

Hashing Algorithms Implementation in Verilog with Pipelining

By: Ahmad Urabi (Student ID. 20200452)

Course: Computer Architecture (2)

Instructor: Dr. Awos Kanaan

# Abstract

A Cryptographic Hashing Algorithm is a complex set of mathematical operations done on a given “message” in order to create a unique message called a “hash”. The resulting hash should in theory be unique and irreversible to it’s original message. One such algorithm is MD4 (Message Digest), created in 1990 by Ronald Rivest and published in RFC 1186 [1].

MD4 is designed to take a message of an arbitrary length, and through three rounds of processing output a 128-bit “message digest”. The algorithm is designed to work quickly on 32-bit machines and can be coded simply with high level programming languages. While intended to be used to create unique fingerprints, and was originally conjectured that it was infeasible to create the same Message Digest with two different messages, cryptographic research managed to create collision attacks that eventually lead MD4 to be labeled as “cryptographically weak” [2]

This project aims to describe how a cryptographic hashing function, in this case a modified version of MD4, can be implemented with hardware, and pipelined such that performance over a set of hash operations is significantly improved over a single cycle processor. Using Verilog HDL, a modified implementation of MD4 managed to successfully create a 128-bit hash using three pipelined stages.

# Introduction

## Algorithm Description

The original MD4 algorithm described by Ronald Rivest, goes through 5 stages of operations in order to produce the final hash. The stages are:

1. Append padding bits
2. Append a 64-bit representation of the length of the message prior to padding
3. Initialize 4 32-bit buffers
4. Process each 16-word block through three rounds
5. Output the result 128-bit message

This project is not a proper implementation of the previously mentioned stages, as the complexity of the first two stages would require complex modules and wouldn’t be beneficial to the main goal of the project, which is pipelining the actual processing of the message (Stages 3 & 4). As such, a preprocessed 512-bit message is used in the processing stages.

This project implements the steps 3,4, & 5, with most of the design revolving around the three rounds of processing in step 4. Each round is composed of 16 operations, for a total of 48 operations. The processor is designed as such that each round is a separate stage, with each stage instantiating 16 modules of the operation function. Each operation module also instantiates a module for an auxiliary function. The auxiliary functions are as such:

F(X,Y,Z) = (X AND Y) OR (NOT(X) AND Z);

G(X,Y,Z) = (X AND Y) OR (X AND Z) OR (Y AND Z)

H(X,Y,Z) = X XOR Y XOR Z

Each stage uses one of the auxiliary functions in order to complete its set of operations.

Four registers (A,B,C,D) are initialized before going into the first stage module. The four registers are initialized the predetermined values, described below (Low order bytes first):

A = 01 23 45 67

B = 89 ab cd ef

C = fe dc ba 98

D = 76 54 32 10

## Design Methodology

Beginning with the auxiliary functions, a module is created for each one, then three operation modules are created, each using a different auxiliary function.

Since circular shift left is not a defined operation in Verilog, a separate module, named CSL is instantiated during every operation, since it’s part of each operation.

Since we are focusing on the pipeline, there is no need to create a module to encase the stages, however if it’s a part of a larger system, you can create a simple module that takes in a 512-bit message input, and output a 128-bit hash.

The operations of the modules have been described behaviorally as to simplify the code, however it’s possible to implement the operations using Verilog primitive modules. The only exclusions would be the circular left shift, and addition operations, which would need complex modules if they were designed structurally.

# Proposed Design

## Modules

The entire design is made up of three stage modules, three function modules, three auxiliary function modules, a circular left shift module, and an output module.

Each stage module is composed of 16 instances of it’s respective operation modules, and takes 4 32-bit inputs (A,B,C,D) and send 4 32-bit outputs (res\_A, res\_B, res\_C, res\_D). There are 16 wires defined inside the module, for the purposes of connecting the output of each operational module to the next one, with the last four outputs of the operational modules going to the output of the stage module.

Each operation module receives four 32-bit inputs (A,B,C,D), one 32-bit input (M), one decimal number (s), and sends a single 32-bit output (res\_out). Inside the module, the corresponding auxiliary module is instantiated, with an internal wire receiving the output from the auxiliary function. The values are then calculated as such:

Stage 1 operation: ( out = ( A + F(B,C,D) + M ) <<< s )

Stage 2 operation: ( out = ( A + G(B,C,D) + M + 32’h5A827999 ) <<< s )

Stage 3 operation: ( out = ( A + H(B,C,D) + M + 32’h6ED9EBA1 ) <<< s )

Where ‘+’ indicates addition, and ‘<<<’ is circular left shift by s bits.

To perform the circular left shift, the value to be circularly shifted is used as the input to a module named CSL, which takes a single 32-bit input, a decimal number s, and outputs a 32-bit result. The module uses two for loops, with the top level one repeating s times, and the lower level one repeating 31 times. The module simply rewires the input to an internal register, and outputs the result of the register. As mentioned earlier, this was described behaviorally in order to reduce the complexity of the code.

The values outputted by the operational function are used in the following operation’s B register, and the values of s and input M are changed according to the list of operations below:

The auxiliary function modules (F,G,H) each take three 32-bit inputs (X,Y,Z), and output a 32-bit result. The operations inside the modules are the ones described in section 2.1, and use the operators (& for bit-wise AND, | for bit-wise OR, ^ for bit-wise XOR).

After the third stage output its set of results, it goes as inputs to an assembly module, which takes eight 32-bit inputs, four of which are from the third stage, and four as the original four values of registers (A,B,C,D). This module performs addition on each input, and it’s corresponding original value and concatenates the four results into a 128-bit value, which is the hash of the original message.

## Single Stage Diagram

# Test Runs

## Test Benches

Each stage was designed to be operated alone so that it may be tested individually. A testbench was created in each of the module files to test the functionality of each stage. Given four 32-bit inputs, the results came out as four 32-bit outputs, after going through sixteen rounds of operations.

The fullcode.v file includes a test bench that instantiated each of the three stages, with each stage taking input from the previous stage, and the first stage taking the four word value initialized as described in section 2.1. The test bench also includes an instance of the assembly module which takes the original values of (A,B,C,D) and the values outputted by stage 3, which outputs the resulting hash.

## Test Results