



國立清華大學  
NATIONAL TSING HUA UNIVERSITY

Department of Computer Science

National Tsing Hua University

IIS500800 Hardware Security

Spring, 2025

Final Project: Data Integrity with Checkpoint  
and ECC

Team01:

110062142 藍少彤

X1136010 黃偉祥

109062220 陳昭旭

X1136018 阮銘福

2025/6/15

# Contents

<b>1</b>	<b>Part1</b>	<b>3</b>
1.1	Prerequisites . . . . .	3
1.2	Clone the Gem5 Repository . . . . .	3
1.3	Build gem5 . . . . .	4
1.4	Verify the Build . . . . .	4
1.5	Test with Hello World Program . . . . .	5
<b>2</b>	<b>Part2</b>	<b>5</b>
2.1	Hamming ECC . . . . .	5
2.1.1	Encoding ( <code>encode</code> function): . . . . .	6
2.1.2	Decoding ( <code>decode</code> function): . . . . .	6
2.1.3	Error Injection ( <code>inject_random_error</code> function): . . . . .	6
2.1.4	Results . . . . .	6
2.2	Reed Solomon ECC . . . . .	8
2.2.1	Finite Field $GF(2^m)$ . . . . .	8
2.2.2	Encoding Process . . . . .	8
2.2.3	Syndrome Calculation . . . . .	9
2.2.4	Error Correction Procedure . . . . .	9
2.2.5	Corresponding Functions in Code . . . . .	10
2.2.6	Encapsulation of RS code . . . . .	11
2.2.7	Results . . . . .	11
2.2.8	Reference . . . . .	11
<b>3</b>	<b>Part3 &amp; Part4</b>	<b>12</b>
3.1	HammingCode . . . . .	13
3.1.1	HammingCode Interface . . . . .	13
3.1.2	Example Usage . . . . .	13
3.1.3	Results . . . . .	13
3.2	RSCode . . . . .	14
3.2.1	RSCode Interface . . . . .	14
3.2.2	Example Usage . . . . .	14

3.2.3	Results . . . . .	15
<b>4</b>	<b>Bonus</b>	<b>15</b>
4.1	Bose–Chaudhuri–Hocquenghem (BCH) . . . . .	15
4.2	Encoding ( <code>encode</code> function): . . . . .	16
4.3	Decoding ( <code>decode</code> function): . . . . .	16
4.4	Error Injection ( <code>inject_random_errors</code> function): . . . . .	17
4.5	BCH code interface . . . . .	17
4.6	Example Usage . . . . .	18
4.7	Results . . . . .	18
<b>5</b>	<b>Part5</b>	<b>20</b>
5.1	Integration and Simulation of ECC in Gem5 Memory System . . . . .	20
<b>6</b>	<b>Part6</b>	<b>24</b>
6.1	Simulation results . . . . .	24
6.2	Original hello world . . . . .	26
6.3	Hamming code example . . . . .	27
6.4	Reed-Solomon example . . . . .	27
6.5	Bose–Chaudhuri–Hocquenghem example . . . . .	28
6.6	Conclusion & Observation . . . . .	28
<b>7</b>	<b>Division of Work</b>	<b>29</b>
7.1	Division of Work . . . . .	29
<b>8</b>	<b>Conclusions</b>	<b>29</b>
8.1	What have you learned from this project? . . . . .	29
8.2	What problem(s) have you encountered in this project? . . . . .	29
8.3	Suggestions and feedback . . . . .	30

# 1 Part1

## 1.1 Prerequisites

First, ensure you have a Linux machine with the necessary dependencies:

```
bash

# For Ubuntu/Debian systems
sudo apt update
sudo apt install build-essential git m4 scons zlib1g zlib1g-dev \
    libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev \
    python3-dev python3-six python-is-python3 libboost-all-dev pkg-config
```

Figure 1: Prerequisites

## 1.2 Clone the Gem5 Repository

```
bash

# Clone the official Gem5 repository
git clone https://github.com/gem5/gem5.git
cd gem5
```

Figure 2: Git clone

### 1.3 Build gem5

```
bash

# Build Gem5 with X86 architecture (this will take some time)
scons build/X86/gem5.opt -j$(nproc)
```

Figure 3: Build gem5

Note:

- Replace *nproc* with the number of CPU cores you want to use for compilation
- The build process can take 30-60 minutes depending on your system

### 1.4 Verify the Build

Check if the build was successful:

```
bash

# The executable should be created at:
ls -la build/X86/gem5.opt
```

Figure 4: Verify the Build

## 1.5 Test with Hello World Program

```
bash␣  
build/X86/gem5.opt configs/learning_gem5/part2/simple.cache.py␣
```

Figure 5: Hello World Test

Result:

```
Global frequency set at 1000000000000 ticks per second  
  0: system.remote_gdb: listening for remote gdb on port 7000  
**** REAL SIMULATION ****  
info: Entering event queue @ 0. Starting simulation...  
Hello world!  
Exiting @ tick [number] because exiting with last active thread context
```

Figure 6: Result

## 2 Part2

### 2.1 Hamming ECC

In this project, we implement a Hamming(71, 64) code, which is capable of detecting and correcting a single-bit error in a 64-bit data word. The class `HammingCode` contains functions for encoding, decoding, and injecting random bit errors for testing purposes.

### 2.1.1 Encoding (**encode** function):

The encoder inserts 64 data bits into a 71-bit codeword, skipping 7 predefined parity bit positions (powers of two: 1, 2, 4, 8, 16, 32, 64). For each parity bit, the function calculates its value as the XOR of all bits in positions covered by that parity (i.e., positions where the parity's bitmask is set). The result is packed into an 8-bit parity byte and returned.

### 2.1.2 Decoding (**decode** function):

To decode, the function reconstructs the full 71-bit codeword from the corrupted data and received parity. It then recalculates the parity bits and compares them to the received parity, generating a binary *syndrome* that identifies the position of a single-bit error (if present). If the error position is valid (i.e., within 1–71), it is corrected by flipping the bit. The corrected 64-bit data is then extracted and returned.

### 2.1.3 Error Injection (**inject\_random\_error** function):

This function simulates a random single-bit error in the 64-bit data word by flipping a randomly selected bit. It is used to validate the error correction capability of the Hamming code.

### 2.1.4 Results

The results can be seen in the Figure 7, Figure 8 and Figure 9. And, we can see the red marks in the Figures. In the implementation, after calling `inject_random_error` function, the original data will be corrupted (flip one bit). And, after calling `decode` function, the data will be corrected by the ecc code.

```
原始資料: 0x058b420000000000  
產生的parity: 0x73  
隨機損壞後資料: 0x0583420000000000  
Corrected bit at position: 58  
修正後資料: 0x058b420000000000
```

Figure 7: Input: 0x058B420000000000

```
原始資料: 0x00000000862aadbfb  
產生的parity: 0x1d  
隨機損壞後資料: 0x00400000862aadbfb  
Corrected bit at position: 61  
修正後資料: 0x00000000862aadbfb
```

Figure 8: Input: 0x00000000862AADBFB

```
原始資料: 0xbdb6400000000000  
產生的parity: 0x75  
隨機損壞後資料: 0xbdb6400800000000  
Corrected bit at position: 42  
修正後資料: 0xbdb6400000000000
```

Figure 9: Input: 0xBDB6400000000000



## 2.2 Reed Solomon ECC

Reed–Solomon (RS) codes are a class of error-correcting codes based on finite fields  $GF(2^m)$ , widely used in digital communication and storage systems due to their strong error correction capability. This section provides an explanation of our implementation of the encoding and decoding process.

### 2.2.1 Finite Field $GF(2^m)$

Reed–Solomon codes operate over the finite field  $GF(2^m)$ , where:

- Each message and codeword symbol is an  $m$ -byte element.
- Operations include field addition (bitwise XOR) and multiplication (modulo a primitive polynomial).
- Multiplication and division are accelerated using logarithm and exponent tables:

$$a \cdot b = \text{expTable}[(\log\text{Table}[a] + \log\text{Table}[b]) \bmod (2^m - 1)].$$

### 2.2.2 Encoding Process

Given a message of  $k$  symbols, we wish to encode it into a codeword of length  $n = k + 2t$ , capable of correcting up to  $t$  symbol errors. The process is as follows:

1. **Generate the generator polynomial:**

$$G(x) = \prod_{i=0}^{2t-1} (x - \alpha^i),$$

where  $\alpha$  is a primitive element of  $GF(2^m)$ .

2. **Multiply the message polynomial by  $x^{2t}$ :** Let  $M(x)$  be the message polynomial.

Shift it to make space for redundancy:

$$M(x) \cdot x^{2t}$$

3. **Compute the remainder  $R(x)$ :** Perform polynomial division to compute:

$$R(x) = (M(x) \cdot x^{2t}) \bmod G(x)$$

4. **Form the codeword  $C(x)$ :** Add the remainder back to the shifted message:

$$C(x) = M(x) \cdot x^{2t} + R(x)$$

The result is a valid RS codeword of length  $n$ .

### 2.2.3 Syndrome Calculation

Given a received polynomial  $C'(x)$ , we compute the syndromes:

$$S_i = C'(\alpha^i) = \sum_{j=0}^{n-1} c'_j \cdot \alpha^{ij}, \quad i = 1, 2, \dots, 2t$$

If all  $S_i = 0$ , then no errors are detected. Otherwise, we proceed to error correction.

### 2.2.4 Error Correction Procedure

The decoding process involves the following main steps:

**1. Finding the Error Locator Polynomial  $\Lambda(x)$**  Using the **Berlekamp–Massey algorithm** [1], we compute the error locator polynomial:

$$\Lambda(x) = 1 + \lambda_1 x + \lambda_2 x^2 + \cdots + \lambda_t x^t$$

The roots of  $\Lambda(x)$  correspond to the inverses of the error locations in the field.

**2. Locating the Errors (Chien Search)** [2]

We find the positions  $j$  such that:

$$\Lambda(\alpha^{-j}) = 0$$

This identifies which symbols in the received word are erroneous.

**3. Computing Error Values (Forney's Algorithm)** [3]

We compute the error evaluator polynomial:

$$\Omega(x) = \Lambda(x) \cdot S(x) \mod x^{2t}$$

Then, the error magnitude at position  $j$  is given by:

$$e_j = \frac{\Omega(\alpha^{-j})}{\Lambda'(\alpha^{-j})}$$

where  $\Lambda'(x)$  is the formal derivative of  $\Lambda(x)$ . These values are then subtracted from the received word to correct it.

### 2.2.5 Corresponding Functions in Code

The actual implementation includes the following functions:

- `createGenerator()`: Constructs the generator polynomial  $G(x)$ .
- `encode()`: Performs encoding using polynomial division.
- `calcSyndromes()`: Computes the syndrome values.
- `findErrorLocator()`: Implements Berlekamp–Massey to find  $\Lambda(x)$ .
- `findErrors()`: Uses Chien search to locate errors.
- `correctErrata()`: Applies Forney’s algorithm to compute and correct error magnitudes.
- `decode()`: Full decoding pipeline combining the above.

### 2.2.6 Encapsulation of RS code

Since Reed-Solomon codes are defined by two parameters,  $n$  and  $k$ , and our project specifically requires the implementation of  $RS(12, 8)$ , we encapsulated the relevant functions into a set of callable interfaces. The implementation details are presented in the following section.

### 2.2.7 Results

The results can be seen in the Figure 10, Figure 11 and Figure 12. And, we can see the red marks in the Figures. In the implementation, after calling `inject_two_byte_error` function, the original data will be corrupted (flip two bytes). And, after calling `decode` function, the data will be corrected by the ecc code.

### 2.2.8 Reference

Due to the complexity of implementing the Reed–Solomon ECC algorithm, we referred to external resources for guidance, including relevant GitHub repositories such as: [4].

```
原始資料: 0x0123456789abcdef  
產生的parity: 0x2106a681  
損壞後資料: 0x0148456789ab1bef  
修正後資料: 0x0123456789abcdef
```

Figure 10: Input: 0x0123456789ABCDEF

```
原始資料: 0xdeadc0de1234abcd  
產生的parity: 0xa7294cef  
損壞後資料: 0xdead7fde1234fbcd  
修正後資料: 0xdeadc0de1234abcd
```

Figure 11: Input: 0xDEADC0DE1234ABCD

```
原始資料: 0x1122334455667788  
產生的parity: 0x610afd1e  
損壞後資料: 0x112233245566778e  
修正後資料: 0x1122334455667788
```

Figure 12: Input: 0x1122334455667788

### 3 Part3 & Part4

To integrate the ECCs into the gem5 simulator, we encapsulated the Hamming code and Reed-Solomon code algorithms into two separate C++ classes, HammingCode and RSCode, each exposing interfaces for external use. Additionally, we implemented functions to inject errors into the input data, simulating bit flips or transmission errors that may occur in real-world scenarios.

## 3.1 HammingCode

### 3.1.1 HammingCode Interface

The `HammingCode` class provides the following interface for encoding and decoding:

- `uint8_t encode(uint64_t data)`

Given a 64-bit data word, this method returns the 7-bit parity required for error detection and correction.

- `uint64_t decode(uint64_t data, uint8_t parity)`

Accepts a data word and its associated parity bits. It returns the corrected 64-bit data, if a single-bit error is detected.

- `uint64_t inject_random_error(uint64_t data)`

Introduces a single random bit-flip in the input data.

### 3.1.2 Example Usage

Below is an example demonstrating how to use the `HammingCode` class:

```
HammingCode hamming;

uint64_t data = 0xBDB6400000000000;

uint8_t parity = hamming.encode(data);

uint64_t corrupted = hamming.inject_random_error(data);

uint64_t recovered = hamming.decode(corrupted, parity);
```

### 3.1.3 Results

The results can be seen in the previous section, such as Figure [7](#), [8](#) and [9](#).

## 3.2 RSCode

### 3.2.1 RSCode Interface

The `RSCode` class encapsulates a Reed-Solomon (RS) ECC module configured with parameters ( $n = 12, k = 8$ ), capable of correcting up to 2 bytes of errors. It provides the following interfaces for parity generation, error correction, and error injection:

- `uint32_t get_parity(uint64_t data)`

This method takes a 64-bit data word and generates a 32-bit parity using the Reed-Solomon algorithm. Internally, the data is split into 8 bytes, encoded into a 12-byte codeword, and the last 4 bytes (parity symbols) are extracted and returned.

- `uint64_t correct(uint64_t data, uint32_t parity)`

Accepts a data word and its corresponding RS parity. It reconstructs the full codeword and attempts to decode and correct up to two erroneous bytes. The method returns the corrected 64-bit data. If decoding fails, it prints an error and returns 0.

- `uint64_t inject_two_byte_errors(uint64_t data)`

Randomly selects two distinct byte positions in the 64-bit data and flips them with random non-zero values. This function is used to simulate 2-byte errors, which is the maximum correctable error range for the configured ( $n = 12, k = 8$ ) RS code.

### 3.2.2 Example Usage

```
RSCode rs;

uint64_t data = 0x1122334455667788;

uint32_t parity = rs.get_parity(data);
```

```
uint64_t corrupted = rs.inject_two_byte_errors(data);  
uint64_t recovered = rs.correct(corrupted, parity);
```

### 3.2.3 Results

The results can be seen in the previous section, such as [10](#), [11](#) and [12](#).

## 4 Bonus

### 4.1 Bose–Chaudhuri–Hocquenghem (BCH)

[\[5\]](#), [\[6\]](#), [\[7\]](#), [\[8\]](#)

In this project’s bonus section, we implement a binary BCH code capable of detecting and correcting multiple bit-errors in a 64-bit data word. Specifically, our implementation supports up to  $t$  random bit-errors (e.g., 3-bit or 5-bit errors) depending on the chosen parameters. The core of the implementation resides in the `BCHCode` class, which encapsulates:

- **Galois field generation** over  $\text{GF}(2^m)$  using a primitive polynomial of degree  $m$ .
- **Generator polynomial construction** for the designed error-correction capability  $t$ .
- **Encoding, decoding, and error-injection** methods for testing.



## 4.2 Encoding (encode function):

The encoding process converts a  $k$ -bit data word into an  $n$ -bit codeword by appending  $(n-k)$  parity bits. Steps:

- **Data-polynomial formation:** Interpret the input data as polynomial of degree  $< k$ .
- **Polynomial long division:** Divide  $x^{n-k} \cdot d(x)$  by the generator polynomial  $g(x)$ , computing the remainder  $r(x)$ .
- **Parity extraction:** The coefficients of  $r(x)$  (degree  $< n-k$ ) form the parity bits.
- **Codeword assembly:** Concatenate the original data bits with the computed parity bits to form the final  $n$ -bit codeword.

## 4.3 Decoding (decode function):

To recover the original data from a (possibly corrupted)  $n$ -bit received word, the decoder performs:

- **Syndrome computation:** Evaluate the received polynomial at the first powers of the primitive element  $\alpha$ , yielding  $S_i = r(\alpha^i)$  for  $1 \leq i \leq 2t$ .
- **Error-locator polynomial:** Run the Berlekamp–Massey algorithm on the syndrome sequence to derive the error-locator polynomial.
- **Chien search:** Find the roots of  $\Lambda(x)$  by evaluating it at  $\alpha^{-j}$  for  $0 \leq j < n$ . Each root indicates a bit-error at the corresponding position.
- **Error correction:** Flip the bits at the located error positions.

- **Data extraction:** Strip off the parity symbols to recover the original data bits.

When the number of errors  $\leq t$ , the above algorithm corrects all errors; otherwise it flags an uncorrectable pattern.

#### 4.4 Error Injection (`inject_random_errors` function):

For testing the robustness of the decoder, the `inject_random_errors` method flips a user-specified number of bits at random positions within the n-bit codeword. Internally, it:

- Seeds a pseudorandom number generator (e.g., `std::mt19937`).
- Selects  $e$  distinct bit-positions uniformly in  $[0, n-1]$ .
- Inverts each chosen bit in the codeword.

This simulates channel corruption, allowing empirical verification that the decoder successfully corrects up to  $t$  errors and flags when  $e > t$ .

#### 4.5 BCH code interface

The public API of the `BCHCode` class includes:

- **Constructor:** `BCHCode(int m, int t, int n = 2m-1)` initializes GF tables, primitive polynomial, and generator polynomial.
- **encode(data\_bits):** Takes a `std::vector<int>` of length  $k$  and returns a `std::vector<int>` of length  $(n-k)$  containing parity bits. Alternatively, for the specialized fixed-parameters version: `uint16_t encode(uint8_t data)` returns the full  $n$ -bit codeword.

- **decode(codeword)**: Accepts a length-n bit-vector (or `uint16_t`), corrects up to `t` bit-errors, and returns the length-k data bits (or the recovered data byte)
- **inject\_random\_errors(codeword, e)**: Flips `e` random bits in the given codeword and returns the corrupted copy.
- Bit-packing helpers: **uint64\_to\_bits(uint64\_t, num\_bits)** and **bits\_to\_uint64(...)** convert between integers and bit-vectors.

This interface cleanly separates data handling from ECC internals, enabling easy integration into memory controllers, file I/O pipelines, or network links for end-to-end error-control testing.

## 4.6 Example Usage

```
BCHCode bch(7, 3, 99);  
  
uint64_t original_data = 0x0123456789ABCDEF;  
  
std::vector<int> data_bits = bch.uint64_to_bits(original_data, bch.get_k());  
std::vector<int> parity = bch.encode(data_bits);  
  
std::vector<int> corrupted = bch.inject_random_errors(codeword, 3);  
  
std::vector<int> decoded = bch.decode(corrupted);  
  
uint64_t decoded_data = bch.bits_to_uint64(decoded);
```

## 4.7 Results

The results can be seen in the Figure 13, Figure 14. In the implementation, after calling `inject_random_errors` function, the original data will be corrupted (flip 3 bits or 5 bits based

on the parameters). And, after calling decode function, the data will be corrected by the ECC code.

```
BCH(99, 78, 7) code initialized
Error correcting capability: 3 errors
BCH Code Parameters:
  Code length (n): 99
  Data length (k): 78
  Min distance (d): 7
  Error correction capability (t): 3
  Galois field: GF(2^7)
Original data: 0x123456789abcdef
Injected error at position: 18
Injected error at position: 12
Injected error at position: 37
Decoded data: 0x123456789abcdef
Correction successful
```

Figure 13: Input: *0x123456789abcdef* with 3-bit errors injection

```
BCH(99, 64, 11) code initialized
Error correcting capability: 5 errors
BCH Code Parameters:
  Code length (n): 99
  Data length (k): 64
  Min distance (d): 11
  Error correction capability (t): 5
  Galois field: GF(2^7)
Original data: 0x123456789abcdef
Injected error at position: 7
Injected error at position: 39
Injected error at position: 38
Injected error at position: 38
Injected error at position: 39
Decoded data: 0x123456789abcdef
Correction successful
```

Figure 14: Input: *0x123456789abcdef* with 5-bit errors injection

## 5 Part5

### 5.1 Integration and Simulation of ECC in Gem5 Memory System

Before modifying the memory system, we added arguments in `MemCtrl.py` to support selection between different ECC algorithms. We also prepared mapping structures for both Hamming and Reed-Solomon parity data, as shown in Figure 15 and Figure 16

```
### modified begin ###
ecc_type = Param.String("none", "ECC type: hamming, rs, or none")
### modified end ###
```

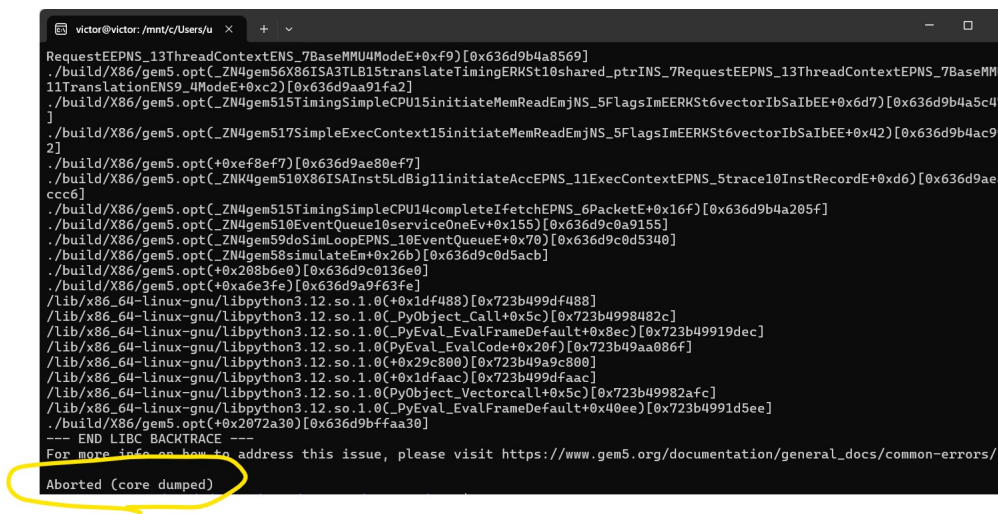
Figure 15: Adding ECC arguments

```
252 class MemCtrl : public qos::MemCtrl
253 {
254     ////////// modified begin //////////
255     std::string eccType;
256     HammingCode hamming;
257     RSCode rs;
258     BCHCode bch;
259
260     std::map<Addr, uint8_t> hammingParityTable;
261     std::map<Addr, uint32_t> rsParityTable;
262     std::map<Addr, std::vector<int>> bchParityTable;
263
264     ////////// modified end //////////
```

Figure 16: Parity Table

To integrate the ECC module into Gem5's memory system, our initial approach was to perform ECC encoding at `addToWriteQueue()`. This placement allowed us to measure the encoding time for each burst, accumulate it in `encTicks`, and propagate the timing

information through to `accessAndRespond()`. However, this implementation caused core dump issues. Specifically, a panic would occur if a random error was injected and decoding was attempted on an address that had never been written—resulting in a missing parity key in the table. Ultimately, this led to returning corrupted data to the CPU and aborting the simulation, as shown in Figure 17.



```

RequestEEPNS_13ThreadContextENS_7BaseMMU4ModeE+0xf9)[0x636d9b4a8569]
./build/X86/gem5.opt(_ZN4gem56X86ISA3TLB15translateTimingERKSt10shared_ptrINS_7RequestEEPNS_13ThreadContextEPNS_7BaseMMU
11TranslationENS9_4ModeE+0xc2)[0x636d9aa91fa2]
./build/X86/gem5.opt(_ZN4gem515TimingSimpleCPU15initiateMemReadEmjNS_5FlagsImEERKSt6vectorIbSaIbEE+0x6d7)[0x636d9b4a5c47
]
./build/X86/gem5.opt(_ZN4gem517SimpleExecContext15initiateMemReadEmjNS_5FlagsImEERKSt6vectorIbSaIbEE+0x42)[0x636d9b4ac99
2]
./build/X86/gem5.opt(+0xf8ef7)[0x636d9ae80ef7]
./build/X86/gem5.opt(_ZN4gem510X86ISAInst5LdBig11initiateAccEPNS_11ExecContextEPNS_5trace10InstRecordE+0xd6)[0x636d9aea
ccc6]
./build/X86/gem5.opt(_ZN4gem515TimingSimpleCPU14completeIfFetchEPNS_6PacketE+0x16f)[0x636d9b4a205f]
./build/X86/gem5.opt(_ZN4gem510EventQueue10serviceOneEv+0x155)[0x636d9c0a9155]
./build/X86/gem5.opt(_ZN4gem59doSimLoopEPNS_10EventQueueE+0x70)[0x636d9c0d5340]
./build/X86/gem5.opt(_ZN4gem58simulateEm+0x26b)[0x636d9c0d5acb]
./build/X86/gem5.opt(+0x208b6e0)[0x636d9c0136e0]
./build/X86/gem5.opt(+0xa6e3fe)[0x636d9a9f63fe]
/lib/x86_64-linux-gnu/libpython3.12.so.1.0(+0x1df488)[0x723b499df488]
/lib/x86_64-linux-gnu/libpython3.12.so.1.0(_PyObject_Call+0x5c)[0x723b4998482c]
/lib/x86_64-linux-gnu/libpython3.12.so.1.0(PyEval_EvalFrameDefault+0x8ec)[0x723b49919dec]
/lib/x86_64-linux-gnu/libpython3.12.so.1.0(PyEval_EvalCode+0x20f)[0x723b49aa086f]
/lib/x86_64-linux-gnu/libpython3.12.so.1.0(+0xc29c800)[0x723b49a0c800]
/lib/x86_64-linux-gnu/libpython3.12.so.1.0(+0x1dfaac)[0x723b499dfaac]
/lib/x86_64-linux-gnu/libpython3.12.so.1.0(PyObject_Vectorcall+0x5c)[0x723b49982afc]
/lib/x86_64-linux-gnu/libpython3.12.so.1.0(PyEval_EvalFrameDefault+0x40ee)[0x723b4991d5ee]
./build/X86/gem5.opt(+0x2072a30)[0x636d9bffa30]
--- END LIBC BACKTRACE ---
For more info on how to address this issue, please visit https://www.gem5.org/documentation/general_docs/common-errors/
Aborted (core dumped)

```

Figure 17: Core Dumped Issue

One possible solution would have been to verify whether each burst had been encoded before decoding, but this approach would have made the implementation overly complex. Therefore, we decided to relocate both the encode and decode steps directly into `accessAndRespond()`.

Within this function, we begin by initializing an ECC timing accumulator, `ECCTicks`, to zero. This variable tracks the total time spent in encoding and decoding operations, measured in picoseconds, allowing us to account for ECC overhead in the memory response timing.

- **Write operations:** As shown in Figure 18, when a packet is a write, we perform

ECC encoding immediately after the memory access. We record the current time, load each 8-byte data chunk into a 64-bit value, and apply the selected ECC algorithm (Hamming or Reed-Solomon) based on `eccType`. The resulting parity is stored in the appropriate table, keyed by burst address. The elapsed encoding time is measured, converted from nanoseconds to picoseconds, and accumulated in `ECCTicks`.

- **Read operations:** As shown in Figure 19, for read packets, we first optionally inject a fault (with 5% probability) using either `hamming.inject_random_error` or `rs.inject_two_byte_errors`, writing the corrupted data back into the packet. We then record the start time for decoding, extract the stored parity, and apply the corresponding ECC decoding or correction method. The corrected data is written back, and the total decoding time is accumulated in `ECCTicks`.

```
// inject error and ECC decode for read
if(pkt->isWrite()){
    auto t0 = std::chrono::high_resolution_clock::now();
    Addr addr = pkt->getAddr();
    // ECC encode for write
    if (eccType == "hamming") {
        uint8_t* ptr = pkt->getPtr<uint8_t>();
        uint64_t data = 0;
        for (int i = 0; i < 8; i++)
            data |= ((uint64_t)ptr[i]) << (8 * i);
        uint8_t parity = hamming.encode(data);
        hammingParityTable[addr] = parity; // store parity
        DPRINTF(MemCtrl, "[Hamming] Hamming ECC encoded: parity = 0x%x for addr 0x%x\n", parity, addr);
    }
    else if (eccType == "rs") {
        uint8_t* ptr = pkt->getPtr<uint8_t>();
        uint64_t data = 0;
        for (int i = 0; i < 8; i++)
            data |= ((uint64_t)ptr[i]) << (8 * i);
        uint32_t parity = rs.get_parity(data);
        rsParityTable[addr] = parity; // store parity
        DPRINTF(MemCtrl, "[RS] RS ECC encoded: parity = 0x%x for addr 0x%x\n", parity, addr);
    }
    auto t1 = std::chrono::high_resolution_clock::now();
    double ns = std::chrono::duration<double, std::nano>(t1-t0).count();
    ECCTicks += static_cast<Tick>(ns * 1000);
}
```

Figure 18: Write Operations

```

else if(pkt->isRead()){
    Addr addr = pkt->getAddr();

    // Inject error with 5% probability
    if (random() % 100 < 5) {
        uint8_t* ptr = pkt->getPtr<uint8_t>();
        uint64_t data = 0;
        for (int i = 0; i < 8; i++)
            data |= ((uint64_t)ptr[i]) << (8 * i);

        if (eccType == "hamming" && hammingParityTable.find(addr) != hammingParityTable.end()) {
            DPRINTF(MemCtrl, "[Hamming] Injected random error for addr 0x%x\n", addr);
            DPRINTF(MemCtrl, "[Hamming] Data before injection: 0x%x\n", data);
            data = hamming.inject_random_error(data);
            for (int i = 0; i < 8; i++)
                ptr[i] = (data >> (8 * i)) & 0xFF;
            DPRINTF(MemCtrl, "[Hamming] Data after injection: 0x%x\n", data);
        } else if (eccType == "rs" && rsParityTable.find(addr) != rsParityTable.end()) {
            DPRINTF(MemCtrl, "[RS] Injected 2-symbol error in data at addr 0x%x\n", addr);
            DPRINTF(MemCtrl, "[RS] Data before injection: 0x%x\n", data);
            data = rs.inject_two_byte_errors(data);
            for (int i = 0; i < 8; i++)
                ptr[i] = (data >> (8 * i)) & 0xFF;
            DPRINTF(MemCtrl, "[RS] Data after injection: 0x%x\n", data);
        }
    }
}

```

Figure 19: Read Operations

In both cases, we schedule the packet's timing response by adding `ECTicks` to the static memory latency, header delay, and payload delay. This ensures our simulation accurately reflects not only the functional behavior of ECC but also the extra time overhead imposed by these operations.

**Simulation of Ticks:** During our development, we noticed that without updating the response time, all ECC algorithms reported identical `simTick` values. Investigating further, we found that the original Gem5 implementation assigned fixed latency values (e.g., 10 ns) for memory operations, as shown in Figure 21. While it would be possible to hardcode a magic number to represent the ECC processing time, we opted for a more accurate approach: measuring the actual time consumption of each ECC algorithm using `std::chrono`, converting the result to picoseconds, and then to Ticks (with 1 Tick  $\approx$  1 ps in Gem5, as shown in Figure 20). This methodology allows us to observe the performance differences between



ECC algorithms in the simulation results.

```
Global frequency set at 1000000000000 ticks per second
```

Figure 20: Gem5 Ticks

```
command_window = Param.Latency("10ns", "Static backend latency")
```

Figure 21: Fixed Latency Value

Table 1: ECC

Index	None	Hamming	Reed-Solomon
simSeconds	0.000371	0.016329	0.013080
simTicks	371,120,000	16,329,404,000	13,079,689,000
finalTick	371,120,000	16,329,404,000	13,079,689,000
simFreq	$10^{12}$	$10^{12}$	$10^{12}$
hostSeconds	0.14	0.21	0.19
hostTickRate	2,583,933,271	77,953,207,273	68,003,311,861
hostMemory	690,368	690,368	690,364
simInsts	6,125	6,125	6,125
simOps	11,036	11,036	11,036
hostInstRate	42,618	29,224	31,829
hostOpRate	76,785	52,653	57,348

As shown in Table 1, our method provides meaningful timing differentiation among the various ECC algorithms, giving insight into their practical overhead in a simulated environment.

## 6 Part6

### 6.1 Simulation results

Simulation results of **None-ECC**, **Hamming-code**, **Reed-Solomon-code**, and **Bose-Chaudhuri-Hocquenghem codes** are shown in Figure 22, Figure 23, Figure 24, and

Figure 25.

```
----- Begin Simulation Statistics -----
simSeconds          0.000128
simTicks            128204000
finalTick           128204000
simFreq             1000000000000
hostSeconds         0.09
hostTickRate        1366197783
hostMemory          695200
simInsts            6181
simOps              11128
hostInstRate        65821
hostOpRate          118498
```

Figure 22: None ECC result

```
----- Begin Simulation Statistics -----
simSeconds          0.012687
simTicks            12687459000
finalTick           12687459000
simFreq             1000000000000
hostSeconds         0.14
hostTickRate        88096342124
hostMemory          695196
simInsts            6181
simOps              11128
hostInstRate        42889
hostOpRate          77212
```

Figure 23: Hamming code result

```
----- Begin Simulation Statistics -----
simSeconds          0.003675
simTicks            3674630000
finalTick           3674630000
simFreq             1000000000000
hostSeconds         0.09
hostTickRate        40028649237
hostMemory          695196
simInsts            6181
simOps              11128
hostInstRate        67254
hostOpRate          121072
```

Figure 24: Reed-Solomon code result

```

----- Begin Simulation Statistics -----
simSeconds          0.228692
simTicks            228692399000
finalTick           228692399000
simFreq             1000000000000
hostSeconds         0.86
hostTickRate        265539141329
hostMemory          719984
simInsts            6181
simOps              11128
hostInstRate        7175
hostOpRate          12917

```

Figure 25: Bose–Chaudhuri–Hocquenghem code result

Table 2: ECC

Index	None	Hamming	Reed-Solomon	BCH
simSeconds	0.000128	0.012687	0.003675	0.228692
simTicks	128,204,000	12,687,459,000	3,674,630,000	228,692,399,000
finalTick	128,204,000	12,687,459,000	3,674,630,000	228,692,399,000
simFreq	$10^{12}$	$10^{12}$	$10^{12}$	$10^{12}$
hostSeconds	0.09	0.14	0.09	0.86
hostTickRate	1,366,197,783	88,096,342,124	40,028,649,237	265,539,141,329
hostMemory	695,200	695,196	695,196	719,984
simInsts	6,181	6,181	6,181	6,181
simOps	11,128	11,128	11,128	11,128
hostInstRate	65,821	42,889	67,254	7,175
hostOpRate	118,498	77,212	121,072	12,917

## 6.2 Original hello world

```

15687  108157000: system.mem_ctrl: Request scheduled immediately
15688  Hello world!
15689  108157000: system.mem_ctrl: QoS Turnarounds selected state READ

```

Figure 26: None error hello world

### 6.3 Hamming code example

```
201558000: system.mem_ctrl: [Hamming] Injected random error for addr 0x94b00
201558000: system.mem_ctrl: [Hamming] Data before injection: 0x0
201558000: system.mem_ctrl: [Hamming] Data after injection: 0x20000000
201558000: system.mem_ctrl: [Hamming] Before correcting, data: 0x20000000
201558000: system.mem_ctrl: [Hamming] the stored parity: 0x0
Corrected bit at position: 36
201558000: system.mem_ctrl: [Hamming] After correcting, data: 0x0
201558000: system.mem_ctrl: Done
```

Figure 27: Hamming code error correction in gem5

```
21456 11881202000: system.mem_ctrl: Request scheduled immediately
21457 Hello world!
21458 11881202000: system.mem_ctrl: QoS Turnarounds selected state READ
```

Figure 28: Hamming hello world

### 6.4 Reed-Solomon example

```
51505000: system.mem_ctrl: [RS] Injected 2-symbol error in data at addr 0x5eb80
51505000: system.mem_ctrl: [RS] Data before injection: 0x415c0e
51505000: system.mem_ctrl: [RS] Data after injection: 0x415edc
51505000: system.mem_ctrl: [RS] Before correcting, data: 0x415edc
51505000: system.mem_ctrl: [RS] the stored parity: 0xad7d07c4
51505000: system.mem_ctrl: [RS] After correcting, data: 0x415c0e
51505000: system.mem_ctrl: Done
```

Figure 29: Reed-Solomon code error correction in gem5

```
18672 3015295000: system.mem_ctrl: Request scheduled immediately
18673 Hello world!
18674 3015295000: system.mem_ctrl: QoS Turnarounds selected state READ
```

Figure 30: RS hello world

## 6.5 Bose–Chaudhuri–Hocquenghem example

```
175916787500: system.mem_ctrl: [BCH] Injected random error for addr 0x814c0
175916787500: system.mem_ctrl: [BCH] Data before injection: 0x7f8
Injected error at position: 28
Injected error at position: 49
Injected error at position: 29
Injected error at position: 53
Injected error at position: 48
175916787500: system.mem_ctrl: [BCH] Data after injection: 0x230000300007f8
175916787500: system.mem_ctrl: [BCH] Before correcting, data: 0x230000300007f8
175916787500: system.mem_ctrl: [BCH] After correcting, data: 0x7f8
175916787500: system.mem_ctrl: Done
```

Figure 31: Bose–Chaudhuri–Hocquenghem code error correction in gem5

```
57315 154536874000: system.mem_ctrl: Request scheduled immediately
57316 Hello world!
57317 154536874000: system.mem_ctrl: QoS Turnarounds selected state READ
```

Figure 32: BCH hello world

## 6.6 Conclusion & Observation

In our project, we implemented several ECC algorithms, and the results are presented in this subsection. We observed that applying ECC increases both simulation time (in seconds and ticks) compared to not using any ECC. Among the implementations, BCH required the most simulation time. From our observations, the Reed–Solomon method is the most time-efficient, while BCH can correct more errors than Reed–Solomon. Furthermore, our implementation showed that BCH consumes more host memory than the other methods. In conclusion, the error-correction capability ranks as  $\text{BCH} > \text{Reed–Solomon} > \text{Hamming}$ , whereas the efficiency ranks as  $\text{Reed–Solomon} > \text{Hamming} > \text{BCH}$ .

## 7 Division of Work

### 7.1 Division of Work

The division of work of our team is shown in the Table 3

Table 3: Division of Work

Members	Contribution
藍少彤	part2-5 implementations, report
陳昭旭	gem5/ECC timer calculation, part5, 8 report
黃偉祥	part6, bonus implementations, report
阮銘福	part1, bonus implementations, report

## 8 Conclusions

### 8.1 What have you learned from this project?

Through this project, we gained valuable insight into the integration of an error-correcting code (ECC) module within the gem5 memory system. We learned the importance of carefully choosing where to place encoding and decoding functions to ensure system stability and accurate timing measurements. Additionally, we understand how gem5 manages timing ticks and realized that dynamically measuring ECC latency at runtime using precise timers such as `std::chrono` leads to more accurate simulation results compared to hardcoded latencies.

### 8.2 What problem(s) have you encountered in this project?

First, some team members encountered difficulties building **gem5** on MacBook M1 processors with only **8GB RAM**. The build process using **Docker** took nearly **5 hours**, and even minor changes required about **2 hours** to rebuild. To speed up the build process, we

found a solution by storing the virtual machine on a USB drive and utilizing public computers with better hardware resources—such as more RAM and CPU cores—running VMware Workstation in the EECS building.

Second, we initially integrated the ECC encoding and decoding algorithms within the `recvTimingReq` function. However, this proved to be the wrong location because, during a read request, the data section of the packet contains random values rather than the actual data to be read. Therefore, we performed ECC encoding in the `addToWriteQueue()` function to measure encoding time and propagate it through the system. However, this caused core dumps when decoding was attempted on addresses without prior writes, due to missing parity keys. This resulted in corrupted data being returned to the CPU and simulation aborts. To fix this, we relocated both encoding and decoding directly to `accessAndRespond()`. Furthermore, despite integrating the ECC algorithms, the simulation ticks did not increase as expected because the ECC tick calculations were initially omitted. We resolved this by implementing the run-time latency measurement using `std::chrono` and aligning our tick calculations with `gem5`'s timing mechanisms.

### 8.3 Suggestions and feedback

Provide an environment (or user guide) that allows students with only ARM-based CPUs to focus on the project itself without being hindered by environmental issues. For example, a dedicated workstation with pre-built `gem5` installations available for each team.

## References

- [1] J. Massey. “Shift-register synthesis and BCH decoding”. In: *IEEE Transactions on Information Theory* 15.1 (1969), pp. 122–127. DOI: [10.1109/TIT.1969.1054260](https://doi.org/10.1109/TIT.1969.1054260).
- [2] R. Chien. “Cyclic decoding procedures for Bose- Chaudhuri-Hocquenghem codes”. In: *IEEE Transactions on Information Theory* 10.4 (1964), pp. 357–363. DOI: [10.1109/TIT.1964.1053699](https://doi.org/10.1109/TIT.1964.1053699).
- [3] G. Forney. “On decoding BCH codes”. In: *IEEE Transactions on Information Theory* 11.4 (1965), pp. 549–557. DOI: [10.1109/TIT.1965.1053825](https://doi.org/10.1109/TIT.1965.1053825).
- [4] FluffyJay1. *ReedSolomon*. <https://github.com/FluffyJay1/ReedSolomon/tree/master>. Accessed: 2025-06-14. 2020.
- [5] R.C. Bose and D.K. Ray-Chaudhuri. “On a class of error correcting binary group codes”. In: *Information and Control* 3.1 (1960), pp. 68–79. ISSN: 0019-9958. DOI: [https://doi.org/10.1016/S0019-9958\(60\)90287-4](https://doi.org/10.1016/S0019-9958(60)90287-4). URL: <https://www.sciencedirect.com/science/article/pii/S0019995860902874>.
- [6] A. Hocquenghem. “Codes correcteurs d’ erreurs”. In: *Chiffres (Paris)* 2 (Sept. 1959), pp. 147–156.
- [7] R.H. Morelos-Zaragoza. *The Art of Error Correcting Coding*. Wiley, 2006. ISBN: 9780470015582. URL: <https://books.google.com.tw/books?id=yIgZAQAIAAJ>.
- [8] Robert Morelos-Zaragoza. *The Error Correcting Codes (ECC) Page*. URL: <https://www.eccpage.com/> (visited on 06/14/2025).