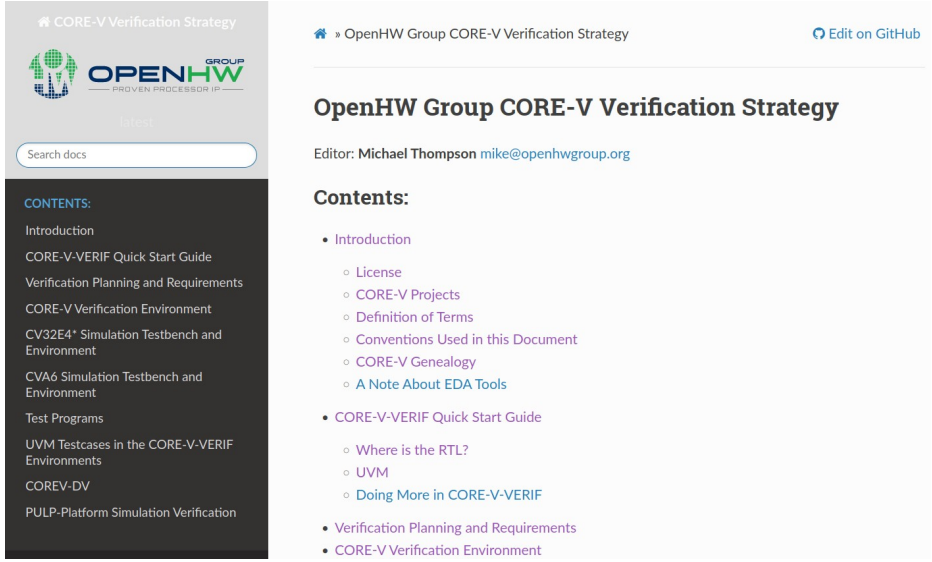
- OpenHW Group maintain separate Github repositories for the RTL and their corresponding verification scripts and utilities



Source: [core-v-verif readthedocs page]

Implementation details

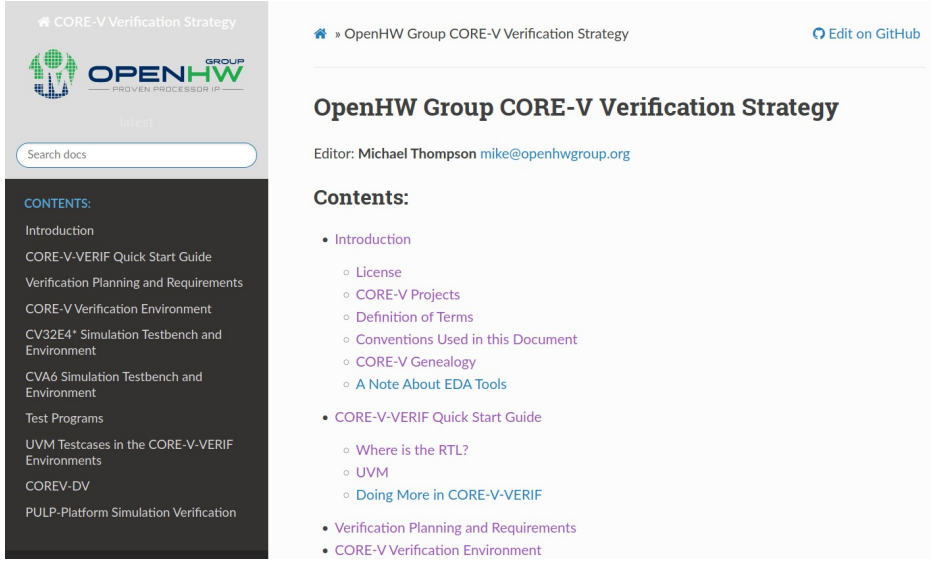
- OpenHW Group maintain separate Github repositories for the RTL and their corresponding verification scripts and utilities
- Source code and RTL for CVA6 in https://github.com/openhwgroup/cva6



Source: [core-v-verif readthedocs page]

Implementation details

- OpenHW Group maintain separate Github repositories for the RTL and their corresponding verification scripts and utilities
- Source code and RTL for CVA6 in https://github.com/openhwgroup/cva6
- In the [core-v-verif](https://github.com/openhwgroup/core-v-verif) repository resides the verification part of all the cores
  - CV32E40P core is the more mature and stable of the projects
  - Others not so much, varying levels of progress…



Source: [core-v-verif readthedocs page](https://core-v-verif.readthedocs.io/)

Implementation details

- Op
rep
cor
- Sou
http
- In t
ver
  -
  -

Source: [core-v-verif readthedocs page]

Implementation details

Why are we mentioning these two repositories? What does core-v-verif add to the table?

Implementation details

Why are we mentioning these two repositories? What does core-v-verif add to the table?

- The verification of (all) the cores includes **RISC-V compliance tests** and **per-instruction testbenches** to confirm functionality of the processor

Implementation details

Why are we mentioning these two repositories? What does core-v-verif add to the table?

- The verification of (all) the cores includes **RISC-V compliance tests** and **per-instruction testbenches** to confirm functionality of the processor
- A key aspect of correct ECC implementation is noticing whether a “flipped” line in cache (so, with errors) is touched or used at all by a program and responds correctly (exception and/or invalidation of cache line, correction of single-bit errors…)
  - Bit-flips could be introduced to cache lines that are not touched by a program, possibly resulting in processor-related errors (introduced by modifications of unaware students, most likely) going unnoticed
  - The more testbenches to confirm this with, the better

Implementation details

Why are we mentioning these two repositories? What does core-v-verif add to the table?

- The verification of (all) the cores includes **RISC-V compliance tests** and **per-instruction testbenches** to confirm functionality of the processor
- A key aspect of correct ECC implementation is noticing whether a “flipped” line in cache (so, with errors) is touched or used at all by a program and responds correctly (exception and/or invalidation of cache line, correction of single-bit errors…)
  - Bit-flips could be introduced to cache lines that are not touched by a program, possibly resulting in processor-related errors (introduced by modifications of unaware students, most likely) going unnoticed
  - The more testbenches to confirm this with, the better
- Depending on the type of bit flip introduced (either no errors, single-bit errors or double-bit errors), the results for any and all testbenches are easily predictable
  - PASSED for no errors or single-bit errors and (many) UNPASSED testbenches for double-bit errors

Implementation details

Why are we mentioning these t… d to the table?

- The verification … **nstruction testbenches**
- A key aspe… che (so, with errors) is t… /or invalidation of cache li…
  - Bit-flip… bly resulting in processo… likely) going unnoticed
  - The more testbenc…
- Depending on the type of bit flip i… , single-bit errors or double-bit errors), the results for any and all testbenches are easily predictable
  - PASSED for no errors or single-bit errors and (many) UNPASSED testbenches for double-bit errors

Also, waveform files from each testbench!!!

Much better and comprehensive than anything we could conceive of…

Implementation details

Integration-hell bullet points:

Implementation details

Integration-hell bullet points:

- CVA6 is a **big project** (also, formerly managed by ETH, so source code is not 100% native to OpenHW Group)

Implementation details

Integration-hell bullet points:

- CVA6 is a **big project** (also, formerly managed by ETH, so source code is not 100% native to OpenHW Group)
- core-v-verif is **even bigger** and depends on many other RISC-V-related projects and tools

Implementation details

Integration-hell bullet points:

- CVA6 is a **big project** (also, formerly managed by ETH, so source code is not 100% native to OpenHW Group)
- core-v-verif is **even bigger** and depends on many other RISC-V-related projects and tools
- Many contributors from different entities (OpenHW, ETH, Thales…)

Implementation details

Integration-hell bullet points:

- CVA6 is a **big project** (also, formerly managed by ETH, so source code is not 100% native to OpenHW Group)
- core-v-verif is **even bigger** and depends on many other RISC-V-related projects and tools
- Many contributors from different entities (OpenHW, ETH, Thales…)
- All this results in:

Implementation details

Integration-hell bullet points:

- CVA6 is a **big project** (also, formerly managed by ETH, so source code is not 100% native to OpenHW Group)
- core-v-verif is **even bigger** and depends on many other RISC-V-related projects and tools
- Many contributors from different entities (OpenHW, ETH, Thales…)
- All this results in:
  - Basic tutorials in the documentation not working
  - Modifications to Makefiles and other scripts
  - A lot of tinkering with Verilator
  - Little time for source-code spelunking

Implementation details

Integration-hell bullet points:

- All this results in:
  - Basic tutorials in the documentation not working
  - Modifications to Makefiles and other scripts
  - A lot of tinkering with Verilator
  - Little time for source-code spelunking

Implementation details

Integration-hell bullet points:

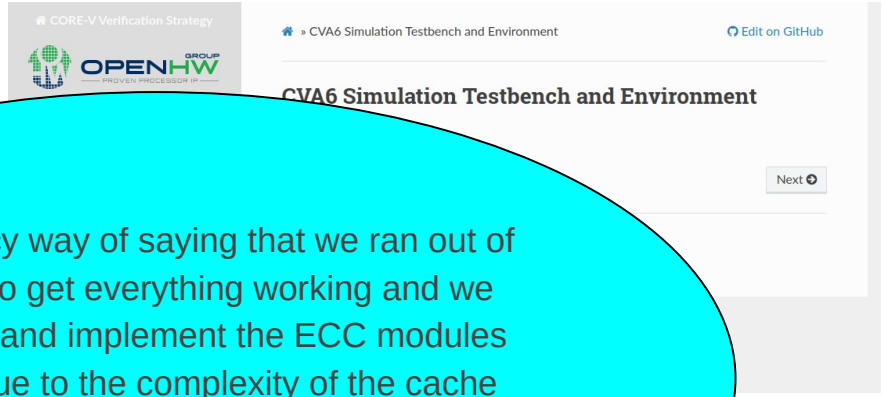
- All this results in:
  - Basic tutorials in the documentation not working
  - Modifications to Makefiles and other scripts
  - A lot of tinkering with Verilator
  - Little time for source-code spelunking

- Our final report consists of a 30-page tutorial on how to modify and “fix” the CVA6 and the core-v-verif repositories to obtain waveform files for all the compliance and instruction tests of the processor, as well as a brief explanation of the cache subsystem within CVA6

Implementation details

Integration-hell bullet points:

- All this results in:
  - Basic tutorials in the documentation not working
  - Modifications to Makefiles and other scripts
  - A lot of tinkering with Verilator
  - Little time for source-code spelunkin
- Our final report consists of a 2
  on how to modify and “fix” t
  core-v-verif repositories to
  files for all the compliance a
  of the processor, as well as a
  of the cache subsystem within C



This is a fancy way of saying that we ran out of time trying to get everything working and we couldn’t test and implement the ECC modules ourselves due to the complexity of the cache subsystem code, but hopefully our findings and explorations save some time for future students that also want to work with the CVA6 processor, either ECC-wise or with other modules…

Questions?