CSE 5095 & ECE 4451 & ECE 5451 – Spring 2017

Assignment 3 (developed by Kamran)

Implementing Memory Integrity Verification in SimpleSim Simulator

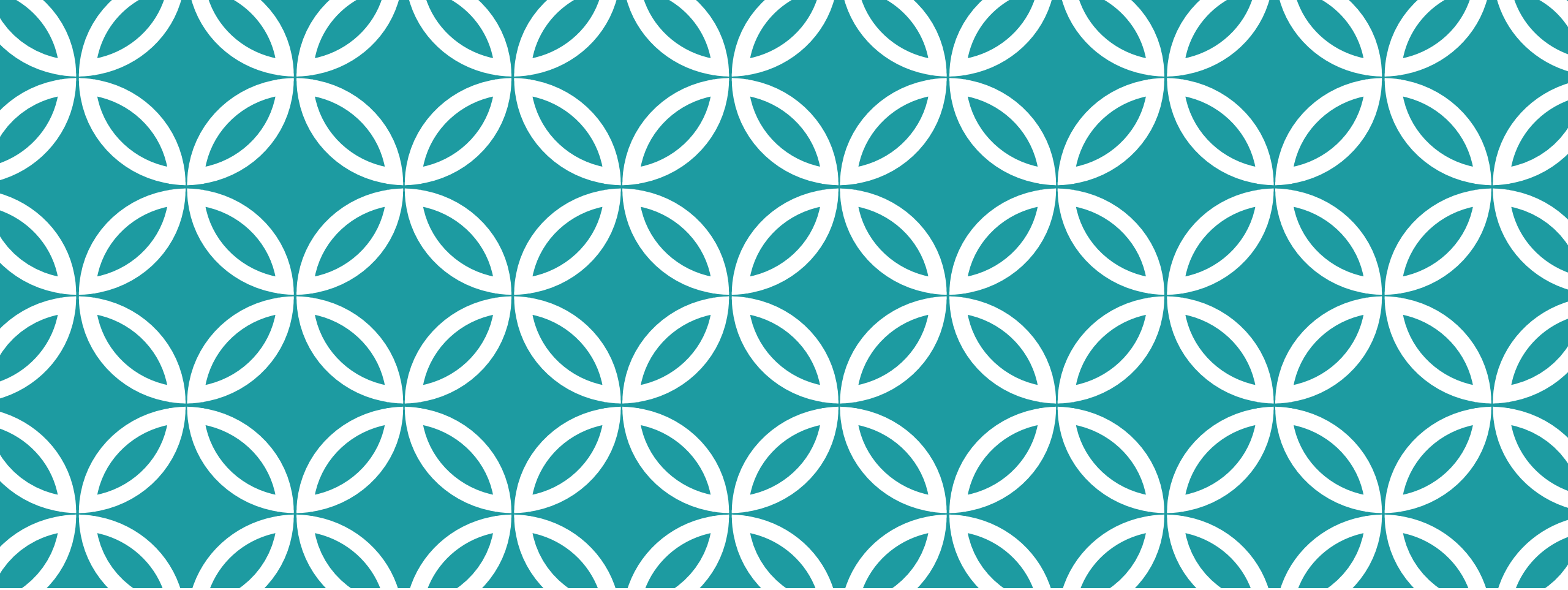
Marten van Dijk

Syed Kamran Haider, Chenglu Jin, Phuong Ha Nguyen

Department of Electrical & Computer Engineering

University of Connecticut

UCONN



Getting Started

SimpleSim Simulator Setup

- It is assumed that:
  - You have Virtual Box installed on your system.
  - You have already setup the "ECE5451-VM" provided in previous assignment.
  - You are able to run test application by running "**`make counters_bench_test`**"
  - You are familiar with SimpleSim directory structure explained in previous assignment.
- Please revisit Assignment 2 slides if you have problems with above mentioned steps.

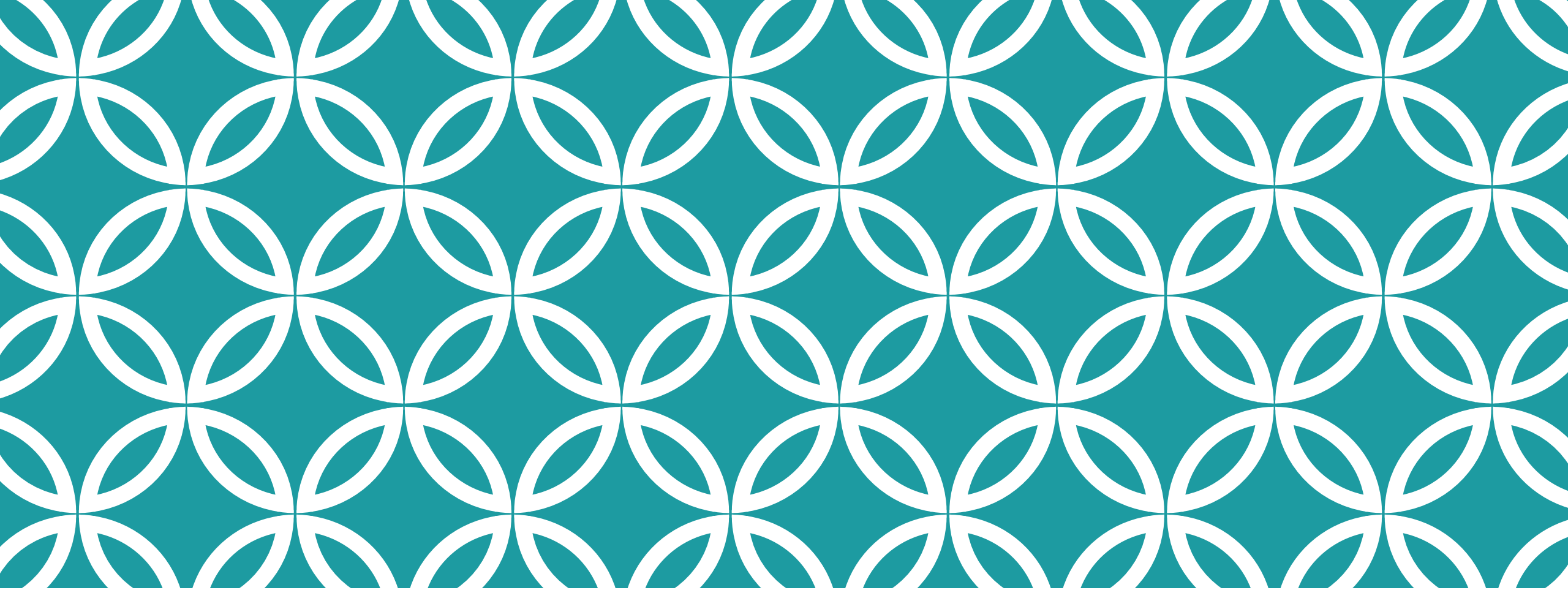
Getting the latest source files

- Login the ECE5451-VM using the following credentials:
  - Username: student
  - Password: student5451
- Enter the SimpleSim_Public directry:
  - `[student@ECE5451-VM ~]$ cd SimpleSim_Public/`
    `[student@ECE5451-VM ~/SimpleSim_Public]$`
- Run “git pull”. This will download all the latest source files in your local machine.
  - `[student@ECE5451-VM ~/SimpleSim_Public]$ git pull`
- Several new files will be added which you can find in `SimpleSim_Public/simulator` directory. Relevant new files are:
  - MerkleTreeDram.cpp, MerkleTreeDram.h, MacTreeDram.cpp, MacTreeDram.h
- Run “make clean” to remove old executables, and then run “make” to compile the simulator
  - `[student@ECE5451-VM ~/SimpleSim_Public]$ make clean`
  - `[student@ECE5451-VM ~/SimpleSim_Public]$ make`

Running a Test Application

- To run a test application, do
  - `make counters_bench_test`
- Currently, Merkle Tree Integrity Verification is turned on.
  - You can change this in Parameters.h → DRAM_TYPE
- For now, you'll notice the following:
  - Cache Hits = 0 because of no cache controller implementation. Full cache controller implementation will be posted later as a solution for Assignment 2.
  - Hash Computations = 0 because of no Integrity Verification implementation.

```
Constructing Merkle Tree:
Total Blocks = 16384
Starting Application...
Done...!
*******************************************************
                SIMULATOR STATISTICS
*******************************************************
                CORE SUMMARY
          Instructions Count 2700007
               Read Accesses 406251
              Write Accesses 206251
                CACHE SUMMARY
                   Cache Hits 0
                 Cache Misses 406251
              Cache Evictions 0
               Cache Hit Rate 0
                 DRAM SUMMARY
                  DRAM Reads 406251
                 DRAM Writes 0
           Hash Computations 0
             Total Time (us) 81939808
*******************************************************
make[1]: Leaving directory `home/student/SimpleSim_Public/tests/benchmarks/counters'
TEST: counters_bench_test PASSED
```



Task 1: Implementing Merkle Tree for Integrity Verification

Merkle Tree based Integrity Verification

- In this task, you need to construct a Merkle Tree to protect 1MB of user's application data.
  - Application visible space is 1MB.
  - A Hash computation function is provided which generates a 16 Byte hash.
  - The cache line size is 64Bytes, therefore 4 hashes can fit in one cache line.
  - Hence, you need to construct a "Quad Tree" where each parent has 4 children.
  - You need to compute the additional space needed for a quad tree for 1MB user level data.
  - Implement memory integrity verification using this quad tree.
- All configuration parameters can be found in Parameters.h file.

Merkle Tree Implementation

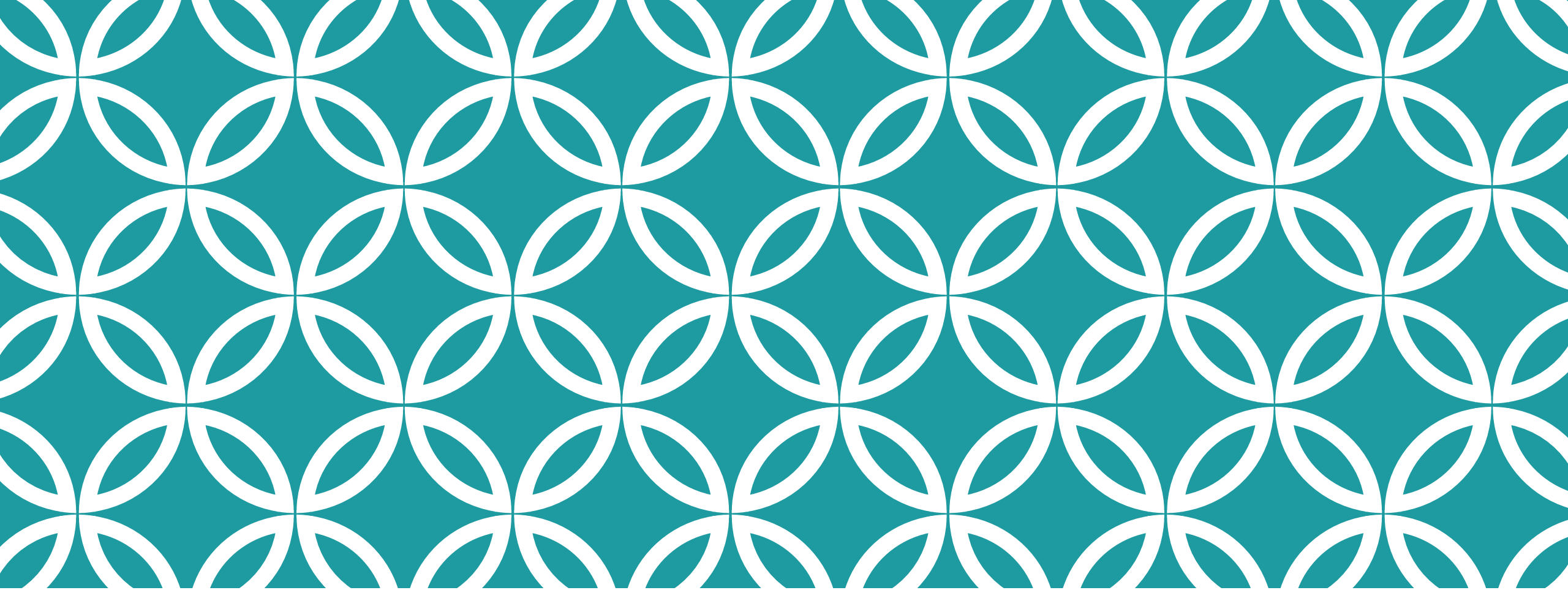
- `simulator/MerkleTreeDram.h` provides an interface to the simulator via `MerkleTreeDram` class.
  - **`bool VerifyHashChain(uint64_t data_cl_num);`**
    - Return '*true*' if integrity of the block 'data_cl_num' is verified.
    - Crash the application by calling 'assert(false);' if an integrity violation detected.
  - **`void UpdateHashChain(uint64_t data_cl_num);`**
    - Update the hash chain for the given block from leaf node up to the root node.
- You need to implement these (and several other) functions in `simulator/MerkleTreeDram.cpp` file.
  - Incomplete function definitions are currently marked with "TODO"
- You are free to add any other functions/variables to achieve the above mentioned desired functionality.

Sample Output

```
Constructing Merkle Tree:
Level[0] Nodes = 16384
Level[1] Nodes = 4096
Level[2] Nodes = 1024
Level[3] Nodes = 256
Level[4] Nodes = 64
Level[5] Nodes = 16
Level[6] Nodes = 4
Total Levels = 7
Total Blocks = 21844
Starting Application...
Done...!
*****************************************************
                SIMULATOR STATISTICS
*****************************************************
                CORE SUMMARY
          Instructions Count 2700007
               Read Accesses 400001
              Write Accesses 200001
               CACHE SUMMARY
                  Cache Hits 393749
                Cache Misses 6252
             Cache Evictions 5229
              Cache Hit Rate 0.98437
                DRAM SUMMARY
                  DRAM Reads 86619
                 DRAM Writes 41832
           Hash Computations 80367
             Total Time (us) 54369501
*****************************************************
make[1]: Leaving directory `/home/syed/SimpleSim/tests/benchmarks/counters'
TEST: counters_bench_test PASSED
```

Checking The Correctness

- To test if your implementation is able to detect ‘malicious modifications’ to the DRAM data contents, you can enable Fault Injector from Parameters.h file.
  - Set FAULT_INJECTION to 1
  - You can set the probability of injecting a fault (in percentage) by setting the parameter FAULT_PROBABILITY
- MerkleTreeDram::InjectFault(uint64_t addr) function injects a fault on-purpose at a random location in the cache line.
  - This function is called by the simulator according to the FAULT_PROBABILITY set by the user.
  - If a fault is injected, a message about it is printed on the screen.
  - After the fault injection, upon subsequent read to this cache line, the system should detect a 'malicious modification' of this data.
- The instructor or TA can also use Fault Injector as a tool to check the correctness of your implementation.



Task 2: Implementing MAC Tree for Integrity Verification

MAC Tree based Integrity Verification

- In this task, you need to construct a MAC Tree to protect 1MB of user's application data.
  - Application visible space is 1MB.
  - A MAC computation function is provided which generates a 8 Byte MAC (64 bits). Each version counter is configured to 7 Bytes (56 bits).
  - The cache line size is 64Bytes, therefore 8 counters and one MAC (also called 'tag') can fit in one cache line.
  - Hence, you need to construct a tree where each parent has 8 children.
  - The format of a cache line with MAC and counters is as follows:
    - <8Byte MAC>|<7Byte counter0>|<7Byte counter1 >|.....|<7Byte counter7>
  - You need to compute the additional space needed for MAC tree for 1MB user level data.
  - Implement memory integrity verification using this MAC tree.
- All configuration parameters can be found in Parameters.h file.

MAC Tree Implementation

- `simulator/MacTreeDram.h` provides an interface to the simulator via `MacTreeDram` class.
  - **`bool VerifyHashChain(uint64_t data_cl_num);`**
    - Return '*true*' if integrity of the block 'data_cl_num' is verified.
    - Crash the application by calling 'assert(false);' if an integrity violation detected.
  - **`void UpdateHashChain(uint64_t data_cl_num);`**
    - Update the MAC chain for the given block from leaf node up to the root node.
- You need to implement these (and several other) functions in `simulator/MacTreeDram.cpp` file.
  - Incomplete function definitions are currently marked with "TODO"
- You are free to add any other functions/variables to achieve the above mentioned desired functionality.

Sample Output

```
Constructing Mac Tree:
Level[0] Nodes = 16384
Level[1] Nodes = 2048
Level[2] Nodes = 256
Level[3] Nodes = 32
Level[4] Nodes = 4
Total Levels = 5
Total Blocks = 20772
Starting Application...
Done...!
*******************************************************
                SIMULATOR STATISTICS
*******************************************************
                CORE SUMMARY
           Instructions Count 2700007
                Read Accesses 400001
               Write Accesses 200001
                CACHE SUMMARY
                   Cache Hits 393749
                 Cache Misses 6252
              Cache Evictions 5229
               Cache Hit Rate 0.98437
                 DRAM SUMMARY
                   DRAM Reads 63657
                  DRAM Writes 31374
            Hash Computations 57405
              Total Time (us) 51284806
*******************************************************
make[1]: Leaving directory `/home/syed/SimpleSim/tests/benchmarks/counters'
```

Checking The Correctness

- To test if your implementation is able to detect ‘malicious modifications’ to the DRAM data contents, you can enable Fault Injector from Parameters.h file.
  - Set FAULT_INJECTION to 1
  - You can set the probability of injecting a fault (in percentage) by setting the parameter FAULT_PROBABILITY
- MacTreeDram::InjectFault(uint64_t addr) function injects a fault on-purpose at a random location in the cache line.
  - This function is called by the simulator according to the FAULT_PROBABILITY set by the user.
  - If a fault is injected, a message about it is printed on the screen.
  - After the fault injection, upon subsequent read to this cache line, the system should detect a 'malicious modification' of this data.
- The instructor or TA can also use Fault Injector as a tool to check the correctness of your implementation.