# CS 401 - Term Project

## MIPS Assembly Implementation of an Encryption Algorithm

Görkem Köse
Deniz Cangı

19 June 2022

Below, you will find the implementation details and analysis for each phase of the term project.

# Phase I:

Small tables method was pretty straightforward, we defined the S-box structures as byte arrays in the data part of MIPS Assembly. Then, we got the defined 16-bit X value, first divided it into 4-bit chunks and each chunk became an input for the respective S-box. After the values of S-box's were computed, we concatenated the 4-bit chunks into a single 16-bit output of S(X).

Large tables method required us to aggregate two S-boxes into a single S-box which takes 8-bit input and gives 8-bit output. Using the python code, located under the folder for Phase I, we aggregated two S-boxes using a 2-level nested loop structure. At the end, we computed two large S-boxes.

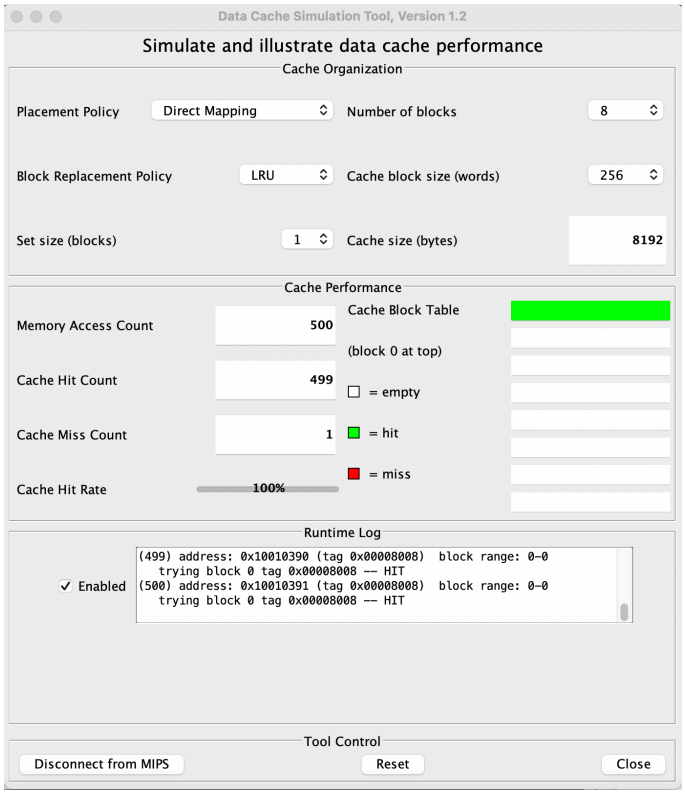
```
sbox01 = []
for item0 in sbox0:
    for item1 in sbox1:
        mergedItem = item0 + item1[2:3]
        sbox01.append(mergedItem)
```

Python code for creating large tables

As a detail to denote, small table S-boxes and large table S-boxes are stored as a byte array instead of a word array, since the largest possible value stored inside a large table is already 8-bit long.

The idea of computing the result of S Function remains the same. Yet, the difference is that the input is now divided into two 8-bit chunks and those 8-bit chunks become an input for the large S-boxes. Instead of 4 cache accesses using small tables, 2 cache accesses are now required using large tables. Below, you can see the overall cache performance for large tables method and small tables method.


Cache performance for small tables.




Cache performance for large tables.

Because of spatial locality, we have a very low cache miss count for both implementations. However, the number of cache accesses is reduced from 700 to 500 using large tables, which is a significant improvement. Therefore, an S Function which opts large tables approach is employed for the rest of the project.

L Function is implemented using 3 helper functions: 6-bit left shifter, 6-bit right shifter and an xor function. In the first phase, the shifter helper functions shifts the input by 6-bit by default. In the next phase of the project, the shifter helper functions are updated such that the amount of shift and required bit masking values are passed as parameter in order to shift the input by any amount.

# Phase II:

Since S Function which uses large tables method and L Function were already implemented, these two functions are called inside of F Function and output of S Function is passed as parameter to L Function in order to compute F(X) = L(S(X)).

For W Function, all required parameters are stored as a single byte array in the data part of the MIPS Assembly, not taken as input in this stage. Then F Function is called iteratively, 5 times; and the return value is saved to register $v0 as it is in all functions.

Initialization stage consist of two parts in the implementation, firstFor and secondFor. firstFor initializes the state vector R to its initial values using initial state vector IV. Then, secondFor starts a sequence to load the parameter values first and call the W function 4 times and update the state vector; where this whole sequence is executed 4 times. At the end of secondFor, the state vector R is initialized.

# Phase III:

In phase III, first the initialization step is performed. Then, for each plaintext, an encryption algorithm is executed and current state vector is updated so that the encryption of a later plaintext depends on the state vector update of the prior plaintext. Each 16-bit ciphertext is stored inside of an array, which is initialized an 32-bit empty space in the data part of MIPS Assembly.

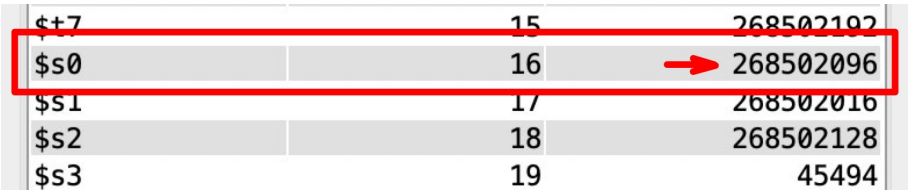
# Phase IV:

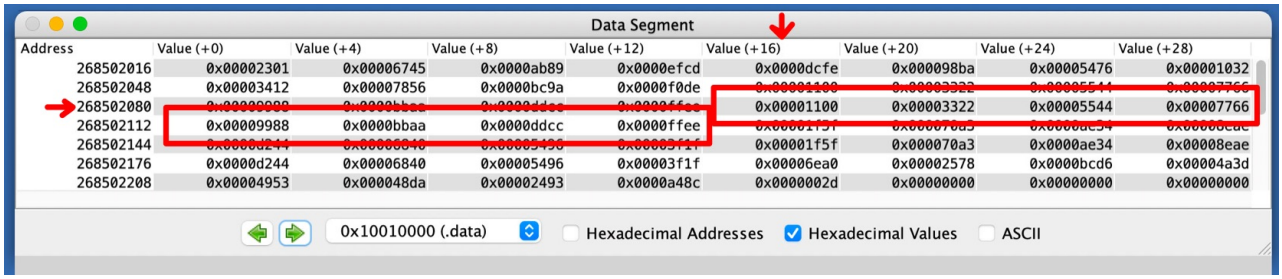
During the last phase, first the inverse of the helper functions are implemented; which is a smooth process since the framework for function calls exists already. Just the content is modified a bit in order to achieve the inverse effect.

Implementing the decryption algorithm is also straightforward since the encryption algorithm is already implemented. There is two different implementations for testing purposes though:

decryption.asm file encrypts and decrypts the given test vectors in the project description document. It does not modify the original array of plaintext P in order to store the result of decryption, but stores the decrypted ciphertext inside a new array called dP, so that the correctness of the implementation becomes more transparent to the grader. Another detail is that the decryption algorithm itself is not implemented as a function on its own, but at the end of the program, register $v0 stores P[8] and the dP array stores the decrypted ciphertext. The address of dP array is stored in register $s0 so that the corresponding address can be checked in memory after the execution completes.

The plaintext test vector is given as: 0x1100, 0x3322, 0x5544, 0x7766, 0x9988, 0xBBAA, 0xDDCC, 0xFFEE. Below, you can see the address of dP array inside the memory and the values stored inside the dP array.



input.asm file encrypts the input of eight 16-bit decimal plaintext taken by the user and decrypts the ciphertext produced by itself. After the initialization, it starts asking for a plaintext 8 times using a loop, and prints the sequence of plaintext taken from the user to the screen, seperated by dashes. Then performs encryption and decryption, stores the result of decryption inside dP array which is stored at a seperate memory location, and prints the decrypted ciphertext to the screen seperated by dashes, again using a loop.

```
Plaintext is: 25359-17920-1234-9083-30000-21209-39102-50000-
Decrypted text is: 25359-17920-1234-9083-30000-21209-39102-50000-
```